LOC

Format **LOC (<file number>)**

Purpose With random access files, returns the record number which will be used by the next GET or PUT statement if that statement is executed without specifying a record number. With sequential files, returns the number of file sectors (128-byte areas) which have been read or written since the file was opened.

When the specified file is the PX-8's RS-232C interface, the LOC function returns the number of bytes of data in the RS-232C receive buffer.

Remarks This function can be used to control the flow of program execution according to the number of records or file sectors which have been accessed by a program since the file was opened.

Example

```
10 ON ERROR GOTO 160
20 OPEN"R",#1,"A:LOCTEST",5
30 PRINT "OUTPUT"
40 FIELD#1,5 AS A$
50 FOR A=1 TO 20:LSET A$=STR$(A):PRINT STR$(A);:PUT#1,A:NEXT
60 PRINT
70 CLOSE
80 OPEN"R",#1,"A:LOCTEST",5
90 PRINT "INPUT"
100 FIELD #1,5 AS A$
110 IF LOC(1)>10 THEN 150
120 GET#1
130 PRINT A$;
140 GOTO 110
150 ERROR 230
160 IF ERR=230 THEN PRINT:PRINT "INPUT PAST LIMIT"
170 END

OUTPUT
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
INPUT
 1    2    3    4    5    6    7    8    9    10   11
INPUT PAST LIMIT
Ok
```

LOCATE

Format **LOCATE [<X>][,[<Y>][,<cursor switch>]]**

Purpose Moves the cursor to the specified screen coordinates.

Remarks This statement moves the cursor to the screen position whose horizontal character coordinate is specified by <X> and whose vertical character coordinate is specified by <Y>. The value specified for <X> must be in the range from 1 to Xmax, where Xmax is the number of columns in the currently selected virtual screen. The value specified for <Y> must be in the range from 1 to Ymax, where Ymax is the number of lines in the currently selected virtual screen.

<cursor switch> is a switch which determines the status of the cursor following execution of the LOCATE statement. Cursor display is turned off if 0 is specified for <cursor switch>, and is turned on if 1 is specified. Normally, the cursor is not displayed during execution of BASIC programs; however, it can be forcibly displayed by executing a LOCATE statement with 1 specified for <cursor switch>. Cursor display is also forcibly turned on during execution of INPUT and LINE INPUT statements and the INPUT$ function, regardless of the status of the cursor switch.

Example

```
10 CLS:LOCATE 10,3,0
20 PRINT"This has been printed starting at line 3 column 10"
30 LOCATE 10,4,0
40 PRINT"and the cursor has been switched off."
50 FOR J = 1 TO 3000:NEXT
60 CLS:LOCATE 6,4,1:PRINT"This has been printed starting at
line 4 column 6,"
70 LOCATE 6,5,1:PRINT"and the cursor has been switched on ag
ain."
80 FOR J = 1 TO 3000:NEXT
```

```
         This has been printed starting at line 3 column 10
         and the cursor has been switched off.

     This has been printed starting at line 4 column 6,
     and the cursor has been switched on again.
```

NOTE:
The expression

```
LOCATE X, Y
```

will print at location X columns across on row number Y. When the length of the string to be printed plus the value of X is greater than 81, the string will be printed at the start of the next line.

LOF

Format **LOF (<file number>)**

Purpose Returns the size of a file.

Remarks When the file specified in <file number> is a disk file, the LOF function returns the size of that file. If the file has been opened in the "R" mode, the size of the file is calculated using the highest record number of that file and the record size under which the file was opened. If the file was opened in the sequential mode, the size is calculated according to the number of records, based on a record size of 128 bytes.

When the specified file is the PX-8's RS-232C interface, the LOF function returns the number of free bytes remaining in the RS-232C receive buffer.

Example

```
10 OPEN "O",#1,"A:LENGTH"
20 FOR J = 1 TO 5
30 READ A$
40 PRINT #1,A$
50 NEXT
60 DATA JIM,FRED,SHEILA,SUSAN,HARRY
70 CLOSE #1
80 OPEN "I",#1,"A:LENGTH"
115 X = LOF(1)
120 PRINT"The length of this file is ";X;" x 128 bytes"
130 CLOSE #1
140 END

run
The length of this file is  1  x 128 bytes
Ok
```

LOG

Format **LOG(X)**

Purpose Returns the natural logarithm of X.

Remarks The value specified for X must be greater than zero. LOG(X) is calculated to the precision of the numeric type of expression X.

To obtain the logarithm to another base the mathematical conversion has to be carried out as in the first example below.

To obtain a number from its logarithm (i.e. its antilogarithm) use EXP (X) as shown in the second example.

Example 1

```
10 CLS
20 INPUT "What base logarithm do you want ";N
30 PRINT:INPUT "What number do you want the log of ";X
40 Z = (LOG(X)/LOG(N)): 'THIS IS THE FORMULA FOR CONVERTING
NATURAL LOGS TO OTHER BASES
50 PRINT: PRINT "Log to the base ";N;" of ";X;" is ";Z
60 END
```

```
What base logarithm do you want ? 10

What number do you want the log of ? 100

Log to the base 10 of 100 is 2
Ok
```

```
What base logarithm do you .want ? 8

What number do you want the log of ? 34

Log to the base 8 of 34 is 1.69582
Ok
```

Example 2

```
10 CLS
20 INPUT "What is the number of which you want the natural
   log ";X
30 PRINT:PRINT"The log to the base e of ";X;" is ";LOG(X)
40 PRINT:PRINT"The antilog (given by EXP(X)) is ";EXP(LOG(X))
50 END
```

```
What is the number of which you want the natural log ? 23

The log to the base e of  23  is  3.13549

The antilog (given by EXP(X)) is  23
Ok
```

```
What is the number of which you want the natural log ? 657

The log to the base e of  657  is  6.48769

The antilog (given by EXP(X)) is  657
Ok
```

LOGIN

Format **LOGIN <program area no.>[,R]**

Purpose Switches between BASIC program areas.

Remarks The number of the BASIC program area to be selected is specified in <program area no.> as a number from 1 to 5. If the R option is not specified, executing this command clears all variables, closes all files, and switches BASIC to the designated program area and causes it to stand by for input of commands in the direct mode.

When the R option is specified, the specified program area is selected and the program in that area is executed immediately, starting with its first line. In this case, all variables remain intact and any files which are open at the time of execution of the LOGIN command remain open.

An "Illegal function call" error will result if a number other than 1 to 5 is specified in <program area no.>.

Example

```
10 PRINT"This program will log in to program area 2
20 PRINT"and execute a resident program"
30 LOGIN 2,R

run
This program will log in to program area 2
and execute a resident program
P2:LOGIN2       60 Bytes
This is the program resident in program area 2
Ok
```

```
10 PRINT"This is the program resident in program area 2"
20 END
```

Program logged in area 2

LPOS

Format **LPOS(X)**

Purpose Returns the current position of the print head pointer in the printer output buffer.

Remarks The maximum value returned by LPOS is determined by the line width which has been set by the WIDTH LPRINT statement, and does not necessarily correspond to the physical position of the print head. This is especially true if a control character has been sent to the printer; see the program below for an example of this.

X is a dummy argument, and may be specified as any numeric expression.

Example In the example below, at the end of line 100 ten characters have been printed. The position in the buffer is thus 11. Line 20 adds two control characters compatible with EPSON printers to cause the printer to change the print style. The first character is a control character, which is ignored by the LPOS function. The second character is used by the printer but not printed; it is in fact the letter "E". The position in the buffer as returned by LPOS is now 12 because only this character has been added. Line 30 adds another ten characters to the line, and thus LPOS returns a value of 22.

```
10 LPRINT "1234567890";:GOSUB 100
20 LPRINT CHR$(27);CHR$(69);:GOSUB 100
30 LPRINT "1234567890";:GOSUB 100
40 END
100 A = LPOS(X):PRINT"Print head pointer is at position ";A
110 RETURN

12345678901234567890

Ok
run
Print head pointer is at position  11
Print head pointer is at position  12
Print head pointer is at position  22
Ok
```

LPRINT/LPRINT USING

Format **LPRINT [<list of expressions>]**
LPRINT USING <format string>;<list of expressions>

Purpose These statements are used to output data to a printer connected to the PX-8.

Remarks These statements are used in the same manner as the PRINT and PRINT USING statements, but output is directed to the printer instead of the display screen.

See also **PRINT, PRINT USING**

Example 1

```
10 A = 3
20 A$ = "There are "
30 LPRINT A$;A;" vowels in `computer'"
40 END

There are  3  vowels in `computer'
```

Example 2

```
10 'THE LPRINT COMMAND
20 A = 3
30 A$ = "There are "
40 LPRINT A$;A;" vowels in 'computer'"
50 LPRINT
60 'THE LPRINT USING "!" COMMAND
70 LPRINT USING "!";"AAA";"BBB";"CCC"
80 LPRINT
90 'THE LPRINT USING "\  \" COMMAND
100 A$ = "123456"
110 B$ = "ABCDEF"
120 LPRINT USING "\ \";A$;B$
130 LPRINT USING "\  \";A$;B$
140 LPRINT
150 'THE LPRINT USING "&" COMMAND
160 LPRINT USING "&";A$;" = ";B$
170 LPRINT
```

```
180 'THE LPRINT USING "#" COMMAND
190 LPRINT USING "####";1;.12;12.6;12345
200 LPRINT
210 LPRINT USING "###.##   ";123;12.34;123.456;.12
220 LPRINT
225 'THE LPRINT USING "+#" COMMAND
230 LPRINT USING "+#### ";123
240 LPRINT
245 'THE LPRINT USING "#-" COMMAND
250 LPRINT USING "####-  ";345;-456
260 LPRINT
265 'THE LPRINT USING "**" COMMAND
270 LPRINT USING "**####.##   ";12.35;123.555;555555.88#
280 LPRINT
290 'THE LPRINT USING "$$" COMMAND
300 LPRINT USING "$$####.##   ";12.35;123.555;555555.88#
310 LPRINT
320 'THE LPRINT USING "$**" COMMAND
330 LPRINT USING "$**####.##   ";12.35;123.555;555555.88#
340 LPRINT
350 'THE LPRINT USING "**$" COMMAND
360 LPRINT USING "**$####.##   ";12.35;123.555;555555.88#
370 LPRINT
380 'THE LPRINT USING "##,.##" COMMAND
390 LPRINT USING "#######,.##";555555.88#
400 LPRINT
405 'THE LPRINT USING "##.##^^^^" COMMAND
410 LPRINT USING "###.##^^^^   ";123.45;12.345;1234.5
415 LPRINT
425 'USING THE UNDERSCORE
435 LPRINT USING "###_%";123
445 LPRINT
455 'USING OTHER CHARAACTERS
465 LPRINT USING "##/##/##";12;34;56
475 LPRINT
485 LPRINT USING "(###)";123
495 LPRINT
505 LPRINT USING "<###>";123
515 LPRINT
```

LSET/RSET

Format

LSET <string variable> = <string expression>
RSET <string variable> = <string expression>

Purpose

These statements move data into a random file buffer to prepare it for storage in a random access file with the PUT statement.

Remarks

<string variable> is a variable which has been assigned to positions in a random file buffer with the FIELD statement. <string expression> is any string constant or string variable.

If the length of <string expression> is less than the number of bytes which were assigned to the specified variable with the FIELD statement, the LSET (Left SET) statement left-justifies the string data in the variable and the RSET (Right SET) statement right-justifies it. The positions following left-justified data and those preceding right-justified data are padded with spaces.

If the length of <string expression> is greater than the number of bytes assigned to the specified variable, excess characters are truncated from the right end of <string expression> when it is moved into the buffer. (This is true for both the LSET and RSET statements.)

Numeric values must be converted to strings before they can be moved into a random file buffer with the LSET or RSET statements. This is done using the MKI$, MKS$, and MKD$ functions described elsewhere in this Chapter, and Chapter 5.

See also

FIELD, GET, OPEN, PUT

Example See Chapter 5 for examples of use of LSET/RSET in a program.

NOTE:
The LSET and RSET statements can also be used to left or right justify a string in a string variable which has not been assigned to a random file buffer. For example, the following program left-justifies character string "CAMERA" in a 20-character field prepared in variable A$ and right-justifies character string "MAY 8, 1984" in variable B$. This procedure can be very useful when formatting data for output.

```
10 A$=STRING$(20, " ")
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Variable A$																				

```
20 B$=A$
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Variable B$																				

```
30 N$="CAMERA":LSET A$=N$
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Variable A$	C	A	M	E	R	A														

Spaces

```
40 N$="MAY 8, 1984":RSET B$=N$
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Variable B$										M	A	Y		8	,		1	9	8	4

Spaces

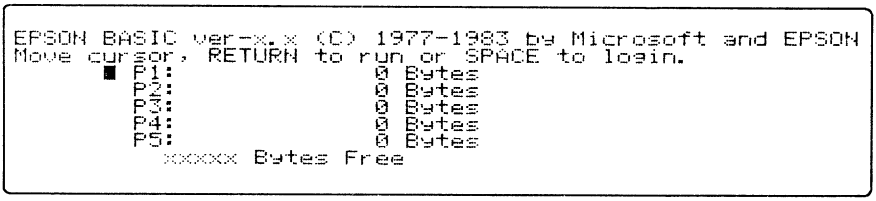
MENU

Format **MENU**

Purpose Returns BASIC to the BASIC program menu.

Remarks Executing this command returns BASIC to the BASIC program menu which is displayed following execution of the BASIC command under CP/M; afterwards, another program area can be selected and logged in with the cursor keys and the space bar or RETURN key. Executing this command also clears all variables and closes any files which are open.
Further, the screen mode and sizes are all reset to their initial values upon execution of this command.

See also **LOGIN**



```
EPSON BASIC ver-x.x (C) 1977-1983 by Microsoft and EPSON
Move cursor, RETURN to run or SPACE to login.
      ■ P1:                0 Bytes
        P2:                0 Bytes
        P3:                0 Bytes
        P4:                0 Bytes
        P5:                0 Bytes
          xxxxx Bytes Free
```

BASIC program menu

MERGE

Format **MERGE <file descriptor>**

Purpose Merges a program from a disk device or the RS-232C interface with the program in the currently logged in program area:

Remarks Specify the device name, file name, and file name extension under which the file was saved in < file descriptor >. The device name can be omitted if the file is in the currently active drive (the drive which was logged in under CP/M at the time BASIC was started). If the file name extension is omitted, ".BAS" is assumed.

The file being merged must have been saved in ASCII format. Otherwise, a "Bad file mode" error will occur.

If any lines of the program being merged have the same numbers as lines of the program in memory, the merged lines will replace the corresponding lines in memory. Thus, a program brought into the BASIC program area with the MERGE statement may be thought of as an overlay which replaces corresponding lines previously included in the program area.

BASIC always returns to the command level following execution of a MERGE command.

See also **SAVE**

Examples First type in and save the following programs, making sure they are saved in ASCII format by using the " , A " extension as shown below.

```
10 'PROGRAM "MERGE1" TO BE MERGED WITH "MERGE2" THEN "MERGE3"
20 PRINT "HELLO MUM"
30 PRINT
Ok
SAVE "A:MERGE1",A
Ok
```

```
NEW
Ok
30 'PROGRAM "MERGE3" TO BE MERGED WITH "MERGE1"
40 PRINT "GOODBYE DAD"
50 END
SAVE "A:MERGE3",A
Ok
```

Clear the program with the NEW command, and type in the following program.

```
50 'PROGRAM "MERGE 2" WITH WHICH "MERGE1" WILL BE MERGED
60 PRINT "GOODBYE MUM"
70 END
```

Now type

```
MERGE "A:MERGE1"
```

If the program in memory is listed it will bě seen to consist of the lines from both programs as follows:

```
10 'PROGRAM "MERGE1" TO BE MERGED WITH "MERGE2" THEN "MERGE3"
20 PRINT "HELLO MUM"
30 PRINT
50 'PROGRAM "MERGE 2" WITH WHICH "MERGE1" WILL BE MERGED
60 PRINT "GOODBYE MUM"
70 END
```

Now type in the following.

```
MERGE "A:MERGE3"
```

List to see that the result is as follows.

```
10 'PROGRAM "MERGE1" TO BE MERGED WITH "MERGE2" THEN "MERGE3"
20 PRINT "HELLO MUM"
30 'PROGRAM "MERGE3" TO BE MERGED WITH "MERGE1"
40 PRINT "GOODBYE DAD"
50 END
60 PRINT "GOODBYE MUM"
70 END
```

MID$

Format

As a statement
MID$ (<string expression 1>,n[,m])= <string expression 2>

As a function
MID$ (X$,J[,K])

Purpose

As a statement, replaces a portion of one string with another. As a function, returns the character string from the middle of string expression X$ which consists of the K characters beginning with the Jth character.

Remarks

When MID$ is used as a statement, n and m are integer expressions and <string expression 1> and <string expression 2> are string expressions. In this case, MID$ replaces the characters beginning at position n in <string expression 1> with the characters of <string expression 2>. If the m option is specified, the first m characters of <string expression 2> are used in making the replacement; otherwise, all of <string expression 2> is used. However, the number of characters replaced cannot exçeed the length of the portion of <string expression 1> which starts with character n.

For example:

```
10 A$="ABCDEFG" : LET B$="wxyz"
20 MID$(A$,3)=B$
30 PRINT A$
```

will give the result "ABwxyzG" for A$. Whereas,

```
10 A$="ABCDEFG" : LET B$="wxyz"
20 MID$(A$,3,2)=B$
30 PRINT A$
```

will give a value of "ABwxEFG".

When MID$ is used as a function, J and K must be integer expressions in the range from 1 to 255. If K is omitted, or there are fewer than K characters to the right of the Jth character,the string returned consists of all characters to the right of the Jth character. If the value specified for J is greater than the number of characters in X$, MID$ returns a null string.

See also **LEFT$, RIGHT$**

Example

```
10 A$ = "Computers  is great!"
20 B$ = "are"
30 PRINT A$
40 MID$(A$,11) = B$
50 PRINT A$
60 PRINT
70 B$ = "super-duper"
80 MID$(A$,15,5) = B$
90 PRINT A$
100 PRINT
110 B$ = MID$(A$,7,12)
120 PRINT B$
130 END
```

```
Computers  is great!
Computers are great!

Computers are super!

ers are supe
```

MKI$/MKS$/MKD$

Format

MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)

Purpose

Converts numeric values to string values for storage in random access files.

Remarks

Numeric values must be converted to string values before they can be placed in a random file buffer by a LSET or RSET statement for storage with a PUT statement. MKI$ converts integers to 2-byte strings, MKS$ converts single precision numbers to 4-byte strings, and MKD$ converts double precision numbers to 8-byte strings.

Unlike the STR$ function, which produces an ASCII string whose characters correspond to the digits of the decimal representation of a number, these functions convert numeric values to characters whose ASCII codes correspond to the binary coded decimal values with which corresponding values are stored in variable memory. In many instances, less disk space is required for storage of numbers which are converted to strings using the MKI$/MKS$/MKD$ functions.

See also

CVI/CVS/CVD

For examples of the use of these functions, see the section of Chapter 6 dealing with random access files.

MOUNT

Format **MOUNT**

Purpose Reads the microcassette tape directory into memory and enables the microcassette drive for access as a disk device.

Remarks A MOUNT command must be executed before data can be written to or read from a microcassette tape. The MOUNT command installs the tape by reading its directory into memory. Until this is done the cassette cannot be used either for reading or writing. The MOUNT command functions in the same way as the MOUNT command on the System Display.

If an attempt is made to MOUNT a tape when a previously installed tape is still in the drive, a "Tape access error" will be generated. If this occurs, insert the previously installed tape back into the PX-8 and execute a REMOVE command. Failure to do this could prevent the data being read from the previous tape.

See also REMOVE and the section on the handling of the Microcassette Drive in the User's Manual.

NAME

Format **NAME <old filename> AS <new filename>**

Purpose Changes the name of a disk device file.

Remarks Both <old filename> and <new filename> are specified as a device name, file name, and extension. The device name may be omitted if the file resides on the disk device which is currently active.

The file name specified in <old filename> must be that of a currently existing file, and that specified in <new filename> must be a name which is not assigned to any other file belonging to the applicable disk device. If <old filename> is not the name of an existing file, a "File not found" error will occur; if <new filename> is already assigned to an existing file, a "File already exists" error will occur. If the file being renamed has a file name extension, that extension must be specified in <old filename>.

If the NAME command is executed with a file which is already open, there is no guarantee that the file will remain intact. Using the CLOSE command will ensure that all files are closed.

This command changes only the name of the specified file; it does not rewrite the file to another area in the storage medium.

See also **FILES**

Example

```
FILES "A:
MERGE1   .BAS   MERGE3   .BAS
Ok
NAME "A:MERGE1.BAS" AS "A:CHAIN.BAS"
Ok
```

```
FILES "A:
CHAIN    .BAS   MERGE3   .BAS
Ok
```

NEW

Format **NEW**

Purpose Deletes the program in the currently logged in program area and clears all variables.

Remarks Enter NEW at the command level to clear the memory before starting to enter a new program. BASIC always returns to the command level upon execution of a NEW command.

An "Illegal function call" error will occur when this command is executed if program editing has been disabled by executing a TITLE command with the P (protect) option specified.

See also **TITLE**

Example

```
LOGIN 1
P1:DEMOPROG      23 Bytes
Ok
NEW
Ok
```

```
LOGIN 1
P1:               0 Bytes
Ok
```

OCT$

Format **OCT$(X)**

Purpose Returns a string which represents the octal value of X.

Remarks The numeric expression specified in the argument is rounded to the nearest integer value before it is evaluated.

A description of using numbers and numeric variables is given in Chapter 2.

Example

```
10 CLS
20 INPUT "What value do you want to convert ";X
30 PRINT
40 PRINT "The octal value of ";X;" is ";OCT$(X)
50 FOR J = 1 TO 3000:NEXT
60 GOTO 10
```

```
What value do you want to convert ? 23
The octal value of  23  is 27
```

```
What value do you want to convert ? 983
The octal value of  983  is 1727
```

ON ERROR GOTO

Format **ON ERROR GOTO [<line number>]**

Purpose Causes program execution to branch to the first line of an error processing routine when an error occurs.

Remarks Execution of the ON ERROR GOTO statement enables error trapping; that is, it causes execution of a program to branch to a user-written error processing routine beginning at the program line specified in <line number> whenever any error (such as a syntax error) occurs. This error processing routine then evaluates the error and/or directs the course of subsequent processing. For example, it may be written to check for a certain type of error or an error occurring in a certain program line, then to resume execution at a certain point in the program depending on the result.

If subsequent error trapping is to be disabled, execute ON ERROR GOTO 0. If this statement is encountered in an error processing routine, program execution stops and BASIC displays the error message for the error which caused the trap. It is recommended that all error processing routines include an ON ERROR GOTO 0 statement for errors for which are not provided for in the error recovery procedures. If <line number> is not specified, the effect is the same as executing ON ERROR GOTO 0.

See also **ERROR, RESUME, ERL/ERR, Appendix A**

Example See the program example under ERROR.

NOTE:
BASIC always displays error messages and terminates execution for errors which occur in the body of an error processing routine; that is, error trapping is not performed within an error processing routine itself.

ON...GOSUB/ON...GOTO

Format **ON <numeric expression> GOSUB <list of line numbers>**
ON <numeric expression> GOTO <list of line numbers>

Purpose Transfers execution to one of several program lines specified in <list of line numbers> depending on the value returned when <numeric expression> is evaluated.

Remarks The value of <numeric expression> determines to which of the line numbers listed execution will branch. If the value of <numeric expression> is 1, execution will branch to the first line number in the list; if it is 2, execution will branch to the second line number in the list; and so forth. If the value is a non-integer, the fractional portion is rounded.

An "Illegal function call" error will occur if the value of <numeric expression> is negative or greater than 256.

With the ON...GOSUB statement, each program line indicated in <list of line numbers> must be a line of a subroutine.

See also **GOSUB...RETURN, GOTO**

Example 1

```
10 CLS
20 INPUT "Type in a number from 5 to 10 ";X
30 Y = X - 4
40 ON Y GOTO 60,80,100,120,140,160
50 END
60 PRINT X;"- 4 = ";Y;" so this is line 60"
70 GOTO 20
80 PRINT X;"- 4 = ";Y;" so this is line 80"
90 GOTO 20
100 PRINT X;"- 4 = ";Y;" so this is line 100"
110 GOTO 20
120 PRINT X;"- 4 = ";Y;" so this is line 120"
130 GOTO 20
140 PRINT X;"- 4 = ";Y;" so this is line 140"
150 GOTO 20
160 PRINT X;"- 4 = ";Y;" so this is line 160"
170 GOTO 20
```

```
Type in a number from 5 to 10 ? 7
 7 - 4 =  3  so this is line 100
Type in a number from 5 to 10 ? 9
 9 - 4 =  5  so this is line 140
```

Example 2

```
10 CLS
20 INPUT "Type in a number from 1 to 5 ";X
30 ON X GOSUB 50,60,70,80,90
40 GOTO 20
50 PRINT "ONE":RETURN
60 PRINT "TWO":RETURN
70 PRINT "THREE":RETURN
80 PRINT "FOUR":RETURN
90 PRINT "FIVE":RETURN
```

```
Type in a number from 1 to 5 ? 2
TWO
Type in a number from 1 to 5 ? 4
FOUR
Type in a number from 1 to 5 ? 3
THREE
```

NOTE:
Only numeric expressions can be used to control branching with the ON...GOSUB and ON...GOTO statements. However, it is possible to derive numeric values from string values by using functions such as ASC and INSTR$. For example, the following sample program derives numeric results for the ON...GOSUB and ON...GOTO statements based on input of string values.

Example 3

```
10 CLS
20 INPUT "Type in a word beginning with A, B or C ";A$
30 X = ASC(A$)
40 Y = X - 64
50 ON Y GOSUB 70,110,150
60 END
70 PRINT "The ASCII code for A is 65, so line 40 subtracts
80 PRINT "64 from this code to give 1, thus causing the"
90 PRINT "subroutine at line 70 to be executed"
100 RETURN
110 PRINT "The ASCII code for B is 66, so line 40 subtracts
120 PRINT "64 from this to give 2, causing the subroutine"
130 PRINT "at line 110 to be executed"
140 RETURN
150 PRINT "The ASCII code for C is 67, and line 40 subtracts
160 PRINT "64 from this giving 3 so that the subroutine on"
170 PRINT "line 150 is executed."
180 RETURN
```

```
Type in a word beginning with A, B or C ? Albatross
The ASCII code for A is 65, so line 40 subtracts
64 from this code to give 1, thus causing the
subroutine at line 70 to be executed
Ok
Type in a word beginning with A, B or C ? Carousel
The ASCII code for C is 67, and line 40 subtracts
64 from this giving 3 so that the subroutine on
line 150 is executed.
Ok
```

OPEN

Format **OPEN "<mode>",[#]<file number>,<file descriptor>, [<record length>]**

Purpose The OPEN statement enables input/output access to a disk device file or other device.

Remarks Disk device files must be OPENed before any data can be input from or output to such files. The OPEN statement allocates a buffer for I/O to the specified file and determines the mode of access in which that buffer will be used. <mode> is a string expression whose first character is one of the following.

O or o	Specifies the sequential output mode.
I or i	Specifies the sequential input mode.
R or r	Specifies the random input/output mode.

Any mode can be specified for a disk device file, but only the "I" or "O" modes can be specified for devices such as the RS-232C interface or printer.

<file number> is an integer expression from 1 to 15 which specifies the number by which the file is to be referenced in I/O statements as long as the file is open. The value of <file number> is limited to the maximum specified in the /F: option if this option is used when BASIC is started up. Since this is 3 in the default mode, it is necessary to ensure the /F: option is specified before a program is run if more than 3 files are required.

<file descriptor> is a string expression which conforms to the rules for naming files (see Chapter 2).

<record length> is an integer expression which, if specified, sets the record length for random access files. If not specified, the record length is set to 128 bytes.

Disk files and files on the RAM disk can be open for sequential input or random access under more than one file number at a time.

However, a given sequential access file can only be opened for sequential output under one file number at a time, and such a file cannot be open in both the sequential input and sequential output modes concurrently.

Microcassette files can only be open for sequential input, random access, or sequential output under one file number at a time. Further, only one microcassette file can be open at any given time.

The RS-232C interface can be open concurrently in the sequential input mode and the sequential output mode, but cannot be opened in the random access mode.

See also Chapters 5 and 6.

Example For programming examples, see Chapters 5 and 6 and the explanation of EOF.

OPTION BASE

Format **OPTION BASE <base number>**

Purpose Declares the minimum value of array subscripts.

Remarks When BASIC is started, the minimum value of array subscripts is set to 0; however, in certain applications it may be more convenient to use variable arrays whose subscripts have a minimum value of 1. Specifying 1 for the value of <base number> in this statement makes it possible to set the minimum subscript base to one.

Once the subscript base has been set by executing this statement, it cannot be reset until a CLEAR statement has been executed; executing a CLEAR statement restores the option base to 0. Further, OPTION BASE 1 cannot be executed if any values have previously been stored in any array variables. A "Duplicate Definition" error will occur if the OPTION BASE statement is executed under either of these conditions.

See also **DIM**

Example

```
10 CLEAR
20 PRINT "Memory free following CLEAR:";FRE(0)
30 OPTION BASE 0
40 DIM A(5,5,5,5)
50 PRINT "Memory free after DIM A(5,5,5,5) with OPTION BASE
0:";FRE(0)
60 CLEAR
70 OPTION BASE 1
80 DIM A(5,5,5,5)
90 PRINT "Memory free after DIM A(5,5,5,5) with OPTION BASE
1:";FRE(0)

run
Memory free following CLEAR: 7324
Memory free after DIM A(5,5,5,5) with OPTION BASE 0: 2125
Memory free after DIM A(5,5,5,5) with OPTION BASE 1: 4809
Ok
```

OPTION COUNTRY

Format **OPTION COUNTRY <character string>**

Purpose Selects one of the international character code sets.

Remarks Executing this statement selects the character set of the country which corresponds to the first letter of <character string>. Characters corresponding to character sets of the various countries are as follows.

"D" or "d"	Denmark
"E" or "e"	England
"F" or "f"	France
"G" or "g"	Germany
"I" or "i"	Italy
"N" or "n"	Norway
"S" or "s"	Spain
"U" or "u"	U.S.A.
"W" or "w"	Sweden

Executing the OPTION COUNTRY statement changes the character set which is used for output to the LCD screen. Further, the currency symbol output by the PRINT USING statement is changed to that of the specified country.

The option selected will remain in force until an exit is made from BASIC or until changed with another similar statement. When exiting to the system, or re-running BASIC from the MENU screen, the character set reverts to the default set by the DIP switches. If it is required to have a permanent change of key assignment, the DIP switches can be set as described in the User's Manual.

Example

```
10 CLS
20 INPUT "Type D, E, F, G, I, N, S, U or W ";A$
30 PRINT "Current special characters"
40 FOR J = 1 TO 12
50 READ A
60 PRINT CHR$(A);
70 NEXT J
80 RESTORE
90 PRINT:PRINT "Newly selected special characters"
100 OPTION COUNTRY A$
110 FOR K = 1 TO 12
120 READ B
130 PRINT CHR$(B);
140 NEXT K
145 OPTION COUNTRY "e"
150 END
160 DATA 35,36,64,91,92,93,94,96,123,124,125,126
```

```
Type D, E, F, G, I, N, S, U or W ? G


Current special characters
£$@[\]^`{¦}~
Newly selected special characters
#$§ÄÖÜ^`äöüß
```

```
Type D, E, F, G, I, N, S, U or W ? S

Current special characters
£$@[\]^`{¦}~
Newly selected special characters
₧$@¡Ñ¿^`¨ñ}~
```

```
Type D, E, F, G, I, N, S, U or W ? I


Current special characters
£$@[\]^`{¦}~
Newly selected special characters
#$@°\é^ùàòèì
```

OPTION CURRENCY

Format **OPTION CURRENCY <string expression>**

Purpose Changes the currency symbol.

Remarks This statement changes the character which is output as the currency symbol by the format string of the PRINT USING statement to that specified in <string expression> when the format string begins with two successive characters whose ASCII code is 36 (&H24 in hexadecimal notation). The first character of <string expression> may be any character which is included in the character set.

For example, executing OPTION CURRENCY "@" while the U.S. character set is selected causes the symbol "@" to be output as the currency symbol by PRINT USING statements whose format strings begin with "$$". If the German character set is subsequently selected, the "§" symbol is output as the currency symbol.

NOTE:
The formatting characters used in PRINT USING statements are based on the U.S. ASCII keyboard. With the character sets of other countries, use characters with the corresponding internal codes as shown below.

```
10 OPTION COUNTRY "E"
20 OPTION CURRENCY "#"
30 PRINT USING "$$####";100
40 OPTION COUNTRY "S"
50 PRINT USING "$$####";100
60 OPTION COUNTRY "E"
```

```
run
  £100
  ₧100
Ok
```

OUT

Format **OUT <integer expression 1>,<integer expression 2>**

Purpose Used to send data to a machine output port.

Remarks The data to be output is specified in <integer expression 2> and the port to which it is to be output is specified in <integer expression 1>. Both values must be in the range 0 to 255.

See also **INP**

NOTE:
Use of this statement requires sound knowledge of the PX-8 firmware. Incorrect use may corrupt programs or data held in memory, including BASIC itself.

PCOPY

Format **PCOPY <program area no.>**

Purpose Copies the contents of the currently selected program area to another program area.

Remarks This command copies the contents of the currently logged in program area to the program area whose number is specified in <program area no.>. The number specified must be in the range from 1 to 5, and must be the number of an area which is empty.

Programs which have been saved using the protect save function and then loaded into memory cannot be copied from one program area to another with this command.

An "Illegal function call" error will occur if a number other than 1 to 5 is specified in <program area no.>, if the number of the currently selected program area is specified, or if the specified program area is not empty. This error also occurs if the program in the currently selected program area is one which has been loaded from a program previously saved with the protect save function.

See also **LOGIN, MENU, STAT, TITLE**

Example

```
10 'THIS PROGRAM WILL BE COPIED FROM AREA 1 TO AREA 3
20 PRINT "COPY-CAT!"
30 END
```

```
EPSON BASIC ver-1.0 (C) 1977-1983 by Microsoft and EPSON
19276 Bytes Free
P1:COPY-CAT      81 Bytes
Ok
```

```
EPSON BASIC ver-1.0 (C) 1977-1983 by Microsoft and EPSON
Move cursor, RETURN to run or SPACE to login.
   ■ P1:COPY-CAT     81 Bytes
     P2:              0 Bytes
     P3:              0 Bytes
     P4:              0 Bytes
     P5:              0 Bytes
        19276 Bytes Free
```

```
EPSON BASIC ver-1.0 (C) 1977-1983 by Microsoft and EPSON
19276 Bytes Free
P1:COPY-CAT     81 Bytes
Ok
PCOPY 3
Ok
```

```
EPSON BASIC ver-1.0 (C) 1977-1983 by Microsoft and EPSON
Move cursor, RETURN to run or SPACE to login.
   ■ P1:COPY-CAT     81 Bytes
     P2:              0 Bytes
     P3:             81 Bytes
     P4:              0 Bytes
     P5:              0 Bytes
        19195 Bytes Free
```

```
EPSON BASIC ver-1.0 (C) 1977-1983 by Micros   t and EPSON
19195 Bytes Free
P3:             81 Bytes
Ok
```

```
P3:             81 Bytes
Ok
RUN
COPY-CAT!
Ok
```

PEEK

Format PEEK(J)

Purpose Returns one byte of data from the memory address specified for J as an integer from 0 to 255.

Remarks As the name suggests, PEEK is a function to look at memory locations and return the value of the contents of the location PEEKed. The contents of the location are not changed by inspecting it. For the beginner learning BASIC, PEEK and the allied command POKE (which allows the contents of a location to be changed) are commands which are difficult to understand, because it is not always easy to see the function of the values used. They can be used in a large number of ways. Also the values are very computer dependent. It is often possible to type many BASIC programs into the PX-8 when they have been written for other computers even if the BASIC is another version of MICROSOFT BASIC. When PEEK and POKE commands are used, they are invariably not directly translatable. For example with many computers it is possible to PEEK and POKE the memory reserved for the screen. This is not possible directly with the PX-8.

The integer value specified for J must be in the range from 0 to 65535.

See also **POKE**

Example If location 4 is PEEKed, the number returned will correspond to the drive which is the default drive when returning to CP/M. This is not the default drive for BASIC. If the value returned is "0" then A: is the default drive, if it is "1" then it is drive "B" and so on.

POINT

Format **POINT (horizontal position, vertical position)**

Purpose Returns the setting of the display dot at the specified graphic screen coordinates.

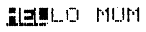
Remarks This function returns a number indicating the setting of the display dot at the specified graphic screen coordinates.
If the dot is set, 7 is returned; if it is reset, 0 is returned.

If the coordinates specified are outside the graphic screen area the value of −1 is returned.

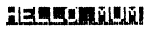
Example

```
10 CLS
20 SCREEN 3
30 PRINT "HELLO MUM"
40 FOR X = 0 TO 54
50 FOR Y = 0 TO 8
60 IF POINT (X,Y) = 0 THEN PRESET (X,Y),1:GOTO 80
70 PRESET (X,Y),0
80 NEXT Y,X
```

```
HELLO MUM
```



```
HELLO MUM
```



```
HELLO MUM
```

POKE

Format **POKE <integer expression 1>,<integer expression 2>**

Purpose Writes a byte of data into memory.

Remarks The address into which the data byte is to be written is specified in <integer expression 1> and the value which is to be written into that address is specified in <integer expression 2>. The value specified in <integer expression 2> must be in the range from 0 to 255.

The complement of the POKE statement is the PEEK function, which is used to check the contents of specific addresses in memory. Used together, the POKE statement and PEEK function are useful for accessing memory for data storage, writing machine language programs into memory, and passing arguments and results between BASIC programs and machine language routines.

See also **PEEK**

Example An example of using BASIC to POKE a machine code routine is described under the CALL command.

In the example under the PEEK command, it was shown how to find out which drive would be the active drive when exiting to CP/M. This can be altered with the BASIC POKE command. If you have any BASIC programs in memory which you want to save, save them first. In direct mode type "POKE 4,2", then type "SYSTEM" and press RETURN. You will be transferred to either the system menu or the CP/M command line. If the menu is active, use ESC to return to the CP/M command line. The active drive should be shown as "C>" which will normally be assigned to one of the ROM sockets.

WARNING:
Since this statement changes the contents of memory, the work area used by BASIC may be destroyed if it is used carelessly. This can result in erroneous operation, so be sure to check the memory map to confirm that the address specified is in a usable area.

POS

Format **POS(<file no.>)**

Purpose Returns the current position of the print head, cursor, or file output buffer pointer.

Remarks When "0" is specified for <file no.> this function returns the current horizontal position of the cursor in the LCD screen. The value returned ranges from 1 to the number of columns in the currently selected virtual screen.

When a number other than "0" is specified for <file no.>, this function returns the current position of the buffer pointer in the output buffer for the specified file. The file must be one which has been opened in the sequential output mode or the random access mode; the value returned for a file opened in the sequential input mode is meaningless.

When the value specified for <file no.> is other than "0", the value returned by the POS function is a number in the range from 1 to 255. The value returned immediately after the file is opened or a carriage return is output is "1".

If the file specified is "LPTØ", this function returns the same value as the LPOS function.

Example

```
10 PRINT "1234567890";:GOSUB 50
20 PRINT "1234567890123";:GOSUB 50
30 PRINT "1234567890123456";:GOSUB 50
40 END
50 A = POS(X)::PRINT"Horizontal cursor position is ";A
60 RETURN

Ok
run
1234567890Horizontal cursor position is  11
1234567890123Horizontal cursor position is  14
1234567890123456Horizontal cursor position is  17
Ok
```

POWER

Format

POWER OFF[,RESUME]
POWER <duration>
POWER CONT

Purpose

Allows the power to be turned off by a program and can be used to set the auto power-off function.

Remarks

Executing POWER OFF turns off the power in the restart mode. This is equivalent to switching the power off in the restart mode. On turning the power on again, command will be returned to CP/M, and to the system menu if it is set to be on. Any ALARM or wake settings will still function as if the power had been switched off normally.

When POWER OFF, RESUME is executed, the power goes off in the continue mode. This is equivalent to switching the power switch off when the control key is held down as well. When the power is then turned back on (by moving the power switch from ON to OFF, and then back ON again), BASIC program execution resumes with the statement following the POWER statement which turned off the power. If the wake time set by the ALARM statement is reached after the power has been switched off by the program, it causes execution to resume at that time with the statement following the POWER statement which turned off the power.

POWER <duration> specifies the amount of time which will elapse before the auto shut-off function automatically turns off the power. <duration> is specified in minutes as an integer expression in the range from 1 to 255. The auto shut-off function automatically turns off the power when the amount of time which has elapsed since the last key was pressed or the last program statement was executed becomes equal to the time specified in <duration>.

Executing POWER CONT disables the auto power-off function. The auto power-off function can later be re-enabled by executing POWER <duration>.

See also **ALARM, ALARM$, AUTO START**

Example The use of POWER to switch the computer off automatically is shown under ALARM. The combination of the two functions allows the PX-8 to switch itself on and off at any predetermined time, totally under the control of a BASIC program.

PRESET

Format **PRESET [STEP](X,Y)[,<function code>]**

Purpose Resets the dot at the graphic display coordinates specified by X and Y.

Remarks Relative coordinates are used when STEP is included; otherwise, absolute coordinates are used.

When <function code> is omitted or is specified as 0, the dot is reset (turned off); if it is specified as a number from 1 to 7, the point is set (turned on).

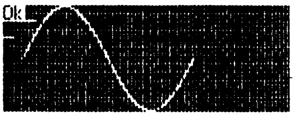
The LCD screen must be in the graphic mode (mode 3) when this statement is executed.

After execution of the PRESET statement, the last reference pointer (LRP) is updated to the coordinates specified for X and Y.

See also **LINE, PSET**

Example

```
10 SCREEN 3
20 LINE (0,0)-(150,50),,BF
30 PI = 3.14159
40 D = PI/180
50 FOR N = 1 TO 360
60 X = 10 + N/4
70 Y = 25-SIN(D*N)*25
80 PRESET (X,Y)
90 NEXT
```



NOTE:
Normally PSET is used without a <function code> to turn a point on, and PRESET is used without a <function code> to turn a point off.
The default value of <function code> is 7 with PSET. With PRESET, the default value of <function code> is 0.
If <function code> is specified, the two commands behave in the same way.

PRINT

Format **PRINT [<list of expressions>]**

Purpose Outputs data to the LCD screen.

Remarks Executing a PRINT statement without specifying any expressions advances the cursor to the line following that on which it is currently located without displaying anything.

When a <list of expressions> is included, the values of the expressions are output to the display screen. Both numeric and string expressions may be included in the list. The positions in which items are displayed is determined by the delimiting punctuation used to separate items in the list.

Under BASIC, the screen is divided up into zones consisting or 14 spaces each. When items in <list of expressions> are delimited with commas, each succeeding value is displayed starting at the beginning of the following zone. When items are delimited with semicolons, they are displayed immediately following one another. Including one or more spaces between items has the same effect as a semicolon. Other display formats can be obtained by including the TAB, SPACE$, and SPC functions.

If a semicolon or comma is included at the end of the list of expressions, the cursor remains on the current display line and values specified in the next PRINT statement are displayed starting on the same line. If the list of expressions is concluded without a semicolon or comma, the cursor is moved to the beginning of the next line.

If values displayed by the PRINT statement will not fit on one display line, display is continued on the next line.

See also **LPRINT, PRINT USING, SPACE$, SPC, TAB**

Example

```
10 PRINT 123;456;789
20 PRINT 123,456,789
30 PRINT "123";"456";"789"
40 PRINT "ABC",
50 PRINT "DEF"
60 PRINT "ABC";
70 PRINT "DEF"

run
 123   456   789
 123             456                 789
123456789
ABC             DEF
ABCDEF
Ok
```

NOTE:
A question mark may be typed in place of the word PRINT when entering the PRINT statement. BASIC automatically converts question marks encountered during statement execution to PRINT statements.

PRINT USING

Format **PRINT USING <format string>;<list of expressions>**

Purpose Displays string data or numbers using a format specified by <format string>.

Remarks <format string> consists of special characters which determine the size and format of the field in which expressions are displayed. <list of expressions> consists of the string expressions or numeric expressions which are to be displayed. Each expression in <list of expressions> must be delimited from the one following it by a semicolon.

The characters which make up the <format string> differ according to whether the expressions included in <list of expressions> are string expressions or numeric expressions. The characters and their functions are as follows:

Format strings for string expressions

"!"
Specifies that the first character of each string included in <list of expressions> is to be displayed in a 1-character field.

Example

```
10 PRINT USING "!";"Aa";"bB";"Dc"

run
AbD
Ok
```

"\n spaces\"
Specifies that 2+n characters of each string in <list of expressions> is to be displayed. Two characters will be displayed if no spaces are included between the backslashes, three characters will be displayed if one space is included between the backslashes, and so on. Extra characters are ignored if the length of any string in <list of expressions> is greater than 2+n. If the length of the field is greater than that of a string, the string is left-justified in the field and padded on the right with spaces.

Example

```
10 A$="1234567"
20 B$="ABCDEFG"
30 PRINT USING "\   \";A$;B$
40 PRINT USING "\         \";A$;B$

run
12345ABCDE
1234567    ABCDEFG
Ok
```

"&"

Specifies that strings included in <list of expressions> are to be displayed exactly as they are.

Example

```
10 READ A$,B$
20 PRINT USING "&";A$;" ";B$
30 DATA EPSON,PX-8

run
EPSON PX-8
Ok
```

Format strings for numeric expressions

With numeric expressions, the field in which digits are displayed by a PRINT USING statement is determined by a format string consisting of the number sign (#) and a number of other characters. When the format string consists entirely of # signs, the length of the field is determined by that of the format string.

If the number of digits in numbers being displayed is smaller than the number of positions in the field, numbers are right justified in the field. If the number of digits is greater than the number of # signs, a percent sign (%) is displayed in front of the number and all digits are displayed.

Minus signs are displayed in front of negative numbers, but (ordinarily) positive numbers are not preceded by a plus sign.

The following is an example of use of the # sign in the numeric format string of a PRINT USING statement.

Example

```
10 PRINT USING "####   ";1;.12;12.6;12345
20 END

run
   1      0     13   %12345
Ok
```

Other special characters which may be included in numeric format strings are as follows:

"."
A decimal point may be included at any point in the format string to indicate the number of positions in the field which are to be used for display of decimal fractions. The position to the left of the decimal point in the field is always filled (with 0 if necessary). Digits to the right of the decimal point are rounded to fit into positions to the right of the decimal point in the field.

```
10 PRINT USING "###.##   ";123;12.34;123.456;.12
20 END

run
123.00   12.34   123.46    0.12
Ok
```

"+"
A plus sign (+) at the beginning or end of the format string causes the sign of the number (plus or minus) to be displayed in front of or behind the number.

Example

```
10 PRINT USING "+####";123;-123
20 END

run
 +123 -123
Ok
```

"_"

A minus sign at the end of the format string causes negative numbers to be displayed with a trailing minus sign.

Example

```
10 PRINT USING "####- ";345;-456
20 END

run
 345   456-
Ok
```

"**"

A double asterisk at the beginning of the format string causes leading spaces to be filled with asterisks. The asterisks in the format string also represent two positions in the display field.

Example

```
10 PRINT USING "******.##   ";
12.35;123.555;555555.88#
20 END

run
****12.35  ***123.56  555555.88
Ok
```

"$$"

A double dollar sign at the beginning of the format string causes the dollar sign (or other character selected with the OPTION CURRENCY statement) to be displayed immediately to the left of numbers displayed. The dollar signs in the format string also represent two positions in the display field (one of which is used for display of the dollar sign).

Example

```
10 PRINT USING "$$####.##   ";
12.35;123.555;555555.88#
20 END

run
   $12.35    $123.56  %$555555.88
Ok
```

"**$"

Specifying **$ at the beginning of the format string combines the effect of the dollar sign and asterisk. Numbers displayed are preceded by a dollar sign, and empty spaces to the left of the dollar sign are filled with asterisks. The symbols **$ also represent three positions in the display field (one of which is used for display of the dollar sign).

Example

```
10 PRINT USING "**$#####.##  ";12.35;
123.555;555555.88#
20 END

run
****$12.35  ***$123.56  $555555.88
Ok
```

","

Including a comma to the left of the decimal point in a format string causes commas to be displayed to the left of every third digit to the left of the decimal point. If the format string does not include a decimal point, include the comma at the end of the format string; in this case, numbers are rounded to the nearest integer value for display.

The comma represents the position of an additional position in the display field, and each comma displayed occupies one position.

Example

```
10 PRINT USING "#######,.##";555555.88#
20 END

run
 555,555.88
Ok
```

"^^^^"
Four carets (exponentiation operators) at the right end of the format string cause numbers to be displayed in exponential format. The four carets reserve space for display of E+XX.

The decimal point may also be included in the format string at any position desired. Significant digits are left-justified, and the exponent and fixed point constant are adjusted as necessary to allow the number to be displayed in the number of positions in the field.

Unless a leading + or trailing + or − sign is included in the format string, one digit position to the left of the decimal point will be used to display a + or − sign.

Example

```
10 PRINT USING "###.##^^^^  ";
   123.45;12.345;1234.5
20 END
```

```
run
 12.35E+01    12.35E+00    12.35E+02
Ok
```

"_"

An underscore mark in the format string causes the following character to be output as a literal together with the number.

Example

```
10 PRINT USING "###_%";123
20 END
```

```
run
123%
Ok
```

Other characters:
If characters other than those described above are placed at the beginning or end of a format string, those characters will be displayed in front of or behind the formatted number. Operation varies from case to case if other characters are included within the format string; however, in general including other characters in the string has the effect of dividing the string up into sections, with formatted numbers displayed in each section together with the delimiting character.

```
10 PRINT USING "##/##/##";12;34,56
20 PRINT USING "(###)";123
30 PRINT USING "<###>";123
40 END
```

```
Ok
run
12/34/56
(123)
<123>
Ok
```

NOTE:
The formatting characters shown above apply to the ASCII character set. If you select a character set other than ASCII with the Option Country statement some of the formatting characters will be output differently as shown below.

Hex.	Dec.	U.S.A	France	Germany	England	Denmark	Sweden	Italy	Spain	Norway
23H	35	#	#	#	£	#	#	#	₧	#
24H	36	$	$	$	$	$	¤	$	$	¤
5CH	92	\	ç	Ö	\	Ø	Ö	\	Ñ	Ø
5EH	94	^	^	^	^	Ü	Ü	^	^	Ü

Example See listing under **OPTION CURRENCY.**

PRINT#/PRINT# USING

Format **PRINT# <file number>,[<list of expressions>]**

PRINT# <file number>, USING <format string>;<list of expressions>

Purpose These statements write data to a sequential output file.

Remarks The value of <file number> is the number under which the file was opened for output. The specification of <format string> is the same as that described in the explanation of the PRINT USING statement, and the expressions included in <list of expressions> are the numeric expressions which are to be written to the file.

Both of the formats above write values to the disk in display image format; that is, data is written to the disk in exactly the same format as it is displayed on the screen with the PRINT or PRINT USING statements. Therefore, care must be taken to ensure that data is properly delimited when it is written to the file (otherwise, it will not be input properly when the file is read later with the INPUT# or LINE INPUT# statements).

Numeric expressions included in <list of expressions> should be delimited with semicolons. If commas are used, the extra blanks that would be inserted between display fields by a PRINT statement will be written to the disk.

String expressions included in <list of expressions> must be delimited with semicolons; further, a string expression consisting of an explicit delimiter (a comma or carriage return code) should be included between each expression which is to be read back into a separate variable. The reason for this is that the INPUT# statement regards all characters preceding a comma or carriage return as one item. Explicit delimiters can be included using one of the using one of the following formats.

PRINT#1,<string expression>;",";<string expression>...

PRINT # 1, < string expression > ;CHR$(13); < string expression > ...

If a string which is to be read back into a variable with the INPUT # statement includes commas, significant leading spaces or carriage returns, the corresponding expression in the PRINT # statement must be enclosed between explicit quotation marks CHR$(34). This is done as follows.

PRINT # 1,CHR$(34);"SMITH, JOHN";CHR$(34);CHR$(34); "SMITH, ROBERT";CHR$(34);...

This would actually be printed to the disk as

"SMITH, JOHN","SMITH, ROBERT".....

When the LINE INPUT # statement is to be used to read items of data back into variables, delimit string expressions in < list of expressions > with CHR$(13) (the carriage return code) as shown in the example above.

See also **INPUT #, LINE INPUT #, WRITE #**, and Chapter 6.

PSET

Format **PSET [STEP] (X, Y) [, <function code>]**

Purpose Sets or resets the dot at the specified graphic coordinates on the screen.

Remarks Relative coordinates are used when STEP is included; otherwise, absolute coordinates are used.
When <function> code is omitted or is specified as a number from 1 to 7, the dot is set (turned on); if 0 is specified, it is reset (turned off). The LCD screen must be in the graphic mode (mode 3) when this statement is executed.

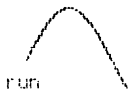
After execution of the PSET statement, the last reference pointer (LRP) is updated to the coordinates specified for X and Y.

NOTE:
Normally PSET is used without a <function code> to turn a point on, and PRESET is used without a <function code> to turn a point off.
The default value of <function code> is 7 with PSET. With PRESET, the default value of <function code> is 0.
If <function code> is specified, the two commands behave in the same way.

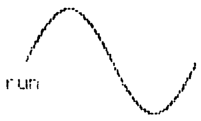
See also **LINE, PRESET**

Example

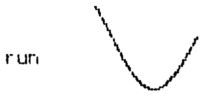
```
10 SCREEN 3
20 PI = 3.14159
30 D = PI/180
40 FOR P = 1 TO 0 STEP -1
50 FOR N = 1 TO 360
60 X = 10 + N/4
70 Y = 25-SIN(D*N)*25
80 PSET (X,Y),P
90 NEXT N,P
```



```
run
```



```
run
```



```
run
```

PUT

Format **PUT[#]<file number>[,<record number>]**

Purpose Writes the contents of a random file buffer to one record of a random access file.

Remarks The random access file must be opened in the "R" mode under the number specified in <file number> before this statement can be executed. Further, data must be set in the random file buffer with the LSET and/or RSET statements.

<file number> is the number under which the file was opened, and <record number> is the number of the file record to which the buffer contents are to be written.

<record number> must have a value in the range from 1 to 32767. If <record number> is omitted, the contents of the random file buffer will be written to the record following the record accessed by the previous PUT or GET statement.

See also **GET,LSET/RSET, OPEN** and Chapter 6.

Example

```
10 OPEN "R",#1,"RANDOM",20
20 FIELD #1,10 AS A$
30 LSET A$="ABCDEFGHIJ"
40 PUT #1
50 WRITE#1,1,2,"YZa"
60 PUT #1
70 PRINT#1,"KLMNOPQ";
80 PUT #1
90 FOR I=1 TO 3
100 GET #1,I
110 PRINT A$
120 NEXT

run
ABCDEFGHIJ
1,2,"YZa"
KLMNOPQa"
Ok
```

NOTE:
String data to be written to a random access file with PUT can be placed in the random file buffer with the PRINT#, PRINT# USING, and WRITE# statements, as well as with the LSET/RSET statements. An example of this is included in the program above. When the WRITE# statement is used for this purpose, extra positions in the buffer are padded with spaces. A "Field overflow" error will occur if an attempt is made to read or write past the end of the buffer.

RANDOMIZE

Format **RANDOMIZE [<expression>]**

Purpose Reinitializes the sequence of random numbers generated by the RND function using the seed number specified in <expression>.

Remarks The value specified in <expression> must be a number in the range from −32768 to 32767. If <expression> is omitted, BASIC suspends program execution and displays the following message to prompt the operator to enter a value from the keyboard.

Random number seed (−32768 to 32767)?

If the random number sequence is not reinitialized, the RND function will return the same sequence of random numbers each time a given program is executed. This can be overcome by placing a RANDOMIZE command at the beginning of each program so that the user can input a random number. It is better however, if the computer changes the random number seed in <expression>. This can be achieved by using the TIME$ function as this is a continually changing string of numbers. A description of this is given in the example program in RND.

See also RND

Example

```
10 FOR J=1 TO 3
20 RANDOMIZE
30 PRINT
40 FOR I=1 TO 5
50 PRINT RND;
60 NEXT I
70 PRINT
80 NEXT J

run
Random number seed (-32768 to 32767)? 1
 .58041  .128928  .928324  .901162  .532818
Random number seed (-32768 to 32767)? 2
 .89341  .823736  .964563  .674916  .963391
Random number seed (-32768 to 32767)? 1
 .826124  .915422  .0593067  .381003  .511101
Ok
```

READ

Format **READ <list of variables>**

Purpose Reads values from DATA statements and assigns them to variables.

Remarks The READ statement must always be used in conjunction with one or more DATA statements. The READ statement assigns items from the <list of constants> of DATA statements to variables specified in <list of variables>. Items from the <list of constants> are substituted into variables in the <list of variables> on a one-to-one basis, and the type of each variable to which data is assigned must be the same as the type of the corresponding constant in <list of constants>.

A single READ statement may access one or more DATA statements, or several READ statements may access the same DATA statement.

If the number of variables specified in <list of variables> is greater than the number of constants specified by DATA statements, an "Out of data" error will occur. If the number of variables specified in <list of variables> is smaller than the number of constants specified in DATA statements, subsequent READ statements begin reading data at the first item which has not previously been read. If there are no subsequent READ statements, the extra items are ignored.

The next item to be read by a READ statement can be reset to the beginning of the first DATA statement on any specified line by means of the RESTORE statement.

See also **DATA, RESTORE**

Example

```
10 FOR J=1 TO 5
20 READ A(J),B$(J),C(J)
30 NEXT J
40 FOR J=1 TO 5
50 PRINT A(J),B$(J),C(J)
60 NEXT J
70 END
80 DATA 1,aaaa,11
90 DATA 2,bbbb,22
100 DATA 3,cccc,33
110 DATA 4,dddd,44
120 DATA 5,eeee,55

run
 1              aaaa            11
 2              bbbb            22
 3              cccc            33
 4              dddd            44
 5              eeee            55
Ok
```

REM

Format **REM <remark>**
' <remark>

Purpose Makes it possible to insert explanatory remarks into programs.

Remarks Either of the above formats may be used to insert explanatory remarks into a program. Remark statements are ignored by BASIC during program execution, but are output exactly as entered when the program is listed.

If program execution is branched to a line which begins with a remark statement, execution resumes with the first subsequent line which contains an executable statement.

If a remark statement is to be appended to a line which includes an executable statement, be sure to precede it with a colon (:). Also, note that any executable statements following a remark statement on a given program line will be ignored.

NOTE:
When a program is listed using LIST or LLIST*, apostrophes indicating remark statements are not output. However, "REM" is output at the beginning of any remark statements beginning with REM.*

REMOVE

Format **REMOVE**

Purpose Writes the directory to the microcassette tape and terminates microcassette read and write access.

Remarks A REMOVE command must be executed before taking a cassette tape out of the microcassette drive. The reason for this is that if the cassette tape is taken out of the drive without executing the REMOVE command, data which has been written to the cassette tape up to that point may be lost. Further, if another tape is inserted in the microcassette drive without executing a REMOVE command for the previous tape, the contents of the new tape may be destroyed.

Before executing the REMOVE command, the tape must have first been installed by executing the MOUNT command.

See also **MOUNT**

Example **REMOVE**

RENUM

Format **RENUM[[<new line number>][,[<old line number>] [,<increment>]]]**

Purpose Renumbers the lines of programs.

Remarks This command renumbers the lines of a program according to the values specified in <new line number>, <old line number>, and <increment>. <new line number> is the first line number to be used in the new sequence of program lines and <old line number> is the line number in the current program with which renumbering is to begin. <increment> is the amount by which each successive line number in the new sequence is to be increased over the number of the preceding line.

The default value for <new line number> is 10, that for <old line number> is the first line of the current program, and that for <increment> is 10.

The RENUM command also changes all line number references included in GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the message "Undefined line xxxxx in yyyyy" is displayed. The incorrect line number indicated by xxxxx is not changed, but that indicated by yyyyy may be changed.

Examples **RENUM**
Renumbers the entire program. The first line number of the new sequence will be 10, and the numbers of subsequent lines will be increased in increments of 10.

RENUM 300,50
Renumbers program lines starting with existing line 50. The number of line number 50 in the current program will be changed to 300, and all subsequent lines will be increased in increments of 10.

RENUM 1000,900,20
Renumbers the lines beginning with 900 so that they start with line number 1000 and are increased in increments of 20.

NOTE:
The RENUM command 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 if this rule is not observed.

RESET

Format **RESET**

Purpose Resets the READ/ONLY condition which results when the floppy disk in a disk drive has been exchanged for another one. When the floppy disk in an external disk drive is replaced with another one when that drive has previously been accessed, subsequent writes to that drive are inhibited. This is to protect the contents of the disk's directory. Executing the RESET command resets the read-only condition and re-enables access to the new disk. It also closes any files which are open in the same manner as the CLOSE command.

Also RESET enables a new ROM capsule for read access after replacement.

The default drive is the default drive for CP/M until a RESET command is executed. The execution of a RESET command sets the default drive to drive A: and so programs should specify the drive to which the data is to be saved.

Example

```
100 CLOSE
110 PRINT "Replace disk in drive E: and press
    enter when ready"
120 A$=INPUT$(1)
130 RESET
140 OPEN"O",#1,"E:FILE1"
```

RESTORE

Format **RESTORE [<line number>]**

Purpose Allows DATA statements to be re-read from a specified program line.

Remarks If <line number> is specified, the next READ statement will access the first item in the DATA statement on the specified line or on the first subsequent line which contains a DATA statement if there is no DATA statement in the specified line. If <line number> is not specified, the next READ statement accesses the first item in the first DATA statement in the program.

See also **DATA, READ**

Example

```
10 FOR I=1 TO 2
20 FOR X=1 TO 8
30 READ A:SOUND A,50
40 NEXT X,I
50 SOUND 0,100:RESTORE
60 FOR I=1 TO 2
70 FOR X=1 TO 8
80 READ A:SOUND A*2,50
90 NEXT X,I
100 DATA 256,288,320,341,384,426,480,512
110 DATA 512,480,426,384,341,320,288,256
```

RESUME

Format

RESUME
RESUME Ø
RESUME NEXT
RESUME <line number>

Purpose

Used to continue program execution after execution has branched to an error processing routine.

Remarks

The RESUME statement makes it possible to resume program execution at a specific line or statement after error recovery processing has been completed. The point at which execution is resumed is determined by the format in which the statement is executed as follows:

RESUME or **RESUME Ø**	Resumes program execution at the statement which caused the error.
RESUME NEXT	Resumes program execution at the statement immediately following that which caused the error.
RESUME <line number>	Resumes program execution at the program line specified in <line number>.

A "RESUME without error" message will be generated if a RESUME statement is encountered anywhere in a program except in an error processing routine.

See also

ERROR, ON ERROR GOTO, ERR/ERL

Example

See the example program under ERROR.

RIGHT$

Format **RIGHT$(X$,J)**

Purpose Returns a string composed of the J characters making up the right hand end of string X$.

Remarks The value specified for J must be in the range from 0 to 255. If J is greater than the length of X$ the entire string will be returned. If J is equal to 0 a null string of zero length is returned.

See also **LEFT$, MID$**

Example

```
10 A$="Epson PX-8"
20 FOR I=1 TO 10
30 PRINT RIGHT$(A$,I)
40 NEXT
run
8
-8
X-8
PX-8
 PX-8
n PX-8
on PX-8
son PX-8
pson PX-8
Epson PX-8
Ok
```

RND

Format **RND[(X)]**

Purpose Returns a random number with a value between 0 and 1.

Remarks RND returns a random number from a sequence determined mathematically by the random number generator in the BASIC interpreter. The same sequence of numbers is generated each time a program containing the RND function is executed. If X is omitted or the number specified for X is greater than 0, the next random number in the sequence is generated. If 0 is specified for X, RND repeats the last random number generated. If the number specified for X is less than 0, RND starts a new sequence whose initial value is determined by the value specified for X.
It is sometimes necessary to generate numbers in a given range. The following examples show how this may be done.

INT (RND (X) * 1ØØ)	Generates numbers in the range 0—99.
INT (RND (X) * 1ØØ) + 1	Generates numbers in the range 1—100.
INT (RND (X) * 1ØØ) + 5Ø	Generates numbers in the range 100—149.
INT (RND (X) * 1Ø) – 5	Generates numbers in the range –5 to +4.

See also **RANDOMIZE**

Example The following program shows how to use RANDOMIZE and the value returned by TIME$, to give a random number which is as random as the computer will allow. Run the program a number of times to see that the numbers output from line 30 are the same each time the program is run. The first value will be the same for the five circuits of the repeating loop (lines 30 to 50) because a negative value of X repeats the number by continually reseeding with the same number. The next values will be the same as these values because X=0 which repeats the last random number. When X is from 1 to 3 a different number is produced each time in the loop lines 30 – 50, but this number will be the same each time the program is run.

The second set of numbers, generated by lines 100 – 130, is again the same every time the program is run, despite RANDOMIZE having been used.

The last set of numbers are different every time the program is run because the number used to generate the seed is different. It is obtained from the string returned by TIME$, by multiplying the hours by the seconds. This requires not only string manipulation to remove the characters from each end of the string, but also using the VAL function to convert them into numbers. More complex algorithms could be used.

```
10 FOR X = -1 TO 3
20 FOR N = 1 TO 5
30 PRINT RND(X);
40 NEXT N
45 PRINT
50 NEXT X
60 '
70 '
80 ' first randomize routine
90 RANDOMIZE(456)
100 FOR J = 1 TO 10
110 PRINT INT(RND(1)*1000);
120 NEXT
130 PRINT
140 '
150 ' second randomize routine
160 FOR J = 1 TO 10
170 RANDOMIZE(VAL(LEFT$(TIME$,2))*VAL(RIGHT$(TIME$,2)))
180 PRINT INT(RND(1)*1000);
190 NEXT
200 PRINT
```

```
 .308601   .308601   .308601   .308601   .308601
 .308601   .308601   .308601   .308601   .308601
 .498871   .670127   .98706   .739354   .783018
 .949844   .55241   .681371   .823571   .244878
 .421504   .775332   .310637   .346463   .056878
 422  292  906  614  667  833  595  910  875  173
 86  120  948  4  839  687  0  507  224  514
Ok
```

RUN

Format

RUN [<line number>]
RUN <file descriptor>[,R]

Purpose

Initiates program execution.

Remarks

The first format is used to start execution of the program in the the currently selected program area. Execution begins at the first line of the program unless <line number> is specified. If <line number> is specified, execution begins at the specified line. All files are closed and variables cleared, even if the <line number> specified is not the first line of the program. To restart a program from a particular line number without using RUN, use GOTO <line number>.

The second format is used to load and execute a program from a disk device, including the microcassette drive and RS-232C interface. Specify the name under which the program was saved in <file descriptor>; if the extension is omitted, ".BAS" is assumed. For the RS-232C interface, specify "COMØ:[(<options>)] as the file descriptor.

The RUN command normally closes all files which are open and deletes the current contents of memory before loading the specified program. However, all data files will remain open if the R option is specified although the variables will be cleared.

See also

GOTO, LOAD, MERGE

Examples

RUN 3ØØ
runs the program from line 300.

RUN "ADDRESS.BAS"
loads and runs the program "ADDRESS.BAS" in the default drive.

RUN "D:ENTRY.BAS",R
loads and runs the program "ENTRY.BAS" from disk drive D: without closing the files which were opened by the previous program.

SAVE

Format

SAVE <file descriptor>[,A]
SAVE <file descriptor>[,P]

Purpose

Used to save programs to disk device files or the RS-232C communications interface.

Remarks

This command saves BASIC programs to disk files or the RS-232C communications interface. In the former case, specify the drive name, file name and extension in <file descriptor>.
The currently active drive is assumed if the drive name is omitted, and ".BAS" is assumed if the extension is omitted. In the latter case, specify "COMØ:[(<options>)]" as the file descriptor.

If the A option is specified, the program will be saved in ASCII format; otherwise it will be saved in compressed binary format. The ASCII format requires more disk space for storage than binary format, but some file access operations require that the file be in ASCII format (for example, the file must be in ASCII format if it is to be loaded with the MERGE command).

If the P (protect) option is specified, the program will be saved in an encoded binary format. When a file is saved using this option it cannot be edited or listed when it is subsequently loaded. Once a program has been saved with the P option the protected condition cannot be cancelled.

See also

LOAD, MERGE

Examples

```
SAVE "ADDRESS"
SAVE "B:ADDRESS.ASC",A
SAVE "COMØ:(A8N3FXN)"
SAVE "SECRET",P
```

SCREEN

Format **SCREEN [<mode>] [,[<virtual screen>]**
[,[<function key switch>] [,[<boundary character>]
[WIDTH [<no. columns>][,[<no. lines 1>][,<no. lines 2>]]]]]]

Purpose Selects the screen mode and sets the various screen parameters.

Remarks The value specified in <mode> determines the mode of operation of the PX-8's display screen. If this parameter is omitted, the value corresponding to the current screen mode is assumed. The value specified for <mode> should be an integer with a value from 0 to 3; screen modes selected for each value are as follows:

Ø: 8Ø-column mode
1: 39-column mode
2: Split screen mode
3: Graphic mode

Either 0 or 1 is specified for <virtual screen>. This parameter determines the virtual screen which is used, with 0 specifying the first virtual screen and 1 specifying the second virtual screen. If this parameter is omitted, the currently selected screen is assumed.

The <function key switch> parameter determines whether the definitions of the programmable function keys are displayed. Function key display is turned off if 0 is specified, and is turned on if 1 is specified.

The <boundary character> parameter determines the character which is displayed at the boundary between virtual screens in screen mode 2. It only functions in this mode. If omitted, the currently specified character is assumed.

Parameters from WIDTH on determine the size of the virtual screens. Rules for specifying these parameters are the same as with the WIDTH statement. When the WIDTH parameter is omitted, the current values are assumed if the mode specified is the same as the current mode, and the following values are assumed if the mode specified differs from the current mode.

Mode	No. columns	No. lines 1	No. lines 2
0	80	24	24
1	39	48	48
2	40	48	48
3	80	7 or 8	Not applicable

If a value of <mode> other than the current mode is specified in the SCREEN statement (or if the boundary character is changed), both virtual screens and the real screen are completely cleared. If the WIDTH parameters are specified, the screens are cleared even if the mode and boundary character are not changed.

See also

WIDTH and Chapter 2 for a detailed description of the screen modes.

Examples

SCREEN 3 simply switches to the graphics screen mode.

SCREEN 1, 0, 0, sets screen mode 1 to display the first virtual screen, without the function key assignments on the bottom line.

SCREEN 2, 0, 1 sets screen mode 2 to display the first virtual screen, with the function key assignments shown on the bottom line.

SCREEN , , 0 clears the bottom line, without changing the virtual screen or screen mode.

SCREEN ,1 changes the virtual screen to the second virtual screen.

**SCREEN 2, 1, 0, "$"** sets screen mode 2 with the cursor on the second virtual screen and therefore on the right hand side of the split screen. The boundary character is a "$". There is no function key display on the bottom line.

SCREEN , , , "&" changes the boundary character if the screen is in screen mode 2, otherwise nothing happens.

SCREEN 2, , , CHR$ (140) clears the screen to screen mode 2 with a boundary character which is the graphics character having ASCII code 140.

SCREEN 2, 0, 1, " = "WIDTH 10 sets screen mode 2 with a width of 10 columns on the left hand side of the screen. The boundary character is an equals sign. The cursor sits in the left hand screen, and the function key display is set on.

SCREEN 1, , , WIDTH 39 sets the screen to screen mode 1, but because the WIDTH must be set to 39 columns, the change to screen mode 1 is the only effective action needed. NOTE that there are TWO SPACES between the delimiting comma and WIDTH. This is the normal space for the format plus a space because no boundary character is present.

SCREEN 1, , ,"$"WIDTH 39, 20 sets screen mode 1. The boundary character is ignored since it can only be changed in screen mode 2. Both virtual screens are set to 20 lines.

SCREEN 1, , , WIDTH , 30, 20 sets screen mode 1. Both virtual screens are set to 30 lines, because the second line parameter is ignored in screen mode 1.

SCREEN 3, , , WIDTH , 2Ø will only set screen mode 3 since the number of lines in this mode is fixed.

SCREEN

Format **SCREEN (<horizontal position, vertical position>)**

Purpose Returns the ASCII character code corresponding to the character displayed at the specified character coordinates on the screen.

Remarks <horizontal position> must be specified as a number with a value in the range from 1 to Xmax, where Xmax is the number of columns in the currently selected virtual screen or graphic screen. <vertical position> must be specified as a number in the range from 1 to Ymax, where Ymax is the number of lines in the currently selected virtual screen or graphic screen.

Example

```
10 CLS
20 PRINT
30 PRINT"  HELLO MUM"
40 PRINT
50 FOR N = 1 TO 12
60 A = SCREEN(N,2)
70 B$ = CHR$(A)
80 PRINT A;"(";B$;")  ";
90 NEXT N


  HELLO MUM

 32 ( )  32 ( )  72 (H)  69 (E)  76 (L)  76 (L)  79 (O)
 32 ( )  77 (M)  85 (U)  77 (M)  32 ( )
Ok
```

SGN

Format SGN(X)

Purpose Returns the sign of numeric expression X.

Remarks If X is greater than 0, SGN(X) returns 1. If X equals 0, SGN(X) returns 0. If X is less than 0, SGN(X) returns −1. Any numeric expression can be specified for X.

Example

```
10 A=1:  PRINT SGN(A)
20 B=1<0:PRINT SGN(B)
30 C=1=1:PRINT SGN(C)

run
 1
 0
-1
Ok
```

SIN

Format **SIN(X)**

Purpose Returns the sine of X, where X is an angle in radians.

Remarks The sine of angle X is calculated to the precision of the type of the numeric expression specified for X.

Example

```
10 CLS
20 INPUT "Enter angle in degrees";A
30 PI=4*ATN(1)
40 D=PI/180
50 PRINT "SIN(";A;")=";SIN(A*D)
60 GOTO 20
```

```
Enter angle in degrees? 0
SIN( 0 )= 0
Enter angle in degrees? 30
SIN( 30 )= .5
Enter angle in degrees? 45
SIN( 45 )= .707107
Enter angle in degrees?
```

SOUND

Format **SOUND <frequency>,<duration>**

Purpose Generates a tone of specified frequency and duration.

Remarks The <frequency> parameter determines the pitch of the sound generated, and is specified as a number from 0 to 2500. Values from 100 to 2500 cause a sound to be generated at the equivalent frequency (in Hertz), and values from 0 to 99 result in no sound. Frequencies corresponding to the notes of the musical scale are as follows, taking the standard A as 440 Hz:

A	110	220	440	880	1760
Bb	116	233	466	932	1864
B	123	247	494	988	1975
C	131	261	523	1046	2093
C#	138	277	554	1109	2217
D	147	294	587	1174	2349
Eb	155	311	622	1244	2489
E	165	329	659	1318	
F	175	349	699	1397	
F#	185	370	740	1480	
G	196	392	784	1568	
Ab	208	415	830	1661	

The <duration> parameter determines the length of the sound generated, and is specified as a number from 15 to 2554. The length of the sound generated is equal to approx. <duration> × 10 msec, with the result rounded off to the nearest millisecond. Durations in crotchet units corresponding to various metronome markings are as follows:

Metronome	♩=200	♩=150	♩=100	♩=60
𝅝.	180	240		
𝅝	120	160	240	
𝅗𝅥.	90	120	180	
𝅗𝅥	60	80	120	200
♩.	45	60	90	150
♩	30	40	60	100
♪.	22	30	45	75
♪	15	20	30	50

When a SOUND statement is executed, BASIC waits for execution of that statement to be completed before going on to the next statement. An example of the use of the SOUND statement in a program which plays music is shown below.

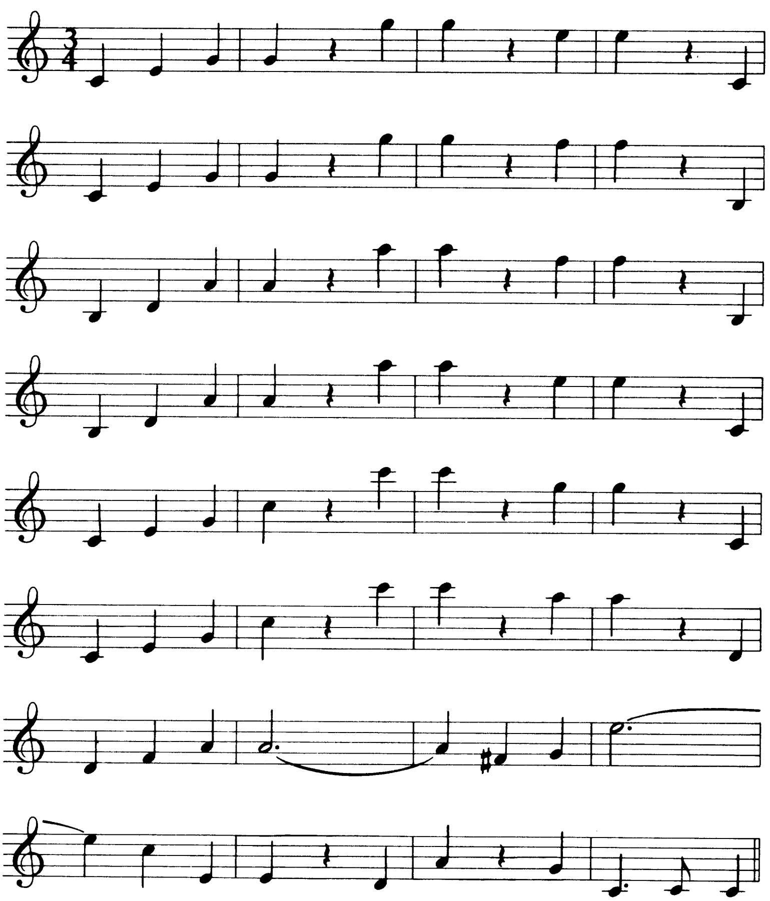
See also **BEEP**

Example

```
10 READ A
20 IF A = 1 THEN 100
30 SOUND A,30
40 GOTO 10
50 DATA 261,329,392,392,0,784,784,0,660,660,0,261,
261,329,392,392,0
60 DATA 784,784,0,699,699,0,247,247,294,440,440,0,
880,880,0,699,699,0
70 DATA 247,247,294,440,440,0,880,880,0,660,660,0,
261,261,329,392,523,0
80 DATA 1046,1046,0,784,784,0,261,261,329,392,523,
0,1046,1046,0
90 DATA 880,880,0,294,294,349,440,1
100 SOUND 440,120:SOUND 370,30:SOUND 392,30:SOUND
660,120
110 SOUND 523,30:SOUND 329,30:SOUND 329,30:SOUND 0
,30
120 SOUND 294,30:SOUND 440,30:SOUND 0,30:SOUND 392
,30
130 SOUND 261,45:SOUND 261,15:SOUND 261,30
140 END
```

The Blue Danube Waltz

by Johann Strauss



SPACE$

Format **SPACE$(J)**

Purpose Returns a string of spaces whose length is determined by the value specified for J.

Remarks The value of J must be in the range from 0 to 255. If other than an integer expression is specified for J, it is rounded to the nearest integer.

See also **SPC**

Example

```
10 FOR J=1 TO 7
20 READ A$,B$
30 P$=A$+SPACE$(20-LEN(A$+B$))+B$
40 PRINT P$
50 NEXT
60 DATA Angie,523-2121,Alfie,456-1010,Robert,21-4444
70 DATA Susan,223-1234,Charlie,234-2324,John,703-7654
80 DATA Randolph,631-1360

run
Angie       523-2121
Alfie       456-1010
Robert       21-4444
Susan       223-1234
Charlie     234-2324
John        703-7654
Randolph    631-1360
Ok
```

SPC

Format SPC (J)

Purpose Returns a string of spaces for output to the display or printer.

Remarks The SPC function can only be used with an output statement such as PRINT or LPRINT — unlike the SPACE$ function, it cannot be used to assign spaces to variables. The value specified for J must be in the range from 0 to 255.

See also SPACE$

Example

```
10 FOR J=1 TO 7
20 READ A$,B$
30 PRINT A$;SPC(20-LEN(A$+B$));B$
40 NEXT
50 DATA Angie,523-2121,Alfie,456-1010,Robert,21-4444,Susan,2
23-1234,Charlie,234-2324,John,703-7654,Randolph,631-1360

run
Angie       523-2121
Alfie       456-1010
Robert       21-4444
Susan       223-1234
Charlie     234-2324
John        703-7654
Randolph    631-1360
```

SQR

Format **SQR (X)**

Purpose Returns the square root of X.

Remarks The value specified for X must be greater than or equal to 0.

Example

```
10 PRINT "X","SQR(X)"
20 FOR X=0 TO 100 STEP 10
30 PRINT X,SQR(X)
40 NEXT
run
X              SQR(X)
 0              0
 10             3.16228
 20             4.47214
 30             5.47723
 40             6.32456
 50             7.07107
 60             7.74597
 70             8.3666
 80             8.94427
 90             9.48683
 100            10
Ok
```

STAT

Format

STAT [<program area no.>]
STAT ALL

Purpose

Displays the status of the BASIC program areas.

Remarks

Executing the STAT command without specifying any parameters displays the status of the currently selected program area. If <program area no.> is specified the status of that program area is displayed. In both cases the display format is as follows.

```
Pn:XXXXXXXX YYYYY Bytes
```

Here n indicates the number of the applicable program area, XXXXXXXX indicates the name assigned to that program area by the TITLE command and YYYY indicates the size of the program area (i.e., the size of the program in that area) in bytes. Spaces are displayed for XXXXXXXX if no name has been assigned with the TITLE command.

When an asterisk (*) is displayed between Pn and the program area name, the edit-inhibit attribute has been set for that area with the TITLE command.

Executing STAT ALL displays the status of all program areas as follows.

```
P1 : AAAAAAAA  aaaaa  Bytes
P2 : BBBBBBBB  bbbbb  Bytes
P3 : CCCCCCCC  ccccc  Bytes
P4 : DDDDDDDD  ddddd  Bytes
P5 : EEEEEEEE  eeeee  Bytes
     xxxxx Bytes Free
```

The number displayed for xxxxx indicates the current number of bytes of memory which are available for use by BASIC.

An "Illegal function call" error will occur if a number other than 1 to 5 is specified for <program area no.>.

See also **LOGIN, MENU, PCOPY, TITLE**

Example

```
stat all
P1:TESTPROG    316 Bytes
P2:            113 Bytes
P3:             59 Bytes
P4:           1302 Bytes
P5:            109 Bytes
   12850 Bytes Free
Ok


login 1
P1:TESTPROG    316 Bytes
Ok
stat 2
P2:            113 Bytes
Ok
```

STOP

Format **STOP**

Purpose Terminates program execution and returns BASIC to the command level.

Remarks STOP statements are generally used to interrupt program execution during debugging to allow the contents of variables to be examined or changed in the direct mode. Program execution can then be resumed by executing a CONT command.

The following message is displayed upon execution of a STOP statement:

Break in nnnnn

Unlike the END statement, no files are closed when a STOP statement is executed.

See also **CONT**

Example

```
10 PRINT "Program line 10"
20 STOP
30 PRINT "Program line 20"

run
Program line 10
Break in 20
Ok
cont
Program line 20
Ok
```

STOP KEY

Format **STOP KEY** |**ON**|
OFF

Purpose Disables or re-enables the STOP key.

Remarks Executing STOP KEY OFF disables the STOP and CTRL + C keys. This prevents processing from being interrupted if the STOP key is pressed accidentally during execution of a BASIC program.

Executing STOP KEY ON re-enables the STOP and CTRL + C keys after they have been disabled.

If a program becomes trapped in an endless loop after executing STOP KEY OFF, execution can only be interrupted by pressing the reset switch on the left side of the PX-8. Therefore, care should be taken to debug programs completely before including the STOP KEY OFF statement.

Example

```
10 STOP KEY OFF         :'Disables STOP key.
20 A$=INPUT$(10)        :'Type in 10 characters to proceed.
30 '  Can't be interrupted by pressing STOP key.
40 PRINT A$
50 STOP KEY ON          :'Re-enables STOP key.
60 B$=INPUT$(10)        :'Type in 10 characters to proceed.
70 '  Can be interrupted by pressing STOP key.
80 PRINT B$
90 END
```

STR$

Format **STR$(X)**

Purpose Converts numeric data to string data.

Remarks This function returns a string of ASCII characters which represent the decimal number corresponding to the value of X. X must be a numeric expression.

This function is complementary to the VAL function.

See also **VAL**

Example

```
10 FOR X=1 TO 10
20 PRINT X;
30 NEXT
40 PRINT
50 FOR X=1 TO 10
60 A$=STR$(X)
70 PRINT A$;
80 NEXT

run
 1  2  3  4  5  6  7  8  9  10
 1 2 3 4 5 6 7 8 9 10
Ok
```

STRING$

Format **STRING$(J, K)**
STRING$(J, X$)

Purpose Returns a string of characters.

Remarks The length of the character string returned by this function is determined by the value of J. If K is specified the function returns a string of J characters whose ASCII code corresponds to the value of K. If a non-integer value is specified for K its value if rounded to the nearest integer before the string of characters is returned.

If X$ is specified this function returns a string of J characters made up of the first character of the specified string.

Example

```
10 A$=STRING$(5,"A")
20 B$=STRING$(5,66)
30 PRINT A$:PRINT B$

run
AAAAA
BBBBB
Ok
```

SWAP

Format **SWAP <variable 1>,<variable 2>**

Purpose The SWAP statement exchanges the values of variables specified in <variable 1> and <variable 2>.

Remarks The SWAP statement may be used to exchange the values of any type of variable, but the same variable types must be specified in both <variable 1> and <variable 2>; otherwise a “Type mismatch” error will occur.

Example

```
10 'Using SWAP for alphabetization
20 FOR J=1 TO 5
30 READ A$(J)
40 NEXT J
50 FOR J=2 TO 5
60 IF A$(J-1)>A$(J) THEN SWAP A$(J-1),A$(J):J=1
70 NEXT J
80 FOR J=1 TO 5
90 PRINT A$(J)
100 NEXT J
110 DATA Mary,Charlie,Angie,Jane,Andy

run
Andy
Angie
Charlie
Jane
Mary
Ok
```

SYSTEM

Format **SYSTEM**

Purpose Terminates BASIC operation and returns control to the CP/M operating system.

Remarks This command may be executed either in the direct mode or the indirect mode.

TAB

Format **TAB(J)**

Purpose Spaces to column J on the LCD screen or printer. If the cursor/print head is already past column J, it is spaced to that column on the next line.

Remarks The character position on the far left side of the LCD screen or printer is column 0, and that on the far right side is the device width minus one. For the LCD screen, the device width is the number of columns determined for the currently selected screen by the SCREEN or WIDTH statement. For a printer, it is the number of columns determined by the WIDTH LPRINT statement.

If the value specified for J is greater than the device width minus one, the number of spaces generated is equal to J MOD n, where n is the device width.

In the expression

PRINT TAB (J) ; A$

the string A$ will be printed with the first character starting at position J. However, if the length of string A$ added to the value of J is greater than 81, the string will be printed on the next line.

The TAB function can only be used with the PRINT and LPRINT statements, and cannot be used to generate strings of spaces for other purposes.

Example

```
10 SCREEN 0
20 PRINT 1;2;3;4;5
30 PRINT 1;TAB(4);2;TAB(9);3;TAB(15);4;TAB(22);5

run
 1  2  3  4  5
 1  2    3     4      5
Ok
```

NOTE:
If a space is included between TAB and the opening bracket in TAB (J), PX-8 BASIC will interpret this as item J of an array with the name TAB. Rather than print the next string at position J, the value 0 will be printed because the value of all the items in this array will be 0. If J is greater than 10, a "Subscript Out of Range" error will be generated because the TAB array has not been dimensioned.

TAN

Format **TAN(X)**

Purpose Returns the tangent of X, where X is an angle in radians.

Remarks The tangent of angle X is calculated to the precision of the type of numeric expression specified for X.

To convert an angle from degrees to radians, multiply it by 1.57080 (for single precision) or by 1.570796326794897/90 (for double precision).

See also **ATN, COS, SIN**

Example

```
10 INPUT "Enter angle in degrees";A
20 PRINT "Tangent";A;"degrees is";TAN(A*3.14159/180)
30 GOTO 10

Enter angle in degrees? 30
Tangent 30 degrees is .57735
Enter angle in degrees? 45
Tangent 45 degrees is .999999
Enter angle in degrees? 60
Tangent 60 degrees is 1.73205
Enter angle in degrees?
```

TAPCNT

Format

As a statement
TAPCNT=J

As a variable
J=TAPCNT

Purpose

Reads or sets the value of the PX-8's microcassette drive counter.

Remarks

TAPCNT is a system variable which is used to maintain the value of the PX-8's microcassette drive counter. When used as a statement, TAPCNT changes the setting of the drive counter. In this case, the tape in the microcassette drive must be in the unmounted condition and the value specified for J must be in the range from −32768 to 32767. A "Tape access error" will result if an attempt is made to set the counter while the tape is in the mounted condition.

As a variable, TAPCNT returns the value of the microcassette drive counter as a number in the range from −32768 to 32767.

Example

```
tapcnt=5000
Ok
print tapcnt
 5000
Ok
```

TIME$

Format

As a statement
TIME$="<HH>:<MM>:<SS>"

As a variable
X$=TIME$

Purpose

Used to read or set the time of the PX-8's built-in clock.

Remarks

TIME$ is a system variable which is used to maintain the time of the PX-8's built-in clock. When specified as a statement, TIME$ changes the setting of the clock. <HH> is a 2-digit number from 00 to 23 which indicates the hour, <MM> is a 2-digit number from 00 to 59 which indicates the minute, and <SS> is a 2-digit number from 00 to 59 which indicates the second.

As a variable, TIME$ returns the time of the built-in clock in "HH:MM:SS" format.

Example

```
time$="17:35:00"
Ok
print time$
17:35:05
Ok
```

TITLE

Format **TITLE [<program area name>][,P]**

Purpose To assign a name to the currently logged in program area.

Remarks This command assigns a name to the currently selected BASIC program area. The name is specified as a string of from 0 to 8 letters. If more than 8 letters are specified, excess letters are ignored.

After execution of this command, the specified program area name is displayed whenever the BASIC program area menu is displayed or the STAT command is executed. If a program is loaded from a disk device, the file name of the program loaded is used as the program area name.

The program area name can be cancelled by executing the TITLE command with a null string ("") specified for <program area name>. The program area name is also cancelled when the NEW command is executed. The program area name is not affected if the <program area name> parameter is omitted.

If the P (protect) option is specified, executing this command sets the edit inhibit attribute for the currently selected program area. Once a program area has been protected in this mannner, any attempt to edit that program or to execute a DELETE command in that program area will result in an "Illegal function call" error.

See also **NEW**

Example

```
stat
P1:            443 Bytes
Ok
title "SAMP1"
Ok
stat
P1:SAMP1       443 Bytes
Ok
```

TRON/TROFF

Format **TRON**
TROFF

Purpose Used to enable or disable the trace mode of execution.

Remarks In the trace mode, the number of each line of a program is displayed on the screen in square brackets at the time that line is executed. This makes it possible to determine the sequence in which program lines are executed, and as such can be used with the STOP and CONT commands during program debugging.

The trace mode is enabled by executing TRON and is disabled by executing TROFF.

See also **CONT, STOP**

Example

```
10 FOR I=1 TO 3:PRINT I;:NEXT
20 PRINT
30 TRON
40 FOR I=4 TO 6:PRINT I;:NEXT
50 TROFF

run
 1  2  3
[40] 4  5  6 [50]
Ok
```

USR

Format **USR [<digit>]<argument>**

Purpose Passes the value specified for <argument> to a user-written machine language routine and returns the result of that routine.

Remarks <digit> is an integer from 0 to 9 which corresponds to the digit specified in the DEF USR statement for the machine language routine. If <digit> is omitted, USRØ is assumed.

A string or numeric expression must be specified for <argument>; this argument is passed to the machine language routine as described in Appendix D.

VAL

Format **VAL(X$)**

Purpose Converts a string composed of numeric ASCII characters into a numeric value.

Remarks This function returns the numeric value of a character string consisting of numeric characters. The first character of string X$ must be "+", "−", ".", "&" or a numeric character (a character whose ASCII code is in the range from 48 to 57); otherwise, this function returns 0.

Some examples of use of the VAL function are shown below.

(1) VAL(X$)

Returns the decimal number which corresponds to the string representation of that decimal number. X$ is composed of the characters "0" to "9" and may be preceded by "+", "−" or ".". Complementary to the STR$ function.

(2) VAL("&H"+X$)

Returns the decimal number which corresponds to the string representation of a hexadecimal number. X$ is composed of the characters "0" to "9" and "A" to "F". This is complementary to the HEX$ function.

(3) VAL("&O"+X$)

Returns the decimal number which corresponds to the string representation of an octal number. X$ is composed of the characters "0" to "7". This is complementary to the OCT$ function.

See also **HEX$, OCT$, STR$**

Example

```
10 INPUT "Type in a hexadecimal number ";A$
20 PRINT "The decimal value of &H";A$;" using VAL(";CHR$(34)
;"&H";CHR$(34);"+X$) is ";VAL("&H"+A$)
30 INPUT "Type in an octal number ";B$
40 PRINT "The decimal value of &O";B$;" using VAL(";CHR$(34)
;"&O";CHR$(34);"+X$) is ";VAL("&O"+B$)
run
Type in a hexadecimal number ? 4F
The decimal value of &H4F using VAL("&H"+X$) is  79
Type in an octal number ? 32
The decimal value of &O32 using VAL("&O"+X$) is  26
Ok
```

VARPTR

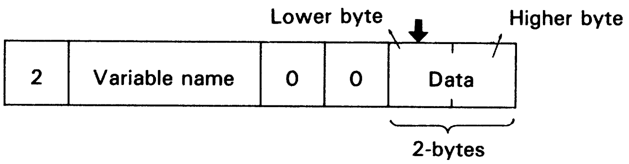
Format **VARPTR(<variable name>)**
VARPTR(# <file number>)

Purpose The first format returns the address in memory of the first data byte of the variable specified in <variable name>.
The second format returns the starting address of the I/O buffer assigned to the file opened under <file number>.

Remarks With the first format, a value must be assigned to <variable name> before executing VARPTR; otherwise, an "Illegal function call" error will result. Any type of variable name (numeric, string, or array) may be specified, and the address returned will be an integer in the range from −32768 to 32767. If a negative number is returned, add it to 65536 to obtain the actual address.

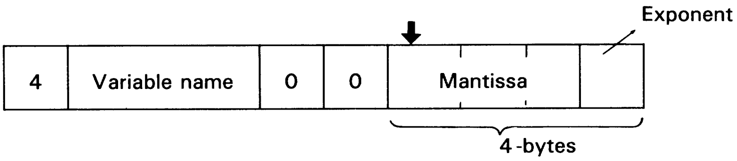
Storage of the various types of data in memory is as follows. (↓ indicates the byte which corresponds to the value returned by VARPTR.)

(1) Integer variables
The data section of integer variables occupies two bytes in memory. Lower-order bits of this number are contained in the byte whose address is returned by the VARPTR function, and high-order bits are contained in the byte at the following address. Thus, if variable A% contains the integer 2, the address returned by the VARPTR function (the low-order byte) will contain 2, and that address plus 1 (the high-order byte) will contain 0.

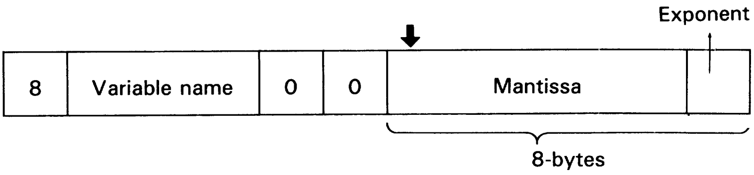


(2) Single precision variables
With single precision variables, numeric values are stored in two parts, using a total of four bytes of memory. The first part which is referred to as the exponent, and the remaining three bytes are referred to as the mantissa.

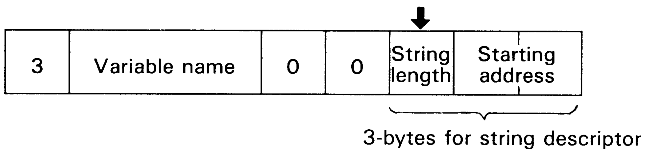
The VARPTR function returns the address of the least significant byte of the mantissa; the VARPTR address + 1 contains the middle byte of the mantissa; and the VARPTR address + 2 contains the most significant byte of the mantissa. The exponent is at the VARPTR address + 3.



(3) Double precision variables
The storage format for double precision values in variables is the same as with single precision variables. However, the mantissa portion of a double precision variable consists of seven bytes instead of three, so the data portion of a double precision variable occupies a total of eight bytes in memory.



(4) String variables
With string variables, the VARPTR function returns the length of the string. The low-order byte of the string's starting address in memory is indicated by the VARPTR address + 1, and the high-order byte of the string's starting address in memory is indicated by the VARPTR address + 2.



The second format returns the address of the first byte of the I/O buffer assigned to the file opened under <file number>.

This function is generally used to obtain the address of a variable prior to passing it to a machine language program. In the case of array variables, the format VARPTR(A(0)) is generally used so that the address returned is that of the lowest-numbered element of the array.

VARPTR is an abbreviation for VARiable PoinTeR.

Example

```
30 A$="abcdefghijklmnopqrstuvwxyz"
40 A=VARPTR(A$):PRINT "Address of variable A$ is ";A
50 B=PEEK(A+2)*&H100+PEEK(A+1)
60 PRINT "Address of string in variable A$ is";B
70 PRINT "String in variable A$ is ";:FOR I=0 TO 25:PRINT CH
R$(PEEK(B+I));
90 NEXT

run
Address of variable A$ is -29624
Address of string in variable A$ is 35638
String in variable A$ is abcdefghijklmnopqrstuvwxyz
Ok
```

NOTE:
The addresses of array variables change whenever a value is assigned to a new simple variable; therefore, all simple variable assignments should be made before calling VARPTR for an array.

WAIT

Format

WAIT <port number>, J [, K]

Purpose

Suspends program execution while monitoring the status of a machine input port.

Remarks

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 exclusive ORed with the value of integer expression K, then ANDed with J. If the result is zero, BASIC loops back and reads the port again. If the result is not 0, execution resumes with the next statement. If K is omitted, 0 is assumed.

NOTE:
Use of this statement requires in-depth knowledge of the PX-8 firmware, and using it incautionsly can result in loss of system control and other problems. See the PX-8 OS Reference Manual for detailed information on the PX-8 firmware.

WHILE...WEND

Format

```
WHILE <expression>
      .
      .
[<loop statements>]
      .
      .
WEND
```

Purpose

Allows the series of instructions between WHILE and WEND to be repeated as long as the condition specified by <expression> is satisfied.

Remarks

This statement causes program execution to loop through the series of instructions between WHILE and WEND as long as the condition specified by <expression> is satisfied. <expression> is specified as any expression which has a truth value of 0 (false) or other than 0 (true). Thus numeric, logical or relational expressions may be used to specify the condition which controls looping.

As with FOR/NEXT loops, WHILE/WEND loops may be nested to any level. They may also be included within FOR/NEXT loops or vice versa. When loops are nested, the first WEND corresponds to WHILE of the innermost loop, the second WEND corresponds to WHILE of the next innermost loop, and so forth.

A "WHILE without WEND" error will occur if WHILE is encountered without a corresponding WEND, and a "WEND without WHILE" error will occur if WEND is encountered without a corresponding WHILE.

See also

FOR...NEXT

Example

```
10 INPUT"Enter arbitrary number";X
20 WHILE X^A<1E+06
30 PRINT STR$(X);"^";MID$(STR$(A),2,5);"=";MID$(STR$(X^A),2,
6)
40 A=A+1
50 WEND

run
Enter arbitrary number? 42.5
 42.5^0=1
 42.5^1=42.5
 42.5^2=1806.2
 42.5^3=76765.
Ok
```

WIDTH

Format

WIDTH [<no. of columns>][,[<no. of lines 1>][,<no. of lines 2>]]
WIDTH <file descriptor>, <no. of columns>
WIDTH # <file no.>,<no. of columns>
WIDTH LPRINT <no. of columns>

Purpose

Sets the column width of the virtual screens or other specified device or file.

Remarks

(1) **WIDTH [<no. of columns>][,[<no. of lines 1>][,<no. of lines 2>]]**
This format specifies the size of the PX-8's virtual screens. When the screen size of virtual screen 1 (VS-1) is expressed as m1 columns × n1 lines and that of VS-2 is expressed as m2 columns × n2 lines, the size of the virtual screens is determined as follows:

(a) Mode 0
When this statement is executed during display in screen mode 0 (the 80-column mode), the value specified in <no. of columns> must be 80. Values for n1 and n2 must be greater than or equal to 8, and n1+n2 must be less than or equal to 48. If <no. of lines 2> is omitted, the value specified for <no. of lines 1> is assumed.

(b) Mode 1
When executed in screen mode 1 (the 39-column mode), the value specified in <no. of columns> must be 39. The value specified in <no. of lines 1> determines both n1 and n2; this value must be greater than or equal to 16, and must be less than or equal to 48. If <no. of lines 1> is omitted, the current value is assumed. The <no. of lines 2> parameter is ignored.

(c) Mode 2
When executed in screen mode 2 (the split screen mode), the value specified in <no. of columns> determines m1, and must be in the range from 1 to 78. m2 is determined by

(79 – <no. of columns>). The value specified in <no. of lines 1> determines both n1 and n2; the value specified must be greater than or equal to 8 and less than or equal to 48. The <no. of lines 2> parameter has no meaning with this mode, and is ignored if specified.

(d) Mode 3
This format is meaningless when executed in screen mode 3 (the graphic mode), and will result in an "Illegal function call" error if executed.

(2) **WIDTH <file descriptor>,<no. of columns>**
This format specifies the number of characters which can be output in each line to the device specified in <file descriptor>. The devices which can be specified in <files descriptor> are "LPTØ:" and "COMØ:". An "Illegal function call" error will result if any other device is specified. When "LPTØ:" is specified, this format performs the same function as WIDTH LPRINT. When the WIDTH statement is executed in this format and the specified device is already open, the output width specified does not become effective until the device has been closed and then re-opened.

(3) **WIDTH # <file no.>,<no. of columns>**
This format specifies the output width for the file (channel) specified by <file no.>. With this format, the file specified by <file no.> must be open at the time the WIDTH statement is executed. Further, the file device must be either "SCRN:", "LPTØ:", or "COMØ:"; an "Illegal function call" error will result if these conditions are not satisfied.
The width specified by executing the WIDTH statement in this format becomes effective immediately for the specified file (channel); this makes it possible to change the width of open device files at any time.

(4) **WIDTH LPRINT <no. of columns>**
This format specifies the maximum number of characters per line which can be printed by the printer.

For devices other than the display screen, any value from 1 to 255 can be specified in <no. of columns>. When a value from 1 to 254 is specified, BASIC monitors the number of characters output and, when the number exceeds that specified, automatically outputs a carriage return/line feed code. When 255 is specified, the output line width is assumed to be infinite and BASIC does not automatically output carriage returns and line feed codes.

Example

```
10 'This program outputs characters input from the keyboard
20 'to virtual screen 1 and the RS-232C interface, and
30 'outputs characters input from the RS-232C interface
40 'to virtual screen 2.
50 SCREEN 2,0,0,"*" WIDTH 38,48
80 OPEN "O",#1,"COM0:(D8N1FXN)"
90 OPEN "I",#2,"COM0:"
100 SCREEN 2,0,0
110 A$=INKEY$:IF A$<>"" THEN PRINT A$;:PRINT#1,A$;
120 A$="":SCREEN 2,1,0
130 IF NOT EOF(2) THEN A$=INPUT$(1,2):PRINT A$;
140 GOTO 100
```

NOTE:
The width values which are effective at the time BASIC is started are as follows.

LPTØ:	**80**
COMØ:	**255**

WIND

Format

```
WIND [ | <counter value> | ]
       | ON              |
       | OFF             |
```

Purpose

Used to turn the microcassette drive motor on or off, to wind the tape in either direction to a specified counter value, or to rewind the tape to its beginning while returning the counter to 0.

Remarks

When this command is executed without specifying any parameters, the tape is rewound to its beginning and the counter is reset to 0. When <counter value> is specified, the tape is wound in one direction or the other until the counter reaches the value specified. This may be an integer or numeric expression whose value lies in the range from −32768 to 32767..

Executing WIND ON places the microcassette drive in the PLAY mode. In this mode, the microcassette drive's output signal is directed to the PX-8's built-in speaker, or to the speaker or earphone connected to the external speaker jack on the PX-8's back panel. After the drive has been placed in the PLAY mode, BASIC goes on to execute any following statements. Microcassette operation in the PLAY mode is terminated by executing WIND OFF.

Note that the BEEP and SOUND statements cannot be executed while the microcassette drive is in the PLAY mode. Further, note that the WIND statement cannot be executed if the tape in the microcassette drive has been MOUNTed.

Example

```
WIND ON
```

WRITE

Format **WRITE[<list of expressions>]**

Purpose Displays data specified in <list of expressions> on the LCD screen.

Remarks If <list of expressions> is omitted, a blank line is output to the LCD screen. If <list of expressions> is included, the values of the expressions are displayed on the LCD screen.

Numeric and string expressions can both be included in <list of expressions>, but each expression must be separated from the one following it with a comma. Commas are displayed between each item included, and strings displayed are enclosed in quotation marks. After the last item has been output, the cursor is automatically advanced to the next line.

The WRITE statement displays numeric values using the same format as the PRINT statement; however, no spaces are output to the left or right of numbers displayed.

See also **PRINT**

Example

```
10 SCREEN 0:CLS
20 A=10:B=90:C$="PX-8"
30 WRITE A,B,C$
40 END
```

```
10,90,"PX-8"
Ok
```

WRITE

Format **WRITE # <file number>, <list of expressions>**

Purpose Used to write data to a sequential disk device file.

Remarks <file number> is the number under which the file was opened for output, and expressions included in <list of expressions> are the numeric and/or string expressions which are to be written to the file. Data is written to the file in the same format as it is output to the screen by the WRITE statement; that is, commas are inserted between individual items and strings are delimited with quotation marks.
Therefore, it is not necessary to specify explicit delimiters in <list of expressions>, as is the case with the PRINT # statement. The following illustrates the difference between use of the PRINT # and WRITE # statements (the statements indicated perform identical functions).

**PRINT # 1,CHR$(34);"SMITH,JOHN";CHR$(34);",";
CHR$(34);"SMITH, ROBERT";CHR$(34)**

WRITE # 1, "SMITH, JOHN","SMITH, ROBERT"

A carriage return/line feed sequence is written to the file following the last item in <list of expressions>.

See also **PRINT #** and **PRINT # USING**

Example

```
10 CLS
20 OPEN "O",#1,"A:DATA"
30 FOR I=1 TO 2
40 PRINT "Enter item";I:LINE INPUT A$(I)
50 NEXT I
60 WRITE#1,A$(1),A$(2)
70 CLOSE
80 OPEN"I",#1,"A:DATA"
90 INPUT#1,A$,B$
100 PRINT A$:PRINT B$
110 CLOSE
120 END
```

```
Enter item 1
SMITH, JOHN
Enter item 2
SMITH, ROBERT
SMITH, JOHN
SMITH, ROBERT
Ok
```