Chapter 3

COMMANDS AND STATEMENTS

This Chapter describes the commands, statements, and functions used with PX-4 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>,<message>[,W]]**

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

Remarks Executing the ALARM statement without the W option sets the alarm time, and executing it with the W option sets the wake time. 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. It is not possible to set both an alarm and wake time simultaneously.

The alarm or wake date is specified in <date> in the same format as with DATE$ (a system variable), and the alarm/wake time is specified in <time> in the same format as with TIME$.

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, and that the lower second digit is always handled as if it were specified as "0".

<message> must be specified as a string expression whose result is no more than 32 characters; this message is displayed on the System Display when the alarm time is reached, or is assumed as the auto start string when the wake time is reached.

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

MO error (Missing operand) — A required operand was not specified in the statement.

FC error (Illegal function call) — The <date> or <time> parameters were incorrectly specified.

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

ALARM$

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

Purpose Used to check the information set by the ALARM statement.

Remarks <function> is specified as a numeric expression whose result is a value from 0 to 2. 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 that, once "P" has been returned by executing ALARM$(0), "B" is returned when ALARM$(0) is subsequently executed.

FC error (Illegal function call) — The value specified for <function> was outside the prescribed range.

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 FC error (Illegal function call) will occur if 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

FC Error 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 WIDTH 40,8: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 The AUTO START statement is used to specify parameters which are passed to BASIC when a hot start is made.

Remarks The <auto start string> is specified as a string expression whose result is no more than 32 characters. When the power switch is turned on and BASIC is started by a hot start, this string is passed to BASIC in the same manner as if it were input from the keyboard. If a null string is specified for <auto start string>, the auto start function is cancelled.

BEEP

Format **BEEP [<duration>]**

Purpose Causes the speaker to beep.

Remarks The BEEP statement causes the speaker built into the keyboard to make a beeping sound. The numeric value specified in <duration> determines the length of the sound; numbers specified must be in the range from 0 to 255. The length of the sound generated is equal to <duration> × 10 msec, with the result rounded off to the nearest millisecond.

If <duration> is omitted, "10" is assumed.

BLOAD

Format **BLOAD <file descriptor>,[,[<load address>][,R]]**

Purpose Loads machine language programs or data.

Remarks This statement is used to load the machine language program or data file specified in <file descriptor> into memory. Ordinarily, the load address is the same as that specified when the file was saved with the BSAVE statement; however, the file can be loaded into a different area by specifying the starting address of that area in <load address>. Therefore, a machine language program which is loaded into an area other than that specified when it was saved must be fully relocatable (it must be capable of being executed properly in a different area in memory).

When the R option is specified, program execution begins immediately after loading has been completed. Any files which are open at the time remain open. If <load address> is not specified, execution begins at the starting address which was specified when the program was saved; otherwise, execution begins at the address specified in <load address>.

This statement will only load programs or data into the area from the address following that specified in the CLEAR statement to that immediately preceding the beginning of BDOS (or to the beginning of the user BIOS area, if any).

This statement can also be used to load user-defined character font files.

FC error (Illegal function call) — An attempt was made to load a file into an inhibited area.

BSAVE

Format **BSAVE <file descriptor>,<starting address>,<length>**

Purpose Saves machine language programs or data to files.

Remarks This statement saves the contents of the machine language programs or data files from memory to the file specified in <file descriptor>. The number of bytes specified in <length> is saved, starting at the address specified in <starting address>.

If the machine language program is written so that execution can begin at the address specified in <starting address>, the program can be executed immediately upon loading with the BLOAD statement.

This statement can also be used to save user-defined character fonts.

If CAS0: is specified as the device in <file descriptor>, the save can be verified by executing the LOAD? statement.

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 statements 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 J 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 $12 \times 2.6 - 0.2 = 31$. 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 size>]]]**

Purpose Clears numeric variables to 0, sets null strings to all string variables, specifies the upper limit of the BASIC memory area, and sets the size of the stack area.

Remarks This statement clears all numeric variables to 0 and sets all null strings in all string variables; at the same time, it cancels all variable type definitions made with the DEFINT, DEFSNG, DEFDBL, and DEFSTR statements and closes all files which are currently open.

<upper memory limit> specifies the highest address in memory which can be used by BASIC. The maximum value which can be specified is &H6000. This is the value which is effective when BASIC is started (unless a different address has been specified with the /M: option). The area from <upper memory limit> to the beginning of BDOS can be used for storage of machine language programs, user-defined character sets, or other data. The upper memory limit is not changed by execution of the CLEAR statement if this parameter is omitted.

The <stack size> parameter specifies the size of the stack area which is used by BASIC. The initial size is 256 bytes. If this parameter is omitted, the stack area size is not affected by execution of the CLEAR statement.

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. An FE error (File already open) 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 BF error (Bad file mode) 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,** Chapter 4

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 screen.

Remarks The CLS statement clears the virtual screen. This is the same as executing 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
```


COM(n) ON/OFF/STOP

Format **COM(n)** **ON** | **OFF** | **STOP**

Purpose Enables, disables, or defers interrupts from the communication line.

Remarks This statement enables, disables, or defers external interrupts from the communication line. The n in COM(n) indicates the communication port, and is specified as a number from 0 to 3. COM(n) ON enables interrupts. After this statement has been executed, the specified communication port is checked each time a statement is executed and, if any signal has been received, an interrupt is generated and processing branches to the communication trap routine designated by the ON COM(n) GOSUB statement.

COM(n) OFF disables communication interrupts. After this statement has been executed, processing does not branch to the communication trap even if an external signal is received.

COM(n) STOP defers generation of communication interrupts. If an external signal is received following execution of this statement, the event is noted but processing does not branch to the communication trap routine. However, processing does branch to the communication trap routine the next time interrupts are enabled by executing the COM(n) ON statement.

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.

COPY

Format COPY

Purpose Copies the contents of the LCD screen to the printer.

Remarks The COPY statement outputs the contents of the LCD screen to the printer. This is the same as pressing CTRL + PF5 .

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)
 .0707371
```

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**

Purpose The CSRLIN function returns the current vertical coordinate of the cursor.

Remarks The value returned by the CSRLIN function indicates the current location of the cursor in the virtual screen. The meaning and range of values returned is the same as for the LOCATE statement.

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 4

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

DATE$

Purpose

Reads the date of the PX-4's built-in clock.

Remarks

The DATE$ function returns the date of the built-in clock as a character string in the following format.

"MM/DD/YY"

Here, MM indicates the month as a value from "01" to "12", DD indicates the day of the month as a value from "01" to "31", and YY indicates the last two digits of the year as a value from "00" to "99".

DATE$ is a system variable, and can be set by executing DATE$ = "MM/DD/YY".

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-4's built-in calendar clock.

Remarks

As a variable, DAY returns the day of the week of the PX-4'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 argument type does not match, a TM error (Type mismatch) occurs.

The DEF FN statement must be executed before the corresponding user function can be called: otherwise, a UF error (Undefined user function) 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 appendixes. 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 L.

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 TM error (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 G 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 FC error (Illegal function call) 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 FC error (Illegal function call) 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 101-190 is correct but DELETE 101-200 will generate an error. However, line 101 does not have to exist.

Examples **DELETE 10** will remove line 10.
DELETE 10-90 will remove lines 10 to 90.
DELETE -90 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 BS error (Subscript out of range); 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 (<disk device name>)**

Purpose Returns the amount of free space on a disk.

Remarks The DSKF function returns the amount of free space on the disk specified in (<disk device name>) in kilobyte (1024-byte) units. The device specified in <disk device name> must be that of a disk device which is supported by PX-4 BASIC.

FC error (Illegal function call) — The <disk device name> argument was incorrectly specified.

DU error (Device unavailable) — The specified device was not available.

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

EDIT

Format **EDIT [<line no.>]**

Purpose Places BASIC in the edit mode.

Remarks This command makes it possible to edit program lines on-screen by displaying the program line specified in <line no.> and positioning the cursor at the beginning of that line. Following occurrence of an error, "EDIT." can be executed to display the line in which the error occurred. The error line is displayed only if the command is executed immediately following occurrence of the error; in this case, the cursor is positioned at the point at which the error occurred.

UL error (Undefined line number) — The program line specified is not included in the program.

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.