arduino

Ramin Zarebidoky (LiterallyTheOne)

07 Dec 2025

Introduction

Why should we use Arduino

Arduino is one of the greatest and easiest ways to start working with electronics. It has a large and supportive community with tons of examples and projects. It has been a go-to for most of the hobbyists, and it has so many use cases at an industrial level. We can do so many fascinating projects with it, from beginner to advanced levels. To name a few, we can start with a simple **Blink LED Project**, and **Traffic light controller**. For more intermediate projects, we can mention:

- **Smart Plant Watering system**: Check the moisture and pump the water.
- **Automatic Night Lamp**: Use a light sensor to turn on the light when it's dark.
- **Distance measurement**: Use a distance measurement sensor and report the output.

If we want to be more advanced, we can mention:

- **Robotics**: Controlling motors, sensors, etc., and getting commands from other sources.
- **Home Automation**: Control lights, fans, and connect them to apps.
- **Weather station**: Check various sensors like (temperature, humidity, e.t.c) and log it.
- **IOT projects**: Use WI-FI modules to send the sensor outputs to the cloud.
- **Wearables**: Smartwatches, fitness trackers, etc.

What is Arduino

Now we know what can be done with **Arduino**, it is time to understand it better. **Arduino** is an Open-source electronics platform. It consists of both easy-to-use hardware (like **Uno**, **Nano**, **Mega**, etc.) and software (Arduino IDE). It is known as one of the best tools for learning electronics.

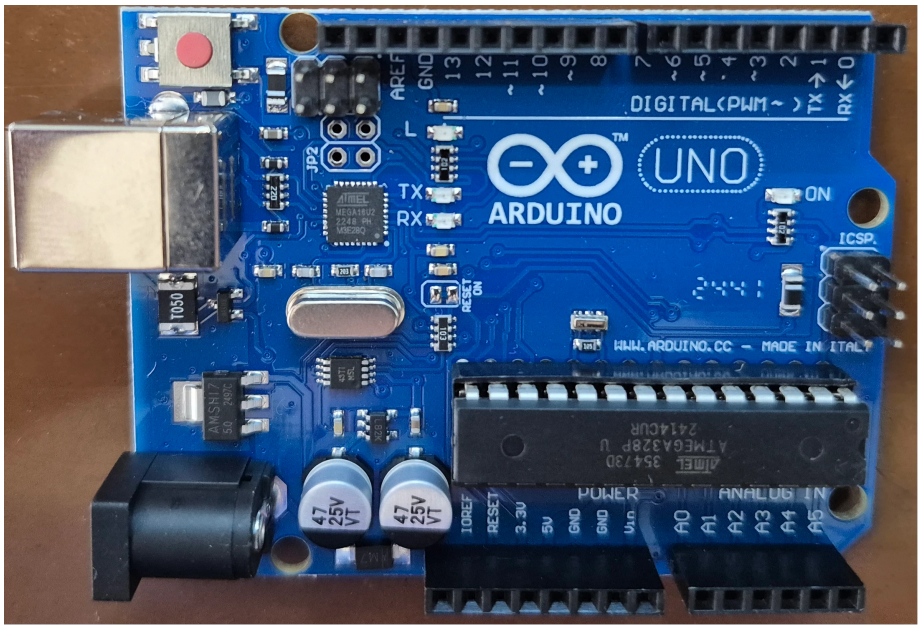


Figure 1: Arduino

In the image above, you can see an **Arduino UNO** board, which is the main focus of this tutorial. As you can see, on the **Arduino UNO's board**, we have a microcontroller. This microcontroller is an **AVR microcontroller**, called **ATMEGA328P**. **Arduino** has made a really great and super-easy-to-use framework for **AVR Microcontrollers** that we are going to use mostly in this tutorial. At the top of this image, you can see there are **14 pins** that we can use for **digital** input and output (D0-D13). Also, there are **6 pins** at the bottom of this image that we can use for **analog** input and output (A0-A5). These pins are called **General-Purpose input/output** (**GPIO**).

Blinking LED

Let's start with a simple project as a **Hello World**. The goal of this project is to connect an **LED** to an **Arduino UNO** in our simulation and make it blink with a certain frequency.

Put an LED on the Board

At first, let's light up an **LED** in **SimulIDE**. To do so, we can follow these steps:

- Open up **SimulIDE**.
- On the left panel, drag an **LED** (Outputs/Leds/LED) and put it on the

board.

- On the left panel, drag a **Fixed Voltage** (Sources/Fixed voltage) and put it on the board.
- On the left panel, drag a **Ground** (Sources/Ground) and put it on the board.
- On the left panel, drag a **Resistor** (Passive/Resistors/Resistor) and put it on the board.
- Connect the **Fixed Voltage** to one of the pins of the **Resistor**.
- Connect the other pin of the **Resistor** to the **anode** (the longest pin) of **LED**.
- Connect the **Ground** to the **cathode** (the shortest pin) of **LED**.
- Click on the **Fixed Voltage** to turn it on.
- Press **Start simulation** in the top panel.

You should have an output like below:

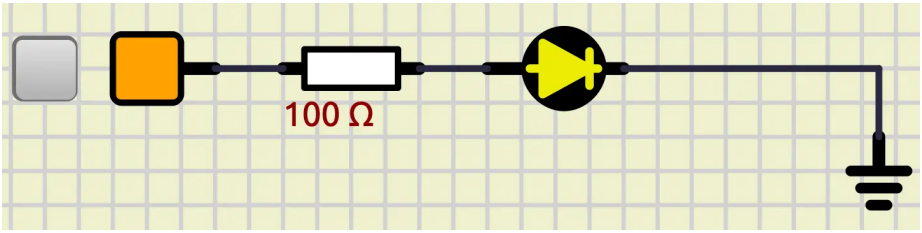


Figure 2: Simple LED

In the image above, we have an **LED** that we can control the state of it using the button on the **Fixed Voltage**.

Connect LED to Arduino

Now, let's put an **Arduino Uno** on the board and control the state of **LED** by that. To do so, we can follow these steps:

- In the left panel, drag an **Arduino UNO** (Micro/Arduino/Uno) and put it on the board.
- Disconnect the **Fixed voltage** from the **Resistor** and connect it to **RST**.
- Let the **Fixed voltage** to be on.
- Connect pin of **Resistor** which was connected to the anode of the **LED**, to the **pin 13**.

You should have something like this:

Create a PlatformIO project

Now, let's create a **PlatformIO project** to be able to program our **Arduino**. To create a new project, you can follow these steps:

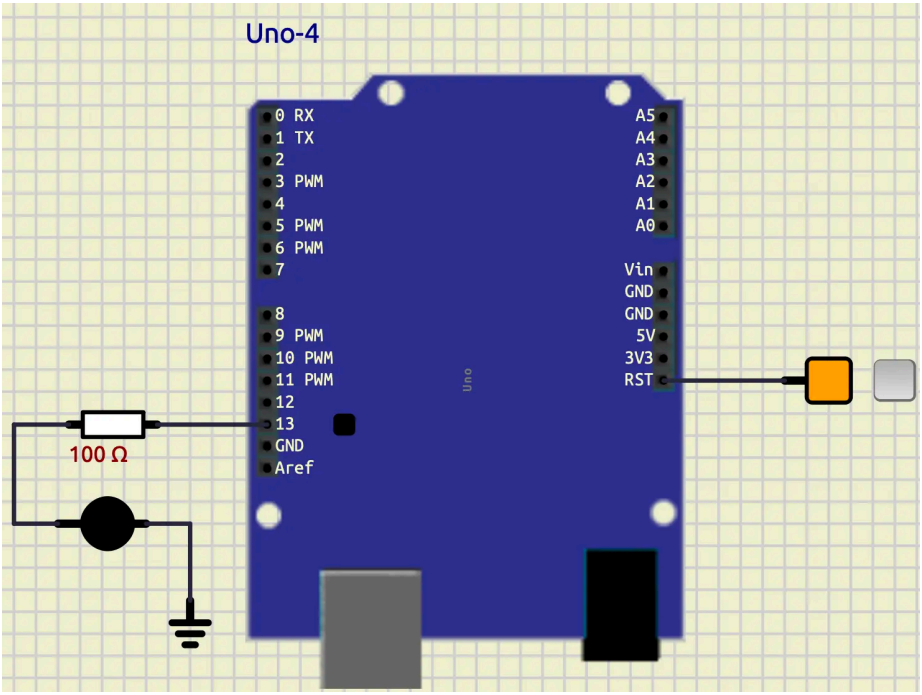


Figure 3: Arduino LED

- Open **VS Code**
- Click on **PlatformIO** on the left panel
- Click on **New project**, A window opens up for you with the title of project wizard
- In the name section, enter the name of your project
- In the **Board** section select **Arduino Uno**
- In the **Framework** section select **Arduino**
- Click on the **Finish button**.

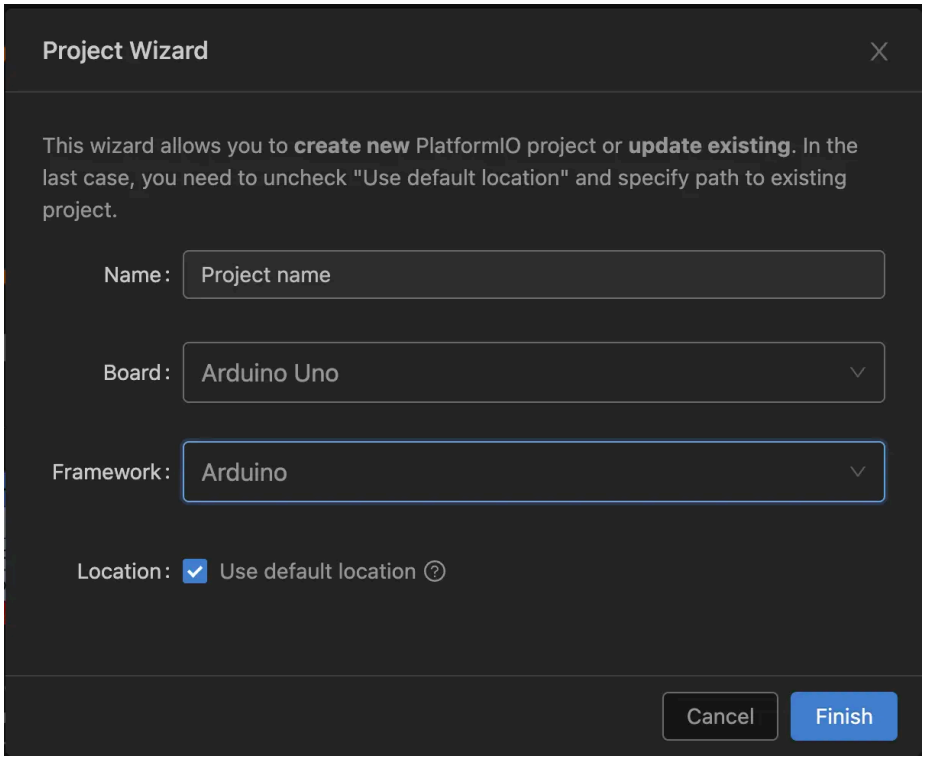


Figure 4: PlatformIO Project

Now you have a project. In the `src` directory, you have a file called `main.cpp`. This file is the file that we are going to write our code in. In this file, you have content like below:

```
#include <Arduino.h>

// put function declarations here:
int myFunction(int, int);

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

```
  int result = myFunction(2, 3);
}

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

// put function definitions here:
int myFunction(int x, int y) {
  return x + y;
}
```

We only need `setup` and `loop` functions, so we are going to remove the rest of it to have a code like below:

```
#include <Arduino.h>

void setup()
{
}

void loop()
{
}
```

`setup` only runs at the start of our program for one time. It is like an initialization. `loop` works like a `while true`. It's always running, and we are going to put the logic of our program on it.

Write the code for the blinking LED

Now, we are ready to write our code. First thing that we should do is to set the **pin mode** of **pin 13**, to output. To do so, we can use the code below:

```
pinMode(13, OUTPUT);
```

This code should be run only once at the start of our program. So, we should put it in the setup. Now, let's write the blinking process. To do that, we can use a function called `digitalWrite`. It takes a **pin** and a **Value**. So, if we write `digitalWrite(13, HIGH)`, it would turn on the **LED**, and if we write `digitalWrite(13, LOW)`, it would turn the **LED** off. Let's put some `delay` between them, and we have a code like below:

```
digitalWrite(13, HIGH);
delay(500);
digitalWrite(13, LOW);
delay(500);
```

As you can see, the code above is the logic of our program, So, we should put that on the `loop` function. As a result, our complete code would look something like this:

```
#include <Arduino.h>

void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  delay(500);
  digitalWrite(13, LOW);
  delay(500);
}
```

Build and upload firmware

Now, let's build our project and see the result on our simulation. To build our **PlatformIO** project, we can use the **tick** button on the **top right** or **bottom left** of **VS Code**. When it is built successfully, there would be a `firmware.hex` file in `.pio/build/uno/`. We need that file to upload to our simulation. To do so, we can follow these steps:

- Go to the **SimulIDE**.
- Right-click on `Arduino Uno`.
- In `mega328`, click on load firmware.
- Select the path to the `firmware` that you have built previously (You might need to show hidden files).
- Press start simulation.

You should have an output like below:

Conclusion

In this tutorial, we have introduced **Arduino**. Then we built a **Hello World** project that showcases how we can make a **Blinking LED**. In the future, we are going to learn more about **GPIO**, and we do more complicated projects together.

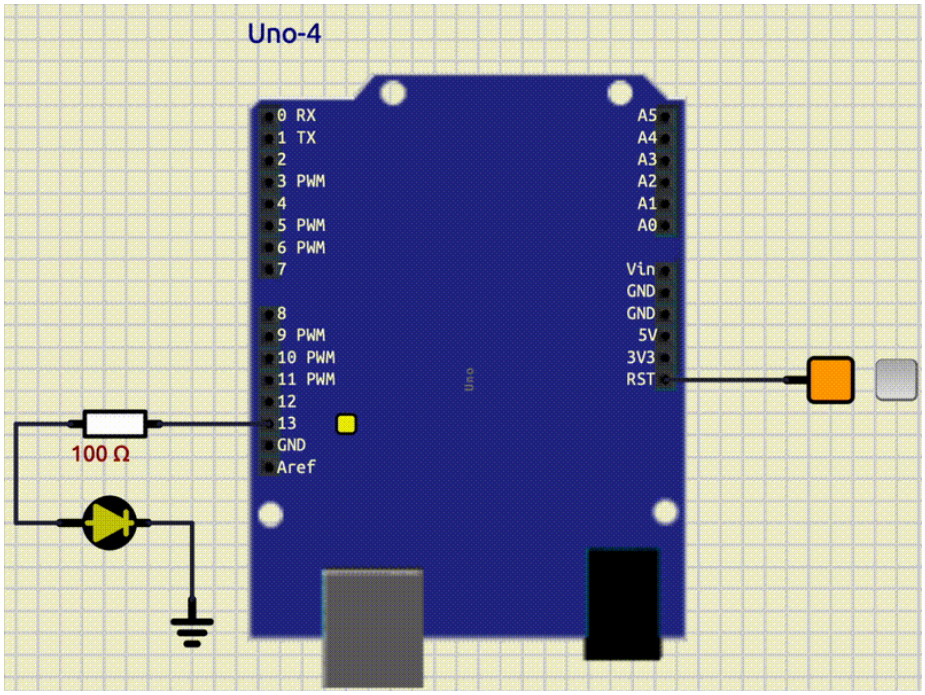


Figure 5: Arduino LED Blink

GPIO

Introduction

In the previous tutorial, we learned about the basics of **Arduino**. We implemented a simple **Blink LED** project as a **Hello World**. In this tutorial, we are learning more about **General-Purpose Input/Output** pins in **Arduino**.

What is GPIO?

GPIO stands for **General-purpose Input/Output**. There are pins in **Arduino** that we can determine if they should be input or output using code.

- Input pin: We **read** data from it.
- Output pin: We **write** data to it.

There are **14 Digital GPIO pins** and **6 Analog GPIO pins**.
In the image below, we have highlighted **Digital GPIO pins** with red, and **Analog GPIO pins** with yellow.

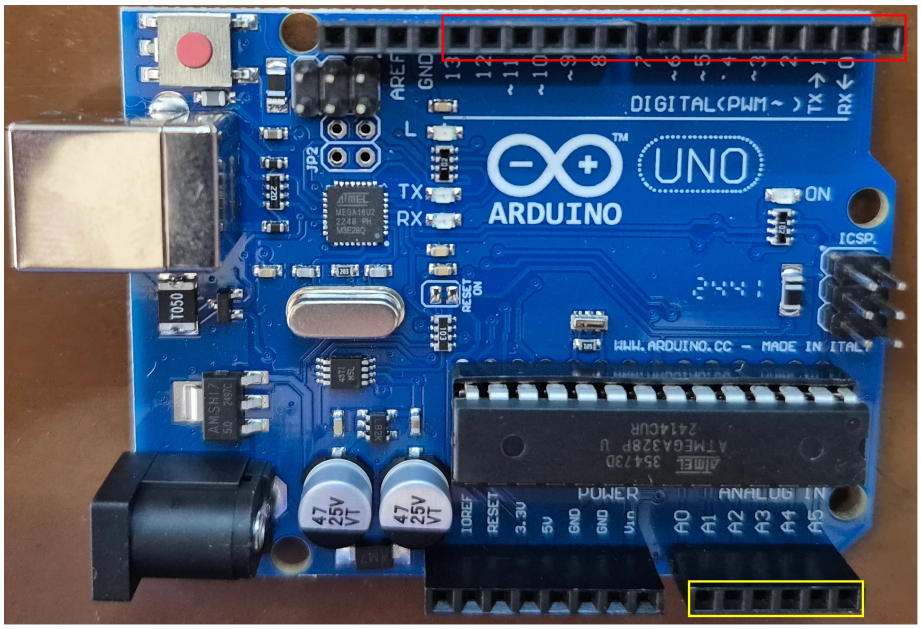


Figure 6: Arduino GPIO

To change the mode of a pin in code, we can use the function below:

```
pinMode(pin_number, mode);
```

- `pin_number`: number of the pin that we want to change (e.g., 13)

- `mode`: we can choose the mode of the pin; two of the choices are:
  - `INPUT`: 0
  - `OUTPUT`: 1

In the previous tutorial, we set the mode of **pin 13** to output, with the code below:

```
pinMode(13, OUTPUT);
```

If we want to change the mode of **pin 3** to input, we can write a code like below:

```
pinMode(3, INPUT);
```

LED and key

Now, let's do a project that turns on an LED whenever we press a key.

Connect an LED and a key to an Arduino

First, let's open up **SimulIDE**. Then we can follow these steps to connect an LED and a key to an **Arduino Uno**:

- Put an **Arduino Uno** on the board.
- Put a **fixed voltage** on the board.
- Connect the **fixed voltage** to the reset pin of the **Arduino Uno** and let it be **on**.
- Put an **LED** on the board.
- Put a **Resistor** on the board.
- Connect the **anode** (tallest pin) of the **LED** to on pin of the **Resistor**.
- Connect the other pin of the **Resistor** to the **pin 13** of **Arduino Uno**.
- Put a **ground** on the board.
- Connect the **cathode** (shortest pin) of the **LED** to the **ground**.
- Put a **button** (Switches/push) on the board.
- Put another **fixed voltage** on the board.
- Connect one pin of the **button** to the **fixed voltage**.
- Connect the other pin to the **pin 2** of **Arduino**.

After the steps above, you should have something like this:

With this setup, we have **pin 13** as an **output pin** and **pin 2** as an **input pin**.

Write the code

Now, let's write the code that turns the LED on and off anytime we press it. At first, we should set the mode of **pin 13** to **output**.

```
pinMode(13, OUTPUT);
```

Next, we should set the mode of **pin 2** to **input**.

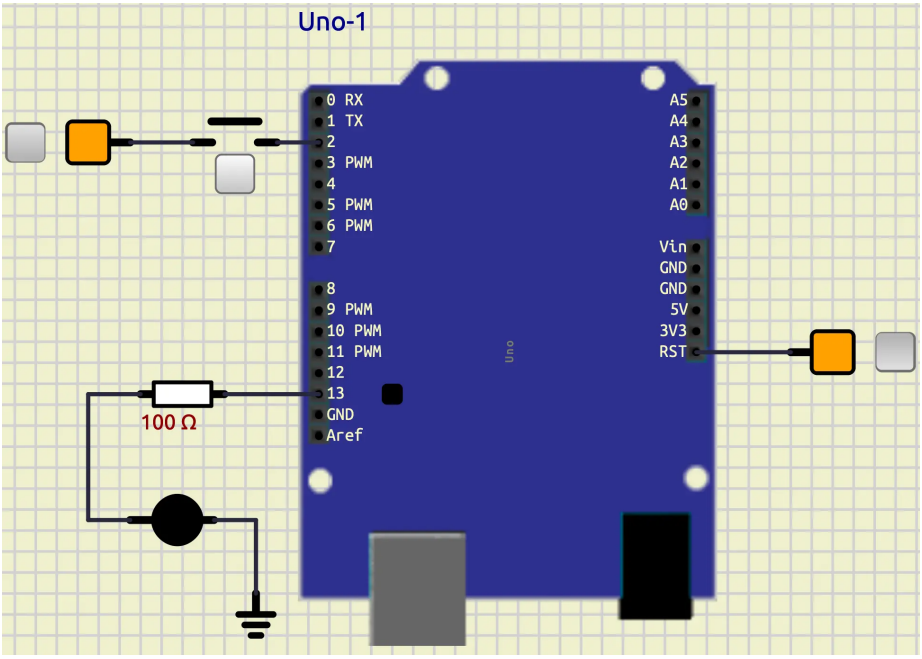


Figure 7: Arduino LED Button

```
pinMode(2, INPUT);
```

These codes are for initialization, so we should put them in `setup()` function. Now, let's get ready to write our logic. We need two variables:

- `led_state`: keeps the state of the LED (on/off).
- `button_pressed`: becomes `true` if the button is pressed.

So, let's define those variables as global variables. (below the includes)

```
bool led_state = false;
bool button_pressed = false;
```

Our logic is:

- Check if the button is pressed
- If it was pressed:
  - Change the `led_state`.
  - Write the `led_state` to output.

So, let's write it down.

```
button_pressed = digitalRead(2);
if (button_pressed)
```

```
{
    led_state = !led_state;
    digitalWrite(13, led_state);
    delay(500);
}
```

In the code above, we follow the logic that we have explained. The reason that we put `delay(500)` is simple. When we press the key down and release it, it takes about `200ms`. In this `200ms`, `button_pressed` is `true`, so our code in the `if` statement would be executed many times. To avoid this from happening, we put a `delay(500)`. You can see the full code here:

```
#include <Arduino.h>

bool led_state = false;
bool button_pressed = false;

void setup()
{
  pinMode(13, OUTPUT);
  pinMode(2, INPUT);
}

void loop()
{
  button_pressed = digitalRead(2);
  if (button_pressed)
  {
    led_state = !led_state;
    digitalWrite(13, led_state);
    delay(500);
  }
}
```

The output would be something like below:

Control three LEDs

Now we know how to set a pin as an input and output. Let's add more LEDs to the board and change our program in a way that whenever we press the button, the LED that is turned on switches. Also, in our simulation, we can control the color of the LEDs. Let's change their colors to **red**, **yellow**, and **green**, like a stoplight. You are free to connect the LEDs to whichever pin you want. Your output should be something like this:

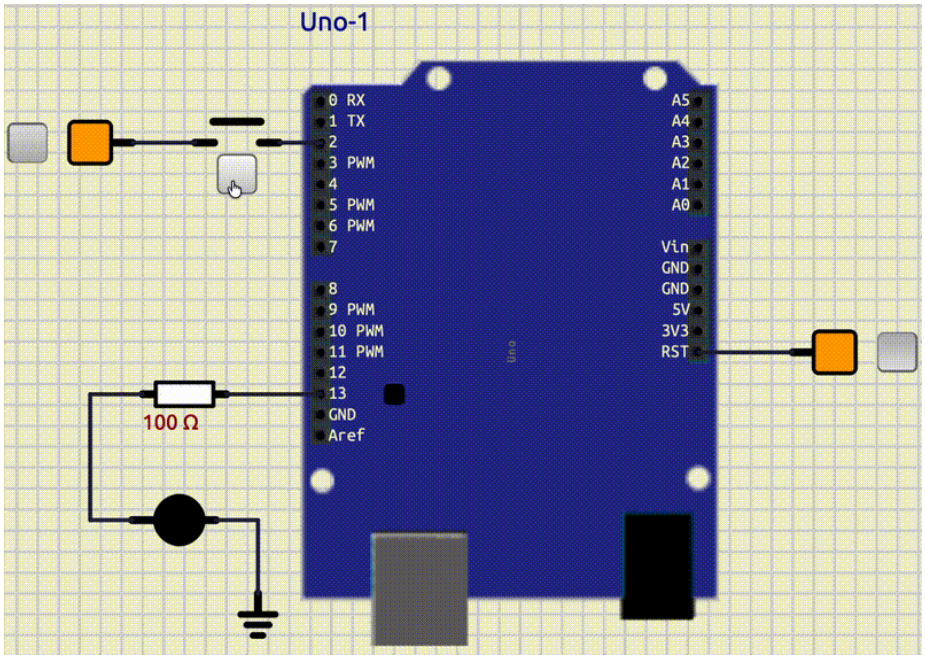


Figure 8: Arduino LED Button On and Off

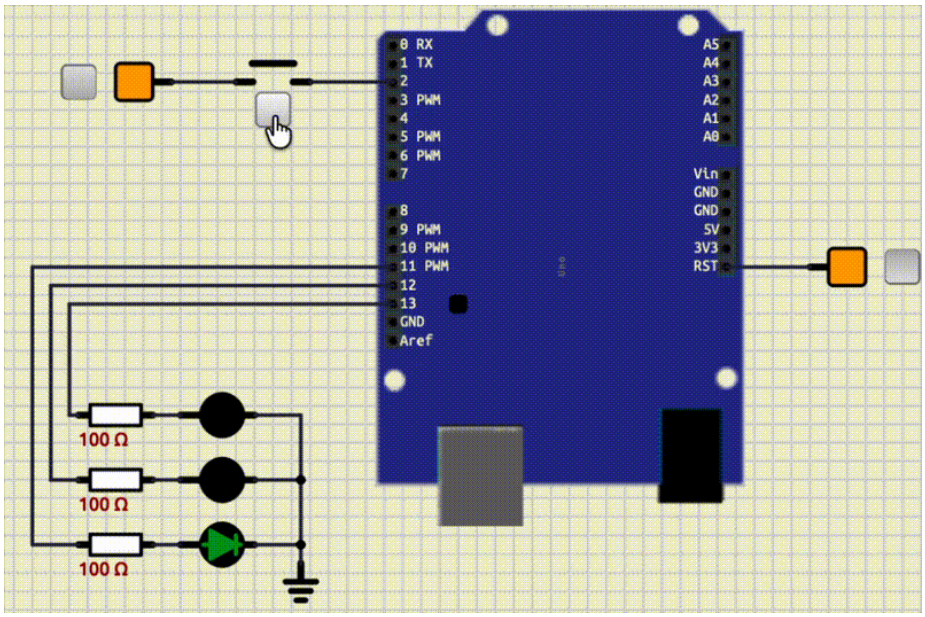


Figure 9: Traffic Light

Add another button

Now, let's add another button that could disable the traffic light, and when we press it again, it would enable it again. Your output should be like this:

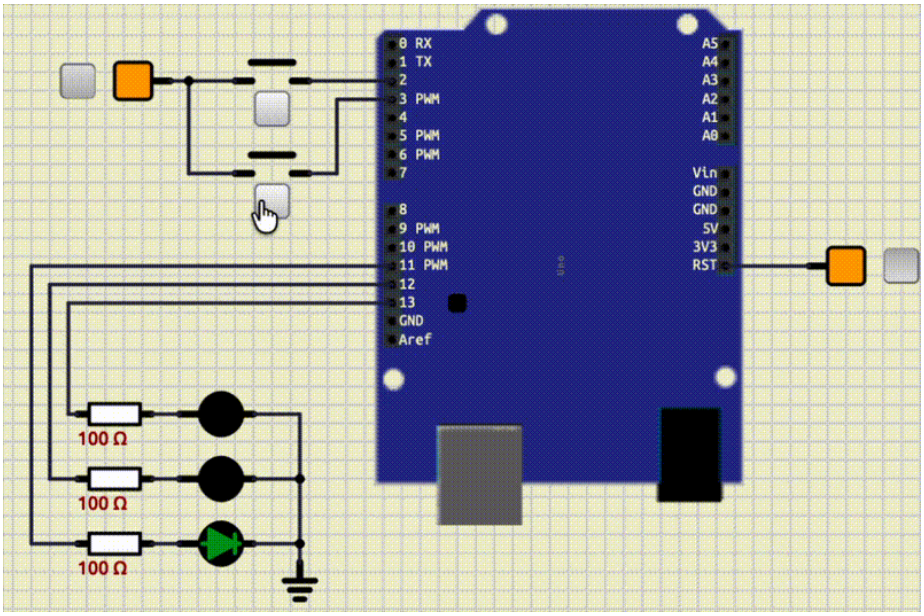


Figure 10: Traffic light two buttons

Conclusion

In this tutorial, we learned how to use **Digital GPIO**. We have introduced all the pins that you can use for **Digital GPIO** and how to program them in order to work correctly. We walked you through how to manage and program a simple LED and button. Then, we asked you to expand the number of LEDs to learn the **Output GPIO** better. Finally, we asked you to add another button to make you more comfortable with **Input GPIO**.

7-Segment

Introduction

In the previous tutorial, we learned about **GPIO**. We managed to work with LEDs and buttons. In this tutorial, we are going to work with **7-segment**.

What is a 7-segment?

A 7-segment is a set of 7 LEDs that can be used to show numbers and some letters. Each LED is called a segment. The segments are named from **a** to **g**. Some 7-segments have a dot that is called **dp** (dot point).

We can find a **7-segment** in output/LEDs/7 Segment.

Numbers on 7-segment

At first, let's put a 7-segment on the board (output/LEDs/7 Segment) and connect all of its pins to fixed voltages like below:

In the image above, we have connected all the pins of the 7-segment to the fixed voltages. In order for the 7-segment to work, we should let the `-` pin have a low value.

Now we can make numbers by turning on and off the segments. For example, if we want to make the number `0`, we should turn on all the segments except `g` and `dp`. (Make sure that you are in a simulation mode.) We are having something like this:

Now we can make all the numbers and store them in the format below:

dp	g	f	e	d	c	b	a
0	0	1	1	1	1	1	1

Connect a 7-segment to an Arduino

Now, let's add our **7-segment** to an **Arduino**. To do that, we can follow these steps:

- Put an **Arduino Uno** on the board.
- Connect the 8 LED pins of the **7-segment** (**a** to **g** and **dp**) to **0-7** pins of **Arduino**.

You should have something like this:

Write a counter

Now that we have all the numbers calculated, let's put them in a global array like below:

```
char digits[10] = {
    0b00111111, // 0
    ...
};
```

After doing that, let's write a function that can be used to display those digits.

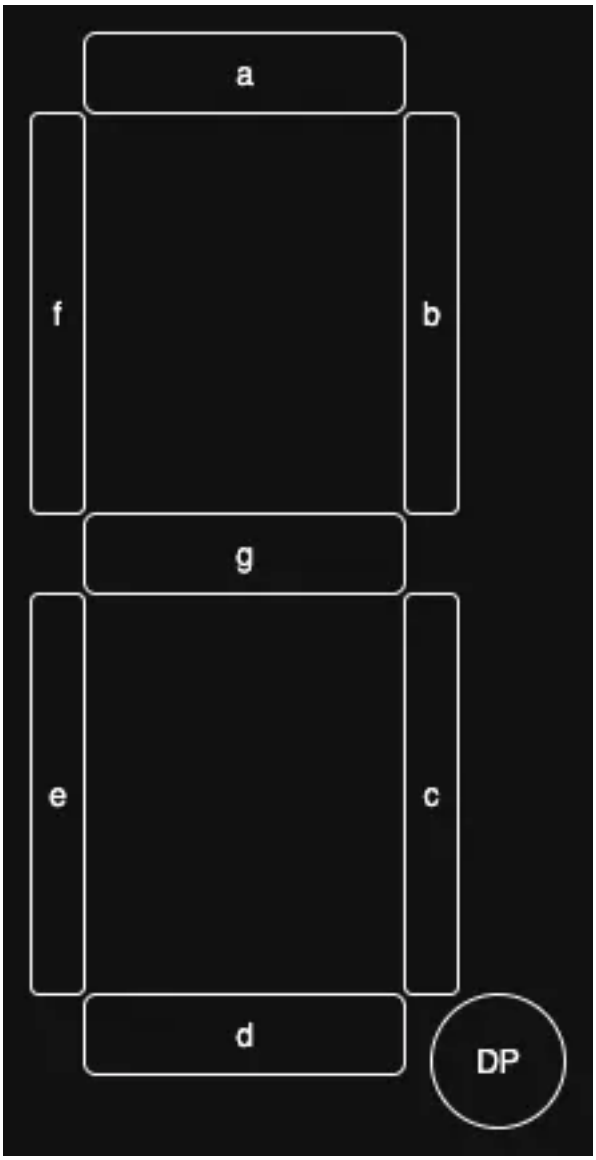


Figure 11: 7segments

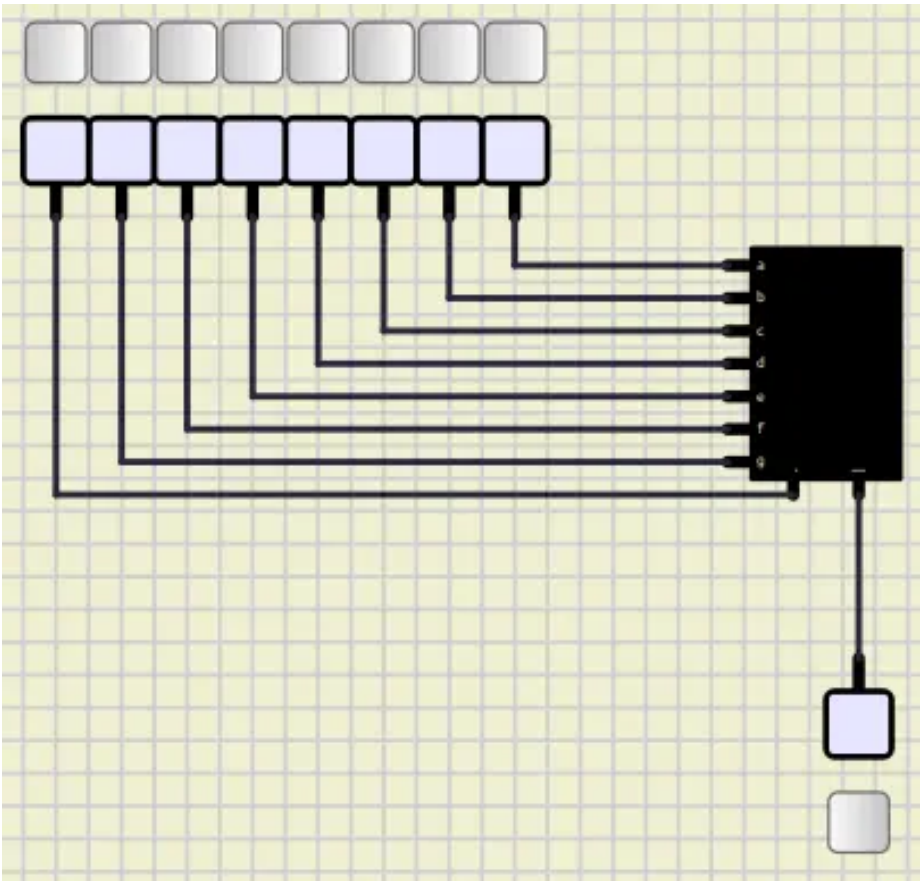


Figure 12: s2_7segment_fixed_voltage

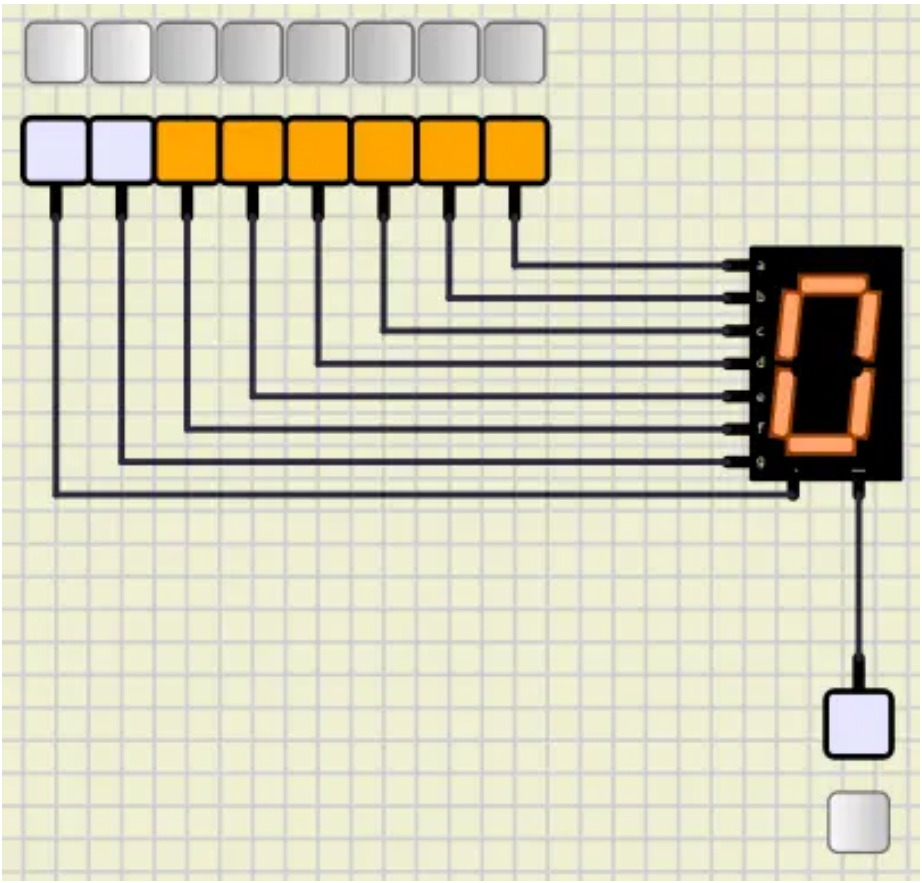


Figure 13: s2_7segment_0

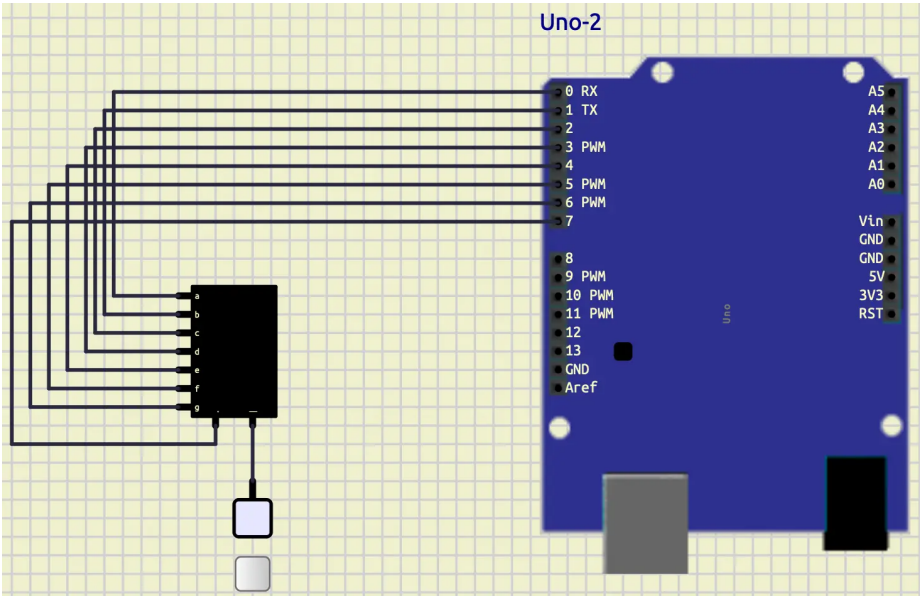


Figure 14: Arduino 7segment

```
void show_digit(int digit)
{
  for (int i = 0; i < 8; i++)
  {
    digitalWrite(i, (digits[digit] >> i) & 0b0000'0001);
  }
}
```

In the code above, we have a function called `show_digit` which takes a `digit` as its argument and uses `digits` (the global array that we have calculated) and the `digitalWrite` function to write them in the respective pin. We have connected our **7-segment** to pins from **0** to **7**, so we should write in those pins. To find out what we should write on them, we can explain it with an example. Imagine that you want to write `0b0110'0101` on these pins. You should write something like this:

pin 7	pin 6	pin 5	pin 4	pin 3	pin 2	pin 1	pin 0
0	1	1	0	0	1	0	1

So, if we shift `0b0110'0101` by `0`, we have: `0b0110'0101`. If we and it with `0b0000'0001` we would have: `0b0000'0001`. That was the result for `pin 0` that we wanted. Now, let's do it for `pin 1` as well. We are going to shift `0b0110'0101`

by `1`, so the result would be: `0b0011'0010`. Next, we and it with `0b0000'0001`, we are going to have: `0b0000'0000`. As you can see, we can do that for all the pins and write the value of the pin on it. The process for **pin i** would be like below:

- Shift the value by `i`
- And it with `0b0000'0001`

Now that we have a function to write any digit that we want on the **7-segment**, you are able to write a counter. Your output should look something like this:

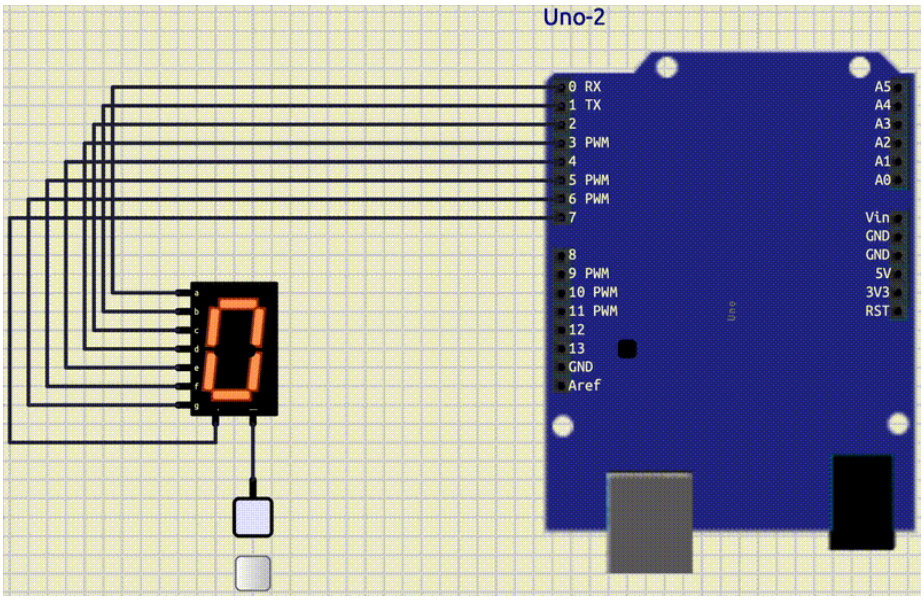


Figure 15: Arduino 7segment counter

Button to pause the counting

We want you to add a button to pause the counting. When we hold that button, the number on the **7-segment** should be frozen, and when we release it, it should start counting again from that number. Your output should be something like this:

> As you might have noticed, we should use **pressing** instead of **clicking**. The reason for that is that counting and clicking can't work simultaneously. If we want to rely on only one click, we might miss it. We are going to fix that when we explain about interrupts.

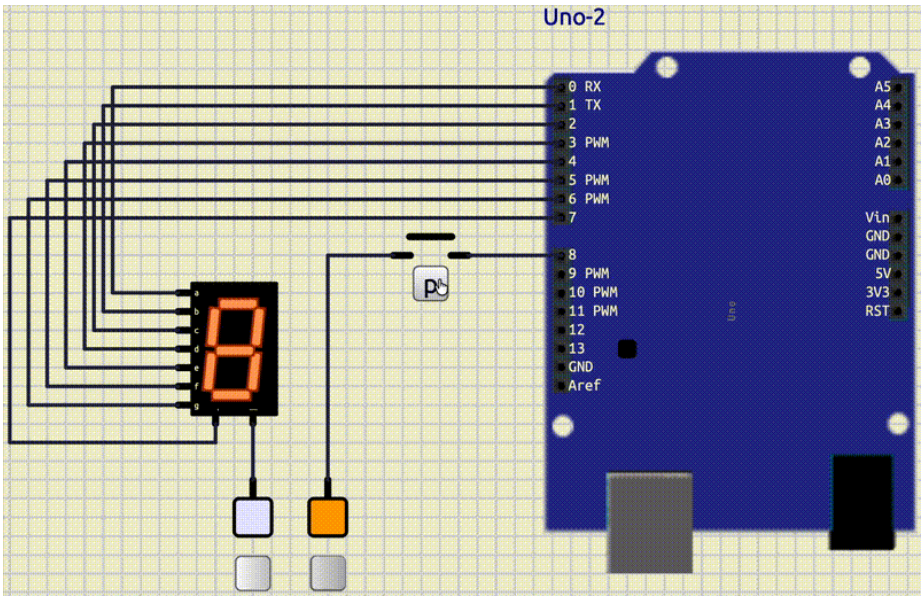


Figure 16: Arduino 7-segment counter pause

Button to reverse the counting

We want you to add a button to reverse the counting. When we hold that button, the numbers should go up, instead of down. When we release it, it would do the counting normally, as it would. Your output should be like this:

Conclusion

In this tutorial, we learned about **7-segment** and one of the ways that we can connect to an **Arduino**. First, we calculated the numbers by connecting **fixed voltage** to each pin of the **7-segment**. Then, we programmed a counter. After that, we added a pause button. Finally, we added a reverse button.

LCD and Keypad

Introduction

In the previous tutorial, we have discussed **7-segment**. Now, we are going to learn how to work with **LCD** and **Keypad**.

LCD

LCD is used to write parameters and status. We have a `16x2 LCD` (16 columns and 2 rows). It has 16-pins.

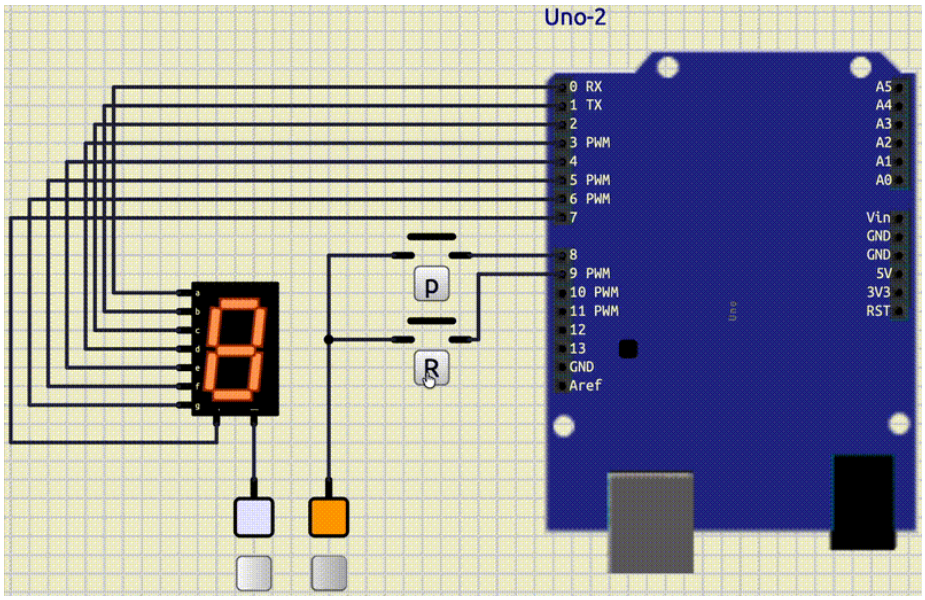


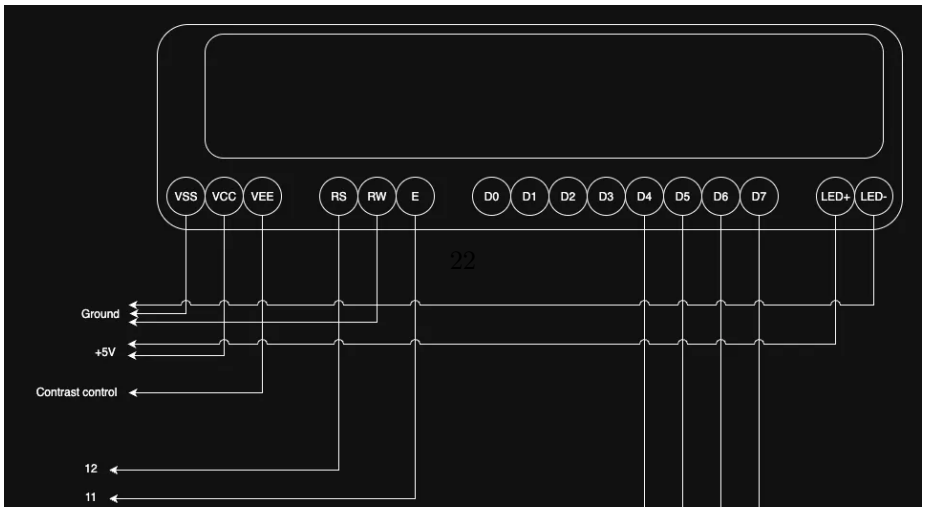
Figure 17: Arduino 7-segment counter pause reverse

VSS	VCC	VEE
Ground	5V+	Contrast Control

RS	RW	E
Register select	Read / Write	Enable

D0	D1	D2	D3	D4	D5	D6	D7
Data pin 0	Data pin 1	Data pin 2	Data pin 3	Data pin 4	Data pin 5	Data pin 6	Data pin 7

LED+	LED-
LED 5V+	LED Ground



Connect an LCD to an Arduino in SimulIDE

Let's put an **LCD** on the board. To do that we can use **Outputs/Displays/HD44780**. After putting that on the board, we should connect the pins of it like below:

- RS: 12
- RW: ground
- E: 11
- D4: 5
- D6: 4
- D7: 3
- D8: 2

The result should be something like below:

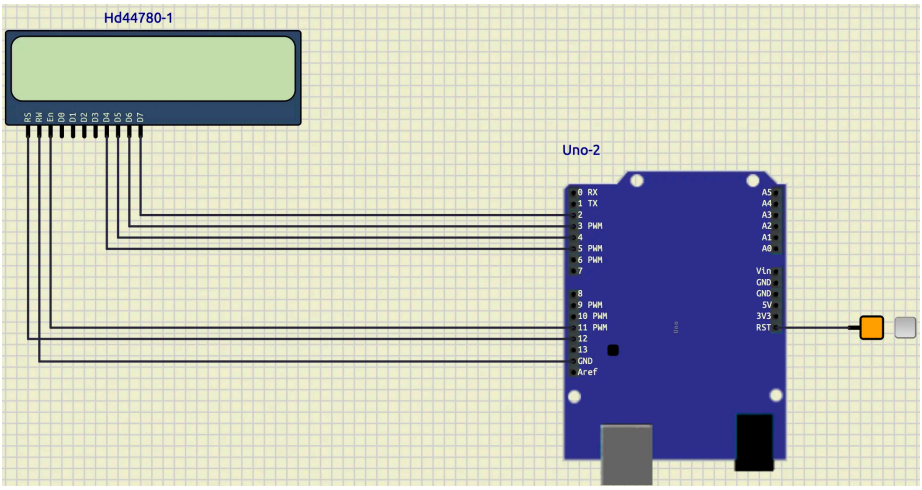


Figure 19: LCD Arduino SimulIDE

Liquid Crystal

Liquid Crystal is a well-known package that helps us to work with **LCD**. This package contain so many great and useful functions. You can see the list of all functions in this link.

Add LiquidCrystal to PlatformIO

To add **LiquidCrystal** to a **PlatformIO** project we should add this code to `platformio.ini`.

```
lib_deps =
  arduino-libraries/LiquidCrystal
```

This code will download the **LiquidCrystal** and then you can import it in your `main.cpp` like below:

```
#include <LiquidCrystal.h>
```

Make an LCD Object

Now, let's make an **LCD** object using **LiquidCrystal**. To do so, we can use the code below:

```
const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
```

In the code above, at first we defined the pins in a way that we connected them to our **Arduino Uno**. Then, we made an object of `LiquidCrystal` with those pins with the name of `lcd`. We put that code on top of the `setup` and `loop` function to be global. Now, let's talk about the functions that we can use for this object.

`begin`

This function initializes the **LCD**. It automatically sets all the pin modes and should be called before any other **LCD** commands.

The usual syntax that we use:

```
lcd.begin(cols, rows);
```

Example:

```
lcd.begin(16, 2);
```

`write`

Write a character on the **LCD**.

Syntax:

```
lcd.write(ch);
```

Example:

```
lcd.write('h');
```

`print`

Write a text on the **LCD**.

Syntax:

```
lcd.print(text);
```

Example:

```
lcd.print("Hello World!");
```

setCursor

Jumps to the given column and row.

Syntax:

```
lcd.setCursor(col, row);
```

Example:

```
lcd.setCursor(5, 1);
```

clear

Clears the **LCD** and jumps to the start.

```
lcd.clear();
```

Example:

```
lcd.clear();
```

LCD Hello World

Now, that we know about **Liquid Crystal** and the functions that we can use, let's write a simple example. We want our **LCD** to show **hello** at the first row, starting from the 3rd column. After some pause, clear the hello and show **world** at the second row starting from the 4th column. Your output should be something like below:

Keypad

Keypad is a series of keys in a matrix. It contains of keys in rows and columns. For example, if we have 4 rows and 3 columns we would have 12 keys. The advantage that a keypad gives us is that we don't require to occupy 12 pins to control 12 keys. Instead, we only need 7 pins, 4 for rows and 3 for columns. Here is an example of a **Keypad** in **SimulIDE**.

Add Keypad to Arduino Uno in SimulIDE

Now, let's add a keypad to our **Arduino Uno** in our **SimulIDE**. We can access to the **Keypad** in **Switches/Keypad**. When you put it on the board, you can

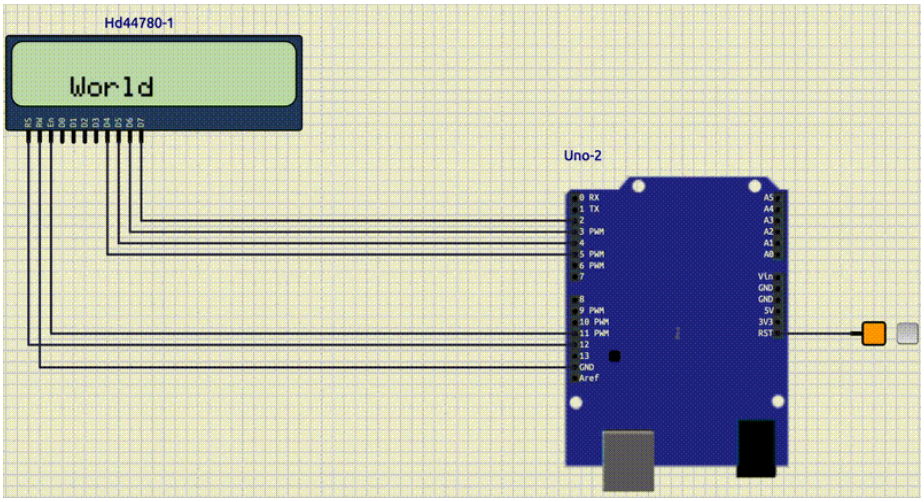


Figure 20: LCD Hello World

Figure 21: Keypad in SimulIDE

see there are **4 rows** and **3 columns**. Let's connect each row and column like below:

- row-0: 0
- row-1: 1
- row-2: 6
- row-3: 7
- col-0: 8
- col-1: 9
- col-2: 10

You should have something like this:

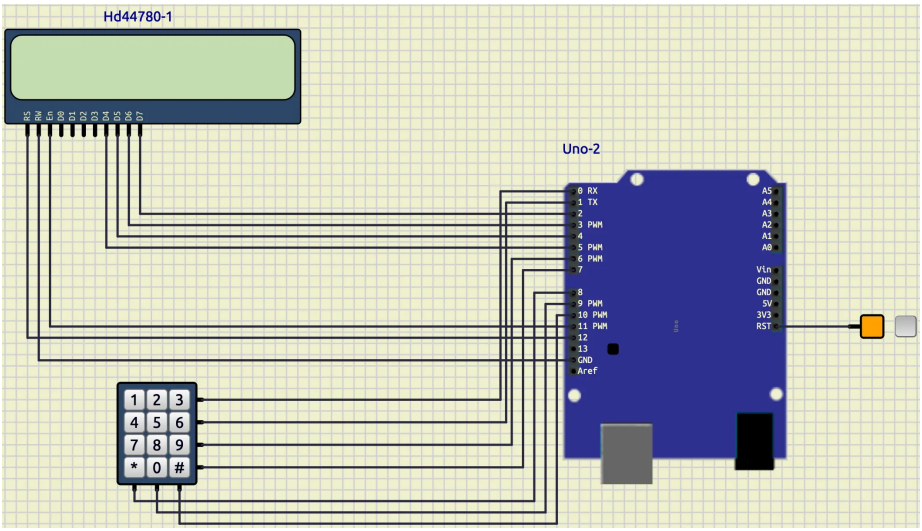


Figure 22: Keypad Connected into an Arduino Uno

Add Keypad to PlatformIO

To use **Keypad** in Arduino, we can use a library called **Keypad**. We can simply add **Keypad** to our **PlatformIO** project by adding it to the **lib_deps** in **platformio.ini**, like below:

```
lib_deps =
  ...
  Keypad
```

Now we are able to import it like in our **main.cpp** like below:

```
#include <Keypad.h>
```

Make a keypad Object

To create a object for our **Keypad** we should at first define a matrix for our keys like below:

```
const byte ROWS = 4;
const byte COLS = 3;

char keys[ROWS][COLS] = {
    {'1', '2', '3'},
    {'4', '5', '6'},
    {'7', '8', '9'},
    {'*', '0', '#'}};
```

As you can see, we have defined two constants called **ROWS** and **COLS** which each of them represent how many rows and columns we have. Then we defined our matrix of keys in a way that we have them in our **Keypad**. Now, it's time to define which pins are connected to each row and column. To do that we can write something like below:

```
byte rowPins[ROWS] = {0, 1, 6, 7};
byte colPins[COLS] = {8, 9, 10};
```

As you recall, in our simulation we have connected our rows to the pins of **0, 1, 6, 7** and columns to **8, 9, 10**. We defined **rowPins** and **colPins** to have those values. Now, let's define our **Keypad Object**. To do so, we can use the code below:

```
Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins, ROWS,
 ↪  COLS);
```

As you can see, in the code above, we have used `Keypad` class to create an instance of a **Keypad**. For the first argument, we changed our `keys` which were in a matrix to the format that `Keypad` likes, using `makeKeymap` function. Then for the other arguments, we gave `rowPins`, `colPins`, `ROWS`, and `Cols` respectively. We named our object `keypad` which we are going to use it later. Now, let's talk about the function that we are mostly going to use.

`getKey`

A function that checks if any keys is pressed. If a key was pressed, it would return the respective key and if not it would return `0`.

Syntax:

```
keypad.getKey();
```

Example:

```
char key = keypad.getKey();
```

Keypad to LCD

Now, let's write a program that displays the output of the **KeyPad** on the **LCD**. Your output should be like this:

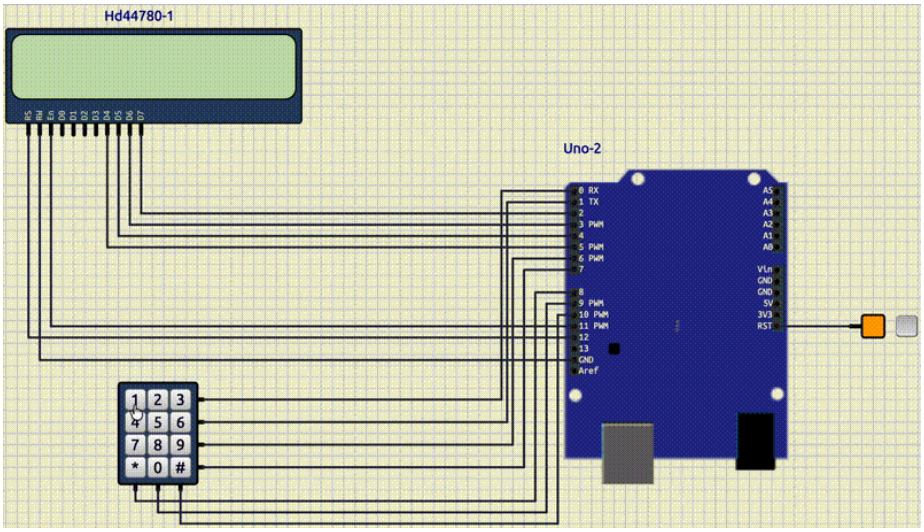


Figure 23: LCD Keypad

Login system

Now, let's write a login system. We have a username and password stored in our code (hard codded). We want user to enter a username and password. After that we are going to check if he has entered the correct ones or not. If it was correct we tell him it is correct, otherwise we tell him it was incorrect. Your output should be something like below:

- Correct username: `12`
- Correct password: `5662`
- Enter: `#`

Conclusion

In this tutorial, we learned how to use **LCD** and **Keypad** in **Arduino Uno**. First, we explained what **LCD** is and how to use **LiquidCrystal** to control it. Then we explained the most important functions and provided an example. After that, we talked about **Keypad**. We introduced a library with the same

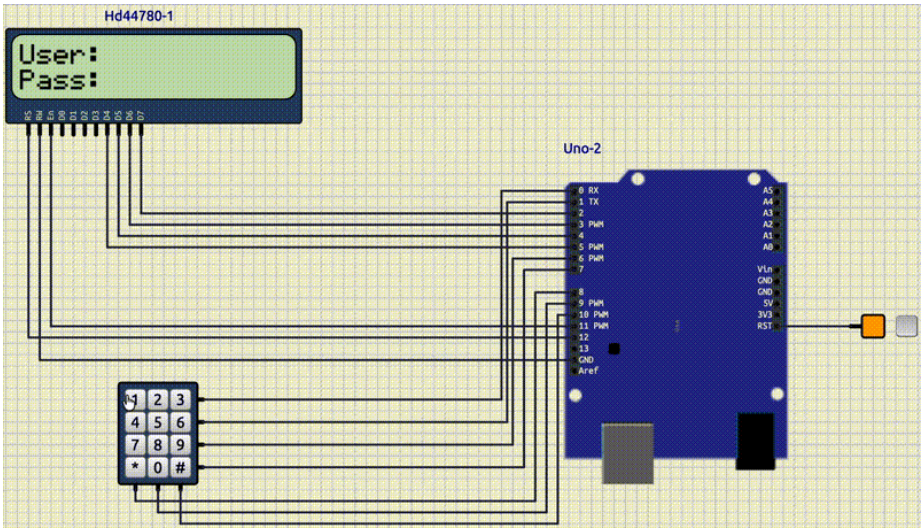


Figure 24: Login System

name to control it. Then, we combined **LCD** and **Keypad**. Finally, we provided an example to make you understand these concepts better.

Serial Communication

Introduction

In the previous tutorial, we learned how to work with **LCD** and **Keypad**. In this tutorial, we are going to learn about **Serial Communication** and some of its usages.

Serial communication

Serial communication is a way that a microcontroller can send and receive data one bit at a time. You can use it to communicate with computers, microcontrollers, and modules (e.g., GPS, Bluetooth, ESP8266). **Serial communication** is one of the most important concepts in microcontrollers. **Arduino Uno** uses **UART** (Universal Asynchronous Receiver-Transmitter) to handle the **Serial communication**. **UART** needs two pins, one for receiving data (RX) and one for transmitting data (TX). These two pins are available in **Arduino Uno** in **pin 0** (RX) and **pin 1** (TX). Also, we can have **Serial Communication** with **USB** as well. One of the most important things in having a **Serial Communication** is setting the correct **baud rate** for both of the devices that are trying to communicate. **Baud rate** indicates the speed of data transfer. The reason that **baud rate** should be the same for both devices is that, we have an

asynchronous communication. The start and the end of the communication are determined with **start bit** and **end bit**.

Serial Terminal on SimulIDE

One of the ways that we can use **Serial communication** is by using a **Serial Terminal**. You can access a **Serial Terminal** in **Micro/Peripherals/Serial Terminal**. Now, let's put a **Serial Terminal** on the board and connect it to an **Arduino Uno**. To do that, we should wire them like below:

- **TX** of **Arduino** -> **RX** of **Serial Terminal**
- **RX** of **Arduino** -> **TX** of **Serial Terminal**

You should have something like this:

Serial Hello World

Now, let's create a **PlatformIO** project and write a **Hello World** for **Serial communication**. At first, let's initialize the **Serial Communication**. To do so, we can use the code below:

```
Serial.begin(9600);
```

In the code above, we set the **baud rate** of our **Serial communication** (default **baud rate** in **Serial Terminal** in **SimulIDE**) to 9600 and initialize the **Serial communication**. (To change the **baud rate** of the **Serial Terminal** in **SimulIDE** you can go to the properties of that **Serial Terminal**). Now, we are ready to write something on it. To do so, we can use the code below:

```
Serial.println("Hello World");
```

In the code above, we have printed `Hello World` into the serial port. The function `println`, prints the given input and makes a new line. Now, let's change our code in a way that it prints `Hello World` every one second. So we have the full code like below:

```
#include <Arduino.h>

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

void loop()
{
  Serial.println("Hello World");
  delay(1000);
}
```

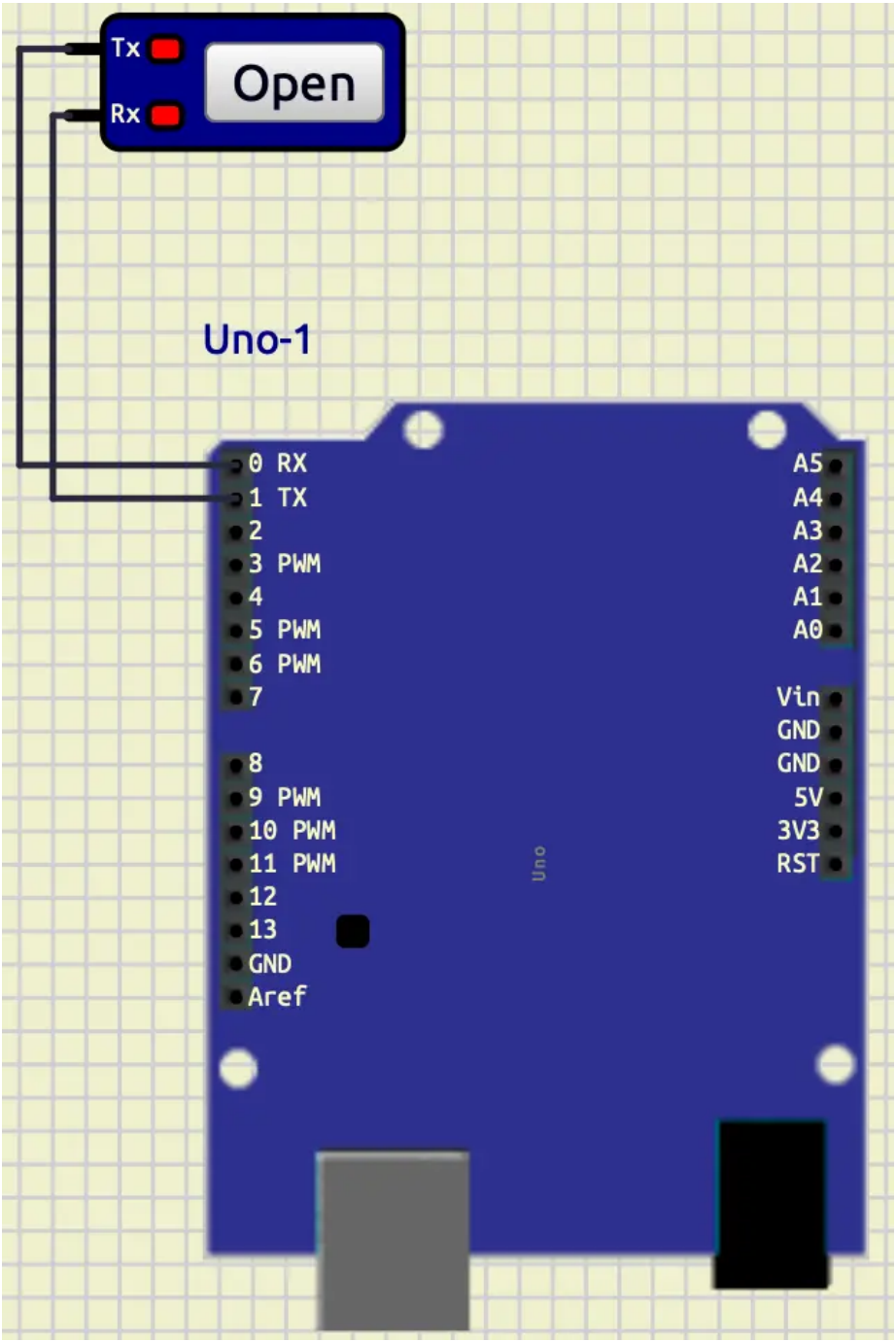


Figure 25: Serial Terminal

Let's upload it into our **SimulIDE**. After pressing start simulation, you should click on the **open** button on the **Serial Terminal**. Your output would be something like below:

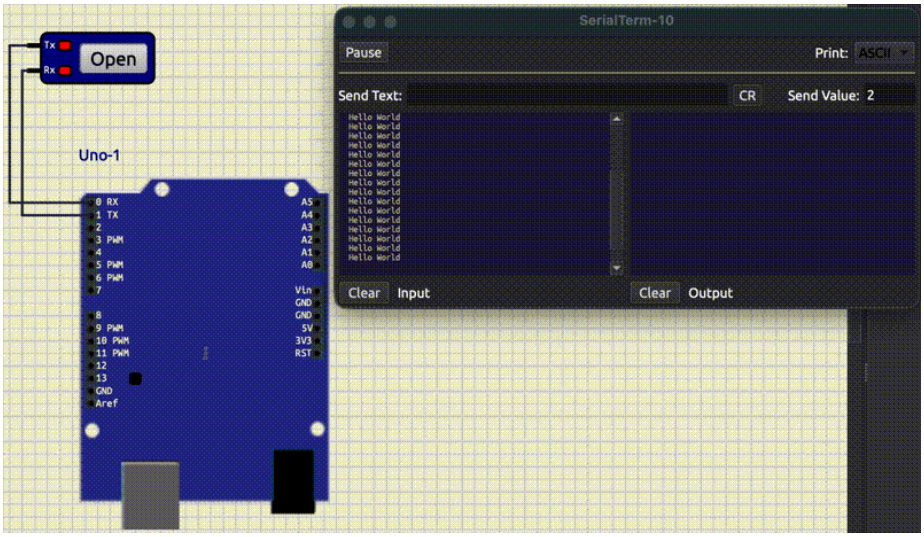


Figure 26: Hello World Serial

As you can see in the left panel **Hello World** is being printed constantly. Also, **Rx** on the terminal becomes **yellow** whenever it receives data.

Read from Serial Terminal

We have managed to send data to the **Serial Terminal**. Now, let's talk about how to read data from it. To do so we can use a function called `Serial.read()`. It would read the incoming **byte**. If there is no data, it would return `-1`. Also, we have another function called `Serial.available()`. This function returns the number of bytes which are available for reading. So if we want to write a code that reads data and prints each character in separate lines, we can write something like below in the loop function:

```
if (Serial.available())
{
    char ch = Serial.read();
    Serial.println(ch);
}
```

The output would be something like below:

In the example above, I wrote **hello** and **world** separately. After I wrote down each one of them I pressed **Enter**. As you can see in the right panel, we can

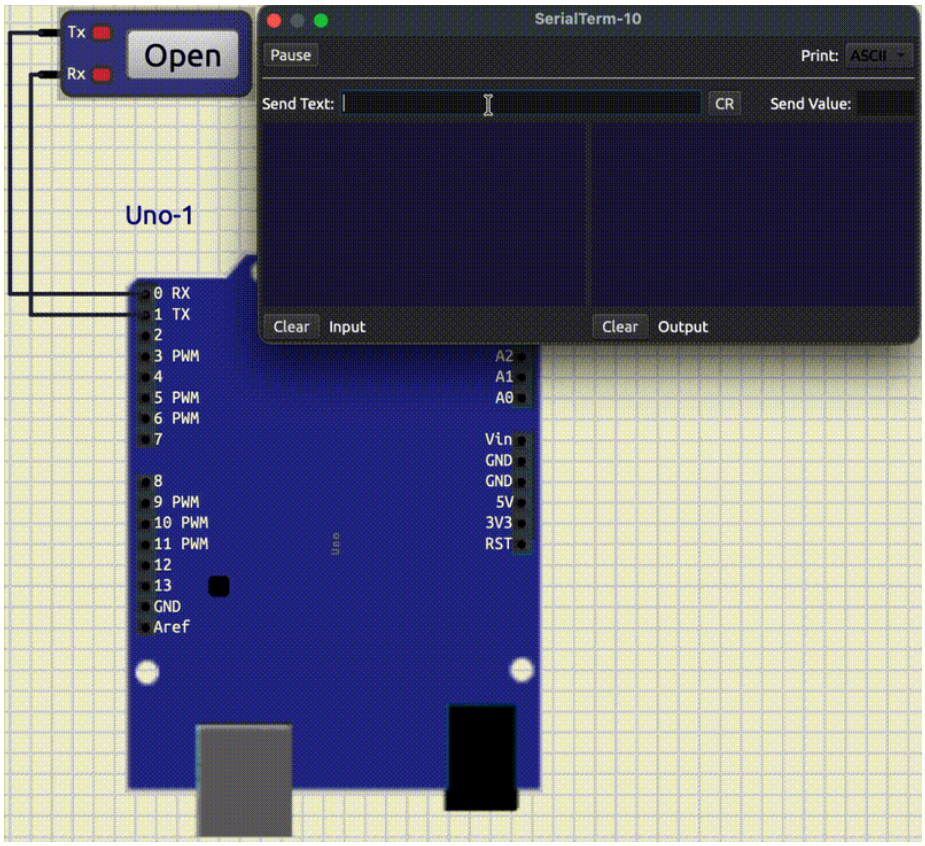


Figure 27: Serial Read

see the input that we sent to the **Arduino** and in the left panel we can see the response of the **Arduino**.

Read until new line

If we want to read the whole string, we can use a function called `Serial.readStringUntil`. This function reads the data in the buffer until it reaches the terminator that we give it as an argument. As a result it returns a **String**. So, if we want to change our code to read the whole line, we can change it like below:

```
if (Serial.available())
{
    String result = Serial.readStringUntil('\n');
    Serial.println(result);
}
```

The output would be something like this:

As you can see, we write a whole sentence and when we press enter after some time it would print us the result.

Built-in Serial Monitor in SimulIDE

For debugging purposes, **SimulIDE** has implemented a **Serial Monitor** that you can see the transmitted and received data through that. To access it, you can right-click on the **Arduino Uno** then select **mega328/Open Serial Monitor/USart**.

Software Serial

The built-in pins for **Serial Communication** in **Arduino Uno** are **pin 0, 1**. This is managed by hardware. If you want to use other pins for the **Serial communication**, you should use a package called `SoftwareSerial`. You can import it like below:

```
#include <SoftwareSerial.h>
```

To tell the `SoftwareSerial` which ports you need, you should make an object like below:

```
SoftwareSerial mySerial(10, 11); // RX, TX
```

In the code above, we have used **pin 10** for **RX** and **pin 11** for **TX**. Now, we are ready to connect the device that we want to communicate with to **pin 10 and 11** and start our **Serial communication** as we would before. The whole code for a **Hello World** example is like this:

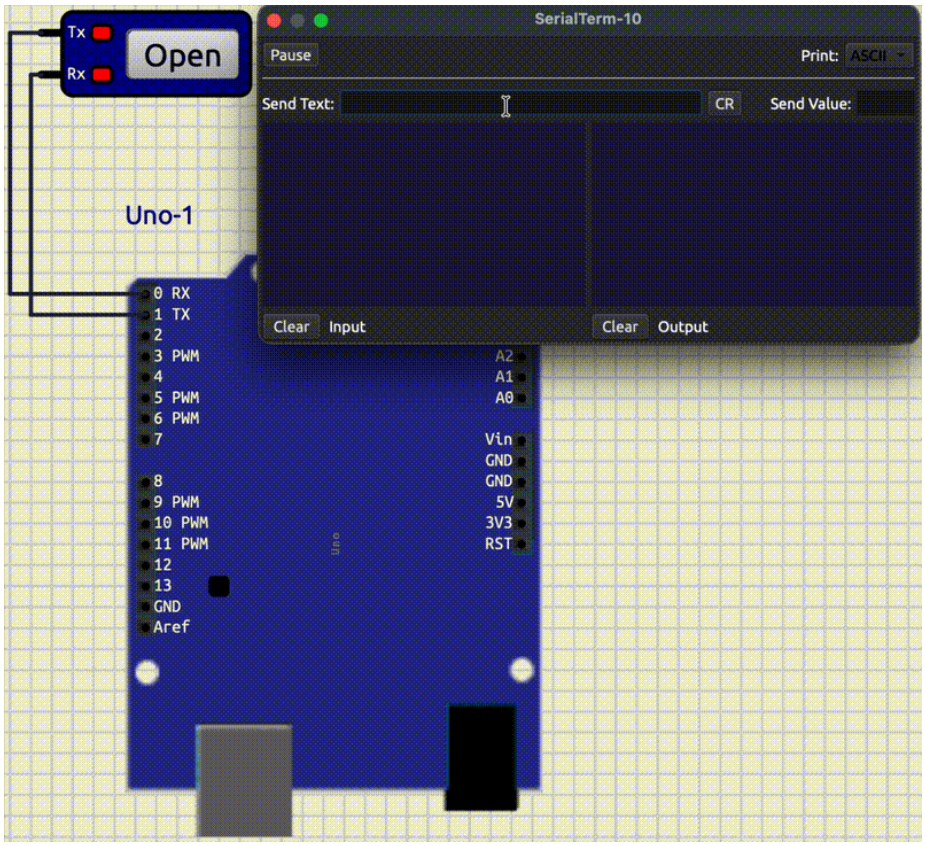


Figure 28: Serial Read String

```
#include <Arduino.h>
#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11); // RX, TX

void setup()
{
  mySerial.begin(9600);
}

void loop()
{
  mySerial.println("Hello SoftwareSerial!");
  delay(1000);
}
```

Your output should look like below:

Counter on two Arduinos

Now, that we know how two devices can communicate using **Serial communication**, let's connect two **Arduinos** together. We add a **Keypad** to the first **Arduino**. Its job is to receive a number and send it to the second Arduino. Then, we add an **LCD** to the second Arduino. Its job is to count down the number that it received to `0`. Your output should look like this:

Conclusion

In this tutorial we walked through the concepts of **Serial Communication**. First, we explained the **Serial Communication**. Then, we provided an example on how to write with it. After that, we learned how to read from it. Next, we explained about `SoftwareSerial` and how to make other pins to work as a **Serial communication** pins. Finally, we provided an example with two arduinos to make you understand the concepts better.

Analog

Introduction

In the previous tutorials, we were focusing on **digital** read and write. We could only write and read **0** and **1** from a pin. In this tutorial, we will discuss how to write and read values between **0** and **1**.

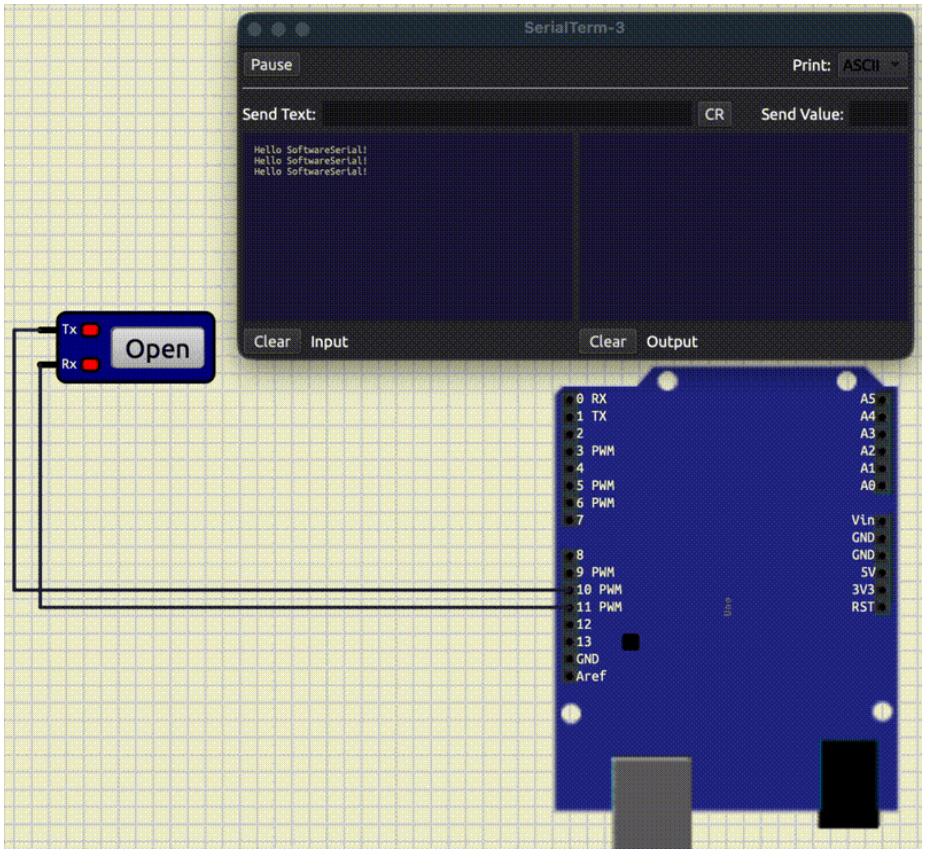


Figure 29: Software serial

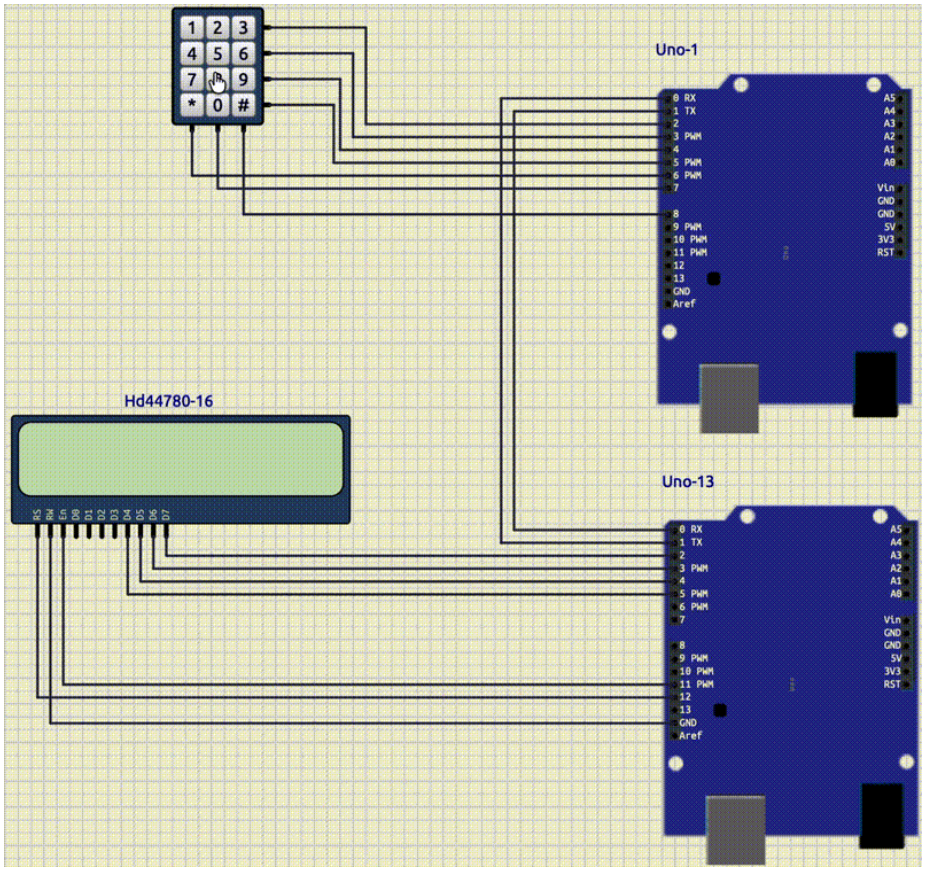


Figure 30: Two Arduinos

Analog Read

In the previous tutorials, we were working with **digital** input and output. As you recall, to read a **digital** input, we had a function called `digitalRead`. To read **analog** input, we have a function as well. This function is called `analogRead`. The syntax of it is pretty similar to the `digitalRead`, the only exception is that it returns a number in a range of $[0, 1023]$. As you might have guessed from the range, `Arduino Uno` allocates `10 bits` for reading analog data.

In `Arduino Uno`, we have 6 pins that we can use to read **Analog** input. These pins are labeled as **A0** to **A5**. In the image below, we show them by drawing a yello rectangle over them.

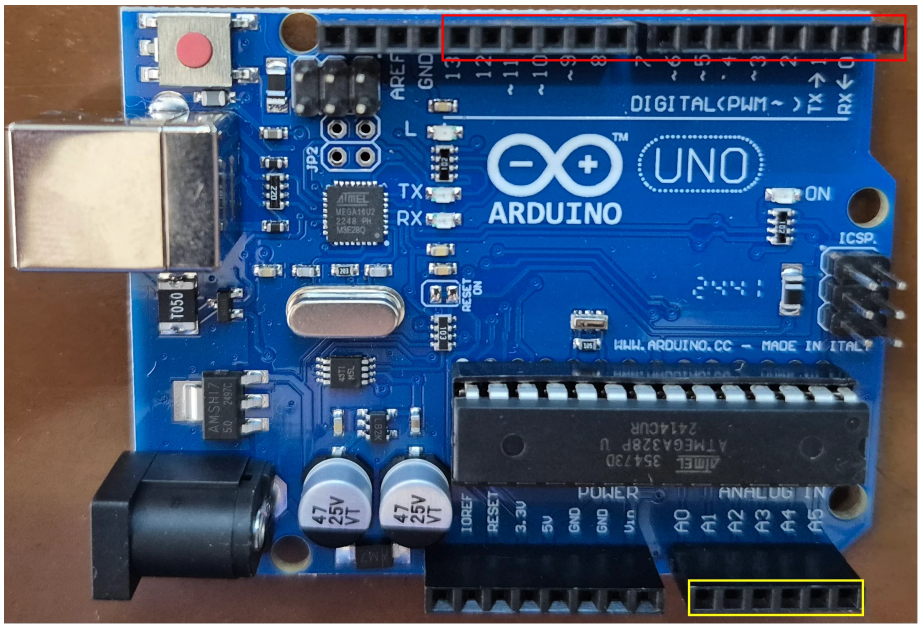


Figure 31: Analog pins

For reading digital values, $5V$ indicates 1 and $0V$ indicates 0. For analog values, $5V$ indicates 1023 and $0V$ indicates 0. As you can see, we divide the values between $5V$ and $0V$ into 1024 parts. With using this technique, we are being able to read voltage between $5V$ and $0V$.

Potentiometer

To create a voltage between $5V$ and $0V$, we can use a device called **Potentiometer**. You can find it at **Passive/Resistors/Potentiometer** in **SimulIDE**. As you can see, a **Potentiometer** has 3 pins and a button to control the output voltage. **Potentiometer** creates voltage with increasing and decreasing the re-

sistance. Let's connect a **Potentiometer** to a fixed voltage to see how it works. To do so, we can follow these steps:

- Put a **Potentiometer** on the board (**Passive/Resistors/Potentiometer**).
- Connect the pin that is closer to a red line, to a **Fixed Voltage**.
- Connect the other pin to the **Ground**.
- Put a **VoltMeter** on the board (**Meters/VoltMeter**).
- Connect the output pin of the **Potentiometer** (the pin with the arrow on it) to the **red pin** of the **VoltMeter**.
- Connect the other pin of the **VoltMeter** (the pin beside the red pin) to the ground.

Your connection, should look like as following:

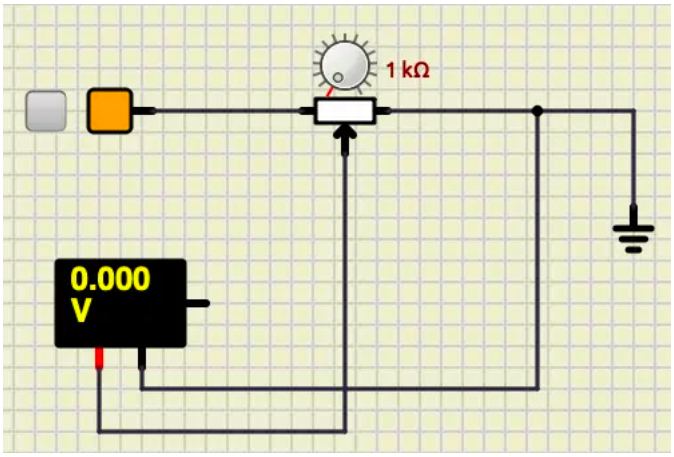


Figure 32: Potentiometer

Now, let's start the simulation and rotate the button on the **Potentiometer**.

As you can see, we can make voltages between $5V$ and $0V$.

Potentiometer and Arduino

Now, let's connect our **Potentiometer** to the Arduino. The steps are pretty much the same.

- Connect the pin closer to the red line to a **5V**.
- Connect the pin on the opposite of the red line to the **Ground**.
- Connect the output pin (pin with an arrow on it) to **A0**.

Your connection should look like this:

Now, let's write a code to read the analog data.

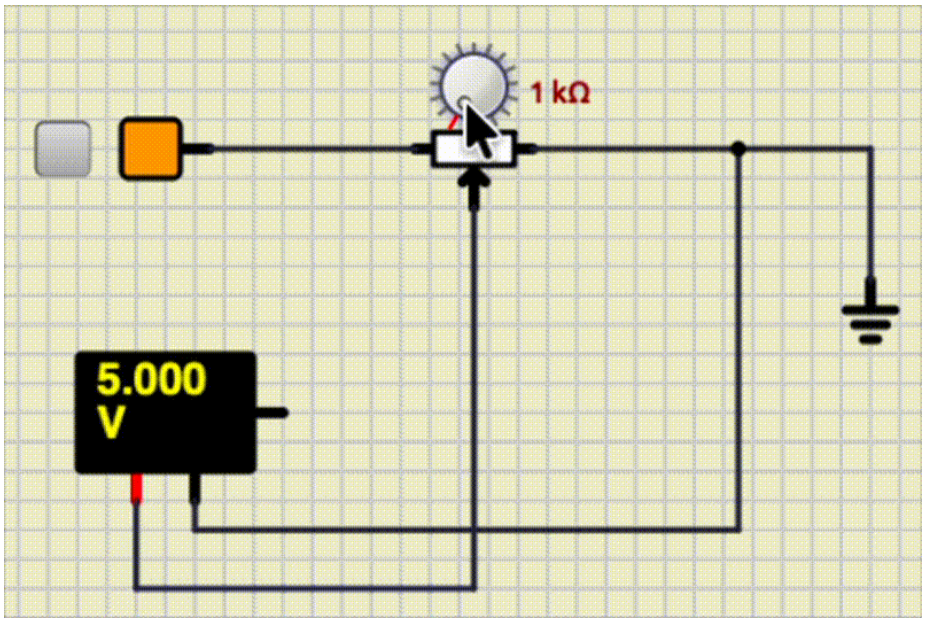


Figure 33: Potentiometer gif

```
#include <Arduino.h>

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

void loop()
{
  int our_input = analogRead(A0);
  Serial.println(String(our_input));
  delay(1000);
}
```

The code above, reads the analog input from `A0`. Then, it prints the read value into Serial terminal. The output looks like as following:

PWM

Before we get to work with **Analog Write**, let's learn about **PWM**. Because **Arduino Uno** uses **PWM** to write **Analog** data. **PWM** (Pulse Width Modulation), is a technique for controlling the power delivered to a component.
In this technique we use different width of pulses in a signal. These signals

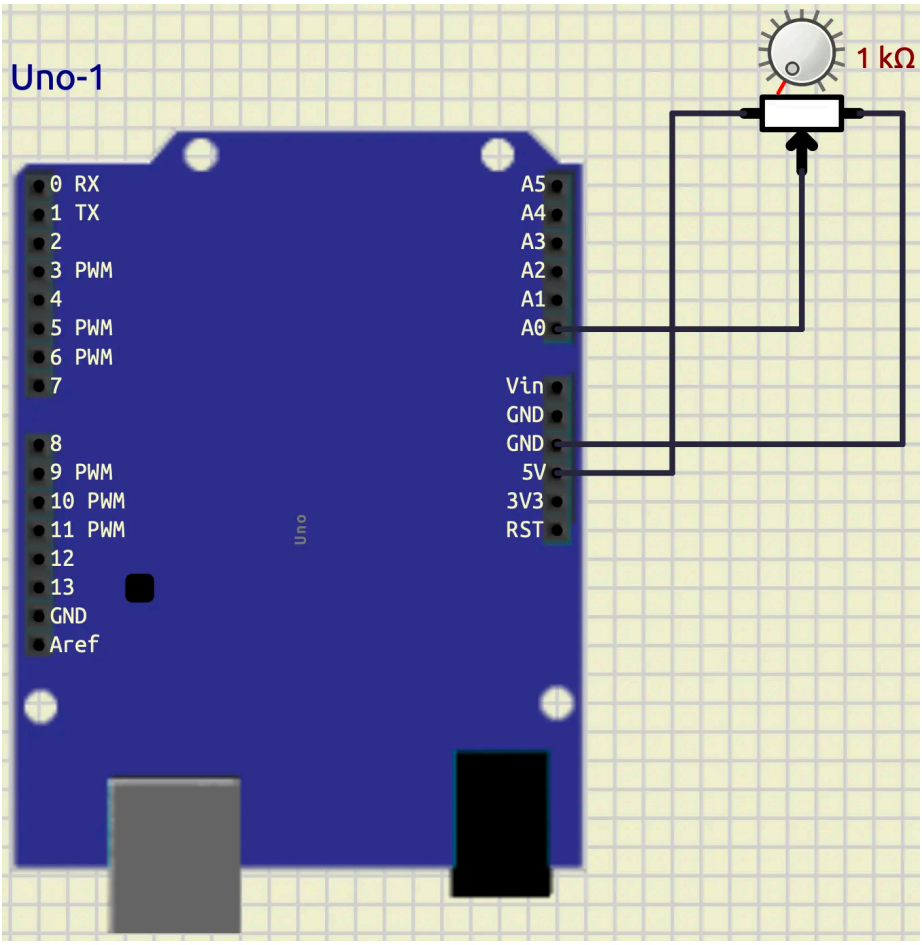


Figure 34: Analog Read

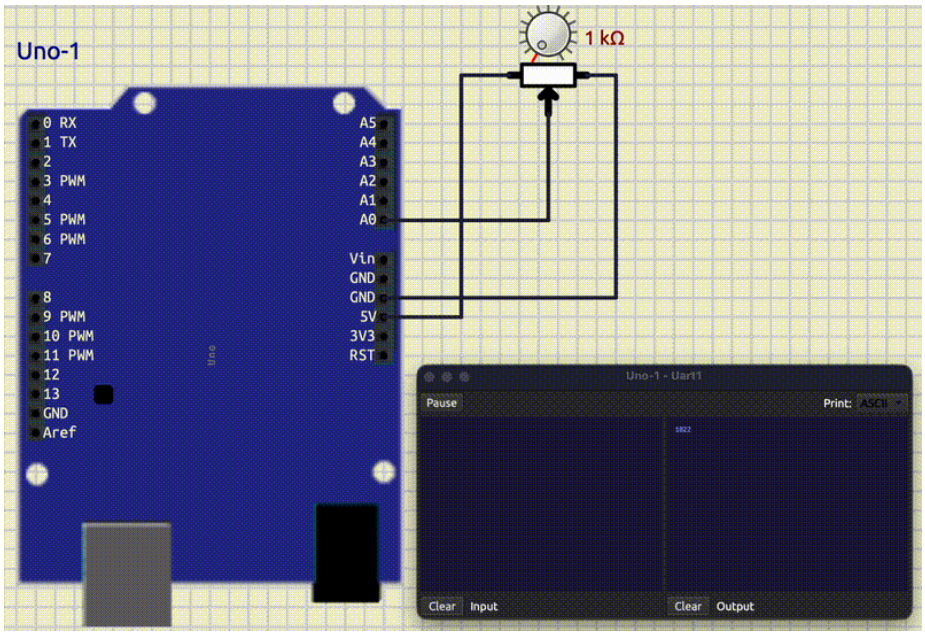


Figure 35: Analog Read gif

switch between 0 and 1. The percentage of the time that a pulse is 1 is called **duty cycle**. The pictures below show two examples of 100Hz **PWM**, one with the 30% duty cycle and the other 60%.

Analog Write

Now, that we know about **PWM**, let's talk about writing analog data in **Arduino uno**. In **Arduino Uno** we have 6 pins that we can create **PWM** on them. Including: 3, 5, 6, 9, 10, and 11 These pins are shown in the board with a ~ beside them and in **SimulIDE** with **PWM**. To create our **duty cycles**, we can use the numbers in range $[0, 255]$. 255 means 100 and 0 means 0 duty cycles.

For writing **digital** values, we had a function called `digitalWrite`. We have a similar function for **analog** values as well, and it is called: `analogWrite`. Their syntax is similar, with the exception that `analogWrite` accepts an integer for its second argument.

Now, let's connect an **LED** to pin 3. Your connection should look like below:

Now, let's write a code to write analog data on pin 3.

```
#include <Arduino.h>
```

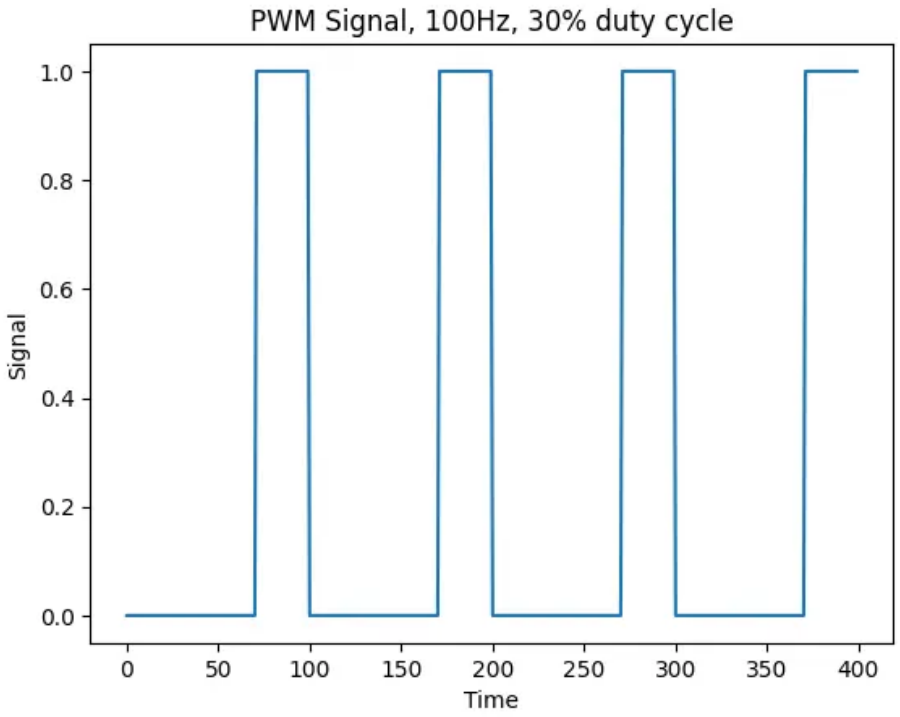


Figure 36: PWM 100hz 30d

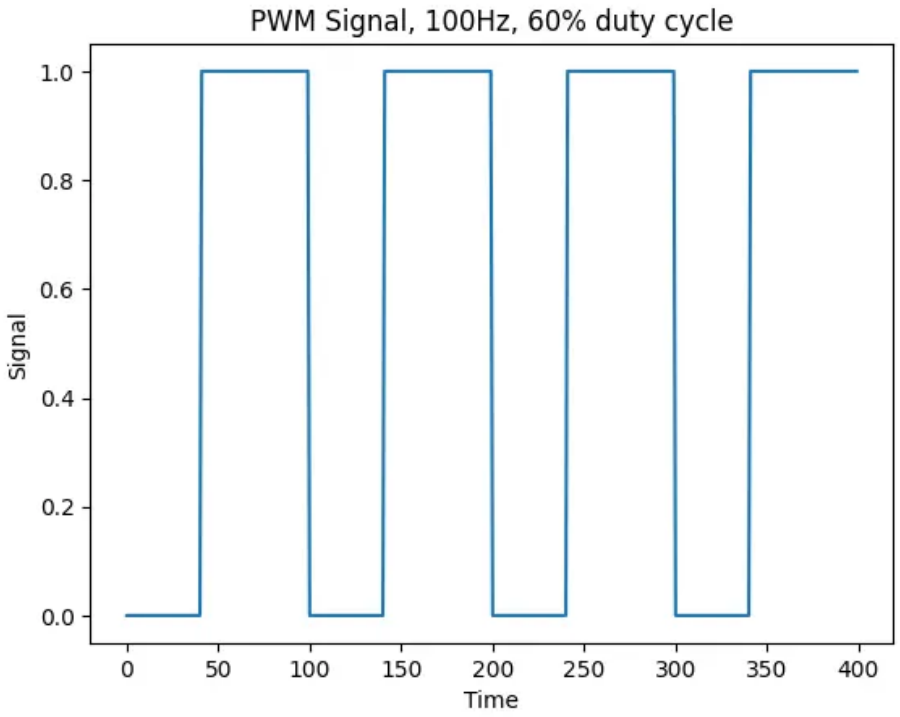


Figure 37: PWM 100hz 60d

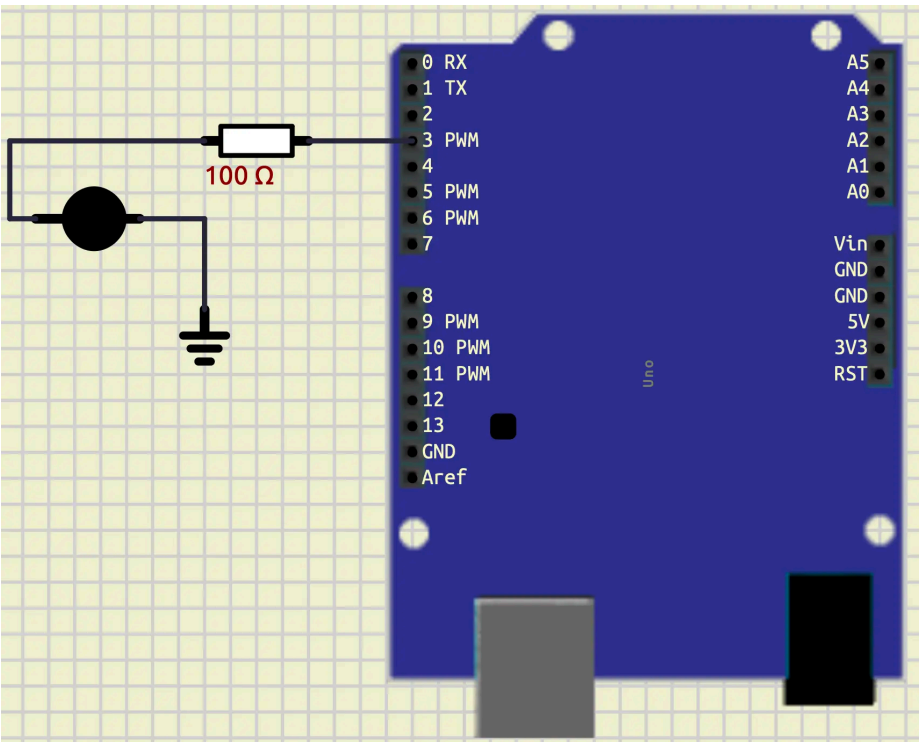


Figure 38: Analog Write

```
void setup()
{
  pinMode(3, OUTPUT);
}

void loop()
{
  for (int i = 0; i < 256; i++)
  {
    analogWrite(3, i);
    delay(10);
  }
}
```

In the code above, first we set the pin mode of pin 3 as output. Then, we have a for loop that starts from 0 and goes until 256. So, we expect the brightness of our **LED** changes from dark to light. As you can see in the output below, this is the exact thing that is happening.

If we want to see the **PWM** that we are generating, we can connect an **Oscope** to our output. To do so, we can follow these steps:

- Put an **Oscope** on the board (**Meters/Oscope**)
- Connect the bottom pin to the ground
- Connect any other pin to the output of pin 3.

Your output should look like as following:

Combining both

Now, let's combine the both reading and writing analog values together. Your connection should look like below:

Now, let's write a code that changes the brightness of the **LED** using the **Potentiometer**.

```
#include <Arduino.h>

void setup()
{
  pinMode(3, OUTPUT);
}

void loop()
{
  int our_input = analogRead(A0);
  int brightness = map(our_input, 0, 1023, 0, 255);
  analogWrite(3, brightness);
```

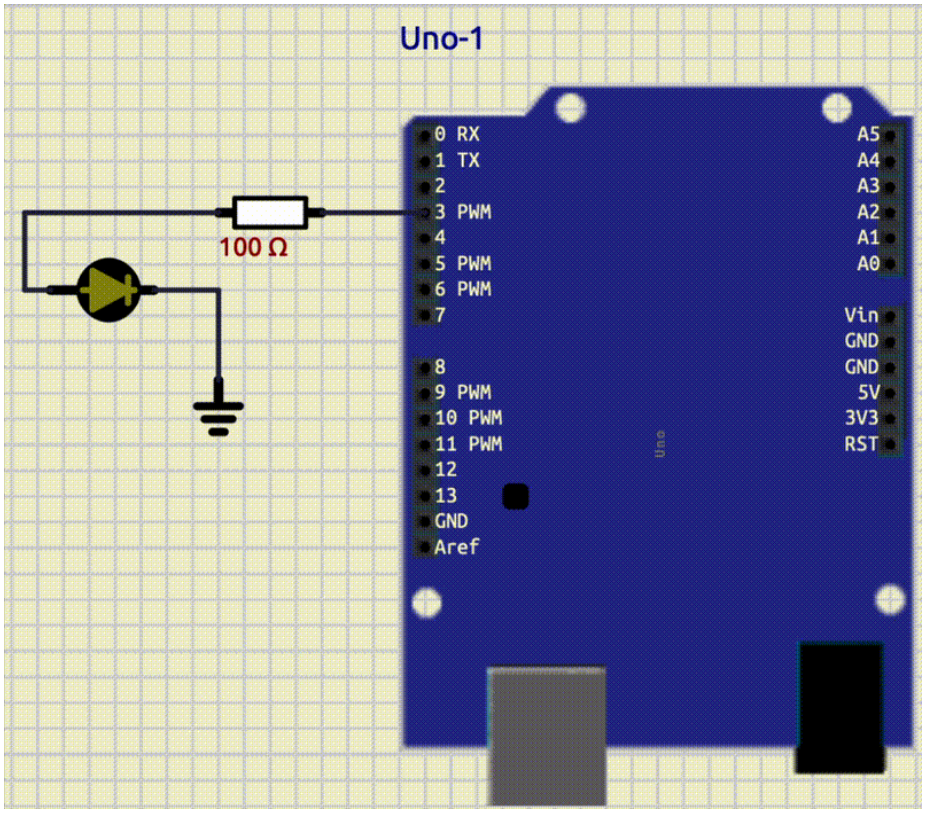


Figure 39: Analog Write gif

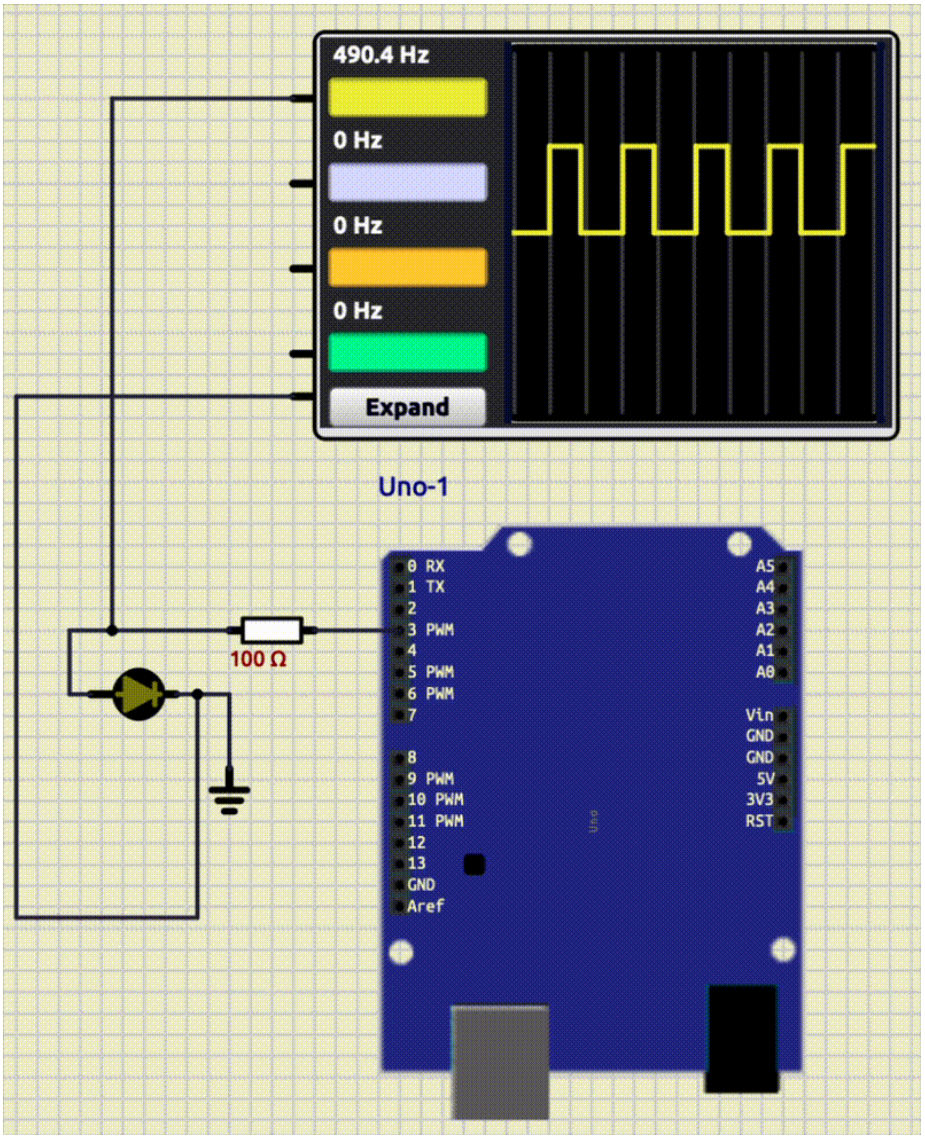


Figure 40: Analog Write Oscope gif

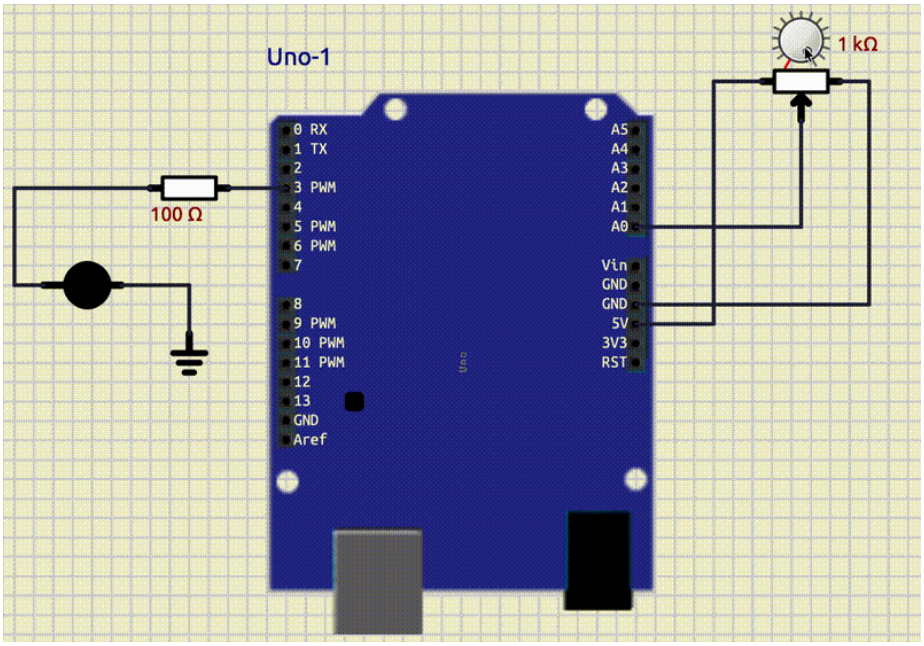


Figure 41: Combining Both

```
  delay(1000);
}
```

As you can see, in the code above, we read analog data from pin `A0`. The value is in the range of $[0, 1023]$. To write it on the pin 3, we need to rescale the value to the range of $[0, 255]$. To do so, we have used a function called `map`. This function takes our input as its first argument, then takes the current range and the target range and outputs the rescaled value. After that, we are able to write that rescaled value to pin 3. The output looks like as following:

If we want to see the **PWM**, we can connect an **Oscope** like below:

DC motor

DC motor is a device that converts electrical energy into rotation. We use a **DC motor** in so many different things like, saw, drill, toys and e.t.c. To connect a **DC Motor** to an Arduino we should consider that, a **DC Motor** takes too much current, and we should not connect it directly to an **Arduino pin**. In **SimulIDE** it is not going to be a problem, but in real life we should not do that. To practice real life connection, we use a battery to power our **DC motor**. Also, we use a **Mosfet** to write analog data on our **DC motor**. Now, let's change our **LED** to a **DC motor** with these following steps:

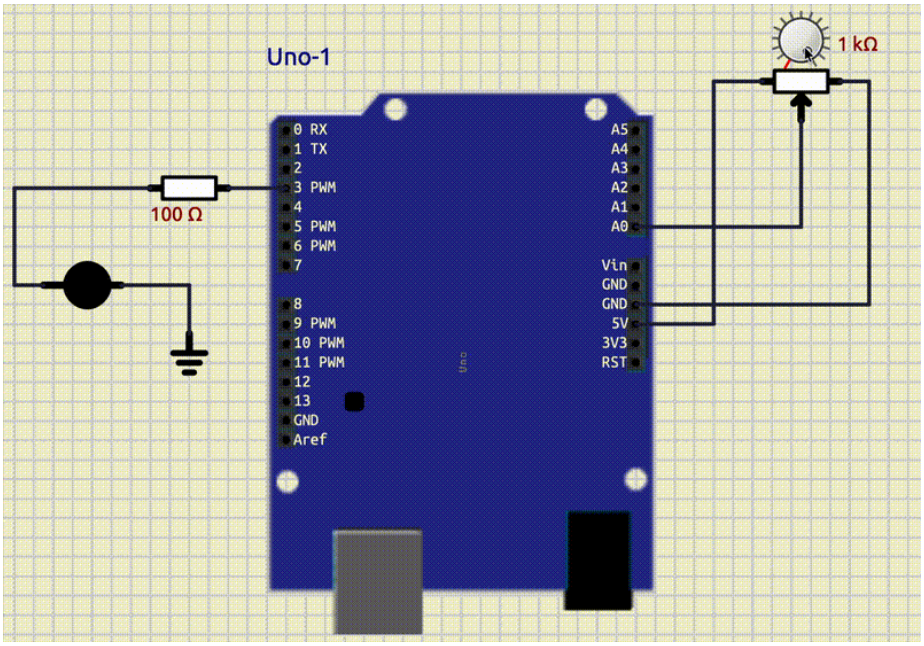


Figure 42: Combining Both gif

- Remove the **LED**
- Put a **DC motor** on the board (**Outputs/Motors/DC Motor**).
- Put a **Battery** on the board (**Sources/Battery**).
- Put a **Mosfet** on the board (**Active/Transistor/Mosfet**).
- Connect pin 3 to the middle pin of the mosfet (pin with the arrow).
- Connect one pin of the mosfet to the ground.
- Connect the other pin of the mosfet to the negative (-) of the **DC Motor**.
- Connect the positive (+) of the motor to the positive (red) of the battery.
- Connect the negative (black) of the battery to the ground.

Your connection should look like below:

There is no need to change our code. So, if we run our simulation with the code that we already have written for the **LED**, the output would look like below:

Servo Motor

Servo motor is a type of electric motor that is designed for precise control. It is widely used in robotics. We can control the position of the servo motor by sending a **PWM** signal to it. In this session, we will be using a simple DC servo motor which only take angles in range of [-90, 90]. The desired `PWM` frequency for this servo motor is `50Hz`. This servo motor has three pins: VCC, GND, and signal. So, let's connect a **Servo motor** to an **Arduino Uno**.

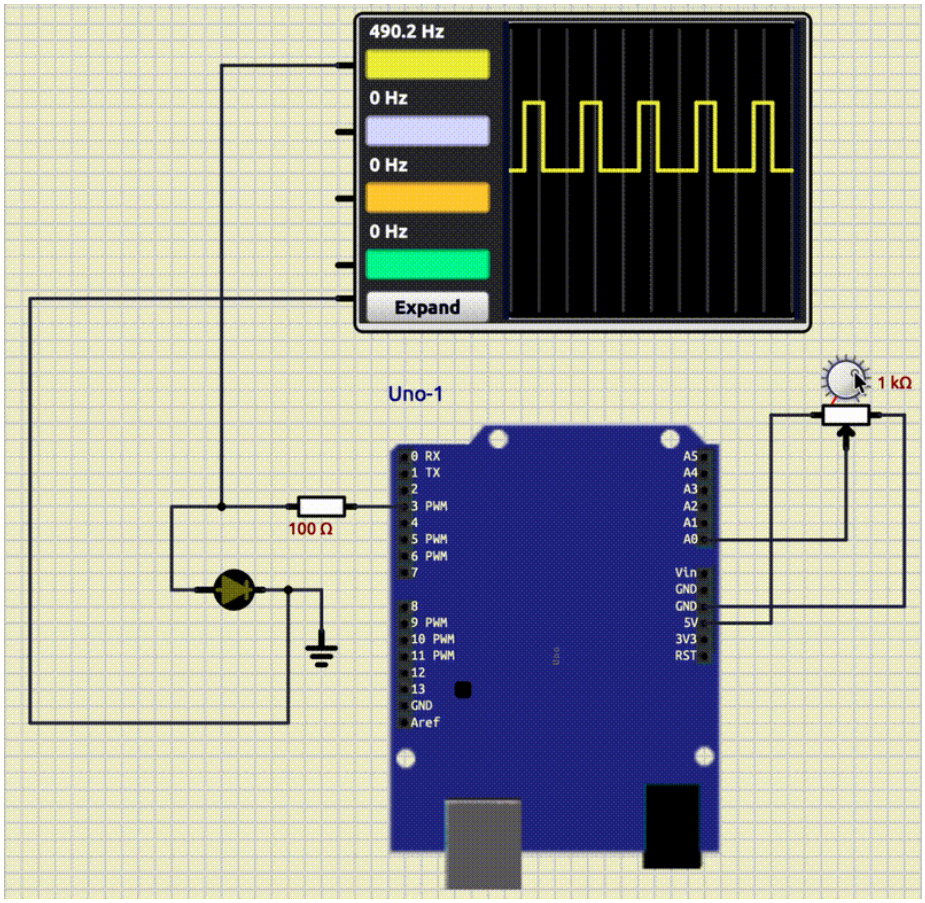


Figure 43: Combining Both Oscope gif

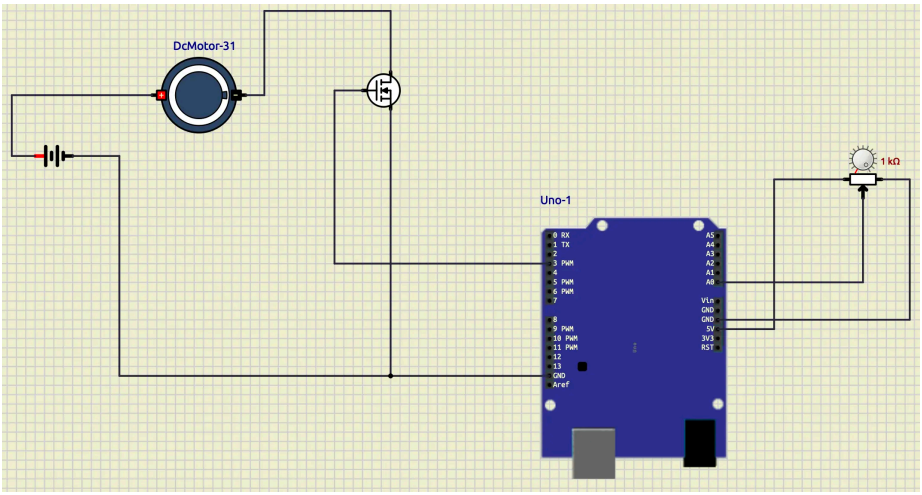


Figure 44: DC Motor

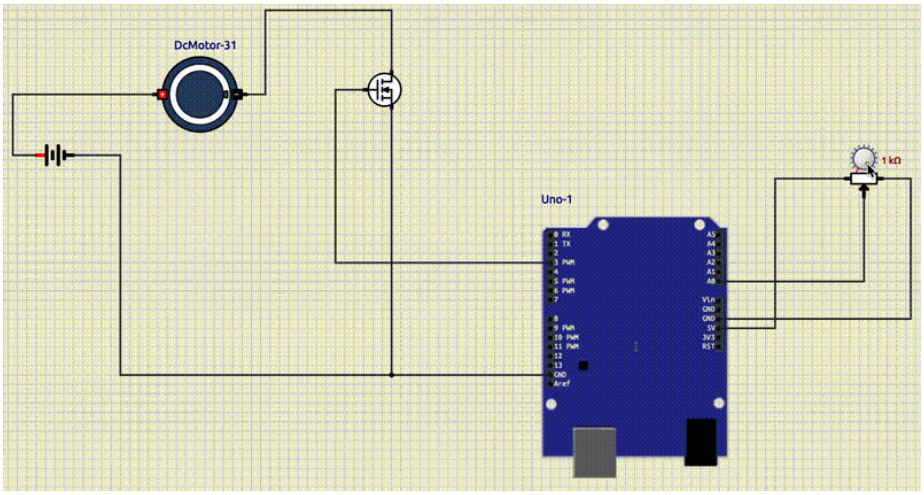


Figure 45: DC Motor gif

- Put a **Servo motor** on the board (**Outputs/Motors/Servo Motor**).

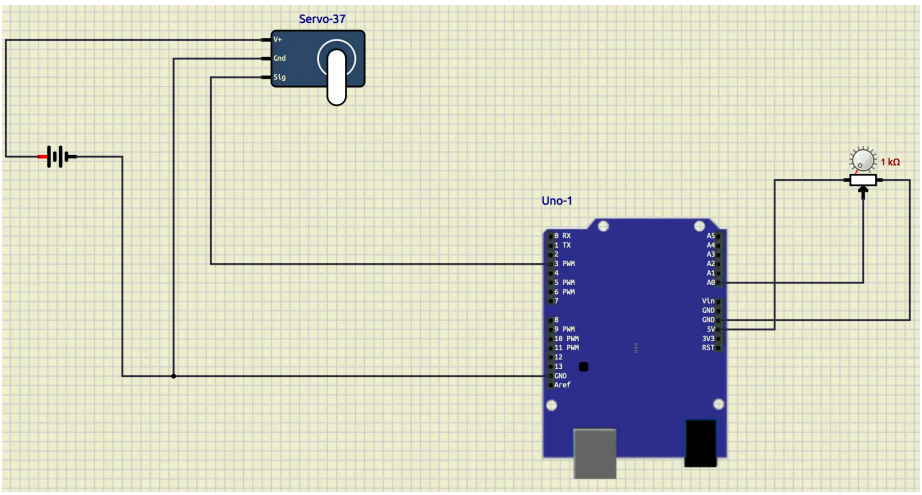


Figure 46: Servo motor

So, to control a **Servo motor** we need to create a **PWM** signal with the frequency **50hz**. To do so, we can use a library called **Servo**. We can add it to our **PlatformIO project like this:

```
lib_deps =
    ...
    arduino-libraries/Servo
```

Then, we can include it like below:

```
#include <Servo.h>
```

Now, we should create an object of `Servo` like below:

```
Servo my_servo;
```

Then, we need to do the initialization by using a function called `attach`. For example, let's attach our servo to pin 3.

```
my_servo.attach(3);
```

Then we can use the `write` function to write the angle that we want. For example let's write 45 on it.

```
my_servo.write(45);
```

Now, write a code that maps the data of the **Potentiometer** to the servo angles. Your output should look like as following:

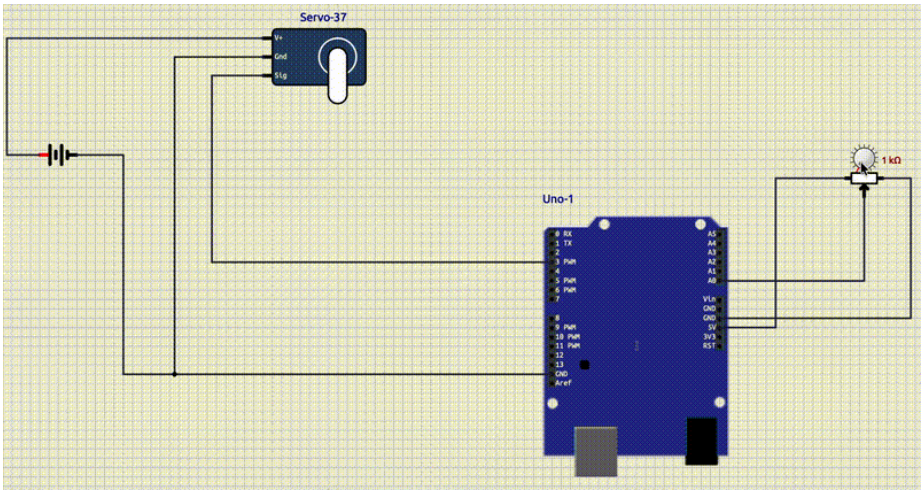


Figure 47: Servo motor gif

If we want to see the generated **PWM**, we can add a **Oscope** like below:

Combine All

Now, let's combine all the things that I have learned. We need two **Potentiometers**, one for controlling the **Servo motor** and the other one for controlling the speed of **DC motor** and the brightness of the **LED**. Your output should be like below:

Conclusion

In this tutorial, we have Learned about how to deal with **Analog** data in **Arduino Uno**. First, we learned about `analogRead`. Then, we introduced **PWM** and that **Arduino** uses **PWM** to write analog data using `analogWrite`. After that, we have explained about two of the use cases of the `analogWrite`, which were **DC motor** and **Servo motor**. Finally, we combined all the things that we have learned so far.

Interrupt

Introduction

In the previous session, we learned how to work with analog values in **Arduino Uno**. In this session we will learn about one of the most important aspects of microcontrollers, which is called **Interrupt**. We have different kinds of **Interrupts**, but in this tutorial, we are going to focus on the **External Interrupt**, which we will explain very soon.

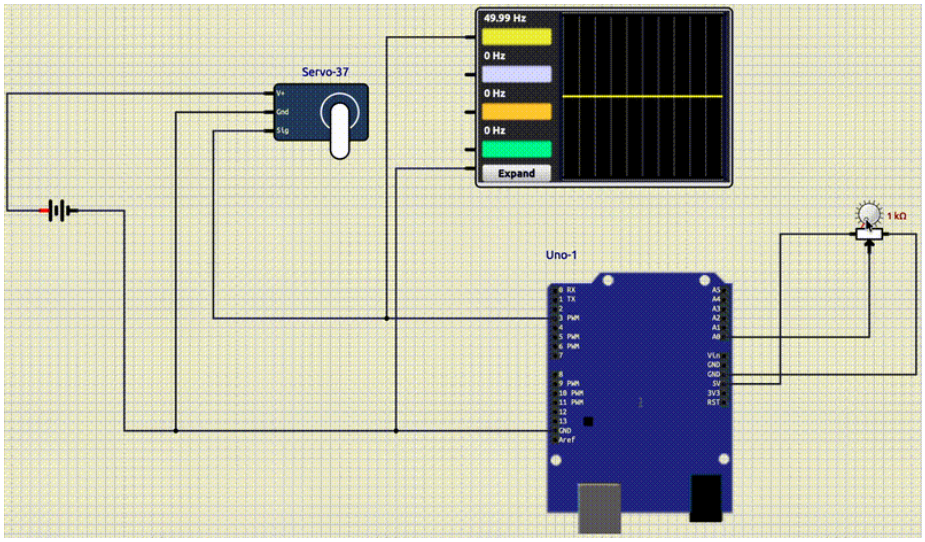


Figure 48: Servo motor oscope gif

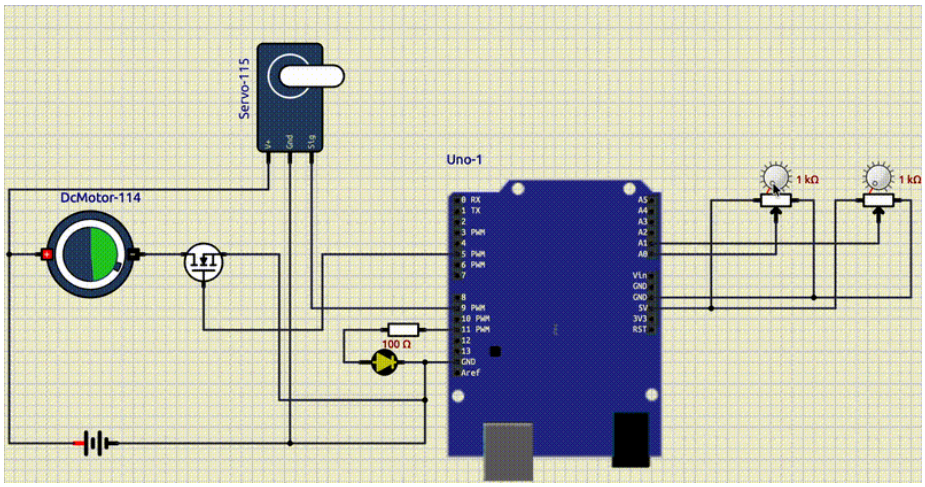


Figure 49: Combine all gif

What is an External Interrupt?

Interrupt is a special signal. It tells the microcontroller to stop (halt) what he is doing right now and execute the given code. This code, should be in a function called **Interrupt Service Routine Function** (**ISR Function**). In **Arduino Uno**, we have 2 external **interrupts**. These **interrupts** are connected to **pin 2** and **pin 3**. So, if we want to work with these interrupts, we should connect a button to one of these two pins.

Setup LEDs

To understand the concept of interrupts better, let's make a routine. The routine that we are going to make, includes 8 LEDs, which are connected like the image below:

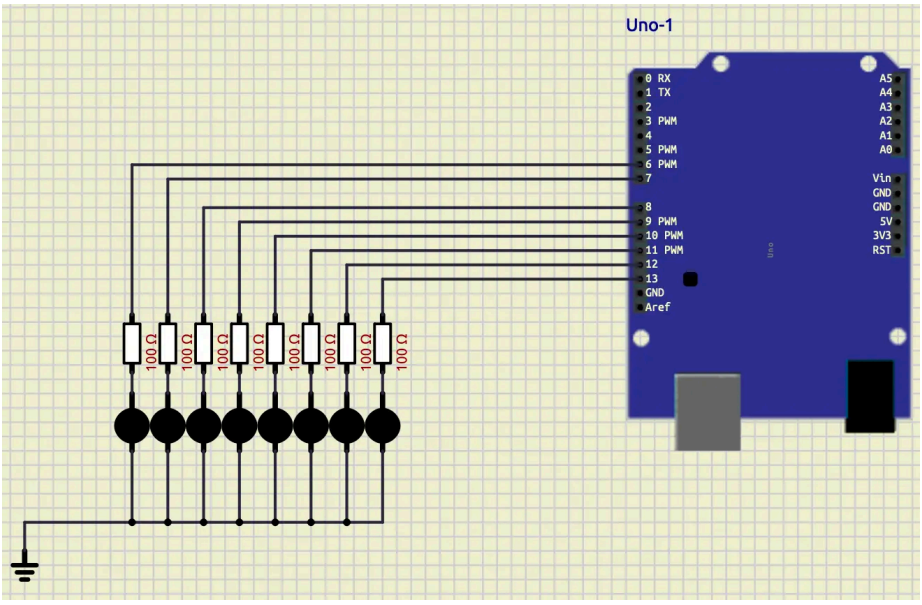


Figure 50: led setup

You are free to connect the LEDs to any pin that you want, except pins 0 to 3. We connected our LEDs to pin 6 to 13. Now, let's write a code for these LEDs to turn on in a sequence.

```
#include <Arduino.h>

int led_pins[8] = {6, 7, 8, 9, 10, 11, 12, 13};

int current_led = 0;
```

```
void setup()
{
  for (int i = 0; i < 8; i++)
  {
    pinMode(led_pins[i], OUTPUT);
  }
}

void loop()
{
  digitalWrite(led_pins[current_led], HIGH);
  delay(200);
  digitalWrite(led_pins[current_led], LOW);

  current_led++;
  current_led %= 8;
}
```

In the code above, first we have defined which pins we have for our LEDs in a variable called `led_pins`. Then, we have defined a variable called `current_led`. `current_led` indicates which led should be on at the moment. In the `setup` we have defined all of our `led_pins` to `OUTPUT`, because we want to write values in them. Then, in the `loop` at first we have turned on the first LED and keep it on for `200ms`. After that, we turned that LED off. For the next step, we increment the value of `current_led` and make sure that it doesn't go more than 8. Our output looks like the following:

Now, we are ready to add an interrupt and see its effect.

Interrupt to pause

Right now, our LEDs are getting turned on in a sequence. Our goal is to pause this routine, using an interrupt. To do so, at first let's connect a button to the **pin 2** like below:

As you can see, in the image above, we set the default position of our button to **Closed**. To do so, we can go to the properties of the button and set it to **Normally Closed**. Because, we want to have **5V** by default to the pin and when we press the button, the value of the pin becomes **0V**.

Now, let's learn how we can define an **interrupt** in **Arduino**. To do so, we have to use a function called `attachInterrupt`. This function, takes three arguments:

1. The **pin** that we want the interrupt to work on
2. The **ISR** function
3. The mode

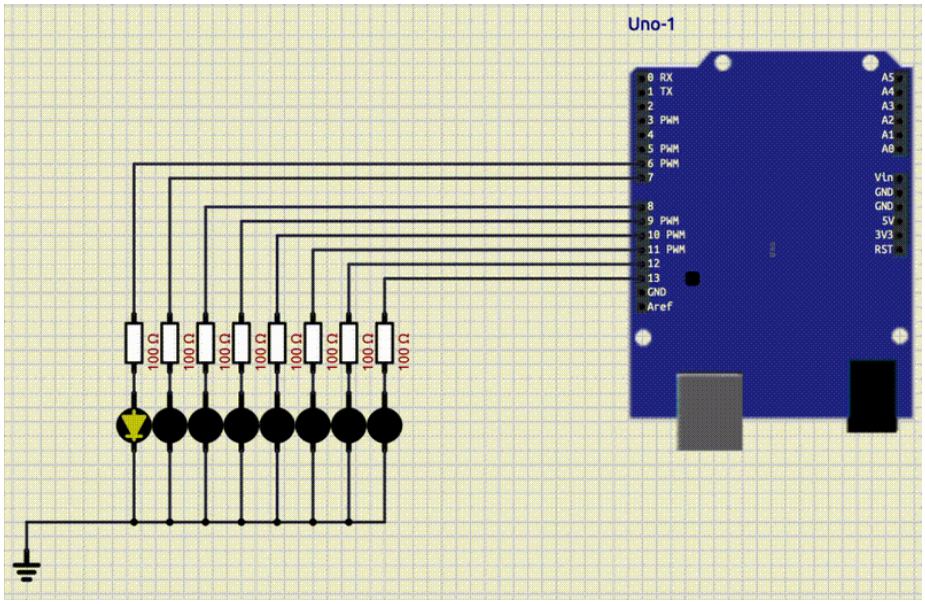


Figure 51: led setup gif

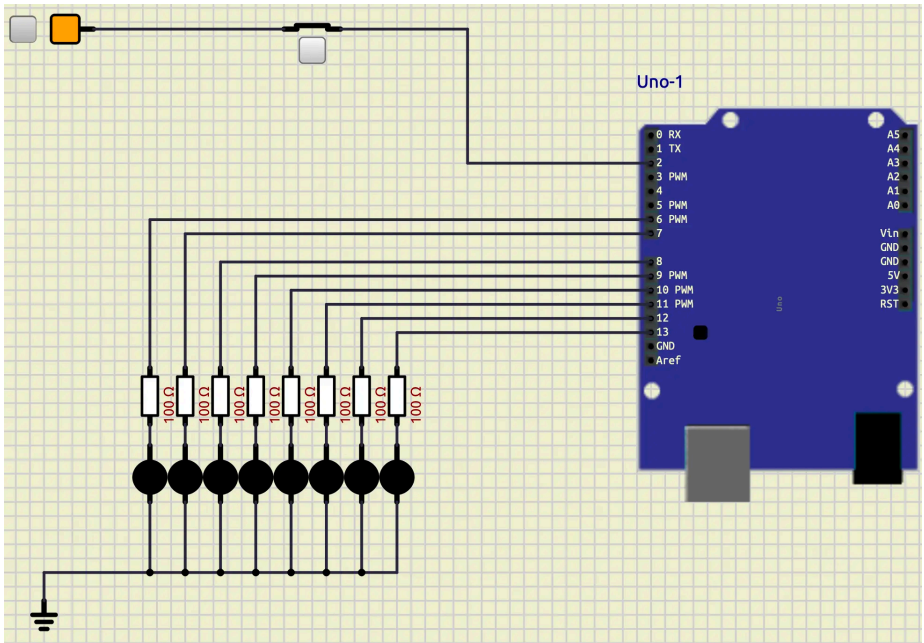


Figure 52: interrupt pause

First, let's start with passing the pin as its first argument. To do so, first we should change the pin to an **Interrupt** pin using a function called **digitalPinToInterrupt**. For example, if we want our **pin 2** to function as an interrupt, we should right something like below (We are going to fill the `...` after):

```
attachInterrupt(digitalPinToInterrupt(2), ... ,...);
```

For the second argument, let's define an empty function, that we are going to fill it later, and pass that function to it. For example, a function like below:

```
void isr_pause()
{
}
```

So, now we have:

```
attachInterrupt(digitalPinToInterrupt(2), isr_pause ,...);
```

The only remaining thing is the mode. In **Arduino Uno** we have 4 modes for the interrupts.

Mode	Description	figure
LOW	trigger the interrupt whenever the pin is low.	
CHANGE	trigger the interrupt whenever the pin changes value.	
RISING	trigger when the pin goes from low to high.	
FALLING	trigger when the pin goes from high to low.	

Now, let's put the mode to **RISING**. It means that we are going to have an **interrupt** when leave the button. So, here is the full function call, which we put it in the `setup`.

```
attachInterrupt(digitalPinToInterrupt(2), isr_pause, RISING);
```

Now, it's time to implement the logic. If an interrupt happens, we want the routine to be paused. So, we are going to define an integer variable called `x`

with default value of `1`. Instead of incrementing `current_led` by 1, we are going to increment it by `x`. It would look like this: `current_led += x`. In your `isr_pause` function, we are going to toggle `x`. So, anytime an interrupt comes, the pausing and resuming happens. The full code, would look like below:

```
#include <Arduino.h>

int led_pins[8] = {6, 7, 8, 9, 10, 11, 12, 13};

int current_led = 0;
int x = 1;

void isr_pause()
{
  x = 1 - x;
}

void setup()
{
  for (int i = 0; i < 8; i++)
  {
    pinMode(led_pins[i], OUTPUT);
  }

  attachInterrupt(digitalPinToInterrupt(2), isr_pause, RISING);
}

void loop()
{
  digitalWrite(led_pins[current_led], HIGH);
  delay(200);
  digitalWrite(led_pins[current_led], LOW);

  current_led += x;
  current_led %= 8;
}
```

And this is going to be our output:

> Take note that when we press the button, the button goes up and value of the pin becomes 0

Volatile in Cpp

When we define a variable in **Cpp**, its compiler tends to optimize the usage of it. Sometimes, this optimization would work against us, specially in **Interrupt Service Routines**. Because **ISR** is a hardware event and **Cpp** might not plan

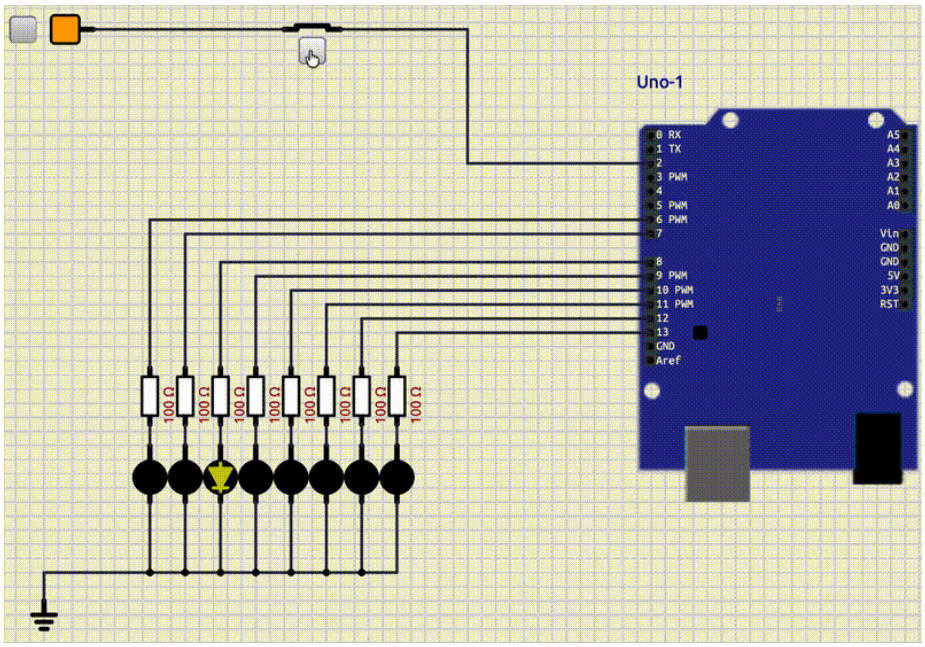


Figure 53: led pause gif

for it. To prevent this optimization from happening we can use a keyword called **volatile**. We should put this keyword before the declaration of our variable. For example, `volatile int v;`. This keyword tells **Cpp** not to apply optimization on this variable.

Reset with Interrupt

Now, add another interrupt to **pin 3**. This interrupt would reset the routine from the start. Take note that you might need to use **volatile**.

Increment pattern

Now, make LEDs with the pattern below. Make sure that the both interrupts work as intended (one for pausing and one for resetting).

LEDs dance

Let's make another routine like below. Again, make sure that the both interrupts work as intended.

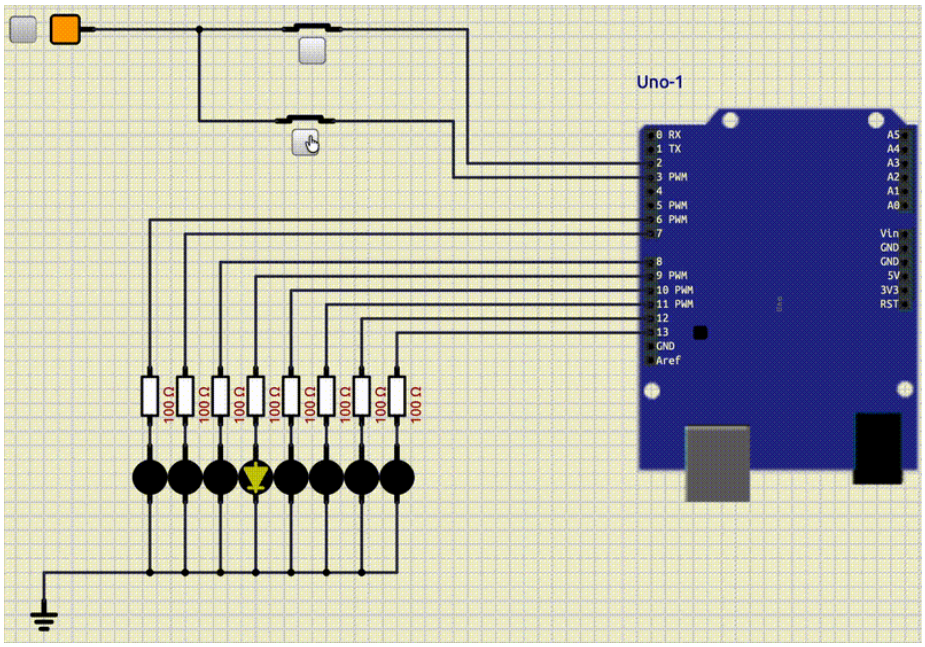


Figure 54: Reset with Interrupt

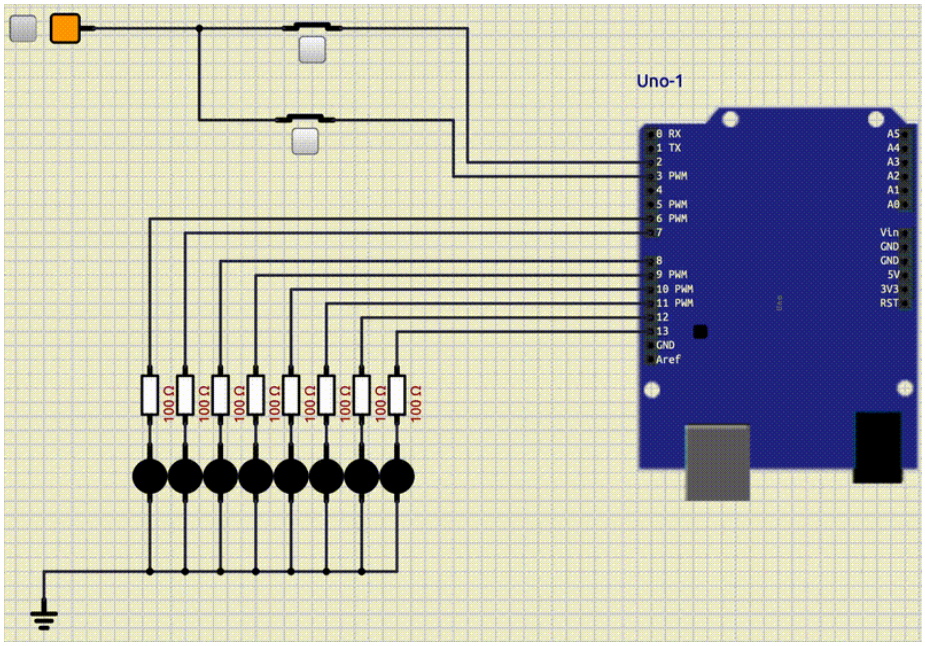


Figure 55: Increment Pattern

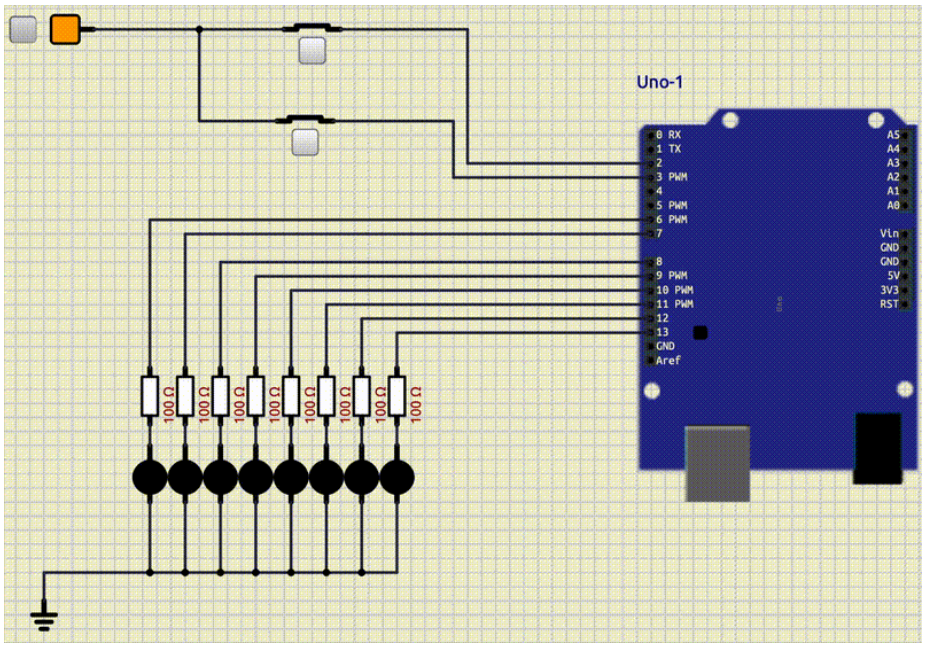


Figure 56: LEDs dance

Conclusion

In this tutorial, we have learned how to work with **External Interrupts** First, we explained what **External Interrupt** really is. Then, we set up a routine to understand the effect of an **Interrupt**. After that, we learned how to configure an **Interrupt** in **Arduino Uno** and saw how an **Interrupt** works. We explained about **volatile** in **Cpp** and why it's necessary to declare some variables as **volatile**. Then, We added another **Interrupt**. Finally, we have made two more routines to understand the concepts better.

I2C: part 1

Introduction

In the previous Tutorial, we learned about **Interrupt**. In this tutorial we will learn about **I2C** communication.

I2C Communication

Inter-Integrated Circuit (**I2C**), is a two-wire communication protocol. This protocol is designed for **short-distance communication** between microcontrollers and peripherals. It uses two pins to set up the communication:

- **SDA**: Serial Data
- **SCL**: Serial Clock

This way of communication, allows us to connect more than 1 component to the same 2 pins. For **I2C** every component has its own address. These addresses are mostly `7-bit`.

I2C is a master and slave protocol. It means that one device (Our Arduino) is a **master**, and the other devices are **slaves**. A **master** device controls the **clock** created in **SCL**. Also, **master** decides that if it wants to communicate with a **slave** or not. Each message of **master** is like below:

Field	Bit Count	Description
START	—	SDA goes LOW while SCL is HIGH → begins communication
Slave Address	7 bits	Unique address of target device (0–127)
R/W Bit	1 bit	0 = Write, 1 = Read
ACK/NACK	1 bit	Receiver pulls SDA LOW to acknowledge (ACK), HIGH for no-ack (NACK)
Data Byte 1	8 bits	First byte of data to write or read
ACK/NACK	1 bit	Receiver acknowledges the byte
Data Byte 2…N	8 bits	Additional data bytes (optional, depends on protocol)
ACK/NACK	1 bit	Acknowledge after each data byte
STOP	—	SDA goes HIGH while SCL is HIGH → ends communication

In the table above, you can see the message structure in **I2C**. First, **master** puts the **SDA** to low. This indicates that all **slaves** should listen. After that, it tells which **slave address** it wants to talk to. Then, with `1 bit` tells the **slave** if it wants to read or write. Next, there would be an acknowledgement bit. If **slave** was available, it would set the acknowledgement bit to `0`. (The default value of SDA is always `1`). After that, there would be a byte of data. Respect to the mode (read or write), **master** can send or receive that byte. Then, whoever receives the data, should set the acknowledgement bit to `0`. These byte transfer and acknowledgement can be repeated multiple times, until a stop signal. Stop signal can be created when we put the SDA to `1`.

To have a **I2C** communication in **Arduino** uno, we should use these pins:

signal	pin
SDA	`A4`

signal	pin
SCL	A5

Wire

To control the **I2C** communication, **Arduino** has a library called `Wire`. We can include `Wire` in our code like below:

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

To set up the **I2C** communication, we can use `.begin()` function, like below:

```
Wire.begin();
```

After doing that, we can start a communication with a **slave** in two ways:

- **write**
- **read**

To start a communication with a **slave** in order to **write**, we can use the code below:

```
Wire.beginTransmission(addr);   // start the communication in
 ↪  order to write with the slave with the address of `addr`
Wire.write(data);               // write data
Wire.endTransmission();         // finish transmission
```

If we want to our communication to be a **read** communication, we can

```
Wire.requestFrom(addr, number); // start a read communication
 ↪  with the slave with the address of `addr` and read `number`
 ↪  bytes
Wire.read();                    // read bytes
```

Let's connect an **I2C** component to the **Arduino** and check these functions.

Finding I2C address

- Connect the clock
- write the code
- Explain the code
- end transmission = 0

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

void setup()
{
```

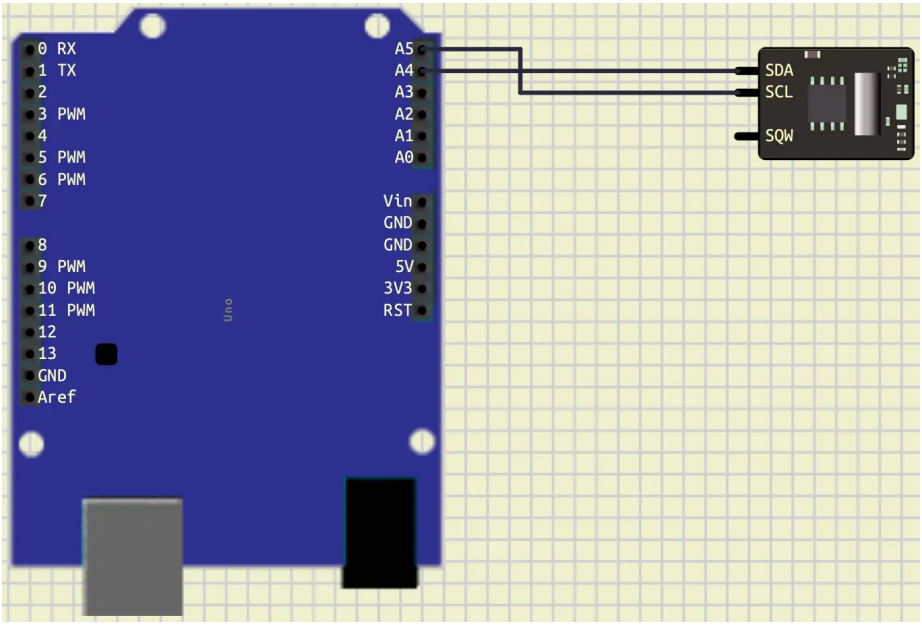


Figure 57: add-clock

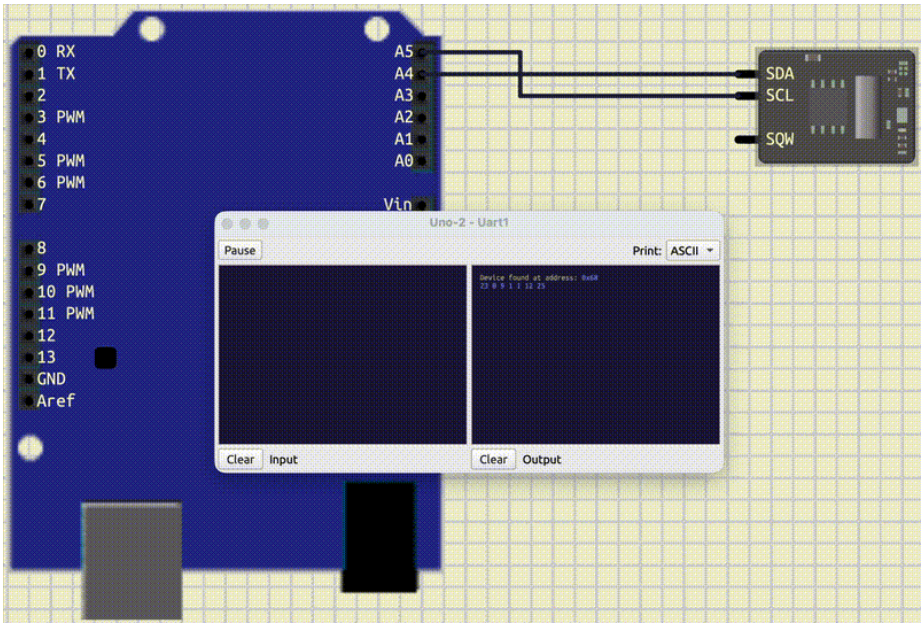
```
  Wire.begin();
  Serial.begin(9600);
}

void loop()
{
  for (int i = 0; i < 127; i++)
  {
    Wire.beginTransmission(i);
    if (Wire.endTransmission() == 0)
    {
      Serial.println("Device found at address: 0x" + String(i,
↪   HEX));
    }
  }
  delay(2000);
}
```

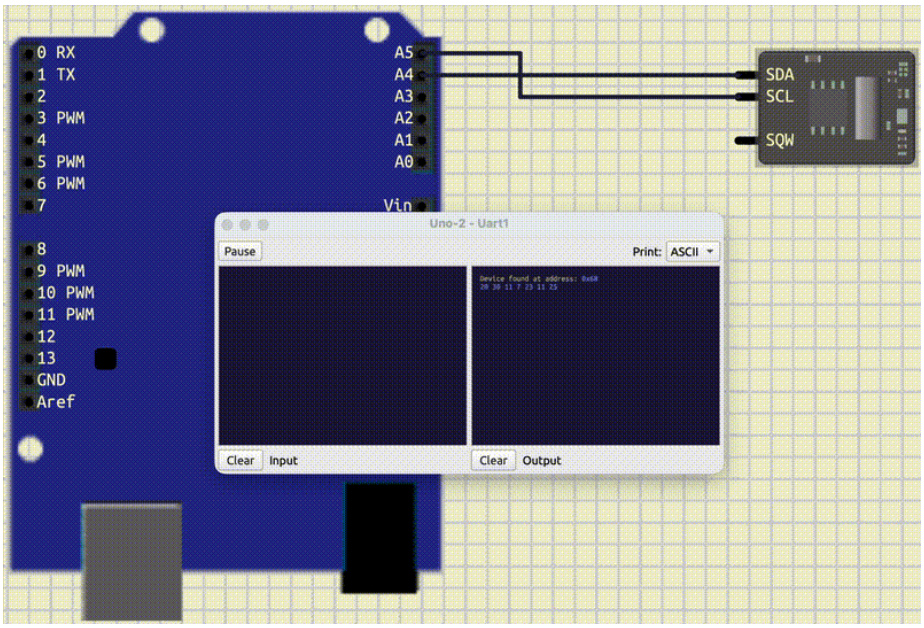
Clock: DS1307

- Storing: 0x22 -> 22 not 2*16+2
- seconds, minutes, hours, weekday, day, month, year

- SQW: Square Wave Output
  - Good for creating interrupts

Register	Address
Seconds	0x00
Minutes	0x01
Hours	0x02
Day of Week	0x03
Day of Month	0x04
Month	0x05
Year	0x06





Link to the Datasheet

OLED: SSD1306

```
lib_deps =
    Adafruit SSD1306
    Adafruit GFX Library
```

```
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
```

```
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64

Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire);
```

```
if (!display.begin(SSD1306_SWITCHCAPVCC, SSD1306_ADDRESS))
{
Serial.println("SSD1306 failed!");
for (;;)
  ;
}

display.setTextSize(1);
display.setTextColor(WHITE, BLACK);
```

```
display.clearDisplay();
display.setCursor(0, 0);
```

```
display.print();
display.display();
```

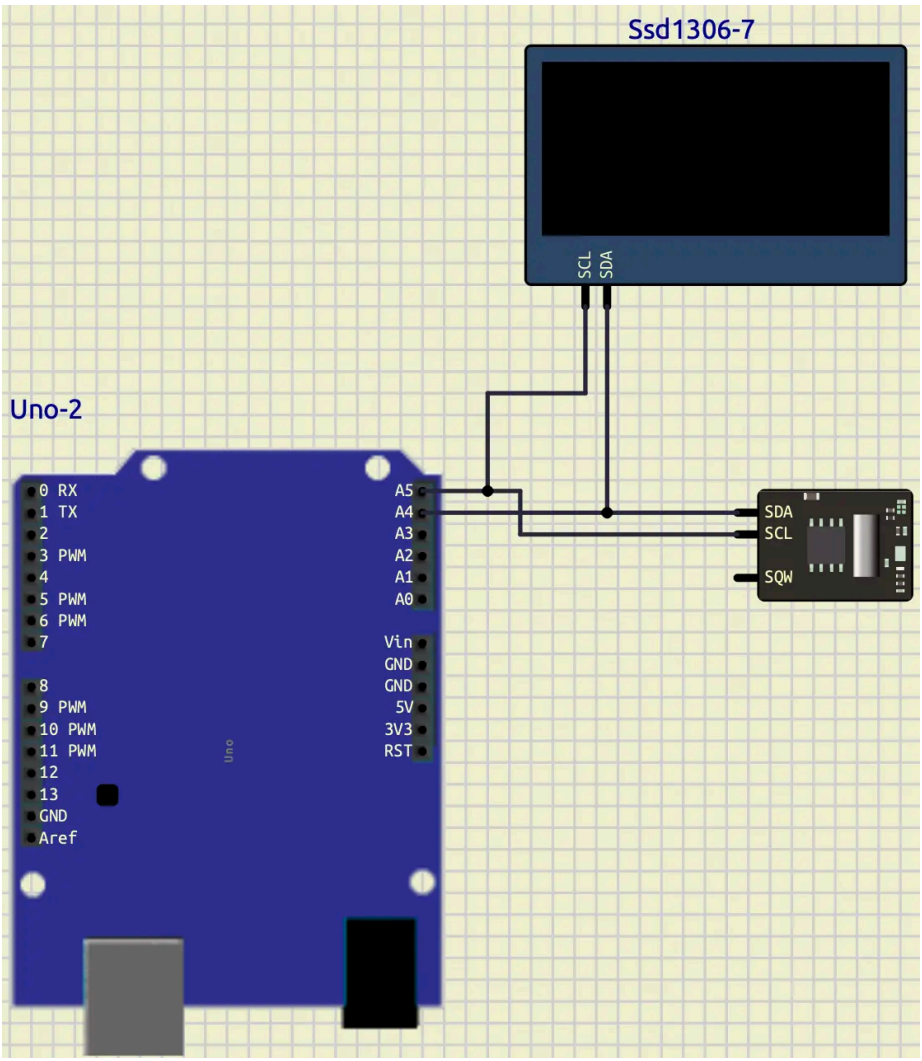


Figure 58: OLED

Good Example

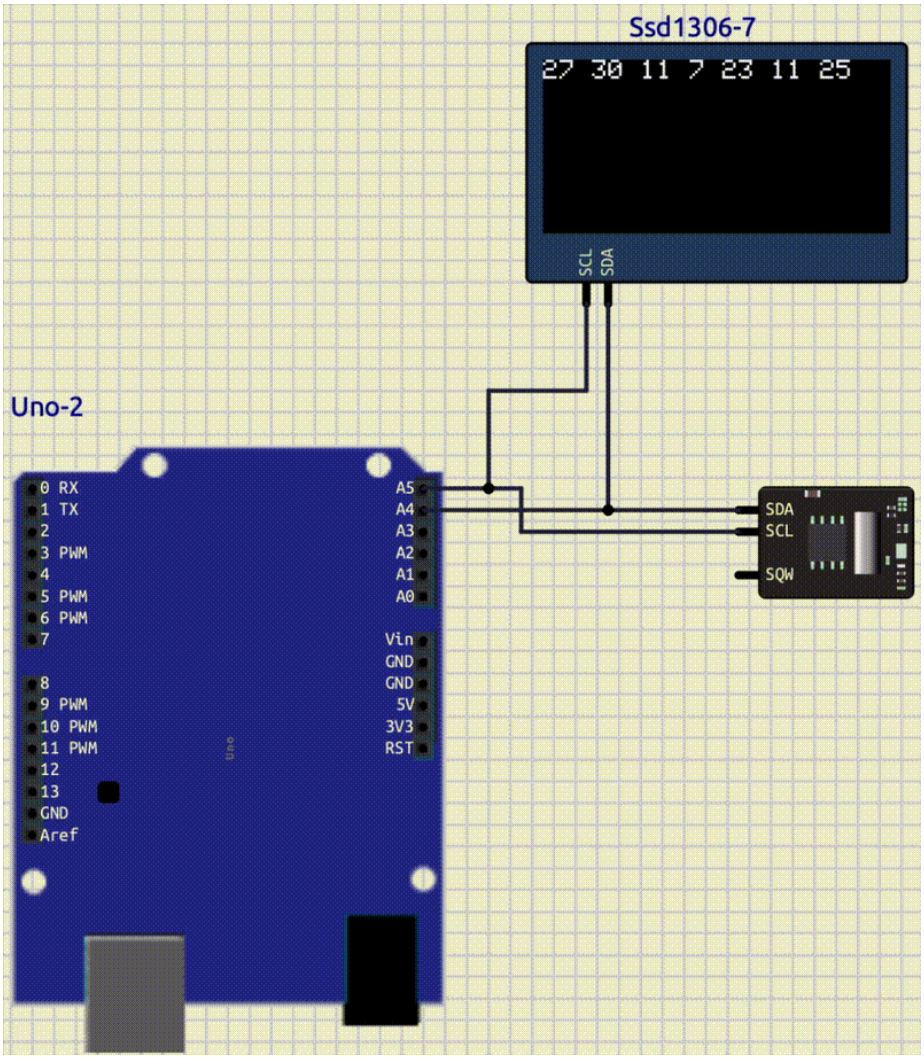


Figure 59: OLED gif

Project

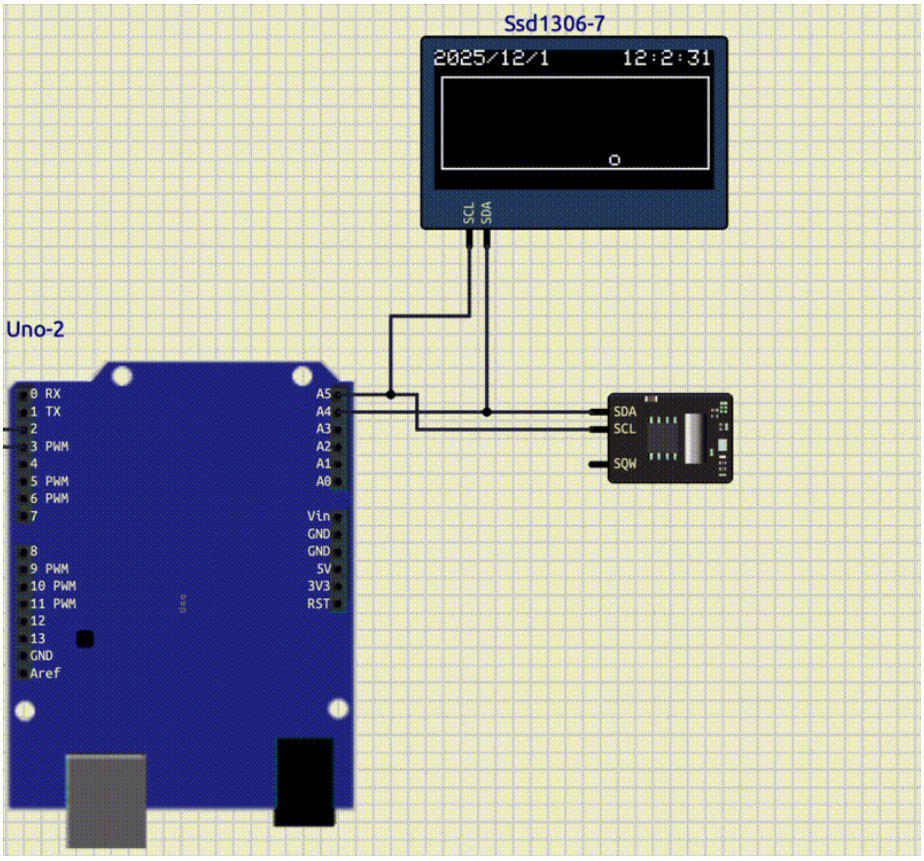


Figure 60: oled ball

Conclusion

I2C: part 2

Introduction

In the previous tutorial, we learned about the **I2C Communication**. Also, we introduced two components and learned how to communicate with them. To understand this way of communication better, let's work with another component and define an **Arduino** as a **slave**.

Temperature: DS1621

Link to the Datasheet

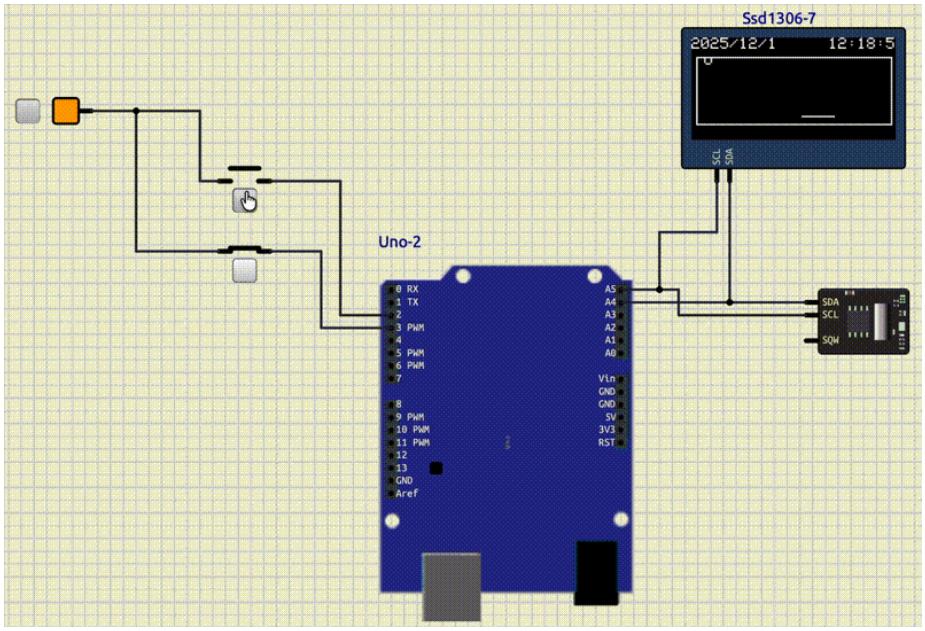


Figure 61: OLED Ball line

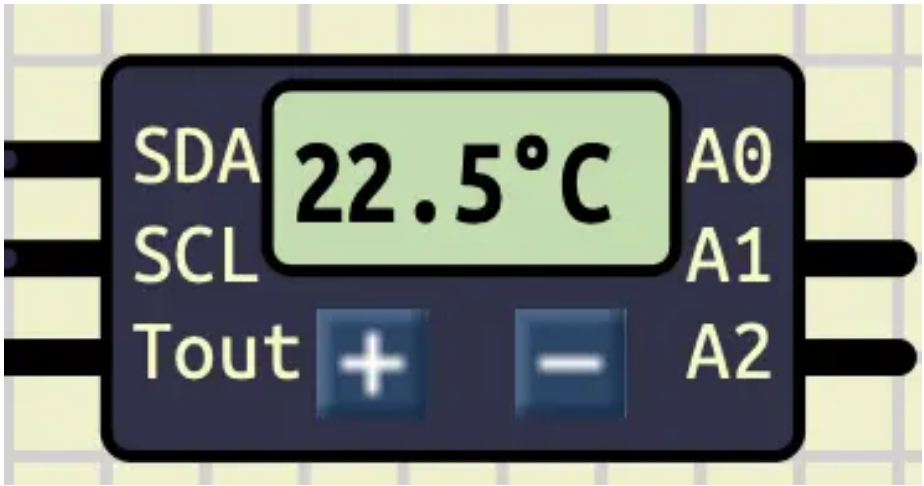


Figure 62: DS1621

Table 3. DS1621 COMMAND SET

INSTRUCTION	DESCRIPTION	PROTOCOL	2-WIRE BUS DATA AFTER ISSUING PROTOCOL	NOTES
TEMPERATURE CONVERSION COMMANDS				
Read Temperature	Read last converted temperature value from temperature register.	AAh	<read 2 bytes data>	
Read Counter	Reads value of Count_Remain	A8h	<read data>	
Read Slope	Reads value of the Count_Per_C	A9h	<read data>	
Start Convert T	Initiates temperature conversion.	EEh	idle	1
Stop Convert T	Halts temperature conversion.	22h	idle	1
THERMOSTAT COMMANDS				
Access TH	Reads or writes high temperature limit value into TH register.	A1h	<write data>	2
Access TL	Reads or writes low temperature limit value into TL register.	A2h	<write data>	2
Access Config	Reads or writes configuration data to configuration register.	ACh	<write data>	2

NOTES:

1. In continuous conversion mode a Stop Convert T command will halt continuous conversion. To restart the Start Convert T command must be issued. In one-shot mode a Start Convert T command must be issued for every temperature reading desired.

2. Writing to the E^2 requires a maximum of 10ms at room temperature. After issuing a write command, no further writes should be requested for at least 10ms.

Figure 63: Command Table

MSb	Bit 6	Bit5	Bit 4	Bit 3	Bit 2	Bit 1	LSb
DONE	THF	TLF	NVB	X	X	POL	1SHOT

Figure 64: Register byte

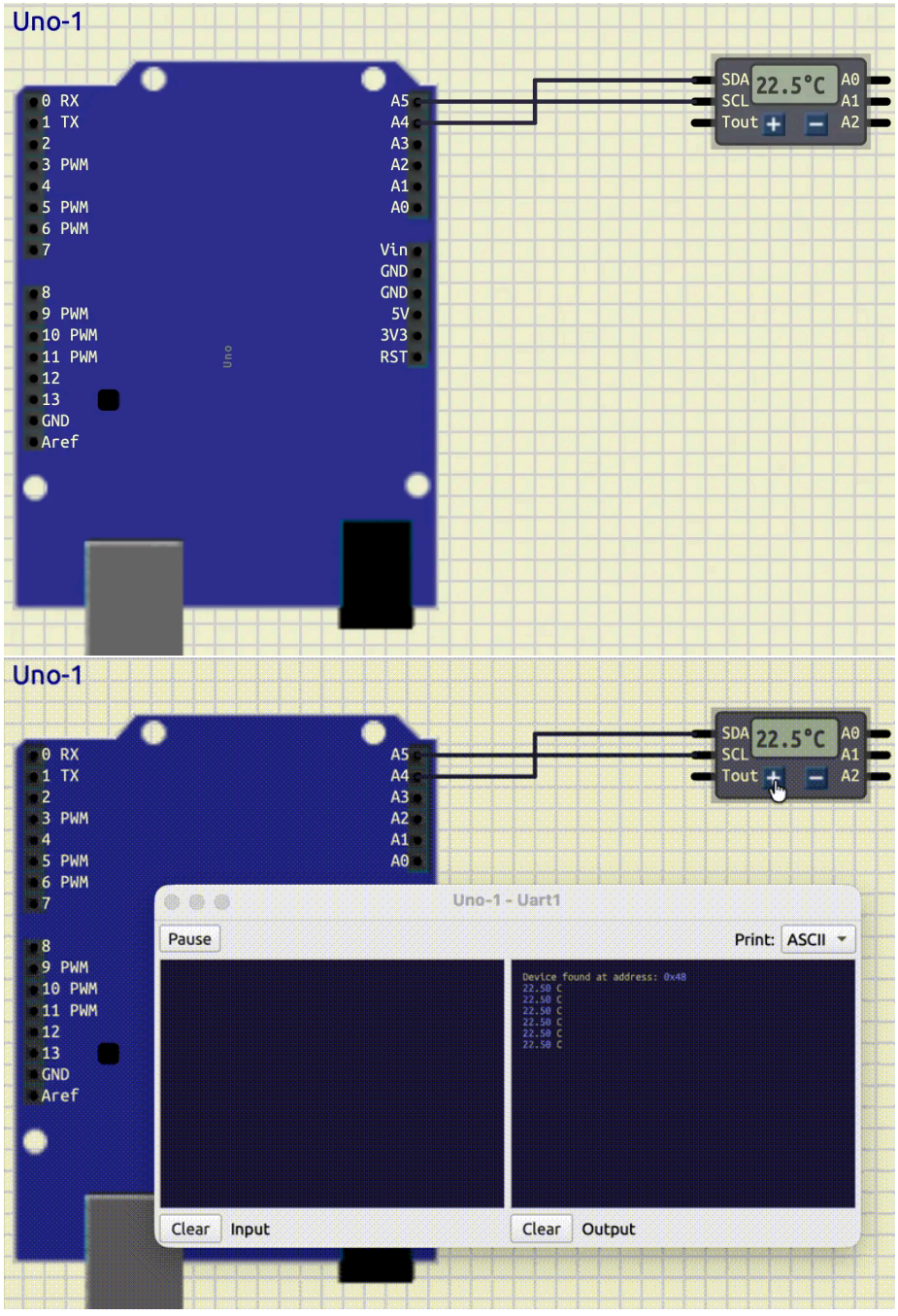
Table 2. TEMPERATURE/DATA RELATIONSHIPS

TEMPERATURE	DIGITAL OUTPUT (Binary)	DIGITAL OUTPUT (Hex)
+125°C	01111101 00000000	7D00h
+25°C	00011001 00000000	1900h
+½°C	00000000 10000000	0080h
+0°C	00000000 00000000	0000h
-½°C	11111111 10000000	FF80h
-25°C	11100111 00000000	E700h
-55°C	11001001 00000000	C900h

Figure 65: output of temperature

Command Table

Configuration registers



Arduino as an I2C Slave

- `onRequest`
- `onReceive`

Conclusion

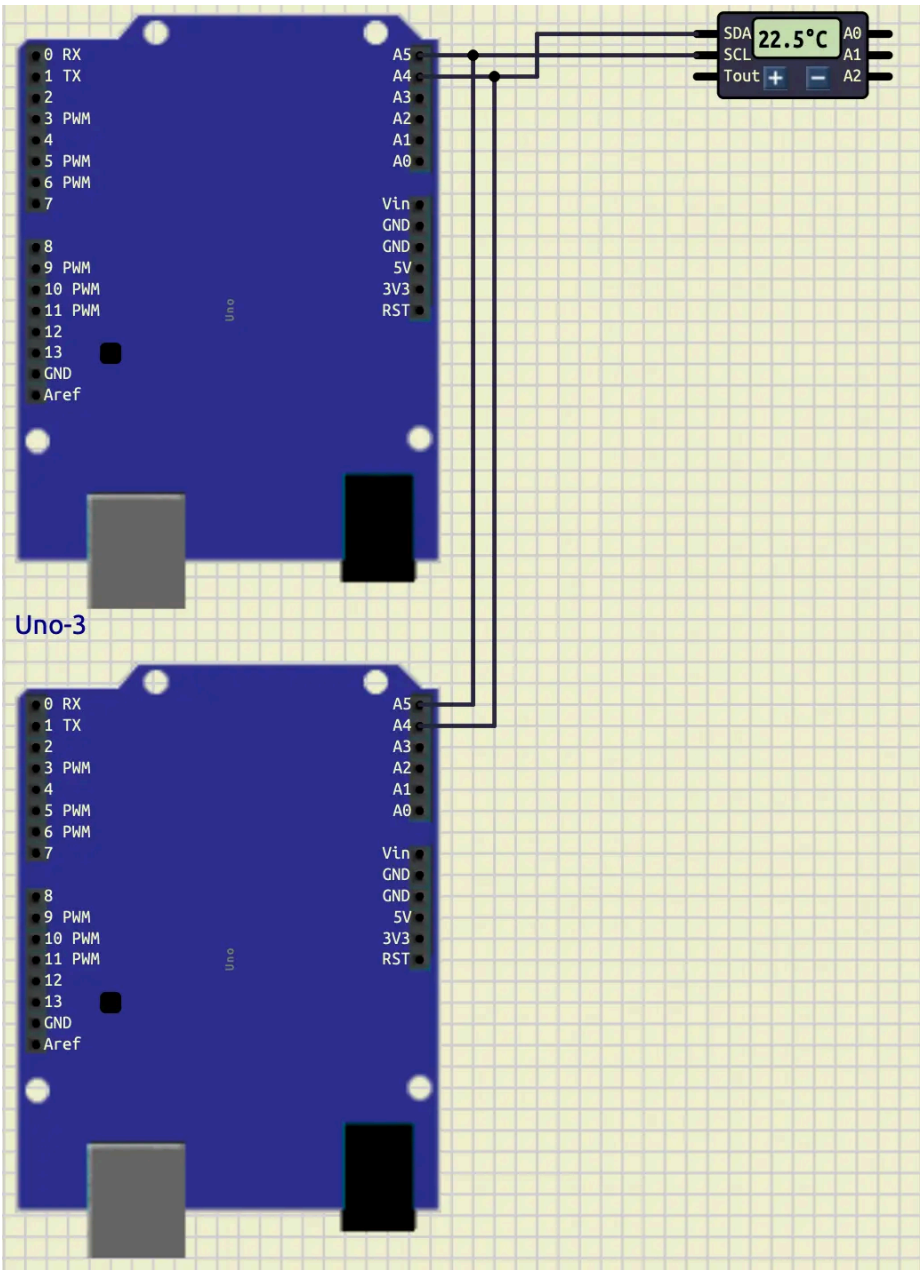


Figure 66: Arduino slave temperature

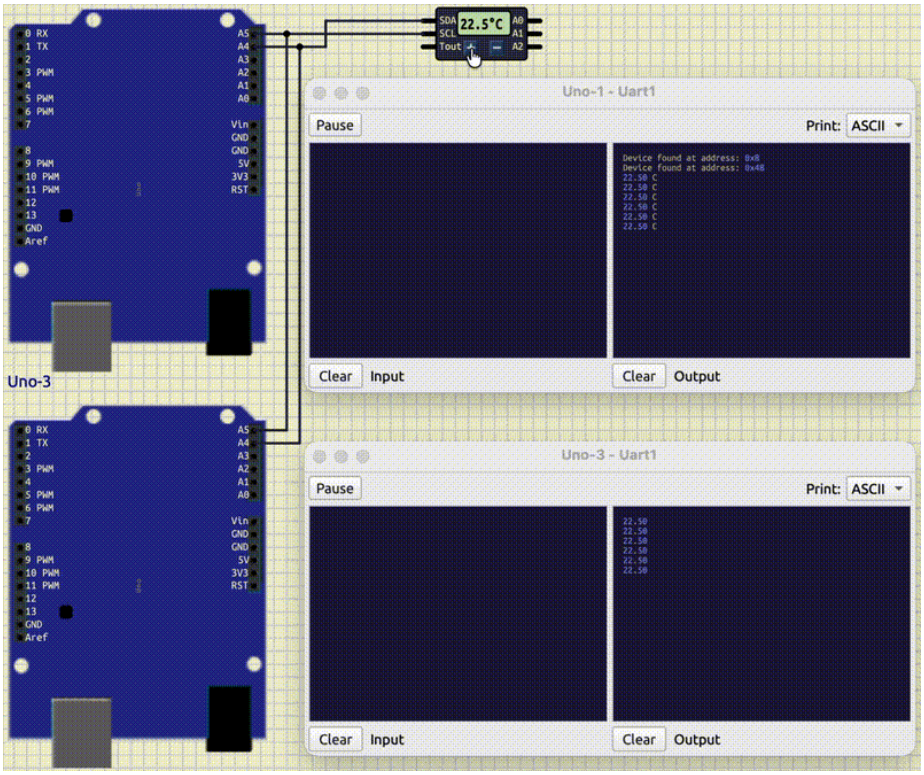


Figure 67: Arduino slave temperature gif

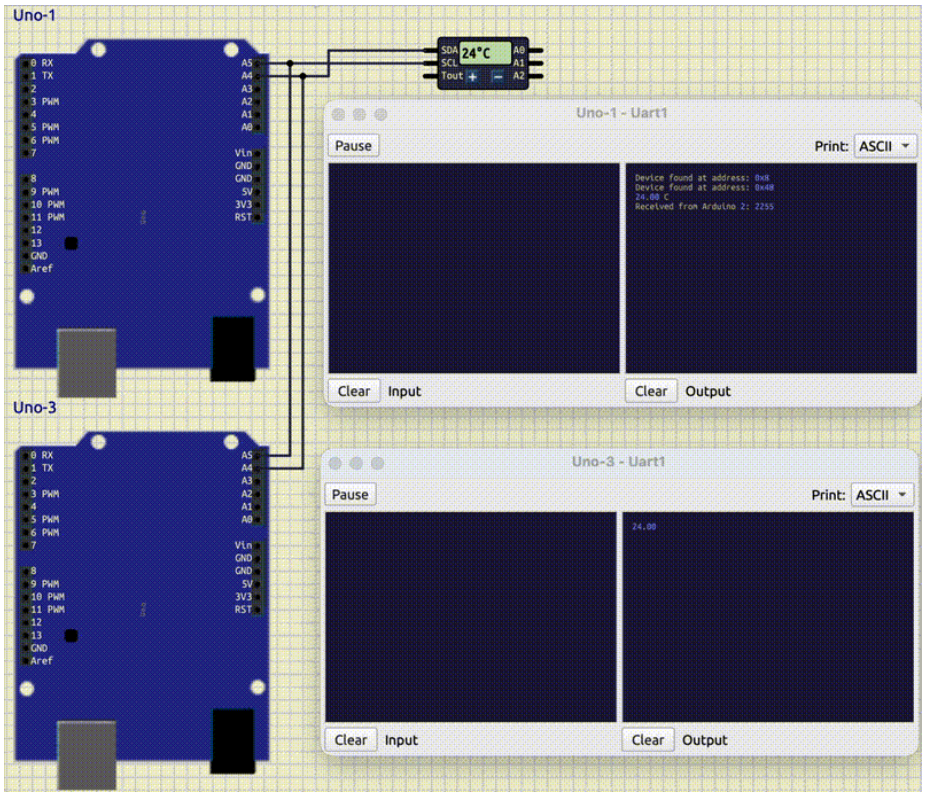


Figure 68: Arduino slave temperature request gif