KEY Statement

Purpose: Sets or displays the soft keys.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: KEY *n, x$*

KEY LIST

KEY ON

KEY OFF

Remarks: *n* is the function key number in the range 1 to 10.

x$ is a string expression which will be assigned to the key. (Remember to enclose string *constants* in quotation marks.)

The KEY statement allows function keys to be designated *soft keys*. That is, you can set each function key to automatically type any sequence of characters. A string of up to 15 characters may be assigned to any one or all of the ten function keys. When the key is pressed, the string will be input to BASIC.

Initially, the soft keys are assigned the following values:

F1	LIST	F2	RUN←
F3	LOAD"	F4	SAVE"
F5	CONT←	F6	,"LPT1:"←
F7	TRON←	F8	TROFF←
F9	KEY	F10	SCREEN 0,0,0 ←

The arrow (←) indicates Enter.

KEY Statement

KEY ON causes the soft key values to be displayed on the 25th line. When the width is 40, five of the ten soft keys are displayed. When the width is 80, all ten are displayed. In either width, only the first six characters of each value are displayed. ON is the default state for the soft key display.

KEY OFF erases the soft key display from the 25th line, making that line available for program use. It does not disable the function keys.

KEY LIST lists all ten soft key values on the screen. All 15 characters of each value are displayed.

KEY *n, x$* assigns the value of *x$* to the function key specified (1 to 10). *x$* may be 1 to 15 characters in length. If it is longer than 15 characters, only the first 15 characters are assigned.

Assigning a null string (string of length zero) to a soft key disables the function key as a soft key.

If the value entered for *n* is not the range 1 to 10, an "Illegal function call" error occurs. The previous key string assignment is retained.

When a soft key is pressed, the INKEY$ function returns one character of the soft key string each time it is called. If the soft key is disabled, INKEY$ returns a two character string. The first character is binary zero, the second is the key scan code, as listed in "Appendix G. ASCII Character Codes."

Note: To avoid complications on the input buffer in Cassette BASIC, you should execute:

```
DEF SEG: POKE 1Ø6,Ø
```

after reassigning any soft keys and after INKEY$ has received the last character you want from a soft key string. This POKE is not required in Disk or Advanced BASIC.

After turning off the soft key display with KEY OFF, you can use LOCATE 25,1 followed by PRINT to display anything you want on the bottom line of the screen. Information on line 25 is not scrolled, as are lines 1 through 24.

See the following section, "KEY(n) Statement," to see how to enable and disable function key trapping in Advanced BASIC.

Example:

```
5Ø KEY ON
```

displays the soft keys on the 25th line.

```
2ØØ KEY OFF
```

erases soft key display. The soft keys are still active, but not displayed.

```
1Ø KEY 1,"FILES"+CHR$(13)
```

assigns the string "FILES"+Enter to soft key 1. This is a way to assign a commonly used command to a function key.

```
2Ø KEY 1,""
```

disables function key 1 as a soft key.

KEY(n) Statement

Purpose: Activates and deactivates trapping of the specified key in a BASIC program. See "ON KEY(n) Statement" in this chapter.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format:

KEY(*n*) ON

KEY(*n*) OFF

KEY(*n*) STOP

Remarks: *n* is a numeric expression in the range 1 to 14, and indicates the key to be trapped:

1-10	function keys F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down

KEY(*n*) ON must be executed to activate trapping of function key or cursor control key activity. After KEY(*n*) ON, if a non-zero line number was specified in the ON KEY(*n*) statement then every time BASIC starts a new statement it will check to see if the specified key was pressed. If so it will perform a GOSUB to the line number specified in the ON KEY(*n*) statement.

If KEY(*n*) is OFF, no trapping takes place and even if the key is pressed, the event is not remembered.

Once a KEY(*n*) STOP statement has been executed, no trapping will take place. However, if you press the specified key your action is remembered so that an immediate trap takes place when KEY(*n*) ON is executed.

KEY (*n*) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

If you use a KEY(*n*) statement in Cassette or Disk BASIC, you will get a "Syntax error." Refer to the previous section, "KEY Statement," for an explanation of the KEY statement without the (*n*).

KILL Command

Purpose: Deletes a file from a diskette. The KILL command in BASIC is similar to the ERASE command in DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: KILL *filespec*

Remarks: *filespec* is a valid file specification as explained under "Naming Files" in Chapter 3. The device name must be a diskette drive. If the device name is omitted, the DOS default drive is used.

KILL can be used for all types of diskette files. The name must include the extension, if one exists. For example, you may save a BASIC program using the command

```
SAVE "TEST"
```

BASIC supplies the extension **.BAS** for the SAVE command, but not for the KILL command. If you want to delete that program file later, you must say KILL "TEST.BAS", not KILL "TEST".

If a KILL statement is given for a file that is currently open, a "File already open" error occurs.

Example:

```
200 KILL "A:DATA1"
```

This example deletes the file named "DATA1" on drive A:.

LEFT$ Function

Purpose: Returns the leftmost *n* characters of *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = LEFT$(*x$*,*n*)

Remarks: *x$* is any string expression.

n is a numeric expression which must be in the range 0 to 255. It specifies the number of characters which are to be in the result.

If *n* is greater than LEN(*x$*), the entire string (*x$*) is returned. If *n*=0, the null string (length zero) is returned.

Also see the MID$ and RIGHT$ functions.

Example:

```
Ok
10 A$ = "BASIC PROGRAM"
20 B$ = LEFT$(A$,5)
30 PRINT B$
RUN
BASIC
Ok
```

In this example, the LEFT$ function is used to extract the first five characters from the string "BASIC PROGRAM".

LEN Function

Purpose: Returns the number of characters in *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = LEN(*x$*)

Remarks: *x$* is any string expression.

Unprintable characters and blanks are included in the count of the number of characters.

Example:

```
10 X$ = "BOCA RATON, FL"
20 PRINT LEN(X$)
RUN
 14
Ok
```

There are 14 characters in the string "BOCA RATON, FL", because the comma and the blank are counted.

LET Statement

Purpose: Assigns the value of an expression to a variable.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: [LET] *variable=expression*

Remarks: *variable* is the name of the variable or array element which is to receive a value. It may be a string or numeric variable or array element.

expression is the expression whose value will be assigned to *variable*. The type of the expression (string or numeric) must match the type of the variable, or a "Type mismatch" error will occur.

The word LET is optional, that is, the equal sign is sufficient when assigning an expression to a variable name.

LET
Statement

Example:

```
11Ø LET DORI=12
12Ø LET E=DORI+2
13Ø LET FDANCE$="HORA"
```

This example assigns the value 12 to the variable DORI. It then assigns the value 14, which is the value of the expression DORI+2, to the variable E. The string "HORA" is assigned to the variable FDANCE$.

The same statements could have also been written:

```
11Ø DORI= 12
12Ø E =DORI+2
13Ø FDANCE$ = "HORA"
```

LINE Statement

Purpose: Draws a line or a box on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Graphics mode only.

Format: LINE [(*x1,y1*)] -(*x2,y2*) [,[*color*] [,B[F]]]

Remarks: *(x1,y1), (x2,y2)*
are coordinates in either absolute or relative form. (See "Specifying Coordinates" under "Graphics Modes" in Chapter 3.)

color is the color number in the range 0 to 3. In medium resolution, *color* selects the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the foreground color, color number 3. In high resolution, a *color* of 0 (zero) indicates black, and the default of 1 (one) indicates white.

The simplest form of LINE is:

```
LINE -(X2,Y2)
```

This will draw a line from the last point referenced to the point (X2,Y2) in the foreground color.

We can include a starting point also:

```
LINE (Ø,Ø)-(319,199) 'diagonal down screen
LINE (Ø,1ØØ)-(319,1ØØ) 'bar across screen
```

We can indicate the color to draw the line in:

```
LINE (1Ø,1Ø)-(2Ø,2Ø),2 'draw in color 2
```

```
1 'draw random lines in random colors
1Ø SCREEN 1,Ø,Ø,Ø: CLS
2Ø LINE -(RND*319,RND*199),RND*4
3Ø GOTO 2Ø
```

```
1 'alternating pattern - line on, line off
1Ø SCREEN 1,Ø,Ø,Ø: CLS
2Ø FOR X=Ø TO 319
3Ø LINE (X,Ø)-(X,199),X AND 1
4Ø NEXT
```

The last argument to LINE is **B** - box, or **BF** - filled box. We can leave out *color* and include the final argument:

```
LINE (Ø,Ø)-(1ØØ,1ØØ),,B 'box in foreground
```

or we may include the color:

```
LINE (Ø,Ø)-(1ØØ,1ØØ),2,BF 'fill box color 2
```

The **B** tells BASIC to draw a rectangle with the points (*x1,y1*) and (*x2,y2*) as opposite corners. This avoids having to give the four LINE commands:

```
LINE (X1,Y1)-(X2,Y1)
LINE (X1,Y1)-(X1,Y2)
LINE (X2,Y1)-(X2,Y2)
LINE (X1,Y2)-(X2,Y2)
```

which perform the equivalent function.

The **BF** means draw the same rectangle as **B**, but also fill in the interior points with the selected color.

When coordinates which are out of range are given to the LINE statement, the coordinate which is out of range is given the closest valid value. In other words, the negative values become zero and Y values greater than 199 become 199. X values greater than 639 become 639. X values greater than 319 in medium resolution wrap to the next horizontal line.

The last point referenced after a LINE statement is point (*x2,y2*). If you use the relative form for the second coordinate, it is relative to the first coordinate. For example,

```
LINE (1ØØ,1ØØ)-STEP (1Ø,-2Ø)
```

will draw a line from (100,100) to (110,80).

Example: This example will draw random filled boxes in random colors.

```
1Ø CLS
2Ø SCREEN 1,Ø: COLOR Ø,Ø
3Ø LINE -(RND*319,RND*199),RND*2+1,BF
4Ø GOTO 3Ø 'boxes will overlap
```

LINE INPUT
Statement

Purpose: Reads an entire line (up to 254 characters) from the keyboard into a string variable, ignoring delimiters.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LINE INPUT[;][*"prompt"*;] *stringvar*

Remarks: *"prompt"* is a string constant that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of the prompt string.

stringvar is the name of the string variable or array element to which the line will be assigned. All input from the end of the prompt to the Enter is assigned to *stringvar*. Trailing blanks are ignored.

In Disk and Advanced BASIC, if LINE INPUT is immediately followed by a semicolon, then pressing Enter to end the input line does not produce a carriage return/line feed sequence on the screen. That is, the cursor remains on the same line as your response.

You can exit LINE INPUT by pressing Ctrl-Break. BASIC returns to command level and displays **Ok**. You may then enter CONT to resume execution at the LINE INPUT.

Example: See example in the next section, "LINE INPUT # Statement."

LINE INPUT # Statement

Purpose: Reads an entire line (up to 254 characters), ignoring delimiters, from a sequential file into a string variable.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LINE INPUT *#filenum, stringvar*

Remarks: *filenum* is the number under which the file was opened.

stringvar is the name of a string variable or array element to which the line will be assigned.

LINE INPUT # reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT # reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved. That is, the line feed/carriage return characters are returned as part of the string.)

LINE INPUT # is especially useful if each line of a file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

LINE INPUT # can also be used for random files. See "Appendix B. BASIC Diskette Input and Output."

LINE INPUT
Statement

Example: The following example uses LINE INPUT to get information from the keyboard, where the information is likely to have commas or other delimiters in it. Then the information is written to a sequential file, and read back out from the file using LINE INPUT #.

```
Ok
10 OPEN "LIST" FOR OUTPUT AS #1
20 LINE INPUT "Address? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "LIST" FOR INPUT AS #1
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN

Address?
```

Suppose you respond with DELRAY BEACH, FL 33445. The program continues:

```
.
.
.
Address? DELRAY BEACH, FL    33445

DELRAY BEACH, FL    33445
Ok
```

LIST Command

Purpose: Lists the program currently in memory on the screen or other specified device.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: LIST [*line1*] [-[*line2*]] [,*filespec*]

Remarks: *line1, line2*
are valid line numbers in the range 0 to 65529. *line1* is the first line to be listed. *line2* is the last line to be listed. A period (.) may be used for either line number to indicate the current line.

filespec is a string expression for the file specification as outlined under "Naming Files" in Chapter 3. If *filespec* is omitted, the specified lines are listed on the screen.

In Cassette BASIC, listings directed to the screen by omitting the device specifier may be stopped at any time by pressing Ctrl-Break. Listings directed to specific devices may not be interrupted, and will list until the range is exhausted. That is, LIST *range* may be interrupted, but LIST *range*, "SCRN:" may not.

In Disk and Advanced BASIC, any listing to either the screen or the printer may be interrupted by pressing Ctrl-Break.

If the line range is omitted, the entire program is listed.

LIST Command

When the dash (-) is used in a line range, three options are available:

- If only *line1* is given, that line and all higher numbered lines are listed.
- If only *line2* is given, all lines from the beginning of the program through *line2* are listed.
- If both line numbers are specified, all lines from *line1* through *line2*, inclusive, are listed.

When you list to a file on cassette or diskette, the specified part of the program is saved in ASCII format. This file may later be used with MERGE.

Example:

```
LIST
```

Lists the entire program on the screen.

```
LIST 35,"SCRN:"
```

Lists line 35 on the screen.

```
LIST 10-20, "LPT1:"
```

Lists lines 10 through 20 on the printer.

```
LIST 100- ,"COM1:1200,N,8"
```

Lists all lines from 100 through the end of the program to the first communications adapter at 1200 bps, no parity, 8 data bits, 1 stop bit.

```
LIST -200,"CAS1:BOB"
```

Lists from the first line through line 200 to a file named "BOB" on cassette.

LLIST Command

Purpose: Lists all or part of the program currently in memory on the printer (LPT1:).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: LLIST [*line1*][- [*line2*]]

Remarks: The line number ranges for LLIST work the same as for LIST.

In Cassette BASIC, LLIST cannot be interrupted by Ctrl-Break. If you want to stop the list, you must turn the printer off.

BASIC always returns to command level after an LLIST is executed.

Example:

```
LLIST
```

Prints a listing of the entire program.

```
LLIST 35
```

Prints line 35.

```
LLIST 10-20
```

Lists lines 10 through 20 on the printer.

```
LLIST 100-
```

Prints all lines from 100 through the end of the program.

```
LLIST -200
```

Prints the first line through line 200.

LOAD Command

Purpose: Loads a program from the specified device into memory, and optionally runs it.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: LOAD *filespec*[,R]

Remarks: *filespec* is a string expression for the file specification. It must conform to the rules outlined under "Naming Files" in Chapter 3, otherwise an error occurs and the load is cancelled.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the specified program. If the **R** option is omitted, BASIC returns to direct mode after the program is loaded.

However, if the **R** option is used with LOAD, the program is run after it is loaded. In this case all open data files are kept open. Thus, LOAD with the **R** option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using data files.

LOAD *filespec*,R is equivalent to RUN *filespec*.

If you are using Cassette BASIC and the device name is omitted, CAS1: is assumed. CAS1: is the only allowable device for LOAD in Cassette BASIC.

If you are using Disk or Advanced BASIC, the DOS default diskette drive is used if the device is omitted. The extension .BAS is added to the filename if no extension is supplied and the filename is eight characters or less.

Notes when using CAS1:

1. If the LOAD statement is entered in direct mode, the file names on the tape will be displayed on the screen followed by a period (.) and a single letter indicating the type of file. This is followed by the message "Skipped." for the files not matching the named file, and "Found." when the named file is found. Types of files and their corresponding letter are:

   .B for BASIC programs in internal format (created with SAVE command)
   .P for protected BASIC programs in internal format (created with SAVE ,P command)
   .A for BASIC programs in ASCII format (created with SAVE ,A command)
   .M for memory image files (created with BSAVE command)
   .D for data files (created by OPEN followed by output statements)

   To see what files are on a cassette tape, rewind the tape and enter some name that is known not to be on the tape. For example, LOAD "CAS1:NOWHERE". All file names will then be displayed.

   If the LOAD command is executed in a BASIC program, the file names skipped and found are not displayed on the screen.

2. Note that Ctrl-Break may be typed at any time during LOAD. Between files or after a time-out period, BASIC will exit the search and return to command level. Previous memory contents remain unchanged.

3. If CAS1: is specified as the device and the filename is omitted, the next program file on the tape is loaded.

Example:

```
LOAD "MENU"
```

Loads the program named MENU, but does not run it.

```
LOAD "INVENT",R
```

Loads and runs the program INVENT.

```
RUN "INVENT"
```

Same as LOAD "INVENT",R.

```
LOAD "B:REPORT.BAS"
```

Loads the file REPORT.BAS from diskette drive B. Note that the **.BAS** did not have to be specified.

```
LOAD "CAS1:"
```

Loads the next program on the tape.

LOC Function

Purpose: Returns the current position in the file.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: *v* = LOC(*filenum*)

Remarks: *filenum* is the file number used when the file was opened.

With random files, LOC returns the record number of the last record read or written to a random file.

With sequential files, LOC returns the number of records read from or written to the file since it was opened. (A record is a 128 byte block of data.) When a file is opened for sequential input, BASIC reads the first sector of the file, so LOC will return a 1 even before any input from the file.

For a communications file, LOC returns the number of characters in the input buffer waiting to be read. The default size for the input buffer is 256 characters, but you can change this with the /C: option on the BASIC command. If there are more than 255 characters in the buffer, LOC returns 255. Since a string is limited to 255 characters, this practical limit alleviates the need for you to test for string size before reading data into it. If fewer than 255 characters remain in the buffer, then LOC returns the actual count.

LOC
Function

Example:
```
2ØØ IF LOC(1)>5Ø THEN STOP
```

This first example stops the program if we've gone past the 50th record in the file.

```
3ØØ PUT #1,LOC(1)
```

The second example could be used to re-write the record that was just read.

LOCATE Statement

Purpose: Positions the cursor on the active screen. Optional parameters turn the blinking cursor on and off and define the size of the blinking cursor.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LOCATE [*row*][,[*col*] [,[*cursor*][,[*start*] [,*stop*]]]]

Remarks:

row is a numeric expression in the range 1 to 25. It indicates the screen line number where you want to place the cursor.

col is a numeric expression in the range 1 to 40 or 1 to 80, depending upon screen width. It indicates the screen column number where you want to place the cursor.

cursor is a value indicating whether the cursor is visible or not. A 0 (zero) indicates off, 1 (one) indicates on.

start is the cursor starting scan line. It must be a numeric expression in the range 0 to 31.

stop is the cursor stop scan line. It also must be numeric expression in the range 0 to 31.

cursor, start and *stop* do not apply to graphics mode.

start and *stop* allow you to make the cursor any size you want. You indicate the starting and ending scan lines. The scan lines are numbered from 0 at the top of the character position. The bottom scan line is 7 if you have the Color/Graphics Monitor Adapter, 13 if you have the IBM Monochrome Display and Parallel Printer Adapter. If *start* is given and *stop* is omitted, *stop* assumes the value of *start*. If *start* is greater than *stop*, you'll get a two-part cursor. The cursor "wraps" from the bottom line back to the top.

After a LOCATE statement, I/O statements to the screen begin placing characters at the specified location.

When a program is running, the cursor is normally off. You can use LOCATE ,,1 to turn it back on.

Normally, BASIC will not print to line 25. However, you can turn off the soft key display using KEY OFF, then use LOCATE 25,1: PRINT... to put things on line 25.

Any parameter may be omitted. Omitted parameters assume the current value.

Any values entered outside of the ranges indicated will result in an "Illegal function call" error. Previous values are retained.

Example:

```
10 LOCATE 1,1
```

Moves the cursor to the home position in the upper left-hand corner of the screen.

```
20 LOCATE ,,1
```

Makes the blinking cursor visible; its position remains unchanged.

```
30 LOCATE ,,,7
```

Position and cursor visibility remain unchanged. Sets the cursor to display at the bottom of the character on the Color/Graphics Monitor Adapter (starting and ending on scan line 7).

```
40 LOCATE 5,1,1,0,7
```

Moves the cursor to line 5, column 1. Makes the cursor visible, covering the entire character cell on the Color/Graphics Monitor Adapter, starting at scan line 0 and ending on scan line 7.

LOF Function

Purpose: Returns the number of bytes allocated to the file (length of the file).

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: *v* = LOF(*filenum*)

Remarks: *filenum* is the file number used when the file was opened.

For diskette files created by BASIC, LOF will return a multiple of 128. For example, if the actual data in the file is 257 bytes, the number 384 will be returned. For diskette files created outside BASIC (for example, by using EDLIN), LOF returns the actual number of bytes allocated to the file.

For communications, LOF returns the amount of free space in the input buffer. That is, *size*-LOC(*filenum*), where *size* is the size of the communications buffer, which defaults to 256 but may be changed with the /C: option on the BASIC command. Use of LOF may be used to detect when the input buffer is getting full. In practicality, LOC is adequate for this purpose, as demonstrated in the example in "Appendix F. Communications."

Example: These statements will get the last record of the file named BIG, assuming BIG was created with a record length of 128 bytes:

```
10 OPEN "BIG" AS #1
20 GET #1,LOF(1)/128
```

LOG Function

Purpose: Returns the natural logarithm of x.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: $v = \text{LOG}(x)$

Remarks: x must be a numeric expression which is greater than zero.

The natural logarithm is the logarithm to the base e.

Example: The first example calculates the logarithm of the expression 45/7:

```
Ok
PRINT LOG(45/7)
 1.86Ø752
Ok
```

The second example calculates the logorithm of e and of e^2 :

```
Ok
E= 2.718282
Ok
? LOG(E)
 1
Ok
? LOG(E*E)
 2
Ok
```

LPOS Function

Purpose: Returns the current position of the print head within the printer buffer for LPT1:.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: v = LPOS(n)

Remarks: n is a numeric expression which is a dummy argument in Cassette BASIC. In Disk and Advanced BASIC, n indicates which printer is being tested, as follows:

0 or 1	LPT1:
2	LPT2:
3	LPT3:

Therefore, we recommend you use 0 or 1 in Cassette BASIC to maintain compatibility with the other versions.

The LPOS function does not necessarily give the physical position of the print head on the printer.

Example: In this example, if the line length is more than 60 characters long we send a carriage return character to the printer so it will skip to the next line.

```
10Ø IF LPOS(Ø)>6Ø THEN LPRINT CHR$(13)
```

LPRINT and LPRINT USING Statements

Purpose: Prints data on the printer (LPT1:).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LPRINT [*list of expressions*] [;]

LPRINT USING *v$; list of expressions* [;]

Remarks: *list of expressions*
is a list of the numeric and/or string expressions that are to be printed. The expressions must be separated by commas or semicolons.

v$ is a string constant or variable which identifies the format to be used for printing. This is explained in detail under "PRINT USING Statement."

These statements function like PRINT and PRINT USING, except output goes to the printer. See "PRINT Statement" and "PRINT USING Statement."

LPRINT assumes an 80-character wide printer. That is, BASIC automatically inserts a carriage return/line feed after printing 80 characters. This will result in two lines being skipped when you print exactly 80 characters, unless you end the statement with a semicolon. You may change the width value with a WIDTH "LPT1:" statement.

Printing is asynchronous with processing. If you do a form feed (LPRINT CHR$(12);) followed by another LPRINT and the printer takes more than 10 seconds to do the form feed, you may get a "Device

Timeout" error on the second LPRINT. To avoid this problem, do the following:

```
1 ON ERROR GOTO 65000
  .
  .
  .
65000 IF ERR = 24 THEN RESUME '24=timeout
```

You might want to test ERL to make sure the timeout was caused by an LPRINT statement.

Example: This is an example of sending special control characters to the IBM 80 CPS Matrix Printer using LPRINT and CHR$. The printer control characters are listed in the *IBM Personal Computer Technical Reference* manual.

```
10 LPRINT CHR$(14);"      Title Line"
20 FOR I=2 TO 4
30 LPRINT "Report line";I
40 NEXT I
50 LPRINT CHR$(15);"Condensed print; 132 char/line"
60 LPRINT CHR$(18);"Return to normal"
70 LPRINT CHR$(27);"E"
80 LPRINT "This is emphasized print"
90 LPRINT CHR$(27);"F"
100 LPRINT "Back to normal again"
```

The output produced by this program looks like this:

```
                Title Line
Report line 2
Report line 3
Report line 4
Condensed print; 132 char/line
Return to normal

This is emphasized print

Back to normal again
```

LSET and RSET Statements

Purpose: Moves data into a random file buffer (in preparation for a PUT (file) statement).

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: LSET *stringvar* = *x$*

RSET *stringvar* = *x$*

Remarks: *stringvar* is the name of a variable that was defined in a FIELD statement.

x$ is a string expression for the information to be placed into the field identified by *stringvar.*

If *x$* requires fewer bytes than were specified for *stringvar* in the FIELD statement, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If *x$* is longer than *stringvar*, characters are dropped from the right.

Numeric values must be converted to strings before they are LSET or RSET. See "MKI$, MKS$, MKD$ Functions" in this chapter.

LSET and RSET Statements

Refer to "Appendix B. BASIC Diskette Input and Output" for a complete explanation of using random files.

Note: LSET or RSET may also be used with a string variable which was not defined in a FIELD statement to left-justify or right-justify a string in a given field. For example, the program lines:

```
11Ø A$=SPACE$(2Ø)
12Ø RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be useful for formatting printed output.

Example: This example converts the numeric value AMT into a string, and left-justifies it in the field A$ in preparation for a PUT (file) statement.

```
15Ø LSET A$=MKS$(AMT)
```

MERGE Command

Purpose: Merges the lines from an ASCII program file into the program currently in memory.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: MERGE *filespec*

Remarks: *filespec* is a string expression for the file specification. It must conform to the rules for naming files as outlined in "Naming Files" in Chapter 3; otherwise an error occurs and the MERGE is cancelled.

The device is searched for the named file. If found, the program lines in the device file are merged with the lines in memory. If any lines in the file being merged have the same line number as lines in the program in memory, the lines from the file will replace the corresponding lines in memory.

After the MERGE command, the merged program resides in memory, and BASIC returns to command level.

In Cassette BASIC, if the device name is omitted, CAS1: is assumed. CAS1: is the only allowable device for MERGE in Cassette BASIC. With Disk and Advanced BASIC, if the device name is omitted, the DOS default drive is assumed.

If CAS1: is specified as the device name and the filename is omitted, the next ASCII program file encountered on the tape is merged.

If the program being merged was not saved in ASCII format (using the A option on the SAVE command), a "Bad file mode" error occurs. The program in memory remains unchanged.

Example:

```
MERGE "A:NUMBRS"
```

This merges the file named "NUMBRS" on drive A: with the program in memory.

MID$
Function and Statement

Purpose: Returns the requested part of a given string. When used as a statement, as in the second format, replaces a portion of one string with another string.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: As a function:

*v$* = MID$(*x$*,*n*[,*m*])

As a statement:

MID$(*v$*,*n*[,*m*]) = *y$*

Remarks: **For the function (v$=MID$...):**

x$ is any string expression.

n is an integer expression in the range 1 to 255.

m is an integer expression in the range 0 to 255.

The function returns a string of length *m* characters from *x$* beginning with the *n*th character. If *m* is omitted or if there are fewer than *m* characters to the right of the *n*th character, all rightmost characters beginning with the *n*th character are returned. If *m* is equal to zero, or if *n* is greater than LEN(*x$*), then MID$ returns a null string.

Also see the LEFT$ and RIGHT$ functions.

MID$
Function and Statement

For the statement (MID$...=y$):

v$ is a string variable or array element that will have its characters replaced.

n is an integer expression in the range 1 to 255.

m is an integer expression in the range 0 to 255.

y$ is a string expression.

The characters in *v$*, beginning at position *n*, are replaced by the characters in *y$*. The optional *m* refers to the number of characters from *y$* that will be used in the replacement. If *m* is omitted, all of *y$* is used.

However, regardless of whether *m* is omitted or included, the length of *v$* does not change. For example, if *v$* is four characters long and *y$* is five characters long, then after the replacement *v$* will contain only the first four characters of *y$*.

> **Note:** If either *n* or *m* is out of range, an "Illegal function call" error will be returned.

Example: The first example uses the MID$ function to select the middle portion of the string B$.

```
Ok
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
RUN
GOOD EVENING
Ok
```

The next example uses the MID$ statement to replace characters in the string A$.

```
Ok
10 A$="MARATHON, GREECE"
20 MID$(A$,11)="FLORIDA"
30 PRINT A$
RUN
MARATHON, FLORID
Ok
```

Note in the second example how the length of A$ was not changed.

MKI$, MKS$, MKD$ Functions

Purpose: Convert numeric type values to string type values.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format:

*v$* = MKI$ (*integer expression*)

*v$* = MKS$ (*single-precision expression*)

*v$* = MKD$ (*double-precision expression*)

Remarks: Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single-precision number to a 4-byte string. MKD$ converts a double-precision number to an 8-byte string.

These functions differ from STR$ in that they do not actually change the bytes of the data, just the way BASIC interprets those bytes.

See also "CVI, CVS, CVD Functions" in this chapter and "Appendix B. BASIC Diskette Input and Output."

Example: This example uses a random file (#1) with fields defined in line 100. The first field, D$, is intended to hold a numeric value, AMT. Line 110 converts AMT to a string value using MKS$ and uses LSET to place what is actually the value of AMT into the random file buffer. Line 120 places a string into the buffer (we don't need to convert a string); then line 130 writes the data from the random file buffer to the file.

```
100 FIELD #1, 4 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
```

MOTOR
Statement

Purpose: Turns the cassette player on and off from a program.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: MOTOR [*state*]

Remarks: *state* is a numeric expression indicating on or off.

If *state* is non zero, the cassette motor is turned on. If *state* is zero, the cassette motor is turned off.

If *state* is omitted, the cassette motor state is switched. That is, if the motor is off, it is turned on and vice-versa.

Example: The following sequence of statements turns the cassette motor on, then off, then back on again.

```
10 MOTOR 1
20 MOTOR 0
30 MOTOR
```

NAME Command

Purpose: Changes the name of a diskette file. The NAME command in BASIC is similar to the RENAME command in DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: NAME *filespec* AS *filename*

Remarks: *filespec* is a file specification as outlined under "Naming Files" in Chapter 3.

filename will be the new filename. It must be a valid filename as outlined in the same section.

The file specified by *filespec* must exist and *filename* must not exist on the diskette, otherwise an error will result. If the device name is omitted, the DOS default drive is assumed. Note that the file extension does not default to .BAS.

After a NAME command, the file exists on the same diskette, in the same area of diskette space, with the new name.

Example:

```
NAME "A:ACCTS.BAS" AS "LEDGER.BAS"
```

In this example, the file that was formerly named ACCTS.BAS on the diskette in drive A will now be named LEDGER.BAS.

NEW Command

Purpose: Deletes the program currently in memory and clears all variables.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: NEW

Remarks: NEW is usually used to free memory before entering a new program. BASIC always returns to command level after NEW is executed. NEW causes all files to be closed and turns trace off if it was on (see "TRON and TROFF Commands," later in this chapter).

Example:

```
Ok
NEW
Ok
```

The program that had been in memory is now deleted.

OCT$ Function

Purpose: Returns a string which represents the octal value of the decimal argument.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = OCT$(*n*)

Remarks: *n* is a numeric expression in the range -32768 to 65535.

If *n* is negative, the two's complement form is used. That is, OCT$(-*n*) is the same as OCT$(65536-*n*).

See the HEX$ function for hexadecimal conversion.

Example:

```
Ok
PRINT OCT$(24)
30
Ok
```

This example shows that 24 in decimal is 30 in octal.

ON COM(n) Statement

Purpose: Sets up a line number for BASIC to trap to when there is information coming into the communications buffer.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: ON COM(*n*) GOSUB *line*

Remarks:

n is the number of the communications adapter (1 or 2).

line is the line number of the beginning of the trap routine. Setting *line* equal to 0 (zero) disables trapping of communications activity for the specified adapter.

A COM(*n*) ON statement must be executed to activate this statement for adapter *n*. After COM(*n*) ON, if a non-zero line number is specified in the ON COM(*n*) statement then every time the program starts a new statement, BASIC checks to see if any characters have come in to the specified communications adapter. If so, BASIC performs a GOSUB to the specified *line.*

If COM(*n*) OFF is executed, no trapping takes place for the adapter. Even if communications activity does take place, the event is not remembered.

If a COM(*n*) STOP statement is executed, no trapping takes place for the adapter. However, any characters being received are remembered so an immediate trap takes place when COM(*n*) ON is executed.

When the trap occurs an automatic COM(*n*) STOP is executed so recursive traps can never take place.

The RETURN from the trap routine automatically does a COM(*n*) ON unless an explicit COM(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

Typically the communications trap routine reads an entire message from the communications line before returning back. It is not recommended that you use the communications trap for single character messages since at high baud rates the overhead of trapping and reading for each individual character may allow the interrupt buffer for communications to overflow.

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

Example:

```
15Ø ON COM(1) GOSUB 5ØØ
16Ø COM(1) ON
 .
 .
 .
5ØØ REM incoming characters
 .
 .
 .
59Ø RETURN 3ØØ
```

This example sets up a trap routine for the first communications adapter at line 500.

ON ERROR
Statement

Purpose: Enables error trapping and specifies the first line of the error handling subroutine.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: ON ERROR GOTO *line*

Remarks: *line* is the line number of the first line of the error trapping routine. If the line number does not exist, an "Undefined line number" error results.

Once error trapping has been enabled, all errors detected (*including direct mode errors*) will cause a jump to the specified error handling subroutine.

To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

> **Note:** If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

You use the RESUME statement to exit from the error trapping routine. Refer to "RESUME Statement" in this chapter.

Example:

```
1Ø ON ERROR GOTO 1ØØ
2Ø LPRINT "This goes to the printer."
3Ø END
1ØØ IF ERR=27 THEN PRINT "Check printer"
    : RESUME
```

This example shows how you might trap a common error — forgetting to put paper in the printer, or forgetting to switch it on.

ON...GOSUB and ON...GOTO Statements

Purpose: Branches to one of several specified line numbers, depending on the value of an expression.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: ON *n* GOTO *line*[,*line*]...

ON *n* GOSUB *line*[,*line*]...

Remarks: *n* is a numeric expression which is rounded to an integer, if necessary. It must be in the range 0 to 255, an "Illegal function call" error occurs.

line is the line number of a line you wish to branch to.

The value of *n* determines which line number in the list will be used for branching. For example, if the value of *n* is 3, the third line number in the list will be the destination of the branch.

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine. That is, you eventually need to have a RETURN statement to bring you back to the line following the ON...GOSUB.

If the value of *n* is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement.

Example: The first example branches to line 150 if L-1 equals 1, to line 300 if L-1 equals 2, to line 320 if L-1 equals 3, and to line 390 if L-1 equals 4. If L-1 is equal to 0 (zero) or is greater than 4, then the program just goes on to the next statement.

```
100 ON L-1 GOTO 150,300,320,390
```

The next example shows how to use an ON...GOSUB statement.

```
1200 ON A GOSUB 1300,1400
 .
 .
 .
1300 REM start of subroutine for A=1
 .
 .
 .
1390 RETURN
```

ON KEY(n) Statement

Purpose: Sets up a line number for BASIC to trap to when the specified function key or cursor control key is pressed.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: ON KEY(*n*) GOSUB *line*

Remarks: *n* is a numeric expression in the range 1 to 14 indicating the key to be trapped, as follows:

1-10	function keys F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down

line is the line number of the beginning of the trapping routine for the specified key. Setting *line* equal to 0 disables trapping of the key.

A KEY(*n*) ON statement must be executed to activate this statement. After KEY(*n*) ON, if a non-zero line number is specified in the ON KEY(*n*) statement then every time the program starts a new statement, BASIC checks to see if the specified key was pressed. If so, BASIC performs a GOSUB to the specified *line*.

If a KEY(*n*) OFF statement is executed, no trapping takes place for the specified key. Even if the key is pressed, the event is not remembered.

If a KEY(*n*) STOP statement is executed, no trapping takes place for the specified key. However, if the key is pressed the event is remembered, so an immediate trap takes place when KEY(*n*) ON is executed.

When the trap occurs an automatic KEY(*n*) STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a KEY(*n*) ON unless an explicit KEY(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

Key trapping may not work when other keys are pressed before the specified key. The key that caused the trap cannot be tested using INPUT$ or INKEY$, so the trap routine for each key must be different if a different function is desired.

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

KEY(*n*) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

ON KEY(n) Statement

Example: The following is an example of a trap routine for function key 5.

```
1ØØ ON KEY(5) GOSUB 2ØØ
11Ø KEY(5) ON
 .
 .
 .
2ØØ REM function key 5 pressed
 .
 .
 .
29Ø RETURN 14Ø
```

ON PEN Statement

Purpose: Sets up a line number for BASIC to transfer control to when the light pen is activated.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: ON PEN GOSUB *line*

Remarks: *line* is the line number of the beginning of the trap routine for the light pen. Using a line number of 0 disables trapping of the light pen.

A PEN ON statement must be executed to activate this statement. After PEN ON, if a non-zero line number is specified in the ON PEN statement, then every time the program starts a new statement BASIC will check to see if the pen was activated. If so, BASIC performs a GOSUB *line*.

If PEN OFF is executed, no trapping takes place. Even if the light pen is activated, the event is not remembered.

If a PEN STOP statement is executed, no trapping takes place, but pen activity is remembered so that an immediate trap takes place when PEN ON is executed.

When the trap occurs, an automatic PEN STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a PEN ON unless an explicit PEN OFF was performed inside the trap routine.

ON PEN Statement

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

PEN(0) is not set when pen activity causes a trap.

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

> **Note:** Do not attempt any cassette I/O while PEN is ON.

Example: This example sets up a trap routine for the light pen.

```
10 ON PEN GOSUB 500
20 PEN ON
 .
 .
 .
500 REM subroutine for pen
 .
 .
 .
650 RETURN 30
```

ON STRIG(n) Statement

Purpose: Sets up a line number for BASIC to trap to when one of the joystick buttons (triggers) is pressed.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: ON STRIG(*n*) GOSUB *line*

Remarks: *n* may be 0, 2, 4, or 6, and indicates the button to be trapped as follows:

- 0 button A1
- 2 button B1
- 4 button A2
- 6 button B2

line is the line number of the trapping routine. If *line* is 0, trapping of the joystick button is disabled.

A STRIG(*n*) ON statement must be executed to activate this statement for button *n*. If STRIG(*n*) ON is executed and a non-zero line number is specified in the ON STRIG(*n*) statement, then every time the program starts a new statement BASIC checks to see if the specified button has been pressed. If so, BASIC performs a GOSUB to the specified *line*.

If STRIG(*n*) OFF is executed, no trapping takes place for button *n*. Even if the button is pressed, the event is not remembered.

If a STRIG(*n*) STOP statement is executed, no trapping takes place for button *n*, but the button

being pressed is remembered so that an immediate trap takes place when STRIG(*n*) ON is executed.

When the trap occurs, an automatic STRIG(*n*) STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a STRIG(*n*) ON unless an explicit STRIG(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

Using STRIG(*n*) ON will activate the interrupt routine that checks the button status for the specified joystick button. Downstrokes that cause trapping will not set functions STRIG(0), STRIG(2), STRIG(4), or STRIG(6).

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

Example: This is an example of a trapping routine for the button on the first joystick.

```
100 ON STRIG(0) GOSUB 2000
110 STRIG(0) ON
 .
 .
 .
2000 REM subroutine for 1st button
 .
 .
 .
2100 RETURN
```

OPEN Statement

Purpose: Allows I/O to a file or device.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: First form:

OPEN *filespec* [FOR *mode*] AS [#] *filenum* [LEN=*recl*]

Alternative form:

OPEN *mode2*, [#] *filenum, filespec* [,*recl*]

Remarks: *mode* in the first form, is one of the following:

OUTPUT	specifies sequential output mode.
INPUT	specifies sequential input mode.
APPEND	specifies sequential output mode where the file is positioned to the end of data on the file when it is opened.

Note that *mode* must be a string constant, *not* enclosed in quotation marks. If *mode* is omitted, random access is assumed.

mode2 in the alternative form, is a string expression whose first character is one of the following:

O	specifies sequential output mode.
I	specifies sequential input mode.
R	specifies random input/output mode.

For both formats:

filenum is an integer expression whose value is between one and the maximum number of files allowed. In Cassette BASIC, the maximum number is 4. In Disk and Advanced BASIC, the default maximum is 3, but this can be changed with the /F: option on the BASIC command.

filespec is a string expression for the file specification as explained under "Naming Files" in Chapter 3.

recl is an integer expression which, if included, sets the record length for random files. It may range from 1 to 32767. *recl* is not valid for sequential files. The default record length is 128 bytes. *recl* may not exceed the value set by the /S: option on the BASIC command.

OPEN allocates a buffer for I/O to the file or device and determines the mode of access that will be used with the buffer.

filenum is the number that is associated with the file for as long as it is open and is used by other I/O statements to refer to the file or device.

An OPEN must be executed before any I/O may be done to a device or file using any of the following statements, or any statement or function requiring a file number:

PRINT #	INPUT #
PRINT # USING	LINE INPUT #
WRITE #	GET
INPUT$	PUT

GET and PUT are valid for random files (or communications files — see next section). A diskette file may be either random or sequential, and a printer may be opened in either random or sequential mode; however, all other devices may be opened only for sequential operations.

BASIC normally adds a line feed after each carriage return (CHR$(13)) sent to a printer. However, if you open a printer (LPT1:, LPT2:, or LPT3:) as a random file with width 255, this line feed is suppressed.

APPEND is valid only for diskette files. The file pointer is initially set to the end of the file and the record number is set to the last record of the file. PRINT # or WRITE # will then extend the file.

> **Note:** At any one time, it is possible to have a particular file open under more than one file number. This allows different modes to be used for different purposes. Or, for program clarity, you may use different file numbers for different modes of access. Each file number has a different buffer, so you should use care if you are writing using one file number and reading using another file number.
>
> However, a file cannot be opened for sequential output or append if the file is already open.

If the device name is omitted when you are using Cassette BASIC, CAS1: is assumed. If you are using Disk or Advanced BASIC, the DOS default drive is assumed.

If CAS1: is specified as the device and the filename is omitted, then the next data file on the cassette is opened.

OPEN
Statement

In Cassette BASIC, a maximum of four files may be open at one time (cassette, printer, keyboard, and screen). Note that only one cassette file may be open at a time. For Disk and Advanced BASIC the default maximum is three files. You can override this value by using the **/F:** option on the BASIC command.

If a file opened for input does not exist, a "File not found" error occurs. If a file which does not exist is opened for output, append, or random access, a file is created.

Any values given outside the ranges indicated will result in an "Illegal function call" error. The file is not opened.

See "Appendix B. BASIC Diskette Input and Output" for a complete explanation of using diskette files. Refer to the next section, "OPEN "COM... Statement," for information on opening communications files.

Example:

```
10 OPEN "DATA" FOR OUTPUT AS #1
  or
10 OPEN "O",#1,"DATA"
```

Either of these statements opens the file named "DATA" for sequential output on the default device (CAS1: for Cassette BASIC, default drive for Disk and Advanced BASIC). Note that opening for output destroys any existing data in the file. If you do not wish to destroy data you should open for APPEND.

```
20 OPEN "B:SSFILE" AS 1 LEN=256
  or
20 OPEN "R",1,"B:SSFILE",256
```

Either of the preceding two statements opens the file named "SSFILE" on the diskette in drive B for random input and output. The record length is 256.

```
25 FILE$ = "A:DATA.ART"
30 OPEN FILE$ FOR APPEND AS 3
```

This example opens the file "DATA.ART" on the diskette in drive A and positions the file pointers so that any output to the file is placed at the end of existing data in the file.

```
Ok
10 OPEN "LPT1:" AS #1' random access
20 PRINT #1,"Printing width 80"
30 PRINT #1,"Now change to width 255"
40 WIDTH #1,255
50 PRINT #1,"This line will be underlined"
60 WIDTH #1,80
70 PRINT #1, STRING$(28,"_")
80 PRINT #1,"Printing width 80 with CR/LF"
RUN
Printing width 80
Now change to width 255
This line will be underlined
Printing width 80 with CR/LF
Ok
```

Line 10 in this example opens the printer in random mode. Because the default width is 80, the lines printed by lines 20 and 30 end with a carriage return/line feed. Line 40 changes the printer width to 255, so the line feed after the carriage return is suppressed. Therefore, the line printed by line 50 ends only with a carriage return and not a line feed. This causes the line printed by line 70 to overprint "This line will be underlined", causing the line to be underlined. Line 60 changes the width back to 80 so the underlines and following lines will end with a line feed.

OPEN "COM... Statement

Purpose: Opens a communications file.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	(**)

Valid only with Asynchronous Communications Adapter.

Format:

OPEN "COM*n*:[*speed*] [,*parity*] [,*data*] [,*stop*]
[,RS] [,CS[*n*]] [,DS[*n*]] [,CD[*n*]] [,LF]"
AS [#]*filenum* [LEN=*number*]

Remarks:

n is 1 or 2, indicating the number of the Asynchronous Communications Adapter.

speed is an integer constant specifying the transmit/receive bit rate in bits per second (bps). Valid speeds are 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, and 9600. The default is 300 bps.

parity is a one-character constant specifying the parity for transmit and receive as follows:

S SPACE: Parity bit always transmitted and received as a space (0 bit).

O ODD: Odd transmit parity, odd receive parity checking.

M MARK: Parity bit always transmitted and received as a mark (1 bit).

E EVEN: Even transmit parity, even receive parity checking.

N NONE: No transmit parity, no receive parity checking.

The default is EVEN (E).

data is an integer constant indicating the number of transmit/receive data bits. Valid values are: 4, 5, 6, 7, or 8. The default is 7.

stop is an integer constant indicating the number of stop bits. Valid values are 1 or 2. The default is two stop bits for 75 and 110 bps, one stop bit for all others. If you use 4 or 5 for *data*, a 2 here will mean 1 1/2 stop bits.

filenum is an integer expression which evaluates to a valid file number. The number is then associated with the file for as long as it is open and is used by other communications I/O statements to refer to the file.

number is the maximum number of bytes which can be read from the communication buffer when using GET or PUT. The default is 128 bytes.

OPEN "COM... allocates a buffer for I/O in the same fashion as OPEN for diskette files. It supports RS232 asynchronous communication with other computers and peripherals.

A communications device may be open to only one file number at a time.

The **RS, CS, DS, CD,** and LF options affect the line signals as follows:

RS	suppresses RTS (Request To Send).
CS[*n*]	controls CTS (Clear To Send).
DS[*n*]	controls DSR (Data Set Ready).
CD[*n*]	controls CD (Carrier Detect).
LF	sends a line feed following each carriage return.

The CD (Carrier Detect) is also known as the RLSD (Received Line Signal Detect).

> **Note:** The *speed, parity, data*, and *stop* parameters are positional, but **RS, CS, DS, CD,** and **LF** are not.

The RTS (Request To Send) line is turned on when you execute an OPEN "COM... statement unless you include the **RS** option.

The *n* argument in the **CS, DS,** and **CD** options specifies the number of milliseconds to wait for the signal before returning a "Device Timeout" error. *n* may range from 0 to 65535. If *n* is omitted or is equal to zero, then the line status is not checked at all.

The defaults are **CS1000, DS1000,** and **CD0**. If **RS** was specified, **CS0** is the default.

That is, normally I/O statements to a communications file will fail if the CTS (Clear To Send) or DSR (Data Set Ready) signals are off. The system waits one second before returning a "Device Timeout." The **CS** and **DS** options allow you to ignore these lines or to specify the amount of time to wait before the timeout.

Normally Carrier Detect (CD or RLSD) is ignored when an OPEN "COM... statement is executed. The CD option allows you to test this line by including the *n* parameter, in the same way as CS and DS. If *n* is omitted or is equal to zero, then Carrier Detect is not checked at all (which is the same as omitting the CD option).

The LF parameter is intended for those using communication files as a means of printing to a serial line printer. When you specify LF, a line feed character (hex 0A) is automatically sent after each carriage return character (hex 0C). (This includes the carriage return sent as a result of the width setting.) Note that INPUT # and LINE INPUT #, when used to read from a communications file that was opened with the LF option, stop when they see a carriage return. The line feed is always ignored.

Any coding errors within the string expression starting with *speed* results in a "Bad file name" error. An indication as to which parameter is in error is not given.

Refer to "Appendix F. Communications" for more information on control of output signals and other technical information on communications support.

If you specify 8 data bits, you must specify parity N. If you specify 4 data bits, you must specify a parity, that is, N parity is invalid. BASIC uses all 8 bits in a byte to store numbers, so if you are transmitting or receiving numeric data (for example, by using PUT), you must specify 8 data bits. (This is not the case if you are sending numeric data *as text*.)

Refer to the previous section for opening devices other than communications devices.

OPEN "COM... Statement

Example:

```
10 OPEN "COM1:" AS 1
```

File 1 is opened for communication with all defaults. The speed is 300 bps with even parity. There will be 7 data bits and one stop bit.

```
10 OPEN "COM1:2400" AS #2
```

File 2 is opened for communication at 2400 bps. Parity, number of data bits, and number of stop bits are defaulted.

```
20 OPEN "COM2:1200,N,8" AS #1
```

File number 1 is opened for asynchronous I/O at 1200 bps, no parity is to be produced or checked, 8-bit bytes will be sent and received, and 1 stop bit will be transmitted.

```
10 OPEN "COM1:9600,N,8,,CS,DS,CD" AS #1
```

Opens COM1: at 9600 bps with no parity and eight data bits. CTS, DSR, and RLSD are not checked.

```
50 OPEN "COM1:1200,,,,CS,DS2000" AS #1
```

Opens COM1: at 1200 bps with the defaults of even parity and seven data bits. RTS is sent, CTS is not checked, and "Device Timeout" is given if DSR is not seen within two seconds. Note that the commas are required to indicate the position of the *parity, start*, and *stop* parameters, even though a value is not specified. This is what is meant by *positional* parameters.

An OPEN statement may be used with an ON ERROR statement to make sure a modem is working properly before sending any data. For example, the following program makes sure we get Carrier Detect (CD or RLSD) from the modem before starting. Line 20 is set to timeout after 10 seconds. TRIES is set to 6 so we give up if Carrier Detect is not seen within one minute. Once communication is established, we re-open the file with a shorter delay until timeout.

```
5 TRIES=6
10 ON ERROR GOTO 100
20 OPEN "COM1:300,N,8,2,CS,DS,CD10000" AS #1
30 ON ERROR GOTO 0
40 CLOSE #1 ' works so can continue
50 GOTO 1000
 .
 .
 .
100 TRIES=TRIES-1
110 IF TRIES=0 THEN ON ERROR GOTO 0 ' give up
120 RESUME
 .
 .
 .
1000 OPEN "COM1:300,N,8,2,CS,DS,CD2000" AS #1
```

The next example shows a typical way to use a communication file to control a serial line printer. The LF parameter in the OPEN statement ensures that lines do not print on top of each other.

```
10 WIDTH "COM1:", 132
20 OPEN "COM1:1200,N,8,,CS10000,DS10000,
      CD10000,LF" AS #1
```

OPTION BASE Statement

Purpose: Declares the minimum value for array subscripts.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: OPTION BASE *n*

Remarks: *n* is 1 or 0.

The default base is 0. If the statement:

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

The OPTION BASE statement must be coded *before* you define or use any arrays.

OUT Statement

Purpose: Sends a byte to a machine output port.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: OUT *n,m*

Remarks: *n* is a numeric expression for the port number, in the range 0-65535.

m is a numeric expression for the data to be transmitted, in the range 0-255.

Refer to the *IBM Personal Computer Technical Reference* manual for a description of valid port numbers (I/O addresses).

OUT is the complementary statement to the INP function. Refer to "INP Function" in this chapter.

One use of OUT is to affect the video output. On some displays attached to the Color/Graphics Monitor Adapter, you may find that the first two or three characters on the line don't show up on the screen. If your display does not have a horizontal adjustment control, you can use the following statements to shift the display:

```
OUT 980,2: OUT 981,43
```

This shifts the display two characters to the right in 40-column width (or 16 points in medium resolution graphics mode, or 32 points in high resolution graphics mode).

OUT
Statement

```
OUT 980,2: OUT 981,85
```

This shifts the display right five characters in 80-column width.

The shift caused by these OUT statements remains in effect until a WIDTH or SCREEN statement is executed. The MODE command from DOS can also be used to shift the display as described here; it has the benefit of remaining in effect until a System Reset.

Example:

```
100 OUT 32,100
```

This sends the value 100 to output port 32.

PAINT Statement

Purpose: Fills in an area on the screen with the selected color.

Versions: Cassette Disk Advanced *** Compiler ***

Graphics mode only.

Format: PAINT (*x,y*) [,*paint* [,*boundary*]]

Remarks: (*x,y*) are the coordinates of a point within the area to be filled in. The coordinates may be given in absolute or relative form (see "Specifying Coordinates" under "Graphics Modes" in Chapter 3). This point will be used as a starting point.

paint is the color to be painted with, in the range 0 to 3. In medium resolution, this color is the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the foreground color, color number 3. In high resolution, *paint* equal to 0 (zero) indicates black, and the default of 1 (one) indicates white.

boundary is the color of the edges of the figure to be filled in, in the range 0 to 3 as described above.

The figure to be filled in is the figure with edges of *boundary* color. The figure is filled in with the color *paint.*

Since there are only two colors in high resolution it doesn't make sense for *paint* to be different from *boundary*. Since *boundary* is defaulted to equal *paint* we don't need the third parameter in high resolution mode.

In high resolution this means "blacking out" an area until black is hit, or "whiting out" an area until white is hit.

In medium resolution we can fill in with color 1 with a border of color 2. Visually this might mean a green ball with a red border.

The starting point of PAINT must be inside the figure to be painted. If the specified point already has the color *boundary* then PAINT will have no effect. If *paint* is omitted the foreground color is used (3 in medium resolution, 1 in high resolution). PAINT can paint any type of figure, but "jagged" edges on a figure will increase the amount of stack space required by PAINT. So if a lot of complex painting is being done you may want to use CLEAR at the beginning of the program to increase the stack space available.

The PAINT statement allows scenes to be displayed with very few statements. This can be a very useful capability.

Example:

```
5 SCREEN 1
10 LINE (0,0)-(100,150),2,B
20 PAINT (50,50),1,2
```

The PAINT statement in line 20 fills in the box drawn in line 10 with color 1.

PEEK Function

Purpose: Returns the byte read from the indicated memory position.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = PEEK(*n*)

Remarks: *n* is an integer in the range 0 to 65535. *n* is the offset from the current segment as defined by the DEF SEG statement, and indicates the address of the memory location to be read. (See "DEF SEG Statement" in this chapter.)

The returned value will be an integer in the range 0 to 255.

PEEK is the complementary function to the POKE statement (see "POKE Statement," later in this chapter).

Example: The following example can be used in a program to test which display adapter is on the system. After line 30 is executed, the variable IBMMONO will have a value of 0 (zero) if the Color/Graphics Monitor Adapter is used, or 1 (one) if the IBM Monochrome Display and Parallel Printer Adapter is used.

```
10 'test display adapter
20 DEF SEG=0
30 IF (PEEK(&410) AND &H30)=&H30
      THEN IBMMONO=1
      ELSE IBMMONO=0
```

PEN
Statement and Function

Purpose: Reads the light pen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

PEN STOP only in Advanced and Compiler.

Format: As a statement:

PEN ON

PEN OFF

PEN STOP

As a function:

v = PEN(*n*)

Remarks: The PEN function, *v*=PEN(*n*), reads the light pen coordinates.

n is a numeric expression in the range 0 to 9, and affects the value returned by the function as follows:

0 A flag indicating if pen was down since last poll. Returns -1 if down, 0 if not.

1 Returns the x coordinate where pen was last activated. Range is 0 to 319 in medium resolution, or 0 to 639 in high resolution.

2 Returns the y coordinate where pen was last activated. Range is 0 to 199.

3 Returns the current pen switch value. -1 if down, 0 if up.

4 Returns the last known valid x coordinate. Range is 0 to 319 in medium resolution, or 0 to 639 in high resolution.

5 Returns the last known valid y coordinate. Range is 0 to 199.

6 Returns the character row position where pen was last activated. Range is 1 to 24.

7 Returns the character column position where pen was last activated. Range is 1 to 40 or 1 to 80 depending on WIDTH.

8 Returns the last known valid character row. Range is 1 to 24.

9 Returns the last known valid character column position. Range is 1 to 40 or 1 to 80 depending on WIDTH.

PEN ON enables the PEN read function. The PEN function is initially off. A PEN ON statement must be executed before any pen read function calls can be made. A call to the PEN function while the PEN function is off results in an "Illegal function call" error.

Conversely, for execution speed improvements, it is a good idea to turn the pen off with a PEN OFF statement when you are not using the light pen.

For Advanced BASIC, executing PEN ON will also allow trapping to take place with the ON PEN statement. After PEN ON, if a nonzero line number was specified in the ON PEN statement, then every time the program starts a new statement BASIC checks to see if the pen was activated. Refer to "ON PEN Statement" in this chapter.

PEN
Statement and Function

PEN OFF disables the PEN read function. For Advanced BASIC, no trapping of the pen takes place and action by the light pen is not remembered even if it does take place.

PEN STOP is only available in Advanced BASIC. It disables trapping of light pen activity, but if activity happens it is remembered so an immediate trap occurs when a PEN ON is executed.

When the pen is down in the border area of the screen, the values returned are inaccurate.

You should not attempt I/O to cassette while PEN is ON.

Example:

```
50 PEN ON
60 FOR I=1 TO 500
70 X=PEN(0): X1=PEN(3)
80 PRINT X, X1
90 NEXT
100 PEN OFF
```

This example prints the pen value since the last poll, and the current value.

PLAY Statement

Purpose: Plays music as specified by *string.*

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: PLAY *string*

Remarks: PLAY implements a concept similar to DRAW by imbedding a "tune definition language" into a character string.

string is a string expression consisting of single character music commands.

The single character commands in PLAY are:

A to G with optional #, +, or –
Plays the indicated note in the current octave. A number sign (#) or plus sign (+) afterwards indicates a sharp, a minus sign (-) indicates a flat. The #, +, or - is not allowed unless it corresponds to a black key on a piano. For example, B# is an invalid note.

O n Octave. Sets the current octave for the following notes. There are 7 octaves, numbered 0 to 6. Each octave goes from C to B. Octave 3 starts with middle C. Octave 4 is the default octave.

N n Plays note n. n may range from 0 to 84. In the 7 possible octaves, there are 84 notes. n=0 means rest. This is an alternative way of selecting notes besides specifying the octave (O n) and the note name (A–G).

PLAY Statement

L n Sets the length of the following notes. The actual note length is 1/n. n may range from 1 to 64. The following table may help explain this:

Length	Equivalent
L1	whole note
L2	half note
L3	one of a triplet of three half notes (1/3 of a 4 beat measure)
L4	quarter note
L5	one of a quintuplet (1/5 of a measure)
L6	one of a quarter note triplet
.	
.	
.	
L64	sixty-fourth note

The length may also follow the note when you want to change the length only for the note. For example, A16 is equivalent to L16A.

P n Pause (rest). n may range from 1 to 64, and figures the length of the pause in the same way as L (length).

(dot or period) After a note, causes the note to be played as a dotted note. That is, its length is multiplied by 3/2. More than one dot may appear after the note, and the length is adjusted accordingly. For example, "A.." will play 9/4 as long as L specifies, "A..." will play 27/8 as long, etc. Dots may also appear after a pause (P) to scale the pause length in the same way.

T n Tempo. Sets the number of quarter notes in a minute. n may range from 32 to 255. The default is 120. Under "SOUND Statement," later in this chapter, is a table listing common tempos and the equivalent beats per minute.

MF Music foreground. Music (created by SOUND or PLAY) runs in foreground. That is, each subsequent note or sound will not start until the previous note or sound is finished. You can press Ctrl-Break to exit PLAY. Music foreground is the default state.

MB Music background. Music (created by SOUND or PLAY) runs in background instead of in foreground. That is, each note or sound is placed in a buffer allowing the BASIC program to continue executing while music plays in the background. Up to 32 notes (or rests) may be played in background at a time.

MN Music normal. Each note plays 7/8 of the time specified by L (length). This is the default setting of MN, ML, and MS.

ML Music legato. Each note plays the full period set by L (length).

MS Music staccato. Each note plays 3/4 of the time specified by L.

X variable;
Executes specified string.

In all of these commands the *n* argument can be a constant like 12 or it can be ***=variable;*** where *variable* is the name of a variable. The semicolon (;) is required when you use a variable in this way, and when you use the X command. Otherwise a semicolon is optional between commands, except a semicolon is not allowed after MF, MB, MN, ML, or MS. Also, any blanks in *string* are ignored.

You can also specify variables in the form VARPTR$(*variable*), instead of =***variable;***. This is useful in programs that will later be compiled. For example:

One Method	Alternative Method
`PLAY "XA$;"`	`PLAY "X"+VARPTR$(A$)`
`PLAY "O=I;"`	`PLAY "O="+VARPTR$(I)`

You can use X to store a "subtune" in one string and call it repetitively with different tempos or octaves from another string.

Example: The following example plays a tune.

```
10 REM little lamb
20 MARY$="GFE-FGGG"
30 PLAY "MB T100 O3 L8;XMARY$;P8 FFF4"
40 PLAY "GB-B-4; XMARY$; GFFGFE-."
```

POINT Function

Purpose: Returns the color of the specified point on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Graphics mode only.

Format: *v* = POINT (*x,y*)

Remarks: (*x,y*) are the coordinates of the point to be used. The coordinates must be in absolute form (see "Specifying Coordinates" under "Graphics Modes" in Chapter 3).

If the point given is out of range the value -1 is returned. In medium resolution valid returns are 0, 1, 2, and 3. In high resolution they are 0 and 1.

Example: The following example inverts the current state of point (I,I).

```
5 SCREEN 2
10 IF POINT(I,I)<>Ø THEN PRESET(I,I)
                    ELSE PSET(I,I)
   or
10 PSET(I,I),1-POINT(I,I)
```

POKE
Statement

Purpose: Writes a byte into a memory location.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: POKE *n,m*

Remarks:

n must be in the range 0 to 65535 and indicates the address of the memory location where the data is to be written. It is an offset from the current segment as defined by the DEF SEG statement (see "DEF SEG Statement" in this chapter).

m m is the data to be written to the specified location. It must be in the range 0 to 255.

The complementary function to POKE is PEEK. (See "PEEK Function" in this chapter.) POKE and PEEK are useful for efficient data storage, loading machine language subroutines, and passing arguments and results to and from machine language subroutines.

Warning:
BASIC does not do any checking on the address. So don't go POKEing around in BASIC's stack, BASIC's variable area, or your BASIC program.

Example:

```
10 DEF SEG: POKE 106,0
```

See "INKEY$ Variable" in this chapter for an explanation of this example.

POS Function

Purpose: Returns the current cursor column position.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = POS(*n*)

Remarks: *n* is a dummy argument.

The current horizontal (column) position of the cursor is returned. The returned value will be in the range 1 to 40 or 1 to 80, depending on the current WIDTH setting. CSRLIN can be used to find the vertical (row) position of the cursor (see "CSRLIN Variable" in this chapter).

Also see the LPOS function.

Example:

```
IF POS(Ø)>6Ø THEN PRINT CHR$(13)
```

This example prints a carriage return (moves the cursor to the beginning of the next line) if the cursor is beyond position 60 on the screen.

PRINT
Statement

Purpose: Displays data on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT [*list of expressions*] [;]

? [*list of expressions*] [;]

Remarks: *list of expressions*
is a list of numeric and/or string expressions, separated by commas, blanks, or semicolons. Any string constants in the list must be enclosed in quotation marks.

If the list of expressions is omitted, a blank line is displayed. If the list of expressions is included, the values of the expressions are displayed on the screen.

> **Note:** The question mark (?) may be used as a shorthand way of entering PRINT only when you are using the BASIC program editor.

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma, semicolon, or SPC or TAB function ends the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions ends without a comma, semicolon, SPC or TAB function, a carriage return is printed at the end of the line; that is, BASIC moves the cursor to the beginning of the next line.

If the length of the value to be printed exceeds the number of character positions remaining on the current line, then the value will be printed at the beginning of the next line. If the value to be printed is longer than the defined WIDTH, BASIC prints as much as it can on the current line and continues printing the rest of the value on the next physical line.

Scrolling occurs as described under "Text Mode" in Chapter 3.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single-precision numbers that can be represented with 7 or fewer digits in fixed point format no less accurately than they can be represented in the floating point format, are output using fixed point or integer format. For example, 10^(-7) is output as .0000001 and 10^(-8) is output as 1E-8.

BASIC automatically inserts a carriage return/line feed after printing *width* characters, where *width* is 40 or 80, as defined by the WIDTH statement. This will cause two lines to be skipped when you print exactly 40 (or 80) characters, unless the PRINT statement ends in a semicolon (;).

LPRINT is used to print information on the printer. See "LPRINT and LPRINT USING Statements" earlier in this chapter.

PRINT
Statement

Example:

```
Ok
1Ø X=5
2Ø PRINT X+5, X-5, X*(-5)
3Ø END
RUN
 1Ø            Ø             -25
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

```
Ok
1Ø INPUT X
2Ø PRINT X "SQUARED IS" X^2 "AND";
3Ø PRINT X "CUBED IS" X^3
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729
Ok
RUN
? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261
Ok
```

Here, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line.

```
Ok
1Ø FOR X = 1 TO 5
2Ø J=J+5
3Ø K=K+1Ø
4Ø ?J;K;
5Ø NEXT X
RUN
 5  1Ø  1Ø  2Ø  15  3Ø  2Ø  4Ø  25  5Ø
Ok
```

Here, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

PRINT USING Statement

Purpose: Prints strings or numbers using a specified format.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT USING *v$; list of expressions* [;]

Remarks: *v$* is a string constant or variable which consists of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

list of expressions
consists of the string expressions or numeric expressions that are to be printed, separated by semicolons or commas.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

!	Specifies that only the first character in the given string is to be printed.
\n spaces\	Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters are printed; with one space, three characters are printed, and so on.

If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.

Example:

```
10 A$="LOOK": B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
50 PRINT USING "\   \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

& Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A$="LOOK": B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces) in the field.

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 Ø.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##    ";1Ø.2,5.3,66.789,.234
1Ø.2Ø    5.3Ø   66.79    Ø.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.

- A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##    ";-68.95,2.4,55.6,-.9
-68.95    +2.40   +55.6Ø     -Ø.9Ø

PRINT USING "##.##-   ";-68.95,22.449,-7.Ø1
68.95-   22.45     7.Ø1-
```

** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks.

The ** also specifies positions for two more digits.

```
PRINT USING "**#.#     ";12.39,-Ø.9,765.1
*12.4   *-Ø.9   765.1
```

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
 $456.78
```

****$** The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces are filled with asterisks and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position.

The comma has no effect if used with the exponential (^^^^) format.

```
PRINT USING "####,.##";1234.5
1,234.5Ø

PRINT USING "####.##,";1234.5
1234.5Ø,
```

^^^^ Four carets may be placed after the digit position characters to specify exponential format. The four carets allow space for E±nn or D±nn to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position is used to the left of the decimal point to print a space or a minus sign.

```
Ok
PRINT USING "##.##^^^^";234.56
 2.35E+Ø2
Ok
PRINT USING ".###^^^^-";-88888
.889E+Ø5-
Ok
PRINT USING "+.##^^^^";123
+.12E+Ø3
Ok
```

_ An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "_ _" in the format string.

If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number. If rounding causes the number to exceed the field, the percent sign is printed in front of the rounded number.

```
Ok
PRINT USING "##.##";111.22
%111.22
Ok
PRINT USING ".##";.999
%1.ØØ
Ok
```

If the number of digits specified exceeds 24, an "Illegal function call" error occurs.

Example: This example shows how you can include string constants in the format string.

```
Ok
PRINT USING "THIS IS EXAMPLE _##"; 1
THIS IS EXAMPLE #1
Ok
```

PRINT # and PRINT # USING Statements

Purpose: Writes data sequentially to a file.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT *#filenum*, [USING *v$*;] *list of exps*

Remarks: *filenum* is the number used when the file was opened for output.

v$ is a string expression comprised of formatting characters as described in the previous section, "PRINT USING Statement."

list of exps is a list of the numeric and/or string expressions that will be written to the file.

PRINT # does not compress data on the file. An image of the data is written to the file just as it would be displayed on the screen with a PRINT statement. For this reason, care should be taken to delimit the data on the file, so that it will be input correctly from the file:

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT #1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields are also written to the file.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the file, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

```
PRINT #1,A$;B$
```

would write CAMERA93604-1 to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT # statement as follows:

```
PRINT #1,A$;",";B$
```

The image written to the file is

```
CAMERA,936Ø4-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to the file surrounded by explicit quotation marks using CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement:

```
PRINT #1,A$;B$
```

writes the following image to the file:

```
CAMERA, AUTOMATIC   936Ø4-1
```

and the statement:

```
INPUT #1,A$,B$
```

inputs the string "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$.

To separate these strings properly on the file, write double quotes to the file image using CHR$(34). The statement:

```
PRINT #1,CHR$(34);A$;CHR$(34);CHR$(34);
  B$;CHR$(34)
```

writes the following image to the file:

```
"CAMERA, AUTOMATIC""   93604-1"
```

and the statement:

```
INPUT #1,A$,B$
```

inputs "CAMERA, AUTOMATIC" to A$ and
" 93604-1" to B$.

The PRINT # statement may also be used with the USING option to control the format of the file. For example:

```
PRINT #1,USING"$$###.##,";J;K;L
```

The easy way to avoid all these problems is to use the WRITE # statement rather than the PRINT # statement. (Refer to "WRITE # Statement," at the end of this chapter.)

Example: For more examples using PRINT # and WRITE #, see "Appendix B. BASIC Diskette Input and Output."

PSET and PRESET Statements

Purpose: Draws a point at the specified position on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Graphics mode only.

Format: PSET (*x,y*) [,*color*]

PRESET (*x,y*) [,*color*]

Remarks: (*x,y*) are the coordinates of the point to be set. They may be in absolute or relative form, as explained in the section "Specifying Coordinates" under "Graphics Modes" in Chapter 3.

color specifies the color to be used, in the range 0 to 3. In medium resolution, *color* selects the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the foreground color, color number 3. In high resolution, a *color* of 0 (zero) indicates black, and the default of 1 (one) indicates white. In high resolution a color value of 2 will be treated as 0, and 3 will be treated as 1.

PRESET is almost identical to PSET. The only difference is that if no *color* parameter is given to PRESET, the background color (0) is selected. If *color* is included, PRESET is identical to PSET. Line 70 in the example below could just as easily be:

```
70 PSET(1,1),Ø
```

If an out of range coordinate is given to PSET or PRESET no action is taken nor is an error given. If *color* is greater than 3, this will result in an "Illegal function call" error.

Example: Lines 20 through 40 of this example draw a diagonal line from the point (0,0) to the point (100,100). Then lines 60 through 80 erase the line by setting each point to a color of 0.

```
10 SCREEN 1
20 FOR I=0 TO 100
30 PSET (I,I)
40 NEXT
50 'erase line
60 FOR I=100 TO 0 STEP -1
70 PRESET(I,I)
80 NEXT
```

PUT
Statement (Files)

Purpose: Writes a record from a random buffer to a random file.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: PUT [#] *filenum* [,*number*]

Remarks: *filenum* is the number under which the file was opened.

number is the record number for the record to be written, in the range 1 to 32767.

If *number* is omitted, the record has the next available record number (after the last PUT).

PRINT #, PRINT # USING, WRITE #, LSET, and RSET may be used to put characters in the random file buffer before a PUT statement. In the case of WRITE #, BASIC pads the buffer with spaces up to the carriage return.

Any attempt to read or write past the end of the buffer causes a "Field overflow" error. Refer to "Appendix B. BASIC Diskette Input and Output."

Because BASIC and DOS block as many records as possible in 512 byte sectors, the PUT statement does not necessarily perform a physical write to the diskette.

PUT can be used for a communications file. In that case *number* is the number of bytes to write to the communications file. This number must be less than or equal to the value set by the LEN option on the OPEN "COM... statement.

Example: See "Appendix B. BASIC Diskette Input and Output."

PUT
Statement (Graphics)

Purpose: Writes colors onto a specified area of the screen.

Versions: Cassette Disk Advanced *** Compiler ***

Graphics mode only.

Format: PUT (*x,y*) ,*array* [,*action*]

Remarks: (*x,y*) are the coordinates of the top left corner of the image to be transferred.

array is the name of a numeric array containing the information to be transferred. See "GET Statement (Graphics)" in this chapter for more information on this array.

action is one of:

PSET
PRESET
XOR
OR
AND

XOR is the default.

PUT is the opposite of GET in the sense that it takes data out of the array and puts it onto the screen. However it also provides the option of interacting with the data already on the screen by the use of the action.

PSET as an action simply stores the data from the array onto the screen, so this is the true opposite of GET.

PRESET is the same as PSET except a negative image is produced. That is, a value of 0 in the array causes the corresponding point to have color number 3, and vice versa; a value of 1 in the array causes the corresponding point to have color number 2, and vice versa.

AND is used when you want to transfer the image only if an image already exists under the transferred image.

OR is used to superimpose the image onto the existing image.

XOR is a special mode which may be used for animation. XOR causes the points on the screen to be inverted where a point exists in the array image. XOR has a unique property that makes it especially useful for animation: when an image is PUT against a complex background *twice*, the background is restored unchanged. This allows you to move an object around without obliterating the background.

In medium resolution mode, AND, XOR, and OR have the following effects on color:

AND

screen \ array value	0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

PUT
Statement (Graphics)

OR

screen \ array value	0	1	2	3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

XOR

screen \ array value	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Animation of an object can be performed as follows:

1. PUT the object on the screen (with XOR).
2. Recalculate the new position of the object.
3. PUT the object on the screen (with XOR) a second time at the old location to remove the old image.
4. Go to step 1, this time putting the object at the new location.

Movement done this way leaves the background unchanged. Flicker can be reduced by minimizing the time between steps 4 and 1, and making sure there is enough time delay between steps 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET action verb. But you should remember to have an image area that will contain the "before" and "after" images of the object. This way the extra area will effectively erase the old image. This method may be somewhat faster than the method using XOR described above, since only one PUT is required to move an object (although you must PUT a larger image).

If the image to be transferred is too large to fit on the screen, an "Illegal function call" error occurs.

RANDOMIZE
Statement

Purpose: Reseeds the random number generator.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: RANDOMIZE [*n*]

Remarks: *n* is an integer expression which will be used as the random number seed.

If *n* is omitted, BASIC suspends program execution and asks for a value by displaying:

```
Random Number Seed (-32768 to 32767)?
```

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the seed with each run.

In Disk and Advanced BASIC, the internal clock can be a useful way to get a random number seed. You can use VAL to change the last two digits of TIME$ to a number, and use that number for *n*.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 4
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)?
```

Suppose you respond with 3. The program continues:

```
Random Number Seed (-32768 to 32767)? 3
 .7655695   .3558607   .3742327   .1388798
Ok
RUN
Random Number Seed (-32768 to 32767)?
```

Suppose this time you respond with 4. The program continues:

```
Random Number Seed (-32768 to 32767)? 4
 .1719568   .5273236   .6879686   .713297
Ok
RUN
Random Number Seed (-32768 to 32767)?
```

If you try 3 again, you'll get the same sequence as the first run:

```
Random Number Seed (-32768 to 32767)? 3
 .7655695   .3558607   .3742327   .1388798
Ok
```

READ Statement

Purpose: Reads values from a DATA statement and assigns them to variables (see "DATA Statement" in this chapter).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: READ *variable* [, *variable*]...

Remarks: *variable* is a numeric or string variable or array element which is to receive the value read from the DATA table.

A READ statement must always be used in conjunction with a DATA statement. READ statements assign DATA statement values to the variables in the READ statement on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in the list of variables exceeds the number of elements in the DATA statement(s), an "Out of data" error occurs. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread data from any line in the list of DATA statements, use the RESTORE statement (see "RESTORE Statement" in this chapter).

Example:

```
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
```

This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) is 3.08, and so on.

```
Ok
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
RUN
CITY          STATE              ZIP
DENVER,       COLORADO           80211
Ok
```

This program reads string and numeric data from the DATA statement in line 30. Note that you don't need quotation marks around COLORADO, because it doesn't have commas, semicolons, or significant leading or trailing blanks. However, you do need the quotation marks around "DENVER," because of the comma.

REM Statement

Purpose: Inserts explanatory remarks in a program.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: REM *remark*

Remarks: *remark* may be any sequence of characters.

REM statements are not executed but are output exactly as entered when the program is listed. However, they do slow up execution time somewhat, and take up space in memory.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution continues with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM. If you put a remark on a line with other BASIC statements, the remark must be the *last* statement on the line.

Example:

```
100 REM calculate average velocity
110 SUM=0: REM initialize SUM
120 FOR I=1 TO 20
130 SUM=SUM + V(I)
 .
 .
 .
```

Line 110 might also be written:

```
110 SUM=0 ' initialize SUM
```

RENUM Command

Purpose: Renumbers program lines.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: RENUM [*newnum*] [,[*oldnum*] [,*increment*]]

Remarks: *newnum* is the first line number to be used in the new sequence. The default is 10.

oldnum is the line in the current program where renumbering is to begin. The default is the first line of the program.

increment is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ELSE, ON...GOTO, ON...GOSUB, RESTORE, RESUME, and ERL test statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

> **Note:** RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

RENUM Command

Example: RENUM

Renumbers the entire program. The first new line number is 10. Lines increment by 10.

```
RENUM 300,,50
```

Renumbers the entire program. The first new line number is 300. Lines increment by 50.

```
RENUM 1000,900,20
```

Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

RESET Command

Purpose: Closes all diskette files and clears the system buffer.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: RESET

Remarks: If all open files are on diskette, then RESET is the same as CLOSE with no file numbers after it.

RESTORE Statement

Purpose: Allows DATA statements to be reread from a specified line.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: RESTORE [*line*]

Remarks: *line* is the line number of a DATA statement in the program.

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If *line* is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:

```
Ok
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
50 PRINT A;B;C;D;E;F
RUN
 57  68  79  57  68  79
Ok
```

The RESTORE statement in line 20 resets the DATA pointer to the beginning, so that the values that are read in line 30 are 57, 68, and 79.

RESUME Statement

Purpose: Continues program execution after an error recovery procedure is performed.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format:

RESUME [0]

RESUME NEXT

RESUME *line*

Remarks: Any of the formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0
Execution resumes at the statement which caused the error.

> **Note:** If you try to renumber a program containing a RESUME 0 statement, you will get an "Undefined line number" error. The statement will still say RESUME 0, which is okay.

RESUME NEXT Execution resumes at the statement immediately following the one which caused the error.

RESUME line Execution resumes at the specified line number.

RESUME
Statement

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to occur.

Example:

```
10 ON ERROR GOTO 900
 .
 .
 .
900 IF (ERR=230)AND(ERL=90) THEN PRINT
            "TRY AGAIN": RESUME 80
```

Line 900 is the beginning of the error trapping routine. The RESUME statement causes the program to return to line 80 when error 230 occurs in line 90.

RETURN Statement

Purpose: To bring you back from a subroutine. See "GOSUB and RETURN Statements" in this chapter.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

line valid only in Advanced and Compiler.

Format: RETURN [*line*]

Remarks: *line* is the line number of the program line you wish to return to. You may use it only in Advanced BASIC.

Although you can use RETURN *line* to return from any subroutine, this enhancement was added to allow non-local returns from the event trapping routines. From one of these routines you will often want to go back to the BASIC program at a fixed line number while still eliminating the GOSUB entry the trap created. Use of the non-local RETURN must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

RIGHT$ Function

Purpose: Returns the rightmost *n* characters of string *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = RIGHT$(*x$*,*n*)

Remarks: *x$* is any string expression.

n is an integer expression specifying the number of characters to be in the result.

If *n* is greater than or equal to LEN(*x$*), then *x$* is returned. If *n* is zero, the null string (length zero) is returned.

Also see the MID$ and LEFT$ functions.

Example:

```
Ok
10 A$="BOCA RATON, FLORIDA"
20 PRINT RIGHT$(A$,7)
RUN
FLORIDA
Ok
```

The rightmost seven characters of the string A$ are returned.

RND Function

Purpose: Returns a random number between 0 and 1.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: v = RND[(x)]

Remarks: x is a numeric expression which affects the returned value as described below.

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded. This is most easily done using the RANDOMIZE statement (see "RANDOMIZE Statement" in this chapter). You may also reseed the generator when you call the RND function by using x where x is negative. This always generates the particular sequence for the given x. This sequence is not affected by RANDOMIZE, so if you want to generate a different sequence each time the program is run, you must use a different value for x each time.

If x is positive or not included, RND(x) generates the next random number in the sequence.

RND(0) repeats the last number generated.

To get random numbers in the range 0 (zero) through n, use the formula:

```
INT( RND * (n+1))
```

RND
Function

Example:

```
Ok
10 FOR I=1 TO 3
20 PRINT RND(I);      ' x>0
30 NEXT I
40 PRINT: X=RND(-6)  ' x<0
50 FOR I=1 TO 3
60 PRINT RND(I);      ' x>0
70 NEXT I
80 RANDOMIZE 853 'randomize
90 PRINT: X=RND(-6)  ' x<0
100 FOR I=1 TO 3
110 PRINT RND;     ' same as x>0
120 NEXT I
130 PRINT: PRINT RND(0)
RUN
 .6291626   .1948297   .6305799
 .6818615   .4193624   .6215937
 .6818615   .4193624   .6215937
 .6215937
Ok
```

The first horizontal line of results shows three random numbers, generated using a positive *x*.

In line 40, a negative number is used to reseed the random number generator. The random numbers produced after this seeding are in the second row of results.

In line 80, the random number generator is reseeded using the RANDOMIZE statement; in line 90 it is reseeded again by calling RND with the same negative value we used in line 40. This cancels the effect of the RANDOMIZE statement, as you can see; the third line of results is identical to the second line.

In line 130, RND is called with an argument of zero, so the last number printed is the same as the preceding number.

RUN Command

Purpose: Begins execution of a program.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: RUN [*line*]

RUN *filespec*[,R]

Remarks: *line* is the line number of the program in memory where you wish execution to begin.

filespec is a string expression for the file specification, as explained under "Naming Files" in Chapter 3. The default extension .BAS is supplied for diskette files.

RUN or RUN *line* begins execution of the program currently in memory. If *line* is specified, execution begins with the specified line number. Otherwise, execution begins at the lowest line number.

RUN *filespec* loads a file from diskette or cassette into memory and runs it. It closes all open files and deletes the current contents of memory before loading the designated program. However, with the R option, all data files remain open. Refer also to "Appendix B. BASIC Diskette Input and Output."

Executing a RUN command will turn off any sound that is running and reset to Music Foreground. Also, PEN and STRIG will be reset to OFF.

RUN Command

Example:

```
Ok
10 PRINT 1/7
RUN
 .1428571
Ok
10 PI=3.141593
20 PRINT PI
RUN 20
 0
Ok
```

In this first example, we use the first form of RUN on two very small programs. The first program is run from the beginning. We used the RUN *line* option for the second example to run the program from line 20. In this case, line 10 does not get executed, so PI does not receive its proper value. A 0 is printed because all numeric variables have an initial value of zero.

```
RUN "CAS1:NEWFIL",R
```

The preceding example loads the program "NEWFIL" from the tape and runs it, keeping files open.

SAVE Command

Purpose: Saves a BASIC program file on diskette or cassette.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: SAVE *filespec* [,A]

SAVE *filespec* [,P]

Remarks: *filespec* is a string expression for the file specification. If *filespec* does not conform to the rules outlined under "Naming Files" in Chapter 3, an error is issued and the save is cancelled.

The BASIC program is written to the specified device. When saving to CAS1:, the cassette motor is turned on and the file is immediately written to the tape.

For diskette files, if the filename is eight characters or less and no extension is supplied, the extension **.BAS** is added to the name. If a file with the same filename already exists on the diskette, it will be written over.

When using Cassette BASIC, if the device name is omitted, CAS1: is assumed. CAS1: is the only allowable device for SAVE in Cassette BASIC.

For Disk and Advanced BASIC, the device defaults to the DOS default drive.

The A option saves the program in ASCII format. Otherwise, BASIC saves the file in a compressed binary (tokenized) format. ASCII files take up more space, but some types of access require that files be

in ASCII format. For example, a file intended to be merged must be saved in ASCII format. Programs saved in ASCII may be read as data files.

The **P** option saves the program in an encoded binary format. This is the protection option. When a protected program is later run (or loaded), any attempt to LIST or EDIT it fails with an "Illegal function call" error. No way is provided to "unprotect" such a program.

> **Note:** The diskette directory entry for a BASIC program file gives no indication that the file is either protected or stored in ASCII format. The **.BAS** extension is used in any case.

See also "Appendix B. BASIC Diskette Input and Output."

Example:

```
SAVE "INVENT"
```

Saves the program in memory as INVENT. The program is saved on cassette if you are using Cassette BASIC. If you are using Disk or Advanced BASIC, the program is saved on the diskette in the DOS default drive and given an extension of **.BAS**.

```
SAVE "B:PROG",A
```

Saves PROG.BAS on drive B: in ASCII, so it may later be merged.

```
SAVE "A:SECRET.BOZ",P
```

Saves SECRET.BOZ on drive A:, protected so it may not be altered.

SCREEN Function

Purpose: Returns the ASCII code (0-255) for the character on the active screen at the specified row (line) and column.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = SCREEN(*row,col*[,*z*])

Remarks: *row* is a numeric expression in the range 1 to 25.

col is a numeric expression in the range 1 to 40 or 1 to 80 depending upon the WIDTH setting.

z is a numeric expression which evaluates to a true or false value. *z* is only valid in text mode.

Refer to "Appendix G. ASCII Character Codes" for a list of ASCII codes.

In text mode, if *z* is included and is true (non-zero), the color attribute for the character is returned instead of the code for the character. The color attribute is a number in the range 0 to 255. This number, *v*, may be deciphered as follows:

(*v* MOD 16) is the foreground color.

(((*v* - *foreground*)/16) MOD 128) is the background color, where *foreground* is calculated as above.

(*v*>127) is true (-1) if the character is blinking, false (0) if not.

SCREEN Function

Refer to "COLOR Statement" for a list of colors and their associated numbers.

In graphics mode, if the specified location contains graphic information (points or lines, as opposed to just a character), then the SCREEN function returns zero.

Any values entered outside of the ranges indicated result in an "Illegal function call" error.

The SCREEN *statement* is explained in the next section.

Example:

```
100 X = SCREEN (10,10)
```

If the character at 10,10 is A, then X is 65.

```
110 X = SCREEN (1,1,1)
```

Returns the color attribute of the character in the upper left hand corner of the screen.

SCREEN Statement

Purpose: Sets the screen attributes to be used by subsequent statements.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Meaningful with the Color/Graphics Monitor Adapter only.

Format: SCREEN [*mode*] [,[*burst*] [,[*apage*] [,*vpage*]]]

Remarks: *mode* is a numeric expression resulting in an integer value of 0, 1 or 2. Valid modes are:

- 0 Text mode at current width (40 or 80).
- 1 Medium resolution graphics mode (320x200). Use with Color/Graphics Monitor Adapter only.
- 2 High resolution graphics mode (640x200). Use with Color/Graphics Monitor Adapter only.

burst is a numeric expression resulting in a true or false value. It enables color. In text mode (*mode*=0), a false (zero) value disables color (black and white images only) and a true (non-zero) value enables color (allows color images). In medium resolution graphics mode (*mode*=1), a true (non-zero) value will disable color, and a false (zero) value will enable color. Since black and white are the only colors in high resolution graphics (*mode*=2), this parameter will not have much effect in high resolution.

apage (active page) is an integer expression in the range 0 to 7 for width 40, or 0 to 3 for width 80. It selects the page to be written to by output statements to the screen, and is valid in text mode (*mode*=0) only.

vpage (visual page) selects which page is to be displayed on the screen, in the same way as *apage* above. The visual page may be different than the active page. *vpage* is valid in text mode (*mode*=0) only. If omitted, *vpage* defaults to *apage*.

If all parameters are valid, the new screen mode is stored, the screen is erased, the foreground color is set to white, and the background and border colors are set to black.

If the new screen mode is the same as the previous mode, nothing is changed.

If the mode is text, and only *apage* and *vpage* are specified, the effect is that of changing display pages for viewing. Initially, both active and visual pages default to 0 (zero). By manipulating active and visual pages, you can display one page while building another. Then you can switch visual pages instantaneously.

> Note: There is only one cursor shared between all the pages. If you are going to switch active pages back and forth, you should save the cursor position on the current active page (using POS(0) and CSRLIN), before changing to another active page. Then when you return to the original page, you can restore the cursor position using the LOCATE statement.

Any parameter may be omitted. Omitted parameters, except *vpage*, assume the old value.

Any values entered outside of the ranges indicated will result in an "Illegal function call" error. Previous values are retained.

If you are writing a program which is intended to be run on a machine that may have either adapter, we suggest you use the SCREEN 0,0,0 and WIDTH 40 statements at the beginning of the program.

Example:

```
10 SCREEN 0,1,0,0
```

Selects text mode with color, and sets active and visual page to 0.

```
20 SCREEN ,,1,2
```

Mode and color burst remain unchanged. Active page is set to 1 and display page to 2.

```
30 SCREEN 2,,0,0
```

Switches to high resolution graphics mode.

```
40 SCREEN 1,0
```

Switches to medium resolution color graphics.

```
50 SCREEN ,1
```

Sets medium resolution graphics with color off.

SGN Function

Purpose: Returns the sign of *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = SGN(*x*)

Remarks: *x* is any numeric expression.

SGN(*x*) is the mathematical signum function:

- If *x* is positive, SGN(*x*) returns 1.
- If *x* is zero, SGN(*x*) returns 0.
- If *x* is negative, SGN(*x*) returns -1.

Example:

```
ON SGN(X)+2 GOTO 1ØØ,2ØØ,3ØØ
```

branches to 100 if X is negative, 200 if X is zero, and 300 if X is positive.

SIN Function

Purpose: Calculates the trigonometric sine function.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: $v = \text{SIN}(x)$

Remarks: x is an angle in radians.

If you want to convert degrees to radians, multiply by PI/180, where PI=3.141593.

SIN(x) is calculated in single precision.

Example:

```
Ok
10 PI=3.141593
20 DEGREES = 90
30 RADIANS=DEGREES * PI/180 ' PI/2
40 PRINT SIN(RADIANS)
RUN
 1
Ok
```

This example calculates the sine of 90 degrees, after first converting the degrees to radians.

SOUND
Statement

Purpose: Generates sound through the speaker.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: SOUND *freq, duration*

Remarks: *freq* is the desired frequency in Hertz (cycles per second). It must be a numeric expression in the range 37 to 32767.

duration is the desired duration in clock ticks. The clock ticks occur 18.2 times per second. *duration* must be a numeric expression in the range 0 to 65535.

When the SOUND statement produces a sound, the program continues to execute until another SOUND statement is reached. If *duration* of the new SOUND statement is zero, the current SOUND statement that is running is turned off. Otherwise, the program waits until the first sound completes before it executes the new SOUND statement.

If you are using Advanced BASIC, you can cause the sounds to be buffered so execution does not stop when a new SOUND statement is encountered. See the MB command explained under "PLAY Statement" in this chapter for details.

If no SOUND statement is running, SOUND *x*,0 has no effect.

The tuning note, A, has a frequency of 440. The following table correlates notes with their frequencies for two octaves on either side of middle C.

Note	Frequency	Note	Frequency
C	130.810	C*	523.250
D	146.830	D	587.330
E	164.810	E	659.260
F	174.610	F	698.460
G	196.000	G	783.990
A	220.000	A	880.000
B	246.940	B	987.770
C	261.630	C	1046.500
D	293.660	D	1174.700
E	329.630	E	1318.500
F	349.230	F	1396.900
G	392.000	G	1568.000
A	440.000	A	1760.000
B	493.880	B	1975.500

*middle C. Higher (or lower) notes may be approximated by doubling (or halving) the frequency of the corresponding note in the previous (next) octave.

To create periods of silence, use SOUND 32767,*duration*.

The duration for one beat can be calculated from beats per minute by dividing the beats per minute into 1092 (the number of clock ticks in a minute).

The next table shows typical tempos in terms of clock ticks:

Tempo		Beats/ Minute	Ticks/ Beat
very slow	Larghissimo		
↓	Largo	40-60	27.3-18.2
↓	Larghetto	60-66	18.2-16.55
↓	Grave		
↓	Lento		
↓	Adagio	66-76	16.55-14.37
slow	Adagietto		
↓	Andante	76-108	14.37-10.11
medium	Andantino		
↓	Moderato	108-120	10.11-9.1
fast	Allegretto		
↓	Allegro	120-168	9.1-6.5
↓	Vivace		
↓	Veloce		
↓	Presto	168-208	6.5-5.25
very fast	Prestissimo		

Example: The following program creates a glissando up and down.

```
10 FOR I=440 TO 1000 STEP 5
20 SOUND I, 0.5
30 NEXT
40 FOR I=1000 TO 440 STEP -5
50 SOUND I, 0.5
60 NEXT
```

SPACE$ Function

Purpose: Returns a string consisting of *n* spaces.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = SPACE$(*n*)

Remarks: *n* must be in the range 0 to 255.

Refer also to the SPC function.

Example:

```
Ok
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
Ok
```

This example uses the SPACE$ function to print each number I on a line preceded by I spaces. An additional space is inserted because BASIC puts a space in front of positive numbers.

SPC Function

Purpose: Skips *n* spaces in a PRINT statement.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT SPC(*n*)

Remarks: *n* must be in the range 0 to 255.

If *n* is greater than the defined width of the device, then the value used is *n* MOD *width*. SPC may only be used with PRINT, LPRINT and PRINT # statements.

If the SPC function is at the end of the list of data items, then BASIC does not add a carriage return, as though the SPC function had an implied semicolon after it.

Also see the SPACE$ function.

Example:

```
Ok
PRINT "OVER" SPC(15) "THERE"
OVER                 THERE
Ok
```

This example prints OVER and THERE separated by 15 spaces.

SQR Function

Purpose: Returns the square root of x.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: $v = \text{SQR}(x)$

Remarks: x must be greater than or equal to zero.

Example

```
Ok
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.162278
 15            3.872984
 20            4.472136
 25            5
Ok
```

This example calculates the square roots of the numbers 10, 15, 20 and 25.

STICK Function

Purpose: Returns the x and y coordinates of two joysticks.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = STICK(*n*)

Remarks: *n* is a numeric expression in the range 0 to 3 which affects the result as follows:

0 returns the x coordinate for joystick A.

1 returns the y coordinate of joystick A.

2 returns the x coordinate of joystick B.

3 returns the y coordinate of joystick B.

Note: STICK(0) retrieves all four values for the coordinates, and returns the value for STICK(0). STICK(1), STICK(2), and STICK(3) do not sample the joystick. They get the values previously retrieved by STICK (0).

The range of values for x and y depends on your particular joysticks.

Example:

```
10 PRINT "Joystick B"
20 PRINT "x coordinate","y coordinate"
30 FOR J=1 TO 100
40 TEMP=STICK(0)
50 X=STICK(2): Y=STICK(3)
60 PRINT X,Y
70 NEXT
```

This program takes 100 samples of the coordinates of joystick B and prints them.

STOP
Statement

Purpose: Terminates program execution and returns to command level.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: STOP

Remarks: STOP statements may be used anywhere in a program to terminate execution. When BASIC encounters a STOP statement, it displays the following message:

```
Break in nnnnn
```

where nnnnn is the line number where the STOP occurred.

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after it executes a STOP. You can resume execution of the program by issuing a CONT command (see "CONT Command" in this chapter).

Example:

```
10 INPUT A, B
20 TEMP= A*B
30 STOP
40 FINAL = TEMP+200: PRINT FINAL
RUN
? 26, 2.1
Break in 30
Ok
PRINT TEMP
 54.6
Ok
CONT
 254.6
Ok
```

This example calculates the value of TEMP, then stops. While the program is stopped, we can check the value of TEMP. Then we can use CONT to resume program execution at line 40.

STR$ Function

Purpose: Returns a string representation of the value of *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = STR$(*x*)

Remarks: *x* is any numeric expression.

If *x* is positive, the string returned by STR$ contains a leading blank (the space reserved for the plus sign). For example:

```
Ok
? STR$(321); LEN(STR$(321))
 321 4
Ok
```

The VAL function is complementary to STR$.

Example: This example branches to different sections of the program based on the number of digits in a number that is entered. The digits in the number are counted by using STR$ to convert the number to a string, then branching based on the length of the string.

```
5 REM arithmetic for kids
1Ø INPUT "TYPE A NUMBER";N
2Ø ON LEN(STR$(N))-1 GOSUB 3Ø,1ØØ,2ØØ,3ØØ
 •
 •
 •
```

STRIG
Statement and Function

Purpose: Returns the status of the joystick buttons (triggers).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: As a statement:

STRIG ON

STRIG OFF

As a function:

v = STRIG(*n*)

Remarks: *n* is a numeric expression in the range 0 to 3. It affects the value returned by the function as follows:

0 Returns -1 if button A1 was pressed since the last STRIG(0) function call, returns 0 if not.

1 Returns -1 if button A1 is currently pressed, returns 0 if not.

2 Returns -1 if button B1 was pressed since the last STRIG(2) function call, returns 0 if not.

3 Returns -1 if button B1 is currently pressed, returns 0 if not.

STRIG
Statement and Function

In Advanced BASIC and the BASIC Compiler, you can read four buttons from the joysticks. The additional values for *n* are:

4 Returns -1 if button A2 was pressed since the last STRIG(4) function call, returns 0 if not.

5 Returns -1 if button A2 is currently pressed, returns 0 if not.

6 Returns -1 if button B2 was pressed since the last STRIG(6) function call, returns 0 if not.

7 Returns -1 if button B2 is currently pressed, returns 0 if not.

STRIG ON must be executed before any STRIG(*n*) function calls may be made. After STRIG ON, every time the program starts a new statement BASIC checks to see if a button has been pressed.

If STRIG is OFF, no testing takes place.

Refer also to the next section, "STRIG(n) Statement" for enhancements to the STRIG function in Advanced BASIC.

STRIG(n) Statement

Purpose: Enables and disables trapping of the joystick buttons.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format:

STRIG(*n*) ON

STRIG(*n*) OFF

STRIG(*n*) STOP

Remarks: *n* may be 0, 2, 4, or 6, and indicates the button to be trapped as follows:

0	button A1
2	button B1
4	button A2
6	button B2

STRIG(*n*) ON must be executed to enable trapping by the ON STRIG(*n*) statement (see "ON STRIG(n) Statement" in this chapter). After STRIG(*n*) ON, every time the program starts a new statement, BASIC checks to see if the specified button has been pressed.

If STRIG(*n*) OFF is executed, no testing or trapping takes place. Even if the button is pressed, the event is not remembered.

If a STRIG(*n*) STOP statement is executed, no trapping takes place. However, if the button is pressed it is remembered so that an immediate trap takes place when STRIG(*n*) ON is executed.

Refer also to the previous section, "STRIG Statement and Function."

STRING$ Function

Purpose: Returns a string of length *n* whose characters all have ASCII code *m* or the first character of *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = STRING$(*n*,*m*)

*v$* = STRING$(*n*,*x$*)

Remarks: *n, m* are in the range 0 to 255.

x$ is any string expression.

Example:

```
Ok
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

The first example repeats an ASCII value of 45 to print a string of hyphens.

```
Ok
10 X$="ABCD"
20 Y$+STRING$(10,X$)
30 PRINT Y$
RUN
AAAAAAAAAA
Ok
```

The second example repeats the first character of the string "ABCD".

SWAP Statement

Purpose: Exchanges the values of two variables.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: SWAP *variable1, variable2*

Remarks: *variable1, variable2*
are the names of two variables or array elements.

Any type variable may be swapped (integer, single-precision, double-precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example:

```
Ok
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
 ONE FOR ALL
 ALL FOR ONE
Ok
```

After line 30 is executed, A$ has the value " ALL " and B$ has the value " ONE ".

SYSTEM Command

Purpose: Exits BASIC and returns to DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: SYSTEM

Remarks: SYSTEM closes all files before it returns to DOS. Your BASIC program is lost.

If you entered BASIC through a Batch file from DOS, the SYSTEM command returns you to the Batch file, which continues executing at the point it left off.

TAB Function

Purpose: Tabs to position *n*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT TAB(*n*)

Remarks: *n* must be in the range 1 to 255.

If the current print position is already beyond space *n*, TAB goes to position *n* on the next line. Space 1 is the leftmost position, and the rightmost position is the defined WIDTH.

TAB may only be used in PRINT, LPRINT, and PRINT # statements.

If the TAB function is at the end of the list of data items, then BASIC does not add a carriage return, as though the TAB function had an implied semicolon after it.

Example: TAB is used in the following example to cause the information on the screen to line up in columns.

```
Ok
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "L. M. JACOBS","$25.00"
RUN
NAME                      AMOUNT

L. M. JACOBS              $25.00
Ok
```

TAN Function

Purpose: Returns the trigonometric tangent of x.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: v = TAN(x)

Remarks: x is the angle in radians. To convert degrees to radians, multiply by PI/180, where PI=3.141593.

TAN(x) is calculated in single precision.

Example:

```
Ok
10 PI=3.141593
20 DEGREES=45
30 PRINT TAN(DEGREES*PI/180)
RUN
 1
Ok
```

This example calculates the tangent of 45 degrees.

TIME$
Variable and Statement

Purpose: Sets or retrieves the current time.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: As a variable:

*v$* = TIME$

As a statement:

TIME$ = *x$*

Remarks: **For the variable (v$ = TIME$):**

The current time is returned as an 8 character string. The string is of the form *hh:mm:ss*, where *hh* is the hour (00 to 23), *mm* is the minutes (00 to 59), and *ss* is the seconds (00 to 59). The time may have been set by DOS prior to entering BASIC.

For the statement (TIME$ = x$):

The current time is set. *x$* is a string expression indicating the time to be set. *x$* may be given in one of the following forms:

hh Set the hour in the range 0 to 23. Minutes and seconds default to 00.

hh:mm Set the hour and minutes. Minutes must be in the range 0 to 59. Seconds default to 00.

hh:mm:ss Set the hour, minutes, and seconds. Seconds must be in the range 0 to 59.

A leading zero may be omitted from any of the above values, but you must include at least one digit. For example, if you wanted to set the time as a half hour after midnight, you could enter TIME$="0:30", but not TIME$=":30". If any of the values are out of range, an "Illegal function call" error is issued. The previous time is retained. If *x$* is not a valid string, a "Type mismatch" error results.

Example: The following program displays the time continuously in the middle of the screen.

```
10 CLS
20 LOCATE 10,15
30 PRINT TIME$
40 GOTO 30
```

TRON and TROFF Commands

Purpose: Traces the execution of program statements.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: TRON

TROFF

Remarks: As an aid in debugging, the TRON command (which may be entered in indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace is turned off by the TROFF command.

Example:

```
Ok
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
TRON
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
```

This example uses TRON and TROFF to trace execution of a loop. The numbers in brackets are line numbers; the numbers not in brackets at the end of each line are the values of J, K, and L which are printed by the program.

USR Function

Purpose: Calls the indicated machine language subroutine with the argument *arg*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: *v* = USR[*n*](*arg*)

Remarks: *n* is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for the desired routine (see "DEF USR Statement" in this chapter). If *n* is omitted, USR0 is assumed.

arg is any numeric expression or string variable, which will be the argument to the machine language subroutine.

The CALL statement is another way to call a machine language subroutine. See "Appendix C. Machine Language Subroutines" for complete information on using machine language subroutines.

Example:

```
10 DEF USR0 = &HF000
50 C = USR0(B/2)
60 D = USR(B/3)
```

The function USR0 is defined in line 10. Line 50 calls the function USR0 with the argument B/2. Line 60 calls USR0 again, with the argument B/3.

VAL Function

Purpose: Returns the numerical value of string *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = VAL(*x$*)

Remarks: *x$* is a string expression.

The VAL function strips blanks, tabs, and line feeds from the argument string in order to determine the result. For example,

```
VAL("    -3")
```

returns -3.

If the first characters of *x$* are not numeric, then VAL(*x$*) will return 0 (zero).

See the STR$ function for numeric to string conversion.

Example:

```
Ok
PRINT VAL("3408 SHERWOOD BLVD.")
 3408
Ok
```

In this example, VAL is used to extract the house number from an address.

VARPTR Function

Purpose: Returns the address in memory of the variable or file control block.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = VARPTR(*variable*)

v = VARPTR(*#filenum*)

Remarks: *variable* is the name of a numeric or string variable or array element in your program. A value must be assigned to *variable* prior to the call to VARPTR, or an "Illegal function call" error results.

filenum is the number under which the file was opened.

For both formats, the address returned is an integer in the range 0 to 65535. This number is the offset into BASIC's Data Segment. The address is not affected by the DEF SEG statement.

The first format returns the address of the first byte of data identified with *variable*. The format of this data is described in Appendix I under "How Variables Are Stored."

> **Note:** All simple variables should be assigned before calling VARPTR for an array, because addresses of arrays change whenever a new simple variable is assigned.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to a USR machine language subroutine. A function call of the

form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

The second format returns the starting address of the file control block for the specified file. This is not the same as the DOS file control block. Refer to "BASIC File Control Block" in "Appendix I. Technical Information and Tips" for detailed information about the format of the file control block.

VARPTR is meaningless for cassette files.

Example: This example reads the first byte in the buffer of a random file:

```
10 OPEN "DATA.FIL" AS #1
20 GET #1
30 'get address of control block
40 FCBADR = VARPTR(#1)
50 'figure address of data buffer
60 DATADR = FCBADR+188
70 'get first byte in data buffer
80 A% = PEEK(DATADR)
```

The next example use VARPTR to get the data from a variable. In line 30, P gets the address of the data. Integer data is stored in two bytes, with the less significant byte first. The actual value stored at location P is calculated in line 40. The bytes are read with the PEEK function, and the second byte is multiplied by 256 because it contains the high-order bits.

```
10 DEFINT A-Z
20 DATA1=500
30 P=VARPTR(DATA1)
40 V=PEEK(P) + 256*PEEK(P+1)
50 PRINT V
```

VARPTR$ Function

Purpose: Returns a character form of the address of a variable in memory. It is primarily for use with PLAY and DRAW in programs that will later be compiled.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: *v$* = VARPTR$(*variable*)

Remarks: *variable* is the name of a variable existing in the program.

Note: All simple variables should be assigned before calling VARPTR$ for an array element, because addresses of arrays change whenever a new simple variable is assigned.

VARPTR$ returns a three-byte string in the form:

Byte 0	Byte 1	Byte 2
type	low byte of variable address	high byte of variable address

type indicates the variable type:

2 integer
3 string
4 single-precision
8 double-precision

The returned value is the same as:

CHR$(*type*)+MKI$(VARPTR(*variable*))

You can use VARPTR$ to indicate a variable name in the command string for PLAY or DRAW. For example:

Method One	Alternative Method
`PLAY "XA$;"`	`PLAY "X"+VARPTR$(A$)`
`PLAY "O=I;"`	`PLAY "O="+VARPTR$(I)`

This technique is mainly for use in programs which will later be compiled.

WAIT Statement

Purpose: Suspends program execution while monitoring the status of a machine input port.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: WAIT *port, n*[,*m*]

Remarks: *port* is the port number, in the range 0 to 65535.

n, m are integer expressions in the range 0 to 255.

Refer to the *IBM Personal Computer Technical Reference* manual for a description of valid port numbers (I/O addresses).

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern.

The data read at the port is XORed with the integer expression *m* and then ANDed with *n*. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *m* is omitted, it is assumed to be zero.

WAIT Statement

The WAIT statement lets you test one or more bit positions on an input port. You can test the bit position for either a 1 or a 0. The bit positions to be tested are specified by setting 1's in those positions in *n*. If you do not specify *m*, the input port bits are tested for 1's. If you do specify *m*, a 1 in any bit position in *m* (for which there is a 1 bit in *n*) causes WAIT to test for a 0 for that input bit.

When executed, the WAIT statement loops testing those input bits specified by 1's in *n*. If *any one* of those bits is 1 (or 0 if the corresponding bit in *m* is 1), then the program continues with the next statement. Thus WAIT does not wait for an entire pattern of bits to appear, but only for one of them to occur.

> **Note:** It is possible to enter an infinite loop with the WAIT statement. You can do a Ctrl-Break or a System Reset to exit the loop.

Example: To suspend program execution until port 32 receives a 1 bit in the second bit position:

```
1ØØ WAIT 32,2
```

WHILE and WEND Statements

Purpose: Executes a series of statements in a loop as long as a given condition is true.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format:

WHILE *expression*
.
.
.
(*loop statements*)
.
.
.
WEND

Remarks: *expression* is any numeric expression.

If *expression* is true (not zero), *loop statements* are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks *expression*. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE...WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example: This example sorts the elements of the string array A$ into alphabetical order. A$ was defined with J elements.

```
90 'bubble sort array A$
100 FLIPS=1 'force one pass thru loop
110 WHILE FLIPS
115  FLIPS=0
120  FOR I=1 TO J-1
130   IF A$(I)>A$(I+1) THEN
         SWAP A$(I),A$(I+1): FLIPS=1
140  NEXT I
150 WEND
```

WIDTH
Statement

Purpose: Sets the output line width in number of characters. After outputting the indicated number of characters, BASIC adds a carriage return.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format:

WIDTH *size*

WIDTH *filenum,size*

WIDTH *device,size*

Remarks:

size is a numeric expression in the range 0 to 255. This is the new width. WIDTH 0 is the same thing as WIDTH 1.

filenum is a numeric expression in the range 1 to 15. This is the number of a file opened to one of the devices listed below.

device is a string expression for the device identifier. Valid devices are SCRN:, LPT1:, LPT2:, LPT3:, COM1:, or COM2:.

Depending upon the device specified, the following actions are possible:

WIDTH *size* or WIDTH "SCRN:",*size*
Sets the screen width. Only 40 or 80 column width is allowed.

If the screen is in medium resolution graphics mode (as would occur with a SCREEN 1 statement), WIDTH 80 forces the screen into high resolution (just like a SCREEN 2 statement).

If the screen is in high resolution graphics mode (as would occur with a SCREEN 2 statement), WIDTH 40 forces the screen into medium resolution (like a SCREEN 1 statement).

> Note: Changing the screen width causes the screen to be cleared, and sets the border screen color to black.

WIDTH *device,size*

Used as a deferred width assignment for the device. This form of width stores the new width value without actually changing the current width setting. A subsequent OPEN to the device will use this value for width while the file is open. The width does not change immediately if the device is already open.

> Note: LPRINT, LLIST, and LIST,"LPTn:" do an implicit OPEN and are therefore affected by this statement.

WIDTH *filenum,size*

The width of the device associated with *filenum* is immediately changed to the new size specified. This allows the width to be changed at will while the file is open. This form of WIDTH has meaning only for LPT1: in Cassette BASIC. Disk and Advanced BASIC also allow LPT2:, LPT3:, COM1: and COM2:.

Any value entered outside of the ranges indicated will result in an "Illegal function call" error. The previous value is retained.

WIDTH has no effect for the keyboard (KYBD:) or cassette (CAS1:).

The width for each printer defaults to 80 when BASIC is started. The maximum width for the IBM 80 CPS Matrix Printer is 132. However, no error is returned for values between 132 and 255.

It is up to you to set the appropriate physical width on your printer. Some printers are set by sending special codes, some have switches. For the IBM 80 CPS Matrix Printer you should use LPRINT CHR$(15); to change to a condensed typestyle when printing at widths greater than 80. Use LPRINT CHR$(18); to return to normal. The IBM 80 CPS Matrix Printer is set up to automatically add a carriage return if you exceed the maximum line length.

Specifying a width of 255 disables line folding. This has the effect of "infinite" width. WIDTH 255 is the default for communications files.

Changing the width for a communications file does not alter either the receive or the transmit buffer; it just causes BASIC to send a carriage return character after every *size* characters.

Changing screen mode affects screen width only when moving between SCREEN 2 and SCREEN 1 or SCREEN 0. See "SCREEN Statement" in this chapter.

Example:

```
10 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
 .
 .
 .
6020 WIDTH #1,40
```

In the preceding example, line 10 stores a printer width of 75 characters per line. Line 20 opens file #1 to the printer and sets the width to 75 for subsequent PRINT #1,... statements. Line 6020 changes the current printer width to 40 characters per line.

```
SCREEN 1,0  'Set to med-res color graphics
WIDTH 80    'Go to hi-res graphics
WIDTH 40    'Go back to medium res

SCREEN 0,1  'Go to 40x25 text color mode
WIDTH 80    'Go to 80x25 text color mode
```

WRITE
Statement

Purpose: Outputs data on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: WRITE [*list of expressions*]

Remarks: *list of expressions*
is a list of numeric and/or string expressions, separated by commas or semicolons.

If the list of expressions is omitted, a blank line is output. If the list of expressions is included, the values of the expressions are output on the screen.

When the values of the expressions are output, each item is separated from the last by a comma. Strings are delimited by quotation marks. After the last item in the list is printed, BASIC adds a carriage return/line feed.

WRITE is similar to PRINT. The difference between WRITE and PRINT is that WRITE inserts commas between the items as they are displayed and delimits strings with quotation marks. Also, positive numbers are not preceded by blanks.

Example: This example shows how WRITE displays numeric and string values.

```
10 A=80: B=90: C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
80,90,"THAT'S ALL"
Ok
```