Name:	Essha Imran
Roll no:	23-NTU-CS-1022
Section:	A
Submitted to:	Sir Nasir
Submission Date:	19-10-2025

Answer1:

Volatile is used for variables that are used by ISRs because these variables are changing continuously according to our function and we take new value every time, and not use only one value.

Answer2:

- **Delay()-based debouncing**: It is used in small projects where we don't need to run complex code or task but during delay(), the CPU stops for working. It is bad in responsiveness and don't use with multiple tasks.

- **Hardware-timer ISR debouncing**: It is used in large or complex projects where multiple tasks are running. The CPU doesn't stop working and it runs the main code. It is good in responsiveness and provides good efficiency.

Answer3:

The ESP32 uses flash memory to store data but we need faster execution for ISR functions. For this, IRAM-ATTR puts the ISR functions into the instruction RAM. So, our code directly and fastly accesses the ISR function even when flash memory is busy.

Answer4:

- **Timer:** It sets the frequency (how fast and smooth the process runs) and resolution (adjust brightness and power).
- **Channel:** The channel is actually the pin which provides PWM signals. We can attach maximum 8 timers with one channel and all timers have same frequency.

- **Duty Cycle:** The percentage or amount of time it shows brightness.

Answer5:

We don't write long coding paths and avoid serial prints when dealing with ISR because both take too much time to execute, however, ISR must finish fast. If we write long coding paths inside ISR, it takes longer time to execute which also block further interrupts to occur.

Answer 6:

Advantages of timer-based task:

- We don't need to use delay()-based debouncing, so our CPU don't sit idle.
- It handles the complex code and run multiple tasks at once.
- It provides accurate timing.
- It is energy efficient.

Answer7:

- **SDA (Serial Data):** It transfer data between one or more devices.
- **SCA (Serial Clock):** It gives the clock cycles to manage communication.

Both are open-drain as devices use pull-up registers to keep them ON.

Answer8:

- **Polling:** In this process, the input is checked continuously. So, it wastes CPU time and not efficient.

- **Interrupt-driven Input**: In this process, the input is not checked continuously and hardware notifies when an interrupt occur. So, it does not waste CPU time and is efficient.

Answer9:

When you press the button physically, it sends multiple ONs and OFFs in a few milliseconds which toggle our LED multiple times at once and disturbs the process. To handle this, we use delay()-based debouncing for small tasks and hardware-timer interrupt debouncing for complex or larger tasks.

Answer10:

LEDC uses hardware timers which further control resolution bits and channels.

We improve PWM precision by:

- Adjust brightness or motor speed.
- Run without flickering.
- Provides Stable frequency.

Answer11:

ESP32 has 8 total hardware timers from timer0 to timer7. Each has its own pre-scalar and interrupt (run independently).

Answer12:

Time prescalar is used to slow down timer counting by dividing the base frequency. It lets our system process smoothly and allows us to reach long time intervals.

Answer13:

Frequency in PWM: How many cycles (ONs and OFFs) run in one second. More cycles per second make the process smooth and avoid flickering.

Duty Cycle: The percentage of the time our signal is high. 60% duty cycle means that our LED is turn on more than half of the time.

Answer14:

If we have 70% brightness and 10-bit resolution then,

Duty Cycle = (0.70 * 1023) *100

= 614

60%

Answer15:

- **Blocking timing:** It uses delay() function to pause and wait the program. It blocks the CPU to running code, wastes its time and is not efficient.
- **Non-blocking timing:** We use hardware-timer interrupts and millis() for handle debouncing. It does not stop CPU and is more responsive and efficient.

Answer16:

The LED Control supports 1-20 bits of resolution. So, our one cycle can have 2^1 from 2^20 steps. Higher resolution gives smooth brightness control. If we have higher resolution, then our frequency is lower.

Answer17:

1. **General-Purpose timer:**
    - Purpose: Used for counting, delays and interrupts.
    - Output: Do not directly connect to the output.
    - Flexibility: Trigger ISRs, measuring Input.
    - Use Case: Precise timing, managing.

2. **LEDC Timer:**
    - Purpose: Specialized used to generate PWM signals.
    - Output: Drives output through the pin.
    - Flexibility: Generate PWM signals in the waveform.
    - Use Case: Control the brightness of LED, control the speed of motor.

Answer18:

- **Adafruit_SSD1306:** It is a driver library which manages or control SSD1306 OLED. ESP32 communicates with SSD1306 OLED through I^2 C and SPI.
- **Adafruit_GFX**: It is a graphics library which provides functions for drawing like drawCircle(), drawLine().

Answer19:

- Use smaller fonts to display as larger fonts take longer time to draw.
- We need to produce the image first. So, don't do frequent calls of display().display().
- When updating, only refresh modify part, not the whole code.

Answer20:

- **Specifications** of ESP32 NodeMCU-32S:
  - Processor: It is Dual Core Tensilica Xtensa LX6
  - Flash: 4MB
  - GPIO pins: 30 (not all usable for some functions)
  - LEDC channels: 16
  - Hardware timers: 4
  - ADC: 12-bit, multiple channels
  - Wi-Fi: 802.11 b/g/n

Question-2 (Logical Questions)

Answer1:

- Frequency= 10kHz
- Time Period = I/f = 0.0001s = 0.1ms
- Duty Cycle = (Ton/ Ton + Toff) * 100
- Duty Cycle = (10/0.1)*100
- **Duty Cycle** = 10,000%

This is **physically not possible** as duty cycle do not exceed 100%.

Answer2:

ESP32 has 4 general-purpose hardware timers and each is 64-bits. Interrupts may be used from one to many. So, we use multiple hardware timers and interrupts concurrently, each with its own ISR, but their timing should not overlap each other.

Answer3:

There are 16 PWM channels and 8 timers. Each PWM channel can take only one frequency. So, we attach one or more timers which provide the same frequency. When each timer provides different frequency, then we have total 8 distinct frequencies at the same time.

Answer4:

1. **For 8-bit resolution at 1kHz**

- Maximum Duty Cycle = 2^8 – 1 = 255
- Duty = 0.30 * 255 = 77
- Actual Duty = 77/255 = 29.8%

2. **For 10-bit resolution at 1kHz:**

- Maximum Duty Cycle = 2^10 -1= 1023
- Duty = 0.30*1023 = 307
- Actual Duty= 307/1023 = 30.1%

So, for LED Brightness 10-bit provides smooth brightness than 8-bit.But it can produce flickering at low frequency.

Answer5:

1. **For Minimum font size (6*8) :**
    - Columns = 128/6 = 21
    - Rows = 64/8 = 8
      168 characters visible.

2. **For Maximum Font size (12*16):**
    - Columns = 128/12 = 10
    - Rows = 64/16 = 4
      40 characters visible.

Question 3 — Implementation

Task A — 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.

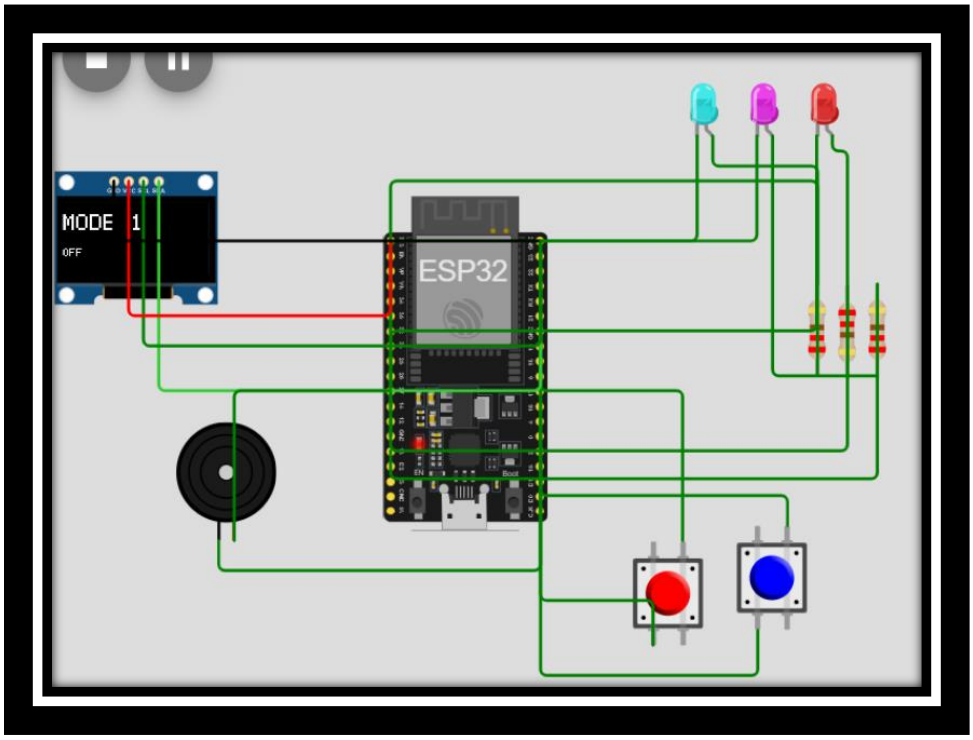
Code:

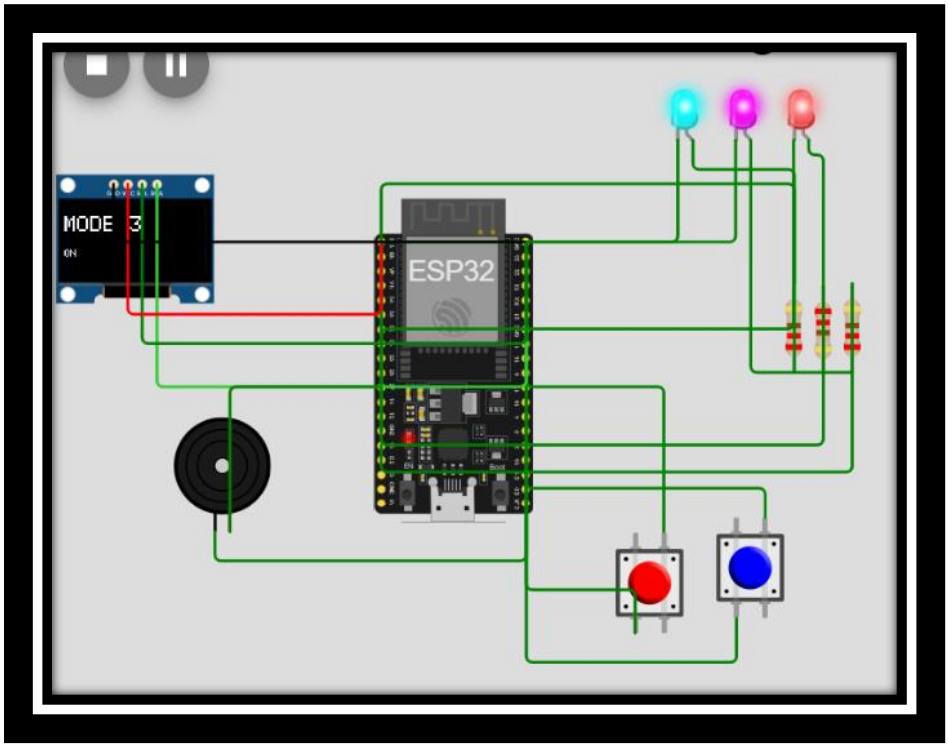
```
#include <Arduino.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

// ==== Pin Definitions ====
#define BTN_MODE 15
#define BTN_RESET 4
#define LED1 13
#define LED2 12
#define LED3 14

// ==== OLED Setup ====
Adafruit_SSD1306 oled(128, 64, &Wire, -1);

// ==== Variables ====
int mode = 0;
unsigned long timer = 0;
bool blink = false;
int fade = 0;
int dir = 5;  // direction of fade change (+/-)

// ==== Function Prototypes ====
void showMode();
void setLEDs(int val);

void setup() {
  pinMode(BTN_MODE, INPUT_PULLUP);
  pinMode(BTN_RESET, INPUT_PULLUP);
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  // Initialize I2C & OLED
  Wire.begin(21, 22);
  oled.begin(SSD1306_SWITCHCAPVCC, 0x3C);
  showMode();
}
```

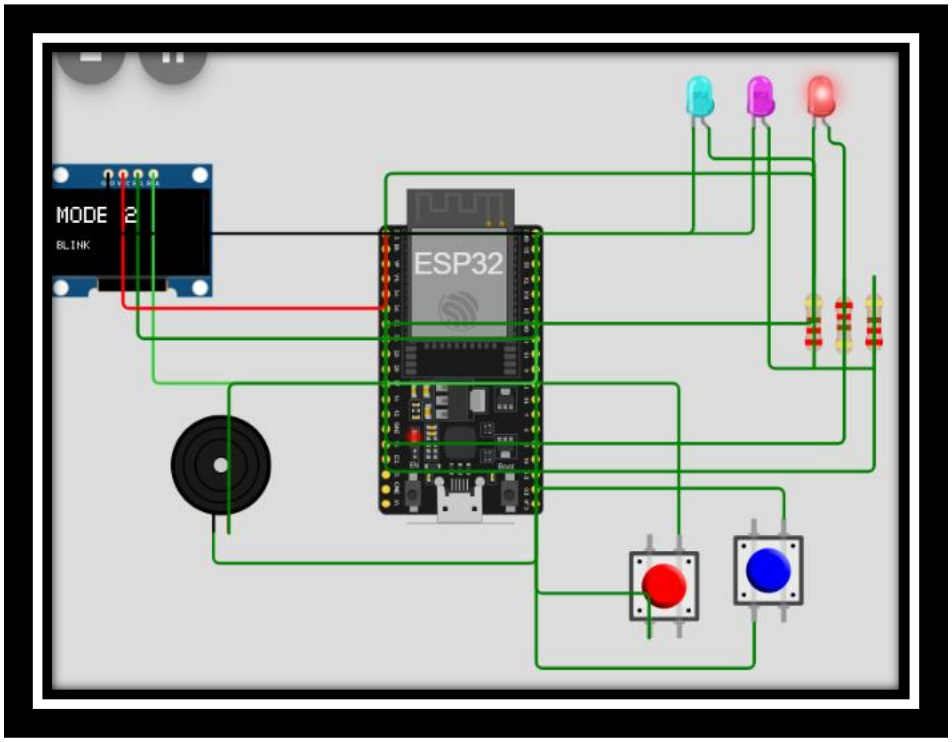
```
void loop() {
  // Mode change button
  if (digitalRead(BTN_MODE) == LOW) {
    delay(200);
    mode = (mode + 1) % 4;
    showMode();
  }

  // Reset button
  if (digitalRead(BTN_RESET) == LOW) {
    delay(200);
    mode = 0;
    setLEDs(0);
    showMode();
  }

  // Mode behaviors
  if (mode == 0) {
    // Mode 1: Both OFF
    setLEDs(0);
  }
  else if (mode == 1) {
    // Mode 2: Alternate Blink
    if (millis() - timer > 500) {
      timer = millis();
      blink = !blink;
          if (blink) {
      analogWrite(LED1, 255);
      analogWrite(LED2, 0);
      analogWrite(LED3, 255);
    } else {
      analogWrite(LED1, 0);
      analogWrite(LED2, 255);
      analogWrite(LED3, 0);
    }
    }
  }
  else if (mode == 2) {
    // Mode 3: Both ON
    setLEDs(255);
  }
  else if (mode == 3) {
    // Mode 4: PWM Fade
    if (millis() - timer > 20) {
```

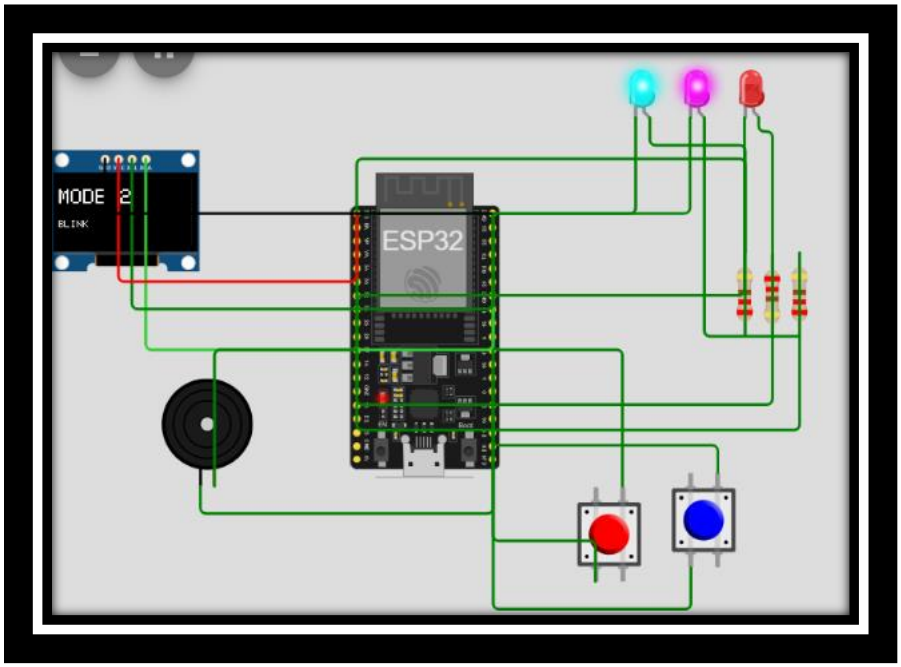
```
      timer = millis();
      fade += dir;
      if (fade >= 255 || fade <= 0) dir = -dir;
      analogWrite(LED1, fade);
      analogWrite(LED2, fade);
      analogWrite(LED3, fade);
    }
  }
}

// ==== Helper Functions ====
void setLEDs(int val) {
  analogWrite(LED1, val);
  analogWrite(LED2, val);
  analogWrite(LED3, val);
}

void showMode() {
  oled.clearDisplay();
  oled.setTextSize(2);
  oled.setTextColor(WHITE);
  oled.setCursor(0, 10);
  oled.print("MODE ");
  oled.println(mode + 1);
  oled.setTextSize(1);
  oled.setCursor(0, 40);

  if (mode == 0) oled.print("OFF");
  if (mode == 1) oled.print("BLINK");
  if (mode == 2) oled.print("ON");
  if (mode == 3) oled.print("FADE");

  oled.display();
}
```

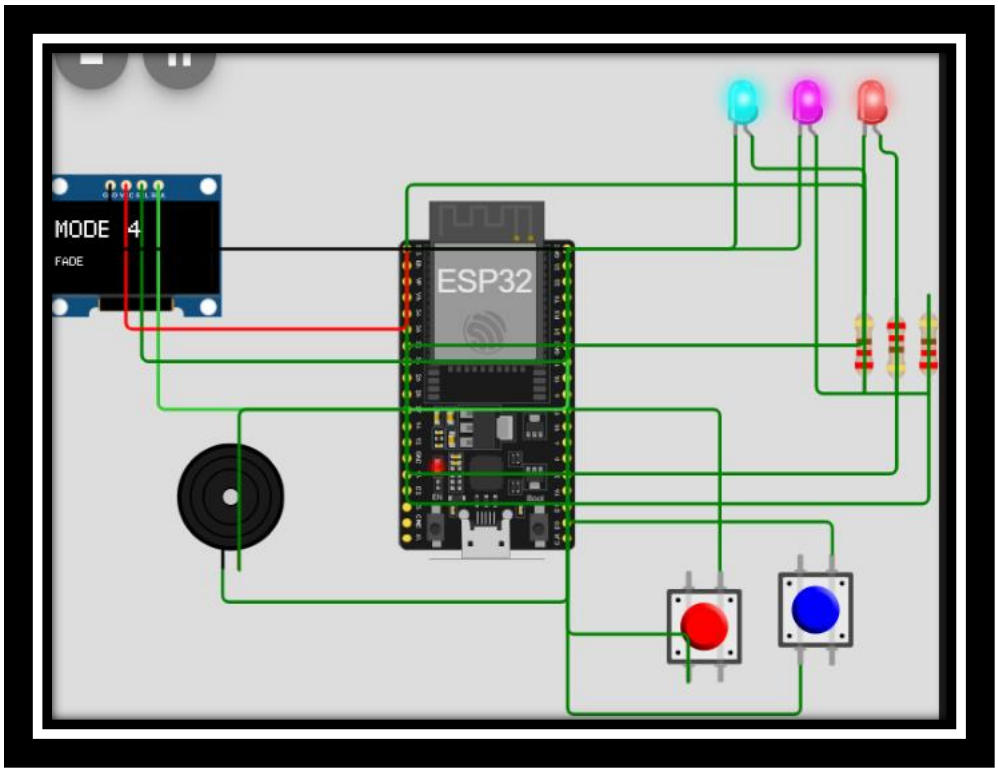
Output:

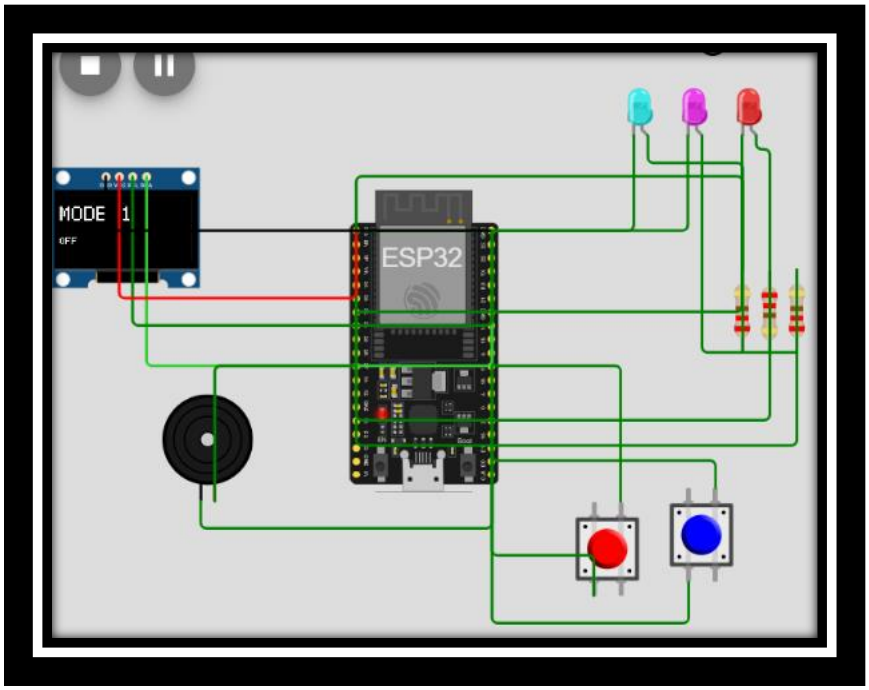





MODE 2
BLINK
ESP32


MODE 2
BLINK
ESP32





Wokwi Link: https://wokwi.com/projects/445502265516977153

Task B — 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

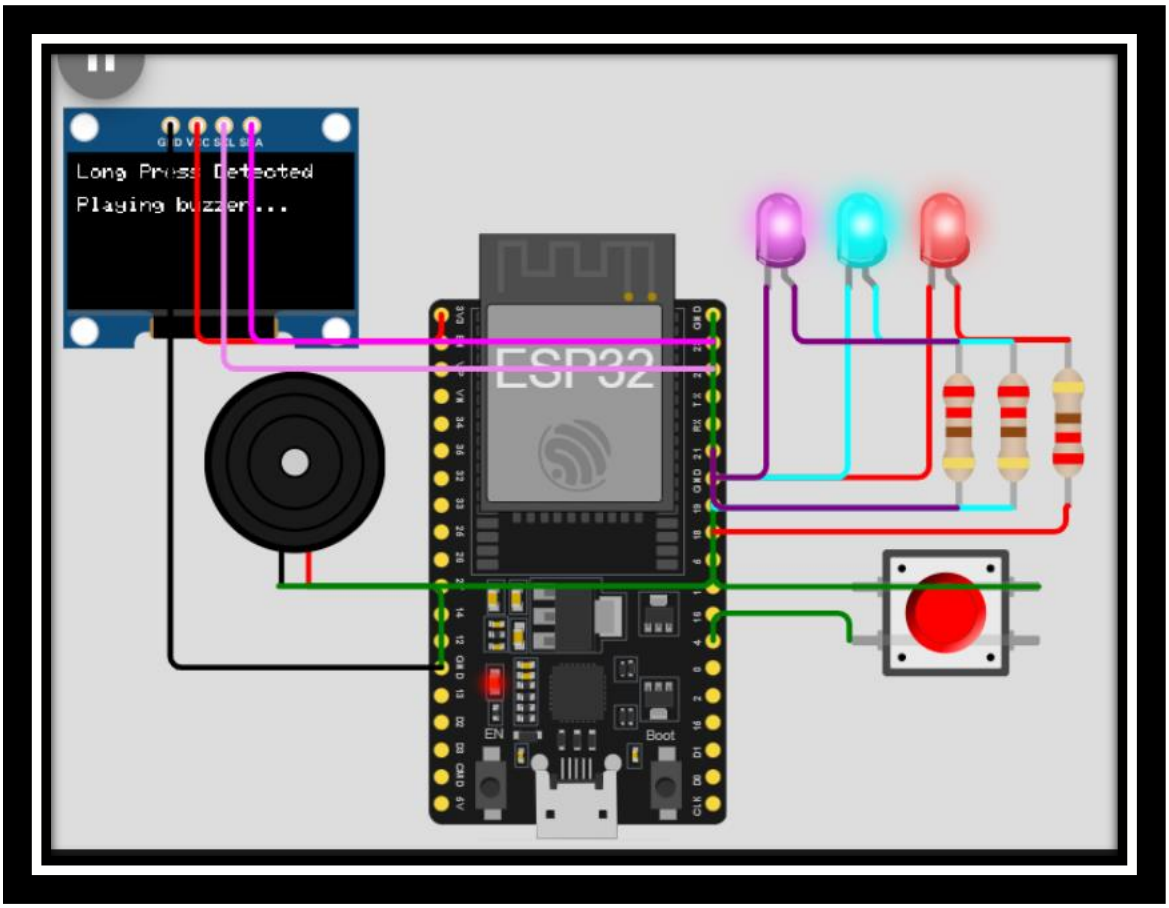
Code:

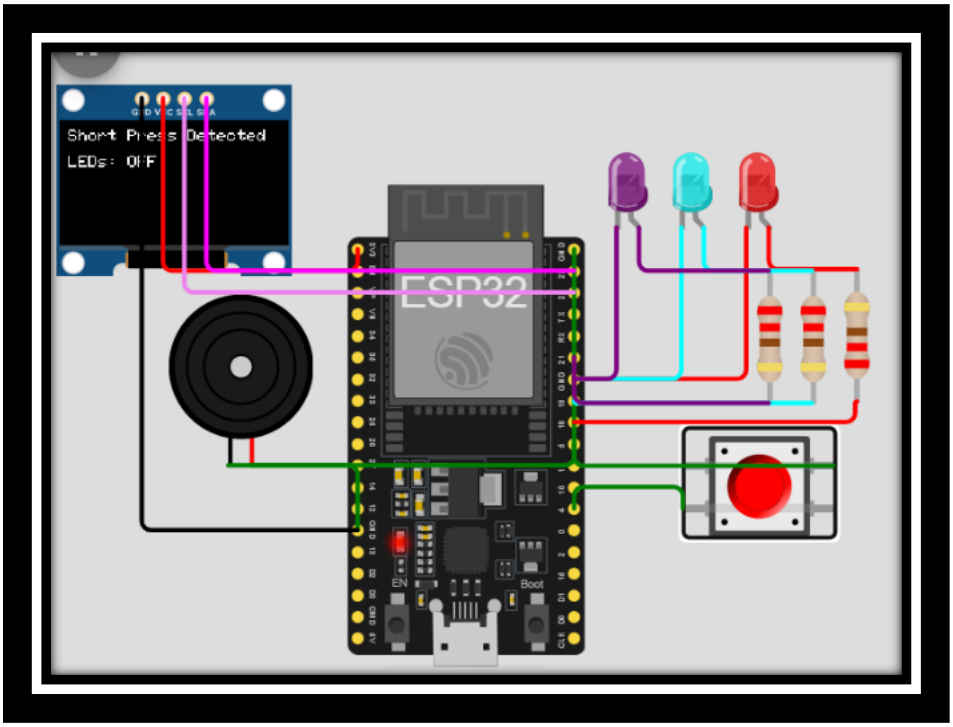
```
#include <Arduino.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

// ==== Pin Definitions (Updated) ====
#define BTN1 4     // Push button
#define LED1 18    // LED 1
#define LED2 19    // LED 2
#define LED3 21    // LED 3
#define BUZZ 5     // Buzzer

// ==== OLED Setup ====
Adafruit_SSD1306 oled(128, 64, &Wire, -1);

// ==== Variables ====
unsigned long pressStart = 0;
bool isPressed = false;
bool ledsOn = false;

void setup() {
  pinMode(BTN1, INPUT_PULLUP);  // Button with internal pull-up
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
  pinMode(BUZZ, OUTPUT);

  digitalWrite(LED1, LOW);
  digitalWrite(LED2, LOW);
```

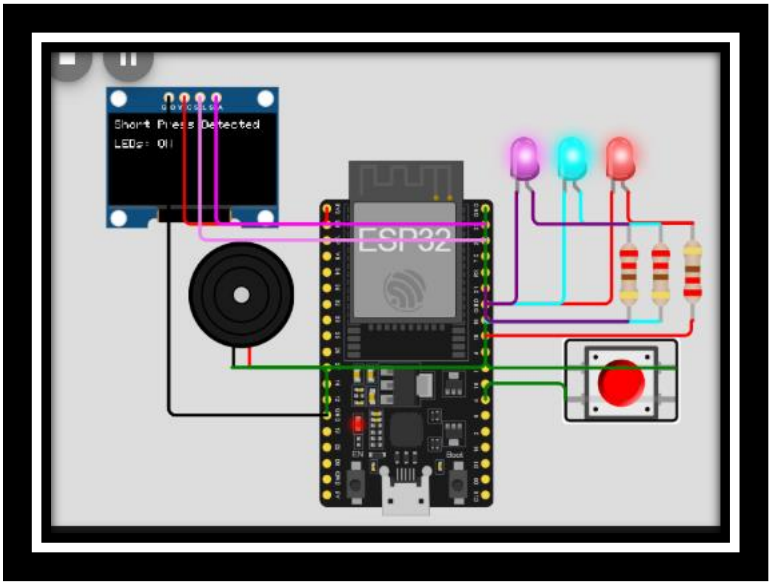
```
  digitalWrite(LED3, LOW);
  digitalWrite(BUZZ, LOW);

  // ==== Initialize OLED ====
  Wire.begin(23, 22); // SDA = 23, SCL = 22 (ESP32 default wiring in Wokwi)
  oled.begin(SSD1306_SWITCHCAPVCC, 0x3C);
  oled.clearDisplay();
  oled.setTextSize(1);
  oled.setTextColor(SSD1306_WHITE);
  oled.setCursor(0, 0);
  oled.println("Ready...");
  oled.display();
}

void loop() {
  int buttonState = digitalRead(BTN1);

  // Detect press start
  if (buttonState == LOW && !isPressed) {
    isPressed = true;
    pressStart = millis();
  }

  // Detect button release
  if (buttonState == HIGH && isPressed) {
    unsigned long duration = millis() - pressStart;
    isPressed = false;

    oled.clearDisplay();

    // Short press -> toggle LEDs
    if (duration < 1500) {
      ledsOn = !ledsOn;
      digitalWrite(LED1, ledsOn);
      digitalWrite(LED2, ledsOn);
      digitalWrite(LED3, ledsOn);

      oled.setCursor(0, 0);
      oled.println("Short Press Detected");
      oled.setCursor(0, 16);
      oled.println(ledsOn ? "LEDs: ON" : "LEDs: OFF");
      oled.display();
    }
    // Long press -> play buzzer
    else {
```

```
      oled.setCursor(0, 0);
      oled.println("Long Press Detected");
      oled.setCursor(0, 16);
      oled.println("Playing buzzer...");
      oled.display();

      tone(BUZZ, 1000, 500); // 1kHz tone for 0.5s
      delay(500);
      noTone(BUZZ);
    }
  }
}
```

Output:




Short Press Detected
LEDs: OFF
ESP32
EN
Boot


Short Press Detected
LEDs: ON
ESP32

Wokwi Link: https://wokwi.com/projects/445781194620551169

Hand Sketch: