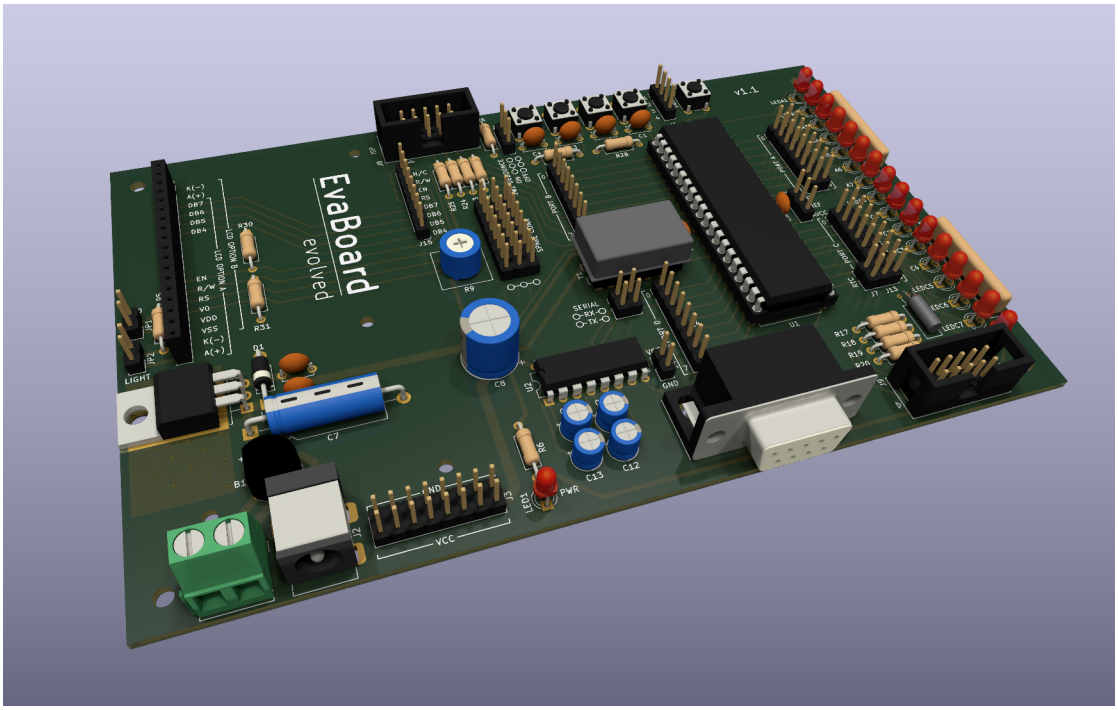
Guide Book

For the ATmega644(A) Evaluation Board



Contents

1 Introduction

This is an evaluation board for the ATmega644(A). It is inspired by (and compatible with) the evaluation board used in the lab course “Praktikum Systemprogrammierung” in the computer science curriculum at RWTH Aachen.

The university encourages students and educators to build the board themselves. However, the resources it provides are outdated, not well-explained, and at times outright faulty.

This project provides a complete remake of the original board with a focus on open source software and budget-friendly components. It also includes a ton of documentation, especially for students who have not worked with electronics before.

There are two add-on boards available in separate repositories:

- SRAMBoard adds an external 128kByte SRAM
- ADDABoard for experimenting with analogue-to-digital and digital-to-analogue conversion

1.1 Getting to Know the Evaluation Board

At the heart of the evalation board is the *ATmega644(A) microcontroller*[1] from Microchip Technology Inc. A microcontroller is essentially a CPU with all the necessary peripherals (memory, pin drivers, communication modules etc.) included, which means it needs almost no additional external components to function. Figure 1 shows the central part of the board's schematic.

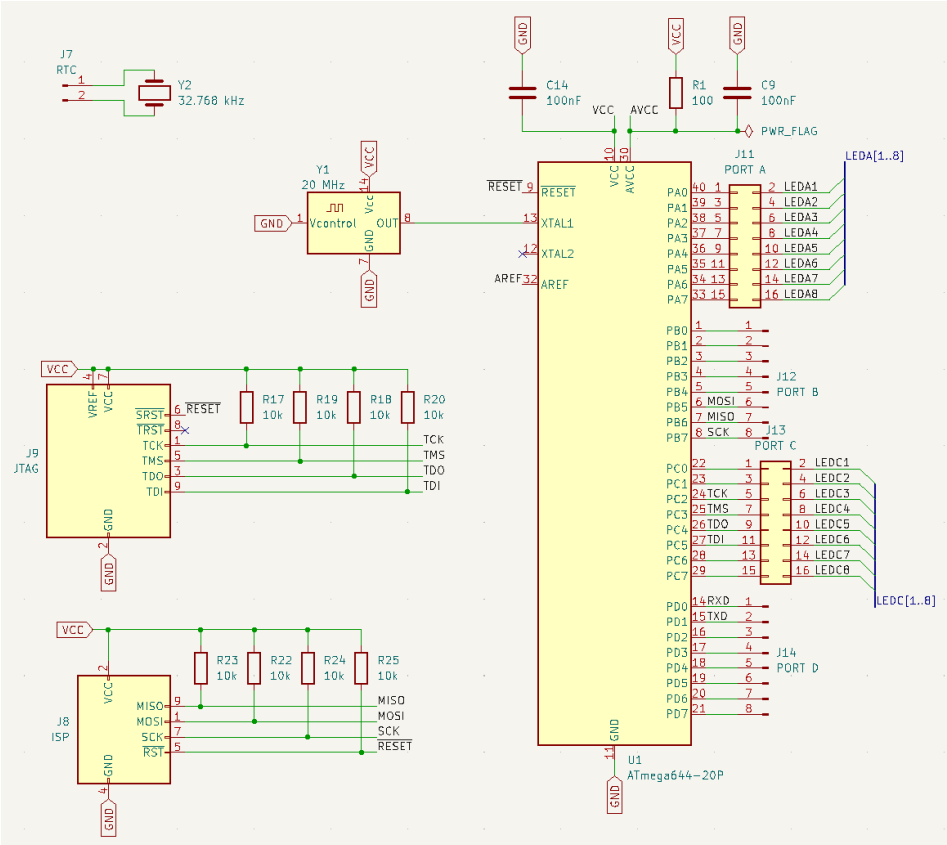


Figure 1: Schematic of the Main Part of the Board

[1]The versions with and without the "A" are for all practical purposes the same

The one thing it does need is a *supply voltage*, in this case 5V. Most of the components in the lower left quadrant of the board are there to provide just that. Figure 2 shows the relevant part of the schematic. A DC power supply can be connected to the board either via the barrel jack J1 or the screw terminal J2. The bridge rectifier B1 ensures that the positive and negative leads are connected to the correct nets, regardless of which way the power is plugged in. The 7805 linear voltage regulator U3 brings the voltage down to a stable 5V.

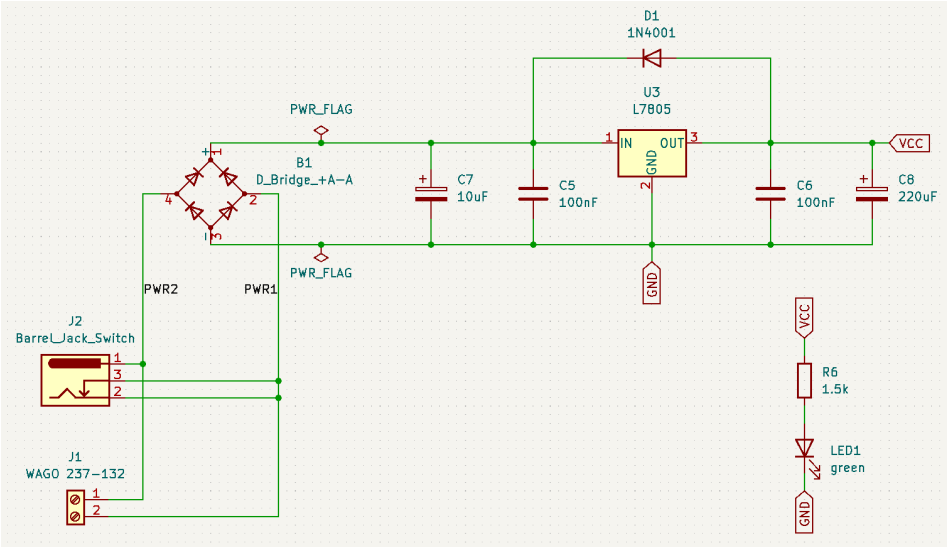


Figure 2: Schematic of the Power Supply Part of the Board

In order to run, the microcontroller needs a *clock source*, i.e. a circuit that outputs a steady stream of alternating high (5V) and low (0V) voltages. While the ATmega644 has an internal oscillator providing an 8MHz clock, higher speeds (up to 20MHz) require an external clock source, in our case the 20MHz crystal oscillator Y1. There is a secondary optional watch crystal (Y2) with much lower frequency but higher accuracy that the microcontroller can use for a real time clock. In order to use it, jumper the lower left two pins of J13 (Port C6 and C7) to J7.

Machine code can be transferred to the microcontroller via one of two interfaces, *ISP* and *JTAG*. There is a pin header for each of them (J8 and J9). These headers are connected to the appropriate pins of the microcontroller, both including the reset pin which suspends the controller's operation during programming. The reset pin is also connected to the *reset button* (SW5), allowing the user to manually hold the microcontroller in reset. The reset line has a pull-up resistor (R25) which keeps the voltage high while no programmer is attached and prevents accidental resets.

Of the microcontroller's 40 pins, 32 are software-controllable *general purpose inputs/outputs* (GPIO), organised in four groups (Port A, B, C, and D) of 8 pins each (0..7). On the evaluation board, these pins are connected to pin headers (J12, J14 and the left-hand columns of J11 and J13), where the user can attach cables or jumpers to connect them to things.

Many pins have alternate functions. Check Figure 1-1 in the datasheet for a complete pinout of the ATmega644(A). For example, Pins C6 and C7 can act as TOSC1 and TOSC2, i.e. connectors

for a secondary oscillator.

The rest of the board is just there to make working with the microcontroller more comfortable:

- 16 *LEDs* (LEDA1..8 and LEDC1..8) with their anodes connected to 5V via current-limiting resistors (RN1 and RN2). Their cathodes are connected to the right-hand columns of J11 and J13. When any of those pins gets connected to GND, the corresponding LED lights up. Typically, you would connect them to micrcontroller pins, configure the latter as outputs and then control the LEDs via software. This is particularly easy to do with the Port A and Port C pins as they are located directly adjacent. Be aware that the logic is inverted: an LED is on when its corresponding pin is connected to GND (logic 0), and off when connected to 5V (logic 1).

- 4 *buttons* (SW1..4) along the top of the board connect the respective pin on J6 to ground while pressed. If you wire any of these to some microcontroller pin and configure the pin as input, you can read the state of the button from software (again with inverted logic). You must activate the internal pull-up resistor for the pin, otherwise you won't get a logic 1 when the button is not pressed.
  Each button has a rudimentary debouncing circuit to avoid spurious inputs. For SW1, this can be disabled via JP3. See Figure 3 for details.

- An HD44780-compatible[2] *LCD* module can be plugged into J16. These types of LCD need a supply voltage (via JP1), a voltage for their LED backlight (via R5 and JP2), and a third voltage to set the contrast (adjustable via R9). Seven signal lines (RS, EN, R/W, and DB4..7) are used to communicate with the LCD and these are routed to J15 from where they can be wired to some microcontroller pins. See Figure 4. Since communicating with these LCDs is a bit complicated, a driver is available in this repository. Most HD44780-compatible LCD modules come with one of two standard pin layouts. This board can accomodate both kinds. For more details, see Section 2.2.2.

- Finally, the board provides a *serial* connector (J10) which can be used for different purposes. The most common one is to connect it to a computer to send text messages back and forth. A MAX232 driver IC converts between the microcontroller's voltage levels (0V and 5V) and those required by RS-232 (-15V..-3V and +3V..+15V). It uses a charge pump involving C10..C13 to generate +9V and -9V from the 5V supply voltage. The ATmega has two pins, RX (Pin D0) and TX (Pin D1) which can be dedicated to serial communication. To connect these to the MAX232, mount jumpers on JP4, see Figure 5. Don't forget to dismount the jumpers before using these pins as GPIOs!

Figure 6 gives an idea of the board's layout. For more details, have a look at the board's complete schematic.

1.2 Changes from the Original Board

While the board was designed to be as similar as possible to the original, there are some notable changes. Some fix issues while others provide more options for the components. Most importantly, the position of the pin headers has not changed which means you can use add-on boards from the original board and vice versa. Here is a list of all the changes:

[2]The HD44780 was a driver IC for dot matrix LCDs by Hitachi. While it is now dicontinued, many clones exist and it has become a quasi-standard for these kinds of LCDs.

- There were two pull-up resistors on the reset line, making it effectively 5kΩ. While that would probably still work, some programmers might have a hard time.

- The power plug was seriously messed up. The intended model was a very specific and hard to get one (Lumberg 1613 18) which the BOM didn't even list correctly. This barrel jack needed a rectangular hole in the PCB which almost completely cut a trace.
  In the remade version, the trace was moved. It is still possible to use the same barrel jack, but now more common (and cheaper) ones fit, too.

- The originally intended LCD is unreasonable expensive (approximately 14€, when you can get similar ones for less than 5€). It also has a very uncommon pin layout, making it difficult to find affordable alternatives.
  The remade board offers two options, allowing for both the original LCD as well as ones with the standard pin layout. The only downside to the latter is that they stick out over the side of the board and only one of the four mounting screws can be used. That is usually not a problem, it should hold even without any screws.

- The unused eighth pin of J15 used to be GND. This is needlessly dangerous when the LCD is connected to a port via an 8-pole cable. If the eighth pin of the microcontroller is accidentally configured as a high output, it gets shorted to GND. In the remade version, the eighth pin is not connected.

- The crystal oscillator was moved slightly upwards to make the serial jumpers more easily accessible.

- Pin 2 of the MAX232 was connected via C13 to GND instead of VCC. While some manufacturers allow this (TI for example, see footnote to Figure 6 in the datasheet), others (like Maxim, see Figure 5 in the datasheet) do not and we've seen parts go up in smoke due to this. The remade board has it connected to VCC instead which works for all MAX232.

- Added D1 to protect U3 in case the board is supplied with power directly via its VCC net (as is the case with the original AD/DA add-on board).

- The original board made a half-hearted attempt at debouncing the buttons by adding capacitors in parallel. While this works to some extend when releasing a button, it causes a brief but very large current when pressing it. The new board adds R26..R29 to avoid that. This required a change to the jumper (JP3) that controls the debouncing of SW1. It is now a three-pin (two-position) jumper instead of a 2-pin (on-off) one. The solution is still not ideal and relies on the (very small) hysteresis of the ATmega's inputs. Fortunately, the buttons are not very bouncy to begin with.

- Added a 24-pin header ("SPARE CONN") to allow three-way connections or just as a storage space for spare jumpers.

Caution

The university hands out an LCD driver to students which has a serious flaw: While querying the LCD's busy flag, it briefly configures both the LCD's and the microcontroller's data pins as outputs, causing a short. This lasts only for one clock cycle but that is enough to sometimes cause damage to an LCD.
For safety, it is highly recommended to use the LCD driver provided in this project instead.

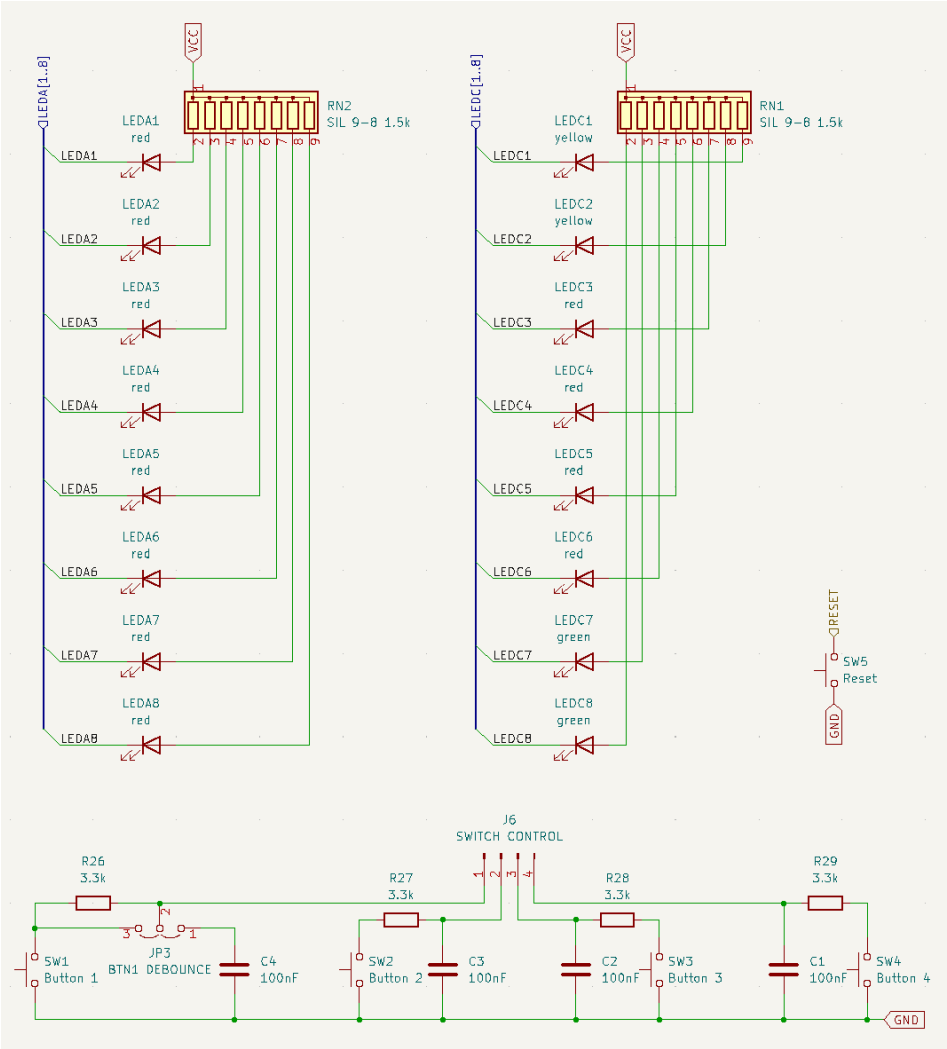


Figure 3: Schematic of the LEDs and Buttons

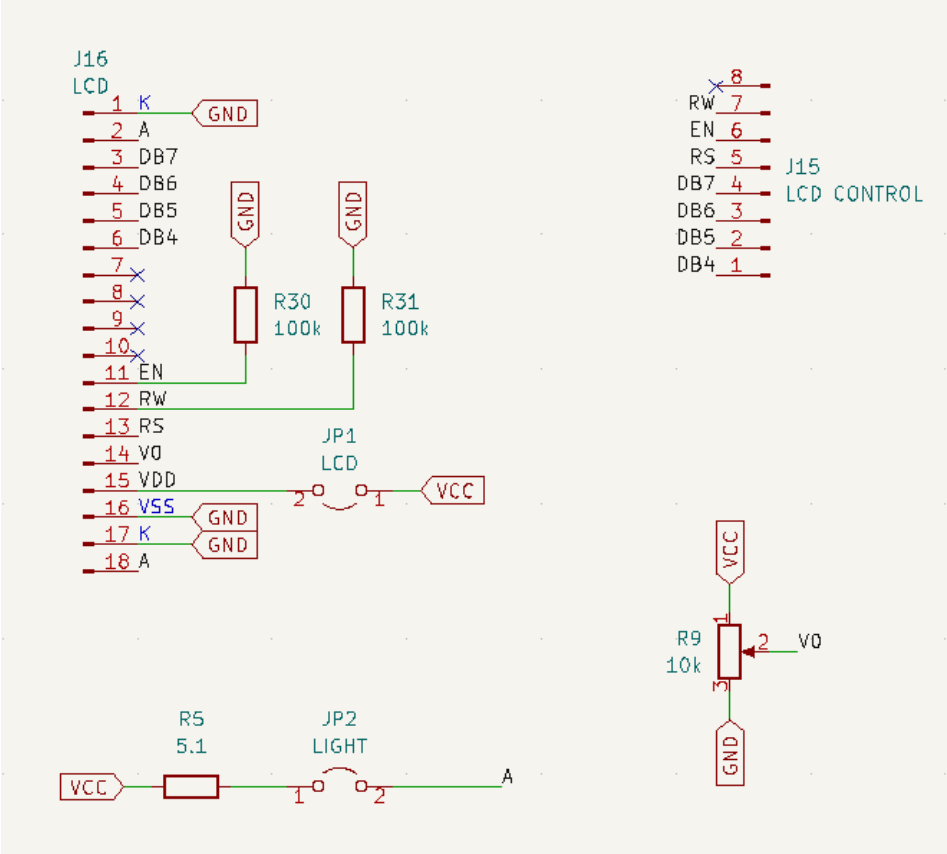


Figure 4: Schematic of the LCD circuitry

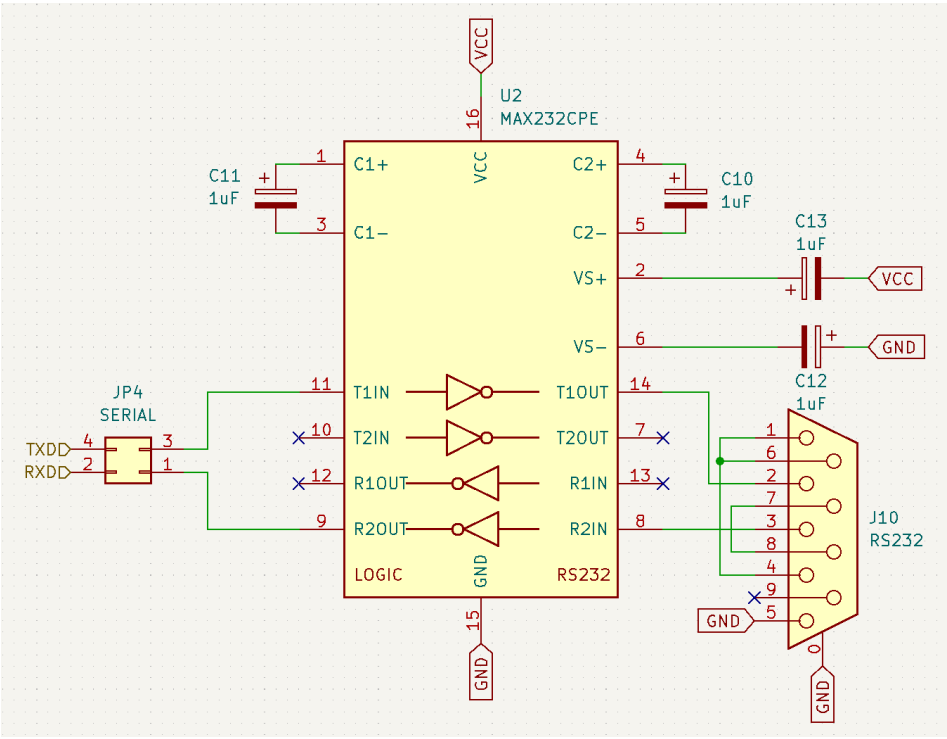


Figure 5: Schematic of the Serial Communication Part of the Board

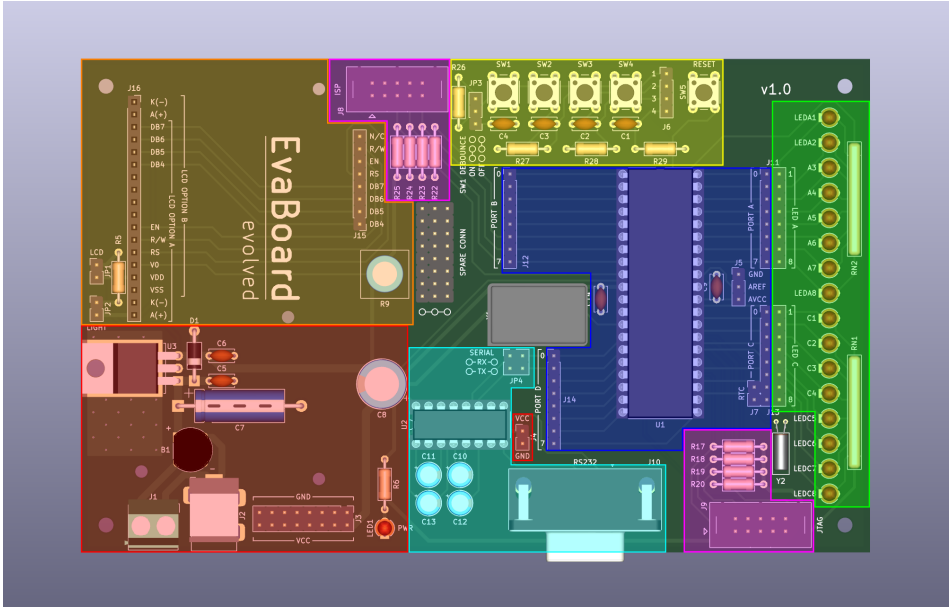


Figure 6: Layout of the board: Microcontroller and GPIO headers (blue), Programming (purple), Power Supply (red), LEDs (green), Buttons (yellow), Serial (mint), LCD (orange)

2 Building the Board

2.1 Necessary Parts and Tools

2.1.1 Tools

As a bare minimum, you need a soldering iron, solder, some pliers, and wire cutters, see Figure 7. Find a workspace with a fireproof surface, make sure it is well-lit and well-ventilated. Keep soldering materials away from any kind of food, small children, and pets.

Figure 7: Basic Tools for Soldering the Board

For the soldering iron, as a rule of thumb, smaller is better when it comes to electronics. But since all components are relatively large, most common soldering irons will work. The solder should not be too thick (something like 1mm diameter works great) and contain a flux core. Leaded solder is slightly easier to work with but poses a health risk. It is highly recommended to use lead-free solder.

If you need to undo a solder joint, you can use a solder pump and/or some solder wick. But any de-soldering carries the risk of damaging components or the PCB. It is much better to avoid it in the first place.

The board needs a DC voltage between 7.5V and 12V. Theoretically, up to 25V are possible but since the voltage regulator U3 turns the surplus into heat, it is not recommended to go that high. You don't need an expensive bench supply. A standard wall plug (10..15€) with up to 1A output current should be more than enough to power your board. Some like the one in Figure 8 have an adjustable voltage so they can be used for other things, too.

If you have a phone charger that supports USB Power Delivery (USB PD) at 9V or 12V, you can

Figure 8: Wall plug as power supply. This one comes with interchangeable plugs.

also use that. In order to get the charger to actually output 9V or 12V rather than the default 5V, you need a "trigger cable" (sometimes called "decoy cable"). These cables contain a chip that negociates the higher voltage with the charger. They can be bought for cheap (3..8€) and some even come with a power jack that fits into the J2 connector.

In order to copy machine code from your computer to the microcontroller, you need a programmer. ISP programmers come as cheap as 5€. See Section 3.2 for details.

Optional: When debugging programs on the microcontroller, it can be handy to have a serial connection to send debug messages to the computer. The board has a 9-pin D-subminiature connector (J10). If your computer still has a serial port, you only need a serial cable. Otherwise, you can buy a USB to serial converter. See Figure 9.

Optional: A multimeter (even a cheap one like in Figure 10) is a great help for testing or debugging your solder work.

2.1.2 Getting the PCB

The PCB (printed circuit board) is the base of the board. It holds all the components and makes the electrical connections between them via copper traces on the top and bottom side (it's a "2-layer board"). It was designed using the open source software KiCAD.

While it is possible to make PCBs yourself, it is definitely not recommended for beginners. There are a number of PCB manufacturers that produce self-designed PCBs in small quantities

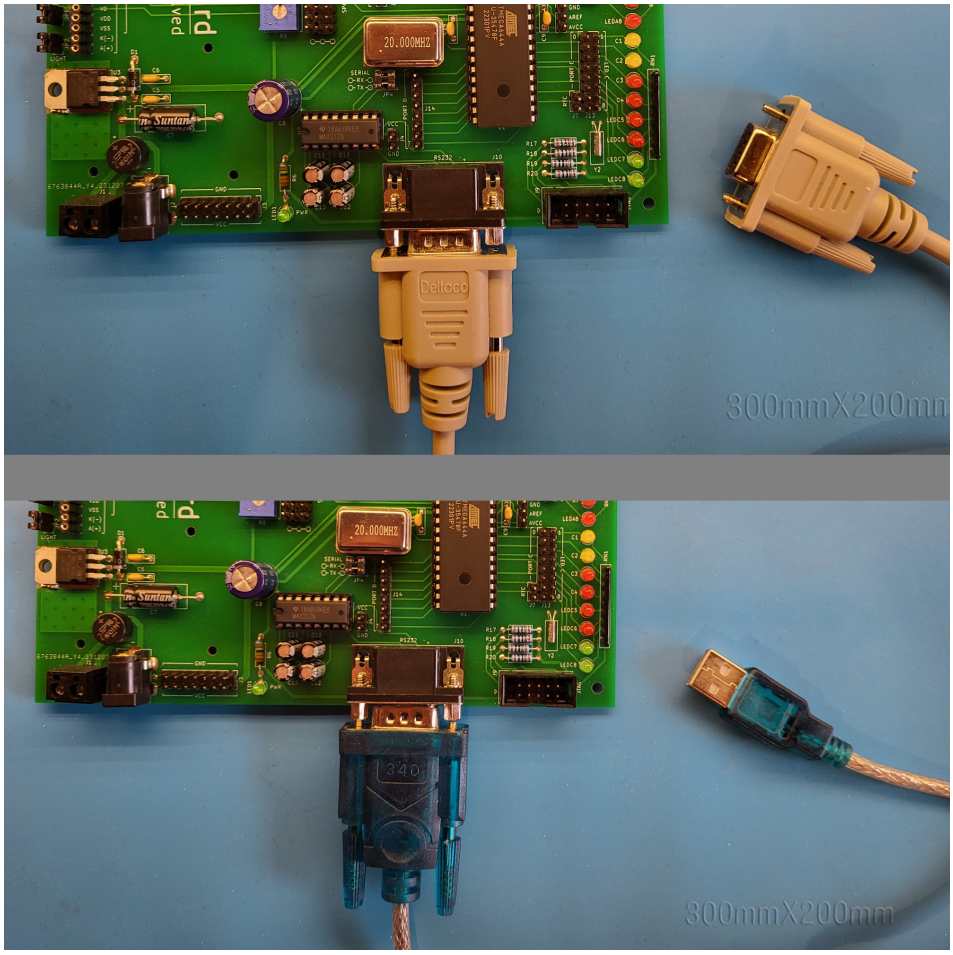


Figure 9: Options for Serial Connection

for hobbyists. The cost of PCB manufacturing has gone down massively in recent years but since most of the affordable ones ship from China, expect a delay of roughly two weeks.

The cost for the PCB itself is usually dwarfed by the shipping cost. That's why you will typically get five or more PCBs for the same total price as a single one. If you find other students who want to build the board as well, you might get away with less than 5€ per PCB. Consider including PCBs for the add-on boards in the same order.

Some manufacturers accept a KiCAD project, but most will want so-called "gerber files". To create those, install KiCAD (the board was designed with version 6, but newer ones probably work, too). For Windows, there is an installer. On Linux, you can use a package manager to install the packages `kicad` and `kicad-packages3d`. Start KiCAD and open the project from the `KiCAD/` subdirectory of this repository. Go to the PCB Editor and in the "File" menu choose "Fabrication Outputs", "Gerbers". Enter "Gerbers/" for the output directory. Now you need to consult the manufacturer's website for the exact options. These are different for every manufacturer but usually easy to find. Hit "Plot" to create the first batch of files. Then click

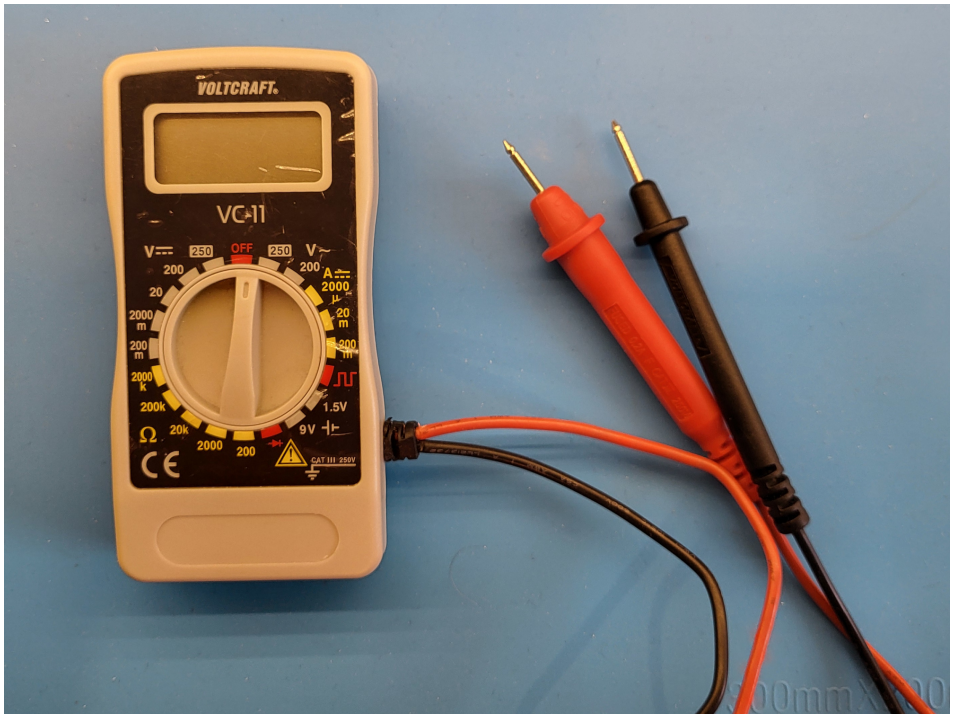


Figure 10: Even the cheapest multimeter will do. This one was literally free.

"Generate Drill Files" and again follow the manufacturer's guidelines to create drill files. All the gerber files are placed in the `Gerbers` subdirectory of the KiCAD project. Pack them all into a zip file. This is what you upload on the manufacturer's website. Before you do that, you can check them in KiCAD's own Gerber Viewer from the KiCAD main window.

During the ordering process, you might be able to choose between different options for the soldermask (the green stuff, but other colors are available) and the surface finish (pre-applied solder on the exposed copper pads). It is highly recommended to get a lead-free finish, even if that costs a little extra. With the leaded option, your board qualifies as toxic waste and you cannot dispose of it in the household rubbish bin. Also lead can cause brain damage.

2.1.3 Purchasing Components

The components you need for the board are listed in the bill of materials ("BOM"). The fastest option is to buy them from a local distributor. In Germany, Reichelt, Conrad, or Pollin are popular stores. Internationally, Mouser or Farnell are well-known. Some of them are more geared towards hobbyist, others target businesses. Pay attention to shipping costs, minimum order value, and whether VAT is included in their listed prices. The BOM lists order codes for Reichelt, but don't read that as an endorsement for that particular distributor.

Keep an eye out for bulk pricing. Some components like resistors have massive discounts above a certain quantity. If you need 8, check if 10 are cheaper, if you need 40, check 100 etc.

In order to save on shipping costs, it is probably best to order from as few places as possible.

Some of the more expensive components like the microcontroller or the LCD might be cheaper when sourced separately. If you're feeling adventurous, you can try eBay or AliExpress. However, with less reputable sources, there is a risk of getting conterfeit or stolen components. Keep in mind that shipping times from China can be anywhere between one week and infinity.

Some optional components are not listed in the BOM. For instance, you might want put some feet under the board, using glue or screws (the PCB has 3mm holes for that purpose).

2.1.4 Checklist

PCB (see Section 2.1.2)

Components from BOM (see also Section 2.1.3)

Tools for soldering: soldering iron, solder, pliers, wire cutters (see Section 2.1.1)

Some kind of power supply (see Section 2.1.1)

Programmer or debugger, including the necessary cables (see Section 3.2)

Optional

Serial cable or USB-to-serial converter

Multimeter

Feet for the board (M2.5)

Screws (M2x18), nuts, washers, and spacers (10mm) for the LCD

2.2 Soldering

Take your time and go slow and methodically. Familiarise yourself with the board first (see Section 1.1). Have the schematic and BOM at hand so you can refer to them for component numbers and values.

Start with the resistors. These are the easiest components to solder and they're reasonably resistant to overheating. Make sure you choose a resistor with the correct resistance value (see BOM). For simple resistors (not resistor networks), the orientation doesn't matter.

Slot a resistor in from the front and bend the wires approximately 45° on the back, so it won't fall out. Turn the PCB around and solder it in from the back. Place the tip of your soldering iron so that it touches both the wire and the pad, then add some solder. The solder should adhere to both parts - if it doesn't, you either haven't heated both parts properly or you're missing flux. Don't use too mush solder: the ideal solder joint is tent-shaped, not a big blob. Don't worry if it doesn't come out perfect right away. Soldering is mainly a matter of practice. You can rework your first joints later once you've got a bit more experience. See Figures 11, 12, 13, and 14 for illustration.

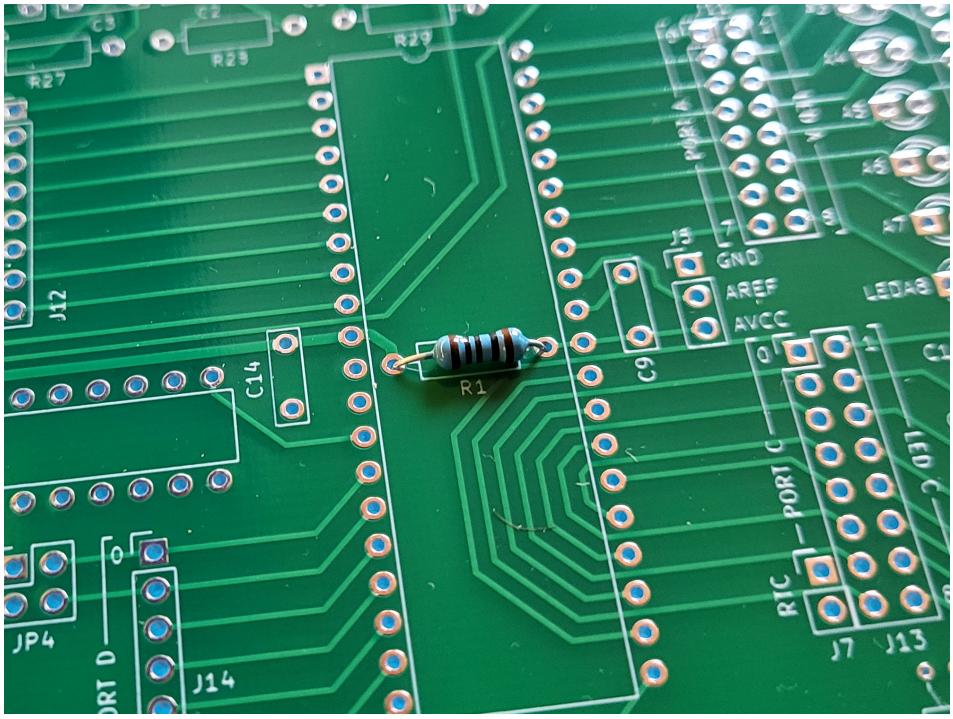


Figure 11: Step 1: Insert a resistor into the PCB

While the order in which you solder the components doesn't matter, it's probably a good idea to solder smaller components before taller, adjacent ones.

Caution

The single exception is R1 which is placed underneath the IC socket for U1. Before soldering, make sure they fit on top of each other. If not, you can place R1 on the back of the board instead.

2.2.1 Orientation of Components

For most components, the orientation is crucial. Only for resistors (but not resistor networks), ceramic capacitors (but not electrolytic ones), crystrals (but not crystal oscillators), and pin headers/sockets it doesn't matter. Components mounted in the wrong orientation can get damaged and/or damage others. They might also emit smoke and catch fire. Here are some clues for how to properly orient components:

- Diodes: The anode is marked by a ring and so is the icon on the board
- LEDs: The anode is on the round side of the casing and has the longer leg. It goes into the round pad of the PCB. The cathode is on the flat side of the casing and goes into the square pad of the PCB.

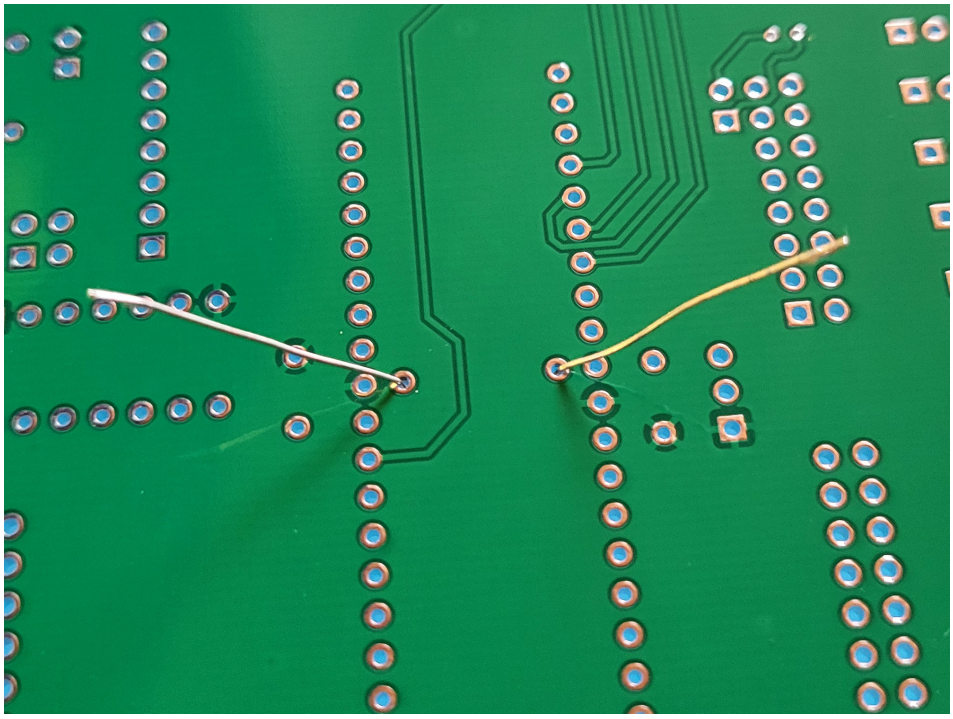


Figure 12: Step 2: Bend the wires so it can't fall out

- Electrolytic capacitors: The plus side (longer leg) goes into the square pad marked "+" of the PCB. The minus side (shorter leg) goes into the round pad of the PCB. The minus side usually has a white marking, as does the PCB.
  C7 is special, since it is axial ("lying down"). The correct orientation is determined by the groove.

- The resistor networks (RN1, RN2) consist internally of eight resistors with one end of each tied together (hence nine pins). The common pin (1) is usually marked with a small dot. If you're unsure, you can use a multimeter in resistance mode to verify which pin has 1.5kΩ to the eight other ones.

- ICs and IC sockets (U1, U2) have a half-circle marking on one side, as does the PCB. While the orientation of the socket technically doesn't matter, it is a good idea to solder it in this orientation to avoid later confusion.

- As sockets for crystal oscillators (Y1) are hard to come by, we use a standard 14-pin IC socket instead. In this case, the marking has no meaning, but be sure to plug the oscillator in the right way later (sharp corner at Pin 1).
  Shorten the legs if it stands out too much. When using add-on boards, make sure their underside contacts don't touch the case of Y1 (or any other conductive part for that matter).

- Bridge rectifier (B1): One of the four legs is longer and marked with a "+". This leg goes into the square pad on the PCB, also marked "+".

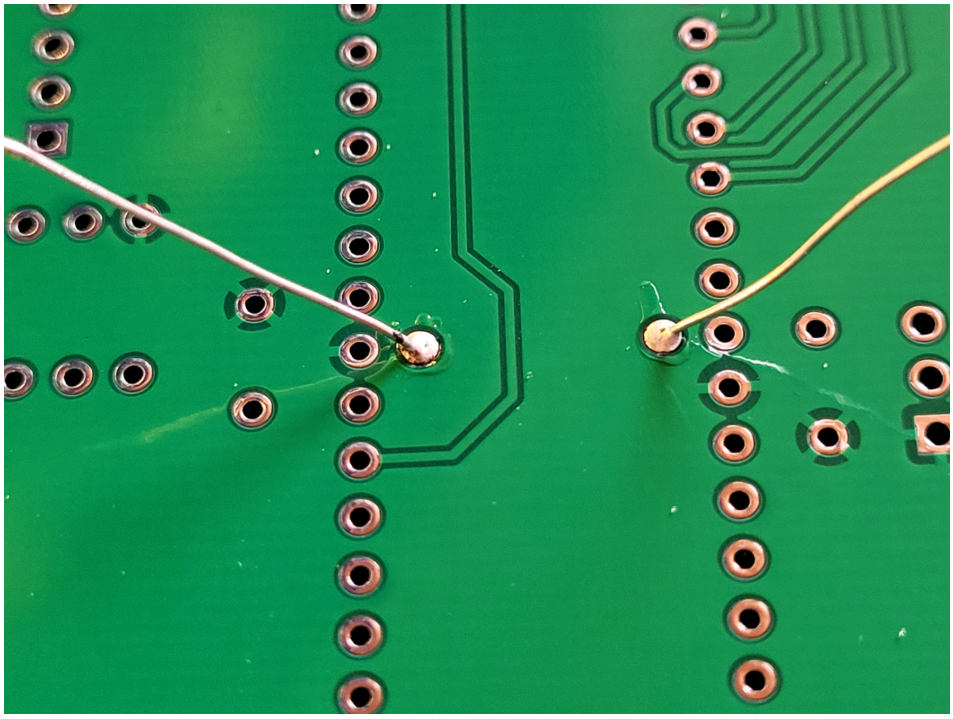


Figure 13: Step 3: Solder it in

- The voltage regulator (U3) needs to have its legs bent 90°. Use small pliers and make sure not to put physical stress onto the internals. The metal tab must lie on the exposed pad of the PCB (it functions as a small heat sink). You can solder the metal tab onto the PCB but if you're having trouble doing that, just screw it on. Don't risk damaging the regulator by soldering for too long.

After soldering, you can wipe away flux residue with some alcohol (isopropyl is best but ethanol works too).

2.2.2 Optional Components

The spare connector ("SPARE CONN") can be used to make 3-way connections or just as a repository for jumpers. It's convenient but not necessary. If you want to use it, solder an 8-long 2-row header and an 8-long 1-row header (or three 1-row headers) next to each other.

There are two kinds of LCD that fit onto the board, as indicated by "LCD OPTION A" and "LCD OPTION B". On one, the backlight power supply pins "A" and "K" are next to DB7, on others they are on the other side, next to VSS (see Figure 15 for some examples). Check which option fits your LCD and solder the 16-pin socket into the corresponding pads, leaving the remaining two pads free. Alternatively, you can use an 18-pin socket which allows you to switch between both types later.

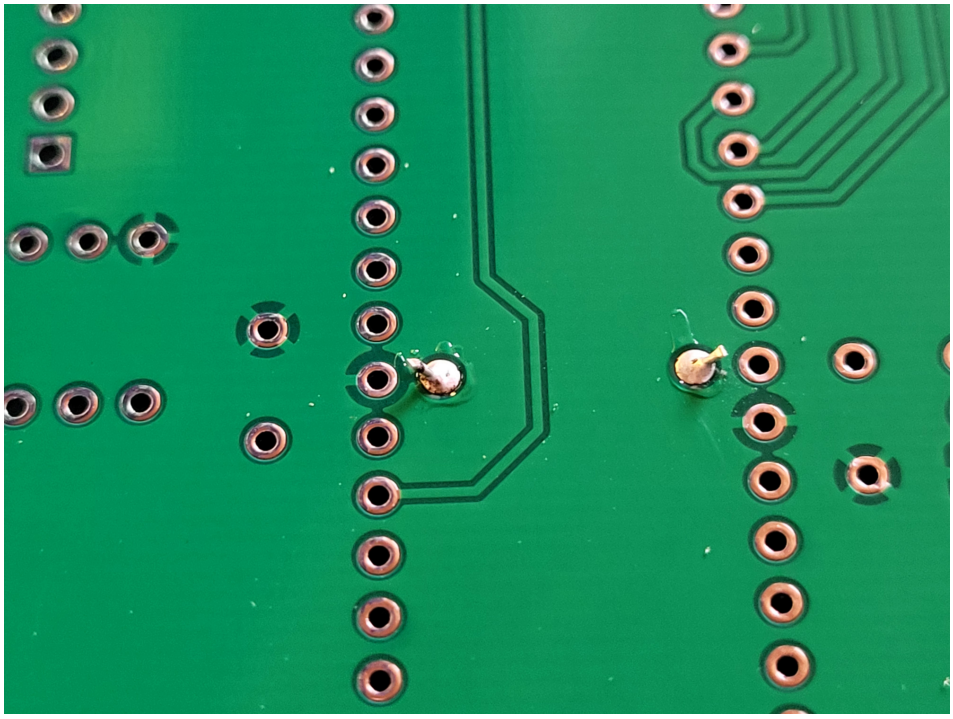


Figure 14: Cut off excess wire

Note

The pins “A” and “K” are sometimes called “L+” and “L-” or “LED+” and “LED-” or “BL+” and “BL-” or simply “15” and “16”.

Resistors R30 and R31 are only necessary if your LCD crashes during programming, see Section 3.4.1. Unless and until that turns out to be a problem, you can leave them out.

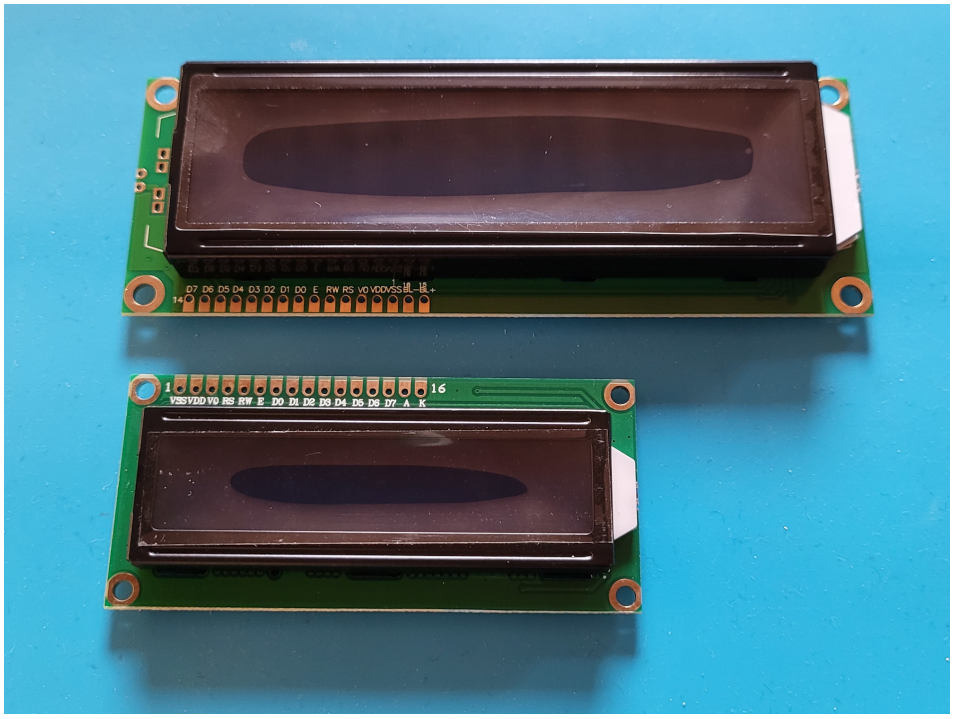


Figure 15: LCDs come in different sizes and with different pin layouts

2.3 Testing (Part 1)

If you have access to a multimeter, there are several things you can test before attaching the most vulnerable (and expensive) parts. Remove the ICs U1, U2, Y1 as well as the LCD from their sockets. Don't connect the power yet.

First, use resistance or continuity mode to check the main nets. As an example, take the GND net: while holding one multimeter probe to one of the GND connectors (e.g. the top row pins of J3) place the other probe on all the other GND points: the VSS pin of J16, the GND pins of J8 and J9, pins 11 and 31 of U1, pin 15 of U2, pins 1 and 7 of Y1, and many more. Use KiCAD's PCB Editor to identify all the points. Do the same for the VCC net and – if you want to be thorough – for all other nets, especially the one involving the programming connectors (J8 and J9).

Perhaps even more importantly, make sure that different nets like VCC and GND are NOT connected to each other.

If a connection is missing, identify the faulty solder joint and redo it.

Once you're satisfied, apply power to the board. LED1 (PWR) should light up. Check if anything gets hot. If so, unplug the power immediately and re-do the previous checks.

Put the multimeter into voltage mode (if necessary, select a range above 5V) and measure the voltage between VCC and GND. It should be near 5V. If it is significantly lower, disconnect the

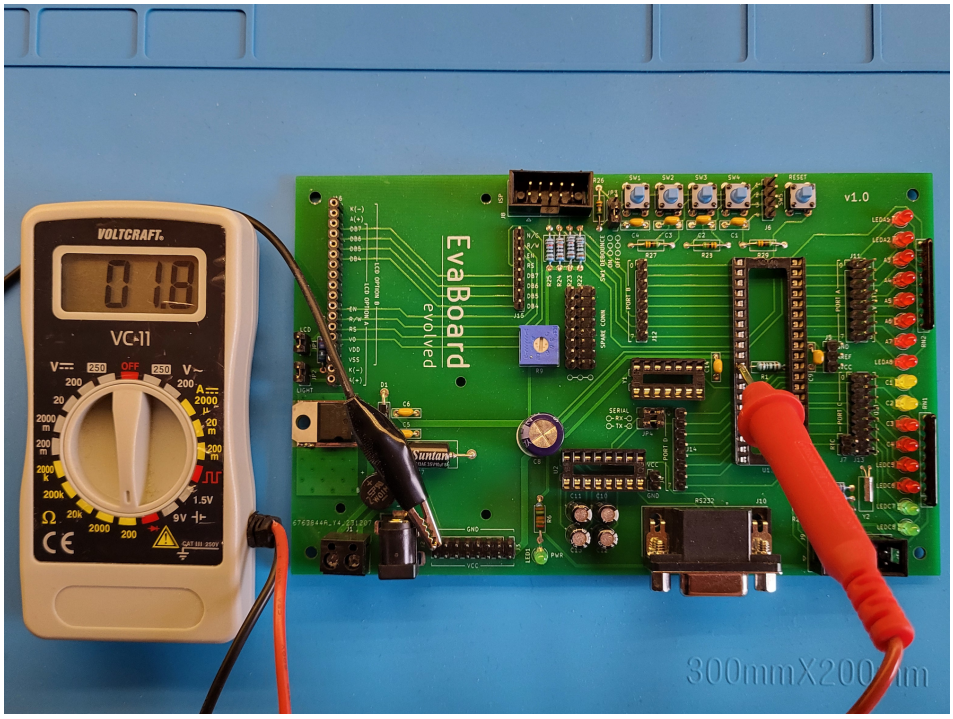


Figure 16: Continuity/Resistance Testing
(The 1.8Ω is due to bad multimeter calibration, in actuality it is almost 0Ω)

power and look for a short. Be careful not to cause a short with the probes while measuring.

You can manually test the LEDs using a jumper cable. Plug one end into one one of the LED pins (right column of J11 and J13) and connect the other end to GND (top row of J3). The LED should light up.

The buttons can also be tested manually: Again using a jumper cable, connect one of the LED pins and one of the button pins (J6). The LED should light up while the button is pressed.

Insert the LCD and mount jumpers on LCD (JP1) and LIGHT (JP2). Since no one is sending commands to it, it should show the default pattern: all pixels in the first line on, all pixels in the second line off. You will probably need to adjust the contrast on R9 with a screwdriver before the pattern becomes visible.

If everything looks good, unplug the power and finish the board by inserting the missing ICs. Mount a jumper on SW1 DEBOUNCE ON (left position of JP3).

Check VCC again. It should not drop significantly from before the ICs were inserted. If it does, unplug the power immediately and make sure the ICs are inserted the right way.

Finally, you can check the charge pump voltages: On Pin 2 of U2 you should get something between 8V and 10V (set the range of the multimeter accordingly). Pin 6 should have roughly the same voltage but negative.

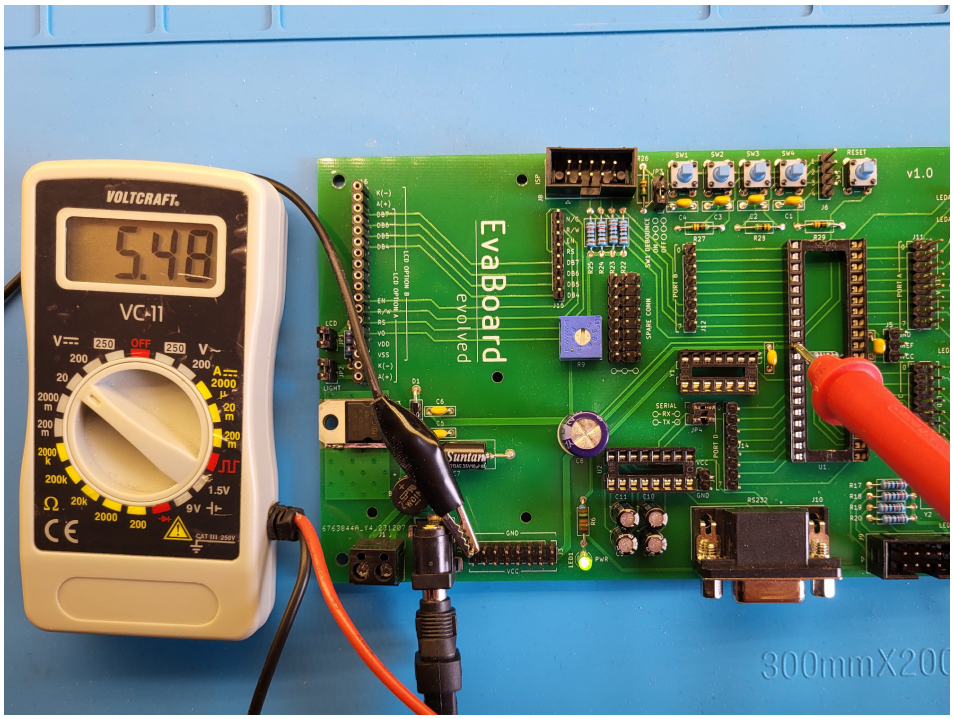


Figure 17: Voltage Testing (Again, the indicated $5.48V$ is in reality $5V$)

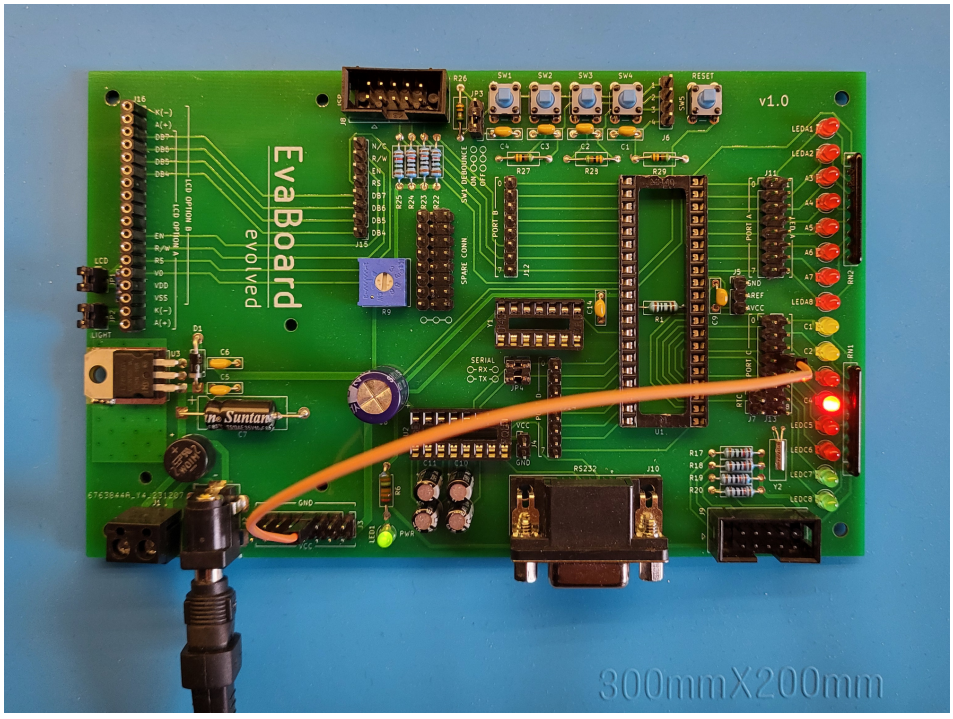


Figure 18: Testing the LEDs using a jumper cable

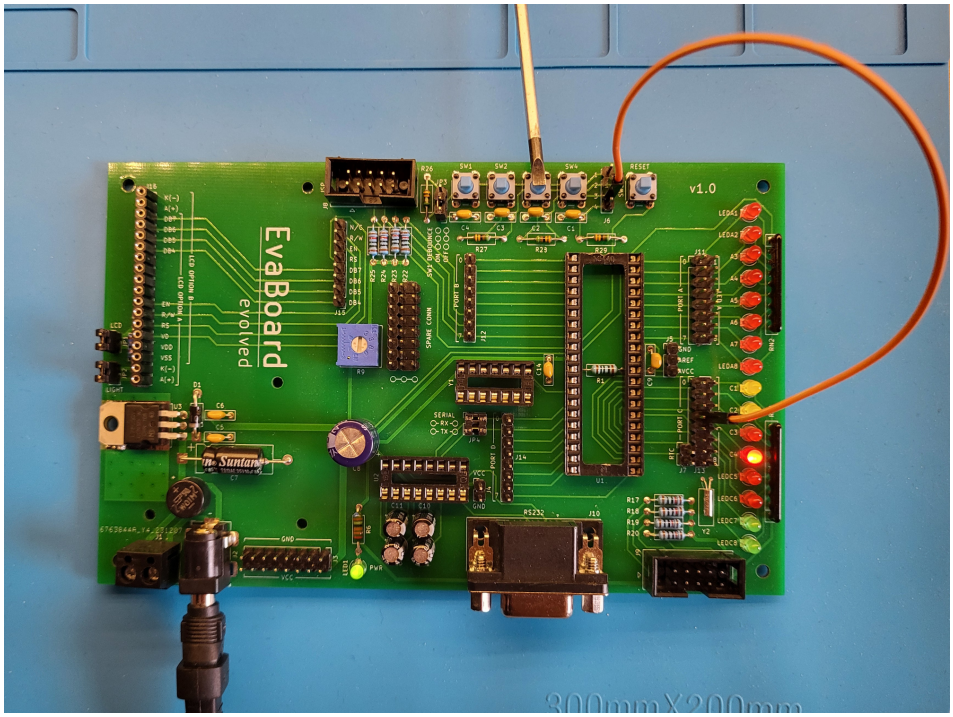


Figure 19: Testing the buttons using a jumper cable

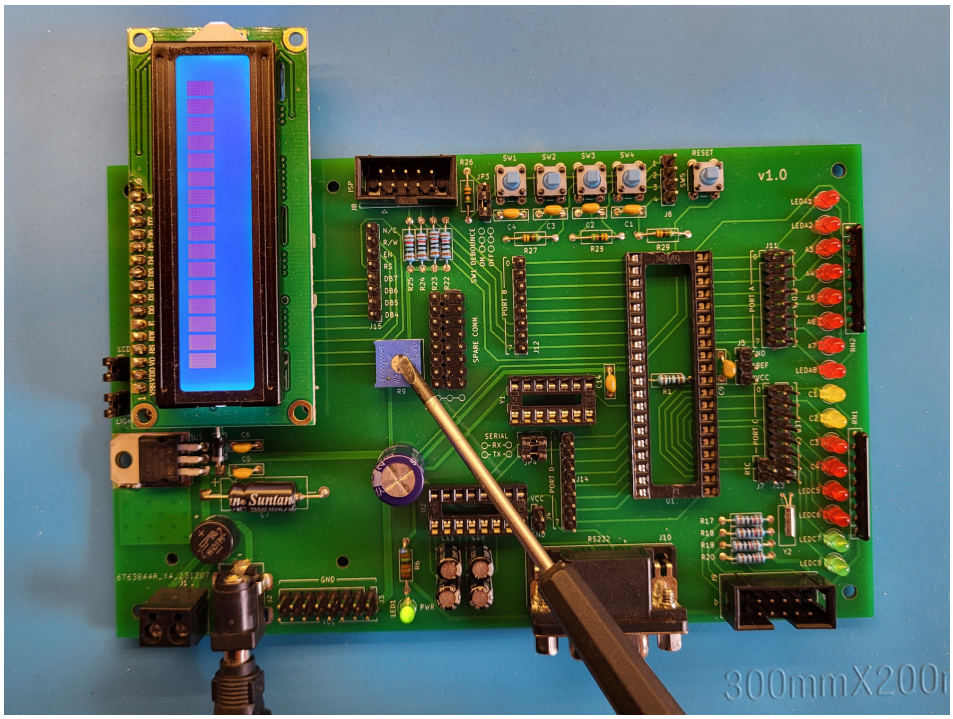


Figure 20: Setting the contrast voltage for the LCD

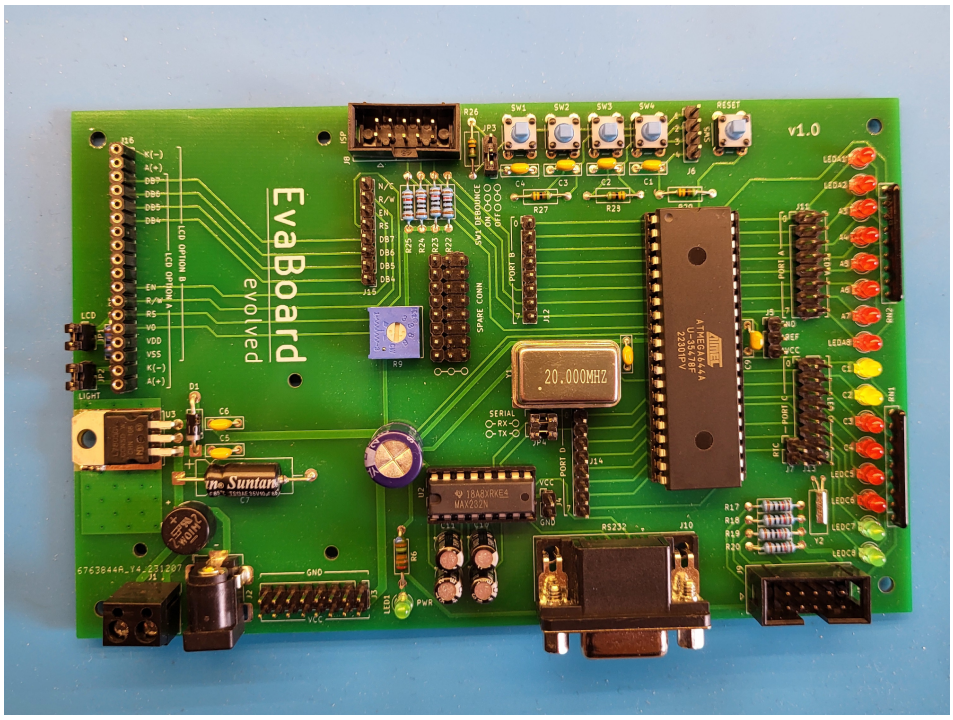


Figure 21: The finished board

3 Writing Firmware

3.1 Compilers and IDEs

Making firmware for microcontrollers requires two pieces of software: a compiler that translates C code into machine code (or an assembler to translate assembly code into machine code) and a programmer software which copies the machine code to the microcontroller's flash with the help of a piece of hardware, confusingly also called a programmer. Integrated Development Environments (IDEs) are optional but simplify the process.

Traditionally, Microchip Studio was used to make firmware for ATmegas. This is an IDE that includes both a compiler and a programmer. It used to be called Atmel Studio before Atmel was purchased by Microchip. Microchip Studio is based on Visual Studio and thus runs only on Windows. It is quite old, missing many functions of modern IDEs and probably won't receive substantial updates since Microchip pushes its own IDE.

MPLAB X is the modern IDE by Microchip. It is based on NetBeans, runs on Windows, Linux, and macOS and receives regular updates. Originally intended for Microchip's own microcontroller series ("PIC"), it now supports all the Atmel ones as well, including ATmegas. Like Microchip Studio, it comes with a compiler (XC8, to be downloaded separately) and a programmer which can be used through the IDE or as stand-alone software (MPLAB IPE).

Finally, there is also open source software for ATmegas:

- AVR-GCC, a C compiler (this is actually the compiler that's included in Microchip Studio nowadays)
- AVRA, an assembler
- AVRDUDE, a programmer software supporting many hardware programmers, both proprietary and open ones

Since these are all command line programs, they can be integrated into most IDEs (including Microchip Studio and MPLAB X which allows third-party programmers to be used through AVRDUDE). Some IDEs have pre-made plugins (e.g. AVR Eclipse plugin).

Note

"AVR" (probably from the inventors' initials) is the collective term used for several families of 8-bit microcontrollers. This includes the ATmega family, the ATtiny family, and many more. When searching for information online, the keyword "AVR" will probably bring up more useful results than "ATmega".

3.1.1 Example: Using AVR-GCC

On Linux, installing AVR-GCC is as simple as installing three packages. For example on Ubuntu:

```
sudo apt-get install gcc-avr avr-libc binutils-avr
```

For Windows, download[3] the AVR 8-bit Toolchain from Microchip. Extract it somewhere and

[3] The toolchain is also part of Atmel Studio, so if you've installed the latter, you're all set.

add the `bin` subdirectory to the PATH environment variable.

Create a file `blink.c` with the contents from Listing 1. This is the microcontroller equivalent of "Hello World", a minimalistic program which flashes an LED connected to Port A0.

Listing 1: blink.c for AVR-GCC

```
#include<avr/io.h>
#include<util/delay.h>

void main(void)
{
    DDRA = 1;
    while(1)
    {
        PORTA = 1;
        _delay_ms(500);
        PORTA = 0;
        _delay_ms(500);
    }
}
```

Run the following commands:

```
avr-gcc -DF_CPU=1000000 -Os -mmcu=atmega644 -o blink.elf blink.c
avr-objcopy -j .text -j .data -O ihex blink.elf blink.hex
```

The first command is the compiler and linker. The command line argument -Os tells the compiler to optimize for space and -mmcu identifies the microcontroller. The -D argument tells the compiler to define a global symbol `F_CPU` set to 1000000. This is the frequency in Hz and it is required by the `_delay_ms()` function.

The second command turns the output of the linker (`blink.elf`) into Intel hex format (`blink.hex`) which most programmers take as input.

3.1.2 Example: Using Microchip Studio

Microchip Studio is available for Windows only. Download it from the Microchip website and install it. During installation, you can uncheck everything except the AVR architecture.

Click "File", "New", "Project", select "GCC C Executable Project", and enter "Blink" as the project name. Next, choose the ATmega644 or ATmega644A from the ATmega device family.

The IDE will automatically create a main.c file with a main function template. Copy the contents of Listing 1 into the file. Select the "Release" configuration from the dropdown below the menu bar.

Right click on the "Blink" project in the Solution Explorer and go to "Properties". On the "Toolchain" page, go to "AVR/GNU C Compiler", "Symbols" and define a new symbol `F_CPU=1000000`.

Hit F7 to compile. The hex file is placed in the `Release` subdirectory of your project folder. See Figures 22, 23, 24, and 25.

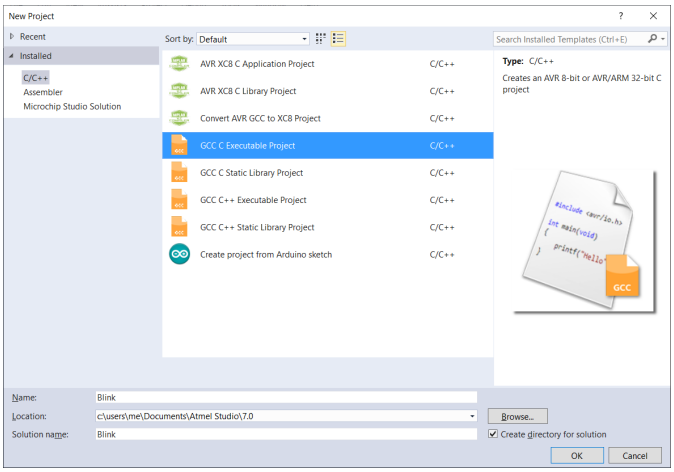


Figure 22: Creating a New Project in Microchip Studio (Step 1)

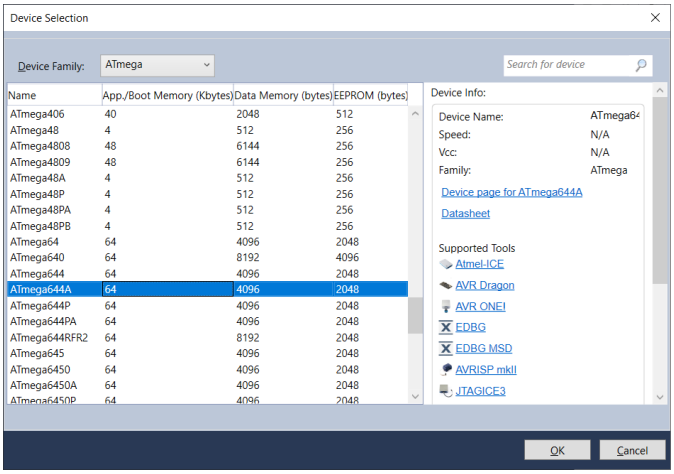


Figure 23: Creating a New Project in Microchip Studio (Step 2)

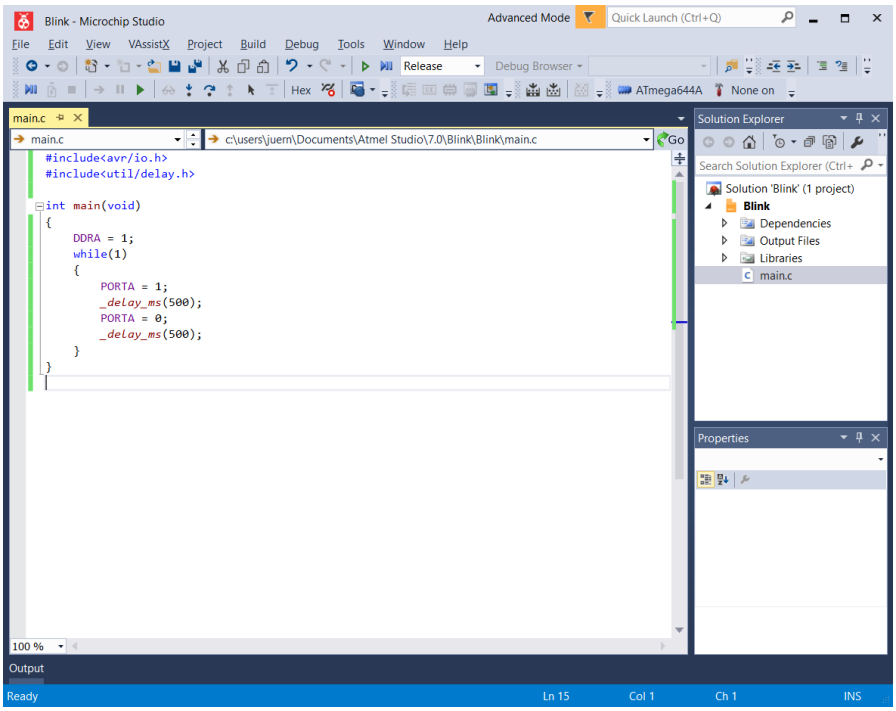


Figure 24: Creating a New Project in Microchip Studio (Step 3)

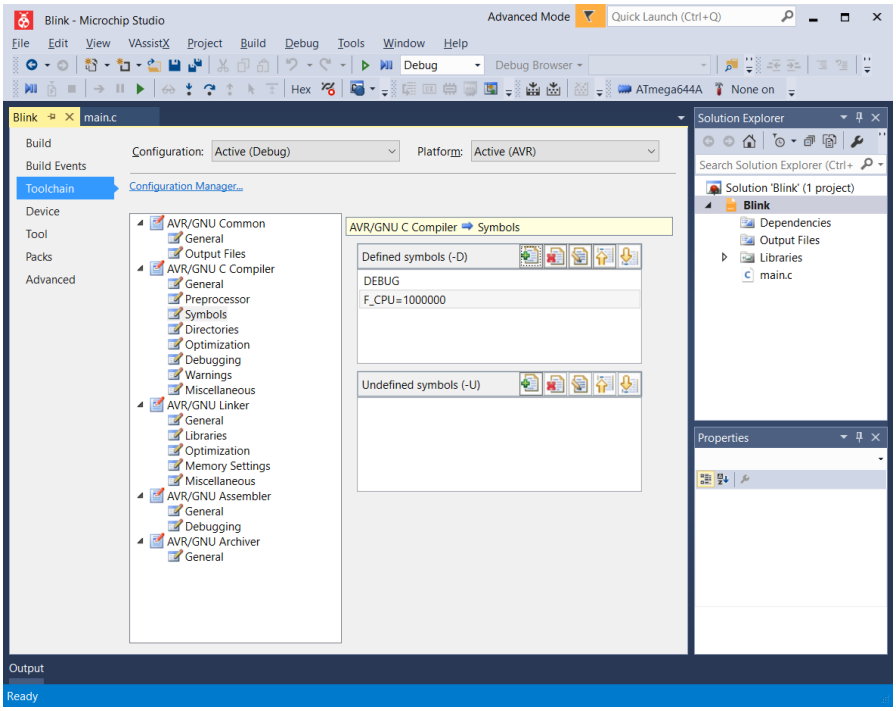


Figure 25: Creating a New Project in Microchip Studio (Step 4)

3.1.3 Example: Using MPLAB X

Download MPLAB X from Microchip's website and install it. On Windows, this is straightforward. On Linux, you have to unpack the installer, make it executable, and run it as root:

```
tar xf MPLABX-vX.XX-linux-installer.tar
chmod u+x MPLABX-vX.XX-linux-installer.sh
sudo ./MPLABX-vX.XX-linux-installer.sh
```

The Linux installer places the software in `/opt/microchip/`.

No compiler is included with the IDE, so you must choose which one(s) to install. One option is XC8, Microchip's compiler for all its 8-bit microcontrollers. However, the free version of XC8 has some limitations when it comes to optimization. Another option is AVR-GCC which MPLAB X should automatically detect if it is installed (see Section 3.1.1 for how install it). From a programmer's perspective, both compilers work roughly the same. Refer to the XC8 User's Guide for details.

Click "File", "New Project" and select "Standalone Project" from the "Microchip Embedded" category, then click "Next". Choose the ATmega644 or ATmega644A from the 8-bit AVR MCU category. If you are using an official Microchip programmer (see Section 3.2), you can select it here, otherwise choose "No Tool". On the next screen, select XC8 or AVR-GCC as the compiler, whichever one you installed previously. Finally, name your project "Blink" and specify a location. Clicking "Finish" generates the project. See Figures 26, 27, 28, and 29.

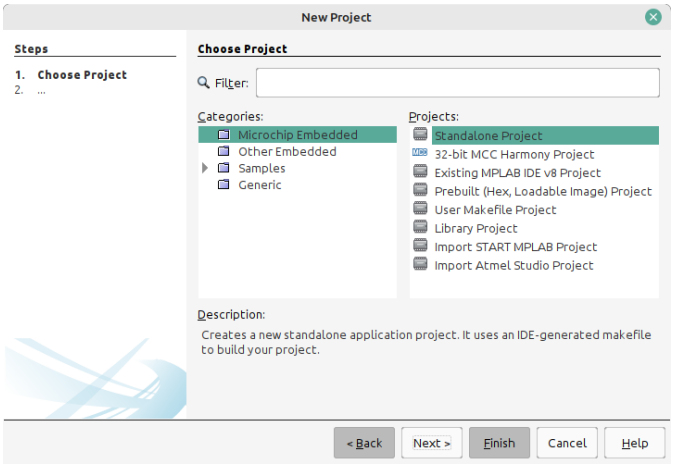


Figure 26: Creating a New Project in MPLAB X (Step 1)

In the "Project" tab on the left-hand side, right click the "Blink" project and go to "Properties". On the compiler page (either "avr-gcc" or "XC8 Compiler") enter `-DF_CPU=1000000` in the "Additional options" field and click "OK". See Figure 30.

Right click "Source Files" in the "Project" tab and select "New", "C Source File". Enter "blink" as the file name and click "Finish". Enter the code from Listing 1 into the empty file, like in Figure 31. Press F11 to compile. MPLAB X places the hex file in the `dist/default/production` subdirectory of the project folder.

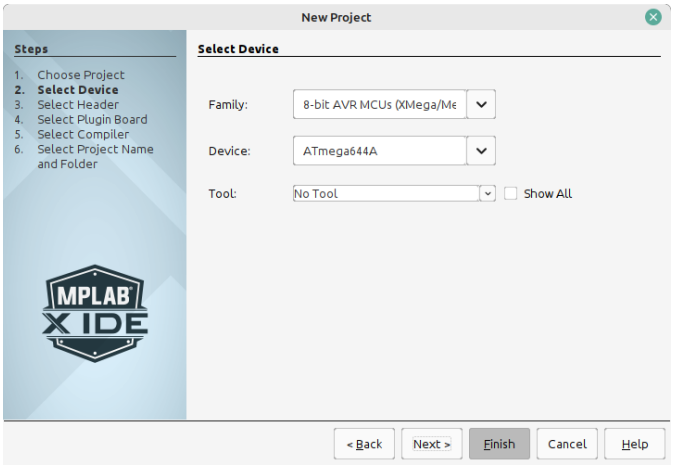


Figure 27: Creating a New Project in MPLAB X (Step 2)

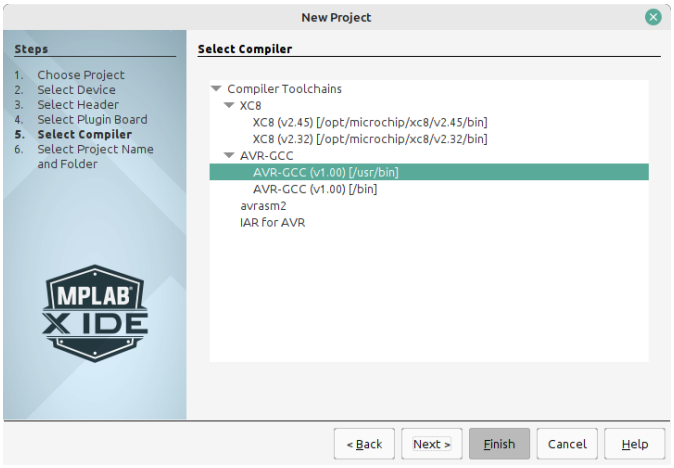


Figure 28: Creating a New Project in MPLAB X (Step 3)

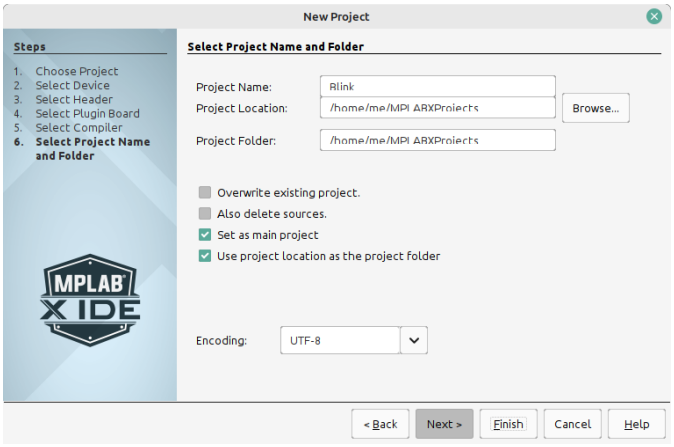


Figure 29: Creating a New Project in MPLAB X (Step 4)

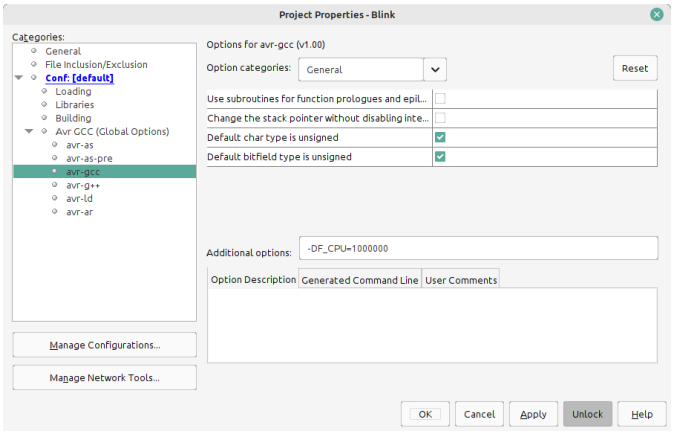


Figure 30: Creating a New Project in MPLAB X (Step 5)

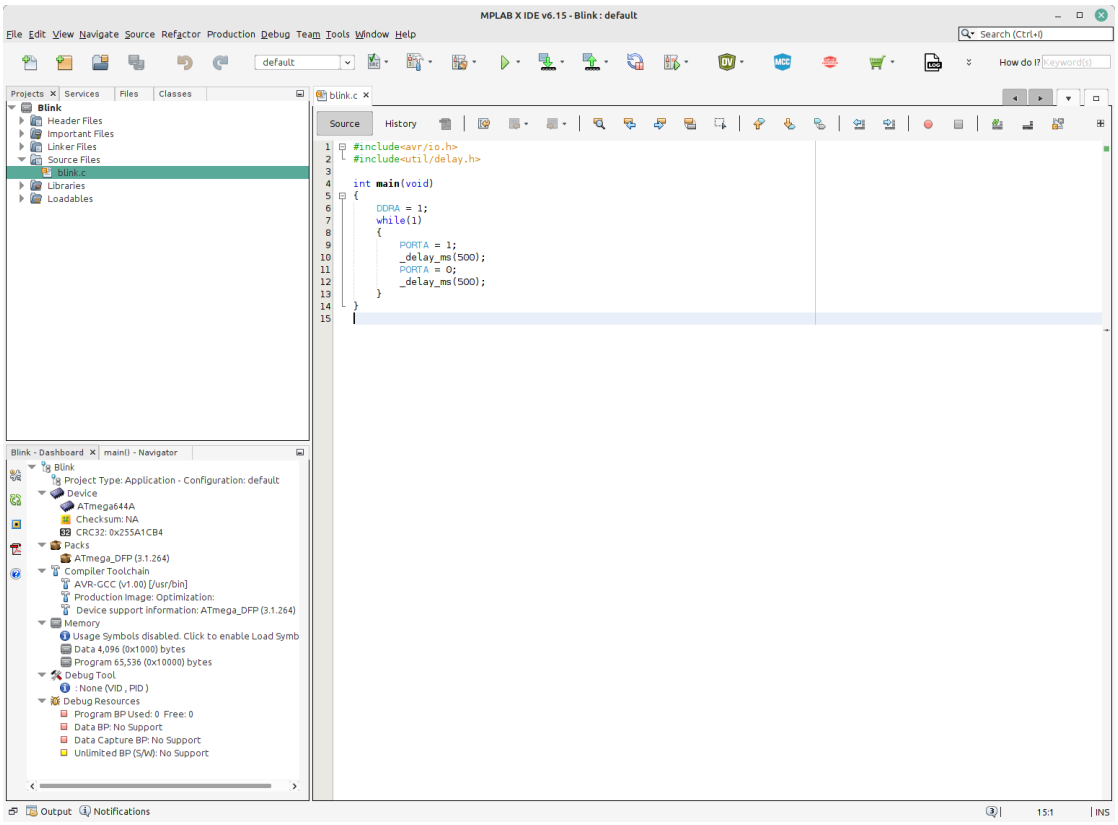


Figure 31: Creating a New Project in MPLAB X (Step 6)

3.2 Programmers and Debuggers

In this context, "Programming" (or "flashing") means copying machine code from your PC to the program memory (flash) of the microcontroller. In addition to being programmed, the microcontroller can also be debugged, which means the computer can halt the microcontroller's execution and examine the contents of its memory and registers. A debugger is necessary for that. Debuggers can also act as programmers, so there's no need to switch back and forth. Programmers tend to be a lot cheaper than debuggers and have wider software support.

The ATmega644(A) has two interfaces for programming: ISP and JTAG (connectors J8 and J9 on the board). JTAG can also be used for debugging, whereas ISP cannot.

There is a wide variety of ISP programmers available both from Microchip and third parties, including open source ones. Things to pay attention to when choosing a programmer: the connection to the computer (USB, serial, parallel port etc.), whether it provides power to the target microcontroller while programming (we don't want that since the board has its own power supply), and what programming software supports it.

- The STK500 is a rather dated programmer by Atmel which supports all ATmegas. It uses a serial connection to the PC and can provide power to the target (optional and with configurable voltage). It is supported by Microchip Studio and AVRDUDE. While it costs over 100€, there is a market for used ones. Its schematic and communication protocol are public. Some open source programmers are based on it and use the same protocol, although few offer the full range of functions of the STK500.
  To some extend, the STK500 can itself be used as an evaluation board: It has sockets for different AVR microcontrollers, LEDs, buttons, and a serial port (but no LCD).
  Despite its age, the STK500 comes with one major benefit: In addition to ISP, it is capable of high-voltage programming (HVP). For HVP, the microcontroller must be removed from the board and placed onto the STK500 (ISP stands for "in system programming" and HVP is not "in system"). HVP works even if the microcontroller is bricked due to wrong fuse settings. See Section 3.3 for more details on fuses.

- The AVRISP mkII is slightly newer and features a USB port. It provides no target power and is supported by Microchip Studio and AVRDUDE. It uses a 6-pin rather than a 10-pin connector, so you will need an adapter (some male-to-female jumper wires will also do).
  The AVRISP mkII is no longer in production, but many clones are available, ranging from 20€ to 30€. Some even come with an additional 10-pin header.

- One of the cheapest (and open source) options is USBasp. You can get one for less than 5€ on eBay or similar marketplaces. It uses USB1.1 low-speed (don't worry, your computer's USB2.0 is downward-compatible) and usually comes with a 10-pole ribbon cable that fits into the ISP connector (J8) of the evaluation board. The available models differ slightly, depending on the manufacturer. Make sure to choose one that lets you disable target power. Most have a three-pin (two-way) jumper for setting the target voltage to 5V (jumper right), 3.3V (jumper left) or off (no jumper). USBasp is supported by AVRDUDE.

Caution

Do not buy the "USBasp" model that comes in a coloured metal case (sometimes with fraudulent Atmel branding). Those are not in fact USBasp clones and they don't work with AVRDUDE.

- Arduino is a popular microcontroller platform for hobbyists. Arduino boards consist of a microcontroller (often an ATmega328P) and a USB-to-serial converter. The microcontroller has a bootloader which allows it to be programmed directly from USB without the need for separate programmer hardware. Some Arduinos can be turned into ISP programmers for other ATmegas (supported by AVRDUDE). Suitable Arduino boards like Uno or Nano are 10€ to 20€.

- There are many other options. Before purchasing, check which software a particular programmer is supported by.

Debuggers are significantly more expensive:

- The cheapest one is probably the MPLAB SNAP by Microchip at around 65€. It connects via USB, has both ISP and JTAG, cannot supply target power, and is supported by MPLAB X and Microchip Studio (newer versions of AVRDUDE can use it as a programmer). Being Microchip's most low budget product, it comes with no case and is sometimes a bit unstable (expect the occasional USB disconnect). You will need an adapter for the 10-pin header of the evaluation board or a couple of male-to-female jumper wires.

- More expensive products like PICkit or ICD are more stable and offer optional target power. Like the SNAP, they were made for Microchip's PIC series and AVR support is more like an afterthought (again, adapter or jumper wires necessary). They all work with MPLAB X, some with Microchip Studio, and a few are supported by AVRDUDE as programmer only.

- Atmel's original JTAG adapter was the now discontinued JTAGICE3. Caution: The predecessors (JTAGICE mkI and mkII) do not support the ATmega644(A). The only ICE adapter still in production is the ATMEL-ICE at 200€ to 300€. Both JTAGICE3 and ATMEL-ICE are supported by Microchip Studio and MPLABX. AVRDUDE can use them as programmers.

Note

We recommend that, as a beginner, you go with the cheapest available ISP programmer. Not only will that still get you pretty far, you also won't burn serious money, should you somehow manage to fry it.
Not having a debugger is less problematic than you might think. Hardware debugging is different from software debugging and comes with serious limitations. For example, while a debugger can halt the ATmega, other connected ICs keep running. Debugging also tends to interfere with interrupts, making it hard to reproduce problems in a debugging environment. Often the easiest way to debug firmware is to insert print commands that write values of interest to the LCD or the serial port.

Due to the many possible combinations of programming hardware and software, we cannot give detailed instructions for every single one of them. Instead, the following subsections will pick a few examples and explain for each one how to program the `blink.hex` file from Section 3.1 onto the microcontroller.

Prepare the board by connecting Pin A0 to LEDA1 using a jumper or jumper wire. Apply power to the board.

3.2.1 Example: Programming with USBasp and AVRDUDE

Remove the target power jumper (see Figure 32) from the USBasp. Connect it to your computer via USB and to the board's ISP connector J8 with the ribbon cable.

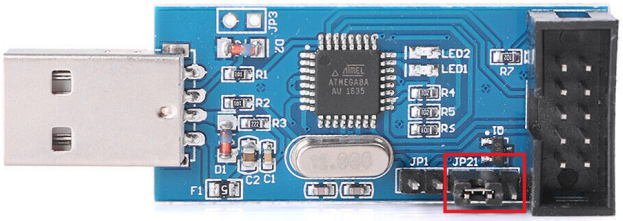


Figure 32: USBasp Programmer with Target Voltage Selection Jumper Highlighted

On Linux, all you have to do is install AVRDUDE via the package manager, e.g. on Ubuntu:

```
sudo apt-get install avrdude
```

On Windows, USBasp will be listed as an unrecognised device in the Device Manager, like in Figure 33.

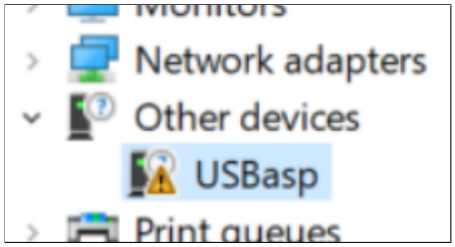


Figure 33: Windows Device Manager before driver installation

Download Zadig and run it. Select USBasp and libusb as shown in Figure 34, then hit "Install Driver".

Now the device manager should have USBasp as an "Atmel USB Device" like in Figure 35.

Next, download AVRDUDE and unpack it somewhere (for example `C:\Program Files\avrdude`). In the system settings, add the directory to the PATH environment variable. Open a terminal window and type `avrdude` to check whether Windows finds the program.

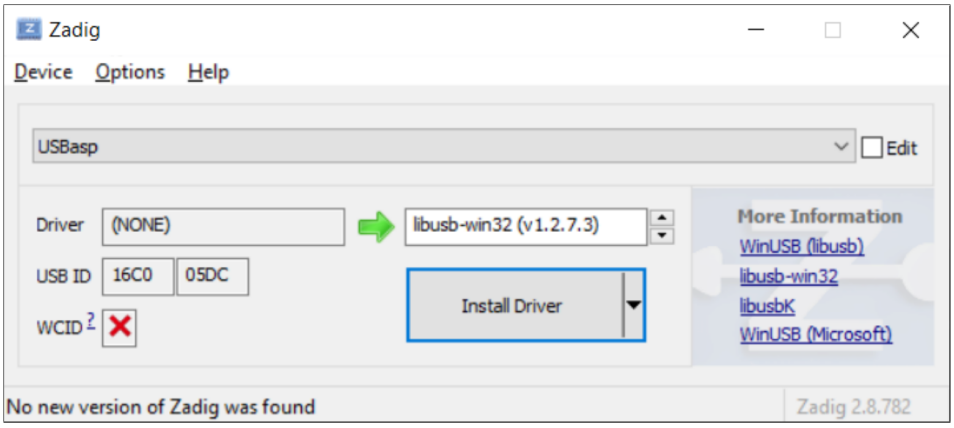


Figure 34: Install Windows Driver for USBasp

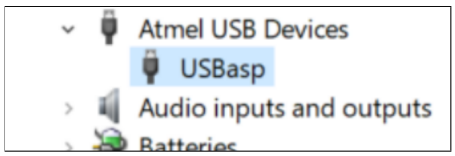


Figure 35: Windows Device Manager after installation

On either platform, check if there is a connection to the ATmega:

```
avrdude -c usbasp -p m644 -v
```

Change to the directory containing the `blink.hex` file, then flash it:

```
avrdude -c usbasp -p m644 -U flash:w:blink.hex:i
```

If the LEDs starts blinking, you have successfully deployed your first microcontroller program.

3.2.2 Example: Programming with USBasp and Microchip Studio

While Microchip Studio doesn't recognise third-party programmers directly, it can be configured to use AVRDUDE. First, make sure that AVRDUDE works from the command line, see Section 3.2.1.

1. Go to Tools → External Tools and add a new item
2. As shown in Figure 36, choose `avrdude.exe` for the command (in whichever directory you unpacked it).
3. Set the arguments to

   ```
   -c usbasp -p m644 -U flash:w:$(ProjectDir)Release\$(TargetName).hex:i
   ```

4. There should now be a new item in the Tools menu.

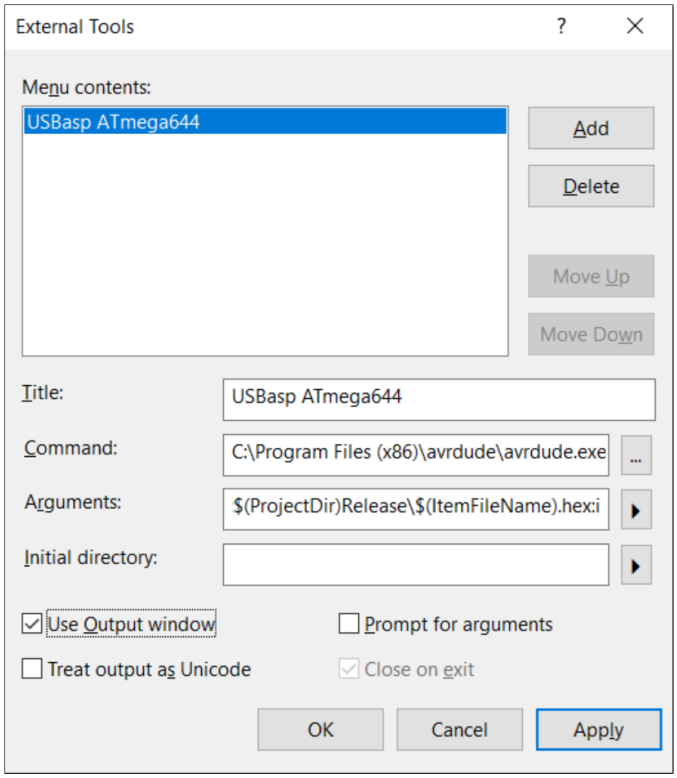


Figure 36: Adding a programming option for USBasp to Microchip Studio

Note

In your projects, you have to select the "Release" target instead of "Debug". Or change the arguments accordingly.

3.2.3 Example: Programming with USBasp and MPLAB X

Like Microchip Studio, MPLAB X also doesn't recognise third-party programmers. There is no option to add a new menu item for an external tool, but you can specify a command to be executed after building. Unfortunately, this has to be done for every single project.

AVRDUDE must be set up as described in Section 3.2.1.

Right-click on the project and go to "Properties". On the "Building" page, check "Execute this line after build" and enter

```
avrdude -c usbasp -p m644 -U flash:w:${ImagePath}:i
```

into the text field. See Figure 37.

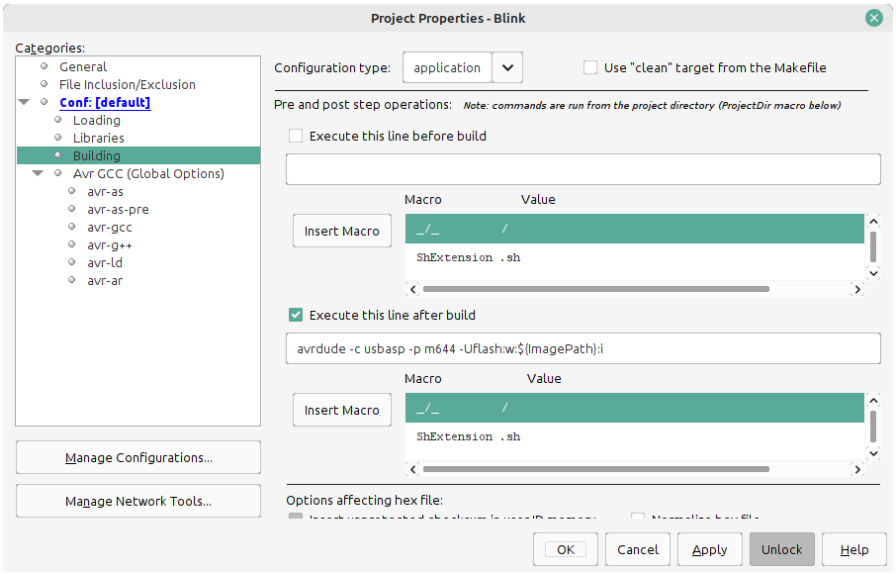


Figure 37: Adding a programming option for USBasp to MPLAB X

3.2.4 Example: Programming with Arduino and AVRDUDE

For this, you need an Arduino, a 10µF capacitor, and some jumper wires. Several types of Arduino work, but it must be 5V tolerant. We recommend an Arduino Uno or Nano based on the ATmega328P.

Install the Arduino IDE and connect the Arduino via USB. Your operating system should recognise the Arduino as a (virtual) serial port. The port is something like COM? on Windows or /dev/ttyUSB? on Linux, where the '?' is some number.

From the "Tools" menu, select the board and processor you're using, as well as the port that the Arduino board is connected to, see Figure 38. In the "File" menu, go to "Examples" and open "ArduinoISP". Click on the "Upload" button to program the Arduino. This is all the Arduino IDE is needed for, you can uninstall it now.

Use five jumper wires to connect the Arduino to the evaluation board. They need to be male-to-female for the Uno and female-to-female for the Nano. Figure 39 lists the necessary connections and shows the pins of the ISP connector J8. You also need to put a 10µF capacitor between one of the Arduino's reset pins ("RST") and GND. On the Uno, you can just stick it into the pin sockets. On the Nano, solder it to the pins or use female-to-female jumper cables. If you are using a polarised (electrolytic) capacitor, pay attention to the correct orientation. See Figure 40 for an example.

Now you can program with AVRDUDE (insert the serial port after the `-P` argument):

```
avrdude -c arduino -P <Serial Port> -b 19200 -p m644 -U flash:w:blink.hex:i
```

Just like in Sections 3.2.2 and 3.2.3, you can integrate AVRDUDE into your IDE.

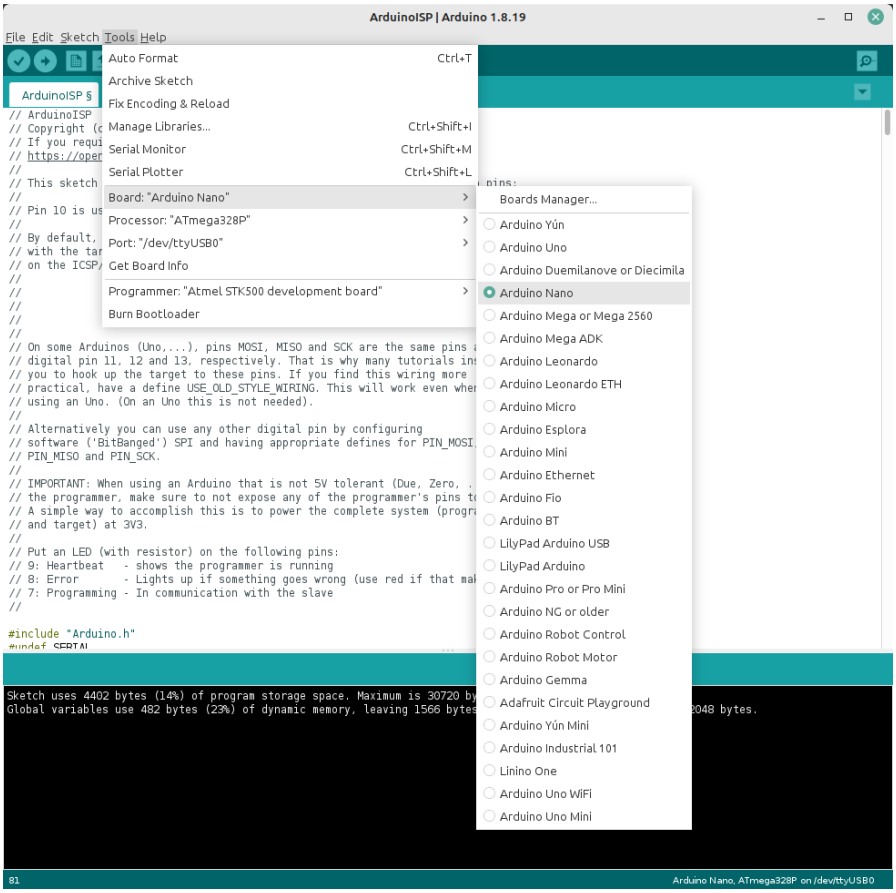


Figure 38: Programming an Arduino Nano to be an ISP Programmer

Arduino	ISP connector (J8)
D10	5 (RESET)
D11 (MOSI)	1 (MOSI)
D12 (MISO)	9 (MISO)
D13 (SCK)	7 (SCK)
GND	10 (GND)

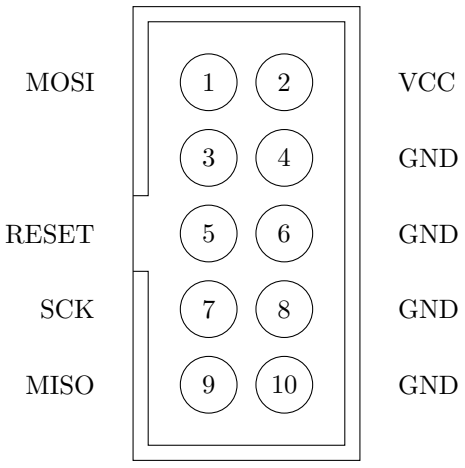


Figure 39: Connections between the Arduino and J8 on the Evaluation Board

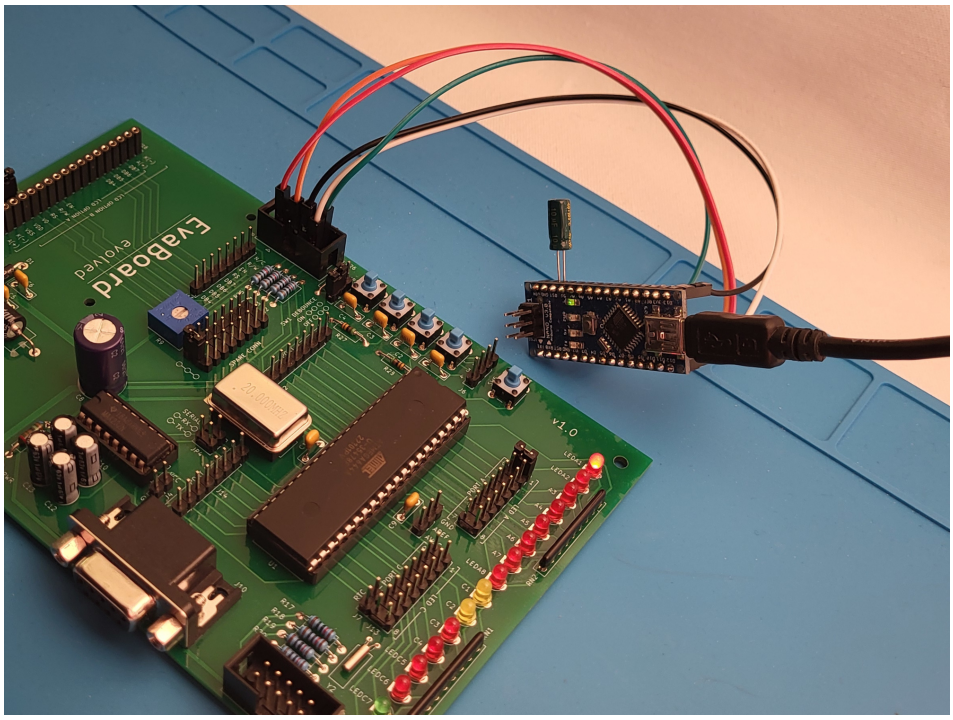


Figure 40: Arduino Nano as ISP Programmer

3.2.5 Example: Programming with STK500 and Microchip Studio

The STK500 needs its own power supply. In theory you can power the evaluation board from the STK500, however, the amount of current the STK500 can provide to a target is limited. The better option is to have two power supplies, one for the STK500 and one for the evaluation board. In this case, you need to remove the VTARGET jumper, see Figure 41.

The STK500 requires a serial connection to the computer. If you are using a USB-to-serial converter, make sure it is properly recognised by Windows. Use the Device Manager to find out which COM port the STK500 is connected to.

Use a 10-pole ribbon cable to connect ISP10PIN on the STK500 and the ISP connector (J8) on the evaluation board. Make sure to plug in the cable in the correct way – the STK500 has Pin 1 labelled and on the evaluation board Pin 1 is marked with a small triangular arrow.

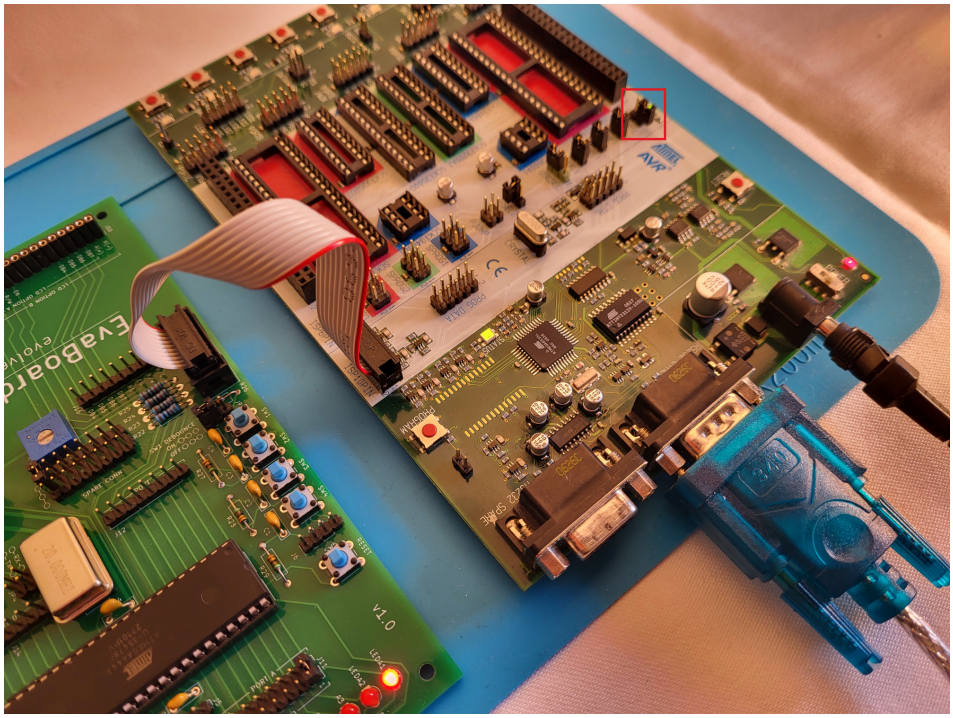


Figure 41: ISP Programming with STK500

Start Microchip Studio and open the Blink project from Section 3.1.2. In the "Tools" menu, click "Add Target". Select STK500 and specify the correct COM port, see Figure 42. Open the "Device Programming" dialog from the "Tools" menu. Select the STK500 entry in the Tool dropdown and make sure the ATmega644(A) is selected under "Device" and ISP under "Interface". Click "Apply", then have it read the device signature to check whether communication can be established. See Figure 43

In the project properties, go to the "Tool" page and select the STK500 and ISP, see Figure 44. Now you can program the microcontroller by clicking on the "Start without Debugging" icon in the toolbar, see Figure 45.

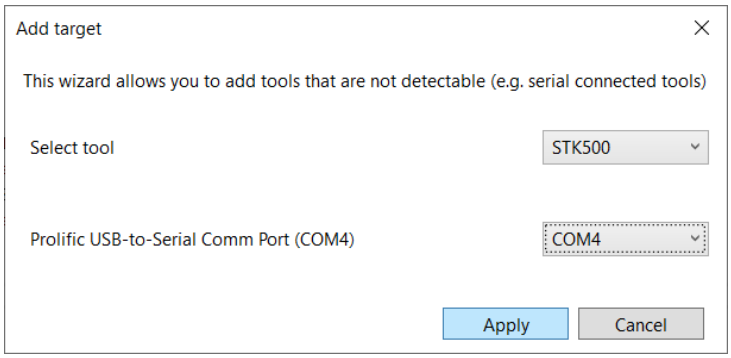


Figure 42: Adding the STK500 as a Target in Microchip Studio

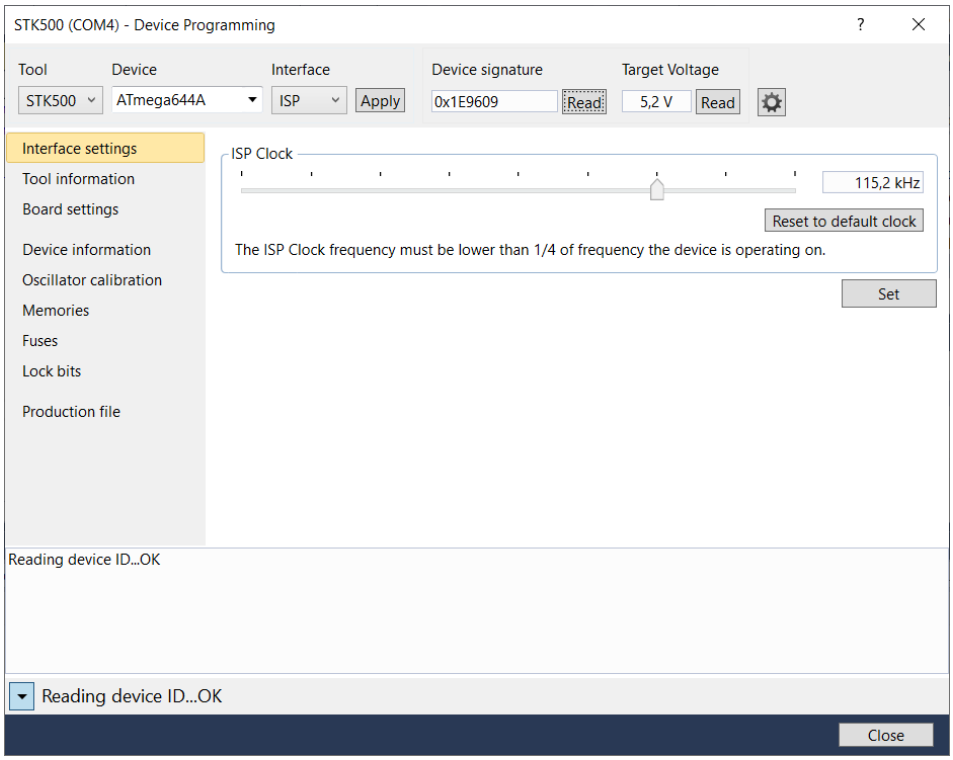


Figure 43: Settings for STK500 in Microchip Studio

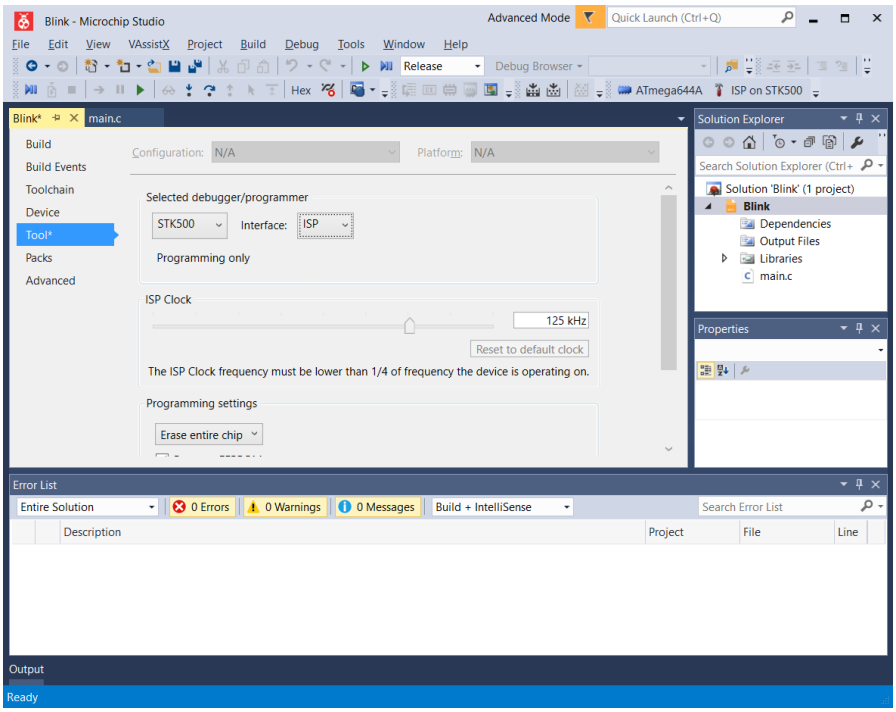


Figure 44: Project Settings for Programming with the STK500 in Microchip Studio

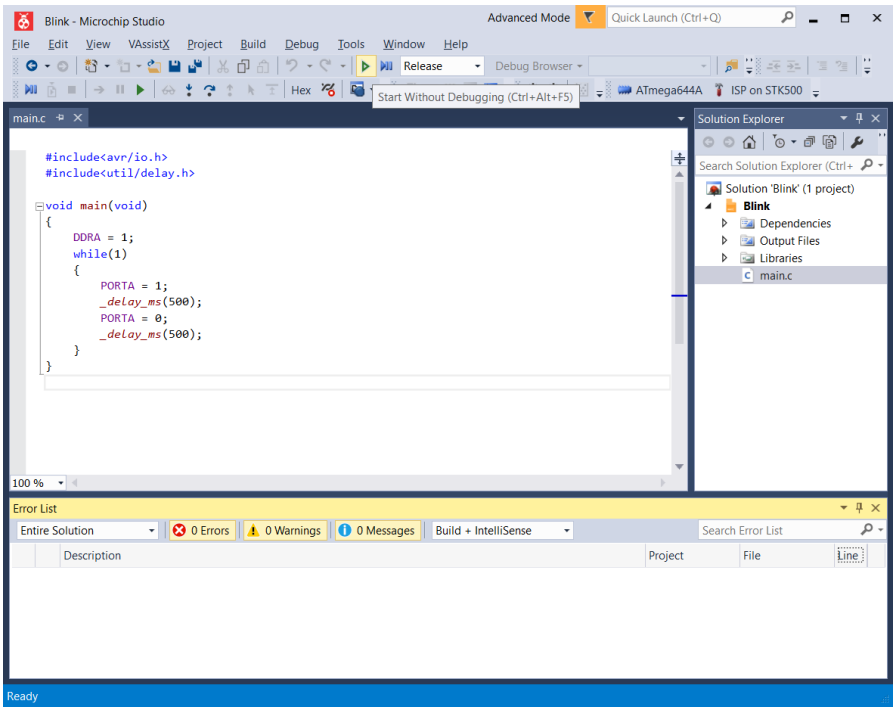


Figure 45: Programming with the STK500 in Microchip Studio

3.2.6 Example: Programming with MPLAB SNAP and MPLAB X

The SNAP supports both ISP and JTAG. In this example we are only looking at JTAG.

The SNAP has an 8-pin socket to connect it to the target. The pin layout is made to fit standard programming connectors of PIC microcontrollers, so it cannnot be plugged directly into J9. Adapters from the 8-pin connector to the AVR JTAG 10-pin connector are available, but you can also just use seven male-to-female jumper cables. Figure 46 lists the connections and shows the pin layout of the JTAG connector (J9). See also Table 10-4 of the SNAP User's Guide. Pins 1..8 on the SNAP are not individually labelled, but Pin 1 is marked by a small triangular arrow.

Caution

The SNAP has a jumper that switches between "AVR" and "PIC", see Figure 47. Make sure it is set in the "AVR" position.

MPLAB SNAP	JTAG connector (J9)
1 (TVPP)	–
2 (TVDD)	4 (VCC)
3 (GND)	2 (GND)
4 (PGD)	3 (TDO)
5 (PGC)	1 (TCK)
6 (TAUX)	6 (RESET)
7 (TTDI)	9 (TDI)
8 (TTMS)	5 (TMS)

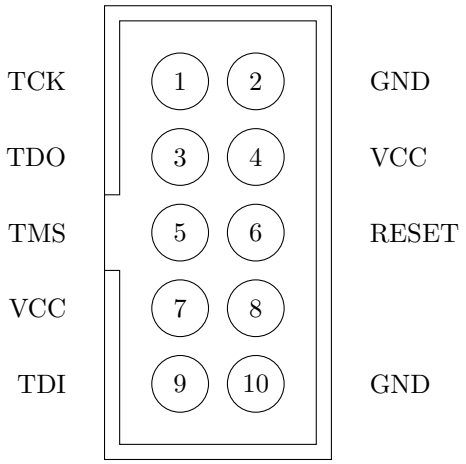


Figure 46: Connections between the MPLAB SNAP and J9 on the Evaluation Board

Open the Blink project from Section 3.1.3 in MPLAB X. Open the project properties and on the "Conf" page, choose the SNAP from the tool dropdown. Once you hit "Apply", a new page named "Snap" appears. On it, go to the "Communication" category and make sure "JTAG" is the selected interface, see Figure 48. Hit the "Make and Program" icon to start the programming, see Figure 49.

Note

The SNAP can be a bit fickle at times. If programming doesn't work, unplug and re-plug the USB cable. Try using a USB port directly on your computer rather than going through a USB hub.

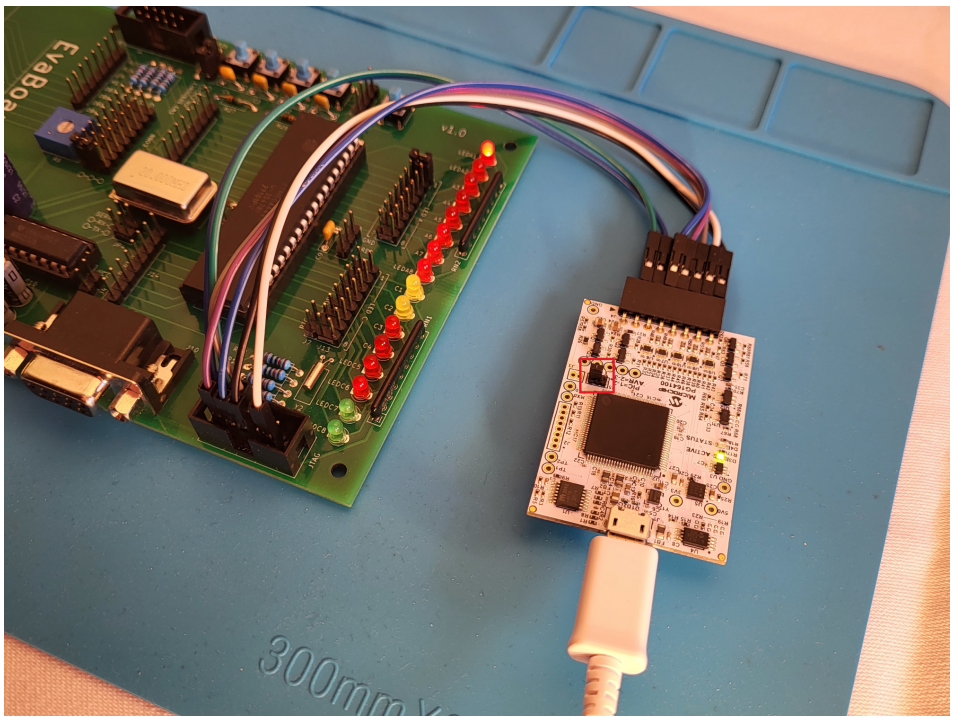


Figure 47: MPLAB SNAP as JTAG Programmer

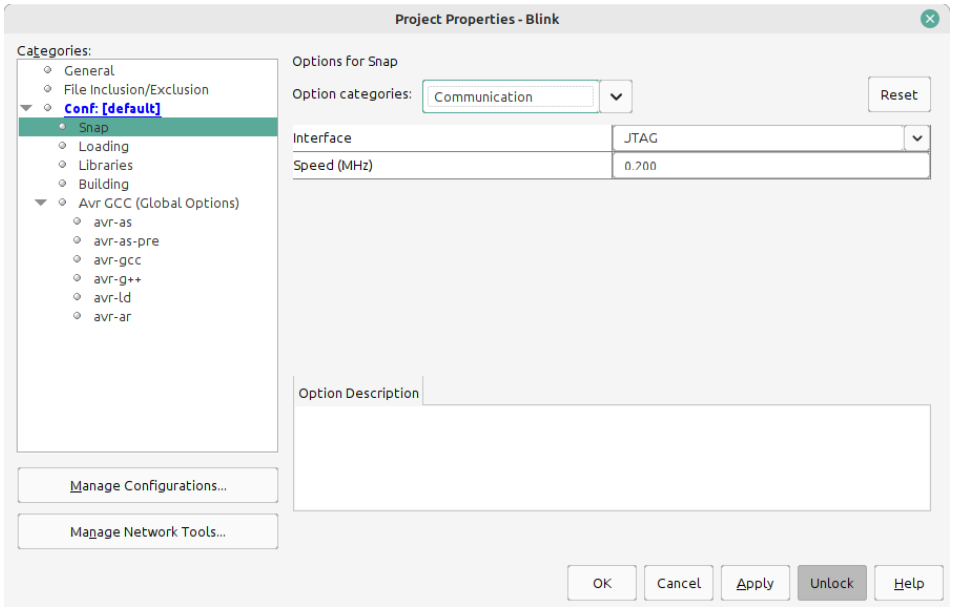


Figure 48: MPLAB X Settings for the SNAP

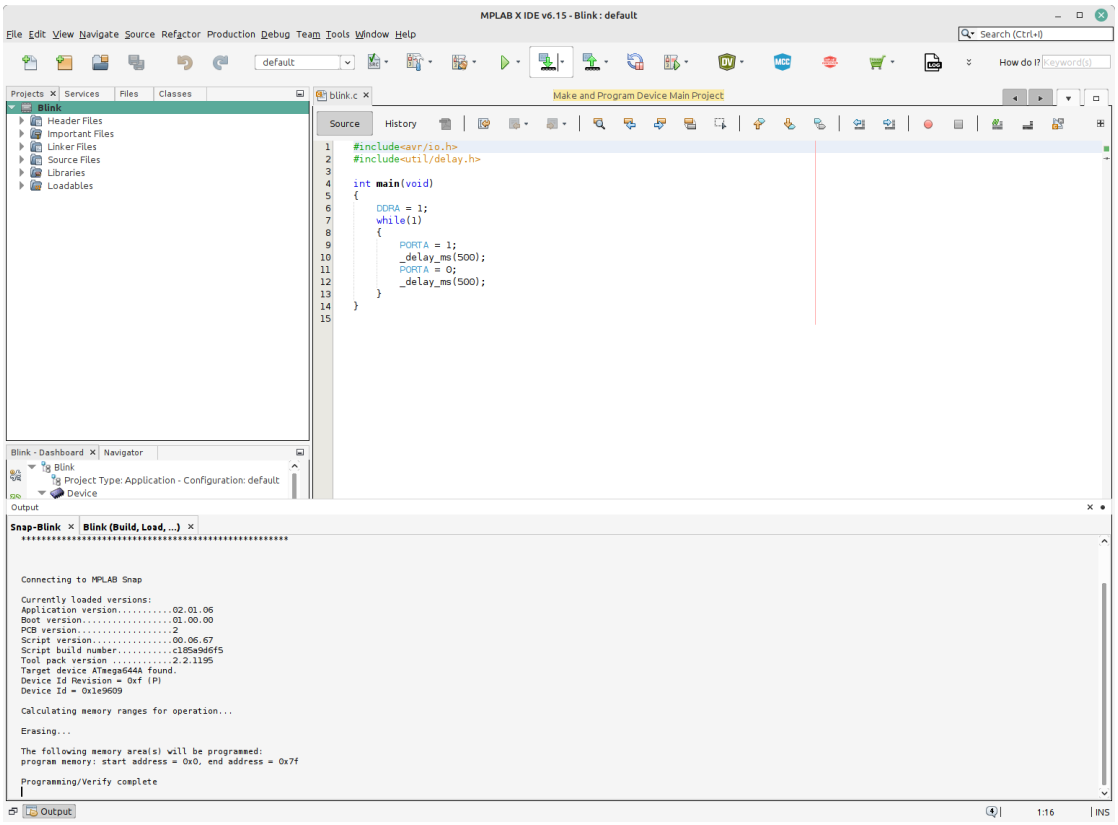


Figure 49: Programming with the SNAP from MPLAB X

3.3 Setting the Fuse Bits

You might have noticed that, in the blink example, we told the compiler that the microcontroller's CPU frequency was 1MHz, not 20MHz. This is because, in its factory settings, the ATmega is configured to use the internal 8MHz clock generator and divide that clock by 8. In order to make it use the external crystal oscillator Y1 and not divide the clock, we need to change these settings, called "fuse bits" on the ATmega.

Caution

Be very careful when doing this. Some fuse bits control the functioning of the programming pins. Setting them wrong can brick the microcontroller. When that happens, only high-voltage programmers (see Section 3.2) can reset it.

For more details on the fuse bits, see Section 25.2 of the datasheet. There are also online tools available, e.g. the AVR Fuse Calculator. For some reason the logic of the fuse bits is inverted. Note the weird language in the datasheet: it calls fuse bits "programmed" (meaning cleared, 0) and "unprogrammed" (meaning set, 1).

We need to set (unprogram) Bit 7 (CKDIV8) of the low fuse byte and clear (program) Bit 1 (CKSEL1). This means changing the low fuse byte from its default value of 0x62 to 0xe0.

In AVRDUDE this works as follows (using USBasp; similar for other programmers):

```
avrdude -c usbasp -p m644 -u -U lfuse:w:0xe0:m
```

Both Microchip Studio and MPLAB X support setting fuses through a graphical user interface. However, this only works when using an officially supported programmer. Neither IDE can set the fuses via AVRDUDE.

In Microchip Studio, open the "Device Programming" dialog from the "Tools" menu and go to the "Fuses" tab. Read the fuses, make the necessary changes, then write them back by clicking on the "Program" button. See Figure 50.

In MPLAB X, click "Set Configuration Bits" in the "Production" menu. First, click the "Read Configuration Bits" icon on the left. Make the necessary changes, then hit the "Program Configuration Bits" icon to write the modified values onto the chip. See Figure 51.

The blink example should now flash 20 times faster. After re-compiling with the `-DF_CPU=20000000` option, it should go back down to once per second.

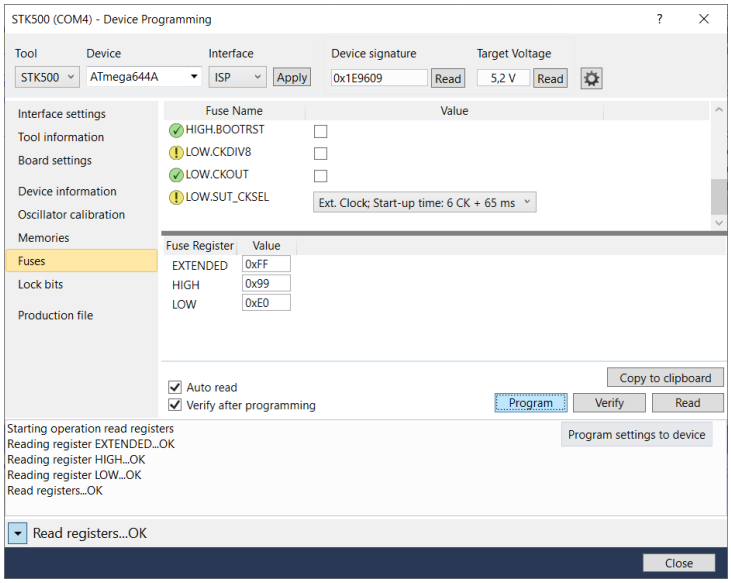


Figure 50: Settings Fuses in Microchip Studio

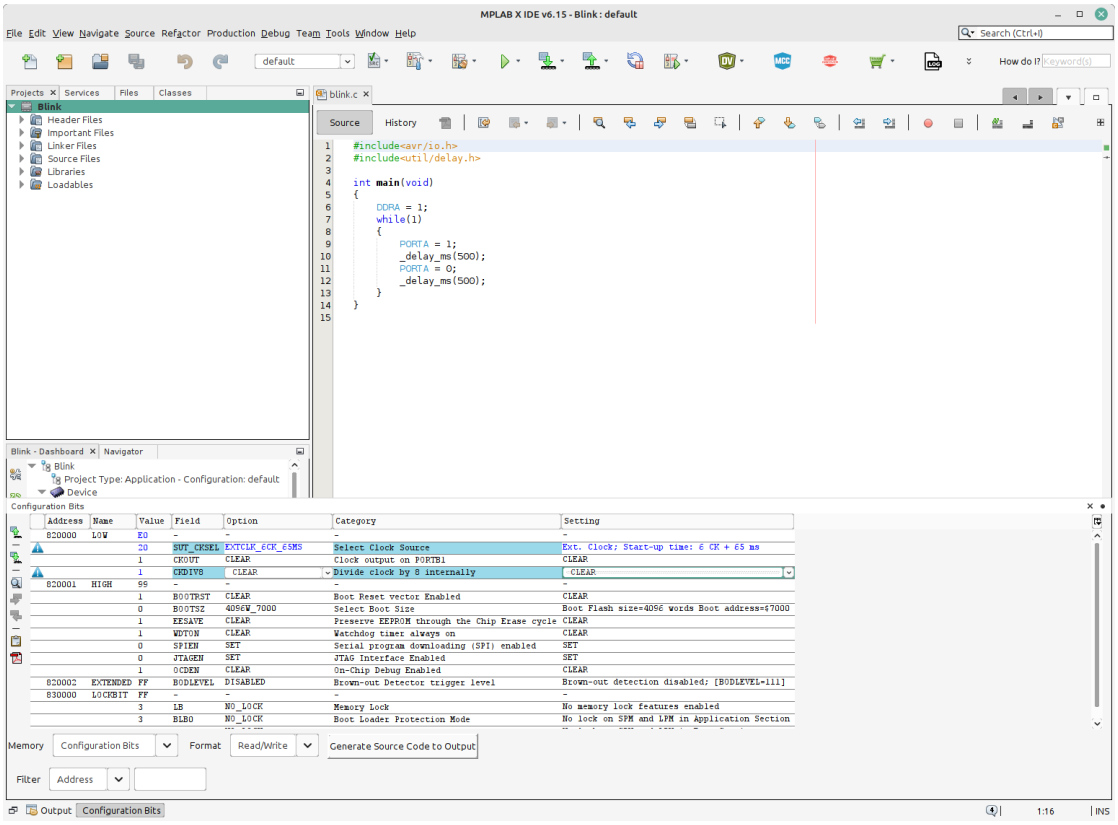


Figure 51: Settings Fuses in MPLAB X

3.4 Testing (Part 2)

Now that you are able to program the microcontroller, you can perform a full check of the hardware.

The test programs can be found in the `Tests/` subdirectory of this repository. They all come with makefiles for AVR-GCC, AVRDUDE, and USBasp. Modify the makefiles if you're using a different programmer. Alternatively, you can create a project in your IDE of choice and import the source files there. Be sure to define the `F_CPU` constant if your IDE doesn't do that automatically.

Caution

Be careful when switching from one project to the next. Changing the wiring of the board while there is still some old firmware running on the controller (or vice versa) might cause a short that can destroy the controller or other components. For example, a pin might previously have been connected to an LED and was thus configured as an output, but now gets connected to a button.
The safest way to start a new project is to disconnect all wires and jumpers from the board, then erase the controller, then re-wire and flash new firmware. In AVRDUDE, erasing can be done like this:

```
avrdude -c usbasp -p m644 -e
```

In the erased state, all pins are by default configured as inputs without pull-up which is usually safe.

3.4.1 Testing the LCD

The code for this test can be found in `Tests/LCD/`.

Preparation Connect the LCD (J15) to Port A (J11). Table 1 lists the individual connection that need to be made. Figure 52 shows the setup.

Expected Result The test program runs through several stages, testing most of the LCD driver's functions. When it is done, it shows the message "~ Finished ~".

LCD (J15)	Port A (J11)
R/W	A6
EN	A5
RS	A4
DB7	A3
DB6	A2
DB5	A1
DB4	A0

Table 1: Connecting the LCD to Port A of the ATmega

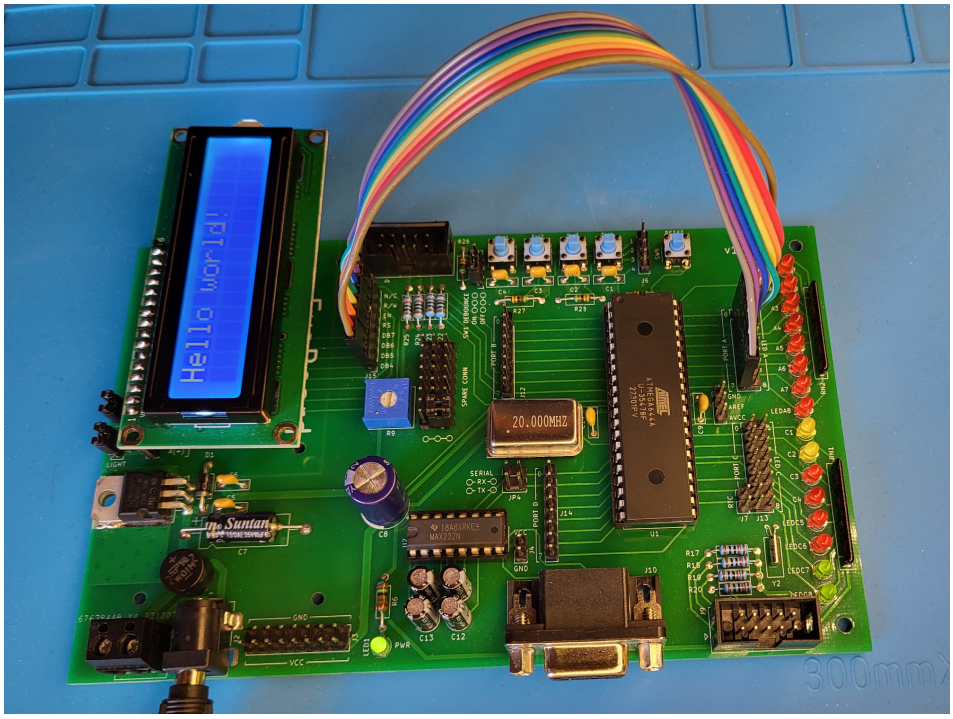


Figure 52: Wiring for the LCD example

Troubleshooting Make sure the connections are correct and the jumpers JP1 (LCD) and JP2 (LIGHT) are set. Check the contrast voltage, as described in Section 2.3. Unplug and re-plug the power after programming.

Since timings are important, the microcontroller must be running at the correct speed. See Section 3.3 for how to verify this using the `blink` example.

Many problems with the LCD stem from the fact that "HD44780-compatible" is not an actual fully specified standard. Therefore, such LCDs may exhibit subtle differences in behaviour. If nothing else works, try a different model. However, there is something else you can try first:

The LCD controller requires very specific timings: After each command from the microcontroller, it needs a pause to execute that command before it can receive the next one. The length of the pause depends on the kind of command, see Table 6 in the datasheet. The LCD driver uses sufficiently long delays for LCDs that conform to the specification, but some clones might need longer pauses.

There is an alternative to delaying: The data pins DB4..DB7 are actually bidirectional. When R/W is pulled high, the LCD will report its current status, including the "busy flag" which tells whether or not the execution of the last command has finished. Instead of waiting for a fixed time, the microcontroller can poll this flag. Uncomment `LCD_BUSY_TIMEOUT` in `lcd.h` to enable this functionality. If, on the other hand, you're not using busy flag polling, you can hard-wire R/W to GND, thus saving one microcontroller pin.

Note

Some LCDs freeze when the microcontroller is being programmed and can only be reset by disconnecting the power. This is because during programming, the microcontroller is held in reset and all its pins are floating (they act as inputs with no pull-up). This might cause problems when it comes to the EN and R/W pins.
You can prevent this by adding two 100kΩ pull-down resistors from R/W and EN to GND. Don't use stronger pull-downs (i.e. smaller resistance values), otherwise the pull-downs start fighting with the 10kΩ external pull-ups that are connected to some of the ATmega's pins. Version 1.1 of the PCB already comes with spaces for such resistors (R30 and R31). In Version 1.0 you can solder them to the back side, see Figure 53.

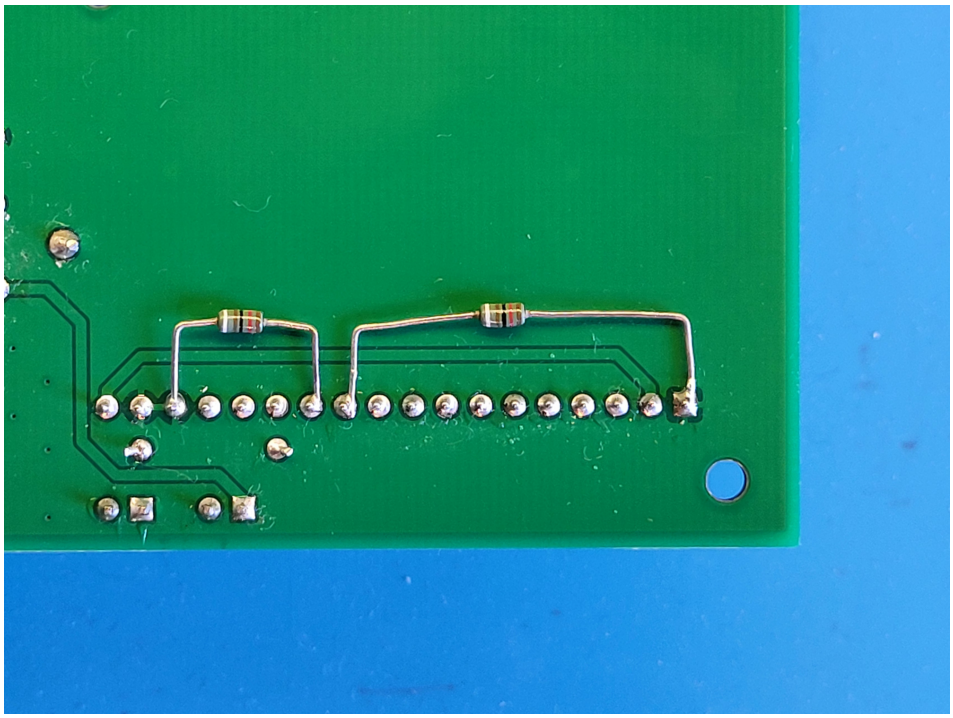


Figure 53: Adding external pull-down resistors to the LCD

3.4.2 Testing the Serial Port

The code for this test can be found in `Tests/Serial/`.

Preparation Mount both jumpers on JP4 to connect the ATmega's RX and TX pins to the MAX232, see Figure 54. Use a serial cable or a USB-to-serial connector, to connect the board's serial port (J10) to your computer.

Start a serial terminal program (for example CuteCom on Linux or PuTTY on Windows) and open a connection on the serial port using the following settings:

- Baud rate: 250000 (250kBaud)

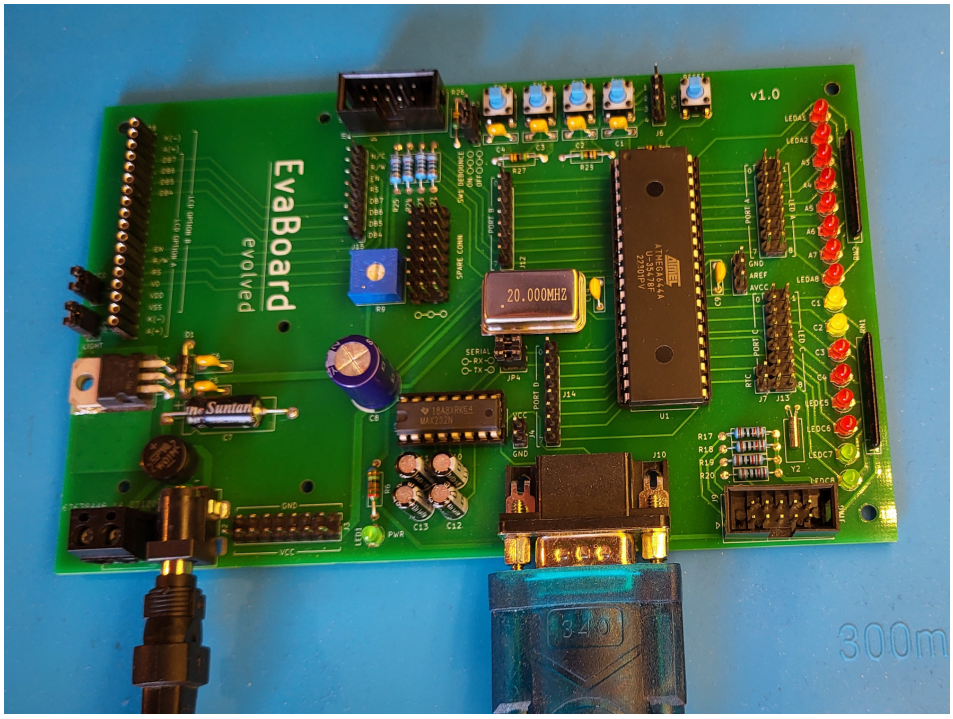


Figure 54: Wiring for the Serial Example

- 8 data bits, no parity, 1 stop bit (“8N1”)
- No flow control

Expected Result After programming the controller, it should print a message to the serial terminal. After that, it will echo any text you send, with the vowels being cyclically shifted by one.

Troubleshooting If it doesn’t work, test the hardware starting from the computer’s end. First, pull the serial cable from the board and instead connect Pins 2 and 3 with a jumper cable as shown in Figure 55. Now, the terminal should echo ever letter verbatim. If it doesn’t, check whether you have opened the right serial port. On Windows, they are called `COM?` and on Linux either `/dev/ttyS?` or `/dev/ttyUSB?` with the ‘?’ being replaced by a number. Try going through the numbers until you find the right one.

Note that USB-to-serial adapters will usually need a driver on Windows. Linux should recognize them out of the box. With some adapters there is an issue with Linux mistaking them for a braille reader. In this case, uninstall the `brltty` package. In order to access serial ports without root priviledges, the user must be added to the `dialout` group.

Once the computer’s serial port is working reliably, reconnect the cable to the board. Instead of the two horizontal jumpers on JP4, place just one vertically on the left two pins (see Figure 56). Again, this creates a loopback, this time involving the MAX232. If it doesn’t work, it is an

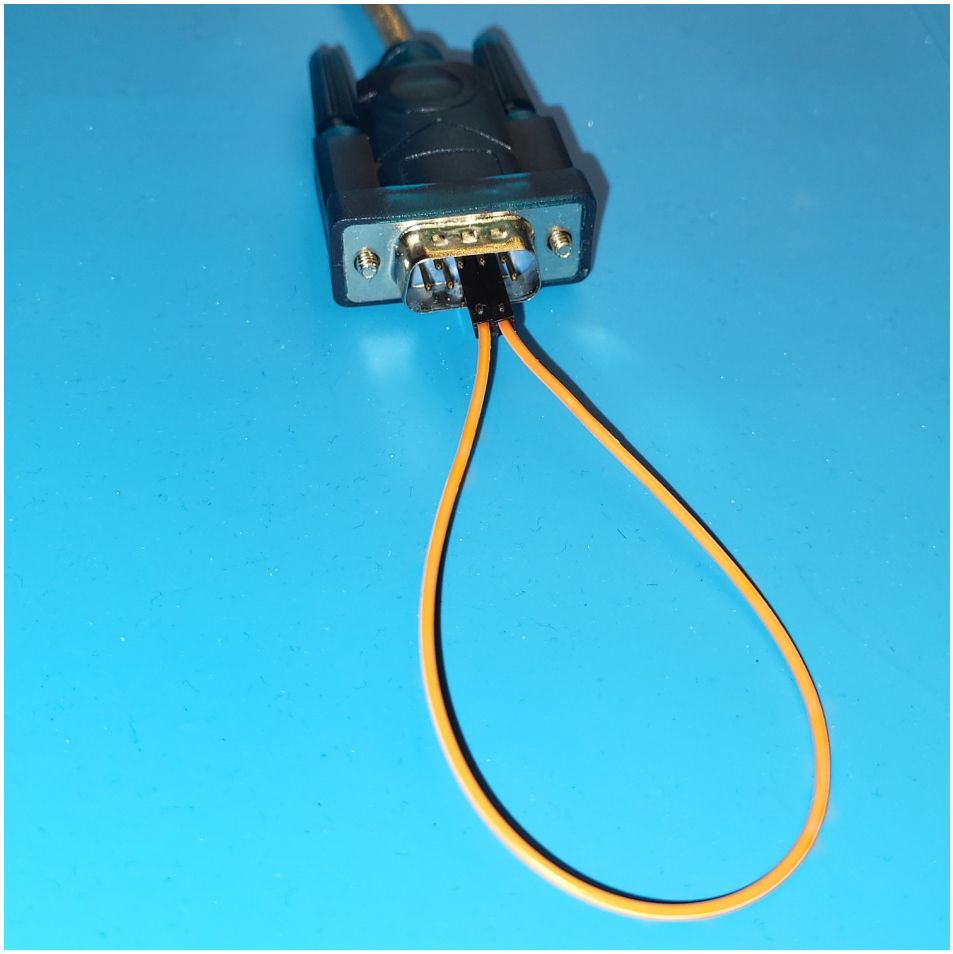


Figure 55: Loopback test with a USB-to-serial adapter

indication that something is wrong with the MAX232. Make sure the capacitors C10..13 have the correct polarity. Check the V+ and V- voltages as described in Section 2.3.

Serial debugging Once the serial port is working, it can be a great help when debugging firmware. Just copy `serial.c` and `serial.h` from the `Drivers/Serial/` subdirectory into your project, #include the latter, and call `serialInit()` at the beginning of `main()`. After that you can use `printf()` to print variables, registers, and other useful information.

3.4.3 Testing the Watch Crystal

The code for this test can be found in `Tests/RTC/`.

Preparation Connect the pins of J7 (RTC) to Port C6 and C7 (the lower left two pins of J13) using two jumpers. With another jumper, connect Port A0 to LEDA1 (the two top pins of J11). Finally, attach the LCD to J16 and connect J15 to J12 like in Table 2. See Figure 57.

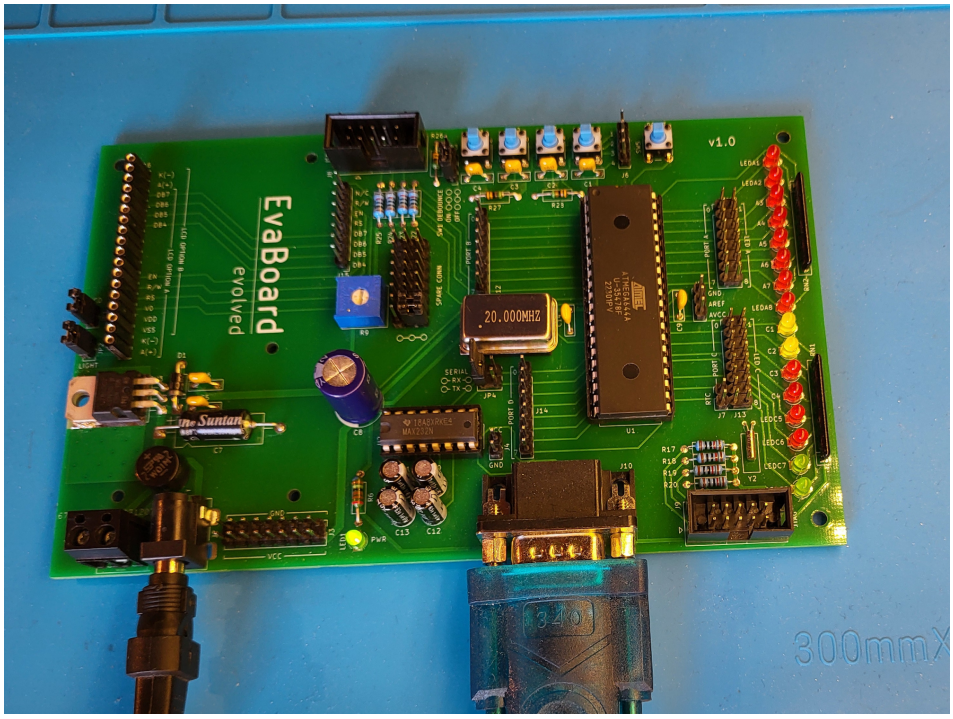


Figure 56: Loopback test involving the MAX232

Expected Result LEDA1 should blink in 1s intervals. After a short initial delay, the LCD should start displaying the current CPU frequency.

Troubleshooting Apart from misplacing wires, not much can go wrong here. If it doesn't work, you have probably fried Y2 while soldering.

Timers on the ATmega Besides testing the watch crystal Y2, this example illustrates working with timers. The watch crystal is used as an input for Timer 2. This is called "asynchronous" mode (as opposed to synchronous, where the a timer gets clocked by the CPU frequency). A 1:128 prescaler brings the frequency down from 32.768kHz to 256Hz. That means Timer2's counter TCNT2 increments 256 times per second. Since the counter uses 8 bits, it overflows once

LCD (J15)	Port B (J12)
R/W	B6
EN	B5
RS	B4
DB7	B3
DB6	B2
DB5	B1
DB4	B0

Table 2: Connecting the LCD to Port B of the ATmega

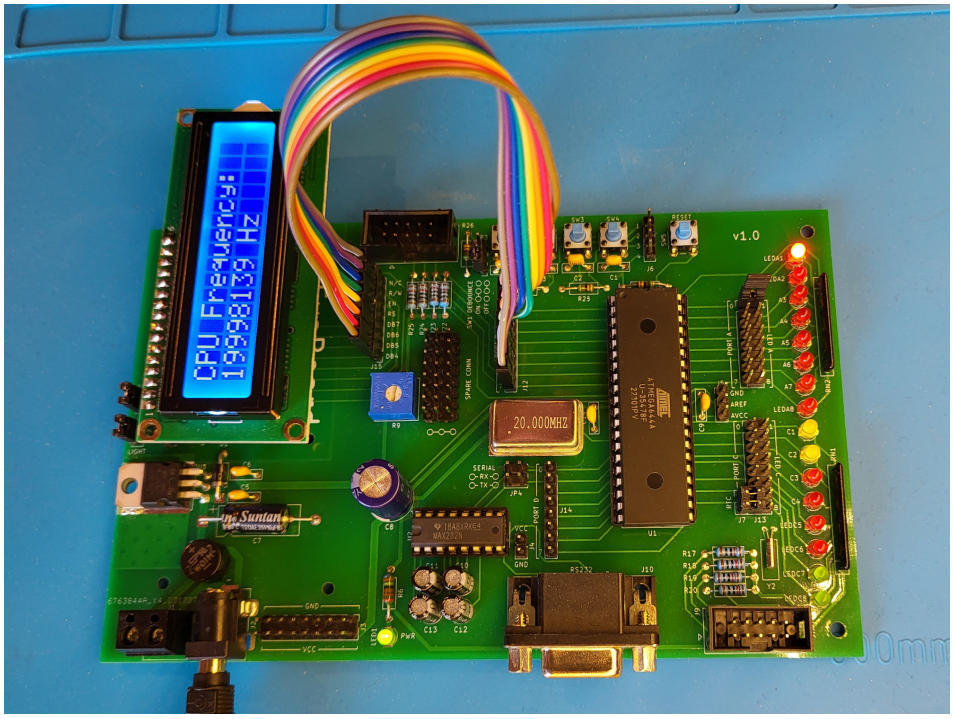


Figure 57: Wiring for the RTC example

a second, producing an interrupt which switches LEDA1 on/off.

We use another feature of Timer 2, the PWM generator. Typically, PWM is used for digital-to-analogue conversion. In this case, we set the threshold (register OCR2B) to 128. That way, every time the 8-bit counter reaches 128, the OC2B pin (Port D6) is set to zero, whereas each time the counter overflows it is set to 1. In other words, we will see a falling edge on that pin once every second, right in the middle between two overflow interrupts. We could have used this pin to drive the LED, but we will do something more interesting instead.

In parallel to all this, Timer 1 runs at CPU clock speed. Even though Timer 1 is a 16-bit timer, it will overflow much more often, since its clock is so much faster (20MHz). Counting the number of Timer 1 overflows between two Timer 2 overflows would give us a rough estimate of the CPU frequency. For a more precise measurement, we also need to take note of the value of Timer 1's counter TCNT1 at the exact moment when Timer 2 overflows. We could do that in Timer 2's overflow interrupt service routine, but there is a more elegant way to do it in hardware: Timer 1 has a feature called "input capture" which stores TCNT1's value in a separate register ICR1 whenever there is a falling edge (or a rising one, this can be configured) on the ICP pin. As luck would have it, the ICP pin is Port D6, i.e. the exact same pin where Timer 2 produces a falling edge once every second. All that is left to do, is read ICR1 in the ICP interrupt service routine. The main program then calculates the CPU frequency from two consecutive ICR1 values and the number of Timer 1 overflows in between and shows it on the LCD.

Note that this method relies on the accuracy of the watch crystal. The latter is not perfect –

its deviation is typically on the order of ±20ppm (parts per million)[4]. This is still better than the crystal oscillator Y1 (±100ppm), let alone the ATmega's internal oscillator (±10%). In fact, an external watch crystal like Y2 could be used to calibrate the internal oscillator (see Section 7.12.1 of the datasheet).

4 Advanced Topics

4.1 Use of the Programming Pins in Applications

Some of the 32 GPIO pins of the ATmega are used for programming and debugging. We have to be careful when connecting these to other hardware.

Note

In all the examples in this document, this has been taken into consideration. But other applications may not do so – pay attention, especially when they use a different type of programmer than you do.

This section only provides the very basics of sharing programming lines. For more detailed information, see Section 4 of the AVR042 Application Note.

ISP ISP programming happens on Pins B5..B7 (and the reset pin). This does not mean, however, that you cannot use them for other things. But you need to make sure that whatever device is connected to the pins satifies the following conditions:

1. During programming, the device must be able to cope with arbitrary signals on these pins.

2. During programming, the device must not interfere. First and foremost, it must not attempt to output anything. Second, it should not put a substantial load on these lines (like a large capacitance or other things that draw large currents or prevent fast switching), although this can sometimes be mitigated by choosing a slower programming speed.

LEDs are usually fine – of course they flash during programming. Buttons (not pressed!) are also ok, even with weak (i.e. high Ω value) external pull-up or pull-down resistors.

One thing you have to be very careful about is the LCD. This is because it outputs data on DB4..DB7 when R/W is high. The best solution is to connect the LCD to some port other than B when programming over ISP. If you insist on using Port B, make sure that either R/W is kept low (and you cannot do this via the microcontroller because it is in reset during programming!) or that DB4..DB7 are connected to pins other than B5..B7. This is one of the reasons why in most examples the LCD is connected "twisted", i.e. DB4..DB7 go to Pins B0..B3.

After programming, when the ATmega is running, any well-behaved ISP programmer stops outputting on the ISP lines. Thus, you can leave it connected.

[4]The error is mostly temperature-induced, as you can observe when carefully placing a finger on Y2. Higher quality RTC circuits auto-calibrate using temperature sensors. This brings the error down from ±20ppm (10min/year) to less than ±1ppm (30s/year)

Note

If you know about SPI (perhaps from using the SRAM add-on board), you can actually understand ISP in a bit more detail. It is no coincidence that "ISP" is an anagram of "SPI". In fact, ISP uses the SPI protocol: The programmer acts as the SPI master and the ATmega as the SPI device. This explains why ISP uses the same MOSI, MISO, and SCK pins as SPI. The reset pin of the ATmega acts as CS.
Remember that SPI devices leave the SPI lines alone when not selected. This means in particular, that any other SPI devices can stay connected during ISP programming, as long as their CS lines have pull-ups that keep them de-selected. The 23LC1024 on the SRAM add-on board is an example of this.

JTAG JTAG uses Pins C2..C5 (and the reset pin). If you are programming and debugging via JTAG, these pins are in use the whole time. Do not connect anything else to them.

If you're only programming via JTAG but not debugging, the same conditions as with ISP apply. However...why would you be doing this? If you're not debugging anyway, just use ISP and save some pins.

Other pins While the ATmega is being programmed, it is held in reset. This means that all other pins are "high impedance" or "floating" (which essentially means they are configured as inputs without pull-up). In most cases, this is perfectly safe, regardless of what other devices are connected to them. Only some LCDs have an issue with floating pins, as we mentioned in Section 3.4.1.

4.2 Debugging

On the surface, debugging a microcontroller program looks similar to "normal" debugging. You can set breakpoints, step through the code, and examine the contents of the memory.

Internally though, hardware debugging is a bit more complicated and limited. Debugging can only work if the microcontroller comes with debugging support built into the hardware. The ATmega644 has an on-chip debugging system, see Section 22 of the datasheet.

There are two kinds of breakpoints: hardware breakpoints and software breakpoints. A hardware breakpoint consists of a register that is constantly compared to the program counter (PC). When a match occurs, an interrupt is generated which causes the microcontroller to halt. There are only a limited number of such registers (3 for the ATmega644) which limits the number of hardware breakpoints.

A software breakpoint is created by replacing an instruction with another one that generates this kind of interrupt. When resuming from a software breakpoint, the original instruction must be restored. This means that each software breakpoint requires two write operations to the program memory which take time and wear down the flash memory over time[5]. The upside is that there is no limit to the number of software breakpoints.

[5]This is not as dramatic as it sounds. The ATmega flash is specified to last at least 100.000 write cycles. In practice, that number is often surpassed by an order of magnitude. It would probably take years of intense programming and debugging before the flash wears out.

4.2.1 Example: Debugging with MPLAB SNAP and MPLAB X

Before you start, make sure MPLAB X and SNAP work for JTAG programming as described in Section 3.2.6.

The ATmega needs to be configured for debugging by setting the OCDEN (On-Chip Debugging ENable) bit in the high fuse byte. Refer to Section 3.3 and Figure 58 for how to do that.

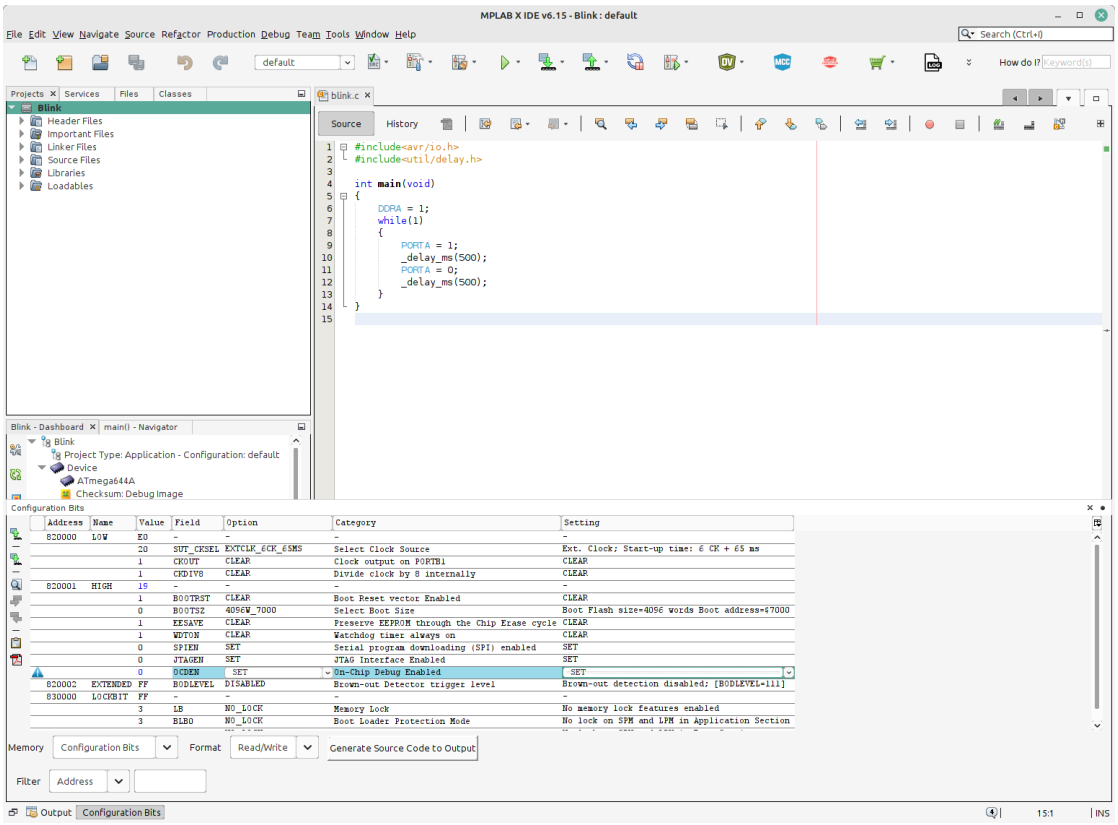


Figure 58: Preparing the Microcontroller for Debugging

Now we can start debugging: Place a breakpoint somewhere in your program by clicking on the line number. The IDE will prefer hardware breakpoints over software breakpoints. The latter are only used once you run out of hardware breakpoints. You can also disable software breakpoints completely with the red square button in the navigator. The navigator also shows the number of used and free breakpoints.

Hit the “Debug” icon. The program stops at the breakpoint and you can examine the state of variables and registers using the different tools from the “Window”, “Debugging” menu and the “Window”, “Target Memory Views” menu. Toolbar icons allow you to pause/unpause and step through the program.

Note

The `_delay_ms()` function is not a real function and thus the debugger has a hard time stepping over it. Instead, set a breakpoint behind it and press "Continue" or use "Run to Cursor".

You can find a lot more information on debugging in the MPLAB X User's Guide.

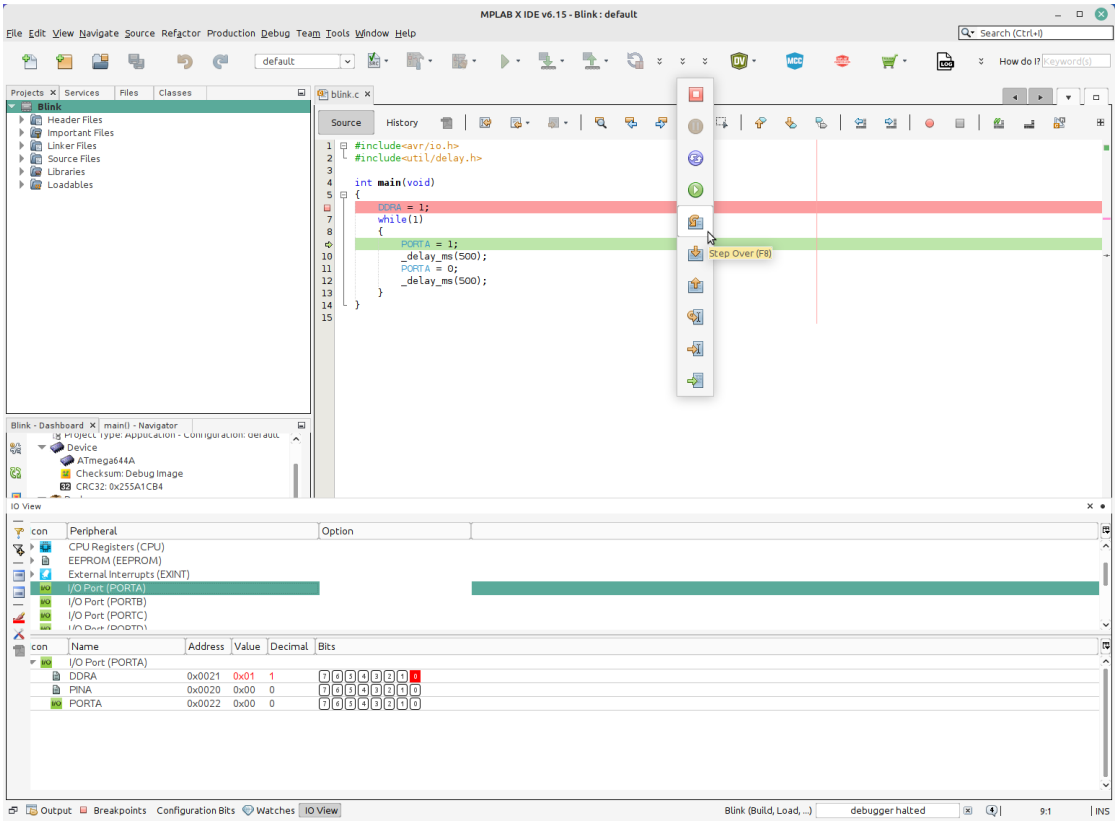


Figure 59: Debugging with MPLAB X

5 Known Issues

Pull resistors on JTAG pins While pull-ups are fine (and recommended) for TMS, TDI, and TDO, for TCK it should be a pull-down. However, since this doesn't seem to cause problems and in the interest of not interfering too much with the pins in GPIO mode, this is probably a WONTFIX.

Unnecessary pull-up resistors on MOSI, MISO, and SCK pins Frankly, we have no idea why these exist. Neither ISP nor SPI require them.

6 Version History

v1.0 Initial release

v1.1 Added pull-down resistors for the R/W and EN pins of the LCD (see note in Section 3.4.1).