An Introduction to Building Electronics Projects with Arduino

Reto Trappitsch

Fall semester 2021

Contents

Preface

The notes are structured into 6 chapters. The first chapter mainly describes basics that you likely already know from your introduction to Physics classes. Subsequently, we will discuss one chapter per workshop session. The notes are prepared as we go, and you can always find the latest version, but also solutions to the examples in the form of code examples, on GitHub. If you find typos, errors, or other issues please let me know. The most recent copy of the LaTeX files and figures can also be found on GitHub.

The lecture notes contain clickable links in dark blue. Furthermore, boxes throughout the text discuss are used for the following contents:

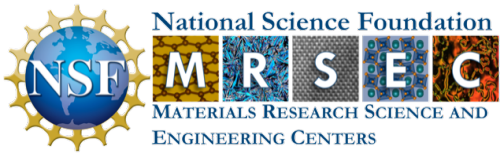
Background information on topics that do not necessarily fit into the text but are important to keep in mind will be given in a box like this.

Think about it more! These boxes will challenge you to think a problem through for yourself and go into more detail.

Exercise 0 Exercises are given in these boxes. Flex your coding muscles and practize what you've learned.

Question 0 Questions will be given in these boxes. They should solidify your background knowledge.

This workshop was first designed for interested students at the Materials Research Science and Engineering Center (MRSEC) at Brandeis University in fall 2021. Support is provided by the Brandeis NSF MRSEC, DMR-2011486.



Acronyms

AC alternating current

ADC analog-to-digital converter

AWG american wire gauge

BOM bill of materials

CAD computer-aided design

CSV comma separated values

DAC digital-to-analog converter

DC direct current

I²C inter-integrated circuits

I/O input / output

IDE integrated development environment

IoT internet of things

LED light-emitting diode

MOSFET metal-oxide-semiconductor field-effect transistor

MRSEC Materials Research Science and Engineering Center

PCB printed circuit board

PID proportional–integral–derivative

PWM pulse width modulation

SQUID Simplifying Quantitive Imaging Development and Deployment

TEC thermoelectric cooler

0. Introduction

Scientific research that focuses on experiments and measurements has rapidly grown in the recent past, mainly thanks to significant improvements in engineering, instrument availability, and computing power. While many companies provide state-of-the-art research instrumentation and setups, cutting-edge scientific discovery often still thrives from home-built setups.

In addition to scientific instrument development and availability, the consumer / hobby marked has seen a huge increase in home-made electronics.[1] This development has especially been facilitated by products such as Arduinos and Raspberry Pis, as well as the huge maker community. Automation of research experiments can often benefit from such existing, low-cost products in order to significantly enhance an experiment or measurement. Furthermore, complete low-cost instruments enabling frugal research have also been developed based on such platforms, see, e.g., the Simplifying Quantitive Imaging Development and Deployment (SQUID) project.

In this workshop, we will develop a temperature regulation system in order to cool a sample. We will read the temperature using a thermistor, use buttons to control the set point, use a display to show the displayed and set temperature, and finally drive a thermoelectric cooler in order to achieve sample cooling.

0.1. Basic Physics to Remember

Building electronics is not just fun because you can hold your final product in your hand and play with it, but also since it is a direct application of basic physics. Remember your introductory classes!

Ohm's law Throughout this workshop, you will encounter Ohm's law very frequently. This law states the current I through a conductor with resistivity R is directly proportional to the voltage U across the conductor. We can write this as:

$$I = \frac{U}{R} \tag{0.1}$$

$$U = RI \tag{0.2}$$

[1] For example, have a look at this article in the New York Times. Looking at the images clearly shows a 3D printed case as well as a standard Arduino cloud interface.

Remember this relationship for when you design your circuits.

Maximum current Arduino pins, as we will discover later, can supply 5 V to, e.g., an light-emitting diode (LED). Since an LED is a diode, it's resistance (if connected properly) is close to zero (see also Wikipedia). Therefore, applying a 5 V voltage would result in an infinite current across this component. How would you add a resistor to limit the potential current to a maximum of 10 mA?

Electric power If a current flows through a resistor, electric energy is transferred. The energy per time that is used in this resistor is the electric power P, which can be calculated as

$$P = UI. \tag{0.3}$$

Often, electric power is dissipated as heat. For example, an incandescent light bulb creates light (and heat!) by applying a voltage to a filament that is generally made of tungsten. The filament heats up and emits light. All electronic components have a maximum power rating, also often expressed as a maximum current rating.

Maximum power versus maximum current Assume you have a component that that consumes at most 5 W of power at a current of 1 A. What is the maximum voltage that you can apply? What is the resistance of this element at the maximum voltage?

Electronic components and symbols You are probably already familiar with basic electronic components such as resistors, capacitors, etc., and their symbols. We will discuss various components during this workshop. A good overview of components to refresh your memories can be found here on Wikipedia. Standard electronic symbols, which are really useful for drawing circuit diagrams, can be found here.

0.2. Analog and Digital

If you turn on a radio, and it is too loud, you can use the volume knob, which is nothing else than an adjustable potentiometer, in order to regulate the volume of the sound. This volume can be adjusted over a whole range of settings. The potentiometer adopts linearly depending on its position, giving you an analog control over the volume. Mapping the volume from 0 (quiet) to 1 (loud), you can reach any value in between. Digital signals on the other hand are either on or off. An Arduino generally has many digital input / output (I/O) pins which can be either high (5 V) or low (grounded). If

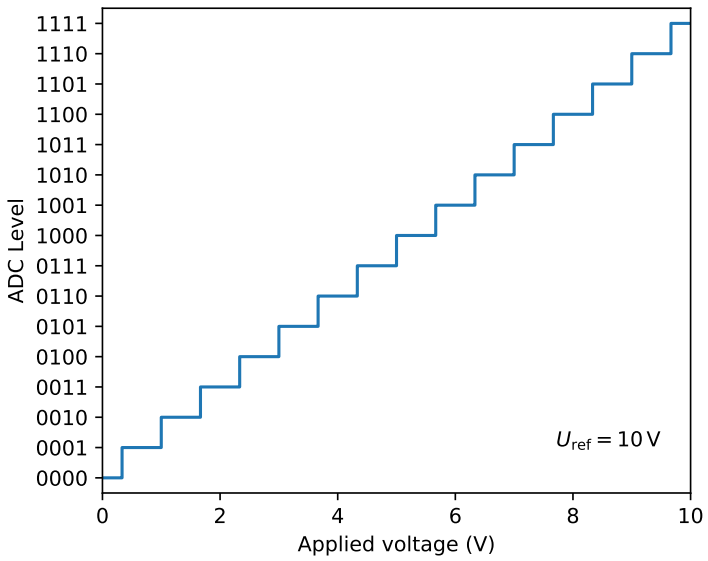


Figure 1.: *A 4 bit ADC with given levels. The reference voltage is 10 V.*

you connect a 3.3 V battery to an input pin, of course via a resistor in order to not exceed the current maximum, the switch would either tell you that it is high or low, depending on the threshold that are actually set in order to determine this. Any kind of microprocessor *only* understands digital signals.

0.2.1. Analog-to-Digital Conversion

In order to measure as signal from a sensor, e.g., a photodetector or a temperature sensor as we will use later, a device called an analog-to-digital converter (ADC) can be used. Above we mentioned that any kind of microprocessor only understands digital signals. The same is also true for an ADC. While a digital I/O pin has two levels (high / on or low / off), an ADC generally has many more levels in between. The resolution of an ADC is generally expressed in bits.

Bit For any microprocessor, the two possible states (high and low) can be expressed as 1 and 0. Binary numbers (base 2) are therefore the ideal representation to express different states. A digital I/O pin has 1 bit resolution, which means it can either be 1 or 0. Higher resolution means that more bits are available to set states. For two bits, i.e., a binary number with with two digits, the possible states are 00, 01, 10, 11. This means that 2 bit resolution has a total of four steps. For n bits, the number of available steps are 2^n.

Figure 1 shows the levels of a 4 bit ADC in binary as a function of the voltage that would be measured. The reference voltage here is $U_{\text{ref}} = 10\,\text{V}$. This reference voltage is the voltage that the ADC can measure at most, i.e., the voltage that it will return when the ADC level is 1111.

Knowing the resolution n of an ADC, we can easily calculate the minimum voltage difference that can be determined as

$$\Delta U = \frac{U_{\text{ref}}}{n}. \tag{0.4}$$

For the given example above in Figure 1, the minimum voltage would thus be $\Delta U = 0.625\,\text{V}$. Anything smaller voltage difference requires a higher resolution ADC.

0.2.2. Digital-to-Analog Conversion

Of course, we sometimes require the opposite of an ADC and need to convert digital signal into the analog world. The device that allows for this transformation is a digital-to-analog converter (DAC). A true DAC takes a digital signal and returns an analog voltage by dividing a reference voltage as many times as necessary. The same resolution limitations as for an ADC also apply to a DAC. For example, a 4 bit DAC with a reference voltage of 10 V can only increase the analog output in steps of 0.625 V.

Pulse width modulation (PWM) An interesting way to have a pseudo DAC is to use a process called pulse width modulation (PWM). Figure 2 shows a schematic of this process for a 5 V pin. The digital pin is rapidly turned on and off. If it is at 5 V for 50% of the time, a 50% duty cycle, the effective, smoothed-out voltage that can be seen by a "slow" component would be 2.5 V. Depending on the duty cycle, a pseudo-analog output can therefore be created. A great example to use PWM is to have a dimmable LED. LEDs generally have only two states, on and off, i.e., is the voltage is high enough to light them, they are bright and otherwise dark. Using PWM however, we can turn an LED on and off in rapid succession such that it looks to the human eye as if the light source itself was dimmed.

Analog output with PWM Can you come up with a way to create a smooth analog output from a PWM pin? Think about how your cellphone charger turns alternating current (AC) into direct current (DC).

0.3. Arduino

For this class, we will be using an Arduino Micro. You can find more information and various alternative Arduino boards for all kinds of projects on the Arduino website.

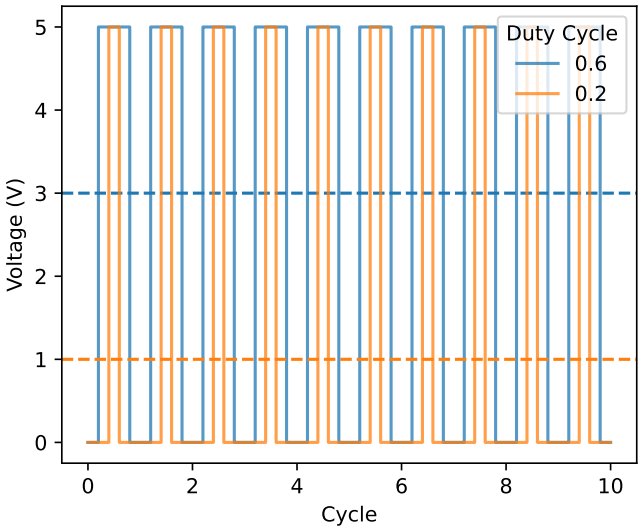


Figure 2.: *PWM cycles and average voltage for two different duty cycles.*

In Appendix A, a schematic of the Arduino Micro board is given. This pinout diagram specifies what all the various pins on the board mean and what they are used for. It is a very handy reference for when you develop your project.

0.3.1. Programming an Arduino

The easiest way to program an Arduino is by using the integrated development environment (IDE) for Arduino that can be found here. The website also has installation guides on how to install the IDE and Arduino driver on your computer, depending on your operating system. Please see this documentation or look at it at least briefly, since it will help you with troubleshooting in case your computer does not find the Arduino board.

The Arduino IDE is very useful when you are starting to learn how to program your Arduino. Under "File", "Examples" you can find eleven categories that give you many well-explained example snippets of code for various applications. We will make a lot of use of these examples, especially during the first few chapters of this workshop. The IDE also allows you to verify your code, i.e., check if it contains any errors (check button in the toolbar) and to upload your code to the Arduino itself (right arrow button in the toolbar). In order to upload your code to the Arduino board, make sure that you have the correct board selected. Go to "Tools", "Board" to select "Arduino Micro". Furthermore, you need to select the port on which your board is connected to the computer. To do

so, go to "Tools", "Port" to select the correct one. Now you are ready to begin uploading example code or your own code.

Programming language overview To program your Arduino, the code must be written in C++. There are many great introductions online that can help you to get started, see also the background information box below. Therefore, we will only discuss very briefly the most important rules here. These will help you to avoid the most common mistakes.

- Commands and instructions are case sensitive
- Variables must be declared. If you need an integer for example and assign it the value three, you can do this by declaring the variable as `int myVar = 3;`
- All command lines must be terminated by a semi colon `;`
- Functions, loops, etc. get surrounded by curly brackets
- It is up to the user to make the code look readable. If you want, you can write everything into one line since line endings and function endings are defined by the above stated rules.
- Line comments are preceeded with `//` while block comments use the following structure:

```
/*
    My comments in a block...
*/
```

In general, the minimum file structure for your Arduino code should look similar to the following.

```
// variable declarations, load libraries

void setup() {
  // setup code
}

void loop() {
  // main code that repeats
}
```

On the very top of your code, put variable declaration and initialization if required and load the necessary libraries. The `setup` function is the part that runs once when you boot up your Arduino. The `loop` function will then run repeatedly and, ideally, until you unplug or reset the Arduino. We will see later how to fill these standard functions. In addition, you can of course write your own functions with any names of your choosing, just make sure they do not collide in naming with these default functions.

Numbers To understand variable declarations in C++ better, we need to remind ourselves how numbers are stored in a computer's / microprocessor's memory. One of the most important types of numbers are integers, declared in C++ as `int`. In the Arduino Micro, an integers is 16 bit in size. This means that 16 positions are available in binary. Integers with a sign can therefore go from -2^{15} to $2^{15}-1$; the first bit is reserved for the sign. An `unsigned int` on the other hand goes from 0 to $2^{16}-1$. Integers of type `long` have 32 bit available.
Decimal point numbers are represented by `float`s, which can take values from $-3.4028235 \times 10^{38}$ to 3.4028235×10^{38}. They require 32 bits of memory to be stored.
You can find more information on different variable types in the respective Arduino reference section.

Debugging When code does not do the thing we expect it to, it is often difficult to exactly see why it does not work. While the Arduino IDE catches many bugs, it cannot catch logic errors. If you simply code a script on your computer, you might debug it by printing out values to the screen. The same can be done with the Arduino. Unless you have a screen connected to the Arduino, you have to send the values to print via the serial connection to your computer for displaying. The statement `Serial.begin(9600);` should go into your setup routine. This begins the serial communication to your computer with a baud rate of 9600. Inside your main loop, you can then use the following print statements to write to your computer:

- `Serial.print("Hello␣");` prints "Hello " without a line break.
- `Serial.println("World!");` prints "World!" on the same line as the print before and then starts a new line.

To see these printed values, click in the Arduino IDE on "Tools" and then select "Serial Monitor". Note that you can also send commands from the computer to the Arduino via Serial. More information can be found here.

Help with programming your Arduino To find further information on how to program for Arduino and get yourself started with C++, see the following links:

- Starters guide for programming for Arduino
- Programming reference specifically for Arduino
- Libraries in Arduino
- Glossary of commonly used terms
- User forum

Note that many components that we use come with detailed instructions and guides. For example, if you buy components from Adafruit, these parts generally come with a guide for Arduino, etc.

More Help As with so many things in life these days, more help is generally just one search on the internet away. There are many forums, articles, etc., on the web that discuss building electronics with Arduino. The hope is that this workshop helps you to discover the vast possibilities and gives you the right keywords to search your way through.

0.3.2. TinkerCAD

If you want to play with a virtual Arduino, give TinkerCAD a try. On its website you can simulate an Arduino, add components, write code, and then run the setup like in real life. It is a great tool to plan a project and start experimenting with setups, e.g., while you are waiting for components to ship.

1. Blink

Like programming tutorials start with a "Hello World!" program, electronics tutorials generally start with a blink example. In this chapter you will learn how to write to and read from digital I/O pins.

1.1. LEDs

1.1.1. Internal LED

Arduinos with a built-in LED allow the user to program and use this LED. Therefore, all the hardware you need is the Arduino and a USB cable in order to connect it to your computer. Feel free to plug the Arduino into a breadboard as shown in Figure 1.1. The internal LED in Figure 1.1 can be seen just on the right side of the label that says

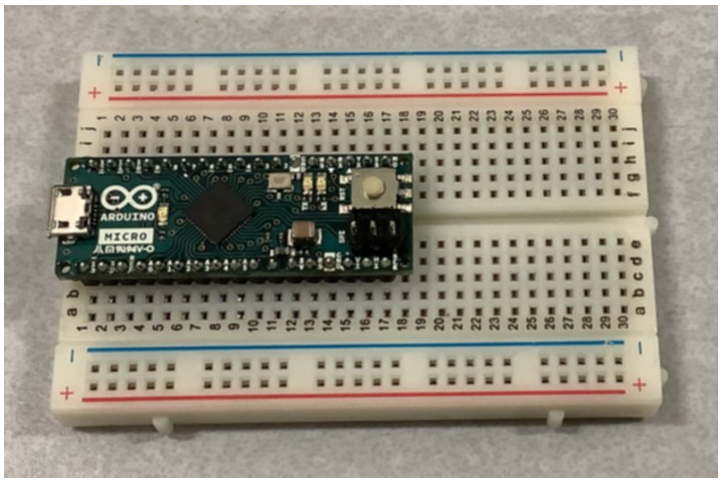


Figure 1.1.: *Arduino micro plugged into a breadboard. On the left the USB connection is visible. On the right of the label where it says "Arduino", the internal LED can be seen.*

"Arduino". In addition, you can see a button on the right-hand side of the board (the reset button) as well as two more LEDs on the left side of this button. These additional LEDs cannot be accessed by the user and are reserved for the system.

If you start the IDE and load the basic example "Blink", you will get some code that will control the LED. The following exercises will use this simple example and slowly extend it.

Exercise 0 Open the example blink file and read the comments. What is done in the setup? What does the variable `LED_BUILTIN` stand for? Study the loop: What will happen when you upload the code to your Arduino? Do so and see if your assumptions were correct. Modify the timings of the program such that the LED blinks at a different rate.

1.1.2. External LED

We can also connect an external LED to a digital I/O pin. To use a pin as an output pin, i.e., to set its level by software, we have to define the `pinMode` to be in `OUTPUT` mode. Furthermore, connecting an LED to a pin and simply driving it can be bad for the LED, since it has, by itself, no resistance. Looking at equation (0.2) we can see that in such a case the current should become infinity, which might destroy the I/O pin. Fortunately, Arduinos have an internal resistor that prevent this from happening. However, the LED might still get too much current, which might significantly reduce its lifetime. You can either add a resistor or use an LED with an internal one, as we are doing here (Figure 1.2).

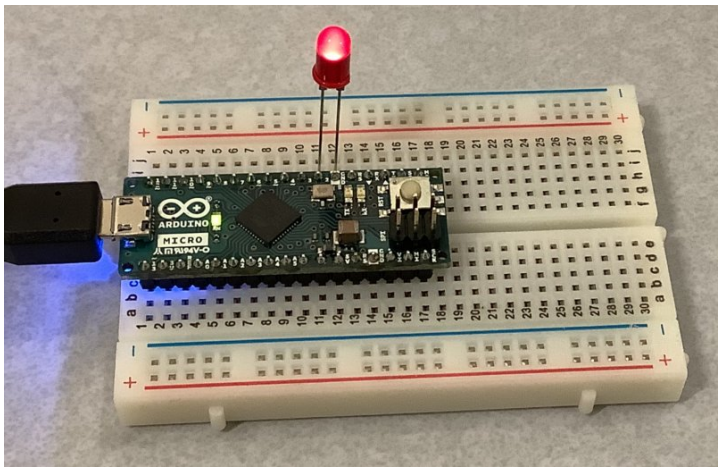


Figure 1.2.: *Arduino with one LED connected.*

Exercise 1 Draw a wiring diagram to connect your own LED to a Arduino output pin. Where do the anode and cathode of the LED connect to? Attach your LED to the arduino and modify the simple blink experiment to use your LED instead of the built-in one. If you have trouble figuring out how to connect the LED, study Figure 1.2 and remember that every electric circiut must be completed.

1.1.3. Dimming an LED

As we have discussed above, LEDs cannot be dimmed in the traditional way, i.e., by using a potentiometer to lower the voltage. However, we can use a PWM output in order to only have the LED on for a certain amount of time. This will result in our brain perceiving the LED as dimmed. We have already described the PWM outputs above in Section 0.2.2, see also Figure 2. In order to identify a PWM output, look at the pinout (Appendix A). Digital pins indicated with ∼ are the ones that can be used in this fashion, e.g., pin 3.

Exercise 2 Connect your LED to a PWM output pin. From the Arduino IDE, load the basic example named "Fade".

1. Read the setup and loop functions. How is the pin output set and what is different from how the pin was set in the previous exercise?
2. Modify the fade amount to 7 and run it again. At the brigthest point, the LED briefly blinks. Why?
3. Can you rewrite the routine in order to prevent it from blinking at the highest point? An `if` statement might be useful.

The `void loop()` function acts as an endless loop. The most commonly used looping structures are `while` and `for` loops. In fact, the loop function iself underneath is a `while(true)` loop.

Exercise 3 Inside the `void loop()` function, write a `for` loop to replace the ramp. Ensure that your loop does not blink, even if you select various fade amounts.

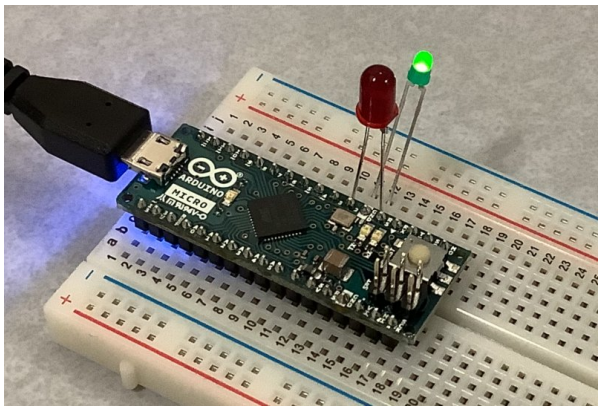


Figure 1.3.: *An Arduino with two LEDs connected.*

Question 0

- How does integer division in C++ work?
- How can you do a division in C++ and round the result to the nearest integer?
- What is a boolean variable and how do you compare two booleans, e.g., in an `if` statement?
- How would you flip the state of a boolean variable?

1.1.4. Multiple LEDs

Since you have more than one pin available, multiple LEDs can be set up. An example of a test setup is shown in Figure 1.3 This can be especially useful if you want to display the status of your setup using different colored LEDs.

Subfunctions in C++ Sometimes it is useful to put some of your code into external functions, i.e., not to write them into the main loop. For example, you can write a function as following:

```
void myFunction(bool toggle) {
    // your code here...
}
```

You can then call this function from the main loop by calling `myFunction(true);` if you want to assign the value `true` to the variable `toggle`.

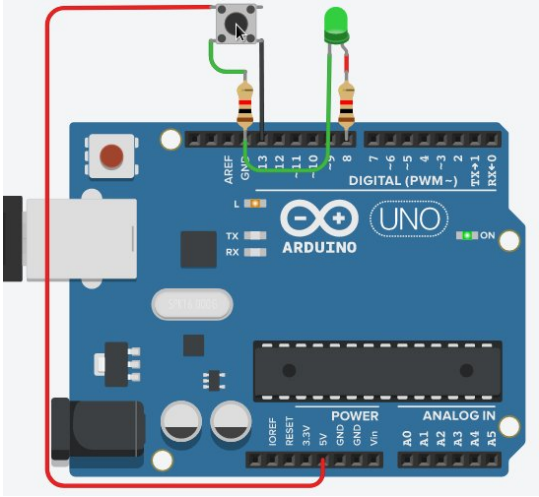


Figure 1.4.: *TinkerCAD simulation of Arduino with a button and LED connected.*

Exercise 4 Setup two LEDs, a red and a green one. Write a program that switches between blinking the red and green LEDs. Write a subroutine that takes a boolean variable as an input and, depending on the value of this input variable, turns either the red or the green LED on (and the other one off).

1.2. Buttons

Most devices that allow for user interactions contain some buttons. We can use the digital I/O pins in order to connect a button. Figure 1.4 shows a TinkerCAD simulation of an Arduino with a button and an LED connected. Here, we connect the button on one side to the 5 V output of the Arduino and the other side via a 1 kΩ resistor to ground and simultaneously to a digital I/O pin. The resistor limits the current to 5 mA, see equation (0.2). In the setup routine, we need to set the `pinMode` to `INPUT`, since we want to read the value that is connected to the pin. If the button is unpressed, the pin is on the same electrical potential as ground and therefore in a low state. If the button is pressed however, the pin is at 5 V and thus in a high state. In the loop you can, e.g., read the state of the button connected to `buttonPin` as:

```
int buttonState = digitalRead(buttonPin);
if (buttonState == HIGH) {
    // the button is pressed
} else {
    // the button is not pressed
}
```

Exercise 5 Connect a button and an LED to your Arduino and write a program that turns the LED on while the button is pressed. Then adjust your routine such that, whenever you press the button, the LED is turned on for 3 s and then turns off after the time has elapsed.

Adding time In the second part of the above exercise, you have (most likely) written your program such that if the button is pressed while the LED is on, nothing happens. The light will still turn off once the set time elapses after the initial button press. Can you come up with code that would allow you to reset the countdown when pressing the button again, i.e., add time to the timer?

2. Display

In order to display the temperature of our sample cooling setup, we will connect a display to our Arduino. Displays come in many shapes and forms, form the monitor you might be reading this on to simple LED seven-segment displays. An overview of various displays that are ready to be deployed on Arduino can, e.g., be found here. For our specific case, a simple seven-segment display will do the trick to display the temperature.

2.1. Seven-segment displays

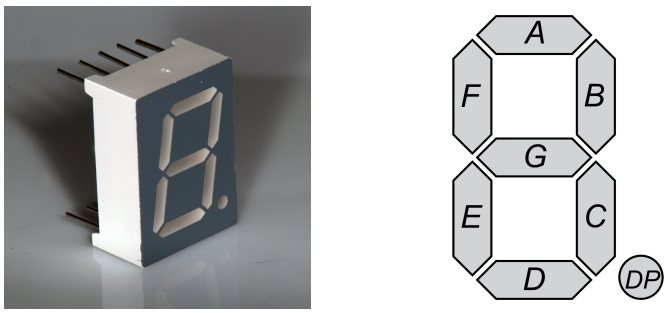


Figure 2.1.: *Seven-segment display: photo (left) and schematic (right). Credit: Peter Halasz (left) and Uln2003 (right) via Wikipedia. License: CC by SA-3.0 (left), Public domain – CC0 (right)*

As displayed in Figure 2.1, seven-segment displays contain seven individual segments that allow us to display any number and even some letters. In addition, they usually contain a decimal point. The right-hand figure shows the schematic and typical labeling of a seven-segment display. Each segment is its own LED and we could drive these LEDs by using multiple I/O pins. However, in order display multiple numbers, the number of I/O pins that we would use up would very fast become too extensive. Figure 2.1 shows that the seven-segment display has five pins on top and, not shown here, also five connectors on the bottom. For each seven-segment display you could look up the wiring diagram and notice that each segment has its own connection and that they have a total of two common grounds.

Chips such as a 74HC595 that allow us to convert serial into parallel data allow us to effectively control multiple outputs, as required for such a display, while only using

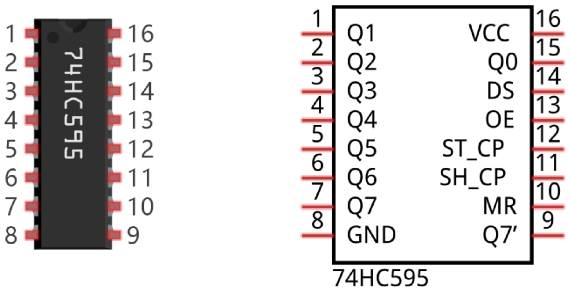


Figure 2.2.: *Schematic and pin names of a 74HC595 chip. Credit: Freenove, License: CC BY-NC-SA 3.0*

a a few pins. Figure 2.2 shows a schematic and pin names of such a chip. Note the notch on the top of the schematic that each chip contains in order to provide you with a reference on the pin assignments. You can in fact find various versions of this chip from different manufacturers, e.g., the data sheet for the Texas Instruments version can be found here. Generally, the pins have the following purposes:

Pin name	Pin number	Description
Q0 - Q7	1-7, 15	Parallel data output
GND	8	Ground
Q7'	9	Serial data output
MR	10	Remove shift register
SH_CP	11	Serial shift clock
ST_CP	12	Parallel update output
OE	13	Enable output
DS	14	Serial data input

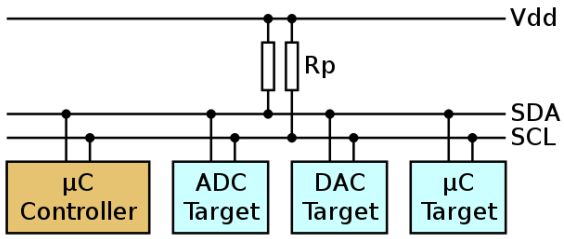
As indicated by the names, the parallel data output pins are the eight pins that can be connected to the individual LEDs of a seven-segment system. The GND pin provides the ground connection of the chip. The serial data output (Q7') can be used to connect another 74HC595 chip in series. The MR and SH_CP take care of clearing the shift register and timing when a shift happens. The ST_CT triggers an update in the parallel output and the OE pin enables this output. Finally, the DS pin is where the serial input is given.

Want do build your own seven segment display driver? Detailed instructions on how to use a 74HC595 chip to drive a seven segment display can be found online, e.g., on in chapter 15 of this tutorial. Building such a driver is outside the scope of our workshop, however, it might be useful for you to read and understand the shifting itself and how it works. Please have a look at the mentioned tutorial.

2.2. Adafruit four-digit, seven-segment display with backpack

As described above, we could build our own driver for a seven-segment display. However, it seems to be fairly cumbersome to develop such a driver, especially since low-cost, pre-fabricated displays with communication "backpacks" exist. Here, we will use an Adafruit four-digit, seven-segment display that communicates with the Arduino via the inter-integrated circuits (I^2C) communications protocol.

I^2C protocol I^2C is a single ended, synchronous, multi-controller/multi-target, single bus communication protocol and is ideally suited for integrated microelectronics. The following figure shows the configuration with one microcontroller and three selected devices. (Credit: Tim Mathias, License: CC BY-SA 4.0)



The microcontroller connects to each device via a data (SDA) and a clock line (SCL). Furthermore, power (Vdd) and ground (GND) are connected as well; to light our LEDs we need power. The clock defines the timing of commands, i.e., triggers sending / receiving and the data line sends the required commands. More details on the I^2C communication protocol can be found on Wikipedia.

2.2.1. Assembly

The four-digit, seven-segment display package requires assembly by soldering. Figure 2.3 shows the individual components of the display that need to be soldered together. On the top left is the display itself. The backpack is shown on the bottom and the header pins, which allow us to connect the display to the breadboard, are shown on the top right. If you have never soldered before, please review the this video that will show you some best practices. Generally, remember to:

- Avoid excessive amounts of solder
- Ensure good contact between your component and the printed circuit board (PCB)

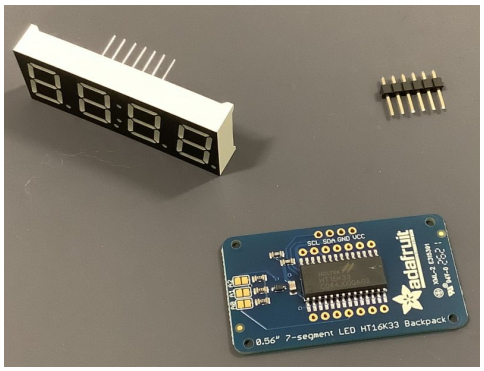


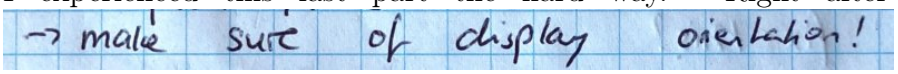
Figure 2.3.: *Parts of the four-digit, seven-segment display.*

- Double check your parts orientation

Types of soldering The two types of soldering generally referred to are through hole and surface mount soldering. In this workshop we will only perform through-hole soldering, i.e., we will stick the leads / pins through the PCB and solder from the other side in order to ensure a connection. Surface mount soldering mounts components on top of PCBs. A great tutorial video for this type can be found here.

One of the great features when using components and kits from Adafruit is that they have very detailed instructions. All instructions for the four-digit, seven-segment display can be found here.

Exercise 6 Assemble the display kit by following the Adafruit assembly instructions. Double, then triple check the orientation of the display prior to soldering; it is difficult to remove the display after soldering and resolder it in the correct orientation. (In fact, I experienced this last part the hard way. Right after I wrote



I soldered the display onto the PCB the wrong way around...)

2.2.2. Controlling the display from Arduino

As for the assembly, Adafruit has detailed instructions on how to connect and control the display. In fact, they even provide an Arduino library that allows us to easily control the display (this was one of the reasons for choosing this display to start with). This tutorial goes in detail through the setup for Arduino, including a how-to for installing

the libraries. For reference, Figure 2.4 shows an image of the display connected to the Arduino.

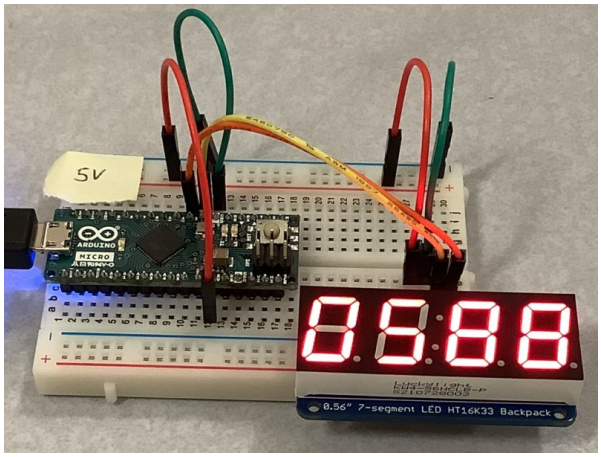


Figure 2.4.: *The display connected to the Arduino Micro on a breadboard.*

Exercise 7 Go through the Adafruit tutorial to connect the display to your Arduino. Then ensure that it works as expected by running the example script that Adafruit provides. Various different ways are given in the tutorial. Make sure you understand the different ways of controlling the display. Feel free to modify the example and display your own values.

Exercise 8 For our final setup we want the display to display the current temperature (2 digits plus +/- sign) and the set point if this is changed by pressing a button. Write a subroutine that takes two input arguments and allows you display the following values: `"␣-9"`, `"␣20"`, `"s-10"`. The last example is for displaying the setpoint with an additional `s` in front of it (which is equal to `5`). *Hint*: Your subroutine might start with something like this:
`void dispTemp(int temperature, bool setPoint) {...}`

3. Temperature sensor

Temperature is generally measured with a thermocouple. In these devices, two metals are brought together that, due to the Seebeck effect, create a temperature-dependent voltage between them. The voltages produced generally are very low and therefore require amplification before they can be read in by an ADC. Such an amplifier for a K-type thermocouple can, e.g., be found here. In this specific case, the amplification circuit would allow us to read temperatures from -250 °C to +750 °C, i.e., a total range of 1000 °C.

Question 1 The Arduino Micro that we are using contains eight 10 bit ADCs. Why is it very difficult to get a precise reading of the thermocouple described above using such an ADC? What would you need to implement / improve the precision?

3.1. Thermistor

While regular thermocouples have a very wide range, this range is not required for our setup. Here we are therefore using a thermistor, which is simply a resistor that is highly temperature dependent.

3.1.1. Voltage divider

In order to understand how we can determine the value of a thermistor with an Arduino, we first need to review how voltage dividers work. Figure 3.1 shows a schematic of a voltage divider built of two resistors with values R_1 and R_2. The current flows from V_{in}

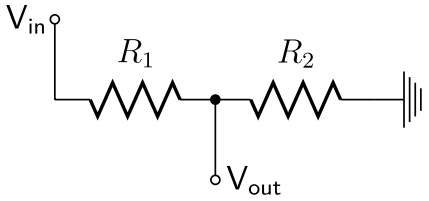


Figure 3.1.: *A resistive voltage divider. After Krishnavedala via Wikipedia, CC0.*

to ground. We can calculate the voltage measured between V_{out} and ground as following.

$$R_{\text{tot}} = R_1 + R_2 \tag{3.1}$$

$$I_{\text{tot}} = \frac{V_{\text{in}}}{R_{\text{tot}}} \tag{3.2}$$

$$V_{\text{out}} = R_2 I_{\text{tot}} \tag{3.3}$$

$$V_{\text{out}} = V_{\text{in}} \frac{R_2}{R_1 + R_2} \tag{3.4}$$

In equation (3.1) we first calculate the total resistance of the setup. Plugging this result into equation (0.2), we can calculate the total current flowing, equation (3.2). Since V_{out} is measured over R_2, we can simply calculate the voltage over this element as in equation (3.3), which when simplified results in equation (3.4). More details, as well as more general voltage divider setups, can be found on Wikipedia.

3.1.2. Using a thermistor with an ADC

In order to determine the resistance of a thermistor and thus to determine the temperature, we can use above described voltage divider setup. For example, we could use a reference resistance R_{ref} instead of R_1 in Figure 3.1 and replace R_2 with the thermistor that has a resistance R_{therm}. Connecting V_{out} to an ADC, we can then determine the thermistor resistance by solving equation 3.4 for $R_2 \equiv R_{\text{therm}}$. We can thus write

$$R_{\text{therm}} = \frac{R_{\text{ref}}}{\frac{V_{\text{in}}}{V_{\text{out}}} - 1}. \tag{3.5}$$

For an ADC with n bit resolution, we furthermore know that the measured value x relates to the voltage as

$$V_{\text{out}} = \frac{x \cdot V_{\text{ref,ADC}}}{2^n - 1}, \tag{3.6}$$

where $V_{\text{ref,ADC}}$ is the reference voltage of the ADC. Plugging equation (3.6) into (3.5) results in

$$R_{\text{therm}} = \frac{R_{\text{ref}}}{\frac{(2^n-1)\cdot V_{\text{in}}}{x \cdot V_{\text{ref,ADC}}} - 1}. \tag{3.7}$$

If the reference voltage of the ADC and the reference voltage of our voltage divider setup are the same, we can further simplify the equation to

$$R_{\text{therm}} = \frac{R_{\text{ref}}}{\frac{2^n-1}{x} - 1}. \tag{3.8}$$

3.1.3. Determining the temperature

Knowing the resistance R_{therm} of the resistor, we can determine the temperature by looking up the correct value in a lookup table, see the comma separated values (CSV) file on GitHub.

The temperature can also be calculated using the Steinhard-Hall equation, which is a model for the resistance of a semiconductor at different temperatures. This model states that the temperature T and resistance R_{therm} depend on each other as

$$\frac{1}{T} = a + b \ln R + c \left(\ln R\right)^3 . \tag{3.9}$$

Here, a, b, and c are constants. Doing a curve fit of this equation to the lookup table for the thermistor, we can determine the constants as $a = 2.78522 \times 10^{-3}\,\text{K}^{-1}$, $b = 2.41846 \times 10^{-4}\ \ln(\text{k}\Omega)^{-1}$, and $c = 8.31227 \times 10^{-7}\ \ln(\text{k}\Omega)^{-3}$.

A simplified version of above equation, the so-called B-parameter equation (see Wikipedia) assumes that the constants are $a = 1/T_0 - (1/B) \ln R_0$, $b = 1/B$, and $c = 0$. We can therefore rewrite equation (3.9) as

$$\frac{1}{T} = \frac{1}{T_0} + \frac{1}{B} \ln\left(\frac{R}{R_0}\right) . \tag{3.10}$$

Here, $T_0 = 298.15\,\text{K}$ is room temperature and $R_0 = 10\,\text{k}\Omega$ the resistance at T_0. The parameter B for our thermistor is $B = 3950\,\text{K}$.

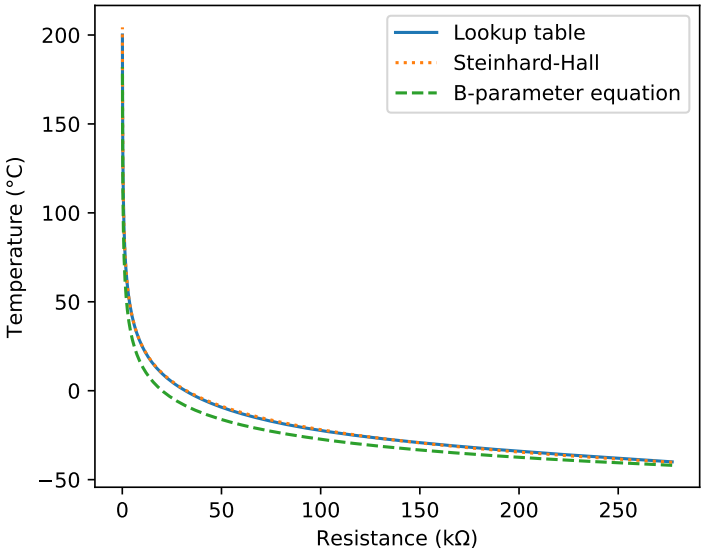


Figure 3.2.: *Temperature as a function of resistance for Adafruit 3950 NTC thermistor*

Figure 3.2 shows the temperature determined using the lookup table in comparison with fitting the Steinhard-Hall equation (3.9) and the B-parameter equation (3.10) results. While the B-parameter equation gets close to the real values, it still has a significant offset. The Steinhard-Hall fit to the lookup table on the other hand seems to be a fairly good match.

Question 2 How will you determine the temperature? The lookup table has the most precision, however, what would you do with a resistance that is in between two values in the table? Our setup will have an actual temperature range between around 25 °C and -20 °C. How well do the methods compare in this range?

Question 3 The Arduino Micro has a 10 bit ADC on board. Assuming a reference resistor $R_{\text{ref}} = 10\,\text{k}\Omega$, calculate the smallest temperature difference that you would be able to determine with this ADC in the above defined temperature range.

3.2. Reading the temperature with the Arduino

We will now use the thermistor in order to measure the room temperature. With above introduction, you should be ready to build the setup by yourself. However, there are also detailed instructions available on Adafruit's website on how to set this up correctly. Note however the uncertainties in the different approaches for determining the temperature shown in Figure 3.2.

Exercise 9 Connect the thermistor using the provided reference resistor to your Arduino and measure the voltage on any of the analog input pins (ADC[0] through ADC[7] in Figure A). If you connect your input to `sensorPin = A0;`, you can read the ADC value by calling `sensorValue = analogRead(sensorPin);`. Read the resistance value out using the serial monitor. If you have trouble reading from the ADC, check out the Arduino example under "Analog", "AnalogInput". *Note:* Make sure that the reference voltage of the ADC is the same voltage as the one you are using for V_{in}.

`int sensorPin = A0;` In above exercise, the ADC pin was initialized as an integer with the value `A0`. Why does this not throw an error?

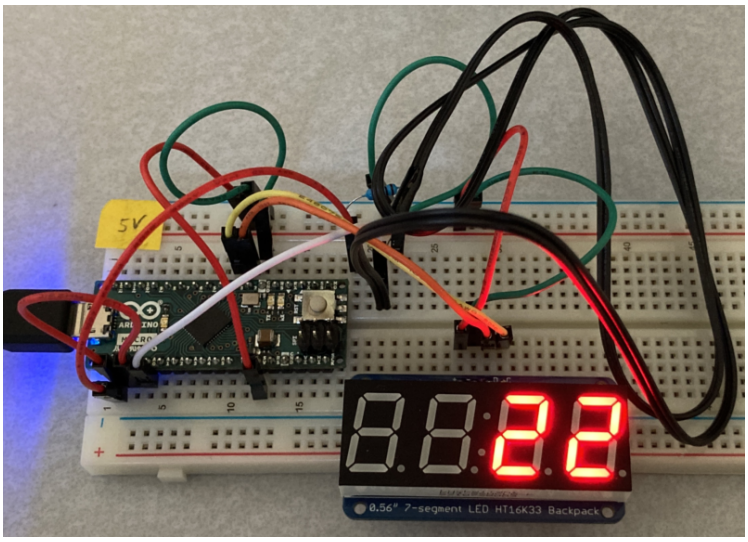


Figure 3.3.: *Temperature reading via thermistor and shown using the the four-digit, seven-segment display.*

Exercise 10 In Section 3.1.3 we have discussed various approaches of determining the temperature. Implement your solution from above's question in order to calculate the temperature and print it out using the serial monitor. How precise are your readings? Does your room temperature measurement make sense?

Exercise 11 Instead of printing the temperature via the serial monitor, display the temperature on the four-digit, seven-segment display that you have controlled in Chapter 2. Be concious about the environment and recycle – even your code!

Figure 3.3 shows the final setup after the last exercise. Note that your setup might of course look slightly different.

Your temperature readings might be varying a lot from measurement to measurement. The reason for this is electronic noise. There are two ways of reducing the effects of such noise: (1) For your result use the average of several measurements. (2) The 5 V line of the Arduino is fairly unfiltered and depends on the power supply you are using. The 3.3 V Arduino line (see Figure A) on the other hand is fairly stable, and it would therefore be advantageous to use this line for ADC measurements. To do so, switch your V_{in} to use the 3.3 V line. You can provide this line to the ADC as a reference voltage. To do so, also connect the `AREF` pin to the same 3.3 V line.

Exercise 12 Using above methods of averaging and using the 3.3 V as your measurement and reference voltage, determine if your temperature measurements improve.

Finally, as Physicists we know that an ice-water mixture at standard pressure and temperature (we are close to the ocean, so let's assume that's given) is by definition at 0 ℃. The thermistor that you have is waterproof, and you can therefore submerge in such a "calibration solution".

Exercise 13 Determine the temperature of an ice-water mixture using your code. How precise is your determination of the temperature? If you are off by several degrees, there might be an error in your temperature measurement capability and your code might need correction.

4. Thermoelectric cooling element

The final part that we need to implement in order to build a sample cooling system is the actual cooling element. As a cooling element, we will use a Peltier cooler from Adafruit, which is mounted onto a heat sink and fan assembly.

4.1. Components and their Physics

4.1.1. The Peltier effect

Thermoelectric coolers make use of the so-called Peltier effect in order to create a heat flux between two different types of materials. This means that heat flows from one element to the other, i.e., that one side gets cold and the other one hot. The Peltier effect was discovered in 1834 by French Physicist Jean Charles Athanase Peltier. In principle, it works similar to a thermocouple, which we discussed above. Figure 4.1 shows a schematic on how the Peltier effect is used to create a thermoelectric cooler. Here, an n- and a p-doped semiconductor are brought together. If we simply measure the voltage across the two metals, we can use the setup as a thermocouple. (Note that this is different from a thermistor, see introduction to Chapter 3.) If we power the setup

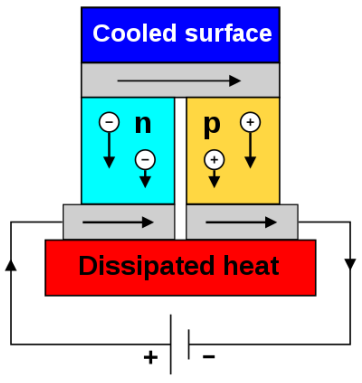


Figure 4.1.: *The Peltier effect acting on a thermoelectric cooler. Credit: Ken Brazier via Wikipedia, License: CC BY-SA 4.0*

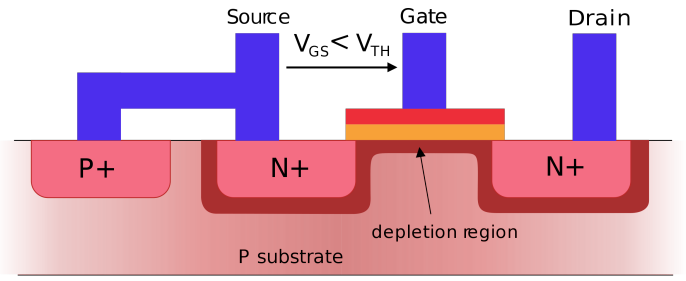


Figure 4.2.: *A cross-section through an nMOSFET. See text for a detailed description. Credit: Wikipedia, License: CC BY-SA 3.0.*

with a DC voltage, a heat flow is generated. The heat flow per unit time is given as

$$\dot{Q} = (\Pi_m - \Pi_n)I. \tag{4.1}$$

Here, Π_m and Π_n are the Peltier coefficients of the two semiconductors and I is the current. Not that the amount of heat flow is directly proportional to the current.

A typical thermoelectric cooler contains many semiconductor junctions added together in series. Furthermore, in order to achieve efficient cooling on one side, the heat from the hot side needs to be removed. This is generally done by mounting the hot side on a heat sink. In our case, the heat sink is connected to a fan which helps to remove heat faster.

4.1.2. MOSFET

The Peltier cooler that we will be using draws 5 A of current at 12 V. The voltage and especially the current are too high for any Arduino pin. We therefore need to implement a switch that we can turn on and off via an Arduino pin and that can handle high currents in order to drive the cooling element. One type of switch that can drive our circuit is a metal-oxide-semiconductor field-effect transistor (MOSFET).

Question 4 One issue of driving the cooler directly from the Arduino is the that the pins only put out 5 V. At this voltage, what current would be drawn by the Peltier cooler? Compare this current with the maximum current that an Arduino pin can supply. What would happen if you try to drive the cooler off a 5 V pin?

MOSFETs are ideal devices to use as high-current switches that can be rapidly turned on and off by simply using an Arduino pin. Figure 4.2 shows the inner workings of

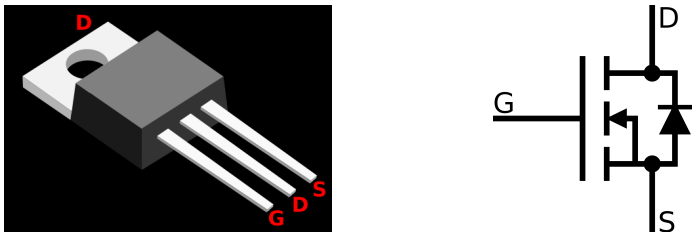


Figure 4.3.: *Schematic with pin assignments of the nMOSFET we are using here (left) and electrical symbol of a nMOSFET (right). Note the diode connecting the source (S) to the drain (D). The gate is labeled G. (Right image credit: ErikBuer via Wikipedia, license: CC BY-SA 4.0.)*

an n-type MOSFET, also often referred to as an NMOS. The source and drain on an nMOSFET are n-type materials, i.e., they contain a surplus of electrons and are therefore electron donors. These two channels are inside a p-type substrate which lacks electrons and is therefore an electron acceptor. When simply applying a voltage in between source and drain, the p-type material in between does not allow a current to flow. However, by applying a positive voltage to the gate, electrons are attracted to the gate area, which creates a conductive band allowing current to flow in between the source and the drain.

MOSFETs can have wildly different specifications. For Arduino electronics, we want to utilize the ones for which the gate switches when applying a low voltage of only 5 V. Furthermore, we need a MOSFET that can at least handle 5 A of current. At high currents, MOSFETs can get hot and might need to be mounted to a cooling element themselves. In our case however, the MOSFET can be air-cooled up to a current of ∼15 A, therefore, a cooling element will not be necessary at the 5 A used here.

Since MOSFETs allow for rapid switching, they can also be driven from PWM pins. This allows you to regulate the current to the cooler and therefore to cool it more and less efficiently.

Some applications for MOSFETs Using MOSFETs allows you to rapidly switch various voltages and work with high currents. Furthermore, the high-frequency switching allows you to control the current by connecting the gate to a PWM pin. For example, MOSFETs are frequently used as motor drivers. In order to have a motor run backwards it is often required to have a setup that can switch polarities. To enable this, an H-bridge can be used.

4.2. Implementing a MOSFET controlled Peltier cooler

Figure 4.3 shows a schematic (left) and the electrical symbol (right) of the nMOSFET that we are using here. The data sheet can be found here. As shown in the electrical symbol, this nMOSFET has an additional diode in between the source and the drain. This means that current can flow between the source and then drain even when we do not apply a gate voltage. For the nMOSFET to actually work as a switch, we need to set it up such that, when switched, the current flows from the drain to the source.

Question 5 With above mentioned considerations in mind, draw a circuit diagram that allows you to control the 12 V power supply to drive the thermoelectric cooler from a 5 V Arduino pin. In which part of your circuit diagram will the full 5 A current flow when the gate is activated? Make sure that the 5 V from the gate is connected to ground via a 10 kΩ resistor, such that it is not floating when turned off (remember that an Arduino I/O pin configured as an ouptut cannot be used as a voltage sink). Also make sure that the 12 V power supply and your Arduino share a common ground! If no common ground is established, the gate will be floating, which could damage the MOSFET by overheating it.

If you are having trouble with this exercise, a mock setup can be found on TinkerCAD. The thermoelectric cooler is here represented by a 2.4 Ω resistor. If you run the simulation, it will slowly increase the voltage every three seconds. The voltage and current applied to the "cooler" are displayed on a volt and ampere meter, respectively.

Exercise 14 Now implement the above wiring diagram in real life. Before you connect the 12 V to the actual power supply, make sure that:

- You did not connect 12 V to any Arduino I/O pins
- The full 5 A current does not flow through the breadboard at any point

Double check this carefuflly since, if not connected properly, you might destroy the Arduino and / or the breadboard. Also connect the fan of the thermoelectric cooler to the 12 V, however, connect it such that it is permanently on when the 12 V are connected.

As mentioned above, the breadboard cannot handle 5 A of current since the leads are too small. Small wires have a higher resistance per unit length compared to wires with larger diameters and therefore get hotter when a given amount of current flows through

them. We thus have to use appropriate wiring. In the US, wire sizes are standardized and given in units of american wire gauge (AWG). The lower the number in AWG, the thicker is the wire we are dealing with. Tables such as these ones show how much current can be carried by any given wire. As you can see, a 20 AWG wire can carry 5 A and would therefore be sufficient to be used for the Peltier cooler.

Exercise 15 Turn on the thermoelectric cooler with different PWM outputs on the gate and verify (1) that the cooler actually gets cold and (2) that the voltage applied across the cooler is what you would expect from your PWM output level. Control the cooler from a subroutine that allows you to easily set the power level from the main loop.

Exercise 16 Using Kapton tape, stick the thermistor on the surface of the thermoelectric cooler. First measure the room temperature and then the lowest temperature that the Peltier element can reach for varying control levels. To record the temperature you can either display it on the seven-segment display or print it out via serial.

Exercise 17 Now that you have an idea on how cool the Peltier element can get, set it to maximum cooling and turn off the cooling fan. What happens to the temperature and why?

5. Sample cooling stage

Over the last few chapters, we have slowly gone through all the individual components that we need in order to assemble a complete sample cooling stage with feedback control. Let us now take this knowledge and assemble the whole project. Figure 5.1 shows an image of a complete setup. In the end, our setup should be able to do the following:

- Run independent of a computer
- Have two LEDs: A green one indicating that the thermocouple is within the specified temperature range and a red LED showing the PWM output to the MOSFET that drives the cooler
- A display showing the current temperature or the set temperature, depending on the mode the device is in
- Buttons to enter / exit the set mode and additional buttons to increase / decrease the set temperature
- MOSFET PWM control to drive the thermoelectric cooler

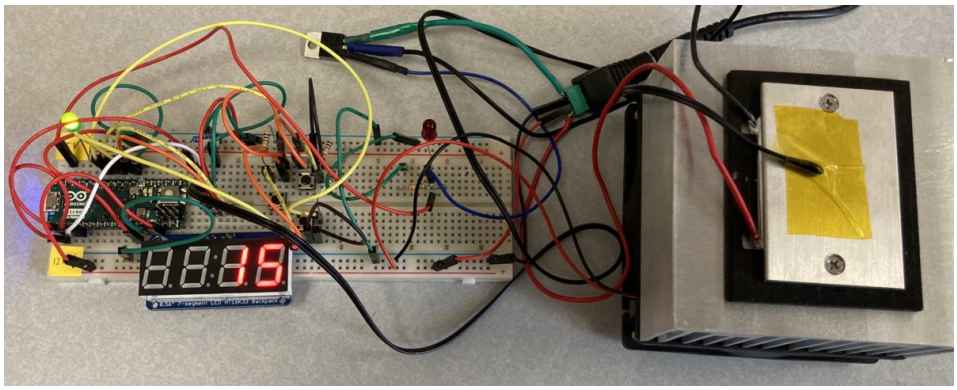


Figure 5.1.: *Complete setup of the sample cooling stage, set to 15 °C.*

Breadboard power busses We use 5 V and 12 V power for various components. It is recommended that you wire both power buses on the breadboard, one with 5 V from the Arduino and one with 12 V from the DC power supply. For the 12 V bus: Be careful to ground it through the DC power supply and do not connect any Arduino pin other than the `VIN` and ground pins (see Figure A). Before you connect even these pins, read Section 5.6.

The following sketch gives one potential setup for implementing the full controller. A template Arduino `ino` file with this content can be found on GitHub.

```
// Load libraries
// Adafruit Display initialization
// Variables and Pins for temperature measurement
// Variables for set temperature
// Thermoelectric cooler variables and Pins
// Button pins to set temperature
// LED Pins and temperature happiness range

void setup() {
  // Initialize Adafruit display
  // Initialize pin modes for buttons, PWM for cooler, LEDs
}

void getTemperature() {
  // Read thermistor, calculate and store temperature
}

void setDisplay(int valueToSet, bool setpoint=false) {
  // Set the display with a value to set (integer)
  // Additionaly display 'S' if we are in set mode
}

void setMode() {
  // Check if we are in set mode and adjust display
  // If in set mode, adjust set point if called for
}

void controlTEC() {
  // Control the termoelectric cooler.
}
```

```
void checkTemperatureOk() {
  // If temperature in given range, turn green LED on
}

void loop() {
  // Your main loop to hang everything together
}
```

In the following chapters, we will slowly expand this template until the controller is working. For all the individual subroutines you should already have existing code from the previous exercises. Recycle your code when adequate! You can also initialize the serial console in order to print to the screen when necessary. Note that the following chapters all build on the above template and on each other.

5.1. Read the temperature

As a first step, we want to read the resistance of the thermocouple and then transform the read value into a temperature in °C. You should already be familiar with this task from Chapter 3. Make sure that your temperature reading is stable, i.e., average a few measurements for the best results.

Exercise 18 Define an overall global variable `currentTemperature`. Then re-use your previous code and put it into `void getTemperature() {...}` such that this subroutine, when called, writes the current temperature to the variable you just defined. Then read the current temperature and print it to the serial console from the main loop.

5.2. Integrate the display

In the next step, we will integrate the display. Make sure that you load the required Adafruit libraries. You have already controlled the display in Chapter 2. You can integrate a way to display the set point already, however, we won't use this integration for now.

Exercise 19 Integrate the display. Re-use your previous routine to display the current temperature in steps of whole degrees. You should also allow the `void setDisplay (int valueToSet, bool setpoint=false){...}` to take two variables, the temperature as an integer to directly display it and a boolean variable. If this boolean is `true`, the display should write an s (a `5`) into the first digit to indicate that we are in set mode (see next Section). After reading the temperature in the main loop, display it on the display.
Hint: It might be useful to round the temperature before displaying the floating point number as an integer. Otherwise, the displayed temperature will be rounded incorrectly when it is high, since C++ simply cuts off everything after the decimal point when converting a `float` to an `int`.

5.3. Set mode for the temperature

Since we want to use our setup independent of a computer, we want to allow the user to change the set temperature. You can use various interfaces for this. I am using a three button setup. The buttons do the following:

- `setModeButton`: Enter / exit (toggle) the set-mode
- `plusButton`: Increase the temperature if in set mode
- `minusButton`: Decrease the temperature if in set mode

Furthermore, you should also set reasonable upper and lower limits that the user cannot exceed. Such limits could, e.g., be 25 °C for the upper level (approximately room temperature) and -10 °C for the lower level. Think of these levels as limits to prevent the user from setting insane, unreachable set points.

Exercise 20 Implement the set mode. Pressing the `setModeButton` should toggle a boolean that allows you to decide if the set mode is on or not. If the set mode is on, the display should display the setpoint temperature. Pressing this button again will deactivate the set mode and display the current temperature again.
If in set mode, the plus and minus buttons should in- and decrase the set point, respectively. Check for button presses in the `void setMode() {...}` routine. Make sure that the button presses cannot happen too fast, e.g., include a delay after each press to allow for user interaction time (typically a few hundred milliseconds).

5.4. Thermoelectric cooler

Now that we have the set mode as well as the temperature reading established, we can implement the thermoelectric cooler. As a first step, we won't go into control theory and use simple on/off logic. The next chapter outlines further and fancier implementations of this control. Therefore, make sure that you already connect the cooler control to an I/O pin that is capable of PWM output. For now, we will simply turn the cooler on if the temperature set point is below the actual temperature, and turn it off otherwise.

Exercise 21 Implement the thermoelectric cooler to turn it on and off, depending on the setpoint and current temperature comparison (see above). Implement this comparison in the `void controlTEC() {...}` subroutine and call this subroutine from your main loop. Connect a red LED to the MOSFET control pin, this LED will then tell you if the cooler is on or off.

5.5. Status LED

Finally, we are going to incorporate a green LED that will allow us to see at once if the system is at temperature or not. Let us, e.g., define a temperature range of ±0.5 °C in which we would call the achieved temperature acceptable.

Exercise 22 Set up a green LED. This LED will be controlled in the routine `void checkTemperatureOk() {...}`. In this routine, compare the current temperature to the set point and turn the LED on when you are inside your defined happiness zone, otherwise turn it off. Note: the function `abs()` might come in handy: it takes the absolute value of a given number.

5.6. Run independent of a computer

So far, we have always powered the Arduino from the 5 V that come from the USB connection from the computer. However, most boards also have a `VIN` pin. This allows us to connect an external power source and not use the computer at all. The recommended voltage on the `VIN` pin is 7-12 V and the maximum voltage 6-20 V.

Since we have a 12 V power supply to drive the thermoelectric cooler, we can use the same power source to drive the Arduino. Since we are using a 5 A power supply, we won't have the full current that can be used by the cooler anymore, however, the Arduino only uses very little power, so this should not matter too much. Otherwise, we could always replace the power supply with one that has a higher current rating.

Exercise 23 Connect an Arduino ground pin to ground (the negative side of the 12 V supply) and the `VIN` pin to the 12 V. Be careful not to mix up the pins, you might destroy your Arduino otherwise! Your setup should run now without a computer connection. If you need to restart the Arduino for any reason, you can use the reset button that is onboard.

5.7. Testing

If all went well, your setup might look similar to the one shown in Figure 3.3. Congratulations!

Finally, before you want to use your temperature controller, you might want to test it for stability. Does it really fulfill your requirements or do you need to adjust the capabilities? Some test ideas are outlined in the following list:

- How stable is your set temperature over the long run? Does the green LED stay lit once it reaches the desired temperature?
- How about changing the "load"? Put a glass of water on the cooler and submerge the thermistor to measure the water temperature. You have now added a ballast. Does the temperature stay constant once the set point is reached?
- Narrow the temperature range that you decide is acceptable and check if your setup can hold this narrower range
- Interface test: Give the device to a friend and check if the interface is intuitive
- Is your setup feasible / can it be used in a production scenario?

For many of these tests you might actually realize that your setup still needs improvements. These improvements are likely in design, setup, and integration of the cooling control. The next chapter gives you an outlook on where to go from here, i.e., what else you might want to think about if you really want to use this setup in your lab.

6. Further improvements

By now, you have developed a Peltier cooler and can control its set temperature using a few buttons. This project has hopefully also given you a look into how to build scientific setups with Arduino. We elaborated on digital and analog I/O pins in order to control and read all kinds of different sensors and devices. In this last chapter, we will now discuss possible further developments for this project in order to direct the interested reader on what could be done next.

6.1. Rigid setup

The breadboard setup is great for testing and quickly rearranging cables, however, it is not well suited for regular use once you have finished developing your project. If you want to use the developed setup further, several steps can be taken to make it more rigid.

6.1.1. Soldering a fixed setup

To have a more robust setup, it is often worth to solder the components onto a board. One possibility on how to solder the components into place is to use a perma-proto board, see here for one example. These boards work similar to the bread board that you used, however, you have to solder all the connections in place. Alternatively, you could use a perfboard, which simply has holes and no lines connected. There are various versions available for these components, and it is ultimately up to you to choose what you prefer.

Figure 6.1 shows an image of the setup that we developed soldered onto an Adafruit perma-proto board. The Arduino and the display in this case are not soldered in place. We rather soldered female headers for these components to plug in. This allows us to easily remove these parts if required.

6.1.2. Case

To have a lab setup that is used actively, it is often useful to not only solder the setup that we want to use regularly, but also to put it in some type of enclosure. Many enclosures

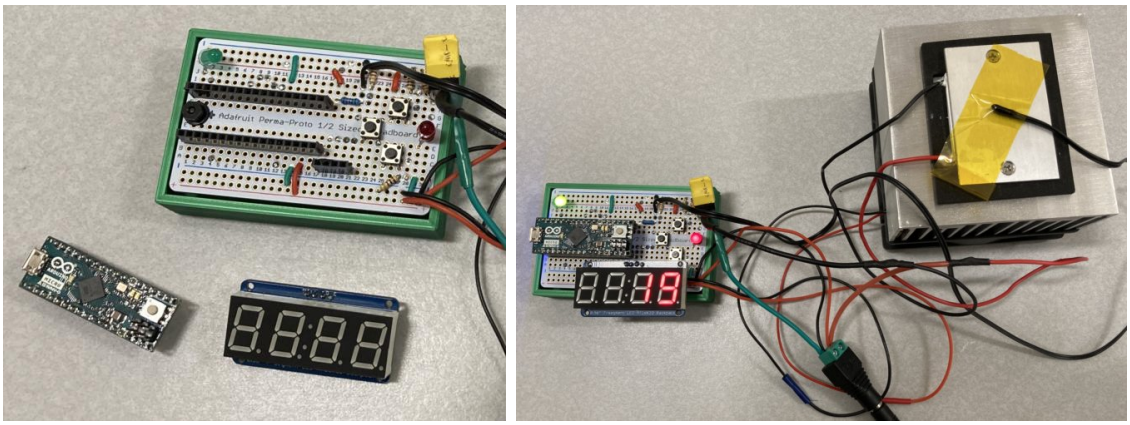


Figure 6.1.: *The assembled project with a 3d printed case. Parts unplugged (left) and the assembled project running (right). Most of the cables run underneath the protoboard and are hidden from view.*

are feasible and possible. For example, you could purchase an existing enclosure and simply adopt it for your needs.

Another possibility is to print an enclosure for your setup using a 3D printer. For example, you could design a case using computer-aided design (CAD) software such as FreeCAD. An example for a simple enclosure that keeps the bottom of our setup enclosed is shown in Figure 6.1. The FreeCAD and associated files can be found on GitHub. Great resources on campus can also be found in the Brandeis Maker Lab.

6.1.3. PCB development

Today, there are suppliers that produce small quantities of PCBs for low prices. It can therefore be worth if your project needs to be duplicated multiple times to design and manufacture a PCB for it. Many hints on how to do so can be found online, e.g., here. Tools to design PCBs are, e.g., KiCAD and EasyEDA. A great resource on campus to consult is the Brandeis Automation Lab.

6.2. Proportional control

When fully assembled, we have seen that our simple implementation was already capable of keeping the Peltier element within 0.5 °C of the set temperature when the thermistor was directly stuck onto the cooling element. This was already achieved when simply turning the controller on and off. However, we have also demonstrated that we can in fact drive the cooler by setting the PWM accordingly.

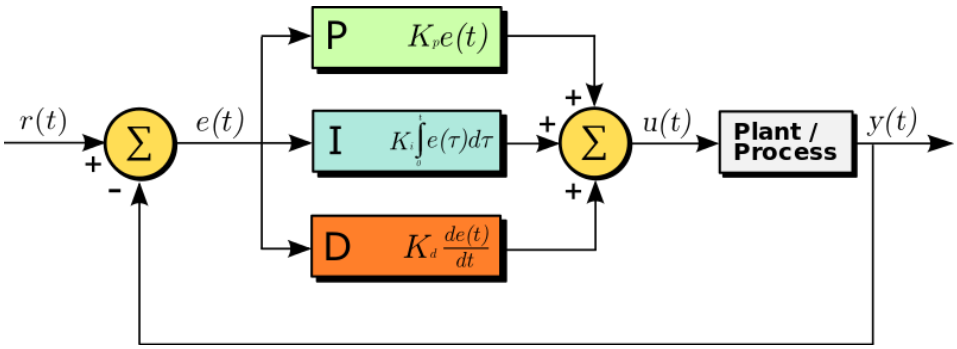


Figure 6.2.: *PID control block diagram in a feedback loop. Credit: Arturo Urquizo, License: CC BY-SA 3.0.*

6.2.1. Proportional–integral–derivative (PID) controller

A PID controller corrects a given system, e.g., our cooler, based on proportional (P), integral (I), and derivative (D) terms. Figure 6.2 shows a block diagram of the control loop for a PID controller. The set point is denoted $r(t)$, the achieved point $y(t)$. The "plant / process" in this diagram would be our thermoelectric cooler. From the set point and current value, an error $e(t)$ is calculated. Using this error in multiple terms, here all three PID terms, a response $u(t)$ is calculated. The overall control function takes the form

$$u(t) = \underbrace{K_p e(t)}_{\equiv P} + \underbrace{K_i \int_0^t e(\tau)d\tau}_{\equiv I} + \underbrace{K_d d\frac{de(t)}{dt}}_{\equiv D}. \tag{6.1}$$

Here, the constants K_p, K_i, and K_d are positive constants that have to be determined for every system.

In PID, the first term (P) will make a proportional response, e.g., if the error $e(t)$ is large, a large correction will be made. The second term (I) integrates all errors that were calculated so far and therefore creates a memory effect. The last term (D) is best described as a prediction value. A much more detailed explanation of PID control theory can be found on Wikipedia.

6.2.2. Proportional controller for our cooler

Let us try and implement a proportional feedback for our cooler. Since we can only cool, we can only apply the proportional part when our sample is too warm, otherwise, we will continue to simply turn the Peltier element off and let it warm up by itself. We can apply the proportionality by in- and decreasing the PWM output.

First, let us define K_p in units of K^{-1}. At full power, we will drive the PWM at 100%. This level will proportionally drop with lower values of $e(t)$. Since we cannot drive the

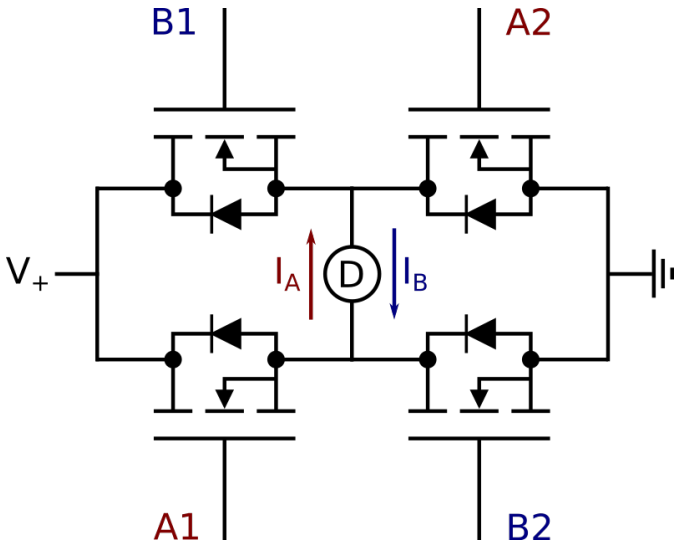


Figure 6.3.: *Schematic of an H-bridge. The "H" shape is laying on the side in this Figure. If the MOSFETs labeled A1 and A2 are closed, the current will flow along arrow I_A through device D. If the MOSFETs labeled B1 and B2 are closed, the current will flow along arrow I_B. The nMOSFET drawing is adopted from ErikBuer via Wikipedia, license: CC BY-SA 4.0.*

gate of the MOSFET with more than 100%, we will automatically generate a control function that contains a step in it. For a given K_p and a given temperature difference ΔT, we can calculate the response $u(t)$ (from 0 to 1) as:

$$u(t) = \begin{cases} K_p \Delta T, & \Delta T \leq \frac{1}{K_p} \\ 1, & \Delta T > \frac{1}{K_p} \end{cases} \tag{6.2}$$

An example implementation for $K_p = 0.5\,\mathrm{K}^{-1}$ and our thermoelectric cooler can be found on GitHub. Note that we only had to adopt the `void controlTEC()` routine.

6.3. Heating and cooling

The Peltier effect (Section 4.1.1) and thus the thermoelectric cooler can be run in either direction. This means that if we would reverse the polarity of the applied voltage and therefore reverse the current flow, the side that has so far been used as a cooler would start to heat up and the other side would cool down. This also allows us to actively drive the thermoelectric cooler as a heater.

So far, we used one nMOSFET in order to drive the cooler, see Chapter 4. Adding three more nMOSFETs, we could build a so-called H-bridge. Figure 6.3 shows a

schematic drawing of an H-bridge. Note that the "H" is laying on its side, the two vertical bars represented by the two nMOSFETs and the horizontal bar with the device D in the middle. If we apply voltage to the gates of the nMOSFETs A1 and A2, the current will flow in direction I_A through the device D. If we, on the other hand, apply voltage to the gates for nMOSFETs B1 and B2, the current flows in the reverse direction, as indicated by arrow I_B.

H-bridges are typically used to drive DC motors since the reversal allows us to drive the motor in either direction. You can also find assembled H-bridges online, e.g., here. Make sure that the current rating is appropriate for your application and that you understand how the H-bridge works. For example, the circuit shown in Figure 6.3 would allow you to short the circuit by opening A1 and B2 or B1 and A2. Such shorts should generally be avoided in order to not damage the components.

6.4. Data recording

6.4.1. Record data on storage medium

Various possibilities exist to log data using an Arduino. For example, you could use a data logging shield, i.e., an extension board, that allows you to plug an SD card into the Arduino and record data on it. Libraries to record data as well as examples can be found on the web, e.g., here.

6.4.2. Data in the cloud

Certain types of Arduino boards that have web capabilities can also be used in combination with the Arduino internet of things (IoT) cloud. This allows you to connect your setup to the internet and record measurement points online. A free plan allows you to test out the Arduino IoT cloud.

6.4.3. Data recording via serial and python

Another, interesting option is to use the serial interface that Arduino comes with by default and read out the serial commands. Below are example files, full, working examples can be found on GitHub.

Arduino setup We have so far used the serial protocol in order to print information from the Arduino to the serial console. Let us set up an Arduino with an LED connected to `ledPin`. We also store a `float result = 23.5`, which is the thing we want to print via serial when asked for it. In the setup, we therefore want to put the following code:

```
void setup() {
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}
```

Here, we first set up the LED as an output, and then initialize the serial interface with a baud rate of 9600 bits per second, see here for more information.

Let us assume that we have set up a function called `blink_led()` to blink the LED. We can then set up the main loop code as following:

```
void loop() {

  char inByte;  // where to store the read data

  if (Serial.available() > 0) {
    inByte = Serial.read();

    if (inByte == '?') {
      Serial.println(result);
      blink_led();
    }
  }
}
```

Here, we first initialize a variable for one single character. We then ask if Serial data are available and if they are, we read each character into `inByte`. It is important. If this character is equal to `?`, we print the defined result back to the serial console and blink the LED, otherwise we do nothing.

Two things that you should notice here: (1) If you send one question mark from the Arduino serial console, and it is set up to send a newline at the end, you in fact send two characters, a question mark and a newline character. However, the program only responds to the question mark and all other characters are ignored. (2) Note that we send the result using the `Serial.println()` command, i.e., we print a newline character at the end of our return. Assuming there are no other newline characters in the result, we can use this line termination scheme to later determine where the result has in fact terminated. This will come in handy further down.

Querying the serial interface with Python While the Arduino serial console is great for debugging, it is not made to record and log data on a regular basis. However, we do not need to communicate with the Arduino serial console via the Arduino IDE but can use any console that can communicate (send and receive commands) via serial in order to build our data logger. Below example shows one very simple way of communicating

with the Arduino via the Python serial interface. Here, we use the `pySerial`[1] library for communication. The following code presents a very simple python script to get and print the result:

```
import serial

PORT = "/dev/ttyACM1"

dev = serial.Serial(PORT, 9600)

dev.write(b"?")
result = dev.readline()

print(result)
```

We first import the `pySerial` package. We then define the `PORT` where the Arduino can be found. On a Unix based system, the ports are in the same form as the above example. On Windows however, you would look for something like `COM3`. We then assign a variable `dev` to the device and use the `pySerial` package to open communication with the serial port. To query the temperature, we have to send a question mark to the Arduino. We can do so using the `write` command and send a question mark that has been encoded to binary, i.e., `b"?"`. We then read the next line that came back from the device, store it as a `result`, and then print this stored value. Note that the `dev.readline()` command works only if you send a newline command from the Arduino, e.g., by using `Serial.println()`.

Once you run this program, which can be found here on GitHub, you will see that (1) the LED blinks when the program runs and (2) that the string `b'23.50\r\n'` is returned. The returned string is of course also binary encoded and, in order to get a regular string as you are used to, you would have to decode this binary string. Furthermore, you might want to strip the carriage return (`\r`) and newline (`\n`) characters at the end of the line. You could clean up the result string as following:

```
result = float(result.decode("utf-8").rstrip())
```

This decodes the result, strips the empty characters on the right side of the string, and then converts it to a float.

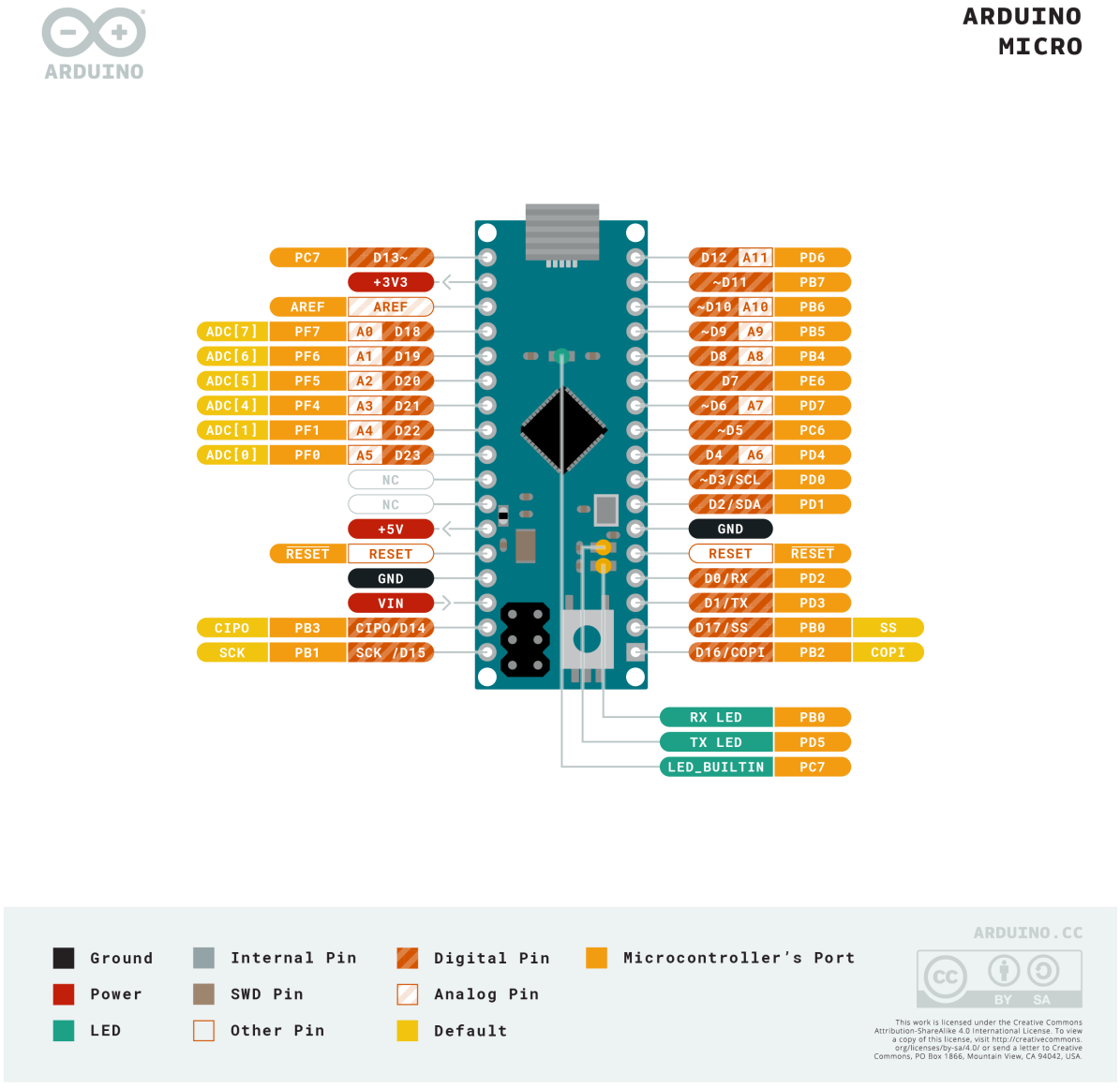
We can of course also extend this simplest possible example into an actual data logger. A simple example how this could be achieved in Python can be found on GitHub with the filename `data_logger.py`. The python file is extensively commented and records a given set of measurements with defined intervals in between and saves them as a CSV file. The important part to remember is the following: The serial interface allows you to easily control and drive the Arduino from your computer. If you adjust your firmware

[1] The `pySerial` interface can be installed as: `pip install pyserial`

accordingly, you will be able to interact with your Arduino by other means than just the Arduino IDE, e.g., via Python. This can be extremely useful for further development and data recording.

Appendices

A. Arduino Micro Pinout



B. Bill of materials (BOM)

The following table states all components used in this workshop. It assumed that soldering stations and supplies are provided.

Component	Supplier	Article #	Amount	Cost total
Arduino Micro	Digikey	A000053	1	$20.70
USB Cable	Digikey	102-5943-ND	1	$2.55
Breadboard	Adafruit	239	1	$5.95
Breadboard wires	Adafruit	153	1	$4.95
LED red	Digikey	516-1339-ND	1	$0.81
LED green	Digikey	516-1334-ND	1	$0.78
Buttons	Adafruit	367	1	$2.50
Resistors 1 kΩ	Adafruit	4294	1	$0.75
Resistors 10 kΩ	Adafruit	2784	1	$0.75
Display	Adafruit	1002	1	$10.95
Thermistor	Adafruit	372	1	$4.00
Thermoelectric cooler (TEC)	Adafruit	1335	1	$34.95
MOSFET	Adafruit	355	1	$1.75
12 V power supply	Adafruit	352	1	$24.95
Power jack adapter	Adafruit	368	1	$2.00
Total cost:				**$118.34**

C. Open source design tools

C.1. Calculators

- **Heat sink calculator** to determine if you need a heat sink or not. https://daycounter.com/Calculators/Heat-Sink-Temperature-Calculator.phtml

C.2. Designing

- **EasyEDA**Online designer for PCBs with large library of components and direct ordering possibilities. https://easyeda.com/
- **Fritzing** Makes electronics design accessible. Easy and simple interface to draw some of your own setups, quick to get started with. Suppliers such as Adafruit have Fritzing libraries with components that you can import. https://fritzing.org/
- **KiCad** Cross-platform electronics design suite. https://www.kicad.org/

C.3. Virtual Hardware

- **TinkerCAD** Electronic playground from Autodesk. Allows you to virtually set up electronic components and write / test code for them. https://www.tinkercad.com/

D. Full wiring diagram

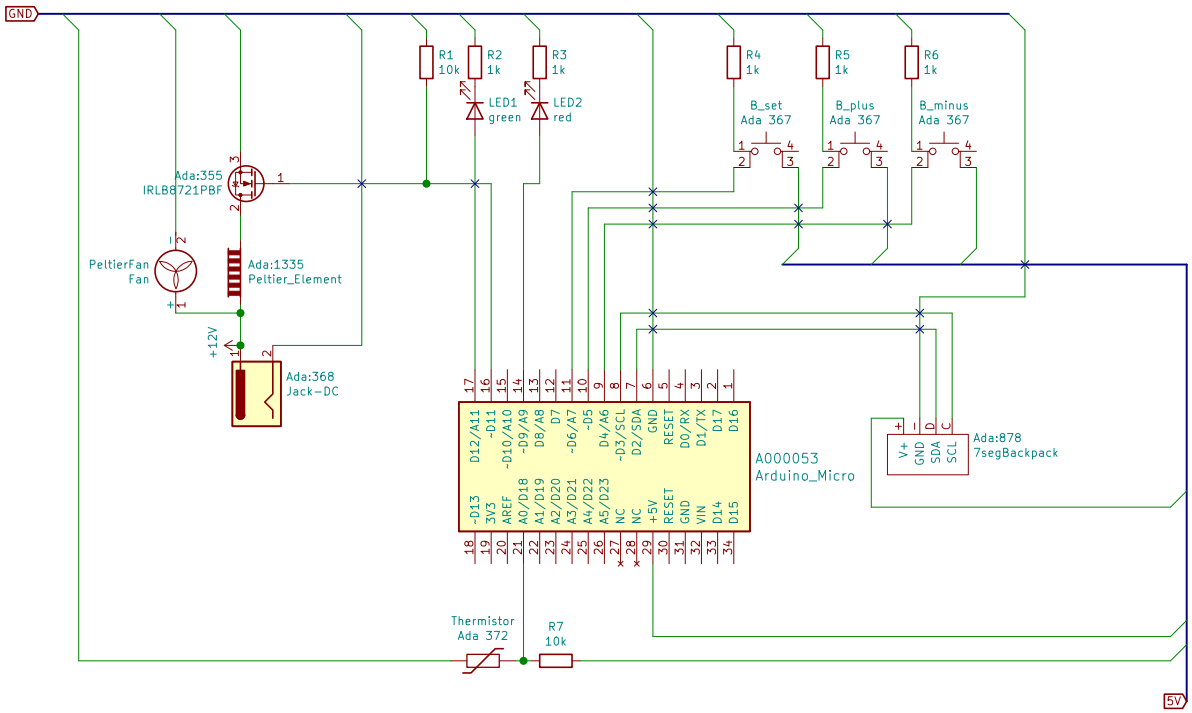


Figure D.1.: *Full wiring diagram of the final project. All KiCAD files can be found on GitHub.*