LM80C BASIC

Language Reference Manual

LM80C BASIC
LM80C 64K 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
and LM80C 64K Color Computers

This release covers the
LM80C BASIC 3.15 (21/11/2020)
LM80C 64K BASIC 1.02 (31/12/2020)

Last revision on 31/12/2020

Copyright notices:

Index

1. OVERVIEW

1.1 Preface

This manual is a language reference for the **LM80C / LM80C 64K BASIC** (from now on, only LM80C BASIC) interpreter built into the **LM80C / LM80C 64K Color Computers** (from now on, only LM80C) a couple of home brew computers designed by Leonardo Miliani. So, 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 output. Everything you digit on the keyboard or is output 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 the LM80C Color Computers basically is like a home-computer of the 8-bit era, so it will always and primarily get input from keyboard and print output on the screen: please adapt/modify the source you are inserting to take care of such behavior.

1.3 Memory

NASCOM Basic was originally released for machines that had a limited amount of RAM, commonly 4/8 KB. Surely, they could be expanded and several users did it, reaching 16/24/32 KB of memory and more, but this wasn't the standard. The LM80C actually comes with 64 KB of SRAM while the LM80C 64K with 64 KB, so don't hesitate to use all of this room. Just keep in mind that the original limit on the length of an input line has been changed from 72 to 88 chars, so this facilitate the input of big 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 it 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 or64 KB of SRAM
- 32 KB of EEPROM with:
    - firmware to drive video/audio/serial/parallel 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 modern 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 internal 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 little beep: during this short time, a check of the system LEDs will be 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 free RAM for BASIC programs (this value may vary from release to release). The text below is printed by LM80C 64K firmware R1.0:

```
    LM80C 64K Color Computer
 by Leonardo Miliani * FW R1.02
LM80C BASIC 64K © 2020 L.Miliani
 Z80 BASIC 4.7 © 1978 Microsoft
        43589 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:

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

If the cold start is chosen, the system will print the copyright notices with the available free memory; if the warm start is chosen, 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 between 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
LINES: set number of lines to be printed simultaneously
MONITOR: it launched the MONITOR program
NULL: set the null chars to be printed at the end of a line
PSET: set a video pixel on
```

Some statements now have a different behavior:

```
POINT: returns the state of a video pixel
RESET: resets the system
SCREEN: changes the video mode
```

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*"). The complete list of the statements available in LM80C BASIC is available in chapter 3.

2.2 Modes of operation

The LM80C BASIC operates in two different modes. In ***direct mode*** it executes the commands as they are entered at the prompt and prints the results of the statements directly on the screen. This mode is useful if you intend to use the computer as a “calculator” but the instructions are lost as soon as they are executed.

In ***indirect mode*** the computer executes the instructions from a program stored into the main memory.

Not all the statements can be executed in direct and indirect mode. As an example, the “INPUT” statement can only be executed inside a program (indirect mode), while the “CLS” command, that clears the screen, can be executed in both direct and indirect modes.

2.3 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 a hexadecimal number is introduced by the prefix `&H`:

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

There is also another way to manage hexadecimal and binary numbers: by using the `VAL` function, that returns the value of a string expression, you can use the trailing "`$`" char for hexadecimal values and "`%`" for the binary values. See VAL for more informations.

2.4 Functions

A ***function*** is a statement that gets an input argument and returns another data, that depends on the way it processes the input. A function can not be used in direct mode, i.e. this statement is not valid:

```
INPUT A
```

and will return an error.

The followings are the new functions that weren't present on the original NASCOM BASIC:

```
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
```

2.5 Commands

Commands are statements that tell the system to perform a specific operation. Commands can get some arguments but usually don't return any value. Commands can be used in direct or indirect mode. Here are the new commands added to manage the features of the hardware of LM80C:

```
CIRCLE: draws a circle
COLOR: set the foreground, background, and border colors
CLS: clears the screen
DRAW: draws a line
HELP: prints the line where an error has occurred
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
```

2.6 Line input

A number before a single or a list of statements introduces a ***program line***. Any line starting with a number is interpreted as a program line. Line numbers must be in range 0 to 65,529.

Examples:

```
10 PRINT "HELLO"
```

When RETURN is pressed, the line is stored into memory. If a line with the same number is already present in memory, then the new one will overwrite the current line into memory. If we digit the following line with the same number of the previous one:

```
10 PRINT "WORLD"
```

Then, when running this program, we will get the following result:

```
WORLD
```

When a number is typed in with anything following but a RETURN, it will be interpreted as a direction to the BASIC to remove from the program the line number whose number is being entered. The example below will remove the line 10:

```
10
```

If such line is not present in memory, then an “Undefined Line Error” will be raised.

2.7 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
```

If the number is positive, then an empty cell will be printed as leading char:

```
PRINT 1.2
 1.2
```

LM80C BASIC integers are 16-bit signed intergers. When printing integer values, i.e. numbers that can be represented using 16 bits, keep in mind that the most significant bit is used for the sign: this means that they can only range between -32,768 and +32,767. Also, consider that when managing such numbers, if you want to use values bigger than 32,767 you have to subtract the number from 65,536. In the example below we try to write a 16-bit unsigned value into RAM:

```
DOKE 45056,45056
?Illegal Function Call Error
```

We get an error because 16-bit integers are managed as signed integers and 40,056 is bigger then the biggest positive signed integer allowed (+32,767).

So, to do such operation we can go through 2 different ways. The first one is to use hex numbers:

```
DOKE &HB000,&HB000
Ok
```

The second one is to use 16-bit signed integers, and to do this we must subtract 65,536 from our value and use the results:

```
?45056-65536
-20480
DOKE -20480,-20480
Ok
```

To get back the word from memory cells 45,056 & 45,057 we must do something similar and get similar results:

```
?DEEK(50000)
?Illegal Function Call Error
?DEEK(-20480)
-20480
?DEEK(&HB000)
-20480
```

In the same manner, to get back the results in 16-bit unsinged integer we must subtract the value from 65,536:

```
?65536-20480
45056
```

Pay great attention that this is not an issue since the BASIC interpreter manages numbers correctly when operating arithmetical operations. You should only consider that numbers that can be stored into 16 bits are represented as signed numbers during input/output, so consider this when interact with the user.

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 convention rules for the sign seen above are also followed for the mantissa. 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 2.9387E-38.

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

```
+1                 1
-1                -1
1000000            1E+06
-12.34567E-10     -1.23456E-09
.01                .01
.000123            1.23E-04
```

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

2.8 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 explicitly by the programmer or assigned as the result of some calculations. The name of a variable may be any length but the alphanumeric characters after the first two are ignored: this means that 2 different variable names that have the same two leading chars will refer to the same variable into memory:

```
WTS          (interpreted as WT)
WTP          (interpreted as WT, too)
```

They are interpreted as the same variable and so they will both refer to the same value into memory:

```
10 WTS=10
20 WTP=20
```

```
30 PRINT WTS
RUN
 20
```

Also, the first character of the name must be a letter, and no reserved words may be used as a 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
RGOTOX      it contains a reserved word (GOTO)
```

2.9 Strings

Despite a numerical variable, that can only store numbers, a ***string*** is a sequence of alphanumeric characters like letters, numbers, punctuation, 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 trailing “$” in the name identifies 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 OF THE WORD HELLO
30 C$=A$+B$:PRINT C$
```

Running the program above will 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 of 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, so `" a" > "a"` is False: note the leading space in the first string.

A string can be up to 255 chars. However, 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.

2.10 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 dimension 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)
```

Also, an array can be resized 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 to re-dimension an array is made in the program:

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

A string array can be defined with DIM statement: each element of the array is a string that may be up to 255 chars:

```
10 DIM A$(10)
20 A$(1)="HELLO":A$(2)="WORLD"
```

2.11 Operator 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 table below have the same precedence and are evaluated from left to right in an expression:

- `( )` → parentheses
- `^` → exponentiation
- `-` → Negation: this is the unary minus operator
- `*, /, %, #` → multiplication, division, modulo, integer division
- `+, -` → addition, subtraction
- 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 return 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 for truth tables of each operator.

2.12 Operators

Let’s get a more accurate overview on the operators supported by LM80C BASIC. Mathematical operators permit to make math calculations like additions, subtractions, and similar:

+ → “plus” operator
Make the sum of two numerical expressions. The results can be assigned to a variable or printed as is. Examples:

```
A=12+3
PRINT 12.6+7.8
```

The “plus” operator can also be used to concatenate two strings (see chapter 2.9)

- → “minus” operator

Subtracts the value of the second expression from the first one. The results can be assigned to a variable or printed as is. Examples:

```
A=12-3
PRINT 12.6-7.8
```

`*` → “times” operator

Return the multiplication of two numerical expressions. Examples:

```
A=12*3
PRINT 12.6*7.8
```

`/` → “division” operator

Return the division between two numerical expressions, the dividend and the divisor. Examples:

```
A=12/3
PRINT 12.6/7.8
```

The value of the expression representing the divisor can not be 0 otherwise a `DIVISION BY ZERO ERROR` will be raised:

```
PRINT 12/0
? Division by Zero Error
```

`%` → “modulo” operator

Return the modulus of a division, namely the remainder of an integer division. Let’s say that we have to compute 12 modulo 5. The result is 2:

```
12 % 5 = 2
```

because 12/5 has a quotient of 2 and a remainder of 2. In fact, 12-(5*2)→ 12-10=2

```
12 % 6 = 0
```

because 12/6 has a quotient of 2 and a remainder of 0. In fact, 12-(6*2) → 12-12=0

The two expressions must be integer values: if they are floating point numbers, they will be truncated and only their integer parts will be considered:

```
12.6 % 4.3 = 0
```

because `INT(12.6)=12` and `INT(4.3)=4` so the previous operation becomes

```
12 % 4 = 0
```

`#` → “integer division” operand

Return the integer part of the division of two numerical expressions. This is equivalent to call the `INT()` function after a regular division:

```
12.6 / 2.7 = 4.66667
12.6 # 2.7 = 4
INT (12.6 / 2.7) = 4
```

^ → "exponent" operand

The exponentiation is a repeated multiplication of the base times the exponent:

```
3^3 = 27
```

that is equivalent to 3*3*3. Note: any number to the 0 power is 1; 0 to any power is 0; 0 to a negative power raises a "Division by zero" error.

```
123^0 = 1
0^5 = 0
0^-1 = Division by Zero Error
```

( ) → parentheses

In mathematic expressions the parentheses are used to clarify or change the precedence of operations set by the operands' precedence. Let's see the example below:

```
A = 12 + 3 / 4
```

By looking at the operators' precedence table in chapter 2.11, it's easy to understand that the division will be executed before the addition because it has a greater precedence, so the result will be 12.75. In fact, the formula is interpreted as follow:

```
A = 12 + (3 / 4)
```

Now, let's take a look at this one:

```
A = 12 * 3 / 4
```

In the above formula, the precedence of the operators says us that when two or more operators have the same precedence they are executed in the order they appear in the code line, so the result is 9, because the expression is interpreted as follow:

```
A = (12 * 3 ) / 4 = 9
```

In fact, if we invert the operations:

```
A = 12 / 3 * 4
```

the result changes, and now it is 16, because the formula is seen as follow:

```
A = (12 / 3) * 4 = 16
```

If we want to change the order of the operations, we can use the parentheses:

```
A = 12 / (3 * 4 ) = 1
```

In the example above, we forced the interpreter to executed first the multiplication and then the division.

Parentheses can be nested, for more complex expressions:

```
A = 12 / (3 * 4 + (45 – 7) * (12 + (7 / 5 )) + 3) = 0.022892
```

= → "equal" operator

The equals sign is used to assign a value to a variable:

```
A = 12 + 4
```

A will get the results of the addition, 16.

It's also used as a relational operator to check that 2 expressions have the same value:

```
10 IF A = B THEN PRINT "OK"
```

The program will print `OK` only if A and B have the same value.

<>, <, >, <=, >= → relational operands

These operands complete the family of relational operands used in boolean expressions to branch in different portion of code, depending the values assumed by expressions. Examples:

```
10 IF A <> B THEN 100: REM JUMP IF A AND B DIFFER
20 IF A < B THEN 200: REM JUMP IF A IS LESS THAN B
30 IF A > B THEN 300: REM JUMP IF A IS GREATER THEN B
40 IF A <= B THEN 400: REM JUMP IF A IS LESS THAN OR EQUAL TO B
50 IF A >= B THEN 500: REM JUMP IF A IS GREATER THAN OR EQUAL TO B
```

2.13 Error management

Errors are managed by LM80C BASIC interpreter. In direct mode every time that the user press the "RETURN" key the interpreter parses the statements entered and execute them. If it finds a non valid statement it raises a message error telling the user what's gone wrong, i.e.: if a sum between a numerical and a string variable is asked:

```
PRINT 1+"2"
```

A `?Type Mis-match Error` will be printed out. Since in direct mode the interpreter doesn't store the instructions that are being executed, the only way to rectify the error and get the correct result is to re-enter the instruction:

```
PRINT 1+2
 3
```

If an error occurs while running a program, the interpreter will stop its execution and will print the message that identifies the error with the addition of the line where the error has occurred. For example, if the following program will be run:

```
10 A=0
20 PRONT A
```

The error will also report the number of line:

```
RUN
?Syntax Error in 20
```

To fix the error, the user must re-enter the line by typing it again or by listing it, move the cursor over the error, type the correct syntax of the statement and then press “RETURN”. It may help to use the HELP command that immediately lists the line containing the error:

```
RUN
?Syntax Error in 20
HELP
20 PRONT A
```

If the “RUN STOP” key is pressed while a program is being executed, the interpreter will print a message to show that the halt has been invoked by the user. Let’s consider the following code:

```
10 A=0
20 FOR I=0 TO 100
30 A=A+I
40 NEXT
```

If, after the `RUN` command has been entered, the `RUN STOP` key is pressed the interpreter will return to the prompt (direct mode) and print a message saying in which line the execution has been interrupted:

```
Break in 30
```

To resume the execution, the user can use the `CONT` command: the interpreter will continue from the line following the one where the program was interrupted. However, if a change is made in the program before the `CONT`, or another error is raised in direct mode, the interpreter won’t be able anymore to resume the execution of the program, raising a `Can’t Continue Error`.

If, after the program has been halted, the user enters the `HELP` command, the line of the break will be printed as it was a normal error:

```
HELP
30 A=A+I
```

3. LM80C BASIC INSTRUCTIONS

3.1 In alphabetical order

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

3.2 Per category

3.2.1 Arithmetical operators

-	Subtraction
+	Addition
/	Division
%	Modulo
#	Integer Division

*	Multiplication
^	Power

3.2.2 Call/Return, Jump and Loop

FOR..NEXT
GOSUB
GOTO
RETURN

3.2.3 Timing

TMR
PAUSE

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...ELSE
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
HELP
LIST
REM
RUN
STOP

3.2.7 Graphics statements

CIRCLE
COLOR
DRAW
GPRINT
PAINT
PLOT
POINT
VPEEK
VPOKE
VREG
VSTAT

3.2.8 Display

CLS
LOCATE
POS
PRINT
SCREEN
SPC
TAB
WIDTH

3.2.8 System & Input/Output

INP
OUT
RESET
SERIAL
WAIT

3.2.9 Keyboard

INKEY
INPUT
KEY

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

INSTR
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 and returns a 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 conjunction. 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 arc-tangent of <*x*>. The result is in radius in range -pi/2 to pi/2 (pi is greek pi, i.e. 3.1415927).

Example:

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

BIN$

`Syntax: BIN$(x)`

Converts an expression into a string containing the binary representation 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. Coordinates origin (0,0) is set at the top left corner of the screen. <*radius*> can range from 0 to 255. Points of the circumference that come out of the screen won't be painted. If <*color*> is passed, the circle will be drawn with the color being specified, otherwise the foreground color set with `COLOR` will be used. <*color*> ranges from 0 to 15: colors from 1 to 15 correspond to the system colors while 0 is a special value that instructs the BASIC to draw a circle using the background color set with `COLOR`, i.e. to reset the pixels that are on.

Example:

```
CIRCLE 128,96,25 => draws a circle centered in the middle of the screen with
a diameter of 50 pixels (radius=25) using the predefined foreground color
```

```
CIRCLE 128,96,25,0 => erases the circle previously drawn using the
background color
```

See also: `COLOR` for color codes and foreground/background colors.

CLEAR

`Syntax: CLEAR [xx][,yy]`

The call of `CLEAR` with no arguments erases any variable in memory: this means that it sets the contents of all the numeric variables to 0 and the string variables to "" (empty string). This command is automatically called when a `RUN` or `NEW` command is given.

If called one numeric expression, it sets the string space to the value of argument <`xx`>. At startup, the string space is set by default to 100. See the string chapter at 2.9 for more info.

If the second numeric argument is also present, then CLEAR sets the top of the RAM available to BASIC programs to `YY`.

Both XX and YY must be evaluated as signed integers (in range -32,768 to +32,767).

Examples:

```
CLEAR => clears every variable
?FRE(0)  => check free RAM for BASIC
-21213  => 44343 - this value can change for different FW versions
?FRE("")  => check string space
 100  => default value
CLEAR 200 => sets the string space to 200 chars
?FRE("")  => check string space
 200
?FRE(0)
-21313  => 44223 - free RAM reduced by 100 bytes
CLEAR 300,-20000  => -20000 = 45536
?FRE("")
 300
?FRE(0)
 24124  => free RAM reduced by ca. 20,000 bytes
```

CLS

`Syntax: CLS`

Clears the current screen, initializing the pattern cells to default values. The filling value is graphic mode dependent: 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 (byte 0). 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 behavior 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 from 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 image screen).

Values range from 1 to 15, as follow:

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

Pay attention that:

- in screen 0 (text mode), only the first 2 arguments are allowed since the background and border colors coincide;
- in screen 1 & 4 (text/graphics modes) the 3 arguments must be passed;
- in screen 2 (bitmap graphics mode) the 3 arguments must be passed. Remember that in screen mode 2 the color of the pixels painted by `PLOT`, `DRAW`, and `CIRCLE` commands, otherwise not specified, is the foreground color set by `COLOR`;
- in screen 3 (multicolor mode) only one argument is accepted and refers to the border 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
```

`PLOT`, `DRAW`, and `CIRCLE` statements can use “0” as a color value. This doesn’t correspond to any of the system colors and it’s just a way to instruct the BASIC interpreter to draw pixels using the background color, i.e. reset the pixel that are on. “0” can not be used as an argument for `COLOR`, raising a `SYNTAX ERROR`.

It is important to know how the VDP works in bitmap graphics mode. For every group of 8 horizontal pixels, corresponding to 8 bits in memory (or 1 byte), only 2 colors are allowed: the foreground and background colors. The former is used to color the pixels that are set (i.e. that are ON, corresponding to bits of value “1”) while the latter is uded to color the pixels that are reset (i.e. that are OFF, corresponding to bits of value “0”). If you try to draw a pixel with another color, i.e. you try to redefine the foreground color, this change will alter the color of all the pixels that are set in such byte. This means that if you draw a line in a part of the screen with a color and then you intersects such line with another one with a different color, the color of the last one will alter the colors of the pixels of the first one.

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. If, after in direct mode, an error is raised, then the continuation of the main program is not possible anymore.

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 contains any space, their use is needed.

Example:

```
10 DATA 20,30,40  => numericals
20 DATA " HELLO ",LM80C
```

Note at line 20 the usage of quotation marks to include the leading and trailing spaces in the first string.

See also READ and RESTORE.

DEEK

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

Reads a word from memory cells given by the expression <*nnnn*> and <*nnnn*>+1. <*nnnn*> must be a numerical expression whose value must be into the range 0 to 65,535. A word is a 16-bit value stored in 2 contiguous 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
```

See also `DOKE`.

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 being passed that will be used by the function (it must be a valid variable name, see chapter on variables at 2.8); <*expression*> is the newly defined function. Limits: everything must reside in 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 maximum 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 from 0 to 5 (36 elements).
```

See also [array chapter at 2.10](#).

DOKE

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

Writes a word value (a 16-bit number) into a couple of contiguous 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. <*nnnn*> and <*val*> must be numerical expressions whose values must be into the range 0 to 65,535.

Example:

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

See also `DEEK`.

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, 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. Origin of the coordinates (0,0) is set to the top-left corner while 255,191 is the bottom-right corner. Due to hardware limitations of the Video Display Processor (VDP), for every single byte of the memory screen (8x1 pixels) there can only be 1 primary color: if a line of one color intersects a line of another color, the pixels drawn into 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. <*color*> ranges from 0 to 15: colors from 1 to 15 correspond to the system colors while 0 is a special value that instructs the BASIC to draw a line using the background color set with `COLOR`, i.e. to reset the pixels that are on.

Example:

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

See also: `COLOR`, for color codes and foreground/background colors.

END

```
Syntax: END
```

Terminates the execution of a program. Any statement and/or program line 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. Example:

```
10 A=1
20 PRINT A
30 END:A=A+1:REM THIS ADDITION WILL NEVER BE DONE
40 PRINT A:REM THIS LINE WILL NEVER BE EXECUTED, TOO
```

See also: STOP.

EXP

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

Returns the mathematical constant *e* (the Euler's number) 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 variable name that will contain the value incremented during the loop and used for the ending test. <*start*>, <*stop*>, and <*step*> are numerical expressions. <*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 to see 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, although 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). Another example:

```
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. The following is a decrement loop:

```
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 subtraction and 0 is less than 1, that is 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 area.

Examples:

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

See also `CLEAR`.

GOSUB

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

Calls a sub-routine by making a jump at line <*line*>. A sub-routine is a sequence of statements that can be called from different points of the program. `GOSUB` stores the current point of the program execution, then jumps to execute the portion of code from line <*line*>. The sub-routine ends when the `RETURN` statement is encountered, after which the BASIC interpreter will resume the execution at the instruction following the calling `GOSUB` statement.

Example:

```
10 A=0:GOSUB 100:PRINT A:REM A is 1
20 GOSUB 100:PRINT A:REM NOW, A is 2
30 END
100 A=A+1
110 RETURN:REM return to caller
```

See also `GOTO`, `ON`, and `RETURN`.

GOTO

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

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

Example:

```
10 GOTO 100
20 PRINT "HELLO"
100 PRINT "WORLD"
```

If you run this program you will get only a “`WORLD`” message.

See also `ON`.

GPRINT

```
Syntax: GPRINT text,X,Y[[,fgcol][,bgcol]]
```

Prints a text on the screen when in graphic 2 mode. <*text*> must be an expression that can be evaluated as a string, <*X*> is the character column in range 0~31 and <*Y*> is the character row in range 0~23. Characters in screen 2 keep a cell whose size is 8x8 pixels. <*fgcol*> is the foreground color: if it’s not passed, the default foreground color or the one set by `COLOR` will be used.

Similarly, <*bgcol*> is the background color and follows the same considerations made for <fgcol>. These arguments are optional and can be omitted. If only one of them is present, it will be assumed as the foreground color. If GPRINT is used outside of the screen 2 mode, a "`No Graphics Mode`" error will be raised.

Examples:

```
SCREEN 2:GPRINT "HELLO",0,0
```

This line will switch to graphic 2 and print "HELLO" in the top left corner of the screen using black text on a white background (default colors of screen 2).

```
SCREEN 2:GPRINT "HELLO",10,10,4,5
```

Same as before, but the text will printed at coordinates 10,10 (in pixels, at 80,80) using a blu text on a light blue background.

```
SCREEN 2:GPRINT "HELLO",30,0,8
```

This time, the text will be printed using a red color (background is white, since it wasn't passed as argument so the default background color has been used). Note that as the text is longer than the place left on the selected row before to reach the end of the line, the printing will continue on the next row. If the bottom right corner is reached while printing, chars that should still be printed will be discarded:

```
SCREEN 2:GPRINT "HELLO",30,23
```

Only "`HE`" will be printed on the screen.

```
SCREEN 2:GPRINT 12,0,0
?Type Mis-match error
```

We have tried to print a non string expression (the number 12) and the interpreter has answered with an error message. To print a number, it must be converted into a string before:

```
SCREEN 2:GPRINT STR$(12),0,0
```

Keep in mind that `STR$` returns a string whose first char is the sign char and, if the number is positive then an empty char will be printed in front of the number, as it occurs when used in text modes.

See also: `SCREEN`, `COLOR`.

HELP

```
Syntax: HELP
```

Prints the line where an error has occurred. This statement can only be used in direct mode after an error has stopped the execution of a program. If called inside a program or when no errors have occurred an “HELP Call Error” will be raised.

Example:

```
10 A=1
20 PRONT A
RUN
?Syntax Error in 20
HELP
20 PRONT A
```

See also: `CONT`.

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 whose value is in range -32,768/+32,767.

Example:

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

IF...THEN...ELSE

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

IF is a conditional branch. It is used to change the order of the execution of the instructions of a program instead of the ordinary sequential flow. <*cond*> can be any valid arithmetic, relational, or logical expression: if it is evaluated true (i.e. non-zero), the statements after `IF` are executed; if it is evaluated false (i.e. zero), then if checks if the statement `ELSE` is present and, if yes, then it executes it.

There are several allowed formats of the statement. If the expression is true, anything following `THEN` will be executed, to the end of the line or up to the `ELSE` statement, if present

```
10 A=0
20 IF A=1 THEN PRINT "A IS EQUAL TO 1":GOTO 100
```

```
30 PRINT "A IS NOT EQUAL":END
```

Prints:

```
A IS NOT EQUAL
```

With the `ELSE` the behavior can change:

```
10 A=1
20 IF A=1 THEN PRINT "A = 1":ELSE PRINT"A <> 0"
```

Prints:

```
A = 1
```

and the rest of the statements after `ELSE` are ignored. If, otherwise, A=0 then the above code prints the following:

```
A <> 0
```

because the condition is evaluated as `FALSE` so the statements between `THEN` and `ELSE` are ignored, and only the code after `ELSE` is executed.

If there is only a `GOTO` instruction after the expression being evaluated, the `THEN` statement can be omitted:

```
10 IF A=1 GOTO 100
20 IF A=2 GOTO 200:ELSE GOTO 300
```

If there is only a jump instruction (`GOTO`) after the `IF`, like in the previous example, another form can be used where `GOTO` is omitted and `THEN` is followed by the line number where the jump must be executed to. Same for `ELSE`, that can be followed by just the line number if the only instruction after the statement `ELSE` is a jump:

```
10 IF A=1 THEN 100
20 IF A=1 THEN 100:ELSE 200
```

Please note that the `ELSE` statement MUST be preceded by a colon (“:”), otherwise a syntax error may be raised by the interpreter.

Another important point to remember is that ELSE used outside the IF...ELSE statement will be ignored and considered as a REM, so that the interpreter will ignore every following instruction:

```
10 ELSE PRINT “HELLO”
```

will do nothing.

INKEY

```
Syntax: INKEY(nnnn)
```

Returns the ASCII code of the key pressed by the user. <*nnnn*> is a numerical expression whose value can vary between 10 and 1,023 and represents the interval the function has to wait for the user's input before to resume the execution of the program: values smaller than 10 will be automatically converted to 10, while values greater than 1,023 will raise an error. If <*nnnn*> is 0 the function won't wait any time: it will read the input buffer and return the code being present. If <nnnn> is greater than zero, then `INKEY` will wait for a key pressure for the number of hundredths of seconds corresponding to the value of the argument. If a key will be pressed during this time, then the wait will be endend and the code corresponding to the pressed key willl be returned: if, otherwise, the user won't press any key, at the end of the waiting time a value of 0 will be returned (meaning 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 <u>after</u> a certain moment, it is recommended 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 key. Since the function reads 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):IF A=0 THEN 10:REM REPEAT UNTIL A KEY IS PRESSED
30 PRINT CHR$(A):REM PRINT PRESSED KEY CODE
```

```
10 INKEY(0):REM EMPTIES THE BUFFER
20 A=INKEY(500):REM WAIT A KEY FOR 5 SECONDS
30 IF A=0 THEN PRINT "NO KEY PRESSED":GOTO 50
40 PRINT "KEY PRESSED: ";CHR$(A)
50 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, they must be separated each other by a comma, and the user must be enter the exact amount of values with the correct type. If the data is invalid, i.e. a string when a number was expected, the interpreter will print a "`Redo From Start?`" message 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:

```
10 INPUT "WHAT'S YOUR NAME";NAME$ : REM  wait for a string
20 INPUT "INPUT NAME,AGE";NAME$,AGE : REM wait for a string and a number
30 INPUT A : REM wait for a number with just a "?"
```

INSTR

```
Syntax: INSTR(string1,string2)
```

Returns the position of the first occurrence of one string within another. <s*tring1*> is the string being searched, while <*string2*> is the string sought. It returns 0 is <*string2*> wasn’t found or it’s longer than <*string1*>, otherwise returns the position of the first occurrence found. Comparison is done using ASCII codes of the chars for both strings (i.e., “a” and “A” are different).

Examples:

```
?INSTR("abcd","bc")
2

?INSTR("abcdcd","bc")
2                       => first occurrence

?INSTR("a","abc")
0                       => string1 is shorter

?INSTR("hello","lo")
4

?INSTR("hello","LO")
0                       => "lo" and "LO" are different
```

```
?INSTR("123",1)
?Type Mis-match Error  => types are different
```

INT

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

Returns the integer part of <*X*>. <*X*> must be a valid numerical expression. The result is obtained 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
```

KEY

```
Syntax: KEY [n[,”text”]]
        KEY 9,del,rep
```

The LM80C Color Computer provides 8 function keys that can be programmed to quickly print short commands. With no arguments, `KEY` just print the text assigned to each text. At startup, the system assigns the following functions to each key:

Key 1: "LIST" + CHR$(13)
Key 2: "RUN" + CHR$(13)
Key 3: "SCREEN1" + CHR$(13)
Key 4: "COLOR1,15,5" + CHR$(13)
Key 5: "SERIAL1,38400" + CHR$(13)
Key 6: "SCREEN2" + CHR$(13)
Key 7: "CONT" + CHR$(13)
Key 8: "HELP" + CHR$(13)

To change a function assigned to any key, `KEY` must be followed by <*n*>, a number between 1 and 8 that corresponds to the function key whose text is to be changed, and a string that will be the new function assigned to the key. Please consider that not all the chars are allowed. You can only use chars whose ASCII codes are in range 32 ~ 122, that include space, quotation marks, punctuations, numbers and several other ones like question mark and parentheses, and letters in upper (‘A’ ~ ‘Z’) and lower case (‘a’ ~ ‘z’), although letters in lower case will be converted into upper case. The only special char that is allowed is RETURN, that you can enter by using the BASIC function `CHR$()`.

Consider also that the max. number of chars allowed is 16 for each function key. Texts longer than this value will be truncated.

Examples:

```
KEY
KEY 1:"LIST"+CHR$(13)
KEY 2:"RUN"+CHR$(13)
KEY 3:"SCREEN1"+CHR$(13)
KEY 4:"COLOR1,15,5"+CHR$(13)
KEY 5:"SERIAL1,38400"+CHR$(13)
KEY 6:"SCREEN2"+CHR$(13)
KEY 7:"CONT"+CHR$(13)
KEY 8:"HELP"+CHR$(13)
Ok
KEY 1,"RUN100"+CHR$(100) → assign RUN100 followed by a RETURN char.
```

By using 9 as argument for key number, the system lets the user to change the speed of the auto-repeat feature. The command will look for 2 additional arguments: the first value <*del*> represents the delay (in hundredths of a second) to wait before to activate the auto-repeat, while <*rep*> represents the delay (in hundredths of a second) between two following prints of the key. Default values are 64 and 8, respectively. Be careful: the usage of short delay can lead to an unusable keyboard, with the reset as the only way to revert the things.

If “0” is used as unique argument, then the command will restore the function keys and auto-repeat to their default behaviours:

```
KEY 0 → revert to original behaviors
```

LEFT$

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

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

Example:

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

See also `RIGHT$` and `MID$`.

LEN

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

Returns the length of string <*str*>. Any type of char is considered, so a space is just a char being counted.

Example:

```
?LEN("HELLO") => 5
?LEN(" HELLO") => 6
```

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
```

LIST

```
Syntax: LIST [<start>][-[<end>]]
```

Lists a program stored in memory. `LIST` without arguments will print the whole program. If only <*start*> is passed, only the specified line will be printed: if the line doesn't exist, none will be printed. <*start*> followed by "-" will print the line passed as argument and all the following ones. If <*end*> is passed, preceded by "-", only the lines up to <*end*> will be printed. If both <*start*> and <*end*> are passed, only the lines between, and included them, will be printed. If <*start*> isn't present in memory, the next line will be considered instead; if <*end*> doesn't exist, the line before will be considered. This means that, i.e., if the program contains lines 10 and 20, then "`LIST 15-`" will just print line 20. During the listing, pressing the `SPACE` key will pause the printing: to resume it, just press the `SPACE` key another time, or, to halt it, press the `RUN/STOP` key. Listing can also be halted by pressing the `RUN/STOP` key at any time.

Examples:

```
LIST          => list the whole program stored into memory
LIST 100      => list only line 100
LIST -100     => list lines from beginning of program to 100
LIST 100-     => list lines from 100 to end of program
LIST 50-100   => list lines from 50 to 100 only
```

LOCATE

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

Places the cursor on the screen at coordinates <*x*>,<*y*>. `LOCATE` works only in screen modes 0, 1, & 4 because screen modes 2 & 3 are pure graphics modes and have no cursor support. <*y*> is in range 0~23, while <*x*> is in range 0~31/39, depending on which mode is active at the moment: screen mode 1 is 32 chars wide while screen modes 1&4 are 40 chars wide. Coordinates 0,0 point to the top left corner.

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 a numerical expression whose value is greater than 0.

Example:

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

MID$

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

Returns a portion of string <*str*>. If <*end*> is omitted, it returns the chars from <*start*> to the end of the string; if <*end*> is passed, it returns the chars between, and included, <*arg*> and <*end*>.

Examples:

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

See also: `LEFT$` and `RIGHT$`.

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
```

See also: `CLEAR`.

NEXT

`Syntax: NEXT <list of variables>`

Used in `FOR` loops. See `FOR` for details.

NMI

`Syntax: NMI addr`

Used to activate the Non-Maskable Interrupt (NMI) hooked to the VDP interrupt signal. <*addr*> is a signed 16-bit integer that points to a location in memory where an user routine is located. If <*addr*> assumes the value 0, then the statement disables the VDP interrupt signal and reset the vector to its defaults.

Example:

```
NMI &H9000
```

This one sets the NMI vector to point to an interrupt service routine located at &H9000.

NOT

`Syntax: NOT arg`

Logic operator used in boolean expressions. `NOT` performs a logical negation, or bit-wise 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 right. `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 correct result is just -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
```

and

```
-(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 whose values must be greater than or equal to 1: the jump is executed by calling the line whose position corresponds to the value of <*expr*>. The lines must be separated by colons.

Examples:

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

See also: `GOTO`, and `GOSUB`.

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

Example:

```
OUT 1,100
```

Note: with no understanding of what is being done, writing into I/O ports can lead to unpredictable behaviors, even the complete freeze of the system resulting in the loss of any data.

PAINT

```
Syntax: PAINT x,y[,col]
```

Fills connected areas of the screen. This command only works in screen mode 2. It starts at the point whose coordinates are <*x*> and <*y*>, where <*x*> can range between 0 and 255, and <*y*> can range between 0 and 191. If the pixel at <*x*> and <*y*> is already set (i.e. it has been drawn with a color other than the background color) then the command exits immediately. Filling is made by horizontal scanlines, and halts when a colored pixel is encountered on the row or when the border of the screen is reached. <*col*> is the color used to fill the area, and can range between 1 and 15. If, instead, <*col*> is not present, then the current foreground color will be used.

Examples:

```
SCREEN 2:CIRCLE 127,95,40:PAINT 127,95
```

See also: `COLOR`.

PAUSE

```
Syntax: PAUSE arg
```

Forces the BASIC interpreter 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. The delay can be interrupted by pressing the "`RUN STOP`" key.

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 anything else that 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.
```

See also `POKE`.

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 at the top left corner of the screen. <*color*> ranges from 0 to 15: colors from 1 to 15 correspond to the system colors while 0 is a special value that instructs the BASIC to draw a circle using the background color set with `COLOR`, i.e. to reset the pixels that are on.

Example:

```
PLOT 0,255   => plot a pixel into the top right corner of the screen using 
the foreground color
PLOT 0,255,0 => reset the pixel previously set using the background color
```

See also `COLOR` for color codes and for foreground/background colors.

POINT

```
Syntax: POINT(x,y)
```

Return the color of a screen pixel in graphic mode 2. If the pixel whose coordinates are given by <*x*> (0~255) and <*y*> (0~191) is set, `POINT` will return the corresponding color, while if the pixel is reset it will return 0. Pay attention that 0 does NOT mean that the pixel is without colors: the TMS9918A uses two different colors for each single 8-bit cell of the video where the foreground color is used for pixels that are set and background color is used for pixels that aren’t set. This means that foreground and background colors can be the same but `POINT` returns a number different than 0 only if the pixel is set with a graphics statement.

Example:

```
10 SCREEN 2:COLOR6,6,6:REM THE SCREEN IS RED
20 PLOT 100,100:REM SET THE PIXEL AT 100,100 BUT IT’S INVISIBLE (RED ON RED)
30 A=POINT(100,100):REM A=6 BECAUSE FOREGROUND COLOR IS 6
40 PLOT 100,101,1
50 A=POINT(100,101):REM NOW, A=1 BECAUSE WE USED A DIFFERENT COLOR
60 A=POINT(99,100):REM NOW, A=0 BECAUSE THIS PIXEL HASN’T BEEN SET YET
```

See also: `COLOR`.

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
```

See also PEEK.

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 internal buffer line is a special temporary memory used by the interpreter to store the text found on a screen line when the `RETURN` key is pressed: the 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 behavior 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 10 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

PRINT A => prints the value of variable A

PRINT "LENGTH:";LN;"METERS" => prints LENGTH 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 printing 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.

```
LOCATE 10,10:PRINT"HELLO"
```

Two special statements can be used in conjunction with `PRINT`: `SPC` and `TAB`. The former prints a certain value of empty spaces:

```
PRINT "HELLO";SPC(5);"WORLD"
HELLO     WORLD
```

TAB moves the cursor to the column whose number corresponds to the value being passed:

```
PRINT "HI";TAB(10);"WORLD"
HI        WORLD
```

See also: LOCATE, SPC, TAB.

READ

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

Reads the informations stored into the program with a `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 first unread data. Types of variables and informations must be coherent (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: `DATA`, and `RESTORE`.

REM

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

Used to enter a comment into the program. Everything following the `REM` statement will be ignored up to 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 the first data in 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
```

See also: GOSUB.

RIGHT$

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

Returns a string that contains <*val*> characters to the right of 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. text mode only: 40x24 chars, no sprites support;
1. text/graphic mode: 32x24 chars or 256x192 pixels with tiled graphic mode and sprites support;
2. graphic mode only: 256x192 pixels bitmapped graphic mode, with sprites support;
3. graphic mode only: 64x48 pixels multicolor graphic mode, with sprites supported;
4. text/graphic mode: mix between mode 1 & mode 2, limited sprite 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 complete 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 officially supported yet well 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
```

In an error occurs inside a program or at the prompt in direct mode while in graphics modes (screen 2 & 3), the interpreter will immediately return to screen 1 mode. Same behavior if you press the “`RUN STOP`” key in direct mode while in graphic modes.

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 that sprites magnification halves their video resolution so that when a sprite is magnified each pixel occupies a 2x2 pixel block 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 relevant only when used in graphic modes that support the sprite visualization: for this reason, when setting the screen mode 0 (a text mode) only <*mode*> is accepted.

Further reading: to better know the video capabilities of the VDP the reading of its reference guide is recommended. 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) will be configured as a block device (actually, it is NOT supported by BASIC). <*bps*> indicates the bauds per second, i.e. the speed of the serial line. <*bps*> can assume one of the following values:

```
57600,38400,28800,19200,14400,9600,4800,3600,2400,1200,600,1,0
```

1 and 0 are special values that will be analyzed below. If a valid value for <*bps*> is entered, 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
```

If nothing follows after <*bps*>, then default values of 8,0,1 will be assigned to <*data*>,<*par*>, and <*stop*>, respectively. So, the two commands below are equivalent:

```
SERIAL 1,38400,8,0,1
SERIAL 1,38400
```

If the user tries to re-open a serial communication on a port already opened, an error will be raised:

```
SERIAL 1,19200,8,0,1
Ok
SERIAL 1,38400,8,0,1
Serial Port Already Open Error
Ok
```

If 0 is entered as <*bps*>, then other arguments are ignored and the command closes the connection opened on channel <*ch*>:

```
SERIAL 1,0 => closes the serial port 1, resetting the SIO channel A.
```

The Z80 SIO peripheral chip has an internal buffer that can stores up to 3 chars: if fourth char is received while the CPU hasn't collected one of the incoming chars yet, then a buffer overrun occurs. The BASIC interpreter will disable the serial line functionalities (no incoming chars will be accepted, no outgoing chars will be sent, while the line will still remain open) and the corresponding status LED will be turned on. This condition will remain until the user will call the SERIAL command with the special value of 1 for bps, that re-activates the normal port operation.

```
SERIAL 1,1  => re-activates the RX/TX transmission after a buffer overrun 
condition
```

The LM80C BASIC has also an input buffer whose length is 88 chars: if, during a serial communication, a line longer than 88 chars is received the interpreter will discard all the exceeding chars.

Further reading: to better know the serial capabilities of the SIO chip, the reading of the Z80 peripherals' user manual is recommended. 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 instructions may 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 tone or a noise with a particular frequency for a specific duration on the selected channel. Its behavior varies according to the king of sound to reproduce. The volume of the tone or noise being generated is set with the `VOLUME` command: for this reason, it must be invoked before any call of `SOUND`, otherwise nothing will be heared.

Tone generation

<*ch*> must be in range 1 to 3 for sounds 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. Below is the formula to get the frequency:

```
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 / (4,096-4,000) → 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 <*tone*> to pass to `SOUND` to generate a tone of frequency <*freq*> is:

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

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

```
tone = 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 truncating, the frequency is a little bit smaller than needed

Example:

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

Noise generation

If <*ch*> is in range 4 to 6, then it is intended that the PSG must activate a noise reproduction on channel whose number is given by "*ch-3*": so, 4 stands for channel 1, 5 stands for channel 2, and 6 stands for channel 3. <*tone*> sets the frequency of the noise being generated and varies from 1 to 31, where 1 is the highest frequency and 31 is the lowest one. To get the frequency refers to the following formula:

```
Fnoise = 1,843,200 / 16 / tone
```

So, i.e., if <*tone*> is 20, then *Fnoise* will be equals to 1,843,200/16/20 = 5,760 Hz. If <*tone*> is set to 0, then the noise reproduction on the selected channel will be halted.

A tone and a noise can co-exist on the same channel: it should, however, be considered that they will be mixed together, affecting each other at the human's hear. The example below will generate an old airplane-like noise.

```
SOUND 1,1000,0:SOUND 4,10
```

When `SOUND` is called to generate a noise, the <*dur*> argument is ignored: actually, there is no automatic mechanism to reproduce a noise just for a specified period.

Sound 0

There is a special usage of `SOUND`. If being 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`. It will also set the volume of all the channels to 0.

Example:

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

See also: `VOLUME`.

SPC

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

Prints <*val*> empty chars on video. It can only be used in conjunction with a `PRINT` statement. Note that `TAB` and `SPC` are similar but have different behaviors 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
```

```
PRINT "HELLO";SPC(3);"WORLD"
HELLO   WORLD
```

See also: [PRINT] and [TAB].

SQR

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

Returns the square root of <*arg*>. <*arg*> must be a numerical expression whose values is 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 range 0-255. Some words must be spent about the register numbering. Most of the PSG data sheets found anywhere on the net report that registers are numbered from 000 to 007 and from 010 to 017, without mentioning that the “octal” format is used to indicate the register numbers, leading in mis-understanding with the PSG that seems to operate with strange behaviors: this is only due to wrong register indexes being used. Simply, the registers are 16 and they are numbered from 0 to 15 in decimal format.

Example:

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

Noe that registers #14 and #15 are used by the system manage the keyboard so they shouldn't be used.

Further reading: to better know the capabilities of the Programmable Sound Generator (PSG) chip of the LM80C Color Computer, the reading of the GI AY-3-8910/Yamaha YM219F’s user manual is recommended. A copy can be found here:

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

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.

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 as the interpreter would have reached the last line of the program.

See also: `CONT` and `END`.

STR$

`Syntax: STR$(arg)`

Returns the string representation of the numerical expression <*arg*>.

Example:

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

SYS

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

Calls a machine language routine starting at <*address*>. <*address*> must be a valid numerical expression in range 0~65,535, or an absolute signed integer number (i.e. in range -32,768 to +32,767). If <*value*> is passed, it will then be passed to the routine into the Accumulator register. <*value*> must be a byte value (0~255). SYS differs from USR in two ways: first, it's a command, so there is no return value to be collected, and second, USR has an indirect indexing method while with SYS the address is explicitly passed.

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 movement.

Example:

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

Note that TAB and SPC are similar but have different behaviors 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. Another difference is that the latter moves the cursor while keeping in account its previous position while TAB just considers the video columns, regardless of the current position of the cursor.

Other examples:

```
PRINT "HELLO";TAB(10);"WORLD"
HELLO     WORLD
```

```
PRINT "HI";TAB(10);"WORLD"
HI        WORLD
```

See how the word "WORLD" is being printed at the same location (corresponding to column number 10) regardless the length of the previous word.

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 only manages 16-bit values, the system timer is divided into 2 "halves", so it can be read as a couple of 16-bit registers: 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 -32,768 to +32,767. To get the unsigned counterpart, if the value returned is negative, you can add it to 65,536 to get a value in the range 0 ~ 65,535.

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 number 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 a 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 $8065 and $8066: the address must be set using the little-endian order, meaning that the less significant byte must be stored into $8065 while the most significant byte must be stored into $8066.

Example: suppose that a subroutine has been stored in RAM from $B0A0. Then the byte $A0 will need to be stored into $8065 while byte $B0 into $8066. This can be done using the `DOKE` command. Then, since `USR` is a function, it must be called in a way by collecting the possible returned value:

```
10 DOKE&H8065,&HB0A0:A=USR(0)
```

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 results will be 0.

Examples:

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

VAL can also be used with strings representing hexadecimal or binary values. If the trailing char is “$” the string will be interpreted as an hexadecimal value while if the trailing char is “%” then it will be interpreted as a binary value.

Examples:

```
?VAL("$0AFF")
```

```
2815
```

```
?VAL("%1100")
12
```

Pay attention that VAL only supports uppercase letters when interpreting hexadecimal values.

Examples:

```
?VAL("$a")
?HEX Format Error
```

```
?VAL("$A")
10
```

```
?VAL("$0a")
0
```

```
?VAL("$0A")
10
```

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, respectively. If <*chn*> is 0, the command will modify all the channels. <*val*> must be a numerical value between 0 (no audio) and 15 (max volume). Another thing to take into account is that the volume is logarithmic: the increments in the "power" of the sound is more evident as long as the value of the volume increases.

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 that sound generation on such channel has been halted. To stop the sound generation, too, `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
```

See also `SOUND`.

VPEEK

`Syntax: VPEEK(nnnn)`

Reads a value from the VRAM (or Video-RAM). <*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 at exclusive use of the VDP (Video Dislay Processor), it can only be accessed by this chip. This means that 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` writes 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
```

See also [VPEEK](#).

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 recommended. A copy can be found here:

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

See also `VSTAT`.

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), since 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)
```

See also `VREG`.

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, too, 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.
```

See also `INP`.

WIDTH

```
Syntax: WIDTH val
```

Sets the length 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 length of the program lines being but only the length 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.APPENDIX

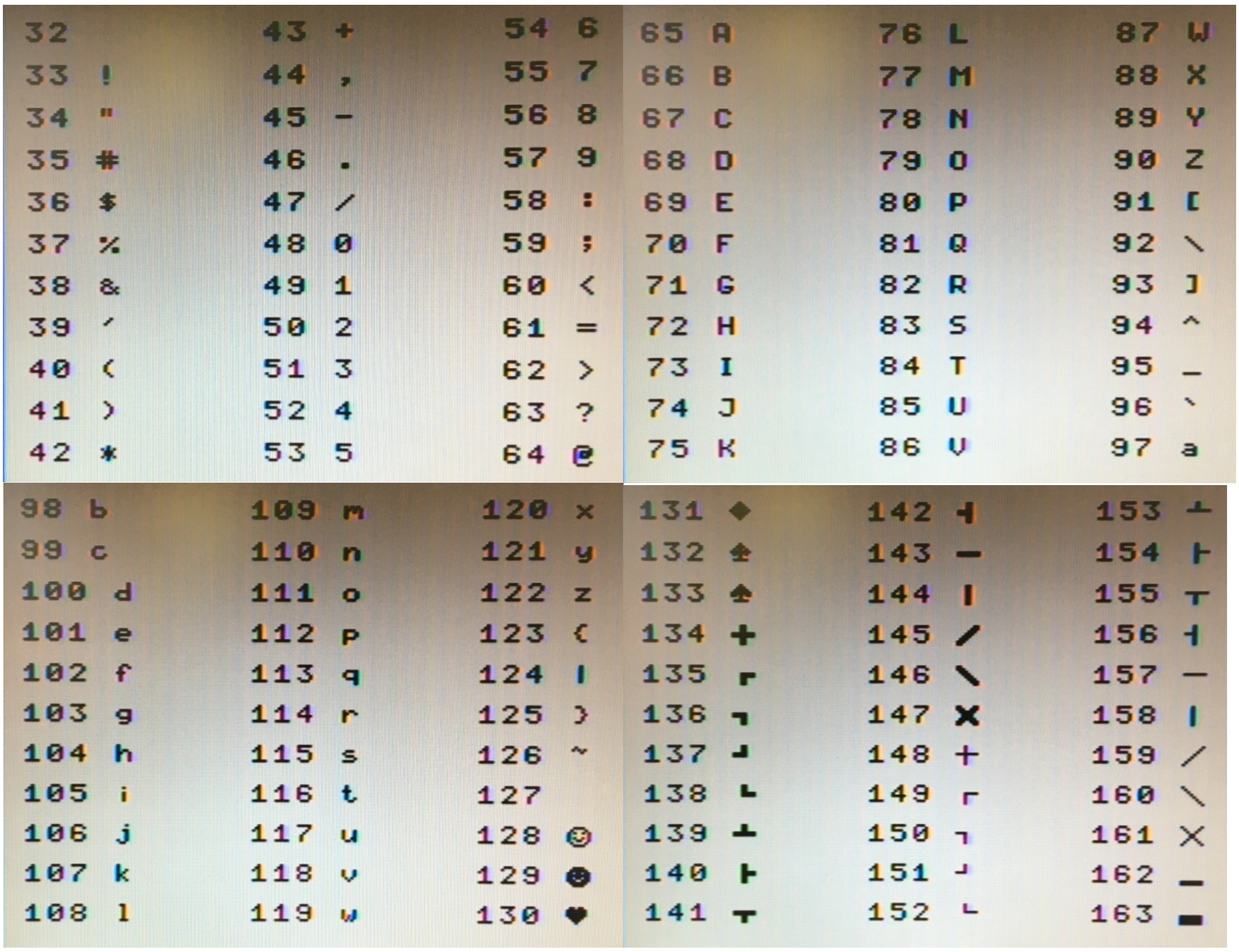
5.1 ASCII table

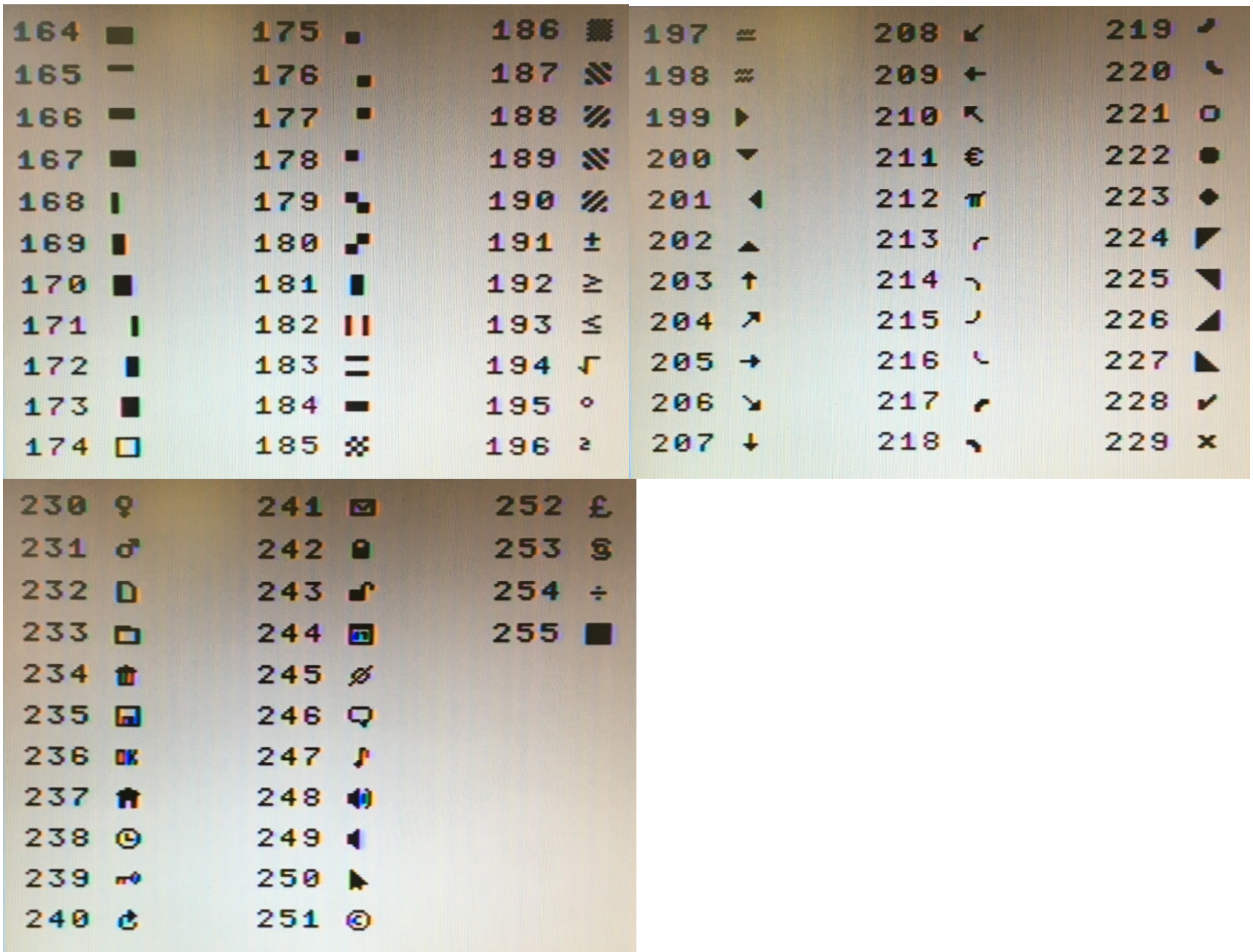
ASCII codes from 0 to 31 (non-graphic chars):

0: NULL	8: DEL key (backspace)	16: C= graph. key	24: F8 (HELP) key
1: F1 key	9: H. tab (on serial)*	17: CTRL-Q*	25: HOME key
2: F2 key	10: Line feed	18: CTRL-R*	26: INSERT key
3: CTRL-C	11: *not used*	19: CTRL-S*	27: ESCAPE key
4: F3 key	12: Clear screen & form feed (on serial)	20: SHIFT key	28: CURSOR LEFT key
5: F4 key	13: RETURN key & carriage return	21: CTRL-U*	29: CURSOR RIGHT key
6: F5 key	14: CTRL key	22: F6 key	30: CURSOR UP key
7: bell (on serial)*	15: CTRL-O*	23: F7 key	31: CURSOR DOWN key

*legacy

ASCII codes from 32 to 255 (graphic chars):





P.S.: char 127 is DELETE – char 255 is cursor char.

P.P.S.: the ASCII table above is valid both for SCREEN 0 (6x8 pixel chars) and SCREEN 1, 2 (with GPRINT), and 4 (8x8 pixel chars).

5.2 Status LEDs

Status LEDs are used by the operating system to communicate special conditions to the users. Their meaning is as follow:

0: ROM/RAM bank #0 switcher
1: VRAM switcher
2: N.C.
3: N.C.
4: Serial 1 buffer overrun
5: Serial 2 buffer overrun
7: Serial 1 line open
8: Serial 2 line open

6. 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

Leonardo Miliani