

AVRPascal

How to start

version 09/27/2025

1. What is it used for?

AVRPascal is a free, intuitive environment for programming Arduino boards and AVR microcontrollers in Pascal. It is intended for a wide audience, from beginner hobbyists to advanced enthusiasts of the language and microcontroller programming. Thanks to integration with the UnoLib library (a translation of the standard Arduino library for the Arduino Uno into Pascal), AVRPascal enables the use of familiar Arduino functions. Users who prefer direct programming of AVR microcontrollers can also work with the tool, as it supports the popular USBasp programmer. AVRPascal uses the FPC compiler for AVR, so it should handle any program written for that compiler.

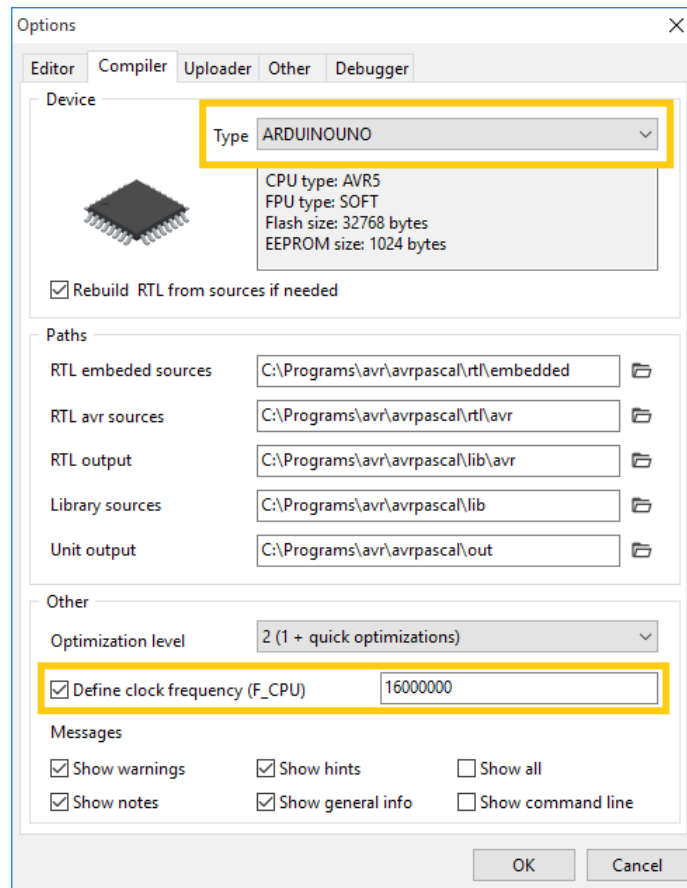
The program was designed for maximum ease of use while providing a wide range of features. Therefore, AVRPascal's interface is similar to popular programming editors, including the Arduino IDE, and should be easy for even beginners to navigate. However, there are a few things you should know before getting started.

Note: This guide assumes that the reader is familiar with the basics of Pascal, so its structures will not be explained. However, readers may be unfamiliar with the structure and work of Arduino boards, AVR microcontrollers, or basic electronics, so this guide provides all the necessary information to understand the examples.

2. Configuration

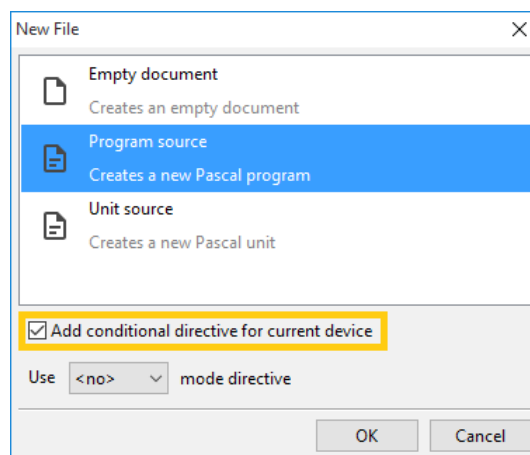
The program does not use any project files. Its configuration is based on global settings available in the *Options* window (*View->Options* menu). To begin, you must specify the target device for which you want to compile your source code. In the *Compiler* tab, select the appropriate device, such as an Arduino board or a specific AVR microcontroller model. However, remember to change this setting when working on a program intended for a different microcontroller to avoid compilation errors or unstable work. The currently selected target device type is displayed in the status bar.

In this same tab, you can also set the F_CPU clock speed constant. This value is required for the UnoLib library to function correctly. The default for the Arduino Uno is 16,000,000 (16 MHz).

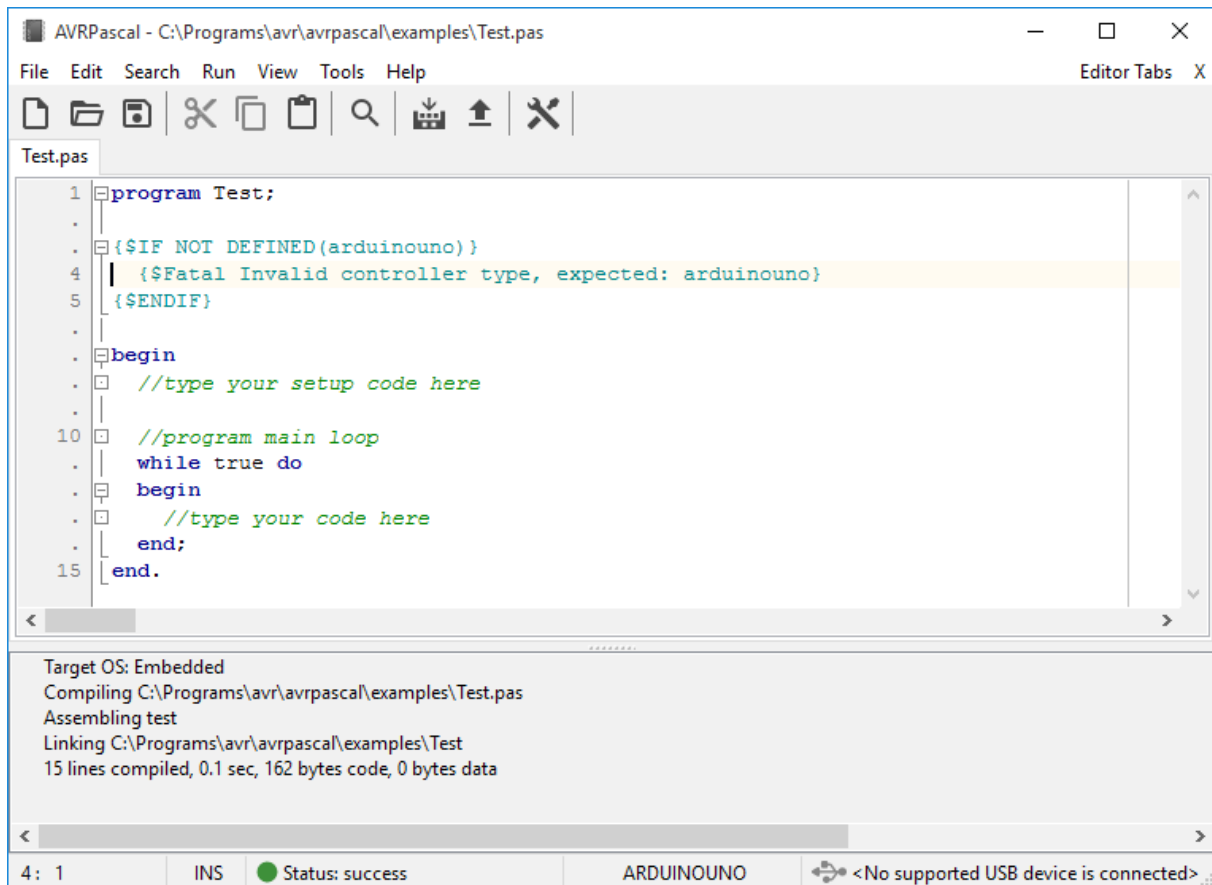


To prepare the compiler for use with the selected device, select *Run->Rebuild RTL Sources*. This will rebuild the Runtime Library sources for that device. Messages from the compilation process will appear in the *Messages* area below the editor tabs.

Because AVR microcontrollers and Arduino boards based on them do not have an operating system, AVRPascal uses only two types of source files: a *program* (the main program file) and a *unit* (the module file). AVRPascal simplifies their creation by providing ready-made templates available in the *File->New* menu (Ctrl+N).



In the *New File* window, in addition to selecting the source file type, you can also select the *Add conditional directive for current device* option. This is useful as it protects against compilation for the incorrect target device. This adds a special directive to the code for the compiler, which will force a compilation error in such a situation (*Invalid controller type, expected...*). If you create a new program file for the Arduino Uno (current device type) and then save it under a name such as *Test.pas*, the generated code should look like this:



```
AVRPascal - C:\Programs\avr\avrpascal\examples\Test.pas
File Edit Search Run View Tools Help
Test.pas
1 program Test;
.
.
4 {$IF NOT DEFINED(arduinouno)}
5 | {$Fatal Invalid controller type, expected: arduinouno}
6 | {$ENDIF}
.
.
7 begin
.
8 //type your setup code here
.
.
10 //program main loop
.
11 while true do
.
12 begin
.
13 //type your code here
.
14 end;
15 end.
```

Target OS: Embedded
Compiling C:\Programs\avr\avrpascal\examples\Test.pas
Assembling test
Linking C:\Programs\avr\avrpascal\examples\Test
15 lines compiled, 0.1 sec, 162 bytes code, 0 bytes data

4: 1 INS ● Status: success ARDUINOUNO <No supported USB device is connected>

This minimalist program can be compiled (*Run menu -> Compile* or F9), and details of the compilation process will appear in the *Messages* area. Compilation should be successful, and the status bar will display ● *Status: success*. If compilation fails for any reason, the message ● *Status: failure* will appear. As you can see, the template program does nothing but execute an infinite loop and is not worth uploading to an Arduino board.

3. Your first Arduino Uno program

Programming microcontrollers typically begins with a program that flashes an LED. This is the equivalent of the "Hello World" program for desktop applications. We will start with that as well. For the Arduino Uno, this is also a good idea because you won't need to build any circuitry, as Arduino boards have a built-in LED.

We will use the template we have and add the procedures from UnoLib. As mentioned earlier, the procedure names should be familiar to those who program Arduino boards in the Arduino IDE. The new code might look like this:

```
program TestBlink_Uno;

{$IF NOT DEFINED(arduinouno)}
  {$Fatal Invalid controller type, expected: arduinouno}
{$ENDIF}

uses
  defs, timer, digital;           //used modules from UnoLib

const
  LedPin = 13;                    //built-in LED in Arduino Uno

begin
  //type your setup code here
  PinMode(LedPin, OUTPUT);       //define LedPin as output
  DigitalWrite(LedPin, LOW);     //set LedPin in low state

  //program main loop
  while true do
    begin
      //type your code here
      DigitalWrite(LedPin, HIGH); //set LedPin in high state
      Delay(1000);                //keep LED on for 1 second
      DigitalWrite(LedPin, LOW);  //set LedPin in low state
      Delay(1000);                //keep LED off for 1 second
    end;
  end.
```

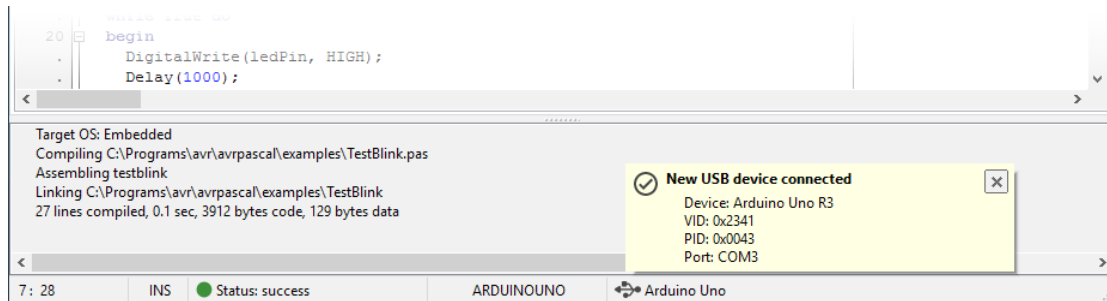
This code is quite simple, but requires some explanation. The *uses* section includes the *defs*, *timer*, and *digital* modules from UnoLib, which contain the declarations of the constants and procedures we will be using. Next, we defined the *LedPin* constant with a value of 13, which corresponds to the built-in LED in Arduino numeration. Next, in the configuration section of the program, we used the *PinMode* procedure, which tells the Arduino that *LedPin* should be treated as an output (OUTPUT constant). Next, using the *DigitalWrite* procedure, the LED pin is set to low (LOW constant), turning off the LED (in case it was previously lit for some reason).

The main part of the program is contained in an infinite loop, meaning the sequence of procedure calls within it will repeat indefinitely (unless the board loses power). First, using the familiar *DigitalWrite* procedure, the LED pin is set to high (constant HIGH), causing it to light up. Then, the *Delay* procedure is called, accepting values in milliseconds. In our case, we specified a delay of 1000 milliseconds, or 1 second. Then, we repeat the operation, but instead of turning the LED on, we turn it off by setting its pin to low.

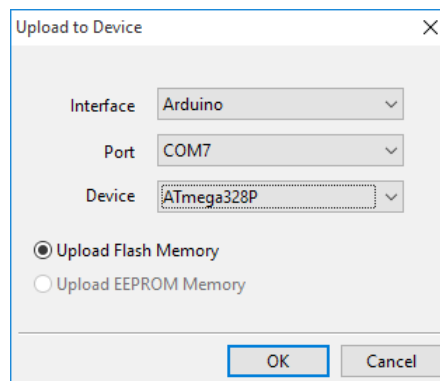
If we didn't make a mistake while copying, this code should compile without any problems. Since we're specifying external modules in the program, it's worth using the *Run->Build All* menu (or Shift+F9). Just remember that the current device in the AVR Pascal configuration should be ARDUINOUNO and F_CPU should be set to 16,000,000.

Note: If during compilation, for some reason the compiler cannot find the modules specified in the *uses* section, you should search for them (they should be in the lib directory) and compile them one by one, and then go back to compiling the program.

With the program compiled, we're ready to connect the board to the computer and upload the code to the Arduino Uno. After connecting the Arduino Uno, AVR Pascal should detect the new device, displaying a message similar to the one below (note: it doesn't detect Chinese clones). If the message doesn't appear, make sure the Arduino board drivers are installed (Windows) or that the user is in the *dialout* group (Linux).



To upload the compiled code to Arduino, select *Run->Upload (F9)*. An *Upload to Device* window similar to the one below will appear, with all necessary options already selected. Just make sure the port number matches if you have more than one virtual serial port open in your system.



The code upload process should be quick, with details visible in the *Messages* area. For code uploading, AVR Pascal (like the Arduino IDE) uses the proven AVR Dude program.

```
Reading 2226 bytes for flash from input file TestBlink_Uno.hex
Writing 2226 bytes to flash
Writing | ##### | 100% 0.51s
Reading | ##### | 100% 0.40s
2226 bytes of flash verified
```

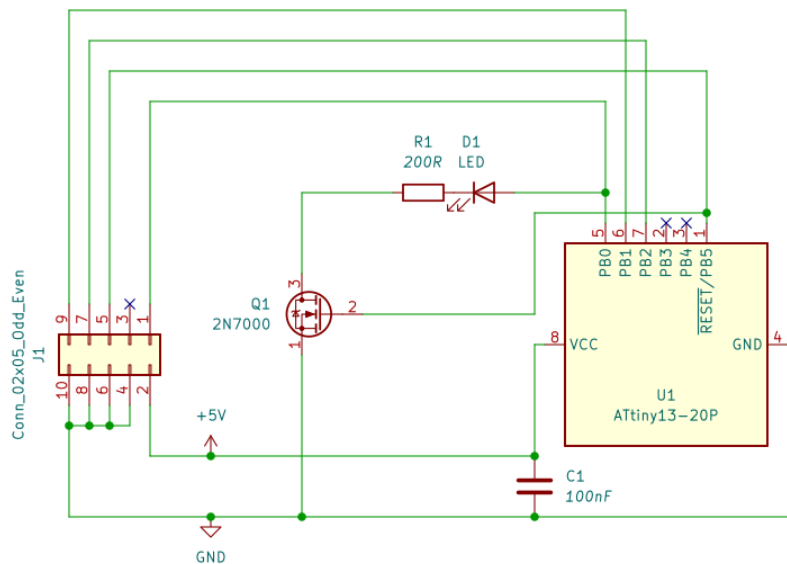
Avrdude done. Thank you.

After a while, the program should work as expected.

4. If you prefer USBasp

Programming with USBasp is a more demanding task, as you must first design and build the appropriate circuit, preferably on a breadboard. This is one reason why electronics are so fascinating! Let's assume our task also involves flashing an LED, but we will choose an inexpensive ATtiny13 microcontroller for this purpose.

To build a simple circuit, in addition to the microcontroller itself (U1), we'll need a red LED (D1), a 200 Ω resistor (R1), a 100 nF ceramic capacitor (C1), a 2N7000 transistor (MOSFET) (Q1), an IDC10 connector for the programmer ribbon cable (J1), and some wires. The circuit diagram might look like this:



The circuit's principle of operation is simple: when the microcontroller (U1 in the diagram) sets the pin connected to the LED (D1) high, the LED lights up; when it sets it low, the LED goes out. However, the microcontroller requires power (3.3-5 V) to work properly. To keep the circuit simple, we'll draw power from the programmer (i.e., the computer). We've also connected the LED (D1) in series with a resistor (R1) to prevent damage to the LED.

The diagram shows two ground lines: the main one extends directly from the programmer connector (J1), and the second one runs from the drain of the transistor (Q1) through the resistor (R1) to the cathode of the LED. Why the complexity? The point is that both the LED and the programmer use pin 5 of the microcontroller, and interference can occur during programming. The transistor acts as a switch, taking advantage of the fact that during normal microcontroller operation, pin 1 (RESET) is connected to 5 V, while the programmer connects this pin to ground during operation. In other words, during programming, the transistor closes, stopping current flowing through the diode and preventing the diode from loading the pin also used for programming.

Note: Of course, you can use another pin to control the LED and avoid the complications with the transistor, but this example allows for further expansion while saving the available pins of the ATtiny13 (only 5 not counting the RESET pin used for programming the microcontroller).

The source code needed to program this circuit requires some knowledge of the ATtiny13's structure. The LED in the schematic is connected to pin PB0. It connects to port B – the only port available for I/O operations in this type of microcontroller. This means we need to properly control port B by first setting our pin as an output and then alternating between high (1) and low (0) states. To tell the microcontroller to treat the pin as an output, we need to set the corresponding bit in the Port B Data Direction Register (DDRB), which specifies the direction of data flow. To turn the voltage on and off at this pin, we need to manipulate the corresponding bit in the PORTB register, which controls the voltage state at the pin.

Additionally, we need a delay routine lasting, say, half a second, to prevent the LED from flashing too rapidly. Writing a precise delay routine is no trivial task. The procedure proposed here is based on counting the duration of individual machine code (assembly) instructions, but we won't discuss it in detail. We'll simply assume it works well enough. The source code based on our assumptions might look like this:

```
program TestBlink_ATTtiny13;

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

procedure HalfSecondDelay;
label
  Delay;
begin
  asm
    ldi R25, 234 // 2 cycles; = Hi(60 000), set high byte to register R25
    ldi R24, 96 // 2 cycles; = Lo(60 000), set low byte to register R24
  Delay:
    sbiw R24,1 // 2 cycles; decrease 16-bit counter by 1
    nop // 1 cycle; do nothing
    nop // 1 cycle; do nothing
    nop // 1 cycle; do nothing
    nop // 1 cycle; do nothing
    nop // 1 cycle; do nothing
    nop // 1 cycle; do nothing
    brne Delay // 2 cycles if jump to label, otherwise 1 cycle
    nop // 1 cycle
  end;
end;

const
  DB0 = %00000001; // bitmask of pin 0 in DDRB
  PB0 = %00000001; // bitmask of pin 0 in PORTB

begin
  //type your setup code here
  DDRB := DDRB or DB0; // pin 0 of port B as output
```

```

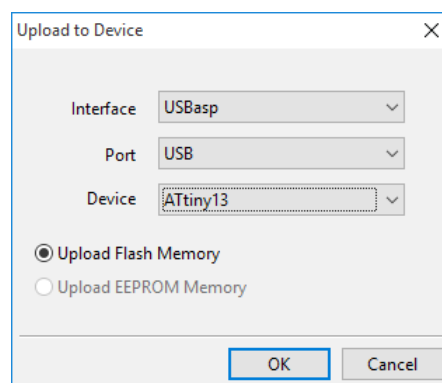
PORTB := PORTB and not PB0; // set low state on pin 0 -> turn LED off

//program main loop
while true do
begin
  //type your code here
  PORTB := PORTB or PB0; //set pin in high state -> turn LED on
  HalfSecondDelay; //keep LED on for half second
  PORTB := PORTB and not PB0; //set pin in low state -> turn LED off
  HalfSecondDelay; //keep LED off for half second
end
end.

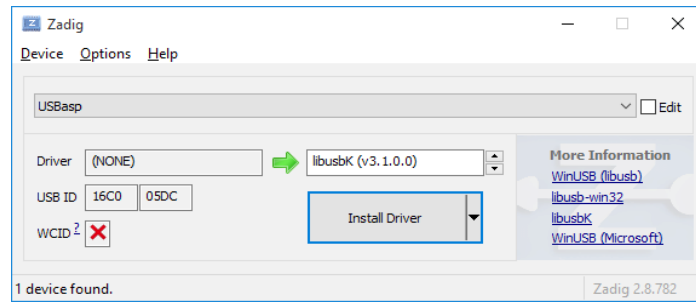
```

The above example does not use any additional modules and contains all the necessary code. First, the *HalfSecondDelay* procedure is defined, which is later used to delay the LED state change. Next, constants are defined as a bit mask for selecting pin 0 on port B. Next, the operations discussed earlier are performed, namely selecting the output direction for the pin on port B in the DDRB register using the DB0 bit mask and the *or* operation. The configuration portion of the code concludes by setting the pin low, this time using the PORTB register and the *and not* operation. The infinite loop is very similar to the Arduino Uno example, with the difference that setting the pin high and low is performed via the *or* and *and not* operations directly on the PORTB register, and using the *HalfSecondDelay* delay procedure defined earlier.

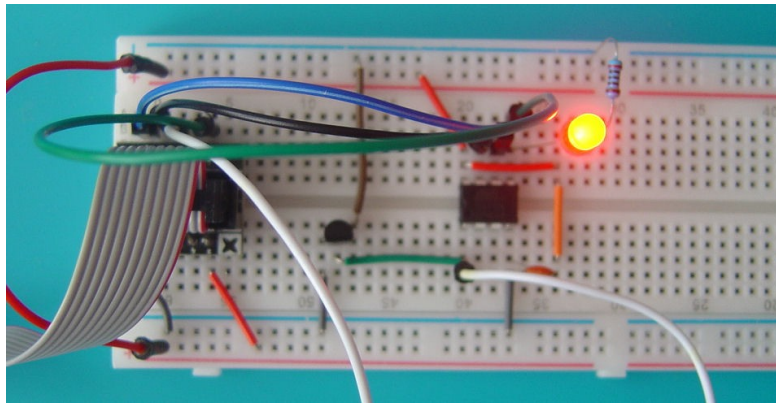
If the code is error-free, it should compile without any issues using the *Run->Compile* menu (Ctrl+F9), and the compilation results will be displayed in the *Messages* area. After connecting the programmer to the computer, AVR Pascal should detect it as USBasp. To upload the compiled code to the microcontroller's flash memory, proceed similarly to the Arduino, using the *Run->Upload* menu (F9). The *Upload to Device* window will display slightly different options, which should not need to be changed. After accepting the settings, AVR Pascal will send the compiled data to the microcontroller using AVR Dude.



If you encounter problems connecting to the programmer in Windows, it's worth making sure you have the LibUsb driver installed, which is necessary for USBasp to work. If not, you can use libusbK (v3.1.0.0) using the installer available at <https://zadig.akeo.ie/>



In the case of Linux systems, distributions such as Ubuntu or Mint already have the LibUsb driver installed, which AVRPascal requires, so you just need to make sure that the logged in user belongs to the *dialout* group.



If everything goes well, our device should start working as expected.

* * *

I hope these tips will encourage you to embark on your AVRPascal adventure, and I wish you the best of luck with your future projects. It's also worth checking out the AVRPascal manual, where all the program's features are described in detail. Have fun!

Andrzej Karwowski
email: ackarwow@gmail.com