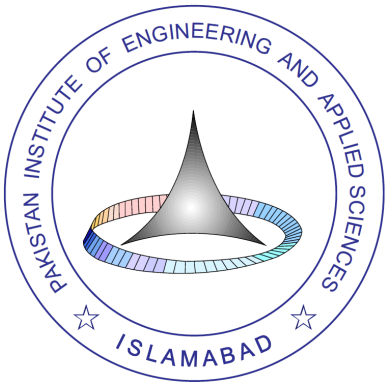

EE-434 Real Time Embedded Systems

PBL: Real Time Smart Helmet for Mining/Industrial Safety

Project Members:

Abdul Ahad (02-3-1-001-2022)

Muhammad Rehan Ali (02-3-1-060-2022)

Submitted to: Dr. Muhammad Aqil

Date of Submission: 19-12-2025

Department of Electrical Engineering
Pakistan Institute of Engineering and Applied Sciences

Abstract

Industrial and mining environments expose workers to multiple life-threatening hazards such as toxic gas leaks, extreme temperatures, high humidity, and accidental falls. This project presents the design and implementation of a Real-Time Smart Helmet using an ESP32 microcontroller integrated with DHT22, MQ-7, and MPU6050 sensors. The system continuously monitors environmental and physiological conditions and triggers a multi-stage alert mechanism when risk thresholds are violated. The alert sequence includes buzzer activation, followed by worker confirmation via pushbutton, and activates the vibration motor and emergency SOS transmission via WiFi to a command center if no response is detected within 10 seconds. The system demonstrates a complete IoT-based safety solution using sensor integration, FreeRTOS multitasking, WiFi communication, and emergency decision-making logic.

Contents

1.1 Project Objectives

The main objectives of this project include:

1. Real-time monitoring of temperature, humidity, CO gas levels, and worker activity(fall detection).
2. Implement a multi-stage alert system:
    - Stage 1: Warning buzzer
    - Stage 2: User acknowledgment
    - Stage 3: Emergency vibration + SOS message (if not acknowledged)
3. Design an IoT-based architecture using ESP32 to communicate with a command center via HTTP over WiFi.
4. Develop a scalable multi-tasking firmware using FreeRTOS tasks, queues, and semaphores.
5. Ensure worker safety through autonomous decision-making even if the worker is unconscious.
6. Create a modular and robust system suitable for mining and industrial applications.

1.2 Introduction

Mining and industrial fields remain among the most dangerous working environments. Workers face hazards such as toxic gases, unsafe heat levels, high humidity causing dehydration or heatstroke, and fall-related injuries. Traditional manual monitoring systems fail to ensure real-time safety. The Smart Helmet proposes a modern IoT-based approach:

- DHT22 monitors temperature & humidity.
- MQ-7 detects carbon monoxide presence, one of the most fatal mining gases.
- MPU6050 accelerometer detects abnormal movement or sudden falls.
- ESP32 handles sensing, processing, decision-making, and WiFi communication.
- Buzzer and vibration motor provide physical alerts.
- Push button allows workers to acknowledge alerts.
- HTTP-based communication enables remote monitoring at the command center.

The overall system improves safety, reduces response time, and increases chances of survival during accidents.

1.3 System Design and Procedure

1.3.1 Hardware Components

Table 1.1: Hardware Components and Their Purpose

Component	**Purpose**
ESP32	Microcontroller for sensor interfacing, data processing, and Wi-Fi communication
DHT22	Temperature and humidity sensing
MQ7	Carbon monoxide (CO) gas detection
MPU6050	Accelerometer & gyroscope for fall detection
Buzzer	Audible alert to notify the worker
Vibration Motor	Haptic alert for unconscious worker scenario
Push Button	User input to cancel alert

1.3.2 Procedure

1. **Sensor Calibration:** Thresholds for each sensor were set based on occupational safety standards:
   - CO level: ADC > 300 (Dangerous)
   - Temperature: $> 50°C$ (Dangerous)
   - Humidity: $> 90\%$ (Uncomfortable)
   - Fall detection: Magnitude > 2.5 g (Potential accident)
2. **System Initialization:**
   - Initialize ESP32 peripherals (GPIO, I2C, ADC, Wi-Fi).
   - Initialize sensors: DHT22, MQ7, MPU6050.
   - Configure FreeRTOS tasks for sensor monitoring, alert management, and status reporting.
3. **Continuous Monitoring:**
   - Each sensor runs in a separate FreeRTOS task.
   - Sensor data are compared against thresholds.
   - On threshold violation, an alert message is sent to the *alert management task* via a FreeRTOS queue.
4. **Alert Escalation Logic:**
   - Threshold violation $\rightarrow$ Buzzer ON, wait 10 seconds for cancel button press.
   - No response $\rightarrow$ Activate vibration motor and send SOS to command center.
   - Manual cancel allowed at any stage.
5. **Status Reporting:** Helmet periodically (every 30 seconds) sends status updates to the command center indicating system health and activity.

1.4 Experiment FlowChart

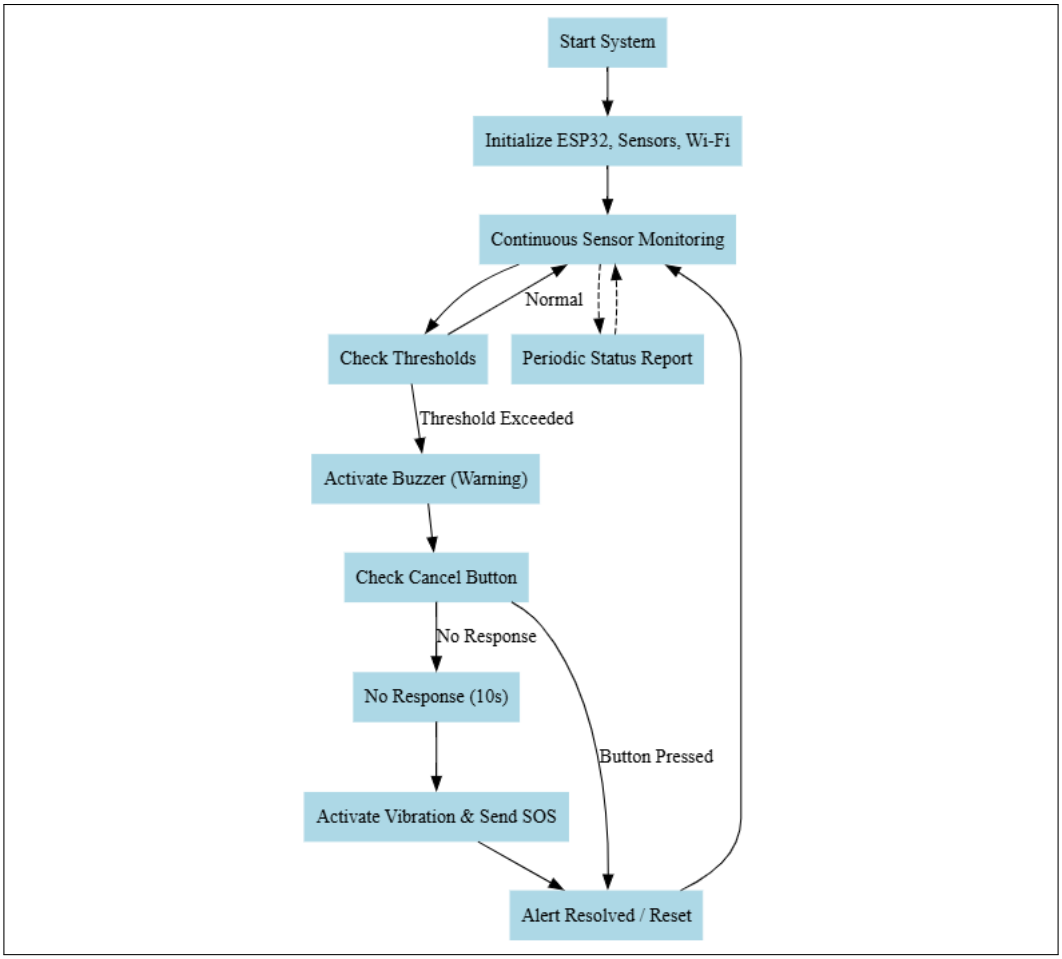

Figure 1.1: Experimental Flowchart for Real time Smart Helmet

1.5 Implementation

1.5.1 PBL Requirements Compliance

Requirement	Implementation in Proposed System
ESP32 as microcontroller	ESP32 executes FreeRTOS tasks, manages sensors, and communication
Input Output Interface	Push button for SOS or cancel input, buzzer and vibration motor for alert output
Sensor Interfacing (Analog/Digital)	IMU (I2C), Gas Sensor (Analog), DHT22 (Digital)
Interrupt Usage	IMU interrupt for fall detection and button interrupt for SOS trigger
Scheduler and RTOS Requirement	FreeRTOS based task scheduling and message queues for sensor data
Communication Interface	WiFi used for sending real time alerts to monitoring unit
Simulation/Testing Tool	Wokwi for circuit design and ESP-IDF for software simulation
Prototype Realization	Helmet mounted prototype for mining environment testing

1.5.2 Real-Time Operating System

FreeRTOS is used to enable concurrent monitoring and alert handling, allowing multiple sensor tasks and alert management tasks to run simultaneously without blocking each other.

1.5.3 Sensors Interfacing

DHT22:

```
    static void dht22_init(void) {
    gpio_set_direction(DHT22_PIN, GPIO_MODE_OUTPUT);
    gpio_set_level(DHT22_PIN, 1);
}

static int dht22_read(float *temperature, float *humidity) {
    uint8_t data[5] = {0};
    uint32_t timeout; // We are adding it to prevent infinite
       loops

    gpio_set_direction(DHT22_PIN, GPIO_MODE_OUTPUT);
    gpio_set_level(DHT22_PIN, 0);
    vTaskDelay(pdMS_TO_TICKS(18));
    gpio_set_level(DHT22_PIN, 1);
    ets_delay_us(30);
    gpio_set_direction(DHT22_PIN, GPIO_MODE_INPUT);

    timeout = 0;
```

```
18      while (gpio_get_level(DHT22_PIN) == 1 && timeout++ < 100)
           ets_delay_us(1);
19      if (timeout >= 100) return -1;
20
21      timeout = 0;
22      while (gpio_get_level(DHT22_PIN) == 0 && timeout++ < 100)
           ets_delay_us(1);
23      if (timeout >= 100) return -1;
24
25      timeout = 0;
26      while (gpio_get_level(DHT22_PIN) == 1 && timeout++ < 100)
           ets_delay_us(1);
27      if (timeout >= 100) return -1;
28
29      for (int i = 0; i < 40; i++) {
30          timeout = 0;
31          while (gpio_get_level(DHT22_PIN) == 0 && timeout++ <
               100) ets_delay_us(1);
32
33          ets_delay_us(30);
34
35          if (gpio_get_level(DHT22_PIN) == 1) {
36              data[i / 8] |= (1 << (7 - (i % 8)));
37          }
38
39          timeout = 0;
40          while (gpio_get_level(DHT22_PIN) == 1 && timeout++ <
               100) ets_delay_us(1);
41      }
42
43      if (data[4] != ((data[0] + data[1] + data[2] + data[3]) &
           0xFF)) {
44          return -1;
45      }
46
47      *humidity = ((data[0] << 8) | data[1]) / 10.0;
48      *temperature = (((data[2] & 0x7F) << 8) | data[3]) / 10.0;
49      if (data[2] & 0x80) *temperature *= -1;
50
51      return 0;
52  }
```

Listing 1.1: DHT22 Sensor Interfacing

MQ7:

```
1           static void gas_sensor_task(void *pvParameters) {
2       int adc_value;
3
4       while (1) {
5           adc_oneshot_read(adc1_handle, MQ7_ADC_CHANNEL, &
               adc_value);
```

```
        ESP_LOGI(TAG, " Gas Sensor ADC: %d", adc_value);

        if (adc_value > CO_THRESHOLD) {
            alert_msg_t msg = {
                .value = (float)adc_value,
                .timestamp = xTaskGetTickCount(),
                .is_critical = true
            };
            strcpy(msg.alert_type, "DANGEROUS_CO_LEVEL");
            xQueueSend(alert_queue, &msg, 0);
            ESP_LOGW(TAG, " Dangerous CO detected! ADC: %d",
               adc_value);
        }

        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}
```

Listing 1.2: MQ7 Gas Sensor

MPU6050:

```
    static esp_err_t i2c_master_init(void) {
    i2c_config_t conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = I2C_MASTER_SDA_IO,
        .scl_io_num = I2C_MASTER_SCL_IO,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = I2C_MASTER_FREQ_HZ,
    };
    esp_err_t err = i2c_param_config(I2C_NUM_0, &conf);
    if (err != ESP_OK) return err;
    return i2c_driver_install(I2C_NUM_0, conf.mode, 0, 0, 0);
}

static esp_err_t mpu6050_write_byte(uint8_t reg, uint8_t data)
    {
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (MPU6050_ADDR << 1) |
       I2C_MASTER_WRITE, true);
    i2c_master_write_byte(cmd, reg, true);
    i2c_master_write_byte(cmd, data, true);
    i2c_master_stop(cmd);
    esp_err_t ret = i2c_master_cmd_begin(I2C_NUM_0, cmd,
       pdMS_TO_TICKS(1000));
    i2c_cmd_link_delete(cmd);
    return ret;
}

```

```
27 static esp_err_t mpu6050_read_bytes(uint8_t reg, uint8_t *data
   , size_t len) {
28     i2c_cmd_handle_t cmd = i2c_cmd_link_create();
29     i2c_master_start(cmd);
30     i2c_master_write_byte(cmd, (MPU6050_ADDR << 1) |
          I2C_MASTER_WRITE, true);
31     i2c_master_write_byte(cmd, reg, true);
32     i2c_master_start(cmd);
33     i2c_master_write_byte(cmd, (MPU6050_ADDR << 1) |
          I2C_MASTER_READ, true);
34     i2c_master_read(cmd, data, len, I2C_MASTER_LAST_NACK);
35     i2c_master_stop(cmd);
36     esp_err_t ret = i2c_master_cmd_begin(I2C_NUM_0, cmd,
          pdMS_TO_TICKS(1000));
37     i2c_cmd_link_delete(cmd);
38     return ret;
39 }
40
41 static void mpu6050_init(void) {
42     mpu6050_write_byte(0x6B, 0x00);
43     vTaskDelay(pdMS_TO_TICKS(100));
44     mpu6050_write_byte(0x1C, 0x10);
45     mpu6050_write_byte(0x1B, 0x08);
46     ESP_LOGI(TAG, "MPU6050 initialized");
47 }
48
49 static void mpu6050_read_accel(float *ax, float *ay, float *az
   ) {
50     uint8_t data[6];
51
52     *ax = 0.0;
53     *ay = 0.0;
54     *az = 0.0;
55
56     if (mpu6050_read_bytes(0x3B, data, 6) == ESP_OK) {
57         int16_t raw_ax = (data[0] << 8) | data[1];
58         int16_t raw_ay = (data[2] << 8) | data[3];
59         int16_t raw_az = (data[4] << 8) | data[5];
60
61         *ax = raw_ax / 4096.0;
62         *ay = raw_ay / 4096.0;
63         *az = raw_az / 4096.0;
64     }
65 }
```

Listing 1.3: MPU6050 Accelerometer+Gyroscope

1.5.4 Alert Management

- Implemented using a **state machine** with states: NORMAL → WARNING → EMERGENCY.
- In the WARNING state, the buzzer is activated to alert the worker.

- If no response is detected, the system escalates to EMERGENCY, activating the vibration motor and sending an SOS to the command center.

```
1   static void alert_management_task(void *pvParameters) {
2   alert_msg_t current_alert;
3   bool has_active_alert = false;
4
5   while (1) {
6       // Check if we have a new alert
7       if (xQueueReceive(alert_queue, &current_alert,
            pdMS_TO_TICKS(BUZZER_CHECK_MS))) {
8           has_active_alert = true;
9           current_alert_state = ALERT_STATE_WARNING;
10          alert_start_time = xTaskGetTickCount();
11          cancel_button_pressed = false;
12
13          ESP_LOGW(TAG, " ALERT TRIGGERED: %s - Value: %.2f"
                , current_alert.alert_type, current_alert.value
                );
14          ESP_LOGW(TAG, " Buzzer activated! Press button to
                cancel within 10 seconds.");
15      }
16
17      // Handle active alert state
18      if (has_active_alert && current_alert_state ==
            ALERT_STATE_WARNING) {
19          // Activate buzzer continuously
20          gpio_set_level(BUZZER_PIN, 1);
21
22          // Check if cancel button was pressed
23          if (cancel_button_pressed) {
24              ESP_LOGI(TAG, "Alert cancelled by user.
                    Situation under control.");
25              gpio_set_level(BUZZER_PIN, 0);
26              gpio_set_level(VIBRATION_PIN, 0);
27              current_alert_state = ALERT_STATE_NORMAL;
28              has_active_alert = false;
29              cancel_button_pressed = false;
30              continue;
31          }
32
33          // Check if 10 seconds elapsed without cancel
34          uint32_t elapsed_time = (xTaskGetTickCount() -
                alert_start_time) * portTICK_PERIOD_MS;
35          if (elapsed_time >= ALERT_TIMEOUT_MS) {
36              // Escalate to EMERGENCY
37              current_alert_state = ALERT_STATE_EMERGENCY;
38
39              ESP_LOGE(TAG, "EMERGENCY! No response from
                    user for 10 seconds!");
40              ESP_LOGE(TAG, "Activating vibration motor and
                    sending SOS to command center!");
41
```

```
                // Activate vibration motor
                gpio_set_level(VIBRATION_PIN, 1);

                // Send SOS to command center
                send_sos_to_command_center(current_alert.
                    alert_type, current_alert.value);

            }
        }

        // In EMERGENCY state, keep both buzzer and vibration
            active
        if (current_alert_state == ALERT_STATE_EMERGENCY) {
            gpio_set_level(BUZZER_PIN, 1);
            gpio_set_level(VIBRATION_PIN, 1);

            // Optional: Allow manual reset even in emergency
            if (cancel_button_pressed) {
                ESP_LOGI(TAG, "Emergency alert manually
                    cancelled");
                gpio_set_level(BUZZER_PIN, 0);
                gpio_set_level(VIBRATION_PIN, 0);
                current_alert_state = ALERT_STATE_NORMAL;
                has_active_alert = false;
                cancel_button_pressed = false;
            }
        }

        vTaskDelay(pdMS_TO_TICKS(BUZZER_CHECK_MS));
    }
}
```

Listing 1.4: Alert Management Task

1.5.5 Wi-Fi Communication

- ESP32 connects to a local Wi-Fi network.
- HTTP POST requests are sent to the command center API endpoints for both emergency SOS alerts and periodic status reports.

```
    static void wifi_init_sta(void) {
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    esp_netif_create_default_wifi_sta();

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    ESP_ERROR_CHECK(esp_event_handler_register(WIFI_EVENT,
        ESP_EVENT_ANY_ID, &wifi_event_handler, NULL));
    ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT,
        IP_EVENT_STA_GOT_IP, &wifi_event_handler, NULL));
```

```
11
12      wifi_config_t wifi_config = {
13          .sta = {
14              .ssid = WIFI_SSID,
15              .password = WIFI_PASSWORD,
16          },
17      };
18      ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
19      ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &
            wifi_config));
20      ESP_ERROR_CHECK(esp_wifi_start());
21
22      ESP_LOGI(TAG, "WiFi initialization complete");
23  }
24
25  // Send SOS to Command Center
26  static esp_err_t send_sos_to_command_center(const char*
        alert_type, float value) {
27      char url[256];
28      snprintf(url, sizeof(url), "http://%s:%s/api/sos",
            COMMAND_CENTER_IP, COMMAND_CENTER_PORT);
29
30      char post_data[512];
31      snprintf(post_data, sizeof(post_data),
32               "{\"helmet_id\":\"%s\",\"alert_type\":\"%s\",\"
                     value\":%.2f,\"timestamp\":%lu,\"status\":\"
                     EMERGENCY\"}",
33               helmet_id, alert_type, value, (unsigned long)(
                     xTaskGetTickCount() / 1000));
34
35      esp_http_client_config_t config = {
36          .url = url,
37          .method = HTTP_METHOD_POST,
38          .timeout_ms = 5000,
39      };
40
41      esp_http_client_handle_t client = esp_http_client_init(&
            config);
42      esp_http_client_set_header(client, "Content-Type", "
            application/json");
43      esp_http_client_set_post_field(client, post_data, strlen(
            post_data));
44
45      esp_err_t err = esp_http_client_perform(client);
46
47      if (err == ESP_OK) {
48          ESP_LOGI(TAG, "SOS sent successfully. Status: %d",
                esp_http_client_get_status_code(client));
49      } else {
50          ESP_LOGE(TAG, "Failed to send SOS: %s",
                esp_err_to_name(err));
51      }
52
```

```
53      esp_http_client_cleanup(client);
54      return err;
55  }
```

Listing 1.5: Wifi Initialization and SOS signal function

1.5.6 Schematics:

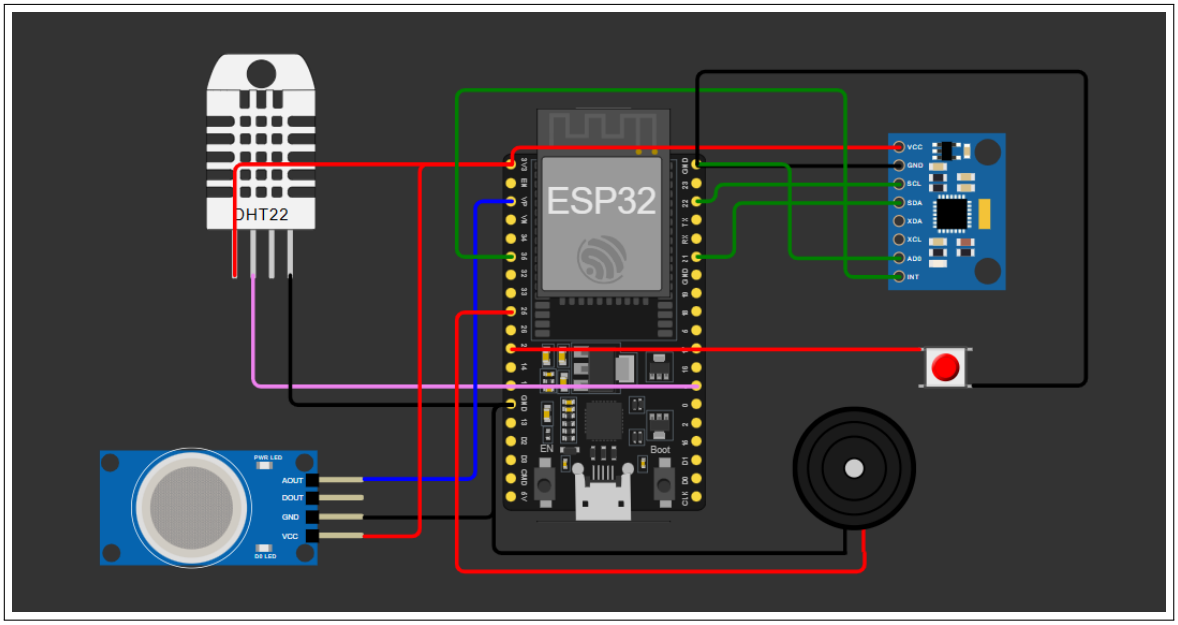

Figure 1.2: Circuit Schematics on Wokwi

1.5.7 Hardware Implementation:

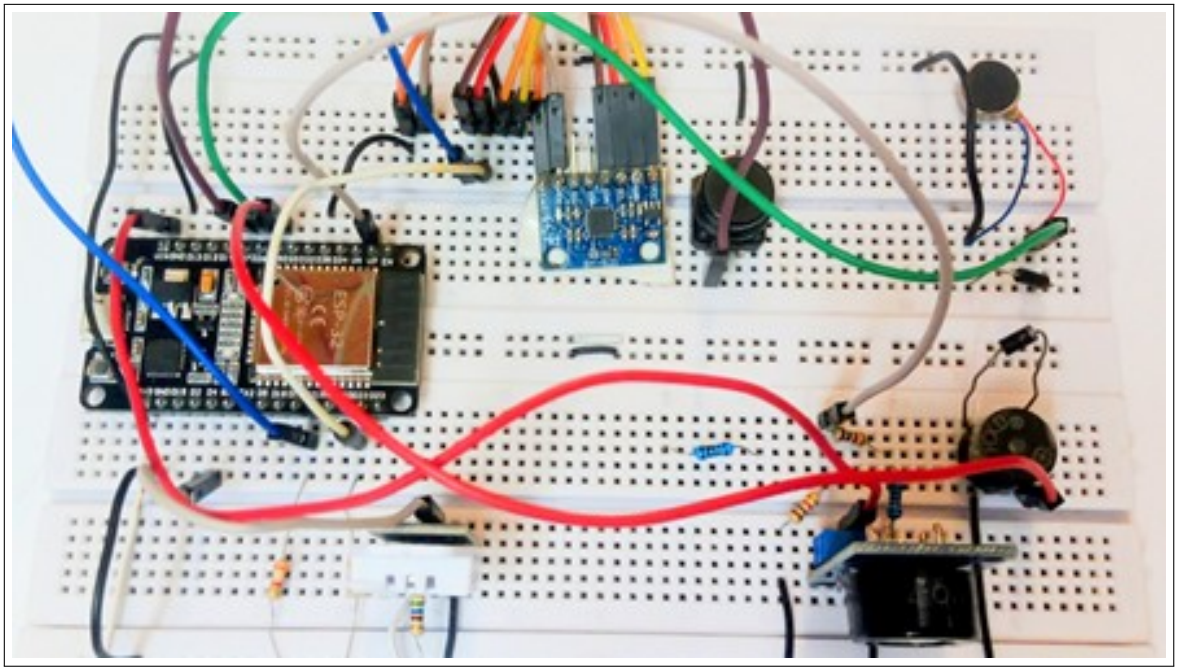

Figure 1.3: Hardware Implementation of Smart Helmet

1.6 Results

In this section, the results obtained from performing the laboratory tasks are presented. The correctness of each implementation was verified by visually observing the Serial Monitor(Console) of the Visual Studio.

1.6.1 Simulation Logs at Wokwi

```
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp
_drv:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:1156
load:0x40078000,len:11456
ho 0 tail 12 room 4
load:0x40080400,len:2972
entry 0x400805dc
CO ADC Value (raw, not ppm): 3628
Temp: 24.00°C  | Humidity: 40.00
Accel Magnitude (m/s²): 9.81
Change in g (relative to 9.81): 0.00
CO ADC Value (raw, not ppm): 3628
```

Figure 1.4: Output logs for Schematics in figure 1.1

1.6.2 Output Logs

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  ESP-IDF

2.168.137.1
I (1932) SmartHelmet: Got IP: 192.168.137.150
I (2412) SmartHelmet: Accel: 0.17, -0.03, 1.10 (Mag: 1.11 g)
I (2902) SmartHelmet: Gas Sensor ADC: 0
I (2912) SmartHelmet: Accel: 0.16, -0.03, 1.09 (Mag: 1.11 g)
I (3412) SmartHelmet: Accel: 0.16, -0.02, 1.10 (Mag: 1.11 g)
I (3912) SmartHelmet: Accel: 0.17, -0.02, 1.10 (Mag: 1.12 g)
I (3932) SmartHelmet: Temp: 19.5°C, Humidity: 57.7%
I (4412) SmartHelmet: Accel: 0.16, -0.03, 1.10 (Mag: 1.11 g)
I (4902) SmartHelmet: Gas Sensor ADC: 0
I (4912) SmartHelmet: Accel: 0.17, -0.03, 1.10 (Mag: 1.11 g)
I (5412) SmartHelmet: Accel: 0.17, -0.03, 1.10 (Mag: 1.12 g)
I (5912) SmartHelmet: Accel: 0.17, -0.03, 1.10 (Mag: 1.11 g)
I (6412) SmartHelmet: Accel: 0.17, -0.02, 1.09 (Mag: 1.10 g)
```

Figure 1.5: Normal Working: System in Normal Condition

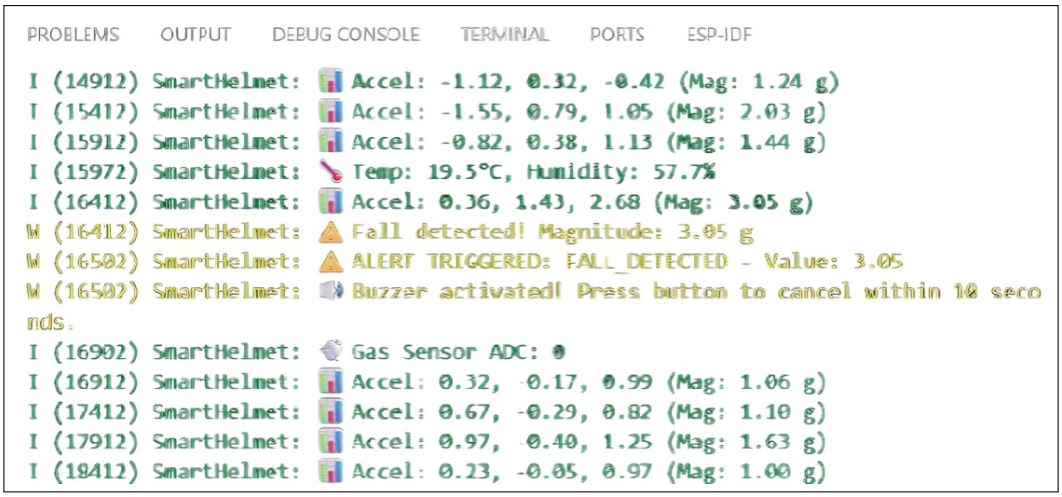

Figure 1.6: Alert Condition due to Fall detection: Buzzer is activated

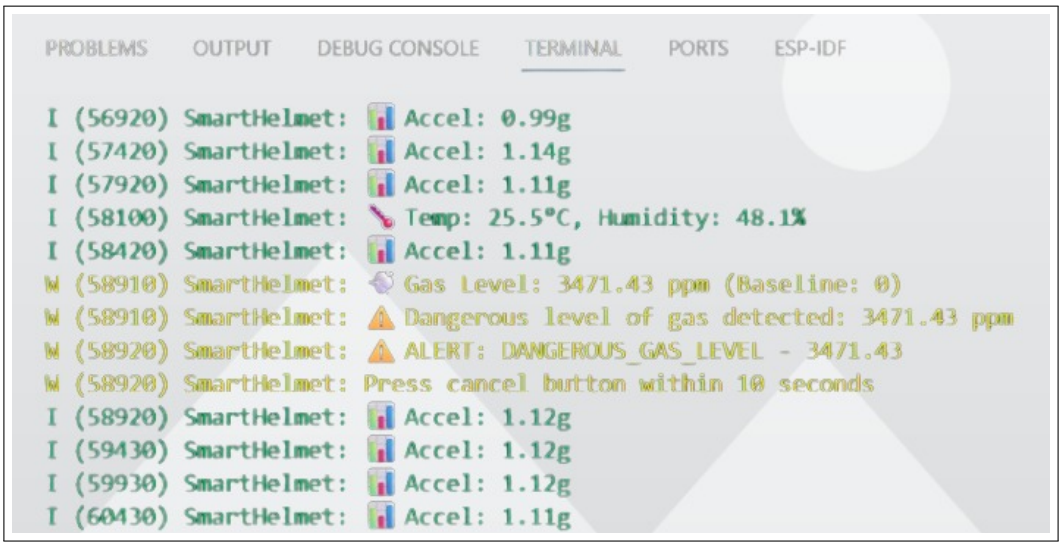

Figure 1.7: Alert Condition due to Gas Detection: Buzzer is activated

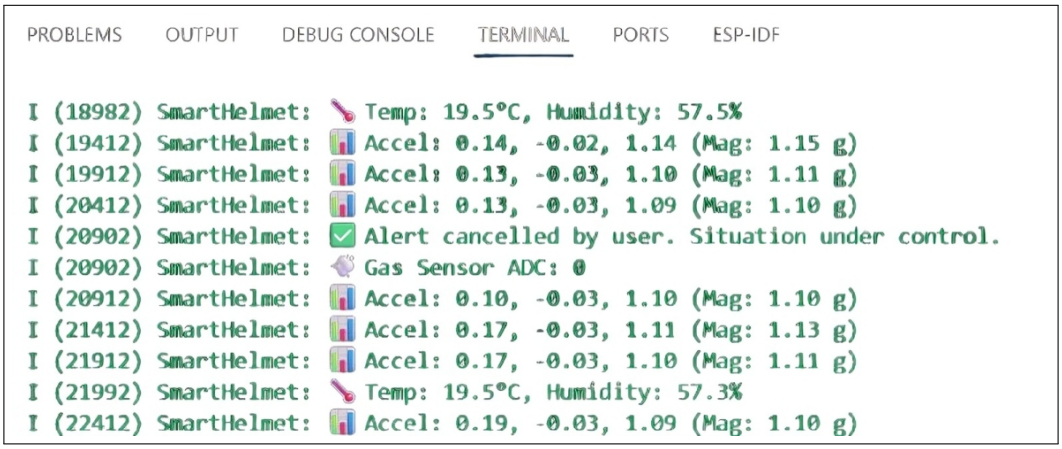

Figure 1.8: Alert Cancel by the User by Pressing the Button

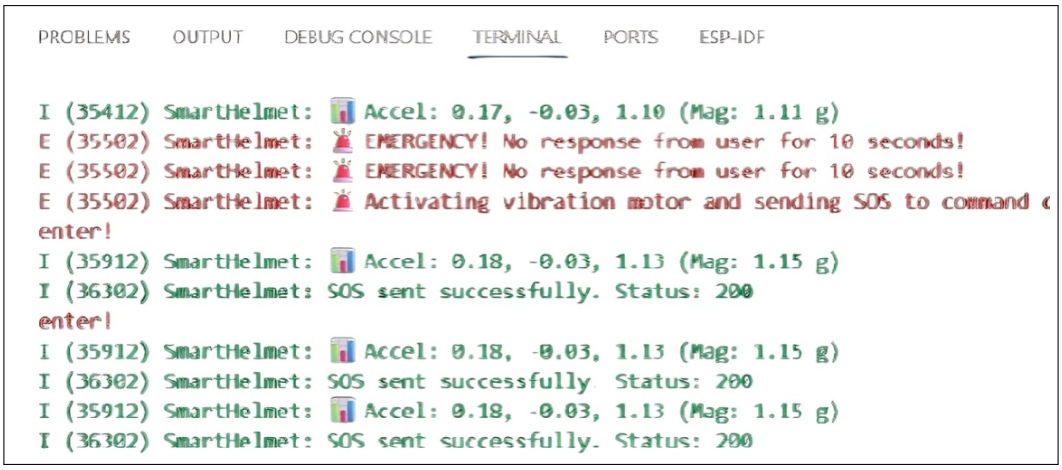

Figure 1.9: User Failed to Respond,Vibration Motor ON and SOS Sent

1.6.3 Command Center GUI

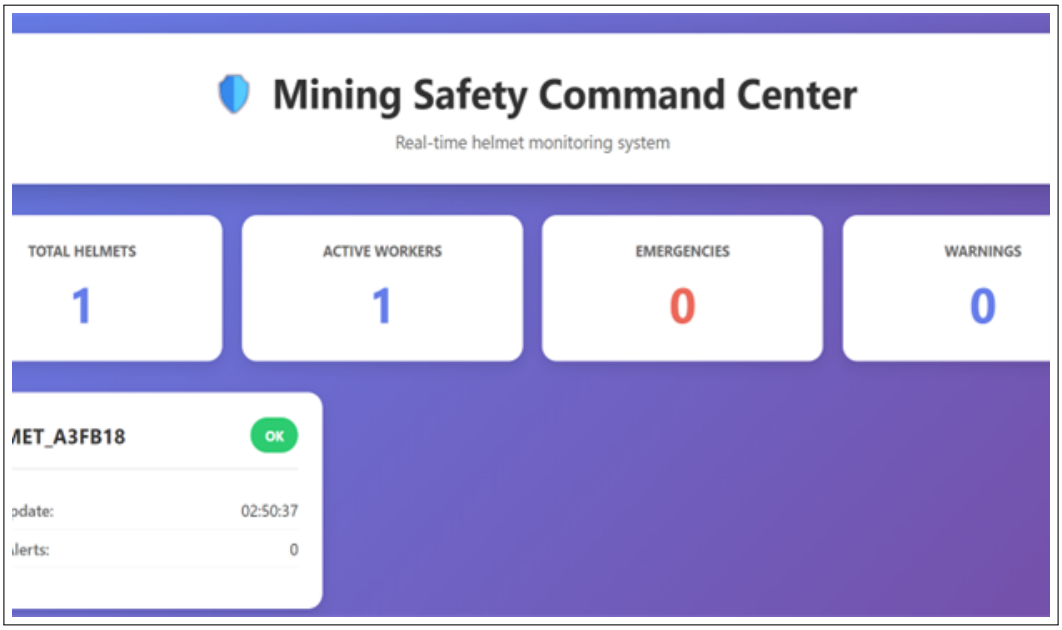

Figure 1.10: Webpage GUI of Command Center

1.7 Discussion

The Smart Helmet system was tested to understand the behavior of each sensor and how the system responds in real time. Overall, the design worked as expected, but each sensor showed different response characteristics.

The **DHT22** temperature and humidity sensor provided stable readings, although it updates slowly. This slow update rate is acceptable because temperature and humidity do not change rapidly in normal conditions. The readings remained consistent, and the sensor performed reliably during continuous monitoring.

The **MPU6050** accelerometer and gyroscope showed very fast and sensitive responses. Even small head movements were detected accurately. This helped the fall-detection feature work smoothly. Since the MPU6050 communicates using the I2C interface, the data transfer was quick, and FreeRTOS handled the continuous

stream without delays. The sensor's quick response makes it suitable for detecting sudden events such as falls, slips, or impacts.

The **MQ-7 gas sensor** had the slowest response among all sensors. Being a metal-oxide semiconductor (MOS) sensor, it requires a long warm-up period before giving accurate readings. The MQ-7 needs **22-48 hours of continuous heating** to stabilize. Before this warm-up time, its output is not available. Because of this behavior, the gas-monitoring feature can only be tested properly after the sensor has been fully conditioned. This requirement was considered in the project plan and the testing schedule.

The alert system, based on a state machine (Normal → Warning → Emergency), worked as intended. The buzzer activated correctly in the warning state, and the emergency state triggered the vibration motor and the SOS message. The Wi-Fi communication through the ESP32 was stable, and HTTP POST requests were delivered to the command center without noticeable delays.

Overall, the sensors and the ESP32 worked well together under FreeRTOS. Sensor reading, alert handling, and Wi-Fi communication tasks ran smoothly in parallel, showing that the system is capable of real-time monitoring and quick response in hazardous conditions.

1.8 Conclusion

The project achieved its goal of creating a Smart Helmet that monitors temperature, humidity, motion, and carbon monoxide levels while sending real-time alerts. The DHT22 and MPU6050 performed reliably, and although the MQ-7 requires a long warm-up time, it will function correctly once stabilized.

In summary, the system is effective, simple to use, and suitable for further development into a practical safety device for industrial and mining workers.

References

[1] R. Barry, *Mastering the FreeRTOS Real Time Kernel*, 10th ed. Real Time Engineers Ltd., 2019.

[2] J. W. Valvano, *Embedded Systems: Real-Time Interfacing to the ARM Cortex-M Microcontrollers*, 2nd ed. CreateSpace Independent Publishing, 2017.

[3] E. Systems, *ESP-IDF Programming Guide*, 2025, available online. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/

[4] ——, *ESP32-WROOM-32 Datasheet*, 2023, technical Reference Manual. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32$_d atasheet_e n.pdf$

Appendix

Code

```
   #include <stdio.h>
#include <string.h>
#include <math.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "freertos/semphr.h"
#include "driver/gpio.h"
#include "esp_adc/adc_oneshot.h"
#include "driver/i2c.h"
#include "esp_log.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "nvs_flash.h"
#include "esp_http_client.h"
#include "rom/ets_sys.h"
#include "esp_mac.h"

// Pin Definitions
#define DHT22_PIN           GPIO_NUM_4
#define MQ7_ADC_CHANNEL     ADC_CHANNEL_0  // GPIO36
#define BUZZER_PIN          GPIO_NUM_25
#define VIBRATION_PIN       GPIO_NUM_26
#define CANCEL_BUTTON_PIN   GPIO_NUM_27

// I2C Configuration for MPU6050
#define I2C_MASTER_SCL_IO   GPIO_NUM_22
#define I2C_MASTER_SDA_IO   GPIO_NUM_21
#define I2C_MASTER_FREQ_HZ  100000
#define MPU6050_ADDR        0x68

// Threshold Values
#define CO_THRESHOLD        300     // ADC value for dangerous
    CO level
#define TEMP_THRESHOLD      50.0    //  C  - dangerous
   temperature
#define HUMIDITY_THRESHOLD  90.0    // % - dangerous humidity
   (above 85% is very uncomfortable)
```

```
#define FALL_THRESHOLD      2.5     // g-force for fall detection

// Alert Timing
#define ALERT_TIMEOUT_MS    10000
#define BUZZER_CHECK_MS     100

// Task Priorities
#define SENSOR_TASK_PRIORITY    5
#define ALERT_TASK_PRIORITY     7
#define COMM_TASK_PRIORITY      6

// WiFi Configuration
#define WIFI_SSID           "Boss"
#define WIFI_PASSWORD       "11223344"
#define COMMAND_CENTER_IP   "192.168.137.1"  // Change to your server IP
#define COMMAND_CENTER_PORT "8080"

static const char *TAG = "SmartHelmet";

// Global Variables and Queues
QueueHandle_t alert_queue;
SemaphoreHandle_t i2c_mutex;
adc_oneshot_unit_handle_t adc1_handle;

// Alert State Management
typedef enum {
    ALERT_STATE_NORMAL,
    ALERT_STATE_WARNING,      // Buzzer active, waiting for cancel
    ALERT_STATE_EMERGENCY     // SOS sent to command center
} alert_state_t;

volatile alert_state_t current_alert_state = ALERT_STATE_NORMAL;
volatile bool cancel_button_pressed = false;
volatile uint32_t alert_start_time = 0;

// Unique Helmet ID (MAC-based)
char helmet_id[32] = {0};

// Alert Message Structure
typedef struct {
    char alert_type[32];
    float value;
    uint32_t timestamp;
    bool is_critical;
} alert_msg_t;

// WiFi event handler
static void wifi_event_handler(void* arg, esp_event_base_t event_base,
```

```
84                                  int32_t event_id, void*
                                       event_data) {
85      if (event_base == WIFI_EVENT && event_id ==
           WIFI_EVENT_STA_START) {
86          esp_wifi_connect();
87      } else if (event_base == WIFI_EVENT && event_id ==
           WIFI_EVENT_STA_DISCONNECTED) {
88          ESP_LOGW(TAG, "WiFi disconnected, reconnecting...");
89          esp_wifi_connect();
90      } else if (event_base == IP_EVENT && event_id ==
           IP_EVENT_STA_GOT_IP) {
91          ip_event_got_ip_t* event = (ip_event_got_ip_t*)
               event_data;
92          ESP_LOGI(TAG, "Got IP: " IPSTR, IP2STR(&event->ip_info
               .ip));
93      }
94  }
95
96  // Initialize WiFi
97  static void wifi_init_sta(void) {
98      ESP_ERROR_CHECK(esp_netif_init());
99      ESP_ERROR_CHECK(esp_event_loop_create_default());
100     esp_netif_create_default_wifi_sta();
101
102     wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
103     ESP_ERROR_CHECK(esp_wifi_init(&cfg));
104
105     ESP_ERROR_CHECK(esp_event_handler_register(WIFI_EVENT,
           ESP_EVENT_ANY_ID, &wifi_event_handler, NULL));
106     ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT,
           IP_EVENT_STA_GOT_IP, &wifi_event_handler, NULL));
107
108     wifi_config_t wifi_config = {
109         .sta = {
110             .ssid = WIFI_SSID,
111             .password = WIFI_PASSWORD,
112         },
113     };
114     ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
115     ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &
           wifi_config));
116     ESP_ERROR_CHECK(esp_wifi_start());
117
118     ESP_LOGI(TAG, "WiFi initialization complete");
119 }
120
121 // Send SOS to Command Center
122 static esp_err_t send_sos_to_command_center(const char*
       alert_type, float value) {
123     char url[256];
124     snprintf(url, sizeof(url), "http://%s:%s/api/sos",
           COMMAND_CENTER_IP, COMMAND_CENTER_PORT);
125
```

```
126     char post_data[512];
127     snprintf(post_data, sizeof(post_data),
128              "{\"helmet_id\":\"%s\",\"alert_type\":\"%s\",\"
                    value\":%.2f,\"timestamp\":%lu,\"status\":\"
                    EMERGENCY\"}",
129              helmet_id, alert_type, value, (unsigned long)(
                    xTaskGetTickCount() / 1000));
130
131     esp_http_client_config_t config = {
132         .url = url,
133         .method = HTTP_METHOD_POST,
134         .timeout_ms = 5000,
135     };
136
137     esp_http_client_handle_t client = esp_http_client_init(&
            config);
138     esp_http_client_set_header(client, "Content-Type", "
            application/json");
139     esp_http_client_set_post_field(client, post_data, strlen(
            post_data));
140
141     esp_err_t err = esp_http_client_perform(client);
142
143     if (err == ESP_OK) {
144         ESP_LOGI(TAG, "SOS sent successfully. Status: %d",
                esp_http_client_get_status_code(client));
145     } else {
146         ESP_LOGE(TAG, "Failed to send SOS: %s",
                esp_err_to_name(err));
147     }
148
149     esp_http_client_cleanup(client);
150     return err;
151 }
152
153 // DHT22 Functions
154 static void dht22_init(void) {
155     gpio_set_direction(DHT22_PIN, GPIO_MODE_OUTPUT);
156     gpio_set_level(DHT22_PIN, 1);
157 }
158
159 static int dht22_read(float *temperature, float *humidity) {
160     uint8_t data[5] = {0};
161     uint32_t timeout; // We are adding it to prevent infinite
            loops
162
163     gpio_set_direction(DHT22_PIN, GPIO_MODE_OUTPUT);
164     gpio_set_level(DHT22_PIN, 0);
165     vTaskDelay(pdMS_TO_TICKS(18));
166     gpio_set_level(DHT22_PIN, 1);
167     ets_delay_us(30);
168     gpio_set_direction(DHT22_PIN, GPIO_MODE_INPUT);
169
```

```
170     timeout = 0;
171     while (gpio_get_level(DHT22_PIN) == 1 && timeout++ < 100)
           ets_delay_us(1);
172     if (timeout >= 100) return -1;
173 
174     timeout = 0;
175     while (gpio_get_level(DHT22_PIN) == 0 && timeout++ < 100)
           ets_delay_us(1);
176     if (timeout >= 100) return -1;
177 
178     timeout = 0;
179     while (gpio_get_level(DHT22_PIN) == 1 && timeout++ < 100)
           ets_delay_us(1);
180     if (timeout >= 100) return -1;
181 
182     for (int i = 0; i < 40; i++) {
183         timeout = 0;
184         while (gpio_get_level(DHT22_PIN) == 0 && timeout++ <
              100) ets_delay_us(1);
185 
186         ets_delay_us(30);
187 
188         if (gpio_get_level(DHT22_PIN) == 1) {
189             data[i / 8] |= (1 << (7 - (i % 8)));
190         }
191 
192         timeout = 0;
193         while (gpio_get_level(DHT22_PIN) == 1 && timeout++ <
              100) ets_delay_us(1);
194     }
195 
196     if (data[4] != ((data[0] + data[1] + data[2] + data[3]) &
           0xFF)) {
197         return -1;
198     }
199 
200     *humidity = ((data[0] << 8) | data[1]) / 10.0;
201     *temperature = (((data[2] & 0x7F) << 8) | data[3]) / 10.0;
202     if (data[2] & 0x80) *temperature *= -1;
203 
204     return 0;
205 }
206 
207 // I2C and MPU6050 Functions
208 static esp_err_t i2c_master_init(void) {
209     i2c_config_t conf = {
210         .mode = I2C_MODE_MASTER,
211         .sda_io_num = I2C_MASTER_SDA_IO,
212         .scl_io_num = I2C_MASTER_SCL_IO,
213         .sda_pullup_en = GPIO_PULLUP_ENABLE,
214         .scl_pullup_en = GPIO_PULLUP_ENABLE,
215         .master.clk_speed = I2C_MASTER_FREQ_HZ,
216     };
```

```
217     esp_err_t err = i2c_param_config(I2C_NUM_0, &conf);
218     if (err != ESP_OK) return err;
219     return i2c_driver_install(I2C_NUM_0, conf.mode, 0, 0, 0);
220 }
221
222 static esp_err_t mpu6050_write_byte(uint8_t reg, uint8_t data)
        {
223     i2c_cmd_handle_t cmd = i2c_cmd_link_create();
224     i2c_master_start(cmd);
225     i2c_master_write_byte(cmd, (MPU6050_ADDR << 1) |
          I2C_MASTER_WRITE, true);
226     i2c_master_write_byte(cmd, reg, true);
227     i2c_master_write_byte(cmd, data, true);
228     i2c_master_stop(cmd);
229     esp_err_t ret = i2c_master_cmd_begin(I2C_NUM_0, cmd,
          pdMS_TO_TICKS(1000));
230     i2c_cmd_link_delete(cmd);
231     return ret;
232 }
233
234 static esp_err_t mpu6050_read_bytes(uint8_t reg, uint8_t *data
       , size_t len) {
235     i2c_cmd_handle_t cmd = i2c_cmd_link_create();
236     i2c_master_start(cmd);
237     i2c_master_write_byte(cmd, (MPU6050_ADDR << 1) |
          I2C_MASTER_WRITE, true);
238     i2c_master_write_byte(cmd, reg, true);
239     i2c_master_start(cmd);
240     i2c_master_write_byte(cmd, (MPU6050_ADDR << 1) |
          I2C_MASTER_READ, true);
241     i2c_master_read(cmd, data, len, I2C_MASTER_LAST_NACK);
242     i2c_master_stop(cmd);
243     esp_err_t ret = i2c_master_cmd_begin(I2C_NUM_0, cmd,
          pdMS_TO_TICKS(1000));
244     i2c_cmd_link_delete(cmd);
245     return ret;
246 }
247
248 static void mpu6050_init(void) {
249     mpu6050_write_byte(0x6B, 0x00);
250     vTaskDelay(pdMS_TO_TICKS(100));
251     mpu6050_write_byte(0x1C, 0x10);
252     mpu6050_write_byte(0x1B, 0x08);
253     ESP_LOGI(TAG, "MPU6050 initialized");
254 }
255
256 static void mpu6050_read_accel(float *ax, float *ay, float *az
       ) {
257     uint8_t data[6];
258
259     *ax = 0.0;
260     *ay = 0.0;
261     *az = 0.0;
```

```
262
263     if (mpu6050_read_bytes(0x3B, data, 6) == ESP_OK) {
264         int16_t raw_ax = (data[0] << 8) | data[1];
265         int16_t raw_ay = (data[2] << 8) | data[3];
266         int16_t raw_az = (data[4] << 8) | data[5];
267
268         *ax = raw_ax / 4096.0;
269         *ay = raw_ay / 4096.0;
270         *az = raw_az / 4096.0;
271     }
272 }
273
274 // ISR Handler for Cancel Button
275 static void IRAM_ATTR cancel_button_isr_handler(void *arg) {
276     cancel_button_pressed = true;
277 }
278
279 // Alert Management Task - Handles buzzer, vibration, and SOS
    escalation
280 static void alert_management_task(void *pvParameters) {
281     alert_msg_t current_alert;
282     bool has_active_alert = false;
283
284     while (1) {
285         // Check if we have a new alert
286         if (xQueueReceive(alert_queue, &current_alert,
               pdMS_TO_TICKS(BUZZER_CHECK_MS))) {
287             has_active_alert = true;
288             current_alert_state = ALERT_STATE_WARNING;
289             alert_start_time = xTaskGetTickCount();
290             cancel_button_pressed = false;
291
292             ESP_LOGW(TAG, "        ALERT TRIGGERED: %s - Value
                   : %.2f", current_alert.alert_type,
                   current_alert.value);
293             ESP_LOGW(TAG, "       Buzzer activated! Press
                   button to cancel within 10 seconds.");
294         }
295
296         // Handle active alert state
297         if (has_active_alert && current_alert_state ==
               ALERT_STATE_WARNING) {
298             // Activate buzzer continuously
299             gpio_set_level(BUZZER_PIN, 1);
300
301             // Check if cancel button was pressed
302             if (cancel_button_pressed) {
303                 ESP_LOGI(TAG, "    Alert cancelled by user.
                       Situation under control.");
304                 gpio_set_level(BUZZER_PIN, 0);
305                 gpio_set_level(VIBRATION_PIN, 0);
306                 current_alert_state = ALERT_STATE_NORMAL;
307                 has_active_alert = false;
```

```
308                 cancel_button_pressed = false;
309                 continue;
310             }
311 
312             // Check if 10 seconds elapsed without cancel
313             uint32_t elapsed_time = (xTaskGetTickCount() -
                    alert_start_time) * portTICK_PERIOD_MS;
314             if (elapsed_time >= ALERT_TIMEOUT_MS) {
315                 // Escalate to EMERGENCY
316                 current_alert_state = ALERT_STATE_EMERGENCY;
317 
318                 ESP_LOGE(TAG, "       EMERGENCY! No response
                        from user for 10 seconds!");
319                 ESP_LOGE(TAG, "       Activating vibration
                        motor and sending SOS to command center!");
320 
321                 // Activate vibration motor
322                 gpio_set_level(VIBRATION_PIN, 1);
323 
324                 // Send SOS to command center
325                 send_sos_to_command_center(current_alert.
                        alert_type, current_alert.value);
326 
327             }
328         }
329 
330         // In EMERGENCY state, keep both buzzer and vibration
                active
331         if (current_alert_state == ALERT_STATE_EMERGENCY) {
332             gpio_set_level(BUZZER_PIN, 1);
333             gpio_set_level(VIBRATION_PIN, 1);
334 
335             // Optional: Allow manual reset even in emergency
336             if (cancel_button_pressed) {
337                 ESP_LOGI(TAG, "Emergency alert manually
                        cancelled");
338                 gpio_set_level(BUZZER_PIN, 0);
339                 gpio_set_level(VIBRATION_PIN, 0);
340                 current_alert_state = ALERT_STATE_NORMAL;
341                 has_active_alert = false;
342                 cancel_button_pressed = false;
343             }
344         }
345 
346         vTaskDelay(pdMS_TO_TICKS(BUZZER_CHECK_MS));
347     }
348 }
349 
350 // Gas Sensor Monitoring Task
351 static void gas_sensor_task(void *pvParameters) {
352     int adc_value;
353 
354     while (1) {
```

```
355         adc_oneshot_read(adc1_handle, MQ7_ADC_CHANNEL, &
               adc_value);
356         ESP_LOGI(TAG, "       Gas Sensor ADC: %d", adc_value);
357
358         if (adc_value > CO_THRESHOLD) {
359             alert_msg_t msg = {
360                 .value = (float)adc_value,
361                 .timestamp = xTaskGetTickCount(),
362                 .is_critical = true
363             };
364             strcpy(msg.alert_type, "DANGEROUS_CO_LEVEL");
365             xQueueSend(alert_queue, &msg, 0);
366             ESP_LOGW(TAG, "        Dangerous CO detected! ADC:
                    %d", adc_value);
367         }
368
369         vTaskDelay(pdMS_TO_TICKS(2000));
370     }
371 }
372
373 // Temperature and Humidity Monitoring Task
374 static void temp_humidity_task(void *pvParameters) {
375     float temperature, humidity;
376
377     while (1) {
378         if (dht22_read(&temperature, &humidity) == 0) {
379             ESP_LOGI(TAG, "           Temp: %.1f C , Humidity:
                    %.1f%%", temperature, humidity);
380
381             if (temperature > TEMP_THRESHOLD) {
382                 alert_msg_t msg = {
383                     .value = temperature,
384                     .timestamp = xTaskGetTickCount(),
385                     .is_critical = true
386                 };
387                 strcpy(msg.alert_type, "DANGEROUS_TEMPERATURE"
                       );
388                 xQueueSend(alert_queue, &msg, 0);
389                 ESP_LOGW(TAG, "        Dangerous temperature:
                       %.1f C", temperature);
390             }
391
392             if (humidity > HUMIDITY_THRESHOLD) {
393                 alert_msg_t msg = {
394                     .value = humidity,
395                     .timestamp = xTaskGetTickCount(),
396                     .is_critical = true
397                 };
398                 strcpy(msg.alert_type, "DANGEROUS_HUMIDITY");
399                 xQueueSend(alert_queue, &msg, 0);
400                 ESP_LOGW(TAG, "        Dangerous humidity: %.1
                       f%%", humidity);
401             }
```

```
402         } else {
403             ESP_LOGE(TAG, "DHT22 read failed");
404         }
405 
406         vTaskDelay(pdMS_TO_TICKS(3000));
407     }
408 }
409 
410 // IMU Monitoring Task (Fall Detection)
411 static void imu_task(void *pvParameters) {
412     float ax, ay, az;
413 
414     while (1) {
415         if (xSemaphoreTake(i2c_mutex, pdMS_TO_TICKS(100))) {
416             mpu6050_read_accel(&ax, &ay, &az);
417             xSemaphoreGive(i2c_mutex);
418 
419             float magnitude = sqrtf(ax*ax + ay*ay + az*az);
420             ESP_LOGI(TAG, "       Accel: %.2f, %.2f, %.2f (Mag
                : %.2f g)", ax, ay, az, magnitude);
421 
422             // Detect fall (sudden change in acceleration)
423             if (magnitude > FALL_THRESHOLD || magnitude < 0.5)
                 {
424                 alert_msg_t msg = {
425                     .value = magnitude,
426                     .timestamp = xTaskGetTickCount(),
427                     .is_critical = true
428                 };
429                 strcpy(msg.alert_type, "FALL_DETECTED");
430                 xQueueSend(alert_queue, &msg, 0);
431                 ESP_LOGW(TAG, "        Fall detected!
                    Magnitude: %.2f g", magnitude);
432             }
433         }
434 
435         vTaskDelay(pdMS_TO_TICKS(500));
436     }
437 }
438 
439 // Status Reporting Task - Sends periodic updates to command
    center
440 static void status_report_task(void *pvParameters) {
441     while (1) {
442         vTaskDelay(pdMS_TO_TICKS(30000));  // Report every 30
            seconds
443 
444         char url[256];
445         snprintf(url, sizeof(url), "http://%s:%s/api/status",
            COMMAND_CENTER_IP, COMMAND_CENTER_PORT);
446 
447         char post_data[256];
448         snprintf(post_data, sizeof(post_data),
```

```
449                 "{\"helmet_id\":\"%s\",\"status\":\"OK\",\"
                       timestamp\":%lu}",
450                 helmet_id, (unsigned long)(xTaskGetTickCount
                       () / 1000));
451
452         esp_http_client_config_t config = {
453             .url = url,
454             .method = HTTP_METHOD_POST,
455             .timeout_ms = 3000,
456         };
457
458         esp_http_client_handle_t client = esp_http_client_init
               (&config);
459         esp_http_client_set_header(client, "Content-Type", "
               application/json");
460         esp_http_client_set_post_field(client, post_data,
               strlen(post_data));
461
462         esp_err_t err = esp_http_client_perform(client);
463         if (err == ESP_OK) {
464             ESP_LOGI(TAG, "       Status update sent
                   successfully");
465         }
466
467         esp_http_client_cleanup(client);
468     }
469 }
470
471 // Main Application
472 void app_main(void) {
473     ESP_LOGI(TAG, "       Smart Helmet System Starting...");
474
475     // Generate unique helmet ID from MAC address
476     uint8_t mac[6];
477     esp_read_mac(mac, ESP_MAC_WIFI_STA);
478     snprintf(helmet_id, sizeof(helmet_id), "HELMET_%02X%02X%02
           X", mac[3], mac[4], mac[5]);
479     ESP_LOGI(TAG, "       Helmet ID: %s", helmet_id);
480
481     // Initialize NVS
482     esp_err_t ret = nvs_flash_init();
483     if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
           ESP_ERR_NVS_NEW_VERSION_FOUND) {
484         ESP_ERROR_CHECK(nvs_flash_erase());
485         ret = nvs_flash_init();
486     }
487     ESP_ERROR_CHECK(ret);
488
489     // Initialize WiFi
490     wifi_init_sta();
491
492     // Configure GPIO pins
493     gpio_config_t io_conf = {
```

```
494         .pin_bit_mask = (1ULL << BUZZER_PIN) | (1ULL <<
               VIBRATION_PIN),
495         .mode = GPIO_MODE_OUTPUT,
496         .pull_up_en = GPIO_PULLUP_DISABLE,
497         .pull_down_en = GPIO_PULLDOWN_DISABLE,
498         .intr_type = GPIO_INTR_DISABLE,
499     };
500     gpio_config(&io_conf);
501
502     // Ensure outputs start LOW
503     gpio_set_level(BUZZER_PIN, 0);
504     gpio_set_level(VIBRATION_PIN, 0);
505
506     // Configure Cancel button with interrupt
507     io_conf.pin_bit_mask = (1ULL << CANCEL_BUTTON_PIN);
508     io_conf.mode = GPIO_MODE_INPUT;
509     io_conf.pull_up_en = GPIO_PULLUP_ENABLE;
510     io_conf.intr_type = GPIO_INTR_NEGEDGE;
511     gpio_config(&io_conf);
512     gpio_install_isr_service(0);
513     gpio_isr_handler_add(CANCEL_BUTTON_PIN,
           cancel_button_isr_handler, NULL);
514
515     // Initialize ADC for MQ-7
516     adc_oneshot_unit_init_cfg_t init_config1 = {
517         .unit_id = ADC_UNIT_1,
518     };
519     ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_config1, &
           adc1_handle));
520
521     adc_oneshot_chan_cfg_t config = {
522         .bitwidth = ADC_BITWIDTH_DEFAULT,
523         .atten = ADC_ATTEN_DB_12,
524     };
525     ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle,
           MQ7_ADC_CHANNEL, &config));
526
527     // Initialize I2C and MPU6050
528     ESP_ERROR_CHECK(i2c_master_init());
529     mpu6050_init();
530
531     // Initialize DHT22
532     dht22_init();
533
534     // Create synchronization primitives
535     alert_queue = xQueueCreate(10, sizeof(alert_msg_t));
536     i2c_mutex = xSemaphoreCreateMutex();
537
538     // Create FreeRTOS tasks
539     xTaskCreate(gas_sensor_task, "gas_sensor", 4096, NULL,
           SENSOR_TASK_PRIORITY, NULL);
540     xTaskCreate(temp_humidity_task, "temp_humidity", 4096,
           NULL, SENSOR_TASK_PRIORITY, NULL);
```

```
541     xTaskCreate(imu_task, "imu_task", 4096, NULL,
           SENSOR_TASK_PRIORITY, NULL);
542     xTaskCreate(alert_management_task, "alert_mgmt", 4096,
           NULL, ALERT_TASK_PRIORITY, NULL);
543     xTaskCreate(status_report_task, "status_report", 4096,
           NULL, COMM_TASK_PRIORITY, NULL);
544
545     ESP_LOGI(TAG, "    All tasks created successfully");
546     ESP_LOGI(TAG, "       Monitoring: Temperature, Humidity,
           CO Level, Fall Detection");
547     ESP_LOGI(TAG, "        Alert System: Buzzer     10s
           timeout     Vibration + SOS");
548 }
```

Listing A.1: Complete code for Real-time Smart Helmet