senseBox:edu

Dokumentation
Open Educational Resources

Table of Contents

senseBox:edu

The senseBox:edu is a toolbox which helps teach programming to students and junior researchers in a playful manner.

The Arduino-based set of electronic components is used to build increasingly complex circuits, which are controlled via a microcontroller.

Along with the learning resources provided on these pages, young students can gather experience in (Arduino) programming and electronic circuits with hands-on training.

About this book

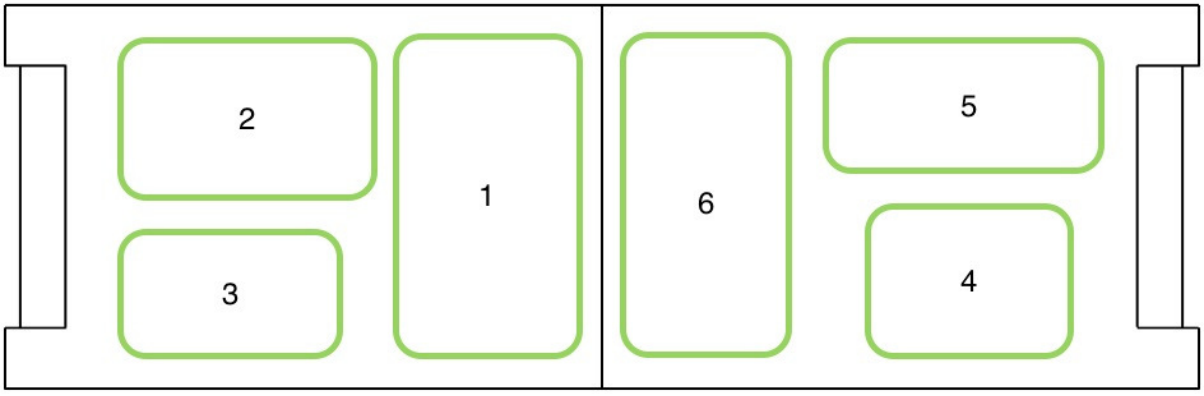
In this book, all the resources regarding senseBox:edu may be found. This includes tutorials, exercises, example projects, as well as links to downloadable material.

Contributions (improvements to existing content, or entirely new content) are appreciated! We're happy to include documentation you have made for projects using senseBox in this book. To do so, have a look at our contribution guide.

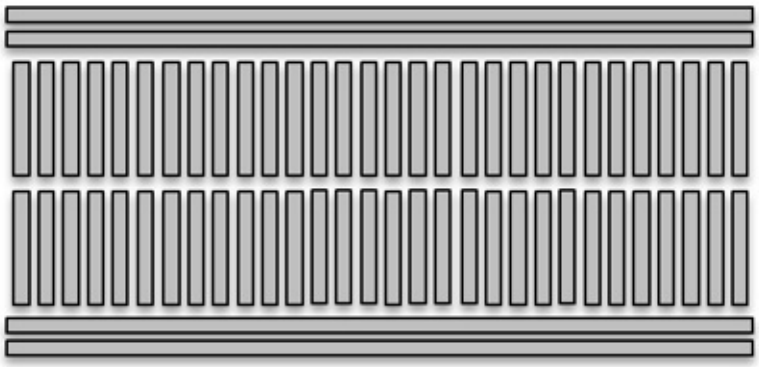
The source code for this book is available on GitHub, where you may also leave feedback about this book.

If you'd like to download this book as a PDF document for printing, check out our download section.

Content of the box



1. Arduino & Breadboard: All of the senseBox experiments are based on the main board. It consists of an Arduino Uno microcontroller, a senseBox shield, as well as a breadboard. The breadboard enables plug & play interconnection of electronic components on its many conductive lines. It has two pairs of vertical lines (marked with `+` and `-`), and 30 horizontal lines with 5 connectors each (marked `a` to `e`, resp. `f` to `j`).

   The senseBox shield is an Arduino extension board, that is plugged in on top of the Arduino connectors. It provides a microSD-card slot for data storage, a real-time clock, as well as additional I^2C connectors. The RTC will be enabled as soon as the battery is inserted into the shield.

2. Power supply: During programming and experiments, you can provide power to the microcontroller via the included USB cable. Once you have completed your project, you may run the Arduino without a computer attached by using the included power supply.

3. Batterypack: A batterypack allows you to use your senseBox on the go! This is especially useful if you want to do measurements at multiple different places.

4. Ethernet- or WiFi-Shield: The senseBox includes another shield for network connectivity. Depending on the senseBox version, this is a red ethernet shield or a blue WiFi-shield. By attaching this to the arduino, you can upload your data to the openSenseMap for example.

5. Wires: Besides the USB-B cable, there are many jumper wires with mixed male and female connectors included. These allow you to build your circuits on the breadboard in no time! Also there is a cable for the batterypack included.

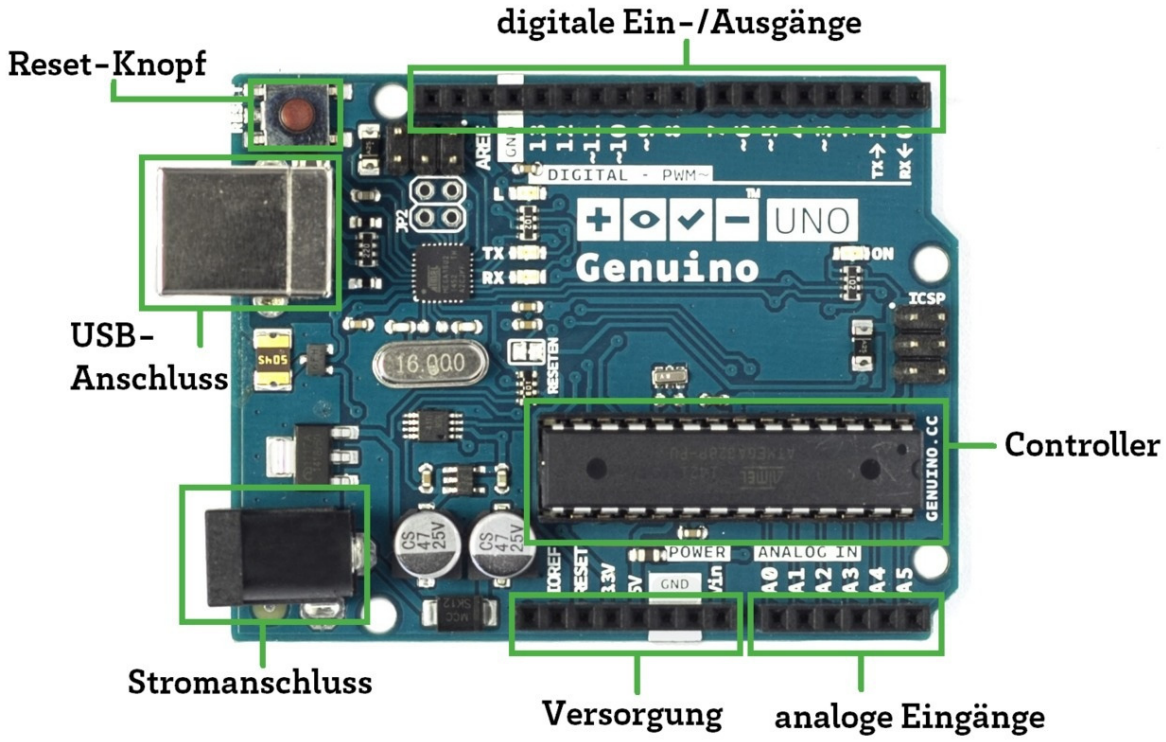
6. Sensors and electronic components: The following list shows all the sensors and other electronic parts that are needed for the experiments:

   - Air temperature and humidity sensor (HDC1008)
   - Air pressure and temperature sensor (BMP280)

- Optical infrared distance sensor (GP2YA)
- Ultrasound distance sensor (HC-SR04)
- Light intensity sensor (TSL45315)
- UV-A light intensity sensor (VEML6070)
- Photo resistor
- Microphone (CEM-C9745JAD462P2.54R)
- 5mm LEDs (red, green, yellow)
- RGB-LED (BL-L515)
- Resistors (470 Ω and 10 kΩ)
- Push buttons
- Potentiometers
- Piezo-Buzzer
- 2GB microSD card and SD-Adapter
- Battery for senseBox shield (CR-1225)

The Genuino board

The Genuino UNO is a microcontroller board that is especially designed for developing prototype circuits. Besides the Genuino UNO, there are a lot of other microcontroller boards on the market.



Above you can see the Genuino UNO and its main components, which we will introduce in the following sections. The other components are important as well, but we would like to focus on the main parts.

USB connection

To transfer the programs you've written on your computer to the Genuino UNO microcontroller, we use the USB connection. In addition to data transfer, the USB connection can also work as a power supply when there is no other power source.

Power supply

As you may have noticed, this connection provides the microcontroller with power. The Genuino UNO runs on 5V. When using an external power supply, however, 6V - 20V are recommended.

Reset button

The Genuino UNO reboots after pressing the reset button. This means that the sketch will reset and start from the beginning. When you are unsure whether the Genuino UNO is running properly, you can press the reset button.

Digital input and output

To communicate with digital sensors and other components, you can connect to the digital pins. This enables you to perform many actions including reading sensor values or prompting LEDs to blink. Furthermore, it is also possible to output analog signals with the digital pins. The difference between analog and digital signals is explained in the following chapters.

Analog input

Similar to digital pins, analog pins can read out sensors and provide data to the Genuino UNO. The microcontroller does not support analog output. This function is provided by the digital pins.

Power supply pins

The power supply pins are neither digital nor analog pins. They provide power to the connected components. Furthermore, it is possible to provide the Genuino UNO itself with power through several pins.

Controller

The controller is the brain of the Genuino UNO. It executes all processes.

Software installation

To program the Arduino, you need to install the Arduino IDE and drivers. Depending on your OS, there are different ways to install the software.

You can find the latest installation files here.

Installation on Windows

After downloading the Arduino installation file, you just need to run it by double clicking on it. Follow the instructions in the setup menu.

Installation of the drivers on Windows XP/Vista/7

After installing the Arduino software, the main drivers will be installed on your computer. The software can have problems, however, with detecting the right driver for the connected board. You need to do the following steps:

1. Connect the Arduino UNO to your computer with the USB cable
2. The computer tries to detect and install the correct driver. This does not always work properly. We need to install the driver manually.
3. Open system preferences (Start → System preferences)
4. Go to System and Secruity → System → Device Manager
5. Search for Arduino UNO / Genuino UNO or unknown device. Right-click on it and choose the option to install driver software manually.
6. Search driver software on the computer and choose the folder for the arduino software. In this folder, there is the folder Driver. Choose this folder and confirm your selection.
7. Windows will install the correct driver and your Arduino UNO should work properly.

Installation on macOS

After downloading the files (see above) you can install them and copy them to your applications folder. It is not necessary to install drivers on macOS.

Installation on Linux

After downloading the files (see above) you need to extract the `.tar.xz` archive. You can run the installation script `install.sh` with the following command from the teminal:

```
cd <path to extracted folder>
./install.sh
```

In a desktop environment you can start the installation with a double-click on the `install.sh` file.

Afterwards there will appear a shortcut to run the Arduino IDE.

Arduino IDE

The Arduino IDE is the programming interface we use for our microcontroller. The white editor window represents the area for your programming code. In the black area, status and error messages will be displayed. The Arduino IDE is able to recognize misspellings and errors in your code's syntax, however logic errors can not be recognized.

In the upper toolbar of the software, you can find the most important elements to control the software:

- The checkmark is used to check your program's code if there are any mistakes.
- The arrow is used to upload your sketch to the Arduino.
- You can start a new sketch, open an existing one, or save your current sketch
- By using the magnifying glass, you can open the serial monitor. For further information about the serial monitor, check here

Digital signals

A digital signal has only two possible values, 1 and 0 or high and low. An Arduino sketch is a very simple design that consists of two main components. These contain blocks with statements, which describe the program sequence:

```
void setup(){
  // statements
}
void loop(){
  // statements
}
```

The setup function will be executed only once after the program start. The loop function will be executed continuously. Both functions are absolutely mandatory for the program to be compiled successfully. “Compiling” means translating program code into machine code.

Controlling digital actuators

To control a digital actuator, for example a LED, you need two commands: The first one is placed in the setup function, while the second command belongs in the loop function. In the setup function, the command `pinMode (13, OUTPUT);` is used to declare that an output is connected to port 13. In the loop function, the command `digitalWrite(13, HIGH);` is used to turn on power at port 13. The counterpart of this command is `digitalWrite(13, LOW);` to turn off power at port 13. The final sketch should look like this:

```
void setup() {
  pinMode(13, OUTPUT); // declare which pin the LED is connected to
                       // connected as an output
}

void loop() {
  digitalWrite(13, HIGH); // turn on the LED
}
```

Reading digital sensors

The same ports used to control digital actuators can be used to read digital signals. To process the incoming signals, it is necessary to save them in variables. To save digital signals, a boolean variable is particularly appropriate. The boolean variable only accepts two values: 1 or 0. To read a digital signal, two commands are necessary: The first one is placed in the setup function and is quite similar to the one used to control digital actuators. To declare pin 13 as input, you can use the command `pinMode(13, INPUT)` . The second command is placed in the void loop: `value = digitalRead(13);` . This command saves the incoming values in the variable called “value”. Keep in mind that you have to initialize this variable before using it. If you don’t know how, check the variables chapter. Your final program should look like this:

` ` arduino boolean value = 0; // initialize a new boolean variable

void setup() { pinMode(13,INPUT); } void loop() { value = digitalRead (13); // write the measured value to the variable } ` ` `

To check out the incoming values, you can display the variable in the serial monitor.

Analog signals

Unlike digital signals, analog signals can have lots of values varying between a high and a low level. The exact number of values the Arduino UNO can handle is 1024 (10 bit). The highest level equals 5V and the lowest 0V, meaning that in-between these levels the Arduino can read 1024 different currents. To read these currents you can use the Arduino's six analog in pins (A0 - A5).

Controlling analog actuators

The most important command to control analog actuators is `analogWrite()`. This command applies a specific current to the specified pin. The Arduino doesn't have analog output pins. Instead it can use the digital pins 3, 5, 6, 9, 10 and 11 to specify the current via pulse width modulation. The syntax of this command is `analogWrite(pin, <value>)`. The value can vary between `0` (always off) and `255` (always on). Keep in mind, that you have to declare the used pin as `OUTPUT`.

An example program that writes the value 60 to pin 9, can look like this:

```
void setup() {
  pinMode(9, OUTPUT);
}

void loop() {
  analogWrite(9, 60);
}
```

Reading analog sensors

To read an incoming analog value, you can use the command `analogRead()` just like you would do with digital signals. The received values between 0V and 5V are processed by the built-in digital-analog converter, converting them into `integer`-values between 0 and 1023. The reading takes approximately 0.0001 seconds. This means approximately 10,000 readings/second are possible. All the values can be displayed in the serial monitor.

An example program can look like this:

```
int analogPin = 3; // potentiometer on pin 3
int val = 0;       // variable to save the incoming values

void setup() {
  pinMode(analogPin, INPUT);
  Serial.begin(9600);  // start of the serial monitor
}

void loop() {
  val = analogRead(analogPin); // reading the potentiometer
  Serial.println(val);         // display the measurements
}
```

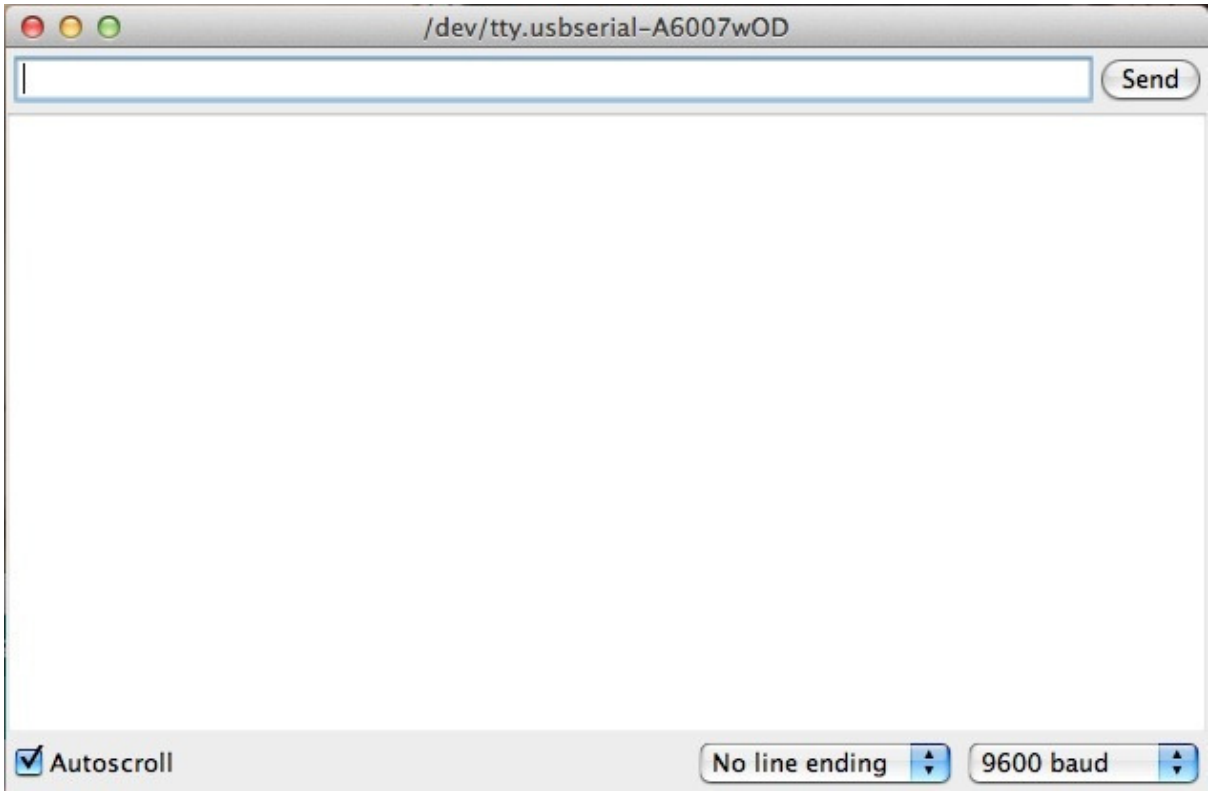
Serial Monitor

The serial monitor allows to transfer data (text or numbers) from and to the Arduino directly from the Arduino IDE. It is an essential tool to check what the Arduino is doing without connecting an LCD to the microcontroller.

Starting the serial monitor

To access the serial monitor, start the Arduino IDE and click on the magnifying glass in the icon bar.

The serial monitor conists of an input field, a button to send and a scrolling text area for the output. The text area shows the messages the microcontroller sends through its serial interface. Ticking the Autoscroll checkbox enables the autoscroll feature which is used to only show the newest messages. New messages will appear at the bottom of the text area.



Displaying Values on the Serial Monitor

To send values from the microcontroller to the PC, first the serial interface has to be enabled through `Serial.begin(9600)` in the `setup()` method. The number 9600 defines the baud rate which is the speed at which the computer and microcontroller communicate. The baud rate defined in the sketch has to be the same in the serial monitor. Otherwise, only scrambled text will be displayed in the serial monitor. The computer and Arduino don't understand each other.

Text or Numbers can be sent over the serial interface using the methods `Serial.print()` and `Serial.println`. The former just submits the data in the parentheses, the latter appends a line break.

You can now try to use the serial interface and serial monitor for yourself. To transmit text you have to put it in quotation marks.

```
Serial.println("senseBox rocks!");
Serial.print("senseBox ");
Serial.println("rocks!");
```

The example should show two lines of the text "senseBox rocks!". Please note the use of `Serial.print()` and `Serial.println()`.

It is also possible to transfer the contents of variables through the serial interface. Just put the variable name in the parentheses of either `Serial.print()` or `Serial.println()`.

```
String helloString = "hello world!";
Serial.println(helloString);
```

if/else Conditions

The `if` expression enables the Arduino to execute different blocks of code by evaluating the given conditional expression. For example, to light up a LED dependent on a switch, you would write the following code:

```
if (digitalRead(BUTTON_PIN) == HIGH) {
  digitalWrite(LED_PIN, HIGH);
}
```

The first line starts with `if` . The following parentheses contain the condition. In this example if the Button is pressed, or not. If the condition appears to be true, the code in the curly brackets is run.

The condition in the parentheses uses a comparision operator. Here it is the double equal sign(`==`). It is important to use this double equal sign for equality comparision. If you use just one equal sign, the Arduino thinks you want to assign the right value to the left variable.

Using else

With `else` you can add an alternative to the `if` block if the condition was not true. Together with the `else` block, the code from above would look like:

```
#define LED_PIN 13
#define BUTTON_PIN 7

void setup() {
  pinMode(LED_PIN, OUTPUT);
  pinMode(BUTTON_PIN, INPUT);
}

void loop() {
  if (digitalRead(BUTTON_PIN)==HIGH){
    digitalWrite(LED_PIN, HIGH);
  } else {
    digitalWrite(LED_PIN, LOW);
  }
}
```

Loops

> Groundhog day

The naive way of writing code to blink a LED 50 times is really tiresome to write:

```
digitalWrite(led, HIGH);
delay(100);
digitalWrite(led, LOW);
delay(100);
digitalWrite(led, HIGH);
delay(100);
digitalWrite(led, LOW);
//...
```

Programmers are lazy, so they invented a really cool solution to this problem: Loops. A loop is a special block which executes the code in the block until a condition is met.

Structure of loops

Loop expressions contain two parts. The head of the loop contains the condition which controls for how often the code in the body of the loop should be repeated. The body usually starts and ends with curly braces (`{ }`).

There are several types of loops each tailored for different needs. The following sections will show you two of the most common loop types.

for loops

`for` loops are usually used for when you exactly want to specify how often the code in the body should be executed. In this example, a LED will be switched on and off 50 times. The head of the `for` loop contains three parts delimited by a semicolon (`;`).

1. Variable initialization. Allows to initialize a variable which holds the count of loops
2. A Condition. The condition allows you to explicitly say when the loop should run. If the expression evaluates truthy, the loop will continue. How to write a condition is explained in the chapter about if-else
3. An expression to run after each loop of the code in the body. Usually it is used to increment the counter variable.

```
for (int counter = 1; counter < 50; counter = counter + 1) {
  // blink LED
}
```

In this example the counter variable is named `counter` . The condition is "while `counter` is below 50 `After each loop iteration, increase` counter ` by one.

> Hint:
>
> - The code *`counter = counter + 1`* is the longer form of *`counter++`* . Both forms do the same. Take the number in the variable *`counter`* , increment by one and store the result in the *`counter`* variable.
> - You can use any name for the counter variable you like. Often the name *`i`* for "index" is used.

Exercise 1

Think of a loop body which prints the current value of the counter variable to the serial monitor.

> Hint: The chapter Serial Monitor explains how to communicate over the serial interface.

- a) Observe the output if you exchange `counter = counter+ 1` with `counter = counter*2` or `counter--`.
- b) Observe the output if you exchange `int counter= 0` with `int counter = 25`.

The while loop

Sometimes you don't know how often the code in the block should be executed. For this, you can use the `while` loop. The while loop only contains a condition and a block. The condition is evaluated before the code in the block runs. If it evaluates truthy, the code in the body is executed.

```
while (condition) {
  // blink LED
}
```

For example, this allows you to evaluate the state of a button attached to the Arduino and only execute the loop if the button is pressed.

> Attention: A common error is to write a condition which is always true. (For example `1 > 0`). In this case, your program will hang indefinitely in the loop and will not continue. This is called an endless loop.

Exercise 2

- a) Write the code for a program that prints `Statement true!` in the serial monitor if a variable `a` is above 0.
- b) Write the code for a program that blinks a LED if a button is pressed.
- c) Every `for` loop can be expressed as a `while` loop. Rewrite this `for` loop so that it uses a `while` loop:

```
for (int i = 10; i > 0; i--) {
  Serial.print("Countdown: ");
  Serial.println(i);
}
```

Using Software Libraries

Similar to the way a shield extends the hardware functionality of your Arduino, software libraries can extend your sketch by useful functions.

In principle such a library can be created by anyone, because it just separates code and functionality into additional files. Usually we use libraries to make communication with specific sensors less complicated, and fortunately most hardware vendors provide such libraries for us.

A lot of libraries are already included with the Arduino IDE, but those for the senseBox sensors have to be installed manually. You should have added these already, if you followed the chapter Software Installation.

Importing additional libraries

When using additional sensors, it is likely that you need to add the associated library manually.

First you have to download the library which is usually a zip file. The Arduino IDE has a function to add external libraries. To do this, click in the menu on Sketch -> Import Library -> Add Library. In the following dialog navigate to the downloaded zip file and select it.

Adding a library to your sketch

To use a library you have to include it in your sketch, after installing the library in the Arduino IDE. To do this, you generally have two options:

Using the menu

In the Arduino IDE menu click on Sketch -> Import Library, and select the desired library in the following list.

When we need additional functionality in our projects, you will be told which libraries to include.

Writing code

When you do this task often, it may be more efficient and convenient to write a single line of code to include a library.

Using the `#include` directive a library can be included. This line has to be at the very top of your sketch, even before the `setup()` function is defined. For example, with the line `#include <Ethernet.h>` the required library for the ethernet shield is included.

To make things more clear, have a look at the following code example:

```
#include <Ethernet.h> // including the library

int Sensor;           // declaration of variables


void setup() {
  Ethernet.begin();   // this function is only available after importing
                      // the Ethernet.h library!
}

void loop() {
  // continuously executed commands, as usual
}
```

Serial Data Busses

The databus is able to communicate with other devices through a data bus. A data bus is a system that allows two or more devices (such as sensors and microcontrollers) to communicate in a well defined manner. This allows an efficient and interoperable way of data transfer between those devices.

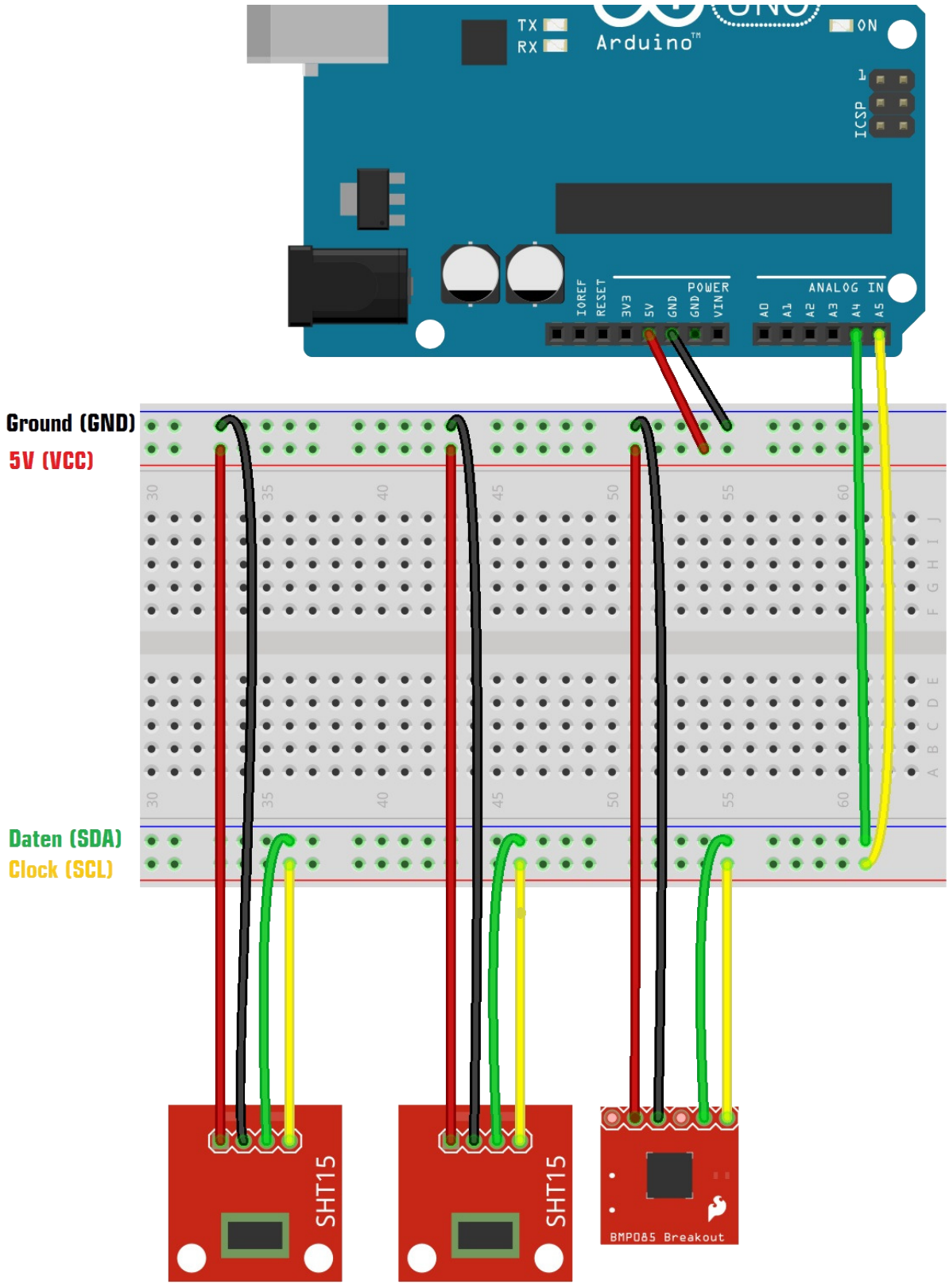
On such a bus the data-bytes are encoded into fast electrical pulses and transmitted using one or more wires. There are many different data buses in use (eg. I²C and OneWire), which differ in the amount of wires, connectable devices, and their bandwith.

The I²C-Protocol

Most of our sensors transfer their data via the I²C bus.

This bus is an easy to use data bus which may connect many devices to a master device. It utilizes two wires to transmit the data; `SDA` (serial data) and `SCL` (serial clock). The SDA wire transmits the actual data bytes, while a timing signal is sent on the SCL wire to keep all devices in sync. Our Arduino Uno has dedicated pins for this bus on `A4` (SDA) and `A5` (SDC).

If we want to connect more than one device to the Arduino, we create interconnect them on a serial circuit: The two wires can be passed on from each slave device to the next.



When using the I²C bus, the Arduino always is the master device, and all other devices slave. Each slave has an hexadecimal address, which is used to uniquely identify the device on the bus when communicating with it. Usually these address is hardcoded in the device, and can be looked up in the device's datasheet.

The `Wire.h` Library

To be able to communicate on the I²C bus, we have to use the `Wire` library. It is included with the Arduino IDE, and has to be included in your sketch with the following line at the top of the sketch:

```
#include <Wire.h>
```

Data is transmitted byte by byte. To send a byte, three commands have to be called:

1. `Wire.beginTransmission(ADDRESS);`

   This command initializes the communication. `ADDRESS` is the corresponding address of the device, eg. `0x76`.

2. `Wire.write(DATA);`

   Send a byte `DATA` to the previously connected device. To send multiple bytes, call this function several times.

3. `Wire.endTransmission();`

   Closes the connection again.

For the transmission in the other direction - ie. recieving data - we use the following 4 commands:

1. `Wire.beginTransmission(ADDRESS);`
2. `Wire.requestFrom(ADDRESS, NUM_BYTES);`

   Request `NUM_BYTES` to be transmitted to us from the device identified by `ADDRESS`.

3. `char incoming = Wire.read();`

   Read one requested byte. Call `NUM_BYTES` times to recieve all data.

4. `Wire.endTransmission();`

Source Code Comments

Did you ever read some code and would not understand what it means? Because programs can be very complicated and abstract, supplementary explanations in the form of code comments are very useful.

Even if you think a piece of code does not need an explanation, you should consider adding a comment to it, in case someone else looks at your code (or even you forgot after a while what you meant to write).

A comment is a line of text in the program code, which is ignored by the compiler, and thus may contain any text. In Arduino you can create two types of comments:

Single-line Comments

These comments are useful to annotate a specific line of code with a short description. They begin with a `//`. Alternatively you can add a comment to the end of a program statement:

```
// I am a comment
int led = 13;  // The value 13 is assigned to the variable led.
```

Additionally these comments are useful, if you want to test a change of your code, without deleting the statements: Just comment the line, and the compiler will ignore the command!

Multi-line Comments

These comments are often times written on top of a program or function. They begin with `/*` and end with `*/`. All text between these two expressions will be ignored by the compiler. This is also useful to 'disable' a whole block of code during development.

```
/*
 *
* I am a multi-line comment.
* I can describe which purpose a program or function has, and who has written it.
*
* by the way:
* <- these stars are created automatically, but not required for the comment..
*
*/
```

How many comments should I add to my program?

This question cannot be answered easily. Some advanced programmers expect to have every line of code commented. This is generally not necessary for the simple Arduino programs we are writing.

In principle at least the following program sections should be commented:

- A comment at the top of the program, describing the programs purpose and usage instructions.
- Each function should be described, especially the type of its parameters.
- Complicated commands like mathematical formulas should be explained in plain language.

Arduino Shields

A shield is a board that can be plugged into the top of an Arduino to extend its functionalities. You can easily plug a shield into the pins of an Arduino to get a compact and modular upgrade.

You can find a shield inside the box of the senseBox:edu created for internet connection. Depending on your version, you will see a red ethernet shield or a blue WiFi shield to connect your senseBox to the internet via ethernet or WLAN.

Additionally you will find a green senseBox shield on which a real-time clock, a microSD card reader and further connections are mounted.

Features of the ethernet shield

Our ethernet shield is a modified version of the official arduino shield. Therefore, the official `Ethernet.h` library will not work. Please use our versions of this library (see Downloads).

> Please note: After installing our library, the Arduino IDE will most likely ask you to update the ethernet library. Please refuse this.

Related topics

- real-time clock (coming soon!)
- SD card logger (coming soon!)
- openSenseMap upload

openSenseMap – connect your senseBox to the internet

By using either the ethernet or WiFi shield of your senseBox (depends on the one you have ordered) you can connect your Arduino to your network. This allows uploading the measured sensor values to the openSenseMap (oSeM). You can get detailed information about the shields here.

Setup

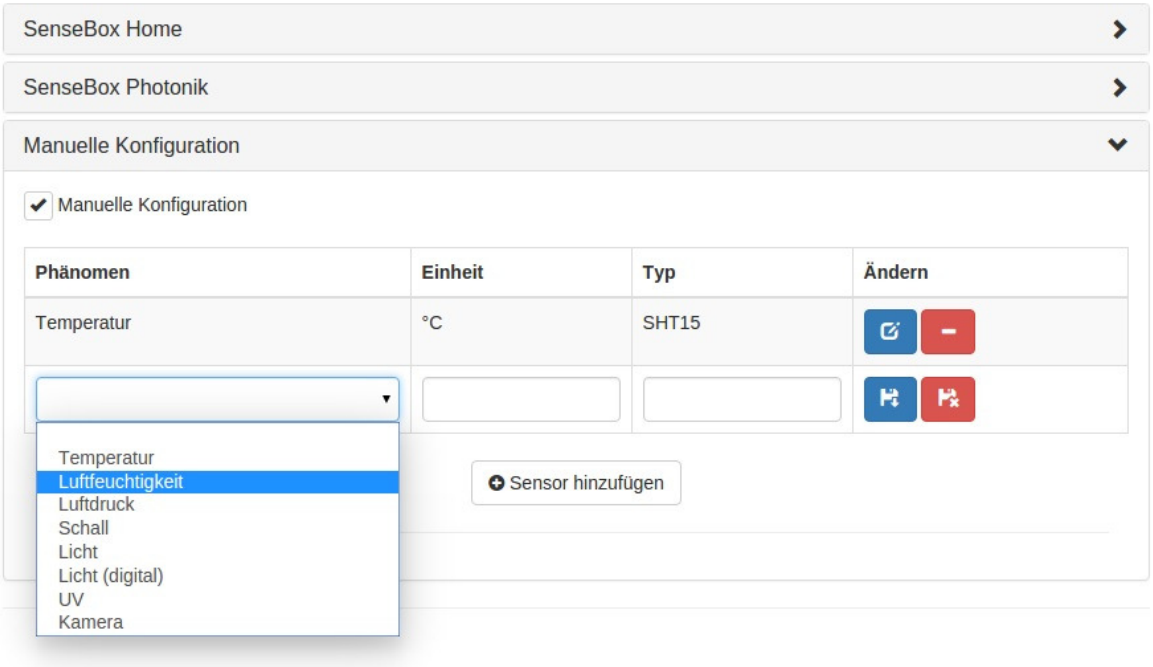
Plug either the ethernet or WiFi shield into the top of the Arduino. If you use the ethernet shield, please connect the shield to your router by using an ethernet cable.

Information: The WiFi shield only supports WiFi network with simple WEP/WPA/WPA2 encryption. Networks that are using certificates are not supported!

openSenseMap registration

Before your senseBox can upload data to the openSenseMap, you have to register a new sensor station. Therefore visit the registration and follow the registration steps.

If all sensors will be connected, you can choose "senseBox:home" under the sensor setup step. If only some or additional sensors will be connected, please choose the "manual setup" and each sensor must be configured manually.



.

After finishing the registration process, you will receive an email that contains an Arduino sketch with basic functionalities to connect your sensor station to the oSeM platform. If you have chosen "manual configuration" during registration, you have to add the code for reading the sensors to your sketch.

Extend the sketch

If you have a senseBox with a WiFi shield, you have to add your network name (`ssid`) and password (`pass`) to the sketch.

You can copy the code for reading the sensors from the previous sections. Normally for each sensor you have to: include a library declare and initialize the sensor (`setup()` function) read the sensor (`loop()` function)

Here is an example for reading the air pressure sensor (BMP280).

Example: BMP280

Add the `BMP280.h` library to the top of your sketch and create an instance `bmp` of it. On this object every function of the BMP280 is called.

```
#include <BMP280.h>
BMP280 bmp;
```

Now the sensor must be initialized inside the `setup()` function.

```
if (!bmp.begin()) Serial.println("BMP init failed!");
bmp.setOversampling(4);
```

After, the sensor must be read inside the `loop` function. Therefore add the following lines of code to your sketch. By using the existing function `postFloatValue()` the measurement is uploaded to the oSeM.

```
double temp, pressure;
char bmpStatus = bmp.startMeasurment();

// if an error occurred on the sensor: stop
if (bmpStatus == 0) {
  Serial.println(bmp.getError());
  return;
}

delay(bmpStatus); // wait for duration of the measurement
bmpStatus = bmp.getTemperatureAndPressure(temp, pressure);

postFloatValue((float)temp, 4, TEMPSENSOR_ID);
postFloatValue((float)pressure, 4, PRESSURESENSOR_ID);
```

Network connection

After you have connected your Arduino to the internet, you can upload the sketch to the Arduino by using the IDE. Inside the Serial Monitor, you can check if the internet connection is working. If the internet connection is working you will see measurements on the openSenseMap!

Connection problems

If the measurements of your station will not show up on the openSenseMap please check the following steps:

- Ethernet shield: Check if the orange LED is blinking. If not please check your cable connection. WiFi shield: Check your WiFi credentials again open the Serial Monitor and check the messages

Traffic counter

Goal

To build a functioning traffic counter. For this purpose we will use an ultrasonic distance sensor. The recorded values will be displayed on the Serial Monitor.

Materials

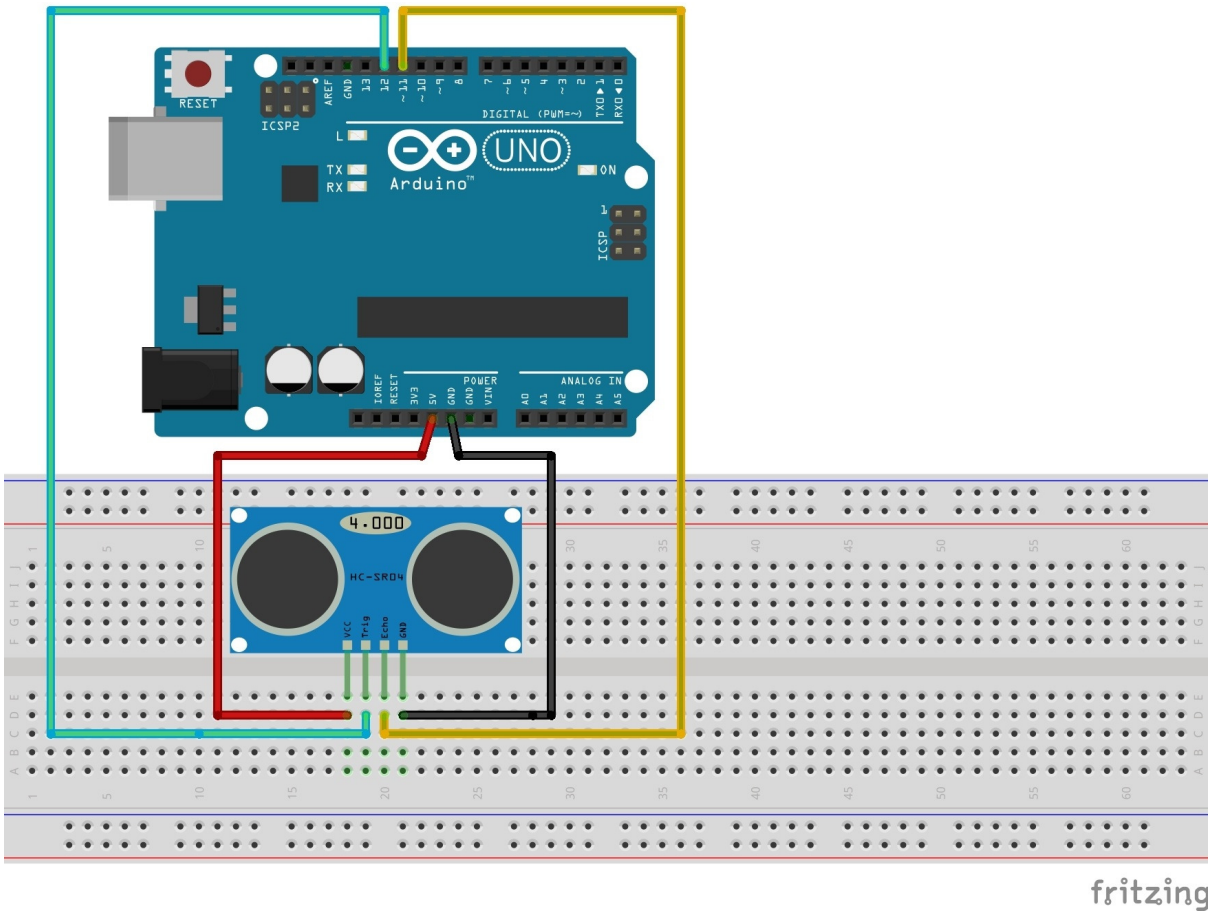
- Arduino Uno with Breadboard
- Ultrasonic distance sensor

Basics

The ultrasonic distance sensor uses sound to calculate the distance between the sensor itself and a given object. The sensor sends out a pulse and measures the time it takes to receive the echo of the pulse. The distance is calculated using the speed of the sound to the object and the measured time.

Construction

The ultrasonic sensor will be connected to four different ports on the Arduino. For the power supply, connect the VCC pin with the 5V port on the Arduino. To close the circuit, connect the GND pin to the GND port of the Arduino. Finally, connect the echo and the trigger pin of the sensor to two different Arduino digital ports (e.g. 12 and 11).



Programming

Define the pins that are connected to the sensor as usual. In addition to that we need to define two variables to save the measured time and the calculated distance.

```
int trig = 12; // Trig pin of the sensor is connected to Pin 12
int echo = 11; // Echo-pin of the sensor is connected to Pin 11
unsigned int time = 0;
unsigned int distance = 0;
```

In `setup()` we will start the Serial Monitor. We can then define which pins will represent the input and output. The sensor's trigger pin must be defined as the output and the echo pin will therefore be defined as the input.

```
Serial.begin(9600);
pinMode(trig, OUTPUT);
pinMode(echo, INPUT);
```

In `loop()` we execute a 10 microsecond long ultrasonic pulse:

```
digitalWrite(trig, HIGH);
delayMicroseconds (10);
digitalWrite(trig, LOW);
```

The subsequent command `time = pulseIn(echo, HIGH);` saves the value for the time it takes to receive the echo into the variable `time`. Finally the distance to the car must be calculated. To do this we use the variable `distance`. The result we than display on the Serial Monitor.

```
distance = time / 58;
Serial.println(distance);
```

Task 1

Try to build a traffic counter.

> Things to consider:
>
> - Try to evaluate only a certain distance range, so that there isn't any interference caused by movements in the background. The sensor measures up to approximately 3 meters.
> - In order to avoid multiple counts of a stationary vehicle, you should program a condition that stops the counting process until the lane is free again.
>
>   For this, you may use a `while` loop.

Traffic light

Goal

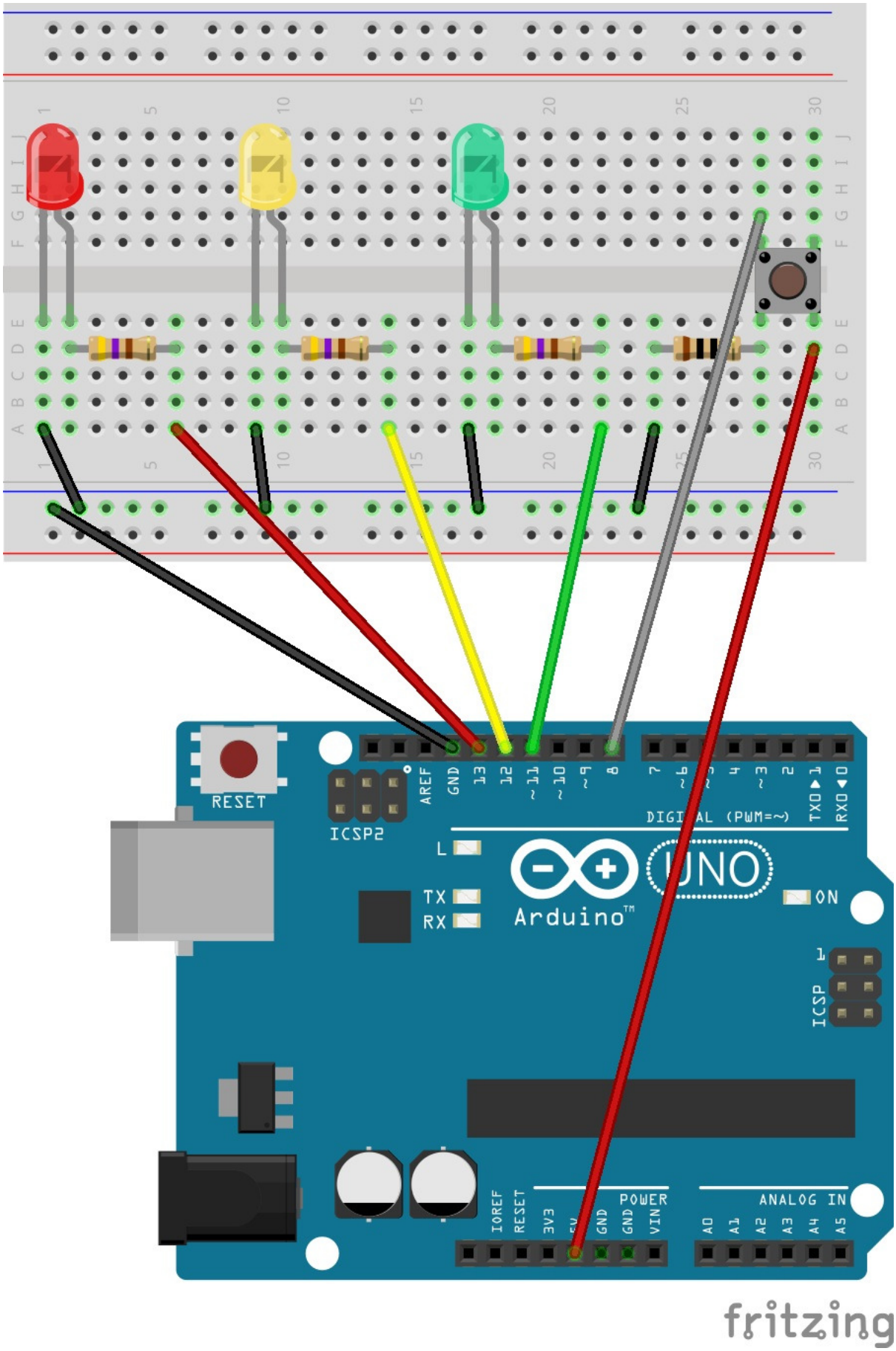
We will simulate a traffic light which can be started using a button.

Materials

- Red LED
- Yellow LED
- Green LED
- 3x 470Ω resistor
- Button
- 10kΩ resistor

Setup Description

Hardware configuration


RESET
ICSP2
AREF
GND
13
12
~11
~10
~9
8
7
~6
~5
4
~3
2
TX→1
RX←0
DIGITAL (PWM=~)
L
TX
RX
Arduino™
UNO
ON
ICSP
POWER
ANALOG IN
IOREF
RESET
3V3
5V
GND
GND
VIN
A0
A1
A2
A3
A4
A5
fritzing

Software Sketch

```
int red = 13;
int yellow = 12;
int green = 11;

int button = 8;

void setup() {
  pinMode(red, OUTPUT);
  pinMode(yellow, OUTPUT);
  pinMode(green, OUTPUT);

  pinMode(button, INPUT);

  // set of traffic lights first to RED
  digitalWrite(red, HIGH);
  digitalWrite(yellow, LOW);
  digitalWrite(green, LOW);
}

void loop() {

  // Check if button is pressed
  if (digitalRead(button) == HIGH) {
    delay(5000);

    // RED to GREEN
    digitalWrite(red, HIGH);
    digitalWrite(yellow, HIGH);
    digitalWrite(green, LOW);

    delay(1000);

    digitalWrite(red, LOW);
    digitalWrite(yellow, LOW);
    digitalWrite(green, HIGH);

    delay(5000);

    // GREEN to RED
    digitalWrite(red, LOW);
    digitalWrite(yellow, HIGH);
    digitalWrite(green, LOW);

    delay (1000);

    digitalWrite(red, HIGH);
    digitalWrite(yellow, LOW);
    digitalWrite(green, LOW);
  }
}
```

- At the beginning of the `loop()` function we check to see if the start button is pressed.
- `digitalRead(button)` reads the current state of the button. If pressed, the function outputs HIGH, otherwise LOW.
- To check whether the button has been pressed, the `digitalRead(button)` must be compared with HIGH. The comparison is made with two equal signs == (comparative operator). A match = is an assignment, such as `int red = 13` .

DIY Eco Station

In this project we will learn how to build a senseBox environmental station. At the end we will be able to measure various environmental phenomena such as temperature, humidity, light and air pressure, and the data will be published on the openSenseMap!

This project is the most extensive, therefore it was divided into several chapters. In each new chapter, an additional module is inserted until a full – senseBox:home like – Environmental Station is built!

You may work through each chapter separately, or extend your program code with each chapter to be able to measure all five phenomena at once.

DIY – Experiments with light

If we watch television, turn on the radio, write a message with our smartphone, or heat up food in the microwave, we are using electromagnetic energy. Today, all people are constantly dependent on this energy. Without it, life as we know it in modern cities would be completely different.

Aim of this lesson

In this lesson we are using a light sensor to detect the illuminance of visible light in lux.

Materials

- Light Sensor `TSL 45315`

Basics

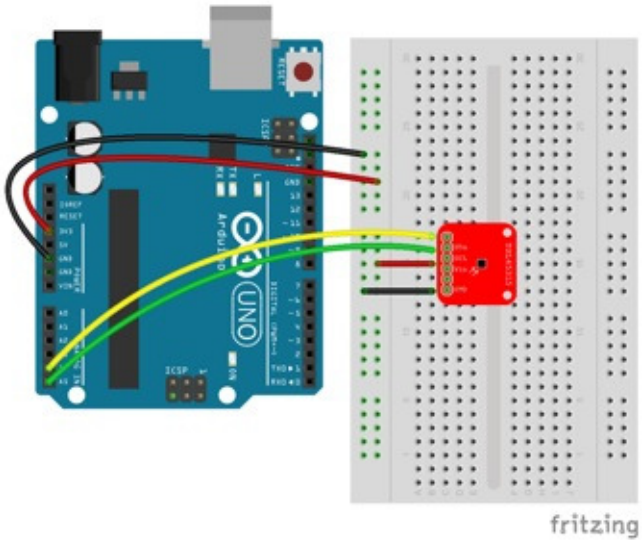
Electromagnetic energy moves in waves through space. These waves range from very long radio waves to very short gamma rays. The human eye can only perceive a very small part of this spectrum; visible light. Our sun is the source of energy over the entire spectrum. The Earth's atmosphere protects us from exposure to high levels of radiation that could be dangerous for us.

In this lesson, the intensity of visible light is of main interest. To measure the intensity of the incident light in the visible part of the spectrum, the unit lux is used. It indicates the ratio of the brightness in lumens per square meter. On a bright sunny day, the light is measured at about 100,000 lux, and for a full moon at night the illuminance is measured at only about 1 lux.

For this measurement we use the I^2C sensor TSL45315 of AMS-TAOS. In the sensor datasheet you can see that its sensitivity is matched to the visible part of the light spectrum, which is approximately between 400 and 700 nm.

According to the data sheet, the sensitivity of this sensor reaches from 2 to 200000 Lux, with a resolution of 3 lux. Furthermore, the sensor needs a power supply of 3.3V.

Construction



Program

We will use the library `Wire` for communication via I²C. In the beginning we need a few constants that are defined with the directive `#define` . Unlike variables, they occupy a permanent place in memory. In our case, the bus address and the following register addresses of the sensor are to be saved.

ADDRESS	RESISTER NAME	R/W	REGISTER FUNCTION	RESET VALUE
--	COMMAND	W	Specifies register address	0x00
0x00	CONTROL	R/W	Power on/off and single cycle	0x00
0x01	CONFIG	R/W	Powersave Enable / Integration Time	0x00
0x04	DATALOW	R	ALS Data LOW Register	0x00
0x05	DATAHIGH	R	ALS Data HIGH Register	0x00
0x0A	ID	R	Device ID	ID

These registers are used for the sensors configuration and its I²C address:

```
#include <Wire.h>
#define I2C_ADDR    (0x29)
#define REG_CONTROL  0x00
#define REG_CONFIG   0x01
#define REG_DATALOW  0x04
#define REG_DATAHIGH 0x05
#define REG_ID       0x0A
```

In the `setup()` function we will connect to the sensor and send the configuration bytes to it:

```
Wire.begin();
Wire.beginTransmission(I2C_ADDR);
Wire.write(0x80 | REG_CONTROL);
Wire.write(0x03); // Power on
Wire.endTransmission();
```

Next, we will set a fixed exposure time of 400 ms:

```
Wire.beginTransmisson(I2C_ADDR);
Wire.write(0x80 | REG_CONFIG);
Wire.write(0x00); // 400 ms
Wire.endTransmission();
```

To change the shutter speed, you can change the corresponding value of `0x00` in `0x01` or `0x02` to reduce the exposure time to 200 or 100 ms in the configuration register of the sensor. In the `loop()` function, we start the measurement routine and request the raw bytes containing the measurement from the sensor:

```
Wire.beginTransmisson(I2C_ADDR);
Wire.write(0x80 | REG_DATALOW);
Wire.endTransmission();
Wire.requestFrom(I2C_ADDR, 2); // Request 2 bytes
uint16_t low = Wire.read();
uint16_t high = Wire.read();
```

If the sensor sends more data than requested, these bytes should be caught afterwards to avoid errors in the next loop.

```
while (Wire.available()) {
  Wire.read();
}
```

Finally, we transform the data to calculate illuminance in lux. In the sensor's datasheet, we can find the matching formula:

```
uint32_t lux;
lux = (high << 8) | (Low << 0);
lux *= 1; // Multiplier for 400ms
```

To adjust this formula to an exposure time of 200 or 100 ms, you can increase the multiplier to 2 or 4.

Exercise 1

Adapt this lesson's code to create a custom function that can print the sensor data to the Serial Monitor.

Exercise 2

Change the exposure time of the sensor and then compare the results of the measurements.

> Tip: Don't forget to adjust both the lux value and exposure time in the configuration register accordingly.

DIY - UV-Light Sensor

Solar radiation in the ultraviolet (UV) spectrum can get hazardous to the skin very quickly. Official weather stations rarely measure the UV-intensity, so we want to measure the UV-light ourselves!

Aim of this lesson

In this lesson we are using an UV-light sensor to detect the amount of solar radiation in the UV-spectrum as power per area ($\mu W / cm^2$).

Additionally, we convert the measured value to the standardized UV-Index which is easier to comprehend.

Materials

- UV-Light Sensor `VEML6070`

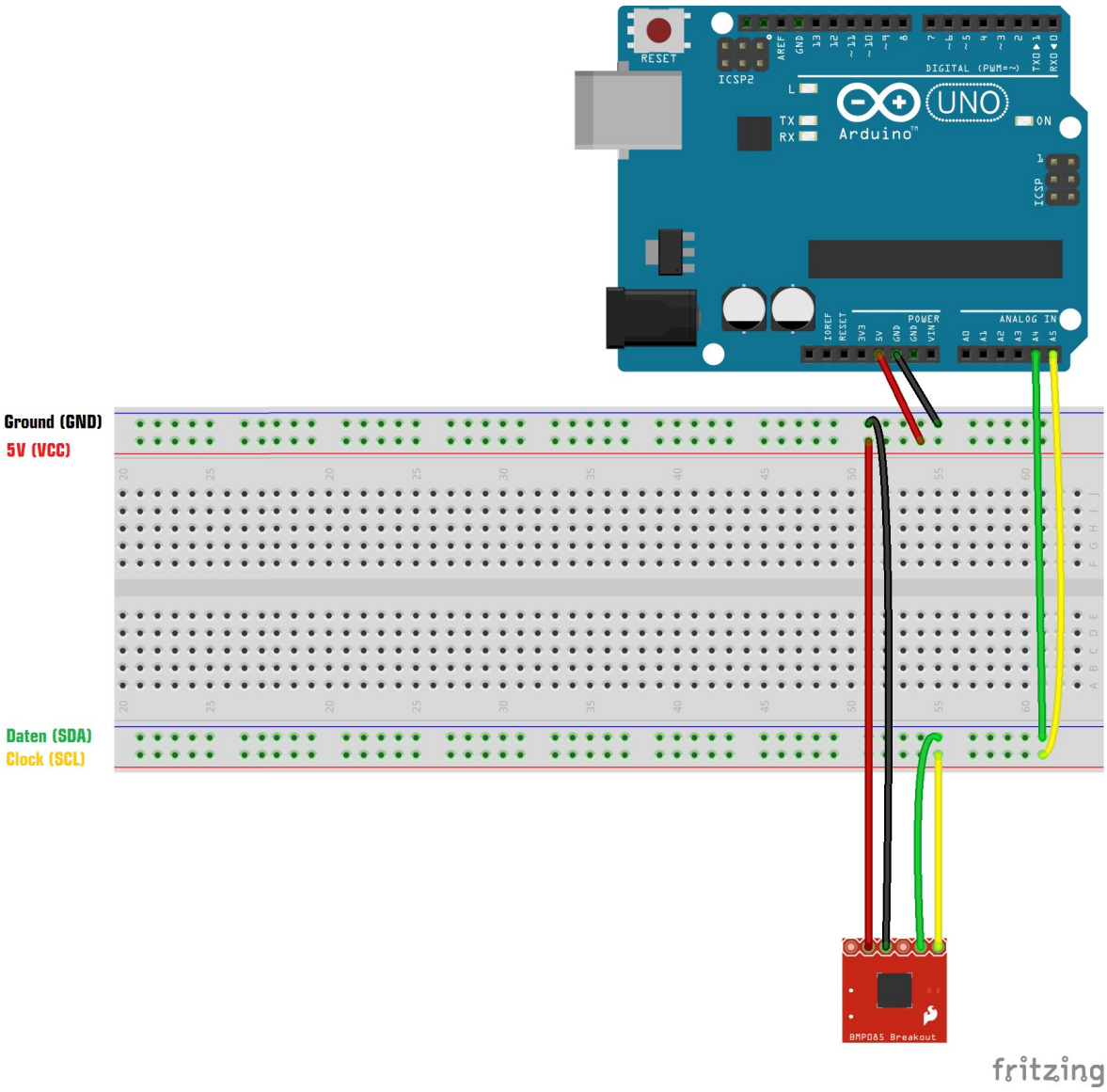
Basics

Ultraviolet "light" is a kind of radiation that is invisible to the human eye. It has a shorter wavelength than visible light, but longer than that of X-rays: The spectrum is defined from 100 nm to 380 nm.

Through absorption of the earth's atmosphere in the ozone layer, the solar radiation in the UV-B spectrum (100 - 300 nm) does barely reach the earth's surface. The less dangerous UV-A radiation (300 - 380 nm) is far less absorbed by the atmosphere.

UV-light intensity is measured in micro-watts per square-centimeter ($\mu W / cm^2$). Our `VEML6070` sensor measures radiation from round about 300 - 400 nm, so it can only detect UV-A radiation (for more precise information, consider the datasheet).

Construction



Connect the VEML6070 sensor to the Arduino as shown in the graphic above.

Program

To communicate with the sensor via the I²C bus, we need to import the `Wire.h` library. We also need some constants that define the sensor's I²C address and some configuration of the sensor. Additionally we define a reference value for the conversion of the measurement to a UV-index.

```
#include <Wire.h>

#define I2C_ADDR_UV 0x38
// integration times
#define IT_0_5 0x0 // 0.5 T
#define IT_1   0x1 // 1 T
#define IT_2   0x2 // 2 T
#define IT_4   0x3 // 4 T

// reference value: 0.01 W/m^2 corresponds to the UV-index 0.4
float refVal = 0.4;
```

Now we configure our sensor in the `setup()` function.

```
void setup() {
  Serial.begin(9600);

  Wire.begin();
  Wire.beginTransmission(I2C_ADDR_UV);
  Wire.write((IT_1<<2) | 0x02);
  Wire.endTransmission();
  delay(500);
}
```

In the `loop()` function we define the behaviour of our main program to read out the sensor at an interval:

```
void loop() {
  byte msb=0, lsb=0; // first and second byte that will be read from the sensor
  uint16_t uv;

  Wire.requestFrom(I2C_ADDR_UV+1, 1); // MSB (read first byte from sensor)
  delay(1);
  if(Wire.available()) {
    msb = Wire.read();
  }

  Wire.requestFrom(I2C_ADDR_UV+0, 1); // LSB (read second byte from sensor)
  delay(1);
  if(Wire.available()) {
    lsb = Wire.read();
  }

  uv = (msb<<8) | lsb; // combine bytes to an integer through a bitshift op

  Serial.print("µW per cm²: ");
  Serial.println(uv, DEC);     // log value as 16bit integer
  Serial.print("UV-Index: ");
  Serial.println(getUVI(uv));

  delay(1000);
}
```

> Attention: If you compile the program before defining the function *`getUVI()`* (see below), you will recieve a error message.

Because we can hardly relate power-per-area measurements in everyday life, we want to convert our measurements to the more widely used UV-index. To do this, we implement a function `getUVI()` :

```
/*
 * getUVI()
 * expects the measurement value from the UV-sensor as input
 * and returns the corresponding value on the UV-index
 */
float getUVI(int uv) {
  float uvi = refVal * (uv * 5.625) / 1000;
  return uvi;
}
```

DIY - Airtemperature & Humidity

To provide the daily weather forecast in TV, web and newspapers, not only satellite imagery is analyzed, but also data from measuring stations on the ground provide important data points. But how does the measurement and visualization of temperature and humidity values work exactly?

Aim of this lesson

In this lesson we measure and display air-temperature and -humidity using the HDC1008 sensor.

Materials

- combined temperature and humidity sensor `HDC1008`

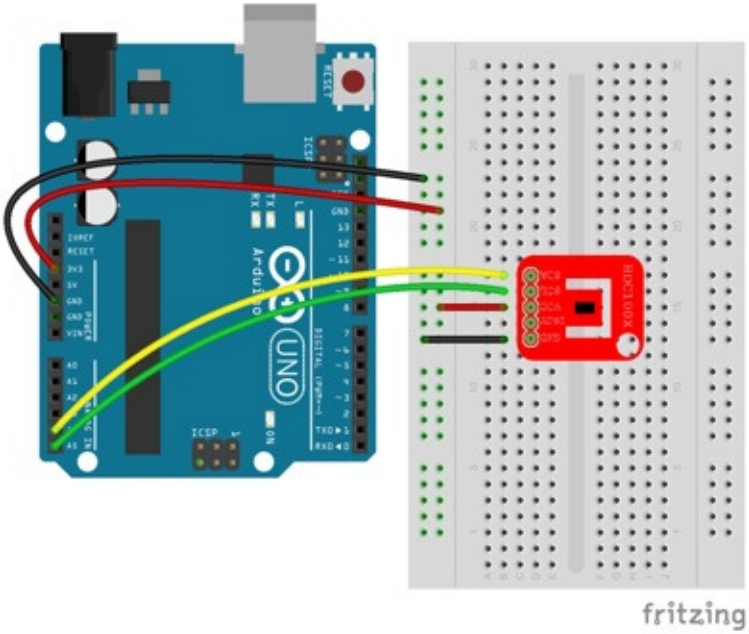
Basics

The `HDC1008` from Texas Instruments series HDX100X combines two sensors on one breakout board: It can measure relative air-humidity from 0% - 100% and temperature from -40°C to 125°C with an accuracy of ±4% and ±0.2°C respectively.

Communication between Arduino and HDC1008 is handled on the serial bus I²C. Unlike simple analog or digital signals, with I²C multiple devices (like sensors or displays) may communicate on the same cables through a parallel circuitry. Each device has therefore a unique identifier that is used to separately address it in communications.

Construction

Connect the devices as shown in the graphic below.



Program

Before starting to write code, we need to install the `HDC100X.h` library. If you have not done this in one of the previous chapters, you should follow the chapter Software Installation.

This library allows us to conveniently speak to the sensor, without manually defining all the configuration registers. To be able to use this library, it has to be included at the top of the program with an `#include` directive. In this case we additionally require the `Wire.h` library to use the I²C bus:

```
#include <Wire.h>
#include <HDC100X.h>
```

> Hint: Unlike expressions, a directive (beginning with `#`) does not end with a semicolon.

The rest of the program may now use the included functions from these libraries.

In order to connect to the sensor, the HDC100X library needs to know the I²C address of the sensor. The HDC1008 listens to the address `0x43` (see datasheet).

```
HDC100X hdc(0x43);
```

With this command we created an instance `hdc` of the sensor, on which we can call functions to talk to the sensor.

In the `setup()` function we initialize the sensor, telling it to measure both temperature and humidity, each with a resolution of 14 bit, and to disable the sensors integrated heater:

```
hdc.begin(HDC100X_TEMP_HUMI, HDC100X_14BIT, DISABLE);
```

Now we may request measurements of the sensor in the `loop()` function using the following two commands:

```
hdc.getHumi();
hdc.getTemp();
```

> Hint: These functions return the measured value as `float`. Be sure that the variables you want to store the measurements in has the correct data type!

Exercise 1

Build the circuit thats depicted above and try to read measurements from the HDC1008 sensor. Print the measured values to the serial monitor.

Hint: If you're stuck, have a look at the example thats included in the folder of the `HDC100X.h` library.

DIY - Air Pressure Sensor

Required Reading

- serial data bus I²C

Aim of this lesson

This lesson shows how to read measurements from the BMP280 air pressure sensor.

Material

- `BMP280` air pressure & -temperature sensor

Basics

BMP280 Sensor
The BMP280 is capable of measuring air pressure in hectopascal (hPa) as well as air temperature (°C). This sensor communicates via the I²C protocol, and requires an operating voltage of 3.3 - 5.0 volts.

The I²C address of the BMP280 can be changed through it's `SDO` pin: If `SDO` is connected to `GND`, the address will be `0x76`, otherwise `0x77`.

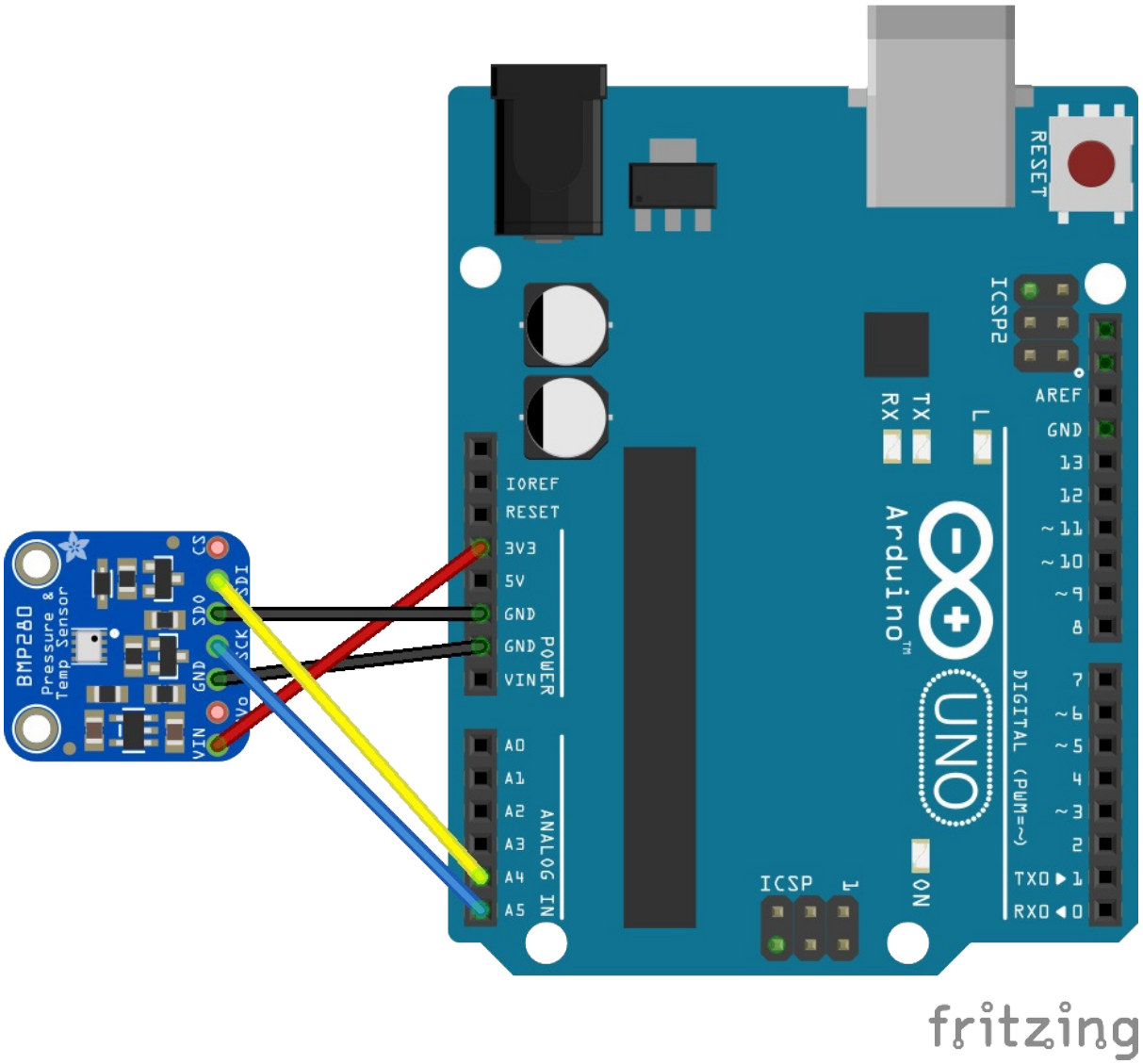
Inferring height from air pressure
Air pressure varies with the height above sea level: The higher you are, the lower the air pressure will get. This means, that we can deduce our height above sea level from a pressure measurement!

To do so, we need a reference value of the current pressure at sea level, usually called `P0`. Because this value constantly changes through the weather, and depends roughly on the current location, we have to recalibrate this value regularly for accurate measurements.

Construction

The operating voltage is provided to the sensor through the wires `3.3V -> VCC` and `GND -> GND`. The I²C bus is set up as usual via the `SDA` and `SCL` pins.

Additionally this sensor requires a wire from `GND` to `SDO` on the BMP280. This changes the sensors address from `0x77` to `0x76`, which is the configured value in the `BMP280.h` library.



Program

The sensor can be controlled with the `BMP280.h` library. Once this file is included in the sketch, we can create an instance `bmp` of it. Now we are able to call the libraries functions on this object:

```
#include <BMP280.h>
#include <Wire.h>
BMP280 bmp;
```

In the `setup()` function we initialize the sensor with the following lines:

```
if (!bmp.begin()) {
  Serial.println("BMP init failed!");
}
bmp.setOversampling(4); // select resolution of the measurements
```

Now we need to can issue a measurement in the `loop()` function. The variables `temp` and `pressure` will then contain the current measurement values:

```
double temp, pressure;
```

```
char bmpStatus = bmp.startMeasurment();

// if an error occured on the sensor: stop
if (bmpStatus == 0) {
  Serial.println(bmp.getError());
  return;
}

delay(bmpStatus); // wait for duration of the measurement
bmpStatus = bmp.getTemperatureAndPressure(temp, pressure);
```

Exercises

Exercise 1
Connect the `BMP280` sensor with the arduino, and create a sketch that prints the current air pressure and temperature on the serial monitor.

Exercise 2
Consider the code above which reads from the `BMP280` . What is the variable `bmpStatus` used for?

Exercise 3
You learned that you can derive the height of your senseBox from the current air pressure. Use the function `bmp.altitude(...)` to calculate the height, and print it on the serial monitor as well.

> Hint: *Have a look at the example file in the BMP280 library. For correct values, the reference value `P0` needs to be adapted to the current weather: You can find your local value here.

The first sound - using a buzzer

Until now, our senseBox has been silent. In this project, we will change this!

Aim of the Project

For the first step, we simply want to get a sound from the buzzer. Next we want to change the volume of the buzzer. Finally, the buzzer should play a simple melody. While the first two steps can be quickly achieved, the third step is a little trickier.

Materials

- buzzer
- Potentiometer

Basics

A buzzer, or a Piezo, is a component that converts electrical signals into sound. The volume can reach up to 80dB. The buzzer has two pins where it can attach to the plug-in board. The operating voltage of the buzzer is between 1V and 12V, and it consumes up to 19mA. Similar to the LED, the buzzer's electricity is only able to flow in one direction. The shorter pin must be connected to the grounding (GND) source and the longer pin must be connected to the voltage source.

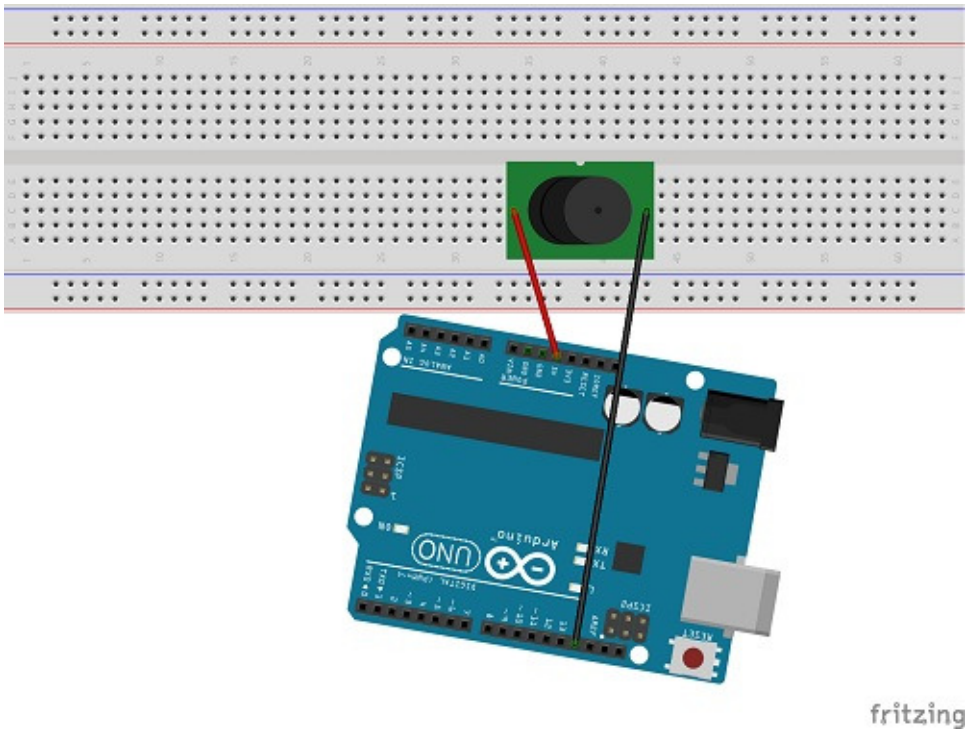
The Potentiometer is an electrical component whose resistance value can be continuously adjusted. Its settings can be adjusted by moving a grinder over the resistor body. Usually a potentiometer has three pins: two for the resistance and a third for the tap. Our potentiometer has a maximum resistance of 10k ohms.

Warning: Small potentiometers are designed only for a small current flow. For the electrical components in senseBox, this potentiometer is sufficient. If you connect components with greater power consumption (for example, a servo motor) however, you need a larger potentiometer.

Construction

Step 1:

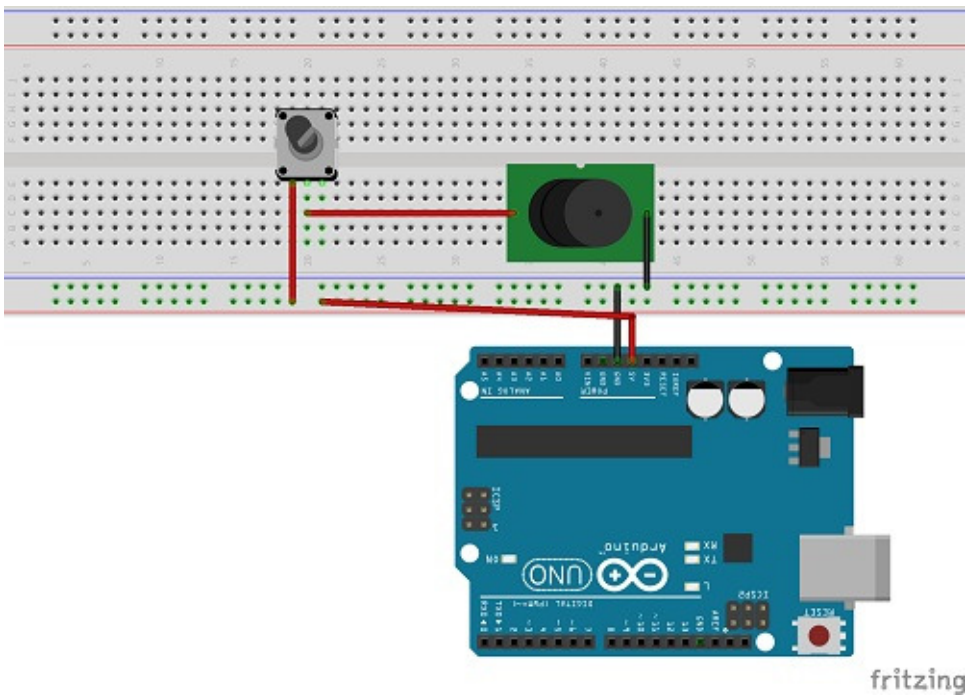
If you build the circuit as shown in the graph and connect the Arduino to the power supply, the buzzer should produce a loud sound. By doing this we have already completed our first step.



Step 2:

Now we'd like to integrate another component into our circuit that will enable us to change the volume. We will use a potentiometer to control the buzzer's volume, similar to the volume control knob that is seen on older radios.

Next, connect the `5V` output of the Arduino and the long pin of the buzzer to the potentiometer. Now you are ready to change the volume using the potentiometer!



Step 3:

A single continuous sound is not really exciting - our buzzer is capable of more! To produce different tones, we can use special outputs from the Arduino that are able to output pulse-width modulation. For more information on pulse-width modulation (PWM), look here.

These pins are marked with the sign ~: The included pins are 4, 5, 6, 9, 10 and 11. A buzzer produces a specific sound for each pulse width. Every tone on the scale (scale: c, d, e, f, g, a, h, c) receives a pulse width. We will use the construct `#define` as follows:

```
#define  h     4064    // 246 Hz
#define  c     3830    // 261 Hz
#define  d     3400    // 294 Hz
#define  e     3038    // 329 Hz
#define  f     2864    // 349 Hz
#define  g     2550    // 392 Hz
#define  a     2272    // 440 Hz
#define  h     2028    // 493 Hz
#define  C     1912    // 523 Hz
#define  E     1518    // 659 Hz
#define  F     1432    // 698 Hz
#define  R     0       // define a note as substitute for a pause
```

Now we need some variables to control the playback behavior of the Arduino. Later you can try different values and check how this affects the melody:

```
// Set overall tempo
long tempo = 26000;
// Set length of pause between notes
int pause = 1000;
// Loop variable to increase Rest length
int rest_count = 50;
```

Now we need some global variables that are used by the playback functions, and we will define the `setup` :

```
// Initialize core variable
int tone = 0;
int beat = 0;
long duration = 0;
int speakerOut = 9;

void setup() {
  pinMode(speakerOut, OUTPUT);
}
```

Now we can write our melody using an array. Another array `beats` is defined, representing how long the corresponding note should be played in `melody` :

```
int melody[] = { g, e, R, R, R, e, f, g, E,  E,  C }; //example melody
int beats[]  = { 8, 8, 8, 8, 8, 8, 8, 8, 16, 16, 32 };
```

Later you can insert your own melody here.

We will next write a help method, which plays one tone of our melody. For this purpose, it will check the first `if` statement and evaluate whether it is a tone or a pause. If it is a tone, the tone will be played in a loop for a certain `duration` measured in milliseconds:

```
void playTone() {
  long elapsed_time = 0;
  if (tone > 0) { // if this isn't a Rest beat
    while (elapsed_time < duration) {
```

```
      digitalWrite(speakerOut,HIGH);
      delayMicroseconds(tone / 2);

      // DOWN
      digitalWrite(speakerOut, LOW);
      delayMicroseconds(tone / 2);

      // Keep track of how long we pulsed
      elapsed_time += (tone);
    }
  }
  else { // Rest beat;
    for (int j = 0; j < rest_count; j++) {
      delayMicroseconds(duration/2);
    }
  }
}
```

After our sound has been played, it's time to play an entire melody with another helper method. We will now implement a method for playing the whole melody. For this is a `for` loop is written to go through the `melody` array, and to retrieve the helper function `playTone()` for each entry, which we have defined above. In addition to each tone, a short pause, or delay, is inserted.

```
int MAX_COUNT = sizeof(melody) / 2; // number of tones

void playMelody(){

  for (int i=0; i<MAX_COUNT; i++) {
    tone = melody[i];
    beat = beats[i];

    duration = beat * tempo; // Set up timing

    playTone();

    delayMicroseconds(pause);
  }
}
```

Now we must include the main loop, which controls the flow of the program:

```
void loop() {
  playMelody();
}
```

Idea: If you would like to include higher or lower notes in your tune, you can define them similar to what we have done in the example above. [here] (http://www.phy.mtu.edu/~suits/notefreqs.html) you can check how much Hertz a tone has.

Warning: All variables defined in the program must have a unique name!

Listening for sounds

Goal

In this station, we will learn how to use the microphone.

Materials

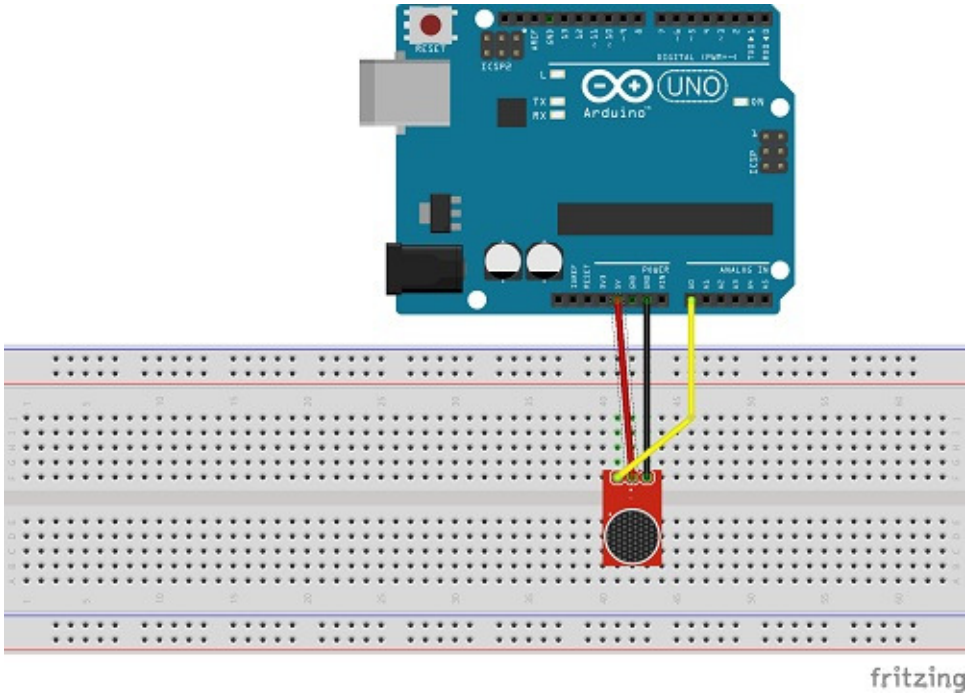
- Mic Breakout

Basics

The microphone is fit with an amplifier (100x) on the board. It requires an operating voltage between 2.7V and 5.5V, and is able to perceive sounds between 58dB and 110dB.

Construction

Consider the graphic below, and build the circuit as shown.



> Note: If your analog Pin `A0` is already occupied, you can use a different pin. Do not forget to change this in the code.

Task 1

Define the pin where your microphone's output rests, as usual. A variable must be defined representing where the values of the microphone are stored:

```
int mic = A0;
```

```
long micVal = 0; // Stores the value of the sound collected by the microphone
```

Now the serial output must be initialized and the pin `mic` must be defined as `INPUT`. We will do this in `setup`:

```
void setup() {
  Serial.begin(9600);
  pinMode(mic, INPUT);
}
```

We then write a function to read the sound value collected by the microphone:

```
long getMicVal() {
  micVal = analogRead(mic);
  return micVal;
}
```

Now you can show the value from the microphone in the Serial Monitor.

```
void loop() {
  Serial.print(getMicVal());
}
```

You will notice that the output varies around the value of 510. Negative values may be returned. To improve the readability of the values, we can change them in the function `getMicVal()`:

```
long getMicVal() {
  int period = 3; // Averaging three values in order to catch any 'outliers'
  int correction_value = 510;
  for (int i = 0; i < period; i++) {
    // Computes the absolute value of the value to intercept negative deflections
    micVal = micVal + abs (analogRead(mic) - correction_value);
    delay(5);
  }
  micVal = constrain(abs(micVal / period), 1, 500);
  return (micVal);
}
```

Now you can test out what happens when the sensor is exposed to certain sounds:

- How strong is the amplitude of conversations?
- What happens if you hold the beeper to the microphone?
- And what happens if you breath into the microphone?

Task 2

Construct a noise operated traffic light using three LEDs. The light should turn green when it is quiet, it should turn orange when there's low volume, and finally it should turn red when it is noisy.

Community Projects

In this section community-contributed projects are documented.

You may also have a look on german project tutorials in the german section of this book.

If you want to contribute a project documentation yourself, have a look on our contribution guide.

Project Title	Author	Date	additional material required
Mobile Sensor Logger	Michelle Gybel & Johannes Schöning (Hasselt University)	07.07.2016	no
Heatmap	Michelle Gybel & Johannes Schöning (Hasselt University)	08.08.2016	no

DIGITAL CITIZENS, CONNECTED COMMUNITIES:

THE ROLE OF SPATIAL COMPUTING AND VOLUNTEERED GEOGRAPHIC INFORMATION

Tutorial: Mobile Sensor Logger

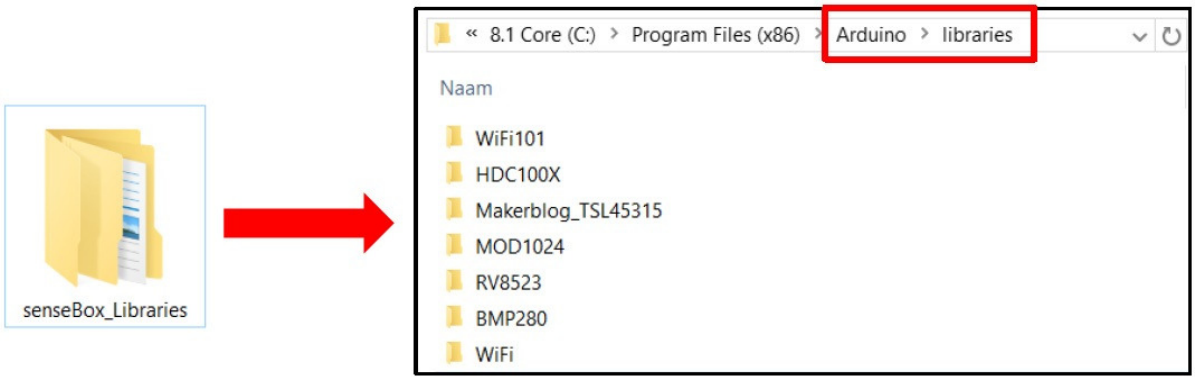
Authors: Michelle Gybel and Johannes Schöning of Hasselt University

Edited by Felix Erdmann using pandoc

This tutorial is a step-by-step guide which will teach you how to build mobile sensor logging device using the senseBox Edu kit.

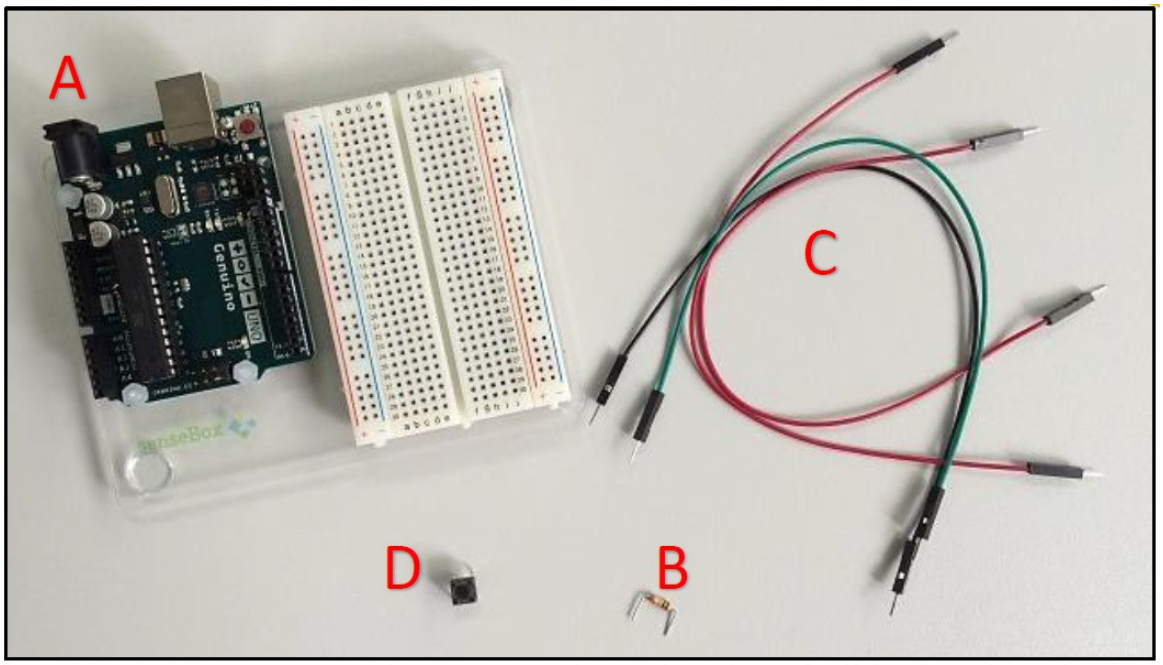
Preparation

1. Install Arduino
    - Download from here
    - (Extra:) Tutorials and Reference Guides can be found here
2. Install senseBox Plugins
    - Download from here
    - Extract the folder and copy the content to the `libraries/` folder in Arduino's installation files.

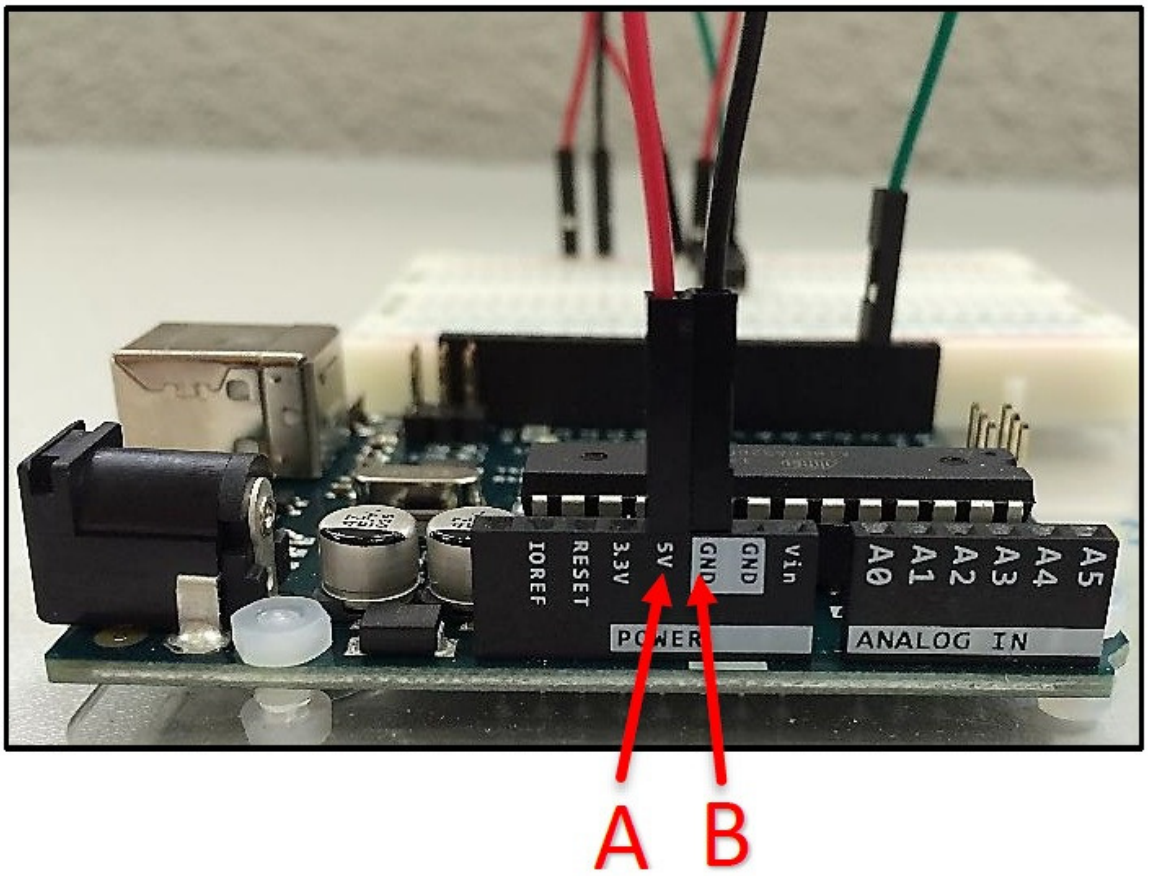


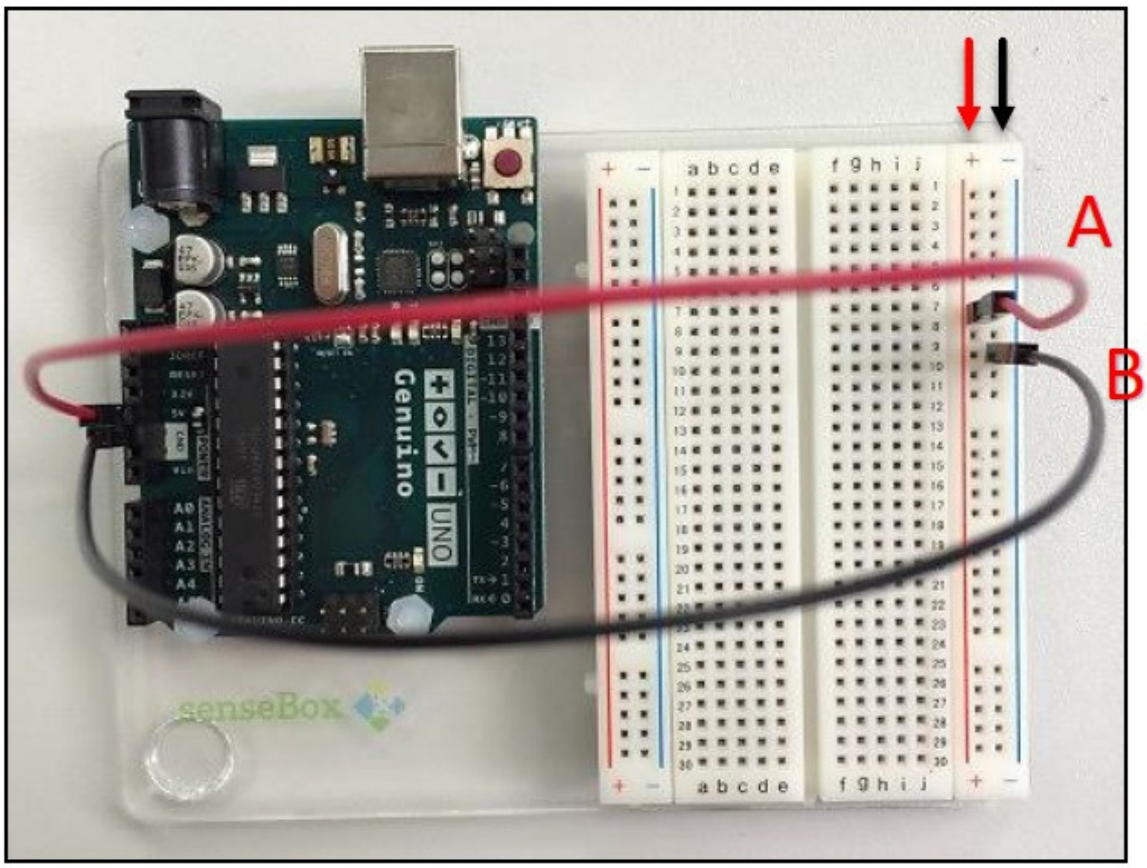
Start with the basics... How to Push a button!

1. What do you need?
    - Genuino Board with breadboard (A)
    - 10K ohm resistor (B)
    - Wires (C)
    - Pushbutton (switch) (D)



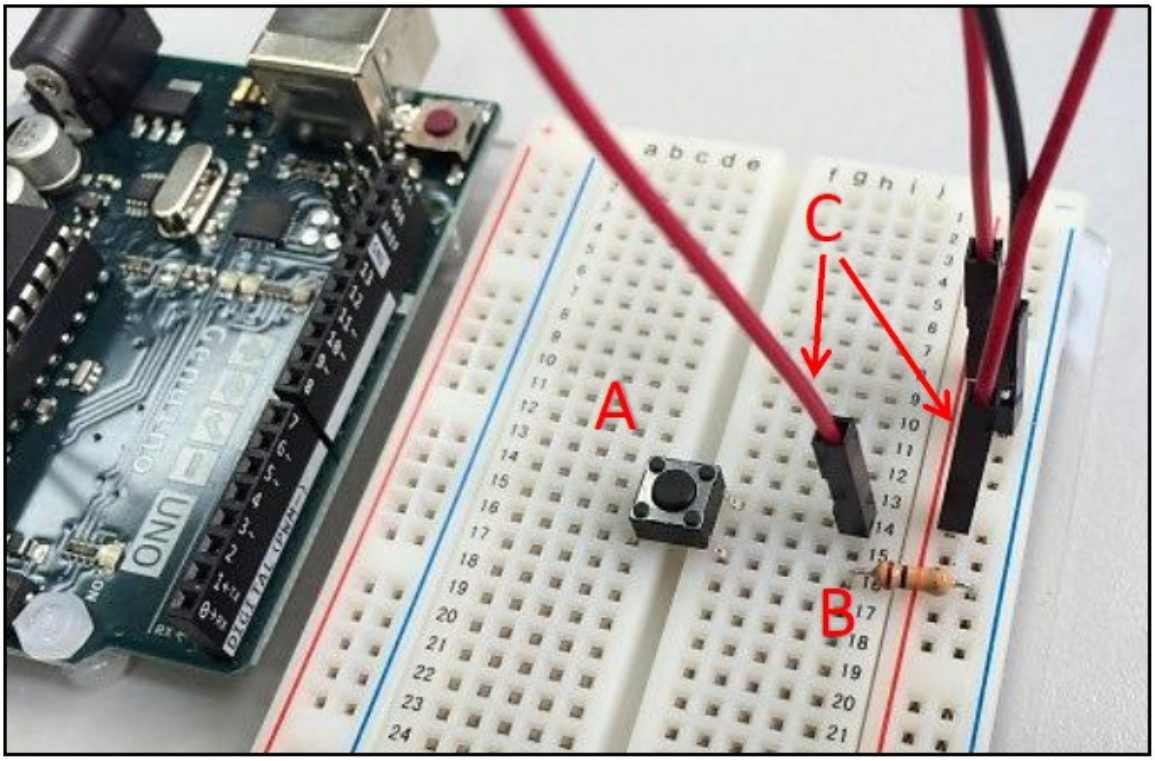
2. Provide voltage and grounding to the breadboard
    - Connect a red wire to the positive vertical row of the breadboard and to the 5 volt supply on the Genuino Board (A).
    - Connect a black wire to the negative vertical row of the breadboard and to the ground (GND) on the Genuino Board (B).



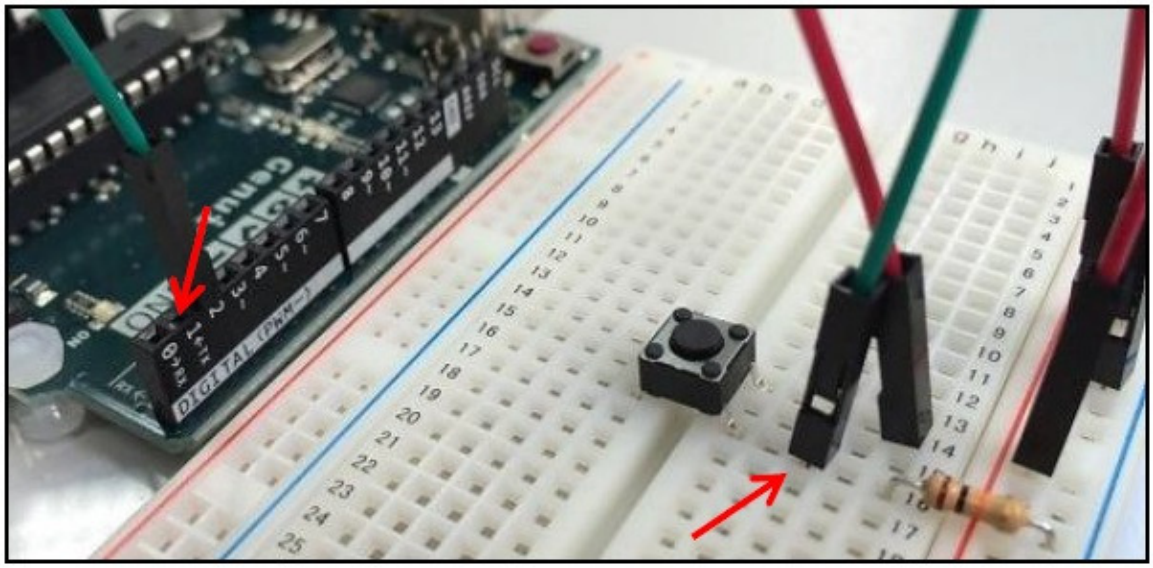


3. Add the pushbutton and provide it with power and grounding

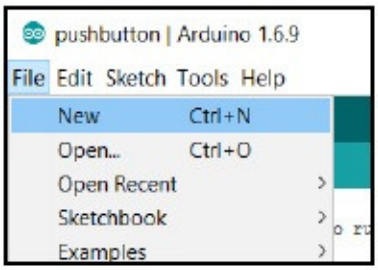
- Attach the pushbutton to the breadboard as presented below (A).
- Create a connection between one of the legs of the pushbutton and the ground with the 10K ohm resistor (B).
- Connect a red wire to the positive vertical row of the breadboard (C).



- Provide with a green wire a connection between digital pin 2 on the Genuino Board and the leg of pushbutton connected to the ground.

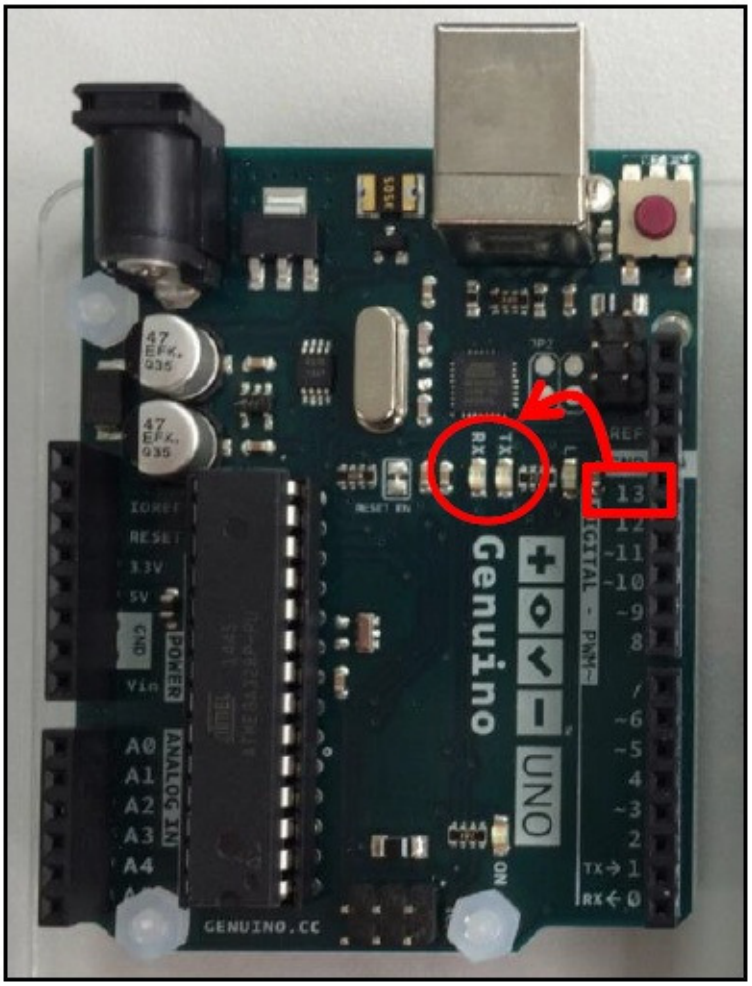


1. Turn on the light, ähm button... by writing some code
    - Start the Arduino IDE.
    - Programs written in the IDE are called sketches. Go to File New and name the file Pushbutton.ino or similar to create your first sketch.



- Each sketch must contain a setup() and a loop() function:

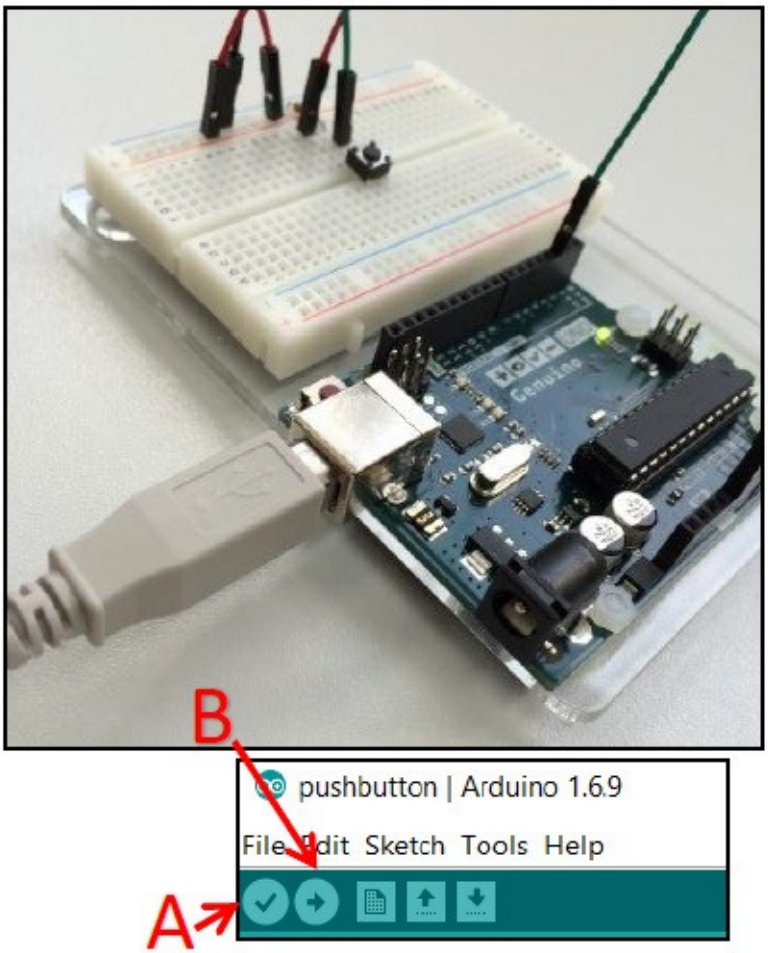
- As a start you can may place the following code in your sketch. It initializes the which pins are connected to the LED light on the Genuino Board and the push button. Next, it provides the functionality to turn the LED light on when the button is pressed in. This LED is built-in in the Genuino board and driven by pin 13.



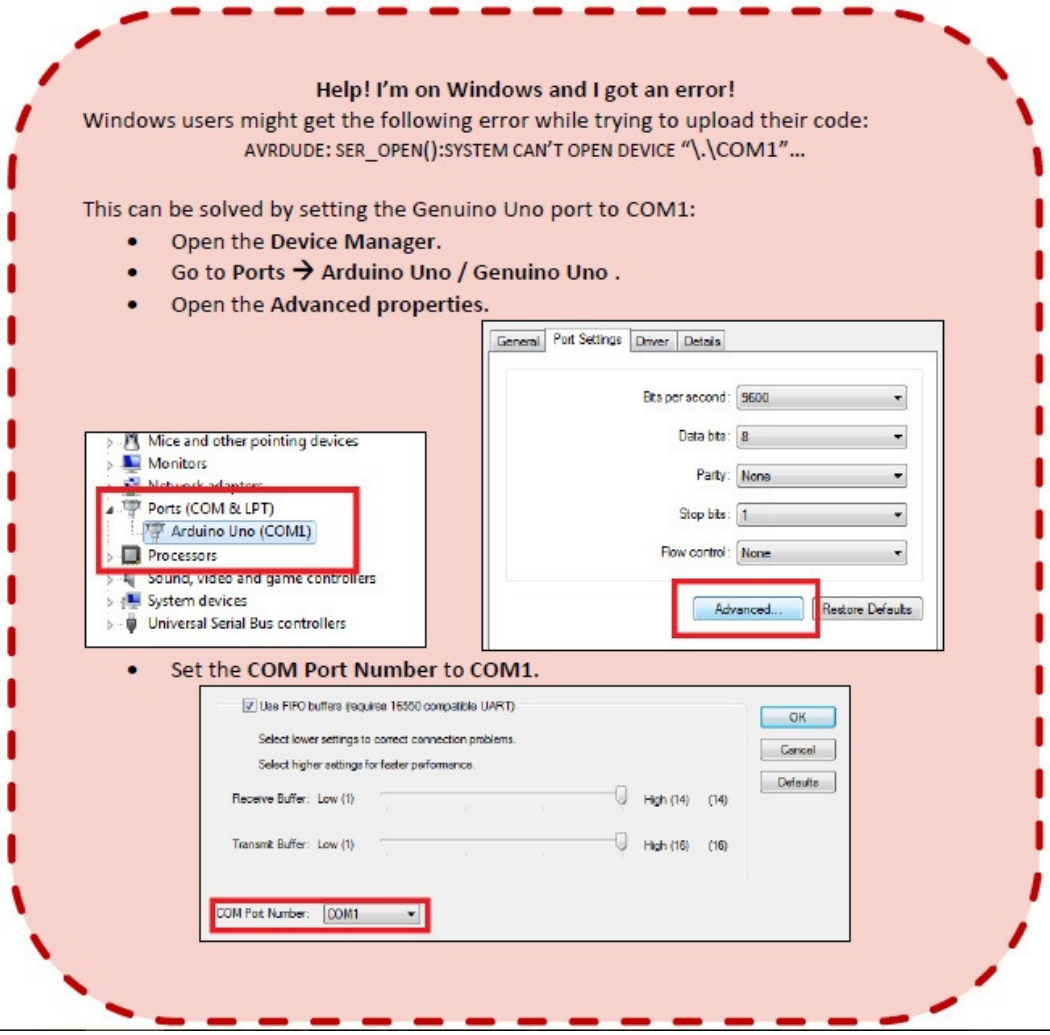

```
const int buttonPin = 2; // the number of the pin to which the pushbutton is connected
const int ledPin = 13; // the number of the pin connected to the LED
int buttonState = 0; // variable for reading the status of the pushbutton
void setup() {
  pinMode(buttonPin, INPUT); // initialize the pushbutton pin as an input
  pinMode(ledPin, OUTPUT); // initialize the LED pin as an output
}
void loop() {
  buttonState = digitalRead(buttonPin); // get the value of the pushbutton
  //check if the pushbutton is pressed in
  if (buttonState == HIGH){
  digitalWrite(ledPin, HIGH); // if the button is pushed, turn LED on
  } else {
  digitalWrite(ledPin, LOW); // if the button is released, turn LED off
  }
```

```
}
```

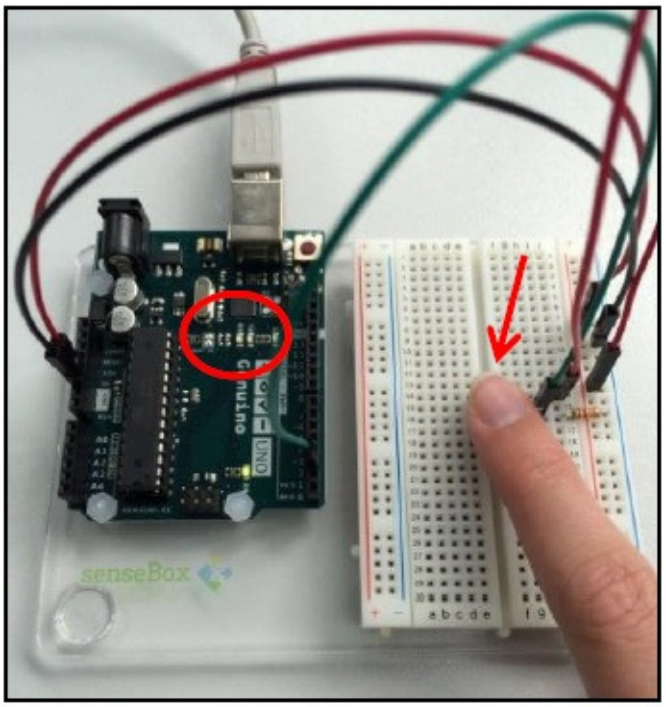
1. Compile and upload the sketch to the board
    - Connect the board to your computer with a USB cable.



- Compile the code by clicking on the Verify button (A).
- Click on the Upload button (B).

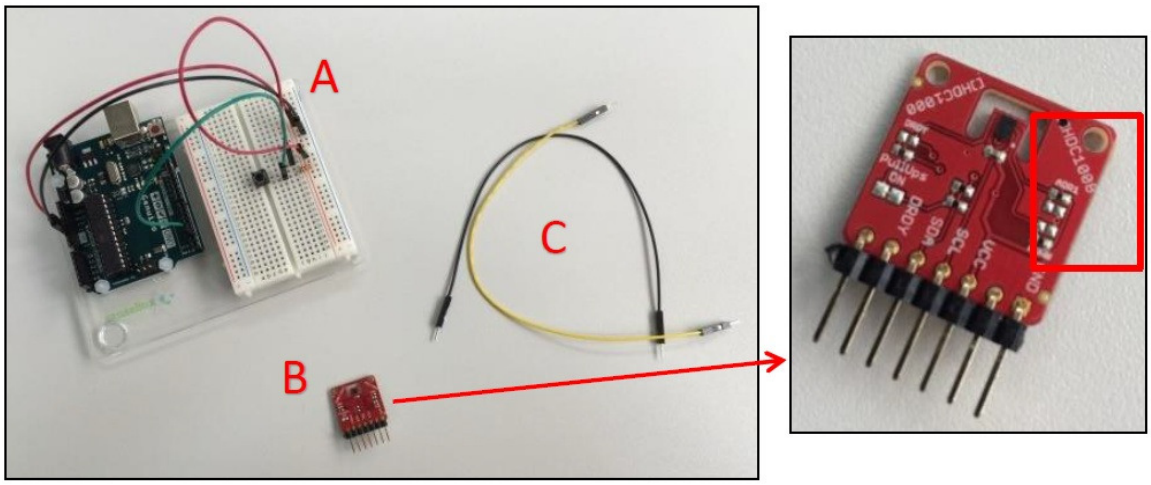


2. Test your program
    - Push the button on the board.
    - A LED light on the Genuino Board should light up.

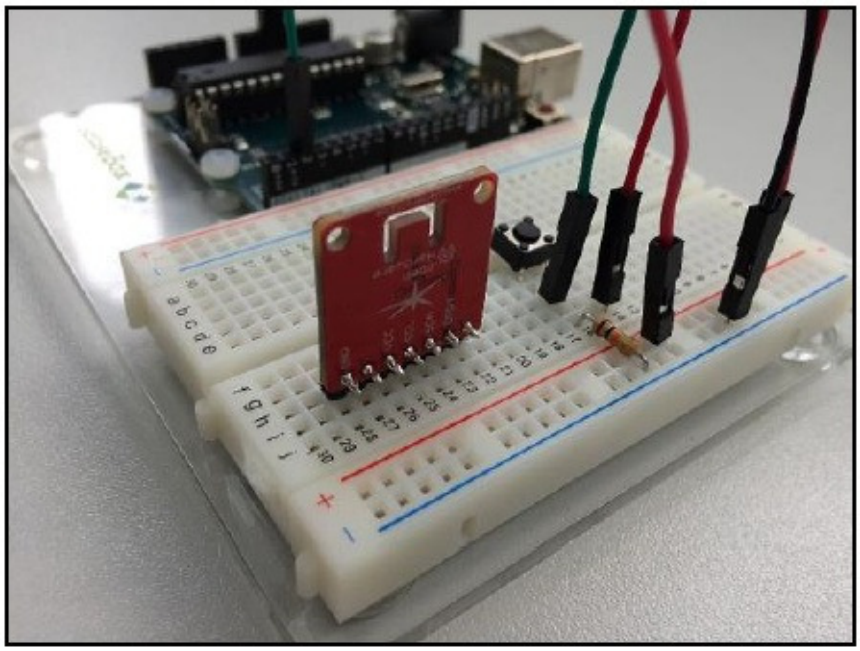


Let's build the Sensor Data logger

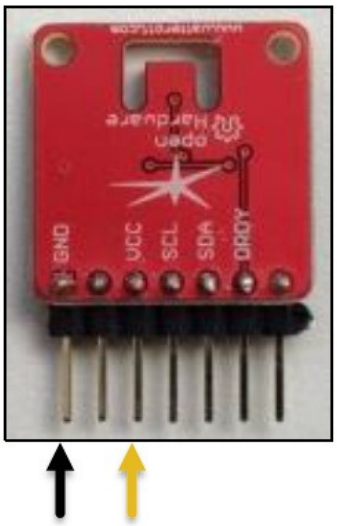
1. What do you need?
    - Your "Push-a-button" construction (A)
    - HDC1008 Temperature and Humidity Sensor (B)
    - Extra wires (C)

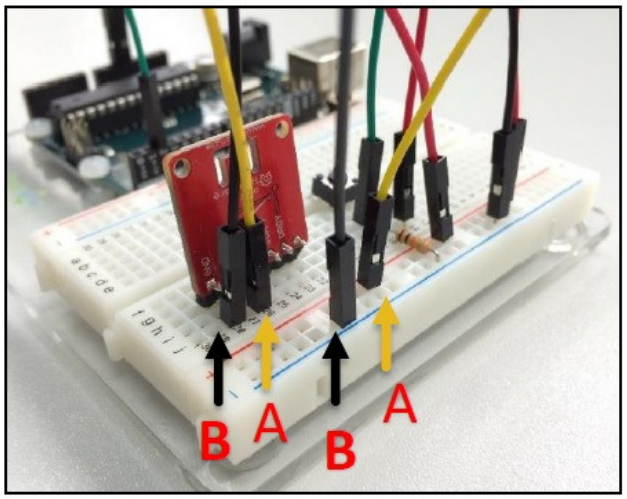


2. Connect the sensor and provide it with power and grounding
    - Connect the sensor to the breadboard.

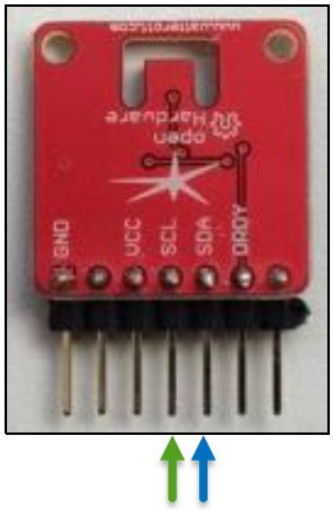


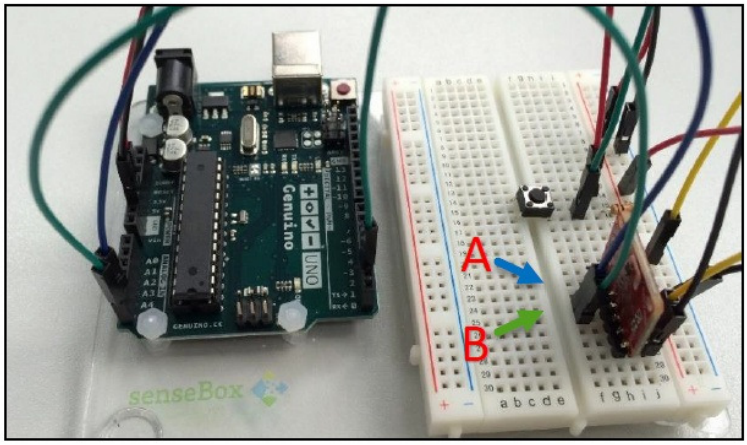
- The labels on the sensor mark tell which pins need to be connected to the power and ground. If you are unsure, please consult the data sheet for your sensor.
- Connect a yellow wire to the positive vertical row and to the VCC pin of the sensor (A).
- Connect a black wire to the negative vertical row and to the GND pin of the sensor (B).

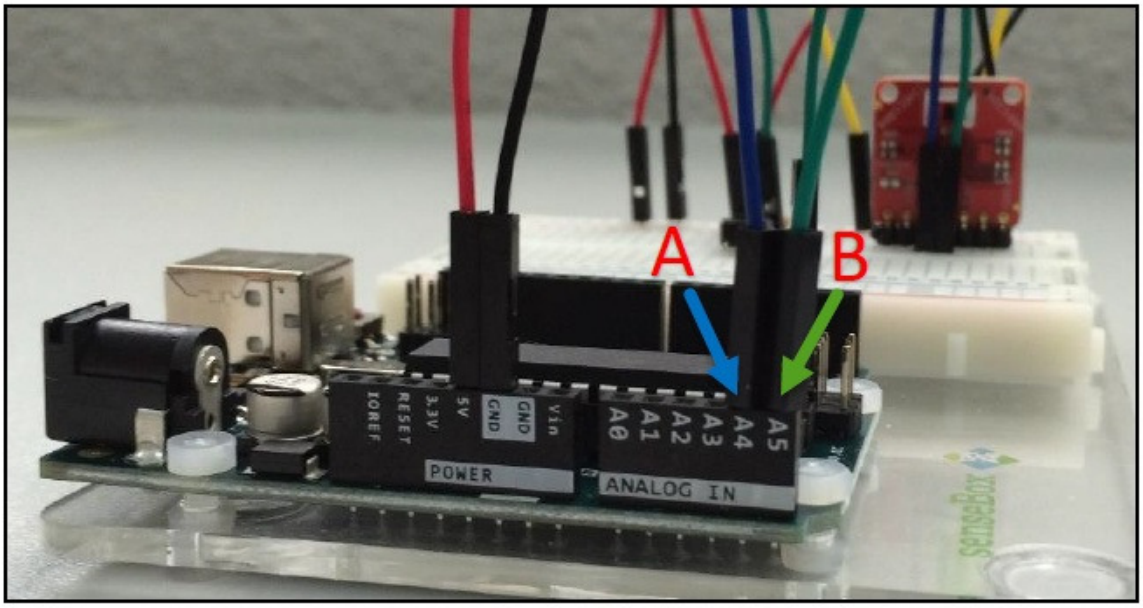




3. Create a connection between the sensor and the microcontroller to transmit measurements
   - Provide with a blue wire a connection between the analog pin AV4 (you may also connect to another free analog pin; then check if you address the sensor in your code correctly) on the Genuino Board and the SDA pin of the sensor.
   - Provide with a green wire a connection between the analog pin AV5 (you may also connect to another free analog pin; then check if you address the sensor in your code correctly) on the Genuino Board and the SCL pin of the sensor.







Write the program for the Weather Station

1. Create a new sketch
    - Start the Arduino IDE.
    - Go to File New and name the file PushbuttonTemperature.ino to create your own sketch.
2. Include the required libraries
    - Add the following code to your program to add two libraries:
        - Wire.h provides functions to communicate with the data line (SDA) pin and clock line (SCL) pin.
        - HDC100X.h makes communication and actions with the SD card possible.

        ```
        #include <Wire.h>;
        ```

        ```
        #include <HDC100X.h>;
        ```
3. Define global variables
    - Create a connection with the sensor and pushbutton.

- Define variables for the button state and id.

```
HDC100X HDC1(0x43); // create a connection to the sensor on address 0x43

int buttonPin = 2; // the number of the pin to which the pushbutton
// is connected, change this if you
// connected the button to a different port

int buttonState = 0; // variable for reading the status of the pushbutton

int lastButtonState = 0; // previous state of the pushbutton

int id = 0; // count for how many times the button has been pressed
// and set this as ID
```

1. Write the setup() function
   - Start the sensor.
   - Initialize the pushbutton as an input.

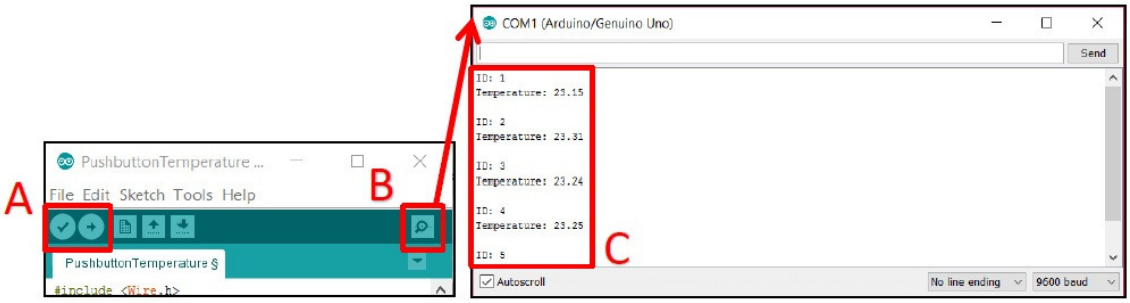
```
void setup() {

Serial.begin(9600); // provide communication with the
// computer with data rate 9600 bits per second

HDC1.begin(HDC100X\_TEMP\_HUMI, HDC100X\_14BIT, HDC100X\_14BIT,
DISABLE); // start the sensor

pinMode(buttonPin,INPUT); // initialize the pushbutton pin
// as an input

}
```

2. Write the loop() function
   - Read the value from the push button.
   - If the button is pressed, measure the temperature and print it to the console.

```
void loop() {

  buttonState = digitalRead(buttonPin); // read value from push button

  // if button is pressed in

  if (buttonState != lastButtonState && buttonState == HIGH) {

  id++; // update id

  Serial.print("ID: "); // print the id to the console

  Serial.println(id);

  Serial.print("Temperature: "); // get the temperature and
  // print it to the console

  Serial.println(HDC1.getTemp());

  Serial.print("\\n");

  }

  lastButtonState = buttonState; // set current state as last button state
```

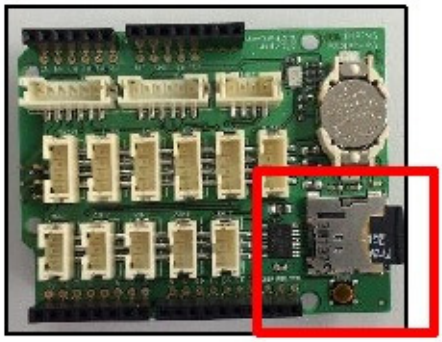
```
}
```

3. Run your program
    - Compile your program and upload it to the board (A).
    - Click on the Serial Monitor button. The COM1 dialog should pop up (B).
    - Push the button on your board. A new result should appear on your screen (C).

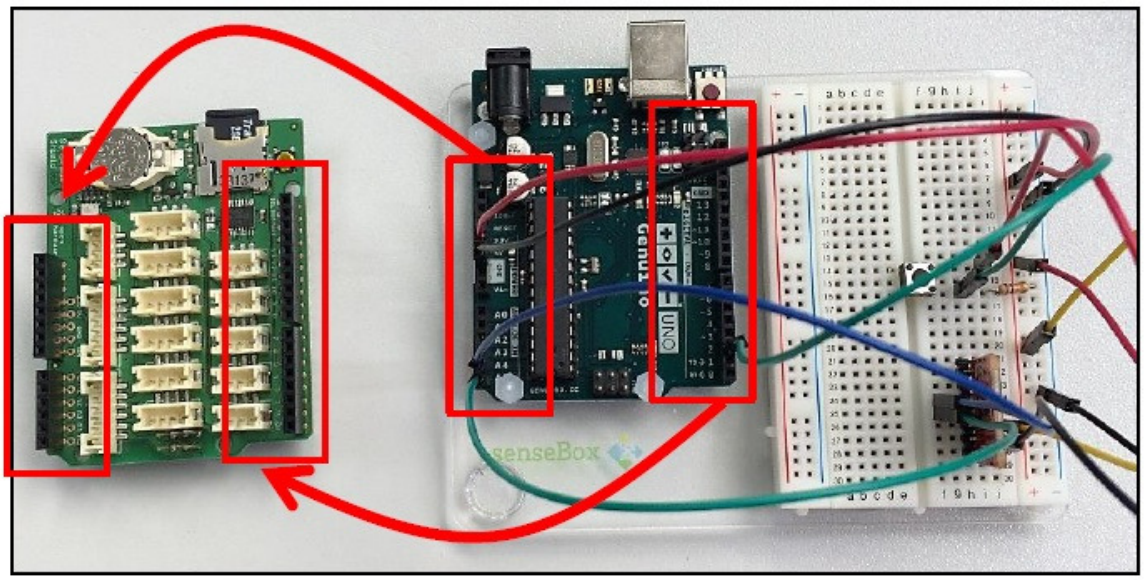


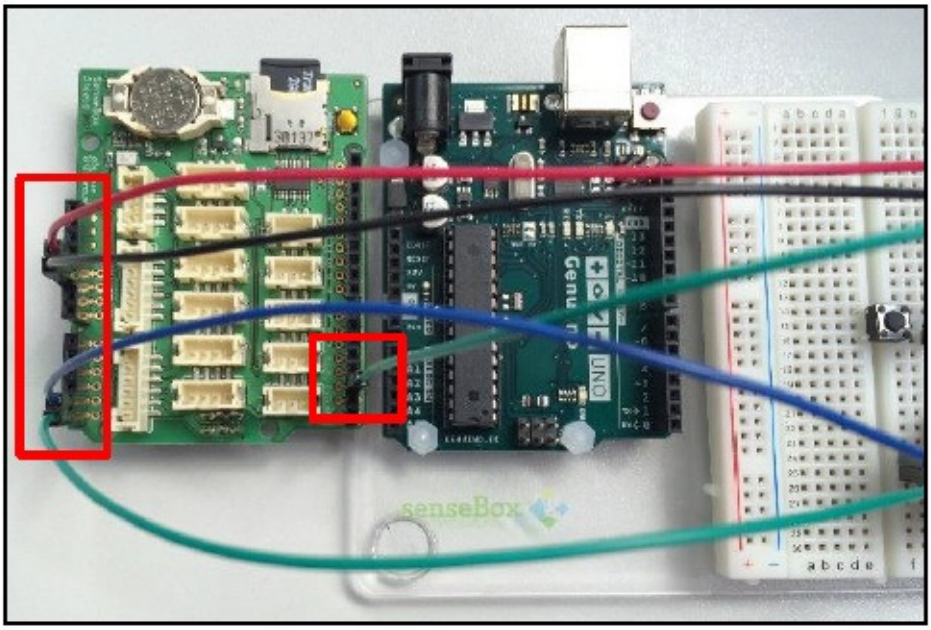
Write the results to a file on a SD card

1. What do you need?
    - Your Arduino construction from the previous steps.
    - A senseBox Shield with SD card slot.

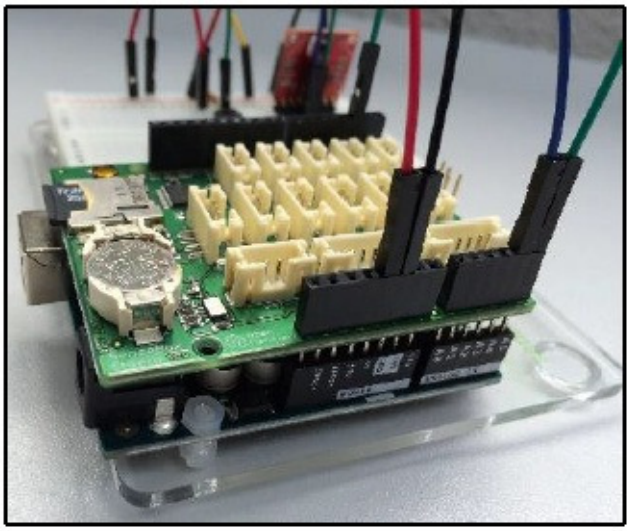


1. Connect the senseBox Shield to the Genuino Board
    - Since the senseBox Shield will be placed on top of the Genuino Board, the wires need to be connected to the senseBox Shield.





2. Place the senseBox Shield on the Genuino Board.



3. Adjust your program

    - Open the file PushbuttonTemperature.ino in the Arduino IDE.
    - Add a library for SD card functionalities:

```
#include <SD.h>
```

- Define a global variable which contains the identifier of the pin connected to the SD card.

```
int chipPin = 4; // pin connected to the SD card
```

    - Initialize the SD card in the setup() function.

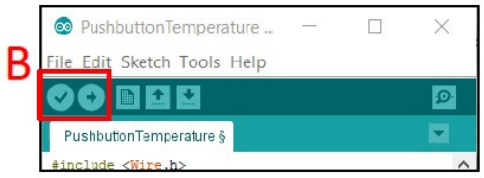
```
SD.begin(chipPin); // initialize the SD card.
```

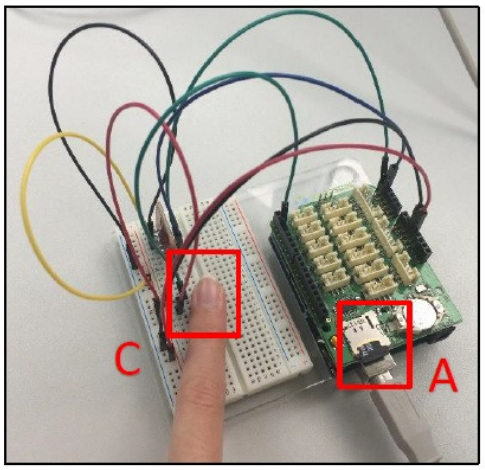
    - Adjust the loop() function.

```
void loop() {

  buttonState = digitalRead(buttonPin); // read value from push button

  // if button is pressed in

  if (buttonState != lastButtonState && buttonState == HIGH) {

    id++; // update id

    File file = SD.open("temp.csv", FILE\_WRITE); // open a file named temp.csv

    file.print(id); // print the id to the file

    file.print(";");

    file.print(HDC1.getTemp()); // get the temperature and print it to the file

    file.print("\\n");

    file.close(); // close the file

  }

  lastButtonState = buttonState; // set current state as last button state

}
```

- Run your program

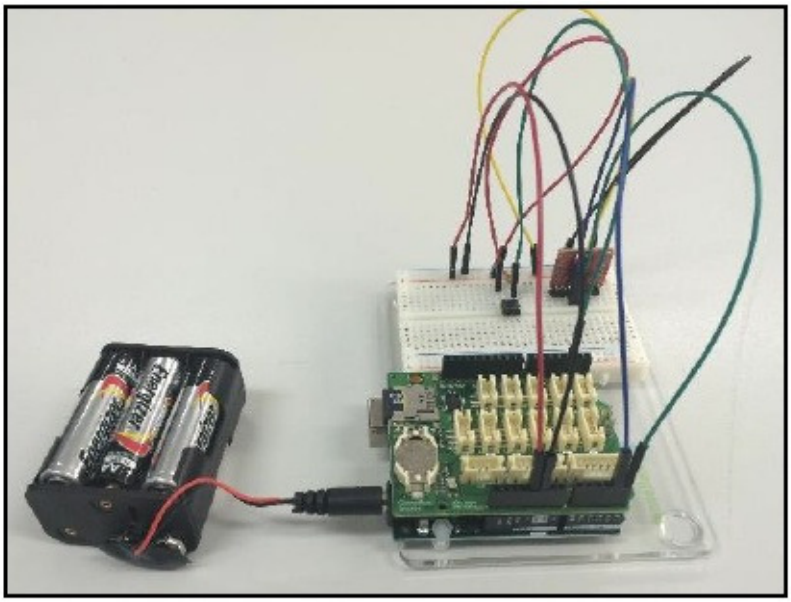
    - Place the SD card in the slot on the senseBox Board (A).
    - Compile your program and upload it to the board (B).
    - Push the button on the breadboard multiple times (C).





- View the results
  - Place the SD card in the card reader slot of your computer.
  - Open the file TEMP.CSV.
  - You can now see the resulting temperature measurements.

Make it mobile



Time to go on a field study and measure the temperatures throughout the building and surroundings. You can make your Arduino weather station more mobile by attaching a battery to the Genuino Board. Good luck during your expedition!

DIGITAL CITIZENS, CONNECTED COMMUNITIES:

THE ROLE OF SPATIAL COMPUTING AND VOLUNTEERED GEOGRAPHIC INFORMATION

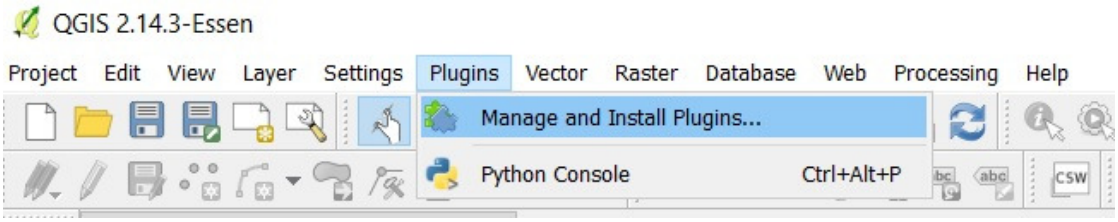
Tutorial: Create a heat map in QGIS

Edited by Felix Erdmann using pandoc

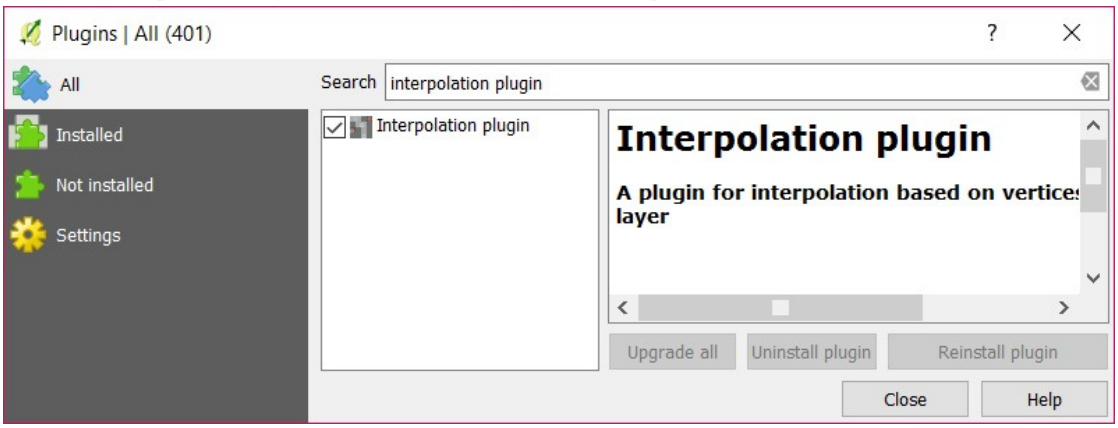
The field measurements you have acquired with your DIY weather station can be used to generate a thematic map. Since the measurements consist of temperature data a heat map is the most visualization type for that. In this tutorial you will learn how to visualize your data as a heat map in QGIS.

Preparation

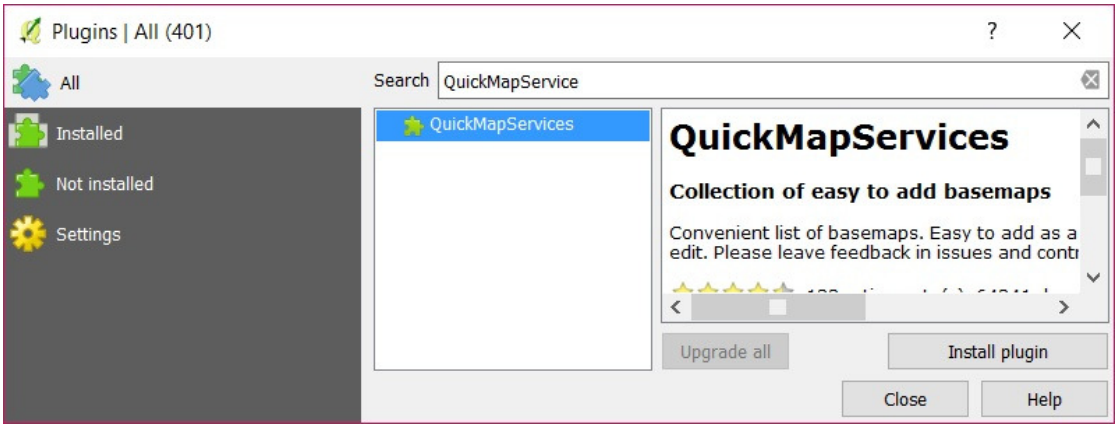
1. Install QGIS
    - http://www.qgis.org/nl/site/index.html
    - (Extra:) Documentation and a QGIS Training Manual can be found here:\ http://www.qgis.org/en/docs/index.html
2. Install Plugins
    - Open QGIS.
    - Go to Plugins Manage and Install Plugins...



    - Search for Interpolation Plugin (A). Select Install Plugin (B) if this option is still available.

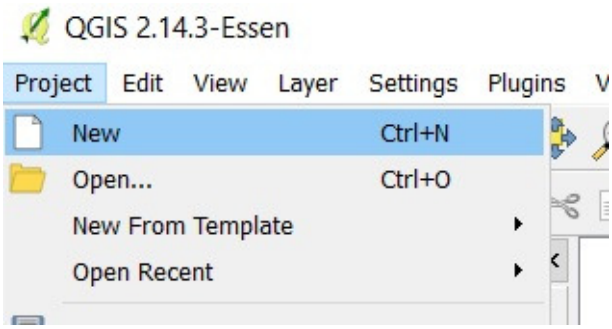


- Search for QuickMapService (A) and select Install Plugin (B).

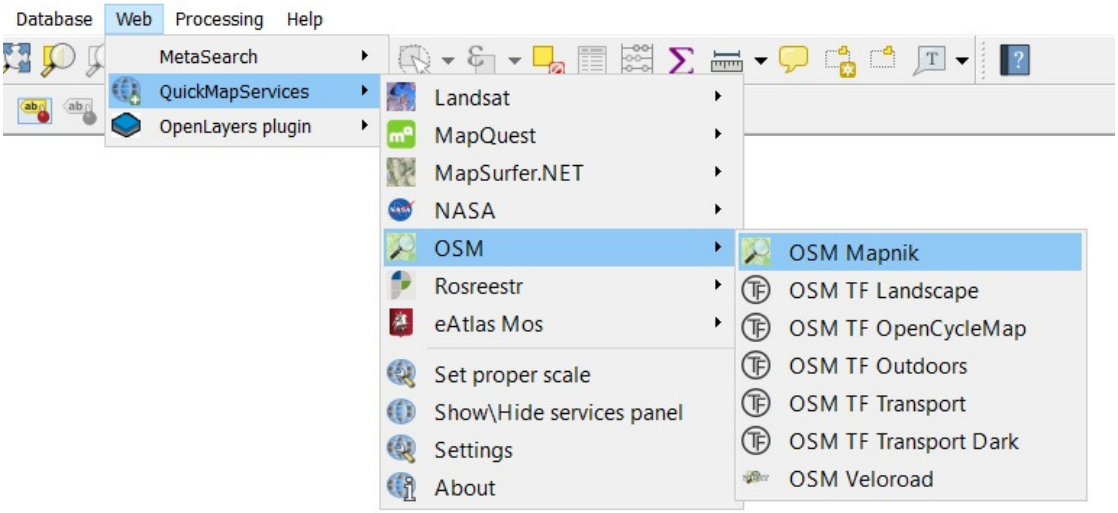


Import the data

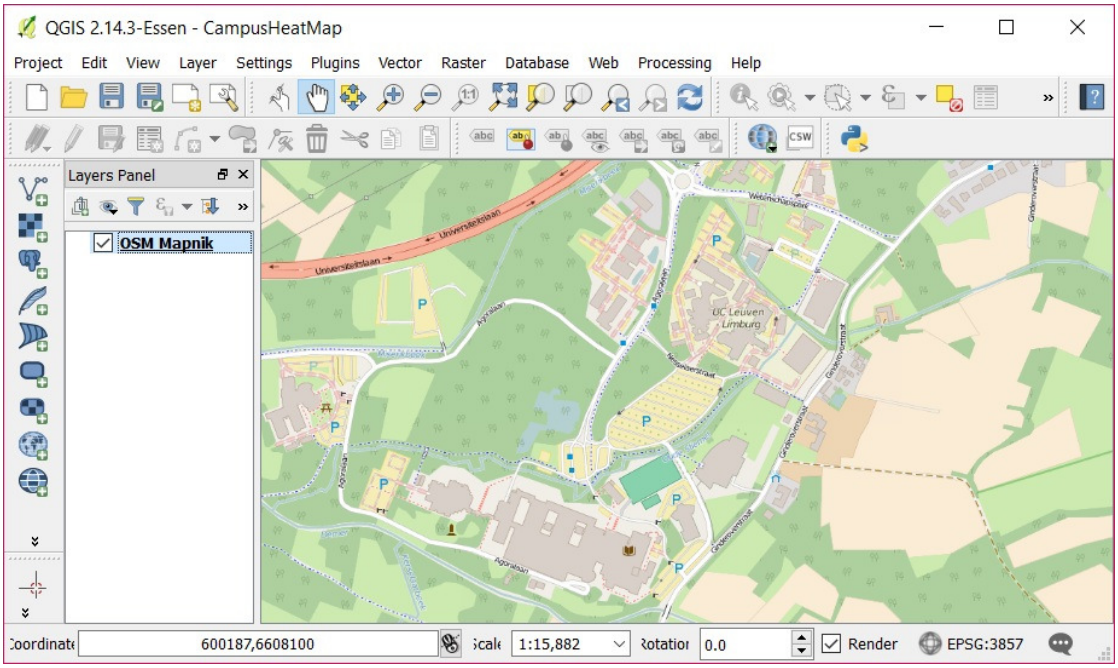
1. Create a new project
   - Go to Project New.



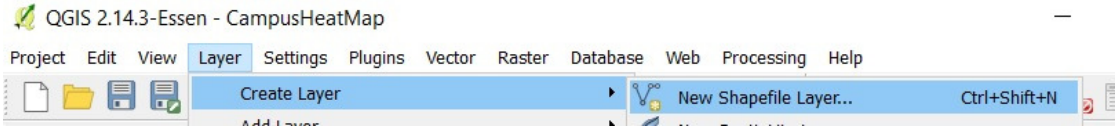
   - Save your project as CampusHeatMap.qgs.
2. Get OpenStreetMap data
   - Go to Web QuickMapServices OSM OSM Mapnik.

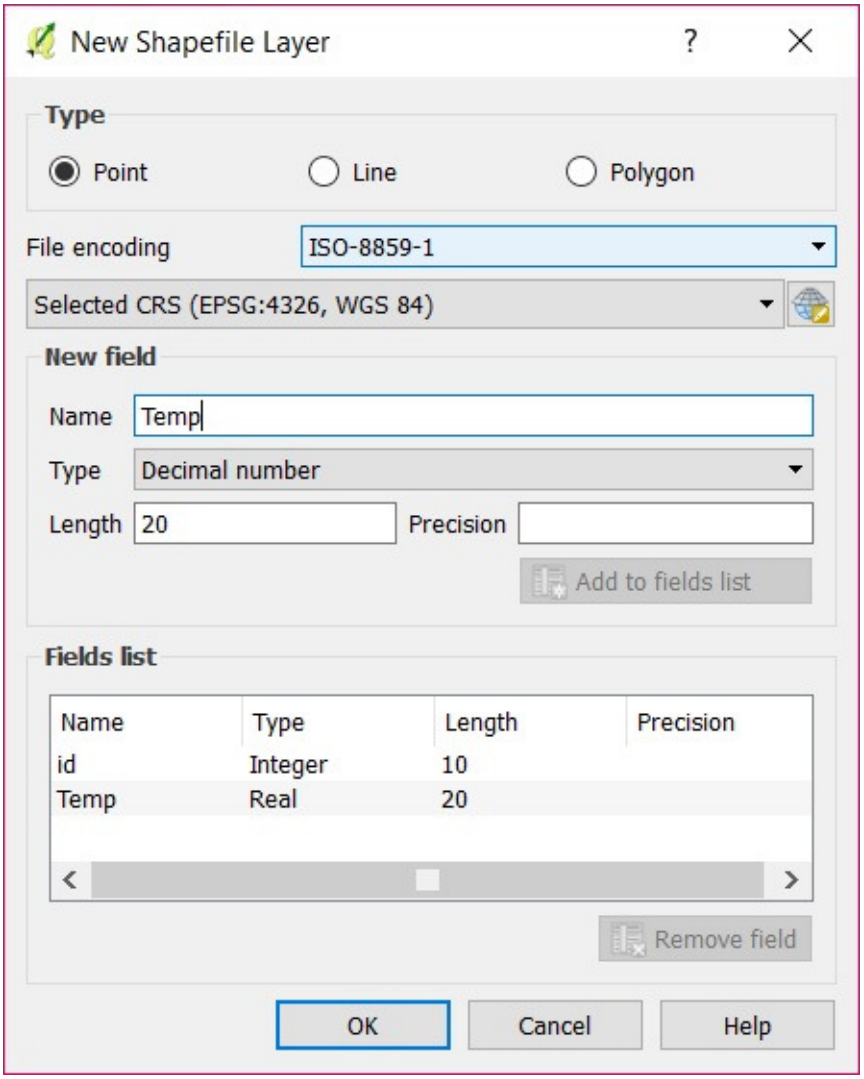


   - Open the layer OSM Mapnik and navigate to Campus Diepenbeek by using the Pan tool.



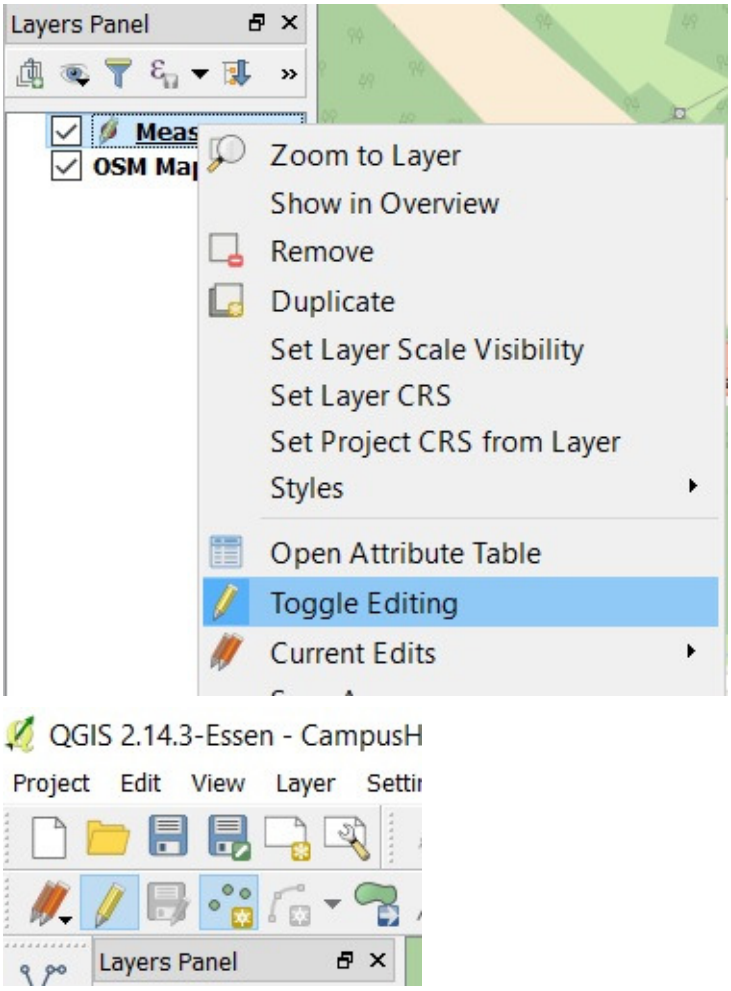
3. Add a layer for the temperature measurements
    - Go to Layer Create Layer New Shapefile Layer...



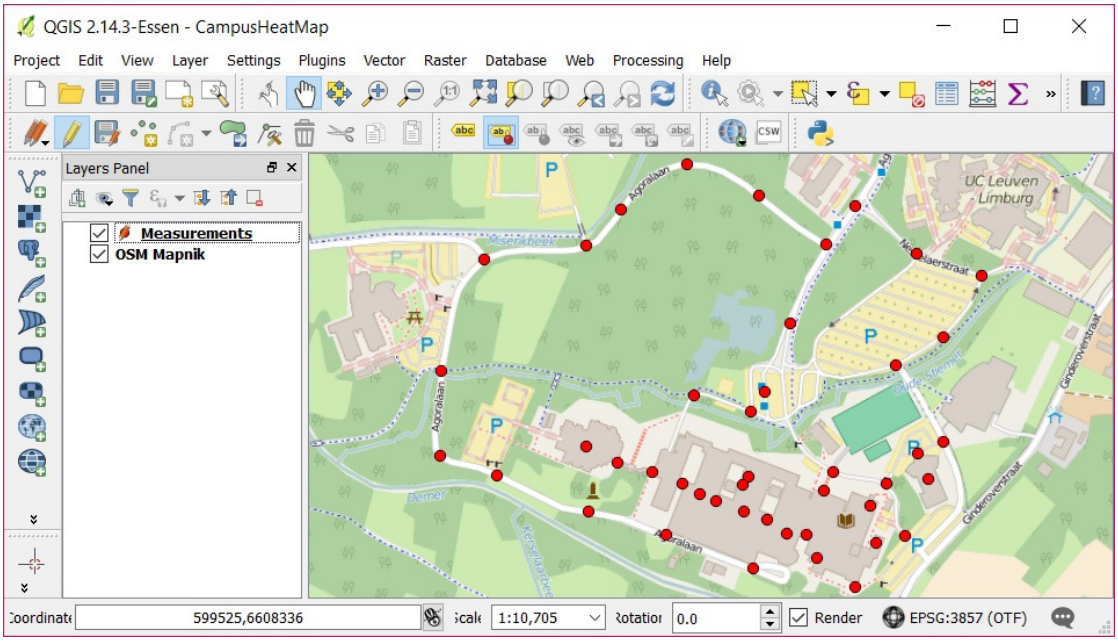


- Set the type of the new shapefile layer to Point (A).
- Add a new field Temp which type is Decimal number. Don't forget to click on the button Add to fields list (B).
- Click on OK (C).
- Name the layer Measurements.

4. Add points to the Measurements layer
   - Right click on Measurements and select Toggle Editing.
   - Click on the Add Feature button. Points will now be added when you click on the map.
   - Place points to mark all locations where you have measured the temperature. You can add the temperature values directly, but other methods will be explained later.



5. Changing the properties of the layers within the project
    - The checkboxes in front of the layers indicate whether a layer is visible or not.
    - By changing the order of the layers you decide which layers are in the front and which will function as backgrounds.
    - Make sure both layers are visible and Measurements is the top layer.

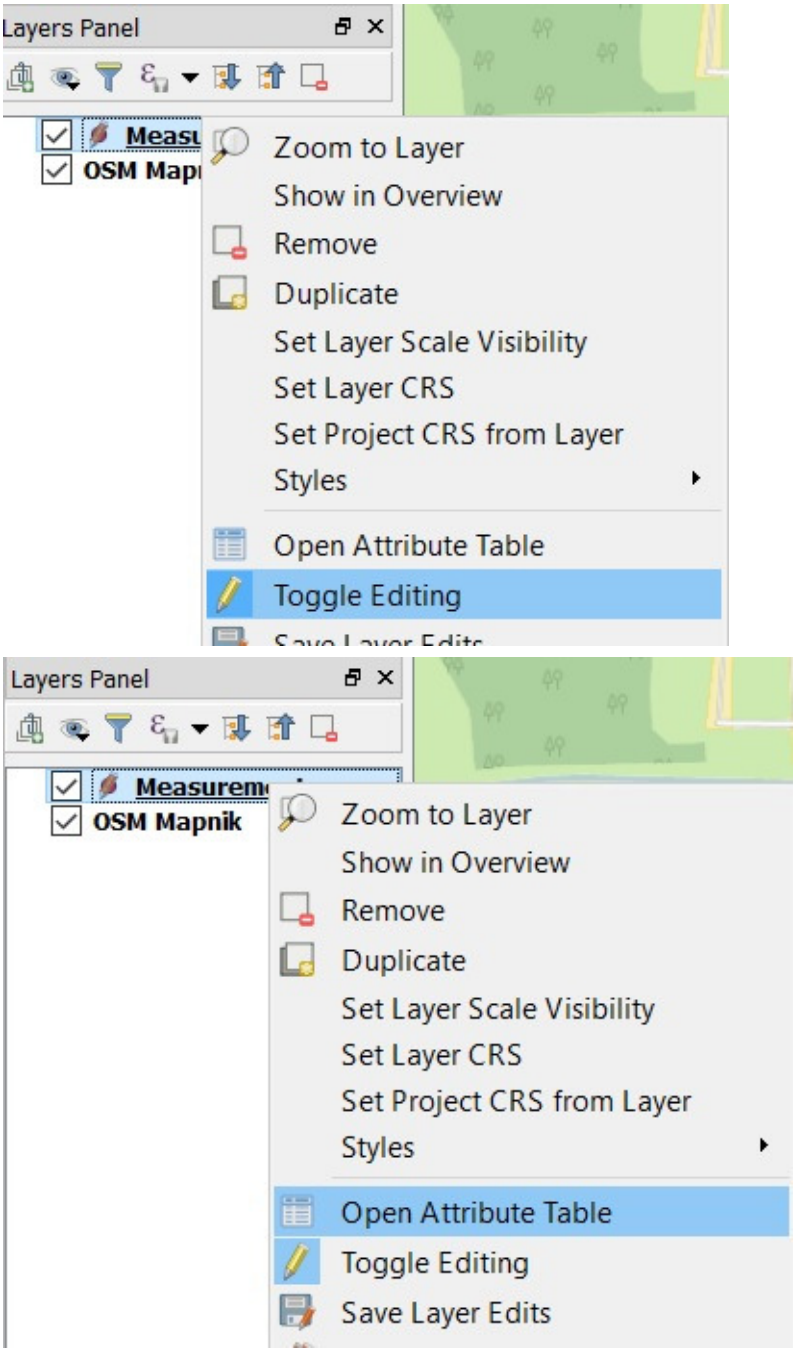


6. Update Measurements with your own data

   This can be done in various ways:

   Option 1: Insert the data manually

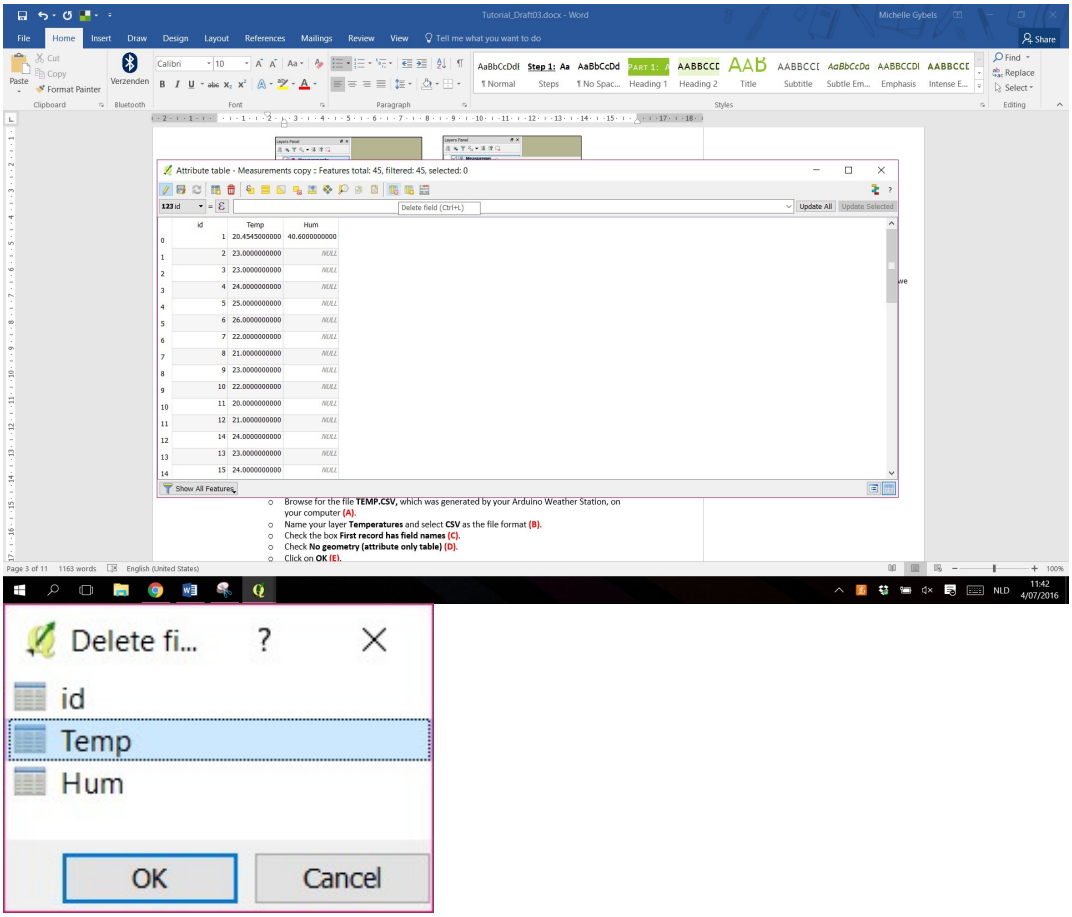
   - Right click on Measurements and select Toggle Editing (A).
   - Right click on Measurements again and select Open Attribute Table (B).



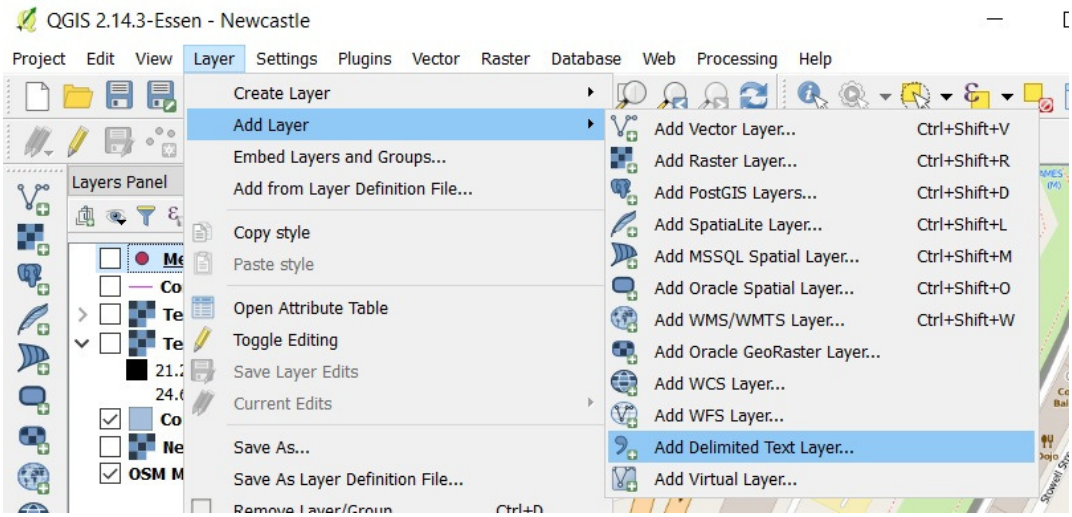
- Select each cell of the Temp field in the attribute table and fill in your obtained values.
- When you are done editing, select Toggle Editing again.

Option 2: Import the data from a CSV file

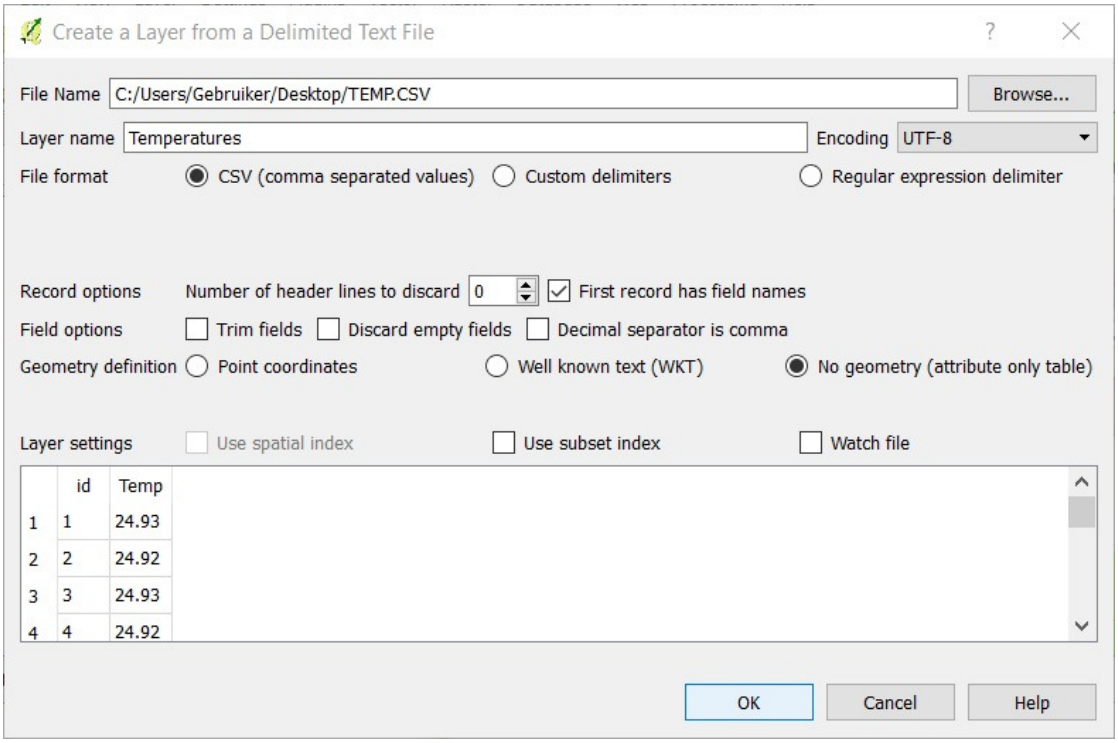
- Delete the Temp field from the attribute table.
  - Right click on Measurements and click on Toggle Editing.
  - Right click on Measurements and open the attribute table.
  - Click on the Delete field button (A).
  - Select Temp and click on OK (B).



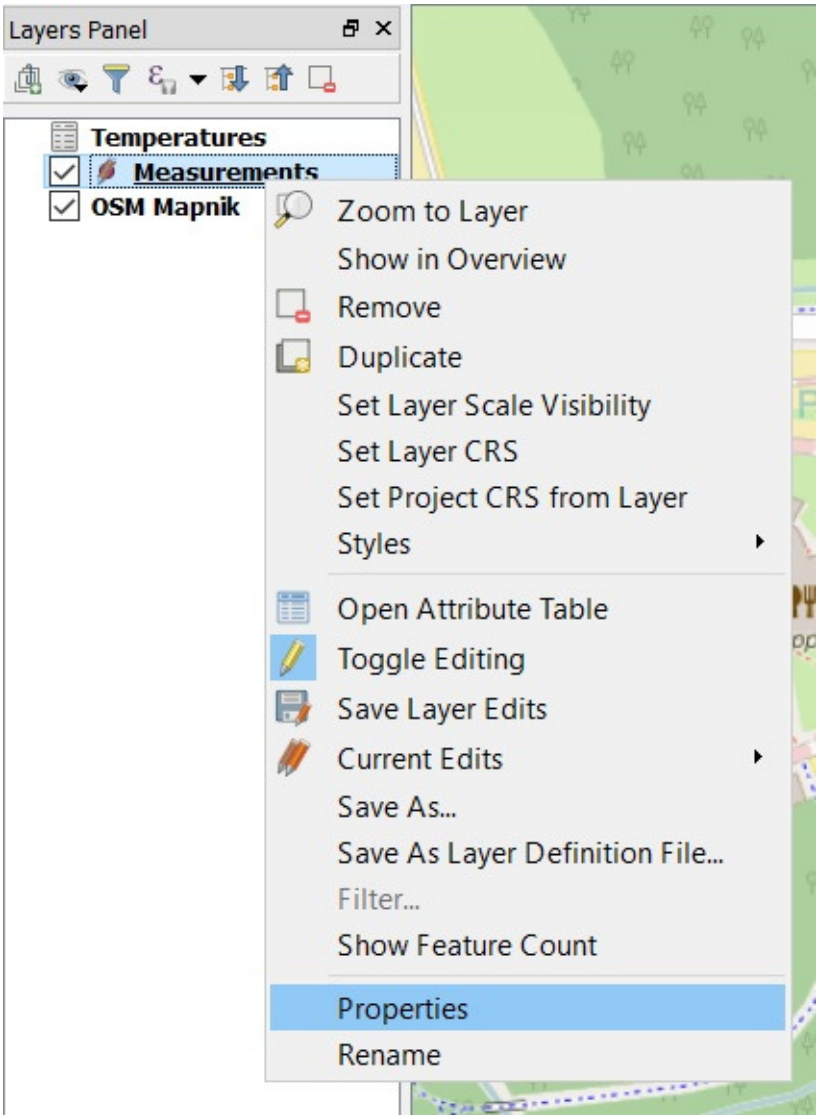
- Import your CSV file as a layer.
  - Go to Layer Add Layer Add Delimeted Text Layer...

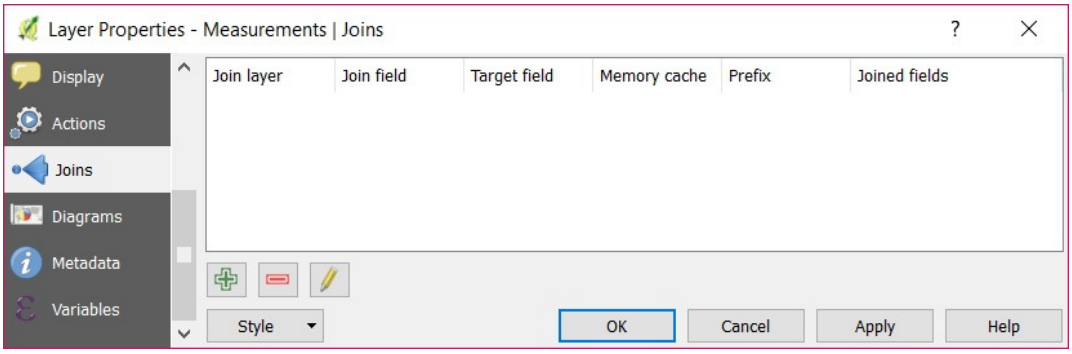


- Browse for the file TEMP.CSV, which was generated by your Arduino Weather Station, on your computer (A).
- Name your layer Temperatures and select CSV as the file format (B).
- Check the box First record has field names (C).
- Check No geometry (attribute only table) (D).
- Click on OK (E).

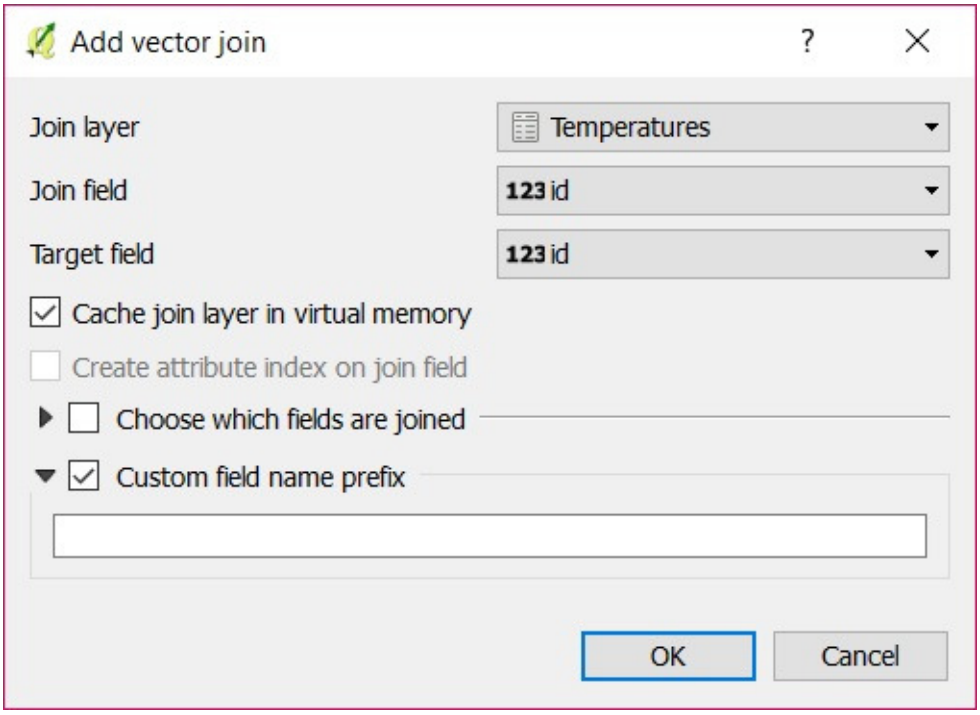


- Join the Temperatures layer with the Measurements layer.
  - Right click on Measurements and select Properties (A).
  - Open the tab Joins and click on Add (B).



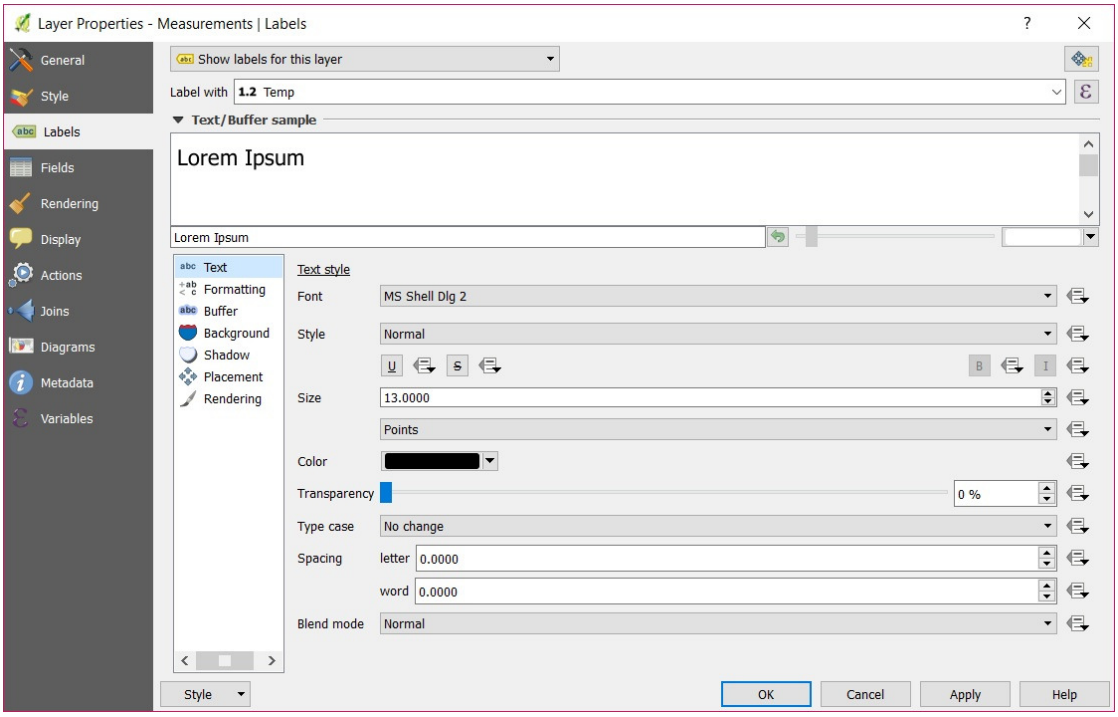


- Set Temperatures as the Join layer. And id as the Join field and Target field (A).
- Make sure the Custom field name prefix field is empty (B).
- Click on OK (C).



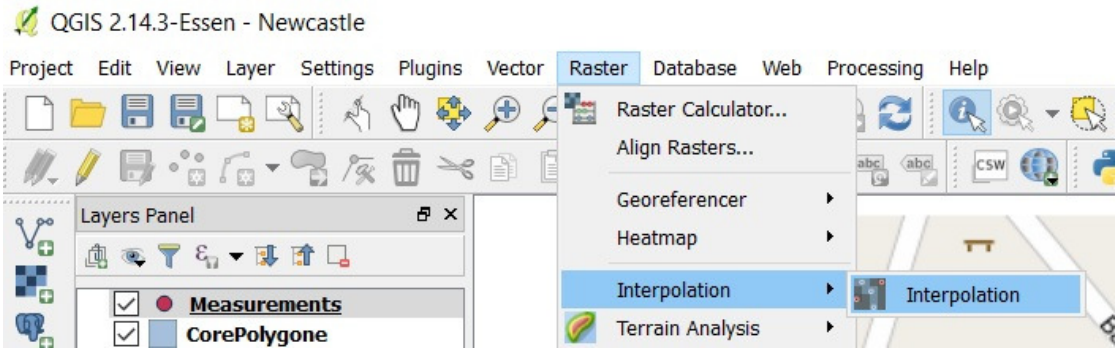
- Click on OK in the Layer Properties window.

7. Change the visualization of the layer Measurements
   - Right click on Measurements and select Properties.
   - Click on Labels and select Temp as variable to label the data points with (A).
   - Optionally, you can change the marker within the Style menu (B).

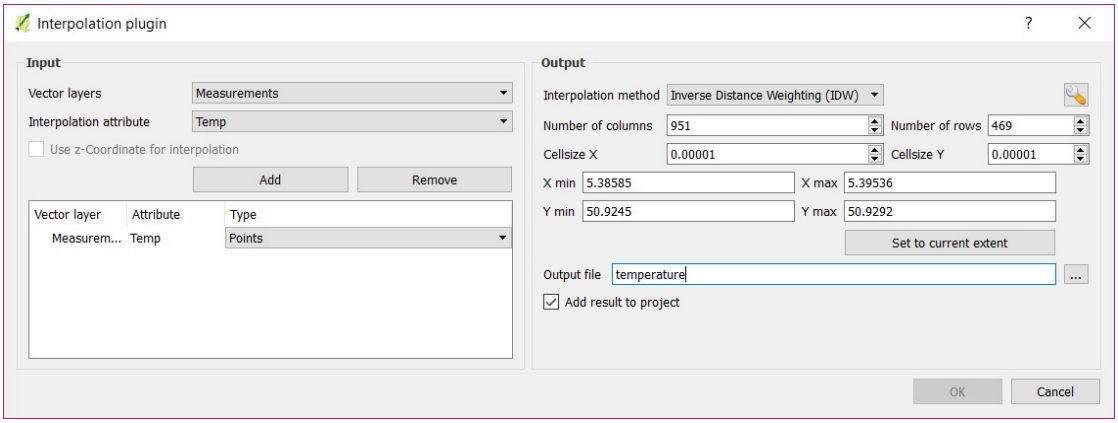


Interpolating Point Data

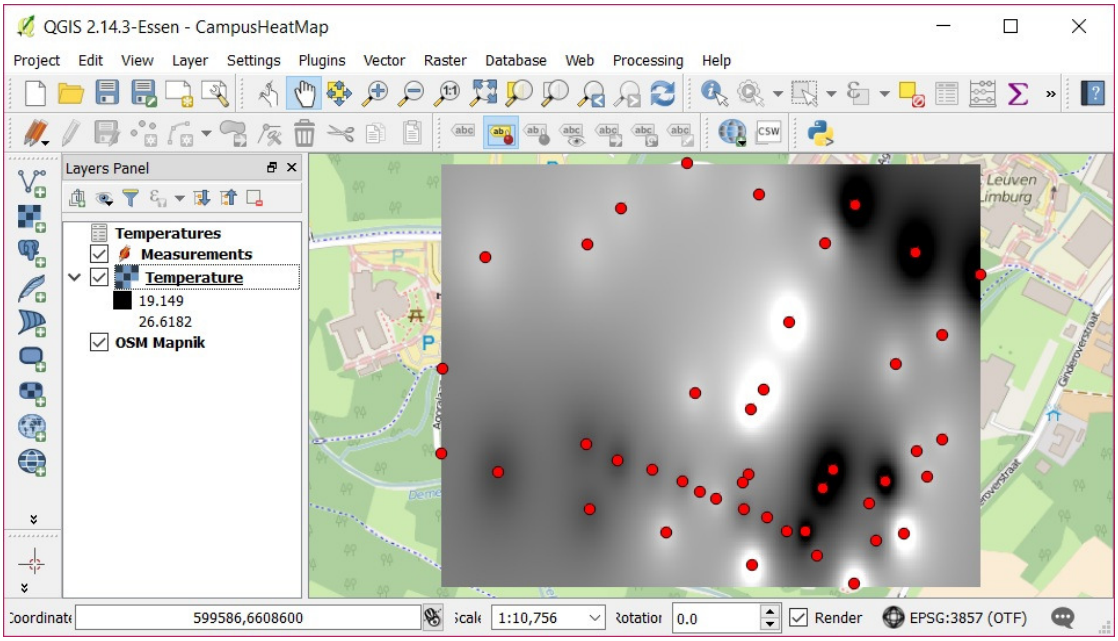
1. Open the Interpolation Plugin
    - Go to Raster Interpolation Interpolation.



2. Fill in the parameters within the Interpolation Plugin dialog
    - Set Measurements as the input vector layer and Temp as the interpolation attribute (A).
    - Click on Add (B).
    - Within the output section, select Inverse Distance Weighting (IDW) as interpolation method (C).
    - Fill in the other values as presented in the image below (D).
    - Navigate to a folder to save the generated output file and name it temperature (E).
    - Click on Ok (F).

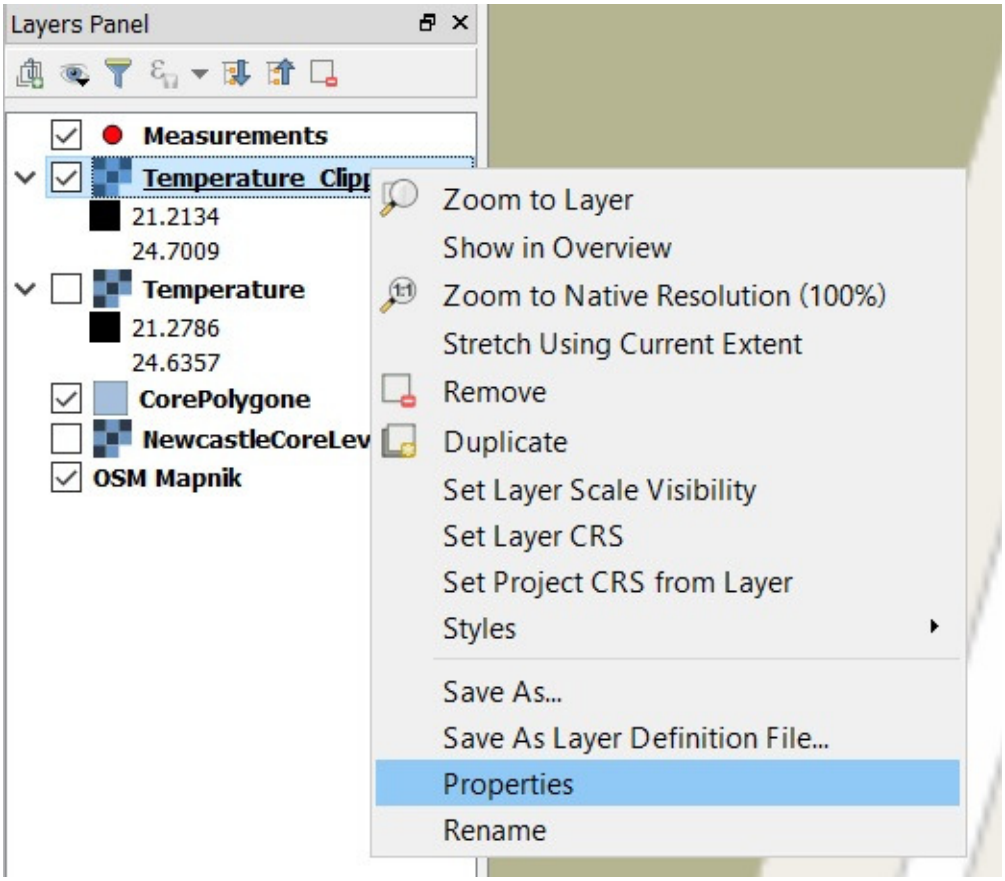


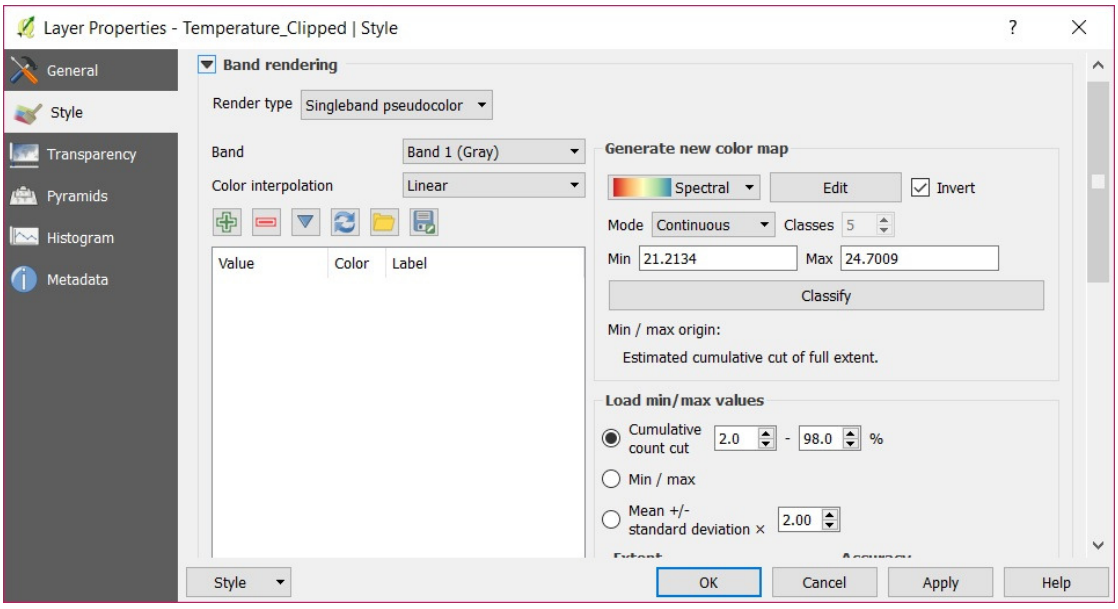
3. Verify your result
    - The generated output file is added as a layer within the project.
    - To zoom in on the layer, right click on Temperature and select Zoom to Layer.
    - Move the Measurements layer on top of Temperature.



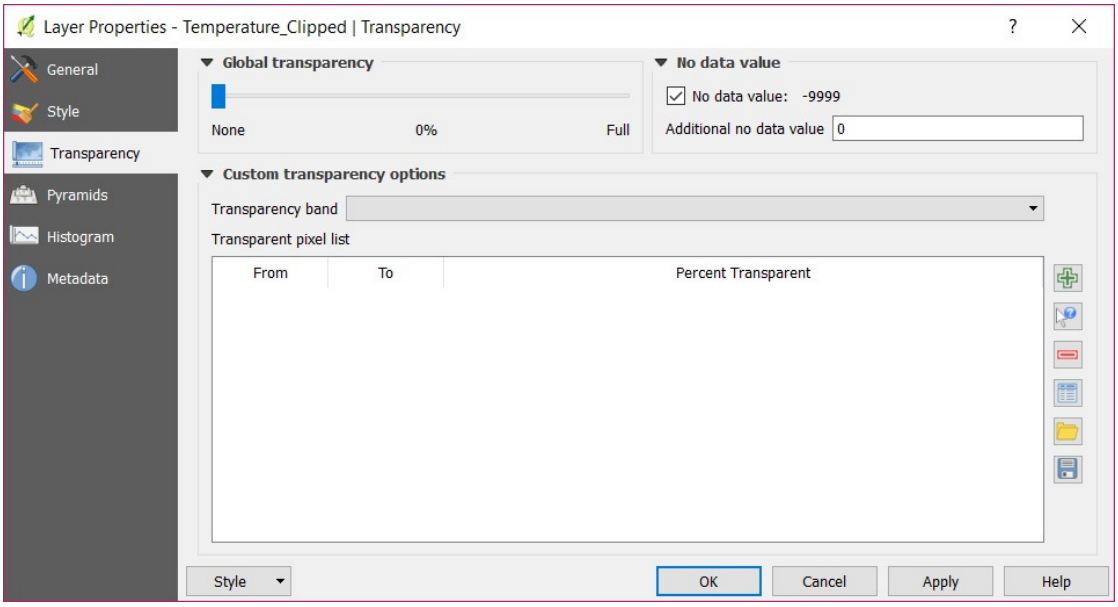
Change the style of the (clipped) layer

1. Adjust the color map
    - Right click the Temperature layer and select Properties (A).
    - Open the Style tab and set the render type to Singleband pseudocolor (B).
    - Select Spectral color map (C).
    - Check the Invert box, so that blue will be assigned to low temperatures and red to high temperatures (C).
    - Click on Classify (D).

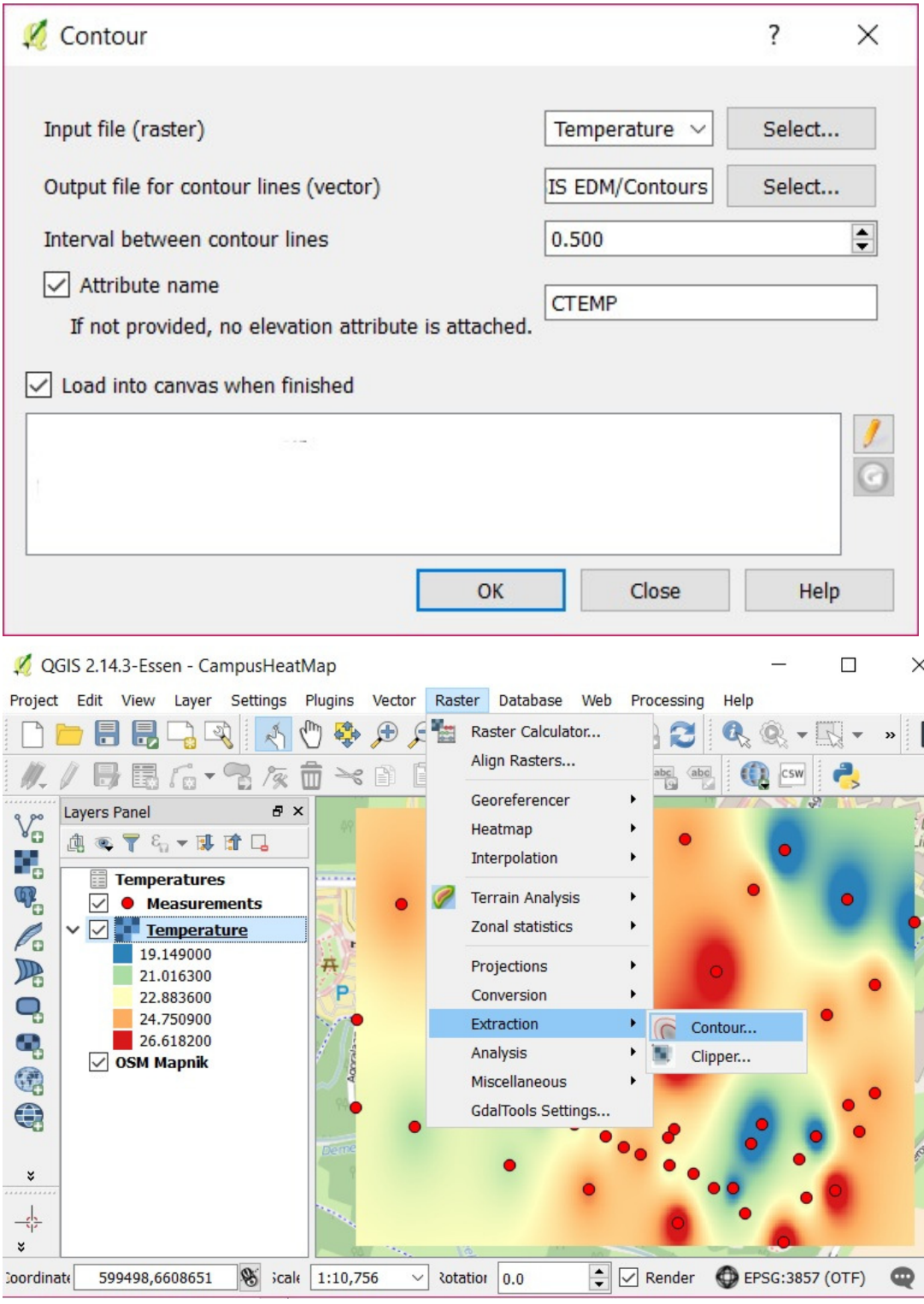




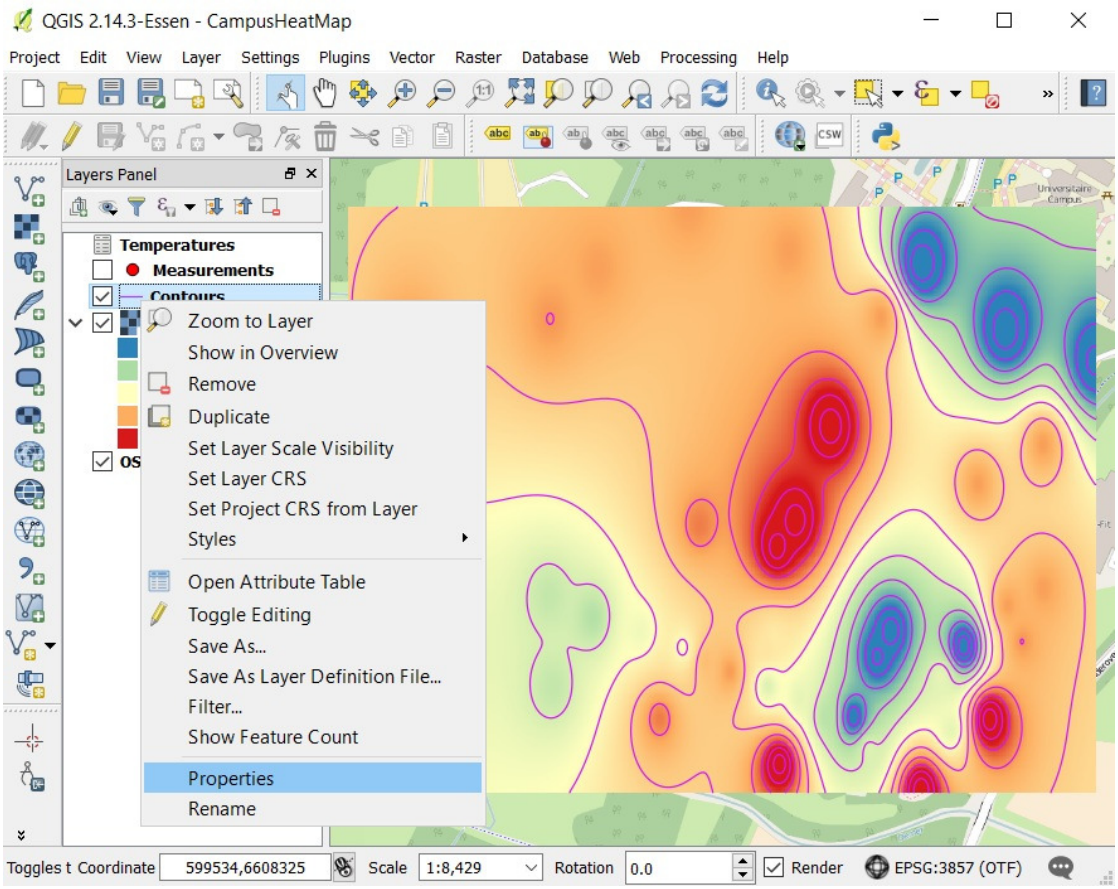
2. Remove the black pixels from the output
    - Open the Transparency tab.
    - Set 0 as the additional no data value.
    - Click on Ok.
    - This results in a temperature relief map for the campus.



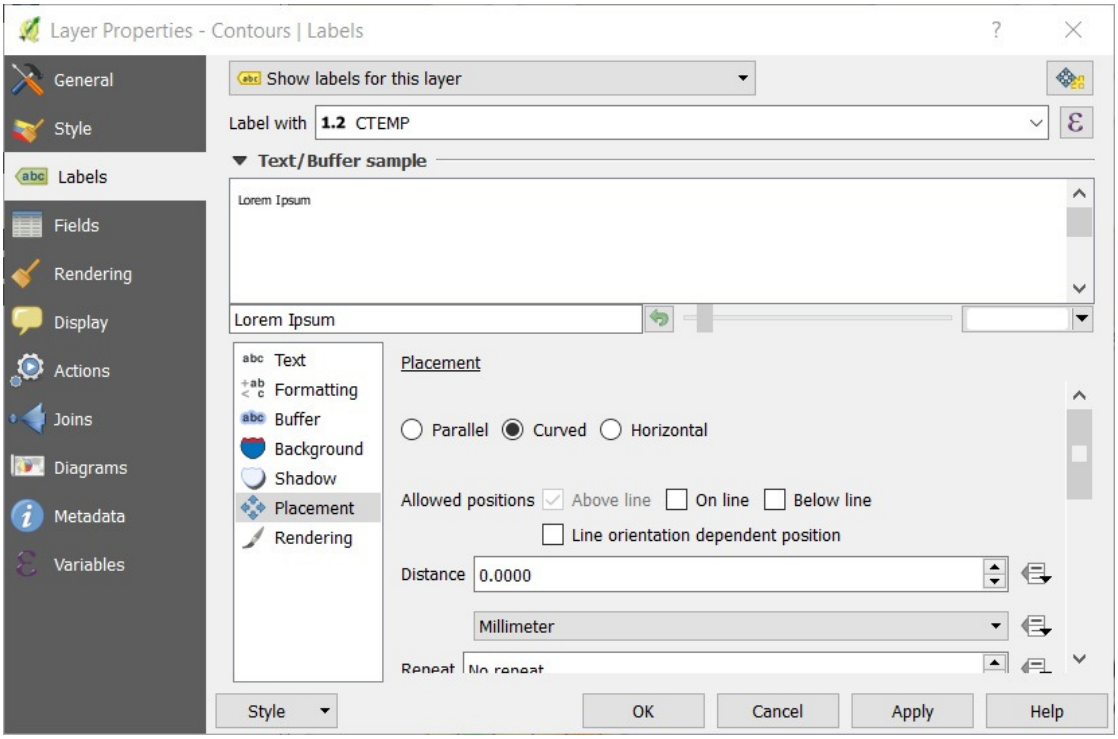
3. [Optional] Generate contours lines
    - Go to Raster Extraction Contour (A).
    - Set the Temperature_Clipped layer as the input file and name the output file Contours (B).
    - Set the interval between contour lines as 0.5 or 1(C).
    - Check the Attribute name box and set this to CTEMP (D).
    - Click on OK (E).



4. Add labels to the contour lines
    - Right click on the layer Contours and select Properties.

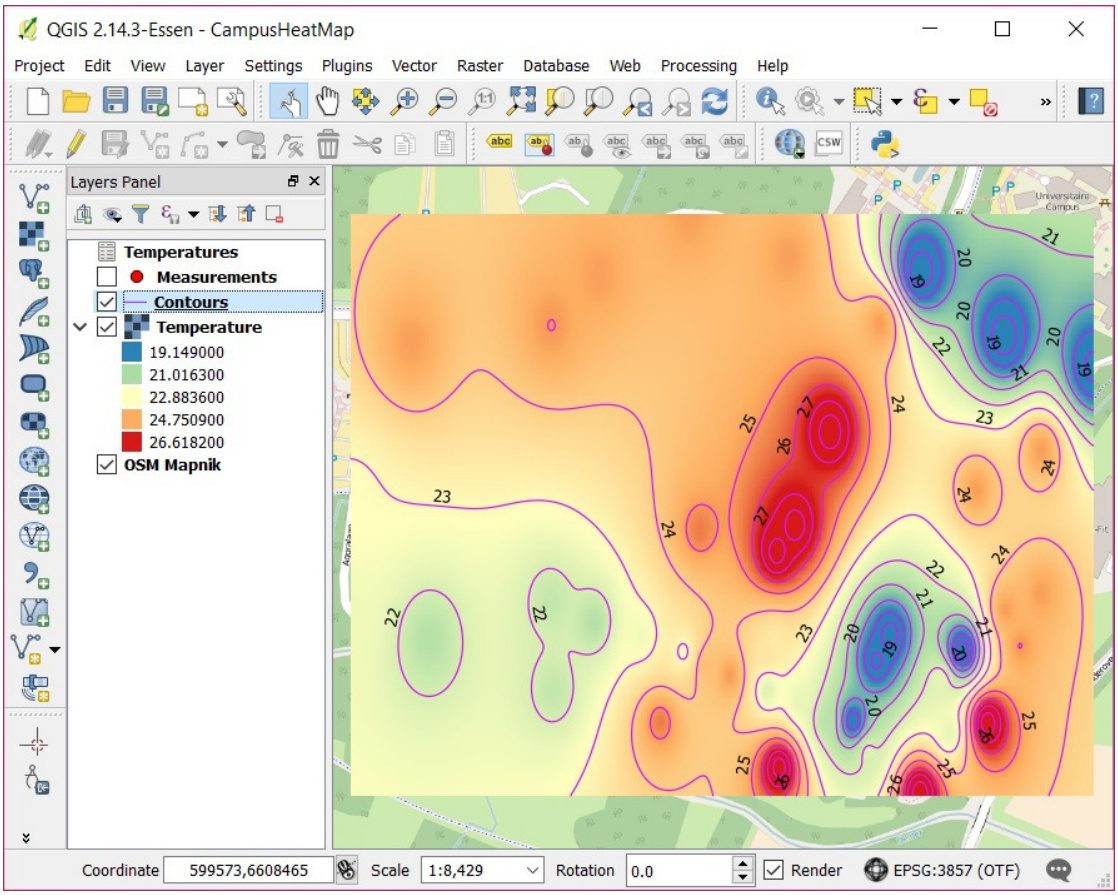


- Open the Labels tab (A).
- Select Show labels for this layer in the dropdown box (B).
- Set CTEMP as the value for Label with (C).
- Select Curved as Placement type (D).
- Click on OK (E).



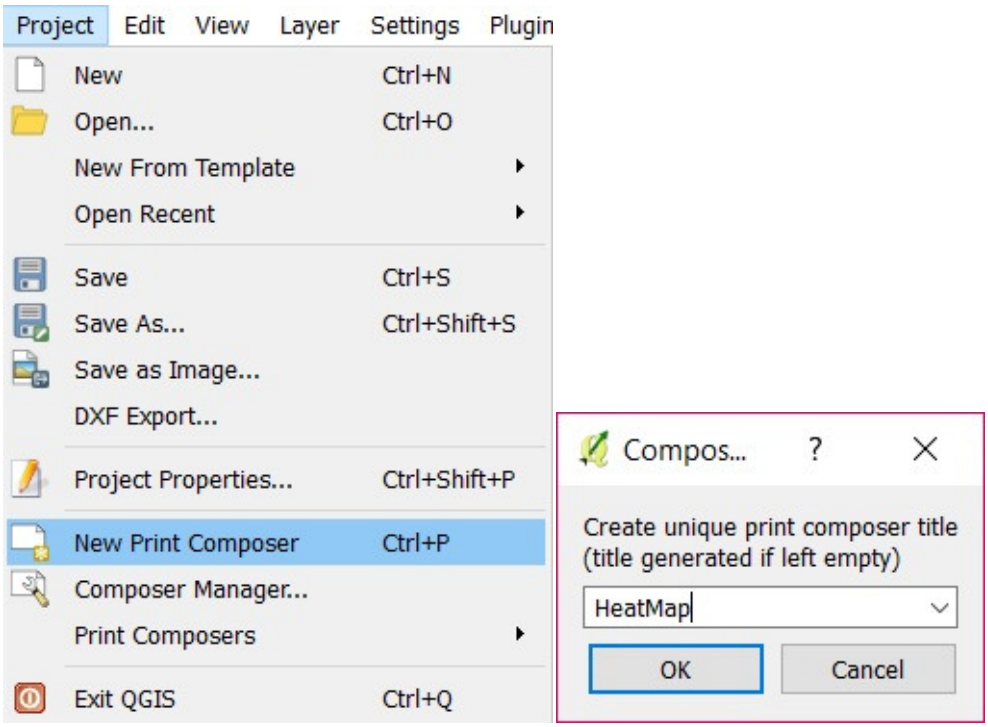
5. Verify the result

   - The result should look like this:

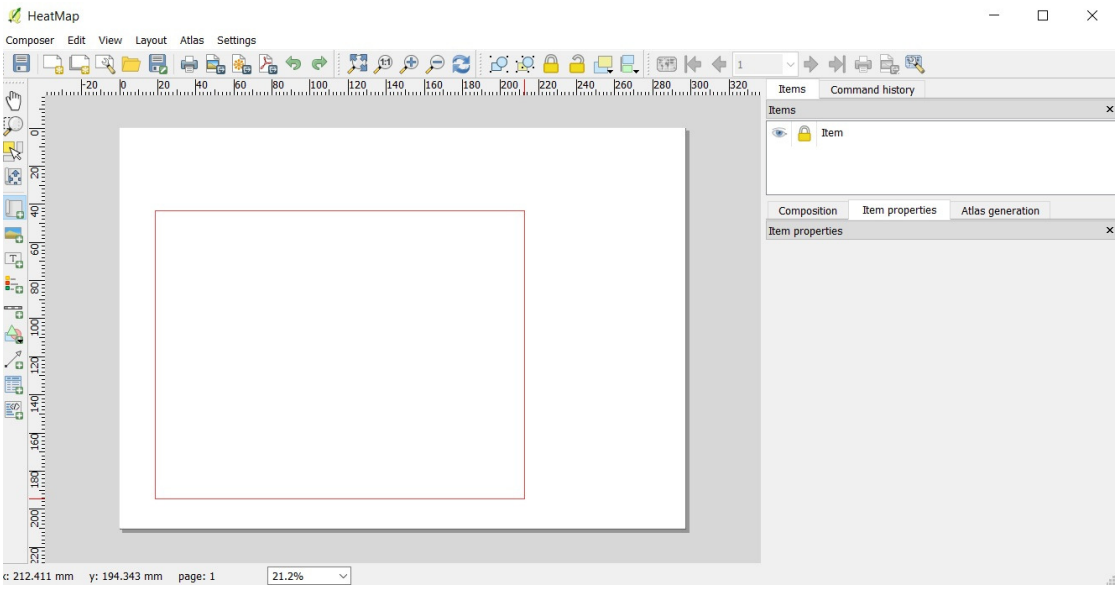


Exporting the result

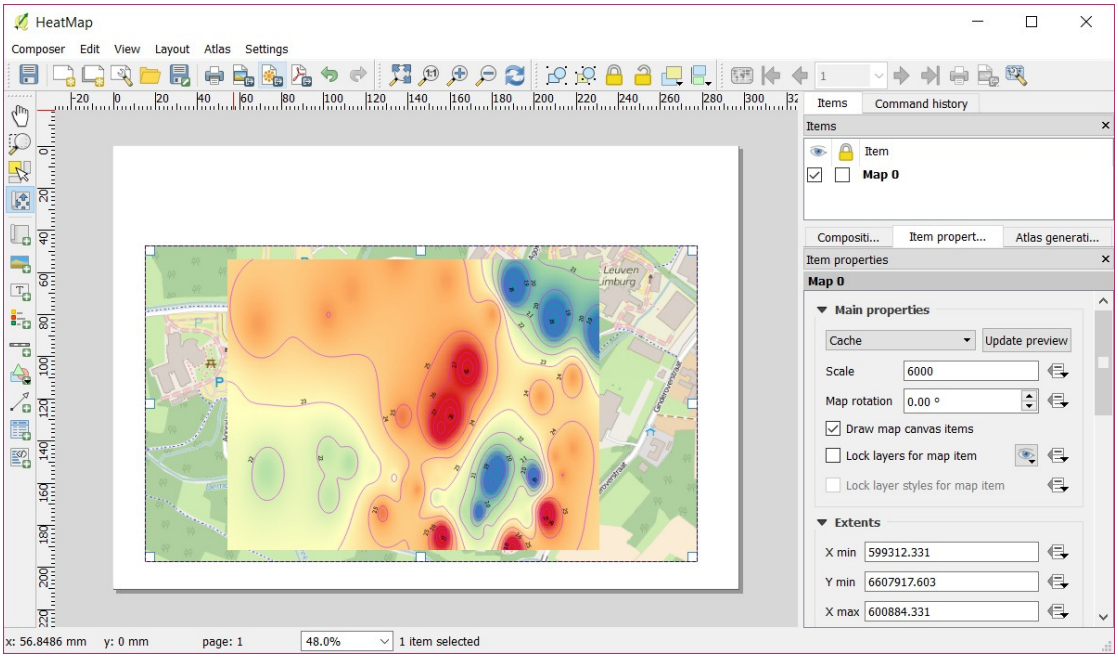
1. Create a new print composer
    - Go to Project New Print Composer (A).
    - Set HeatMap as the print composer title (B).

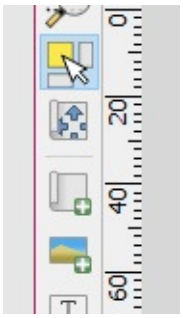


2. Add the created heat map
    - Click on the Add new map button (A).
    - Draw a rectangle on the presented page (B).

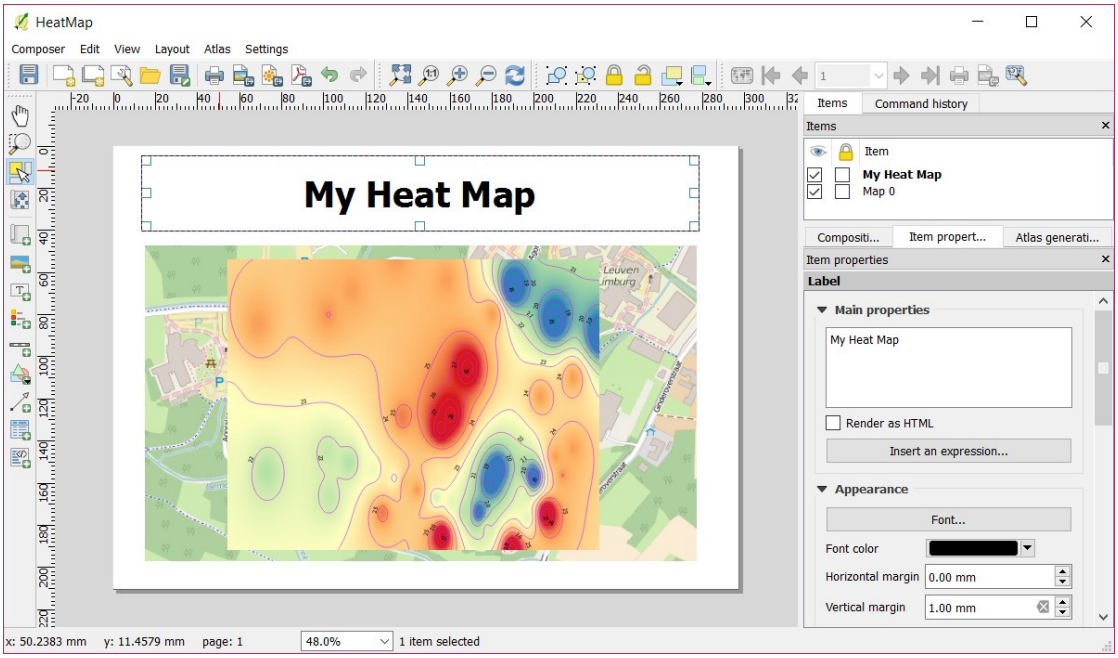


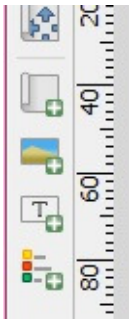
    - Click on the Move item content button (C).
    - Click on the map and drag it until the heat map is centered within the rectangle.
    - Set Scale to 6000 (D).



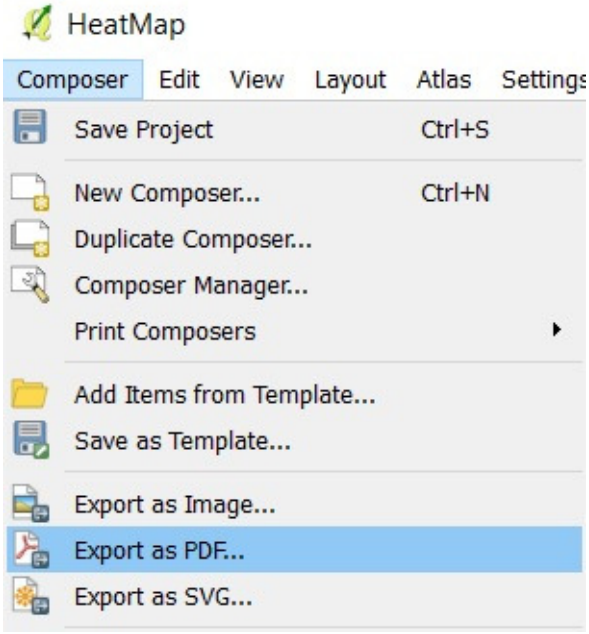


3. Add a title to the page
    - Click on the Add new label button (A).
    - Draw a rectangle on the page (B).
    - Set a title for the heat map (C).
    - Choose a font for the title (D).





4. Save your creation



   - Go to Composer Export as PDF.
   - Name your file MyHeatMap.pdf.

Contributing Projects

If you developed a senseBox project on your own and did document it, you may add it to this book!

Besides new project tutorials we also appreciate any improvements or corrections to the existing content!

Content

Such a documentation should provide an overview over the project as well as a step-by-step tutorial on how to build the project, photos and code may be included as well.

Make shure you provide all information required to reproduce your project with a senseBox:edu kit!

If you are familiar with GitHub, you may send your documentation to us through a Pull Request in the markdown format. Please refer to the guide below if you encounter issues.

Alternatively you can easily author the documentation in Office, and send it to us via mail. We will incorporate the file into this book.

Should it be applicable to your project, please use our project template (markdown, `odt` Office document) as a starting point.

Content License

Your contribution will be added under the same CC BY-SA 4.0 license as our content. This means, that the content may be freely adopted by others, as long as the authors name is provided with the content.

> Thank you for your contribution!

Writing the documentation in Markdown

For this book's management we use github.com and the tool GitBook, all content is written in markdown.

The documentation should be written in markdown, so the content can be directly included in the book. If you are not familiar with markdown, have a look at an explanation and syntax-explanation here.

To streamline the writing of markdown we recommend dedicated editors, such as the web-editor stackedit.

File structure

Place your documentation file in the directory `en/community_projects/` .

If you want to include additional resources, place them in a subfolder with the same name:

```
mobile-weatherstation.md
mobile-weatherstation/overview.jpg
mobile-weatherstation/mobile-weatherstation.ino
```

Filenames may not include any spaces!

Uploading the documentation

To provide us your documentation, you can insert it in the book yourself by submitting a pull request on GitHub. The source code of this book is hosted on GitHub. Fork this repository, add your content there, and create a new pull request.

In case you are unfamiliar with the GitHub process, have a look at a GitHub contribution guide.

Alternatively to working directly on GitHub, there is also a Gitbook.com editor, wich might simplify the process. However we didn't try that one.

If none of that works, just send us your contribution written in markdown via mail.

> Having issues? Mail our support!

Downloads

In this area various downloads and helpful resources regarding the senseBox are listed.

Libraries

A package of all the required libraries for the various sensors we use is provided here. Note that we use a customized `Ethernet.h` library, as the stock-library is not compatible with our ethernet shield.

Datasheets

Most manufacturers provide datasheets for their sensors and other components, which provide specifications and further insights into the inner workings of the devices:

Sensor	Description	Manufacturer	Download
BMP280	air-pressure sensor	Bosch	Datasheet
HC-SR04	supersonic distance-sensor	KT-Electronic	Datasheet
HDC1008	temperature- & humidity-sensor	Texas Instruments	Datasheet
TSL4531	digital lightintensity sensor	TAOS Texas Advanced Optoelectronics Solutions	Datasheet
VEML6070	UV-light sensor	VISHAY	Datasheet
GP2Y0A21YK0F	IR distance sensor	Sharp	Datasheet

Documentation as PDF

This book is also available as printable PDF!