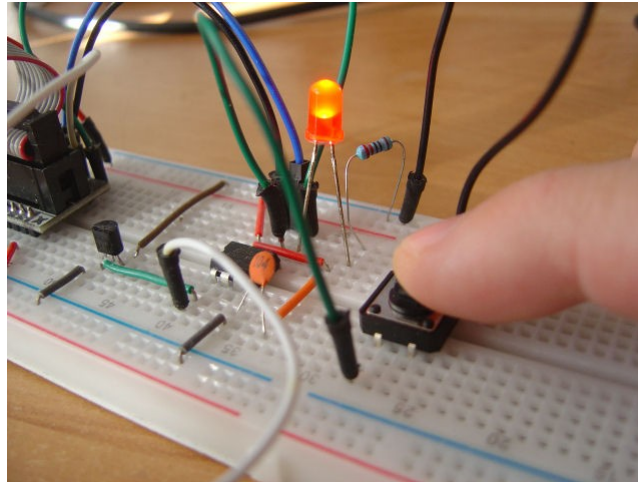


Arduino Tailored to Our Needs

AVR Microcontroller Engineering with Pascal, Lesson 3

version 02/13/2026



1. "Push", or the Button

In this lesson, we will introduce another modification to our circuit. This time, we will use a button to turn the LED on and off. The task itself is trivial, but as usual, the surrounding circumstances are more interesting. Although various buttons are available on the market, we will use a monostable switch button of the Tact Switch type here. So, to our list of needs, we will add:

1 Tact Switch type button

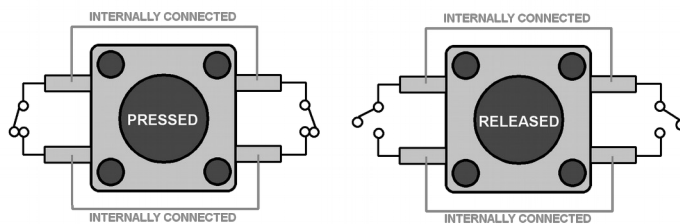
under 0.5\$

The price is approximate, according to the cheapest offers on the popular online marketplace¹.

The button is a special type of switch, distinguished by its construction, and above all the way it is activated – pressing (not, e.g., moving a lever). A monostable button is a type of button that can change its state for the duration of the press, after which it returns to the previous state. In the case of our button, the default state is OFF (off, no current flows through the button), which changes to ON (on, current flows) for the duration of the press. There are also bistable buttons, i.e., permanently changing state until the next press. Such buttons are also suitable for our task, as we will examine the button's state, not the way it changes.



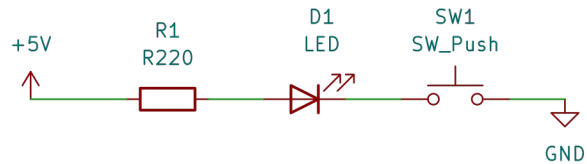
Button and its symbol



Way the monostable button works

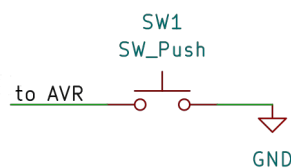
¹ <http://www.ebay.com/>

Our button has 4 legs (pins), which can be a bit misleading, e.g., suggesting that it handles two separate channels. In reality, opposite legs are internally connected, and this construction is dictated by stability considerations for mounting the button. Inside, there is also a spring that causes a return to the default state after releasing the button. I chose a button with dimensions 12×12 mm, whose legs (with some difficulty) can be placed on the breadboard². Generally, switches (and thus buttons) of this type are denoted by the abbreviation SPST-NO³.



To prepare a circuit implementing our issue, no microcontroller is needed. It is enough to connect the LED appropriately through a resistor on one side to power, on the other through the button to ground. Then pressing the button will close the circuit and the LED will light up. However, our goal is to use the microcontroller for this task. This can be achieved by connecting the button to one of the microcontroller's legs, which we will set as input, and the LED to another leg set as output. Let's think now about how to implement this in practice.

AVR microcontrollers handle determining high and low states on a given input-output line well, doing it in a simple way: voltage below half the supply voltage is considered low state (digital 0), belonging to the second half – high state (digital 1). Therefore, using the button, we should cause a change in voltage on the given input line from high to low state and back. Intuition suggests that the appropriate leg can be connected either to the supply voltage, which theoretically should force a high state on the pin, or to ground, which should result in a low state on that leg.

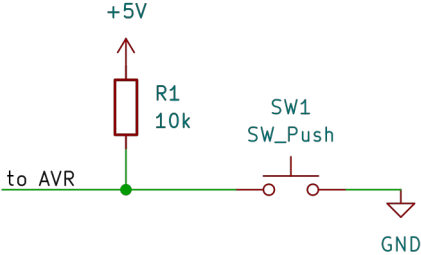


Let's look at the second possibility. Pressing the button will force a low state on the input line, but what will be the state of the line after releasing the button? It will be unstable and, like an unconnected wire, will act like an antenna constantly changing voltage between low and high states. Such behavior is unacceptable and something must be done to stabilize the line in a high state when the button is released.

2 It is difficult to find information about the manufacturer, but it is certainly a button manufactured in the People's Republic of China (PRC). According to the seller's note, the maximum operating voltage of the switch is 12 V and current 50 mA.

3 This name refers to the switch's operation. The first two letters refer to the number of circuits the switch handles. It can be SP (single pole – meaning one circuit), DP (double pole – handles two circuits), 3P and 4P. The next two letters refer to the number of contact positions, e.g., ST (single throw – connects the circuit only in one position), DT (double throw – connects the circuit in two positions, e.g., as a switch from mains power to battery). The last abbreviation is NO (circuit normally open) or NC (circuit normally closed).

The simplest solution is to connect the microcontroller's input line leg to power, but this would end catastrophically after pressing the button. Directly connecting the supply voltage to ground would inevitably cause a short circuit in the circuit. But there's a solution: a pull-up resistor to power.

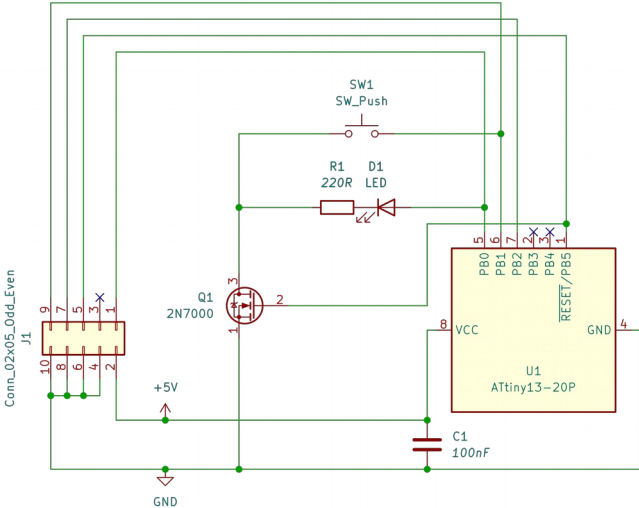


A pull-up resistor is a resistor with relatively high resistance (from 10 kΩ to 100 kΩ), which, when the button is not pressed, pulls the input line up to power, in other words, defines the default voltage value on the leg. After pressing the button, a connection to ground with very low resistance is created compared to the connection to power through the resistor, so only a small current will flow through that resistor. At the same time, the voltage on the microcontroller's leg will be close to 0 V. The solution with the resistor looks promising.

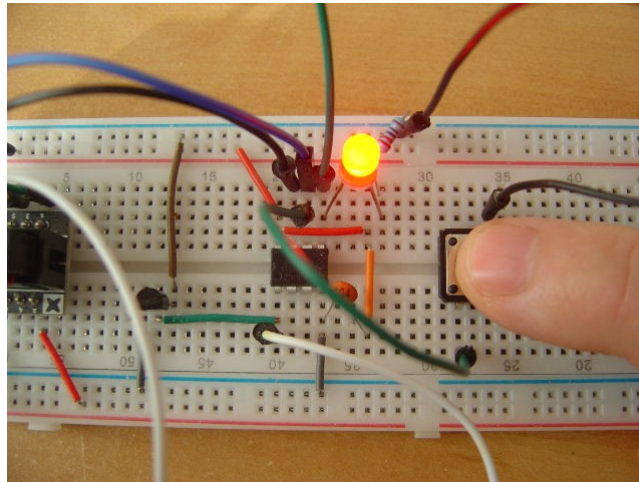
There is more good news: in AVR microcontrollers, each input-output line has an additional built-in pull-up resistor, so there is no need to add an external resistor to the circuit. You just need to activate the internal pull-up resistor on the given pin, and by pressing the button on that line, a low state will appear, which after releasing the button will return to the high level. Activating the internal pull-up resistor is now a matter of code.

2. Circuit

After this lengthy introduction, let's return to our circuit. We will certainly need to dismantle part of the circuit, as we won't need three LEDs and three resistors; one LED and one resistor will suffice. So we will leave the LED connected to pin 5 (PB0) of the microcontroller and the 220 Ω resistor connecting it to the ground line. We will add the button connected to pin 6 (PB1) in place of one of the removed LEDs.



As a reminder: during microcontroller programming using the USBasp programmer, the ground line of the LED and button is disconnected, so they cannot interfere with the programmer's operation. However, during normal operation of the microcontroller, the LED connected to its pin 5 should light up after pressing the button connected to PB1 and turn off after releasing the button.



3. Code

Our source code should allow programming the microcontroller in such a way that it can perform two tasks: 1) programmatically detect pressing and releasing the button; 2) react to these events by turning the LED on and off.

To achieve the first task, besides the already gathered elements, we can use part of the code from the previous episode, i.e., keep the definitions of constants PB0 and PB1. The differences will appear in setting the direction register: to define pin PB0 as output, we can use the OR operation setting the corresponding bit, while to set pin PB1 as input, we need to clear its bit, using e.g., AND NOT operation. Additionally, we should enable the pull-up resistor on PB1. To do this, we need to refer to the PORTB register setting the PB1 bit⁴.

The second task we will implement in an infinite loop. In each iteration of the loop, we should check if the bit corresponding to pin 6 of the microcontroller (PB1) is cleared, i.e., if there is a low state on that pin. This can be done by checking the result of the AND operation on the contents of the input register. Such a register in our microcontroller is PINB. The result of the bit conjunction will be 1 if bit PB1 is in high state, 0 – if it is in low state. Depending on this result, we should light or turn off the LED on PB0.

Our code could look as follows:

```
program button;
```

⁴ Using the PORTB register to enable the pull-up resistor of a pin set in input mode seems a bit confusing, after all, the same register is used to handle pins in output mode, setting high state (turning on the bit) or low (turning off the bit) on them. One can assume that the designers of AVR microcontrollers aimed to save space in the microcontroller by using one register for two purposes.

```

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

const
  PB0 = %00000001; // pin PB0 = microcontroller leg 5
  PB1 = %00000010; // pin PB1 = microcontroller leg 6

begin
  // direction register
  DDRB := DDRB or PB0;           // pin PB0 as output
  DDRB := DDRB and not PB1;     // pin PB1 as input

  PORTB := PORTB or PB1;        // enable pull-up resistor on PB1

  while true do                // infinite loop
  begin
    if (PINB and PB1) = 0 then // if button pressed (bit=0)
      PORTB:=PORTB or PB0      // turn on LED
    else
      PORTB:=PORTB and not PB0; // turn off LED
    end;
  end.
end.

```

It is worth noting that by default, all microcontroller pins are set as inputs, i.e., all bits of the DDRB register are set to 0. At the same time, all bits of the PORTB register also have a default value of 0, which means that no internal pull-up resistor is enabled, i.e., the inputs are not pulled up to power. It is worth programming them to enable to protect against unstable voltage on the input pins, especially when we examine their state.

4. State Change

Basically, our task has been completed. To not end the episode so quickly, we can diversify it a bit and adapt it to more "real-life" applications. Often a button is needed to permanently change the state of some device. For example, when turning on a TV, we want it to remain on even after releasing the button, and turn off only after pressing it again. So let's modify our task as follows: the LED should light up after pressing the button and turn off after pressing it again⁵.

We can leave the circuit unchanged, but we need to change the code to detect the button state change, not just the fact that the button is pressed or not. Additionally, the program must know if the LED is currently on or off. For these purposes, we can use two Boolean variables, which we'll name ButtonPressed and LedOn, which can take two values: true and false. The ButtonPressed variable will remember the current button state and will help determine if the button has just been pressed. The LedOn variable will remember the LED state so that it can be easily changed to the opposite using negation (NOT). Our code may look as follows:

```
program button2;
```

⁵ That is, in such a way that the program reacts to pressing the button as if it were a bistable button. Naturally, our further code will not be suitable for bistable buttons.

```

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

const
  PB0 = %00000001; // pin PB0 = microcontroller leg 5
  PB1 = %00000010; // pin PB1 = microcontroller leg 6

var
  ButtonPressed: boolean;
  LedOn: boolean;

begin
  // direction register
  DDRB := DDRB or PB0;           // pin PB0 as output
  DDRB := DDRB and not PB1;     // pin PB1 as input

  PORTB := PORTB or PB1;       // enable pull-up resistor on PB1

  ButtonPressed := false;
  LedOn := false;

  while true do                // infinite loop
  begin
    if (PINB and PB1) = 0 then // if button pressed (bit=0)
    begin
      if ButtonPressed = false then
      begin
        ButtonPressed := true;

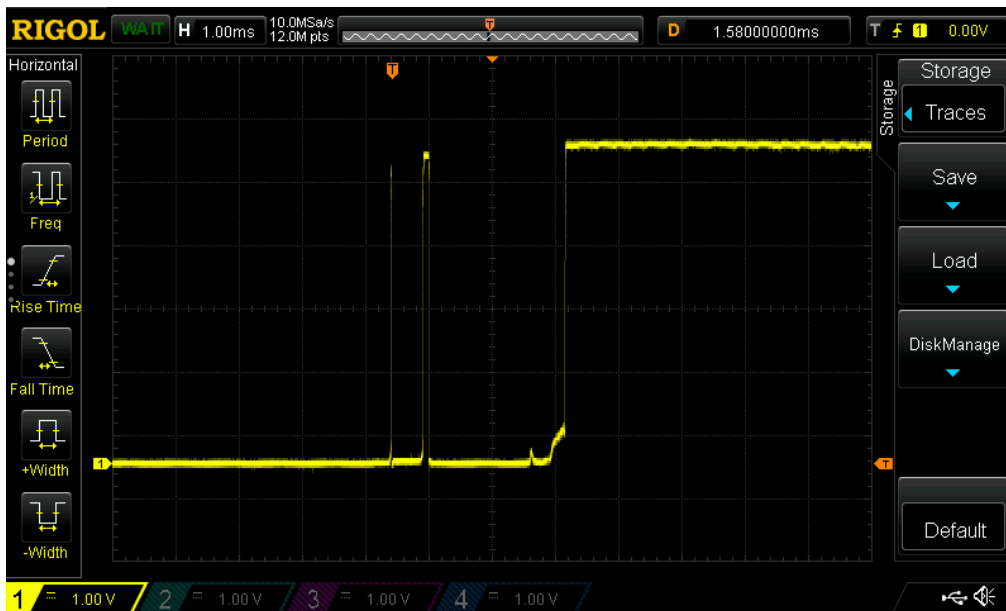
        LedOn := not LedOn; // set flag to opposite
        if LedOn then
          PORTB := PORTB or PB0 // turn on LED
        else
          PORTB := PORTB and not PB0; // turn off LED
        end
      end
    end
  else
    ButtonPressed := false;
  end;
end.

```

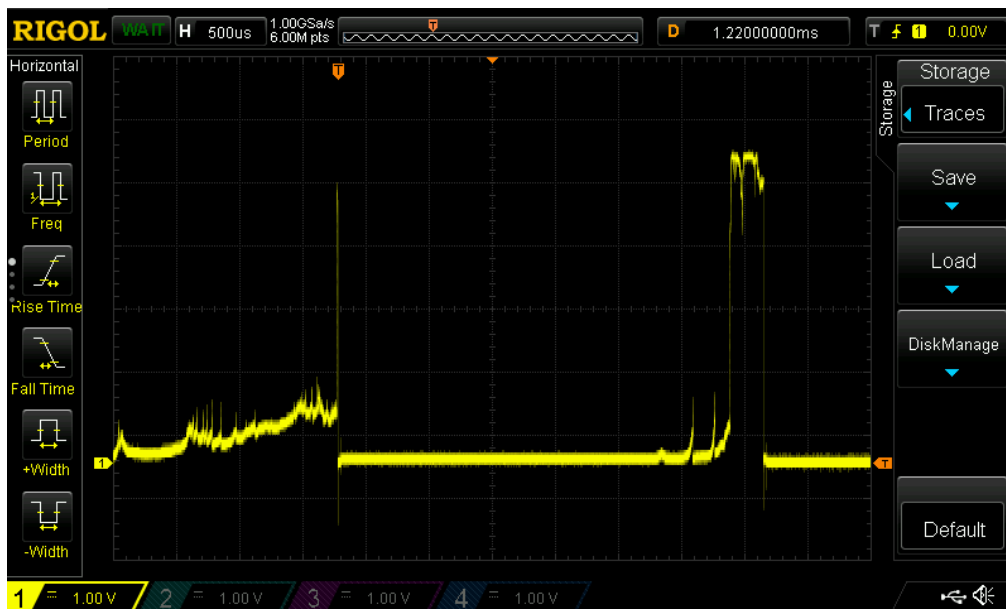
Most changes in the code are focused on the event of changing the button state to pressed by checking the value of the *ButtonPressed* variable. After changing the value from *false* to *true*, we change the value of the *LedOn* variable to the opposite and, depending on that, turn on or off the LED.

Essentially, our change should work. Well... Actually... almost works. After a short play with the button, at some point the LED starts reacting strangely, instead of lighting up – it turns off, seemingly at random moments. What's going on? It looks like we need to check what actually happens during pressing and releasing the button, reaching for an oscilloscope⁶.

⁶ Theoretically, to illustrate the problem, a logic analyzer and a good program analyzing voltage changes over time, e.g., Saleae Logic, would suffice. This is a tool much cheaper than an oscilloscope.



In most cases, the oscillogram looked completely fine, and the voltage level fluctuations gave a rectangular waveform consistent with button presses. But a few times, strange spikes or unstable, fluctuating voltage levels appeared.



The instability time on my oscillograms ranged from about 3 to 5 ms. Their cause is prosaic: the contact surfaces of the button are not perfectly smooth, and for a short moment, the voltage on them fluctuates between high and low states, but our code cannot catch this and interprets every low state as a button press. A natural solution is to check if after a while the button state will be the same and, depending on that, make further decisions⁷.

So, we need to return to our *HalfSecondDelay* procedure, modifying it so that it can implement various delays, starting from a few ms. To use this procedure without changing its

⁷ Another way – used before the era of microcontrollers – was to add a smoothing capacitor. Nowadays, however, such solutions are considered expensive (after all, a capacitor costs something, even if little) and it's cheaper to write a few extra lines of code to solve the problem.

logic, we need to make a compromise. Our counter is 16-bit, so its maximum value is 65535, which is a value slightly larger than the value for half a second (60000), as we know from the previous part of the course. We can therefore assume that the upper range of sufficiently correct operation of our procedure is 500 ms. What will be the lower range? 1 ms is for our counter $120000 / 1000$, i.e., 120, and the counter is decremented by 1 every about 10 microcontroller cycles. Remember that before the actual loop, we perform a multiplication operation to get the initial counter value for the given number of milliseconds. This operation will also take the microcontroller's time, depending on what assembler code Free Pascal Compiler generates⁸. It can be estimated that it will be several dozen cycles, and adding some margin - close to 1 ms. This means that our procedure will work in the range of 1-500 ms, but its operation time may be longer even by 1 ms⁹. Our new delaying procedure can be written as:

```

procedure DelayMs(Value: UInt16);
var
    wCounter: UInt16;
    bHi, bLo: byte;
label
    Delay;
begin
    wCounter:= Value * 120; // several dozen clock cycles
    bHi:=Hi(wCounter);      // several clock cycles
    bLo:= Lo(wCounter);
    asm
        ldd R25, bHi // 2 clock cycles
        ldd R24, bLo // 2 clock cycles
    Delay:
        sbiw R24,1    // 2 cycles; decrement 16-bit counter by 1
        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['r24', 'r25'];
end;

```

Certainly, this procedure is not very accurate, but for our purposes, it doesn't have to be. If we conservatively set the input parameter to 10 ms (even if in reality it will be 11 or 12 ms), it's unlikely that any human could press the button a second time before that time. So, we can modify our program's main loop to re-check if the button is still pressed after 10 ms:

```

while true do                                // infinite loop
begin
    if (PINB and PB1) = 0 then                // if button pressed (bit=0)
    begin
        DelayMs(10);                            // conservatively, 10 ms delay
        if (PINB and PB1) = 0 then          // re-check button
        begin

```

⁸ The issue is not trivial, because the ATtiny13 does not have built-in hardware support for multiplication operations (MUL), so the solution must be based, for example, on bit shifts.

⁹ Of course, one could improve our procedure by subtracting from the counter the number of clock cycles that the multiplication operation itself takes, but for our current needs we can leave this part of the code unchanged for now.

```

    if ButtonPressed = false then
    begin
        ButtonPressed := true;

        LedOn := not LedOn;           // set flag to opposite
        if LedOn then
            PORTB := PORTB or PB0     // turn on LED
        else
            PORTB := PORTB and not PB0; // turn off LED
        end;
    end;
end
else
    ButtonPressed := false;
end;

```

Yes, now we've eliminated the button contact bounce problem. And we could end the episode here, if not for the fact that our code is not very readable. There's a lot of bit operations in it, and bit masks too. Maybe we can simplify it somehow?

5. Code Simplification and First Library

Well, with the *DelayMs* procedure, little can be done besides adding even more assembler to refine the delay resulting from multiplication (so far, we only estimate it). We'll leave it unchanged for now. But what to replace bit operations with? Well, we can use custom data types corresponding to byte size, but giving easier access to bits:

```

type
    TBinary = 0..1; // allowed values are 0 and 1

    TBitRec = bitpacked record // declaration of byte as record with fields
representing bits
        Bit0: TBinary;
        Bit1: TBinary;
        Bit2: TBinary;
        Bit3: TBinary;
        Bit4: TBinary;
        Bit5: TBinary;
        Bit6: TBinary;
        Bit7: TBinary;
    end;

```

In this way, we've defined two custom types: *TBinary*, which takes values 0 or 1, and *TBitRec* – a record (structure) consisting of eight fields of type *TBinary*, with names corresponding to bits from 0 to 7¹⁰. *TBitRec* is simply a byte consisting of fields – bits. Now we can use it in variable declarations where we need access to bits. Free Pascal allows casting variables to registers using the *absolute* keyword, so we'll declare variables *DDRBits*, *PORTBBits*, and *PINBBits* to have the same memory addresses as the registers *DDRB*, *PORTB*, and *PINB* respectively. This way, we'll have simple access to the registers' bits we use:

```

var
    ButtonPressed: boolean;

```

¹⁰ The names of the fields of this record type can be arbitrary, but there must be exactly 8 of them and they must all be of type *TBinary*.

```

LedOn: boolean;

DDRBBits: TBitRec absolute DDRB;
PORTBBits: TBitRec absolute PORTB;
PINBBits: TBitRec absolute PINB;
begin
  // direction register
  DDRBBits.Bit0 := 1;           // PB0 as output
  DDRBBits.Bit1 := 0;           // PB1 as input

  PORTBBits.Bit1 := 1;         // enable pull-up resistor on PB1

  ButtonPressed := false;
  LedOn := false;

  while true do
  begin
    if PINBBits.Bit1 = 0 then // if button pressed (bit=0)
    begin
      DelayMs(10);           // conservatively, 10 ms delay
      if PINBBits.Bit1 = 0 then // re-check button
      begin
        if ButtonPressed = false then
        begin
          ButtonPressed := true;

          LedOn := not LedOn; // set flag to opposite
          if LedOn then
            PORTBBits.Bit0 := 1 // turn on LED
          else
            PORTBBits.Bit0 := 0 // turn off LED
          end;
        end
      end
    end
  else
    ButtonPressed := false;
  end;
end.

```

As you can see, instead of more complicated bit operations on whole bytes, we can use direct comparison of a single bit or assigning it a value of 0 or 1. Such code is much easier to read, although there are no functional differences compared to the previous one.

Finally, let's look at the code from a "further" perspective. If we'll be writing more programs for ATtiny13, we'll certainly use the *TBitRec* type definition multiple times due to its convenience. We can also assume that in the future, the *DelayMs* procedure will come in handy. It's worth extracting their code to a new unit, thus creating the first library (for now containing one unit). We can name the new unit *attiny13_basics.pas*, as it will contain basic procedures and types universal enough to be useful in other programs. The unit must have an interface section, where we'll declare our types and procedure, and an implementation, where we'll place the procedure's code:

```

unit attiny13_basics;

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

```

```

interface

type
  TBinary = 0..1; // allowed values are 0 and 1

  TBitRec = bitpacked record // declaration of byte...
    //[...] field declaration here
  end;

procedure DelayMs(Value: UInt16);

implementation

procedure DelayMs(Value: UInt16);
var
  //[...] variable list here
label
  //[...] label list here
begin
  //[...] procedure code here
end;

end.

```

Save our new unit in the same directory where we saved the program file¹¹. To use it in the program, we need to add a uses section with the list of units used in the program. Now the structure of our program will be as follows:

```

program button4;

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

uses
  attiny13_basics; // list of used units

var
  ButtonPressed: boolean;
  LedOn: boolean;

  // using types from the unit
  DDRBBits: TBitRec absolute DDRB;
  PORTBBits: TBitRec absolute PORTB;
  PINBBits: TBitRec absolute PINB;
begin
  //[...] initialization of register bits and variables here
  while true do // infinite loop
  begin
    if PINBBits.Bit1 = 0 then
    begin
      DelayMs(10); // using procedure from the unit
      //[...] further code
    end
    //[...] further code
  end;
end.

```

¹¹ You can also save the unit file in another directory, then in the program you would need to indicate the relative path to the unit or save the unit to the AVR Pascal *lib* directory. However, I think that when writing the first units, it's not worth overcomplicating things...

In this way, we've reached the end of the third episode of the course. The episode has not exhausted the topic, of course, as I've presented only one of the possible ways of interfacing the circuit with a button using a pull-up resistor. Equally well (?) one could build the circuit "in reverse", using an external pull-down resistor¹² and connecting the button to the power line. However, this would require some rebuilding of our circuit. The program code also has potential for optimization.

Have fun!

Andrzej Karwowski

Nobody is perfect – found an error or inaccuracy, have a question or suggestion – write to ackarwow@gmail.com

Acknowledgments

Thanks to user sp3ots from the forum www.elektroda.pl for inspiration to prepare this lesson.

¹² ATtiny13 does not have an internal pull-down resistor.