LM80C

LM80C BASIC

Language reference

LM80C BASIC

BASIC interpreter derived by NASCOM BASIC
(based on Microsoft BASIC 4.7)
with additional statements to take advantage
of the features of the LM80C Color Computer

This release covers the
LM80C BASIC 3.2 (09/03/2020)
integrated into LM80C firmware R3.2

Copyright notices:

Index

Copyright notices:....3
1. OVERVIEW....7
1.1 Preface....7
1.2 Video VS Console....7
1.3 Memory....7
1.4 System overview....7
1.5 Boot....8
2. LM80C BASIC....10
2.1 Differences with NASCOM BASIC....10
2.2 New numeral systems....10
2.3 New functions....10
2.4 New commands....11
2.5 Line input....11
2.6 Numbers....12
2.7 Variables....13
2.8 Strings....13
2.9 Array variables....14
2.10 Operators and precedence....15
3. LM80C BASIC INSTRUCTIONS....17
3.1 In alphabetical order....17
3.2 Per category....17
3.2.1 Arithmetical operators....17
3.2.2 Call/Return, Jump and Loop....18
3.2.3 Timing....18
3.2.4 Conditions....18
3.2.5 Conversion Functions....18
3.2.6 Code flow and programming....18
3.2.7 Graphics statements....18
3.2.8 Display....19
3.2.8 System & Input/Output....19
3.2.9 Keyboard....19
3.2.10 Logical operators....19
3.2.11 Mathematical functions....19
3.2.12 Machine Language Functions....20
3.2.13 Memory management....20
3.2.14 Sound....20
3.2.15 String Handling....20
3.2.16 Variable settings, data management, & user defined functions....20
4. LANGUAGE REFERENCE....21
&B....21
&H....21
ABS....21
AND....22
ASC....22
ATN....22
BIN$....23
CHR$....23

CIRCLE....23
CLEAR....23
CLS....24
COLOR....24
CONT....25
COS....25
DATA....25
DEEK....26
DEF FN....26
DIM....26
DOKE....27
DRAW....27
END....27
EXP....28
FOR...TO...STEP....28
FRE....29
GOSUB....29
GOTO....30
HEX$....30
IF...THEN IF...GOTO....30
INKEY....31
INP....31
INPUT....32
INT....32
LEFT$....33
LEN....33
LET....33
LINES....33
LIST....34
LOCATE....34
LOG....34
MID$....35
NEW....35
NEXT....35
NOT....35
ON....36
OR....37
OUT....37
PAUSE....37
PEEK....38
PLOT....38
POKE....39
POS....39
PRINT....39
READ....40
REM....40
RESET....41
RESTORE....41
RETURN....41
RIGHT$....42

RND....42
RUN....43
SCREEN....43
SERIAL....44
SGN....45
SIN....45
SOUND....46
SPC....47
SQR....47
SREG....47
SSTAT....48
STOP....48
STR$....48
SYS....49
TAB....49
TAN....49
TMR....50
USR....50
VAL....51
VOLUME....51
VPEEK....52
VPOKE....52
VREG....53
VSTAT....53
WAIT....53
WIDTH....54
XOR....54
5. USEFUL LINKS....55

1. OVERVIEW

1.1 Preface

This manual is not intended as a BASIC guide to learn the language. We assume that the reader has already a BASIC knowledge of his/her own so that he/she is able to read and understand what's discussed here. This book is just a reference manual to the LM80C BASIC, useful to learn how to use the peculiar statements of the LM80C.

1.2 Video VS Console

The most significant difference between the original Microsoft BASIC released for NASCOM computers and the LM80C BASIC is the usage of the screen instead of the terminal or console devices as main ouput. Everything you digit on the keyboard or is ouput by the computer is printed on the screen: you can open a serial line to communicate with a host computer but the primary device will always be the video screen. When writing a program of your own or running one such of those found anywhere on internet, keep in mind that you have to think that the LM80C Color Computer (from now on, simply LM80C for convenience) will always and primarily print output on the screen, so please adapt/modify the source you are inserting to take care of such behaviour.

1.3 Memory

NASCOM Basic was originally released for machines that had a limited amount of RAM, i.e. 4/8 KB. Surely, they could be expanded and several users reached 16/24/32 KB of memory and more but this wasn't the standard. The LM80C actually comes with 32 KB od SRAM, so don't hesitate to use all of this room. Just keep in mind that the original limit on the lenght of a input line is still fixed at 72 chars. But, apart that, you can enter bigger programs.

1.4 System overview

The LM80C is a powerful computer made around the Z80 CPU from Zilog. It runs at 3.68 MHz and is able to generate a video image of 256x192 pixels at 15 colors with 3 audio channels and serial&parallel I/O. Here are the main technical characteristics:

- CPU: Zilog Z80B at 3.68 MHz
- 32 KB of SRAM
- 32 KB of EEPROM
  - firmware to drive video/audio/serial/paraller peripheral chips
  - integrated LM80C BASIC interpreter
- Video: TMS9918A at 10.7 MHz
  - dedicated 16/32 KB of VRAM
  - screen resolution: 256x192 pixel
  - 15 colors + transparent
  - 32 monochrome sprites from 8x8 to 32x32 pixels
  - NTSC output signal ad 60 Hz (can be seen on any moderm TV set, even PAL)
  - text mode: 40x24 chars (6x8 pixels per char)
  - text/graphic mode: 32x24 chars (8x8 pixels per char) with tile graphics and sprite support

    - graphics mode: 256x192 pixels with bitmap graphic and sprite support
    - multicolor mode: 64x48 pixels
- Audio: YM2149F (or AY-3-8910):
    - 3 analog outputs
    - 8 octaves
    - white noise on any audio channel
    - envelope support
    - 2x8 bit I/O ports
- Peripherals:
    - Z80 SIO:
        - 2x serial ports with a/synchronous support
        - up to 57,600 bps
        - 5/6/7/8/9 pixels per char
        - parity and stop bits supported
    - Z80 PIO:
        - 2x8 bit parallel ports
        - port B is connected to a 8xLEDs matrix for status messages
    - Z80 CTC:
        - 4x channels timer/counter
        - used to generate the 1/100th of a second system tick signal that is used to temporize some interal operations
        - also used to generate the software selectable serial clock for the Z80 SIO

1.5 Boot

At boot the LM80C visualizes a colorful logo with a litlle beep: during this shot time, a check of the system LEDs has made, turning them on one after one. Lastly, the current version of the firmware is shown with the copyright messages of the Z80 BASIC from Microsoft (©1978) and the amount of RAM that is free for BASIC. The text below is printed by LM80C firmware R3.2:

```
LM80C by Leonardo Miliani
Firmware R3.2

Z80 BASIC Ver 4.8
Copyright © 1978 by Microsoft
32250 Bytes free
Ok
```

After a reset the user is asked to choose between a “cold” or a “warm” start. A cold start is just like a power off/power on of the system: every register in memory is re-initialized and the previous contents of the RAM are cleaned, including the current BASIC program. Instead, a warm start preserves the BASIC program in RAM. The choose is made by pressing “C” or “W” keys when prompted for:

```
LM80C by Leonardo Miliani
Firmware R3.2

<C>old or <W>arm start?
```

If the cold start is choosen, the system will print the copyright notices with the available free memory; if the wrm start is choosen, then only a “Ok” will be printed.

2. LM80C BASIC

2.1 Differences with NASCOM BASIC

The LM80C BASIC is a complete BASIC interpreter written to take advantage of the hardware features of the LM80C Color Computer. Due to the differences berween the original NASCOM systems and the LM80C hardware, some statements have been removed from the interpreter because they laid over the original NASCOM machines. They are:

```
CLOAD/CSAVE: loaded/stored data from/to an external mass storage device
PSET: set a video pixel on
POINT: returned the state of a video pixel
MONITOR: it launched the MONITOR program
NULL: set the null chars to be printed at the end of a line
```

Some statements now have a different behaviour:

```
SCREEN: changes the video mode
RESET: resets the system
```

There are a lot of new statements, used to take advantage of the much powerful hardware of the LM80C. But, firstly let's start with a big thanks to Grant Searle, from whose BASIC comes the LM80C version, that added some useful statements (they are marked with a "*GS*").

2.2 New numeral systems

The same number can be represented in several different numeral systems. The usual one is the decimal numeral system, where every single digit can only be the usual numerical digits from 0 to 9. LM80C BASIC supports other systems for numbers: binary and hexadecimal numeral systems. The first one uses only 0 & 1 digits and it’s the “language” of the digital signals used inside the computers, while the hexadecimal is a representation used in assembly language which uses a base of 16, so that every digit can be any number from 0 to 9 plus the letters from A to F to represent the values from 10 to 15. A base-2 number is preceded by the prefix &B while an hexadecimal number is introduced by the prefix &H:

```
&Hxxxx: hexadecimal base. *GS*
&Bnnnn: binary base. *GS*
```

2.3 New functions

A *function* is a statement that gets an input argument and returns another data, that depends on the way it processed the input.

```
BIN$: return the binary representation of a number. *GS*
HEX$: return the hexadecimal representation of a number. *GS*
INKEY: return the ASCII code of the key being pressed
SSTAT: read the registers of the PSG
TMR: return the value of the system timer
VPEEK: read from VRAM
```

```
VSTAT: return the value of status register of the VDP
```

2.4 New commands

Commands are statements that tell the system to perfom a specific operation. Commands can get some arguments but usually don't return any value. Here are the new commands:

```
CIRCLE: draws a circle
COLOR: set the foreground, background, and border colors
CLS: clears the screen
DRAW: draws a line
LOCATE: position the cursor onto the screen
PAUSE: pauses the execution of the code for a certain bit of time
PLOT: draws a pixel point
RESET: resets the system
SCREEN: changes the display mode
SERIAL: open a serial communication line
SOUND: plays a sound tone
SREG: writes into a PSG register
SYS: executes an assembly routine
VOLUME: sets the volume of the PSG audio channels
VPOKE: writes into VRAM
VREG: writes into a VDP register
XOR: make a XOR between operators
```

2.5 Line input

A number before a single or a list of statements introduces a *program line* that, when RETURN is pressed, is stored into memory. If the line doesn't exist, the new line is saved as it, otherwise if the line is already present it will be replaced byt the current one. Any line starting with a number is interpreted as a program line: if nothing follows the number the BASIC interpreter first will look for a stored line with the same number. If found, it will be deleted from memory. If no program lines corresponds, an error of "Undefined line number" will be raised. The maximum allowed value for a line number is 65,529.

Examples:

```
10 PRINT "HELLO"
```

When RETURN is pressed, the line is stored into memory

```
10 PRINT "WORLD"
```

When RETURN is pressed, this line will overwrite the current line into memory

```
10
```

When RETURN is pressed, the line whose number is 10 will be removed from memory.

2.6 Numbers

LM80C BASIC accepts any integer or floating point number as constants. Allowed formats are with or without exponent notation, i.e.:

```
123
0.456
1.25E+06
```

They are all acceptable numeric constants. Any number of numerical digits can be input up the maximum allowed number of chars per single line, however only the first 7 digits of a number are significant and the seventh digit is rounded up. So the following instruction:

```
PRINT 1.234567890123
```

will produce the following output:

```
 1.23457
```

A printed number is preceded by the sign “-” if it is negative:

```
PRINT -1.2
-1.2
```

Otherwise an empty cell will be printed before the number if it’s positive:

```
PRINT 1.2
 1.2
```

If the absolute value of a number is in the range 0 to 999,999, the number will be printed as an integer:

```
PRINT 123567
 123567
```

If the absolute value of a number is greater than or equal to 0.01 and less than or equal to 999,999, it will be printed in fixed point notation with no exponent:

```
PRINT 99000.1
  99000.1
```

If a number doesn’t fall into the previous two categories, it will be printed using the scientific notation, whose format is as follow:

```
sX.XXXXXEsTT
```

where “s” stands for the sign of the mantissa (the part to the left of the decimal point) and the exponent (the part to the right of the decimal point), “X” for the digits of the mantissa and “T” for the digits of the exponent. Non-significant zeroes in the mantissa are suppressed while two digits are always printed in the exponent. The exponent is in range -38 to +38. The largest number that may be represented is 1.70141E+38 while the smallest positive number is 2.9387E38.

The followings are output examples of how the LM80C BASIC prints some numbers:

```
+1                1
-1                -1
1000000           1E06
-12.34567E-10     -1.23456E-09
.01               .01
.000123           1.23E-04
```

In all formats, an empty char is printed after the last digit.

2.7 Variables

A variable is a kind of pointer that represents a certain value into the memory of the computer. A variable can lead any value, assigned explicitely by the programmer or assigned as the result of some calculations. The name of a variable may be any lenght but the alphanumeric characters after the first two are ignored. Some rules must be respected: the first character of the name must be a letter, and no reserved words may be used as variable name or appear within a variable name. Here are some examples of valid names:

```
A
P1
CRONO
```

And here are some names that are not valid:

```
%W          first char must be a letter
ON          reserved word
RGOTO       it contains a reserved word
```

2.8 Strings

Despite a numerical variable, that can only store numbers, a string is an sequence of alphanumeric characters like letters, numbers, puntuation, and other chars. A string must be included between double quotation marks at the beginning and at the end of the chars that form the string itself. A “$” as trailing char in the name indicate a string variable. The following is an example of a string definition:

```
10 A$=”TEST”
```

Strings can be concatenated by using the “+” operator. The concatenation results in the second string being attached at the end of the first one:

```
10 B$=”WORLD”
20 A$=HELLO “ : REM NOTE THE SPACE AT THE END
30 C$=A$+B$ : PRINT C$
```

Running the program above result in this message being printed on the screen:

```
HELLO WORLD
```

Strings can be evaluated with relational operators. In this case the strings are taken into account the ASCII value of their characters until a difference is found. So, for example, `“a” > “A”` is true because ASCII value of “a” is 97 while “A” is 65. Spaces are considered, too: a space corresponds to ASCII value 32. So `“ a” > “a”` is False: note the leading space in the first string.

A string can be up to 255 chars. A string array can be defined with DIM statement: each element of the array is a string that may be up to 255 chars. Hovewer, at any time string characters must not exceed the space allocated by the system to store strings: the LM80C by default reserves 100 bytes. If the program tries to create a string that exceeds the string space an OUT OF STRING SPACE error will be raised. This amount can be changed with CLEAR statement.

2.9 Array variables

An array variable is a series of several variables refered to by the same name. An array is created by using the DIM statement:

```
DIM VV(<subscript>[,<subscript>,…)
```

where VV is a variable valid name (see above) and <subscripts> are the dimensions of the array. An array may have as many dimensions as will fit on a single line and can be accomodated into the memory. A subscript must be an integer value: the smallest subscript is zero. This means that an array always has as many items as the subscript plus 1. See the examples:

```
DIM A(10,10)
```

defines an array of 121 elements, because each dimensions is formed by 11 elements, from 0 to 10.

```
DIM A(5)
```

defines a mono-dimensional array of 6 elements (whose indexes go from 0 to 5).
Each DIM statement can apply to more than 1 array:

```
DIM A(10,5),B(8,2)
```

An array can also be dimensioned dinamically during program execution:

```
DIM A(I+2)
```

At runtime, the expression is evaluated and the results truncated to an integer.

If an array has not been dimensioned before it is used in a program, the LM80C BASIC assumes that is has default subscript of 10 (11 items) for each dimension. A “SUBSCRIPT OUT OF RANGE” error is raised if an attempt is made to access an item outside the limits fixed by a DIM statement:

```
10 DIM A(10,10)
20 B=A(5,5,5)  => error: indexes outside the assigned space
```

A “REDIMENSIONED ARRAY” error is raised if an attempt of a redimensioning of an array is made in the program:

```
10 DIMA(10,10)
20 DIMA(20,20) => error: array already dimensioned
```

2.10 Operators and precedence

LM80C BASIC provides a full range of mathematical and logical operators. The order of execution of the operations in an expression is done in accordance to their precedence, as shown below in decreasing order. Operators on the same line in the able below have the same precedence and are evaluated from left to right in an expression:

- `()` → parentheses
- `^` → exponentiation
  Note: any number to the 0 power is 1; o to a negative power raises a “Division by zero” error.
- `-` → Negation: this is the unary minus operatore
- `*, /` → multiplication & division
- relational operators:
  - `=` → equal
  - `<>, ><` → not equal
  - `<` → less than
  - `>` → greater than
  - `<=, =<` → less than or equal to
  - `>=, =<` → greater than or equal to
- `NOT` logical, bitwise negation
- `AND` logical, bitwise disjunction
- `XOR` logical, bitwise exclusive disjunction
- `OR` logical, bitwise conjunction

Relational operators may be used in any expressions: relational expressions retun a value of -1 (True) or 0 (False).

Logical operators may be used for bit manipulation and boolean algebraic expressions. AND, OR, XOR, and NOT convert their values into 16-bit signed, two’s complement integers in range -32,768 to +32,767: if the arguments are not in this range an “Illegal function call” error is raised. See language references below for truth tables for those operators.

3. LM80C BASIC INSTRUCTIONS

3.1 In alphabetical order

&B	**INP**	**RUN**
&H	**INPUT**	**SCREEN**
ABS	**INTERNALLEFT$**	**SERIAL**
AND	**LEN**	**SGN**
ASC	**LET**	**SIN**
ATN	**LINES**	**SOUND**
BIN$	**LIST**	**SPC**
CHR$	**LOCATE**	**SQR**
CIRCLE	**LOG**	**SREG**
CLEAR	**MID$**	**SSTAT**
CLS	**NEW**	**STEP**
COLOR	**NEXT**	**STOP**
CONT	**NOT**	**STR$**
COS	**ON**	**SYS**
DATA	**OR**	**TAB**
DEEK	**OUT**	**TAN**
DEF	**PAUSE**	**THEN**
DIM	**PEEK**	**TMR**
DOKE	**PLOT**	**TO**
DRAW	**POKE**	**USR**
END	**POS**	**VAL**
FN	**PRINT**	**VOLUME**
FOR	**READ**	**VPEEK**
FRE	**REM**	**VPOKE**
GOSUB	**RESET**	**VREG**
GOTO	**RESTORE**	**VSTAT**
HEX$	**RETURN**	**WAIT**
IF	**RIGHT$**	**WIDTH**
INKEY	**RND**	**XOR**

3.2 Per category

3.2.1 Arithmetical operators

-	Subtraction
+	Addition
/	Division
*	Multiplication
^	Power

3.2.2 Call/Return, Jump and Loop

FOR..NEXT
GOSUB
GOTO
RETURN

3.2.3 Timing

TMR

3.2.4 Conditions

<	Checks if the first argument is less than the second
>	Checks if the first argument is greater than the second
=	Checks if the first argument is equal to the second
<=	Checks if the first argument is less than or equal to the second
>=	Checks if the first argument is greater than or equal to the second
<>	Checks if the first argument is different than the second

IF...GOTO
IF...THEN
ON...GOSUB
ON...GOTO

3.2.5 Conversion Functions

&B
&H
ASC
BIN$
CHR$
HEX$
STR$
VAL

3.2.6 Code flow and programming

CONT
END
LIST
PAUSE
REM
RUN
STOP

3.2.7 Graphics statements

CIRCLE
COLOR
DRAW

PLOT
VPEEK
VPOKE
VREG
VSTAT

3.2.8 Display

CLS
LINES
LOCATE
POS
PRINT
SCREEN
SPC
TAB
WIDTH

3.2.8 System & Input/Output

INP
OUT
RESET
SERIAL
WAIT

3.2.9 Keyboard

INKEY
INPUT

3.2.10 Logical operators

AND
NOT
OR
XOR

3.2.11 Mathematical functions

ABS
ATN
COS
INT
LOG
RND
SGN
SIN
TAN

3.2.12 Machine Language Functions

SYS
USR

3.2.13 Memory management

CLEAR
DEEK
DOKE
FRE
NEW
PEEK
POKE

3.2.14 Sound

SOUND
SREG
SSTAT
VOLUME

3.2.15 String Handling

LEFT$
LEN
MID$
RIGHT$

3.2.16 Variable settings, data management, & user defined functions

CLEAR
DATA
DEF FN
DIM
LET
READ
RESTORE

4. LANGUAGE REFERENCE

&B

```
Syntax: &Bnnnn
```

Binary base. It interprets the argument <*nnnn*> as a binary value (signed int). <*nnnn*> can be made only by "0" and "1" digits.

Example:

```
?&B1000  => prints value 8
```

&H

```
Syntax: &Hxxxx
```

Hexadecimal base. It interprets the argument <*xxxx*> value as an hexadecimal value (signed int). Each char of <*xxxx*> can only be any number between 0 and 9 and any letter between A (10) and F (15).

Example:

```
?&H0F  => prints value 15
```

ABS

```
Syntax: ABS(x)
```

Returns the absolute value (i.e. with no sign) of the expression <*x*>.

Examples:

```
ABS(12.4) => 12.4
ABS(-75) => 75
```

AND

```
Syntax: arg1 AND arg2
```

Logic operator used in boolean expressions. AND performs a logical conjuntion. The interpreter supports 4 boolean operators: AND, OR, XOR, and NOT, each of them work with 16-bit signed integers. They convert their inputs in 16-bit integers and return a value in such format. This means that the result is true only if both expressions <*arg1*> and <*arg2*> are true. The result of an AND operation follows the truth table below (1=T=true, 0=F=false):

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

Examples:

```
10 AND 1 => 0 (because 10=1010b so 1010 AND 0001 = 0000)
11 AND 1 => 1 (because 11=1011b so 1011 AND 0001 = 0001)
```

See also: NOT, OR, and XOR.

ASC

```
Syntax: ASC(X$)
```

Returns the ASCII code of the first character of the string <*X$*>.

Example:

```
ASC("A") => 65
ASC("AB") => 65
```

ATN

```
Syntax: ATN(x)
```

Returns the arctangent of <*x*>. The result is in radius in range -pi/2 to pi/2.

Example:

```
ATN(2) => 1.10715
```

BIN$

```
Syntax: BIN$(x)
```

Converts an expression into a string containing the binary value of argument <*x*>.

Example:

```
BIN$(12) => "1010"
```

CHR$

```
Syntax: CHR$(x)
```

Returns a string containing the character whose ASCII code is represented by expression <*x*>.

Example:

```
CHR$(65) => "A"
```

CIRCLE

```
Syntax: CIRCLE x,y,radius[,color]
```

Draws a circle whose center is at <*x*>,<*y*> coordinates. <*x*> can range from 0 to 255, while <*y*> can range from 0 to 191. <*radius*> can range from 0 to 255. If <*color*> is passed, the circle will be drawn with the specified color, otherwise the foreground color will be used (see COLOR). Points of the circumference that come out of the screen won't be painted.

Example:

```
CIRCLE128,96,25 => draws a circle centered in the middle of the screen with
                   a diameter of 50 pixels (radius=25)
```

CLEAR

```
Syntax: CLEAR [xx]
```

The call of CLEAR with no arguments set the contents of all the numeric variables stored in memory to 0 and of the string variables to "" (empty string). If called with a numeric expression, it sets the string space to the value of expression <*xx*>. At startup, the string space is set by default to 100. See the string chapter for more info.

Examples:

```
CLEAR => clears every variable
CLEAR 200 => sets the string space to 200 chars
```

CLS

```
Syntax: CLS
```

Clears the current screen, intializes the pattern cells. The filling value is graphic mode dependant: in modes 0, 1, & 4 the video buffer is filled up with ASCII value $00 (the "null" character) while in modes 2 & 3 the command simply resets the pixels of the screen. Finally, in text modes, it moves the cursor at coordinates 0,0 (the top left corner). CLS clears the screen by using the background color set with COLOR statement. Same behaviour can be obtained in text modes by pressing together the SHIFT + CLEAR/HOME keys.

COLOR

```
Syntax: COLOR foreground[,background][,border]
```

Sets the colors of the screen. It can have 1 to 3 arguments, depending of the current video mode. <*foreground*> sets the color of the text or of the pixels being printed on screen; <*background*> sets the color of the background parts, while <*border*> sets the colors of the borders of the image (the parts over the top, right, bottom, and left of the display).

Values range from 1 to 15:

1: black	9: light red
2: medium green	10: dark yellow
3: light green	11: light yellow
4: dark blue	12: dark green
5: light blue	13: magenta (purple)
6: dark red	14: gray
7: cyan (aqua blue)	15: white
8: medium red	

In SCREEN 0 (text mode), only the first 2 arguments are allowed since the background and border colors coincide. In SCREEN 3 (multicolor mode) only the background color is allowed. When in graphic mode 2, the color of the pixels painted by PLOT, DRAW, and CIRCLE commands, otherwise not specified, is the foreground color set by COLOR.

Examples:

```
COLOR 1,15,5 => in SCREEN 1 it sets the text in black, the background in
                white, and the border in light blue
COLOR 15,5   => in SCREEN 0 it sets the text to white on light blue
                background (this is the default combination)
COLOR 3      => in SCREEN 3 sets the border color to light green
```

CONT

`Syntax: CONT`

Given after an error that halted the execution of a program, forces the interpreter to continue with the line after the one that raised the error.

COS

`Syntax: COS(X)`

Returns the cosine of <*x*>. The result is in radius in range -pi/2 to pi/2.

Example:

```
COS(10) => -0.839072
```

DATA

`Syntax: DATA <list>`

Introduces a list of informations used by the program itself. <*list*> can be a list of numerical or strings separated by commas and read with the READ statement. A string can be inserted without quotation marks but, if the string must contain any space, their use is needed.

Example:

```
10 DATA 20,30,40  => numericals
20 DATA " HELLO ",LM80C => Note the usage of quotation marks to include the
                           leading and trailing spaces in the first string
```

See also RESTORE.

DEEK

```
Syntax: DEEK(nnnn)
```

Reads a word from memory cells given by the expression <*nnnn*> and <*nnnn*>+1. A word is a 16-bit value stored in 2 contiguos memory cells: <*xxxx*> contains the low byte while <*nnnn*>+1 contains the high byte.

Example:

```
A=DEEK(1000) => reads the word at 1000/1000+1 and stores it into A
```

DEF FN

```
Syntax: DEF FNname(arg)=<expression>
```

Creates an user-defined function that expands the built-in functions of the interpreter. <*name*> is the name of the new function and must follow the FN statement and must be a valid variable name; <*arg*> is the name for the argument passed that will be used by the function too, , and it must be a valid variable name too; <*expression*> is the newly defined function. Limits: everything must reside in 72 chars (the lenght of a standard line); only one argument is allowed. To call the function just use the <*FNname*> name.

Example:

```
10 DEF FNRAD(DEG)=3.14159/180*DEG:REM SETS THE FUNCTION
20 PRINT FNRAD(100):REM FUNCTION CALL (PRINTS 1.74532)
```

DIM

```
Syntax DIM <name>(<dim1>,[dim2])[,....]
```

Allocates space for array variables. More than one array can be dimensioned with one DIM statement. Array can have one ore two dimensions. If an array is used without being dimensioned, it is assumed to have a maimum subscript of 10 (i.e., eleven elements from 0 to 10). So, for example A(I,J) is assumed to have 121 elements, unless otherwise dimensioned.

Examples:

```
DIM A(10) => monodimensional array, elements numbered from 0 to 10
DIM B(5,5) => bidimensional array, indexes fro 0 to 5 (36 elements).
```

DOKE

```
Syntax: DOKE nnnn,val
```

Writes a word value (a 16-bit number) into a couple of contiguos memory cells. <*nnnn*> and <*val*> must be valid numericals, in any supported format. The low byte of <*val*> will be written into location <nnnn> while the high byte of <*val*> will be written into location <*nnnn*>+1.

Example:

```
DOKE &H8100,&HAABB => cell $8100 will contain $BB and
                      cell $8101 will contain $AA
```

DRAW

```
Syntax: DRAW x1,y1,x2,y2[,color]
```

Draws a line starting from point with coordinates <*x1*>,<*y1*> to point with coordinates <*x2*>,<*y2*>. If <*color*> is specified, the line will be drawn with that color (see "COLOR" for color codes), otherwise the foreground color will be used. <*x1*> and <x2> must be in range 0~255, while <*y1*> and <*y2*> must be in range 0~191: 0,0 if the top-left corner while 255,191 is the bottom-right corner. Due to hardware limitations of the VDP "Video Display Processor", for every single portion of screen of 8x1 pixels there can only be 1 primary color: if a line of one color intersects a line of another color, the pixels of the crossing byte will get the last color used. The direction of drawing is always from <x1>,<y1> to <x2>,<y2>, regardless the position on the screen of each pair of coordinates.

Example:

```
DRAW0,0,255,191,7 => draw a diagonal from top left to bottom right in
                     cyan color
```

END

```
Syntax: END
```

Terminates the execution of a program. Any statements and/or program lines following the END statement will be ignored and control will return to the editor. There is another instruction to halt the execution of the program, STOP: the difference is that the latter interrupts the code flow with a "BREAK" message (as if the RUN STOP key would have been pressed) while END behaves has the interpreter would have reached the last line of the program.

See also: STOP

EXP

```
Syntax: EXP(x)
```

Returns “e” to the power <*x*>. <*x*> must be less than or equal to 87.3365.

Example:

```
EXP(2) => 7.38905
```

FOR...TO...STEP

```
Syntax: FOR <var> = <start> TO <end> [STEP <step>] /instructions/ NEXT 
<var>[,...]
```

The FOR..NEXT statements are used to create loops, i.e. a sequence of instructions that have to be repeated for a certain number of times. <*var*> is a valid name of a variable that will contain the value incremented during the loop and used for the ending test. <*start*>, <*stop*>, and <*step*> are expression. <*start*> is the starting value and it is assigned to <*var*> at the beginning of the loop. Then, the instructions between FOR and NEXT statements are executed. When the NEXT is reached, <*start*> is checked if it’s less than <*stop*>: if the value of <*step*> if present, it is added to <var> and its value tested against <stop>. If <var> is greater then <stop> the loop is terminated and the execution continues from the first instruction after the NEXT statement; otherwise the loop is repeated. If <*step*> is not present, the increment is assumed to be 1. If <*step*> is negative, the loop decrements the value of <*var*> going down from <*start*> to <*stop*>.

FOR..NEXT loops can be nested: the only limit is the amount of memory.

One or more loop variables can follow the NEXT statement, altough the first variable must refer to the recent loop, the second of the most recent one and so on. If no variable is present, the NEXT statement refers to the most recent FOR statement.

Examples:

```
FOR I=1 TO 10:PRINT I:NEXT I
```

Repeat the loop 10 times (from 1 to 10)

```
10 FOR I=1 TO 10
20 FOR J=1 TO 20
30 PRINT A(I,J)
40 NEXT J,I
```

Repeat 2 loops: J is the first one because is the most recent one.

```
10 FOR I=10 TO 1 STEP -2
20 PRINT I
30 NEXT
```

This loop will go from 10 to 1 with decrements of 2, so the printed values will be:

```
10
8
6
4
2
```

Note that “1” won’t be printed because the last decrement gets 0 as result of the subraction and 0 is less then 1, the <stop> value of the loop. Also note the absence of the variable after NEXT: in this case the interpreter will refer to the most recent FOR statement, that is “I”.

FRE

```
Syntax: FRE(X)
```

If <*X*> is a numerical expression, FRE returns the memory available for BASIC environments. If the expression is a string, it returns the available space in the string space. See CLEAR for more details.

Examples:

```
FRE(0) => returns the memory available for BASIC (programs and vars)
FRE("") => returns the free space in the string space
```

GOSUB

```
Syntax: GOSUB <line>
```

Jumps at <*line*> to execute a portion of code. When the interpreter will encounter the RETURN statement, it will resume the execution at the instruction following the GOSUB statement.

Examples:

```
10 GOSUB 30 => jumps to 30
20 END => resumes from here
30 PRINT "HELLO"
40 RETURN => return to caller
```

See also RETURN.

GOTO

```
Syntax: GOTO <line>
```

Makes an unconditional jump to another point of the program indicated by <*line*>.

Example:

```
10 GOTO 100
```

HEX$

```
Syntax: HEX$(arg)
```

Converts the numerical expression <*arg*> into a string containing the hexadecimal representation of the value of <*arg*>. <*arg*> must be a signed integer with value in range -32,768/+32,767.

Example:

```
HEX$(1000) => "3E8"
```

IF...THEN
IF...GOTO

```
Syntax: IF cond THEN [...]
        IF cond GOTO <line>
        IF cond THEN <line>
```

IF is a conditional branch. It is used to change the order of the execution of the instructions of a programs instead of the ordinary sequential flow. <*cond*> can be any valid arithmetic, relational, or logical expression: if it is evaulated true (i.e. non-zero), the statements after IF are executed.

There are several allowed formats of the statement. If the expression is true, anything that follows THEN will be executed:

```
10 IF A=1 THEN PRINT "A IS EQUAL TO 1":GOTO 100
```

If there is only a GOTO instruction after the expressio to be evaluated, the THEN statement can be omitted:

```
10 IF A=1 GOTO 100
```

If there is only a jump instruction (GOTO) after the IF, another form can be used where GOTO is omitted and THEN is followed by the line number <*line*>:

```
10 IF A=1 THEN 100
```

INKEY

```
Fomart: INKEY(nnnn)
```

When used inside a program (usage in indrect mode is now allowed), this function returns the ASCII code of the code being pressed by the user. <*nnnn*> can vary between 0 and 1,023 and represents the interval the function has to wait for the user's input before return control to the program. If <*nnnn*> is 0 the function won't wait any time: it will read the input buffer and return the code being present, otherwise it will wait for the corresponding number of hundredths of seconds before to return. The results are 0 for no key being pressed. The last key being pressed is always inserted into a temp buffer used by INKEY so if the user is asked to press a key AFTER a certain moment, it is reccomended to make a null read before the real one. Another good way of working is to make a little IF..THEN loop and leave it when INKEY returns 0 to be sure to read only the requested keys. SInce the function read the keys very fast, it is suggested to introduce a delay of at least 5 to let the user be able to press a key.

Examples:

```
10 INKEY(0):REM EMPTIES THE BUFFER
20 A=INKEY(10):IFA=0THEN10:REM REPEAT UNTIL A KEY IS PRESSED
30 PRINT CHR$(A):REM PRINT PRESSED KEY CODE
```

```
10 A=INKEY(500):REM WAIT A KEY FOR 5 SECONDS
20 IF A=0 THEN PRINT "NO KEY PRESSED":GOTO 40
30 PRINT "KEY PRESSED: ";CHR$(A)
40 END
```

INP

```
Syntax: INP(X)
```

Reads a byte (an 8-bit value) from the I/O port specified by the expression <*X*>. <*X*> must be in range 0-255.

Example:

```
A=INP(1) => read a byte from port 1 and assign it to A
```

INPUT

```
Syntax: INPUT [<prompt text>;]<list of variables>
```

Reads one or more data from the standard input and assign it to the same number of variables. The interpreter prompts the user with a question mark, then he/she must insert some data and press the ENTER key. If more than one data is requested, each one must be separated by a comma, and the user must be enter the exact amount of values with the correct type, separated by commas. If the data is invalid, i.e. a string when a number was expected, the interpreter will print a "REDO FROM START?" error and the user will be asked to re-enter the data; if more data was requested than entered, a "??" will be printed and the interpreter will ask for the missing ones; if more data was entered than requested, an "EXTRA IGNORED" advice will be printed and the execution will continue discarding the extra values. If a prompt text is passed, it will be printed on screen before the question mark. The prompt text must be enclosed in double quotation marks and followed by a semicolon.

Examples:

```
INPUT "WHAT'S YOUR NAME";NAME$ => wait for a string
```

```
INPUT "NAME,AGE";NAME$,AGE => wait for 2 data (a string and a number)
```

```
INPUT A => wait for a number with just a "?"
```

INT

```
Syntax: INT(X)
```

Returns the integer part of <*X*>. <*X*> must be a numerical expression. The result is obtainded by truncating <*X*> to the decimal point. If <*X*> is negative, the round is made to the first integer greater than <*X*>.

Examples:

```
INT(3.14) => 3
INT(-3.14) => -4
```

LEFT$

```
Syntax: LEFT$(str,val)
```

Returns a string that contains the <*val*> characters to the left of the string <*str*>.

Example:

```
LEFT$("HELLO",2) => "HE"
```

See also RIGHT$ and MID$.

LEN

```
Syntax: LEN(str)
```

Returns the lenght of the string <*str*>.

Example:

```
LEN("HELLO") => 5
```

LET

```
Syntax: LET <var>=<val>
```

Assigns the value <*val*> to the variable with name <*var*>. The use of LET is optional and this statement can be omitted.

Example:

```
LET A=10   is equal to   A=10
```

LINES

```
Syntax: LINES <num>
```

Sets the number of lines to print before to pause the listing of a program made with LIST: see below.

LIST

```
Syntax: LIST [start]
```

Lists a program stored in memory. If <*start*> is passed, the list will start at the line set by <*start*>, otherwise from the first line of the program. LIST will print the number of lines set with LINES then will pausing, waiting for a key.

Examples:

```
LIST  => list the whole program
LIST 100  => list from 100 to the end of the program
```

See also: LINES

LOCATE

```
Syntax: LOCATE x,y
```

Places the cursor on the screen at coordinates <*x*>,<*y*>. LOCATE works only in graphic modes 0 & 1. <y> is in range 0~23, while <x> is in range 0~31/39, depending on which mode is active at the moment: graphic 1 is 32 chars wide while graphic 1&4 are 40 chars wide. Coordinates 0,0 point to the top left corner. The usage of this command makes no sense in screen modes 2 & 3 since they don't have any cursor support.

Examples:

```
LOCATE 0,0 => place the cursor in the top left corner
LOCATE 0,9 => place the cursor at the first cell of the 10th row
```

LOG

```
Syntax: LOG(expr)
```

Returns the natural logarithm of <*expr*>. <*expr*> must be an expression whose value is greater than 0.

Example:

```
LOG(10) => 2.30259
```

MID$

```
Syntax: MIDS$(str,arg[,end])
```

Returns a portion of string <*str*>. If <*end*> is omitted, it returns the <arg> right-most characters of <str>; if <end> is passed, it returns the chars between the <*arg*>th and the <*end*>th char.

Examples:

```
MID$("HELLO",3) => "LLO"
MID$("HELLO",2,2) = > "EL"
```

NEW

```
Syntax: NEW
```

Deletes the current program in memory and clears every variable. Used before to enter a new program without the need to reset the system.

Example:

```
NEW
```

NEXT

```
Syntax: NEXT <list of variables>
```

Used in FOR loops. See FOR for details.

NOT

```
Syntax: NOT arg
```

Logic operator used in boolean expressions. NOT performs a logical negation, or bitwise complement, of <*arg*>. The interpreter supports 4 boolean operators: AND, OR, XOR, and NOT, each of them work with 16-bit signed integers. They convert their inputs in 16-bit integers and return a value in such format. NOT returns the complement of value <*arg*>, meaning that the result is true when <*arg*> is false and vice-versa. The truth table of NOT operator is shown below (1=T=true, 0=F=false):

```
NOT 1 => 0
NOT 0 => 1
```

Since the interpreter works with 16-bit signed numbers, one could think that in some cases it works in a strange manner:

```
NOT 1 => -2
NOT 0 => -1
```

Indeed, the results are OK. NOT 0 should be expected to return 1 but, since the two's complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1, the result is -1. Similarly, NOT 1 should be expected to return 0 but since the 16-bit value of 1 is the binary 0000000000000001, the bit complement of it is 1111111111111110 that is the two's complement of -2.

Eventually:

```
NOT X
-(X+1)
```

are equivalent.

See also AND, OR, and XOR.

ON

```
Syntax: ON expr GOTO <list of lines>
        ON expr GOSUB <list of lines>
```

Used in conjunction with GOTO and GOSUB to introduce a series of unconditional jumps (GOTO) or calls to subroutines (GOSUB). <*expr*> must be a numerical expression: the jump is executed by calling the line whose position corresponds to the value of <*expr*>. The lines must be separated by colons.

Examples:

```
ON A GOTO 100,200,300
ON B GOSUB 1000,2000,3000
```

OR

```
Syntax: arg1 OR arg2
```

Logic operator used in boolean expressions and bitwise operations. OR performs a logical disjunction operation. It returns a true value when one or both the expressions are true. The interpreter supports 4 boolean operators: AND, OR, XOR, and NOT, each of them work with 16-bit signed integers. They convert their inputs in 16-bit integers and return a value in such format. Its truth table is the following (1=T=true, 0=F=false):

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

Example:

```
4 OR 2 => 6 (4 is 100b, 2 is 10b, so 100b OR 010b = 110b, that is 6 in
            decimal representation)
```

See also AND, NOT, and XOR.

OUT

```
Syntax: OUT port,value
```

Sends <*value*> to the peripherical device connected to output port <*port*>. <*port*> and <*value*> must be expressions with values in range 0~255.

Example:

```
OUT 1,100
```

PAUSE

```
Syntax: PAUSE arg
```

Forces the system to halt and wait for a specific interval of time set by <*arg*>. <*arg*> must be a numerical expression in range 0~65,535 that represents the number of 100ths of a second to wait.

Examples:

```
PAUSE 1000 => wait for 10 seconds (1,000 x 0.01s)
PAUSE 0 => no wait
```

The latter is similar to the assembly instruction NOP since it doesn’t do any more than the execution of the base code.

PEEK

```
Syntax: PEEK(nnnn)
```

Returns the byte stored in RAM at location <*nnnn*>. <*nnnn*> must be a numerical expression in range -32,768 to 32,767 (i.e. a 16-bit signed integer).

Example:

```
PRINT PEEK(8000) => prints the contents of address 8000.
```

PLOT

```
Syntax: PLOT x,y[,color]
```

Plots a pixel onto the screen. This command only works in SCREEN mode 2. <*x*> and <*y*> are the coordinates where to set the pixel. <*color*> is optional: if not passed, the color used is the default one set with the COLOR command or, in case you are plotting into an 8x1 pixel area that already has a pixel colored with a color different from the foreground color set with COLOR, the former will be used. <*x*> must be in range 0~255 while <*y*> in range 0~191. The coordinates 0,0 correspond to the pixel in the top left corner of the screen.

Example:

```
PLOT 0,255  => plot a pixel into the top right corner of the screen
```

POKE

```
Syntax: POKE nnnn,val
```

Writes the byte <*val*> into the location of RAM whose address is <*nnnn*>. <*nnn*> must be a numerical expression in range -32,768 to 32,767 (i.e. a 16-bit signed integer), <*val*> an expression whose value is in range 0~255.

Example:

```
POKE -28672,128 => writes 128 into memory cell located at 36,864
```

Please note that numbers are 16-bit signed integers so everything greater than 32,767 must be provided in two's complement format. A quick way to get the unsigned value of <nnnn> is to add it to 65,536:

```
-10,000 => 65,536 + (-10,000) = 55,536
```

POS

```
Syntax: POS(X)
```

In origin it returned the current position of the terminal's printer head. Now it returns the horizontal position of the cursor on the internal buffer line. The interanl buffer line is a special temporary memory used byt the interpreter to store the text found on a screen line when the RETURN key is pressed: yhe text is copied into the buffer and evaluated. Similarly, any text generated by the interpreter is put in this buffer before to be printed on video. Note: it doesn’t return the position of the cursor on the screen, and it could change its behaviour in future releases.

PRINT

```
Syntax: PRINT [data|variables|text|operations]
```

The PRINT statement prints something on screen. It supports different types of expressions: it can print the contents of variables, text included between double quotation marks, the results of numerical expressions, numerical literals. The interpreter considers the printing line divided in zones of 14 spaces each: if expressions are separated by commas the interpreter prints each expression at the beginning of such spaces. If a semicolon is used (;) the expressions are printed one after another. If neither of them are present, then the interpreter will go to the next line after the ending of the print statement.

Examples:

```
PRINT => just go to next line
PRINT "HELLO" => prints HELLO and then a carriage return is inserted

PRINT A => prints the value of variable A

PRINT "LENGHT:";LN;" METERS" => prints LENGHT followed by the contents
                                of LN and then by METERS

PRINT A,B => prints the contents of A and B with tabulation
```

The print begins from the current cursor position. Is the text being printed exceeds the size of the line (32 or 40 chars), the printint will continue from the beginning of the next line. If the printing involves the last line of the screen, the entire contents of the screen will be scrolled one row up when the bottom right cell will be filled up.

It is possible to change the position of the cursor by using the LOCATE statement.

See also: LOCATE.

READ

```
Syntax: READ <list of variables>
```

Reads the informations stored into the program with DATA statement. <*list of variables*> is a set of variables names separated by commas. The The effect of READ statement is to read the value introduced by DATA and store it into the variables following READ, from left to right. If the informations to be read are less than the variables names an error will be raised. If there are more values stored in DATA than are read from a READ, the next READ will continue to read from the next unread data. Types of variables and informations must be coherents (i.e. a string can not be assigned to a numerical variable).

Example:

```
10 DATA 10,20,"HELLO"
20 READ A,B,C$ => 10 will be written into A, 20 into B, and "HELLO" in C
```

See also RESTORE.

REM

```
Syntax: REM <comment>
```

Used to enter a comment into the program. Everything that follows the REM statement will be ignored until the end of current program line.

Example:

```
10 X=10:REM SETS HERE THE STARTING VALUE
```

RESET

```
Syntax: RESET
```

Performs a system reset. Equivalent to pushing the reset button, with the only difference that the software reset doesn't reset the peripheral chips of the computer but simply re-initialize them by executing the boot code.

RESTORE

```
Syntax: RESTORE [line_num]
```

After a RESTORE statement the next information read with a READ statement will be the first value of the first DATA into the program. If <*line_num*> is passed, the next information will be read from such program line: this permits to read the same informations several times.

Example:

```
10 DATA 10,20,30
20 READ A => A will contain 10
30 RESTORE: READ B => B will also contain 10
```

See also DATA and READ.

RETURN

```
Syntax: RETURN
```

Used to leave a sub-routine called by GOSUB. After the RETURN statement, the execution of the program will continue with the instruction following the GOSUB statement.

Example:

```
10 GOSUB 100
20 PRINT "FINISH":END
100 PRINT "START"
110 RETURN  => the execution will continue from line 20
```

RIGHT$

```
Syntax: RIGHT$(str,val)
```

Returns a string that contains the <*val*> characters to the right of the string <*str*>.

Example:

```
RIGHT$("HELLO",2) => "LO"
```

See also LEFT$ and MID$.

RND

```
Syntax: RND(val)
```

Returns a random number between 0 and 1. A negative value of <*val*> will start a new sequence; if <*val*> is greater than zero the function will return the next value in the random sequence; a value of zero will return the last number returned. The same negative number will generate the same random sequence.

Examples:

```
A=RND(1) => returns a random number
A=INT(RND(1)*6)+1 => rolls a dice: will return a number between 1 and 6.
```

If you need to create a good random sequence you can use the system tick timer used to increment the 100ths of second counter:

```
RND(-ABS(TMR(0)))
```

See also TMR.

RUN

```
Syntax: RUN [numline]
```

Start the execution of the program currently stored in memory. If <*numline*> is present, the execution will start from such line, otherwise it will start from the first line.

Example:

```
RUN  => start the execution of the program from the first line
RUN 1000 => start the execution of the program from line 1000
```

SCREEN

```
Syntax: SCREEN mode[,spriteSize][,spriteMagn]
```

Changes the screen mode. <*mode*> is a number between 0 and 4 and sets the screen as follow:

0: 40x24 chars text mode (no sprites support)

1: 32x24 chars text mode or 256x192 pixels graphic mode (sprites supp.)

2: 256x192 pixels graphic mode (sprites supported, bitmap mode)

3: 64x48 pixels multicolor graphic mode (sprites supported)

4: 256x192 pixels text/graphics mode (limited sprites support)

SCREEN initializes the screen with default settings, meaning that it sets for each mode specific foreground, background, and border colors: in modes 0, 1, & 4, the default settings include background and border colors set to light blue while text set to white; in modes 2 the background is white, the foreground is black and the border is light blue; in mode 3 the background is white and the border is light blue, while no specific foreground is set. In modes 0 & 1 it also loads and configures a complet charset with 256 chars, since these two are also text modes: the big difference is that mode 0 is a real text mode, with no support for sprites and graphics, while mode 1 is a graphics mode where each char is in reality a tile. In every mode, SCREEN also performs a screen clear.

SCREEN 4 is a special, not ufficially supported yet still documented mode that is a mix between SCREEN 1 and SCREEN 2. In SCREEN 4 the video buffer is divided into 768 cells each of those can be assigned to one 8x8 pattern, like in mode 1, but where the colors are managed for single bytes (8x1 pixels). It lacks in sprite supporting: only 8 sprites can be used, if you try to some more they will start to duplicate themselves.

Examples:

```
SCREEN 0  => sets the 40x24 chars text mode
SCREEN 2  => sets the bitmap graphic mode
```

If <*spriteSize*> and <*SpriteMagn*> are passed, then SCREEN also sets the size and magnification attributes for sprites. Arguments can assume value of 0 or 1. If <*spriteSize*> is 0 then sprites are set to 8x8 pixels, while if it's 1 then sprites are set to 16x16 pixels (this is obtained combining together 4 8x8 sprites). If <*spriteMagn*> is 0 then no sprite magnification is set, while if it's 1 than sprite magnification is set to ON: this means that 8x8 sprites become 16x16, and 16x16 sprites become 32x32.

Note the sprites magnification halves their video resolution so that when a sprite is magnified each pixel occupies a 2x2 pixels on the video grid.

You can just pass one argument of the two, in this case it is assumed that the argument is the sprite size.

Examples:

```
SCREEN 1,0,0 => this corresponds to SCREEN 1, sprites are set to 8x8
                pixels and sprite magnification is off
SCREEN 1,1,0 => screen mode 1 with sprites size set to 16x16 (the last
                argument can be omitted)
SCREEN 1,1   => same as above
SCREEN 1,0,1 => sprites size set to 8x8, sprite magnification on
```

Obviously, the sprite size and magnification arguments are revelant only when used in graphics modes that support the sprite visualization: for this reason, when setting the text mode 0 only <mode> is accepted.

Further reading: to better know the video capabilities of the VDP the reading of its reference guide is reccomended. A copy can be found here:

https://github.com/leomil72/LM80C/tree/master/manuals

See also: CLS and COLOR

SERIAL

```
Syntax: SERIAL ch,bps[,data,par,stop]
```

Opens a serial connection between the computer and an external device. <*ch*> can be 1 or 2: 1 corresponds to SIO channel A, while 2 to SIO channel B. In LM80C computer the channel A is connected to a USB-to-serial converter so that serial port 1 works only as a char device, while channel B (port 2) is configured as a block device. <*bps*> indicates the bauds per second, i.e. the speed of the serial line. <*bps*> can assume one of the following values:

```
57600,38400,28800,19200,14400,9600,4800,3600,2400,1200,0
```

If 0 is entered, the other arguments are ignored and the command closes the connection opened on channel <*ch*>. Otherwise, the serial line will be set to run at the selected speed. <*data*> represents the number of bits that compose the data to be sent: allowed values are 5/6/7/8. 5 is used to instructs the SIO to work with a number of 5 bits (or less) per single char. <*par*> is the parity bit: it can be 0/1/2. 0 means no parity bit; 1 means an odd parity; 2 means an even parity. <*stop*> sets the number of bits sent after the data bits. Its value must be: 0, for no stop bits; 1, for 1 stop bit; 2, for 1.5 stop bits; 3, for 2 stop bits. Opening a serial line leads to the corresponding status LED to be turned on.

Example:

```
SERIAL 1,19200,8,0,1  => normal settings to open a serial line with
                         19,200 bps, 8 data bits, no parity bits,1 stop
                         bit
SERIAL 1,0 => closes the serial port 1, resetting the SIO channel A.
```

Further reading: to better know the serial capabilities of the SIO chip, the reading of the Z80 peripherals' user manual is reccomended. A copy can be found here:

https://github.com/leomil72/LM80C/tree/master/manuals

SGN

```
Syntax: SGN(arg)
```

Returns the sign of <*arg*>: it returns -1 if <*arg*> is negative, 1 if it is positive, or 0 if it is zero.

Example:

```
PRINT SGN(-345) => -1
```

An interesting usage of SGN is in combination with conditional jumps where a specific set of istructions can be executed depending of the sign of a variable being checked:

Example:

```
10 ON SGN(X)+2 GOSUB 100,200,300
```

In this example, the program will continue from line 100 if X<0, from line 200 if X=0, and from line 300 if X>0.

SIN

```
Syntax: SIN(arg)
```

Returns the sine of <*arg*>. The result is in radius in range -pi/2 to pi/2.

Example:

```
SIN(5) => -0.958924
```

SOUND

```
Syntax: SOUND ch,tone,dur
```

This command instructs the PSG to emit a sound with a particular tone for a specific duration on the selected channel. <*ch*> can be 1,2, or 3, and corresponds to the same analog channel of the PSG. <*tone*> can vary between 0 and 4,095 and it is inversely related to the real frequency of the emitted tone. The formula to get the frequency is below:

```
Freq = 1,843,200 / 16 / (4,096 – tone)
```

where 1,843,200 Hz is the clock of the PSG while "16" is a fixed internal prescaler

I.e.: if tone is equal to 4,000, the frequency of the sound generated by the PSG will be 1,200 Hz. In fact, 1,843,200 / 16 / 96 = 1,200. The lower frequency will be 28 Hz while the higher frequency will be 115,200 Hz (115 KHz). Obviously, anything over 15/20 KHz won't be audible.

<*dur*> is the duration in 100ths of a second and its range is between 0 and 16,383 (0.0~163.8 seconds). By setting a tone of <*tone*> 0 we force the audio to quit immediately. By setting a tone with duration equals to 0, the tone will last forever, unless you quit the volume off or you set another tone on the same channel.

The inverse formula to calculate the value <*dur*> to pass to SOUND to generate a tone of frequency <*freq*> is:

```
dur = 4,096 - (1,843,299 / 16 / (freq))
```

I.e.: if a tone with a frequency of 2,000 Hz (2 KHz) is required, <*dur*> equals to

```
dur = 4,096 - (1,843,200 / 16 / 2,000)) = 4,096 - 57.6 => 4,038(.4)
```

In fact, if we use the previous formula, we get:

```
Freq = 1,843,200 / 16 / (4,096 - 4,038) = 115,200 / 58 = 1,986 Hz
```

Due to integer truncatings, the frequency is a little bit smaller than needed

Example:

```
SOUND 1,3500,100 => plays a tone of freq. 193 Hz for 1 second.
```

There is a speciale usage of SOUND. If used with just a 0 as argument, SOUND will reset the PSG registers and shut down every sound generated by the audio chip, including the white noise and the envelops activated with SREG.

Example:

```
SOUND 0  => shut down every king of sound/tone
```

SPC

```
Syntax: SPC(val)
```

Prints <*val*> empty chars on video. It can only be used in conjunction with a PRINT statement. Please keep in mind that TAB and SPC are similar but have different behaviours since the former moves the cursor without altering the chars before the position to reach while SPC prints empty chars deleting every char during the movement.

Example:

```
PRINT "A";SPC(5);"B"
A     B
```

See also: TAB

SQR

```
Syntax: SQR(arg)
```

Returns the square root of <*arg*>. <*arg*> must be greater than zero.

Example:

```
SQR(9) => 3
```

SREG

```
Syntax: SREG reg,val
```

Writes the byte <*val*> into the register <*reg*> of the PSG. <*reg*> must be in the range 0-15 while <*val*> in the range 0-255.

Example:

```
SREG 8,15 => set the volume of analog channel A to 15
```

See also SSTAT.

SSTAT

```
Syntax: SSTAT(reg)
```

Reads the PSG (Programmable Sound Generator) register set by <*reg*> and returns a byte. <*reg*> must be in the range 0-15. Some words must be spent about the register numbering. Most of the PSG datasheets found report registers are numbered from 000 to 007 and from 010 to 017, without mentioning that the “octal” format is used to indicate the registers, leading in mis-understanding with the PSG that seems to operatestrange behaivours only because the wrong register indexes are used. So, simply the registers are 16 and are numbered from 0 to 15 in decimal format. Please remember that registers #14 and #15 are used to read the keyboard so they shouldn't be used.

Example:

```
SSTAT(8) => return the value of register #8
```

See also SREG.

STOP

```
Syntax: STOP
```

Halts the execution of a program with a “BREAK” message, as if the RUN/STOP key would have been pressed. See CONT on how to resume running. There is another instruction to halt the execution of the program, END: the difference is that the latter behaves has the interpreter would have reached the last line of the program.

See also: END.

STR$

```
Syntax: STR$(arg)
```

Converts to a string representation the expression <arg>.

Example:

```
STR$(12) => "12"
```

SYS

```
Syntax: SYS address[,value]
```

Calls a machine language routine starting at <*address*>. <*address*> must be a signed integer (i.e. in range -32,768 to +32,767). If <*value*> is passed, it will be passed to the routine into the Accumulator register. <*value*> must be a byte value (0~255). SYS differs from USR in the way that it's a command so there is no return value to be collected.

Example:

```
SYS &HB000,100
```

See also: USR

TAB

```
Syntax: TAB(val)
```

Moves the cursor to column <*val*> on the video. <*val*> must be in the range 0~255: 0 means no movements.

Example:

```
PRINT TAB(5);"*" =>
     *
```

Please keep in mind that TAB and SPC are similar but have different behaviours since the former moves the cursor without altering the chars before the position to reach, while SPC prints empty chars deleting every char under the cursor during its movement.

See also: SPC

TAN

```
Syntax: TAN(arg)
```

Returns the tangent of <arg>. The result is in radians in range -pi/2 to pi/2.

Example:

```
TAN(1) => 1.55741
```

TMR

```
Syntax: TMR(val)
```

Returns the value of the system tick timer, that is a 32-bit counter that is incremented every hundredth of a second. Since the BASIC interpreter that only manage 16-bit values, the system timer can be read by considering it divided into 2 halves: if <val> is 0 than the function will return the first two less significant bytes of the counter. If <val> is 1, than the two most significant bytes will be returned instead.

Example:

```
TMR(0) => 3456
```

Since the BASIC operates with signed integers, values returned by TMR go from -32768 to +32767. To get the unsigned counterpart, if the value returned is negative, you can add it to 65536 to get a value in the range 0 ~ 65535.

Example:

```
10 A=TMR(0):IF A<0 THEN A=65536+A
```

The system counter can be used to measure the passing of time by using the whole 32-bit value of the system tick timer. Also, if you divide this value by 100 you get the numbers of seconds elapsed since the computer has been powered on.

Examples:

```
PRINT (TMR(1)*65536+TMR(0)) => 273636  (100dths of seconds)
PRINT INT((TMR(1)*65536+TMR(0))/100) => 2736  (seconds)
```

USR

```
Syntax: USR(arg)
```

Calls an user-defined machine language subroutine with argument <*arg*>. <*arg*> is mandatory: even if it's not used, it must be passed to USR. The call of the subroutine is made through an entry point in RAM that must be initialized with the address of the first cell where the user code is stored in RAM. The entry point is located at locations $8049 and $804A: the address must be set using the little-endian order, meaning that the less significant byte must be stored into $8049 while the most significant byte must be stored into $804A. Suppose that a subroutine has been stored in RAM from address $B0A0. Then the byte $A0 will be stored at $8049 while byte $B0 will be stored at $804A. USR is a function, so it must be called in a way that can collect the possible returned value.

Example:

```
A=USR(0) => call a subroutine with argument 0 and assign its returned value 
to A.
```

After a system reset, the USR points, by default, to a call to an “Illegal function call” error. This is done to avoid system hangs by calling the function without setting it properly.

See also: SYS

VAL

```
Syntax: VAL(str)
```

Returns the numerical value of string <*str*>. If the first character of <*str*> isn't a “+”, an “&”, or a digit then the result will be 0.

Examples:

```
VAL("12") => 12
A$="a":PRINT VAL(A$) => 0, because A$ can not be represented as a number
```

VOLUME

```
Syntax: VOLUME chn,val
```

Sets the volume of the selected channel <*chn*> to value <*val*>. <*chn*> must be in the range 0~3: 1,2, and 3 works on the corresponding analog channel, resepectively. If <chn> is 0, the statement will apply to all the channels. <*val*> must be a numerical value between 0 (no audio) and 15 (max volume).

Example:

```
VOLUME 1,15 => sets the volume of channel 1 to the max. value
VOLUME 0,0 => quits the volume of all the channels.
```

Please note that by quitting the volume of a sound channel doesn’t mean that the sound generation has been stopped on such channel. To also stop the sound generation SOUND must be used.

Example:

```
10 VOLUME 1,15 : REM CHANNEL 1 VOLUME SET TO MAX
20 SOUND 1,3000,0 : REM GENERATING A TONE WITH NO ENDING
30 PAUSE 200 : REM A PAUSE OF 2 SECONDS
40 VOLUME 1,0 : REM SET CHANNEL 1 VOLUME TO OFF
50 PAUSE 200 : REM ANOTHER PAUSE OF 2 SECONDS
60 VOLUME 1,15 : REM SET AGAIN VOLUME TO MAX – SOUND IS STILL PRESENT
70 PAUSE 200 : REM ANOTHER PAUSE OF 2 SECONDS
80 SOUND 0 : REM SOUND FROM ALL CHANNELS ARE OFF
```

VPEEK

```
Syntax: VPEEK(nnnn)
```

Reads a value from the VRAM (or Video-RAM), the memory that is at exclusive use of the VDP (Video Dislay Processor). <*nnnn*> is a value between 0 ($0000) and 16,383 ($3FFF), since the VRAM is 16 KB wide. It returns a byte value (range 0~255).

Example:

```
A=VPEEK(0)
```

Note: since the VRAM (or Video-RAM) is dedicated to the VDP, it can be accessed only by this chip so it is out of the normal address space of the CPU, therefore specific instructions to read from or write to VRAM have been implemented.

See also: VPOKE.

VPOKE

```
Syntax: VPOKE nnnn,val
```

Like its POKE counterpart, VPOKE writes into VRAM (or Video-RAM). VPOKE write the value <*val*> into the cell of VRAM whose address is <*nnnn*>. <*nnnn*> must be in the range 0~16,383, while <*val*> is a byte value (0-255).

Example:

```
VPOKE 1000,100
```

VREG

```
Syntax: VREG reg,value
```

Writes <*value*> into the VDP register <*reg*>. <*reg*> must be a number in the range 0~7, while <*value*> is a byte (0~255).

Example:

```
VREG 7,15 = writes 15 into register #7
```

Further reading: to better know the serial capabilities of the VDP chip, the reading of the TMS9918A user manual is reccomended. A copy can be found here:

https://github.com/leomil72/LM80C/tree/master/manuals

VSTAT

```
Syntax: VSTAT(x)
```

Reads the status register of the VDP. <*x*> is ignored and can be any signed integer (between -32,768 and +32,767). This is due to the fact that there is only a read-only register in VDP so any value of <*x*> will always refer to the same register. The function returns a byte value (0-255).

Example:

```
?VSTAT(0)
```

WAIT

```
Syntax: WAIT port,Vor[,Vand]
```

Reads the I/O port <*port*> and performs and OR between the byte being returned and <*Vor*>. If <*Vand*> is passed, the result of the OR operation is then ANDed with <*Vand*>. The execution continues with a non-zero result.

Example:

```
WAIT 0,0,128 => the execution continues only if port 0 returns 128
                because 128 OR 0 = 128 and 128 AND 128 = 128.
```

WIDTH

```
Syntax: WIDTH val
```

Sets the lenght of a line for inputs or outputs. <*val*> must be in the range 0~255. Default is 255. Please consider that this value doesn’t affect the lenght of the program lines being but only the lenght of lines on the terminal console, if a serial port is connected to a remote device.

XOR

```
Syntax: arg1 XOR arg2
```

Logic operator used in boolean expressions and bitwise operations. It performs a logical exclusive disjunction between expression <*arg1*> and <*arg2*>. The interpreter supports 4 boolean operators: AND, OR, XOR, and NOT, each of them work with 16-bit signed integers. They convert their inputs in 16-bit integers and return a value in such format. It returns a true value only when an odd number of inputs are true, meaning that its results are true only when its operators are one false and one true, and it returns a false value when its operators have the same value. Its truth table is the following:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

Example:

```
11 XOR 2 => 9 (11 is 1011b, 2 is 0010b, so 1011b XOR 0010b is equal to
              1001b, that is 9 in decimal representation)
```

XOR is commonly used in cryptographic algorithms since one of its features is to revert a XORed bit to its original value when it's XORed again with the same value: first, we XOR a value with a "key" to get an encrypted result, then we revert to the original value the encrypted result by XORing it again with the "key". I.e.: 1 XOR 1 = 0 and 0 XOR 1 = 1. Let say that the "key" is value 167 and let say that we have to encrypt the text "A". Since the ASCII code of "A" is 65, the operation is:

```
65 XOR 167 = 230
```

Now, to revert to the original text, let's XOR 230 with the same "key":

```
230 XOR 167 => 65
```

See also: AND, OR, and NOT.

---

5. USEFUL LINKS

Project home page:

https://www.leonardomiliani.com/en/lm80c/

Github repository for source codes and schematics:

https://github.com/leomil72/LM80C

Hackaday page:

https://hackaday.io/project/165246-lm80c-color-computer

LM80C
Color Computer

Enjoy home-brewing computers