

Introduction to robotics

3rd lab

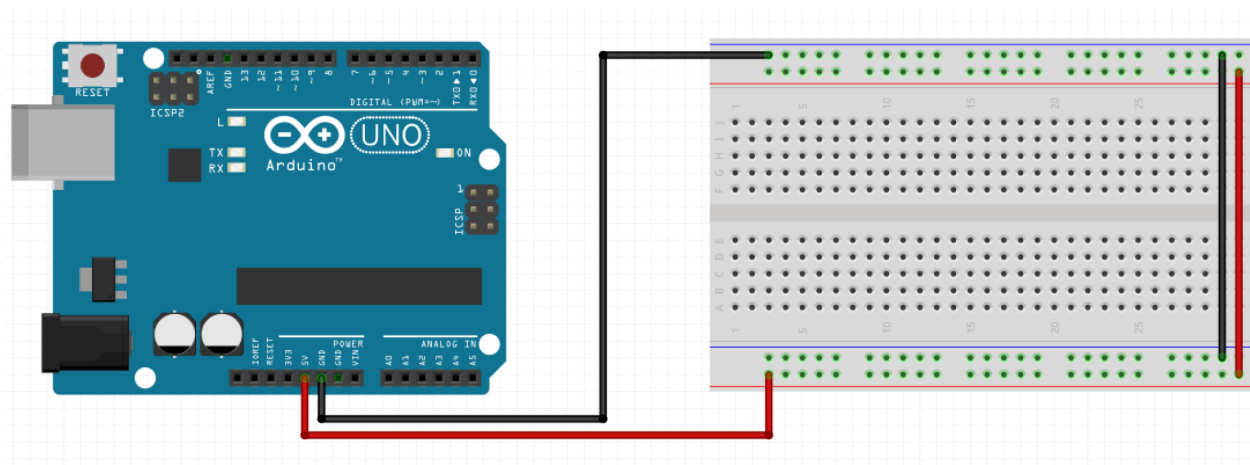
Remember, when possible, choose the wire color accordingly:

- **BLACK** for **GND** (dark colors if not available)
- **RED** for **POWER (3.3V / 5V / VIN)** (bright colors if not available)
- **Bright Colored** for read and write signal (use **red** when none available and **black** only as a last option)
- We know it is not always possible to respect this due to lack of wires, but the first rule is **DO 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.

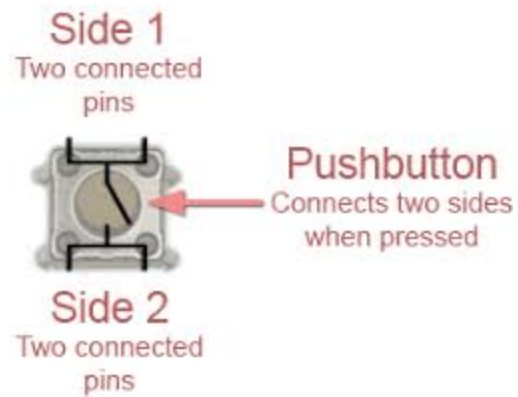
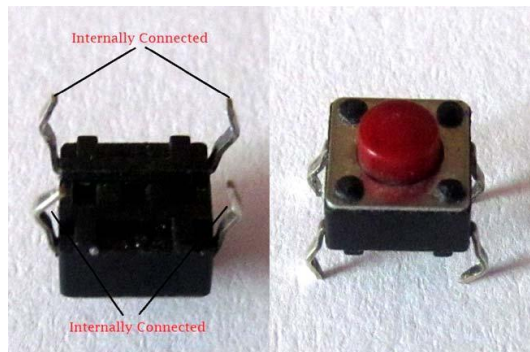


0. Homework check

1. Understanding and Working with Pushbuttons

1.1 Pushbuttons

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



Always remember that the oppositely oriented pins are connected.

Connect a button on the breadboard and use a multimeter to measure the connected pins when pressed or not pressed.

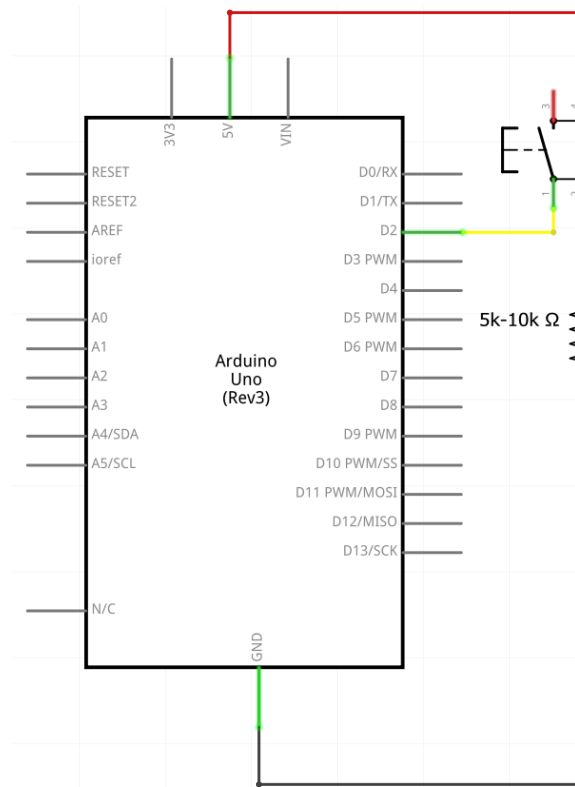
1.2 Reading Pushbutton Values with Pulldown Resistors

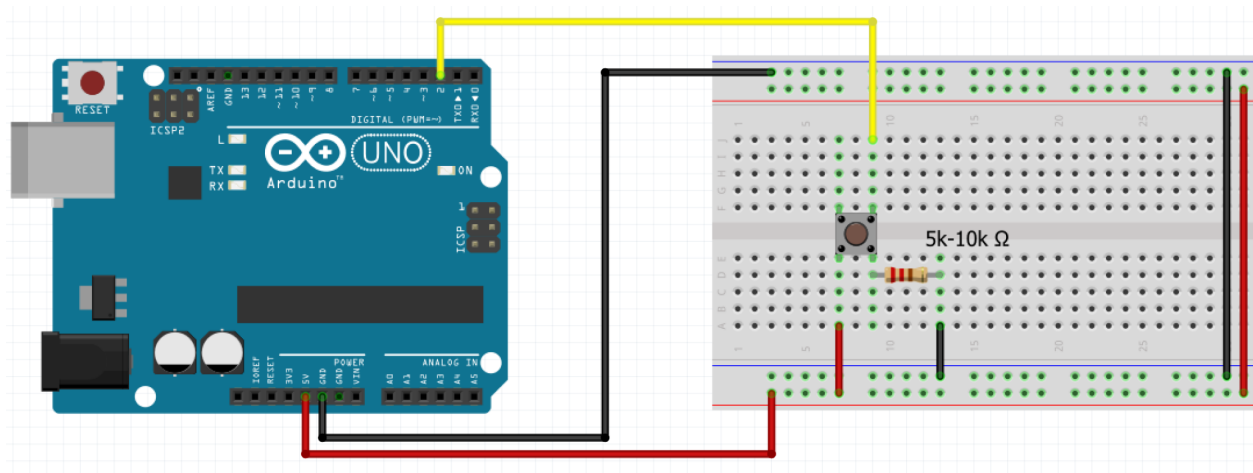
Pushbuttons or switches connect two points in a circuit when you press them. When the pushbutton is open (unpressed) there is no connection between the two legs of the pushbutton, so the pin is connected to ground (through the pull-down resistor) and reads as **LOW**, or **0**. When the button is closed (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.**

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





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);
}

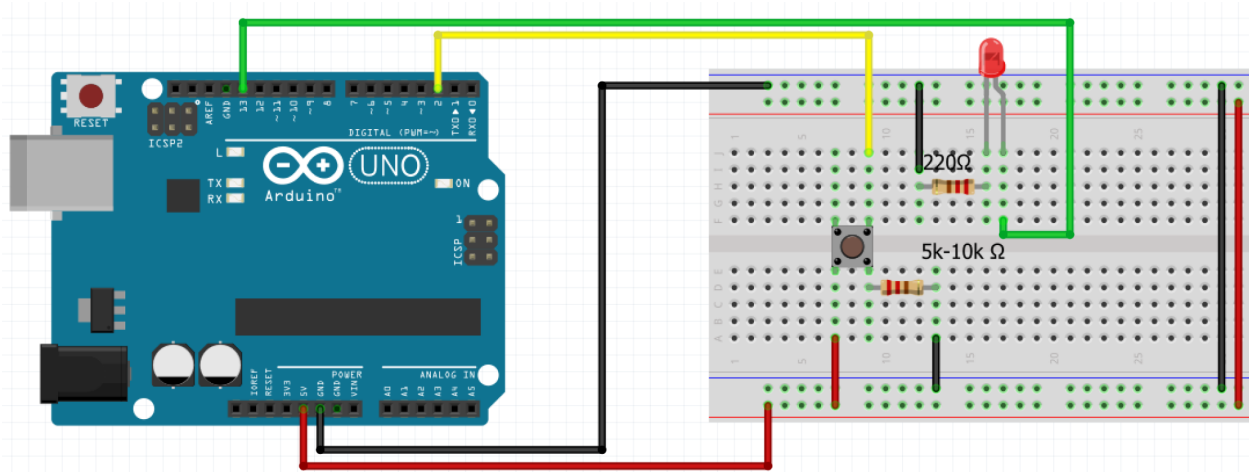
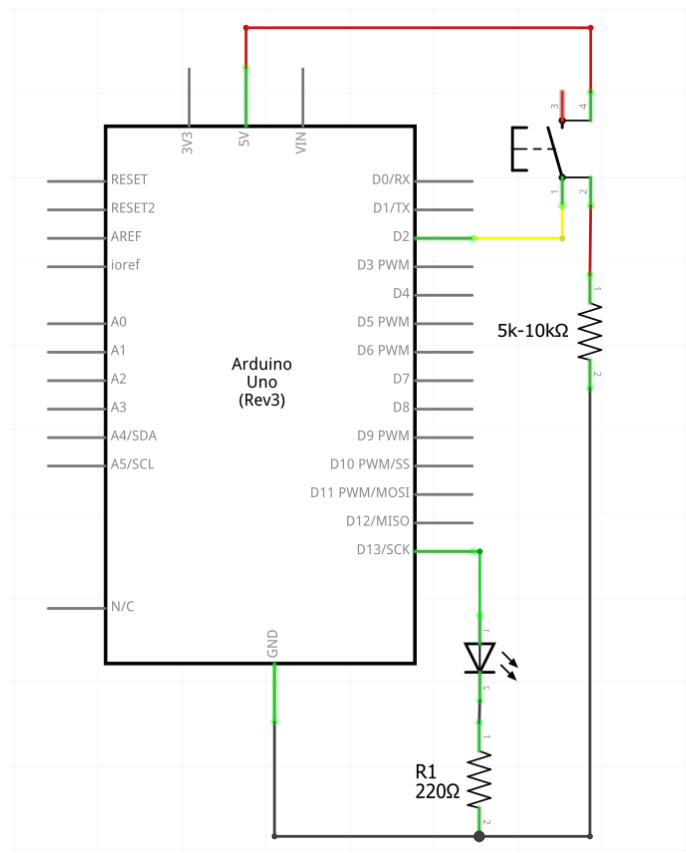
void loop() {
  // read the input pin:
  buttonState = digitalRead(buttonPin);
  // print out the state of the button:
  Serial.println(buttonState);
  delay(1); // delay in between reads for stability
}
```

Source: <https://docs.arduino.cc/built-in-examples/basics/DigitalReadSerial>

Questions:

1. What happens if we remove the connection to **GND**?
 - a. **A:** This is called a “Floating Pin”. Digital Input pins are very sensitive to change, and unless positively driven to one state or another (HIGH or LOW), will pick up stray capacitance from nearby sources, like breadboards, human fingers, or even the air. Any wire connected to it will act like a little antenna and cause the input state to change.
2. So what do we do, in this example, to counter this?
 - a. **A:** We drive the state to **LOW**, by sinking current to GND, with a **PULLDOWN** resistor.
3. What else could we do?
 - a. **A:** Drive the value to HIGH, with a **PULLUP** resistor.

1.3 Lab exercise 1: controlling a led with the pushbutton



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);
  digitalWrite(ledPin, buttonState);
}
```

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

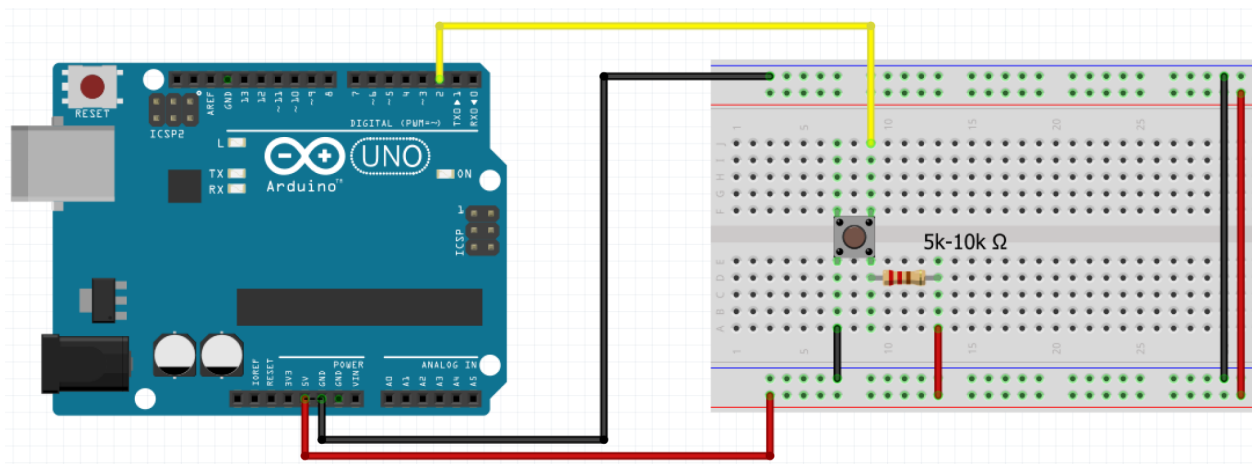
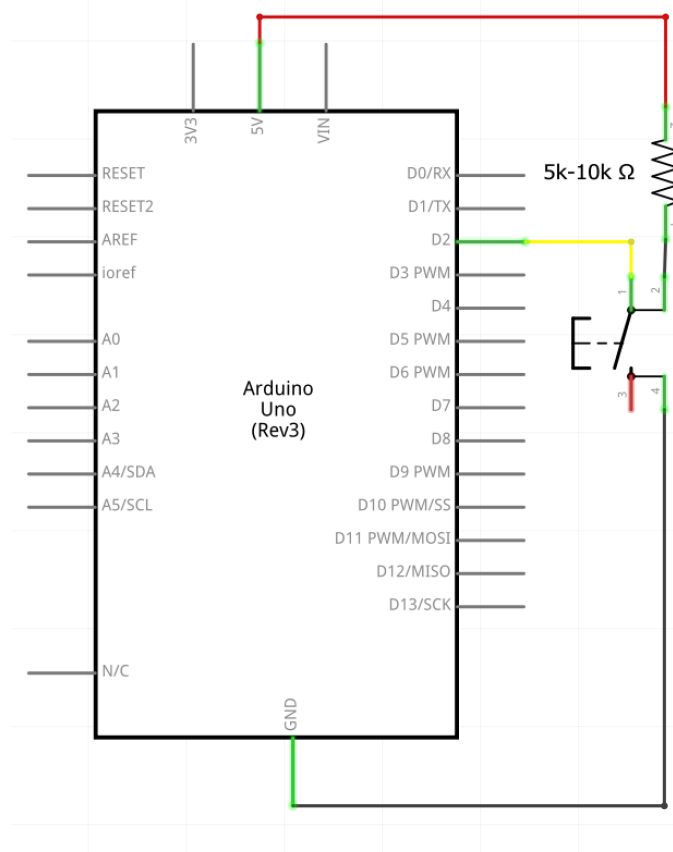
Works great and it's quite simple: we digitally read the value of the button, which can be either **HIGH** or **LOW** and directly digitally write it to the **LED** value which also accepts the values **HIGH** or **LOW**. Seeing that we used a PULLDOWN resistor, the default button value is **LOW** and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

1.4 Changing from PULLDOWN to PULLUP

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?



Be careful and note the different connections to the button

Questions:

1. Using the same code, what happens now?
 - a. **A:** The default button state (not pressed) is **HIGH** and the pressed value is **LOW**, thus the LED will be on when the button is not pressed and off when the button is pressed.
2. Why?
 - a. **A:** Because we are using a **PULLUP** resistor, thus driving the button default value to **HIGH**.
3. Why do we need a pulldown resistor in this setup?
 - a. **A:** So the value won't float
4. What would happen if we removed the resistor?
 - a. **A:** The value will float
5. How can we modify this setup to make the LED stay ON after one press and turn OFF after another?
 - a. **A:** This is where the next exercises come in, but make them think about it.

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. 2 - because it is a detailed variant - but any version is fine.

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



```
}  
Serial.println(buttonState);  
}
```

#1

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

#2

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

1.5 Internal PULLUP resistor

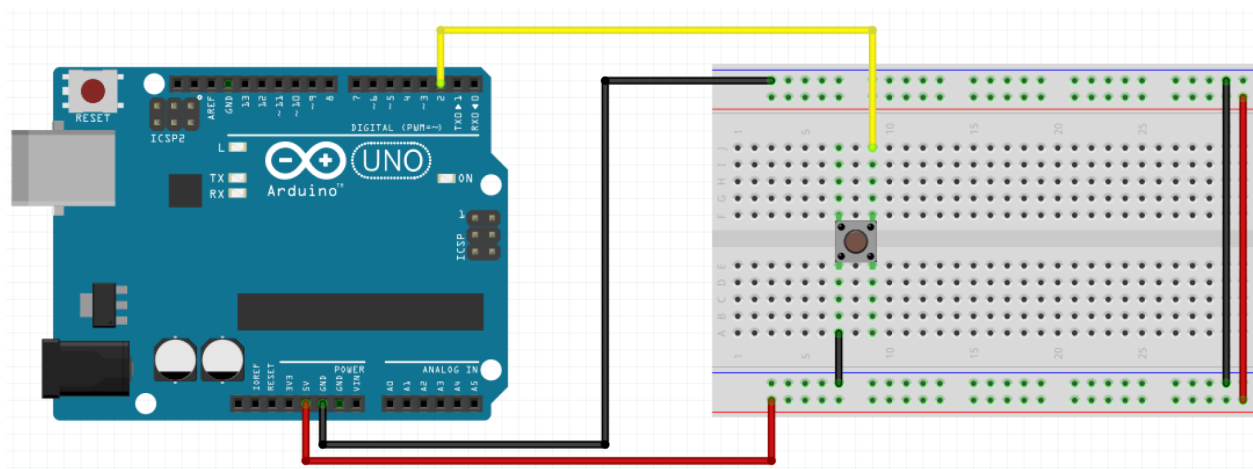
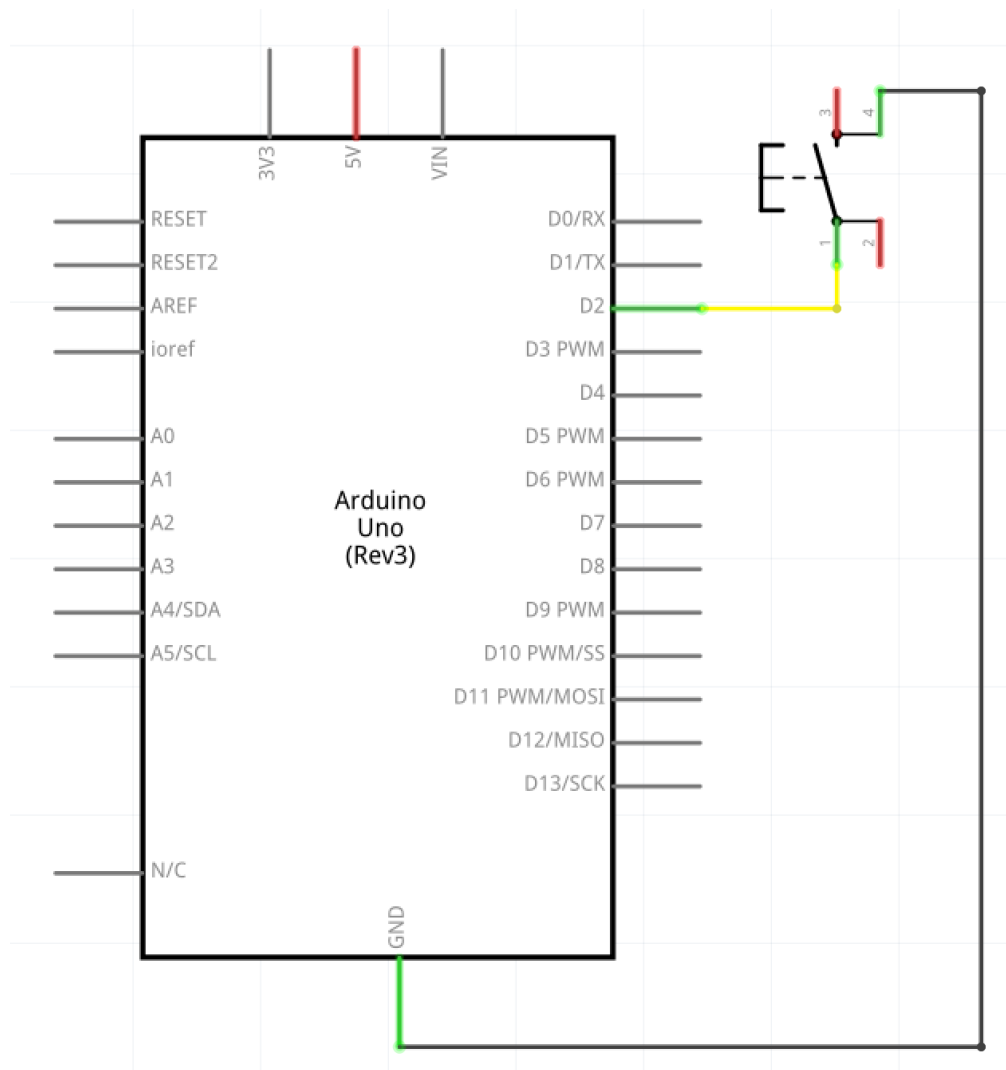
We've started with a **hardware pulldown resistor**, switched it to a **hardware pullup** and now we'll see a way to remove the resistor completely.

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 20k Ω and 50k Ω . On the Arduino Due, it is between 50k Ω and 150k Ω . 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, INPUT_PULLUP);
  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. **A:** no specific answer, I was just curious
2. What are the advantages?
 - a. **A:** no hardware
3. What are its drawbacks?
 - a. **A:** it makes the code a lot less portable, as you are not sure the new system supports INPUT_PULLUP.

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

2. Counting Button Presses, State Change, and Debouncing

2.1 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;
int buttonPushCounter = 0;

byte buttonState = LOW;
byte ledState = LOW;

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);

  if (buttonState == LOW) {
    buttonPushCounter++;
  }
  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. **A:** The led counter is considerably lower.
2. How does the counter grow?
A: The button counter grows faster.
3. Why?
A: Because it does many readings in a fraction of a second.
4. How can we fix this?
A: We need state change detection.

2.2 Lab exercise 2: 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 = 13;

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);

  if (buttonState != lastButtonState) {
    if (buttonState == LOW) {
      buttonPushCounter++;
    }
  }
}
```

```
    ledState = !ledState;
    digitalWrite(ledPin, ledState);
    Serial.println(buttonPushCounter);
  }
  delay(50);
}
lastButtonState = buttonState;
}
```

Questions:

1. What are the drawbacks of this method?

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

2.3 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 = 13;

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

byte reading = HIGH;
byte lastReading = HIGH;

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() {  
  reading = digitalRead(buttonPin);  
  
  if (reading != lastReading) {  
    lastDebounceTime = millis();  
  }  
  if ((millis() - lastDebounceTime) > debounceDelay) {  
  
    if (reading != buttonState) {  
      buttonState = reading;  
  
      if (buttonState == LOW) {  
        ledState = !ledState;  
      }  
    }  
  }  
  digitalWrite(ledPin, ledState);  
  lastReading = reading;  
}
```

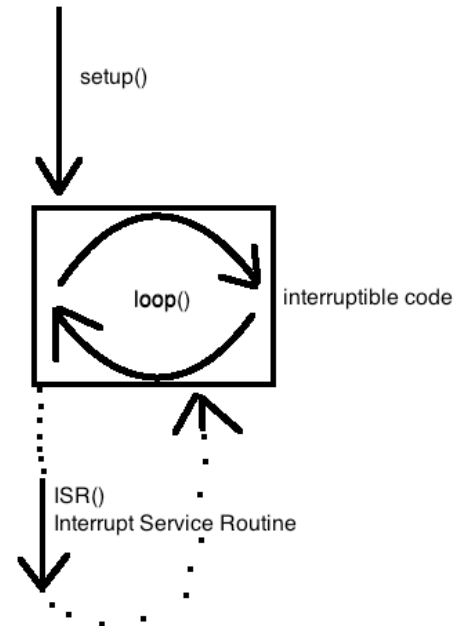
We finally learned how to turn the light on and off with a pushbutton.

3: Introduction to Software Interrupts

3.1 Introduction to Interrupts

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



3.2 Setting Up Interrupts in Arduino

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, RISING);
}

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

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

3.3 Implementing Software Interrupts

Set up a software interrupt to detect a button press and toggle an LED accordingly.

```
const int buttonPin = 2;
const int ledPin = 13;
volatile bool toggleLED = false;

void setup() {
  pinMode(buttonPin, INPUT);
  pinMode(ledPin, OUTPUT);
  attachInterrupt(digitalPinToInterrupt(buttonPin), toggleISR, RISING);
}

void loop() {
  if (toggleLED) {
    digitalWrite(ledPin, !digitalRead(ledPin)); // Toggle LED state
    toggleLED = false;
  }
}

void toggleISR() {
  toggleLED = true;
}
```

3.4 Debounce with interrupts

There are two common ways to implement debounce with interrupts.

3.4.1 Debounce in the loop (works, but kind of beats the purpose of the interrupt)

```
// Pin definitions
const int buttonPin = 2;
const int ledPin = 13;

// Flags to keep track of button press state and debounce
volatile bool possiblePress = false; // Indicates that a button press might have occurred
volatile bool shouldStartCounter = false; // Indicates the need to start debounce counter

// Debounce parameters
```

```
unsigned long lastDebounceTime = 0;    // Keeps track of the last time the button was pressed
unsigned int debounceDelay = 50;       // Debounce delay in milliseconds

// State variables
bool ledState = LOW;                   // Current state of the LED (ON/OFF)
bool buttonState = HIGH;               // The previous state of the button (HIGH when not pressed due
to INPUT_PULLUP)
bool buttonReading = HIGH;             // The current reading from the button pin

void setup() {
    pinMode(ledPin, OUTPUT);           // Initialize the LED pin as an output
    pinMode(buttonPin, INPUT_PULLUP);  // Initialize the button pin as an input with internal pull-up
resistor
    // Attach an interrupt to the button pin to detect FALLING edge (button press)
    attachInterrupt(digitalPinToInterrupt(buttonPin), handleInterrupt, FALLING);
}

void loop() {
    // Check if a possible button press was detected
    if (possiblePress) {
        // Start the debounce counter if needed
        if (shouldStartCounter) {
            lastDebounceTime = millis();
            shouldStartCounter = false;
        }

        // If the debounce period has elapsed
        if ((millis() - lastDebounceTime) > debounceDelay) {
            // Read the current state of the button
            buttonReading = digitalRead(buttonPin);

            // If the button state has changed (i.e., if it was not a false press)
            if (buttonReading != buttonState) {
                // Toggle the LED state if the button was truly pressed (read as LOW due to INPUT_PULLUP)
                if (buttonReading == LOW) {
                    ledState = !ledState;
                    digitalWrite(ledPin, ledState);
                }
            }
            // If the states match, it was a false alarm due to noise

            // Reset the possiblePress flag
            possiblePress = false;
        }
    }
}
```

```
}  
}  
}  
  
// Interrupt service routine (ISR) executed when the button is pressed  
void handleInterrupt() {  
    possiblePress = true;           // Indicate that a button press might have occurred  
    shouldStartCounter = true;     // Indicate the need to start the debounce counter  
}
```

Advantages of handling debounce in the main loop:

1. Flexibility: The main loop can be more easily adjusted for various conditions. For example, if you wanted to implement features like long presses or double clicks, it might be more straightforward outside of the ISR context.
2. ISR Efficiency: ISR remains extremely quick, ensuring minimal disruption to the main program.
3. The interrupt only sets flags indicating a potential button press (possiblePress and shouldStartCounter).
4. The loop() function then waits for the debounce delay duration, post-interrupt, to actually check the button's state with digitalRead(). This ensures the button remains pressed even after the debounce delay, thus offering a more reliable debouncing mechanism.
5. The delayed verification ensures that short noises or fluctuations on the button pin do not immediately trigger a state change of the LED. The button state is confirmed only after waiting out the entire debounce duration.

Disadvantages:

1. Latency: The response to the button press might not be as immediate since it depends on where in the main loop the code is when the button is pressed.
2. Main Loop Complexity: Introduces more logic into the main loop, which might make it less readable if the main loop is handling various tasks.
3. Kind of beats the purpose of using an interrupt in the first place, since you still do polling in the loop().

3.4.2 Debounce in the Interrupt Service Routine (with micros())

This code is optimized for scenarios where the button isn't rapidly pressed or held down for extended periods. Rapid presses might sometimes go unnoticed if they occur within the debounce delay. Moreover, if the button is held down and then released, the button during release can inadvertently trigger another FALLING edge. This unintended edge is mistakenly treated as a legitimate button press by the current setup, causing unexpected behavior like toggling the LED state again. These can be solved with different approaches, such as lockout period after press, a state machine implementation and many more.

```
// Pin definitions
const int buttonPin = 2;
const int ledPin = 13;

// Flag to indicate a valid button press event
volatile bool buttonPressed = false;

// Timestamp for when the last interrupt occurred, used for debouncing
volatile unsigned long lastInterruptTime = 0;

// Debounce time in milliseconds
const unsigned long debounceDelay = 200;

// Variable to track the state of the LED
bool ledState = LOW;

void setup() {
  pinMode(buttonPin, INPUT_PULLUP); // Initialize the button pin as an input with internal pull-up resistor
  pinMode(ledPin, OUTPUT); // Initialize the LED pin as an output
  digitalWrite(ledPin, ledState); // Set the initial state of the LED

  // Attach an interrupt to the button pin to detect FALLING edge (button press)
  attachInterrupt(digitalPinToInterrupt(buttonPin), handleInterrupt, FALLING);
}

void loop() {
  // Check if a valid button press was detected
  if (buttonPressed) {
    ledState = !ledState; // Toggle LED state
    digitalWrite(ledPin, ledState);
  }
}
```

```
    buttonPressed = false; // Reset the button press flag after handling it
  }
}

// Interrupt service routine (ISR) executed when the button is pressed
void handleInterrupt() {
    static unsigned long interruptTime = 0; // Timestamp for the current interrupt, retains its value
between ISR calls
    interruptTime = micros(); // Capture the current time in microseconds

    // Debounce logic: If interrupts come in faster succession than the debounce delay, they're ignored
    if (interruptTime - lastInterruptTime > debounceDelay * 1000) { // Convert debounceDelay to
microseconds for comparison
        buttonPressed = true;
    }
    // Update the last interrupt timestamp
    lastInterruptTime = interruptTime;
}
```

Explanations:

Why use micros() instead of millis() in ISR?

In the context of the Arduino, certain functions, like millis(), rely on interrupts to work and won't function properly within an ISR. The micros() function, however, can be safely used at the beginning of an ISR but will start to behave erratically if the ISR runs for an extended time. For brief debounce delays typically encountered with button presses, it's reasonable.

Why use static for interruptTime?

Declaring interruptTime as static allows it to retain its value between invocations of the ISR. This can be a minor optimization in terms of stack space since the variable isn't re-declared each time the ISR runs. However, in this particular example, it's more of an optimization rather than a necessity and we could've just as well declared it as a global volatile variable.

Advantages of handling debounce within the ISR:

1. Immediate Response: There's a more immediate response to a button press since the action is taken directly in the interrupt handler and doesn't rely on being polled in the main loop.
2. Simpler Main Loop: The main loop stays cleaner and more focused on other tasks.

Disadvantages:

1. **ISR Length:** Even though this isn't a lengthy process, the interrupt handler becomes longer, which could be problematic if there are other time-sensitive operations happening in the code. Remember, while in an ISR, other interrupts are typically disabled.
2. **Inflexibility:** If you want to change the action taken after a button press or introduce more complex button interactions (like detecting long presses), the ISR approach can become cluttered.
3. **Limited Functions:** Some Arduino functions don't work well (or at all) inside ISRs, which can limit your options for debouncing methods.
4. As soon as the interrupt is triggered by a falling edge (button press), it checks the time since the last registered press. If this difference exceeds the debounce delay, it immediately acknowledges a valid button press and toggles the button state. A drawback: This immediate approach does not wait to confirm if the button remains pressed after the debounce period. This can result in incorrect behavior when you keep the button pressed long. Specifically, any noise or unintended fluctuation (e.g., during button release) after the debounce delay can be falsely interpreted as a new legitimate press.