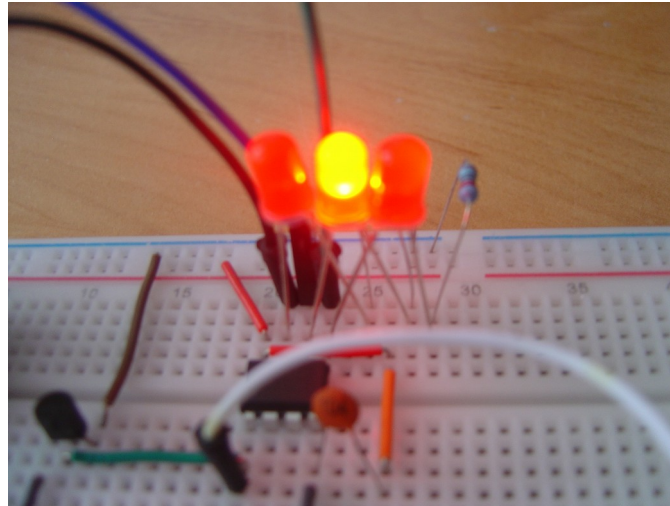


# Arduino Tailored to Our Needs

## *AVR Microcontroller Engineering with Pascal*

version 01/17/2026



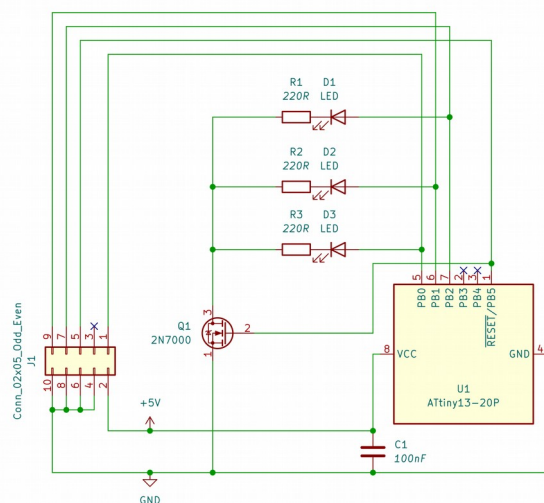
### 1. More LEDs

We will try to make the circuit built according to the previous course episode a bit more attractive. We will add two additional LEDs and modify the circuit's operation so that the LEDs blink sequentially. For our modification, besides the elements already gathered, we will need:

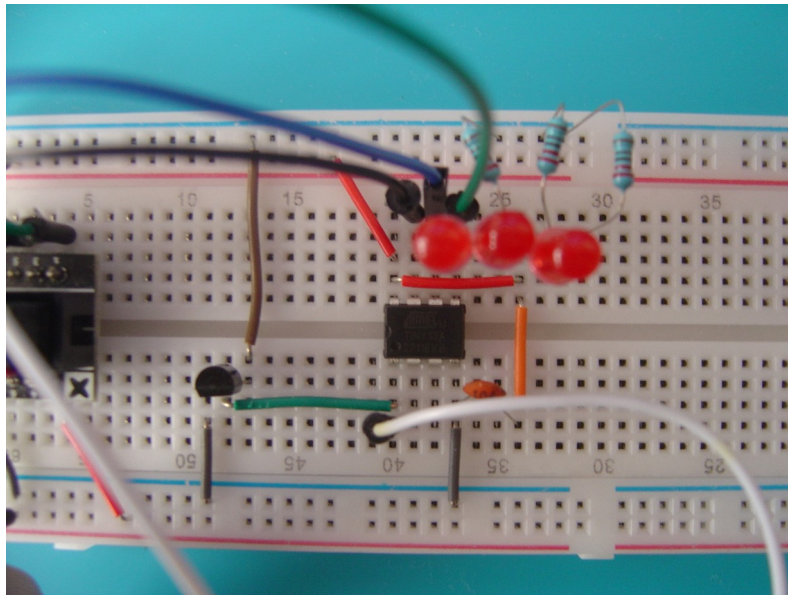
2 red LEDs	less than 1\$
2 resistors	less than 1\$

Prices are approximate, according to the cheapest offers on <http://www.ebay.com>.

### 2. Circuit



We will add two more red LEDs and two resistors to our circuit. We will connect the LEDs to the free lines (pins) of port B, i.e., PB1 (pin 6) and PB2 (pin 7) of the microcontroller, connecting resistors in series with the LEDs, as each LED requires its "own" resistor. I didn't have 200  $\Omega$  resistors on hand, so I used 220  $\Omega$  resistors, which work very well. Thanks to this connection, we can control each LED separately. We will connect the resistors to the ground line (-), then to the cathode of each diode, and connect the anodes of the diodes to the mentioned pins: PB0 (as in the previous course episode), PB1, and PB2. Driving these pins high (about 5 V) sequentially will cause the given LED to light up; setting a low state (no voltage) will cause the given LED to turn off.



Assembling the circuit on a breadboard requires no further explanation. Just pay attention not to connect to already occupied tracks on the breadboard, as it has become quite "dense" around the microcontroller.

### 3. Source Code

Programming our circuit will require a bit more code than before. The main problem we need to solve is programming the delay for turning the LEDs on and off so that it is regular and not too fast. Despite appearances, this is not a trivial issue if we want to program a specific delay time.

The ATtiny13 processor is clocked at 1.2 MHz, meaning it performs 1.2 million individual operations (clock cycles) per second<sup>1</sup>. Each instruction written in machine code (assembly) requires 1 to 2 (sometimes even more) clock cycles, and one clock cycle lasts 1/1.2 million seconds, i.e., 8.33 ns. Suppose we want to turn off the LED after a specific time, say 0.5 seconds. To do this, we would set a low state on pin PB0, where we connected the LED, which, like turning it on, will take 2 clock cycles. The ATtiny13 also supports the NOP (no operation) instruction, which takes 1 clock cycle and does nothing. It is useful for delaying the execution of subsequent instructions. But how to use it?

---

<sup>1</sup>This is default value. In practice, a clock frequency may differ from the catalog value by up to 10%.

FPC allows adding assembly block in code written in Pascal, and even writing entire procedures in assembly. Assembly blocks are fragments of assembly code inserted between the keywords *asm* and *end*. The considered problem could be roughly written as follows:

```
begin
  DDRB := DDRB or PB0;           // pin 0 of port B as output
  PORTB := PORTB or PB0;        // high state on PB0 -> turn on LED
  asm                             // start of assembly instructions
    nop                           // do nothing, 1 clock cycle
    //                             more nop instructions
  end;                             // end of assembly instructions
  PORTB := PORTB and not PB0;    // low state on port 0 -> turn off LED
end.
```

But how many NOP instructions should we use? Let's first think about how many such instructions we can use. The Flash memory of the ATtiny13 microcontroller has a capacity of 1024 bytes, so if we wanted to fill the entire available memory with NOP instructions, we could use a maximum of 1024 such instructions<sup>2</sup>. This amounts to 8533 ns (0.0008533 s), so we probably won't even notice the LED lighting up. This direction is a dead end.

Let's look at the problem from another perspective. How many clock cycles do we need to occupy 0.5 seconds? Exactly 600,000 (1.2 MHz \* 0.5). Due to the limitations of Flash memory capacity, the only solution is to execute a certain number of NOP instructions in a loop, repeating a specific number of times. In assembly, this can be achieved using labels and so-called jumps. Labels are conventional names ending with a colon, which after compilation only point to a location in the microcontroller's address space. Besides assembly, they are used in batch scripts and some programming languages, e.g., in BASIC<sup>3</sup>. There are several jump instructions, one of them is RJMP (*Relative Jump*). Here's an example:

```
asm                             // start of assembly instructions
  rjmp Delay                     // unconditional jump to label Delay
Return:
  //[...]                         some code, may contain more jumps
Delay:
  nop
  nop
  rjmp Return                     // unconditional jump to label Return
end;                             // end of assembly instructions
```

First, there is a jump to the label Delay and execution of two NOP instructions, and from there a jump to the label Return and execution of further code. However, these are unconditional jumps, and we would like to execute the NOP instructions a specific number of times. So we need to add a variable that will serve as a counter, assign it an initial value, and after executing the NOP instruction, decrement it by 1. If we wanted to execute the NOP instruction 600 thousand times, our algorithm could be as follows:

---

<sup>2</sup> In practice, of course, less, as the available memory is also limited by the remaining code added by the FPC compiler, which also occupies the microcontroller's flash memory. Here, it's more about presenting the scale of the issue.

<sup>3</sup> In other high-level languages, e.g., Pascal or C, labels and jump instructions (*goto*) can be used, but for the clarity of the source code, it is usually discouraged. In practice, after compilation, equivalents of *goto* instructions are common in machine code.

- 1) assign the variable a value of 600,000,
- 2) execute the NOP instruction,
- 3) decrement the variable's value by 1,
- 4) if the variable's value > 0, return to point 2.

AVR microcontrollers have 32 general-purpose registers, named R0 to R31, suitable for storing variables, e.g., a counter. These are 8-bit registers, so the numbers in them must fit within 0-255 (i.e., a byte). 255 is too small a number for our needs. Eight of these registers allow storing 16-bit numbers (0-65,535). Such numbers are stored in two bytes in register pairs: R25:R24, R27:R26, R29:R28, and R31:R30. We can therefore modify the algorithm as follows:

- 1) assign the variable a value of 60,000, e.g., in registers R25:R24,
- 2) execute the NOP instruction 10 times,
- 3) decrement the variable's value by 1,
- 4) if the variable's value > 0, return to point 2.

How will we know that the variable has reached 0? AVR microcontrollers have an additional interesting 8-bit register called the Status Register. One of its bits, called Z (from *Zero*), is set if a given instruction that is a mathematical or logical operation gave a result of 0. This property is used by another instruction we need, BRNE (*Branch if not Equal*), which performs a jump to the specified label if the Z bit is not set<sup>4</sup>. We also need instructions for assigning a constant to a register and decrementing the register value by 1. These are LDI (*Load Immediate*) and SBIW (*Subtract Immediate Word*; *Word* means a 2-byte number). Our code could look as follows:

```
asm
    ldi R25, 234 //Hi(60 000), insert "high" byte into register R25
    ldi R24, 96  //Lo(60 000), insert "low" byte into register R24
Delay:
    sbiw R24,1 // decrement the 16-bit counter by 1, will execute 60,000
times
    nop
    // next 9 NOP instructions
    brne Delay // conditional jump to label Delay
    nop
end;
```

After running this code, the compiler will report an error about an undeclared label *Delay*, which means that before the *begin-end* block, we need to declare it (i.e., add *label Delay*;). However, it turns out that the delay loop lasts too long. The reason is that each instruction takes microcontroller time. SBIW takes 2 clock cycles, BRNE - 2 cycles if there is a jump to the label, 1 if not. Additionally, 2 cycles are taken by each LDI instruction. We can try to optimize our code by getting rid of excess NOP instructions:

```
label
Delay;
begin
asm
    ldi R25, 234 // 2 clock cycles
    ldi R24, 96  // 2 clock cycles
```

---

<sup>4</sup> Yes, the name of the instruction is misleading, it could be called e.g. *Branch if not Zero*.

```

Delay:
  sbiw R24,1    // 2 cycles
  nop          // 1 cycle
  nop          // 1 cycle
  nop          // 1 cycle
  nop          // 1 cycle
  nop          // 1 cycle
  nop          // 1 cycle
  nop          // 1 cycle
  brne Delay   // 2 cycles if jump to label, 1 cycle if not
  nop          // 1 cycle
end;
end;

```

Excess in terms of execution time remained the two LDI instructions, but since we are not building a clock, we can afford such inaccuracy. To even out the irregularity of the BRNE instruction, we can add one NOP instruction at the end, but since our code is not ideal, it is not necessary. The error is 3-4 clock cycles, compared to 600,000, which is almost nothing.

We have prepared a sufficiently good half-second delay loop. Let's return to the issue of blinking three LEDs sequentially. The algorithm could look like this:

- 1) Set pins PB0, PB1, and PB2 as outputs,
- 2) Set high state on pin PB0 (turn on LED 1),
- 3) Run the delay loop,
- 4) Set low state on pin PB0 (turn off LED 1),
- 5) Set high state on pin PB1 (turn on LED 2),
- 6) Run the delay loop,
- 7) Set low state on pin PB1 (turn off LED 2),
- 8) Set high state on pin PB2 (turn on LED 3),
- 9) Run the delay loop,
- 10) Set low state on pin PB2 (turn off LED 3),
- 11) Return to point 2.

To avoid copying unnecessary code, it's worth enclosing our delay loop in a procedure that can later be called from any place in the program, naming it e.g., *HalfSecondDelay*. Based on our algorithm, we can write the following code:

```

program attiny13_blink;

{$IFDEF attiny13}
  {$Fatal Invalid controller type, expected: attiny13}
{$ENDIF}

procedure HalfSecondDelay;
label
  Delay;
begin
  asm
    ldi R25, 234 // 2 clock cycles
    ldi R24, 96  // 2 clock cycles
  Delay:
    sbiw R24,1   // 2 cycles
    nop         // 1 cycle
    nop         // 1 cycle
  endasm
end;

```

```

        nop          // 1 cycle
        nop          // 1 cycle
        nop          // 1 cycle
        nop          // 1 cycle
        brne Delay  // 2 cycles if jump to label, 1 cycle if not
        nop          // 1 cycle
    end;
end;

const
    PB0 = %00000001;
    PB1 = %00000010;
    PB2 = %00000100;

begin
    //direction register
    DDRB := DDRB or PB0; // pin 0 of port B as output
    DDRB := DDRB or PB1; // pin 1 of port B as output
    DDRB := DDRB or PB2; // pin 2 of port B as output

    while true do // infinite loop
    begin
        PORTB := PORTB or PB0; // high state on pin 0 -> turn on LED
        HalfSecondDelay; // waiting
        PORTB := PORTB and not PB0; // low state on pin 0 -> turn off LED

        PORTB := PORTB or PB1; // high state on pin 1 -> turn on LED
        HalfSecondDelay; // waiting
        PORTB := PORTB and not PB1; // low state on pin 1 -> turn off LED

        PORTB := PORTB or PB2; // high state on pin 2 -> turn on LED
        HalfSecondDelay; // waiting
        PORTB := PORTB and not PB2; // low state on pin 2 -> turn off LED
    end;
end.

```

The *while-do* loop used in the code requires some explanation. Loops of this type will execute as long as the expression between *while* and *do* is true. Our code doesn't use any comparisons; we hardcode the result of the expression to *true*, so our loop will execute indefinitely. In fact, it will execute indefinitely as long as the microcontroller has power. After compiling the code and loading it into Flash memory, the microcontroller should do what we want, i.e., flash three LEDs in sequence.

Have fun!

Andrzej Karwowski

*Nobody is perfect* – found an error or inaccuracy, have a question or suggestion – write to [ackarwow@gmail.com](mailto:ackarwow@gmail.com)

Thanks to user *sp3ots* from the forum [www.elektroda.pl](http://www.elektroda.pl) for pointing out the inconsistencies in the description of the LED connections in the circuit.