How to get started with Tiny BASIC

For complete beginners, the first steps can sometimes be very difficult. To help you get started, here are some examples and tips for getting started with the TinyBasRV microcomputer.

Command or program line

Computers oriented to various versions of the BASIC language usually have a language interpreter running after startup, which simultaneously performs the tasks of an "operating system", a command line and a language translator. It is similar with TinyBasRV. After startup, the computer waits for a command (or several commands) to be executed.
The user can decide whether to directly give the computer the command "to perform a certain action immediately", or to "make a list" of what the computer will do when the user instructs it to start the program.
Both modes (direct command and program writing) work with commands that the computer understands. There are 28 of them in total and you will find them described in detail in the TinyBasLang manual and at the end of it there is a summary table with an overview of the commands.
Tiny BASIC language commands are not entered into the TinyBasRV computer by typing them out one by one using individual characters. The command is passed to the computer by pressing the left Ctrl key and then simultaneously pressing the key according to the table at the end of the TinyBasLang manual.
A typical first example in programming textbooks is a task that forces the computer to say hello. With TinyBasRV, we can achieve this in 2 ways:
1) Enter the command to print the greeting directly

```
PRINT "Hello"
```

and then press Enter. The PRINT command is entered as Ctrl+P and then the text is typed between quotes. The computer prints the desired greeting on the next line. The disadvantage is that you have to do this every time you want the computer to say hello.
2) The second option is to write the greeting as a program and then run it. You can imagine that the computer is a servant waiting with a notebook 128 lines long. The computer has lines in the notebook numbered 0 to 127. On each line it can write down the command you enter. After you dictate your commands to it, you can tell it to "now do this according to the list". Some lines may be empty, others may have up to 3 commands per line.
In our case, we can enter into the computer, for example:

```
1 PRINT "Hello"
```

After pressing Enter, nothing seems to happen. But the computer has saved our command in its notebook on line 1. And it will wait for more commands. If we now enter the RUN command (Ctrl+R) and press Enter, the computer will start performing tasks in the notebook. On line 1, it will find the command to execute. The result will be the same output as in point 1). The difference is that now we can call up the greeting using the RUN command whenever we want.

More commands per line

Since the microcomputer has a 32-character command line, it would be a shame not to use it. It can write up to 3 commands per line, even with parameters separated by a **;** character. You are limited to only 32 characters. You can try writing on one line:

```
INPUT A,B ; PRINT "Area: " , A*B
```

And try what it does after you run it with Enter. If you didn't make a mistake in any character while entering the commands, you have a simple tool for calculating the area of a rectangle.
You can take advantage of this if you need to write a specific character somewhere on the display screen. You can use a combination of commands

```
CURSOR 400 ; PUTCH 143
```
The specified parameter values for commands are printed in a solid square in the middle of the screen.

How a computer processes program lines

If you decide to create a program that the computer will later execute, you can start "dictating" to it on the lines of its notebook what it will do. You can always write on any line from 0 to 127. When you have completed the program, you run it with the RUN command.
The computer will start to go through the individual lines starting from the number 0 and execute the commands it finds on them. It will skip an empty line and move to the next line 1 higher. When it reaches line 127 and should go to the non-existent line 128, it will stop executing the program.
Try the following program:
```
1 INPUT A
5 PRINT "Volume: " , A*A*A
```
And you have a simple program to calculate the volume of a cube. Try running it.
You can also terminate the program execution at any line using the END command. Add the following line to the program:
```
3 END
```
Now, after starting, the computer will prompt you to enter the value A, but it will no longer calculate the volume of the cube. This is because on line 3 it encountered the END command and will not go to line 5.

As a piece of advice, you can take the recommendation for special use of line 0. It is a good idea to learn to write a REM command on line 0 and add a short description of what the program does. After a while, you may not remember what you wanted to create with the program, and a note at the beginning of the program can refresh your memory.

Creating a program - an infinite and counted loop

Computers make their work easier by telling them what to repeat and not having to prescribe each step individually. If we wanted to print out even numbers, we wouldn't prescribe:
```
1 PRINT 2
2 PRINT 4
3 PRINT 6
…
```
We will let the computer count by 2 (that's what a computer is for, after all) and we will write the values on lines below each other. For example, like this:
```
1 LET A=0
2 LET A=A+2 ; PRINT A ; PUTCH 10
3 GOTO 2
```
The program starts printing even numbers at high speed after it is run. On the first line, we set the value of A to 0. The second line increases the value of A by 2, prints it, and uses the PUTCH command to move the cursor to the next line. Because the GOTO command on line 3 returns the program to line 2 again, an infinite loop has been created that keeps the program running until you turn off or reset the computer. The program can also be interrupted by typing the END command on the keyboard.
If you want to print only a limited number of even numbers (e.g. up to 20), modify line 3 to the new version:
```
3 IF A<20 THEN GOTO 2
```
Now when you start the computer, it will print only even numbers up to 20. The conditional IF statement evaluates the condition and returns the program to line 2 only for numbers less than 20.

And if you did not delete line 5 from the previous example, it will print as an extra bonus the volume of a cube with sides of 20. You can simply delete line 5 by typing a blank line 5:

```
5
```

Statements that are repeated multiple times in a row are usually referred to as a program loop. The IF statement can be entered at the beginning of a loop – for example:

```
5 IF condition THEN GOTO 21
        …
        loop statements
        …
20 GOTO 5
```

at the end of the loop, as was shown in the example with the list of even numbers. At first glance, it may seem that it does not matter where the evaluation of the condition is placed. In practice, this often leads to the same program behavior. The main difference, however, is that a loop with an evaluation statement at the end must always be executed at least once. If the test is at the beginning and the condition is met when the loop is first entered, the loop does not have to be executed even once.
Using the IF/THEN and GOTO statements, even in such a simple language, it is possible to create program constructs for which higher-level languages have special statements.

If you need to perform the same operations repeatedly with different parameters during a program, you can use a so-called subroutine. As an example of a subroutine, we can demonstrate something that, when called, will cause the speaker to beep several times according to the value in variable A.

```
40 BEEP 1000,200 ; DELAY 200
41 LET A=A-1; IF A>0 THEN GOTO 40
42 RETURN
```

In the program, I can now insert the line: whenever I need to signal several beeps with sound:

```
23 LET A=5 ; GOSUB 40
```

On the given line, I set the number of times to beep in variable A and then call the subroutine on line 40. It will make the appropriate number of beeps and using the RETURN command on line 42, the program will return to the line 1 number higher than where it was called from. In our case, the program would continue on line 24.

Where the program is stored (FILE and DISK commands)

The microcomputer stores the entered program in its serial EEPROM memory. Since one notebook is not enough for a good servant, our computer also has the possibility of equipping itself with multiple notebooks. It is clear that a program for taking care of the garden is different from a program for cleaning the house or maintaining a car.
Depending on the type of installed memory, 1 to 16 programs can be stored in it. The types of memories and the number of programs are summarized in the table in the TinBasHW manual.
You can switch between programs using the FILE command. For example:

```
FILE 3
```

switches to the file with program 3 (gives our servant notebook number 3 in his hand). This program (notebook in his hand) will remain current until a new FILE command. After the computer starts, program 0 is default. If you are not sure which program is currently switched, enter the FILE command without a parameter and the computer will answer you with its number.
To make it not so simple, our servant can have 2 bookcases and each of them has more notebooks. It is possible to mount 2 serial EEPROM memories on the computer board. In order to distinguish between them, one is marked "h" (like a hard disk) and is firmly soldered to the board. The other is marked "f" (like a flash or floppy disk). If a socket is used, it can be removed when turned off and a new one inserted in its place.

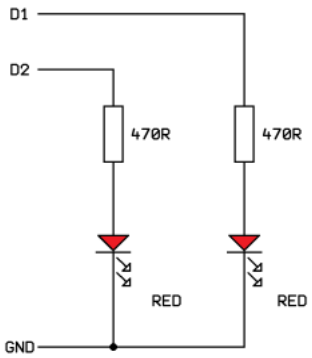
Switching between EEPROM memories is done with the DISK command – for example:

```
DISK f
```

Again, if you are unsure, you can enter the command without any parameters and the computer will tell you which disk is active. After booting, it is always disk h.

Interface with the outside world – warning light

To try working with outputs and simple control of external peripherals, you can try connecting two red LEDs to digital outputs D1 and D2 via resistors according to the following diagram:



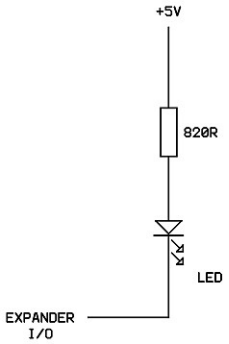
Let's load a simple program into memory:

```
1 DOUT 1,1;  DOUT 0,2
2 BEEP  2000,100;  BEEP  800,200;  DELAY 200
3 DOUT 0,1;  DOUT 1,2
4 BEEP  2000,100;  BEEP  800,200;  DELAY 200
5 GOTO  1
```

This will give us a simple warning device for a railway crossing with an audible signal. On lines 1 and 3 we always turn on one LED (and turn off the other). Lines 2 and 4 are the same and make an audible signal and a short delay. Line 5 then loops the program.

Expander

If you have an expansion board with a PCF8575 expander available, you can control a larger number of LEDs. This allows you to create, for example, a "running arrow" or other lighting effect. Connect the required number of LEDs (max. 16 pcs) to the expander outputs, always according to the picture:



You can turn on the LEDs you want to light up with the command (for 8 LEDs):

```
I2CW b11110100
```

The binary digits in the parameter determine what will be lit – position 0, and what will be off – 1. If you want a running light, you can create it for 8 LEDs like this:

```
1 LET  A=-2
2 I2CW  A ;  DELAY 200
3 LET  A=A*2+1
4 IF  A < 300 THEN  LET  A=-2
5 GOTO  2
```

This is where the magic of computer mathematics comes into play. The number -2 is made up of all 1s and one 0. The operation A*2+1 then moves this 0 up 1 position.