ESG Social #2: Arduinio & AVR

Josue Hernandez: 19 Sep 2023 @ HackerGarage

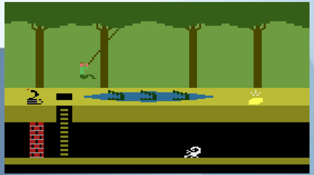
Overview

- *Developing for embedded systems:* ***a crash course***
- *What is Arduino?*
- *Programming Arduino*
- *Introduction to the Arduino IDE*
- *Connecting LEDs: what you need to know*
- *Interactivity: using* ***buttons*** *in your project*
- *How to talk to embedded devices: the serial interface*
- *Extending Arduinos via add-on boards*
- *How to proceed from here!*

Developing for Embedded Systems

How does developing for embedded systems differ from other disciplines of software engineering / development?

- We are not just creating software but also designing hardware.
    - Requires *some knowledge* of **electronics** and **telecommunications**.
    - We need to understand how hardware components interact (**bus**/**com** protocols, voltage and current, physical effects such as noise, clock synchronisation, etc.)
- Our target SoC (CPU, memory, clock speed, etc.) often has limited capabilities that we need to accept and work around.
    - Atari 2600 only had **128 bytes** (!) of RAM (which included the run-time stack)
    - Watch https://www.youtube.com/watch?v=MBT1OK6VAIU on creating Pitfall
- Debugging can be way more challenging …



Developing for Embedded Systems (cont'd)

How does developing for embedded systems differ from other disciplines of software engineering / development?

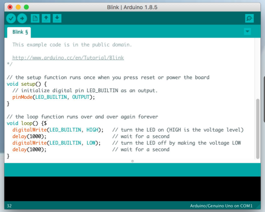
- Programming languages we use are traditionally more low-level:
  - C/C++ the *de facto* ones; shift towards Rust & Python in recent years
  - We also find pure assembly / machine language and Forth (stack machine)
  - With more powerful MCUs, possible now to port higher-level (interpreted) language, such as *Circuit*Python
  - Arduino uses the **Wiring** platform/API/language which is based on C/C++
- Programs have to be cross-compiled on a host (development environment) and then uploaded to the target device.
  - Requires us to install a (compilation) **tool-chain** for the target architecture

Arduino Workflow (in pictures)

avr tool-chain

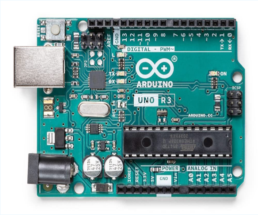
USB
boot-loader

USB → UART
Bridge

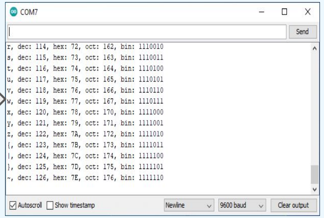


SOURCE CODE
(WIRING / .INO FILES)

CROSS-COMPILED
BINARY EXECUTABLE



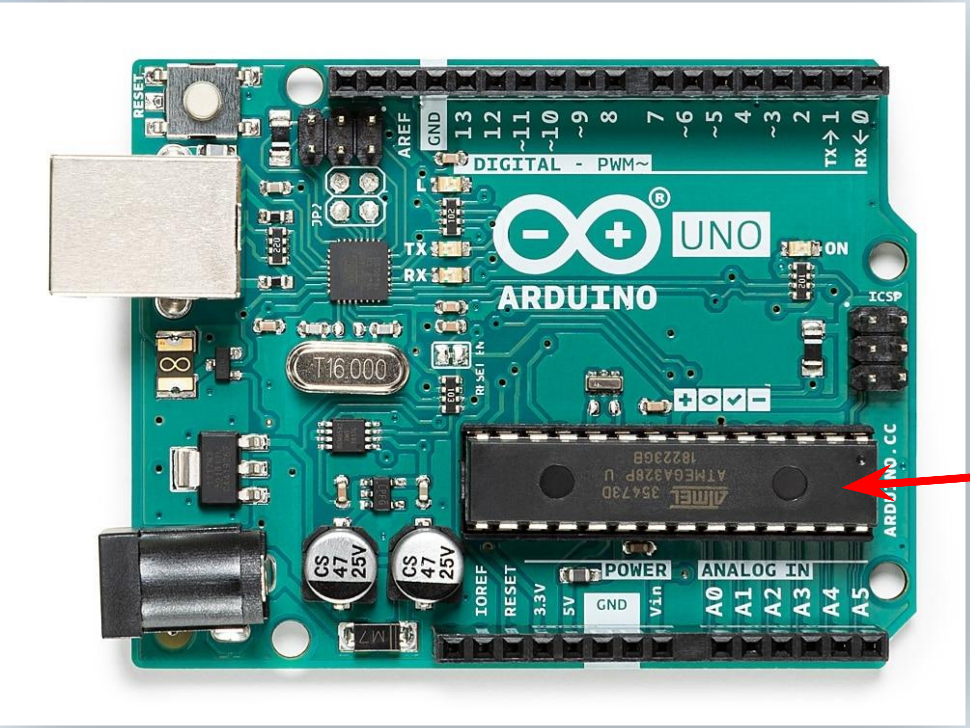
UPLOADED PROGRAM
ONTO ATMEGA328



SERIAL MONITOR
(VIA UART)

CARRIED OUT
ON THE HOST

What is Arduino?



The heart of an Arduino UNO:

ATmega328P
MCU

What is Arduino?

Microcontroller	**ATmega328P**
Digital I/O pins	14
Analog Input pins	6
PWM pins	6
Communication	**UART/USART**, **I2C**, **SPI**
I/O Voltage	5V (**TTL**)
Clock Speed Main Processor	16 MHz
Clock Speed USB Serial Processor	16 MHz
Memory: ATmega328P	2KB **SRAM**, 32KB **Flash**, 1KB **EEPROM**

What is Arduino? (more on this in the 2nd talk)

Microcontroller	**ATmega328P**
Digital I/O pins	14
Analog Input pins	6
PWM pins	6
Communication	**UART/USART**, **I2C**, **SPI**
I/O Voltage	5V (**TTL**)
Clock Speed Main Processor	16 MHz
Clock Speed USB Serial Processor	16 MHz
Memory: ATmega328P	2KB **SRAM**, 32KB **Flash**, 1KB **EEPROM**

Programming Arduino

Bare-Metal Programming

- Application programs usually do not exist in a **vacuum**
- The OS provides a sophisticated environment in which program run
- Includes **system calls** for file I/O, ICP, memory management, etc.
- And an elaborate set of libraries for **video**, **sound**, **GUI**s, etc.
- When programming **MCU**s / embedded devices:
  - the OS environment is **minimal** to **no existing** …
  - often limited **MMU**, no paging, simplified protection mechanism, etc.
  - no full **glibc** — consider yourself lucky if a compliant **printf**(...) is available
  - even writing a “Hello World!” program can be rather complex …
  - bare-metal programs usually do **not** **finish** or **exit** …

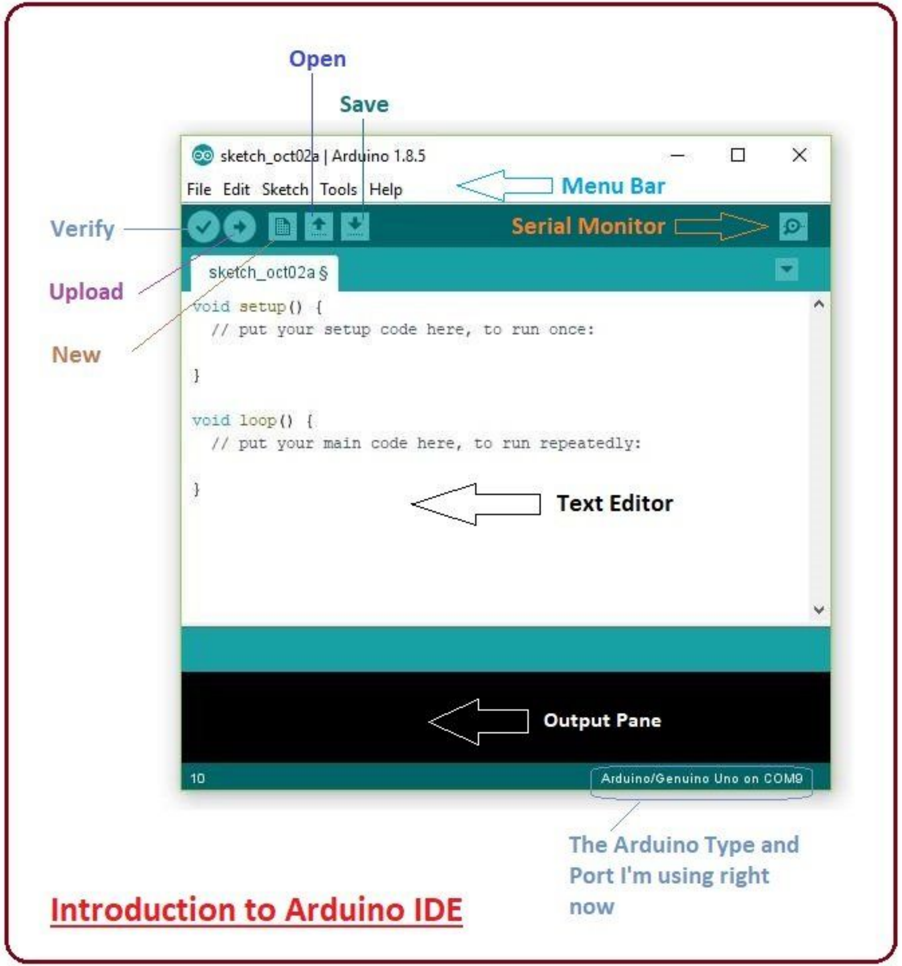
Bare-Metal Programming

- AVR Compiler (avr-gcc)
- AVR Tooling (avr-binutils)
- AVR Libraries (avr-libc)
- AVR Loader (avrdude)

```c
#include <avr/io.h>
#include <util/delay.h>

#define MS_DELAY 3000

int main (void) {
    /*Set to one the fifth bit of DDRB to one
    **Set digital pin 13 to output mode */
    DDRB |= _BV(DDB5);

    while(1) {
        /*Set to one the fifth bit of PORTB to one
        **Set to HIGH the pin 13 */
        PORTB |= _BV(PORTB5);

        /*Wait 3000 ms */
        _delay_ms(MS_DELAY);

        /*Set to zero the fifth bit of PORTB
        **Set to LOW the pin 13 */
        PORTB &= ~_BV(PORTB5);

        /*Wait 3000 ms */
        _delay_ms(MS_DELAY);
    }
}
```

Arduino IDE Basics

- Arduino applications are called **sketch**.
- The  button compiles the sketch for the target
- The  button uploads the (compiled) sketch to the device via USB
- The sketch starts executing on the target *immediately* after upload.



Arduino Code

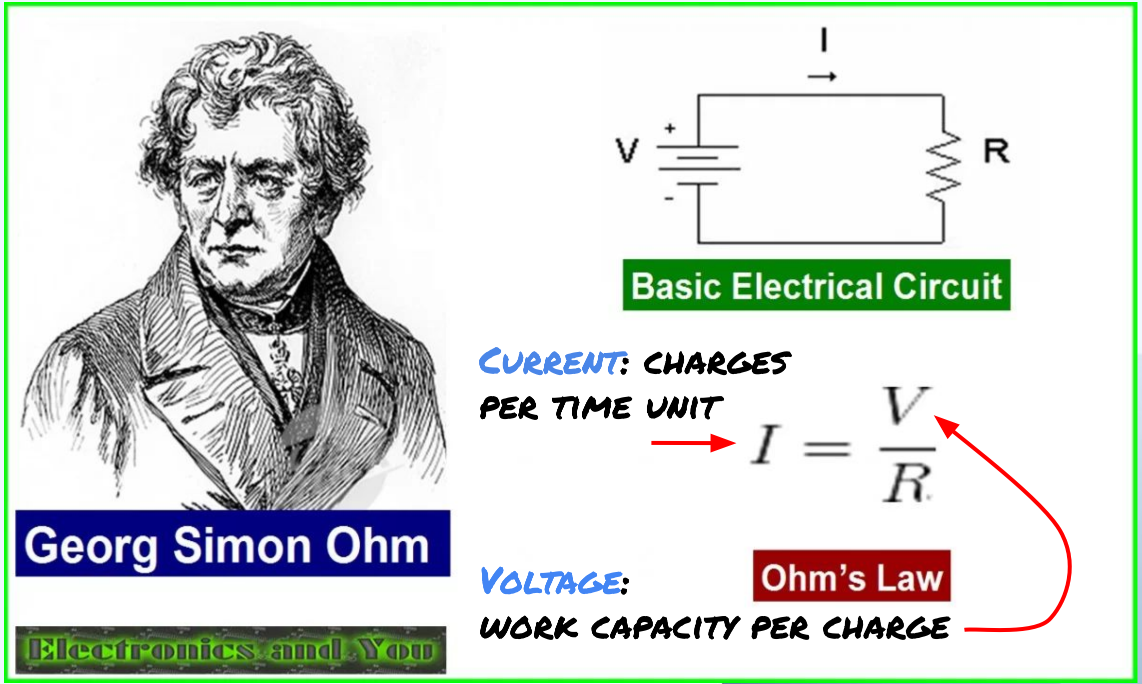
- Arduino IDE
  - Arduino Libraries
  - Arduino Compiler
  - Arduino Loader

```
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

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

Connecting LEDs

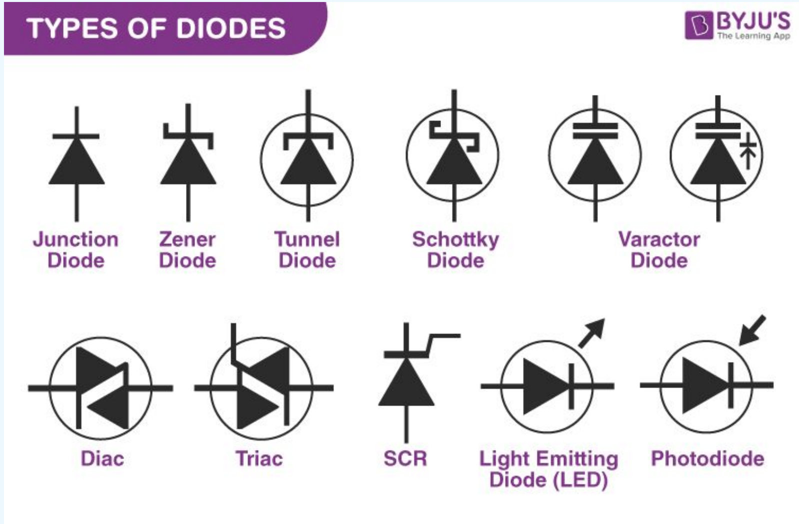
Recap Ohm’s law: voltage, current and resistor

- **Current** (I) increases *proportionally* to the applied **voltage** (V)
- The larger the resistor, the less current can flow.
- Wires typically have a resistor close to 0Ω.
- **Not all** consumers behave according to Ohm’s law.



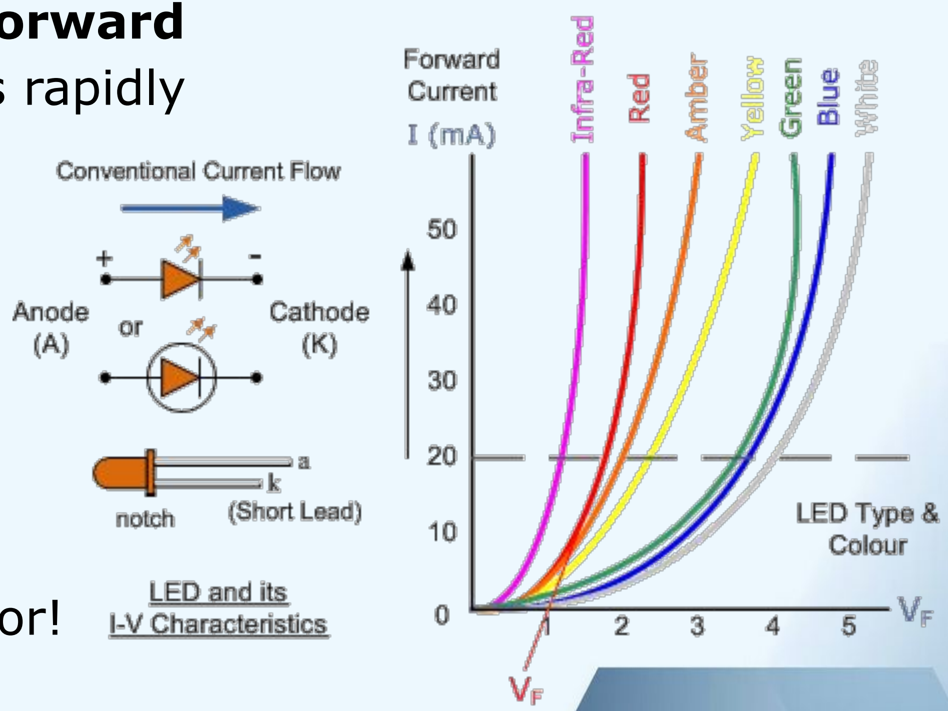
LEDs: What you need to know!

- LEDs are members of a larger class of so-called *semi-conductor* components: diodes
- Diodes are non-linear components: they **block** current flow up to a given voltage (the forward voltage) And then suddenly become *very conductive*.
- They do not obey ohm's law (!)
- Once a LED lights up, it effectively creates a short-circuit …



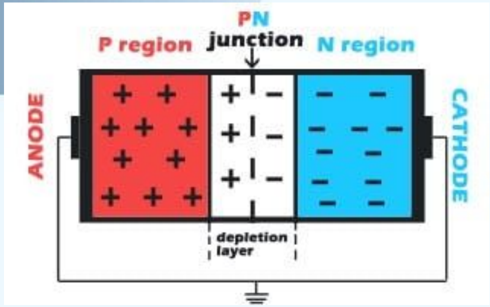
LEDs: What you need to know! (characteristics)

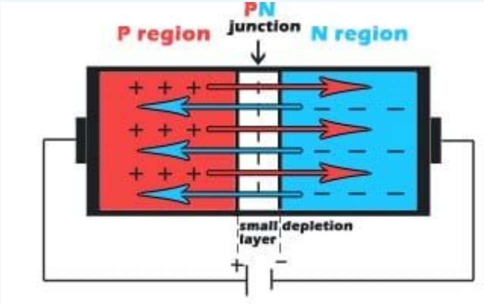
- Once we reach the LED’s **forward voltage**, current increases rapidly
- The forward voltage is *typically* around 1.8 V but might vary depending on size, color, etc.
- The operating current of a typical LED should not exceed 20 mA.
- We hence **require** a resistor!

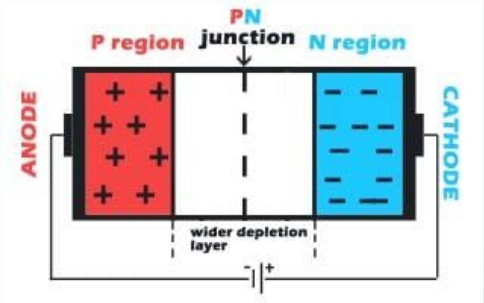


LEDs: What you need to know! (inner workings)

See: https://911electronic.com/semiconductor-diode/

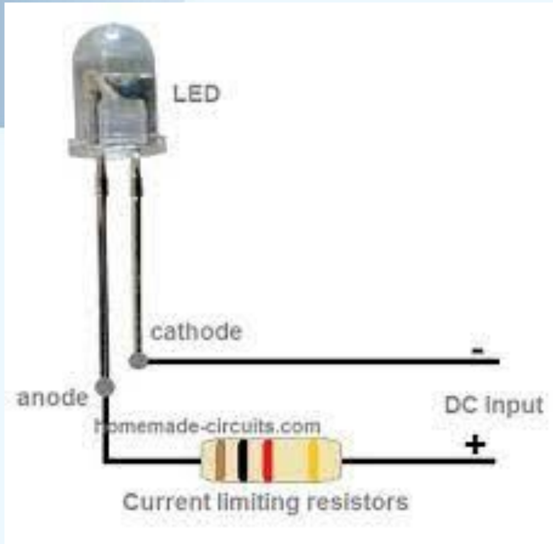


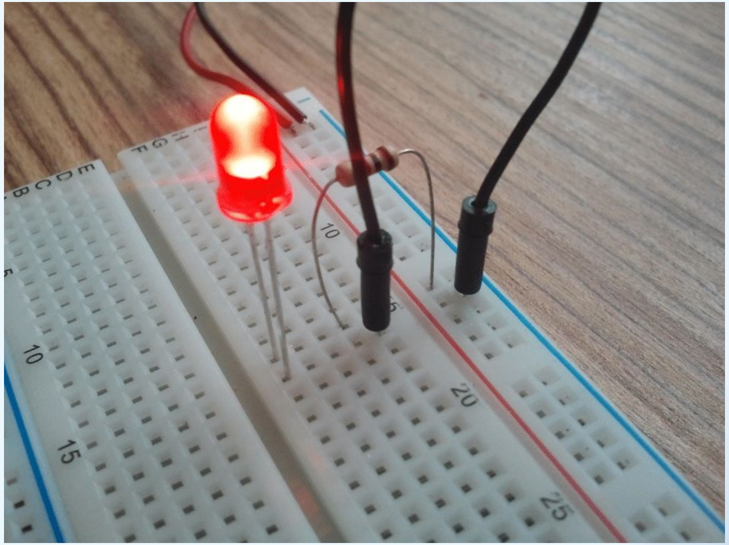




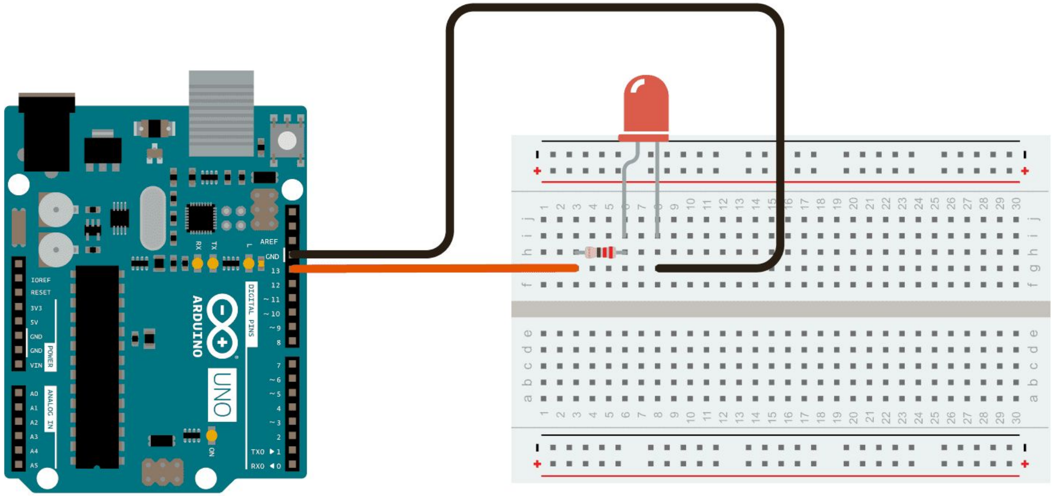
LEDs: What you need to know! (safely connecting)

See: https://911electronic.com/semiconductor-diode/



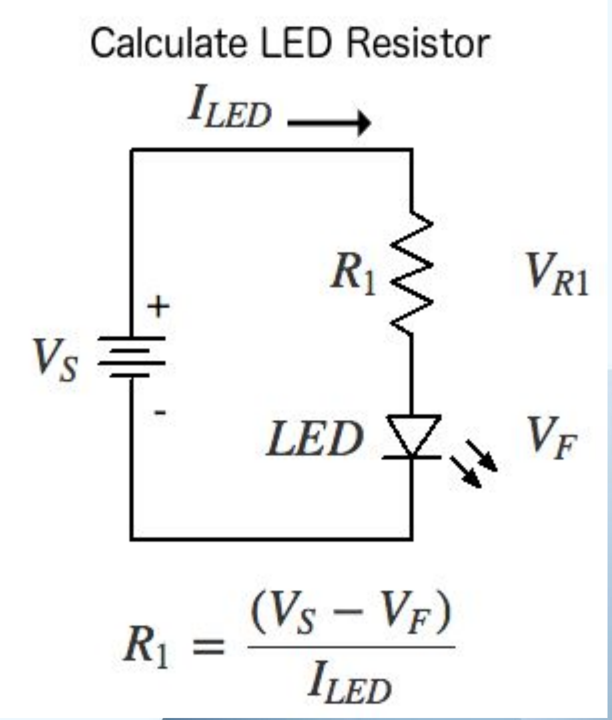


A simple example: blinking LED



LEDs: What you need to know! (calculate resistor)

- **LEDs** are usually characterised in terms of a voltage drop (voltage that the LED **requires** / **consumes** in order to close the depletion gap)
- The voltage at the resistor is hence: V_S - V_F (source voltage - voltage drop)
- We want to limit that voltage to the admissible current that may flow through the LED (I_{LED})
- Example: LED has V_F = 1.8 V and I_{LED} = 10 mA.
- Assuming 5 V driving current, we have: R_1 = (5V - 1.8V) / 10mA = 3.2V / 0.01A = 320 Ω



A simple example: blinking LED

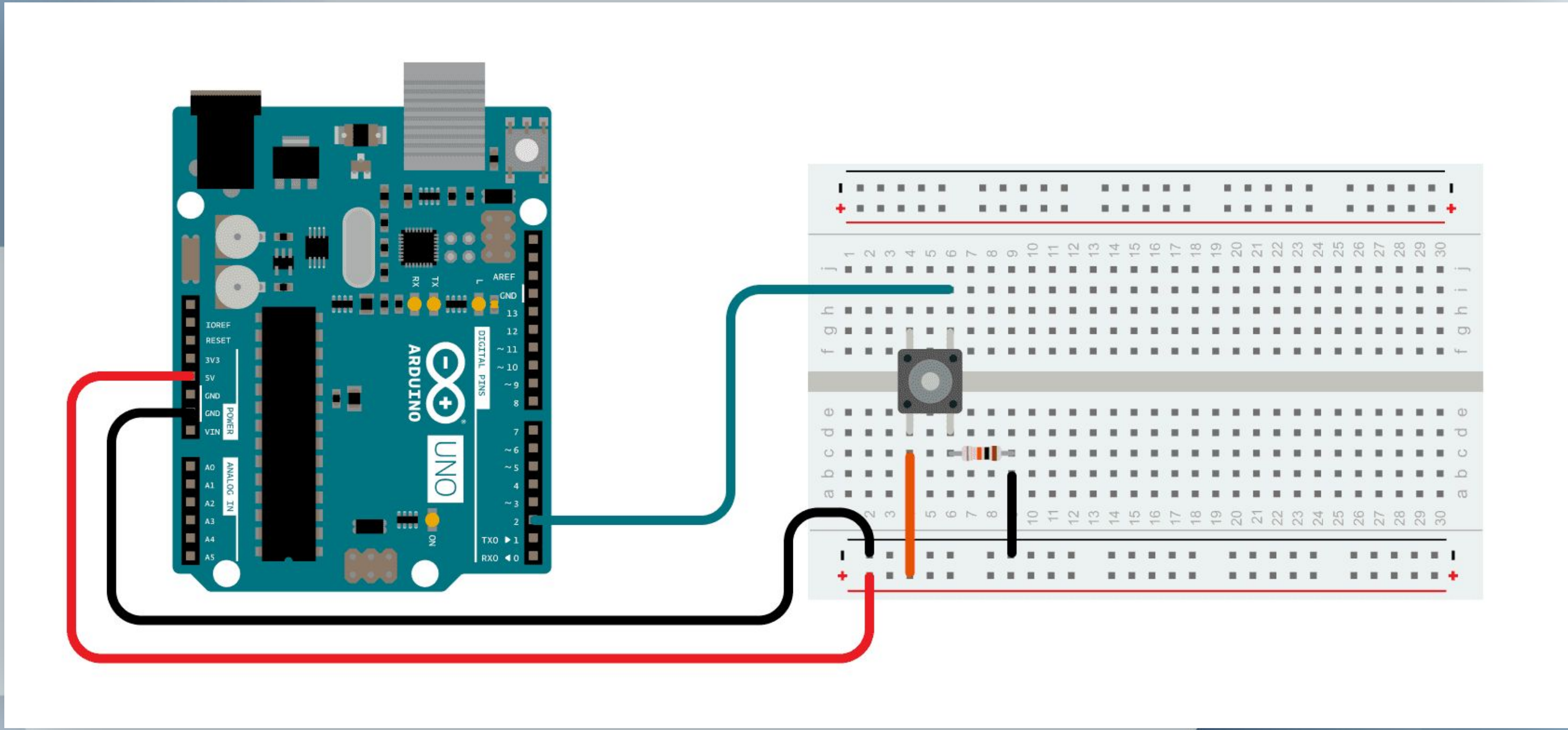
- Setup the let to OUTPUT to be able to write to it
- During every loop execution turn on the led Writing HIGH to the pin
- Wait for the timeout having the led on
- Turn down the led
- Wait for the time out to run a new loop execution

```
const int PIN_LED = LED_BUILTIN;
const int TIMEOUT = 500;

void setup() {
  // initialize digital pin 13 as an output.
  pinMode(PIN_LED, OUTPUT);
}

void loop() {
  // turn the LED on (HIGH is the voltage level)
  digitalWrite(PIN_LED, HIGH);
  // wait for 500 milliseconds
  delay(TIMEOUT);
  // turn the LED off by making the voltage LOW
  digitalWrite(PIN_LED, LOW);
  // wait for 500 milliseconds
  delay(TIMEOUT);
}
```

Interactivity: using buttons in your project

Types of buttons – a brief overview

Using Button



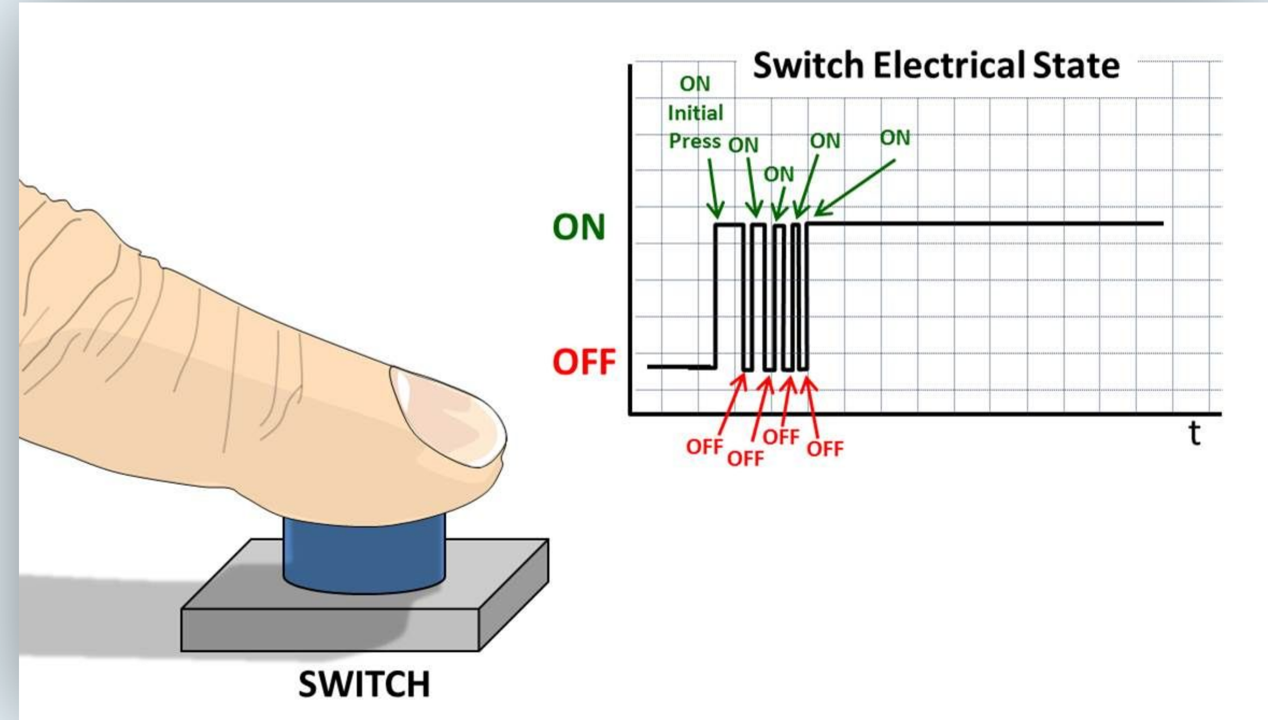
Using Button

- This is a simple way to check for the button status
- It read the pin that is connected to the push button every loop cycle.

```
// the number of the pushbutton pin
const int BUTTON_PIN = 2;

// the previous state from the input pin
int lastState = LOW;
// the current reading from the input pin
int currentState;

void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  // initialize the pushbutton pin as an pull-up input
  // the pull-up input pin will be HIGH when the switch
  // is open and LOW when the switch is closed.
  pinMode(BUTTON_PIN, INPUT_PULLUP);
}

void loop() {
  // read the state of the switch/button:
  currentState = digitalRead(BUTTON_PIN);

  if(lastState == HIGH && currentState == LOW)
    Serial.println("The button is pressed");
  else if(lastState == LOW && currentState == HIGH)
    Serial.println("The button is released");

  // save the the last state
  lastState = currentState;
}
```

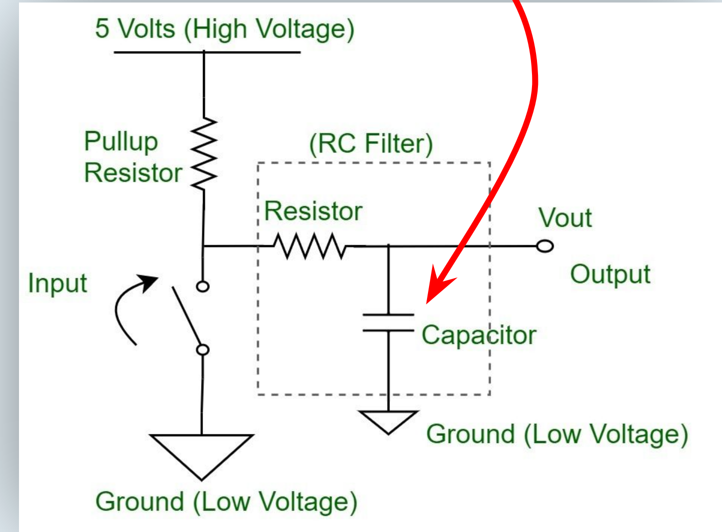
Mechanical implications: “instability during push”

- Buttons usually do **not** switch instantly but “**bounce**” between open / close for a short period of time (several ms)
- This can cause un-expected effects in software & control.



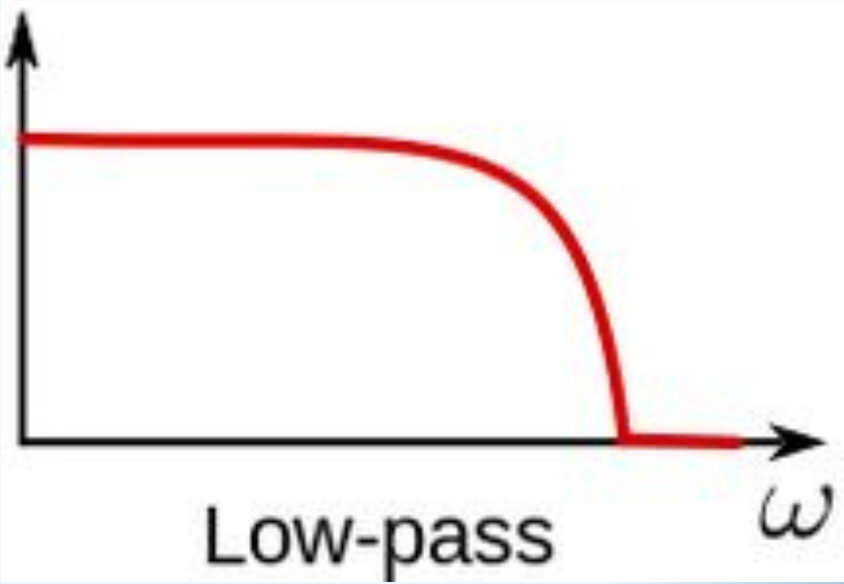
Debouncing Buttons

- Can be done either in hardware or in software.
  - Hardware solutions commonly use a **capacitor**.
  - Software solutions have to rely on a HW/SW **timer**.
- Resistor-Capacitor (RC) circuit acts as a **low-pass** filter, suppressing higher frequency signals.
- See "Further Reading" slide for useful online material on HW debouncing.
- Since most of us are SW engineers, let's try and do it in software …



Low-pass RC filter

- The **low-pass** filter attenuates the signal with frequencies higher than the cutoff frequency
- Fast oscillation i.e. **instabilities** of a button inherently have a high frequency
- We want to filter these out (!)



Debouncing Buttons in Software

- This method records the time of the last observed state change of the button
- The state change is only committed if it **persists** for a given amount of time
- Quick **oscillations** of the button state are thus ignored *on purpose*
- Relies on availability of a **timer** API / function

```
const int BUTTON_PIN = 7;
const int DEBOUNCE_DELAY = 50;

int lastSteadyState = LOW;
int lastFlickerableState = LOW;
int currentState;

unsigned long lastDebounceTime = 0;

void setup() {
  Serial.begin(9600);
  pinMode(BUTTON_PIN, INPUT_PULLUP);
}

void loop() {
  currentState = digitalRead(BUTTON_PIN);

  // Check for the state during this loop
  if (currentState != lastFlickerableState) {
    lastDebounceTime = millis();
    lastFlickerableState = currentState;
  }

  // Check for the delay
  if ((millis() - lastDebounceTime) > DEBOUNCE_DELAY) {
    if (lastSteadyState == HIGH && currentState == LOW)
      Serial.println("The button is pressed");
    else if (lastSteadyState == LOW && currentState == HIGH)
      Serial.println("The button is released");

    lastSteadyState = currentState;
  }
}
```

Debouncing Buttons in Software

- Arduino has a library that do practically the same but encapsulate the behaviour in a class **ezButton.**
- Can be instantiated for multiple buttons
- avoids a mess of variables
- **object-oriented** principles are readily available in C/C++ and Wiring

```
#include <ezButton.h>

// create ezButton object that attach to pin 7;
ezButton button(7);

void setup() {
  Serial.begin(9600);
  // set debounce time to 50 milliseconds
  button.setDebounceTime(50);
}

void loop() {
  // This loop is basically checking the
  // state of the button and the delay
  // to update the variables
  button.loop();

  if(button.isPressed())
    Serial.println("The button is pressed");

  if(button.isReleased())
    Serial.println("The button is released");
}
```

The Serial Interface

- All Arduino Boards provide at least **one** serial interface (UART)
- Directly exposed via USB when you plug in your Arduino
- Via some trickery this is also used for **uploading** sketches by the Arduino IDE and boot-loader code
- We can use the serial interface to output messages while the sketch executes on the target
- To see&read those messages, we have to connect a **serial terminal** on the host to the respective USB interface
- The Arduino IDE already includes such a terminal for our convenience …

The Serial Interface (code example)

- Setup the bits per second (9600)
- For reading check if there is data available first
- For writing pass what we want to write

```
int counter = 0;
const int MAX_VALUE = 10;
int byte_read = 0;

void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}

void loop() {

  // Reading from serial
  if (Serial.available()) {
    byte_read = Serial.read();
    Serial.print("Received = ");
    Serial.println(byte_read);
    return;
  }

  // Writing to serial
  Serial.print("Counter = ");
  Serial.println(counter);
  counter = ++counter % MAX_VALUE;
  delay(1000);
}
```

How to proceed from here …

- Prototyping complex solutions using Arduino
- Go deeper into the AVR libc
- Design custom PCBs using ATmega328P

Conclusion

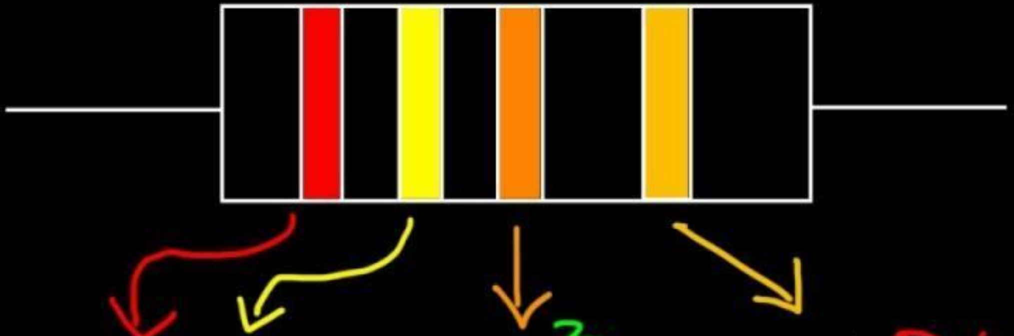
- Arduino is easy to understand
- There is an extensive documentation from official sources and hobbyists
- Arduino have an strong community (!)

Further Reading & Study

- Debouncing Buttons in Hardware: https://www.electronics-tutorials.ws/filter/filter_2.html
- Arduino Get Started https://arduinogetstarted.com/arduino-tutorials
- Arduino Examples https://docs.arduino.cc/built-in-examples
- Understanding Arduino Hardware Design https://www.allaboutcircuits.com/technical-articles/understanding-arduino-uno-hardware-design/

Appendix

Resistor Color Code



$$24 \times 10^3 \pm 5\%$$

$$24\,K\Omega \pm 1.2\,K\Omega$$

Color	Number	Multiplier	Tolerance
Black	0	1	
Brown	1	10^1	
Red	2	10^2	
Orange	3	10^3	
Yellow	4	10^4	
Green	5	10^5	
Blue	6	10^6	
Violet	7	10^7	
Gray	8	10^8	
White	9	10^9	
Gold		10^-1	5%
Silver		10^-2	10%
No Color			20%

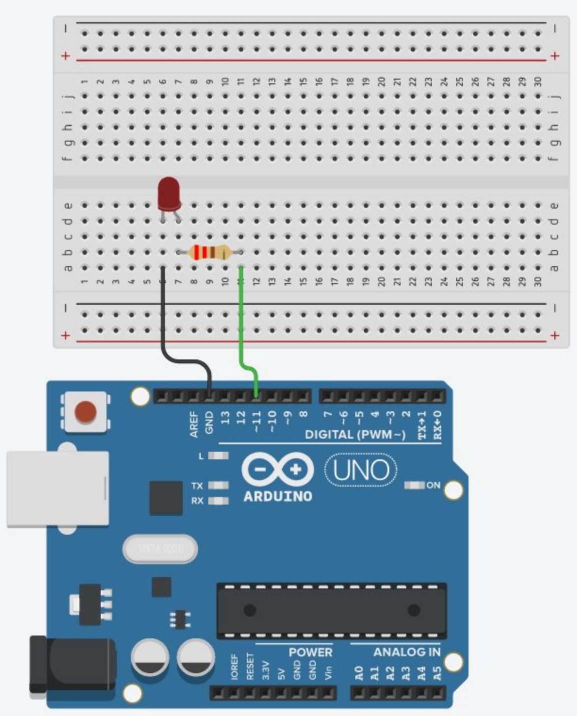
LEDs: What you need to know! (typical resistors at 5 V)

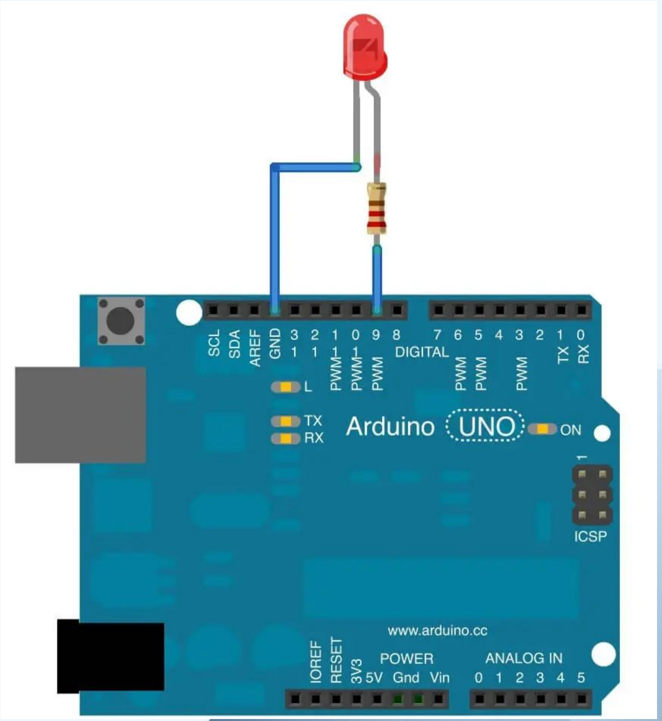
- Typical current-limiting resistors used for a single LED at 5 V:
- 330 Ω   470 Ω

Resisator values are **color-coded** (!)

Arduino design with a single LED

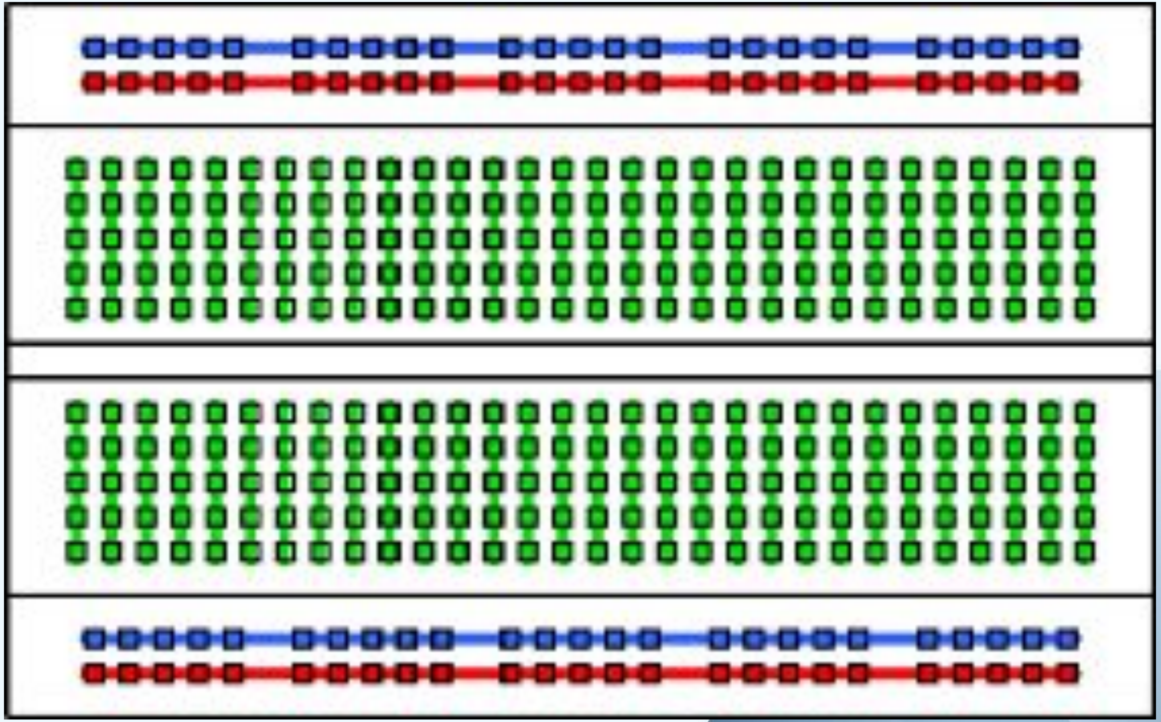
- Connect LED Kathode (short end) to **GND** (ground)
- Connect LED Anode (long end) to (digital) **I/O PIN**
- Understand how implicit connections on a **breadboard** are laid out!





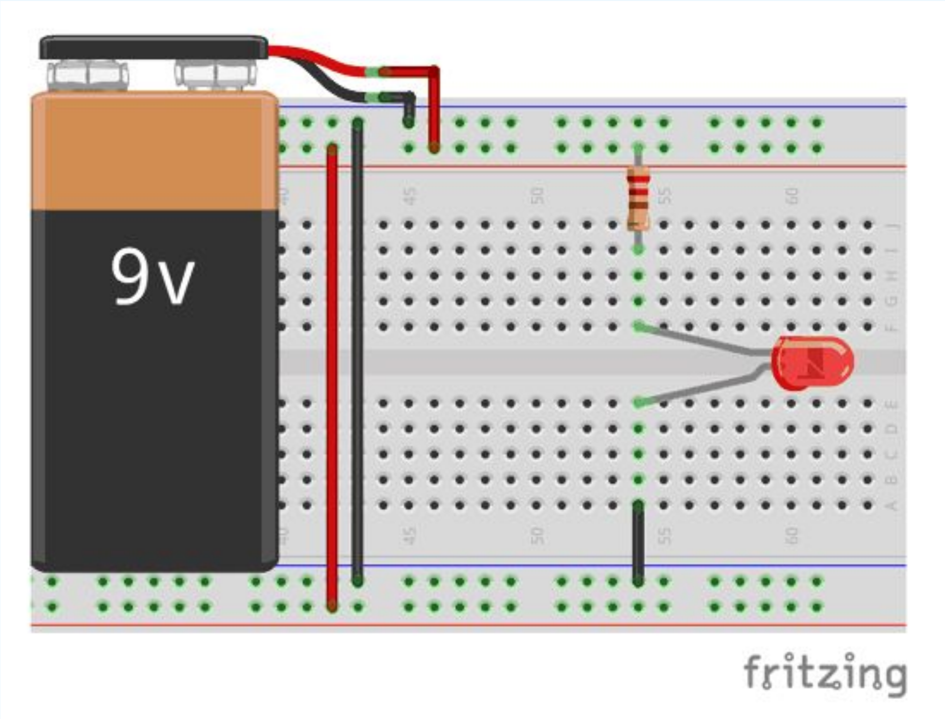
Understanding how breadboards work!

- **blue** and **red** holes are used for **power rails**: they are connected *horizontally*.
- The rest is structured into **two grids** whose holes are respectively connected *vertically*.



Understanding how breadboards work! (example)

- First connect V_{cc} and **GND** to the respective power rails.
- The use the two middle grids To place your components.
- Avoid a **short-circuit** (!)
- **Double** and **triple** check before you connect your Arduino to **power** and your **laptop** / **PC**.
- A few guidelines to protect your equipment are on the next slide!



Avoiding Damage to your Arduino and Laptop/PC

- Before connecting wires and components, always **unplug** the Arduino from your computer (**USB**) and the power supply.
- Before you plug the Arduino into your computer: **double** and **triple** check the connections on your breadboard.
- Use color-coded wires: e.g., red for **Vcc** and black for **GND**.
- Using Genuine Arduino boards and chips can reduce the risk of damage, i.e., to your computer.
- Ensure that your I/O PINs are properly **configured** as inputs and outputs, as required by your design in the setup() function.
- Consider **load** and **current** **ratings** before connecting any device.
- If something smokes or smells burned, immediately unplug.
- **Disclaimer**: Sadly, I cannot take responsibility if your laptop or Arduino gets damaged.