Institute of Technology of Cambodia

Department of Electrical and Energy Engineering

Student Project

Topic : IOT Controlling and Monitoring System

Group: I5-GEE-EA-Gruop2

Lecturer by : SUM Rithea

Student : PUN Phorn (e20200408)

Academic Year5

2024-2025

Contents

STUDENT PROJECT

1. INTRODUCTION

1.1 Project Statement

In recent years, the Internet of Things (IoT) has emerged as a transformative technology with applications spanning smart homes, industrial automation, healthcare, and environmental monitoring. IoT systems enable devices to collect, process, and exchange data in real time, offering unprecedented levels of connectivity and control. One prominent application of IoT is in smart lighting systems, which enhance energy efficiency, convenience, and user experience by dynamically adjusting lighting based on environmental conditions or user preferences.

This project involves designing a custom ESP32 development board integrated with relays for smart home functions and monitoring data that capture by sensors. The board is programmed using the ESP-IDF framework, enabling it to communicate with Firebase for data storage, updates, and retrieval. A web application is developed to control the ESP32 and view real-time data.

1.2 Objective of the Study

The main objectives of this study are:

- To design and manufacture a custom PCB (Printed Circuit Board) for the ESP32 with relays integration.
- To write the program using ESP32 -IDF framework to read data that is captured from sensors, send it to firebase, and retrieve data from firebase to control relays.
- To create a user-friendly web application for monitoring and controlling devices connected to the ESP32.

1.3 Scope of Work

The project covers the following aspects:

- Writing a program in the ESP-IDF framework to interact with Firebase for storing sensors data and controlling relays.
- Developing a website to interface with Firebase for remote control and data visualization.
- Hosting Web app with Firebase.

2. DESIGN METHODOLOGY

This section outlines the methodology used to design and implement the IoT-based home monitoring and control System. The process was divided into three main phases: Hardware Design, Software Development, and Testing and Validation. Each phase is described in detail below.

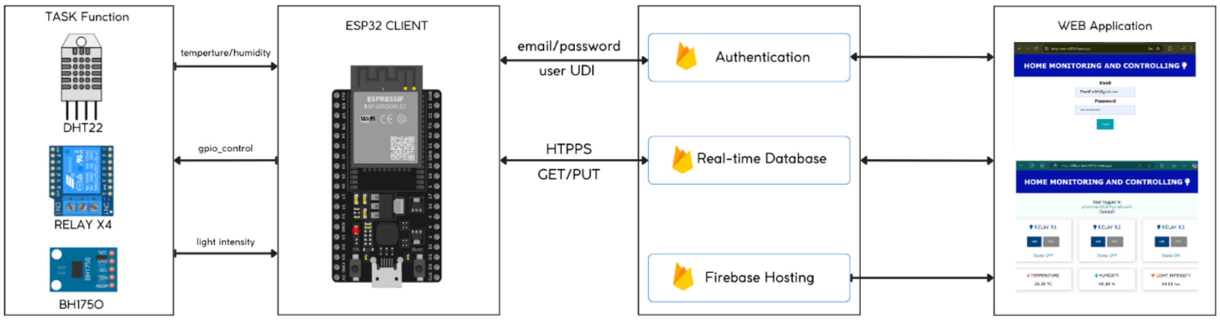


Figure 1: System Overview

The Figure 1 demonstrates the system overview. The proposed system includes an ESP32 and a sensor functioning as a client, while Firebase serves as the server, providing authentication, a real-time database for data storage, and Firebase Hosting for web application hosting.

2.1 Hardware Design

The hardware design focused on integrating all components into a compact and reliable system. Key considerations included component selection, PCB layout, and electrical calculations to ensure proper operation.

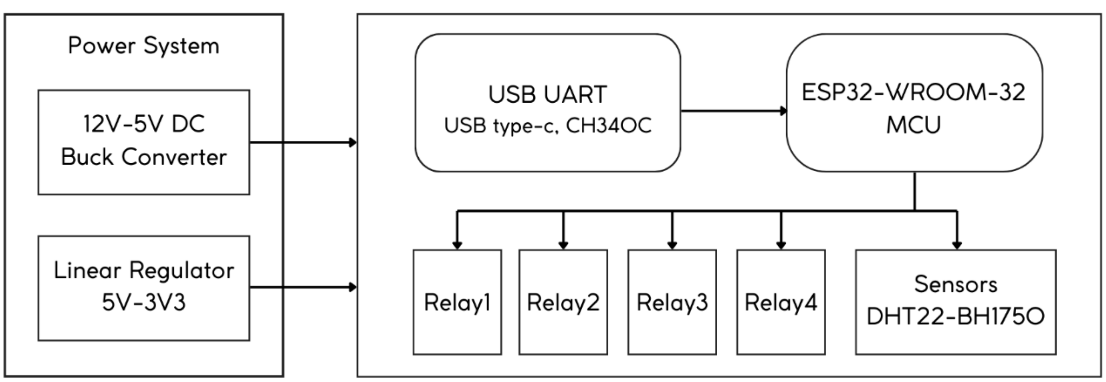


Figure 2: Block diagram of PCB design

Figure 2 illustrates the block diagram of the PCH design. The block includes the power system, which consists of a DC buck converter from 12V to 5V and a linear regulator from 5V to 3.3V.

This power is supplied to the entire board. The right block represents the diagram for the USB UART connected to the ESP32, and four relays and sensors are controlled by the ESP32.

2.1.1 Optocoupler Circuit

The PC817C optocoupler was used to provide electrical isolation between the ESP32 microcontroller and the relay driver circuit. This ensures that high-voltage spikes from the relay coil do not damage the sensitive GPIO pins of the ESP32.

- Circuit Design

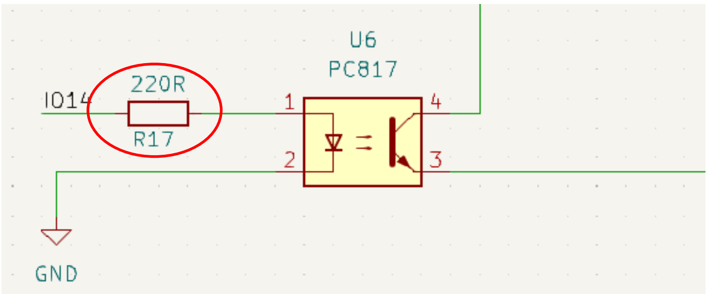


Figure 3: optocoupler circuit

The optocoupler consists of an internal LED and a phototransistor. The LED is driven by the ESP32's GPIO pin through a current-limiting resistor (R).

- LED Control Circuits

Figure 4(a) shows an example of controlling LED drive current by switching the supply voltage VIN on and off. Figure 4(b) indicates a load line in the (a) circuit. In this case, the resistor R is as follows.

$$R = \frac{V_{IN} - V_F}{I_F}$$

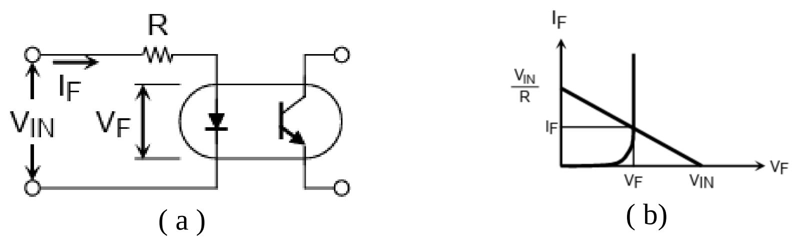


Figure 4: optocoupler DC drive

The forward current has chosen $I_F = 10mA$, $V_{IN} = 3.3V$ (ESP32 GPIO), and $V_{F(max)} = 1.2V$ (PC817 specification).

$$\Longrightarrow \mathrm{R} = \frac{3.3\mathrm{V} - 1.2\mathrm{V}}{10\mathrm{mA}} = 210\Omega \approx 220\Omega$$

Therefore, the resistor should be selected as $R = 220\Omega$.

- ❑ Simulation:

The optocoupler circuit was simulated in LTSPICE software to verify its operation:

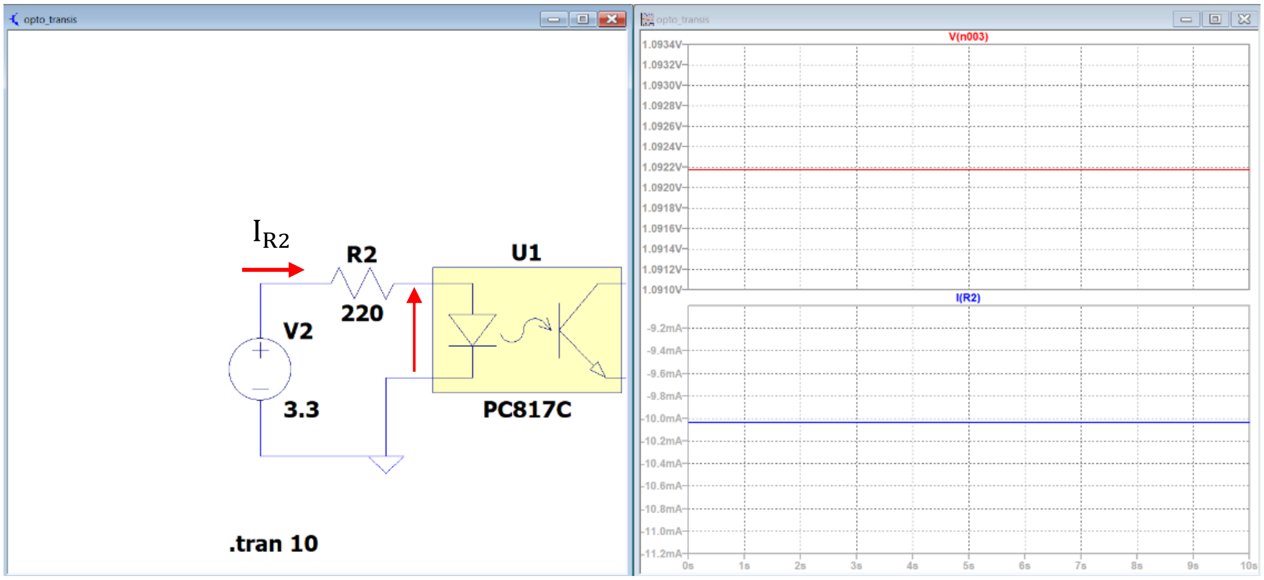


Figure 5: Simulation of PC817C circuit

Figure 5 displays the LTSPICE simulation for the PC817c with resistor R2, revealing that the forward voltage (VF) exceeds 1.0922V and the forward current (IF) is above 10mA. The results indicate a slight discrepancy between the simulation and calculations.

2.1.2 Transistor and Relay Circuit

The **2N3904 NPN transistor** was used to drive the relay coil, ensuring sufficient current to activate the relay while protecting the optocoupler.

- ❑ Circuit Design

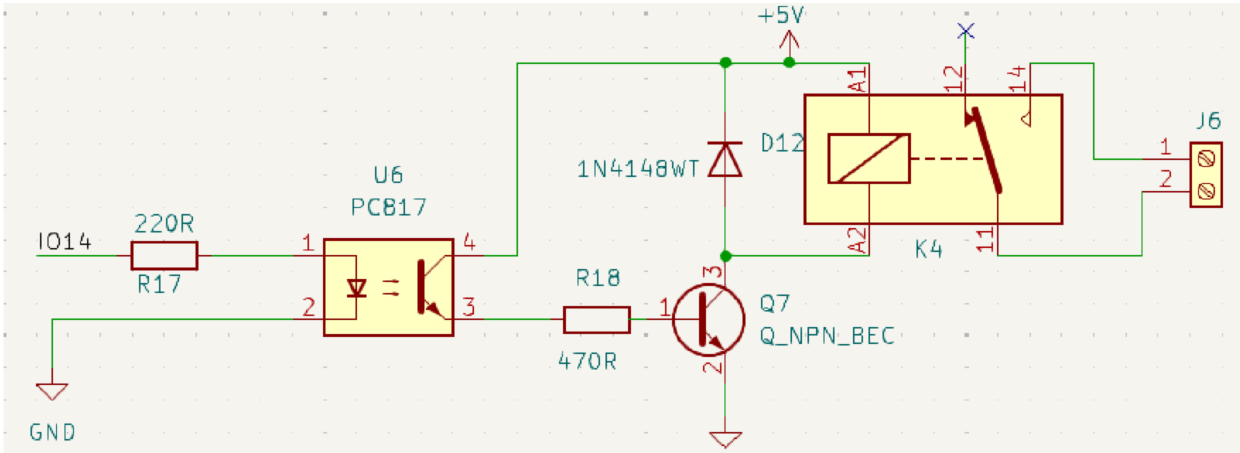


Figure 6: Circuit control relay completed

The transistor acts as a switch, controlled by the output of the optocoupler. A base resistor (**R17**) limits the current flowing into the transistor's base. The relay coil is connected to the collector, with the emitter grounded.

❑ Resistor Calculation

In the saturation region, the phototransistor's collector current (I_C) becomes largely independent of the LED forward current (I_F), provided that (I_F) is sufficient to drive the phototransistor into saturation.

Saturation Behavior

In **saturation mode,** the phototransistor operates in a region where: the collector-emitter voltage (V_{CE}) is very low, typically less than 0.2–0.3V. At this point, the collector current I_C is primarily determined by the external load connected to the phototransistor, not by the base current (I_F).

First define the I_C collector current:

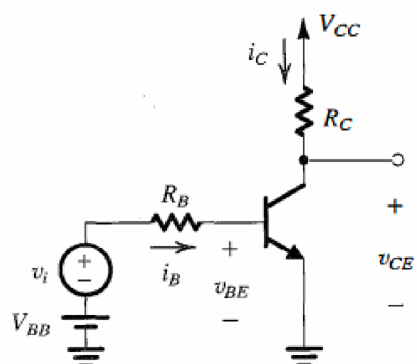
$$V_{CC} = V_C + V_{CE}$$

$$\Rightarrow I_C = \frac{V_{CC} - V_{CE}}{R_C}$$

$$\text{since, } V_{CE} \approx 0.2\text{V(saturation)}$$

$$R_C = 55\Omega \text{ (resistor in relay)}$$

$$V_{CC} = 5\text{V}$$



$$\Longrightarrow \mathrm{I_C} = \frac{5\mathrm{V} - 0.2\mathrm{V}}{55} = 87\mathrm{mA}$$

Following by 2N3904 datasheet $\beta = 10$ (saturation region)

$$\Longrightarrow \mathrm{I_B} = \frac{I_C}{\beta} = 8.7mA$$

$$R_B = \frac{V_i - V_{BE}}{I_B} = \frac{4.75 - 0.7}{8.7mA} = 465\Omega \approx 470\Omega$$

Therefore, the resistor should be selected as $R = 470\Omega$.

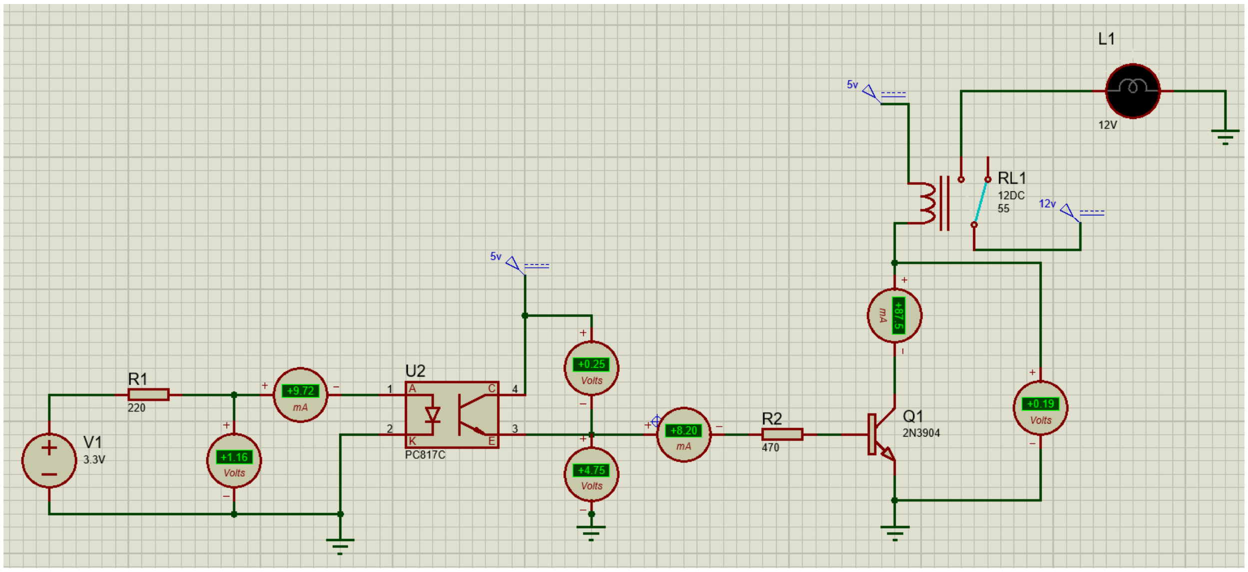


Figure 7: The rest of relay circuit simulation

After calculating the collector current, base current, and resistor values for the 2N3904 used with the relay, the Proteus simulator was employed to simulate the circuit. The simulation results indicate a slight discrepancy between theory and practice.

2.2 Software Development

The software development phase was a critical component of the project, involving programming the ESP32 microcontroller using the ESP-IDF platform**,** integrating it with Firebase Realtime Database for secure data storage and retrieval, and developing a web application for user interaction. The web application was hosted on Firebase to ensure seamless real-time communication between the hardware and the user interface.

2.2.1 FreeRTOS Task with HTTPS Communication Implementation

The ESP-IDF platform uses FreeRTOS to handle multitasking, ensuring efficient operation of sensor readings, data transmission, and relay control. Each task was designed to run independently while communicating securely with Firebase via HTTPS using the esp_http_client library.

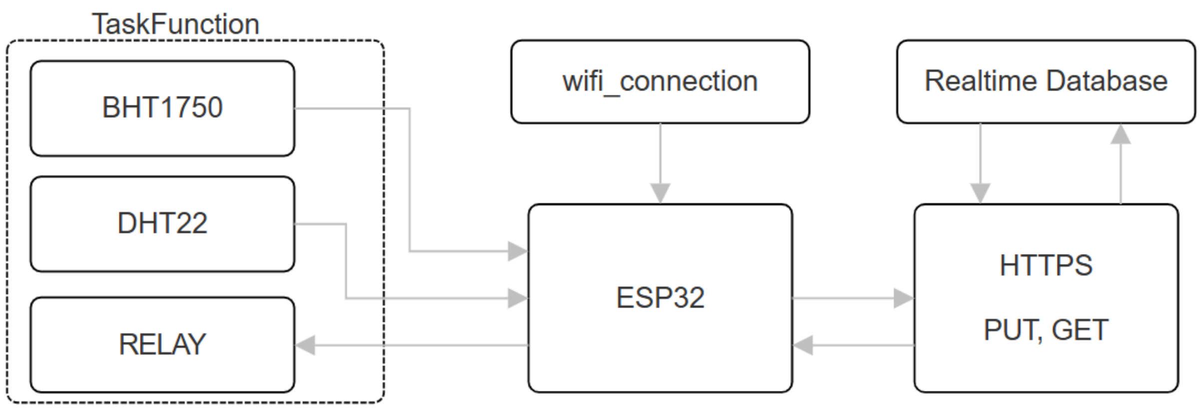


Figure 8: Structure of FreeRTOS and HTTPS Implement in ESP-IDF

- Task 1: DHT22 Sensor Reading and Data Transmission (dht_task)

Reads temperature and humidity data from the **DHT22 sensor** every 1 seconds and sends it to Real-time data through URL.

- Task 2: BH1750 Sensor Reading and Data Transmission (bh1750_task)

Read light intensity data from the BH1750 sensor via I2C every second, and send the data as a JSON payload to Firebase.

- Task 3: Button Control and Relay State Update (button_task)

Retrieve button states from Firebase via an HTTP GET request, parse the JSON response, and update the GPIO pins controlling the relays accordingly.

- Secure HTTPS Communication Between ESP32 and Firebase Using TLS Encryption.

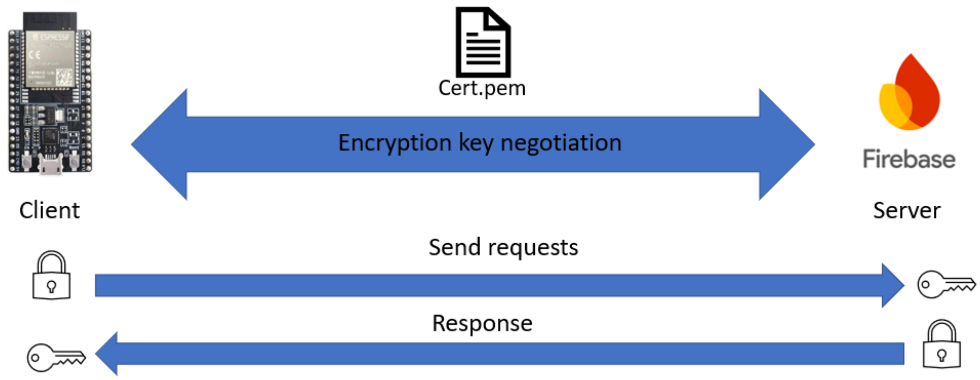


Figure 9: ESP32 with Firebase data encryption

This diagram shows secure communication between an ESP32 client and Firebase server using HTTPS and TLS encryption. The TLS handshake establishes a secure connection with encryption key negotiation, verified by a certificate (Cert.pem). The client encrypts requests before sending them, and the server decrypts, processes, and encrypts responses before returning them. This ensures data integrity, confidentiality, and authentication for secure ESP32-Firebase communication.

2.2.2 Firebase Integration

Firebase serves as the backend service for real-time data storage, retrieval, and synchronization.

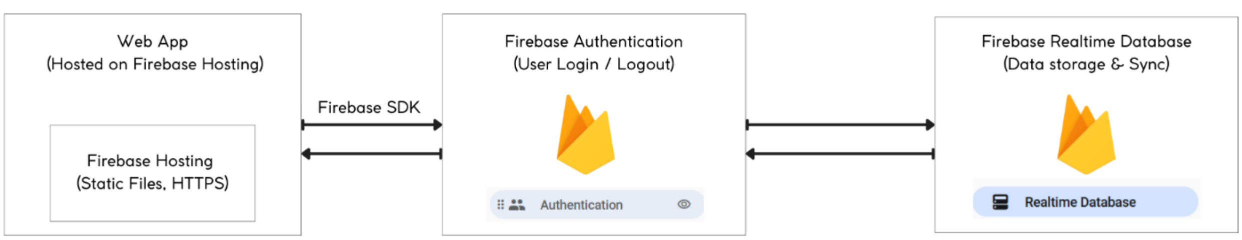


Figure 10: Block diagram of Firebase integrate with Web app

❑ Firebase Authentication

Firebase Authentication ensures secure access to the system by managing user identities and permissions. Users are required to authenticate themselves (e.g., via email/password or Google Sign-In) before accessing the system. Once authenticated, Firebase generates a unique token for the user, which is used to authorize subsequent requests to the Realtime Database. Firebase Authentication rules ensure that only authenticated users can read or write data to specific nodes in the database.

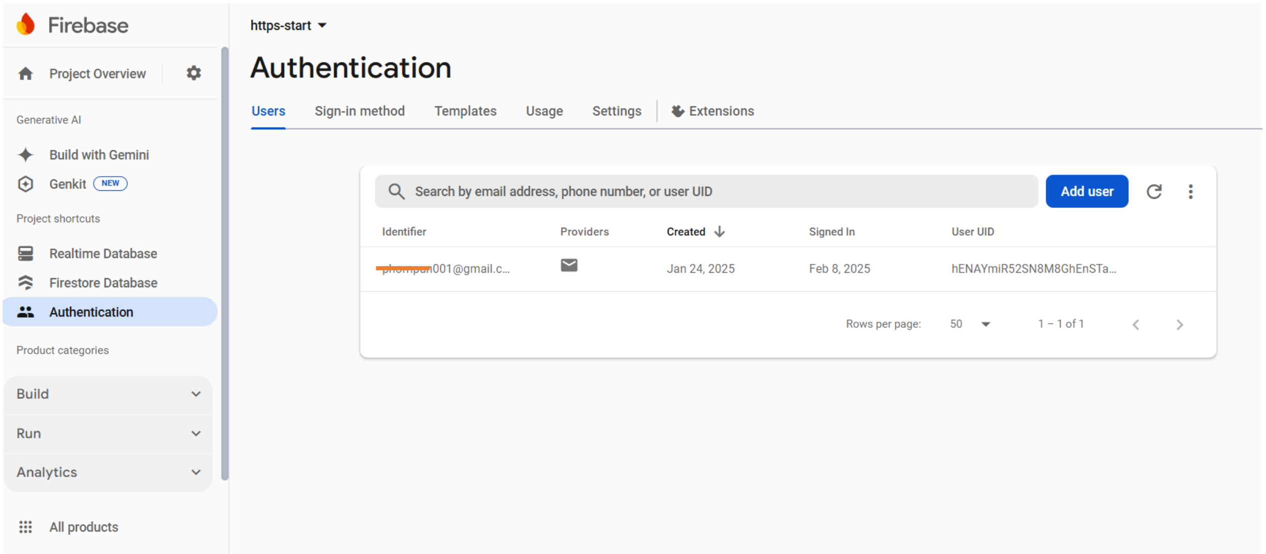


Figure 11: Set up Authentication

❑ Firebase Realtime Database

The Realtime Database serves as the central hub for storing and synchronizing data between the ESP32 and the web application. Stores sensor data (temperature, humidity, light intensity) collected by the ESP32 and control commands Relay states sent by users via the web application.

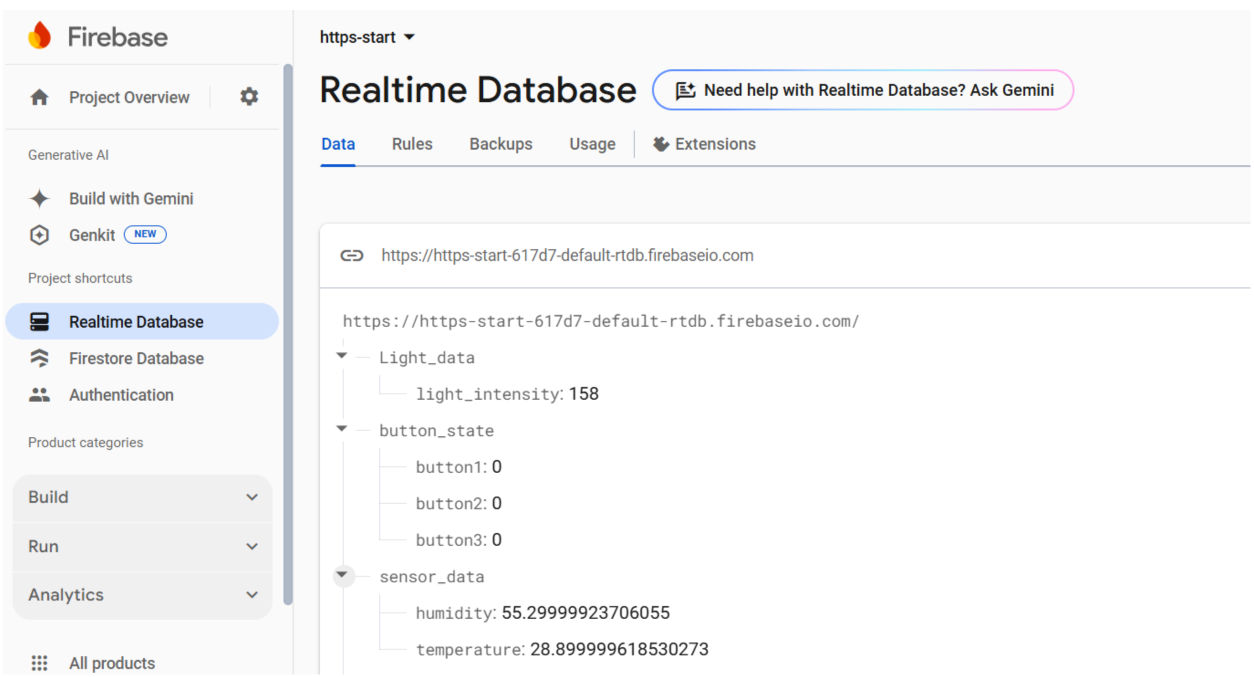
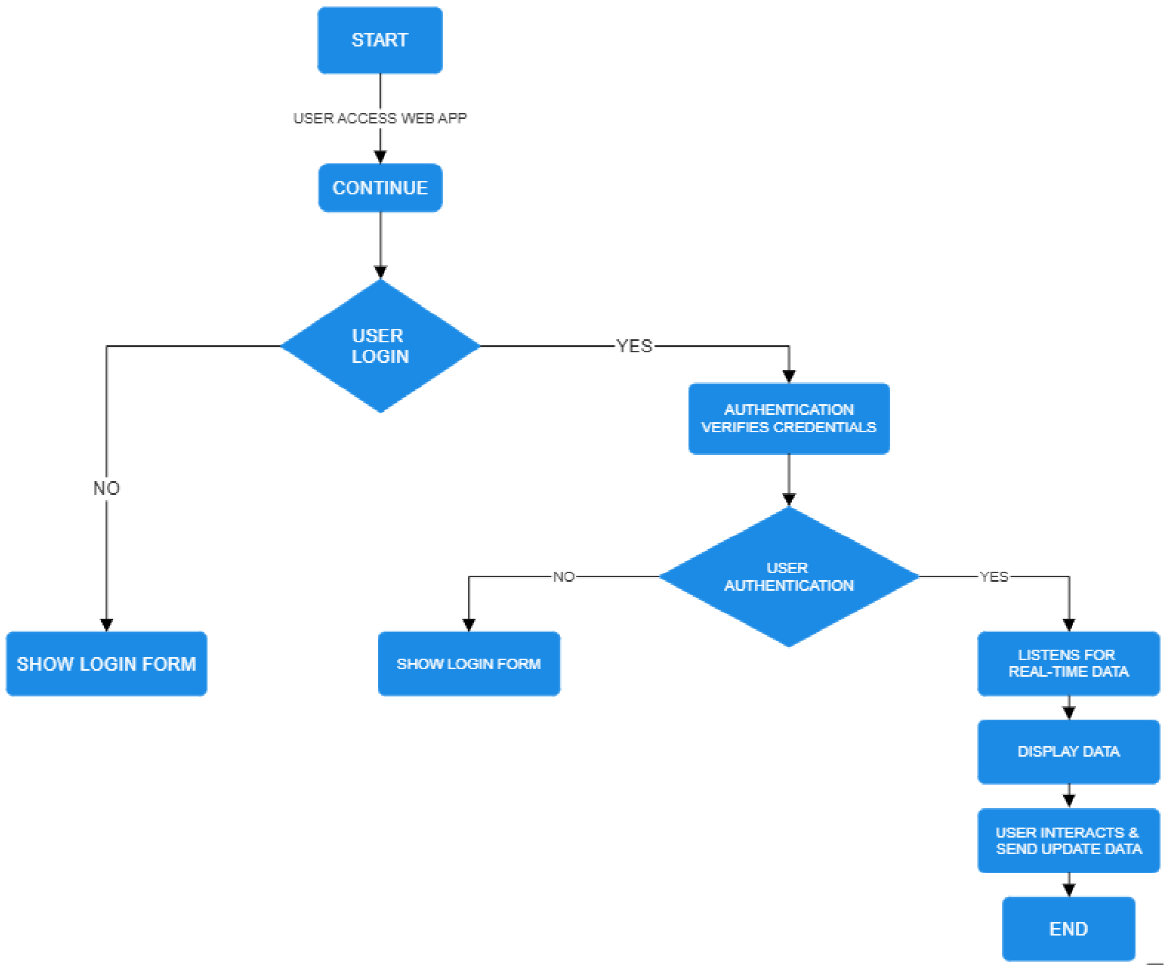


Figure 12: set up Realtime Database

2.2.3 Web Application Development

A web application was developed to provide users with a user-friendly interface for monitoring sensor data and controlling the LEDs remotely. The web app was built using HTML, CSS, and JavaScript and hosted on Firebase for real-time updates and cross-platform accessibility.



The flowchart outlines the web app's process for real-time monitoring. It begins with user login via email and password, verified by Firebase Authentication. If valid, users access the main interface. The app retrieves and displays real-time sensor data from Firebase. Users can interact with the app to update relay states or perform actions, which are stored and synchronized by Firebase.

2.2.4 Firebase Hosting

Firebase Hosting is a fully managed service provided by Firebase that allows developers to deploy and serve web applications securely and efficiently. It is designed to host static assets such as HTML, CSS, JavaScript, images, and other files required for a web application. Firebase Hosting ensures that your web app is delivered over HTTPS with minimal latency, leveraging Google's global Content Delivery Network (CDN) for fast and reliable performance.

- ❑ Deployment of Web App Files:

The deployment process involves preparing web application files, configuring Firebase Hosting, and deploying the files to Firebase's servers.

- ❑ Log In to Firebase

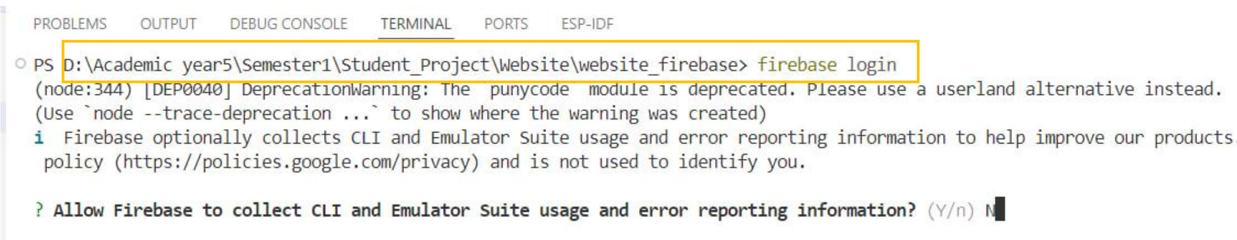


Figure 13: login firebase

This command opens a browser window where log in to Firebase account. Once logged in, the CLI gains access to Firebase projects.

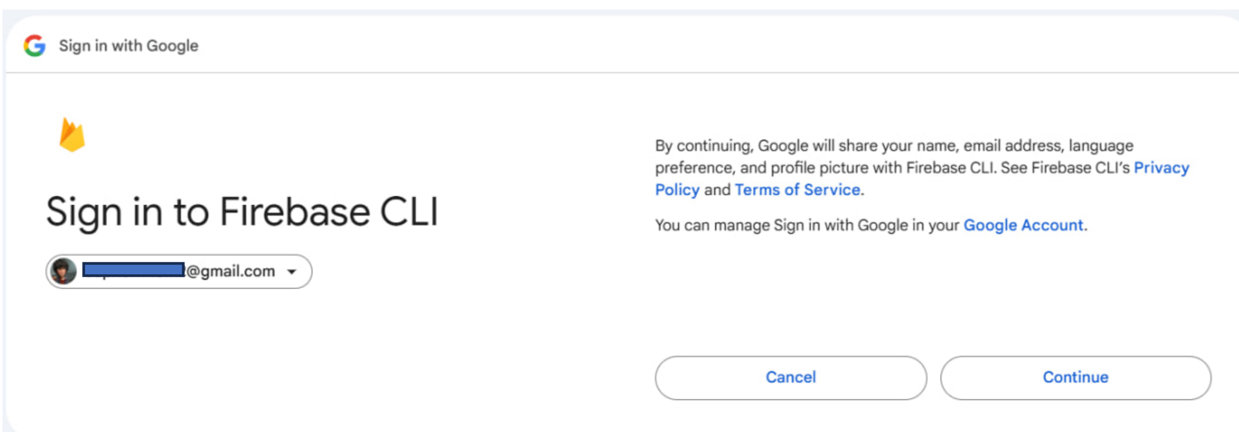


Figure 14: Sign in firebase with google

After this, a new window will open up in web browser, allowing us the opportunity to log into our Firebase account where we can access all our projects.

- Initialize Firebase in Project

```
D:\Academic year5\Semester1\Student_Project\Website\website_firebase

? Are you ready to proceed? Yes
? Which Firebase features do you want to set up for this directory? Press Space to select features, then Enter to confirm your choices.
all, <i> to invert selection, and <enter> to proceed)
 (*) Hosting: Configure files for Firebase Hosting and (optionally) set up GitHub Action deploys
 ( ) Storage: Configure a security rules file for Cloud Storage
 ( ) Emulators: Set up local emulators for Firebase products
>( ) Remote Config: Configure a template file for Remote Config
 ( ) Extensions: Set up an empty Extensions manifest
 (*) Realtime Database: Configure a security rules file for Realtime Database and (optionally) provision default instance
 ( ) Data Connect: Set up a Firebase Data Connect service
(Move up and down to reveal more choices)
```

In the initialize Firebase in Project was selected:

- Realtime Database: Configure security rules file for Realtime Database and (optionally) provision default instance.
- Hosting: Configure files for Firebase Hosting and (optionally) set up GitHub Action deploys

- Deploy to Firebase Hosting

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

PS D:\Academic year5\Semester1\Student_Project\Website\web_app_firebase> firebase deploy
(node:7692) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was created)

=== Deploying to 'https-start-617d7'...

i  deploying database, hosting
i  database: checking rules syntax...
+  database: rules syntax for database https-start-617d7-default-rtdb is valid
i  hosting[https-start-617d7]: beginning deploy...
i  hosting[https-start-617d7]: found 6 files in public
+  hosting[https-start-617d7]: file upload complete
i  database: releasing rules...
+  database: rules for database https-start-617d7-default-rtdb released successfully
i  hosting[https-start-617d7]: finalizing version...
+  hosting[https-start-617d7]: version finalized
i  hosting[https-start-617d7]: releasing new version...
+  hosting[https-start-617d7]: release complete                URL

+  Deploy complete!

Project Console: https://console.firebase.google.com/project/https-start-617d7/overview
Hosting URL: https://https-start-617d7.web.app
PS D:\Academic year5\Semester1\Student_Project\Website\web_app_firebase>
```

Figure 15: Deploy to Firebase Hosting

After running the Firebase deploy command, Firebase uploads the files from the public directory to its servers and provides a public URL for the app upon successful deployment.

3. RESULT AND DISCUSSION

3.1 PCB Result and Testing

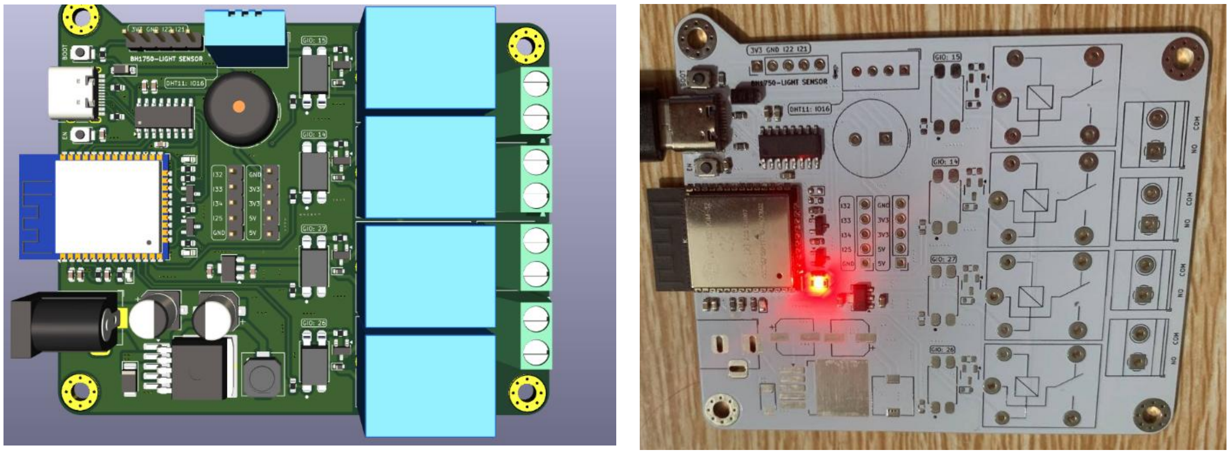


Figure 16: The PCB 3D and outcome

The Figure 16, the left figure shows a 3D of the PCB, including components such as an ESP32 module, relays, capacitors, and connectors**.** The right image is the actual manufactured PCB**,** populated with components, showing a close resemblance to the 3D model.

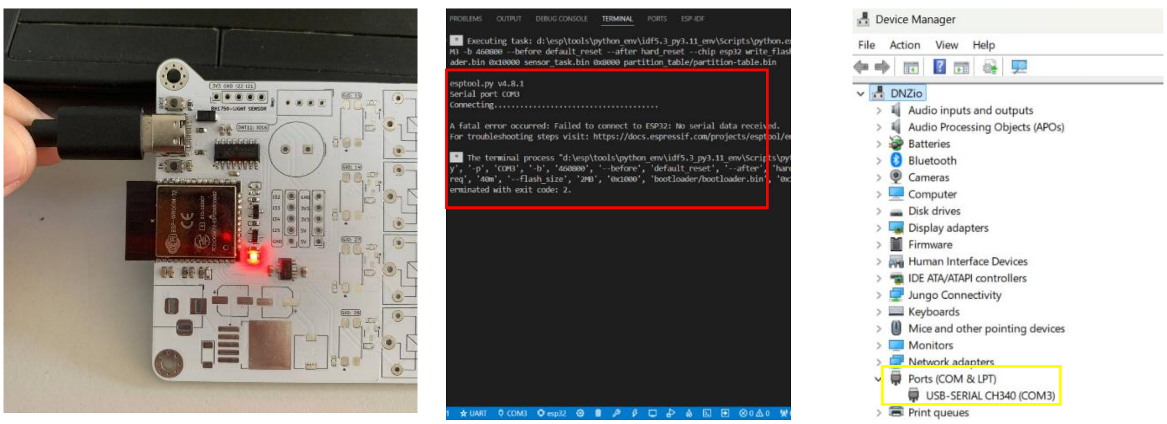


Figure 17: ESP32 Flashing

The first image shows the ESP32 connected via USB, with a red LED indicating power. The second image displays terminal output indicating a fault during the firmware flashing process with ESP-IDF. The final image is a screenshot of Device Manager, confirming the computer recognizes the USB-SERIAL CH340 (COM3).

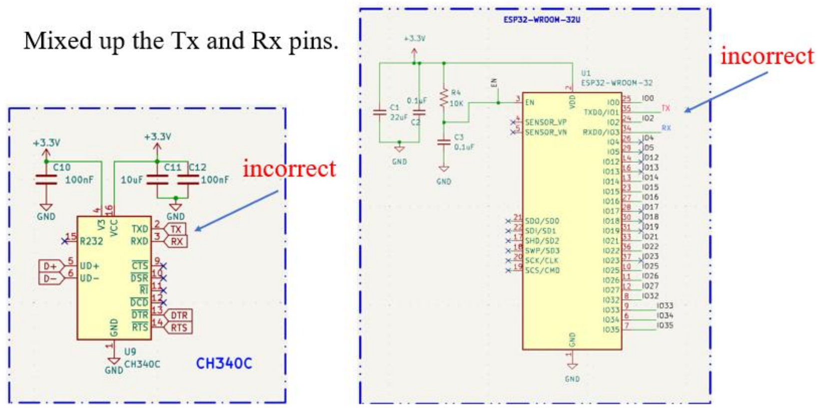


Figure 18: Fault between Tx and Rx of the ESP32 and CH340

This figure highlights a wiring error in the schematic design, where the Tx (transmit) and Rx (receive) pins of the ESP32 and CH340 USB-to-serial converter were swapped.

3.2 Firebase and ESP32 client testing for a web app.

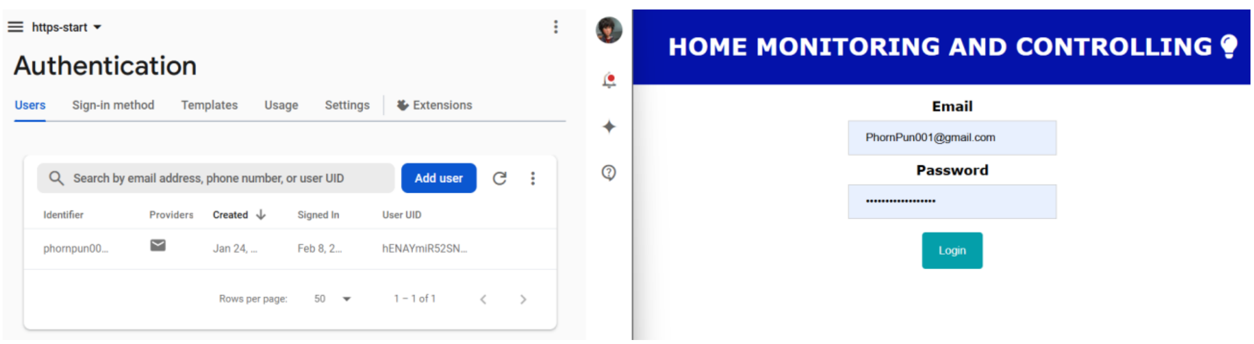


Figure 19: Authentication verifies

The figure shows the web app's login interface, where users enter their email and password to authenticate and gain access.

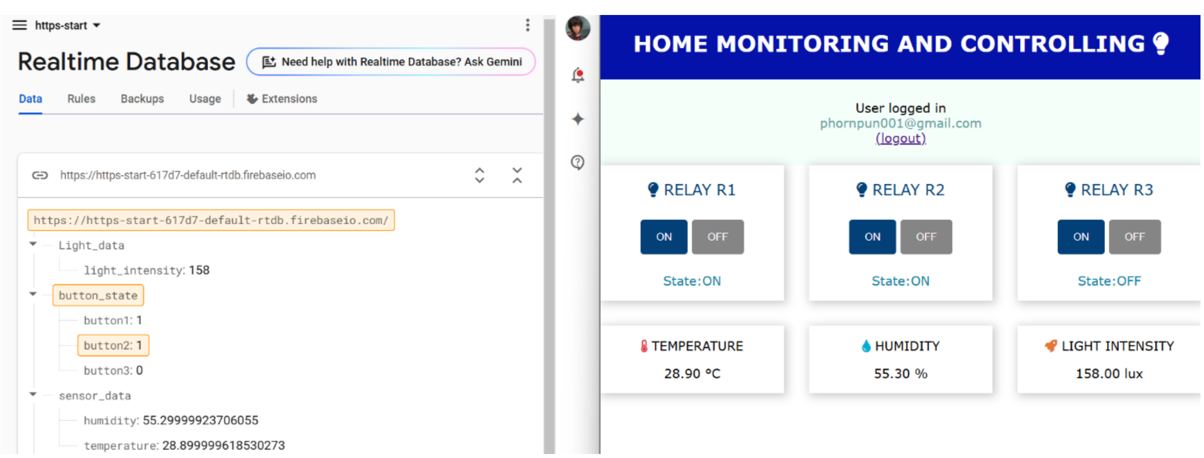


Figure 20: The Real-time Database interact with Web app

Figure 20 displays the interaction between the Firebase Realtime Database and the web application. On the left, the Firebase interface shows real-time data storage, including sensor readings (temperature, humidity, and light intensity) and button states. On the right, the web app interface provides a dashboard for controlling relays and viewing sensor data, with a user logged in.

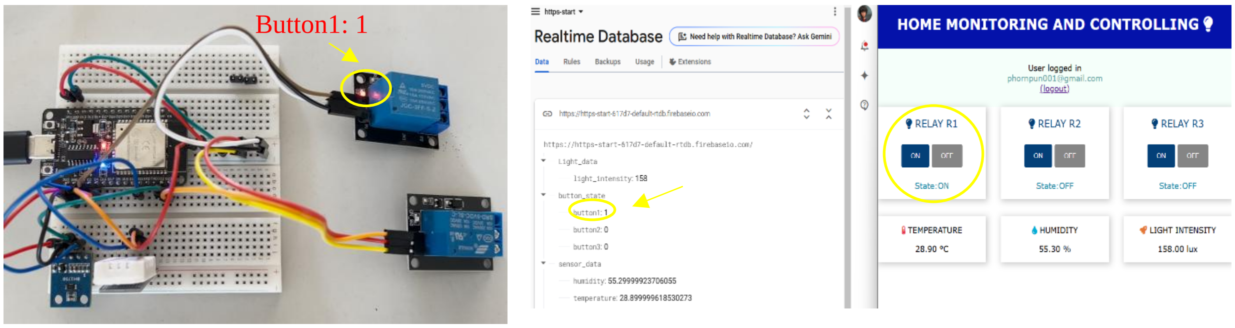


Figure 21: Testing ESP32 with Web app

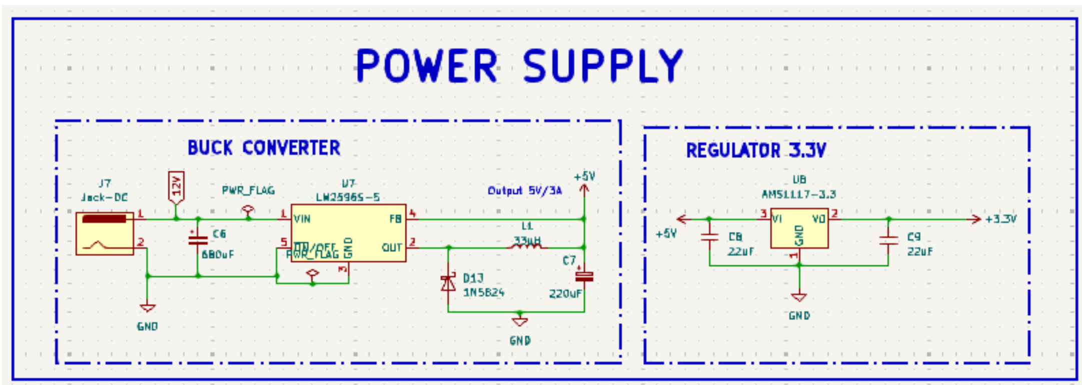
Figure 21 demonstrates the testing phase of the ESP32 with the web app. The left side shows a hardware setup on a breadboard, including an ESP32, relays, and wiring. The middle section shows Firebase reflecting the change in button state, while the right side highlights the corresponding update in the web interface, where the relay status is toggled.

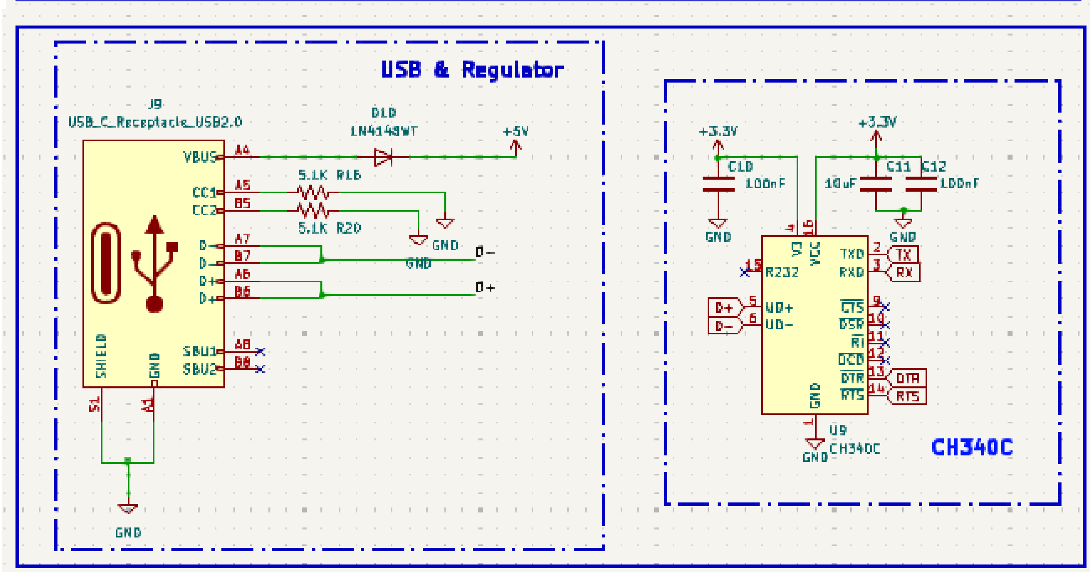
4. CONCLUSION

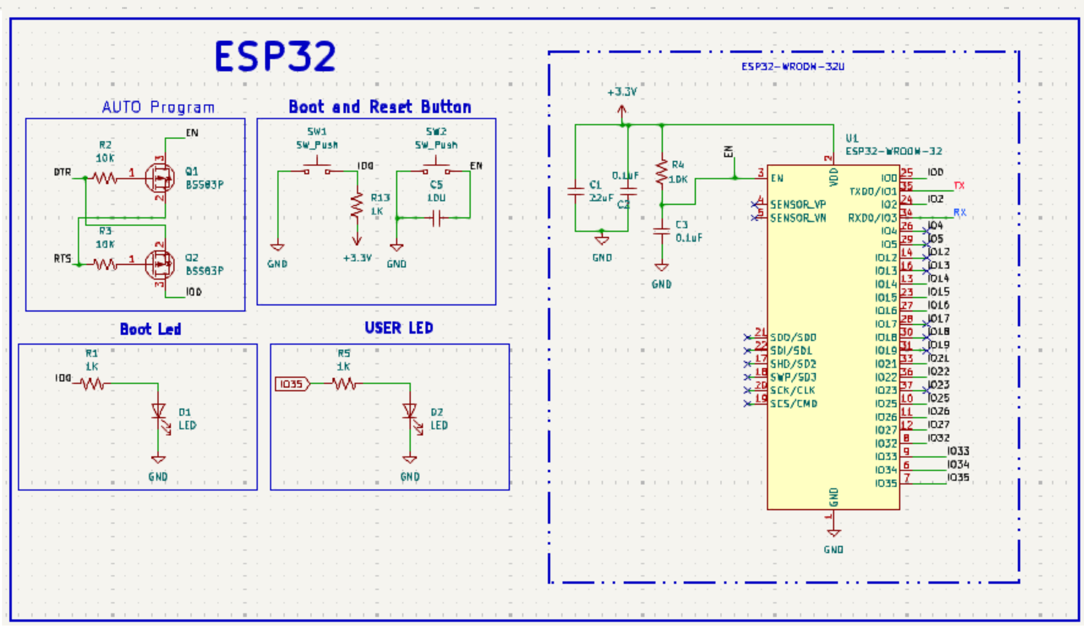
This project aims to develop a comprehensive IoT solution featuring an ESP32-based board equipped with sensors and a relay for real-time monitoring and control applications. The ESP32 is programmed using the ESP-IDF framework, enabling efficient hardware interaction and secure communication via the HTTPS protocol for data exchange with Firebase. Additionally, a web application hosted on Firebase Hosting provides a user-friendly interface for monitoring sensor data and controlling the relay remotely.

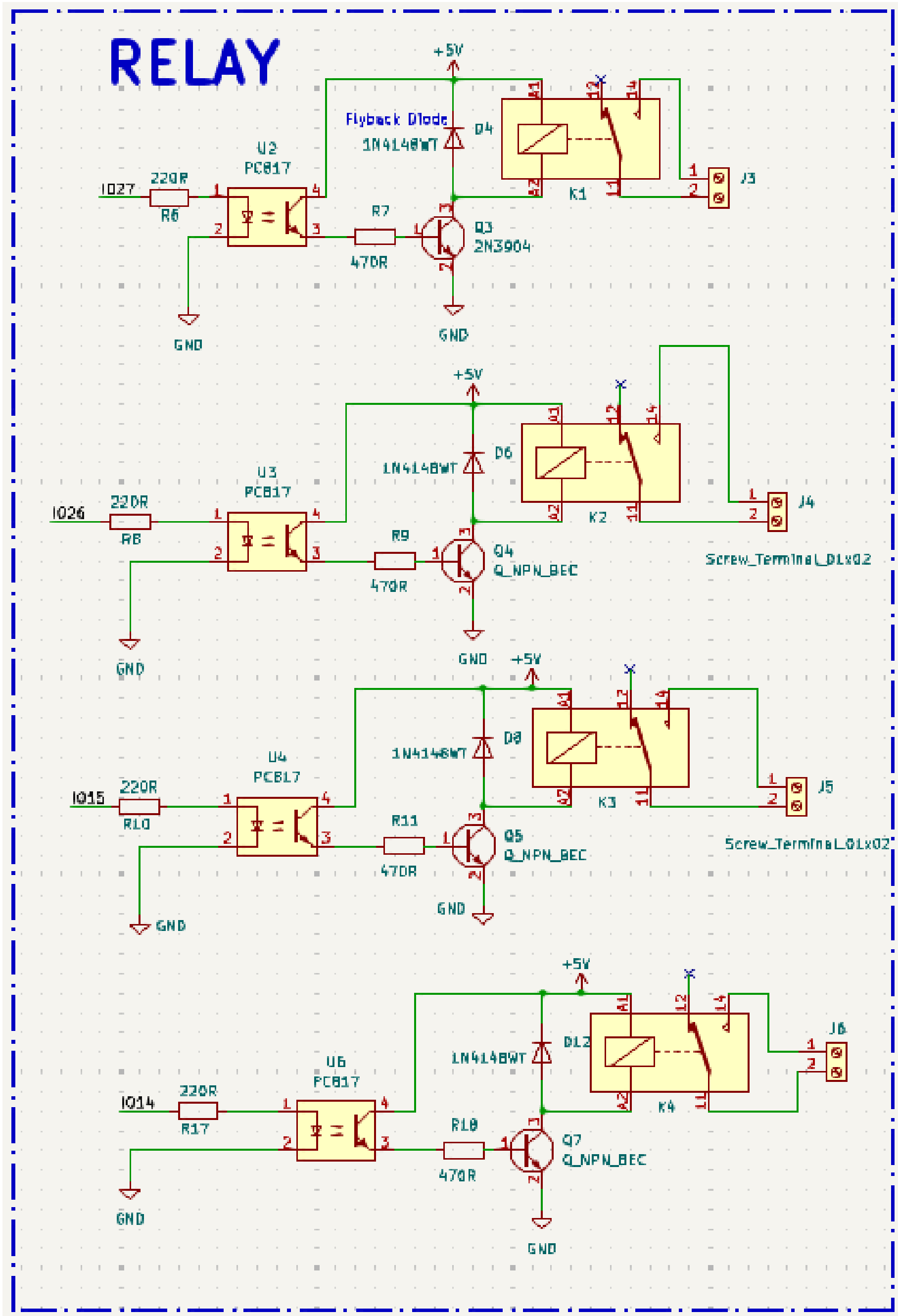
Appendix:

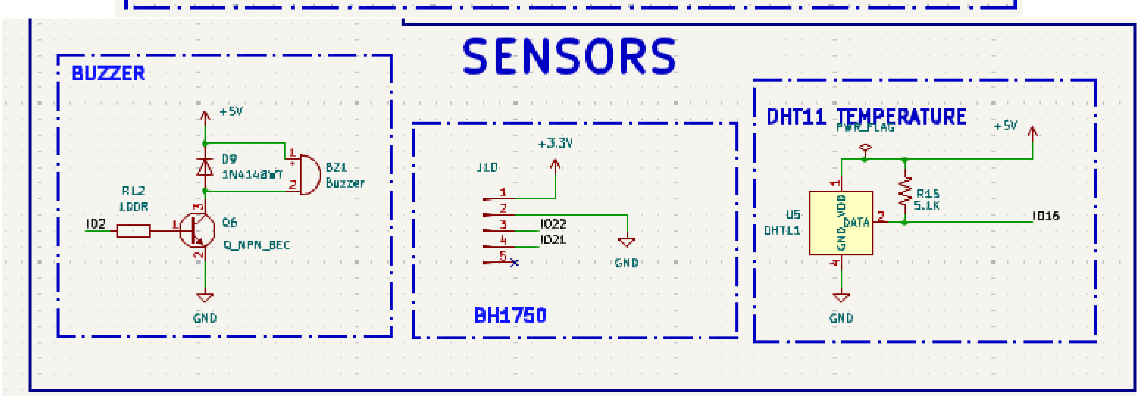
❑ PCB








RELAY
+5V
Flyback Diode
1N4148WT
D4
U2
PC817
IO27
220R
R6
R7
470R
Q3
2N3904
K1
J3
GND
+5V
1N4148WT
D6
U3
PC817
IO26
220R
R8
R9
470R
Q4
Q_NPN_BEC
K2
J4
Screw_Terminal_01x02
GND
+5V
1N4148WT
D8
U4
PC817
IO15
220R
R10
R11
470R
Q5
Q_NPN_BEC
K3
J5
Screw_Terminal_01x02
GND
+5V
1N4148WT
D12
U6
PC817
IO14
220R
R17
R18
470R
Q7
Q_NPN_BEC
K4
J6
GND


SENSORS
BUZZER
+5V
D9
1N4148WT
BZ1
Buzzer
R12
LDDR
IO2
Q6
Q_NPN_BEC
GND
+3.3V
J10
IO22
IO21
GND
BH1750
DHT11 TEMPERATURE
PWR_FLAG
+5V
U5
DHT11
VDD
DATA
GND
R15
5.1K
IO16
GND

❑ Program:

```
#include <stdio.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "esp_log.h"
#include "nvs_flash.h"
//#include "esp_netif.h"
#include "esp_http_client.h"
#include "cJSON.h"
#include "dht.h"
#include "bh1750.h"
#include "protocol_examples_common.h"

// --- Constants and Definitions ---
#define I2C_SDA GPIO_NUM_21
#define I2C_SCK GPIO_NUM_22
#define SENSOR_TYPE DHT_TYPE_AM2301
#define CONFIG_DATA_GPIO GPIO_NUM_4
//button pins
#define BUTTON1_GPIO GPIO_NUM_2
#define BUTTON2_GPIO GPIO_NUM_19
#define BUTTON3_GPIO GPIO_NUM_23
// Firebase URLs
#define FIREBASE_DHT_URL "https://https-start-617d7-default-
rtdb.firebaseio.com/sensor_data.json"
#define FIREBASE_LIGHT_URL "https://https-start-617d7-default-
rtdb.firebaseio.com/Light_data.json"
#define FIREBASE_BUTTON_URL "https://https-start-617d7-default-
rtdb.firebaseio.com/button_state.json"
#define MAX_BUFFER_SIZE 256
// External Certificates
extern const uint8_t certificate_pem_start[]
asm("_binary_certificate_pem_start");
extern const uint8_t certificate_pem_end[]
asm("_binary_certificate_pem_end");

// Event Groups and Tags
static const char *TAG_WIFI = "WiFi";
static const char *TAG = "HTTP_CLIENT";
static const char *TAG_DHT = "DHT_SENSOR";
static const char *TAG_BH1750 = "BH1750_SENSOR";
static const char *TAG_BUTTON = "BUTTON";

// --- Function Prototypes ---
void dht_task(void *params);
void bh1750_task(void *params);
void button_task(void *params);
void firebase_task(void *pvParameters);
```

```

/* Callback or event handler */
esp_err_t http_event_handler(esp_http_client_event_t* evt)
{
    switch (evt->event_id)
    {
    case HTTP_EVENT_ERROR:
        ESP_LOGI(TAG, "HTTP_EVENT_ERROR");
        break;

    case HTTP_EVENT_ON_CONNECTED:
        ESP_LOGI(TAG, "HTTP_EVENT_ON_CONNECTED");
        break;

    case HTTP_EVENT_ON_DATA:
        ESP_LOGI(TAG, "HTTP_EVENT_ON_DATA, len=%d, data=%.*s", evt-
>data_len, evt->data_len, (char*)evt->data);

        if (evt->data_len > MAX_BUFFER_SIZE)
            return ESP_FAIL;

        if (evt->user_data)
        {
            memcpy(evt->user_data, evt->data, evt->data_len);
        }
        break;

    case HTTP_EVENT_DISCONNECTED:
        ESP_LOGD(TAG, "HTTP_EVENT_DISCONNECTED");
        break;

    default:
        break;
    }
    return ESP_OK;
}
/* Functions GET method */
esp_err_t http_client_get_req(char* data, const char* url)
{
    esp_err_t ret_code = ESP_FAIL;
    // HTTP client configuration
    esp_http_client_config_t config = {
        .event_handler = http_event_handler,
        .method = HTTP_METHOD_GET,
        .port = 80,
        .url = url,
        .user_data = data,
        .cert_pem = (const char *)certificate_pem_start,
    };
    esp_http_client_handle_t client = esp_http_client_init(&config);
    esp_err_t err = esp_http_client_perform(client);
    if (err == ESP_OK)
```

```
    {
        int status = esp_http_client_get_status_code(client);

        if (status == 200)
        {
            ESP_LOGI(TAG, "HTTP GET status: %d", status);
            ret_code = ESP_OK;
        }
        else
        {
            ESP_LOGE(TAG, "HTTP GET status: %d", status);
        }
    }
    else
    {
        ESP_LOGE(TAG, "Failed to send GET request");
    }
    esp_http_client_cleanup(client);

    return ret_code;
}

/* HTTP POST function */
esp_err_t http_client_post_req(const char* data_post, const char*
post_url)
{
    esp_err_t ret_code = ESP_FAIL;
    // HTTP client configuration
    esp_http_client_config_t config = {
        .event_handler = http_event_handler,
        .method = HTTP_METHOD_PUT,
        .port = 80,
        .url = post_url,
        .cert_pem = (const char *)certificate_pem_start, // Use your
certificate

    };
    esp_http_client_handle_t client = esp_http_client_init(&config);
    esp_http_client_set_header(client, "Content-Type",
"application/json");
    esp_http_client_set_post_field(client, data_post, strlen(data_post));

    esp_err_t err = esp_http_client_perform(client);
    if (err == ESP_OK)
    {
        int status = esp_http_client_get_status_code(client);
        if (status == 200)
        {
            ESP_LOGI(TAG, "HTTP POST status: %d", status);
            ret_code = ESP_OK;
        }
        else
```

```
        {
            ESP_LOGE(TAG, "HTTP POST status: %d", status);
        }
    }
    else
    {
        ESP_LOGE(TAG, "Failed to send POST request");
    }
    esp_http_client_cleanup(client);
    return ret_code;
}

void button_task(void* arg) {
    char data[MAX_BUFFER_SIZE] = {0};
    gpio_config_t io_conf = {
        .pin_bit_mask = (1ULL << BUTTON1_GPIO) | (1ULL << BUTTON2_GPIO) |
(1ULL << BUTTON3_GPIO),
        .mode = GPIO_MODE_OUTPUT,
        .pull_up_en = GPIO_PULLUP_DISABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_DISABLE
    };
    gpio_config(&io_conf);
    gpio_set_level(BUTTON1_GPIO, 0);
    gpio_set_level(BUTTON2_GPIO, 0);
    gpio_set_level(BUTTON3_GPIO, 0);
while (1)
    {
        // Send HTTP GET request to Firebase to retrieve button states
        if (http_client_get_req(data, FIREBASE_BUTTON_URL) == ESP_OK)
        {
            ESP_LOGI(TAG_BUTTON, "Received button states: %s", data);

            // Parse the JSON response
            cJSON* json = cJSON_Parse(data);
            if (json == NULL)
            {
                ESP_LOGE(TAG_BUTTON, "Failed to parse JSON response.");
            }
            else
            {
                // Extract button states from the JSON
                cJSON* button1 = cJSON_GetObjectItem(json, "button1");
                cJSON* button2 = cJSON_GetObjectItem(json, "button2");
                cJSON* button3 = cJSON_GetObjectItem(json, "button3");

                if (button1 && button2 && button3)
                {
                    int button1_state = button1->valueint;
                    int button2_state = button2->valueint;
                    int button3_state = button3->valueint;

```

```
                    ESP_LOGI(TAG_BUTTON, "Button1: %d, Button2: %d, Button3: %d", button1_state, button2_state, button3_state);

                    // Control LEDs based on Firebase button states
                    gpio_set_level(BUTTON1_GPIO, button1_state);
                    gpio_set_level(BUTTON2_GPIO, button2_state);
                    gpio_set_level(BUTTON3_GPIO, button3_state);
                }
                else
                {
                    ESP_LOGE(TAG_BUTTON, "Failed to extract button states from JSON.");
                }

                cJSON_Delete(json); // Free the JSON object
            }
        }
        else
        {
            ESP_LOGE(TAG_BUTTON, "Failed to retrieve button states from Firebase.");
        }

        vTaskDelay(pdMS_TO_TICKS(1000));
    }
    vTaskDelete(NULL);
}
// --- DHT Sensor Task ---
void dht_task(void* arg)
{
    float temp, humidity;

    while (1)
    {
        if (dht_read_float_data(SENSOR_TYPE, CONFIG_DATA_GPIO, &humidity, &temp) == ESP_OK)
        {
            ESP_LOGI(TAG_DHT, "Humidity: %.1f%%, Temp: %.1fC", humidity, temp);

            // Create JSON payload
            cJSON* json = cJSON_CreateObject();
            cJSON_AddNumberToObject(json, "temperature", temp);
            cJSON_AddNumberToObject(json, "humidity", humidity);
            char* data = cJSON_Print(json);

            // Send data to Firebase
            if (http_client_post_req(data, FIREBASE_DHT_URL) == ESP_OK)
            {
                ESP_LOGI(TAG, "DHT data uploaded successfully.");
            }
            else
```

```
            {
                ESP_LOGE(TAG, "Failed to upload DHT data.");
            }

            cJSON_Delete(json);
            free(data);
        }
        else
        {
            ESP_LOGE(TAG_DHT, "Failed to read DHT sensor.");
        }

        vTaskDelay(pdMS_TO_TICKS(1000));
    }
    vTaskDelete(NULL);
}

void bh1750_task(void* arg)
{
    i2c_dev_t dev = { 0 };
    if (bh1750_init_desc(&dev, BH1750_ADDR_LO, 0, I2C_SDA, I2C_SCK) !=
ESP_OK ||
        bh1750_setup(&dev, BH1750_MODE_CONTINUOUS, BH1750_RES_HIGH) !=
ESP_OK)
    {
        ESP_LOGE(TAG_BH1750, "Failed to initialize BH1750.");
        vTaskDelete(NULL);
    }

    while (1)
    {
        uint16_t lux;
        if (bh1750_read(&dev, &lux) == ESP_OK)
        {
            ESP_LOGI(TAG_BH1750, "Light Intensity: %d lux", lux);

            // Create JSON payload
            cJSON* json = cJSON_CreateObject();
            cJSON_AddNumberToObject(json, "light_intensity", lux);
            char* data = cJSON_Print(json);

            // Send data to Firebase
            if (http_client_post_req(data, FIREBASE_LIGHT_URL) == ESP_OK)
            {
                ESP_LOGI(TAG, "BH1750 data uploaded successfully.");
            }
            else
            {
                ESP_LOGE(TAG, "Failed to upload BH1750 data.");
            }

            cJSON_Delete(json);
```

```
            free(data);
        }
        else
        {
            ESP_LOGE(TAG_BH1750, "Failed to read BH1750.");
        }

        vTaskDelay(pdMS_TO_TICKS(1000)); // Delay for 1 second
    }
    vTaskDelete(NULL);
}

void app_main(void) {

    ESP_ERROR_CHECK(nvs_flash_init());
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());

    esp_err_t err = ESP_FAIL;
    while (err != ESP_OK)
    {
        err = example_connect();
        if (err != ESP_OK)
        {
            ESP_LOGE(TAG, "Unable to connect to WiFi.");
            vTaskDelay(pdMS_TO_TICKS(10000));
        }
    }
    if (i2cdev_init() != ESP_OK) {
        ESP_LOGE(TAG_WIFI, "Failed to initialize I2C.");
        return;
    }
    // Start sensor tasks
    xTaskCreate(dht_task, "DHT Task", 4096, NULL, 2, NULL);
    xTaskCreate(bh1750_task, "BH1750 Task", 4096, NULL, 2, NULL);
    xTaskCreate(button_task, "Button Task", 4096, NULL, 2, NULL);
    vTaskDelete(NULL);
}
```