Chapter 4

COMMANDS AND FUNCTIONS

This Chapter describes the commands, statements, and functions used with PX-8 BASIC.

Commands and statements are words in the BASIC language which control operation of the computer or which set up parameters which are manipulated during computer operation. The distinction between commands and statements is as follows.

Commands — Generally executed in the direct mode, and used in manipulating BASIC program files. The LOAD command, which is used to bring a program into memory from external storage, is an example of a command.

Statements — Instructions which are included in a program to control operation of a computer or establish parameters which are manipulated during program operation. For example, execution of the GOTO statement causes execution to branch from one part of a program to another.

In practice, most commands and statements can be executed in either the direct mode or the indirect (program execute) mode, so the distinction between them is more traditional than qualitative.

Functions are procedures built into the BASIC language which return specific results for given data. Functions differ from commands and statements in that the former controls operation of the computer, while the latter produces a result and passes it to the program. An example is the SIN function, which returns the sine of a specified value. Functions may be used at any time, either from within a program or in the direct mode; there is no need for definition on the part of the user.

The following format is used in describing commands, statements, and functions in this Chapter.

Format Illustrates the general format for specification of the statement or function concerned. Meanings of the symbols used in the format descriptions are described in "Format Notations" below.

Purpose Explains the purpose of the statement or function.

Remarks Gives detailed instructions for using the statement or function.

See also Refers the reader to descriptions of other statements or functions whose operation is related in some way to that of the statement or function being described.

Example Gives examples of use of the statement or function in programs.

NOTE:
Outlines precautions which should be observed when using the statement/function, or presents other related information.

Format Notations

The following rules apply to specification of commands, statements, and functions.

(1) Items shown in capital letters are BASIC reserved words, and must be input letter for letter exactly as shown. Any combination of upper- or lowercase letters can be used to enter reserved words; BASIC automatically converts lowercase letters to uppercase except when they are included between quotation marks or in a remark statement.

e.g. **CLS PRINT STEP**
PRINT
STEP

(2) Angle brackets "< >" indicate items which must be specified by the user.

(3) Items in square brackets "[ ]" are optional.

In either case the brackets themselves have no meaning except as format notation, and should NOT be included when the statement/function is entered. If angle brackets are included inside square brackets, this means optional items to be specified by the user.

e.g. **PSET [STEP] (X,Y) ,<function code>**

might be typed to appear with various values as follows. These are particular cases, to show what is actually typed.

PSET (10,10)
is the minimum format for plotting a point at position (10,10) on the screen.

PSET (10,10), 1
plots the same point, but with <function code> having a value of 1.

PSET STEP (10,10)
plots a point relative to the last plotted point.

PSET STEP (10,10),7
plots the point relative to the last plotted point, with <function code> set to 7.

ALARM [<date>,<time>,<string> [,W]]
means that it is optional to input the <date>, <time> and a <string>; however, if it is required to use one or other of these three options, the others must be typed in as well. It is optional to use the "W" extension, but this option cannot be used without the other three options. Examples of the three valid types of statement are

ALARM
ALARM "/**/**","**:13:00","LUNCH TIME"**
ALARM, "/**/**","**:09:30","APPOINTMENT",W**

(4) All punctuation marks (commas, parentheses, semicolons, hyphens, equal signs, and so forth) must be entered exactly as shown.
When round brackets ( ) are included they MUST be typed in as shown.

(5) Where a set of full stops "..." is included, the items may be repeated any number of times, provided the length of the logical line is not exceeded.

e.g. **CLOSE [[# <filenumber>][, # <filenumber>]....]**

means that any number of files can be closed. Valid examples are

CLOSE
CLOSE #4
CLOSE #1, #3, #4

(6) Items included between vertical bars are mutually exclusive; and only one of the items shown can be included when the statement is executed.

e.g. **STOP KEY |ON |**
|OFF|

The following abbreviations are used in explaining the arguments or parameters of commands, statements, and functions.

X or Y........... Represent any numeric expressions.
J or K .... Represent integer expressions.
X$ or Y$........ Represent string expressions.

With functions, any floating point value specified as an argument will be automatically rounded to the nearest integer value if the function in question only works with integer values.

ABS

Format **ABS(X)**

Purpose Returns the absolute value of expression X.

Remarks Any numeric expression may be specified for X.

Example

```
10 CLS
20 A = 25
30 B = -25
40 C = 2.545
50 D = -2.545
60 PRINT "VARIABLE","VALUE","ABSOLUTE VALUE"
70 PRINT "A",A,ABS(A)
80 PRINT "B",B,ABS(B)
90 PRINT "C",C,ABS(C)
100 PRINT "D",D,ABS(D)

VARIABLE       VALUE          ABSOLUTE VALUE
A               25             25
B              -25             25
C               2.545          2.545
D              -2.545          2.545
Ok
```

ALARM

Format **ALARM [<date>, <time>,<string>[,W]]**

Purpose Specifies the alarm or wake time. Only one of these can be set at a time.

Remarks **USING THE ALARM COMMAND IN ALARM MODE**

Executing the ALARM statement without the W option sets the alarm time. When the alarm time is reached, the power goes on, the speaker generates a warbling sound, and a screen similar to the one below is displayed. If a program is being executed, it will be interrupted.

```
<ALARM TIME> 01/04 03:35  <ALARM  MSG> Time to get up!

           Press ESC key
```

Pressing the [ESC] key at this point returns the PX-8 to the state it was in when the alarm time was reached; if the [ESC] key is not pressed, the PX-8 automatically returns to its previous state after about 50 seconds have elapsed.

The alarm date is specified in <date> in the same format as with the DATE$ statement, and the alarm time is specified in <time> in the same manner as with the TIME$ statement. They are thus both strings.

<string> must be specified as a string expression whose length is not greater than 40 characters; this string is displayed next to the message statement when the alarm time is reached.

Asterisks can be specified as wildcard characters for any of the digits in <date> or <time>. When asterisks are specified, those positions in <date> and/or <time> will be regarded as always matching the corresponding digit in the DATE$ or TIME$ system variable. For example, executing the following statement will result in alarm operation every day at ten minute intervals from 8.00 am to 8.50 am.

```
ALARM "**/**/**","Ø8:*Ø:Ø*",A$
```

Note that the two year digits are always handled as if they were specified as asterisks. They must be specified even if only as asterisks. The second digit of the seconds value is always handled as zero.

The use of wildcards does not allow complete flexibility of operation. The alarm can only be set with this option to go off at intervals of one minute, ten minutes, one hour, ten hours or twenty-four hours. As shown in the above example, this can be within a time band. If more flexibility is required, a BASIC program has to change the ALARM strings continually. An example of this is shown in the program below. Examples of using the wildcard options are as follows; in all cases A$ denotes the message string to be printed with the alarm time.

ALARM "/**/**", "**:**:45", A$** will sound the alarm at one minute intervals when the seconds change to "40", since the second digit is always treated as a zero.

ALARM "/**/**", "**:**:00", A$** will sound the alarm as the minute changes.

ALARM "/**/**", "16:**:00", A$** will sound the alarm as the minute changes, but only when the hour matches "16" i.e. from 16.00 to 16.59.

ALARM "/**/**", "**:*5:0*", A$** will sound the alarm every ten minutes, i.e. at 5, 15, 25 etc minutes past the hour, as this is when the "5" of the minutes will match.

ALARM "/**/**", "**:32:00", A$** will sound the alarm every hour at 32 minutes past the hour on the minute.

NOTE HOWEVER:

ALARM "/**/**", "**:32:**", A$** will sound the alarm whenever the "32" minutes matches. It will sound first at 32 minutes past the hour. If the ESC key is not pressed, after 50 seconds the program will either continue or the power will be switched off again. If the ESC key is pressed before the 50 sec-

onds is up, the power will go off or the program will continue, but since the "32" minutes still matches when the next 10,20,30, etc. seconds is reached, the alarm will sound again.

ALARM "* */* */* *", "*9:30:00", A$ will sound the alarm at 9.30 am and 19.30.

ALARM "* */* */* *", "09:30:00", A$ will sound the alarm daily at 9.30 am.

USING THE ALARM FUNCTION IN WAKE MODE

Executing the ALARM statement with the W option sets the wake time. If the wake time is reached while the power is on, operation is the same as when the alarm time is reached. However, if the wake time is reached while the power is off, the string expression specified in <string> is assumed as an auto start string and not a message. The auto start string behaves as if a command was typed in at the keyboard. This means that the equivalent of pressing the [RETURN] key must be included in the string. This is achieved by adding the ASCII code for a carriage return using the CHR$( ) function, i.e. by adding CHR$(13) to the end of the string.

The time setting is carried out in the same format as for simply producing an alarm message, including the use of wildcards. See above for a detailed description.

The form of the auto start string will depend on whether BASIC is resident in memory, and whether the BASIC program it is required to run is also in memory.

If the MENU is not active the string would then have to specify that BASIC is loaded, and the appropriate BASIC program loaded and run. The extensions listed in Chapter 3 may need to be used in some cases. The following examples show various possibilities:

ALARM "* */* */* *","09:15:00","C:BASIC A:MORNING" + CHR$(13), W will load BASIC from drive C:

and then run the BASIC program with the name "MORNING" which is located in drive A: and since no program area has been specified, it will be executed in program area 1. The time setting, with wildcard options for the date, will execute this sequence every day at 9.15 am.

ALARM "* */* */* *", "* *:3Ø:ØØ", "C:BASIC A:HOURLY /F:5 /R:3"+CHR$(13), W will run the BASIC program "HOURLY" in program area 3 after setting the maximum number of files which can be opened to 5. The program will run once an hour on the half hour.

The MENU is not used when the AUTOSTART or WAKE string is invoked on power up. This means that if BASIC is already resident, it is not possible to start up a BASIC program by loading it directly, or to run a program already in one of the program areas. However, it is possible to use a trick to overcome this problem.
First go to the CP/M command line and type

SAVE Ø A:GO.COM

This saves a special file onto the RAM disk. Details are given under the SAVE command in the PX-8 User's Manual.

In order to have an AUTOSTART or WAKE string run a program in one of the BASIC program areas, the program "GO" can be used instead of the BASIC interpreter program. For example if the PX-8 is set up so that BASIC is in memory the previous two examples would have the format:

ALARM "* */* */* *", "Ø9:15:ØØ",
"A:GO A:MORNING"+CHR$(13),W

ALARM "* */* */* */", "* *:3Ø:ØØ",
"A:GO A:HOURLY" /F:5/R:3"+CHR$(13),W

To run a program which is already resident in one of the program areas, it is necessary to use the /R extension to BASIC together with the program area number. For example to run the

program in area 4 every hour at 10 minutes past the hour the following format would be required. It is still necessary to invoke the "GO" file.

ALARM "/**/**", "**:1Ø:ØØ", "A:GO /R:4"+CHR$(13),W**

To run the program in the first program area, simply use the above expression without the extension "/R:4"

When the wake time is reached while the power is off, the power goes on and stays on during execution of the program. After program execution is completed, whether and how long the power stays on is as determined by the last previously executed POWER statement.

An alarm or wake time set with the ALARM statement is the same as one specified from the System Display; execution of an ALARM statement will cancel any alarm or wake time setting made from the System Display, and vice versa.

NOTE:
It is not possible to set both an alarm and wake time simultaneously.

The alarm or wake time setting can be cleared by executing the ALARM statement without specifying any parameters.

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

The following program illustrates the use of a BASIC program to control the PX-8 so that it can switch itself on and off at fixed times. This could be used for monitoring a signal through the A/D port, or sending and receiving data through the RS232 port at specific times. Normally it is necessary to leave a computer running continually, even though the time it is required to function may only be minutes. The program is meant as a demonstration. Thus the time is deliberately altered at the beginning of the program, and will need to be reset to the correct time; TIME$ can be used to carry this out. Also, short intervals have been used so that the full effect can be shown in a convenient time. Different actions are taken each time the power is switched on.

```
10 TIME$="11:59:30"
20 SCREEN 0,0,0
30 CLS
40 LOCATE 25,1:PRINT"Power Control of the PX-8"
50 GOSUB 190
60 FOR X=1 TO 100
70 LOCATE 10,2,0:PRINT"The time is :";TIME$
80 NEXT
90 GOSUB 190
100 PRINT"Power switched on again"
110 GOSUB 190
120 PRINT"Today's date is :";DATE$
130 GOSUB 190
140 PRINT"Here we can count from 1 to 100 ";
150 FOR X=1 TO 100:LOCATE 40,5,0:PRINT X;:NEXT
160 GOSUB 190
170 'Repeat all the above
180 GOTO 30
190 FOR DLY=1 TO 1000:NEXT
200 READ WST$
210 IF WST$="End" THEN 250
220 ALARM "**/**/**",WST$,"",W
230 POWER OFF,RESUME
240 RETURN
250 CLS:PRINT"Press any key to turn the power off
completely !!"
260 PRINT "Remember to reset the time using time$"
270 A$=INPUT$(1)
280 POWER OFF
290 DATA "12:00:00"
300 DATA "12:01:00"
310 DATA "12:01:20"
320 DATA "12:01:40"
330 DATA "12:02:00"
340 DATA "12:02:10"
350 DATA "End"
```

ALARM$

Format **ALARM$ (<function>)**

Purpose Returns information regarding the setting made by the alarm statement.

Remarks <function> is specified as an integer expression whose value is from 0 to 3. The value returned by the ALARM$ function varies according to <function> as follows.

0: Returns the status of the setting made by the ALARM statement as a 1-character string. Characters returned and their meanings are as follows.

"N" — No alarm setting has been made.

"B" — An alarm setting has been made, but the specified time has not yet been reached.

"P" — An alarm setting has been made and the specified time has been reached.

1: Returns the date set by the ALARM statement in the same format as the date returned by the DATE$ function.

2: Returns the time set by the ALARM statement in the same format as the time returned by the TIME$ function.

3: Returns the message set by the ALARM statement as a character string.

NOTE:
Once "P" has been returned by executing ALARM$(Ø), "B" is returned when ALARM$(Ø) is subsequently executed.

See also **ALARM, AUTO START, POWER.**

ASC

Format ASC(X$)

Purpose Returns the numeric value which is the ASCII code for the first character of string X$. (See Appendix F for the ASCII codes.)

Remarks X$ must be a string expression. An "Illegal function call" error will occur is X$ is a null string (a string variable which contains no data, or a pair of quotation marks without any intervening characters or spaces).

See also CHR$

Example

```
10 CLS
20 A$ = "A"
30 B$ = "BOO"
40 C$ = "1234"
50 D$ = ""
60 PRINT "STRING", "ASCII value of first letter"
70 PRINT A$, ASC(A$)
80 PRINT B$, ASC(B$)
90 PRINT C$, ASC(C$)
100 PRINT D$, ASC(D$)

STRING          ASCII value of first letter
A                65
BOO              66
1234             49

Illegal function call in 100
Ok
```

ATN

Format ATN(X)

Purpose Returns the arc tangent in radians of X.

Remarks This function returns an angle in radians for expression X as a value from $-\pi/2$ to $\pi/2$. The angle will be returned as a double precision number if X is a double precision number, and as a single precision number if X is a single precision number or an integer. ATN(X) can also be used to derive a value for the constant PI. From elementary trigonometry PI = 4 * ATN(1).

As PI times radius equals 180 degrees, conversion of radians to degrees, is a matter of simple proportion. Lines 100 and 110 in Example 1 show how to obtain angles in degrees.

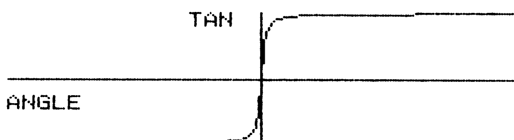
Example 1

```
10 CLS
20 INPUT "Type in the tangent of an angle" ; T
30 Y = ATN(T)
40 PRINT "The angle whose tangent is";T "is";Y;"RADIANS"
100 PI = 4 * ATN(1)
110 Z = Y*180/PI
120 PRINT "The angle whose tangent is";T "is";Z;"DEGREES"

Type in the tangent of an angle? 0.7071
The angle whose tangent is .7071 is .615475 RADIANS
The angle whose tangent is .7071 is 35.2641 DEGREES
Ok
```

Example 2

```
10 'Graphic representation of angles whose tangents
20 'range from -99 to 100.  Range of angles is from
30 '-1.5607 radians to +1.5608 radians.
40 SCREEN 3,,0:CLS
50 LINE (100,0)-(100,62)
60 LINE (0,32)-(200,32)
70 LOCATE 1,6:PRINT"ANGLE"
80 LOCATE 13,1:PRINT"TAN"
90 I=-100
100 X=I+100
110 Y=63-(ATN(I)+1.5708)*20.3718
120 PSET(X,Y)
130 FOR I=-99 TO 100 STEP 1
140 X=I+100
150 Y=63-(ATN(I)+1.5708)*20
160 LINE -(X,Y)
170 NEXT
```



AUTO

Format **AUTO [<line number>][,[<increment>]]**

Purpose Entered in the direct mode to initiate automatic program line number generation during program entry.

Remarks Executing this command causes program line numbers to be generated each time the [RETURN] key is pressed to complete the input of a program line. Numbering starts at <line number>, and subsequent line numbers differ by the value specified for <increment>. If either <line number> or <increment> is omitted, 10 is assumed as the default value; however, if a comma is specified following <line number> but no <increment> is specified, the increment specified for the last previous AUTO command is assumed.

If the currently selected program area already contains a line whose line number is the same as one generated by AUTO numbering, an asterisk (*) is displayed immediately following the number to warn the user that that line contains statements. If the [RETURN] key is pressed without entering any characters, the line is skipped without affecting its current contents; if any characters are entered before the [RETURN] key is pressed, the former contents of the line are replaced with the characters entered.

AUTO line number generation is terminated and BASIC returned to the command level by pressing [CTRL] and [C] or the [STOP] key; the contents of the last line number displayed at this time are not stored in the program area.

Example 1 **AUTO 100,50**

Generates line numbers in increments of 50, starting with line number 100. (100, 150, 200 ...)

Example 2 **AUTO**

Generates line numbers in increments of 10, starting with line number 10. (10, 20, 30...)

AUTO START

Format **AUTO START <auto start string>**

Purpose Sets the auto start string.

Remarks <auto start string> must be specified as a string expression whose length is not greater than 40 characters. When the PX-8's power is turned on, this string is handled as if it were typed in from the keyboard. If the power is turned on as a result of the wake time being reached, the string declared in the wake string will override the <auto start string>.

There must be a space between AUTO and START.

The form of the <auto start string> will depend on whether BASIC is resident in memory, and whether the BASIC program it is required to run is also in memory.

If the MENU screen is switched off, the string would have to specify loading and execution of both BASIC and the appropriate BASIC program. The extensions listed in Chapter 3 may need to be used in some cases. The following examples show various possibilities:

AUTO START "C:BASIC A:UPANDGO"+CHR$(13)

will load BASIC from drive C: and then run the BASIC program with the name "UPANDGO" which is located in drive A:. Since no program area has been specified, it will be executed in program area 1.

AUTO START "C:BASIC A:FIRST /F:5 /R:3"+CHR$(13)

will run the BASIC program "FIRST" in program area 3 after setting the maximum number of files which can be opened to 5.

The MENU is not used when the AUTOSTART or WAKE string is invoked on power up. This means that if BASIC is already resident, it is not possible to start up a BASIC program by loading it directly, or to run a program already in one of the program areas.

However, it is possible to use a trick to overcome this problem. First go to the CP/M command line and type

SAVE 0 A:GO.COM

This saves a special file onto the RAM disk. Details are given under the SAVE command in the PX-8 User's Manual.

In order to have an AUTOSTART or WAKE string run a program in one of the BASIC program areas, the program "GO" can be used instead of the BASIC interpreter program. For example if the PX-8 is set up so that BASIC is in memory the previous two examples would have the format:

AUTOSTART "A:GO A:UPANDGO"+CHR$(13)
AUTOSTART "A:GO A:FIRST /F:5 /R:3"+CHR$(13)

To run a program which is already resident in one of the program areas, it is necessary to use the /R extension to BASIC together with the program area number. For example to run the program in area 4 every hour at 10 minutes past the hour the following format would be required. It is still necessary to invoke the "GO" file.

AUTOSTART "A:GO /R:4"+CHR$(13)

The AUTO START string can be cancelled by executing the AUTO START statement with a null string (" ") specified for the <auto start string>.

The <auto start string> specified from BASIC using the AUTO START command, will appear in the appropriate area of the system display.

See also **ALARM, ALARM$, POWER**

BEEP

Format **BEEP <duration>**

Purpose Sounds the PX-8's built-in speaker.

Remarks This statement causes the speaker built into the PX-8 to make a beeping sound. The length of the sound is determined by the numeric value specified in <duration>. The value specified must be in the range from 15 to 255, and the length of the sound generated is equal to approx. <duration> × 10 msec. The frequency of the sound generated is approximately 1000 Hz.

See also **SOUND**

Example **BEEP 1ØØ**
Generates a tone with a duration of one second.

CALL

Format **CALL <variable name>[(<argument list>)]**

Purpose Calls a machine language subroutine.

Remarks The CALL statement is one method of transferring BASIC program execution to a machine language subroutine. (See also the discussion of the USR function.) <variable name> is the name of a variable which indicates the machine language subroutine's starting address in memory. The starting address must be specified as a variable (not as a numeric expression), and the variable name specified must not be an element of an array. <argument list> is the list of parameters which is passed to the machine language subroutine by the calling program. See Appendix D for further details on use of the CALL statement.

See also **USR,** Appendix D

Example The following program is an example of the use of the CALL function. It simply increments by one the number in location &HC009, and then displays the new value found by the PEEK function in line 140.

```
C000   3A 09 C0   LD A, (C009)     ;Load register A with contents of
                                    location &HC009 which has been
                                    POKEd in by a BASIC program.
C003   C6 01      ADD A, 01H       ;Add 1 to it.
C005   32 09 C0   LD (C009), A     ;Move the contents of register A
                                    back to location &HC009.
C008   C9         RET              ;RETURN to BASIC.
C009   00         NOP              ;The value obtained by the BASIC
                                    program and the location used for it.
```

```
10 CLS
20 CLEAR ,&HBFFF
30 ADRS = &HC000
40 FOR J = 0 TO 9
50 READ A
60 POKE ADRS+J,A
70 NEXT J
90 DATA &h3a, &h09, &hc0, &hc6, &h01, &h32, &h09, &hc0, &hc9
, &h00
110 INPUT "Type in a number in the range 1 to 254";B
120 POKE &HC009,B
130 CALL ADRS
140 C = PEEK(&HC009)
150 PRINT B "+ 1 ="; C

Type in a number in the range 1 to 254? 99
 99 + 1 = 100
Ok
```

CDBL

Format **CDBL(X)**

Purpose Converts numeric expression X to a double precision number.

Remarks This function converts the values of integer or single precision numeric expressions to double precision numbers. Significant decimal places added to converted numbers will contain random numbers.

Example

```
10 CLS
20 INPUT "TYPE IN TWO NUMBERS ";X,Y
30 PRINT "THE VALUE OF X multiplied by Y is ";X*Y
40 PRINT "CONVERTED TO DOUBLE PRECISION IT IS "; CDBL(X*Y)


run
TYPE IN TWO NUMBERS ? 3.45,3.141597
THE VALUE OF X multiplied by Y is  10.8385
CONVERTED TO DOUBLE PRECISION IT IS  10.83850955963135
Ok
```

CHAIN

Format

CHAIN [MERGE] <filename> [,[<line number exp>][,ALL] [,DELETE<range>]]

Purpose

Calls the BASIC program designated by <filename> and passes variables to it from the program currently being executed.

Remarks

The CHAIN statement makes it possible for one BASIC program to call (load and execute) another one. <filename> is the name of the program being called by this statement. The program called may be stored on floppy disk, in RAM disk, or on microcassette tape. However, the program called must be one which is not contained in program memory (in another program area).

If the MERGE option is not specified, the program called replaces the calling program in the program area from which the call is made. If the MERGE option is specified, the program called is brought into program memory as an overlay; statements in program lines of the calling program are replaced by similarly numbered lines in the program called. In this case, the program called must be an ASCII file (see the explanation of the SAVE command). Since it is usually desirable to ensure that the range of program line numbers in the two programs are mutually exclusive, the DELETE option may be used to delete unneeded lines in the calling program.

<line number exp> is a variable or constant indicating the line number at which execution of the called program is to begin. If omitted, execution begins with the first line of the program called.

If the ALL option is specified, all variables being used by the calling program are passed to the program called. If the ALL option is omitted, the calling program must contain a COMMON statement to list variables that are to be passed. (See the explanation of the COMMON statement.)

Note that user defined functions and variable type definitions made with the DEFINT, DEFSNG, DEFDBL, or DEFSTR state-

ments are not preserved if the MERGE option is omitted. Therefore, these statements must be restated in the program called that program is to use the corresponding variables or functions.

See also COMMON, MERGE, SAVE

Example 1 The first example shows how the chained program can replace the calling program but still preserve the variables.

```
150 'prog 2
160 X = X * X
170 PRINT "The values of X,Y, and Z from the chained program
are :- "
180 PRINT X,Y,Z
```

Save the above lines to the RAM disk using SAVE "A: PROG2",A. Delete them then type in and execute the following.

```
50 'The first program to be run
100 READ Y , Z
110 X = Y + Z
120 PRINT "The value of X,Y and Z from the first program
are:-"
130 PRINT X,Y,Z
140 DATA 2, 5
150 CHAIN "prog2",,ALL
```

```
run
The value of X,Y and Z from the first program are:-
 7              2              5
The values of X,Y, and Z from the chained program are :-
 49             2              5
Ok
```

Example 2 The second example shows how lines can be merged and lines of the original calling progam deleted. The line number from which the chained program is executed is also included in this example.

```
80 PRINT "This line is printed after line 100"
90 RETURN
100 PRINT "*** This is the chained program ***"
110 GOSUB 80
```

Save the above lines to the RAM disk using SAVE"A: SUB", A. Delete them then type in and execute the following.

```
200 PRINT "*** This is the first program *** ": PRINT
210 CHAIN MERGE "SUB",100,DELETE 200-210

Ok
run
*** This is the first program ***

*** This is the chained program ***
This line is printed after line 100
Ok
```

CHR$

Format **CHR$(J)**

Purpose Returns the character whose ASCII code equals the value of integer expression J. (See Appendix F for the ASCII codes.)

Remarks The CHR$ function is frequently used to send special characters to terminal equipment such as the console or printer. For example, executing PRINT CHR$(12) clears the entire virtual screen and returns the cursor to the home position; executing PRINT CHR$(7) causes the speaker to beep; and executing PRINT CHR$(11) moves the cursor to the home position (the upper left corner of the virtual screen) without clearing the screen. See the description of the ASC function for conversion of ASCII characters to numeric values.

It is also easier to program using numbers, so that often manipulations with ASCII codes are used in programs instead of using the actual alphabetic characters.

Example

```
10 PRINT CHR$(12)                    :'clear screen
20 FOR J = 65 TO 90                  :'Display characters A-Z
30 PRINT CHR$(J);
40 NEXT J
50 PRINT
60 FOR J = 97 TO 122                 :'Display characters a-z
70 PRINT CHR$(J);
80 NEXT J
90 '
100 PRINT CHR$(7)                    :'sound buzzer


ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
Ok
```

CINT

Format CINT(X)

Purpose Rounds the decimal portion of numeric expression (X) to the nearest whole number and returns the equivalent integer value.

Remarks X must be a numeric expression which, when rounded, is within the range from -32768 to +32767; otherwise, an "Overflow" error will occur.

See the descriptions of the FIX and INT functions for other methods of converting numbers to integers.

See the descriptions of the CDBL and CSNG functions for conversion of numeric expressions to double and single precision numbers.

NOTE:
Differences between the CINT, FIX, and INT functions are as follows.

CINT(X) *Rounds X to the nearest integer value.*
FIX(X) *Truncates the decimal fraction of X.*
INT(X) *Returns the integer value which is less than or equal to X.*

Number	Result of Function		
	CINT	FIX	INT
−1.6	−2	−1	−2
−1.2	−1	−1	−2
1.2	1	1	1
1.6	2	1	1

Although numbers are printed to the screen as you would expect, they are not always stored as such in the computer. This can lead to erroneous results with INT(X) and FIX(X) as the following program shows:

```
10 CLS
20 K1 = 2.6 : K2 = .2
30 PRINT "X","X%","INT(X)","FIX(X)","CINT(X)"
40 N = 2 : GOSUB 100
50 N = 12: GOSUB 100
60 PRINT :PRINT "The value of X - INT(X) is "; X - INT (X)
80 END
90 '
95 'subroutine to print values
100 X = K1 * N - K2
110 X% = K1 * N - K2
120 PRINT X,X%,INT(X),FIX(X),CINT(X)
130 RETURN

X              X%             INT(X)         FIX(X)         CINT(X)
 5              5              5              5              5
 31             31             30             30             31

The value of X - INT(X) is  .999998
Ok
```

The values of INT(X) and FIX(X) are apparently wrong since simple mental arithmetic will show that 12 × 2.6 − 0.2 = 31. However, the output from line 60 shows that X is stored in the computer as 30.999998 and so the functions INT (X) and FIX (X) are returning logically correct values. The error is due to the fact that numbers are converted to and handled as binary numbers by the computer. Such rounding errors are overcome by adding a small number to the answer before executing INT (X) or FIX (X) — eg: if line 120 is altered to:

```
120 PRINT X, X%, INT (X+0.0005), FIX (X+0.0005), CINT (X)
```

all values would be correct.

```
10 CLS
20 K1 = 2.6 : K2 = .2
30 PRINT "X","X%","INT(X)","FIX(X)","CINT(X)"
40 N = 2 : GOSUB 100
50 N = 12: GOSUB 100
60 PRINT :PRINT "The value of X - INT(X) is "; X - INT (X)
80 END
90 '
95 'subroutine to print values
100 X = K1 * N - K2
110 X% = K1 * N - K2
120 PRINT X,X%,INT(X + .0005),FIX(X + .0005),CINT(X)
130 RETURN
```

```
X              X%             INT(X)         FIX(X)         CINT(X)
 5              5              5              5              5
 31             31             31             31             31

The value of X - INT(X) is  .999998
Ok
```

CLEAR

Format **CLEAR [[< dummy >][,[< upper memory limit >][, < stack area size >]]]**

Purpose Resets all numeric variables to 0 and all string variables to null. When < upper memory limit > is specified, it also reserves an area in memory for machine language programs; this is done by setting an upper boundary in memory which defines the highest address (< upper memory limit > – 1) which can be used by BASIC.

Remarks The CLEAR command destroys all variables including definitions made with the DEF statements (DEFINT, DEFSNG, DEFDBL, and DEFSTR) and closes all files which are currently open. CLEAR will also clear the BASIC stack, so that if CLEAR is used in any subroutine, the program will not be able to return.

< stack area size > specifies the number of bytes of memory to be reserved as stack space. The stack space is used for purposes such as storing return addresses during execution of GOSUB statements. When BASIC is started, 256 bytes are reserved as stack space. If more stack size is required, it can be increased using the stack area size option.

For example CLEAR,, 512 will set the stack area to 512 bytes.

< dummy > is included to provide compatibility with other versions of BASIC, and has no purpose in BASIC for the PX-8.

Apart from being used for clearing variables, the main use of CLEAR is to enable space to be reserved into which to POKE machine code programs. This is done by moving down the highest memory location BASIC can use for variables, etc. To move the highest memory limit down to &HCØØØ the expression would be:

CLEAR, &HCØØØ

With the PX-8, the programmer needs to know where the start of BDOS etc lies, since the machine code area reserved is below this. This can change depending on how the system is configured.

The RAM disk and user BIOS area in particular will affect the location of the start of BDOS. Programs should determine this from page zero (locations 6 and 7) and set the start of the machine code area a suitable number of bytes below it, rather than use absolute values in a program. If an attempt is made to reserve memory above the start of BDOS, an "Out of memory" error message will be generated.

The second example shows how to set the memory limit in the correct position below BDOS.

Having set the upper memory limit, the program would normally start at that location. For example, CLEAR, &HC000 would allow the program to start at &HC000.

Example The first example illustrates that variables are destroyed by CLEAR.

```
10 A = 55 : B = 33 : A$ = "ABC"
20 PRINT A,B,A$
30 CLEAR                              :' Sets A and B to zero and
40                                    :' A$ to null
50 PRINT A,B,A$

 55              33              ABC
 0               0
Ok
```

The second example shows how to reserve space for a machine code program, both using an absolute address and relative to the start of BDOS.

```
10 CLEAR , &HAFFF ,512      :' Clears space for machine code
20 '                           from HB000 and sets stack to
30 '                           512 bytes
40 PRINT FRE(0)              :' Prints available memory
50 '
60 MT = PEEK (6) + PEEK (7) * 255    :' Find start of BDOS
70 CLEAR , MT - 100                   :' Clear 100 bytes of
80 '                                     memory for Machine
90 '                                     code beneath BDOS
100 '
110 PRINT FRE(0)                       :' Print the available
120 '                                     memory
```

```
run
 7895
 15531
Ok
```

CLOSE

Format **CLOSE[[#]<file number>[,[#]<file number>...]]**

Purpose Executing this statement terminates access to device files.

Remarks This statement terminates access to files opened previously under specified file numbers. If no file numbers are specified, this statement closes all files which are currently open.

Once opened, a file must be closed before it can be reopened under a different file number or in a different mode, or before the file number under which that file was opened can be used to open a different file. A "File already open" error will occur if an attempt is made to open a file which is already open, or if an attempt is made to open a different file using the file number assigned to a file which is already open; a "Bad file mode" error will occur if an attempt is made to use a different file number to open a file which is already open.

Executing this statement to close a random file or a sequential file which has been opened in the output mode causes the contents of the output buffer to be written to the file's end. Therefore, be sure to close any disk or microcassette files which are open for output before removing the medium from the drive; otherwise, data stored in the file will not be usable, and there is a possibility that the contents of other files may be destroyed when a CLOSE statement is executed if another disk or magnetic tape is inserted in that drive.

All files are closed automatically upon execution of an END, CLEAR, or NEW statement.

See also **END, OPEN**, Chapters 5 and 6

Example

```
10 OPEN "O",#1, "A:TEST.DAT"      :' Opens the file "TEST"
20 '                                 for output on drive A:
30 FOR J = 65 TO 90
40 PRINT #1, CHR$(J) ;               :' Writes letters A to Z
50 NEXT J                          :' to the file
60 '
70 CLOSE #1                        :' Closes file "A:TEST.DAT"
80 '
90 OPEN "I" ,#1, "A:TEST.DAT"      :' Opens file for input
100 IF EOF (1) THEN 160            :' Checks whether End of File
110 '                                 marker of file 1 has been
120 '                                 reached, and if so goes to 160.
130 A$ = INPUT$(2,1)               :' Inputs two characters from file #1
140 PRINT A$ : GOTO 100              :' Prints the characters input
150 '                                 from the file and returns for more.
160 END                            :' Ends program, Closing file
```

```
run
AB
CD
EF
GH
IJ
KL
MN
OP
QR
ST
UV
WX
YZ
Ok
```

CLS

Format **CLS**

Purpose Clears the LCD screen.

Remarks The CLS statement clears the currently selected virtual screen. This statement performs the same function as PRINT CHR$(12).

COMMON

Format **COMMON <list of variables>**

Purpose Passes variables to a program executed with the CHAIN statement.

Remarks The COMMON statement is one method of passing variables from one program to another when execution of programs is chained with the CHAIN statement, the other being to specify the ALL option in the CHAIN statement of the calling program. The COMMON statement may be included anywhere in the calling program, but is usually placed near its beginning. More than one COMMON statement may be specified in a program, but the same variables cannot be specified in more than one COMMON statement. Array variables are specified by appending "( )" to the array name.

See also **CHAIN**

Example

```
10 PRINT "Main program"
20 A$ = "Tom"
30 B$ = "Dick"
40 C$ = "Harry"
50 COMMON A$,B$,C$
60 CHAIN "A:COMMON2"
```

```
10 PRINT "The COMMON statement passes 3 variables to this pr
ogram"
20 PRINT "The first is ";A$
30 PRINT "The second is ";B$
40 PRINT "The third is ";C$
50 END
```

```
Main program
The COMMON statement passes 3 variables to this program
The first is Tom
The second is Dick
The third is Harry
Ok
```

CONT

Format CONT

Purpose Resumes execution of a program which has been interrupted by a STOP or END statement, or by pressing [CTRL] + [C] or the [STOP] key.

Remarks This command causes program execution to resume at the point at which it was interrupted. If execution was interrupted while a prompt ("?" or a user-defined prompt string) was being displayed by an INPUT statement, the prompt is displayed again when program execution resumes.

The CONT command is often used together with the STOP command or the [STOP] key during programming debugging. When execution is interrupted by the STOP statement or the [STOP] key, statements can be executed in the direct mode to examine or change intermediate values, then execution can be resumed by executing CONT (or by executing a GOTO statement in the direct mode to resume execution at a different line number). The CONT statement can also be used to resume execution of a program which has been interrupted by an error; however, program execution cannot be resumed if any changes are made in the program while execution is stopped.

Example

```
10 CLS
20 FOR J = 1 TO 500
30 X = J * 5
40 LOCATE 40,3 : PRINT X
50 NEXT J

                                        700
Break in 40
Ok
print x
 370
Ok
cont
```

COPY

Format COPY

Purpose Outputs the contents of the display to the printer.

Remarks The COPY statement outputs the contents of the PX-8's LCD screen to a dot-matrix printer. In screen modes 0, 1, and 2, this statement outputs the contents of the screen window to the printer in ASCII format. In screen mode 3, it outputs the contents of the real screen to the printer in bit image format. Pressing [CTRL] and [PF5] will perform the same function, interrupting the program to do so.

```
10 SCREEN 3
20 FOR N = 33 TO 159
30 PRINT CHR$(N);
40 IF N = 75 OR N = 117 THEN PRINT:PRINT
50 NEXT N
60 COPY
70 END
```

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJK
LMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstu
vwxyz{¦}~▵┼┴┬┤├─│┌┐└┘▒▀▄█●○♠♥♦♣♪♂♀✦¥大↑↓×÷±
```

COS

Format **COS(X)**

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

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

Example

```
PRINT COS (1.5)
.07073711
```

NOTE:
The value returned by this function will not be correct if (1) X is a single precision value which is greater than or equal to 2.7E7, or (2) if X is a double precision value which is greater than or equal to 1.2D17.

CSNG

Format **CSNG(X)**

Purpose Returns the single precision number obtained by conversion of the value of numeric expression X.

Remarks See the descriptions of the CDBL and CINT functions for conversion of numeric values to double precision or integer type numbers.

Example

```
PRINT CSNG(5.123456789#)
      5.12346
```

CSRLIN

Format **CSRLIN [(<function>)]**

Purpose Returns the number corresponding to the current vertical position of the cursor in the virtual screen or the vertical position of the first line of the screen window in the virtual screen.

Remarks The value of function must be in the range 0 to 255.

The CSRLIN function returns the line of the virtual screen on which the cursor lies when the function is called. The lines of the virtual screens are numbered as follows:

Screen modes Ø, 1, and 2:	1 to the maximum number of virtual screen lines specified in the SCREEN or WIDTH statement.
Screen mode 3:	1 to 8.

If CSRLIN or CSRLIN (Ø) is executed, the value returned is the vertical position of the cursor in the virtual screen.

If CSRLIN (X) is executed, where X is in the range 1-255, the position of the first line of the screen window in the virtual screen is returned.

The meaning and range of values are the same as in the LOCATE statement.

The following program illustrates this dynamically by printing out the positions of the cursor and window, for whichever combination of screen display you choose.

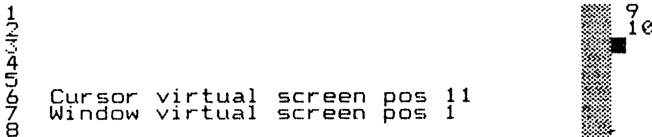
Example

```
10 CLS
20 INPUT "TYPE IN SCREEN NUMBER (0-3)" ; SC
30 INPUT "TYPE IN VIRTUAL SCREEN NUMBER (0 = vs 1, 1 = vs 2)
 ";VS
40 INPUT "FUNCTION KEY DISPLAY (0 = off, 1 = on) ";FK
50 SCREEN SC,VS,FK
60 CLS
70 FOR J = 1 TO 10 : PRINT J : NEXT
80 C  =  CSRLIN : D = CSRLIN(1)
90 LOCATE  5,6 : PRINT "Cursor virtual screen pos";C
100 LOCATE  5,7 : PRINT "Window virtual screen pos";D
110 LOCATE  1,C : PRINT CHR$(140);:GOTO 110
```

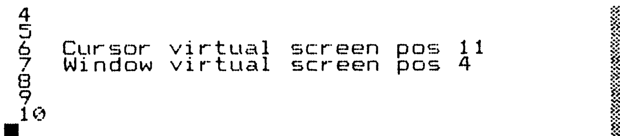
(Screen 0, virtual screen 1, function key display off)

```
4
5
6   Cursor virtual screen pos 11
7   Window virtual screen pos 4
8
9
10
■
```

(Screen 1, virtual screen 1, function key display off)



(Screen 2, virtual screen 1, function key display off)



NOTE:
Line 110 places a graphics character on the screen where the cursor was before the text was printed to illustrate the results indicated by line 80.

CVI/CVS/CVD

Format **CVI (<2-byte string>)**
CVS (<4-byte string>)
CVD (<8-byte string>)

Purpose These functions are used to convert string values into numeric values.

Remarks Numeric values must be converted to string values for storage in random access files. This is done using the MKI$, MKS$, or MKD$ functions depending on whether the numeric value being converted is an integer, single precision number, or double precision number. When such strings are then read back in from the file, they must be converted back into numeric values for display or use as operands in numeric operations. This is done using the CVI, CVS and CVD functions.

CVI returns an integer for a 2-byte string, CVS returns a single precision number for a 4-byte string, and CVD returns a double precision number for an 8-byte string.

See also **MKI$/MKS$/MKD$**, Chapter 5

Example

```
a$=mki$(12849)
Ok
?a$
12
Ok
?cvi(a$)
 12849
Ok
```

DATA

Format **DATA <list of constants>**

Purpose Lists numeric and/or string constants which are substituted into variables by the READ statement. (See the explanation of the READ statement.)

Remarks DATA statements are non-executable, and may be located anywhere in the program. Constants included in the list must be separated from each other by commas, and are substituted into variables upon execution of READ statements in the order in which they appear in the list. A program may include any number of DATA statements.

When more than one DATA statement is included in a program, they are accessed by READ statements in the order in which they appear (in program line number order); therefore, the lists of constants specified in DATA statements can be thought of as constituting one continuous list, regardless of the number of constants on each individual line or where the lines appear in the program.

Constants of any type (numeric or string) may be included in <list of constants>; however, the types of the constants must be the same as the types of variables into which they are to be substituted by the READ statements.

Numeric DATA statements can contain negative numbers but no operators. String constants must be enclosed in quotation marks if they include commas, colons or significant leading or trailing spaces; otherwise, quotation marks are not required.

Once the <list of constants> of a DATA statement has been read, it cannot be read again until a RESTORE statement has been executed.

See also **READ, RESTORE**

Example

```
10 CLS
20 FOR J = 1 TO 5
30 READ A$,B
40 PRINT A$,B
50 NEXT
60 END
70 DATA "ANGELA:ANGIE",12,"  BRIAN",-20,CHARLIE,39,DIANA,
-16,ERIC,34
```

```
ANGELA:ANGIE    12
  BRIAN        -20
CHARLIE         39
DIANA          -16
ERIC            34
Ok
```

DATE$

Format As a statement
DATE$ = "<MM>/<DD>/<YY>"

As a variable
X$ = DATE$

Purpose DATE$ is a system variable which contains the date of the PX-8's built-in calendar clock.

Remarks As a statement, DATE$ is used to set the date of the PX-8's calendar clock. <MM> is a number from 01 to 12 which indicates the month, <DD> is a number from 01 to 31 which indicates the day, and <YY> is a number from 00 to 99 which indicates the year. As a variable, DATE$ returns the date of the built-in clock in "MM/DD/YY" format.

See also **DATE, DAY**

DAY

Format

As a statement
DAY = <W>

As a variable
X% = DAY

Purpose

DAY is a system variable which maintains the day of the week of the PX-8's built-in calendar clock.

Remarks

As a variable, DAY returns the day of the week of the PX-8's calendar clock as a number from 0 to 6. Sunday is represented by 0, 6 is used to represent Saturday, and so forth.

The day can be set independently of the value assigned as the calendar date (by the DATE$ statement); therefore, as a statement DAY can be used to assign any number from 0 to 6 to the current day of the week. However, if the current day of the week is altered from the above representation, the System Display and other software will print out the day incorrectly. For example if you choose to assign the first day of January 1985 as 6, the system display will show Saturday when it should in fact be a Sunday.

Example

```
10 PRINT "Day is ";DAY
20 DAY=3
30 PRINT "Day is now";DAY
40 END

run
Day is  5
Day is now 3
Ok
```

DEF FN

Format **DEF FN<name>(<parameter list>) = <function definition>**

Purpose Used to define and name user-written functions.

Remarks A user defined function is a numeric or string expression which can be executed by BASIC programs in the same manner as intrinsic functions (e.g., TAN or SIN). When such a function is called, the variables specified as its arguments (either in the function definition or in the parameter list of the calling statement) are substituted into the expression and the equivalent value is returned as the result of the function.

<parameter list> comprises those variables in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

If a <parameter list> is included in the <function definition>, then a list with a corresponding number of parameters must be specified in the statement calling the function; the values of variables specified in the calling statement's parameter list are then substituted into the <parameter list> of the function definition on a one-to-one basis.

<function definition> is an expression that performs the operation of the function. It is limited to one program line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name.

A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the ar-

gument type does not match, a "Type mismatch" error occurs.

The DEF FN statement must be executed before the corresponding user function can be called: otherwise, an "Undefined user function" error will occur.

DEF FN statements cannot be executed in the direct mode.

Examples showing extensive use of the DEF FN command are shown in the example programs in the appendices. The following programs outline simpler applications.

Example

```
10 DEF FN SQ(X) = X * X    :' define the function SQ to give
20 '                          the square of a number
30 PRINT FN SQ(9)          :' print the square of 9
40 X = 12                  :' set the variable X equal to 12
50 PRINT FN SQ(X)          :' and print the square of X
60 Y = 10
70 PRINT FN SQ(Y)          :' use the value of the variable Y
80 '                          as a substitute for X
90 '
100 PRINT:PRINT
110 DEF FN NM(X,Y) = X * X + Y:' define a function NM to give
120 '                              a function of two numbers
130 PRINT FN NM(10,20)     :' print the value using 10 and 20
140 X = 5 : Y = 6
150 PRINT FN NM(X,Y)       :' Print the value of the function
160 '                         using the variables X and Y
170 PRINT FN NM(Y,X)       :' The values of the variables are
180 '                         used according to position and
190 '                         NOT VARIABLE NAME


run
 81
 144
 100


 120
 31
 41
Ok
```

The only inverse trigonometric function available is ARC TAN. DEF FN is useful to provide functions for ARC COS etc.,and the formulae for obtaining such functions are listed in Appendix E.

The following program illustrates the use with ARC SIN, and also in converting from radians to degrees. (See ATN for this computation.)

Example

```
10 DEF FN ARCSIN(X) = ATN (X / SQR (1-X * X) )   :' Defines a
20 '                  function to give the angle from its SINE
30 '
40 DEF FN DEG (X) = X * 45 / ATN(1)    :' function to convert
50 '                                      radians to degrees
60 CLS
70 INPUT "Type in SINE of angle "; X
80 R = FN ARCSIN (X) : D = FN DEG (R) :' Find the values of
the angle
90 PRINT "The angle whose SINE is "; X ; " is " ; R ;
"Radians or "; D ; "Degrees"


Type in SINE of angle ? .5
The angle whose SINE is .5  is  .523599 Radians or 30 Degrees
Ok
```

DEFINT/SNG/DBL/STR

Format

DEFINT <range(s) of letters>
DEFSNG <range(s) of letters>
DEFDBL <range(s) of letters>
DEFSTR <range(s) of letters>

Purpose

Declares the type of variables specified in <range(s) of letters>.

DEFINT as an INTEGER variable
DEFSNG as a SINGLE PRECISION variable
DEFDBL as a DOUBLE PRECISION variable, and
DEF STR as a STRING variable

Remarks

This statement defines the type of the specified variable or ranges of variables, making it unnecessary to indicate their type by appending the type definition characters (%, !, #, and $). Type declarations made using this statement apply to all variable names which begin with the letters included in <range(s) of letters>. For example, execution of DEFSTR A-C declares all variables whose names begin with the letters A, B, and C as string variables even though the declaration character $ is not appended to their names. Variable types specified in DEF statements do not become effective until those DEF statements have been executed; therefore, the BASIC interpreter assumes that all variables without type declaration characters are single precision variables until a type definition statement is encountered. When a DEF statement is encountered, variables without type definition characters are cleared if the first letter of their names is specified in that statement.

NOTE:
Note that trying to assign a numerical value to R when it has been declared as a string variable results in a "Type mismatch error".

DEF USR

Format **DEF USR[<digit>] = <integer expression>**

Purpose Specifies the starting address in memory of a user-written machine language program.

Remarks Machine language programs whose starting addresses are defined with the DEF USR statement can be used as functions in BASIC programs. This is done using the USR function; see the explanation of the USR function and Appendix D for more information. <digit> is a number from 0 to 9 by which the machine language program is identified when called with the USR function. If <digit> is not specified, 0 is assumed.

<integer expression> is the starting address of the machine language program. Up to 10 starting addresses (USR0 to USR9) may be concurrently defined; if more addresses are required, additional DEF USR statements may be executed to redefine starting addresses for any of USR0 to USR9.

Machine language programs used as subroutines by BASIC programs must be written into memory before they can be called; further, the starting address of the area into which machine language programs are written must be specified with the CLEAR statement.

See also **USR, CALL**

NOTE:
Appendix D describes how to use DEF USR.

DELETE

Format **DELETE [<line number 1>][-<line number 2>]**

Purpose Deletes specified lines of the program in the currently logged in BASIC program area.

Remarks If both <line number 1> and <line number 2> are present, all the lines from <line number 1> to <line number 2> inclusive will be deleted. If the second parameter is omitted, only the line specified in <line number 1> is deleted. If the first parameter is omitted, all lines from the beginning of the program to <line number 2> will be omitted.

An "Illegal function call" error will result if a specified line number does not exist or if a hyphen is specified without specifying the second line number.

Although DELETE can be used in a BASIC program, control always returns to the command level after execution.

If you wish to delete the last lines of a program, you cannot specify a line number greater than that of the last line of the program, otherwise an "Illegal function call" error will be printed and no action will be taken. Thus, if the last line number in a program is 190 and you wish to delete lines greater than 100, DELETE 1Ø1-19Ø is correct but DELETE 1Ø1-2ØØ will generate an error. However, line 101 does not have to exist.

Examples **DELETE 1Ø** will remove line 10.
DELETE 1Ø–9Ø will remove lines 10 to 90.
DELETE –9Ø will remove lines up to line 90.

DIM

Format **DIM <list of subscripted variables>**

Purpose Specifies the maximum range of array subscripts and allocates space for storage of array variables.

Remarks The DIM statement defines the extent of each dimension of variable arrays by specifying the maximum value which can be used as a subscript for each dimension; it also clears all variables in the specified array(s). For example, DIM A(25,50) defines a two-dimensional array whose individual variables are designated as A(N1,N2), where the maximum value of N1 is 25 and the maximum value of N2 is 50. Since the minimum value of a subscript is 0 (unless otherwise specified with the OPTION BASE statement), this array includes 26 × 51 = 1346 individual variables. Any attempt to access an array element with subscripts greater than those specified in the DIM statement for that array will result in a "Subscript out of range" error; if no DIM statement is specified, the maximum value which can be used for subscripts is 10. Once an array has been dimensioned with the DIM statement it cannot be redimensioned until it has been erased by a CLEAR or ERASE statement.

See also **ERASE, OPTION BASE, CLEAR**

Example 1 **10 DIM A(20, 15)**
Defines two-dimensional array A and specifies 20 and 15 as its maximum subscript values. Unless otherwise specified by a DEF <type> statement, BASIC will handle this as a single precision numeric array.

Example 2 **10 DIM A$(30)**
Defines one-dimensional string array A$ and specifies 30 as its maximum subscript value.

Example 3 **10 DIM G%(25), F%(25)**
Defines one-dimensional arrays G and F and specifies 25 as the maximum values of their subscripts.

DSKF

Format **DSKF(<drive name>)**

Purpose Returns the amount of space available for storage of files or programs in the disk device specified by <drive name>.

Remarks This function returns the amount of unused space in the disk device specified by <drive name> as an integer which indicates the amount of free space in kilobytes. The <drive name> must be specified as a string expression from "A:" to "F:". However, note that a "Device unavailable" error will occur if the specified device is not connected; further, the value returned for the microcassette drive (usually drive "H:") has no meaning.

Example PRINT DSKF ("A:") will return the amount of space available for drive A:.

EDIT

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

Purpose Places BASIC in the edit mode for editing of the program in the program area which is currently logged.

Remarks The EDIT command is used in direct mode to place the currently selected virtual screen in the edit mode. If <line number> is specified, BASIC clears the screen and prints the specified program line with the cursor positioned on the first character of the line; an error message will be displayed if the line does not exist. If no line number is specified the edit mode will be entered at the first line of the program.

The edit mode is an enhanced form of the normal screen editor. The cursor keys in combination with the [CTRL] key allow the wole of the program to be scrolled regardless of the number of lines in the virtual screen. This prevents continually having to list sucessive portions of the program and is particularly useful if the editing is being attempted in screen mode 3. If an attempt is made to enter the edit mode when the screen is in screen mode 2 and the width of the screen in which the cursor lies is less than 38 columns, an "Illegal function error" will be generated.

When the EDIT command is executed while in screen modes 0, 1 or 2, the edit mode is only entered on the current screen. The other virtual screen remains in the normal screen editor mode. By changing the screen using [CTRL] + [→] or [CTRL] + [←] and executing the EDIT command on the other virtual screen both screens can be used in the edit mode. If the edit mode is terminated on one screen using [ESC], [CLR] or [CTRL] + [L], it will not be terminated on the other screen. However, if a direct command (other than EDIT) is used it will be terminated on both virtual screens.

To exit the edit mode, use the [ESC] key and clear the screen with [CLR] or [CTRL] + [L], or execute any direct mode command other than EDIT. The PX-8 remains in the edit mode if the virtual screens are switched using [CTRL] + [→] or [CTRL] + [←].

See also Chapter 2 Section 2.5 for an extensive explanation of using the Screen Editor and EDIT.

END

Format END

Purpose Stops program execution, closes all files and returns BASIC to the command level.

Remarks END statements may be included anywhere in a program to stop execution. However, it is not necessary to place an END statement in the last line of the program if the last program line is always the last line executed.

The END statement is often used together with the IF ... THEN ... ELSE statement to terminate program execution under specific conditions.

As with the STOP statement, program execution terminated by the END statement can be resumed by executing a CONT command. However, the END statement does not result in display of a BREAK message.

It often happens that subroutines are placed at the end of a program. An END command is often placed before the first such subroutine so that the program does not continue into the subroutine when the main part of the program has been completed. The following example illustrates this.

See also STOP

Example

```
10 GOSUB 50
20 PRINT "Having executed the subroutine at line 50"
30 PRINT "this program halts at the END statement on line 40"
40 END
50 PRINT "The program is now executing the subroutine at line
50"
60 PRINT
70 RETURN
run
The program is now executing the subroutine at line 50

Having executed the subroutine at line 50
this program halts at the END statement on line 40
Ok
```

EOF

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

Purpose Returns a value indicating whether the end of a sequential file has been reached during sequential input.

Remarks During input from a sequential file, an "Input past end" error will occur if INPUT# statements are executed against that file after the end of the file has been reached. This can be prevented by testing whether the end of file has been reached with the EOF function.

<file number> is the number under which the file was opened. The function will return "false" (0) if the end of file has not been reached, and "true" (−1) if the end of file has been reached.

Example

```
10 OPEN "O",#1,"TEST"
20 FOR J = 1 TO 5
30 PRINT #1,J
40 NEXT
50 CLOSE #1
60 OPEN "I",#1,"TEST"
70 IF EOF(1) THEN 110
80 INPUT #1,J
90 PRINT J,
100 GOTO 70
110 PRINT "The end of the file has been reached"
120 CLOSE #1

run
 1              2              3              4              5
The end of the file has been reached
Ok
```

ERASE

Format **ERASE <list of variables>**

Purpose Cancels array definitions made with the DIM statement.

Remarks The ERASE statement erases the specified variable arrays from memory, allowing them to be redimensioned and freeing the memory they occupied for other purposes.

An "Illegal function call error" will result if an attempt is made to erase a non-existent array.

It is not possible to redimension an array without destroying it completely using ERASE.

See also **DIM**

ERL

Format **ERL**

Purpose Used in an error processing routine to return the line number of the program line at which an error occurred during command or statement execution.

Remarks The ERL function returns the line number of the command/statement causing an error during program execution.

If an error occurs during execution of a command or statement in the direct mode, this function returns the number 65535 as the line number.

The ERL function is normally used with IF ... THEN statements in an error processing routine to control the flow of program execution when an error occurs.

See also ERROR, ON ERROR GOTO, RESUME, ERR

Example See under ERROR.

ERROR

Format **ERROR <integer expression>**

Purpose Simulates the occurrence of a BASIC error. Also allows error codes to be defined by the user.

Remarks The value of <integer expression> must be greater than 0 and less than 255. If the value specified equals one of the error codes which is used by BASIC (see Appendix A), occurrence of that error is simulated and the corresponding error message is displayed. If the value specified is not defined in BASIC, the message "Unprintable error" is displayed.

You can also use the ERROR statement to define your own error messages; this is illustrated in the example below. When using the ERROR statement for this purpose the value of <integer expression> must be a number which does not correspond to any error code which is defined in BASIC. Such user-defined error codes can then be handled in an error processing routine.

See also **ERR, ERL, ON ERROR GOTO, RESUME**

Example 1

```
10 ON ERROR GOTO 80
20 X = "FRED"
30 X$(20) = 23
40 ERROR 199
50 PRINT
60 PRINT "There are no more errors to demonstrate"
70 END
80 IF ERR = 13 THEN PRINT "There is a Type Mismatch in line
20":RESUME 30
90 IF ERL = 30 THEN PRINT "There is a Subscript error in lin
e 30":RESUME 40
100 IF ERR = 199 THEN PRINT "I defined this error number 199
 myself":RESUME 50

There is a Type Mismatch in line 20
There is a Subscript error in line 30
I defined this error number 199 myself

There are no more errors to demonstrate
Ok
```

ERR

Format **ERR**

Purpose Used in an error processing routine to return the code of an error occurring during program execution.

Remarks The ERR function returns the code of errors occurring during command or statement execution.

As with the ERL function, the ERR function is normally used with IF ... THEN statements in an error processing routine to control the flow of program execution when an error occurs. An example of program control using the ERR function is shown below.

See also **ERL, ON ERROR GOTO, RESUME**

Example See under ERROR

IF ERR=11 THEN RESUME 1000

When included in an error processing routine, this line causes program execution to resume at line 1000 if the error being processed is a "Division by zero" error.

EXP

Format EXP(X)

Purpose Returns the value of the natural base *e* to the power of X.

Remarks The value specified for X must not be greater than 87.3365; otherwise an "Overflow" error will occur.

To raise another number to a power use the operator "^". See the program below for an example.
EXP can also be used to obtain antilogarithms.

See also LOG

Example

```
10 FOR J = 0 TO 80 STEP 10
20 LPRINT "e^";J;"=";EXP(J)
30 NEXT
40 LPRINT
50 LPRINT "The cube of 2 is: ";2^3
```

```
e^ 0 = 1
e^ 10 = 22026.5
e^ 20 = 4.85165E+08
e^ 30 = 1.06865E+13
e^ 40 = 2.35385E+17
e^ 50 = 5.1847E+21
e^ 60 = 1.142E+26
e^ 70 = 2.51544E+30
e^ 80 = 5.54063E+34

The cube of 2 is:  8
```

FIELD

Format **FIELD[#]<file number>, <field width> AS <string variable>, <field width> AS <string variable>, ...**

Purpose Assigns string variables to specific positions in a random file buffer.

Remarks When a random access file is opened, a buffer is automatically reserved in memory which is used for temporary storage of data while it is being transferred between the storage medium (RAM disk or other disk device) and the computer's memory. Data is read into this buffer from the storage medium during input, and is written from the buffer to the storage medium during output. During input, data is read into the buffer upon execution of a GET statement, and is output to the buffer upon execution of a PUT statement.

However, before any GET or PUT statement can be executed, a FIELD statement must be executed to assign positions in the file access buffer to specific variables. Doing this causes data substituted into that variable by a PUT statement to be stored in assigned positions in the file access buffer, rather than in normal string space. Conversely, items brought into the buffer from the file by a GET statement are accessed by checking the contents of variables to which positions in the buffer have been assigned. See Chapter 5 for a detailed description of procedures for accessing random access files.

The <file number> which is specified in the FIELD statement is that under which the file was opened. <field width> specifies the number of positions which is to be allocated to the specified <string variable>. For example:

```
FIELD 1,20 AS N$,10 AS ID$,40 AS ADD$
```

assigns the first 20 positions of the buffer to string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$.

The total number of positions assigned to variables by the FIELD

statement cannot exceed the record length that was specified when the file was opened; otherwise, a "FIELD overflow" error will occur (the default record length is 128 bytes).

If necessary, any number of FIELD statements may be executed for the same file. If more than one such statement is executed, all assignments made are effective concurrently.

See also **GET, LSET/RSET, OPEN, PUT**
For full details of use of the FIELD command see Chapter 5.

Example

```
10 OPEN "R",#1,"E:EMPDAT.DAT"
20 '   Opens file "EMPDAT.DAT" in drive E: as a random
30 '   access file.
40 FIELD #1,6 AS NO$,2 AS DP$,120 AS RE$
50 '   Assigns the first six bytes of file buffer #1 to
60 '   variable NO$, the second two bytes to variable
70 '   DP$, and the last 120 bytes to variable RE$.
80 '   Data can now be placed in the buffer with the
90 '   LSET/RSET statement, then stored in the file with
100 '  the PUT statement. Or, data can be brought into
110 '  the buffer from the file with the GET statement,
120 '  after which the contents of the buffer variables
130 '  can be displayed with the PRINT statement or
140 '  used for other processing.
```

NOTE:
Once a variable name has been specified in a FIELD statement, only use RSET or LSET to store data in that variable. FIELDing a variable name assigns it to specific positions in the random file buffer; using an INPUT or LET statement to store values to FIELDed variables will cancel this assignment and reassign the names to normal string space.

FILES

Format **FILES[<ambiguous file name>]**

Purpose Displays the names of files satisfying the <ambiguous file name>.

Remarks If <ambiguous file name> is omitted, this command displays the names of all files in the disk device which is the currently selected drive.

When specified, <ambiguous file name> is composed of the following elements.

[<drive name>:][<file name>[<.extension>]]

In this case, the FILES command lists the names of all files which satisfy the <ambiguous file name>. An ambiguous file name is used to find files whose file names and/or extensions include common character strings. The form for specifying ambiguous file names is similar to that used for normal file descriptors, except that the question mark (?) can be used as a wild card character to indicate any character in a particular position, and the asterisk (*) can be used as a wild card character to indicate any combination of characters for a file name or extension. Some examples of ambiguous file names are shown below.

If there are no files on a particular disk, the "File not found" message will be displayed.

Example 1 **FILES "A:L???.BAS"**
Displays all files on disk drive A: whose file names begin with L, are up to four characters long and are also BASIC files.

Example 2 **FILES "L???????.BAS"**
Displays the names of all files on the currently active disk device whose file names begin with the letter L and whose extensions are ".BAS", because all possible character positions are used with the '?' wild card.

Example 3 **FILES "D:*.*" or FILES "D:"**
Displays the names of all files on the disk in drive D.

Example 4 **FILES "D:D???.*"**
Displays the names of all files on the disk in drive D which begin with the letter D and which include not more than four characters in the file name.

Example 5 **FILES "D:*.COM"**
Displays all the files on disk drive D: which have .COM as their extension.

PRINTING FILE NAMES FROM BASIC

In order to print the directory of a disk from BASIC, a screen dump must be carried out. If there are more files than the screen will show at one time, scroll the screen using the cursor keys and perform multiple screen dumps. Screen 0 is the best screen to use for screen dumps of files because it can be scrolled, and will hold the greatest number of file names at any one time.

FIX

Format FIX(X)

Purpose Returns the integer portion of numeric expression X.

Remarks The value returned by FIX(X) is equal to the sign of X times the integer portion of the absolute value of X. Thus −1 is returned for −1.5, −2 is returned for −2.33333 and so forth. Compare with the INT function, which returns the largest integer which is less than or equal to X.

See CINT for an explanation and comparison of CINT, FIX and INT, and also other problems associated with their use.

FOR...NEXT

Format

FOR <variable> = <expression 1> TO <expression 2> [STEP <expression 3>]

.

.

.

NEXT [<variable>][,<variable>...]

Purpose

The FOR...NEXT statement allows the series of instructions between FOR and NEXT to be repeated a specific number of times.

Remarks

This statement causes program execution to loop through the series of instructions between FOR and NEXT a specific number of times. The number of repetitions is determined by the values specified following FOR for <expression 1>, <expression 2>, and <expression 3>.

<variable> is used as a counter for keeping track of the number of loops which have been made. The initial value of this counter is that specified by <expression 1>. The ending value of the counter is that specified by <expression 2>. Program lines following FOR are executed until the NEXT statement is encountered, then the counter is incremented by the amount specified by <expression 3>. An increment of 1 is assumed if STEP <expression 3> is not specified; however, a negative value must be specified following STEP if the value of <expression 2> is less than that of <expression 1>.

Next, a check is made to see if the value of the counter is greater than the value specified by <expression 2>. If not, execution branches back to the statement following the FOR statement and the sequence is repeated. If the value is greater, execution continues with the statement following NEXT. Otherwise statements in the loop are skipped and execution resumes with the first statement following NEXT.

FOR...NEXT loops may be nested; that is, one FOR...NEXT loop may be included within the body of another one. When loops are nested, different variable names must be specified for <variable>

at the beginning of each loop. Further, the NEXT statement for inner loops must appear before those for outer ones.

If nested loops end at the same point, a single NEXT statement may be used for all of them. In this case, the variable names must be specified following NEXT in reverse order to that in which they appear in the FOR statements of the nested loops; in other words, the first variable name following NEXT must be that which is specified in the nearest preceding FOR statement, the second variable name following NEXT must be that which is specified in the next nearest preceding FOR statement, and so forth.

If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" message is displayed and execution is aborted. If a FOR statement without a corresponding NEXT statement is encountered, a "FOR without NEXT" message is displayed and execution is aborted.

See also **WHILE...WEND**

Example

```
10 PRINT "1. Single loop incremented in steps of 2"
20 FOR J = 1 TO 9 STEP 2
30 PRINT J,
40 NEXT
50 FOR L = 1 TO 100:NEXT
60 PRINT:PRINT "2. Single loop with default increment of 1"
70 FOR K = 1 TO 5
80 PRINT K,
90 NEXT
100 FOR L = 1 TO 100:NEXT
110 PRINT:PRINT "3. Nested loop with two NEXT statements"
120 FOR J = 1 TO 5
130 FOR K = 1 TO 3
140 PRINT J;"/";K,
150 NEXT:NEXT
160 FOR L = 1 TO 100:NEXT
170 PRINT:PRINT "4. Nested loop with one NEXT statement
    specifying both variable
180 FOR J = 1 TO 5
190 FOR K = 1 TO 3
200 PRINT J;"/";K,
210 NEXT K,J
```

```
1. Single loop incremented in steps of 2
 1            3            5            7            9

2. Single loop with default increment of 1
 1            2            3            4            5

3. Nested loop with two NEXT statements
 1 / 1        1 / 2        1 / 3        2 / 1        2 / 2
 2 / 3        3 / 1        3 / 2        3 / 3        4 / 1
 4 / 2        4 / 3        5 / 1        5 / 2        5 / 3

4. Nested loop with one NEXT statement specifying both
   variables
 1 / 1        1 / 2        1 / 3        2 / 1        2 / 2
 2 / 3        3 / 1        3 / 2        3 / 3        4 / 1
 4 / 2        4 / 3        5 / 1        5 / 2        5 / 3
```

FRE

Format **FRE(X)**
FRE(X$)

Purpose Returns the number of bytes of memory which are not being used by BASIC.

Remarks The value returned by this function provides an indication of the number of bytes of memory which are available for use as string variables, numeric variables or BASIC program text. However, the value returned also includes a work area which is used by BASIC during program execution and thus does not provide a direct indication of the number of unused bytes which are available for this purpose.

The arguments of this function have no meaning; FRE returns the same value regardless of whether a string expression or a numeric expression is specified as the argument.

When a FRE(X$) function is executed, the BASIC interpreter has to perform 'garbage collection'. When strings are defined BASIC stores them in memory; they are often changed frequently as the program is executed. Every so often BASIC has to collect the values of the strings which are still valid and erase the unwanted ones, otherwise there will be no space for more strings to be stored. If a program executes a great deal of string handling there can be times while the program is running that it appears to halt because it is carrying out this 'garbage collection'. By executing FRE(X$), BASIC is forced to carry out this operation. A line which checks the free memory is often placed throughout a BASIC program so that by forcing garbage collection in small steps rather than one big one the long gap can be avoided.

FRE(X) carries out garbage collection for numerical variables in a manner similar to that in which FRE(X$) performs garbage collection for string variables.

Example The following program shows how assigning a value to a variable decreases the amount of free memory available.

Note that when the program is run, line 10 shows there are 23665 bytes of memory. When line 20 has allocated a value to variable B, available memory is reduced to 23657 bytes. After string manipulation, line 70 shows the memory is reduced a great deal further. In executing the FRE(X$) command, much of this memory can be retrieved.

```
10 PRINT "Free memory for programs and variables using FRE(X)
is";FRE(X)
20 B= 10
30 PRINT "Free memory for programs and variables using FRE(X)
is";FRE(X)
40 FOR J = 65 TO 75
50 A$ = A$ + CHR$(J)
60 NEXT J
70 PRINT "Free memory for programs and variables using FRE(X)
is";FRE(X)
80 PRINT "and using FRE(X$) is ";FRE(X$)

run
Free memory for programs and variables using FRE(X) is 23665
Free memory for programs and variables using FRE(X) is 23657
Free memory for programs and variables using FRE(X) is 23565
and using FRE(X$) is  23631
Ok
```

GET

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

Purpose The GET statement reads a record into a random file access buffer from a random disk file.

Remarks This statement reads a record into a random file access buffer from the corresponding random access file. <file number> is the number under which the file was opened and <record number> is the number of the record which is to be read into the random file buffer. Both <file number> and <record number> must be specified as integer expressions.

If <record number> is omitted the record read is that following the one read by the preceding GET statement. The highest possible record number is 32767.

Note that records must be read sequentially if the file being accessed is a microcassette file. Also note that the FIELD statement must be executed to assign space in the random file buffer to variables prior to executing a GET statement.

See also **FIELD, LSET/RSET, OPEN, PUT**
For full details of use of the **FIELD** command see Chapter 5.

Example

```
10 'Lines 20-80 create a random data file with 5 records.
20 OPEN"R",#1,"A:TESTDAT",10
30 FIELD#1,10 AS A$
40 FOR I=1 TO 5
50 PRINT"Type 1-10 characters for record";I;:INPUT B$
60 LSET A$=B$
70 PUT#1,I
80 NEXT
90 '
100 'Lines 110 to read specified records from the file.
110 INPUT"Enter record no.(1-5)";R
120 GET#1,R
130 PRINT A$
140 GOTO 110
```

```
run
Type 1-10 characters for record 1 ? Alfie
Type 1-10 characters for record 2 ? Betty
Type 1-10 characters for record 3 ? Charlie
Type 1-10 characters for record 4 ? Dean
Type 1-10 characters for record 5 ? Edward
Enter record no.(1-5)?



Enter record no.(1-5)? 5
Edward
Enter record no.(1-5)? 4
Dean
Enter record no.(1-5)? 3
Charlie
Enter record no.(1-5)? 2
Betty


Enter record no.(1-5)? 1
Alfie
Enter record no.(1-5)?
```

GOSUB...RETURN

Format

GOSUB <line number>

•

•

RETURN

Purpose

The GOSUB and RETURN statements are used to branch to and return from subroutines.

Remarks

The GOSUB statement transfers execution to the program line number specified in <line number>. When a RETURN statement is encountered, execution then returns to the statement following the one which called the subroutine, either on the same line or the next one. Subroutines may be located anywhere in a program; however, it is recommended that they be made readily distinguishable from the main routine. Since a "RETURN without GOSUB" error will occur if a RETURN statement is encountered without a corresponding GOSUB statement, care must be taken to ensure that execution does not move into a subroutine without it having been called. This can be avoided with the STOP, END or GOTO statements; the STOP and END statements halt execution when encountered, while the GOTO statement can be used to route execution around the subroutine.

Subroutines may include more than one RETURN statement if the program logic dictates a return from different points in the subroutine. Further, a subroutine may be called any number of times in a program, and one subroutine may be called by another. Nesting of subroutines in this manner is limited only by the amount of stack space available for storing return addresses. An "Out of memory" error will occur if the stack space is exceeded. The stack space size may be changed with the CLEAR statement if it is insufficient to accomodate the number of levels of subroutine nesting used by a program, but this must be executed at the beginning of the program outside the subroutines as CLEAR destroys all references to RETURN line numbers on the stack and you will not be able to RETURN from a subroutine if CLEAR is used within it.

See also **CLEAR**

Example

```
10 GOSUB 70
20 PRINT "Resuming execution after returning from subroutine
at line 70"
30 GOSUB 60
40 PRINT "Resuming execution after return from the nested
subroutines startin at line 50"
50 END
60 GOSUB 70:RETURN
70 PRINT "Now executing the subroutine at line 70"
80 RETURN
```

```
run
Now executing the subroutine at line 70
Resuming execution after returning from subroutine at line 70
Now executing the subroutine at line 70
Resuming execution after return from the nested subroutines st
arting at line 50
Ok
```

GOTO or GO TO

Format GOTO <line number>
GO TO <line number>

Purpose Unconditionally transfers program execution to the program line specified by <line number>.

Remarks This statement is used to make unconditional "jumps" from one point in a program to another. If the first statement on the line specified by <line number> is an executable statement (other than a REM or DATA statement), execution resumes with that statement; otherwise, execution resumes with the first executable statement encountered following <line number>.

It is also possible to leave out the GOTO in conditional statements, e.g., line 20 in the following program.

An "Undefined line number" error will occur if <line number> refers to a non-existent line.

GOTO can also be used in direct mode. In this case, variables are not destroyed (unlike RUN <line number> which does destroy variables), and can in fact be assigned from the command line in the direct mode.

Example

```
10 READ A,B:'Reads numbers into A and B from line 70
20 IF A=0 AND B=0 THEN 80:'Jumps to line 80 if A and B
25 '                          both equal 0.
30 PRINT "A=";A,"B=";B
40 S=A*B
50 PRINT "Product is";S
60 GOTO 10                   :'Jumps to line 10.
70 DATA 12,5,8,3,9,0,0,0
80 END

run
A= 12          B= 5
Product is 60
A= 8           B= 3
Product is 24
A= 9           B= 0
Product is 0
Ok
```

HEX$

Format **HEX$(X)**

Purpose Returns a character string which represents the hexadecimal value of X.

Remarks The value of the numeric expression specified in the argument must be a number in the range from −32768 to 65535. If the value of the expression includes a decimal fraction, it is rounded to the nearest integer before the string representing the hexadecimal value is returned.

To convert from hexadecimal to decimal use &H before the hexadecimal value. This will give a numerical constant.

HEX$ is a string.
&H is a numeric.

See also OCT$

Example 1

```
10 CLS
20 LOCATE 1,1: PRINT "Convert Hex to Decimal (H) or ":
PRINT "Decimal to Hex (D)"
30 INPUT C$
40 IF C$ = "H" OR C$ = "h" THEN GOSUB 100 ELSE IF C$ =
"D" OR C$ = "d" THEN GOSUB 200 ELSE 20
50 INPUT "Any more (yes/no) "; YN$
60 IF LEFT$(YN$,1) <> "y" AND LEFT$(YN$,1) <> "Y" THEN
END ELSE RUN
100 INPUT "Type in number in hexadecimal ";H$
110 V  = VAL ("&H" + H$)
120 PRINT V
130 RETURN
200 INPUT "Type in number in decimal "; D
210 V$ = HEX$ (D)
220 PRINT V$
230 RETURN
```

```
run
Convert Hex to Decimal (H) or
Decimal to Hex (D)
? h
Type in number in hexadecimal ? 4d
 77
Any more (yes/no) ?


Convert Hex to Decimal (H) or
Decimal to Hex (D)
? d
Type in number in decimal ? 34
22
Any more (yes/no) ?
```

IF..THEN [...ELSE]/IF...GOTO

Format Possible alternatives are

IF <logical expression> THEN <statement> [ELSE <statement>]

IF <logical expression> THEN <line No.> [ELSE <line No.>]

IF <logical expression> THEN <statement> [ELSE <line No.>]

IF <logical expression> THEN <line No.> [ELSE <statement>]

IF <logical expression> GOTO <line No.> [ELSE <statement>]

IF <logical expression> GOTO <line No.> [ELSE <line No.>]

Purpose Changes the flow of program execution according to the results of a logical expression.

Remarks The THEN or GOTO clause following <logical expression> is executed if the result of <logical expression> is true (– 1). Otherwise, the THEN or GOTO clause is ignored and the ELSE clause (if any) is executed; execution then proceeds with the next executable statement.

When a THEN clause is specified, THEN may be followed by either a line number or one or more statements. Specifying a line number following THEN causes program execution to branch to that program line in the same manner as with GOTO. When a GOTO clause is specified, GOTO is always followed by a line number.

IF..THEN...ELSE statements may be nested by including one such statement as a clause in another. Such nesting is limited only by the maximum length of the program line.

For example, the following is a correctly nested IF...THEN statement

20 IF X>Y THEN PRINT "X IS LARGER THAN Y" ELSE IF Y>X THEN PRINT "X IS SMALLER THAN Y" ELSE PRINT "X EQUALS Y"

Because of the logical structure of the line only one of the strings can be printed.

If a statement contains more THEN than ELSE clauses, each ELSE clause is matched with the nearest preceding THEN clause. For example, the following statement displays "A=C" when A=B and B=C. If A=B and B< >C it will display "A< >C". And if A< >B, it displays nothing at all.

IF A=B THEN IF B=C THEN PRINT "A=C" ELSE PRINT "A< >C"

It is also possible to have a number of statements where <statement> occurs in the above format expressions. For example it is common to have a line such as:

IF A=2 THEN B=3:C=7:A$="ORANGE"

Only if A has the value 2 will B be set equal to 3. The value of C will also only be set equal to 7 and A$ given the value "ORANGE" if A=2. If the expression "A=2" is false then all the rest of the line will be ignored.

This also applies if the sequence of statements exist in a line such as the following:

IF A=2 THEN PRINT "TRUE":B=5:A$="APPLES":GOTO 200 ELSE PRINT "FALSE":B=7:A$="PEARS":GOTO 300

In this case if A has the value 2 the expression "A=2" is true and the variable B is made equal to 5, the value "APPLES" is assigned to A$ and the program branches to line 200. However, if A does not have the value 2, the expression "A=2" is false and the commands following ELSE are executed; B is set equal to 7, A$ is as-

signed the value "PEARS" and the program will branch to line 300.

When using IF together with a relational expresion which tests for equality, remember that the results of arithmetic operations are not always exact values. For example, the result of the relational expression SIN(1.5708)=1 is false even though "1" is displayed if PRINT SIN(1.5708) is executed. Therefore, the relational expression should be written in such a way that computed values are tested over the range within which the accuracy of such values may vary. For example, if you are testing for equality between SIN(1.5708) and 1, the following form is recommended:

IF ABS(1 – SIN(1.5708)) < 1.0E – 6 THEN...

This relational expession returns "true".

Example

```
10 SCREEN 0,0,0:CLS
20 RANDOMIZE VAL(RIGHT$(TIME$,2))
30 '     Reinitializes the sequence of numbers returned
40 '     by the RND function
50 '
60 PRINT "Guess what number I am thinking of."
70 '
80 N=INT(RND(1)*9999)
90 '      Generates a random 4-digit number between
100 '     0 and 9999 and stores it in variable N
110 '
120 INPUT"Enter your guess";G
130 I=I+1
140 '     Keeps track of the number of guesses
150 '
160 IF G=N THEN PRINT "That's just right--in";I;"guesses!":E
LSE IF G<N THEN PRINT "You're too low! Try again.":GOTO 120:
ELSE PRINT "Sorry--you're too high! Try again.":GOTO 120
170 '     Displays the first message and the number
180 '     of guesses you have made if you correctly
190 '     guess the number generated on line 40; if
200 '     your guess is too low, displays the second
210 '     message and branches to line 50; if your
220 '     guess is too high, displays the third
230 '     message and branches to line 50
```

INKEY$

Format INKEY$

Purpose Checks the keyboard buffer during program execution and returns a null string if no key has been pressed.

Remarks INKEY$ returns a null string if the keyboard buffer is empty. If any key whose code is included in the ASCII code table has been pressed, INKEY$ reads that character from the keyboard buffer and returns it to the program. Characters read from the keyboard buffer by INKEY$ are not displayed on the screen.

INKEY$ simply examines the keyboard buffer. If does not wait for a key to be pressed. If this function is required, use INPUT$(1).

See also INPUT$

Example

```
10 SCREEN 0,0,0
20 WIDTH 80,8:CLS
30 X=1:Y=1:LOCATE X,Y:PRINT "*";:'Displays an asterisk
40 '                                in the upper left
50 '                                corner of the screen.
60 '
70 A$=INKEY$:IF A$="" THEN 70
80 '     Checks for input from the keyboard; repeats
90 '     until input is detected.
100 '
110 ON INSTR(CHR$(30)+CHR$(31)+CHR$(29)+CHR$(28),A$) GOSUB 2
50,300,350,400
120 '     Checks whether the key pressed is one of the
130 '    cursor control keys; if so, goes to the
140 '    corresponding subroutine.  Otherwise,
150 '    continues to the GOTO statement on the line
160 '    below.
170 '
180 GOTO 70:'Transfers execution to line 70.
190 '
200 'The four subroutines below move the asterisk
210 'in the direction indicated by the arrow on
220 'the applicable cursor key.
230 '
240 'Move asterisk up
```

```
250 Y=Y-1:IF Y<1 THEN Y=8
260 LOCATE X,Y:PRINT "*";:IF Y=8 THEN LOCATE X,1:PRINT " ":E
LSE LOCATE X,Y+1:PRINT " "
270 RETURN
280 '
290 'Move asterisk down
300 Y=Y+1:IF Y>8 THEN Y=1
310 LOCATE X,Y:PRINT "*";:IF Y=1 THEN LOCATE X,8:PRINT " ";:
ELSE LOCATE X,Y-1:PRINT " "
320 RETURN
330 '
340 'Move asterisk left
350 X=X-1:IF X<1 THEN X=80
360 LOCATE X,Y:PRINT "*";:IF X=80 THEN LOCATE 1,Y:PRINT " ";
:ELSE LOCATE X+1,Y:PRINT " ";
370 RETURN
380 '
390 'Move asterisk right
400 X=X+1:IF X>80 THEN X=1
410 LOCATE X,Y:PRINT "*";:IF X=1 THEN LOCATE 80,Y:PRINT " ";
:ELSE LOCATE X-1,Y:PRINT " ";
420 RETURN
```

INP

Format **INP (J)**

Purpose Returns one byte of data from machine port J.

Remarks The machine port number specified for J must be an integer expression in the range from 0 to 255.

The full use of this command is beyond the scope of this manual. Please see the OS Reference Manual for details of the ports.

See also **OUT**

Example

```
10 'See I/O port &H02 if LED is ON
20 A=INP(&H2)
30 LED=A AND &H4       :'See BIT 2 status
40 IF LED=0 GOTO 60    :'If LED is OFF, set LED
50 END
60 OUT &H2,&H4         :'Set LED
```

INPUT

Format **INPUT[;"<prompt string>"]** |;| **<list of variables>**
|,|

Purpose Makes it possible to substitute values into variables from the keyboard during program execution.

Remarks Program execution pauses when an INPUT statement is encountered to allow data to be substituted into variables from the keyboard. One data item must be typed in for each variable name specified in <list of variables>, and each item typed in must be separated from the following one by a comma. If any commas are to be included in a string substituted into a given variable, that string must be enclosed in quotation marks when it is typed in from the keyboard. The same applies to leading and trailing spaces; leading and trailing spaces are not substituted into a string variable by the INPUT statement unless the string is enclosed in quotation marks.

If the number or type of items entered is incorrect, the message "?Redo from start" is displayed, followed by the prompt string (if any). When this occurs, all the data items must be re-entered. No values are substituted into variables until a correct response has been made to all the items of the list.

When BASIC executes the line a question mark prompt is displayed if no prompt string is specified. However, if a prompt string is specified, the string is displayed, but whether a question mark is displayed depends on the character before the list of variables. It is possible to enter a comma or a semi-colon before the list of variables. In the latter case, a question mark will be displayed. If the prompt string is followed by a comma, the question mark is suppressed. If no prompt string is given it is not possible to suppress the question mark.

The optional semicolon following INPUT prevents the cursor from advancing to the next line when the user types a [RETURN] on completion of entry of the data. The next PRINT statement or error statement will be printed directly after the last character input by the user before pressing [RETURN].

When more than one variable name is specifed in <list of variables>, each variable name must be separated from the following one by a comma. Items entered in response to an INPUT statement are substituted into the variables specified in <list of variables> when the RETURN key is pressed. The user must input each variable and separate it from the following one by a comma. If the user tries to press RETURN after each variable, a "?Redo from start" message will be printed, and the user will have to begin at the first item of the list of variables. The values entered are only substituted into the variables when the RETURN key is pressed at the end of the list.

When the RETURN key is pressed for a single item or for the last item of a list, the variable is set to a null string if it is a string variable and to zero if it is a numeric variable.

Example

```
10  INPUT A                   :'Inputs value from keyboard
20                            :'into A, then moves cursor
30                            :'to next line.
40                            :'
50  INPUT;B                   :'Inputs value into B and
60                            :'keeps cursor on current
70                            :'line.
80                            :'
90  INPUT"Enter C",C          :'Displays prompt without
100                           :'question mark and inputs
110                           :'value into C.
120                           :'
130  INPUT"Enter D,E";D,E     :'Displays prompt and
140                           :'question mark and inputs
150                           :'values into D and E.
160                           :'
170  INPUT;"Enter F,G";F,G:'Displays prompt, inputs
180                           :'values into F and G, and
190                           :'keeps cursor on current line.
200  PRINT "END"
```

INPUT

Format **INPUT # <file number>, <variable list>**

Purpose This statement is used to read items from a sequential disk file in a similar way to that in which the INPUT statement reads data from the keyboard.

Remarks The sequential file from which data is to be read with this statement must have been previously opened for input by executing an OPEN statement. <file number> is the number under which the file was opened.

As with INPUT, <variable list> specifies the names of variables into which items of data are to be read when the INPUT # statement is executed. Variables specified must be of the same type as data items which are read. Otherwise, a "Type mismatch" error will occur.

Upon execution of this statement, data items are read in from the file in sequence until one item has been assigned to each variable in <variable list>. When the file is read with this statement, the first character encountered which is not a space is assumed to be the start of a data item. With string items, the end of one item is assumed when the following character is a comma or a carriage return, however, individual string items may include commas and carriage returns if they are enclosed in quotation marks when they were saved to the file. The end of a data item is also assumed if 255 characters are read without encountering a comma or carriage return.

With numeric items, the end of each item is assumed when a space, comma, or carriage return is encountered. Therefore, care must be taken to ensure that proper delimiters are used when the file is written to the disk file with the PRINT # statement.

Examples of use of the INPUT # statement are shown in the program below.

See also **INPUT, LINE INPUT, LINE INPUT #, OPEN, PRINT #, WRITE, WRITE #**
Chapter 5

Example

```
10 OPEN "O",#1,"a:test1.dat"
20 FOR I=1 TO 16:'    Saves numeric data items "1"
30 '                  to "16" to file "a:test1.dat"
40 PRINT#1,I
50 NEXT I
60 PRINT#1,"a";CHR$(13);"b"
70 '   Saves "a" and "b" to file "a:test1.dat"
80 '   as separate data items.  Items are separated
90 '   by a carriage return code (CHR$(13)).
100 PRINT#1,CHR$(34)+"c,d,e"+CHR$(34)
110 '   Saves "c,d,e" to file "a:test1.dat" as one
120 '   data item.  This is regarded as one item because
130 '   it is enclosed in quotation marks (CHR$(34))
140 '   to indicate that the commas are part of the
150 '   string, and not delimiting characters.
160 PRINT#1,"f,g"
170 '   Saves "f,g" to file "a:test1.dat" as separate
180 '   data items.  The reason for this is that commas
190 '   are regarded as delimiters unless quotation marks
200 '   are saved to the disk to indicate that the commas
210 '   are part of a string.  Quotation marks are saved
220 '   to a file by specifying their ASCII codes with
230 '   the CHR$ function as shown above with "c,d,e".
240 CLOSE
250 DIM A(15)
260 OPEN "I",#1,"a:test1.dat"
270 FOR I=0 TO 15
280 INPUT#1,A(I)
290 NEXT I
300 FOR I=0 TO 15
310 PRINT A(I);
320 NEXT
330 PRINT
340 INPUT#1,A$,B$,C$,D$,E$
350 PRINT A$:PRINT B$:PRINT C$:PRINT D$:PRINT E$
360 CLOSE

run
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
a
b
c,d,e
f
g
Ok
```

INPUT$

Format **INPUT$(X[,[#]<file number>])**

Purpose Reads a string of X characters from the keyboard buffer or the file opened under <file number>.

Remarks INPUT$(X) reads the number of characters specified by X from the keyboard buffer and returns a string consisting of those characters to the program. If the keyboard buffer does not contain the specified number of characters, INPUT$ reads those characters which are present and waits for other keys to be pressed. Characters read are not displayed on the screen.

Unlike the INPUT and LINE INPUT statements, INPUT$ can be used to pass control characters such as RETURN (character code 13) to the program.

INPUT$(X,[#]<file number>) reads the number of characters specified by X from a sequential file opened under the specified file number. As with the first format, characters which would be recognized as delimiters between items by the INPUT# or LINE INPUT# statements are returned as part of the character string.

Execution of the INPUT$ function can be terminated by pressing the STOP key.

The BASIC statement

```
A$=INPUT$(1)
```

is useful for waiting for a single key to be pressed, in contrast to

```
100 A$=INKEY$ : IF INKEY$=" "THEN 100
```

whereas INKEY$ can scan the keyboard buffer simply to test if a key has been pressed without waiting for it to be pressed.

Example

```
10 OPEN"O",#1,"test"
20 PRINT#1,"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 CLOSE
40 OPEN"I",#1,"test"
50 '
60 '
100 A$=INPUT$(10,#1)  :'Inputs 10 characters into
110                   :'A$ from sequential file opened
120                   :'under file number 1.
125 PRINT A$
130                   :'
140 B$=INPUT$(10)     :'Inputs 10 characters into B$
150                   :'from keyboard.
160 PRINT B$

run
ABCDEFGHIJ
qwertyuiop
Ok
```

INSTR

Format **INSTR([J,]X$,Y$)**

Purpose Searches for the first occurrence of string Y$ in string X$ and returns the position at which a match is found.

Remarks If J is specified, the search for string Y$ begins at position J in string X$. J must be specified as an integer expression in the range from 1 to 255; otherwise, an "Illegal function call" error will occur. If a null string is specified for Y$, INSTR returns the value which is equal to that specified for J. A value of "0" is returned if J is greater than the length of X$, if X$ is a null string, or if Y$ cannot be found. Both X$ and Y$ may be specified as string variables, string expressions or string literals.

Example

```
10 'Example of using INSTR with ON...GOTO to control
20 'flow of program execution.
30 '
40 INPUT"Enter a,b,or c";X$
50 ON INSTR(1,"abc",X$) GOTO 70,90,110
60 PRINT"Illegal entry, try again.":GOTO 40
70 PRINT"Character entered is ";CHR$(34);"a.";CHR$(34)
80 END
90 PRINT"Character entered is ";CHR$(34);"b.";CHR$(34)
100 END
110 PRINT"Character entered is ";CHR$(34);"c.";CHR$(34)
120 END

run
Enter a,b,or c? a
Character entered is "a."
Ok
```

INT

Format INT(X)

Purpose Returns the largest integer which is less than or equal to X.

Remarks Any numeric expression may be specified for X.

See also CINT, FIX

The explanation of CINT also contains information on the differences between CINT, FIX and INT and describes why some problems arise from conversion and storage of numbers in connection with these functions.

Example

```
10 PRINT "I","INT(I)"
20 FOR I=-5 TO 5 STEP .8
30 PRINT I,INT(I)
40 NEXT I
I             INT(I)
-5            -5
-4.2          -5
-3.4          -4
-2.6          -3
-1.8          -2
-1            -1
-.2           -1
 .6            0
 1.4           1
 2.2           2
 3             3
 3.8           3
 4.6           4
```

KEYn/KEY LIST/KEY LLIST

Format

KEY <n>,<X$>
KEY LIST
KEY LLIST

Purpose

Used to define or list the functions of the programmable function keys.

Remarks

The statement KEY <n>,<X$> is used to define or list the functions of the programmable function keys. Here, <n> is an integer from 1 to 10 which indicates the number of the function key being defined (with numbers 6 to 10 denoting keys [PF1] to [PF5] with the [SHIFT] key pressed), and <X$> is a character string of up to 15 characters which is to be assigned to that key. For example,

KEY 1, "LIST"

assigns the LIST function to programmable function key [PF1].

A control code can be included in the character string assigned to a programmable function key by adding +CHR$(J) to X$, where J is the ASCII code for that control character.
For example,

KEY 1,"LIST"+CHR$(13)

assigns the character string "LIST" plus a return code to programmable function key [PF1]; subsequently, pressing [PF1] will list the entire contents of any program contained in the currently logged in program area.

The KEYn statement is useful for changing the definitions of programmable function keys in BASIC application programs.

KEY LIST and KEY LLIST output a list of the current programmable function key definitions to the display and printer. For example KEY LIST will show the following if the keys have not been changed after entering BASIC.

```
PF1         auto
PF2         list
PF3         edit
PF4         stat
PF5         run^M
PF6         load"
PF7         save"
PF8         system
PF9         menu^M
PF10        login
```

Where a control character has been added such as the carriage return with PF5 and PF9 it will be shown as the circumflex character "^" followed by the letter associated with the control character. Thus in these examples the carriage return (ASCII code 13) is shown as ^M.

KILL

Format **KILL <file descriptor>**

Purpose Used to delete files from a disk device.

Remarks The KILL command can be used to delete any type of disk file. The full file descriptor must be specified if the file to be deleted is in a drive other than that which is currently selected. Otherwise, only the file name and extension need to be specified.

Example **KILL "A:GRAPH.BAS"**

This will delete the file in drive A: "GRAPH" which has the extension ".BAS".

NOTE:
Operation of the KILL command is not assured if it is issued against a file which is currently OPEN.

LEFT$

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

Purpose Returns a string composed of the J characters making up the left 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 string X$, the entire string will be returned. If J is zero, a null string of zero length will be returned.

See also **MID$, RIGHT$**

Example 1 A$ = LEFT$("CARROT",3) will return "CAR" to be stored in A$.

Example 2

```
10 A$ = "EPSON"
20 FOR J = 1 TO 6
30 PRINT LEFT$(A$,J)
40 NEXT
```

```
E
EP
EPS
EPSO
EPSON
EPSON
```

LEN

Format **LEN(X$)**

Purpose Returns the number of characters in string X$.

Remarks The number returned by this function also indicates any blanks or non-printable characters included in the string (such as the return and cursor control codes).

Example

```
10 CLS
20 INPUT "Type in a word or phrase";A$
30 PRINT "The length of :- "
40 PRINT A$
50 PRINT "is"; LEN(A$); "characters"
60 GOTO 20

Type in a word or phrase? FRED
The length of :-
FRED
is 4 characters
Type in a word or phrase?



Type in a word or phrase? CHARLIE IS SUPER
The length of :-
CHARLIE IS SUPER
is 16 characters
Type in a word or phrase?
```

LET

Format **[LET] <variable> = <expression>**

Purpose Assigns the value of <expression> to <variable>.

Remarks Note that the word LET is optional. Thus, in the example below, the variables A$ and B$ give the same result when printed, as do A and B.

Example

```
10 CLS
20 LET A$ = "THIS IS A STRING"
30 B$ = "THIS IS A STRING"
40 PRINT A$
50 PRINT B$
60 LET A = 3*4
70 B = 3 * 4
80 PRINT A,B

THIS IS A STRING
THIS IS A STRING
 12             12
Ok
```

LINE

Format **LINE[[STEP] (X1,Y1)] – [STEP](X2,Y2)[,[<function code>] [,[B[F]][,<line style>]]]**

Purpose Draws a line between two specified points.

Remarks This statement is a graphics command which can only be used in screen mode 3. It draws a straight line between two specified points on the graphic screen. The coordinates of the first point are specified as (X1, Y1) and those of the second point are specified as (X2, Y2).

If STEP is omitted, (X1, Y1) and (X2, Y2) are absolute screen coordinates; if STEP is specified, (X1, Y1) indicate coordinates in relation to the last dot specified by the last graphic display statement executed (PSET, PRESET or LINE). The coordinates of the last previously specified dot are maintained by a pointer referred to as the last reference pointer (LRP); this pointer is updated automatically whenever a PSET, PRESET, or LINE statement is executed.

For example

LINE (0,0) – (479,63)
draws a line diagonally from the top left hand corner to the bottom right hand corner.

LINE – (100,50)
draws a line from the last plotted point (i.e. the LRP) to the point (100,50).

LINE (10,10) – STEP (100,50)
draws a line from point (10,10) to a point 100 points to the right and 50 down from point (10,10); i.e., to point (110,160).

LINE – STEP (100,50)
draws a line 100 points to the right and 50 points down from the coordinates of the LRP.

LINE STEP (10,10) – (100,50)
draws a line from a point 10 to the right and 10 down from the LRP to the absolute point (100,50).

LINE STEP (10,10) – STEP (100,50)
draws a line from a point 10 to the right and 10 down from the LRP. The LRP is then updated and the line drawn 100 points to the right and 50 down from the first end point of the line. Thus if the last point plotted before this command was executed was (5,3), the line would be drawn from the point (15,13) to (115,63).

<function code> is a number from 0 to 7 which specifies the line function. If 0 is specified, the LINE statement resets (turns off) dots along the line between the specified coordinates. If a number from 1 to 7 is specified, dots along the line are set (turned on) when the statement is executed. If no <function code> is specified, 7 is assumed.

Specifying "B" causes the LINE statement to draw a rectangle whose diagonal dimension is defined by the two points specified. If the F option is specified together with the B option, the rectangle is filled in. However, the BF option cannot be specified together with <line style>, although simply using B will allow rectangles to be drawn using different line types.

If you want to use the B or BF function without using the <function code>, a comma must be used as separator.

For example

LINE (0,0) — (20,15) ,,BF
will fill a box 20 points wide and 15 points high in the top left hand corner of the screen.

The <line style> option is a parameter which determines the type of line drawn between the two specified points. The line style is specified as any number which can be represented with 16 binary digits; i.e., the line style can be specified as any number from 0 to 65535 (in hexadecimal notation, from &HO to &HFFFF). There is a one-to-one correspondence between the settings of the binary

digits of <line style> and the settings of each 16-dot segment of the line drawn when the statement is executed. When the <function code> is 1 to 7 or defaults to 7 because no value is inserted, all points corresponding to "1" bits are set (i.e., plotted). When the <function code> is specified as 0, all points corresponding to "1" bits are reset (i.e., erased). This is illustrated in the second example program below. In both cases where dots correspond to "0" bits no action is taken. When the length of the line is greater than 16 dots, the pattern is repeated for each 16-dot segment.

For example, dot settings are as follows when <line style> is specified as 1, 43690, and 61680.

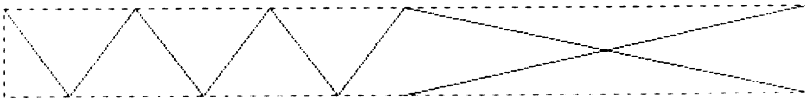
<line style>	Binary equivalent	
1 (&H1)	0000000000000001	
	---------------*	Dot settings (* for on, – for off)
43690 (&HAAAA)	1010101010101010	
	--*-*-*-*-*-*-	Dot settings
61680 (&HF0F0)	1111000011110000	
	****----****----	Dot settings

The LINE statement can only be executed during display in the graphic mode (screen mode 3).

See also **PRESET, PSET**

Example

```
10 SCREEN 3,0,0                        :'Set Graphics screen
15 CLS
20 LINE (0,0)- (479,63),,B,&HF0F0      :'Draw a dotted border to the screen
25 LINE - (240,0)                      :'Draw a line from the LRP to the top
26 '                                     right hand corner
40 LINE STEP(0,63)- (479,0 )            :'Move vertically and draw to top
41 '                                      right hand corner
45 PSET (0,0)                           :'Move LRP to top left corner
50 FOR J = 1 TO 3                       :'Draw lines in a zig zag by
60 LINE -STEP (40,63)                   :'moving across and up or down
70 LINE -STEP (40,-63)
80 NEXT
90 IF INKEY$ = "" THEN 90               :'Wait until a key is pressed
100 '                                     otherwise end of program will
110 '                                     destroy part of graphics
```



The following program illustrates the use of <function codes> with <line style>. A line is drawn in full and then a dashed line is drawn on top of it, using the <line style> to make it dashed. This is repeated, but with the <function code> changed to 0. The dashed line is also drawn below each of the complete lines, for comparison. In the first case with the <function code> set to 7, the dashed line is visible, below the complete line, because where the bits are set to "1" they are plotted. Also they reinforce the complete line when plotted on top of this line. The bits set to "0" cause no change and so the line remains unchanged. When the <function code> is set to "0", and the process repeated, the complete line is converted to a dashed line. Where bits are "1", the points of the complete line are erased. Where they are "0" no action is taken and the points previously plotted remain. The attempt to plot the line below the complete line (i.e. program line 70) gives no visible result because where the bits are "1" erasure of a blank line results in no plotted points. Where the bits are "0", no action is taken, so line 70 appears to do nothing.

```
10 SCREEN 3,0,0:CLS                    :'Set up graphics screen and clear it
20 LINE (0,40) - (300,40)              :'Draw a complete line
30 LINE (0,42) - (300,42),7,,&HFF00 :'Draw a dashed line below the first line
40 LINE (0,40) - (300,40),7,,&HFF00 :'Draw the dashed line on top of the
50 '                                     first line with <function code> = 1
60 LINE (0,50) - (300,50)              :'Draw another line lower down
70 LINE (0,52) - (300,52),0,,&HFF00 :'Draw a dashed line below this line
80 '                                     with <function code> = 0
90 LINE (0,50) - (300,50),0,,&HFF00 :'Draw this dashed line on top of the
100 '                                    second line

Ok
```

LINE INPUT

Format **LINE INPUT[;][<prompt string>;]<string variable>**

Purpose Used to substitute string data including all punctuation into string variables from the keyboard during program execution.

Remarks The LINE INPUT statement is similar to the INPUT statement in that it is used to substitute values into variables from the keyboard during program execution.
However, whereas the INPUT statement can be used to input both numeric and string values, the LINE INPUT statement can only be used for input of string values. Further, only one such string can be input each time the LINE INPUT statement is executed. It is not possible to specify a list of variables separated by commas as is the case with the INPUT statement, because commas are accepted as part of the string.

INPUT allows commas to be entered if the first character typed is the quotation marks character. Quotation marks can be entered as long as they are not input as the first character. LINE INPUT on the other hand allows all characters to be substituted into the specified variable exactly as entered. Further, no question mark is displayed when a LINE INPUT statement is executed unless one has been included in <prompt string> by the user.

As with the INPUT statement, a semicolon immediately following LINE INPUT suppresses the carriage return typed by the user. The cursor is positioned after the last character entered by the user before pressing RETURN .

See also **INPUT**

Example

```
5 CLS
10 LINE INPUT "TYPE IN SOME CHARACTERS ";A$
15 PRINT " This is on the next line"
20 PRINT "The characters were ";A$
30 LINE INPUT;"AND SOME MORE ";B$
40 PRINT " This is on the same line"
50 PRINT "The characters were ";B$

TYPE IN SOME CHARACTERS CHARLIE IS MY DARLING
 This is on the next line
The characters were CHARLIE IS MY DARLING
AND SOME MORE OVER THE SEA TO SKYE This is on the same line
The characters were OVER THE SEA TO SKYE
Ok
```

LINE INPUT

Format **LINE INPUT # <file number>,<string variable>**

Purpose Used to read data into string variables from a sequential access file, in the same way that LINE INPUT is used to read strings from the keyboard.

Remarks The LINE INPUT # statement is similar to the INPUT # statement in that it is used to read data into variables from a sequential access file. The value of <file number> is the number under which the file was opened, and <string variable> is the name of the variable into which data is read when the statement is executed.

Whereas the INPUT # statement can be used to read both numeric and string values, the LINE INPUT # statement can only be used to read character strings. Further, only one such string can be read each time the LINE INPUT # statement is executed. It is not possible to specify a list of variables, as is possible with the INPUT # statement.

Another difference between the INPUT # and LINE INPUT # statements is that whereas the former recognizes both commas and carriage returns as delimiters between data items, the LINE INPUT # statement regards all characters up to a carriage return (up to a maximum of 255 characters) as one data item. Any commas encountered are regarded as part of the string being read. The carriage return code itself is skipped, so the next LINE INPUT # statement begins reading data at the character following the carriage return.

This statement can be used to read all values written by a PRINT # statement into one variable. It also allows lines of a BASIC program which has been saved in ASCII format to be input as data by another program.

See also **INPUT #**

Example See Chapter 6

LIST

Format **LIST [*][<line number>][– <line number>]**

LIST [*][<file descriptor>,][<line number>][– <line number>]

Purpose Lists all or part of a BASIC program on the display screen.

Remarks Executing the LIST command without specifying line numbers or a file descriptor causes the lines of the program in the currently logged in program area to be displayed on the LCD screen.

For other formats, program lines listed and the device to which the list is output are as follows.

LIST <line number> – <line number>
Lists the program in the currently logged in program area on the LCD screen, starting with the first <line number> and ending with the second.

LIST <line number> –
Lists the program in the currently logged in program area on the LCD screen, starting with the specified line and ending with the last line of the program.

LIST – <line number>
Lists all lines of the program from the first line to that specified in <line number>.

LIST <line number>
Lists the program line specified in <line number>.

LIST <file descriptor>
Outputs the program in memory to the device specified in <file descriptor> in ASCII format. (This is the same as using the SAVE statement with the A option to output a program to the specified device in ASCII format.) Devices which can be specified include the LCD screen, the RS-232C interface, RAM disk, external floppy disk drives, and a printer. If line numbers are specified, only the specified lines are output.

When a program is being listed, the listing can be terminated before execution of the command is completed by pressing the STOP key or CTRL and C. BASIC always returns to the command level after execution of a LIST command.

Example 1 **LIST**
Displays a listing of the program in the currently logged in program area.

Example 2 **LIST***
Same as above, but displays program lines without line numbers.

Example 3 **LIST 500**
Displays program line 500.

Example 4 **LIST 150 –**
Displays all program lines from line 150 to the end of the program.

Example 5 **LIST – 1000**
Displays all lines from the beginning of the program to line 1000 (inclusive).

Example 6 **LIST 150-1000**
Displays program lines from 150 to 1000 (inclusive).

Example 7 **LIST "A:CARROT"**
Saves the program in the currently logged in area to drive A: in ASCII format.

LLIST

Format **LLIST[*][<line number>][– <line number>]**

Purpose Lists all or part of the lines of the program in the currently logged in program area to a printer.

Remarks The LLIST command is used in the same manner as LIST, but output is always directed to the printer connected to the PX-8. BASIC always returns to the command level after execution of a LLIST command.

See also LIST

Example 1 **LLIST**
Prints all lines of the program in the currently logged in program area.

Example 2 **LLIST***
Same as above, but prints program lines without line numbers.

Example 3 **LLIST 500**
Prints program line 500.

Example 4 **LLIST 150-**
Prints all program lines from line 150 to the end of the program.

Example 5 **LLIST -1000**
Prints all lines from the beginning of the program to line 1000 (inclusive).

Example 6 **LLIST 150-1000**
Prints program lines from 150 to 1000 (inclusive).

LOAD

Format **LOAD <file descriptor> [,R]**

Purpose Loads a program into memory from a disk drive, RAM disk, the RS-232C interface, or the microcassette drive.

Remarks Specify the device name, file name, and extension under which the program was saved in <file descriptor>. If the device name is omitted, the currently selected drive is assumed; if the file name extension is omitted, ".BAS" is assumed.

When a LOAD command is executed without specifying the "R" option, all files which are open are closed, all variables are cleared, and all lines of any program in the currently logged in program area are cleared; after loading is completed, BASIC returns to the command level.

However, if the "R" is specified, any files which are currently open remain open and program execution begins as soon as loading has been completed. Thus, LOAD with the "R" option may be used to chain execution of programs which use the same data files. The following restrictions must be noted when using LOAD with the "R" option to chain execution of programs.

- All variables are cleared by execution of the LOAD command, regardless of whether the "R" option is specified. Further, the COMMON statement cannot be used to pass variables to the program called. Therefore, some other provision must be made for passing data to the program called (for example, intermediate data could be saved in a file in RAM disk).
- All assignments of variables to positions in random file buffers are cancelled even though the random access files to which the buffers belong remain open. Therefore, the FIELD statement must be executed in the called program to remake these assignments.

See also **CHAIN, MERGE, RUN, SAVE**

Example 1 **LOAD"A:PROG1.BAS"**

Example 2 (Example of program call using LOAD)

```
10 CLS
20 PRINT "This is the calling program, sometimes called the
loader"
30 PRINT "It will now load the program LOAD2.BAS...."
40 LOAD "A:LOAD2.BAS",R

This is the calling program, sometimes called the loader
It will now load the program LOAD2.BAS....
```

Example 3 (Example of program called by Example 2)

```
10 PRINT
20 PRINT"This is the program called LOAD2.BAS which has been
 loaded by the loading program LOAD1.BAS"
30 END

This is the program called LOAD2.BAS which has been loaded b
y the loading program LOAD1.BAS
Ok
```