PLUSIVO

WIRELESS SUPER STARTER KIT WITH ESP8266

Table of Contents

1. Introduction

In the last period technology has evolved exponentially and devices became smaller and smaller, with less power consumption and, in the same time, with more powerful chips and low prices. The development board included in this kit is built around the powerful microcontroller ESP8266. This development board is WiFi Ready, so it is a solid option for most **IoT** (Internet of Things) projects.

ESP8266 features:

- 32-bit RISC CPU: Tensilica L106 running at 80 Mhz, but can go up to 160 Mhz;
- WiFi 802.11 b/g/n;
- Wi-Fi Direct, soft-AP;
- Serial Peripheral Interface (SPI);
- Analog to Digital Converter (ADC);
- Pulse Width Modulation (PWM);
- +19.5 dBm output power in 802.11b mode;
- Deep sleep: consumes less than 10 mA;

Development board features:

- Supply voltage:
  - 3.3 V on a pin marked as **3.3 V**;
  - 5 V via micro USB;
  - 5 V – 9 V via **VIN** pin (regulated by AMS1117);
- 9 digital pins (D0 – D8);
- 1 analog pin;
- USB driver CH340;
- Breadboard friendly;
- 3 x 3.3 V outputs;

The development board included in this kit is equipped with the powerful module ESP8266. It features WiFi 802.11 b/g/n, 9 digital pins, multiple 3.3 V output pins, special functions such as Pulse Width Modulation (**PWM**), Analog-to-Digital Converter (**ADC**), Inter-Integrated Circuit (**I^{2}C**) or Serial Peripheral Interface (**SPI**).

CH340 is the driver used to communicate with the computer.

Due to its reduced size, the board is breadboard friendly, which means that building a project from scratch is easy for a beginner as there is no need to solder the parts together.

There are 9 digital pins available (**D0 – D8**), but some of them (**D0, D3, D4, D7, D8**) are linked to different components mounted on the PCB and it is recommended not to use them.

2. Install and configure Arduino IDE

2.1 Download and install Arduino IDE

The Arduino Integrated Development Environment (IDE) is the software side of the Arduino platform. It is used to program development boards, such as Arduino or ESP8266 based ones.

It is available for free on their official website and it provides support for most of the Operating Systems, such as Windows, Mac OS X and Linux. The environment is open-source and it is based on Processing and other open-source software.

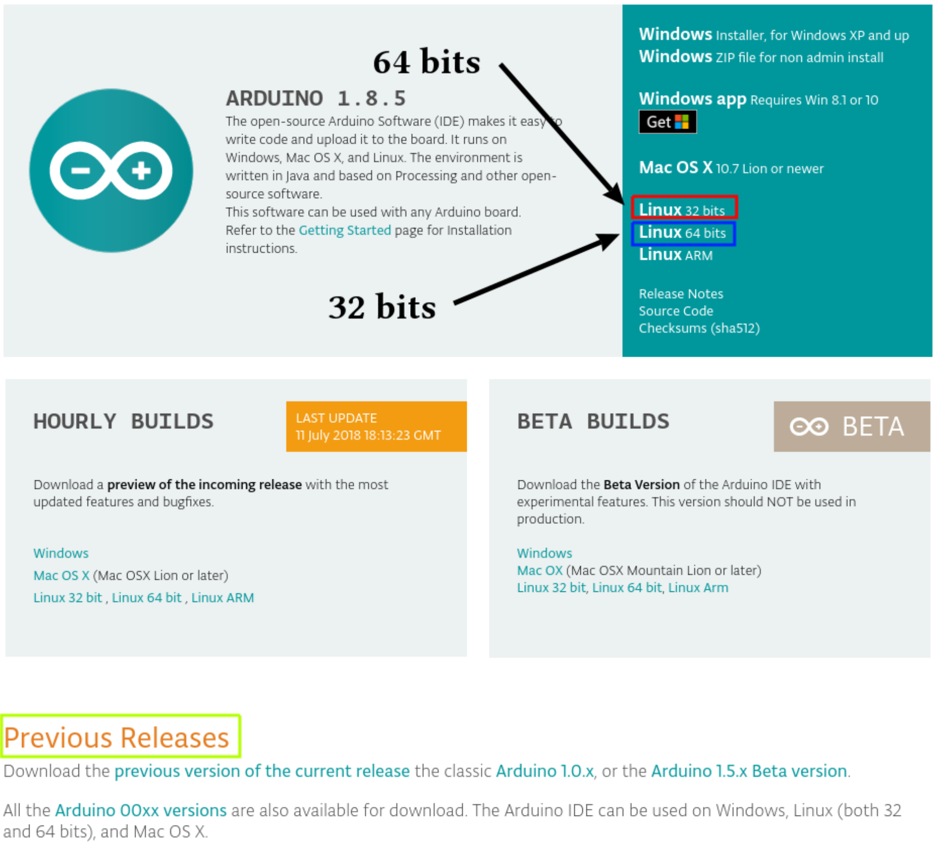
It is recommended to download the version 1.8.5 in order to be compatible with this guide. Other versions may have a different layout than the one used to create this tutorial.

2.2.1 For Linux users

Download

You can download Arduino IDE from https://www.arduino.cc/en/Main/Software.

Probably you have a **64 bits** Linux based operating system, but, if you are running a 32 bits operating system, select **Linux 32 bits** (red rectangle), or **Linux 64 bits** (blue rectangle), if you are running a 64 bits operating system.



ARDUINO 1.8.5

The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Processing and other open-source software.
This software can be used with any Arduino board.
Refer to the Getting Started page for Installation instructions.

Windows Installer, for Windows XP and up
Windows ZIP file for non admin install

Windows app Requires Win 8.1 or 10
Get

Mac OS X 10.7 Lion or newer

Linux 32 bits
Linux 64 bits
Linux ARM

Release Notes
Source Code
Checksums (sha512)

HOURLY BUILDS

LAST UPDATE
11 July 2018 18:13:23 GMT

Download a **preview of the incoming release** with the most updated features and bugfixes.

Windows
Mac OS X (Mac OSX Lion or later)
Linux 32 bit , Linux 64 bit , Linux ARM

BETA BUILDS

BETA

Download the **Beta Version** of the Arduino IDE with experimental features. This version should NOT be used in production.

Windows
Mac OX (Mac OSX Mountain Lion or later)
Linux 32 bit, Linux 64 bit, Linux Arm

Previous Releases

Download the previous version of the current release the classic Arduino 1.0.x, or the Arduino 1.5.x Beta version.

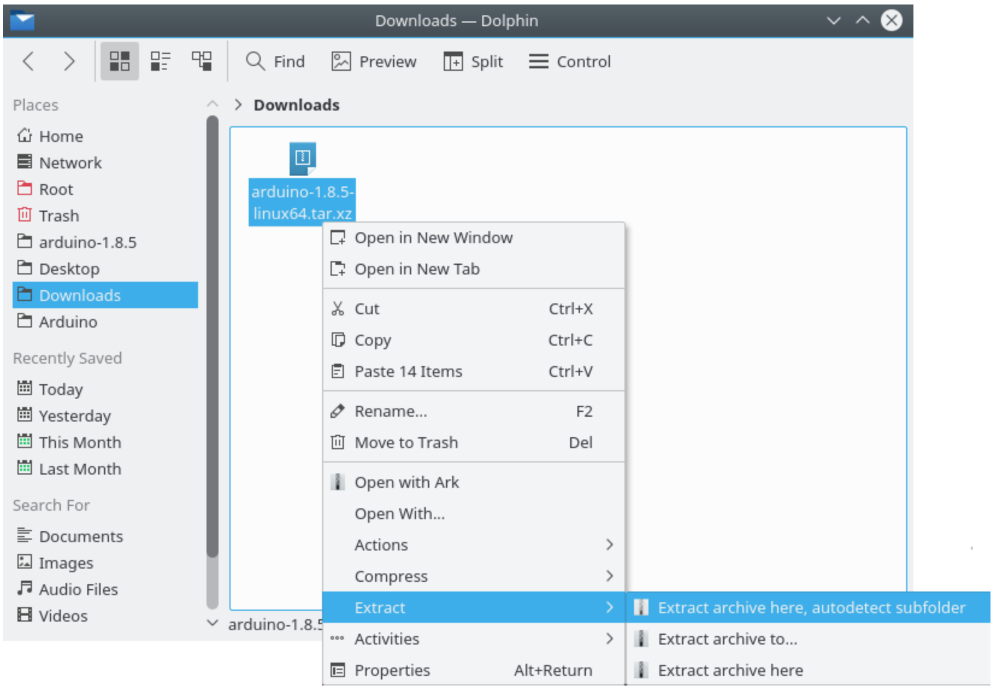
All the Arduino 00xx versions are also available for download. The Arduino IDE can be used on Windows, Linux (both 32 and 64 bits), and Mac OS X.

Also, if the version was updated, check **Previous Releases** to make sure you have the same version, 1.8.5, for best compatibility with this guide.

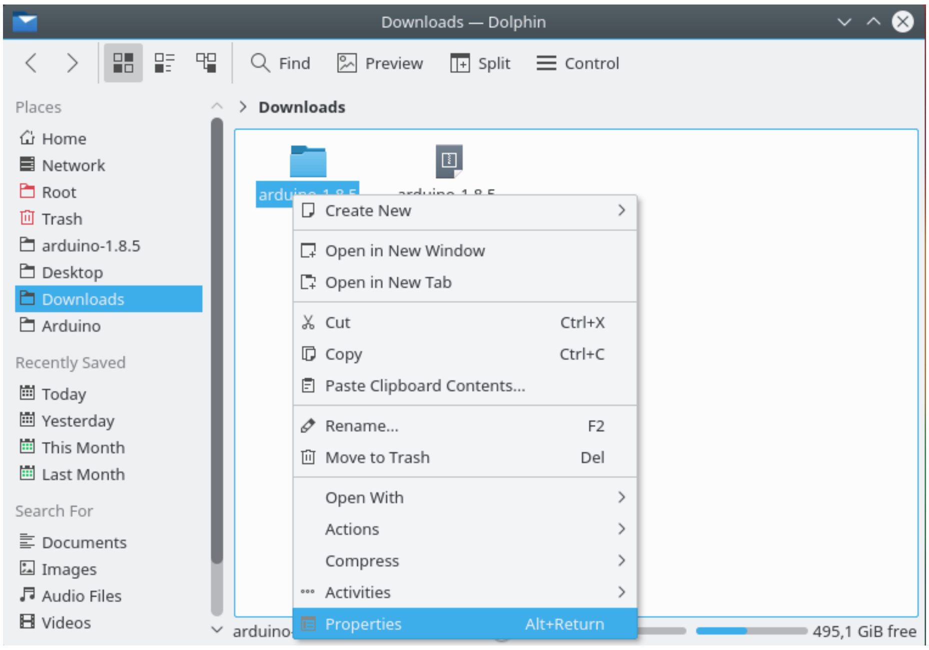
Install

a) Find the folder which contains the **.tar.xz** file downloaded from **www.arduino.com**;

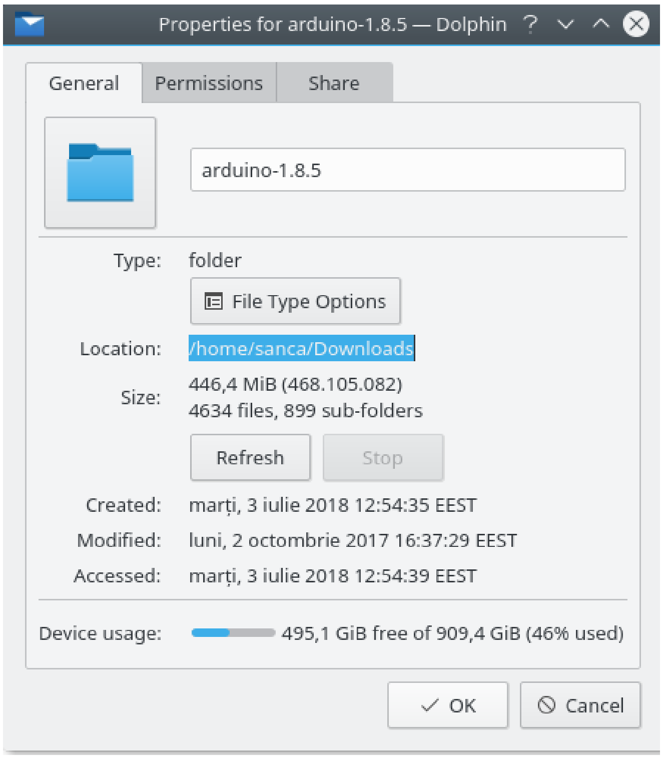
b) Right-click on the archive and extract the files in a folder of your choice;



c) Right-click on the folder containing the files extracted and go to Properties;



d) Copy the location of the folder

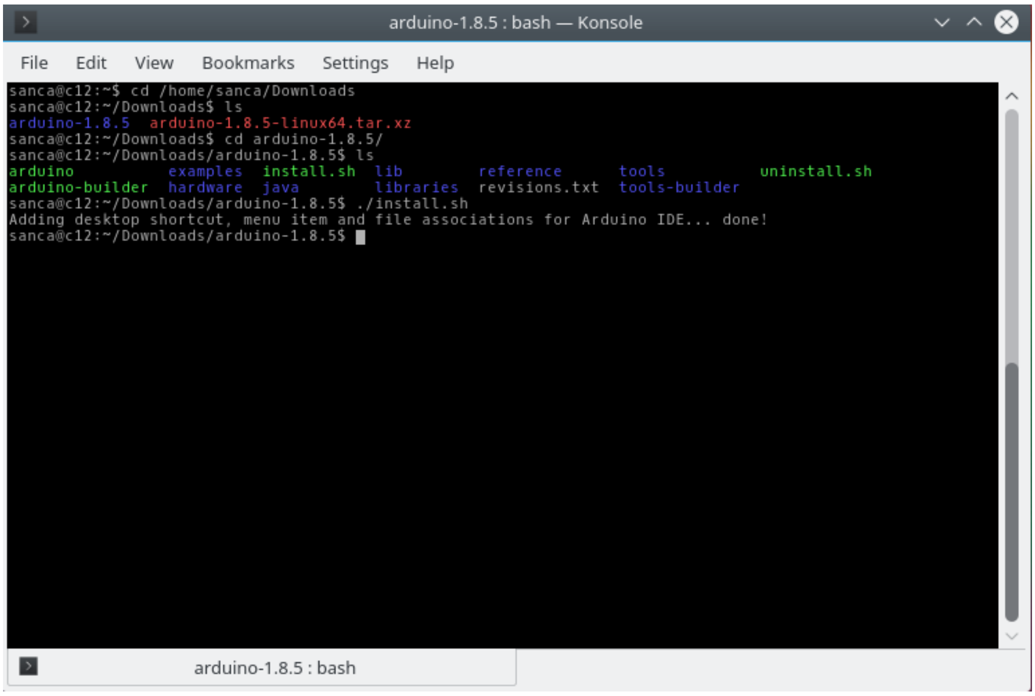

Properties for arduino-1.8.5 — Dolphin
General
Permissions
Share
arduino-1.8.5
Type: folder
File Type Options
Location: /home/sanca/Downloads
Size: 446,4 MiB (468.105.082)
4634 files, 899 sub-folders
Refresh
Stop
Created: marți, 3 iulie 2018 12:54:35 EEST
Modified: luni, 2 octombrie 2017 16:37:29 EEST
Accessed: marți, 3 iulie 2018 12:54:39 EEST
Device usage: 495,1 GiB free of 909,4 GiB (46% used)
OK
Cancel

e) Now, open a terminal and **cd** to that location;

f) List all the files in that folder using **ls;**

g) And **cd** to the folder containing the files extracted;

h) And now just type **./install.sh** and press **Enter;**



```
sanca@c12:~$ cd /home/sanca/Downloads
sanca@c12:~/Downloads$ ls
arduino-1.8.5  arduino-1.8.5-linux64.tar.xz
sanca@c12:~/Downloads$ cd arduino-1.8.5/
sanca@c12:~/Downloads/arduino-1.8.5$ ls
arduino          examples  install.sh  lib        reference      tools          uninstall.sh
arduino-builder  hardware  java        libraries  revisions.txt  tools-builder
sanca@c12:~/Downloads/arduino-1.8.5$ ./install.sh
Adding desktop shortcut, menu item and file associations for Arduino IDE... done!
sanca@c12:~/Downloads/arduino-1.8.5$ 
```

i) You can close the terminal;

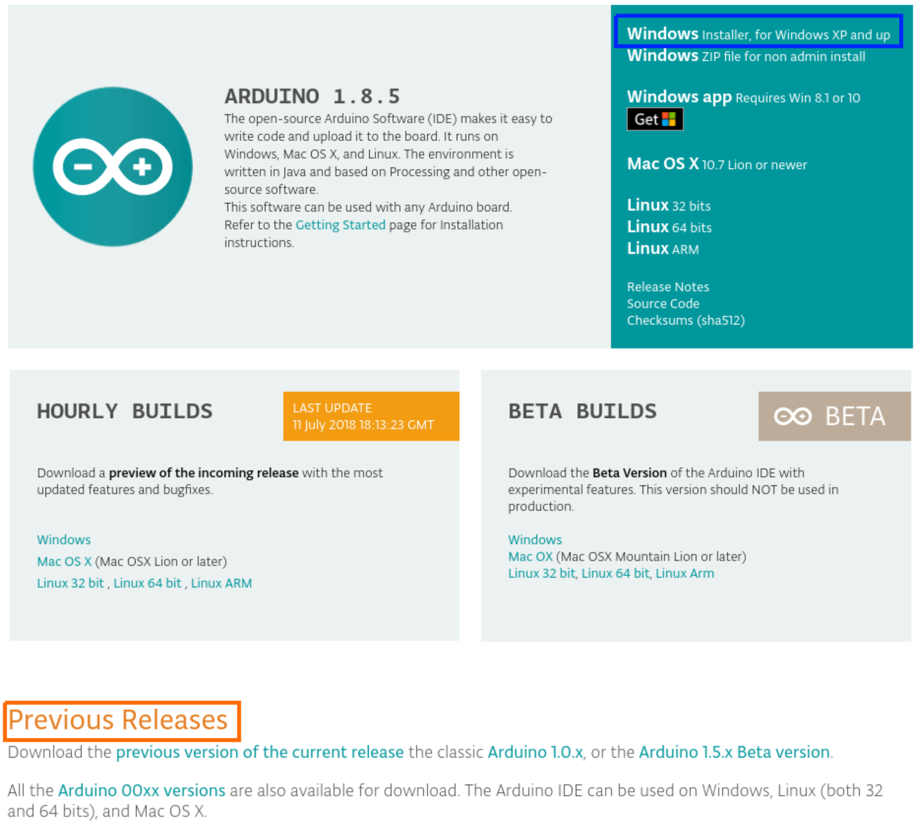
j) Congratulations! You have successfully installed Arduino IDE on your linux-based computer.

2.2.2 For Windows users

Download

You can download Arduino IDE from https://www.arduino.cc/en/Main/Software.

Now, click on **Windows** (blue rectangle).



ARDUINO 1.8.5

The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Processing and other open-source software.

This software can be used with any Arduino board.

Refer to the Getting Started page for Installation instructions.

Windows Installer, for Windows XP and up
Windows ZIP file for non admin install

Windows app Requires Win 8.1 or 10
Get

Mac OS X 10.7 Lion or newer

Linux 32 bits
Linux 64 bits
Linux ARM

Release Notes
Source Code
Checksums (sha512)

HOURLY BUILDS

LAST UPDATE
11 July 2018 18:13:23 GMT

Download a **preview of the incoming release** with the most updated features and bugfixes.

Windows
Mac OS X (Mac OSX Lion or later)
Linux 32 bit , Linux 64 bit , Linux ARM

BETA BUILDS

BETA

Download the **Beta Version** of the Arduino IDE with experimental features. This version should NOT be used in production.

Windows
Mac OX (Mac OSX Mountain Lion or later)
Linux 32 bit, Linux 64 bit, Linux Arm

Previous Releases

Download the previous version of the current release the classic Arduino 1.0.x, or the Arduino 1.5.x Beta version.

All the Arduino 00xx versions are also available for download. The Arduino IDE can be used on Windows, Linux (both 32 and 64 bits), and Mac OS X.

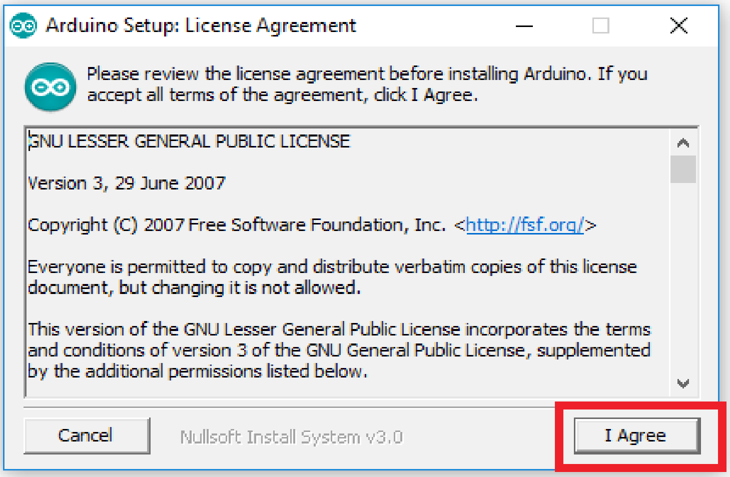
Also, if the version was updated, check **Previous Releases** to make sure you have the same version, 1.8.5, for best compatibility with this guide.

Install

a) Find the file you have just downloaded from **www.arduino.com**.

b) Double click on it.

c) Read the “License Agreement”, then click on **“I agree”** if you accept the agreement.


Arduino Setup: License Agreement

Please review the license agreement before installing Arduino. If you accept all terms of the agreement, click I Agree.

GNU LESSER GENERAL PUBLIC LICENSE

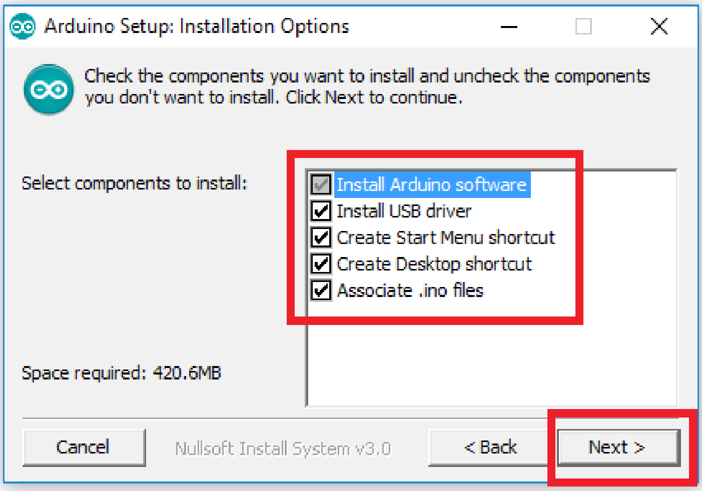
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

d) Select **all components** to install, then click on **“Next”.**


Arduino Setup: Installation Options

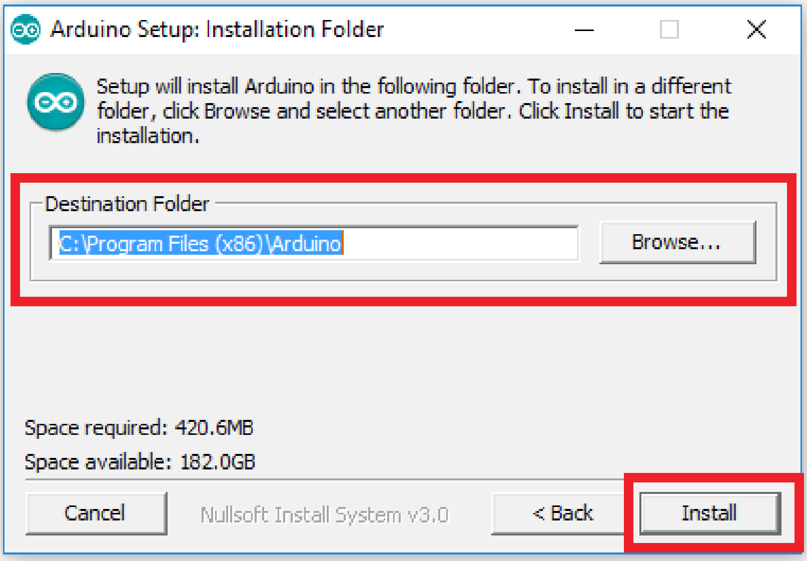
Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

- [✓] Install Arduino software
- [✓] Install USB driver
- [✓] Create Start Menu shortcut
- [✓] Create Desktop shortcut
- [✓] Associate .ino files

Space required: 420.6MB

e) Select the location of the installation. **Do not forget it**.


Arduino Setup: Installation Folder

Setup will install Arduino in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

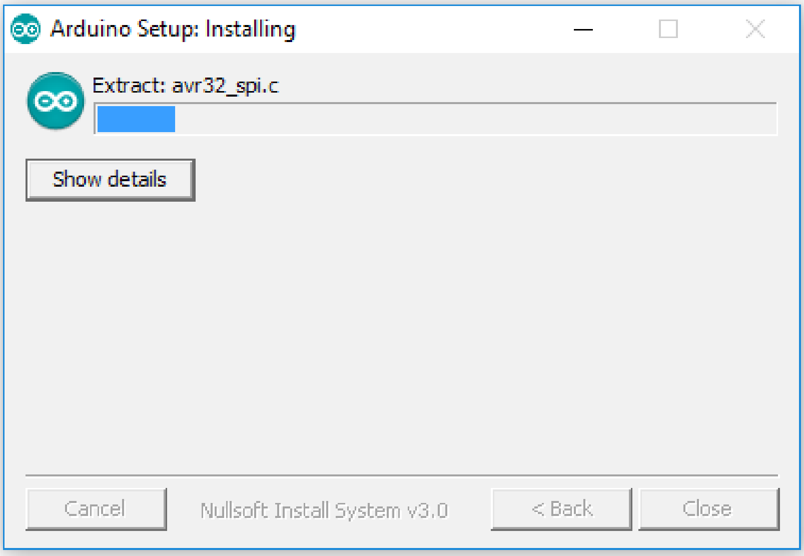
Destination Folder

C:\Program Files (x86)\Arduino

Space required: 420.6MB

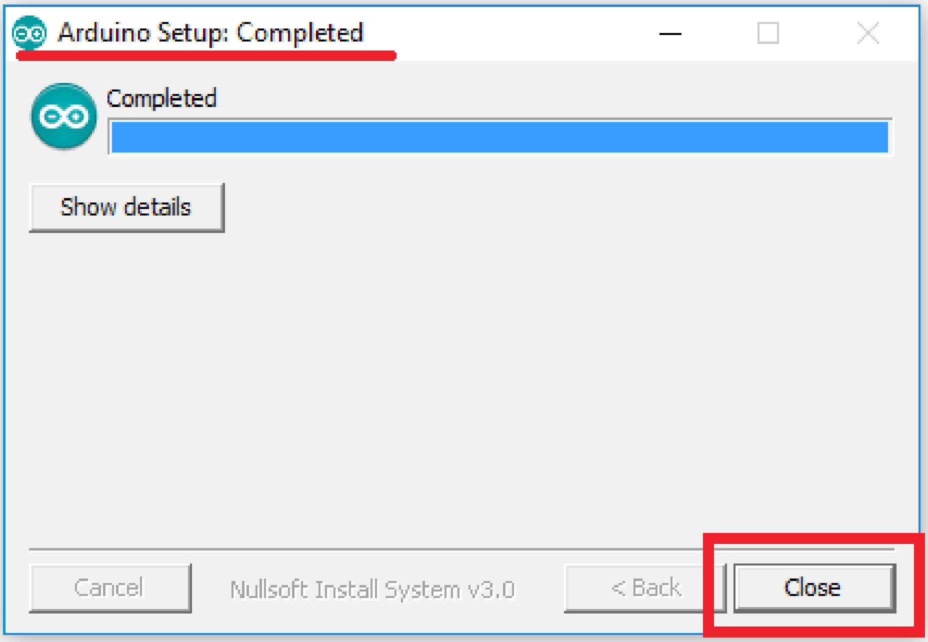
Space available: 182.0GB

f) Wait until the installation is finished.


Arduino Setup: Installing

Extract: avr32_spi.c

g) When the setup is finished, click on the **Close** button.

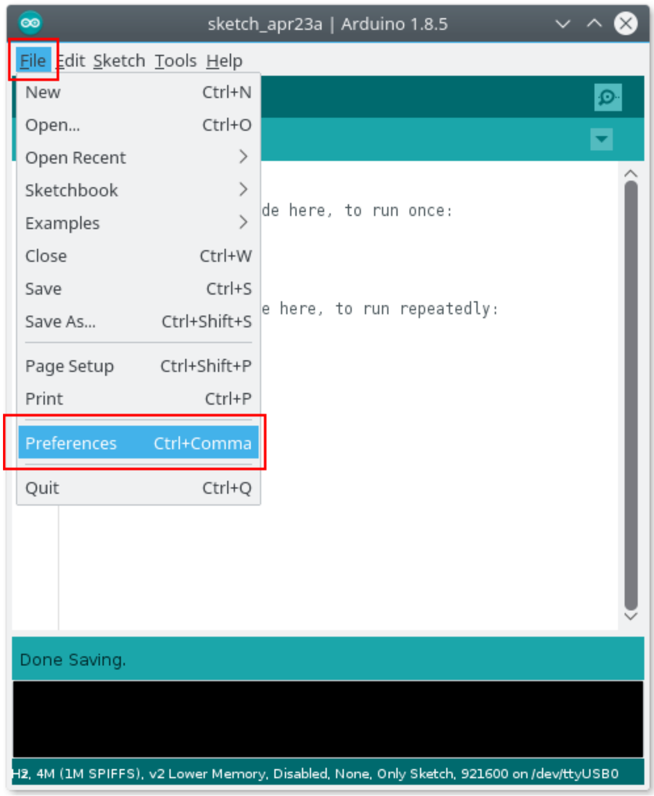


h) Congratulations! You havesuccessfully installed **Arduino IDE** on your computer.

2.2 Add ESP8266 based boards into the Arduino IDE

Uploading code to the development board is not supported by default. In order to be able to program the board, you have to follow these steps:

1) Open **Arduino IDE**.
2) Click on **File tab** and select **Preferences**, or you can use the shortcut **CTRL + COMMA.**

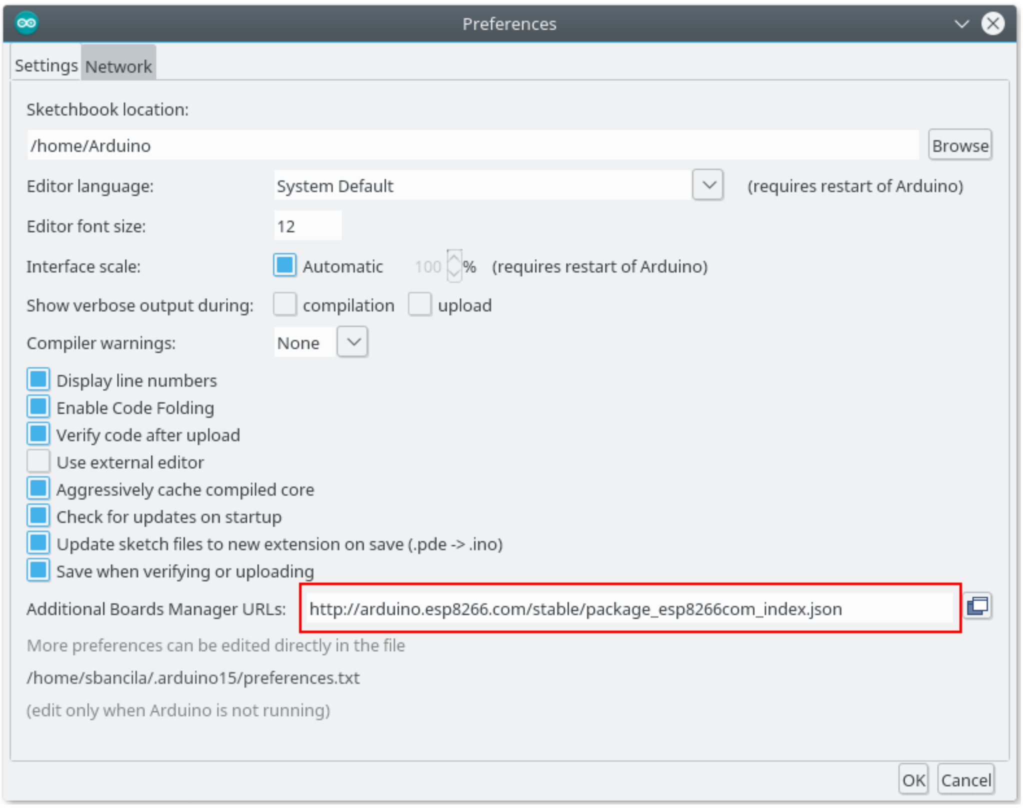


```
de here, to run once:

e here, to run repeatedly:
```

H2, 4M (1M SPIFFS), v2 Lower Memory, Disabled, None, Only Sketch, 921600 on /dev/ttyUSB0

3) In the Preferences window, search for **“Additional Boards Manager URLs”** and copy – paste the following link:

http://arduino.esp8266.com/stable/package_esp8266com_index.json



Sketchbook location: /home/Arduino

Editor language: System Default (requires restart of Arduino)

Editor font size: 12

Interface scale: [✓ Automatic] 100 % (requires restart of Arduino)

Show verbose output during: ☐ compilation ☐ upload

Compiler warnings: None

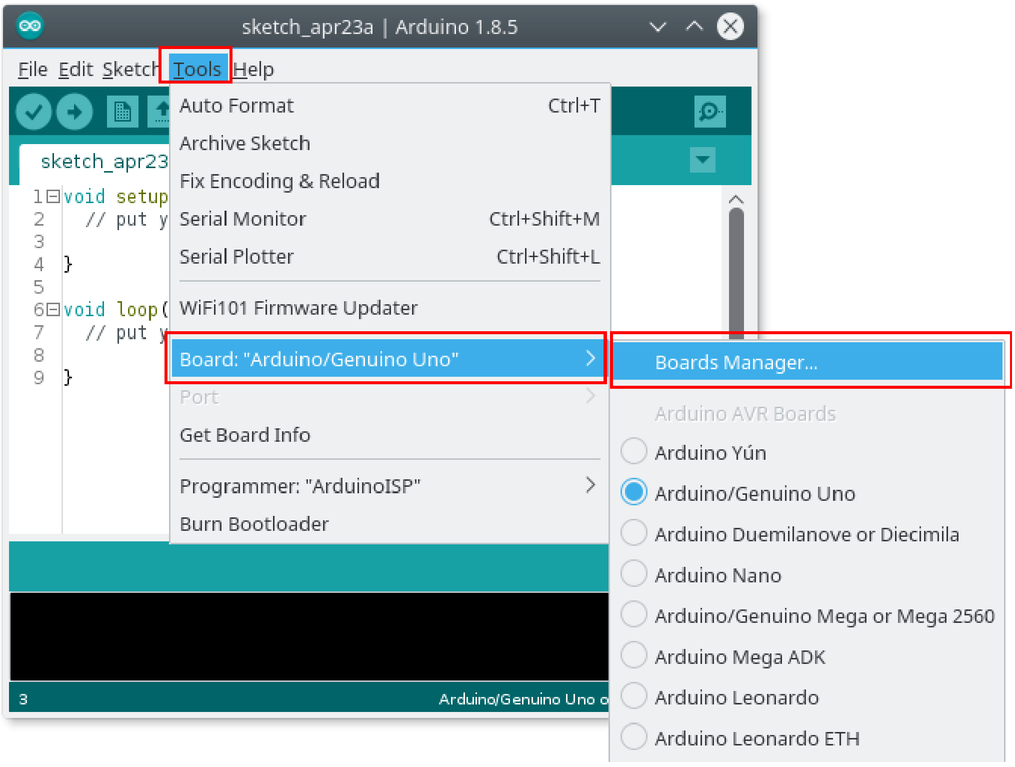
[✓] Display line numbers
[✓] Enable Code Folding
[✓] Verify code after upload
[ ] Use external editor
[✓] Aggressively cache compiled core
[✓] Check for updates on startup
[✓] Update sketch files to new extension on save (.pde -> .ino)
[✓] Save when verifying or uploading

Additional Boards Manager URLs: http://arduino.esp8266.com/stable/package_esp8266com_index.json

More preferences can be edited directly in the file
/home/sbancila/.arduino15/preferences.txt
(edit only when Arduino is not running)

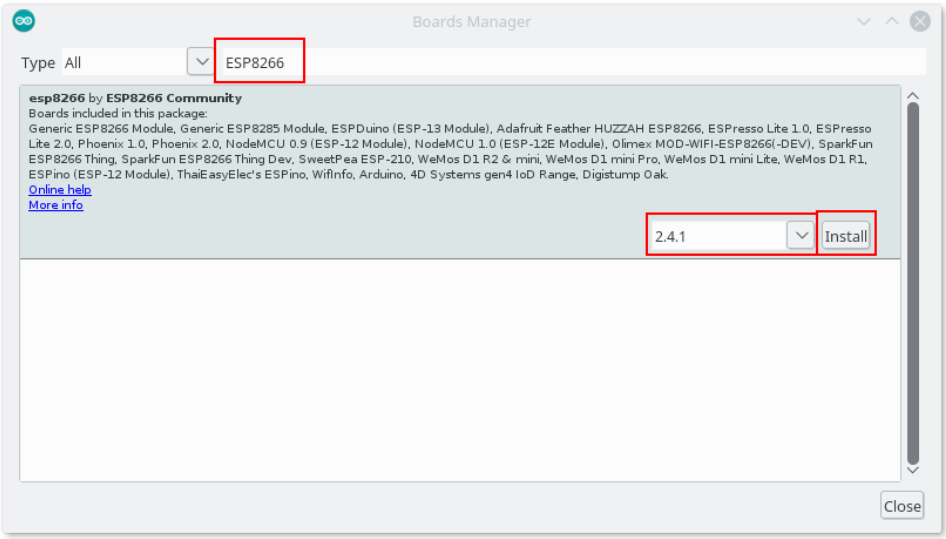
After you entered the code in the text area, click on the **OK** button and the window should close.

4) In the **Arduino IDE**, click on the **TOOLS** tab, hover the cursor over the selected board, then click on **Boards Manager...**



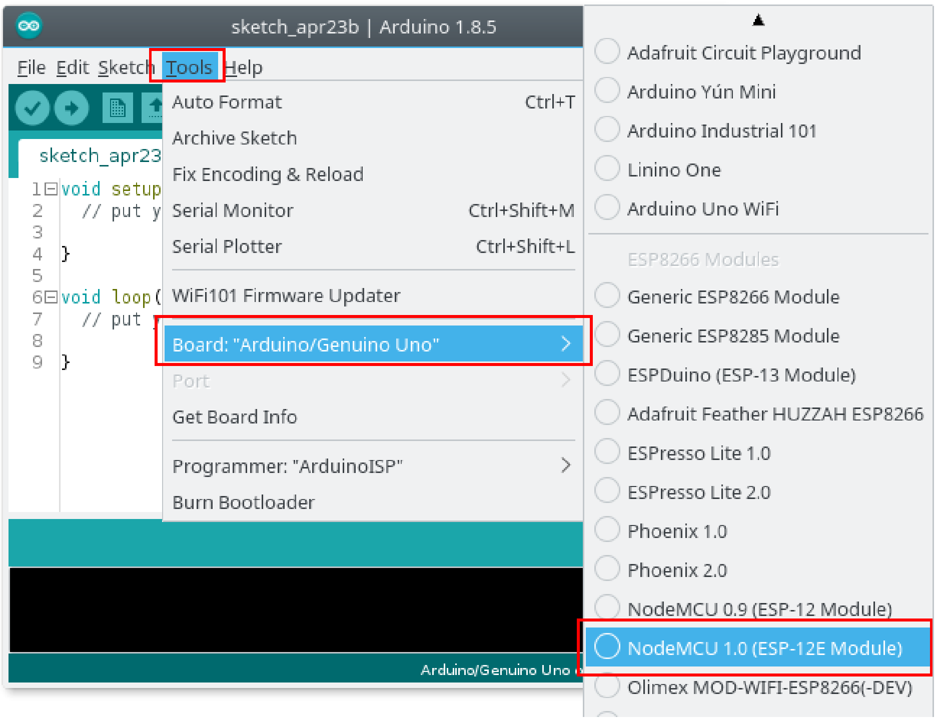
```
void setup
  // put y
}

void loop(
  // put y

}
```

5) In the opened window, search for the **ESP8266** boards, and install the latest version available. At the moment of writing this guide, the latest version was **2.4.1**.



esp8266 by **ESP8266 Community**
Boards included in this package:
Generic ESP8266 Module, Generic ESP8285 Module, ESPDuino (ESP-13 Module), Adafruit Feather HUZZAH ESP8266, ESPresso Lite 1.0, ESPresso Lite 2.0, Phoenix 1.0, Phoenix 2.0, NodeMCU 0.9 (ESP-12 Module), NodeMCU 1.0 (ESP-12E Module), Olimex MOD-WIFI-ESP8266(-DEV), SparkFun ESP8266 Thing, SparkFun ESP8266 Thing Dev, SweetPea ESP-210, WeMos D1 R2 & mini, WeMos D1 mini Pro, WeMos D1 mini Lite, WeMos D1 R1, ESPino (ESP-12 Module), ThaiEasyElec's ESPino, WifInfo, Arduino, 4D Systems gen4 IoD Range, Digistump Oak.
Online help
More info
2.4.1 Install

6) Open the **Tools** tab, hover over **Board** and select **NODEMCU 1.0 (ESP – 12E module).**

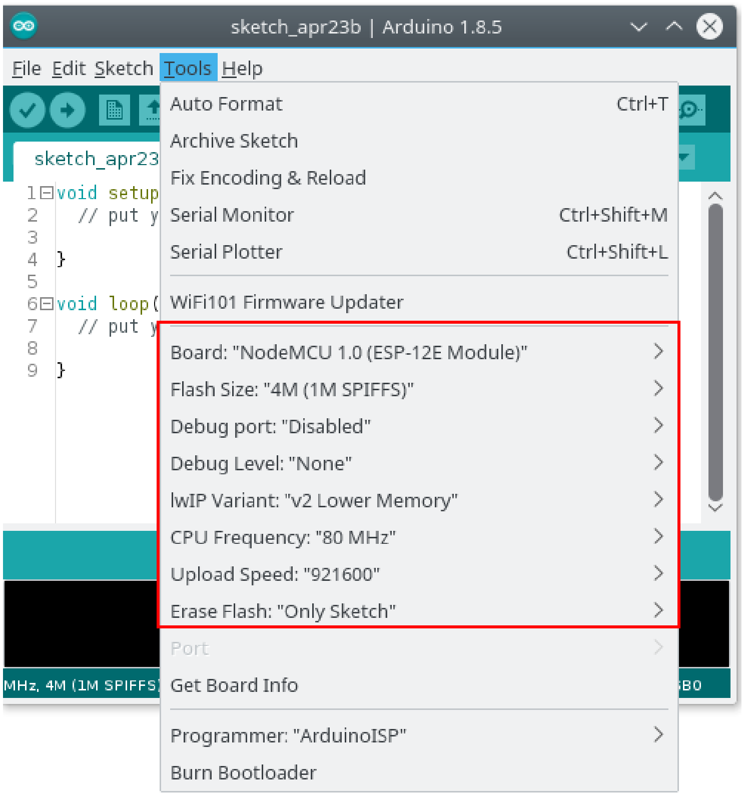


```
void setup
  // put y
}

void loop(
  // put
}
```

7) Configure the settings as shown below. This is not the only working configuration.

Here are the settings:

- Board: NodeMCU 1.0 (ESP-12E Module)
- Flash Size: 4M (1M SPIFFS)
- Debug port: Disabled
- Debug level: None
- IwIP: v2 Lower Memory
- CPU Frequency: 80 Mhz
- Upload Speed: 921600
- Erase flash: Only Sketch


sketch_apr23b | Arduino 1.8.5

File Edit Sketch Tools Help

```
void setup() {
  // put y
}

void loop() {
  // put y
}
```

Auto Format	Ctrl+T
Archive Sketch	
Fix Encoding & Reload	
Serial Monitor	Ctrl+Shift+M
Serial Plotter	Ctrl+Shift+L
WiFi101 Firmware Updater	
Board: "NodeMCU 1.0 (ESP-12E Module)"	
Flash Size: "4M (1M SPIFFS)"	
Debug port: "Disabled"	
Debug Level: "None"	
lwIP Variant: "v2 Lower Memory"	
CPU Frequency: "80 MHz"	
Upload Speed: "921600"	
Erase Flash: "Only Sketch"	
Port	
Get Board Info	
Programmer: "ArduinoISP"	
Burn Bootloader	

MHz, 4M (1M SPIFFS) ... B0

8) Congratulations. Now, the **Arduino IDE** is configured and ready to have code uploaded to the development board.

3. Add Libraries

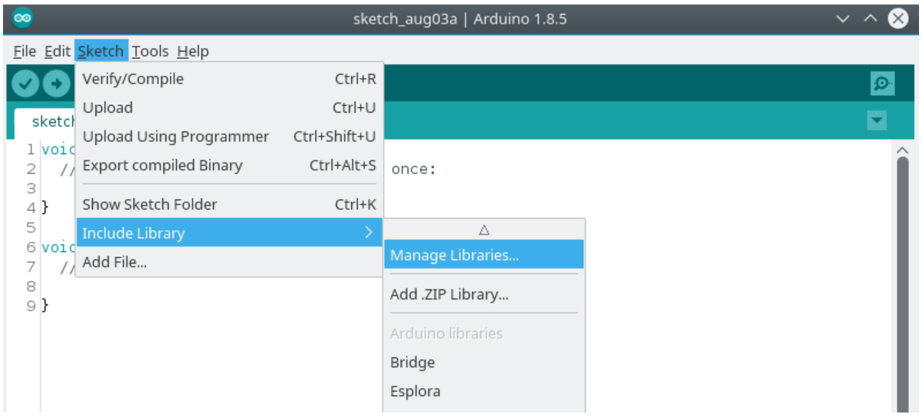
Once you are comfortable with the Arduino IDE software, you may want to extend the abilities of your development board with additional libraries. Throughout this guide we will need some libraries for connecting different sensors and, at the right moment, you will learn how to create your own library and add it as a **.zip** archive.

3.1 What are Libraries

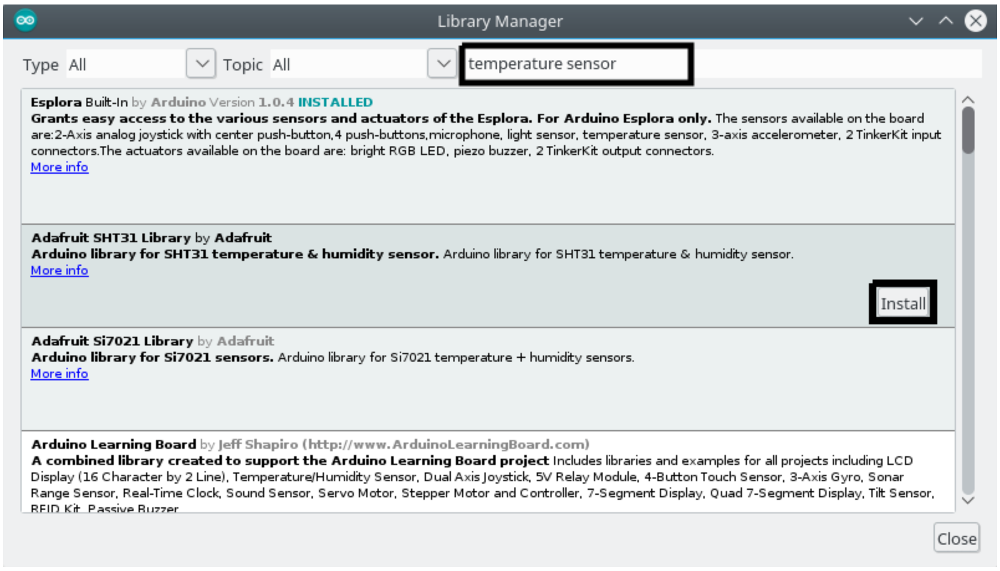
Libraries are a collection of code that makes it easy for you to connect to a sensor, display, module, etc, through predefined functions and commands for a specific programming language. There are hundreds of additional libraries on the Internet written by different people.

3.2 Installing a library

To install a new library into your **Arduino IDE** you can use the Library Manager. Open **Arduino IDE** and click on the **Sketch** menu and then go to **Include Library > Manage Libraries**.


sketch_aug03a | Arduino 1.8.5
File Edit Sketch Tools Help
Verify/Compile Ctrl+R
Upload Ctrl+U
Upload Using Programmer Ctrl+Shift+U
Export compiled Binary Ctrl+Alt+S
Show Sketch Folder Ctrl+K
Include Library
Add File...
Manage Libraries...
Add .ZIP Library...
Arduino libraries
Bridge
Esplora

In the opened window, you can find all the libraries available to install. These are libraries written by **Arduino**, or partner companies, but there are libraries written even by contributors and uploaded so that anyone can use them. Search for any library and hit **Install**.



Esplora Built-In by **Arduino** Version **1.0.4** **INSTALLED**
Grants easy access to the various sensors and actuators of the Esplora. For Arduino Esplora only. The sensors available on the board are:2-Axis analog joystick with center push-button,4 push-buttons,microphone, light sensor, temperature sensor, 3-axis accelerometer, 2 TinkerKit input connectors.The actuators available on the board are: bright RGB LED, piezo buzzer, 2 TinkerKit output connectors.
More info

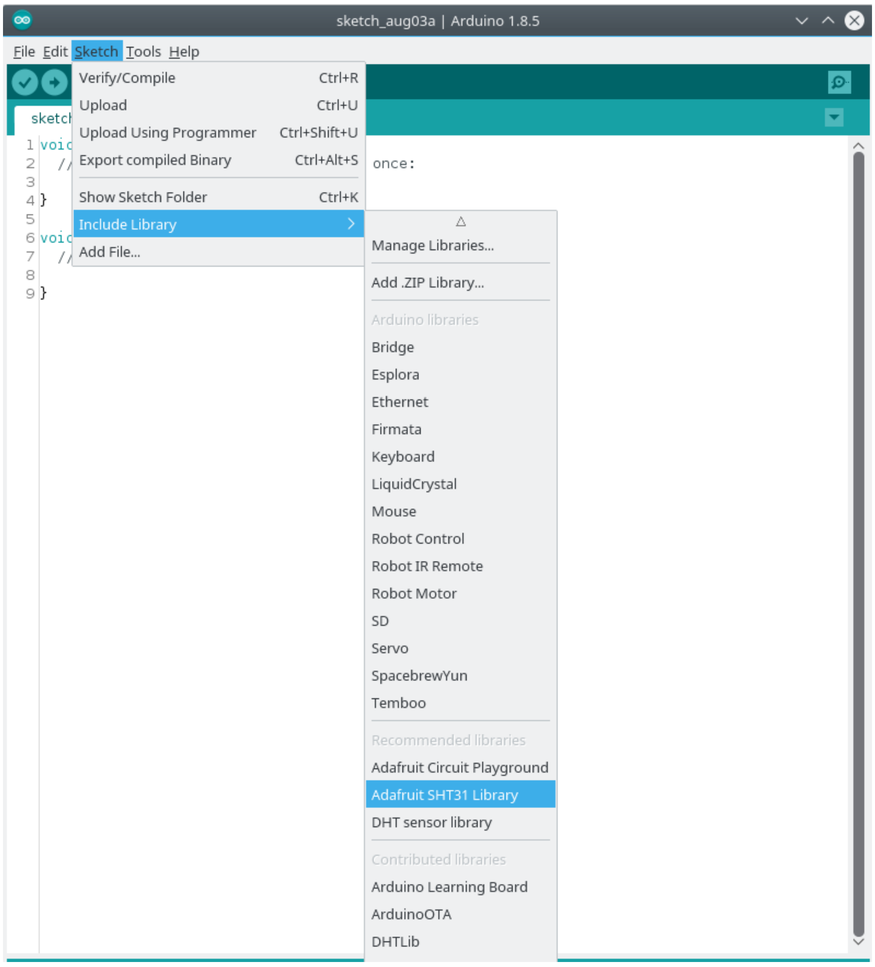
Adafruit SHT31 Library by **Adafruit**
Arduino library for SHT31 temperature & humidity sensor. Arduino library for SHT31 temperature & humidity sensor.
More info
Install

Adafruit Si7021 Library by **Adafruit**
Arduino library for Si7021 sensors. Arduino library for Si7021 temperature + humidity sensors.
More info

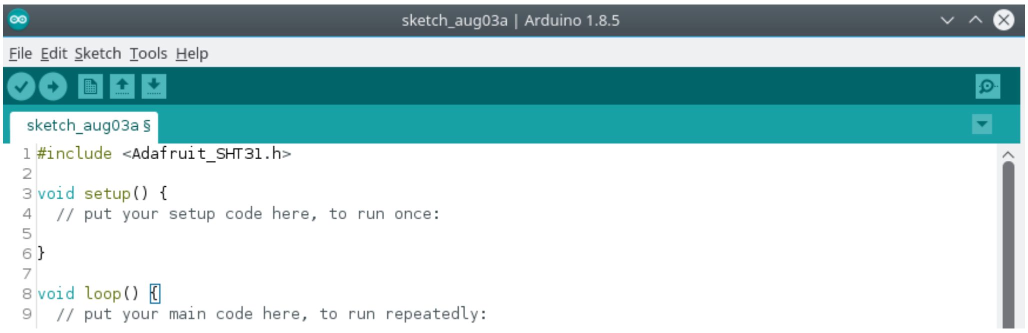
Arduino Learning Board by **Jeff Shapiro (http://www.ArduinoLearningBoard.com)**
A combined library created to support the Arduino Learning Board project Includes libraries and examples for all projects including LCD Display (16 Character by 2 Line), Temperature/Humidity Sensor, Dual Axis Joystick, 5V Relay Module, 4-Button Touch Sensor, 3-Axis Gyro, Sonar Range Sensor, Real-Time Clock, Sound Sensor, Servo Motor, Stepper Motor and Controller, 7-Segment Display, Quad 7-Segment Display, Tilt Sensor, RFID Kit, Passive Buzzer

Close

After installing the library, to include it in your code go to **Sketch** > **Include Library** and select the libray you have just installed.



And in your code you should see something like this:



```
#include <Adafruit_SHT31.h>

void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:
```

4. Lesson 1: Blink an LED

4.1 Overview

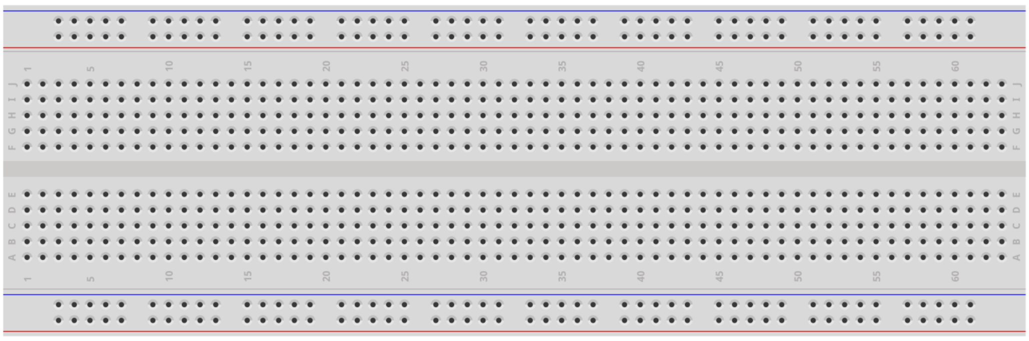
In this lesson you will learn how to use a breadbord, how to place components on it and how to control an LED.

4.2 Components required

- Development board;
- 1 x LED;
- 1 x 150 Ω resistor;
- Breadboard 830p;
- 2 x male-to-male jumper wires;
- Micro USB – Type A USB Cable;

4.3 Component Introduction

➢ Breadboard 830p



A breadboard enables you to prototype circuits quickly, without having to solder the components together. There are multiple types of breadboards that differ in size and configuration, but the breadboard included in this kit has 830 points.

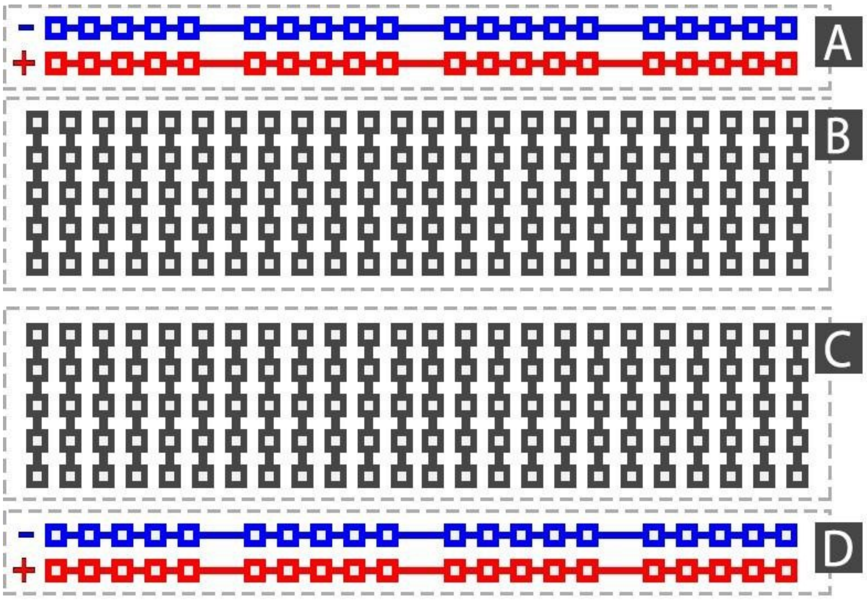
A breadboard consists of a block of plastic with multiple holes. Inside the block there are strips of metal that provide electrical connection between holes.

While a breadboard is very good for prototypes as you can simply insert a component into it in order to have a connection. Sometimes these connections are not very stable and can make components work improperly.

A picture with a similar breadboard included in the kit can be seen below.

On the edges (area **A** and **D**) there are the **rails.** They are used to provide power and have suggestive colours: blue for ground and red for VCC. Please note that the blue and red lines are not connected between them.

In the middle (area **B** and **C**), the space is used to mount components such as LED, sensors, resistors etc. A deep channel running down the middle indicates that there is a break in connections there, meaning that you can push a chip in with the legs at either side of the channel without connecting them together.



➢ Light Emitting Diode

An **L**ight **E**mitting **D**iode (LED) is a component used to provide visual feedback, as they use very little electricity (about 15-30 mA for a standard 5 mm LED) and they can last forever (unless you burn them). One of the most common type of LED is the 5mm red LED. The size of 5 mm refers to the diameter of the LED, and red represents the colour it emits when it is powered.

An LED is always used with a resistor in series in order to limit the amount of current flowing through it, as otherwise it will burn out and you can't repair it. The formula to calculate the resistance needed is the following: $R=\frac{Vcc-V_{LED}}{I_{LED}}$

where:

- Vcc is the supply voltage (this development board has only 3.3V outputs);

- V_{LED} is the LED forward voltage (2 V for red, 2.1 V for green, 3.2 V for blue)

- I_{LED} is the LED forward curent (15 mA for red, 20 mA for green, 25 mA for blue)

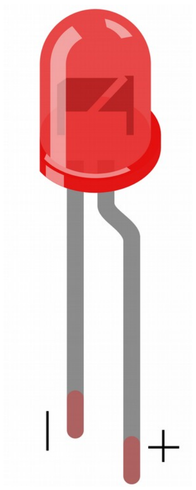
Example: Let's say that you want to power a green led from 3.3 V. The led has the forward voltage of about 2.1 V and the forward current of about 10 mA (max current on each pin of the board is 12mA). The resistor required has the following value:

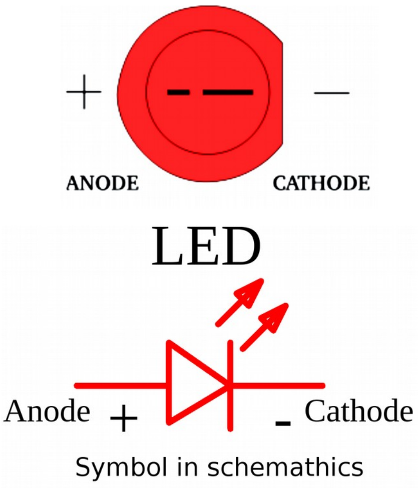
$$R=\frac{3.3\,V-2.1\,V}{10\cdot10^{-3}\,A}=\frac{1.2\,V}{10\cdot10^{-3}\,A}=120\,\Omega$$

So, the required resistor needs to have a resistance of 120 Ω. Sometimes, you won't find the desired resistance and it is recommended to pick a resistor with a higher resistance. In this case, you should pick a 150 Ω.

IMPORTANT! When you power an LED, the polarity matters. It means you have to correctly identify the anode (positive lead of the LED) and the cathode (negative lead of the LED). Depending on the LED used, there are multiple ways to determine how to power it:

- If the leads don't have the same length, it means that the longer one is the anode (positive terminal) and the shorter one is cathode (negative terminal) as shown below.
- If the leads have the same length, you have to check the body of the LED as there is a flat edge and that is the place where the negative lead enters into the body, as shown below.



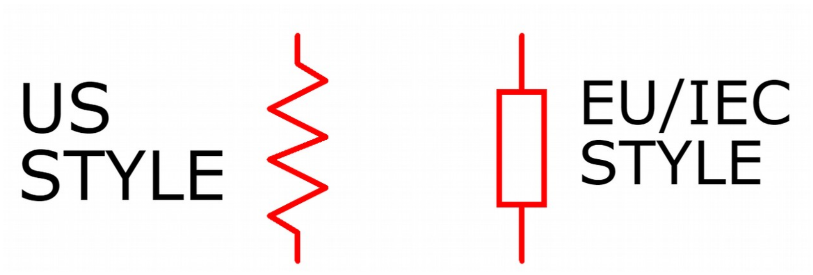


➢ Resistor

A resistor is a passive two-terminal electrical component that is used to reduce the current flow, adjust signal levels, divide voltages, etc. The higher the value of the resistor, the less current will flow through it. Unlike the LEDs, resistors don't have polarity so you can connect them either way around.

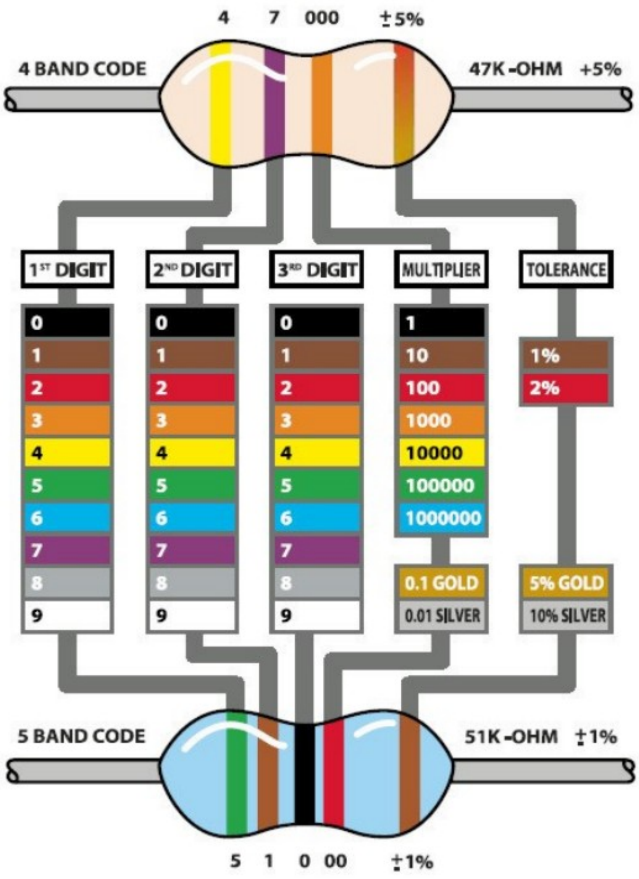
The unit of resistance is called **OHM**, which is usually shortened to **Ω** (the Greek letter Omega). Sometimes, you can see another other latin letter next to the Ω, such as kΩ (1kΩ = 1,000Ω) or MΩ (1MΩ = 1,000kΩ = 1,000,000Ω).

In schematics, there are two common ways to represent a resistor: US style and EU style. Below you have an example.



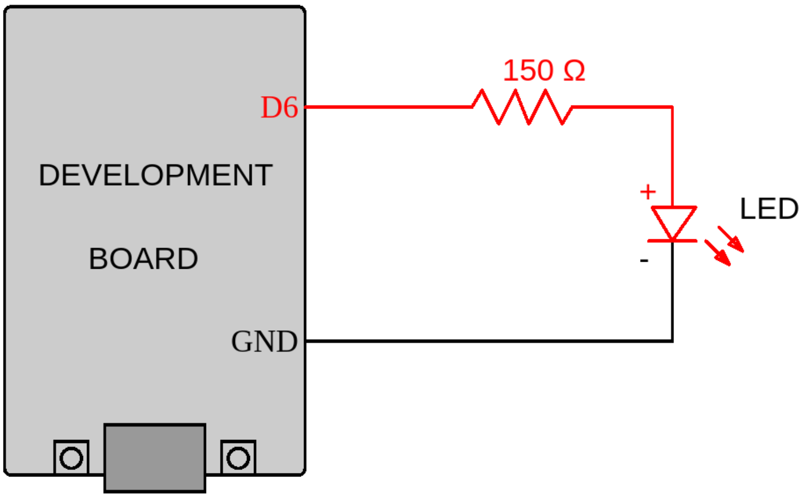
Most of the resistors have the same body, so it is useful to know how to determine the value of a resistor. There are 2 ways:

- You can simply use a multimeter to measure the resistance, then to approximate to the closest common value.
- You can decode the pattern printed on them.

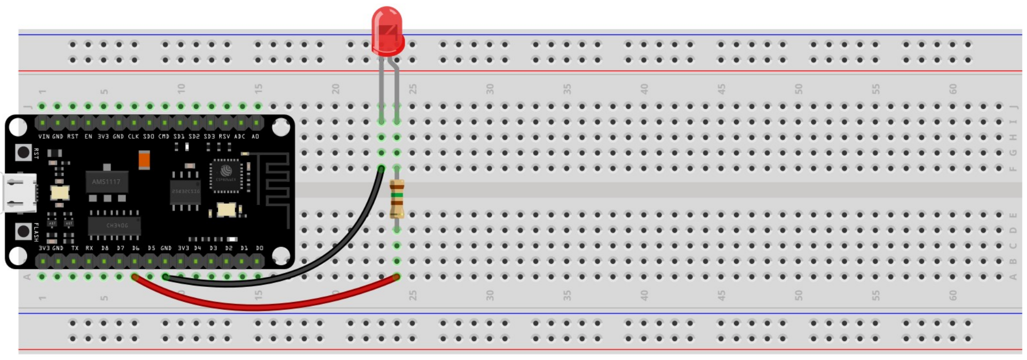


4.4 Connection

In order to blink an LED, you have to connect the negative terminal (please check above to see how to determine it) to a **GND** pin of the development board, and the positive terminal to a digital pin. In this example, we are going to use **D6**, as shown below.



Next, you can see a visual representation of how to connect the LED to the breadboard and development board.



The board runs at 3.3 V, which is enough to power the LED. In order to power the board, you have to plug the USB cable to a computer or to a travel adapter. It has an onboard 3.3 V regulator, which lowers the voltage from 5 V to 3.3 V. The resistor is picked accordingly to the equation presented before (check the Resistor section). The required resistance is about 120 Ω, but we are going to pick a 150 Ω resistor as the 120 Ω is not a common value.

4.5 Code

Every Arduino or ESP8266 code is based on two main functions:

- **void setup()**
  - It runs only once, when the board starts. Usually, it is used to start the

  serial communication or different components.

- **void loop()**
  - This function will continuously run. It is used to check the state of the sensors attached to the board, or to control components.

The code required to blink the LED can be found in the folder called "**Lesson 1: Blink an LED**".

First step is to declare two variables, one to store the pin used by the LED, and another to store the time in milliseconds.

Code 4.5.1 Declaring the variables used

```
//the int variable "LED" stores the pin used by the LED
const int LED = D6;

//the int variable "delayTime" stores the time (in milliseconds) between the
blinks
//1000 milliseconds = 1 second
const int delayTime = 1000;
```

The **setup()** function runs one time when the development board is powered On. In this function we have to initialise the pin as **OUTPUT** using the function **pinMode(pin, mode)**, where **pin** is the pin used to connect something and **mode** can be **OUTPUT**, **INPUT** or **INPUT_PULLUP**.

When a pin is configured as **INPUT**, the pin is in a high-impedance state, meaning that it takes very little current to move the input pin from one state to another; this mode is used primarily when connecting sensors and we want to read from them.

On the development board there are pullup resistors that can be accessed by setting the **pinMode** as **INPUT_PULLUP**. This inverts the behavior of the **INPUT** mode, where **HIGH** means the sensor is off, and **LOW** means the sensor is on.

Pins configured as **OUTPUT** are in a low-impedance state. This means that the impedance of the pin is lowered so it can provide a substantial amount of current to other circuits.

Code 4.5.2 The setup() function

```
void setup()
{
  //the following instruction initialises the pin stored in the
  //variable LED as OUTPUT
  //this instruction lowers the impedance of the pin so it can provide
  //a higher current to other components.
  pinMode(LED, OUTPUT);
}
```

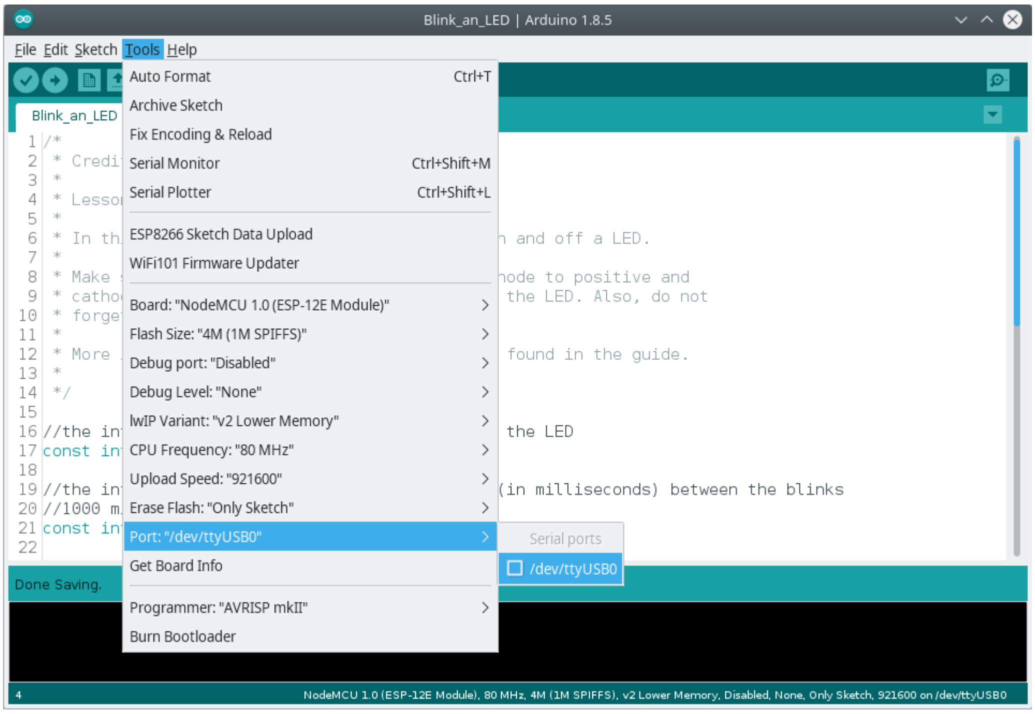
Further, the **loop** function is used to actually turn the LED on and off. Changing the state of the led is done with the instruction **digitalWrite(pin, state)**. State can be either **HIGH** (on) or **LOW** (off). **delay(time)** stops the code for a certain period of time, where time is a value in milliseconds (1000 ms = 1 s).

Code 4.5.3 The loop() function

```
void loop()
{
  //in order to turn on/off the LED, you have to use the
  //instruction digitalWrite(pin, state);
  //Parameters:
  //pin: it can be one of the 9 available on the board (from D0-D8)
  //state: it can be either HIGH, either LOW.

   //the next two instructions are used to turn off the LED and wait for 1 second
  digitalWrite(LED, LOW);
  delay(delayTime);

   //the next two instructions are used to turn on the LED and wait for 1 second
  digitalWrite(LED, HIGH);
  delay(delayTime);
}
```

Next step is to gently connect the development board to the computer via the included micro USB cable. There is only one way to plug it in, so do not use force to insert the cable.

In the Arduino IDE, click on the **Tools** tab, hover the cursor over the **Port** and select the port used by the development board. Depending on the OS used, the port can be different.

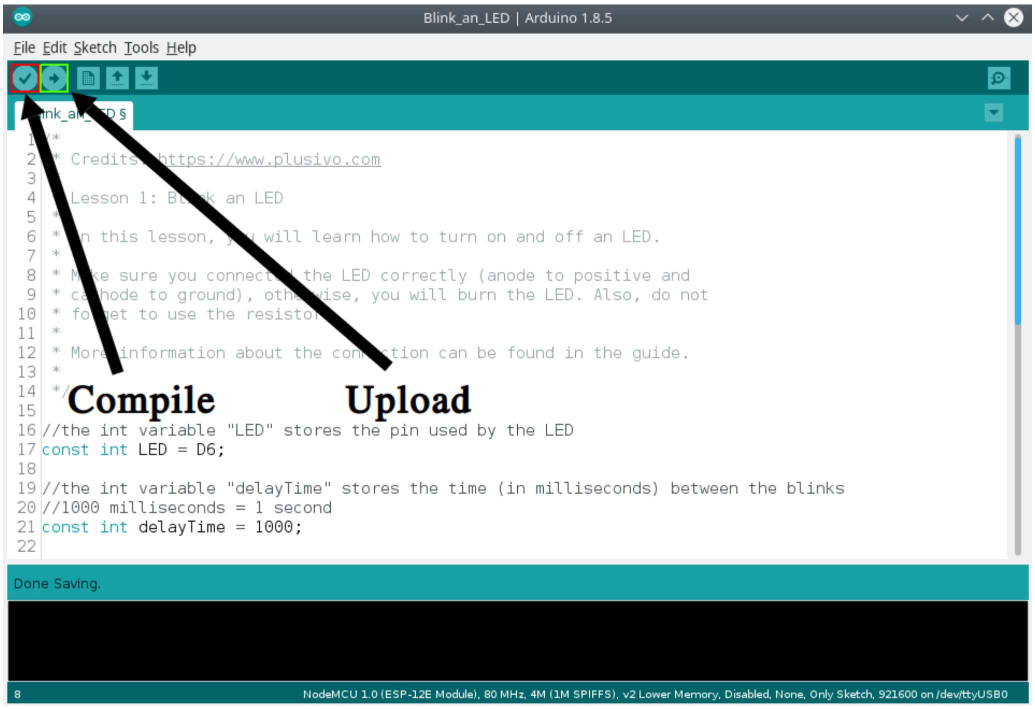


Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
ESP8266 Sketch Data Upload
WiFi101 Firmware Updater
Board: "NodeMCU 1.0 (ESP-12E Module)"
Flash Size: "4M (1M SPIFFS)"
Debug port: "Disabled"
Debug Level: "None"
lwIP Variant: "v2 Lower Memory"
CPU Frequency: "80 MHz"
Upload Speed: "921600"
Erase Flash: "Only Sketch"
Port: "/dev/ttyUSB0"
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader

NodeMCU 1.0 (ESP-12E Module), 80 MHz, 4M (1M SPIFFS), v2 Lower Memory, Disabled, None, Only Sketch, 921600 on /dev/ttyUSB0

Now click on the **Upload** button (green rectangle) or use the shortcut **CTRL + U**. After you click on the button, the code is verified, compiled and then uploaded to the board.

Sometimes, you just want to verify the code without uploading to the board. In that case, you can use the **Verify** button ✓ **(red rectangle)**. This action does not require the board to be connected to the computer.

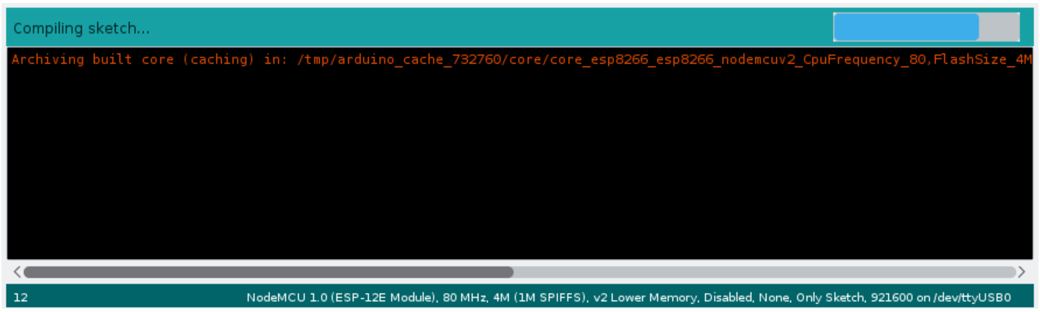


```
/*
 * Credits: https://www.plusivo.com
 *
 * Lesson 1: Blink an LED
 *
 * In this lesson, you will learn how to turn on and off an LED.
 *
 * Make sure you connected the LED correctly (anode to positive and
 * cathode to ground), otherwise, you will burn the LED. Also, do not
 * forget to use the resistor.
 *
 * More information about the connection can be found in the guide.
 *
 */

//the int variable "LED" stores the pin used by the LED
const int LED = D6;

//the int variable "delayTime" stores the time (in milliseconds) between the blinks
//1000 milliseconds = 1 second
const int delayTime = 1000;
```

```
Done Saving.
```

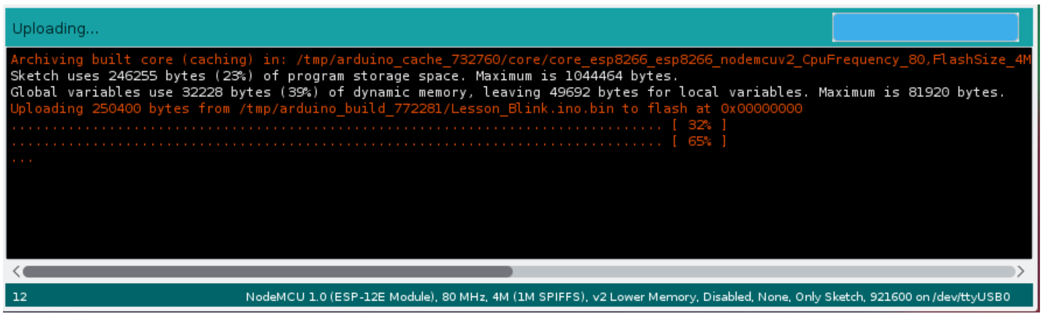
```
8        NodeMCU 1.0 (ESP-12E Module), 80 MHz, 4M (1M SPIFFS), v2 Lower Memory, Disabled, None, Only Sketch, 921600 on /dev/ttyUSB0
```

Below, you can see the steps the code goes through. It goes from "**Compiling Sketch**" to "**Uploading**", and, finally, to "**Done uploading**". In the first step, the code is verified, and if there are any syntax errors, you can see them in the bottom of the window. Then, the code is uploaded to the board. During this step, you can get errors that are more likely to be hardware mistakes rather than software ones, such as the port not being correctly selected, the board is broken or there is not enough storage space.



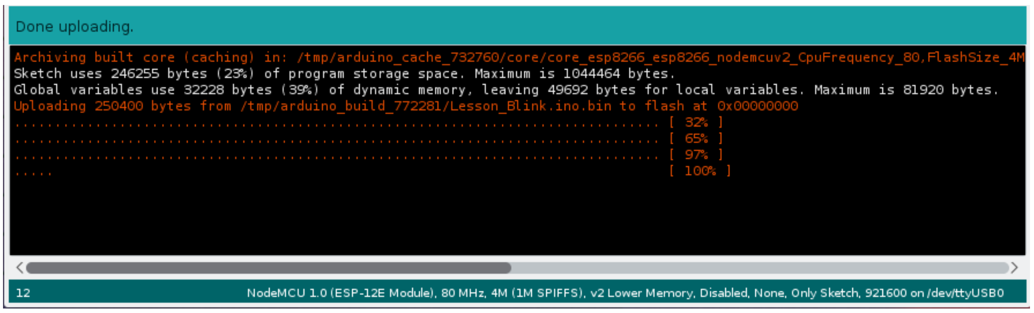
```
Compiling sketch...
Archiving built core (caching) in: /tmp/arduino_cache_732760/core/core_esp8266_esp8266_nodemcuv2_CpuFrequency_80,FlashSize_4M
```

```
12          NodeMCU 1.0 (ESP-12E Module), 80 MHz, 4M (1M SPIFFS), v2 Lower Memory, Disabled, None, Only Sketch, 921600 on /dev/ttyUSB0
```



```
Uploading...
Archiving built core (caching) in: /tmp/arduino_cache_732760/core/core_esp8266_esp8266_nodemcuv2_CpuFrequency_80,FlashSize_4M
Sketch uses 246255 bytes (23%) of program storage space. Maximum is 1044464 bytes.
Global variables use 32228 bytes (39%) of dynamic memory, leaving 49692 bytes for local variables. Maximum is 81920 bytes.
Uploading 250400 bytes from /tmp/arduino_build_772281/Lesson_Blink.ino.bin to flash at 0x00000000
................................................................................ [ 32% ]
................................................................................ [ 65% ]
...
```

```
12          NodeMCU 1.0 (ESP-12E Module), 80 MHz, 4M (1M SPIFFS), v2 Lower Memory, Disabled, None, Only Sketch, 921600 on /dev/ttyUSB0
```



```
Done uploading.
Archiving built core (caching) in: /tmp/arduino_cache_732760/core/core_esp8266_esp8266_nodemcuv2_CpuFrequency_80,FlashSize_4M
Sketch uses 246255 bytes (23%) of program storage space. Maximum is 1044464 bytes.
Global variables use 32228 bytes (39%) of dynamic memory, leaving 49692 bytes for local variables. Maximum is 81920 bytes.
Uploading 250400 bytes from /tmp/arduino_build_772281/Lesson_Blink.ino.bin to flash at 0x00000000
................................................................................ [ 32% ]
................................................................................ [ 65% ]
................................................................................ [ 97% ]
.....                                                                            [ 100% ]
```

```
12          NodeMCU 1.0 (ESP-12E Module), 80 MHz, 4M (1M SPIFFS), v2 Lower Memory, Disabled, None, Only Sketch, 921600 on /dev/ttyUSB0
```

After you get the confirmation message, you can check if the LED is blinking. During the uploading process, you can see the other led blinking very fast.

Congratulations! You have just completed your very first tutorial. You have just learned how to upload code to a development board and how the code is structured.

5. Lesson 2: Dim an LED

5.1 Overview

In this lesson you will learn how to control the brightness of an LED.

5.2 Components required

- Development board;
- 1 x LED;
- 1 x 150 Ω resistor ;
- Breadboard 830p;
- 2 x male-to-male jumper wire;
- Micro USB – Type A USB Cable;

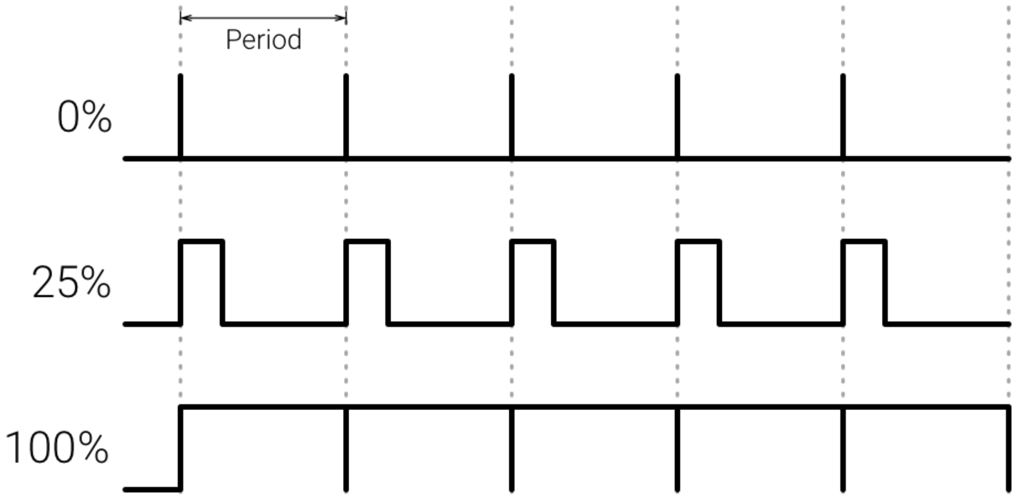
5.3 Theory

Pulse Width Modulation (PWM)

PWM is a modulation technique used to control the speed of motors or the brightness of LEDs. **PWM** creates a square wave by switching between on and off.

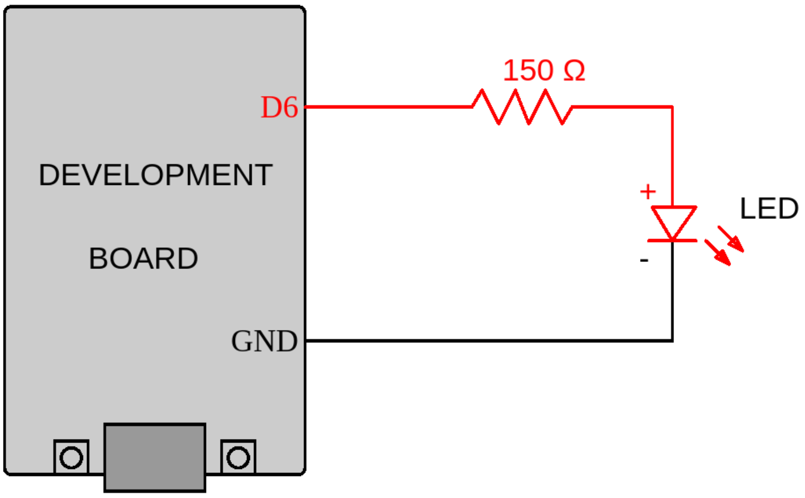
The value that characterizes the PWM is the **Duty Cycle**. It represents the duration in which the voltage is kept on. For example, 25% duty cycle means that 25% of the period the voltage is high and 75% of the period the voltage is low. The PWM frequency is about 1000 Hz, so the period is about 1 ms.

The duty cycle can be set using the instruction "**analogWrite(value);**". The value is stored on 10 bits, so **0** coresponds to 0% duty cycle and **1023** coresponds to 100% duty cycle.

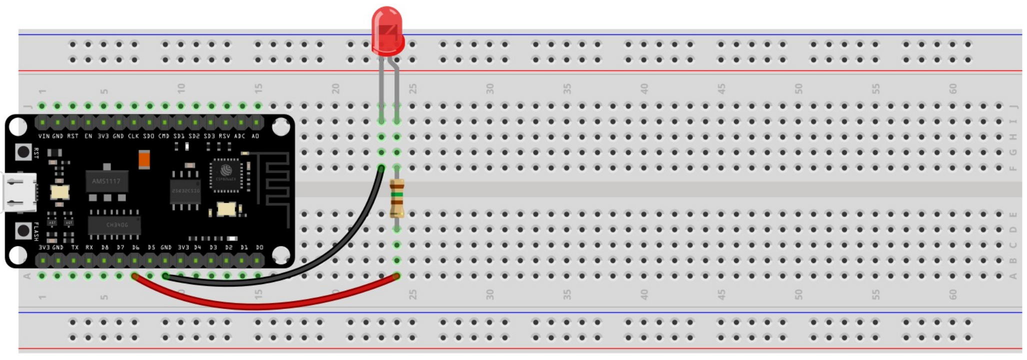


5.4 Connection

Below, you can find the schematic:



Below, you can find a visual representation of the connections:



5.5 Code

The code is pretty similar to the one included in the previous lesson. In addition to the last code, now you will learn how to use the PWM signal. The code for this lesson can be found in the folder **"Lesson 2: Dim an LED"**.

We have to declare the pin used, and in the **setup()** function we need to set the pin as **OUTPUT** using **pinMode()**.

Code 5.5.1 Declaration and the setup() function

```
//the int variable "LED" stores the pin used by the LED
const int LED = D6;

//the setup function runs only once when the board is powered
void setup()
{
  //the following instruction initialises the pin stored in the
  //variable LED as OUTPUT
  pinMode(LED, OUTPUT);
}
```

In the **loop()** function we will use the function **analogWrite(pin, value);** to set the brightness on the LED. The **value** can be from **0** (0% duty cycle, which means that the LED is off) to **1023** (100% duty cycle and the LED will be at maximum brightness). Using a **for** loop we will set the value from **0** to **1023** with a delay of **2** milliseconds to have time to see the fading. Also we are using another **for** loop to set the value from **1023** to **0** with a delay of **2** milliseconds. The first **for** loop is used to set the brightness of the LED from **0%** to **100%**, and the second is used to set the brightness from **100%** to **0%**.

Code 5.5.2 The loop() function

```
void loop()
{
  //PWM is generated using 10 bits, so it ranges between
  //0 and 1023 (2^10 = 1024)

  for(int fade = 0; fade < 1024; fade++)
  {
    //set the brightness of the LED using analogWrite();
    analogWrite(LED, fade);

    //wait 2 milliseconds
    delay(2);
  }

  //keep the LED at the maximum brightness for 500 ms
  delay(500);

  for(int fade = 1023; fade >= 0; fade--)
  {
    //set the brightness of the LED using analogWrite();
    analogWrite(LED, fade);

    //wait 2 milliseconds
    delay(2);
  }

  //keep the LED off for 500 ms
  delay(500);
}
```

6. Lesson 3: RGB LED

6.1 Overview

RGB LEDs are pretty similar to the LEDs used in the last lesson, but they light multiple colours, rather than one. Basically, an RGB LED consists of 3 LEDs. RGB stands for **R**ed, **G**reen and **B**lue. Using these 3 colours, you can create almost any colour. They mostly come in 2 versions: common anode or common cathode. The common pin is usually the longest one.

NOTE! Try not to mistake the common anode for common cathode LEDs as it is very difficult to identify them.

They have either a common anode or a common cathode in order to reduce the pins used, from 6 pins to just 4 pins. The common anode means that one positive line (usually 3.3 V or 5 V) is used by all LEDs. On the other hand, the common cathode means that the one ground line is shared by all LEDs.

6.2 Components required

- Development board;
- 4 x male-male jumper wires;
- 3 x 150 Ω resistors;
- 1 x RGB LED;

6.3 Components introduction

RGB LED

At first sight, an RGB LED looks pretty similar to a regular LED. However, inside the body, there are 3 LEDs: red, green and blue. By adjusting the brightness of these 3 LEDs, you can get almost any colour.

Adjusting the brightness can be done in two ways:

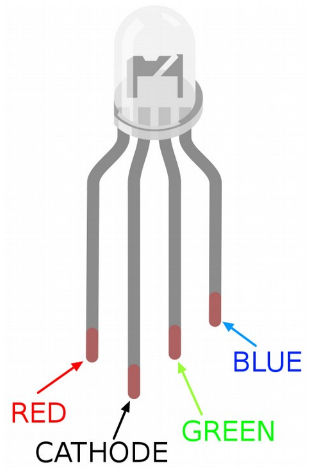
- Using resistors of different resistance.

By limiting the current flowing through each LED, you can get different colours. This method is not recommended as you need to have multiple resistors and to intensively test in order to get the desired colour.

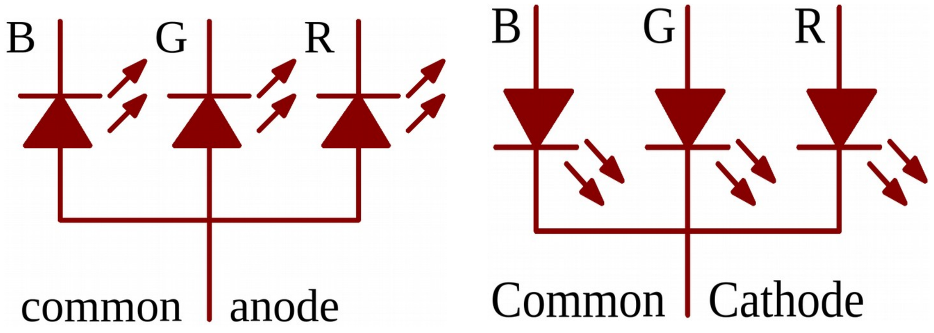
- Using the **Pulse Width Modulation (PWM)** technique.

The main advantage of this method is that you don't need to change the resistors in order to modify the brightness of the LED. The development board included in the kit has 8 digital pins capable of **PWM** (D1 – D8).

Here is a representation that helps you identify the terminal corresponding the each colour:



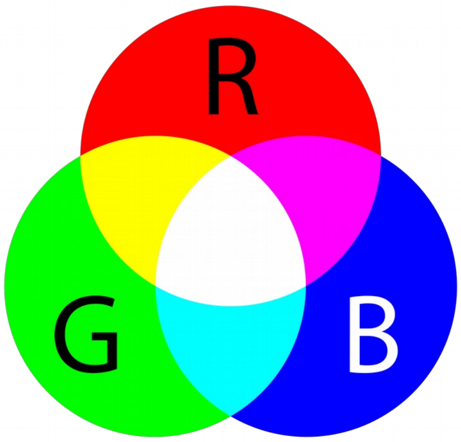
In schematics, it is represented as follows:



6.4 Theory

Colour

The reason that you can mix these 3 colours (red, green, blue) in order to get any colour is that our eyes have three types of light receptors (red, green, blue). By processing the brightness of these 3 colours, our brain associate it with a colour of the visible spectrum. The same technique is used in LCD TVs or smartphones.

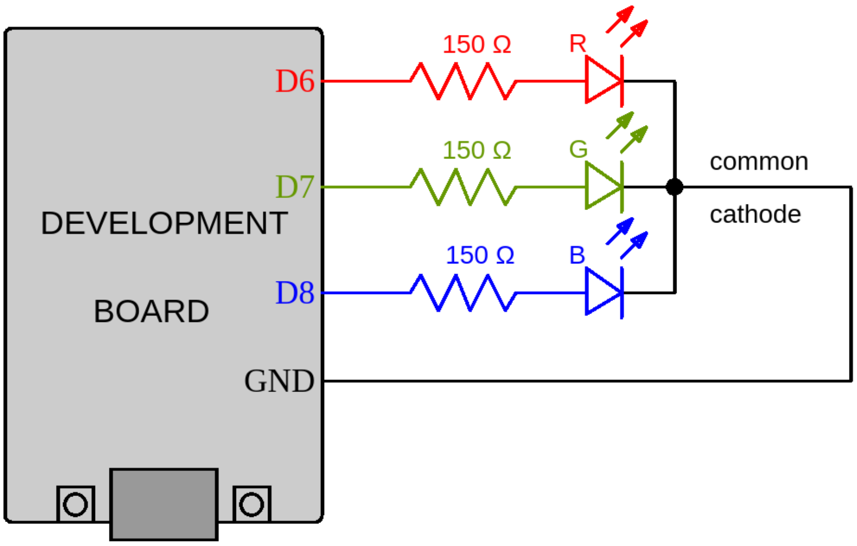


For example, in order to get yellow, you have to turn on the RED and GREEN LEDs, and turn off the BLUE LED.

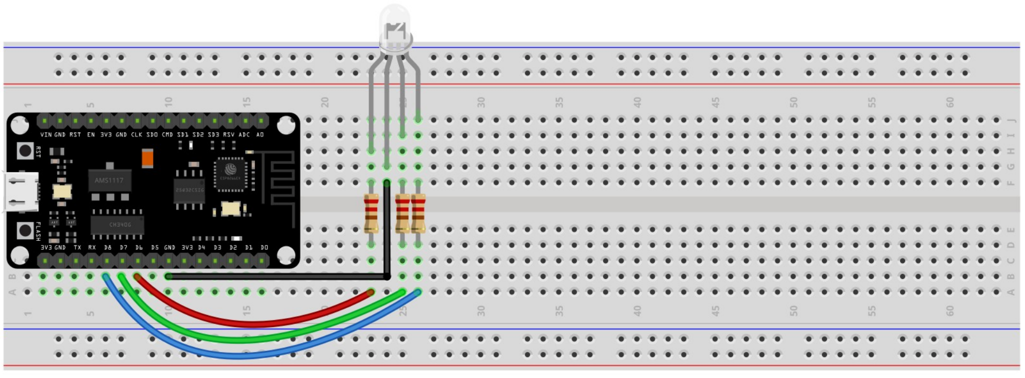
The problem with this method is that you can't get black. So, the closest we can come to black with a RGB LED is to turn off all the three LEDs.

6.5 Connection

Here is the schematic:



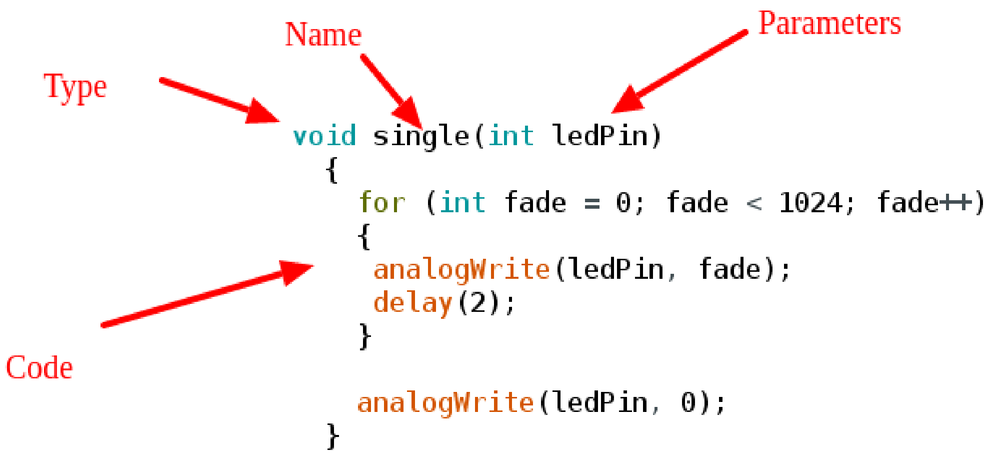
Below, you can find a visual representation of the connections:



6.6 Code

The code is pretty similar to the one included in the previous two lessons, but this time we will use an RGB LED, resulting in working with three LEDs. Also in this code we have created our own functions, so you will learn how to do that. The code for this lesson can be found in the **"Lesson 3: RGB LED"** folder.

In order to create a function, you have to know some basic things. Below, you can find a picture that shows the core of a function. It must have a type, a name, and code. The parameters are optional.



```
void single(int ledPin)
 {
   for (int fade = 0; fade < 1024; fade++)
   {
    analogWrite(ledPin, fade);
    delay(2);
   }

   analogWrite(ledPin, 0);
 }
```

The type of the function depends on the value expected to be returned. In the previously attached picture, there is no value returned so the type of the function is **void.**

In the next example, the type of the function is **int** as it is returning an int variable.

```
int sum(int a, int b)
{
   int c;
   c = a + b;
   return c;
}
```

The code begins with the declaration of two variables that stores the pins used by the RGB LED. In the **setup()** function we are setting the pins as **OUTPUT**.

Code 6.6.1 Declaration and setup() function

```
//create 3 variables that are used to store the pins to which is the LED
//attached
const int red = D6;
const int green = D7;
const int blue = D8;

void setup()
{
  //declase the pins as OUTPUT
  pinMode(red, OUTPUT);
  pinMode(green, OUTPUT);
  pinMode(blue, OUTPUT);
}
```

We will create 3 functions to fade one LED, two LEDs and three LEDs. The first function will fade just one LED and it contains a **for** loop, the same as the first **for** loop from the code from the previous lesson.

Code 6.6.2 Dimming one LED

```
void single (int ledPin)
{
  //PWM is generated using 10 bits, so it ranges between
  //0 and 1023 (2^10 = 1024)

  for (int fade = 0; fade < 1024; fade++)
  {
    //set the brightness of the LED using analogWrite();
    analogWrite(ledPin, fade);

    delay(2);//wait for 2 milliseconds
  }

  //turn off the led
  analogWrite(ledPin, 0);
}
```

Another function created is used to fade 2 LEDs at a time. The function is similar with the **single** function, but we need to add in the **for** loop another

analogWrite for the second LED, and also turn off the LED at the end of the function.

Code 6.6.3 Dimming two LEDs

```
void duo (int firstLed, int secondLed)
{
  for (int fade = 0; fade < 1024; fade++)
  {
    //set the brightness of the leds using analogWrite();
    analogWrite(firstLed, fade);
    analogWrite(secondLed, fade);
    delay(2);//wait for 2 milliseconds
  }

  //turn off both leds
  analogWrite(firstLed, 0);
  analogWrite(secondLed, 0);
}
```

The third new function is the one used to dim all the LEDs. This function is also similar with the previous ones, all we need to to is to add another **analogWrite** call for the third LED to control its brightness.

Code 6.6.4 Dimming all three LEDs

```
void all (int firstLed, int secondLed, int thirdLed)
{
  for (int fade = 0; fade < 1024; fade++)
  {
    //set the brightness of the LEDs using analogWrite();
    analogWrite(firstLed, fade);
    analogWrite(secondLed, fade);
    analogWrite(thirdLed, fade);
    delay(2);//wait for 2 milliseconds
  }

  //turn off all LEDs
  analogWrite(firstLed, 0);
  analogWrite(secondLed, 0);
  analogWrite(thirdLed, 0);
}
```

The only function left is **loop()**, where we will call all the functions we have just created. Firstly, the **single** function will be called three times, because this function will control each individual LED, the **duo** function will be called also three times to combine two colours at a time and the **all** function will be called only one time because it controls all the three LEDs of the RGB LED.

Code 6.6.5 The loop() function

```
void loop()
{
  //in the loop function we are going to call
  //the previously created functions

  //we are going to turn on the LEDs one by one
  single(red);
  single(green);
  single(blue);

  //turn on LEDs two by two
  duo(red, green);
  duo(red, blue);
  duo(green, blue);

  //turn on all 3 LEDs
  all(red, green, blue);
}
```

7. Lesson 4: Motor Control

7.1 Overview

In this lesson you will learn how to control a motor.

7.2 Components required

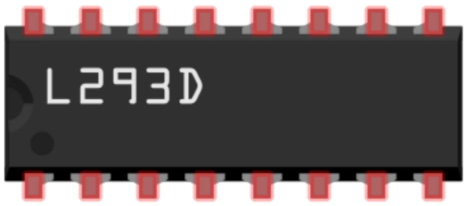
- Development board;
- Micro USB – Type A USB cable;
- L293D H-Bridge Motor Driver;
- Breadboard 830p;
- Breadboard power supply;
- 9 x male-to-male jumper wires;
- 1 x DC motor;

7.3 Components introduction

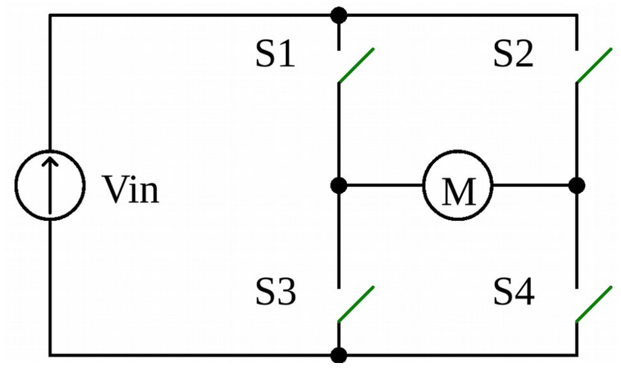
L293D H-Bridge

Because the board isn't powerful enough alone to power a motor we are going to use a **DC** motor driver in between the development board and the motors.

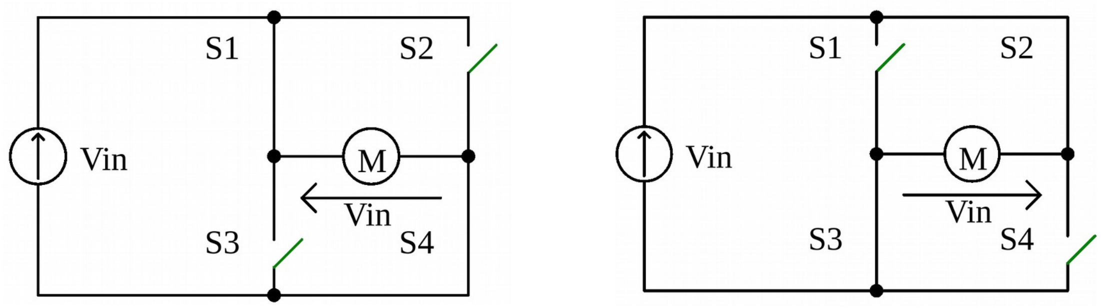
For this tutorial we use the **L293D H-Bridge Motor Driver**.



An **H-bridge** is an electronic circuit that enables a voltage to be applied across a load in opposite direction. These bridges are often used in robotics as they allow DC motors to run forwards or backwards.



This is an **open** H-Bridge schematic. The motor is not connected to any of the poles of the power supply, as all the four switches are open (S1, S2, S3, S4).



These are the two basic states of a H-Bridge. As you can see the voltage runs in opposite directions depending to what switches are closed.

We are going to represent shortly in a small table what each switch state does to the motor. **1** means that the switch is closed and **0** that the switch is open.

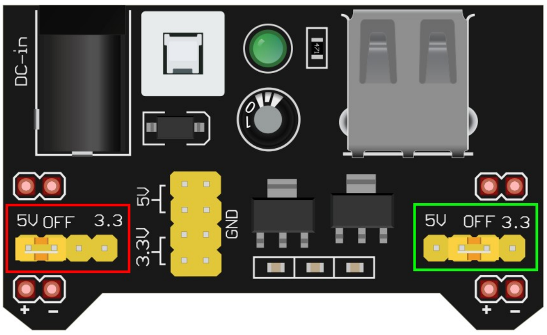
S1	S2	S3	S4	Result
1	0	0	1	Motor moves right
0	1	1	0	Motor moves left
0	0	0	0	Motor coasts
1	0	0	0	Motor coasts
0	1	0	0	Motor coasts
0	0	1	0	Motor coasts
0	0	0	1	Motor coasts
0	0	1	1	Motor brakes
1	1	0	0	Motor brakes
1	0	1	0	Short circuit
0	1	0	1	Short circuit
0	1	1	1	Short circuit
1	0	1	1	Short circuit
1	1	0	1	Short circuit
1	1	1	0	Short circuit
1	1	1	1	Short circuit

The **L293D Motor Driver** has two voltage entrances to power the driver, and subsequently the motors. The **Vss** entrance is used to power the driver, and the **Vs** entrance to power the motors.

The pins we will control are **EN1, EN2, IN1, IN2, IN3, IN4**. **EN1** and **EN2** are enable pins (we use them basically to start the motor). The **IN1**, **IN2** and respectively the **IN3**, **IN4** control the direction of the rotation.

Breadboard power supply

This power supply is pretty useful in prototyping because it is breadboard friendly and has variable outputs, 3.3 V or 5 V.



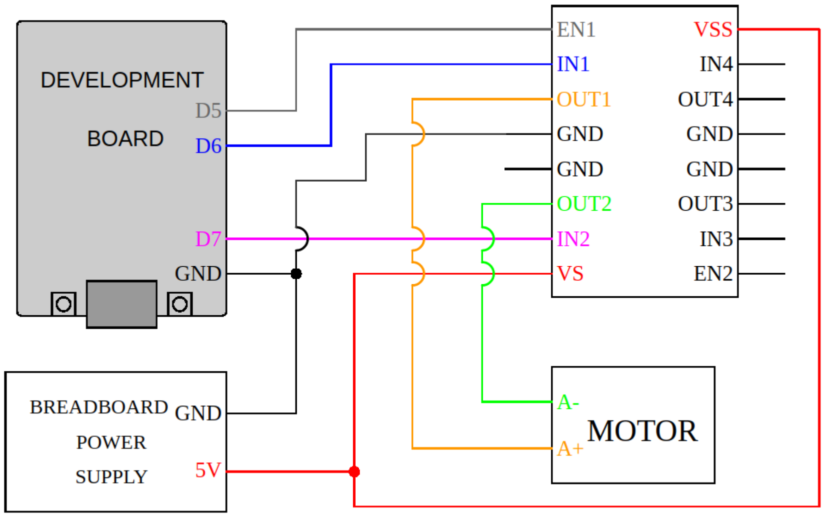
The only way to power this power supply is through DC jack. Supply voltage must be between 6.5 V and 12 V. **DO NOT** try to power the module via USB port, it is only for output.

In order to select the output voltage, you have to move a jumper between different pins. For example, in the above image, the jumper in the **red rectangle** is set to 5 V and the jumper in the **green rectangle** is set to OFF (this means that there is no output voltage).

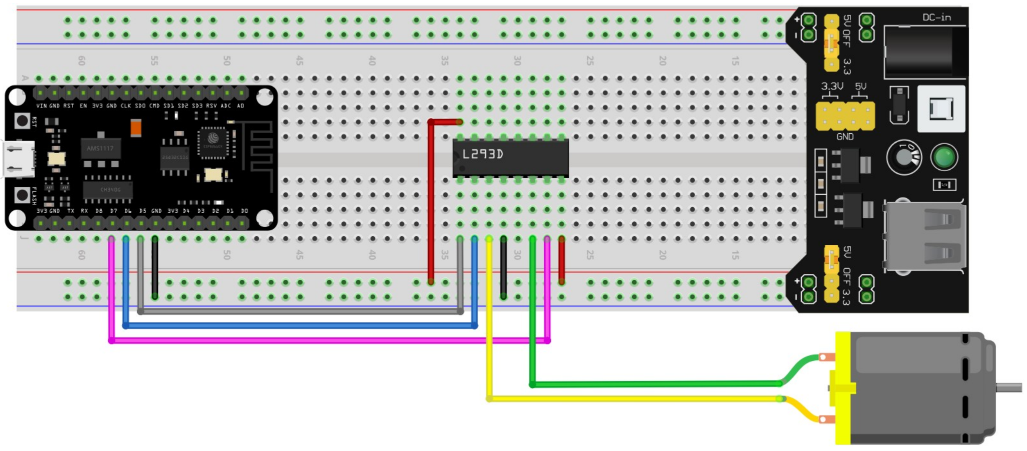
Moreover, there is a button, located next the input jack, that turns the module on/off.

7.4 Connections

Below, you can find the schematic:



Below, you can find a visual representation of the connections:



7.5 Code

The code required to control a motor can be found in the folder called "**Lesson 4: Motor Control**", and it is similar to the one included in the previous lesson. In addition to that code, you will learn how to use the PWM signal to set the speed of a motor from 0 (off) to full speed.

The code starts with the declaration of three variables that stores the three pins used to control the motor: the **speed** pin and the **direction** pins. In the **setup()** function we will set the pins as **OUTPUT**.

Code 7.5.1 The setup() function

```
void setup()
{
  //the following instruction initialise the pin stored in the
  //variable motorspeed_pin(also DIRA and DIRB) as OUTPUT
  pinMode(motorspeed_pin, OUTPUT);
  pinMode(DIRA, OUTPUT);
  pinMode(DIRB, OUTPUT);
}
```

We will create a new function that will turn off the motor by setting the speed pin and direction pins to **LOW**, using the **digitalWrite**.

Code 7.5.2 Function for turning off the motor

```
void turnOff()
{
  //this instruction is used to set the speed of the motor to 0 (off)
  digitalWrite(motorspeed_pin, LOW);
  //in these instructions the state is irrelevant because the motor is off
  digitalWrite(DIRA, LOW);
  digitalWrite(DIRB, LOW);
  //wait 1.5 seconds
  delay(delayOff);
}
```

In the **loop()** function we will first turn on the motor at maximum speed, by setting the **motorspeed_pin** to **HIGH**, and the **DIRA** pin to **HIGH** and **DIRB** pin to **LOW** for turning the motor in one direction, and the **DIRA** pin to **LOW** and **DIRB** pin to **HIGH** for turning the motor in the opposite direction. The first time we will turn on the motor at maximum speed in one direction, then turn it off for some time, then turn it on at maximum speed in the opposite direction, then turn it off.

Next, we will turn on the motor in one direction at 50% speed using the **analogWrite(motorspeed_pin, value)** function with a value of **512**, which is approximately 50% speed, and **DIRA** pin to **HIGH** and **DIRB** pin to **LOW**. We will turn off the motor and then repeat the previous instructions, but this time **DIRA** pin will be set to **LOW** and **DIRB** pin to **HIGH** so that the speed of the motor will be 50% and the motor will rotate in the opposite direction.

Code 7.5.3 The loop() function

```
void loop()
{
  //this instruction is used to set the maximum speed of the motor
  digitalWrite(motorspeed_pin, HIGH);
  //these instructions are used to turn on the motor in one direction
  digitalWrite(DIRA, HIGH);
  digitalWrite(DIRB, LOW);
  delay(delayOn);

  //turn off the motor
  turnOff();

  //this instruction is used to set the maximum speed of the motor
  digitalWrite(motorspeed_pin, HIGH);
  //these instructions are used to turn on the motor in the opposite direction
  digitalWrite(DIRB, HIGH);
  digitalWrite(DIRA, LOW);
  //wait 3 seconds
  delay(delayOn);

  //turn off the motor
  turnOff();

  //this instruction sets the motor speed to about 50%
  //you can put any integer from 0 to 1023
  analogWrite(motorspeed_pin, 512);
  //these instructions are used to turn on the motor in one direction
  digitalWrite(DIRA, HIGH);
  digitalWrite(DIRB, LOW);
  //wait 3 seconds
  delay(delayOn);

  //turn off the motor
  turnOff();

  //this instruction sets the motor speed to about 50%
  analogWrite(motorspeed_pin, 512);
  //these instructions are used to turn on the motor in the opposite direction
  digitalWrite(DIRB, LOW);
  digitalWrite(DIRA, HIGH);
  //wait 3 seconds
  delay(delayOn);

  //turn off the motor
  turnOff();
}
```

8. Lesson 5: Ultrasonic HC-SR04+

8.1 Overview

In this lesson you will learn how to use the ultrasonic module in order to calculate the distance from 2 centimeters and up to 4 meters.

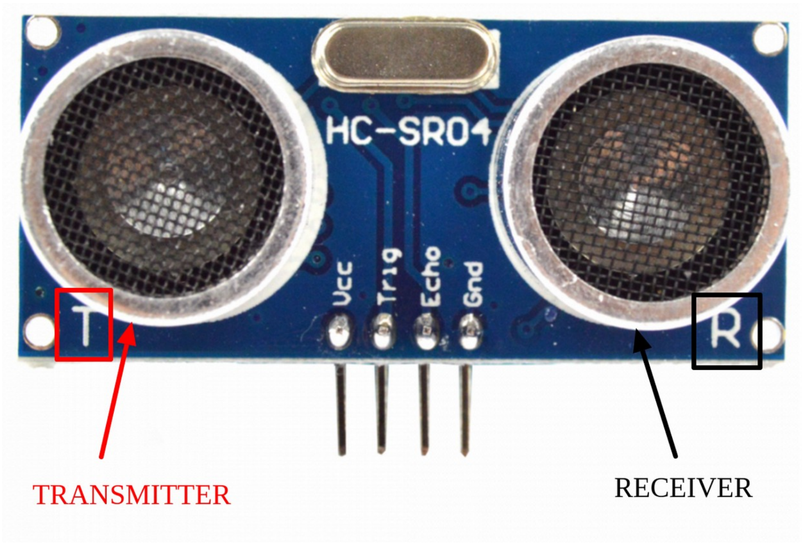
8.2 Components required

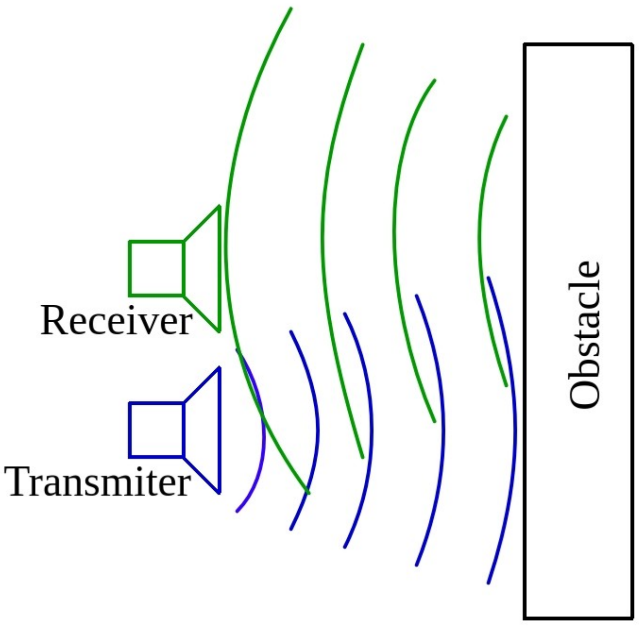
- Development board;
- Ultrasonic module HC-SR04+;
- Breadboard 830p;
- 4 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

8.3 Components introduction

Ultrasonic HC SR04+

The ultrasonic module consists of an ultrasonic transmitter and an ultrasonic receiver.





The transmitter is used to generate an ultrasonic sound at about 40 kHz. When there is an object in front of the transmitter, the ultrasound is bounced back to the receiver. Knowing the speed of sound, we can calculate the distance to that object with an accuracy of up to 3mm.

Below, you can learn how to calculate the distance using the speed of sound.

1) Transmitter emits an ultrasonic sound for about 10 microseconds. 1 microsecond (1μs) is equally to 10^{-6} seconds (0.000001 seconds).
2) If there is an object, the ultrasonic sound is bounced back to the sensor, where the receiver is listening.
3) Now, we have to record the time it takes the sound to travel from the transmitter to object and back to the receiver.
4) Using the speed of sound (0.034 cm/μs), you can calculate the distance using the following formula:

$$d=\frac{t\cdot v}{2}$$

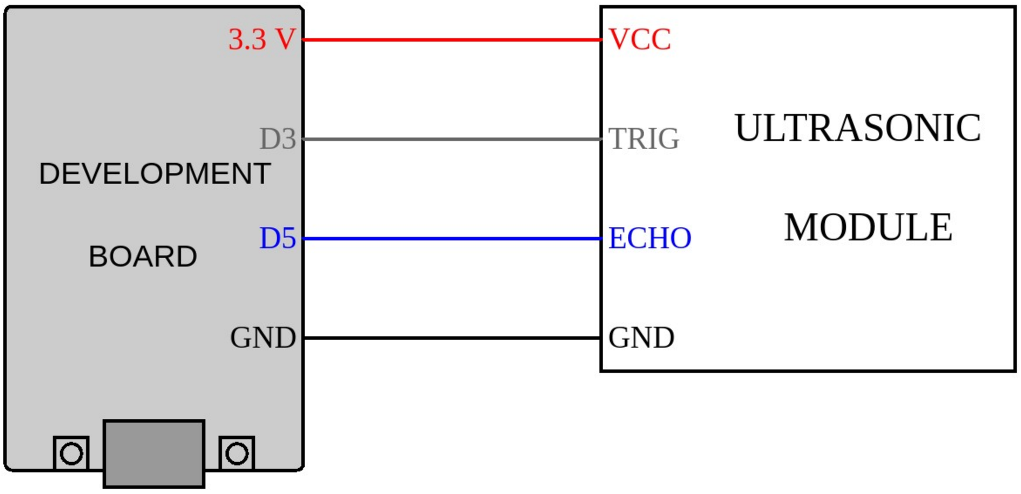
where:

- **d** represents the distance in centimeters;
- **t** represents the time passed for the sound to return to the receiver;
- **v** represents the speed of sound (0.034 cm/μs);

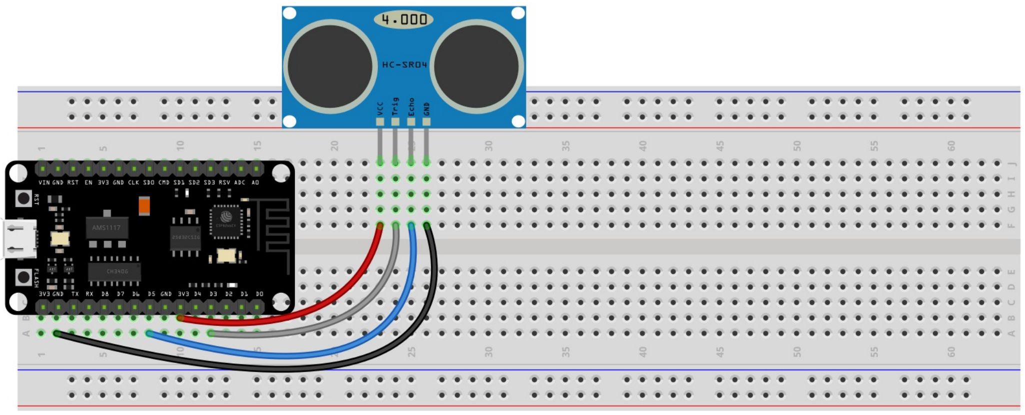
This ultrasonic module can work with voltages between 3V and 5.5V and has a small operating current (3 mA).

8.4 Connections

Below you can see the schematic:



Next you will find the visual representation:



8.5 Code

The code required for this lesson can be found in the folder called "**Lesson 5: Ultrasonic HC-SR04+**".

The code begins with the declaration of the pins used by the ultrasonic module, **echo** and **trigger**. Also, we will need two variables, **duration** and **distance**, to calculate the distance. These two variables will help us later. In the **setup()**

function we will start a serial communication with the computer at a baud rate of 115200, using the instruction **Serial.begin(115200)**, so that we can display the distance in **Serial Monitor**. Next we will set the trigger pin as **OUTPUT** and the echo pin as **INPUT**.

Code 8.5.1 Variables declaration and setup() function

```
//declare the pins used by the module
const int echoPin = D5;
const int trigPin = D3;

//declare 2 variables which help us later to calculate the distance
long duration;
double distance;

void setup()
{
  //start the serial communication with the computer at 115200 bits/s
  Serial.begin(115200);

  Serial.println("The board has started");

  //the trigger pin (transmitter) must be set as OUTPUT
  pinMode(trigPin, OUTPUT);

  //the echo pin (receiver) must be set as INPUT
  pinMode(echoPin, INPUT);
}
```

The **loop()** function starts by setting the trigger pin to **LOW,** in order to prepare for a reading, and set a delay of 2 microseconds.

In order to calculate the distance, we need the time that the sound travels from the transmitter to object and back to the receiver. For that, we have to generate an ultrasound by turning the transmitter **ON** for 10 microseconds (1 microsecond = 10^{-6} seconds) then record the sound wave travel time using the following instruction:

```
duration = pulseIn(pin, VALUE, timeout);
```

where:

- **pin** (type int) : represents the number of the pin on which we want to read the pulse
- **value**: type of pulse (either HIGH or LOW);
- **timeout** (unsigned long): the number of microseconds to wait for the pulse to start.

Now that you have the duration of the pulse, you can calculate the distance using the following formula:

$$d=\frac{t \cdot v}{2}$$

where:

- **d** – represents the distance in centimeters;
- **t** – represents the duration of the pulse;
- **v** – represents the speed of sound in air (about 0.034 cm/µs).

Do not forget to divide the result by 2 as the sound has to travel the same distance twice (forwards and bounce backwards).

After calculating the distance, we will display it in **Serial Monitor**. Below is the **loop()** function:

Code 8.5.2 Calculate the distance and display it in Serial Monitor in real time

```
void loop()
{
  //set the trigPin to LOW in order to prepare for the next reading
  digitalWrite(trigPin, LOW);

  //delay for 2 microseconds
  delayMicroseconds(2);

  //generate an ultrasound for 10 microseconds then turn off the transmitter
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  //reads the echoPin, returns the sound wave travel time in microseconds
  duration = pulseIn(echoPin, HIGH);

  //using the formula shown in the guide, calculate the distance
  distance = duration*0.034/2;

  //print the distance in Serial Monitor
  Serial.print("Distance: ");
  Serial.print(distance);
  Serial.println(" cm");
}
```

After uploading the code, in the **Serial Monitor** you can observe the distance, in real time, between the module and the nearest obstacle/object. For opening the **Serial Monitor**, in the Arduino IDE, you can use the shortcut **Ctrl+Shift+M** or go to **Tools -> Serial Monitor.**

9. Lesson 6: RGB LED and Ultrasonic

9.1 Overview

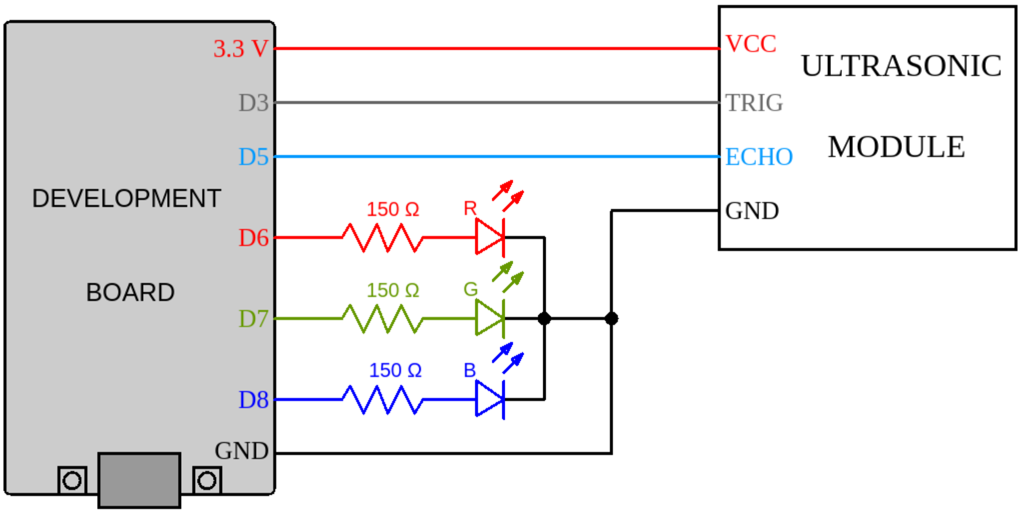
This lesson combines two lessons, lesson 3, RGB LED, and lesson 5, Ultrasonic HC-SR04+, and you will learn how to flash an LED depending on distance.

9.2 Components required

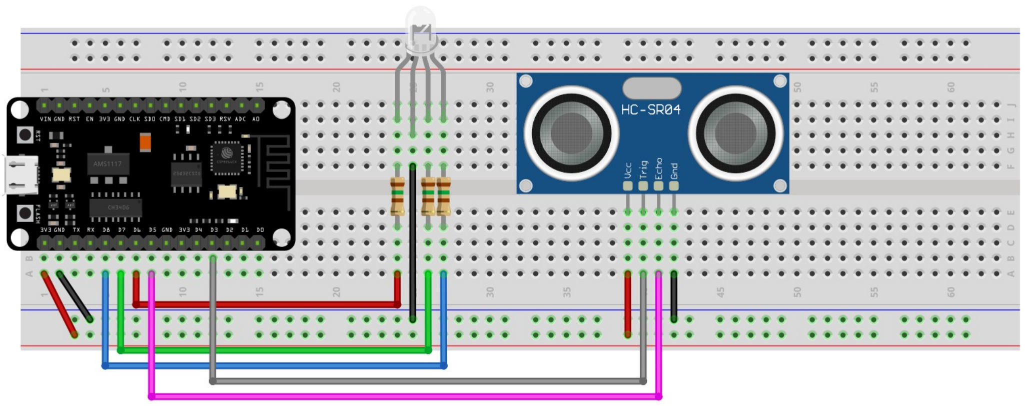
- Development board;
- Breadboard 830p;
- 1 x RGB LED;
- 3 x 150Ω resistor;
- 10 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

9.3 Connections

Below, you can find the schematic:



Next, you can find a visual representation of the project:



9.4 Code

The code for this lesson combines the code from two lessons, RGB LED and Ultrasonic HC-SR04+. You can find the code in the folder **"Lesson 6: RGB LED and Ultrasonic"**.

The code starts with the declaration of the variables used and sums up the variables from the codes of the two lessons. In the **setup()** function, firstly, we have to start the serial communication with the computer, and then set the pins used by the RGB LED as **OUTPUT**, set the trigger pin as **OUTPUT** and the echo pin to **INPUT**.

Code 9.4.1 The setup() function

```
void setup()
{
  //start the serial communication with the computer at 115200 bits/s
  Serial.begin(115200);

  //set the mode of the pins used by the RGB LED as OUTPUT
  pinMode(red, OUTPUT);
  pinMode(green, OUTPUT);
  pinMode(blue, OUTPUT);

  //the trigger pin (transmitter) must be set as OUTPUT
  pinMode(trigPin, OUTPUT);

  //the echo pin (receiver) must be set as INPUT
  pinMode(echoPin, INPUT);
}
```

In the **loop()** function we will calculate the distance, as in the previous lesson, and then, using an **if** statement, we will decide if the distance is lower than 25 centimeters, the red LED will flash at a interval of 35 ms. If the distance is greater than 25 cm, the green LED will slowly flash.

Code 9.4.2 Calculate the distance and flash the LEDs depending on distance

```
void loop()
{
  //set the trigPin to LOW in order to prepare for the next reading
  digitalWrite(trigPin, LOW);

  //delay for 2 microseconds
  delayMicroseconds(2);

  //generate an ultrasound for 10 microseconds then turn off the transmitter.
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  //reads the echoPin, returns the sound wave travel time in microseconds
  duration = pulseIn(echoPin, HIGH);

  //using the formula shown in the guide, calculate the distance
  distance = duration*0.034/2;

  //display the distance in Serial Monitor
  Serial.println(distance);

  if(distance < 25)
  {
    //the next 4 instructions are used
    //to create the flashing effect
    //turn on the LED and wait 35 ms
    digitalWrite(red, HIGH);
    delay(35);

    //turn off the LED and wait 35 ms
    digitalWrite(red, LOW);
    delay(35);
  }
  else
  {
    //turn on the green LED and wait 300 ms
    digitalWrite(green, HIGH);
    delay(300);

    //turn off the green LED and wait 200 ms
    digitalWrite(green, LOW);
    delay(200);
  }
}
```

10. Lesson 7: Digital Inputs

10.1 Overview

In this lesson, you are going to learn how to use a push button as a digital input to turn an LED on and off.

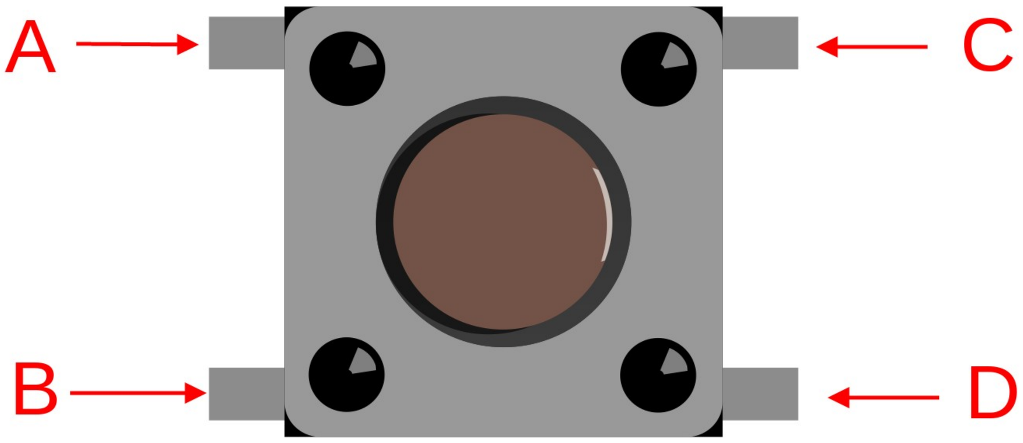
10.2 Components required

- Development board;
- Breadboard 830p;
- 2 x push buttons;
- 1 x LED;
- 1 x 150 Ω resistor;
- 7 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

10.3 Components introduction

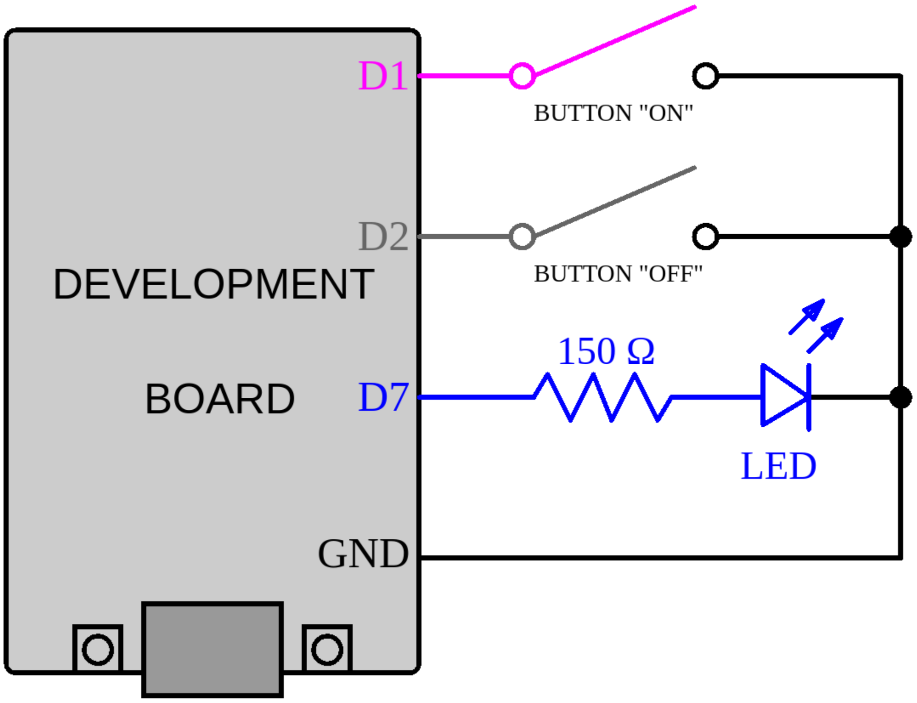
Momentary push button switch

This type of button is very common in prototyping as it is breadboard friendly, cheap and reliable. The way it works is pretty basic: there are 2 lines: A-C and B-D, which are always connected. When the button is pressed, these two lines A-C and B-D, are connected together, which means that the current will flow from A to D and from B to C.

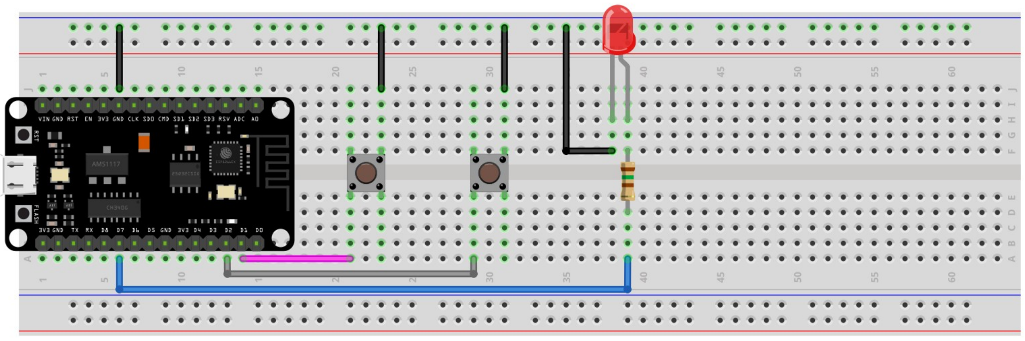


10.4 Connections

Below you can find the schematic:



Next, you can find a visual representation of the project:



10.5 Code

The code for this lesson can be found in the folder **"Lesson 7: Digital Inputs"**.

The code is very simple. First, we have to declare the pins used by the buttons and the LED. We will connect the "On" button to **D1** and the "Off" button to **D2**. The

LED will be connected to **D7**. In the **setup()** function we will set the pins for the buttons as **INPUT_PULLUP** and the pin for the LED as **OUTPUT**.

Code 10.5.1 The setup() function

```
void setup()
{
  //in order to read the state of a button, firstly, we have
  //to set the mode of the pin used by it as INPUT_PULLUP.
  pinMode(buttonON, INPUT_PULLUP);
  pinMode(buttonOFF, INPUT_PULLUP);

  //set the mode of the pin used by the led as OUTPUT
  pinMode(led, OUTPUT);
}
```

The **loop()** function will wait for the buttons to be pressed, and if the On button was pushed, then the LED will turn On, and if the Off button was pushed, then the LED will be turned Off. In order to check the state of a button (if it is pressed or not), you have to use the instruction **digitalRead(pin)**. If the value returned by this instruction is **0** (or **LOW**), it means that the button is pressed. Contrary, if the value is **1** (or **HIGH**), it means that the button is not pressed.

Code 10.5.2 The loop() function

```
void loop()
{
  //check if the button "ON" was pressed and turn on the LED
  if(digitalRead(buttonON) == 0)
  {
    //turn on the LED
    digitalWrite(led, HIGH);
  }

  //check if the button "OFF" was pressed and turn off the LED
  if(digitalRead(buttonOFF) == 0)
  {
    //turn off the LED
    digitalWrite(led, LOW);
  }

  //wait for 0.1s (100 ms)
  delay(100);
}
```

11. Lesson 8: Control an LED using push buttons

11.1 Overview

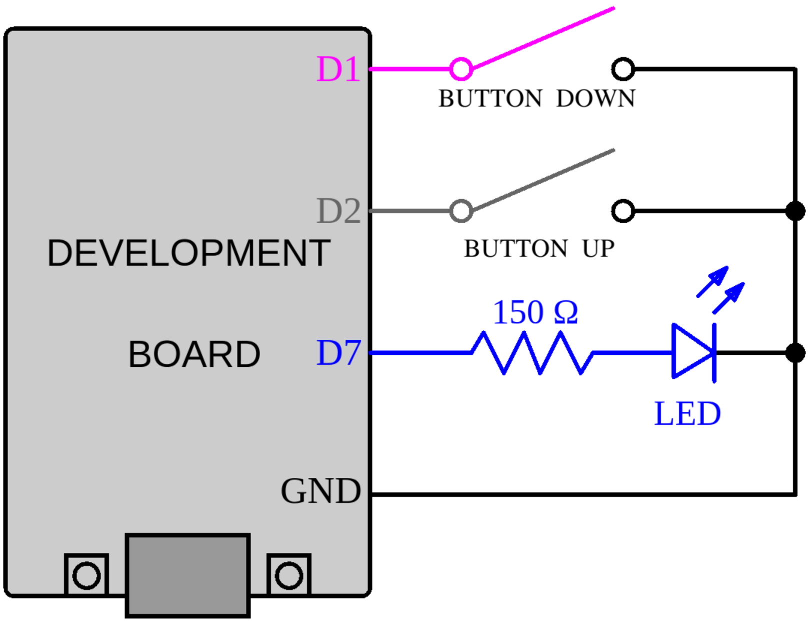
In this lesson, you are going to learn how to use two push buttons to control the brightness of an LED.

11.2 Components required

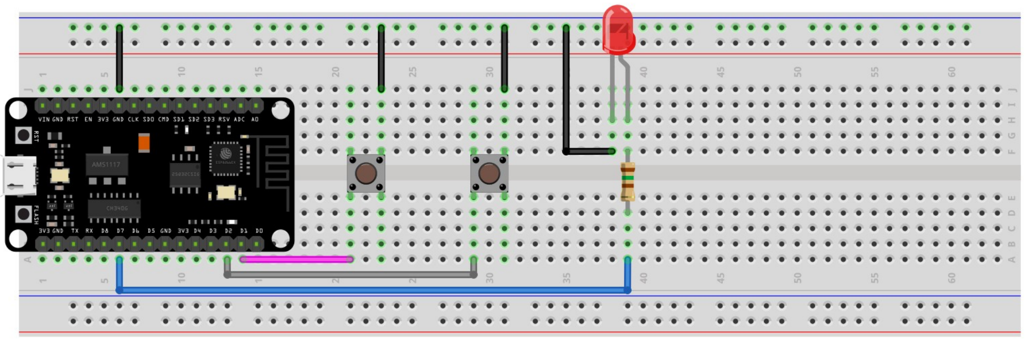
- Development board;
- Breadboard 830p;
- 2 x push buttons;
- 1 x LED;
- 1 x 150 Ω resistor;
- 7 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

11.3 Connections

Below you can find the schematic:



Next, you can find a visual representation of the project:



11.4 Code

The code for this lesson can be found in the folder **"Lesson 8: Control an LED using push buttons"** and it is easy to understand. The code starts with the declaration of the pins used by the buttons and the LED. The button that will reduce the brightness is connected to **D1** and the button for increasing the brightness is connected to **D2**. Also, we need an integer variable that will store a value from **0** (0% duty cycle) to **1023** (100% duty cycle).

Code 11.4.1 Variables declaration

```
//define the pins used by the buttons and led
const int button_down = D1;
const int button_up = D2;
const int led = D7;

//declare a variable that will store a value
//from 0 to 1023
int container = 0;
```

In the **setup()** function we will set the pins used by the buttons as **INPUT_PULLUP** and the pin used by the LED as **OUTPUT**.

Code 11.4.2 The setup() function

```
void setup()
{
  //in order to read the state of a button, firstly, you have
  //to set the mode of the pin used by it as INPUT_PULLUP.
  pinMode(button_down, INPUT_PULLUP);
  pinMode(button_up, INPUT_PULLUP);

  //set the mode of the pin used by the LED as OUTPUT
  pinMode(led, OUTPUT);
}
```

In the **loop()** function we have two **while** loops, one for each button. When the **down** button is pressed, an if statement will change the value of **container**, and turn on the LED at the new value. Also, we will need a delay, in this case we have chosen 50 ms, between runs.

Code 11.4.3 Reduce the brightness

```
while(digitalRead(button_down) == 0)
{
  //the minimum value for container is 0, so if the
  //value is greater than 50, we can substract 50
  //else, the value will be set to 0
  if(container > 50)
  {
    //change the value
    container = container - 50;
  }
  else
  {
    //set the value to 0
    container = 0;
  }

  //turn on the LED
  analogWrite(led, container);
  //wait 50 ms before the next run
  delay(50);
}
```

The second **while** loop deals with increasing the brightness, and it is similar with the **while** loop for reducing the brightness, the only difference is that the value of **container** will increase in this case.

Code 11.4.4 Increase the brightness

```
  while(digitalRead(button_up) == 0)
  {
    //the maximum value for container is 1023
    //if the value is less than 972, we can add 50
    //otherwise, the container will be set to 1023
    if(container < 972)
    {
      container = container + 50;
    }
    else
    {
      container = 1023;
    }

    //turn on the LED
    analogWrite(led, container);
    //wait 50 ms before the next run
    delay(50);
  }
```

12. Lesson 9: Buzzer

12.1 Overview

In this lesson you will learn how to connect and use a buzzer.

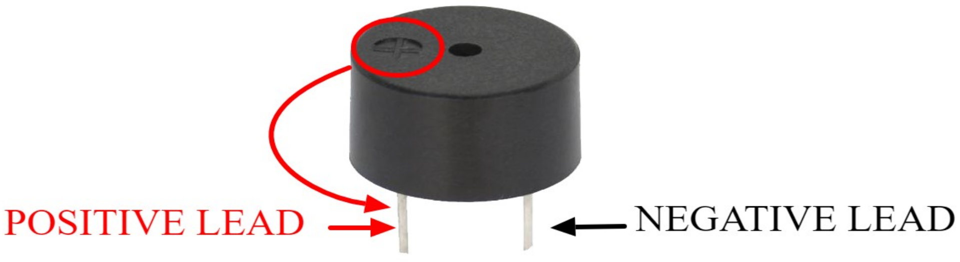
12.2 Components required

- Development board;
- Breadboard 830p;
- 1 x passive buzzer;
- Diode;
- Transistor;
- 1 x 1000 Ω resistor;
- 7 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

12.3 Components introduction

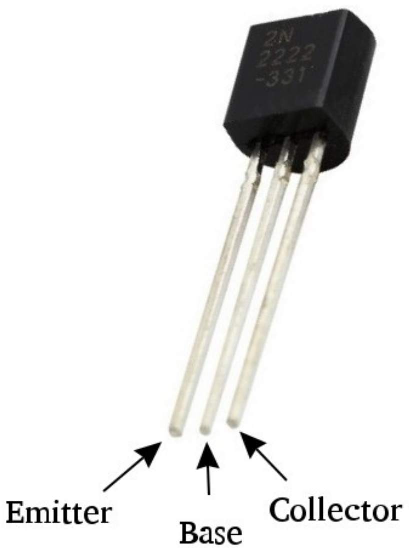
Buzzer

A buzzer is an audio signaling device that makes a buzzing sound at different frequencies. There are two types of buzzers, active and passive. In our project, we use a passive one. A passive buzzer can make different tones, but it requires a PWM signal from the device which controls the buzzer, in order to produce a noise.

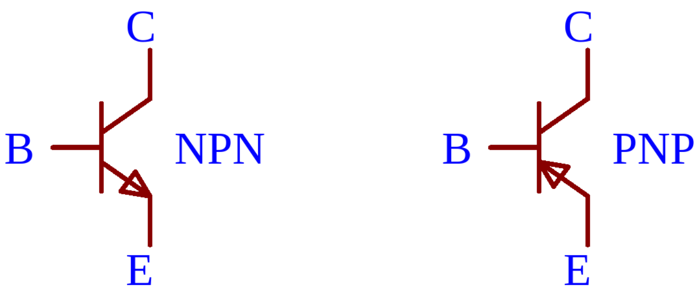


Make sure that you respect the polarity of the component. Usually, there is a **plus sign** on top of the buzzer that shows which pin is the positive one.

Transistor



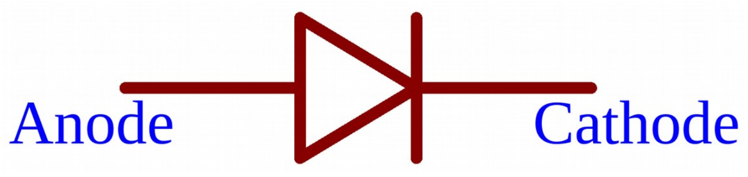
A transistor is a device that regulates current or voltage flow and acts as a switch or gate for electronic signals. Transistors are composed of semiconductor material, with three layers. The transistor's three-layer structure contains an N-type semiconductor layer sandwiched between P-type layers (a PNP configuration) or a P-type layer between N-type layers (an NPN configuration). The most common transistor configuration used is the NPN Transistor. You can find below a basic scheme for both types:

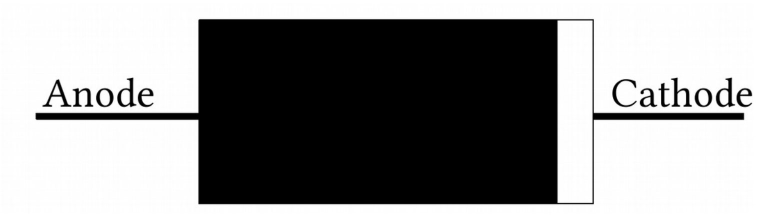


The three pins are: Base (B), Collector (C) and Emitter (E). In a standard NPN transistor, you need to apply a voltage of about 0.7V between the base and the emitter to get the current flowing from base to emitter. When you apply 0.7V from base to emitter you will turn the transistor ON and allow a current to flow from collector to emitter.

Diode

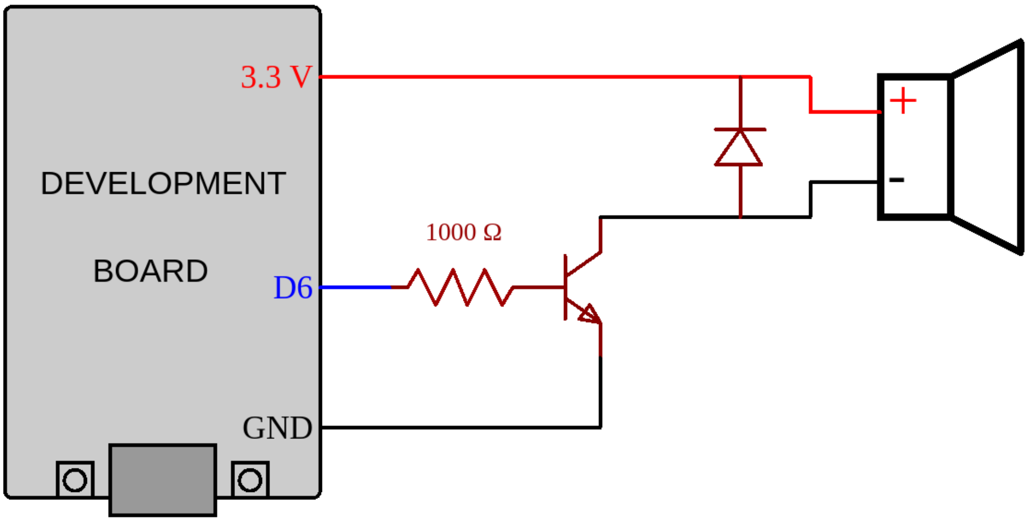
A diode is an electrical device allowing current to move in one direction. The most common kind of diode in modern circuit design is the semiconductor diode. Diodes can be used as rectifiers, signal limiters, voltage regulators, switches, signal modulators, oscillators etc. Below, you can find the schematic symbol and the real component appearance.



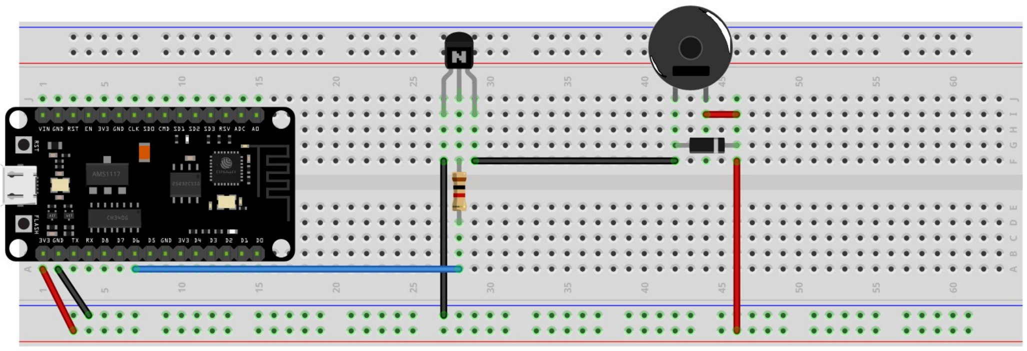


12.4 Connections

Below you can find the schematic:



Also, you can find below a visual representation of the project:



12.5 Code

The concept of the code is similar to the one used to turn on/off an LED, but this time we will change the code to turn on/off a buzzer.

The code can be found in the folder called **"Lesson 9: Buzzer"**.

The code starts with the declaration of the pin used by the buzzer. We will use **D6** for the buzzer. In the **setup()** function we will set the pin as **OUTPUT**.

Code 12.5.1 The setup() function

```
void setup()
{
  //set the mode of the pin used by the buzzer as OUTPUT
  pinMode(buzzer, OUTPUT);
}
```

In the **loop()** function we will turn on the buzzer at a frequency of 1000 Hz for 1 second, then turn it off for 1 second. To turn on the buzzer we use a function called **tone(pin, frequency)**, where the frequency can be as low as 31 Hz. To turn off the buzzer, we use **noTone(pin)**. Let's also turn on the buzzer at 500 Hz for 1 second, then turn it off for 1 second.

Code 12.5.2 The loop() function

```
void loop()
{
  //turn on the buzzer and wait 1 s
  //we use a frequency of 1kHz(1000Hz)
  //you can change this frequency so the sound can be
  //more pleasant
  tone(buzzer, 1000);
  delay(1000);

  //turn off the buzzer and wait 1 s
  noTone(buzzer);
  delay(1000);

  //turn on the buzzer at a frequency
  //of 500Hz and wait 1 s
  tone(buzzer, 500);
  delay(1000);

  //turn off the buzzer
  noTone(buzzer);
  delay(1000);
}
```

13. Lesson 10: Buzzer and Digital Inputs

13.1 Overview

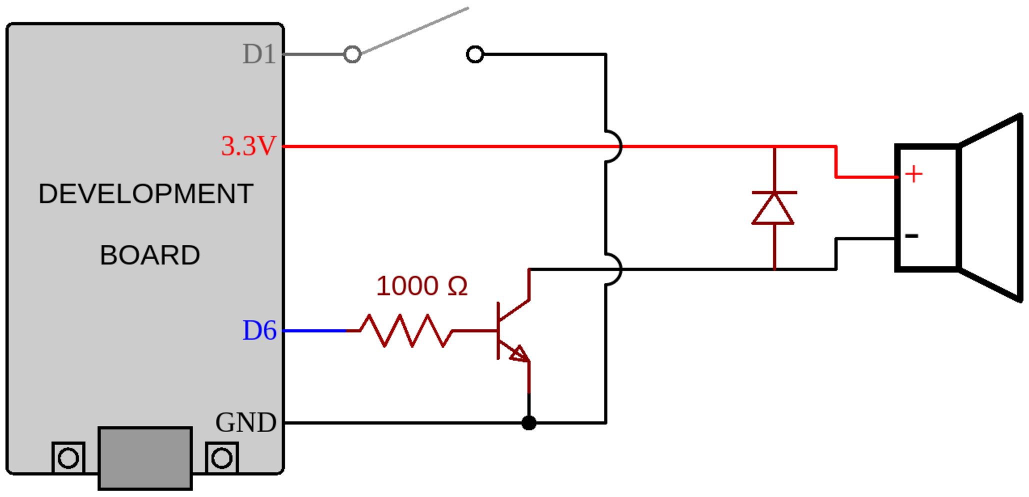
In this lesson, you will learn how to connect and use a buzzer along with a push button as a digital input.

13.2 Components required

- Development board;
- Breadboard 830p;
- 1 x push button;
- 1 x passive buzzer;
- Diode;
- Transistor;
- 1 x 1000 Ω resistor;
- 10 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

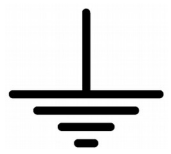
13.3 Connections

Below you can find the schematic:

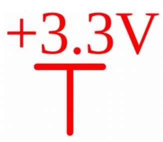


This scheme looks a little complicated, so let's simplify it by using some new electrical symbols. Sometimes, on really busy schematics, you can assign special symbols to node voltages. You can connect devices to these one-terminal symbols, and it'll be tied directly to 5V, 3.3V, VCC, or GND (ground).

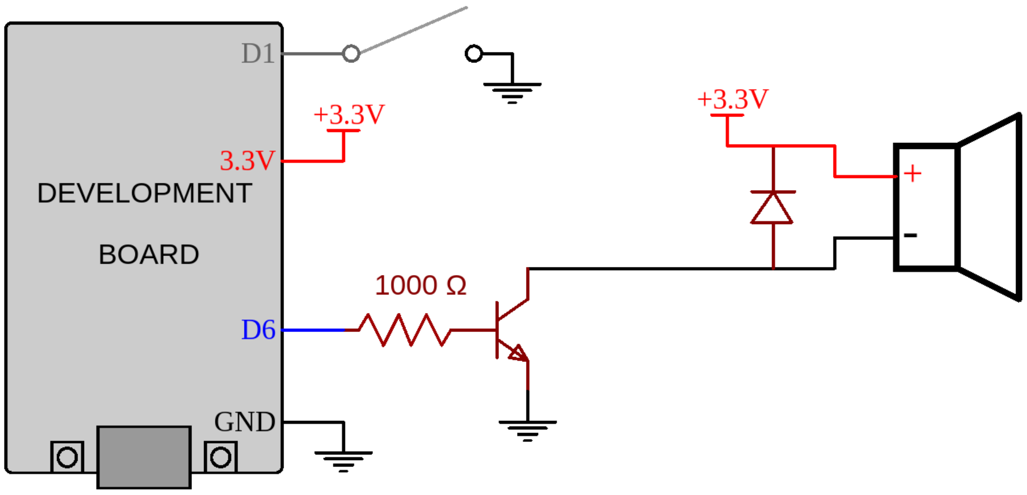
For GND (ground) we will use the next symbol:



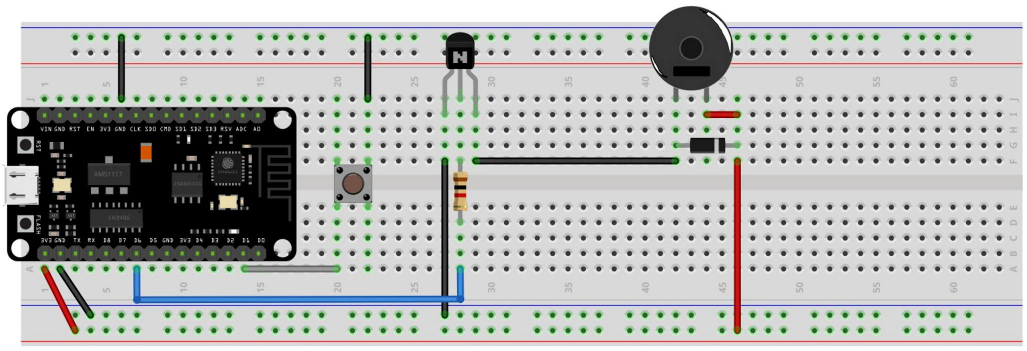
For +3.3V we will use:



Using the above defined symbols, our scheme is equivalent with:



We will be using these symbols when our schemes will be too complicated to follow, so keep them in mind. Also, below is a visual representation of the project, check it too for a better understanding.



13.4 Code

The code is similar with the one in the previous lesson, the only difference is the push button. You can find the code in the folder **"Lesson 10: Buzzer and Digital Inputs"**.

After the declaration of the pins used by the buzzer and the button, in the **setup()** function we need to set the mode of the pins, therefore the buzzer will be set as **OUTPUT** and the button as **INPUT_PULLUP**.

Code 13.4.1 The setup() function

```
void setup()
{
  //set the mode of the pin used by the buzzer as OUTPUT
  pinMode(buzzer, OUTPUT);

  //in order to read the state of a button, firstly, you have
  //to set the mode of the pin used by it as INPUT_PULLUP
  pinMode(button, INPUT_PULLUP);
}
```

In the **loop()** function we have to read the state of the button using **digitalRead(button)**, and if the button is pressed (**digitalRead(button)** == **0**), the buzzer will turn on at a frequency of 1000 Hz (you can modify this frequency). After the **if** statement we need to put a **while** loop to stop the **loop()** function from rolling, with the argument **digitalRead(button)** == **0**. If we leave the **while** loop without any instruction to run, the board will crash. So we need to put a **yield()** (this function is created only for ESP8266), which calls on the background functions to allow them to do their things. At the end of the **loop()** function we will put a command to turn off the buzzer and a delay of 100 ms.

Code 13.4.2 The loop() function

```
void loop()
{
  //read the current state of the button
  //if the button is pressed, turn on the buzzer
  if(digitalRead(button) == 0)
  {
    //we use a frequency of 1kHz(1000Hz)
    //you can change this frequency so the sound can be
    //more pleasant
    tone(buzzer, 1000);
  }

  //if the button is pushed down, we need something
  //to keep the loop() function from running
  while(digitalRead(button) == 0)
  {
    //if we leave the while with no instructions to do
    //the board will crash, because it will stay too long
    //in a while loop() and other processes will not be able to run
    yield();
  }

  noTone(buzzer);

  //wait 0.1 s
  delay(100);
}
```

14. Lesson 11: Buzzer and Ultrasonic

14.1 Overview

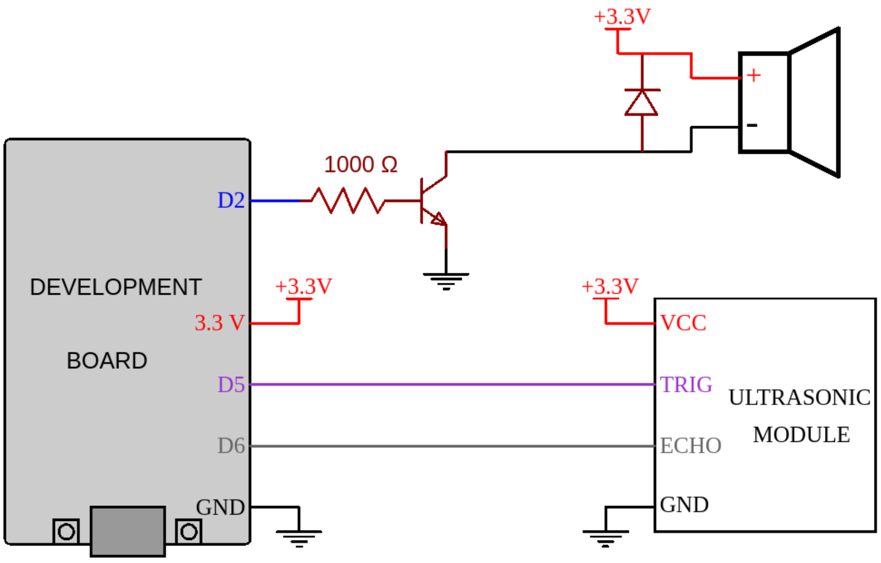
This lesson combines two lessons, lesson 9, Buzzer, and lesson 5, Ultrasonic HC-SR04+, and you will learn how to trigger a buzzer depending on distance.

14.2 Components required

- Development board;
- Breadboard 830p;
- Ultrasonic module HC-SR04+;
- 1 x passive buzzer;
- Diode;
- Transistor;
- 1 x 1000 Ω resistor ;
- 11 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

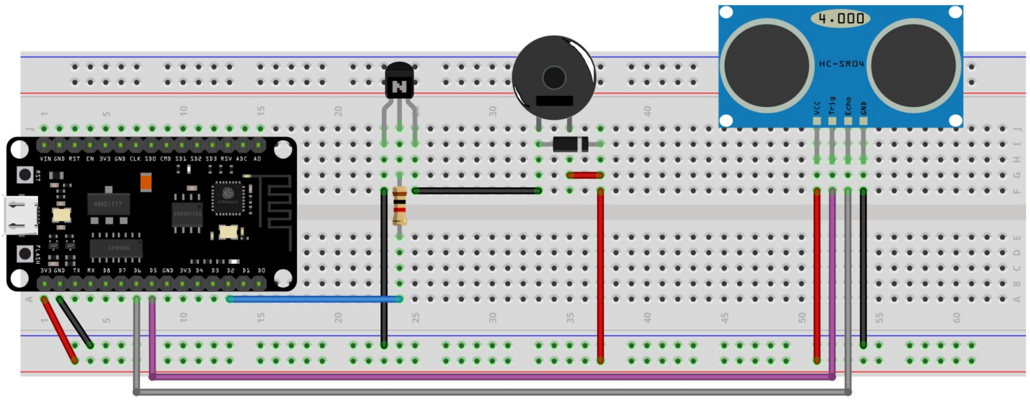
14.3 Connections

Below, you can find the schematic:



In the scheme above we used two symbols defined in the previous lesson. If you do not understand how to do all the connections, please check the previous lesson, the visual representation below and, also, you can find the scheme without using the symbols for +3.3V and GND in the folder **"Lesson 11: Buzzer and Ultrasonic"**.

Next, you can find a visual representation of the project:



14.4 Code

The code for this lesson combines the code from the two lessons described at the beginning of this lesson.

This lesson has a similar principle with the lesson **RGB LED and Ultrasonic**. In that lesson, when the distance was lower than 25 cm, the red LED started to flash. This time, when the distance is lower than 25 cm, the buzzer will start to beep at an interval of 50 ms. When distance is higher than 25 cm, the buzzer will be turned off. The code for this lesson can be found in the folder **"Lesson 11: Buzzer and Ultrasonic"**.

Like in the lesson **RGB LED and Ultrasonic**, we have to declare the pins used by the ultrasonic module, **trigger** and **echo**, the pin used by the buzzer and two variables used later for calculating the distance. In the **setup()** function we will begin by starting a serial communication with the computer, then set the pin used by the buzzer as **OUTPUT**, the trigger as **OUTPUT** and the echo as **INPUT**.

Code 14.4.1 The setup() function

```
void setup()
{
  //start the serial communication with the computer at 115200 bits/s
  Serial.begin(115200);

  //set the mode of the pin used by the buzzer as OUTPUT
  pinMode(buzzer, OUTPUT);

  //the trigger pin (transmitter) must be set as OUTPUT
  pinMode(trigPin, OUTPUT);

  //the echo pin (receiver) must be set as INPUT
  pinMode(echoPin, INPUT);
}
```

In the **loop()** function we will first calculate the distance. To calculate the distance we need the travel time of the sound wave generated by turning the transmitter **ON** for 10 microseconds. After finding the travel time (**duration** in our code), the distance can be calculated with the following formula:

```
distance = duration*0.034/2;
```

Next, we will use an **if** statement to decide if the distance is lower than 25 cm, then we will turn On and Off the buzzer at a frequency of 1000 Hz with a 50 ms interval. If the distance is higher than 25 cm, the buzzer is turned Off.

Code 14.4.2 The loop() function

```
void loop()
{
  //set the trigPin to LOW in order to prepare for the next reading
  digitalWrite(trigPin, LOW);

  //delay for 2 microseconds
  delayMicroseconds(2);

  //generate an ultrasound for 10 microseconds then turn off the transmitter
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  //reads the echoPin, returns the sound wave travel time in microseconds
  duration = pulseIn(echoPin, HIGH);

  //using the formula shown in the guide, calculate the distance
  distance = duration*0.034/2;

  if(distance < 25)
  {
    //creating the beeping effect
    //turn on the buzzer at a
    //frequency of 1kHz(1000Hz)
    //and wait 50 ms
    tone(buzzer, 1000);
    delay(50);

    //turn off the buzzer
    //and wait 50 ms
    noTone(buzzer);
    delay(50);
  }
  else
  {
    //turn off the buzzer
    noTone(buzzer);
  }

  //wait for 0.1s
  delay(100);
}
```

15. Lesson 12: Play songs with a buzzer

15.1 Overview

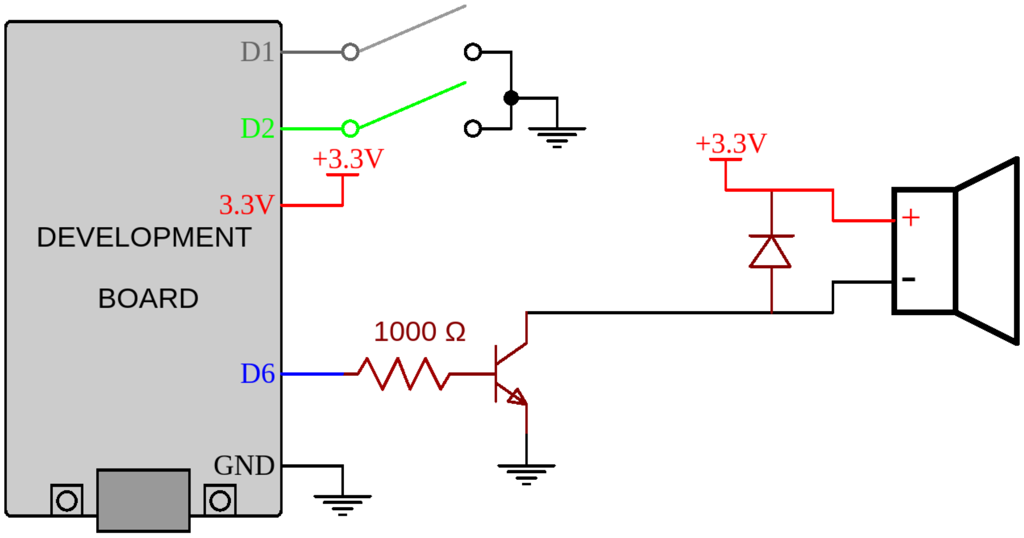
In this lesson you will learn how to play songs using the buzzer. We have created two songs that you can play with the buzzer. If you are more talented, you can write more complicated songs.

15.2 Components required

- Development board;
- Breadboard 830p;
- 1 x passive buzzer;
- Diode;
- Transistor;
- 2 x push button;
- 1 x 1000 Ω resistor ;
- 12 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

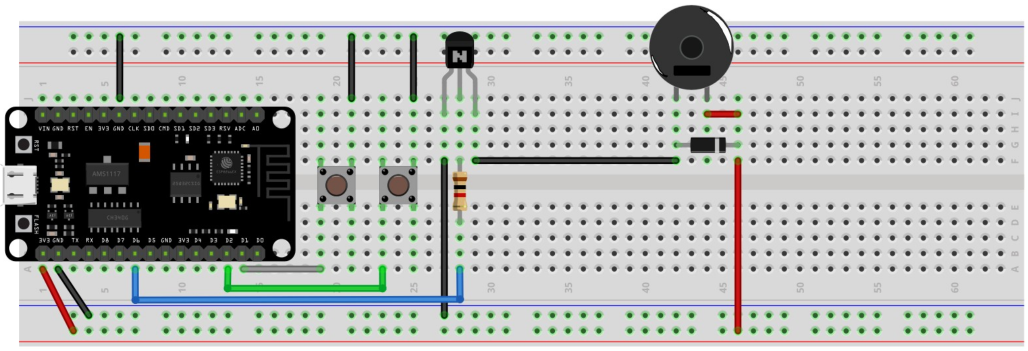
15.3 Connections

Below you can find the schematic:



You can find the scheme without the electrical symbols for GND and 3.3V in the folder called **"Lesson 12: Play song with a buzzer"**.

Below, you can find a visual representation of the project:



15.4 Code

The code for this lesson can be found in the folder called **"Song"** which is inside the folder **"Lesson 12: Play song with a buzzer"**.

The code starts with the declaration of the approximative frequencies for the notes for all the seven octaves, then declaring the pins used by the buzzer and the two buttons. In the **setup()** function we will set the pin used by the buzzer as **OUTPUT** and the pins for the buttons as **INPUT_PULLUP**.

Code 15.4.1 The setup() function

```
void setup()
{
  //set the mode of the pins
  pinMode(buzzer, OUTPUT);
  pinMode(button1, INPUT_PULLUP);
  pinMode(button2, INPUT_PULLUP);
}
```

In the **loop()** function we will use two **if** statements and if the first button was pressed, then the buzzer will play the first song (ABC Alphabet), or if the second button was pressed, then the buzzer will play the second song (Old Mcdonald had a farm). Every song is defined by a function: first song (**abc_song()**) and the second song (**old_mcdonald()**).

Code 15.4.2 The loop() function

```
void loop()
{
  //if the first button was pressed, play the first song
  if(digitalRead(button1) == 0)
  {
    //play ABC Alphabet song
    abc_song();
  }

  //if the second button was pressed, play the second song
  if(digitalRead(button2) == 0)
  {
    //play the second song
    old_mcdonald();
  }
}
```

The **abc_song()** function created for the first song starts with a declaration of the waiting times between notes and the playing times for notes. The song has six parts, each part contains multiple notes. We will turn on the buzzer and play a note for a specified time, then turn off the buzzer for the specified time. In the snipped below, you can find one of the six parts of the song.

Code 15.4.3 The function for the first song

```
//define the waiting times
int delay1 = 700;
int delay2 = 1000;
int delay3 = 300;
int delay4 = 100;

//play every note indicated for delay1 (or delay2, or delay3)
//milliseconds, then turn off the buzzer for delay4 milliseconds
//then play the next note

//first sequence
//play: C C G G A A G
tone(buzzer, c7);
delay(delay1);
noTone(buzzer);
delay(delay4);

tone(buzzer, c7);
delay(delay1);
noTone(buzzer);
delay(delay4);

tone(buzzer, g7);
delay(delay1);
noTone(buzzer);
delay(delay4);

tone(buzzer, g7);
delay(delay1);
noTone(buzzer);
delay(delay4);

tone(buzzer, a7);
delay(delay1);
noTone(buzzer);
delay(delay4);

tone(buzzer, a7);
delay(delay1);
noTone(buzzer);
delay(delay4);

tone(buzzer, g7);
delay(delay2);
noTone(buzzer);
delay(delay4);
```

The **old_mcdonald()** function created for the second song starts with the declaration of the waiting times between notes and the playing times for notes. The song has eight parts, each part contains multiple notes. We will turn on the buzzer and play a note for a specified time, then turn off the buzzer for the specified time. In the snipped below, you can find one of the eight parts of the song.

Code 15.4.4 The function for the second song

```
//play every note indicated for time1 (or time2,
//or time3, or time4, or time5) milliseconds,
//then turn off the buzzer for time0 milliseconds
//then play the next note
int time0 = 100;
int time1 = 300;
int time2 = 600;
int time3 = 900;
int time4 = 550;
int time5 = 200;

//first part
//play: C C C G A A G
tone(buzzer, c4);
delay(time1);
noTone(buzzer);
delay(time0);

tone(buzzer, c4);
delay(time1);
noTone(buzzer);
delay(time0);

tone(buzzer, c4);
delay(time1);
noTone(buzzer);
delay(time0);

tone(buzzer, g3);
delay(time1);
noTone(buzzer);
delay(time0);

tone(buzzer, a4);
delay(time1);
noTone(buzzer);
delay(time0);

tone(buzzer, a4);
delay(time1);
noTone(buzzer);
delay(time0);

tone(buzzer, g3);
delay(time2);
noTone(buzzer);
delay(time0);
```

This code is too long, so let's simplify it. The modified code can be found in the folder called **Song_simplified**, which is inside the **"Lesson 12: Play songs with a buzzer"** folder. The declaration, **setup()** function and **loop()** function will remain the same, we only have to modify the **abc_song()** function and **old_mcdonald()** function. For each sequence we will create two arrays that will store the order of the notes and the playing time for each note. Then, we will use a **for** loop to go through all the elements of the arrays and play the notes in order for the right time (same as in the long code). Below you can find two sequences of the first song:

Code 15.4.5 The code for the first song, simplified

```
//first sequence
//play: C C G G A A G
//create two arrays: one that stores the notes in order and
//another that stores the playing time for each note
const int sequence_1_notes[] = {c7, c7, g7, g7, a7, a7, g7};
const int sequence_1_delays[] =
{
  delay1, delay1, delay1, delay1, delay1, delay1, delay2
};

//using the for loop we will play all the notes in the correct
//order
for(int i = 0; i < 7; i++)
{
  //play "i" note
  tone(buzzer, sequence_1_notes[i]);
  //time for playing the "i" note
  delay(sequence_1_delays[i]);
  //turn off the buzzer
  noTone(buzzer);
  delay(delay4);
}

//second sequence
//play: F F E E D D D D C
const int sequence_2_notes[] = {f7, f7, e7, e7, d7, d7, d7, d7, c7};
const int sequence_2_delays[] =
{
  delay1, delay1, delay1, delay1, delay3, delay3, delay3, delay3, delay2
};

for(int i = 0; i < 9; i++)
{
  tone(buzzer, sequence_2_notes[i]);
  delay(sequence_2_delays[i]);
  noTone(buzzer);
  delay(delay4);
}
```

The structure of the function created for the second song, **old_mcdonald()**, is the same as for the one created for the first song. Below you can find a part of the second song:

Code 15.4.6 The code for the second song, simplified

```
  //first part
  //play: C C C G A A G
  const int sequence_1_notes[] = {c4, c4, c4, g3, a4, a4, g3};
  const int sequence_1_delays[] =
  {
    time1, time1, time1, time1, time1, time1, time2
  };

  for(int i = 0; i < 7; i++)
  {
    tone(buzzer, sequence_1_notes[i]);
    delay(sequence_1_delays[i]);
    noTone(buzzer);
    delay(time0);
  }
```

One sequence is different because we are using only one note. So, the array for the notes is composed from a single note, and in the **for** loop we will play on the buzzer that note for the entire running of the **for** loop.

Code 15.4.7 The sixth part of the second song

```
  //sixth part
  //play: C C C C C C C C C C C C
  const int sequence_6_notes[] = {c4};
  const int sequence_6_delays[] =
  {
      time5, time5, time1, time5, time5, time1, time5, time5, time5, time5,
time1, time1
  };

  for(int i = 0; i < 12; i++)
  {
    tone(buzzer, sequence_6_notes[0]);
    delay(sequence_6_delays[i]);
    noTone(buzzer);
    delay(time0);
  }
```

16. Theory lesson: Object-Orienteed Programming (OOP)

Programming paradigms are a way of classifying programming languages by their features. There are several paradigms (procedural, object oriented, functional, logic etc.), but the one that we are going to cover in this tutorial is the object-oriented one.

Object Oriented Programming (OOP) is a programming paradigm based on an abstract concept of objects. Objects are instances of classes, and they may contain **data** in the form of fields or/and **code** in the form of procedures, also known as methods. One of the most important features in OOP is that a procedure can access and even modify the **data fields** of the object.

Languages that support OOP use (in most cases) inheritance to reuse code and extend to a more complex implementation. For this, there are created classes, sub-classes and prototypes. The concepts of OOP are:

- **classes**: the definitions of the data format and procedures for a given type of object.
- **objects**: instances of classes.

A class example in **C++** and **arduino**:

Code 16.1 Example of a class

```
class Triangle
{
	// Access specifier. Don't worry about it right now.
	public:

	// Data Members
	int id;
	float edge1;
	float edge2;
	float edge3;
};
```

This creates a **Triangle** class that has the attributes of an **id, edge1, edge2, edge3.** The 'edge' attributes represent the length of the edges. The **id** is an integer, and the **edges** are float(real numbers).

Great! Now we can create objects using this "pattern".

To create an object of this type we can declare it like any other data types:

```
Triangle firstTriangle;
```

To change an attribute we can access it using the '.' (dot) operator:

```
firstTriangle.id = 1234;
```

Now the **id** attribute of the **firstTriangle** object has the value 1234.

A class can also contain **procedures** or **methods**. They are called like this because there can be more that one way to do what you intend in the program, but you use the one that is appropriate for the purpose of your program.

As an example, to calculate the semi-perimeter of a triangle we can add all the edges together and divide them by two, or divide every edge by two and then add them together:

Code 16.2 Calculate the semi-perimeter of a triangle

```
float getSemiPerimeter()
{
  float semiPerimeter = (this->edge1 + this->edge2 + this->edge3)/2;
  return semiPerimeter;
}
```

The **this->** pointer accesses the fields of the object for which the method was called. To call any of these methods we use the following line of code:

```
firstTriangle.getSemiPerimeter();
```

We can translate it like this: for the **firstTriangle** object do the **getSemiPerimeter()** method. But as this method returns the 'semiPerimeter' as a **float** we should store it in a variable like so:

```
float semiPerimeter = firstTringle.getSemiPerimeter();
```

The code for a full program in **C++** that prints to the **CLI** (command line interface) the semi-perimeter of a triangle should look like this:

Code 16.3 The full C++ program

```
#include <iostream>
using namespace std;

class Triangle
{
    public:

    int id;
    float edge1;
    float edge2;
    float edge3;

    float getSemiPerimeter()
    {
        float semiPerimeter = this->edge1/2 + this->edge2/2 + this->edge3/2;
        return semiPerimeter;
    }
};

int main()
{
    Triangle firstTriangle;

    firstTriangle.id = 1234;
    firstTriangle.edge1 = 5;
    firstTriangle.edge2 = 4;
    firstTriangle.edge3 = 2;
    float semiPerimeter = firstTriangle.getSemiPerimeter();

      cout << firstTriangle.id << " triangle has a semiperimeter of " <<
semiPerimeter;
    return 0;
}
```

You can run this program using an online service, for example:

http://cpp.sh/

When we created the **firstTriangle** object earlier we declared it like any other data type, but ‘in the back’ the compiler calls the default constructor in this case, which creates the instance.

C++ constructors

A constructor is a special type of member function that initialises an object automatically when it’s called.

NOTE! A constructor is **declared in the same class** as the object that needs to be created.

The default constructor is the constructor that takes no arguments (has no parameters). It does not need to be declared first. Here it how it looks like:

```
Triangle (){  }
```

And here is how it can be declared in the class:

Code 16.4 Declare a constructor in a class

```
class Triangle
{
    public:

    int id;
    float edge1;
    float edge2;
    float edge3;

    Triangle() {  }
};
```

You can change this constructor for your purposes. For example:

Code 16.5 Constructor in a class

```
class Triangle {
    public:

    int id;
    float edge1;
    float edge2;
    float edge3;

    Triangle()
    {
        edge1 = -1;
        edge2 = -1;
        edge3 = -1;
    }
};
```

This constructor initializes the edge **lengths** of a newly created triangle to **-1** to show that a **true** length has not been given (lengths can't be lower than 0 physically). Now if you declare a new object of the **Triangle** type it will automatically have those edge values because that constructor will be called.

Parameterized constructors

As said earlier a constructor is a special type of method function, so it can get arguments or parameters when it's called. In general these arguments help initialise an object. For example in our case we can set all the edges and the **id** of the triangle.

Code 16.6 Parameterized constructors

```
Triangle (int id, float edge1, float edge2, float edge3)
{
    this->id = id;
    this->edge1 = edge1;
    this->edge2 = edge2;
    this->edge3 = edge3;
}
```

You can call this constructor using:

```
Triangle secondTriangle(3, 4, 5, 6);
```

This will create an **object** of the **Triangle** class with the given parameters. **NOTE!** If you create a constructor with parameters, and then want to use the default constructor, you need to re-declare it in the class body.

Code 16.7 Default constructor and constructor with parameters

```
class Triangle
{
    public:

    int id;
    float edge1;
    float edge2;
    float edge3;

    //we need to re-declare the default constructor if we want to use it
    //it’s good to declare it anyway
    Triangle (){ }

    //this is the parameterized constructor
    Triangle (int id, float edge1, float edge2, float edge3)
    {
        this->id = id;
        this->edge1 = edge1;
        this->edge2 = edge2;
        this->edge3 = edge3;
    }
};
```

C++ destructors

As said earlier, a constructor is a special type of member function that initialises an object automatically when it’s called. A **destructor,** also a member function, as opposed to the constructor is used to deallocate the memory used when creating the objects. A destructor is **automatically called** when that object becomes out of scope or it is explicitly deleted. It has the same name as the class prefixed by a '~'(tilde). The destructor for the **Triangle** class looks like this:

```
~Triangle () {  }
```

A destructor takes no arguments and has no return type. If there are no user-defined destructors, and one is needed, the compiler implicitly declares a destructor.

You can use the destructor to show when an object is deleted. In the next example we print to the command line interface(**CLI**) that the destructor has been called.

```
~Triangle ()
{
   cout<<"Destructor has been called";
}
```

Inheritance

In **OOP** inheritance is the mechanism of basing an object or a class upon another object or class. In classic OOP languages an object created from a subclass acquire all the properties and behaviors from the parent object (with a few exceptions that are logical).

Let's say we have the **equilateralTriange** subclass which extends the **Triangle** class. We declare a subclass like so in C++:

```
class equilateralTriangle: public Triangle {  };
```

But there would be no point in having an empty class, so as an example we can put a method that returns the triangle area, as for an equilateral triangle we can calculate it with the formula:

$$A=\frac{l^2\times\sqrt{3}}{4}$$

As the formula has a square root in it, we need to import the 'math.h' library so we can use the 'sqrt()' function. So, the subclass should look like this:

Code 16.8 Subclass

```
class equilateralTriangle: public Triangle
{
    public:

    float getArea()
    {
        return (this->edge1 * this->edge1 * sqrt(3)) / 4;
    }

    equilateralTriangle () { }
};
```

Now you can create objects that are of the **equilateralTriangle** type which have all the data fields in the **Triangle** class as the equilateralTriangle class **extends** (inherits) the Triangle class.

This is an example of how to use the subclass in a program.

Code 16.9 Program with subclasses

```
#include <iostream>
#include <math.h>

using namespace std;

class Triangle
{
    public:

    int id;
    float edge1;
    float edge2;
    float edge3;

    float getSemiPerimeter()
    {
        float area = this->edge1/3 + this->edge2/3 + this->edge3/3;
        return area;
    }

    Triangle () { }
};

class equilateralTriangle: public Triangle
{
    public:

    float getArea()
    {
        return (this->edge1 * this->edge1 * sqrt(3)) / 4;
    }

    equilateralTriangle () { }
};

int main()
{
    equilateralTriangle eTriangle;
    eTriangle.id = 1234;
    eTriangle.edge1 = 2;

    float area = eTriangle.getArea();

    cout<<eTriangle.id<<" triangle has an area of "<<area;

    return 0;
}
```

The subclasses, being classes, can hold data members, not only methods (functions). Like so:

Code 16.10 Subclasses with data members

```
class equilateralTriangle: public Triangle
{
    public:

    float area;

    float getArea()
    {
        return this->area;
    }

    equilateralTriangle () { }
};
```

We can set it from another function, although it is better to have methods in the class to do so, or to use the constructor to set the area.

Code 16.11 Subclass

```
class equilateralTriangle: public Triangle
{
    public:

    float area;

    float getArea()
    {
        return this->area;
    }

    equilateralTriangle(int edge)
    {
        this->edge1 = this->edge2 = this->edge3 = edge;
        this->area = (this->edge1 * this->edge1 * sqrt(3)) / 4;
    }

    equilateralTriangle () { }
};
```

This class example has a setArea(); method which sets the area of the object that the method is called for.

Code 16.12 Create new methods in subclass

```
class equilateralTriangle: public Triangle
{
    public:

    float area;

    void setArea()
    {
        this->area = (this->edge1 * this->edge1 * sqrt(3)) / 4;
    }

    float getArea()
    {
        return this->area;
    }

    equilateralTriangle(int edge)
    {
        this->edge1 = this->edge2 = this->edge3 = edge;
        this->setArea();
    }

    equilateralTriangle () { }
};
```

This is the way we recommend you to do it, as you can set the area through your parameterized constructor and if you don't use the constructor in which you call the method, you can call it another time.

Overloading functions or methods

Method (function) overloading is a feature that allows a class to have more than one function with the same name while their arguments are different.

This feature stands on the function signature concept. A function signature is the name of the function, plus the arguments that are passed to it.

There are 3 ways to overload a method (let's suppose we have a function **myFunction** that does something):

- by numbers of parameters:

```
myFunction(int, int);
myFunction(int, int, int);
```

If we call **myFunction** with two integers as parameters, the program will use the first function definition. If we call **myFunction** with three integers as parameters the program will use the second function definition.

- by the data type of the parameters:

```
myFunction(int, int);
myFunction(float, int);
```

If we call **myFunction** with two integers as parameter, the program will use the first function definition. If we call **myFunction** with a float parameter and then an integer the program will use the second function definition.

- by the order of the parameters data type:

```
myFunction(int, float);
myFunction(float, int);
```

If we call **myFunction** with an integer and then a float the program will use the first function definition. If we call **myFunction** with a float parameter and then an integer the program will use the second function definition.

Note! To overload functions you only need one of the ways from the above, but, also you can use combinations of the three.

Overriding functions or methods

Method (function) overriding in OOP is a language feature that allows a subclass to have it's own implementation of a function that is also implemented in a superclass. This is used for more efficient approaches in processing tasks that may be easier to compute in a special way due to subclasses objects properties.

So, in our examples we can conclude that in a triangle to calculate the semi-perimeter, we add all the edges together and divide by two, while to calculate the semi-perimeter of an equilateral triangle we can do it like this or, easier, multiply one edge by three and divide the result by two.

Code 16.13 Overriding

```
class Triangle
{
    public:

    int id;
    float edge1;
    float edge2;
    float edge3;

    float getSemiPerimeter()
    {
        float semiPerimeter = this->edge1/2 + this->edge2/2 + this->edge3/2;
        return semiPerimeter;
    }

    Triangle () { }
};

class equilateralTriangle: public Triangle
{
    public:

    float area;

    float getSemiPerimeter()
    {
        return this->edge1 * 3 / 2;
    }

    void setArea()
    {
        this->area = (this->edge1 * this->edge1 * sqrt(3)) / 4;
    }

    float getArea()
    {
        return this->area;
    }

    equilateralTriangle(int edge)
    {
        this->edge1 = this->edge2 = this->edge3 = edge;
        this->setArea();
    }

    equilateralTriangle () { }
};
```

So, if you call the **getSemiPerimeter()** method on a **Triangle** object the program will call the function implemented in the **Triangle** class, while if you call the **getSemiPerimeter()** method on an **equilateralTriangle** object, the program will call the **getSemiPerimeter()** method implemented in the **equilateralObject** class.

```
firstTriangle.getSemiPerimeter();
```

VS.

```
eTriangle.getSemiPerimeter();
```

Access specifiers

Access specifiers (or access modifiers) are keywords that, in **OOP** languages set the accesibility of classes, methods or other members of objects. These keywords are used for **data encapsulation**.

Data encapsulation refers to the restricting of access to the objects, and objects components, as well as method function of thee classes.

So, as a conclusion, we use the access specifiers for security purposes, as well as a better data handling.

In **C++** there are three access specifiers:

- **public:** the members that are declared as public are accessible from anywhere outside the class through an object of the class.
- **protected:** the members that are declared as protected are accessible from outside the class **only if** the class from which they are used is a subclass.
- **private:** the members that are declared as private are only accessible from the same class.

So, in this **C++** program:

Code 16.14 Access Specifiers

```
class Triangle
{
    public:
        int edge1;
    protected:
        int edge2;
    private:
        int edge3;
};

class equilateralTriangle: public Triangle
{
    void setEdges(int edge1, int edge2, int edge3)
    {
        this->edge1 = edge1;//allowed, as edge1 is public
        this->edge2 = edge2;//allowed, as equilateral
        //Triangle extends Triangle, and edge2 is a protected member
        this->edge3 = edge3;//not allowed, as edge3 is a
        //private member and can only be seen from inside the class.
    }
};

int main()
{
    Triangle triangle;

    triangle.edge1 = 10;     //allowed, edge1 is a public
    //member and can be accesed outside the class

    triangle.edge2 = 20;     //not allowed, edge2 is
    //protected and in main() function we are not in the class or
    //subclass

    triangle.edge3 = 30;     //not allowed, edge2 is private
    //and in the main() function we are not in the same class as
    //the object
}
```

17. Lesson 13: DHT11

17.1 Overview

In this lesson you will learn how to use a DHT11 Temperature and Humidity Sensor to check the temperature and humidity in a room.

17.2 Components required

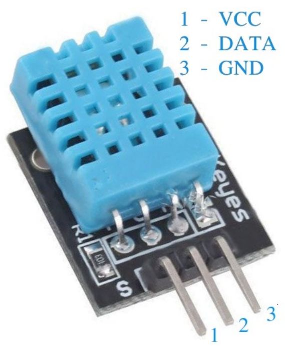
- Development board;
- Micro USB – Type A USB cable;
- DHT11 Humidity and Temperature sensor;
- Breadboard 830p;
- 1 x 5000 Ω resistor;
- 4 x male-to-male jumper wires;

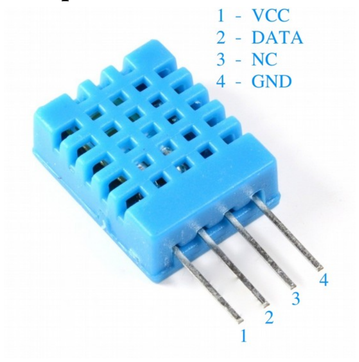
17.3 Component Introduction

DHT11

DHT11 features a temperature and humidity sensor with a calibrated digital signal output. The DHT11 detects water vapor by measuring the electrical resistance between two electrods.

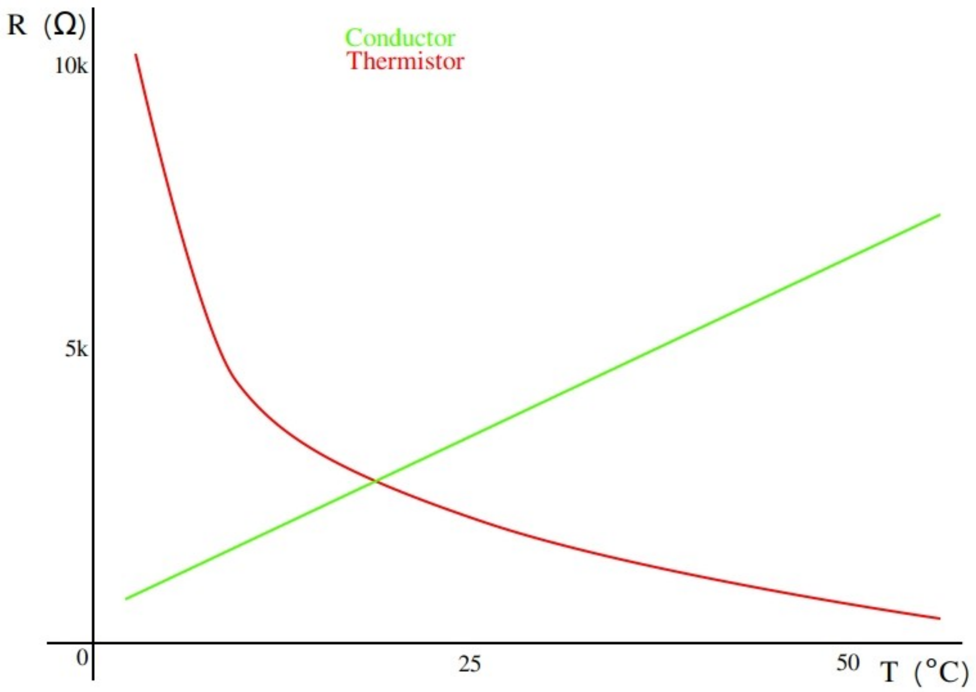
There are two types of a DHT11 sensor that you may come across. One with three pins and a 10k resistor included, or the one included in this kit that does not have a pull up resistor inside it and has 4 pins.





The humidity sensing component is a moisture holding substrate with electrodes applied to the surface. When water vapor is absorbed by the substrate, ions are released by the substrate which increases the conductivity between the electrodes. The change in resistance between the two electrodes is proportional to the relative humidity.

Here is a graph to show you what this means:

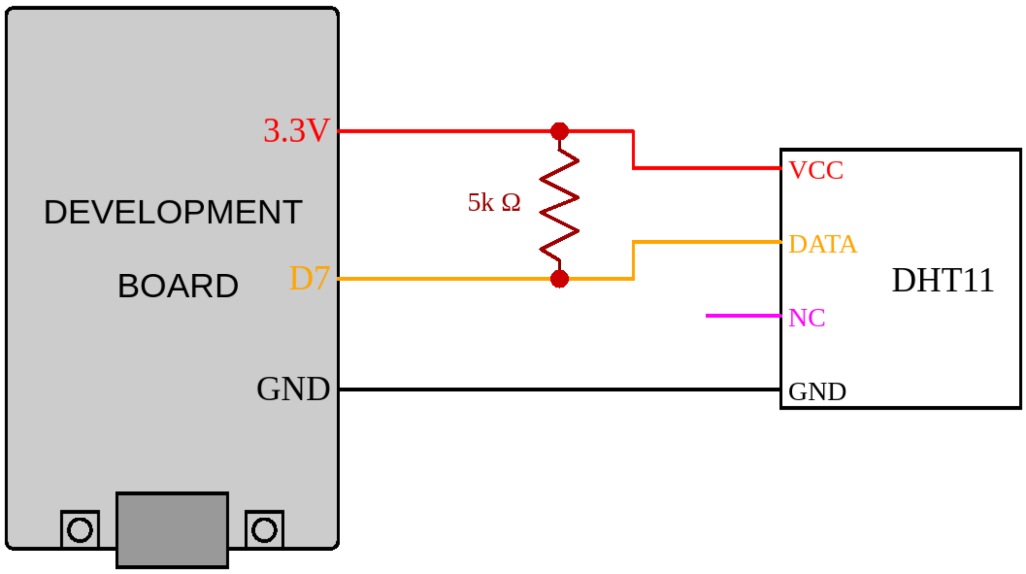


To read the temperature the sensor uses a thermistor. A thermistor is a doped semiconductor. Semiconductors resistance drops with a rise in temperature as opposed to conductors (whose resistance rises with a rise in temperature).

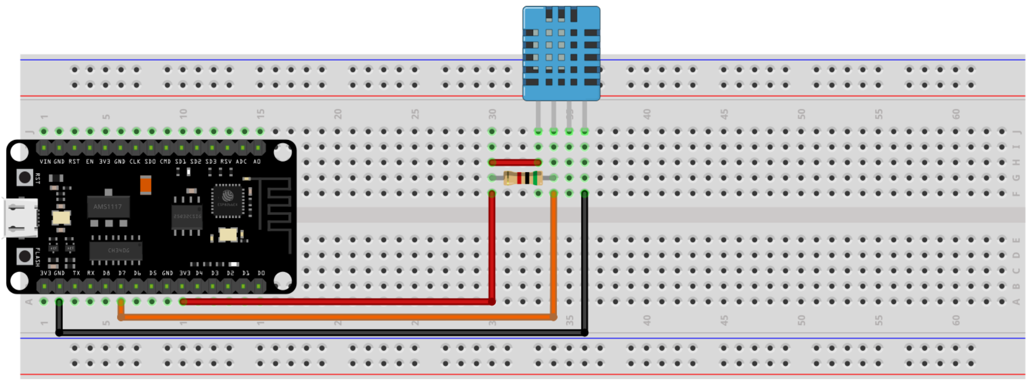
So basically the DHT11, to read the temperature, reads the resistance of the thermistor. The steeper slope at lower temperatures of the thermistor means that you can calculate the resistance with better accuracy and hence the temperature will be more accurate.

17.4 Connections

Below, you can find the schematic:



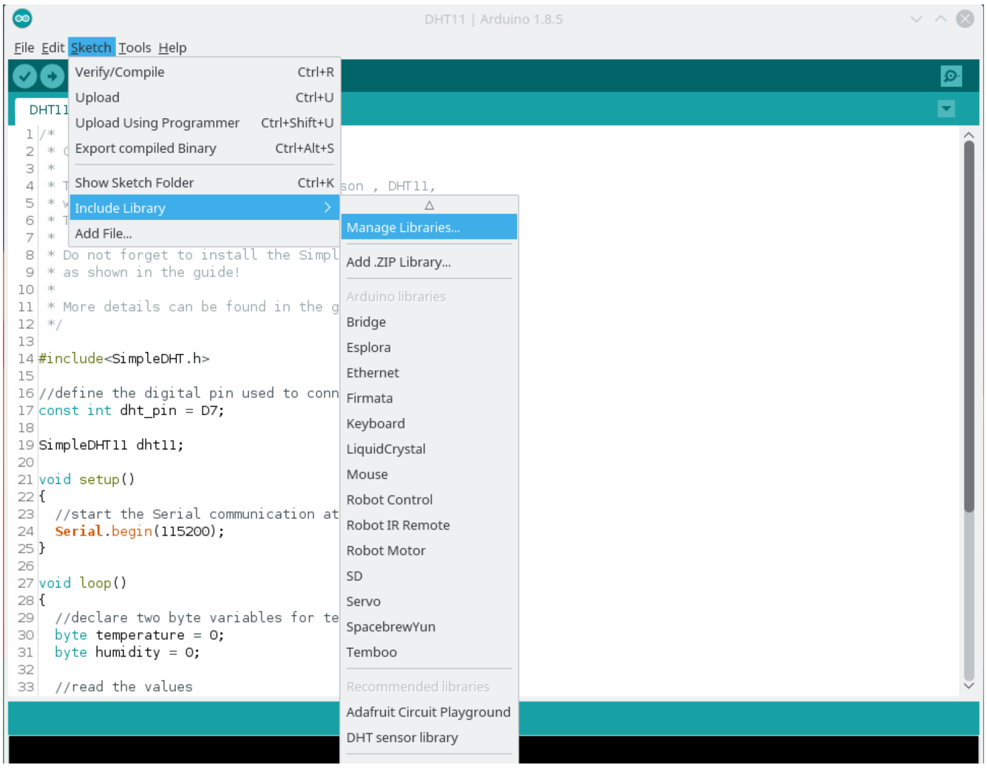
Below, you can find a visual representation of the connections:



17.5 Code

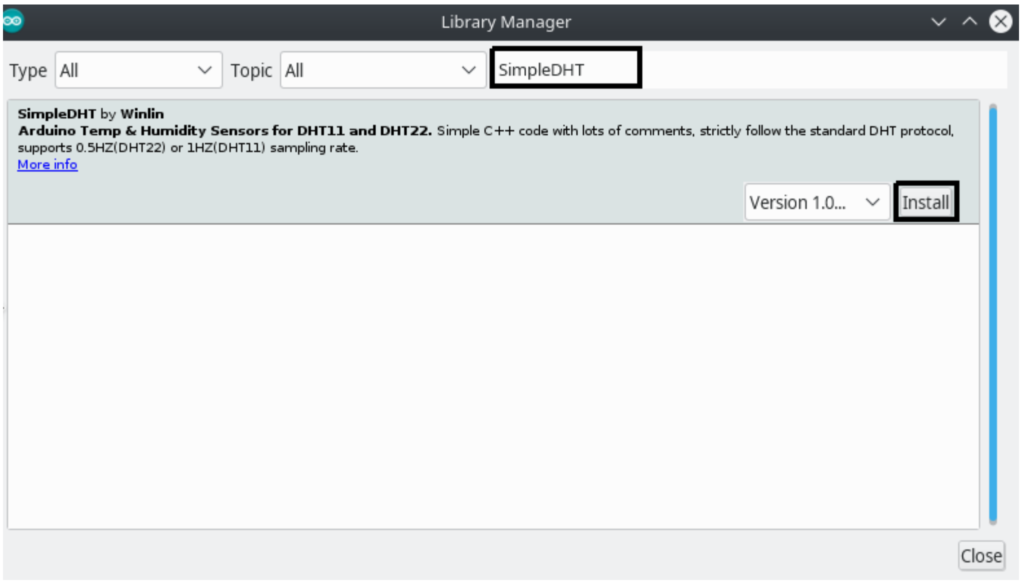
You can find the code in the folder **"Lesson 13: DHT11"**. Before uploading the code, make sure to download the **SimpleDHT** library:

- After opening the code, go to **Sketch>Include Library>Manage Libraries...**



```
/*
 * (
 *
 * T                                   son , DHT11,
 * v
 * T
 *
 * Do not forget to install the Simpl
 * as shown in the guide!
 *
 * More details can be found in the g
 */

#include<SimpleDHT.h>

//define the digital pin used to conn
const int dht_pin = D7;

SimpleDHT11 dht11;

void setup()
{
  //start the Serial communication at
  Serial.begin(115200);
}

void loop()
{
  //declare two byte variables for te
  byte temperature = 0;
  byte humidity = 0;

  //read the values
```

- Now search for **SimpleDHT** and hit **Install**.



SimpleDHT by **Winlin**
Arduino Temp & Humidity Sensors for DHT11 and DHT22. Simple C++ code with lots of comments, strictly follow the standard DHT protocol, supports 0.5HZ(DHT22) or 1HZ(DHT11) sampling rate.
More info

The code for this lesson is very simple. First, you have to include the previously installed library, then declare the pin used by the module and declare an instance of the **SimpleDHT11** library.

Code 17.5.1 Library and declaration

```
#include <SimpleDHT.h>

//define the digital pin used to connect the module
const int dht_pin = D7;

SimpleDHT11 dht11;
```

In the **setup()** we have to start a serial communication to communicate with the computer. In **loop()** we have to declare two byte variables to store the values registered by the module. Using the next command we read the values from the module. This is the syntax defined in the library, so do not modify it.

```
dht11.read(dht_pin, &temperature, &humidity, NULL);
```

Now, all you need to do is print the values in the Serial Monitor and wait 2 s before next read.

Code 17.5.2 The setup() and loop() functions

```
void setup()
{
  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);
}

void loop()
{
  //declare two byte variables for temperature and humidity
  byte temperature = 0;
  byte humidity = 0;

  //read the values
  dht11.read(dht_pin, &temperature, &humidity, NULL);

  //display the values in Serial Monitor
  Serial.print("Temperature: ");
  Serial.print(temperature);
  Serial.println(" *C");
  Serial.print("Humidity: ");
  Serial.print(humidity);
  Serial.println(" H");
  Serial.println();

  //wait 2 s
  delay(2000);
}
```

18. Lesson 14: Potentiometer and Servo Motor

18.1 Overview

In this lesson you are going to learn how to use a potentiometer to control a servo.

18.2 Components required

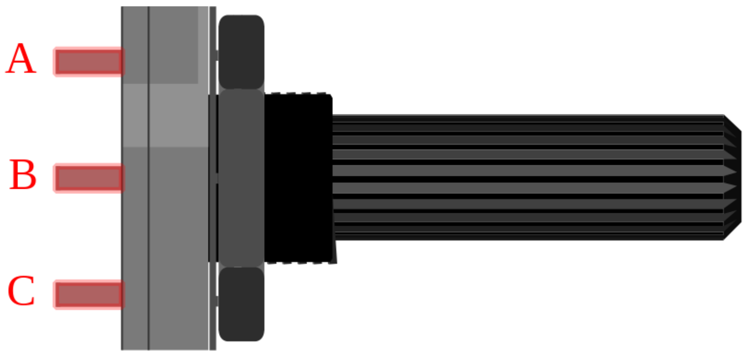
- Development board;
- Micro USB – Type A USB cable;
- 10k ohms Potentiometer;
- Servo SG90;
- 9 x male-to-male jumper wires;
- Breadboard power supply;
- Breadboard 830p;
- 9 V battery;
- Cable 9 V battery to DC jack.

18.3 Components introduction

Potentiometer

In the past lessons, you learned what a resistor is. A potentiometer is pretty similar to a resistor, the main difference is that the resistor has a fixed resistance, whereas a potentiometer can change its resistance (it is a variable resistor).

Below, you can see how a generic potentiometer looks like.



How it works:

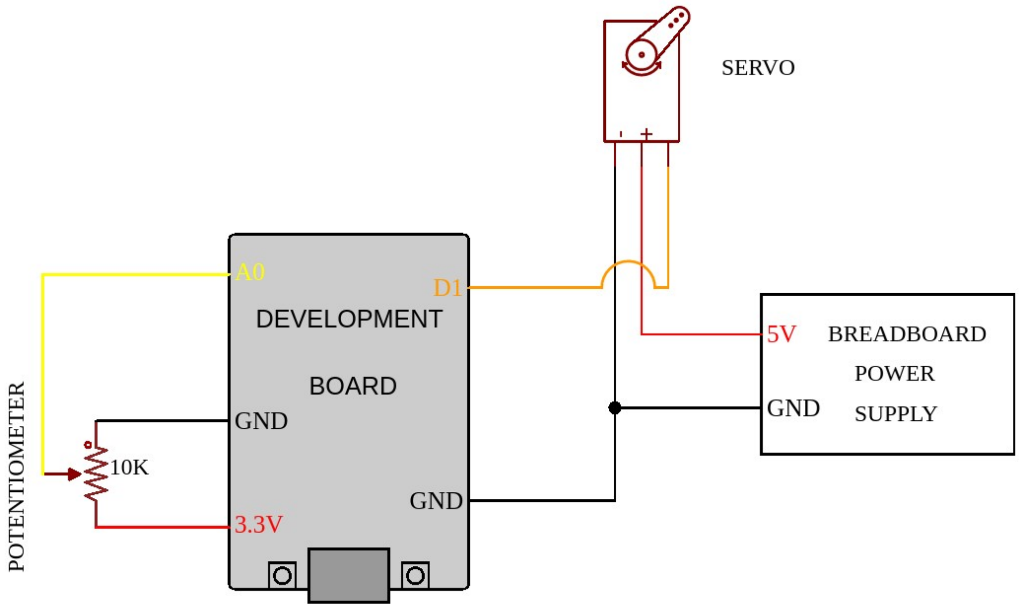
- the resistance measured between **A** and **C** is always 10k ohms, no matter if you turn the rod or not;
- when you turn the rod clockwise, the resistance between **A** and **B** increases (from 0 to 10k) and the resistance between **B** and **C** decreases (from 10k to 0).
- when you turn the rod counter clockwise, the resistance between **A** and **B** decreases (from 10k to 0) and the resistance between **B** and **C** increases (from 0 to 10k).

Servomotor

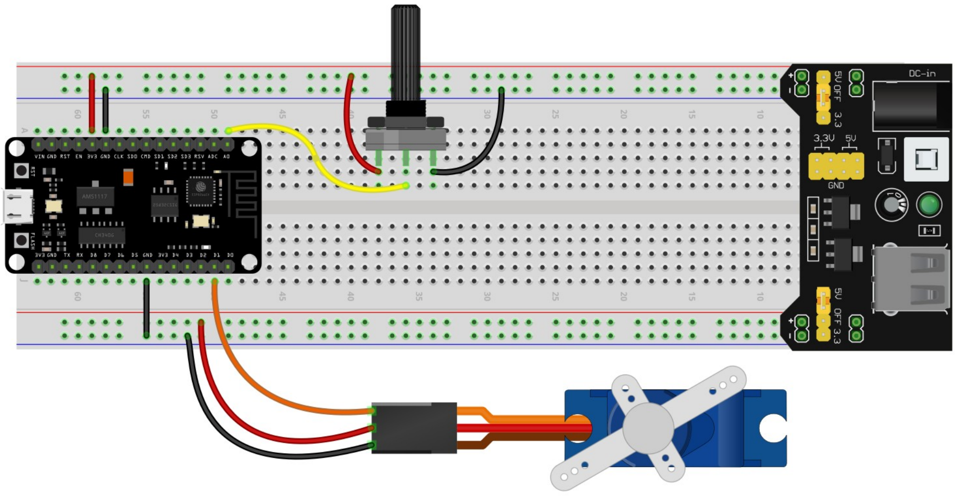
A servomotor is an active component that is able of turning its arm precisely. The servomotor included in the kit is able to turn from 0 degrees to 180 degrees. In order to control its movement, you have to send it a PWM signal. However, you do not have to worry as there is a library built in the Arduino IDE that is used to control servos.

18.4 Connections

Below, you can find the schematic:



Below, you can find the visual representation:



18.5 Code

In this code you will see how to use an analog pin. In contrast to a digital pin, an analog one is able to read multiple values (not only 0 and 1 as a digital one). It is able to read between 0 and 1024, as there is a 10 bit converter. This value represents a voltage between 0 and 3.3 V, as the board is operating at 3.3 V. The development board only has one analog pin and it is marked as **A0**. You can find the code in the folder called **"Lesson 12: Potentiometer and Servo Motor"** .

The code starts by including the library used for the Servo Motor, which is **Servo.h**, then we will declare an object of this class named **servo**.

Code 18.5.1 The library used to control the Servo Motor

```
//the library "Servo.h" is used to control a servo motor using
//PWM technique
#include <Servo.h>

//declare a new object called servo
Servo servo;
```

In the **setup()** function we will start a serial communication with the computer and then use the instruction **servo.attach()** to attach the servo to digital pin **D1**.

Code 18.5.2 The setup() function

```
void setup()
{
  //start the serial communication with the computer at 115200 bits/s
  Serial.begin(115200);

  //attach the servo on digital pin D1
  servo.attach(D1);
}
```

In the **loop()** function we will read the value from the analog pin using **analogRead()**, which is the position of the potentiometer, then remap this value, which is between **0** and **1024**, to the **0-180** interval so that we can get the angle that the potentiometer is currently at. A new function used is the **map** function, which is used to remap a number from one range to another.

This method requires 5 parameters: map(value, fromLOW, fromHIGH, toLOW, toHIGH) :

- value -> the number to map
- fromLOW -> the lower bound of the value's current range
- fromHIGH -> the upper bound of the value's current range
- toLOW -> the lower bound of the value's target range
- toHIGH -> the upper bound of the value's target range

The next instruction is to write the value to the servo using **servo.write(value)**. Now, we can also print in Serial Monitor the angle of the potentiometer (which is also the angle of the Servo Motor).

Code 18.5.3 The loop() function

```
void loop()
{
  //read the value on pin A0
  //the pin is able to read a value between 0 and 1024 corresponding
  //to 0 V and 3.3 V
  int value = analogRead(A0);

  //remap the analog value to a new range (from 0 to 180) as the
  //servo can turn max 180 degrees.
  value = map(value, 0, 1024, 0, 180);

  //turn the servo motor accordingly to the angle stored in value
  servo.write(value);

  //print in Serial Monitor the current angle of the servo
  Serial.print("Angle: ");
  Serial.println(value);

  //pause the code for 50ms;
  delay(50);
}
```

19. Lesson 15: Wireless Connectivity

19.1 Overview

In this lesson you will learn how to connect the development board to a network and how to host a simple WEB page.

19.2 Components required

- Development board;
- Micro USB – Type A USB cable;

19.3 HTTP

HTTP (Hypertext Transfer Protocol) is an application protocol for encoding and transporting data between a client and a web server. It is primarily used on the World Wide Web. An HTTP client (usually a web browser) makes a request and the server issues a response that includes not only the requested content, but also status information about the request.

A basic HTTP request involves the following steps:

- a connection to the HTTP server is opened
- a request is sent to the server
- server is processing the request
- the server sends back a response
- the connection is closed

There are three main HTTP types of requests: **GET**, **POST**, **HEAD**.

GET:

The **GET** method requests a representation of the specified resource, and the requests using **GET** should only retrieve data.

HEAD:

The **HEAD** method asks for a response identical to that of a **GET** request, but without the response body.

POST:

The **POST** method requests that the server accepts the entity enclosed in the request as a new subordinate of the web resource identified by the URI.

19.4 Code

The main advantage of this development board is that it can connect to a WiFi network and host a small website. The code for this lesson can be found in the folder **"Lesson 15: Wireless Connectivity"**.

In this lesson, you will connect the board to a WiFi network. You have to know the name of the network (**SSID** = **S**ervice **S**et **Id**entifier) and its password.

The code starts by including the library **ESP8266WebServer.h**. This library also has included the **ESP8266WiFi.h** library necessary for the WiFi part of the code. The next step is to declare two **const char*** variables, **ssid** and **password**, used for storing the credentials of the network.

Next, you have to set up the server side. First, you have to create a webserver object that listens for HTTP requests on a specified port. The default HTTP port is 80 but you can change it however you want. Make sure it is not used by another service.

If you use a port different than 80, you need to be sure you access it correctly from the browser. For example, if the **IP** is **100.32.12.39** and the port is **2093**, you have to access the following address: "**100.32.12.39:2093**". In case you changed the port, it is recommended to use one greater than 1023.

Here are some ports reserved by other services:

1 – TCP Port Service Multiplexer (TCPMUX)	118 – SQL Services
5 – Remote Job Entry (RJE)	119 – Newsgroup (NNTP)
7 - ECHO	137 – NetBIOS Name Service
18 – Message Send Protocol (MSP)	139 – NetBIOS Datagram Service
20 – FTP - Data	143 – Interim Mail Access Protocol (IMAP)
21 – FTP - Control	150 – NetBIOS Session Service
22 – SSH Remote Login Protocol	156 – SQL Server
23 - Telnet	161 - SNMP
25 – Simple Mail Transfer Protocol (SMTP)	179 – Border Gateway Protocol (BGP)
29 – MSG ICP	190 – Gateway Access Control Protocol (GACP)
37 - Time	194 – Internet Relay Chat (IRC)
42 – Host Name Server (Nameserv)	197 – Directory Location Service (DLS)
43 - WhoIs	389 – Lightweight Directory Access Protocol (LDAP)
49 – Login Host Protocol	396 – Novell Netware over IP
53 – Domain Name System (DNS)	443 - HTTPS
69 – Trivial File Transfer Protocol (TFTP)	444 – Simple Network Paging Protocol (SNPP)
70 – Gopher Services	445 – Microsoft-DS
79 - Finger	458 – Apple QuickTime

80 - HTTP	546 – DHCP Client
103 – X.400 Standard	547 – DHCP Server
108 – SNA Gateway Access Server	563 - SNEWS
109 - POP2	569 - MSN
110 - POP3	1080 - Socks
115 – Simple File Transfer Protocol (SFTP)	

Code 19.4.1 Library and declaration

```
#include <ESP8266WebServer.h>

//here you have to insert your wireless network name and password
const char* ssid = "ssid";
const char* password = "password";

//create a new object ESP8266WebServer
//the parameter "80" represents the port that the board listens to.
//if you use a port different than 80, you need to be sure you access it
//correctly from the browser.
//For example:
//if the ip is 100.32.12.39 and the port is 2093, you have to access
//the following address: "100.32.12.39:2093"
ESP8266WebServer server(80);
```

Next, we will have a function that handles the server side. Firstly, we have to organize the content of the website. This can be done using method **server.on(location, content);**.

- The default **location** can be "/", but you can change it to whatever you want.
- The **content** parameter is a function called when you access the specific **location**, declared earlier.

Then, we need to start the server and listening for HTTP requests using **server.begin()**. Also we will print a message in Serial Monitor when the server has started.

Code 19.4.2 The function for set up the server

```
void setupServer()
{
  //the method "server.on()" is to call a function when
  //the user access the location
  //the default location is "/"
  server.on("/", htmlIndex);

  //start the server
  server.begin();

  //print in serial manager that the HTTP server is started
  Serial.println("HTTP server started");
}
```

Next step is to create the functions used as **content** parameters. Inside the new functions, you have to make sure that you use the method server.**send**(). Method server.**send**() requires three parameters in order to work:

- **replyCode** refers to the HTTP code sent together with the message. However, the user will not be able to see this HTTP code.
- **contentType** is a String variable that defines the type of the message sent back to the user
- **message** parameter is the actual text sent to the user. Depending on the content of the message, it might contain HTML tags or not. You will learn how to display a HTML formatted page in a later lesson.

Here are some common codes:

HTTP code	**Meaning**
200	OK
400	Bad Request
403	Forbidden
404	Not Found
500	Internal Server Error

Here are some content types:

Content Type	Meaning
text/plain	The message is not interpreted and formatted by the browser. It is used just to display a text.
text/html	It is pretty similar to "text/plain" but you can format the text using HTML.

Code 19.4.3 The content parameter

```
void htmlIndex()
{
  //You can find on the internet a list with all HTML codes but here
  //are the most used ones:
  //200 -> OK
  //400 -> Bad Request
  //403 -> Forbidden
  //404 -> Not found
  //500 -> Internal Server Error
  int replyCode = 200;

  //contentType refers to the type of the mssage sent to the user
  //The most used ones are:
  //text/plain -> used just for text, without using HTML
  //text/html -> used to create a page using HTML

  String contentType = "text/plain";

  //the mssage variable is used to store the message sent to the
  //user
  String message = "Hello world!";

  //send the message to the user
  server.send(replyCode, contentType, message);
}
```

We also need a function for connecting to a wireless network. In the **connectToWiFi()** function we will start by displaying a message in Serial Monitor. Then we are going to set the WiFi mode as **WiFi_STA**, meaning that the board will act as a station, and connect to the WiFi network using the variables declared at the beginning of the code and the command **WiFi.begin(ssid, password)**. Using a **while** loop we are going to wait for the connection to be made. In the end of the function we will display in Serial Monitor a message that the board is connected to the specifiend network and also we will display the IP address of the board. This IP will be used to connect to the server from a browser.

Code 19.4.4 Connecting to a wireless network

```
void connectToWiFi()
{
  Serial.println("Connecting to the WiFi");
  //set the WiFi mode to WiFi_STA.
  //the WiFi_STA means that the board will act as a station,
  //similar to a smartphone or laptop
  WiFi.mode(WIFI_STA);

  //connect to the WiFi network using the ssid and password strings
  WiFi.begin(ssid, password);

  //below we check the status of the WiFi and wait until the
  //board is connected to the network
  Serial.println("Waiting for connection");
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }

  //when the board is successfully connected to the network,
  //display the IP assigned to it in the Serial Monitor.
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
}
```

In the **setup()** function, we will start a serial communication with the computer and call the two functions created for connecting to WiFi and setup the server: **connectToWiFi()** and **setupServer()**.

Code 19.4.5 The setup() function

```
void setup()
{
  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  //wait 1 s for the Serial communication to start
  delay(1000);

  //call the two functions used to connect connect to wifi
  //and setup the server
  connectToWiFi();
  setupServer();
}
```

Finally we have the **loop()** function, which consists of a single instruction used to check if there is somebody trying to access the website.

Code 19.4.6 The loop() function

```
void loop()
{
  //the method below is used to manage the incoming request
  //from the user
  server.handleClient();
}
```

20. Theory lesson: Web pages

20.1 HTML

HTML (**H**yper **T**ext **M**arkup **L**anguage) is the standard markup language for creating Web pages and Web applications. A markup language is a computer language that uses tags to define elements inside a text file.

In HTML, as in many others markup languages, the tags have the next structure:

```
<tag>
```

An HTML element usually consists of a **start** tag and **end** tag, and the content inserted between them:

```
<tag> Text </tag>
```

To explain the standard structure of a HTML page, let's do the standard **Hello world!** page:

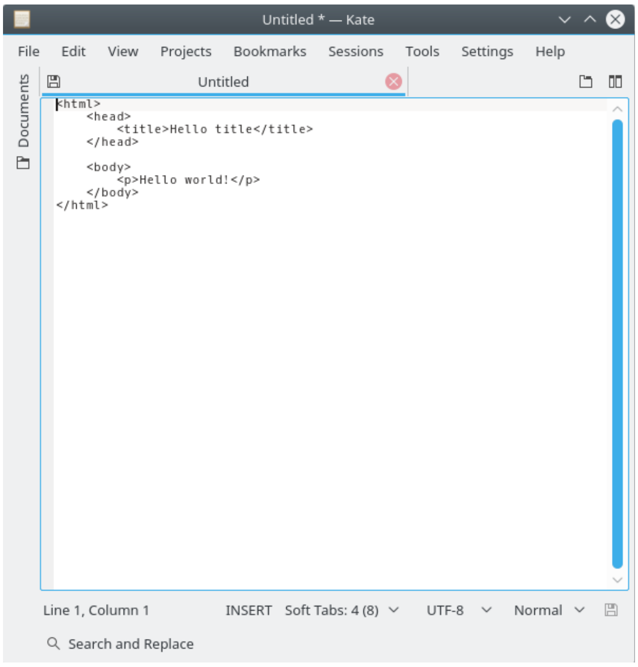
Code 20.1.1 First HTML page

```
<html>
    <head>
        <title>Hello title</title>
    </head>

    <body>
        <p>Hello world!</p>
    </body>
</html>
```

Elements explanation:

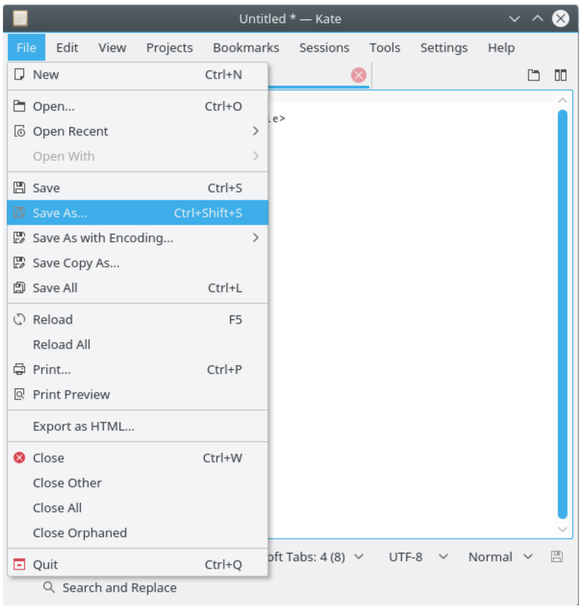
- <**html**> is the root element of an HTML page
- <**head**> contains meta information about the document
- <**title**> element specifies a title for the document
- <**body**> element contains the visible page content
- <**p**> defines a paragraph

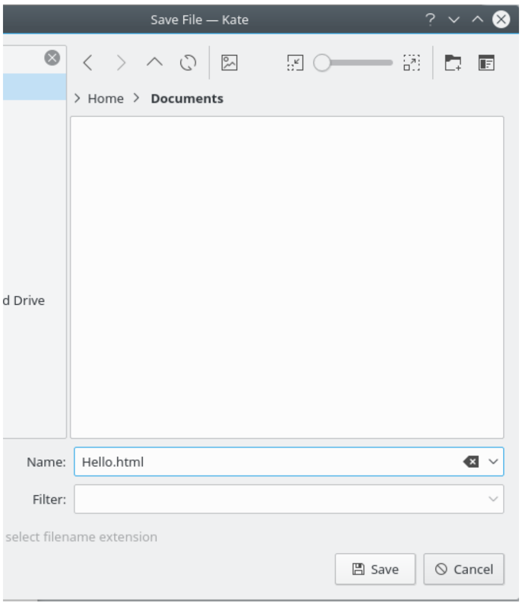
Now, open a text editor and insert the text above.


Untitled * — Kate
File Edit View Projects Bookmarks Sessions Tools Settings Help
Documents
Untitled

```
<html>
    <head>
        <title>Hello title</title>
    </head>

    <body>
        <p>Hello world!</p>
    </body>
</html>
```

Save the file to a preferred location with the extension **.html**.

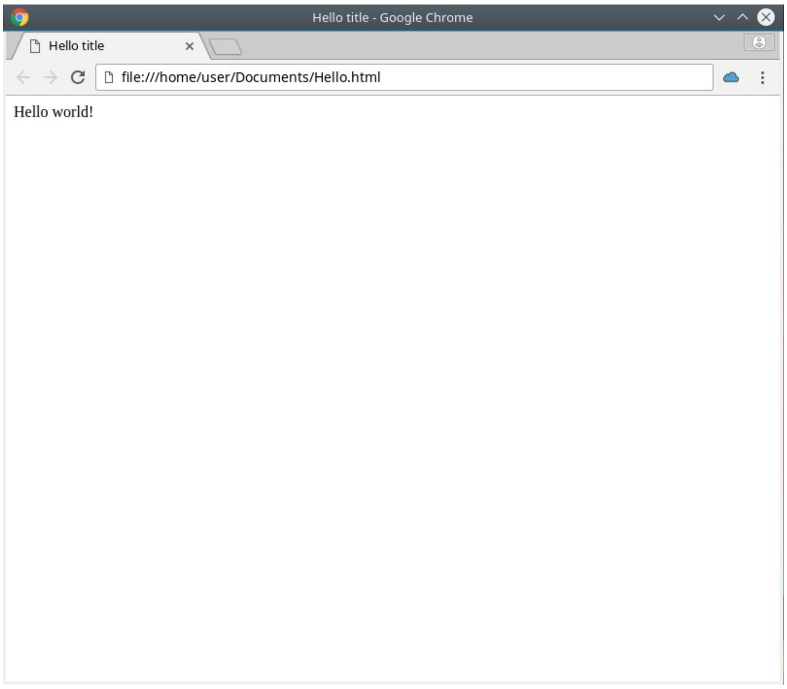




Name: Hello.html

Filter:

You can open the file with any browser, and you should see something similar with:


Hello title - Google Chrome
Hello title
file:///home/user/Documents/Hello.html

Hello world!

Congratulations! You have just created a HTML file. Also, you can find this code in the folder **HTML > Hello.html**.

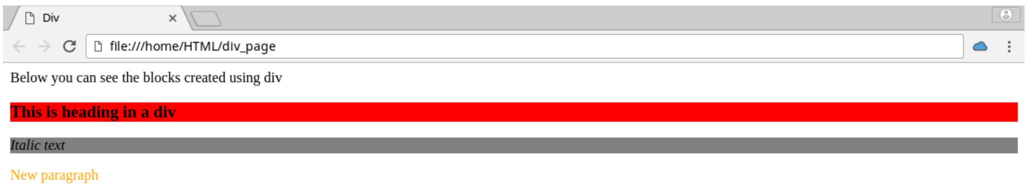
To define a section or a division in a HTML file, we use **<div>** tag. Also **<div>** element is often used as a container for other HTML elements to style them with CSS or to perform certain tasks with JavaScript. Let's create a HTML file with the **<div>** tag.

Styles in HTML describe how a document will be presented on a browser. There are 3 ways of implementing style in HTML: **Inline Style**, **Embedded Style** and **External Style Sheet**. In the **Inline Style** method, the **style** attribute is used inside the HTML start tag and has the following syntax: <tagname style="property:value;">. The **property** is a CSS property. The **value** is a CSS value.

Code 20.1.2 HTML page using <div> tag and styling

```
<html>
    <head>
            <title>Div</title>

    </head>
    <body>
            <p>Below you can see the blocks created using div</p>

            <div style="background-color:red">
                    <h3>This is heading in a div</h3>
            </div>
            <div style="background-color:gray">
                    <i>Italic text</i>
            </div>
            <div style="color:orange">
                    <p>New paragraph</p>
            </div>
    </body>
</html>
```

Open the file with any browser and you should see something like this:



Below you can see the blocks created using div

This is heading in a div

Italic text

New paragraph

CSS has several different units for expressing a length. The length is composed of a number followed by a length unit, such as 20px, 3em, etc. There are two types of length units: absolute and relative.

The absolute length units are fixed and a length expressed in any of these will appear at exactly that size:

cm	centimeters
mm	milimeters
in	inches (1in = 96px = 2.54 cm)
px	pixels (1px = 1/96th of 1in)
pt	points (1pt = 1/72 of 1in)
pc	picas (1pc = 12pt)

Relative length units specify a length relative to another length property.

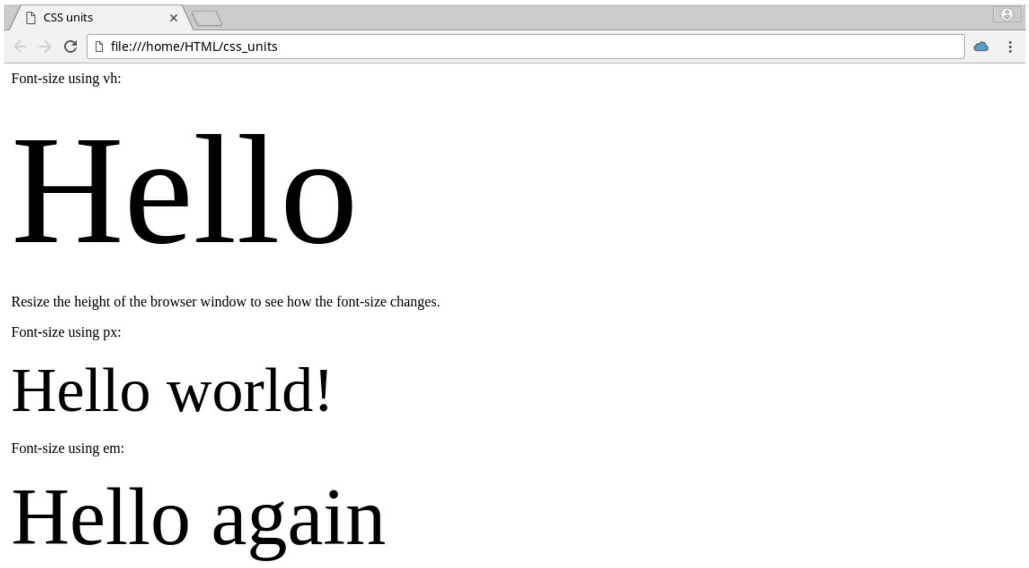
em	Relative to the font-size of the element (2em means 2 times the size of the current font)
ex	Relative to the x-height of the current font (rarely used)
ch	Relative to width of the "0" (zero)
rem	Relative to font-size of the root element
vw	Relative to 1% of the width of the viewport
vh	Relative to 1% of the height of the viewport
vmin	Relative to 1% of viewport's* smaller dimension
vmax	Relative to 1% of viewport's* larger dimension
%	Relative to the parent element

Below, you can find an example of a HTML page in which we used both, absolute length and relative length. Resize the browser to see the effects:

Code 20.1.3 CSS units

```
<html>
    <head>
        <title> CSS units</title>
    </head>
    <body>
        <p>Font-size using vh:</p>
        <div style="font-size: 20vh">Hello</div>
          <p>Resize the height of the browser window to see how the font-size
changes.</p>

        <p>Font-size using px:</p>
        <div style="font-size: 70px">Hello world!</div>

        <p>Font-size using em:</p>
        <div style="font-size: 30px">
            <div style="font-size: 3em">Hello again</div>
        </div>
    </body>
</html>
```

Open the file with any browser and you should see something like this:



Font-size using vh:

Hello

Resize the height of the browser window to see how the font-size changes.

Font-size using px:

Hello world!

Font-size using em:

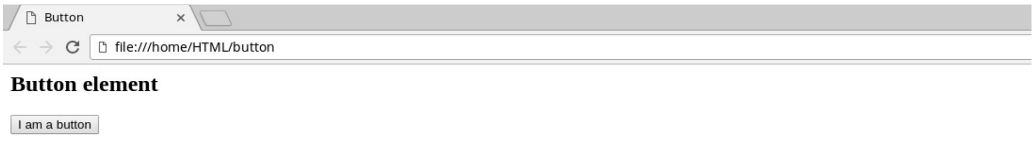
Hello again

The **<button>** tag defines a clickable button. Inside a button you can put text or images. Make sure to specify the type attribute for a **<button>** element, because different browsers use different default types for the **<button>** element. Below you can find an example:

Code 20.1.4 HTML <button> tag

```
<html>
    <head>
            <title>Button</title>
    </head>

    <body>
            <h2>Button element</h2>
            <button type="button">I am a button</button>
    </body>

</html>
```

Open the file with any browser and you should see something like this:



Button element

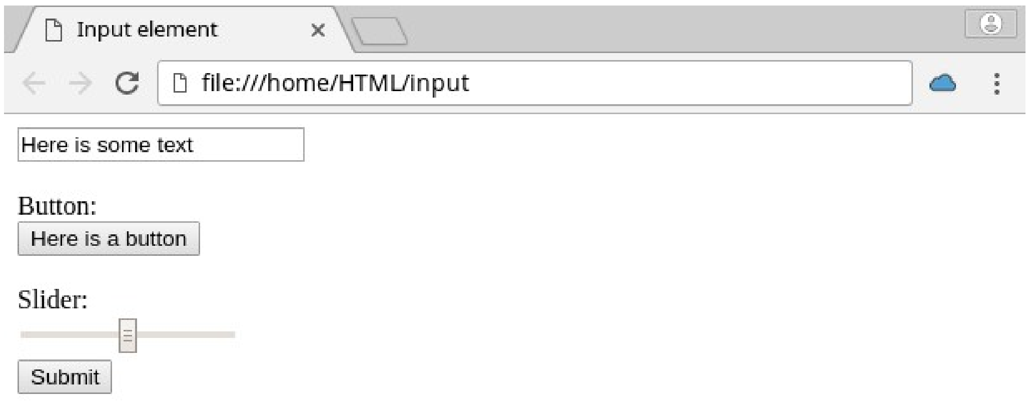
Next, we will discuss about <**input**> element. Firstly, <**form**> element represents a document that contains interactive controls for submitting informations to a web server. The HTML <**input**> element is used to create interactive controls for web-based forms in order to accept data from the user. An input field can vary in many ways, depending on the type attribute. There are many types, for example: button, checkbox, radio, range, submit, text, time etc.

In our project, we will use the range type to define a slider. Also we can define a button using <**input**> element. In the following example we will use **button** and **range** types.

Code 20.1.5 The <input> element

```
<html>
    <head>
            <title>Input element</title>
    </head>
    <body>
        <input type="text" value="Here is some text">
        <br/>
        <br/>
        Button:
        <br/>
        <input type="button" value="Here is a button">
        <br/>
        <br/>
        Slider:
        <br/>
        <input type="range" name="position" min="0" max="100">
        <br/>
        <input type="submit">
    </body>
</html>
```

And the page should look like this:

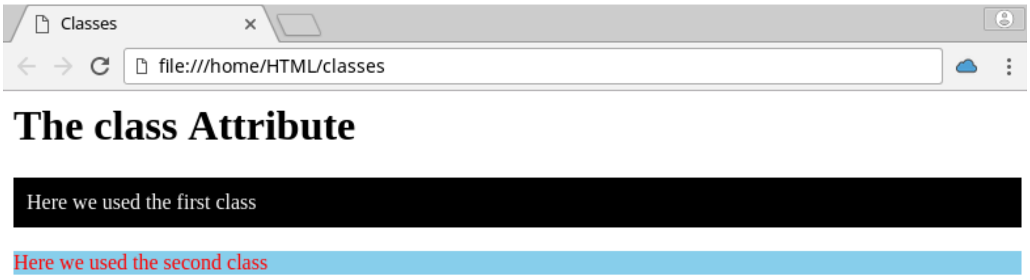


The **class** attribute specifies one or more classes for an element. **class** in HTML is reference to the CSS, but it can also be used by JavaScript. Syntax: <element class="name">. Note that the class name is case sensitive! You can find below an example in which we used classes.

Code 20.1.6 CSS classes

```
<html>
    <head>
        <style>
            .class1 {
                background-color: black;
                color: white;
                padding: 10px;
            }
            .class2 {
                background-color: skyblue;
                color: red;
            }
        </style>
        <title>Classes</title>
    </head>
    <body>
        <h1>The class Attribute</h1>

        <div class="class1">Here we used the first class</div>
        <br/>
        <div class="class2">Here we used the second class</div>

    </body>
</html>
```

And the page should look like this:


Classes

file:///home/HTML/classes

The class Attribute

Here we used the first class

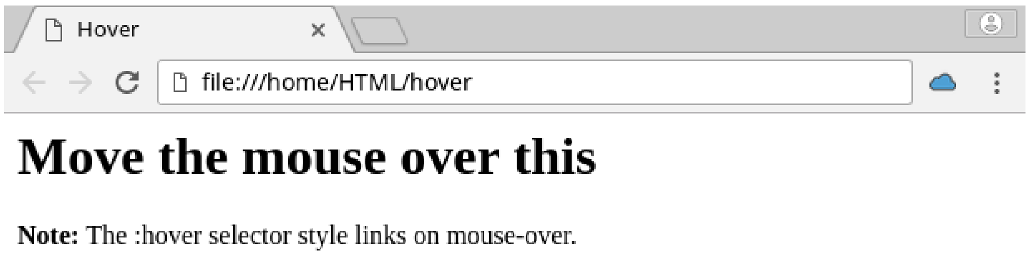
Here we used the second class

In CSS, selectors are patterns used to select the element(s) you want to style. Examples of CSS selectors: **.class** (used above), **#id**, **:hover,** etc. The **:hover** selector is used to select elements when you move the mouse over them. Below, you can find an example with **:hover**.

Code 20.1.7 CSS selectors

```
<html>
    <head>
        <title>Hover</title>
        <style>
            h1:hover {
                color: skyblue;
            }
        </style>
    </head>
    <body>
        <h1>Move the mouse over this</h1>

        <p><b>Note:</b> The :hover selector style links on mouse-over.</p>

    </body>
</html>
```

And the page should look like this:



Move the mouse over this

Note: The :hover selector style links on mouse-over.

Further information can be found by accessing the links attached:

https://www.w3schools.com/html/default.asp

https://developer.mozilla.org/en-US/docs/Learn/HTML

https://www.w3schools.com/css/default.asp

All the HTML files can be found in the folder called **HTML**.

20.2 JavaScript Object Notation

Java**S**cript **O**bject **N**otation **(JSON)** is a syntax for storing and exchanging data. While exchanging information between server and a web page, the data can only be **text.**

Below, there is an example how a JSON looks like:

```
{
    "id": 10 ,
    "company":"Plusivo"
}
```

This is an example of a JSON object. JSON objects are surrounded by curly braces {} and are written in **key/value** pairs. **key** must be strings and **value** must be a data type (string, number, object, array, boolean or null).

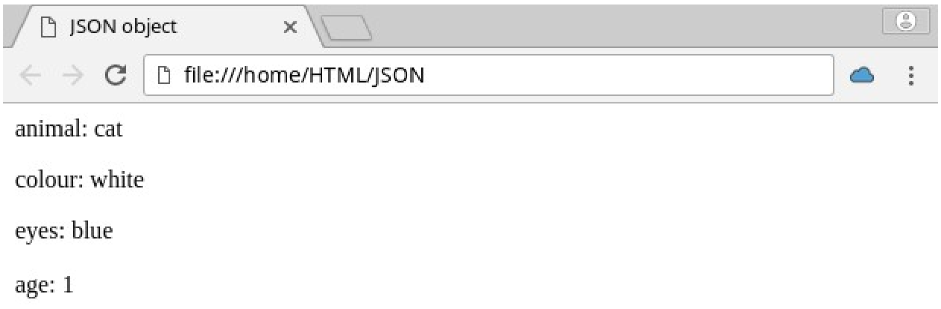
In the example above, there are 2 fields (**keys**) called **"id"** and **"company"**. These two fields are separated by a comma. **"id"** stores an integer and **"company"** stores a string. Please note that an integer doesn't need to be in quotes, whereas a string has to be in quotes.

Let's make an example with JSON objects:

Code 20.2.1 JSON objects

```
<html>
    <head>
        <title>JSON object</title>
    </head>
    <body>
        <p id="type"></p>
        <p id="colour"></p>

        <script>

            var object =
            {
                "animal": "cat",
                "colour": "white",
                "eyes": "blue",
                "age": 1
            };

            document.getElementById("type").innerHTML = "animal: " + object.animal;

            document.getElementById("colour").innerHTML = "colour: " + object.colour;

            document.writeln("eyes: " + object.eyes + "<br/>" + "<br/>");

            document.writeln("age: " + object.age);

        </script>
    </body>
</html>
```

And the page should look like this:



```
animal: cat

colour: white

eyes: blue

age: 1
```

innerHTML is used to change the content of the page without refreshing. Along **getElementById()** is used in the JavaScript code to refer to an HTML element and change its contents.

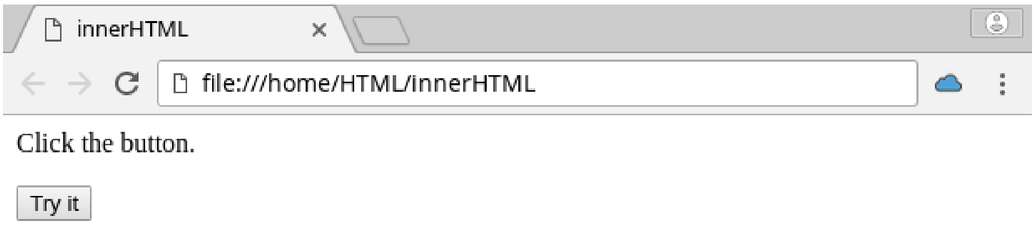
document.writeln() writes a string of text to a document.

Below you can find another example using **document.getElementByID().innerHTML** in which we are using a button, and when the button is pressed, the text will change.

Code 20.2.2 Example with innerHTML

```
<html>
    <body>

        <p id="change">Click the button.</p>

        <button onclick="magic_function()">Try it</button>

        <script>
            function magic_function() {
                document.getElementById("change").innerHTML = "Magic";
            }
        </script>

    </body>
</html>
```

And the page should look like this:



Click the button.

You can find more details about **JSON** by accessing the links below:

https://www.w3schools.com/js/js_json_intro.asp

https://beginnersbook.com/2015/04/json-tutorial/

https://www.json.org/

https://www.tutorialspoint.com/json/

20.3 jQuery

jQuery is a free, open-source software using the MIT License, fast, easy to learn and to use, JavaScript library that can be used to simplify animation, HTML document traversing, event handling and Ajax.

There are several ways to start using jQuery on your web site. We will use it by including jQuery from Google. So, in the HTML file, in head, we have to include the script:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
```

The basic syntax of jQuery is: **$(selector).action()**.

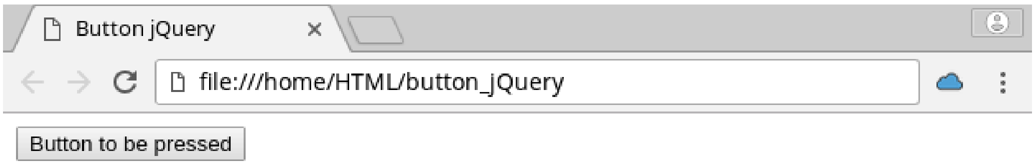
jQuery selectors allow you to select and manipulate HTML elements. The **#id** selector uses the **id** attribute of an HTML tag to find the specific element and this **id** should be unique for each element.

The interaction of the user with the web page is called **event**. These events are actions of a user and there are many interactions with the web page, for example: clicking on elements, move the mouse over elements, typing in textboxes, etc. To tell the browser what to do when an event triggers, we provide a fuction, known as **event handler**. Let's make a simple example to display a message when clicking a button. Note that this is also a HTML page.

Code 20.3.1 jQuery selectors

```
<html>
    <head>
        <script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
        <title>Button jQuery</title>
        <script>

            $(document).ready(function(){
                $("#my_button").click(function(){
                    alert("The button was pressed");
                });
            });

        </script>
    </head>
    <body>
        <button id="my_button" type="button">Button to be pressed</button>
    </body>
</html>
```

And the page should look like this:



We have to use the **.on(events [, selector] [, data], handler)** method in order to bind an event handler function to one or more events, like **mousedown**,

touchend, **change** etc, and attach the handler to the selected elements.

Below is the description of all the parameters used by this method:

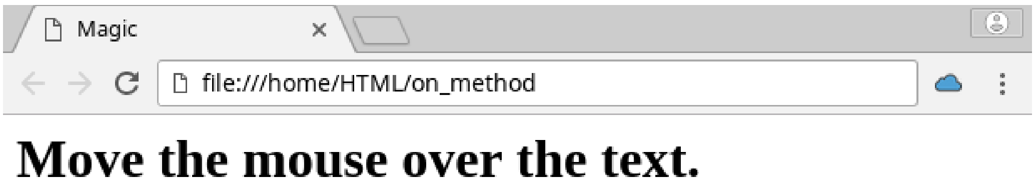
- **events** – events types separated by spaces
- **selector** – a selector string
- **data** – data to be passed to the handler in **event.data**
- **handler** - a function to execute when the event is triggered, and also is required

Let's do an example of this method. When you move the mouse over the text, the colour of the text will stay blue while the mouse stays over the text.

Code 20.3.2 .on() method

```
<html>
    <head>
        <title>Magic</title>
        <script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
        <script>

            $(document).ready(function(){
                   $("h1").on("mouseover mouseout", function(){
                        $(this).toggleClass("h1-class");
                    });
            });
        </script>
        <style>
            .h1-class {
                color: blue;
            }
        </style>
    </head>
    <body>

            <h1>Move the mouse over the text.</h1>

    </body>
</html>
```

And the page should look like this:



Move the mouse over the text.

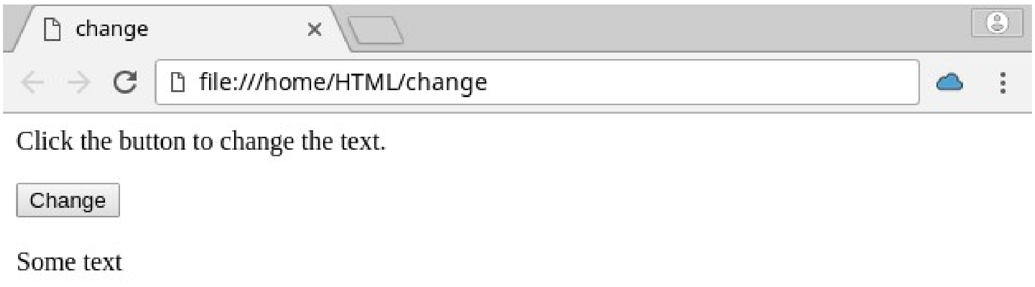
Using **$(selector).html(string)** we can refer to a HTML element and change its contents. It is similar with **document.getElementByID().innerHTML**.

Example:

Code 20.3.3 .html() for changing HTML element content

```
<html>
    <head>
        <script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
        <script>
                $(document).ready(function(){
                    $("button").click(function(){
                        var x = new Date();

                        $("div").html("Date and time: " + x);
                    });
                });
        </script>
    </head>
    <body>
        <p>Click the button to change the text.</p>
        <button>Change</button>
        <br><br>
        <div>Some text</div>

    </body>
</html>
```

And the page should look like this:



Click the button to change the text.

Some text

The **ajax()** method is used for sending asynchronous HTTP requests to the server. Working with AJAX, exchanging data with a server is without reloading the entire page.

There are two most-used methods for loading data from the server: **GET** and **POST**.

GET is used for loading data from the server, and may return cache data. **POST** is used for loading data from the server, but never caches data and is used to send data along with the request.

The syntax of these two methods are similar:

```
jQuery.get( url [, data ] [, success ] [, dataType ] )

jQuery.post( url [, data ] [, success ] [, dataType ] )
```

And their equivalents are:

For GET:

```
$.ajax({
  url: url,
  data: data,
  success: success,
  dataType: dataType
});
```

For POST:

```
$.ajax({
  type: "POST",
  url: url,
  data: data,
  success: success,
  dataType: dataType
});
```

Parameters:

- **url** is a string containing the URL to which the request is sent
- **data** is a string that is sent to the server along the request
- **success** is a function that is executed if the request succeeds
- **dataType** is the data type expected from the server

For further details and examples with jQuery, Ajax and the two methods, you can access the links below:

https://jquery.com/

https://learn.jquery.com/

https://www.w3schools.com/jquery/default.asp

https://www.w3schools.com/jquery/jquery_ajax_get_post.asp

20.4 Bootstrap

Bootstrap is a free and open-source front-end framework for easy web development. It is the most popular front-end library and works with HTML, CSS and JavaScript.

You can download and host Bootstrap yourself, or you can include it from a CDN, for example:

```
<link rel="stylesheet" 
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap
.min.css"/>
```

Bootstrap is designed to work with touch devices, so to ensure proper rendering and touch zooming, add the following **<meta>** tag:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Bootstrap also requires a containing element to wrap site contents, and there are two container classes:

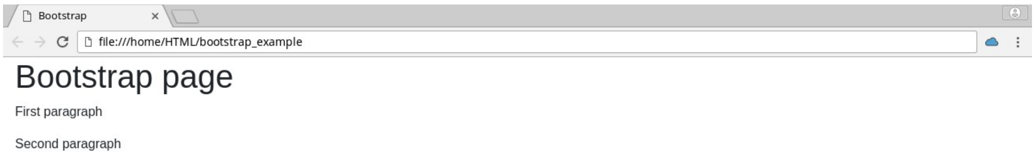
- **.container** class provides a responsive fixed width container
- **.container-fluid** provides a full width container

Below you can find a simple example with a HTML web page using Bootstrap.

Code 20.4.1 Bootstrap, first example

```
<html lang="en">
    <head>
        <title>Bootstrap</title>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css"/>
    </head>

    <body>

        <div class="container-gluid">
            <h1>Bootstrap page</h1>
            <p>First paragraph</p>
            <p>Second paragraph</p>
        </div>

    </body>
</html>
```

And the page should look like this:



Bootstrap page

First paragraph

Second paragraph

Bootstrap's grid system is built with flexbox and allows up to 12 columns across the page. The Bootstrap grid system has five classes:

- **.col-** (extra small devices)
- **.col-sm-** (small devices)
- **.col-md-** (medium devices)
- **.col-lg-** (large devices)
- **.col-xl-** (extra large devices)

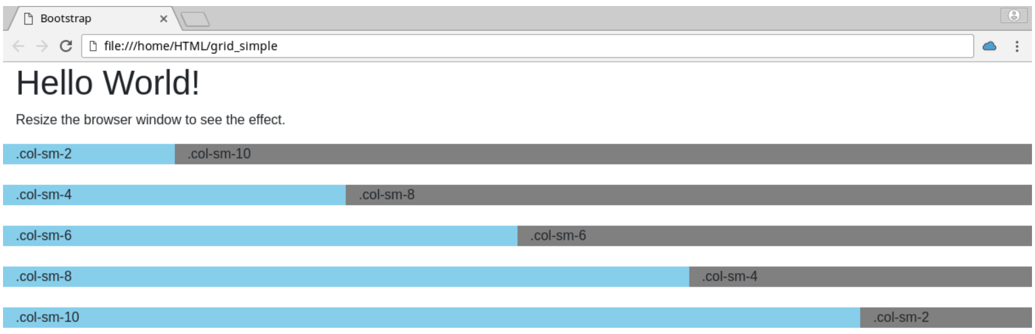
You can also combine the classes for good compatibility with all types of devices.

Below you can see an example with one of the classes:

Code 20.4.2 Bootstrap grid example

```
<html lang="en">
    <head>
        <title>Bootstrap</title>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css"/>
    </head>
    <body>

    <div class="container-fluid">
        <h1>Hello World!</h1>
        <p>Resize the browser window to see the effect.</p>
        <div class="row">
            <div class="col-sm-2" style="background-color:skyblue;">.col-sm-2</div>
            <div class="col-sm-10" style="background-color:gray;">.col-sm-10</div>
        </div>
        <br/>
        <div class="row">
            <div class="col-sm-4" style="background-color:skyblue;">.col-sm-4</div>
            <div class="col-sm-8" style="background-color:gray;">.col-sm-8</div>
        </div>
        <br/>
        <div class="row">
            <div class="col-sm-6" style="background-color:skyblue;">.col-sm-6</div>
            <div class="col-sm-6" style="background-color:gray;">.col-sm-6</div>
        </div>
        <br/>
        <div class="row">
            <div class="col-sm-8" style="background-color:skyblue;">.col-sm-8</div>
            <div class="col-sm-4" style="background-color:gray;">.col-sm-4</div>
        </div>
        <br/>
        <div class="row">
            <div class="col-sm-10" style="background-color:skyblue;">.col-sm-10</div>
            <div class="col-sm-2" style="background-color:gray;">.col-sm-2</div>
        </div>

    </div>
    </body>
</html>
```

And the page should look like this:



Hello World!

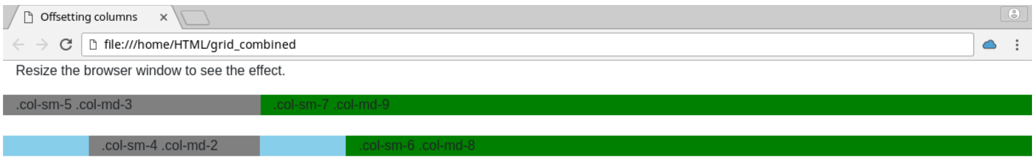
Resize the browser window to see the effect.

You can move columns to the right using **offset-md-*** classes. These classes increase the left margin of a column by * columns:

Code 20.4.3 Offseting columns

```
<html lang="en">
    <head>
        <title>Offsetting columns</title>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css"/>
    </head>
    <body>
    <div class="container-fluid">
        <p>Resize the browser window to see the effect.</p>
        <div class="row" style="background-color:skyblue;">
              <div class="col-sm-5 col-md-3" style="background-color:gray">.col-sm-5 .col-
md-3</div>
             <div class="col-sm-7 col-md-9" style="background-color:green">.col-sm-7 .col-
md-9</div>
        </div>
        <br/>
        <div class="row" style="background-color:skyblue;">
                <div class="col-sm-4 offset-sm-1 col-md-2 offset-md-1" style="background-
color:gray">.col-sm-4 .col-md-2</div>
                <div class="col-sm-6 offset-sm-1 col-md-8 offset-md-1" style="background-
color:green">.col-sm-6 .col-md-8</div>
        </div>
    </body>
</html>
```

And the page should look like this:



Resize the browser window to see the effect.

.col-sm-5 .col-md-3 .col-sm-7 .col-md-9

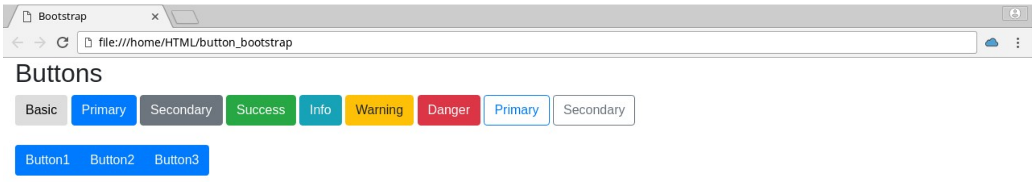
.col-sm-4 .col-md-2 .col-sm-6 .col-md-8

Using Bootstrap, any HTML element looks better. Next, let's create a HTML file using Bootstrap to create different styles of buttons. Also, using the **.btn-group** class, we can create a button group.

Code 20.4.4 Bootstrap buttons

```
<html lang="en">
    <head>
        <title>Bootstrap</title>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css"/>
    </head>
    <body>

    <div class="container-fluid">
        <h2>Buttons</h2>
        <button type="button" class="btn">Basic</button>
        <button type="button" class="btn btn-primary">Primary</button>
        <button type="button" class="btn btn-secondary">Secondary</button>
        <button type="button" class="btn btn-success">Success</button>
        <button type="button" class="btn btn-info">Info</button>
        <button type="button" class="btn btn-warning">Warning</button>
        <button type="button" class="btn btn-danger">Danger</button>
        <button type="button" class="btn btn-outline-primary">Primary</button>
        <button type="button" class="btn btn-outline-secondary">Secondary</button>
        <br/>
        <br/>
        <div class="btn-group">
            <button type="button" class="btn btn-primary">Button1</button>
            <button type="button" class="btn btn-primary">Button2</button>
            <button type="button" class="btn btn-primary">Button3</button>
        </div>
    </div>

    </body>
</html>
```

And the page should look like this:

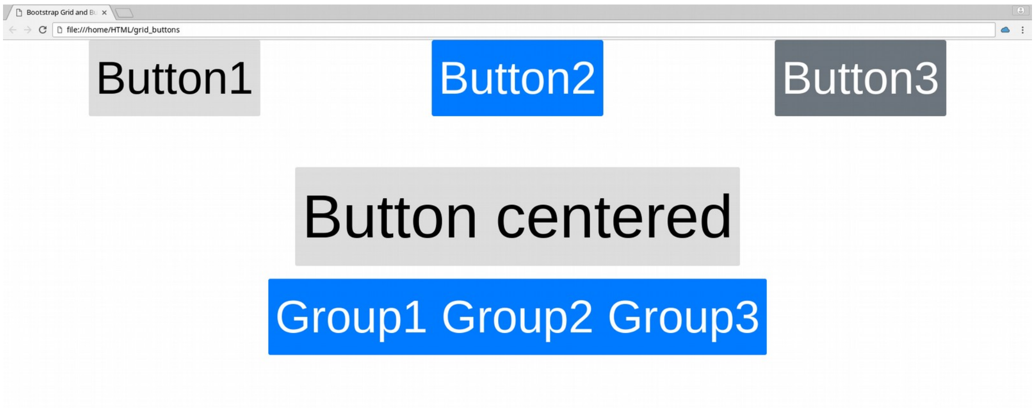


And, also, let's make an example with Bootstrap grid and buttons:

Code 20.4.5 Bootstrap grid and buttons

```
<html>
    <head>
        <title>Bootstrap Grid and Buttons</title>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css"/>
    </head>
    <body>
        <div class="container-fluid">
            <div class="row">
                <div class="col-sm-4 text-center">
                    <button type="button" class="btn" style="font-size: 
9vh">Button1</button>
                </div>
                <div class="col-sm-4 text-center">
                    <button type="button" class="btn btn-primary" style="font-size: 
9vh">Button2</button>
                </div>
                <div class="col-sm-4 text-center">
                    <button type="button" class="btn btn-secondary" style="font-size: 
9vh">Button3</button>
                </div>
            </div>

            <div class="row" style="height: 10vh;"> </div>

            <div class="row">
                <div class="col-sm-12 text-center">
                    <button type="button" class="btn" style="font-size: 12vh">Button 
centered</button>
                </div>
            </div>
            <br/>
            <div class="row">
                <div class="col-sm-12 text-center">
                <div class="btn-group">
                    <button type="button" class="btn btn-primary" style="font-size: 
9vh">Group1</button>
                    <button type="button" class="btn btn-primary" style="font-size: 
9vh">Group2</button>
                    <button type="button" class="btn btn-primary" style="font-size: 
9vh">Group3</button>
                </div>
                </div>
            </div>
        </div>
    </body>
</html>
```

And the page should look like this:



For more information about Bootstrap, click the links below:

https://www.w3schools.com/bootstrap4/default.asp

https://getbootstrap.com/

20.5 Font awesome

Font Awesome is the web's most popular icon set and toolkit. It has many awesome vector icons and social logos.

You can use Font Awesome by including their CDN in the head of your HTML file:

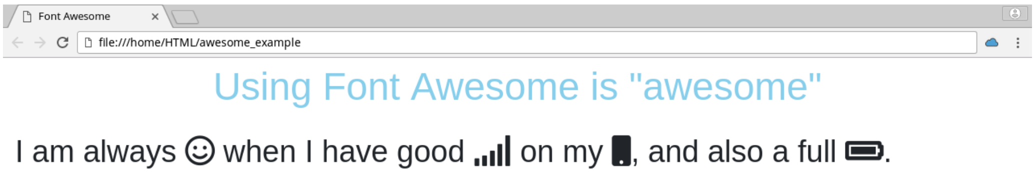
```
<link                                                    rel="stylesheet"
href="https://use.fontawesome.com/releases/v5.1.0/css/all.css"/>
```

Now, let's use a few icons to create a HTML page:

Code 20.5.1 Font awesome, first example

```
<html lang="en">
    <head>
        <title>Font Awesome</title>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet"
href="https://use.fontawesome.com/releases/v5.1.0/css/all.css"/>
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css"/>
    </head>
    <body>
    <div class="container-fluid">
        <div class="row" style="font-size:48px">
            <div class="col-sm-12 text-center" style="color:skyblue">
            <p>Using Font Awesome is "awesome"</p>
            </div>
        </div>
        <div class="row">
        <div  class="col-sm-12" style="font-size:38px">
            I am always
            <i class="far fa-smile" style="color:red"></i>
            when I have good
            <i class="fas fa-signal"></i>
            on my
            <i class="fas fa-mobile"></i>,
            and also a full
            <i class="fas fa-battery-full"></i>.
        </div>
        </div>
    </div>
    </body>
</html>
```

And the page should look like this:



Using Font Awesome is "awesome"

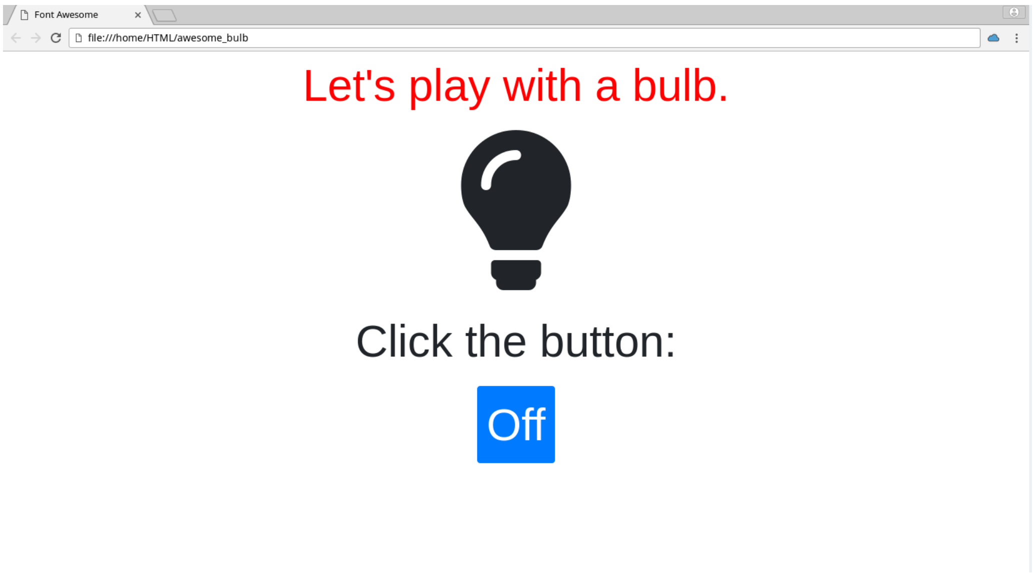
I am always ☺ when I have good 📶 on my 📱, and also a full 🔋.

Let's play with another example, also using notions of jQuery. In this example we will bootstrap, fontawesome and jQuery. Using fontawesome, we will define a bulb icon and apply to that bulb a specific size. Also, we are using a bootstrap button with the initial value set to **Off**. Using JavaScript and jQuery, we will animate the button and icon, previously defined. The **.click(handler)** event is used to bind an event handler to the **click** JavaScript event. When the button is clicked, a function, represented by handler, will execute. In that function we will use a variable **clicked** which value can be **true** or **false**. We defined another two variables, **color** and **button**, that stores the color of the bulb and, respectively, the state, **On** or **Off**. You can see that we used another jQuery method **.css("propertyname", "value")**. This method sets or returns one or more style properties for the selected elements. Using **.val()** method we will modify the value of the button.

Code 20.5.2 Font awesome and jQuery

```
<html lang="en">
    <head>
        <title>Font Awesome</title>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.1.0/css/all.css"/>
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css"/>
        <script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
    </head>
    <body>
    <div class="container-fluid text-center">
        <div class="row text-center">
            <div class="col-sm-12" style="color:red;font-size:7vh">
                <p>Let's play with a bulb.</p>
            </div>
        </div>

        <div class="row">
            <div  class="col-sm-12">
                <i id="id_bulb" class="fas fa-lightbulb" style="font-size:25vh"></i>
            </div>
        </div>
            <br>
        <div class="row">
            <div  class="col-sm-12" style="font-size:7vh">
                <p>Click the button:</p>
            </div>
        </div>

        <div class="row">
            <div  class="col-sm-12">
                <input type="button" class="btn btn-primary" id="id_button" value="off"  style="font-
size:7vh">
            </div>
        </div>
    </div>

    <script>
        clicked = true;
        $(document).ready(function(){
            $("#id_button").click(function(){
                var color = clicked ? 'orange' : 'black';

                var button = clicked ? 'On' : 'Off';

                $("#id_bulb").css('color', color);
                $("#id_button").val(button);
                clicked = !clicked;
            });
        });
    </script>
    </body>
</html>
```

And the page should look like this:



In the example above, we used three variables for storing the last state, color and value. We will make another example that will do the same things, but this time we will use css classes to change the color of the bulb, and, based on the last class used, we will change the color of the bulb and the value of the button. The **.hasClass()** method will return **true** if the class is assigned to the specified element (in our case, the bulb icon).

In the HTML body, where we defined the icon, we have to set the **css_off** class. When the button is clicked, we will check if the class assigned is **css_off** and if that is true, we will set the value to **On** and remove the **css_off** class using **.removeClass()** method and add the new class **css_on**. If the statement is false, then we will set the value to **Off**, remove the **css_on** class and add **css_off** class. The page will look the same, so below is presented only the code.

Code 20.5.3 Awesome bulb with css classes

```
<html lang="en">
    <head>
        <title>Font Awesome</title>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.1.0/css/all.css"/>
        <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css"/>
        <script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
        <style>
            .css_on
            {
                color: orange;
            }
            .css_off
            {
                color: black;
            }
        </style>
    </head>
    <body>
    <div class="container-fluid text-center">
        <div class="row text-center">
            <div class="col-sm-12" style="color:red;font-size:7vh">
                <p>Let's play with a bulb.</p>
            </div>
        </div>

        <div class="row">
            <div  class="col-sm-12">
                <i id="id_bulb" class="fas fa-lightbulb css_off" style="font-size:25vh"></i>
            </div>
        </div>
            <br>
        <div class="row">
            <div  class="col-sm-12" style="font-size:7vh">
                <p>Click the button:</p>
            </div>
        </div>

        <div class="row">
            <div  class="col-sm-12">
                <input type="button" class="btn btn-primary" id="id_button" value="Off"  style="font-
size:7vh">
            </div>
        </div>
    </div>

    <script>
        $(document).ready(function(){
            $("#id_button").click(function(){
                var current_state = $("#id_bulb").hasClass("css_off");

                if(current_state == true)
                {
                    $("#id_button").val("On");
                    $("#id_bulb").removeClass("css_off").addClass("css_on");
                }
                else
                {
                    $("#id_button").val("Off");
                    $("#id_bulb").removeClass("css_on").addClass("css_off");
                }
            });
        });
    </script>
    </body>
</html>
```

You can check their website for many cool icons, and learn how you can integrate all that icons in your code to make your page look astonishing.

```
https://fontawesome.com/
```

21. Lesson 16: Control an LED from web

21.1 Overview

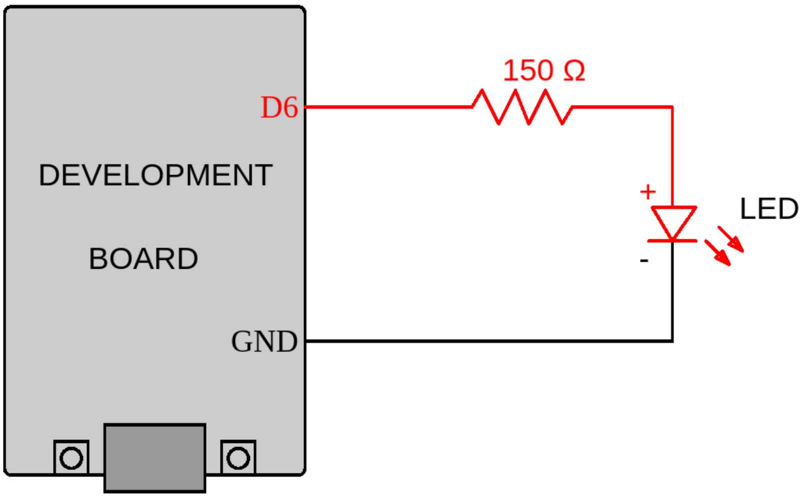
In this lesson you will learn how to turn on or off an LED from a browser. This lesson combines two separate lessons: **Blink an LED** and **Wireless Connectivity**.

21.2 Components required

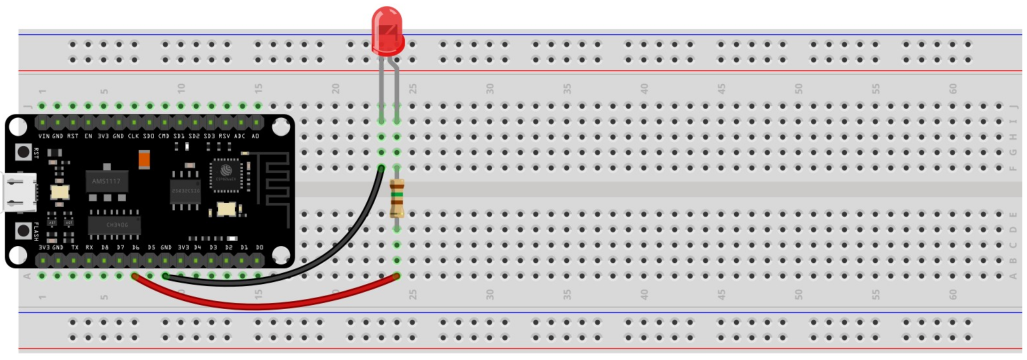
- Development board;
- 1 x LED;
- 1 x 150Ω resistor ;
- Breadboard 830p;
- 2 x male-to-male jumper wire;
- Micro USB – Type A USB Cable;

21.3 Connections

Below is the schematic:



Next, you can see a visual representation of the project:



21.4 Code

The code for this lesson combines multiple lessons: **Blink an LED**, **Wireless Connectivity** and **Theory lesson**, and can be found in the folder **"Lesson 16: Control an LED from web"**.

The HTML page is identical with the one presented at the end of the previous lesson where we used an icon of a bulb and a button, and, when the button was pressed, the value of the button changed, and so the color of the bulb. The **head** and the **body** of the HTML page will remain unchanged, only the JavaScript section is modified.

Before talking about the JavaScript side, let's discuss the server and wireless connectivity side. At the beginning of the code, we will include the **ESP8266WebServer.h** library for the web server, declare two variables, **ssid** and **password**, that will store the credentials for the wireless network, declare an instance of the **ESP8266WebServer** class, declare the pin used by the LED and a string variable that will store later the state of the button (**On** or **Off**). We will use the same **connectToWiFi()** function as in the **Wireless Connectivity** lesson.

The **setupServer()** function contains two **server.on(location, content)** instructions used to tell to the server what URIs it needs to respond to. The first one, **server.on("/", htmlIndex)**, is used to setup the main page and the second one, **server.on("/led", led)**, is used to setup the page for LED.

Code 21.4.1 The setupServer() function

```
void setupServer()
{
  //the method "server.on()" is to call a function when
  //the user access the location
  //the default location is "/"
  server.on("/", htmlIndex);
  server.on("/led", led);

  //start the server
  server.begin();

  //print in Serial Monitor that the HTTP server is started
  Serial.println("HTTP server started");
}
```

The HTML page is stored as a string variable. The **htmlIndex()** function contains only one instruction used to send the page (the string) to the user.

Code 21.4.2 The htmlIndex() function

```
void htmlIndex()
{
  //send the message to the user
  server.send(200, "text/html", page);
}
```

Another function created that will execute when sending a message from the web page, using JavaScript and jQuery, is **led()**. This function is created for controlling the LED. A new method that appears in the code is "server.**arg**(**name**)". It is used to identify an argument called **name** and returns the value (as a String) of the argument.

Example:

```
$.get('/', {value: val});
```

In the above example, the **value** represents the name of the argument and **val** represents the value of the argument **value**.

Our **storage** string variable will contain the current state of the button (**On** or **Off**) and using an **if** statement we will decide if the state is **On**, then we will turn **On** the LED, otherwise the LED will be turned **Off**. At the end of this function we will send an OK message using **server.send(200, "text/html", "ok")**.

Code 21.4.3 The function for controlling the LED

```
void led()
{
  //server.arg(name) returns the value (as string) of the argument "name"
  storage = server.arg("state");

  //if the value returned is "On", we will turn On the LED
  //else, we will turn Off the LED
  if(storage == "On")
  {
    //turn On the LED
    digitalWrite(LED, HIGH);
  }
  else
  {
    //turn Off the LED
    digitalWrite(LED, LOW);
  }

  //send a message to the user
   server.send(200,"text/html","ok");
}
```

Before talking about the rest of the HTML page, let's talk about the **setup()** and **loop()** functions. In the **setup()** function we have to set the pin as **OUTPUT**, then start a serial communication with the computer and call the **connectToWiFi()** and **setupServer()** functions.

Code 21.4.4 The setup() function

```
void setup()
{
  //set the pin used by the LED as OUTPUT
  pinMode(LED, OUTPUT);

  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  //wait 1 s for the Serial communication to start
  delay(1000);

  //call the two functions used to connect connect to the
  //wireless network and setup the server
  connectToWiFi();
  setupServer();

  //wait 4 s for the server to start
  delay(4000);
}
```

In the **loop()** function we need to put one instruction for listening to incoming requests from the user.

Code 21.4.5 The loop() function

```
void loop()
{
  //the method below is used to manage the incoming request
  //from the user
  server.handleClient();
}
```

The JavaScript part will remain partially the same. When the button is clicked, we will check if the class assigned is **css_off** and if that is true, we will set the value to **On** and remove the **css_off** class using **.removeClass()** method and add the new class **css_on**. Also here we will have an **ajax** call for sending a request to the server and send some data to the server, in this case we are sending "On". If the statement is false, then we will set the value to **Off**, remove the **css_on** class, add **css_off** class and make an **ajax** call for sending a request to the server and send some data to the server, in this case we are sending "Off". So, in the HTML we are only adding two ajax calls.

Code 21.4.6 JavaScript

```
<script>
    $(document).ready(function(){
        $('#id_button').click(function(){
            var current_state = $("#id_bulb").hasClass("css_off");

            if(current_state == true)
            {
                $.ajax({
                  url:'/led',
                  type: 'POST',
                  data: {state: 'On'},
                });
                $('#id_button').val('On');
                $('#id_bulb').removeClass('css_off').addClass('css_on');
            }
            else
            {
                $.ajax({
                  url:'/led',
                  type: 'POST',
                  data: {state: 'Off'},
                });
                $('#id_button').val('Off');
                $('#id_bulb').removeClass('css_on').addClass('css_off');
            }
        });
    });
</script>
```

22. Lesson 17: Dim an LED from web

22.1 Overview

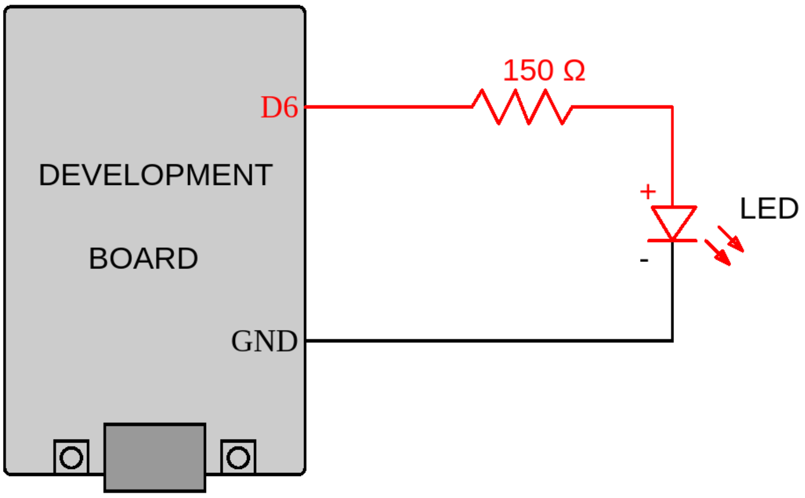
In this lesson you will learn how to control the brightness of an LED from a browser. This lesson is similar with the previous one, but this time we will use a slider for controlling the brightness.

22.2 Components required

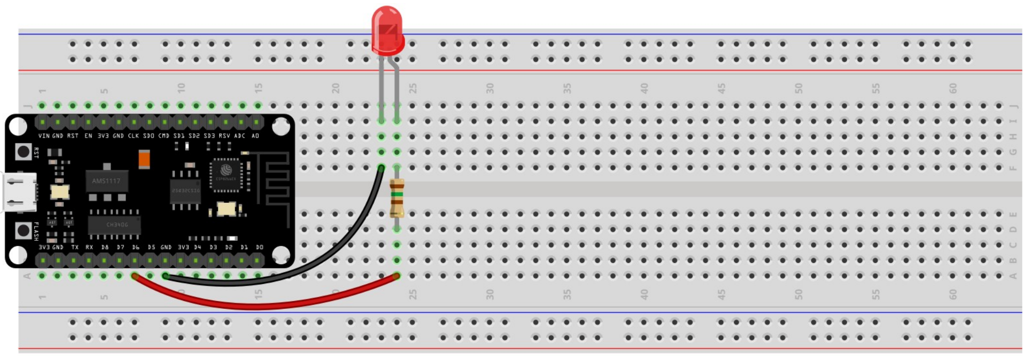
- Development board;
- 1 x LED;
- 1 x 150Ω resistor ;
- Breadboard 830p;
- 2 x male-to-male jumper wire;
- Micro USB – Type A USB Cable;

22.3 Connections

Below is the schematic:



Next, you can see a visual representation of the project:



22.4 Code

The code for this lesson is similar with the one from the previous lesson, but we will use a slider for controlling the brightness of the LED. Please check the **Therory lesson**, where we defined a slider. Initially, the LED will be turned off, and you will need to adjust the slider tot turn it on and control the light intensity. The code for this lesson can be found in the folder **"Lesson 17: Dim an LED from web"**.

The code starts with the inclusion of the **ESP8266WebServer.h** library, then with the declaration of the two variables for storing the credentials of the wireless network that the board is going to connect to, creating an instance of the **ESP8266WebServer** and the declaration of the pin used by the LED. The **connectToWiFi()** function will remain the same as in the previous lesson. The **setupServer()** function contains the instructions to set up the main page and to start the server.

Code 22.4.1 The setupServer() function

```
void setupServer()
{
  //the method "server.on()" is to call a function when
  //the user access the location
  //the default location is "/"
  server.on("/", htmlIndex);

  //start the server
  server.begin();

  //print in Serial Monitor that the HTTP server is started
  Serial.println("HTTP server started");
}
```

The **htmlIndex()** starts by returning a String using **server.arg(String)** and the method **String.toInt()** converts the String variable to int, and, along with the **map** function, we will convert the values of the slider, from 0 to 100, to int values, from 0 to 1023.

Next, using an **if** statement we will check if the value is **0**, then the LED will be set to **LOW**, otherwise using **analogWrite (pin, value)**, we will send a PWM signal that will change the brightness of the LED. At the end of the function, we will send the page to the user.

Code 22.4.2 The function that deals with the HTML page and controlling the LED

```
void htmlIndex()
{
  //server.arg(name) returns the value (as string) of the argument "name"
  //using the method String.toInt(), we convert a String variable to int
  int value = server.arg("state").toInt();

  //remap the value to a new range (from 0 to 100)
  value = map(value, 0, 100, 0, 1023);

  //in case the value is 0, turn off the led using digitalWrite();
  //analogWrite(pin,0) doesn't turn off completely the led
  if(value == 0)
     digitalWrite(LED, LOW); //turn off the LED
  else
     analogWrite(LED, value); //change the brightness of the led

  //send the message to the user
  server.send(200, "text/html", page);
}
```

In the **setup()** function we will set the pin used by the LED as **OUTPUT**, then start a serial communication with the computer and call the two functions used for connecting to WiFi and setup the server, **connectToWiFi()** and **setupServer()**.

Code 22.4.3 The setup() function

```
void setup()
{
  pinMode(LED, OUTPUT);

  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  //wait 1 s for the Serial communication to start
  delay(1000);

  //call the two functions used to connect connect to wifi
  //and setup the server
  connectToWiFi();
  setupServer();
}
```

The **loop()** function listens for any request from the user, using the **server.handleClient()** instruction.

Code 22.4.4 The loop() function

```
void loop()
{
  //the method below is used to manage the incoming request
  //from the user
  server.handleClient();
}
```

Finally, the HTML page. This page is stored as a String and sent to the user in the **htmlIndex()** function. In the head of the HTML page we will include the link for Bootstrap and jQuery. The body of the HTML page is stylized with the Bootstrap grid system and contains a div block that stores a paragraph, another block for the slider and one for telling the user the current brightness.

Code 22.4.5 The HTML body

```
<body>
  <div class='container-fluid text-center'>
      <div class='row'>
          <div class='col-sm-12' style='color:red;font-size:7vh'>
              <p>Use the slider to control the brightness.</p>
          </div>
      </div>
      <br/>
      <div class='col-sm-12'>
              <input type='range' min='0' max='100' class='form-control-range'
id='slider'/>
      </div>
      <div class='row'>
        <div class='col-sm-12'>
          <h3 id='status'></h3>
        </div>
      </div>
  </div>
</body>
```

The JavaScript part is composed from one instruction that reads the current state of the slider and stores it in a variable **state_slider**. Also using this variable and the **.html()** method, we will display the brightness in the browser. Next, is the **ajax** call which will send the **state_slider** value to the server. Then, in the **htmlIndex()** function, this value is converted to **int** and remapped.

Code 22.4.6 JavaScript

```
    <script>
        $('#slider').on('change', function(){
            var state_slider = $('#slider').val();

            $('#status').html('Brightness: ' + state_slider + ' %');
            $.ajax({
              url: '/',
              type: 'POST',
              data: {state: state_slider}
            });
        });
    </script>
```

23. Lesson 18: Dim an RGB LED from web

23.1 Overview

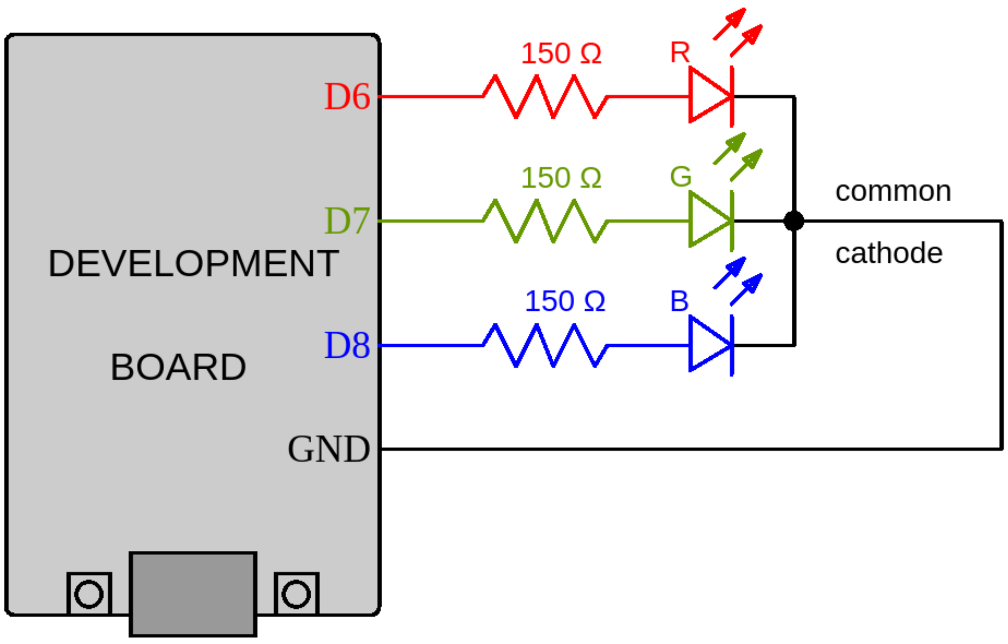
In this lesson you will learn how to control the brightness of each colour of a RGB LED. This lesson is very similar with the previous one, but this time we have a RGB LED, inside which there are 3 LEDs: red, green and blue.

23.2 Components required

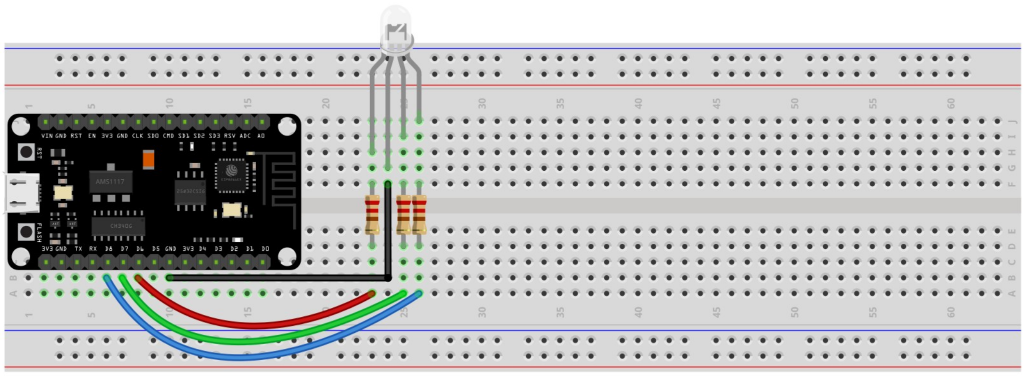
- Development board;
- 1 x RGB LED;
- 3 x 150Ω resistor ;
- Breadboard 830p;
- 4 x male-to-male jumper wire;
- Micro USB – Type A USB Cable;

23.3 Connections

Here is the schematic:



Below, you can find a visual representation of the connections:



23.4 Code

The code for this lesson is similar with the one from the previous lesson, but we will use three sliders, one for each colour. More details can be found in the code, located in folder **"Lesson 18: Dim an RGB LED from web"**.

At the beginning of the code we need to declare the pins used by the RGB LED, which contains three LEDs, a red one, a green one and a blue one. The **connectToWiFi()** function is the same as in the previous lesson. The **setupServer()** starts with the instruction for setting up the main page, and it is followed by three handlers for each LED. Also this function contains the instruction **server.begin()** which starts the server.

Code 23.4.1 The setupServer() function

```
void setupServer()
{
  //the method "server.on()" is to call a function when
  //the user access the location
  //the default location is "/"
  server.on("/", htmlIndex);
  server.on("/redFunction", redFunction);
  server.on("/greenFunction", greenFunction);
  server.on("/blueFunction", blueFunction);

  //start the server
  server.begin();

  //print in Serial Monitor that the HTTP server is started
  Serial.println("HTTP server started");
}
```

The **htmlIndex()** contains only the instruction for sending the HTML page stored as a String to the user.

Code 23.4.2 The htmlIndex() function

```
void htmlIndex()
{
  //send the message to the user
  server.send(200, "text/html", page);
}
```

The other three functions called in the **setupServer()** contains the same instructions. When one of the sliders is moved, the state will be sent to the server and the corresponding function will be called. In each function there are the instructions for converting the **String** to **int** and remap the values. Then in the **if** statement we will check if the value is **0**, then the respective LED will be turned Off, otherwise the value will be applied using **analogWrite()**. In the end of each function will be a instruction for sending an OK message.

Code 23.4.3 The function for controlling the red LED

```
void redFunction()
{
  int value1 = server.arg("state1").toInt();
  value1 = map(value1, 0, 100, 0, 1023);

  if(value1 == 0)
     digitalWrite(red, LOW);//turn off the led
  else
     analogWrite(red, value1);//change the brightness of red

  server.send(200, "text/html", "red");
}
```

Code 23.4.4 The function for controlling the green LED

```
void greenFunction()
{
  int value2 = server.arg("state2").toInt();
  value2 = map(value2, 0, 100, 0, 1023);

  if(value2 == 0)
     digitalWrite(green, LOW);//turn off the led
  else
     analogWrite(green, value2);//change the brightness of green

  server.send(200, "text/html", "green");
}
```

Code 23.4.5 The function for controlling the blue LED

```
void blueFunction()
{
  int value3 = server.arg("state3").toInt();
  value3 = map(value3, 0, 100, 0, 1023);

  if(value3 == 0)
     digitalWrite(blue, LOW);//turn off the led
  else
     analogWrite(blue, value3);//change the brightness of blue

  server.send(200, "text/html", "blue");
}
```

In the **setup()** function we will set all LEDs as **OUTPUT**, then start a serial communication with the computer and call the **connectToWiFi()** and **setupServer()** functions.

Code 23.4.6 The setup() function

```
void setup()
{
  pinMode(red, OUTPUT);
  pinMode(green, OUTPUT);
  pinMode(blue, OUTPUT);

  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  //wait 1 s for the Serial communication to start
  delay(1000);

  //call the two functions used to connect connect to wifi
  //and setup the server
  connectToWiFi();
  setupServer();
}
```

In the **loop()** function we will listern for any incoming request from the user using **server.handleClient()**.

Code 23.4.7 The loop() function

```
void loop()
{
  //the method below is used to manage the incoming request
  //from the user
  server.handleClient();
}
```

The HTML page is similar with the one from the previous lesson, the only difference is that this time we will have three sliders. In the body of the page we need to create three sliders and besides each slider will be a block that displays the current state of the slider (which is also the brightness of the LED).

Code 23.4.8 The HTML page body

```
<body>
<div class='container-fluid text-center'>
    <div class='row'>
        <div class='col-sm-12' style='color:red;font-size:7vh'>
            <p>Use the sliders to control the brightness.</p>
        </div>
    </div>
    <br/>
    <div class='row'>
        <div class='col-sm-12' style='color:red;font-size:3vh'>
            Red:
            <b id='status1'></b>
        </div>
    </div>
    <div class='col-sm-12'>
          <input type='range' min='0' max='100' class='form-control-range' id='slider_red'/>
    </div>

    <div class='row'>
        <div class='col-sm-12' style='color:green;font-size:3vh'>
            Green:
            <b id='status2'></b>
        </div>
    </div>
    <div class='col-sm-12'>
          <input type='range' min='0' max='100' class='form-control-range' id='slider_green'/>
    </div>

    <div class='row'>
        <div class='col-sm-12' style='color:blue;font-size:3vh'>
            Blue:
            <b id='status3'></b>
        </div>
    </div>
    <div class='col-sm-12'>
          <input type='range' min='0' max='100' class='form-control-range' id='slider_blue'/>
    </div>
</div>
</body>
```

The JavaScript part is also multipled by 3. We will use **.on()** method, with the **change** event association, three times and we will proceed the same as in the previous lesson.

Code 23.4.9 JavaScript

```
    <script>
        $('#slider_red').on('change', function(){
            var state_slider_red = $('#slider_red').val();

            $('#status1').html(state_slider_red + ' %');

            $.ajax({
              url: '/redFunction',
              type: 'POST',
              data: {state1: state_slider_red}
            });
        });

        $('#slider_green').on('change', function(){
            var state_slider_green = $('#slider_green').val();

            $('#status2').html(state_slider_green + ' %');

            $.ajax({
              url: '/greenFunction',
              type: 'POST',
              data: {state2: state_slider_green}
            });
        });

        $('#slider_blue').on('change', function(){
            var state_slider_blue = $('#slider_blue').val();

            $('#status3').html(state_slider_blue + ' %');

            $.ajax({
              url: '/blueFunction',
              type: 'POST',
              data: {state3: state_slider_blue}
            });
        });
    </script>
```

24. Lesson 19: Control a motor from web

24.1 Overview

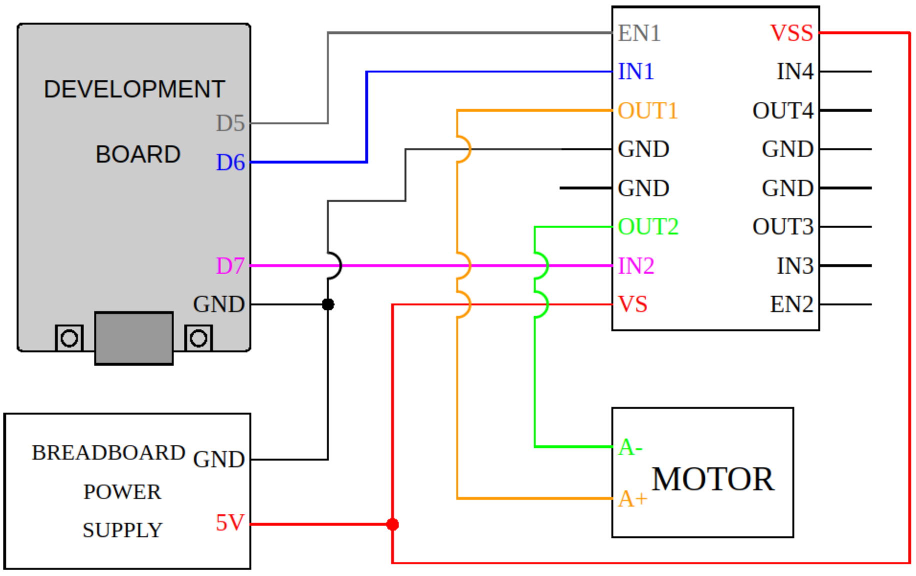
In this lesson you will learn how to control the speed and steering of a motor from a browser. This lesson combines the lesson **Motor Control** and lesson **WiFi**.

24.2 Components required

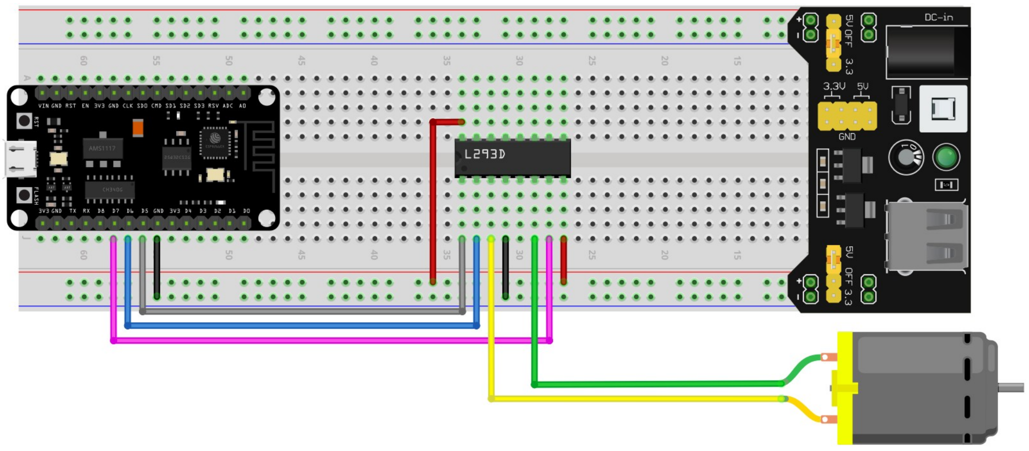
- Development board;
- Micro USB – Type A USB cable;
- L293D H-Bridge Motor Driver;
- Breadboard 830p;
- Breadboard power supply;
- 9 x male-to-male jumper wires;
- 1 x DC motor;

24.3 Connections

Below, you can find the schematic:



Below, you can find a visual representation of the connections:



24.4 Code

The code is based on the lesson **Motor Control**, but this time we will modify the speed of the motor using a slider and the direction using two buttons. The code can be found in the folder **"Lesson 19: Control a motor from web"**.

The code starts with the inclusion of the library used for the web server, declaration of two variables that will store the credentials of the wireless network and then start an instance of the **ESP8266WebServer** class. Next we have to declare the pins used for controlling the speed and direction of the motor and a variable that will store the speed value.

Code 24.4.1 Variables declaration

```
#include <ESP8266WebServer.h>

const char* ssid = "..................";
const char* password = "..............";

ESP8266WebServer server(80);

//the int variable "motorspeed_pin" stores the pin used to control the speed
of the motor
const int motorspeed_pin = D5;

//the next two variables store the pins used to control the direction of the
motor
const int DIRA = D6;
const int DIRB = D7;

//the int variable "motorspeed" stores a value between 0 and 1023
//used with the function analogWrite(pin, value);
//more details can be found in the lesson Motor Control
//initially is set to 100%
int motorspeed = 1023;
```

The **connectToWiFi()** function is unchaged and in the **setupServer()** function we have a instruction for setting up the main page, and we need handlers for another four functions that will be called when the specified location is accessed: **forward()** is the function which rotates the motor in one direction (forward), **backward()** is the function that rotates the motor in the opposite direction (backward), **stopp()** turns off the motor and **setmotorspeed()** reads the state of the slider and updates the value of the speed. At the end of the **setupServer()** function we have a instruction that will start the server.

Code 24.4.2 The function for setting up the server

```
void setupServer()
{
  //the method "server.on()" is to call a function when
  //the user access the location
  //the default location is "/"
  server.on("/", htmlIndex);;
  server.on("/forward", forward);
  server.on("/stopp", stopp);
  server.on("/backward", backward);
  server.on("/setmotorspeed", setmotorspeed);

  //start the server
  server.begin();

  //print in Serial Monitor that the HTTP server is started
  Serial.println("HTTP server started");
}
```

In the **setup()** function we will set the pins used by the motor as **OUTPUT**, then start a serial communication with the computer and call the **connectToWiFi()** and **setupServer()** functions.

Code 24.4.2 The setup() function

```
void setup()
{
  //the following instruction initialises the pin stored in the
  //variable motorspeed_pin(also DIRA and DIRB) as OUTPUT
  pinMode(motorspeed_pin, OUTPUT);
  pinMode(DIRA, OUTPUT);
  pinMode(DIRB, OUTPUT);

  //start the Serial communication with a baud rate of 115200
  Serial.begin(115200);

  //wait 1 s for the Serial communication to start
  delay(1000);

  //call the two functions used to connect connect to wifi
  //and setup the server
  connectToWiFi();
  setupServer();
}
```

In the **loop()** function we will listen for any request from the user.

Code 24.4.3 The loop() function

```
void loop()
{
  //the method below is used to manage the incoming request
  //from the user
  server.handleClient();
}
```

The **htmlIndex()** function only sends the web page to the user.

Code 24.4.4 The htmlIndex() function

```
void htmlIndex()
{
  //send the message to the user
  server.send(200, "text/html", page);
}
```

The **setmotorspeed()** function is called everytime the slider changes its value, and the current state of the slider it is syncronized with the server. In this function we will convert the value sent from the user from **String** to **int**, then check if that value is **0**, then set the **motorspeed** value to **0**, otherwise remap that value to a convenient one.

Code 24.4.5 Read the value of the slider

```
void setmotorspeed()
{
  //server.arg(name) returns the value (as string) of the argument "name"
  //using the method String.toInt(), we convert a String variable to int
  motorspeed = server.arg("motorspeed").toInt();

  if(motorspeed == 0)
  {
    motorspeed = 0;
  }
  else
  {
      //if using the slider at 1-5%(or more) the motor does not start
      //you can increase the value "toLow" to the point when the motor
      //starts spinning even from 1%
      motorspeed = map(motorspeed, 0, 100, 200, 1023);
  }

  //display a message in Serial Monitor to
  //see when the function is called
  Serial.println(motorspeed);

  //send an OK message(see more details below)
  server.send(200,"text/html","ok");
}
```

In the **forward()** function, created to turn on the motor forward, we have three main instructions: the first one is for setting the speed of the motor using **analogWrite()**, the second one is for setting the **DIRA** pin to **LOW** and the last one is setting the **DIRB** pin to **HIGH**, using **digitalWrite()**.

Code 24.4.6 Turn on the motor forward

```
void forward()
{
  //set the speed of the motor
  analogWrite(motorspeed_pin, motorspeed);

  //set the direction of the motor forward
  digitalWrite(DIRA, LOW);
  digitalWrite(DIRB, HIGH);

  //send an OK message(see more details below)
  server.send(200,"text/html","forward");

  //display a message in Serial Monitor to
  //see when the function is called
  Serial.println("Forward");
}
```

The **backward()** function turns on the motor backward, and it is similar with the **forward()** function, the only difference it is that we need to set the **DIRA** pin to **HIGH** and the **DIRB** pin to **LOW** (this will change the direction).

Code 24.4.7 Turn on the motor backward

```
void backward()
{
  //set the speed of the motor
  analogWrite(motorspeed_pin, motorspeed);

  //set the direction of the motor backward
  digitalWrite(DIRA, HIGH);
  digitalWrite(DIRB, LOW);

  //send an OK message(see more details below)
  server.send(200,"text/html","backward");

  //display a message in Serial Monitor to
  //see when the function is called
  Serial.println("Backward");
}
```

We also need another function that will stop the motor. This function is **stopp()** and here we need to set the speed pin to **LOW**. In this case, the direction pins are irrelevant, but we can set them to **LOW**.

Code 24.4.8 Turn off the motor

```
void stopp()
{
  //this instruction is used to set the speed of the motor to 0 (off)
  digitalWrite(motorspeed_pin, LOW);

  //in these instructions the state is irrelevant because the motor is off
  digitalWrite(DIRA, LOW);
  digitalWrite(DIRB, LOW);

  //send an OK message(see more details below)
  server.send(200,"text/html","stop");

  //display a message in Serial Monitor to
  //see when the function is called
  Serial.println("Stop");
}
```

Now, let's discuss about the HTML page. This page is stored as a string and will be sent to the user from the **htmlIndex()** function. In the **head** of the page we will put the links for Bootstrap, jQuery and Font Awesome. In the **body** we will create two buttons using Font Awesome, and when one button is pressed, using JavaScript we will call a function to rotate the motor. Also, there is a slider that controls the speed of the motor. When the state of the slider is modified, the new value will be sent to the server and the **motorspeed** variable will be changed. The

HTML page is structured using Bootstrap grid system, and if you have trouble understanding the HTML **body**, please check the theory lesson.

Code 24.4.9 The HTML body

```
<body>
  <div class='container-fluid' style='font-size:10vh;color:red'>
    <div class='row'>
      <div class='col-sm-12 text-center'>
        <h1 id='status'>Control App</h1>
      </div>
    </div>

    <div class='row'>
      <div class='col-sm-12 text-center'>
        <i id='button-forward' class='fas fa-chevron-circle-up'></i>
      </div>
    </div>

    <div class='row'>
      <div class='col-sm-12 text-center'>
        <i id='button-backward' class='fas fa-chevron-circle-down'></i>
      </div>
    </div>

    <div class='row' style='height: 10vh;'> </div>

    <div class='row'>
      <div class='col-sm-12 text-center'>
        <h3>Use the slider for adjusting the motorspeed</h3>
      </div>
    </div>
    <div class='row'>
      <div class='col-sm-12 offset-lg-4 col-lg-4'>
        <input type='range' class='form-control-range' id='motorspeed-slider'/>
      </div>
    </div>

  </div>
</body>
```

In order for the buttons and the slider to work, we had to insert into the page a script that sends the values for the slider to the server. For the slider we are using the event **change**, a variable that reads the state of the slider and an **ajax** call that will trigger the **setmotorspeed()** function, sending the slider value along with it. For the buttons we are using two events, so that the page can run flawlessly on devices with or without touch capabilities. **touchstart** and **mousedown** are used when the button is pressed and, using **ajaxb,** we will call a function that will start the motor forward or backward, depending on which button is pressed. **touchend** and **mouseup** are used when the button is released and a function will be called to stop the motor.

Code 24.4.10 JavaScript

```
<script language='javascript'>
    $('#motorspeed-slider').on('change', function() {
        var variable = $('#motorspeed-slider').val();

        $('#status').html('Speed: ' + variable + '%');
        $.ajax({
            url: '/setmotorspeed',
            type: 'POST',
            data: {motorspeed: variable},
        });
    });

    $('#button-forward').on('touchstart mousedown', function() {
        $('#status').html('Forward');
        $.ajax('/forward');
    });

    $('#button-forward').on('touchend mouseup', function() {
        $('#status').html('Stop');
        $.ajax('/stopp');
    });

    $('#button-backward' ).on('touchstart mousedown', function() {
        $('#status').html('Backward');
        $.ajax('/backward');
    });

    $('#button-backward').on('touchend mouseup', function() {
        $('#status').html('Stop');
        $.ajax('/stopp');
    });
</script>
```

25. Lesson 20: Display the distance in web

25.1 Overview

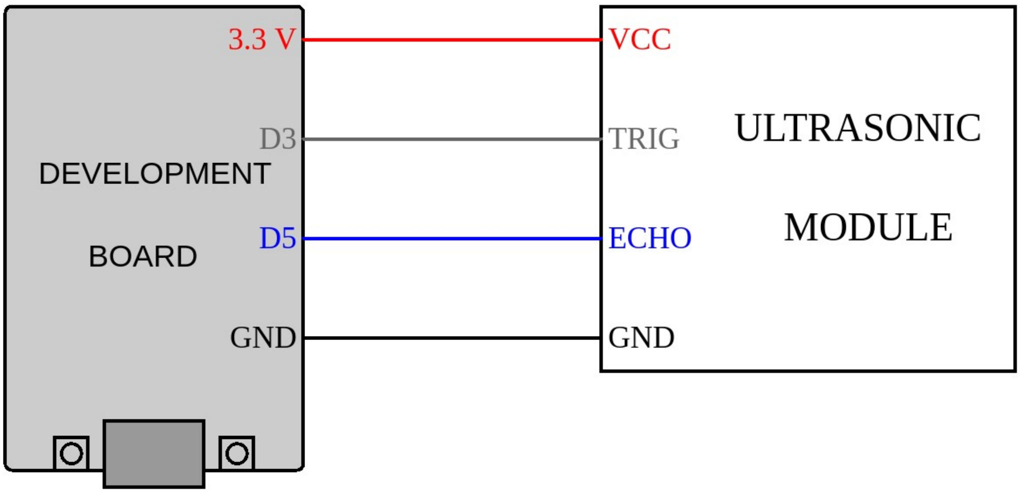
This lesson is similar to the lesson Ultrasonic HC-SR04+, but in addition, you will learn to display the distance on a web page hosted by the development board.

25.2 Components required

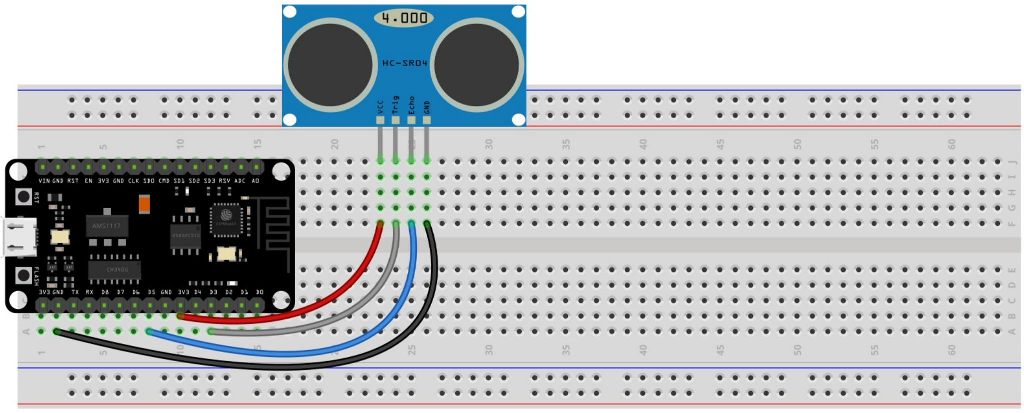
- Development board;
- Ultrasonic module HC-SR04+;
- Breadboard 830p;
- 4 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

25.3 Connections

Below you can see the schematic:



Next you can see the visual representation:



25.4 Code

The code is based on the lesson **Ultrasonic HC-SR04+** but adds many new concepts, however, most of them are explained in the **Theory lesson**, including JSON (JavaScript Object Notation) that we are using in our code. You can find this code is folder **"Lesson 20: Display the distance in web"**.

In the lesson **Control a LED from web** the development board received data from web, while in this lesson the development board sends data to web.

The code starts by including the **ESP8266WebServer.h** library and declaring two variables that will store the credentials of the wireless network. Then, like in the **Ultrasonic HC-SR04+** lesson, we will declare the pins used by the ultrasonic module and two variables that will help us calcutate the distance, **duration** and **distance**. Also, we need an instance for the **ESP8266WebServer** class. The **connectToWiFi()** function remains unchanged, you can go to the **Wireless Connectivity** lesson for more details.

In the **setupServer()** function we need to pass to the **on** function a handler, **htmlIndex**, that will be triggered when a HTTP request is received on the root ("/"). Also we need to pass to the **on** method a handler, **refresh**, that will send to the user the current distance. In this function we will use the **server.begin()** instruction to start the server.

Code 25.4.1 The setupServer() function

```
void setupServer()
{
  //the method "server.on()" is to call a function when
  //the user access the location
  //the default location is "/"
  server.on("/", htmlIndex);

  server.on("/refresh", refresh);

  //start the server
  server.begin();

  //print in serial manager that the HTTP server is started
  Serial.println("HTTP server started");
}
```

The **htmlIndex()** function is created for sending the HTML page to the user when the default location ("/") is accessed by a user.

Code 25.4.2 Send the HTML page to the user

```
void htmlIndex()
{
  //send the message to the user
  server.send(200, "text/html", page);
}
```

In the **setup()** function we will set the pins used by the ultrasonic module, **trigger** as **OUTPUT** and **echo** as INPUT, then start a serial communication with the computer and call the **connectToWiFi()** and **setupServer()** functions.

Code 25.4.3 The setup() function

```
void setup()
{
  //the trigger pin (transmitter) must be set as OUTPUT
  pinMode(trigPin, OUTPUT);

  //the echo pin (receiver) must be set as INPUT
  pinMode(echoPin, INPUT);

  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  //wait 1s so the serial communication has enough time to start
  delay(1000);

  //call the two functions used to connect to the wireless network
  //and setup the web server
  connectToWiFi();
  setupServer();
}
```

In the **loop()** function we will calculate the distance, but we also need the time that the sound travels from the transmitter to object and back to the receiver. In order to obtain the duration, we have to generate an ultrasound by turning the transmitter **ON** for 10 microseconds and the duration is recorded using the following instruction:

```
duration = pulseIn(echoPin, HIGH)
```

For the distance we will use the next instruction:

```
distance = duration*0.034/2
```

Code 25.4.4 The loop() function

```
void loop()
{
  //the method below is used to manage the incoming requests
  //from the user
  server.handleClient();

  //set the trigPin to LOW in order to prepare for the next reading
  digitalWrite(trigPin, LOW);

  //delay for 2 microseconds
  delayMicroseconds(2);

  //generate an ultrasound for 10 microseconds then turn off the transmitter
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  //reads the echoPin, returns the sound wave travel time in microseconds
  duration = pulseIn(echoPin, HIGH);

  //using the formula shown in the guide, calculate the distance
  distance = duration*0.034/2;
}
```

Another function passed to the **on** method in **setupServer()** is **refresh()**. In this function we will define a char array that will act as a buffer for **sprintf(char *buffer, const char *string, ...)**. **sprintf** stands for "String print", which, instead of printing in console, it stores the output in a char buffer. In that buffer we will store the distance, and, using **server.send**, the distance will be send to JavaScript.

Code 25.4.5 Send data to user

```
void refresh()
{
  //create a char array
  char messageFinal[100];

  //put the distance value in buffer
  sprintf(messageFinal, "%.2f", distance);

  //send data to user
  server.send(200, "application/javascript", messageFinal);
}
```

The HTML page is stored as a String, the **head** contains the jQuery inclusion, and in the **body** we will make a table with one row and two columns, one containing a block with a text, and in the other one will be displayed the distance to the closest object.

Code 25.5.6 The HTML body

```
  <body>
    <h2>Hello from Plusivo!</h2>
    <table style='font-size:20px'>
      <tr>
        <td>
          <div>Distance:  </div>
        </td>
        <td>
          <div id='Distance'></div>
        </td>
      </tr>
    </table>
  </body>
```

For exchanging data between server and user, we need to insert a script. In order to update the page rapidly and periodically, you have to create two functions. The main function is used to call another function at a specified interval.

```
setInterval(refreshFunction, interval);
```

Parameters:

- **refreshFunction:** is a function that is called periodically;
- **interval:** represents the duration (in ms), it waits to call the **refreshFunction** again

The second function is used to update the **HTML elements**. Example:

```
function refreshFunction(){
  $.getJSON('/refresh', function(result){
    $('#Distance').html(result);
  });
}
```

Parameters**:**

- **$.getJSON**() is used to access a page and get the value available there.
- /**refresh** represents the location where the value is stored.
- **$('id').html("content")** is used to change the content of a HTML tag (identified by "**id**") .

Code 25.5.7 JavaScript

```
<script>
 $(document).ready(function(){
   setInterval(refreshFunction,100);
 });

 function refreshFunction(){
   $.get('/refresh', function(result){
     $('#Distance').html(result);
   });
 }
</script>
```

26. Lesson 21: Potentiometer, servo, DHT11 and web server

26.1 Overview

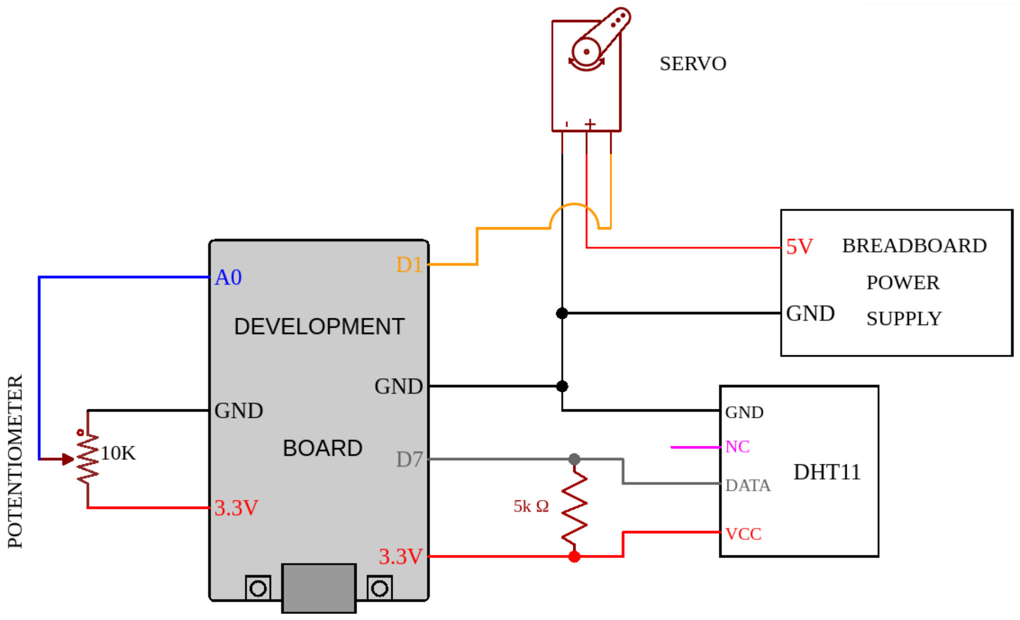
This lesson is similar with the previous one and combines three lessons, **DHT11**, **Potentiometer and Servo Motor** and **Wireless Connectivity**. In this lesson we are going to display in a browser the temperature and humidity, read using the DHT11 sensor, and the angle of the potentiometer.

26.2 Components required

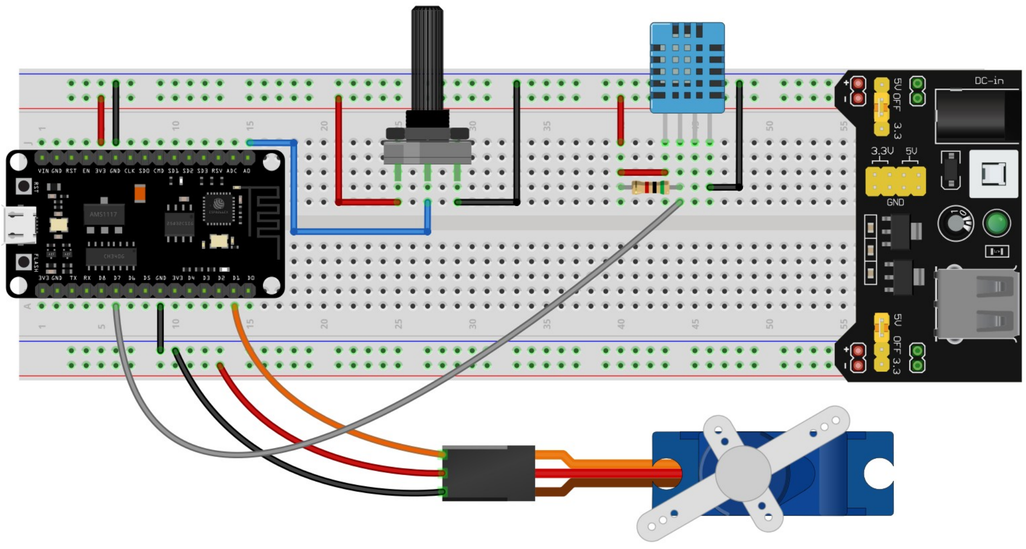
- Development board;
- Potentiometer;
- Servomotor SG90;
- 13 x male-to-male jumper wires;
- Breadboard power supply;
- 9 V battery;
- Cable 9 V battery to DC jack;
- Micro USB – Type A USB cable;
- DHT11 Humidity and Temperature sensor;
- Breadboard 830p;
- 1 x 5000 Ω resistor;

26.3 Connections

Below you can find the schematic:



Also, you can find below a visual representation of the project:



26.4 Code

The code for this lesson is similar with the one provided for the previous lesson because we are sending some data to the user that will be displayed in the web page. The code for this lesson can be found in the **"Lesson 21: Potentiometer, servo, DHT11 and web server"** folder.

As usual, at the beginning of the code we will include the libraries used, which are **ESP8266WebServer.h**, **SimpleDHT.h** and **Servo.h**, and create an instance for each library. Also we need two variables that will store the credentials of the wireless network that the board will connect to, a variable that will store the pin used by the DHT11 module and three variables that will store the temperature, humidity and angle read using the two modules.

Code 26.4.1 Include the libraries and define the variables used

```
#include <ESP8266WebServer.h>

//create an instance of the ESP8266WebServer library
ESP8266WebServer server(80);

//create two variables that will store the credentials of the wireless
//network
const char* ssid = "................";
const char* password = "...........";

//the library "Servo.h" is used to control a servo motor using
//PWM technique
#include <Servo.h>

//declare a new object called servo
Servo servo;

#include <SimpleDHT.h>

//create an instance for the SimpleDHT library
SimpleDHT11 dht11;

//define the digital pin used to connect the module
const int dht_pin = D7;

//declare two byte variables for temperature and humidity
byte temperature = 0;
byte humidity = 0;

//create a variable that will store the state of the potentiometer
int value = 0;
```

In the **setup()** function we will start a serial communication with the computer, then attach the servo to digital pin **D1** and call the **setupServer()** and **connectToWiFi()** functions.

Code 26.4.2 The setup() function

```
void setup()
{
  //start the serial communication with the computer at 115200 bits/s)
  Serial.begin(115200);

  //attach the servo on digital pin D1
  servo.attach(D1);

  //call the two functions used to connect to wifi
  //and setup the web server
  connectToWiFi();
  setupServer();

  //wait 4 s for the server to start
  delay(4000);
}
```

The **connectToWiFi()** function remains unchanged and in the **setupServer()** function we will use the **.on** method to call a function to send the web page to the user, which is **htmlIndex()**, and to call the **refresh()** function that will send data to the user.

Code 26.4.3 Set up the server

```
void setupServer()
{
  //the method "server.on()" is to call a function when
  //the user access the location
  //the default location is "/"
  server.on("/", htmlIndex);
  server.on("/refresh", refresh);

  //start the server
  server.begin();

  //print in serial manager that the HTTP server is started
  Serial.println("HTTP server started");
}
```

As previously stated, the **htmlIndex()** function sends the HTML page to the user.

Code 26.4.4 Send the HTML page to the user

```
void htmlIndex()
{
  //send the message to the user
  server.send(200, "text/html", page);
}
```

As in the previous lesson, **Display distance in web**, we have a function that will be called from JavaScript every 100 ms and will send to the user the new values of the angle, the temperature and the humidity.

Code 26.4.5 The refresh function

```
void refresh()
{
  //create a char array
  char messageFinal[100];

  char container[100] =
  R"(
    {
      "angle": %d,
      "temperature":  %d,
      "humidity": %d
    }
  )";

  //put the values in messageFinal
  sprintf(messageFinal, container, value, temperature, humidity);

  //send data to user
  server.send(200, "application/javascript", messageFinal);
}
```

In the **loop()** function we will get the temperature and the humidity, and show them in Serial Monitor. Next, we will get the state of the potentiometer, update the angle of the servo and display it in Serial Monitor.

Code 26.4.6 The loop() function. Calculate the temperature, humidity and angle

```
void loop()
{
  //the method below is used to manage the incoming request
  //from the user
  server.handleClient();

  //read the values
  dht11.read(dht_pin, &temperature, &humidity, NULL);

  //display the values in Serial Monitor
  Serial.print("Temperature: ");
  Serial.print(temperature);
  Serial.println(" *C");
  Serial.print("Humidity: ");
  Serial.print(humidity);
  Serial.println(" H");
  Serial.println();

  //read the value on pin A0
  //the pin is able to read a value between 0 and 1024 corresponding
  //to 0 V and 3.3 V
  value = analogRead(A0);

  //remap the analog value to a new range (from 0 to 180) as the
  //servo can turn max 180 degrees.
  value = map(value, 0, 1024, 0, 180);

  //turn the servo motor accordingly to the angle stored in value
  servo.write(value);

  //print in Serial Monitor the current angle of the servo
  Serial.print("Angle: ");
  Serial.println(value);

  //pause the code for 100 ms;
  delay(100);
}
```

Now, let's talk about the HTML page. In the **head** we will include the link for jQuery and in the **body** we will have a table with three rows and two columns. In the first column you can find the name and in the second column you can find the value.

Code 26.4.7 The HTML page

```
<head>
  <script src='https://code.jquery.com/jquery-3.3.1.min.js'></script>
  <title>Plusivo</title>
</head>

<body>
  <h2>Hello from Plusivo!</h2>
  <table style='font-size:20px'>
    <tr>
        <td>
            <div>Angle:  </div>
        </td>
        <td>
            <div id='Angle'></div>
        </td>
    </tr>

    <tr>
        <td>
            <div>Temperature:  </div>
        </td>
        <td>
            <div id='Temperature'></div>
        </td>
    </tr>

    <tr>
        <td>
            <div>Humidity:  </div>
        </td>
        <td>
            <div id='Humidity'></div>
        </td>
     </tr>
  </table>
</body>
```

The JavaScript part has a **setInterval** instruction that will call the **refreshFunction** every 100 ms. In this function you can find the **getJSON** method that will get some data from the server. Inside this method we will update the values in the defined table from the HMTL body, using the **.html** method.

Code 26.4.8 JavaScript

```
<script>
 $(document).ready(function(){
   setInterval(refreshFunction,100);
 });

 function refreshFunction(){
   $.getJSON('/refresh', function(e){
     $('#Angle').html(e.angle);
     $('#Temperature').html(e.temperature);
     $('#Humidity').html(e.humidity);
   });
 }
</script>
```

27. Lesson 22: Buzzer from web

27.1 Overview

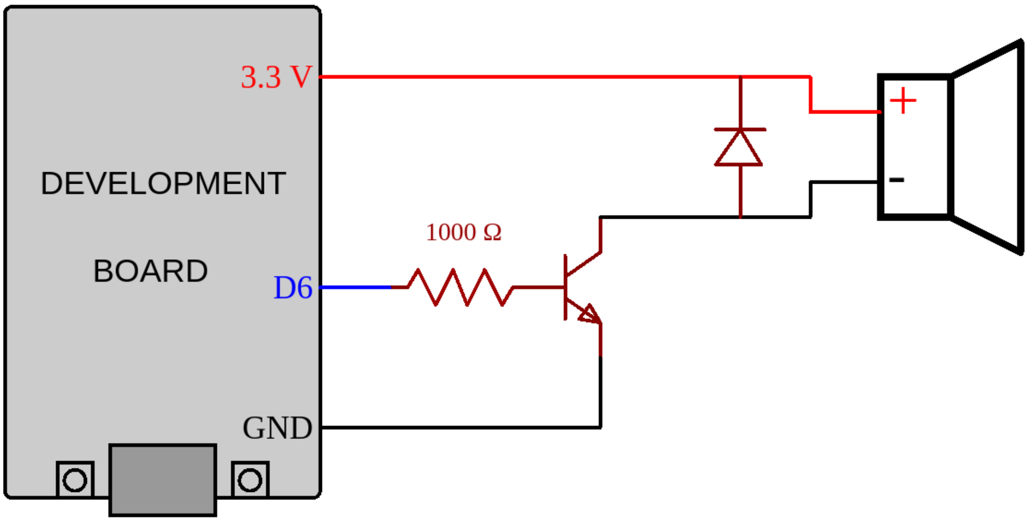
In this lesson you will learn how to turn On and Off a buzzer from a web browser.

27.2 Components required

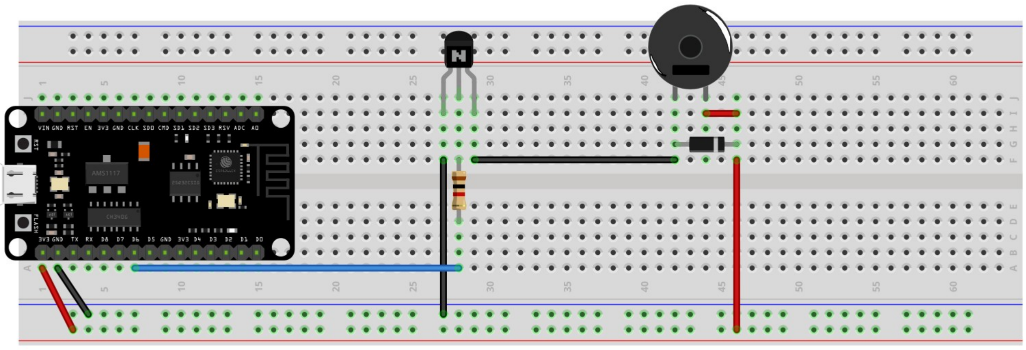
- Development board;
- Breadboard 830p;
- 1 x passive buzzer;
- Diode;
- Transistor;
- 1 x 1000 Ω resistor;
- 7 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

27.3 Connections

Below you can find the schematic:



Also, you can find below a visual representation of the project:



27.4 Code

The code for this lesson is almost identical with the one from the lesson **Control an LED from web**. You can find this code in the folder "**Lesson 22: Buzzer from web**".

Firstly, we need to include the **ESP8266WebServer.h** library, declare two variables for the credentials of the wireless network, declare the pin used by the buzzer, declare a String variable that will help us later and create an instance of the **ESP8266WebServer** class.

The **connectToWiFi()** function still remains the same, in the **setupServer()** function we need to pass to the **on** method a handler that will send the HTML page to the user, and another handler that will be triggered when the button is clicked. Also in this function is an instruction that will start the server.

Code 27.4.1 Set up the server

```
void setupServer()
{
  server.on("/", htmlIndex);
  server.on("/buzzer_function", buzzer_function);

  //start the server
  server.begin();

  //print in Serial Monitor that the HTTP server is started
  Serial.println("HTTP server started");
}
```

The **htmlIndex()** function contains a single instruction that will send the HTML page to the user.

Code 27.4.2 Send the HTML page to the user

```
void htmlIndex()
{
  //send the message to the user
  server.send(200, "text/html", page);
}
```

The function that will control the buzzer is **buzzer_function()**. This function is called everytime the button is clicked, and, using the String variable **container** in which we will store the current state of the button, we will decide if the state is **On**, then the buzzer will turn on at a frequency of 1000 Hz, otherwise the buzzer will be turned off. At the end of this function we need to send on OK message.

Code 27.4.3 The function for controlling the buzzer

```
void buzzer_function()
{
  container = server.arg("state");
  if(container == "On")
  {
    //turn on the buzzer at a frequency of 1000Hz
    tone(buzzer, 1000);
  }
  else
  {
    //turn off the buzzer
    noTone(buzzer);
  }
  server.send(200, "text/html", "ok");
}
```

The **setup()** function has an instruction to set the pin used by the buzzer as **OUTPUT**, another instruction for starting a serial communication with the computer and, also, contains a call of the **connectToWiFi()** function and one of the **setupServer()** function.

Code 27.4.3 The setup() function

```
void setup()
{
  //set the mode of the pin used by the buzzer as OUTPUT
  pinMode(buzzer, OUTPUT);

  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  delay(1000);

  connectToWiFi();
  setupServer();
}
```

In the **loop()** function we will put only one instruction used for listening to any incoming requests.

Code 27.4.4 The loop() function

```
void loop()
{
  //the method below is used to manage the incoming request
  //from the user
  server.handleClient();
}
```

The HTML page is very simple. In the **head** we will include links for Font Awesome, Bootstrap and jQuery, and we will define two CSS classes that contains only color.

Code 27.4.5 The HTML head

```
    <head>
        <title>Buzzer control</title>
        <meta name='viewport' content='width=device-width, initial-scale=1'>
        <link rel='stylesheet'
href='https://use.fontawesome.com/releases/v5.1.0/css/all.css'/>
        <link rel='stylesheet'
href='https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css'/>
        <script src='https://code.jquery.com/jquery-3.3.1.min.js'></script>
        <style>
            .css_on
            {
                color: black;
            }
            .css_off
            {
                color: white;
            }
        </style>
    </head>
```

The **body** is created using Bootstrap grid system and has three rows: one that displays a text, one that has the button and one with the speaker from Font Awesome. The most important aspect is that the speaker tag to have the **css_off** class.

Code 27.5.6 The HTML body

```
  <body>
    <div class='container-fluid text-center'>
        <div class='row text-center'>
            <div class='col-sm-12' style='color:red;font-size:7vh'>
                <p>Click/touch the button.</p>
            </div>
        </div>
        <div class='row'>
            <div  class='col-sm-12'>
                <input type='button' class='btn btn-primary' id='id_button' value='Off'
style='font-size:7vh'>
            </div>
        </div>
        <br/>
        <div class='row'>
            <div  class='col-sm-12'>
                <i id='speaker' class='fas fa-volume-up css_off' style='font-size:15vh'></
i>
            </div>
        </div>
    </div>
    </body>
```

In the JavaScript part we will modify the value of the button, the color of the speaker and send the current value to the server using **ajax**. When the button is clicked, we will check if the class assigned is **css_off** and if that is true, we will set the value to **On** and remove the **css_off** class using **.removeClass()** method and add the new class **css_on**. Also here we will have an **ajax** call for sending a request to the server and send some data to the server, in this case we are sending "On" (the buzzer will turn on at a frequency of 1000 Hz). If the statement is false, then we will set the value to **Off**, remove the **css_on** class, add **css_off** class and make an **ajax** call for sending a request to the server and send some data to the server, in this case we are sending "Off" (the buzzer will turn off).

Code 27.5.7 JavaScript

```
    <script>
      $(document).ready(function(){
        $('#id_button').click(function(){
            var current_state = $('#speaker').hasClass('css_off');

            if(current_state == true)
            {
                $.ajax({
                  url:'/buzzer_function',
                  type: 'POST',
                  data: {state: 'On'},
                });
                $('#id_button').val('On');
                $('#speaker').removeClass('css_off').addClass('css_on');
            }
            else
            {
                $.ajax({
                  url:'/buzzer_function',
                  type: 'POST',
                  data: {state: 'Off'},
                });
                $('#id_button').val('Off');
                $('#speaker').removeClass('css_on').addClass('css_off');
            }
        });
    });
    </script>
```

28. Lesson 23: Set the frequency of a buzzer from web

28.1 Overview

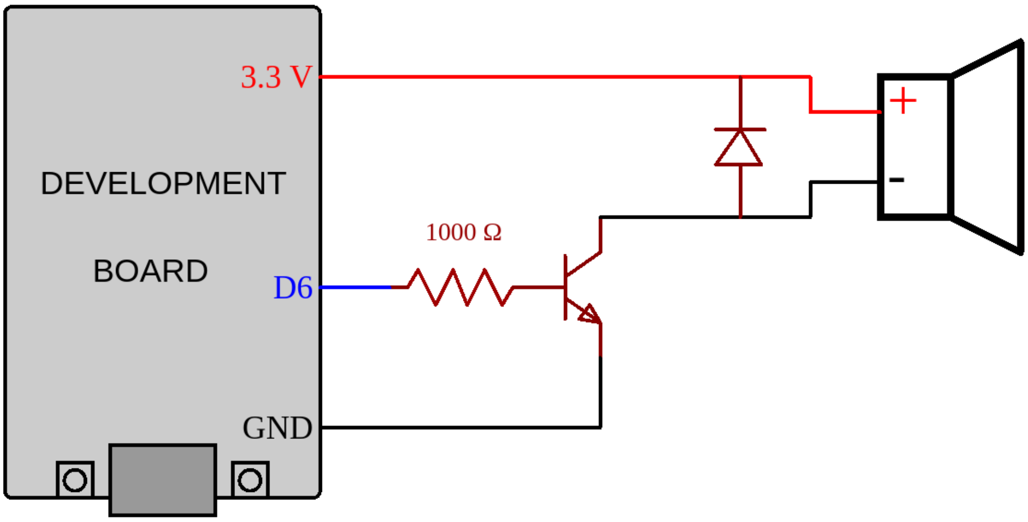
In this lesson you will learn how to turn On and Off a buzzer and control its frequency from a web browser.

28.2 Components required

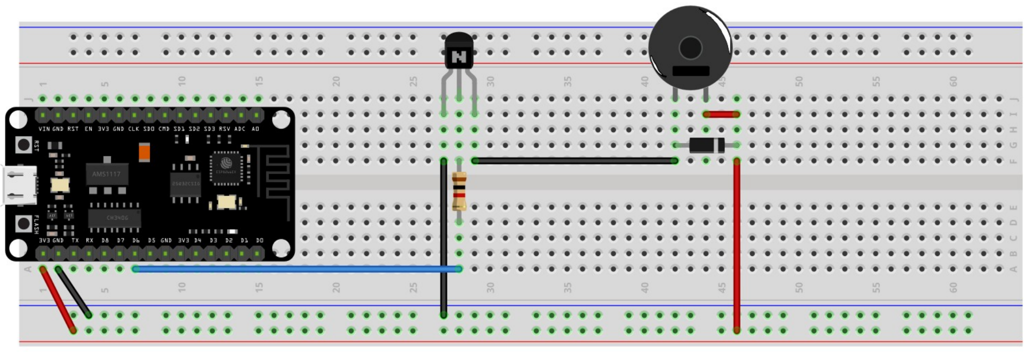
- Development board;
- Breadboard 830p;
- 1 x passive buzzer;
- Diode;
- Transistor;
- 1 x 1000 Ω resistor;
- 7 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

28.3 Connections

Below you can find the schematic:



Also, you can find below a visual representation of the project:



28.4 Code

The code for this lesson is an advanced version of the one from the previous lesson, because we will add a slider to control the frequency of the buzzer, where in the previous one, the frequency was fixed. You can find this code in the **"Lesson 23: Set the frequency of a buzzer from web"** folder.

The beginning of the code is identical with the previous one: include the **ESP8266WebServer.h** library, declare two variables that store the credentials of the wireless network, declare the pin used by the buzzer, define a String variable that will store the state of the button and an instance of the **ESP8266WebServer** class.

The **connectToWiFi()** function is unchanged, the **setupServer()** function starts by passing to the **on** method four handlers: one that sends the HTML page to the user, one that is triggered when the button is clicked and will modify the **state** variable, another function that will be triggered when the page is reloaded and will send to the user the previous states of the button and slider and the last function reads the state of the slider defined in the HTML page and will change the frequency of the buzzer. Also in the **setupServer()** function we will put an instruction that will start the server.

Code 28.4.1 Set up the server

```
void setupServer()
{
  server.on("/", htmlIndex);
  server.on("/button_state", button_state);
  server.on("/buzzer_frequency", buzzer_frequency);
  server.on("/refresh", refresh);

  //start the server
  server.begin();

  //print in Serial Monitor that the HTTP server is started
  Serial.println("HTTP server started");
}
```

In the **htmlIndex()** function we will send the HTML page to the user.

Code 28.4.2 Make the HTML page available to the user

```
void htmlIndex()
{
  //send the message to the user
  server.send(200, "text/html", page);
}
```

The **button_state()** function will be triggered everytime the button is pressed in the HTML web page. In this function we have an instruction that reads the value sent to the server using an **ajax** call. This value is the state of the button, which will be **On** or **Off**. Next, depending on the value of **container**, the buzzer will be turned on at the specified frequency, or will be turned off.

Code 28.4.3 Read the state of the button defined in the HTML page

```
void button_state()
{
  container = server.arg("state");

  if(container == "On")
  {
    tone(buzzer, frequency);
  }
  else
  {
    noTone(buzzer);
  }
  server.send(200, "text/html", "ok");
}
```

The **buzzer_frequency()** function is triggered when any change is made to the slider. The main instruction in this function is **server.arg("sound").toInt()**, which translates the value of the slider sent to the server using an **ajax** call. This value is stored in the **frequency** variable. Using an **if** statement and the value of the **container**, we will decide whether the buzzer will be turned On at the specified frequency or will be turned Off.

Code 28.4.4 Read the state of the slider and modify the frequency

```
void buzzer_frequency()
{
  frequency = server.arg("sound").toInt();

  if(container == "On")
  {
    tone(buzzer, frequency);
  }
  else
  {
    noTone(buzzer);
  }
  server.send(200, "text/html", "ok");
}
```

The last handler called in the **setupServer()** function is **refresh()**. This function will be triggered each time the page will be reloaded and will send to user the last state of the button and the last frequency set by the slider.

Code 28.4.5 Update the states when refreshing the page

```
void refresh()
{
  //store the last state of the button
  //"On", state = 1
  //"Off", state = 0;
  int state = 0;

  if(container == "On")
  {
    state = 1;
  }

  char data[60] = R"(
    {
      "state": %d,
      "frequency": %d
    }
  )";
  char output[100];
  sprintf(output, data, state, frequency);
  server.send(200, "application/javascript", output);
}
```

The **setup()** function is identical with the one from the previous lesson. Here we will set the pin used by the buzzer as **OUTPUT**, start a serial communication with the computer and call the **connectToWiFi()** and **setupServer()** functions.

Code 28.4.6 The setup() function

```
void setup()
{
  //set the mode of the pin used by the buzzer as OUTPUT
  pinMode(buzzer, OUTPUT);

  //start the Serial communication at the baudrates 115200
  Serial.begin(115200);

  delay(1000);

  connectToWiFi();
  setupServer();
}
```

In the **loop()** function we have an instruction that listens to any incoming requests from the user.

Code 28.4.7 The loop() function

```
void loop()
{
  //the method below is used to manage the incoming request
  //from the user
  server.handleClient();
}
```

The HTML page is similar with the one from the previous lesson. The **head** is similar, only in the **css_off** css class the color is changed from **white** to **lightgray**.

Code 28.4.8 The head of HTML page

```
    <head>
        <title>Buzzer control</title>
        <meta name='viewport' content='width=device-width, initial-scale=1'>
        <link rel='stylesheet'
href='https://use.fontawesome.com/releases/v5.1.0/css/all.css'/>
        <link rel='stylesheet'
href='https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css'/>
        <script src='https://code.jquery.com/jquery-3.3.1.min.js'></script>
        <style>
            .css_on
            {
              color: black;
            }
            .css_off
            {
              color: lightgray;
            }
        </style>
    </head>
```

In the HTML **body** we will add a row for the slider, and another that will display in real time the frequency of the buzzer.

Code 28.4.9 The HTML body

```
    <body>
    <div class='container-fluid text-center'>
        <div class='row'>
            <div class='col-sm-12' style='color:red;font-size:7vh'>
                <p>Click/touch the button.</p>
            </div>
        </div>
        <div class='row'>
            <div  class='col-sm-12'>
                <input type='button' class='btn btn-primary' id='id_button' style='font-size:7vh'>
            </div>
        </div>
        <br/>
        <div class='row'>
            <div  class='col-sm-12'>
              <i id='speaker' class='fas fa-volume-up css_off' style='font-size:15vh'></i>
            </div>
        </div>
        <div class='col-sm-12'>
          <input type='range' min='31' max='6000' class='form-control-range' id='slider'/>
        </div>
        <div class='row'>
          <div class='col-sm-12'>
            <h3 id='status'></h3>
          </div>
        </div>
    </div>
    </body>
```

In the JavaScript part, the instruction **$(document).ready(refreshFunction)** is used to trigger the **refreshFunction** every time the HTML page is reloaded. The **refreshFunction** contains a **getJSON** instruction that will read from the server the last registered state for the button and also the last frequency. Using these values, the visual representation of the page will remain the same after reloading the page.

Code 28.4.10 Get data from server when the page is reloaded

```
        $(document).ready(refreshFunction);

        function refreshFunction(){
          $.getJSON('/refresh', function(result){
            $('#slider').val(result.frequency);

            if(result.state == 1)
            {
                $('#id_button').val('On');
                $('#speaker').removeClass('css_off').addClass('css_on');
            }
            else
            {
                $('#id_button').val('Off');
                $('#speaker').removeClass('css_on').addClass('css_off');
            }
            $('#status').html('Frequency: ' + result.frequency + ' Hz');
          });
        };
```

Next, when the button is clicked it will trigger a function that will check for the class assigned to the speaker, and on the returned value, an **ajax** call will be made and will send to the server the state of the button (**On** or **Off**). Also the text on the button will be changed and, also, the css class of the speaker.

Code 28.4.11 Send the state of the button to the server

```
        $('#id_button').click(function(){
            var current_state = $('#speaker').hasClass('css_off');

            if(current_state == true)
            {
                $.ajax({
                  url:'/button_state',
                  type: 'POST',
                  data: {state: 'On'},
                });
                $('#id_button').val('On');
                $('#speaker').removeClass('css_off').addClass('css_on');
            }
            else
            {
                $.ajax({
                  url:'/button_state',
                  type: 'POST',
                  data: {state: 'Off'},
                });
                $('#id_button').val('Off');
                $('#speaker').removeClass('css_on').addClass('css_off');
            }
        });
```

The last part of the JavaScript is the one which deals with the slider. In a variable will be stored the value of the slider, this value will be displayed in the browser in the last row defined in the HTML **body**. Also, there is an **ajax** call that will send to the server the value of the slider, which will be the frequency of the buzzer.

Code 28.4.12 Send the state of the slider to the browser

```
        $('#slider').on('change', function(){
            var state_slider = $('#slider').val();

            $('#status').html('Frequency: ' + state_slider + ' Hz');
            $.ajax({
              url: '/buzzer_frequency',
              type: 'POST',
              data: {sound: state_slider}
            });
        });
```

29. Lesson 24: Piano

29.1 Overview

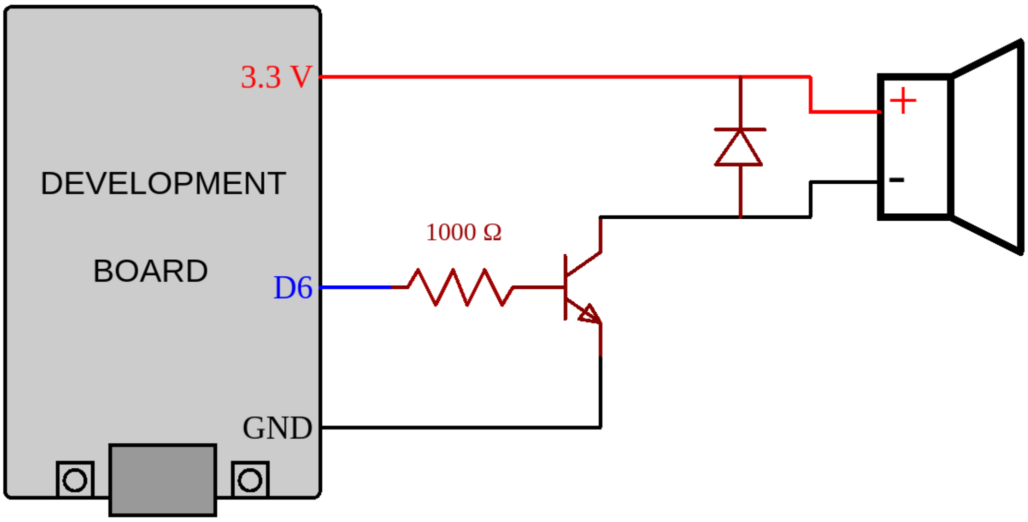
In this lesson you will learn how to create an octave from a piano and play different sounds on the buzzer. The octave will be created using CSS and the keyboard of the piano will be responsive.

29.2 Components required

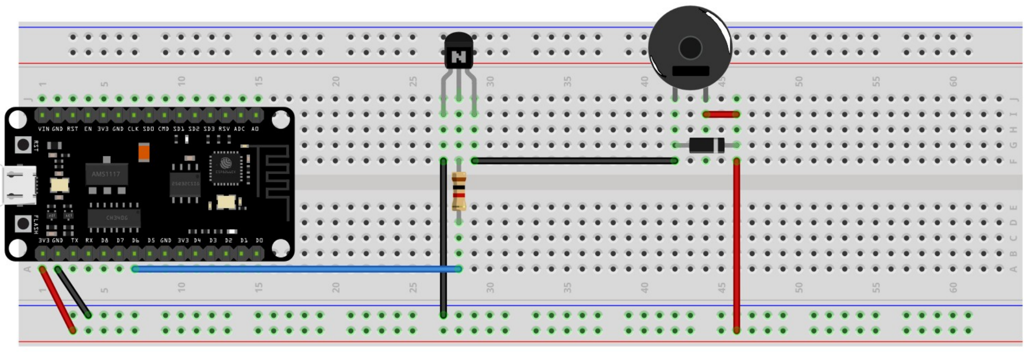
- Development board;
- Breadboard 830p;
- 1 x passive buzzer;
- Diode;
- Transistor;
- 1 x 1000 Ω resistor;
- 7 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

29.3 Connections

Below you can find the schematic:



Also, you can find below a visual representation of the project:



29.4 Code

For this lesson, the code is pretty simple for the server side and the HTML page is complicated because we are using CSS and JavaScript to design the piano and send data to the server. We will discuss about the functions created for the server and then about the HTML page. This code is located in the **"Lesson 24: Piano"** folder.

At the beginning of the code we will include the library used, then create two variables that will store the credentials of the wireless network that the board will be connecting to, create a variable that will store the pin used by the buzzer. Also, there are some variables for storing the frequencies of the notes and, then, an instance of the **ESP8266WebServer** class will be created.

Code 29.4.1 Variables declaration

```
#include <ESP8266WebServer.h>

//declare the credentials of the wireless network
const char* ssid = ".........";
const char* password = "........";

//declare the pin used by the buzzer
const int buzzer = D6;

//declare variables that will hold the frequencies of the
//notes from the sixth octave
const int c6 = 1047;
const int cs6 = 1109;
const int d6 = 1175;
const int ds6 = 1245;
const int e6 = 1319;
const int f6 = 1397;
const int fs6 = 1480;
const int g6 = 1568;
const int gs6 = 1661;
const int a6 = 1760;
const int as6 = 1865;
const int b6 = 1976;

ESP8266WebServer server(80);
```

In the **setup()** function we will set the pin used by the buzzer as **OUTPUT**, then start a serial communication with the computer and call the function for connecting to the wireless network and the function that will deal with the server part.

Code 29.4.2 The setup() function

```
void setup()
{
  pinMode(buzzer, OUTPUT);

  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  //wait 1 s for the Serial communication to start
  delay(1000);

  //call the two functions used to connect connect to wifi
  //and setup the server
  connectToWiFi();
  setupServer();

  //wait 4 s for the server to start
  delay(4000);
}
```

The **connectToWiFi()** function still remains the same. In the **setupServer()** function we will need a handler that will send the web page to the user and handlers for playing the notes. The most important instruction is to initialise the server using **server.begin()**.

Code 29.4.3 Set up the server

```
void setupServer()
{
  //the method "server.on()" is to call a function when
  //the user access the location
  //the default location is "/"
  server.on("/", htmlIndex);
  server.on("/c_note", c_note);
  server.on("/cs_note", cs_note);
  server.on("/d_note", d_note);
  server.on("/ds_note", ds_note);
  server.on("/e_note", e_note);
  server.on("/f_note", f_note);
  server.on("/fs_note", fs_note);
  server.on("/g_note", g_note);
  server.on("/gs_note", gs_note);
  server.on("/a_note", a_note);
  server.on("/as_note", as_note);
  server.on("/b_note", b_note);
  server.on("/off", off);

  //start the server
  server.begin();

  //print in Serial Monitor that the HTTP server is started
  Serial.println("HTTP server started");
}
```

The job of the **htmlIndex()** function is to send the HTML page to the user.

Code 29.4.4 Send the page to the user

```
void htmlIndex()
{
  //send the page to the user
  server.send(200, "text/html", page);
}
```

The next part is for the functions that will play the notes. These functions are basically the same, only the note played is changed.

Code 29.4.5 Play the notes on the buzzer

```
void c_note()
{
  //play the note on the buzzer
  tone(buzzer, c6);
  delay(10);
  server.send(200,"text/html","ok");
}

void cs_note()
{
  //play a note on the buzzer
  tone(buzzer, cs6);
  delay(10);
  server.send(200,"text/html","ok");
}

void d_note()
{
  //play a note on the buzzer
  tone(buzzer, d6);
  delay(10);
  server.send(200,"text/html","ok");
}
```

We also need a function that will stop the buzzer. This function will be called every time any key of the piano will be released.

Code 29.4.6 Stop the buzzer

```
void off()
{
  //turn off the buzzer
  noTone(buzzer);
  delay(10);
  server.send(200,"text/html","ok");
}
```

The last function is **loop()** where we we will listen to any incoming requests.

Code 29.4.7 The loop() function

```
void loop()
{
  //the method below is used to manage the incoming request
  //from the user
  server.handleClient();
}
```

The HTML body is very simple, it only has two rows. On the first row we have a paragraph that shows the "Piano style" text. The next row is for the keyboard of the piano which is composed of a single octave because we want to play the notes using our computer's keyboard. All the blocks for the keys of the piano are inside a **div** block, which has attached some classes defined in Bootstrap. Inside this **div** are 12 blocks for the 12 keys of the piano. Each block has a specific id and has a class attached (**whiteKey** or **blackKey**) and contains a paragraph that deals with the text that will be displayed on that key. These paragraphs also have a class that will arrange the text on the keys (**whiteKeyText** or **blackKeyText**).

Code 29.4.8 The body of the HTML page

```
    <body>
    <div class='container-fluid'>
        <div class='row'>
            <div class='col-sm-12 col-xs-12 text-center'>
                <p style='color: blue; font-size: 7vh'>Piano style</p>
            </div>
        </div>
        <div class='row'>
            <div class='col-sm-12 col-xs-12 d-flex justify-content-center'>
                <div id='c1' class='whiteKey'> <p class='whiteKeyText'>C6</p> </div>
                <div id='c2' class='blackKey'> <p class='blackKeyText'>CS6</p> </div>
                <div id='c3' class='whiteKey'> <p class='whiteKeyText'>D6</p> </div>
                <div id='c4' class='blackKey'> <p class='blackKeyText'>DS6</p> </div>
                <div id='c5' class='whiteKey'> <p class='whiteKeyText'>E6</p> </div>
                <div id='c6' class='whiteKey'> <p class='whiteKeyText'>F6</p> </div>
                <div id='c7' class='blackKey'> <p class='blackKeyText'>FS6</p> </div>
                <div id='c8' class='whiteKey'> <p class='whiteKeyText'>G6</p> </div>
                <div id='c9' class='blackKey'> <p class='blackKeyText'>GS6</p> </div>
                <div id='c10' class='whiteKey'> <p class='whiteKeyText'>A6</p> </div>
                <div id='c11' class='blackKey'> <p class='blackKeyText'>AS6</p> </div>
                <div id='c12' class='whiteKey'> <p class='whiteKeyText'>B6</p> </div>
            </div>
        </div>
    </div>
    </body>
```

The keys of the piano are created using CSS. There are two classes for this, one for the white keys and the other one for the black keys. Each class has a relative position, a solid black border, specific border width and radius, centered text, specific width and height and color. In addition to these, the **blackKey** class has a **0** top, so it can be on the top of the block, **1** z-index and negative left margin, so it will be on top of the white keys.

Code 29.4.9 CSS classes for the keys

```
.whiteKey
{
    position: relative;
    border-style: solid;
    border-color: #000000;
    border-width: 0.42vh 0.21vh 0.42vh 0.21vh;
    border-radius: 0.84vh 0.84vh 0.84vh 0.84vh;
    text-align: center;
    width: 6.32vh;
    height: 31.61vh;
    background: #FFFFFF;
}

.blackKey
{
    position: relative;
    border-style: solid;
    border-color: #000000;
    border-width: 0.42vh 0.21vh 0.42vh 0.21vh;
    border-radius: 0 0 0.84vh 0.84vh;
    text-align: center;
    top: 0;
    background: #000000;
    z-index: 1;
    width: 4.21vh;
    height: 21.07vh;
    margin: 0 0 0 -2.1vh;
}
```

Now, for the text that will be displayed on the keys, we have to create two CSS classes, one for the text on the white keys and one for the text on the black keys. Here we have to put a top margin, so the text will be on the bottom of the keys, a color (black for the white keys and white for the black keys) and the size of the font.

Code 29.4.10 CSS classes for the text on the keys

```
.whiteKeyText
{
    margin-top: 27.5vh;
    color: black;
    font-size: 2.3vh;
}

.blackKeyText
{
    margin-top: 17.91vh;
    color: white;
    font-size: 1.8vh;
}
```

The first and the last keys of the piano are a little bit more special, because the left, respectively the right, border of those keys should be more thickened (the border will be doubled).

Code 29.4.11 Modify the border on the first and last keys

```
#c1
{
    border-left-width: 0.42vh;
}

#c12
{
    border-right-width: 0.42vh;
}
```

Also, we combined the two classes for the white and black keys, with a negative left margin, so the black keys will be on top of the white keys. At the end, we have the **div** selector, which selects all **<div>** elements and makes them unselectable, so that when we have a long click, or a long touch, on a key, the text will be unselectable.

Code 29.4.12 Bind keys and make text unselectable

```
.blackKey + .whiteKey
{
    margin-left: -2.1vh;
}

div
{
    webkit-user-select: none; /* Safari 3.1+ */
    moz-user-select: none; /* Firefox 2+ */
    ms-user-select: none; /* IE 10+ */
    user-select: none; /* Standard syntax */
}
```

Now let's talk about the most complex part, which is JavaScript, where we will send data to server when a key is pressed, so that a specific function will be called and the buzzer will play a note or will stop.

The first part is created so that we can play on the piano using a mouse or a touch capable display. For that, we will bind, using **.on** method, the **mousedown** and **touchstart** events, and when any of those events will be triggered, a function will execute. This function contains an instruction, **.preventDefault()**, which prevents the **mousedown** event to trigger first on the touch capable devices. In this function you can find an **ajax** call that will send a message to the server, which will execute a function that will play a note, depending on what key of the piano's keyboard was pressed, and a call of the **.css** method that will change the background color of the current key pressed. We have 12 such bindings, one for each key of the piano, with small differences between, at the ajax call, because we have to play different notes on the buzzer, and at the color of the background.

Code 29.4.13 Send data to server and change color of keys at pressing

```
$('#c1').on('mousedown touchstart', function(e){
    e.preventDefault();
    $.ajax('/c_note');
    $('#c1').css('background', '#D8D5D4');
});

$('#c2').on('mousedown touchstart', function(e){
    e.preventDefault();
    $.ajax('/cs_note');
    $('#c2').css('background', '#585757');
});

$('#c3').on('mousedown touchstart', function(e){
    e.preventDefault();
    $.ajax('/d_note');
    $('#c3').css('background', '#D8D5D4');
});

$('#c4').on('mousedown touchstart', function(e){
    e.preventDefault();
    $.ajax('/ds_note');
    $('#c4').css('background', '#585757');
});

$('#c5').on('mousedown touchstart', function(e){
    e.preventDefault();
    $.ajax('/e_note');
    $('#c5').css('background', '#D8D5D4');
});
```

Now, when we stop pressing on a key, we have to send some data to the server and call a function to stop the buzzer, and, also, change the color to the initial one.

Code 29.4.14 Stop the buzzer when releasing a key

```
$('#c1,#c3,#c5,#c6,#c8,#c10,#c12').on('mouseup touchend', function(e){
    e.preventDefault();
    $.ajax('/off');
    $(this).css('background', 'white');
});

$('#c2,#c4,#c7,#c9,#c11').on('mouseup touchend', function(e){
    e.preventDefault();
    $.ajax('/off');
    $(this).css('background', 'black');
});
```

Next, we will attach to the white keys and black keys the **.mouseout** event that will occur when the mouse pointer leaves the selected elements. The color of the keys will be changed back to the initial ones and a function will be called to stop the buzzer.

Code 29.4.15 Stop the buzzer when the mouse pointer leaves the keys

```
$('#c1,#c3,#c5,#c6,#c8,#c10,#c12').mouseout(function(){
    $(this).css('background', 'white');
    $.ajax('/off');
});

$('#c2,#c4,#c7,#c9,#c11').mouseout(function(){
    $(this).css('background', 'black');
    $.ajax('/off');
});
```

When the mouse pointer enters a key, the **mouseover** event will trigger and the color of its background will change.

Code 29.4.16 Change the keys color when hover the mouse over them

```
$('#c1').mouseover(function(){
    $(this).css('background', '#E7E0DF');
});
$('#c3').mouseover(function(){
    $(this).css('background', '#E7E0DF');
});
$('#c5').mouseover(function(){
    $(this).css('background', '#E7E0DF');
});
$('#c6').mouseover(function(){
    $(this).css('background', '#E7E0DF');
});
$('#c8').mouseover(function(){
    $(this).css('background', '#E7E0DF');
});
$('#c10').mouseover(function(){
    $(this).css('background', '#E7E0DF');
});
$('#c12').mouseover(function(){
    $(this).css('background', '#E7E0DF');
});
```

Code 29.4.17 Change the keys color when hover the mouse over them

```
$('#c2').mouseover(function(){
    $(this).css('background', '#464646');
});
$('#c4').mouseover(function(){
    $(this).css('background', '#464646');
});
$('#c7').mouseover(function(){
    $(this).css('background', '#464646');
});
$('#c9').mouseover(function(){
    $(this).css('background', '#464646');
});
$('#c11').mouseover(function(){
    $(this).css('background', '#464646');
});
```

For the second part, which is created so that we can play on the piano using our computer's keyboard, we need two events, **keydown** and **keyup**, which are attached to the HTML body and will be triggered when a key on the keyboard is pressed, and, respectively, released. Also, we need a variable initialised with **true**, which will become **false** when a key is pressed and back to **true** when the key is released. This variable will help us with long key presses.

When the **keydown** method will be triggered, we will use a variable, that stores the current key pressed, and use a **switch** statement to send data to the server and call the correct function to play the note on the buzzer. Along with the **ajax** call, we will change the color of the key's background so that we will know what key of the piano was pressed.

Code 29.4.18 Keydown event

```
var action = true;
        $('body').on('keydown', function(e){
            if (action == true){
              action = false;
              var key = e.which;
              switch(key)
              {
                  case 90:
                      $.ajax('/c_note');
                      $('#c1').css('background', '#D8D5D4');
                      break;
                  case 83:
                      $.ajax('/cs_note');
                      $('#c2').css('background', '#585757');
                      break;
                  case 88:
                      $.ajax('/d_note');
                      $('#c3').css('background', '#D8D5D4');
                      break;
                  case 68:
                      $.ajax('/ds_note');
                      $('#c4').css('background', '#585757');
                      break;
                  case 67:
                      $.ajax('/e_note');
                      $('#c5').css('background', '#D8D5D4');
                      break;
                  case 86:
                      $.ajax('/f_note');
                      $('#c6').css('background', '#D8D5D4');
                      break;
                  case 71:
                      $.ajax('/fs_note');
                      $('#c7').css('background', '#585757');
                      break;
                  case 66:
                      $.ajax('/g_note');
                      $('#c8').css('background', '#D8D5D4');
                      break;
                  case 72:
                      $.ajax('/gs_note');
                      $('#c9').css('background', '#585757');
                      break;
                  case 78:
                      $.ajax('/a_note');
                      $('#c10').css('background', '#D8D5D4');
                      break;
                  case 74:
                      $.ajax('/as_note');
                      $('#c11').css('background', '#585757');
                      break;
                  case 77:
                      $.ajax('/b_note');
                      $('#c12').css('background', '#D8D5D4');
                      break;
              }
            }
        });
```

On the computer's keyboard, each key has a specific code and you can find the ones used in this code in a table below:

Key	Code
Z	90
S	83
X	88
D	68
C	67
V	86
G	71
B	66
H	72
N	78
J	74
M	77

When the **keyup** event will be triggered, a function will modify the **action** variable and make it **true**. In the **key** variable we will store the current digit pressed and, depending on its value, we have two if statements: one for the white keys, and one for the black keys, that will call the function to stop the buzzer and change the background color to the initial one.

Code 29.4.19 Keyup event

```
$('body').on('keyup', function(e){
   action = true;
   var key = e.which;
   if(key==90 || key==88 || key==67 || key==86 || key==66 ||key==78 ||key==77)
   {
       $.ajax('/off');
       $('#c1,#c3,#c5,#c6,#c8,#c10,#c12').css('background', 'white');
   }

   if(key==83 || key==68 || key==71 || key==72 || key==74)
   {
       $.ajax('/off');
       $('#c2,#c4,#c7,#c9,#c11').css('background', 'black');
   }
});
```

30. Lesson 25: Piano with 7 octaves

30.1 Overview

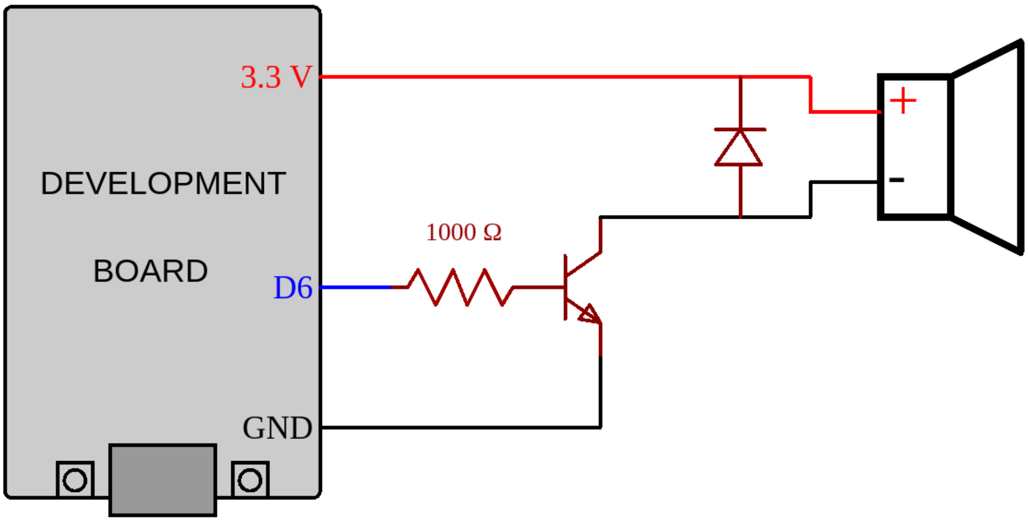
In this lesson you will learn how to create a full piano and play all the notes from the seven octaves. The piano will be created using CSS and the keyboard of the piano will be responsive.

30.2 Components required

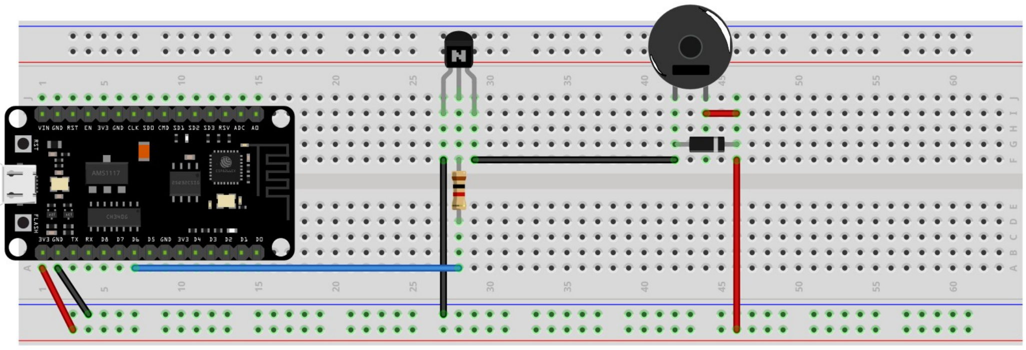
- Development board;
- Breadboard 830p;
- 1 x passive buzzer;
- Diode;
- Transistor;
- 1 x 1000 Ω resistor;
- 7 x male-to-male jumper wires;
- Micro USB – Type A USB cable;

30.3 Connections

Below you can find the schematic:



Also, you can find below a visual representation of the project:



30.4 Code

The code for this lesson is an improved version of the one from the previous lesson. This time we will have additional buttons that will change the octave. We need this because the space on the computer's keyboard is limited. You can find the code in the "**Lesson 25: Piano with 7 octaves**" folder.

The code starts with the declaration of the variables that will store the frequencies for all 7 octaves, then we need to include the **ESP8266WebServer** library, declare two variables that will store the credentials of the wireless network that the board will connect to, declare the pin used by the buzzer, declare the **octave** variable that will store the current octave selected and, finally, we need an instance of the **ESP8266WebServer** class. The variables that stores the frequencies will be put in arrays, one array for each octave, and then these arrays will be included in another array.

Code 30.4.1 Store the notes in arrays

```
//create arrays with the notes for each octave (in order)
int octave1[] = {c1, cs1, d1, ds1, e1, f1, fs1, g1, gs1, a1, as1, b1};
int octave2[] = {c2, cs2, d2, ds2, e2, f2, fs2, g2, gs2, a2, as2, b2};
int octave3[] = {c3, cs3, d3, ds3, e3, f3, fs3, g3, gs3, a3, as3, b3};
int octave4[] = {c4, cs4, d4, ds4, e4, f4, fs4, g4, gs4, a4, as4, b4};
int octave5[] = {c5, cs5, d5, ds5, e5, f5, fs5, g5, gs5, a5, as5, b5};
int octave6[] = {c6, cs6, d6, ds6, e6, f6, fs6, g6, gs6, a6, as6, b6};
int octave7[] = {c7, cs7, d7, ds7, e7, f7, fs7, g7, gs7, a7, as7, b7};


int *octave_array[] = {octave1, octave2, octave3, octave4, octave5, octave6, octave7};
```

The **setup()**, **loop()**, **connectToWiFi()**, **htmlIndex()**, **off()** functions are unchanged, so we will not discuss about them, but you can look in the full code to remember them. The **setupServer()** function has two additional handlers, **set_octave** and **refresh_function**, which reads from the user the current selected octave and, when the default location is reloaded, the octave value will be sent to the user so that the content of the page will not be changed.

Code 30.4.2 Read the octave from the user

```
void set_octave()
{
  //store the current octave
  octave = server.arg("octave").toInt();
  Serial.println(octave);
  server.send(200,"text/html","ok");
}
```

Code 30.4.3 Send the value of octave to the user

```
void refresh_function()
{
  //send the last known octave to the user
  //we will use this so that even if we will
  //refresh the page, the content will not
  //change
  char c = octave + '0';
  String a = String(c);
  server.send(200, "application/javascript", a);
}
```

The functions created for notes are almost the same, the only thing that changed is the second parameter of the **tone** function, which is the frequency at which the buzzer resonates. We retrieve this parameter from the two dimensional array in which there are stored the frequencies corresponding to each musical octave (line) and key (collumn).

Code 30.4.4 Functions created for playing the notes

```
void c_note()
{
  //play a note on the buzzer
  tone(buzzer, octave_array[octave-1][0]);
  Serial.println(octave_array[octave-1][0]);
  delay(10);
  server.send(200,"text/html","ok");
}

void cs_note()
{
  //play a note on the buzzer
  tone(buzzer, octave_array[octave-1][1]);
  Serial.println(octave_array[octave-1][1]);
  delay(10);
  server.send(200,"text/html","ok");
}

void d_note()
{
  //play a note on the buzzer
  tone(buzzer, octave_array[octave-1][2]);
  Serial.println(octave_array[octave-1][2]);
  delay(10);
  server.send(200,"text/html","ok");
}

void ds_note()
{
  //play a note on the buzzer
  tone(buzzer, octave_array[octave-1][3]);
  Serial.println(octave_array[octave-1][3]);
  delay(10);
  server.send(200,"text/html","ok");
}
```

Now, let's talk about the HTML page, which has a couple of new things added. The HTML body still has the piano, but has two new rows, one that display a text to inform the user what to do, and the other one contains seven bootstrap-buttons that will change the octave.

Code 30.4.5 HTML body

```
<div class='row'>
   <div class='col-sm-12 col-xs-12 text-center' style='font-size: 5vh'>
         <p>Select the octave:</p>
   </div>
</div>
<div class='row'>
   <div class='col-sm-12 col-xs-12 text-center'>
       <button id='b1' type='button' class='btn btn-outline-primary'>First</button>
       <button id='b2' type='button' class='btn btn-outline-primary'>Second</button>
       <button id='b3' type='button' class='btn btn-outline-primary'>Third</button>
       <button id='b4' type='button' class='btn btn-outline-primary'>Fourth</button>
       <button id='b5' type='button' class='btn btn-outline-primary'>Fifth</button>
       <button id='b6' type='button' class='btn btn-outline-primary'>Sixth</button>
       <button id='b7' type='button' class='btn btn-outline-primary'>Seventh</button>
    </div>
</div>
```

In addition to the already defined CSS classes, we have added one that will override the default font-size of the buttons, and is also marked as important to be sure that will apply to the buttons.

Code 30.4.6 CSS classes

```
button
{
     font-size: 2vh !important;
}
```

In JavaScript, we need a variable to store the number of current octave and then create a function to send it to the server via an ajax call.

Code 30.4.7 Send the current octave to the sever

```
var octave;

function change_octave(){
    $.ajax({
        url: '/set_octave',
        type: 'POST',
        data: {octave: octave},
    });
}
```

For the text that will be displayed on the keys, we need a function that adds the note plus the current octave. To set the text, we are going to use the **.html** method for each key.

Code 30.4.8 Modify the text displayed on the keys

```
function keys_text(octave)
{
    $('#k1').html('C' + octave);
    $('#k2').html('CS' + octave);
    $('#k3').html('D' + octave);
    $('#k4').html('DS' + octave);
    $('#k5').html('E' + octave);
    $('#k6').html('F' + octave);
    $('#k7').html('FS' + octave);
    $('#k8').html('G' + octave);
    $('#k9').html('GS' + octave);
    $('#k10').html('A' + octave);
    $('#k11').html('AS' + octave);
    $('#k12').html('B' + octave);
};
```

After we refresh the page, or we access it for the first time, after it is fully loaded, a function will be called and get from the server the last octave returned and modify the text displayed on them by calling the function **keys_text()** that has as argument the octave received from the server.

Code 30.4.9 Reinstate the octave

```
$(document).ready(function(){
    $.get('/refresh', function(octave){
        keys_text(octave);
    });
});
```

When a click is registered on any of the defined buttons, a function will be triggered, and depending on which button was pressed, the value of the octave will be changed, the **keys_text()** will be called to update the text on the keys and the **change_octave()** function will be called to send the new value of **octave** to the server.

Code 30.4.10 Change the octave

```
$('button').on('click', function(){
    octave = this.id;
    keys_text(octave);
    change_octave();
});
```

The rest of the JavaScript is already detailed in the last lesson, so go check that for a better understanding of the code.

31. Lesson 26: Shift Register

31.1 Overview

In this lesson you will learn how to use a shift register to control multiple LEDs from an LED bar.

31.2 Components required

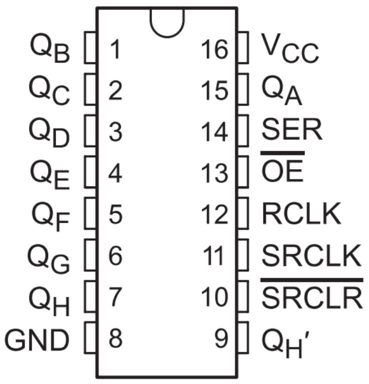
- Development board;
- Breadboard 830p;
- Micro USB – Type A USB cable;
- 74HC595 Shift Register
- 22 x male-male jumper wires;
- 10 x 150 Ω resistors;
- 10 Segment LED Bar

31.3 Component Introduction

74HC595

The 74HC595 Serial to Parallel Converter is a simple 8-bit shift register, which has 8 outputs and three inputs that we are using to feed data into it a bit at a time. A shift register consists of several single bit **D-Type Data Latches**, one for each data bit, either a logic **0** or a **1**, connected together in a serial type daisy-chain arrangement so that the output from one data latch becomes the input of the next latch and so on. Basically, this shift register is a device that allows additional inputs or outputs to be added to a microcontroller by converting data between parallel and serial formats, using only three pins of the development board.

To set the value of each memory location on or off, we feed in the data using the **Data** and **Clock** pins of the chip. Below is the schematic of the chip:



8 – GND	Ground Pin
16 - VCC	Power Pin
15 - Q_A	Output
1-7 – Q_B-Q_H	Output
9 - $Q_{H'}$	Extension pin (usually used for connecting an additional shift register)
14 – SER	Data Pin
12 – RCLK	Latch Pin
11 – SRCLK	Clock Pin
13 – OE	Output Enable
10 – SRCLR	Master Reset

The **Clock** pin needs to receive eight pulses. At each pulse, if the data pin is high, then a **1** gets pushed into the shift register, otherwise, a **0**. When all eight pulses have been received, enabling the **Latch** pin copies the eight values to the latch register.

The **OE** pin, which is used to enable or disable the outputs all at once. You could attach this to a PWM pin and control, for example, the brightness of the LEDs. This pin is active low, so we connect it to GND.

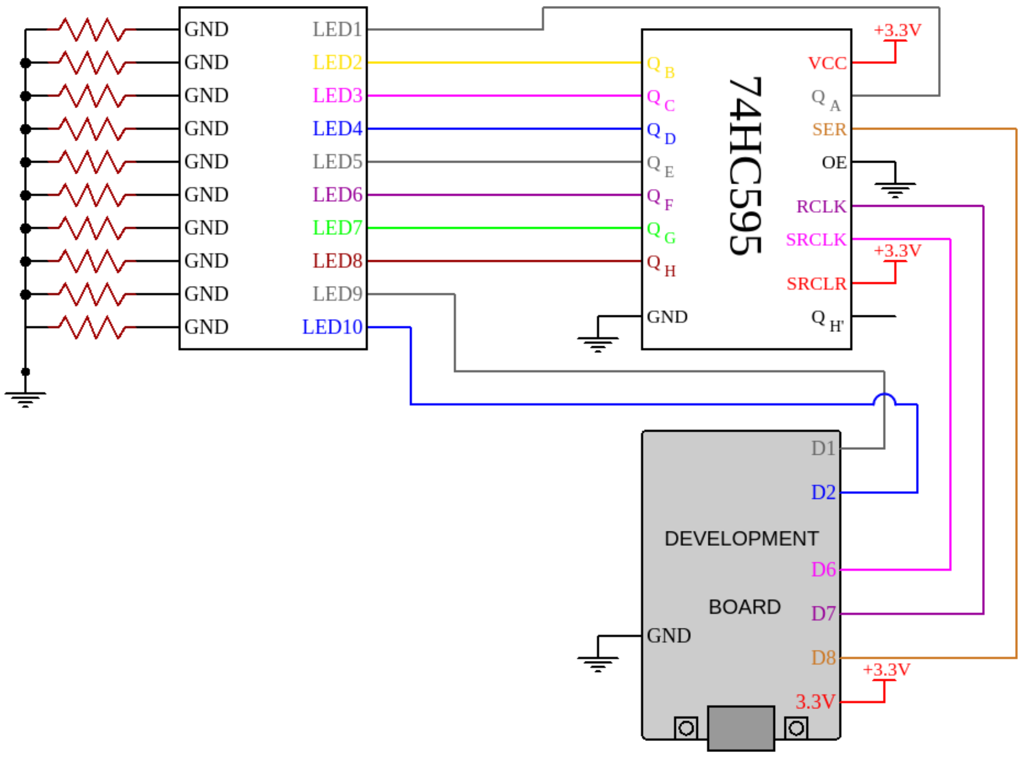
LED bar

A 10 Segment LED Bar is composed of 10 individual LEDs housed together, each with an individual anode and cathode connection. They are perfect for prototyping because of their compact footprint and simple hookup. You can identify the anode side by the left corner, which is rounded, and the others form a right angle.

31.4 Connections

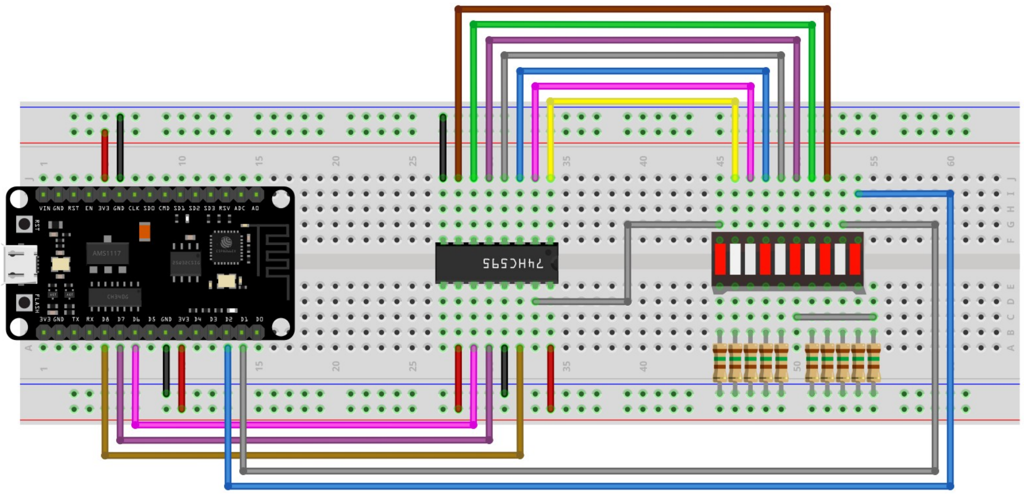
Because we are using an 8-bit shift register, we only have 8 outputs on the chip, and our LED bar has 10 segments, so we have to use 2 additional pins from our development board to connect all the individual LEDs.

Below, you can find the schematic:



The scheme without the symbols for GND and +3.3V can be found in the **"Lesson 26: Shift Register"** folder.

Below, you can find a visual representation of the connections:



31.5 Code

The principle of this code is similar with the one from lesson **Blink an LED**, but this time we have to control 10 LEDs, two of them connected to the development board, and eight connected to the shift register. The code can be found in the folder **"Lesson 26: Shift Register"**.

As you previously saw in the scheme, we need three pins from the development board, used for **Data**, **Latch** and **Clock**, to connect our shift register. At the beginning of the code we have to declare these pins used by the shift register and, moreover, we need to declare the pins used by the two LEDs connected to the development board.

Code 31.5.1 Variables declaration

```
//declare the pins used by the shift register
const int dataPin = D8;
const int latchPin = D7;
const int clockPin = D6;

//declare two pins used by the remaining LEDs
const int led_9 = D1;
const int led_10 = D2;
```

In the **setup()** function we will start a serial communication with the computer, then set the pins used by the two LEDs and shift register as **OUTPUT**.

Code 31.5.2 The setup() function

```
void setup()
{
  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  //set the pins used by the two LEDs to output
  pinMode(D1, OUTPUT);
  pinMode(D2, OUTPUT);

  //set pins to output so you can control the shift register
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}
```

In the **loop()** function we will call a function that will turn on each LED, wait 500 ms and then call another function that will turn on the LEDs and create a charging effect. After this we will wait another 500 ms before the next run of the **loop()** function.

Code 31.5.3 The loop() function

```
void loop()
{
  //turn On each LED and wait 1 s
  individual();

  //wait 500 ms before the next effect
  delay(500);

  //turn On the LED and show a charging effect
  charging();

  //wait 500 ms
  delay(500);
}
```

In the **individual()** function we will declare eight byte variables because we are working with an 8-bit shift register. This shift register has 8 outputs, and to make an output **HIGH**, we need to send an **1** on that specific **OUTPUT**, and to make it **LOW**, we will send a **0**. So, we have eight byte variables and each variable has only one bit set to **HIGH**, and this bit corresponds to our LED on the ledbar. Keep in mind that we are starting from the MSB (Most Significant Bit), which is the rightmost bit, and go to the LSB (Least Significant Bit), which is the leftmost bit. Therefore, the first LED will turn on when we will send **first_led** = **0b00000001**, and the eight LED will turn on when we will send **eight_led** = **0b10000000**. These eight variables will be stored in a byte array, so we can use a **for** loop to turn each LED one at a time.

Code 31.5.4 Declaration of byte variables that will be sent later to the shift register

```
//declare byte data types to be sent to the
//shift register
//a byte data stores 8 bits that corresponds
//to the 8 outputs of the shift register
//"0" represents LOW state, and "1" HIGH
byte first_led = 1;    //equivalent: 0b00000001
byte second_led = 2;   //equivalent: 0b00000010
byte third_led = 4;    //equivalent: 0b00000100
byte fourth_led = 8;   //equivalent: 0b00001000
byte fifth_led = 16;   //equivalent: 0b00010000
byte sixth_led = 32;   //equivalent: 0b00100000
byte seventh_led = 64; //equivalent: 0b01000000
byte eighth_led = 128; //equivalent: 0b10000000

//store the 8 byte variables into an array
byte led[] =
{
  first_led, second_led, third_led, fourth_led,
  fifth_led, sixth_led, seventh_led, eighth_led
};
```

The principale of the shift register is: when the **clockPin** goes from **LOW** to **HIGH**, the shift register reads the state of the **dataPin**. As the data gets shifted in it, it is saved in the internal memory register. When the **latchPin** goes from **LOW** to **HIGH**, the sent data gets moved into the latch register, the output pins.

We have to use **byte** variables to send 8 bits at a time to the shift register. To send the data to the shift register we need the following command:

```
digitalWrite(latchPin, 0);
shiftOut(dataPin, clockPin, bitorder, value);
digitalWrite(latchPin, 1);
```

The **shiftOut()** function shifts out a byte of data one bit at a time. Each bit is written in to a data pin, after which a clock pin is pulsed to indicate that the bit is available. The **bitorder** argument indicates the order to shift out the bits, **MSBFIRST** (Most Significant Bit First) or **LSBFIRST** (Least Significant Bit First), and the **value** is the **byte** to shift out.

To update the output pins we have to set the **latchPin** to **HIGH**, but it is not enough to set **latchPin** to **HIGH**, data transfer happens on transition from **LOW** to **HIGH** (rising edge).

These instructions will be put in a **for** loop that will turn on the eight LEDs with a delay of 1 s.

Code 31.5.5 Turn on all the LEDs connected to the shift register, one at a time

```
//using the for loop we are going to send each byte
//data to the shift register and turn on the individual
//LEDs
for(int i = 0; i < 8; i++)
{
  //set the clock pin LOW before shiftOut() call
  digitalWrite(clockPin, LOW);

  //set latchPin to LOW so the LEDs don't flash while
  //sending in bits
  digitalWrite(latchPin, 0);

  //shift the 8 bits out with Most Significant Bit First
  shiftOut(dataPin, clockPin, MSBFIRST, led[i]);

  //copy the values to the latch register
  digitalWrite(latchPin, 1);

  //wait 1 s before the next instruction
  delay(1000);
}
```

We turned on all the LEDs connected to the shift register, but we have another two connected directly to the developemtn board. Before turning these two LEDs on, we have to send a **0** to the shift register, because it will keep the last state and the eight LEDs will remain on.

Code 31.5.6 Turn on the rest LEDs

```
//send a 0 byte value to turn all the LEDs off
//before any other instruction
digitalWrite(latchPin, 0);
shiftOut(dataPin, clockPin, MSBFIRST, 0);
digitalWrite(latchPin, 1);

//turn On the nineth LED of the ledbar
//for 1 s
digitalWrite(led_9, HIGH);
delay(1000);
digitalWrite(led_9, LOW);

//turn On the tenth LED of the ledbar
//for 1 s
digitalWrite(led_10, HIGH);
delay(1000);
digitalWrite(led_10, LOW);
```

The **charging()** function is similar with the **individual()**, but this time we are not going to use separate byte variables, we will put them directly into an array and in the **for** loop we are going to send the byte variables each 500 ms. If you look at the array, you can notice that the variables create stairs and that will give us that charging effect.

Code 31.5.7 The charging() function

```
//declare an byte array to store the states for the
//8 LEDs connected to the shift register
byte charging_array[] =
{
  0b00000001,
  0b00000011,
  0b00000111,
  0b00001111,
  0b00011111,
  0b00111111,
  0b01111111,
  0b11111111
};

for(int i = 0; i < 8; i++)
{
  //set the clock pin LOW before shiftOut() call
  digitalWrite(clockPin, LOW);

  //set latchPin to LOW so the LEDs don't flash while
  //sending in bits
  digitalWrite(latchPin, 0);

  //shift the 8 bits out with Most Significant Bit First
  shiftOut(dataPin, clockPin, MSBFIRST, charging_array[i]);

  //copy the values to the latch register
  digitalWrite(latchPin, 1);

  //wait 500 ms
  delay(500);
}
```

This time we will not send a **0** to turn off all the LEDs connected to the shift register, we will turn on the last two LEDs and then turn all of them off.

Code 31.5.8 Turn on the last two LEDs and then turn all LEDs off

```
//turn On the nineth LED and wait 500 ms
digitalWrite(led_9, HIGH);
delay(500);

//turn On the tenth LED and wait 500 ms
digitalWrite(led_10, HIGH);
delay(500);

//turn Off all the LEDs of the ledbar
digitalWrite(led_9, LOW);
digitalWrite(led_10, LOW);

digitalWrite(latchPin, 0);
shiftOut(dataPin, clockPin, MSBFIRST, 0);
digitalWrite(latchPin, 1);
```

32. Lesson 27: Multiple Shift Registers

32.1 Overview

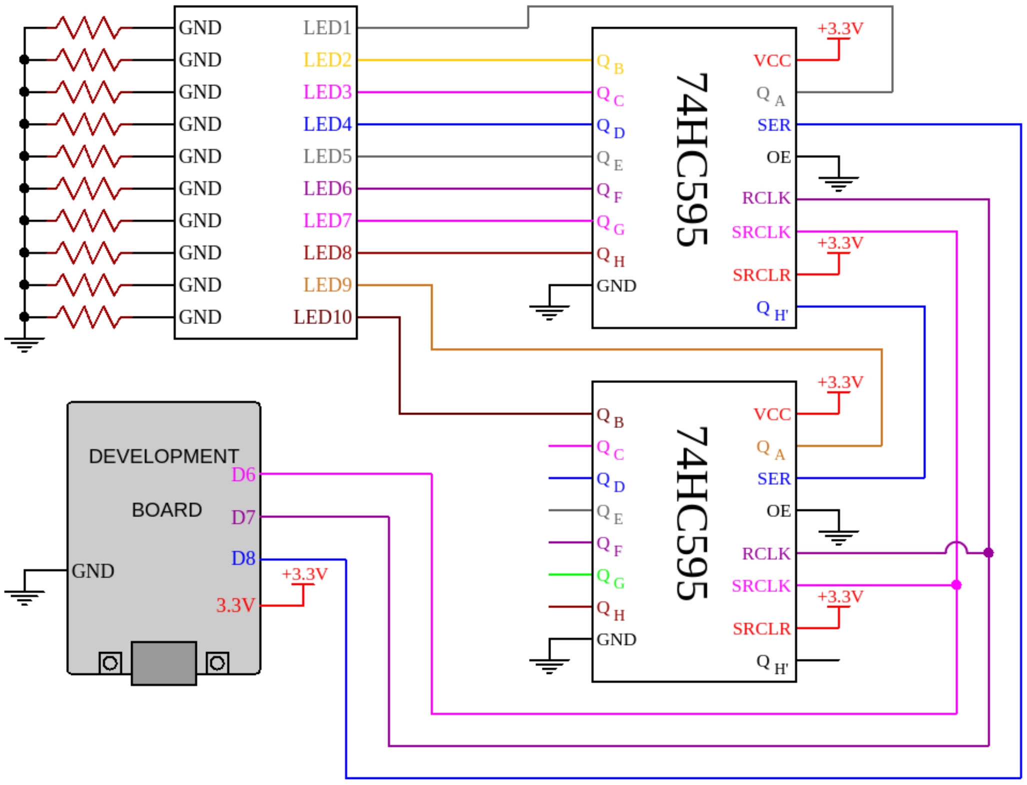
In this lesson you will learn how to connect multiple shift registers to the development board. The big advantage is that we do not use any supplimentary pins for the second shift register, we only have to connect the **dataPin** of the second shift register to **$Q_{H'}$** pin of the first shift register.

32.2 Components required

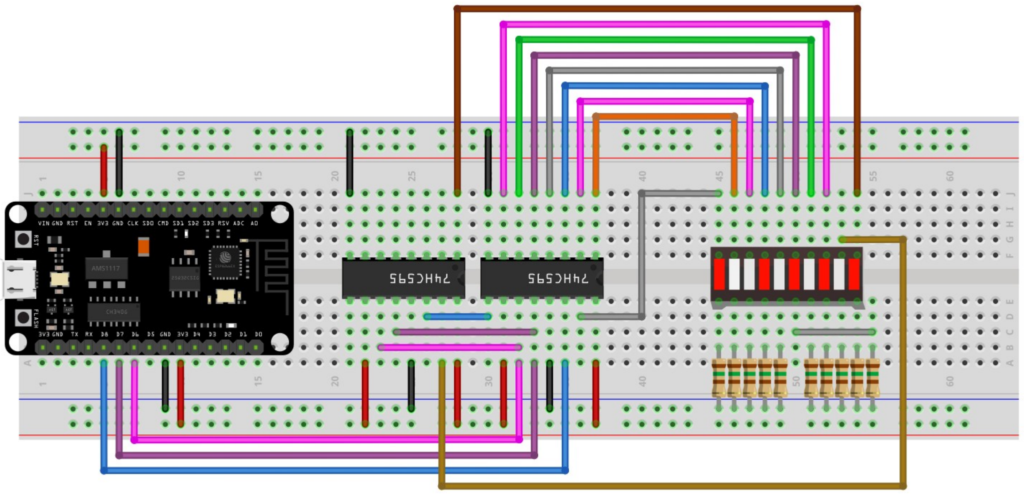
- Development board;
- Breadboard 830p;
- Micro USB – Type A USB cable;
- 2 x 74HC595 Shift Register
- 21 x male-male jumper wires;
- 10 x 150 Ω resistors;
- 10 Segment LED Bar;

32.3 Connections

Below, you can find the schematic:



Below, you can find a visual representation of the connections:



32.4 Code

The code is very similar with the one from the previous lesson. The only difference is that the two LEDs connected to the development board in the previous lesson, we will connect them to the second shift register. This code can be found in the **"Lesson 27: Multiple Shift Registers"** folder. Also, previously we used **byte** variables, which are data type represented on 8 bits (1 byte), and working with 2 shift registers, we need 16 bits (2 bytes).

The downside for the **shiftOut** function is that it can support only 8 bits, so we have to call the function twice. Doing that, we can use sepparate **byte** variables for each call, or use **uint16_t** data type and bit shifting.

Bit shifting is moving each digit in a number's binary representation left or right with a specified number of bits. The bits from the direction of the shift are lost, and **0** bits are inserted on the other end. There are two arithmetic shift operators as following:

- >> is the arithmetic right shift operator
- << is the arithmetic left shift operator

Now let's make an example with each operator:

Left shift

1 stored as 8 bit is **00000001**. Shifted to the left with one position will give as number **2**.

00000001 << 1 = 00000010

255 stored as 8 bit is **11111111**. Shifted to the left with one position will result in number **254**.

11111111 << 1 = 11111110

Number **2** stored as 8 bit is **00000010**. Shifted to the left with 2 positions (2 << 2) will result in number **8**.

00000010 << 2 = 00001000

Right shift

Number **255** stored as 8 bit is **11111111**. Shifted to the right with one position will result in number **127**.

11111111 >> 1 = 01111111

Number **64** stored as 8 bit is **01000000** and shifted with 2 positions to the right (64 >> 2) will result in number **16**.

01000000 >> 2 = 00010000

We will work with 16 bit values, so we have to use right shifting to move the first 8 bits to the right, and they will be replaced with 0, and these will be ignored. For example:

0111111100000000 >> 8 = 0000000001111111 = 01111111

0101010101010101 >> 8 = 0000000001010101 = 01010101

0101010101010101 >> 0 = 0101010101010101

The **shiftOut()** is working with 8 bit values, so the first 8 bits (from left to right) will be ignored. **1111111100000000** for the **shiftOut()** function will be **00000000**.

Before we start talking about the code, let's discuss about the **shiftOut()** function to see how it works internally.

Code 32.4.1 The shiftOut() function

```
void shiftOut(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder, uint8_t
val)
{
    uint8_t i;

    for (i = 0; i < 8; i++)  {
        if (bitOrder == LSBFIRST)
            digitalWrite(dataPin, !!(val & (1 << i)));
        else
            digitalWrite(dataPin, !!(val & (1 << (7 - i))));

        digitalWrite(clockPin, HIGH);
        digitalWrite(clockPin, LOW);
    }
}
```

As previously stated, the principle of the **shiftOut()** function is that it shifts out a byte of data one bit at a time, each bit is written in to a data pin, after which a clock pin is pulsed to indicate that the bit is available.

The shifts are done using a **for** loop and inside it there is an **if** statement that will verify if the bitorder is **LSBFIRST**, then the **digitalWrite(dataPin, !!(val & (1 << i)));** instruction will be executed, otherwise the **digitalWrite(dataPin, !!(val & (1 << (7 – i))));** instruction will be executed. Let's assume that the bitorder is **LSBFIRST**. Using the **digitalWrite()** instruction, each bit (**0** or **1**) will be send to the indicated pin. The value to be send is given by the following expression: **!!(val & (1 << i))** (because the bitorder is **LSBFIRST**). This expression is explained below:

- **i** is the bit number
- **1** is **00000001** in binary
- **1 << i** is **00000001** shifted left by **i** positions

- **&** is the "bitwise and operator", where **any_bit & 0** is **zero** and **any_bit & 1** is **any_bit**
- **val & (1 << i)** is **0...0 (i-th bit of val) 0...0** in binary, where the **i**-th bit of **val** is in the **i**-th position of the result
- **!!** is a double negation: it converts zero to zero and any non-zero value to one
- **!!(val & (1 << i))** is either **0** or **1**, and is exactly the i-th bit of **val**

The code starts by declaring the pins used by the shift registers and in the **setup()** function we will add an instruction to start the serial communication with the computer and, also, we will set the pins used by the shift registers as **OUTPUT**.

Code 32.4.2 Declaration and the setup() function

```
//declare the pins used by the shift register
const int dataPin = D8;
const int latchPin = D7;
const int clockPin = D6;

void setup()
{
  //start the Serial communication at 115200 baudrate
  Serial.begin(115200);

  //set pins to output so you can control the shift register
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}
```

In the **loop()** function we will call another functions at an interval of 500 ms that will light up the LEDs of the ledbar and show different effects.

Code 32.4.3 The loop() function

```
void loop()
{
  //turn On each LED and wait 1 s
  individual();

  //wait 500 ms before the next effect
  delay(500);

  //turn On the LEDs and show a charging effect
  charging();

  //wait 500 ms
  delay(500);

  another_effect();

  //wait 500 ms
  delay(500);

  barcode();

  //wait 500 ms
  delay(500);
}
```

The first function will turn on each LED at an interval of 1 s. This function is similar with the one from the previous lesson, but this time we are working with two shift registers so we need to make two calls for the **shiftOut()** function to send, first, the high byte (which deals with the LEDs from the second shift register) and then the low byte (which controls the LEDs connected to the first shift register). This time, instead of using an array to store ten 16 bits values, we will use a variable that has the initial value of **0b0000000000000001** (meaning that only the first LED will light up) and then we will use left shifting for change its value and turn on each LED.

Code 32.4.4 Turn on each LED

```
void individual()
{
  //declaration of a 16 bit variable with the most
  //significant bit set to HIGH
  uint16_t led = 0b0000000000000001;

  //using the for loop we are going to send each byte
  //data to the shift register and turn on the individual
  //LEDs
  for(int i = 0; i < 10; i++)
  {
    byte high_byte = led >> 8;
    byte low_byte = led;

    //set the clock pin LOW before shiftOut() call
    digitalWrite(clockPin, LOW);

    //set latchPin to LOW so the LEDs don't flash while
    //sending in bits
    digitalWrite(latchPin, 0);

    //shift out the high byte
    shiftOut(dataPin, clockPin, MSBFIRST, high_byte);
    //shift out the low byte
    shiftOut(dataPin, clockPin, MSBFIRST, low_byte);

    //copy the values to the latch register
    digitalWrite(latchPin, 1);

    //left shift with one position
    led = led << 1;

    //wait 1 s before the next instruction
    delay(1000);
  }

  //send a 0 byte value to turn all the LEDs off
  //before any other instruction
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, 0);
  digitalWrite(latchPin, 1);
}
```

The charging effect is similar with the one form the previous lesson, but this time we will use an array that stores ten **uint16_t** values, and two calls for the **shiftOut()** function to send the values.

Code 32.4.5 Turn on the LEDs and show a charging effect

```
void charging()
{
  //declare an 16 bit array to store the states for the
  //10 LEDs connected to the shift registers
  uint16_t charging_array[] =
  {
    0b0000000000000001,
    0b0000000000000011,
    0b0000000000000111,
    0b0000000000001111,
    0b0000000000011111,
    0b0000000000111111,
    0b0000000001111111,
    0b0000000011111111,
    0b0000000111111111,
    0b0000001111111111
  };

  for(int i = 0; i < 10; i++)
  {
    byte high_byte = charging_array[i] >> 8;
    byte low_byte = charging_array[i];

    //set the clock pin LOW before shiftOut() call
    digitalWrite(clockPin, LOW);

    //set latchPin to LOW so the LEDs don't flash while
    //sending in bits
    digitalWrite(latchPin, 0);

    //the if statement is used to differentiate the shift registers
    //shift out the high byte
    shiftOut(dataPin, clockPin, MSBFIRST, high_byte);
    //shift out the low byte
    shiftOut(dataPin, clockPin, MSBFIRST, low_byte);

    //copy the values to the latch register
    digitalWrite(latchPin, 1);

    //wait 500 ms
    delay(500);
  }

  //send a 0 byte value to turn all the LEDs off
  //before any other instruction
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, 0);
  digitalWrite(latchPin, 1);
}
```

Another function created in this code is another_effect() and in this function we have an **uint16_t** array with 8 values and, using a **for** loop, we will send these values to the shift registers.

Code 32.4.6 Display another effect

```
void another_effect()
{
  uint16_t effect[] =
  {
    0b0000001000000001,
    0b0000000100000010,
    0b0000000010000100,
    0b0000000001001000,
    0b0000000000110000,
    0b0000000001001000,
    0b0000000010000100,
    0b0000000100000010,
    0b0000001000000001
  };

  for(int i = 0; i < 9; i++)
  {
    byte high_byte = effect[i] >> 8;
    byte low_byte = effect[i];

    //set the clock pin LOW before shiftOut() call
    digitalWrite(clockPin, LOW);

    //set latchPin to LOW so the LEDs don't flash while
    //sending in bits
    digitalWrite(latchPin, 0);

    //shift out the high byte
    shiftOut(dataPin, clockPin, MSBFIRST, high_byte);
    //shift out the low byte
    shiftOut(dataPin, clockPin, MSBFIRST, low_byte);

    //copy the values to the latch register
    digitalWrite(latchPin, 1);

    //wait 500 ms
    delay(500);
  }

  //send a 0 byte value to turn all the LEDs off
  //before any other instruction
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, 0);
  digitalWrite(latchPin, 1);
}
```

The last effect created is the one that will turn on the LEDs like a barcode. This function contains an array with three values and in the **for** loop, that will repeat 10 times, we will play them one after another.

Code 32.4.7 Barcode effect

```
void barcode()
{
  uint16_t barcode_array[] =
  {
    0b0000001001001001,
    0b0000000100100100,
    0b0000000010010010
  };

  //repeat 10 times
  for(int i = 0; i < 10; i++)
  {
    //set the clock pin LOW before shiftOut() call
    digitalWrite(clockPin, LOW);

    //set latchPin to LOW so the LEDs don't flash while
    //sending in bits
    digitalWrite(latchPin, 0);
    //shift out the high byte
    shiftOut(dataPin, clockPin, MSBFIRST, barcode_array[0] >> 8);
    //shift out the low byte
    shiftOut(dataPin, clockPin, MSBFIRST, barcode_array[0]);
    //copy the values to the latch register
    digitalWrite(latchPin, 1);
    delay(200);

    digitalWrite(latchPin, 0);
    //shift out the high byte
    shiftOut(dataPin, clockPin, MSBFIRST, barcode_array[1] >> 8);
    //shift out the low byte
    shiftOut(dataPin, clockPin, MSBFIRST, barcode_array[1]);
    //copy the values to the latch register
    digitalWrite(latchPin, 1);
    delay(200);

    digitalWrite(latchPin, 0);
    //shift out the high byte
    shiftOut(dataPin, clockPin, MSBFIRST, barcode_array[2] >> 8);
    //shift out the low byte
    shiftOut(dataPin, clockPin, MSBFIRST, barcode_array[2]);
    //copy the values to the latch register
    digitalWrite(latchPin, 1);
    delay(200);
  }

  //send a 0 byte value to turn all the LEDs off
  //before any other instruction
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, 0);
  digitalWrite(latchPin, 1);
}
```

33. Lesson 28: 4 Digit 7 Segment Display

33.1 Overview

In this lesson you will learn how to use two shift registers and how to control a 4 Digit 7 Segment with Common Cathode display.

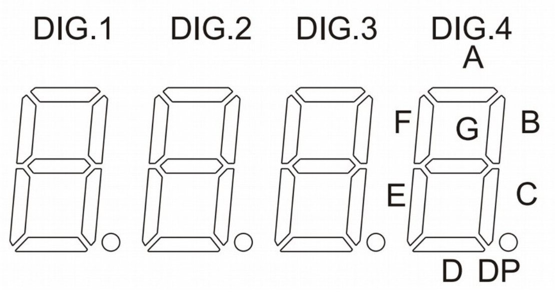
33.2 Components required

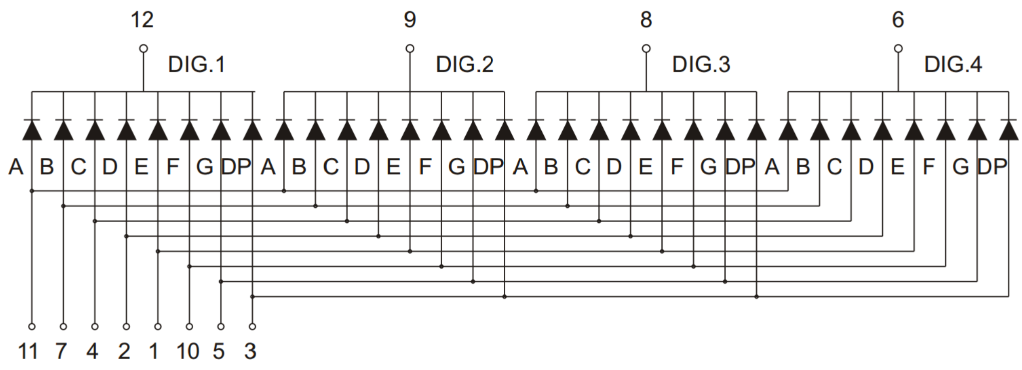
- Development board;
- Breadboard 830p;
- Micro USB – Type A USB cable;
- 2 x 74HC595 Shift Register
- 34 x male-male jumper wires;
- 4 x 150 Ω resistors;
- 4 Digit 7 Segment Common Cathode Display;

33.3 Component Introduction

4 Digit 7 Segment with Common Cathode display

This display is made from 4 digits and has 7 segments, meaning that it is made up of 7 LEDs to represent a digit from 0 to 9 when the 7 LEDs are turned on in a specific manner. Also, it has an 8 th LED for the dot (decimal point). This display is with common cathode, which means that the cathode is common for the 8 LEDs of each digit. Below you can find two schemes for the display to understand the linking between LEDs and the 12 pins.

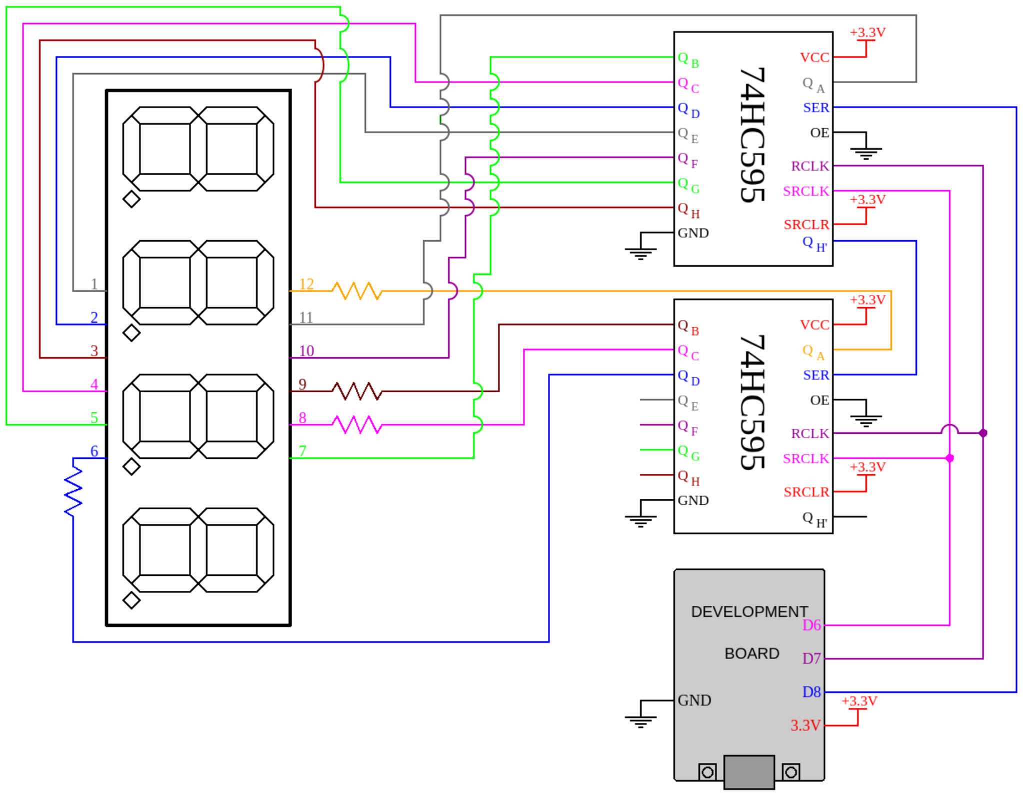



12
9
8
6
DIG.1
DIG.2
DIG.3
DIG.4
A B C D E F G DP A B C D E F G DP A B C D E F G DP A B C D E F G DP
11 7 4 2 1 10 5 3

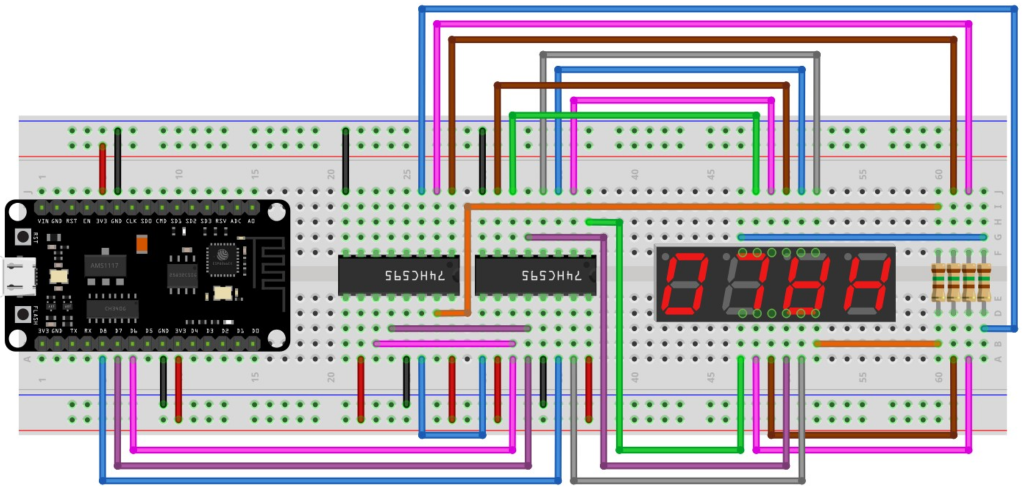
Each digit has 7 LEDs plus 1 LED for the dot, named from **A** to **G**, and **DP** for the dot. The anode for the four digits is common, and the cathode for all LEDs of each digit is common, as shown in the scheme above. We have 12 pins, 8 for the letters A – G and DP (these are anodes connected to digital pins on the development board), and we have 4 pins (also called the digit pins) for the cathode of each digit (12, 9, 8 and 6).

33.4 Connections

Below, you can find the schematic:



Below, you can find a visual representation of the connections:



33.5 Code

The code for this lesson can be found in the folder **"Lesson 28: 4 Digit 7 Segment Display"** and is similar with the previous one, because we are using two shift registers and we have to use the **shiftOut()** function to send the data to the shift registers. Because we are working with two shift registers, we need to send a 16 bit value, or two 8 bits values. So, to make the code easy to understand, we are going to connect the 8 pins, for the segments and decimal point, to the first shift register and the 4 pins for the digits to the second shift register.

The **A** (also **B** – **G** and **DP**) LEDs have common anode for all 4 digits, meaning that if we connect all 4 digit pins to the ground, and the **A** pin to 3.3V the **A** LEDs will light up on all 4 digits.

The code starts by declaring the pins used by the two shift registers and a byte variable called **digit** that will store the current digits active. Next, we have created some byte variables whose values have set to **HIGH** only one bit that corresponds to an output pin of the shift register. For example, we have the **a** variable that has the **B00000001** value. When we will send this value to the shift register, using the **shiftOut()** function with **MSBFIRST**, only the **Q_A** is set to **HIGH**. To this output is connected the **11** pin of the **4 Digit 7 Segment Display**, and this pin is for the **A** LEDs that will light up after sending the **a** value to the shift register.

Code 33.5.1 Variables declaration

```
//declare the pins used by the shift registers
const int dataPin = D8;
const int latchPin = D7;
const int clockPin = D6;

//declare a byte variable to store the current digits active
byte digit;

//create byte variables that have only one segment set to HIGH
byte a = B00000001;
byte b = B00000010;
byte c = B00000100;
byte d = B00001000;
byte e = B00010000;
byte f = B00100000;
byte g = B01000000;
byte dot = B10000000;
```

In the **setup()** function we need to set the pins used by the shift registers as OUTPUT.

Code 33.5.2 The setup() function

```
void setup()
{
  //set the pins used by the shift registers to output
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}
```

A function created for this code is **Digit()** and contains a switch statement that will update the value of the **digit** variable, depending on the value of the **x** parameter. You can notice that we give to the **digit** variable integer values, but you can find the value written as 8 bits commented next to the integer value.

Code 33.5.3 Select the digit

```
//this function is used to select the digit to be turned on
void Digit(int x)
{
  //the switch statement is used to select the right digit
  switch(x)
  {
  case 1:
    //prepare to turn on only the first digit
    digit = 14; //B00001110
    //break is used to get out from the switch function
    break;
  case 2:
    //prepare to turn on only the second digit
    digit = 13; //B00001101
    break;
  case 3:
    //prepare to turn on only the third digit
    digit = 11; //B00001011
    break;
  case 4:
    //prepare to turn on only the fourth digit
    digit = 7; //B00000111
    break;
  case 5:
    //turn on all the digits
    digit = 0; //B00000000
    break;
  }
}
```

Before talking about the **loop()** function, we have created special functions for all the digits, from **0** to **9**, with suggestive names. In each function we will first set the latch pin to **LOW**, then send, using **shiftOut()**, the value of **digit** to the second shift register, which deals with the digits, and to the first shift register an 8 bit value that will have set to **HIGH** some LEDs to reproduce on the display a digit from **0** to **9**. At the end, the latch pin will be set to **HIGH**. There are two functions for each digit, for example we have **zero()** and **zero_no_dot()**, with a single difference between them, namely the bit for the decimal point. In the **zero()** function the bit for the decimal point is set to **HIGH**, while in the **zero_no_dot()** function, the decimal point is set to **LOW**. You can find below two examples of pair functions created for **0** and **1**, but in the code located in **4_Digit_Display** folder you can find the functions created for all digits.

Code 33.5.4 Functions for reproducting **0** and **1**

```
void zero()
{
  //display on the selected digits a "0"
  //and the dots are turned on
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, 191); //B10111111
  digitalWrite(latchPin, 1);
}

void zero_no_dot()
{
  //display on the selected digits a "0"
  //and the dots are turned off
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, 63); //B00111111
  digitalWrite(latchPin, 1);
}

void one()
{
  //display on the selected digits a "1"
  //and the dots are turned on
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, 134); //B10000110
  digitalWrite(latchPin, 1);
}

void one_no_dot()
{
  //display on the selected digits a "1"
  //and the dots are turned off
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, 6); //B00000110
  digitalWrite(latchPin, 1);
}
```

In the **loop()** function we will select all the digits to be turned on and then send the eight byte variables, created at the beginning of the code, one after another at a interval of **1000 ms**. After this, you will see that every segment and the decimal point will light up.

Code 33.5.5 The loop() function

```
  //call the "Digit" function in order to update
  //the value of the "digit" variable
  Digit(5);
  //send two byte values tothe shift registers
  //after this, all the A LEDs will turn on, because
  //all the digits are turned on (they are set to LOW)
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, a);
  digitalWrite(latchPin, 1);
  delay(1000);

  //turn on all the B LEDs
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, b);
  digitalWrite(latchPin, 1);
  delay(1000);

  //turn on all the C LEDs
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, c);
  digitalWrite(latchPin, 1);
  delay(1000);

  //turn on all the D LEDs
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, d);
  digitalWrite(latchPin, 1);
  delay(1000);

  //turn on all the E LEDs
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, e);
  digitalWrite(latchPin, 1);
  delay(1000);

  //turn on all the F LEDs
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, f);
  digitalWrite(latchPin, 1);
  delay(1000);

  //turn on all the G LEDs
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, g);
  digitalWrite(latchPin, 1);
  delay(1000);

  //turn on the LEDs for the dots
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, dot);
  digitalWrite(latchPin, 1);
  delay(1000);
```

Further, we will select only the first digit to be turnd on and set the **A** segment to **HIGH**. After this, all the digits will be selected and a **0** will appear on all of them, with the dot turned off.

Code 33.5.6 The loop() function

```
//select the first digit
Digit(1);
//turn on the A LED on the selected digit
digitalWrite(latchPin, 0);
shiftOut(dataPin, clockPin, MSBFIRST, digit);
shiftOut(dataPin, clockPin, MSBFIRST, a);
digitalWrite(latchPin, 1);
delay(2000);

//select all the digits
Digit(5);
//display on the selected digits a "0"
//the LEDs for the dots are turned off
zero_no_dot();
delay(2000);
```

In the last part of the **loop()** function the first digit will be selected and on it will be displayed a **1**, then the second digit will be selected and on it will be displayed **2**. Next, on the third digit will be displayed a **3** and on the fourth digit will be displayed a **4**.

Code 33.5.7 The loop() function

```
//select the first digit and display on it
//a "1" for 1 s
//the LEDs for the dots are turned on
Digit(1);
one();
delay(1000);

//select the second digit and display on
//it a "2" for 1 s
//the LEDs for the dots are turned on
Digit(2);
two();
delay(1000);

//select the third digit and display on
//it a "3" for 1 s
//the LEDs for the dots are turned on
Digit(3);
three();
delay(1000);

//select the last digit and display on
//it a "4" for 1 s
//the LEDs for the dots are turned on
Digit(4);
four();
delay(1000);
```

34. Lesson 29: Multiplexing

34.1 Overview

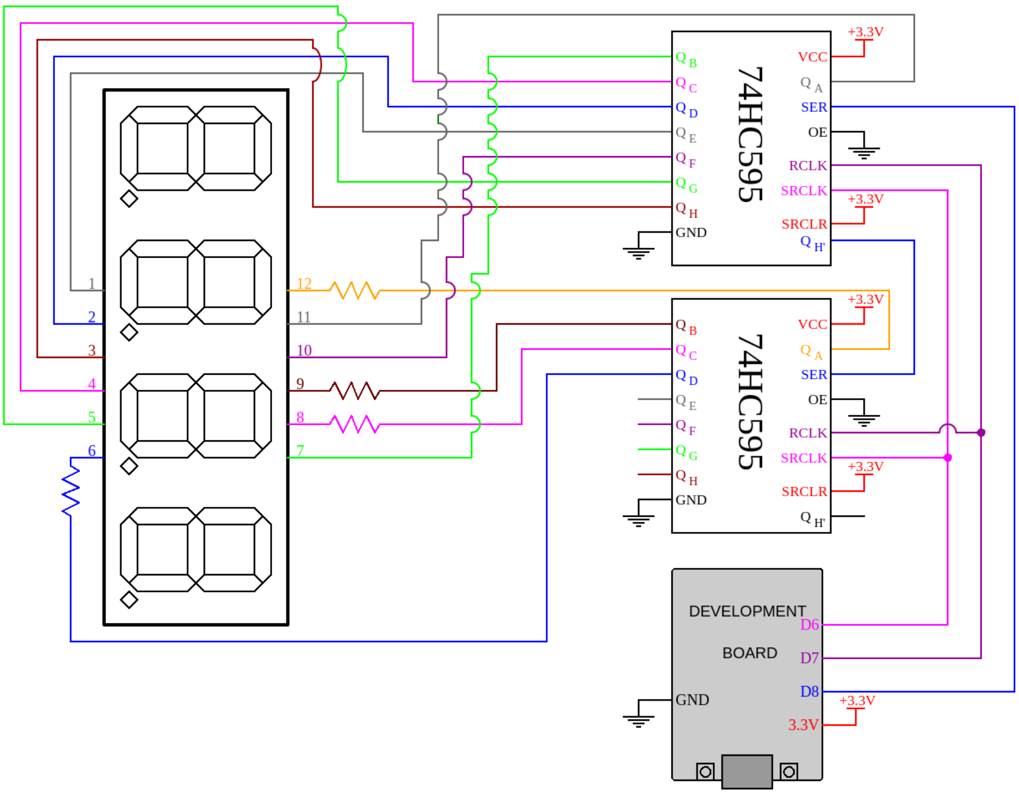
In this lesson you will learn how to use a 4 Digit 7 Segment with Common Cathode display to create a counter that can display from 0 to 1000 seconds.

34.2 Components required

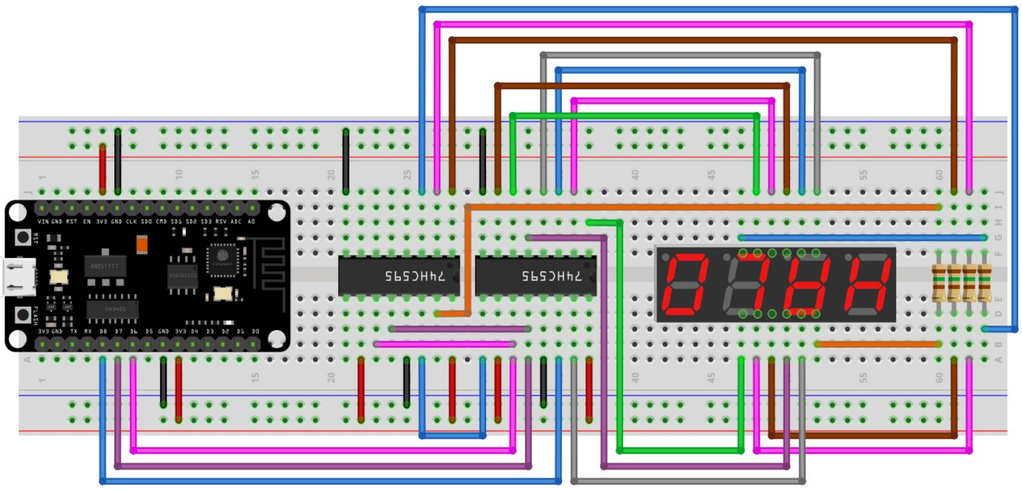
- Development board;
- Breadboard 830p;
- Micro USB – Type A USB cable;
- 2 x 74HC595 Shift Register
- 34 x male-male jumper wires;
- 4 x 150 Ω resistors;
- 4 Digit 7 Segment Common Cathode Display;

34.3 Connections

Below, you can find the schematic:



Below, you can find a visual representation of the connections:



34.4 Code

To display different figures on each digit, for example we want to display 1234, we have to turn on one digit at a time. To do that we need a function to select the digit and call it 4 times, each time for one digit. Giving the fact that we are working with 4 digits and we want to display different figures on each one, we have to place the 4 successive calls in the **loop()** function and, based on the principle of Persistence of Vision, we can see four 7 – segment displays, all displaying different numbers, because the loop() function, where we only have simple functions to call and delays of the order of microseconds, will run aproximately 1 time every 1 milisecond, since this speed is too fast for us to notice, we will be able to see 4 sepparate figures on each digit (multiplexing).

The code for this lesson can be found in the **"Lesson 29: Multiplexing"** folder and is based on the one from the previous lesson and has an additional function.

At the beginning of the code, besides the variables for the pins used by the shift registers and the **digit** variable, we will add a variable that will count how many times the loop function will run, a boolean variable that will tell us when the third digit is active and an integer variable with a predetermined value.

Code 34.4.1 Variables declaration

```
//declare a long variable to count how many times the loop()
//function ran
long n = 0;

//the next boolean variable will tell us when the third digit
//is active, so the dot next to it will turn on
boolean dot;

//fine-tuning value for clock
//at the time of writing the code, with this exact final code,
//this value was right and the loop function was running every 1 ms
//so the value on the third digit will increase every 1 second
//creating a good cronometer, and can count 1000 seconds
const int time_to_wait = 150;
```

The **setup()** function has an additional delay of 1000 ms so that the board will be fully initialised before start counting.

Code 34.4.2 The setup() function

```
void setup()
{
  //set the pins used by the shift registers to output
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);

  //wait 1 s before start counting
  delay(1000);
}
```

In the **Digit()** function we will add some instructions that will turn off all the LEDs of the display before any other instruction, and then we will initialise the **dot** variable with **0**. The value of the **dot** variable will be modified in the **loop()** function before calling the function for displaying a figure on the third digit.

Code 34.4.3 Select the digit

```
void Digit(int x)
{
  //turn off all the digits and segments
  //because we use a common cathode display, to turn off the digits
  //we have to set them to HIGH
  //15 written as 8 bits is B00001111
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, 15);
  shiftOut(dataPin, clockPin, MSBFIRST, 0);
  digitalWrite(latchPin, 1);

  //reset the dot variable
  dot = 0;

  //the switch statement is used to select the right digit
  switch(x)
  {
  case 1:
    //prepare to turn on only the first digit
    digit = 14; //B00001110
    //break is used to get out from the switch function
    break;
  case 2:
    //prepare to turn on only the second digit
    digit = 13; //B00001101
    break;
  case 3:
    //prepare to turn on only the third digit
    digit = 11; //B00001011
    break;
  default:
    //prepare to turn on only the fourth digit
    digit = 7; //B00000111
    break;
  }
}
```

The **Number()** function has an **if** statement to decide if we are at the third digit, then call a function to display a number from 0 to 9, and also the dot will turn on. If we are at digit 1, 2 or 4, then call a function to display a number from the 0 to 9, but with the dot turned off. The pair of functions like **zero()** and **zero_no_dot()**, or **one()** and **one_no_dot()** and so on, are almost the same, the only difference is that the first one has also the dot turned on, when the second has the dot turned off. These functions are defined in the code from the previous lesson, so we will not talk about them.

Code 34.4.4 Display a number on the selected digit

```
void Number(int x)
{
  if (dot == 1)
  {
    switch(x)
    {
      default:
        zero();
        break;
      case 1:
        one();
        break;
      case 2:
        two();
        break;
      case 3:
        three();
        break;
      case 4:
        four();
        break;
      case 5:
        five();
        break;
      case 6:
        six();
        break;
      case 7:
        seven();
        break;
      case 8:
        eight();
        break;
      case 9:
        nine();
        break;
    }
  }
  else
  {
    switch(x)
    {
      default:
        zero_no_dot();
        break;
      case 1:
        one_no_dot();
        break;
      case 2:
        two_no_dot();
        break;
      case 3:
        three_no_dot();
        break;
      case 4:
        four_no_dot();
        break;
      case 5:
        five_no_dot();
        break;
      case 6:
        six_no_dot();
        break;
      case 7:
        seven_no_dot();
        break;
      case 8:
        eight_no_dot();
        break;
      case 9:
        nine_no_dot();
        break;
     }
  }
}
```

All the defined functions are combined in the **loop()** function. Here, we will select the digit, then display a figure on it. After selecting the third digit, we will initialise the **dot** variable with **1**, so in the **Number()** function we will execute the first block of the **if** statement. At the end of the **loop()** function we will increment the **n** number. The **loop()** function will run approximately one time every 1 ms.

Code 34.4.5 The loop() function

```
void loop()
{
  //select first digit
  Digit(1);
  //display a number from 0 to 9
  //because the loop function has to run 100 times per second,
  //we use n/100 to get a 4 digit number and
  // result/1000%10 for first digit
  // result/100%10 for second digit
  // result/10%10 for third digit
  // result%10 for the fourth digit
  Number((n/100/1000)%10);
  delayMicroseconds(time_to_wait);

  //select second digit
  Digit(2);
  //show a number on the display
  Number((n/100/100)%10);
  delayMicroseconds(time_to_wait);

  //select third digit
  Digit(3);
  //change the dot variable so the following functions know
  //that we are at the third digit and turn on the dot
  dot = 1;
  //show a number on the third digit of the display
  Number((n/100/10)%10);
  delayMicroseconds(time_to_wait);

  //select fourth digit
  Digit(4);
  //show a number on the fourth digit of the display
  Number((n/100)%10);
  delayMicroseconds(time_to_wait);

  //increment the number
  n++;

  //we have a four digit display, multiplied by 100
  //we will obtain a 6 digit number, but the loop function
  //will run over and over again, so we have to reset the counter
  //when on our display the number 9999 will appear
  if(n == 1000000)
  {
    n = 0;
  }
}
```

35. Lesson 30: Show Distance on 4 Digit Display. Timer

35.1 Overview

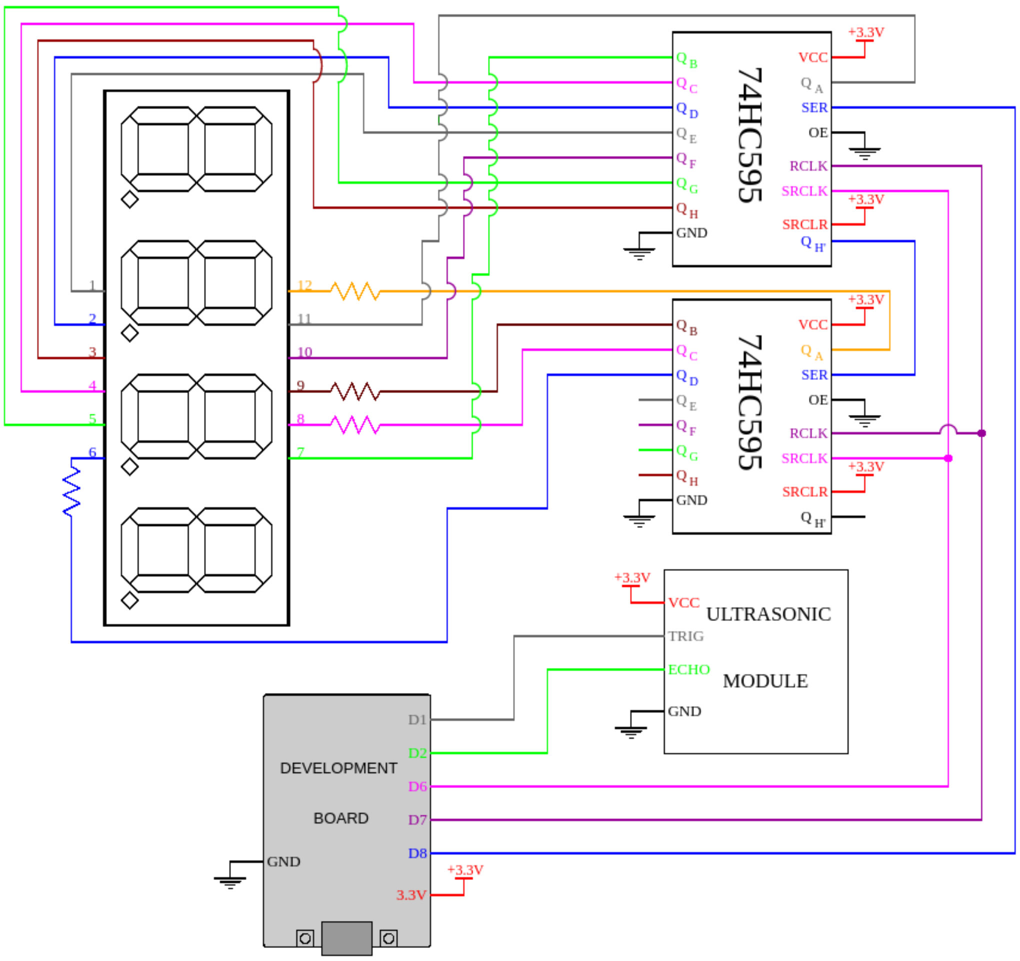
In this lesson you will learn how to display the distance calculated using the Ultrasonic sensor on a 4 Digit 7 Segment Display with Common Cathode. Because we also need to calculate the distance, which takes some time, we will use interrupts to guarantee the multiplexing effect.

35.2 Components required

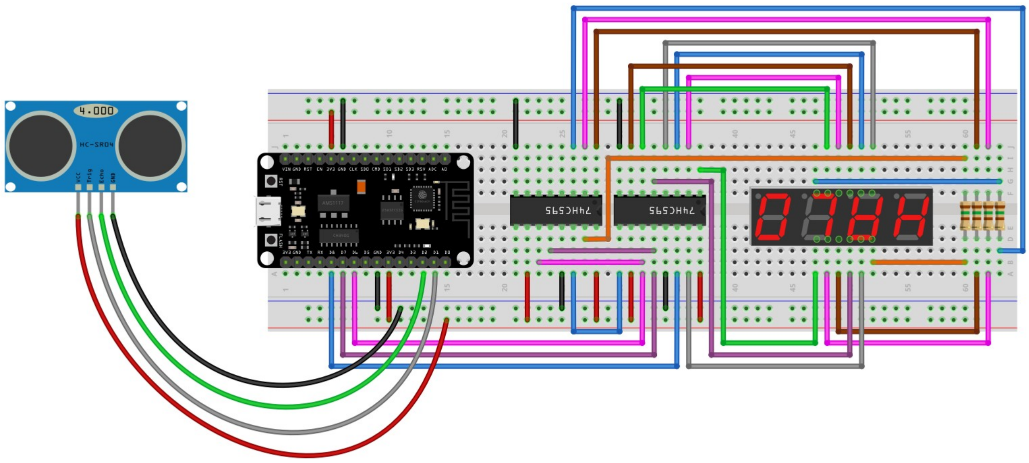
- Development board;
- Breadboard 830p;
- Micro USB – Type A USB cable;
- 2 x 74HC595 Shift Register
- 34 x male-male jumper wires;
- 4 x male-female jumper wires;
- 4 x 150 Ω resistors;
- 4 Digit 7 Segment Common Cathode Display;
- Ultrasonic module HC-SR04+;

35.3 Connections

Below, you can find the schematic:



Below, you can find a visual representation of the connections:



35.4 Code

The code for this lesson can be found in the folder **"Lesson 30: Show Distance on 4 Digit Display. Timer"** and starts by declaring the pins used by the shift registers, the pins for the Ultrasonic module, a byte variable called **digit** that stores the current digit active, a byte variable called **dot** that will modify when we are at the third digit and will have the LSB set to high, otherwise, it will be **0**. Also we need an array with 4 elements that stores the figure that will be displayed on each digit and a **int** variable that will store the number of the current digit.

Code 35.4.1 Variables declaration

```
//declare the pins used by the shift registers
const int dataPin = D8;
const int latchPin = D7;
const int clockPin = D6;

//declare the pins used by the ultrasonic module
const int echoPin = D2;
const int trigPin = D1;

//declare a byte variable to store the current digit active
byte digit;

//the next byte variable will tell us when the third digit
//is active, so the dot next to it will turn on
byte dot;

//declare an array of 4 int elements in which we will be
//storing the figures that are going to be displayed on each
//digit of the display
int digits[4];

//this variable will tells us which is the current digit
int currentDigit = 0;
```

In the **setup()** function we will set the pins used by the shift registers and the trigger pin as **OUTPUT** and the echo pin as **INPUT**. The next part is for the initialization of the timer. We will use timer because we need to create the multiplexing effect by turning the power on and off rapidly on the selected digits and we need to guarantee the timing. We also could have done this by putting our code in the **loop()** function, but the problem is that we need to calculate the distance, which takes some time. Using the **timer1_attachInterrupt(timer_function)** function we will add interrupts to the **timer_function** and using **timer1_enable(TIM_DIV16, 0, 0)** we will enable the timer with 5 ticks/microsecond. The indicated function will execute every 4 ms (**timer1_write(20000)**, with 20000 ticks at 5 ticks/microsecond will result in 4000 microseconds).

Code 35.4.2 The setup() function

```
void setup()
{
  //set the pins used by the shift registers to output
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);

  //the trigger pin (transmitter) must be set as OUTPUT
  pinMode(trigPin, OUTPUT);

  //the echo pin (receiver) must be set as INPUT
  pinMode(echoPin, INPUT);

  //initialize the timer every 4 ms
  //enable the timer with the divider TIM_DIV16 which
  //has a value of 5 ticks/microsecond
  timer1_attachInterrupt(timer_function);
  timer1_enable(TIM_DIV16, 0, 0);

  //divide the ticks by 5 and that is the value in microseconds
  timer1_write(20000);
}
```

In the **loop()** function we will call the function that calculates the distance.

Code 35.4.3 The loop() function

```
void loop()
{
  //calculate the distance
  calculate_distance();
}
```

In the **calculate_distance()** function we will calculate the distance using the same methods as in the previous lessons, then create a new **int** variable, **dist**, in which will be stored the distance multiplied by 10, so we can get also the first decimal. Then, using an **if** statement we will check if the value of **dist** id greater than 0 (the ultrasonic module calculates distances greater than 2 cm), the extract the figures and put them in the **digits** array. At the end, we would want to wait 100 ms before next reading so we can see the results better on the display.

Code 35.4.4 Calculate the distance and store the figures in the declared array

```
void calculate_distance()
{
  //set the trigPin to LOW in order to prepare for the next reading
  digitalWrite(trigPin, LOW);

  //delay for 2 microseconds
  //1 microsecond = 10^(-6) seconds
  delayMicroseconds(2);

  //generate a ultrasound for 10 microseconds then turn off the transmitter
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  //reads the echoPin, returns the sound wave travel time in microseconds
  long duration = pulseIn(echoPin, HIGH, 400*2/0.034);

  //using the formula shown in the guide, calculate the distance
  double distance = duration*0.034/2;

  //multiply the distance by 10 to get also the first decimal
  int dist = (int)(distance*10);

  //if the modified distance is greater than 0, then extract the figures
  if(dist > 0)
  {
    digits[3] = dist%10;
    digits[2] = (dist/10)%10;
    digits[1] = (dist/100)%10;
    digits[0] = dist/1000;
  }

  //wait 100 ms before next reading
  delay(100);
}
```

The main function that will create the multiplexing effect and display on each digit a number is **timer_function()** which has attached **ICACHE_RAM_ATTR** for moving it to the RAM. In this function we will select a digit, then verify if we are at the third one so the **dot** variable will have the bit for the dot set to **HIGH**. Then, using the **Number(digits[currentDigit])** instruction we will display on the current digit its corresponding value from the array. The **currentDigit** will be incremented and reset to **0** if its value is greater than **3**. The last instruction is to call this function again after 4 ms.

Code 35.4.5 The timer function

```
void ICACHE_RAM_ATTR timer_function()
{
  //select the digit
  Digit(currentDigit);

  //if we are at the third digit, then modify the bit
  //for the dor from 0 to 1
  if(currentDigit == 2)
      dot = 0b10000000;

  //display the number on the selected digit
  Number(digits[currentDigit]);

  //go to the next digit
  currentDigit++;

  //if we are at the last digit, then go to the first one
  if(currentDigit > 3)
     currentDigit = 0;

  //initialize this function again after 4 ms
  timer1_write(20000);
}
```

The **Digit()** function remains unchanged, and in the **Number()** function we will have only one **switch** statement that, depending on the value of the **x** paramether, a function will be called and show on the current digit a number.

Code 35.4.6 Call a function to display a figure on the current digit

```
void Number(int x)
{
    //deppending on the value of x, using the switch statement
    //we are calling a specific function to display on the selected
    //digit a number indicated by the name of the function
    switch(x)
    {
      default:
        zero();
        break;
      case 1:
        one();
        break;
      case 2:
        two();
        break;
      case 3:
        three();
        break;
      case 4:
        four();
        break;
      case 5:
        five();
        break;
      case 6:
        six();
        break;
      case 7:
        seven();
      case 8:
        eight();
        break;
      case 9:
        nine();
        break;
    }
}
```

The functions from the previous **switch** statement are a little changed comparing with the ones from the previous lesson. When we will shift the 8 bits for the segments and dot, we will use the bitwise **OR** operator that will modify only the bit for the dot. Implicitly, this bit is **0** and the **dot** byte is also **0**, so there will be no changes. But when we will be at the third digit, the value of **dot** will be **0b10000000**, the result of the operation will change. For example, if we want to display a **0** on the third digit, we will have to do the next operation:

0b00111111 | 0b10000000 and the result will be **0b10111111**

Code 35.4.7 A part of the functions that will display a number on a digit

```
void zero()
{
  //display on the selected digit a "0"
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, 0b00111111 | dot);
  digitalWrite(latchPin, 1);
}

void one()
{
  //display on the selected digit a "1"
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, 0b00000110 | dot);
  digitalWrite(latchPin, 1);
}

void two()
{
  //display on the selected digit a "2"
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, 0b01011011 | dot);
  digitalWrite(latchPin, 1);
}

void three()
{
  //display on the selected digit a "3"
  digitalWrite(latchPin, 0);
  shiftOut(dataPin, clockPin, MSBFIRST, digit);
  shiftOut(dataPin, clockPin, MSBFIRST, 0b01001111 | dot);
  digitalWrite(latchPin, 1);
}
```

36. Lesson 31: Exponential Moving Average

36.1 Overview

This lesson is similar with the previous lesson as we also display the distance calculated using the Ultrasonic module, but as an addition we create a smooth transition between the readings using an Exponential Moving Average.

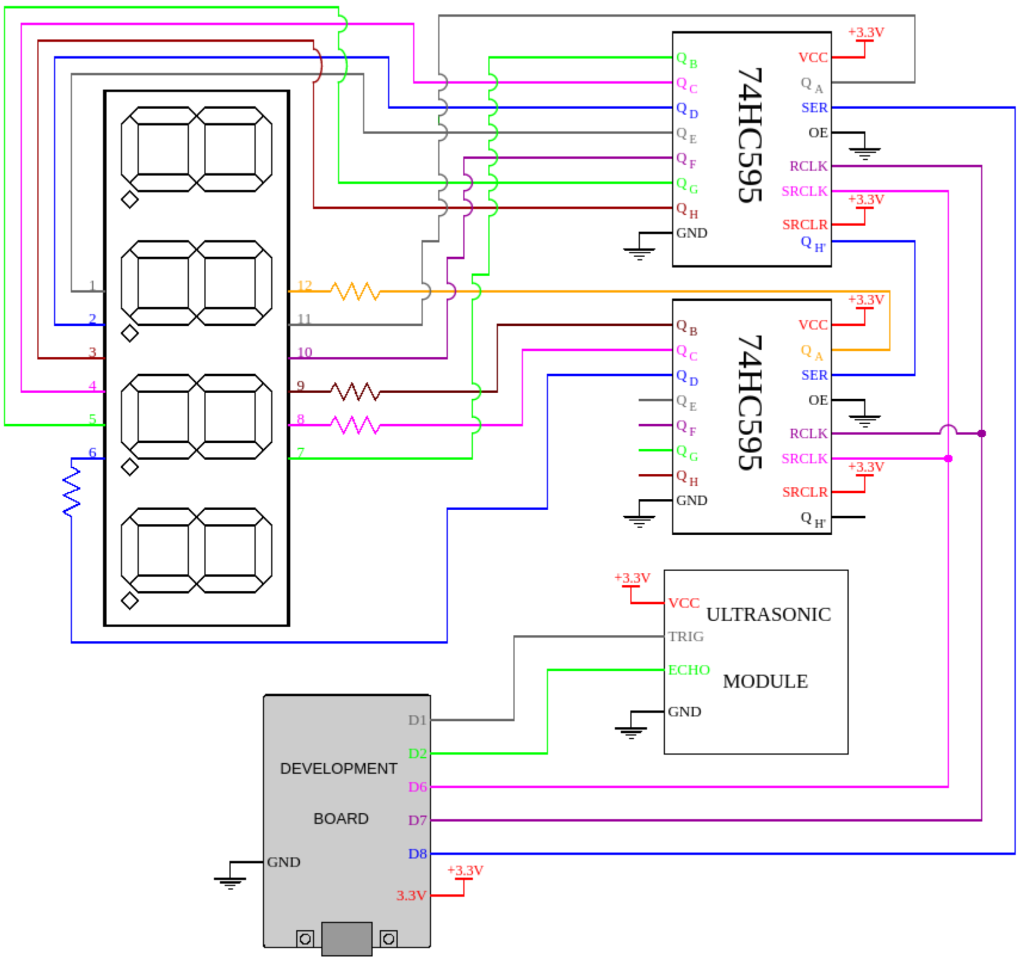
An exponential moving average (EMA) places a greater weight and significance on the most recent data points.

36.2 Components required

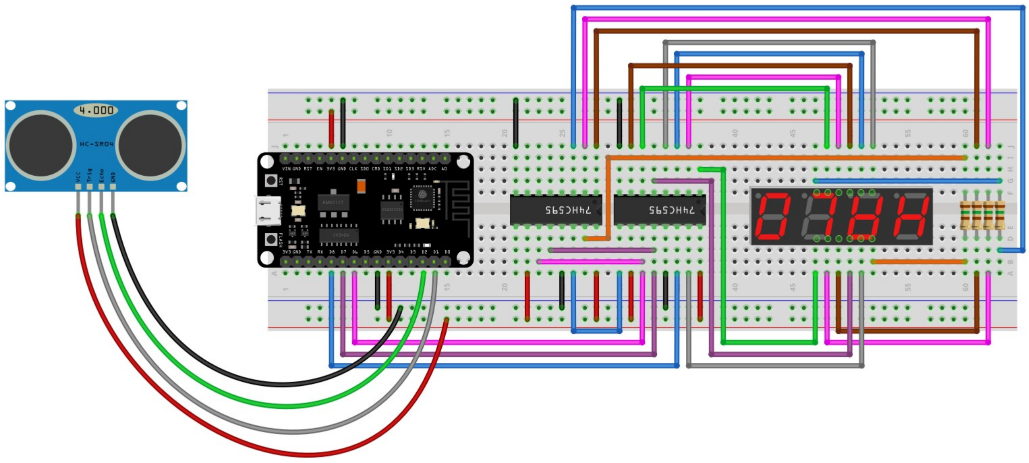
- Development board;
- Breadboard 830p;
- Micro USB – Type A USB cable;
- 2 x 74HC595 Shift Register
- 34 x male-male jumper wires;
- 4 x male-female jumper wires;
- 4 x 150 Ω resistors;
- 4 Digit 7 Segment Common Cathode Display;
- Ultrasonic module HC-SR04+;

36.3 Connections

Below, you can find the schematic:



Below, you can find a visual representation of the connections:



36.4 Code

The code for this lesson is similar with the one included in the previous one and can be found in the **"Lesson 31: Exponential Moving Average"** folder. The only difference is in the **calculate_distance()** function, where we will use the formula for Exponential Moving Average, which also uses the previous reading, to get a smoother transition between the values showed on the display. The formula for EMA is:

EMA = C*K + P*(1 – K)

Where:

- K = 2.0/(N + 1), with N = length of the EMA
- C = current distance
- EMA = the current EMA value
- P = the previous EMA value

And the new **calculate_distance()** function is:

Code 36.4.1 Calculate the distance

```
void calculate_distance()
{
  //set the trigPin LOW in order to prepare for the next reading
  digitalWrite(trigPin, LOW);

  //delay for 2 microseconds
  delayMicroseconds(2);

  //generate a ultrasound for 10 microseconds then turn off the transmitter.
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  //Reads the echoPin, returns the sound wave travel time in microseconds
  long duration = pulseIn(echoPin, HIGH, 400*2/0.034);

  //using the formula shown in the guide, calculate the distance
  double distance = duration*0.034/2;

  //calculate the Exponential Moving Average
  double k = 2.0/(10+1);
  ema = distance*k + ema*(1-k);

  //multiply the ema by 10 to get also the first decimal
  int dist = (int)(ema*10);

  //if the modified distance is greater than 0, then extract the figures
  if(dist > 0)
  {
    digits[3] = dist%10;
    digits[2] = (dist/10)%10;
    digits[1] = (dist/100)%10;
    digits[0] = dist/1000;
  }

  //wait 50 ms before next reading
  delay(50);
}
```

37. Lesson 32: OTA Upload

37.1 Overview

This lesson is an advanced one as you might run into different problems until you master how to upload code **O**ver **T**he **A**ir **(OTA)**. It means that you can load a sketch on your development board without a USB cable. All you need is to be conected to the same network as the board.

37.2 Components required

- Development board;
- Micro USB – Type A USB cable;

37.3 Code

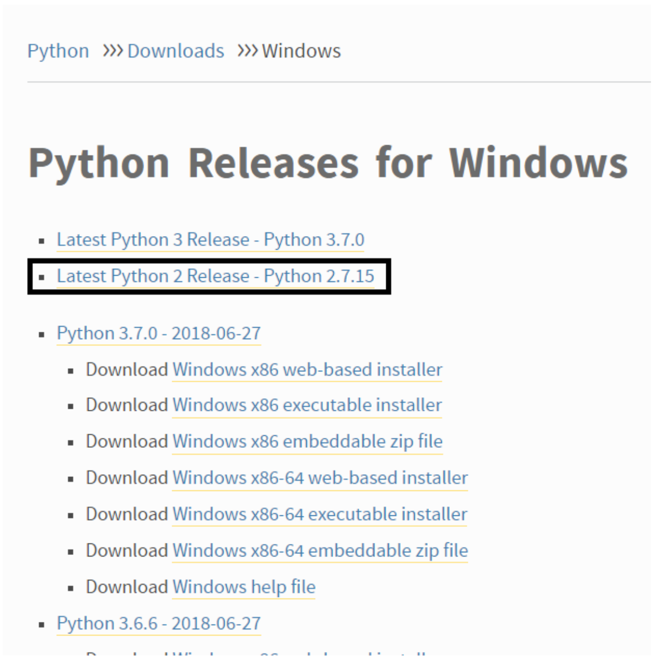
For this to work, besides the Arduino IDE software, you need to install Python 2 (python 3 is not supported). Python comes preinstalled on most Linux distributions.

For windows users:

Download

You can download python by accessing the following link: https://www.python.org/downloads/windows/.

Then, click on **Latest Python 2 Release**:

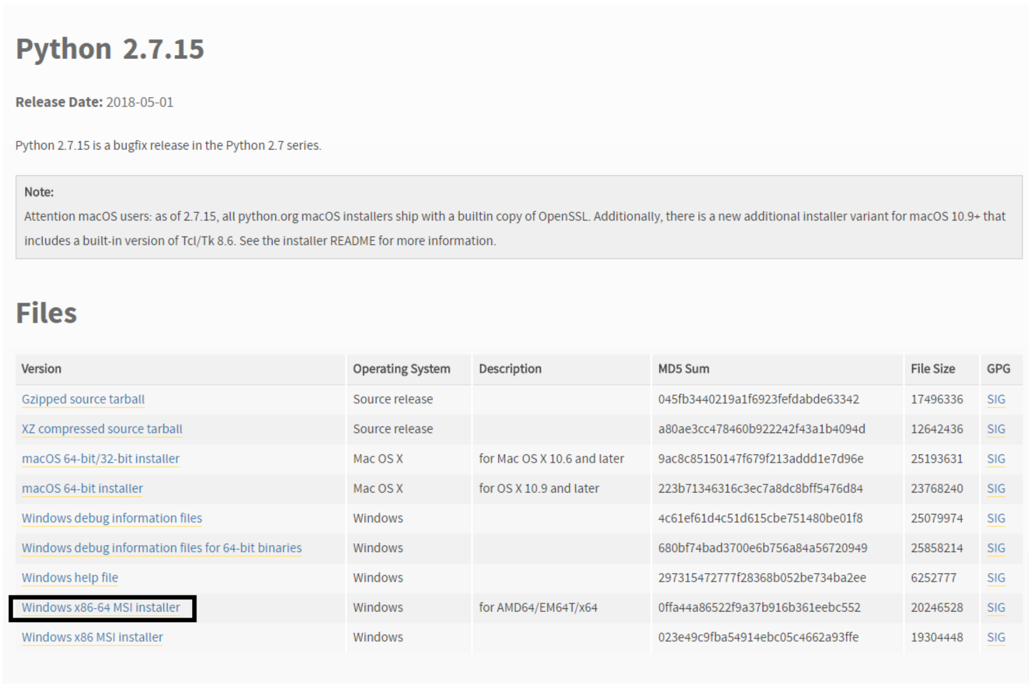


Python ⋙ Downloads ⋙ Windows

Python Releases for Windows

- Latest Python 3 Release - Python 3.7.0
- Latest Python 2 Release - Python 2.7.15
- Python 3.7.0 - 2018-06-27
  - Download Windows x86 web-based installer
  - Download Windows x86 executable installer
  - Download Windows x86 embeddable zip file
  - Download Windows x86-64 web-based installer
  - Download Windows x86-64 executable installer
  - Download Windows x86-64 embeddable zip file
  - Download Windows help file
- Python 3.6.6 - 2018-06-27

And click on **Windows x86-64 MSI Installer**:



Python 2.7.15

Release Date: 2018-05-01

Python 2.7.15 is a bugfix release in the Python 2.7 series.

Note:
Attention macOS users: as of 2.7.15, all python.org macOS installers ship with a builtin copy of OpenSSL. Additionally, there is a new additional installer variant for macOS 10.9+ that includes a built-in version of Tcl/Tk 8.6. See the installer README for more information.

Files

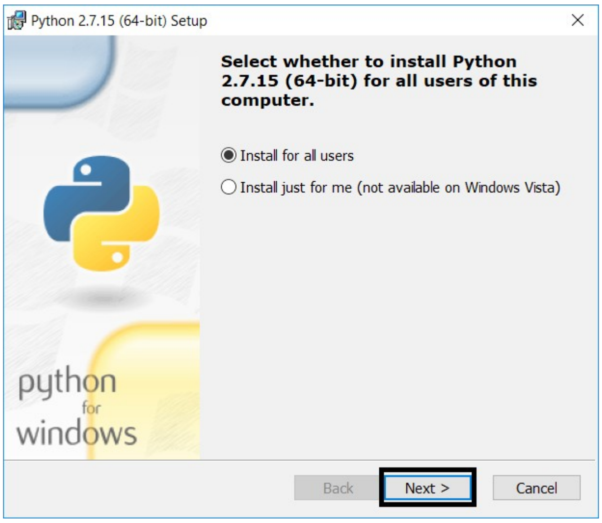
Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		045fb3440219a1f6923fefdabde63342	17496336	SIG
XZ compressed source tarball	Source release		a80ae3cc478460b922242f43a1b4094d	12642436	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	9ac8c85150147f679f213addd1e7d96e	25193631	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	223b71346316c3ec7a8dc8bff5476d84	23768240	SIG
Windows debug information files	Windows		4c61ef61d4c51d615cbe751480be01f8	25079974	SIG
Windows debug information files for 64-bit binaries	Windows		680bf74bad3700e6b756a84a56720949	25858214	SIG
Windows help file	Windows		297315472777f28368b052be734ba2ee	6252777	SIG
Windows x86-64 MSI installer	Windows	for AMD64/EM64T/x64	0ffa44a86522f9a37b916b361eebc552	20246528	SIG
Windows x86 MSI installer	Windows		023e49c9fba54914ebc05c4662a93ffe	19304448	SIG

Install

a) Find the **.exe** file you just downloaded.

b) Double click on it.

c) Now click on **Next**.



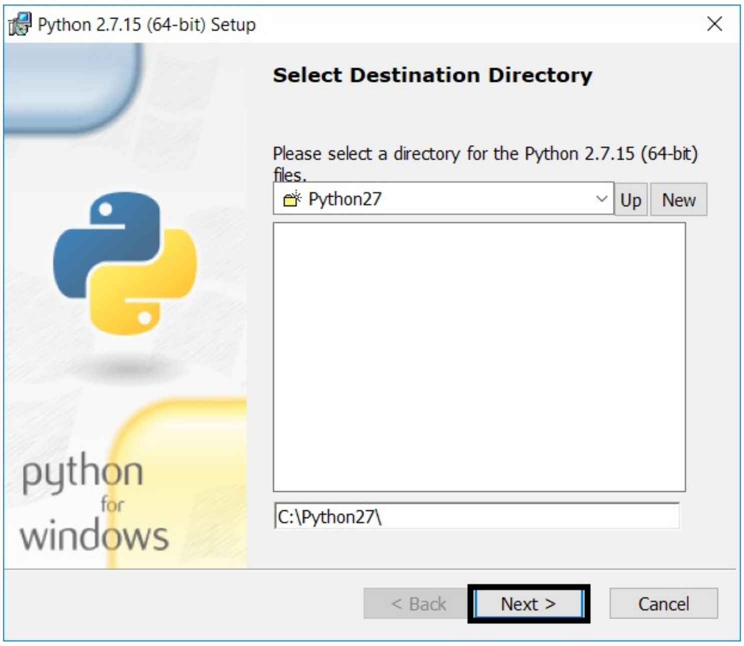
Python 2.7.15 (64-bit) Setup

Select whether to install Python 2.7.15 (64-bit) for all users of this computer.

(●) Install for all users

( ) Install just for me (not available on Windows Vista)

d) Select location where you want to install python and then click **Next**.



Python 2.7.15 (64-bit) Setup

Select Destination Directory

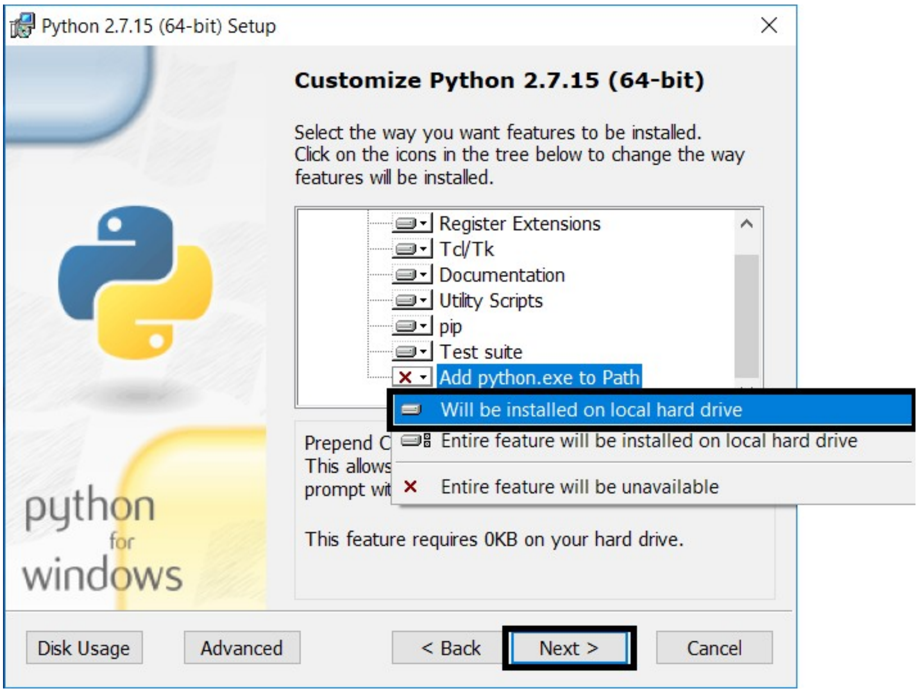
Please select a directory for the Python 2.7.15 (64-bit) files.

Python27

C:\Python27\

e) Make sure you select **Add python.exe to Path** and then **Will be installed**

on local hard drive. Then click **Next**.


Python 2.7.15 (64-bit) Setup

Customize Python 2.7.15 (64-bit)

Select the way you want features to be installed.
Click on the icons in the tree below to change the way
features will be installed.

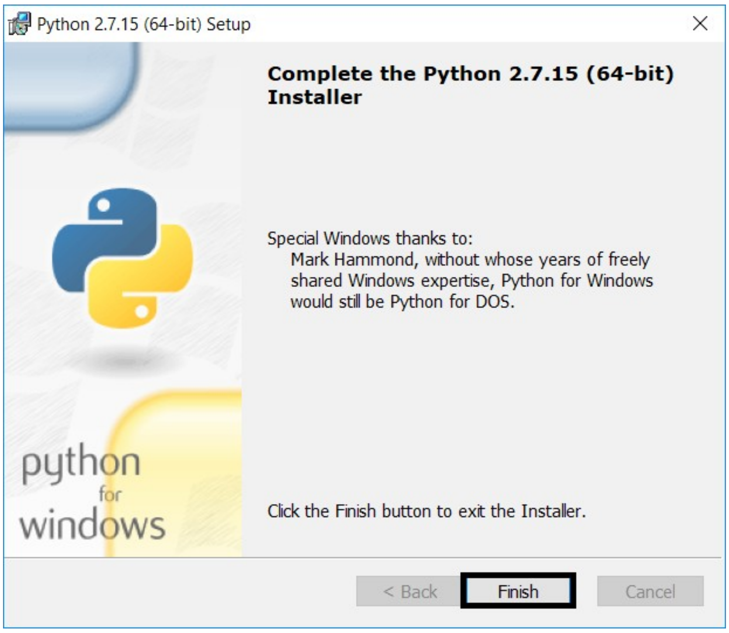
- Register Extensions
- Tcl/Tk
- Documentation
- Utility Scripts
- pip
- Test suite
- Add python.exe to Path

Will be installed on local hard drive
Entire feature will be installed on local hard drive
Entire feature will be unavailable

Prepend C
This allows
prompt wit

This feature requires 0KB on your hard drive.

f) Now click on **Finish**.


Python 2.7.15 (64-bit) Setup

Complete the Python 2.7.15 (64-bit) Installer

Special Windows thanks to:
Mark Hammond, without whose years of freely shared Windows expertise, Python for Windows would still be Python for DOS.

Click the Finish button to exit the Installer.

Congratulations! Python is now installed. Now let's go to the code.

You can find the code in the folder called **"Lesson 32: OTA Upload"** -> **"OTA"**. Also, in the **"Lesson 32: OTA Upload"** folder you can find some of the previous lesson updated with OTA, so check them to exercise. You can find in the comments some details. But let's explain here a part of the code.

First of all, we have to include the library **ArduinoOTA.h**. This step is very important, because some of the following instructions in the code will not work without this library. Next, we have to connect the development board to the wireless network. You have to modify the credentials of the network. Replace the dots with the **SSID** and **PASSWORD** of your WiFi network.

Code 37.3.1 The library used and the declaration of two variables for wireless

```
#include <ArduinoOTA.h>

const char* ssid = "..............";
const char* password = "..........";
```

Next, in the **setup()** function, set up the serial communication with the computer and connect to the WiFi. The baud rate is **115200** and the board is configured as **WiFi_STA**.

Code 37.3.2 The setup() function

```
  //start the serial communication with the computer at 115200 bits/s
  Serial.begin(115200);

  //print a message into the Serial Monitor
  Serial.println("Booting");

  //set the WiFi mode to WiFi_STA.
  //the WiFi_STA means that the board will act as a station,
  //similar to a smartphone or laptop
  WiFi.mode(WIFI_STA);

  //connect to the WiFi network using the ssid and password strings
  WiFi.begin(ssid, password);
```

Now, check if the ESP is connected to the network, otherwise, restart it. Repeat these steps until the board is finally connected to the network. In case your board restarts several times, check if the network is up and if you entered correctly the credentials.

Code 37.3.3 The setup() function

```
  //check if the esp is connected to the network
  //otherwise, restart the board until it finally connects
  //to the network
  while (WiFi.waitForConnectResult() != WL_CONNECTED) {
    Serial.println("Connection Failed! Rebooting...");
    delay(5000);
    ESP.restart();
  }
```

The last instructions you have to write in **setup()** function is to initialise the **OTA** and to display the IP of the board in serial monitor.

Code 37.3.4 The setup() function

```
  //initialise the OTA
  ArduinoOTA.begin();

  //display the IP of the board in Serial Monitor
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
```

Finally, in the **loop()** function you have to listen if there is any available request to upload the code over the air.

Code 37.3.5 The loop() function

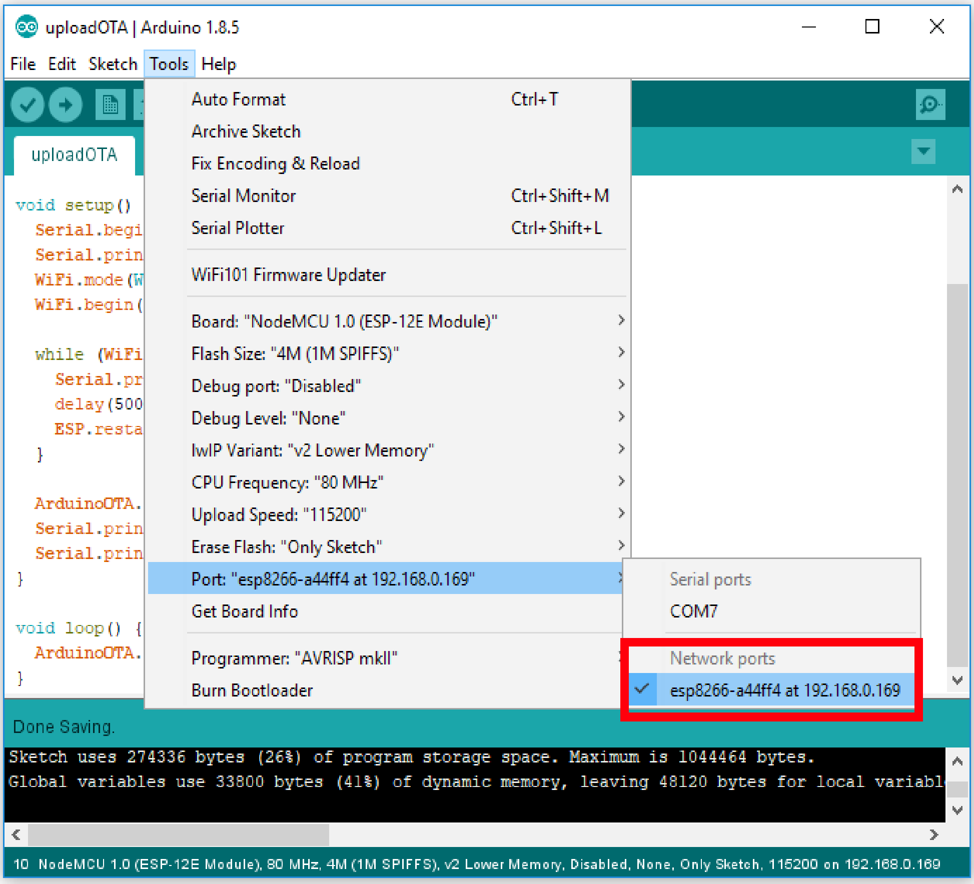
```
void loop()
{
  //listen if there is any available request to upload the code OTA
  ArduinoOTA.handle();
}
```

NOTE! In order to be able to upload over the air, you have to put these instructions in the code you upload. Otherwise, you will be able to upload only once.

Sometimes, you might get some errors that can be easily solved by cutting the power of the board for some seconds then powering it up again.

One disadvantage of this method to upload code is that the Serial Monitor is not available.

The only noticeable difference between upload over the air and over the USB cable is that instead of selecting a **Serial port**, you now have to pick a **Network Port**. In case you uploaded the code and your board doesn't appear at **Network Port**, please restart the **Arduino IDE** and wait several seconds.


uploadOTA | Arduino 1.8.5

File Edit Sketch Tools Help

Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 Firmware Updater
Board: "NodeMCU 1.0 (ESP-12E Module)"
Flash Size: "4M (1M SPIFFS)"
Debug port: "Disabled"
Debug Level: "None"
IwIP Variant: "v2 Lower Memory"
CPU Frequency: "80 MHz"
Upload Speed: "115200"
Erase Flash: "Only Sketch"
Port: "esp8266-a44ff4 at 192.168.0.169"
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader

Serial ports
COM7
Network ports
✓ esp8266-a44ff4 at 192.168.0.169

```
void setup()
  Serial.begi
  Serial.prin
  WiFi.mode(W
  WiFi.begin(

  while (WiFi
    Serial.pr
    delay(500
    ESP.resta
  }

  ArduinoOTA.
  Serial.prin
  Serial.prin
}

void loop() {
  ArduinoOTA.
}
```

Done Saving.

```
Sketch uses 274336 bytes (26%) of program storage space. Maximum is 1044464 bytes.
Global variables use 33800 bytes (41%) of dynamic memory, leaving 48120 bytes for local variabl
```

10 NodeMCU 1.0 (ESP-12E Module), 80 MHz, 4M (1M SPIFFS), v2 Lower Memory, Disabled, None, Only Sketch, 115200 on 192.168.0.169

38. Lesson 33: Soft Access Point

38.1 Overview

In this lesson you will learn how to configure the development board, ESP8266, to run in soft access point mode so that WiFi enabled stations can connect to it. Additionally, we will add a web server to work on top of it and print a simple message in the web browser. In lesson 15 you have learned how to connect the development board to a WiFi network, but, this time, the development board will act as an Access Point. The best part is that the board will always have the following IP address, **192.168.4.1**, but the downside is that it won't have access to the internet, so using a web server, in the HTML page you will not be able to include any library from a CDN, the libraries shoult be stored locally.

38.2 Components required

- Development board;
- Micro USB – Type A USB cable;

38.3 Code

This lesson will have two codes, a easy one in which you will learn how to set up the development board to run in the Soft Access Point mode, and in the second one we will add a web server and print a simple "Hello world!" in the browser. The setup for the web server is the same from the previous lessons, so you can configure it and add any web page you want.

First code can be found in the folder called **Access_Point**, which is inside the folder called **"Lesson 33: Soft Access Point"**, and additional details can be found in the code, as comments.

The code is very easy to understand. Firstly, we have to include the **ESP8266WiFi.h** library, which will provide all the functionality needed to set the access point, and we have to define two global variables, **ssid** and **password**, which are the name and password of the network hosted by the development board.

Code 38.3.1 The libray used and the credentials of the hosted network

```
#include <ESP8266WiFi.h>

//here you have to insert your desired credentials
const char* ssid = "ssid";
const char* password = "password";
```

The **setup()** function will start by opening a serial connection, to show some messages in the Serial Monitor. Now, we have to call the **softAP** method on the WiFi extern variable, passing as input both the ssid and password variables defined above.

Code 38.3.2 The setup() function

```
void setup()
{
  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  //wait 1 s for the Serial communication to start
  delay(1000);

  //display a message in Serial Monitor
  Serial.print("Configuring access point...");

  //set the AP with the ssid and password entered above
  WiFi.softAP(ssid, password);
}
```

And that's it. Now, the development board will act as an Access Point and you can connect to it using the credentials defined above (replace them with your own).

For the second code, which can be found in the **AP_webserver** folder, which is inside the folder called **"Lesson 33: Soft Access Point"**, we will need to add the web server functionality to the previous code, things defined already in the previous lessons.

We have to include the **ESP8266WebServer.h** library, in order to be able to set up our web server. Next, you have to set up the server side. First, you have to create a new object ESP8266WebServer. Moreover, you have to specify the port it is listening to. The default HTTP port is 80 but you can change it however you want. Make sure it is not used by another service.

Code 38.3.3 The libraries used and the credentials of the hosted wireless network

```
#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>

//here you have to insert your desired credentials
const char* ssid = "ssid";
const char* password = "password";

ESP8266WebServer server(80);
```

Now, as in the **Wireless Connectivity** lesson, we will add the **setupServer()** function and create a handler for the default location (which is "/"). In the **htmlIndex()** function we will send a message that will be displayed in the HTML page.

Code 38.3.4 Set up the server and send a message to the client

```
//setupServer() function is used to set up and organise
//the website
void setupServer()
{
  //the method "server.on()" is to call a function when
  //the user access the location
  //the default location is "/"
  server.on("/", htmlIndex);

  //start the server
  server.begin();
  //print in serial manager that the HTTP server is started
  Serial.println("HTTP server started");
}

//the htmlIndex() is called everytime somebody access the address
//of the board in the browser and sends back a message.
void htmlIndex()
{
  //the mssage variable is used to store the message sent to the
  //user
  String message = "Hello world!";

  //send the message to the user
  server.send(200, "text/html", message );
}
```

In the **setup()** function we will start a serial communication with the computer and put the necessary instructions for the Access Point.

Code 38.3.5 The setup() function

```
  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  //wait 1 s for the Serial communication to start
  delay(1000);

  //display a message in Serial Monitor
  Serial.print("Configuring access point...");

  //set the AP with the ssid and password entered above
  WiFi.softAP(ssid, password);
```

You can add the following lines in your **setup()** function to print the IP adress of the server in the Serial Monitor, but usually the IP adress in the Soft Access Point mode is **192.168.4.1**. At the end of the **setup()** function we will call the **setupServer()** function.

Code 38.3.6 The setup() function

```
  //display the server IP address in Serial Monitor
  IPAddress myIP = WiFi.softAPIP();
  Serial.print("AP IP address: ");
  Serial.println(myIP);

  //call the function used to setup the server
  setupServer();
```

Finally, in the **loop()** function we need to put an instruction to listen for incoming requests.

Code 38.3.7 The loop() function

```
void loop()
{
  //the method below is used to manage the incoming request
  //from the user
  server.handleClient();
}
```

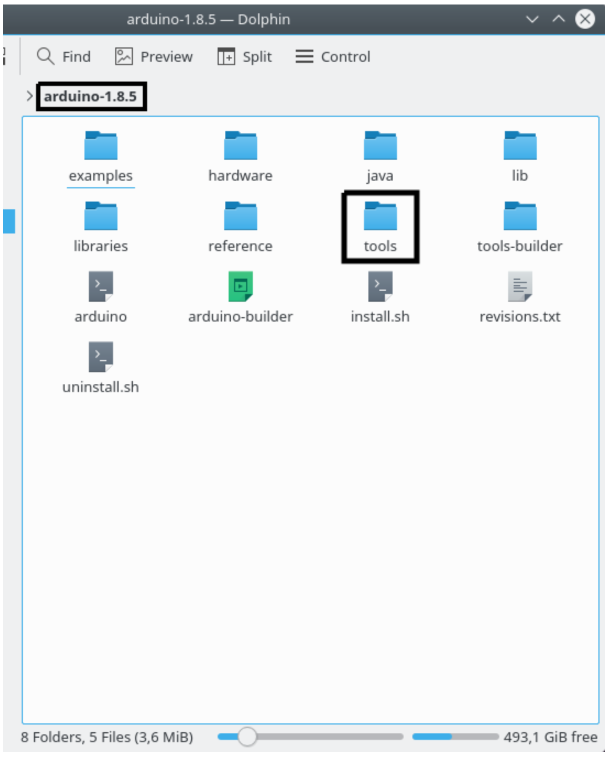
39. Lesson 34: SPIFFS

SPIFFS stands for SPI (**S**erial **P**eripheral **I**nterface**)** Flash **F**illing **S**ystem. It is very useful because it allows you to partition the system flash so it can be used both for code and support a file system. You can store there web pages, configurations or data that is not erased when the board is powered off.

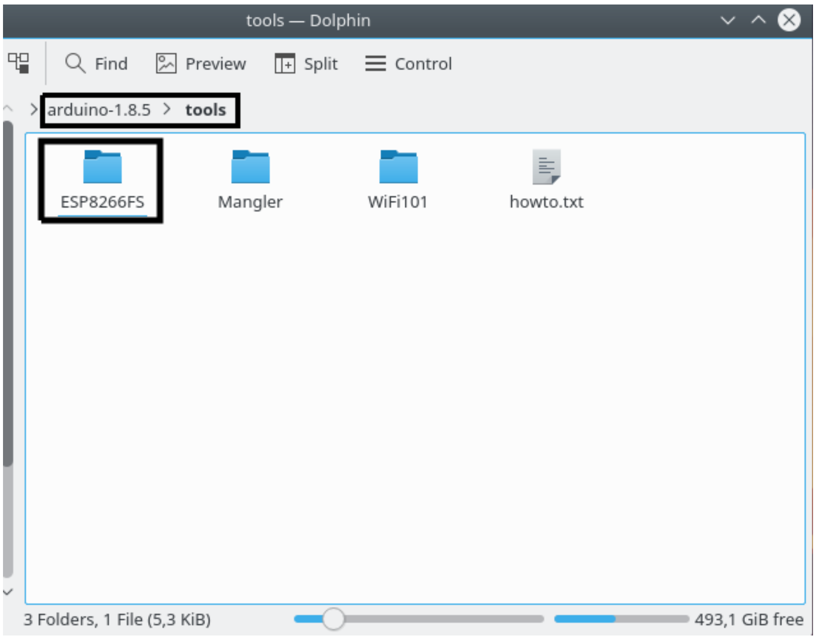
39.1 Configure Arduino IDE

For linux users

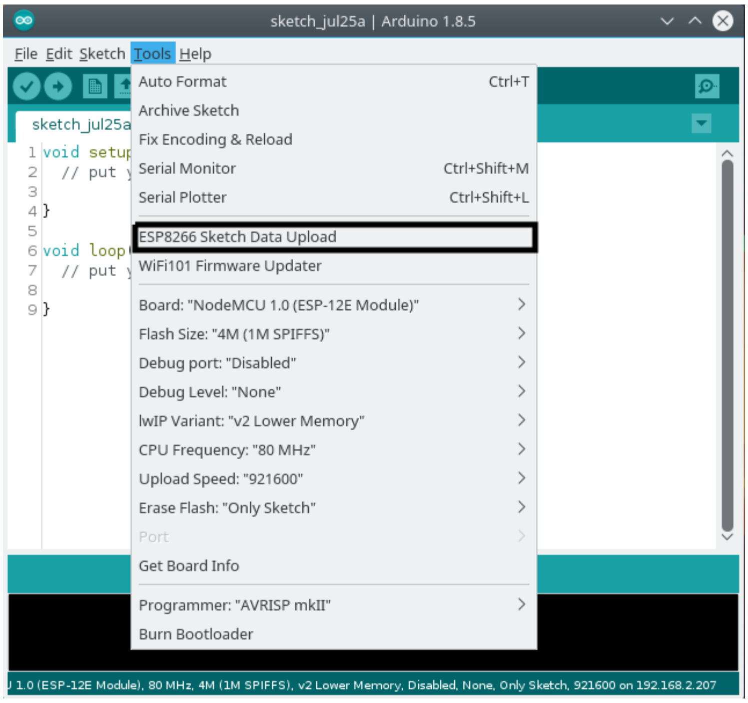
- In the **Lesson 34: SPIFFS -> Spiffs** folder, you will find an archive named **"ESP8266FS-0.2.0.zip".** Unzip it.
- Locate the **ESP8266FS** folder. Copy it using the shortcut **CTRL + C** or **RIGHT CLICK → COPY.**
- Locate the **arduino-1.8.5** installation folder. Usually, it can be found in the **Downloads** folder, because, in our case, there we extracted the archive when we installed the **Arduino IDE**.
- Inside **arduino 1.8.5** folder, you should find a folder called **tools.**



- Paste the **ESP8266FS** folder inside the **arduino 1.8.5/tools** folder.



- Congratulations! You have just configured the **Arduino IDE** to upload data in SPIFFS.


sketch_jul25a | Arduino 1.8.5
File Edit Sketch Tools Help
sketch_jul25a
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
ESP8266 Sketch Data Upload
WiFi101 Firmware Updater
Port
Get Board Info
Burn Bootloader

```
void setup
  // put y
}

void loop(
  // put y
}
```

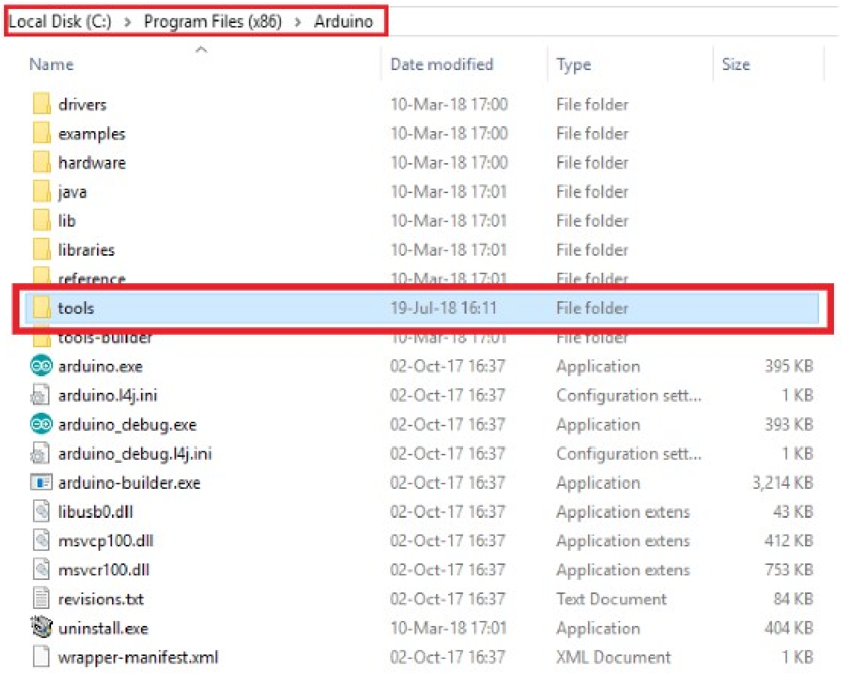
Board: "NodeMCU 1.0 (ESP-12E Module)"
Flash Size: "4M (1M SPIFFS)"
Debug port: "Disabled"
Debug Level: "None"
lwIP Variant: "v2 Lower Memory"
CPU Frequency: "80 MHz"
Upload Speed: "921600"
Erase Flash: "Only Sketch"
Programmer: "AVRISP mkII"

1.0 (ESP-12E Module), 80 MHz, 4M (1M SPIFFS), v2 Lower Memory, Disabled, None, Only Sketch, 921600 on 192.168.2.207

For windows users

- In the **Lesson 34: SPIFFS -> Spiffs** folder, you will find an archive named **"ESP8266FS".** Unzip it.
- Locate the **ESP8266FS** folder that is inside **ESP8266FS-0.2.0** folder. Copy it using the shortcut **CTRL+C** or **RIGHT CLICK → COPY.**
- Locate the **Arduino** installation folder. Usually, it can be found it

  C:\Program Files (x86).
- Inside **Arduino** folder, you should find a folder called **tools.**

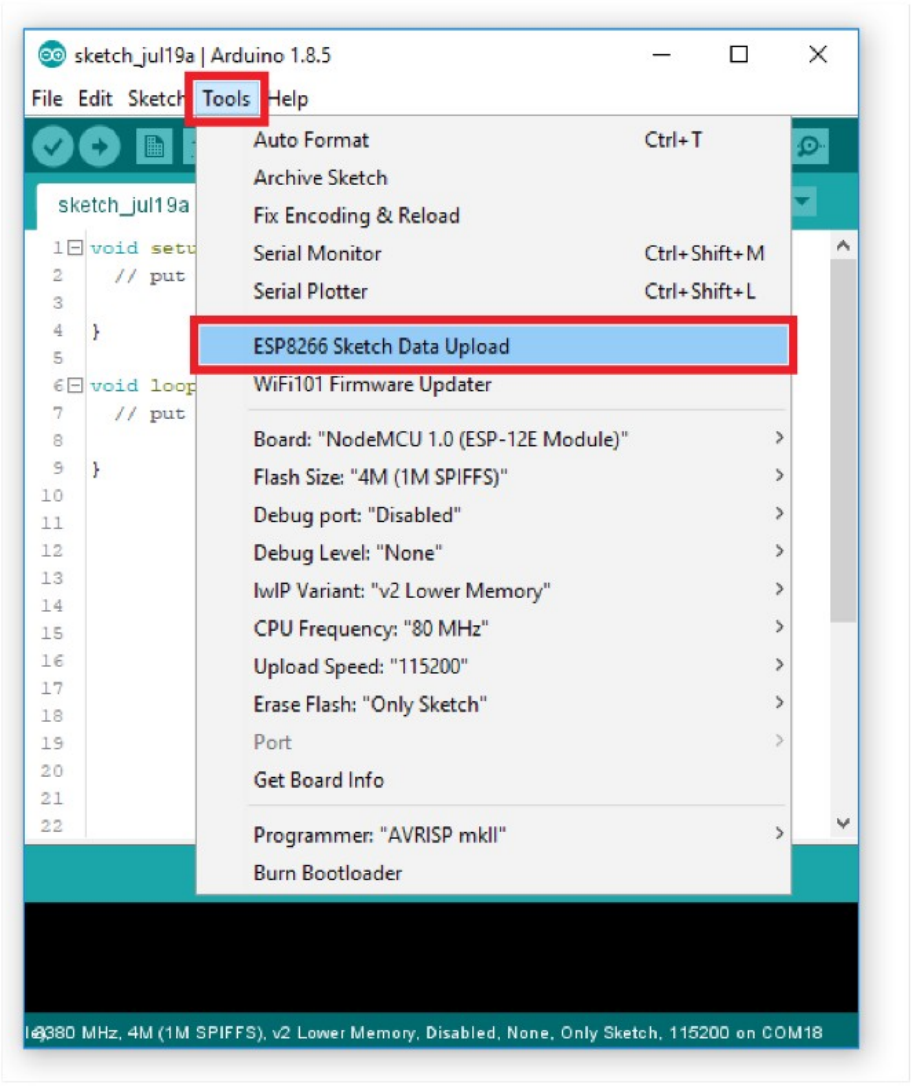


Local Disk (C:) > Program Files (x86) > Arduino

Name	Date modified	Type	Size
drivers	10-Mar-18 17:00	File folder	
examples	10-Mar-18 17:00	File folder	
hardware	10-Mar-18 17:00	File folder	
java	10-Mar-18 17:01	File folder	
lib	10-Mar-18 17:01	File folder	
libraries	10-Mar-18 17:01	File folder	
reference	10-Mar-18 17:01	File folder	
tools	19-Jul-18 16:11	File folder	
tools-builder	10-Mar-18 17:01	File folder	
arduino.exe	02-Oct-17 16:37	Application	395 KB
arduino.l4j.ini	02-Oct-17 16:37	Configuration sett...	1 KB
arduino_debug.exe	02-Oct-17 16:37	Application	393 KB
arduino_debug.l4j.ini	02-Oct-17 16:37	Configuration sett...	1 KB
arduino-builder.exe	02-Oct-17 16:37	Application	3,214 KB
libusb0.dll	02-Oct-17 16:37	Application extens	43 KB
msvcp100.dll	02-Oct-17 16:37	Application extens	412 KB
msvcr100.dll	02-Oct-17 16:37	Application extens	753 KB
revisions.txt	02-Oct-17 16:37	Text Document	84 KB
uninstall.exe	10-Mar-18 17:01	Application	404 KB
wrapper-manifest.xml	02-Oct-17 16:37	XML Document	1 KB

- Paste the **ESP8266FS** folder inside the **Arduino/tools** folder.

- Congratulations! You have just configured the **Arduino IDE** to upload data in SPIFFS.


sketch_jul19a | Arduino 1.8.5
File Edit Sketch Tools Help
sketch_jul19a
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
ESP8266 Sketch Data Upload
WiFi101 Firmware Updater
Port
Get Board Info
Burn Bootloader

```
void setu
  // put

}

void loop
  // put

}
```

Board: "NodeMCU 1.0 (ESP-12E Module)"
Flash Size: "4M (1M SPIFFS)"
Debug port: "Disabled"
Debug Level: "None"
lwIP Variant: "v2 Lower Memory"
CPU Frequency: "80 MHz"
Upload Speed: "115200"
Erase Flash: "Only Sketch"
Programmer: "AVRISP mkII"

```
80 MHz, 4M (1M SPIFFS), v2 Lower Memory, Disabled, None, Only Sketch, 115200 on COM18
```

39.2 Check the flash memory

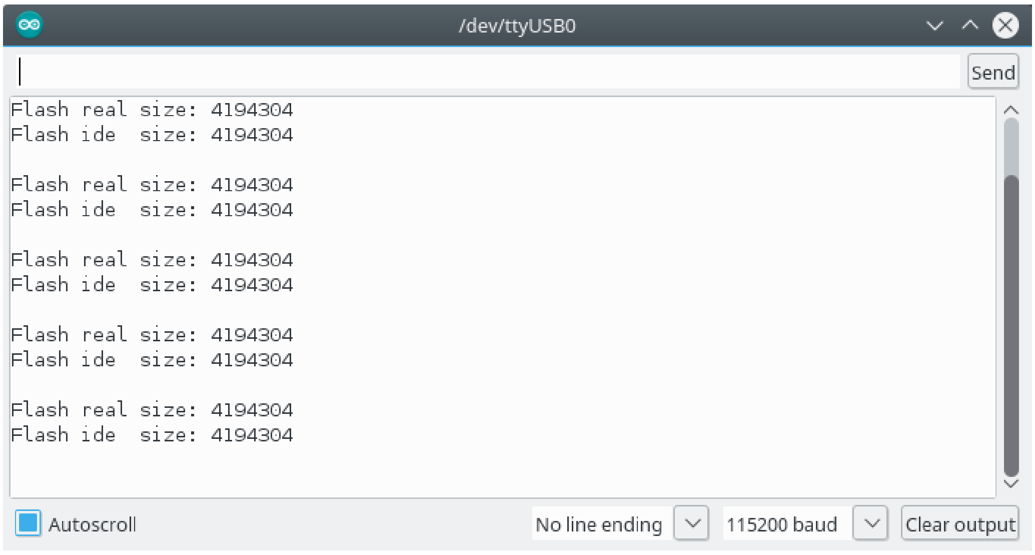
It is possible to upload a code on the development board and find how much flash memory it has.

You can find it in the folder **Lesson 34: SPIFFS -> CheckFlashConfig.** Upload it, open the Serial Monitor and set the baud rate at **115200**. You should be able to see the size of the flash memory in bytes. You should divide that value to 1024*1024 to find the size in megabytes.

For example:

$$\frac{4194304}{1024*1024}=\mathbf{4}$$

which means the board has **4** megabytes of flash memory.



```
Flash real size: 4194304
Flash ide  size: 4194304

Flash real size: 4194304
Flash ide  size: 4194304

Flash real size: 4194304
Flash ide  size: 4194304

Flash real size: 4194304
Flash ide  size: 4194304

Flash real size: 4194304
Flash ide  size: 4194304
```

This code is very simple. In the **setup()** function we only need to start a serial communication with the computer.

Code 39.2.1 The setup() function

```
void setup()
{
  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);
}
```

In the **loop()** function we need to create two 32 bit variables that will store the real flash size and the size from IDE. After creating them, we need to display the values in the Serial Monitor and put a delay of 5000 ms before the next reading.

Code 39.2.2 The loop() function (read the real flash size and the IDE size)

```
void loop()
{
  //read the real size of the chip
  //and also get the size set from Arduino the IDE
  //they should match
  uint32_t real = ESP.getFlashChipRealSize();
  uint32_t ide = ESP.getFlashChipSize();

  //display in Serial Monitor the real size
  Serial.printf("Flash real size: %u\n", real);

  //display in Serial Monitor the size set from Arduino IDE
  Serial.printf("Flash ide  size: %u\n\n", ide);

  //wait 5 s
  delay(5000);
}
```

39.3 Upload data in flash memory

A big advantage of uploading data in the flash memory is that it is independently of the code uploaded. It means that you can upload code while keeping the data in flash.

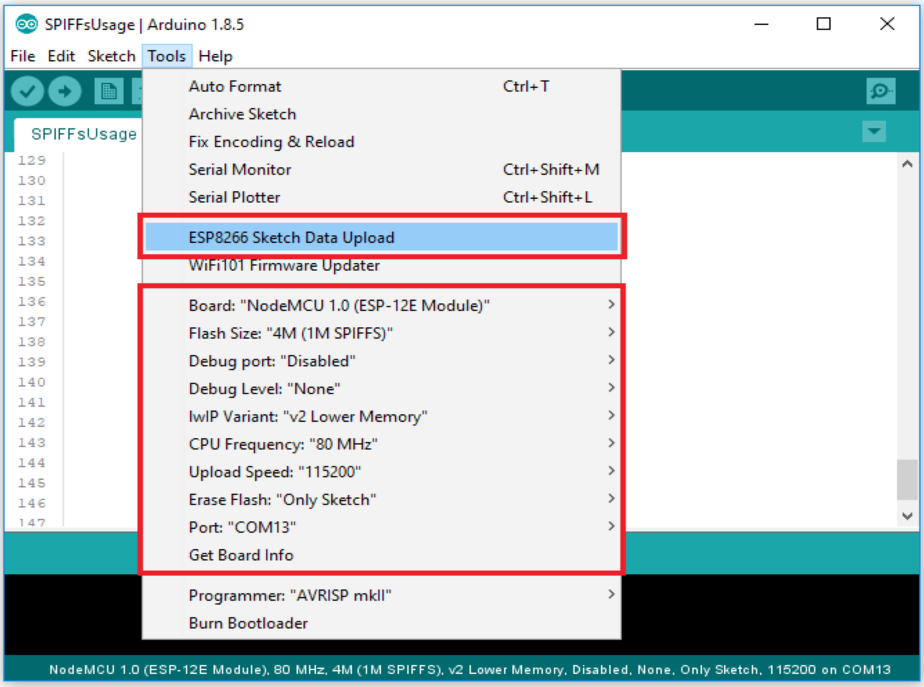
Follow these steps to upload code in the flash:

- Open **Lesson 34: SPIFFS -> SPIFFsUsage** folder.
- Data that is going to be uploaded on the board **MUST** be inside a folder called **data.** It is recommended to be a text file.
- Open **SPIFFsUsage.ino** sketch.
- Connect the development board and make sure the settings in **Tools** tab are correctly configured and select the right communication port. Then click on **ESP8266 Sketch Data Upload.**

IMPORTANT!!! Make sure that the Serial Monitor is closed, otherwise you won't be able to upload data.

Note! If you get the error "SPIFFS Upload failed!", try to reconnect the board.

Note! It takes more time to upload using SPIFFS, so do not open the Serial Monitor while it is uploading.

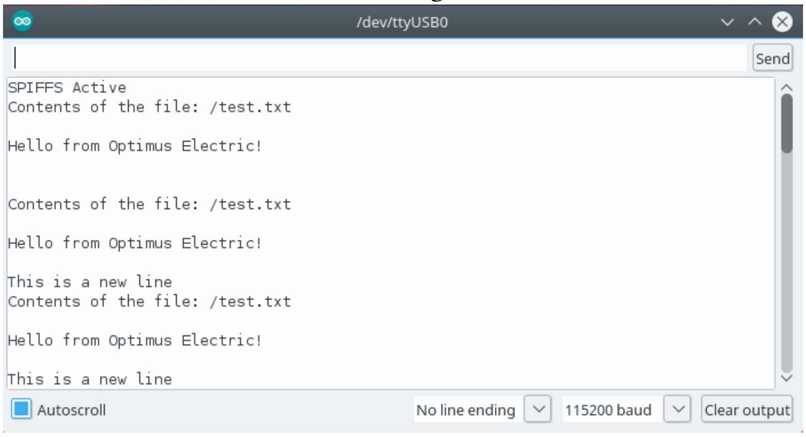

SPIFFsUsage | Arduino 1.8.5

File Edit Sketch Tools Help

Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
ESP8266 Sketch Data Upload
WiFi101 Firmware Updater
Board: "NodeMCU 1.0 (ESP-12E Module)"
Flash Size: "4M (1M SPIFFS)"
Debug port: "Disabled"
Debug Level: "None"
lwIP Variant: "v2 Lower Memory"
CPU Frequency: "80 MHz"
Upload Speed: "115200"
Erase Flash: "Only Sketch"
Port: "COM13"
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader

NodeMCU 1.0 (ESP-12E Module), 80 MHz, 4M (1M SPIFFS), v2 Lower Memory, Disabled, None, Only Sketch, 115200 on COM13

Upload **SPIFFsUsage.ino** on the board, then open serial monitor and set the baud rate at **115200**. You should see something like:


/dev/ttyUSB0

```
SPIFFS Active
Contents of the file: /test.txt

Hello from Optimus Electric!


Contents of the file: /test.txt

Hello from Optimus Electric!

This is a new line
Contents of the file: /test.txt

Hello from Optimus Electric!

This is a new line
```

Autoscroll — No line ending — 115200 baud — Clear output

Below you can find the code explained.

39.4 Code

The code can be found in the folder called **Lesson 34: SPIFFS -> SPIFFsUsage**, and supplimentary details can be found in the comments.

In order to use SPIFFS, you have to include a library **"FS.h"** using the following command:

Code 39.4.1 The SPIFFS library

```
#include "FS.h"
```

Next, you have to initialize the **SPIFFS** and check if it is available, by using an **if** statement.

Code 39.4.2 The setup() function

```
void setup()
{
  //start the Serial communication at 115200 bits/s
  Serial.begin(115200);

  //wait 1 s
  delay(1000);

  //initialise the SPIFFS and display
  //a message in Serial Monitor if it is available or not
  if (SPIFFS.begin())
  {
      Serial.println("SPIFFS Active");
  }
  else
  {
      Serial.println("Unable to activate SPIFFS");
  }

  //wait 2 s
  delay(2000);
}
```

Check the following table to see different commands for SPIFFS:

Command	Explanation
SPIFFS.**exists**(“**name**”);	It is used to check if there is a file called **name** saved in flash
File f=SPIFFS.**open**(“**name**”, "x");	Open a file called **name.** **X** represents the access mode: • **r** : read; • **w** : write (it erases the content of the file); • **a** : append (the same as write but it does not erase the content of the file); • r+ : can both read and write, but it doesn’t create the file if it doesn’t exist; • w+: can both read and write, but it creates a file if it doesn’t exist;
f.**position**();	Returns the position of the pointer in the file
f.**size**();	Returns the size of the file
s.**trim**();	Get a version of the String with any leading and trailing whitespace removed
f.**readStringUntil**();	Reads characters from the serial buffer into a string

In the **loop()** function we have an instruction that checks if there is a file called **test.txt**, and if that is true, then the main block will run. In that block we have an instruction that opens the file, and then in the Serial Monitor, and using the commands from the table above, we will display the content of that file. At the end we will close the opened file.

Code 39.4.3 Open the file

```
      //open the file
      File f = SPIFFS.open("/test.txt", "r");
      if (!f)
      {
        //display a message
        Serial.print("Unable To Open");
      }
      else
      {
        String s;
        Serial.print("Contents of the file: ");
        Serial.print("/test.txt \n\n");

        //f.position() returns the position of the pointer
        //in the file
        //f.size() returns the size of the file
        while (f.position() < f.size())
        {
          //readStringUntil() reads characters from the
          //serial buffer into a string
          s = f.readStringUntil('\n');

          //get a version of the String with any leading
          //and trailing whitespace removed
          s.trim();

          //print the string in Serial Monitor
          Serial.println(s);
        }
        //close the file
        f.close();
      }
```

Next, we will open the file and add a new line at the end of it. Then, the new content of the file will be displayed in Serial Monitor.

Code 39.4.4 Add a new line and display the content in Serial Monitor

```
      //add new lines in the file
      f = SPIFFS.open("/test.txt", "a");
      if (!f)
      {
        Serial.print("Unable To Open");
      }
      else
      {
        //add new line in the file
        f.println("\n");
        f.println("This is a new line");
        f.close();
      }

      //wait 5 s
      delay(5000);

      //display the modified file in Serial Monitor
      f = SPIFFS.open("/test.txt", "r");
      if (!f)
      {
        Serial.print("Unable To Open");
      }
      else
      {
        String s;
        Serial.print("Contents of the file: ");
        Serial.print("/test.txt \n\n");

        while (f.position() < f.size())
        {
          s = f.readStringUntil('\n');
          s.trim();
          Serial.println(s);
        }
        f.close();
      }
```

You can find another example with **SPIFFS** in the folder **Lesson 34: SPIFFS -> Spiffs/HTML_spiffs**. In this example we will store the HTML page on the development board, instead of including the code for the page in the arduino code. We will connect to a wireless network, create a web server and display a simple web page in a browser. Do not forget to upload the data before uploading the sketch. And check if the page is in the **data** folder. Check the code for more details.

And an advanced example is located in the folder **Lesson 34: SPIFFS/Spiffs/ Control_led_web**.