The P2 Retromachine Basic

Version 0.49 – beta 1

Introduction

The P2 Retromachine Basic is a Basic interpreter for a Propeller 2 platform with an external RAM. Currently tested/developed version works on a P2-EC32 board.

This interpreter was inspired mostly by 8-bit Atari Basic and Atari Softsynth. The look and feel of the interpreter can be selected from 5 predefined modes: 8-bit Atari, PC with amber/green/white monitor and Atari ST with a monochrome monitor. Selecting the mode changes default colors, font, and sound of the keyboard click. These parameters, however, can be set individually at any time using interpreter's commands.

The P2-EC32 board with the Retromachine Basic installed is an independent computer that can be programmed and used without any external PC. The interpreter allows to write a powerful program in a very short code, in a very short time.

Version 0.49 is the first beta. Most important commands, functions and language elements are now implemented, and the interpreter can be used for writing useful programs or games. However, there are still bugs to detect and remove, and a lot of new features can be added in next versions.

Preparing the hardware

The interpreter needs a P2-EC32 board with a breakout board and accessories: a digital video board, an AV board, and a serial host board.

Get the interpreter from the repository https://github.com/pik33/P2-Retromachine-Basic or from the Parallax forum topic: https://forums.parallax.com/discussion/175426/a-retromachine-basic-interpreter-0-32-prepared-to-play#latest

Connect the DV board (with a HDMI socket) to the pin 0 bank, the AV board (with VGA and audio outputs) to the pin 8 bank, and the serial host board (with full size USB Type A sockets) to the pin 16 bank. If you have another configuration of acessories/outputs, you have to recompile the interpreter from the source, changing pin definitions in their drivers. Use the latest FlexProp to compile.

Connect the monitor to the P2 system using the HDMI cable. The system generates the picture at 1024x600 resolution @ 50 Hz refresh rate. Connect the audio output to the amplifier, computer speakers or headphones (warning: may be too loud if headphones have no volume control in them). Connect the hub to the upper USB port on the accessory board. Connect a keyboard, a mouse and a gamepad or a joystick to the hub. Up to 4 joysticks/gamepads can be connected.

Prepare the SD card. It has to be FAT32 formatted: if not in FAT32 format, format it as FAT32. Unzip and copy 'bas', 'bin' and 'media' directories from the interpreter's zip archive.

'bas' directory is a working directory for loading and saving Basic programs. The archive contains several examples of simple programs that use several features of the interpreter.

'bin' directory is a default place for binary files that you can compile using Flexprop, Propeller Tool or another available compiler. Instead of flashing or directly uploading them using a serial connection to the PC, you can place compiled binaries on the SD card and run them from inside the interpreter using "BRUN" commands

'media' directory contains several audio samples and audio envelope definitions for the audio subsystem. They are based on Atari Softsynth and copied from PC Softsynth, a program that interprets Atari Softsynth tunes on a PC.

Flash the compiled interpreter binary to the P2-EC32 flash memory using Flexprop.

The interpreter itself, in its current version, needs one pin bank (8 pins) for HDMI, 2 pins for USB and 2 pins for audio output. 28 pins are still avalilable for control purposes. Future versions can use more pins for audio input, MIDI input/output and a real time clock.

Using the interpreter

The interpreter starts in an interactive mode. After power on or reset, it displays a header with the version number, available free space of RAM and the “Ready” prompt.

You can enter commands using a keyboard. They are executed immediately after you press Enter.

The Retromachine Basic has a full screen editing capability. You can move the cursor using arrows, ‘Home’, ‘End’, ‘PgUp’ and ‘PgDn’. The arrows move the cursor in the direction of the arrow. ‘PgUp’ moves the cursor to the first line and doesn't change its horizontal position. ‘PgDn’ moves the cursor to the last line. ‘Home’ moves the cursor to the editing start, that is set as 2 characters from the right; ‘End’ moves the cursor to the first space after the line.
‘Insert’ toggles the editing mode between inserting (narrow cursor) and overwriting (big cursor)

After pressing Enter, the interpreter tries to interpret the line, on which there is a cursor. In the overwriting mode it also adds an empty line under the current line.

This allows to list a fragment of code and edit it directly on this listing without a need of rewriting the full line. The immediate lines can be also edited this way and entered again.

To switch to the programming mode, write a line starting from an unsigned integer number. The line will then be added to a program, and no "ready" prompt will be displayed. You can then write another line of a program. All lines have to begin with a number and they are ordered using these numbers. To delete a line from the program, enter a new, empty line with a line number that you want to delete.

You can use more commands in one program line, separated by a colon (:). The line can not exceed 125 characters and has to fit in the one line of the screen.

After pressing the ‘Enter’ key the new line it is interpreted, precompiled and, if there were no errors detected, placed in the memory. If there were errors, the line will not be saved.

Writing a line without a number exits the programming mode. The line written will be executed and a "Ready" prompt will be displayed. To leave the programming mode without executing anything, after entering the last line of the code press Enter again.

The program can be listed using *list* and saved on an SD card with *save "filename"*. File name can be typed without "" string delimiters. The standard working directory is /sd/bas. If no argument is given to *save*, it will save the program as a last loaded file name, or 'noname.bas' if no file was loaded.

To list a fragment of a program, *list startnum, endnum* can be used.

To run the program, use *run* command in the immediate mode.

If the program doesn't end itself, ctr-c stops it. If it executes a long wait instruction, the keyboard state is sampled in one second interval, so you have to keep pressing the ctrl-c for about one second to make the program stop.

To delete the program from memory and start a new one, use *new*. To load a saved program use *load filename*

Any compiled P2 binary file saved on an SD card can be executed from within the interpreter using *brun*. That allows to have a collection of P2 binary programs on the SD card and call them when needed. The default directory for binary files is /sd/bin

The interpreter is case insensitive, you can write *RUN*, *run*, *Run* or even *rUn* or *RuN*.

Comments

To make a comment use rem or '

```
10 rem this is a comment
30 ' this is also a comment
40 print x : rem the comment can be also here : print y
```

Memory access and memory map

To directly access the memory, use ‘poke’, ‘dpoke’, ‘lpoke’, ‘peek’, ‘dpeek’ and ‘lpeek’ commands/functions. (see command list below). These commands use unified memory map:

- $0000_0000 - $0007_FFFF – HUB RAM
- $0008_0000 - $020F_FFFF – PSRAM

poke/dpoke/lpoke address, value places the value at the address. *Poke* is 8 bit, dpoke is 16 bit and lpoke is 32bit. *peek/lpeek/dpeek* retrieves a value from the address.
adr(var) returns a variable address in the memory. Aliases can be used instead of *adr*: *addr* and *varptr.* If var is an array, *adr(var)* returns the pointer to its header while *adr(var(index))* returns the pointer to the element of the array.

The code:

```
10 dim b(10) as ubyte
20 poke addr(b(1)),123
30 ?b(1)
```

will print 123

The lower part of the HUB RAM is occupied by the compiled interpreter itself. Then there is a heap, a stack and driver’s buffer on top of the HUB RAM. Standard Basic variables are also placed in the HUB RAM.

The precompiled program and its source are placed starting from the $0008_0000 in the PSRAM.

The 600 kB frame buffer is placed near the top of the PSRAM, so it ends at $200_0000. Below the frame buffer, there is an additional character buffer for the full screen editor, and then the memory is allocated for arrays declared by *dim* command.

The memory between the top of the program and the bottom of the array space is free for use. The current version of the interpreter has no memory manager for this space: you may handle it on your own. Use *memlo* and *memtop* to determine the free addresses range and *fre* to determine the free memory amount. Every redefining of an array or a program line causes memory leaking (old variables and lines are still kept there). To free the wasted memory, save the program and load it again. This will clean the unused memory.

To direct access to the free memory, a *peek/poke* family can be used.

Variables

There are several types of variables available in the P2 Retromachine Basic.

A standard variable is created by using it. Assigning a value to the variable determines its type. Suffixes like #,!,$ are optional and are simply a part of a variable name. Use them for compatibility with other Basic interpreters and compilers, and to make reading the program easier. A string variable name should end with '$' and a floating point variable name should end with '!'

A variable can be

- a string,
- a single precision float,
- a 32 bit unsigned integer,
- a 32 bit signed integer

These variables are kept in the HUB RAM

A variable type can be changed at any time, so this code

```
10 a=200
20 a=str$(a)
30 print a+"q"
```

will execute without errors and print '200q'

To define an array or a typed variable, use 'dim' instruction.

```
dim varname(size1,size2,size3)
```

An array can have up to 3 dimensions.

If no type is given, an untyped array is created. Every element of the array can be of a different type and its type can be changed at any time by assigning to it. Such arrays are kept in the PSRAM and need 12 bytes per element.

To save the memory space, a typed array can be declared:

```
dim varname(dimensions...) as typename
```

The type can be:

- 1 byte per element: byte, ubyte
- 2 bytes per element: short, ushort
- 4 bytes per element: long (=integer), ulong, single, string(or rather a pointer to it)
- 8 bytes per element: reserved for the future types: 64-bit integers and double precision floats

Using *dim* without sizes:

```
dim a as integer
```

declares a typed variable that is placed in the PSRAM – in the reality, this is a one-element array, that has an array header (16 bytes) before the variable itself.

There is no checking while assigning to typed arrays/variables. Assigning a string to an integer array element will assign a pointer to the string, assigning the float will end with its binary representation:

```
10 dim a(10)as integer
20 a(2)=1.0
30 ?a(2)
```

will output 1065353216, that is $3F800000

Operators:

Supported operators are:

- assign: =

- comparison: =, <,>,>=,<=, <>

- arithmetic: +,-,*,/,*div*, *mod*, ^

- logic: *and*, *or*, *shl*, *shr*

Priority: (highest) mul/div/mod -> add/sub/logic -> comparison (lowest)

There are 2 divide operators: '/' is a float divide and the result is always float. '*div*' is an integer divide.

Program control:

The current version implements *goto*, *gosub*, one line *if-then-else* and *for-next* to control the program execution

goto jumps to the line with the given number.

```
20 goto 100
```

An expression can be used instead a number:

```
20 goto x+1
```

In this case, however, the pointer to the target line can not be computed by the precompiler, and the target line has to be found by the runtime every time the *goto* is executed. This is much slower. Better solution is to use

```
20 on expr goto 100,110,120,130
```

This computes the expression and if 1, jumps to 100, if 2 jumps to 110, etc – the interpreter itself allows up to 64 line numbers, but the real limit is the length of the program line. If the expression result is float, it will be rounded. If it is less than 1 and more than the target lines in the list, no jump will be performed.

gosub jumps to the line and places the return address on the stack. This allows to use *return* to return to the command that is directly after *gosub*.

```
10 for i=1 to 10
20 gosub 100
30 next i
100 print i
110 return
```

This will print numbers from 0 to 100

on expr *gosub* and *gosub* expr is also allowed

pop pops the return address from the stack. This was implemented for the compatibility with Atari Basic programs and allows do write something like this:

```
10 gosub 100
20 print "Returned"
30 end
100 gosub 200
110 print "this line will not be executed"
120 return
200 pop
210 return
```

The second procedure started from 200 pops the return address so the return in line 210 will jump to 20 and not to 110

if-then-else construction has a syntax like this

```
20 if a=0 then b=1 : else c=2
```

There has to be a colon before 'else'.
There can be more than one if, but only one else in the line. Every 'if' in the line that fails will jump to the 'else' section.

```
20 if a=0 then c=1 : if b=0 then c=2 : else c=3 : rem c will never be 1 after executing this line
```

Multiple lines *if-else-endif* construction is not yet supported. Use *goto i*nstead.

for-next syntax is:

```
10 for intvar=startvalue to endvalue step stepvalue
(commands)
90 next intvar
```

When *for* is executed, it assigns a start value to the variable. When *next* is encountered, the *step* value is added to the variable. Then the variable is compared to the end value. If it is greater than the end value (or less then, if negative step), the program will continue. If not, it will jump to the instruction, that is placed immediately after *for*.

Start, end and step values can be expressions. They are evaluated only once by the *for* instruction and then saved, so this program:

```
10 a=1 : b=2 : c=10
20 for i=a to c step b : c=c+5 : b=b*2 : print i,b,c : next i
```

will print

```
1       4       15
3       8       20
5       16      25
7       32      30
9       64      35
```

Step value can be negative. Also, if there is no 'step', 1 will be assumed.
After the loop, the control variable has the value that caused the loop to exit, so after

```
for i=1 to 10 step 50
```

i will be 51.

The loop can be done in one line:

```
10 for i=1 to 10 : print i: next i
```

One line *for* loops are much faster, as the line resides in the HUB RAM cache while running

for loops can be nested:

```
10 for i=1 to 10 : for j=1 to 10 : print i,j : next j : next i
```

but not interleaved. This line will generate errors while executing

```
10 for i=1 to 10 : for j=1 to 10 : print i,j : next i : next j
```

The control variable can be changed inside a loop and a 0 step can be used, so this code:

```
10 for i=1 to 10 step 0
20 i=i+1
30 print i
40 next i
```

will print numbers from 2 to 11, simulating do..loop until i>10 that is not implemented yet.

Graphics

The Retromachine Basic uses a digital video output and generates 1024x600 graphic screen in 256 colors.

After the interpreter starts, the palette is similar to the Atari 8-bit: 16 hues in 16 luminances, colors 0 to 16 are greys.

The color can be read from the palette using *getcolor(color#).* It returns the color as 32 bit integer, that can be interpreted as $00rrggbb. *setcolor color#,r,g,b* or *setcolor color,value* changes the color in the palette. The value in the shorter syntax has the same format as in *getcolor*

There are several commands that draws graphic primitives on the screen.
color c sets the current color (from the position c in the palette) for drawing operations. ‘
plot x,y makes the pixel at x,y to have color that has been set by the *color* instruction. It also sets the starting point for *draw x,y*, that draws the line from the starting point to the new point x,y.
circle x,y,r draws an empty circle with a center at x,y and a radius r., *fcircle x,y,r* draws a filled circle.
frame x1,y1,x2,y2 draws an empty rectangle, while *box x1,y1,x2,y2* draws a filled rectangle.
color doesn’t change a color of the printed text. There is *ink* instruction to do this, while *paper* changes the color of the

background
cls clears the screen using the color set by *paper*.
font changes the font definition. 2 of them are available, 0 is ‘pc’, or ‘ms-dos’ type font, 1 is based on Atari ST mono font. Both of them are defined in 8x16 matrix.
blit allows to copy a rectangle from one place in the memory to another. There are 2 ways to use it:
- simpler: *blit x1,y1,x2,y2,x3,y3* will copy a rectangle defined by x1,y1,x2,y2 so its new upper left pixel will be at x3,y3 on the screen
- full: *blit framebuf1, x1,y1,x2,y2,pitch1, framebuf2, x3,y3,pitch2*
The full syntax allows to copy the rectangle between different framebuffers, where framebuf1 and framebuf2 are pointers to statrt of these framebuffer, pitch1 and pitch2 are lengths of the line in bytes.

Sprites

There are 16 sprites, each up to 512x576 pixel size. They reside in the HUB RAM, so its size limits the amount and size of sprites that can be simultaneously used.
You can make a sprite out of the rectangle fragment of the screen using *defsprite sprite#,x,y,w,h*, where sprite# is a number from 0 to 15 and x,y,w,h are screen cooordinates to get a sprite from.
The sprite can be moved by *sprite spritenum,x,y* where x,y is a upper left point of the sprite
The color #0 (normally black) is transparent.
To hide the sprite, move it out of the screen.

Mouse, joysticks and gamepads

If connected, a mouse can be accessed via ‘*mousex*’, ‘*mousey*’, ‘*mousek*’ and ‘*mousew*’. These functions return x and y position of a mouse pointer, mouse key status and mouse wheel status.
Not all mice reports the wheel in a so called "boot mode" used in the USB HID driver.
Up to 4 USB joysticks or gamepads can be connected.

A *stick(joynum)* function returns the digital joystick state as a 4-bit number. If 0 returned, the joystick is not present. If joystick present, its position is returned as shown below

5	6	7
9	10	11
13	14	15

strig(joynum) returns the joystick/pad button(s) state. 0 if not pressed

padx,*pady*,*padz*,*padrx*,*padry*,*padrz* and ‘p*adh*’ can be used to read an analog joystick or gamepad state. The syntax is *padx(pad#)* where pad# ia s number of pad/joystick. They are numbered from 0, and these numbers are not determined. If there is one joystick/pad, it is always be 0, if two of them connected, they will be 0 and 1, etc.
Up to 6 axes can be read. The result is a float in range from -1 to 1.
padh returns the hat position. Normally it is an integer in range from 0 to 8 if the hat moved. Different pads return different values when the hat is in the neutral position, but it is something that is not in the range 0..8

Audio system

The P2 Retromachine Basic has an 8-channel audio generator that plays sampled waveforms. Every audio channel can play a sound that has its frequency, volume, waveform, envelope, length (=time of playing), wait after command, stereo position ('pan') and sustain point.

It has also 8 sets of these parameters that can be set and then assigned to any audio channel.
There are 32 slots available for waveforms and 8 slots available for envelopes.

At the start, and after 'new' command waveforms are initialized as:

- 0 - sine,
- 1 - triangle,
- 2 - sawtooth,
- 3 - pulse, 12.5%,
- 4 - square,
- 5 - pulse, 25%,

- 30 - Atari Pokey waveform 12,
- 31 - Atari Pokey waveform 2,
- 32 - noise.

Waves 6..29 are left undefined and can be defined by ‘defsnd’
Wave 32 is not a wave. Setting 32 switches the driver into the noise generating mode, so at every sample it gets a random value. Changing the sample rate of the channel changes the noise color, from brown to white.

- envelopes are initialized as :

- 0: instant attack, linear release,
- 1: instant attack, exponential release,
- 2: instant attack, long sustain, instant release,
- 3: smooth both ends,
- 4: slow linear attack, instant release,
- 5: slow attack as in 4, smooth release,
- 6: triangle wave,
- 7: "classic ADSR"
- 8: no envelope

- all lengths are set to 1 (second), delays to 0, stereo position to center, volumes to 4.0, that's about 1/4 of maximum to avoid clipping, and sustain point to 255 (=no sustain)
- every audio channels gets the waveform and envelope the same number as the channel

To play the sound, there is *play* instruction that has 0 to 9 parameters. For parameters that aren't provided, defaults will be used. If a parameter is provided, it becomes a new default.
The negative numbers will be ignored and defaults used instead except pan, that has range from -1.0 to 1.0. To ignore it, use less than -1

The full syntax is: *play channel, frequency, delay, volume, waveform, envelope, length, pan, sustain*

Channel is the integer number from 0 to 7
Frequency is float, in Hz. There are note frequencies predefined in constants. The format is #notename(#)octave. #c4 is C note in octave 4 (about 261 Hz), while #c#4 is C sharp in 4th octave. There is no flats, use sharps instead (D^{b} has the same frequency as C$^{\#}$)
Delay is integer, the time in ms, that the play instruction will wait as if waitms was added at the end
Volume is a float from 0 to 16.384
Waveform is an integer number from 0 to 32. 0..31 selects one of predefined waveforms (as described above, if not changed by the user, see below). 32 is noise
Envelope is an integer number from 0 to 8. 0..7 selects one of predefined waveforms (as described above, if not changed by the user, see below). 8 is no envelope, the sound will not stop until next play command.
Length is a float, and it is the time in seconds for the sound to play
Pan is a float in range -1 .0 to 1.0, -1 is left, 1 is right
Sustain is the sustain point needed to play an instrument using the keyboard. The sound will stay at the sustain volume until released. Setting the sustain point at the value lower than 255 will cause the envelope to stop there. To end playing, there is the command *release channel#.* The envelope will then continue to the end.
To silence the system use *shutup*.

Envelopes and waveforms can be redefined.

defsnd defines the waveform at given slot from 0 to 31. The syntax can ve:

defsnd slot, "filename" - loads the file from /media/s directory. This is PC-Softsynth type .s2 file that has 16 bytes of a header and 1024 16-bit signed samples that define one wave period.
The interpreter doesn't check the header, it simply loads 1024 words from the offset #16, so the user can define anything and save it in such a file
defsnd slot, harmonic,harmonic... (up to 16 of them) synthesizes the waveform from harmonics
defsnd slot, negative number,number also synthesizes the wave from harmonics, but these numbers are dampening coefficients for even and odd harmonics

defenv defines the envelope.

defenv slot, "filename" loads the file from /media/h directory. This is PC-Softsynth type .h2 file, that has 16 bytes of header and 256 unsigned 8-bit samples
defenv slot,a,d,s,r defines standard ADSR envelope. A, d and r can be anything that is not negative, s should be a float in range 0.0..1.0

Setting ADSR envelope will also compute its sustain point that can be retrieved by *getenvsustain(slot)*

While the interpreter remembers all parameters from a previous play command, there is a set of commands to change them individually and then simply use play channel, freq only
These are *setenv*, *setlen*, *setpan*, *setvol*, *setwave* and *setsustain*. They all have syntax *setxxx(channel#,parameter)*

For MIDI files/input playing there is *getnotevalue* that translates midi note# to its frequency in Hz. It may be also useful to fill an array with notes to play.

Data input

Data can be input from external files using *get*. However, there are more ways to enter the data to the program:

- *inkey$* – reads the current state of the keyboard. If the key is pressed, returns its character code, else returns an empty string
- *input "prompt",var,var…* - if a string is specified as the first parameter, it prints it on the screen, then waits for the user to enter data and press Enter. After the user press Enter, it tries to read and decode the line of the text and assign read values to the listed variables.
- *read var,var…* - tries to find a *data* line, then reads the variables from it.
- *data value,value,…,value ' comment* - data line for *read*. Data line has to be 'one command only': this:
  `20 data 12,34,56 : do_something`
  is not allowed. However, the simple comment at the end of the data line is allowed:
  `20 data 12,34,56 ' hcf`
- *restore* resets the internal pointer for data lines so the next *read* command will start from the first value in the first data line in the program

Pin control

There is a standard P2 command set for the pin control: *pinwrite, pinread, pinfloat, pintoggle, pinhi, pinlo, pinstart, wrpin, rdpin, rqpin, wxpin, wypin*. Spin 2 constants are not yet implemented, numerical values have to be used instead. Addpin syntax is also not yet implemented. Use pin+addpin_val shl 6

Running a code in the independent core (cog)

The Propeller 2 microcontroller has 8 cores called cogs. 6 of them are used internally by the interpreter and hardware drivers it needs. 2 cogs are still free for use.
To run an assembly program in the independent cog, use any available method to load the program code into the memory. Then call

```
cog=coginit(address,parameter)
```

The 'address' is the address of the program stored in the memory, the parameter will be passed to the PTRA register. The returned value is the cog number. If the value is not in range from 0 to 7, there was an error and no cog was started.

If the program is no longer needed, you can stop and free the cog using

```
cogstop cog
```

You can also use the extended version of the command

```
cog=coginit(cog#,address,parameter)
```

This, however is not safe as you can reinitialize the cog, that is used in the interpreter. Also, *cogstop* doesn't check what it tries to stop, so you can stop a cog that is used in the interpreter. Use only the 2-parameter *coginit* and a value returned from it to stop a cog.

Shortcuts/abbreviations

Abbreviations are shorter versions of the commands, that end with a dot. The dot also replaces the space after the command. This Atari Basic inspired feature speeds up the writing and saves the limited space in the program line (while at the same time can make the code to be very hard to read).

```
10 f.i=1 to 100 s.10:?i:n.i
```

The *?* abbreviation replaces *print*. It has no dot after it. The ‘ replaces *rem* and also has no dot after it, but has to have a space after it

File handling

The Retromachine Basic interpreter uses FAT32 file system on the microSD card inserted into the slot on the P2-EC32. It starts in /sd/bas directory, that is default for saving and loading Basic files
To change the current directory, use c*d newpath*.
cd name will append the name to the current path. *cd /name* will use /name as a new path. *cd.*. or *cd* .. goes to the parent directory.
‘*mkdir name*’ makes a new directory in the current directory
‘*delete name*’ deletes a file or a directory
‘*copy name1 name2*’ copies the file name1 to the new file name2

To open the file, use *open #channelnum, #mode, "filename"*
#mode can be #read, #write or #append (or 1,2,4)
Use channels 2..9 - 0 and 1 are system channels, they can be reopened, but the effect may be unpredictable.
Reopening channel #0 will stop the screen output.
Use *get #channel,address,amount,position* to get amount bytes from position and place them at address - position starts at 0 (not 1!) and amount is always in bytes. The address can be either in PSRAM or in the hub (see memory map)
Use *put #channel,address,amount,position* to save bytes from the address to the file
close #channel closes the file
Load, *run* and *brun* can be called from the program. Because of the simplified syntax for the interactive mode use, if the program filename to run has to be passed in a string variable, that variable name has to have the “$” suffix, so you have to use

```
10 brun filename$
```

and not

```
10 brun filename
```

In the second case the program will try to open a file called ‘filename’.

Commands and functions list

Alphabetically sorted

abs(x)	if x is negative, multiply by -1, else left intact
acos(x)	returns the inverse cosine
adr(var) (or addr or varptr)	returns the address of the variable (= a pointer) in the memory
asc(string)	returns ASCII code of the first character in the string
asin(x)	returns the inverse sine
atn(x)	returns the inverse tangent
beep frequency, time (b.)	generates the square vave at frequency in Hz for the time in ms
bin$(value,length)	returns a string that is a binary representation of the argument, in 'length' digits Length is optimal
blit (see graphics)	copies a rectangle
box x1,y1,x2,y2	draws a filled rectangle from x1,y1 to x2,y2
brun filename (br.)	loads and executes a binary file compiled for a P2 by anything
cd newpath	changed the directory
changefreq channel,freq (cf.)	changes the frequency of the audio channe;
changepan channel, pan (cp.)	changes the stereo position of the sound in a channel
changevol channel, vol (cv.)	changes the audio volume
changewave channel, waveform# (cw.)	changes the audio waveform, 32=noise

chr$(value)	returns a one-character string that represents a given ASCII code
circle x,y,r (ci.)	draws the empty circle with the center at x,y and radius r
click on/off or 1/0	switches the keyboard click on and off
close #channel	close the file opened with this channel
cls	clears the screen
coginit ((cog#),address, parameter)	inits a cog (CPU core), returns a cog number that was initialized or an error code if not
cogstop cog#	stops a cog
color colornum (c.)	sets a color for graphic operations. There are 256 colors, 16 hues (high nibble) and 16 brightnesses (low nibble) similar to the 8-bit Atari. 0 to 15 are greys
copy filename1, filename2	copies the file filename1 to the new file filename2. If filename2 exists, it will be overwritten
cos(x)	returns a cosine of x. The unit can be switched by by deg and rad commands
cursor on/off or 1/0	switches the text cursor on and off. Because of a bug in this version, use 1/0 in programs.
data value, value… ‘comment	a data line for a read command
defenv channel,params	defines the sound envelope for the channel
defsnd channel, params	defines the waveform for the channel
defsprite spritenum,x,y,w,h (ds.)	makes a sprite from a screen rectangle
deg	switches trigonometric functions to 360 degree system
delete filename	deletes a file or a directory (the directory has to be empty)
dim	declares an array or a typed variable
dir	lists the working directory of an SD card
dpeek(x)	returns a 16-bit unsigned value from the memory at address x
dpoke x,val	writes a 16-bit unsigned val to the memory at address x
draw end_x, end_y (dr.)	draws the line to point end_x, end_y. This will be the starting point for the new draw. The first starting point can be set with 'plot'
else	use with ‘if’ to fork the program flow
end	end of the program. If there are procedures called by gosub, use end to not allow the program to go there
enter filename	loads the program without clearing the memory. May be used to append the program from another file
fcircle x,y,r (fc.)	draws the filled circle with the center at x,y and radius r
fill x,y,newcolor,oldcolor (fi.)	flood fill. Starts at x,y and replaces all pixels with oldcolor to newcolor until other color boundary found.
font fontnum	sets the font family for characters to print. 2 fonts are implemented, 0=Atari ST mono, 1=PC DOS
for (f.)	starts a loop (see program control)
frame (fr.)	draws an empty rectangle from x1,y1 to x2,y2
framebuf	returns the start address of the framebuffer
fre	returns amount of free memory available for the user
get #channel,addr,amount,pos	get bytes from a file
getcolor(color#)	returns the color from the palette as a number in format $rrggbb
getenvsustain(slot)	gets the sustain point set by defenv, to use with setsustain
getnotevalue(midinote)	returns the frequency of the MIDI note in Hz
getpixel(x,y)(ge.)	returns the color of the pixel at screen position x,y. The function returns the bacgkround pixel color even if there is a sprite drawn over this pixel.
gettime	gets a counter (ct) register, 64bit internally but 32 bits exposed now
gosub line	calls a subroutine. Put the return address on the stack and goto line
goto line (g.)	jumps to the line
hex$(value)	returns a string that is a hexadecimal representation of the argument, in 'length' digits Length is optimal, 8 used if not provided
if	with 'then' and 'else' controls the program flow
ink colornum (i.)	sets the color of the characters to print
inkey$	returns one-character string of the last pressed key, or "" if no key pressed
input prompt, var,var…	allows the user to input data using a keyboard
int	converts anything, including strings if doable, to integer, truncating the non integer part. Int(2.999)=2
left$(string,num)	returns first num characters of the string
len(string)	returns the length of the string
list startline, endline (l.)	outputs the code on the screen form startline to endline. If no endline specified, the program will be listed to the end.
load "filename" (lo.)	clears the memory, then loads a Basic program from the file. May be used in format load filename - without ""
log (value, base)	logarithm. If base not specified, then base=e
lpeek(x)	returns a 32-bit unsigned value from the memory at address x
lpoke x,val	writes a 32-bit unsigned val to the memory at address x
memlo	returns the lowest address of the free memory
memtop	returns the highest address of the free memory
mid$(string,pos,amount)	returns amount character of tthe string starting from pos. The base position is 1
mkdir dirname	makes a directory
mode modenum (m.)	sets "look and feel" of the interpreter. 0 - Atari style, 1 - PC amber, 2 - PC green, 3 - PC white, 4 - Atari ST mono. Names can be used instead of numbers: 'atari', pc_amber'', 'pc_green', 'pc_white', 'st'
mouse on/off or 1/0	switches the mouse pointer on and off
mousex	returns the x coordinate of a mouse pointer
mousey	returns the y coordinate of a mouse pointer
mousek	returns the mouse key state: 0 - not pressed, 1-left, 2-right, 4-middle
mousew	returns the mousewheel position. It is unbounded 16-bit signed integer.
new	clears the program memory and all variables
next (n.)	closes a 'for' loop
open #channel,#mode,"filename"	opens a file and attaches it to the channel
on expr goto, on expr gosub	evaluates the expression, then select the line to go to.
padh	returns the position of gamepad’s hat
padx	returns the position of gamepad’s X axis
pady	returns the position of gamepad’s Y axis
padz	returns the position of gamepad’s Z axis
padrx	returns the position of gamepad’s X rotation
padry	returns the position of gamepad’s Y rotation
padrz	returns the position of gamepad’s Z rotation
paper colornum(p.)	sets the background of the characters to print
peek(x)	returns an 8-bit unsigned value from the memory at address x
pinfloat pin	sets the pin to float
pinhi pin	sets the pin to logic 1 (normally about 3.3V)
pinlo pin	sets the pin to logic 0 (normally about 0V)
pinread pin	returns the pin logic state (0 or 1)

pinstart pin, mode, x,y	starts the smart pin mode and sets x,y registers of the pin
pintoggle pin	toggles the state of the pin (from 0 to 1 or from 1 to 0). Doesn't work if the pin floats.
pinwrite pin, value	if value=0, sets the pin output to low, else sets the pin output to high. Use to blink a led.
play channel, (parameters)	plays a sound. See audio system
plot x,y (pl.)	sets a pixel color at x,y to the color determined by a previous "color" command and sets a new starting point for 'draw'
poke x,val	writes an 8-bit unsigned val to the memory at address x
pop	pops the return address from the gosub stack, enabling to return from the nested subs directly to the main code
position x,y (po.)	sets a curcor position. The x resolution is half a character, so multiply number of characters by 2. This allows centering strings that have odd number of characters.
print (?)	outputs to the screen and moves the cursor to the new line if , or ; is not used. Use , after an argument to print the new one after a tab (8 characters), use ; to print the next argument directly after the previous.
put #channel,addr,amount,pos	put amount bytes to the file opened by open #channel, from addr, to file position pos
rad	switches trigonometric functions to radians
rdpin pin	returns the smartpin output register value, notifies the smart pin
read var,var,...	reads variables from 'data' lines
rem (')	a comment
release channel	makes the envelope in the audio channel to continue to the end.
restore	resets the data line pointer for the next read command
return	returns from a subroutine
right$(string,num)	returns last num characters of the string
rnd(value)	returns a random value. If no arguments, it is a 32-bit unsigned integer. If integer or float parameter is given, it returns integer or float that is less than,0 the parameter.
rqpin pin	returns the smartpin output register value, does not notify the smart pin
round	converts anything to the integer value with rounding – round(2.999)=3
run (filename)	starts the program. You can use "run" inside the program to restart it. If filename specified, loads it and then runs
save filename (s.)	saves the program to the file
setcolor color,r,g,b or setcolor, color (sc.)	sets the color in the palette. If 4 parameters given, they are color, r,g,b. If 2 parameters used, the new color is $rrggbb
setdelay channel, delay(sd.)	sets the delay in ms for the audio channel to wait after 'play' instruction. Default=0
setenv channel,env# (se.)	sets the predefined envelope for the audio channel. Redefine it with defenv if needed
setlen channel, len (sl.)	sets the time of the sound in seconds. Default 1.0
setpan channel, pan (sp.)	sets the stereo position, -1.0 left, +1.0 right, 0.0 center. Default 0.0
setvol channel, vol (sv.)	sets the volume of the channel, from 0.0 to 16.384. Higher values will cause clipping; default=4.0
setwave channel, wave# (sw.)	sets the predefined waveform for the channel. Redefine it with defsnd.
setsustain channel,point (ss.)	sets the sustain point for the envelope. The envelope will stop there until 'release' command is executed
shutup #channel (sh.)	silences the channel. If channel not specified, silences all of them.
sin(x)	returns a sine of x. The unit can be changed by deg and rad commands as in Atari Basic)
sprite spritenum,x,y (sp.)	move the sprite number sprite# (from 0 to 15
sqr(x)	returns the square root of x
stick(joynum)	returns the position of a digital joystick
strig(joynum)	returns the button state of a joystick or a gamepad
str$(value)	converts a value to a string.
tan(x)	returns a tangent of x. The unit can be changed by deg and rad commands as in Atari Basic)
val(string)	convert a string to a number. If it can, it returns an integer, if not, returns a single. If failed, returns 0
waitms time	waits "time" miliseconds. Doesn't have any upper limits as it creates an internal loop when time>5000 ms
waitclock (wc.)	waits for the internal 5 ms/200Hz clock tick. The clock is vblank synchronized so there is 1 vblank for 4 ticks
waitvbl (wv.)	waits for the screen vertical blank. Use to synchronize the program with the screen refresh
wrpin pin	writes to the mode register of the smart pin
wxpin pin	writes to the X register of the smar tpin
wypin pin	writes to the Y register of the smart pin

Credits

The interpreter uses the *usbnew* driver by Wuerfel21, https://github.com/Wuerfel21/usbnew and the PSRAM driver by rogloh, https://forums.parallax.com/discussion/171176/memory-drivers-for-p2-psram-sram-hyperram-was-hyperram-driver-for-p2/p1

Changelog:

0.49
First beta.
Cleaned, added a lot of functions, written a new manual in .odt format

0.28:
Enhanced screen editor
String related functions added: asc,chr$,len,mid$,left$,right$,str$,bin$,hex$,val

0.27:

Math function added: abs
Added poke,dpoke,lpoke,peek,dpeek,lpeek,adr,fre,inkey$
Added getnotevalue, changed order of 'play' parameters
'run' can now load and run the program if a filename provided
If a filename is provided without ".bas" and the file doesn't exist. the interpreter will try to add ".bas" itself
A loaded file name is kept by interpreter, then 'save' without parameters saves to that filename

If no file was loades yet, the default name is 'noname.bas'
'for-next' loop doesnt crash/work weird if a float parameters used: they are now rounded
Symbolic note names can be used in format #c4 or #c#4. They are internally converted to frequencies in Hz, as single
This means you can print #a4 and got 440.0
'dir' now lists the directory in 4-columns format
Several more bugs fixed.

0.25:

Math functions added : deg, rad, int
First version of an overengineered audio subsystem(tm) now works.

0.24:

Basic math functions added : cos, tan, atn, asin, acos, sqr (sin was already there)
8-bit Atari style command shortcuts added
'mode' command can be now used with a text names instead of numbers

0.23b:

Atari ST mode has a proper Atari ST keyboard click
Added "position" command that sets the cursor position before print
A mouse pointer has now a proper black outline.
String arays now work
Fixed a bug that disabled printing string literals while still printing string variables.
Fixed a bug that causes errors when : character was placed inside a string

0.23a:

Pin operations added
Atari ST mono look and feel (minus ST key click click that I have yet to sample)(mode=4)
PC look and feel modes (1,2,3) have now keyclick switched off as these PCs did.

0.22:

Load/save format changed to a plain text.
Solved the problem of disappearing strings.
Added dim - arrays and typed variables.

0.19:

Commands and function added: mouse, cursor, click, defsprite, sprite, sin, mousex, mousey, mousew, mousek, stick, strig, getpixel
List can have parameters that select the fragment of the program to list.
rnd can now have a parameter.

0.17:

- proper GOTO that can be written at any time
- new commands: "paper", "ink", "font" and "mode"
- "load", "save" and "brun" file names can be written without ""
- only ctrl-c is now used to break the program

0.16: for-next loop added