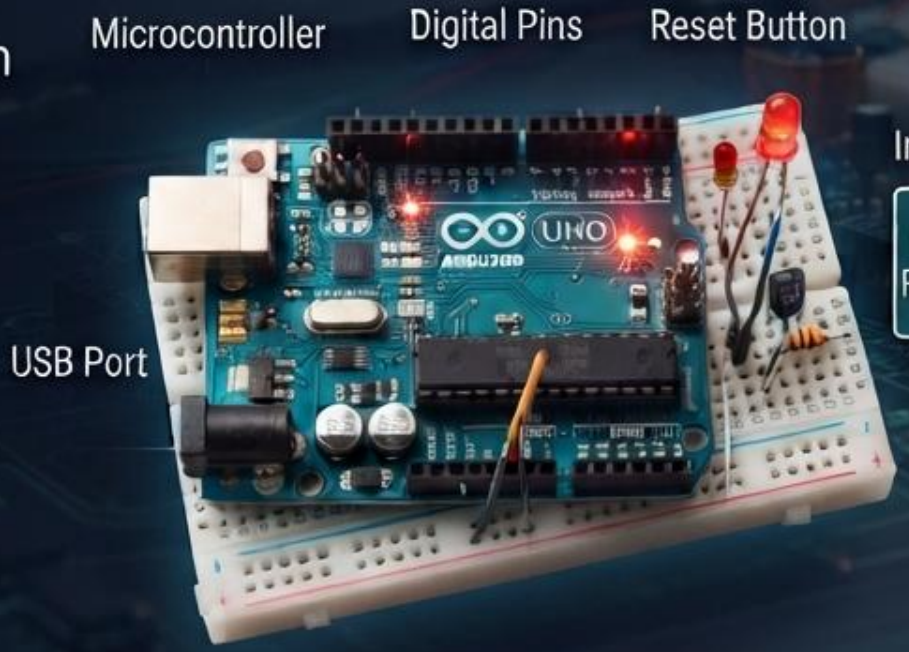
EMBEDDED SYSTEMS: ARDUINO FUNDAMENTALS

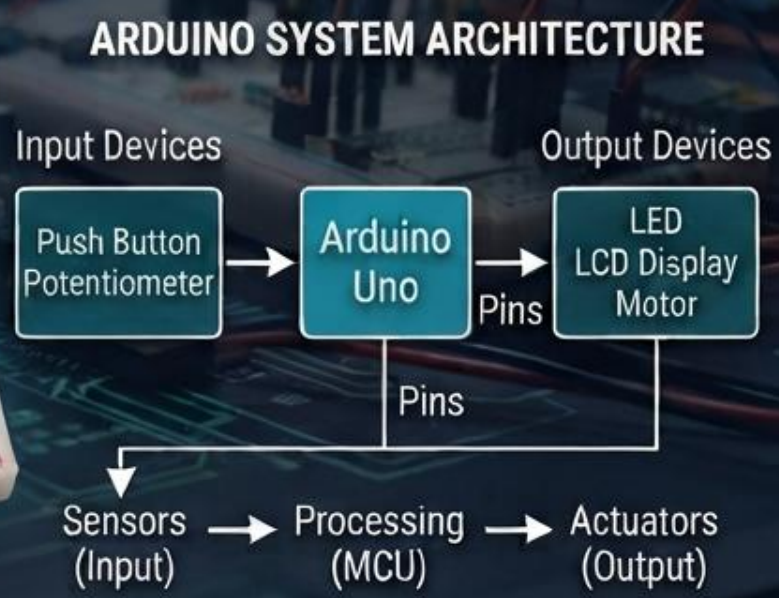
ARDUINO OVERVIEW

- Open-Source Electronics Platform
- Microcontroller (ATmega328P)
- Versatile I/O (Digital & Analog)
- Easy Programming (C++ based IDE)
- Engineering & Prototyping

KEY COMPONENTS



SYSTEM ARCHITECTURE



Nabil Karami, PhD



Week 1

Lecture 1 / LO1

Introduction to Embedded Systems



What is an Embedded system

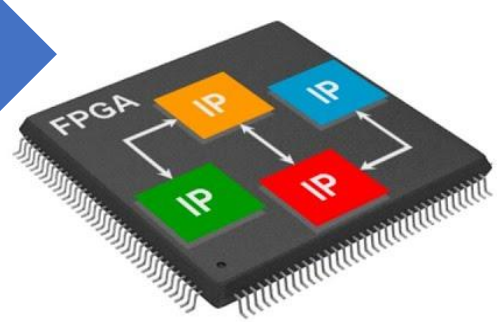
An embedded system is an electronic programmed and operating system that performs a dedicated task within a larger electrical system.

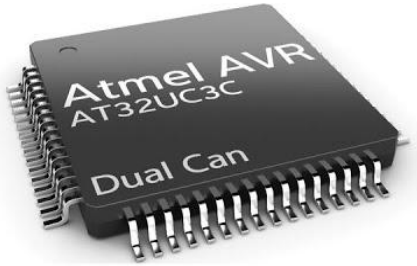
It acts as a central component and is specifically designed for limited functionality.

These systems are different from the general-purpose computers (desktops and laptops) – the general-purpose computer can handle a wide range of processing tasks unlike embedded systems.

- **Embedded System:**
  - **"Embedded" means hidden inside so one can't see it.**
  - **"System" means multiple components interfaced Together for a common purpose.**





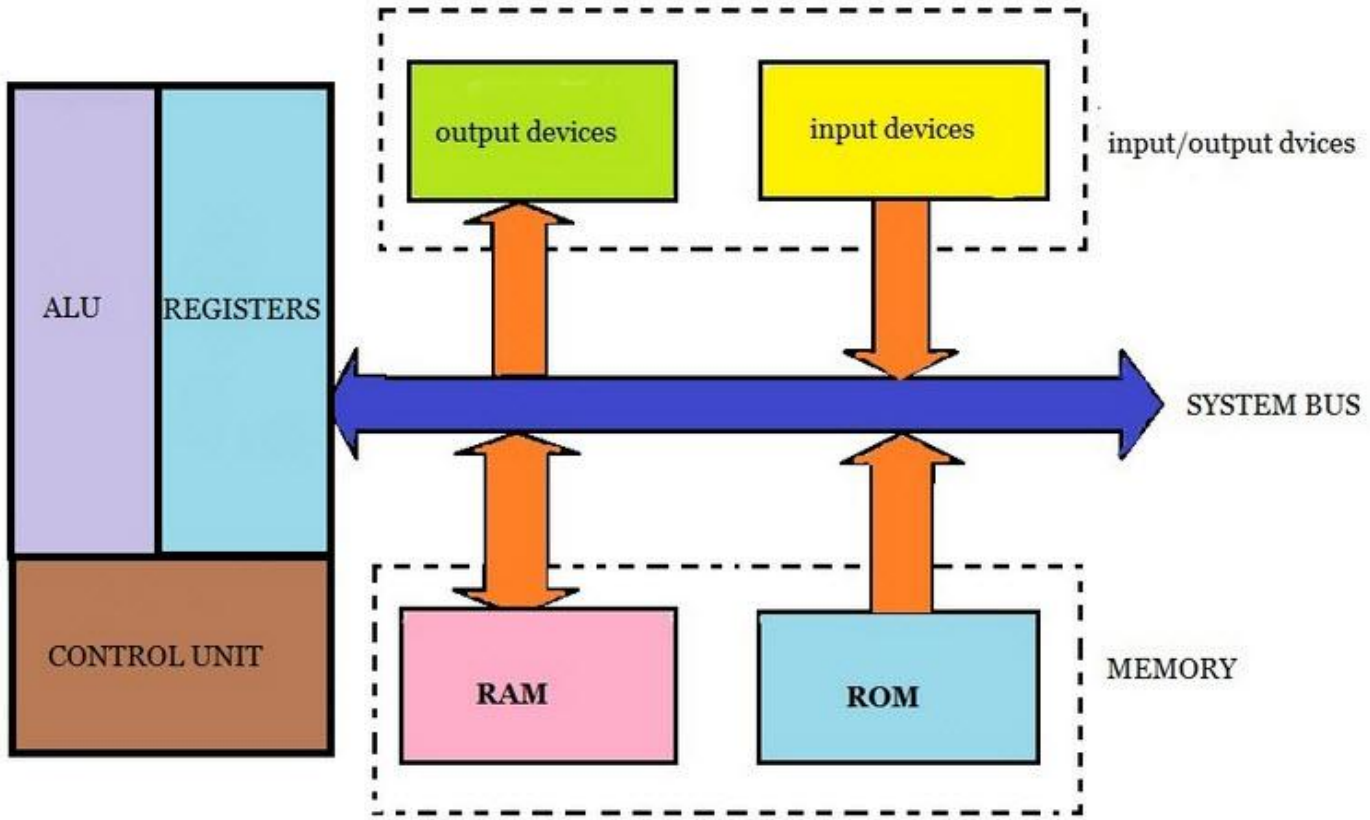






What does an Embedded System consist of?

- We have defined an embedded system as a central component that performs a computational task within a system. This component could be a simple microcontroller, a microprocessor, a digital signal processor or even an FPGA (field-programmable gate array).



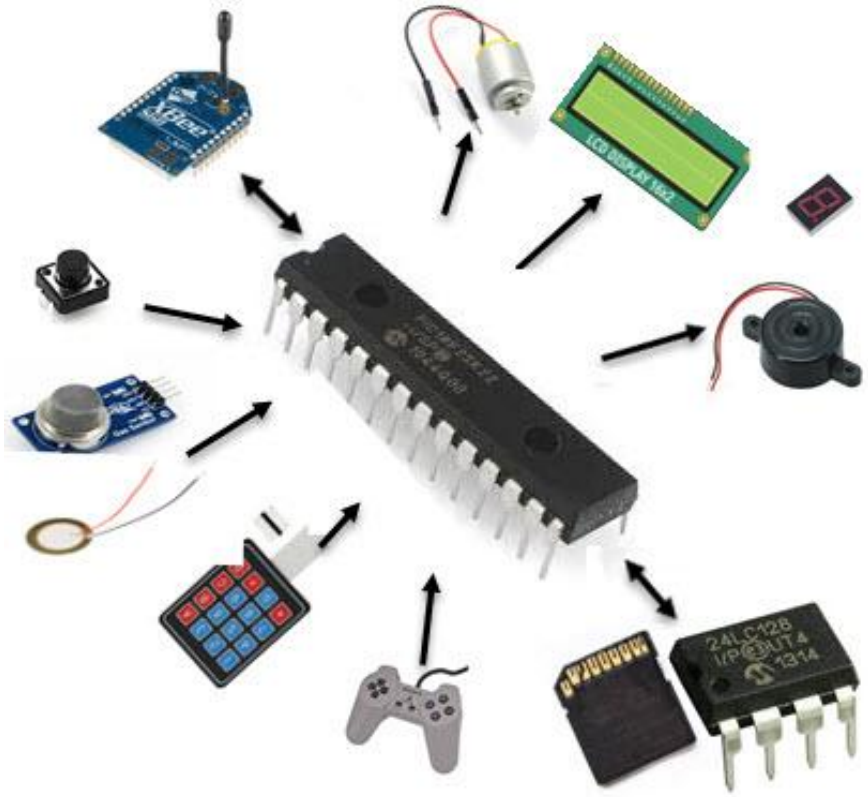


Embedded system requirements

For an embedded designer, you should be familiar with firmware development. Firmware Development includes the following tasks:

1: Writing the Code: Code is usually written in C. You should have some insight on programming languages since these will help you understand the way your processor works.

2: Configuring the peripherals: Embedded system communicates with the outside world via their peripherals. There are many types of peripherals that an embedded system might use, for example, Serial Communication Interfaces (SCI), USBs, Analog to Digital Converters, Synchronous Serial Communication Interfaces such as I2C, etc. As a designer, you should know how these peripherals work and how to implement them into the system.





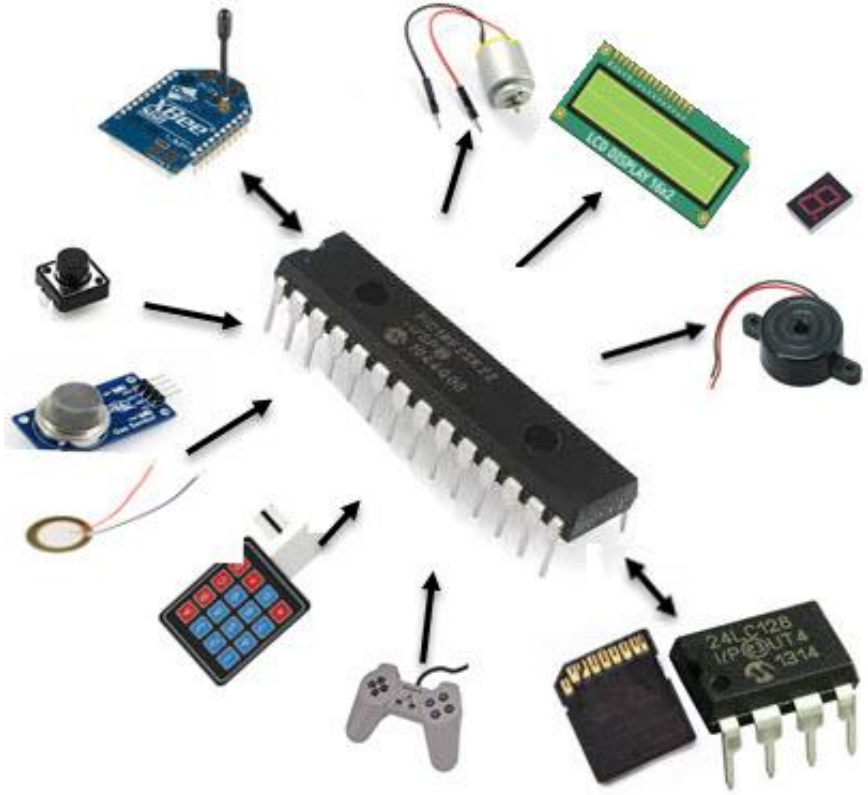
Embedded system requirements

3: Testing the Code: Once you finish the design and the code, you will have to test the functionality of the device in different operating conditions. This may include unit tests, high-level tests, and integration tests.

4: Refining your Code: You will have to bring your code to a refined and functional state by identifying all the adjustments you may need to make to your code before it is fully ready.

5: Debugging your Code: Debugging is basically correcting all the errors and mistakes in a system or in this case, a code. It is impossible for a system to not show errors, so you must thoroughly debug the entire code carefully. Proper skills in debugging only come from experience.

6: Verifying the Code: The last step is to verify your code. In other words, you will test to see how well it can function or how well it performs under specific cases. One type of test is the 'monkey test', in which you will give random continuous input to the device and see that it doesn't malfunction.

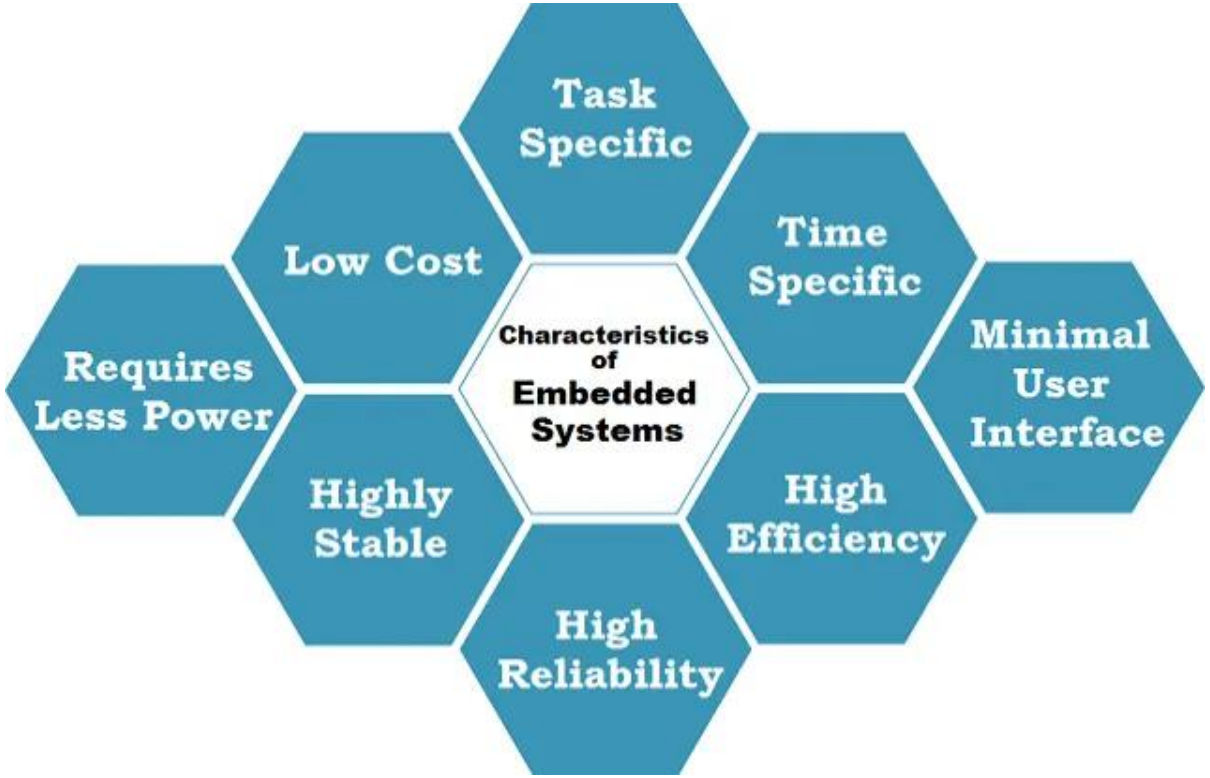




Characteristic of embedded system

1: All Embedded Systems are task specific. They do the same task repeatedly /continuously over their lifetime. An mp3 player will function only as an mp3 player.

2: Embedded systems are created to perform the task within a certain time frame. It must therefore perform fast enough. A car's brake system, if exceeds the time limit, may cause accidents.

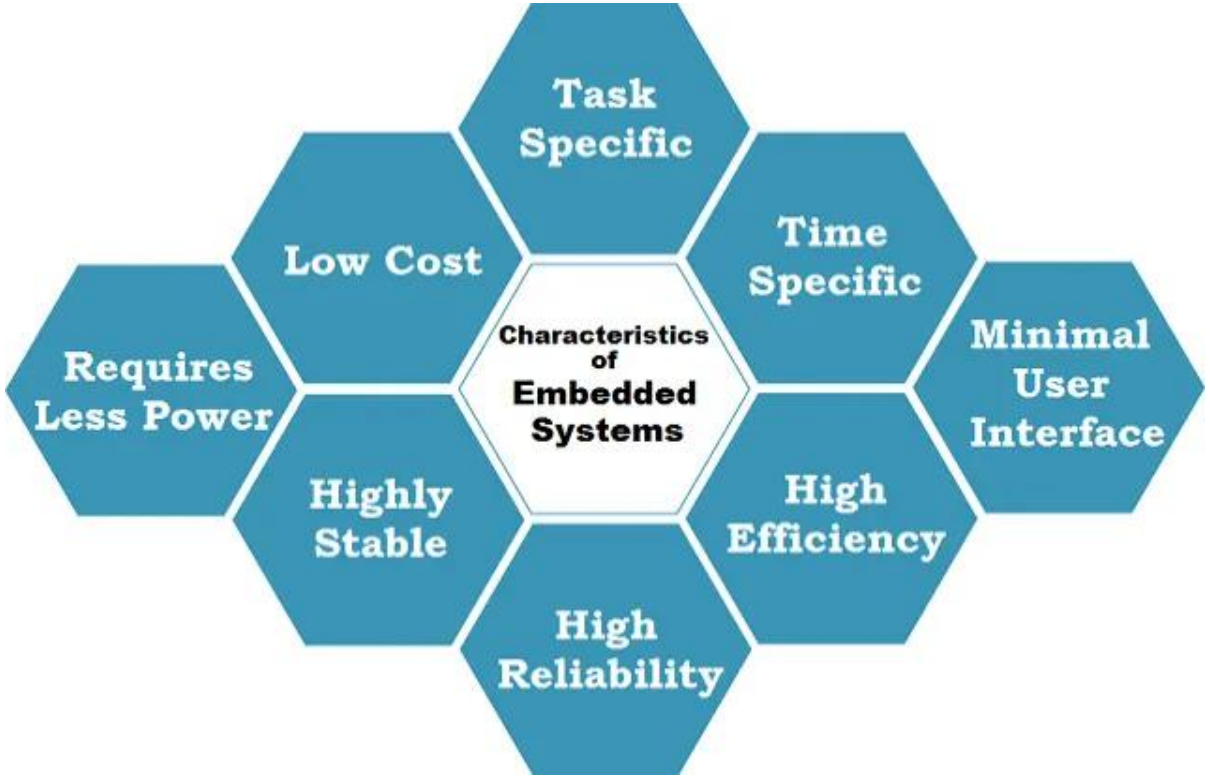




Characteristic of embedded system

3: They have minimal or no user interface (UI). A fully automatic washing machine works on its own after the program is set and stops once the task is over.

4: Embedded systems are built to achieve certain efficiency levels. They are small sized, can work with less power and are not too expensive.

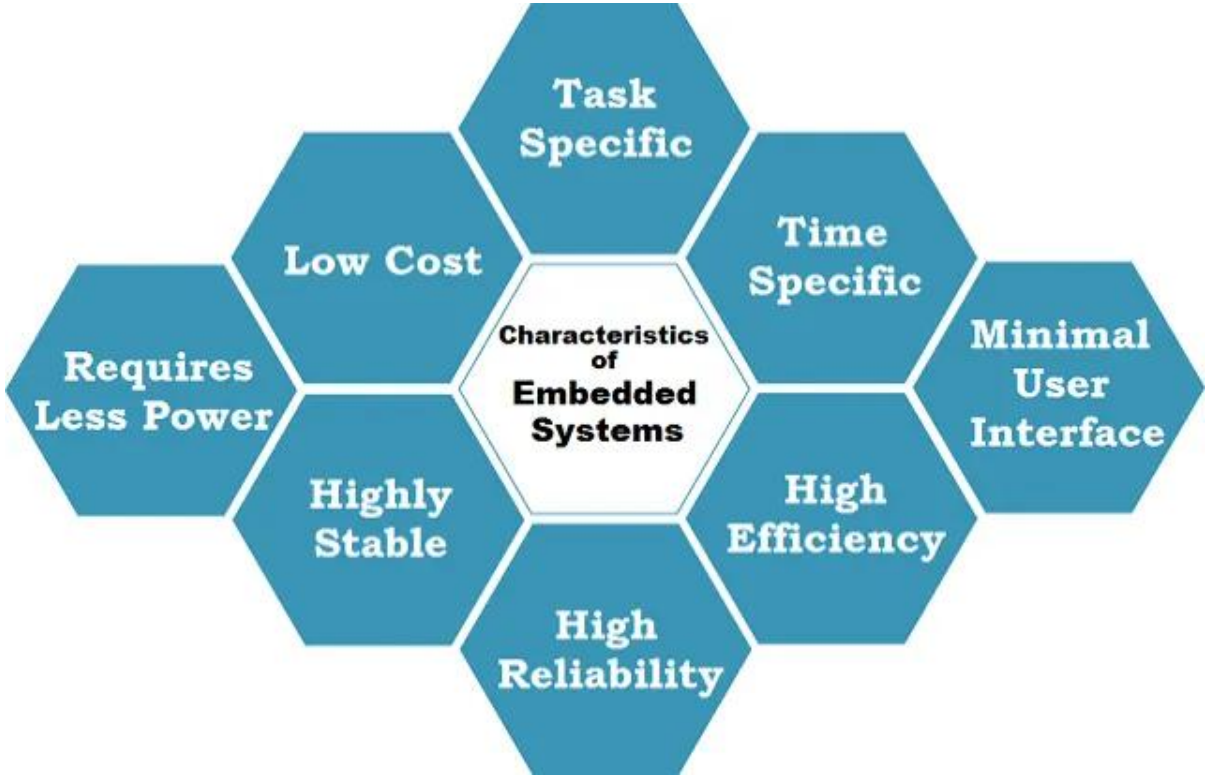




Characteristic of embedded system

5: Embedded systems cannot be changed or upgraded by the users. Hence, they must rank high on reliability and stability.

They are expected to function for long durations without the user experiencing any difficulties

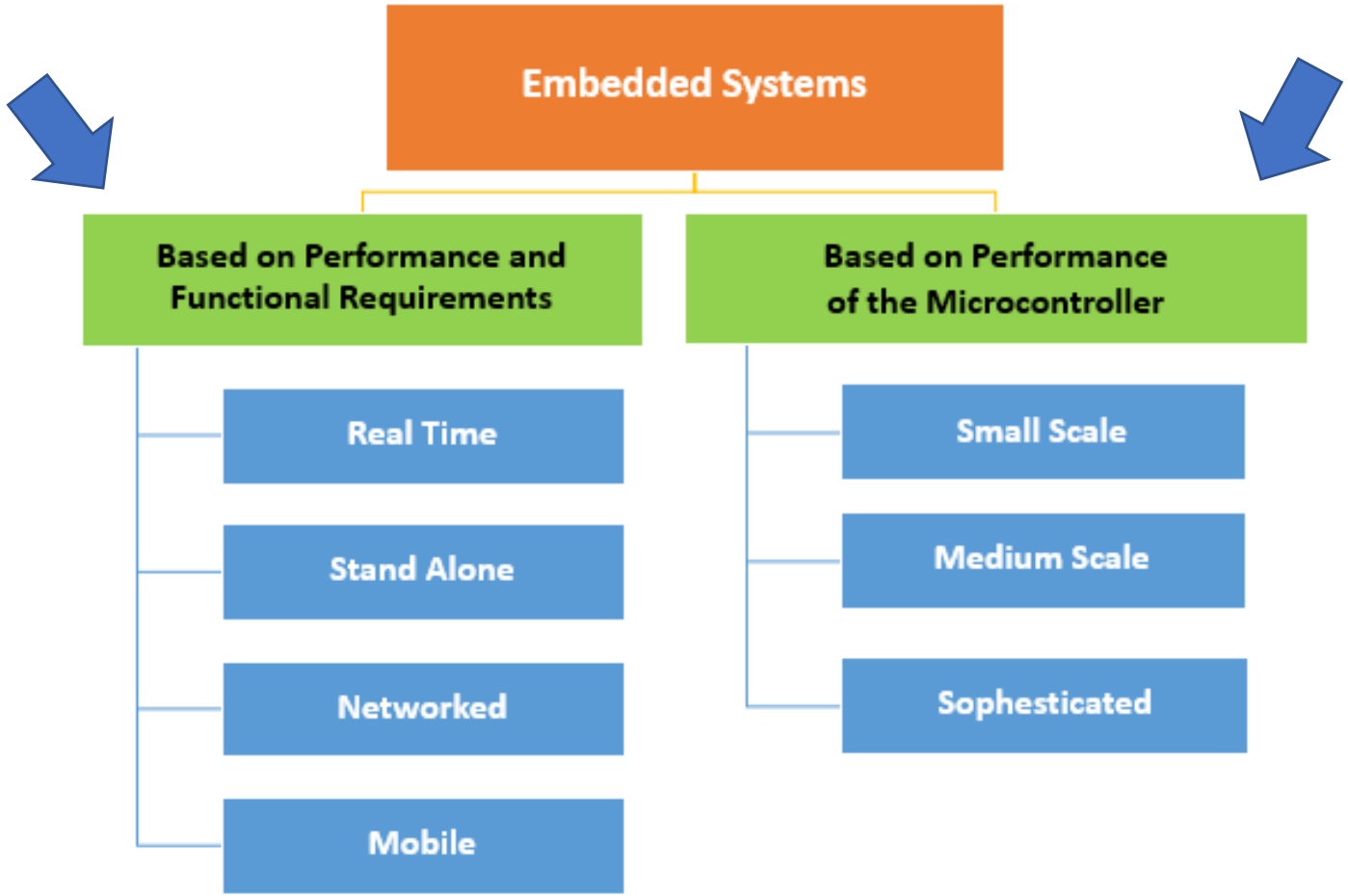




Types of Embedded System

Embedded systems can be classified based on:

- Performance and Functional requirements
- Performance of the Microcontroller

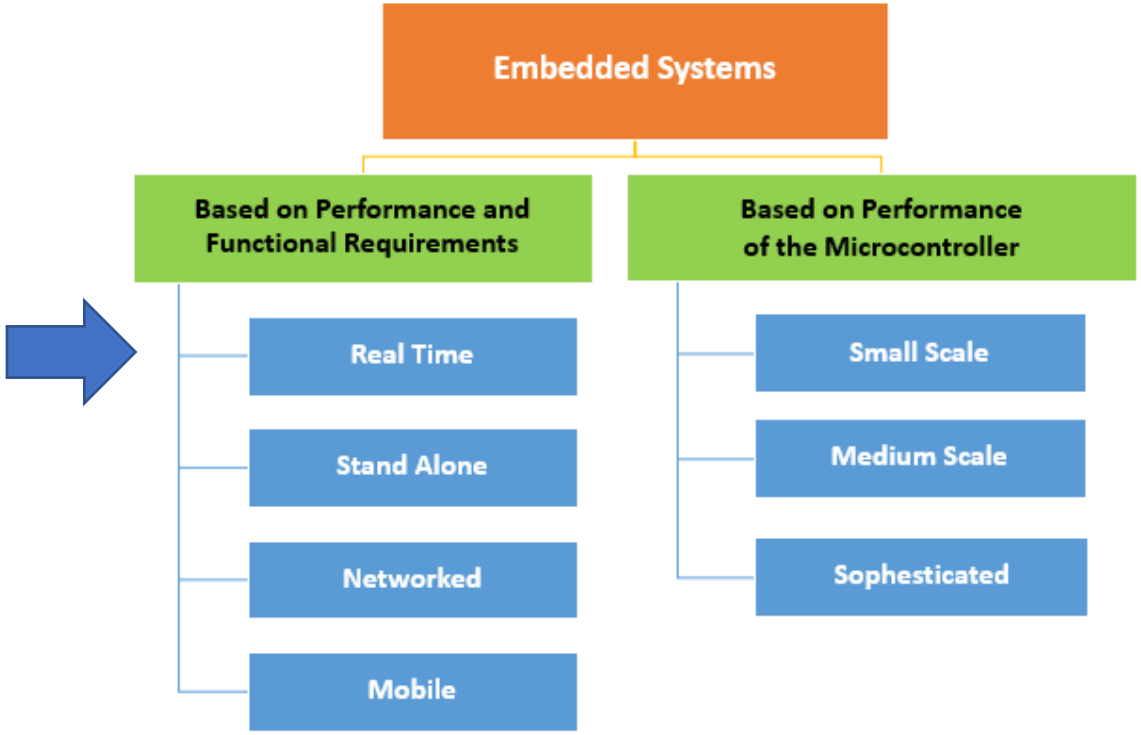




Types of Embedded System

Real Time Embedded Systems

- A Real Time Embedded System provides output within a defined specific time.
- That is, real time embedded systems are designed and created to perform some specific work in pre-specified time.

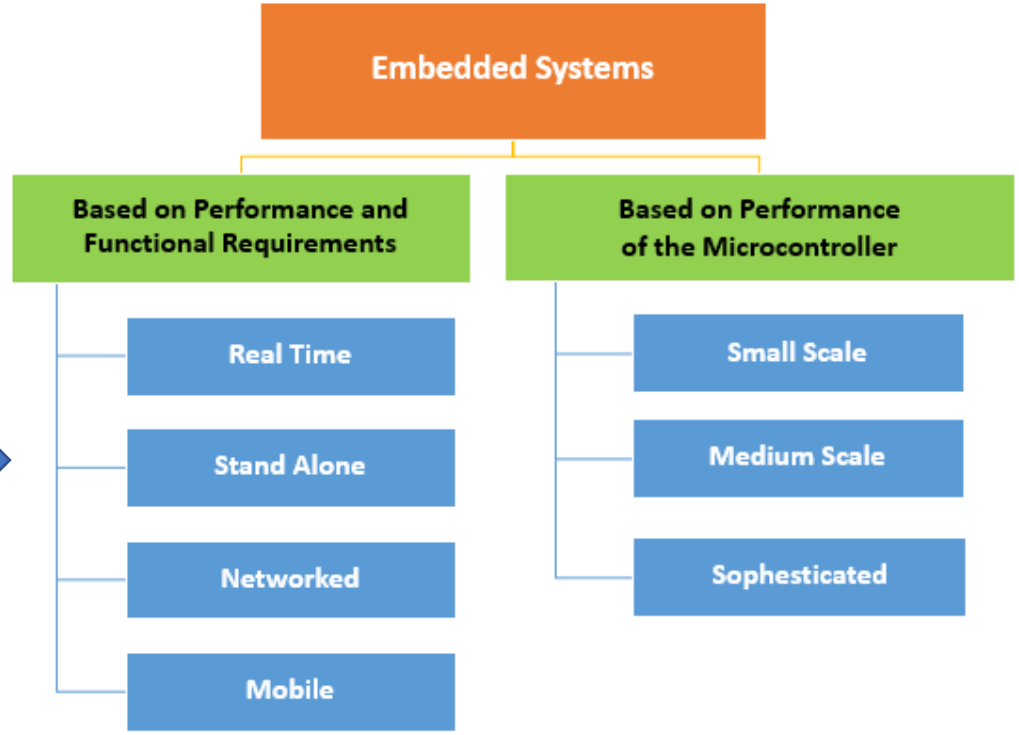




Types of Embedded System

Stand Alone Embedded Systems

- Stand-Alone Embedded Systems are those that can work by themselves i.e. they are self-sufficient and do not depend on a host system.
- Stand-alone embedded systems are made in a way such that an input is received, processed and thereafter the desired output is produces.
- Input can be received via sensors, keyword or push button.

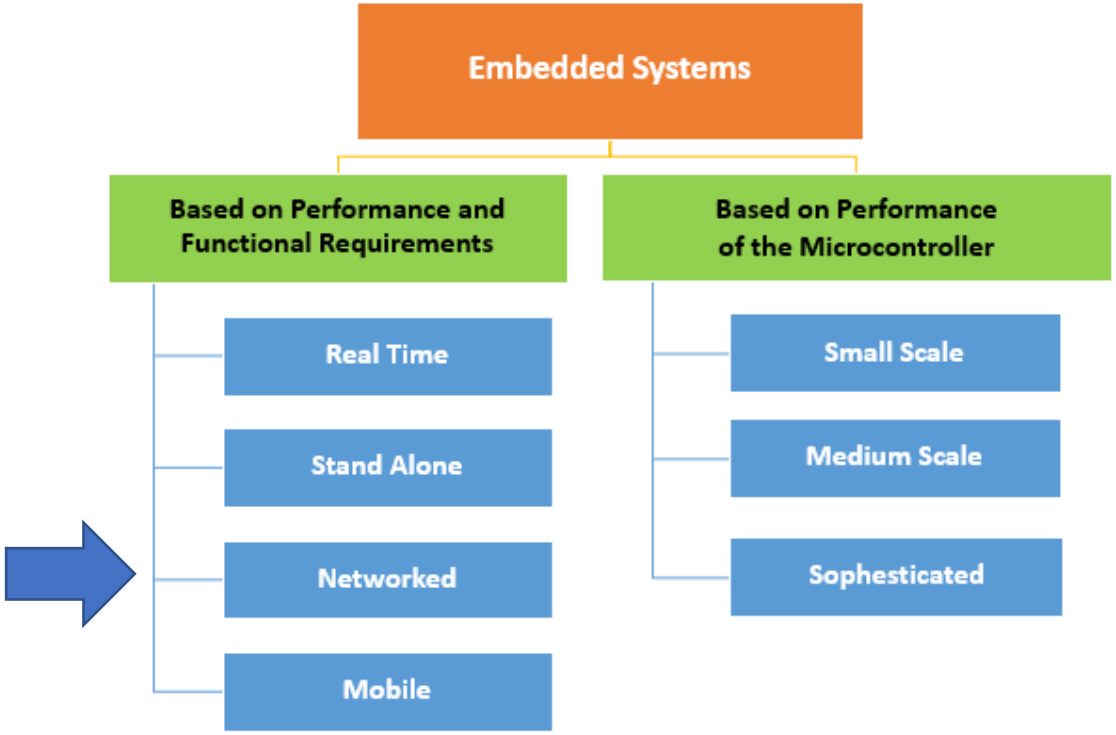




Types of Embedded System

Networked Embedded Systems

- Networked Embedded Systems depend on connected network to perform its assigned tasks.
- These systems consist of components like sensors, controllers etc. which are interconnected.
- Many of these systems are built on general purpose processors

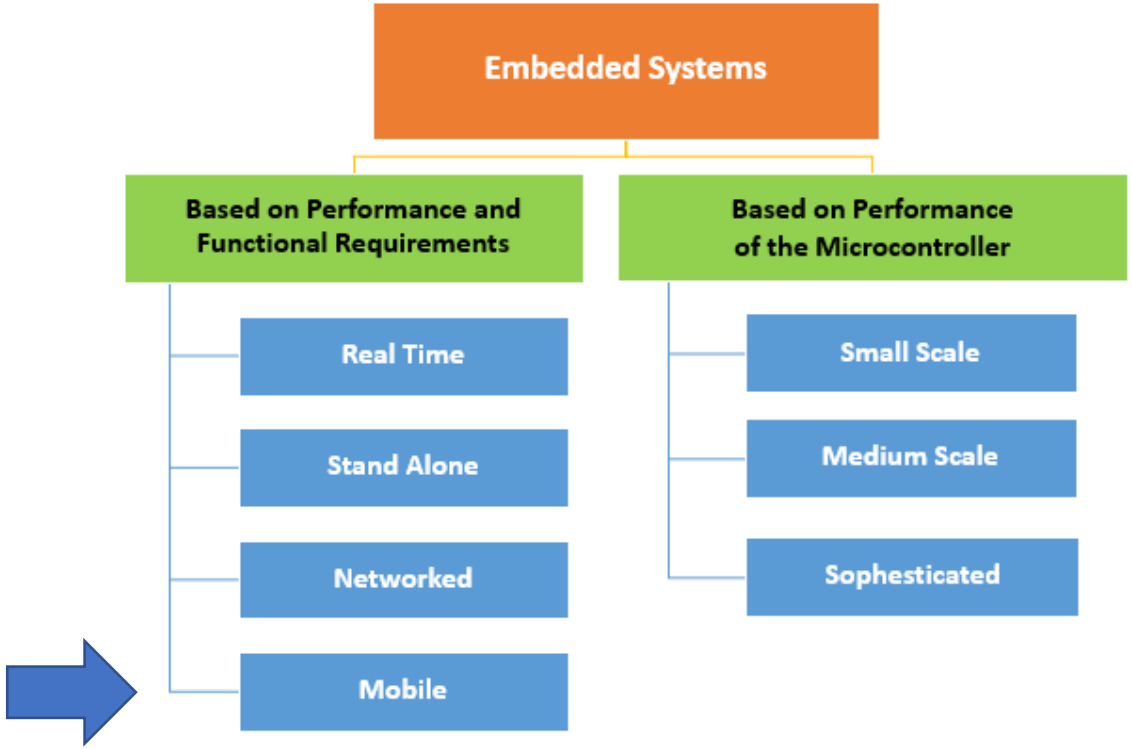




Types of Embedded System

Mobile Embedded Systems

- Mobile Embedded Systems are those that are small sized and can be used in smaller devices.
- They are used in mobile phones and digital cameras because of the small size.
- They often have memory constraints and lacks good user interface.





Types of Embedded System

Small Scale Embedded System

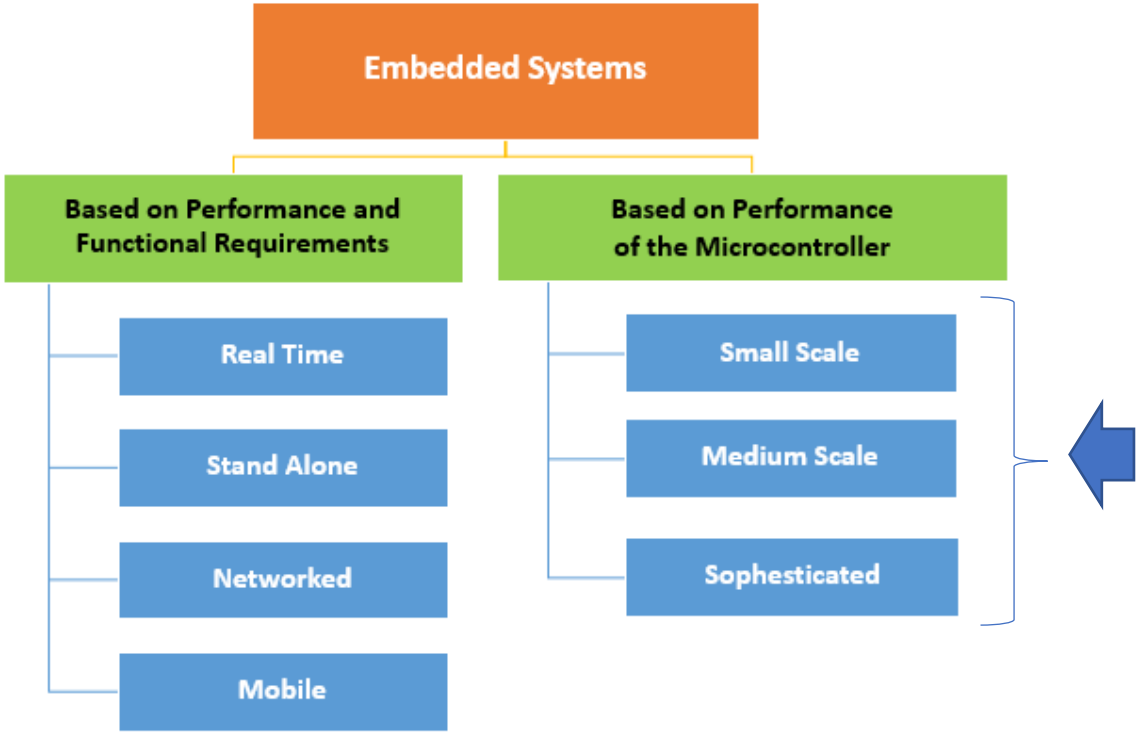
- Small Scale Embedded System is normally designed and created using an 8-bit microcontroller.
- This microcontroller can be battery activated.

Medium Scale Embedded System

- Medium Scale Embedded System uses a single 16-bit or 32-bit microcontroller or multiple microcontrollers linked together.
- These systems have a lot of hardware as well as software complexities, hence are not preferred by many.

Sophisticated Embedded System

- Sophisticated Embedded System often function on multiple algorithms that results in complexities in both hardware and software.
- They often need a processor that is configurable and logic array that can be programmed.





Applications of Embedded System

Some real-life applications of Embedded Systems are as follows:

- *Consumer electronics* – Televisions and digital cameras; computer printers; video game consoles and home entertainment systems like PS4.
- *Household appliances* – Refrigerators; washing machines, microwave ovens, air conditioners
- *Medical equipment* – Scanners like those for MRI, CT; ECG machines; devices to monitor blood pressure and heartbeat
- *Automobiles* – Fuel injection systems, anti-lock braking systems, music and entertainment systems, controls for air-conditioner
- *Industrial applications* – Assembly lines, systems for feedback, systems for data collection
- *Aerospace* – Systems for navigation and guidance, GPS
- *Communications* – Routers, satellite phones





Advantages of Embedded System

The advantages of Embedded Systems are:

- They are convenient for mass production. This results in low price per piece.
- These systems are highly stable and reliable.
- Embedded systems are made for specific tasks.
- The embedded systems are very small in size, hence can be carried and loaded anywhere.
- These systems are fast.
- They also use less power.



The Core of the Embedded Systems

- **The core of the embedded system falls into any one of the following categories.**

1. **General Purpose and Domain Specific Processors**
   - Microprocessors
   - Microcontrollers
   - Digital Signal Processors

2. **Programmable Logic Devices (PLDs)**

3. **Application Specific Integrated Circuits (ASICs)**



The Core of the Embedded Systems

1: Microprocessor

- A silicon chip representing a Central Processing Unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of Instructions, which is specific to the manufacturer
- In general the CPU contains the Arithmetic and Logic Unit (ALU), Control Unit and Working registers
- Microprocessor is a dependent unit and it requires the combination of other hardware like Memory, Timer Unit, and Interrupt Controller etc for proper functioning.



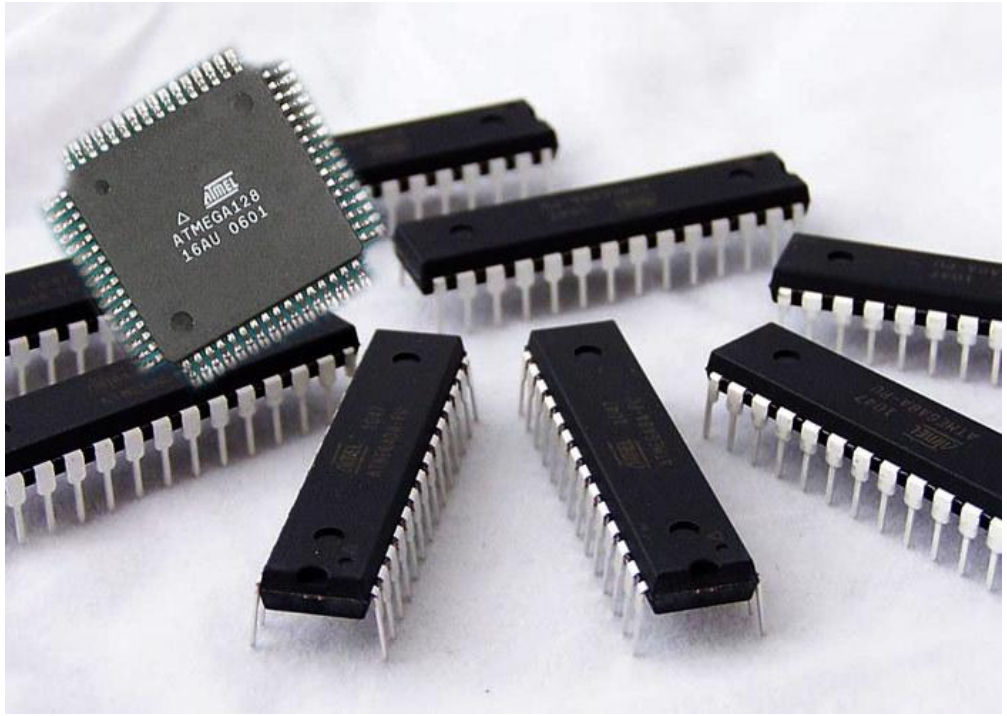




The Core of the Embedded Systems

2: Microcontroller:

- A highly integrated silicon chip containing a CPU, RAM, Special and General purpose Register Arrays, On Chip ROM/FLASH memory for program storage, Timer and Interrupt control units and dedicated I/O ports
- Since a microcontroller contains all the necessary functional blocks for independent working, they found greater place in the embedded domain in place of microprocessors
- Microcontrollers are cheap, cost effective and are readily available in the market





The Core of the Embedded Systems

3:Digital Signal Processor (DSP)

- is a specialized microprocessor chip, with its architecture optimized for the operational needs of digital signal processing.
- DSPs are fabricated on MOS integrated circuit chips. They are widely used in audio signal processing, telecommunications, digital image processing, radar, sonar and speech recognition systems
- The goal of a DSP is usually to measure, filter or compress continuous real-world analog signals.
- Most general-purpose microprocessors can also execute digital signal processing algorithms successfully, but may not be able to keep up with such processing continuously in real-time. Also, dedicated DSPs usually have better power efficiency, thus they are more suitable in portable devices such as mobile phones because of power consumption constraints



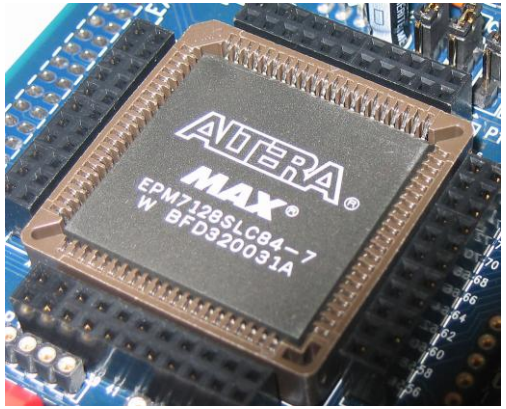


The Core of the Embedded Systems

4: Programmable Logic Device (PLD)

- is an electronic component used to build reconfigurable digital circuits.
- Unlike digital logic constructed using discrete logic gates with fixed functions, a PLD has an undefined function at the time of manufacture.
- Before the PLD can be used in a circuit it must be programmed to implement the desired function.
- Compared to fixed logic devices, programmable logic devices simplify the design of complex logic and may offer superior performance.
- Unlike for microprocessors, programming a PLD changes the connections made between the gates in the device.
- PLDs can broadly be categorized into,
  - Simple Programmable Logic Devices (SPLDs), comprising programmable array logic, programmable logic array and generic array logic;
  - Complex Programmable Logic Devices (CPLDs) and Field-Programmable Gate Arrays (FPGAs).







The Core of the Embedded Systems

5: Application-Specific Integrated Circuit (ASIC)

- is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use.
- For example, a chip designed to run in a digital voice recorder or a high-efficiency bitcoin miner is an ASIC.

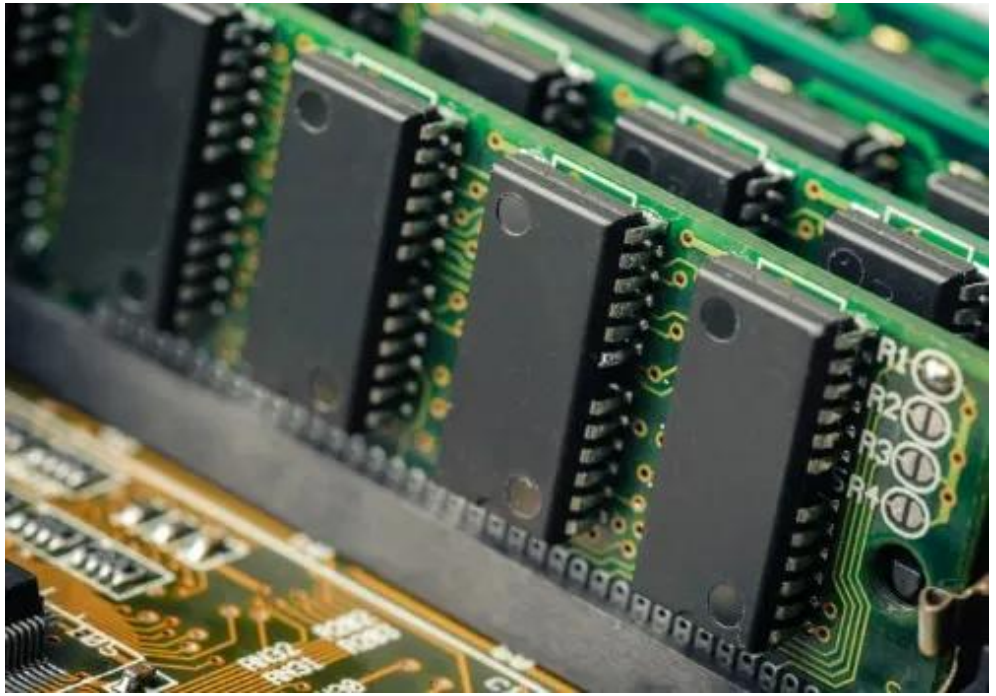


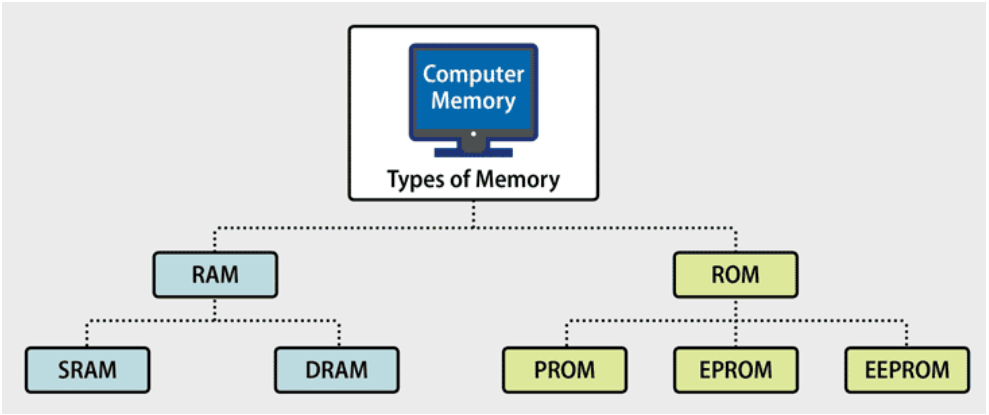




The Core of the Embedded Systems

- Memory is an important part of an embedded system.
- The memory used in embedded system can be either Program Storage Memory (**ROM**) or Data memory (**RAM**)
- Certain Embedded processors/controllers contain built in program memory and data memory and this memory is known as on-chip memory
- Certain Embedded processors/controllers do not contain sufficient memory inside the chip and requires external memory called off-chip memory or external memory.







Week 1

Lab 0 / LO1

Introduction to Arduino Software Installation



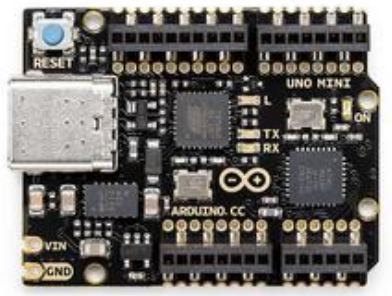
Intro to Arduino

- Arduino is a microcontroller board with different model and controller family.
- Mostly Arduino are based on the AMTEGA chips
- Arduino is designed for fast prototyping (no need for external programmer)
- Arduino includes the controller, the programming part and the power supply unit.





The Many Flavors of Arduino

			
Arduino UNO R3	Arduino Mega 2560 Rev3	Arduino Leonardo	Arduino UNO Mini Limited Edition
			
Arduino Due	Arduino Micro	Arduino Zero	Arduino UNO WiFi Rev2



The Many Flavors of Arduino

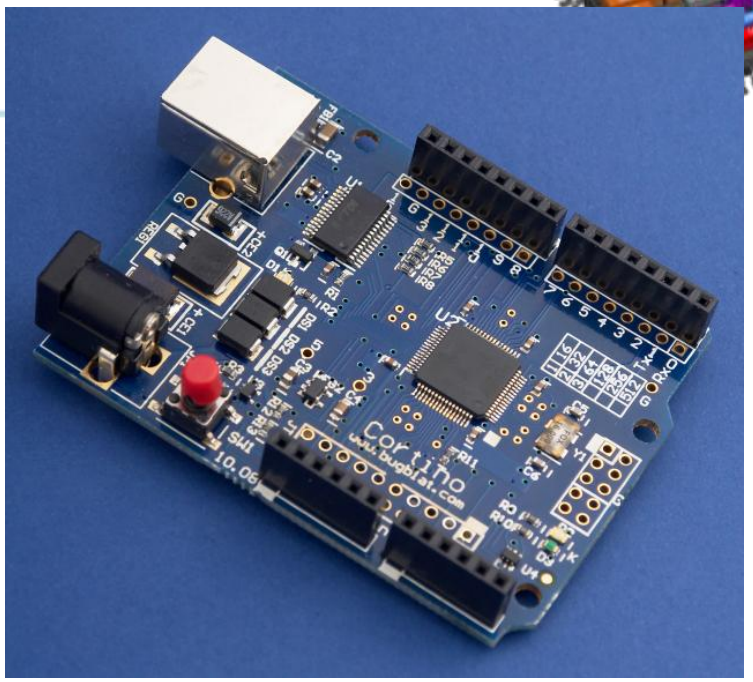
		
Arduino Nano 33 IoT	Arduino Nano RP2040 Connect	Arduino Nano 33 BLE Sense

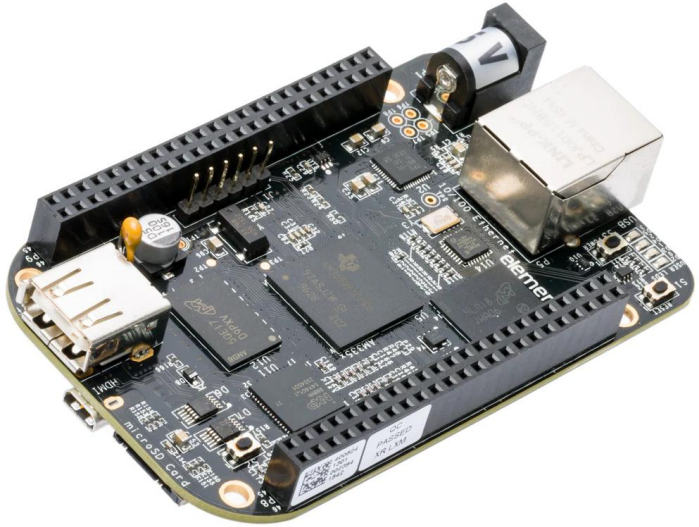
			
Arduino Nano 33 BLE	Arduino Nano Every	Arduino Nano	Arduino Nano Motor Carrier

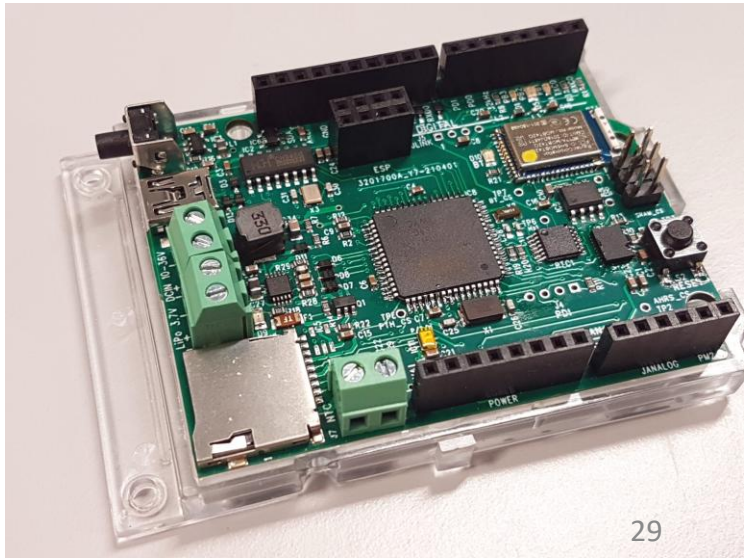
Arduino-like Systems

- Cortino (ARM)
- Xduino (ARM)
- LeafLabs Maple
- BeagleBoard (Linux)
- Wiring Board
- (Arduino predecessor)





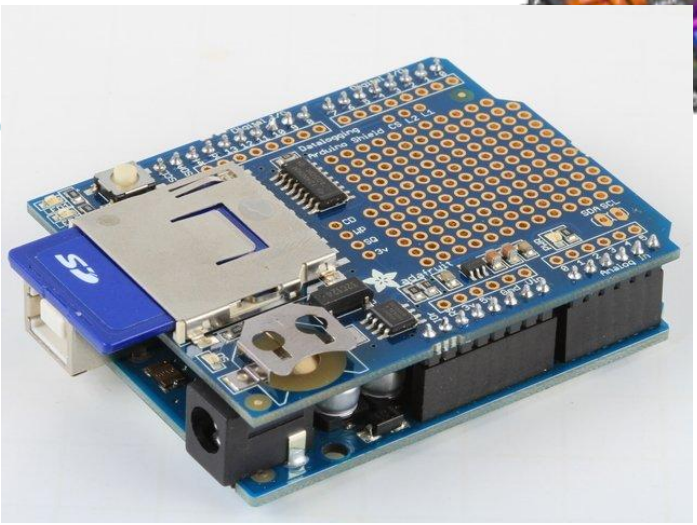


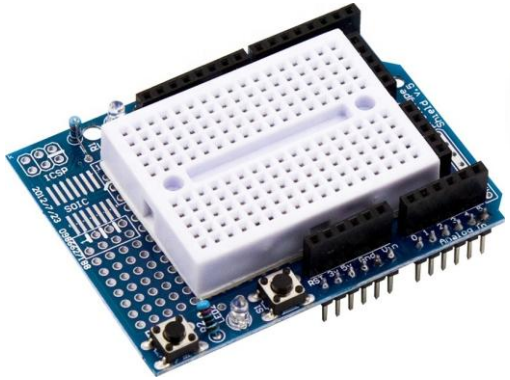


Arduino shields

- TFT Touch Screen
- Data logger
- Motor/Servo shield
- Ethernet shield
- Audio wave shield
- Cellular/GSM shield
- WiFi shield
- Proto-shield
- ...many more







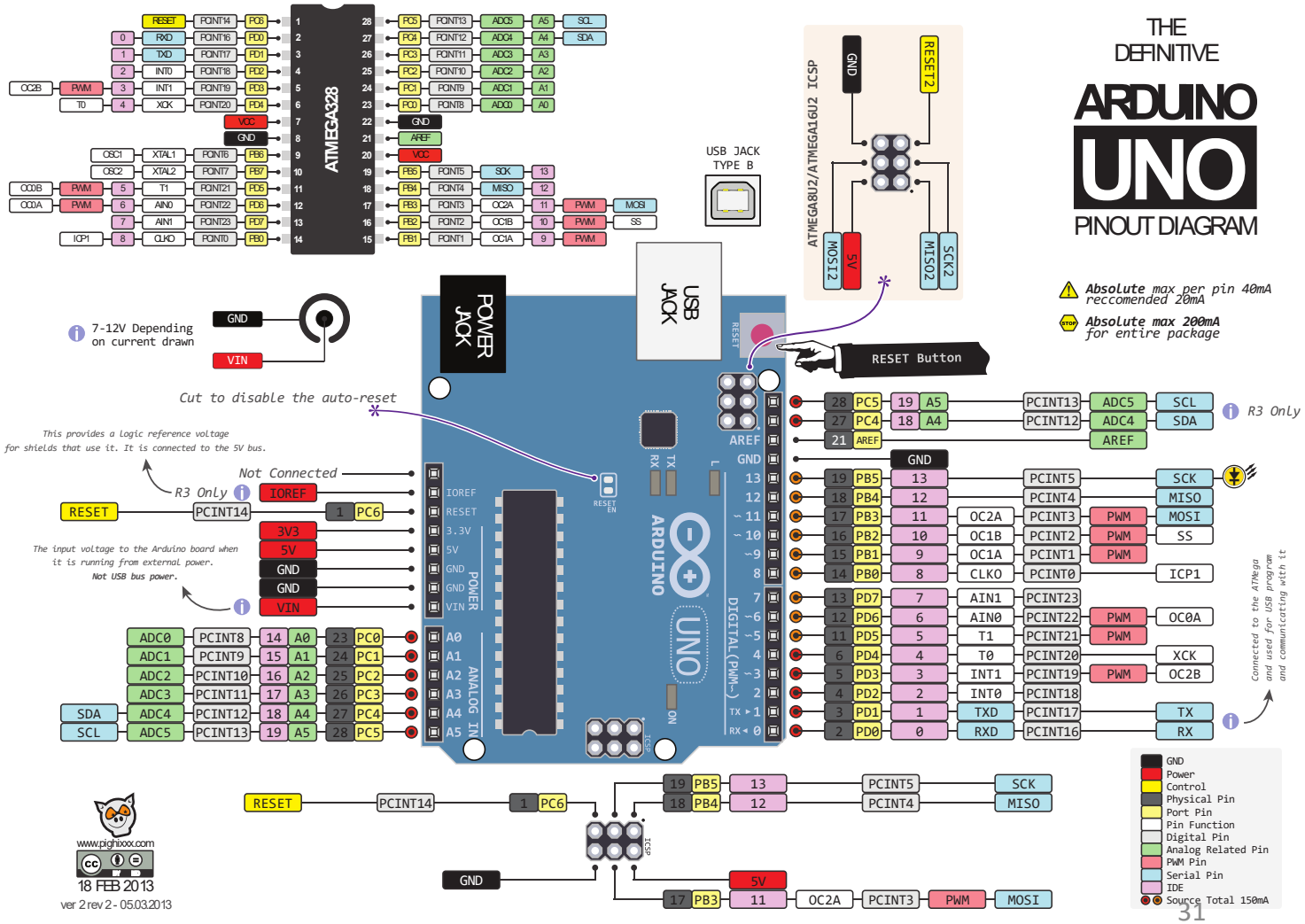






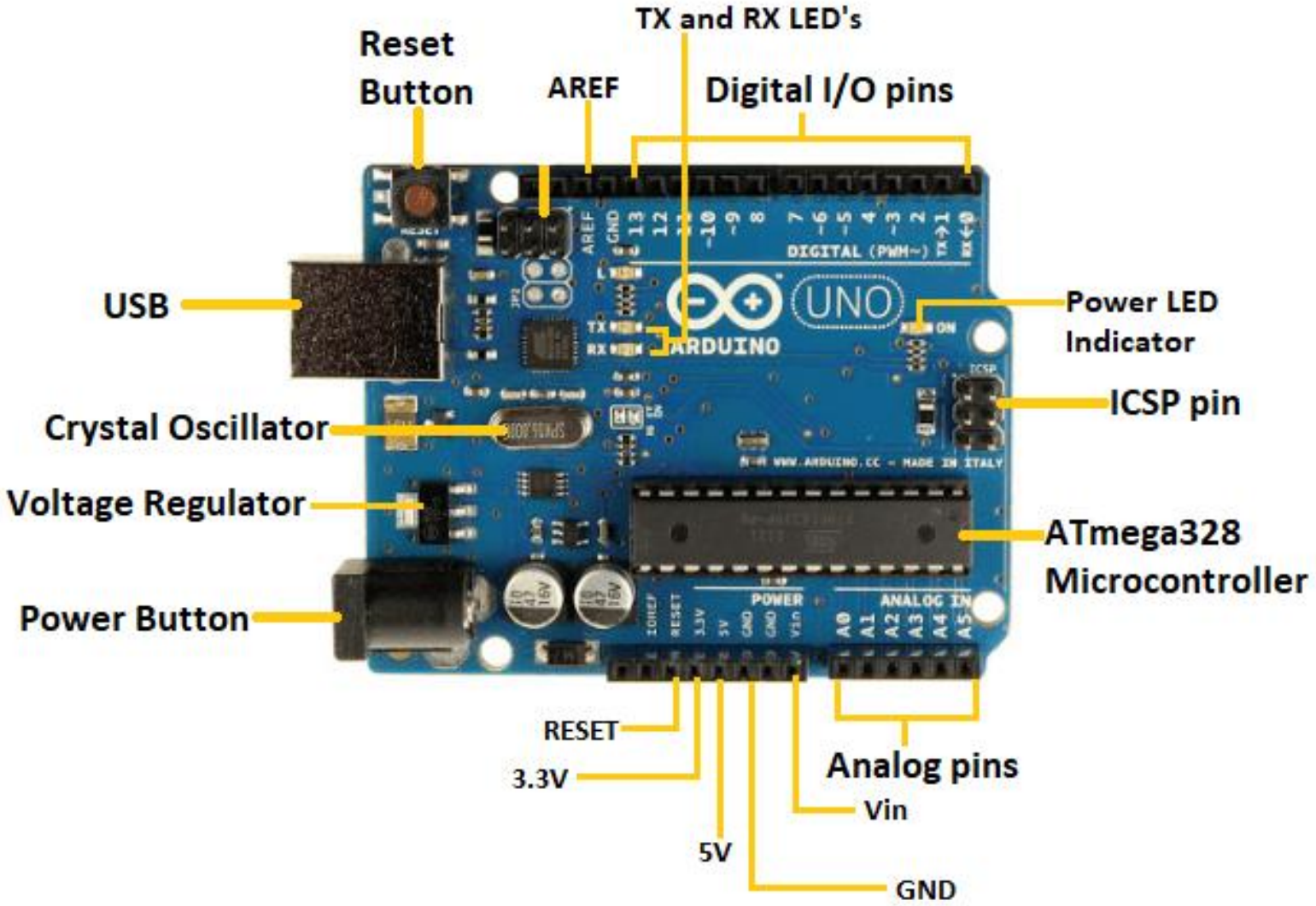
The ATmega328P Microcontroller (used by the Arduino UNO, MINI, NANO)

- AVR 8-bit RISC architecture
- Available in DIP package
- Up to 20 MHz clock
- 32kB flash memory
- 1 kB SRAM
- 23 programmable I/O channels
- Six 10-bit ADC inputs
- Three timers/counters
- Six PWM outputs





Getting to know the Arduino: Electrical Inputs and Outputs

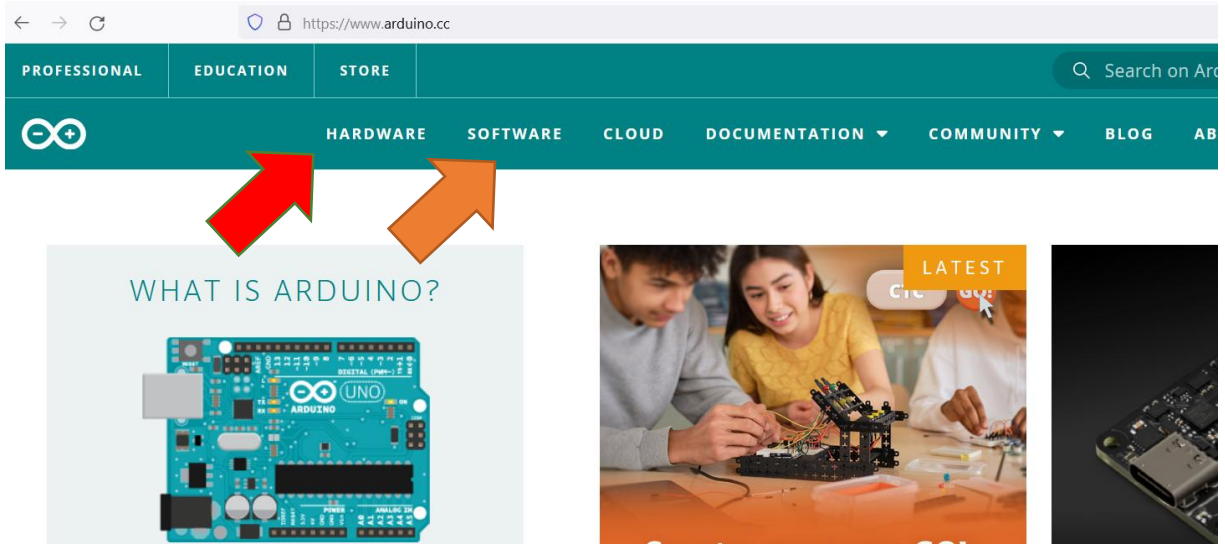




Official website

To check more details about Arduinos, we can click on the following link **Arduino.cc**

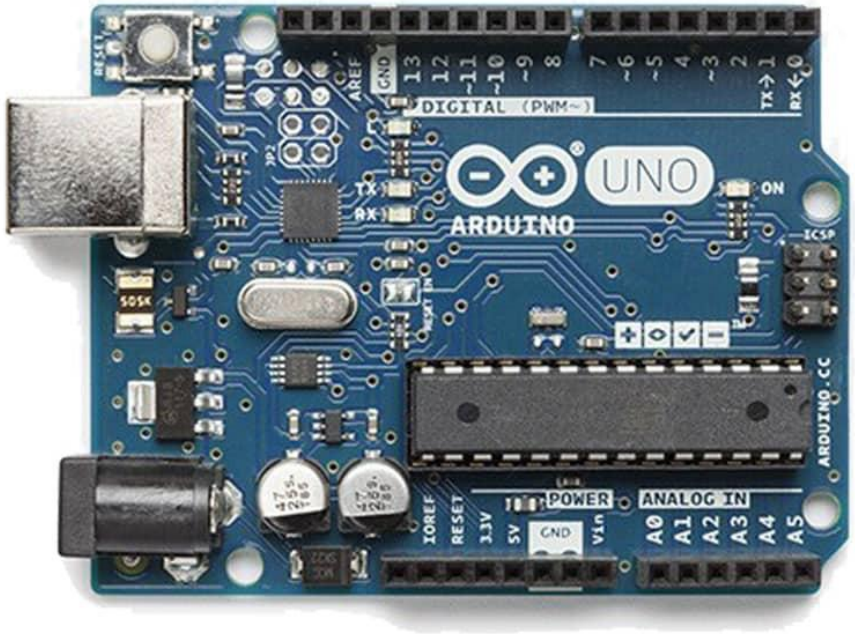
In the link we can discover new model of Arduino board and download the Arduino IDE software

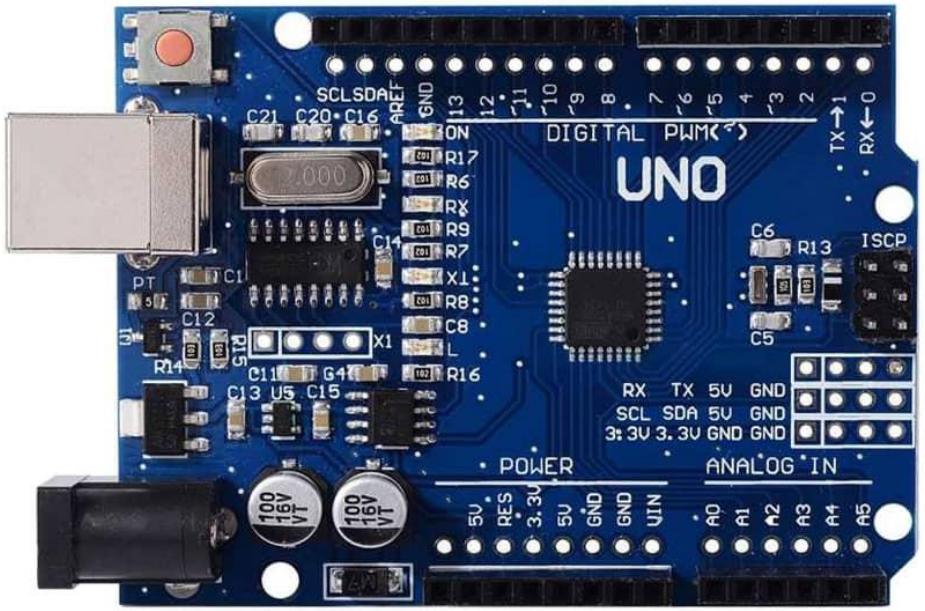




Original vs. clone Arduinos

- Two Arduino UNO models are available in the market
  - The original one (for $ 23)
  - Clone Chinese version (for $6)

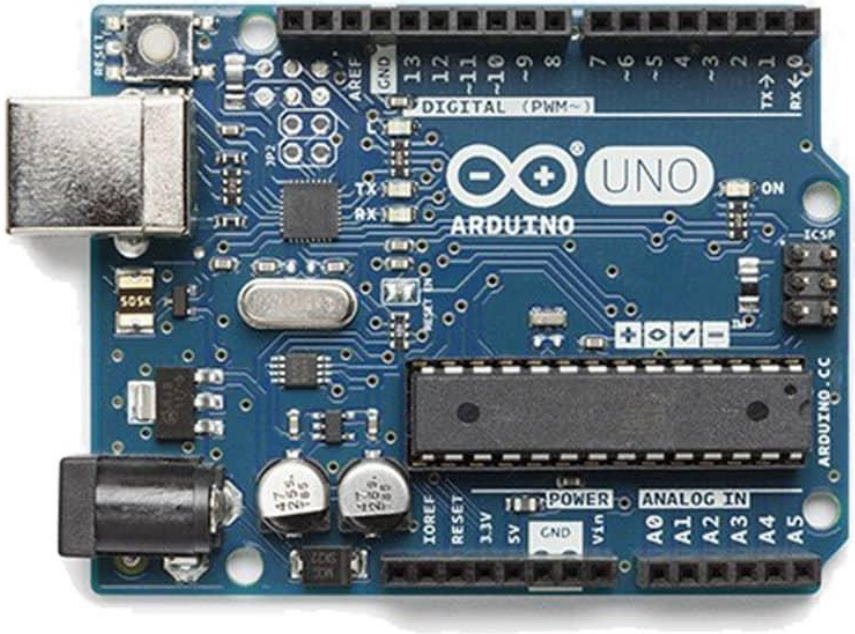


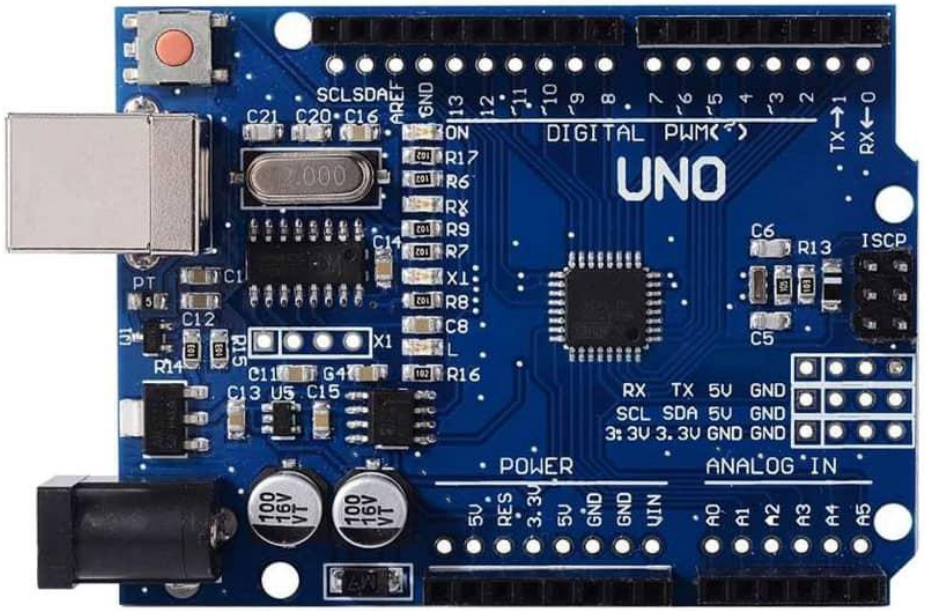




Original vs. clone Arduinos

- To identify the original one we can check the Arduino logo and the USB interface chip that should be from ATMEGA also
- The clone version uses the CH341 chip for the usb interfacing (or some model uses the FT2303 chips)





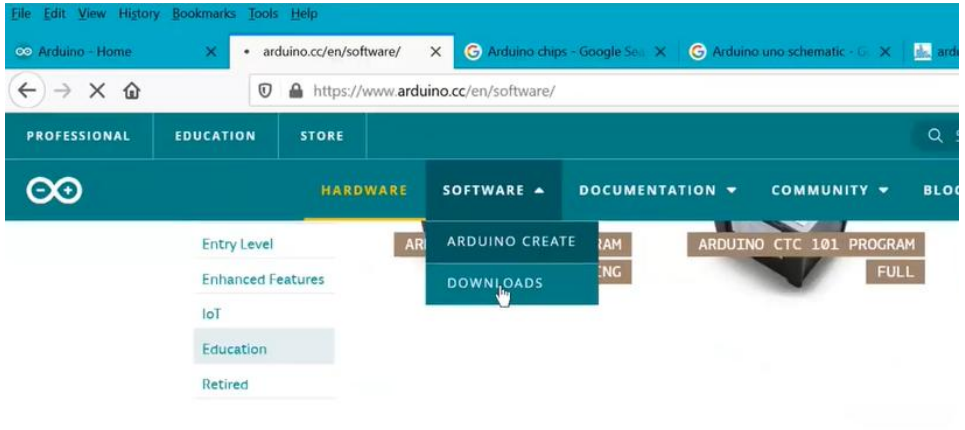


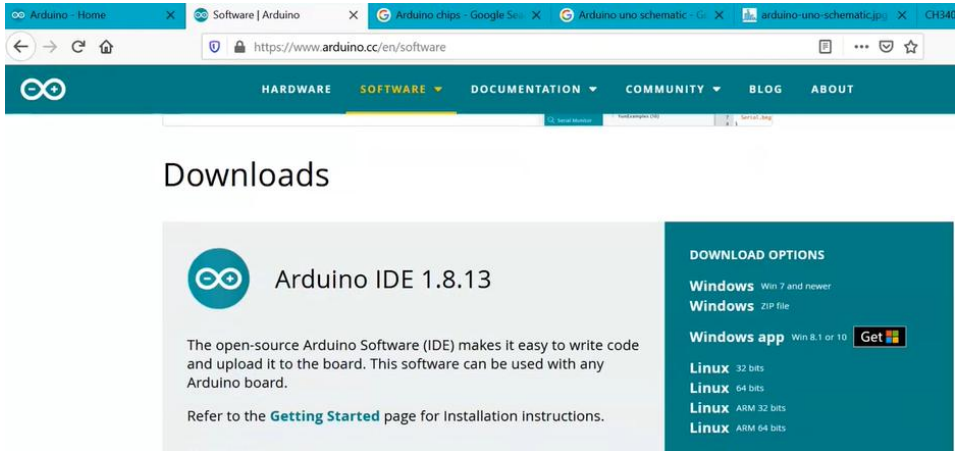
Download The Arduino IDE

Arduino IDE is an open-source software, designed by Arduino.cc and mainly used for writing, compiling & uploading code to almost all Arduino Modules.

It is an official Arduino software, making code compilation too easy that even a common person with no prior technical knowledge can get their feet wet with the learning process.

It is available for all operating systems i.e. MAC, Windows, Linux and runs on the Java Platform that comes with inbuilt functions and commands that play a vital role in debugging, editing and compiling the code.



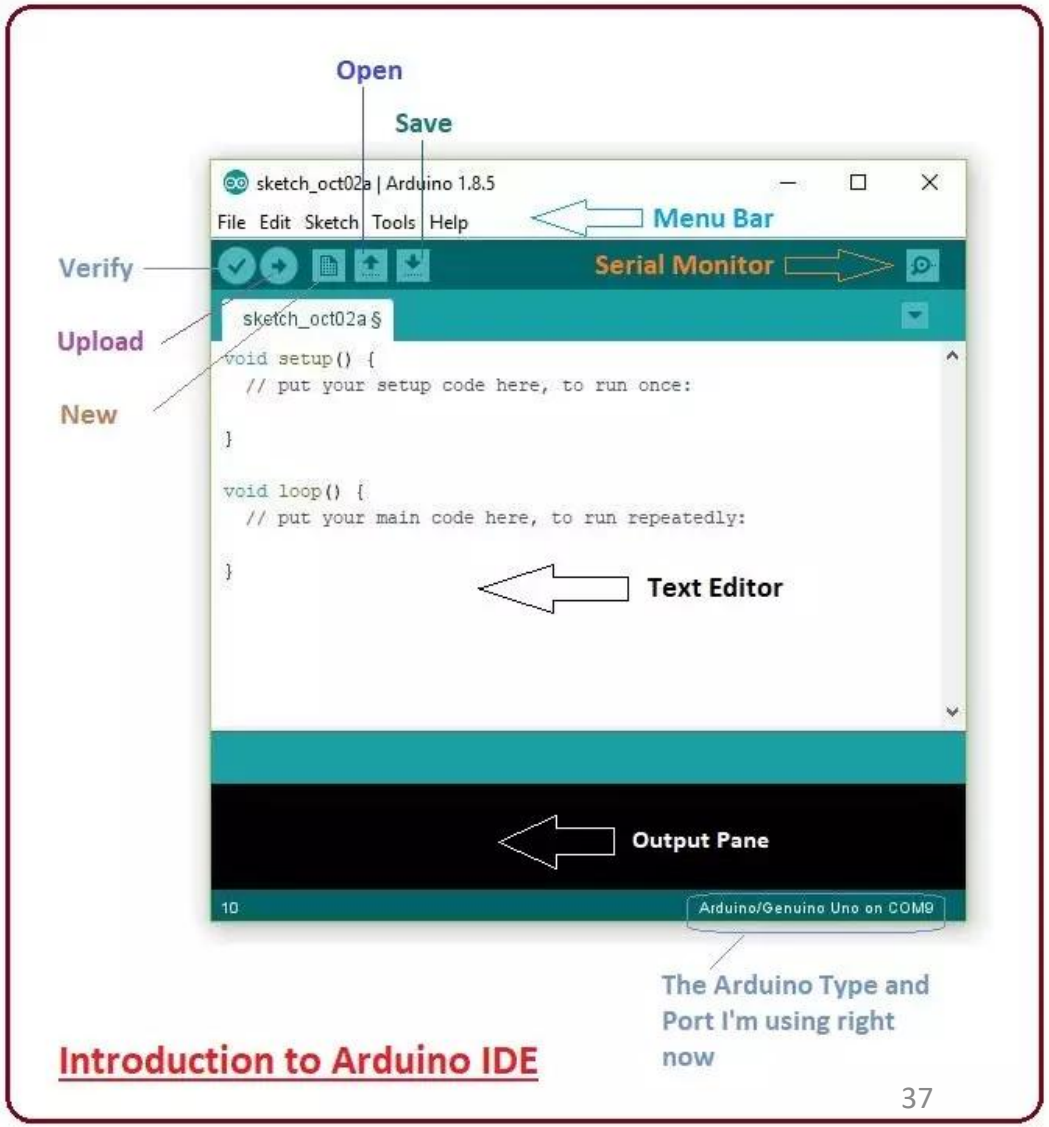




Download The Arduino IDE

The Arduino Integrated Development Environment - or Arduino Software (IDE) - contains

- a text editor for writing code,
- a message area,
- a text console,
- a toolbar with buttons for common functions and a series of menus.
- It connects to the Arduino hardware to upload programs and communicate with them.





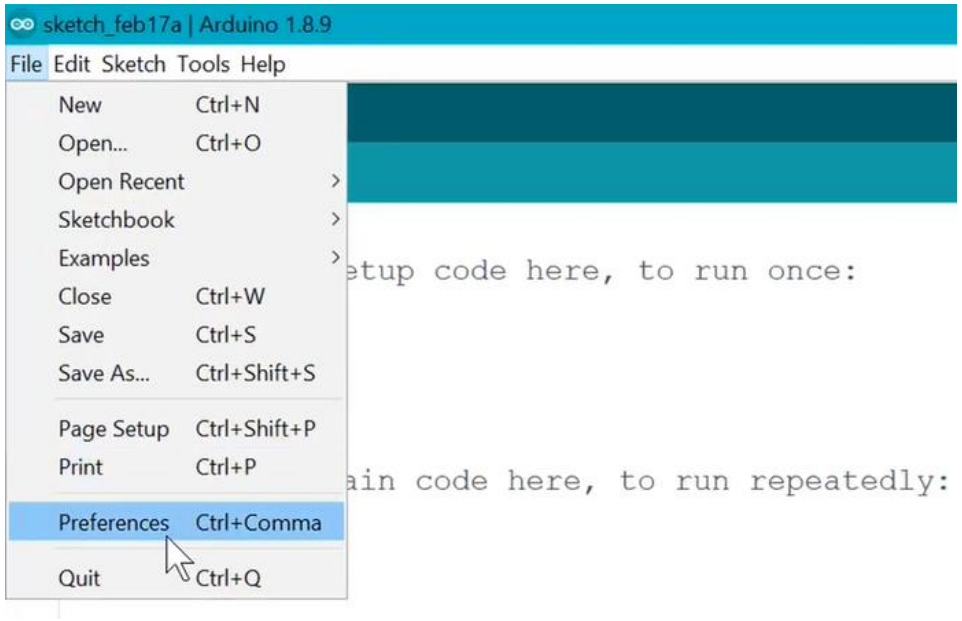
What is a sketch!

- An Arduino code is called sketch. (Default name is sketch_date_day_x)

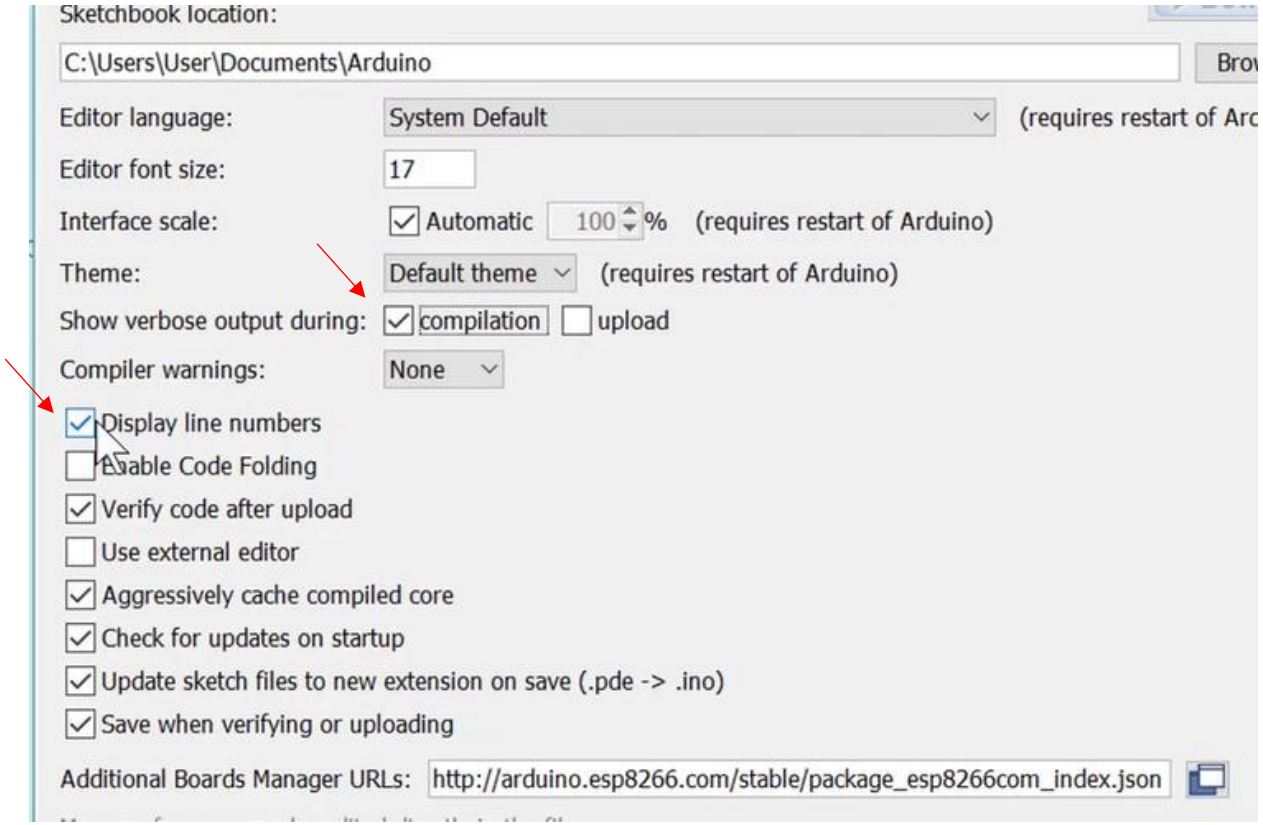




Setting up IDE for the first time



Check the "Show verbose output during **Compilation**
Check the **"Display line numbers"**

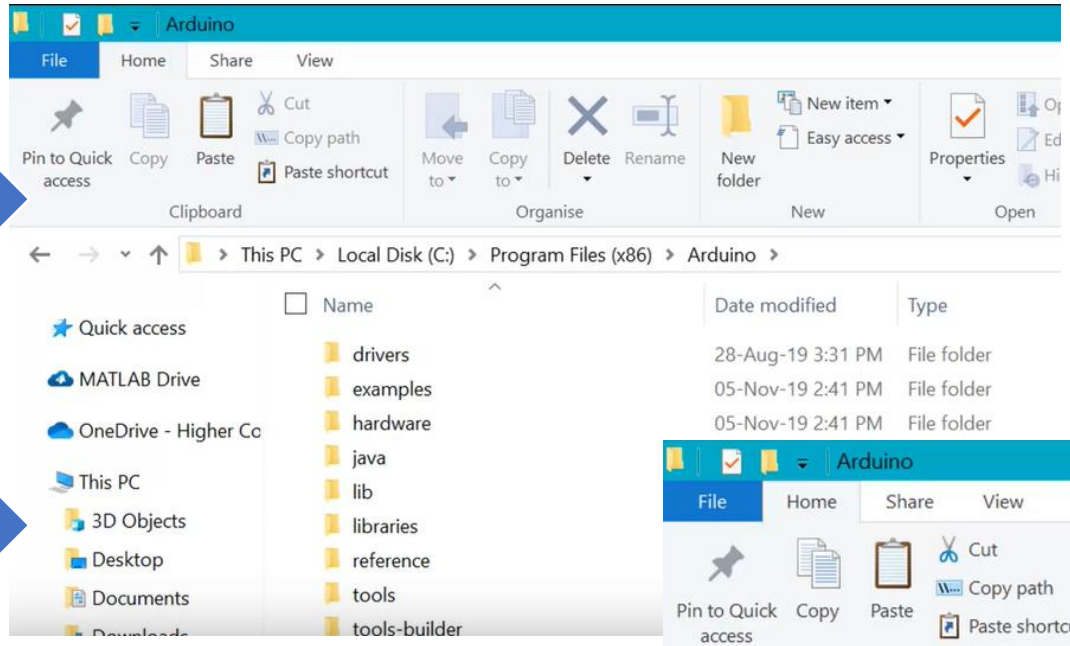


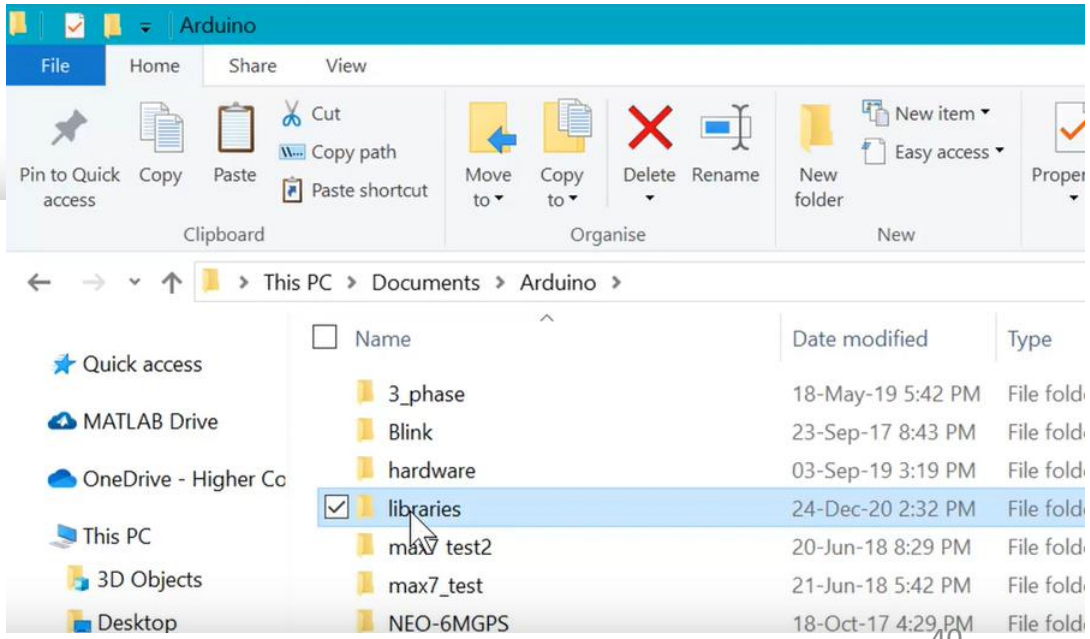


Library folders

Usually, the Arduino installation files are located in program files(x86) (in windows)

It includes the main **libraries** and **examples**





Library folder is the location in which all **recent installed libraries** are placed



Working With Simulators

Arduino codes can be simulated on different platform

1. Proteus (best solution, rich library)
2. Tinkercad (online; limited to Arduino UNO only)
3. Wokwi (Online; All AVR based Arduino are available)

PROTEUS

WOKWi
Simulate IoT Projects in Your Browser

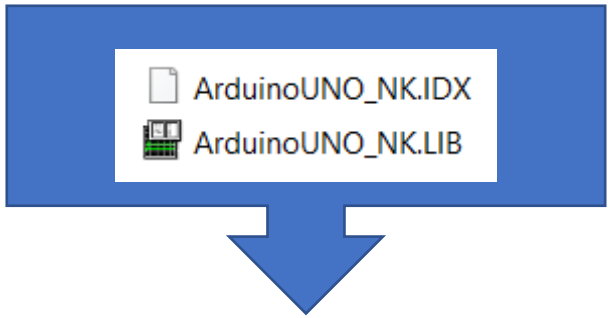
TIN KER CAD AUTODESK Tinkercad

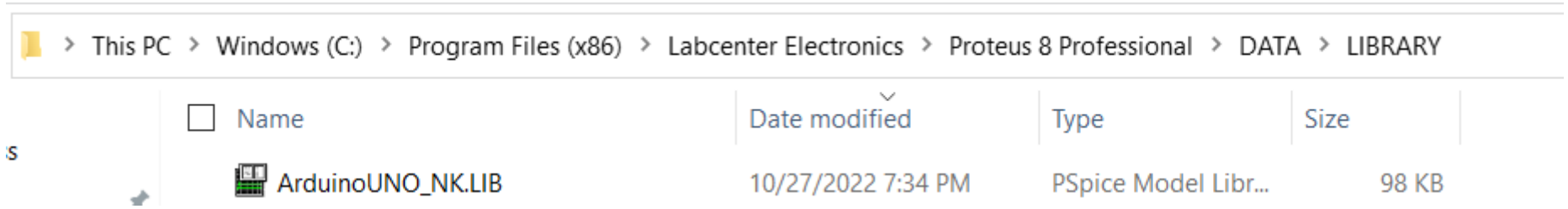


Working With Simulators (Proteus)

Arduino library is not available in proteus by default.

It can be added by copying the following files in the library folder

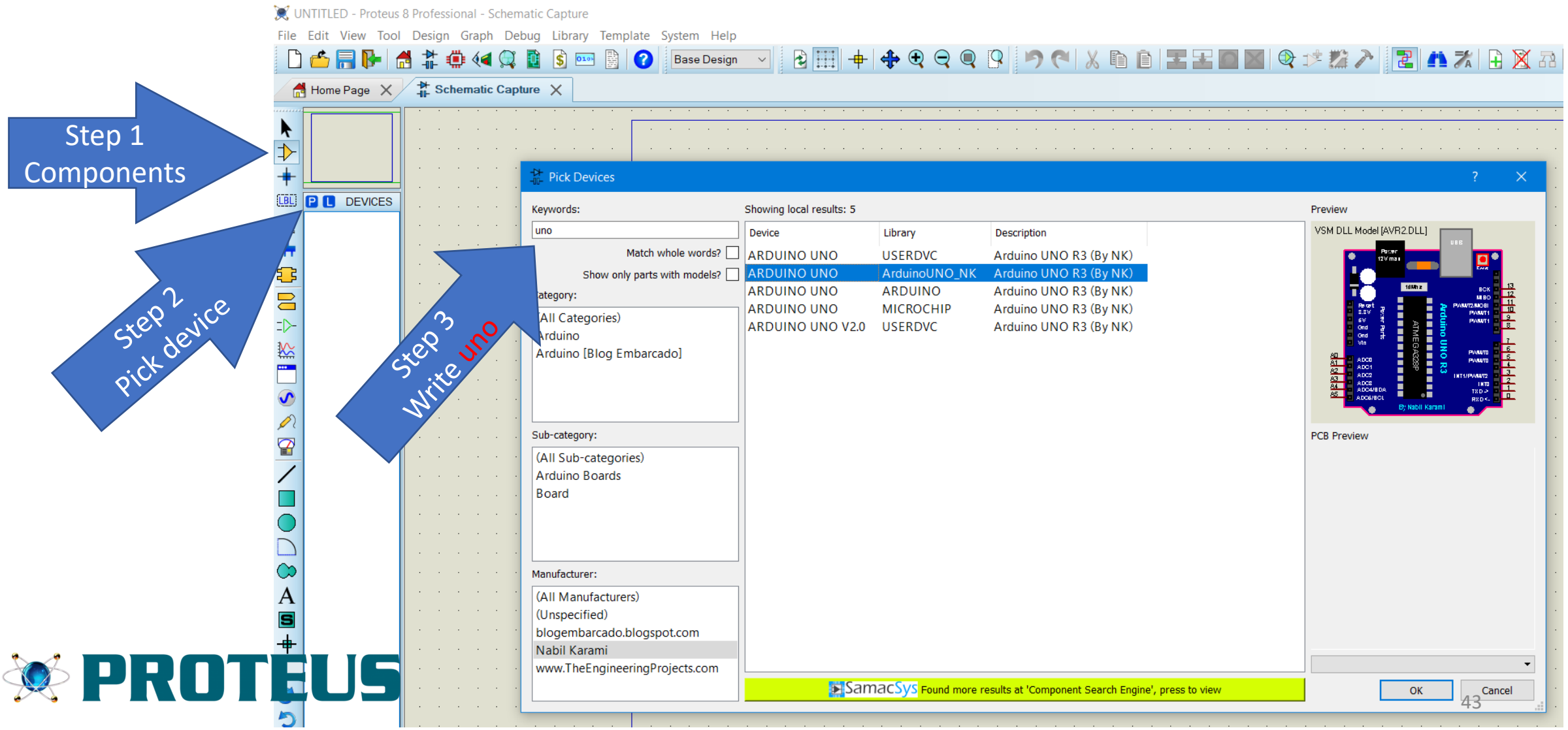




PROTEUS

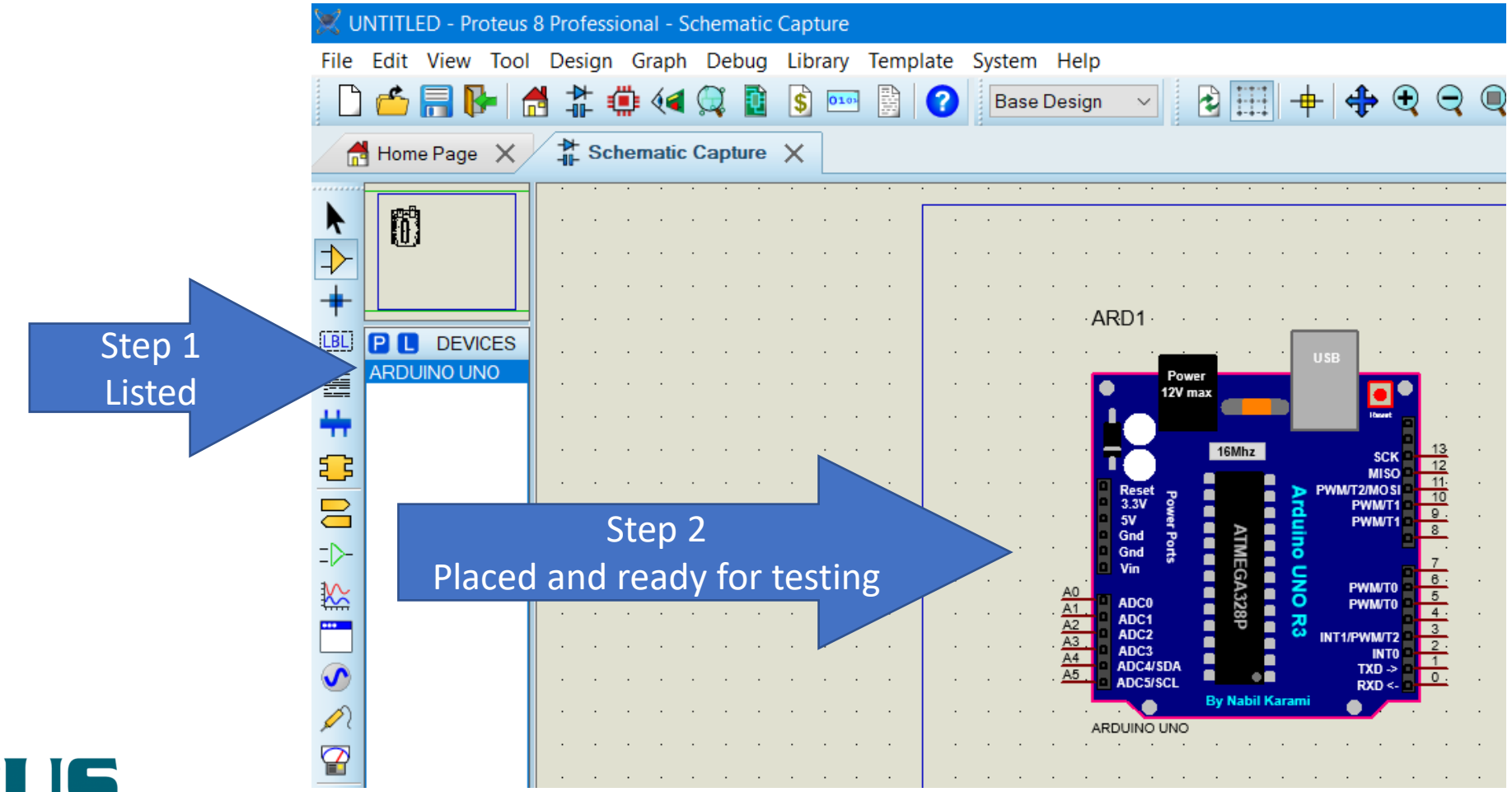


Working With Simulators (Proteus)





Working With Simulators (Proteus)



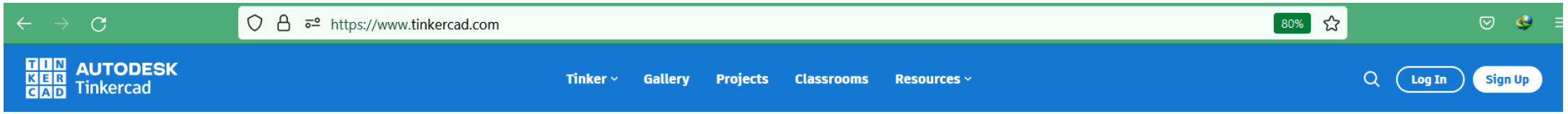
PROTEUS



Working With Simulators (Tinkercad)

To simulate Arduino on Tinkercad,

Step 1: Create account



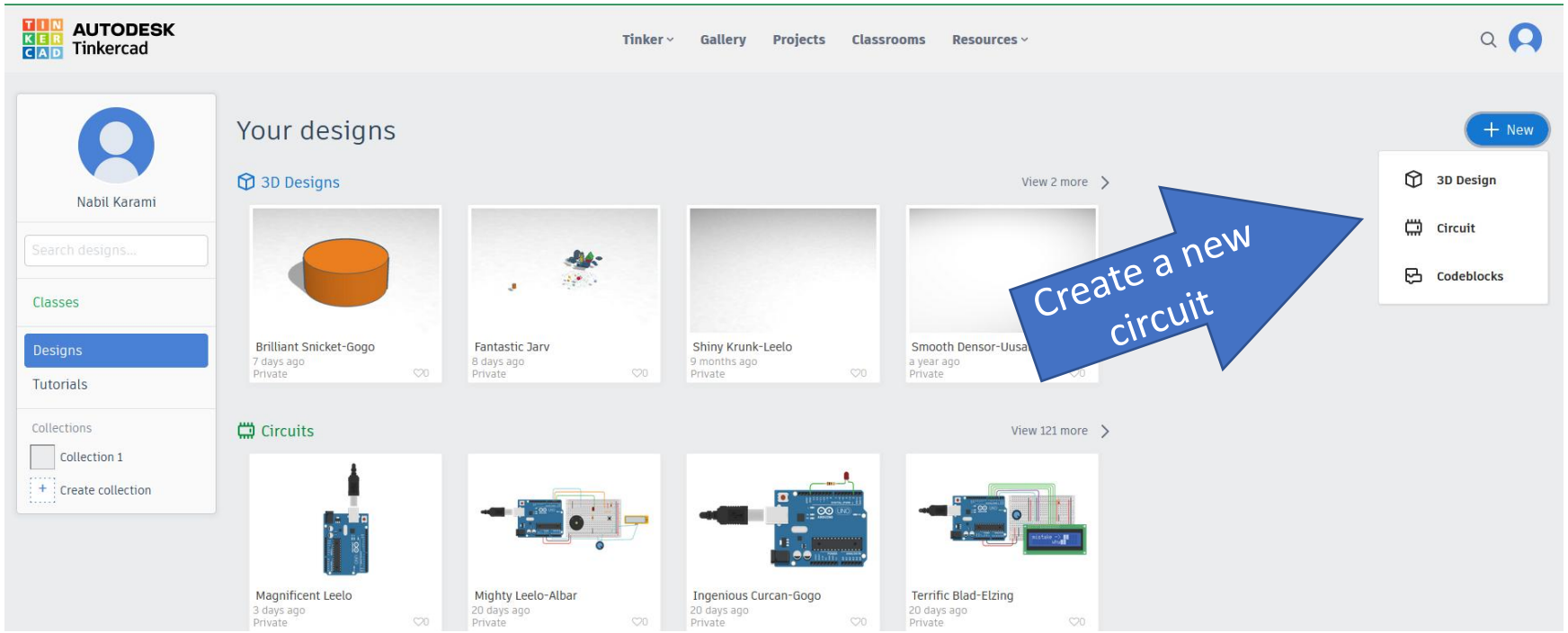
Step 1
Sign up

AUTODESK Tinkercad



Working With Simulators (Tinkercad)

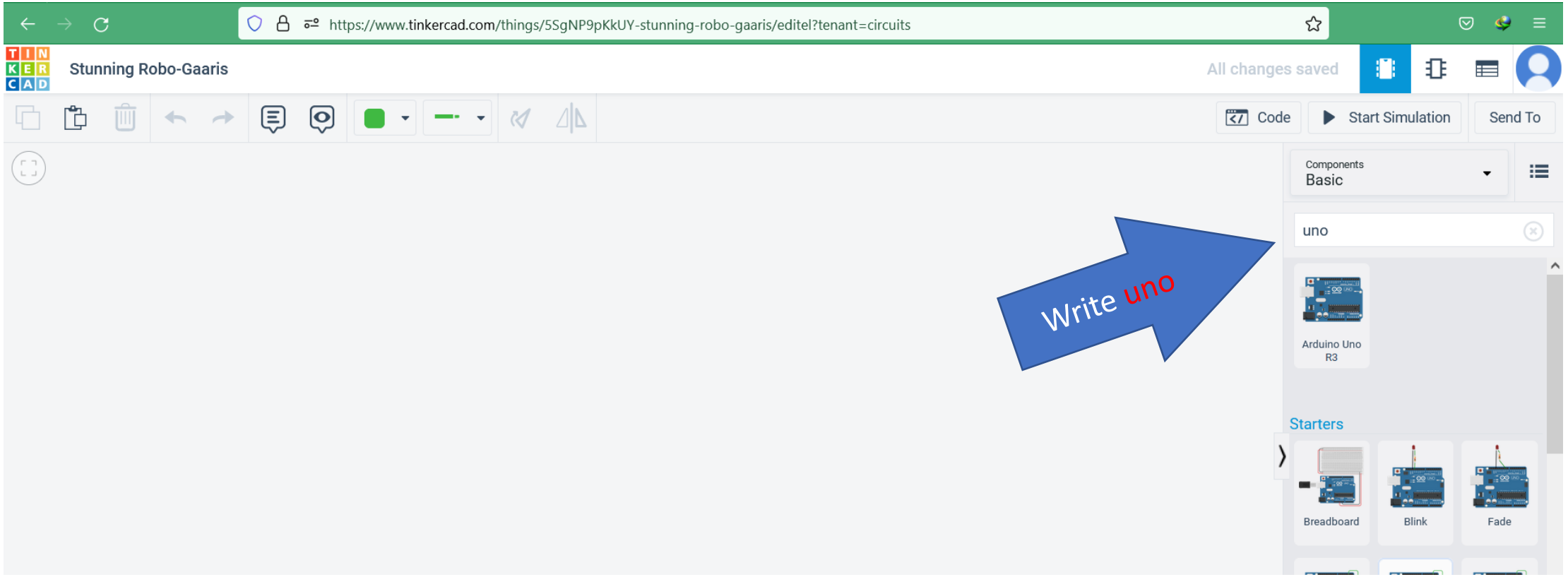
Step 2: Create a new circuit





Working With Simulators (Tinkercad)

Step 3: Start placing components





Week 2

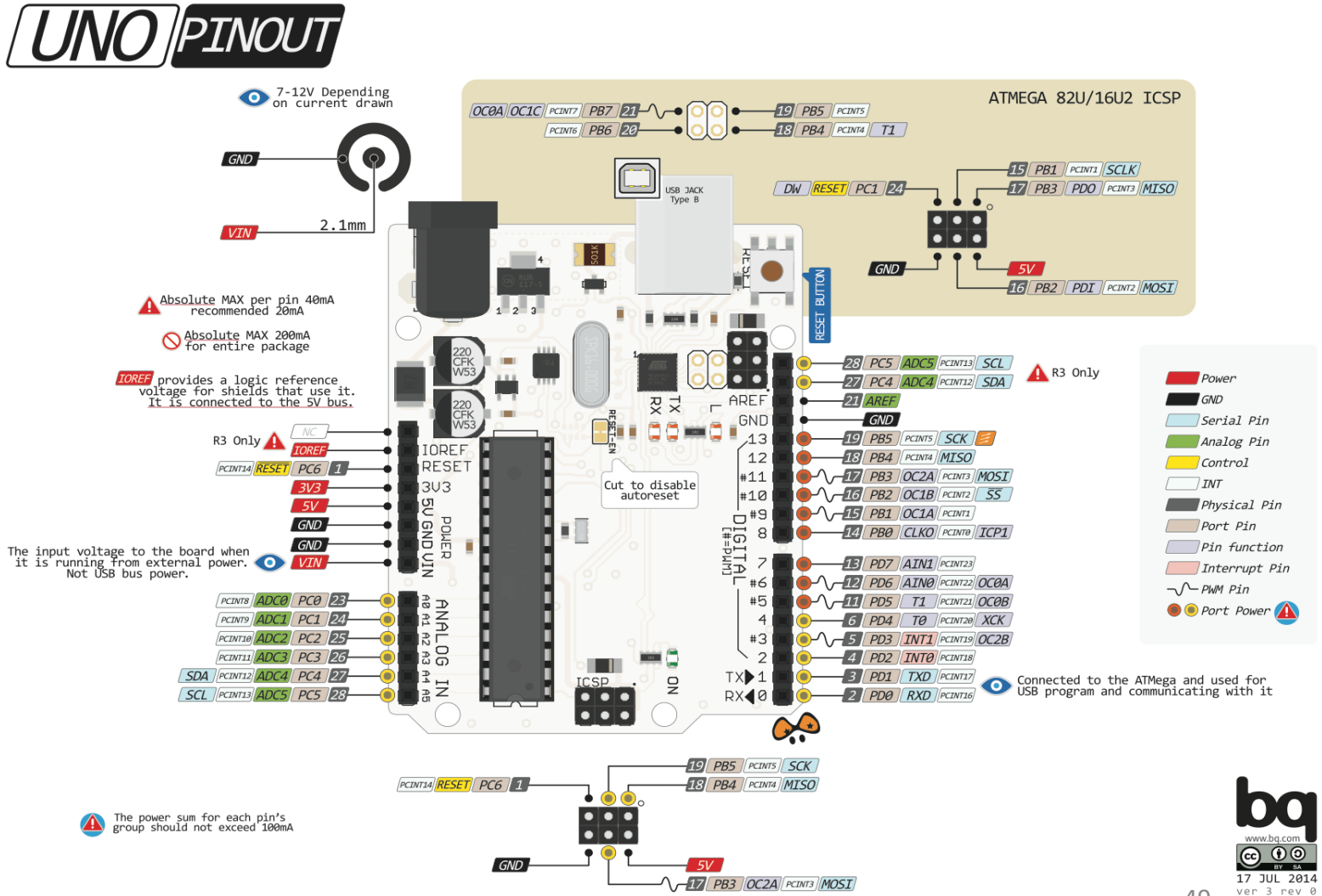
Lecture 2 / LO1

Introduction to Arduino UNO



Arduino UNO pinout

- The Arduino Uno pinout consists of
  - 14 digital pins,
  - 6 analog inputs,
  - a power jack,
  - USB connection
  - ICSP header.
- The versatility of the pinout provides many different options such as driving motors, LEDs, reading sensors and more.



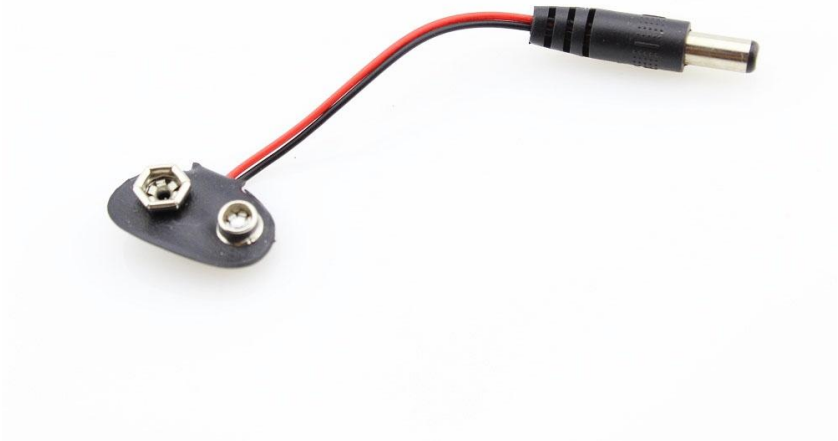
www.bq.com
17 JUL 2014
ver 3 rev 0

Arduino UNO pinout- Power Supply



There are 4 ways to power the Arduino Uno:

1: Barrel Jack - The Barrel jack, or DC Power Jack can be used to power your Arduino board. The barrel jack is usually connected to a wall adapter. The board can be powered by 5-20 volts but the manufacturer recommends to keep it between 7-12 volts. Above 12 volts, the regulators might overheat, and below 7 volts, might not suffice.

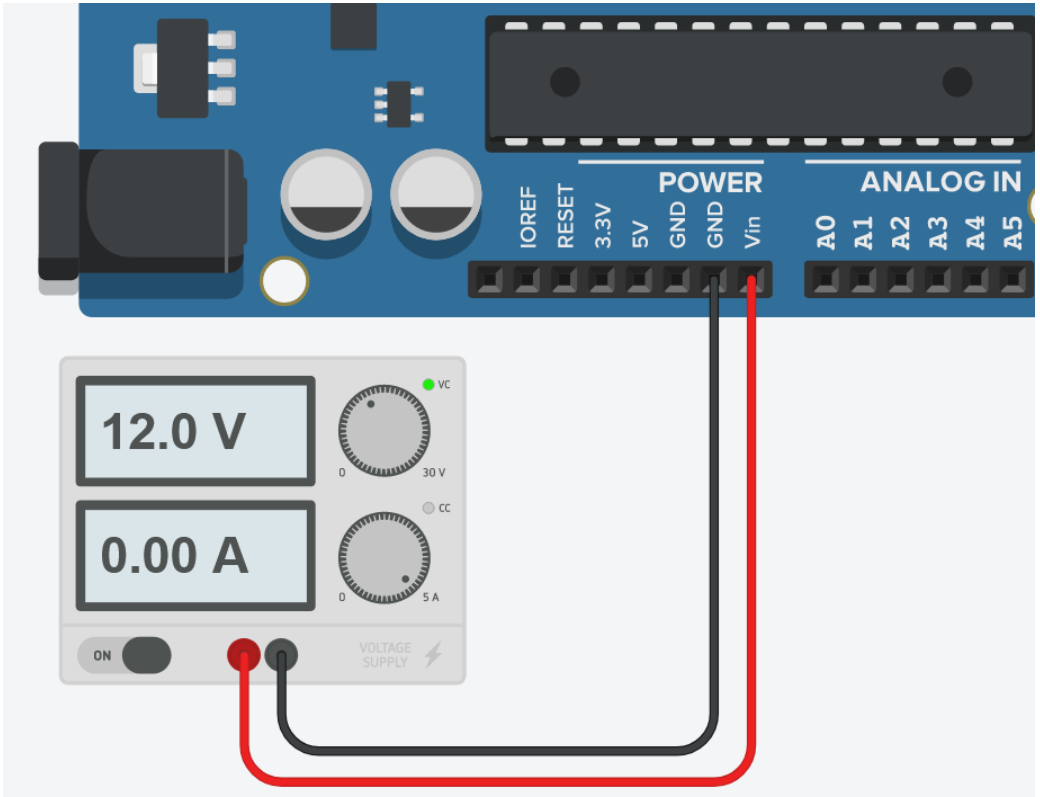






Arduino UNO pinout- Power Supply

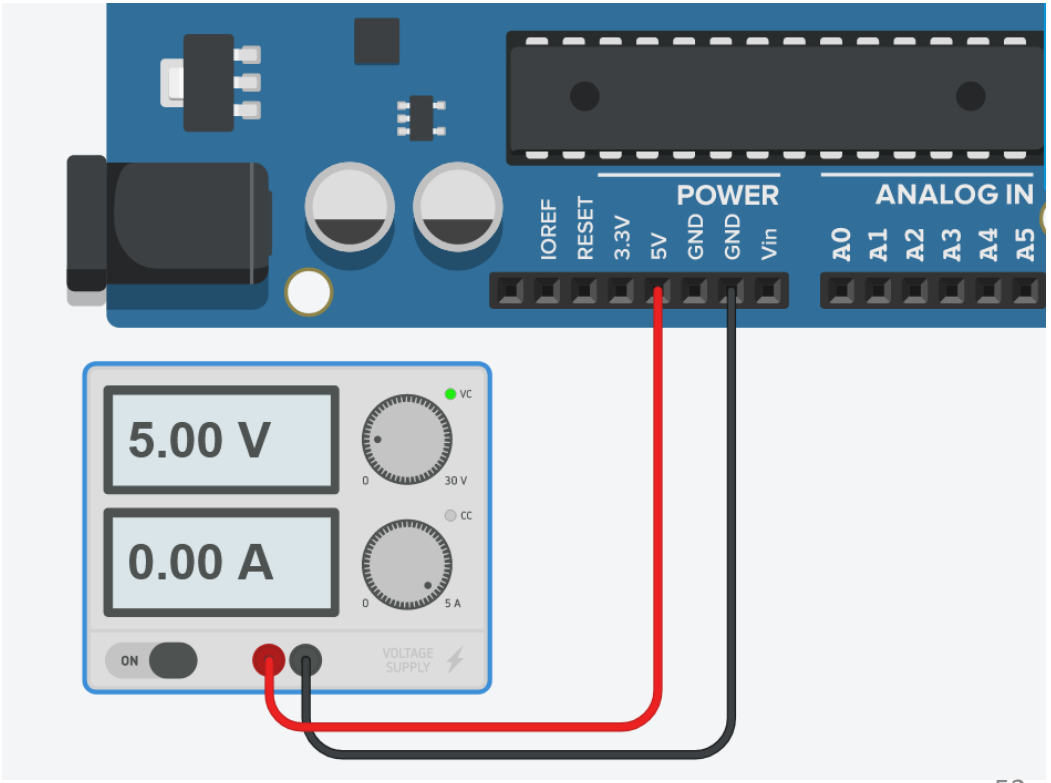
2: VIN Pin - This pin is used to power the Arduino Uno board using an external power source. The voltage should be within the range mentioned above.





Arduino UNO pinout- Power Supply

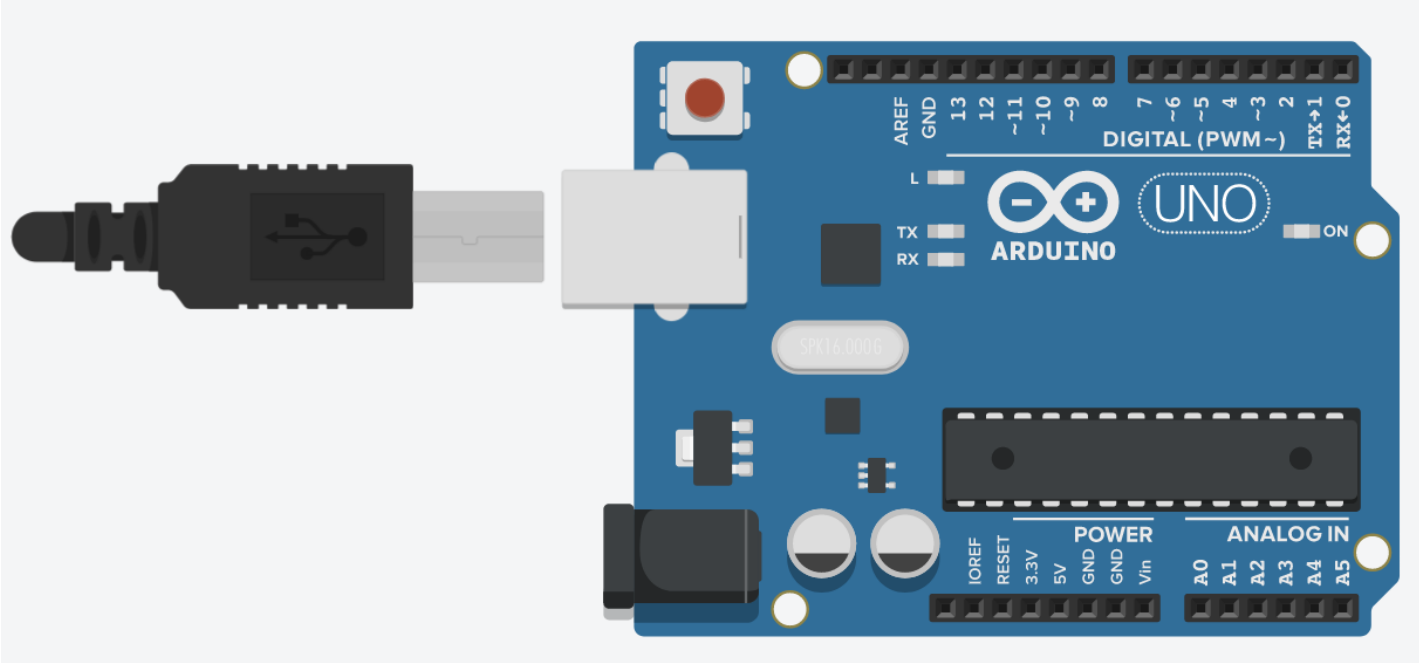
3: 5V Pin - This pin is used to power the Arduino Uno board using an external power source of **5 volt**.





Arduino UNO pinout- Power Supply

4: USB cable - when connected to the computer, provides 5 volts at 500mA.

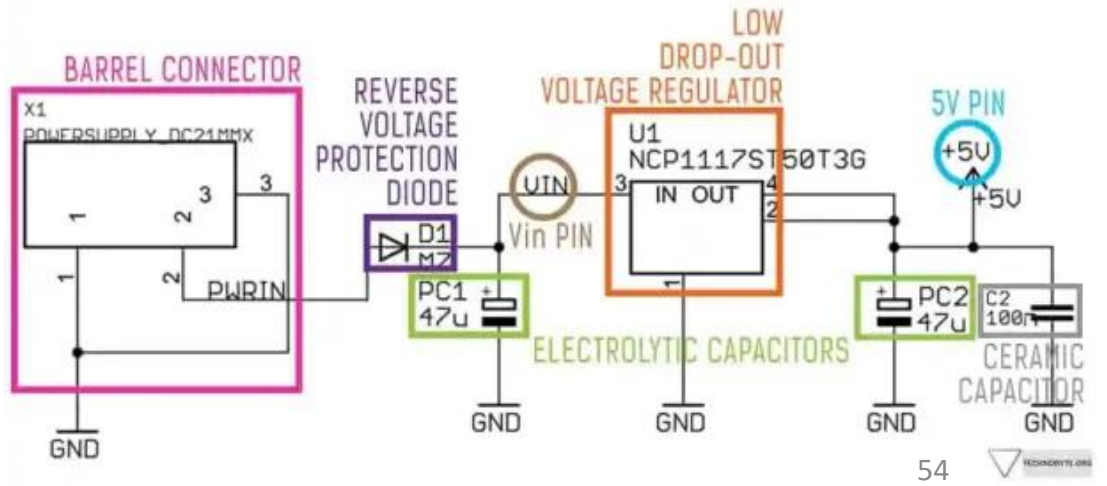




Arduino Uno pinout - Power Supply

- There is a polarity protection diode connecting between the positive of the barrel jack to the VIN pin, rated at 1 Ampere.
- The power source you use determines the power you have available for your circuit.
- For instance, powering the circuit using the USB limits you to 500mA.
- When powering your circuit through the barrel jack or VIN, the maximum capacity available is determined by the 5 and 3.3 volts regulators on-board the Arduino.







Arduino Uno pinout - Power Supply

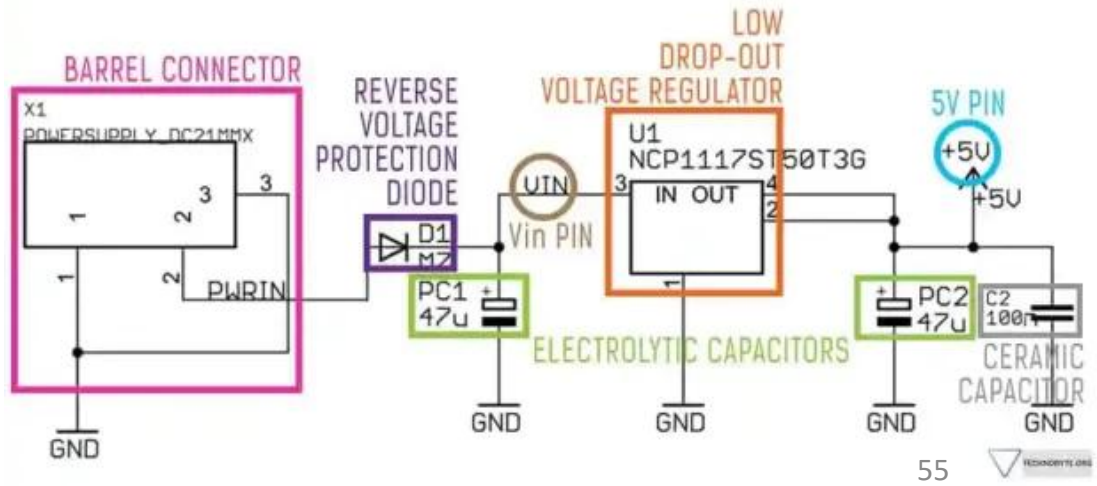
5v and 3v3

- They provide regulated 5 and 3.3v to power external components according to manufacturer specifications.

GND

- In the Arduino Uno pinout, you can find 5 GND pins, which are all interconnected.
- The GND pins are used to close the electrical circuit and provide a common logic reference level throughout your circuit. Always make sure that all GNDs (of the Arduino, peripherals and components) are connected to one another and have a common ground.







Arduino Uno pinout - Reset

- **RESET** - resets the Arduino. It can be provided by the on board push button or through the rest pin.

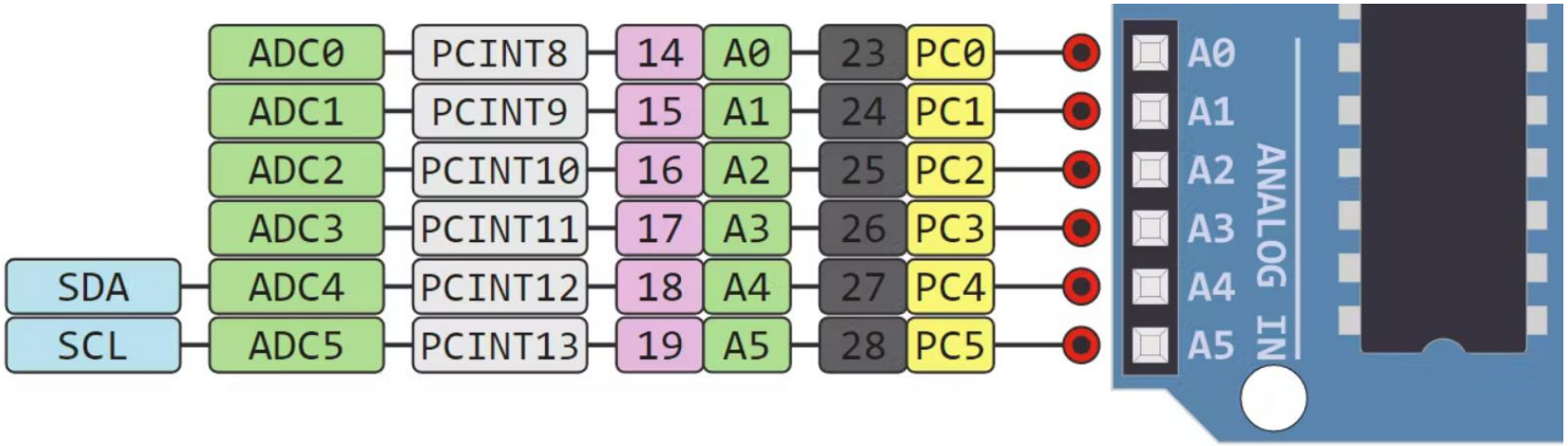
- **IOREF** - This pin is the input/output reference. It provides the voltage reference with which the microcontroller operates.





Arduino Uno Pinout - Analog IN

- The Arduino Uno has 6 **analog pins**, which utilize ADC (Analog to Digital converter).
- These pins serve as analog inputs but can also function as digital inputs or digital outputs.

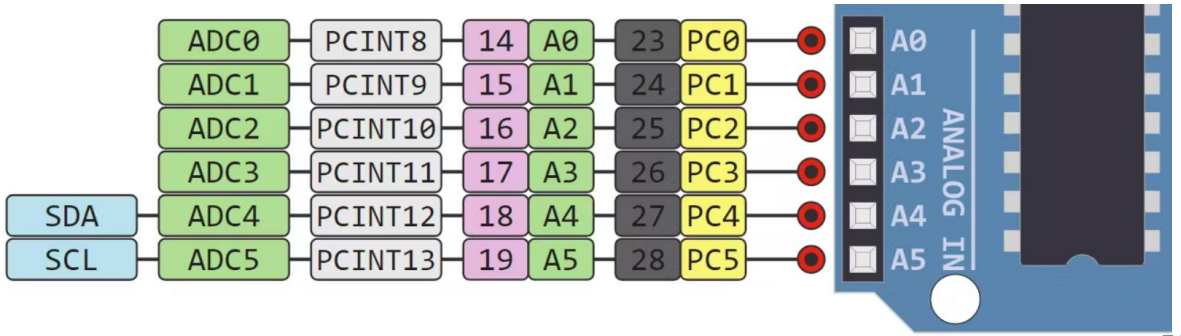




Arduino Uno Pinout - Analog IN

Analog to Digital Conversion

- ADC stands for Analog to Digital Converter.
- ADC is an electronic circuit used to convert analog signals into digital signals.
- This digital representation of analog signals allows the processor (which is a digital device) to measure the analog signal and use it through its operation.
- On Arduino the ADC has 10-bit resolution, meaning it can represent analog voltage by 1,024 digital levels.

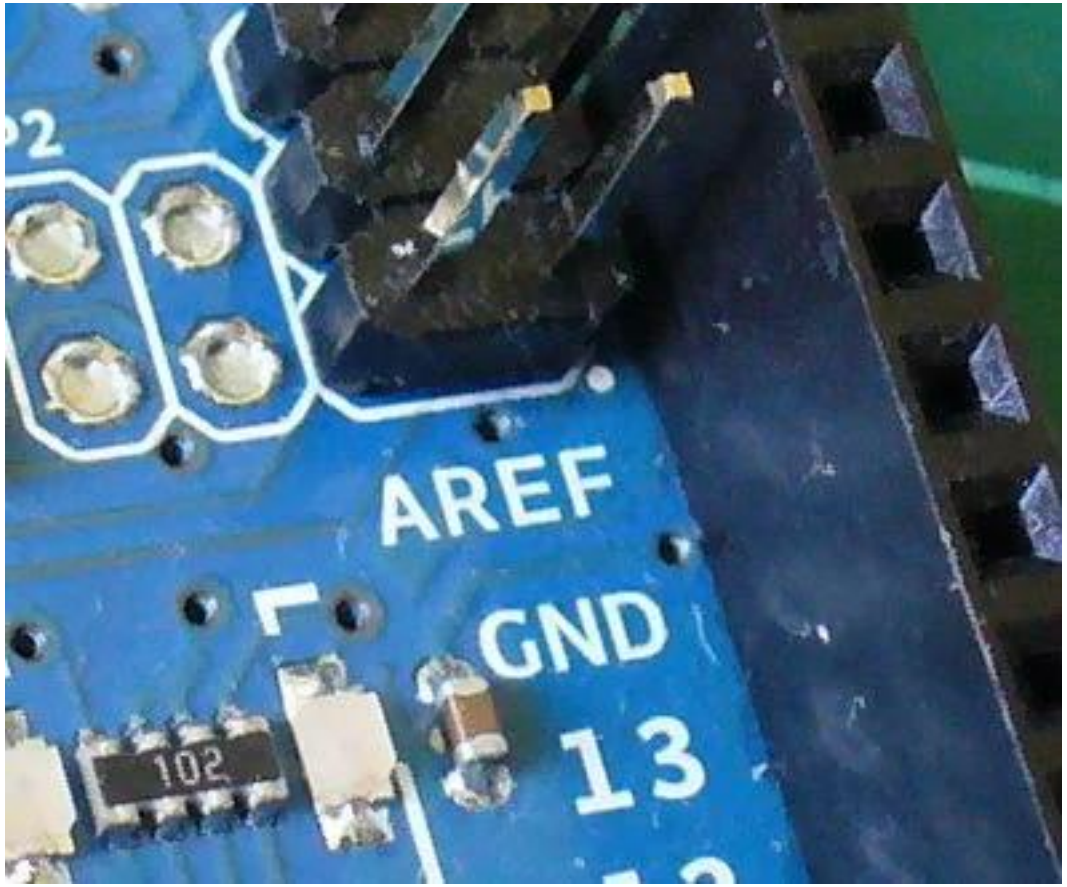




Arduino Uno Pinout - Analog Reference

Aref - Reference voltage for the analog inputs.

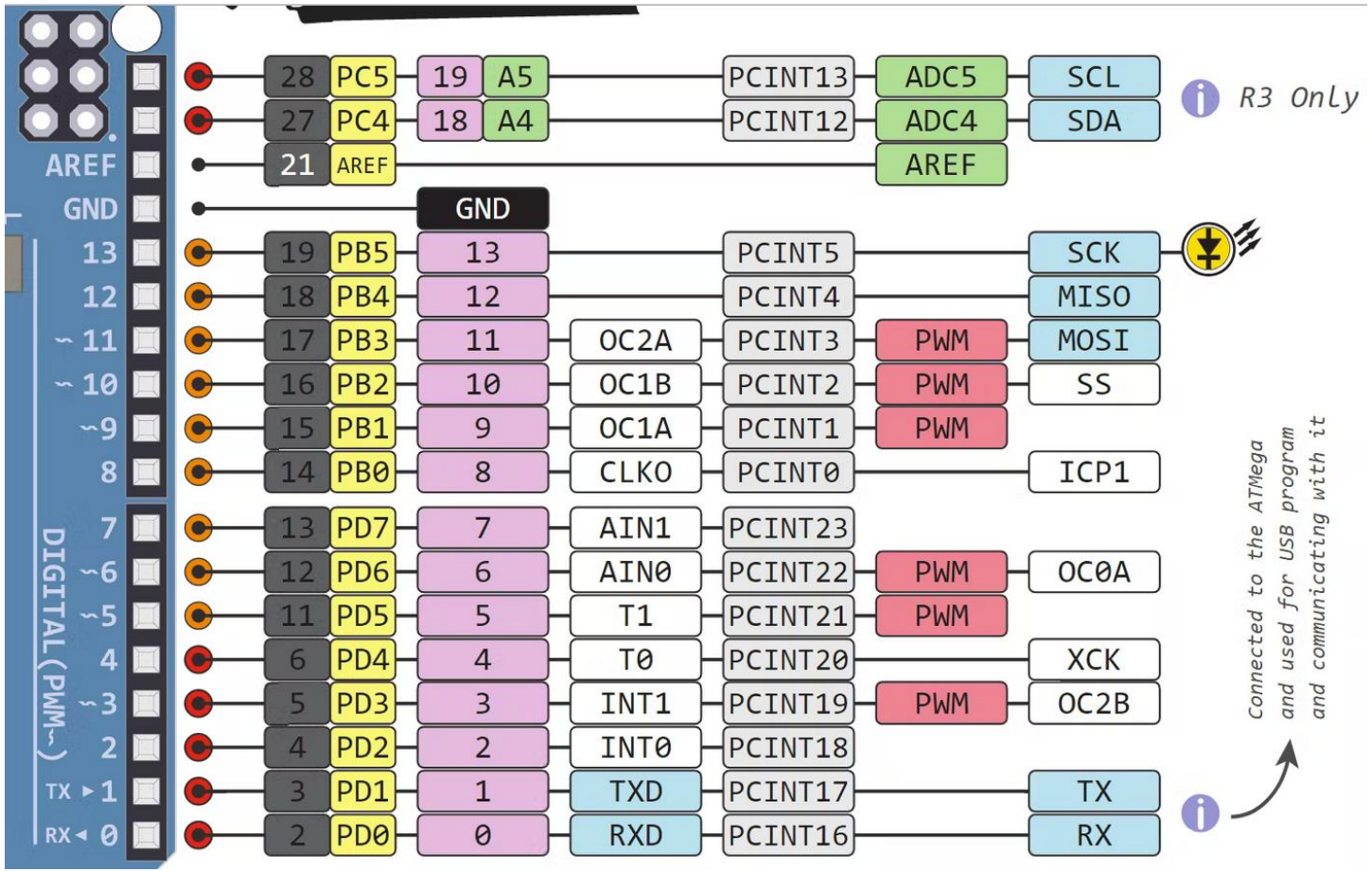
- It allows us to feed the Arduino a reference voltage from an external power supply in order to configure the reference voltage used for analog input (i.e. the value used as the top of the input range).
- For example, if we want to measure voltages with a maximum range of 3.3V, we would feed a nice smooth 3.3V into the AREF pin





Arduino Uno Pinout - Digital Pins

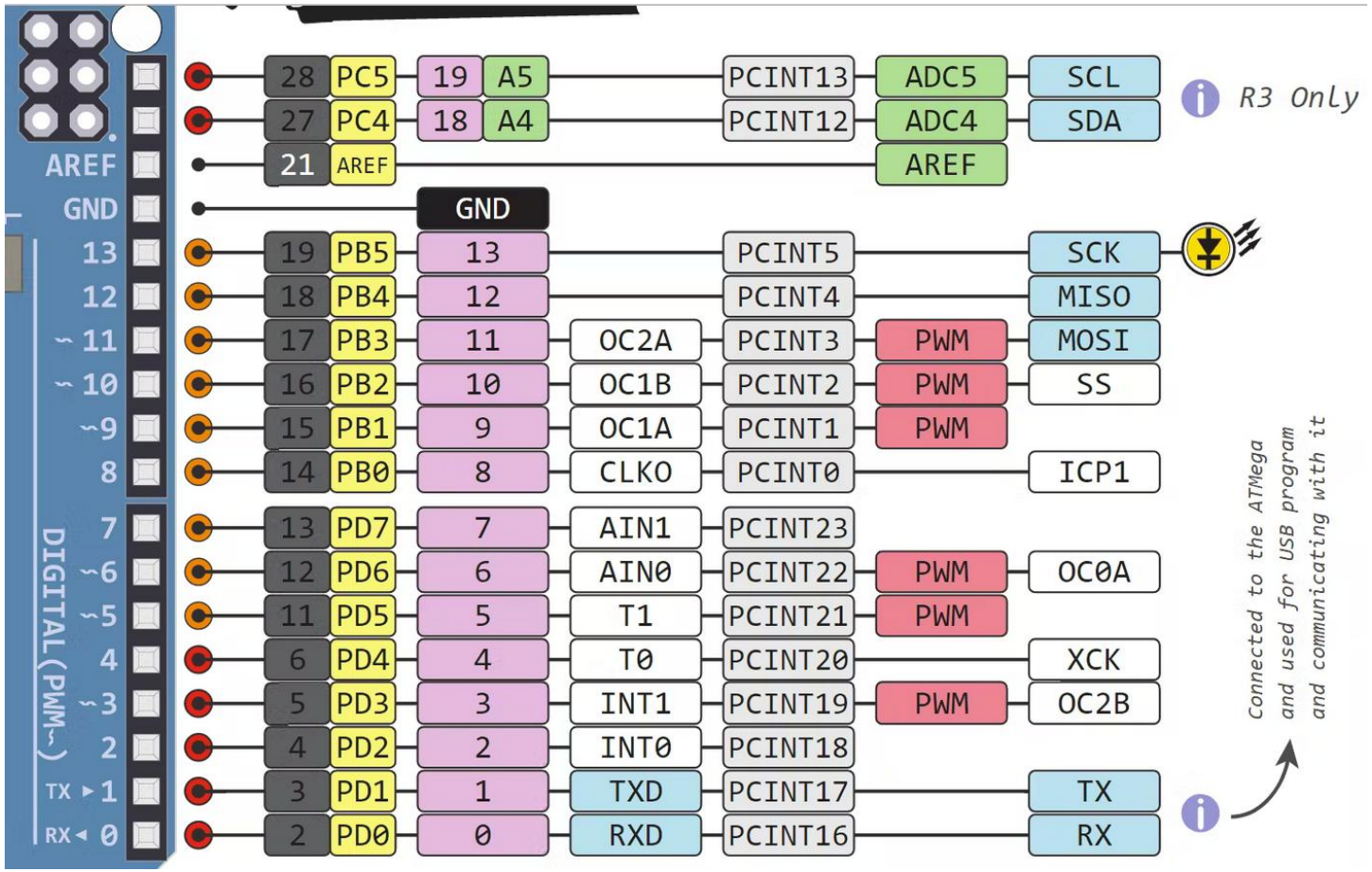
- Pins 0-13 of the Arduino Uno serve as digital input/output pins.
- Pin 13 of the Arduino Uno is connected to the built-in LED.
- In the Arduino Uno - pins 3,5,6,9,10,11 have PWM capability.





Arduino Uno Pinout - Digital Pins

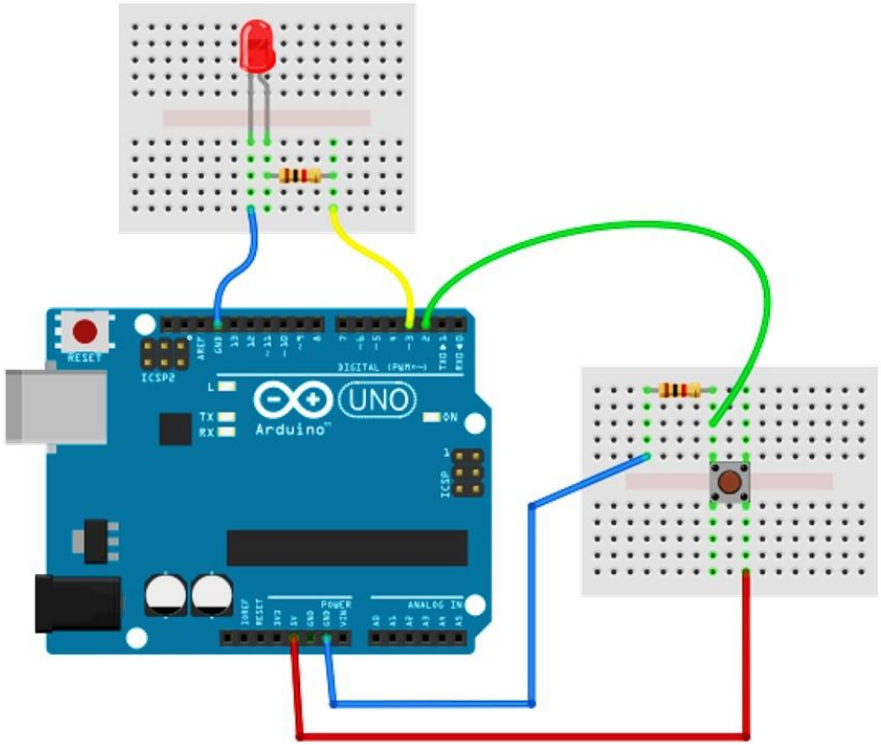
- Each pin can provide/sink up to 40 mA max. But the recommended current is 20 mA.
- The absolute max current provided (or sank) from all pins together is 200mA





What does digital mean?

- Digital is a way of representing voltage in 1 bit: either 0 or 1.
- Digital pins on the Arduino are pins designed to be configured as inputs or outputs according to the needs of the user.
- Digital pins are either on or off. When ON they are in a HIGH voltage state of 5V and when OFF they are in a LOW voltage state of 0V.
- On the Arduino, When the digital pins are configured as **output**, they are set to 0 or 5 volts.



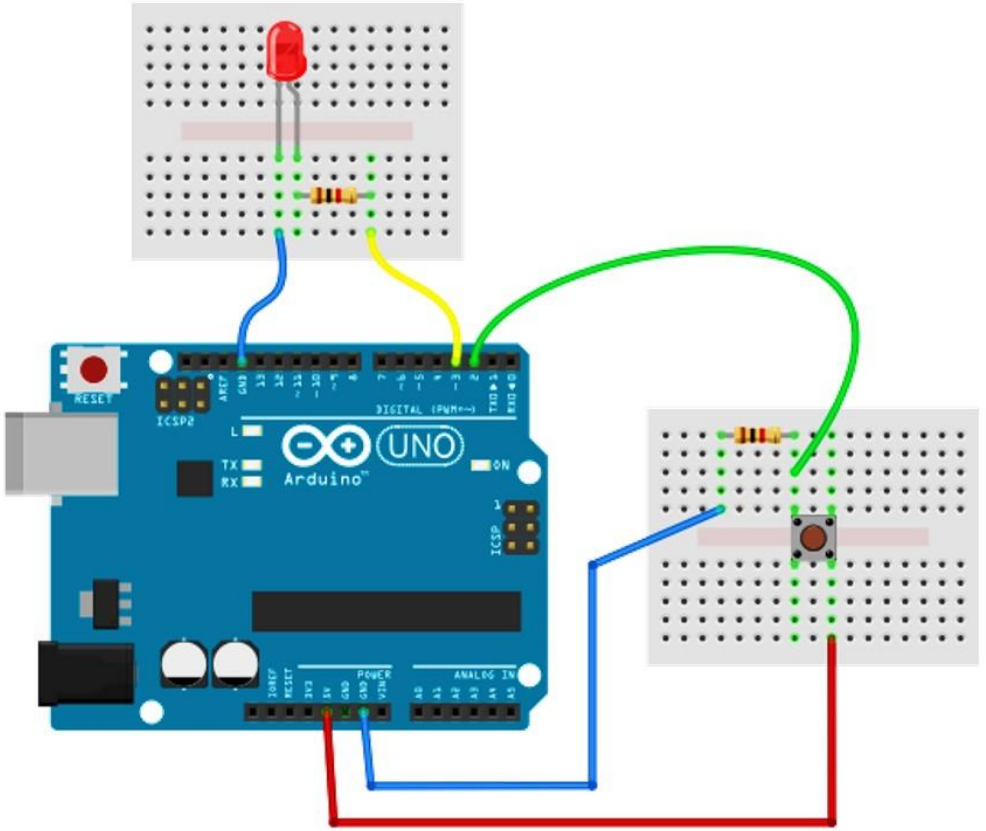


What does digital mean?

- When the digital pins are configured as **input,** the voltage is supplied from an external device.
- This voltage can vary between 0-5 volts which is converted into digital representation (0 or 1).

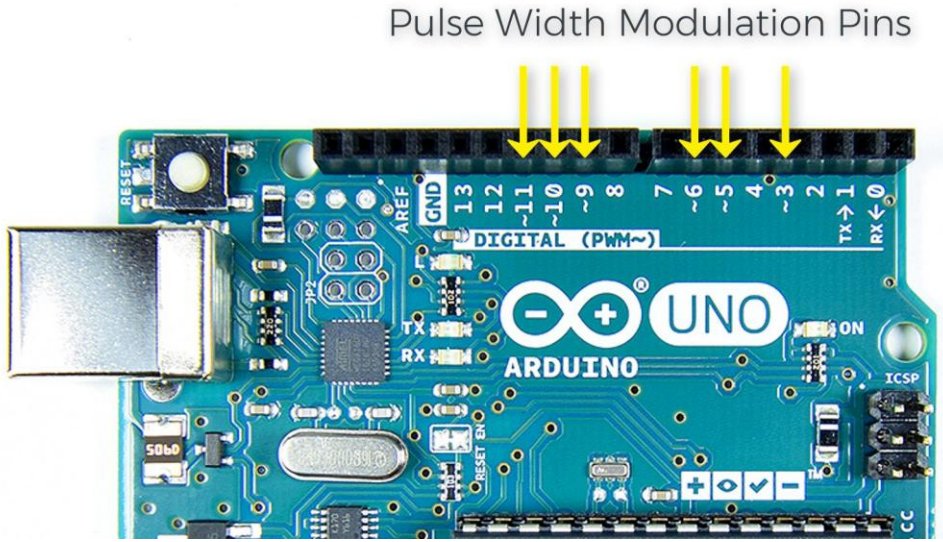
To determine this, there are 2 thresholds:

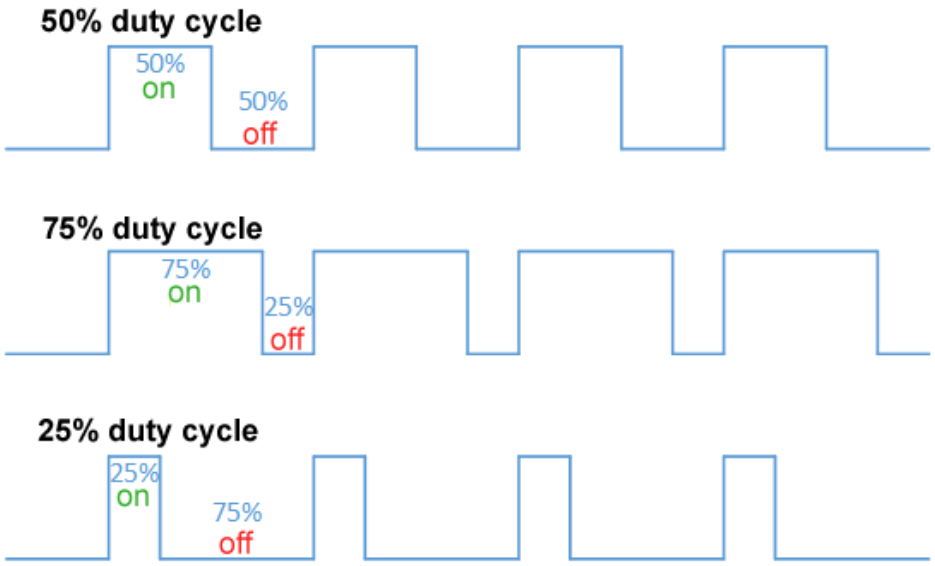
- Below 0.8v - considered as 0.
- Above 2v - considered as 1.



PWM channels

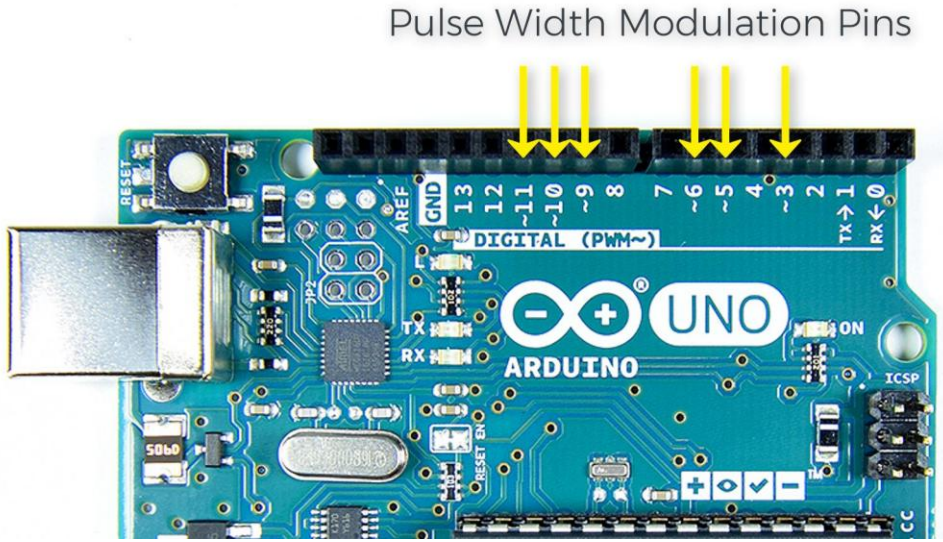
- In general, Pulse Width Modulation (PWM) is a modulation technique used to encode a message into a pulsing signal.
- Also it is used to drive motors at different speed through special drivers.
- A PWM is comprised of two key components: **frequency** and **duty cycle**.

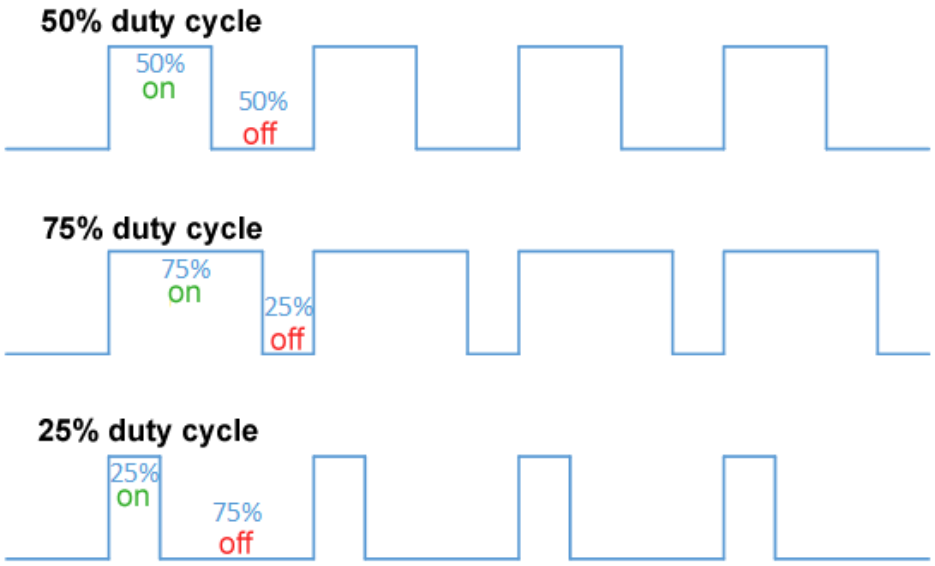




PWM channels

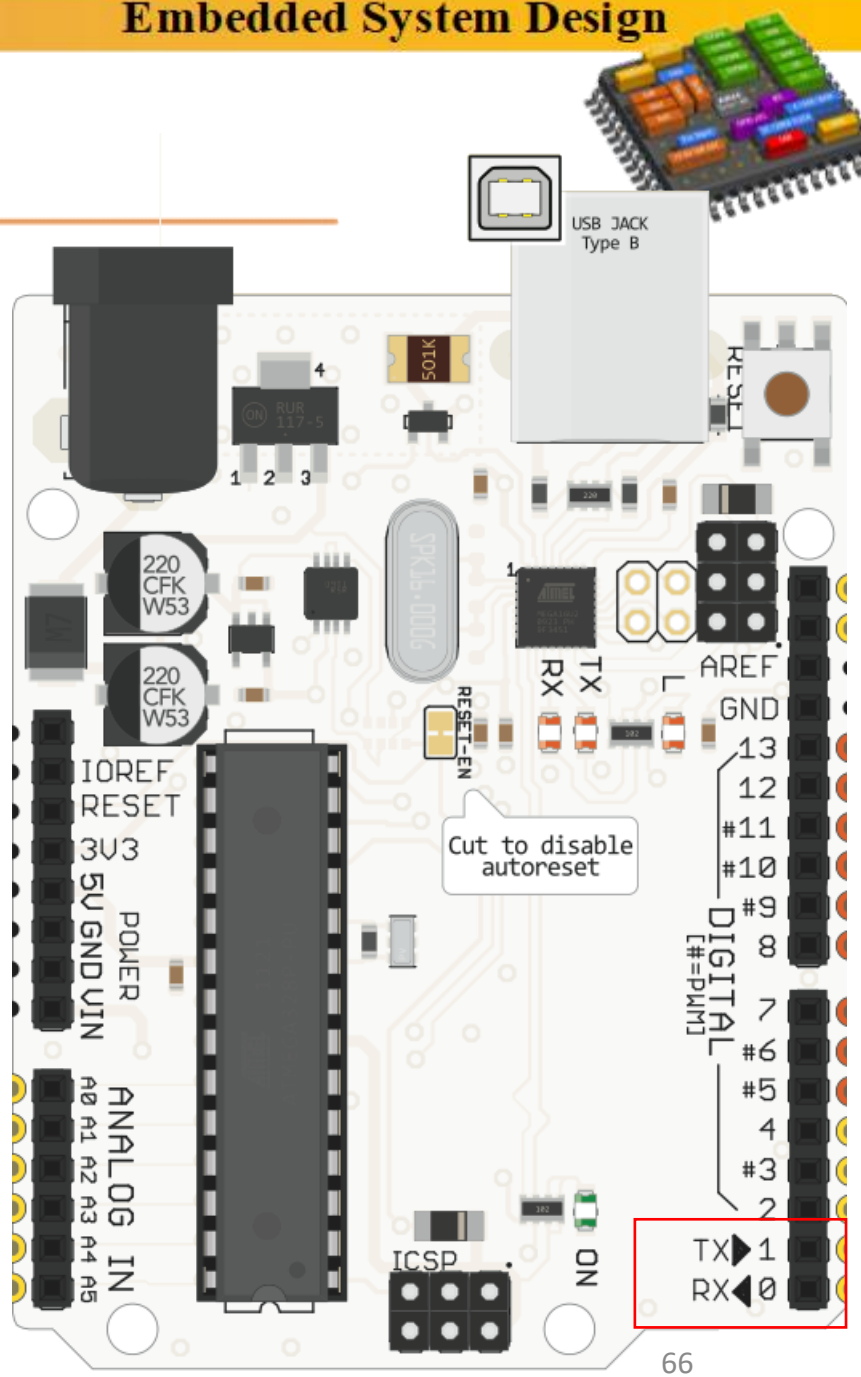
- In Arduino, the PWM enabled pins produce a constant frequency of ~ 490Hz and ~ 976 (depending on the channel), while the duty cycle changes according to the parameters set by the user.





Communication Protocols

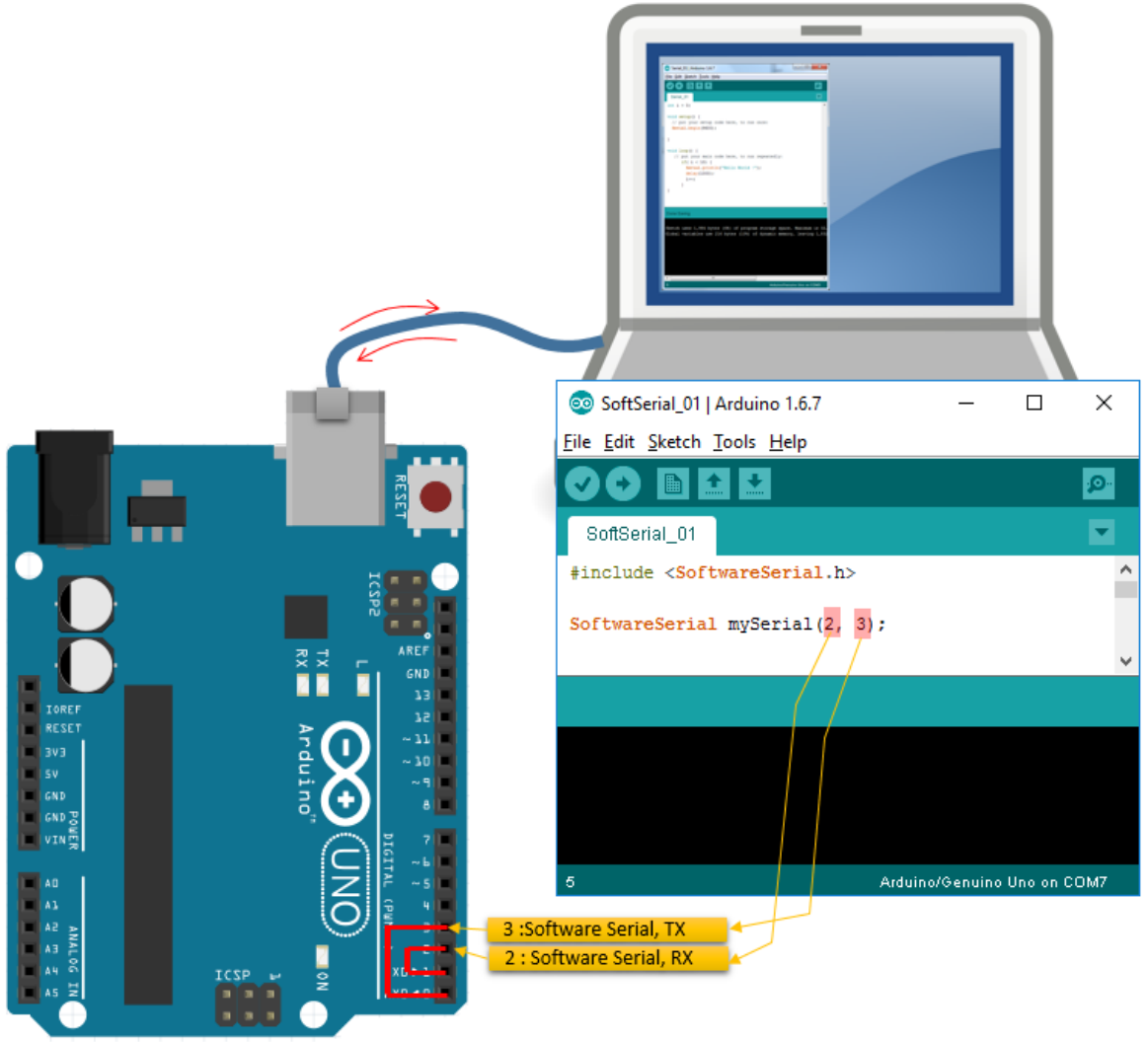
- **Serial (TTL) -** Digital pins 0 and 1 are the serial pins of the Arduino Uno.
- They are used by the onboard USB module.
- Serial communication is used to exchange data between the Arduino board and another serial device such as computers, displays, sensors and more.
- Each Arduino board has at least one serial port.
- Serial communication occurs on digital pins 0 (RX) and 1 (TX) as well as via USB.





Serial Communication Protocol

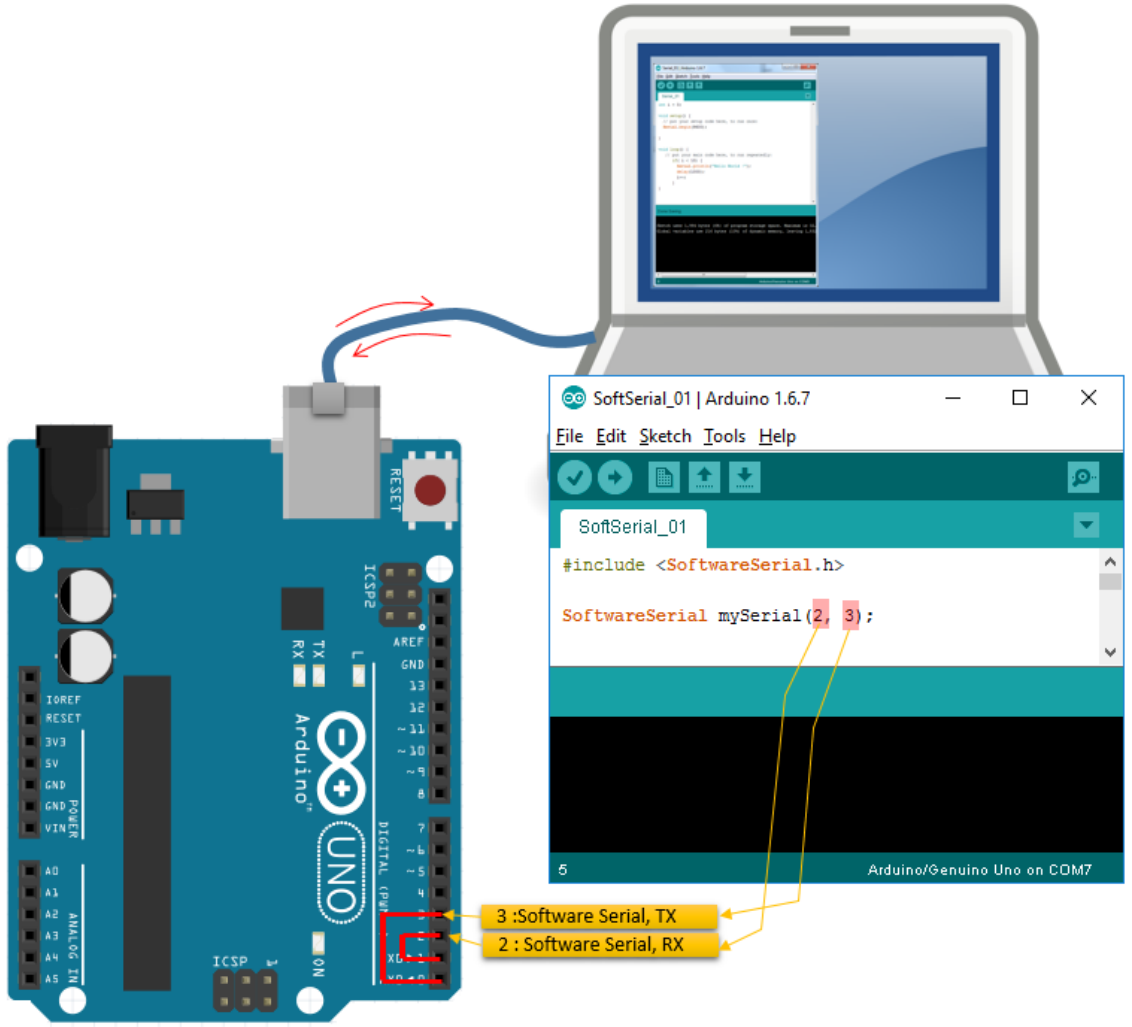
- Arduino supports serial communication through digital pins with the SoftwareSerial Library as well.
- This allows the user to connect multiple serial-enabled devices and leave the main serial port available for the USB.
- Software serial and hardware serial - Most microcontrollers have hardware designed to communicate with other serial devices.





Serial Communication Protocol

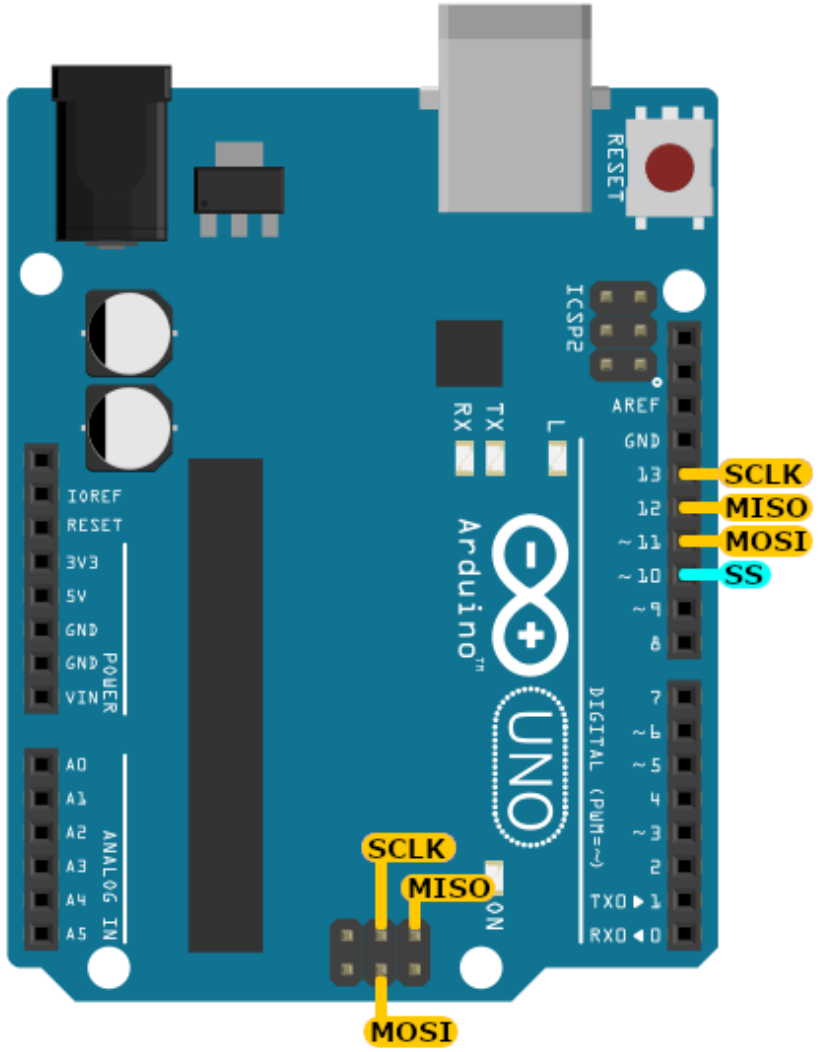
- Software serial ports use a pin-change interrupt system to communicate.
- There is a built-in library for Software Serial communication.
- Software serial is used by the processor to simulate extra serial ports.
- The only drawback with software serial is that it requires more processing and cannot support the same high speeds as hardware serial.





SPI Protocol

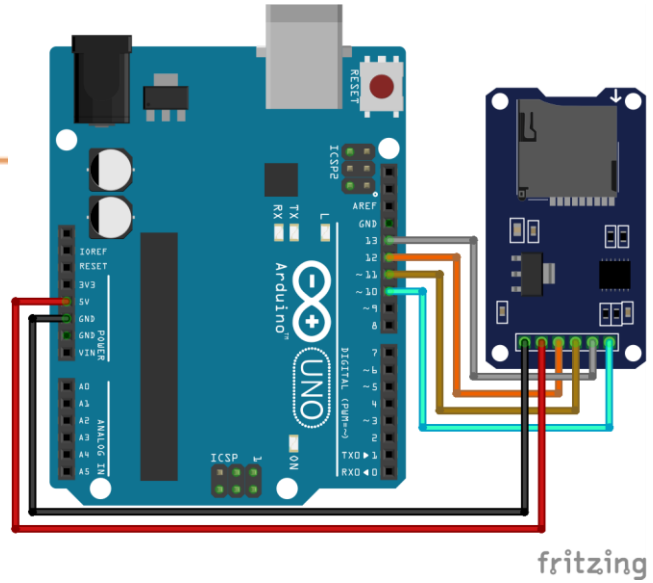
- **SPI -** SS/SCK/MISO/MOSI pins are the dedicated pins for SPI communication.
- They can be found on digital pins 10-13 of the Arduino Uno and on the ICSP headers.
  - MISO (Master In Slave Out) - A line for sendin data to the Master device
  - MOSI (Master Out Slave In) - The Master line for sending data to peripheral devices
  - SCK (Serial Clock) - A clock signal generated b the Master device to synchronize data transmission.

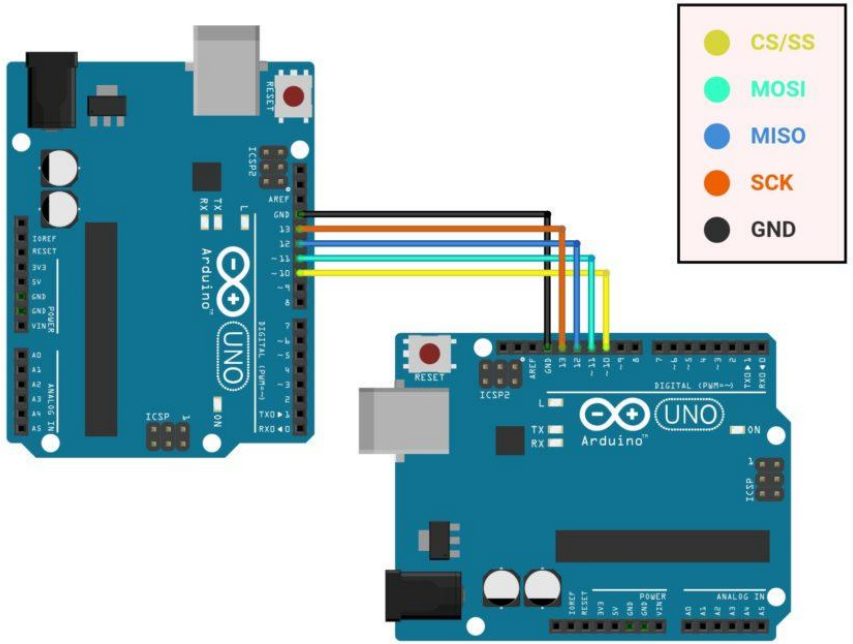




SPI Protocol

- Serial Peripheral Interface (SPI) is a serial data protocol used by microcontrollers to communicate with one or more external devices in a bus like connection.
- The SPI can also be used to connect 2 microcontrollers.

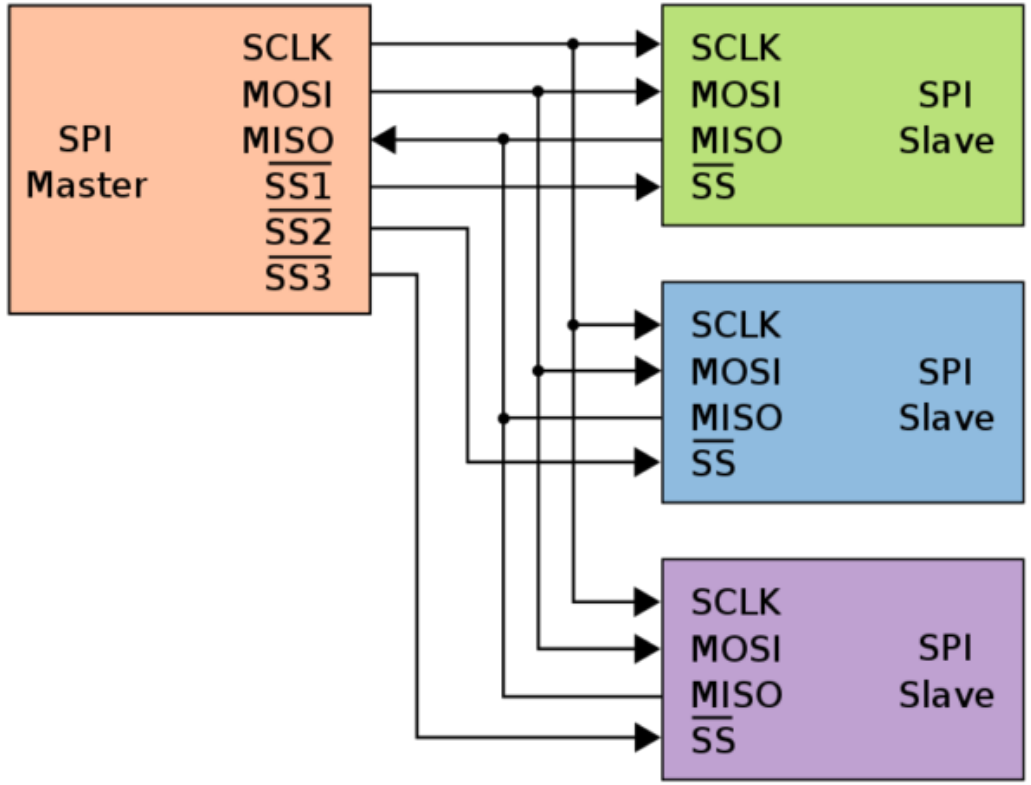






SPI Protocol

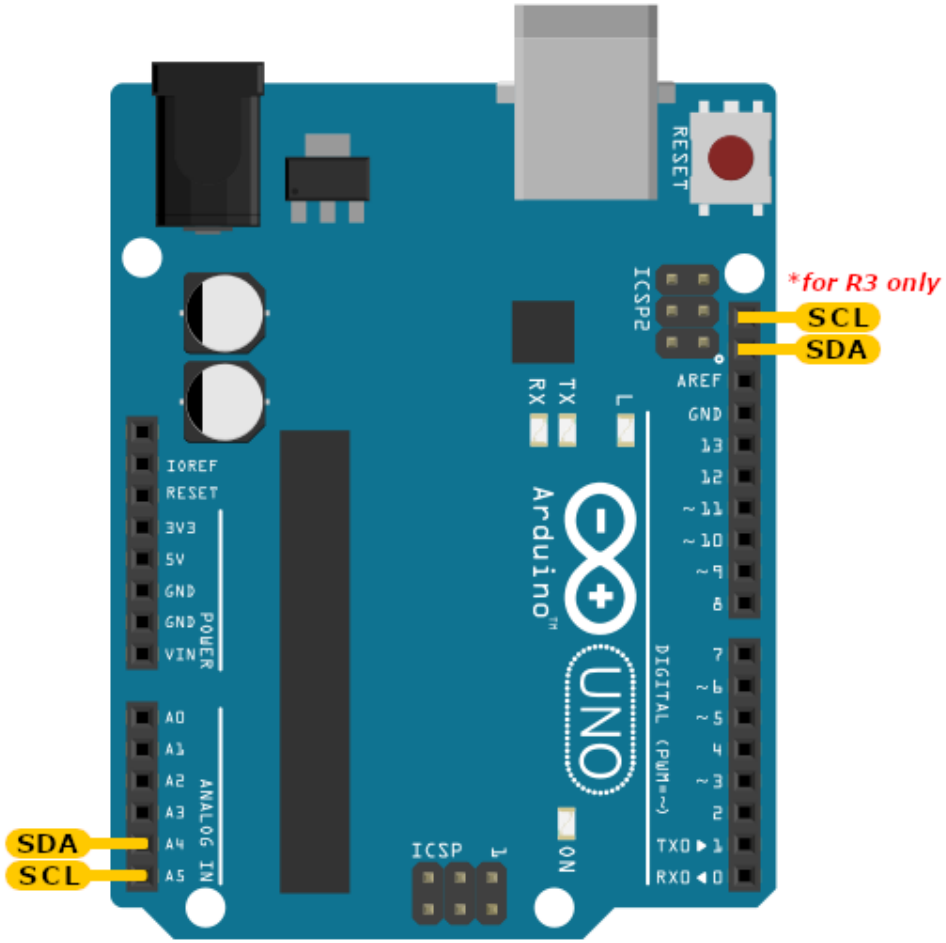
- On the SPI bus, there is always one device that is denoted as a Master device and all the rest as Slaves.
- In most cases, the microcontroller is the Master device.
- The SS (Slave Select) pin determines which device the Master is currently communicating with.





I2C Protocol

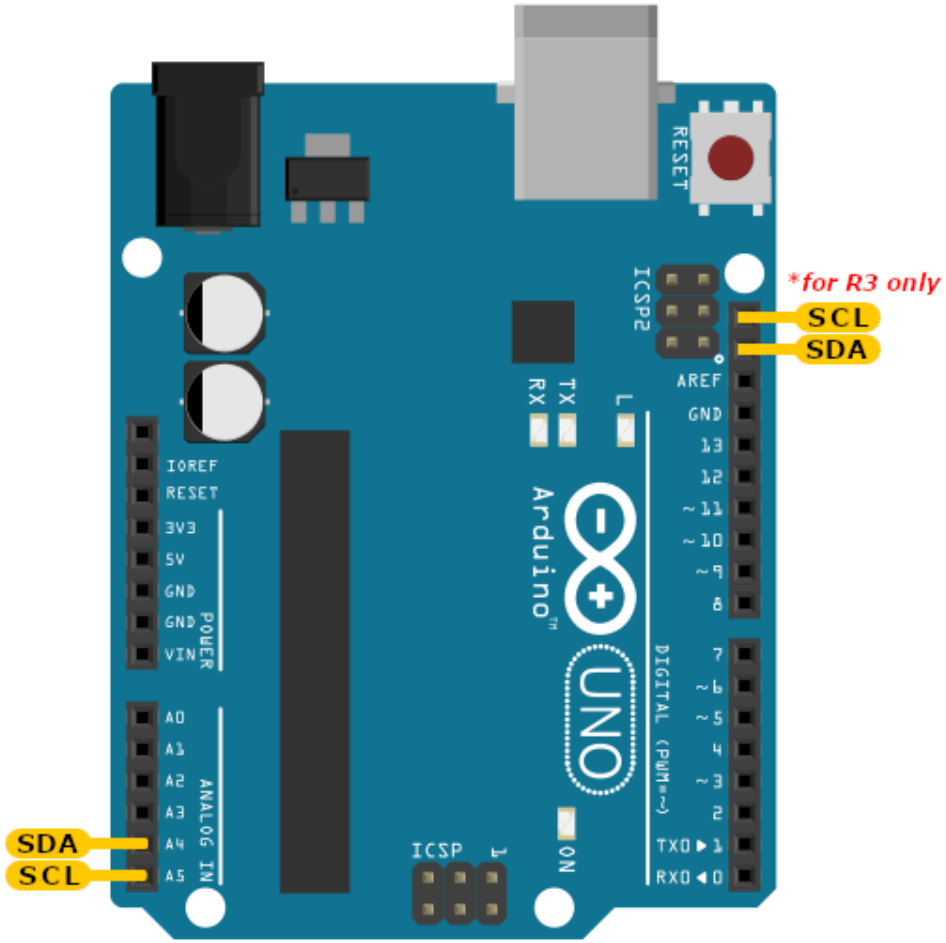
- **I2C -** SCL/SDA pins are the dedicated pins for I2C communication. On the Arduino Uno they are found on Analog pins A4 and A5.
- I2C is a communication protocol commonly referred to as the “I2C bus”.
- The I2C protocol was designed to enable communication between components on a single circuit board.





I2C Protocol

- With I2C there are 2 wires referred to as SCL and SDA.
- SCL is the clock line which is designed to synchronize data transfers.
- SDA is the line used to transmit data.
- Each device on the I2C bus has a unique address, up to 255 devices can be connected on the same bus.





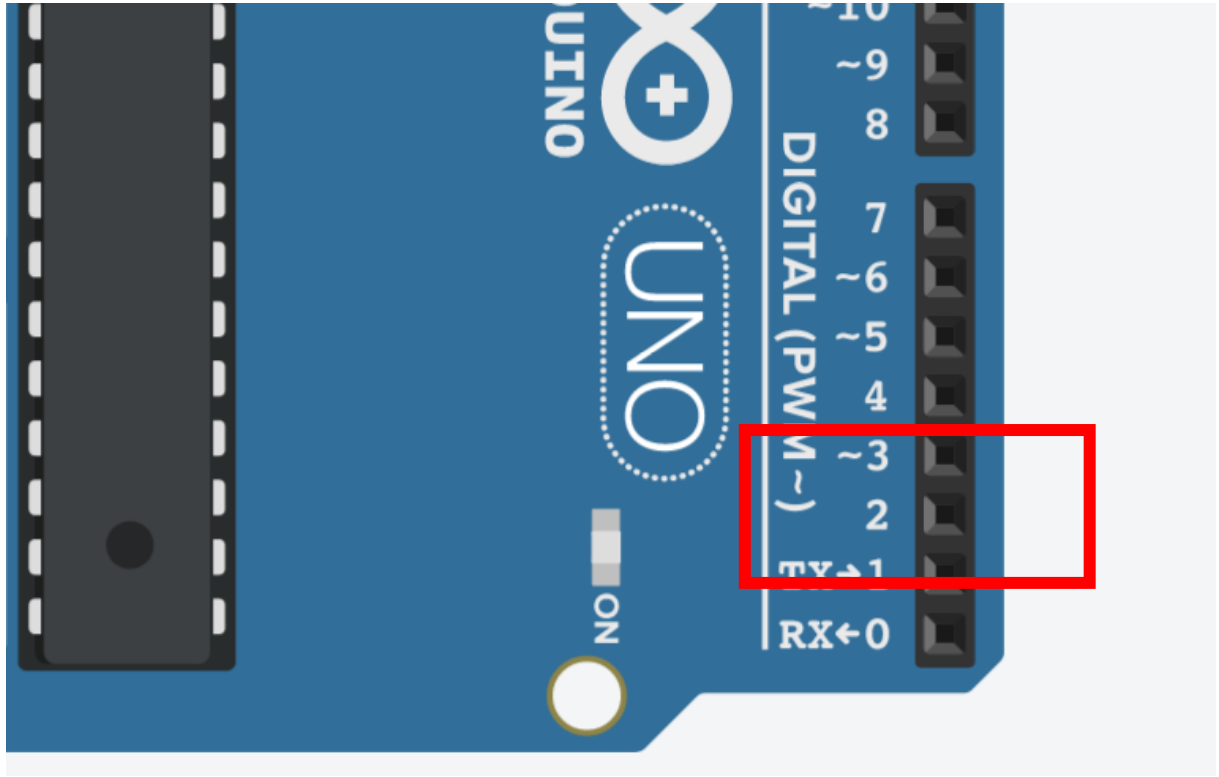
Additional pins/functions

- **Interrupt -** INT0 and INT1. Arduino Uno has two external interrupt pins.
- **External Interrupt -** An external interrupt is a system interrupt that occurs when outside interference is present. Interference can come from the user or other hardware devices in the network. Common uses for these interrupts in Arduino are reading the frequency a square wave generated by encoders or waking up the processor upon an external event.



Interrupt pins

- Arduino has two forms of interrupt:
- External
- Pin Change
- There are two external interrupt pins on the ATmega168/328 called INT0 and INT1.
- both INT0 and INT1 are mapped to pins 2 and 3.
- In contrast, Pin Change interrupts can be activated on any of the pins.



ICSP port



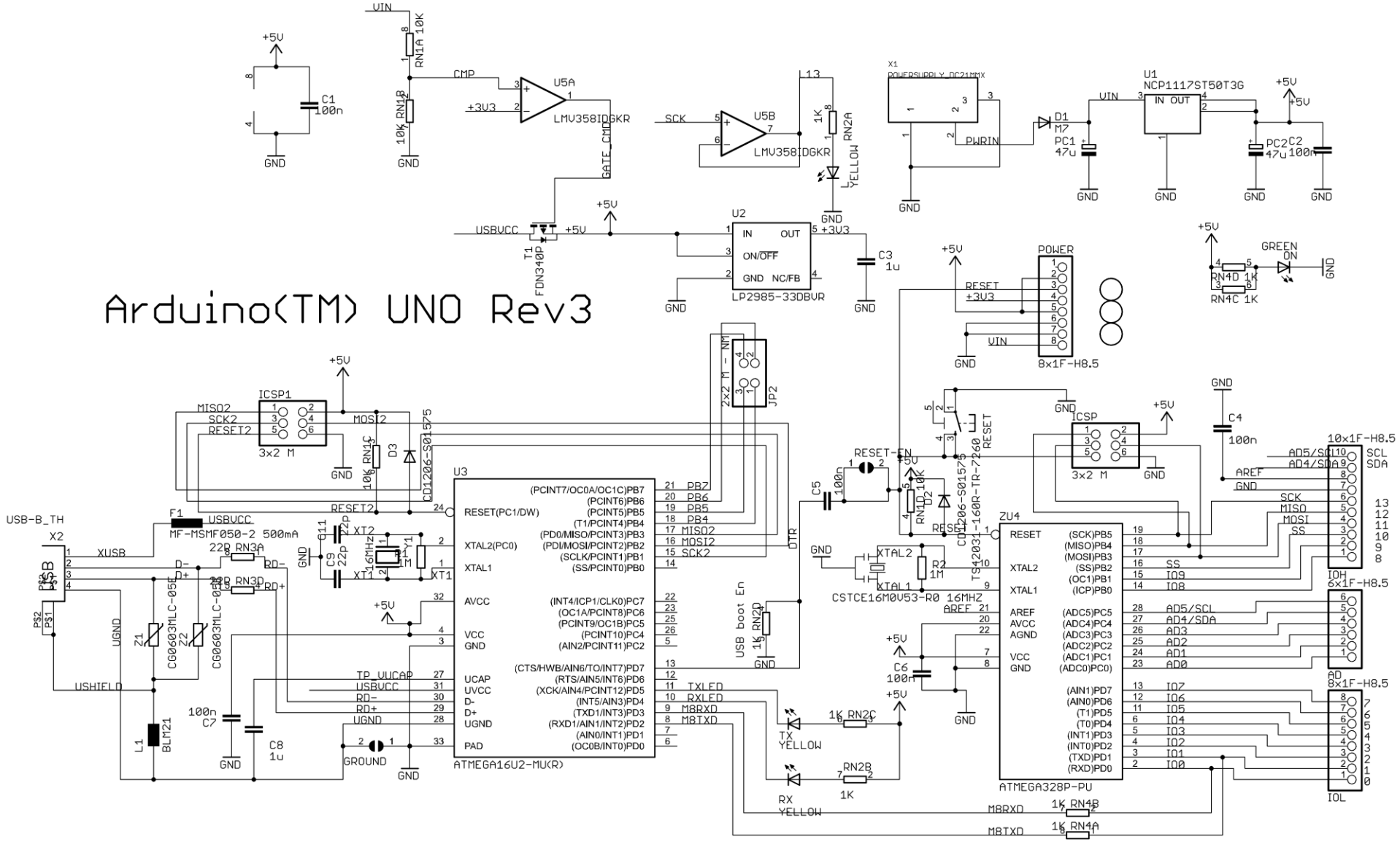
Arduino Uno Pinout - ICSP Header

- ICSP stands for In-Circuit Serial Programming. The name originated from In-System Programming headers (ISP).
- Manufacturers like Atmel who work with Arduino have developed their own in-circuit serial programming headers.
- These pins enable the user to program the Arduino boards' firmware.
- There are six ICSP pins available on the Arduino board that can be hooked to a programmer device via a programming cable.





Arduino UNO Schematic





Lab

Lab 1 / LO1

Working with Outputs



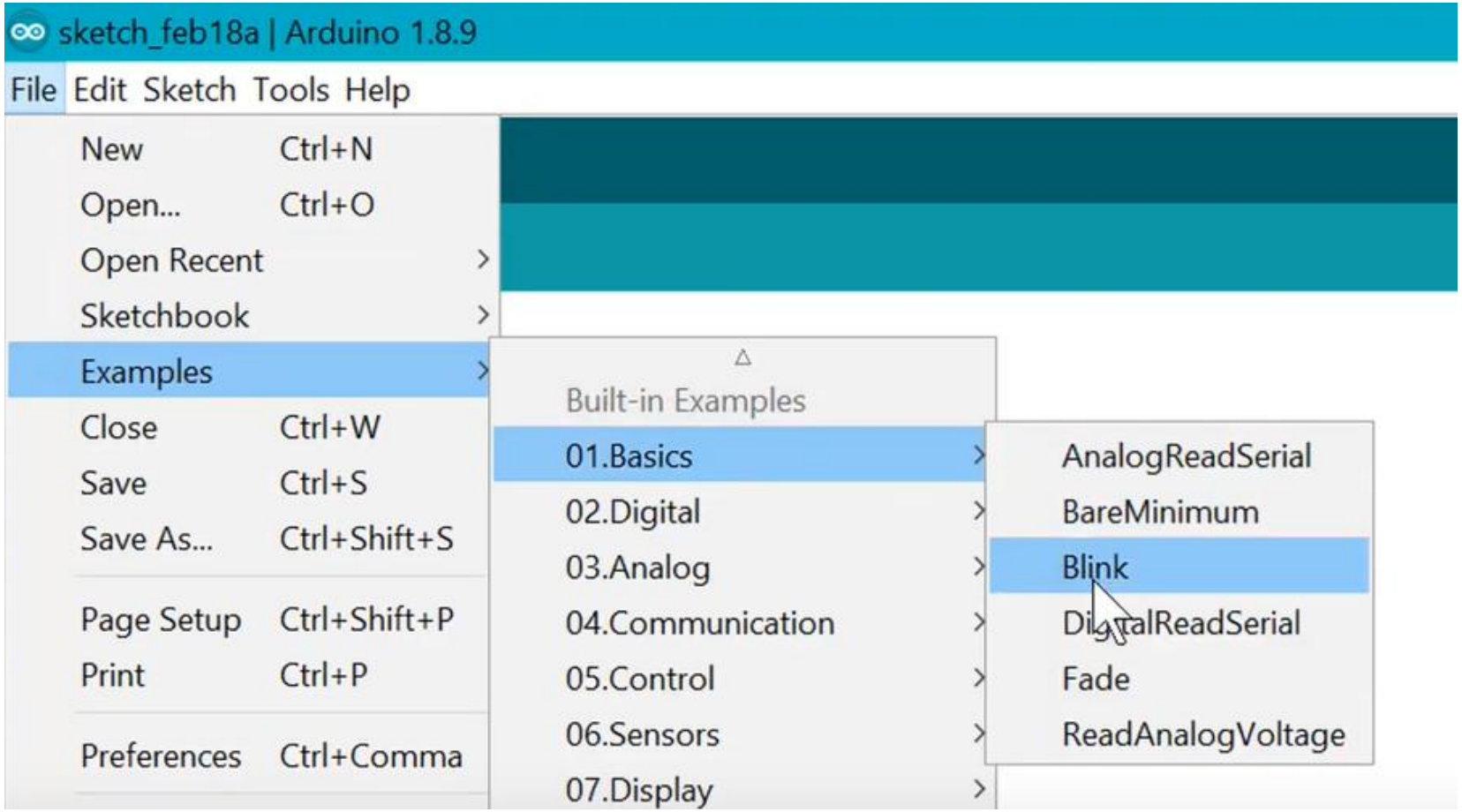
Tools and components

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. Breadboard wires (male-male)
  4. Oscilloscope

- Required components:
  1. Arduino UNO + USB cable
  2. Led x4
  3. Resistor from 200 to 470 ohm x 4



Starting with the simplest code (blink example)





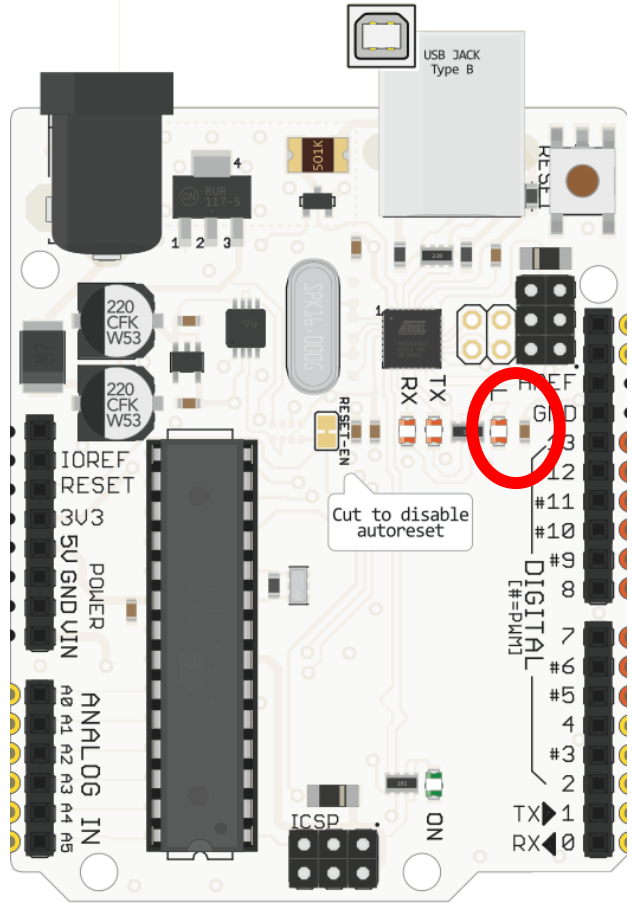
Starting with the simplest code (blink example)

The easiest example to start with is blinking the onboard LED connected already on pin13 of the Arduino boards.

Blink

```
// Pin 13 has an LED connected on most Arduino boards.
// Pin 11 has the LED on Teensy 2.0
// Pin 6  has the LED on Teensy++ 2.0
// Pin 13 has the LED on Teensy 3.0
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}
```





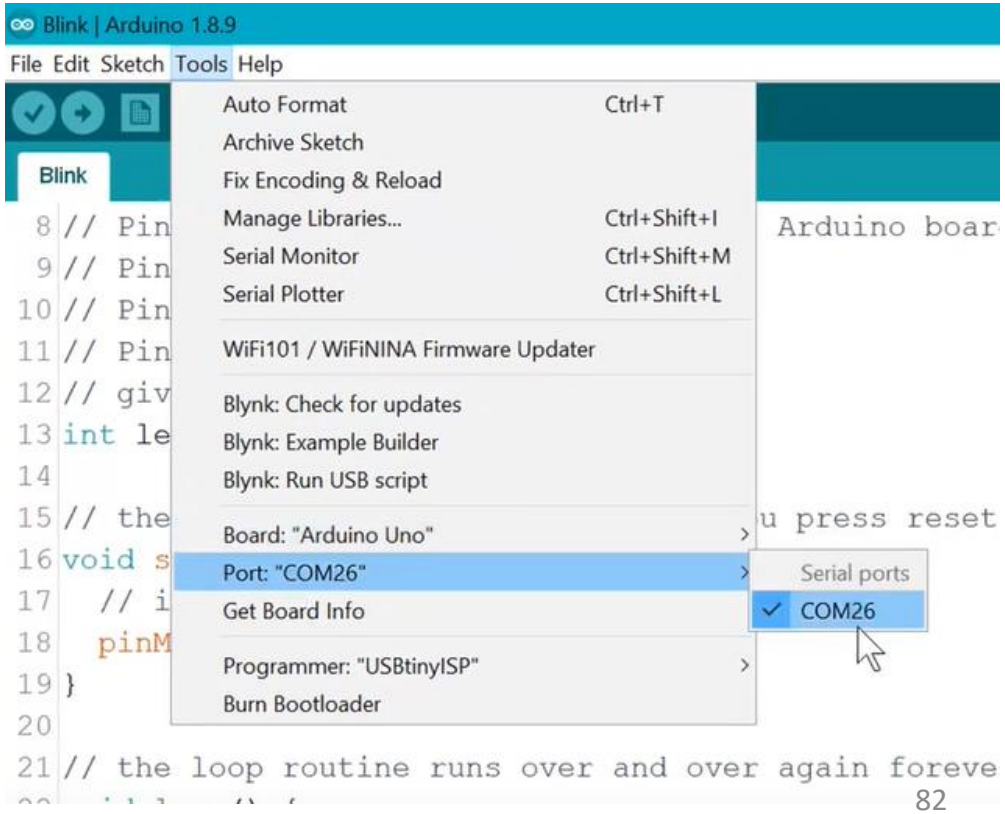
Starting with the simplest code (blink example)

Establish the communication with PC

- Before loading the code onto the Arduino flash memory, it is required to check the communication port between the PC and the Arduino board.

- The com port can be any number (in this case COM26)

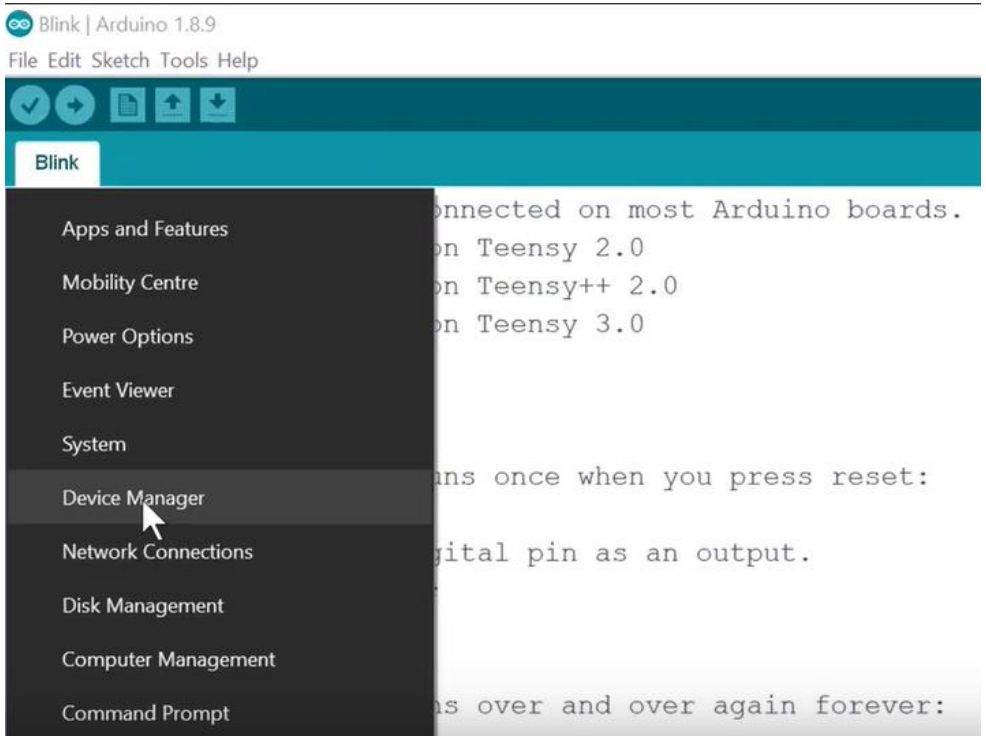
- NOTE: make sure to connect the USB cable

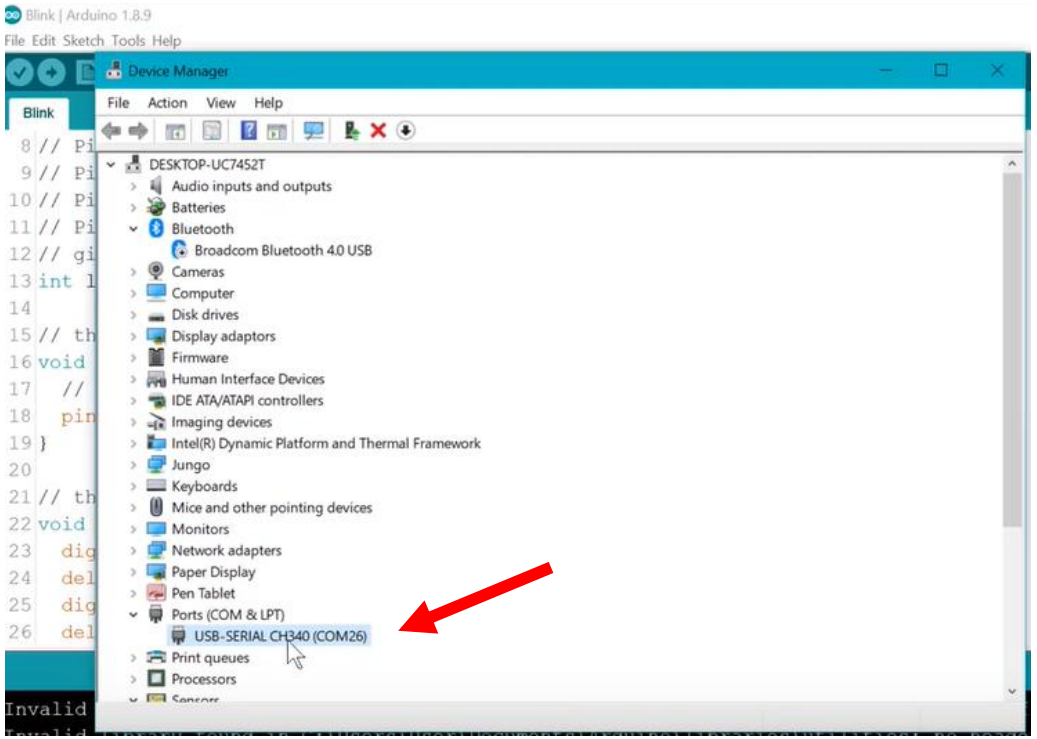




Starting with the simplest code (blink example)

- In order to check which COM port is attached to the Arduino, you can press "Window" + "X" button on the keyboard to open the device manager.
- Here you can check the com number from the list (in case of many com port available)

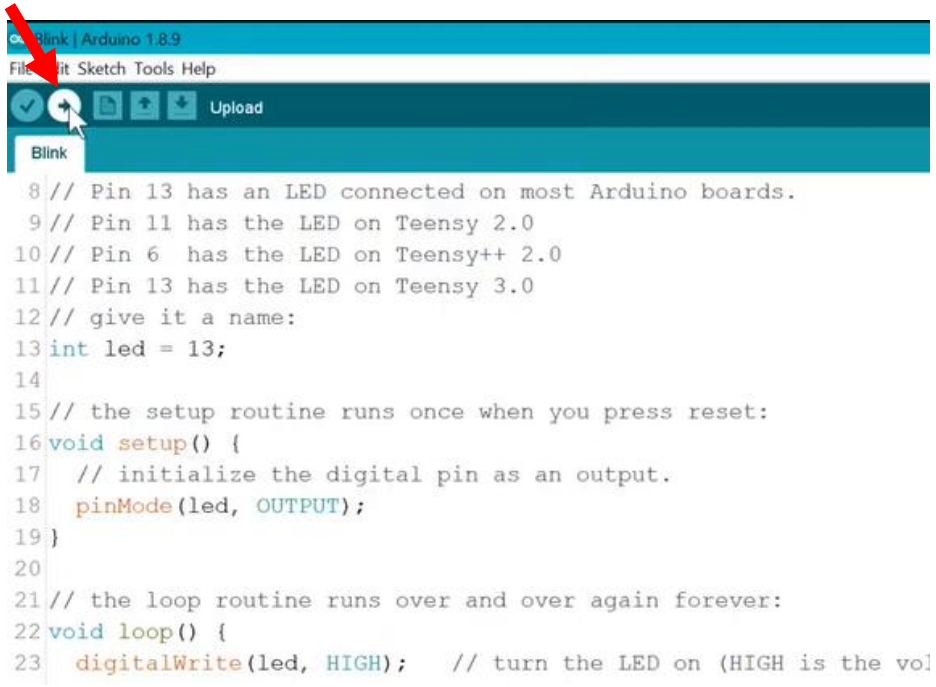


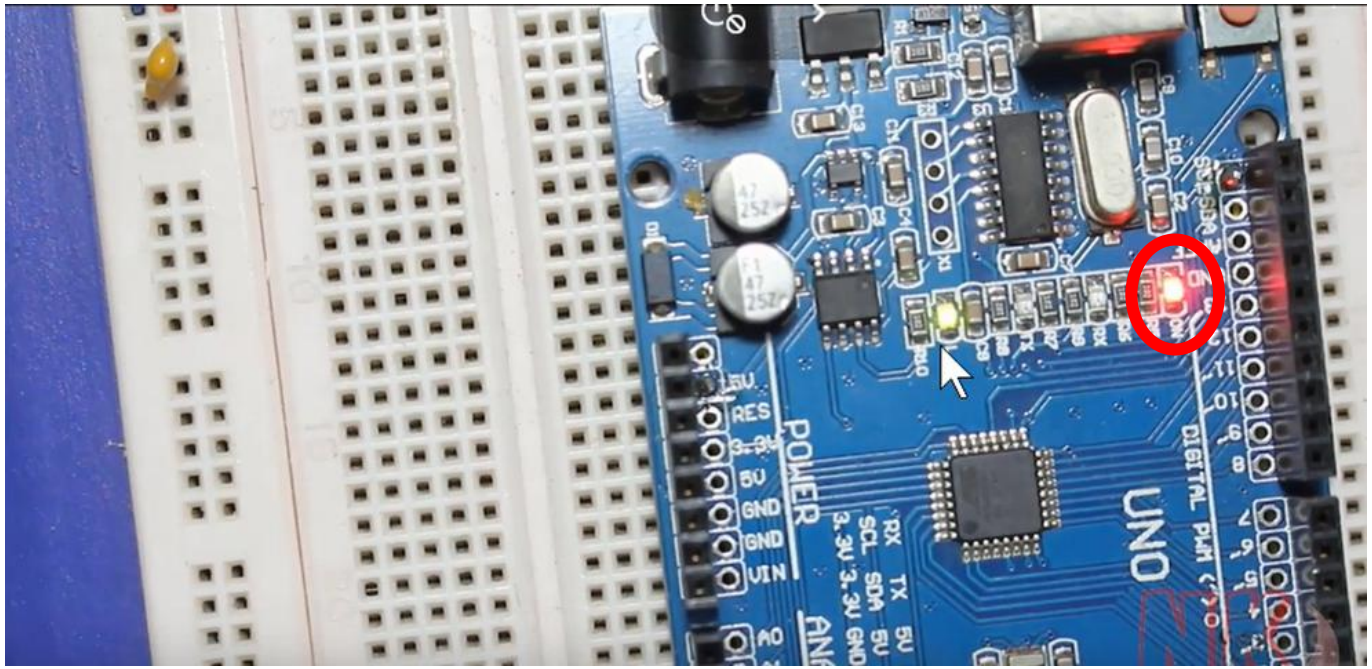




Starting with the simplest code (blink example)

- Click on the UPLOAD button to compile the code and upload it to the Arduino memory (The Program memory)
- When done, the onboard led will blink for 1s ON and 1s OFF







Structure of a blank sketch

- The setup() function is called when a sketch starts. Use it to initialize variables, pin modes, start using libraries, etc.
- **The setup() function will only run once**, after each powerup or reset of the Arduino board.

- After creating a setup() function, which initializes and sets the initial values, the loop() function does precisely what its name suggests, and **loops consecutively**, allowing your program to change and respond. Use it to actively control the Arduino board.

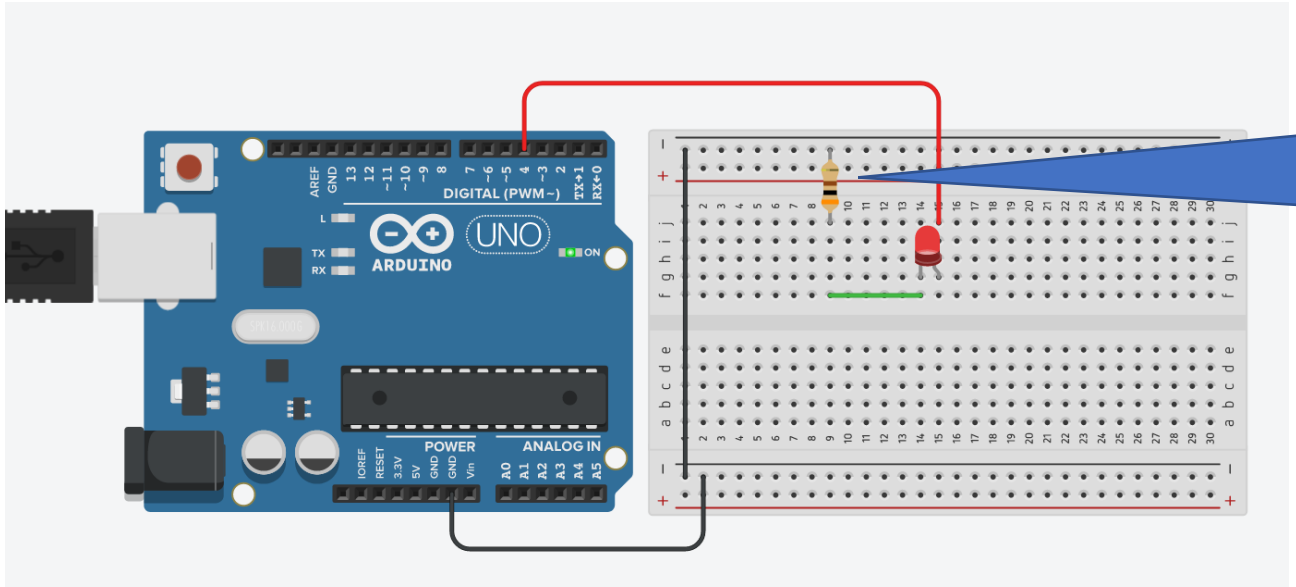




Writing our first code

Design requirement

It is required to turn on an LED connected to Pin4.
The circuit shows how to connect the UNO with the LED through a current limiting resistor of 300 Ohm.



R is calculated as

$$R = \frac{5 - V_{led}}{I_{led}}$$

$$= \frac{5 - 2}{10mA} = 300\ \Omega$$



Writing our first code

In this example, there is no need to write anything in the loop() function as the role of the Arduino is just to turn the LED ON.
Thus, the code will be written inside the setup() function

led_on.ino §

```
1  // this code turns on an LED connected to pin # 4 of the Arduino UNO
2
3  void setup() {
4    pinMode(4, OUTPUT);    // this instruction defines the pin4 as output
5    digitalWrite(4, HIGH); // send 1 logic on pin # 4, i.e, turn on the LED
6  }
7
8  void loop() {
9
10
11 }
```



Writing our first code

We can enhance the code by defining the LED as pin4. This will help us understand the code more easily.

- We can use #define directive or
- Declare the led as integer equal to 4

led_on.ino §

```
1  // this code turns on an LED connected to pin # 4 of the Arduino UNO
2
3  #define led 4                // defining led as pin # 4
4
5  //int led = 4;               // also this line can be used to define led
6
7  void setup() {
8    pinMode(led, OUTPUT);      // this instruction defines the pin4 as output
9    digitalWrite(led, HIGH);   // send 1 logic on pin # 4, i.e, turn on the LED
10 }
11
12 void loop() {
13
14
15 }
```



Writing our second code

Design requirement

It is now required to modify our code to blink the LED (same as the blink example but on pin4)
Since the code will flip periodically the state of the LED, thus it is better to write the code inside the loop() function

blink §

```
1  // this code blinks an LED connected to pin # 4 every 1s
2  // The total time (period) is 2s, i.e., frequecny is 0.5 Hz
3
4  #define led 4                 // defining led as pin # 4
5
6  void setup() {
7    pinMode(led, OUTPUT);       // this instruction defines the pin4 as output
8  }
9
10 void loop() {
11   digitalWrite(led, HIGH);    // turn on led
12   delay(1000);                // for 1000 ms   (1sec)
13   digitalWrite(led, LOW);
14   delay(1000);
15
16 }
```



Modify the code to blink the LED at higher Frequency

Higher frequency means the delay time is much shorter.
For example, for a frequency of 20 Hz, the period will be 1/f = 1/20 = 50ms.
Thus, the ON time is 25ms and the OFF time is also 25ms.

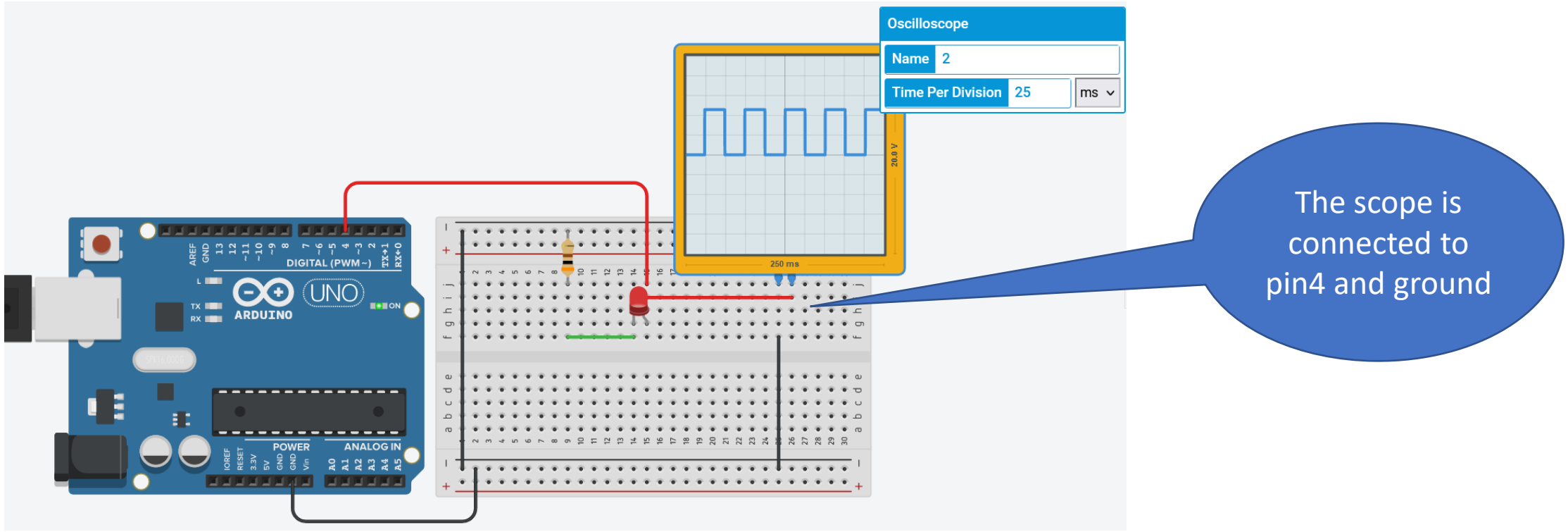
blink §

```
1  // this code blinks an LED connected to pin # 4 at 20Hz
2  // The total time (period) is 50ms,
3  // The ON time is 25ms and the OFF time is 25ms
4
5
6  #define led 4                // defining led as pin # 4
7
8  void setup() {
9    pinMode(led, OUTPUT);      // this instruction defines the pin4 as output
10 }
11
12 void loop() {
13   digitalWrite(led, HIGH);   // turn on led
14   delay(25);                 // for 25 ms
15   digitalWrite(led, LOW);
16   delay(25);
17
18 }
```



Modify the code to blink the LED at higher Frequency

At a high blinking rate (above 20 Hz), the human eyes cannot perceive the blinking effect of the LED.
In order to make sure that the code is generating the ON/OFF signal, a scope can be used.

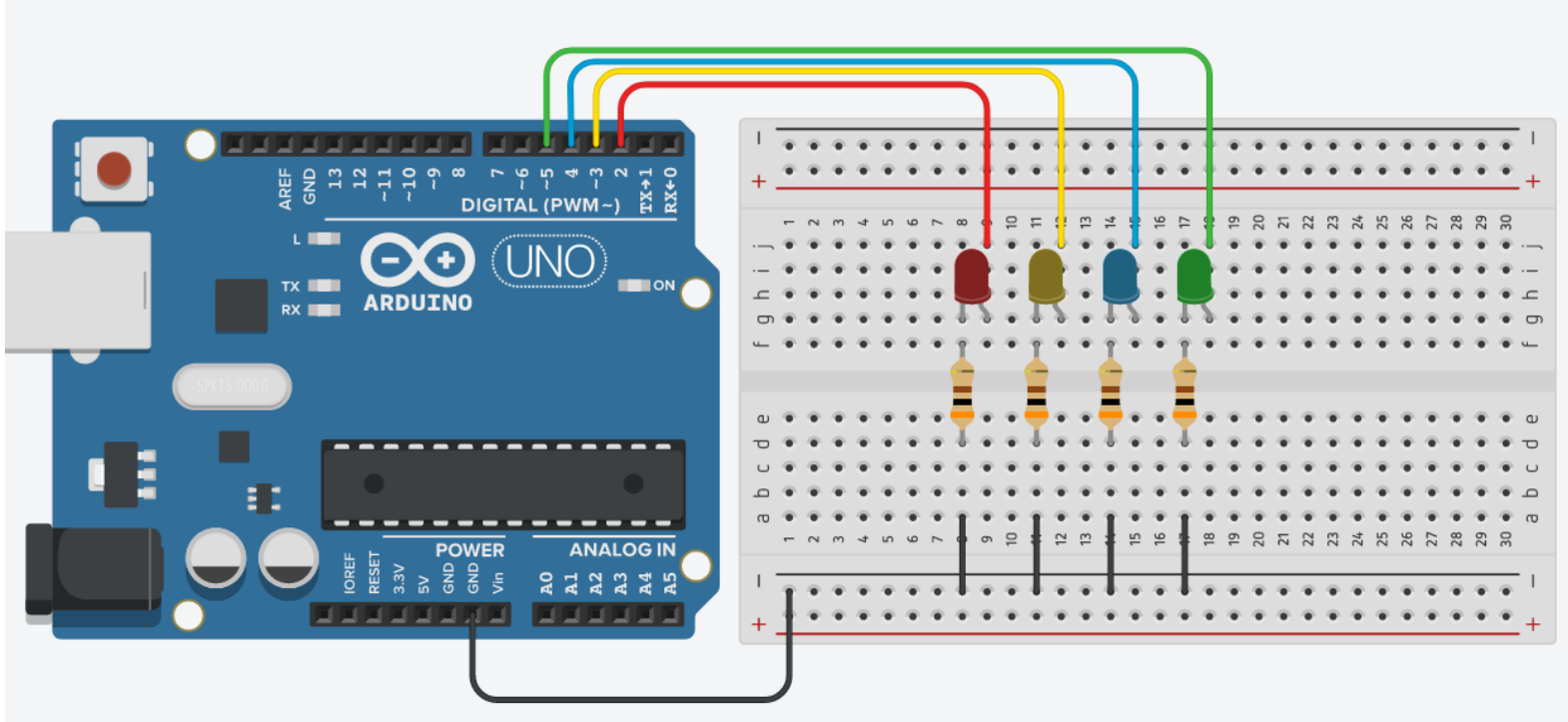




Code 3: Chasing LEDs

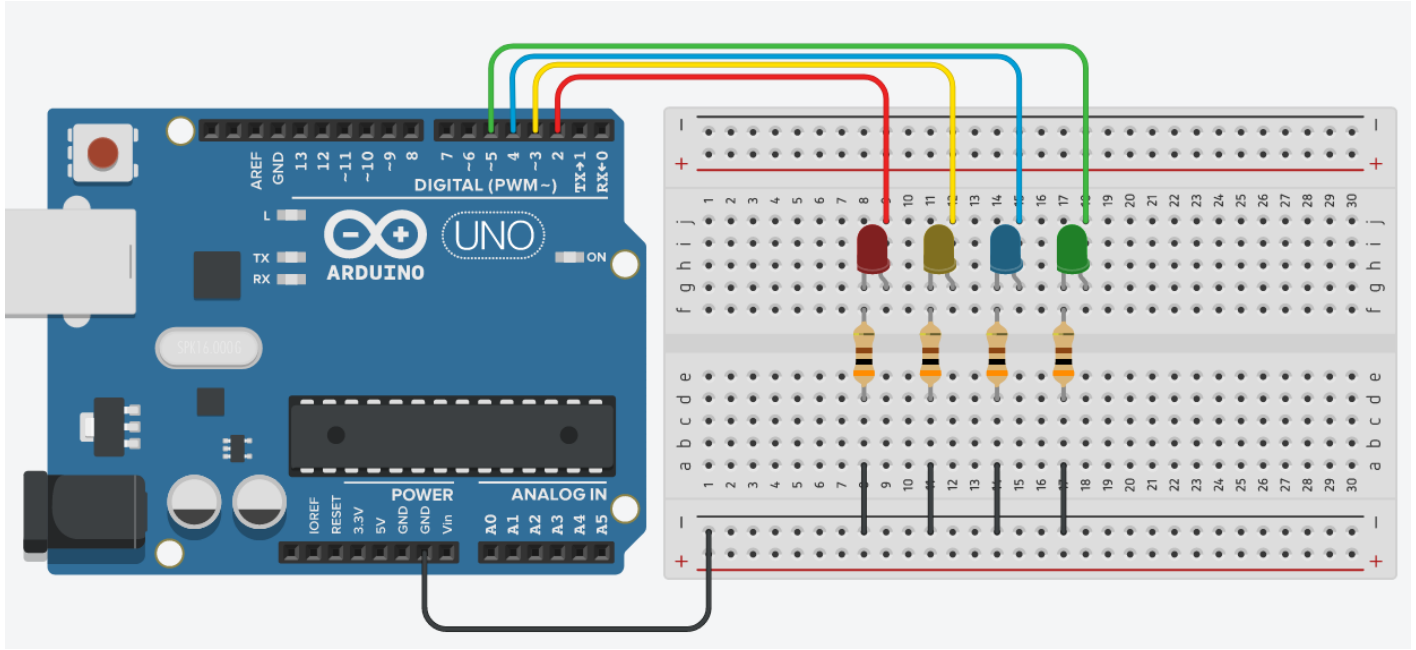
Design requirement

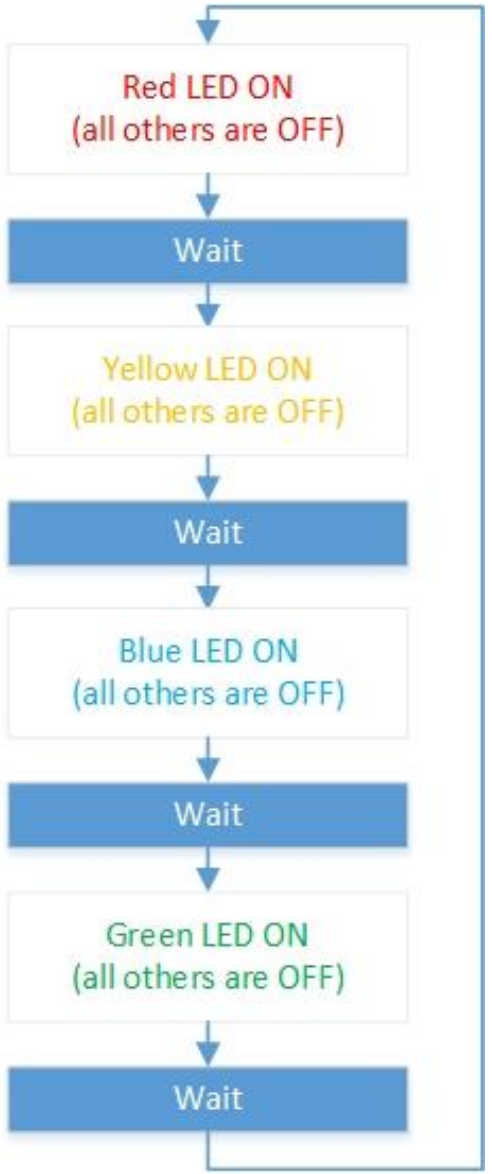
It is required now to control the scrolling LEDs (chasing LEDs).
The concept is to turn ON the LEDs one by one spaced with some delay





Code 3: Chasing LEDs







Code 3: Chasing LEDs

chassing_leds §

```
// this code
#define led_red     2
#define led_yellow  3
#define led_blue    4
#define led_green   5

void setup() {
  pinMode(led_red, OUTPUT);
  pinMode(led_yellow, OUTPUT);
  pinMode(led_blue, OUTPUT);
  pinMode(led_green, OUTPUT);
}

void loop() {
  digitalWrite(led_red, HIGH);       // red led is on
  digitalWrite(led_yellow, LOW);     // yellow led is off
  digitalWrite(led_blue, LOW);       // blue led is off
  digitalWrite(led_green, LOW);      // green led is off
  delay(100);                        // wait for 100 ms

  digitalWrite(led_red, LOW);        // red led is off
  digitalWrite(led_yellow, HIGH);    // yellow led is on
  digitalWrite(led_blue, LOW);       // blue led is off
  digitalWrite(led_green, LOW);      // green led is off
  delay(100);                        // wait for 100 ms

  digitalWrite(led_red, LOW);        // red led is off
  digitalWrite(led_yellow, LOW);     // yellow led is off
  digitalWrite(led_blue, HIGH);      // blue led is on
  digitalWrite(led_green, LOW);      // green led is off
  delay(100);                        // wait for 100 ms

  digitalWrite(led_red, LOW);        // red led is off
  digitalWrite(led_yellow, LOW);     // yellow led is off
  digitalWrite(led_blue, LOW);       // blue led is off
  digitalWrite(led_green, HIGH);     // green led is on
  delay(100);                        // wait for 100 ms

}
```



Week 3

Lecture 3 / LO2

C language



C++ language for Arduino

Variables and types

- C is a statically typed language. This means that any variable has an associated type, and this type is known at compilation time.
- When you create a variable in C, you have to specify the type of a variable at the declaration.
- In this example we initialize a variable age with type int :

```
int age;
```

- You can also initialize a variable at declaration, specifying the initial value:

```
int age = 37;
```

- The C built-in data types are int , char , short , long , float , double , long double .



C++ language for Arduino

Variables and types (Integer)

C provides us the following types to define integer values:

1. char
2. int
3. long

- Most of the times, you'll likely use an int to store an integer. But in some cases, you might want to choose one of the other 3 options.
- The **char** type is commonly used to store letters of the ASCII chart, but it can be used to hold small integers from -128 to 127 . It takes 1 byte.
- **int** takes 2 bytes ranging from -32,768 to 32,767 .
- **long** takes 4 bytes -2,147,483,648 to 2,147,483,647



C++ language for Arduino

Variables and types (Integer)

For all the above data types, we can prepend unsigned to start the range at 0, instead of a negative number.

- This might make sense in many cases.
- unsigned **char** will range from 0 to at least 255
- unsigned **int** will range from 0 to at least 65,535
- unsigned **long** will range from 0 to at least 4,294,967,295



C++ language for Arduino

Variables and types (Floating)

- Floating point types can represent a much larger set of values than integers can, and can also represent fractions, something that integers can't do.
- Using floating point numbers, we represent numbers as decimal numbers times powers of 10.
- You might see floating point numbers written as 1.29e-3 or -2.3e+5
- The following types:
  1. float
  2. double
  3. long double
- are used to represent numbers with decimal points (floating point types). All can represent both positive and negative numbers.
- The minimum requirements for any C implementation is that float can represent a range between 10^-37 and 10^+37, and is typically implemented using 32 bits. double can represent a bigger set of numbers.



C++ language for Arduino

Variables and types

Data Type	Size (Bits)	Range	Comments
char	8	-128 to +127	Signed Character
signed char	8	-128 to +127	Signed Character
unsigned char	8	0 to +255	Unsigned Character
short int	8	-128 to +127	Signed short integer
signed short int	8	-128 to +127	Signed short integer
unsigned short int	8	0 to +255	Unsigned short integer
int	16	-32,768 to +32,767	Signed integer
signed int	16	-32,768 to +32,767	Signed integer
unsigned int	16	0 to +65,535	Unsigned integer
long int	32	-2147,483,648 to +2,147,483,647	Signed long integer
signed long int	32	-2147,483,648 to +2,147,483,647	Signed long integer
unsigned long int	32	0 to +4,294,967,295	Unsigned long integer
float	32	3.4E-38 to 3.4E+38	Signed floating point
double	64	1.7E-308 to 1.7E+308	Signed floating point (Double precision)
long double	80	3.4E-4932 to 3.4E+4932	Signed floating point (Long Double precision)



C++ language for Arduino

Operators

- C offers us a wide variety of operators that we can use to operate on data.
- In particular, we can identify various groups of operators:
  - arithmetic operators
  - comparison operators
  - logical operators
  - compound assignment operators
  - bitwise operators
  - pointer operators
  - structure operators
  - miscellaneous operators



C++ language for Arduino

Operators (Arithmetic)

- Binary operators work using two operands:

Operator	Name	Example
`=`	Assignment	`a = b`
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulo	`a % b`



C++ language for Arduino

Operators (Arithmetic)

- Unary operators only take one operand:

Operator	Name	Example
`+`	Unary plus	`+a`
`-`	Unary minus	`-a`
`++`	Increment	`a++` or `++a`
`--`	Decrement	`a--` or `--a`

- The difference between a++ and ++a is that a++ increments the a
- variable after using it. ++a increments the a variable before using it.
- For example:

```
int a = 2;
int b;
b = a++ /* b is 2, a is 3 */
b = ++a /* b is 4, a is 4 */
```



C++ language for Arduino

Operators (Comparison)

Operator	Name	Example
`==`	Equal operator	`a == b`
`!=`	Not equal operator	`a != b`
`>`	Bigger than	`a > b`
`<`	Less than	`a < b`
`>=`	Bigger than or equal to	`a >= b`
`<=`	Less than or equal to	`a <= b`



C++ language for Arduino

Operators (Logical)

- ! NOT (example: !a)
- && AND (example: a && b)
- || OR (example: a || b)

Those operators are great when working with Boolean values.



C++ language for Arduino

Operators (Compound assignment operators)

- Those operators are useful to perform an assignment and at the same time
- perform an arithmetic operation:

Operator	Name	Example
`+=`	Addition assignment	`a += b`
`-=`	Subtraction assignment	`a -= b`
`*=`	Multiplication assignment	`a *= b`
`/=`	Division assignment	`a /= b`
`%=`	Modulo assignment	`a %= b`



C++ language for Arduino

Conditionals

- Any programming language provides the programmers the ability to perform choices.
- We want to do X in some cases, and Y in other cases.
- We want to check data, and do choices based on the state of that data.
- C provides us 2 ways to do so.
- The first is the if statement, with its else helper, and the second is the switch statement.



C++ language for Arduino

Conditionals (If Condition)

- In an if statement, you can check for a condition to be true, and then execute the block provided in the curly brackets:

```
int a = 1;

if (a == 1) {
  /* do something */
}
```

- You can append an else block to execute a different block if the original condition turns out to be false;

```
int a = 1;

if (a == 2) {
  /* do something */
} else {
  /* do something else */
}
```



C++ language for Arduino

Conditionals (If Condition)

- You can have multiple else blocks by stacking together multiple if statements:

```
int a = 1;

if (a == 2) {
  /* do something */
} else if (a == 1) {
  /* do something else */
} else {
  /* do something else again */
}
```



C++ language for Arduino

Conditionals (switch)

- If you need to do too many if / else / if blocks to perform a check, perhaps because you need to check the exact value of a variable, then switch can be very useful to you.
- You can provide a variable as condition, and a series of case entry points for each value you expect:

```
int a = 1;

switch (a) {
  case 0:
    /* do something */
    break;
  case 1:
    /* do something else */
    break;
  case 2:
    /* do something else */
    break;
}
```

- We need a break keyword at the end of each case, to avoid the next case to be executed when the one before ends. This "cascade" effect can be useful in some creative ways.



C++ language for Arduino

Conditionals (switch)

- You can add a "catch-all" case at the end, labeled default :

```
int a = 1;

switch (a) {
  case 0:
    /* do something */
    break;
  case 1:
    /* do something else */
    break;
  case 2:
    /* do something else */
    break;
  default:
    /* handle all the other cases */
    break;
}
```



C++ language for Arduino

Loops

- C offers us three ways to perform a loop:
  - **for loops**,
  - **while loops**
  - **do while loops**.
- They all allow you to iterate over arrays, but with a few differences. Let's see them in details.



C++ language for Arduino

Loops (for loop)

- The first, and probably most common, way to perform a loop is **for loops**.
- Using the for keyword we can define the *rules* of the loop up front, and then provide the block that is going to be executed repeatedly. Like this:

```
for (int i = 0; i <= 10; i++) {
  /* instructions to be repeated */
}
```



C++ language for Arduino

Loops (for loop)

- The (int i = 0; i <= 10; i++) block contains 3 parts of the looping details:
  - the initial condition (int i = 0)
  - the test (i <= 10)
  - the increment (i++)
- We first define a loop variable, in this case named i . i is a common variable name to be used for loops, along with j for nested loops (a loop inside another loop). It's just a convention.
- The variable is initialized at the 0 value, and the first iteration is done. Then it is incremented as the increment part says (i++ in this case, incrementing by 1), and all the cycle repeats until you get to the number 10.
- Inside the loop main block we can access the variable i to know at which iteration we are. This program should print 0 1 2 3 4 5 5 6 7 8 9 10 :

```
for (int i = 0; i <= 10; i++) {
  /* instructions to be repeated */
  printf("%u ", i);
}
```



C++ language for Arduino

Loops (for loop)

- Loops can also start from a high number, and go a lower number, like this:

```
for (int i = 10; i > 0; i--) {
  /* instructions to be repeated */
}
```

- You can also increment the loop variable by 2 or another value:

```
for (int i = 0; i < 1000; i = i + 30) {
  /* instructions to be repeated */
}
```



C++ language for Arduino

Loops (while loop)

- **While loops** is simpler to write than a for loop, because it requires a bit more work on your part.
- Instead of defining all the loop data up front when you start the loop, like you do in the for loop, using while you just check for a condition:

```
while (i < 10) {

}
```

- This assumes that i is already defined and initialized with a value. And this loop will be an **infinite loop** unless you increment the i variable at some point inside the loop.
- An infinite loop is bad because it will block the program, nothing else can happen.



C++ language for Arduino

Loops (while loop)

- This is what you need for a "correct" while loop:

```
int i = 0;

while (i < 10) {
  /* do something */

  i++;
}
```



C++ language for Arduino

Loops (Do while loops)

- While loops are great, but there might be times when you need to do one particular thing: you want to always execute a block, and then *maybe* repeat it.
- This is done using the do while keyword, in a way that's very similar to a while loop, but slightly different:

```
int i = 0;

do {
  /* do something */

  i++;
} while (i < 10);
```

- The block that contains the /* do something */ comment is always executed at least once, regardless of the condition check at the bottom.
- Then, until i is less than 10, we'll repeat the block.



C++ language for Arduino

Loops (Breaking out of a loop using break)

- In all the C loops we have a way to break out of a loop at any point in time, immediately, regardless of the conditions set of the loop.
- This is done using the break keyword.
- This is useful in many cases. You might want to check for the value of a variable, for example:

```
for (int i = 0; i <= 10; i++) {
  if (i == 4 && someVariable == 10) {
    break;
  }
}
```

```
int i = 0;
while (1) {
  /* do something */

  i++;
  if (i == 10) break;
}
```



C++ language for Arduino

Arrays

- An array is a variable that stores multiple values.
- Every value in the array, in C, must have the **same type**. This means you will have arrays of int values, arrays of double values, and more.
- You can define an array of int values like this:

```
int prices[5];
```

- You must always specify the size of the array. C does not provide dynamic arrays out of the box (you have to use a data structure like a linked list for that).
- You can use a constant to define the size:

```
const int SIZE = 5;
int prices[SIZE];
```



C++ language for Arduino

Arrays

- You can initialize an array at definition time, like this:

```
int prices[5] = { 1, 2, 3, 4, 5 };
```

```
int prices[5];

prices[0] = 1;
prices[1] = 2;
prices[2] = 3;
prices[3] = 4;
prices[4] = 5;
```

- And you can reference an item in the array by using square brackets after the array variable name, adding an integer to determine the index value. Like this:

```
prices[0]; /* array item value: 1 */
prices[1]; /* array item value: 2 */
```



C++ language for Arduino

Strings

- In C, strings are one special kind of array: a string is an array of char values:

```
char name[7];
```

- I introduced the char type when I introduced types, but in short it is commonly used to store letters of the ASCII chart. A string can be initialized like you initialize a normal array:

```
char name[7] = { 'F', 'l', 'a', 'v', 'i', 'o' };
```

- Or more conveniently with a string literal (also called string constant), a
- sequence of characters enclosed in double quotes:

```
char name[7] = "Flavio";
```



C++ language for Arduino

Functions

- Functions are the way we can structure our code into subroutines that we can:
- 1. give a name to
- 2. call when we need them

- A function can have arguments. They are optional. If it does not have them, inside the parentheses we insert void , like this:

```
void doSomething(void) {
   /* ... */
}
```

- In this case, when we invoke the function we'll call it with nothing in the
- parentheses:

```
doSomething();
```



C++ language for Arduino

Functions

- If we have one parameter, we specify the type and the name of the parameter, like this:

```
void doSomething(int value) {
   /* ... */
}
```

- When we invoke the function, we'll pass that parameter in the parentheses,
- like this:

```
doSomething(3);
```



C++ language for Arduino

Functions

- We can have multiple parameters, and if so we separate them using a comma, both in the declaration and in the invocation:

```
void doSomething(int value1, int value2) {
    /* ... */
}

doSomething(3, 4);
```



C++ language for Arduino

Functions

- Also functions can be designed to return a value, such as,

```
int doSomething(int value1, int value2) {
  printf("%d %d\n", value1, value2);
  return 1;
}
```



C++ language for Arduino

Functions

- Make sure you define the function before calling it, or the compiler will raise a warning and an error

- make sure you declare the function before using it.
- Either move the function up, or add the function prototype in a header file.



C++ language for Arduino

Variables scope

- When you define a variable in a C program, depending on where you declare it, it will have a different **scope**. This means that it will be available in some places, but not in others.
- The position determines 2 types of variables:
  - **global variables**
  - **local variables**
- This is the difference: a variable declared inside a function is a local variable, like this:

```
int main(void) {
  int age = 37;
}
```

- A variable defined outside of a function is a global variable, like in this example:

```
int age = 37;

int main(void) {
  /* ... */
}
```

Week 3



Lab 2 / LO2

Working with Inputs



Tools and components

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. Breadboard wires (male-male)

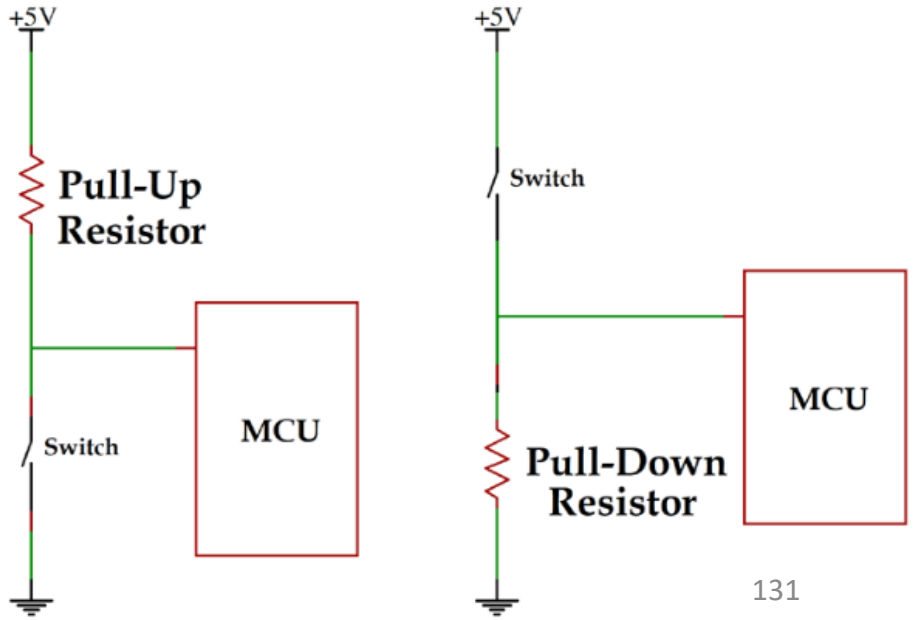
- Required components:
  1. Arduino UNO + USB cable
  2. Led x2
  3. Resistor from 200 to 470 ohm x 2
  4. Push button for breadboard x2
  5. Resistor from 1k to 47k ohm x 2



Interfacing with Push button or Switch

- The digital pins on the Arduino UNO controller can be used as input and output.
- In this section, the digital pin will be used as an input.
- To detect the logic state, the input is linked to a switch such as a pushbutton.
- HIGH and LOW are the two digital states (0 or 1).
- There are two approaches for correctly setting the controller's digital input and using a resistor.
- These are referred to as pull-down and pull-up resistors.
- Resistors are used to force the pin to not be floating and so will not be affected by static charges or noise

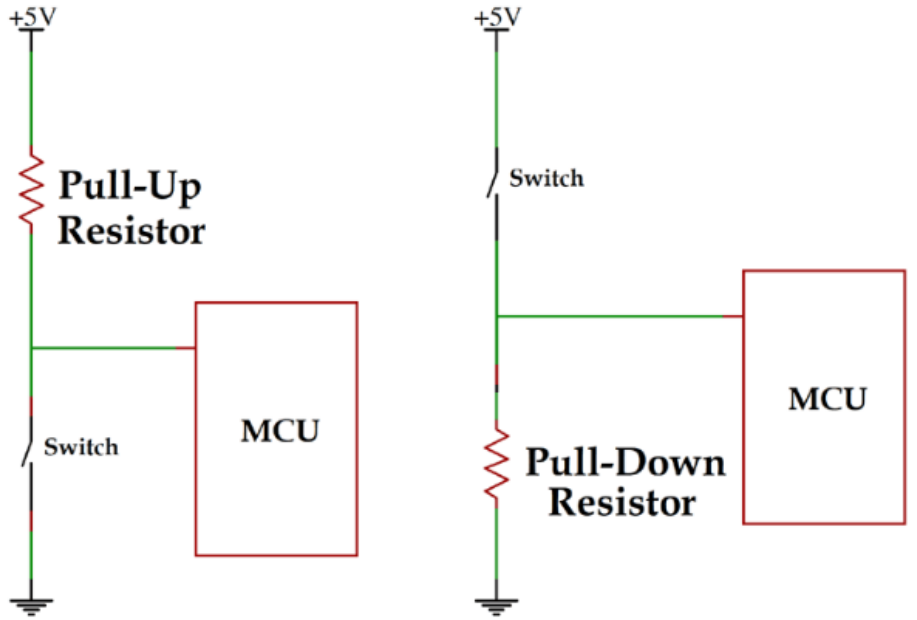






Interfacing with Push button or Switch

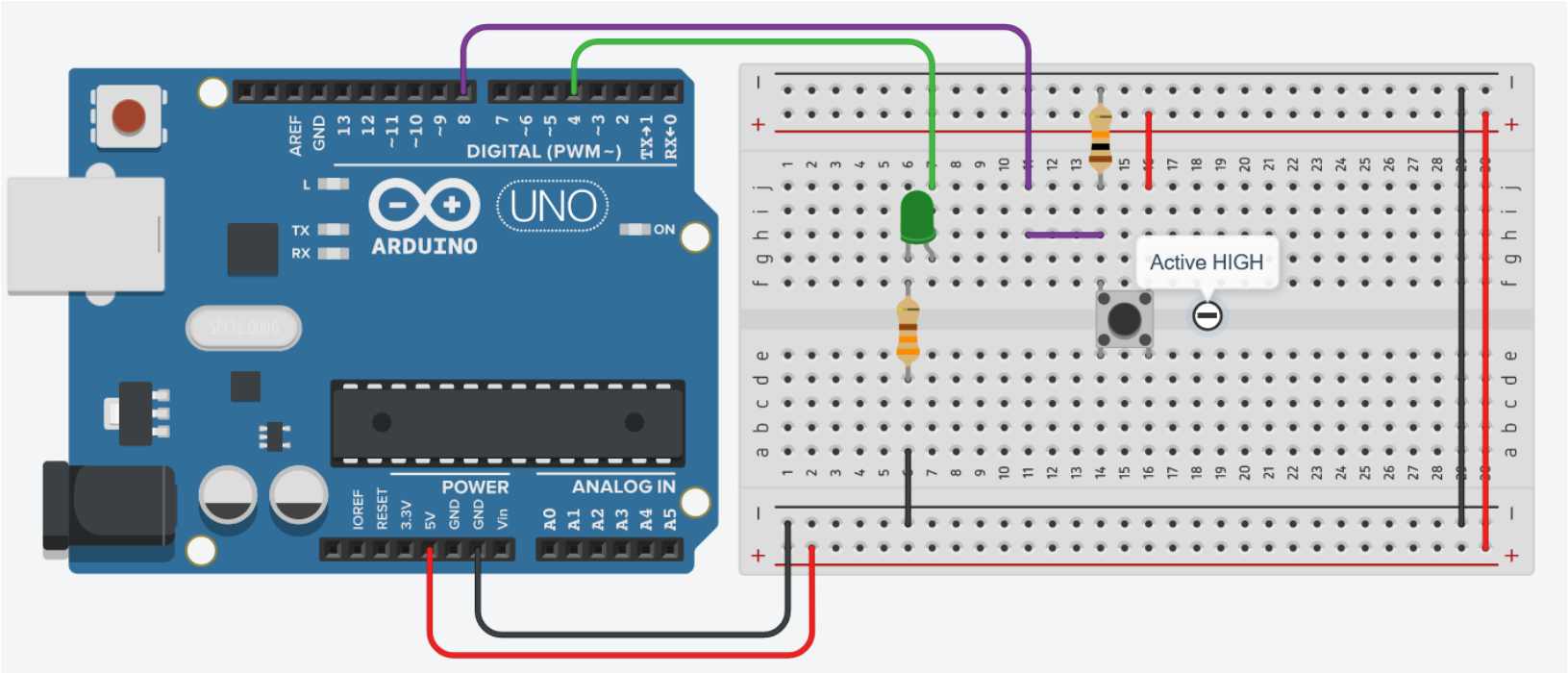
- In the pull-down resistor method, a resistor is added in series with the digital input switch. The 5V is connected to another side of the switch, and input is taken between the switch and the resistor.
- When the switch is not closed, the microcontroller digital input will get a logic of 0.
- When the switch is closed, it will get a logic of 1.
- In this method, we can state that when the switch is pressed, the input will be active high otherwise, it remains active low.
- In the pull-up resistor method, when the switch is pressed, the controller's input receives low logic; otherwise, it typically receives high logic.
- In pull-up resistor configuration, the resistor on one side is connected with high potential (5V) and on another with a switch.
- The connection for microcontroller input is given from the middle of the switch and resistor connection

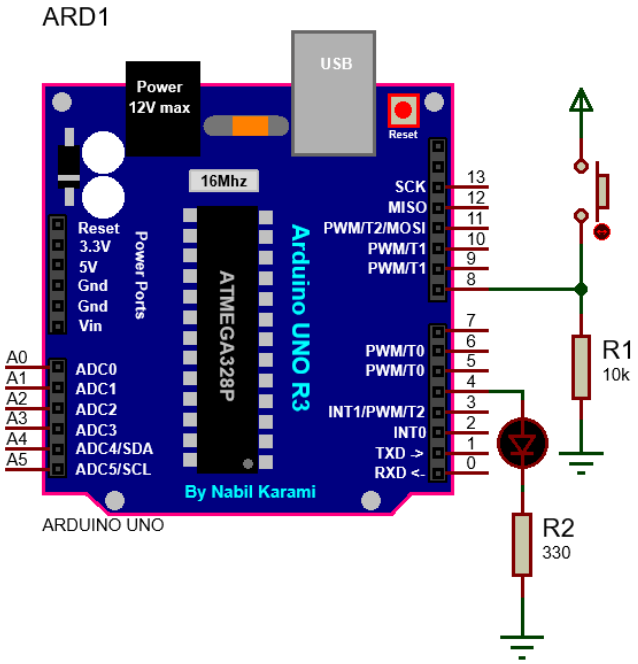




Code 4: Led with Push button (Active HIGH PB)

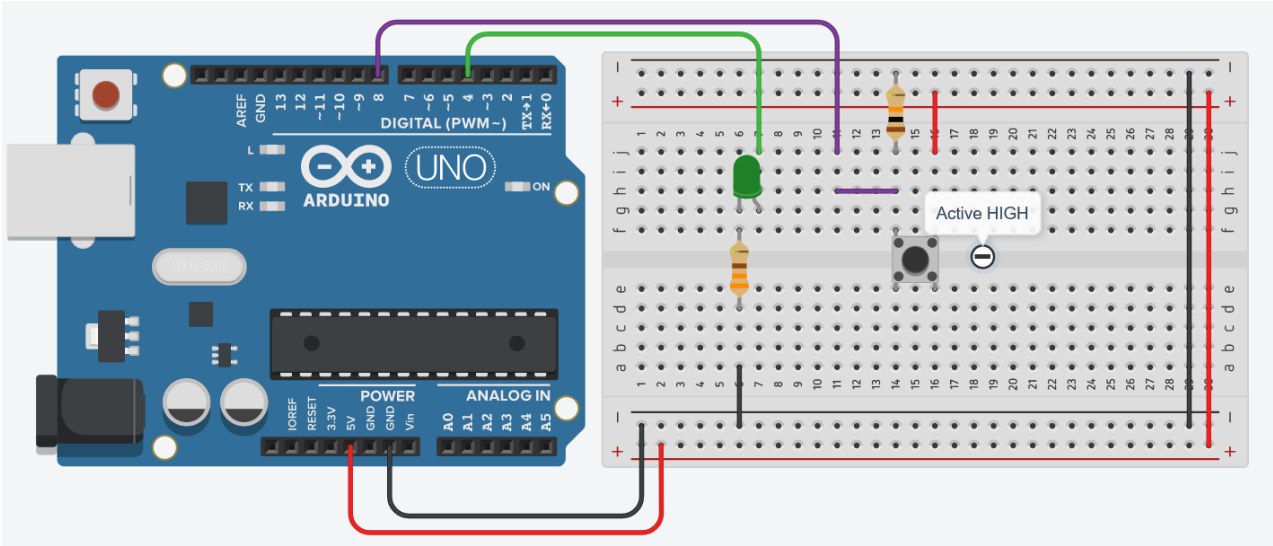
It is required now to turn ON the LED when the Push button is pressed.
The push button is active high (it delivers 5 volt when pressed). When the push button is pressed, a 5 volt (1 logic) is read on the pin8.
When the push button is released, the Pull down resistor (10KΩ) ensure that Pin8 will not be floating and force the level to be 0 logic or 0 volt.

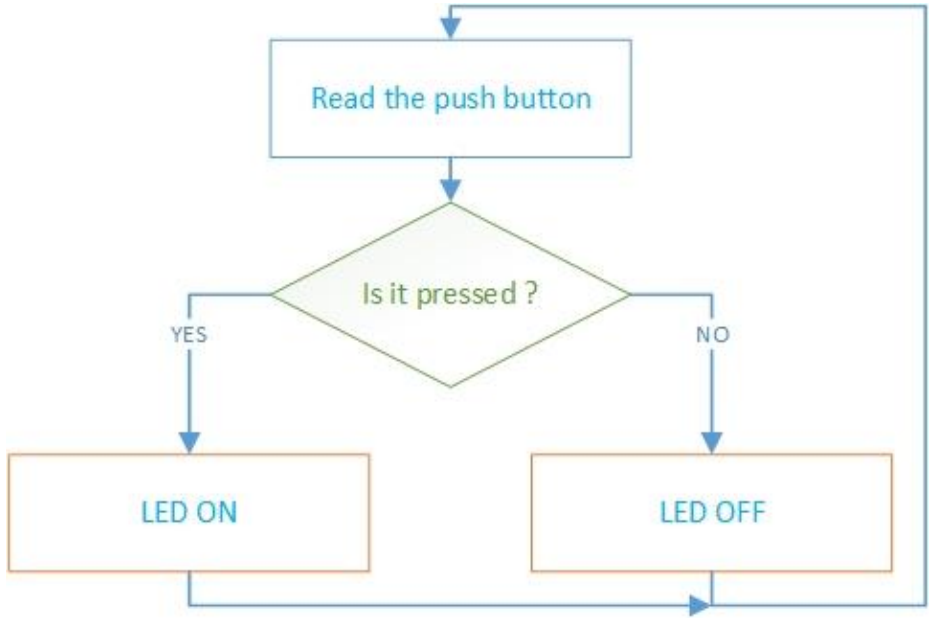






Code 4: Led with Push button (Active HIGH PB)







Code 4: Led with Push button (Active HIGH PB)

Code

```
1  #define pb  8
2  #define led_green 4
3
4  bool pb_state = false;
5
6  void setup() {
7    pinMode(pb, INPUT);
8    pinMode(led_green, OUTPUT);
9  }
10
11 void loop() {
12   bool pb_state = digitalRead(pb);
13   if (pb_state == true) digitalWrite(led_green, HIGH);
14   else digitalWrite(led_green, LOW);
15 }
```



Code 4: Led with Push button (Active HIGH PB)

Code enhancement

```
1  #define pb  8
2  #define led_green 4
3
4  void setup() {
5    pinMode(pb, INPUT);
6    pinMode(led_green, OUTPUT);
7  }
8
9  void loop() {
10   if (digitalRead(pb)) digitalWrite(led_green, HIGH);
11   else digitalWrite(led_green, LOW);
12 }
```

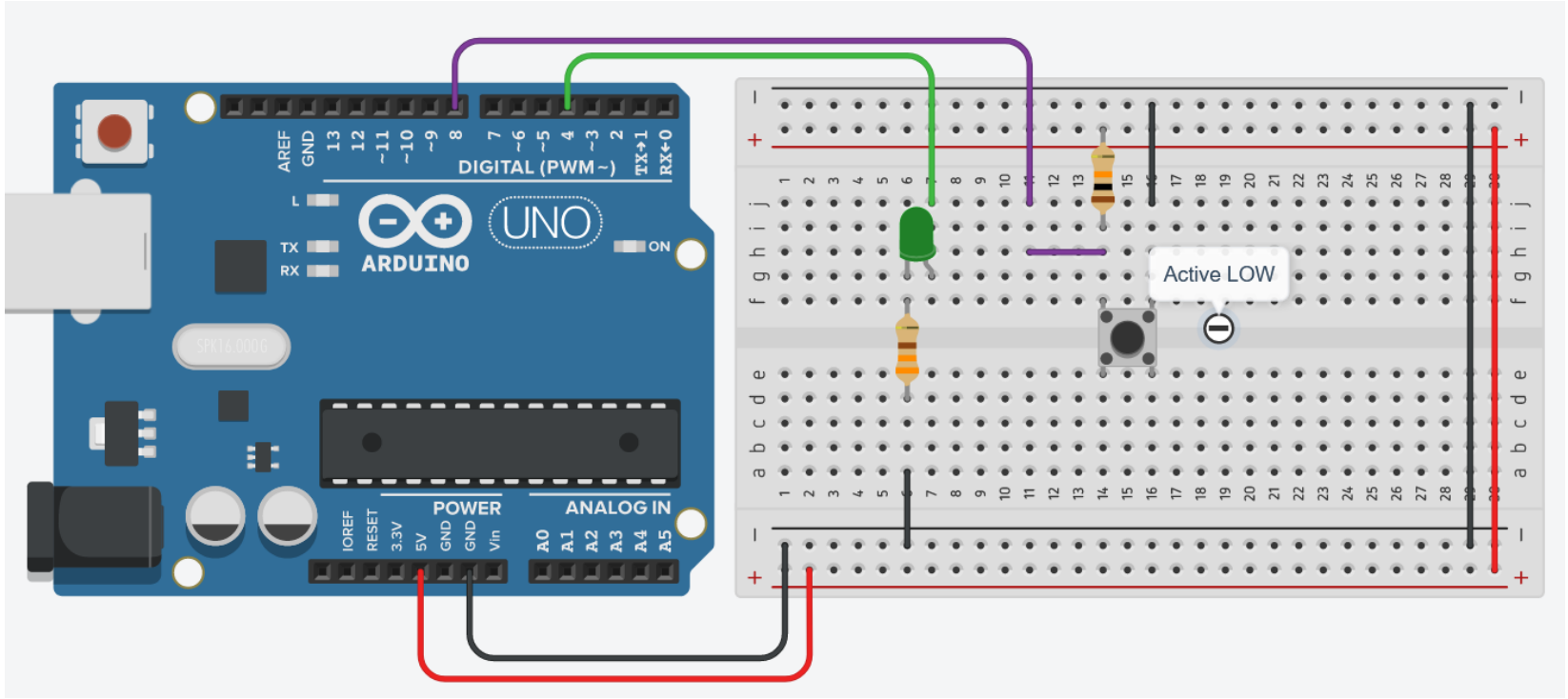
1. No need to save the state of the switch in a variable

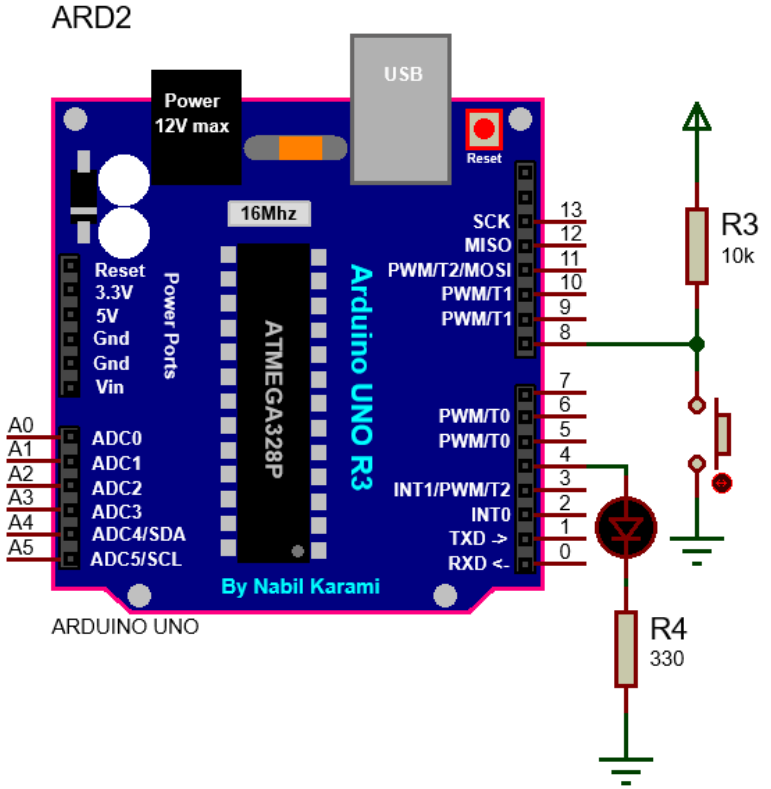
2. No need to write
If (digitalRead(pb) == true)



Code 5: Led with Push button (Active LOW PB)

In this case, the push button delivers 0 volt (0 logic) when pressed, the resistor now is called Pull-up and force the pin8 to be 1 logic (5 volt) is the push button is released.







Code 5: Led with Push button (Active LOW PB)

The code now tests if the level is 0 logic (PB is pressed) by using the !digitalRead(pb)
The "!" represents the NOT sign

```
1  #define pb  8
2  #define led_green 4
3
4  void setup() {
5    pinMode(pb, INPUT);
6    pinMode(led_green, OUTPUT);
7  }
8
9  void loop() {
10   if (!digitalRead(pb)) digitalWrite(led_green, HIGH);
11   else digitalWrite(led_green, LOW);
12 }
```

If (digitalRead(pb) == false)

Is same as

If (!digitalRead(pb))

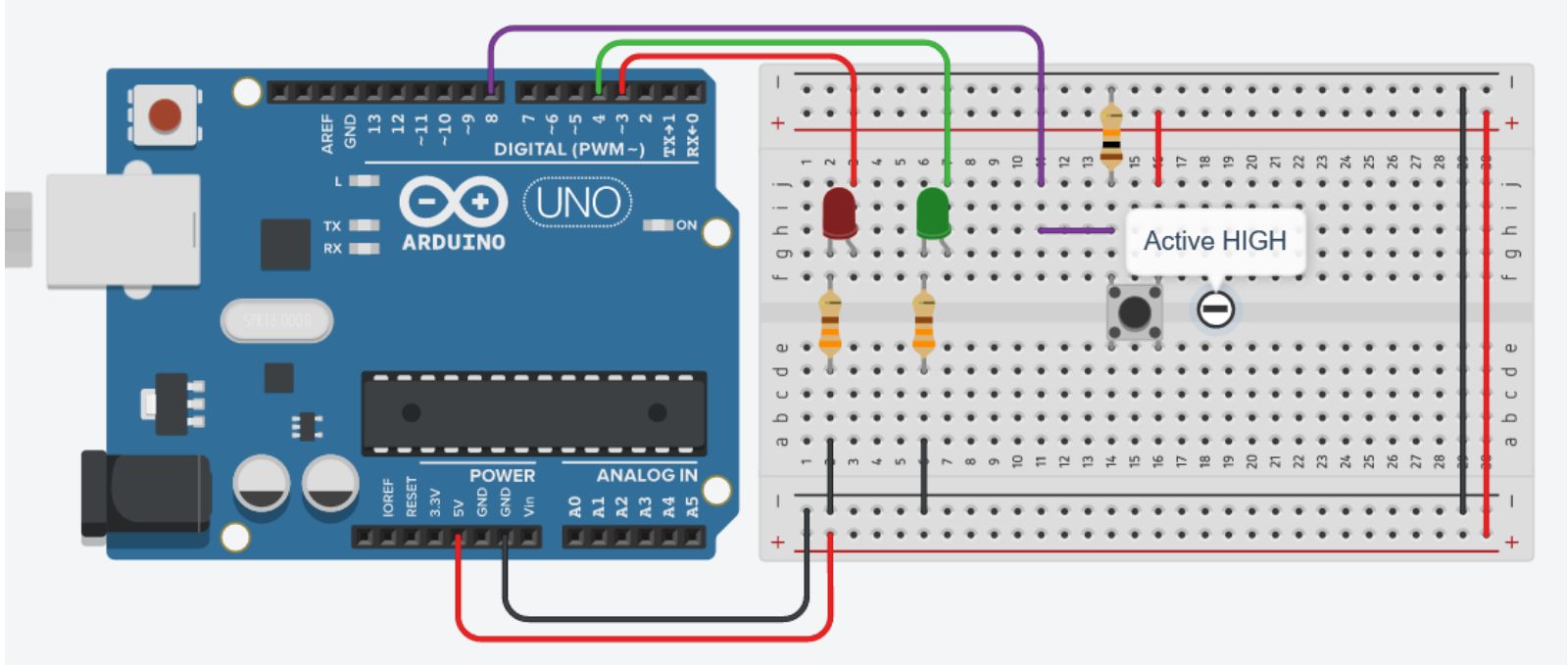


Code 6: Two task code

In this experiment, we want to merge the two previous examples in one code.

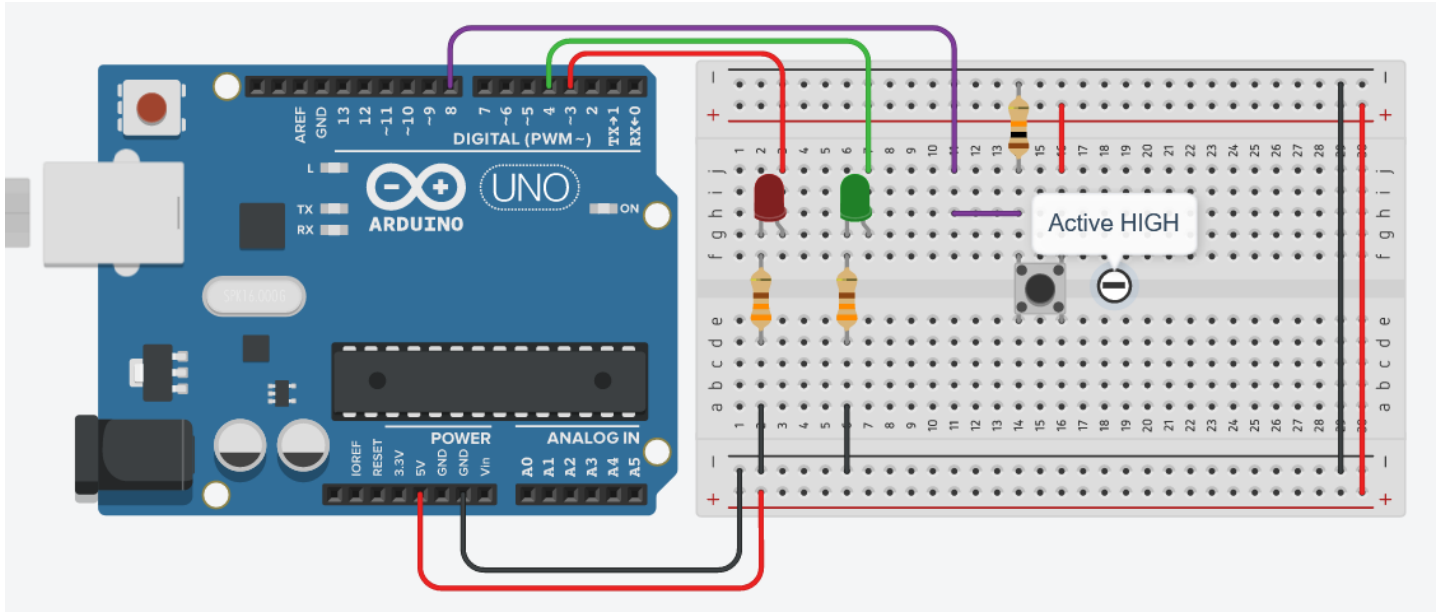
- Blink the red led
- Turn on the green led when the pb is pressed

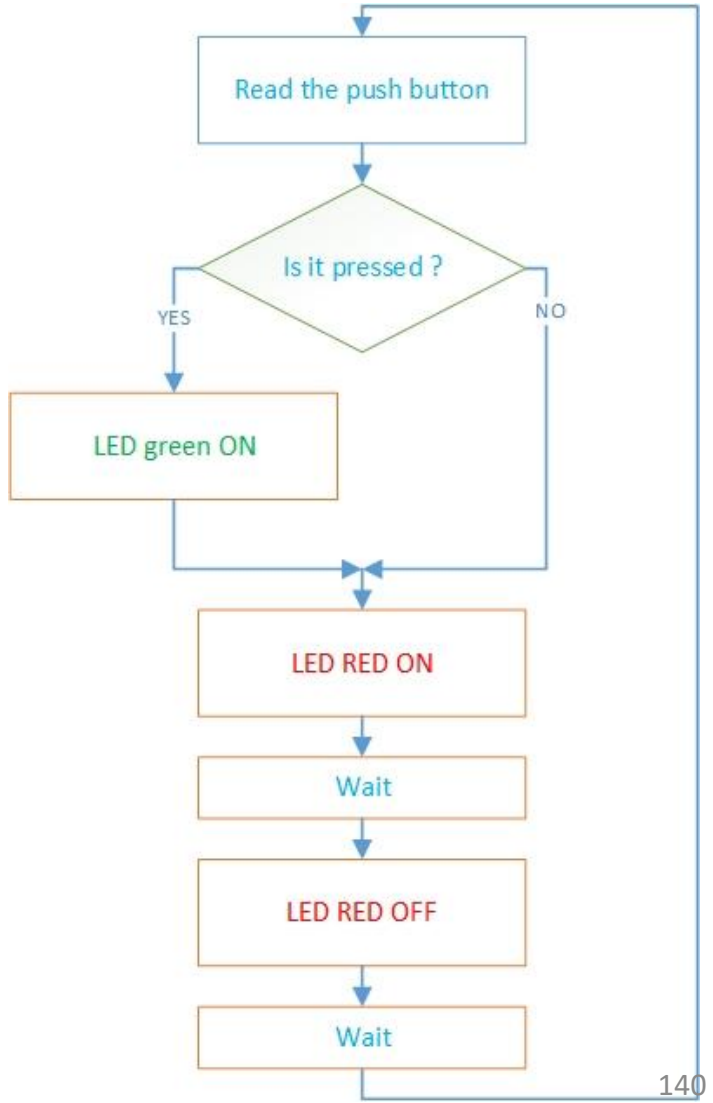
- The code is designed to be as simple as possible





Code 6: Two-task code (Polling PB)

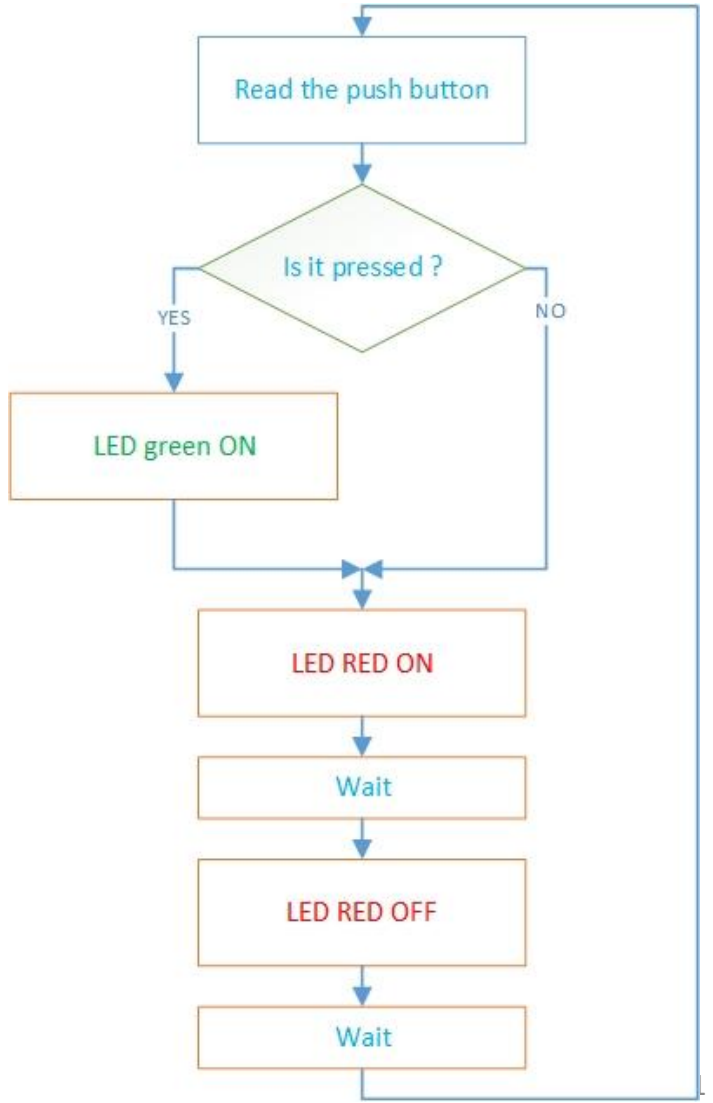






Code 6: Two-task code (Polling PB)

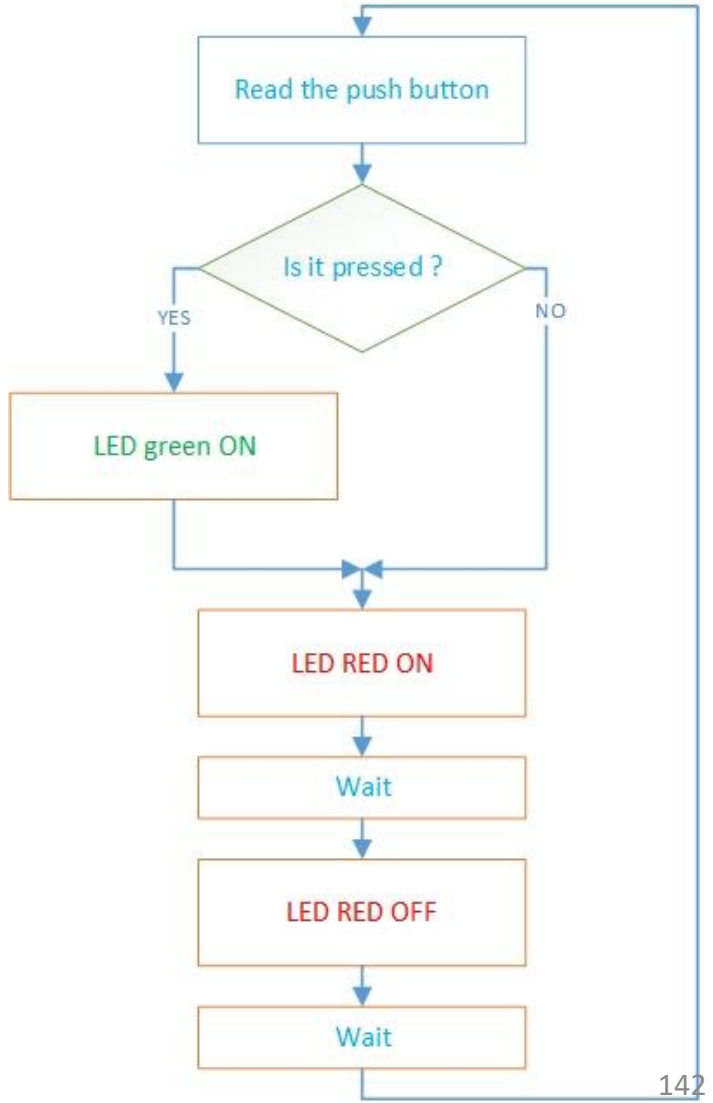
```
1  #define pb  8
2  #define led_red 3
3  #define led_green 4
4
5  void setup() {
6    pinMode(pb, INPUT);
7    pinMode(led_red, OUTPUT);
8    pinMode(led_green, OUTPUT);
9  }
10
11 void loop() {
12   if (digitalRead(pb)) digitalWrite(led_green, HIGH);
13   digitalWrite(led_red, HIGH);
14   delay(1000);
15   digitalWrite(led_red, LOW);
16   delay(1000);
17 }
```





Code 6: Two-task code (Polling PB)

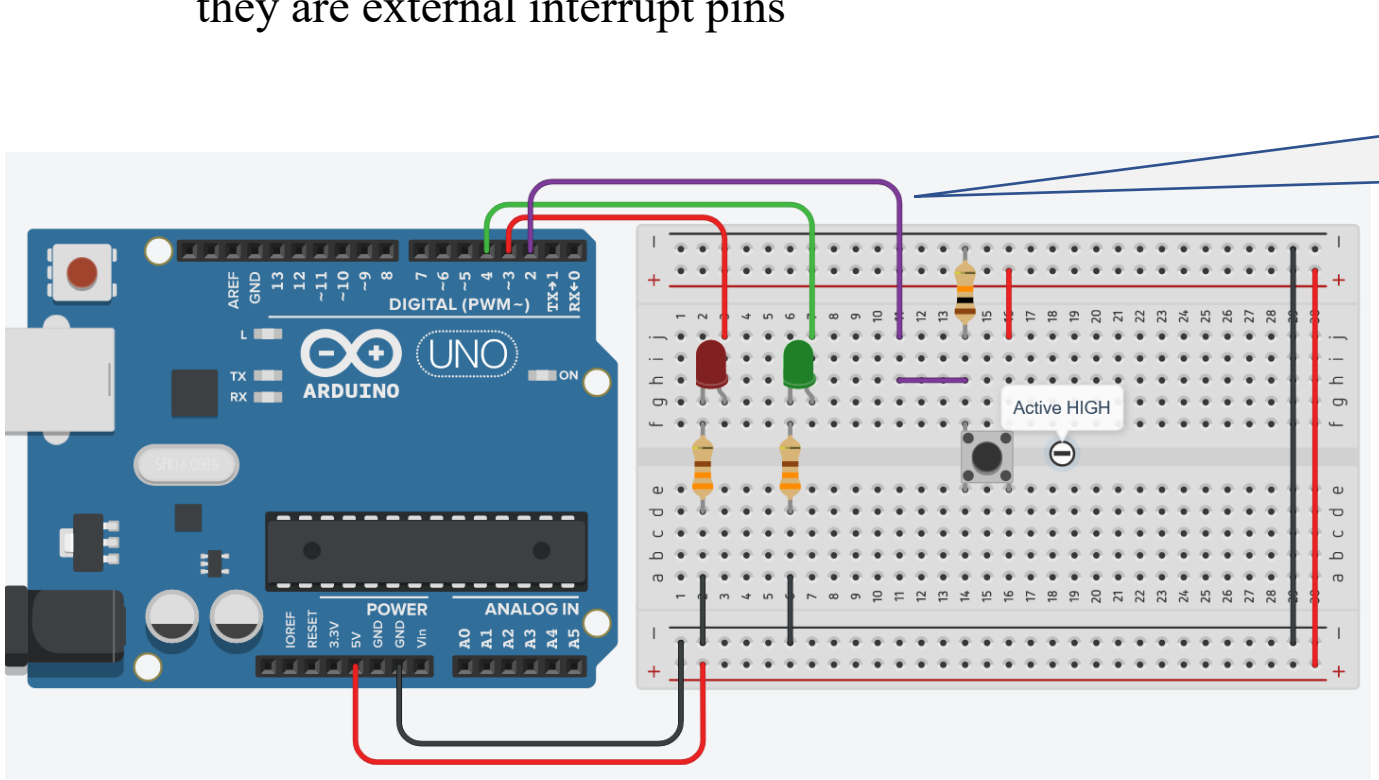
- It is clearly observed that the green LED will not go ON directly when the push button is pressed.
- This is because the flow of execution of the code is achieved line by line.
- In other words, the code may be busy in the delay functions and may not check the push button when clicked.
- The worst case response time is in this case 2 seconds.
- The best case (fastest response) is when the push button is clicked exactly on the if condition line.

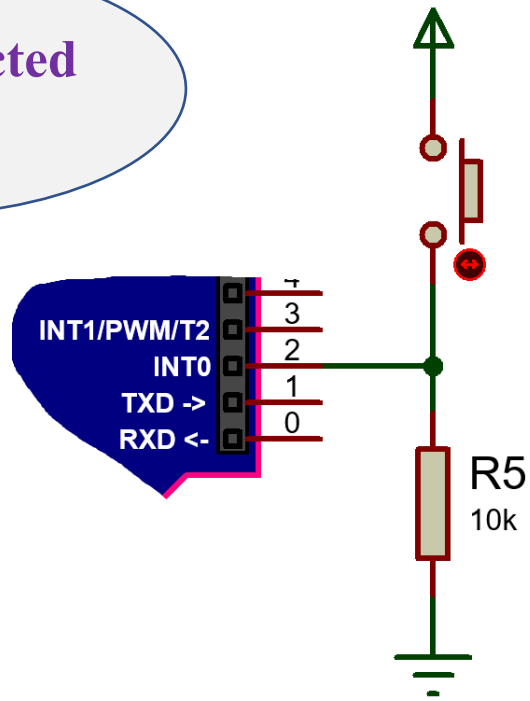




Code 7: Two-task code (Using interrupt)

- In this experiment, we will introduce the **interrupt** event that is executed when the push button is pressed
- Will keep on the same connection, but we have to change the push button connection to **pin 2 or 3** in which they are external interrupt pins







Code 7: Two-task code (Using interrupt)

- Interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention.
- It alerts the processor to a high priority process requiring interruption of the current working process.
- There are software and Hardware interrupts :
  - **Hardware Interrupts** - These occur in response to an external event, like a pin going high or low.
  - **Software Interrupts** - These occur in response to a software instruction.

• When the event or interrupt happens, the processor takes immediate notice, **saves its execution state**, runs a small chunk of code (often called the **interrupt handler** or **interrupt service routine**), and then returns back to whatever it was doing before.



Code 7: Two-task code (Using interrupt)

In Arduino function called *attachInterrupt()* to do this task, its syntax looks like

attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)

- **digitalPinToInterrupt(pin):** Pin number of the interrupt, which tells the microprocessor which pin to monitor.
- **ISR:** The location of code we want to execute if this interrupt is triggered.
- **mode :** defines when the interrupt should be triggered. Four constants are predefined as valid values:
  - **LOW** to trigger the interrupt whenever the pin is low,
  - **CHANGE** to trigger the interrupt whenever the pin changes
  - **RISING** to trigger when the pin goes from low to high,
  - **FALLING** for when the pin goes from high to low.





Code 7: Two-task code (Using interrupt)

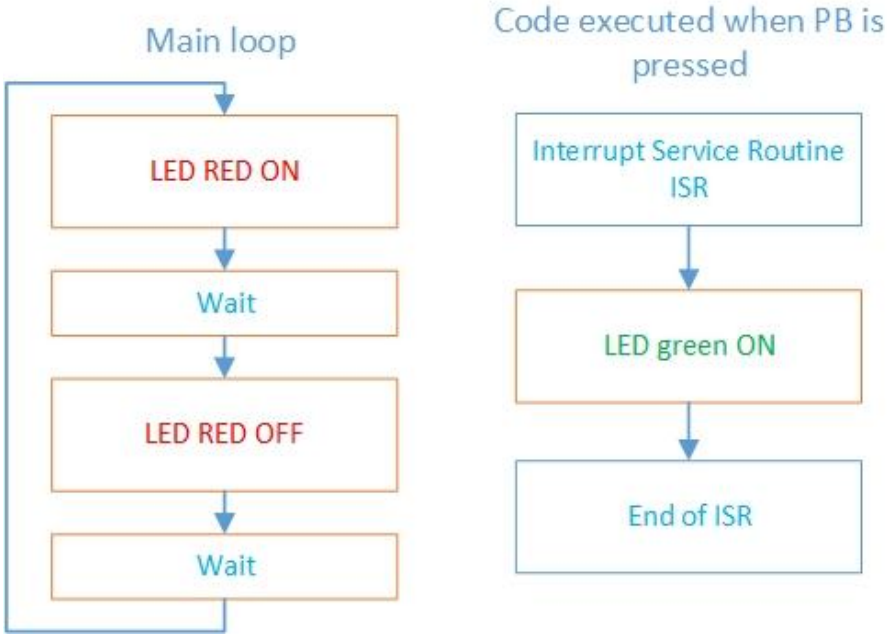
Interrupt Pins

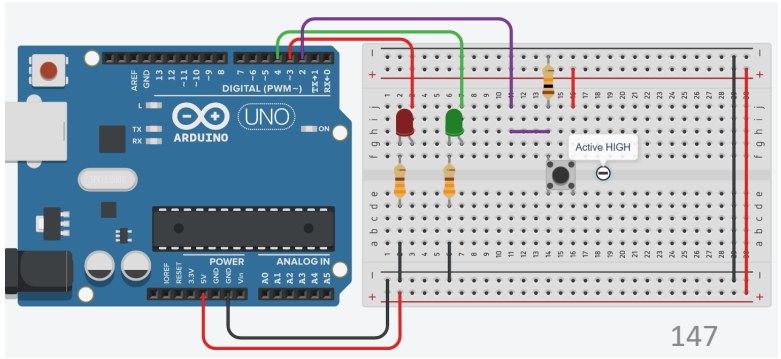
BOARD	INT0	INT1	INT2	INT3	INT4	INT5	...	INT15
UNO	Pin 2	Pin 3						
MEGA	Pin 2	Pin 3	Pin 21	Pin 20	Pin 19	Pin 18		
ESP8266	GPIO 0	GPIO 1	GPIO 2	GPIO 3	GPIO 4	GPIO 5	...	GPIO 15



Code 7: Two-task code (Interrupt)

```
1  #define pb  2
2  #define led_red 3
3  #define led_green 4
4
5  void setup() {
6    pinMode(pb, INPUT);
7    pinMode(led_red, OUTPUT);
8    pinMode(led_green, OUTPUT);
9    attachInterrupt(digitalPinToInterrupt(pb), turn_on_led_fct, RISING);
10 }
11
12 void loop() {
13   digitalWrite(led_red, HIGH);
14   delay(1000);
15   digitalWrite(led_red, LOW);
16   delay(1000);
17 }
18
19 // interrupt fucntion
20 void turn_on_led_fct() {
21   digitalWrite(led_green, HIGH);
22 }
```







Week 4

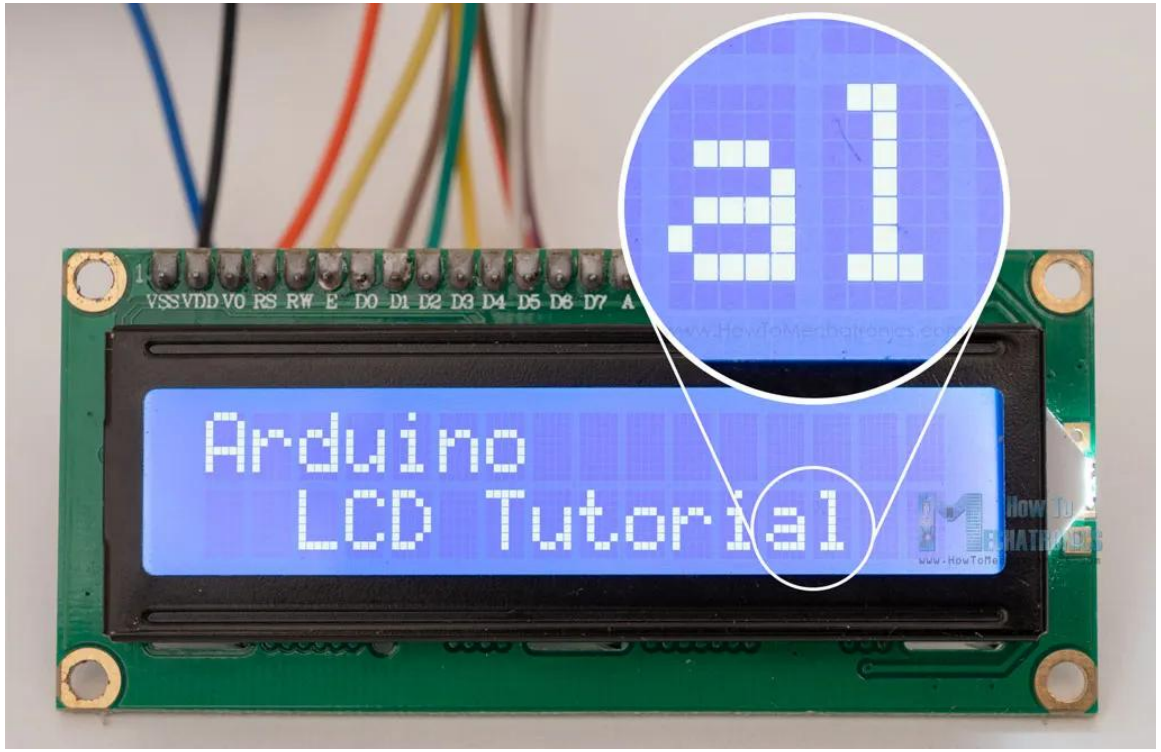
Lecture 4 / LO3

LCD and UART



Working with LCD

- An LCD character display is a unique type of display that can only output individual ASCII characters with fixed size.
- Closer look at the display we can notice that there are small rectangular areas composed of 5×8 pixels grid. Each pixel can light up individually, and generates characters within each grid.

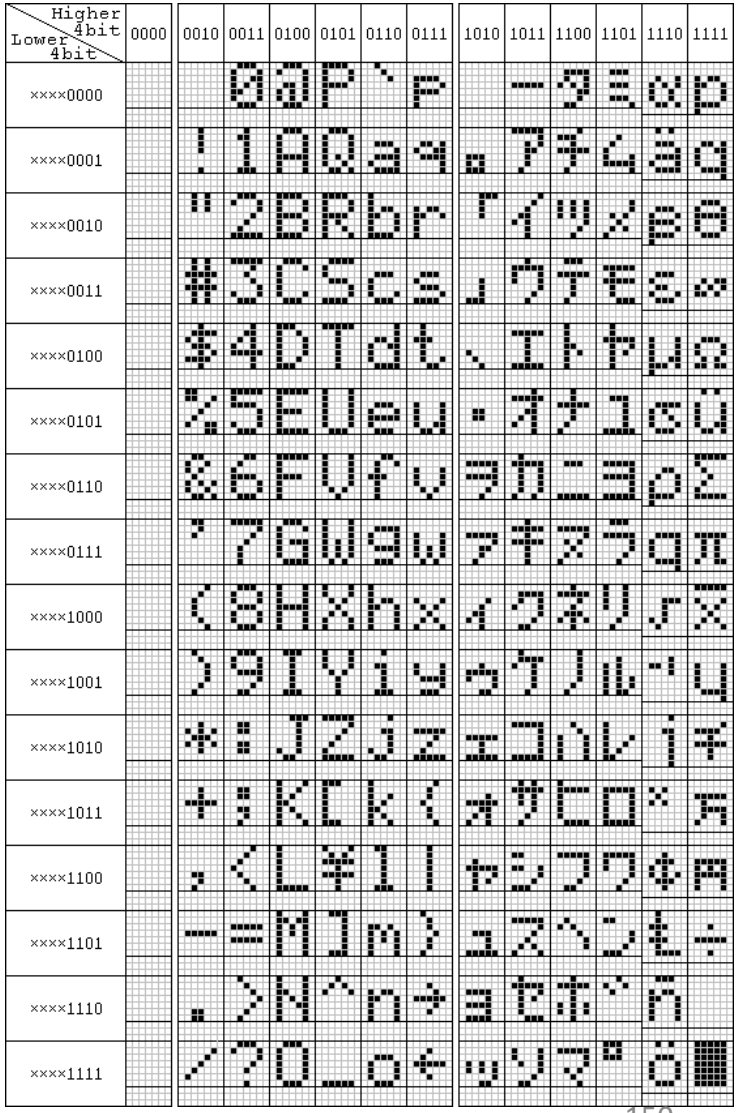




Working with LCD

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]





Working with LCD

- The number of the rectangular areas define the size of the LCD.
- The most popular LCD is the 16×2 LCD, which has two rows with 16 rectangular areas or characters.
- There are other sizes like 8x1, 8x2, 16×1, 16×4, 20×4, 40x4 and so on, but they all work on the same principle. Also, these LCDs can have different background and text color.



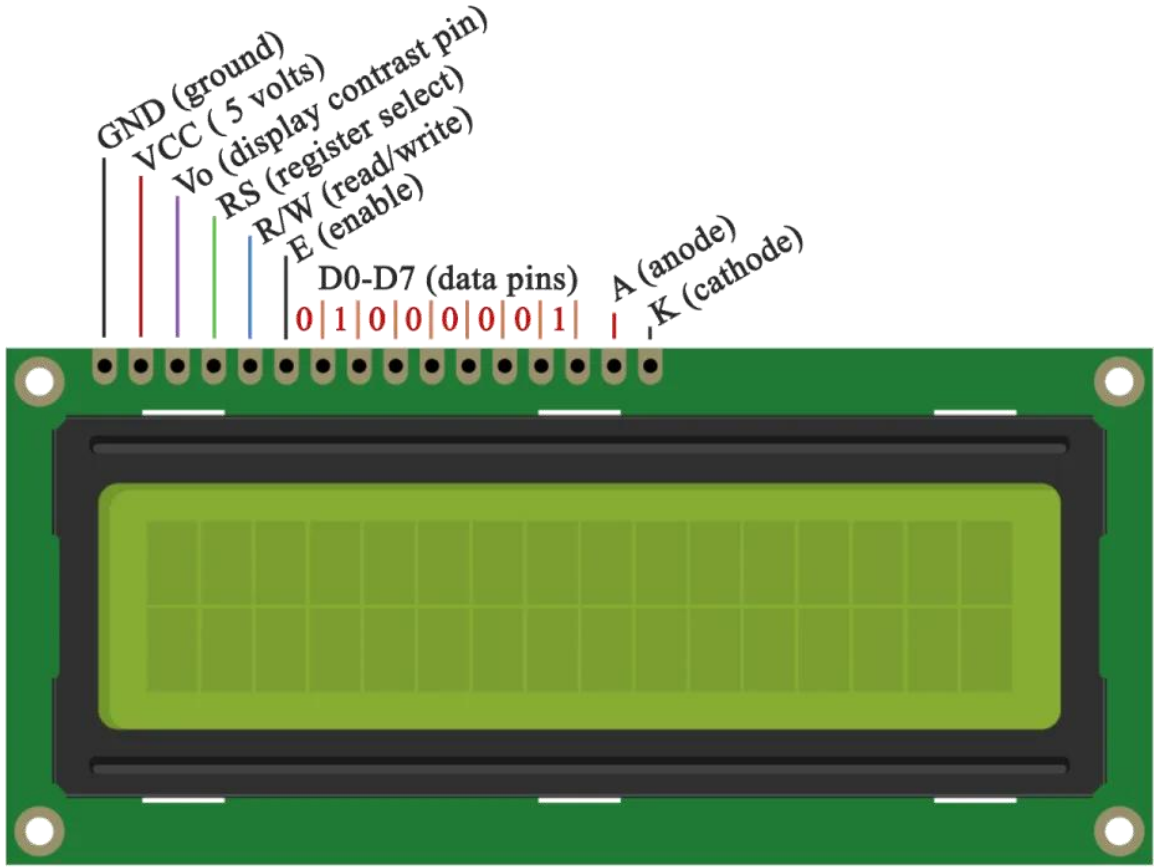




Working with LCD

LCD Pinout

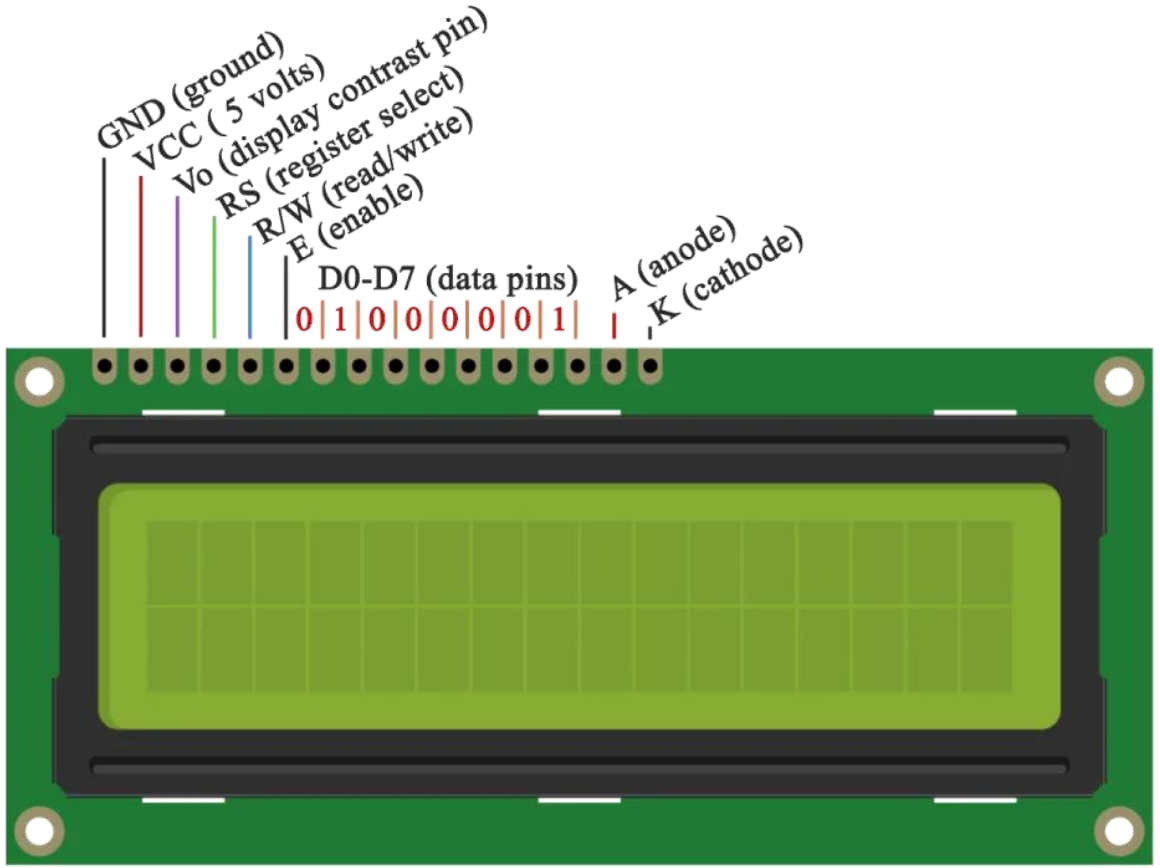
- It has 16 pins and the first one from left to right is the ***Ground*** pin.
- The second pin is the ***VCC*** which we connect the 5 volts pin on the Arduino Board.
- Next is the ***Vo*** pin on which we can attach a potentiometer for controlling the contrast of the display.





Working with LCD

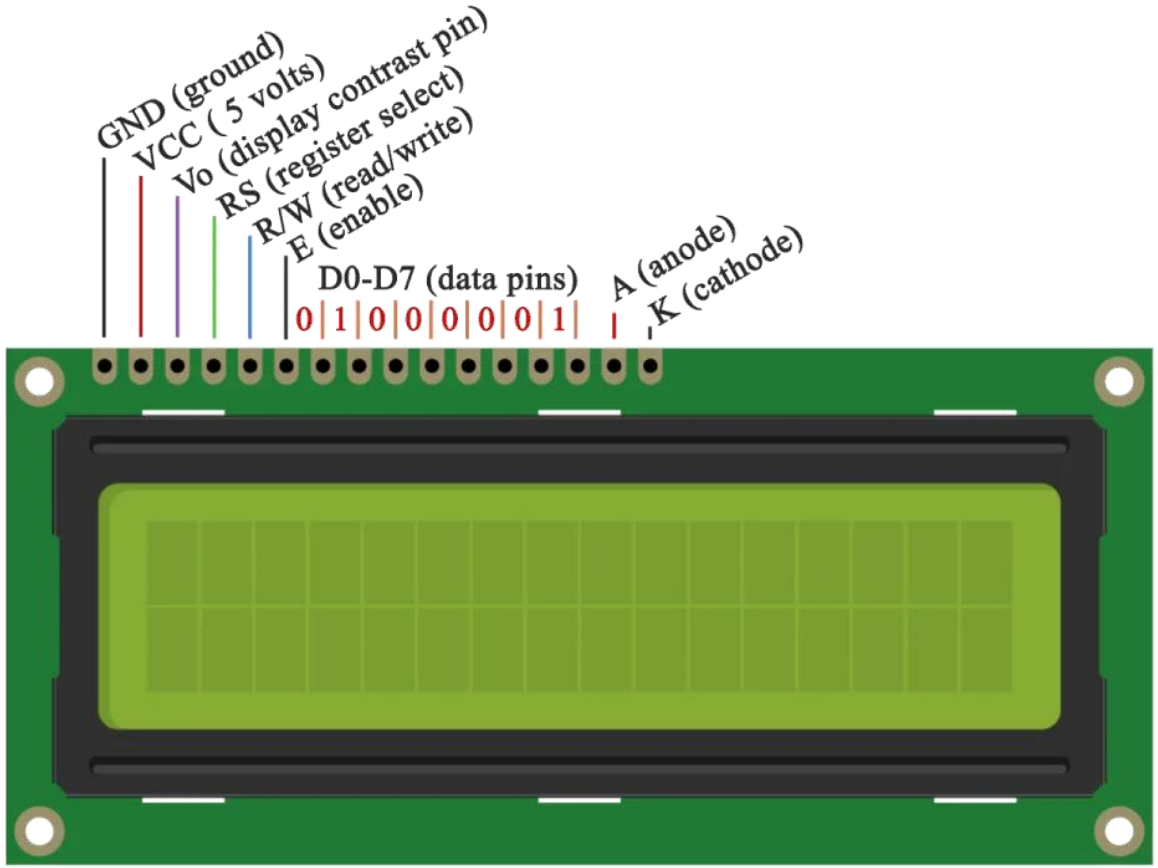
- ***RS*** pin or register select pin is used for selecting whether we will send commands or data to the LCD.
- For example if the *RS* pin is set on low state or zero volts, then we are sending commands to the LCD like: set the cursor to a specific location, clear the display, turn off the display and so on.
- And when *RS* pin is set on High state or 5 volts we are sending data or characters to the LCD.





Working with LCD

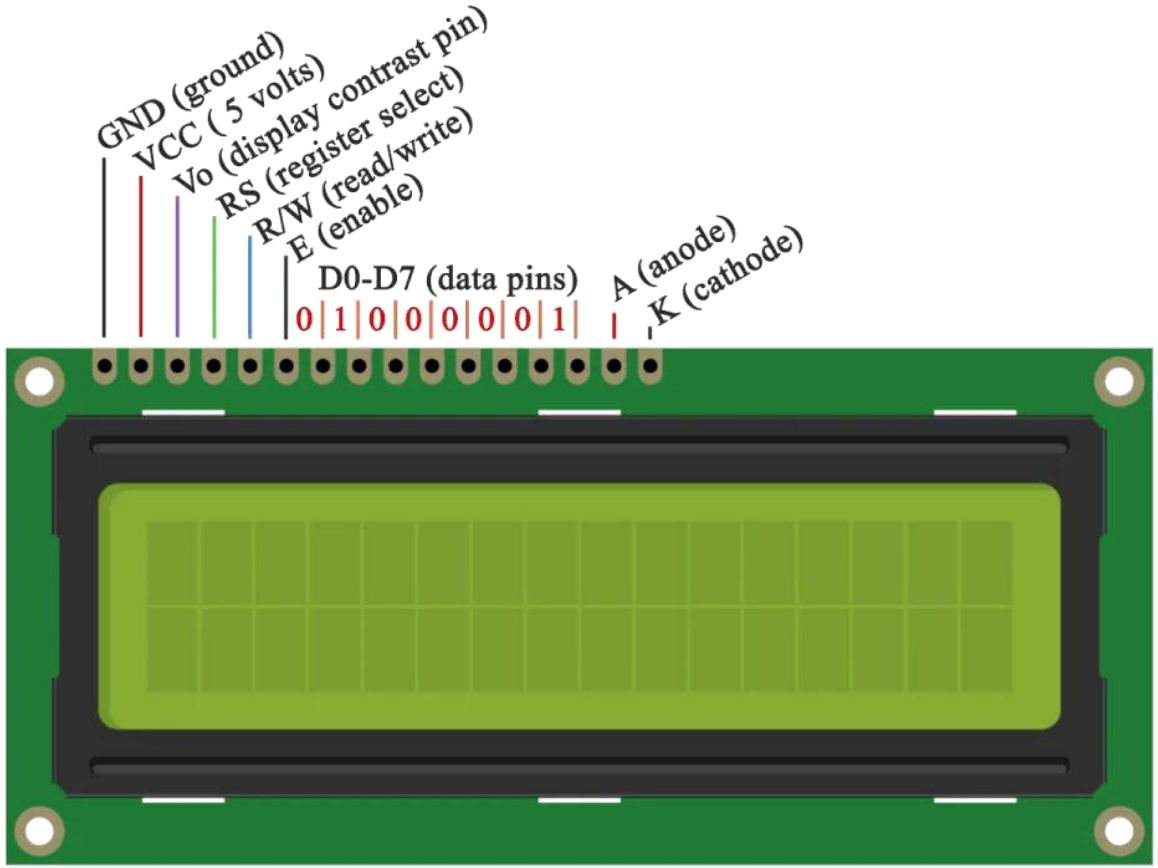
- ***R/W*** pin which selects the mode whether we will read or write to the LCD.
- Here the write mode is obvious and it is used for writing or sending commands and data to the LCD.
- The read mode is used to read the CGRAM and the busy flag





Working with LCD

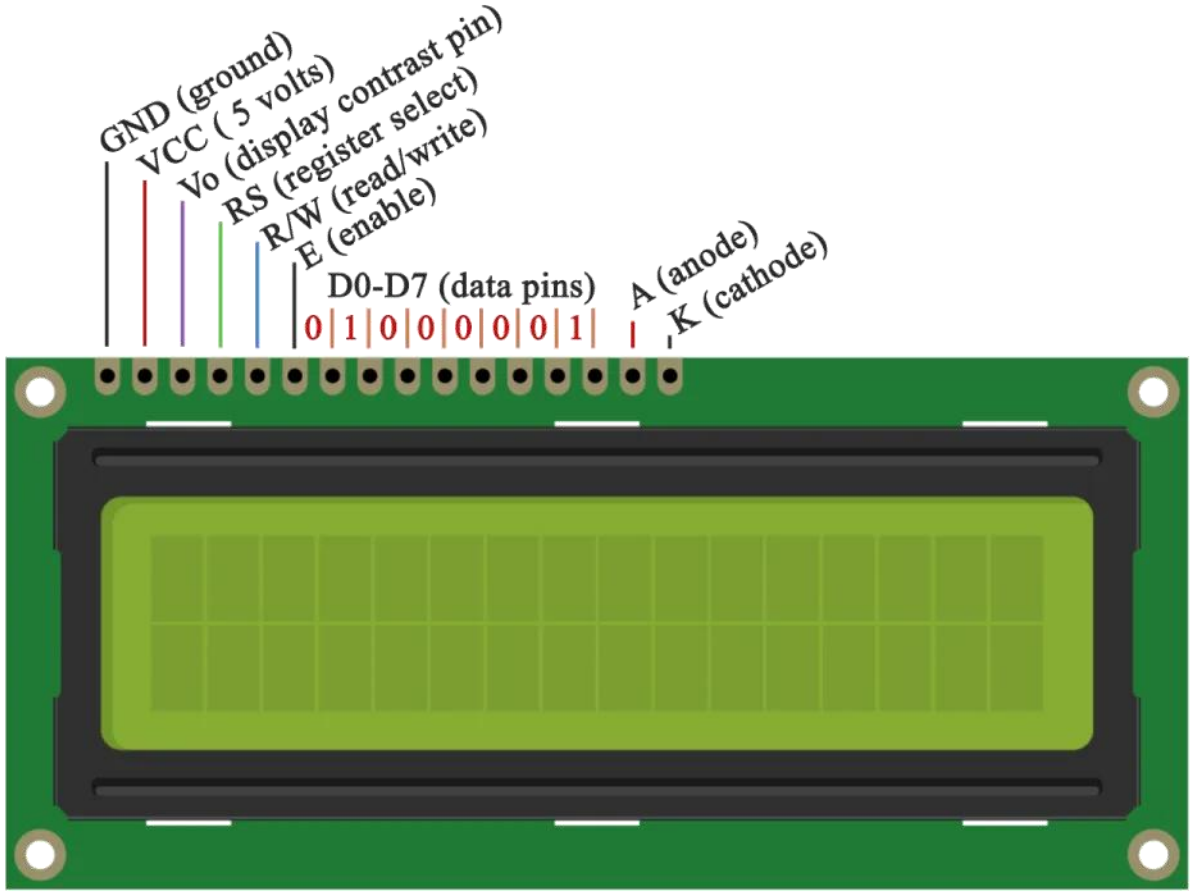
- ***E*** pin enables the writing to the registers, or the next 8 data pins from D0 to D7.
- So through this pins we are sending the 8 bits data when we are writing to the registers or for example if we want to see the letter uppercase A on the display we will send 0100 0001 to the registers according to the ASCII table.





Working with LCD

- **LED Backlight (*A* and *K*)** pins that are used to turn on and off the LED backlight.
- A resistor should be connected in series to protect the LED. The value of the resistor can be in a range of 100 to 470 Ohm (when supplied by a 5V source)

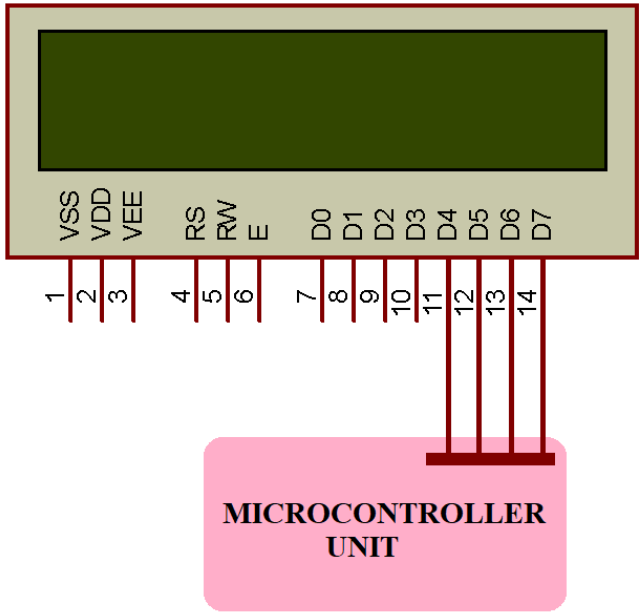




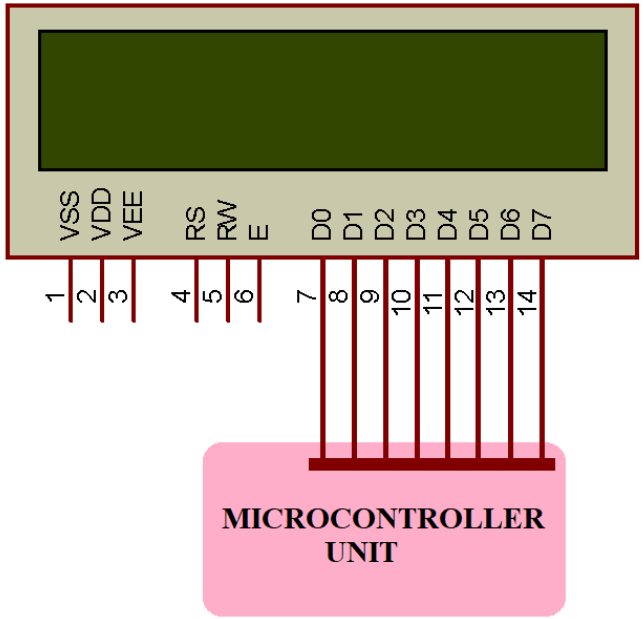
Working with LCD

The Hitachi-compatible LCDs can be controlled in two modes:

- 4-bit or 8-bit.



LCD : 4 BIT MODE



LCD : 8 BIT MODE



Working with LCD

- The LiquidCrystal library allows to control LCD displays that are compatible with the Hitachi HD44780 driver.

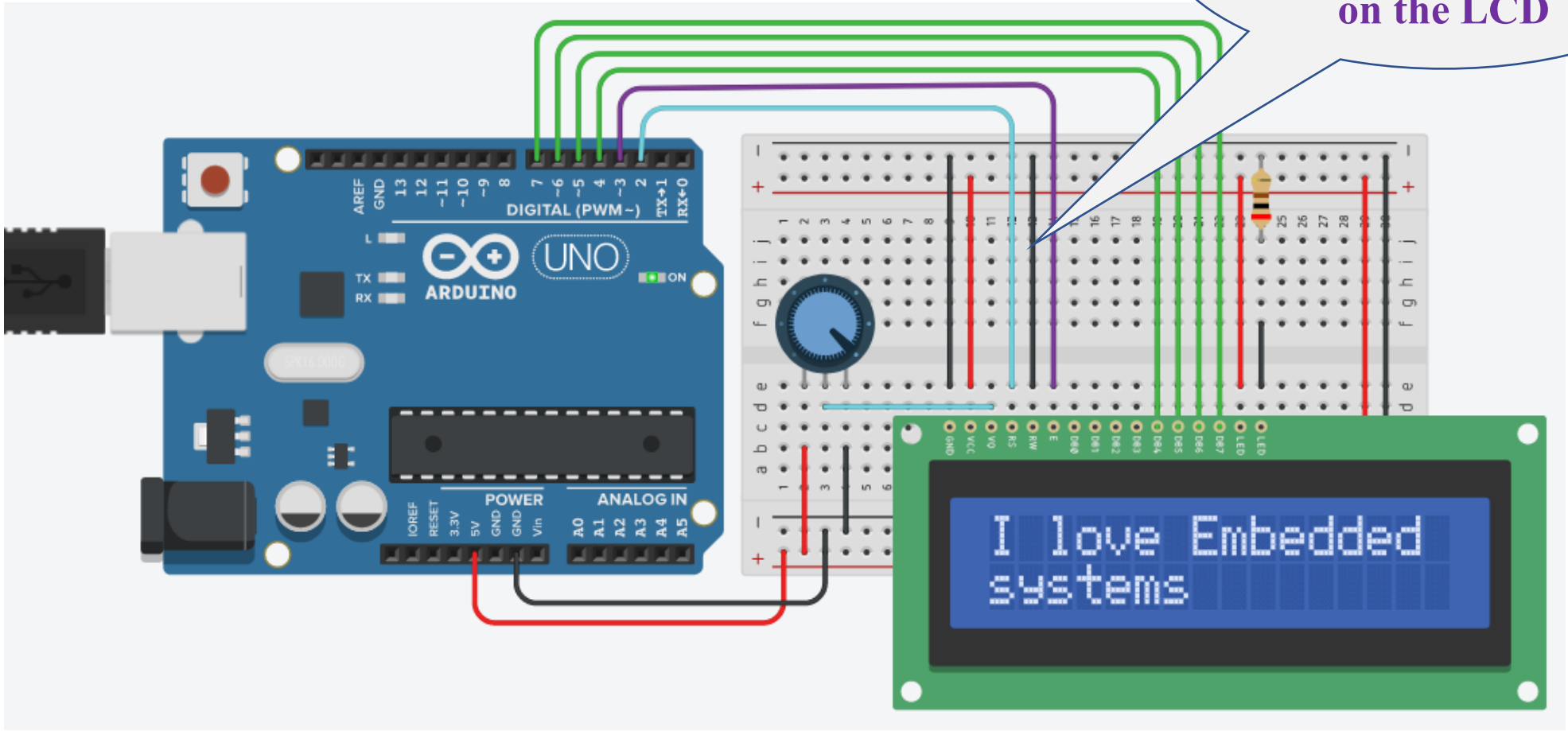






Connecting LCD to Arduino

The R/W pin is always connected to ground for writing on the LCD





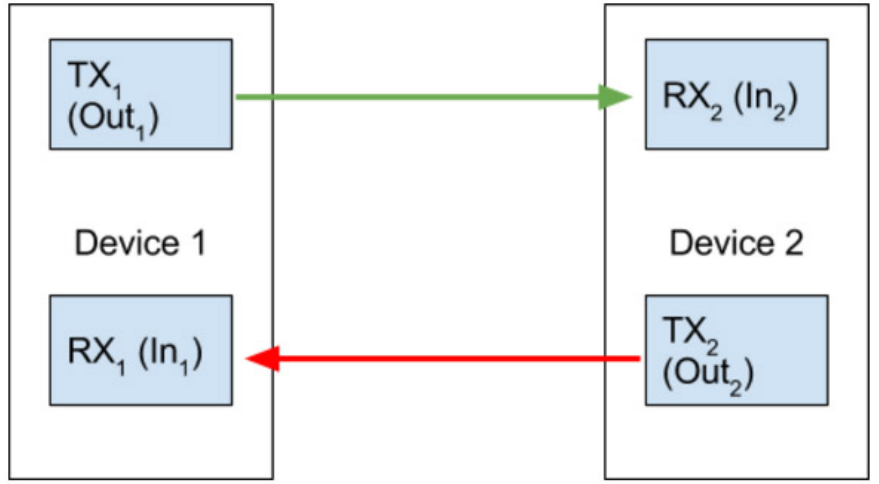
UART Communication Protocol

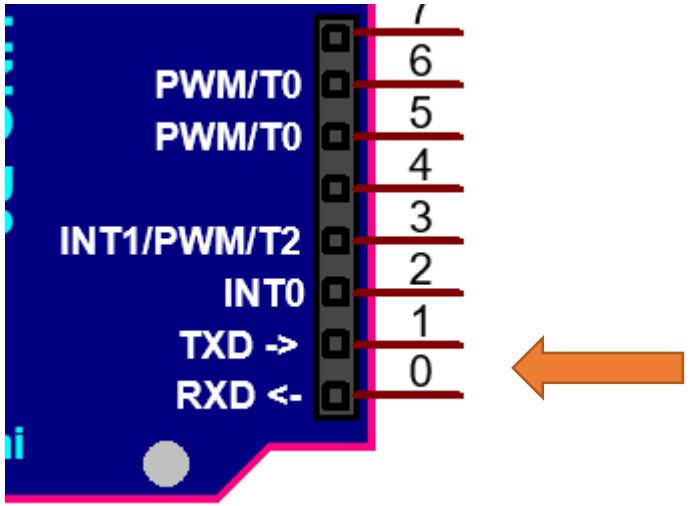
- In a digital signal, data is transferred as a sequence of *high to low* and *low to high* switching that occur very rapidly.
- These highs and lows in a digital signal represent 1s and 0s respectively that, when put together in sequence, carry information that can be interpreted by microcontrollers.
- Individually, these 1s and 0s are *bits*, and when they are in groups of 8, they are called *bytes*!
- A byte might look something like this: 10111001



UART Communication Protocol

- The first communication protocol we'll cover is *Universal Asynchronous Receiver/Transmitter (UART)*.
- UART is a form of *serial communication* because data is transmitted as sequential bits (we'll get to this in a bit).
- The wiring involved with setting up UART communication is very simple: one line for transmitting data (TX) and one line for receiving data (RX).

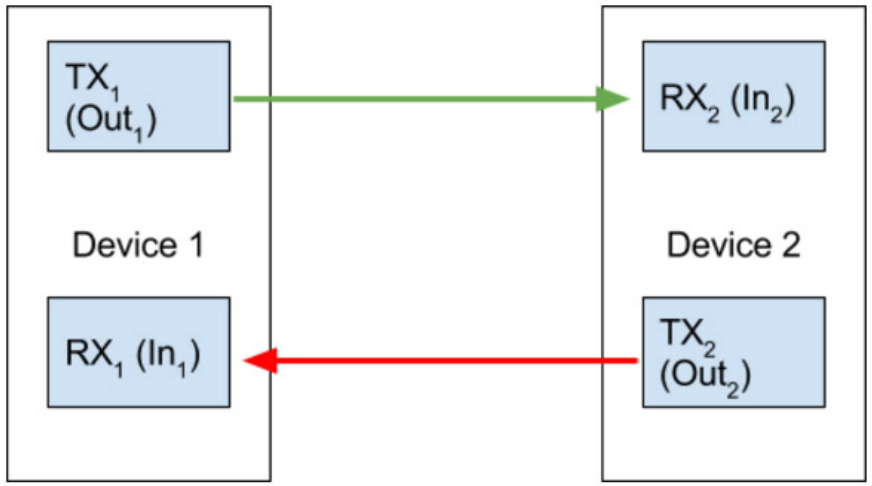


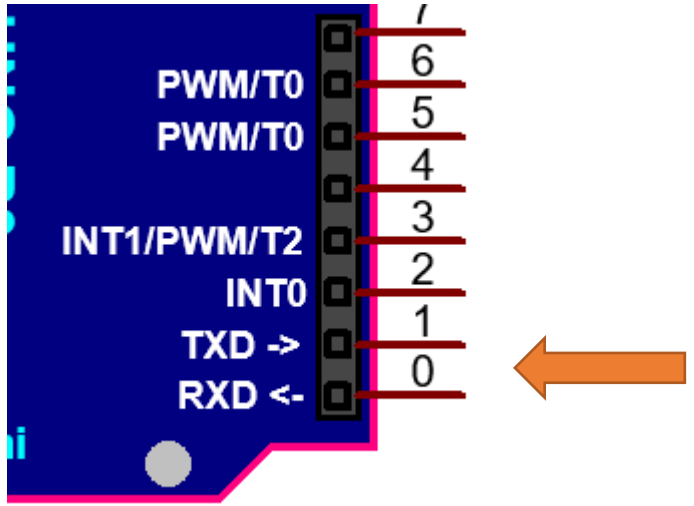




UART Communication Protocol

- TX line is used to for the data to send device, and the RX line is used to receive data.
- Together the TX and RX lines of a device using serial communication form a *serial port* through which communication can occur.







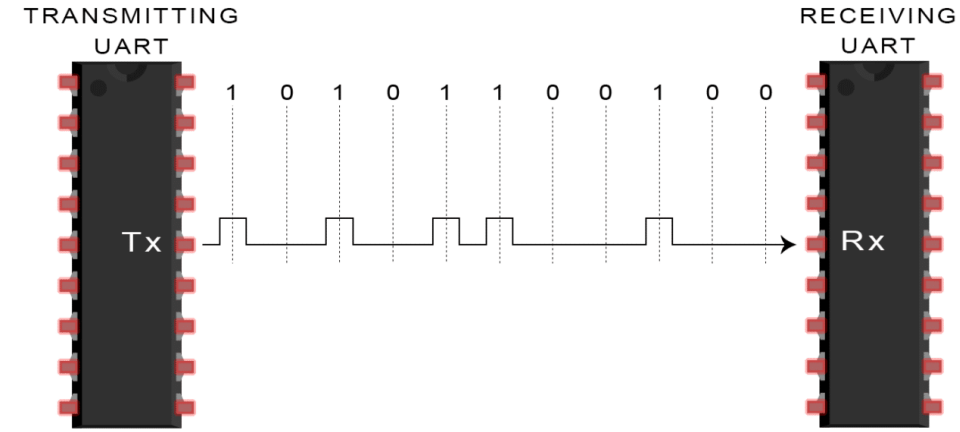
UART Communication Protocol

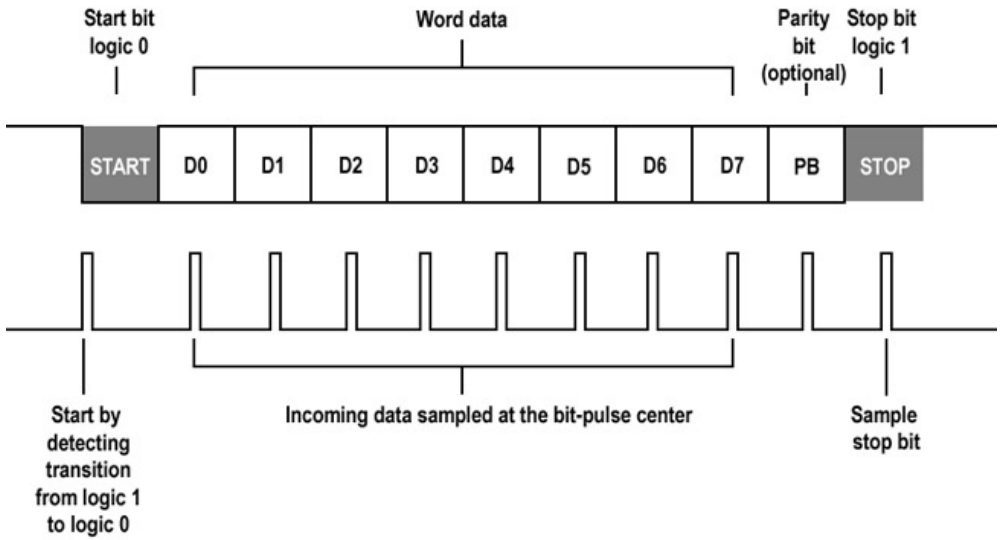
- The term *UART* actually refers to the onboard hardware that manages the packaging and translation of serial data.
- For a device to be able to communicate via the UART protocol, it must have this hardware!
- On the Arduino Uno, there is one serial port dedicated for communication with the computer the Arduino is connected to.
- USB, which stands for *Universal Serial Bus*, is a serial port!
- On the Arduino Uno, this USB connection is broken out through onboard hardware into two digital pins, *GPIO 0* and *GPIO 1*, which can be used in projects that involve serial communication with electronics other than the computer.



UART Communication Protocol

- UART is called *asynchronous* because the communication does not depend on a synchronized clock signal between the two devices attempting to communicate with each other.
- Because the communication speed is not defined via this steady signal, the "sender" device cannot be certain that the "receiver" obtains the correct data.
- Therefore, the devices break data into fixed-size chunks to ensure that the data received is the same as the data that was sent.

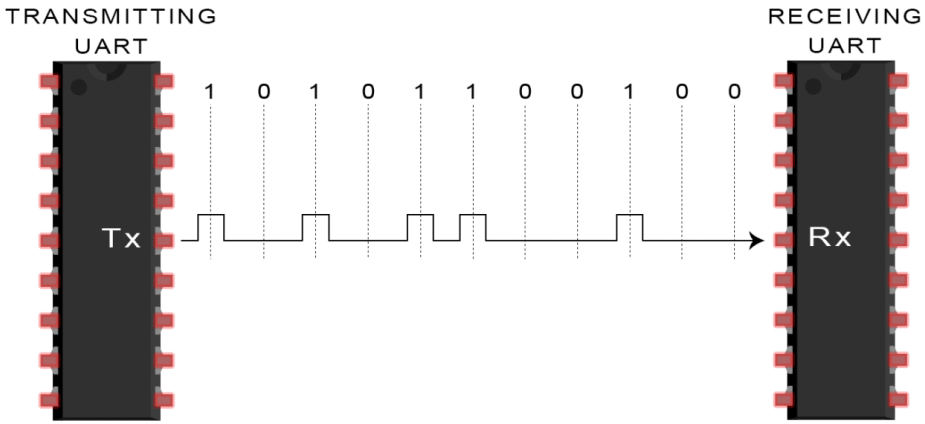


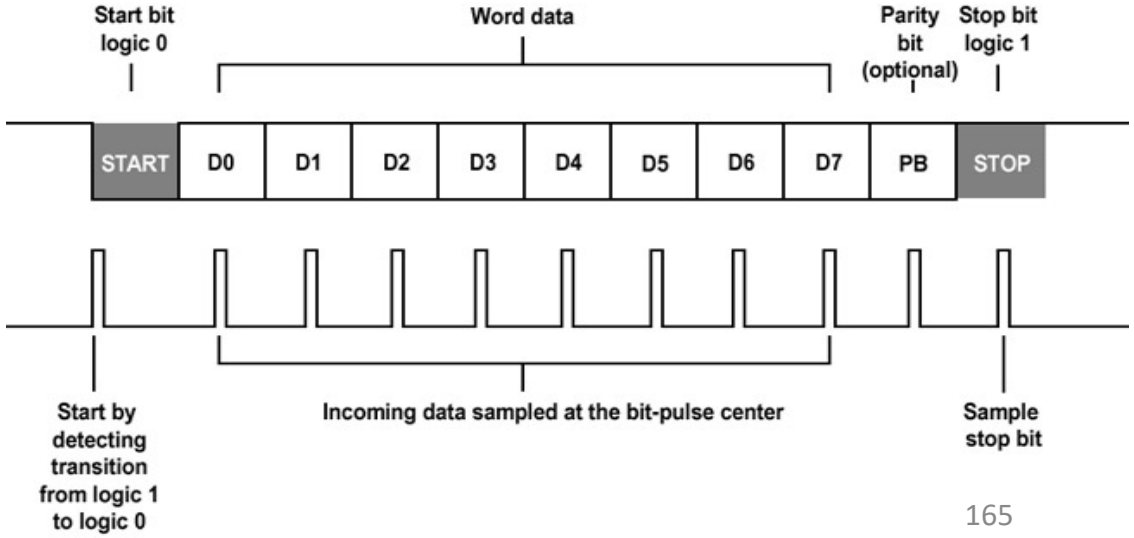




UART Communication Protocol

- Devices that communicate via UART send packets of pre-defined size that contain additional information regarding the start and end of the message and confirmation of whether the message was received correctly.
- For example, to begin communication, the transmitting device pulls the transmit line low, indicating the start of a data packet.

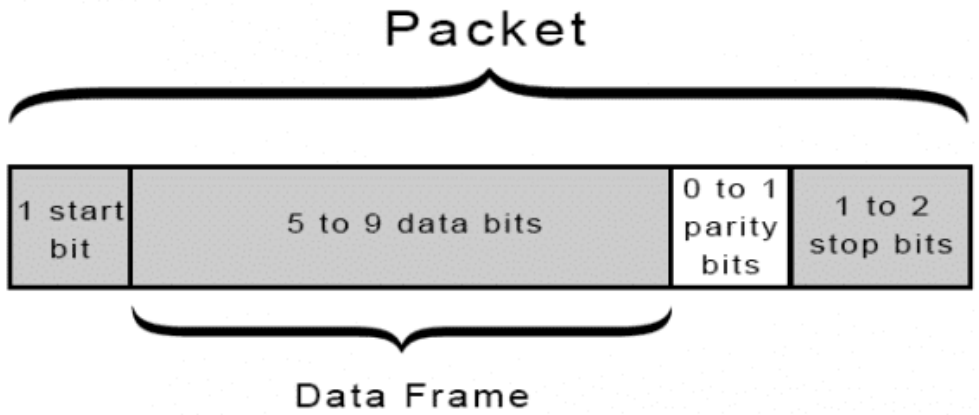






UART Communication Protocol

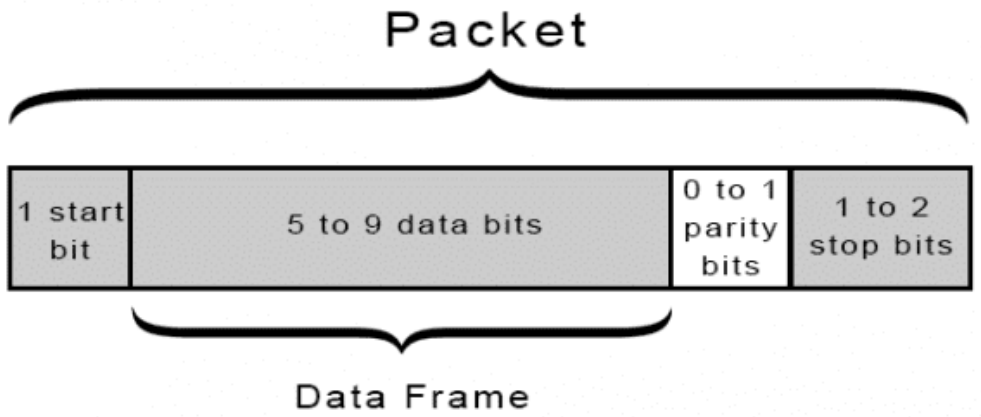
- UART has settings that need to be the same on both devices to have proper communication.
- These UART settings are the *baud rate*, *data length*, *parity bit*, *number of stop bits,* and *flow control*.





UART Communication Protocol

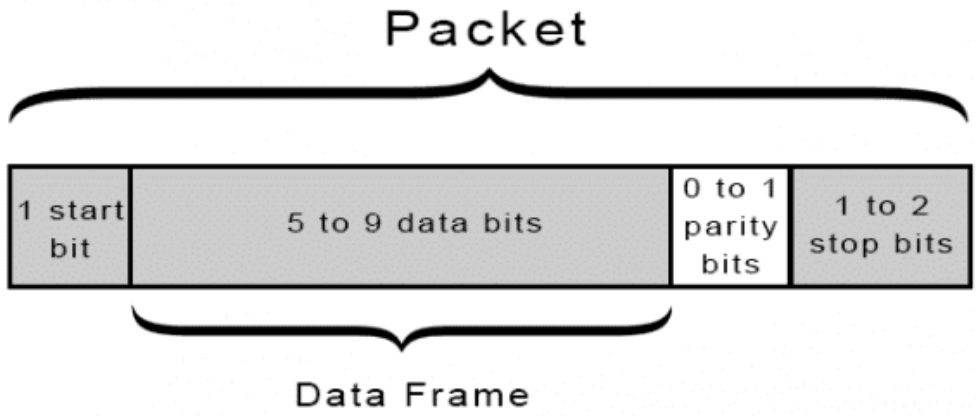
- **Baud Rate:** is the number of *bits per second (bps)* a UART device can transmit/receive.
- We need to set both UART devices with the same baud rate to have the proper transmission of data. Common values for baud rate are 9600, 1200, 2400, 4800 , 19200, 38400, 57600, and 115200 bps.





UART Communication Protocol

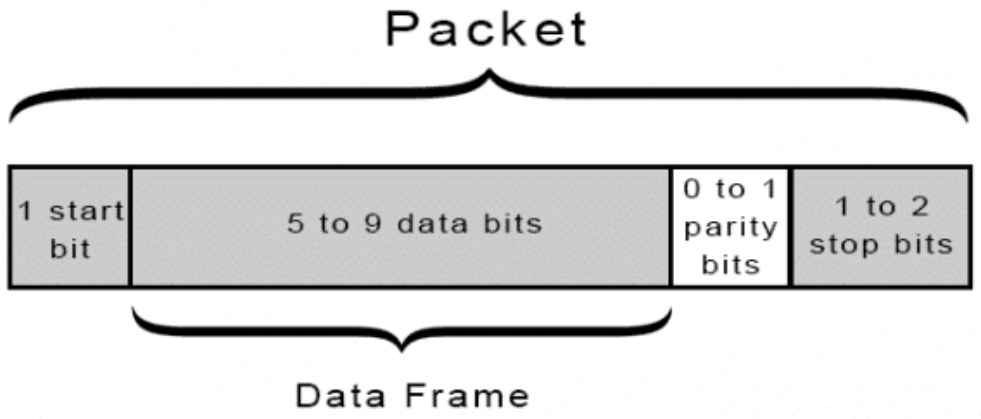
- **Data Length:** refers to the number of bits per byte of data.





UART Communication Protocol

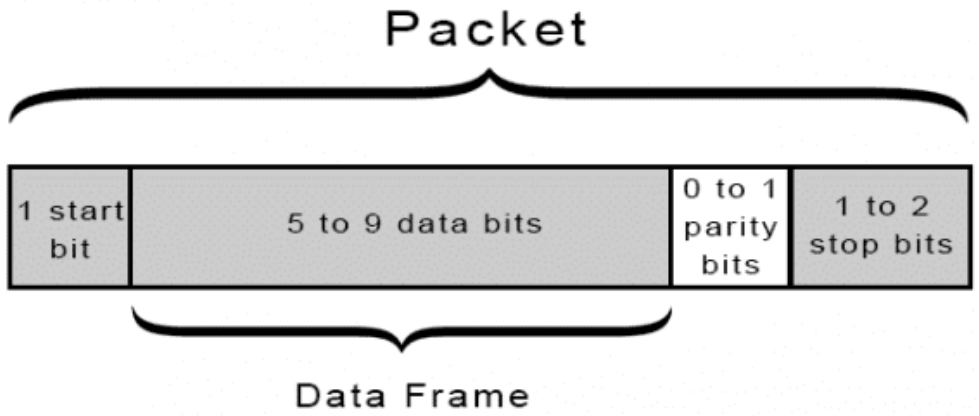
- **Parity Bit:** is a bit added to the transmitted data and tells the receiver if the number of 1's in the data transmitted is odd or even. The possible setting for Parity Bit is *Odd* or *Even*.
  - **ODD** – the parity bit is '1' if there is an *odd* number of 1's in the data frame
  - **EVEN** – the parity bit is '0' if there is an *even* number of 1's in the data frame





UART Communication Protocol

- **Number of Stop bits:** UART devices can use *one* or *two* stop bits to mark the end of a set of bits (called packets) transmitted.





UART Communication Protocol

- When implementing UART serial communication on most embedded platforms such as the Arduino, the user does not have to deal with communication at the bit level.
- Instead, the platform often provides higher level software libraries that are the only aspect of the communication process the user has to deal with.
- In the Arduino platform, users can use the *Serial* and *SoftwareSerial* libraries to implement UART communication for their projects.



UART Communication Protocol

- All Arduino boards have at least one serial port (also known as a UART or USART), and some have several.

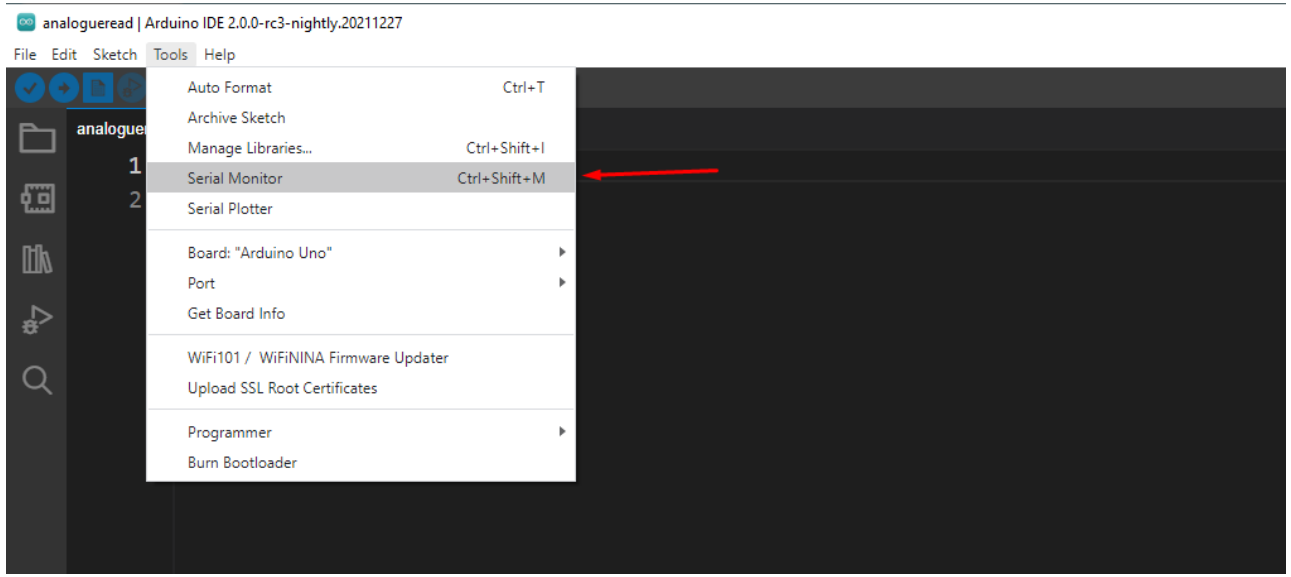
BOARD	USB CDC NAME	SERIAL PINS	SERIAL1 PINS	SERIAL2 PINS	SERIAL3 PINS
Uno, Nano, Mini		0(RX), 1(TX)			
Mega		0(RX), 1(TX)	19(RX), 18(TX)	17(RX), 16(TX)	15(RX), 14(TX)
Leonardo, Micro, Yún	Serial		0(RX), 1(TX)		
Uno WiFi Rev.2		Connected to USB	0(RX), 1(TX)	Connected to NINA	
MKR boards	Serial		13(RX), 14(TX)		
Zero	SerialUSB (Native USB Port only)	Connected to Programming Port	0(RX), 1(TX)		
Due	SerialUSB (Native USB Port only)	0(RX), 1(TX)	19(RX), 18(TX)	17(RX), 16(TX)	15(RX), 14(TX)
101	Serial		0(RX), 1(TX)		



UART Communication Protocol

- On Uno, Nano, Mini, and Mega, pins 0 and 1 are used for communication with the computer.
- Connecting anything to these pins can interfere with that communication, including causing failed uploads to the board.
- You can use the Arduino environment's built-in serial monitor to communicate with an Arduino board.

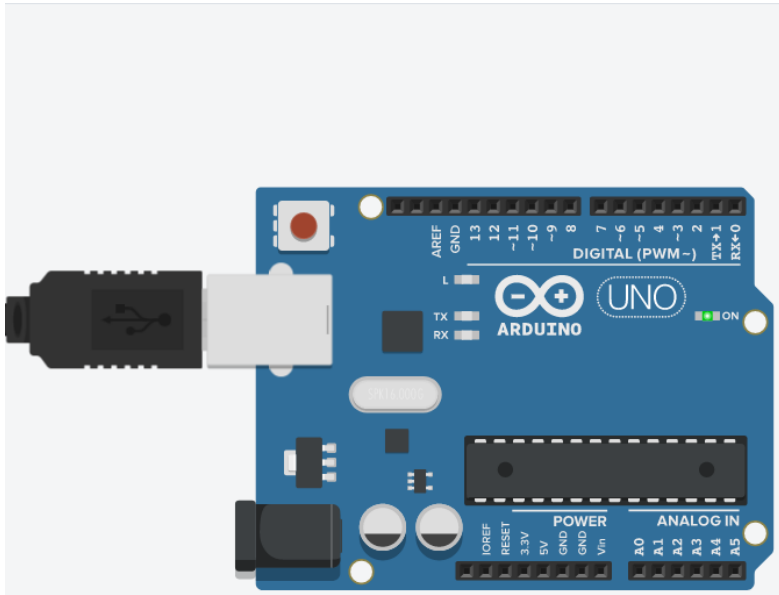
- Click the serial monitor button in the toolbar and select the same baud rate used in the call to begin().
- Serial communication on pins TX/RX uses TTL logic levels (5V or 3.3V depending on the board).





Sending msg to PC via UART

- This demo code will send a text message at 9600 bit/sec



1 (Arduino Uno R3)

```
1  void setup() {
2    Serial.begin(9600);
3    Serial.print("hello");
4  }
5
6  void loop() {
7    // put your main code here, to run repeatedly:
8
9  }
```

Serial Monitor

```
hello
```

Send Clear



UART Communication Protocol

- One important aspect to note about UART communication is that it is designed for communication between only two devices at a time.

- Because the protocol only sends bits indicating the start of a message, the message content, and the end of a message, there is no method of differentiating multiple transmitting and receiving devices on the same line.

- If more than one device attempts to transmit data on the same line, *bus contention* occurs, and the receiving devices will most likely receive garbage unusable data!

https://www.deviceplus.com/arduino/arduino-communication-protocols-tutorial/



UART Communication Protocol

- Furthermore, UART is *half-duplex*, which means that even though communication can occur bidirectionally, both devices *cannot* transmit data to each other at the same time.

- In a project where two Arduinos are communicating with each other via a serial connection, for example, this would mean that at a given time, only one of the Arduinos can be "talking" to the other.



UART Communication Protocol

- A standard Arduino has a single hardware serial port, but serial communication is also possible using **software Serial libraries** to emulate additional ports (communication channels) to provide connectivity to more than one device.

- Software serial requires a lot of help from the Arduino controller to send and receive data, so it's not as fast or efficient as hardware serial.



Week 4

Lab

Lab 3 / LO3

Exporting data to LCD and UART



Tools and components

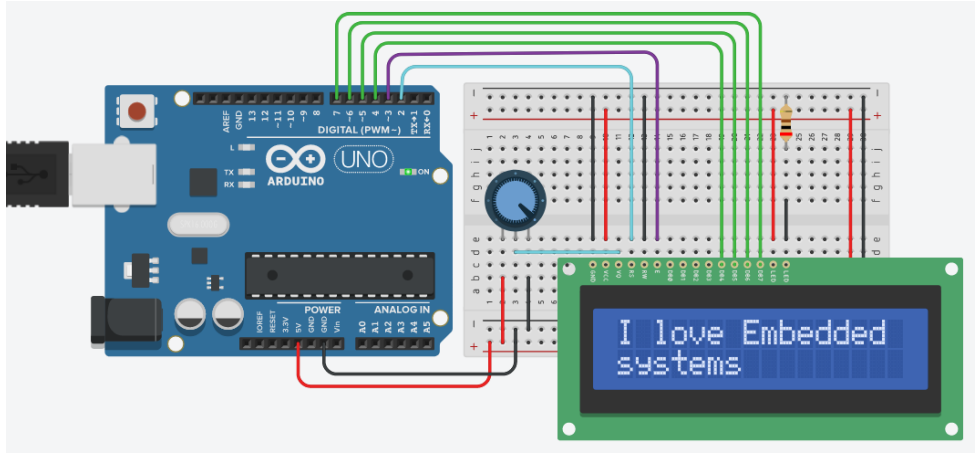
- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. Breadboard wires (male-male)

- Required components:
  1. Arduino UNO + USB cable
  2. Led x1
  3. Resistor from 200 to 470 ohm x 2
  4. LCD 16 character, 2 lines x1
  5. Potentiometer 1k to 10k ohm x1



Code 7: Display msg on LCD

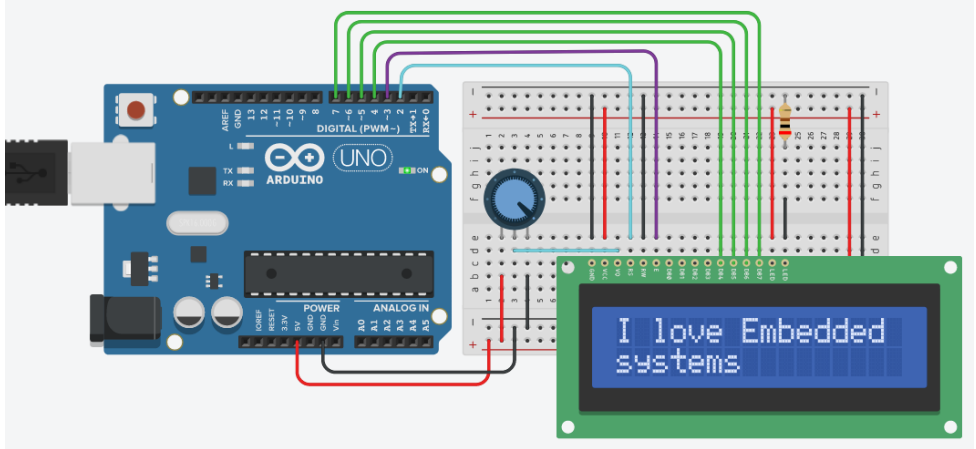
```
1  #include <LiquidCrystal.h>
2  // initialize the library by associating any needed LCD interface pin
3  // with the arduino pin number it is connected to
4  const int rs = 2, en = 3, d4 = 4, d5 = 5, d6 = 6, d7 = 7;
5  LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
6
7  unsigned int counter = 0; // define a global variable
8  void setup() {
9    lcd.begin(16, 2);        // set up the LCD's number of columns and rows:
10 }
11
```





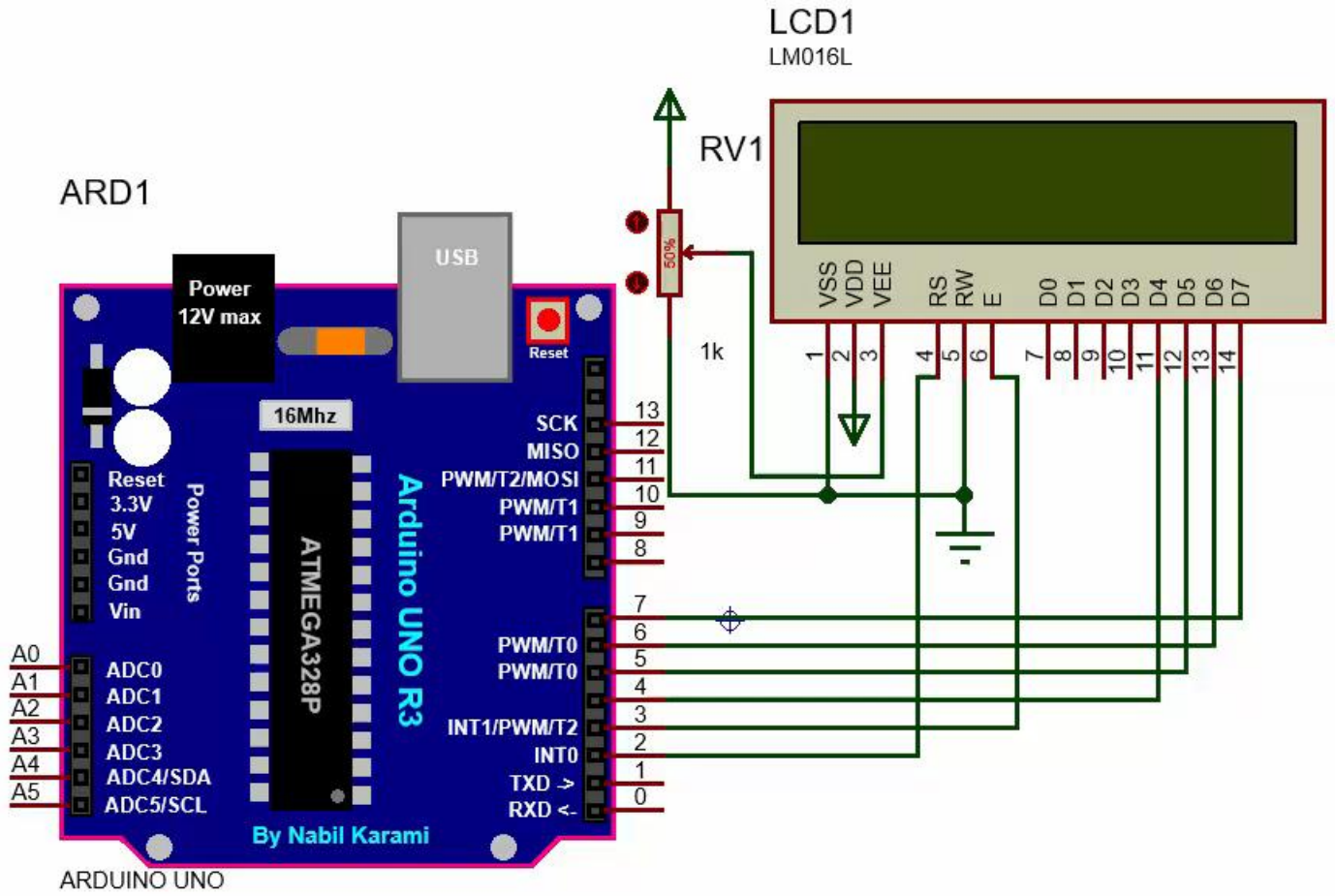
Code 7: Display msg on LCD

```
12 void loop() {
13   lcd.clear();              // Clear the LCD
14   lcd.setCursor(0, 0);      // set the cursor to the first location of the first line
15   lcd.print("I love Embedded");
16   lcd.setCursor(0, 1);      // set the cursor to the first location of the second line
17   lcd.print("systems");
18   delay(2000);
19
20   lcd.clear();              // Clear the LCD
21
22   while (1) {
23     lcd.setCursor(0, 0);
24     lcd.print("Count in D'");
25     lcd.print(counter, DEC);     // display value in decimal
26     lcd.print("'");
27     lcd.setCursor(0, 1);
28     lcd.print("Count in H'");
29     lcd.print(counter, HEX);     // display value in Hexadecimal
30     lcd.print("'");
31     delay(300);
32     counter++;
33   }
34 }
```





Code 7: Display msg on LCD



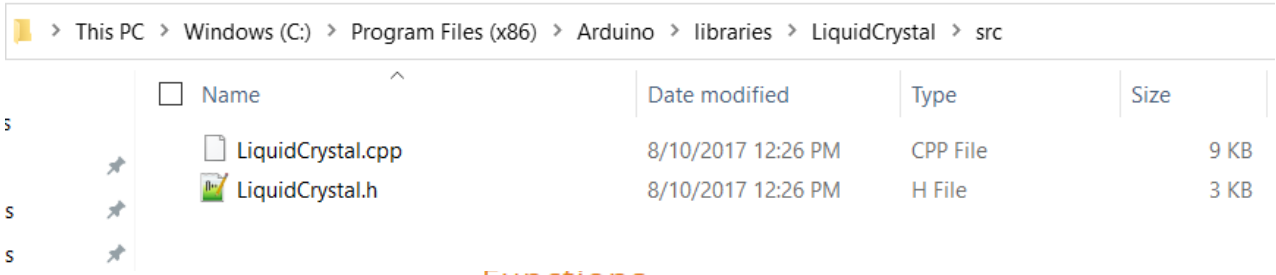
Click to Play video



Code 7: Display msg on LCD

LiquidCrystal library for Arduino:

- This library allows an Arduino board to control LCDs based on the Hitachi HD44780 (or a compatible) chipset, which is found on most text-based LCDs.
- The library works with in either 4- or 8-bit mode (i.e. using 4 or 8 data lines in addition to the RS, E, and, optionally, the RW control lines).



Functions

- LiquidCrystal()
- begin()
- clear()
- home()
- setCursor()
- write()
- print()
- cursor()
- noCursor()
- blink()
- noBlink()
- display()
- noDisplay()
- scrollDisplayLeft()
- scrollDisplayRight()
- autoscroll()
- noAutoscroll()
- leftToRight()
- rightToLeft()
- createChar()

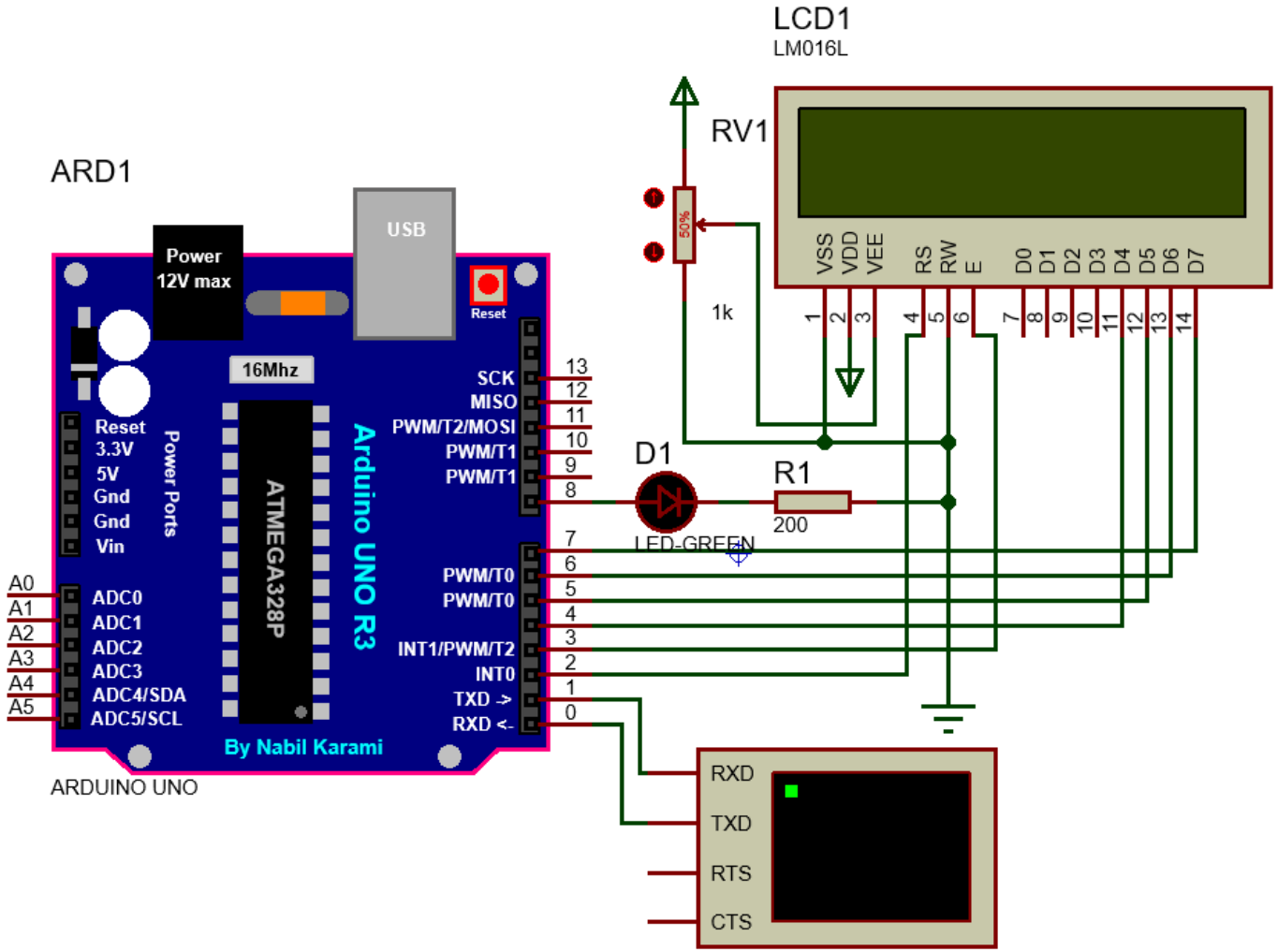


Code 8: Receive data through UART

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(2, 3, 4, 5, 6, 7);
#define led 8

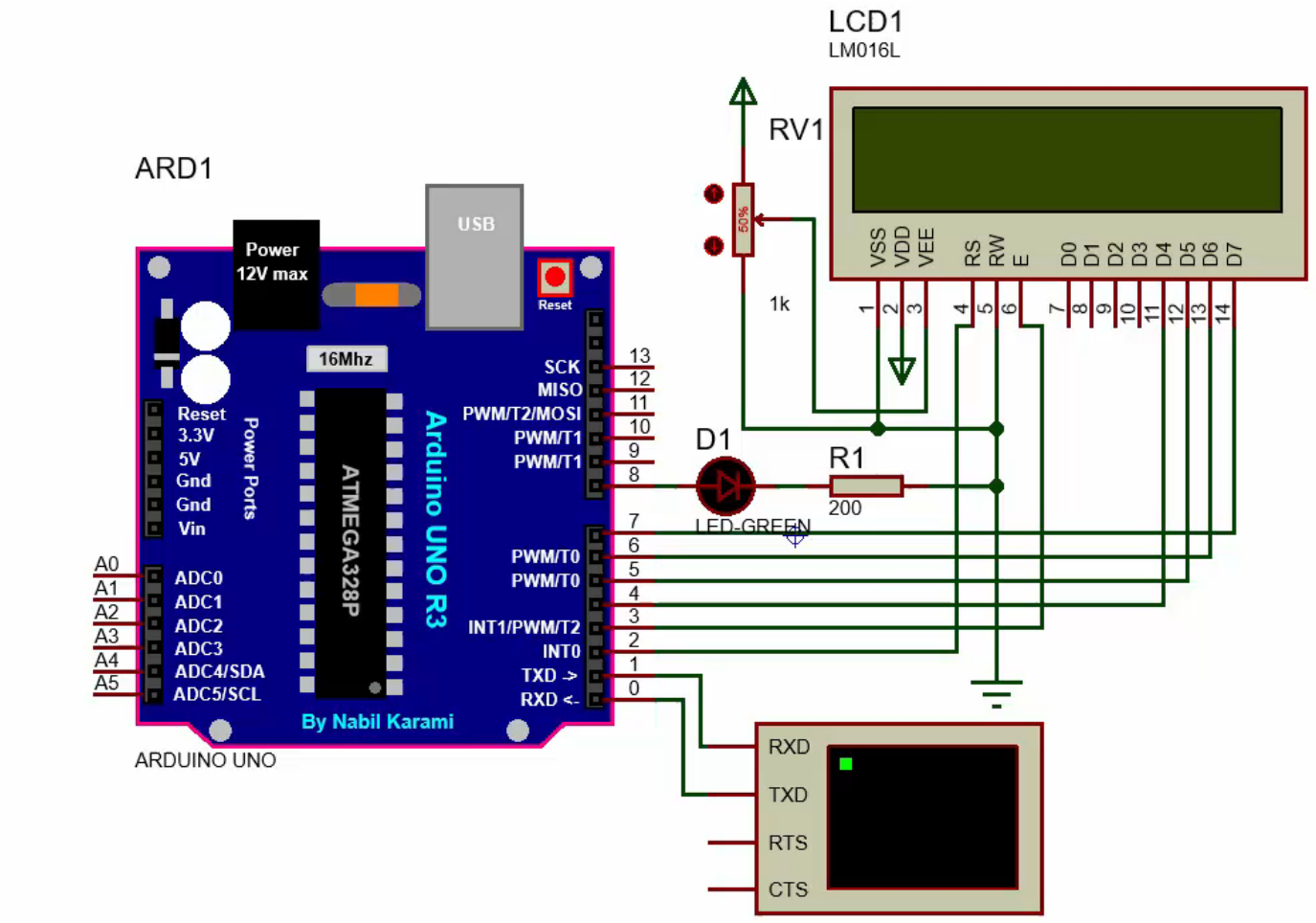
void setup() {
  lcd.begin(16, 2);
  Serial.begin(9600);
  Serial.println("Send H or L to control the LED");
  pinMode(led, OUTPUT);
}

void loop() {
  if (Serial.available()) {
    char inChar = (char)Serial.read();
    if (inChar == 'H') {
      digitalWrite(led, HIGH);
      lcd.clear();
      lcd.setCursor(0, 0);
      lcd.println("Led is ON");
    }
    if (inChar == 'L') {
      digitalWrite(led, LOW);
      lcd.clear();
      lcd.setCursor(0, 0);
      lcd.println("Led is OFF");
    }
  }
}
```





Code 8: Receive data through UART



Click to Play video



UART Communication Protocol

Serial and SoftwareSerial Method	Purpose	Code	Explanation
Constructor (SoftwareSerial only)	Define the GPIO pins that will serve as the UART RX and TX lines	SoftwareSerial comms (2 , 3);	Defines a serial connection with RX line on GPIO 2 and TX line on GPIO 3.
begin	Define the baud rate (transmission speed) for the serial connection in range 4800 to 115200	comms.begin(9600);	Communication on the "comms" serial port will occur at 9600 baud.
print	Write byte data converted into human-readable characters over serial connection	comms.println("Hello World");	Writes bytes equivalent to *Hello World* (human-readable characters)
write	Write raw byte data over the serial connection	comms.write(45);	Writes byte with value 45.
available	Evaluates to true when data is available over the serial connection	if (comms.available())	Enter *if* statement if there is data available to read over the serial connection
read	Read data from the serial connection	comms.read();	Reads from serial connection

Functions

if(Serial)
available()
availableForWrite()
begin()
end()
find()
findUntil()
flush()
parseFloat()
parseInt()
peek()
print()
println()
read()
readBytes()
readBytesUntil()
readString()
readStringUntil()
setTimeout()
write()
serialEvent()

Lab



Week 5

Lecture 5 / LO3

Keypad and 7-segment display



Keypad

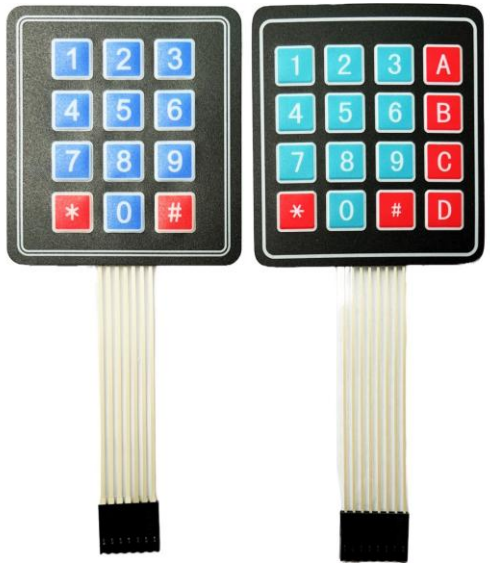
- A keypad is a set of buttons arranged in a block that usually bear digits and other symbols but not a complete set of alphabetical letters.
- Keypads are found on many devices such as calculators, combination locks and telephones, which require largely numeric input.
- Keypads are usually available in two dimensions; 4*4 and 3*4. These are the number of button in row and columns.

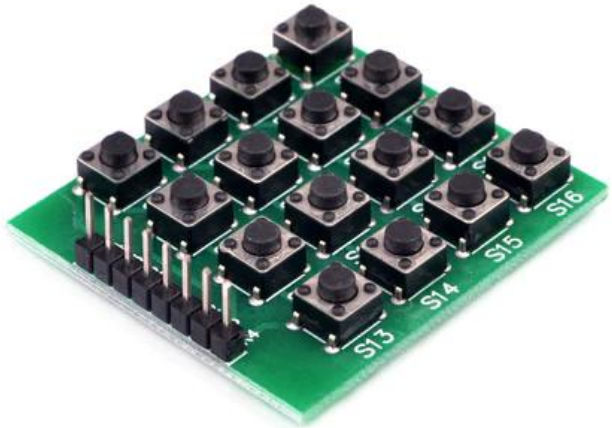
4*4 Matrix



4*3 Matrix



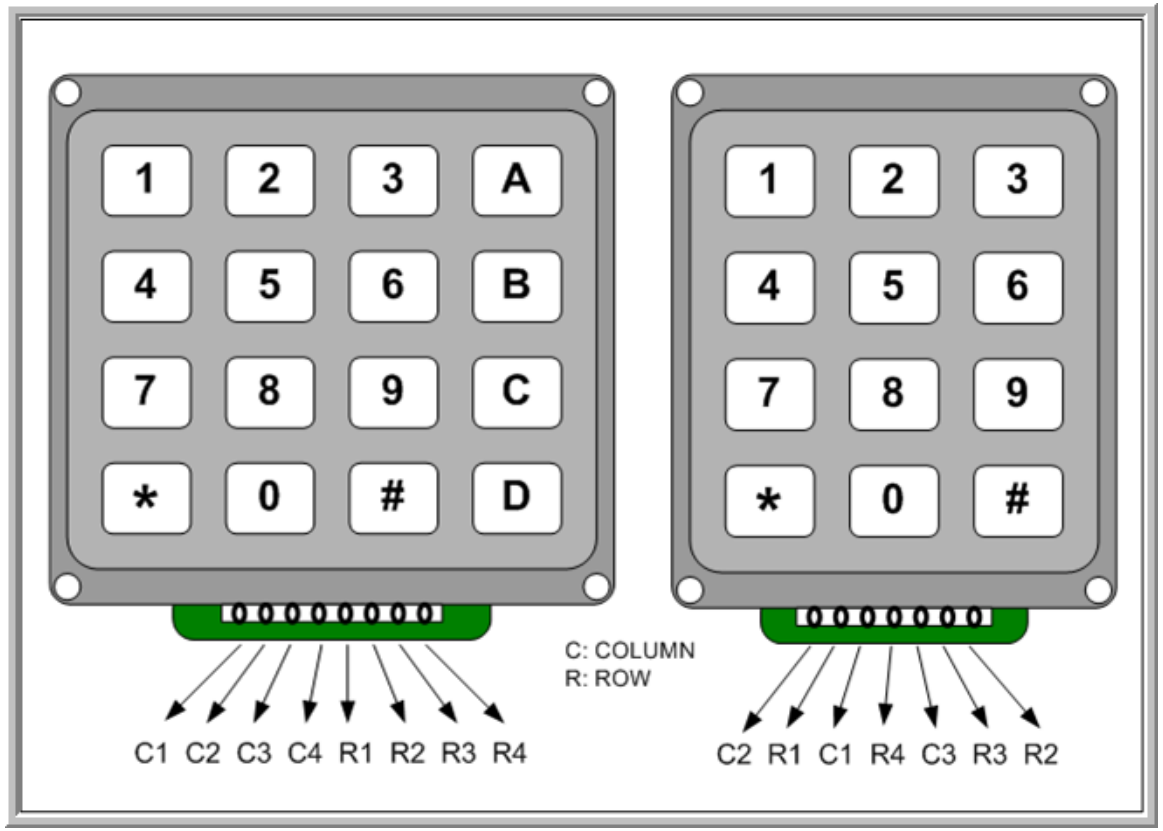


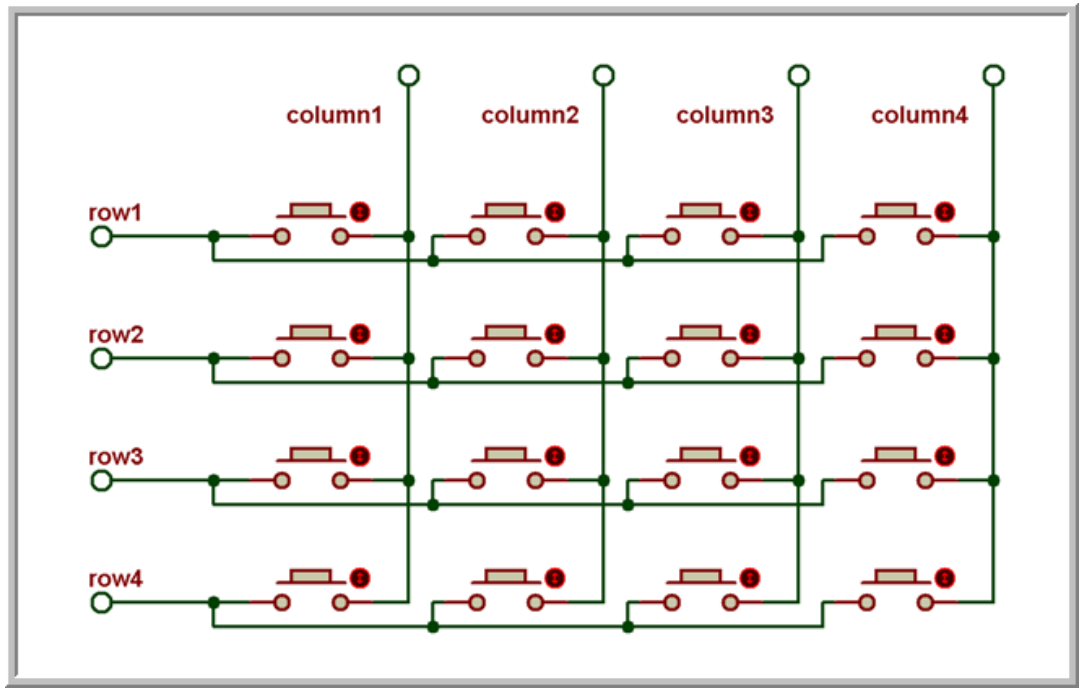




Keypad

- A 4x4 keypad has 8 lines (4 for columns and 4 for rows)
- A3x4 keypad has 7 lines (3 for columns and 4 for rows)

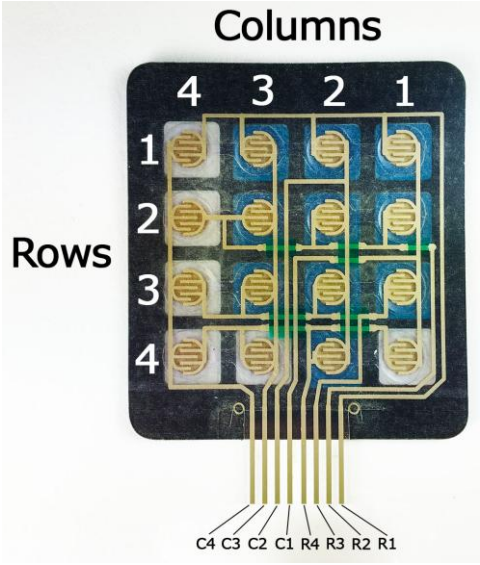


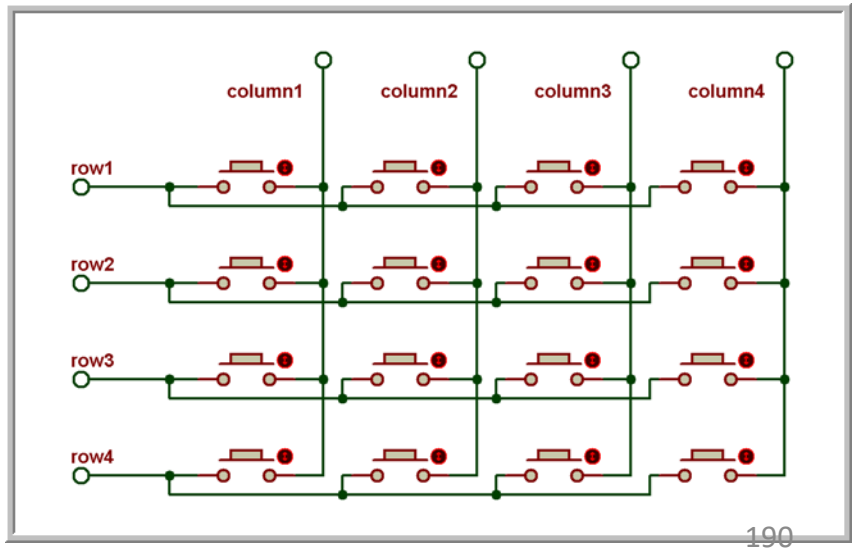




Keypad structure

- The main feature of a keypad is its internal buttons connection, where rows and columns are connected in a matrix way.
- So, rather than having 32 terminals for the 16 buttons, the keypad has only 8 terminals.
- Therefore, the decoding of a keypad using logic circuit is cumbersome.
- It needs many gates to scan the row and columns.
- However, a microcontroller can handle the keypad scanning easily using the “trial and error” algorithm.

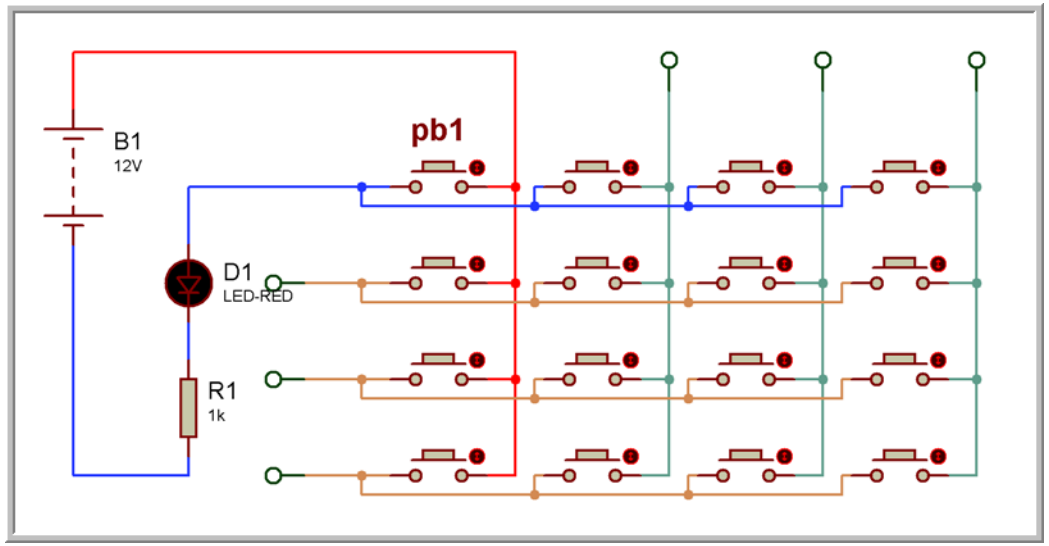






Keypad structure

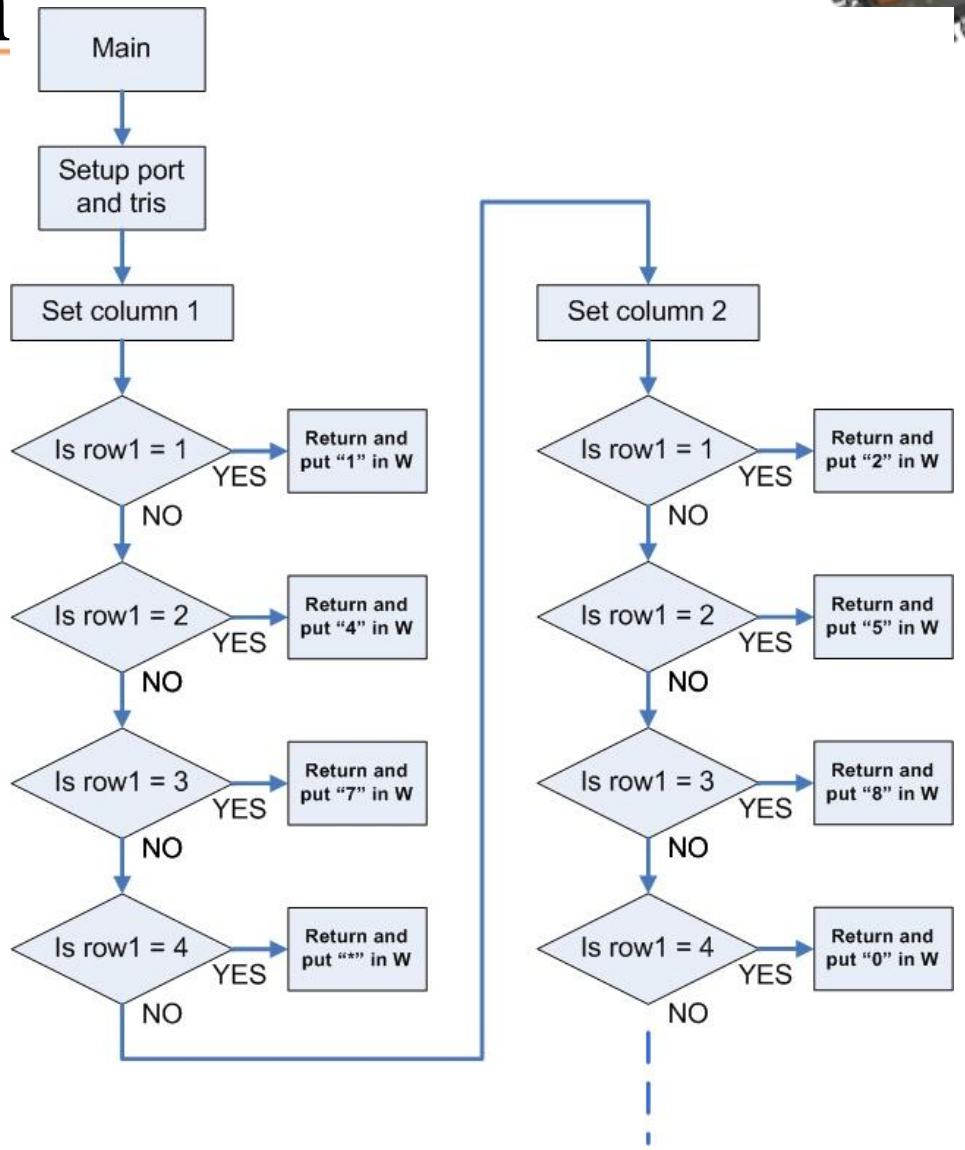
- The scanning method consists of sending a signal to a column and reading the entire rows.
- If no signal is received, means that no row key is pressed.
- If for example, the signal is received from the second row that means the second key is pressed.
- This scenario is repeated to scan all columns.
- In the Figure, the battery anode is connected to the first column, and the LED anode is connected to the first row. When the key “pb1” is pressed, the LED will turn on.





Keypad scanning Algorithm

- The Arduino role is to send logic “1” on every column consecutively and read all rows. Therefore, four pins are configured as input and four other are configured as output.
- First, the Arduino sends a signal on the first column and starts the test of rows consecutively. If no signal is received from rows, it switches to the next columns and send a signal again.
- This pinging and testing loop is repeated until a key is pressed. The scan can also be inverted so you can send signal to rows and test columns.





Keypad demo

We'll use the Keypad library by Mark Stanley and Alexander Brevig. This library takes care of setting up the pins and polling the different columns and rows.

```
#include <Keypad.h>

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

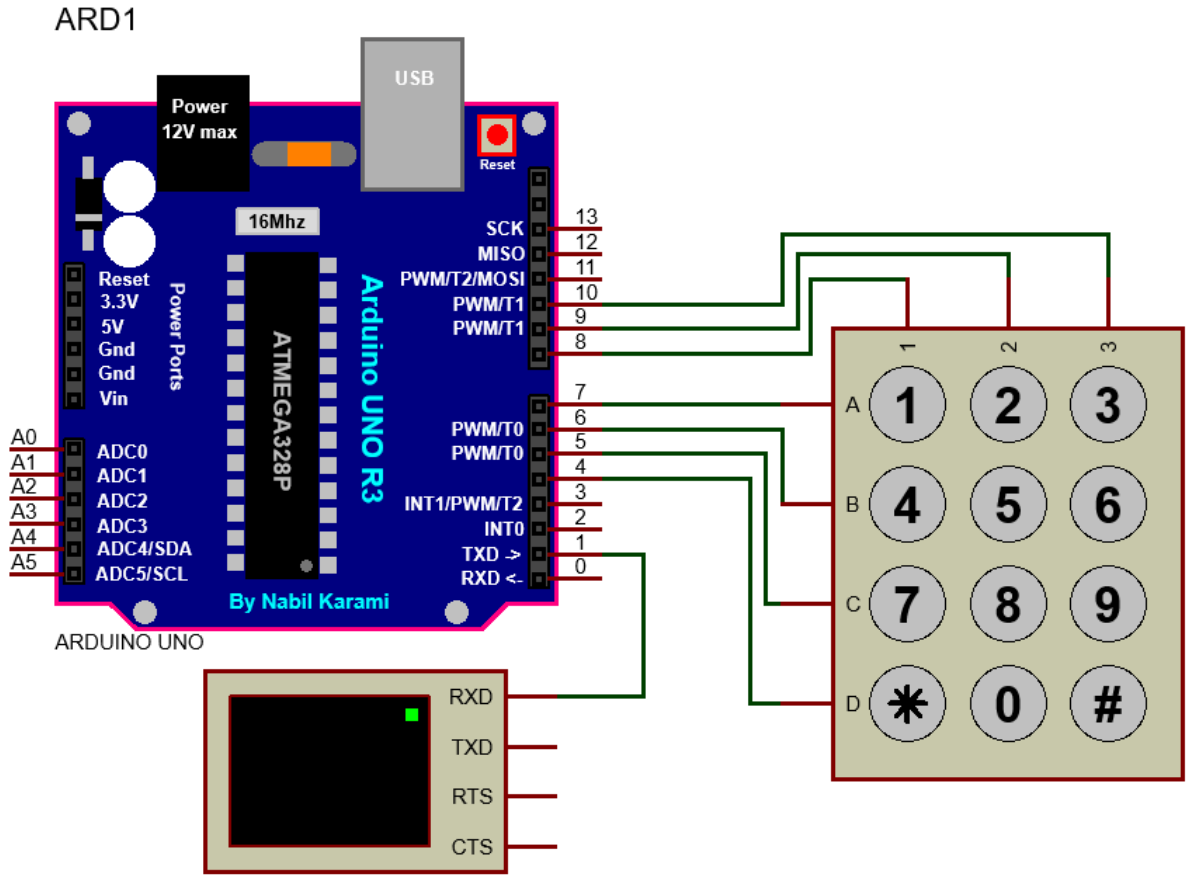
char hexaKeys[ROWS][COLS] = {
  {'1', '2', '3'},
  {'4', '5', '6'},
  {'7', '8', '9'},
  {'*', '0', '#'}
};

byte rowPins[ROWS] = {7, 6, 5, 4};
byte colPins[COLS] = {8, 9, 10};

Keypad customKeypad = Keypad(makeKeymap(hexaKeys), rowPins, colPins, ROWS, COLS);

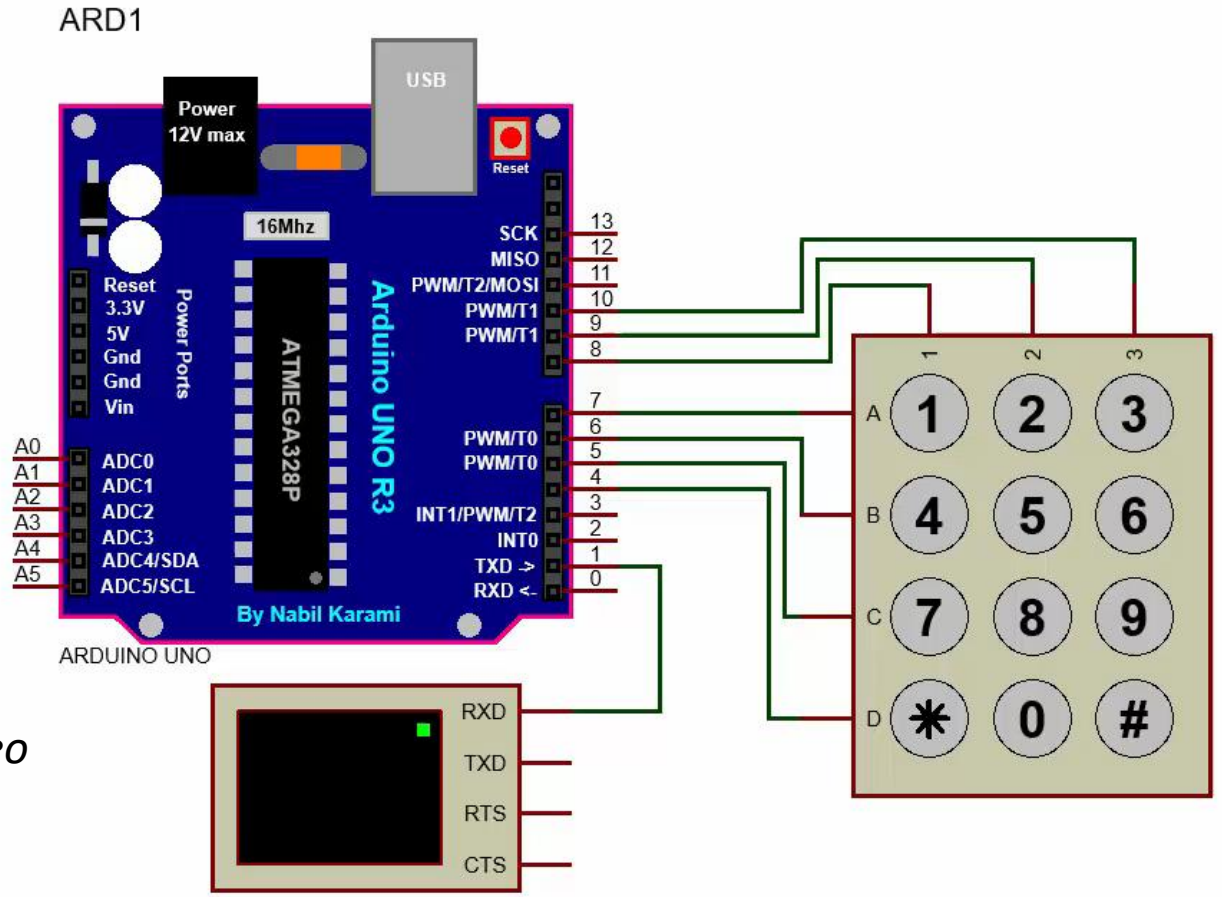
void setup(){
  Serial.begin(9600);
}

void loop(){
  char customKey = customKeypad.getKey();
  if (customKey){
    Serial.println(customKey);
  }
}
```





Keypad demo



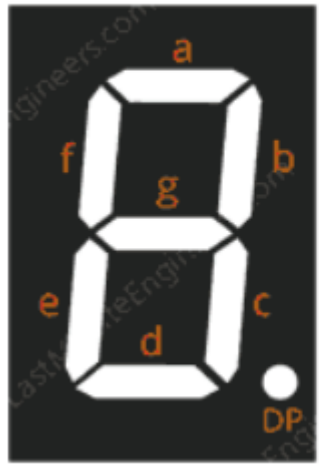
Click to Play video

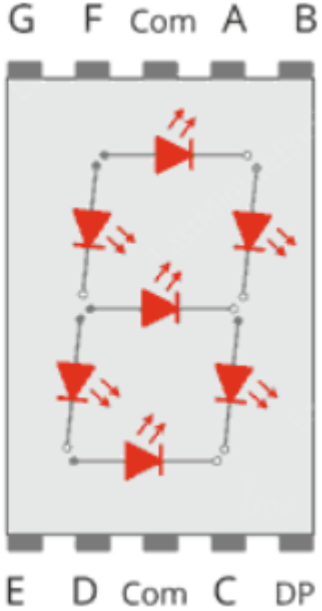


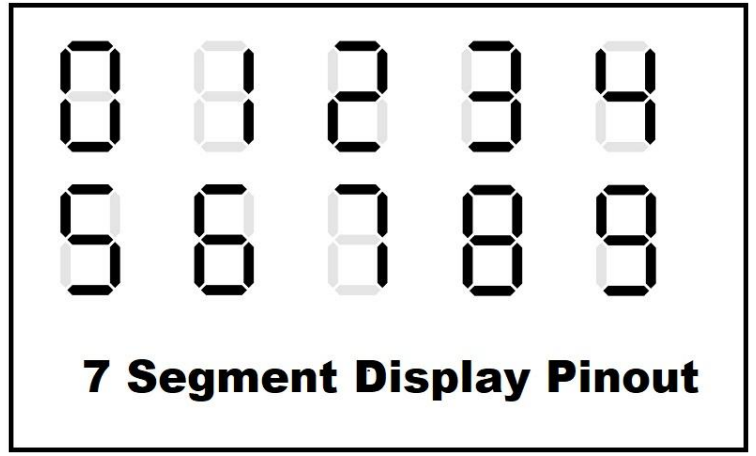
7-Segment Display

- One digit seven segment displays are used in many embedded system and industrial applications where the range of outputs to be shown is known beforehand.
- Basic 1 digit seven segment display can show numbers from 0-9 and a few characters.
- 7 segment displays are of different types; especially they differ in the number of digits/character it can display.
- Basically a 7 segment display is a single unit, which can display only 1 digit or 1 character.
- More digits are displayed by multiplexing single unit 7 segment displays together to form 2 digit display, 3 digit display or 4 digit 7 segment display.

- A 7 segment display has many limitations, especially in the range of characters it can display.
- There are displays in market which are much more advanced than seven segment displays and can display almost every character in the alphabet.



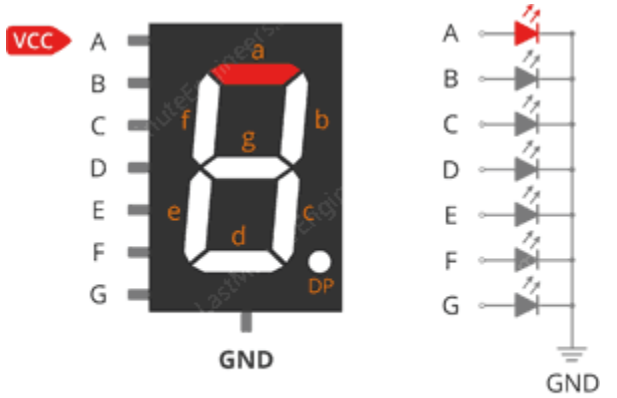




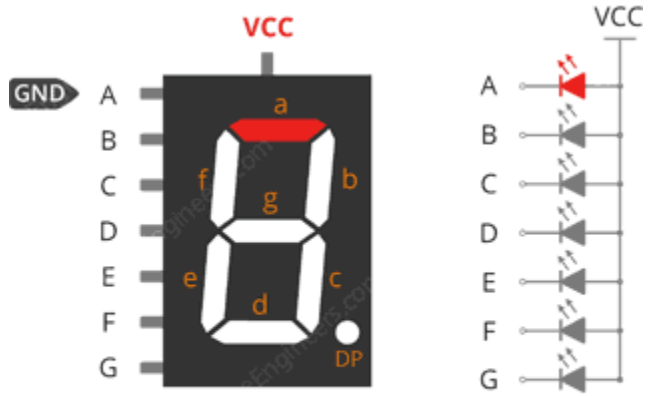


7-Segment Display

- Each of the LEDs in the display is given a positional segment with one of its connection pins led out from the rectangular plastic package.
- These LED pins are labeled from “a” through to “g” representing each individual LED.
- The other LED pins are connected together forming a common pin.
- So by forward biasing the appropriate pins of the LED segments in a particular order, some segments will brighten and others stay dim, thus showing the corresponding character on the display.



- The common pin of the display generally tells its type.
- There are two types of pin connection: a pin of connected cathodes and one of connected anodes, indicating Common Cathode (CC) and Common Anode (CA).
- As the name suggests, a CC display has all the cathodes of the 7 LEDs connected when a CA display has all the anodes of the 7 segments connected.





7-Segment Display

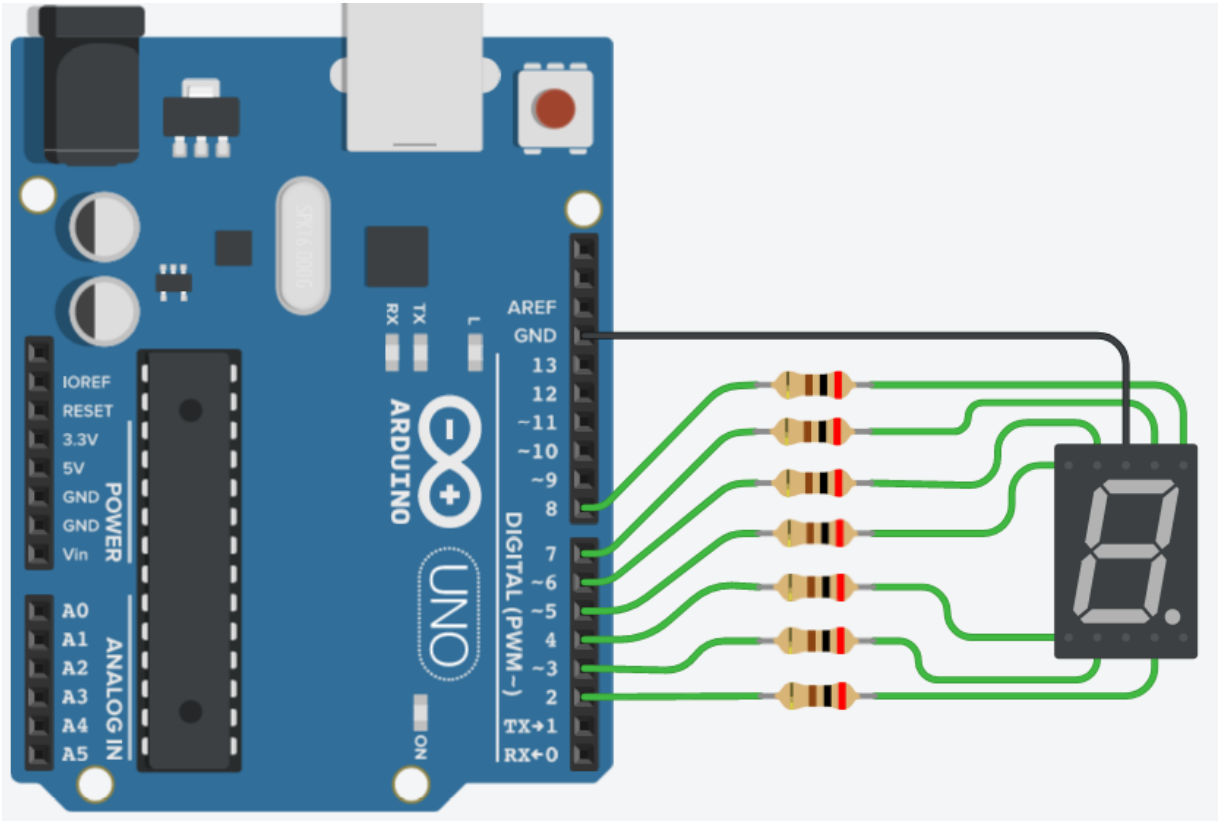
```
#define a 7
#define b 8
#define c 2
#define d 3
#define e 4
#define f 6
#define g 5

void setup() {
  pinMode(a, OUTPUT);
  pinMode(b, OUTPUT);
  pinMode(c, OUTPUT);
  pinMode(d, OUTPUT);
  pinMode(e, OUTPUT);
  pinMode(f, OUTPUT);
  pinMode(g, OUTPUT);
}
```

```
void seg_0() {
  digitalWrite(a, HIGH);
  digitalWrite(b, HIGH);
  digitalWrite(c, HIGH);
  digitalWrite(d, HIGH);
  digitalWrite(e, HIGH);
  digitalWrite(f, HIGH);
  digitalWrite(g, LOW);
}

void seg_1() {
  digitalWrite(a, LOW);
  digitalWrite(b, HIGH);
  digitalWrite(c, HIGH);
  digitalWrite(d, LOW);
  digitalWrite(e, LOW);
  digitalWrite(f, LOW);
  digitalWrite(g, LOW);
}

void seg_2() {
  digitalWrite(a, HIGH);
  digitalWrite(b, HIGH);
  digitalWrite(c, LOW);
  digitalWrite(d, HIGH);
  digitalWrite(e, HIGH);
  digitalWrite(f, LOW);
  digitalWrite(g, HIGH);
}
```





7-Segment Display

```
50 void seg_3() {
51   digitalWrite(a, HIGH);
52   digitalWrite(b, HIGH);
53   digitalWrite(c, HIGH);
54   digitalWrite(d, HIGH);
55   digitalWrite(e, LOW);
56   digitalWrite(f, LOW);
57   digitalWrite(g, HIGH);
58 }
59
60 void seg_4() {
61   digitalWrite(a, LOW);
62   digitalWrite(b, HIGH);
63   digitalWrite(c, HIGH);
64   digitalWrite(d, LOW);
65   digitalWrite(e, LOW);
66   digitalWrite(f, HIGH);
67   digitalWrite(g, HIGH);
68 }
69
70 void seg_5() {
71   digitalWrite(a, HIGH);
72   digitalWrite(b, LOW);
73   digitalWrite(c, HIGH);
74   digitalWrite(d, HIGH);
75   digitalWrite(e, LOW);
76   digitalWrite(f, HIGH);
77   digitalWrite(g, HIGH);
78 }
```

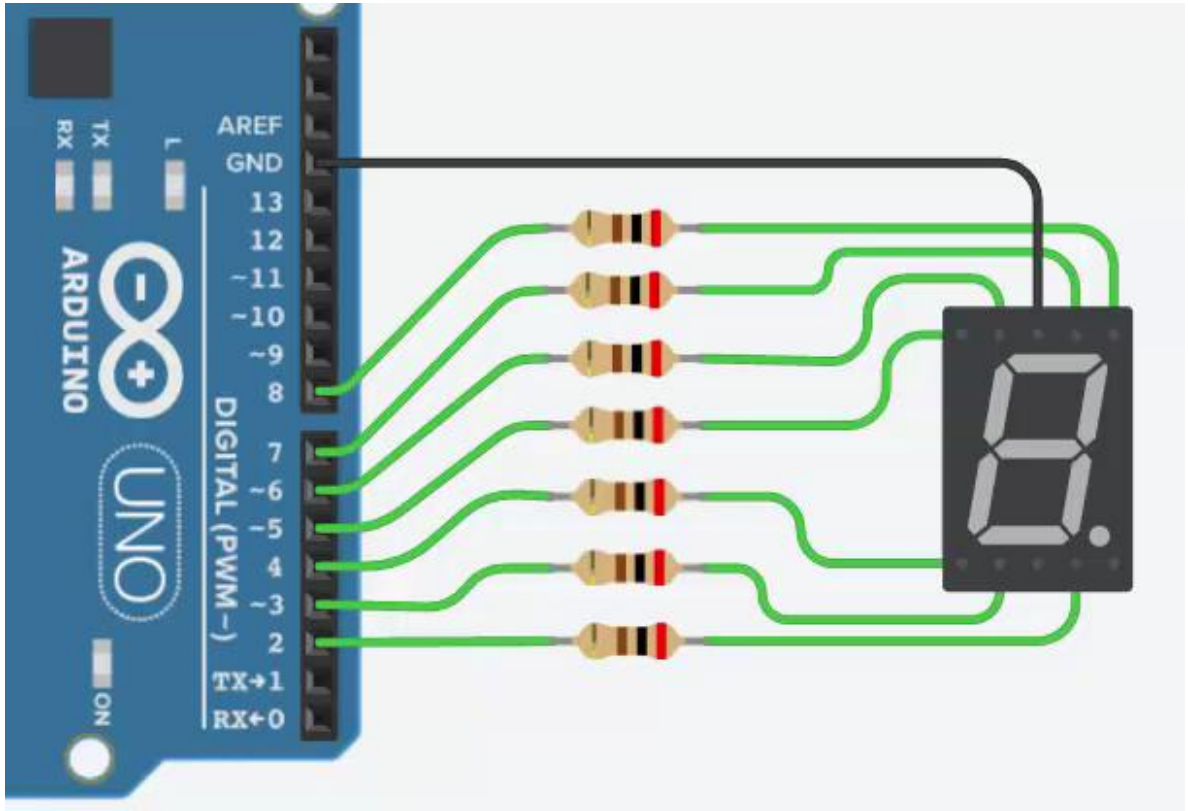
```
80 void seg_6() {
81   digitalWrite(a, HIGH);
82   digitalWrite(b, LOW);
83   digitalWrite(c, HIGH);
84   digitalWrite(d, HIGH);
85   digitalWrite(e, HIGH);
86   digitalWrite(f, HIGH);
87   digitalWrite(g, HIGH);
88 }
89
90 void seg_7() {
91   digitalWrite(a, HIGH);
92   digitalWrite(b, HIGH);
93   digitalWrite(c, HIGH);
94   digitalWrite(d, LOW);
95   digitalWrite(e, LOW);
96   digitalWrite(f, LOW);
97   digitalWrite(g, LOW);
98 }
99
100 void seg_8() {
101   digitalWrite(a, HIGH);
102   digitalWrite(b, HIGH);
103   digitalWrite(c, HIGH);
104   digitalWrite(d, HIGH);
105   digitalWrite(e, HIGH);
106   digitalWrite(f, HIGH);
107   digitalWrite(g, HIGH);
108 }
```

```
110 void seg_9() {
111   digitalWrite(a, HIGH);
112   digitalWrite(b, HIGH);
113   digitalWrite(c, HIGH);
114   digitalWrite(d, HIGH);
115   digitalWrite(e, LOW);
116   digitalWrite(f, HIGH);
117   digitalWrite(g, HIGH);
118 }
```

```
120 void loop() {
121   seg_0();
122   delay(500);
123   seg_1();
124   delay(500);
125   seg_2();
126   delay(500);
127   seg_3();
128   delay(500);
129   seg_4();
130   delay(500);
131   seg_5();
132   delay(500);
133   seg_6();
134   delay(500);
135   seg_7();
136   delay(500);
137   seg_8();
138   delay(500);
139   seg_9();
140   delay(500);
141 }
```



7-Segment Display



Week 5



Lab 4 / LO3

Working with keypad/7-seg display



Tools and components

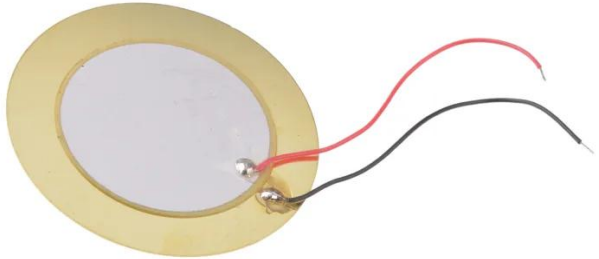
- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. Breadboard wires (male-male)

- Required components:
  1. Arduino UNO + USB cable
  2. Resistor from 200 to 470 ohm x 7
  3. Piezo buzzer x1
  4. Common cathode 7 seg display x1
  5. 3x4 keypad x1



Keypad with 7seg and buzzer

- In this experiment, a keypad and a 7-seg display are interfaced with the Arduino.
- The purpose if the code is to display the pressed number on the display. Numbers to be displayed are 0..9. the extra keypad buttons (* and #) are used to generate sound on the piezo buzzer

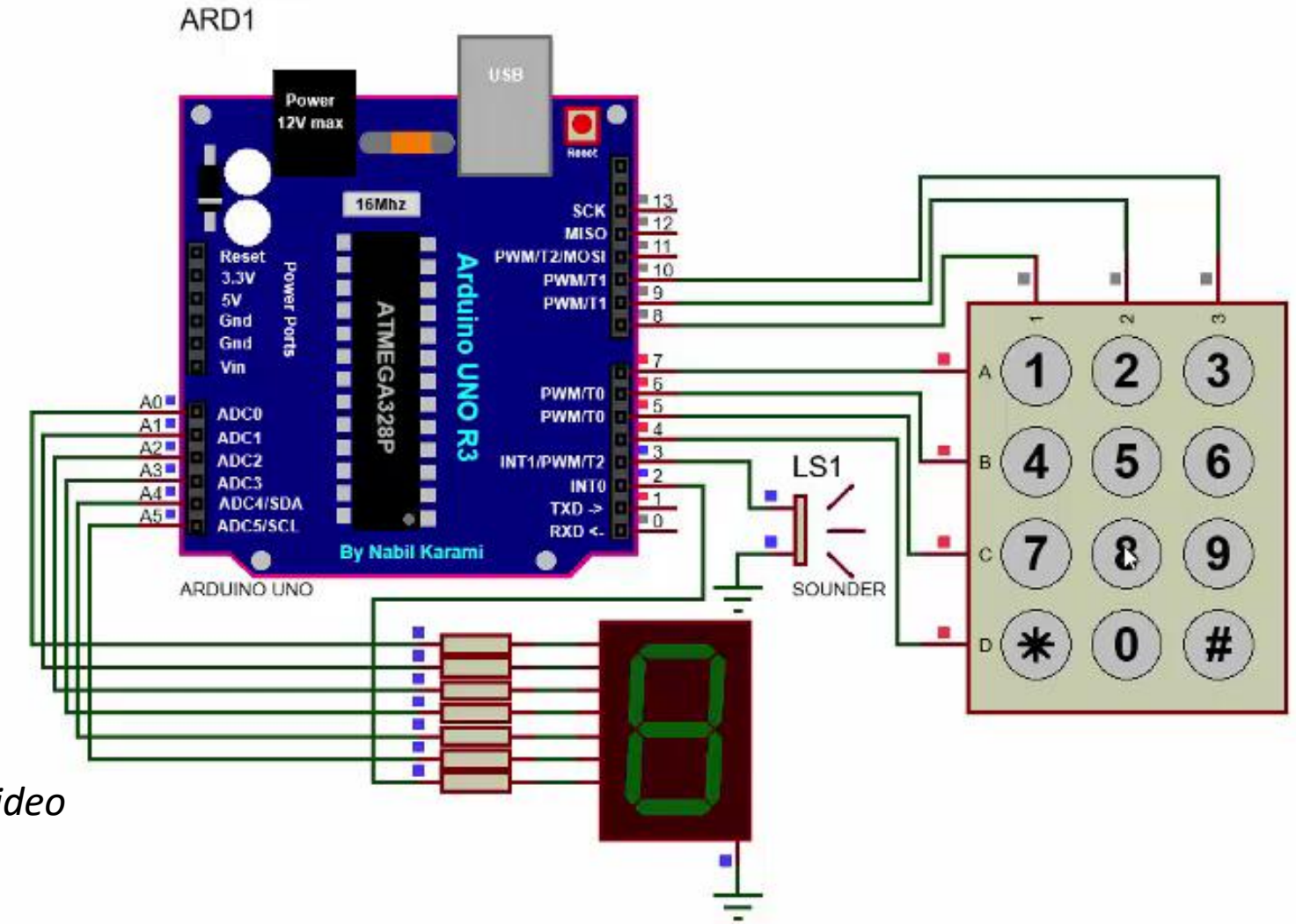


- A piezoelectric buzzer is a loudspeaker that uses the piezoelectric effect for generating sound.
- The initial mechanical motion is created by applying a voltage to a piezoelectric material, and this motion is typically converted into audible sound using diaphragms and resonators.
- Compared to other speaker designs piezoelectric speakers are relatively easy to drive; for example they can be connected directly to TTL outputs, although more complex drivers can give greater sound intensity.
- Typically they operate well in the range of 1 - 5kHz and up to 100kHz in ultrasound applications





Keypad with 7seg and buzzer



Click to Play video



Keypad with 7seg and buzzer

```
#include <Keypad.h>

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

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

#define a A0
#define b A1
#define c A2
#define d A3
#define e A4
#define f A5
#define g 2
#define piezo 3
```

```
void loop() {
  char customKey = customKeypad.getKey();
  switch (customKey) {
    case '0':
      seg_0();
      break;
    case '1':
      seg_1();
      break;
    case '2':
      seg_2();
      break;
    case '3':
      seg_3();
      break;
    case '4':
      seg_4();
      break;
    case '5':
      seg_5();
      break;
    case '6':
      seg_6();
      break;
```



Keypad with 7seg and buzzer

```
64      case '7':
65        seg_7();
66        break;
67      case '8':
68        seg_8();
69        break;
70      case '9':
71        seg_9();
72        break;
73      case '*':
74        tone(piezo, 500, 500);
75        break;
76      case '#':
77        tone(piezo, 1000, 500);
78        break;
79    }
80
81  }
```

- The tone function generates a square wave on the piezo pin at 500 Hz for 500ms

- The tone function generates a square wave on the piezo pin at 1000 Hz for 500ms



Week 6

Lecture

Lecture 6 / LO3

RTOS



Intro to RTOS and multitasking

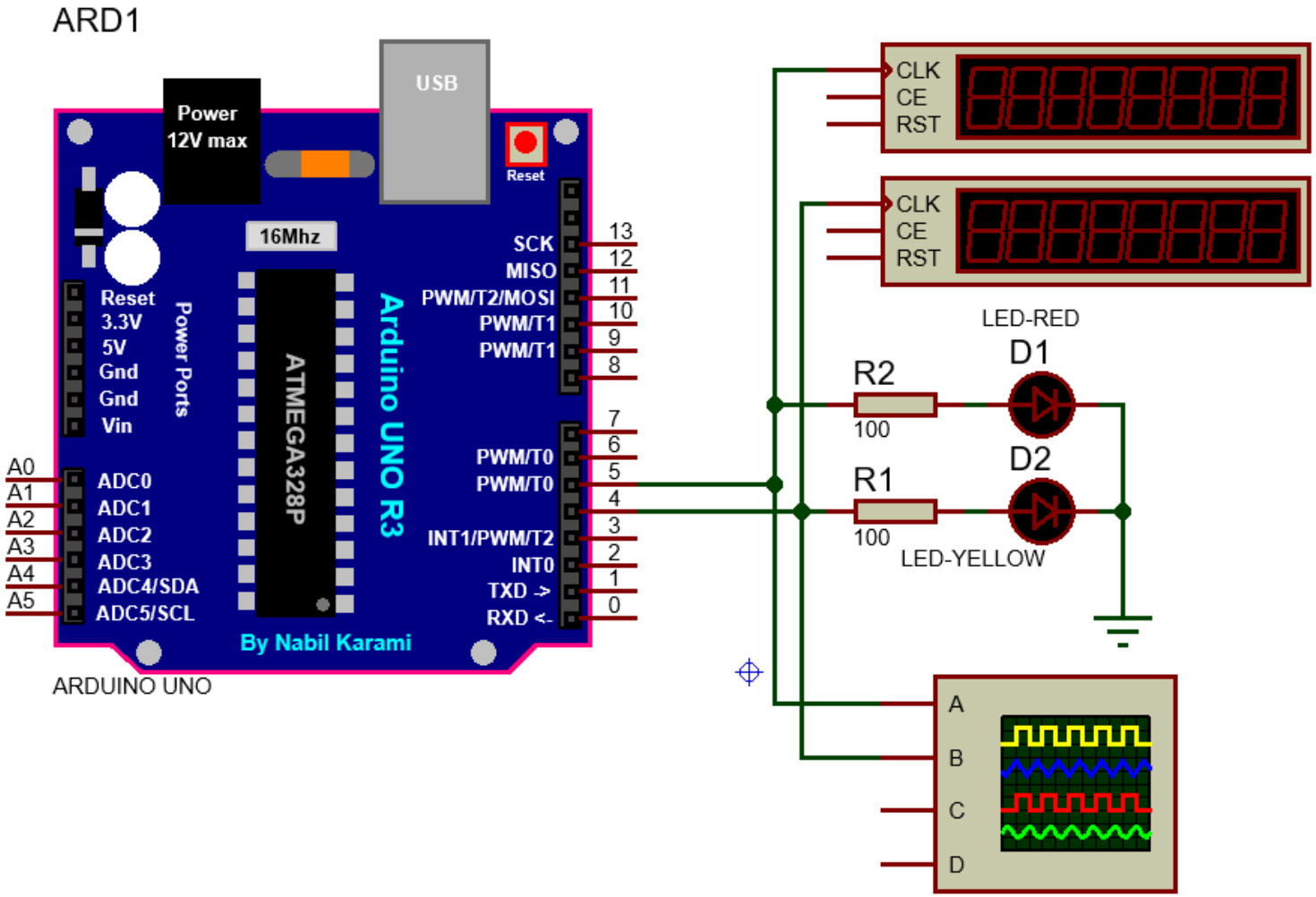
It is required to blink 2 leds at different frequencies at the same time.

In fact, this task is not straight forward and required lot of attention in coding.

Assume that we need to blink the leds at 2Hz and 5Hz (i.e. the period of the first led is 0.5sec and the second led is 0.2sec)



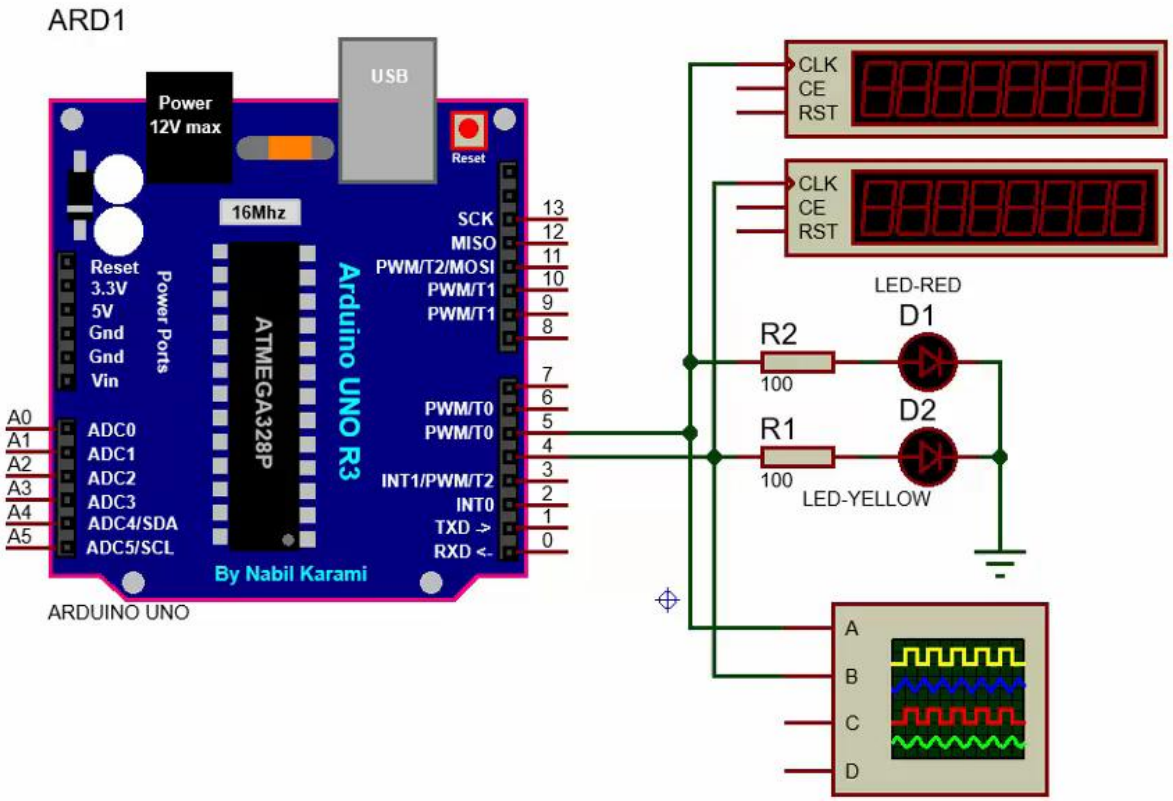
Intro to RTOS and multitasking





Intro to RTOS and multitasking

Click to Play video





Intro to RTOS and multitasking

```
#define led1 4
#define led2 5
unsigned long previousMillis_led1 = 0;
unsigned long previousMillis_led2 = 0;
const long interval_led1 = 250; // for 2 hz
const long interval_led2 = 100; // for 5 hz

void setup() {
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
}

void loop() {
  unsigned long currentMillis_led1 = millis();  // read the actual time and save
  unsigned long currentMillis_led2 = millis(); // read the actual time and save

  if (currentMillis_led1 - previousMillis_led1 >= interval_led1) {  // check if 250ms ellapsed
    previousMillis_led1 = currentMillis_led1;                       // if, yes reschedule for the next cycle
    digitalWrite(led1, !digitalRead(led1));                         // toggle led1
  }

  if (currentMillis_led2 - previousMillis_led2 >= interval_led2) {  // check if 100ms ellapsed
    previousMillis_led2 = currentMillis_led2;                       // if, yes reschedule for the next cycle
    digitalWrite(led2, !digitalRead(led2));                         // toggle led 2
  }
}
```

Millis() Returns the number of milliseconds passed since the Arduino board began running the current program.
It is like a stopwatch

Lecture



Intro to RTOS and multitasking

scheduler



Is time = 250 ms ?

If yes, toggle led 1
And reset time for the next cycle





Is time = 100 ms ?

If yes, toggle led 2
And reset time for the next cycle





Intro to RTOS and multitasking

Observation

The two tasks are happening at the same time due to a schedule defined by a timer (using the millis() function)

For a more advanced tasking, the millis() use become difficult and thus we need a real time operating system to make the scheduling for the task.

So… we need RTOS



Real Time Operating System (RTOS)

- Real time system
  - What is RTOS
  - What are the types of RTOS
  - How to achieve real time processing

- Operating System
  - What is OS
  - Why to use OS
  - How to use OS



Real Time Systems

- In a Real-Time System the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced
- Late answer is a wrong answer
- Specification of a Real time system include both
  - Logical correctness requirement (Produce correct answer)
  - Temporal correctness requirement (Produce outputs at the right time)



Real Time Systems

- Real-Time systems can be classified into a hard real-time and soft real-time

- The response time requirements of hard real-time systems are in the order of milliseconds or less and can result in a catastrophe if not met. For example:
  - Air bag
  - Anti-lock braking system (ABS)
  - Aircraft control systems

- In contrast, the response time requirements of soft real-time systems are higher and not very stringent. For examples
  - Video processing
  - Audio processing
  - Digital Cameras
  - Mobile phones



Free RTOS

- **FreeRTOS stands for Free Real-Time Operating System.**
- **It is an open-source operating system targeted on embedded applications that run on a microcontroller and need real-time event processing.**

- FreeRTOS is both free for commercial use and the source code is available for everyone to see as it is open-source, same as Linux, which is another famous open-source operating system.
- Since it is free to use, open-source and professionally maintained by an organization, it is considered to be the best operating system for embedded and IoT applications.



Free RTOS

- Before we can learn about RTOS let's take a moment to cover one more idea, that is multitasking.
- What is multitasking? **Multitasking means running many tasks/threads/processes on a processor simultaneously at the same time.**
- Operating Systems, in general, are used to support multitasking.
- For example, consider a device



- that can sense the temperature in your room.
- that can update the readings over wifi to your smartphone.
- simultaneously it controls an LED indicator that blinks once every 2 seconds if the temperature is above 20 degrees and blinks every second if the temperature is below 20 degrees.



Free RTOS

- It has a single processor that needs to do all these three tasks at the same time, in other words, it needs to do some multitasking. But a processor can only handle one task at a time.
- **To give us the illusion that all three processes are running at the same time, operating systems use a scheduler to let the 3 processes to share the processor time.**
- So sensor reading is taken in the first 5 milliseconds, in the next 10 milliseconds it is sent to your smartphone and then in the next 5 milliseconds the LED status is updated.
- This is done over and over again to make the end-user think that all 3 processes are running parallelly even though **they are running one after the other in a pseudo-parallel manner.**



Free RTOS

FreeRTOS place among other embedded software.

Now that we understand what FreeRTOS is, lets next try and understand its place among other design-patterns and operating systems by taking a look at some examples.



Complexity level #0: Regular Watch

Consider a Simple Digital Watch. It's only functionality is that it tells time. Such a product can be developed using a simple 8-bit microcontroller and an operating system is of very little use here as it only runs a single thread continuously. We will have a look at the concept of threads in a coming section, for now, you just need to know that **a thread is a step by step process.**



Free RTOS

Complexity level#1: Sports watch

- Now let's add a little bit more complexity, Let's add a stopwatch to the timer, and an alarm clock and a count-down timer. Now the system has to run multiple tasks at the same time. These tasks include
  - it has to keep track of the time,
  - if a stopwatch is running it has to run count the number of seconds that have expired since the start button has been clicked and
  - if it's time to ring the alarm to wake you up it needs to start beeping.



- For such an application there are **effective software design patterns** that can be used so that interrupts and loops can be used to implement this without an operating system on a microcontroller.
- **The software complexity without an operating system is still considered manageable** and hence they are usually built without an OS.



Free RTOS

Complexity level #2: Fitness monitor

- Now let's add more functionality to our digital watch, lets put in a pedometer, an altimeter and a temperature sensor. Now the tasks that it needs to compute parallelly includes
  - Keeping track of time
  - Stopwatch
  - Alarm clock
  - Pedometer signal processing
  - Temperature sensing
  - Altitude sensing
  - Display information on an LCD/LED screen



- So, on one side it has to do the job of a normal digital watch, on the other hand it needs to do some smart features by taking in raw accelerometer and gyroscope values from an IMU sensor and do some signal processing to calculate your steps, continuously read the temperature values from the temperature sensor, read altitude measurements and display it all on your screen.



Free RTOS

- Even this application can be designed without an operating system, by just architecting some good design patterns, but usually, this kind of system can be benefited by adding a simple embedded operating system **as the software complexity without an operating system is usually considered to be not very easy to develop and maintain**.
- Also, such a system are usually built using 32bit microcontrollers with enough resources to accommodate an operating system and this makes engineers of such products to go for an operating system based software.
- This is where simple real-time operating systems like FreeRTOS comes in.





Free RTOS

Complexity level #3: Smart watch

- Lets now add-in a few more features, let's add in a heart rate monitor, a sleep monitor, a Bluetooth communication system to communicate with your phone, an mp3 player to stream music to your Bluetooth earphones and a notification system to make your smartwatch vibrate every time you receive a call on your paired smartphone. Essentially all the features of smartwatches these days.
- Now the system is not simple enough to be made using simple embedded operating systems like FreeRTOS.
- It is still possible to do it but then the added complexity is not easy for the development and maintenance of the system.
- Now we will probably need a much more sophisticated Operating system like **Embedded Linux** to accomplish our tasks.





Free RTOS

Operating System Concepts

There are 7 concepts and associated terminology you must master to start developing using FreeRTOS

- Tasks
- Multitasking
- Context switching
- Schedulers
- Kernels
- Interrupts and
- Intertask communications



Free RTOS

Tasks/Processes

- In the examples above of using a digital watch, **each separate activity/feature is a Process.**
- **Process is a single line of actions, that are executed one after the other**
- In FreeRTOS these processes are called **Tasks.**
- Each such task has its own stack in combination with another block of memory to store information about itself known as a **Task Control Block** which is used by FreeRTOS kernel to manage the task.
- In FreeRTOS, **these tasks are implemented as C functions with a never-ending loop.**

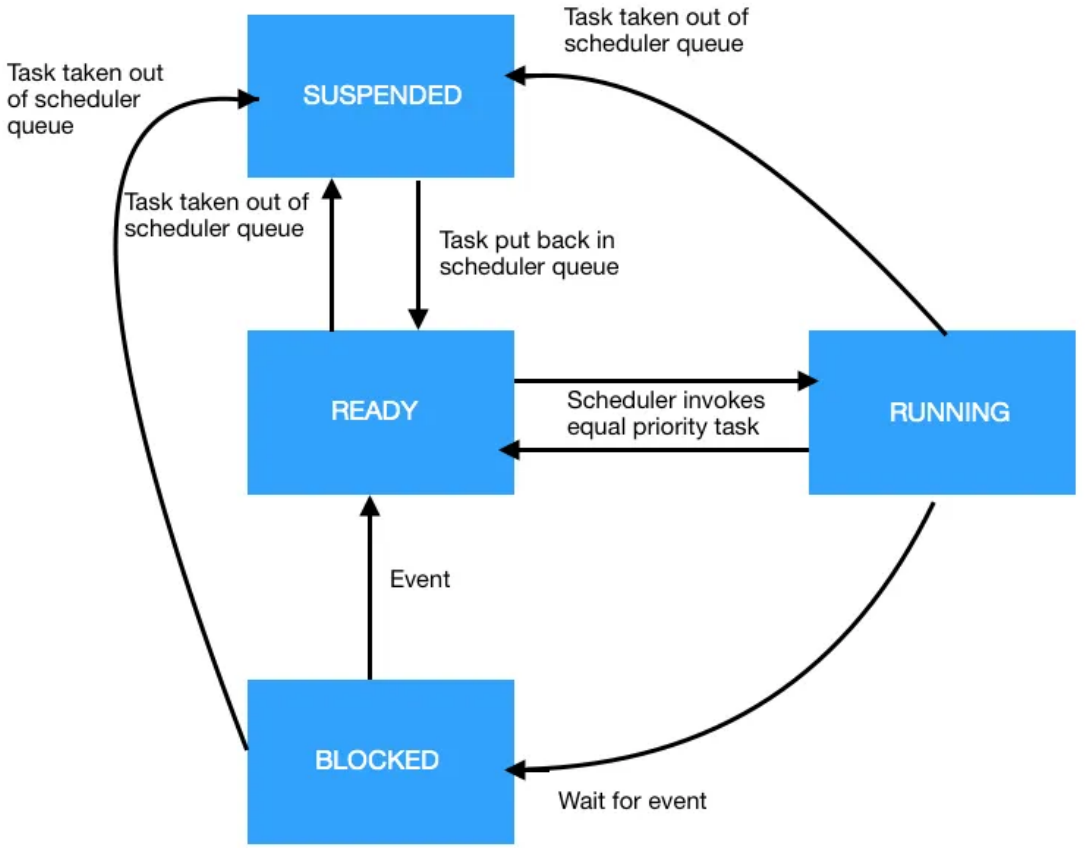


Free RTOS

Multitasking and state changes

- As we saw earlier, multitasking is the way of sharing the processor's time, so that a number of tasks can execute pseudo-parallelly by taking turns to use the CPU. At any given point in time, only one process can be present in the running state. Other processes are usually in one of the following 3 states

1. **Ready state**
2. **Blocked state**
3. **Suspended state**

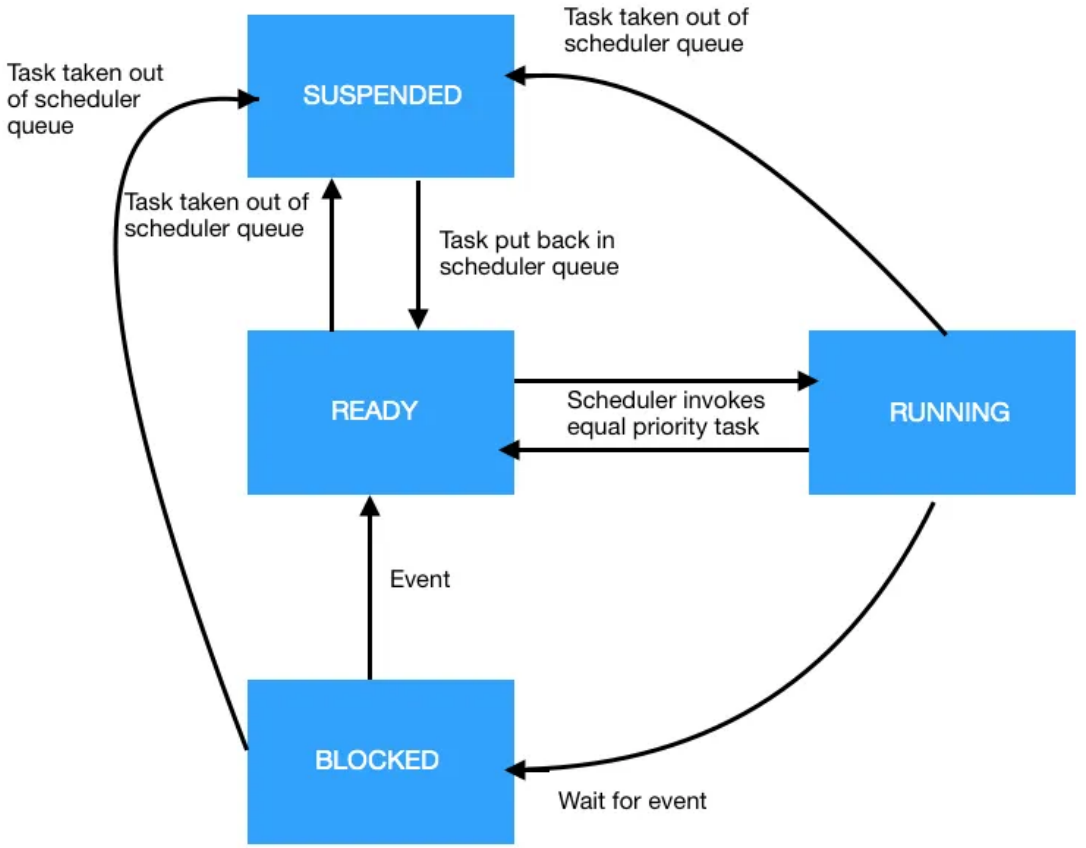




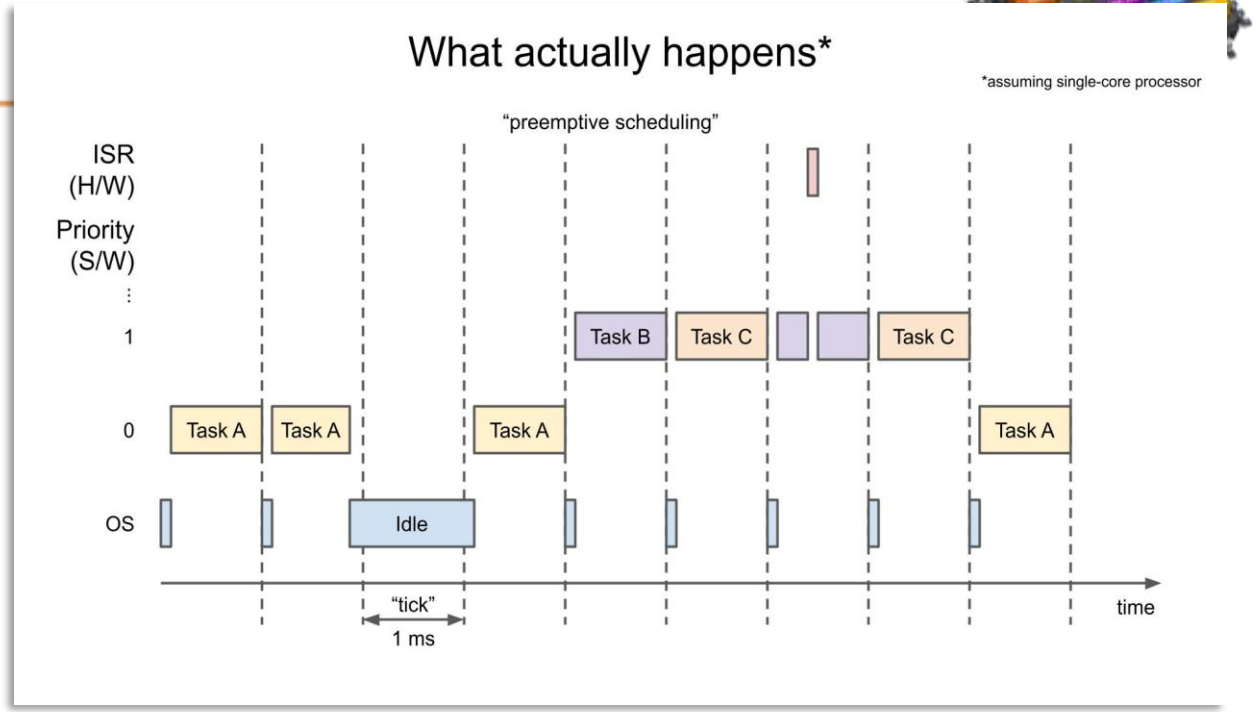
Free RTOS

Tasks changing states

- In the ready state, the task is ready to run but the CPU is currently being occupied by some other task. In our smart-watch example, you can think of the stop-watch as a task that is in the running stage, while the regular clock function is in the ready state.
- In the blocked state, the task is waiting for some data or an event to occur. A good example is the alarm task on our smartwatch example. It will be in kept blocked state till the time arrives for the alarm to ring.
- Tasks in the suspended state are not active anymore. Let's take the same alarm task for example, it will in a suspended state once you turn it off.
- Tasks are made to change from running state to one of the inactive states by the **scheduler** by the process of **preempting**.



Free RTOS



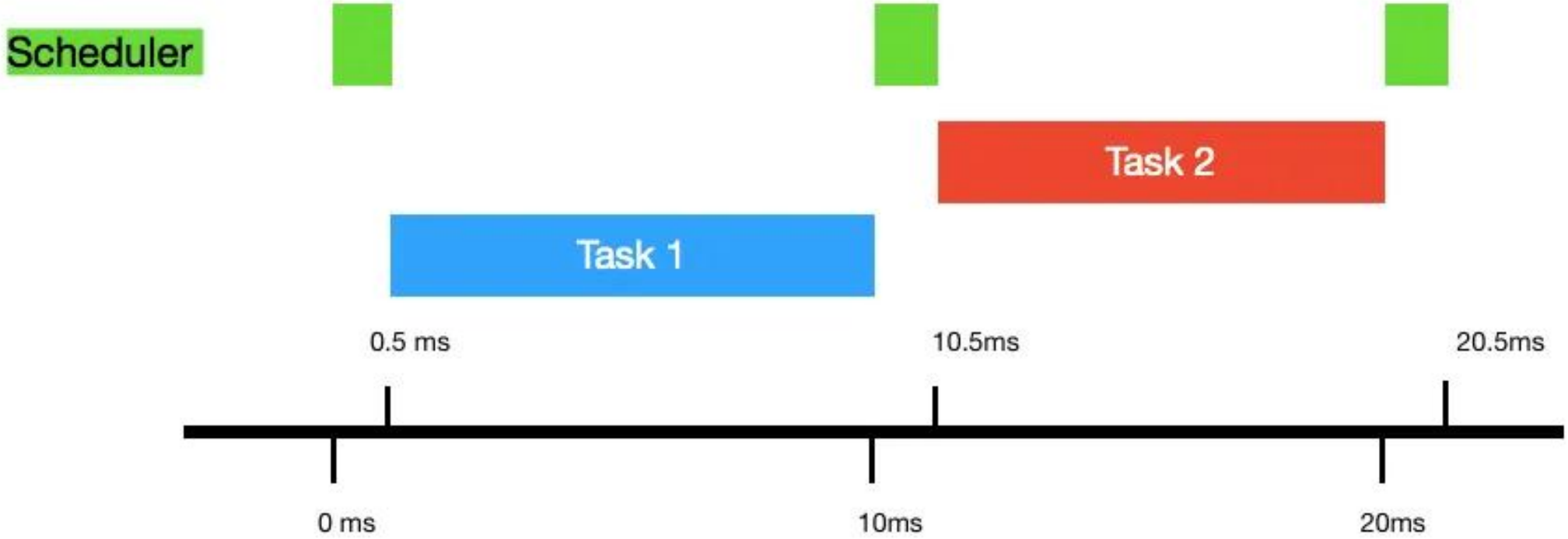
- What is **preempting**? A currently running task is stopped without any notice, in favor of a higher priority task, and its contents and status information are stored in memory so that when it runs again, it can start from where it left off.

- What is **Scheduler**? It is a task from the FreeRTOS operating system, whose task is to manage the state of the other tasks.
  - It is the most important part of any real-time operating system. Its duty is to make sure no lower priority tasks can be in running state while a higher priority task is in the ready state so that the timing requirements can be met.
  - But as we just saw, the scheduler is also a task, which means it can only take a limited amount of processor's time.



Free RTOS

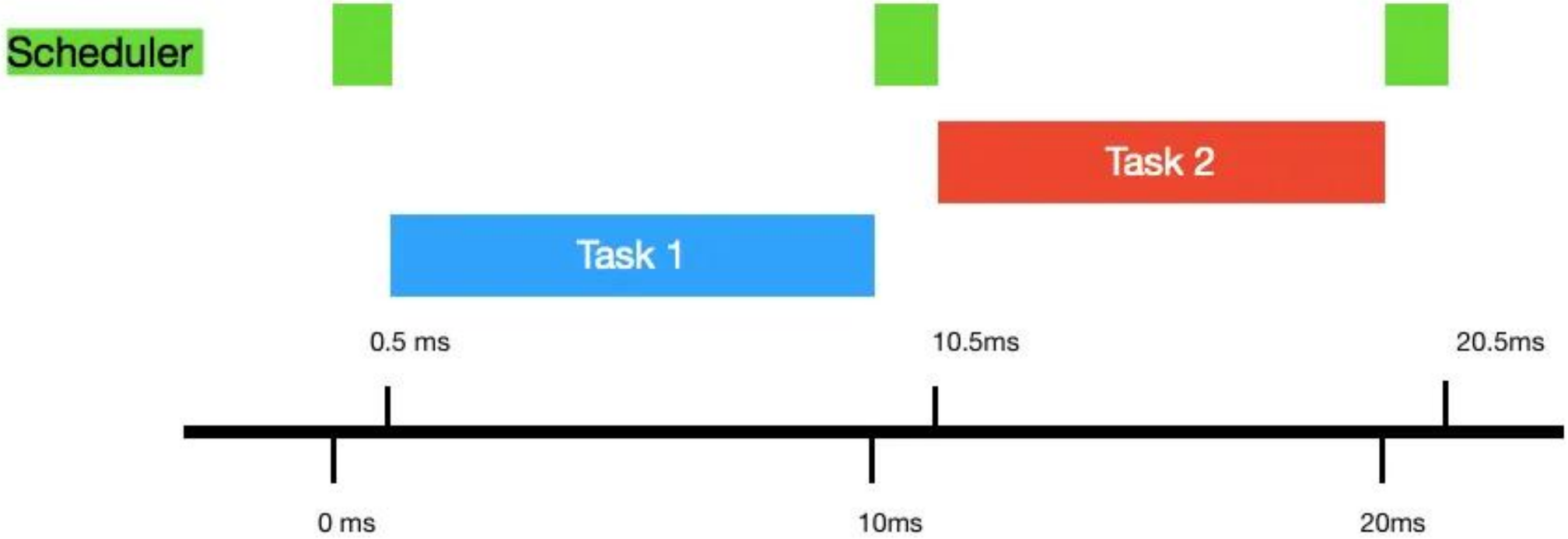
The scheduler usually runs once every fixed duration of time, say for example once every 10 milliseconds. The scheduler occupies the CPU, say for 0.5ms and during this time, it will see the manage the tasks so that only the highest priority task in the ready state gets the next slot in time.





Free RTOS

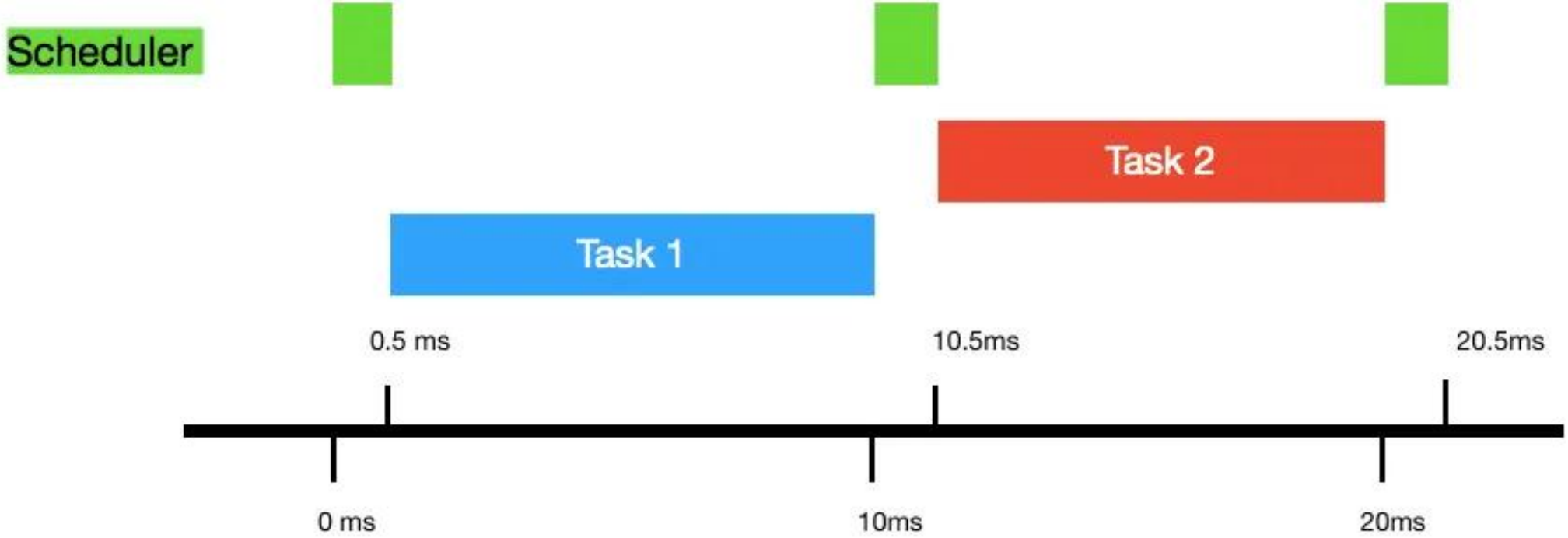
Here the green task is the scheduler. It starts at 0ms and runs till 0.5ms and chooses Task1 to run till 10ms. Then at 10ms the scheduler comes back again and this time picks another task to run from the ready queue which is Task 2 and now Task 2 runs till 20ms.





Free RTOS

If 2 tasks of the same priority are in the ready state, then the scheduler will choose the one that hasn't had a chance to run for the longest period of time and give it the next 9.5ms period to use the processor. Once that's done the scheduler comes in again and gives the CPU to the other same priority task. Thus both these tasks can take turns running of using the processor.

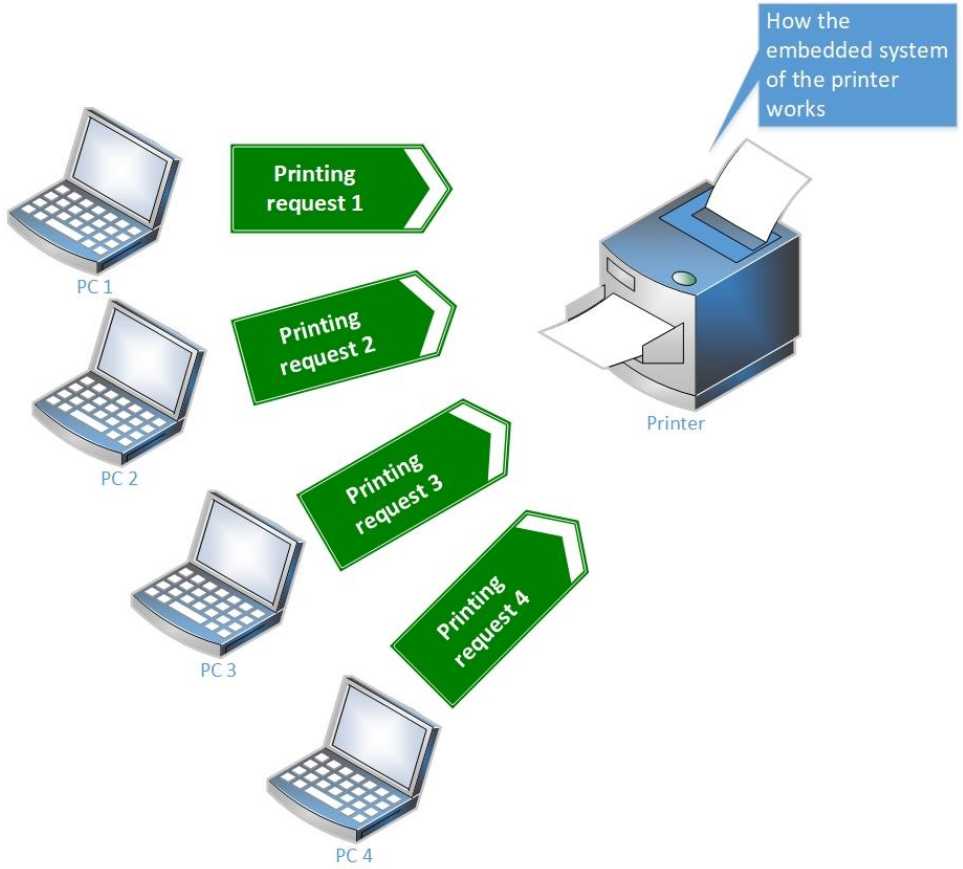




Free RTOS

Round Robin

- The main loop simply checks each of the inputs in turn and services any that need request
- If the task are not critical, their deadline is long and the task service (printing) requires a short time, this system can be used as a real time system with a simple architecture



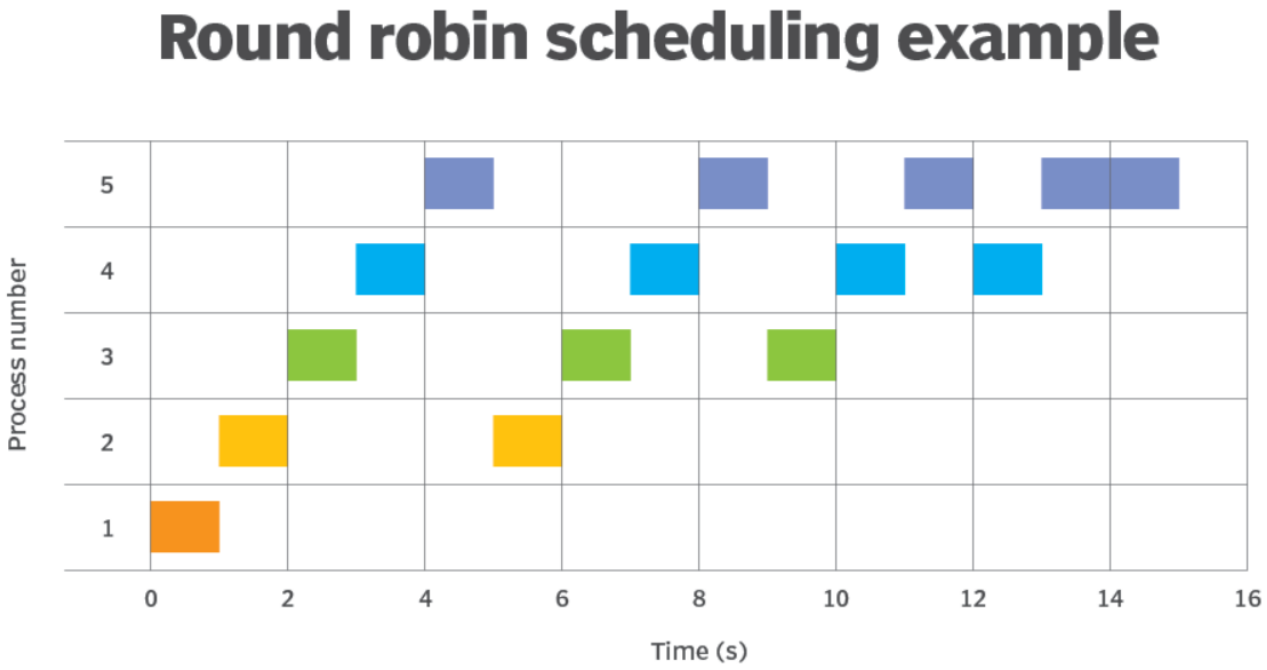

```
// code of the printer
while (1) {
  if (PC1_request_print == true)   Print_task_of_PC1();
  if (PC2_request_print == true)   Print_task_of_PC2();
  if (PC3_request_print == true)   Print_task_of_PC3();
  if (PC4_request_print == true)   Print_task_of_PC4();
}
```

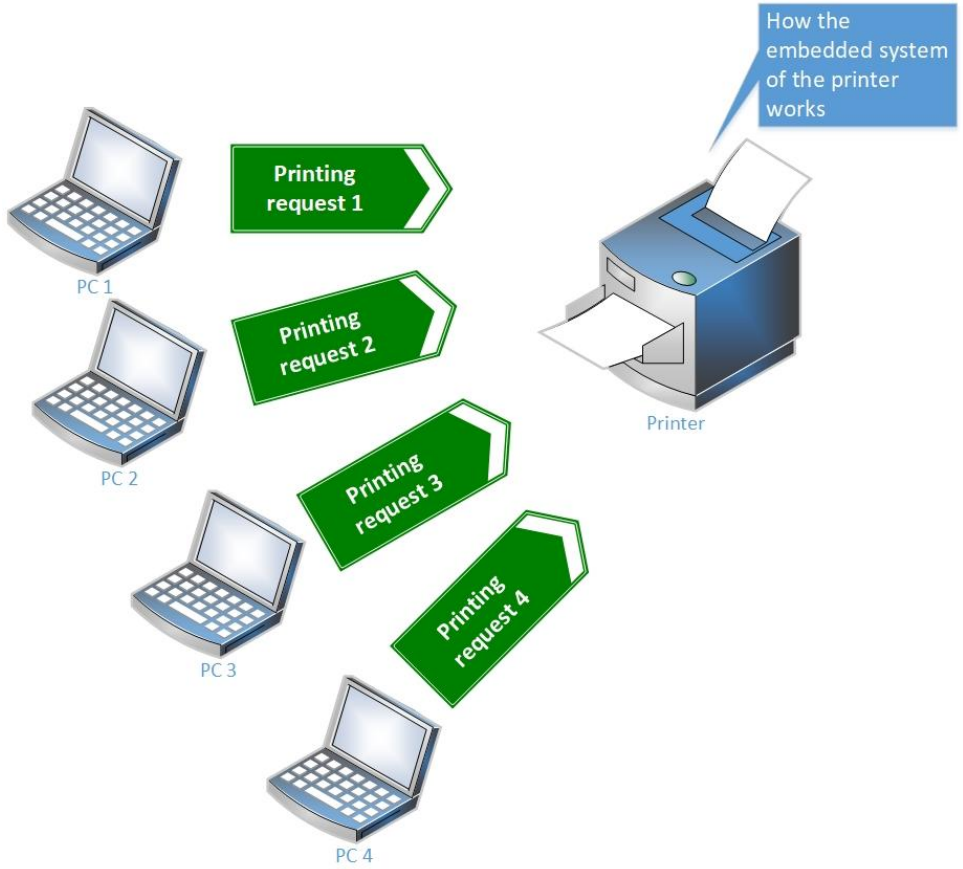
Free RTOS



The printer is a shared resource

Round Robin







Free RTOS

Round Robin

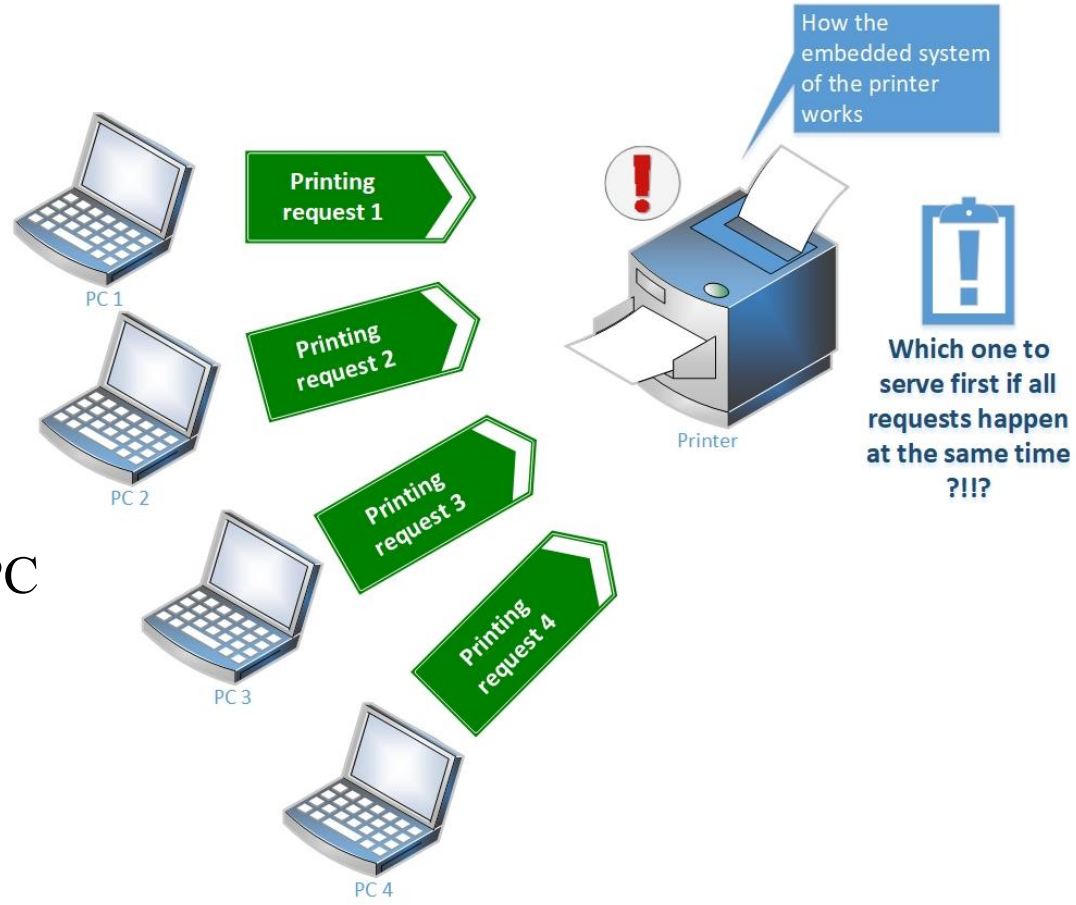
The worst case scenario happens when all PCs request printing at the same time ?!!

Now we have to check how much the printing service requires time to be accomplished for one PC

For example, 5 second for each printing task for each PC

Is the last PC able to wait for 15 seconds (5s +5s +5s) until processing its task or not !?

If yes, the system will be a real time system,





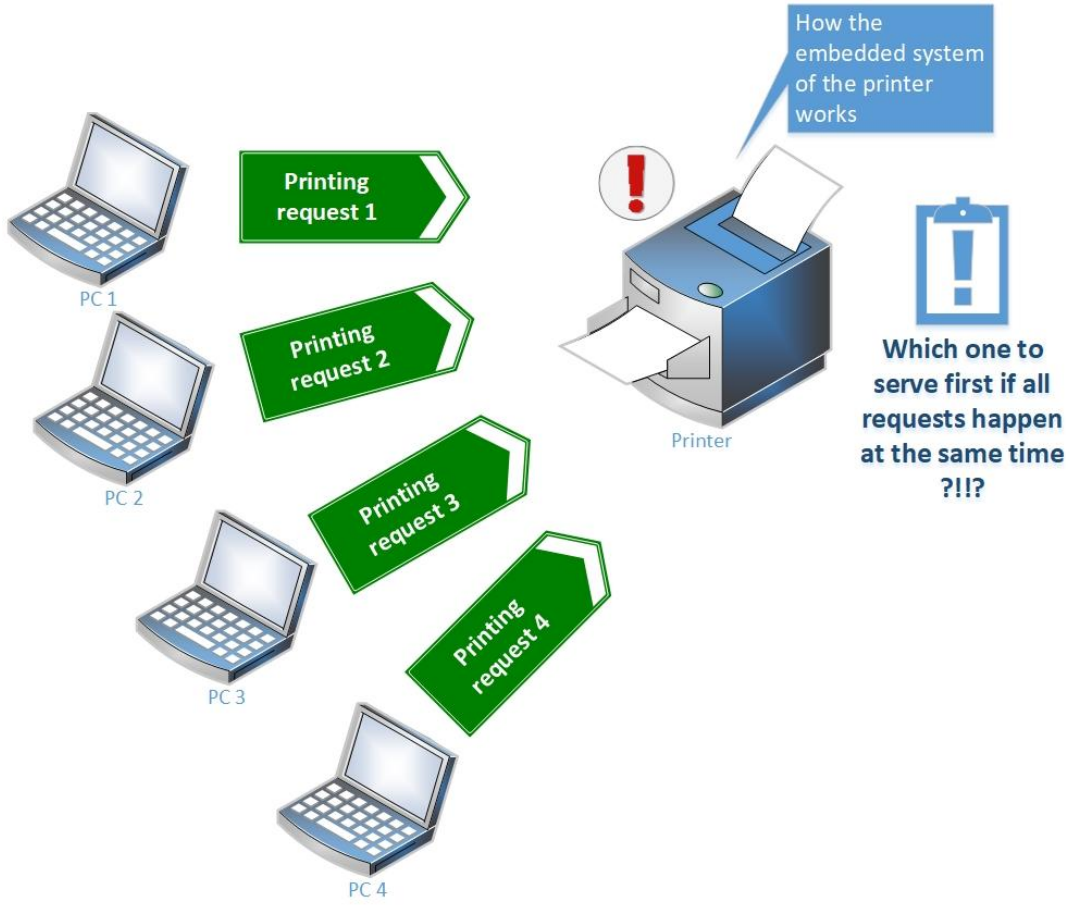
Free RTOS

Round Robin

Thus, in this case, each PC has a deadline of waiting for 15 seconds

Now if the deadline of PCs is less than 15 second, the system will be suitable for real time processing.

If for any reason, the process of printing of any PC takes more than 5 seconds, the total time will be bigger than 15 seconds, and the processor won't always serve the last PC during its deadline time frame



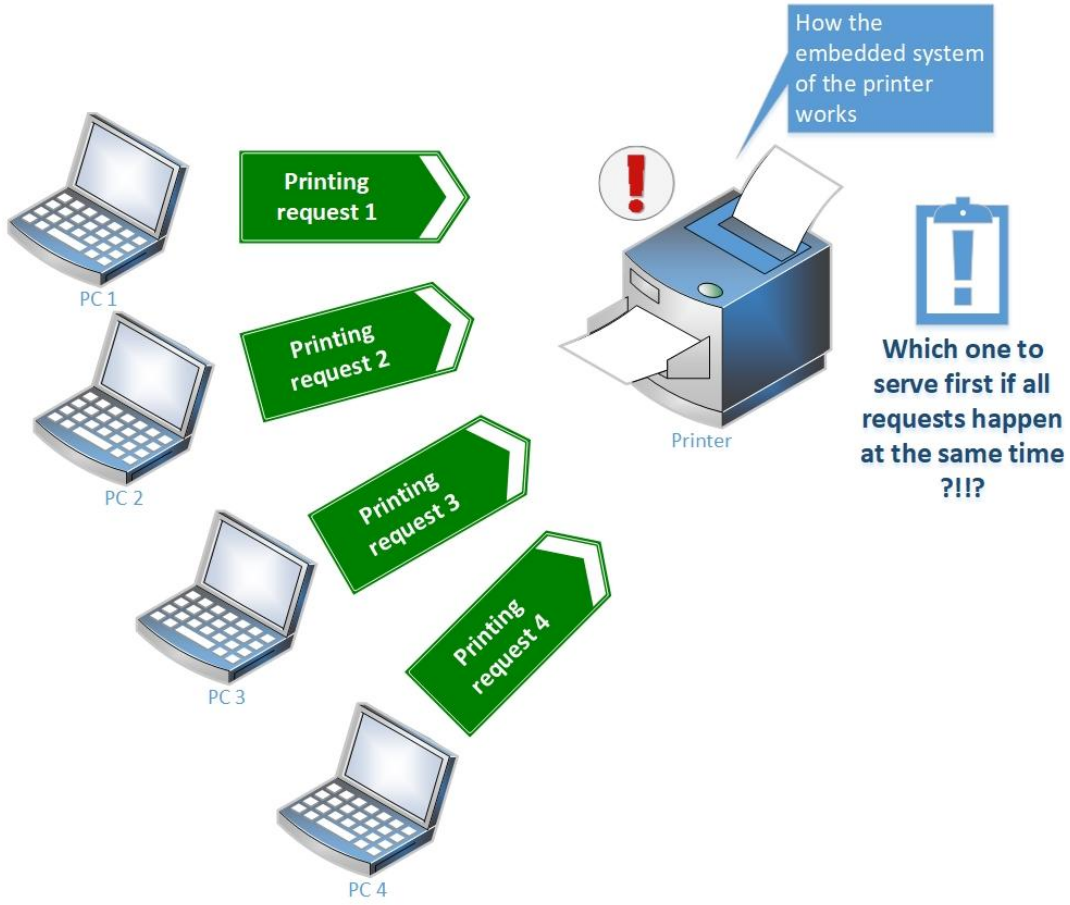


Free RTOS

The major question here is

What if the request is urgent (cannot wait for 15 sec)?

What if we have a priority request ?

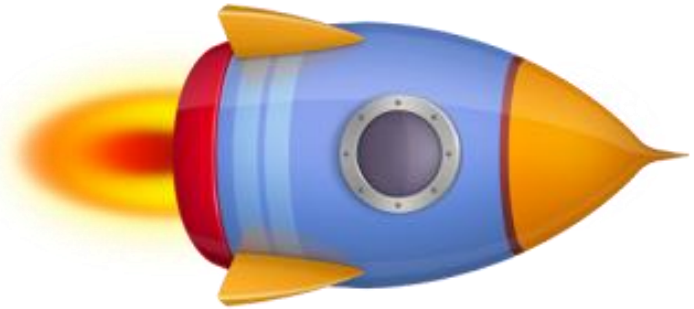




Free RTOS

- Let's take another example from space exploration. Say we are sending a rocket to collect data about some outer planets in our solar system. As it travels, one of its tasks is to continuously click pictures and send it to us. Another task is to avoid asteroids as it travels so that it does not hit anything and get destroyed in the process, so it also has the scan the field ahead to see there are rocks floating around.
- Okay, what if there is an asteroid dead ahead? Then it has to take evasive maneuvers by changing its direction.
- **So far we have 3 tasks**

1. Continuously click and send pictures
2. Continuously scan the path for asteroids
3. If asteroids are found on the path, take evasive maneuvers





Free RTOS

- Out of these 3, task #3 is the highest priority, task#2 is the next and task#1 is the lowest priority. Since we need the rocket to be safe, we don't care if it is not able to send pictures for the 10mins it spends on escaping from the asteroid since these missions usually last for decades and loss of 10min data means nothing compared to the safety of the satellite.
- **In FreeRTOS the higher the number, the higher the priority**. So a priority table can look like this.

Task	Priority number
Evasive maneuvers	10
Scan path	8
Click and send pics	7

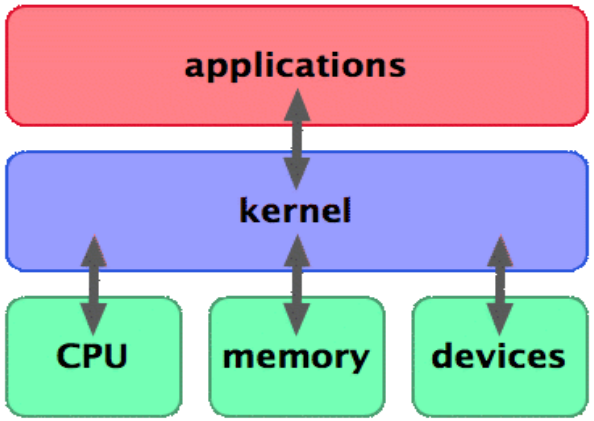
Basically what this means is, we are saying the system to **drop everything and run if an asteroid approaches** it!

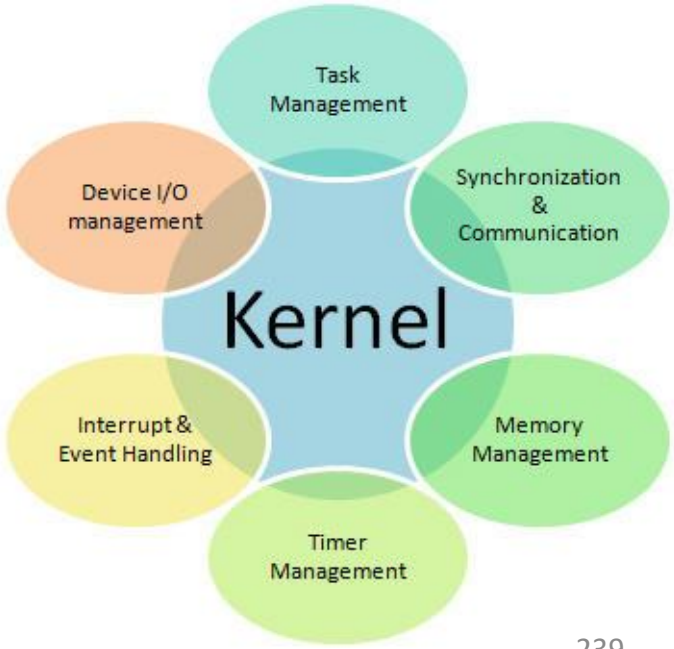


Free RTOS

Kernel

- The kernel is considered to be the core of an operating system. Its main task is to manage the hardware and the processes running on it.
- The scheduler we saw above is the most important part of a kernel.
- Other important parts include inter-task communications and synchronizations.







Free RTOS

Inter-Task Communication

- Let's go back to our satellite example. There the "scan path" task must inform the "take evasive maneuver" task if it actually finds some obstacle. So there is a need for the 2 tasks **to share** some information with each other.
- There are several options available for tasks to communicate with each other through the kernel of FreeRTOS like queues, mutex, semaphores and notifications. Let's take a brief look at them.



Free RTOS

Queues

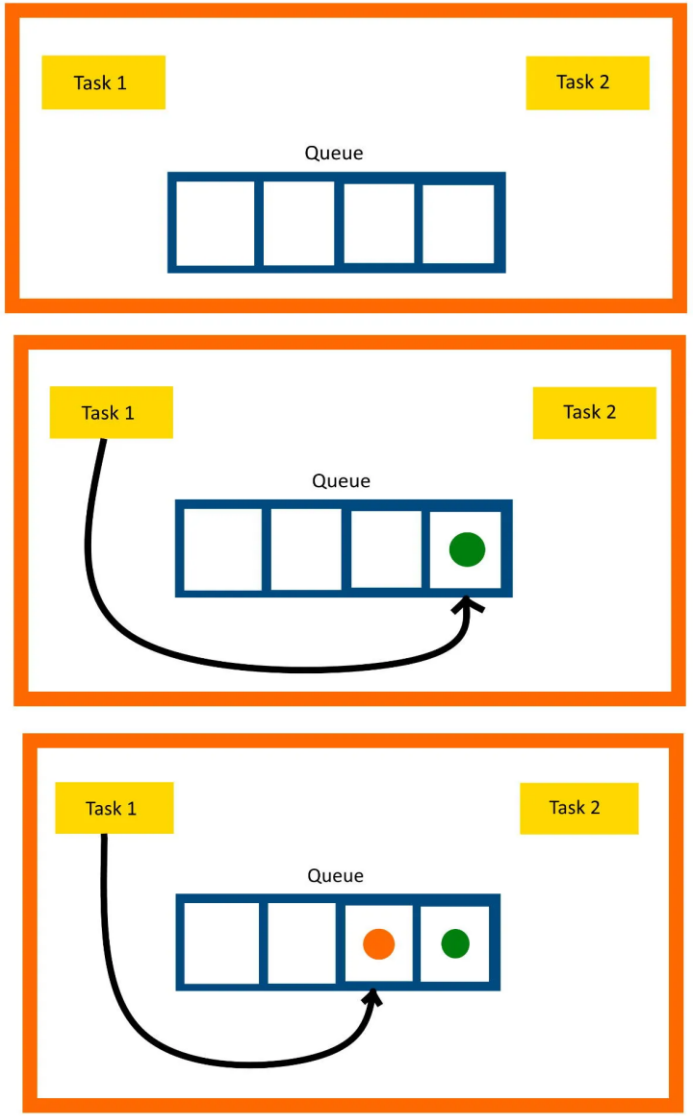
- To explain queues let's take another example. Clicking and sending pictures from the satellite can also be implemented as 2 separate tasks.
- Let's assume our satellite is going past Saturn and we will be able to get a very good view of it in the next 2 hours.
- Let's assume our camera can click 10 pics per second but our radio can only send 1 pic per second.
- So in that situation, we may need to continuously click pictures and store them in a memory buffer, so that it can send it one by one later when the processor is free.
- This is where queues come in.



Free RTOS

Queue illustration 1

- So the operation of the queue goes like this
  - the queue is empty at the beginning
  - task one is running and it produces the green ball and puts it in the front of the queue
  - task one produces another ball, this time an orange one and puts that also in the queue



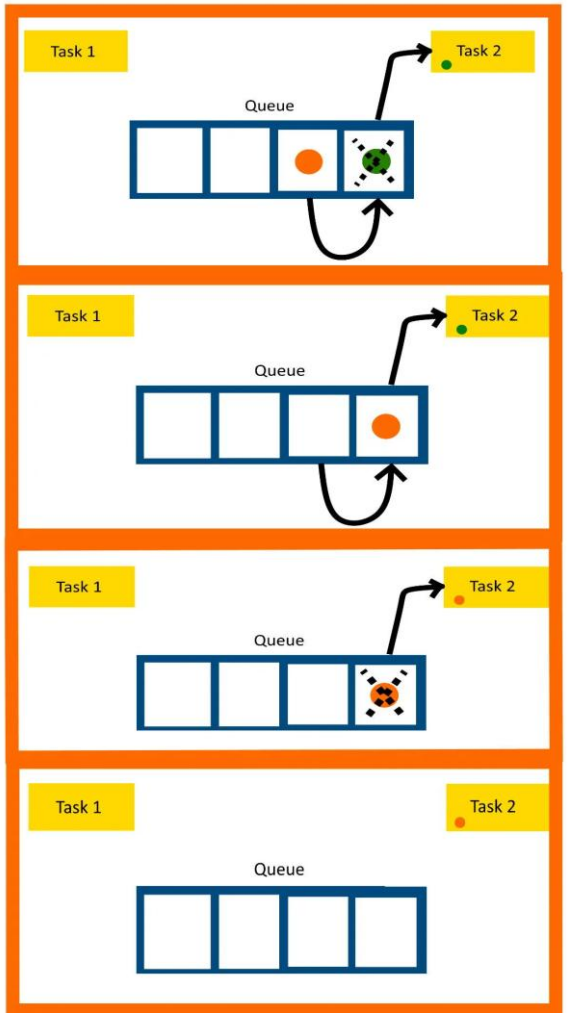


Free RTOS

Queue illustration 2

- next task 2 starts running and it notices that there are 2 balls in the queue
- first, it takes the green ball and processes it thereby removing it from the queue
- so once the green ball is removed, the kernel automatically move the orange ball to the front of the queue
- once task 2 finishes processing the data in the green ball, it goes ahead and takes the orange ball from the queue then the queue is empty again.
- It's a simple operation no different than standing in a queue to buy something at a shop. It works on a First-In-First-Out basis.







Free RTOS

The text file is a shared resource

Mutex

- Let's take the example of printing something to a **shared text file**. Consider 2 tasks, Task#1 and Task#2.
- They both are executing pseudo-parallelly and they can both read and write to this text file.

- Task#1 will print the line "1 is the best number!" and
- Task#2 will print the line "**2 is bigger than one**"
- in bold, and they will both print continuously onto the text file.

```
void task1()
{
    while (true)
    {
        print(abc.txt, "1 is the best number\n");
    }
}

void task2()
{
    while (true)
    {
        print(abc.txt, "2 is bigger than one\n");
    }
}

void main()
{
    start_scheduler();
    start_task(task1);
    start_task(task2);
}
```



Free RTOS

Mutex

- Consider both these tasks to be of the same priority. Now as we saw previously, the scheduler will make sure each will get its turn to run since they are of the same priority.
- So if the scheduler decides to switch in the middle of printing as the Task#1 is running you will get something like this
- 1 is the best number!
- 1 is **2 is bigger than one**
- **2 is bigger than one**
- **2 is big**1 is the best number!
- 1 is the best number!
- Which makes no sense. Just imagine 100 Tasks writing to this text file instead of 2, which is very normal for a Windows or Linux Operating systems that run on our computers. Unless we install some policies this multiple writer scenario will probably render the text file corrupted and totally useless!

```
void task1()
{
    while (true)
    {
        print(abc.txt, "1 is the best number\n");
    }
}

void task2()
{
    while (true)
    {
        print(abc.txt, "2 is bigger than one\n");
    }
}

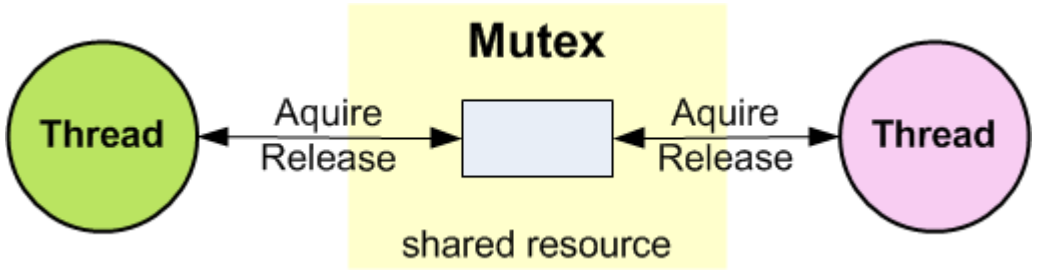
void main()
{
    start_scheduler();
    start_task(task1);
    start_task(task2);
}
```



Free RTOS

Mutex

- **What is Mutex?** Mutex stands for **MU**Tual **EX**clusion. It is a type of variable that **controls access** to shared resources.
- Let's learn more about it by looking at an example.
- Let's take the example of printing something to a shared text file. Consider 2 tasks Task#1 and Task#2. They both are executing pseudo-parallelly and they can both read and write to this text file.
- So we need some sort of mechanism to control the access to the text file so that if one task starts writing to it, the other tasks that need the write-access must wait before the previous task has given up its write-access.




```
void task1()
{
    while (true)
    {
        print(abc.txt, "1 is the best number\n");
    }
}

void task2()
{
    while (true)
    {
        print(abc.txt, "2 is bigger than one\n");
    }
}

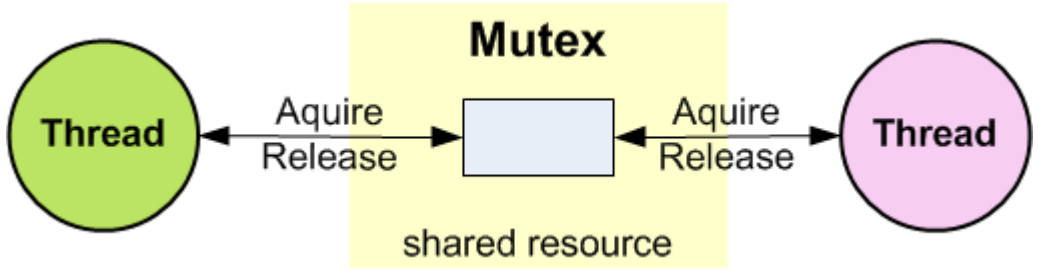
void main()
{
    start_scheduler();
    start_task(task1);
    start_task(task2);
}
```



Free RTOS

Mutex

- This is actually simple to do using a simple Boolean status flag called **file_in_use** as seen in the pseudo-code below.
- Now, this status flag is a global variable which makes it easier to maintain the "one at a time" write-access to the abc.txt file.
- In other words, a mutex is just a more robust version of our status flag. They act as keys to our shared object. There is only one key, so only one person can access the resource at a time.




```
bool file_in_use;

void task1()
{
    while (true)
    {
        if (file_in_use == false) {
            file_in_use = true;
            print(abc.txt, "1 is the best number\n");
            file_in_use = false;
        }
    }
}

void task2()
{
    while (true)
    {
        if (file_in_use == false) {
            file_in_use = true;
            print(abc.txt, "2 is bigger than one\n");
            file_in_use = false;
        }
    }
}

void main()
{
    start_scheduler();
    start_task(task1);
    start_task(task2);
}
```



Free RTOS

Semaphores

- Semaphores are just an extended version of mutexes. The shared resource guarded by a mutex has only one key. On the other hand, the resource guarded by semaphores can have multiple keys, so that a number of processes can access that particular resource simultaneously.

- Think of an elevator, which can support only 4 people at a time. Here the elevator is the shared resource. If you need access to it, you need to press the button outside the elevator signaling the elevator system that you need to use it.

- Now if there is space in the elevator once it opens, say only one person is already inside, then you can go in.

- But if there are already 4 people inside the elevator, then you can either wait and take the next one.

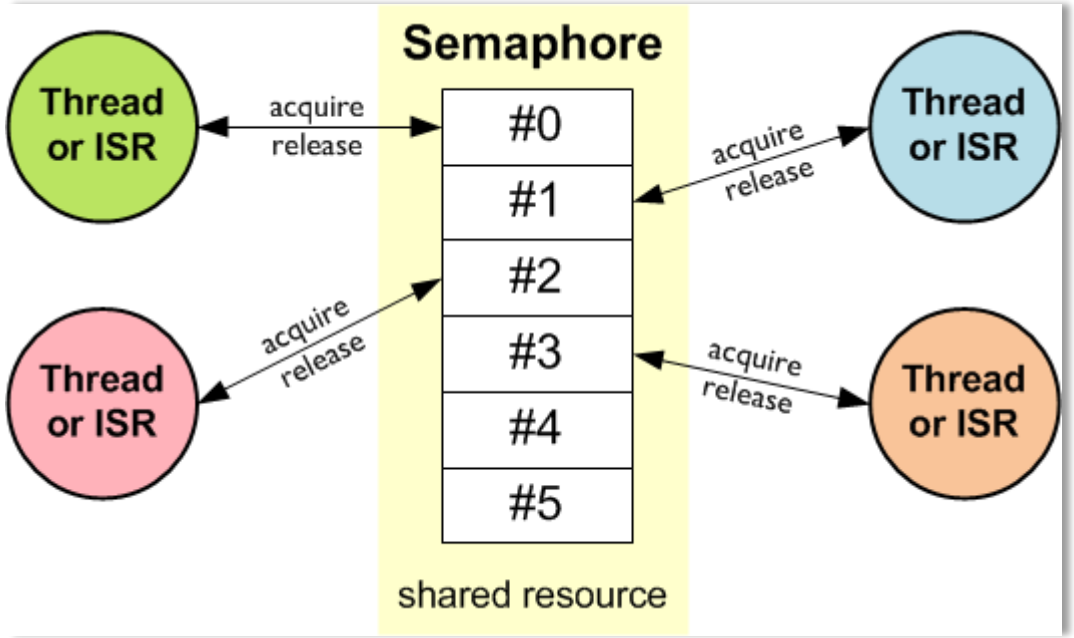
The elevator is a shared resource





Free RTOS

- Semaphores work much the same way. The elevator control system is analogous to the kernel and the number of spaces available at any given time is analogous to the number of free keys available to our shared resource at a given instant.
- If you need a key, you need to ask the kernel and if one is available, it will be given to you.





Free RTOS

- Let's take the same text file example that we saw in the previous section and see how controlled access to multiple tasks will be useful. If a task needs to write to it, then of course only one should do it at a time, but if a task just needs to read it, then it does not change the contents of the file so in this case, more than one task can read the text file at any given instant.
- So here this shared text file will have just 1 mutex key to control the write-access and depending on the use case maybe 5 semaphore keys to control the reading access of the file.
- Since mutex and semaphores are so similar to each other, **mutexes are also called binary semaphores.** Collectively **mutexes, semaphores, queues, etc., are called communication objects** as they are used to communicate events, statuses and data from one task to another task.



Free RTOS

Interrupts

- Interrupts behave in a way similar to the task switching mechanism available in the scheduler so that the currently running code is preempted if an interrupt occurs,
- Interrupts should not be confused with task switching.
  - task switching is done by the scheduler through software to make sure that a lower priority task cannot run while another higher priority task is ready to run.
  - Interrupts are produced by hardware events like a user pressing a button.
  - Interrupts are used to capture events from the physical world the device is working in while the task switching is done primarily for multitasking.



Free RTOS

Interrupt controllers

- **Interrupt controllers listen to the peripherals for events and reports to the processor once an event occurs.**
- Lets first talk about what interrupts are by considering some real-life examples of interrupts.
  - You are sleeping and in the morning the alarm you set goes off and wakes you up (by interrupting your sleep)
  - You are reading a book and your phone rings to alert you of a call (by interrupting your reading session.)



Free RTOS

Interrupt controllers

- This concept has been ported to embedded systems using special devices called interrupt controllers. I like thinking of them as event listener peripherals. Their main job is to listen to the events generated by the other peripherals and report them to the processor.
- Examples of events that can produce interrupts include
  - GPIO reads 1 or 0
  - Timer countdown reached 0
  - Serial communication received a packet of data and
  - DMA has completed a transfer
- The particular event that we need to listen to can be programmed by us by registering them to the interrupt controller and unnecessary events can be ignored.
- Once a registered event happens, the interrupt controller goes and tells the CPU about it, so that the CPU can take appropriate actions.



Week 6

Lab 5 / LO3

Working With FREE RTOS



Tools and components

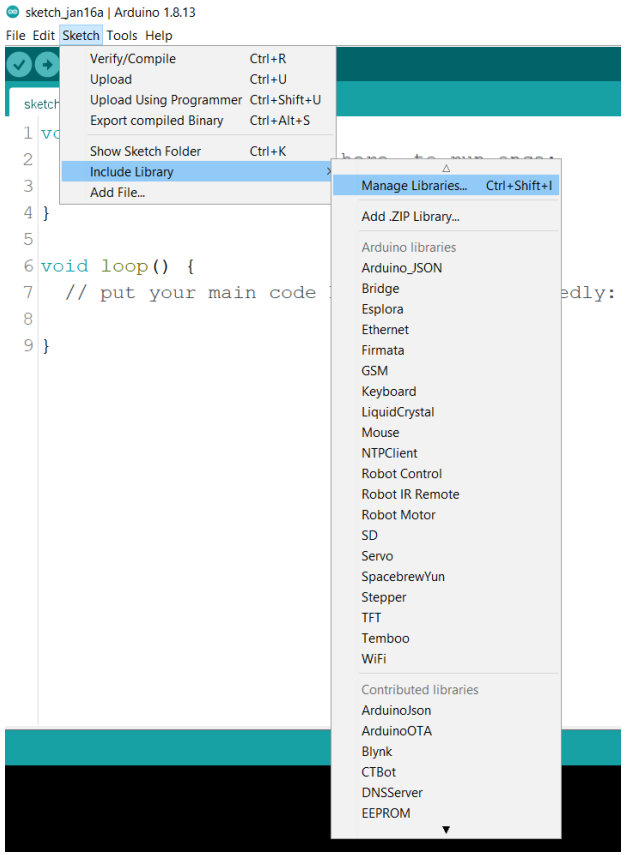
- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. Breadboard wires (male-male)
  4. Oscilloscope

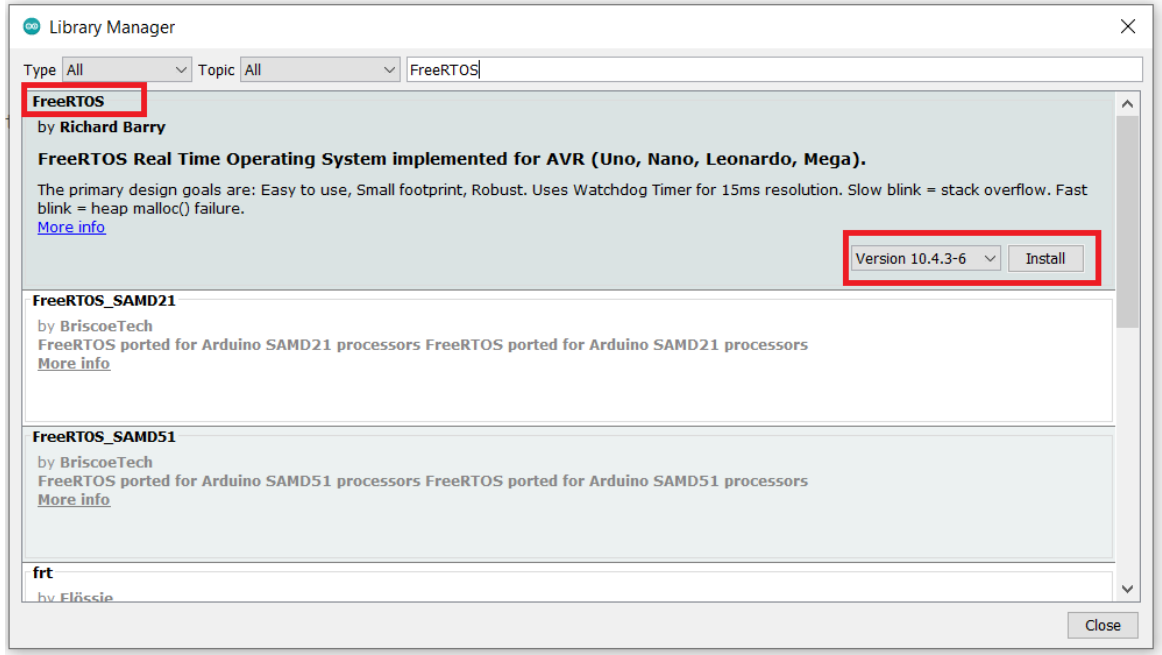
- Required components:
  1. Arduino UNO + USB cable
  2. Led x2
  3. Resistor from 200 to 470 ohm x 2



Working with Free RTOS

First lets install the library from the Arduino IDE



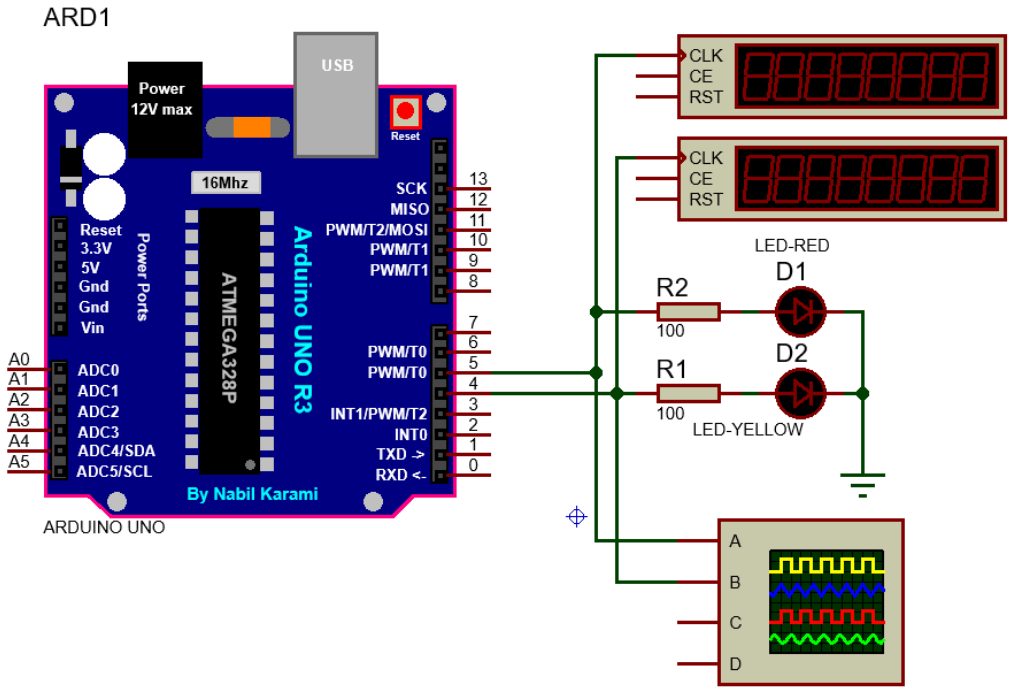




Working with Free RTOS

Let us take the same example of blinking two leds at different frequencies at the same time… but now with Free RTOS.

The task is to blink the leds at 2Hz and 5Hz (ie the period of the first led is 0.5sec and the second led is 0.2sec)





Working with Free RTOS

```
1  #include <Arduino_FreeRTOS.h>
2  #define led1 4
3  #define led2 5
4
5  void setup() {
6    pinMode(led1, OUTPUT);
7    pinMode(led2, OUTPUT);
8    xTaskCreate(Led1Task, "Led 1 Task", 128, NULL, 1, NULL);
9    xTaskCreate(Led2Task, "Led 2 Task", 128, NULL, 1, NULL);
10 }
11 void loop() {}
12
13 void Led1Task(void *pvParameters) {
14   while (1) {
15     digitalWrite(led1, HIGH);
16     delay(250);
17     digitalWrite(led1, LOW);
18     delay(250);
19   }
20 }
21 void Led2Task(void *pvParameters) {
22
23   while (1) {
24     digitalWrite(led2, HIGH);
25     delay(100);
26     digitalWrite(led2, LOW);
27     delay(100);
28   }
29 }
30
```

- The code starts by using the Arduino_FreeRTOS library
- Tasks are defined for LED1 and LED2
- The loop function is empty
- Each task is written as nothing else is in code

Using an infinite while loop

Lab

Working with Free RTOS

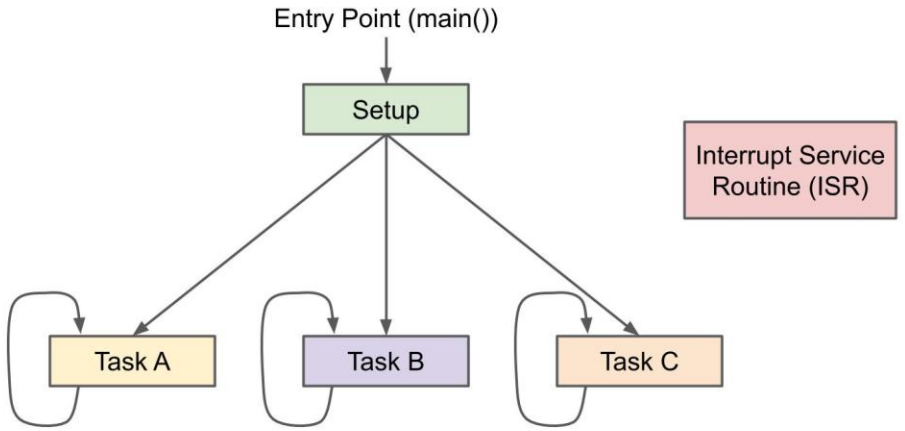
```
#include <Arduino_FreeRTOS.h>
#define led1 4
#define led2 5

void setup() {
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
  xTaskCreate(Led1Task, "Led 1 Task", 128, NULL, 1, NULL);
  xTaskCreate(Led2Task, "Led 2 Task", 128, NULL, 1, NULL);
}
void loop() {}

void Led1Task(void *pvParameters) {
  while (1) {
    digitalWrite(led1, HIGH);
    delay(250);
    digitalWrite(led1, LOW);
    delay(250);
  }
}
void Led2Task(void *pvParameters) {

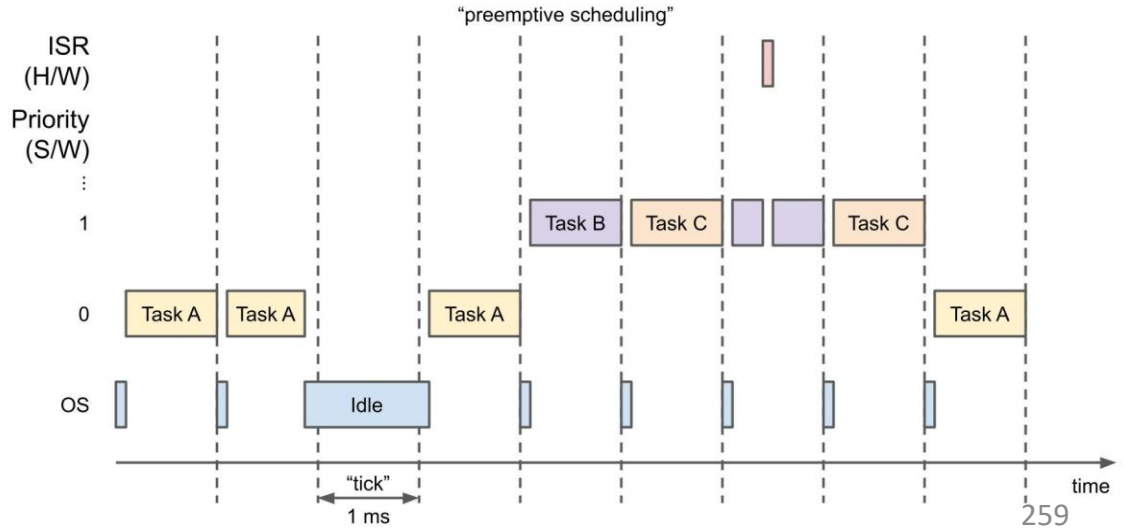
  while (1) {
    digitalWrite(led2, HIGH);
    delay(100);
    digitalWrite(led2, LOW);
    delay(100);
  }
}
```

What our code looks like



What actually happens*

*assuming single-core processor





Working with Free RTOS

```
8   xTaskCreate(Led1Task, "Led 1 Task", 128, NULL, 1, NULL);
```

It is simply a pointer to the function that implements the task (Just the name of the function).

It is a descriptive name for the task. This is not used by FreeRTOS. It is included purely for debugging purpose. It can also be used to obtain the task handle.

Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The value specifies the number of words the stack can hold, not the number of bytes.

A value that is passed as the paramater to the created task.

It is priority of the task (0 is the lowest priority)

Used to pass a handle to the created task out of the xTaskCreated() function. It is a optional and can be set to NULL.



Week 7

Lecture 7 / LO3

A2D conversion



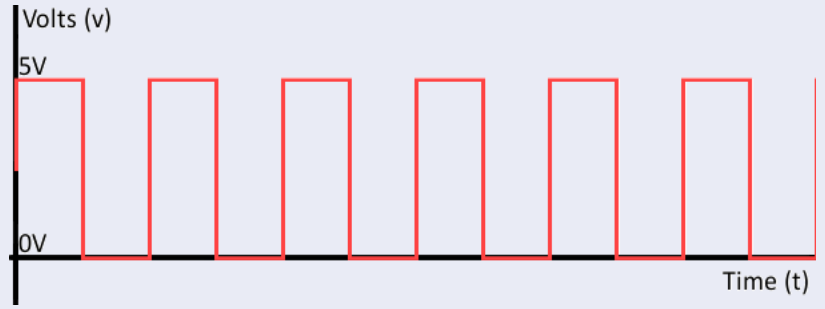
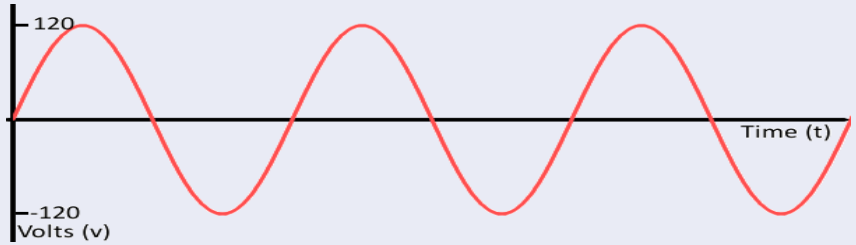
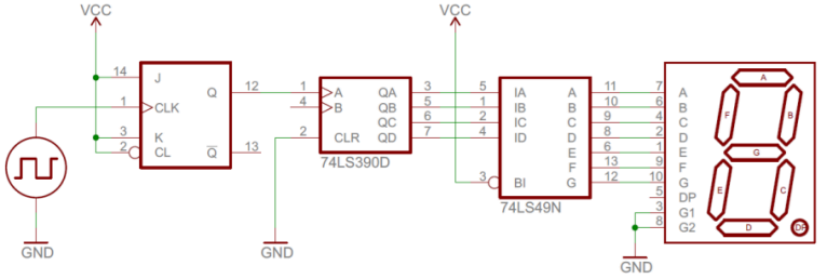
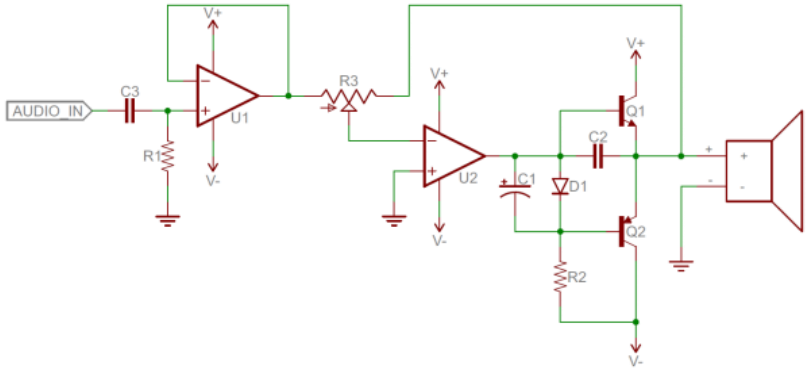
Analog To Digital Conversion

- Microcontrollers are capable of detecting binary signals: is the button pressed or not? These are digital signals.
- When a microcontroller is powered from five volts, it understands zero volts (0V) as a binary 0 and a five volts (5V) as a binary 1.

- What if the signal is 2.72V? Is that a zero or a one?
- We often need to measure signals that vary; these are called analog signals.
- A 5V analog sensor may output 0.01V or 4.99V or anything in between.
- Luckily, nearly all microcontrollers have a device built into them that allows us to convert these voltages into values that we can use in a program to make a decision.



Analog To Digital Conversion

	Digital Signal	Analog Signal
Format	0 & 1	time-varying “quantities”
Graphs		
Circuit		
Components	Microcontrollers, logic gates, chips	Resistors, capacitors, inductors, op-amp
Characteristics	Easy to design, but expensive	Very sensitive to noise



Analog To Digital Conversion

- A digital signal is superior to an analog signal because it is
  - more robust to noise
  - can easily be recovered, corrected and amplified.
  - compatible with digital electronics e.g. computers, microcontrollers

- For this reason, the tendency is to change an analog signal to digital data.



Analog To Digital Conversion

- An Analog to Digital Converter (ADC) is a very useful feature that converts an analog voltage on a pin to a digital number.
- By converting from the analog world to the digital world, we can begin to use electronics to interface to the analog world around us.
- Arduino UNO has 6 analog pins (A0… A5)





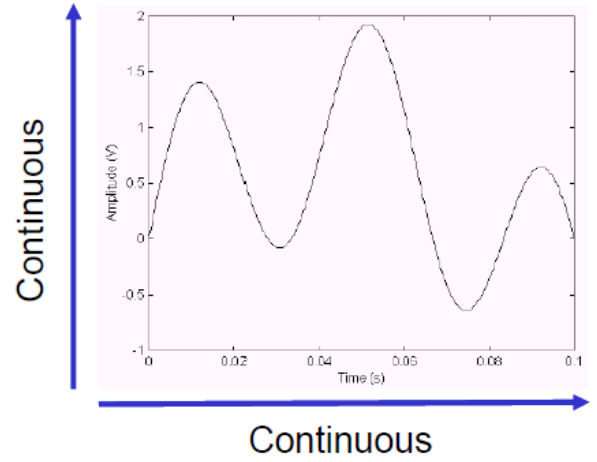
Analog To Digital Conversion

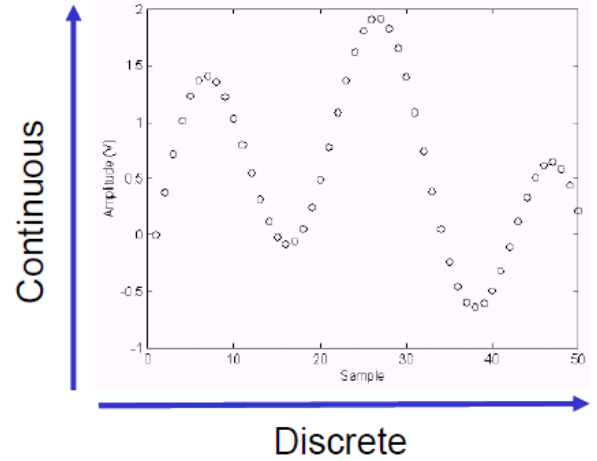
- ADCs can vary greatly between microcontroller.
- The ADC on the Arduino is a 10-bit ADC meaning it has the ability to detect 1,024 (2^{10}) discrete analog levels.
- Some microcontrollers have 8-bit ADCs ($2^8 = 256$ discrete levels)
- and some have 16-bit ADCs ($2^{16} = 65{,}536$ discrete levels).



Analog To Digital Conversion

- The digital representation of the signal on the other hand is different in two important respects:
  - It is sampled: it has been modified from continuous time to discrete time
  - It is quantized: it has been modified from continuous values to discrete levels
- Both of these operations restrict how much information a digital signal can contain.

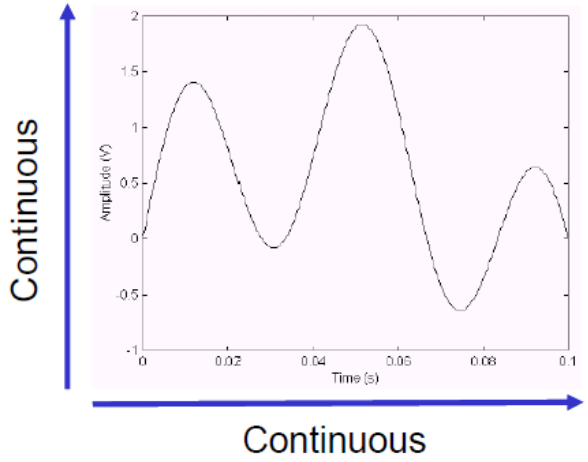


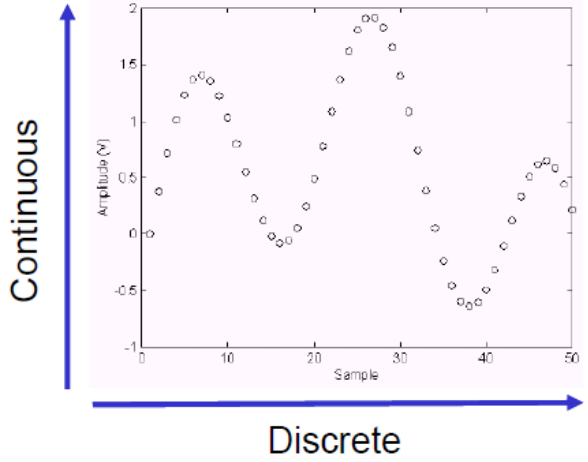




Analog To Digital Conversion

- A discrete-time signal is defined for only discrete instants of time.
- In other words, the "time variable" of the signal may take on only certain values.
- The process of converting an analog signal into a discrete-time signal is called *sampling*.
- Sampling converts the independent variable (time in this example) from continuous to discrete.







Analog To Digital Conversion

Nyquist Sampling Rate

- In signal processing, the Nyquist rate is the minimum sampling rate required to accurately rebuild the original analog signal from its digital representation.
- To avoid errors, the rate at which you sample a signal must exceed twice the highest meaningful frequency contained within the signal.

$$F_{sampling} > 2F_{max}$$

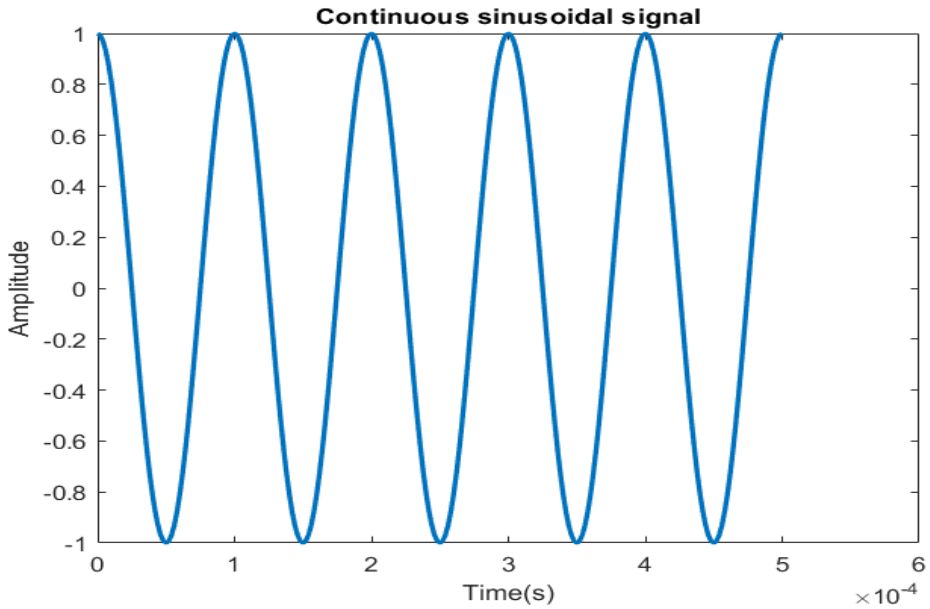
- Or, if the sampling rate is fixed (*Fsampling*), then the fastest signal that can be sampled must have a maximum meaningful frequency less than half *Fsampling* :

$$F_{max} < \frac{F_{sampling}}{2}$$

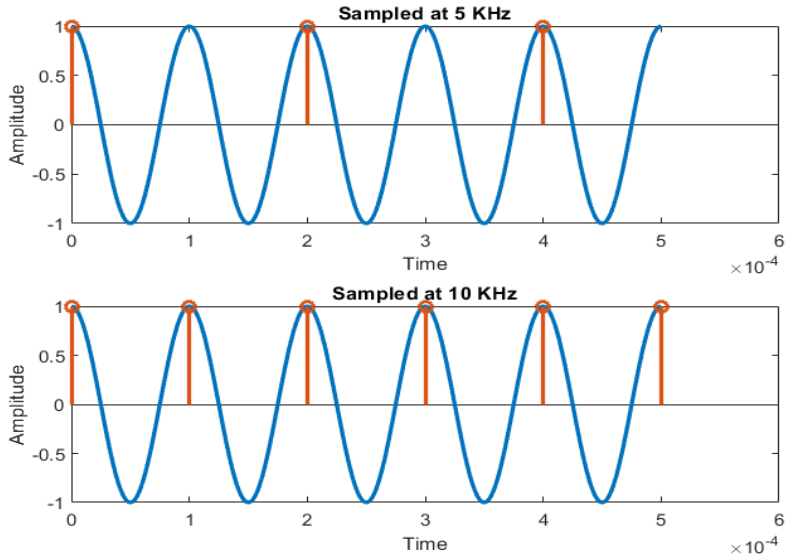


Analog To Digital Conversion

Nyquist Sampling Rate



Here is the **10 KHz signal** (the **maximum frequency** of the signal is 10 KHz):



Now, if we sample the signal at 5 kHz and 10 kHz, the signal will look like as shown below (the brown points)

It is clear that we can't get any useful information from the sampled signal.

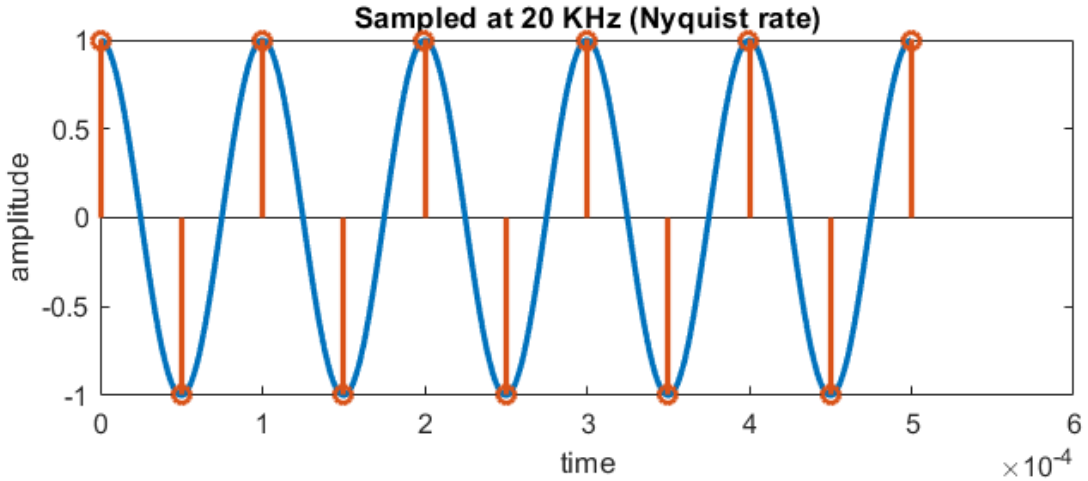


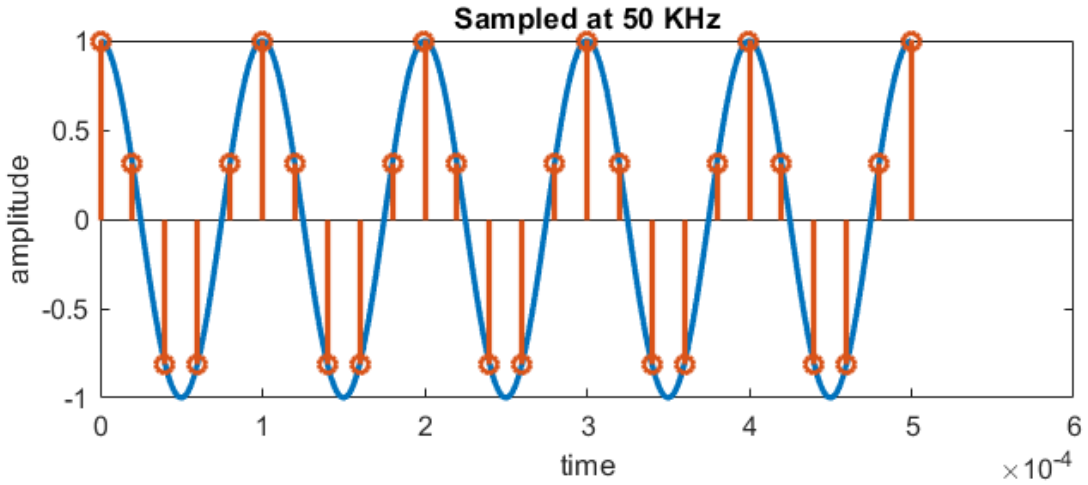
Analog To Digital Conversion

Nyquist Sampling Rate

Now, to get some information about the signal, Nyquist said that we must sample the signal at least **2 x max signal frequency**. Now, here are the signals sampled at the **Nyquist frequency** and more than it:

It can be seen that by sampling at the Nyquist rate, we can get the frequency information about the signal. However, to faithfully reconstruct the signal, we have to increase the sampling rate even more.

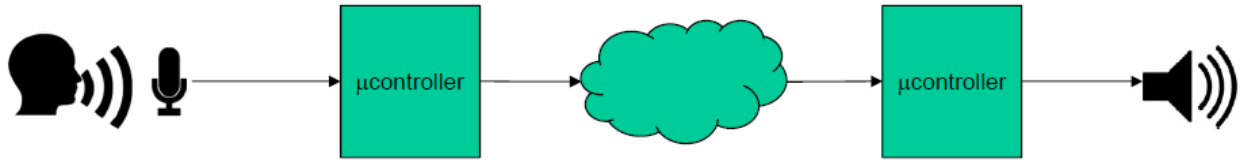






Analog To Digital Conversion

Nyquist Sampling Rate Example



We want to digitize a speech signal.

- The highest meaningful frequency in human speech occurs at 4 kHz. • Therefore, *Fmax*= 4 kHz
- We must sample at least twice as fast. => Therefore, $F_{sampling} = 2 \times 4\ KHz = 8\ KHz$
- Time between samples ($T_{sampling} = 1/F_{sampling} = 0.125\ ms$)
- Therefore, you must take a sample every 0.125 msec in order to preserve the important information in the original human speech signal to be able to recreate it.



Analog To Digital Conversion

Quantized (Discrete Value) Signals

- We cannot represent the unlimited possible values in a continuous value analog signal.
- We have to reduce the complexity by changing the signal from continuous value to discrete values.
- We will approximate the value of the signal at any point in time to one of a set of possible levels
- This process is called quantization
- For example, 2.56000 volts, 2.56001 volts, and 2.56002 volts of an input voltage will be approximated as 2.56 volts.
- There is a certain unavoidable error that is introduced by the quantization process.
- This is known as the “quantization error”.



Analog To Digital Conversion

- **Relating ADC Value to Voltage**
- The ADC reports a *ratiometric value*. This means that the ADC assumes 5V is 1023 and anything less than 5V will be a ratio between 5V and 1023.

$$\frac{2^{\#ofbits}-1}{V^+-V^-}=\frac{ADCreading}{V_{in}-V^-}\rightarrow ADCreading\ =(2^{\#ofbits}-1)\frac{(V_{in}-V^-)}{V^+-V^-}$$

- Analog to digital conversions are dependent on the system voltage. Because we predominantly use the 10-bit ADC of the Arduino on a 5V system, we can simplify this equation slightly:

$$ADC\ reading\ =1023\frac{(V_{in})}{5}$$

- If the system is 3.3V, simply change 5V out with 3.3V in the equation.
- If the system is 3.3V and the ADC is reporting 512, what is the voltage measured? It is approximately 1.65V.



Analog To Digital Conversion

- On the other side, if we have the read value from the Arduino and we need to figure out what is the input voltage, the following equation can be used

$$V_{in} = \frac{(V^+ - V^-) \times ADCreading}{(2^{\#ofbits} - 1)} + V^-$$

with

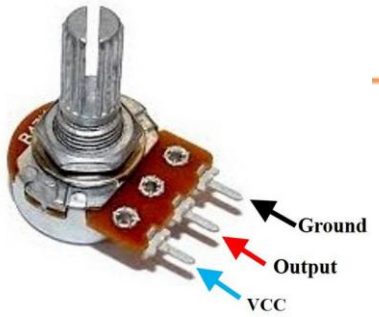
$$0 < ADC_{reading} < 2^{\#ofbits} - 1$$

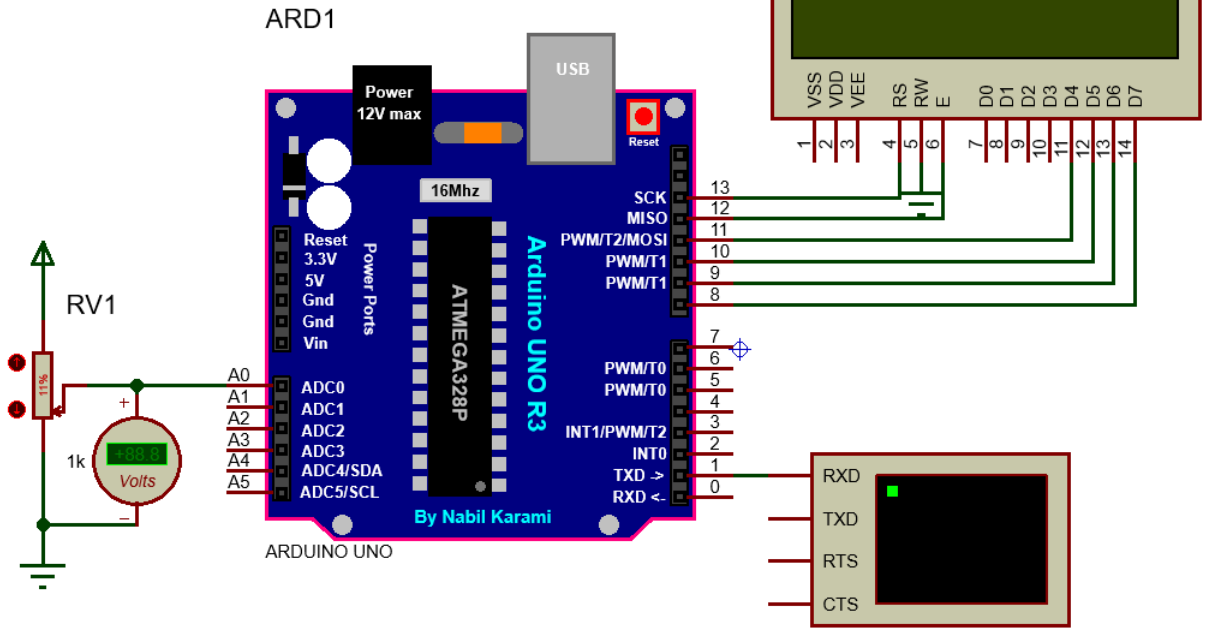
$$V^- < V_{in} < V^+$$



Demo A2D code

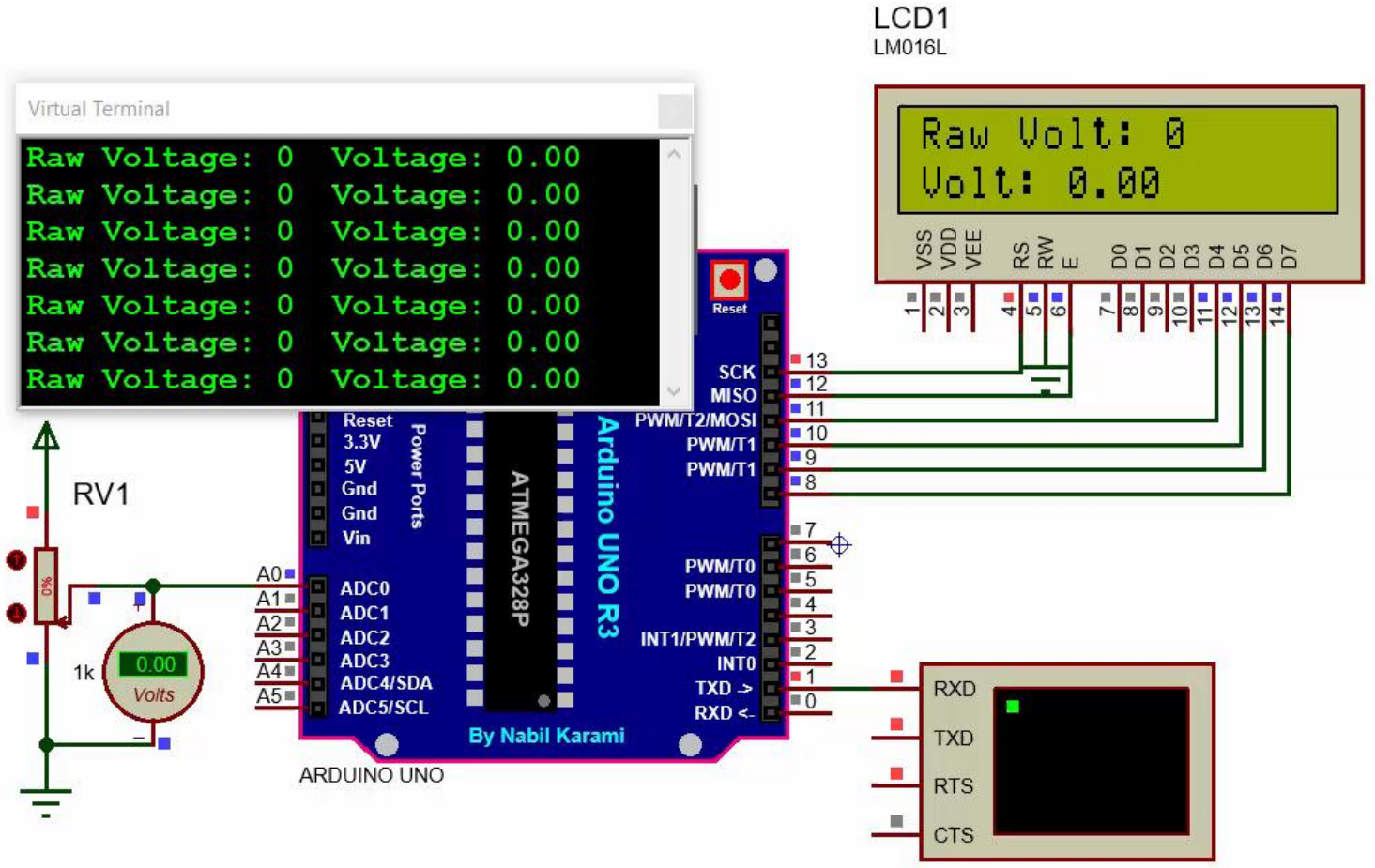
```
1  #include <LiquidCrystal.h>
2  LiquidCrystal lcd(13, 12, 11, 10, 9, 8);
3
4  int raw_voltage;
5  float voltage;
6  void setup() {
7    Serial.begin(9600);
8    lcd.begin(16, 2);
9  }
10
11 void loop() {
12   raw_voltage = analogRead(A0);
13   voltage = (float)raw_voltage * 5.0 / 1023.0;
14   lcd.clear();
15   lcd.setCursor(0, 0);
16   lcd.print("Raw Volt: "); lcd.print(raw_voltage);
17   lcd.setCursor(0, 1);
18   lcd.print("Volt: "); lcd.print(voltage, 2);
19   Serial.print("Raw Voltage: "); Serial.print(raw_voltage);
20   Serial.print("  Voltage: "); Serial.println(voltage, 2);
21   delay(100);
22 }
```







Demo A2D code



Click to Play video



Week 7

Lab 6 / LO3

Working With A2D/ Temperature sensor



Tools and components

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. Breadboard wires (male-male)

- Required components:
  1. Arduino UNO + USB cable
  2. Potentiometer 1k to 10k ohm x1
  3. LM35

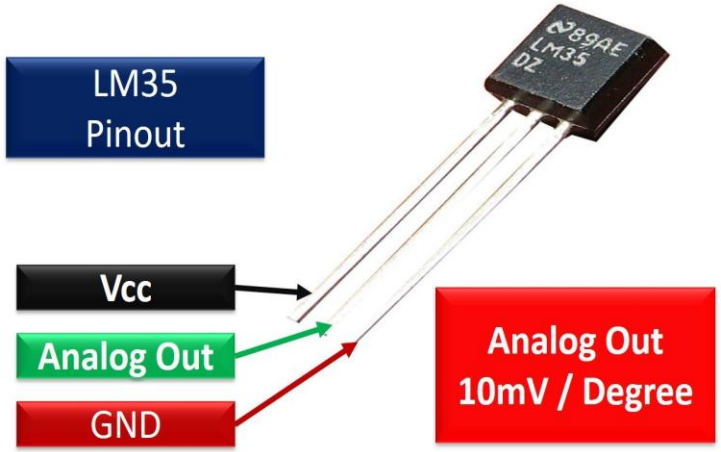


A2D example

Temperature reading using LM35 temperature sensor

Temperature Sensor	Output Voltage	Linearity	Range
LM335	Proportional to temperature in Kelvin (°K)	10mV/°K	−40°C to 100°C
LM34	Proportional to temperature in Fahrenheit (°F)	10mV/°F	−50° to 300°F
LM35	proportional to temperature in Celsius (°C)	10mV/°C	−55°C to 150°C
LM356	proportional to temperature in Celsius (°C)	10mV/°C	−40°C to 125°C

- The LM35 series are precision integrated-circuit temperature devices with an output voltage linearly proportional to the Centigrade temperature.
- LM35 is three terminal linear temperature sensor from National semiconductors. It can measure temperature from -55 degree Celsius to +150 degree Celsius.
- **The voltage output of the LM35 increases 10mV per degree Celsius rise in temperature.**
- LM35 can be operated from a 5V supply and the stand by current is less than 60uA. The pin out of LM35 is shown in the figure below.
- **Features**
  - Calibrated Directly in Celsius (Centigrade)
  - Linear + 10mV/°C Scale Factor





A2D example

Temperature reading using LM35 temperature sensor

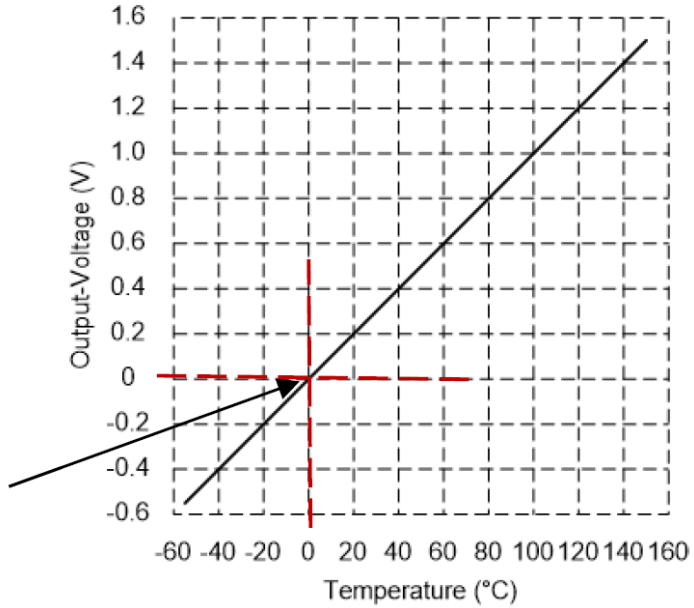
Converting the LM35 output voltage into temperature

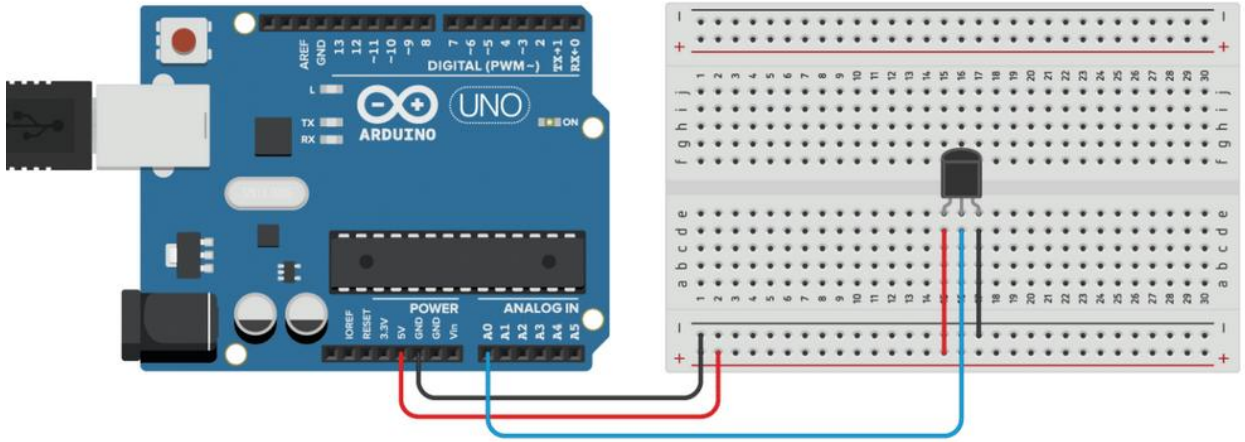
The output scale factor of the LM35 is 10 mV/°C and it provides an output voltage of 250 mV at 25°C

To convert the output voltage of the sensor into the temperature in degree Celsius, the following formula is used

Temperature (°C) = V_{OUT} / 10

with V_{OUT} in millivolt (mV).
So if the output of the sensor is 750 mV, the temperature is 75°C.







A2D example

Temperature reading using LM35 temperature sensor

- Arduino boards contain a multichannel, 10-bit analog to digital converter (ADC), which will map input voltages between 0 and the operating voltage (5 V or 3.3 V) into integer values between 0 and 1023. On an Arduino Uno, for example, this yields a resolution between readings of 5 volts / 1024 units or, 0.0049 volts (4.9 mV) per unit.
- So if you use **analogRead()** to read the voltage at one of the analog inputs of the Arduino, you will get a value between 0 and 1023.
- To convert this value back into the output voltage of the sensor, you can use:

$$\mathbf{V_{OUT} = reading\ from\ ADC * (Vref / 1024)}$$

- We will use these formulas in the code examples below.



A2D example

Temperature reading using LM35 temperature sensor

```
// Define to which pin of the Arduino the output of the LM35 is connected:
#define sensorPin A0

void setup() {
  // Begin serial communication at a baud rate of 9600:
  Serial.begin(9600);
}

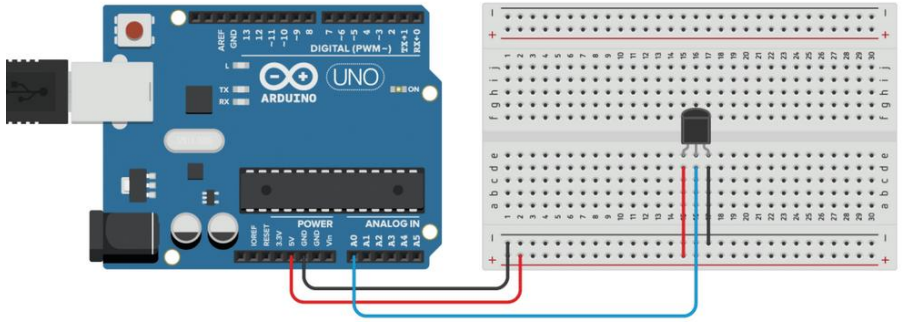
void loop() {
  // Get a reading from the temperature sensor:
  int reading = analogRead(sensorPin);

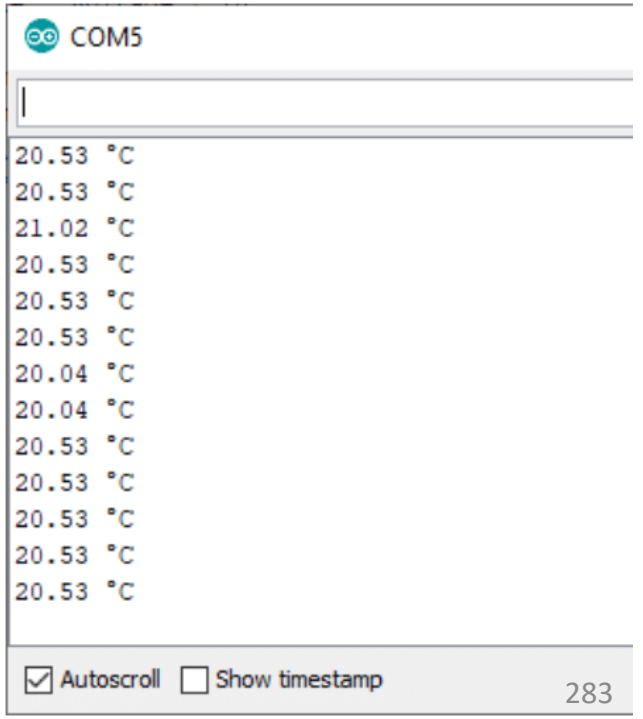
  // Convert the reading into voltage:
  float voltage = reading * (5000 / 1024.0);

  // Convert the voltage into the temperature in degree Celsius:
  float temperature = voltage / 10;

  // Print the temperature in the Serial Monitor:
  Serial.print(temperature);
  Serial.print(" \xC2\xB0"); // shows degree symbol
  Serial.println("C");

  delay(1000); // wait a second between readings
}
```







A2D example

Temperature reading using LM35 temperature sensor

- **Improve the precision of the readings**
- Because we used the default reference voltage of the Arduino for analog input (i.e. the value used as the top of the input range), the maximum resolution we get from the ADC is 5000/1024 = 4.88 mV or 0.49°C.
- If we want a higher precision, we can use the built-in 1.1 V reference from the Arduino instead.
- This reference voltage can be changed using the function analogReference().
- With 1.1 V as the reference voltage, we get a resolution of 1100/1024 = 1.07 mV or 0.11°C. Note that this limits the temperature range that we can measure to 0 to 110 degrees Celsius.



A2D example

Temperature reading using LM35 temperature sensor

```
// Define to which pin of the Arduino the output of the LM35 is connected:
#define sensorPin A0

void setup() {
  // Begin serial communication at a baud rate of 9600:
  Serial.begin(9600);

  // Set the reference voltage for analog input to the built-in 1.1 V reference:
  analogReference(INTERNAL);
}

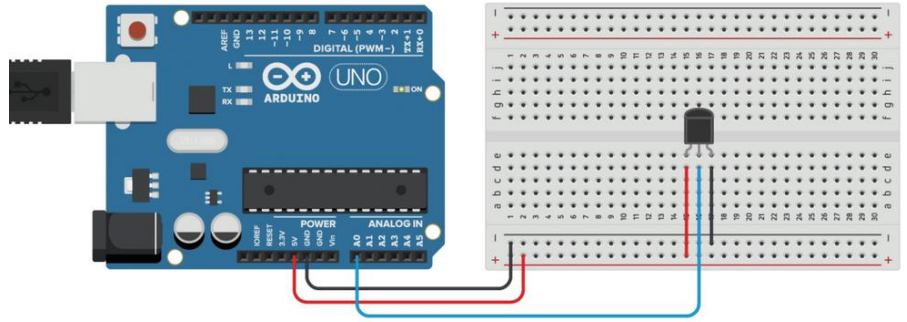
void loop() {
  // Get a reading from the temperature sensor:
  int reading = analogRead(sensorPin);

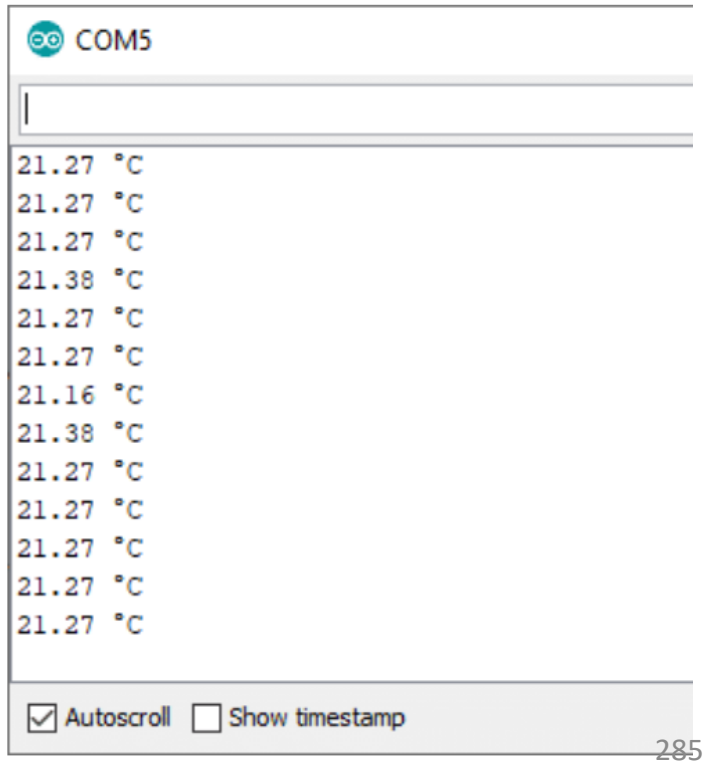
  // Convert the reading into voltage:
  float voltage = reading * (1100 / 1024.0);

  // Convert the voltage into the temperature in degree Celsius:
  float temperature = voltage / 10;

  // Print the temperature in the Serial Monitor:
  Serial.print(temperature);
  Serial.print(" \xC2\xB0"); // shows degree symbol
  Serial.println("C");

  delay(1000); // wait a second between readings
}
```







Week 8

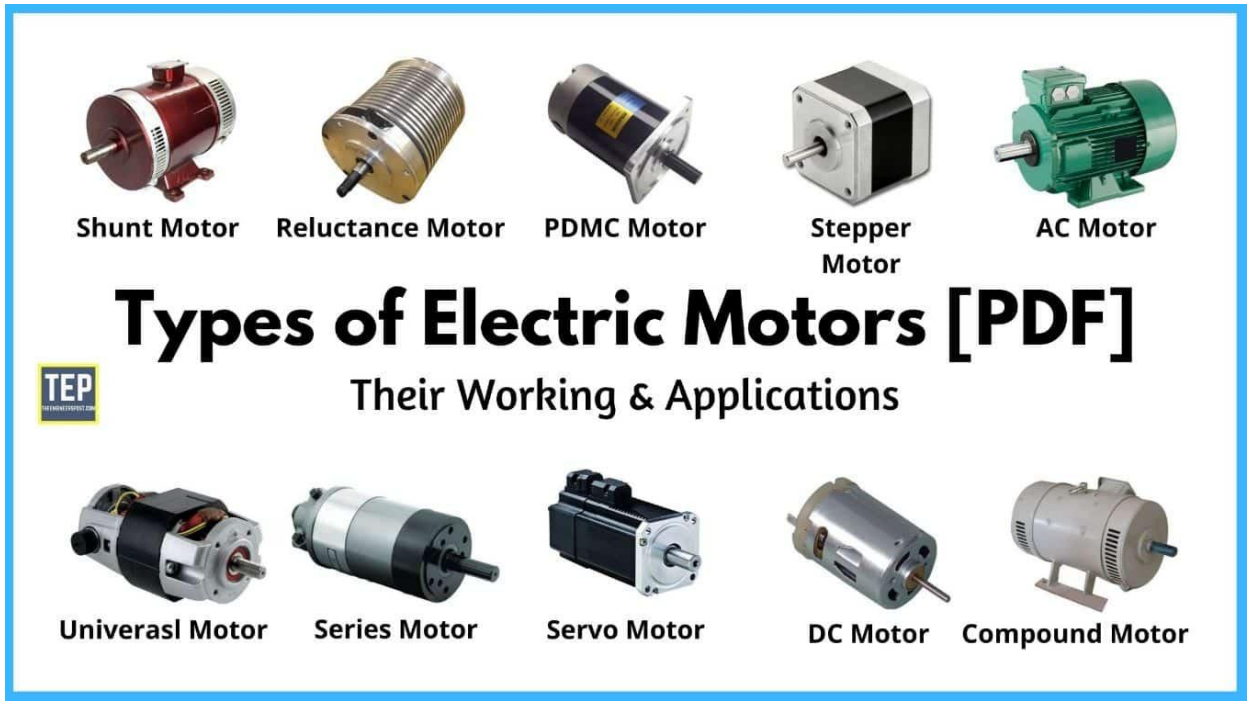
Lecture 8 / LO4

DC motors control



General overview on Motors

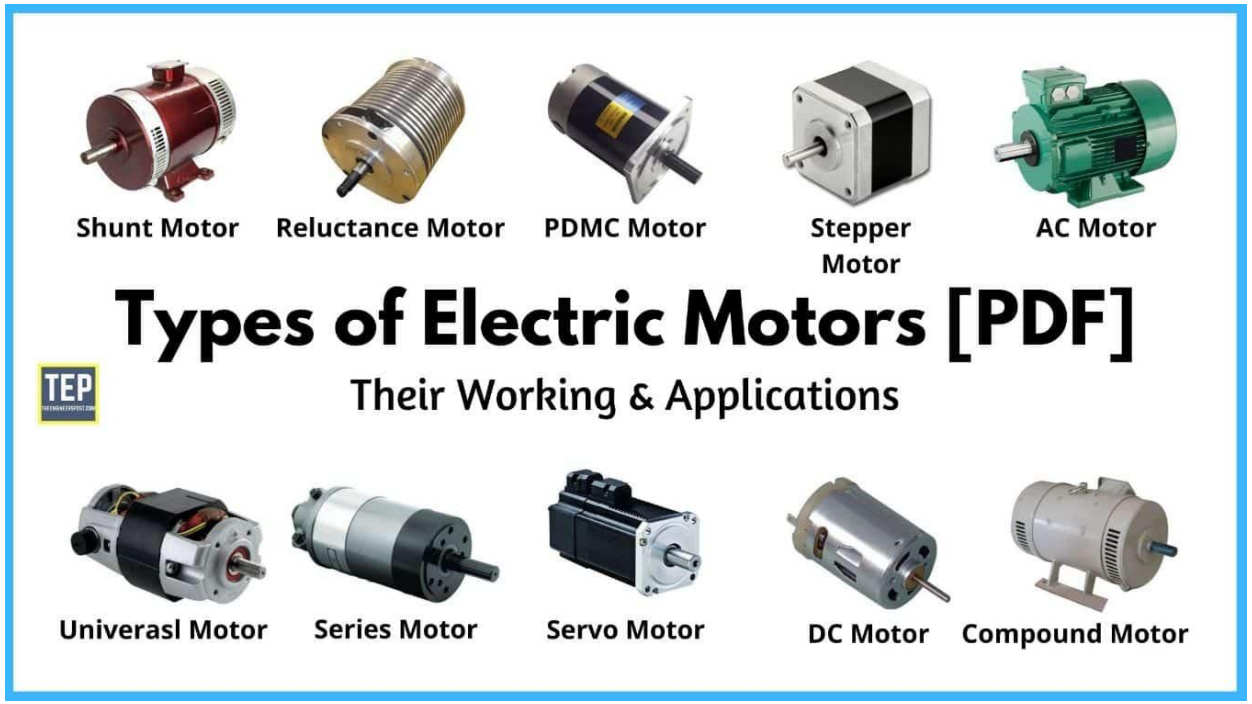
- The electric motor is defined as a machine that electrical energy into mechanical energy.
- Typically, motors are driven by the interaction between the magnetic field and electric current in a coil winding.
- Due to this, it generates force in the form of torque applied to the motor's shaft.





General overview on Motors

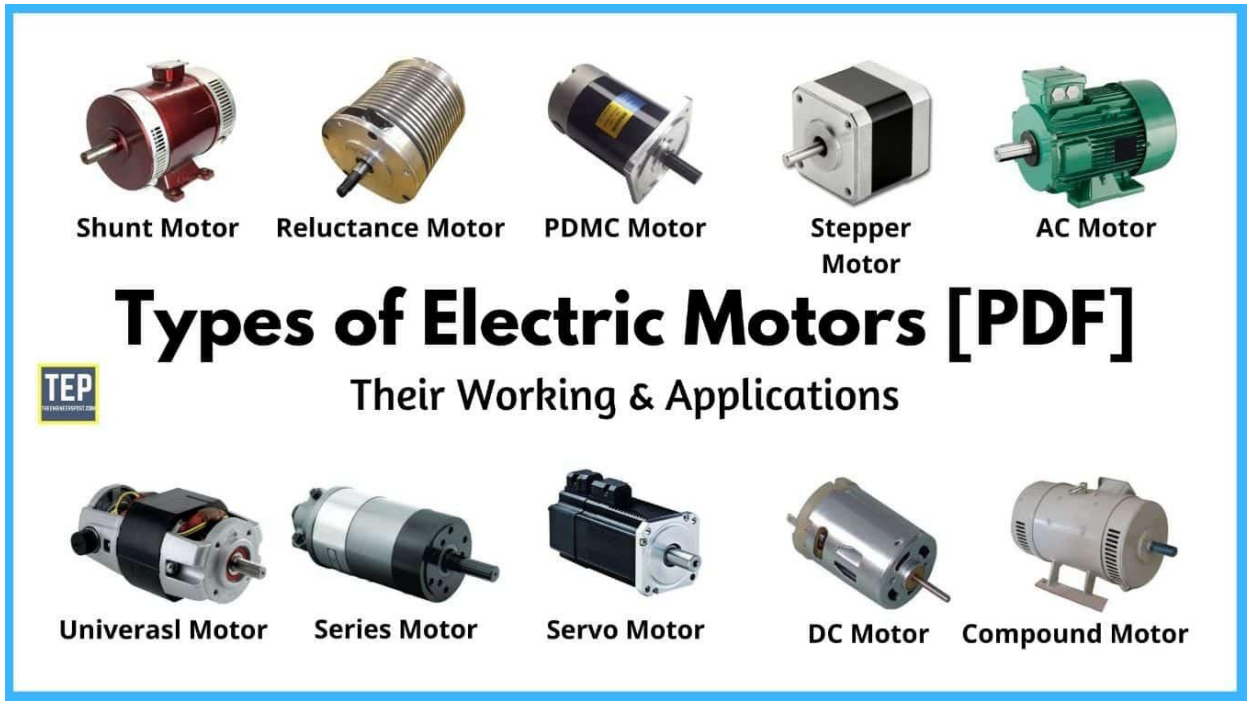
- Electric motors are basically powered by DC sources such as batteries, or rectifiers, and/or by AC sources such as power grids, inverters, or electric generators.
- The motors can be classified by considerations such as power source type, construction, application, and type of speed output.
- In this course, we will see how to drive a brushed DC motor, a stepper motor and a servo motor





General overview on Motors

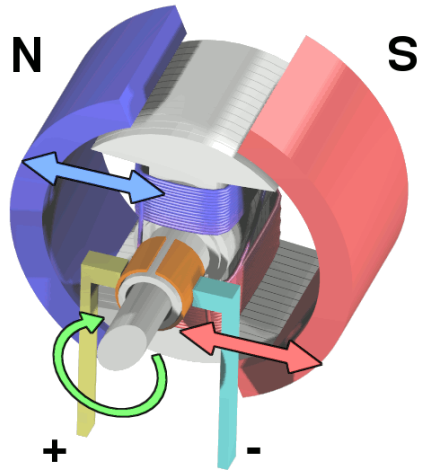
- All types of motors can be controller by any type of microcontrollers. The main difference is in the driving circuit and the components used to supply the motors.
- Arduino can drive all types of motors. All depends on the driving circuit (transistor/thyristor/triac circuits)





Brushed DC motor

- The design of the brushed DC motor is quite simple. A permanent magnetic field is created in the stator by either of two means:
  1. Permanent magnets
  2. Electro-magnetic windings
- If the field is created by permanent magnets, the motor is said to be a "permanent magnet DC motor" (PMDC).
- If created by electromagnetic windings, the motor is often said to be a "shunt wound DC motor" (SWDC).
- Today, because of cost-effectiveness and reliability, the PMDC motor is the motor of choice for applications power DC motors, as well as mos bou













Brushed DC motor driver design (in one direction)



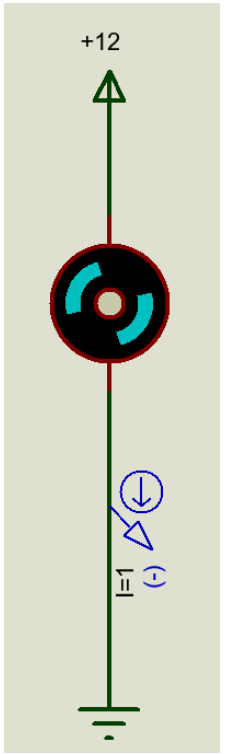
How to design an amplifier circuit to drive a motor

- Assume that we have a 12V DC motor with internal resistance of 12Ω.
- The current required by the motor will be

$$I_m = \frac{V_m}{R_m} = \frac{12}{12} = 1A$$

- This current cannot be supplied by the Arduino pin and thus a transistor amplifier circuit is required.

- The transistor will be used to amplify the Arduino current to be able to drive the motor.





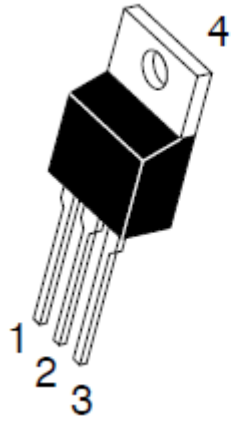
Brushed DC motor driver design (in one direction)

How to design an amplifier circuit to drive a motor

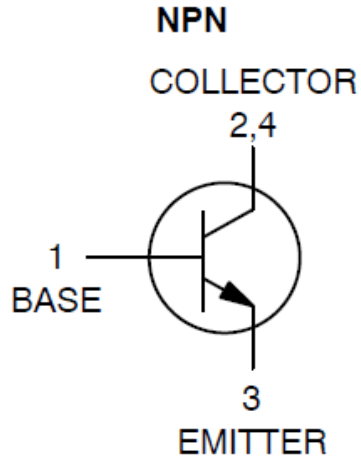
- The first thing to consider is the current of the transistor
- In this case, we are using the TIP120 (Darlington transistor) that has a collect current of 5A continuous

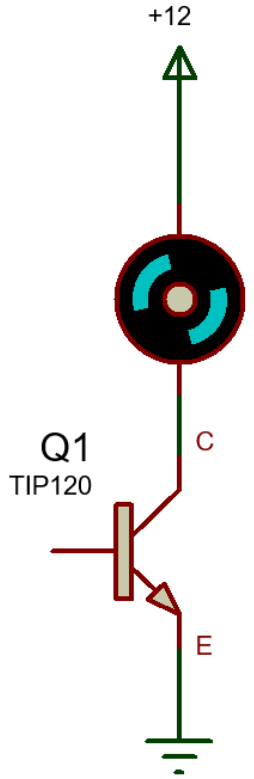
I_C	Collector Current (DC)	5	A
I_{CP}	Collector Current (Pulse)	8	A

h_{FE}	* DC Current Gain	V_{CE} = 3V,I_C = 0.5A V_{CE} = 3V, I_C = 3A	1000 1000



TO–220
CASE 221A
STYLE 1







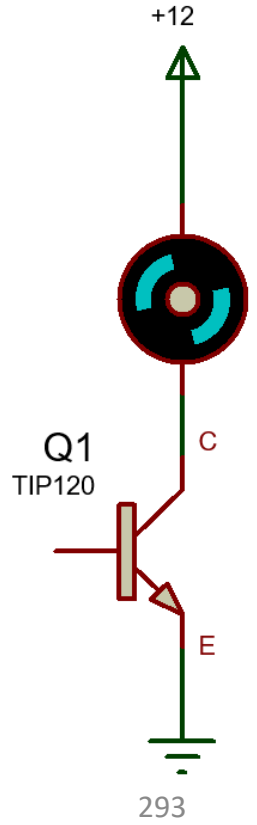
Brushed DC motor driver design (in one direction)

How to design an amplifier circuit to drive a motor

- The saturation current required is the motor current I_m which is the collector current I_C
- Knowing I_C, we can calculate the base current I_B using

$$I_B = \frac{I_C}{\beta} = \frac{1}{1000} = 0.001A = 1mA$$

- With *beta* is the grain in current of the transistor (also called *hef*)
- NOTE: the base current will be delivered from the Arduino and should be less that 20mA (as recommended by the ATMEGA328 Datasheet)

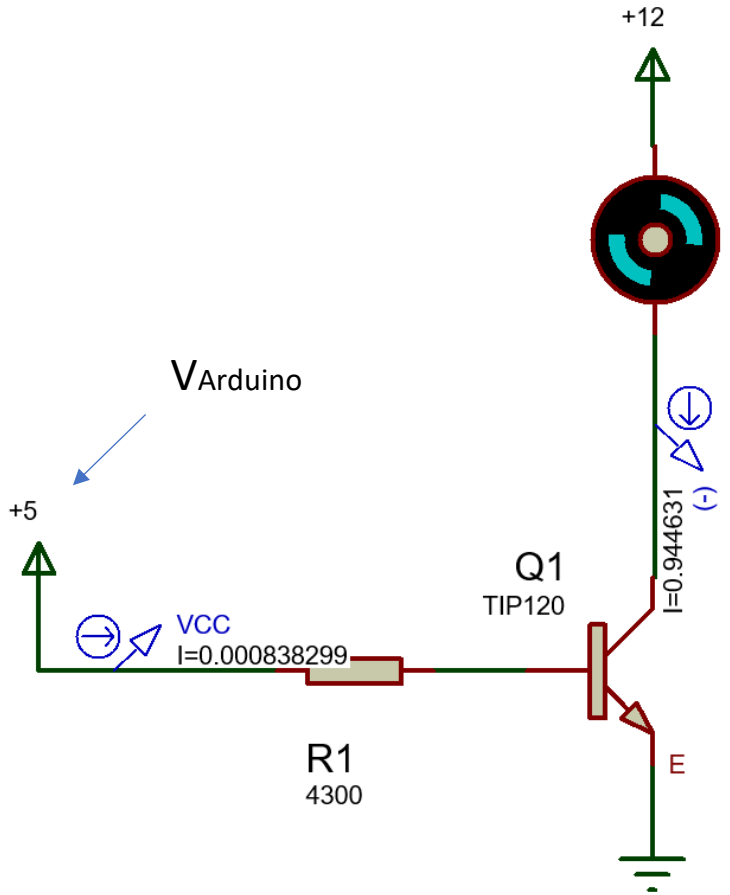




Brushed DC motor driver design (in one direction)

How to design an amplifier circuit to drive a motor

- Knowing IB, we need now to find the series resistor between the Arduino pin and the base of the transistor. This is required to drop the Arduino voltage (5 volt) to the VBE of the transistor which is around 0.7 volt (voltage across the BE pins which is a PN junction)



- The value of the resistor can be calculated using KVL and Ohm's law as:

$$R = \frac{V_{Arduino} - V_{BE}}{I_B} = \frac{5 - 0.7}{0.001} = 4300\Omega = 4.3k\Omega$$



Brushed DC motor driver design (in one direction)

How to design an amplifier circuit to drive a motor

- After making sure that the current delivered from the Arduino will not exceed the limit, it is safe to connect the motor circuit to the Arduino pin. The motor can be controlled by:

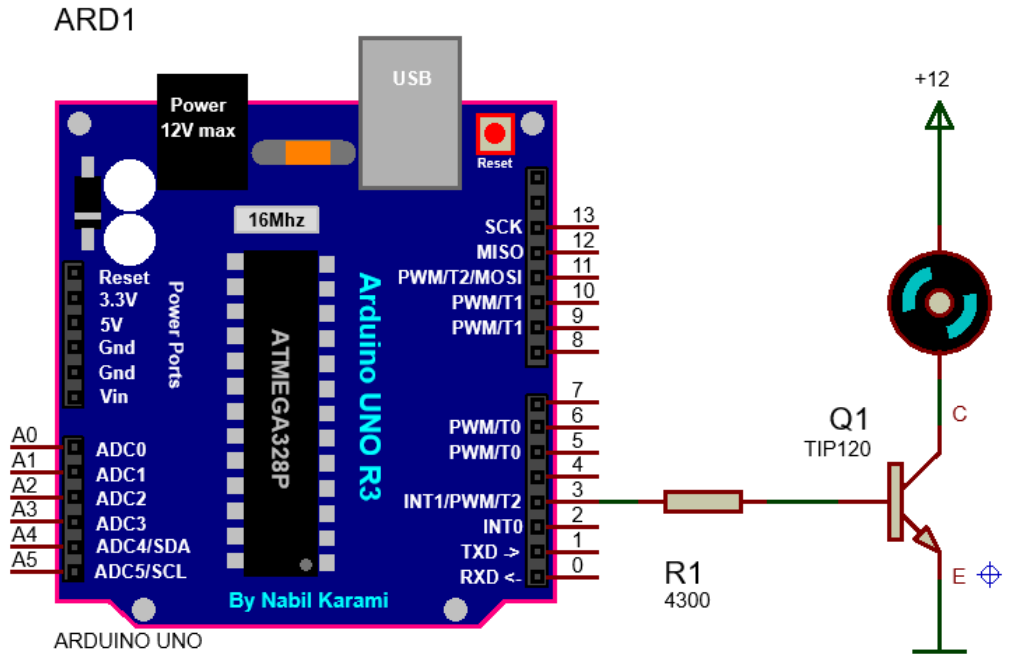
- digitalWrite(8, HIGH) to turn ON the motor

Or

- digitalWrite(8, LOW) to turn OFF the motor

Or

Using analogWrite instruction that will be treated later





Brushed DC motor driver design (in one direction)

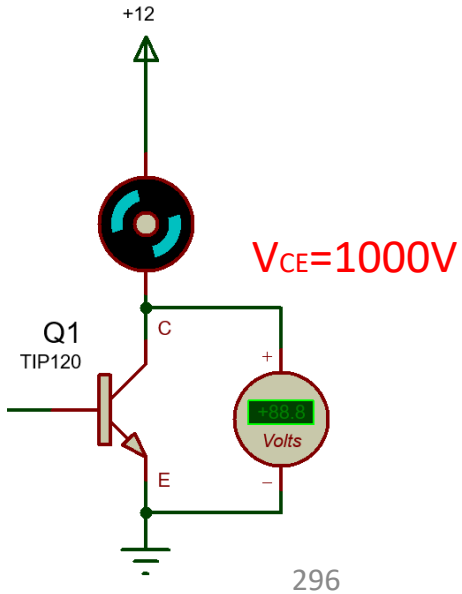
How to design an amplifier circuit to drive a motor

- The circuit need enhancement to protect the transistor for the Inductor current of the motor.

- Assume that the motor is normally running and a 1A is flowing through the transistor.
- When switching it off by sending the digitalWrite(8,LOW) instruction, the current should be zero (logical analysis)
- But the fast switching will generate a high voltage from the motor that can be modeled as:

- $V_M = V_L = L\frac{di_M}{dt} = L\frac{\Delta i}{\Delta t}$
- Taking some numerical values for L=1mH , ΔiM=1A, and Δt=1µs

- $V_M = V_L = 1000V$ which represents the voltage across the CE pins of the transistor.
- The VCE voltage of the transistor will be given in the datasheet



V_{CEO}	Collector-Emitter Voltage : TIP120	60	V



Brushed DC motor driver design (in one direction)

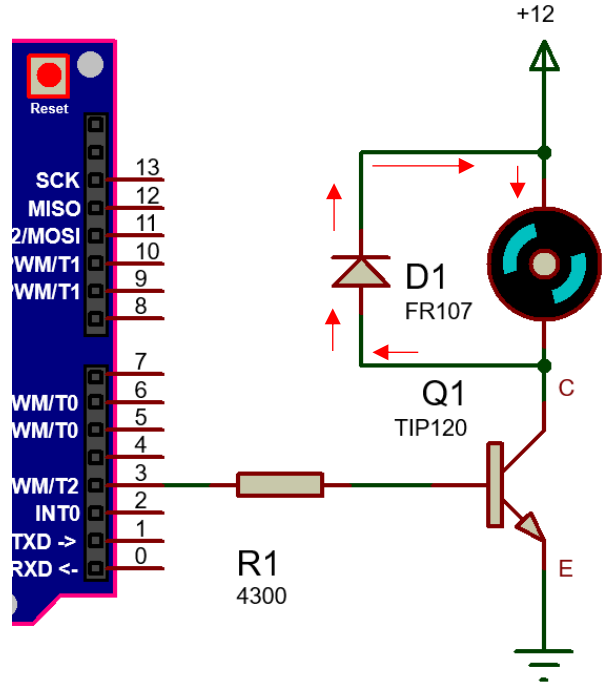
How to design an amplifier circuit to drive a motor

- Thus it is required to find solution for the high voltage generated by the inductor of the motor when switched off.
- It can be seen from the equation $V_M = V_L = L\frac{di_M}{dt} = L\frac{\Delta i}{\Delta t}$ that the only parameter that can be controlled is time

- Thus increasing Δt will decrease the voltage VL
- This can be done by using a fast diode (fast recovery diode)
- The role of the diode is to open a path for the current to circulate inside the Motor winding (made of inductor and resistance)

- The current will drop slowly until the energy is fully dissipated inside the resistance of the motor

The diode here is called free-wheeling diode





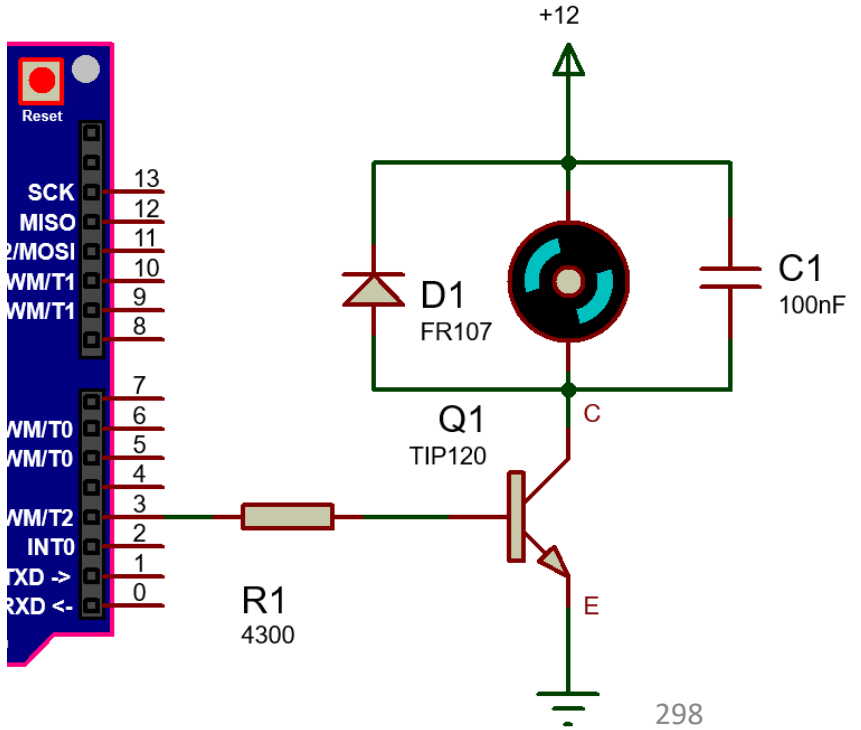
Brushed DC motor driver design (in one direction)

How to design an amplifier circuit to drive a motor

- Further enhancement is required to remove the RF noise of the motor



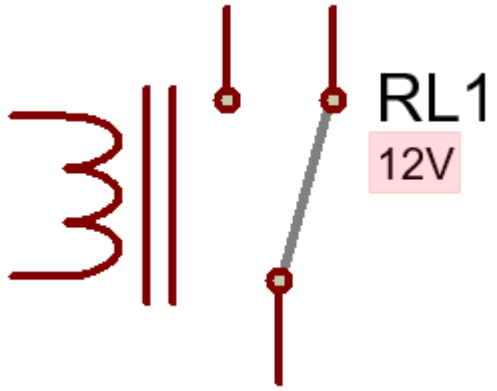
- When motors with brushes are running normally, the motor brushes produce sparks, which cause noise.
- It happens even when these motors are connected directly across a battery.
- If we did nothing, the cable running from the electronics board (or directly from the battery) to the motor would act like an antenna, radiating TV and other radio interference.
- One way people fix that problem is to attach small ceramic capacitors directly to the motor to absorb much of that noise





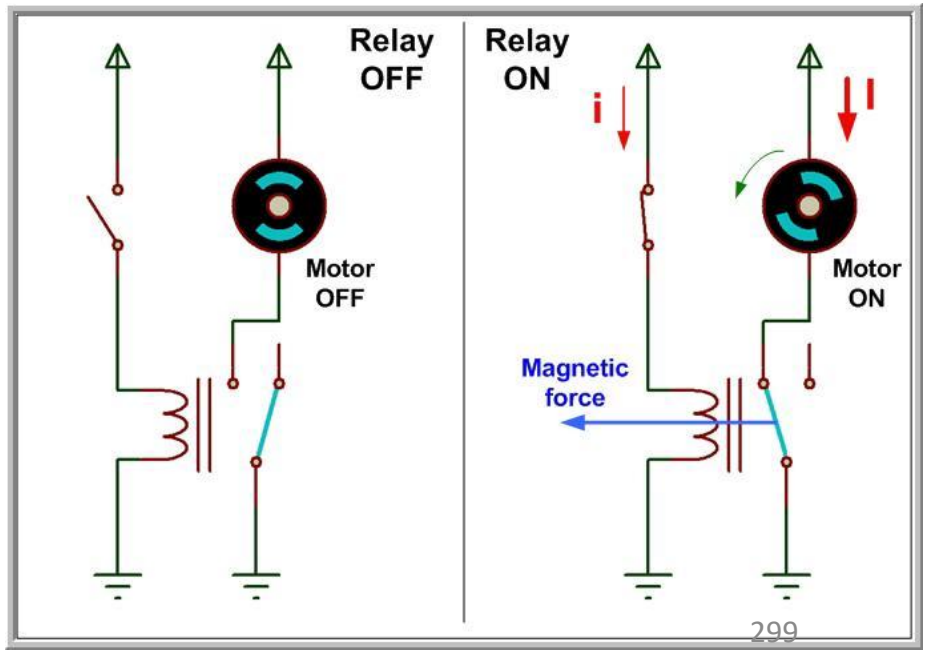
Brushed DC motor driver design (in one direction through relay)

- A relay is an electrical switch that opens and closes under the control of another electrical circuit.
- In the original form, the switch is operated by an electromagnet to open or close one or many sets of contacts.



- When a current flows through the coil, the resulting magnetic field attracts an armature that is mechanically linked to a moving contact.
- The movement either makes or breaks a connection with a fixed contact.
- When the coil current is switched off, the armature is returned by a force approximately half as strong as the magnetic force to its relaxed position.





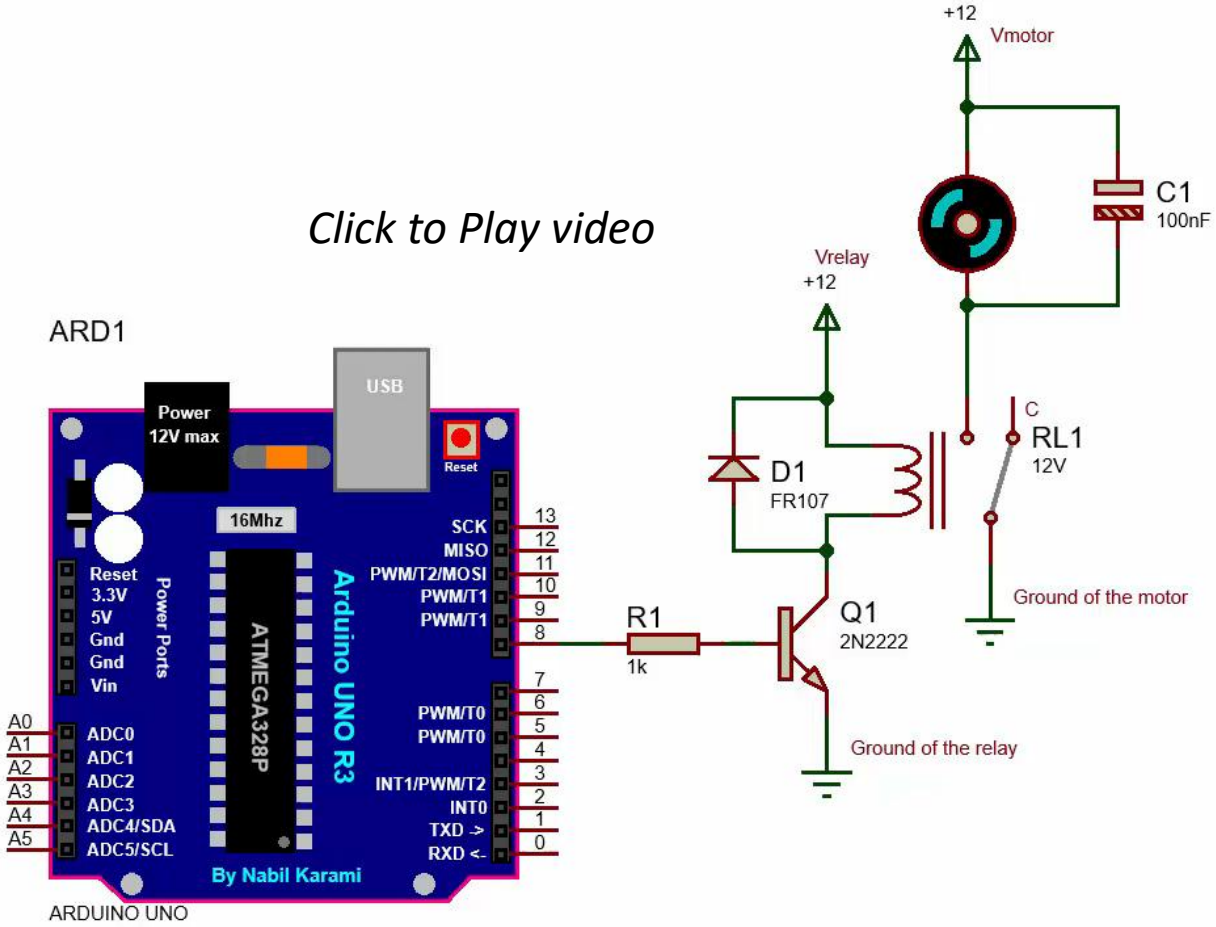


Brushed DC motor driver design (in one direction through relay)

How to drive Motor through relay

- The relay is used if we want to drive a high power motor without using a power transistor circuits.
- The relay is driven using a signal transistor since its coil requires around 30 mA and 100 mA depending on its size and manufacturer.

Click to Play video

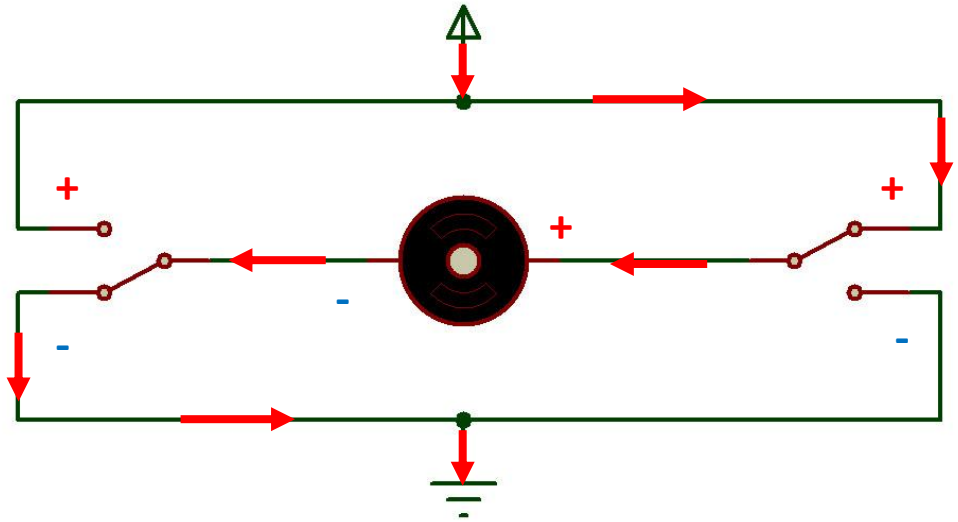




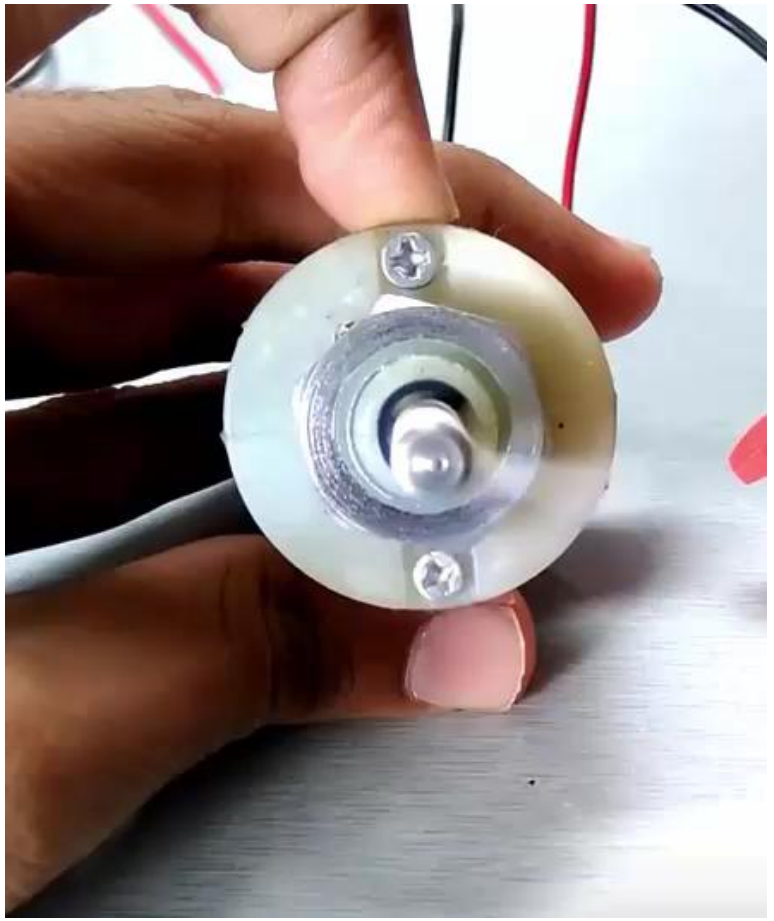
Brushed DC motor driver design (in all directions through relays)

DC motor changes direction when the polarity of the power supply changes.

We can flip the polarity of each electrode using two switches. Thus the motor can turn left, right or stop



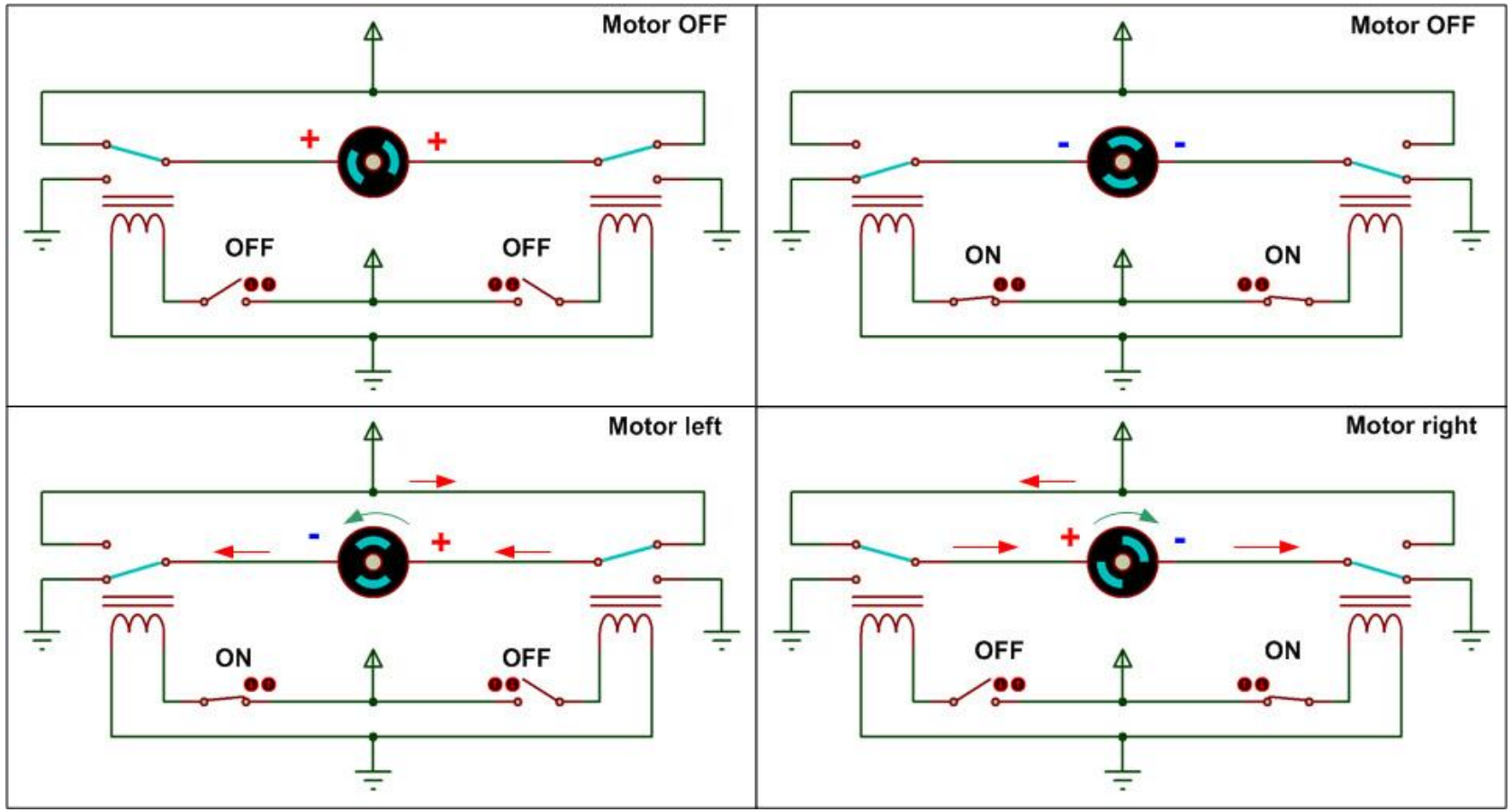
Click to Play video





Brushed DC motor driver design (in all directions through relays)

Switches now can be replaced by relays; motors can be driven left, right or stop based on the relay states

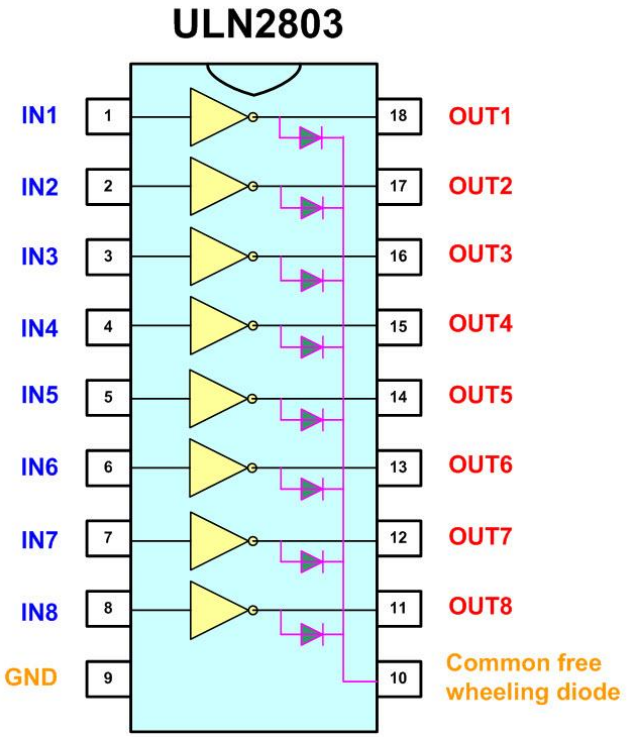


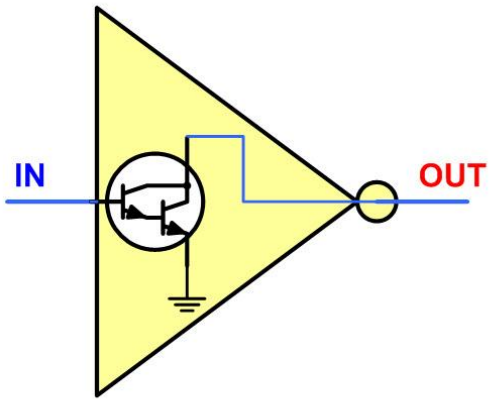


Brushed DC motor driver design (in all directions through relays)

- Relays can be driven by transistors. Here we are using a transistor array chip called ULN2803.
- The main objective of this application is to present the ULN2803 chip that contains eight Darlington transistors
- The ULN2803 also contains eight free wheeling diodes connected to Darlington transistors and connected together to the common pin (pin 10).
- All transistor bases are protected by internal resistors; therefore, you will not need any external components to protect it.









Brushed DC motor driver design (in all directions through relays)

```
#define relay1 8
#define relay2 9

void setup() {
  pinMode(relay1, OUTPUT);
  pinMode(relay2, OUTPUT);
}

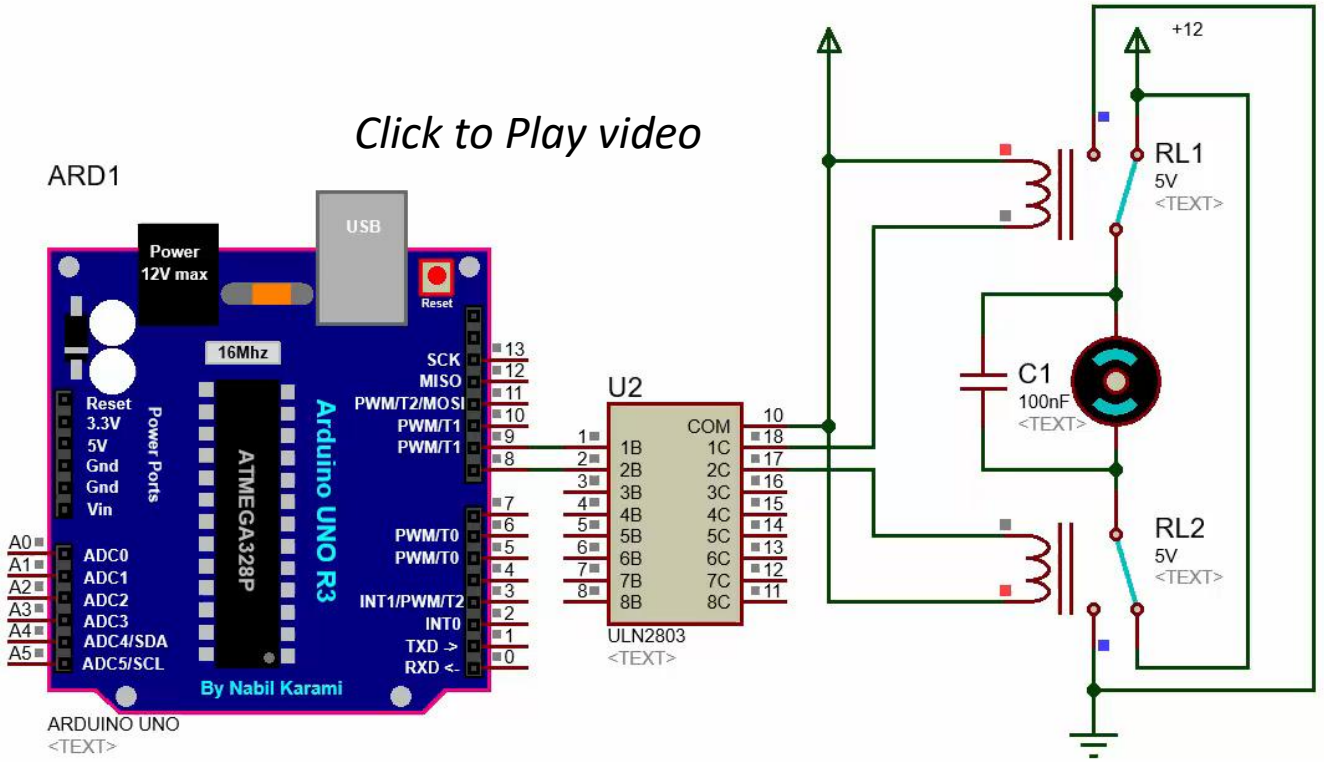
void Motor_left() {
  digitalWrite(relay1, HIGH);
  digitalWrite(relay2, LOW);
}

void Motor_right() {
  digitalWrite(relay1, LOW);
  digitalWrite(relay2, HIGH);
}

void Motor_stop() {
  digitalWrite(relay1, LOW);
  digitalWrite(relay2, LOW);
}

void loop() {
  Motor_left();
  delay(1000);
  Motor_stop();
  delay(1000);
  Motor_right();
  delay(1000);
  Motor_stop();
  delay(1000);
}
```

Click to Play video





Week 9

Lab 7 / LO5

Driving DC motors using relays



Tools and components

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. Breadboard wires (male-male)
  4. Power supply 12 volt

- Required components:
  1. Arduino UNO + USB cable
  2. Potentiometer 1k to 10k ohm x1
  3. Push button x3
  4. ULN2803 or ULN2003 x1
  5. Small DC motor
  6. Relay 5 or 12V x2



Manual control of DC motor

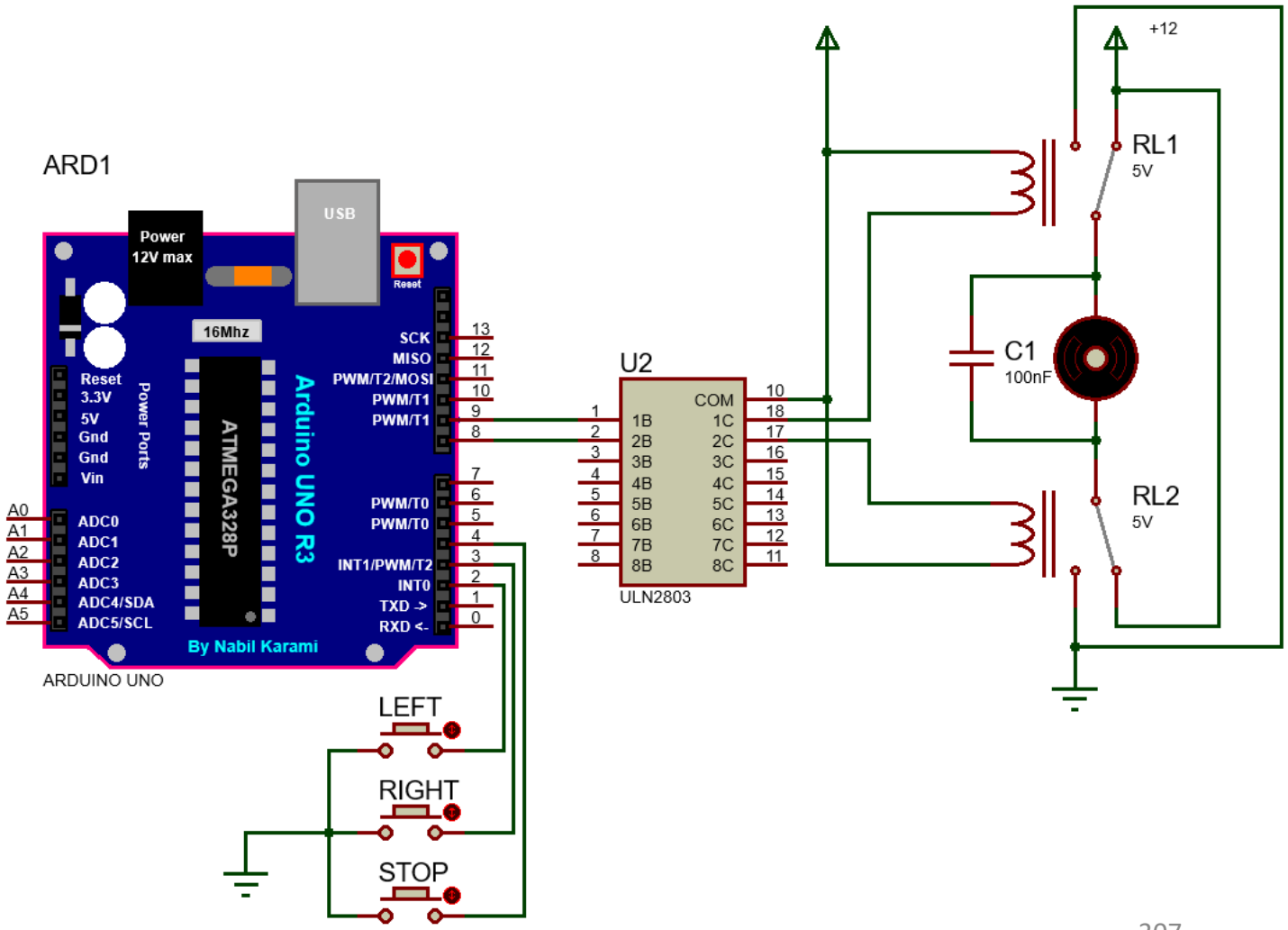
```
#define relay1 8
#define relay2 9
#define pb_left 2
#define pb_right 3
#define pb_stop 4
void setup() {
  pinMode(relay1, OUTPUT);
  pinMode(relay2, OUTPUT);
  pinMode(pb_left, INPUT_PULLUP);
  pinMode(pb_right, INPUT_PULLUP);
  pinMode(pb_stop, INPUT_PULLUP);
}

void Motor_left() {
  digitalWrite(relay1, HIGH);
  digitalWrite(relay2, LOW);
}

void Motor_right() {
  digitalWrite(relay1, LOW);
  digitalWrite(relay2, HIGH);
}

void Motor_stop() {
  digitalWrite(relay1, LOW);
  digitalWrite(relay2, LOW);
}

void loop() {
  if (!digitalRead(pb_left)) Motor_left();
  if (!digitalRead(pb_right)) Motor_right();
  if (!digitalRead(pb_stop)) Motor_stop();
}
```





Week 10

Lecture

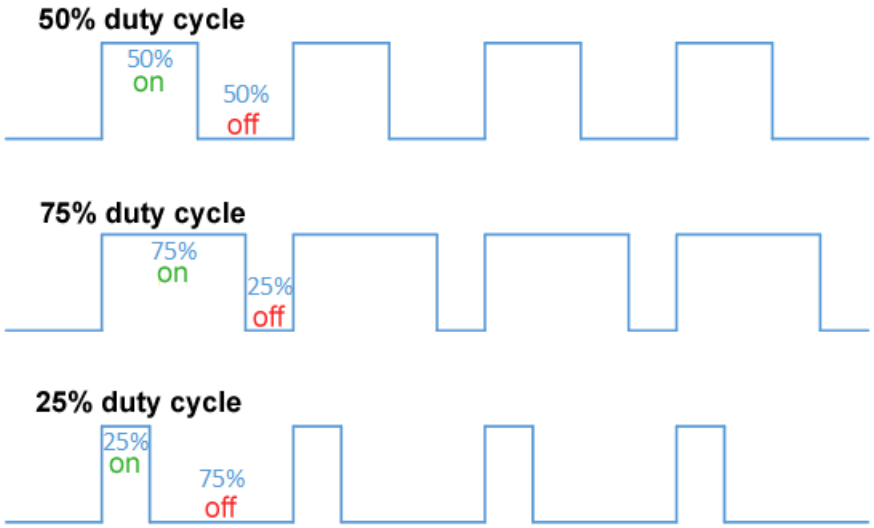
Lecture 9 / LO5

PWM



Brushed DC motor driver design (in one direction with different speed)

- To change the speed of the motor we have to change the voltage applied to its armature
- The Arduino or any microcontroller cannot reduce the pin voltage as the pins are digital (0 or 5 volts; sometimes 0 or 3.3V depends on the chip voltage)
- The trick to change the motor voltage is to turn ON and OFF the output pin at different timing.
- Since the motor is made of an inductor, the average voltage will be a function of the ON time over the period of the signal.
- This signal is called a PWM signal (Pulse Width Modulation signal)



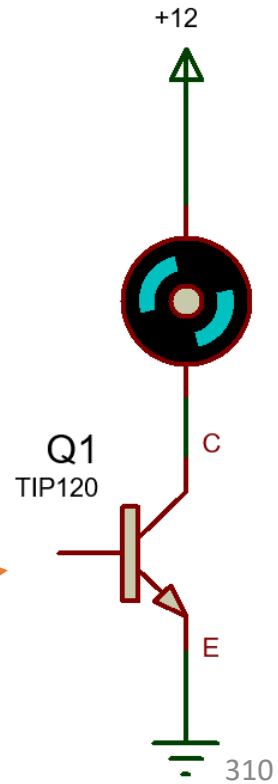


Brushed DC motor driver design (in one direction with different speed)

- If, for example, we apply the PWM signal on the base of the transistor, the motor will turn ON and OFF, but due to the high switching frequency and the motor inductor, the average voltage is reduced and the speed is reduced as well.

- For example, if the motor is turned ON for 1 second and OFF for 1 second, the motor will see the half of the applied voltage (which is 6 volt in this case) which corresponds to half speed (we are assuming a linear relationship between the voltage and the speed for the moment)





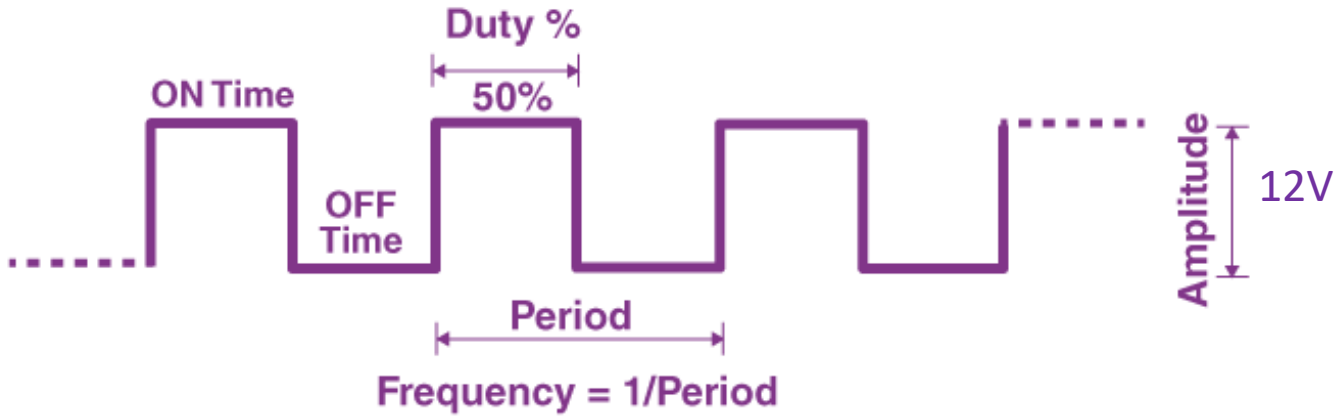


Brushed DC motor driver design (in one direction with different speed)

- Duty cycle: Per definition, the Duty cycle is the ratio of the ON time Over the complete Period.

$$Duty\ Cycle = \alpha = \frac{T_{ON}}{T}$$

It is clear here that the motor will be supplied by 12V during TON and OFF during TOFF





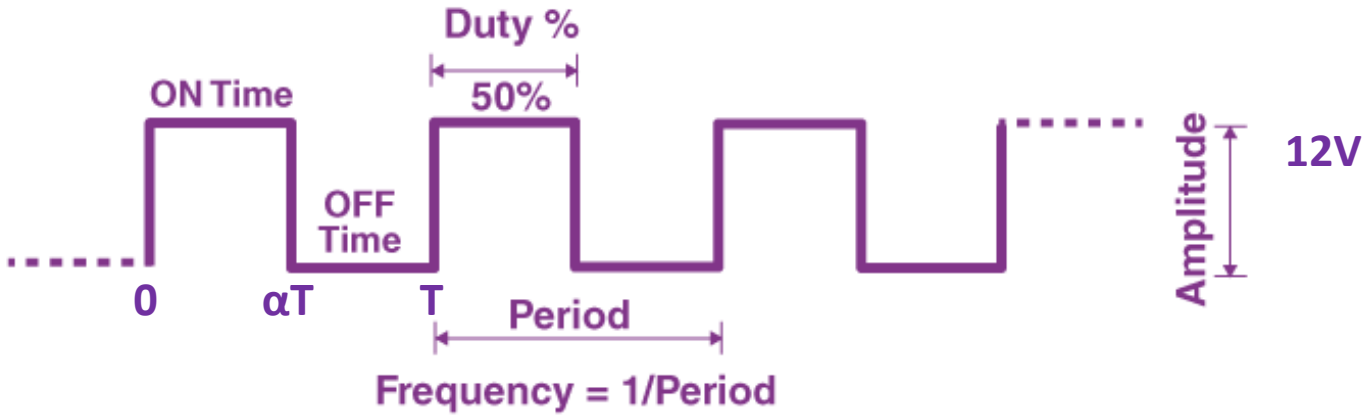
Brushed DC motor driver design (in one direction with different speed)

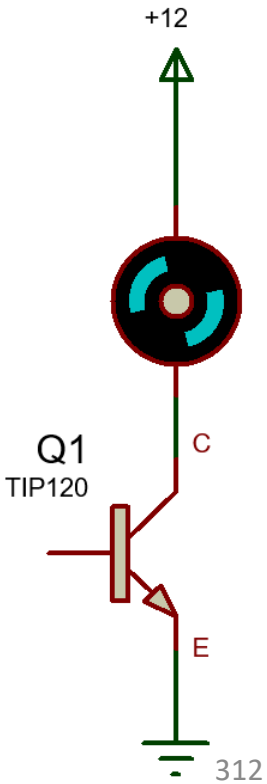
- The Average voltage seen by the motor can be calculated by finding the energy of the signal for a complete period. This can be done by using the integration of the area under the signal

$$V_{avg} = V_{motor} = \frac{1}{T}\int_0^T V_{in}dt = \frac{1}{T}\left[\int_0^{\alpha T} V_{in}dt + \int_{\alpha T}^{T} V_{in}dt\right] = \alpha V_{in}$$

The equation means that the motor voltage will be part of the input voltage as α is between 0 and 1

So as a conclusion, the motor voltage will be: $V_{avg} = V_{motor} = \alpha V_{in} = \frac{T_{ON}}{T} V_{in}$



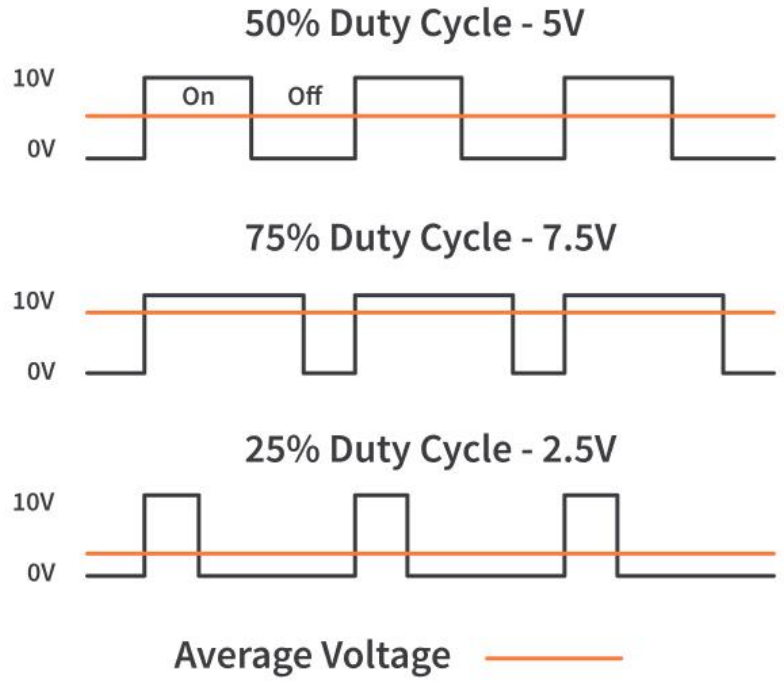




Brushed DC motor driver design (in one direction with different speed)

For example, in case a motor is supplied by 10 volts, the motor will see

- 5 volts because α = 50% (T_{ON} = T/2)
  (Almost half speed)
- 7.5 volts because α = 75% (T_{ON} = 0.75T)
  (Almost 75% of the max speed)
- 2.5 volts because α = 25% (T_{ON} = 0.25T)
  (Almost 25% of the max speed)





Brushed DC motor driver design (in one direction with different speed)

- An important thing to note is that the speed and intensity at which a brushed DC motor spins is directly related to the duty cycle and voltage at VM and not to the frequency in which the PWM switches from high to low.

- There are however considerations to be had when choosing an ideal PWM switching frequency. These are stability, switching losses, and audible noise.

- If the switching frequency is too low, the motor will experience a voltage ripple.
- As a rule of thumb, you want the frequency of your PWM signal to be greater than 5 over $2\pi\tau$, where τ is the electrical time constant for your desired motor.
- If you go to high, however, the transistors inside of your driver may experience thermal limitations due to switching losses.
- Finally, if audible noise is a concern, it would be a good idea to stay above the audible human range. We'd end for most at 20 kilohertz.



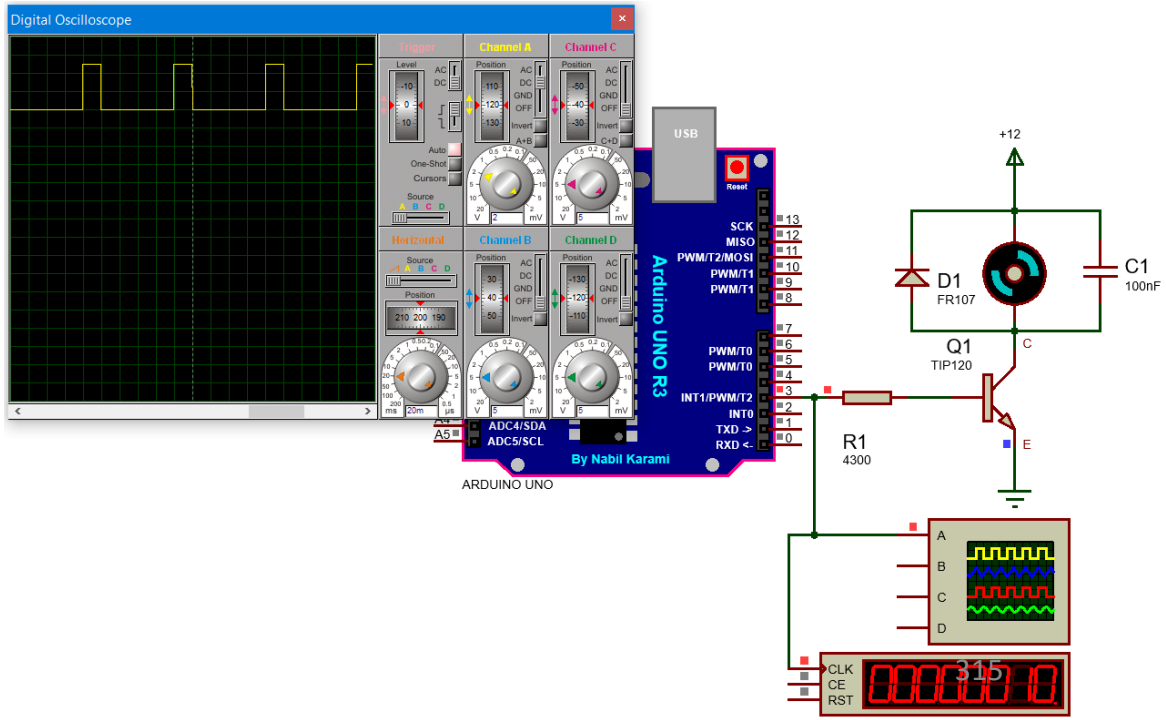
Brushed DC motor driver design (in one direction with different speed)

- The Arduino is able to control the motor speed in hardware and software; let's start by the software solution (the bad and cumbersome solution)

- This method is about blinking the motor at 100ms period. The ON time is 20ms and the OFF time is 80ms
- The drawback of this code is that it is not allowed to do any other task in the code otherwise the time will be stretched

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

void loop() {
  digitalWrite(motor, HIGH);
  delay(20);
  digitalWrite(motor, LOW);
  delay(80);
}
```



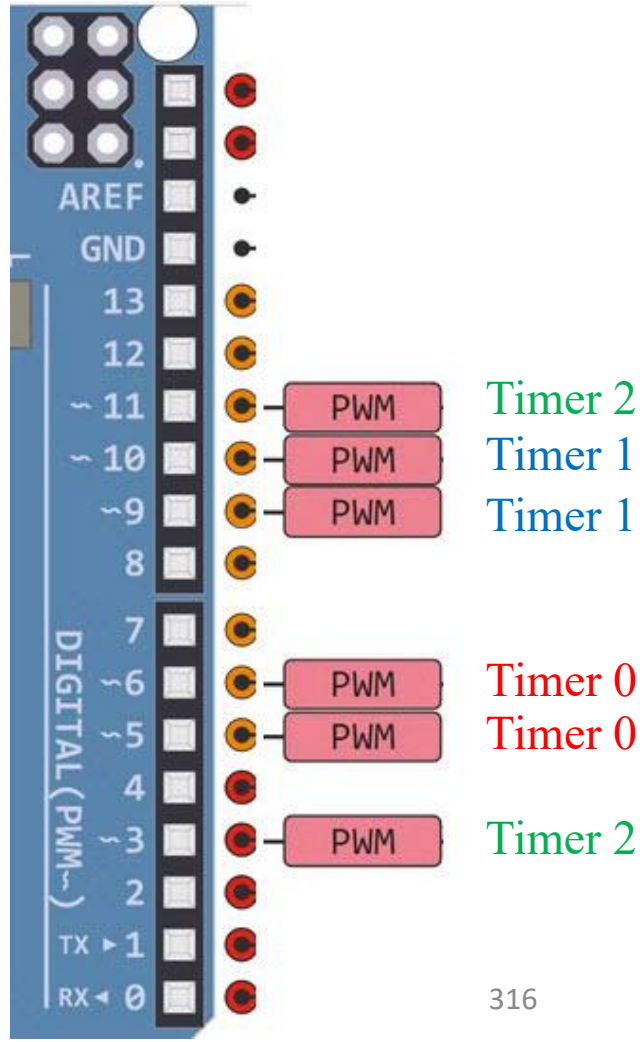


Brushed DC motor driver design (in one direction with different speed)

- The smartest method is to use the hardware of the Arduino in which a PWM signal generator block is already available to generate signals on 6 different channels and at the same time

- The Arduino's programming language makes PWM easy to use; simply call analogWrite(pin, VALUE), where VALUE is from 0 to 255, and pin is one of the PWM pins (3, 5, 6, 9, 10, or 11).
- The analogWrite() function provides a simple interface to the hardware PWM, but doesn't provide any control over frequency. (Note that despite the function name, the output is a digital signal.)

  •Pins 5 and 6: controlled by Timer 0 (Default frequency is 976.56 Hz)
  •Pins 9 and 10: controlled by timer 1 (Default frequency is 490.20 Hz)
  •Pins 11 and 3: controlled by timer 2 (Default frequency is 490.20 Hz)





Brushed DC motor driver design (in one direction with different speed)

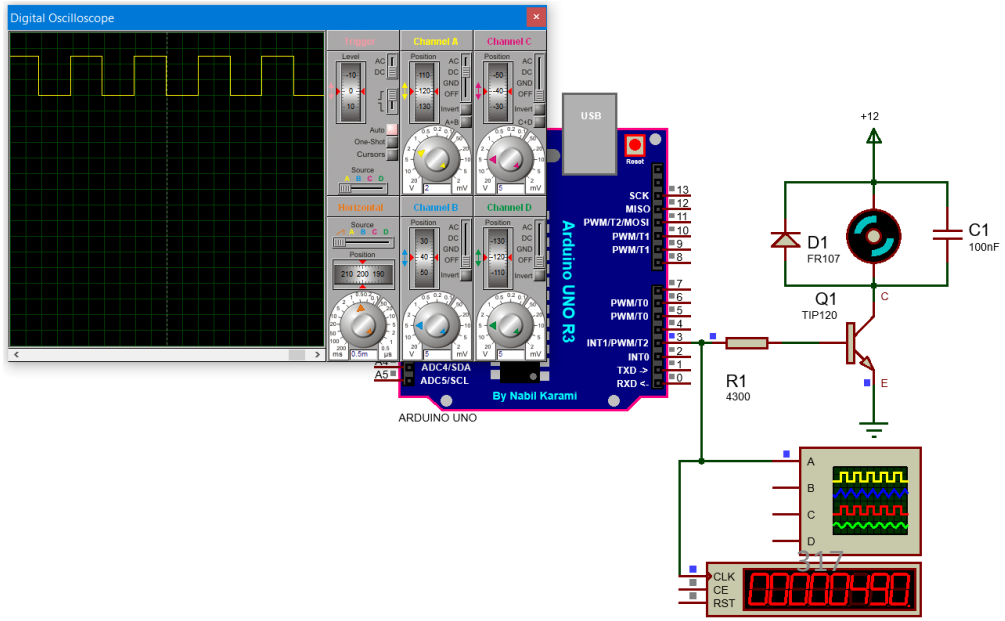
The hardware solution is very easy to implement. It is all about the instruction analogWrite(pin, VALUE)

- The VALUE here is an 8bit, i.e. 256 different values (from 0 to 255)
- Means that we have 256 different speed of the motor (0 = OFF (0% speed) and 255 = Full speed (100% speed)
- Thus the motor average voltage is calculated as

$$V_M = \frac{\text{VALUE}}{255} \times V_{in} = \frac{127}{255} \times 12 = 6V$$

```
1 #define motor 3
2
3 void setup() {
4   pinMode(motor, OUTPUT);
5   analogWrite(motor, 127);
6 }
7
8 void loop() {
9 }
```

127 out of 255
Almost 50%

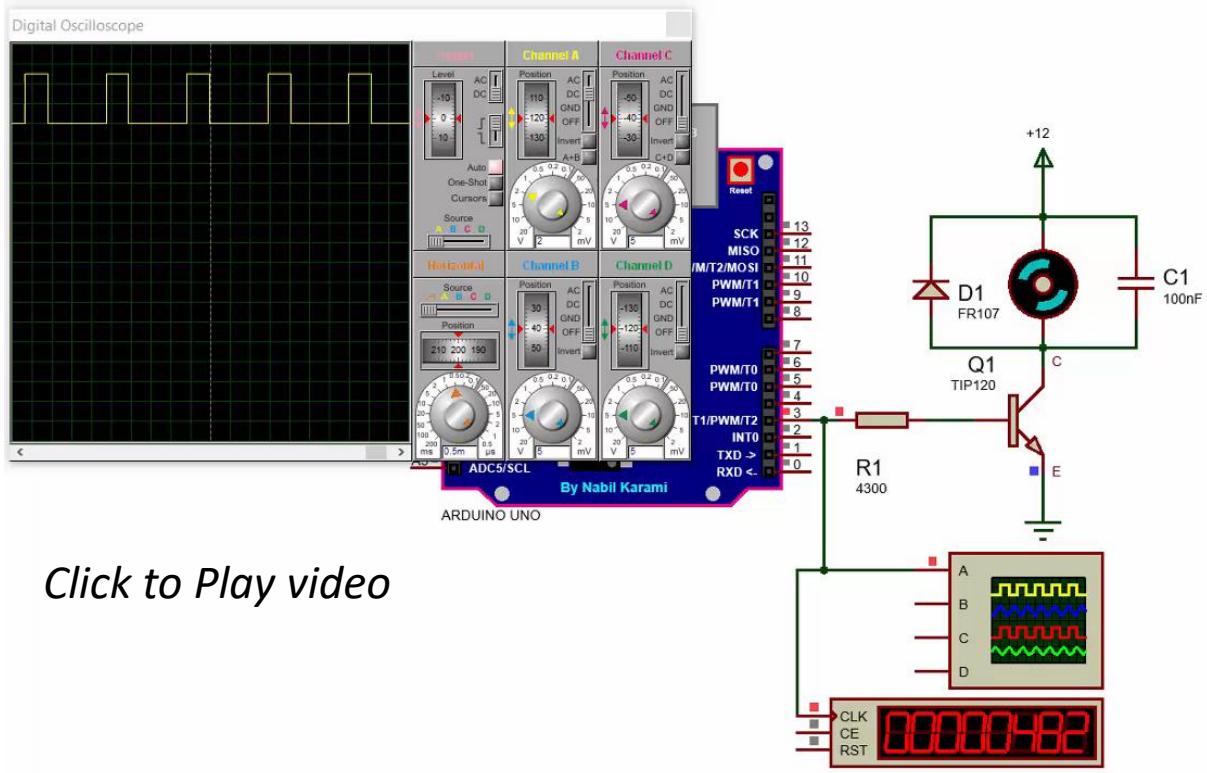




Brushed DC motor driver design (in one direction with different speed)

In this demo, we are incrementing the duty cycle from 0 to 100% by incrementing the VALUE from 0 to 255
The speed of the motor increases from o to 100%

```
1  #define motor 3
2  void setup() {
3    pinMode(motor, OUTPUT);
4  }
5
6  void loop() {
7    for (int i = 0; i < 256; i++) {
8      analogWrite(motor, i);
9      delay(10);
10   }
11 }
```

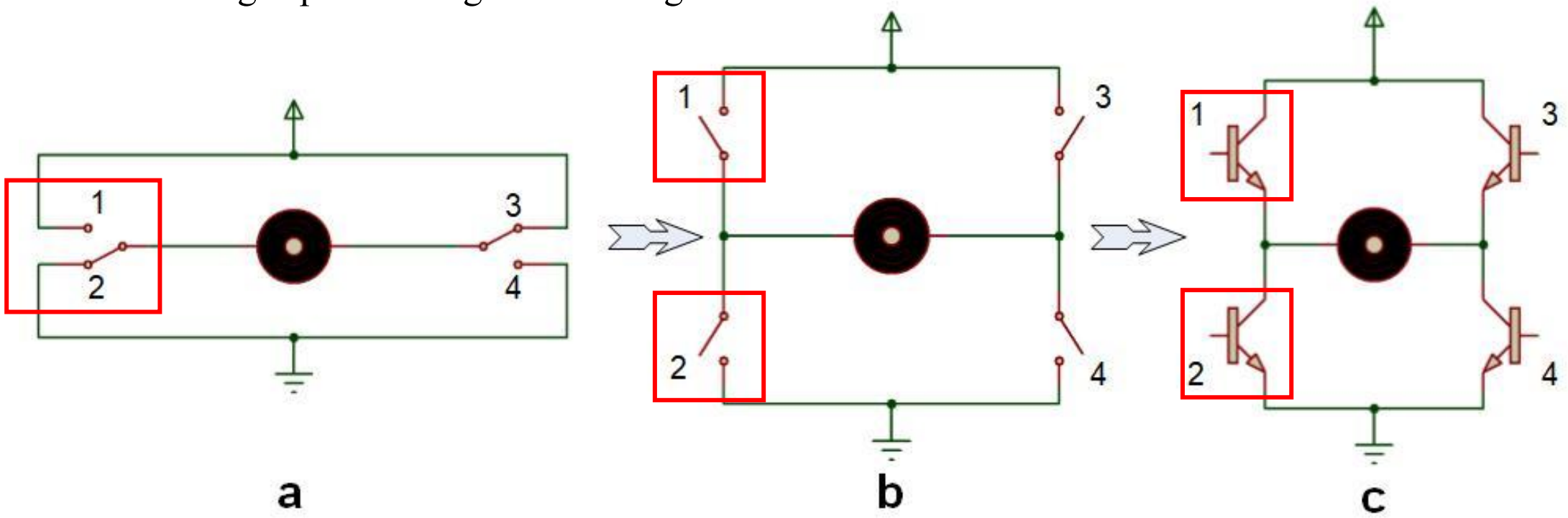


Click to Play video



Brushed DC motor driver design (in different directions and speed)

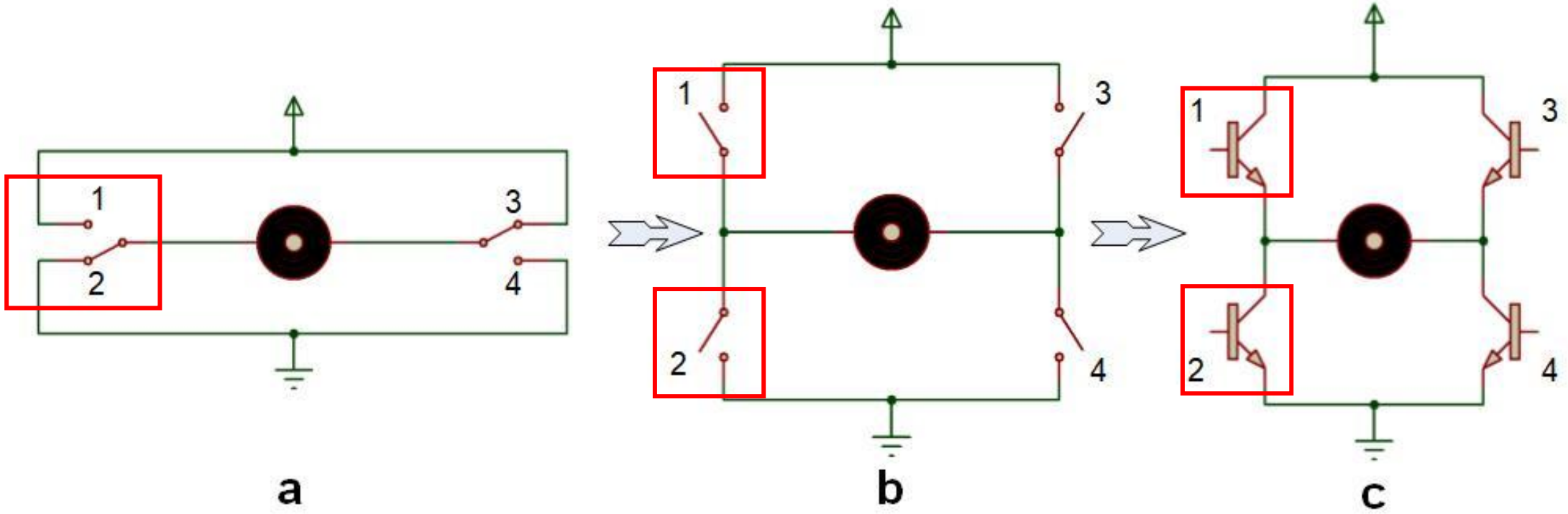
- In this application, the DC motor is driven in double directions with different speed.
- Different speed means a PWM signal and different direction means relay that switch polarity on the motor armature.
- In fact, when merging these two techniques together we can reach what is called the Hbridge.
- We will be using 4 transistor to design an H-Bridge so we can flip the motor polarity and switch it ON and OFF at high speed through a PWM signal





Brushed DC motor driver design (in different directions and speed)

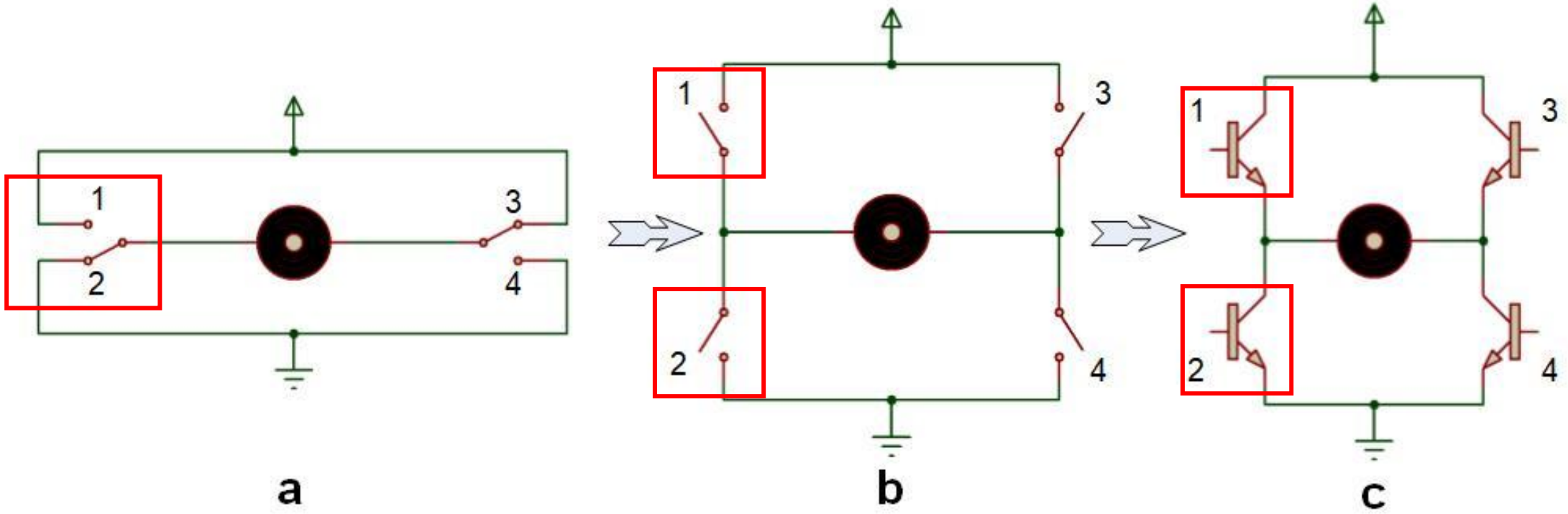
- The Figure “a” is what we have started in switching motor polarity. We have divided each double position switch into two simple switches in Figure “b”.
- If the switches 1and 4 are pressed, the motor will turn, and if 2 and 3 are pressed the motor will turn in the other direction.
- Note that if 1 and 2 are pressed at the same time, a short circuit happened as well if 3 and 4 are pressed together.





Brushed DC motor driver design (in different directions and speed)

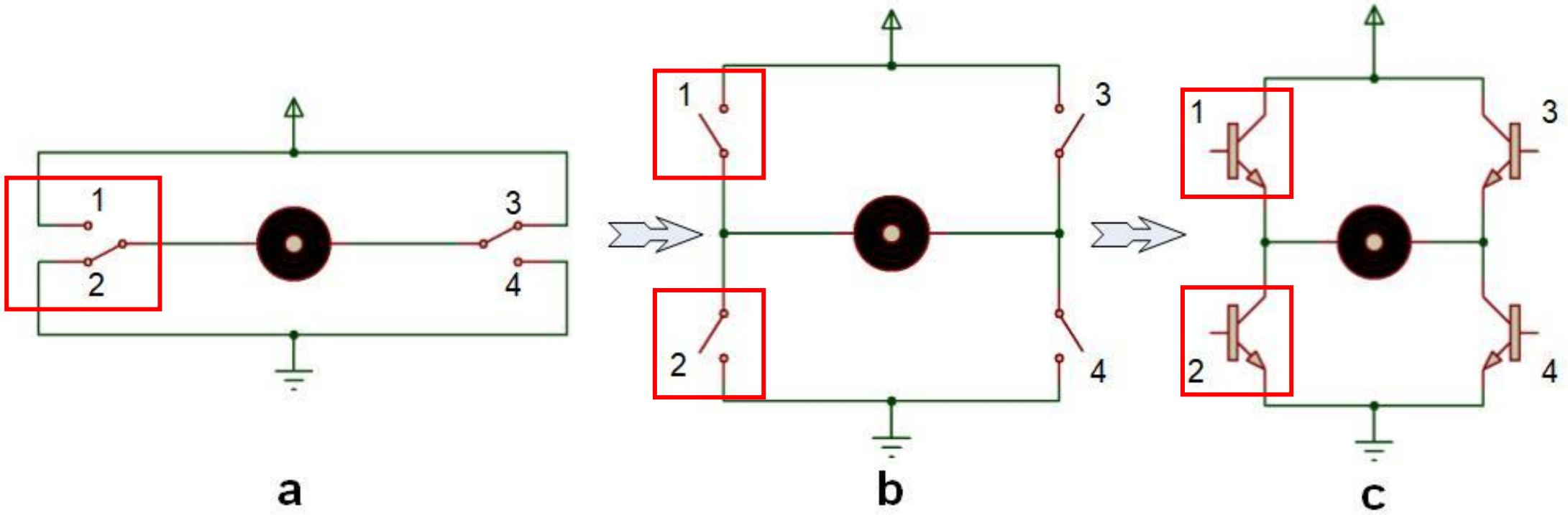
- In the Figure “c”, transistors replace the switches to get the H-bridge that we need to drive our motor in this application.
- By connecting the transistor bases of 1 and 4 and apply a PWM signal, the motor will turn at low speed.
- Also by connecting the transistor base of 2 and 3 and apply the PWM signal, the motor will turn in the other direction.





Brushed DC motor driver design (in different directions and speed)

- You may ask why we are not driving relays of Figure "a" by a PWM signal!
- The answer is that the relay response time is too small and does not switch ON and OFF at more than 20 Hz. Therefore, we replace the relays by transistors, which can do the same switching job at high frequency.

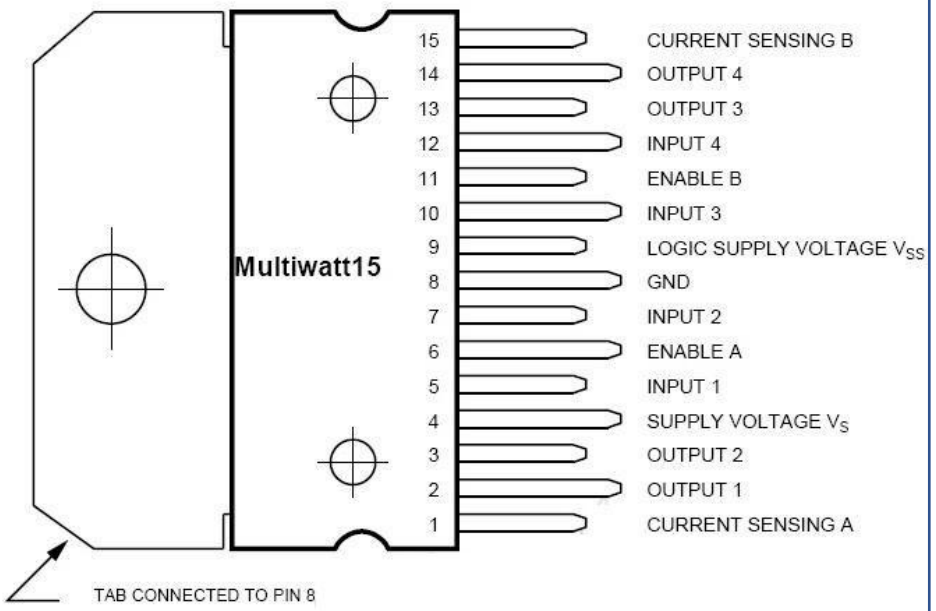




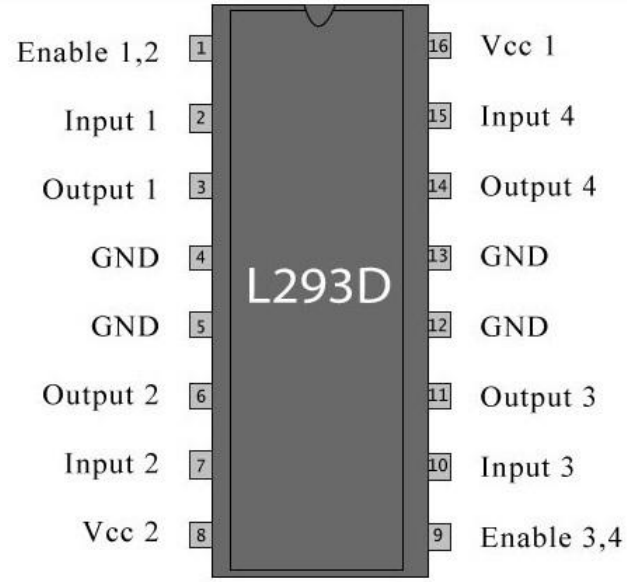
Brushed DC motor driver design (in different directions and speed)

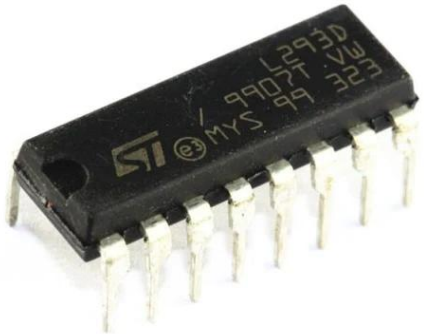
The L298 and L293 are two ships that contain two H-bridges that support 4A and 1A, respectively.
The L298 main feature is the internal protection against short circuit and overheating.





L298





L293

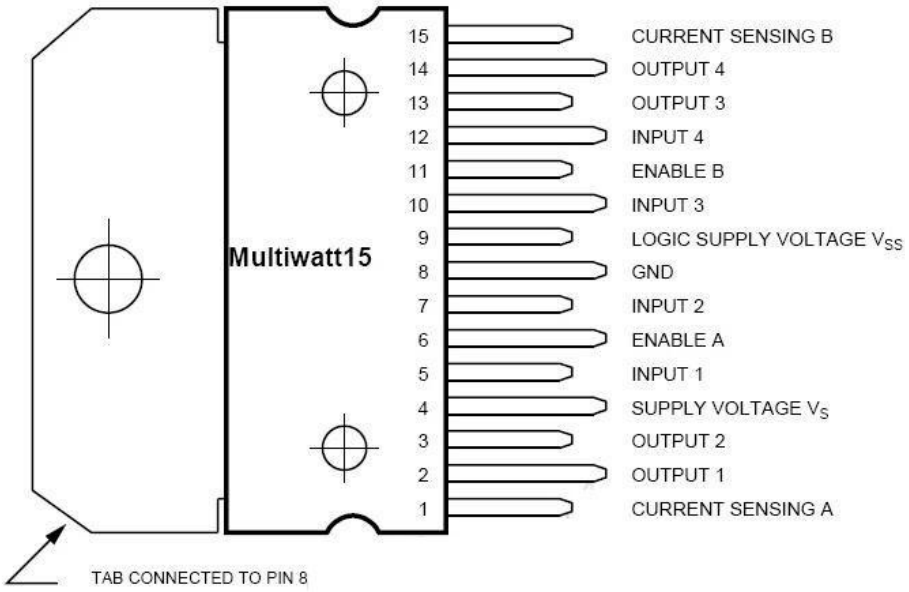


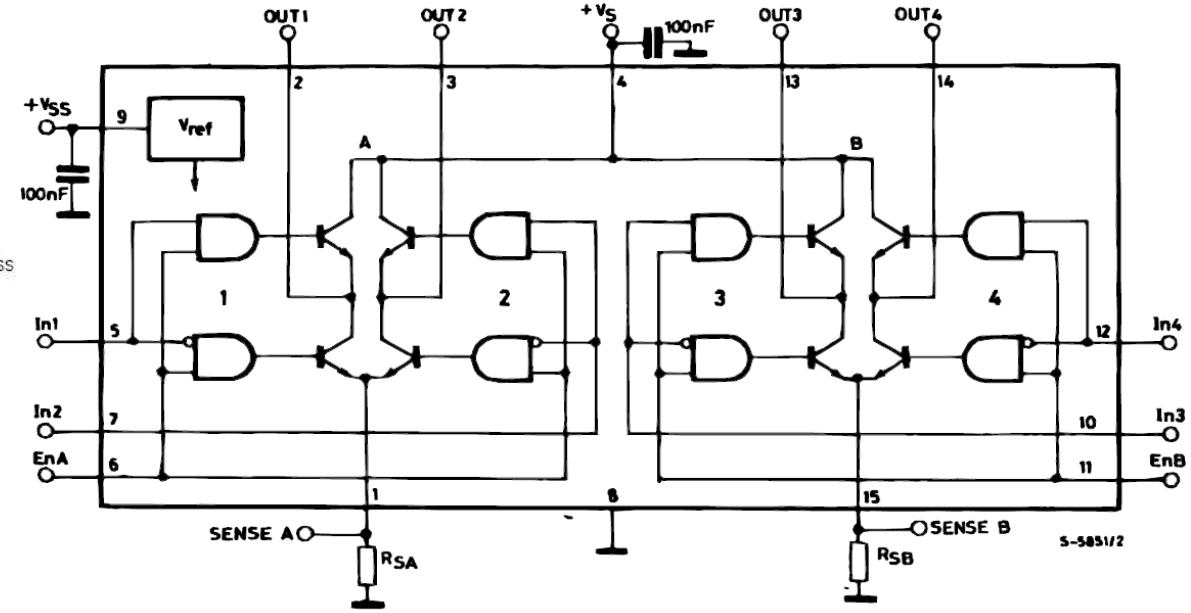
Brushed DC motor driver design (in different directions and speed)

The L298 can be used to drive two DC motors at the same time (It has two H-bridge circuits)



L298



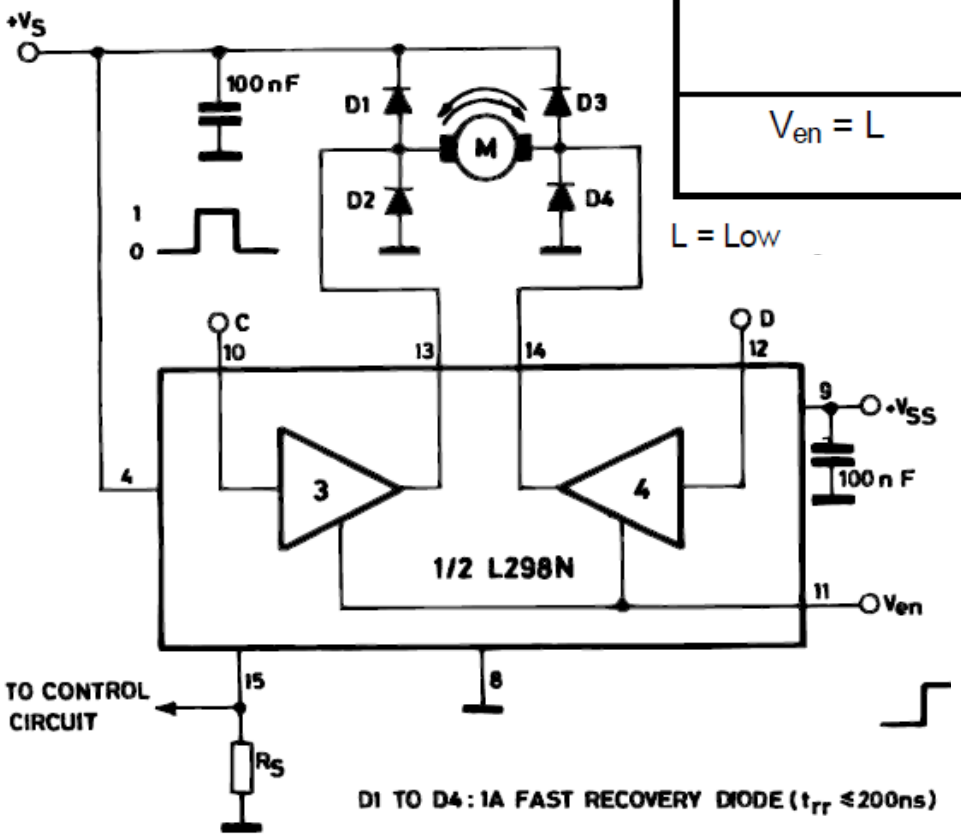


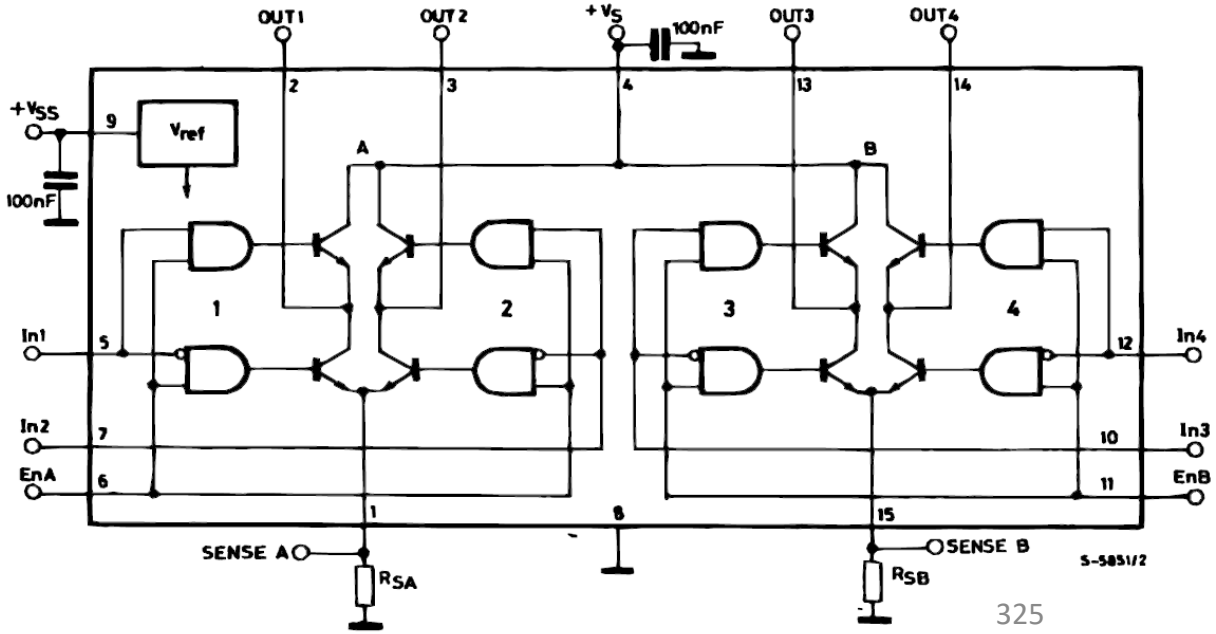


Brushed DC motor driver design (in different directions and speed)

Inputs		Function
V_{en} = H	C = H ; D = L	Forward
	C = L ; D = H	Reverse
	C = D	Fast Motor Stop
V_{en} = L	C = X ; D = X	Free Running Motor Stop

L = Low H = High X = Don't care

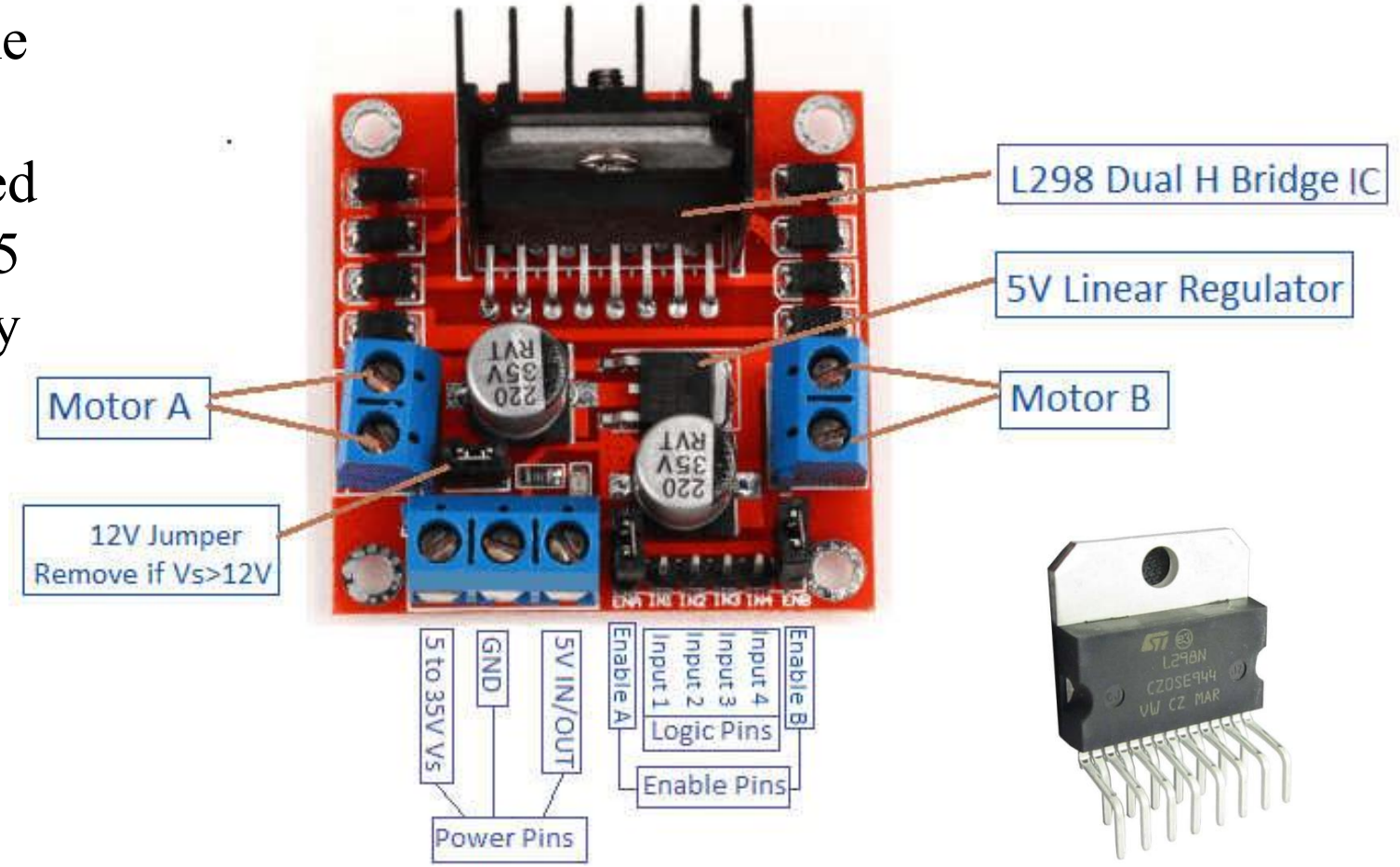






Brushed DC motor driver design (in different directions and speed)

- The L298 module is used to reduce the implementation time and difficulties.
- The module is already equipped with freewheeling diodes and 5 volt voltage regulator to supply external devices.



Week 10



Lab 8 / LO5

Working with PWM and DC motor



Tools and components

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. Breadboard wires (male-male)
  4. Power supply 12 volt

- Required components:
  1. Arduino UNO + USB cable
  2. Led x2
  3. Resistor from 1k to 4.7k ohm x1
  4. L298 chip or module x1
  5. Small DC motor



Brushed DC motor driver design (in different directions and speed)

```
#define in2 8
#define in1 9
#define ena 3
void setup() {
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  pinMode(ena, OUTPUT);
}

void Motor_left() {
  digitalWrite(in1, HIGH);
  digitalWrite(in2, LOW);
}

void Motor_right() {
  digitalWrite(in1, LOW);
  digitalWrite(in2, HIGH);
}

void Motor_stop() {
  digitalWrite(in1, LOW);
  digitalWrite(in2, LOW);
}
```

```
void loop() {
  analogWrite(ena, 255);
  Motor_left();
  delay(1000);

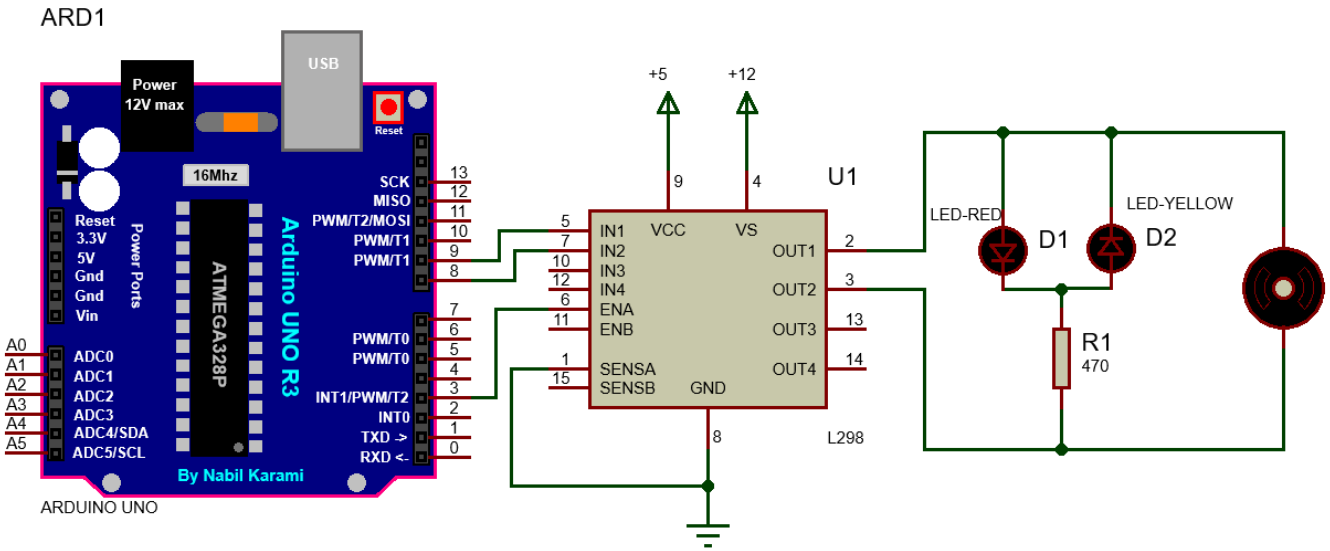
  Motor_stop();
  delay(1000);

  analogWrite(ena, 127);
  Motor_right();
  delay(1000);

  Motor_stop();
  delay(1000);

}
```

- The following code turns the motor left at full speed for 1 sec, then switch off the motor for 1sec, then right at half speed for 1sec, then off for 1 sec.





Week 11

Lecture 10 / LO5

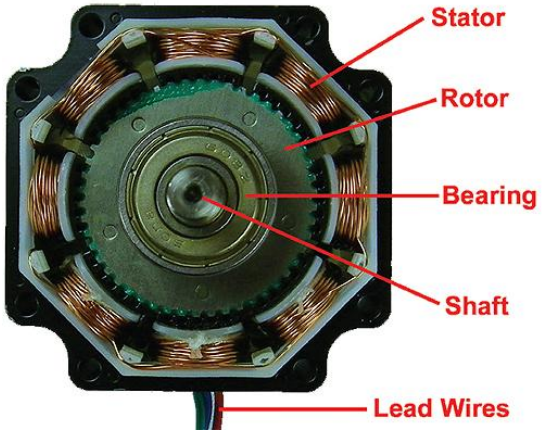
Stepper motors



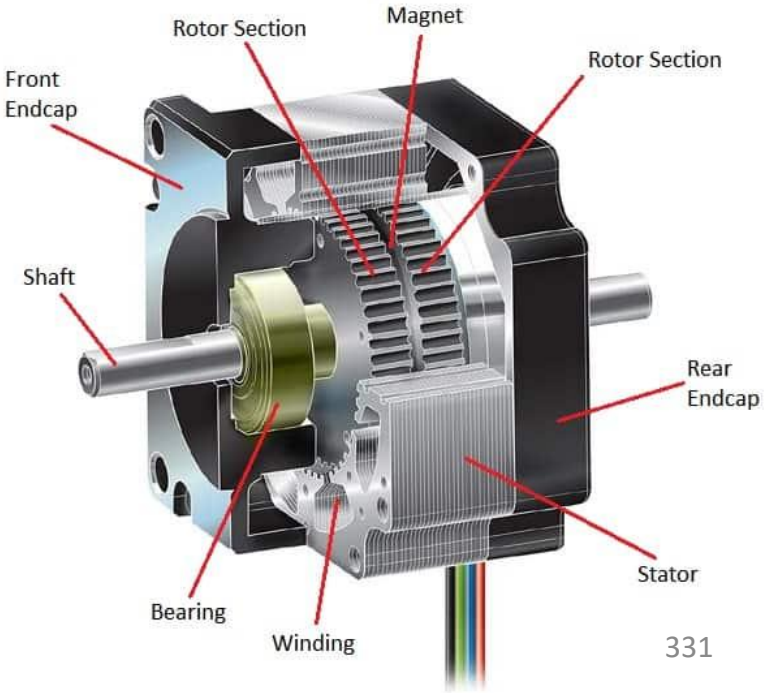
Stepper Motors

- A stepper motor is a brushless, synchronous electric motor that divides a full rotation into a large number of steps; therefore, the motor can be turned to a precise angle.
- Stepper motors operate differently from normal DC motors, which simply spin when voltage is applied to their terminals.
- They have multiple "toothed" electromagnets arranged around a central metal gear.
- The distance between teeth determines the stepper motor step angle.
- There are two basic arrangements for the electromagnetic coils: bi-polar and unipolar.





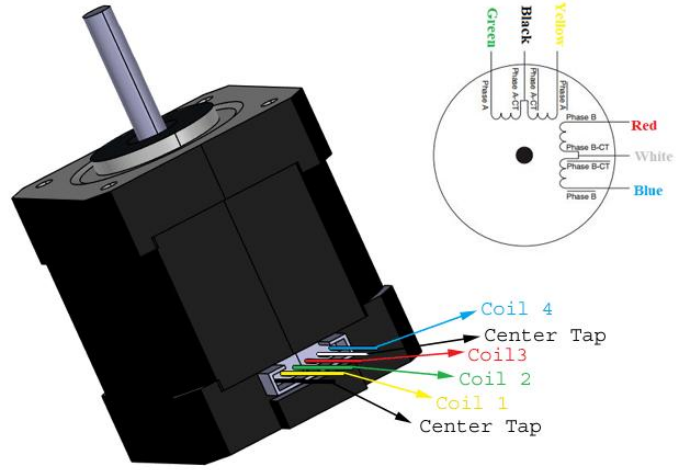


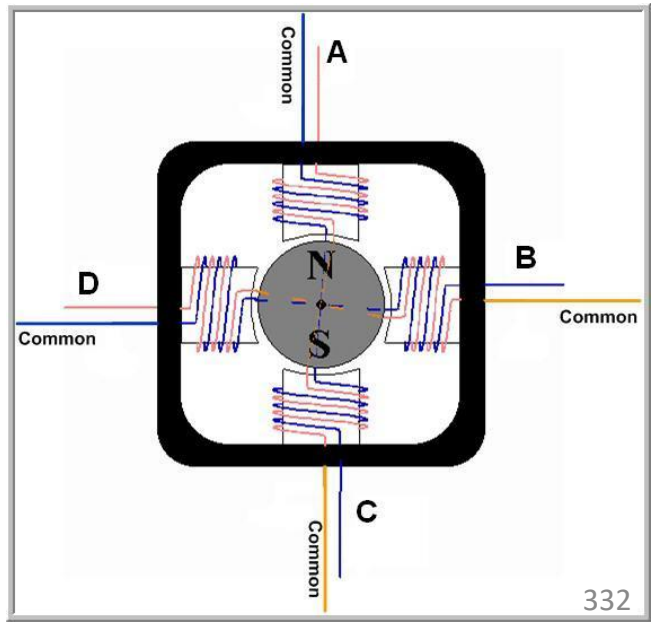




Unipolar Stepper Motors

- In a unipolar stepper motor, there are four separate electromagnets.
- To turn the motor on, first coil "1" is given current, then it is turned OFF and coil 2 is given current, then coil 3, then 4, and then 1 again in a repeating pattern.
- Current is only sent through the coils in one direction; thus the name unipolar.
- The stepper motor internal architecture is very difficult to understand; therefore, the Figure shows a theoretical design of stepper that turn 90-degree per step.
- A unipolar stepper motor will have five or six wires coming out of it.
- Four of those wires are each connected to one end of a coil. The extra wire (or 2) is called "common".

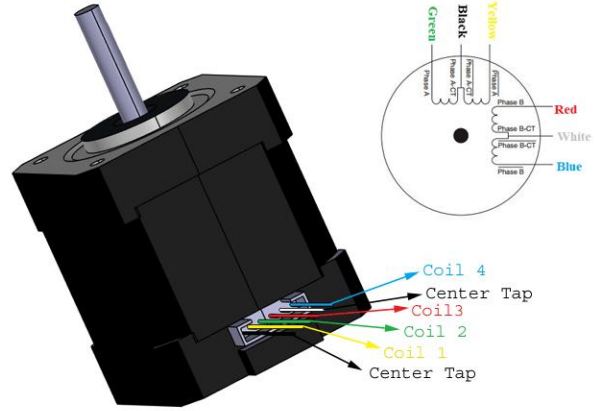


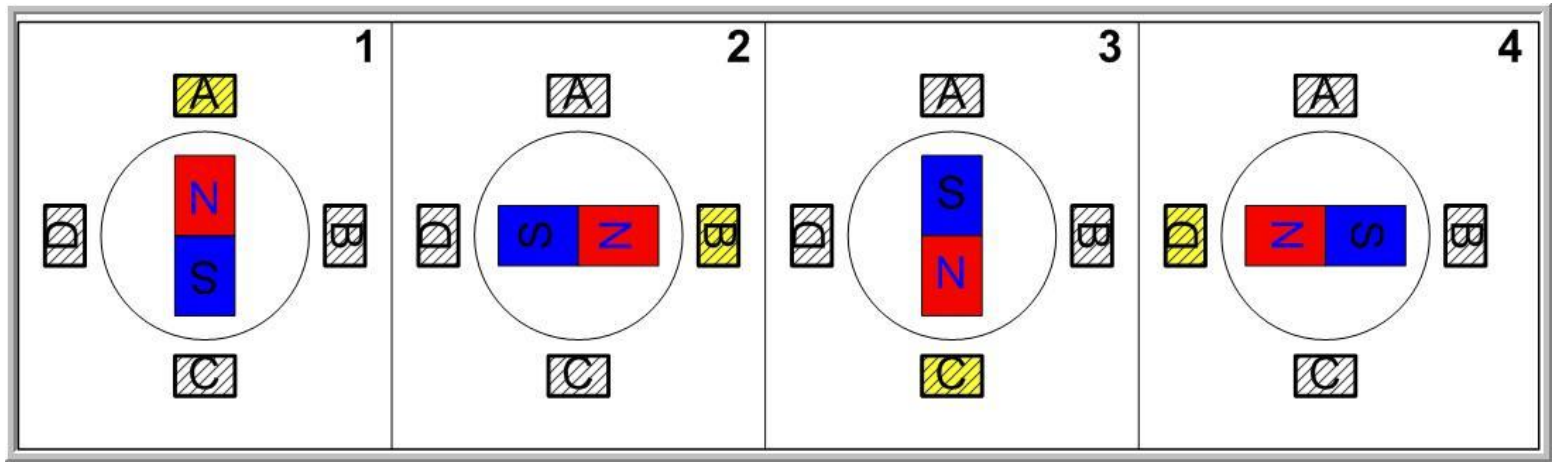


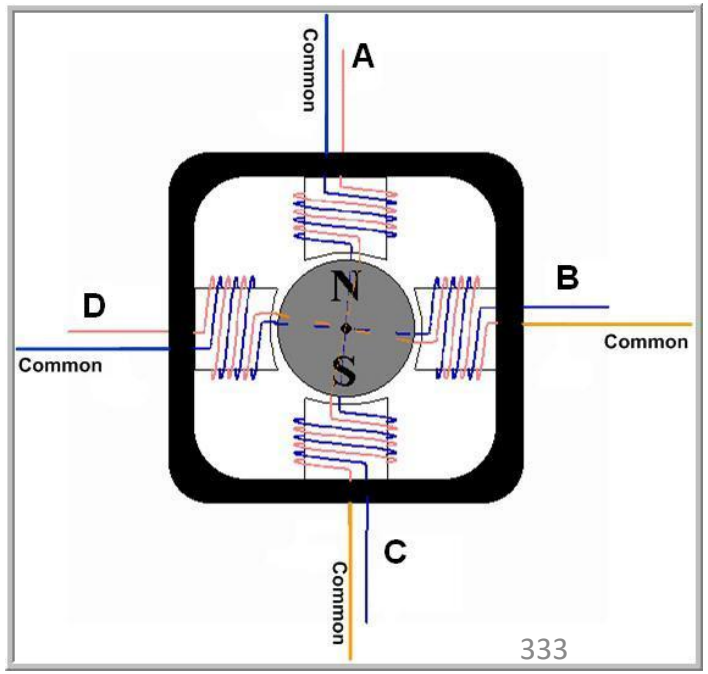


Unipolar Stepper Motors

- To operate the motor, the "common" wire(s) is (are) connected to the supply voltage, and the other four wires are connected to ground through transistors, so the transistors control whether current flows or not.
- The Arduino is used to activate the transistors in the right order.
- This case of operation makes unipolar motors popular with hobbyists; they are probably the cheapest way to get precise angular movements.







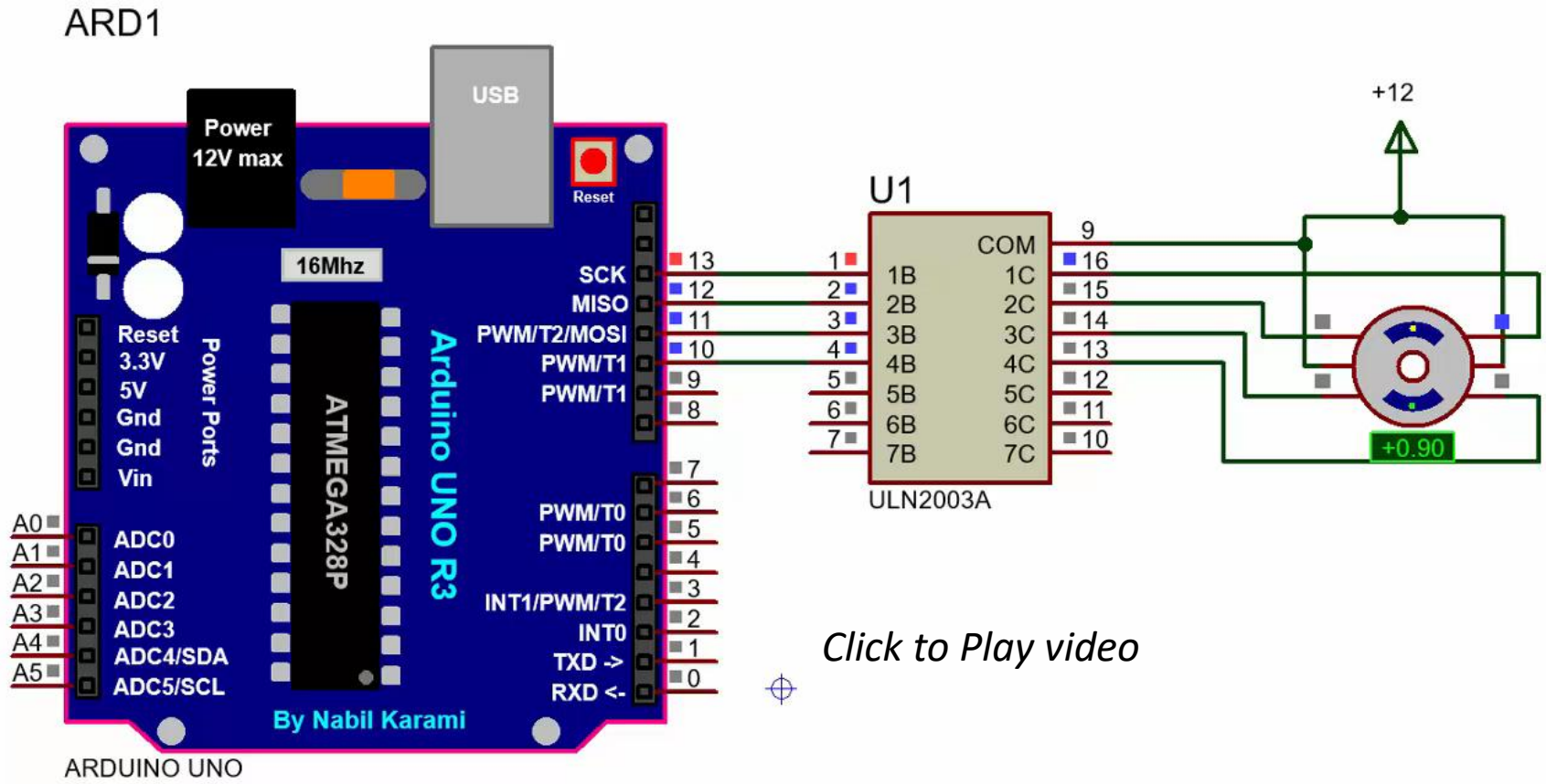


Unipolar Stepper Motors (Full step)

```
#define coilA 13
#define coilB 12
#define coilC 11
#define coilD 10

void setup() {
  pinMode(coilA, OUTPUT);
  pinMode(coilB, OUTPUT);
  pinMode(coilC, OUTPUT);
  pinMode(coilD, OUTPUT);
}

void loop() {
  digitalWrite(coilA, HIGH);
  delay(200);
  digitalWrite(coilA, LOW);
  digitalWrite(coilB, HIGH);
  delay(200);
  digitalWrite(coilB, LOW);
  digitalWrite(coilC, HIGH);
  delay(200);
  digitalWrite(coilC, LOW);
  digitalWrite(coilD, HIGH);
  delay(200);
  digitalWrite(coilD, LOW);
}
```

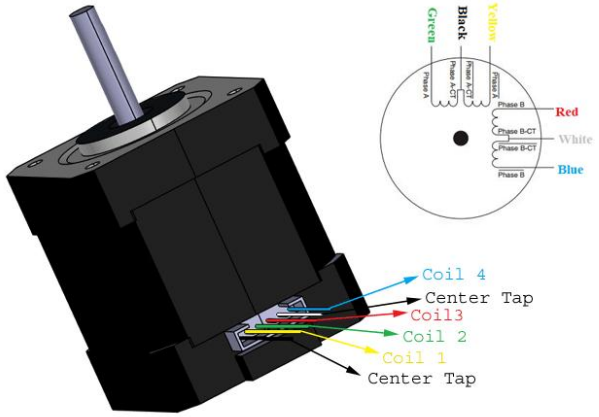


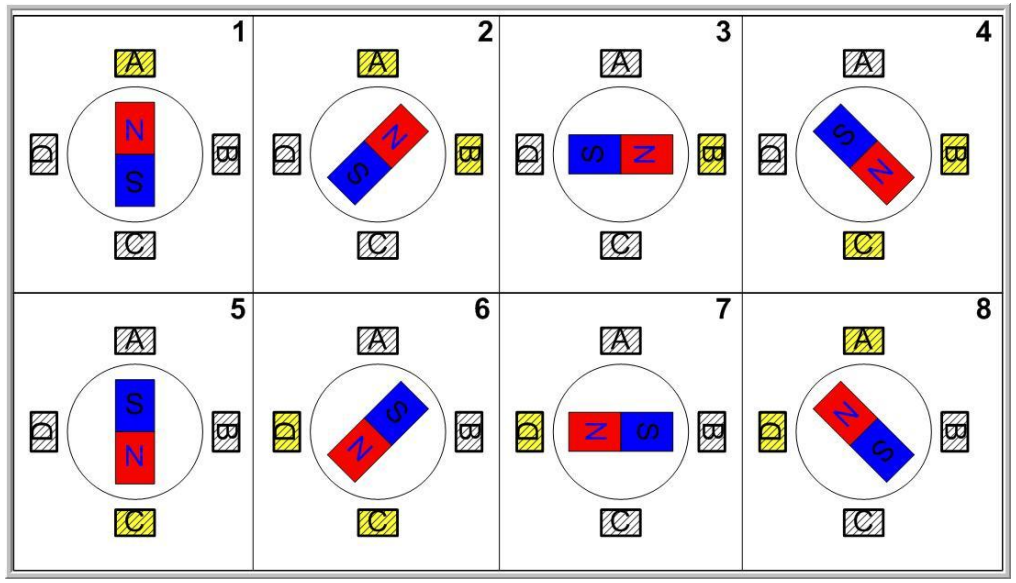
Click to Play video

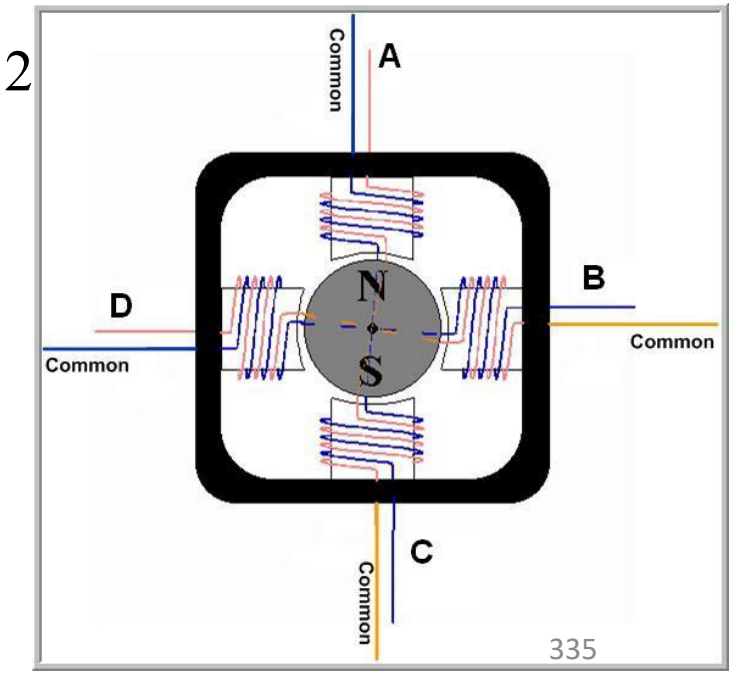


Unipolar Stepper Motors

- Another control method is also used in case half step and high torque is required.
- This techniques consists of driving coil A for example, then A and B at the same time.
- This will create a resulting force bigger than the force generated by one electromagnet.
- The step will be divided by 2 and the precision will be multiplied by 2







Lecture

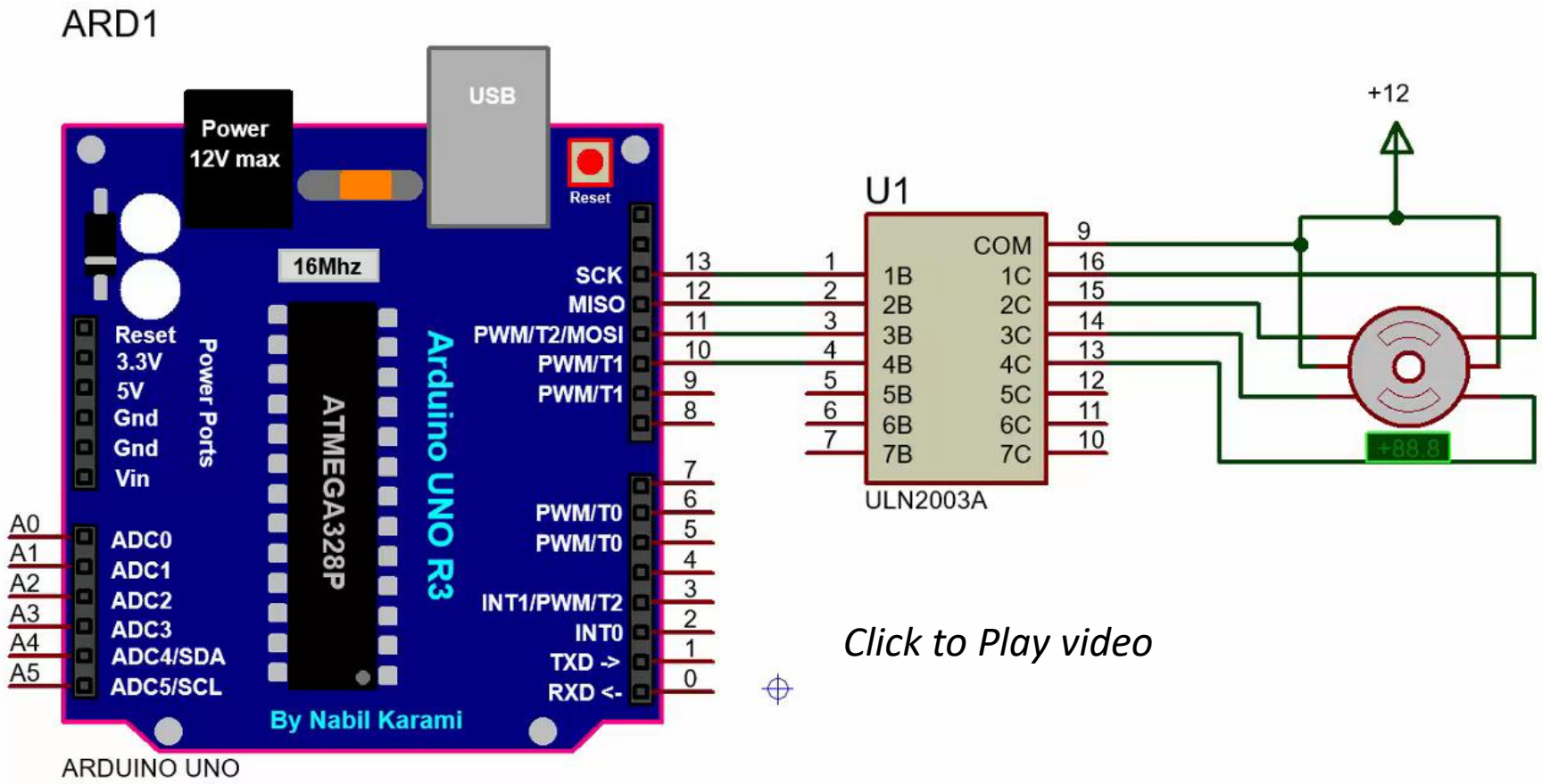


Unipolar Stepper Motors (Half step)

```
#define coilA 13
#define coilB 12
#define coilC 11
#define coilD 10

void setup() {
  pinMode(coilA, OUTPUT);
  pinMode(coilB, OUTPUT);
  pinMode(coilC, OUTPUT);
  pinMode(coilD, OUTPUT);
}

void loop() {
  digitalWrite(coilA, HIGH);
  delay(200);
  digitalWrite(coilB, HIGH);
  delay(200);
  digitalWrite(coilA, LOW);
  delay(200);
  digitalWrite(coilC, HIGH);
  delay(200);
  digitalWrite(coilB, LOW);
  delay(200);
  digitalWrite(coilD, HIGH);
  delay(200);
  digitalWrite(coilC, LOW);
  delay(200);
  digitalWrite(coilA, HIGH);
  delay(200);
  digitalWrite(coilD, LOW);
}
```



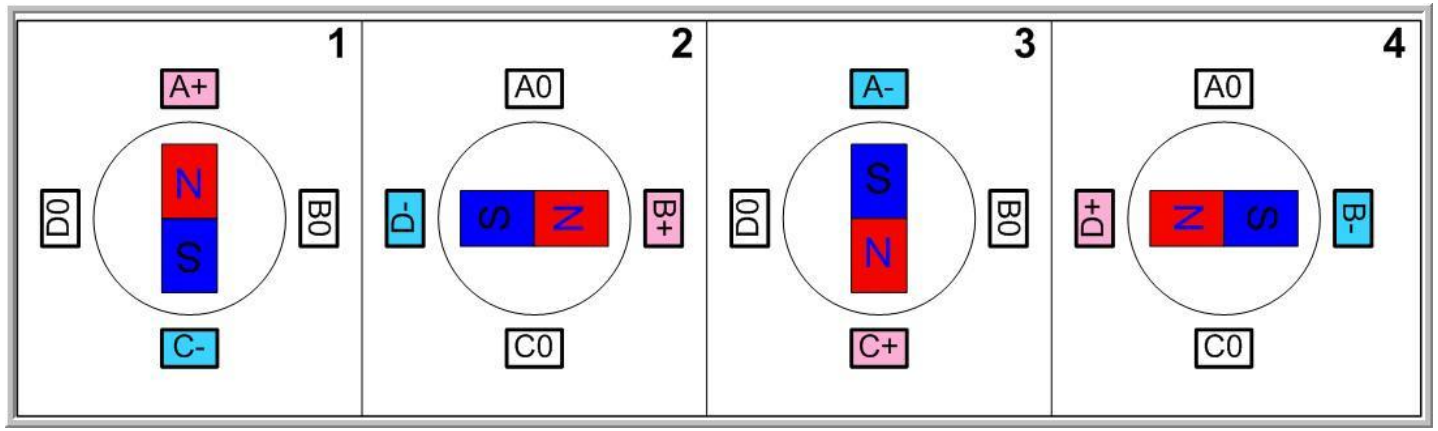
Click to Play video

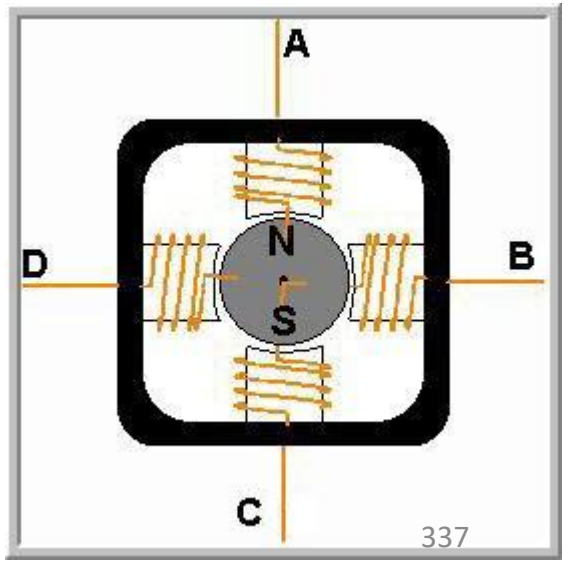


Bipolar Stepper Motors (Full step)

- The external design of bipolar motor is similar to the unipolar stepper, but they can be recognized via the number of wires.
- The bipolar motor has only four wires connected internally to two coils. Current must be sent through a coil first in one direction and then in the other direction; thus the name bipolar.
- Bipolar motors need more than four transistors to operate, but they are also more powerful than a unipolar motor of the same weight.
- To be able to drive current in both directions, we can use an H-bridge to control each coil or the L298 driver chip.







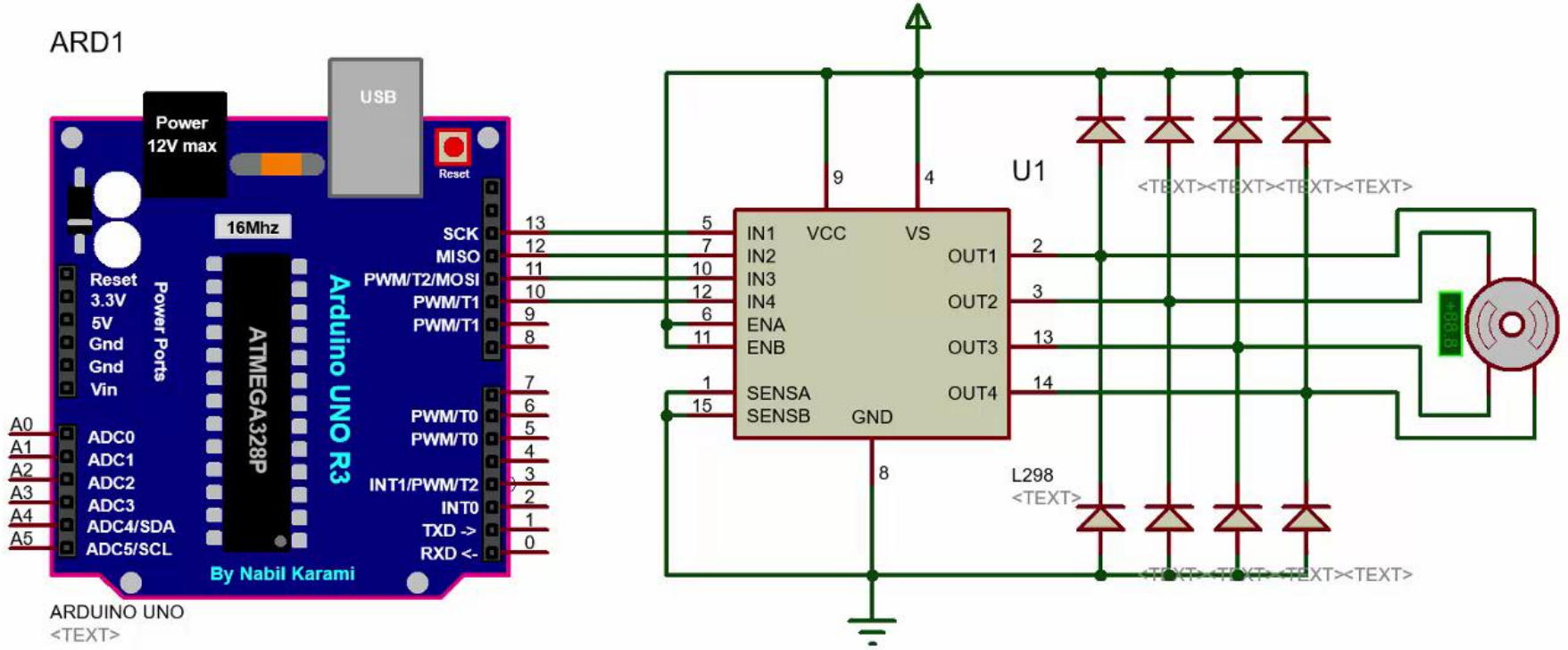


Bipolar Stepper Motors (Full step)

```
#define in1 13
#define in2 12
#define in3 11
#define in4 10

void setup() {
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  pinMode(in3, OUTPUT);
  pinMode(in4, OUTPUT);
}

void loop() {
  digitalWrite(in1, HIGH);
  delay(200);
  digitalWrite(in1, LOW);
  digitalWrite(in3, HIGH);
  delay(200);
  digitalWrite(in3, LOW);
  digitalWrite(in2, HIGH);
  delay(200);
  digitalWrite(in2, LOW);
  digitalWrite(in4, HIGH);
  delay(200);
  digitalWrite(in4, LOW);
}
```

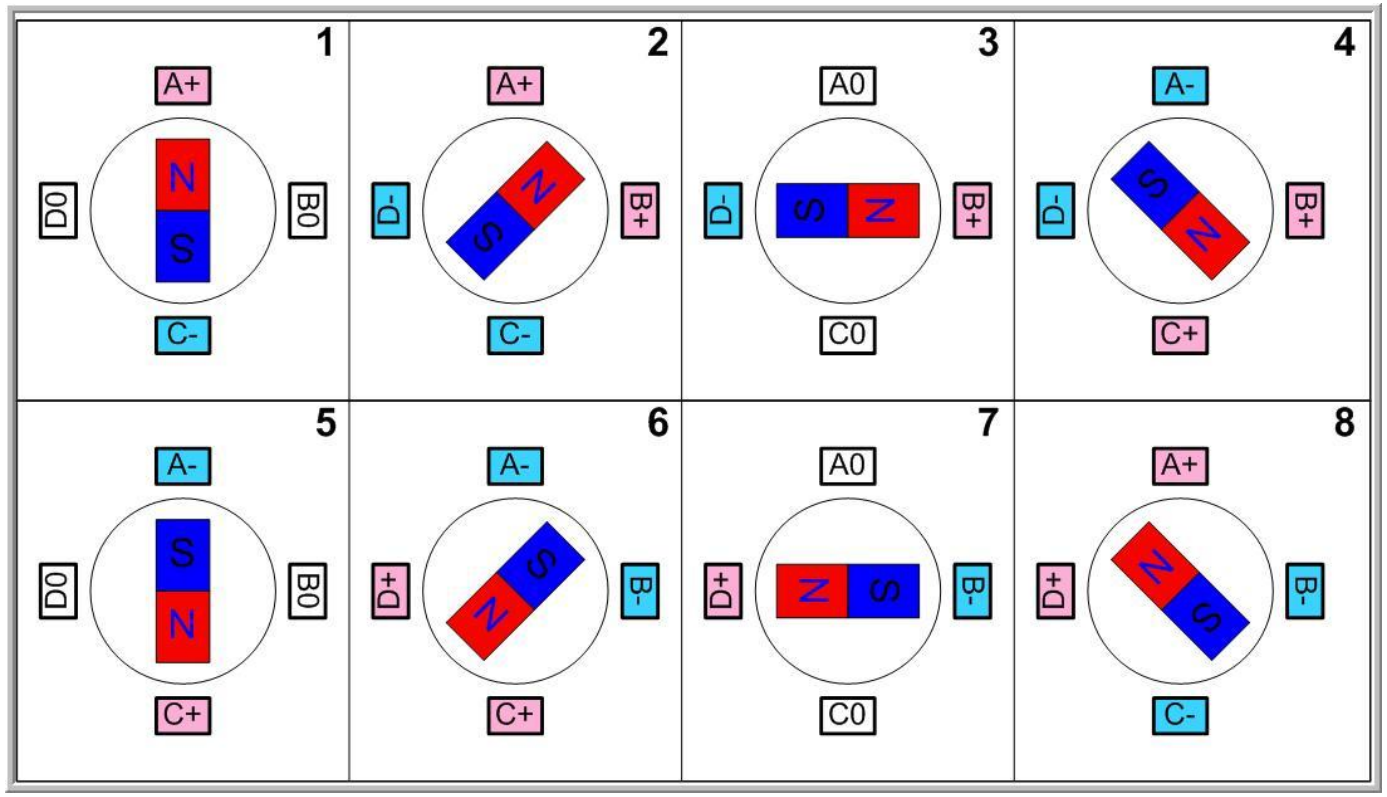


Click to Play video

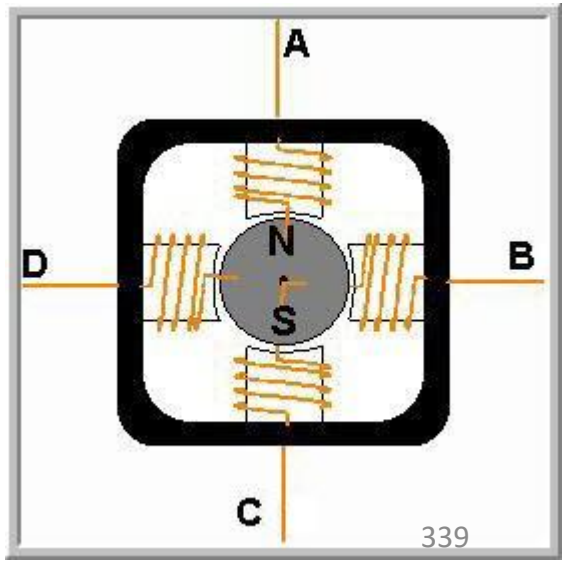


Bipolar Stepper Motors (Half Step)

- Similar to the unipolar stepper, the bipolar can also bi driven by a half step mode.
- This will reduce the step size and increase the torque







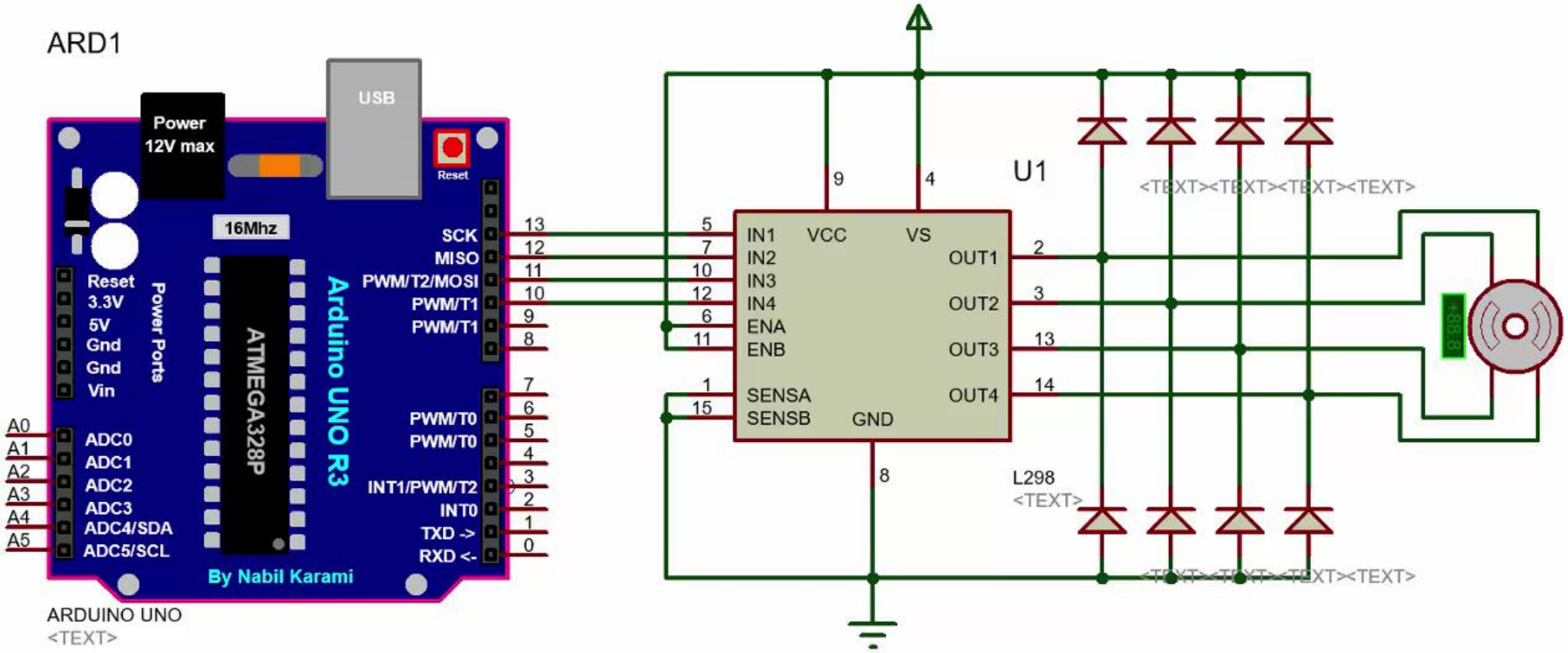


Bipolar Stepper Motors

```
#define in1 13
#define in2 12
#define in3 11
#define in4 10

void setup() {
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  pinMode(in3, OUTPUT);
  pinMode(in4, OUTPUT);
}

void loop() {
  digitalWrite(in1, HIGH);
  delay(200);
  digitalWrite(in3, HIGH);
  delay(200);
  digitalWrite(in1, LOW);
  delay(200);
  digitalWrite(in2, HIGH);
  delay(200);
  digitalWrite(in3, LOW);
  delay(200);
  digitalWrite(in4, HIGH);
  delay(200);
  digitalWrite(in2, LOW);
  delay(200);
  digitalWrite(in1, HIGH);
  delay(200);
  digitalWrite(in4, LOW);
}
```



Click to Play video



Week 11

Lab 9 / LO5

Working with Stepper Motors



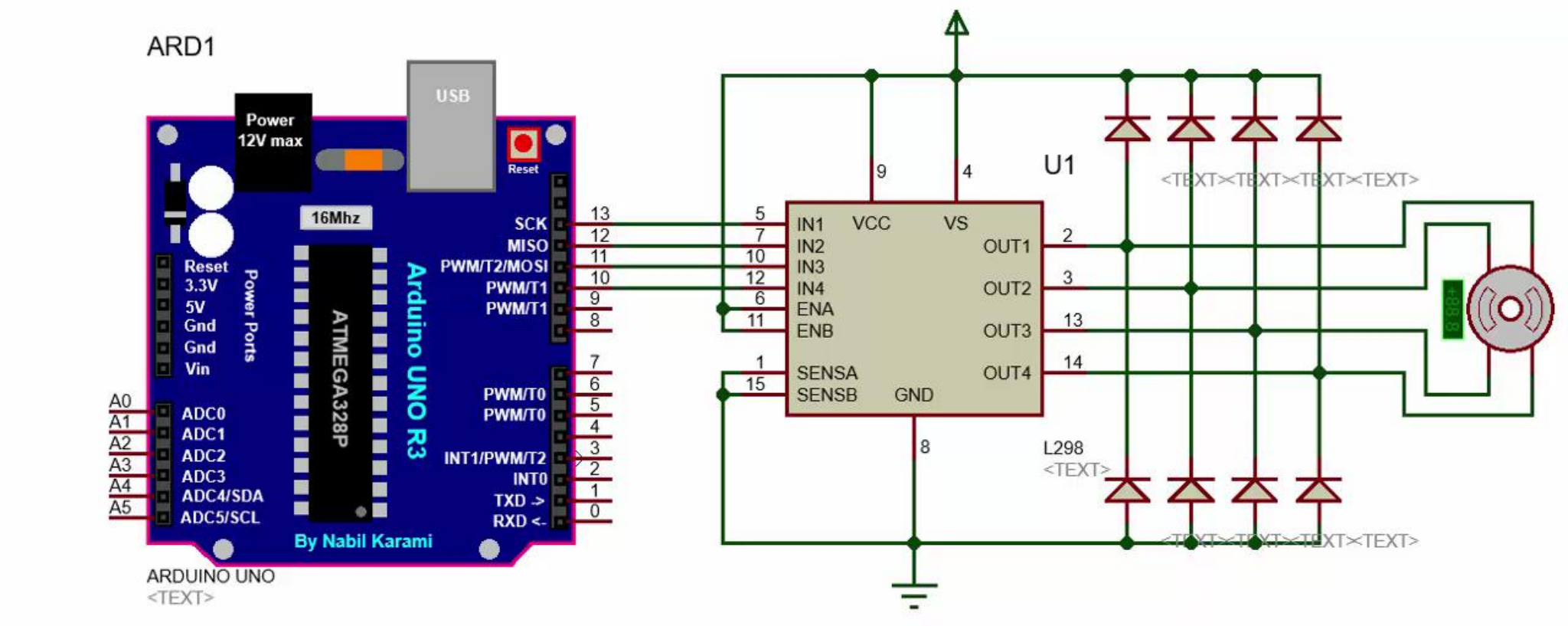
Tools and components

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. Breadboard wires (male-male)
  4. Power supply 12 volt

- Required components:
  1. Arduino UNO + USB cable
  2. L298 chip or module x1
  3. Bipolar stepper motor x1
  4. Fast recovery diodes UF4001 x8 (if available). No need if using L298 module



Bipolar Stepper Motors (Full Step/Half Step/CW/CCW)





Bipolar Stepper Motors (Full Step/Half Step/CW/CCW)

```
#define in1 13
#define in3 11
#define in2 12
#define in4 10

void setup() {
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  pinMode(in3, OUTPUT);
  pinMode(in4, OUTPUT);
}
```

```
void stepper_cw_full_step(int _speed) {
  digitalWrite(in1, HIGH);
  delay(_speed);
  digitalWrite(in1, LOW);
  digitalWrite(in3, HIGH);
  delay(_speed);
  digitalWrite(in3, LOW);
  digitalWrite(in2, HIGH);
  delay(_speed);
  digitalWrite(in2, LOW);
  digitalWrite(in4, HIGH);
  delay(_speed);
  digitalWrite(in4, LOW);
}
void stepper_ccw_full_step(int _speed) {
  digitalWrite(in1, HIGH);
  delay(_speed);
  digitalWrite(in1, LOW);
  digitalWrite(in4, HIGH);
  delay(_speed);
  digitalWrite(in4, LOW);
  digitalWrite(in2, HIGH);
  delay(_speed);
  digitalWrite(in2, LOW);
  digitalWrite(in3, HIGH);
  delay(_speed);
  digitalWrite(in3, LOW);
}
```

```
void stepper_cw_half_step(int _speed) {
  digitalWrite(in1, HIGH);
  delay(_speed);
  digitalWrite(in3, HIGH);
  delay(_speed);
  digitalWrite(in1, LOW);
  delay(_speed);
  digitalWrite(in2, HIGH);
  delay(_speed);
  digitalWrite(in3, LOW);
  delay(_speed);
  digitalWrite(in4, HIGH);
  delay(_speed);
  digitalWrite(in2, LOW);
  delay(_speed);
  digitalWrite(in1, HIGH);
  delay(_speed);
  digitalWrite(in4, LOW);
}
```



Bipolar Stepper Motors (Full Step/Half Step/CW/CCW)

```
62 void stepper_ccw_half_step(int _speed) {
63   digitalWrite(in1, HIGH);
64   delay(_speed);
65   digitalWrite(in4, HIGH);
66   delay(_speed);
67   digitalWrite(in1, LOW);
68   delay(_speed);
69   digitalWrite(in2, HIGH);
70   delay(_speed);
71   digitalWrite(in4, LOW);
72   delay(_speed);
73   digitalWrite(in3, HIGH);
74   delay(_speed);
75   digitalWrite(in2, LOW);
76   delay(_speed);
77   digitalWrite(in1, HIGH);
78   delay(_speed);
79   digitalWrite(in3, LOW);
80 }
```

```
84 void loop() {
85   for (int i = 0; i < 10; i++)
86   {
87     stepper_cw_full_step(50);
88   }
89
90   for (int i = 0; i < 10; i++)
91   {
92     stepper_ccw_full_step(50);
93   }
94
95   for (int i = 0; i < 10; i++)
96   {
97     stepper_ccw_half_step(50);
98   }
99
100  for (int i = 0; i < 10; i++)
101  {
102    stepper_cw_half_step(80);
103  }
104 }
```



Week 12

Lecture 11 / LO5

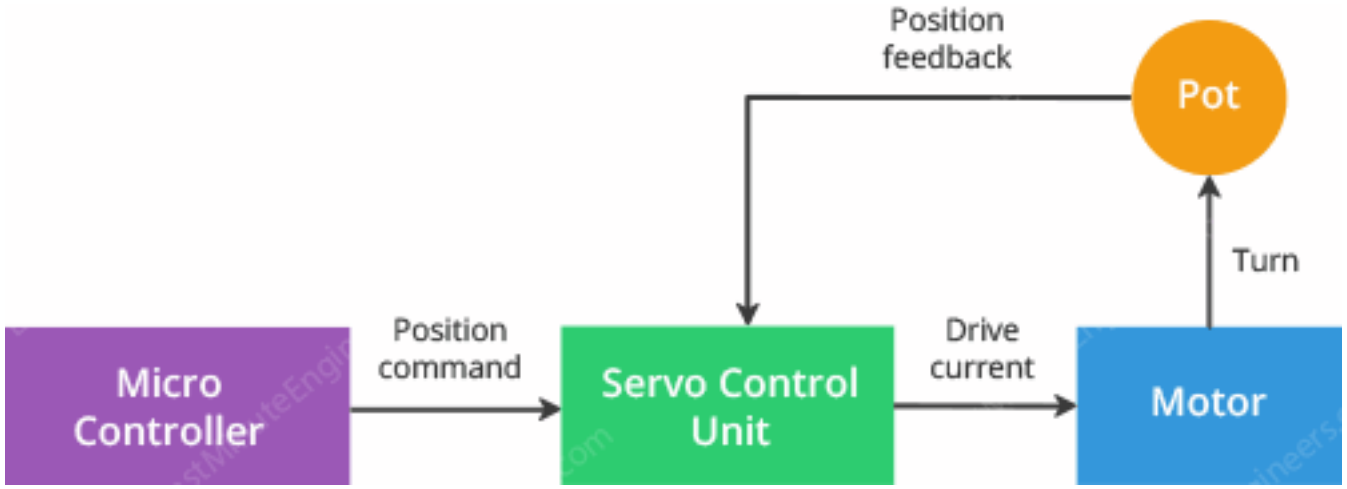
Servo Motor



Servo motor

- Unlike DC motors, you can precisely control the positioning of these motors. Instruct them where to point, and they'll do it for you.
- They are useful in many robotics projects, such as for turning the front wheels on an RC model for steering or pivoting a sensor to look around on a robotic vehicle.
- Servo is a general term for a **closed loop control system**.
- A closed loop system uses the feedback signal to adjust the speed and direction of the motor to achieve the desired result.
- RC servo motor works on the same principal.
- It contains a small DC motor connected to the output shaft through the gears.



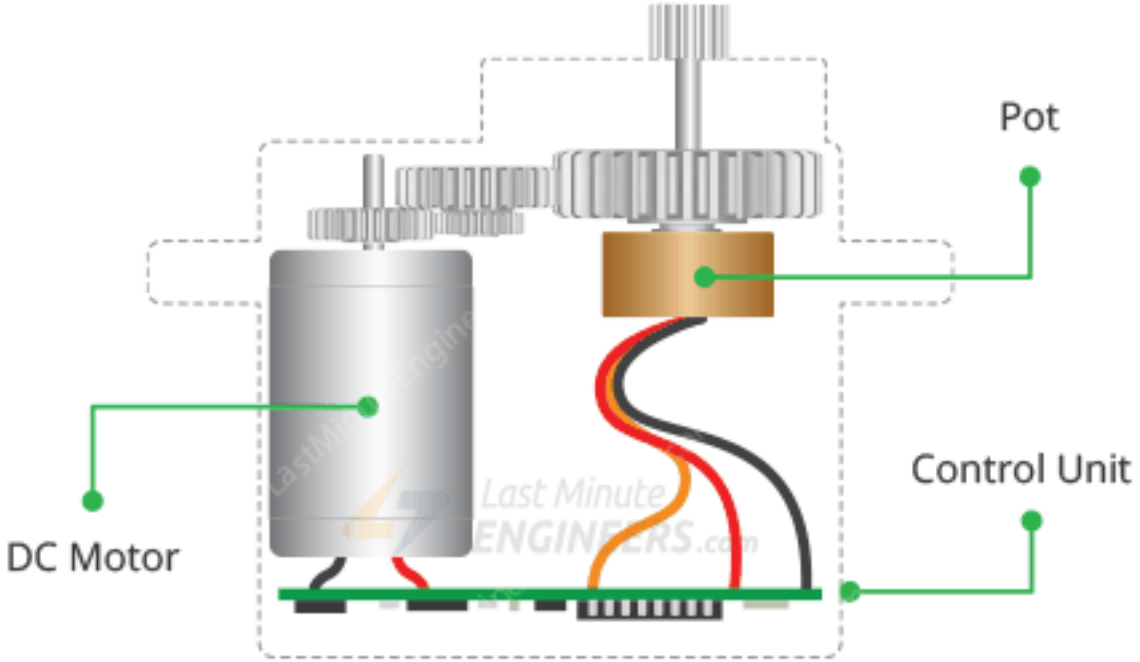






Servo motor

- The output shaft drives a servo arm and is also connected to a potentiometer (pot).
- The potentiometer provides position feedback to the servo control unit where the current position of the motor is compared to the target position.
- According to the error, the control unit corrects the actual position of the motor so that it matches the target position.

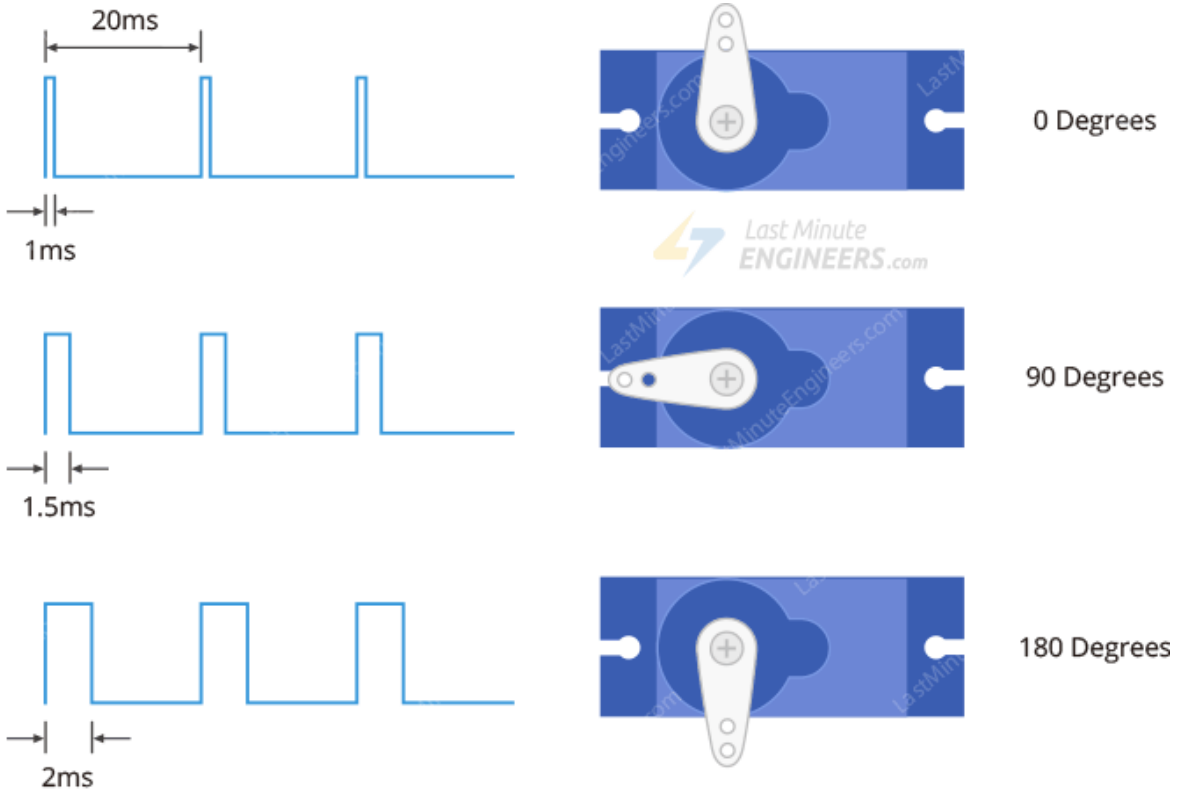






Servo motor

- You can control the servo motor by sending a **series of pulses** to the signal line. A conventional analog servo motor expects to receive a pulse roughly every 20 milliseconds (i.e. signal should be 50Hz).
- The length of the pulse determines the position of the servo motor.



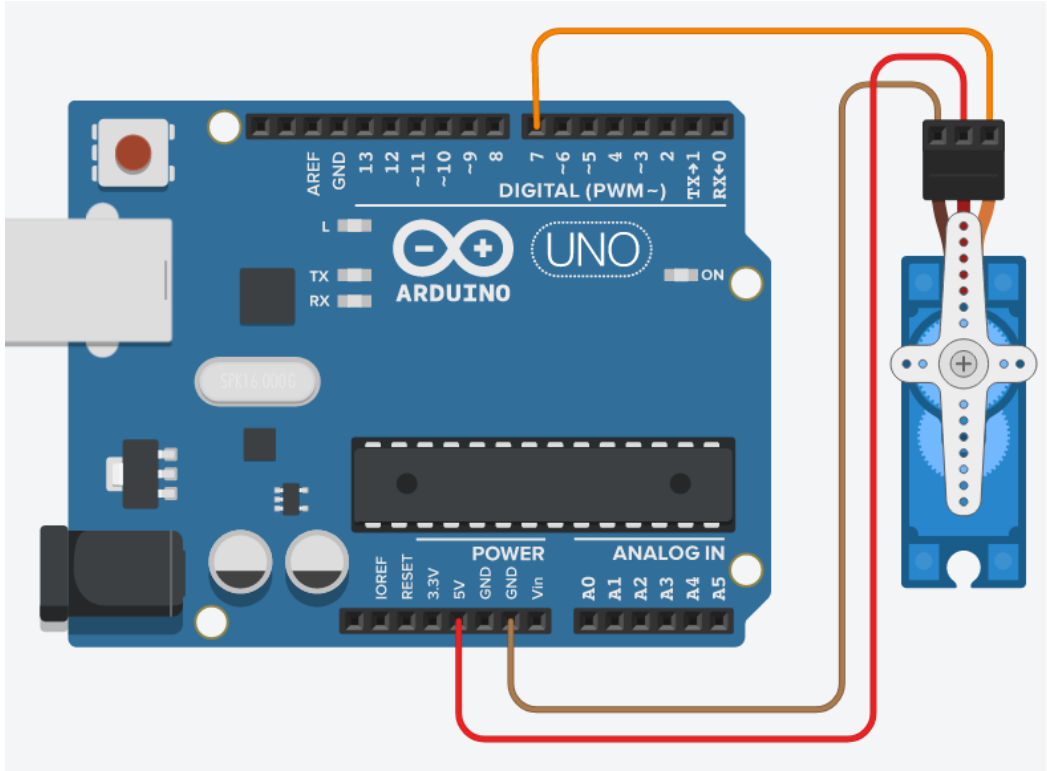
•If the pulse is high for 1ms, then the servo angle will be zero.
•If the pulse is high for 1.5ms, then the servo will be at its center position.
•If the pulse is high for 2ms, then the servo will at 180 degrees.
•Pulses ranging between 1ms and 2ms will move the servo shaft through the full 180 degrees of its travel.



Servo motor

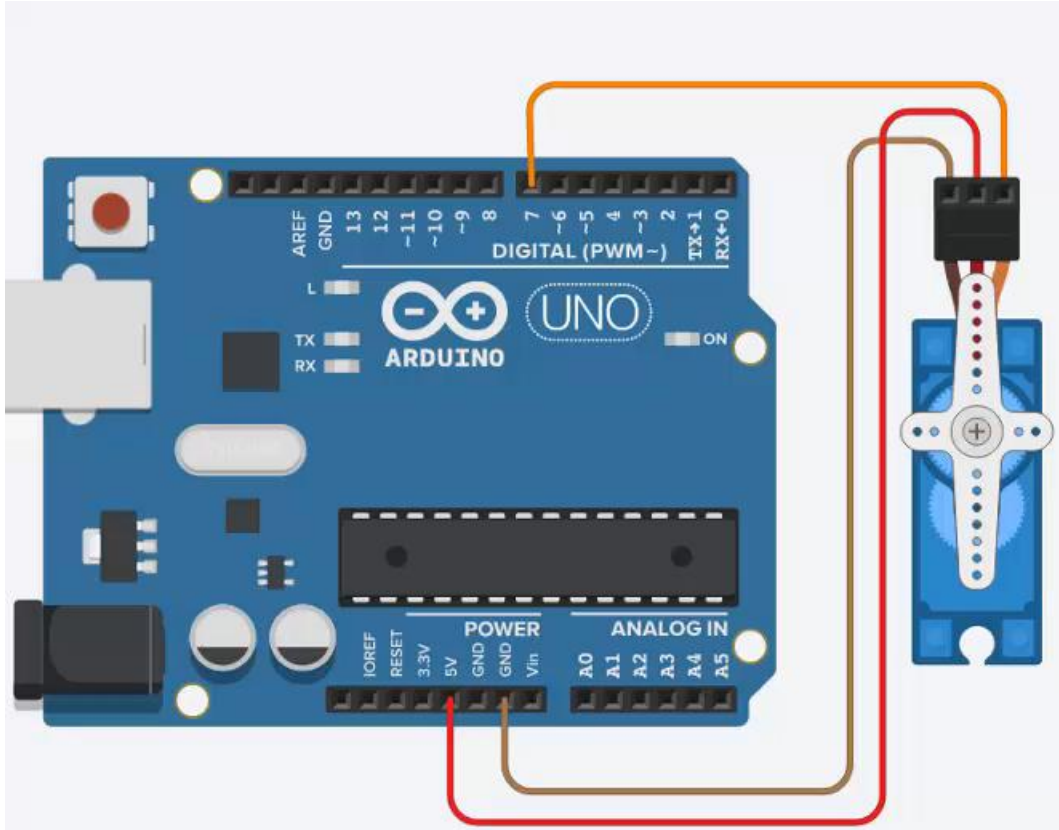
```
#include <Servo.h>
Servo my_servo;
void setup() {
  Serial.begin(9600);
  my_servo.attach(7);
}

void loop() {
  for (int i = 0; i < 180; i++) {
    my_servo.write(i);
    Serial.print("Angle is: "); Serial.println(i);
    delay(50);
  }
}
```





Servo motor




```
#include <Servo.h>
Servo my_servo;
void setup() {
  Serial.begin(9600);
  my_servo.attach(7);
}

void loop() {
  for (int i = 0; i < 180; i++) {
    my_servo.write(i);
    Serial.print("Angle is: "); Serial.println(i);
    delay(50);
  }
}
```

Serial Monitor



Week 12

Lab 10 / LO5

Servo Motor with UART



Week 13

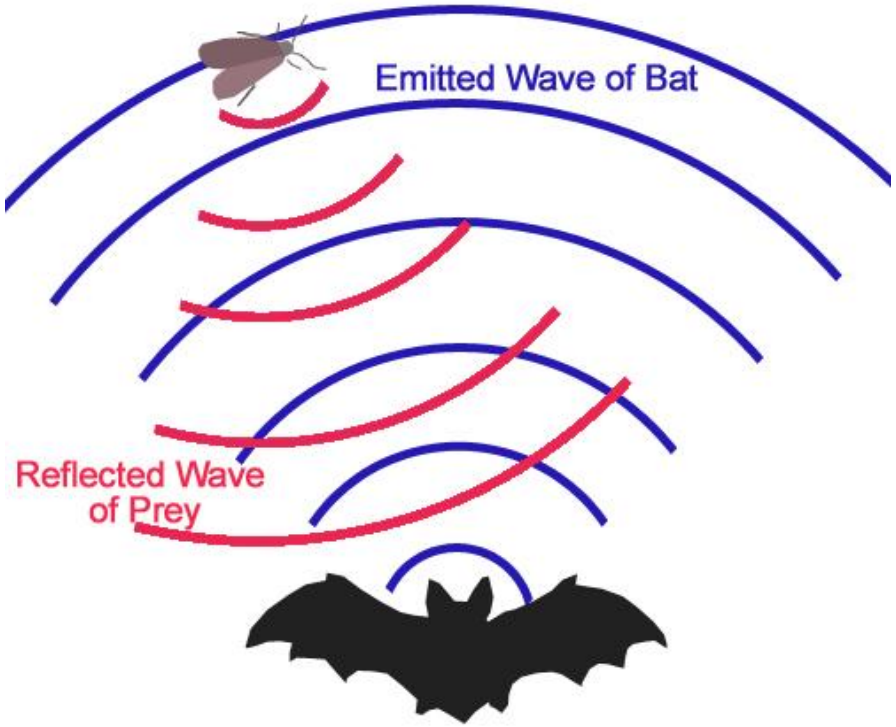
Lecture 12 / LO5

Ultrasonic sensor, PIR and LDR sensors



Ultrasonic sensor

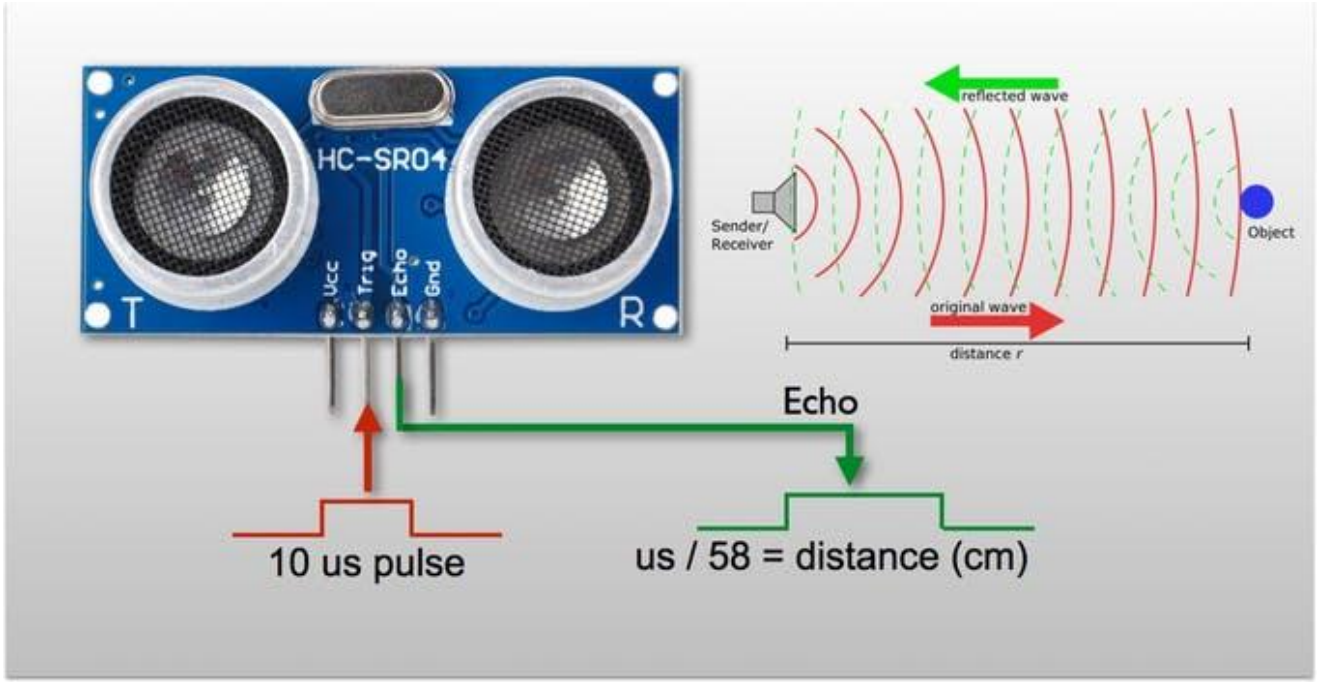
- Bats are not blind, but no animal can see in complete darkness. Since many kinds of bats hunt insects at night, they have an additional trick called echolocation for finding their way in the dark.
- The bat uses its mouth to create sounds that bounce off nearby objects, such as a moth.
- When the sound comes back to the bat's ears, the bat can tell where the object is.
- Using echolocation, the bat can find its prey and snatch it in midair.





Ultrasonic sensor

- An **ultrasonic sensor** is an electronic device that detects a target object by emitting ultrasonic sound waves, and converts the reflected sound into an electrical signal.
- Ultrasonic sensors work by using sound frequencies higher than the audible limit of human hearing (around 20 kHz), which is typically in the range of 25 to 50 kHz.

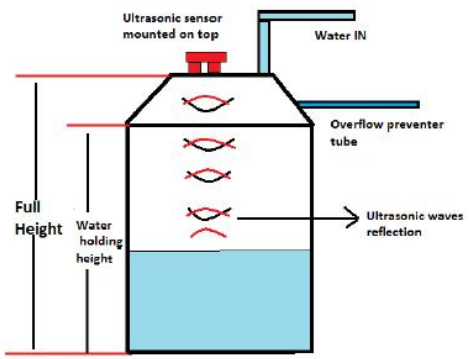




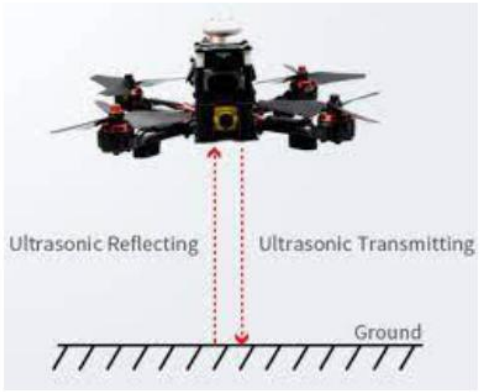
Ultrasonic sensor

There are two types of ultrasonic sensors

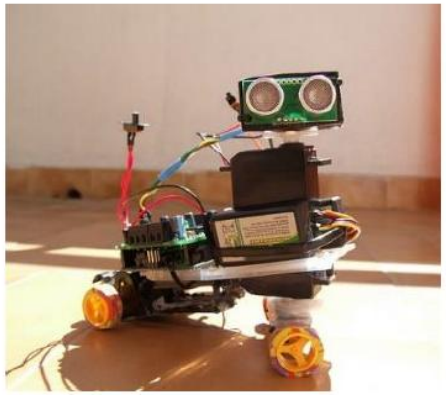
- **Proximity Detection:** An object passing within the preset range will be detected and generate an output signal. The detect point is independent of target size, material or reflectivity.
- **Ranging Measurement:** Precise distance(s) of an object moving to and from the sensor are measured via time intervals between transmitted and reflected bursts of ultrasonic sound. Distance change is continuously calculated and outputted.



Liquid Level Measurement



Distance Measurement



Robot Navigation

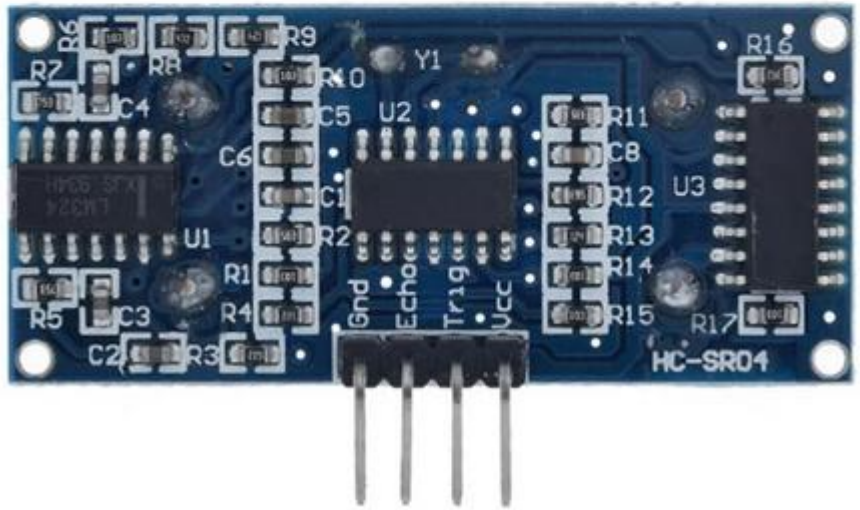




Ultrasonic sensor

- The HC-SR04 ultrasonic sensor uses sonar to determine distance to an object like bats do.
- It offers excellent non-contact range detection with high accuracy and stable readings in an easy-to-use package.
- Range from 10 cm to 400 cm.
- It operation is not affected by sunlight or black material, although acoustically soft materials like cloth can be difficult to detect. It comes complete with ultrasonic transmitter and receiver module.
- If you look at the back of the range finder, you will see:-
  - Behind the transmitting transducer, there is the MAX3232 IC which controls the transmitting transducer.
  - Behind the receiving transducer, there is the LM324 IC which is a quad Op-Amp that amplifies the signal generated by the receiving transducer into a signal that is strong enough to transmit to the Arduino.



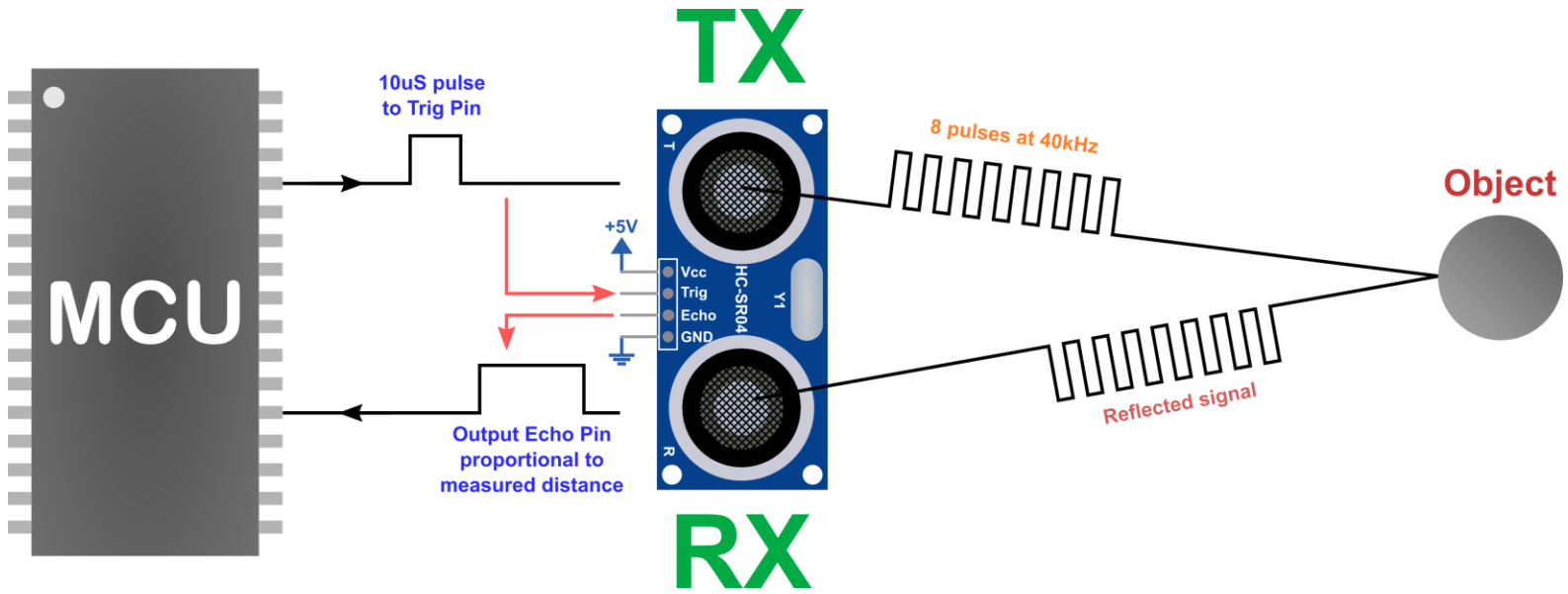




Ultrasonic sensor

The sensor works as follows, for a single measurement:

- 1. The sensor receives a trigger signal through the **Trig** pin. This signal should be a high (5V) pulse for at least 10µs.
- 2. The sensor transmit out 8 cycle of ultrasonic burst at 40kHz. At this time, it sets the **Echo** pin to HIGH (5V).When the sensor detects echo on the receiver, it will set the **Echo** pin to LOW.



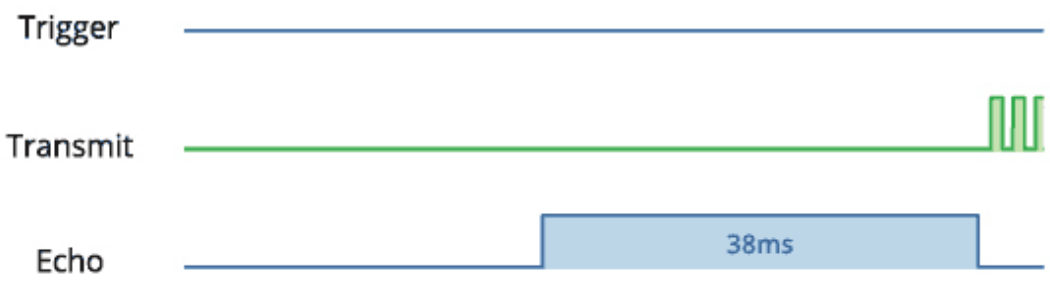


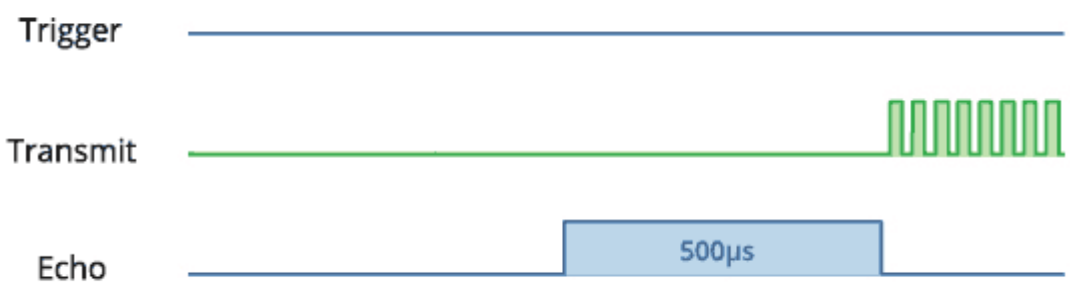
Ultrasonic sensor









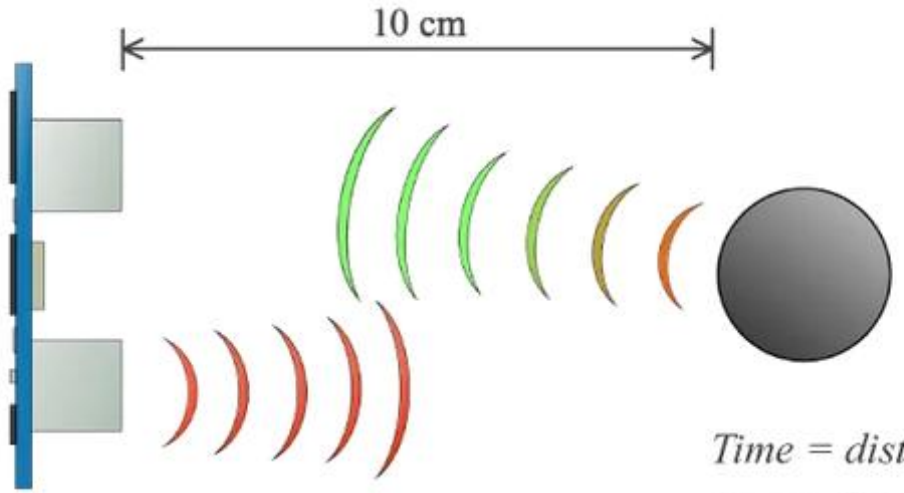




Ultrasonic sensor

To get the distance

- 1. Measure the width of the **Echo** pulse (in seconds)
- 2. Calculate the distance = ½ x Echo width x speed of sound (where speed of sound in the air is 346 m/s @ 20°C.)



speed of sound:

$v = 340\ m/s$

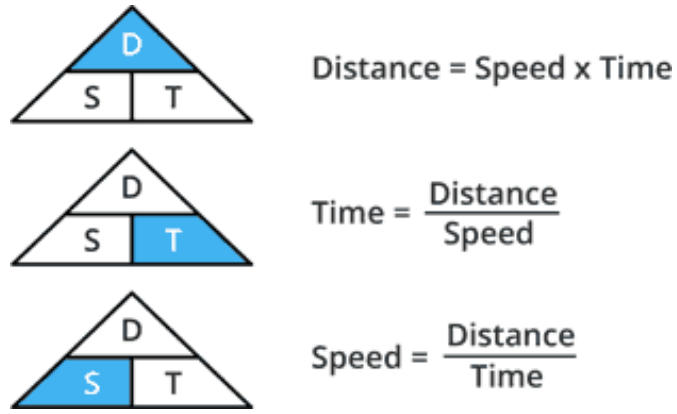
$v = 0{,}034\ cm/\mu s$

Time = distance / speed:

$t = s / v = 10 / 0{,}034 = 294\ \mu s$

Distance:

$s = t \cdot 0{,}034 / 2$

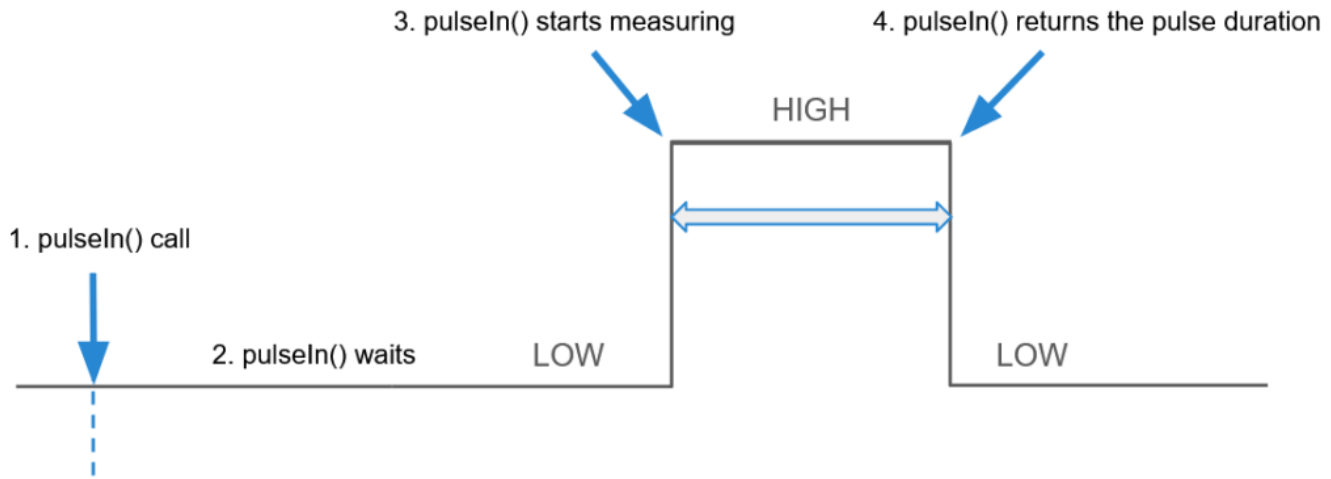


Lecture



Ultrasonic sensor

- The Arduino pulseIn() function measures the time period of a high or low pulse input signal.
- The pulseIn() function accepts as arguments the pin and the state of the pulse (either HIGH or LOW). It returns the length of the pulse in *microseconds*.
- pulseIn(pin, HIGH/LOW);
- A typical use is to measure the output from an ultrasonic distance module (HC-SR04) which outputs a signal period proportional to the round trip sound reflection time from which you can calculate distance.
- Use the pulseIn() function to get the sound wave travel time. The pulse length corresponds to the time it took to travel to the object and the time traveled on the way back.
- duration = pulseIn(echoPin, HIGH);



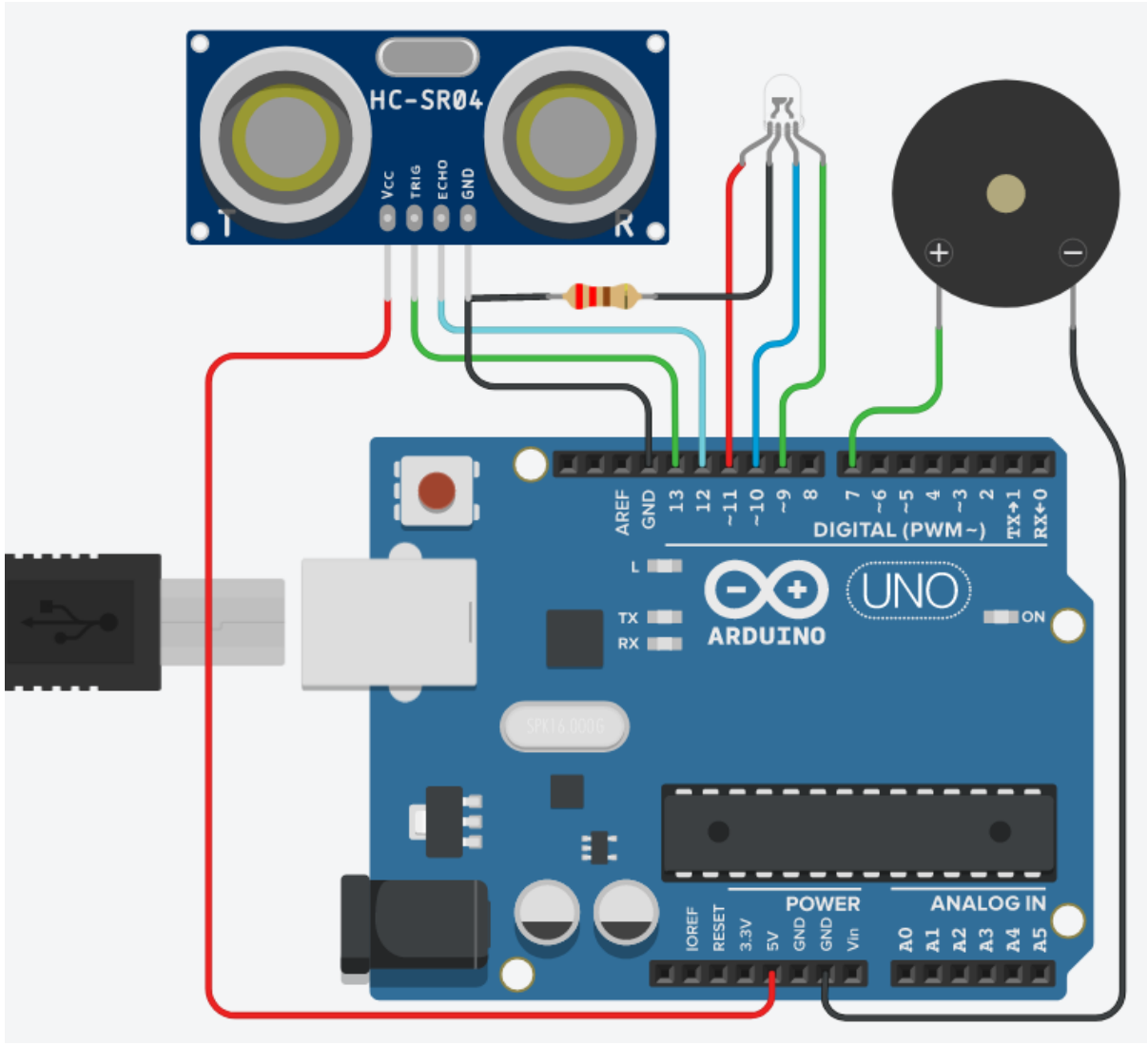


Ultrasonic demo code

Design a vehicle safety distance alarming system

- The system has 3 color led with a buzzer.
- If the distance > 100cm , all leds and buzzer are off
- If 70 <distance < 100 , led green is ON and buzz at 300 hz
- If If 50 < distance < 70 , led blue is ON and buzz at 400 hz
- If If distance < 50 , led red is ON and buzz at 500 hz







Ultrasonic demo code

```
#define led_green    9
#define led_blue   10
#define led_red      11
#define EchoPin      12
#define TriggerPin   13
#define buzzer 7
float Duration, Distance_cm;

void setup() {
  Serial.begin(9600);
  pinMode(TriggerPin, OUTPUT);
  pinMode(EchoPin, INPUT);
  pinMode(led_red, OUTPUT);
  pinMode(led_blue, OUTPUT);
  pinMode(led_green, OUTPUT);
   pinMode(buzzer, OUTPUT);
}

void loop() {
  read_distance();
  control_leds();
  show_distance();
  delay(100);
}
```

```
void control_leds() {
  if (Distance_cm >= 100) {
    digitalWrite(led_red, LOW);
    digitalWrite(led_blue, LOW);
    digitalWrite(led_green, LOW);
    noTone(buzzer);
  }
  else if (Distance_cm >= 70 && Distance_cm < 100) {
    digitalWrite(led_red, LOW);
    digitalWrite(led_blue, LOW);
    digitalWrite(led_green, HIGH);
    tone(buzzer, 300,100);
  }
  else if (Distance_cm >= 40 && Distance_cm < 70) {
    digitalWrite(led_red, LOW);
    digitalWrite(led_blue, HIGH);
    digitalWrite(led_green, LOW);
    tone(buzzer, 400,100);
  }
  else if (Distance_cm >= 0 && Distance_cm < 40) {
    digitalWrite(led_red, HIGH);
    digitalWrite(led_blue, LOW);
    digitalWrite(led_green, LOW);
    tone(buzzer, 500,100);
  }
}
```

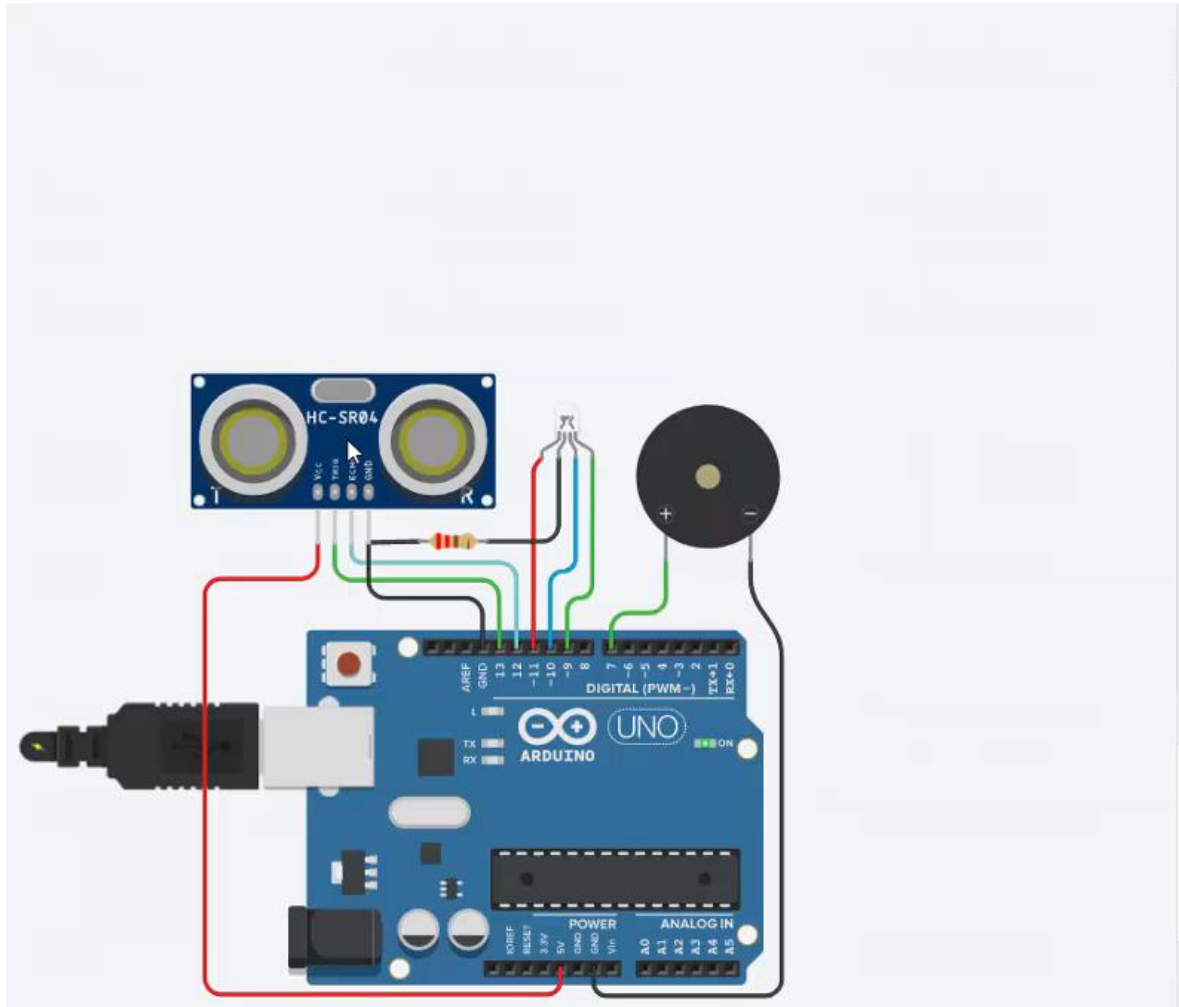


Ultrasonic demo code

```
53 void show_distance() {
54   Serial.print("Duration: "); Serial.println(Duration);
55   Serial.print("distance: ");  Serial.println(Distance_cm);
56 }
57
58 void read_distance() {
59   digitalWrite(TriggerPin, LOW);
60   delayMicroseconds(4);
61   digitalWrite(TriggerPin, HIGH);
62   delayMicroseconds(10);
63   digitalWrite(TriggerPin, LOW);
64   Duration = pulseIn(EchoPin, HIGH);
65   Distance_cm = 0.0343 * (Duration / 2.0);
66 }
```



Ultrasonic demo code




```
34      digitalWrite(led_red, LOW);
35      digitalWrite(led_blue, LOW);
36      digitalWrite(led_green, HIGH);
37      tone(buzzer, 300,100);
38    }
39    else if (Distance_cm >= 40 && Distance_cm <
40      digitalWrite(led_red, LOW);
41      digitalWrite(led_blue, HIGH);
42      digitalWrite(led_green, LOW);
43      tone(buzzer, 400,100);
44    }
45    else if (Distance_cm >= 0 && Distance_cm < 4
46      digitalWrite(led_red, HIGH);
47      digitalWrite(led_blue, LOW);
48      digitalWrite(led_green, LOW);
49      tone(buzzer, 500,100);
50    }
51  }
52
53  void show_distance() {
54    Serial.print("Duration: "); Serial.println(Du
55    Serial.print("distance: ");  Serial.println(I
56  }
```

Serial Monitor

```
Duration: 8455.00
distance: 145.00
Duration: 8455.00
distance: 145.00
Duration: 8455.00
distance: 145.00
Duration: 8467.00
distance: 145.21
```