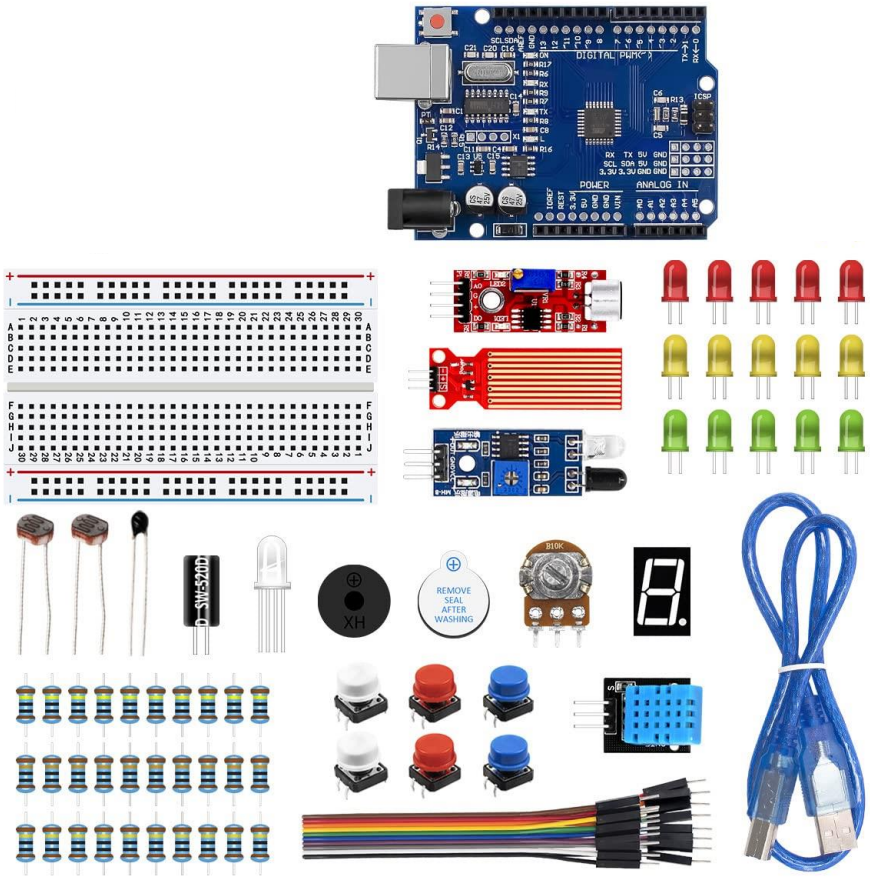
OPTIMUS



Optimus Arduino Student Kit User Manual

Table of Contents

List of Figures

1. Introduction

1.1. Arduino Uno

Figure 1: Arduino UNO board

The Arduino Uno is an open-source microcontroller board based on the Microchip ATmega328P microcontroller and is developed by Arduino.cc and initially released in 2010. The board is equipped with sets of digital and analog input/output (I/O) pins that may be interfaced to various expansion boards (shields) and other circuits. The board has 14 digital I/O pins (six capable of PWM output), 6 analog I/O pins, and is programmable with the Arduino IDE (Integrated Development Environment), via a type B USB cable. It can be powered by a USB cable or a barrel connector that accepts voltages between 7 and 20 volts, such as a rectangular 9-volt battery. It is similar to the Arduino Nano and Leonardo. The hardware reference design is distributed under a Creative Commons Attribution Share-Alike 2.5 license and is available on the Arduino website. Layout and production files for some versions of the hardware are also available.

1.1.1. Technical Specifications

- Microcontroller:
    - IC: Microchip ATmega328P
    - Clock Speed: 16 MHz on Uno board, though IC is capable of 20MHz maximum at 5 Volts.
    - Flash Memory: 32 KB, of which 0.5 KB is used by the bootloader.
    - SRAM: 2 KB
    - EEPROM: 1 KB
    - UART peripherals: 1
    - I2C peripherals: 1
    - SPI peripherals: 1
    - Operating Voltage: 5 Volts
- Digital I/O Pins: 14
- PWM Pins: 6 (Pin # 3, 5, 6, 9, 10 and 11)
- Analog Input Pins: 6
- DC Current per I/O Pin: 20 mA
- DC Current for 3.3V Pin: 50 mA
- Length: 68.6 mm
- Width: 53.4 mm
- Weight: 25 g
- ICSP Header: Yes
- Power Sources:
    - USB connector. USB has a voltage range of 4.75 to 5.25 volts. The official Uno boards have a USB-B connector, but 3rd party Uno boards may have a mini-USB or micro-USB connector.
    - 5.5mm/2.1mm barrel jack connector. Official Uno boards support 6 to 20 volts, though 7 to 12 volts is recommended. The maximum voltage for 3rd party Uno boards varies between board manufacturers because various voltage regulators are used, each having

a different maximum input rating. Power into this connector is routed through a series diode before connecting to VIN to protect against accidental reverse voltage situations.

- VIN pin on shield header. It has a similar voltage range to the barrel jack. Since this pin doesn't have reverse voltage protection, power can be injected or pulled from this pin. When supplying power into VIN pin, an external series diode is required in case barrel jack is used. When the board is powered by barrel jack, power can be pulled out of this pin.

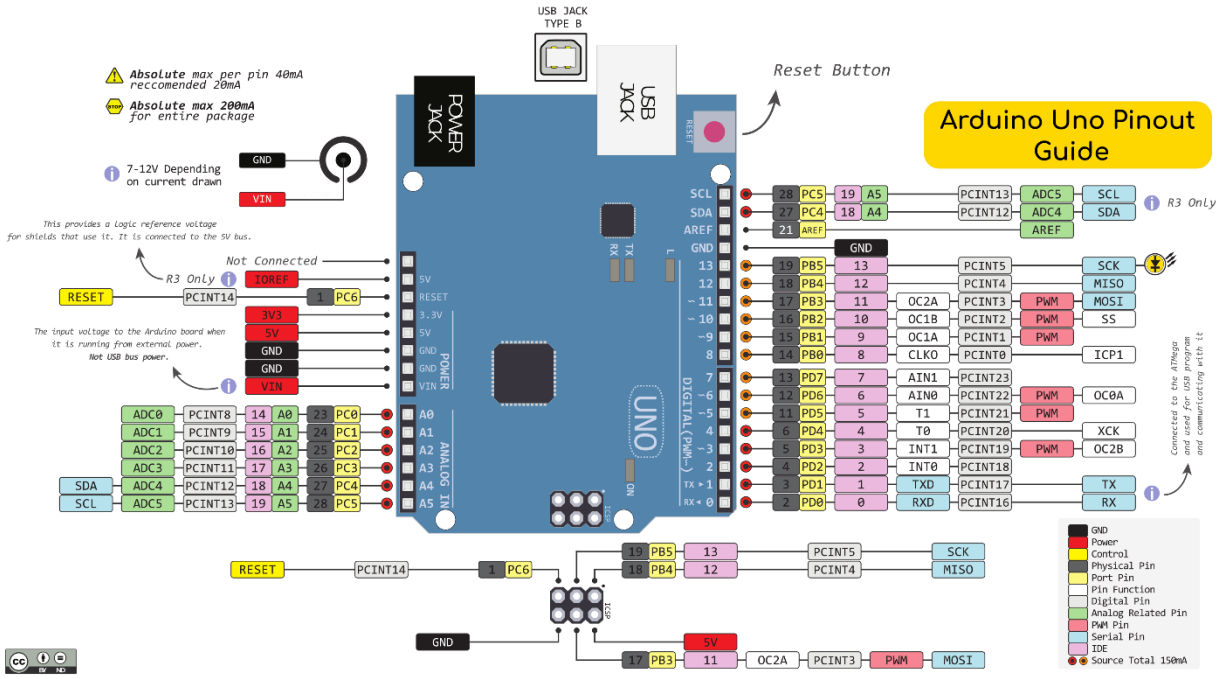


Figure 2: Arduino Uno Pin Configuration

1.1.2. General pin functions

- **LED**: There is a built-in LED driven by digital pin 13. When the pin is HIGH, the LED is on, when the pin is LOW, it is off.

- **VIN**: The input voltage to the Arduino/Genuino board when it uses an external power source (as opposed to 5 volts from the USB connection or other regulated power source). You can supply voltage through this pin, or, if supplying voltage via the power jack, access it through this pin.

- **5V**: This pin outputs a regulated 5V from the regulator on the board. The board can be supplied with power either from the DC power jack (7 - 20V), the USB connector (5V), or the VIN pin of the board (7-20V). Supplying voltage via the 5V or 3.3V pins bypasses the regulator and can damage the board.
- **3V3**: A 3.3 Volt supply generated by the on-board regulator. The maximum current draw is 50 mA.
- **GND**: Ground pins.
- **IOREF**: This pin on the Arduino/Genuino board provides the voltage reference with which the microcontroller operates. A properly configured shield can read the IOREF pin voltage and select the appropriate power source or enable voltage translators on the outputs to work with the 5V or 3.3V.
- **Reset**: Typically used to add a reset button to shields that block the one on the board.

1.1.3. Special pin functions

Each of the 14 digital pins and 6 analog pins on the Uno can be used as an input or output, under software control (using pinMode(), digitalWrite(), and digitalRead() functions). They operate at 5 volts. Each pin can provide or receive 20 mA as the recommended operating condition and has an internal pull-up resistor (disconnected by default) of 20-50K ohm. A maximum of 40mA must not be exceeded on any I/O pin to avoid permanent damage to the microcontroller. The Uno has 6 analog inputs, labeled **A0** through **A5**; each provides 10 bits of resolution (i.e., 1024 different values). By default, they measure from ground to 5 volts, though it is possible to change the upper end of the range using the AREF pin and the analogReference() function.

In addition, some pins have specialized functions:

Serial / UART: pins 0 (RX) and 1 (TX). Used to receive (RX) and transmit (TX) TTL serial data. These pins are connected to the corresponding pins of the ATmega8U2 USB-to-TTL serial chip.

- **External interrupts**: pins 2 and 3. These pins can be configured to trigger an interrupt on a low value, a rising or falling edge, or a change in value.
- PWM (pulse-width modulation): pins 3, 5, 6, 9, 10, and 11. Can provide 8-bit PWM output with the analogWrite() function.

- SPI (Serial Peripheral Interface): pins 10 (SS), 11 (MOSI), 12 (MISO), and 13 (SCK). These pins support SPI communication using the SPI library.
- **TWI** (two-wire interface) / I²C: pin SDA (A4) and pin SCL (A5). Support TWI communication using the Wire library.
- **AREF** (analog reference): Reference voltage for the analog inputs.

1.1.4. Pullup & Pulldown in Arduino

In digital electronics, 0s and 1s are used to transmit data, and since you cannot send a number 0 or 1 through a cable, voltage values are used to indicate each of the states. For the HIGH or 1 a value of 5V will be used, and for LOW or 0 a value of 0v will be used. Of course, nothing is perfect, and the signal is not going to be exactly at 0V or 5V (resistance and noise of the cables for example). To avoid interpretation errors, the circuits have a noise margin that allows them to interpret the signals as HIGH or LOW even though the voltage is not exactly 0V or 5V.

FLOAT, is when a pin is not in a fixed state and fluctuates (for example a loose cable connected to an Arduino pin). This can cause the state of our pin to vary between 0 and 1, and end up giving false positives.

Following diagrams will show how pullup & pulldown resistors are connected to a pushbutton, signal pin, digital input, etc.

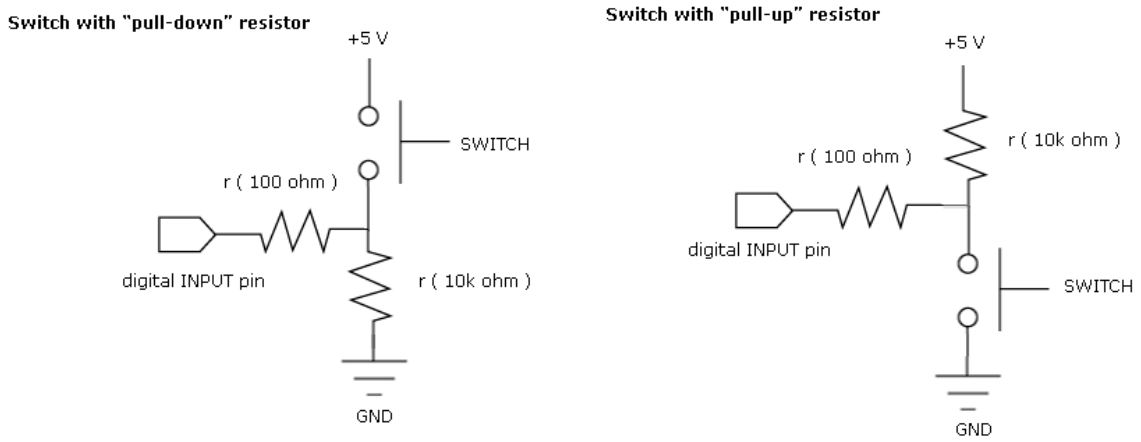


Figure 3: Pullup & Pulldown Resistors

2. LEDs

A light-emitting diode (LED) is a semiconductor device that generates light when current passes through it. Electrons in the semiconductor recombine with electron holes, producing energy in the form of photons. The energy required for electrons to pass the band gap of the semiconductor determines the hue of the light (equivalent to the energy of photons). Multiple semiconductors or a layer of light-emitting phosphor on the semiconductor device are used to produce white light.

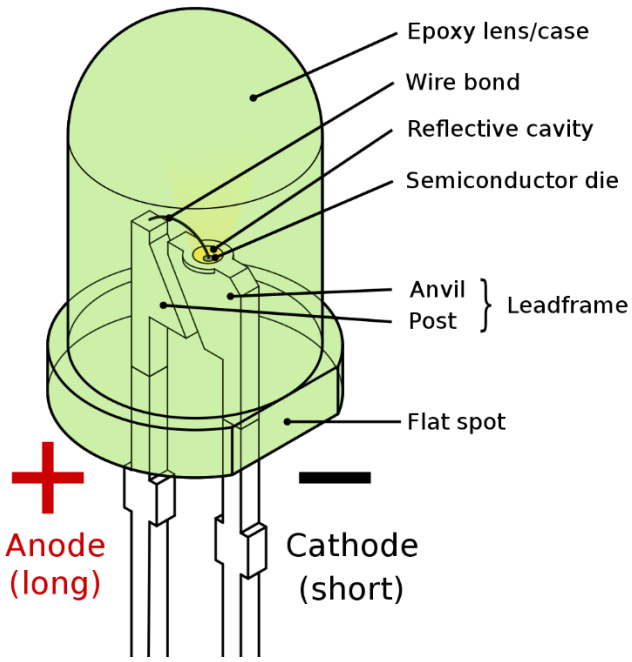


Figure 4: Parts of a LED

Table 1: LED Voltage & Current

Colour	Forward Voltage V_f	Forward Current I_f
White	3.2 - 3.8V	20mA – 30mA
Warm White	3.2 - 3.8V	20mA – 30mA
Blue	3.2 - 3.8V	20mA – 30mA
Red	1.8 - 2.2V	20mA – 30mA
Green	3.2 - 3.8V	20mA – 30mA
Yellow	1.8 - 2.2V	20mA – 30mA
Orange	1.8 - 2.2V	20mA – 30mA
Pink	3.2 - 3.8V	20mA – 30mA
UV	3.2 - 3.8V	20mA – 30mA

3. Resistors

A resistor is a two-terminal passive electrical component that acts as a circuit element by implementing electrical resistance. Resistors are used in electronic circuits to, among other things, reduce current flow, regulate signal levels, split voltages, bias active devices, and terminate transmission lines. High-power resistors, which may waste hundreds of watts of electrical power as heat, can be utilized in motor controllers, power distribution systems, or as generator test loads. Resistances of fixed resistors vary only minimally with temperature, time, or operating voltage. Variable resistors can be used to alter circuit components (such as a volume control or a lamp dimmer) or as heat, light, humidity, force, or chemical activity detecting devices.

The electrical function of a resistor is described by its resistance: typical commercial resistors are made across a range of more than nine orders of magnitude. The nominal value of the resistance is within the manufacturing tolerance listed on the component.

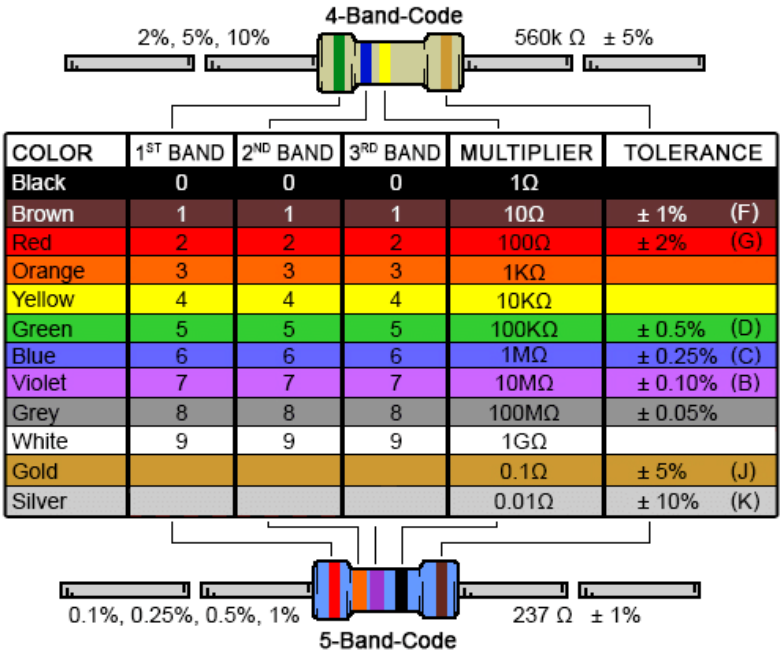
Ohm's Law

Ohm's law describes the behavior of an ideal resistor:

$$V = IR$$

Ohm's law says that the voltage (V) across a resistor is proportional to the current (I) running through it, with the resistance (R) serving as the proportionality constant. For example, if a 300-ohm resistor is connected across the terminals of a 12-volt battery, a current of 12 / 300 = 0.04 amperes runs through it.

The ohm (symbol: **Ω**) is the SI unit of electrical resistance was named after Georg Simon Ohm. A volt per ampere is equal to one ohm. Because resistors are specified and manufactured in a wide range of values, Figure 4 shows the resistor color code which represents the resistance of a resistor.



COLOR	1ST BAND	2ND BAND	3RD BAND	MULTIPLIER	TOLERANCE
Black	0	0	0	1Ω	
Brown	1	1	1	10Ω	± 1% (F)
Red	2	2	2	100Ω	± 2% (G)
Orange	3	3	3	1KΩ	
Yellow	4	4	4	10KΩ	
Green	5	5	5	100KΩ	± 0.5% (D)
Blue	6	6	6	1MΩ	± 0.25% (C)
Violet	7	7	7	10MΩ	± 0.10% (B)
Grey	8	8	8	100MΩ	± 0.05%
White	9	9	9	1GΩ	
Gold				0.1Ω	± 5% (J)
Silver				0.01Ω	± 10% (K)

Figure 5: Resistor Colour Code

3.1. Potentiometer

A potentiometer is a three-terminal variable resistor of which the resistance can be adjusted manually. Two terminals are attached to opposing ends of a resistive element, while the third terminal is connected to a sliding contact, known as a wiper, that moves over the resistive element.

Figure 6: A Potentiometer

3.2. LDR

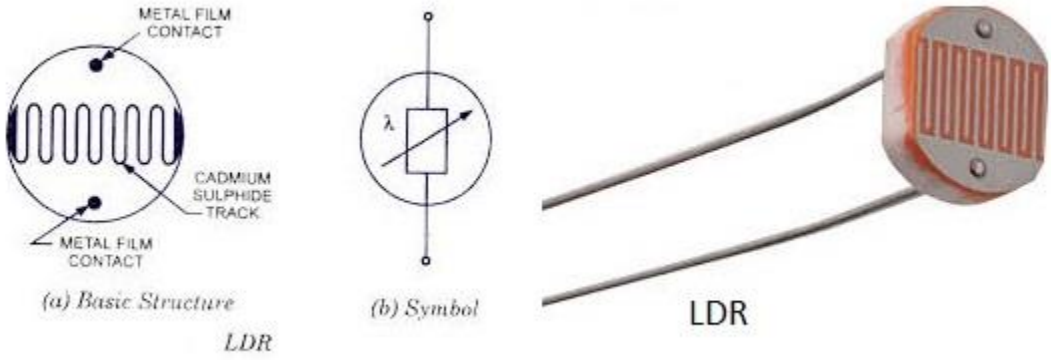


Figure 7: LDR

An LDR (Light Dependent Resistor), also known as a photoresistor, is constructed with a serpentine or zigzag pattern of photo-sensitive material. This material's resistance changes in response to the intensity of light falling on it.

The photoresistor's behavior is directly linked to the amount of light it receives. In darkness or low-light conditions, the photoresistor exhibits a high resistance, which can reach up to several megaohms (millions of ohms). In such conditions, with minimal light exposure, the material's structure limits the flow of current through the photoresistor, resulting in high resistance.

Conversely, when the photoresistor is exposed to light, whether natural or artificial, its resistance significantly decreases. As the intensity of light increases, the resistance of the photoresistor drops. In well-lit environments, the resistance of the photoresistor decreases to a few hundred ohms, allowing more current to flow through it.

This characteristic behavior of the LDR is crucial in various light-sensing applications. It enables the sensor to detect changes in light levels and provides an analog output that correlates with the ambient light conditions. The resistance versus illumination characteristic curve showcases the inverse relationship between the resistance of the LDR and the intensity of light falling on it, demonstrating how the sensor's resistance changes with varying levels of illumination.

4. Exercise 1: Blinking a LED

Getting Started

But first, we need to download and install the Arduino IDE (if you haven't already). Please download and install it.

Introducing digital output

Now, we are going to write code to turn on our LED by setting Pin 3 to HIGH (or 5V). Then, we will modify this code to flash the LED both on *and* off. To do this, we must introduce **digital output.**

The Arduino Uno has 14 general-purpose input/output (GPIO) pins that can be used for digital input/output (I/O)—that is, to read or write digital information (HIGH or LOW) using digitalRead() and digitalWrite(), respectively. We could have selected any of these pins for this exercise, but we chose Pin 3.

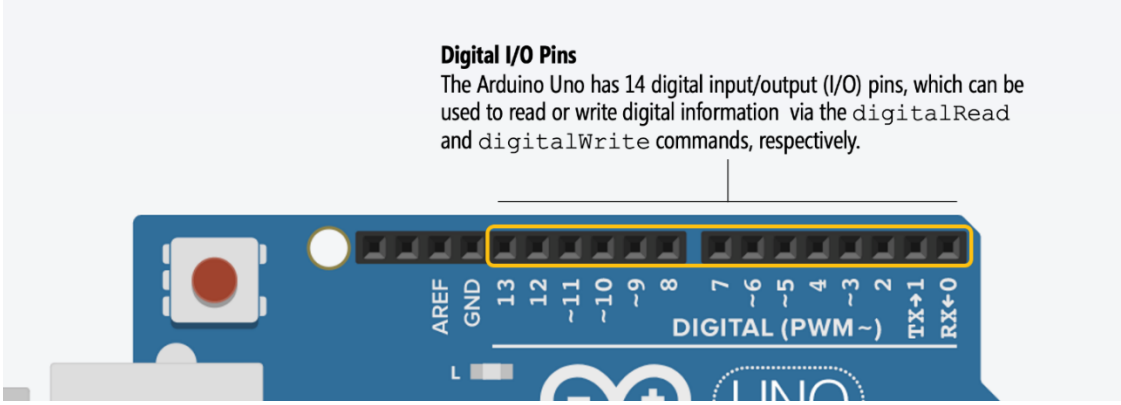


Figure 8: Digital Pins of Arduino Uno

You can control any of these 14 digital I/O pins with three functions:

1. pinMode(int pin, int mode) configures a specified pin as either an INPUT or OUTPUT. In this case, we want to specify OUTPUT because we want to **output** a signal to turn on the LED.
2. digitalRead(int pin) reads digital input from the specified pin, either HIGH or LOW. We will cover digitalRead in our Intro to Input lesson series.
3. digitalWrite(int pin, int value) writes digital output to the specified pin, either HIGH or LOW. We'll be using digitalWrite in this lesson.

What do we mean by HIGH and LOW?

An Arduino's supply voltage is often written as VS, VCC, and VDD in datasheets.

On the Arduino Uno and Leonardo, the supply voltage (VCC) is **5V**. So, when a pin as configured as an output via pinMode(<pin>, OUTPUT), the pin can provide either a HIGH voltage (VCC) or a LOW voltage (0V). Some microcontrollers operate at 3.3V. In this case, a HIGH state would be 3.3V but a LOW state would still be 0V.

What can we use digital output pins for?

In general, digital output pins on microcontrollers are designed to send **control signals** and not act as **power supplies**. So, while these pins can supply enough current to use LEDs, piezo speakers, or control servo motors, if you need to control a high-current DC load such as a DC motor, you'll need to use a transistor which is an electronically controlled switch.

What's the maximum amount of current a digital output pin can supply?

The Arduino Uno uses the ATmega328P microcontroller, which can supply an absolute maximum of 0.04A (40 mA) per digital output pin or about ~4 LEDs in parallel (with 10mA per branch).

According to Section 28.1 in the ATmega328P datasheet, anything beyond these limits "*may cause permanent damage to the chip*". The maximum total current draw **across all I/O pins** together should not exceed 200mA. Again, this limit is not a concern for our introductory lessons (unless you deviate significantly from them).

Importantly, once you configure a digital I/O pin as OUTPUT, do not connect it directly to GND or VCC or you may damage the microcontroller (typically, just that particular pin will be damaged). So, for example, if you've accidentally connected Pin 3 directly to 5V and write pinMode(3, OUTPUT); digitalWrite(3, LOW);, a whole bunch of current will "sink" into Pin 3 and potentially damage the pin.

Required components for the Exercise 1:

- Red LED x 1
- Male to Male 20cm Jumper Cables x 2
- 220 Ohm Resistor x 1
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

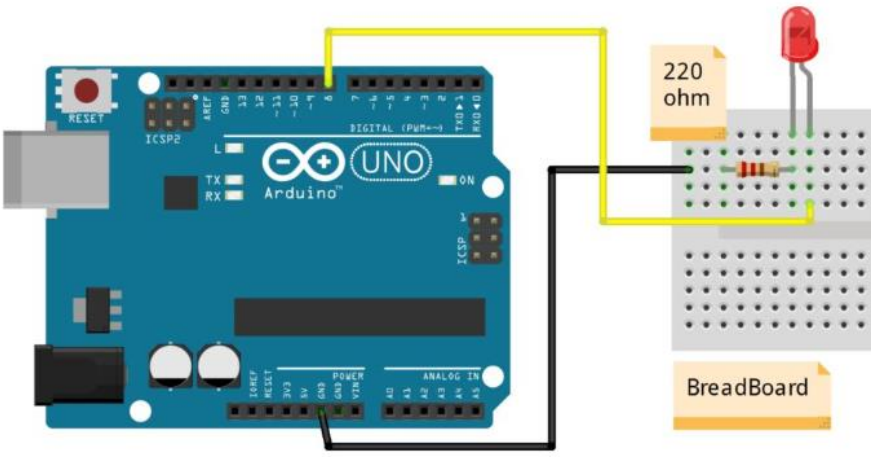


Figure 9: Wiring Diagram for Exercise 1

Sample Code

```
const int LED_OUTPUT_PIN = 3;
void setup() {
  // set Pin 3 to output
  pinMode(LED_OUTPUT_PIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_OUTPUT_PIN, HIGH);  // turn LED on (output 5V)
  delay(1000);                         // wait one second
  digitalWrite(LED_OUTPUT_PIN, LOW);   // turn LED off (output 0V)
  delay(1000);                         // wait another second
}
```

Exercise 2: Blinking four LEDs

In this exercise we will blink four LEDs one after the other. Use 4 LEDs as your preference.

Required components for the Exercise 2:

- Red LED x 2
- Yellow LED x 1
- Green LED x 1
- Male to Male 20cm Jumper Cables x 5
- 220 Ohm Resistors x 5
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

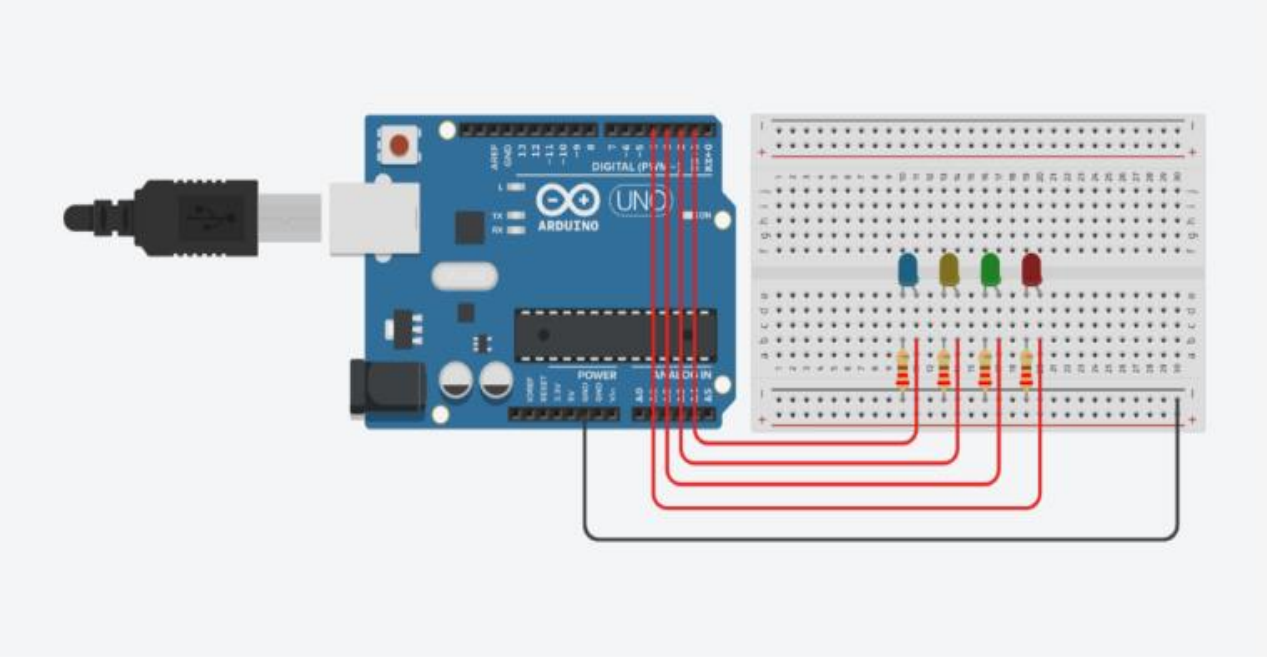


Figure 10: Wiring Diagram for Exercise 2

Sample Code

```
const int LED1_OUTPUT_PIN = 3;  // Anode faces Pin 3 (cathode connected to 0V)
const int LED2_OUTPUT_PIN = 4;  // Anode faces Pin 4 (cathode connected to 0V)
const int LED3_OUTPUT_PIN = 5;  // Anode faces Pin 5 (cathode connected to 0V)
const int LED4_OUTPUT_PIN = 6;  // Anode faces Pin 6 (cathode connected to 0V)
const int DELAY_MS = 1000;      // delay for 1 sec between blinks

// The setup function runs once when you press reset or power the board
void setup() {
  // Set our LED pins as output
  pinMode(LED1_OUTPUT_PIN, OUTPUT);
  pinMode(LED2_OUTPUT_PIN, OUTPUT);
  pinMode(LED3_OUTPUT_PIN, OUTPUT);
  pinMode(LED4_OUTPUT_PIN, OUTPUT);
}

// The loop function runs over and over again forever
void loop() {

  digitalWrite(LED1_OUTPUT_PIN, HIGH);  // turns ON LED1
  digitalWrite(LED2_OUTPUT_PIN, LOW);   // turns OFF LED2
  digitalWrite(LED3_OUTPUT_PIN, LOW);   // turns OFF LED3
  digitalWrite(LED4_OUTPUT_PIN, LOW);   // turns OFF LED4
  delay(DELAY_MS);                      // delay is in milliseconds; so wait one
second

  digitalWrite(LED1_OUTPUT_PIN, LOW);   // turns OFF LED1
  digitalWrite(LED2_OUTPUT_PIN, HIGH);  // turns ON LED2
  digitalWrite(LED3_OUTPUT_PIN, LOW);   // turns OFF LED3
  digitalWrite(LED4_OUTPUT_PIN, LOW);   // turns OFF LED4
  delay(DELAY_MS);                      // wait for a second

  digitalWrite(LED1_OUTPUT_PIN, LOW);   // turns OFF LED1
  digitalWrite(LED2_OUTPUT_PIN, LOW);   // turns OFF LED2
  digitalWrite(LED3_OUTPUT_PIN, HIGH);  // turns ON LED3
  digitalWrite(LED4_OUTPUT_PIN, LOW);   // turns OFF LED4
  delay(DELAY_MS);                      // wait for a second

  digitalWrite(LED1_OUTPUT_PIN, LOW);   // turns OFF LED1
  digitalWrite(LED2_OUTPUT_PIN, LOW);   // turns OFF LED2
  digitalWrite(LED3_OUTPUT_PIN, LOW);   // turns OFF LED3
  digitalWrite(LED4_OUTPUT_PIN, HIGH);  // turns ON LED4
  delay(DELAY_MS);                      // wait for a second
}
```

5. Exercise 3: Fading a LED

In this exercise, we'll look at how to use **analogWrite ()** function to modify the output voltage at finer gradations.

Required components for the Exercise 3:

- Red LED x 1
- 2 Male to Male 20cm Jumper Cables x 2
- 220 Ohm Resistor x 1
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

Pulse-width modulation (PWM)

The Arduino Uno, Leonardo, Nano, Mega, and many more Arduino boards, despite their names, do not enable true analog output through a digital-to-analog converter (DAC). Instead, they simulate analog output using a technique known as Pulse-Width Modulation (PWM). This is irrelevant for most applications, such as changing the brightness of an LED or regulating the speed of a motor.

So, what exactly does the analogWrite function do? The 8-bit value (0-255) determines how long a 5V signal is supplied to the output pin during one "analog write" period. So, analogWrite(pin>, 127) would produce a 5V value for half of the period (because 127/255 = 50%) and analogWrite(pin>, 191) would produce a 5V for 75% of the period (because 191/255 = 75%). The duty cycle is the percentage of time that the signal is HIGH.

The hardware PWM pins in Arduino Uno are 3,5,6,9,10 & 11.

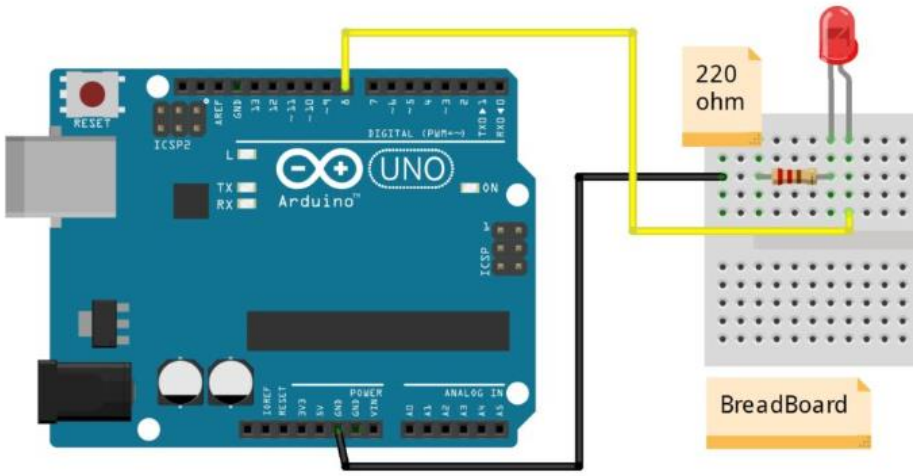


Figure 11: Wiring Diagram for Exercise 3

Sample Code

```
const int LED_OUTPUT_PIN = 3;
const int DELAY_MS = 5;          // delay between each fade value
const int MAX_ANALOG_OUT = 255;  // the max analog output on the Uno is 255

void setup() {
  // set the LED pin to an output
  pinMode(LED_OUTPUT_PIN, OUTPUT);
}

void loop() {
  // fade-in
  for (int i = 0; i <= MAX_ANALOG_OUT; i += 1) {
    analogWrite(LED_OUTPUT_PIN, i);
    delay(DELAY_MS);
  }

  //fade-out
  for (int i = MAX_ANALOG_OUT; i >= 0; i -= 1) {
    analogWrite(LED_OUTPUT_PIN, i);
    delay(DELAY_MS);
  }
}
```

6. Exercise 4: Cross Fading RGB LED Module

The crossfade approach in this exercise works by increasing one LED color value (from 0 to 255) while reducing another (from 255 to 0). For example, the code begins by reducing the value of the red LED while raising the value of the green LED. When the red LED value approaches zero, we start decrementing another LED (in this example, green). Similarly, when the green LED value hits 255, we start incrementing another LED (in this example, the blue LED), and so on.

Required components for the Exercise 4:

- RGB LED Module
- Male to Male 20cm Jumper Cable x 4
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

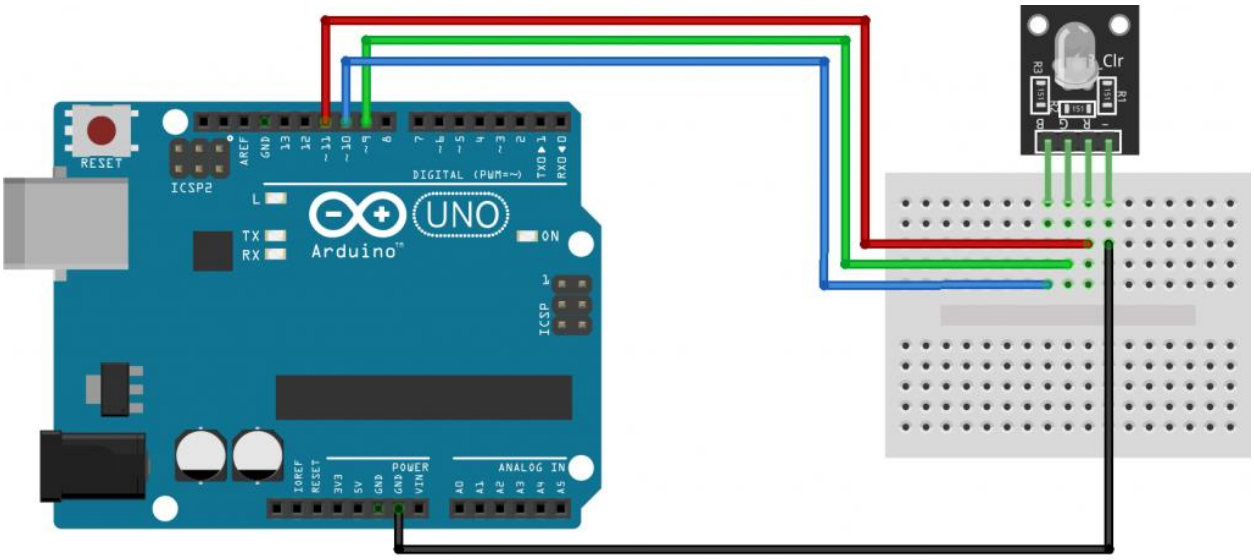


Figure 12: Wiring Diagram for Exercise 4

Sample Code

```
// Change this to based on whether you are using a common anode or common cathode
// If you are working with a common cathode RGB LED, set this to false.
const boolean COMMON_ANODE = false;

const int RGB_RED_PIN = 11;
const int RGB_GREEN_PIN  = 9;
const int RGB_BLUE_PIN  = 10;
const int DELAY_MS = 20; // delay in ms between changing colors
const int MAX_COLOR_VALUE = 255;

enum RGB{
  RED,
  GREEN,
  BLUE,
  NUM_COLORS
};

int _rgbLedValues[] = {255, 0, 0}; // Red, Green, Blue
enum RGB _curFadingUpColor = GREEN;
enum RGB _curFadingDownColor = RED;
const int FADE_STEP = 5;

void setup() {
  // Set the RGB pins to output
  pinMode(RGB_RED_PIN, OUTPUT);
  pinMode(RGB_GREEN_PIN, OUTPUT);
  pinMode(RGB_BLUE_PIN, OUTPUT);

  // Turn on Serial so we can verify expected colors via Serial Monitor
  Serial.begin(9600);
  Serial.println("Red, Green, Blue");

  // Set initial color
  setColor(_rgbLedValues[RED], _rgbLedValues[GREEN], _rgbLedValues[BLUE]);
  delay(DELAY_MS);
}

void loop() {

  // Increment and decrement the RGB LED values for the current
  // fade up color and the current fade down color
  _rgbLedValues[_curFadingUpColor] += FADE_STEP;
  _rgbLedValues[_curFadingDownColor] -= FADE_STEP;
```

```
  // Check to see if we've reached our maximum color value for fading up
  // If so, go to the next fade up color (we go from RED to GREEN to BLUE
  // as specified by the RGB enum)
  // This fade code partially based on: https://gist.github.com/jamesotron/766994
  if(_rgbLedValues[_curFadingUpColor] > MAX_COLOR_VALUE){
    _rgbLedValues[_curFadingUpColor] = MAX_COLOR_VALUE;
    _curFadingUpColor = (RGB)((int)_curFadingUpColor + 1);

    if(_curFadingUpColor > (int)BLUE){
      _curFadingUpColor = RED;
    }
  }

  // Check to see if the current LED we are fading down has gotten to zero
  // If so, select the next LED to start fading down (again, we go from RED to
  // GREEN to BLUE as specified by the RGB enum)
  if(_rgbLedValues[_curFadingDownColor] < 0){
    _rgbLedValues[_curFadingDownColor] = 0;
    _curFadingDownColor = (RGB)((int)_curFadingDownColor + 1);

    if(_curFadingDownColor > (int)BLUE){
      _curFadingDownColor = RED;
    }
  }

  // Set the color and then delay
  setColor(_rgbLedValues[RED], _rgbLedValues[GREEN], _rgbLedValues[BLUE]);
  delay(DELAY_MS);
}

/**
 * setColor takes in values between 0 - 255 for the amount of red, green, and
blue, respectively
 * where 255 is the maximum amount of that color and 0 is none of that color. You
can illuminate
 * all colors by intermixing different combinations of red, green, and blue
 *
 * This function is based on https://gist.github.com/jamesotron/766994
 */
void setColor(int red, int green, int blue)
{
  Serial.print(red);
  Serial.print(", ");
  Serial.print(green);
```

```
  Serial.print(", ");
  Serial.println(blue);

  // If a common anode LED, invert values
  if(COMMON_ANODE == true){
    red = MAX_COLOR_VALUE - red;
    green = MAX_COLOR_VALUE - green;
    blue = MAX_COLOR_VALUE - blue;
  }
  analogWrite(RGB_RED_PIN, red);
  analogWrite(RGB_GREEN_PIN, green);
  analogWrite(RGB_BLUE_PIN, blue);
}
```

7. Exercise 5: Arduino Knight Rider LED Pattern

The Knight Rider LED circuit is a popular circuit that creates a scanning or sweeping effect resembling the front lights of the KITT car from the TV show Knight Rider. The circuit typically consists of LEDs arranged in a line or a row and controlled by a microcontroller like an Arduino.

Required components for the Exercise 5:

- Red LED x 5
- Male to Male 20cm Jumper Cable x 9
- 220 Ohm Resistor x 5
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

This code sets up the pins for 5 LEDs and creates a loop that lights them up in sequence from one end to the other and then back again. Adjust the delay duration to change the speed of the effect. Upload this code to your Arduino and connect the LEDs to the specified pins.

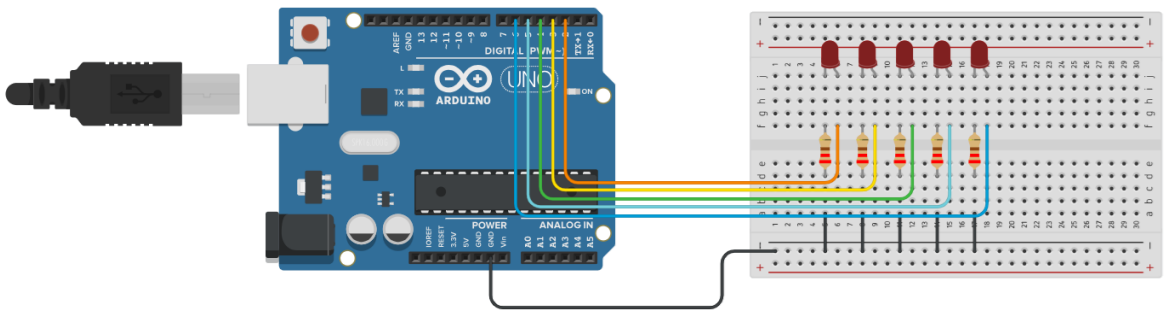


Figure 13: Wiring Diagram for Exercise 5

Tip: Place resistors and LEDs to use minimum number of jumper wires, above diagram is for demonstration.

Sample Code

```
const int numLEDs = 5;  // Number of LEDs
const int ledPins[] = {2, 3, 4, 5, 6};  // Pins for the LEDs

void setup() {
  // Initialize LED pins as outputs
  for (int i = 0; i < numLEDs; i++) {
    pinMode(ledPins[i], OUTPUT);
  }
}

void loop() {
  // Knight Rider effect: LEDs moving forward
  for (int i = 0; i < numLEDs; i++) {
    digitalWrite(ledPins[i], HIGH);  // Turn on LED
    delay(100);  // Adjust the delay for the speed of the effect
    digitalWrite(ledPins[i], LOW);  // Turn off LED
  }

  // Knight Rider effect: LEDs moving backward
  for (int i = numLEDs - 1; i >= 0; i--) {
    digitalWrite(ledPins[i], HIGH);  // Turn on LED
    delay(100);  // Adjust the delay for the speed of the effect
    digitalWrite(ledPins[i], LOW);  // Turn off LED
  }
}
```

8. Exercise 6: Arduino Light Sensor Using LDR

The below code will turn on the LED when it is dark and turn off the LED when it is bright. Upload the code to your Arduino board and change the lighting condition of your environment and see the output.

Required components for the Exercise 6:

- Red LED x 1
- LDR x 1
- Male to Male 20cm Jumper Cable x 6
- 220 Ohm Resistor x 1
- 100k Resistor x 1
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

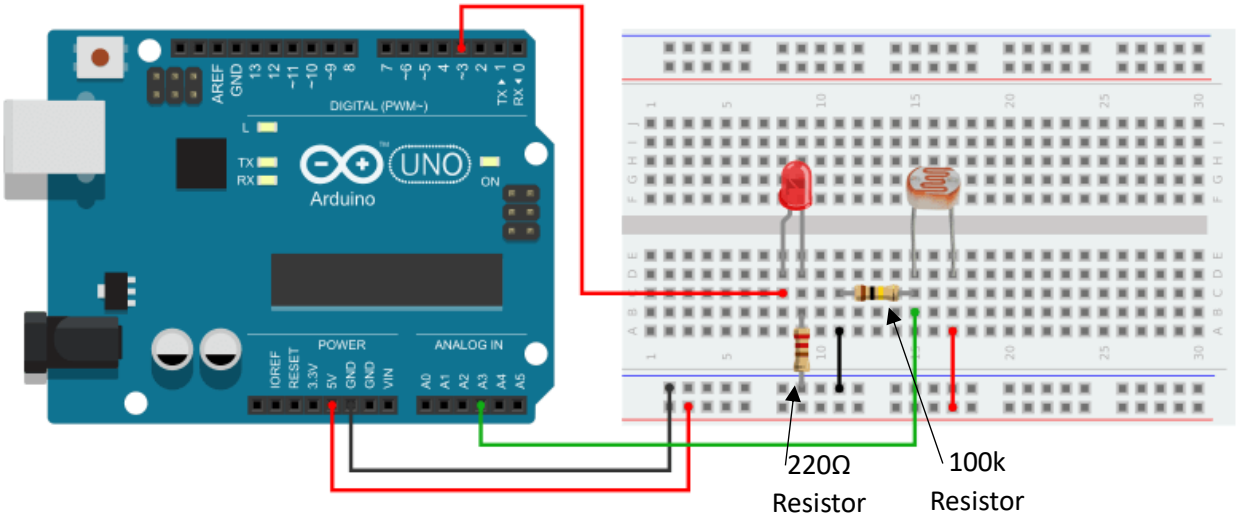


Figure 14: Wiring Diagram for Exercise 6

Sample Code

```
const int ldrPin = A2;    // select the input pin for the potentiometer
const int ledPin = 11;    // select the pin for the LED
int ldrValue = 0;   // variable to store the value coming from the sensor

void setup() {
  // declare the ledPin as an OUTPUT:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  ldrValue=analogRead(ldrPin);
  if(ldrValue > 20)
  digitalWrite(ledPin,HIGH);
  else
  digitalWrite(ledPin,LOW);
  delay(100);
}
```

9. Exercise 7: LM35 Temperature Sensor

The LM35 sensor is moderately precise and its robust construction makes it suitable for various environmental conditions. Additionally, you don't need any external component to calibrate this circuit and it has a typical accuracy of ±0.5°C at room temperature and ±1°C over a full −55°C to +155°C temperature range. It has an operating voltage of 4V to 30V and consumes 60-uA current while it's in working state, this also makes it perfect for battery-powered applications.

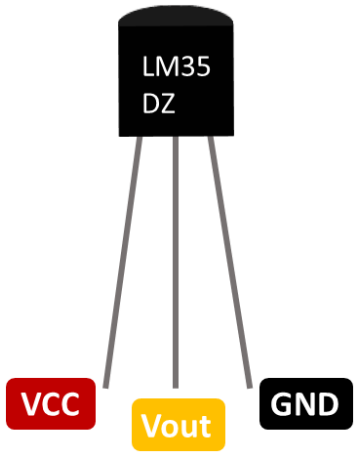


Figure 15: LM35 Temperature Sensor

9.1. Working Principle & Calculation of Temperature from Thermistor

The LM35 temperature sensor uses the basic principle of a diode to measure known temperature value. As we all know from semiconductor physics, as the temperature increases the voltage across a diode increases at a known rate. By accurately amplifying the voltage change, we can easily generate a voltage signal that is directly proportional to the surrounding temperature. The screenshot below shows the internal schematic of LM35 temperature sensor IC according to the datasheet.

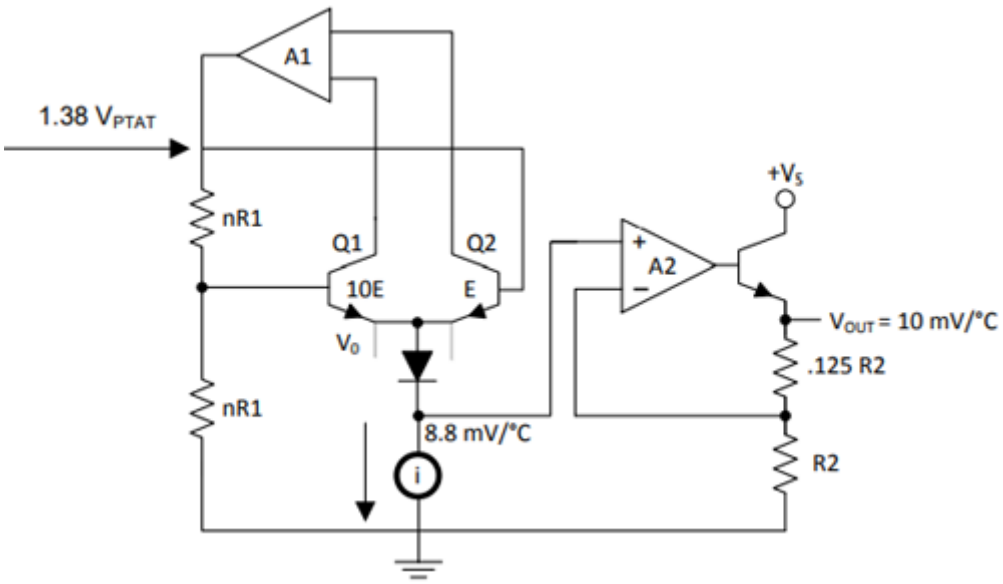


Figure 16: Inside schematic of LM35

In practice, this diode that they are using to measure the temperature is not actually a PN Junction diode but it's a diode-connected transistor. That is why the relationship between the forward voltage and the transistor is so linear. The temperature coefficient vs collector current graph below gives you a better understanding of the process.

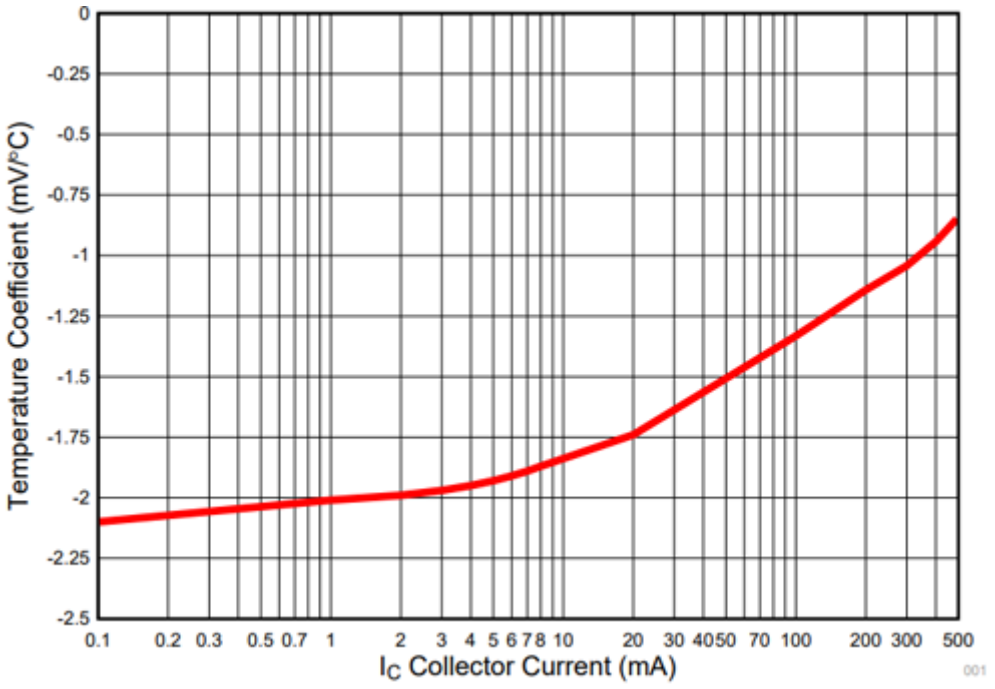


Figure 17: The temperature coefficient vs collector current graph

Required components for the Exercise 7:

- LM35 Sensor
- Male to Male 20cm Jumper Cable x 5
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

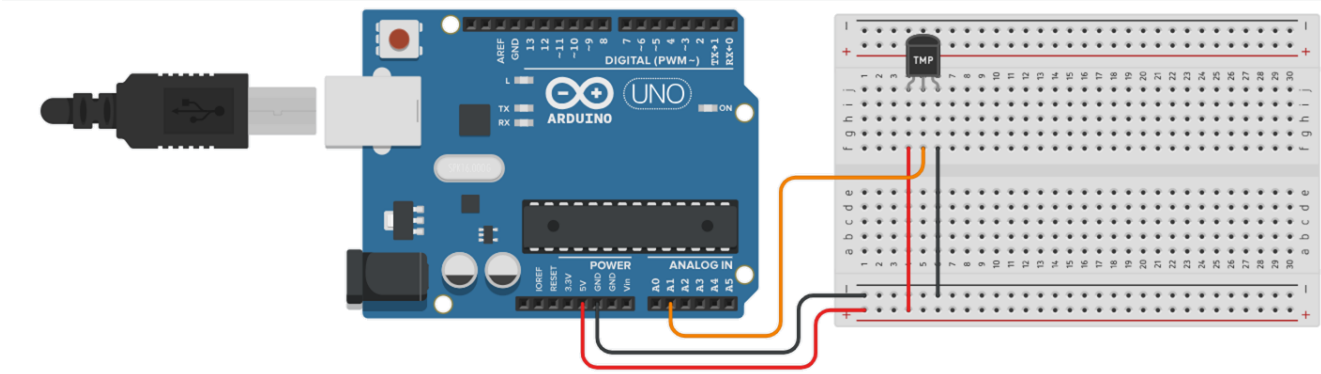


Figure 18: Wiring Diagram for Exercise 7

Sample Code

```
// Define the analog pin the LM35 sensor is connected to
const int sensorPin = A1;

void setup() {
  Serial.begin(9600); // Start serial communication at 9600 baud rate
}

void loop() {
  int sensorValue = analogRead(sensorPin); // Read the analog value from the
sensor
  float voltage = (sensorValue / 1023.0) * 5.0; // Convert analog value to
voltage (5V Arduino)
  float temperatureC = (voltage - 0.5) * 100.0; // Convert voltage to temperature
in Celsius

  Serial.print("Voltage: ");
  Serial.print(voltage);
  Serial.print("V, Temperature: ");
  Serial.print(temperatureC);
  Serial.println("°C");
  delay(1000); // Delay between temperature readings (adjust as needed)
}
```

Exercise 8: Obstacle Avoidance Sensor with Buzzer

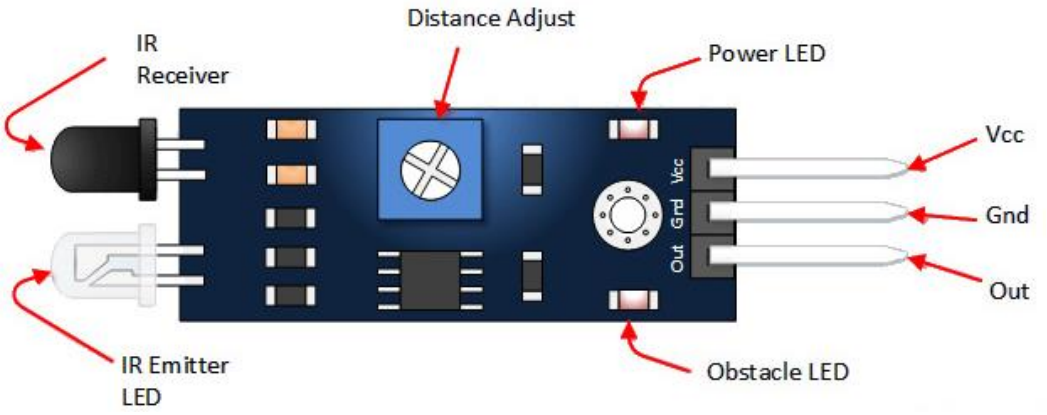


Figure 19: Obstacle Sensor

The IR LED emits infrared light, which bounces off objects and gets reflected. The IR receiver (photo diode) detects the reflected infrared light waves. The current flow through the photo diode varies based on the intensity of the received IR light.

The LM393 comparator IC compares the voltage from the variable resistor (used for adjusting detection distance) and the voltage from the IR receiver diode.

If an obstacle is detected within the adjustable detection range, the voltage levels change, and the output LED on the IR sensor module turns on, indicating the detection of an obstacle. Conversely, if no obstacle is within the detection range, the output LED remains off, signaling the absence of an obstacle.

Modes of Operation:

Digital Output Sensor: In this mode, the IR sensor module provides a simple on/off signal (high/low) through the output LED, indicating the presence or absence of an obstacle.

Analog Output Sensor: By adjusting the Arduino wiring and code, the sensor's analog signals can be interpreted to provide distance or proximity information in a more continuous and graded manner.

Usage with Arduino:

The IR sensor module can be interfaced with an Arduino board to detect obstacles or measure distances. Depending on the application, the Arduino can interpret the sensor's output as digital or analog signals to take specific actions or provide distance-related information in projects like obstacle avoidance robots, automated door systems, or object detection setups.

Required components for the Exercise 8:

- Obstacle Sensor
- Active Buzzer
- Male to Male 20cm Jumper Cable x 5
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

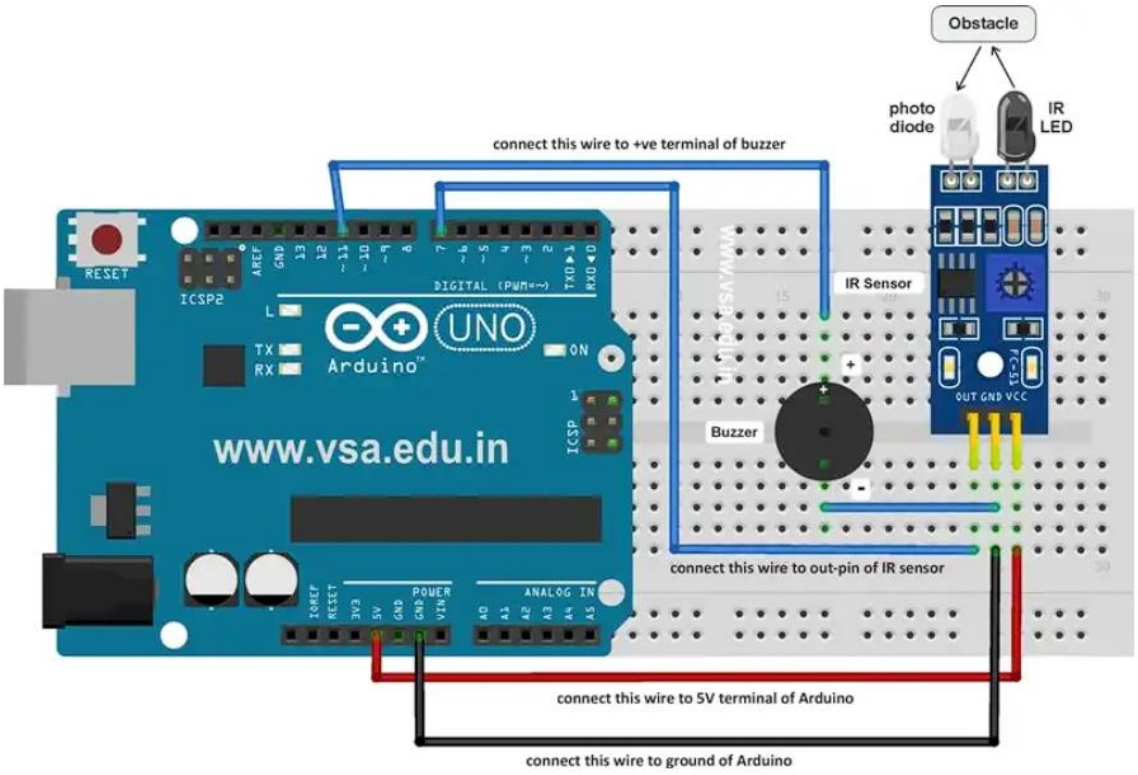


Figure 20: Wiring Diagram for Exercise 8

Sample Code

```
#define IR_SENSOR_PIN 7 // Pin connected to the IR obstacle sensor
#define BUZZER_PIN 11 // Pin for the active buzzer

void setup() {
  pinMode(IR_SENSOR_PIN, INPUT);
  pinMode(BUZZER_PIN, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  int obstacleDetected = digitalRead(IR_SENSOR_PIN);

  if (obstacleDetected == HIGH) {
    // Obstacle detected, activate the buzzer
    digitalWrite(BUZZER_PIN, HIGH); // Turn on the buzzer
    delay(500); // Buzz for 500 milliseconds
    digitalWrite(BUZZER_PIN, LOW); // Turn off the buzzer
  }

  delay(100); // Add a short delay between readings
}
```

10. Exercise 9: Generating a Tone using a Passive Buzzer

Figure 21: Passive Buzzer

10.1. Working Principle of Passive Buzzer

There are two types of buzzers, active buzzers, and passive buzzers. Most of the active buzzer works at a voltage range of 3.3V – 5V and generate only one sound frequency. It can only generate a sound of fixed frequency when you provide the required voltage to it.

On the other hand, you have a passive buzzer. Passive buzzers can generate a sound of a wide frequency range (> 31Hz). It needs a fixed frequency signal to generate a specific tone. This tone can be changed by changing the input signal frequency. We can use a PWM signal or the tone() function in Arduino to generate this type of input signal and generate a tone. Thus we can get more control over the buzzer tone when we use the tone() function.

The basic syntax for the tone function is given below.

- tone(pinNumber, frequency) and
- tone(pinNumber, frequency, duration)

Where pinNumber is the Arduino pin number on which we generate the tone. frequency determines the frequency of the tone in hertz. and duration determines the duration of the tone in milliseconds.

So if we want to generate 500 Hertz tone for 500ms at pin no 9, we will use tone(9, 500, 500)

A call to the noTone() function will stop the tone. The syntax for the noTone() is noTone(pinNumber) where pinNumber represents the pin number that the buzzer is attached.

Required components for the Exercise 9:

- Passive Buzzer
- Male to Male 20cm Jumper Cable x 2
- Arduino UNO Board
- USB 2.0 Cable Type A/B

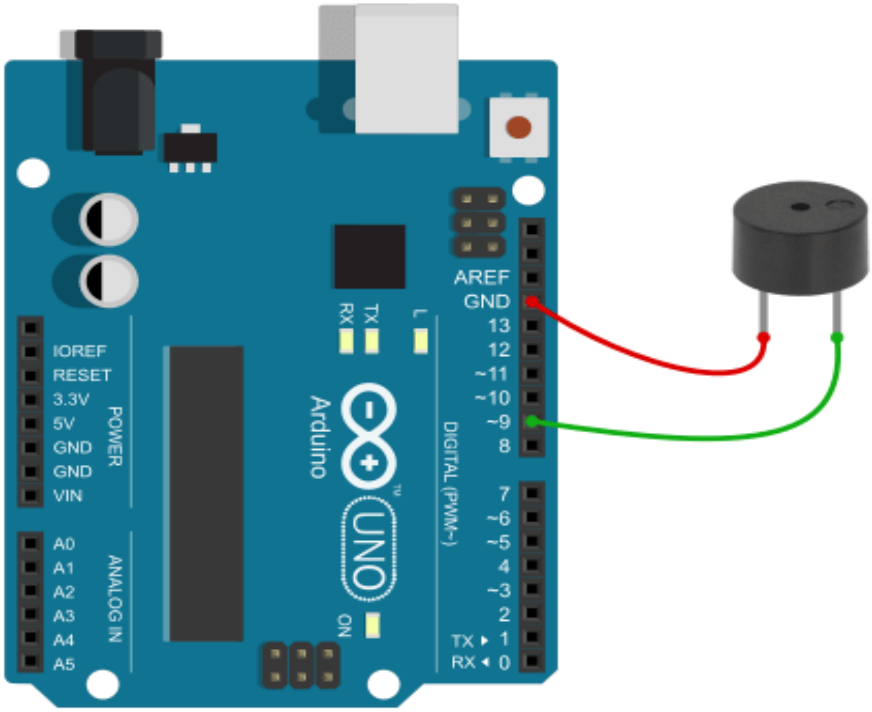


Figure 22: Wiring Diagram for Exercise 9

Sample Code

```
/*
  Game of Thrones
  Connect a piezo buzzer or speaker to pin 11 or select a new pin.
  More songs available at https://github.com/robsoncouto/arduino-
songs

                                            Robson Couto, 2019
*/
#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
```

```
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
#define NOTE_E5  659
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
#define NOTE_FS7 2960
```

```
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978
#define REST      0


// change this to make the song slower or faster
int tempo = 85;

// change this to whichever pin you want to use
int buzzer = 9;

// notes of the moledy followed by the duration.
// a 4 means a quarter note, 8 an eighteenth , 16 sixteenth, so on
// !!negative numbers are used to represent dotted notes,
// so -4 means a dotted quarter note, that is, a quarter plus an eighteenth!!
int melody[] = {

  // Game of Thrones
  // Score available at https://musescore.com/user/8407786/scores/2156716

  NOTE_G4,8, NOTE_C4,8, NOTE_DS4,16, NOTE_F4,16, NOTE_G4,8, NOTE_C4,8,
NOTE_DS4,16, NOTE_F4,16, //1
  NOTE_G4,8, NOTE_C4,8, NOTE_DS4,16, NOTE_F4,16, NOTE_G4,8, NOTE_C4,8,
NOTE_DS4,16, NOTE_F4,16,
  NOTE_G4,8, NOTE_C4,8, NOTE_E4,16, NOTE_F4,16, NOTE_G4,8, NOTE_C4,8, NOTE_E4,16,
NOTE_F4,16,
  NOTE_G4,8, NOTE_C4,8, NOTE_E4,16, NOTE_F4,16, NOTE_G4,8, NOTE_C4,8, NOTE_E4,16,
NOTE_F4,16,
  NOTE_G4,-4, NOTE_C4,-4,//5

  NOTE_DS4,16, NOTE_F4,16, NOTE_G4,4, NOTE_C4,4, NOTE_DS4,16, NOTE_F4,16, //6
  NOTE_D4,-1, //7 and 8
  NOTE_F4,-4, NOTE_AS3,-4,
  NOTE_DS4,16, NOTE_D4,16, NOTE_F4,4, NOTE_AS3,-4,
  NOTE_DS4,16, NOTE_D4,16, NOTE_C4,-1, //11 and 12

  //repeats from 5
  NOTE_G4,-4, NOTE_C4,-4,//5
```

```
  NOTE_DS4,16, NOTE_F4,16, NOTE_G4,4, NOTE_C4,4, NOTE_DS4,16, NOTE_F4,16, //6
  NOTE_D4,-1, //7 and 8
  NOTE_F4,-4, NOTE_AS3,-4,
  NOTE_DS4,16, NOTE_D4,16, NOTE_F4,4, NOTE_AS3,-4,
  NOTE_DS4,16, NOTE_D4,16, NOTE_C4,-1, //11 and 12
  NOTE_G4,-4, NOTE_C4,-4,
  NOTE_DS4,16, NOTE_F4,16, NOTE_G4,4,  NOTE_C4,4, NOTE_DS4,16, NOTE_F4,16,

  NOTE_D4,-2,//15
  NOTE_F4,-4, NOTE_AS3,-4,
  NOTE_D4,-8, NOTE_DS4,-8, NOTE_D4,-8, NOTE_AS3,-8,
  NOTE_C4,-1,
  NOTE_C5,-2,
  NOTE_AS4,-2,
  NOTE_C4,-2,
  NOTE_G4,-2,
  NOTE_DS4,-2,
  NOTE_DS4,-4, NOTE_F4,-4,
  NOTE_G4,-1,

  NOTE_C5,-2,//28
  NOTE_AS4,-2,
  NOTE_C4,-2,
  NOTE_G4,-2,
  NOTE_DS4,-2,
  NOTE_DS4,-4, NOTE_D4,-4,
  NOTE_C5,8, NOTE_G4,8, NOTE_GS4,16, NOTE_AS4,16, NOTE_C5,8, NOTE_G4,8,
NOTE_GS4,16, NOTE_AS4,16,
  NOTE_C5,8, NOTE_G4,8, NOTE_GS4,16, NOTE_AS4,16, NOTE_C5,8, NOTE_G4,8,
NOTE_GS4,16, NOTE_AS4,16,

  REST,4, NOTE_GS5,16, NOTE_AS5,16, NOTE_C6,8, NOTE_G5,8, NOTE_GS5,16,
NOTE_AS5,16,
  NOTE_C6,8, NOTE_G5,16, NOTE_GS5,16, NOTE_AS5,16, NOTE_C6,8, NOTE_G5,8,
NOTE_GS5,16, NOTE_AS5,16,
};

// sizeof gives the number of bytes, each int value is composed of two bytes (16
bits)
// there are two values per note (pitch and duration), so for each note there are
four bytes
int notes = sizeof(melody) / sizeof(melody[0]) / 2;

// this calculates the duration of a whole note in ms
```

```
int wholenote = (60000 * 4) / tempo;

int divider = 0, noteDuration = 0;

void setup() {
  // iterate over the notes of the melody.
  // Remember, the array is twice the number of notes (notes + durations)
  for (int thisNote = 0; thisNote < notes * 2; thisNote = thisNote + 2) {

    // calculates the duration of each note
    divider = melody[thisNote + 1];
    if (divider > 0) {
      // regular note, just proceed
      noteDuration = (wholenote) / divider;
    } else if (divider < 0) {
      // dotted notes are represented with negative durations!!
      noteDuration = (wholenote) / abs(divider);
      noteDuration *= 1.5; // increases the duration in half for dotted notes
    }

    // we only play the note for 90% of the duration, leaving 10% as a pause
    tone(buzzer, melody[thisNote], noteDuration * 0.9);

    // Wait for the specief duration before playing the next note.
    delay(noteDuration);

    // stop the waveform generation before the next note.
    noTone(buzzer);
  }
}

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

11. Exercise 10: Arduino Tone Keyboard

This exercise involves creating a tone keyboard using six pushbuttons and a passive buzzer with an Arduino. This setup allows users to generate different tones by pressing the buttons.

To accomplish this, you'd connect each pushbutton to an analog pin (Analog pins also can be used as digital pins, this is an example for that) on the Arduino and connect the passive buzzer to a digital pin. Then, when a button is pressed, the corresponding tone will be played through the buzzer.

Required components for the Exercise 10:

- Passive Buzzer x 1
- Male to Male 20cm Jumper Cable x 10
- 100 Ohm Resistor x 1
- 10k Resistor x 6
- Pushbutton x 6
- Arduino Breadboard
- Arduino UNO Board
- USB 2.0 Cable Type A/B

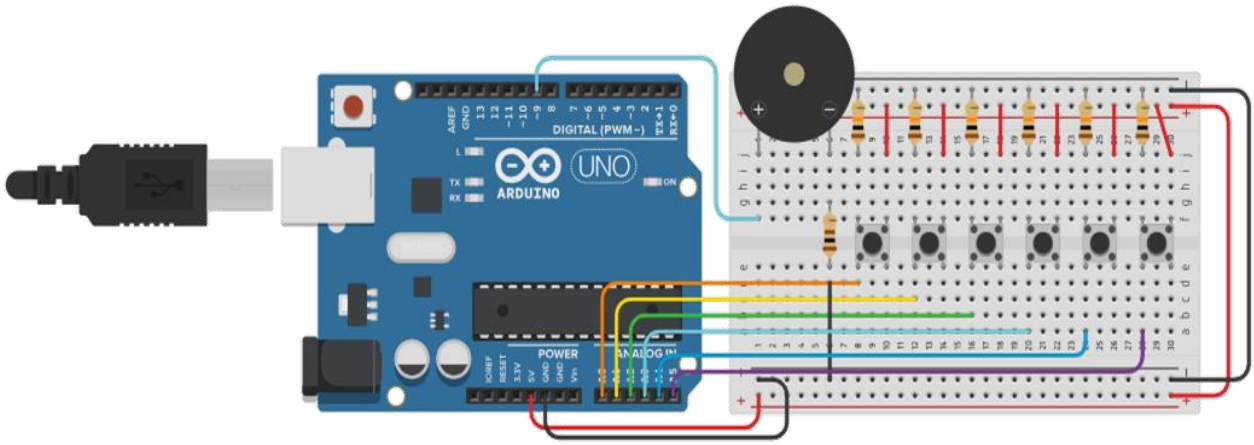


Figure 23: Wiring Diagram for Exercise 10

Tip: Place resistors and pushbuttons to use minimum number of jumper wires, above diagram is for demonstration.

Sample Code

```
int pos = 0;

void setup()
{
  pinMode(9, OUTPUT);
  pinMode(A0, INPUT);
  pinMode(A1, INPUT);
  pinMode(A2, INPUT);
  pinMode(A3, INPUT);
  pinMode(A4, INPUT);
  pinMode(A5, INPUT);
}

void loop()
{
  // if button press on A0 is detected
  if (digitalRead(A0) == HIGH) {
    tone(9, 262, 100); // play tone (C4 = 262 Hz)
  }
  // if button press on A1 is detected
  if (digitalRead(A1) == HIGH) {
    tone(9, 294, 100); // play tone (D4 = 294 Hz)
  }
  // if button press on A2 is detected
  if (digitalRead(A2) == HIGH) {
    tone(9, 324, 100); // play tone (E4 = 324 Hz)
  }
  // if button press on A3 is detected
  if (digitalRead(A3) == HIGH) {
    tone(9, 344, 100); // play tone (F4 = 344 Hz)
  }
  // if button press on A4 is detected
  if (digitalRead(A4) == HIGH) {
    tone(9, 386, 100); // play tone (G4 = 386 Hz)
  }
  // if button press on A5 is detected
  if (digitalRead(A5) == HIGH) {
    tone(9, 433, 100); // play tone (A4 = 433 Hz)
  }
  delay(10); // Delay a little bit to improve simulation performance
}
```

12. Exercise 11: 7 Segment Display

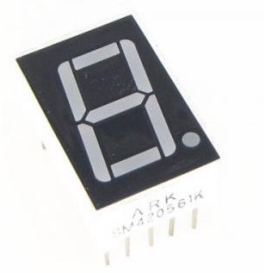


Figure 24: 7 Segment Display

There are two main types of 7 Segment LED Displays:

1. Common Anode: In this type of display, a positive voltage source is shared by the display segments. Segments are lit by setting their individual pins to Ground.
2. Common Cathode: A common path to ground is shared by the segments. Individual segments are lit when their pin receives sufficient positive voltage.

The idea behind having a common cathode or anode display is that by sharing either positive voltage or a path to ground, fewer pins are necessary than if every segment had its own pair of cathode and anode pins. Because most displays also include a decimal point, colon or apostrophe, there are usually eight segments, seven for the digit and one for punctuation. With one common pin, there should be at least 9 pins.

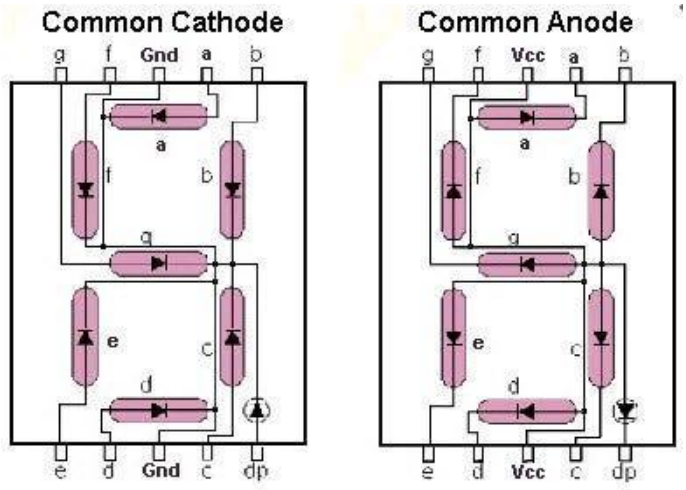


Figure 25: Pin Diagram

Required components for the Exercise 11:

- Common Cathode 7 Segment Display
- Male to Male 20cm Jumper Cable x 10
- 220 Ohm Resistor x 7
- 10k Resistor x 1
- Pushbutton x 1
- Arduino Breadboard
- Arduino UNO Board
- USB 2.0 Cable Type A/B

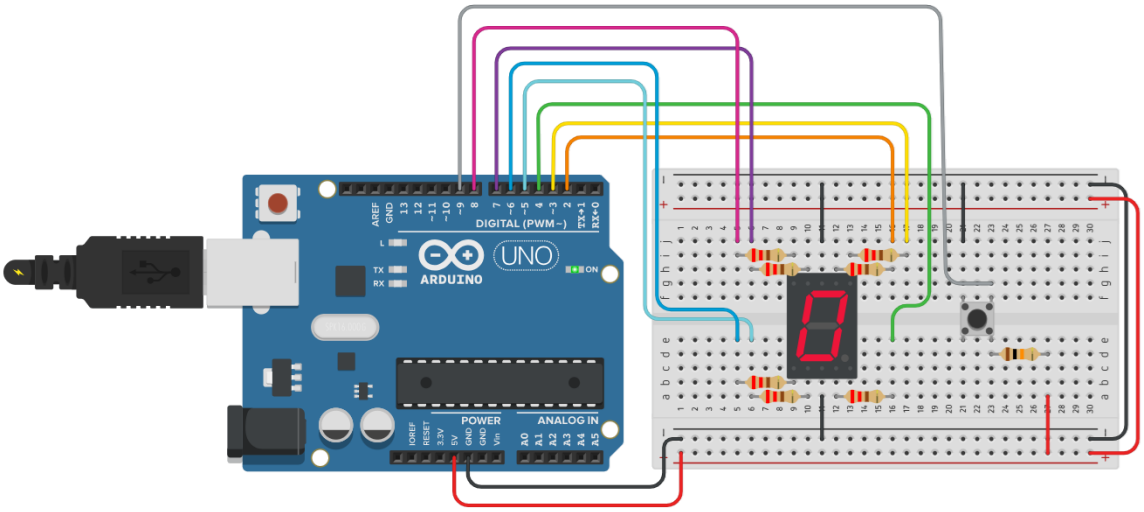


Figure 26: Wiring Diagram for Exercise 11

Tip: Place resistors to use minimum number of jumper wires, above diagram is for demonstration.

Sample Code

```
// Define the pins for seven segment display
const int segA = 2;
const int segB = 3;
const int segC = 4;
const int segD = 5;
const int segE = 6;
const int segF = 7;
const int segG = 8;

// Define the pin for the push button
const int buttonPin = 9;

int counter = 0; // Counter variable to track the count
bool buttonState = LOW; // Store the current state of the button
bool lastButtonState = LOW; // Store the previous state of the button
unsigned long lastDebounceTime = 0; // Store the last time the button state
changed
unsigned long debounceDelay = 50; // Debounce time in milliseconds

// Define the patterns for each digit (0-9) in a common cathode 7-segment display
int digit[10][7] = {
  {1, 1, 1, 1, 1, 1, 0}, // 0
  {0, 1, 1, 0, 0, 0, 0}, // 1
  {1, 1, 0, 1, 1, 0, 1}, // 2
  {1, 1, 1, 1, 0, 0, 1}, // 3
  {0, 1, 1, 0, 0, 1, 1}, // 4
  {1, 0, 1, 1, 0, 1, 1}, // 5
  {1, 0, 1, 1, 1, 1, 1}, // 6
  {1, 1, 1, 0, 0, 0, 0}, // 7
  {1, 1, 1, 1, 1, 1, 1}, // 8
  {1, 1, 1, 1, 0, 1, 1}  // 9
};

void setup() {
  // Set pins as outputs for seven segment display
  pinMode(segA, OUTPUT);
  pinMode(segB, OUTPUT);
  pinMode(segC, OUTPUT);
  pinMode(segD, OUTPUT);
  pinMode(segE, OUTPUT);
  pinMode(segF, OUTPUT);
  pinMode(segG, OUTPUT);
```

```
  // Set the button pin as input
  pinMode(buttonPin, INPUT);

  // Initialize the seven segment display to show 0
  displayDigit(0);
}

void loop() {
  // Read the state of the button
  int reading = digitalRead(buttonPin);

  // Check if the button state has changed
  if (reading != lastButtonState) {
    lastDebounceTime = millis(); // Reset the debounce timer
  }

  // Check if the button state has remained the same for the debounce delay
  if (millis() - lastDebounceTime > debounceDelay) {
    if (reading != buttonState) {
      buttonState = reading; // Update the button state

      // If the button is pressed, increment the counter and display the new
count
      if (buttonState == HIGH) {
        counter = (counter + 1) % 10;
        displayDigit(counter);
      }
    }
  }

  // Update lastButtonState
  lastButtonState = reading;
}

// Function to display a digit on the seven segment display
void displayDigit(int digitToDisplay) {
  for (int i = 2; i <= 8; i++) {
    digitalWrite(i, digit[digitToDisplay][i - 2]);
  }
}
```

13. Exercise 12: Water Level Sensor

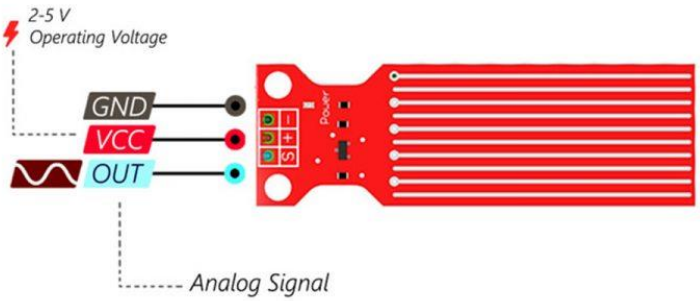


Figure 27: Water Level Sensor

13.1. Working Principle of the Water Level Sensor

The water level sensor operates based on the principle of variable resistance. It consists of a series of exposed conductors arranged in parallel. This configuration effectively functions as a variable resistor, and its resistance changes in response to the water level present in the tank or container it is placed in.

The sensor's conductors react to the presence or absence of water. When the sensor is submerged in water, the conductivity between these conductors improves significantly. This increased conductivity leads to a decrease in resistance across the sensor. In simpler terms, as more of the sensor is submerged in water, the electrical path between the conductors becomes more conductive, allowing current to flow more easily. This results in a lower overall resistance across the sensor.

Conversely, when the sensor is less submerged or not in contact with water, the conductivity decreases, and the resistance across the sensor increases. This happens because without water, the electrical connection between the conductors becomes weaker, impeding the flow of current and thereby raising the overall resistance.

The output from the water level sensor is directly related to this resistance. It generates a voltage signal that is proportional to the resistance of the water present. This signal can be interpreted by microcontroller or other electronics to determine the water level, as higher water levels correspond to lower resistance

and vice versa. This variable voltage signal can then be processed to provide an indication of the water level in the tank or container being monitored.

Required components for the Exercise 12:

- Green LED x 1
- Red LED x 1
- Yellow LED x 1
- Male to Male 20cm Jumper Cable x 7
- 220 Ohm Resistor x 3
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

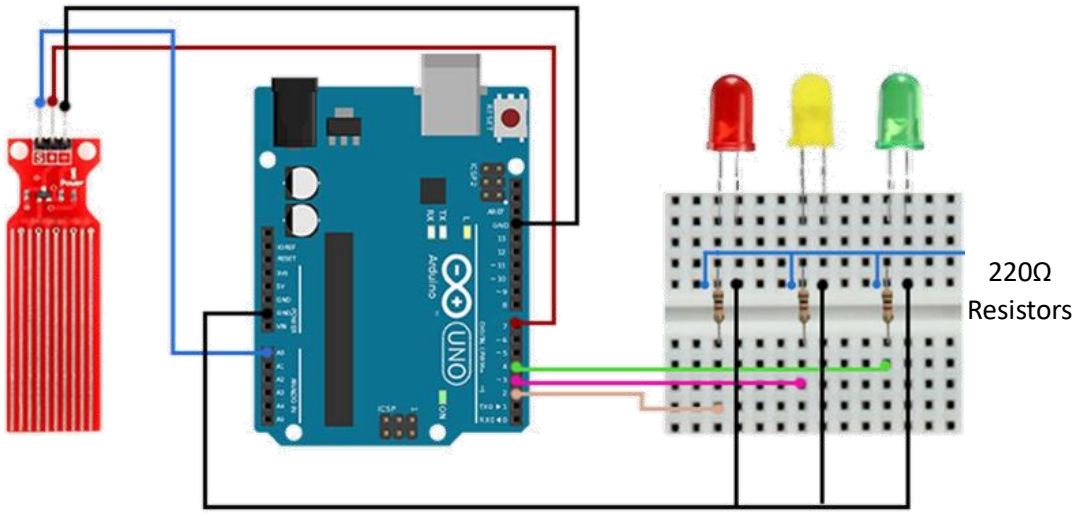


Figure 28: Wiring Diagram for Exercise 12

Sample Code

```
/* Change these values based on your calibration values */
int lowerThreshold = 420;
int upperThreshold = 520;

// Sensor pins
#define sensorPower 7
#define sensorPin A0

// Value for storing water level
int val = 0;

// Declare pins to which LEDs are connected
int redLED = 2;
int yellowLED = 3;
int greenLED = 4;

void setup() {
    Serial.begin(9600);
    pinMode(sensorPower, OUTPUT);
    digitalWrite(sensorPower, LOW);

    // Set LED pins as an OUTPUT
    pinMode(redLED, OUTPUT);
    pinMode(yellowLED, OUTPUT);
    pinMode(greenLED, OUTPUT);

    // Initially turn off all LEDs
    digitalWrite(redLED, LOW);
    digitalWrite(yellowLED, LOW);
    digitalWrite(greenLED, LOW);
}

void loop() {
    int level = readSensor();

    if (level == 0) {
        Serial.println("Water Level: Empty");
        digitalWrite(redLED, LOW);
        digitalWrite(yellowLED, LOW);
        digitalWrite(greenLED, LOW);
    }
    else if (level > 0 && level <= lowerThreshold) {
        Serial.println("Water Level: Low");
```

```
        digitalWrite(redLED, HIGH);
        digitalWrite(yellowLED, LOW);
        digitalWrite(greenLED, LOW);
    }
    else if (level > lowerThreshold && level <= upperThreshold) {
        Serial.println("Water Level: Medium");
        digitalWrite(redLED, LOW);
        digitalWrite(yellowLED, HIGH);
        digitalWrite(greenLED, LOW);
    }
    else if (level > upperThreshold) {
        Serial.println("Water Level: High");
        digitalWrite(redLED, LOW);
        digitalWrite(yellowLED, LOW);
        digitalWrite(greenLED, HIGH);
    }
    delay(1000);
}

//This is a function used to get the reading
int readSensor() {
    digitalWrite(sensorPower, HIGH);
    delay(10);
    val = analogRead(sensorPin);
    digitalWrite(sensorPower, LOW);
    return val;
}
```

14. Exercise 13: DHT11 Temperature & Humidity Sensor

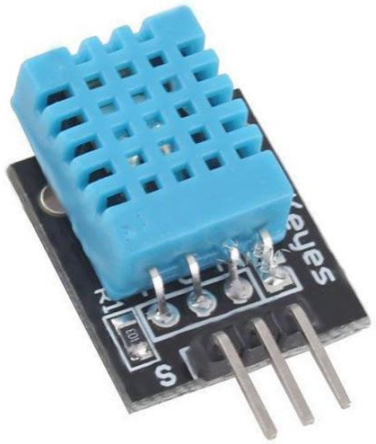


Figure 29: DHT11 Sensor Module

DHT11 Temperature & Humidity Sensor features a temperature & humidity sensor complex with a calibrated digital signal output. By using the exclusive digital-signal-acquisition technique and temperature & humidity sensing technology, it ensures high reliability and excellent long-term stability. This sensor includes a resistive-type humidity measurement component and an NTC temperature measurement component and connects to a high performance 8-bit microcontroller, offering excellent quality, fast response, anti-interference ability and cost-effectiveness.

14.1. Specifications

- Operating Voltage: 3.5V to 5.5V
- Operating current: 0.3mA (measuring) 60uA (standby)
- Output: Serial data
- Temperature Range: 0°C to 50°C
- Humidity Range: 20% to 90%
- Resolution: Temperature and Humidity both are 16-bit
- Accuracy: ±1°C and ±1%

14.2. Working Principle of the DHT11 Sensor

The DHT11 Sensor is factory calibrated and outputs serial data and hence it is easy to set it up. The connection diagram for this sensor is shown below.

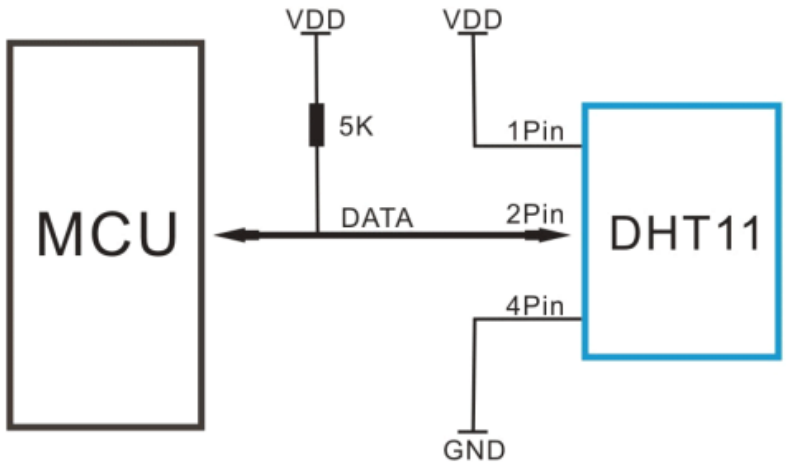


Figure 30: Connection diagram of DHT11

As you can see the data pin is connected to an I/O pin of the MCU and a 5k pull-up resistor is used. This data pin outputs the value of both temperature and humidity as serial data. If you are trying to interface DHT11 with Arduino, then there are ready-made libraries for it which will give you a quick start.

If you are trying to interface it with some other MCU, then the datasheet given below will come in handy. The output given out by the data pin will be in the order of 8bit humidity integer data + 8bit the Humidity decimal data +8-bit temperature integer data + 8bit fractional temperature data +8-bit parity bit. To request the DHT11 module to send these data the I/O pin must be momentarily made low and then held high as shown in the timing diagram below.

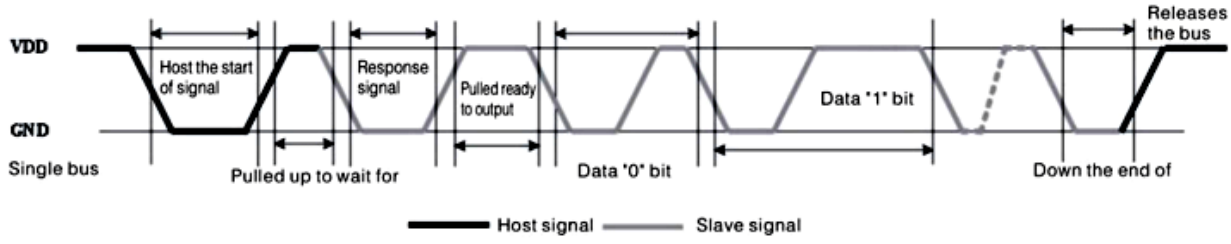


Figure 31: Data signal of DHT11

The duration of each host signal is explained in the DHT11 datasheet, with neat steps and illustrative timing diagrams.

Required components for the Exercise 13:

- DHT11 Sensor
- Yellow LED x 1
- Male to Male 20cm Jumper Cable x 3
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

Now that we understand how a DHT11 Sensor works, we can connect all of the necessary wires to Arduino and write the code to extract all of the data from the sensor. The circuit schematic for integrating the DHT11 sensor module with Arduino is shown in the figure below. In this exercise we measure the room temperature and humidity using DHT11 sensor and display the values in the serial monitor.

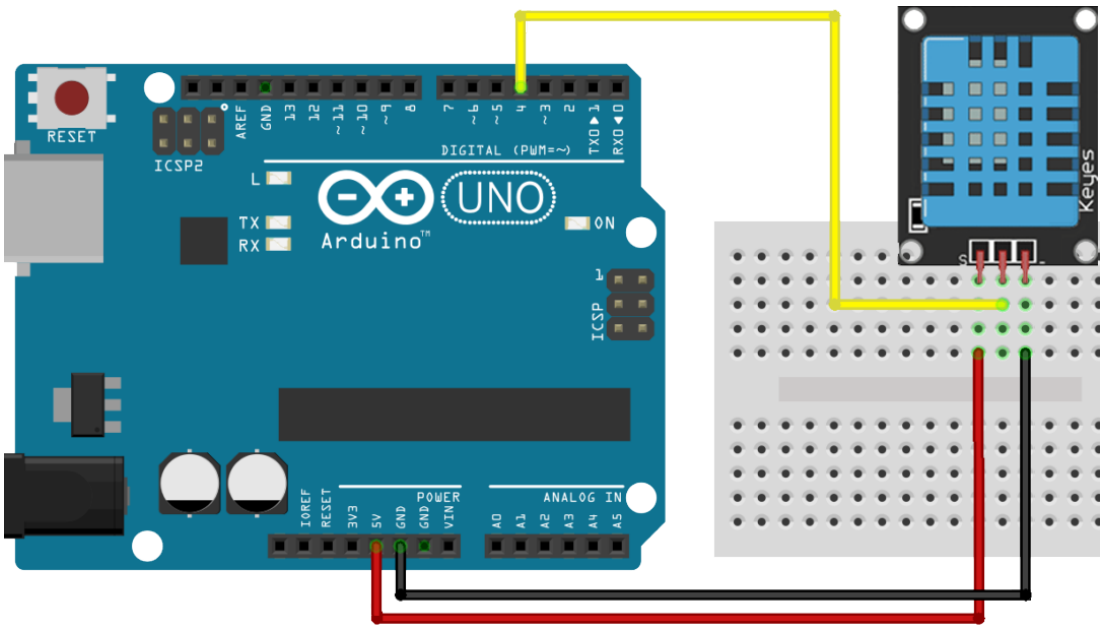


Figure 32: Wiring Diagram for Exercise 13

Sample Code

```
#include <Adafruit_Sensor.h>
#include <DHT.h>
#include <DHT_U.h>

#define DHTTYPE DHT11   // DHT 11

#define DHTPIN 2

DHT_Unified dht(DHTPIN, DHTTYPE);

uint32_t delayMS;

void setup() {

  Serial.begin(9600);
  dht.begin();
  sensor_t sensor;
  delayMS = sensor.min_delay / 1000;
}


void loop()

{
  sensors_event_t event;

  dht.temperature().getEvent(&event);

  Serial.print(F("Temperature: "));
  Serial.print(event.temperature);
  Serial.println(F("°C"));
  dht.humidity().getEvent(&event);
  Serial.print(F("Humidity: "));
  Serial.print(event.relative_humidity);
  Serial.println(F("%"));

  delay(delayMS);
}
```

15. Exercise 14: VU Meter using Sound Detection Sensor

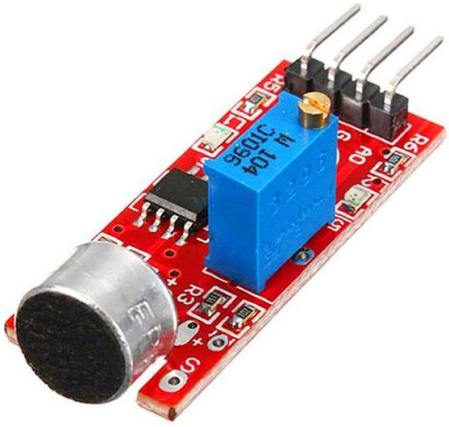


Figure 33: Sound Detection Sensor

The Sound Detection Sensor module consists of a capacitance sensitive microphone (50Hz-10kHz) and an amplification circuit. This module converts sound waves to electrical signals. It detects the sound with the help of a microphone and then feeds this sound to processing circuitry which consists of an LM393 operational amplifier. It also consists of a potentiometer which is used for setting the sound level and by setting this sound level the output of this sound sensor module could be easily controlled. Similarly, the output of this sensor could be checked by connecting the LED or any other device at output pins.

15.1. Working Principle of Sound Detection Sensor

The working of the sound sensor module is very simple and easy to understand, the main component in this module is a condenser microphone. The microphone gives out only analog signals when a sound wave hits the diaphragm of the sensor, this analog signal gets processed by the op-amp and we get the digital output.

The main component of a sound sensor is a microphone. There are many different types of microphones, like Carbon Microphone, Fiber Optic Microphone, Ribbon Microphone, and Laser Microphone, but the sound sensor module we are using has a condenser microphone. An image of the sound sensor module is shown below,

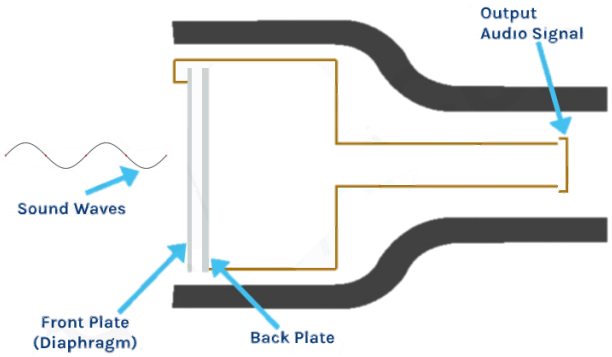


Figure 34: Inside of a condenser mic

As you can see from the image, a condenser microphone consists of two charged metal plates. The first plate is called the diaphragm, and the second plate is the backplate of the microphone. These two plates together form a capacitor. When a sound wave hits the diaphragm of the microphone the diaphragm starts to vibrate, and the distance between the two plates changes. The movement of the diaphragm and the change in spacing produces the electrical signal that corresponds to the sound that's picked up by the microphone and this signal then gets processed by the onboard op-amp. This module also has two built-in onboard LEDs, one of which lights up when power is applied to the board and the other one lights up when the incoming audio signal exceeds the threshold value set by the potentiometer.

15.2. Pin Configuration

The Sound sensor module has 4 pins VCC, GND, Digital Out, and Analog Out. We can either use the AO pin as an output for analog reading or the DO pin as an output for digital readout. The Sound sensor pinout is as follows:

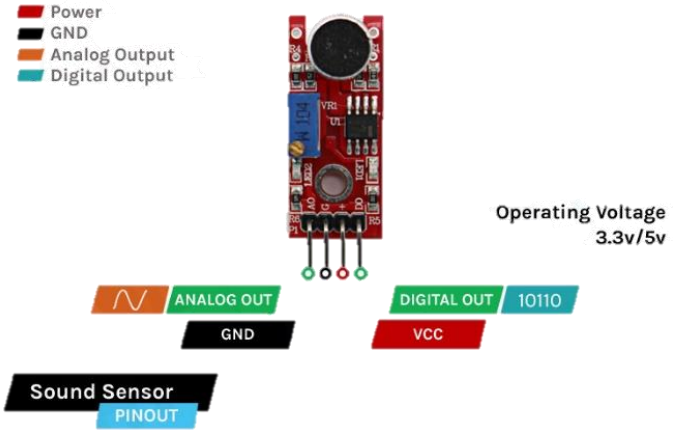


Figure 35: Pin Configuration

- VCC: Is the power supply pin of the Sound Sensor that can be connected to 3.3V or 5V of the supply. But note that the analog output will vary depending upon the provided supply voltage.
- GND: Is the ground pin of the Sound Sensor module and it should be connected to the ground pin of the Arduino.
- DOUT: Is the Digital output pin of the board, low output indicates that no sound is detected by the sensor, and high indicates that the sensor has detected sound.
- AOUT: Is the Analog output pin of the board that will give us an analog reading directly from the Sound sensor.

Required components for the Exercise 14:

- Sound Detection Sensor
- Red LEDs x 5
- 14 Male to Male 20cm Jumper Cables x 9
- 5 220 Ohm Resistors x 5
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

It is quite simple to connect the Sound Sensor to the microcontroller. As we all know, the sensor produces an analog signal that is simple to process. We can use the Arduino's ADC to analyze the signal and light up some LEDs to display the strength of the sound received by the microcontroller. The circuit is also quite easy; we just connected the Arduino board's VCC and ground to the sensor module, then we used GPIO2 through GPIO6 to connect the LEDs. The ground is shared by the LEDs and the Sensor. A hardware picture of the configuration is provided below.

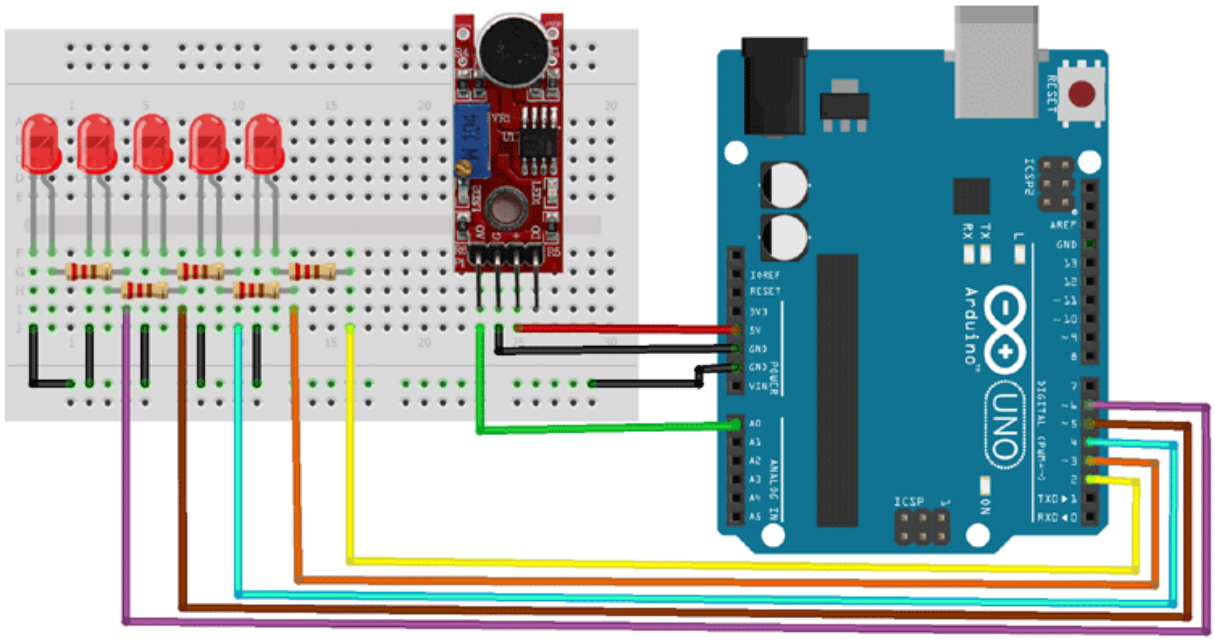


Figure 36: Wiring Diagram for Exercise 14

Tip: Place resistors and LEDs to use minimum number of jumper wires, above diagram is for demonstration.

Sample Code

```
/* Arduino pins where the LED is attached*/

#define LED_1 2
#define LED_2 3
#define LED_3 4
#define LED_4 5
#define LED_5 6

#define sensorPin A0  // Analog input pin that the Sensor is attached to

/* boolean variables to hold the status of the pins*/

bool ledPin1Status;
bool ledPin2Status;
bool ledPin3Status;
bool ledPin4Status;
bool ledPin5Status;

void setup() {
```

```
  pinMode(LED_1, OUTPUT);
  pinMode(LED_2, OUTPUT);
  pinMode(LED_3, OUTPUT);
  pinMode(LED_4, OUTPUT);
  pinMode(LED_5, OUTPUT);
  pinMode(sensorPin, INPUT);

  Serial.begin(9600);  // initialize serial communications at 9600 bps:
}

void loop() {

  int sensorValue = analogRead(sensorPin);

  Serial.println(sensorValue);
  if (sensorValue > 555)
    ledPin1Status = 1;
  if (sensorValue > 558)
    ledPin2Status = 1;
  if (sensorValue > 560)
    ledPin3Status = 1;
  if (sensorValue > 562)
    ledPin4Status = 1;
  if (sensorValue > 564)
    ledPin5Status = 1;
  if (ledPin1Status == 1 || ledPin2Status == 1 || ledPin3Status == 1 ||
ledPin4Status == 1 || ledPin5Status == 1)

  {

    if (sensorValue > 555 || sensorValue < 537)
      digitalWrite(LED_1, HIGH);
    if (sensorValue > 558 || sensorValue < 534)
      digitalWrite(LED_2, HIGH);
    if (sensorValue > 560 || sensorValue < 534)
      digitalWrite(LED_3, HIGH);
    if (sensorValue > 562 || sensorValue < 531)
      digitalWrite(LED_3, HIGH);
    if (sensorValue > 564 || sensorValue < 528)
      digitalWrite(LED_4, HIGH);
    if (sensorValue > 568 || sensorValue < 525)
      digitalWrite(LED_5, HIGH);

    delay(200);
```

```
    ledPin5Status = 0;
    ledPin4Status = 0;
    ledPin3Status = 0;
    ledPin2Status = 0;
    ledPin1Status = 0;
  }

  digitalWrite(LED_1, LOW);
  digitalWrite(LED_2, LOW);
  digitalWrite(LED_3, LOW);
  digitalWrite(LED_4, LOW);
  digitalWrite(LED_5, LOW);
}
```

16. Exercise 15: Arduino with Random LED Illumination

This code plays the defined melody through the passive buzzer and randomly lights up one LED out of the 5 connected LEDs for each note in the melody. Adjust the melody[] array and noteDurations[] array to change the melody and the durations of each note.

Required components for the Exercise 15:

- LEDs x 5 (Choose colours as you prefer)
- Male to Male 20cm Jumper Cables x 10
- 220 Ohm Resistors x 5
- Passive Buzzer
- Arduino UNO Board
- Arduino Breadboard
- USB 2.0 Cable Type A/B

Melody and Note Durations:

The melody[] array contains musical notes (like NOTE_C4, NOTE_D4, etc.) that represent the pitches you want to play. The noteDurations[] array contains the durations of each note in the melody (e.g., 4 for a quarter note, 2 for a half note, etc.). The for loop in the loop() function iterates through each note in the melody.

Playing the Melody:

Within each iteration of the loop, the tone() function is used to play a note from the melody[] array through the passive buzzer. The duration for which the note is played is determined by the corresponding value in the noteDurations[] array.

Illuminating LEDs:

The function illuminateRandomLED() is called after starting each note. illuminateRandomLED() selects a random LED from the set of 5 LEDs connected and turns it on. It uses the random() function to generate a random number between 0 and 4 (inclusive) to select an LED pin. The selected LED remains illuminated for the duration of the note being played.

Turning Off LEDs:

After each note is played and the LED has been randomly illuminated, the turnOffLEDs() function is called. turnOffLEDs() turns off all LEDs by setting their corresponding pins to LOW. By adjusting the melody[] array and noteDurations[] array, you can change the tune that plays through the buzzer. Each note's duration determines how long the LED stays lit during that note. The code randomly selects an LED to light up, creating a random lighting pattern synchronized with the melody being played.

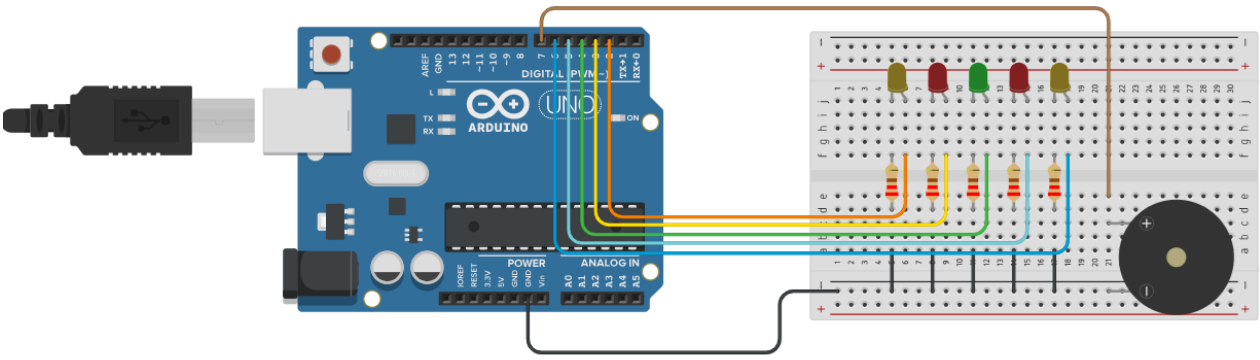


Figure 37: Wiring Diagram for Exercise 15

Tip: Place resistors and LEDs to use minimum number of jumper wires, above diagram is for demonstration.

Sample Code

```
// Melody notes definitions
#define NOTE_C4 262
#define NOTE_CS4 277
#define NOTE_D4 294
#define NOTE_DS4 311
#define NOTE_E4 330
#define NOTE_F4 349
#define NOTE_FS4 370
#define NOTE_G4 392
#define NOTE_GS4 415
#define NOTE_A4 440
#define NOTE_AS4 466
#define NOTE_B4 494
#define NOTE_C5 523
// Melody notes and durations
```

```
int melody[] = {NOTE_C4, NOTE_D4, NOTE_E4, NOTE_F4, NOTE_G4, NOTE_A4, NOTE_B4,
NOTE_C5};
int noteDurations[] = {4, 4, 4, 4, 4, 4, 4, 4}; // Change durations as needed

// Pins for LEDs
const int numLEDs = 5;
const int ledPins[] = {2, 3, 4, 5, 6}; // Pins for the LEDs

// Pin for the passive buzzer
const int buzzerPin = 7; // Change this pin if your buzzer is connected to a
different pin

void setup() {
  // Initialize LED pins as outputs
  for (int i = 0; i < numLEDs; i++) {
    pinMode(ledPins[i], OUTPUT);
  }
  // Initialize buzzer pin as output
  pinMode(buzzerPin, OUTPUT);
}

void loop() {
  // Play melody and randomly illuminate LEDs
  for (int i = 0; i < 8; i++) {
    tone(buzzerPin, melody[i]);
    illuminateRandomLED();
    delay(1000 / noteDurations[i]); // Adjust timing for the notes' durations
    noTone(buzzerPin);
    turnOffLEDs();
    delay(50); // Short delay between notes
  }
}

// Function to randomly illuminate an LED
void illuminateRandomLED() {
  int randomLED = random(0, numLEDs); // Generate a random LED index
  digitalWrite(ledPins[randomLED], HIGH); // Turn on the randomly chosen LED
}

// Function to turn off all LEDs
void turnOffLEDs() {
  for (int i = 0; i < numLEDs; i++) {
    digitalWrite(ledPins[i], LOW);
  }
}
```

18. References

Arduino. (2023). Retrieved from https://www.arduino.cc

AUTODESK. (2023). *Tinkercad*. Retrieved from www.tinkercad.com

Circuit Geeks. (2023). Retrieved from https://www.circuitgeeks.com

CodeMagic LTD. (2023). *WOKWI*. Retrieved from WOKWI: https://wokwi.com

GitHub. (2023). Retrieved from https://github.com

END

Doc Version: 1.0.0
Product Version: 1.0.0
Report issues to: info@optimus.lk