

Microcontroller Basics Course (3)

part 3: BASIC-52

By B. Kainka

In the first two instalments of the course, we worked with assembler, but now it's time to use a high-level language: BASIC-52.

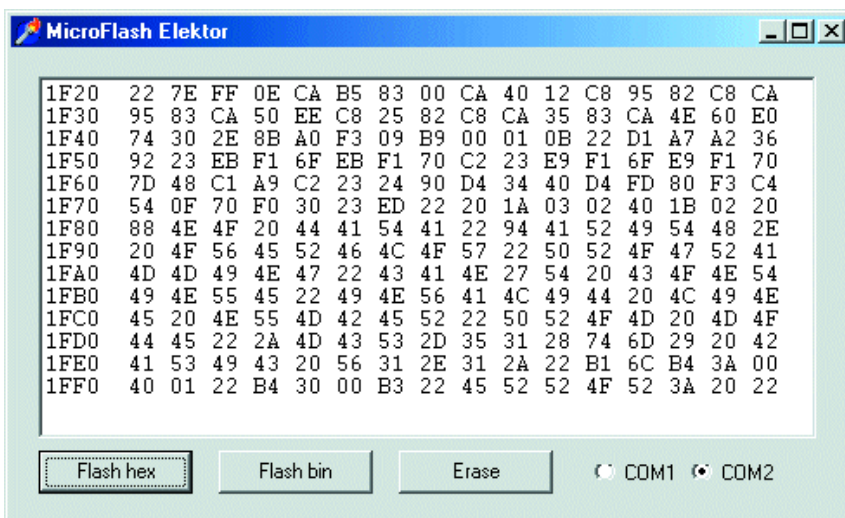


Figure 1. Downloading the Basic interpreter.

This interpreter for 8052 microcontrollers is well known to all microcontroller users and universally popular. It is located on the program diskette for the Flash Board in the form of a program file in Intel hex format with the name BASIC-52.hex. All you have to do is to use the MicroFlash.exe program to load this file into the microcontroller (Figure 1). The interpreter is 8 kB long and takes around a minute to transfer.

In order to use the interpreter, you need a terminal emulator program that can send instructions and programs to the microcontroller via the serial interface. Here we use the program BASIC.EXE from H.-J. Berndt. When you first start the program, you must configure the interface to be used (COM1: or COM2:). Communications between the microcontroller system and the PC then take place via connector K1. As shown in Figure 2, the program has two text

windows. The upper editing window is used to edit Basic source texts. Each line can be edited as much as desired; it is not sent to the microcontroller until you press <Return>. The lower text window is a direct terminal. All characters entered using this window are immediately sent to the microcontroller. The responses from the system also appear in this window.

Following a Reset, BASIC-52 first initialises itself and checks the available RAM in the system. After this, the interpreter waits for a 'space' character from the terminal. This is used to automatically determine the baud rate being used and to configure the serial interface of the microcontroller accordingly. This means that you must not press any key other than the 'space' key (ASCII 20h). BASIC-52 will respond with the following start-up message:

```
*MCS-51(tm) BASIC V1.1*
READY
>
```

Now you can enter direct commands or program lines. The query Print MTOP returns the highest address of the connected and recognised memory (Memory Top), which in this case is thus '32767' for 32 kB of RAM. When entering known keywords, you do not have to pay attention to upper or lower case. However, the interpreter converts everything internally into upper case, so program listings always appear in upper case. You can try this

out using a small test program called Test1.bas (see **Figure 3**):

```
10 for n= 1 to 10
20 print n
30 next n
```

The LIST command causes the program to be output via the serial interface and appear in the terminal window. It can be started using RUN. Our first program generates an increasing series of numbers in the terminal window.

Commands such as LIST, RUN and NEW can be directly entered as text. However, you can also use the built-in functions in the Program menu. Program/List causes the listing to be copied to the editing window. Once it is in this window, it can be either be modified or stored on the hard disk.

Now it's time for a brief explanation of our sample program. In BASIC-52, each line has a line number (although this is no longer required in modern Basic systems). In line 10, the variable N is defined. This is then incremented from 1 to 10 in ten steps. FOR... TO... NEXT forms a loop that is executed until the value 10 is reached. This means that N receives the value '1' on the first pass through the loop, '2' on the second pass and so on, up to the value '10' on the final pass. In line 20, the current value is output via the serial interface and thereby written to the terminal screen.

A counting loop can also be built in assembler with a minimum of effort. However, the command PRINT is so complex that it would take a relatively elaborate assembler program to achieve the same function. To start with, BASIC-52 uses not only bytes (range 0–255), which can be directly understood by the microcontroller, but also real numbers, each is of which is composed of six bytes representing eight significant digits, a sign and an exponent. The PRINT command must convert these numbers into text and send this text character by character via the serial interface. This requires the initialisation of the interface, which is performed by the interpreter when it starts up.

This is not the place for a complete and boring explanation of all of the interpreter's keywords and commands. A brief command summary can be found in the Help file for the editor program. An extensive command summary, complete with a user's manual, can be found in the Free Downloads area of the *Elektor Electronics* website at www.elektor-electronics.co.uk. In the course of the following experiments, you will be introduced to an increasing number of new keywords and learn how to use them. We will start with very simple examples that are functionally

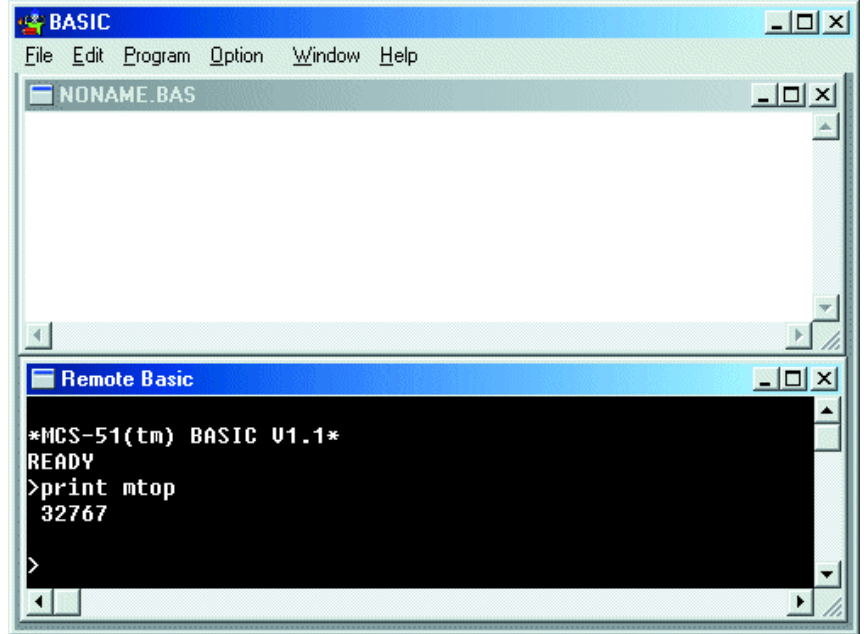


Figure 2. The terminal emulator program used in the course.

fully comparable to the assembler programs that you have already seen. It is particularly interesting to compare the speeds of these functionally equivalent programs. We start with Test2.bas:

```
10 FOR N=0 TO 255
20 PORT1=N
30 NEXT N
40 GOTO 10
```

Here we again use our well-known counting loop, this time with a range of 0–255, which covers the numerical range of a single byte. In BASIC-52, a number can be output to Port 1 just as easily as to the monitor screen. PORT1=N transfers the value of the variable N to the port. In the opposite direction, the states of all eight port lines can be read using the instruction N=PORT1.

As soon as you start the program

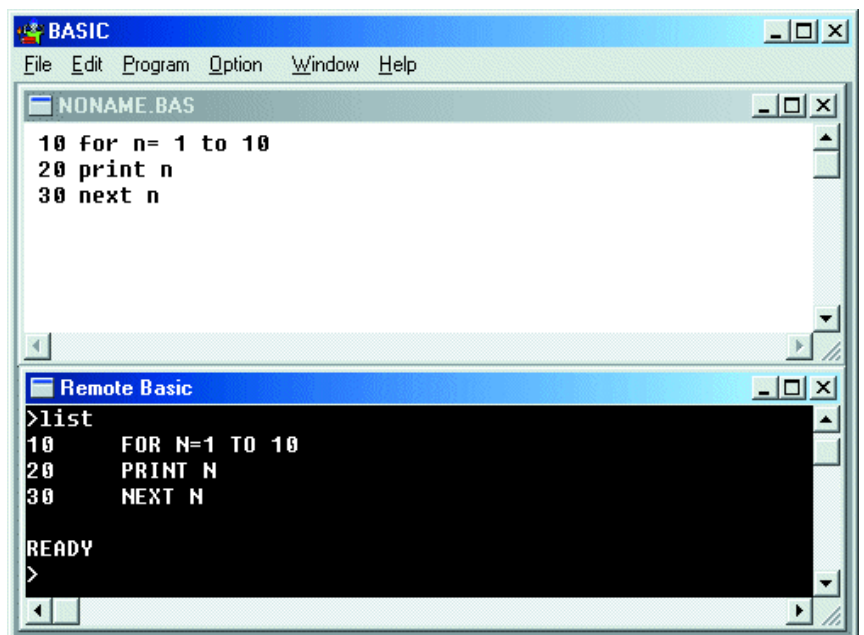


Figure 3. Entering a new program.

using RUN, you can see how fast it is. A square-wave signal with a frequency of around 200 Hz will appear at P1.0. At P1.7, the signal level will change approximately once per second, so it can be used to directly drive an LED. The comparable assembler program was somewhat faster, even with a supplementary delay loop. Basic programs run between 100 and 1000 times slower than equivalent assembler programs. This is no surprise, since BASIC-52 is an interpreter, which means that it is a rather complicated program that interprets and executes Basic source text word by word. The fact that this is a complicated task can already be seen from the size of the interpreter (8 kB). All of our previous sample assembler programs in the first two instalments of the course, taken together, amount to less than 100 bytes. In these examples, we only wrote very simple, linear programs. The interpreter, by contrast, must perform a very extensive list of tasks. At the run time of a Basic program, the keywords must be recognised and evaluated. Furthermore, the interpreter treats all numerical values as real numbers, which requires significantly more effort than processing bytes.

A frequency divider in Basic

Now it's time to give BASIC-52 a concrete assignment: an input must be constantly monitored for pulses. After exactly 10 pulses, an output toggles. After a total of 20 pulses, the output resumes its original state. The program thus effectively forms a frequency divider with a division factor of 20. Here we are using a program to carry out a task that is usually handled by a few ICs. The interesting question is to determine the limits within which this is possible.

In BASIC-52, there is no simple way to address the individual port bits. Consequently, we always have to read or write an entire port. Information about the state of an individual bit can be obtained by performing an AND operation on the total state of the port (masking). BASIC-52 has the .AND. byte operator for this purpose. The two full

MCS BASIC-52 VI.3

The most recent revision to the original version of the interpreter, which was made by H.-J. Böhling and D. Wulf, has already been presented in the February 2001 issue of *Elektor Electronics*. This version of the interpreter has also been successfully tested with the Flash Board. The program has special commands for programming and erasing EEPROMs. Several different Basic programs can be stored in a single EEPROM and started selectively. The individual steps, as listed below, have been tested using an 8-kB EEPROM:

1. Write a program.
2. Using XFER, program an EEPROM with this program.
3. Using XFER, load additional programs.
4. PROG returns the number of stored programs (e.g. 12).
5. Use ROM to switch to the EEPROM.
6. Use ROM2 to activate the second program.
7. Use PROG2 to make the first program in the EEPROM an autostart program.

Now whichever program is the first program in the EEPROM will start automatically after a reset, with no need for a terminal.

stops located before and after 'AND' indicate bit processing, in contrast to the logical linking of expressions (IF condition 1 AND condition 2 ...) In the following listing, this masking is applied in lines 110 and 130. Practically speaking, this command performs eight simultaneous AND operations between two different sets of eight bit states. The value of each resulting bit will be '1' only if both bits in the same position have the value '1'. This can be illustrated by the following examples:

```
10101010 .AND.
00000001 =
-----
00000000

11110001 .AND.
00000001 =
-----
00000001
```

As you can see, an .AND. operation with 00000001b = 01h = 1 yields only the values '0' and '1'. This means that quite deliberately, only bit 0 of the port is observed, with the behaviour of the remaining bits being masked out. Effectively, we have covered the port with a mask that allows only one bit to be seen, which is the bit that has been chosen as the input. The same thing can also be done in assembler, by the way, although in this case it would not be necessary, since it is possible to directly poll individual bits.

The listing of our frequency divider program (divide.bas) looks like this:

```
100 COUNT=0
110 INP=PORT1.AND.1
120 IF INP=1 THEN GOTO 110
130 INP=PORT1.AND.1
140 IF INP=0 THEN GOTO 130
150 COUNT=COUNT+1 :
    REM PRINT COUNT
160 IF COUNT=10 THEN
    GOTO 200
170 IF COUNT=20 THEN
    GOTO 250
180 GOTO 110
200 REM Output low
210 PORT1=253:REM P1.1 = 0
220 GOTO 110
250 REM Output high
260 PORT1=255:REM P1.1 = 1
270 COUNT=0
280 GOTO 110
```

Basic programs frequently contain many GOTO jumps, which reduces their readability, particularly in the case of large projects. This is often referred to as 'spaghetti code', which means code that is difficult to unravel and chaotic. The frequency divider program is a good example of this. However, it also shows that this style is quite reasonable for relatively small projects. Large programs, on the other hand, unconditionally require a better structure, which can for example be achieved using subroutines.

The program uses one variable (COUNT) as a pulse counter and a second variable (INP) for the input state of Port 1.0. At the beginning of the program, COUNT is set to zero. The program then passes through two loops in

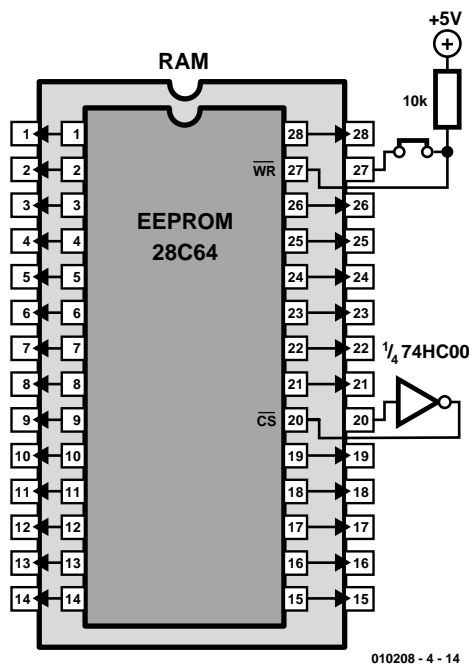


Figure 4. EEPROM wiring.

which the port state is continuously read and evaluated. As long as a High state is seen, the program remains within lines 110 and 120. Lines 130 and 140 handle the Low state in exactly the same manner. The loop is only closed when a new High state again occurs, causing a jump from line 180 back to line 110, where the whole process starts all over again. However, on the way to line 180 the value of the counter is incremented in line 150. Lines 160 and 170 test whether certain count states have been reached (10 and 20) and execute jumps to other parts of the program that switch an output state if this is the case. The code for a Low state starts at line 200, while that for a High state starts at line 250. Besides this, the counter is reset to zero at these locations.

The important feature of this program is the testing of conditions that either cause certain jumps to be executed or avoid making precisely these jumps. IF... THEN... ELSE... thus means that if the specified condition is satisfied, then jump to line such and such, and if it is not, then simply execute the next line. The condition is given in line 120 as $INP=1$. This means that here a comparison is made, with the result of the comparison being true if the input had a High state at exactly the same time as it was sampled in the previous line.

The program contains several comments, introduced by 'REM', that are intended to increase the readability of the program. A line may contain more than one instruction,

but the instructions must then be separated by a colon (:). Basic makes it easier to search for errors, since it is possible to 'momentarily' insert a PRINT instruction at any desired location in order to view whatever intermediate results may be present. In line 150, you can see the remnants of such a test. In order to see whether the actual counter was working, PRINT COUNT was added. This made it very easy to check whether input pulses were being correctly evaluated. Our first version of the sample program had an error that could no longer remain hidden once this instruction was added. The error arose because during the program development, we forgot to reset the counter to zero (line 270), with the result that the comparisons with specific counts (lines 160 and 170) led nowhere. Once everything was in order, we could 'comment out' the PRINT instruction using REM, which means that it is practically disabled. Of course, we could also delete it, but if exactly the same output instruction again proves to be useful in some future testing, we can simply delete 'REM' and thereby restore the old state.

The interesting question now is, what is the upper frequency limit that this counter can achieve? To answer this question, we connect a function generator to Port P1.0 and observe the P1.1 output using an oscilloscope. Our first impression is disappointing. Although simple LS TTL ICs can easily handle this task at an input frequency of 50 MHz, with this Basic counter the limit is already reached at 50 Hz. The program is thus around a million times slower. If the input frequency is raised above 50 Hz, input pulses are missed since at the time that the pulse should be sampled, the program has not yet returned to the line that samples the input. Still, the program can count pulses somewhat faster than a person can.

Comparing the program with a purely electronic solution or a manual solution is actually not particularly fair. The proper question is instead, in which cases can the Basic solution be the proper solution? There are potential applications wherever relatively slow results must be counted

and evaluated. For example, we could build a coil-winding machine for which we want the motor to be switched off after exactly 1650 turns. In this case, the Basic program proves to have the advantages that it can be quickly and easily modified and that the counting range is not limited by hardware.

Autostart for BASIC-52

Assembler programs have the advantage that they are autostart-capable with the Flash Board. This means that they can automatically start to work on their own each time the power is switched on. Unfortunately, this is not directly possible with BASIC-52, since the actual program is stored as data bytes in the RAM. When the power is switched off, everything in the RAM is lost. Even if we use a battery-backed RAM, the previous program would not automatically start. Instead, it would be fully deleted by the Basic interpreter, since BASIC-52 likes to start with a completely empty RAM.

However, from the very beginning the developers of the interpreter provided a function that allows a program to be held in EPROM in the memory region starting at 8000h. This capability can also be used with a supplementary EEPROM located in this memory region. There are two possible ways to implement this extension. The first is to use the system bus available at connector K8, while the second is to solder a second socket 'piggyback' on top of the RAM. Almost all of the leads can be connected one to one, with only the \overline{CS} (pin 20) and \overline{WR} (pin 27) leads requiring special treatment (see Figure 4).

The RAM occupies the address range 0-8FFFh. The address decoding is very simple, since it only amounts to connecting A15 to the \overline{CS} pin of the RAM. Whenever A15 is High, which means for memory accesses above 8000h, the RAM is inactive. In accordance with this simple arrangement, address signal A15 must be inverted before being applied to the \overline{CS} input of the supplementary EEPROM. In this way, the EEPROM is blocked in the lower memory range and occupies the range above 8000h.

From experience with EEPROMS,

it appears that there is a good chance of their contents being altered when power is switched on. This is presumably due to a brief pulse on the WR input. If this happens, the EEPROM is of little use, since the autostart will only work if stored program is free of errors. The proper solution is a jumper that can be used to block accidental programming of the EEPROM. **Figure 4** shows the full modifications for a 28C64 8-kB EEPROM, including an inverter and write protection.

The original version of BASIC-52 can program its own EPROMs, but this requires special hardware that is not present here. However, that does not present a problem, since the programming process for the EEPROM is so simple that it can be carried out using a small Basic program. The utility program listed below can do the job. It is added on top of an existing program (in this case, the program in lines 10–40) and started using GOTO 9000. It simply copies all program bytes from the RAM to the EEPROM, starting at

address 200h in the RAM and address 8011h in the EEPROM. Three additional bytes provide a problem-free autostart by supplying important information, such as the baud rate setting, to the operating system. For correct operation, it is also important to use MTOP to limit the upper boundary of the RAM.

```
10 FOR N=0 TO 255
20 PORT1=N
30 NEXT N
40 GOTO 10

9000 N=200h : E=8011H
9010 D=XBY(N)
9020 PRINT N,D
9025 XBY(E)=D
9030 N=N+1 : E=E+1
9040 FOR T=1 TO 10 :
      NEXT T
9050 IF D<>1 THEN
      GOTO 9010
9100 XBY(8000H)=32H
9105 FOR T=1 TO 10 :
      NEXT T
9110 XBY(8001H)=0FFH
9115 FOR T=1 TO 10 :
```

```
      NEXT T
9120 XBY(8002H)=0DCH
9125 FOR T=1 TO 10 :
      NEXT T
9130 XBY(8010H)=055H
mtop = 8191
goto 9000
```

As soon as the EEPROM has been programmed, you can use the ROM command to switch to the upper address region. The stored program can now be started in a perfectly normal manner using RUN and stopped using Ctrl-C, but it cannot be edited. You can use RAM to switch back to the normal memory region. In this manner, it is effectively possible to have two separate programs available in the system.

Each time the supply voltage is reapplied, the program currently stored in the EEPROM is automatically started. If this program is to be used permanently, the programming jumper should be removed in order to prevent accidental modifications to the program.

This concludes our brief introduction to working with BASIC-52. In the next installment, we will work with the C language. This will complete our comparison of the three most important programming languages.

(010208-4)

bahramelectronic.tk