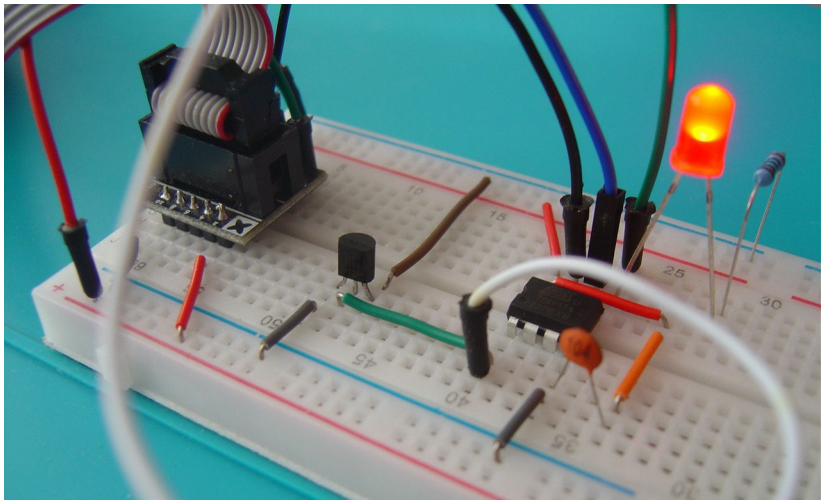
Arduino Tailored to Our Needs

AVR Microcontroller Engineering with Pascal

version 12/26/2025



1. Our Own Arduino Replacement

Many people start their adventure with electronics by buying an Arduino. That's not a bad idea, but if the first experiment is to light up an LED, I suggest an alternative and cheaper solution. Instead of buying a fairly expensive original Arduino module (costs about 20$, Chinese clones are cheaper), we can build the circuit ourselves, learning about the operation and programming of microcontrollers along the way. We will use a microcontroller from the AVR family, produced by Microchip (formerly Atmel). More expensive AVR microcontrollers are used in Arduino (ATmega variants), but we will use the ATtiny13, a smaller and cheaper microcontroller that is completely sufficient for our needs. To start, we will build a circuit on a breadboard, using which we will program the ATtiny13 so that a red LED lights up. This requires practically two lines of code, but first let's prepare the necessary electronic components and tools.

We will need:

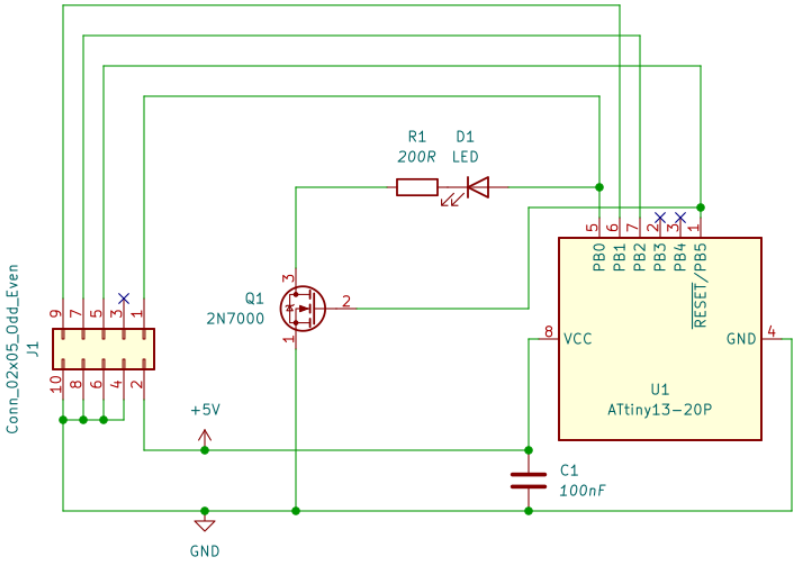
1 breadboard	from 2$
1 ATtiny13 microcontroller	from 1$
1 USBasp programmer with ribbon cable	from 3$
1 IDC10 connector for breadboard[1]	from 1$
1 red LED	under 0.5$
1 resistor	under 0.5$
1 ceramic capacitor	under 0.5$
1 2N7000 field-effect transistor	under 0.5$
set of wires and jumpers for breadboards (you can save by pulling from e.g. a network cable, but I don't recommend)	a few $

1 The IDC10 standard is the most popular in cheap USBasp programmers, but the 6-pin standard can also be found.

multimeter, can be the cheapest (it will come in handy sooner or later)	from 6$
code editor and compiler for AVR microcontrollers	0$

The listed components and tools are reusable by design and can be used in other projects later. Prices are approximate, based on the cheapest offers on https://www.ebay.com/.

2. Principle of Operation



The principle of operation of our circuit is extremely simple: the microcontroller (U1 in the schematic) sets a high state on the pin connected to the LED (D1), causing the diode to light up. But for the microcontroller to work properly, it must receive power (3.3 – 5V). Moreover, it's not a computer with its own operating system like DOS (does anyone remember it?), Windows, or Linux. So we have to program it ourselves using the microcontroller's registers and ports. To program the microcontroller, we will use the USBasp programmer, which will also serve as the power source for our circuit.

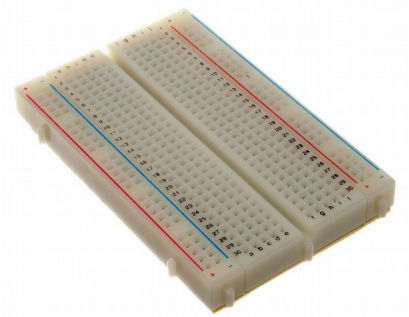
In the schematic, we have two ground lines: the main one coming straight from the programmer connector (J1) and the second one running from the transistor drain (Q1) through the resistor (R1) to the LED cathode. Why this complication? The point is that both the diode and the programmer use pin 5 of the microcontroller, and interference may occur during programming. The transistor acts as a switch here, utilizing the fact that during normal operation of the microcontroller, pin 1 (RESET) has 5V applied, while during programming, the programmer connects this pin to ground. In other words, during programming, the diode's ground line is cut off, and the diode does not negatively affect the programmer's operation[2].

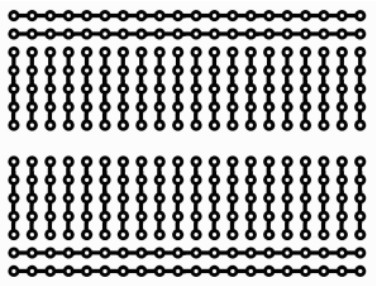
2 The use of a 2N7000 transistor allows the RESET line to be used to control the diode without changing the fuse bits. Thanks to the transistor's high input impedance, the USBasp programmer can still freely reset the chip during programming. Without the transistor, using this pin as an output would be risky and could lead to blocking the microcontroller's easy programming, requiring the use of a high-voltage programmer.

A curious reader might ask: why is the diode connected to pin 5 instead of, say, 2 or 3, which are not used by the programmer? Thanks to the transistor trick, a larger number of microcontroller pins are available for use and future expansion of the circuit. These are pins: 2, 3, 5, 6, and 7; pins 4 and 8 are occupied by power, while pin 1 is used by the programmer.

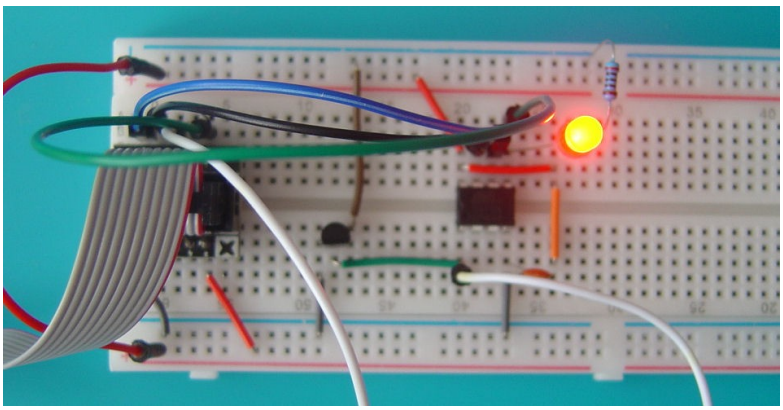
3. Breadboard

We will build the device on a breadboard, ideal for prototypes. Its main advantage is reusability. The downside – over time, after many experiments, the contacts stop connecting, and you need to buy a new board.





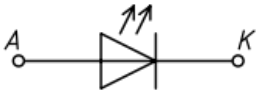
The rectangular breadboard consists of a grid of holes connected by parallel tracks. The tracks along the longer edges of the board run parallel and are intended for connecting the circuit's power (+ and -), while the rest of the tracks run perpendicular to them, with a gap in the middle of the board. It is over this gap that integrated circuits, various connectors, and electronic components are usually mounted. The distances between the holes on the board perfectly match the pin spacing of our microcontroller.



Before connecting the programmer to power via the computer's USB port, it's worth checking with a multimeter if all connections are working properly. Set the multimeter to diode test mode, place one probe e.g. on the programmer's VCC output (pin 2), the other on pin 8 of the microcontroller. The connection should be signaled by a buzzer. It's worth checking all remaining connections according to the schematic in this way. Be especially careful not to create a short on the power line, which could damage the microcontroller.

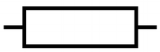
4. Diode and Resistor

According to conventionally accepted rules, current flows from the positive pole (+) to the negative (-). In the case of an LED, the positive pole is the anode (smaller element) and the negative is the cathode (larger element). In the reverse direction, the LED does not conduct current and will not light up.



LED symbol

To use an LED in a circuit, an appropriate resistor must be used; otherwise, the diode may burn out or damage the circuit. Different LEDs have different safe voltage values depending on the diode color. For a red diode, it's 1.6 – 2.2 V. Additionally, for the diode to shine clearly, it needs current in the range of 3 – 10 mA, and the maximum safe current is 20 – 30 mA. The voltage supplied by the computer's USB port is 5 V. So what resistor should be chosen to limit the current?

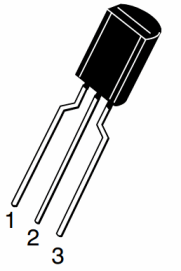


resistor symbol

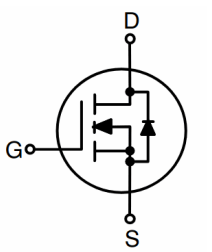
Let's assume average values for our diode's forward voltage as 1.9 V and current as 15 mA. The rest can be calculated from Ohm's law: $R = (5V - 1.9V) / 0.015 A ≈ 207 Ω . In practice, available resistor values come in series, i.e., only some are produced. In the circuit, I used a 200 Ω resistor, which for voltages from 1.6 to 2.2 V gives a safe current in the range of 14 – 17 mA. You can also use a resistor with higher resistance, e.g., 220 Ω. The resistor can be placed before or after the diode; the order doesn't matter.

5. Field-Effect Transistor

The N-type MOSFET transistor 2N7000 has three pins. Looking at the flat side of the transistor, the first from the left is the source, the second – gate, the third – drain. The normal state of the transistor is "closed," i.e., no current can flow between drain and source, but it "opens" if the gate voltage exceeds the threshold value of about 2 V.



1 - source (S); 2 – gate (G); 3 – drain (D)

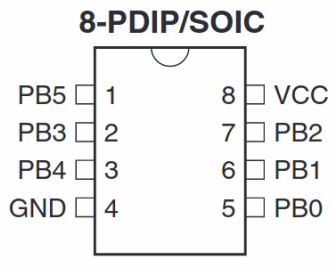


transistor symbol

In our circuit, the transistor's gate is connected to pin 1 of the microcontroller, which during operation has about 5 V, resulting in the transistor opening, while during circuit programming, the programmer shorts pin 1 to ground and "closes" the transistor[3].

6. Microcontroller

The ATtiny13 microcontroller has 8 pins numbered counterclockwise. The first pin is indicated by a special "dot" or notch in the edge of the microcontroller housing. Pins 4 and 8 are for power, the rest can serve as input-output lines, i.e., the microcontroller can set or read voltage on them. In our circuit, we omit pin 1, which is occupied by the programmer.



The ATtiny13, besides the mentioned 6 input-output lines, has a 9.6 or 4.8 MHz clock[4], 1024 bytes of Flash memory, 64 bytes of SRAM, 64 bytes of EEPROM, and 1 8-bit timer. Compared to the capabilities of modern PCs, this is an extremely modest set, but for our circuit's needs, the resources are more than enough. We will use only 1 input-output line and 4 bytes of Flash memory occupied by the source code after compilation.

non-polarized
capacitor symbol

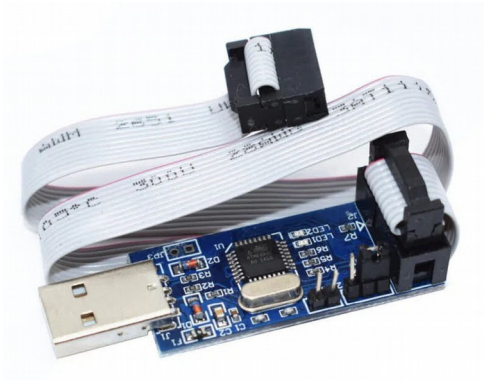
When connecting the microcontroller to power, a good practice is to add a 100 nF ceramic capacitor between VCC (+) and GND (-). This is a non-polarized capacitor, so it doesn't matter which leg we connect to VCC and which to GND. The 100 nF capacitor serves for power filtering, i.e., eliminating external interference. Although in the case of power from the computer's USB port it's not necessary, for external power e.g. via a power supply and voltage regulator, it's also recommended to add two more electrolytic capacitors.

3 In professional systems, it's a good idea to add a high-value resistor (e.g., 100kΩ) between the transistor's gate (G) and ground (GND). This resistor prevents accidental LED illumination due to electrostatic charges when the processor pin is in a high-impedance state (e.g., during system startup or after disconnecting the programmer). In our simple design, we omit this resistor, keeping the number of components to a minimum.

4 Although the ATtiny13 internal oscillator runs at 9.6 MHz by default, it has a factory-enabled clock divider of 8 (CKDIV8 bit), which means that in practice the ATTiny13 runs at 1.2 MHz (1,200,000 Hz) by default.

7. Programmer

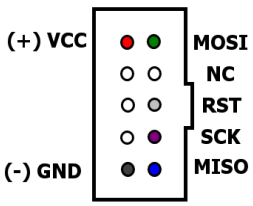
The USBasp programmer is intended for AVR microcontrollers, including the ATtiny13. It has a jumper (usually marked as JP1) that changes the power voltage of the programmed circuit to 3.3 or 5 V (ATtiny13 supports both standards, I set it to 5V). The next jumper JP2 is for programming the programmer, but we won't do that. Jumper JP3 is the SLOW-SCK function for programming microcontrollers at low clock frequencies, including the ATtiny13. My programmer doesn't have a connector there, just holes; I connected them provisionally by twisting a wire.



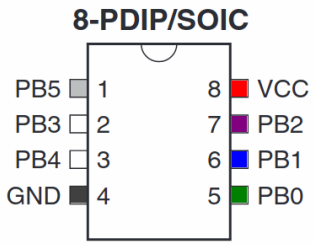
USBasp programmer with ribbon cable

IDC10 connector for breadboard

To program the microcontroller placed on the breadboard, an IDC10 connector is useful, included in the schematic, to which the programmer's ribbon should be plugged. If there's a problem buying such a connector, the ribbon socket can be connected with wires directly to the microcontroller pins, according to the schematic below.



USBasp programmer connector



Microcontroller programming pins

The programmer has two LEDs – one signals the presence of power after connecting it to the computer's USB port, the other lights up during microcontroller programming. Additionally, it should have a fuse (F1) around 500 mA protecting the computer's USB port from damage.

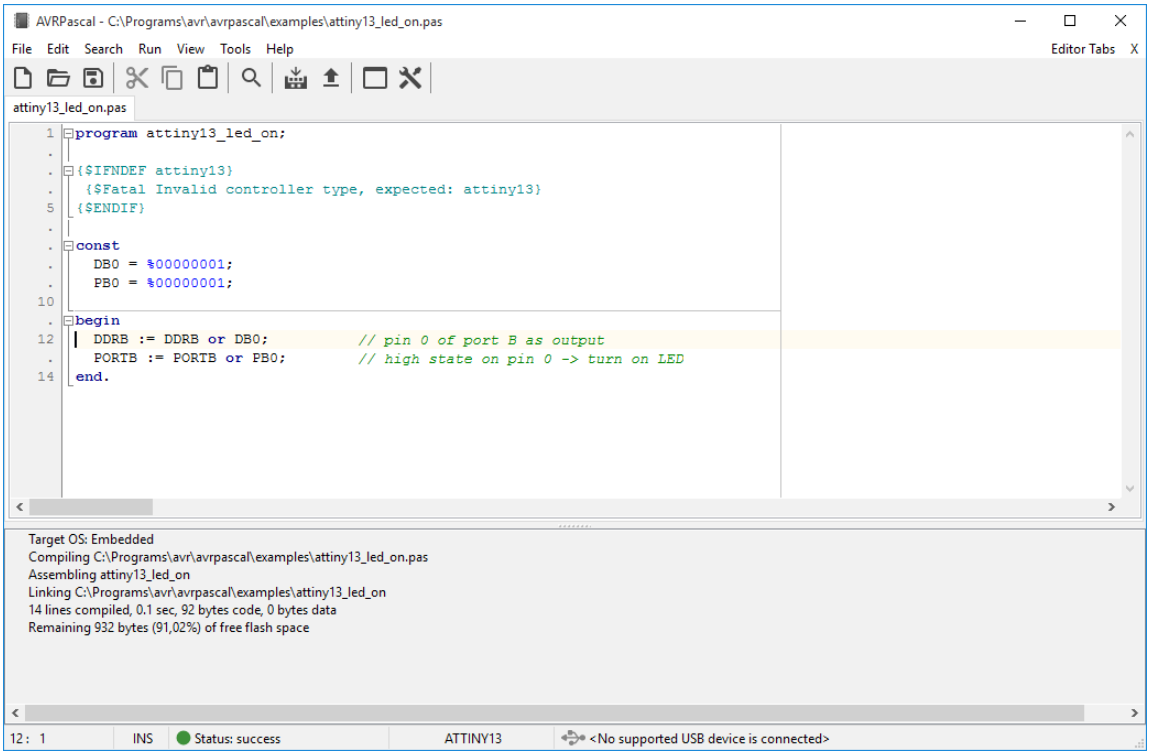
8. Programming Language and Machine Code

We move on to programming our microcontroller. What programming language to choose? There are several languages to choose from: so-called high-level like C or Pascal, or low-level assembler. In the case of assembler, the programmer has full control over the machine code that results after compiling the source code they write. In other words, exactly what is written in assembler will be executed by the microcontroller, without any additional

code usually added by high-level language compilers. The disadvantage of assembler is low intuitiveness and readability, and high code complexity. A golden mean seems to be using assembler where necessary for performance reasons, while maintaining the clarity and simplicity of high-level language code. Such possibilities are provided by the Pascal language and its free compiler Free Pascal Compiler (FPC).

9. AVRPascal Editor and Source Code

To program an AVR microcontroller, you need a text editor, compiler, and a program to send the compiled code to the microcontroller's Flash memory. All these functions are provided by AVRPascal of my authorship, which I provide free of charge for both non-commercial and commercial use on the website https://www.akarwowski.pl/ (or via domain https://www.avrpascal.org/). AVRPascal is available in versions for Windows, Linux, and macOS (64-bit), so it should appeal to more picky fans of different operating systems. It uses the FPC compiler and AVRdude as a tool to write the compiled binary code to the microcontroller's memory.



Let's move to the source code. It can look like this:

```
program attiny13_led_on;

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

const
  DB0 = %00000001;
  PB0 = %00000001;

begin
  DDRB := DDRB or DB0;          // pin 0 of port B as output
```

```
  PORTB := PORTB or PB0;        // high state on pin 0 -> turn on LED
end.
```

The source code starts with the keyword program, which means we are dealing with a program and not, e.g., an included module. The actual code is between the keywords begin and end. Below the word const, we have constant definitions, while comments start with //. The conditional definition block between {IFNDEF attiny13} and {$ENDIF} also requires explanation. This is an AVRPascal mechanism that triggers an error if attempting to compile this code for a microcontroller other than ATtiny13. In our case, remember to select ATTINY13 from the list of microcontrollers available for the compiler in the *View->Options* menu. This allows the compiler to load the appropriate constants and variables specific to ATtiny13.

From the programmer's perspective, such variables in our code are DDRB and PORTB[5], and they require some explanations. In AVR microcontrollers, communication with input-output ports occurs via registers. ATtiny13 has one register bank called "B" (this letter is also the port identifier, the only one it has), other microcontrollers may have more.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x18	PORTB	–	–	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x17	DDRB	–	–	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x16	PINB	–	–	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0

Bank "B" consists of three registers: DDRB, PORTB, and PINB, each occupying one byte (8 bits) in the microcontroller's address space. Each register can be treated as a variable that can be read or assigned a new value. Within this byte, bits 0 to 5 correspond to successive microcontroller pins (bit 0 – pin 5; bit 1 – pin 6; bit 2 – pin 7; bit 3 – pin 2; bit 4 – pin 3; bit 5 – pin 1). Bits 6 and 7 are unused since there are no more pins left for them (pins 4 and 8 are occupied by power).

DDRB is the so-called Data Direction Register, controlling the direction of data flow, i.e., whether data is input or output through the given port (in this case, port B). Upon microcontroller startup, the default register state is all zeros, meaning all input-output lines are configured as inputs (i.e., for reading). To configure a given line (pin) as output, set value 1 to the corresponding bit. In turn, PORTB is the data register. If a given bit in DDRB has value 1 (i.e., output), this register can control the corresponding line, where value 1 is high state (VCC), 0 is low (GND). This way, you can e.g. turn on or off a device connected to this pin. The PINB address register allows reading the voltage value on each line configured as input. We won't use this register now.

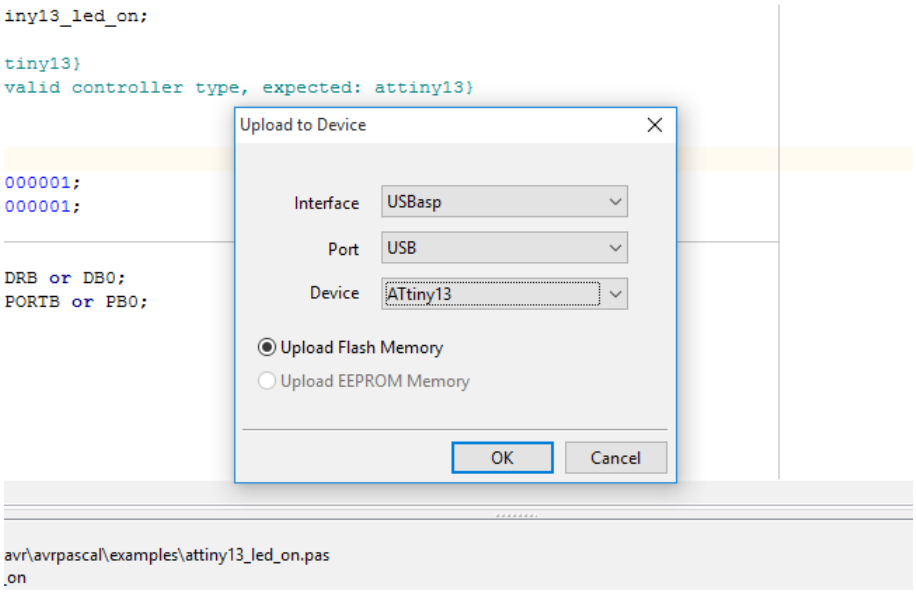
Back to our code. Upon microcontroller startup, all input-output lines are configured as inputs. However, we want to set line 5, to which the diode is connected, as output and set it to high state so the diode lights up. So first, we need to "set" bit 0 of the DDRB register to define the data direction, and then "set" the same bit of the PORTB register to set high state (VCC) on our pin. To "set" a single bit in a given register, we use the OR bit mask. For both registers, we defined the masks as constants, whose value starts with %. This is a hint to the compiler that it's dealing with a number written in binary form, where only bit 0 is set to 1, the rest are not. This is one way to "set" bit 0 first in DDRB and then in PORTB[6].

5 In reality, DDRB and PORTB are locations in the microcontroller's address space.

6 Since we know that both registers' values are default empty (==0), we could equally well assign byte value 1 to both, i.e., "set" bit 0 for each e.g. like this: DDRB:=1; PORTB:=1.

We have the source code ready. It's worth saving this code to a file with extension *.pas or *.pp, and then compile it (*Run->Compile* or Ctrl+F9). If the compiler detects an error in the code (e.g., a typo), the editor should highlight the line where the error occurred in red. If there are no errors – a green dot and "Status: success" should appear on the status bar (at the bottom). The compiler's output is machine code understandable by the microcontroller.

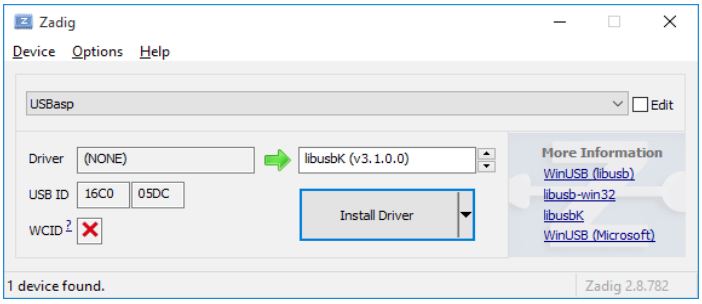
The final step is to upload the machine code to the microcontroller's Flash memory. Connect the programmer to your computer's USB port (if it's not already connected) and select the *Run->Upload menu* (or F9). After confirming the above options, the programmer will upload the machine code to the microcontroller. After a moment, the microcontroller should begin operating, with the LED



10. Technical Notes

Finally, some technical notes regarding AVRPascal. For communication with the USBasp programmer, the program requires the LibUsb driver installed:

- In Windows, you can use libusbK (v3.1.0.0) using the installer available at https://zadig.akeo.ie/



- In Linux distributions like Ubuntu or Mint, the LibUsb driver required by AVRPascal is already installed. Run the program with administrator privileges or add the logged-in user to the *dialout* group.

- In macOS, no additional libraries or privileges are required.

Have fun!

Andrzej Karwowski

Sources of illustrations: Wikipedia, manufacturers' and sellers' datasheets, own photographs.
Nobody is perfect – found an error or inaccuracy, have a question or suggestion – write to ackarwow@gmail.com

I would like to thank @ccrause from the Free Pascal forum (forum.lazarus.freepascal.org) for his valuable technical feedback on this section of the course.