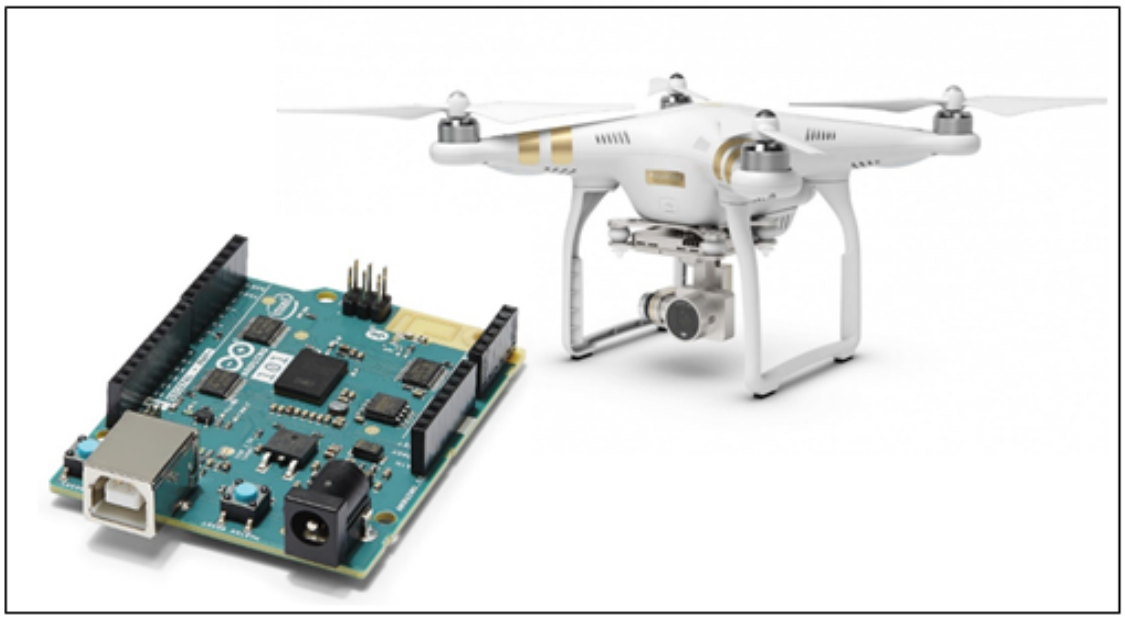
Make your first mini-satellite

If you learn to build an electronically controlled experiment, it will facilitate STEM challenges at school. And it helps you to participate in ESERO and ESA space education projects. In this training we produce a simple satellite that can measure temperature and air pressure. Then we launch it with a drone. For absolute beginners!



Belgium esero

FAB LAB KLEIN-BRABANT

Edition November 2017

Target group

Prim 0 1 2 3 4 5 6

Secondary 1 2 3 4 5 6

Make your first mini-satellite

Measurements on a flight – Arduino for absolute beginners

Fast facts

Target age of the students
14-18 years old

Type
Informative document with exercises and instructions

Hardware needed
- Arduino UNO microcontroller and accessoires
- Pressure sensor and temperature sensor
- Solder station
- LED's and resistors
- (Micro) SD card and datalog shield

Curriculum
Variable STEM curricular topics
(ideal for project education, flex STEM hours, ...)

Summary
How can you do measurements on an unmanned flight, like drones, miniature planes, weather balloons, rockets, ...)? You need basic electronics. In this course the absolute beginner can learn to measure pressure and temperature using an Arduino microcontroller. You will learn simple skills like soldering, controlling a LED, communicate between the laptop and the microcontroller, adapt a program code, and save measurement data.
These skills are an important base to participate in STEM projects like ASGARD (Balloons for science in Belgium), CanSat (33cl sat in a rocket) or Bifrost (parabolic flights in Belgium). In these projects students are challenged to launch their selfmade experiment.

Colofon

Edition November 2017

Last update 9 nov 2017

AUTHORS

ESERO Belgium

- Resource content and layout
- Training for teachers: organisation
- Author: Pieter Mestdagh

FabLab Klein-Brabant

- Additions
- Trainer in the ESERO teacher training : Davy Van den Bergh

Your opinion is important

ESERO Belgium resources are offered online in dynamic form. Every useful feedback will lead to the publication of an adapted version on www.esero.be (Nederlandstalig).
You can help future users by sending your remarks or additions by email (www.esero.be > NL > contact).
If you add relevant parts, your name will be put in the authors list.

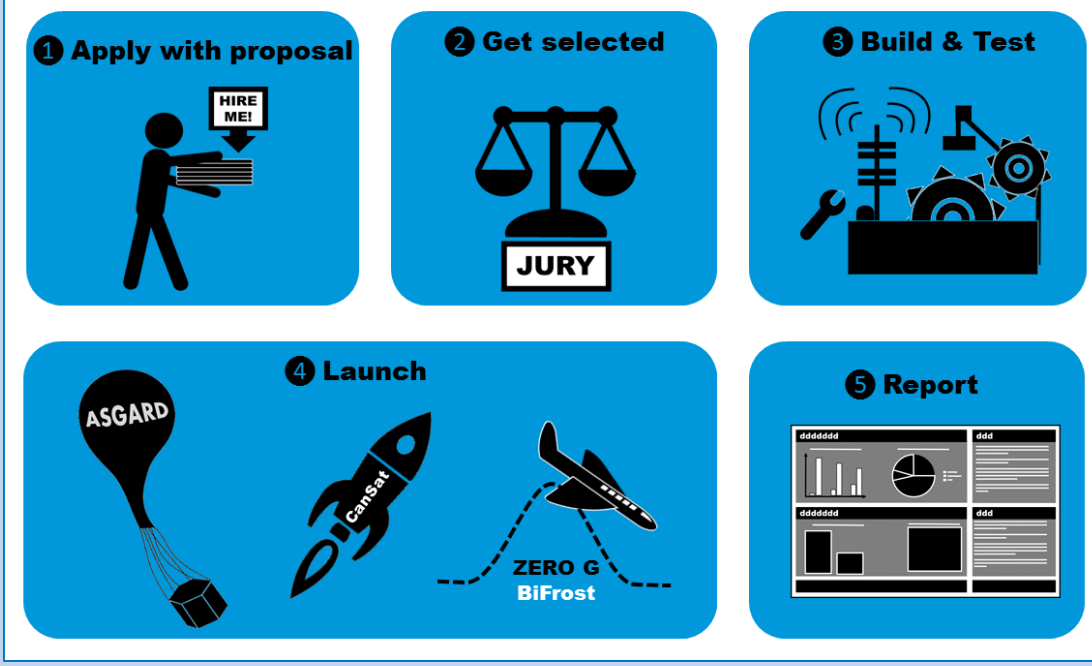
Content

1 Build your first satellite

1a A space mission at school:

Each year, Belgian and non-Belgian schools can participate in STEM projects, challenging students to launch their own experiment in an educative space mission. The pattern is the same in all of the projects:

1) **Apply**: The students think of an experiment (brainstorming), and write a description of their experiment in a document. Each student team needs to be guided by a teacher.
2) **Selection**: A jury of experts selects as many experiments (and teams) as possible. The jury is not necessarily searching for the 'best' proposals. The learning experience is the main focus of the projects.
3) **Build the experiment**: Once selected, the students will build their self-invented experiment and test it. A hard but educationally very rewarding phase.
4) **Launch**: The launches we provide in the ESERO Belgium projects are : flight with a weather balloon, small rocket flight, parabolic flight (zero gravity).
5) **Reporting**: Zowel tijdens als na het projectverloop moeten de teams rapporteren over hun werk, los van he al dan niet bekomen van zinvol resultaat.



The five phases of participation in the STEM projects ASGARD, Bifrost or CanSat. The five phases are spread over one school year. Source: ESERO BE.

Three different "space missions"

The table below contains an overview of the three STEM projects offered to Belgian schools. For more information: go to www.esero.be (> projects).

Project	Description	When	Teams
ASGARD	Your experiment reaches a height of +/- 30 km. It is above the ozone layer. The conditions are similar as on the surface of Mars.	Each school year (apply before Nov 11th)	• Max 5 students • 12-20 yo • 1 teacher (max 2) • Open for all secondary schools of the world
Bifrost	You put your experiment (and yourself) in a small air plane. It flies multiple paraboles. Each parabole you will be in weightlessness for about 20 seconds.	Every two years (next edition: 2018-2019)	• 16-20 yo • 1 teacher • For Belgian schools
CanSat Belgium	You launch your 33cl satellite with a small rocket. The altitude of the flight is 1 to 2 km. You need to design a way to safely land and recover the satellite. The data have to be sent to a ground station during the flight.	Every two years (next edition: 2017-2018)	• 18 teams (12 cansats are launched) • 4 to 6 students • 16-20 yo • 1 or 2 teachers • For Belgian schools
CanSat Europe	The national winners meet somewhere in Europe to launch their cansats in a European competition.	End of each school year. (Belgian participation every two years)	• 1 team per country • 4 to 6 students • 14-20 jaar • 1 leraar

Brief overview of the 3 STEM projects simulating a space mission. The 3 Belgian space projects are sometimes abreviated as 'the ABC projects' (initial letters). Source: ESERO BE.

Good to know

- ASGARD is not a competition. Once selected, the teams share experiences and knowledge to maximalize the succes of each experiment.
- ASGARD is a Belgian project with mainly Belgian school teams, but it is open for schools all over the world.
- CanSat is organized all over Europe. Their is a yearly European competition, organizzed by ESA. Each country can send 1 national winner.
- English is the language for all communications and reporting for ASGARD and CanSat.
- Bifrost is only open for Belgian schools. The teams are allowed to communicate and report in their own language (Flemish, French, German of English).
- Every year two teams from primary education are allowed join the ASGARD flight with their own experiment.

Elektronically controlled experiments for absolute beginners

ASGARD and Bifrost have some non-electronic experiments on board every year. However, most student teams use a simple microcontroller to build their experiment.

ESERO Belgium offers a training in which teachers have a first experience in using the Arduino microcontroller.

Make your first mini-satellite

In this training you will learn to make your first satellite that can measure the temperature and the air pressure. At the end of the training we put all the teachers satellite on a drone to fly. After the flight, the data are put in a graph to make a reconstruction of the flight.
This training wants to stimulate absolute beginners to apply for the ESERO STEM projects ASGARD, Bifrost and CanSat.

This document is an extensive guide for the training. Of course, this document can also serve as a support for any other electronically controlled project at school.

1b Components of a satellite (payload, bus)

You are producing a satellite

Strictly spoken you are not making a satellite in this training. A satellite is a smaller object in orbit around a bigger celestial body. We know natural satellites (like moons) and artificial ones. The latter are human made devices circling around the Earth or around other bodies. Still, the device you will be making is still calledd a satellite in the training, because of the following similarities:

- What you are producing is the “**payload**”. The drone flying this payload is called the “**bus**”. These words are explained later.
- Your **measurements** need to run completely **autonomous** once the payload is flying. Everything has to be prepared and programmed beforehand.

- A satellite works autonomously, but is usually partly controlled via the "**ground control**". The ground control is also used to receive the measuring data. In our training, the ground control is represented by the remote control used by the drone pilot. The data reception is not represented, as we recover the payload and its data after the flight (except when you have chosen to send the data during the flight with radio communication – not included in this training)

Payload

The payload is the part of the satellite by which measurements are made and experiments are performed. It is also called the “useful cargo”.

In this training the payload contains these components

- The micorcontroller (Arduino UNO)
- Sensors for temperature and pressure
- The memory card or the data transmitter
- Connection cables
- Resistors and LEDs

They are explained one by one below.

Bus

A **bus** (in electronics) is a medium exchanging multiple electronic signals.It forms a network connecting several components or devices.

In satellites, the **bus** is the total of subsytems that allow the satellite to function correctly, and to stay in a correct orbit. In the table below we compare general satellite subsystems with the subsystems of our own drone ‘satellite’.

General satellite bus components	Our drone satellite components
The spacecraft body structure	The drone and the harness carrying the payload
Communication mechanisms to talk with ground control	Communication mechanism between the dorne and the pilot’s remote control
Navigation systems and telemetry	Altitude meter and GPS of the drone
Motors and fuel cells	Motors and batteries of the drone
Temperature control systems	Not needed
Energy supply (fuel or solar panels)	Battery (power bank) that we add to the payload

2 Payload

All components needed to produce the payload in this training are explained extensively below. First read this part, to make sure you know them well and you know how they work. The next step will be integrating the components in the Arduino circuits.

List of components that you get when you follow the ESERO BE training:

Category	Components
Microcontroller	• Arduino UNO board
Sensors	• TMP36: temperature sensor • BMP280: sensor for pressure, temperature and altitude
Accessoires	• USB kabel A/B • Colored Jumper cables • Resistors 10 K and 100 Ohm • LEDs R470 green, orange, red • Tactile button (switch) 12 mm • Breadboard medium size • Stacking headers for Arduino R3
Data logging	• DataLog Shield with RTC 1307 • Micro SD card 16 GB

Power supply

During the training you will connect the Arduino with your computer. Then the microcontroller will be supplied with power via the USB connection.
During the flight ESERO BE will supply an external battery to feed the Arduinos. A power bank or battery is not delivered in your personal ESERO training equipment.

2a Peripheric components of the payload

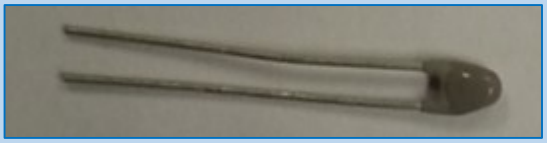
TEMPERATURE SENSOR

In the training we will use a specific temperature sensor, called TMP36. But it is also interesting to get to know another frequently used temperature sensor: the NTC.

The NTC thermistor

NTC stands for ”negative temperature coefficient”. It’s a logic name: when the temperature rises, then the electric resistance of this thermistor will decrease. When the temperature drops, the resistance of the sensor will increase. As a consequence tha Arduino will register a changed voltage over this sensor. The change in the voltage can be calculated to know the change in resistance, and finally the original change in temperature.

The NTC thermistor is very simple. It doesn’t perform internal calculations or data processing. Like a common resistor, it has only two ‘legs’.



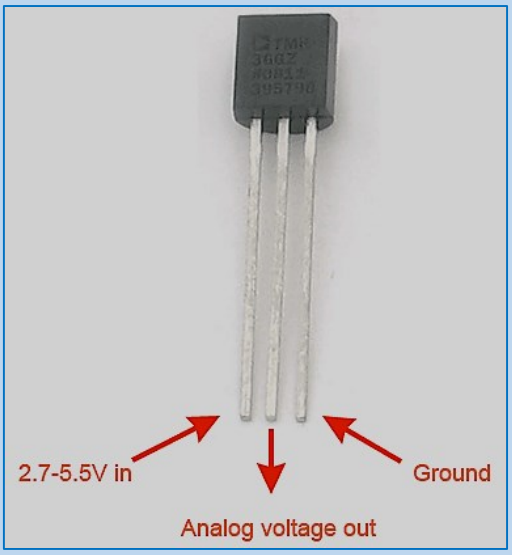
NTC thermistor. Source: ESA/ESERO working group Space Robotics.

The word **thermistor** is a contraction of the words “thermo” and “resistor”. The NTC is an electrical resistor, and the exact resistance value depends on the temperature.

The TMP36 sensor: introduction

Most of the contemporary sensors contain more then only the active part that is resposible for the variable output (like for example the temperature sensitive resistor). Many sensors process data internally using a set of microtransistors, making the output better adapted to the desired measuring value.

This is also the case for the TMP temperature sensors.



The TMP36 sensor. Source: Adafruit.

The TMP36 sensor: functioning

If the temperature increases, then the voltage over a diode will decrease with a known quantity. This is the principle that is used in a TMP sensor to register changes in temperatures. But instead of a diode the TMP uses a transistor. For varying temperatures, you get varying voltages over the basis and the emitter os such transistor. The variations in voltage is amplified to allow the measurements.

The calculations needed to convert amplified voltage variations into the corresponding temperature variations all happen inside the sensor. Consequently, the output signal can

directly be used to get temperature values, using a simple formula. The use of this sensor is therefore very easy, and the user doesn't have to calibrate.

There is a microchip inside the sensor. Therefore you should take care not to put it close to electrical fields or static electricity.

The TMP36 sensor: function limits

For any sensor you want to use, you should always verify if the measurements conditions fit within the limits proper to the sensor. These limits are specified by the manufacturer in an online datasheet. Some important 'stats' proper to the TMP36 sensor are listed below.

Stat	Limits
Dimensions	0.2" x 0.2" x 0.2"
Temperature range	-40°C to 150°C
Error rate	1°C (within the temperature range)
Price	€ 1,50 - € 2,00
Output range	0.1V (-40°C) to 2.0V (150°C) but accuracy decreases after 125°C
Power supply	2.7V to 5.5V only, 0.05 mA current draw

Note that the output range (voltage) only has positive values. This facilitates the data processing.

Tip:
Always consult the official datasheet. Don't use any info sheet about the sensor on the internet. Everyone can publish a self made info sheet, and the accuracy is not guaranteed if it is not published by the manufacturer himself.

The TMP36 sensor: How to use?

- Conect the left leg with the Arduino power supply pin (2,7-5,5V).
- Connect the right leg with the GND pin (earth, zero).
- Connect the middle leg with an anlogue Arduino pin, where you want the measuring data to enter.
- Perform the measurement (next chapter). You will get a series of output values V_{out} in mV (milliVolt).
- Apply the temperature formula on the output values. The results express the temperature in °C.

$$\text{Temperature} = (V_{out} - 500) / 10$$

Remark
Sometimes the output of the temperature sensor can be meaningless values, when multiple sensors are connected to the system. In that case, there was an unwanted interaction between the TMP sensor and another. You can avoid this by deactivating the code and provide a delay after every measurement.

COMMUNICATION WITH SENSORS VIA I2C OR SPI

Before explaining the sensor, we need to learn something about the communication that takes place between a microprocessor and the connected components. There are several types of communication systems or so-called communication interfaces. In this training we use the following systems:

	I2C bus	SPI bus
Main characteristics	1 master 1 slave 2 connection pins needed	1 master 1 or more slaves (CS) 4 connection pins needed
Advantages Disadvantages	Simple (2 pins) Slow for a lot of data Cheap Higher energy use	A bit more complex (4 pins) Faster No limiet on word size Low energy use
Use in this training	Temp sensor (TMP36) Pressure sensor (BMP280)	SD card

The difference between these two communication systems is in the way to recognize exchanged data bits. With other words: there are specific rules within the I2C bus and the SPI bus that define wether the processor should recognize a certain sequence of 0's and 1's as :

- the start of incoming data
- the end of incoming data
- the sensor data themselves
- etc.

In both systems there is a track or signal that serves as reference time signal. It is called the SCL (Serial Clock) signal. It forms a background of constant time pulses.

In both systems it is possible for a 'master' (the microcontroller) to communicate with one or more 'slaves' (sensors or other peripheral components). For example, the master can send out a command to a slave to start delivering data or to stop delivering data.

This training doesn't have the objective to explain the communication interfaces in detail. Below is provided a limited explanation that is useful for the ESERO training. Participants need to have a basic insight if they have to connect the sensors in a correct way to the Arduino microprocessor.

I2C bus

I²C = **Inter-Integrated Circuit** (pronunciation: i square c)

What?
A certain unique combination of the SCL signal and the data signal is recognized by the microprocessor as a START of registering incoming data. Just after this unique combination of sequences the device will start receiving and registering sensor data.

Another unique combination of SCL signal and data signal is a sign for STOP. When this sequence combination is detected, the microprocessor will stop reading data, and the sensor will stop delivering them.

The exchange of start-stop-commands and of measuring data between the microcontroller and the sensor all happens over 1 line. This line is called SDA, and carries signals in both directions. But the data can not simultanuously be transferred from master to slave ánd back.

In the I2C communication, only two pins are used for data transfer. And both can transfer bits in both directions:

- The clock signal **SCL** (Serial CLock)
- The data signal **SDA** (Serial DAta)

Apart from the data signals, you also need 2 pins for the power supply:
- **5V in** (source of voltage)
- **GND** (earth, zero)

Master
The microporcessor is called the master. It controls the transferred bits on the I2C bus. The processor will give the start and stop signal and will send out the SCL signal.

Slave
Every sensor or other peripheral component connecter to the porcessor is called a slave. You could say that the sensor obeys to the processor (master). The slave also delivers the data to its master, the processor.

SPI bus

SPI = **Serial Peripheral Interface**

What?
In this communication system there is a simultanuous transfer from the master to the slave and back. You need four lines for data transfer.
The communication starts with the defining of the clock speed, using the signal on the SCL line. Following on that, another line (CS) is used by the microprocessor (master) to activate one of the connected sensors (slaves), and to start reading incoming data. This continues automatically until the master (again using the CS line) gives the command to stop reading data.

Four pins are needed for data transfer:
- **SCL** or **SCK** : serial clock
- **MOSI** : Master output slave input: on this line only data are transferred that are sent out by the master and received by the slave, not in the opposite direction.
- **MISO**: Master input slave output: on this line only data are transferred that are sent out by the slave and received by the master, not in the opposite direction.
- **CS**: Chip select (or SS Slave Select): This line is used to activate one of the slaves. When this line is on LOW (zero Volts), the activation will start. When the CS is on HIGH (5 Volts), then the slave will be deactivated (‘stop’ sign). Each slave should have a unique CS line.

Note that the sending out of a start-stop command by the master can happen simultanuously with the sending out of measuring data by the slave.

Additionally you need 2 extra pins for power supply of the sensor:

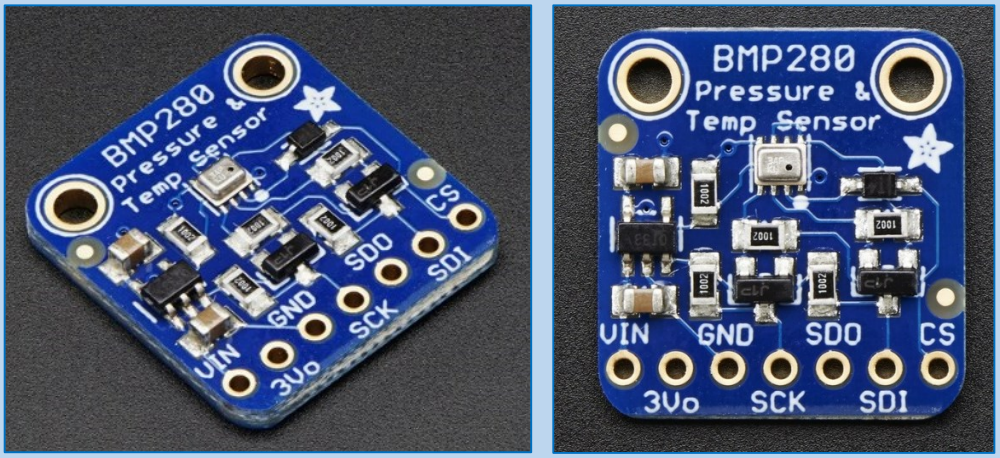
- **5V in** (source of voltage)
- **GND** (earth, zero)

PRESSURE SENSOR

BMP280: Introduction

Above you were introduced to a simple sensor with one function. However, in electronics it is a habit to limit the work as much as possible. Therefore sensors are often used with multiple functions at once.

The Bosch **BMP series** are sensors that can measure the temperature and air pressure at the same time, and immediately deduce the altitude (based on pressure values). The error range on the pressure measurement is about 1 hPa. The error range of the temperature is about 1°C. This makes the sensor a good tool to be used as a barometer and to know the altitude with an error range of about 1 meter.



The BMP280 sensor. Source: Adafruit.

BMP280: Available pins

There are 7 pins at the bottom of the sensor. The first three are power pins:

- **Vin**
  Here the power enters. This sensor uses 3,3 Volt DC. But you can also put 5V on this pin, because there is an internal voltage switcher.
- **3Vo**
  Here the power can leave the sensor with a voltage of 3,3V. So you can use this pin as a power source for something else, with a mximum current for all components of 100 mA. These pin can also stay unused.
- **GND**

The following 4 pins are called **'logic pins'**. Using the I2C communication interface, we only need 2 pins. For SPI communication (faster data transfer and less power consumption) we need 4 logic pins.

I2C communication:

- **SCK** = Serial ClocK
  This pin sends time pulses to the µcontroller. It should be connected to the Arduino SCL pin (I2C clock pin): the last pin in the row of digital pins. The time pulses on this line form the reference tim for the whole system. It is the time reference for the measuring data as well as for the execution of the coded program.
- **SDI** = Serial Data In/Out
  This pins sends out data from the sensor to the Arduino, as well as the opposite direction. It should be connected with the Arduino SDA pin: the last but one pin in the row of digital pins, this is the Arduino I2C data pin. On this line the measuring data are exchanged, as well as the commands sent by the µcontroller and the signals to activate or deactivate a slave component.

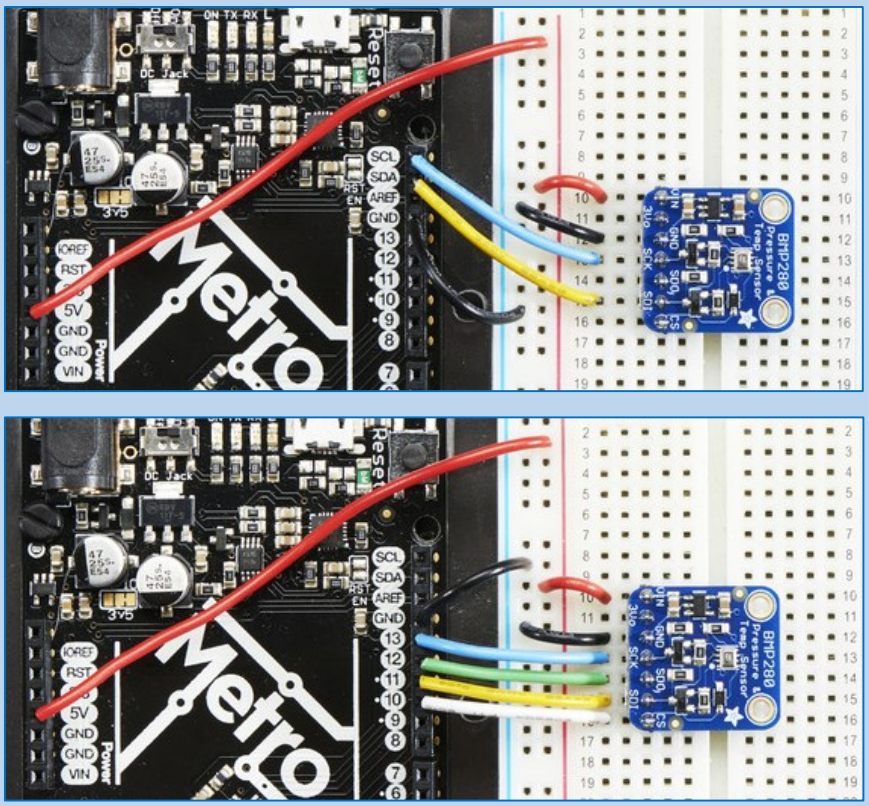
The other two pins will not be used.

SPI communication:

- **SCK** = Serial ClocK
  This pin sends time pulses to the µcontroller, just as in the I²C interface.
- **SDO** = Serial Data Out
  This pin sends data from the sensor towards the µcontroller (MISO = Master In Slave Out pin).
- **SDI** = Serial Data In
  This sensor pin receives data from the µcontroller (MOSI = Master Out Slave In pin).
- **CS** = Chip Select
  Every component that is connecter to the µcontroller needs to be connected to a unique CS pin (any digital i/o pin). They are used to determine which sensor has to be (de)activated on a given moment.

All of these four pins have to be connected with a unique digital pin (nrs 2-13) of the Arduino.

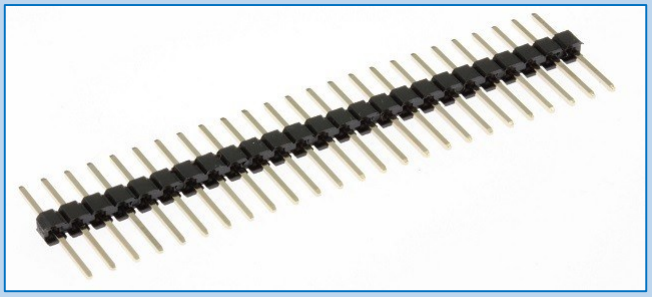
When you connect more then one sensor with the SPI interface, then all the SDO, SDI and SCK pins can be shared, eahc time on 1 Arduino pin. But the ssensors CS pins have to be connected each of them with a unique Arduino pin.



The BMP280 sensor connected to the Arduino using the I²C interface (above) and via SPI interface (below). Source: Adafruit.

Working with the BMP280

On the picture above you see the BMP sensor connected to Arduino by means of a breadboard. This is an easy way to make your circuits and test them before you finally solder all the components. To connect the sensor pins (small holes, female pins) with the breadboard holes, you can use a header : a series of male pins that all have the same distance towards each other and perfectly fit into the pin holes of any component.



Header. Source: Adafruit.

Having connected all the pins as above, the rest of the work is coding a program. You'll have to define in the code the correct function of each of the Arduino pins used, corresponding with the sensor pin that they are connected with. Then you'll have to specify how the measuring data will be read and communicated. We will learn this later in the training.

When you choose your own pressure sensor

In this training the sensor was already chosen. But for any STEM projecct at school, you'll probably have to search yourself what sensor to use. It is advised to pay attention for the following characteristics:

- **Sensitivity**
  What is the minimal pressure change that this senor can detect?
- **Response Time**
  How fats is this sensor?
- **Linearity**
  Do the output data have a linear relation with the air pressure (for the value range that you expect to measure)?
- **Value limits: Range**
  What are the minimal and maximal absolute pressure values that the sensor can measure?
- **Hysteresis**
  When the pressure decreases, a sensor can sometimes output slightly different absolute values then in the case of increasing pressure. This slight difference is called hysteresis. It is an indication of a certain internal slowliness by which the sensor is taking over the ambient value.

µPROCESSOR OR µCONTROLLER

The µcontroller is the most essential part of the Arduino UNO. This component has an internal memory. It is on this memory that the code will be written once you have written it and uploaded it to the Arduino.

The Arduino UNO is explained under chapter 2b.

BATTERY / POWER BANK

Energy for satellites

The energy for most of the real satellites is provided by solar panels containing photovoltaic cells placed in series. Actually, photovoltaic cells are inversed LEDs. When light is hitting the cells, a small electrical current is created. Usually the solar pannels charge a battery, and then this battery will feed the instruments of the payload. This is most certainly needed when a satellite is in a low Earth orbit. For most of these orbits, the satellte goes through the Earth's shadow (night side) in each rotation, and solar energy is not available for a while.

The energy consumption of a satellite has to be calculated in detail, so the manufacturer can provide the precise amounts of energy when it is needed.

However, in this training we will use a battery that we charge fully, so there will be plenty of energy available during the flight.

What battery do we use?

For the power supply of our payload we choose either a classic 9V battery or a power bank.

A classical 9 V battery with + and – poles, and a common powerbank to be charged using any USB connection. Source: ESERO BE..

Both are light and cheap batteries, that you can buy everywhere.

Good to know: 9V battery

- Not controlled electronically. It will always provide voltage (and current) when it is connected to any device.
- Slightly higher mass then most power banks.
- Has a smaller energy density (alkaline battery). That means that the payload will run out of power sonner then with a power bank. For thihs training, this is no problem. But for example during an ASGARD balloon flight, this is relevant difference.

Good to know: power bank

- Power banks are sometimes controlled electronically. For example, some power banks are programmed to give no current at all when the connected component is only consuming a very low amount of energy. This function is built in to avoid that the energy leaks away when the device is switched off. But it can also be a problem when we connect an Arduino UNO with just one sensor, because our payload is a very low energy consumer. Then the power bank might not provide any current.
- Slightly lower mass then a 9V battery.
- A higher energy density (Lithium batterij).
- Can be recharged many times using USB power supply.

RESISTORS

What?

This component has a name that defines its function. An electrical **resistor limits the passing through of an electrical current**, and causes locally an intentional decrease of the conductivity. The higher the resistor value (resistance), the more conductivity is reduced. Conductors like copper and aluminium have a very low resistance. Isolators like pvc or glass allow almost no current al all, so they have a very high resistance.
Resistors in electronics have a **predefined fixed value** somewhere between the resistance of a perfect conductor and a perfect isolator.
The letter for resistors as used on product lists and circuit schemes is: **R (of Resistance).**

The resistor letter R is sometimes replaced by the sign **Ω (ohm)**. A correct way to express is for example: R = 512 Ω. Ohm is the unit of resistance.

Common resistors (with legs) are marked with colored rings to show their resistance value, because they are too small to put letters on it.
They are usually not ESD sensitive (ESD = Electrostatic discharge). That's why they are not necessarely packed in small plastic bags. You can touch them as much as you want, without any damage or change of value.

Use

We need resistors to limit thee current in certain sensitive components or in the whole circuit or system. This is of huge importance for Arduino users. When the maximum allowed current is exceeded, then your µcontroller will be destroyed. Below we will learn how to avoid this.

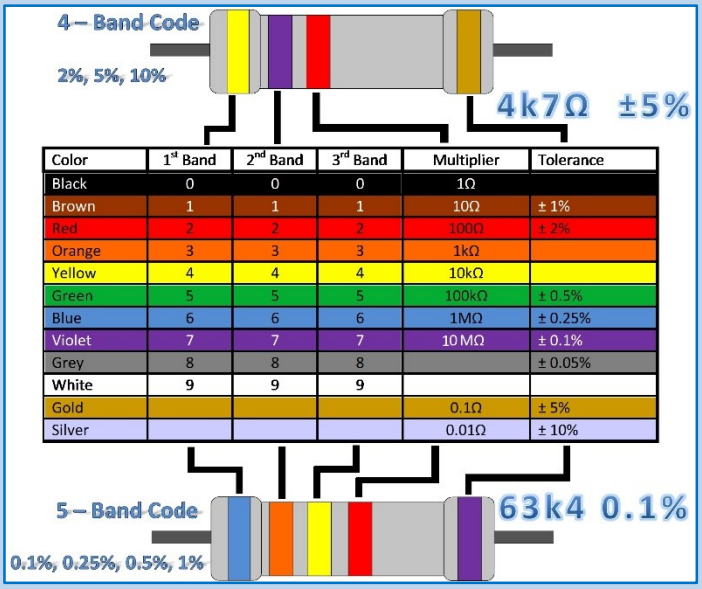
Color codes

resistors with four rings

- The first 2 rings determine the basic number.
  Example: first ring yellow, second ring purple → basic number 47
- The 3rd ring determines the multiplication factor, or how many zero's you should puut behind the basic number.
  Example: 3rd ring red → multiply with 10^2 (x100).
- The 4rd ring determines the error range.
  Example: 5th ring gold → the real value of the resitor (in Ohm) can have a difference of maximum 5% with the given value.

Our example can be summarized:
Yellow – purple – red – gold → 4700 Ω (+ or – 235 Ω) (or 4k7Ω +/– 5%)

The first ring is the one closest to the side leg, and the last ring is a bit wider then the others.



Color	1st Band	2nd Band	3rd Band	Multiplier	Tolerance
Black	0	0	0	1Ω	
Brown	1	1	1	10Ω	± 1%
Red	2	2	2	100Ω	± 2%
Orange	3	3	3	1kΩ	
Yellow	4	4	4	10kΩ	
Green	5	5	5	100kΩ	± 0.5%
Blue	6	6	6	1MΩ	± 0.25%
Violet	7	7	7	10 MΩ	± 0.1%
Grey	8	8	8		± 0.05%
White	9	9	9		
Gold				0.1Ω	± 5%
Silver				0.01Ω	± 10%

Significance of the color codes for resistors with 4 or 5 rings.

If a resitor has **5 rings** in stead of 4, then the rules change a little bit:

- The first 3 rings determine the basic number
- The 4th ring determines the muliplier
- The 5th ring determines the tolerance.

CABLES

Your ESERO kit will contain two kinds of cables essential for each Arduino project.

USB connection for Arduino UNO

The cable type : USB 2.0 A to B (male/male).
This cable is used to connect the Arduino with a computer. Most printers use the same cable.

When the Arduino is connected with the computer,a LED will light up. This is how we see that the µcontroller is power supplied.
The cable exchanges power and data (codes, input/output data).

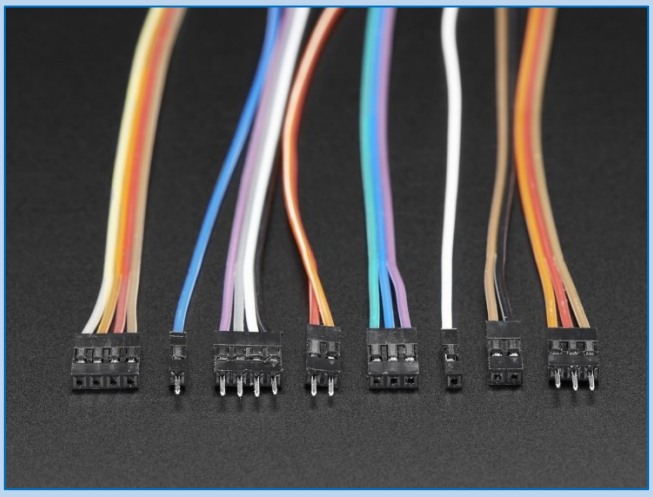
USB connection cable to connect the computer with the Arduino. Source: Sparkfun.

Colored set of jumper cables

The set of the colored thin cables (in our training they are male/male) are used to connect pins. They are called jumper cables.

When you are designing an electronical circuit, you will probably draw it first in a scheme. The many different colors of the jumper cables allow you to copy your scheme to a real hardware circuit in a non confusing way,using the breadboard.

In general the colors are used randomly, except for red (power supply + or V_{in}) and black (power supply – or GND).



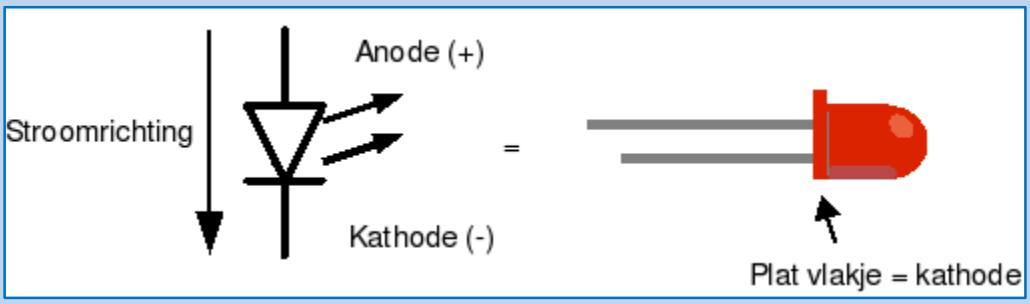
Flat jumper cables with male and female ends. Source: Adafruit.

LEDs

What is a LED?

LED is the abbreviation of **Light Emitting Diode**. Indeed the LED has the same characteristics of a diode: it only allows **current in one direction**. The current can only run from the anode (positive pole) to the kathode (negative pole). This is an important difference with other components, like a classical lamp, in which the polarity is of no importance.

Another difference between a LED and a classical lamp is that the current is not decreased in a LED, because it has no resistance. That is why we need to **decrease the current** ourselves by building in a **resistor** before or after the LED.



Symbol (left) and drawing (right) of a common LED.

You can identify the anode and kathode of a common LED as follows:

- The **anode** (the positive pole) is the longest leg.
- The **kathode** (the negative pole) is the shortest leg. If it is a rounded LED 5 mm or bigger, then the kathode is also marked by a small flattened side close to the leg.

Build a LED in the circuit

LEDs appear in many colours, forms, sizes and variations. Therefore it is not always so easy to identify the anode and kathode. Multicolor LED for example have more then two legs.

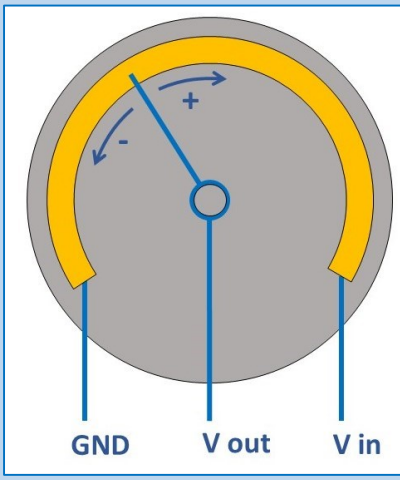
Every LED has to be accompanied by a resistor, placed in series, to avoid that the passing current would be too strong. The user has to calculate the needed value of resistance.

Each LED has a specific datasheet, showing two important values:

- U_{LED} is the **minimal voltage** needed to get the semiconductor function. At lower voltages there will be no light.
- The **maximal current** that can go through the LED without being burned.

POTENTIOMETER

A potentiometer or potmeter is a resistor that can be adapted by making a rotating or dragging movement. They are used to decrease a voltage gradually as desired.



Scheme of a rotating potentiometer.
Source: ESERO BE.

SWITCH

In this training we will use a **tactile switch** that you put on and off by pushin it once. It has a dimension of 6 mm and four legs of 2,5 mm. We will use it to switch the Arduino on and off. It will be combined with resistors to protect our Arduino against strong currents.

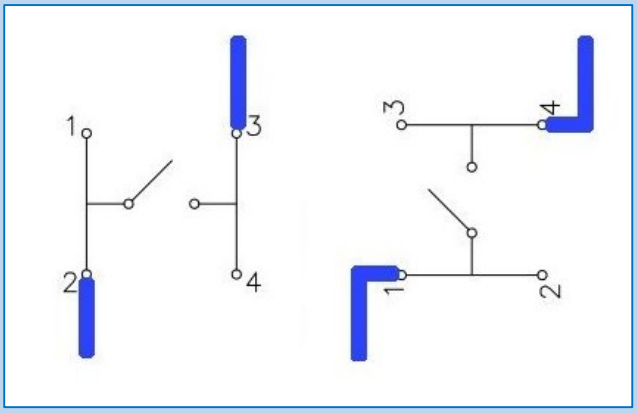


Foto and scheme of a tactile switch of MC Hobby Shop. Building the switch into the circuit, one needs to use the diagonally opposite legs.
Source: Shop MC Hobby Belgium.

The 4 legs fit perfectly in the pins of an Arduino or other components, like a breadboard.

2b Arduino

TERMINOLOGY

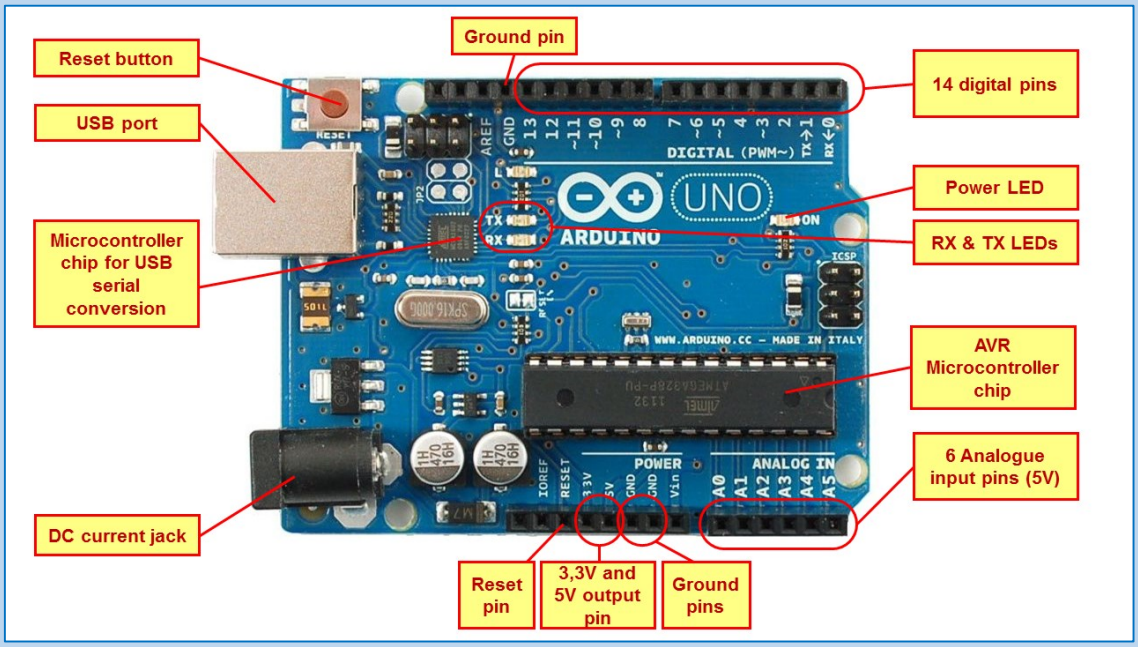
Arduino and other microcontrollers have their own terminology. The most important words are listed and explained here:

Term	Explanation
Master/Slave	Usually thee microcontroller is seen as the master. It is the central device sending commands to the peripheral components. The latter receive the commands and execute them, and they send back 'their work'.
Sketch	This is the total code forming an Arduino program. It is written by the user in the Arduino IDE window. After finishing the code, the user will upload it to the Arduino, which consequently executes it. The code language is C. You could consider the word "sketch" as an Arduino word for "file". It is saved with the extension .ino (formerly: .pde).
Library	This is a piece of code with a certain function that you can find online. In an Arduino sketch, you can refer to a library at the start of the sketch, using the command "`include`". Libraries are used to avoid "inventng the wheel" over and over agian by every individual user of certain (frequently used) functions, or to facilitate thee use of specific components. Usually the libraries get a name that refers to its function. A library has to be stored on your laptop to be able to refer to it.
Shield	This is a plastic plate (Printed Circuit Board or PCB) carrying built-in connections and components (or components added by the user). It can be connected to the Arduino board by means of a series of parallel pins that perfectly fit on the Arduino pins. This way you can build an Arduino with one or more 'floors' (etages) carying multiple functions.
Compiling	The sketch written by the user in the Arduino IDE window uses the C programming language. We call this the 'human readable' version of the program. The Arduino software changes the code when it is uploaded to the Arduino microchip in a 'machine readbale' translation. Then the Arduino chip is capable of executing the program. The translating-and-uploading proces is called compiling.
Hex code	The C code you use for programming is the human readable version of the program. The translation to a 'machine readable' version happens in the process of compiling. The translated code itself is called a 'hex code'.
Syntax	A human language has grammar and leestekens to allow a correct and clear communication. This is also the case for thee programming language C++, but here it is called syntax. It is a set of rules specific for the programming language.

HARDWARE ARDUINO UNO

The Arduino products are '**open source**'. This means anyone can make variattions on the Arduino boards and sell them. That is perfectly legal. Many clones are just as good as the original.

What **comonents** can we find on the Arduino UNO board?



Main components of the Arduino UNO board. Source: ESERO BE.

AVR Microcontroller chip or microprocessor

This is the brain of the Arduino. It also is the place where the codes are copied and where it runs.

The chip has a flash memory of 32 kbytes.

Microcontroller chip for USB serial conversion

The communication between the computer (supported by the IDE software) and the Arduino is only possible via this chip for USB serial conversion.

USB port

This is a connection for the USB cable that connects the Arduino with the computer. This connection serves as power supply and for data transfer.
An alternative power supply is possible via the DC power jack.

DC power jack

A power source can be connected here, like for example a default 9V battery. This way, the Arduino can get power without being connected to a computer. This is needed during the flight. The connected battery has to deliver a voltage between 7V and 12V.
An alternative can be to connect a power bank using the USB plug.

6 analogue input pins

Analogue data of peripheral components (sensors) have to enter here (max 5V).

14 digital pins

These pins can receive data from and send data to any connected digital component. In both directions, variations in voltages are used of maximum 5V. Binary bits (0 and 1, LOW or HIGH) are represented by 0V or 5V.

Digital pins with intermediate values
The digital pins with the sign **~** (pins **3, 5, 6, 9, 10, 11**) can be used for lower voltage values then 5V. A pin without this sign will only have 0V or 5V, no other values.

But how is it possible that a digital pins works with intermediate values? Actually, it is switching between the digital values (0V, 5V) in a very high frequency, thousands of times in 1 second. By changing the duration of 0V intervals and 5V intervals, the average ooutput signal will be a value between those two extremes. For example, if the 0V intervals are exactly the same as the 5V intervals, then it will measure an (average) value of 2,5V.

So the time is an important reference. The ‘crystal’ (metal colored ellips component) on the Arduino board is the component that makes it run on a constant fixed clock speed.

Pin 0 and 1: TX and RX
T stands for Transmit, R stands for Receive. They can be use for serial communication between the Aduino and another device. When data pass this channel, then the corresponding LEDs (TX and RX) will start blinking. This can be interesting information for trouble solving when you want to check if the Arduino is actually sending or recieving data.

The input mode is the default mode of a digital pin. When you want to use them as output pin, then this has to be specified in the code.

3,3V and 5V output pins

This pins serve as power sources for the Arduino and all connected components. They have a voltage of either 3,3V or 5V.

Reset knop and Reset pin

Pushing this button will make the Arduino stop the running program and rerun the program all from the beginning. The code on the chip will not be deleted.
You will have exactly the same reaction when 0V is put on the reset pin.

Ground pins

The GND pins give acces to the lowest voltage of the system. They are considered as earth.

To sink and source

You can use the Arduino for '**sourcing**'. This means that the Arduino is used as a power source for a connected component. The total current that you can source with an Arduino UNO is **200 mA**. (max 40 mA per pin).

When an external device is acting as a power supply, the Arduino will receive the current and let in end in the GND pin. This is called '**sinking**'. The Arduino has 2 GND pins for this. That means that we can send a total current through the Arduino of 2 x 200 mA, or **400 mA**. (max 40 mA per pin).

Power LED

The power LED is a kind of on/off LED. It shows us if there is a voltage on the system or not.

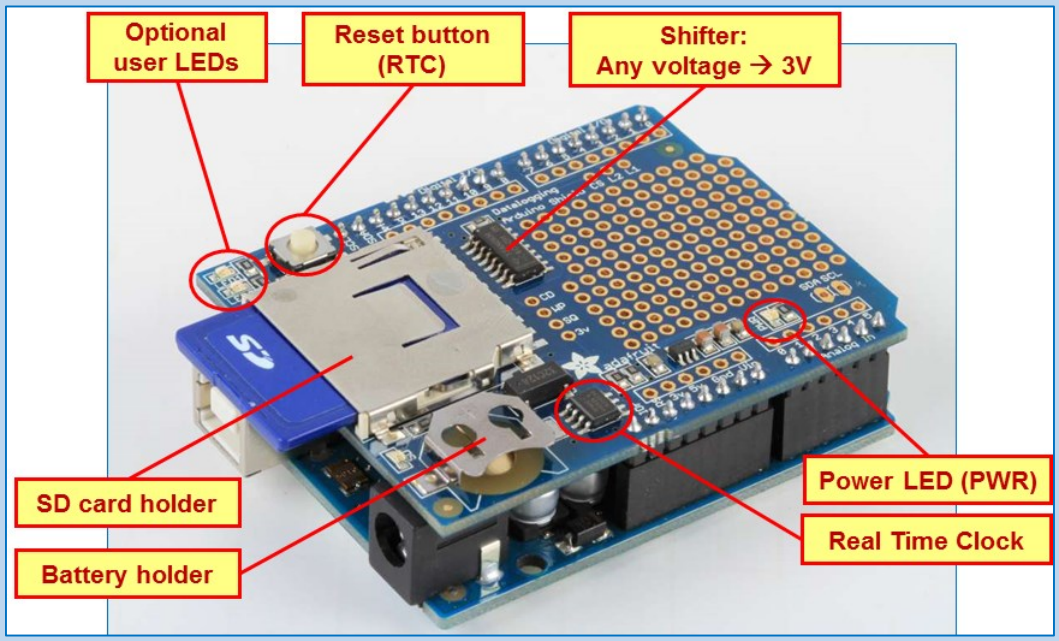
SHIELD CONNECTIONS – SD CARD DATALOG SHIELD

What is a shield?

You have noticed that there is a long black plastic block with holes attached to the pins. All Arduino boards (also the non-official copies) have the same layout. That's why we can find all kinds of shields that fit perfectly well on the Arduino boards.
A shield is a kind of extra floor (etage). It is used for connecting several devices or components, for example: a touchscreen, an SD card reader, a Sim card reader, motors, etc.

Datalog shield

In this training you get a datalog shield that has an SD Card reader. We need it to save the measuring data. The Arduino chip has a small internal memory of a few hundreds bytes. This is not enough to store all of our data on the flight, so we need an extra memory.



Shield for SD card with real time clock (RTC). Here the SD card is already inserted and the shield is soldered on an Arduino UNO board. Source: ESERO BE.

- **RTC:**
  The SD card shield in this training has its own Real Time Clock, that will supply the data on the card with a time reference.
- **Battery holder:**
  You can put a small battery in the shield allowing the clock to run for many years autonomous, without being connected to an Arduino or another external power source.
- **SD card holder:**
  Every SD/MMC memory card can be inserted here. When using a micro SD card, you need to have a plastic adapter to make it fit in the slot.
- **Voltage shifter:**
  An SD card needs to be feeded with a voltage of 3 Volts. Higher voltage can damage the card and the data. This voltage shifter assures a constant 3V voltage on the card, independant of the voltage entering from the Arduino.
- **Optional user LEDs:**
  The Arduino data output pins can be connected with the L1 and/or L2. This way, you will be able to see when the signal on these pins are HIGH (LED on) or LOW (LED off).
- **Power LED (PWR):**
  This LED shows you if the datalog shield is on or off (if there is a voltage on the shield or not).

The use of the datalog shield will be explained later under “Save data on the SD card”.

CURRENT CONTROL

Limitng the current

On the Arduino datasheet we find some very important **maximum current** values.

Absolute Maximum Ratings - the point where damage will start to happen
- DC Current per I/O Pin ..... 40.0 mA
- DC Current VCC and GND Pins..... 200.0 mA

These maximum currents have to be respected by all means. Otherwise the microcontroller will be damaged.
- There is 1 VCC pin. So the Arduino can send out a maximum current of 200 mA.
- There are 2 GND pins. So the Arduino can receive a maximum current of 400 mA.

To make sure the maximum values are not passed, we need to put resistors in the circuit, set in serial with the power consumer (like a LED, a sensor, etc.). Such a circuit with included resistors to control the voltage is called a **voltage divider**.

How to calculate the ideal resistor to be added to a voltage divider circuit?

To calculate the resistor value we need, we will use the **Ohm's law**:

$$U = I \times R$$

U is the voltage,
I is the current, in Ampère,
R is the resistance in Ohm.

This formula can give you the voltage over a known resistance when you send a known current through it.

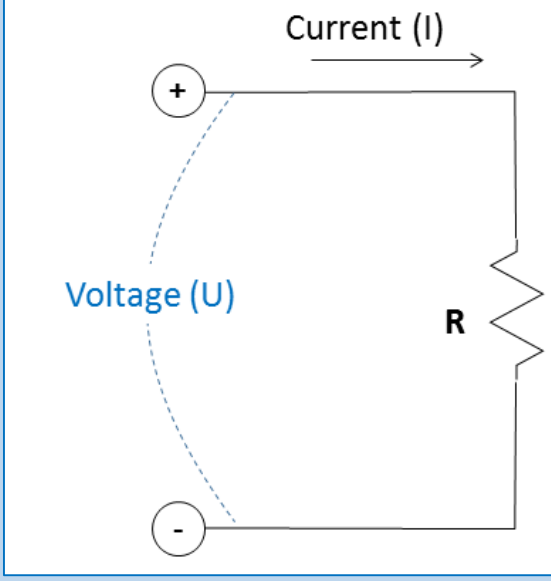


Illustration of the Ohm's law in a circuit. circuit Source: ESERO BE.

What if you want to know the resistance, rather then the voltage? You can change the formula this way:

I = U / R and **R = U / I**

The previous illustration supported the Ohm's law, but we didn't include a power consumer yet: The **LED**. It will be put in serial with the resistance.

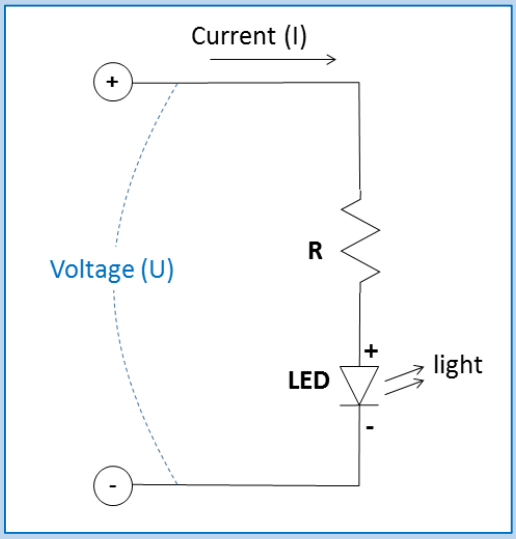


Illustration of a circuit with a LED and a resistor in serial.
Source: ESERO BE.

The **order** of the LED and the resistor are not important, but the LED has to be put into the circuit with the correct polarity. The + pole of the LED has to be connected to the + pole of the power source. The same counts for the – pole.

Furthermore, the **power source** has to deliver a **higher voltage** then the nominal voltage interval of the LED. If not, then the current will not flow. With nominal voltage, we mean the actual voltage that is constantly needed over the LED to emit light.

In this **example** we will study a red LED wit a nominal voltage of 1,9 Volts. So, if we have a power source with a voltage of 5V, then we should see light. Now we decide to want a current of 20 mA. How do we calculate the correct resistance?

One would think that you simply have to fill in the voltage (5V) and the current (0,02A) in the Ohm's law. This is not enough. We also have to take into account the **nominal voltage over the LED**, called U_{LED}. The latter has to be subtracted from the battery voltage **U+** to know the voltage over the resistor. The formula will be like this:

$$R = (U+ - U_{led}) / I$$

You can fill in the values now:

$$R = (5 – 1,9) / 0,02 = 3,1 / 0,02 = 155 \text{ ohm}$$

So we will finally take a resistor of **180 Ohm** as to add to our cicuit to get the right **voltage divider**. The 180 ohm is a is a default resistor on the market. Because this resistance is a bit

higher then the calculated 155 ohm, the final current through the LED will be a bit lower. This is ok, because we don't need to have a very specific lihgt intensity emitted by the LED. We want to control the current as much as possible. So it is better to **know the exact current** when we use the 180 ohm resistor in stead of the calculated 155 ohm. Again, we'll have to subtract the nominal LED voltage from the battery voltage:

$$I = (U+ - U_{led}) / R = (5 – 1,9) / 180 = 17,22 \text{ mA}$$

Now you see that the actual current is a bit lower then the desired value of 20 mA. In this case, it doesn't really matter. The difference in light intensity will not be relevant.

BE AWARE !

The good news is that you are now capable to calculate the resistor for a voltage divider for a LED. But there is still an important detail you need to learn.

If the voltage over the resistor **$(U+ - U_{led})$** becomes **very small** compared to the battery voltage, then a very small change in battery voltage or in nominal voltage over the LED can cause a big change in current value.

Suppose you have a blue LED with a nominal voltage U_{led} of 3,6 volts, and a battery voltage of 3,7 volts. You want to end up with a current of 20 mA. Then you will calculate the resistor needed:

$$R = (3,7 - 3,6) / 0,02 = 5 \text{ ohm}$$

So you will take a resistor of 5,6 ohm. The current will be (3,7 - 3,6) / 5,6 = 18 mA. That is fine ... or not?

Let's see what happens when the voltage **varies only some tenths of a volt**. If the battery voltage would be 3,5 volt (0,2 volt under the expected voltage), then we are already under the nominal voltage over the LED. It will **hardly emit any light**. But it becomes worse when the battery voltage is 0,2 volt higher then the default value. The current will theen be (3,9 - 3,6) / 5,6 = 54 mA.

This more then most of the LED can take AND more important: it is **above the maximal current** allowed on an Arduino i/o pin. You will probably see the LED light up very strongly, and then go out forever ...

Combine resistors

If you can't find a resistor with a certain resistance value, then you can put resistors in serial or in parallel into the circuit. The total resistance is to be calculated as follows:

- Resistors in serial: $R_{total} = R_1 + R_2 + R_3...$
- Resistors in parallel: $1/R_{total} = 1/R_1 + 1/R_2 + 1/R_3 + ...$

Pull up resistor

In any normal environment, we are surrounded with electrical fields and static electricity. They are produced by the many devices (like cell phones) around us and by general

movements of people and things. These electrical field variations find their way in the Arduino pins as small currents.
These small currents form a **noise** on the system. This way, we can detect signals even when the power supply is off. In other words : when we have a main switch to cut off the power, some pins will still give a signal. These pins are called **floating pins**: there voltage is 'floating' between min and max, between LOW and HIGH.

To avoid this noise to disturb our measurements, we will put a **circuit sideway to the GND** between the battery and the i/o pin. Noise currents can then flow away in this sideway. Of course we need to put a resistor on this GND sideway to protect our battery against too much leaking. The concept is explained below.

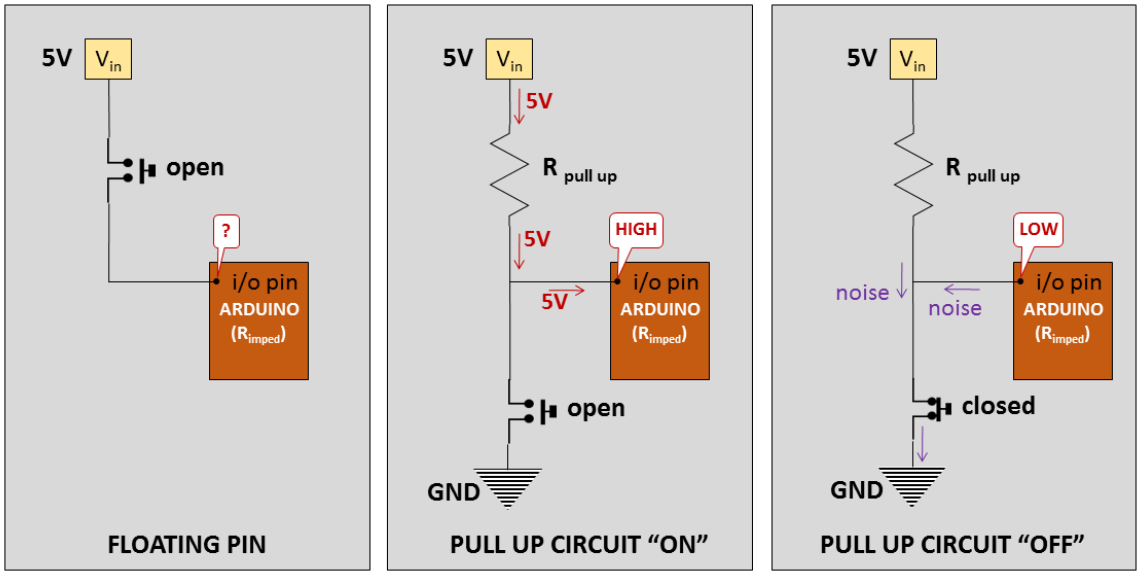


Illustration of a floating pin (left), an i/opin with pull up resistor that is "on" (middle), and an i/o pin with pull up resistor that is "off" (right). The i/o pin also has an internal resistor (R_{imped}, not shown). This internal resistor determines the impedance of the Arduino. R_{imped} is bigger then $R_{pull\ up}$. Source: ESERO BE.

- When the on/off switch would simply be put between the battery (5V) and the i/o pin, then the noise currents will cause uncertainty on the pin. The input voltage on this pin will then vary ('floating'), and the digital value can be read as HIGH or LOW (while we expect only LOW). This is illustrated by a question mark in the illustration above, left.

Now we connect the battery with GND and the i/o pin. We put a switch and a resistor on this connection.

- When the switch is open (off), then the current can only flow to the i/o pin. Then the value of the pin will be **HIGH** (= 5V).
- When the switch is closed (on), then the current will flow to the earth GND. The resistor $R_{pull\ up}$ will make sure the battery will not simply decharge. An internal resistor in the microchip (high resistance R_{imped}) will make sure that the current will run to the GND, and not to the Arduino. Th elogic value of the i/o pin will then be **LOW** or 0 volt.

So in this set up we have to put the switch 'off' to provide power to the Arduino. The used resistor is called a pull up resistor. It reassures that the voltage on the i/o pin is on its maximum (5V, HIGH) in stead of having a floating value. We could say the the pin value is 'pulled up' towards its maximum.

Nowadays the **pull up resistors** are already **built in** in Arduino pins. So you don't have to solder them yourself into the circuit. You can make use of the internal pull up resistors by commanding it in the code of your sketch.

Pull down resistor

The concept explained above can also be treated otherwise. We will now put a resistor between the i/o pin and the GND.

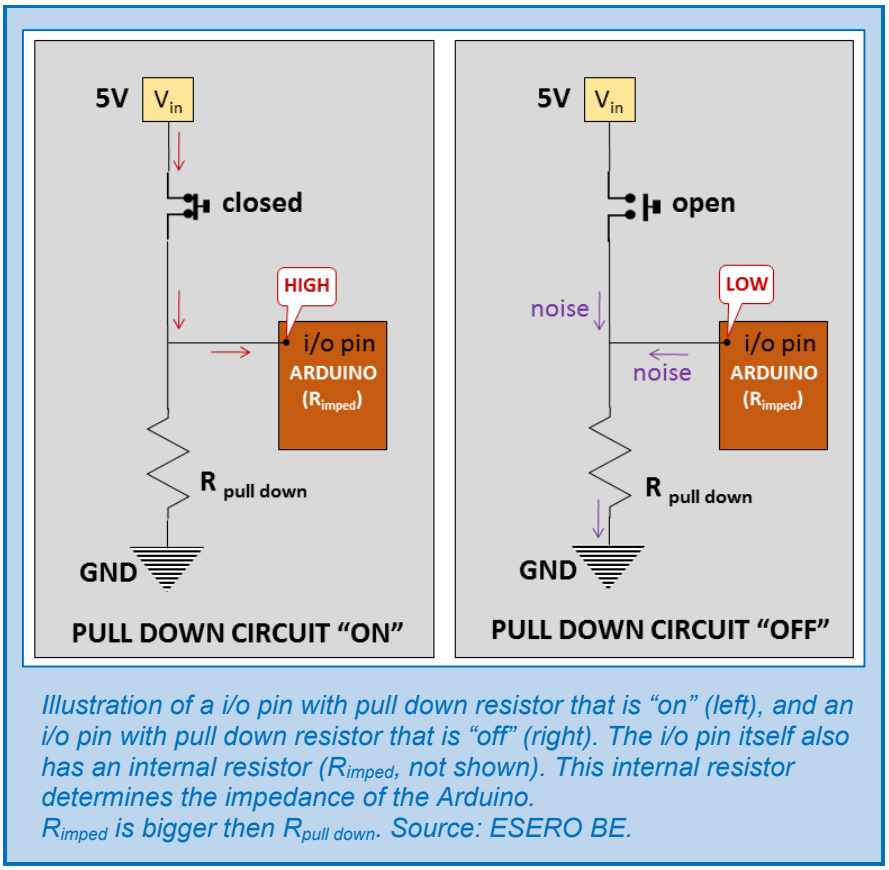


Illustration of a i/o pin with pull down resistor that is "on" (left), and an i/o pin with pull down resistor that is "off" (right). The i/o pin itself also has an internal resistor (R_{imped}, not shown). This internal resistor determines the impedance of the Arduino.
R_{imped} is bigger then $R_{pull\ down}$. Source: ESERO BE.

If the switch is open (off) in this set up, then the voltage on the i/o pin will be 0 Volt. The noise currents flow towards the GND via the $R_{pull\ down}$. This resistor is actually **pulling down** the i/o pin to the logic state of **LOW**.

Both pull up and pull donw resistors are used to keep Arduino pins on clear values of LOW (0V) or HIGH (5V) and to avoid floating values.

SOFTWARE

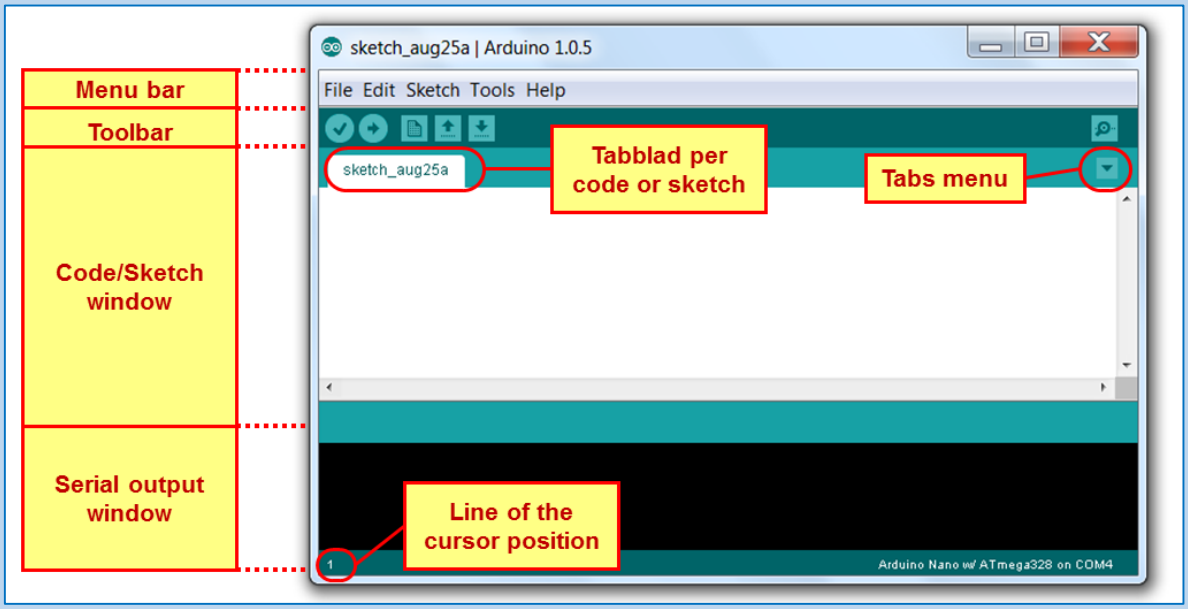
Download

The software you need to install on your computer to work with Arduino is called “Arduino IDE”. IDE stands for Intergrated Development Environment. By putting this word into your favorite internet search engine, you will find immediately the downloadpage of Arduino. Then you can download the software for free (a button called “just download”).

This is the download link: www.arduino.cc/en/Main/Software

Choice your operating system (Windows, iOS, LINUX), and download the program. Then you have to unzip the files? Then you click on the installation file (extension .exe). The software will get installed on your computer.

The interface



Interface of the Arduino IDE with labels for the main components. Source: ESERO BE.

Menu bar

Clicking on the pop up menus you can find any application provided by the software. The most used applications however, are found also as a shortcut button on the toolbar.

Most options under these menus are obvious without any explanation. We will only discuss them in the training as far as we need them doing the exercises.

You should not forget to choose the correct Arduino board you are using and the correct USB port. This can be found under the menu “tools”. This is necessary for a good communication between the Arduino and the computer:

- Tools > serial port > click the right USB port, where the Arduino is actually connected with the computer.

- Tools > board > click the Arduino board that you are currently using (in this training it is Arduino UNO).

Toolbar

Verify
Clicking this button will start the software controlling your code on errors. With 'errors' we mean errors on the code syntax that will make it impossible for the Arduino to execute the sketch program.
After this control, the IDE will show you the location of the error by its cursor position and a brief description of the syntax problem. The it's your turn to find out what the correct version should be.

Upload Using Programmer
With this button you can **upload the sketch** to the Arduino. Before clicking this button, you have to make sure:

a) Make sure you have selected the correct USB port (on what USB port is your Arduino connected) via the tools menu. You have to make this choice only once, in the beginning of the project.
b) Control your sketch on syntax errors using the 'verify' button.
c) Save your sketch with the 'save' button'. Sometimes the system gets blocked unexpectedly, and your work can be lost when you didn't save it before.

When uploading a sketch, the previous one is deleted and overwritten. This can also be a way to empty the Arduino chip (by uploading an empty sketch).

New Editor Window
Clicking this button will give you a **new tab** with a **blanco sketch** that you can fill up with code. The software will ask you to choose a name and location to save the sketch. It is adviced to use the default location (proposed by the program). In the file name the *space* character not allowed.
After confirming, the name of the sketch will be the title of the new tab.

Open in Another Window
You can open a new sketch in a newe window – in stead of the current window – wit this button.

Save
Clicking this button will **save the active sketch** with the name and location you have chosen before. If you didn't make location choice before, the program will save it in the list of your

personal sketches. This way you can easily find back your own work using the IDE menu. You will find all your personal sketches in: file > sketchbook.
After saving, the message “done saving” appears at the bottom. It is a good habit to click the save button after each small change you have made.

Serial Monitor
You can **see the data** that are **sent out by the Arduino** by clicking the Serial Monitor button. The data appear in the bottom black part of the program window, which is called the ‘serial board’. In this case, the serial communication line is the USB cable.
You can choose the speed by which the data are exchanged, using the ‘Baud’ button above the black window. The default speed is 9600. This means that the exchange of data happens at 9600 bits per second.
Serial communication can also happen in the opposite direction. Using the ‘send’ window you can send a message to the Arduino.
Of course, the data that appear in the serial board will be nothing more then the data you have asked the Arduino to send in your uploaded sketch.

Code window

Here you will write the complete code.
The program can change some text color automatically when some codes are recognized as ‘valid’ commands or functions.

You can also add any text you like in common language, that serves as explanation about the function of the code you just wrote. For this you can use some specific symbols telling the program that the actual text you are typing is no code. With these symbols you tell the program: “what I am going to write now is no code, and should therefore not be written to the compiler.”
With compiler we mean the translation program that converts your code in commands that the Arduino will understand after uploading.

If you only want to add **1 line of comments**, then you must open this comment with 2 slashes (//). You can end the comment by pressing ‘enter’ (new line):

```
Code // Here I write my comment: some explanations about the code
before the slashes.
More code
```

Using this code, you will notice that the software changes the text color in grey, the slashes theemselves included. It shows that the software has recognized your comment as ... comment.

If you want to add **multiple comment lines**, then you must use a slash + asterix (/*) to open thee comment, and a asterix + slash (*/) to close it:

```
Code
/*
Here I can write as many lines with comment as I like.
Here I can write as many lines with comment as I like.
Here I can write as many lines with comment as I like.
*/
More code
```

What all other instructions about writing code we refer to the exercises further in the training (simple codes").

Serial output window

As explained under 'toolbar': here youu can see the data sent out by the Arduino.
But this is also the window where the program communicates about errors when you click 'verify', or when the systems uses other internal error messages.

In the colored line under the serial output window there is always a number. This is the number of the line of the actual cursor position.

PROGRAMMING : GENERAL RULES

While programming you have to respect the rules of the programming language. This set of rules is called **syntax**. When you don't apply the rules correctly, then the software will not only give a short description of the error, but it will also show the location (line) where the error was detected. A syntax control happens when you click 'verify'.

There are many syntax rules, but we will only give s short introduction of syntax rules that are important for our sketches. Of course, the best way to learn the rules is by programming yourself. We will do some exercises in the training to get familiar with the general rules.

Set up function

Every code has to be preceded by a 'set up'.
In the set up, some things are defined before the operational code can start. Things to define are for example:

- Define the Arduino pins that will serve as output pins.
- Defining variables that will be used in the code.
- Identification of the sketch that will be communicated to the serial monitor.
- Defining libraries that the code refers to (and that are saved on our computer).
- ...

Everything that is written in set up, will **only** be **executed once** at the start of the program.
The complete set up must be written between left and right accolades (curly brackets):

```
Void setup() {
Ccccccccccccccccccc
Cccccccccccccc
ccccccccccccccccc
}
```

Loop function

The 'loop' contains the actual task that the Arduino will perform line by line. The loop is **executed continuously** until the user interrupts it.
Also the loop must be written between left and right accolades:

```
Void loop() {
Ccccccccccccccccccc
Ccccccccccccccc
ccccccccccccccccc
}
```

The word **void** is put before the word loop and the word set up. It means literally emptiness. It is used because the set up and the loop are both autonomous entities, independant fragment of programs taht are not framed within another program.

Working with an opening an closing **accolade** is facilitated. When you put the cursor in any piece of code within the loop, then the software immediately puts a small rectangle around the loop you are in. The same facilitating method is applied for the opening and closing normal brackets that contain a function.

The semi-colon

Every statement in the code has to be finished with a semicolon. It defines the end of a statement.

Variables

For programming, you need variables. They have two fixed characteristics:

- A name
- A data type

Both characteristics have to be determined by the user (you) in the set up. These characteristics will not change anymore in the rest of the program.
However, the content of a variable can change very easily. You can define the variable characteristics as follows:

```
Datatype name ;
```

When you write the datatype, then it will immediately get the color orange: IDE recognizes the datatype you have chosen.

The name

- Cannot only be a number, and cannot start with a number.
- Cannot contain spaces or special symbols.
- Can contain an underscore (_) or hashtag (#).
- Cannot contain keywords from the programming language.
- Can best be a name that describes the variable.
- The convention says: when the name has multiple words: write them together as one word, but put a capital at the beginning of each partial word (except the first).

Example:

```
int numberOfSeconds;
numberOfSeconds = 5;
```

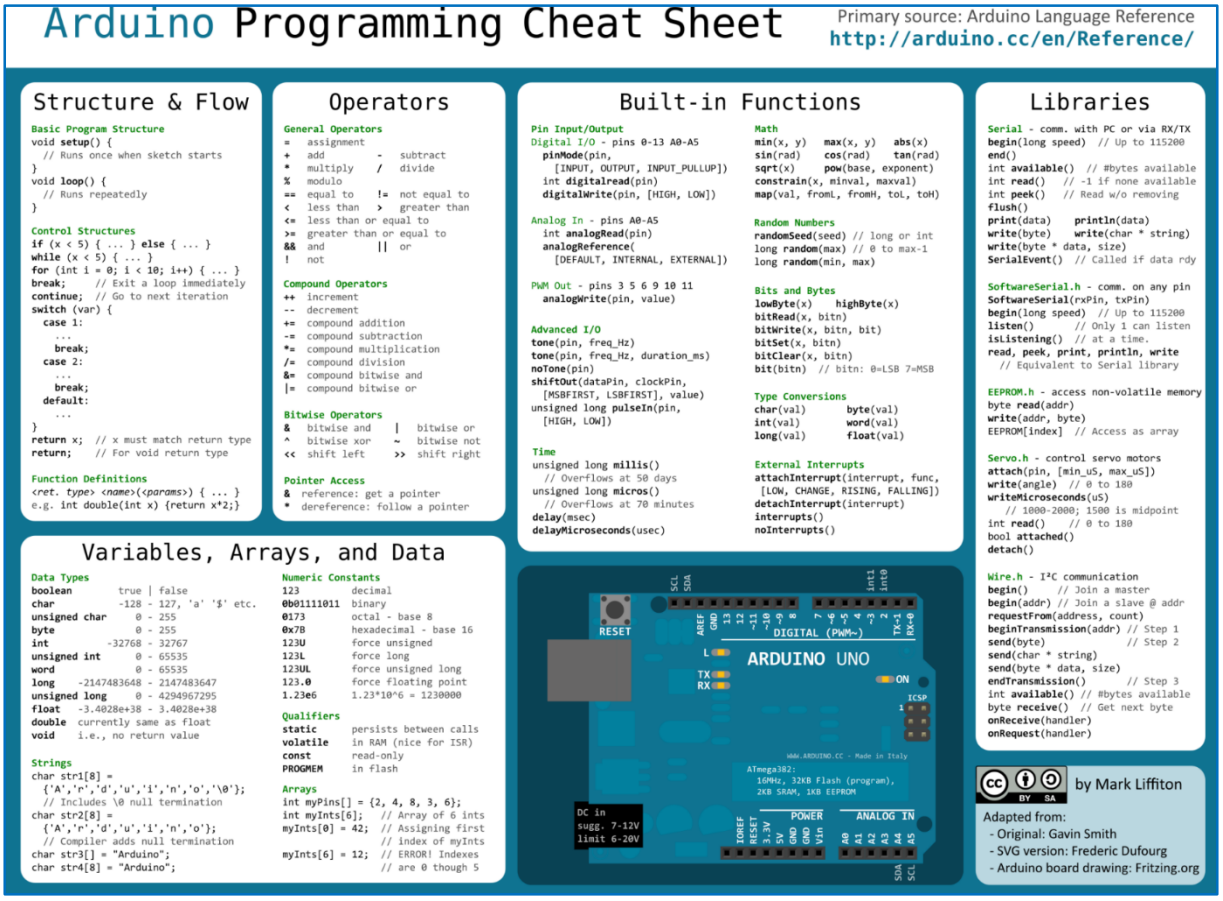
This variable has the name numberOfSeconds is of datatype integer.
Integer variables have no decimal numbers and have a value between -32.768 and +32.768.

What happens when the min or max number gets passed?
Suppose our variable equals at a certain moment +32.768, and we add +1. Then the system will go on counting at the other end of the range. The new value will then be -32.768.

In the example code above we have given the variable numberOfSeconds an initial value 5. Then the program starts running, and the variable can change it's value.

Arduino Cheat Sheet

Arduino has made a practicaal overview of the most used codes on an A4 sheet. This is called the Arduino Cheat Sheet. We advise you to find it in the internet on high resolution, and print it. You will learn fast if you put the cheat sheet next to you while programming.



The programming language is **sensitive for capitals**. For example if you want to send something to a digital pin, you have to use digitalWrite. The command digitalwrite will do nothing. Neither will it change in orange text color, because Arduino cannot recognize it.

Download a Sketch/Library

As said before, most Arduino users never write a sketch themselves.You can find any sketch online. The best sources of good sketches are the component manufacturers pages or the pages of large electronics deliverers like Adafruit.

When you download a sketch, the best thing to do is to save it in the Library folder of the Arduino IDE; Then it can be found via the menu in: sketch > import library > add library ...

SIMPLE EXERCISES

To learn some basic skills in coding, we present some simple exercises below.

SIMPLE EXERCISES : blinking LED

Challenge

We want a blinking LED.

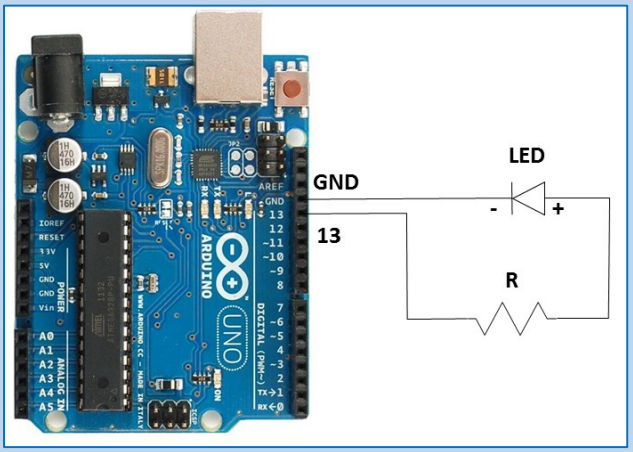
You could do this with the built in LED of the Arduino board on pin 13. However the didactical value of the exercise will be bigger if we also try to connect an external LED. More-over, this is also a bit more cool.

Hardware

We need:

- An Arduino UNO
- A LED
- A resistor of 220 Ω
- A Breadboard
- Jumper cables
- A Laptop + USB kabel

Circuit



Circuit for the exercise "blinking LED".
Source: ESERO BE.

We will use a **digital pin** of the Arduino (in this case 13), because the values aof a LED can be considered as binary: on or off / 0V or 5V.

The LED can have any color. The long leg is the **anode** (+ pole), and the short one is the **kathode** (- pole).
The **resistor** has no defined direction of current (no polarity), and can be connected to the + and – pole just as you like.

Sketch

We will not write the sketch ourselves. To let a LED blink, there is a default sketch in the exaples list of the Arduino IDE:

In the menu bar:
File > Examples > basics > **blink**
You will get this sketch:

```
/*
Blink
Turns on an LED on for one second, then off for one second,
repeatedly.

 This example code is in the public domain.
 */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
```

```
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the
                                voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the
                                voltage LOW
  delay(1000);               // wait for a second
}
```

Let's have a look at some parts of this code:

First we look at the **Setup**:

```
int led = 13;
```

Here we defined a variable called "led". It has the datatype integer, and at this moment we have put value 13 into it. We will use it to send a signal (5V) to a certain pin later on. The desired pin has a pin number, and it is actually this number that will be determined by the variable "led".

```
  pinMode(led, OUTPUT);
```

This is what the statement above is saying:
The pin with the pin number that equals the value of "led" has to function as an output pin.

In this case it is pin 13 that will be used as our output pin, knowing that we have put the initial value of our variable on 13. This is the correct value, because we have actually connected our LED to this pin 13 and we want to send a 5V signal to this location.

The mode of the pin ('pinMode') can only be output or input. By chosing **output** we will make it possible to send a signal to this pin by using the function '**digitalWrite**'.

If we would want to receive data on a pin – using the function 'digitalRead' – then the pin mode has to be put on **input**.

Now we will discuss the **Loop**:

```
  digitalWrite(led, HIGH);
```

Here we command to provide a voltage of 5V (corresponds with the state 'HIGH') to the output pin that has a pin number that equals the value of the variable "led" (this value is still 13). The word HIGH stands for a binary state that is recognized by the Arduino IDE, and therefore it changes automatically its text color in blue.
DigitalWrite means: write the binary state HIGH to a pin with nr. "led".

```
  delay(1000);
```

Here it says: now wait for 1000 miliseconds before running the next step.

```
  digitalWrite(led, LOW);
```

Here we provide the pin with pin number “led” (still 13) with a minimum voltage 0V (= binary state LOW)

Upload

Now click on “upload”. After a while (the uploading time) your code is copied on the Arduino chip. It will now start running your code, so the LED on pin 13 will start blinking: 1 second on, 1 second off, etc.
This will be repeated as long as the user doesn’t stop the process, for example by cutting the power supply.

Change the blinking pattern

Now you fully understand the sketch for blinking a LED, you can start playing with the frequency of blinking. Just change the number of milliseconds that is mentioned in ‘delay’.

Make the blinking visible in words

Now we don’t only want the LED blinking, we also want the Arduino to ‘tell us’ what it is doing. In other words: it has to communicate to our computer wether the LED is currently on or off. This can be very useful when you work with component that doesn’t easily show wether it is working or not – as opposed to our LED that is clearly working when you see the light.

```
int ledPin = 13; // An integer variable gets the name 'ledPin' and
                    is filled in with the initial value 13.

void setup() {
pinMode(ledPin, OUTPUT); // Pin 13 functions as output pin.
Serial.begin(9600); // Serial monitor receives data with baud rate
                       9600 (meaning: 9600 bits per second).
}

void loop() {
int a = 1000; // An integer variable gets the name a and is filled
                 in with the value 1000.
Serial.println("led on"); // Write the message "led on" to the
                            serial monitor.
digitalWrite(ledPin, HIGH); // Send 5V to the pin 13.
delay(a); // Now wait 1000 milliseconds.
Serial.println("led off"); // Write the message "led off" to the
                            serial monitor.
digitalWrite(ledPin, LOW); // Send 0V to the pin 13
delay(a); // Now wait 1000 milliseconds
}
```

Extra exercise

You can now connect a second LED on pin 12, and share the GND pin with the two LEDs (12 and 13). Now try to make blink alternately both LEDs.

SIMPLE EXERCISES : traffic light

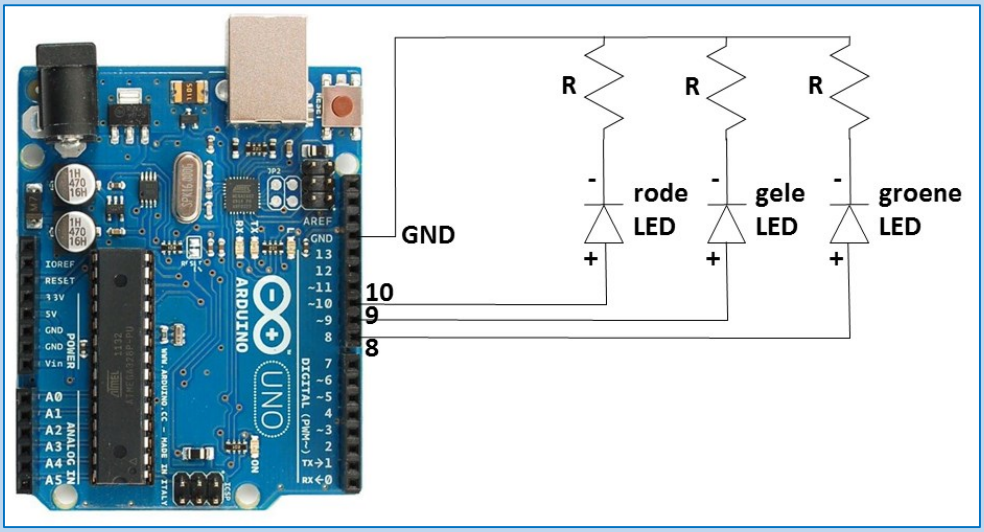
Challenge

We will program a traffic light in England. The light has to be on red or on green, in both directions with an orange transition. Like for a real traffic light, this loop should repeat itself endlessly.

Hardware

- Breadboard
- Red LED, orange LED, green LED
- 3 resistors 220 Ω
- Arduino UNO
- Jumper cables

circuit



Circuit for the exercise "Traffic light". Source: ESERO BE.

Sketch

```
/*
Traffic light:

The traffic light has to be on red or on green, in both directions
with an orange transition. Like for a real traffic light, this
should repeat itself endlessly. The time interval for red and
green must be controlled by 1 parameter.
```

```
*/

int ledDelay = 5000; // This variable will determine the time
                        interval between the color changes.
int redPin = 10;
int yellowPin = 9;
int greenPin = 8;

void setup() {
pinMode(redPin, OUTPUT);
pinMode(yellowPin, OUTPUT);
pinMode(greenPin, OUTPUT);
}
void loop() {
digitalWrite(redPin, HIGH);     // Put on the red LED
delay(ledDelay);                // Wait for ledDelay milliseconds
digitalWrite(yellowPin, HIGH); // Put on the orange LED
delay(2000);                    // Wait for 2 seconds
digitalWrite(greenPin, HIGH);   // Put on the green LED
digitalWrite(redPin, LOW);      // Put off the red LED
digitalWrite(yellowPin, LOW);   // Put off the orange LED
delay(ledDelay);                // Wait for ledDelay milliseconds
digitalWrite(yellowPin, HIGH); // Put on the orange LED
digitalWrite(greenPin, LOW);    // Put off the green LED
delay(2000);                    // Wait for 2 seconds
digitalWrite(yellowPin, LOW);   // Put off the orange LED af

}                               // Now the loop will repeat
```

MEASURE TEMPERATURE

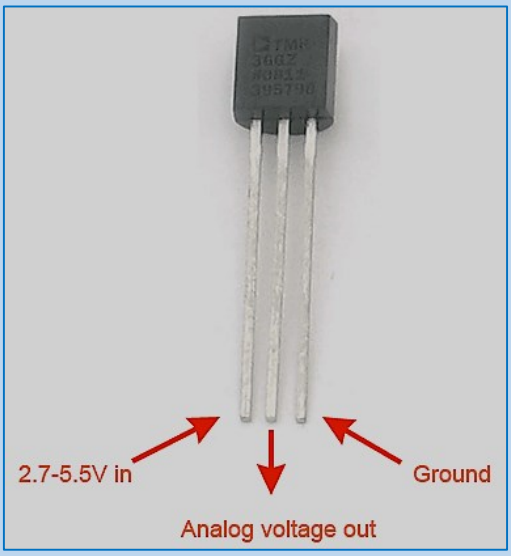
Challenge

Measure the temperature every half second with the provided TMP sensor. Show the measuring data (including the unit °C) in the serial monitor window. Test the system by holding the sensor in your hand to warm it up or cool it down with ice.

Hardware

We need:

- 1 Arduino Uno
- 1 breadboard
- 1 temperature sensor TMP36
- 1 resistor 10kΩ (bruin/zwart/zwart/rood/bruin)
- 3 jumper cables



The TMP36 sensor. Source: Adafruit.

From output voltage to temperature

The output pin ('analog voltage out') will produce a value between 0 and 1,75 Volts, independant of the input voltage (that can be 3,3V or 5V).

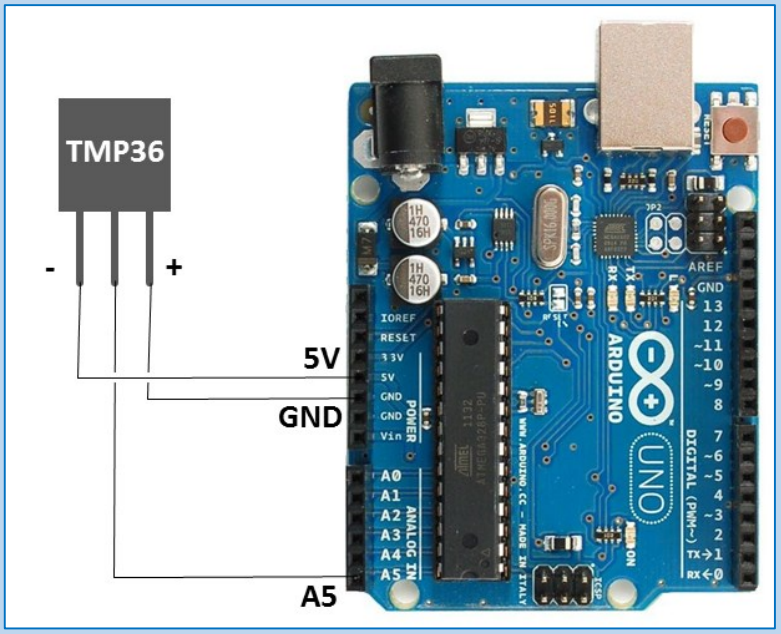
The Arduino analogue pin (ADC) uses 10-bit values between 0 and 1023. These values are first converted to millivolts with this formula:

Value in milliVolt = (ADC-reading) x (input voltage/1024)

The input voltage can be 3300 mV or 5000 mV.
This value in millivolt now has to be converted into temperature value in degrees of Celsius. We use this formula:

Temperature in °C = [(value in mV) – 500] / 10

Circuit



Circuit for the exercise "Measuring temperature". Source: ESERO BE.

Sketch

In this sketch we find some new commands:

- **Float**
  Dit is a datatype with decimals. For an overview of datatypes we refer to the appendix.
- **Serial.println(...)**
  With this command you can write anything to the serial monitor. For example if you want to see the value of a variable, then you need to add the name of this variable between the brackets. If you want to see soem specific non-variable text, then you add this text between the brackets surrounded by quotation marks: Serial.prinln("text")
- **Serial.println** means: write something, then go to the next line.
  Serial.print means: write something without going to the next line.

```
int sensorPin = 5; // SensorPin is the pin-nummer on which the
                     sensor output is connected.
                   // The resolution is 10 mV/°C.
                   // 500 mV offset to make it possible the T
                     could be a negative value.
void setup()
{
Serial.begin(9600);
}

void loop()
{
int reading = analogRead(sensorPin); // 'reading' is the variable
                                        carrying the 10-bit value
                                        that we read on pin A5.
```

```
float voltage = reading * 5000.0;
voltage /= 1024.0;              // We convert the variable
                                   'reading' in an output voltage
                                   between 0mV and 5000mV
Serial.print(voltage);
Serial.println(" mV");          // We write the output voltage with
                                   the unit.
float temperatureC = (voltage - 500.0) * 0.1 ;
            // The variable 'voltage' is converted in a variable
              for temperature in °C. The 500 offset is needed to
              make possible negative values. The resolution is 10
              mV per °C.
Serial.print(temperatureC);
Serial.println(" degrees C"); // We write the temperature with its
                                   unit.
delay(500);      // Wait half a second before we do the next
                    measurements.
}
```

MEASURE AIR PRESSURE

Challenge

Measure the air pressure with the BMP280 sensor. Show the measured values (unit Pa) in the serial monitor. Create a higher and lower pressure around the sensor to test it while the program is running.

Hardware

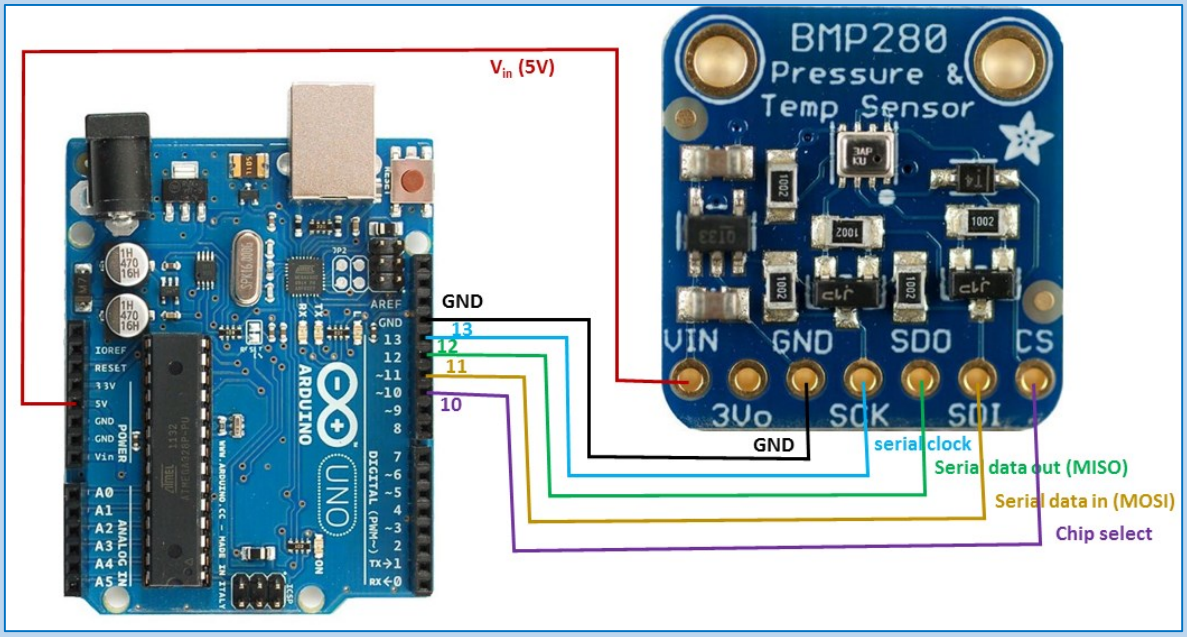
We need:

- 1 Arduino Uno
- 1 breadboard + header stacks (6)
- 1 sensor BMP280
- 6 jumper cables

Circuit

We connect the sensor using the SPI interface. The six sensor pins have to be connected as follows:

- Sensor Vin on the 5V Arduino pin
- Sensor GND pin on the Arduino GND pin
- Sensor SCK pin on the Arduino digital 13
- Sensor SDO pin on the Arduino digital 12
- Sensor SDI pin on the Arduino digital 11
- Sensor CS pin on the Arduino digital 10



Circuit for the exercise "Measuring air pressure". Source: ESERO BE.

Sketch

You can find the sketch for BMP280 sensor on the Adafruit website on this link:
https://github.com/adafruit/Adafruit_BMP280_Library/archive/master.zip

You will get a zip file. When you unpack it, you will find these files:

Adafruit_BMP280_Library-master	.github	ISSUE_TEMPLATE.md PULL_REQUEST_TEMPLATE.md	
	examples	bmp280test	bmp280test.ino
	Adafruit_BMP280.cpp Adafruit_BMP280.h Library.properties README.md		

- A **header file (*.h)** contains a list of definitions and statements that are needed in a library.
- A **source file (*.cpp)** contains the actual code of the library.
- A **sketch (*.ino)** is the sketch you have to open in the Arduino IDE code window.
- A **Markdown file (*.md)** is an informative text file that you can open with any program that can read HTML, like for example your internet browser. Of course you can also use the Microsoft Notepad or Wordpad or the Apple TextEdit, because it is a simple text file.

The folder "Adafruit_BMP280_Library-master" has to get another name. You delete the last part of the name. The new name will then be: "Adafruit_BMP280"

Then you have to copy the whole folder to :
C:/program files (x86)/Arduino/libraries

Now restart the Arduino IDE program. The program screens the Arduino folder for new libraries during start up and adds automatically the copied libraries to the library list.

Then you click it in the list using this menu choice:
File > Examples > Adafruit_BMP280 > bmp280test
Now the sketch for reading BMP280 data is opened in the code window:

```
/***************************************************************
  This is a library for the BMP280 humidity, temperature &
  pressure sensor
  Designed specifically to work with the Adafruit BMEP280 Breakout
  ----> http://www.adafruit.com/products/2651
  These sensors use I2C or SPI to communicate, 2 or 4 pins are
  required to interface.

  Adafruit invests time and resources providing this open source
  code, please support Adafruit andopen-source hardware by
  purchasing products from Adafruit!

  Written by Limor Fried & Kevin Townsend for Adafruit Industries.
  BSD license, all text above must be included in any
  redistribution
***************************************************************/

#include <Wire.h>
#include <SPI.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BMP280.h>

#define BMP_SCK 13
#define BMP_MISO 12
#define BMP_MOSI 11
#define BMP_CS 10

Adafruit_BMP280 bmp; // I2C
//Adafruit_BMP280 bmp(BMP_CS); // hardware SPI
//Adafruit_BMP280 bmp(BMP_CS, BMP_MOSI, BMP_MISO,  BMP_SCK);

void setup() {
  Serial.begin(9600);
  Serial.println(F("BMP280 test"));

  if (!bmp.begin()) {
    Serial.println(F("Could not find a valid BMP280 sensor, check
                                                        wiring!"));
    while (1);
  }
}

void loop() {
    Serial.print(F("Temperature = "));
    Serial.print(bmp.readTemperature());
    Serial.println(" *C");

    Serial.print(F("Pressure = "));
```

```
    Serial.print(bmp.readPressure());
    Serial.println(" Pa");

    Serial.print(F("Approx altitude = "));
    Serial.print(bmp.readAltitude(1013.25)); // this should be
                                             adjusted to your
                                             local forcase
    Serial.println(" m");

    Serial.println();
    delay(2000);
}
```

In this sketch we find some new commands that we need to explain.

```
#include <Wire.h>
#include <SPI.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BMP280.h>
```

The command "**#include**" is referring to library auxillary files. Note: there is no semicolon behind this statement (otherwise you would get an error message).

The first two references (**Wire.h en SPI.h**) are libraries that allow communication in I2C and in SPI. The next two references (**Adafruit_sensor.h en Adafruit_BMP280.h**) are libraries that are specific for this sensor.

```
#define BMP_SCK 13
#define BMP_MISO 12
#define BMP_MOSI 11
#define BMP_CS 10
```

The command "**#define**" creates a constant value with a certain name. For example, in the rest of the program then word BMP_SCK will always be replaced by the value 13.

We don't advise to use the define command when making your own code. Whenever you will use the word that is chosen for the constant value (in our example: BMP_SCK) it will be replaced by the value (in our example: 13). This can cause accidental errors in the output, and these errors will not be detected by the 'verify' function.

```
  Serial.println(F("BMP280 test"));
```

When the 'F' is added to a print command, then the text will be saved in the flash memory in stead of in the RAM. It is a way to save RAM space. It has no other effect.

The RAM (Random Acces memory) is used for temporary commands and data, and can be overwritten endlessly. It is the memory that stores the programming code and that is used for running the program (for example to store and process variables). The size of the Arduino UNO RAM is 2 Kbytes.

The flash memory is used for permanent data and cannot be used endlessly. After overwriting many times it will be broken. The size of the Arduino UNO flash memory is 32 Kbytes.

So we should use the F function only for data that will not change over and over again when the program runs.

```
  if (!bmp.begin()) {
    Serial.println(F("Could not find a valid BMP280 sensor, check
                                                    wiring!"));
    while (1);
  }
```

Bmp.begin() is a command that initializes the sensor.
The exclamation mark "!" means: 'not' (a denial, not equaling something, ...).
So : **If (!bmp.begin())** means : "when the BMP sensor is not initialized" (for example because the sensor is not connected or it is broken).

While means: you have to run this loop endlessly until the conditions between brackets is true.
While (1) is a while-loop that has some kind of absurd condition (that can never be 'true'). Therefore it will just keep on repeating forever, until the user resets the system or disconnects the Arduino.

```
    Serial.print(bmp.readTemperature());

    Serial.print(bmp.readPressure());

    Serial.print(bmp.readAltitude(1013.25));
```

The variables 'bmp.readTemperature' , 'bmp.readPressure' , and 'bmp.readAltitude' are variables that are defined in the header files (auxillary files on your computer).

The **Temperature** has the data type 'float' (with decimals), and is given in °C.

The **Pressure** has the data type 'int' (integer, no decimals), and is given in Pa.

The **Altitude** has the data type 'int', and is given in meter. For calculating the altitude a meteo parameter is used that you will have to find on the internet: the actual air pressure on the ground. You have to put the correct air pressure for your location and for today between the brackets behind bmp.readAltitude().

SIMPLE EXERCISES :
Check the temperature with three color LEDs

Challenge

We measure the temperature, and show if it is too hot (red LED), too cold (yellow LED) or within the desired range (green LED).

Hardware

< to be completed >

circuit

< to be completed >

sketch

< to be completed >

SAVE DATA ON THE SD CARD

Data file

The Arduino microchip has an internal data memory of 200 bytes. This is not enough to save a large quantity of measuring data. That is why we want to save the data on a memory card. We will save the data in a **text file (.txt)**. Afterwards, you can change the extension in .CSV (= comma separated values). You can easily do this by overwriting the extension .txt by the new extension .csv (in Windows Explorer or Macbook Finder). A csv file can be read by several programs, one of them is MS Excel.

Exercise: Write a message to the SD card – introduction

To learn how to write data on the SD card we will first do a simple exercise: write the actuel time (hour) on the card.
At the end of the training, we will write the measuring data on the SD card, using the same method.

The time?

How do we connect an SD card to an Arduino Microcontroller? We will need an extra piece of hardware that iss called an SD Card shield.
A shield is a boardthe fits perfectly onto the Arduino board. By putting it on top of the Arduino, we connect all pins correctly at once. The shield we use in the training has an SD cardholder, but also a real time clock (RTC). We can use the seconds produced by the RTC as reference data to be saved on the SD card.

Exercise: Write a message to the SD card - Hardware

< to be completed >

Exercise: Write a message to the SD card - Sketch

< to be completed >

SEND AND RECEIVE DATA WITH RADIO COMMUNICATION: Brief introduction

At the moment, the ESERO training has not the objective to learn you how to communicate with radio waves. Nevertheless, you will find a brief introduction below, just because you may find it interesting to know.

Space link

A “space link” is the communication system between a satellite and one or more ground stations. It has essentially two parts:

- **Telemetry (TM) downlink**
  The payload’s measuring data are sent out by the satellite. These data are received in the ground station.
- **Telecomand (TC) uplink**
  Command to control the satelite’s behaviour are sent out by the ground station, and received by the satellite.

In this training the TC uplink will be the signals of the remote control of the drone sent out by the drone pilot. The TM downlink is something we have to provide ourselves in our payload (nnot in this training, but you do need it in the CanSat project).

Radio waves and frequencies

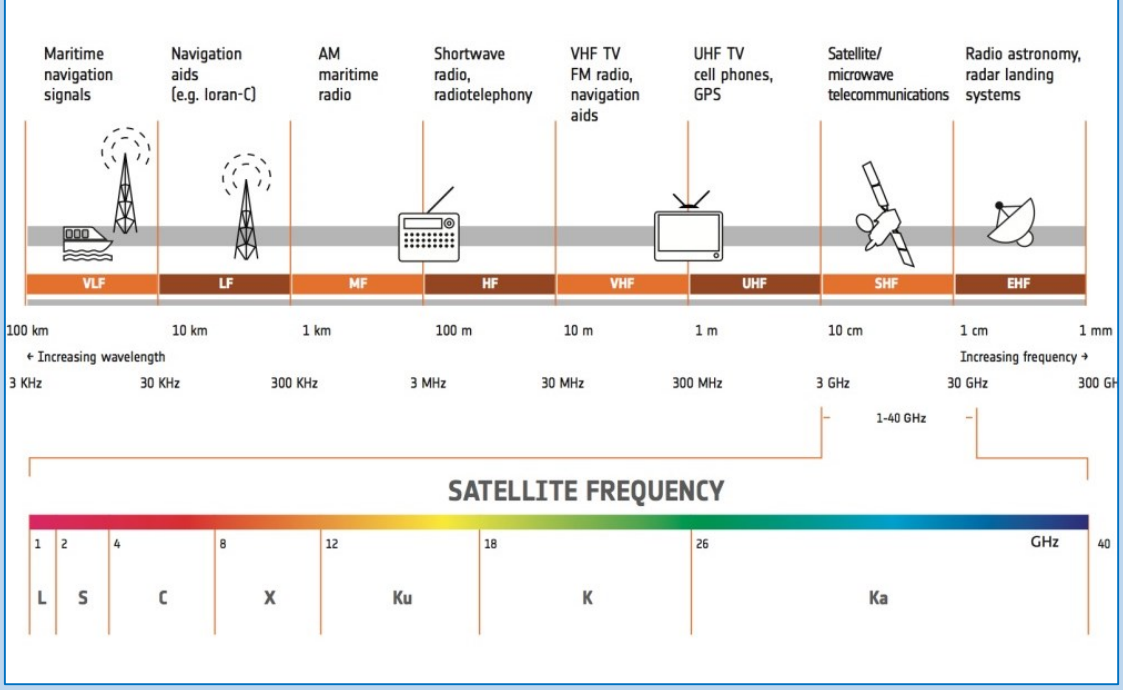
Radiowaves are at the long wavelength extreme of the electromagnetic spectrum. All electromagnetic waves travel at the speed of light c. In vacuum c equals 300.000 km/sec (300.000.000 m/s).

Radiowaves have wavelengths between 10 cm and many kilometers. Like for any electromagnetic wave, you can easily calculate their frequency when you know the wavelength:

$$\lambda f = c$$

λ = wavelength
f = frequency
c = speed of light

So a quick calculation shows that the radio wave frequencies most be smaller (or equal to) 30.000.000 Hz (30 MHz).
Related waves with smaller wavelengths (and thus higher frequencies) are called microwaves and radar waves. Normal radiowaves ($\lambda \geq 10$ cm) are used on Earth for radio communication. But microwaves and radar waves are more often used for space applications, because they can pass the atmosphere without being deformed. They are not reflected by the ionosphere.



Use of radio waves for communication. Source: ESA-ESERO working group Space Robotics.

How can an electromagnetic wave carry information?

You can imagine a radio wave as a sinusoid (a regularly fluctuating electrical field). If you receive such a wave in a device, it will cause a varying voltage. This varying voltage can be measured at any time. The voltage measured at one specific moment is called the 'displacement'. The displacement y (in Volt) can be calculated:

$$y = A \sin (2 \pi f t)$$

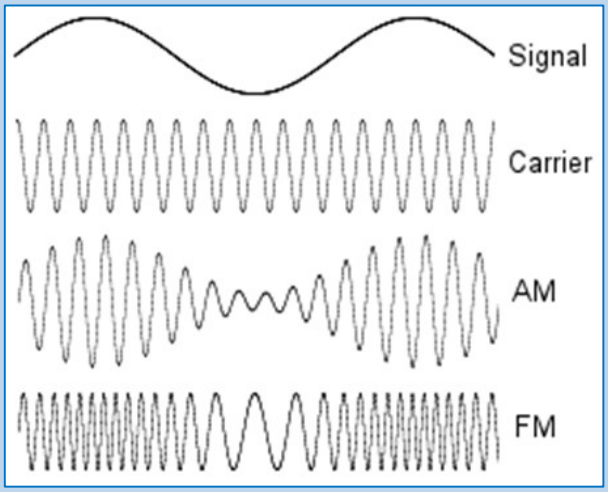
A is the maximal amplitude of the sinus wave.
f is the frequenty of the wave.
t is the moment of the measurement.

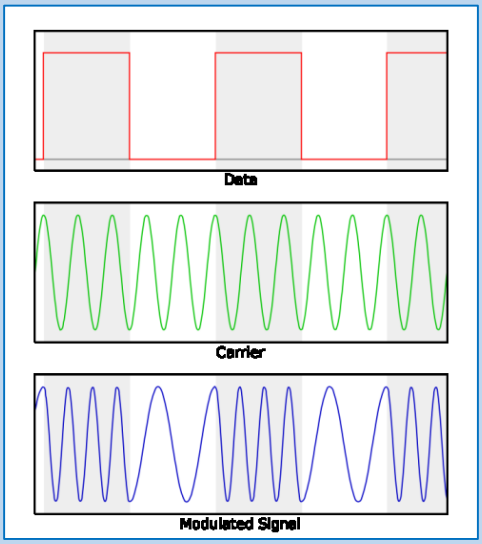
You can understand that 'reading' a perfect sinusoid wave is not very interesting. The displacement will show a boring regular pattern: low-high-low-high-... with a fixed frequency and fixed amplitude. However, this wave is suitable as a **carrier wave** because of the fixed frequency: we can design our receiving device so that we only receive this wave with fixed frequency. But at the moment, it is carrying no information. How will we add information to a carrier wave?

Modulation

We will combine a carrier wave with the actual data or message, the **signal wave**. This is called wave modulation. The combined product of a carrier wave and a signal wave is called a **modulated wave**.

The data are transferred in bots (0 or 1, HIGH or LOW) in a specific pattern. The modulated wave can tranfer these bits simply wiht variations in amplitude or frequency. If it happens with amplitude variations, we call it Amplitude Modulation (AM). If it happens with frequency variations, we call it Frequency Modulation (FM).





Radio wave modulations.
Left: principle of AM and FM.
Right: an example of FM: binary 'Frequency Shift Keying', using two fixed frequencies to express the 0 and 1 values of the binary code.
Source: ESA-ESERO working group Space Robotics.

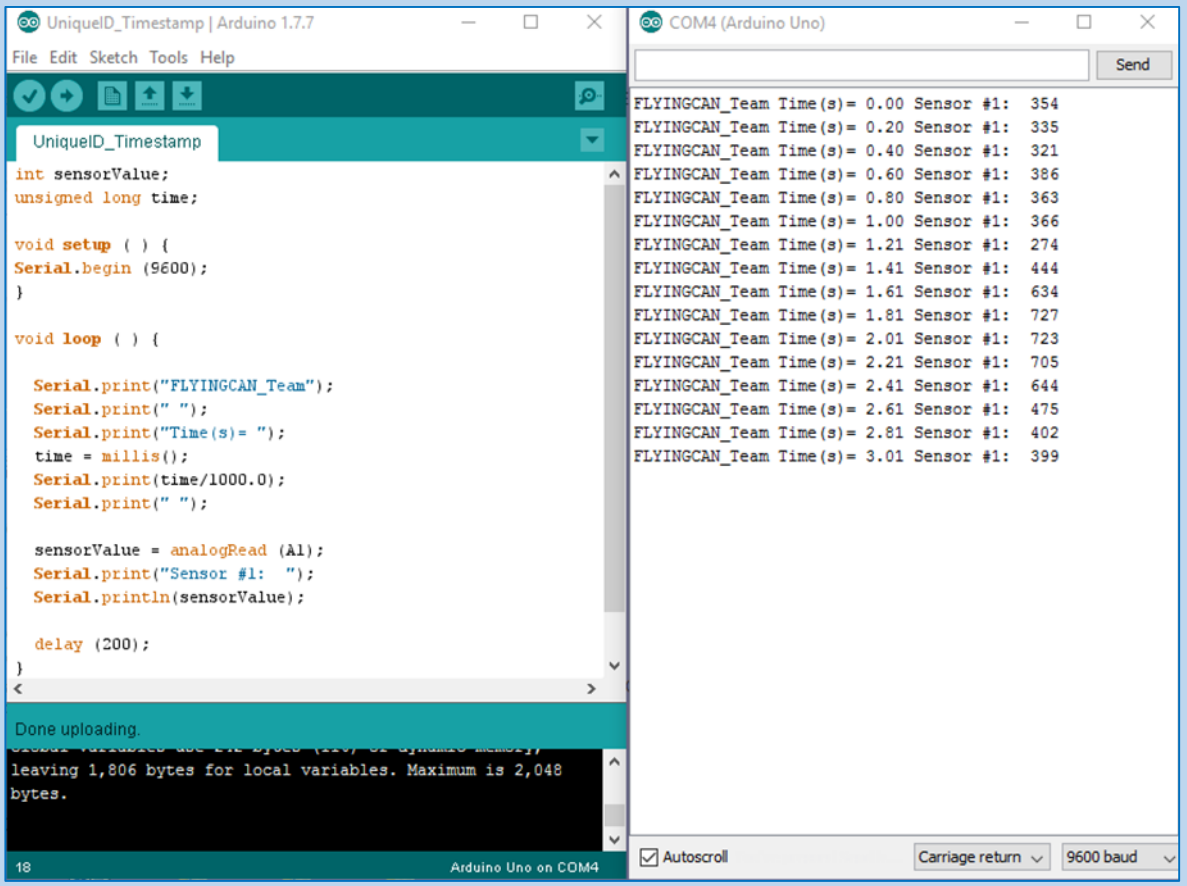
If we produce a small satellite for the CanSat project, we will use **FM**. It has these advantages:

- Less noise that will also get amplified
- Less power is needed to send the same quantity of information
- Larger band width

Metadata

'identifier'
For a school project with satellites, you will ususally receive a unique frequency for your team that you have to use for data communication. But still, when many teams use their unique frequencies all at the same moment, their might be some interference between the different satellites. That is why it is important that you send a unique code or word together with the measuring data. It is a way to identify your own data.

Timestamp
It is also very practical to send the date and hour together with the measuring data. Then the data can not easily get mixed up.



Example of a sketch where an identifier and a timestamp were added.
Source: ESA-ESERO working group Space Robotics.

2c Preparing your satellite

THE FINISHED SATELLITE: HARDWARE

Your satellite will be ready when the datalog shield is soldered on the Arduino, and the BMP280 sensor is soldered on the correct pins of the shield.
Using the I2C communication this time, we connect these pins on the shield:

- Sensor Vin – Arduino 5V
- Sensor GND – Arduino GND
- Sensor SCL – Arduino A4
- Sensor SDA – Arduino A5

The clock's time reference pulse (of the datalog shield) enters the Arduino on analogue pin 4.
The measuring data of the sensor enter the Arduino on analogue pin 5.
The data will be written to the SD card in a newly created text file. This is possible because the shield is connected with all of its pins to all of the Arduino pins.

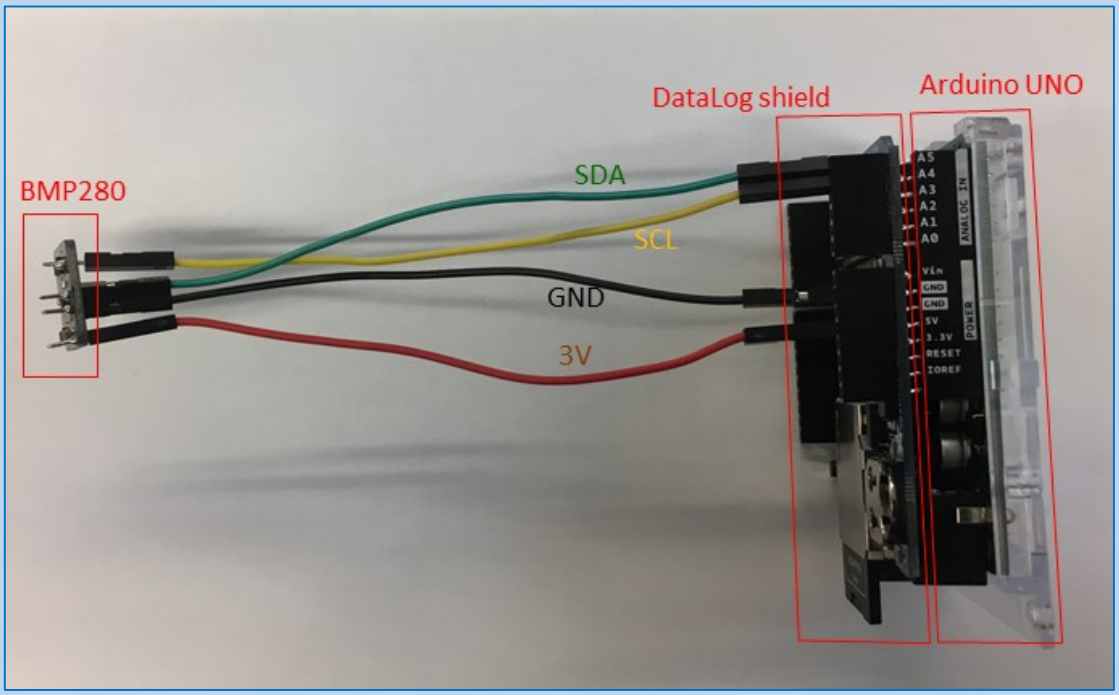


Photo of one of the satellites made by teachers in the ESERO BE training.
Source: ESERO BE.

INTEGRATING ALL CODES IN 1 SKETCH

Structure of the complete sketch

To fully understand this sketch, we must first understand its structure:

1) References to the **libraries**
2) Defining the input/output function of particular **pins**

SET UP

3) Determining **communication speed** between the microcontroller and the serial monitor.
4) **Intitializing** the **sensor** and the **SD card** : check if these components are recognized by the microcontroller as well functioning components.
5) Opening the **datalog file** that will store all the data (datalog.txt).
6) **Initializing** the real time **clock** : check the functionality of the clock and define the date and hour on T0.

Repeating LOOP

7) We write the current **time** (year, month, day, hour, min, sec) to the datalog file.
8) We write the current **temperature** (°C) to the datalog file.
9) We write the current **altitude** (m) to the datalog file.
10) We take the **next line** in the datalog file.
11) We write the current **time** (year, month, day, hour, min, sec) to the serial port.
12) We write the current **temperature** (°C) to the serial port.
13) We write the current **altitude** (m) to the serial port.
14) We write the **next line** to the serial port.
15) We write the new data in the datalog file (time, temp, altitude) to the **SD card.**

16) We wait for **half a second.**

Code of the complete sketch

The sketch below has to be copied completely to the Arduino IDE code window. Then we will upload it to the Arduino.

```
// First we make some references (#include) to libraries that
   we have downloaded on our computer. These libraries were
   saved under:
   C:/program files(x86)/Arduino/libraries
#include <SPI.h>
#include <SD.h>
#include <Wire.h>  // This library supports SPI as well as I2C
                      connections.
#include "RTClib.h"
#include <Adafruit_Sensor.h>
#include <Adafruit_BMP280.h>

// Then we define the input or output functions of some pins
   (sensor communication).
#define BMP_SCK 13
#define BMP_MISO 12
#define BMP_MOSI 11
#define BMP_CS 10

Adafruit_BMP280 bmp; // With this command we define that we
                        will use the sensor in a I2C
                        communication interface (it refers to a
                        certain mode from the sensor library).

const int chipSelect = 4; // We fix the value of a variable
                             called chipSelect on 4 (cannot be
                             changed afterwards).
File dataFile; // Make a file with the name dataFile.

RTC_PCF8523 rtc; // Reference to the Real Time Clock of the
                    datalog shield.
char daysOfTheWeek[7][12] = {"Sunday", "Monday", "Tuesday",
   "Wednesday", "Thursday", "Friday", "Saturday"};
     // Make an array in which 7 = Zondag, 8 = Maandag, enz.

void setup()
{
  Serial.begin(57600); // This is the baud speed that we need
                          to exchange data. It has to be the
                          same as the baud of the serial
                          monitor (you can click it at the
                          bottom right of the serial monitor
                          window).
    if (!bmp.begin()) {  // Check if the sensor BMP280 is
                            recognized as a well functioning
                            sensor.
    Serial.println(F("Could not find a valid BMP280 sensor,
     check wiring!"));
```

```
    while (1);            // Error: Do nothing until the user
                             resets the system.
  }

  Serial.print("Initializing SD card...");
  pinMode(CS, OUTPUT); // Make sure the chip select pin is
                          connected as an output pin, even
                          when we don't use it.
  if (!SD.begin(chipSelect)) {  // Check if the SD card can be
                                   accessed by pin 4.
    Serial.println("Card failed, or not present");
    while (1) ;                // Error: Do nothing until the
                                  user resets the system.
  }
  Serial.println("card initialized.");

  dataFile = SD.open("datalog.txt", FILE_WRITE);
                          // Open a new file to log our data.
  if (! dataFile) {     // Check if the new datafile exists.
    Serial.println("error opening datalog.txt");

    while (1) ;         // Error: Do nothing until the user
                           resets the system.
  }

  // RTC initialization:
  if (! rtc.begin()) {  // Check if the real time clock
                           functions correctly and can be
                           recognized.
    Serial.println("Couldn't find RTC");

    while (1);          // Error: Do nothing until the user
                           resets the system.
  }

  if (! rtc.initialized()) {   // Check if the time is running.

    Serial.println("RTC is NOT running");
    rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
    // Choose the moment that the program is compiled as the
       starting date and hour in the RTC chip. If you prefer to
       choose your own particular date and time as the start,
       then you need this command:
       rtc.adjust(DateTime(2014, 1, 21, 3, 0, 0));
  }

void loop()
{
  // We write the actual time to the datafile:
  DateTime now = rtc.now();
  dataFile.print(now.year(), DEC);
  dataFile.print('/');
  dataFile.print(now.month(), DEC);
  dataFile.print('/');
  dataFile.print(now.day(), DEC);
```

```
      dataFile.print(' ');
      dataFile.print(now.hour(), DEC);
      dataFile.print(':');
      dataFile.print(now.minute(), DEC);
      dataFile.print(':');
      dataFile.print(now.second(), DEC);
      dataFile.print(';');
      // We write the actual temperature to the datafile:
      dataFile.print(bmp.readTemperature());
      dataFile.print(';');
      // We write the actual altitude to the datafile:
      dataFile.println(bmp.readAltitude(1021));
      // Don't forget to adjust the figure between brackets to the
       current air pressure (in hPa). This is necessary to make a
       correct calculation. You can find the detailed calculation
       in the BMP280 library.

      // Below we repeat the same actions, but now we write to the
         serial port:
      Serial.print(now.year(), DEC);
      Serial.print('/');
      Serial.print(now.month(), DEC);
      Serial.print('/');
      Serial.print(now.day(), DEC);
      Serial.print(' ');
      Serial.print(now.hour(), DEC);
      Serial.print(':');
      Serial.print(now.minute(), DEC);
      Serial.print(':');
      Serial.print(now.second(), DEC);
      Serial.print(';');
      Serial.print(bmp.readTemperature());
      Serial.print(';');
      Serial.println(bmp.readAltitude(1021));
      // The following line will 'save' the file to the SD card
         after every line of data - this will use more power and
         slow down how much data you can read but it's safer! If
         you want to speed up the system, remove the call to
         flush() and it will save the file only every 512 bytes -
         every time a sector on the SD card is filled with data.
      dataFile.flush();  // Save the file on the SD card after each
                            line of data that was added.

      delay(500); // Wait for half a second, then repeat the loop.
      }
```

Details of this sketch

Below we explain some expressions that were used in the sketch:

const

```
    const int chipSelect = 4;
```

With the "**const**" command, you can put a fixed value in a variable. Once this fixed value is chosen, it becomes impossible to change it again in the rest of the sketch. It has become a 'read only' value.

Mind: with the 'const' command, the 'equals' sign (=) is used, and the statement is ended with a semicolon (;). This is not the case with the '#define' command. The latter can also be used to put a certain value into a variable, but this value can still be changed afterwards.

The statement above can be replaced by:

```
#define int chipSelect 4
```

In the original statement from our sketch (**`const int chipSelect = 4;`**) the variable chipSelect gets a fixed value 4. So, any place below where we use the variable chipSelect, it will equal 4. If you would try to give it another value, the system would see it as an error.

!SD.begin

```
      if (!SD.begin(chipSelect)) {
```

The exclamation mark is a negation ('not').
For example, the statement **`If (!x < 0)`** literally means: if x is not smaller then zero.

In the shield (with SD card slot) that we use, pin4 is used a the chip select pin (CS) of the SD card. You could see it as the 'address' of the SD card.
You remember that the value of the variable chipSelect was 4. So in the statement above, you simply have to consider 'chipSelect' as a 4. So actually it says:

```
if (!SD.begin(4))
```

The word '**begin**' stands for the opening or starting up of the SD card. The statement means literally: "if the SD on pin 4 doesn't open...".

While (1)

```
    if (!bmp.begin()) {
      Serial.println(F("Could not find a valid BMP280 sensor,
                       check wiring!"));
      while (1);
    }
```

While loops will repeat themselves endlessly until the condition between brackets is wrong.
For example:

```
While (OurVariable < 200)
{
      doSomething;
      OurVariable++;
}
```

The statement above means:
As long as the variable is smaller then 200 you repeat doing the following:

a) Do domething
b) Add +1 to the variable

Now you could say that the experssion “while (1)” is a bit strange. The condition to repeat the loop is just “1”. This doesn’t seem to make sense.
This expression is used when the loop has to be repeated again and again until something radical changes in the initial situation. You could say: 1 will always be 1. And therefore, the condition for “while(1)” will always be correct. So the loops keeps repeating forever.

In our sketch the expression “while(1)” is used in combination with “if”. The condition for this if-function is about recognizing a well functioning sensor. When this sensor cannot be found, then the loops keep repeating (writing the message “could not find...” to the serial monitor) until the user interrupts the program and changes the hardware.

dataFile

```
dataFile = SD.open("datalog.txt", FILE_WRITE);
```

We have to create a text file on the SD card to store the data. We will give this file the name datalog.txt, but it could as well be any other name.

In the following code we will refer to this file with “**dataFile**”.

The command `SD.open("fileName",FILE_WRITE)` will create a new file with this name on the SD card, unless if there would already be a file with this name. The expression `FILE_WRITE` will assure that we have access to the fiel for reading and writing.
An alternative expression is `FILE_READ`, when we want to have a file that is read-only.

The filename between quotation marks (`"datalog.txt"`) can also contain a series of folders. For example: `"ESEROvorming/20171109/datalog.txt"`. If there is no folder name, then the file is simply put in the main directory of the SD card.

Flush

```
dataFile.flush();
```

The command `Flush` is used to overwrite the complete file (that is stored at this moment on the temporary Arduino memory) on the SD card. With the flush command, it is overwritten each time a new data line was added. Using this flush command will cost you some extra power consumption and it slows down the process a little bit. But it is safer, your data will not easily get lost.

If you don’t use the `Flush` command, then the datafile will only be overwritten each time that 512 bytes of new data were added. That is because the microcontroller can only store 512 bytes on data internally. In that case, the reading of data can go a bit faster and using less power, but the risk for lost data is bigger when something goes wrong.

SOLDERING

WHY SOLDERING ?

The advantage of soldering all the satellite components is very clear: they will stay firmly connected during the flight or launch. A well soldered connection is very reliable!
Is the soldering mandatory in an electronically controlled experiment? There is only one possible answer to this: YES! It is absolutely necessary. The risk that all your work will get lost because of a contact that gets disconnected is quite big if it is not soldered.

SAFETY

- A soldering iron has an operational temperature of 300 à 400°C. So please be careful with the iron tip to avoid burns.
- Always put the soldering iron away far enough from other objects.
- Put the soldering iron off as soon as the work is done, and give it time to cool down before putting it away.
- Please wear safety glasses.

MATERIALS

Soldering metal

With soldering metal we mean the metal wire you melt on the connection point.

The traditional soldering metal is an alloy containing lead. This is an alloy that runs very smoothly and fluently in the connection point. Therefore it is the ideal soldering metal for beginners. But the use of this alloy is often discouraged in schools because of the lead containing vapor during heating.

Solderins station

The soldering iron is the central tool for a soldering job. It has a metal tip that heats up to about 400°C. Usually it is delivered with a useful handy holder that allows you to free your hands without causing any danger.

The price of a cheap soldering station is about 25 euros. Such tools are good enough to do good soldering work. The advantage of more expensive soldering stations is usually that they have a longer lifetime, especially when you leave them on daily and for many hours while you are working.

Accessoires

We advice to have these tools with you:

- A small pincer with pointed ends.
- A wire cutter.
- Wire strippers to remove the isolation plastics at the cable ends.

- Facultative: a hands-free magnifier (glass or stereoscopic microscope).
- Good lightning of the working place (day light is better then artificial light).

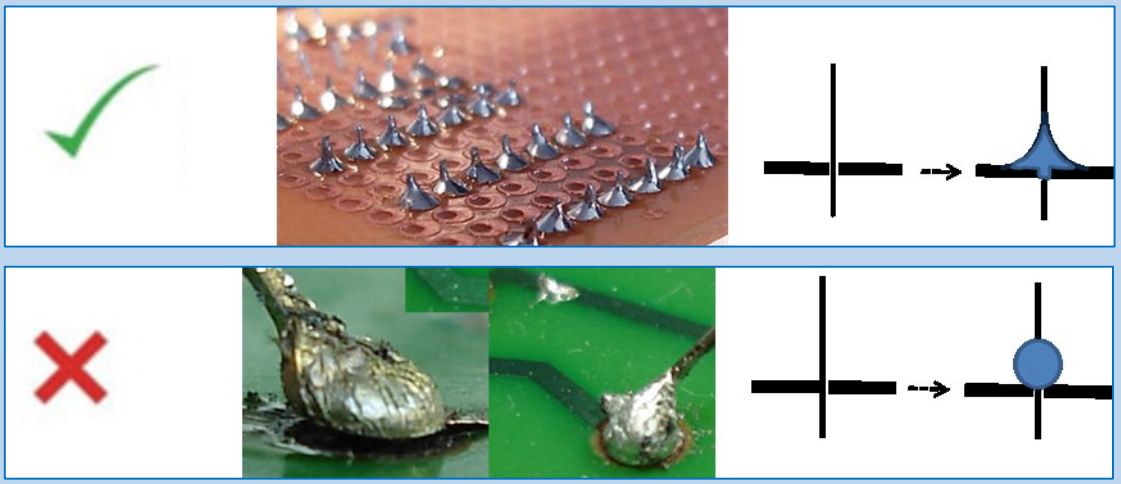
PREPARATIONS

1. Put the soldering iron in its holder and put the power cable in the contact. The soldering iron needs some minutes to **reach the temperature** of about 350 - 400°C.
2. Moisten the **sponge** of the soldering station. It has to be slightly moisted, but not really wet. Squeeze out the excess of water before starting.
3. Wait some minutes for the iron to get the **working temperature**. You can check this by holding a tiny bit of soldering metal against the iron tip. If it melts immediately, then it has reached the right temperature.
4. Now you **clean the iron tip** by rubbin it to the sponge.
5. **Melt a little bit** of soldering metal on the iron tip. This is called tinning. It helps to relocate the heath of the tip to the point where the connection has to be made. You should only do this after cleaning the tip on the sponge.

SOLDERENING : INSTRUCTIONS

Now you are ready to start soldering.

1. Hold the soldering iron **like you hold a pen**. Act as if you would write your name, but mind not to touch the hot tip with your fingers.
2. **Touch the spot** that you have to solder **with the iron tip**. Make sure the tip touches both surfaces of the connection well. Push the iron pin for 3 seconds against the spot and then ...
3. **Put some soldering metal** on the connection spot. If everything is on temperature, the metal will flow smoothly over the printlane and the component. It has to take the shape of a volcano. Lake sure to put the metal against the connection and not against the iron tip.
4. **First remove the soldering metal wire**, and then the soldering iron. The soldered connection should cool down for a moment now, without any movement.
5. **Inspect the connection**. The solder has to blink a bit and should have the shape of a volcano. If this is not the case, you should rewarm the connection and add some more soldering metal. This time make sure that the printlane as well as the component or heated well enough.



Images of good and bad soldered connections. The bad connection may be caused for several reasons: too much soldering metal, the soldering metal was heated rather then the component, the connection was not clean, ...
Source: ESA-ESERO working group Space Robotics.

You can solder a connection from two sides:

- **Thrue hole soldering**
  The component's pin or leg is put into a hole from the print board, the soldering metal is molted into the hole from thee other side (underside).
- **Smt**
  Everything from the same side. This technique is slightly more difficult with a common soldering iron.

A checklist for a perfectly soldered connection

1. All parts should be perfectly clean and not oxidized.
2. Immobilize the print board in a standard as a helping hand or in another way.
3. Tin the iron tip with a small quantity of metal. Always do this with the new irons that are used for the first time.
4. Clean up the hot iron tip on the sponge.
5. Heat up both parts of the connection with the iron tip for about a second.
6. Keep heating up and then add some soldering metal to form a firm connection.
7. Remove the iron and put it back into the station.
8. It will not take more then three seconds to make a general soldering connection.
9. Don't move any parts before the soldering metal is cooled down and the connection has become hard.
10. Check if the new connection is a good one.

Read more:
https://learn.sparkfun.com/tutorials/how-to-solder-through-hole-soldering

3 LAUNCH

Drone flights
Harness
Launches at ABC projects
< to be completed >

4 Processing and presenting data

Calibration and processing: introduction

You can program the Arduino in a way that the raw data (measurements) are calibrated and processed immediately in the payload. The the ground station will receive directly the measurement results as you finally wanted them. We will do this in the ESERO training becausse our training time is very limited.
But this way of working is not advised in general. It is better to receive all the raw data and to process them with the computer. Then the risk that something goes wrong is smaller. Especially when you end up with 'strange' results, it is most valuable to havt he raw data. Then you have to overdo the processing and try to find out what went wrong.
It happens often for student teams with limited or no experience that their electronically controled experiment deliver s set of data that are absurd or meaningless (at first sight). In that case, there is some hope left if the raw data can be reprocessed.

TEMPERATURE

Using the BMP280 sensor in the ESERO training, we will get directly the processed temperature, air pressure and altitude on the SD card. However, we think it is useful in this course to explain how youu can calibrate temperature data when using an analogue sensor. It allows you to do measurements at school with a very simple thermistor.

Calibration of the thermistor output

The output data of a simple thermistor are varying voltages with the unit Volt. You can convert them into the corresponding temperatures in degrees Celsius.

For most temperatures on Earth, we can assume the the thermistor output (measured voltage) and the temperature have a linear relation. So the calibration will not be complicated. We just need to do some simple tests before the actual measurements with known temperatures (measured with a common thermometer) for which we determine the corresponding output voltages. Putting these known points in a graph (x = voltage, y = temperature), we can easily find the best fitting straight line that represents the relation between the output voltage and temperature for all points.

Of course we are not happy wwwith only a graph, and we want a mmmathematical formula to convert the voltages into corresponding temperatures. Then we simply have to write the formula for a linear relation:

$$y = mx + c$$

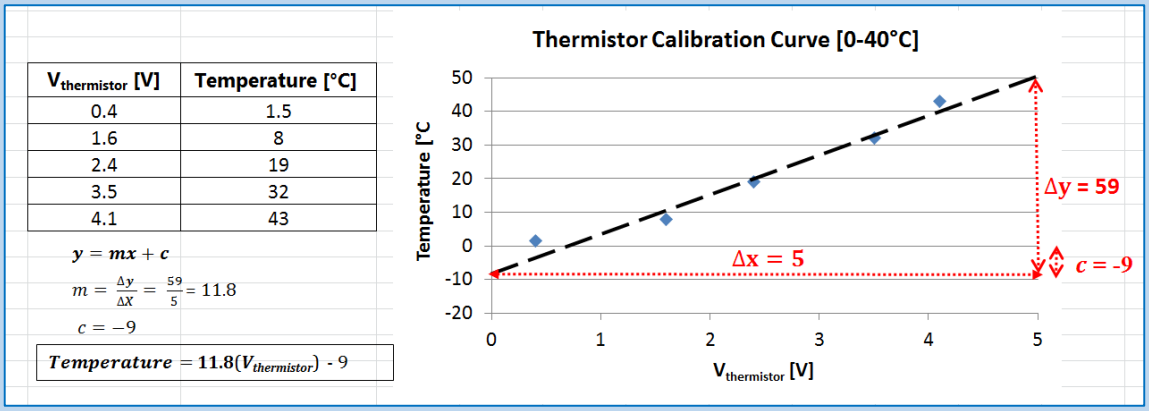
The value of m determines the inclination of the line.
The value of c determines where the line reaches zero (crosses the x-axis).

Applicating this on the linear relation between the thermistor voltage and the temperature, you can say:
Y = temperature
X = thermistor voltage

In our graph that was based on some test measurements, we can find the m value:

$$m = \Delta y / \Delta x$$

Now we only have to fill in the Δy and Δx of the extreme test values, and we will know the inclination m.



$V_{thermistor}$ [V]	Temperature [°C]
0.4	1.5
1.6	8
2.4	19
3.5	32
4.1	43

$y = mx + c$

$m = \frac{\Delta y}{\Delta x} = \frac{59}{5} = 11.8$

$c = -9$

$Temperature = 11.8(V_{thermistor}) - 9$

Calibration of the thermistor output values to find corresponding temperatures. Example of 5 measured test values, corresponding graph and calibration formula.
Source: ESA-ESERO working group Space Robotics.

AIR PRESSURE AND ALTITUDE

Calculating the altitude based on the air pressure values

The relation between the atmospheric pressure and the temperature is given below in this formula:

$$P = 101325\ (1 - 2.25577\ 10^{-5}\ h)^{5.25588}$$

P is the air pressure given in Pascal (Pa). h is the altitude (height) in meter above sea level. The value of 101325 is just an average air pressure at sea level. You should adapt this value to the pressure measured on your location at the moment of your flight (or just today). You

can find this measurement value on the website of the meteorological services in your country.

To find the altitude when you only know the air pressure (pressure is what we measure with our sensor), you hev to rewrite the formula as follows:

$$h = \frac{10^{\left(\frac{log\left(\frac{P}{101325}\right)}{5.25588}\right)} - 1}{-2.25577 \times 10^{-5}}$$

That is how you can reconstruct the altitude of your flight every second. Again: don't forget to adapt the value 101325 to your local and actual situation. A not adapted value can cause an error in the calculated altitudes up to 10 meters.

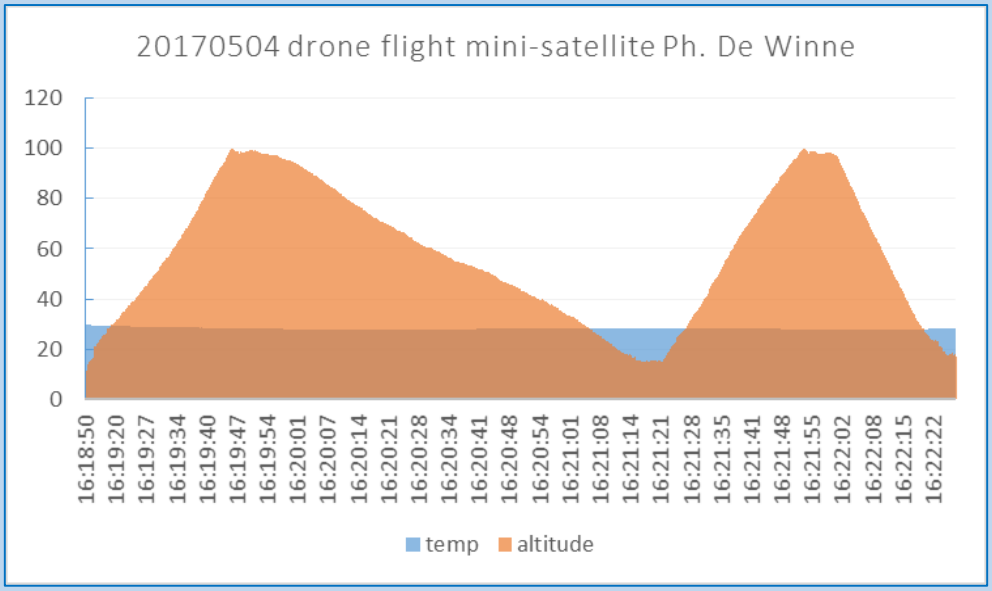
Porcessing the data of the BM280

Using the sketch we have given, again, you still have to put in code the adapted value for air pressure. But apart from that, in this sketch everything is provided to end up with an output .txt file written on your SD card, that you can read in MS Excel. It will contain measurements per time unit of temperatures and altitudes, the latter calculated from the original air pressure values.

On the SD card we willl find a txt file after the flight (it is written to the card as soon as you activate the Arduino). Follow these guidelines:

- Copy the txt file somewhere on your computer.
- Ga to Windows Explorer (or Mac Finder) and find back the txt file.
- Doubleclick on the file name: you will get the possibility to change its name.
- Don't only change the file name (as you wish), but also change the extension .txt to .csv (= comma separated values).
- Now opne this file in MS Excel.
- Now you will have all thee data in an easy to process form. You might want to add some layout formats, giving you a better overview af data and titles.
- Make a graph in Excel.

timestamp	time	temp	altitude
17/05/2017 16:18	16:18:50	29.83	11.37
17/05/2017 16:18	16:18:51	29.84	11.2
17/05/2017 16:18	16:18:52	29.87	11.39
17/05/2017 16:18	16:18:53	29.89	11.22
17/05/2017 16:18	16:18:54	29.89	11.3
17/05/2017 16:18	16:18:55	29.9	11.37
17/05/2017 16:18	16:18:56	29.92	9.94
17/05/2017 16:18	16:18:57	29.91	12.69
17/05/2017 16:18	16:18:58	29.92	13.84
17/05/2017 16:18	16:18:59	29.88	13.89
17/05/2017 16:19	16:19:00	29.85	14.29
17/05/2017 16:19	16:19:01	29.79	15.11
17/05/2017 16:19	16:19:02	29.67	15.05



The processed measuring data of our training "make your first mini-satellite" in a table (fragment, upper image) and in a graph (lower image). Not that there are no significant temperature differences in function of height. Source: ESERO BE.

Sources of information

- ESERO BE Teacher training given by FabLab Klein-Brabant April-may 2017 (Davy Vanden Bergh)
- ESERO BE Teacher training given by Collège Saint-Michel May-June 2017 (Nicolas de Generet)
- Resources about the CanSat primary mission by ESA Education (working group ESA/ESERO): draft version June 2017.
- Teacher training Primary CanSat mission in Europe Jan 2015 (T-Minus).
- Arduino for beginners: Ohm my God https://www.youtube.com/watch?v=zWIsXmtp3Ow&list=PL3ZWCJtjleYHCB0cpVPtunimiERmMy3I5&index=3#t=780.552917
- Evans, Brian W. (2007). Arduino programming notebook. http://creativecommons.org/licenses/by-nc-sa/3.0/
- Adafruit learning resources: https://learn.adafruit.com

Appendix

Data types

Data type	RAM memory		Extremes
boolean	1 byte		0 to 1 (True or False)
byte	1 byte = 8 bits	Numerical values without decimals	0 to 255
char	1 byte		-128 to 127
unsigned char	1 byte		0 to 255
int	2 bytes = 16 bits	Numerical values without decimals	-32,768 to 32,767
unsigned int	2 byte		0 to 65,535
word	2 byte		0 to 65,535
long	4 bytes = 32 bits	Numerical values without decimals	-2,147,483,648 to 2,147,483,647
unsigned long	4 byte		0 to 4,294,967,295
float	4 bytes = 32 bits	Numerical values with decimals	-3.4028235E+38 to 3.4028235E+38
double	4 byte		-3.4028235E+38 to 3.4028235E+38
string	1 byte + x		Arrays of chars
array	1 byte + x	Values carrying an index number. They are referred to by these index numbers. For each index number the value has to be defined in advance by the user.	Collection of variables