TECHDOM

Master Arduino

Your Guide to Robotics

Lecture 1

Lecture outcomes:

Introduction to Robotics
History of Electric Circuits
How Arduino Came to Be
Arduino Board Overview
Arduino IDE
Coding Basics

Introduction to Robotics

When we hear the word "robot," various images might come to mind: humanoid machines, mechanical arms in factories, or even smart vacuum cleaners in our homes. But what exactly is a robot? Simply put, a robot is a device or machine that can perform specific tasks automatically or based on pre-programmed instructions. But what makes a robot unique and special? The secret lies in three main functions that any robot performs:

1. Sensing

Every robot begins its work by sensing the world around it. This is done using what we call "sensors." These sensors are like human senses: cameras act like eyes, microphones like ears, and temperature sensors like skin that feels heat. The role of sensors is to gather information from the surrounding environment.

2. Thinking and Decision Making

After collecting information using sensors, the "controller" comes into play. This part is the brain of the robot. The controller analyzes the data it has gathered and decides what the robot should do based on it. For example, if the robot detects an obstacle in its path, it must decide: should it stop? Should it turn? Or should it take another route?

3. Actuation

The final step is execution, where the "actuators" come into play. These parts are like the muscles of the robot. Actuators can be motors to move the robot, arms to pick up objects, or wheels for navigation. They execute the commands decided by the controller based on the collected data.

A Simple Example

Imagine a small vacuum cleaning robot:

- Sensors: Detect dirt or obstacles like walls.
- Controller: Decides where to move after analyzing the data.
- Actuators: Operate the wheels to move and the brushes to clean the floor.

In this way, Robotics is the study and design of these amazing systems that can sense, think, and act. As technology advances, robots are playing a larger role in our daily lives, from helping with household tasks to working in factories and even exploring space! In this course, we will dive deeper into how to design these systems and understand how they work in an engaging and detailed way.

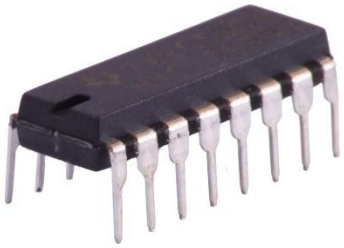
TECHDOM

History of Electric Circuits

1. Traditional Electric Circuits

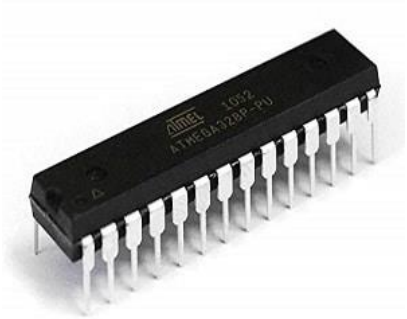
Traditional electric circuits refer to the basic electrical setups that utilize discrete components like resistors, capacitors, and transistors to perform various functions. These circuits form the foundation of early electronics, where each component had to be individually wired together, requiring significant space and manual assembly. Their simplicity makes them fundamental for understanding basic electronic principles.

2. Integrated Circuits (ICs)



Integrated circuits represent a significant advancement in electronics, where multiple components, such as transistors, resistors, and capacitors, are fabricated onto a single semiconductor chip. This miniaturization allows for more complex and reliable circuits, drastically reducing size, power consumption, and cost. ICs are the building blocks of modern electronic devices, enabling the development of sophisticated technologies.

3. Microcontrollers



Microcontrollers are compact integrated circuits designed to govern specific operations within an embedded system. Unlike general-purpose processors, microcontrollers are optimized for tasks that require direct control over hardware components, such as sensors and actuators. They are the heart of many automated systems, from simple household appliances to complex industrial machines, offering both programmability and flexibility in electronic design.

How Arduino Came to Be

Microcontrollers form the backbone of modern automation and embedded systems. However, working directly with microcontrollers can be challenging for beginners due to the complexity of their architecture and programming. This is where Arduino comes in—a user-friendly platform that simplifies working with microcontrollers. With Arduino, you can quickly prototype and build projects by combining sensors, actuators, and simple programming, making it the perfect starting point for learning about robotics and embedded systems.

ATmega 328P

Arduino UNO

For example : Arduino Uno is built around the ATmega328P microcontroller, offering a powerful and versatile platform for electronics and robotics projects. It includes additional features like a USB interface, voltage regulation, and ready-to-use I/O pins, making it easier to use for both beginners and experienced developers.

Arduino Board Overview

Arduino is an open-source electronics platform based on easy-to-use hardware and software. It consists of a microcontroller, typically the **ATmega328P** (on Arduino Uno), along with various additional components that make it easy to interact with the physical world through sensors, motors, and other electronic components.

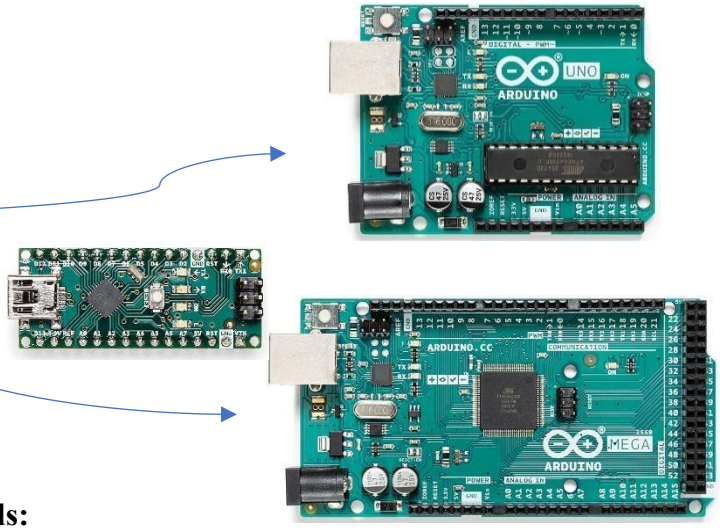
Key Features of Arduino:

1. **Open-source**: Both the hardware and software are open-source, which allows for easy customization and community collaboration.
2. **User-friendly**: Arduino is designed to be accessible to beginners, with a simple programming environment (Arduino IDE) and an extensive online community for support.
3. **Flexible Input/Output**: Arduino boards feature a variety of I/O pins (digital, analog, PWM) that make it easy to interface with sensors, actuators, and other devices.
4. **USB Interface**: The board includes a USB connection for easy programming and communication with a computer, simplifying development.
5. **Versatile Power Options**: Arduino can be powered via USB or external power sources, making it suitable for both small and large projects.
6. **Wide Community Support**: With a large, active community, there are countless tutorials, libraries, and resources available to help with every project, from simple to complex.

Arduino boards are ideal for a variety of applications, including robotics, home automation, wearable devices, and more, offering a solid platform for both educational purposes and professional prototyping.

Arduino Board Types:

- **Arduino Uno**
- **Arduino Nano**
- **Arduino Mega**
- Arduino Mini
- Arduino Lilypad
- Arduino Duemilanove
- Boarduino

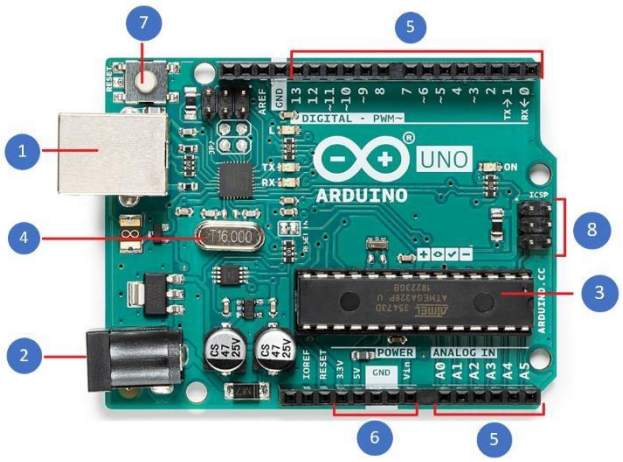


The Differences Between Arduino Boards:

The main differences between various Arduino boards can be summarized as follows:

1. **Number of I/O Pins**: Each board offers a different number of digital and analog input/output pins, catering to projects of varying complexity.
2. **Microcontroller Used**: Different boards are built around different microcontrollers (e.g., ATmega328P, SAMD21), which affect their capabilities and features.
3. **Processor Speed**: The clock speed of the processor varies across boards, impacting the processing power and performance.

Arduino UNO



1. **USB Port**: Used for programming the board and providing power when connected to a computer.
2. **External Power**: Allows the board to be powered using an external power source, such as a battery or adapter.
3. **Microcontroller (ATmega328P)**: The brain of the board, responsible for executing the uploaded code.
4. **Crystal Oscillator (16 MHz)**: Provides the clock signal to keep the microcontroller running at the correct speed.
5. **I/O Pins (Digital, Analog, PWM)**: Interface points for connecting sensors, actuators, and other devices.
6. **Power Pins (5V, 3.3V, GND)**: Provide power to external components connected to the board.
7. **Reset Button**: Resets the microcontroller, restarting the execution of the uploaded program.
8. **ICSP Pins**: Used for low-level programming and firmware updates of the microcontroller.

Source of Power

The Arduino Uno can be powered through three sources:

1. **USB**:
   Power is supplied through the USB cable, typically connected to a computer or adapter, which also allows code to be uploaded to the board.
2. **External Power**:
   Power can be supplied using an external source such as a battery or power bank.
3. **Vin Pin**:
   Power is provided by connecting a battery's positive terminal to the **Vin pin** and the negative terminal to the **GND pin**.

Power Pins

1. **Vin**:
   This pin is used as a source of power, allowing the board to be powered from an external source.
2. **5V**:
   This pin provides a 5-volt power source for connected components.
3. **3.3V**:
   This pin provides a 3.3-volt power source for low-voltage components.
4. **GND**:
   This pin acts as the ground connection for the circuit.

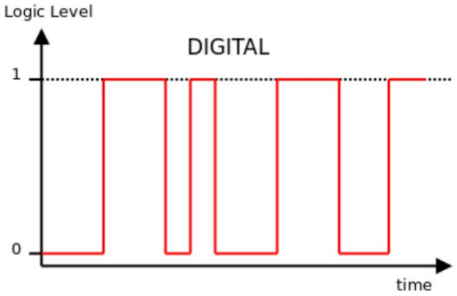
Input and Output Pins

The Arduino has **20 I/O pins**, divided as follows:

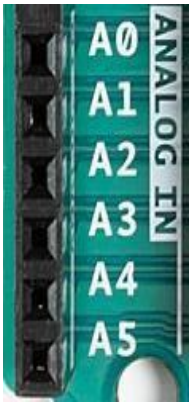
1. Digital Pins (14 pins)

- **Definition**:
  Digital pins allow the Arduino to interact with the external world by reading or sending digital signals.
- **Input Mode**:
  When configured as input, digital pins detect voltage to read the state of devices like buttons or sensors.
- **Output Mode**:
  When configured as output, digital pins send voltage signals to control devices such as LEDs, relays, or motors.
- **Operating States**:
  Digital pins operate in binary mode:
  - **HIGH (on)**: Voltage is present.
  - **LOW (off)**: Voltage is absent.

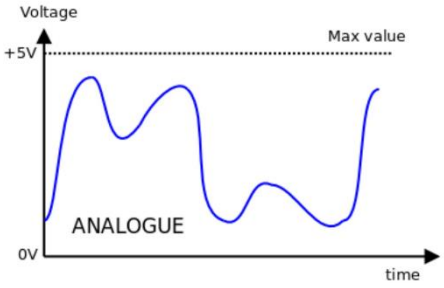
- **PWM Support**:
  Some digital pins support **Pulse Width Modulation (PWM)**, providing analog-style output signals for controlling devices like motors or dimmable LEDs.
- **Importance**:
  Digital pins are fundamental for creating interactive and flexible Arduino projects, enabling a wide range of applications.



2. **Analog Pins** (6 pins)



- **Definition**:
  Analog pins are designed to read continuous (analog) signals from sensors and other input devices.
- **Input Mode**:
  Analog pins read varying voltage levels from external sources, converting these signals into digital data for processing.
- **ADC (Analog-to-Digital Converter)**:
  Each analog pin connects to an ADC (Analog-to-Digital Converter) within the Arduino.
  - The ADC converts input voltage (0V to 5V) into digital values ranging from **0 to 1023**, offering a 10-bit resolution.
- **Output Mode**:
  Analog pins can also function as digital outputs, though they won't produce true analog signals in this mode.
- **Importance**:
  Analog pins are crucial for reading real-world data, such as temperature, light, or sound, making them essential for sensor-based and environmental monitoring projects.

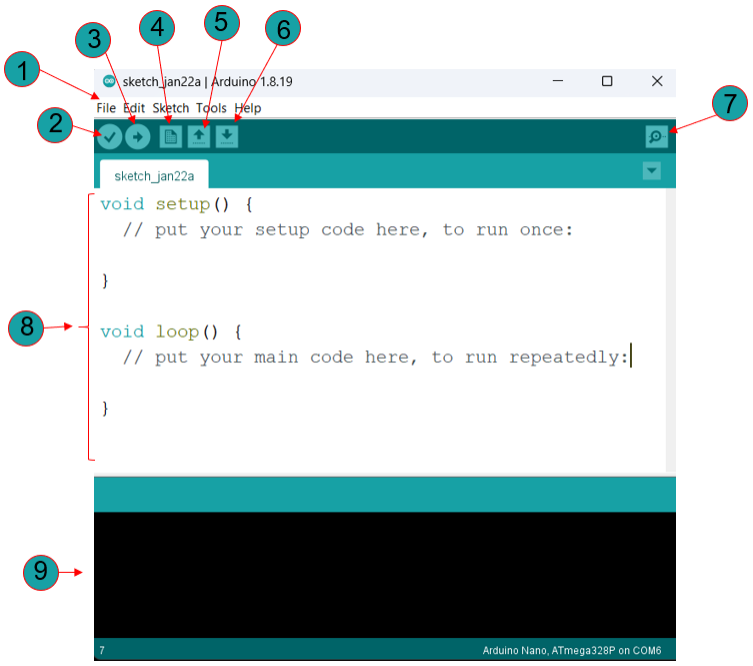


Arduino IDE

The **Arduino IDE** (Integrated Development Environment) is the software used to write, upload, and test programs on Arduino boards.

Why Use Arduino IDE?

- **Easy to Use**: Simple interface for beginners to write and edit code (called "sketches").
- **Built-in Tools**: Comes with pre-installed libraries and examples to help you get started quickly.
- **Serial Monitor**: Lets you see data from your Arduino board in real time for debugging.
- **One-Click Upload**: Easily upload your code to the board with a single button.
- **Supports Multiple Boards**: Works with most Arduino models, making it versatile.



1. **Menu Bar**:
   This is the topmost bar containing menus such as:
   - **File**: Used for creating, saving, or opening sketches (projects).
   - **Edit**: Provides options to cut, copy, paste, and edit code.
   - **Sketch**: Includes tools to verify, compile, and manage libraries.
   - **Tools**: Allows configuration of board settings, port selection, and programmer options.
   - **Help**: Offers documentation and support for Arduino IDE.

2. **Verify Button (✓)**:
   - Checks the code for errors and compiles it without uploading to the board.
   - Useful for debugging before finalizing the upload.

3. **Upload Button (→)**:
    - Compiles the code and uploads it to the connected Arduino board.
    - The board executes the uploaded program immediately after a successful upload.

4. **New**: Creates a blank sketch to start a new project.

5. **Open**: Allows you to open an existing sketch saved on your computer.

6. **Save**: Saves the current sketch to your local storage.

7. **Serial Monitor Button**:
    - Opens the Serial Monitor window, which displays real-time data sent from the Arduino board.
    - Also allows you to send data back to the board for communication.

8. **Code Editor**:
    - The main workspace for writing and editing Arduino sketches.
    - Includes features like syntax highlighting and auto-indentation for better readability.

9. **Output Console**:
    - Displays messages from the IDE during operations such as:
        - Compilation results.
        - Errors and warnings.
        - Successful upload notifications.
    - It helps users debug their code and monitor progress during the upload process.

Coding

Variables

Variables are named storage locations in a program that hold data values. They allow you to store, retrieve, and manipulate data during program execution.

For example:

X = 20 → This means we have stored the value 20 in the variable X. Whenever we need to use the value 20, we can simply refer to X instead.

Variable naming rules

1. **Variable Can Contain**:
   - Capital Letters: **A to Z**
   - Small Letters: **a to z**
   - Numbers: **0 to 9**
   - Underscore: **_**
2. **First Character**:
   - Must be an **alphabet** (A-Z or a-z) or an **underscore (_)**.
3. **Blanks and Commas**:
   - **Not Allowed** in variable names.
4. **Special Symbols**:
   - Only **underscore (_) is allowed**.
   - Symbols like ?, #, etc., are **not allowed**.
5. **Reserved Words**:
   - Variable names **cannot use reserved words**, such as int, float, if, etc.
6. **Uniqueness in Scope**:
   - Variable names **cannot be repeated** in the same scope.
   - (This will be clarified in further discussions).

Variables have different types, which depend on the kind of data they store and the amount of memory required. This concept is known as data types

Data Types

1. **void**
    a. Represents no data type, used to indicate that a function does not return any value.
2. **boolean**
    a. Represents a logical value, either **true** or **false**.
    b. **Size**: 1 byte.
3. **char**
    a. Represents a single character.
    b. **Size**: 1 byte.
    c. **Range**: -128 to 127.
4. **byte**
    a. An unsigned data type representing a small integer.
    b. **Size**: 1 byte.
    c. **Range**: 0 to 255.
5. **int**
    a. Represents an integer.
    b. **Size**: 2 bytes.
    c. **Range**: -32,768 to 32,767.
6. **unsigned int**
    a. An unsigned data type representing an integer.
    b. **Size**: 2 bytes.
    c. **Range**: 0 to 65,535.
7. **word**
    a. An unsigned data type representing an integer.
    b. **Size**: 2 bytes.
    c. **Range**: 0 to 65,535.
    d. **Note**: A word is the same as an unsigned int on most platforms.
8. **long**
    a. Represents a large integer.
    b. **Size**: 4 bytes.
    c. **Range**: -2,147,483,648 to 2,147,483,647.
9. **unsigned long**
    a. An unsigned data type representing a large integer.
    b. **Size**: 4 bytes.
    c. **Range**: 0 to 4,294,967,295.
10. **short**
    a. Represents a small integer.
    b. **Size**: 2 bytes.
    c. **Range**: -32,768 to 32,767.
11. **float**
    a. Represents a floating-point number.
    b. **Size**: 4 bytes.
    c. **Range**: ±3.4028235E+38.
12. **double**
    a. A floating-point data type with double precision.
    b. **Size**: 4 bytes.
    c. **Range**: Same as float on most standard Arduino boards.

Conditional Statements

- **if**
- **if ... else**
- **if ... else if ... else**
- **switch**

1. [if] Statement

Syntax:

```
if (condition)
{
  // statements to execute if condition is true
}
```

- If the condition is true, the statement will be executed.
- If the condition is false, the statement will not be executed.

Example:

Assume that **x** represents the temperature value. The condition checks if the temperature is greater than or equal to 30, in which case the fan will be turned on.

```
if (x >= 30)
{
  digitalWrite(fan,HIGH);
}
```

2. [if ... else] Statement

Syntax:

```
if (condition)
{
  // statements to execute if condition is true
}
else
{
 // statements to execute if condition is false
}
```

- If the condition is true, **statement 1** will be executed.
- If the condition is false, **statement 2** will be executed.

Example:

Assume the same condition as the previous example, but this time, if the temperature is less than 30, the fan will be turned off.

```
if (x >= 30)
{
  digitalWrite(fan,HIGH);
}
else
{
  digitalWrite(fan,LOW);
}
```

3. [if ... else if ... else] Statement

Syntax:

```
if (condition)
{
 // statements to execute if condition 1 is true
}
else if
{
 // statements to execute if condition2 is true
}
else
{
 // statements to execute if both conditions are false
}
```

- If **condition 1** is true, **statement 1** will be executed.
- If **condition 2** is true, **statement 2** will be executed.
- If all previous conditions are false, **statement 3** will be executed.

Example:

In this example, we will read a value from a potentiometer connected to an analog pin. Based on the reading, we will light up different LEDs to indicate the range of the value.

```
if (sensorValue >= 0 && sensorValue <= 341) {
    digitalWrite(ledRed, HIGH);
    digitalWrite(ledBlue, LOW);
    digitalWrite(ledGreen, LOW);
} else if (sensorValue >= 342 && sensorValue <= 682) {
    digitalWrite(ledRed, LOW);
    digitalWrite(ledBlue, HIGH);
    digitalWrite(ledGreen, LOW);
} else {
    digitalWrite(ledRed, LOW);
    digitalWrite(ledBlue, LOW);
    digitalWrite(ledGreen, HIGH);
}
```

TECHDOM

Master Arduino

Your Guide to Robotics

Lecture 2

Lecture outcomes:

Switch statement
Loops
Arrays
Functions

4. [switch ... case] Statement

Syntax:

```
switch (variable) {
  case value1:
   // statements to execute if var equals value1
    break;
  case value2:
   // statements to execute if var equals value2
    break;
  default:
   // statements to execute if var doesn't match any case
    break;
}
```

- If the variable equals **value 1**, **statement 1** will be executed.
- If the variable equals **value 2**, **statement 2** will be executed.
- If the variable doesn't match any case, **statement 3** will be executed (the **default** case).

Example:
In this example, we will read a value from a potentiometer connected to an analog pin, map this value to a range of 0 to 3, and light up different LEDs according to the mapped value.

```
switch (mappedValue) {
      case 0:
          digitalWrite(ledRed, LOW);
          digitalWrite(ledBlue, LOW);
          digitalWrite(ledGreen, LOW);
          break;
      case 1:
          digitalWrite(ledRed, HIGH);
          digitalWrite(ledBlue, LOW);
          digitalWrite(ledGreen, LOW);
          break;
```

```
        case 2:
             digitalWrite(ledRed, HIGH);
             digitalWrite(ledBlue, HIGH);
             digitalWrite(ledGreen, LOW);
             break;
         case 3:
             digitalWrite(ledRed, HIGH);
             digitalWrite(ledBlue, HIGH);
             digitalWrite(ledGreen, HIGH);
             break;
         //default :
             //leaved empty
}
```

Note:

The break statement is essential in switch statements to ensure that once a specific case is executed, the control flow exits the switch block and does not continue executing the subsequent cases.

Switch Statement Rules:

- Case constants must be unique.
- Case constants cannot be variables.
- Case constants must be an integral value.
- Only one **default** case is allowed.
- **Default** is optional and can be placed anywhere in the switch.
- **Break** statements end the switch.
- Once a match occurs, all following case sections are executed.
- If the **break** statement does not exist, all the following code will be executed until a **break** is found or the switch ends.
- Nesting in switch statements is allowed.

Loops

1. **for loop**
2. **while loop**
3. **do-while loop**

1. for Loop

Syntax:

```
for (initialization; condition; increment) {
  // statements to execute in the loop
}
```

- Used to iterate a specific number of times.
- The initialization statement is executed first, followed by a condition check.
- If the condition is true, the body of the loop is executed.
- After the body runs, the action statement executes, and the condition is checked again. This process continues.

Example:
In this example, we will use a for loop to sequentially turn five LEDs connected to digital pins on and off.

```
for (int i = 0; i < 5; i++)
{
  digitalWrite(ledPins[i], HIGH);
  delay(500);
  digitalWrite(ledPins[i], LOW);
}
```

2. while Loop

Syntax:

```
while (condition) {
  // statements to execute while condition is true
}
```

Repeatedly executes the code as long as the condition remains true.

If the condition is true, the loop's body will be executed and will continue repeating until the condition becomes false.

The condition is checked at the beginning of each iteration.

Also known as an event-driven loop.

Example:
In this example, we will use a while loop to blink an LED connected to a digital pin, and the blinking will continue until a button connected to another digital pin is pressed.

```
while(buttonState == LOW){
  digitalWrite(ledGreen, HIGH);
  delay(500);
  digitalWrite(ledGreen, LOW);
  delay(500);
  buttonState = digitalRead(buttonPin);
}
```

3. do-while Loop

Syntax:

```
do {
  // statements to execute while condition is true
} while (condition)
```

- The body is executed at least once, regardless of the condition.
- The body runs first, and then the condition is checked.
- If a variable is declared inside the loop's body, it will not be accessible in the condition.

Example:

In this example, we will use a `while` loop to blink an LED connected to a digital pin, and the blinking will stop when a button connected to another digital pin is pressed.

```
do{
   digitalWrite(ledGreen, HIGH);
   delay(500);
   digitalWrite(ledGreen, LOW);
   delay(500);
   buttonState = digitalRead(buttonPin);
}while(buttonState == LOW);
```

Arrays

An array is a collection of data that holds a fixed number of values, all of which are of the same type.

```
Array_Type Array_Name [Length] ;
```

- The array index always starts at **0**, meaning an array with 5 elements has indices ranging from 0 to 4.
- The array length must be a constant value and cannot be a variable.

Arrays Initialization:

An array can be initialized at the time of definition. The syntax for initialization is as follows:

Syntax:

```
Array_Type Array_Name[Length] = { value1, value2, ... };
```

Special Case:

If the array is initialized with fewer values than its length, the remaining elements will automatically be set to **0**.

Accessing Array Elements:

All elements of an array can be accessed at the same time only during initialization. After initialization, array elements can only be accessed one at a time.

Syntax:

```
Array_Name [Element_Index]
```

Functions

A function is a block of code that can be called from different parts of the program. Functions help organize the code and make it easier to understand. In Arduino, there are two main types of functions: built-in functions and user-defined functions.

Types of Functions in Arduino:

1. **The `setup()` Function:**
   - It is used to execute code only once at the start of the program.
   - Typically used to initialize settings like input and output pins.

   Syntax:

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

}
```

   - The code inside `setup()` is executed only when the program starts or when the board is reset.

2. **The `loop()` Function:**
   - This function runs continuously after the `setup()` function has finished.
   - It is used to execute the code that should run repeatedly throughout the program.

   Syntax:

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

}
```

   - Once the code inside `loop()` is finished, it starts again from the top, running continuously.

Defining Your Own Function (User-defined Function):

You can define your own function in Arduino to perform specific tasks separately from the built-in functions. User-defined functions are useful for reducing code repetition and making your program more organized.

Syntax:

```
return_type Function_name(parameters)
{
    body;
}
```

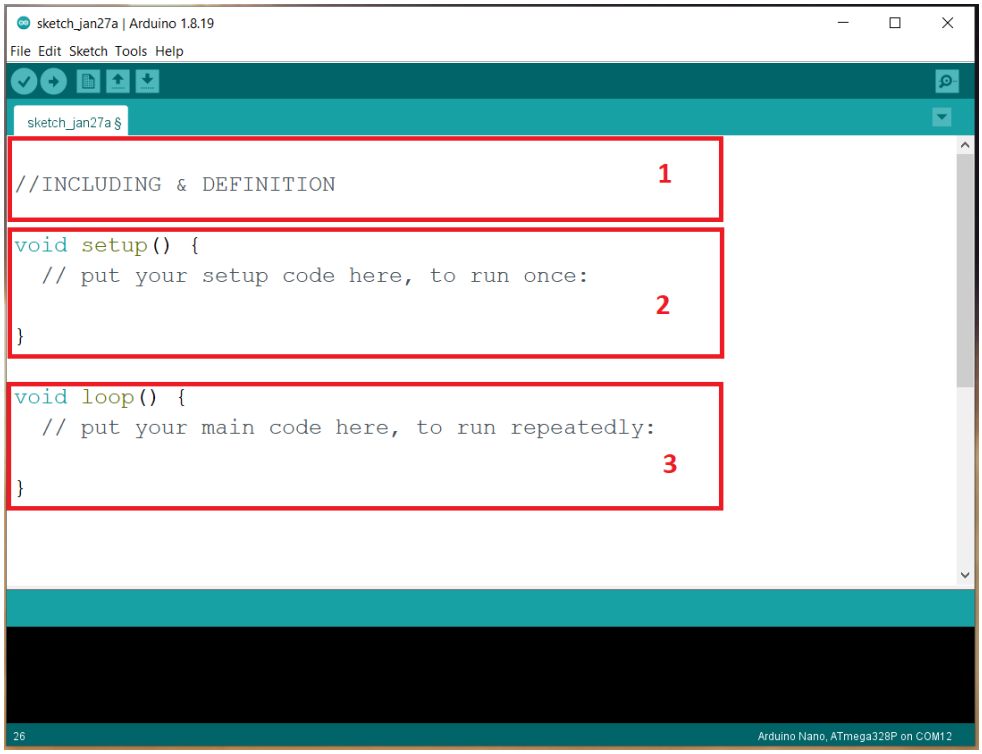
- **Return_type**: The type of data the function will return (e.g., `int`, `void` if the function doesn't return any value).
- **Function_name**: The name of the function.
- **parameters**: The parameters the function accepts (if any).
- **body**: the function code and the value the function returns (if it returns a value).

Notes:

- The `setup()` and `loop()` functions are essential in every Arduino program.
- You can define your own functions to organize your code better, especially for larger programs.
- Functions can accept parameters and return values as needed.

Coding Structure in Arduino

1. **Library Inclusions and Definitions**.
2. **The `void setup()` Function**.
3. **The `void loop()` Function**.



Library Inclusions and Definitions

This part of the code is where you prepare everything the program needs to work. It includes:

1. **Including Libraries**:
   - Libraries are like tools that provide extra functionality for your Arduino program.
   - For example, if you are working with sensors or displays, you may need to include a library that provides specific functions for those components.
   - To include a library, use the `#include` command.

Example:

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

2. **Defining Variables and Constants**:
    - In this step, you set up important values that will be used throughout your program.
    - Variables can store data that might change during the program. Constants store fixed values that won't change.
    - You can also use definitions to give meaningful names to pins or settings.

    Example:

```
#define LED_PIN 13 //Pin number for the LED
int sensorValue = 0; //variable to store sensor reading
```

The `void setup()` Function

The `void setup()` function is one of the most important parts of any Arduino program. It is used to initialize the settings that the program needs before starting its main operations. This function runs only once when the Arduino board is powered on or reset.

Here's what typically happens in the `setup()` function:

1. **Setting Up Pin Modes**:
    - You can configure pins as either **input** (to read data, e.g., from sensors) or **output** (to control components, e.g., LEDs or motors).
    - Use the `pinMode()` function to set the mode for each pin.

    Example:

```
void setup() {
  pinMode(LED_PIN, OUTPUT);
  pinMode(Sensor_PIN, INPUT);
}
```

2. **Initializing Serial Communication**:
   - If your program needs to communicate with your computer, you can set up the Serial Monitor using `Serial.begin(baudRate)`.
   - The `baudRate` is the speed of communication (e.g., 9600 bits per second).

   Example:

```
void setup() {
  Serial.begin(9600); //start serial communication
}
```

3. **Starting Components or Libraries**:
   - Some components, like displays or motors, may need to be initialized in the `setup()` function before they can be used.

   Example:

```
void setup() {
  Servo.attach(9);  //attach a servo motor to pin 9
  lcd.begin(16, 2); //Initialize a 16x2 LCD display
}
```

Key Notes:

- The `void setup()` function is executed only once, so it's the best place to set up everything your program needs to work correctly.
- After the `setup()` function finishes, the Arduino will automatically move on to the `loop()` function.

The `void loop()` Function

The `void loop()` function is the heart of any Arduino program. After the `setup()` function finishes executing, the Arduino board moves on to the `loop()` function, where the main logic of the program runs continuously.

Here's what typically happens in the `loop()` function:

1. **Repeating Tasks**:
   - The code inside the `loop()` function keeps running over and over, allowing you to perform repeated actions like reading sensors, turning on/off components, or sending data.

   Example:

```
void loop() {
   int sensorValue = analogRead(A0); //read a sensor value
   digitalWrite(LED_PIN, HIGH);      //turn on the LED
   delay(1000);                      //wait for a second
   digitalWrite(LED_PIN, LOW);       //turn off the LED
   delay(1000);                      //wait for 1 second
}
```

2. **Making Decisions**:
   - You can use conditional statements (e.g., `if`, `else`) in the `loop()` function to make decisions based on sensor readings or other conditions.

   Example:

```
void loop() {
  if(sensorValue > 500){
    digitalWrite(LED_PIN, HIGH);
   }
else{
    digitalWrite(LED_PIN, LOW);
  }
}
```

3. **Continuous Monitoring**:
    - The `loop()` function allows you to continuously monitor inputs like buttons, sensors, or communication data.

Example:

```
void loop() {
  if(digitalRead(BUTTON_PIN) == HIGH){
    Serial.print("Button Pressed !");
    }
}
```

Key Notes:

- Unlike the `setup()` function, the `loop()` function runs **indefinitely** as long as the Arduino is powered on.
- It's the ideal place for running tasks that need to happen repeatedly or continuously.
- You can think of the `loop()` function as a cycle that keeps the program running and responding to real-time changes.

Basic I/O Functions in Arduino

Arduino provides several essential functions for working with digital and analog pins. These functions allow you to read input from sensors or buttons and control outputs like LEDs, motors, or other components. Below is an explanation of the most commonly used functions:

1. `digitalRead()`

This function is used to read the state of a digital pin (HIGH or LOW).

- **Use Case**: Detect the status of a button, switch, or other digital input device.
- **Syntax**:

```
digitalRead(pin);
```

  - **pin**: The digital pin you want to read from.
  - **Returns**: `HIGH` (if the pin is receiving 5V) or `LOW` (if it is receiving 0V).

Example:

```
int buttonState;
void setup() {
  pinMode(2 ,INPUT); //set pin 2 as input
}

void loop() {
  buttonState = digitalRead(2); //reaad the state of pin 2
  if(buttonState == HIGH){
    Serial.println("Button Pressed !");
  }
  else{
    Serial.println("Button Released !");
  }
}
```

2. `digitalWrite()`

This function is used to set a digital pin to either HIGH (5V) or LOW (0V).

- **Use Case**: Turn on/off an LED, activate a relay, or control any digital output.
- **Syntax**:

```
digitalWrite(pin, value);
```

  - **pin**: The digital pin you want to control.
  - **value**: `HIGH` or `LOW`.

Example:

```
void setup() {
  pinMode(13, OUTPUT); //set pin 13 as output
}


void loop() {
   digitalWrite(13, HIGH);
   delay(1000);
   digitalWrite(13, LOW);
   delay(1000);
}
```

3. `analogRead()`

This function is used to read the value from an analog pin (values range from 0 to 1023).

- **Use Case**: Read sensor values, such as from a potentiometer or light sensor.
- **Syntax**:

```
int value = analogRead(pin);
```

  - **pin**: The analog pin (A0 to A5) you want to read from.
  - **Returns**: A value between 0 (0V) and 1023 (5V).

Example:

```
#define SENSOR_PIN A0

int sensorValue;

void setup() {
  Serial.begin(9600);
  pinMode(SENSOR_PIN, INPUT);
}

void loop() {
  sensorValue = analogRead(SENSOR_PIN);
  Serial.println(sensorValue);
  delay(500);
}
```

4. `analogWrite()`

This function is used to write an analog (PWM) value to a pin (values range from 0 to 255).

- **Use Case**: Control the brightness of an LED or the speed of a motor.
- **Syntax**:

```
analogWrite(pin,value);
```

  - **pin**: The digital pin that supports PWM (pins marked with ~ on most Arduino boards).
  - **value**: A PWM value between 0 (0% duty cycle, fully off) and 255 (100% duty cycle, fully on).

Example:

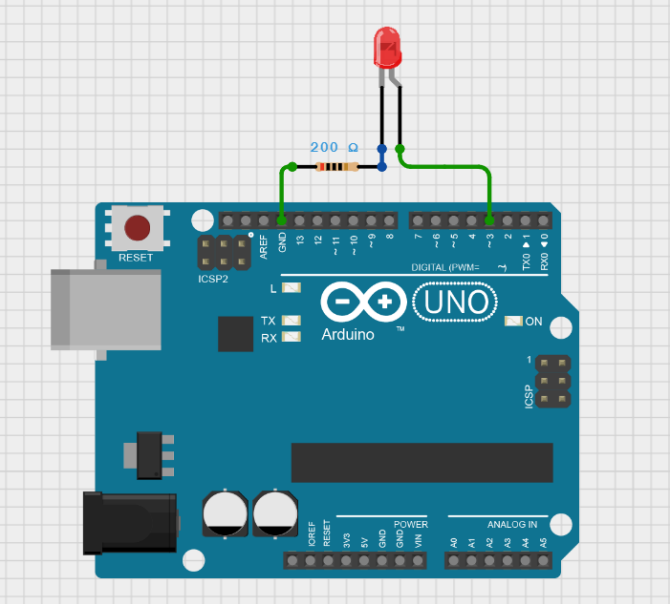
```
void setup() {
  pinMode(9, OUTPUT); //set pin 9 as output
}

void loop() {
  for(int i = 0 ; i <= 255 ; i++){
    analogWrite(9, i);
    delay(10);
  }
  for(int i = 255 ; i >=0 ; i--){
    analogWrite(9, i);
    delay(10);
  }
}
```

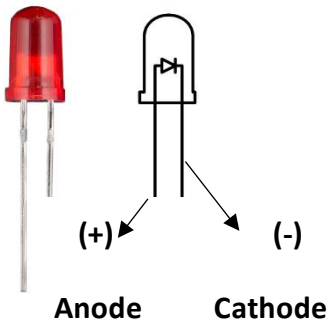
Key Notes:

- `digitalRead()` and `digitalWrite()` are used for **digital pins**, which can only have two states: HIGH or LOW.
- `analogRead()` is used for **analog pins**, which return a range of values (0–1023).
- `analogWrite()` works with PWM-enabled **digital pins** to simulate analog output.

Hardware:

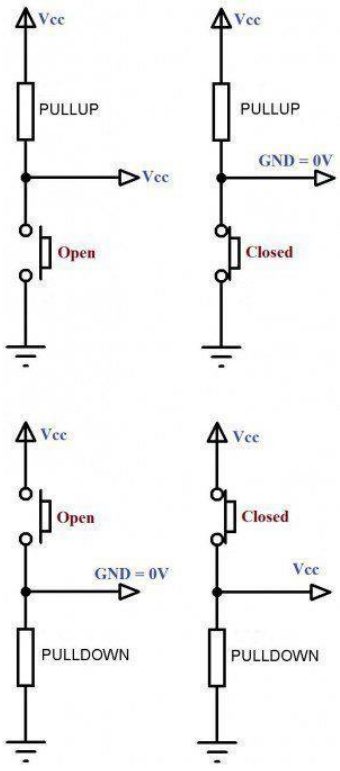
LEDs:

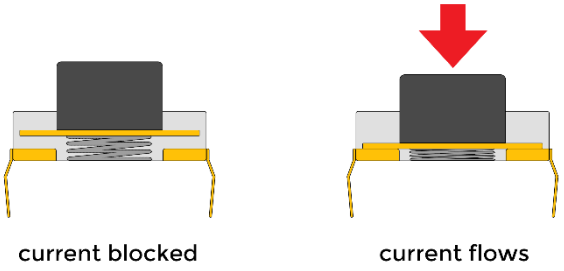


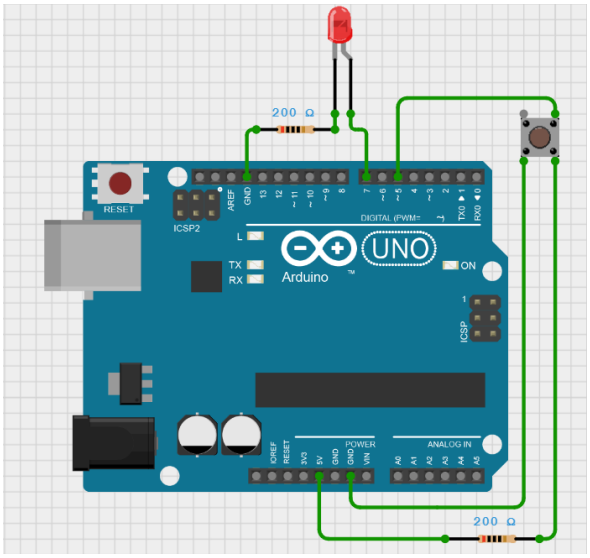



```
#define LED 3 //define the LED pin as pin 3

void setup(){
  pinMode(LED, OUTPUT); //set pin 3(LED) as an output
}

void loop(){
  digitalWrite(LED, HIGH); //Turn the LED on
  delay(2000);             //wait for 2 seconds
  digitalWrite(LED, LOW);  //Turn the LED off
  delay(1000);             //wait for 2 seconds
}
```

Pushbutton:

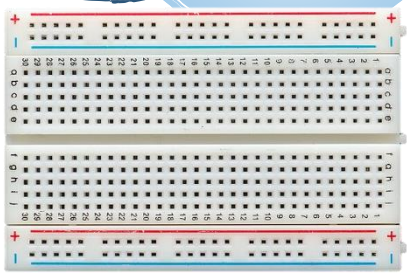






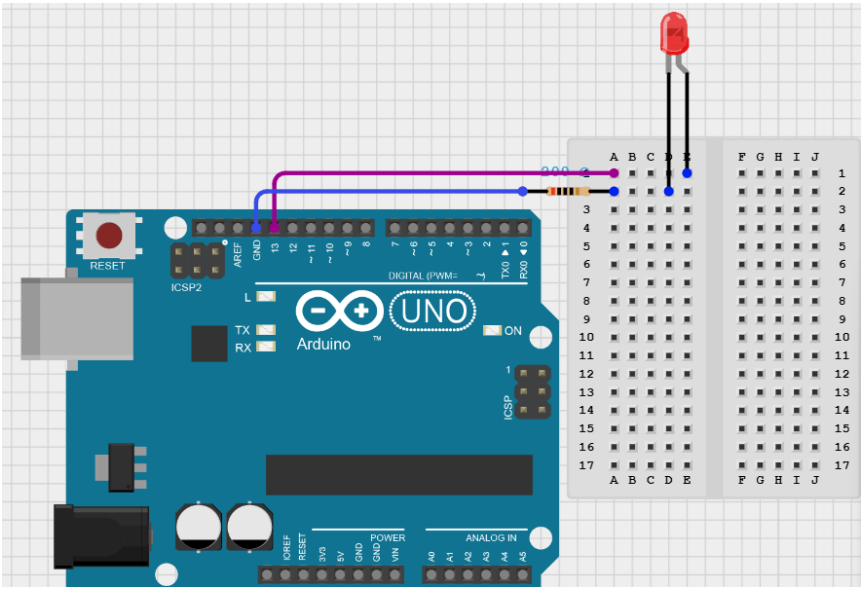

```
#define LED 7 //define the LED pin as pin 7
#define PB 5 //define the push button pin as pin 5

int PB_Value;

void setup(){
  pinMode(LED, OUTPUT);      //set pin 7(LED) as an output pin
  pinMode(PB, INPUT_PULLUP);//set pin 5(push button) as an input
                            //pin with the internal pull-up
                            //resistor enabled
}

void loop(){
  PB_Value = digitalRead(PB);

  if(PB_Value == LOW){
    digitalWrite(LED, HIGH);
  }
  else{
    digitalWrite(LED, LOW);
  }
}
```

Breadboard:

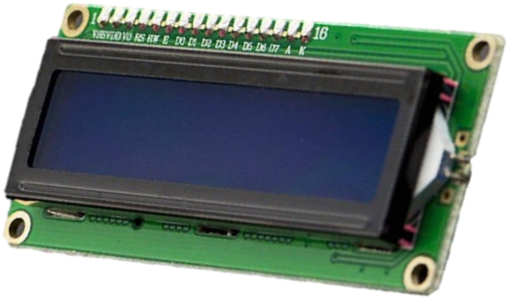


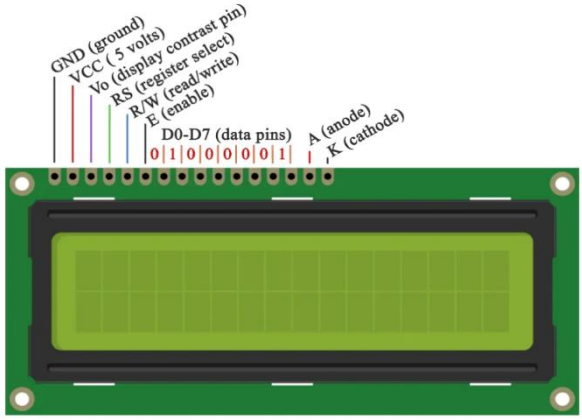
It’s used for building and testing electronic circuits without soldering. Here's an explanation of its design and functionality without referring to the letters or labels:

1. **Horizontal rows at the top and bottom:**
   - The board has two horizontal rows at both the top and bottom.
   - Each row consists of holes that are electrically connected horizontally.
   - These rows are usually used as power rails: one for the positive voltage (e.g., +5V) and the other for the ground (GND).
2. **Vertical columns in the central area:**
   - The central section contains holes arranged in vertical columns.
   - Each column consists of holes that are electrically connected vertically.
   - These columns are used to connect components like resistors, capacitors, or wires together in a circuit.
3. **The middle gap:**
   - The gap in the middle separates the top and bottom sections of the central area.
   - This separation ensures that the columns on each side are not electrically connected.
   - It's often used to place components with multiple legs, like integrated circuits (ICs), so their pins can connect to both sides.
4. **Electrical connections:**
   - In the horizontal rows, holes are connected horizontally, making them ideal for power distribution.
   - In the vertical columns, holes are connected vertically, allowing components to connect within the circuit.



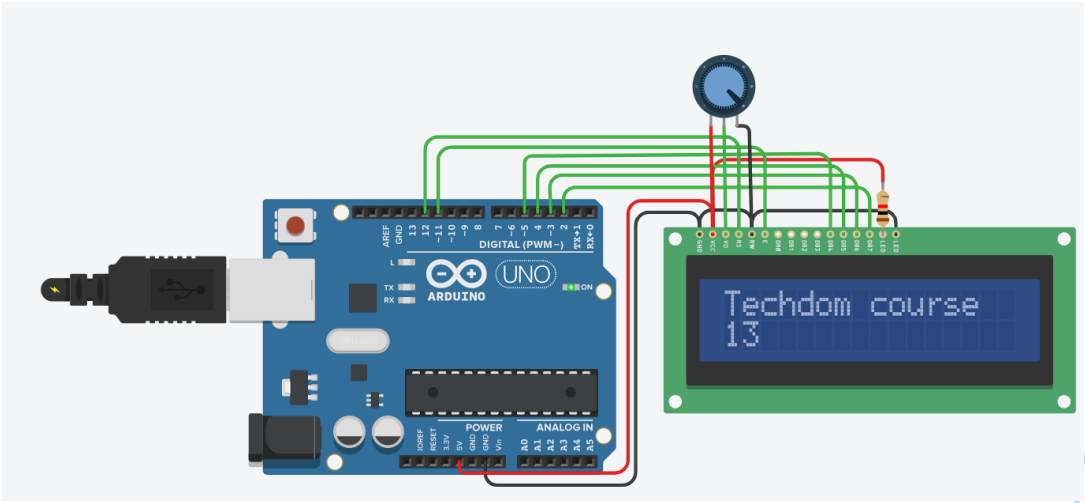
LCD:



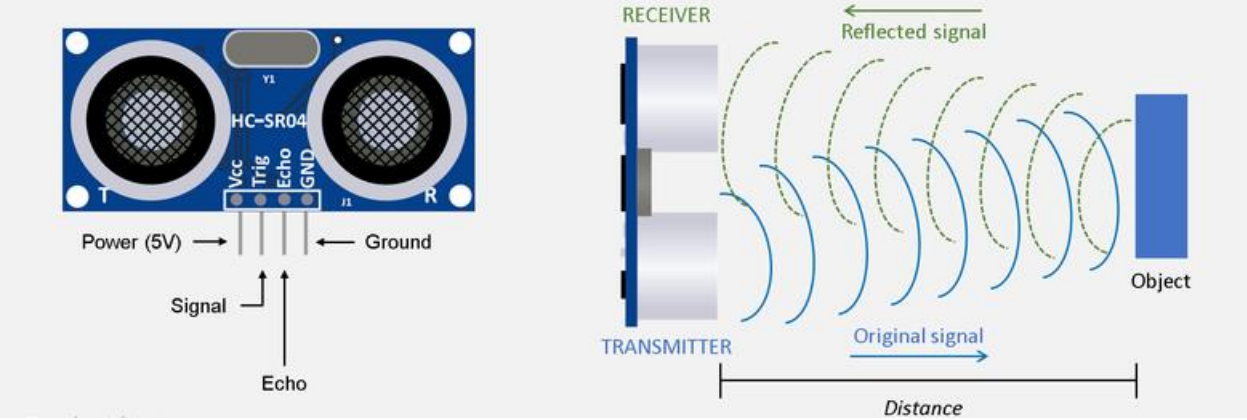



```
#include <LiquidCrystal.h> // Include the LiquidCrystal library to control
the LCD display

// Define the pins for connecting the LCD to the Arduino
const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;

// Create an instance of the LiquidCrystal class with the specified pins
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);

int count = 0; //Initialize a counter to keep track of the numbers displayed

void setup() {
  lcd.begin(16, 2); // Initialize the LCD with 16 columns and 2 rows
  lcd.print("Techdom course"); // Print "Techdom course" on the first line
}

void loop() {
  lcd.setCursor(0, 1); //Set the cursor to the beginning of the second line
                       //(column 0, row 1)
  lcd.print(count);    //Print the value of the counter on the second line
  delay(1000);         // Wait for 1 second before continuing
  count++;             // Increment the counter by 1
}
```

Simulation :



Ultrasonic sensor:



The ultrasonic sensor, as shown in the image, is used to measure the distance between the sensor and an object by emitting and detecting ultrasonic waves. Here's a detailed explanation of how it works:

Components:

1. **The Sensor :**
   - **Trig Pin (Trigger):** Used to send a signal to initiate the measurement process.
   - **Echo Pin (Echo):** Used to measure the time it takes for the reflected signal to return.
   - **Vcc:** Power supply (5V).
   - **GND:** Ground connection.
2. **Internal Parts:**
   - **Transmitter:** Emits ultrasonic waves (high-frequency sound waves).
   - **Receiver:** Detects the ultrasonic waves that are reflected back from the object.

How It Works:

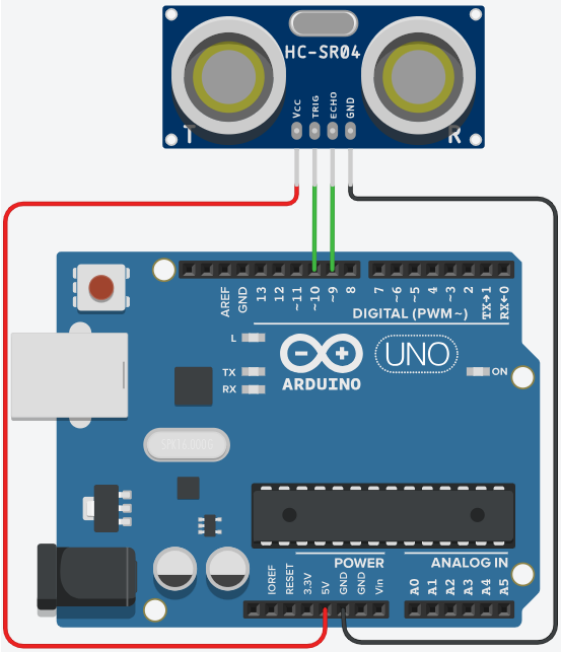
1. **Sending the Ultrasonic Wave:**
   - When a pulse signal (10 microseconds) is sent to the **Trigger Pin**, the sensor emits an ultrasonic wave at a frequency of 40 kHz.
   - This wave travels through the air at the speed of sound (approximately 343 m/s at room temperature).
2. **Wave Reflection:**
   - When the ultrasonic wave hits an object, it reflects back toward the sensor.
3. **Receiving the Reflected Wave:**
   - The receiver detects the reflected wave, and the sensor measures the time it took for the wave to travel to the object and return.
4. **Calculating Distance:**
   - The distance to the object is calculated using the formula:

Distance = Speed of Sound × Time Taken / 2

343 m/s in

The time for the wave to travel to the object and back (measured in

Schematic :

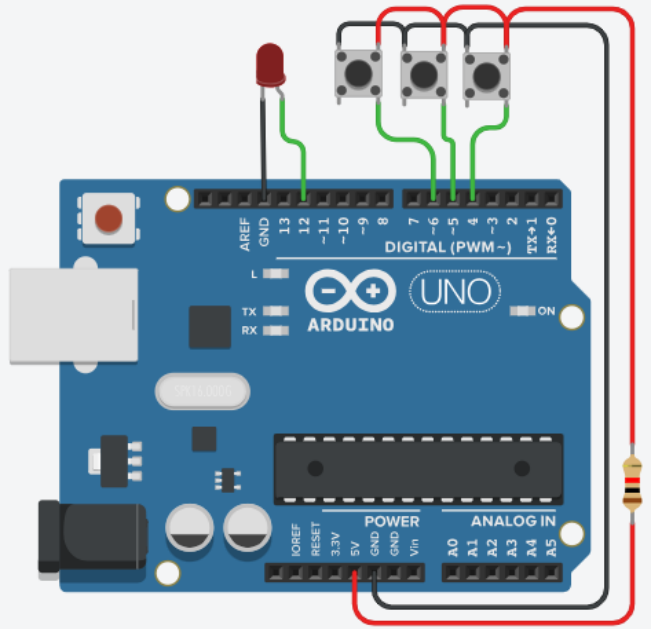



```
const int trigPin = 10; // Pin connected to Trig
const int echoPin = 9; // Pin connected to Echo

void setup() {
  Serial.begin(9600); // Initialize Serial Monitor
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}

void loop() {
  // Send a 10-microsecond pulse to trigger the sensor
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  // Read the echo pin and measure the time (in microseconds)
  long duration = pulseIn(echoPin, HIGH);
  // Calculate the distance (in cm)
  float distance = (duration * 0.0343)/2;
  //Print the distance to the Serial Monitor
  Serial.print(distance);
  delay(500); //wait for 0.5 second
}
```

Task 1 :

- 3 pushbuttons and one led:
    1. if button1 pressed → blink time 1 sec.
    2. if button2 pressed → blink time 2 sec.
    3. if button3 pressed → blink time 3 sec.

Hardware :



Software :

```
#define LED 12 // Pin connected to the LED

// Define pins for the push buttons
#define PB1 4 // Pin connected to the first push button
#define PB2 5 // Pin connected to the second push button
#define PB3 6 // Pin connected to the third push button

void setup() {
  // Set the LED pin as an output
  pinMode(LED, OUTPUT); // Configure the LED pin as OUTPUT

  // Set push button pins as input with internal pull-up resistors
  pinMode(PB1, INPUT_PULLUP);
  pinMode(PB2, INPUT_PULLUP);
  pinMode(PB3, INPUT_PULLUP);
}
```

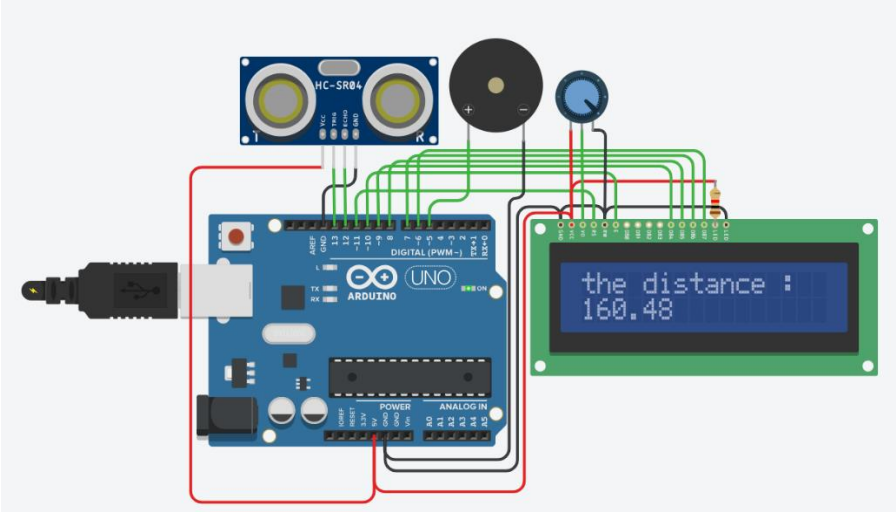
```
void loop() {
  // Read the state of the push buttons
  int PB1_value = digitalRead(PB1);
  int PB2_value = digitalRead(PB2);
  int PB3_value = digitalRead(PB3);


  if (PB1_value == LOW && PB2_value == HIGH && PB3_value == HIGH)
  {// If only the first button is pressed, call blink_1()
    blink_1();
  }
  else if (PB1_value == HIGH && PB2_value == LOW && PB3_value == HIGH)
  {// If only the second button is pressed, call blink_2()
    blink_2();
  }
  else if (PB1_value == HIGH && PB2_value == HIGH && PB3_value == LOW)
  {// If only the third button is pressed, call blink_3()
    blink_3();
  }
  else {// For any other scenario, turn off the LED
    digitalWrite(LED, LOW);
  }
}

// Function to blink the LED every 1 second
void blink_1() {
  digitalWrite(LED, HIGH);
  delay(1000);
  digitalWrite(LED, LOW);
  delay(1000);
}

// Function to blink the LED every 2 seconds
void blink_2() {
  digitalWrite(LED, HIGH);
  delay(2000);
  digitalWrite(LED, LOW);
  delay(2000);
}

// Function to blink the LED every 3 seconds
void blink_3() {
  digitalWrite(LED, HIGH);
  delay(3000);
  digitalWrite(LED, LOW);
  delay(3000);
}
```

Task 2 :

- 1 LCD + 1 Ultrasonic Sensor + 1 Buzzer + 1 Potentiometer

  read distance using Ultrasonic sensor and print its value on LCD

Hardware :

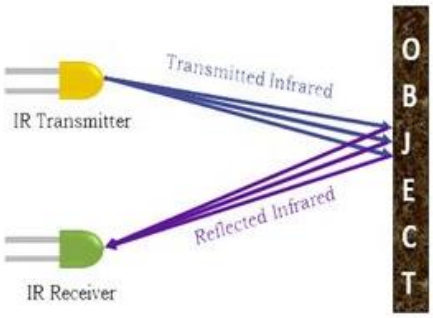


Software:

```
#include <LiquidCrystal.h> //LCD Library

// Define pins for the LCD
const int rs = 11, en = 10, d4 = 9, d5 = 8, d6 = 7, d7 = 6;

LiquidCrystal lcd(rs, en, d4, d5, d6, d7); //create LCD object

#define BUZ 5 // Pin connected to the buzzer
#define TRIG 13 // TRIG pin of the ultrasonic sensor
#define ECHO 12 // ECHO pin of the ultrasonic sensor

void setup() {
  // Initialize the LCD with 16 columns and 2 rows
  lcd.begin(16, 2);

  pinMode(BUZ, OUTPUT); // Set the buzzer pin as output

  // Set the ultrasonic sensor pins
  pinMode(TRIG, OUTPUT); // TRIG pin as output
  pinMode(ECHO, INPUT);  // ECHO pin as input
}
```

```
void loop() {
  // Send a low pulse to the TRIG pin to ensure it's ready
  digitalWrite(TRIG, LOW);
  delayMicroseconds(2); // Wait for 2 microseconds

  // Send a high pulse to the TRIG pin for 10 micro sec.
  //to trigger the ultrasonic signal
  digitalWrite(TRIG, HIGH);
  delayMicroseconds(10); // Wait for 10 microseconds
  digitalWrite(TRIG, LOW); // Turn off the TRIG signal

  // Measure the time(in microseconds) it takes for the ECHO pin
  //to go HIGH
  float duration = pulseIn(ECHO, HIGH);

  // Calculate the distance (in cm) using the duration
  // The formula is: distance = (speed of sound * duration) / 2
  float distance = 0.034 * duration / 2;

  // Display the distance on the LCD
  lcd.setCursor(0, 0); // Set the cursor to 1st row, 1st column
  lcd.print("the distance : "); // Print the label

  lcd.setCursor(0, 1); // Set the cursor to 2nd row, 1st column
  lcd.print(distance); // Print the calculated distance

  // Check if the distance is less than 100 cm
  if (distance < 100) {
    digitalWrite(BUZ, HIGH);
  }
  else {
    digitalWrite(BUZ, LOW);
  }
}
```

Infrared (IR) sensor :



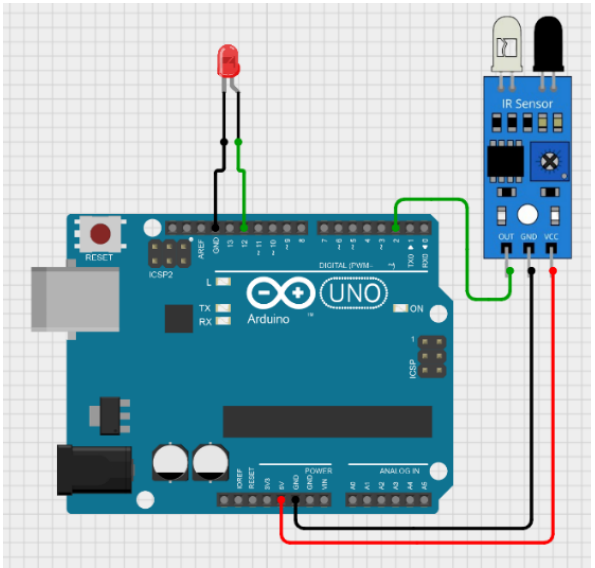
The IR sensor is used to detect objects by utilizing infrared light. It consists of:

1. **IR Transmitter (LED):** Sends infrared light toward an object.
2. **IR Receiver (Photodiode):** Detects reflected infrared light from the object.

How It Works:

1. **Transmission:** The IR transmitter emits infrared light.
2. **Reflection:** If an object is in the path, the light reflects back.
3. **Detection:** The receiver captures the reflected light and converts it into an electrical signal.
4. **Processing:** The circuit processes the signal to detect the object's presence.

Connections :

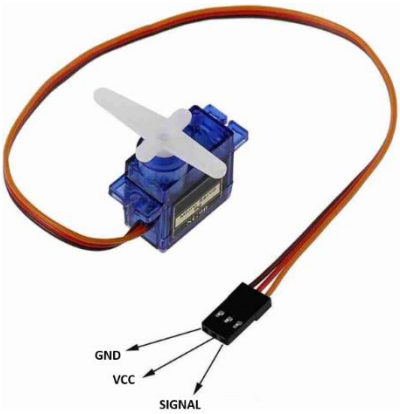


Code for IR sensor Circuit :

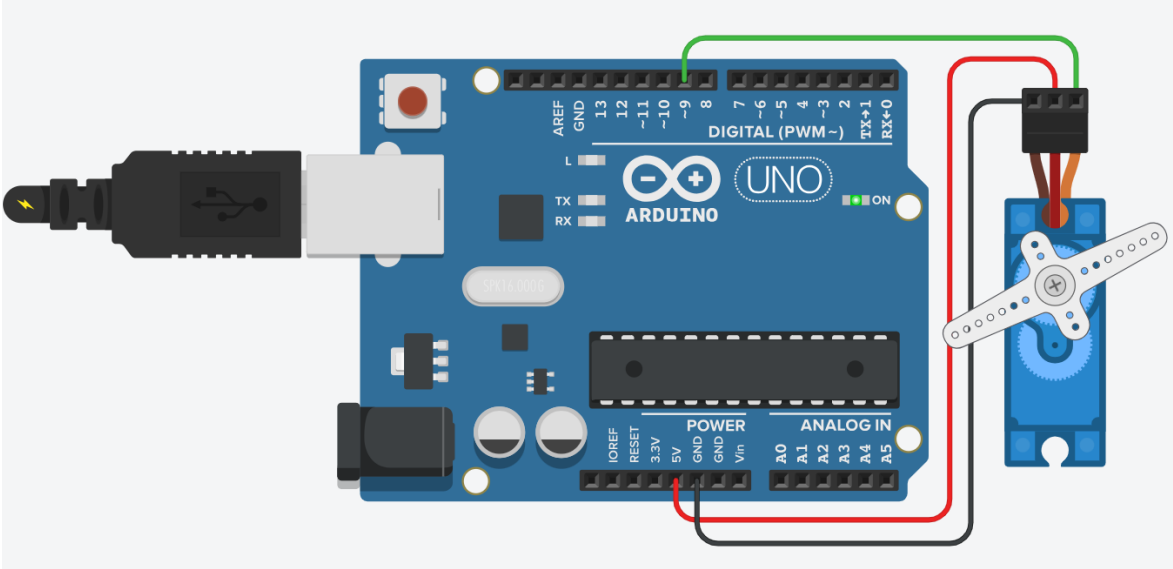
```
const int irSensorPin = 2; // IR sensor signal pin
const int ledPin = 13;     // LED pin

int sensorValue;

void setup() {
  pinMode(irSensorPin, INPUT); // Set IR sensor pin as input
  pinMode(ledPin, OUTPUT);     // Set LED pin as output
}

void loop() {
  sensorValue = digitalRead(irSensorPin); // Read IR sensor
output

  if (sensorValue == LOW) {
    digitalWrite(ledPin, HIGH);
  }
  else {
    digitalWrite(ledPin, LOW);
  }
  delay(100); // Small delay for stability
}
```

Servo motor:

A **servo motor** is a type of motor designed for precise control of angular or linear position, speed, and acceleration. It is commonly used in robotics, automation, and RC devices. Servo motors are equipped with a control circuit that adjusts their position based on input signals.



Connections :

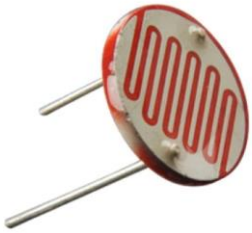


Code :

```
#include <Servo.h>

Servo myservo;  // create servo object to control a servo

int pos = 0;    // variable to store the servo position

void setup() {
  myservo.attach(9); // attach servo on pin 9 to the servo object
}

void loop() {
  for(pos = 0;pos <= 180;pos++) // goes from 0 degrees to 180 degrees
  {
    myservo.write(pos);
    delay(15);
  }
  for(pos = 180;pos >= 0;pos--) // goes from 180 degrees to 0 degrees
  {
    myservo.write(pos);
    delay(15);
  }
}
```

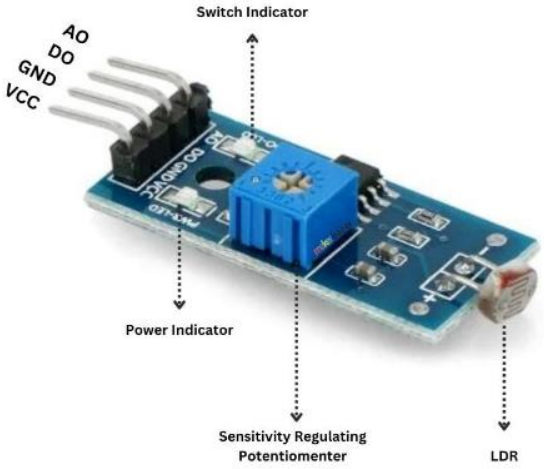
LDR:

An **LDR** (Light Dependent Resistor) is a type of resistor whose resistance changes based on the amount of light falling on it.

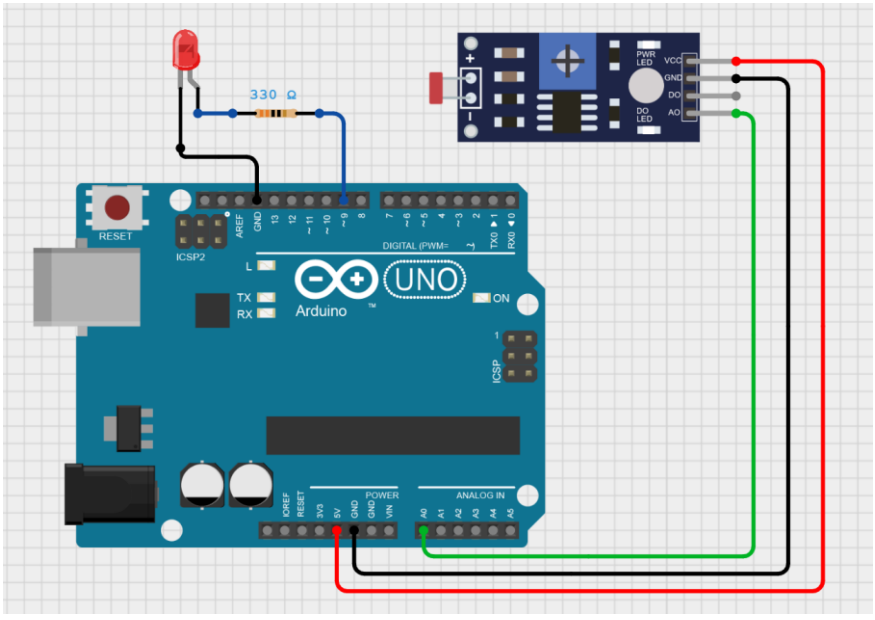


- How it works:
  - In bright light, its resistance is low, allowing more current to pass through.
  - In darkness or low light, its resistance is high, reducing the current flow.
- Uses:
  - Commonly used in light-sensing applications like automatic streetlights, brightness controls, and light alarms.

LDR Module :



Connections:



Code :

```
const int LDR_SensorPin = A0;   // LDR signal pin
const int ledPin = 9;           // PWM pin connected to the LED

void setup() {
  pinMode(LDR_SensorPin, INPUT); // Set LDR sensor pin as input
  pinMode(ledPin, OUTPUT);       // Set LED pin as output
}

void loop() {
  // Read the analog value from the sensor (0 to 1023)
  int sensorValue = analogRead(LDR_SensorPin);

  // Map the sensor value (0-1023) to PWM range (0-255)
  int brightness = map(sensorValue, 0, 1023, 0, 255);

  // Set the LED brightness based on the mapped value
  analogWrite(ledPin, brightness);
  delay(100); // Small delay for signal stability
}
```

Explaining the `map()` Function :

The `map` function in Arduino is used to re-map a number from one range to another. It's especially useful when working with sensors and output devices that operate on different scales.

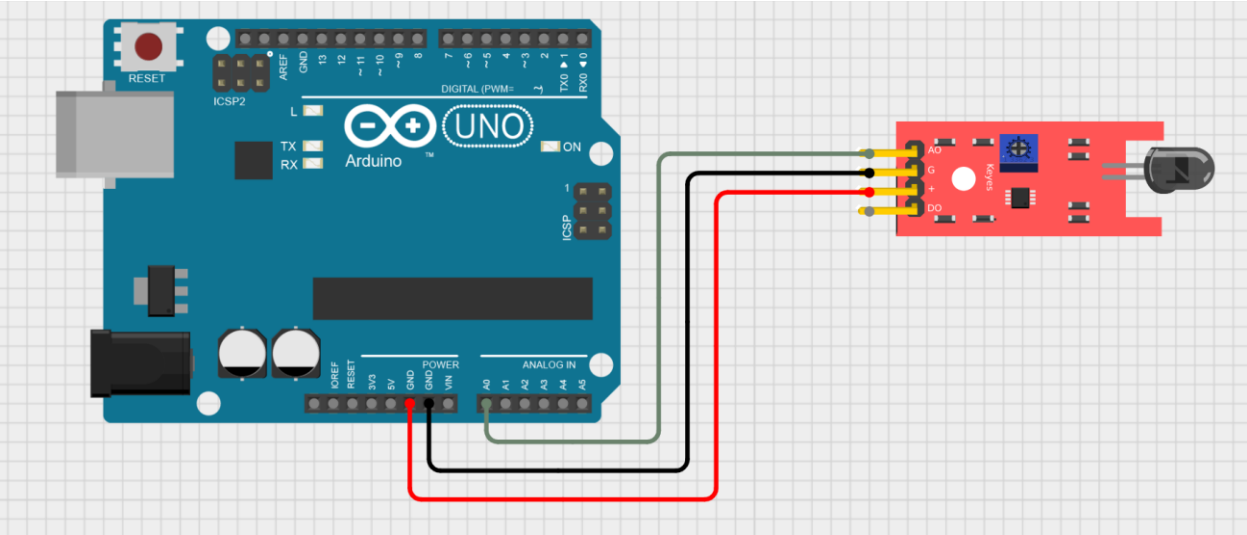
Syntax: `map(value, fromLow, fromHigh, toLow, toHigh)`

- **`value`**: The number you want to map.
- **`fromLow & fromHigh`**: The current range of the value.
- **`toLow & toHigh`**: The desired range to map the value to.

Flame sensor :

It's primarily used for **fire detection**. It can quickly and accurately identify the presence of flames by detecting the infrared (IR) or ultraviolet (UV) light emitted during combustion. This makes it an essential component in **fire alarm systems** to provide early warning and ensure safety in homes, factories, and industrial settings.

Connections :



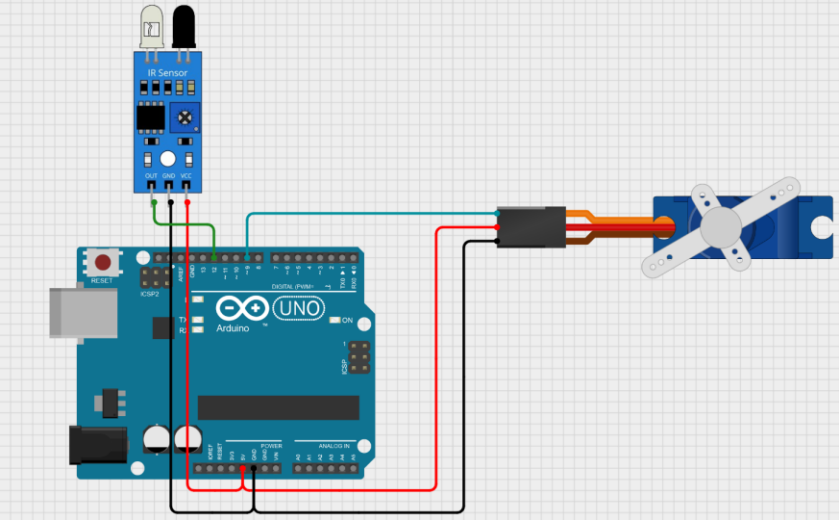
Code :

```
#define FLAME_SENSOR_PIN A0 //analog pin connected to the flame sensor

void setup() {
  Serial.begin(9600); // Start Serial communication at 9600 baud rate
  // Set the flame sensor pin as an input (optional for analog pins)
  pinMode(FLAME_SENSOR_PIN, INPUT);
}

void loop() {
  // Read the analog value from the flame sensor
  int flameValue = analogRead(FLAME_SENSOR_PIN);
  // Display the sensor reading on the Serial Monitor
  Serial.print("Flame Sensor Value: ");
  Serial.println(flameValue);
  delay(500); // Delay of 500ms
}
```

Task 3 :

- 1 IR Sensor + 1 Servo motor
- Read digital value from IR
- if object detected move servo motor 90°
- if not, return to closed position (0°)

Hardware :



Software :

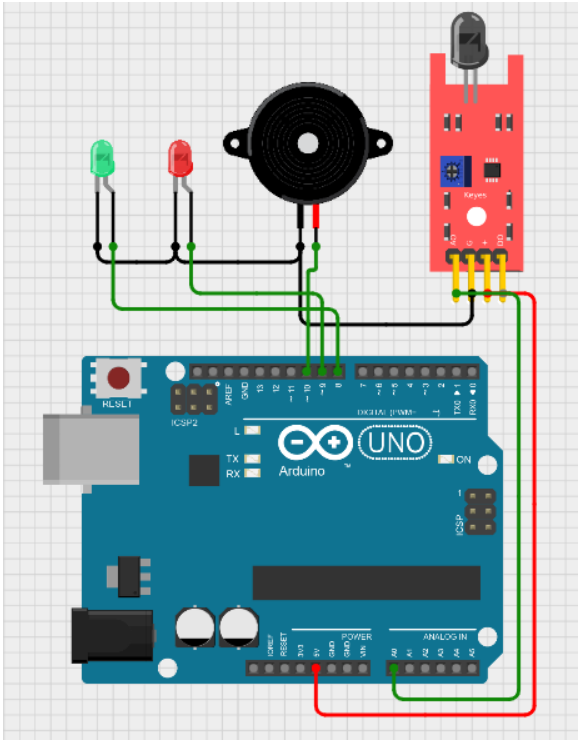
```
#include <Servo.h> // Include the Servo library

const int irSensorPin = 12; // IR sensor signal pin
Servo myServo;              // Create a Servo object

const int closedPosition = 0;   // Servo position when "closed"
const int openPosition = 90;    // Servo position when "open"
int sensorValue;


void setup() {
  pinMode(irSensorPin, INPUT);  // Set IR sensor pin as input
  myServo.attach(9);            // Attach the servo to pin 9
  myServo.write(closedPosition); // Start servo at the "closed"
position
}
```

```
void loop() {
  sensorValue = digitalRead(irSensorPin); // Read sensor signal

  if (sensorValue == LOW) { // If the sensor detects an object
    myServo.write(openPosition); // Move servo to "open" position
  }
  else { // If no object is detected
    myServo.write(closedPosition); //Move servo to "closed" position
  }
  delay(100); // Small delay for stability
}
```

Task 4 :

- 1 Flame Sensor + 1 Buzzer + 1 LED Green + 1 LED Red
- Read flame sensor value
- If reading is less than pre-defined value
  - Open Green LED
- else
  - Open Buzzer
  - Open Red LED

Hardware:

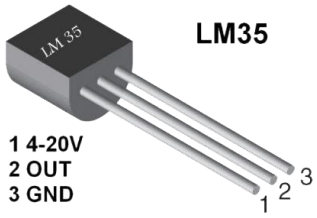


Software :

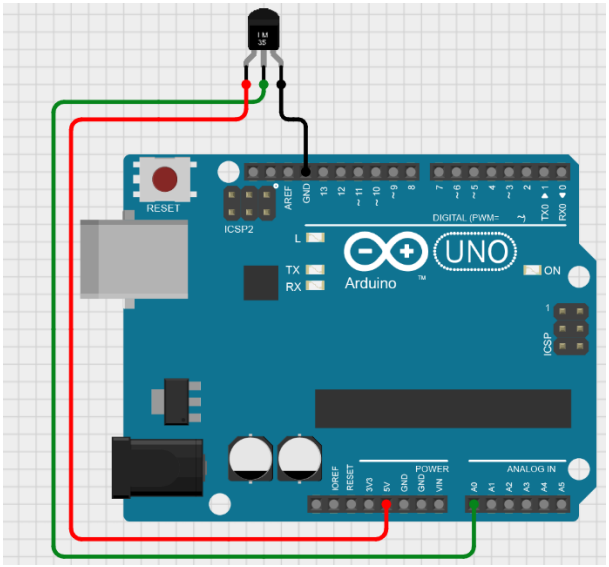
```
#define FLAME_SENSOR_PIN  A0 // Flame Sensor signal pin
#define LED_GREEN_PIN  8     // Digital pin connected to the Green LED
#define LED_RED_PIN  9       // Digital pin connected to the Red LED
#define BUZZER_PIN 10        // Digital pin connected to the Buzzer

int flameThreshold = 500; // Pre-defined level for flame detection
int flameValue;

void setup() {
  pinMode(FLAME_SENSOR_PIN, INPUT);  // Set flame sensor pin as input
  pinMode(LED_GREEN_PIN, OUTPUT);    // Set Green LED pin as output
  pinMode(LED_RED_PIN, OUTPUT);      // Set Red LED pin as output
  pinMode(BUZZER_PIN, OUTPUT);       // Set Buzzer pin as output
}

void loop() {
  // Read the flame sensor value (0-1023)
  flameValue = analogRead(FLAME_SENSOR_PIN);

  // Check if the flame value is below or above the threshold
  if (flameValue < flameThreshold) {
    digitalWrite(LED_GREEN_PIN, HIGH);
    digitalWrite(LED_RED_PIN, LOW);
    digitalWrite(BUZZER_PIN, LOW);
  } else {
    digitalWrite(LED_GREEN_PIN, LOW);
    digitalWrite(LED_RED_PIN, HIGH);
    digitalWrite(BUZZER_PIN, HIGH);
  }
  delay(100); // Small delay for stability
}
```

Temperature sensor (LM35) :

The LM35 is a temperature sensor widely used in various electronics projects and applications. It measures temperature in Celsius with an output voltage that is directly proportional to the temperature. For each degree Celsius rise in temperature, the LM35 outputs a voltage of 10mV, making it simple to interface with an analog-to-digital converter (ADC) for accurate readings.



Connections :

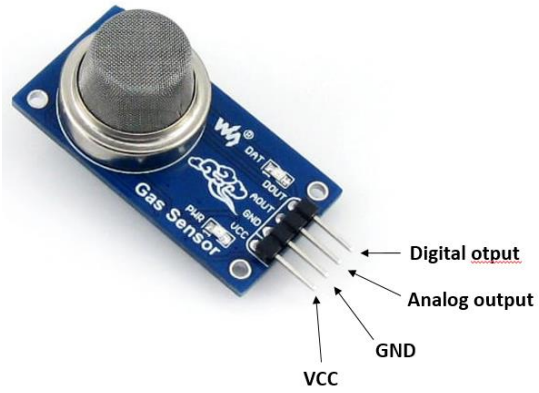


Code :

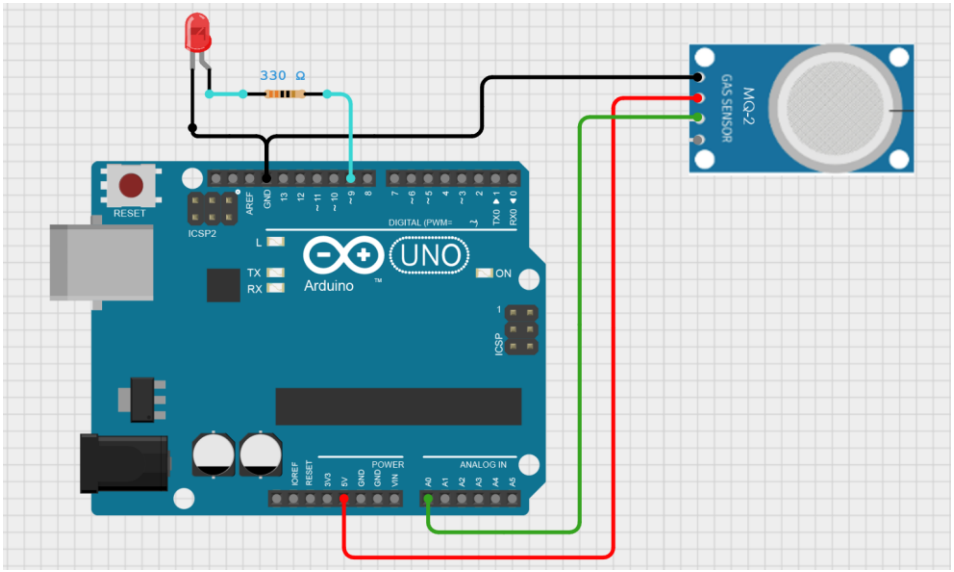
```
#define LM35_PIN A0  // Define the pin where the LM35 is connected

void setup() {
  Serial.begin(9600);   // Start the serial communication
}

void loop() {
  int analogValue = analogRead(LM35_PIN); // Read LM35  value
  float voltage = map(analogValue, 0, 1023, 0, 5); // Convert to volts

  // Convert the voltage to temperature in Celsius by dividing by 0.01
  float temperatureC = voltage / 0.01; // calculate temperature in °C

  Serial.print("Temperature: ");
  Serial.print(temperatureC);
  Serial.println(" °C");
  delay(1000);  // Update every 1 second
}
```

Smoke sensor (MQ2) :

It's used to detect the presence of smoke or combustible gases in the air. It works by measuring changes in resistance caused by gas concentrations, producing an analog signal that can be read by a microcontroller like an Arduino. These sensors are commonly used in fire alarms and safety systems due to their sensitivity to smoke and gases like LPG, methane, and carbon monoxide.



Connections :



Code:

```
#define SMOKE_SENSOR_PIN A0   // Smoke Sensor signal pin
#define LED_PIN 13            // Digital pin for the LED

#define SMOKE_THRESHOLD 300   // Threshold for detecting smoke

int sensorValue;

void setup() {
  pinMode(LED_PIN, OUTPUT);
  pinMode(SMOKE_SENSOR_PIN, INPUT);
}
```

```
void loop() {
  sensorValue = analogRead(SMOKE_SENSOR_PIN);

  // Check if the sensor value exceeds the threshold
  if (sensorValue > SMOKE_THRESHOLD) {
    digitalWrite(LED_PIN, HIGH); // Turn on the LED
  }
  else {
    digitalWrite(LED_PIN, LOW); // Turn off the LED
  }
  delay(1000); // Wait 1 second before the next reading
}
```

Motor driver (L298n) :

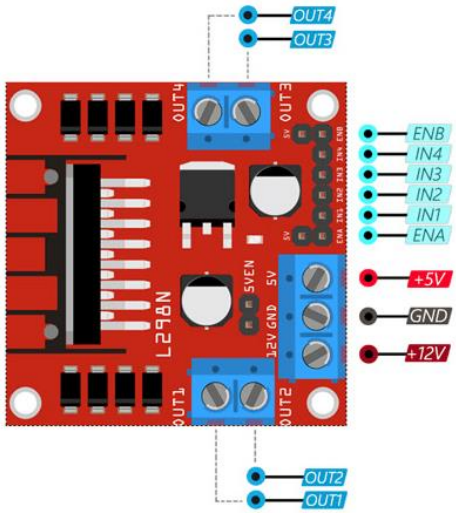
The **L298N motor driver** allows control of two DC motors or one stepper motor. Here's a breakdown of its key components:

Power Pins:

- **+12V:** Connect to an external power source (up to 12V) for motors.
- **+5V:** Powers the L298N logic. If the jumper is in place, the module generates 5V internally.
- **GND:** Common ground for the motor driver, power source, and microcontroller.

Control Pins:

- **ENA/ENB (Enable):** Control the speed of motors via PWM signals.
- **IN1/IN2 and IN3/IN4:** Control motor direction:
  - **HIGH/LOW:** Spins the motor in one direction.
  - **LOW/HIGH:** Spins in the opposite direction.
  - **LOW/LOW:** Stops the motor.



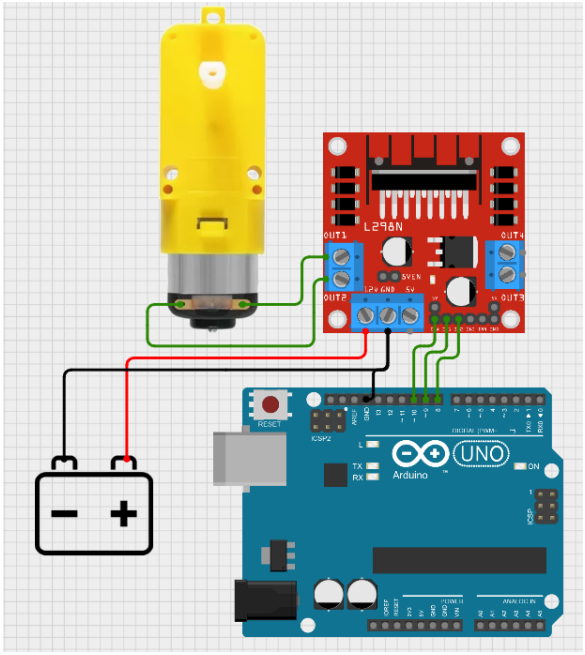
Output Pins:

- **OUT1/OUT2:** Connect to Motor 1.
- **OUT3/OUT4:** Connect to Motor 2.

Basic Arduino Connections:

1. Connect **IN1, IN2** to digital pins for Motor 1.
2. Connect **ENA** to a PWM pin for speed control (or leave the jumper for full speed).
3. Repeat for **IN3, IN4, ENB** for Motor 2.
4. Supply power via **+12V** and ensure common ground with Arduino.

Connections:



Code :

```
#define IN1 9  // Motor direction pin 1 connected to digital pin 9
#define IN2 8  // Motor direction pin 2 connected to digital pin 8
#define ENA 10 // Motor speed control(ENA) connected to PWM pin 10

void setup() {
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(ENA, OUTPUT); // (for PWM speed control)
}

void loop() {
  // Spin the motor in one direction
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  analogWrite(ENA, 128);   // Set speed to 50% duty cycle
  delay(2000);

  // Spin the motor in the opposite direction
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, HIGH);
  analogWrite(ENA, 200);   // Set speed to around 80% duty cycle
  delay(2000);

  // Stop the motor
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, LOW);
  delay(2000);
}
```