Introduction to robotics

3rd lab

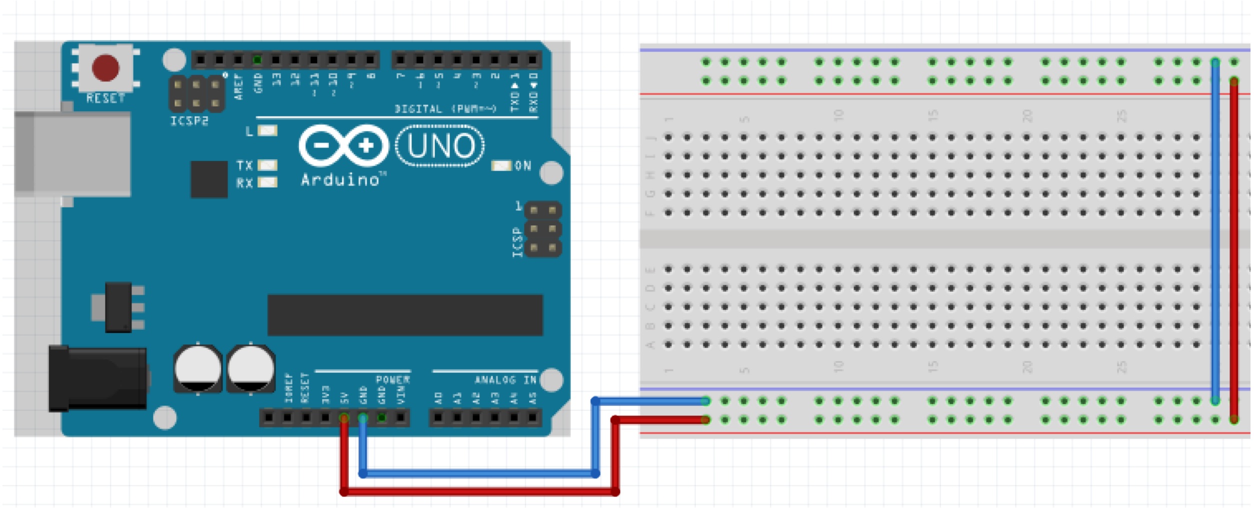
Remember, when possible, choose the wire color accordingly:

- **BLACK** (or dark colors) for **GND**
- **RED** (or colored) for **POWER (3.3V / 5V / VIN)**
- **Remember** than when you use digitalWrite or analogWrite, you actually send power over the PIN, so you can use the same color as for **POWER**
- **Bright Colored** for read signal
- We know it is not always possible to respect this due to lack of wires, but the first rule is **NOT USE BLACK FOR POWER OR RED FOR GND!**

Now, let's pick it up where we left off…

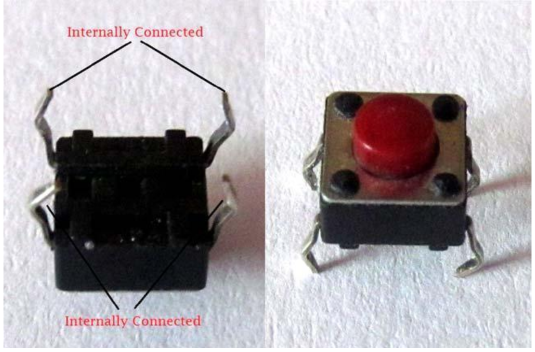
Pull out your Arduino and breadboard and connect them like in the schematic. This is to "power up" the breadboard so we can easily have access to **5V** and **GND**.

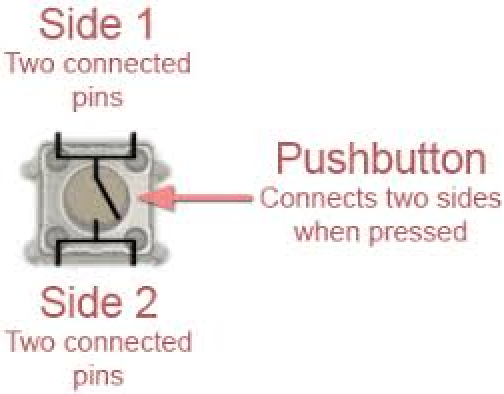
Attention! Remember how the breadboard works. Use correct wire colors.



1. Pushbuttons

Let's recap the course a bit: What is a push button?

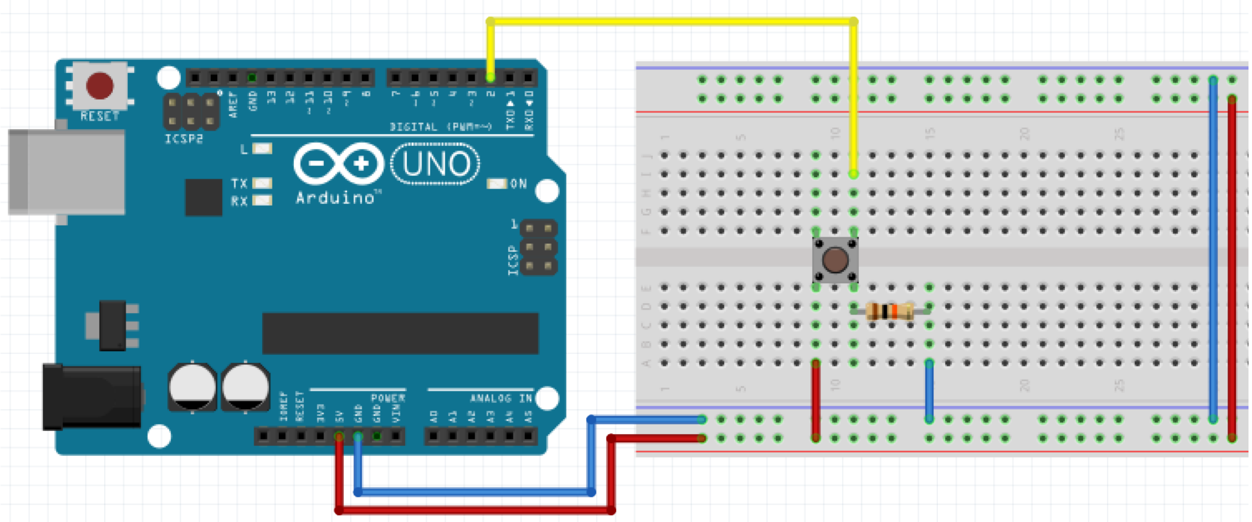




Always remember that the oppositely oriented pins are connected.

https://docs.arduino.cc/built-in-examples/basics/DigitalReadSerial
Pushbuttons or switches connect two points in a circuit when you press them. When the pushbutton is open (unpressed) there is no connection between the two legs of the pushbutton, so the pin is connected to ground (through the pull-down resistor) and reads as **LOW**, or **0**. When the button is closed (pressed), it makes a connection between its two legs, connecting the pin to 5 volts, so that the pin reads as **HIGH**, or **1**.

Connect the button, wires and the resistor like in the schematic below. **Make sure you connect the 5V and GND to the red and blue columns on the breadboard.**



5k or **10k** ohm resistors work well

```
const int buttonPin = 2;
byte buttonState = 0;

void setup() {
  // make the pushbutton's pin an input:
  pinMode(buttonPin, INPUT);
  Serial.begin(9600);
}

void loop() {
  // read the input pin:
  // TODO
  // print out the state of the button:
  Serial.println(buttonState);
  delay(1);     // delay in between reads for stability
}
```

Questions:

1. What happens if we remove the connection to **GND**?
    a.
2. So what do we do, in this example, to counter this?
    a.
3. What else could we do?
    a.

> In electronic logic circuits, a pull-up resistor or pull-down resistor is a resistor used to ensure a known state for a signal. It is typically used in combination with components such as switches and transistors, which physically interrupt the connection of subsequent components to ground or to VCC. When the switch is closed, it creates a direct connection to ground or VCC, but when the switch is open, the rest of the circuit would be left floating (i.e., it would have an indeterminate voltage). For a switch that connects to ground, a pull-up resistor ensures a well-defined voltage (i.e. VCC, or logical high) across the remainder of the circuit when the switch is open. Conversely, for a switch that connects to VCC, a pull-down resistor ensures a well-defined ground voltage (i.e. logical low) when the switch is open.

(source: https://en.wikipedia.org/wiki/Pull-up_resistor)

2. Controlling an LED with the pushbutton

https://docs.arduino.cc/built-in-examples/digital/Button

The idea is simple: reading the button value returns **HIGH** or **LOW** which we can write directly to the ledPin. For this, we can use ledPin 13 (and add an LED directly on pin 13 and the **GND** next to it).

```
const int buttonPin = 2;
const int ledPin = 13;

byte buttonState = 0;

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

void loop() {
  buttonState = digitalRead(buttonPin);
  // TODO
}
```

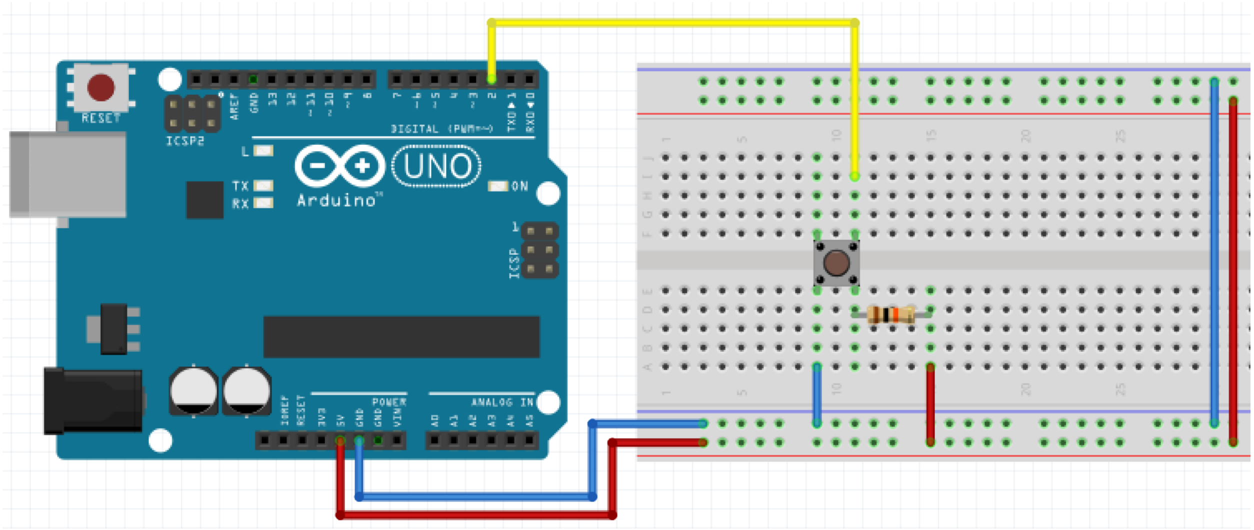
Works great and it’s quite simple: we digitally read the value of the button, which can be either **HIGH** or **LOW** and directly digitally write it to the **LED** value which also accepts the values **HIGH** or **LOW.** Seeing that we used a PULLDOWN resistor, the default button value is **LOW** and the default led state is off.

Now, let’s try to switch it to a **PULLUP resistor**.

Only a small change is needed.

Questions:

1. Which one?
    a. **A:** Connect the wire with the resistor to **5V** and the other one to **GND.**
2. What do you expect to happen now?



Be careful and note the different connections to the button

Questions:

1. Using the same code, what happens now?
    a.
2. Why?
    a.

Here, we can keep the same code, or go for 3 variations:

1. We can negate the read value by adding “!” in front of the digitalRead(buttonState).
2. We use an if-else statement and digitalWrite **LOW when reading HIGH** and else.
3. We can create a new variable, ledState which receives the negated buttonState.

As your projects increase in complexity you will start doing more with the value of the button. That is why it is good practice to instantiate a **ledState** variable from the beginning and digitalWrite it to the ledPin, instead of writing the buttonValue to the LED.

We’ll go for version no. 3 - because it is a clean variant - but show code snippets on version 1 and 2 as well.

```
const int buttonPin = 2;
const int ledPin = 13;

byte buttonState = 0;
byte ledState = 0;

void setup() {
```

```
  pinMode(buttonPin, INPUT);
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  buttonState = digitalRead(buttonPin);
  ledState = !buttonState;
  digitalWrite(ledPin, ledState);
  Serial.println(buttonState);
}
```

#1

```
buttonState = !digitalRead(buttonPin);
```

#2

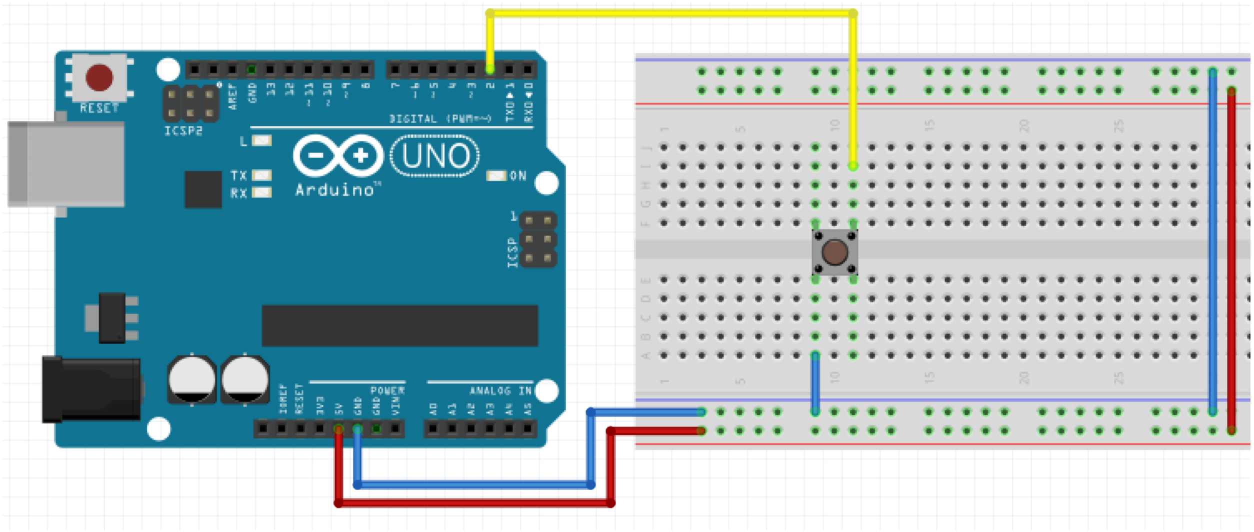
```
if (digitalRead(buttonPin) == HIGH) {
      digitalWrite(ledPin, LOW);
}
else {
      digitalWrite(ledPin, HIGH);
}
```

3. Internal PULLUP resistor

We've started with a **hardware pulldown resistor**, switched it to a **hardware pullup** and now we'll see a way to remove the resistor completely.
Let's do so: remove the resistor and the button's connection to **5V** completely. Again, the values read on the button pin are floating. Meet **INPUT_PULLUP**.

> There are 20K pullup resistors built into the Atmega chip that can be accessed from software. These built-in pullup resistors are accessed by setting the pinMode() as INPUT_PULLUP. This effectively inverts the behavior of the INPUT mode, where HIGH means the sensor is off, and LOW means the sensor is on.
>
> The value of this pullup depends on the microcontroller used. On most AVR-based boards, the value is guaranteed to be between 20kΩ and 50kΩ. On the Arduino Due, it is between 50kΩ and 150kΩ. For the exact value, consult the datasheet of the microcontroller on your board.

source: https://www.arduino.cc/en/Tutorial/DigitalPins



```
const int buttonPin = 2;
const int ledPin = 13;

byte buttonState = 0;
byte ledState = 0;

void setup() {
  pinMode(buttonPin, // TODO);
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  buttonState = digitalRead(buttonPin);
  ledState = !buttonState;
  digitalWrite(ledPin, ledState);

  Serial.println(buttonState);
}
```

Questions:

1. What do you think of this solution?
    a.
2. What are the advantages?
    a.
3. What are its drawbacks?
    a.

From now on, we'll be using the button this way, when possible.

4. Button Press Counter

Now that we've learned how a button works, how we can use it in multiple ways and understood some of its inner workings, let's do a button press counter.

```
const int buttonPin = 2;
const int ledPin = 13;
byte buttonState = 0;
// TODO

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

void loop() {
  // read the input pin:
  buttonState = digitalRead(buttonPin);
  ledState = !buttonState;
  digitalWrite(ledPin, !ledState);
  // TODO
  Serial.println(buttonPushCounter);
}
```

Press the button, lifting the finger as fast as possible.

Questions:

1. Count the number of times the LED lights. How do you think this value compares to the counter?
    a.
2. How does the counter grow?
    a.
3. Why?
    a.
4. How can we fix this?
    a.

5. State change detector

Let’s do a simple state change detector, saving the current and the previous state and comparing them to know if a change occurred.

```
const int buttonPin = 2;
const int ledPin = 11;

byte buttonState = LOW;
byte ledState = LOW;
byte lastButtonState = LOW;

int buttonPushCounter = 0;

void setup() {
  // put your setup code here, to run once:
  pinMode(buttonPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
  buttonState = digitalRead(buttonPin);
  // TODO
}
```

Questions:

1. What are the drawbacks of this method?

   A: It uses delay, for starters. But it is still sensitive to noise.

6. Debounce

https://www.arduino.cc/en/Tutorial/Debounce
Pushbuttons often generate spurious open/close transitions when pressed, due to mechanical and physical issues: these transitions may be read as multiple presses in a very short time fooling the program. This example demonstrates how to debounce an input, which means checking twice in a short period of time to make sure the pushbutton is definitely pressed. Without debouncing, pressing the button once may cause unpredictable results. This sketch uses the millis() function to keep track of the time passed since the button was pressed.

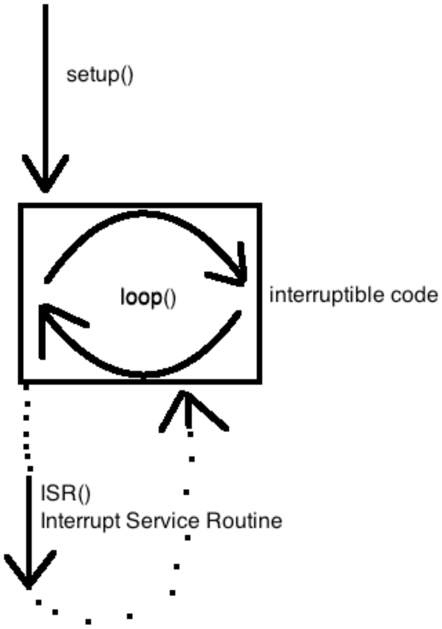
```
const int buttonPin = 2;
const int ledPin = 11;

byte buttonState = LOW;
byte ledState = HIGH;
int buttonPushCounter = 0;
byte lastButtonState = LOW;

unsigned int lastDebounceTime = 0;
unsigned int debounceDelay = 50;

void setup() {
  // put your setup code here, to run once:
  pinMode(buttonPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  // TODO
}
```

We finally learned how to turn the light on and off with a pushbutton.

7. Interrupts (mostly home exercise)

Interrupts, in arduino, work the following way:

- in setup, we tell the processor that we are open to interrupts (it is open by default in Arduino)
- from then on, whenever a button push is detected, the program stops whatever it is doing and executes the interrupt
- we tell the microcontroller how to do this by writing a special function, an ISR() - Interrupt Service Routine
- it is not doing two things at the same time: the program stops, saves any information it might need, goes and executes the interrupt, returns, loads the data and continues

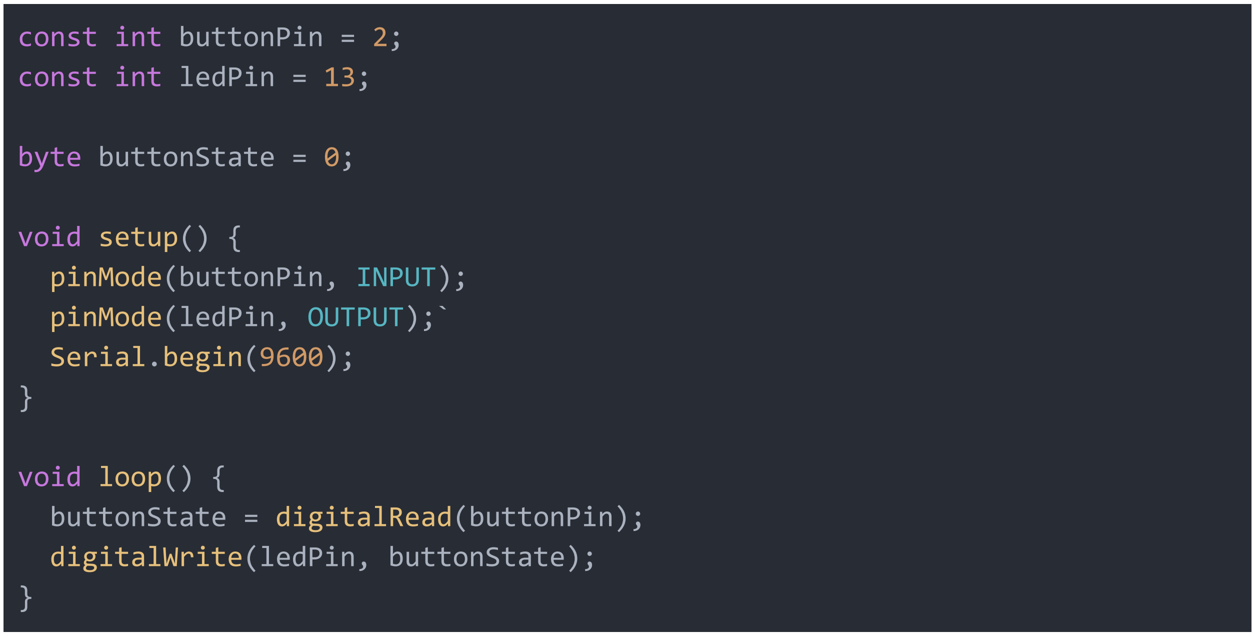


In arduino, we use the function attachInterrupt(interruptNumber, ISR(), STATE)

- interruptNumber - this is the interrupt number, not the same as the pin. Also check digitalPinToInterrupt()
- ISR() - the function that is called
- STATE - on which state to call the interrupt

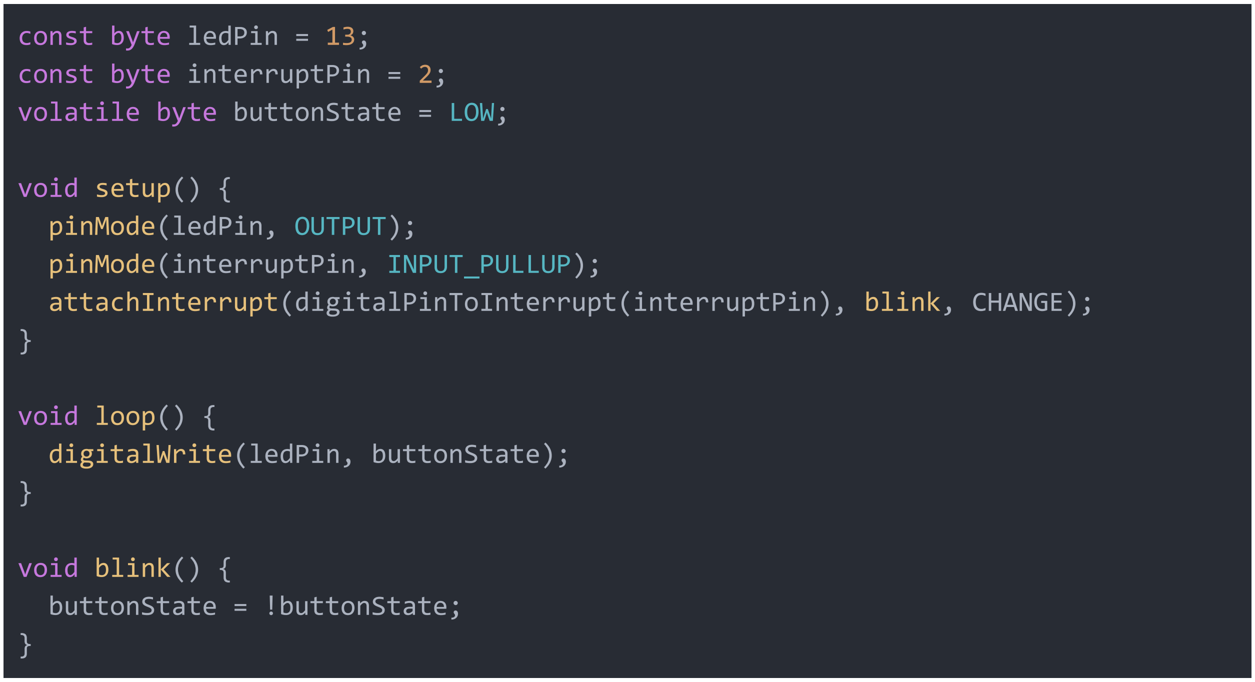
https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/

You remember the code for the ledState change from before?



```
const int buttonPin = 2;
const int ledPin = 13;

byte buttonState = 0;

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

void loop() {
  buttonState = digitalRead(buttonPin);
  digitalWrite(ledPin, buttonState);
}
```

How can we do it with an interrupt?



```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte buttonState = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
  digitalWrite(ledPin, buttonState);
}

void blink() {
  buttonState = !buttonState;
}
```

HOME ASSIGNMENT: Program button press counter, state change detector and debounce using an interrupt instead of polling.

8. Basic sounds

8.1 Active and passive piezo buzzers

In your kit you have **2 buzzers, one passive and one active.**
They look quite similar, so be careful when telling them apart:

- The height of the two is slightly different, the active buzzer has a height of 9mm, while the passive has height of 8mm.
- if you apply a DC voltage to them and it buzzes, it is active.
- Pins side, you can see a green circuit board in the passive buzzer. No circuit board and closed with a black is an active buzzer.
- Using an ohmmeter, if you find the resistance between buzzer pins is about 16ohm (or 8ohm), the buzzer is passive.

Active buzzer Passive buzzer

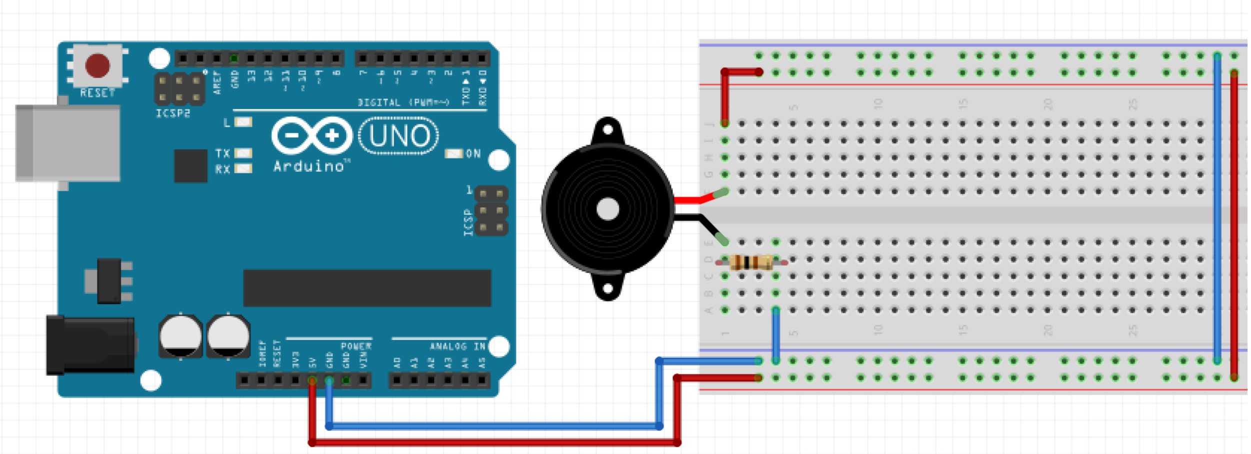
-

Although they look similar, inside they are quite different. But let's start with what they have in common: they are both **piezo buzzers.** A piezo buzzer has a thin piezoelectric plate inside it

that vibrates mechanically whenever a voltage is applied to it. It's the same principle as a quartz crystal that is used in watches, that vibrates whenever energy is applied to it.
They are sometimes called buzzers or speakers, but buzzers is more appropriate since they can only play tones, and not complicated sound effects.

As far as functionality is concerned, the **active speaker has active electrical components built into it.** The passive one is **just piezoelectric material** so it needs active components externally in order to generate the wave and work.

Let's make the simplest connection we can make. You can recognise the + and **-** pins by their length (+ is longer) or you can look at the case, as they both have a **+** written on the top.



Use a small resistor, no higher than **100 ohms**

First, let's connect the **active** buzzer (careful, this will be annoying). **Then**, connect the **passive** buzzer (don't worry this time).
Questions:

1. Why didn't the passive buzzer play the same tone?
    a.

It's time to play some music. Let's play a little melody that you might've heard before.

Now, let's control the tone. Connect the **+** to **D11.** You can use either the passive or the active buzzer, but let's use the passive one just to see it works.

8.2 Introducing the tone() function

The **tone()** function generates a square wave of the specified frequency (and 50% duty cycle) on a pin. A duration can be specified, otherwise the wave continues until a call to noTone(). The pin can be connected to a piezo buzzer or other speaker to play tones.

Only one tone can be generated at a time. If a tone is already playing on a different pin, the call to tone() will have no effect. If the tone is playing on the same pin, the call will set its frequency.

Attention! Use of the tone() function will interfere with PWM output on pins 3 and 11.

Syntax:

```
tone(pin, frequency)
tone(pin, frequency, duration)
```

Parameters
pin: the Arduino pin on which to generate the tone.
frequency: the frequency of the tone in hertz. Allowed data types: unsigned int.
duration: the duration of the tone in milliseconds (optional). Allowed data types: unsigned long.

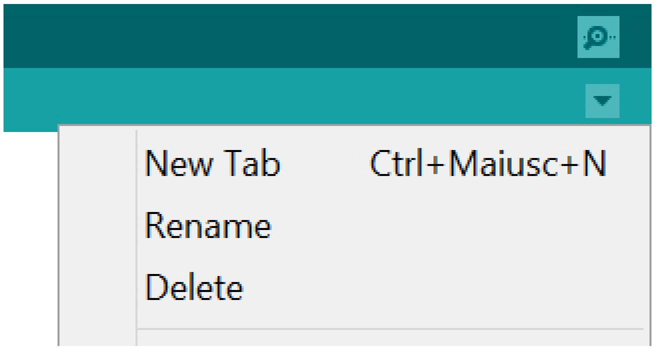
(source: https://www.arduino.cc/en/Tutorial/toneMelody)

The tone() function is all that is needed to play a tone. **Careful, this gets annoyingly fun fast!**

```
const int buzzerPin = 11;
int buzzerTone = 1000;

void setup() {
}

void loop() {
  tone(buzzerPin, buzzerTone, 500);
}
```

Now, let’s cycle through the tones a bit.

Prerequisite: in your project, create a new tab and add these: (CTRL + SHIFT + T)



```
/*************************************************
 * Public Constants
 *************************************************/
```

```
#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
```

```
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
#define NOTE_E5  659
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
#define NOTE_FS7 2960
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
```

```
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978
```

```
const int buzzerPin = 11;
int buzzerTone = 1000;

void setup() {

}

void loop() {
  tone(buzzerPin, buzzerTone, 500);
  delay(100);

  noTone(buzzerPin);
  delay(100);
  buzzerTone += 50;
}
```

Change both delays to 10 instead of 100.

Question:

1. Where would these be useful?
    a. **A**: in a game where you need sounds (a matrix game is coming), in alarms and generally in projects where you need audio feedback.

Copy paste this code. Source: https://www.arduino.cc/en/Tutorial/toneMelody (with modified pin)

```
#include "pitches.h"
const int buzzerPin = 11;
// notes in the melody:
int melody[] = {
  NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3, NOTE_C4
};

// note durations: 4 = quarter note, 8 = eighth note, etc.:
int noteDurations[] = {
  4, 8, 8, 4, 4, 4, 4, 4
```

```
};

void setup() {
  // iterate over the notes of the melody:
  for (int thisNote = 0; thisNote < 8; thisNote++) {

      // to calculate the note duration, take one second divided by the note type.
      //e.g. quarter note = 1000 / 4, eighth note = 1000/8, etc.
      int noteDuration = 1000 / noteDurations[thisNote];
      tone(buzzerPin, melody[thisNote], noteDuration);

      // to distinguish the notes, set a minimum time between them.
      // the note's duration + 30% seems to work well:
      int pauseBetweenNotes = noteDuration * 1.30;
      delay(pauseBetweenNotes);
      // stop the tone playing:
      noTone(8);
  }
}

void loop() {
  // no need to repeat the melody.
}
```

Now, let’s do a tone pitch follower