# Stefan's IoT BASIC in a nutshell

## Version information -

This manual is for the released version 1.4 and 2.0 of Stefan's IoT BASIC interpreter.

## Supported hardware

BASIC runs on Mac, Windows, Raspberry PI, MSDOS and many Arduino plattforms. I test regularly on 8 and 32 bit AVR, ESP8266. ESP2, RP2040 but also on some less common platforms like MKR, XMC and SMT. Have a look in the hardware section for more information. Many peripherals are supported. You find details in the hardware section of the manual.

## BASIC language sets

This BASIC interpreter is structured in language sets. They can be compiled into the code separately. With every language set there are more features and command. This makes it adaptable to the purpose. The manual is structured according to the language sets.

The intepreter has two data types - numbers and strings. The number type can be set at compile time. It is either an integer or foating point. Depending on the definition of number\_t in the code the interpreter can use everything from 16 bit integers to 64 bit floats as basic number type. Default is single precision floats for all real OSes and integer on all Arduinos. In practice all but the UNO boards can run single precision floats.

Arrays and string variables are part of the Apple 1 language sets. Strings are static. They reserve the entire length of the string on the heap. Depending on the definition of the string index type in the code, strings can be either 255 characters or 65535 characters maximum length.

Array can be maximum two dimensional and string arrays can be one dimensional.

Variable names are two letters or one letter and one digit in the smaller language sets. Compiled with HASLONGNAMES, the interpreter can have names up to MAXNAME length. This is 16 by default. The HASLONGNAMES setting is default now.

Keywords and variable names are not case sensitive. They are printed as uppercase with the LIST command. Strings and string constants are case sensitive.

There is a set of examples program in examples/00tutorial. They are referred to here as "the tutorial".

As a proof of concept many of the 101 BASIC computer games in David Ahl's original book from 1975 have been ported. Look into examples/14games for some 70s gaming fun.

Most Arduino built-in examples from https://docs.arduino.cc/built-in-examples/ have been ported to BASIC. They are in examples in the folders 01-10.

## Language Personalities in Version 2.0

BASIC has a lot of Apple 1 and Palo Alto DNA. Some of the language constructs and conventions are taken from these languages and not from the more common Microsoft type BASICs. The subtle differences of BASIC dialects made programs hard to port even in the old days of the 1970s and 1980s. From version 2.0 on this BASIC interpreter has the feature of setting personalities at runtime.

One command controls how boolean arithemtic, string syntax, random number generator behaviour and array dimensions are handled.

SET 22, "Microsoft"

activates the Microsoft mode. In this mode, random numbers are always between 0 and 1 and the argument of RND controls the random number generators seed. Arrays start at 0 and have n+1 elements. Substring notation for strings is switched of and string commands can be used normally. Booleans false is -1.

SET 22, "Apple1"

activates Apple 1 mode. Random numbers are generated in the range from 0 to n-1 if RND(n) is called. Arrays start at 1 and have n elements. Strings can be manipulated with the substring syntax. Boolean false is 1.

SET 22, "PaloAlto"

activates Palo Alto BASIC mode. Random numbers are from 1 to n if RND(n) is called. Arrays start at 0 and have n elements. Substrings can be used. In addition to this, the interpreter is in forced integer mode. All variables and operations are done in integer. Boolean false is 1.

All of these settings can be controlled individually at runtime of a BASIC program with the SET command.

Default of the interpreter is "Apple1" with the exception of the boolean mode which is -1 at interpreter startup.

## Compiling BASIC

Basic 2.0 consists now of 6 files. basic.c is the interpreter and basic.h the prototypes. runtime.c or runtime.cpp are the runtime environments. They differ for real computers (runtime.c) and Arduinos (runtime.cpp). The prototypes i.e. the interface the the runtime code are identical and are called runtime.h. There are almost no configuarable parameters in these files.

Configuration of the hardware is done in hardware.h. Please look into the hardware section of this manual for more details.

Configuration of the language features is done in language.h.

To compile BASIC on an Arduino, copy the folder IotBasic into your Arduino sketch folder and compile. I still use 1.18.5 to compile but test the 2.x versions of the Arduino IDE occasionally.

To compile on Posix system, type

gcc basic.c runtime.c -lm

this works everywhere. On an Raspberry, if you use the Wiring or PIGPIO library, type

gcc basic.c runtime.c -lm -lwiringPi

or

gcc basic.c runtime.c -lm -lpigpiod\_lf2

On MSDOS, Turbo C 2.x compiles BASIC like this

tcc BASIC.C RUNTIME.C

The tc editor does not work as BASIC is too big for it.

## Core language set

### Introduction

The core language set is based on the Palo Alto BASIC language. This is the great grandfather of many BASIC currently on the market. Commands are PRINT, LET, INPUT, GOTO, GOSUB, RETURN, IF, FOR, TO, STEP, NEXT, STOP, REM, LIST, NEW, RUN, ABS, INT, RND, SIZE.

In the core language set there are 26 static variables A-Z and a special array @() which addresses the free memory. If an EEPROM is present or the EEPROM dummy is compiled to the code, the array @E() addresses the EEPROM.

SAVE and LOAD is also part of the core language set to allow saving and loading of programs.

### PRINT

PRINT prints a list of expressions and string constants. The separator ';'' causes data to be concatenated while ','' inserts a space. This is different from MS BASIC print where ','' prints into data fields and ';'' inserts spaces is numbers are printed. Formatted printing is supported with the '#' modifier.

Examples:

PRINT "Hello World"

PRINT A, B, C

PRINT #6, A, B, C

Tutorial programs: hello.bas and table.bas.

A positive number aligns to the right Palo Alto BASIC style. A negative number aligns to the right side. Setting #0 removes the format.

BASIC I/O is stream based. Streams are numbered. The console I/O on a POSIX system or Serial on an Arduino is stream number 1. Other prefefined stream numbers are 2 for displays, 4 for secondary serial and 16 for files. Please look at the file I/O section and the hardware driver chapter for more information.

Printing to a display would be done with

PRINT &2, "Hello World"

while

PRINT &16, "Hello World"

prints to a file.

PRINT ends a line with LF. No CR is sent at the end of a line.

### LET

LET assigns an expression to a variable. It can be ommited and is only added to the language set to ensure compatibility. Typical LET statement would be

LET A=10

A=B/2+C

Expression evaluation is done left to right with multiplication and division precedence before addition and subtraction. Comparision have lowest precedence and evalute true to -1 and false to 0. This is not C style boolean logic but closer to the old MS BASIC definition. With this, NOT, AND and OR in the Apple 1 language set can be used as bitwise operators because NOT -1 is 0.

Strings evaluate to the ASCII value of the first character.

A="A" : PRINT A

outputs 65.

The operator % calculates the integer divisions modulus. Example:

B=A%2

to test divisibility.

### INPUT

INPUT accepts a list of string constants and variable. String constants are output and variable request a user input. A "?" prompt is only displayed if not string constant has been supplied. Example:

INPUT "Input a number", A

INPUT "First number: ", A, "Second number: ", B

Until version 1.4. INPUT cannot read elements of string arrays. Only not indexed string variables are implemented. This is possible in 2.0 now.

Like PRINT, the & modifier can be used to specify an input stream.

INPUT &2, A

would read data from the keyboard of a Arduino standalone system.

In version 2.0 INPUT can do most of the things MS BASIC INPUT does. Arrays and string arrays are fully supported. The string substring notation can also be used in INPUT.

Comma seperated input of numbers is fed into comma separated variable lists. Hence

10 INPUT A, B

accepts the user responses

1, 2

or

1

2

This works only for numbers. Strings have to be given in one line. INPUT still avoids buffering and does in place variable assignment for strings. This saves a lot of memory on small systems.

For INPUT the # modifier limits the number of bytes in string input. This can be used to control I2C code and other low level functions.

If the interpreter is compiled with HASTINYBASICINPUT as compile flag, number input can be expressions including variables. This means that

INPUT A

can process input in the form

2\\*4

X

3+4\\*(4+5)

This was used in some Palo Alto BASIC programs like trek.bas.

### GOTO

GOTO branches to the line number specified. Expressions are accepted. Examples:

GOTO 100

GOTO 100+I

The later statement can be used to program ON GOTO constructs of Dartmouth BASIC.

BASIC has a line cache and remembers jump targets to speed up the code. The size of this line cache depends on the hardware. 64 is default on real computers.

### GOSUB and RETURN

GOSUB is identical to GOTO and saves the location after the GOSUB statement to a return stack. RETURN ends the execution of the subroutine.

The GOSUB stack is shallow with a depth of 4 elements on small Arduinos. This can be increased at compile time. On real computers default size is 64.

### IF

IF expects an expression and executes the command after it if the condition is true. 0 is interpreted as false and all non zero values as true in . Examples:

IF A=0 GOTO 100

IF B=A PRINT "Equal"

There is not THEN in the core language set. THEN is part of the Apple 1 language set.

Boolean expression evaluation depends on the language modes. Default is that a true condition is -1. This can be changed with a SET command.

### FOR loops

FOR have the form

FOR I=1 TO 10 STEP 2: PRINT I: NEXT I

Specifying the parameter I in NEXT is optional. STEP can be ommited and defaults to STEP 1.

Unlike in other BASIC dialects the loops.

FOR I=10: PRINT I: NEXT

and

FOR I: PRINT I: NEXT

are legal. They generate infinite loops that can be interrupted with the BREAK command which

is part of the extension language set. CONT advances the loop. This is borrowed from C.

All parameters in FOR loops are evaluated once when the FOR statement is read. FOR loops use the memory location as jump target in NEXT. They are faster than GOTO loops.

The statement NEXT I, J to terminate two loops is not supported. Each loop needs to have their own NEXT statement.

### STOP

STOP stops a program. There is no END in the core language set.

STOP STOP ends the interpreter.

### REM

Comment line start with the REM statement. As BASIC tokenizes the entire input, comments should be enclosed in quotes. Example:

10 REM "This is a comment"

Alternatively

10 REM This is a comment

can also be used in BASIC > 1.3.

### LIST

Lists the program lines. Examples:

LIST

LIST 10

LIST 10, 100

The first statement lists the entire program, the second only line 10. The last statement lists all line between 10 and 100.

### NEW

Deletes all variables and the program code. Language settings made with SET are not reset.

### RUN

Starts to run a program. A line can be specified as first line to be executed. RUN deletes all variables and clears the stacks. GOTO linnumber can be used for a warm start of a program.

### ABS

Calculates the absolute value of a number. Example:

A=ABS(B)

### INT

Calculates the integer value of a number. Example:

A=INT(B)

### RND

Calculates a random number. The builtin random number generator is a 16/32 bit congruence code and is good enough for games and simple applications but repeats itself rather fast. The argument of the function is the upper bound.

On an integer BASIC

PRINT RND(8)

would produce numbers from 0 to 7, while

PRINT RND(-8)

would produce numbers between -8 and -1.

The random number seed can be changes by using the special variable @R. See the "special variable" section for more information.

With SET 19, 1 the base of the random number can be changed to follow Palo Alto BASIC conventions. In this case numbers are created from 1 to 8.

SET 19, -1 sets the random number generator to Microsoft mode. In this mode RND(x) always produces numbers between 0 and 1 (exclusively). Called with a negative argument, the random number generator is initlialized with the absolute value of the argument to start a new sequence. This is equivalent to setting the sequence with @R. Called with 0 as an argument the same number is created. Called with any positive argument new random numbers are produced. This setting can be used to make the interpreter more compatible to Microsoft BASIC interpreters.

### SIZE

Outputs the space between the top of the program and the bottom of the variable heap. It is the free memory the interpreter has.

### Storing programs with SAVE and LOAD

LOAD and SAVE are part of the core language set if the hardware has a filesystem. On POSIX systems (Mac, Windows, Linux) they take a file name as argument. Default is "file.bas".

Arduino systems with only EEPROM support save the program to the EEPROM in binary format, overwriting all data on the EEPROM by it.

If a file system is compiled to the Arduino code. The commands work like on a POSIX system. LOAD "!" and SAVE "!" can be used in these cases to access programs stored in EEPROM.

File names must be strings. To load a program you need to enter

LOAD "myprog.bas"

i.e. with the filename enclosed quotes.

Started from the command line, LOAD will merge a program loaded with the program already in memory. In RUN mode a LOAD replaces the program in memory, clears all FOR, GOSUB and DATA pointer but keeps the variables. This way a program can chain another program.

### Special variables and arrays

The character @ is a valid first character in variable names and addresses special variables. These variables give access to system properties.

@() is an array of numbers starting from the top of memory down to the program area. It can be used like any other array in BASIC. Indexing starts at 0.

@ is the upper array boundary of the array @() i.e. the number of elements in the array.

The program area is protected by BASIC. The maximum index prevents a program to overwrite the program.

@E() is the EEPROM array. It indexes the entire EEPROM area down from the highest address.

@E is the number of elements in the EEPROM array. Unlike with the memory array @() there is no protection of saved programs in the EEPROM. @E() can overwrite the program space.

@R is the random number seed. It can be written and read.

@O is the currently active output stream.

@I is the currently active input stream.

@C is the next character of the input stream or output stream.

@A is the number of available character of the current input stream.

@S is the I/O error status. See the file I/O section for more information on it.

@V is the number of characters processed in VAL().

@U is the user variable. It can be used to extend BASIC.

The variables @O, @I, @C, and @A can be used for byte I/O on any stream.

The array @T() is the real time clock array. @T\$ is a string containing date and time. See the hardware drivers section for more information.

The string @A\$ is the command line argument on POSIX systems. See the hardware section for more details.

## Apple 1 language set

### Introduction

The Apple 1 language set is based on an early version of the Apple Integer BASIC. I never worked with the language and just took the information from the the manual. The language set adds many useful features like arrays, strings, two letter variables, boolean expressions and a few functions. Some BASICs for professional computers like CROMEMCO used BASICs like this.

### Variables

In addition to the static variable A-Z all two letter combinations which are not a keyword are legal variable names. Examples:

A0, BX, TT

Variables are placed on a heap that is searched every time a variable is used. With HASLONGNAMES enables, names can be up to 16 characters long. This is supported in BASIC 2.0.

### Strings

String variables can also have two characters followed by the \$ symbol. Example:

ME\$="Hello World"

Unlike in the original Apple 1 BASIC, strings can be used without explicitely dimensioning them. They are auto dimensioned to length 32. Strings are static. The entire space is allocated on the heap and stays reserved. This is very different from MS BASIC with a dynamic heap. The saves memory but requires a garbage collector. Static strings like in this BASIC need more memory but make the execution of time critical code more deterministic.

A string can be dimensioned with DIM to any length. Example:

DIM A\$(100), C\$(2)

A string of maximum 100 bytes and a string of 2 bytes are created. The maximum length of a string is 65535 in the default settings of the code. See the hardware section of the manual for more information on this.

Strings are arrays of signed integer. String handling is done through the substring notation. Example:

A\$="Hello World"

PRINT A\$(1,4), A\$(6,6), A\$(6)

yields

Hello W World

as an output.

Substrings can be the lefthandside of an expression

A\$="Hello World"

A\$(12)=" today"

PRINT A\$

outputs

Hello world today

Please look into the tutorial files string1.bas - string3.bas for more information. The commands LEFT\$, RIGHT\$, and MID\$ do not exist in the core language set but are supplied as an extension in BASIC 2.0.

The length of a string can be found with the LEN command. Example:

A\$="Hello"

PRINT LEN(A\$)

String arrays are possible if the BASIC interpreter is compiled with the respective option. This is an extension of the Apple 1 language set.

In BASIC 2.0 strings can be enclosed in two ways "This a string" or 'This is a string'. With this both, "'" and '"' are valid strings.

### Arrays

Arrays are autodimensioned to length 10. They start with index 1. If a different array length is needed the DIM command can be used. Example:

DIM A(100)

Array variables can be used like normal variables except that they cannot be active variables in a FOR loop.

Compiled with the multidim options arrays can be two dimensional. This is an extension of the Apple 1 language set.

Arrays range from 1 to the maximum index by default. The statement above reserves array elements from A(1) to A(100).

With

SET 12, 0

the array lower bound can be changed to 0. The array ranges from A(0) to A(99) now. It still has 100 elements. SET 12 can be used to set any postive lower bound. This holds for all arrays except the special arrays starting with @. It can be changed at any time as SET 12 only modifies the offset but not the memory location.

Any setting with SET 12 will remain active even after NEW or CLR clears the interpreter state. See the section of the SET command for more information.

SET 21, 1 changes the number of array elements that are allocated. In combination with SET 12, 0 the interpreter allocates array elements A(0) to A(100) if DIM A(100) is called.

### Logical expressions NOT, AND, OR

Logical operators NOT, AND, and OR have lowest precedence. They are at the same time bitwise logical operations on integers. Example:

PRINT NOT 0

yields -1 which is an integer with all bits set. Logical expressions like

IF A=10 AND B=11 THEN PRINT "Yes"

can be formed.

### Memory access with PEEK and POKE

PEEK is a function which reads one byte of BASIC memory. Example:

PRINT PEEK(0)

outputs the first byte of BASIC memory.

POKE writes a byte of memory. Example:

POKE 1024, 0

If PEEK and POKE are used with negative numbers they address the EEPROM of an Arduino. -1 is the first memory cell of the EEPROM. Peek and poke return bytes as signed 8 bit integers.

### THEN

THEN is added for comaptibity reasons. Typical statements would be

IF A=0 THEN PRINT "Zero"

IF B=10 THEN 100

### END

END ends a program. On systems with EEPROM dummies in flash memory, END also flushed the buffer. On system with file operations, END flushed the file buffers.

### CLR

CLR removes all variables and stacks.

### HIMEM

HIMEM is the topmost free memory cell. The difference SIZE-HIMEM is the size of the stored program.

### TAB

TAB outputs n whitespace characters. Example:

TAB 20: PRINT "Hello"

Statements like

PRINT TAB(20); "hello"

are supported in BASIC 2. Until BASIC 1.4 is was not supported in the core language.

### SGN

The SGN function is 1 for positive arguments and -1 for negative arguments. SGN(0)=0.

### Multidim and String Array capability

If BASIC is compiled with the HASMULTIDIM option, arrays can be twodimensional. Example:

DIM A(8,9)

A(5,6)=10

It also activates one dimensional string arrays.

DIM A\$(32,10)

dimensions an array of 10 strings of length 32. Assignments are done use double subscripts

A\$()(3)="Hello"

A\$()(4)="World"

PRINT A\$()(3), A\$()(4)

outputs

Hello World

The first pair of parentheses are the substring part and the second pair the array index.

## Stefan's extension language set

### Introduction

The language extensions here are things I found useful and that were missing in Apple Integer BASIC. They are rather simple extensions of the language.

### FOR loops with CONT and BREAK

FOR loops can be ended with BREAK. Example:

10 FOR I=1 TO 10

20 PRINT I

30 IF I=8 THEN BREAK

40 NEXT

will end the loop after the 8th iteration. BREAK always ends the innermost loop just like the C break command.

Alternatively CONT can be used to move forward in a loop. Example:

10 FOR I=1 TO 10

20 IF I%2=0 THEN CONT

30 PRINT I

40 NEXT

will only display the odd values. CONT skips the entire rest of the loop and skips to the next I.

Using CONT and BREAK reduces the number of GOTO statements in a program and makes them more readable.

### Command line CONT

In interactive mode, CONT has a different function. It restarts a program previously stopped with STOP.

### ELSE

ELSE can be used after THEN in one line.

IF A=0 THEN PRINT "Zero" ELSE PRINT "Not zero"

In programs ELSE can also be on the subsequent line

10 IF A=0 THEN PRINT "Zero"

20 ELSE PRINT "Not zero"

With the structured programming language set activated, multiline IF THEN ELSE is possible.

### Character output with PUT and GET

Single characters can be written and read with the PUT and GET statement. Like PRINT and INPUT, these commands can take the I/O stream as an argument after the & modifier. Example:

GET &2, A

reads a character into variable A from the keyboard I/O stream.

PUT &2, "H", 65

writes to characters to the display I/O stream. The behaviour of PUT and GET is hardware dependent. On Arduinos they are non blocking byte streams. On POSIX systems GET is blocking input.

### Low level functions USR

USR is a two argument function accessing system properties. See the program hinv.bas in the tutorial for examples. USR can also be used to quickly add functionality to the interpreter. This will be explained in the hardware section of the manual.

Example for USR:

PRINT "The EEPROM size is ", USR(0,10)

### Low level calls CALL

CALL expects a numerical expression as an argumen. Currently only CALL 0 is implemented. It flushes all open files and ends the interpreter on POSIX systems. On Arduinos CALL 0 restarts the interpreter.

### Setting system properties with SET

SET is a two argument function setting system property. Please look into the hardware section of the manual for more information. Most but not all settings are properties of the runtime environment.

Example:

SET 0, 1

activates the debug mode of the interpreter.

SET 0, 0

ends the debug mode.

### SQR

On an integer BASIC the function calculates an integer approximation of the square root. It is the number closest to the exact result. It can be larger or smaller then the exact result. Example:

PRINT SQR(8)

In floating point system the library function is used.

See math.bas in the tutorial for more information.

### POW

This two argument function calulates the n-th power of a number. It replaces the '^' operator of some BASIC dialects. Example:

PRINT POW(2,4)

In integer systems POW calculates the result in a loop. It returns 0 for negative arguments of the exponent. In floating point systems the C function pow() is used and the result is returned.

In BASIC 1.4 and the 2.x versions '^' is added in addition to POW to make BASIC more compatible.

See math.bas in the tutorial for more information.

### MAP

MAP is taken from the Arduino library. It maps an integer in a certain range to another ranges using (at least) 32bit arithmetic. This is useful on 16 bit integer systems. Example:

M=MAP(V, 1000, 0, 200, 100)

maps the variable V from the range 0 to 1000 to the range 100 to 200.

See math.bas in the tutorial for more information.

### CLS

Clears the screen on display systems. See VT52 capabilities for more information.

### LOCATE

Locates the cursor to position X, Y on the screen. Locate goes through the VT52 interface. The underlying terminal must either be ANSI compatible on POSIX systems or use the HASVT52 extension.

### Debugging with DUMP

DUMP writes a memory and EEPROM range to the output. Example:

DUMP 0, 100

writes the first 100 bytes of the program memory.

DUMP !0, 100

writes the first 100 bytes of the EEPROM.

### Error Message capability

Error messages are stored in Arduino program memory. For systems with low program memory undefining the #HASERRORMSG flag removes explicit error messages.

### Terminal emulation and VT52 capability on display systems

BASIC has a minimal set of terminal control characters build in. Compiling the code with the HASVT52 option adds a subset of the VT52 terminal commands for systems with displays.

Most importantly

PUT &2, 12

clears the screen.

The special variables @X, @Y contain the cursor position. The variables are read write.

@X=10: @Y=10: PUT "X"

would write the letter X at position 10, 10.

The array @D() is the display character buffer by column and row. It is also read write and can be used to display characters directly.

Please look into the hardware section for the display driver commands.

## Arduino I/O language set

### Introduction

Most of the standard Arduino I/O functions are available on BASIC. The Arduino standard examples have been ported to BASIC to showcase how they are used. Please look into the repo under examples/01basics to examples/10starter for these programs.

Many Arduino I/O commands are also working an an Raspberry PI with Wiring or PIGPIO.

### PINM

PINM sets a pin to input, output, or input-pullup. Example:

PINM 10, 0

PINM 11, 1

PINM 12, 2

sets pin 10 to input, pin 11 to output, and pin 12 to input with the pullup active. The values for the second argument may be implementation specific. Some microcontroller cores use additional values. The number given in BASIC is directly handed through to the pinMode() C++ function.

### DWRITE, DREAD, and LED

Digital I/O is done with DWRITE and DREAD. Examples:

DWRITE 11, 1

A=DREAD(10)

sets pin 11 to high and reads the value of pin 10 to variable A.

The constant LED is the builtin LED on system. This can be microcontroller dependent.

PINM LED, 1

DWRITE LED, 1

lights the buildin led.

### AWRITE, AREAD, and AZERO

For analog I/O the command AWRITE and the function AREAD can be used. The pin number in these commands has to be the low level pin code of the microcontroller. For Arduino AVR systems the lowest analog pin number is the pin after the highest digital pin. On an UNO this is 14. The constant AZERO is the number of the lowest analog pin. Example:

PINM AZERO, 1

PINM AZERO+1, 1

PINM AZERO+2, 0

sets the analog pin a0 and a1 of an Arduino to output and the pin a2 to input.

AWRITE AZERO, 100

AWRITE AZERO+1, 200

V=AREAD(AZERO+2)

write 100 to pin a0, 200 to pin a1 and reads pin a2.

Pin numbering can be tricky. More information can be found in the hardware section.

### Timing with MILLIS and DELAY

DELAY triggers a delay of n ms. Example:

DELAY 1000

waits for one second.

The function MILLIS measures the time in milliseconds divided by a divisor. Example:

A=MILLIS(1)

B=MILLIS(1000)

A contains the time since start in milliseconds. B is the time since start in seconds. The divisor can be used in 16 bit systems to control overflow of a variable.

### PULSE and PLAY extensions

Two more compley Arduino I/O function are available from BASIC.

PULSE reads a pulse on a pin if used as a function. The first argument is the pin number, the second whether a LOW=0 or HIGH=1 state is expected. The third argument is the timeout in milliseconds. Note the difference to the original Arduino pulseIN() where this is microseconds. The low level Arduino commands delivers the pulse length in microseconds. PULSE delivers the pulse length in 10 microsecond units to be compatible with Integer BASIC number ranges. Example:

A=PULSE(4, 0, 100)

PULSE was named PULSEIN in ealier vesions of BASIC. This was changed in version 1.4.

As a command, PULSE writes pulses on a pin. Example:

PULSE 4, 10

writes a 100 microsecond pulse on pin 4. Optional arguments are the value the pin is to be changed. Default is 1 assuming that the pin has been set to 0 before. Example:

DWRITE 4, 1

PULSE 4, 10, 0

writes a 100 microsecond LOW pulse to the pin and then sets the pin again to HIGH=1.

An number of repetitions and an intervall can be specified as well. Example:

PULSE 4, 1, 1, 10, 100

writes 10 pulses with 1000 microseconds time difference of length 10 microseconds on pin 4.

The time unit of PULSE can be changes with SET 14, timeunit. Example:

SET 14, 1

sets the unit PULSE uses to 1 microsecond.

PLAY is a wrapper around the Arduino tone() function for Arduino systems. For ESP32 VGA systems it is mapped to the sound generator function of the FabGL library. More information on this can be found in the hardware section.

## File I/O language set

### Introduction

For systems with a filesystem there is a set of file access commands. Only one directory is supported. The file system command cannot handle subdirectories. One open file for read and one open file for write or append is supported. The file I/O commands mask the different underlying filesystems. Currently EEPROM EFS, SD, ESPSPIFFS and LittleFS are supported. Look for more information in the hardware section.

### Opening and closing files with OPEN and CLOSE

Files are opened with the OPEN command. Open needs a filename and optionally a file open mode as argument. As a file open mode 0 is read, 1 is write and 2 is append. Read mode is the default is no argument is given. Examples:

OPEN "data.txt"

OPEN "data.txt", 0

OPEN "temp.txt", 1

OPEN "append.txt", 2

The first two commands are equivalent, the file "data.txt" is opened for read. The second line opens "temp.txt" for write. A new file is created if it doesn't exist. If the file exists writing starts at the beginning and existing data is overwritten. The third commands opens the file for append.

File name lengths are filesystem specific. The maximum length in BASIC is 32 characters.

Files that have been written need to be closed not to lose the data. CLOSE without an argument closes the read file. This is not really necessary. CLOSE 1 or CLOSE 2 close the write file. Opening a new file automatically closes an open file.

OPEN and CLOSE and be used to open and close other I/O streams as well. For this, a file modifier can be added. Example:

OPEN &7, 64, 1

This command opens the Wire stream for write addressing the device 64 as a target.

If the modifier & is ommited, it is always assumed that the file stream &16 is meant.

See the hardware section for more information on supported filesystems.

### DOS commands CATALOG, DELETE, FDISK

CATALOG lists the files in the filesystem. Am optional string pattern can be used to list only file that begin with it. Example:

CATALOG

CATALOG "da"

The first command lists all file, the second one all file beginning with the pattern "da". This is equivalent to the da\* notation on the UNIX or MSDOS command line.

DELETE deletes one file. The name has to match exactly. No pattern matching is supported.

FDISK formats a file system. This is not supported for all file systems. Arguments depend on the hardware implementation. Please look in the hardware section for more information.

### Reading and writing with INPUT, PRINT, and the error variable @S

Files can be written and read with PRINT and INPUT adding the file stream modifier. The I/O stream number for files is 16. Example:

PRINT &16, "Hello World"

writes "Hello World" to the open write file.

INPUT &16, A\$

reads the string A\$ from the file. INPUT can have comma seperated arguments and reads comma separated data from BASIC 2.0 on.

I/O operations ususally report no errors on the console and keep the the program running if an error occurs. The variable @S contains the error status of the operation. @S has to be reset explicitly before using it because it remembers and error status of previous operations and is never reset by the interpreter. Example:

@S=0

INPUT &16, A$

IF @S=-1 THEN PRINT "End of file reached"

@S takes the value -1 for the end of file condition. For general error it takes the value 1.

Tutorial: fileio.bas

### Character IO with GET and PUT

Like PRINT and INPUT, the byte I/O command can take a stream modifier. Example:

GET &16, A

would read one byte from the file putting the signed ASCII value to variable A.

## Float language set

### Introduction

Floating point arithemtic is taken from the standard C float library of the platform. The size of the numbers is the size of the base type number\_t in the C code. It is set to float as a default but can be changed to double.

### SIN, COS, TAN, ATAN

The trigonometric function are standard BASIC. They are using radians as angle format. Examples:

PI=4\\*ATAN(1)

PRINT SIN(PI)\\*SIN(PI)+COS(PI)\\*COS(PI)

PI is not a predefined constant but can be calculated using ATAN. The second line should output 1.

Tutorial: trig.bas

### LOG and EXP

LOG is the natural logarithm and EXP the exponantial function.

Tutorial: stir.bas

### Floating point precision

Floating point numbers are displayed as integers if they are smaller then 1e8. Numbers from 1e8 on are displayed in exponential notation. Numbers smaller then 1e-8 are also displayed in exponential notation.

The biggest accurate integer in a 32 bit float is 16777216. The number can be recalled in BASIC by USR(0, 5).

## Dartmouth language set

### Introduction

Since the early days of BASIC there were a few language features which I summarized as Dartmouth extension. DATA and DEF were in the First Edition of Dartmouth BASIC in the early 1960. ON appeared in the Fourth Edition in 1968.

### DATA, READ, and RESTORE

Data statements can be anywhere in the program. Data items can be strings or numbers. Example:

10 DATA "Hello", 3.141, "A"

READ A\$ : PRINT A\$

READ A: PRINT A

READ C: PRINT C

Will print

Hello

3.141

65

Strings have to be in quotes. Reading data with the READ command converts the data in the same way than the assignment commands. Strings reading numbers with contain the respective ASCII character. Numbers reading strings will contain the numerical ASCII code.

Unlike other BASIC versions, reading past the end of DATA will not lead to an error. The status variable @S will contain the value 1 once one reads past the last DATA item. @S has to be resetted explicitely after this.

RESTORE resets the data pointer.

From BASIC v1.4 on argument lists are also supported in READ.

In READ, the two index substring notation like A\$(3,3) is supported from BASIC 2 on. In earlier versions READ will interpret this as A\$(3) and append the read string.

RESTORE can also be used with an argument. In the code segment

10 RESTORE 3

20 READ A: PRINT A

30 DATA 1, 4, 9, 16

the READ statement will access the third argument of the DATA list, hence the output will be 9.

DATA can be used in expressions. After the code above has run

PRINT DATA

will output 4 indicating that the data pointer will read the 4th argument after the next read. This feature can be used to store a position of the DATA list and then return to it.

10 R=DATA

20 ... some lines doing many READs

100 RESTORE R

the data pointer will be reset to the original position before reading was done.

Tutorial: readdata.bas

### DEF FN

Functions can have one numerical argument and a two character name. Example:

10 DEF FN TK(X) = X+SIN(X)

Functions have to be DEFed before use. Redefinition is allowed. Functions without an argument are allowed as DEF FNPI()=4\\*ATAN(1).

Tutorial: func.bas

In BASIC 2 there is an extended language set, activated by the macro HASMULTILINEFUNCTIONS. With this, constructs like

10 DEF FNA(X)

20 IF X>10 THEN RETURN 1 ELSE RETURN 2

30 FEND

are possible. Function can be multi line. Return values are give as an argument of RETURN. In multiline functions, GOSUB statements are not allowed. The interrupts of AFTER, EVERY, and EVENT are disabled in multiline functions. ERROR handling is also disabled.

These restrictions are needed for interpreter state consistency as the multiline function calls a separate interpreter instance. It is a bit like starting BASIC in BASIC. Currently all interrupt functions are implemented single instance and do not know about their function context. This would require a major rewrite of this code and will not happen for some time.

Multiline functions invoke a new instance of the core interpreter loop. This costs memory. Maximum depth of multiline function calls is 4. This is a parameter set in basic.h. For real computers this limit can be extended a lot an recursive programming is possible. The limit of 4 is set only for smaller Arduinos.

With the multiline function extension, functions can be called as statements. The functions

10 DEF FNP(X)

20 PRINT X+4

30 FEND

can be called as

100 FNP(10)

The multiline function construct was rarely implemented in microprocessor BASICs. They appeared in many early dialects from Dartmouth to DEC BASIC.

All variables in multiline functions are local. New variables will be deleted after RETURN. Global variables have to be defined in the gobal name space.

### ON

ON is used in combination with GOTO or GOSUB arguments. Examples:

10 ON X GOTO 100,200,300

Tutorial: ongo.bas

ON can also be used in combination with the ERROR and EVENT command. This is for compatibility with other BASIC dialects.

## Darkarts language set

### Introduction

The dark arts language set contain a set of command which can cause problems. BASIC is a beginner language and protects the user from dangerous things. This makes it somehow rigid. The three dark arts commands give access to the inner working of the heap and the program storage. They have side effects and can destroy a running program. CLR is extended in dark arts. It can delete variables from the heap.

### MALLOC

MALLOC allocates a junk of memory on the heap and returns the address. The function receives two arguments. One is the identifier. It is a 16 bit integer. The second argument is the number of bytes to be reserved. Example:

A = MALLOC(1, 64)

This reserves 64 bytes which can be accessed with PEEK and POKE as 8 bit signed integers.

MALLOC helps to reserve storage in a controlled way. It is no side effects and can be used savely as long as the POKEs stay within the allocated range.

Tutorial: malloc.bas

### FIND

FIND finds a memory segement on the heap. It can be used to find memory segements allocated with MALLOC but it also finds variables, strings and arrays.

In all of these examples the argument of the function is the name of an object and not an expression. It has to be the pure string or array without any subscripts.

All heap variables can be searched. Example:

A0=10

PRINT FIND(A0)

This command outputs the address of the variable A0. The variable is stored with the least significant byte first. The size of the variable area is the size of number\_t. It is 2 bytes for integer BASICs and 4 bytes for floating point BASICs.

Arrays are searched by specifiying the array name without arguments. Example:

DIM A(8)

PRINT FIND(A())

Arrays are stored like variables with the lowest array element first.

Strings can be searched as well. Example:

A\$="Hello World"

PRINT FIND(A\$)

This command outputs the address of a string. If A$ is a pure string and not a string array, the first byte is the least significant byte of the length of the string. The second byte is the most significant byte of the length for 16 bit strings. All following bytes are payload. Strings are not necessarily 0 terminated. One has to process the length in all code accessing strings directly.

If a memory segment allocated with MALLOC needs to be searched, a numerical expression can be specified. Example:

A = MALLOC(1, 64)

PRINT FIND(1)

If the segment to be searched is stored in a variable, FIND has to be called like in this example:

M=1

PRINT FIND((M))

Note the inner braces around M. They are needed here as they trigger expression evaluation. This is unusual for BASIC interpreters.

FIND itself has no side effects but POKEing around in the heap area can kill the interpreter if the structure of the heap list is destroyed.

Tutorial: find.bas

### EVAL

EVAL adds a new line to the BASIC code at runtime. Doing so has many side effects and EVAL does not protect you in any way. Examples:

10 EVAL 20, "PRINT X"

20 PRINT "Hello World"

RUN

LIST

In the examples above line 20 is replaced by the string in the EVAL command.

EVAL includes the line to the program heap. This changes memory addresses of all code behind the inserted line. FOR loops, GOSUB commands, the linenumber cache and READ/DATA statements use plain memory addresses. This means that the DATA pointer, all active FOR loops and active GOSUBs break. A safe way to use EVAL is to change only lines further down in the code outside of any active FOR loop and to RESTORE the DATA pointer after EVAL.

See eval.bas in the tutorial for more information.

### CLR

In the standard language sets CLR deletes all variables and resets stacks and caches. With dark arts enables it is extended to delete variables from the heap. Example:

10 DIM A(10)

20 CLR A()

30 DIM A(20)

In this example the array A() is deleted and can be redefined.

All variables defined after A() are also deleted. Example:

10 DIM A(10)

20 B$="Hello World"

30 CLR A()

40 DIM A(20)

In this examples B$ is also deleted. All variables defined after the object to be clear are deleted as well. The heap is simply reset to the previous state. This mechanism can be used to define local variables in subroutines. Simply clear the first variable defined in the subroutine before calling RETURN.

## IOT language set

### Introduction

The IoT language set are commands needed to use IoT services. They help to implement I/O function. This part of the interpreter is under active development. Some commands may change in the future.

### AVAIL

AVAIL returns the number of characters ready for input on an input stream. Example:

PRINT AVAIL(4)

returns the number of characters on input stream 4 which is the secondary serial port. The function is hardware dependent.

### String commands INSTR, VAL, and STR

INSTR finds a character in a string and returns the index. Example:

A\$="1, 2, 3"

A=INSTR(A\$, ",")

finds the first comma in the string. INSTR can be used to split strings. See splitstr.bas in the tutorial for more information.

In BASIC 2 INSTR is not fully implemented. The second argument can be a string and not just a single character.

VAL scans a string for a number and returns the value. If no number is found the return value is 0. Example:

A\$="125"

A=VAL(A\$)

VAL uses the special variable @V to report back the number of characters in the number. The status of the conversion is stored in @S. If @S is 0 after a conversion a number was found. Otherwise @S is 1. @V is set to the number of characters only if the conversion was succesful.

STR converts a number to a string. Example:

A\$=STR(125)

Tutorial: splitstr.bas, converst.bas

### SENSOR

SENSOR is the generic sensor interface. It handles sensor drivers in the hardware dependent code. The first argument of the function is the sensor number. The second argument can be used to indentify interal properties of the sensor. If the second argument is zero, SENSOR returns if the sensor code is available. Example:

10 IF NOT SENSOR(1, 0) THEN PRINT "no DHT11 sensor detected": END

20 PRINT SENSOR(1, 2)

The first command looks if there is a DHT temperature sensor. The second command read the temperature value.

### SLEEP

SLEEP puts a system into deep sleep mode. It requires one argument which is the sleep time in milliseconds. The command is currently only implemented on ESP8266 and ESP32 systems. The microcontroller restarts after deep sleep. This command needs to be used with an autostart program.

### NETSTAT

In command mode NETSTAT displays the network status. It is implemented for all Wifi systems and Arduino Ethernet. As a arithemtic constant it returns a network status. With 0 the system is not connected. 1 means connected but no MQTT connection. 3 is network and MQTT connected.

The network code is under active development.

## Timers and interrupts

BASIC can handle time interrupt and interrupts from external sources.

There are two timers implemented. The AFTER timer causes an action once and is then reset. The EVERY timer triggers events periodically until stopped. Both timers use millis() as time source. Timer events are only processed if a program is running. In interactive mode both timers are stopped but continue running.

EVENT processes external interrupts.

### AFTER

The AFTER timer is started with the command

10 AFTER 1000 GOTO 100

After 1000 milliseconds the interpreter branches to line 100. The command

10 AFTER 500 GOSUB 200

will GOSUB to line 200. A RETURN will resume program execution whereevery the program has been interrupted.

AFTER 0

disables the execution of the after command.

AFTER 1000

will start the after timer again with a new time period of 1000 ms. The GOTO and GOSUB argument of the previous after are reused.

### EVERY

EVERY syntax is exactly like AFTER but the event is running periodically.

### EVENT

EVENT listens to an interrupt and branches to a line number. In the statement

EVENT 2,0 GOSUB 1000

pin 2 is used as an interrupt pin with mode 0, which is interrupt on low. Every time the pin goes to low BASIC calls the subroutine at line 1000.

EVENT 2,0

without the GOSUB argument deletes the interrupt.

After the event is processed, the interrupt is automatically disabled. The RETURN statement enables the interrupt again.

Allowed interrupt modes are

0: LOW

1: CHANGE

2: FALLING

3: RISING

The EVENT statement does not set the pin mode of the pin. This has to be done with PINM in the BASIC program. A typical program could look like this:

10 PINM 2,2

20 EVENT 2,2 GOSUB 1000

30 Some code

1000 PRINT "Event triggered": RETURN

The PINM command sets the pin to INPUT\_PULLUP and the EVENT commands waits for the pin to be pulled down to low. It triggers the interrupt on the falling signal. While the PRINT statement is processed, no further interrupt is accepted. After return the interrupt is reenabled.

Events can be disabled with the command

EVENT STOP

Events can be reenabled with the command

EVENT CONT

Like the ERROR command, EVENT can be used with ON.

ON EVENT 2,0 GOSUB 1000

is equivalent to the command above.

### Credits and a word on timing

Both AFTER and EVERY have been taken from the legendary Locomotive BASIC. In this BASIC dialect, only GOSUB was available and the time scale was 20 ms. There were 4 individual timers. Full featured Locomotive BASIC timers are on the feature list for future releases.

BASIC can handle 1ms interrupts even on an Arduino UNO if there is not much I/O going on. Typically, 35 BASIC commands are processed in a ms by the interpreter core. It is good practice to disable interrupts with EVERY 0 at the beginning of the interrupt subroutine and to reenable it with EVERY n immediately before return.

## Error handling

### Errors in general

In BASIC there are two types of errors. Some conditions are just exceptions and the program will continue normally. Examples are reading past the end of a file or reading past the last DATA item. These operations set the flag @S to an appropriate value. BASIC language errors clear the stacks, print an error message and the program is terminated. Variables and the program pointer are preserved. The program can be continued with CONT. Effectively, an error leads to a STOP of the program plus loss of GOSUB and FOR stack information by default.

### Error trapping

If BASIC is compiled with the HASERRORHANDLING option, error behaviour can be controlled with the ERROR command.

The code line

10 ERROR GOTO 1000

would jump to line 1000 when an error is encountered. As the FOR and GOSUB stacks are cleared on error, the code from 1000 on must handle all operations to restart the program if needed. After the jump, the error handling is reset to normal behaviour, i.e. stop on error. Using the keyword ERROR as a variable returns the error code.

ERROR STOP

will switch off error handling and go to normal behaviour. It also resets the error code to 0.

ERROR CONT

causes the program to continue even in case of an error.

ERROR can be used with ON. So

10 ON ERROR GOTO 1000

is the same as the line above.

### Error codes

Currently the following error codes are supported.

Syntax error: 10

Number error: 11

Division by zero: 12

Unknown line: 13

Out of memory: 14

Stack error: 15

Index or parameter out of range: 16

String operation error: 17

Error in variable handling: 18

Error in loop handling: 19

File errors: 20

Function errors: 21

Number or type of arguments wrong: 22

EEPROM errors: 23

SDcard errors: 24

Some of the errors are technical errors and should never appear if the interpreter works properly. In addition to this error codes all token values can appear here. This happens if a specific error in a command occurs. Typically this would be RETURN without a GOSUB or the loop commands.

## Graphics language set

### Introduction

The graphics language set is a thin layer over the low level graphics functions of the graphics driver. They try to be as generic as possible and avoid driver specific code. Some of the commands behave differently on different platforms. The graphics commands were first implemented with an SD1963 TFT in mind. Currently various monochrome and color displays are supported.

### COLOR

Sets the color of the graphics pen. COLOR either accepts either 3 arguments for rgb color or one argument for vga like colors. The exact color palette depends on the graphics driver. Example:

COLOR 255

COLOR 100, 100, 100

### PLOT

PLOT draws one pixel on the display. The coordinate system starts at the upper left corner of the display. First arguments is the x coordinate, second argument is the y coordinate. Example:

PLOT 100, 100

A pixel is drawn in the colour specified by the COLOR command.

### LINE

LINE draws a line from one point to another. Example:

LINE 100, 100, 200, 200

### CIRCLE and FCIRCLE

CIRCLE and FCIRCLE draw a circle or filled circle around a point on the canvas with radius r. The last argument is the radius. Example:

CIRCLE 100, 100, 50

### RECT and FRECT

RECT and FRECT draw a rectangle or filled rectangle. The first pair of argument is the left lower corner. The second pair is the right upper corner. Example:

RECT 100, 100, 200, 200

## Structured Language Set

### Introduction

The structured language set introduces a few loop and conditional statements into BASIC for a more GOTO free code. The curent implementation is not GOTO save. This means that jumping out of loops and into loops is possible but not handled by the interpreter in a save way. Nested loop errors are also not always detected.

### WHILE and WEND

WHILE starts the loop and tests a condition. The terminal statement is WEND. Loops can be advanced with the CONT command just like in FOR. BREAK ends the loop.

Example:

A=0: WHILE A<10: PRINT A: A=A+1: WEND

### REPEAT and UNTIL

For loops with a condition at the end, use REPEAT and UNTIL.

Example:

A=0: REPEAT: PRINT A: A=A+1: UNTIL A>10

### SWITCH, CASE and SWEND

BASIC has a construction similar to the C style switch.

Example:

10 INPUT A

20 SWITCH A

30 PRINT "default"

40 CASE 1,2: PRINT "one or two"

50 CASE 3: PRINT "three"

60 SWEND

Unlike C the default code has to come immediately after the SWITCH statement before the first CASE. CASES can have comma separated lists as arguments. All statement until the next CASE are executed if the variable A matches the argument. Unlike C no BREAK statement is needed.

In CASE statements can be multiline.

In the current version the SWITCH statement cannot be nested. There can be no SWITCH in a SWITCH. This will be changed in future.

Example:

10 INPUT A

20 SWITCH A

30 PRINT "default"

40 CASE 1,2

41 PRINT "one ";

42 PRINT "or two"

50 CASE 3: PRINT "three"

60 SWEND

### Grouping statements with DO and DEND

Statements can be grouped with DO and DEND in IF THEN ELSE constructions.

Example:

10 INPUT A

20 IF A=0 THEN

30 DO

40 PRINT "A is "

50 PRINT "zero"

60 DEND

70 ELSE PRINT "A is not zero"

## BASIC 2 language extensions

BASIC 2 seperates the runtime environment from the interpreter code. In addition to this more technical change and removal of technical debts it also adds new features. As tokens can now be two byte long, there is no limit any more on the number of commands.

### String compatibility with MS Basics

In BASIC 2, string functions are supported. Currently RIGHT\$(), LEFT\$(), MID\$(), CHR\$() and ASC() are implemented. Strings can be concatenated using the + operator. These functions are there for compatibility. They are not really needed as the subscript notation offers all the necessary functionality.

Note that the underlying string code is still static, i.e. it has no garbage collection and no dynamic string space. For this reason functionality and compatibility are limited. This is done by design.

Examples:

10 A\$="abcd"

20 PRINT RIGHT\$(A\$, 2)

30 PRINT LEFT\$(A\$, 2)

40 PRINT MID\$(A\$, 2, 2)

On RUN produces the output

cd

ab

bc

as in all classical BASIC dialects.

String function cannot be nested. It is illegal to do

RIGHT\$(LEFT\$(A\$))

as the string code is internally not recursive at the moment. This might change in the future. Only string variables and constants are allowed as arguments.

String addition is supported in the standard way but only in assignments. The program

10 A\$ = "abcd"

20 B\$ = "efgh"

30 C\$ = A\$ + B\$

40 PRINT C\$

produces the output

abcdefgh

String addition is done in place. The right hand side is copied step by step to the left hand side. If a string variable appears on both sides the result depends on the order.

A\$ = A\$ + "hello"

adds "hello" at the end of the variable as expected.

A\$ = "hello" + A\$

produces

PRINT A\$

hellohello

as the first assignment copies the string "hello" to the variable and the second term is then just taken from the variable again. There is no string buffer right now storing intermediate results. Everything is directly done in place. That's why the MS string code is only implemented in assignments. Addition can not be used in functions like RIGHT\$(A\$+B\$) because there is no static space for the string to be copied to.

The compatibility code also includes CHR\$() and ASC(). They convert a number to a string and back.

TAB can be used in print commands as well.

PRINT TAB(20); "hello" or PRINT TAB(20) "hello" is legal now and produces the same result as TAB(20): PRINT "hello"

### The line editor

The EDIT command starts a vi style line editor optimized for serial terminals. The line number has to be given as an argument.

With the program

10 PRINT "hello world"

in memory, the editor can be started by calling

EDIT 10

It produces the two line output

10 PRINT "hello"

^

The arrow below the line is the cursor position. The following commands are supported

j: move cursor left

k: move cursor right

^: go to beginning of line

$: go to end of line

:: got to the next colon

x: delete cursor character

X: delete all characters until end of line

s: substitute cursor character

i: insert at cursor position

a: append at end of line

Q: exit the editor without saving

u: undo all edits and stay in editor

Key bindings to terminals are in preparation. A full editor is also in the backlog but not yet scheduled for implementations.

# Hardware drivers

## Buildin Programs

BASIC 2.x can have a few BASIC programs compiled into the code. They have to be provided in modules like buildin.h. This feature is activated by setting HASBUILDIN in hardware.h and setting the right BUILDINMODULE.

Buildin programs can be loaded and saved with LOAD and SAVE. They can be openend and read with OPEN and they can be shown with CATALOG.

These programs are always scanned first, hence a program with the same name on the filesystem cannot be accessed. Best use a name convention or prefix that distinguished the build in programs from normal programs. If the characters \_ or . are used as prefixes for the build in program, they are invisible for CATALOG.

## I/O Streams

### Introduction

BASIC I/O goes through input output streams. Every stream has an integer number. Sending data to a stream sends it through to a device driver. These device drivers handle the underlying OS or microcontroller cores. The logic is as close as possible to the Arduino stream class.

Streams are

1: Console input and output

2: Keyboard or keypad input and display output

4: Secondary serial stream

7: I2C protocol

8: RF24 radio interface

9: MQTT network connection

16: File I/O

All other stream numbers are currently unused but will be allocated to drivers in the future. The stream numbers below 32 should not be used for extensions.

Stream numbers are agruments of the modifier of OPEN, CLOSE, PRINT, INPUT, GET and PUT. Typical commands would look like:

INPUT &2, A$

PRINT &4, "Hello world"

GET \$2, A

PUT &16, "A", "B"

The expression after & is the I/O stream number.

There is a default I/O stream compiled into the code. For systems not compiled with the STANDALONE option this default stream is the console stream 1. If the STANDALONE option is used, the default stream is 2.

The active I/O stream can be changed by stetting the special variable @O for output and @I for input. The two code lines

10 @O=2: PRINT "Hello World"

and

10 PRINT &2, "Hello World"

both write "Hello World" to the display.

The active I/O stream is always reset to the default stream once the interpreter changes to interactive mode. This is a failsafe to avoid a program to shut out a user permanently.

The default stream can be changed permanently with the SET command.

SET 3,0 sets the default output stream to console.

SET 3,1 sets the default output stream to display.

SET 5,0 sets the default input stream to console.

SET 5,1 sets the default input stream to display.

### Serial Stream

Stream 1 is the default serial I/O stream on Arduino IDE programmed microcontrollers and it is standard input and output on POSIX like systems. The behaviour of stream 1 differed between the two version.

On microcontrollers stream 1 is a true byte I/O stream and non blocking. The command

GET A

will return immediately and return 0 in the variable A if no character is available on the stream. The function AVAIL(1) or the special variable @A will return the correct number of characters in the serial buffer. If there is no character, the result will be 0.

On POSIX systems, getchar() is used on OS level. This means that I/O on stream 1 is blocking.

GET A

will wait for a character. AVAIL(1) or @A will always return 1 to make sure that a program like this

10 IF AVAIL(1)<>0 THEN GET A

runs correctly on both archtitectures.

For line oriented input with the INPUT command, the difference is irrelevant. INPUT works in the same way on both system types.

There is currently no mechanism to change the precompiled baud rate of the serial stream at runtime of a BASIC program. Default in the interpreter code is 9600 baud. This can be changed at compile time by changing serial\_baudrate in the code.

A line end is LF, which means ASCII 10. This is UNIX style end of line. DOS and other systems usually send CR LF which is 13 10 in decimal ASCII. If you use a terminal program to interact with BASIC, "only LF", should be configured in the settings. Currently stream 1 cannot be reconfigured to use LF CR. Stream 4, the secondary serial stream, can use CR LF. This is mostly relevant for input.

### Display and keyboard drivers

Keyboard and display are accessed through stream 2. Only one device can be present, i.e. one keyboard and one display. Multidisplay or multikeyboard systems are not supported. Stream 3 is reserved for these usecases but currently not implemented.

BASIC currently supports either PS2 keyboards and the keypad of the LCDSHIELD as input device. For PS2 keyboards on standard Arduino systems the patches PS2keyboard library should be downloaded from my repo for optimal functionality. It implements keyboard.peek() which has a few advantages.

PS2 keyboard input is buffered and non blocking. AVAIL(2) gives the correct number of characters in the keyboard buffer.

GET &2, A

return 0 if no character is present and the ASCII value if a character is there.

GET &2, A\$

transfers the result directly in the string A$ as a first character. The string is empty if no key was pressed.

The keymap is compiled into the code. A few standard US or European keymaps are supported.

If the code is compiled with the LCDSHIELD option, the keys of the shield can be used to input data. The "select" key of the shield is mapped to ASCII 10 means "end of line". The arrow keys of the shield are mapped to the digits 1 to 4.

INPUT &2, A

can be used on an lcd shield for number input. The command returns after select and A has a valid number. Alternatively

INPUT &2, A\$

will enter a string consisting of the digits 1 to 4.

GET &2, A

will contain either 0 if no key is pressed or the ASCII code of the key. The command will wait until the key is released. Keypad input is unbuffered. No interrupt or timer function is currently implemented on keypads. This will change for future releases of the code.

ASCII output to a display is sent through the output stream 2. Displays are handled by a generic display driver. For low memory systems the display driver supports a non scrolling / unbuffered mode. Only a few basic display functions are supported. If there is enough memory, the display driver can be compiled in scrolling / buffered mode. For the the option DISPLAYCANSCROLL has to be defined in hardware-arduino.h (1.x) or hardware.h (2.x). All characters are buffered in a display buffer which can be accessed byte wise.

In non buffered mode the display driver has the following functions:

The cursor of the display can be accessed special variables @X and @Y. Both variables are read/write. One can find out where the cursor is and set its position. Output with PUT &2 and PRINT &2 goes to the next cursor postion.

ASCII character 7 is the bell character. It calls the dspbell() function which is empty by default. Any action can be implemented here.

ASCII character 8 is backspace and delete. This is needed for compatibility with certain terminals.

ASCII character 9 is tab to the next multiple of 8 tab stop.

ASCII character 10 goes to the beginning of next line (Unix style LF).

ASCII character 11 is vertical tab, going to the next line in the same column.

ASCII character 12 clears the display. PUT &2, 12 is clear screen on the display.

ASCII character 13 goes to the beginning of the currently line (true CR).

ASCII character 127 goes back one cursor position and clears the character. It is "delete".

ASCII character 27 (ESC) sets the terminal to esc mode and triggers vt52 character interpretation.

ASCII character 2 (STX) sends the cursor to the home position. This feature will also be used for Epaper support in the future.

ASCII character 3 (ETX) is used for page mode displays to trigger the redraw.

Buffered displays add scrolling. Once the cursor goes beyond the last line, the display is redraw with the first line disappearing. This is simple line based software scrolling.

Each character in the display is buffered a display buffer. This buffer can be accessed through the special array D(). Writing to the array immediately display the character on the display. Reading from it gives the displayed character at the index position. The array starts from 1. The index advances by column and then by row.

Currently LCD displays 16x2 and 20x4, Nokia 5110, ILI9486, ILI9488, SSD1306, and SD1963 are supported. All these displays use 8x8 or 16x16 fonts. Nokia has 10 columns and 6 rows. SSD1306 character buffer dimension depend on the display size. 16x8 is the most common size. ILI9488 has 20 columns and 30 rows. It is portrait mode by default. A 7 inch SD1963 has 50 columns and 30 rows. ILI9486 displays are based on the MCUFRIEND library with parallel access.

Some displays use page based low level graphics drivers. These displays mirror the entire canvas in memory on a pixel basis. When one draws to the canvas, nothing is shown until an update command transfers the buffer to the display. SPI bus based monochrome display use this technique. The ILI9488 and SSD1306 Oled driver use this mechanism. By default these displays behave like the other displays and page mode is of. Drawing of graphics and text appears slow.

Activating the page mode is done with

SET 10, 1

after this all updates stay in memory until an ETX code is sent with

PUT &2, 3

only then the display is redrawn. The display can be set back to character mode by

SET 10, 0

For epaper displays page mode is default. Epaper disply integration is in preparation.

In addition to the displays, VGA output is supported with the FabGL library on ESP32 systems. These systems are described in the special systems section.

### VT52 terminal emulation

If compiled with HASVT52, BASIC has a full VT52 terminal emulation, including many of the GEMDOS addons. Please look at the wikipedia page https://en.wikipedia.org/wiki/VT52 for a full list of the originally supported ESC sequences.

All standard commands except the keypad and graphics sequences are supported. All GEMDOS extensions except reverse video are supported where it makes sense. On color displays, the color code is sent as a VGA color in the range from 0 to 15.

The most useful VT52 sequences in a BASIC program are the ons manipulating entire line.

PUT &2, 27, "L"

inserts a line, while

PUT &2, 27, "M"

deletes a line.

ON LCD displays, switching on the cursor is supported.

PUT &2, 27, "e"

switches on a blinking cursor, while

PUT &2, 27, "f"

switches it off again.

The following ESC sequences are supported right now

ESC A Cursor up - Move cursor one line upwards.

ESC B Cursor down - Move cursor one line downwards.

ESC C Cursor right - Move cursor one column to the right.

ESC D Cursor left - Move cursor one column to the left.

ESC H Cursor home - Move cursor to the upper left corner.

ESC I Reverse line feed - Insert a line above the cursor, then move the cursor into it.

ESC J Clear to end of screen - Clear screen from cursor onwards.

ESC K Clear to end of line - Clear line from cursor onwards.

ESC L Insert line - Insert a line.

ESC M Delete line - Remove line.

ESC Yrc Set cursor position - Move cursor to position c,r, encoded as single characters.

ESC Z ident Identify what the terminal is, see notes below.

ESC E Clear screen - Clear screen and place cursor at top left corner.

ESC b# Foreground color - Set text colour to the selected value (only implemented on color displays).

ESC c# Background color - Set background colour (only implemented on color displays).

ESC d Clear to start of screen - Clear screen from the cursor up to the home position.

ESC e Enable cursor - Makes the cursor visible on the screen (only implemented on LCDs).

ESC f Disable cursor - Makes the cursor invisible (only implemented on LCDs).

ESC j Save cursor - Saves the current position of the cursor in memory.

ESC k Restore cursor - Return the cursor to the settings previously saved with j.

ESC l Clear line - Erase the entire line and positions the cursor on the left.

ESC o Clear to start of line - Clear current line from the start to the left side to the cursor.

ESC p Reverse video - Switch on inverse video text (implemented but with no function).

ESC q Normal video - Switch off inverse video text (implemented but with no function).

ESC v Wrap on - Enable line wrap, removing the need for CR/LF at line endings.

ESC w Wrap off - Disable line wrap.

### VT52 graphics extension

On displays with graphics capabilities, the graphics commands can also be sent to the display with ESC sequences. This is not part of the standard VT52 command set and also not part of the GEMDOS extension.

The graphics code uses three general purpose registers x, y, and z. x and y are 14 bit registers while z is a 7 bit register. Setting a register is done with the sequences

ESC x low high

ESC y low high

ESC z char

Like in cursor control the arguments are transfered as printable characters. 32 is subtracted from the ASCII value of low, high or char.

Graphics commands are initiated by

ESC g command

where command is one character. The following graphics sequences are currently implemented

ESC g s - set the graphics cursor to position x and y

ESC g p - plot a point at the graphics cursor

ESC g l - draw a line from the graphics cursor to position x and y

ESC g L - draw a line from the graphics cursor to position x and y and move the graphics cursor to x, y

ESC g r - draw a rectangle between the graphics cursor position and x, y (like RECT)

ESC g R - draw a filled rectangle

ESC g c - draw a circle at graphics cursor position with radius x

ESC g C draw filled circle

Color is set with ESC c.

### Secondary serial stream

The secondary serial stream is addressed through stream number 4. For microcontroller platforms with a hardware secondary serial port, the Serial1 object on the Arduino library is used. This can be changed in hardware-arduino.h (1.x) or hardware.h (2.x) by modifiying the functions beginning with prt. For microcontrollers without multiple hardware serial ports Softwareserial is used. The RX and TX pins are set through the macro SOFTSERIALRX and SOFTSERIALTX. Default are pin 11 and 12. This conflicts with the SPI pins. If SPI is to be used the soft serial pins have to be changed.

Default baudrate on the secondary serial port is 9600 baud.

The secondary serial port implements a blockmode function. Blockmode is on by default. Without blockmode an INPUT command would wait until a LF is received. This means that

INPUT &4, A$

would never return if the sender on the serial port does not send LF. If a modem with AT commands or any other device is on the input port, the BASIC program would stop indefinitely if the sender side misbehaves. With blockmode active, the INPUT command handles I/O differently.

With the command

SET 7, 1

the blockmode parameter can be set to 1. In this mode the entire serial buffer is transfered to the string A$. The number of characters waiting in the serial buffer for transfer can be found by the function AVAIL(4). This number of character is in the string after INPUT &4 with the length of the string set to the value of AVAIL(4) before INPUT. Immediately after read, AVAIL(4) is zero. For secondary serial blockmode 1 is the default.

Setting the blockmode parameter to a value greater 1 the timeout mode is activated. INPUT reads all characters in the buffer and waits timeout microseconds for further input until it returns. It also returns if the string is full. Example:

SET 7, 500

INPUT &4, A$

In this mode the INPUT command would read all characters, wait for 500ms for further input, and returns latest after this 500ms or when the string is full.

For communication with modems, the secondary serial channel has one more parameter. Normally only LF is sent after the end of a line. Some modems and other devices on a serial port expect a LF CR sequence at the end of a line. The communication is blocked if CR is not sent. With the command

SET 6, 1

CR mode is activated. A print command to the serial channel like

PRINT &4, "AT"

would print the characters "A", "T", followed by a LF (ASCII 10) and a CR (ASCII 13).

Answer of a model could be collected with the input sequence above.

Unlike the primary serial port, the baudrate of the secondary port can be changed by the BASIC program. Example:

SET 8, 31250

would set the baudrate to the standard MIDI baudrate of 31250. Any allowed baudrate of the hardware platform can be used. The serial port is reset after this command.

### Wire communication

Stream &7 is used for Wire i.e. the I2C protocol. BASIC uses the implementation of the Arduino Wire library and the Wire object. Actually, the I2C protocol is very compley and the Wire library hides this complexity making it a byte stream. PRINT, INPUT, GET and PUT can deal well with byte streams, therefore most of the Wire libraries functionality can be used from BASIC. There is a master and a slave mode implementation.

In master mode, the OPEN statement for the I2C stream is used in read mode. Example:

OPEN &7, 104

prepares I2C to control the slave with address 104. This address is a real time clock. Any valid I2C slave address can be used as an argument in the OPEN statement. OPEN &7 does not transfer any data and it also does not make any calls to the Wire library except if the microcontroller was in slave mode before. Then OPEN restarts the Wire object in master mode.

Sending data can be done with PUT for sequences with just a few bytes. Alternatively the information can be collected in a string and then sent with print. Examples:

PUT &7, 0

sends a null byte to the device.

PRINT &7, A\$

sends as many bytes as are stored in A$ to the device. LEN(A$) is the number of bytes transferred. The maximum is 31 bytes, a limit given by the wire libraries internal buffers.

Requesting a byte from the device can either be done with GET or with INPUT.

GET &7, A

requests one byte from the device.

INPUT &7, A\$

requests as many bytes as fit into the string, maximum 32. After INPUT the string will always have a length of 32 bytes on AVR. The Wire library does not transmit the number of transferred bytes properly despite some comments that indicate otherwise in the documentation. The library returns the number of requested bytes.

One can request fewer bytes than fit into the string by using the # modifier in the INPUT command.

INPUT &7, #8, A$

would request 8 bytes from the device. The length of the string will be 8 after the command. Again, this does not mean that the device really sent 8 bytes.

On stream 7 the status variable @S is set after each transmission to the status of the request or sent method. From this one can detect if the transmission has worked.

Wire can be reopen in master mode if one want to communicate with another device by a new OPEN command with a different address. This way multiple devices can be controlled.

See examples/13wire/master\* for more info on master mode. This is a port of the example programs of the Arduino library from C++ to BASIC. It shows how easy it is to use Wire from BASIC.

Slave mode is open with OPEN in write mode, i.e. with 1 as a third parameter. Example:

OPEN &7, 8, 1

makes your BASIC Arduino a slave that listens on Wire port 8. This OPEN statement stops the Wire object, interaction with devices where the Arduino has to be master is no longer possible. Any open in master mode will stop the slave mode.

After slave mode is started the BASIC program continues normally. In the background there is an interrupt handler collecting the data the master sends. The program has to check periodically to see if the master has sent or requested data.

If the master has sent data, AVAIL(7) will be larger than 0. The data sent by the master can be collected by

INPUT &7, A\$

and then processed by the program. It the master sends additional data before the BASIC program has recalled the message, the buffer is overwritten. Only one transaction is stored. This means that a Wire slave needs to check AVAIL(7) in a tight loop.

If the master has requested data, USR(7, 1) has the number of requested bytes. The program should now prepare a string with this number of bytes and send it with

PRINT &7, A\$

After this, USR(7, 1) will be zero once the master has collected the data from the buffer.

Alternatively, the slave can deposite data in the buffer before the master has even requested it. A master request at a later time will be answered with the data. This would be useful for sensors of all kind. The slave measures data, puts it in the buffer with PRINT &7 and waits a while e.g. with DELAY until it does the next measurement. The new data overwrite the old one after every measurement cycle. A master can request data asynchronously at any time and will receive the data in the buffer. This BASIC program does not need to handle this.

See examples/13wire/slave\* for more info on slave mode.

There is a number of example programs in examples/13wire to show how Wire programming works.

### Radio communication

Stream &8 is used for RF2401 radio communication. For this, BASIC needs to be compiled with the ARDUINORF24 flag and the respective pin definitions set. The open statement can be used to open one stream for read and one writing simulataneously. Example:

OPEN &8, "0815x", 0

OPEN &8, "0815y", 1

The first command opens a stream for reading, using the string given as file name to generate the 5 byte pipe address. The second command opens a write pipe with a different pipe address.

OPEN on the radio interface sets @S if there is in a error in the module communication. Typically this would mean that the module is not connected properly.

After opening INPUT, PRINT, PUT and GET on stream &8 work similar to Wire. The maximum message length is restricted by the unterlying library to 32 bytes.

The radio module is started with maximum power. To power it down SET 9, x can be used. x is an integer between 0 and 3. 3 is maximum power and 0 is minumum power.

### MQTT

Stream &9 is used for networking with MQTT. MQTT is the only network protocoll that is supported and the code is experimental. The network code is generic and more protocols will be integrated in the future.

Supported network adapters are Ardunio Ethernet shields, ESP8266 and ESP32 WLAN, Arduino MKR and RP2040 WifiNINA. These network codes have some timing needs like yielding the CPU and refreshing DHCP leases. BASIC integrates this in the central statement loop processing all time critical tasks in the function byield(). This code is best tested and very stable on ESP8266 and ESP32. Other platforms are alpha.

For Wifi systems, the SSID and password are hardcoded in the BASIC interpreter binary by setting the respective lines in "wifisettings.h". In this file there are two lines defining the MQTT server and port. This data is also compiled into the code. Currently only unencrypted and unautheticated MQTT is supported as this is only a toy. The fiths line in the code is a MAC address which is needed for the Arduino Ethernet shield. For all other hardware platforms this line is ignored.

After startup of the microcontroller it tries to connect to Wifi or Ethernet. The command NETSTAT can be used to see the connection status. This command is meant for interactive use only.

Finding the connection status in a program can be done by using NETSTAT as a function.

PRINT NETSTAT

or

A=NETSTAT

will return 0 is the network is not connected, 1 if the network is connected but MQTT is not connected, 3 is network and MQTT are connected. More status values will be added here in the future.

An MQTT topic an be open with the OPEN command using the topic name as a file name. Example:

OPEN &9, "iotbasic/data", 1

would open the topic "iotbasic/data" for writing because 1 was provided as a third parameter.

OPEN &9, "iotbasic/command", 0

would open the topic "iotbasic/command" for read.

There can be one read and write topic simulataneously. After opening the topics, data can be written by the commands PRINT, INPUT, GET, and PUT.

Every command will generate one MQTT message.

PRINT &9, "humidity", A

would send a message consisting of the string "humidity" and a number from variable A.

The maximum MQTT message length is hardcode to 128 bytes but can be extended. The maximum topic length is 32 bytes.

Incoming messages are buffered in the background. A new message overwrites the old one as there is only one buffer. The function AVAIL(9) has the number of bytes currently in the buffer. If AVAIL(9) is greater then 0 then

INPUT &9, A\$

transfers the entire message into A\$. From the string the message can be parsed with INSTR and VAL if numerical data was sent. The implementation is "binary data safe". Any set of bytes can be send as a message and be processed.

On POSIX systems not networking is supported as I haven't found a suitable multi platform MQTT library and was too lazy to write one myself.

The network code will be extended in version 1.4. Currently it is only a proof of concept.

### Filesystems

BASIC runs on very different platforms. From small 8 bit AVR based system to 64 bit Macs. These computers have different filesystems and capabilities. The BASIC commands are designed to make a minimal file I/O possible. Only one directory is supported. One file can be open for reading and another for writing simulateously. Stream &16 is used for file I/O.

Supported filesystems are SD on all microcontroller platforms that have it, ESPSPIFFS for ESP8266 and ESP32, LittleFS for Arduino MKR and RP2040 based systems. The file system driver removed leading rootfs names and everything else starting with a '/'. It shows a flat namespace of filenames. BASIC itself limits filenames to a maximum of 32 bytes. Some filesystems cannot handle this. In this case the name is truncated. No check is made if the filename is legal for the particular filesystem. For ESPSPIFFS and LittleFS the FDISK command is supported. Example:

FDISK

would answer with the prompt

Format disk (y/N)?

and then format the disk. No parameters are used here.

In addition to the standard filesystems, I developed a minimalist EEPROM filesystem. This filesystem partitions an I2C EEPROM into n equal size slots that can be used for files. Formating this filesystem requires the number of slots as a parameter. Example:

FDISK 4

would partition the EEPROM to 4 equal slot. For the very common 32kB serial EEPROM it would mean that 4 files of maximum of 8k can be stored. The EEPROM size has to be hard coded into the BASIC interpeter at the moment. This will change in the future.

On POSIX and SD no formating is supported.

### Real time clock support

Currently the built-in clocks of ESP32, STM32 and Arduino MKR and I2C clocks DS1307, DS3231 and DS3232 are supported and can be accessed through the special array @T() and the special string @T$. The array elements are

@T(0): seconds

@T(1): minutes

@T(2): hours

@T(3): days

@T(4): month

@T(5): year (0-99)

@T(6): day of week (0-6)

@T(7): temperature (DS1307 only)

@T(8): direct bytewise access to the clocks registers. On a DS1307 the addresses from 8 onwards are the NVRAM memory. On DS3231 and DS3232 these are the clock internal registers for alarm followed by the memory on DS3232.

Real time clock support for I2C is now built-in. No external library is used for this. For STM32 and Ardunio MKR the clock and low power libraries are automatically used by BASIC.

For systems without a hardware clock ARDUINORTCEMULATION can be activated in hardware-ardunio.h (1.x) or hardware.h (2.x). It uses millis() to emulate a Unix style real time clock.

## Special systems and hardware components

### Introduction

The hardware section of the interpreter contains mechanism to make the platform look the same. This works for many platforms but there are exceptions. BASIC language features work differently on these systems. In this chapter these exceptions are described.

### POSIX systems

Systems compiled with the POSIX hardware driver use the features of the OS for most I/O operations. They differ from Arduino's and other MCU in some aspects.

Ususally BASIC is started from the command line on these systems. The BASIC interpreter can be started with a file argument. In this case the interpreter starts and runs the file and terminates after the program has ended.

./basic file.bas

will run the program and after completion the OS shell command prompt will reappear.

The compiler flag TERMINATEAFTERRUN controls this feature. It is set to 1 by default. If set to 0 in basic.h, BASIC will not terminate after the end of the program.

./basic file.bas hello

will pass the argument to the special string @A$. Only the first argument is processed and stored in the string. Multiple arguments need to be quoted

./basic file.bas "hello world"

will pass both words to @A$.

### Raspberry PI special features

On a Raspberry PI, frame buffer graphics is supported. The graphics commands COLOR, PLOT. LINE, RECT, FRECT, CIRCLE and FCIRCLE will write 24bit color pixels directly to the frame buffer. This feature is under development and will be fully supported in the 1.5 release after more testing.

Wiring is supported with the Arduino style commands PINM, DWRITE, AWRITE, DREAD and AREAD.

I2C and other wiring features like PULSE are not yet supported.

### Wemos D1R1 systems and shields

These UNO form factor 8266 are really popular as they are cheap and powerful. BASIC supports these boards as datalogger and standalone systems. They are not really UNO hardware compatible, so some precautions are required.

BASIC uses the raw ESP822 pinout on the right side of this table:

static const uint8\_t D0 = 3;

static const uint8\_t D1 = 1;

static const uint8\_t D2 = 16;

static const uint8\_t D3 = 5;

static const uint8\_t D4 = 4;

static const uint8\_t D5 = 14;

static const uint8\_t D6 = 12;

static const uint8\_t D7 = 13;

static const uint8\_t D8 = 0;

static const uint8\_t D9 = 2;

static const uint8\_t D10 = 15;

static const uint8\_t D11 = 13;

static const uint8\_t D12 = 12;

static const uint8\_t D13 = 14;

static const uint8\_t D14 = 4;

static const uint8\_t D15 = 5;

The values on the lefthand side are the lables on the board.

A good discussion can be found here: https://forum.arduino.cc/t/wemos-d1-pins/523831

More on ESP8266: https://tttapa.github.io/ESP8266/Chap04%20-%20Microcontroller.html

### ESP32 VGA with FabGL

BASIC runs on TTGO VGA ESP32 hardware boards. These boards are specifically designed for retro computin applications. They have a PS2 keyboard and mouse, a VGA adapter and an SD card slot. Many old OSes and computer games can run on them.

I ported the BASIC interpreter to it with a VGA 640x200 screen. The sound generator and the SD is also supported. This makes the TTGO VGA board a fully function 64 kB home computer.

Unlike other systems with displays, the generic BASIC display driver is not used. Instead, the built-in VT52 terminal emulation of the graphics library FabGL processes text output. This means that cursor positioning with the special variables @X and @X as well as the display array @D() does not exist on these systems. Cursor positioning has to be done with the standard VT52 control sequences. See the extensive documentation of FabGL for more information http://www.fabglib.org.

The sound generator of FabGL can be used with the BASIC PLAY command but the syntax is different from Arduinos. PLAY is called like this

PLAY n, frequency, duration, volume.

The parameters volume and duration are optional. Duration is measured in ms. This is like Arduinos. Setting the volume is not possible for Arduinos. The parameter is directly passed to the FabGL library. On Arduinos this parameter is ignored.

The first parameter n would be the pin number in an Arduino configuation. For FabGL it sets the wave form generator instead. The following values are possible:

128: Sine wave

129: Symmetric square wave

130: Sawtooth

131: Triangle

132: VIC noise

133: noise

256-511: Square wave with variable duty cycles, pulse length is n-255.

TTGO VGA hardware is supported for graphics with a 16 color VGA palette. Colors are mapped to this palette if the COLOR command is used with rgb color arguments as best as possible.

The SD card is supported but some cards will cause stability problems of the system.

Networking is possible but not really supported. There are internal memory conflicts that required to reduce BASIC RAM to 10 kB. This will be solved in version 1.4 of the interpreter.

### SPI RAM systems

8bit AVR systems have rather small RAM sizes. The UNO has just 2 kB which limits the number of appliactions. Otherwise an Arduino UNO would be a great computer for BASIC. The RAM limitation of the smaller 8bit microcontrollers can be removed by using 64kB serial RAM modules. The chips are really cheap and the can be connected easily.

Activating the ARDUINOSPIRAM flag on compile time will use the SPI RAM interface for memory access. If a 16 bit integer BASIC is compiled, the computer will show 32 kB of BASIC memory. If a floating point BASIC is compiled, it will see 64 kB of memory. Only 64 kByte ATMEGA 23LCV512 SPI RAM modules are supported. There is no library needed, the device driver is integrated in BASIC.

All BASIC commands work the same. Most of them with a similar performance as local RAM. This is due to the fact that the interpreter has two memory access mechanisms, one for the program token stream and one for read/write of variables. Both are buffered separately, reducing SPI memory access.

The one exception is string commands. Strings have to be copied to local memory, can be processed there and have to be compied back. String buffers of 128 bytes handle this task. For this reason, the maximum string length is restructed to 128 bytes when using the SPI RAM interface. Also, string commands should not be nested in complicated expressions. This part of the code is not tested a lot. String code is considerably slower then in direct memory situations but still faster than many old 8 bit homecomputers.

Well behaved BASIC programs like the game library of David Ahl in examples/14games have been tested and run on SPI RAM.

### Raspberry PI Wiring

BASIC for Raspberry PI uses the wiring pinout and not the header pinout. Please look at this table for more details: http://wiringpi.com/pins/

Some more can be found here: https://pi4j.com/1.2/pins/model-b-plus.html

## Low level commands

### SET

SET changes internal variables of the interpreter. Set has two arguments, the variable or class to be changed and a value. There is no systematic in the numbering of the variables and classes.

Parameters changed with SET will stay active until the system is rebooted or the interpreter restarted. They are global variables of the interpreter. NEW and CLR will not affect them.

SET 0,n switches on the debug mode. The token stream in the statement loop is displayed. SET 0,0 resets the interpreter to normal mode. SET 0,1 shows the token stream of the statement loop. SET 0,2 shows the entire token stream, including arithmetic operations, SET 0,3 displays the memory addresses with this data. The latter two settings produce a lot of output and are meant for interpreter testing. SET 0,-1 will switch on the REM pribt mode. In this mode the arguments all REM commands will be printed but no other debug output. This makes it easier to identify the location of an error.

SET 1,1 activates the autorun mode of the EEPROM. SET 1,0 resets the autorun mode. SET 1,255 marks the EEPROM as not to contain a program. SET 1,1 should only be used if a program was stored with SAVE "!" to the EEPROM. There is no safety net here. A running program in EEPROM autorun mode can always interrupted by sending the break character. This is '#' by default and defined in the BREAKCHAR macro. Alternatively the BREAKPIN macro can be defined in hardware-arduino.h (1.x) or hardware.h (2.x). This this case the pin it is set to will interrupt the program if set to low.

SET 2,1 sets the output to display mode, SET 2,0 to serial mode. This is a deprecated feature. Using @O is a better way to do this.

SET 3,1 sets the default output to display, SET 3,0 to serial mode. This is a risky command. If the output channel is not connected to a device the user can control, they are locked out.

SET 4,1 sets the input to keyboard mode, SET 4,0 to serial mode. This is a deprecated feature. Using @I is a better way to do this.

SET 5,1 sets the default input to keyboard, SET 5,0 to serial mode. This is a risky command. If the input channel is not connected to a device the user can control, you they are locked out.

SET 6,1 sets the CR output of the secondary serial port. A CR is sent after each line. SET 6,0 switches of CR. Default is off.

SET 7,1 switches on blockmode of the secondary serial port input. SET 7, timeout with a number timeout>1 sets an input timeout in ms. In both cases INPUT &4 does not wait for a line feed. SET 7,0 resets block mode.

SET 8,baud will set the baudrate of the secondary serial port and resets the port. Default is 9600.

SET 9,n will set the radio signal strength. n is between 0 and 3. 3 is maximal signal strength.

SET 10,1 sets the display update to page mode. In this mode the display does not display text or graphics until am ETX (ASCII 3) is sent.

SET 11,1 sets the TAB command to Microsoft mode. In this mode the TAB position is the absolute character count and not a relative postion. SET 11,0 resets the behaviout.

SET 12,0 sets the lower bound of all arrays to 0. This means that arrays with 8 elements DIMed with DIM A(8) go from 0 to 7 instead of 1 to 8. SET 12, 1 resets the value. This command does not change the array or anything in memory. It merely adds an offset. Any positive value can be used as an argument.

SET 13, 1 sets the keypad to repeat mode. This setting is only used in analog keypads of LCD shields. These keypads normally block on key press until the key is released. This makes GET a semi blocking function. With SET 13, 1 the GET command does not block. It directly returns the pressed key and does not wait for key release. SET 13, 0 resets to blocking mode.

SET 14 changes the time scale of the PULSE command. Default is 10. The unit is microseconds. With SET 14,1 the time scale is 1 microsecond. All positive values are supported.

SET 15 controls on the VT52 to ANSI emulation, by default it is on. This means that VT52 ESC sequences are translated to ANSI sequences. This way programs written for microcontrollers are compatible with POSIX system programs. SET 15,0 switches off the emulation. All ESC sequences are directly sent to the terminal.

The following SET commands control language behaviour. They are not part of the runtime properties and listed here for completeness. They only exist in BASIC 2.

SET 16 controls the default dimension of a string on creation. Default is 32. SET 16,20 would set the default size of undimensioned strings to 20. This parameter is only used when the string is created. Previously created string remain unaffected.

SET 17 controls the boolean mode of the interpreter. With the default value -1 the interpreter behaves like a MS BASIC interpreter. True is -1 and NOT -1 is 0. The operations NOT, AND, OR are all three bitwise logical operations of 16 bit signed integers. SET 17,1 sets the boolean mode to C style logic. This was also used by some of the old BASIC interpretes like Apple1, Palo Alto, and Cromeco. True is 1 and NOT 1 is 0. AND and OR still are bitwise logical operations but NOT behaves like the C logical NOT. This setting helps when BASIC programs of older interpreters are used. Check out trek.bas in the expamples section.

SET 18,1 sets the interpreter to integer mode even if if is compiled with HASFLOAT. Variables still stay floats internally but values are truncated on store or print. The arithmetic stays integer in this mode hence A=1/2 produces A to be 0 but A=1/2\\*2 produces A to be 1. This setting is used for compatibility with some programs of Integer BASICs including the #undef HASFLOAT variants of this interpreter.

SET 19 controls the behaviour of the random number generator. One some BASIC interpreters RND(8) would produce numbers from 0 to 7 (or 7.999999 in floating point BASICs). The value 8 is never reached. This is the default behaviour. Other BASIC would produce numbers in the range from 1 to 8. SET 19,1 switches on this mode.

SET 20 controls the substring logic. With SET 20, 0 the substring syntax is surpressed and the language feels more Microsofty. Default is 1: substrings on.

SET 21 controls the array creation behaviour. With SET 21, 1 the language creates arrays in the range of 0 to 10 with DIM A(10) i.e. in Microsoft conventions. With SET 21, 0 the array limits are 1 to 10.

SET 22 sets a group of parameters and can be used to set the interpreter personality. SET 22, "Microsoft" makes the language as MS BASIC compatible as possible. SET 22, "Apple1" makes it Apple 1 BASIC compatible. This is the default. SET 22, "PaloAlto" makes the language Palo Alto BASIC compatible and forces integer mode.

More SET parameter will be implemented in the future.

### USR

USR calls a low level function, passes one value to it and returns the value. Typically USR would be used like

A=USR(function, parameter)

USR(0,x) returns an interpreter parameter or capability. The program can find out which platform it is running on. Please look at examples/00tutorial/hinv.bas for a list of the parameters and return values.

Function numbers 1 to 31 are assigned to the I/O streams. Currently only USR(f, 0) is implemented for all I/O streams. They output the status of the stream.

Function numbers 32 and above can be used to implement individual commands. See below for more information.

### CALL

Currently only CALL 0 is implemented. Call 0 flushes all buffers. On POSIX systems it ends the interpreter and returns to the OS. On Arduino AVR and ESP the microcontroller kernel is restarted.

CALL parameters 0 to 31 are reserved. Values from 32 on can be used for implementing own commands. See below for more information.

### SLEEP

SLEEP n enters sleep mode. This is only implemented on ESPs and SAMD right now. n is the time in milliseconds. For ESP8266 the wiring for sleep mode has to be right. ESP32 can awake from SLEEP without additional wiring. ESPs restart after wakeup. An autorun program is needed for this. Once the restart happens, the program starts from the beginning. Reentry has to be handled in the program. On Arduino SAMD the interpreter uses the low power library. The program resumes after the sleep command. Sleep is experimental right now.

### Stopping programs

There are two mechanism implemented to stop programs. One listens to the default input stream, in general this is either default serial or the keyboard. The other monitors a pin.

If BREAKCHAR is defined in the BASIC code, this character will stop the program if it is encountered in the input stream. It has to be found as a first character. Default BREAKCHAR is '#'. BREAKCHAR only works on Arduino implementations with keyboard and serial I/O. Posix systems cannot use BREAKCHAR as I/O is blocking on them.

If BREAKPIN is defined, the interpreter will stop once this pin is pulled to low. By default, BREAKPIN is not defined, i.e. there is no BREAKPIN. This mechanism is for use cases where using BREAKCHAR is not practical. One can implement a separate stop button with it.

If HASSIGNALS is defined in hardware-posix.h (1.x) or hardware.h (2.x) then Ctrl-C will stop a running program. This does not work on Arduino because there are no POSIX signals on the platform. This feature is alpha and currently only briefly tested on Mac and Windows.

### Extending basic

The BASIC interpreter has several mechanisms to extend the language set without having to work directly with the interpreter data structures.

USR(N, X) with N greater than 32 will call the function usrfunction(). It gets both arguments as values and can return one number as reault.

CALL N with N greater than 32 will call the function usrcall(). It gets N as an argument.

@U is a special variable. Reading it will call getusrvar(), writing it will call setusrvar().

@U() is a special array. Reading it will call getusrarray(), writing it will call setusrarray(). In both cases the index is the array is passed on as argument.

@U\$ is the user string. It is read only. Reading it will trigger the function makeusrarray(). It can pass an set of characters to BASIC.