National Textile University
Department of Computer Science

Assignment no.

01

Submitted to:

Sir Nasir Mehmood

Submitted by:

Arooj Komal (23-NTU-CS-1140)

Subject:

Machine Learning

Section:

BSAI

Semester:

5th

Assignment 1

Question 1 — Short Questions

Answer concisely, in your own words.

1. **Why is volatile used for variables shared with ISRs?**
   Volatile ensures that the compiler must use the most recent values of the variable that are changed by the ISR, preventing unnecessary or incorrect optimizations.
2. **Compare hardware-timer ISR debouncing vs. delay()-based debouncing.**
   - **Hardware-timer ISR debouncing** is precise, efficient and non-blocking. It is faster because it uses a hardware timer interrupt to measure a stable timer after a button edge. It is a real-time system for example ESP32. It is slightly more complex because it requires ISR + timer setup.
   - **Delay based debouncing** is less accurate, simple but it blocks the program. It is slow because it ignores next button press until the delay () ends. It is used in simple, low-priority applications like Arduino beginners. It is very simple because it uses only delay() after the input read.
3. **What does IRAM_ATTR do, and why is it needed?**
   IRAM-ATTR places an ISR in instruction RAM so it can run instantly and safely, even when the flash memory is unavailable or busy.
4. **Define LEDC channels, timers, and duty cycle.**
   - **LEDC channels** controls individual PWM outputs (upto 16 on the ESP32).
   - **LEDC timers** set the frequency and resolutions for one or more channels in ESP32.
   - **Duty cycle** is the percentage of time the PWM signal remains high in a single cycle.
5. **Why should you avoid Serial prints or long code paths inside ISRs?**
   Because serial prints and long code makes the ISR slower, causing missed interrupts, timing issues, or system instability, ISRs should be short and fast.
6. **What are the advantages of timer-based task scheduling?**
   Timer based scheduling makes the task execution precise, periodic, and non-blocking task execution, improving timer accuracy, CPU efficiency, and real-time performance.
7. **Describe I^{2}C signals SDA and SCL.**
   I^{2}C uses two types of wires to communicate between a slave (sensors) and a master devices (ESP32).
   - **SDA (Serial Data):** Transfers the actual data bits between the devices.
   - **SCL (Serial Clock):** Generated by the master to synchronize when each bit on SDA is read or written.

   Both of them are open-drain and pull-up resistors so multiple devices can use the same bus safely and efficiently.
8. **What is the difference between polling and interrupt-driven input?**
   Polling: The CPU repeatedly checks the input state in a loop. In this way it wastes time and may miss quick events.

Interrupt-driven input: The CPU reacts only when an interrupt occurs, saving processing time and responding instantly.

9. **What is contact bounce, and why must it be handled?**
   Contact bounce is a quick ON/OFF flickering that happens when a mechanical switch is pressed or released. It should be handled efficiently to avoid false multiple triggers and ensure reliable input detection.
10. **How does the LEDC peripheral improve PWM precision?**
    The LEDC peripheral uses the dedicated hardware timers with high-bit resolution, allowing stable frequency and fine duty-cycle control, which improves PWM precision.
11. **How many hardware timers are available on the ESP32?**
    The ESP32 has mainly 4 general-purpose 64-bit hardware timers, divided into 2 timer groups with 2 timers each. Each timer has its own set of control and status registers for configuration and operation.
12. **What is a timer prescaler, and why is it used?**
    A timer prescalar divides the main clock frequency before it reaches the timer. It is used to slow down the timer control rate, allowing longer timing intervals and better control over timing precision.
13. **Define duty cycle and frequency in PWM.**
    Duty-cycle: The percentage of a PWM period that the signal remains high is called duty cycle.
    Duty cycle = (ON time / total period) * 100%
    Frequency: The number of PWM cycles that occur per second that is measured in Hz.
    Freq = 1 / period
14. **How do you compute duty for a given brightness level?**
    Duty = (Desired Brightness / Maximum Brightness) * Maximum duty value
    Example: For 50% brightness at 8-bit resolution → Duty = 0.5 × 255 = 128.
15. **Contrast non-blocking vs. blocking timing.**
    Blocking Timing: It uses functions like delay() to stop all other code from running.
    Non-blocking timing: it uses timers or millis() to track the time, allowing other tasks to run simultaneously.
16. **What resolution (bits) does LEDC support?**
    The LEDC peripheral supports up to 20-bit resolution, depending on the selected PWM frequency.
17. **Compare general-purpose hardware timers and LEDC (PWM) timers.**
    General-Purpose timers: It is used for timing, delays, and periodic interrupts; fully programmable for various tasks.
    LEDC timers: It is specialized for generating PWM signals with set frequency and resolution for LED or motor control.
18. **What is the difference between Adafruit_SSD1306 and Adafruit_GFX?**
    Adafruit_SSD1306: It handles the OLED display hardware like communication, initialization and pixel control, etc.

Adafruit_GFX: It provides graphics function like drawing shapes, text, and fonts, used by many display drivers including SSD1306.

19. **How can you optimize text rendering performance on an OLED?**
    By updating only the changed screen areas, use smaller fonts, minimize screen refresh calls and avoid clearing the whole display to improve rendering speed.
20. **Give short specifications of your selected ESP32 board (NodeMCU-32S).**

    It has Dual-core **ESP32** (240 MHz), **520 KB SRAM**, **Wi-Fi + Bluetooth**, **30 GPIO pins**, **12-bit ADC**, **4 hardware timers**, **16 PWM (LEDC) channels**, and built-in **USB-to-UART** interface.

Question 2 — Logical Questions

1. **A 10 kHz signal has an ON time of 10 ms. What is the duty cycle? Justify with the formula.**
   Duty = (t_{on} / T) × 100%
   T = 1 / 10 kHz = 0.0001 s = 100 µs
   Duty = (10 ms / 100 µs) × 100 = 10,000% (impossible).
   If ON time is 10 µs, then Duty = (10 µs / 100 µs) × 100 = **10%** (possible)

   In this question statement is some problematic so I have made my own assumption.
2. **How many hardware interrupts and timers can be used concurrently? Justify.**

   The ESP32 includes two timer groups, each containing two 64-bit general-purpose hardware timers, giving 4 timers total, all of which can run independently and concurrently**.**

   It also has 32 interrupt lines per core, and through the interrupt matrix, any peripheral interrupt source can be mapped to these lines.
   This hardware design allows multiple timers and interrupts to operate at the same time without interfering with each other.

3. **How many PWM-driven devices can run at distinct frequencies at the same time on ESP32? Explain constraints.**

   The ESP32's LEDC (PWM) controller has 8 timers and 16 channels.
   Each timer sets one PWM frequency and resolution, while channels share that timer's settings.
   So, up to 8 devices can run at different frequencies simultaneously.

   Constraints:

   - Channels using the same timer must share the same frequency.
   - Higher frequency reduces available duty resolution (trade-off).
   - Limited by available GPIO pins and LEDC resources.

4. **Compare a 30% duty cycle at 8-bit resolution and 1 kHz to a 30% duty cycle at 10-bit resolution (all else equal).**

Given:

Duty cycle = 30%

Frequency = 1kHz → Period = 1/1000 = 1ms = 1000 μs

At 8-bit resolution:

Total steps = 2^8 = 256

Duty steps for 30%:

$$\mathbf{0.30 \times 256 = 76.8 \approx 77}$$

Each step size = 1/256 *100% = 0.39%

- ON time = 30%×1000μs = 300μs

At 10-bit resolution:

Total steps = 2^{10} = 1024

Duty steps for 30%:

$$\mathbf{0.30 \times 1024 = 307.2 \approx 307}$$

Each step size = 1/1024 *100% = 0.098%

- ON time = 30%×1000μs = 300μs

Result:
Both have 300 μs ON time (30%)**,** but 10-bit PWM gives finer control (0.098% per step) vs 8-bit's 0.39% per step, so it produces smoother and more precise output.

5. **How many characters can be displayed on a 128×64 OLED at once with the minimum font size vs. the maximum font size? State assumptions.**

Assumptions

- Display resolution = **128 × 64 pixels** (width × height).

- Characters are placed in a fixed grid; each character cell = **(char_width × char_height)** pixels including any inter-character spacing.

1) Minimum (tiny) font — 6 × 8 px

- Columns = ⌊128 / 6⌋ = **21**
- Rows = ⌊64 / 8⌋ = **8**
- Total chars = **21 × 8 = 168**

2) Small/standard font — 8 × 8 px

- Columns = ⌊128 / 8⌋ = **16**
- Rows = ⌊64 / 8⌋ = **8**
- Total chars = **16 × 8 = 128**

3) Medium font — 8 × 16 px (taller)

- Columns = ⌊128 / 8⌋ = **16**
- Rows = ⌊64 / 16⌋ = **4**
- Total chars = **16 × 4 = 64**

4) Large font — 16 × 32 px

- Columns = ⌊128 / 16⌋ = **8**
- Rows = ⌊64 / 32⌋ = **2**
- Total chars = **8 × 2 = 16**

5) Very large / maximum (example) — 32 × 48 px

- Columns = ⌊128 / 32⌋ = **4**
- Rows = ⌊64 / 48⌋ = **1**
- Total chars = **4 × 1 = 4**
-

With a tiny 6×8 font you can fit ~168 chars; with very large display fonts you might be down to 4–16 chars; exact numbers depend on the font glyph size and inter-character spacing that someone choose.

Question 3 :

<u>Task 1:</u>
Coding: Use one button to cycle through LED modes (display the current state on the OLED):
1. Both OFF
2. Alternate blink

3. Both ON
4. PWM fade Use the second button to reset to OFF.

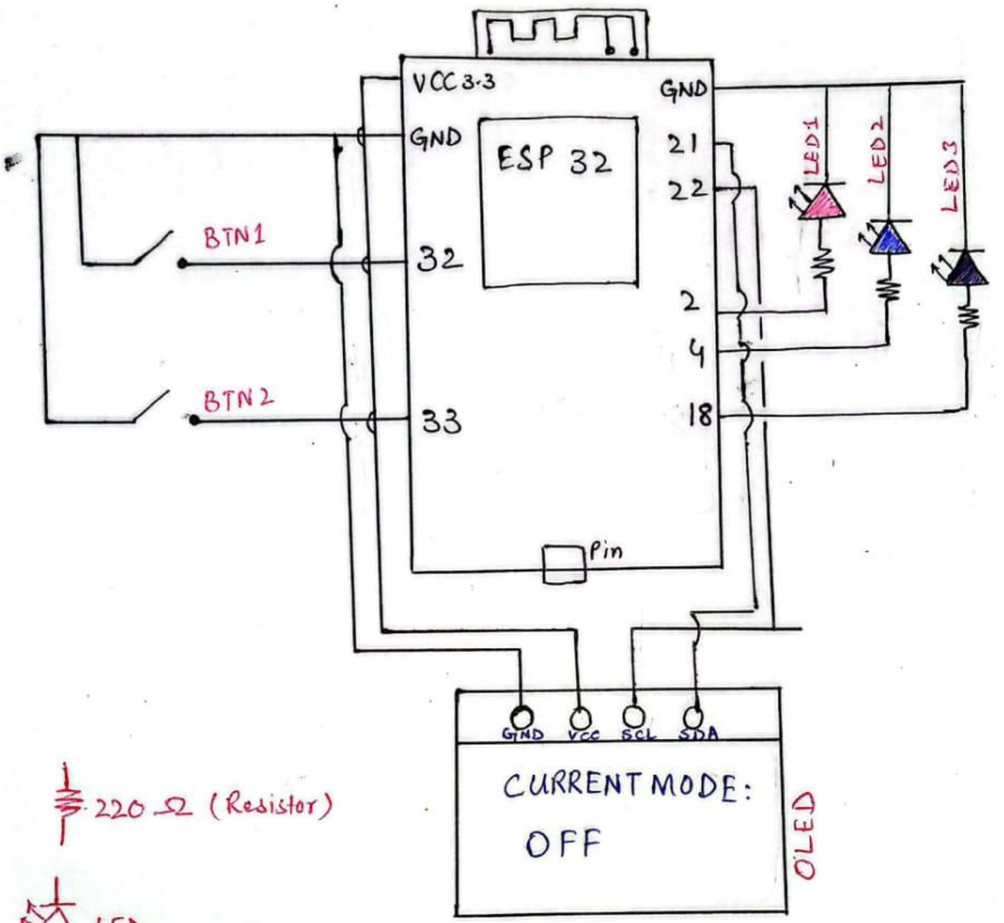
Code:

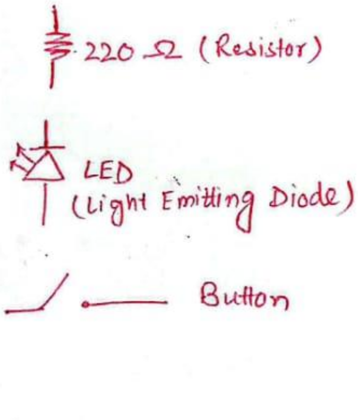
```
#include <Arduino.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

// ---- Pin Setup ----
#define BUTTON_MODE 32    // Button to change LED mode
#define BUTTON_RESET 33   // Button to reset to OFF
#define LED1 2
#define LED2 4
#define LED3 18

// ---- OLED Setup ----
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

// ---- Global Variables ----
int mode = 0;
int lastButtonState = HIGH;

// ---- Function to Update OLED Display ----
void updateDisplay(const char* message) {
  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0, 20);
  display.print("Current Mode:");
  display.setCursor(0, 40);
  display.print(message);
  display.display();
}

// ---- Setup ----
void setup() {
  pinMode(BUTTON_MODE, INPUT_PULLUP);
  pinMode(BUTTON_RESET, INPUT_PULLUP);
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
  Serial.begin(9600);

  // Initialize OLED
  if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println("OLED not found!");
    while (true);
  }

  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0, 10);
  display.println("ESP32 LED Controller");
  display.setCursor(0, 30);
  display.println("Press button to start");
  display.display();

  Serial.println("ESP32 LED Mode Controller Ready!");
}

// ---- Main Loop ----
void loop() {
  int buttonState = digitalRead(BUTTON_MODE);
  int resetState = digitalRead(BUTTON_RESET);

  // --- Change Mode Button ---
  if (buttonState == LOW && lastButtonState == HIGH) {
    mode++;
    if (mode > 3) mode = 0;
    Serial.print("Mode changed to: ");
    Serial.println(mode);
    delay(200);
  }

  // --- Reset Button ---
  if (resetState == LOW) {
    mode = 0;
    Serial.println("Reset to OFF");
    delay(200);
  }

  lastButtonState = buttonState;

  // --- LED Behavior Based on Mode ---
  switch (mode) {
    case 0: // OFF
      digitalWrite(LED1, LOW);
      digitalWrite(LED2, LOW);
      digitalWrite(LED3, LOW);
      updateDisplay("OFF");
      break;

    case 1: // Alternate Blink
      updateDisplay("Alternate Blink");
      digitalWrite(LED1, HIGH);
      digitalWrite(LED2, LOW);
      digitalWrite(LED3, LOW);
      delay(300);
      digitalWrite(LED1, LOW);
      digitalWrite(LED2, HIGH);
      digitalWrite(LED3, LOW);
      delay(300);
      break;

    case 2: // All ON
      updateDisplay("All ON");
      digitalWrite(LED1, HIGH);
      digitalWrite(LED2, HIGH);
      digitalWrite(LED3, HIGH);
      break;

    case 3: // Fade LED1
      updateDisplay("PWM Fade");
      for (int d = 0; d <= 255; d++) {
        analogWrite(LED1, d);
        delay(5);
        analogWrite(LED2, d);
        delay(5);
      }
      for (int d = 255; d >= 0; d--) {
        analogWrite(LED1, d);
        delay(5);
        analogWrite(LED2, d);
        delay(5);
      }
      break;
  }
}
```

Diagram:

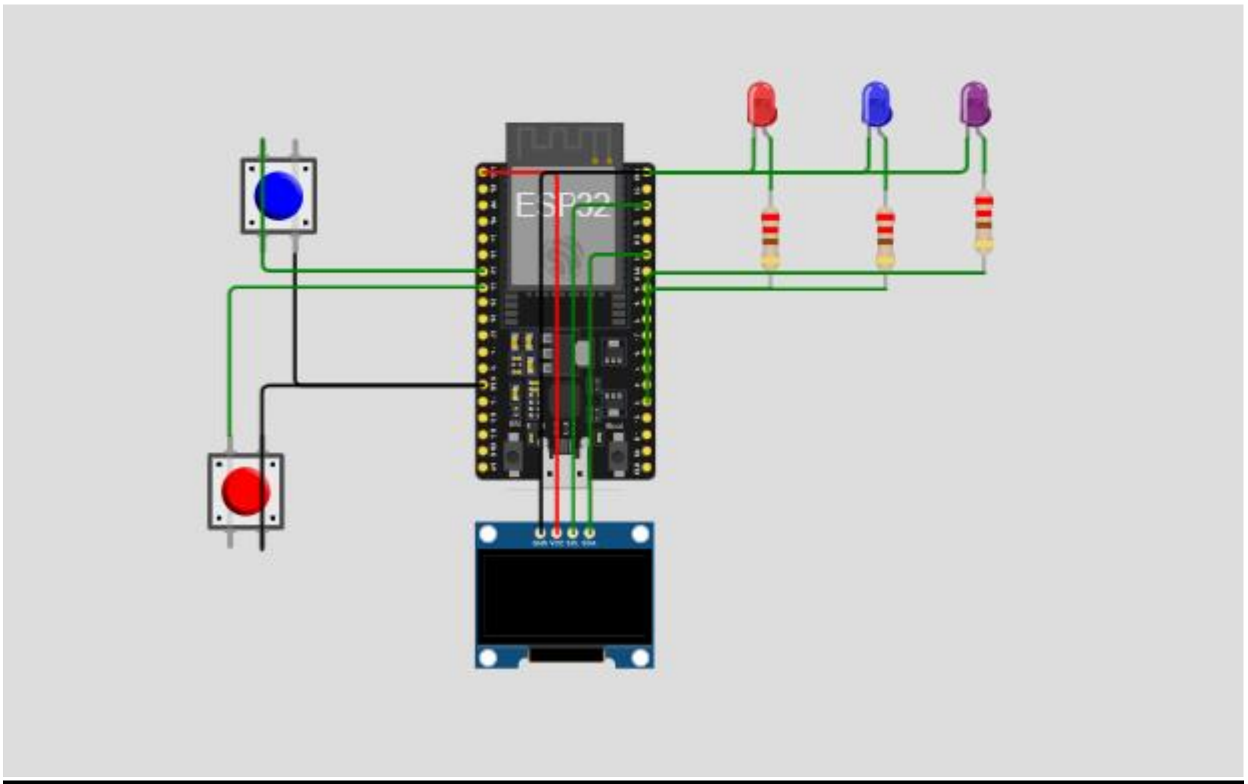
ASSIGNMENT # 1

TASK 1



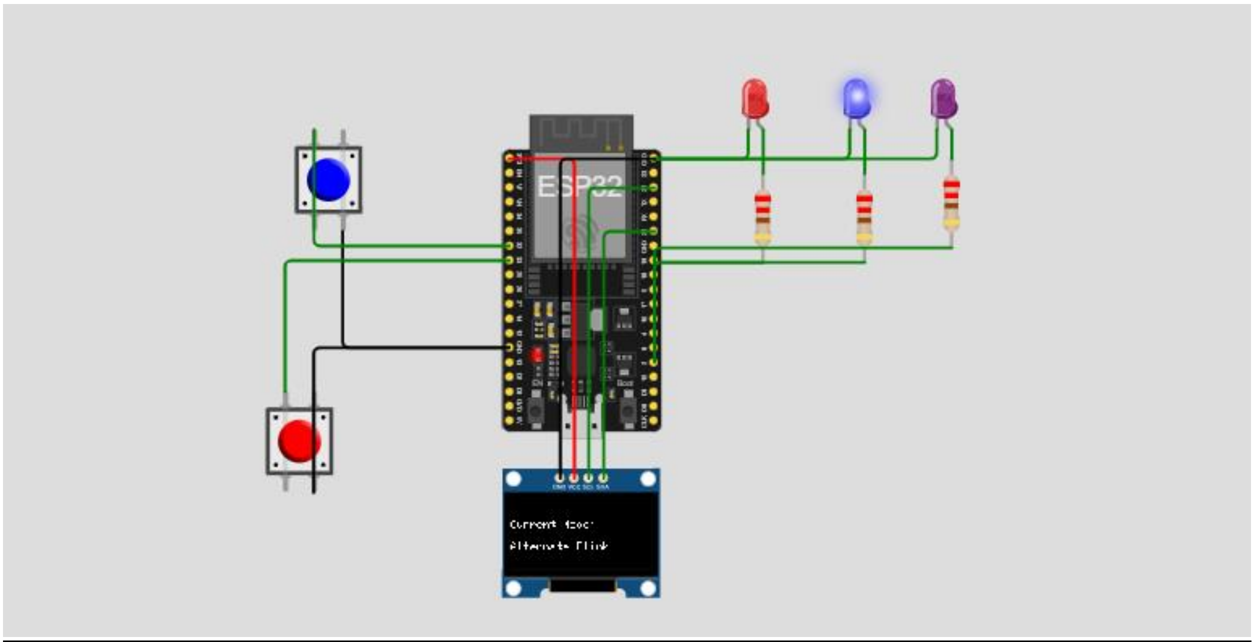


All on:



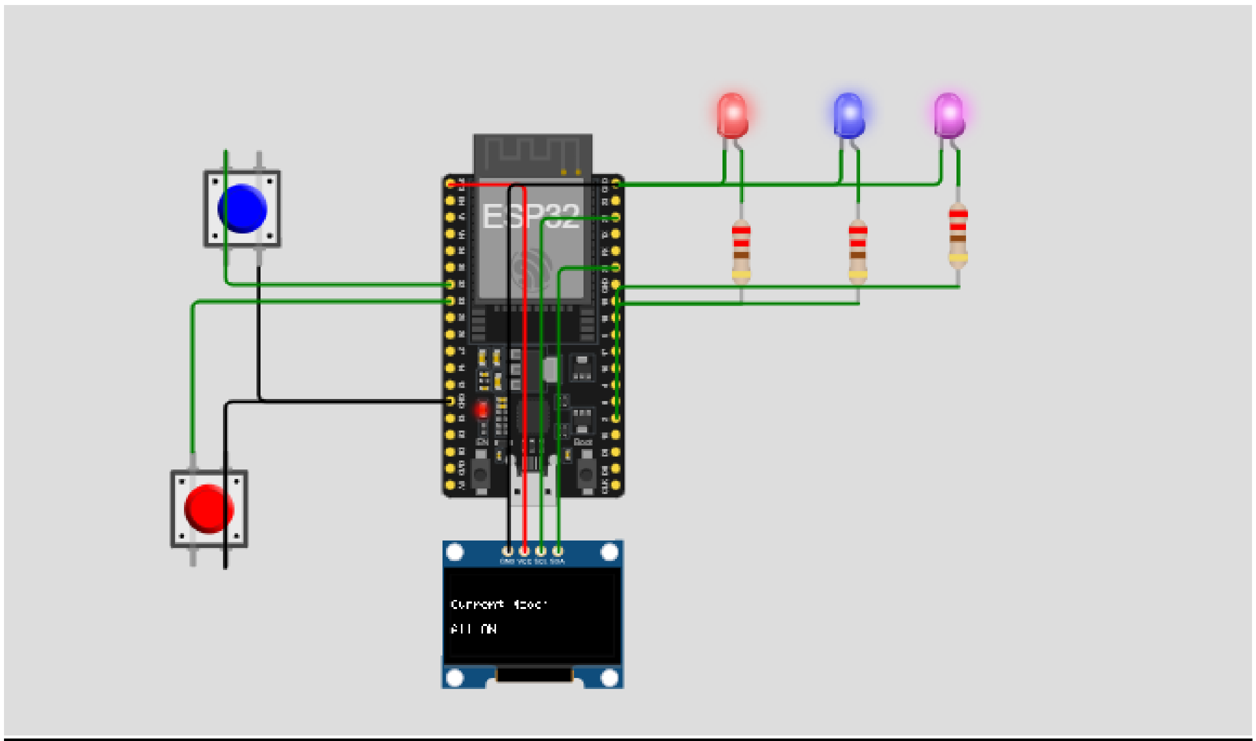
Output:

One at a time:



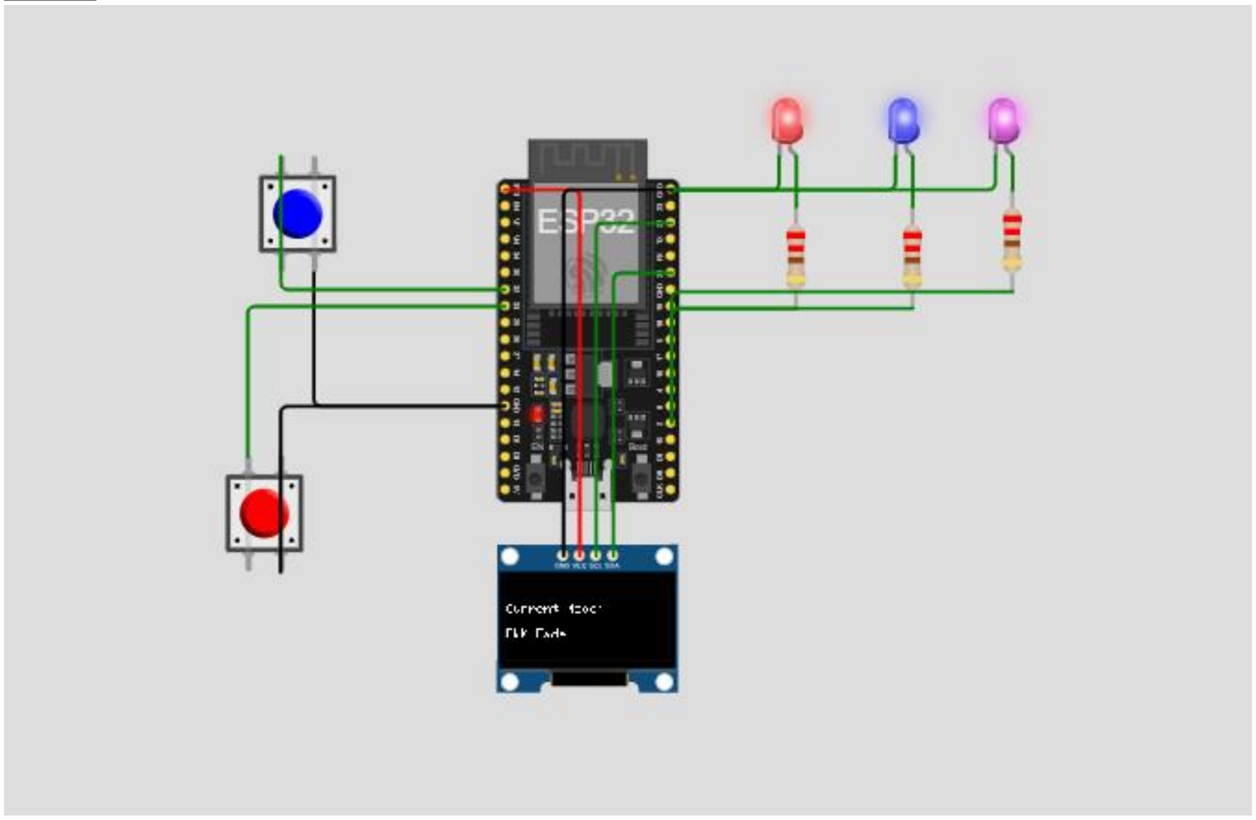
```
Current Mode:
Alternate Blink
```

All on:



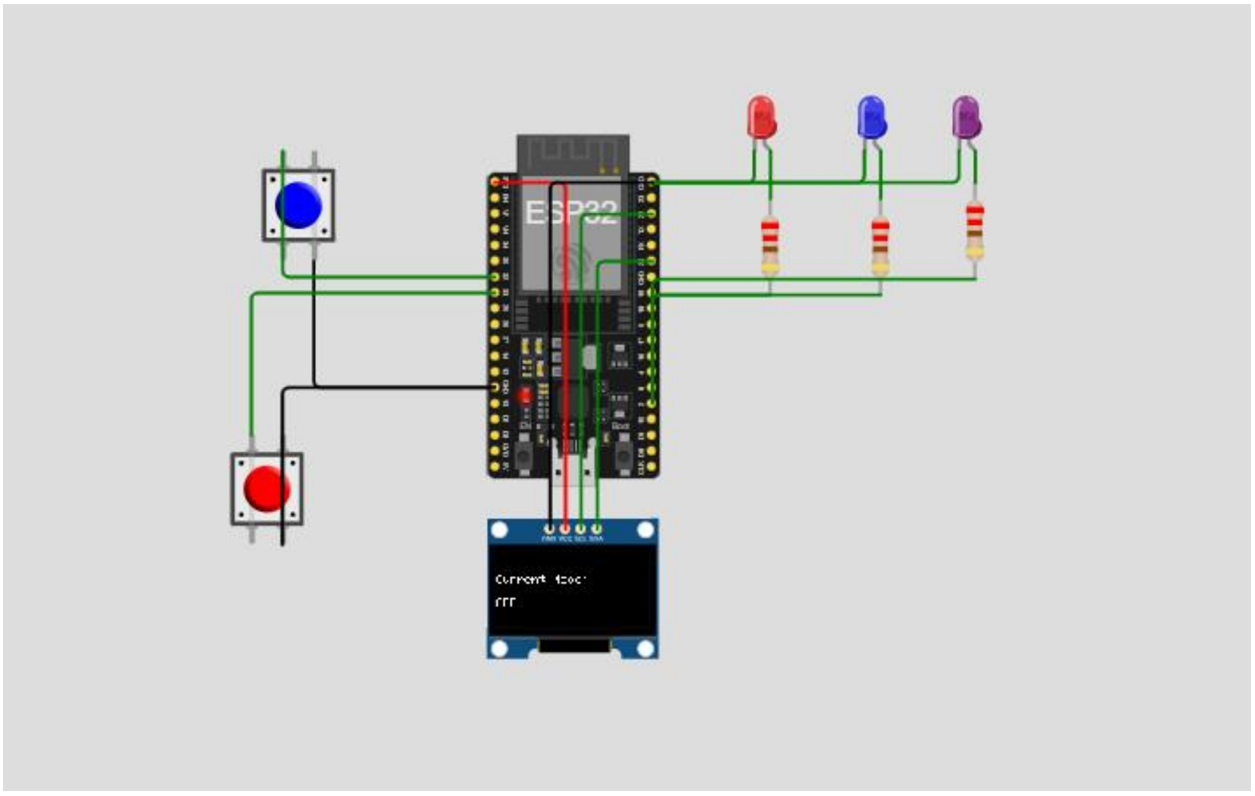
```
Current Mode:
All ON
```

PWM:



```
Current Mode:
PWM Fade
```

Reset:



Task2:

Coding: Use a single button with press-type detection (display the event on the OLED):

• Short press → toggle LED

• Long press (> 1.5 s) → play a buzzer tone

Code:

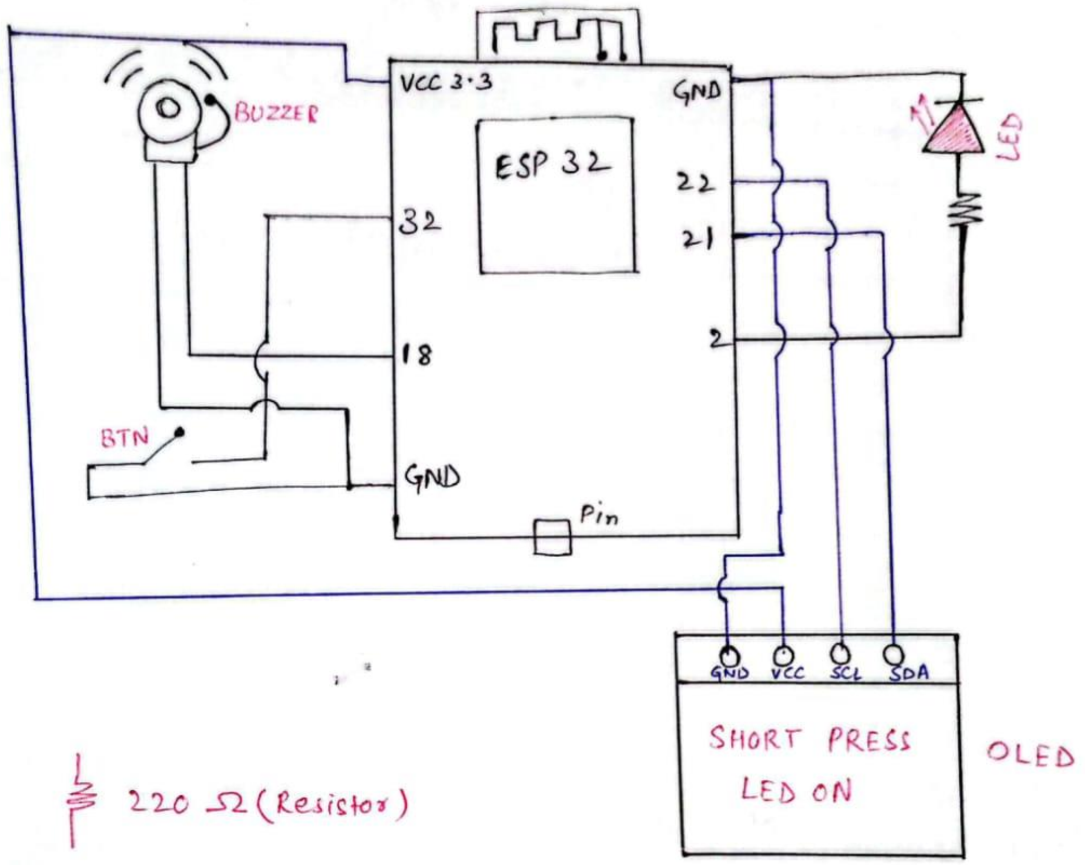
```
/* Name : Arooj Komal
   Reg no. : 23-NTU-CS-1140
   Section: BSAI 5th
*/



#include <Arduino.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

// --- Pin setup ---
#define BUTTON_PIN 32
```

```
#define LED_PIN 2
#define BUZZER_PIN 18

// --- OLED setup ---
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

// --- Variables ---
bool ledState = false;             // Keeps track of LED ON/OFF
unsigned long pressStartTime = 0;  // When button was pressed
bool buttonPressed = false;        // Flag for press detection
const unsigned long longPressTime = 1500; // 1.5 seconds

// --- Helper function to update OLED text ---
void showMessage(const char* msg) {
  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0, 25);
  display.print(msg);
  display.display();
}

// --- Setup ---
void setup() {
  pinMode(BUTTON_PIN, INPUT_PULLUP); // Button reads HIGH when not pressed
  pinMode(LED_PIN, OUTPUT);
  pinMode(BUZZER_PIN, OUTPUT);

  Serial.begin(9600);

  // Initialize OLED
  if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println("OLED not found!");
    while (true);
  }

  showMessage("Ready! Press Button");
  digitalWrite(LED_PIN, ledState);
}

// --- Loop ---
void loop() {
  int buttonState = digitalRead(BUTTON_PIN);
  if (buttonState == LOW && !buttonPressed) {
    // Button just pressed
    pressStartTime = millis();
    buttonPressed = true;
  }

  if (buttonState == HIGH && buttonPressed) {
    // Button just released
    unsigned long pressDuration = millis() - pressStartTime;
    buttonPressed = false;

    if (pressDuration < longPressTime) {
      // Short press → toggle LED
      ledState = !ledState;
      digitalWrite(LED_PIN, ledState);
      showMessage(ledState ? "Short Press:\nLED ON" : "Short Press:\nLED OFF");
      Serial.println("Short press detected");
    } else {
      // Long press → play buzzer
      showMessage("Long Press:\nBuzzer Tone!");
      Serial.println("Long press detected");
      tone(BUZZER_PIN, 1000, 500); // 1kHz tone for 500 ms
      delay(500);
      noTone(BUZZER_PIN);
    }
  }
}
```

Diagram:

ASSIGNMENT # 1

TASK 2

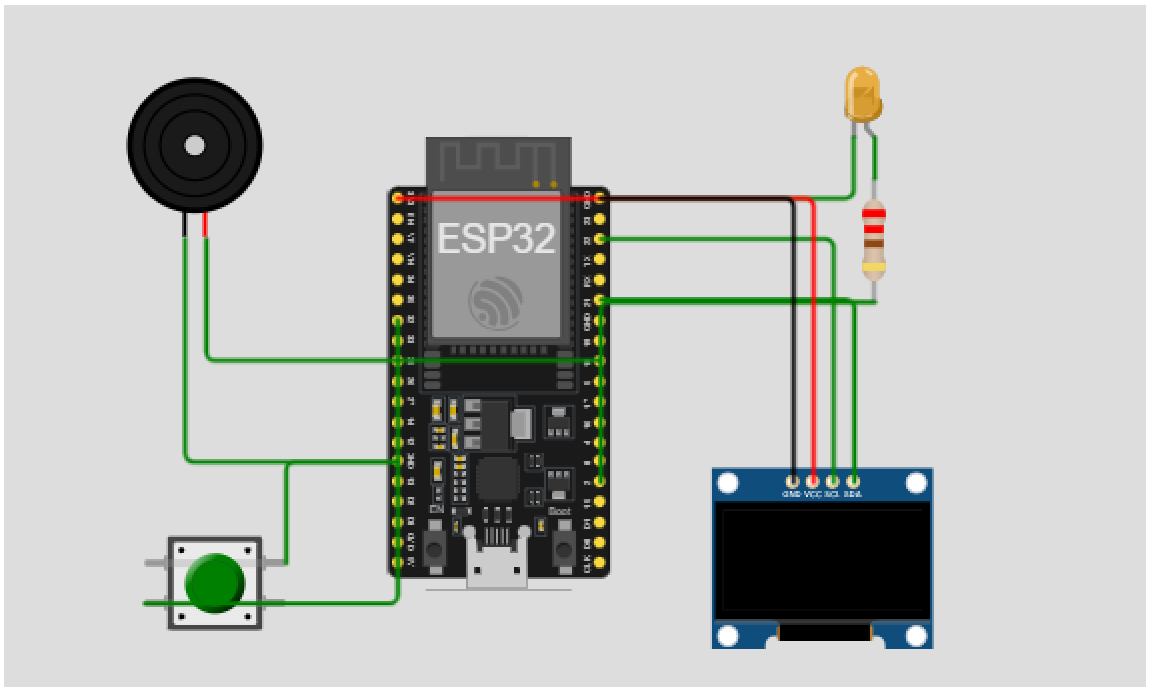


220 Ω (Resistor)

LED
(LIGHT EMITTING DIODE)

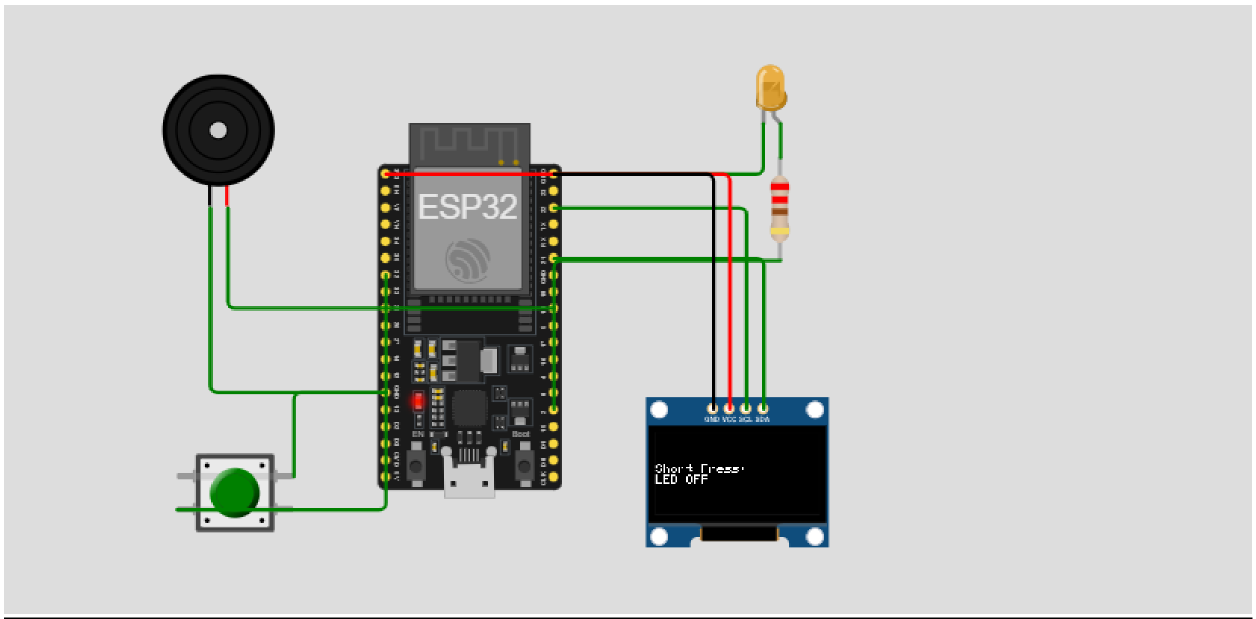
BTN

Buzzer



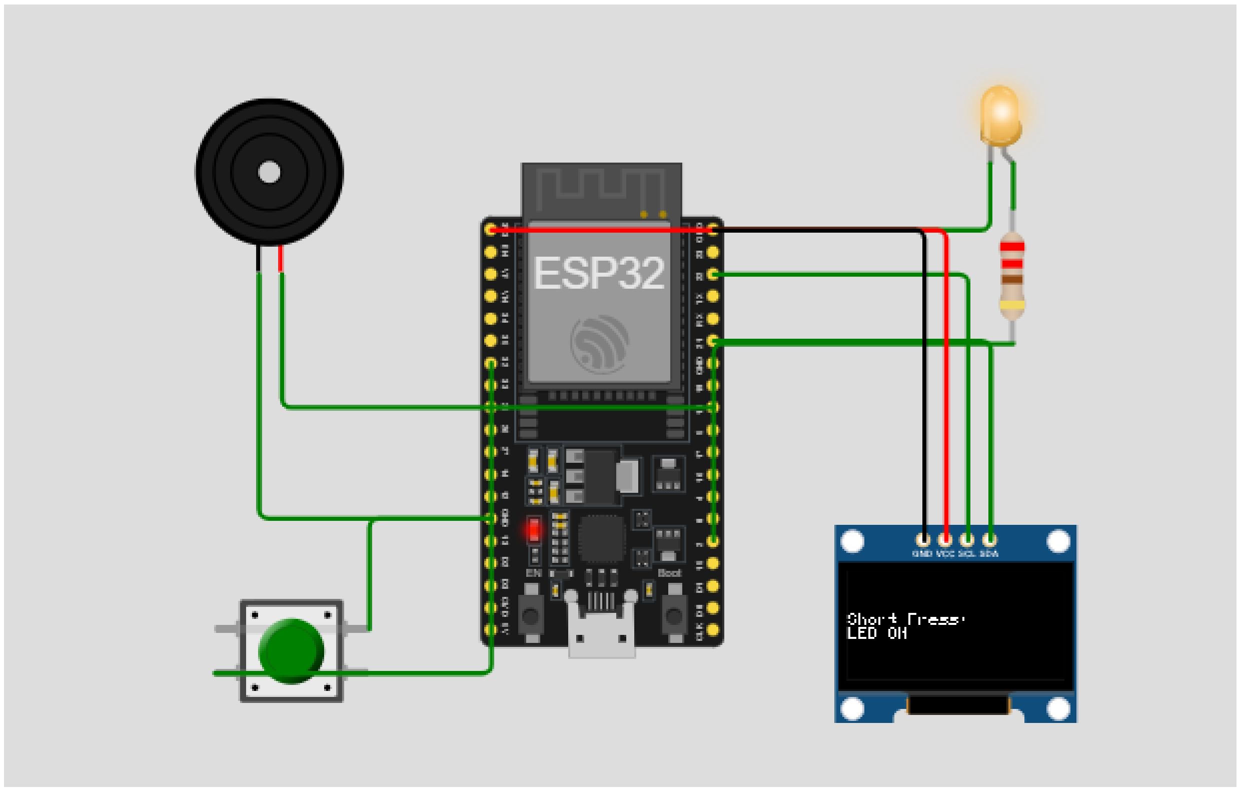
Output:

Led off:

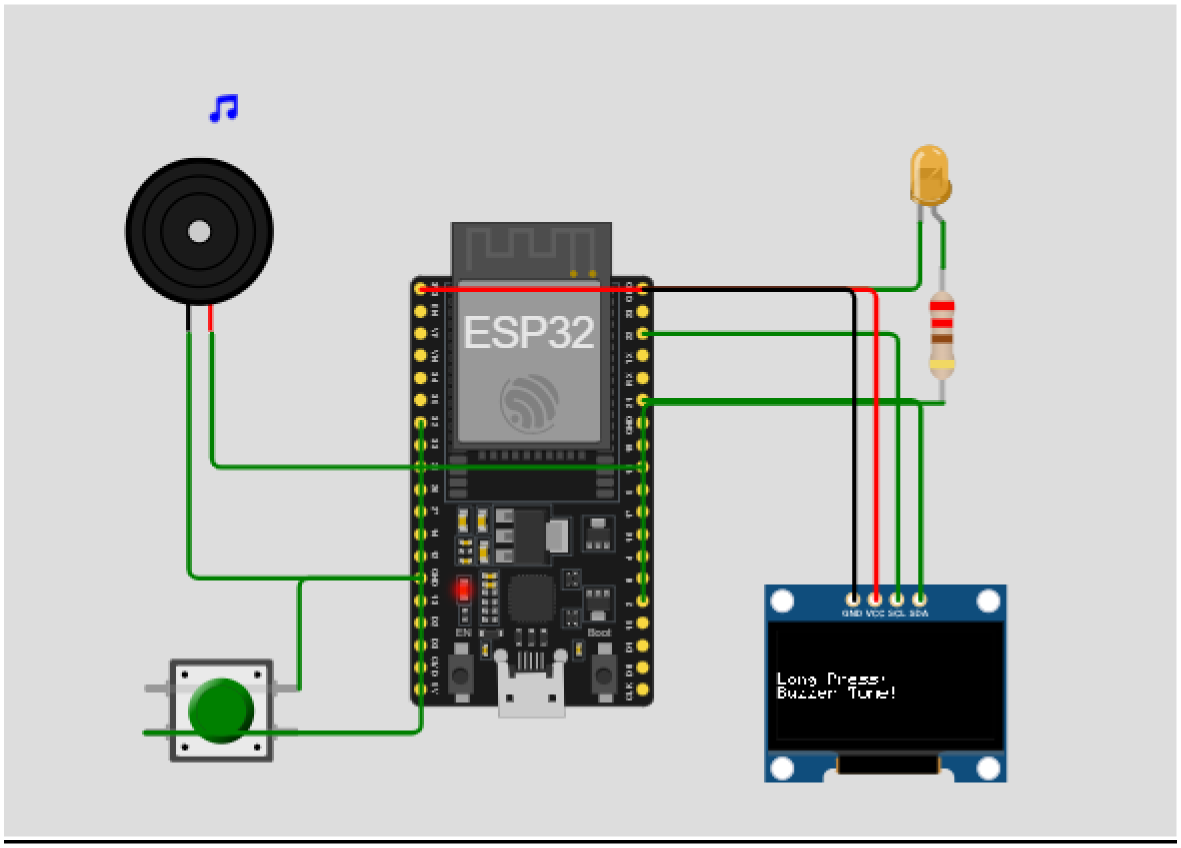


```
Short Press:
LED OFF
```

Led on:


ESP32
GND VCC SCL SDA
Short Press:
LED ON

Buzzer:


ESP32
GND VCC SCL SDA
Long Press:
Buzzer Tune!