RISETECH INTERNSHIP
IoT Application Development with the ESP32 Using the ESP-IDF

INTERN NAME:

HUZAIFA SHAHBAZ

SUPERVISIOR:

SIR USMAN AKARM

SIR SAJID GUL KHAWAJA

DATE:

12TH JULY,2024

DEPARTMENT OF COMPUTER ENGINEERING

NUST EME COLLEGE, RAWALPINDI

PERFACE

I WOULD DEDICATE MY WORK TO MY FAMILY, TEACHERS, FRIENDS AND MY SUPERVISIOR.THANK YOU SO MUCH FOR YOUR SUPPORT.

IoT Application Development with the ESP32 Using the ESP-IDF

Contents

CHAPTER #1 ABSTRACT 17

1.1ABSTRACT: 17

CHAPTER#2 INTRODUCTION 17

2.1 INTRODUCTION: 17

CHAPTER#3 OBJECTIVES 18

3.1 OBJECTIVES: 18

3.1.1 HARDWARE INTEGRATION: 18

3.1.2 WiFi CONNECTIVITY: 18

3.1.3 WEB SERVER IMPLEMENTATION: 18

3.1.4 OTA FIRMWAE UPDATES: 18

3.1.5 SENSOR INTERGRATION: 18

3.1.6 AWS IOT INTEGRATION: 18

3.1.7 OPTIMIZED POWER MANAGEMENT: 18

CHAPTER#4 HARDWARE/SOFTWARE TOOLS 19

4.1 HARDWARE TOOLS: 19

4.2 SOFTWARE TOOLS: 19

CHAPTER# 5 FUNCTIONAL REQUIREMENTS 19

5.1 FUNCTIONAL REQUIREMENTS: 19

5.1.1 HARDWARE INTEGRATION 20

5.1.1.1 SENSOR INTEGRATION: 20

5.1.1.2 ACTUATOR CONTROL: 20

5.1.2 NETWORK CONNECTIVITY 20

5.1.2.1 WIFI CONNECTIVITY: 20

5.1.2.2 SECURE COMMUNICATION: 20

5.1.3 APPLICATION LOGIC 20

5.1.3.1 DATA PROCESSING: 20

5.1.3.2 DECISION MAKING: 20

5.1.4 USER INTERFACE: 21

5.1.4.1 WEB SERVER: 21
5.1.4.2 MOBILE APP INTEGRATION: 21
5.1.5 FIRMWARE UPDATES: 21
5.1.5.1 OVER-THE-AIR(OTA) UPDATES: 21
5.1.6 POWER MANAGEMENT: 21
5.1.6.1 POWER EFFICENCY: 21
5.1.7 ERROR HANDLING AND LOGGING: 21
5.1.7.1 ERROR HANDLING: 21
5.1.7.2 LOGGING: 22
5.1.8 SECURITY: 22
5.1.8.1 DEVICE AUTHENTICATION: 22
5.1.8.2 DATA ENCRYPTION: 22
5.1.9 TIME SYNCHRONIZATION: 22
5.1.9.1 SNTP INTEGRATION: 22
5.1.10 INTEGRATION WITH CLOUD SERVICES: 22
5.1.10.1 CLOUD CONNECTIVITY: 22
CHAPTER#6 NON-FUNCTIONAL REQUIREMENTS 22
6.1 NON-FUNCTIONAL REQUIREMENTS: 22
6.1.1 PERFORMANCE: 23
6.1.1.1 RESPONSE TIME: 23
6.1.1.2 THROUGHPUT: 23
6.1.2 SCALABILITY: 23
6.1.3 RELIABILITY: 23
6.1.3.1 UPTIME: 23
6.1.3.2 FAULT TOLERANCE: 23
6.1.4 MAINTAINABILITY: 23
6.1.4.1 CODE QUALITY: 23
6.1.4.2 UPDATE MANAGEMENT: 24
6.1.5 USABILITY: 24
6.1.5.1 USER INTERFACE: 24
6.1.5.2 CONFIGURATION: 24
6.1.6 SECURITY: 24
6.1.6.1 DATA SECURITY: 24

6.1.6.2 ACCESS CONTROL: 24
6.1.7 INTEROPERABILITY: 24
6.1.7.1 STANDARS COMPLIENCE: 24
6.1.7.2 COMPATABILITY: 24
6.1.8 PORTABILITY: 25
6.1.8.1 PLATFORM INDEPENDENCE: 25
6.1.8.2 DEVELOPMENT ENVIRONMENT: 25
6.1.9 EFFICIENCY: 25
6.1.9.1 RESOURCE UTILIZATION: 25
6.1.9.2 BATTERY LIFE: 25
6.1.10 REGULATOR COMPLIANCE: 25
6.1.10.1 CERTIFICATION: 25
6.1.11 SCALABILITY: 25
6.1.11.1 VERTICAL SCALABILITY: 25
6.1.11.2 HORIZONTAL SCALABILITY: 26
6.1.12 LOCALIZATION: 26
6.1.12.1 LANGUAGE SUPPORT: 26
6.1.12.2 TIME ZONE SUPPORT: 26
6.1.13 ENVIRONMENTAL CONSIDERATIONS: 26
6.1.13.1 OPERATING CONDITIONS: 26
6.1.14 BACKUP AND RECOVERY: 26
6.1.14.1 DATA BACKUP: 26
6.1.14.2 RECOVERY PROCEDURES: 26
CHAPTER#7 INTRODUCTION OF ESP32 USING ESP-IDF 27
7.1 INTRODUCTION OF ESP32 USING ESP-IDF: 27
7.2 DEVELOP A WLAN ON THE ESP32 AND CONNECT TO AWS IOT: STEP BY STEP PROGRAMMING IN C LANGUAGE: 27
7.2.1 WIFI APLICATION: 27
7.2.2 HTTP SERVER: 27
7.2.3 OTA FIRMWARE UPDATE: 28
7.2.4 DHT22 SENSOR: 28
7.2.5 NON-VOLATILE STORAGE: 28
7.2.6 RGB LED: 28

7.2.7 SNTP: 28
7.2.8 BUTTON WITH INTERRUPT AND SEMAPHORE: 28
7.3 SIMPLIFEID OVERVIEW OF WIFI APPLICATION: 28
7.3.1 Connecting Devices: 29
7.3.2 ESP32 SOFTAP/STATION: 29
7.3.3 ACCESS POINT: 30
7.4 AWS IOT CORE CONFIGURATION AND IMPLEMENTATION: 30
7.4.1 INTEGRATE THE ESP AWS IOT FRAMEWORK: 30
7.4.2 CONFIGURE AWS IOT CORE WITH ESP32: 30
7.4.3 SOURCE CODE UPDATE TO PUBLISH/SUBSCRIBE DATA UTILIZING MQTT: 30
7.4.4 TEST USING THE MQTT TEST CLIENT: 31
7.5 THE ESP-IDF: 31
7.5.1 ESPRESSIF'S OFFICAL IOT DEVELOPMENT FRAMEWORK: 31
7.5.2 FREERTOS DETERMINISTIC TASK SCHEDULER FOR ESP-IDF: 31
7.6 APPLICATION DEVELOPMENT USING ESP-IDF APIS AND FREERTOS: 31
7.6.1 EXTENSIBLE, MODULAR APPLICATION PROGRAMMING USING ESP IDF AND FREERTOS: 31
7.7 PROGRAMMING LANGUAGES USED 32
7.7.1 USING THE C PROGRAMMING LANGUAGE THROUGHOUT THE COURSE: 32
7.8 A COURSE ON USING THE ESP-IDF: 32
7.8.1 ESP-IDF for serious embedded software development: 32
CHAPTER#8 HARDWARE AND SOFTWARE REQUIREMENTS 32
8.1 HARDWARE AND SOFTWARE REQUIREMENTS: 32
8.1.1 HARDWARE REQUIREMENTS: 33
8.1.1.1 ESP32 DEVELOPMENT: 33
8.1.1.2 USB CABLE: 33
8.1.1.3 POWER SUPPLY: 33
8.1.1.4 SENSORS AND ACTUATORS: 33
8.1.1.5 BREADBOARD AND JUMPER WIRES: 33
8.1.1.6 DEBUGGING TOOLS: 33
8.1.2 SOFTWARE REQUIREMENTS: 33
8.1.2.1 OPERATING SYSTEMS: 33
8.1.2.2 ESP-IDF: 34

8.1.2.3 TOOLCHAIN: 34
8.1.2.4 PYTHON: 34
8.1.2.5 SERIAL COMMUNICATION TOOL: 34
8.1.2.6 INTEGRATED DEVELOPMENT ENVIRONMENT(IDE): 34
8.1.2.7 DRIVERS: 34
8.1.2.8 NETWORK ACCESS: 34
CHAPTER#9 ESP32 DEVKITS 35
9.1 ESP32 DEVKITS: 35
9.1.1 ESP32-DEVKIT C: 35
9.1.1.1 FEATURES: 35
9.1.1.2 USE CASE: 35
9.1.2 ESP32-WROVER-KIT: 35
9.1.2.1 FEATURES: 35
9.1.2.2 USE CASE: 36
9.1.3 ESP32-PICO KIT: 36
9.1.3.1 FEATURES: 36
9.1.3.2 USE CASE: 36
9.1.4 ESP32-S2-SAOLA-1: 36
9.1.4.1 FEATURES: 36
9.1.4.2 USE CASE: 37
9.1.5 ESP32-S3-DEVKITC-1: 37
9.1.5.1 FEATURES: 37
9.1.5.2 USE CASE: 37
9.1.6 ESP32-CAM: 37
9.1.6.1 FEATURES: 37
9.1.6.2 USE CASE: 38
CHAPTER#10 Development Environment Setup, and ESP-IDF Build System & CMake Overview 38
10.1 Development Environment Setup: 38
10.1.1 INSTALL PREREQUISTS: 38
10.1.1.1 WINDOWS: 38
10.1.1.2 MAC OS: 38
10.1.1.3 LINUX: 38

10.2 SET UP THE ESP-IDF: 38
10.3 ESP-IDF BUILD SYSTEM AND CMAKE OVERVIEW: 39
10.4 ECLIPSE IDE: 39
CHAPTER#11 ESP-IDF FREERTOS 40
11.1 FREERTOS RESOURCES: 40
11.2 DIFFERENCES FROM VANILLA FREERTOS: 40
11.3 FREERTOS TASK CREATION: 40
11.4 x Task Create Pinned to core parameters: 40
11.5 FREERTOS TASKS: 41
11.6 vTASKDELAY: 41
CHAPTER#12 ERROR HANDLING 41
12.1 RECOVERABLE ERRORS: 41
12.2 FATAL ERRORS: 42
12.3 ERROR CODES: 42
12.4 CONVERTING ERROR CODES TO ERROR MESSAGES: 42
12.5 ESP_ERROR_CHECK MARCO: 42
12.6 ERROR HANDLING PATTERNS: 43
12.6.1 ATTEMPT TO RECOVER: 43
12.6.2 PROPAGATE THE ERROR TO THE CALLER: 43
12.6.3 Convert to an Unrecoverable Error: 43
CHAPTER#13 CODE OF SUPPLEMENT LESSONS: 43
13.1 CODE OF CODING SYLE.H: 43
13.2 CODE OF CODE STYLE.C: 45
13.3 CODE OF MAIN.C: 46
CHAPTER#14 ESP-IDF COMPONENTS LIBRARY 47
14.1 ESP-IDF COMPONENTS LIBRARY: 47
14.1.2 CORE TEMPERATURES: 47
14.1.2.1 FREERTOS: 47
14.1.2.2 ESP SYSTEM: 48
14.1.2.3 LOG: 48
14.1.2.4 NWELIB: 48
14.1.3 PERIPHERAL DEVICES: 48
14.1.3.1 GPIO: 48

14.1.3.2 ADC/DAC: 48
14.1.3.3 12C, SPI, URAT: 48
14.1.3.4 PWM: 48
14.1.3.5 TOUCH SENSOR: 48
14.1.4 NETWORKING: 49
14.1.4.1 WIFI: 49
14.1.4.2 BLUETOOTH: 49
14.1.4.3 TCP/IP: 49
14.1.4.4 MQTT, HTTP/HTTPS: 49
14.1.5 STORAGE: 49
14.1.5.1 NVS: 49
14.1.5 SPIFFS, FATFS: 49
14.1.6 SECURITY: 49
14.1.6.1 mbedTLS: 49
14.1.6.2 SECURE BOOT, FLASH ENCRYPTION: 49
14.1.7 UTILITIES: 50
14.1.7.1 CONSOLE: 50
14.1.7.2 ESP TIMER: 50
14.1.7.3 ESPRESSIF OTA: 50
14.1.8 EXAMPLES: 50
14.1.8.1 EXAMPLE GPIO: 50
14.1.8.2 WIFI EXAMPLE: 50
CHAPTER#15 PROJECT STRUCTURE 51
15.1 APPLICATION PROJECT STRUCTURE: 51
CHAPTER#16 INTEGRATION STEPS PREVIEW 51
16.1 INTEGRATE LIBRARY: 51
16.2 INTEGRATE EXAMPLES: 51
CHAPTER#17 EXAMPLE DEMONSTRATION 52
17.1 EXAMPLE DEMONSTRATION: 52
CHAPTER#18 BME680 SENSOR 52
18.1 CODE OF BME680 SENSOR.H: 52
18.2 CODE OF BME680 SENSOR.C: 52
18.3 CODE OF BME680_SENSOR MAIN.C: 54

18.4 CODE OF BME680_SENSOR TASKS_COMMON.H: 55
CHAPTER#19 GIT REPOSITORY 56
19.1 Setting Up a Git Repository for ESP32 Development with ESP-IDF: 56
19.1.1 INSTALL GIT: 56
19.1.2 Initialize a Git Repository: 56
19.1.3 Add and Commit Files: 57
19.1.4 CREATE A REMOTE REPOSITORY: 57
CHAPTER#20 RGB APPLICATION STATUS LED 57
20.1 RGB LED STATUS COLORS: 57
20.2 LED Control (LEDC) ESP-IDF: 57
20.3 COLOR FUNCTIONS: 58
20.3.1 RGB Set Color Function: 58
20.3.2 The Status Functions Call the Set Color Function at Specified Duty Cycles: 58
20.4 RGB LEDS: 58
CHAPTER#21 CODE OF RGB APPLICATION STATUS LED 59
21.1 CODE OF RGB_LED.H: 59
21.2 CODE OF RGB_LED.C: 60
21.3 CODE OF MAIN.C: 63
CHAPTER#22 WIFI APPLICATION 64
22.1 WIFI IMPLEMENTATION: 64
22.2 SIMPLIFEID OVERVIEW OF WIFI APPLICATION: 64
22.2.1 Connecting Devices: 64
22.2.2 ESP32 SOFTAP/STATION: 65
22.2.3 ACCESS POINT: 65
22.3 Notable ESP-IDF Functions Used in This Section: 66
CHAPTER#23 CODE OF WIFI APP 66
23.1 CODE OF WIFI_APP.H: 66
23.2 CODE OF WIFI_APP.C: 68
23.3 CODE OF MAIN.C: 75
23.4 CODE OF TASKSCOMMON.H: 76
CHAPTER#24 HTTP SERVER 76
24.1 HTTP SERVER REQUIREMENTS: 76
24.2 WEB PAGES FILES: 77

24.3 CREATING AN HTTP SERVER USING THE ESP-IDF: 77
CHAPTER#25 CODES OF HTTP SERVER 78
25.1 CODE OF HTTP SERVER.C: 78
25.2 CODE OF HTTP SERVER.H: 85
25.3 CODE OF TASK_COMMON.H: 87
25.3 CODE OF WIFI_APP.C: 87
25.4 CODE OF APP.CSS: 94
25.4 CODE OF APP.JS: 96
25.4 CODE OF INDEX.HTML: 100
CHAPTER#26 OTA FIRMWARE UPDATES 101
26.1 ABOUT OTA UPDATES: 101
26.2 ABOUT PARTITION TABLES: 101
26.3 MORE ABOUT PARTITION TABLES: 102
26.4 OTA UPDATE REQUIREMENTS: 102
26.5 UTILIZING ESP-IDF FOR OTA UPDATES: 103
CHAPTER#27 CODES OF OTA FIRMWARE UPDATES 103
27.1 CODE OF HTTP_SERVER.H: 103
27.2 CODE OF HTTP_SERVER.C: 105
CHAPTER#28 DHT22 SENSOR 118
28.1 AM2302 DHT22 SENSOR TEMOERATURE AND HUMDITITY SENSOR MODULE: 118
28.2 PIN CONNECTION: VCC, DATA (GPIO 25) AND GROUND: 119
28.3 About the DHT22 Sensor Implementation: 119
CHAPTER#29 CODES OF DHT22 SENSOR 120
29.1 CODE OF DHT22.C: 120
29.2 CODE OF DHT22.H: 125
29.3 CODE OF HTTP_SERVER.C: 126
29.4 CODE OF MAIN.C: 140
29.5 CODE OF APP.CSS: 141
29.5 CODE OF APP.JS: 143
29.6 CODE OF INDEX.HTML: 147
CHAPTER#30 WIFI CONNECT ESP 148
30.1 WIFI CONNECTION: 148
30.2 MAKING A WIFI CONNECTION USING THE ESP-IDF: 148

30.3 ESP IDF APIs USED (DISPLAYING CONECTION INFORMATION TO THE WEB PAGE…): 149
30.4 WIFI APPLICATION: 149
30.4.1 CONNECTING DEVICES: 149
30.4.2 ESP32 SOFTAP: 150
30.4.3 ACCESS POINT: 150
CHAPTER#31 CODES OF WIFI CONNECT ESP 151
31.1 CODE OF WIFI_APP.H: 151
31.2 CODE OF WIFI_APP.C: 153
31.3 CODE OF HTTP_SERVER.H: 162
31.4 CODE OF HTTP_SERVER.C: 164
31.5 CODE OF INDEX_HTML: 181
31.6 CODE OF APP.CSS: 183
31.7 CODE OF APP.JS: 185
CHAPTER#32 DISPLAYING WIFI CONNECTION INFORMATION ON THE WEB PAGE 192
32.1 DISPLAYING WIFI CONNECTION INFORMATION ON THE WEB PAGE: 192
CHAPTER#33 CODE OF WIFI CONNECTION INFORMATION ON THE WEB PAGE: 193
33.1 CODE OF HTTP_SERVER.C: 193
33.2 CODE OF APP.CSS: 211
33.3 CODE OF APP.JS: 214
33.4 CODE OF INDEX.HTML: 222
CHAPTER#34 WIFI DISCONNECT BUTTON 224
34.1 WIFI DISCONNECT BUTTON: 224
CHAPTER#35 CODES OF WIFI DISCONNECT BUTTON 225
35.1 CODE OF HTTP_SERVER.C: 225
35.2 CODE OF WIFI_APP.C: 244
35.3 CODE OF WIFI_APP.H: 253
35.4 CODE OF APP.JS: 255
CHAPTER#36 NON-VOLATILE STORAGE (NVS) 263
36.1 SAVING, LOADING, CLEARING WIFI CREDENTIALS: 263
36.2 NVS FOR WIFI CREDENTIALS USING THE ESP-IDF: 264
36.2.1 SAVING THE CREDENTIALS: 264
36.2.2 GETTING THE CREDENTIALS: 264

36.2.3 CLEARING THE CREDENTIALS: 264
CHAPTER#37 CODES OF NON-VOLATILE STORAGE 265
37.1 CODE OF APP_NVS.C: 265
37.2 CODE OF APP_NVS.H: 269
37.3 CODE OF HTTP_SERVER.C: 270
37.4 CODE OF HTTP_SERVER.H: 289
37.5 CODE OF WIFI_APP.C: 291
37.5 CODE OF WIFI_APP.H: 303
CHAPTER#38 BUTTON WITH INTERRUPT AND SEAMAPHORE 306
38.1 DISCONNECTING WIFI AND CLEARING CREDENTIALS USING THE BOOT BUTTON ON THE DEVKIT: 306
38.2 GPIO INTERRUPT FOR WIFI RESET BUTTON: 306
CHAPTER#39 CODES OF BUTTON WITH INTERRUPT AND SEAMAPHORE 307
39.1 CODE OF WIFI_APP.C: 307
39.2 CODE OF WIFI_RESET_BUTTON.C: 320
39.3 CODE OF WIFI_RESET_BUTTON.H: 322
CHAPTER#40 SNTP TIME SYNCHRONIZATION 322
40.1 SNTP TIME SYNCHRONIZATION: 322
40.2 SNTP: 323
40.3 Utilizing ESP-IDF for SNTP Time Synchronization: 323
CHAPTER#41 TIME ZONES 324
41.1 AUSTRAILA TIME ZONE: 324
41.2 EUROPE TIME ZONE: 325
41.3 NEW ZELAND TIME ZONE: 326
41.4 USA AND CANADA: 326
41.5 ASIA TIME ZONE: 327
41.6 CENTRAL AND SOUTH AMERICA: 327
CHAPTER#42 CODES OF SNTP TIME SYNCHRONIZATION 327
42.1 CODE OF MAIN.C: 327
42.2 CODE OF SNTP_TIME_SYNC.C: 328
42.3 CODE OF SNTP_TIME_SYNC.H: 331
42.4 CODE OF TASKS_COMMON.H: 332
42.4 CODE OF WIFI_APP.C: 333

42.5 CODE OF WIFI_APP.H: 346
42.6 CODE OF APP.CSS: 349
42.7 CODE OF APP.JS: 353
42.8 CODE OF INDEX.HTML: 362
42.9 CODE OF HTTP_SERVER.H: 364
42.10 CODE OF HTTP_SERVER.C: 367
CHAPTER#43 DISPLAYING THE ESP32'S ACCESS POINT SSID ON THE WEBPAGE 396
43.1 DISPLAYING THE ESP32'S ACCESS POINT SSID ON THE WEBPAGE: 396
CHAPTER#44 CODES OF DISPLAYING THE ESP32'S ACCESS POINT SSID ON THE WEBPAGE 397
44.1 CODE OF HTTP_SERVER.C: 397
44.2 CODE OF APP.CSS: 427
44.3 CODE OF APP.JS: 431
44.4 CODE OF INDEX.HTML: 445
CHAPTER#45 AWS IOT IMPLEMENTATION 449
45.1 AWS IOT CORE RESOURCES: 449
45.1.1 AWS IOT: 449
45.1.2 ABOUT MQTT AND AWS IOT: 449
45.2 ACCOUNT SETUP: 450
CHAPTER#46 AWS IOT TECHNICAL SUPPLEMENT 451
46.1 MQTT: 451
46.2 MQTT IOT CORE: 451
46.3 HOW MQTT WORKS: 452
46.4 AWS IOT DEVICE AUTHENTICATION: 452
46.5 MUTUAL TLS AUTHENTICATION: 453
46.6 AWS SECURITY OVERVIEW: 453
46.7 AWS IOT POLICIES: 454
CHAPTER#47 CODES OF AWS IOT 455
47.1 CODE OF AWS_IOT.C: 455
47.2 CODE OF AWS_IOT.H: 463
47.3 CODE OF WIFI_APP.H: 464
47.4 CODE OF WIFI_APP.C: 468
47.5 CODE OF DEMO_CONFIG.H: 486

47.6 CODE OF MAIN.C: 492
47.7 CODE OF MQTT_DEMO_MUTUAL_AUTH.C: 494
CHAPTER#48 CONCLUSION 558
48.1 CONCLUSION: 558
REFERENCES: 558

IoT Application Development with the ESP32 Using the ESP-IDF

Figure 7.3.1:CONNECTING DEVICES OF WIFI APPLICATION TO ESP32 29
Figure 7.3.2:ESP32 SOFTAP/STATION OF WIFI APPLICATIONS 29
Figure 7.3.3:ACCESS POINT OF CONNECTING DEVICES OF WIFI APPLICATION 30
Figure 11.6.4:CODE OF VTASKDELAY 41
Figure 12.5.5:OUTPUT OF ESP_ERROR_CHECK MARCO 43
Figure 15.1.6:PROJECT STRUCTURE OF ESPRESSIF IDF 51
Figure 22.1.7:CONNECTING DEVICES OF WIFI APPLICATION TO ESP32 65
Figure 22.2.8:ESP32 SOFTAP/STATION OF WIFI APPLICATIONS 65
Figure 22.2.9:ACCESS POINT OF CONNECTING DEVICES OF WIFI APPLICATION 65
Figure 26.3.10: Description of “Factory app, two OTA definitions” configuration 102
Figure 28.2.11:PIN CONNECTION:VCC ,DATA(GPIO 25) AND GROUND 119
Figure 30.4.12: CONNECTING DEVICES OF WIFI APPLICATION TO ESP32 150
Figure 30.4.13: ESP32 SOFTAP/STATION OF WIFI APPLICATIONS 150
Figure 30.4.14: ACCESS POINT OF CONNECTING DEVICES OF WIFI APPLICATION 151
Figure 38.2.15:GPIO INTERRUPT FOR WIFI RESET BUTTON 307
Figure 41.1.16AUSTAILA TIME ZONES 324
Figure 41.2.17:EUROPE TIME ZONE 326
Figure 41.3.18:NEW ZELAND TIME ZONE 326
Figure 41.4.19:USA AND CANADA TIME ZONE 326
Figure 41.5.20:ASIA TIME ZONE 327
Figure 41.6.21:CENTRAL AND SOUTH AMERICA TIME ZONES 327
Figure 45.1.22:AWS IOT CORE RESOURCES 450
Figure 45.2.23:AWS ACCOUNT SETUP 451
Figure 45.5.24:BASIC SUPPORT PLAN 451
Figure 45.3.25:WORKING OF MQTT 452
Figure 46.4.26:AWS IOT DEVICE AUTHENTICATION 453
Figure 46.6.27:AWS SECURITY OVERVIEW 454

IoT Application Development with the ESP32 Using the ESP-IDF

CHAPTER #1 ABSTRACT

1.1 ABSTRACT:

In this abstract the IOT application development with the ESP32 using the ESP-IDF.In this internship I had done my work on Espressif hardware ESP32 Devkits, Development Environment Setup, and ESP-IDF Build System & CMake Overview, Supplemental Lessons, Components & Sensors Library , Course Source Code, RGB Application Status LED, WiFi Application, HTTP Server, OTA Firmware Update, DHT22 Sensor, WiFi Connect Implementation, Displaying WiFi Connection Information on the Web Page, WiFi Disconnect Button, Non-Volatile Storage (NVS), Button with Interrupt & Semaphore, SNTP Time Synchronization, Displaying the ESP32's Access Point SSID on the Web Page and AWS IoT. Eclipse IDE, ESPRESSIF IDF, ESP32, DHT22 -SENSOR.

CHAPTER#2 INTRODUCTION

2.1 INTRODUCTION:

IoT Application Development with the ESP32 using ESP-IDF involves leveraging its robust framework for hardware integration and WiFi connectivity. It includes setup via ESP-IDF and CMake, integrating sensors like DHT22, implementing HTTP servers, OTA updates, and AWS IoT integration for scalable deployments. The development focuses on efficient WiFi management, non-volatile storage (NVS) for data persistence, and leveraging hardware interrupts for responsive applications.

CHAPTER#3 OBJECTIVES

3.1 OBJECTIVES:

Here are key objectives for IoT Application Development with the ESP32 using ESP-IDF:

3.1.1 HARDWARE INTEGRATION:

Utilize ESP-IDF for seamless integration of sensors, actuators, and peripherals.

3.1.2 WiFi CONNECTIVITY:

Implement reliable WiFi connections for data transmission and remote access.

3.1.3 WEB SERVER IMPLEMENTATION:

Develop HTTP servers to facilitate communication with web and mobile interfaces.

3.1.4 OTA FIRMWAE UPDATES:

Enable Over-The-Air updates for efficient maintenance and feature enhancements.

3.1.5 SENSOR INTERGRATION:

Incorporate sensors like DHT22 for environmental data collection and monitoring.

3.1.6 AWS IOT INTEGRATION:

Integrate with AWS IoT services for cloud-based data storage and processing.

3.1.7 OPTIMIZED POWER MANAGEMENT:

Implement strategies for power efficiency to extend device operation in IoT applications.

CHAPTER#4 HARDWARE/SOFTWARE TOOLS

4.1 HARDWARE TOOLS:

The HARDWARE TOOLS ARE FOLLOWING ARE:

1.ESP32-DevKitC V4 board

2. USB A/micro–USB B cable

3. DHT22SENSOR

4. RGB LED SENSOR

5. BREADBORAD

6. JUMPING WIRES

4.2 SOFTWARE TOOLS:

The SOFTWARE TOOLS ARE FOLLOWING ARE:

1.ECLIPSE IDE

2.ESPRESSIF IDF

CHAPTER# 5 FUNCTIONAL REQUIREMENTS

5.1 FUNCTIONAL REQUIREMENTS:

Developing an IoT application with the ESP32 using the ESP-IDF involves several functional requirements to ensure the device operates as intended. Here are some key functional requirements typically considered:

5.1.1 HARDWARE INTEGRATION

5.1.1.1 SENSOR INTEGRATION:

Connect and interface with sensors (like DHT22 for temperature and humidity) to gather data.

5.1.1.2 ACTUATOR CONTROL:

Interface with actuators (like LEDs, motors) to perform actions based on data or commands.

5.1.2 NETWORK CONNECTIVITY

5.1.2.1 WIFI CONNECTIVITY:

Establish and maintain a WiFi connection to communicate with other devices or servers.

5.1.2.2 SECURE COMMUNICATION:

Implement secure communication protocols (like HTTPS or TLS) for data transmission.

5.1.3 APPLICATION LOGIC

5.1.3.1 DATA PROCESSING:

Process sensor data (temperature, humidity) and apply logic (e.g., threshold checks).

5.1.3.2 DECISION MAKING:

Implement decision-making algorithms (e.g., control logic for HVAC systems based on sensor data).

5.1.4 USER INTERFACE:

5.1.4.1 WEB SERVER:

Host a web server on the ESP32 for configuration, monitoring, or control via a web interface.

5.1.4.2 MOBILE APP INTEGRATION:

Integrate with mobile apps (iOS/Android) for remote monitoring or control.

5.1.5 FIRMWARE UPDATES:

5.1.5.1 OVER-THE-AIR(OTA) UPDATES:

Support OTA updates to ensure the device's firmware can be updated remotely.

5.1.6 POWER MANAGEMENT:

5.1.6.1 POWER EFFICENCY:

Implement strategies for power management to optimize battery life (if applicable).

5.1.7 ERROR HANDLING AND LOGGING:

5.1.7.1 ERROR HANDLING:

Implement mechanisms to handle errors gracefully (e.g., network disconnections).

5.1.7.2 LOGGING:

Record events and data for troubleshooting and analysis purposes.

5.1.8 SECURITY:

5.1.8.1 DEVICE AUTHENTICATION:

Ensure only authorized devices can communicate with the ESP32.

5.1.8.2 DATA ENCRYPTION:

Ensure only authorized devices can communicate with the ESP32.

5.1.9 TIME SYNCHRONIZATION:

5.1.9.1 SNTP INTEGRATION:

Synchronize time with SNTP servers to timestamp data accurately.

5.1.10 INTEGRATION WITH CLOUD SERVICES:

5.1.10.1 CLOUD CONNECTIVITY:

Integrate with cloud platforms (like AWS IoT, Google Cloud IoT) for data storage, analytics, or remote control.

CHAPTER#6 NON-FUNCTIONAL REQUIREMENTS

6.1 NON-FUNCTIONAL REQUIREMENTS:

Non-functional requirements are crucial for ensuring the overall performance, reliability, and usability of the IoT application developed

with the ESP32 using the ESP-IDF. Here are some key non-functional requirements to consider:

6.1.1 PERFORMANCE:

6.1.1.1 RESPONSE TIME:

Ensure minimal latency in sensor data processing and actuator response.

6.1.1.2 THROUGHPUT:

Ensure minimal latency in sensor data processing and actuator response.

6.1.2 SCALABILITY:

The system should be able to handle an increasing number of devices without significant performance degradation.

6.1.3 RELIABILITY:

6.1.3.1 UPTIME:

The system should have high availability and minimal downtime.

6.1.3.2 FAULT TOLERANCE:

The system should handle failures gracefully, ensuring continuous operation.

6.1.4 MAINTAINABILITY:

6.1.4.1 CODE QUALITY:

Ensure the code is modular, well-documented, and follows best practices for ease of maintenance.

6.1.4.2 UPDATE MANAGEMENT:

Facilitate easy deployment of updates, including OTA updates.

6.1.5 USABILITY:

6.1.5.1 USER INTERFACE:

If applicable, ensure the web interface or mobile app is intuitive and user-friendly.

6.1.5.2 CONFIGURATION:

The system should be easy to configure and set up.

6.1.6 SECURITY:

6.1.6.1 DATA SECURITY:

Ensure data integrity and confidentiality through encryption and secure communication protocols.

6.1.6.2 ACCESS CONTROL:

Implement robust authentication and authorization mechanisms.

6.1.7 INTEROPERABILITY:

6.1.7.1 STANDARS COMPLIENCE:

Adhere to relevant industry standards and protocols (e.g., MQTT, CoAP) for device communication.

6.1.7.2 COMPATABILITY:

Ensure compatibility with other devices and systems in the ecosystem.

6.1.8 PORTABILITY:

6.1.8.1 PLATFORM INDEPENDENCE:

The application should be easily portable across different hardware platforms (e.g., different versions of ESP32).

6.1.8.2 DEVELOPMENT ENVIRONMENT:

Maintain consistency in the development environment setup across different machines.

6.1.9 EFFICIENCY:

6.1.9.1 RESOURCE UTILIZATION:

Optimize the use of CPU, memory, and power resources.

6.1.9.2 BATTERY LIFE:

Optimize the use of CPU, memory, and power resources.

6.1.10 REGULATOR COMPLIANCE:

6.1.10.1 CERTIFICATION:

Ensure the system complies with relevant regulatory standards and certifications (e.g., FCC, CE).

6.1.11 SCALABILITY:

6.1.11.1 VERTICAL SCALABILITY:

Ability to enhance performance by adding more resources (e.g., CPU, memory).

6.1.11.2 HORIZONTAL SCALABILITY:

Ability to enhance performance by adding more devices or instances.

6.1.12 LOCALIZATION:

6.1.12.1 LANGUAGE SUPPORT:

Provide support for multiple languages if the system is deployed in different regions.

6.1.12.2 TIME ZONE SUPPORT:

Ensure correct time handling across different time zones.

6.1.13 ENVIRONMENTAL CONSIDERATIONS:

6.1.13.1 OPERATING CONDITIONS:

Ensure the device operates reliably under specified environmental conditions (e.g., temperature, humidity).

6.1.14 BACKUP AND RECOVERY:

6.1.14.1 DATA BACKUP:

Implement mechanisms for data backup to prevent data loss.

6.1.14.2 RECOVERY PROCEDURES:

Define clear procedures for system recovery in case of failure.

CHAPTER#7 INTRODUCTION OF ESP32 USING ESP-IDF

7.1 INTRODUCTION OF ESP32 USING ESP-IDF:

The ESP32, developed by Espressif Systems, is a versatile microcontroller with integrated Wi-Fi and Bluetooth, ideal for IoT applications. It features a dual-core processor, rich peripherals, and extensive memory. The ESP-IDF (Espressif IoT Development Framework) is the official framework for developing applications on the ESP32, offering a modular component-based structure and FreeRTOS support for multitasking. It includes extensive libraries for peripherals, networking, and security. With OTA update capability, comprehensive documentation, and strong community support, ESP-IDF facilitates robust and scalable IoT application development on the ESP32 platform.

7.2 DEVELOP A WLAN ON THE ESP32 AND CONNECT TO AWS IOT: STEP BY STEP PROGRAMMING IN C LANGUAGE:

WLAN APPLICATION DEVELOPMENT IMPLEMENTATING THE FOLLOWING FEARTURES:

7.2.1 WIFI APLICATION:

Extendable, connect to ESP's access point to access an HTTP server, connect the ESP32 to an access point.

7.2.2 HTTP SERVER:

Support OTA (over the air) firmware updates via a web page, connecting and disconnecting the ESP32 to an AP and displaying data on the page (temperature/humidity sensor data, local time using a SNTP, and ESP32's SSID).

7.2.3 OTA FIRMWARE UPDATE:

OTA updates allows the ESP32 to update itself wirelessly based on data received while the normal firmware is running (for example, over WiFi). In our implementation, the update is done within the WLAN via a web page.

7.2.4 DHT22 SENSOR:

Temperature and humidity sensor driver provided, and we will program the webserver and .html, .js and .css.

7.2.5 NON-VOLATILE STORAGE:

Saving and loading WiFi credentials.

7.2.6 RGB LED:

Will create functions to create different colors used to indicate applications status.

7.2.7 SNTP:

Used for getting local time and display on the webpage.

7.2.8 BUTTON WITH INTERRUPT AND SEMAPHORE:

Used for disconnecting WiFi and clearing credentials.

7.3 SIMPLIFEID OVERVIEW OF WIFI APPLICATION:

The simplified overview of WiFi application are:

7.3.1 Connecting Devices:

Becomes “station” of ESP32’s SoftAP when connected. DHCP service from the ESP32’s SoftAP will dynamically assign an IP to your device. Interact with the ESP32 via the web page.

Figure 7.3.1:CONNECTING DEVICES OF WIFI APPLICATION TO ESP32

7.3.2 ESP32 SOFTAP/STATION:

AP/STA combination mode. We assign an IP to the SOFTAP; the interface of the ESP32 is statically configured. DHCP server dynamically assigns an IP for connecting stations. We set a maximum number of stations allowed to connect.

Figure 7.3.2:ESP32 SOFTAP/STATION OF WIFI APPLICATIONS

7.3.3 ACCESS POINT:

When the ESP32 connects to an AP, local time can be obtained utilizing SNTP (if the AP is connected to the internet). DHCP service from the AP, will dynamically assign an IP to the ESP32 in our application.

Figure 7.3.3:ACCESS POINT OF CONNECTING DEVICES OF WIFI APPLICATION

7.4 AWS IOT CORE CONFIGURATION AND IMPLEMENTATION:

7.4.1 INTEGRATE THE ESP AWS IOT FRAMEWORK:

Update the project to include the framework into the build, which enables AWS IOT connectivity.

7.4.2 CONFIGURE AWS IOT CORE WITH ESP32:

Create a “Thing” in AWS.Create a policy and attach it to the device (“Thing”) certificate. Generate device certificate, Root CA and keys. Embedded certificates and keys to ESP32.

7.4.3 SOURCE CODE UPDATE TO PUBLISH/SUBSCRIBE DATA UTILIZING MQTT:

Temperature, Humidity and WiFi RSSI.

7.4.4 TEST USING THE MQTT TEST CLIENT:

View the data from AWS IOT core.

7.5 THE ESP-IDF:

7.5.1 ESPRESSIF'S OFFICAL IOT DEVELOPMENT FRAMEWORK:

Open-source and freely available on GitHub. Supports Windows, MacOS, and Linux. Development in C and C++.Production-ready and has a well-defined release process and support policy and each release undergoes a rigorous QA process. Feature-rich Software Components; including RTOS, Peripheral Drivers, Networking stack and various protocol implementations.

7.5.2 FREERTOS DETERMINISTIC TASK SCHEDULER FOR ESP-IDF:

ESP-IDF version of the FREERTOS kernel is modified for multicore support.

7.6 APPLICATION DEVELOPMENT USING ESP-IDF APIS AND FREERTOS:

7.6.1 EXTENSIBLE, MODULAR APPLICATION PROGRAMMING USING ESP IDF AND FREERTOS:

STEP-BY-STEP programming, with brief explanations of ESP-IDF APIs. The focus is not on theory, or about the ESP32, itself, this a hands-on-course programming, project-based course. We are utilizing FREERTOS;

tasks, message queues for intertask communication, event group for our state machine and binary semaphore for our reset button task.

7.7 PROGRAMMING LANGUAGES USED

7.7.1 USING THE C PROGRAMMING LANGUAGE THROUGHOUT THE COURSE:

HTML, JAVASCRIPT AND .CSS also used for the webpage. I used C PROGRAMMING LANGUAGE.

7.8 A COURSE ON USING THE ESP-IDF:

7.8.1 ESP-IDF for serious embedded software development:

Arduino for the ESP32 is just a wrapper around the ESP-IDF. Working directly with the IDF provides you with a better understanding of what goes on under the hood. ESP-IDF for professional ESP32-based on projects. Project setup using the IDF is easier than ever using the Eclipse plugin.

CHAPTER#8 HARDWARE AND SOFTWARE REQUIREMENTS

8.1 HARDWARE AND SOFTWARE REQUIREMENTS:

THE HARDWARE AND SOFTWARE REQUIREMENTS following are:

8.1.1 HARDWARE REQUIREMENTS:

The HARDWARE REQUIREMENTS FOLLOWING ARE:

8.1.1.1 ESP32 DEVELOPMENT:

Any ESP32-based development board (e.g., ESP32-WROOM-32, ESP32-DevKitC).

8.1.1.2 USB CABLE:

Micro-USB or USB-C cable for connecting the development board to the computer.

8.1.1.3 POWER SUPPLY:

External power supply if not powered via USB (typically 3.3V).

8.1.1.4 SENSORS AND ACTUATORS:

Various sensors (e.g., DHT22 for temperature and humidity, motion sensors). Actuators like LEDs, relays, or motors for control applications.

8.1.1.5 BREADBOARD AND JUMPER WIRES:

For prototyping and connecting components.

8.1.1.6 DEBUGGING TOOLS:

Optional: JTAG debugger for advanced debugging.

8.1.2 SOFTWARE REQUIREMENTS:

THE SOFTWARE REQUIREMENTS following are:

8.1.2.1 OPERATING SYSTEMS:

Windows, macOS, or Linux.

8.1.2.2 ESP-IDF:

Espressif IoT Development Framework, available for download from the Espressif GitHub repository or website.

8.1.2.3 TOOLCHAIN:

GCC compiler for the ESP32.CMake and Ninja build tools.

8.1.2.4 PYTHON:

Python 3.x for running ESP-IDF scripts.

8.1.2.5 SERIAL COMMUNICATION TOOL:

minicom, screen, or PuTTY for serial communication and monitoring.

8.1.2.6 INTEGRATED DEVELOPMENT ENVIRONMENT(IDE):

Visual Studio Code with the ESP-IDF extension for enhanced development experience.

8.1.2.7 DRIVERS:

USB-to-Serial drivers (e.g., CP210x or FTDI) if required for the development board.

8.1.2.8 NETWORK ACCESS:

For downloading tools, libraries, and for OTA updates and cloud integration during development.

CHAPTER#9 ESP32 DEVKITS

9.1 ESP32 DEVKITS:

THE ESP32 DEVKITS FOLLOWING ARE:

9.1.1 ESP32-DEVKIT C:

A compact development board featuring the ESP32-WROOM-32 module.

9.1.1.1 FEATURES:

1: USB-to-UART bridge

2: Multiple GPIOs available

3: Integrated antenna

4: Power supply options via USB or external sources

9.1.1.2 USE CASE:

General-purpose development, prototyping, and learning.

9.1.2 ESP32-WROVER-KIT:

An advanced development board with an ESP32-WROVER module, which includes additional PSRAM.

9.1.2.1 FEATURES:

1: 4.3" LCD display connector

2: MicroSD card slot

3: JTAG interface for debugging

4: USB-to-UART bridge

9.1.2.2 USE CASE:

Advanced applications requiring more memory, display interfacing, and debugging capabilities.

9.1.3 ESP32-PICO KIT:

A mini development board that features the ESP32-PICO-D4 SIP (System in Package).

9.1.3.1 FEATURES:

1: Integrated antenna

2: USB-to-UART bridge

3: Minimalist design

4: Breadboard-friendly

9.1.3.2 USE CASE:

Space-constrained applications, minimalistic designs, and quick prototyping.

9.1.4 ESP32-S2-SAOLA-1:

A development board based on the ESP32-S2 module, which is a single-core version of the ESP32 with improved security and USB-OTG functionality.

9.1.4.1 FEATURES:

1: Integrated USB-OTG

2: Multiple GPIOs

3: USB-to-UART bridge

9.1.4.2 USE CASE:

Applications requiring USB connectivity, enhanced security, and power efficiency.

9.1.5 ESP32-S3-DEVKITC-1:

Development board featuring the ESP32-S3 module, which offers enhanced AI and machine learning capabilities.

9.1.5.1 FEATURES:

1: Dual-core Xtensa LX7 processor

2: Integrated USB-OTG

3: Multiple GPIOs

4: USB-to-UART bridge

9.1.5.2 USE CASE:

AI and ML applications, advanced sensor processing, and complex control systems.

9.1.6 ESP32-CAM:

A development board featuring the ESP32-S and an integrated camera module.

9.1.6.1 FEATURES:

1: OV2640 camera module

2: MicroSD card slot

3: Wi-Fi and Bluetooth

4: Multiple GPIOs

9.1.6.2 USE CASE:

Image capture, video streaming, and camera-based IoT applications.

CHAPTER#10 Development Environment Setup, and ESP-IDF Build System & CMake Overview

10.1 Development Environment Setup:

Setting up the development environment for ESP32 using ESP-IDF involves installing the necessary tools and configuring the build environment. Here's a step-by-step guide:

10.1.1 INSTALL PREREQUISTS:

10.1.1.1 WINDOWS:

Install Python 3.7 or later from the website. Install Git from the website. Install the ESP-IDF Tools Installer from the Espressif download page.

10.1.1.2 MAC OS:

Install Homebrew from the website. Install Python and Git via Homebrew

brew install python git

10.1.1.3 LINUX:

Install Python and Git via the package manager:

sudo apt-get update

sudo apt-get install python3 python3-pip git

10.2 SET UP THE ESP-IDF:

1: Clone the ESP-IDF repository:

```
git clone --recursive https://github.com/espressif/esp-idf.git
cd esp-idf
```

2: Set up the environment variables:

```
./install.sh  # This script installs all the required tools.
. ./export.sh  # This script sets the environment variables for ESP-IDF.
```

3: Configure the Build Environment:

Use the ESP-IDF Command Prompt on Windows or a terminal on macOS/Linux.
Run the following commands to configure your build environment:

```
idf.py menuconfig
```

10.3 ESP-IDF BUILD SYSTEM AND CMAKE OVERVIEW:

The ESP-IDF build system uses CMake to configure and manage the build process for ESP32 applications. Projects and components are defined using `CMakeLists.txt` files, specifying source files and include directories. The `idf.py` tool simplifies project management, including configuration (`idf.py menuconfig`), building (`idf.py build`), flashing (`idf.py flash`), and monitoring (`idf.py monitor`). This setup provides a flexible and powerful environment for developing, testing, and deploying applications on the ESP32 platform.

10.4 ECLIPSE IDE:

Setting up Eclipse IDE for ESP32 development involves installing Eclipse and the necessary plugins, importing your ESP-IDF project, and configuring build settings. This provides a powerful graphical interface for developing, building, flashing, and monitoring ESP32 applications using ESP-IDF.

CHAPTER#11 ESP-IDF FREERTOS

11.1 FREERTOS RESOURCES:

FreeRTOS is a real-time operating system kernel for embedded devices.

11.2 DIFFERENCES FROM VANILLA FREERTOS:

The ESP-IDF FreeRTOS is adapted for multi-core support. Unlike Vanilla FreeRTOS, we don't have to call vTaskStartScheduler.FreeRTOS task stack size is specified in Bytes in ESP-IDF FreeRTOS not Word.

11.3 FREERTOS TASK CREATION:

Create tasks using the FreeRTOS xTaskCreate APIs.We must include freertos/task.h.xTaskCreate →Lets the ESP-IDF FreeRTOS choose which core the task runs on.xTaskCreatePinnedToCore →Allows for specifying which core the task runs on.

11.4 x Task Create Pinned to core parameters:

BaseType_t xTaskCreatePinnedToCore(TaskFunction_t pvTaskCode, const char *const pcName, const uint32_t usStackDepth, void *const pvParameters, UBaseType_t uxPriority, TaskHandle_t *const pvCreatedTask, const BaseType_t xCoreID)
pvTaskCode->This is the custom C function (task) that runs in an infinite loop. pcName ->Descriptive name for the task. Only used as a debugging aid. usStackDepth->Memory in bytes that should be allocated by the kernel to the task. pvParameters->Optional parameter. Pointer that can be used by the task.uxPriority->The priority at which the task should run on. Higher priority number takes precedence.pvCreatedTask->Optional task handle by which the created task can be referenced, e.g., if you need to use various FreeRTOS APIs, like, vTaskDelete. xCoreID->The core of the ESP32 the task is assigned to (Core 0 or Core 1).

11.5 FREERTOS TASKS:

Running →Executing and utilizing the processor.
Ready →Able to execute (not in the Blocked or Suspended states).
Blocked →Because of temporal event, e.g., call to vTaskDelay or external event, e.g., waiting for a queue, semaphore, event group, notification or semaphore event.

Suspended →There is no timeout and tasks only enter and exit the Suspended state when explicitly commanded to do so using vTaskSuspend or xTaskResume API calls.

11.6 vTASKDELAY:

vTaskDelay is used to send a task into Blocked state for a set number of Ticks. The actual time that the task remains blocked depends on the tick. The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate – with the resolution of one tick period.

```
void vTaskFunction( void * pvParameters )
{
// Block for 500ms.
const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

  for( ;; )
  {
      // Simply toggle the LED every 500ms, blocking between each toggle.
      vToggleLED();
      vTaskDelay( xDelay );
  }
}
```

Figure 11.6.4:CODE OF VTASKDELAY

CHAPTER#12 ERROR HANDLING

12.1 RECOVERABLE ERRORS:

Errors indicated by functions through return values (error codes).

12.2 FATAL ERRORS:

Failed assertions (using assert macro and equivalent methods, see Assertions) and abort() calls. CPU exceptions: access to protected regions of memory, illegal instruction, etc. System level checks: watchdog timeout, cache access error, stack overflow, stack smashing, heap corruption, etc.

12.3 ERROR CODES:

Most ESP-IDF-specific functions use esp_err_t type to return error codes. esp_err_t is a signed integer type.
Success (no error) is indicated with ESP_OK code, which is defined as zero.
Common error codes for generic failures (out of memory, timeout, invalid argument, etc.) are defined in esp_err.h file.
Various ESP-IDF header files define possible error codes using preprocessor defines. Usually these defines start with ESP_ERR_ prefix.

12.4 CONVERTING ERROR CODES TO ERROR MESSAGES:

For each error code defined in ESP-IDF components, esp_err_t value can be converted to an error code name using esp_err_to_name() or esp_err_to_name_r() functions.

12.5 ESP_ERROR_CHECK MARCO:

ESP_ERROR_CHECK macro checks esp_err_t value rather than a bool condition. If the argument of ESP_ERROR_CHECK is not equal ESP_OK, then an error message is printed on the console, and abort() is called.

Error message will typically look like this:

```
ESP_ERROR_CHECK failed: esp_err_t 0x107 (ESP_ERR_TIMEOUT) at 0x400d1fdf

file: "/Users/user/esp/example/main/main.c" line 20
func: app_main
expression: sdmmc_card_init(host, &card)

Backtrace: 0x40086e7c:0x3ffb4ff0 0x40087328:0x3ffb5010 0x400d1fdf:0x3ffb5030 0x400d0816:0x3ffb5050
```

Figure 12.5.5:OUTPUT OF ESP_ERROR_CHECK MARCO

12.6 ERROR HANDLING PATTERNS:

12.6.1 ATTEMPT TO RECOVER:

Retry the call, reinitialize the driver in use, reset an external peripheral which is not responding.

12.6.2 PROPAGATE THE ERROR TO THE CALLER:

Return the error codes that applies to the situation.

12.6.3 Convert to an Unrecoverable Error:

e.g., using ESP_ERROR_CHECK.
Do not simply write ESP_ERROR_CHECK everywhere if your intention is to write production-ready code. Determine a strategy that suits your application needs!

CHAPTER#13 CODE OF SUPPLEMENT LESSONS:

13.1 CODE OF CODING SYLE.H:

```
/*
 * codingstyle.h
 *
 *  Created on: Jul 10, 2024
```

```
 *      Author: Huzaifa
  */
// Include standard library header files e.g.:
#include<stdbool.h>
#include<stdio.h>
#include<string.h>
// Include ESP-IDF headers:
#include "esp_err.h"
#include "esp_log.h"
#include "nvs_flash.h"
// Include application headers:
#include "coding_style.h"
// Global netif objects for the WiFi station and access point
esp_netif_t*coding_style_esp_netif_sta =NULL;
esp_netif_t* coding_style_esp_netif_ap =NULL;
/**
 * Static function definitions require a description and of course, use the
"static" keyword.
 * If you want to call static functions before they are defined (meaning
call it  above this
 * definition), then you'll need to include prototypes for them.
 * @Note use the "file_name" prefix for the function name e.g.
"coding_style"
 * @param var_a describe the input parameter for var_a
 * @param var_b describe the input parameter for var_b
 * @param var_c describe the input parameter for var_c
 * @return ESP_OK, if successful
 */
static esp_err_t coding_style_static(int var_a)
{
        esp_err_t esp_err =ESP_OK;
        if(var_a<0)
        {
                esp_err=ESP_FAIL;
        }
```

```
	return esp_err;
}
// Public functions go below the private/static functions
// Note: a description of public funcitons is not necessary
// as it is already included in the header file.
esp_err_t coding_style_public(int var_a,int var_b,int var_c)
{
	int result=(var_a + var_b)-var_c;
	esp_err_t esp_err=coding_style_static(result);
	// Example use of ESP_ERROR_CHECK
	ESP_ERROR_CHECK(esp_err);
	return esp_err;
}
```

13.2 CODE OF CODE STYLE.C:

```
/*
 * codingstyle.c
 *
 *  Created on: Jul 10, 2024
 *      Author: Nobel
 */
#ifndef MAIN_CODING_STYLE_H
#define MAIN_CODING_STYLE_H
// Includes
#include "esp_netif.h"
// Macros and defines
#define CODING_STYLE_EXAMPLE_DEFINE "CODING STYLE"
// Externed variables
extern esp_netif_t* coding_style_esp_netif_sta;
extern esp_netif_t* coding_style_esp_netif_ap;
// Typedefs
/**
 * Provide a description for your typedef
 * @note Add notes if necessary.
```

```
 * @note postfix enums with "_e".
 */
typedef enum
{
	CODING_STYLE_FIRST=0,
	CODING_STYLE_SECOND,
	CODING_STYLE_THIRD,
}coding_style_message_e;
/**
 * Provide a description for your typedef
 * @note postfix structs with "_t".
 */
typedef struct
{
	coding_style_message_e msgID;
	void *param;
}coding_style_queue_message_t;
// Public function prototypes
/**
 * Public function defintions require a description in the header file only.
 * @Note use the "file_name" prefix for the function name e.g.
"coding_style"
 * @param var_a describe the input parameter for var_a
 * @param var_b describe the input parameter for var_b
 * @param var_c describe the input parameter for var_c
 * @return ESP_OK, if successful
 */
esp_err_t coding_style_public(int var_a,int var_b,int var_c);
#endif/* MAIN_CODING_STYLE_H_ */
```

13.3 CODE OF MAIN.C:

```
/**
 * Application entry point.
 */
```

```
#include "esp_log.h"
#include "nvs_flash.h"
#include "coding_style.h"
void app_main(void)
{
	// Initialize NVS
	esp_err_t ret=nvs_flash_init();
	if(ret==ESP_ERR_NVS_NO_FREE_PAGES || ret==
ESP_ERR_NVS_NEW_VERSION_FOUND)
	{
		ESP_ERROR_CHECK(nvs_flash_erase());
		ret=nvs_flash_init();
	}
	ESP_ERROR_CHECK(ret);
	// Example call to coding_style function
	coding_style_public(1,2,3);
}
```

CHAPTER#14 ESP-IDF COMPONENTS LIBRARY

14.1 ESP-IDF COMPONENTS LIBRARY:

The ESP-IDF (Espressif IoT Development Framework) provides a comprehensive set of libraries and components for developing ESP32 applications, covering a wide range of functionalities:

14.1.2 CORE TEMPERATURES:

14.1.2.1 FREERTOS:

Manages tasks and scheduling.

14.1.2.2 ESP SYSTEM:

Handles system initialization and clock configuration.

14.1.2.3 LOG:

Provides logging for debugging.

14.1.2.4 NWELIB:

Standard C library for embedded systems.

14.1.3 PERIPHERAL DEVICES:

14.1.3.1 GPIO:

Configures and controls GPIO pins.

14.1.3.2 ADC/DAC:

Manages analog-to-digital and digital-to-analog conversions.

14.1.3.3 12C, SPI, URAT:

Communication protocols.

14.1.3.4 PWM:

Controls motor speed, LEDs, etc.

14.1.3.5 TOUCH SENSOR:

Capacitive touch sensing.

14.1.4 NETWORKING:

14.1.4.1 WIFI:

Supports 802.11 b/g/n with various modes and security.

14.1.4.2 BLUETOOTH:

Classic and BLE support.

14.1.4.3 TCP/IP:

Lightweight IP stack (lwIP).

14.1.4.4 MQTT, HTTP/HTTPS:

Protocols for IoT communication.

14.1.5 STORAGE:

14.1.5.1 NVS:

Key-value storage for configuration data.

14.1.5 SPIFFS, FATFS:

File systems for flash memory and SD cards.

14.1.6 SECURITY:

14.1.6.1 mbedTLS:

Cryptographic library for SSL/TLS.

14.1.6.2 SECURE BOOT, FLASH ENCRYPTION:

Ensures secure firmware execution and data storage.

14.1.7 UTILITIES:

14.1.7.1 CONSOLE:

Command-line interface.

14.1.7.2 ESP TIMER:

High-resolution timers.

14.1.7.3 ESPRESSIF OTA:

Over-the-air firmware updates.

14.1.8 EXAMPLES:

14.1.8.1 EXAMPLE GPIO:

```
gpio_config_t io_conf = { .pin_bit_mask = GPIO_SEL_2, .mode = GPIO_MODE_OUTPUT };

gpio_config(&io_conf);

gpio_set_level(GPIO_NUM_2, 1);  // Set GPIO2 high
```

14.1.8.2 WIFI EXAMPLE:

```
esp_netif_init();

esp_event_loop_create_default();

esp_netif_create_default_wifi_sta();

wifi_config_t wifi_config = {.sta = {.ssid = "your_ssid", .password = "your_password"}};

esp_wifi_set_mode(WIFI_MODE_STA);

esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config);
```

esp_wifi_start();

CHAPTER#15 PROJECT STRUCTURE

15.1 APPLICATION PROJECT STRUCTURE:

Figure 15.1.6:PROJECT STRUCTURE OF ESPRESSIF IDF

CHAPTER#16 INTEGRATION STEPS PREVIEW

16.1 INTEGRATE LIBRARY:

Clone ESP-IDF components/sensors library. Adjust the top-level CMakeLists.txt file to include the sensor library component.

16.2 INTEGRATE EXAMPLES:

1: Either use one of mine or choose one from the library.

2: Adjust the CMakeLists.txt file in the main folder to include the example files.

3: Add the ‘Kconfig.projbuild’ file from example to the ‘main’ folder.

4: Adjust application files (main.c, tasks_common.h).

CHAPTER#17 EXAMPLE DEMONSTRATION

17.1 EXAMPLE DEMONSTRATION:

BME680 SENSORS AND MANY MORE.

CHAPTER#18 BME680 SENSOR

18.1 CODE OF BME680 SENSOR.H:

```
/*
 * bme680sensor.h
 *
 *  Created on: Jul 10, 2024
 *      Author: HuzaifaShahbaz
 */
#ifndef MAIN_BME_680_SENSOR_H_
#define MAIN_BME_680_SENSOR_H_
/**
 * Starts the BME680 task
 */
void BME680_task_start(void);
#endif /* MAIN_BME680_SENSOR_H_ */
```

18.2 CODE OF BME680 SENSOR.C:

```
/*
 * bme680_sensor.c
 *
 *  Created on: Jul 10, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include <stdio.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
```

```
#include <esp_system.h>
#include <bme680.h>
#include <string.h>
#include "bme680_sensor.h"
#include "tasks_common.h"
#define PORT 0
#if defined(CONFIG_EXAMPLE_I2C_ADDRESS_0)
#define ADDR_BME680 BME680_I2C_ADDR_0
#endif
#if defined(CONFIG_EXAMPLE_I2C_ADDRESS_1)
#define ADDR_BME680 BME680_I2C_ADDR_1
#endif
#ifndef APP_CPU_NUM
#define APP_CPU_NUM PRO_CPU_NUM
#endif
void bme680_test(void *pvParameters)
{
   bme680_t sensor;
   memset(&sensor, 0, sizeof(bme680_t));
   ESP_ERROR_CHECK(bme680_init_desc(&sensor,
ADDR_BME680, PORT, CONFIG_EXAMPLE_I2C_MASTER_SDA,
CONFIG_EXAMPLE_I2C_MASTER_SCL));
   // init the sensor
   ESP_ERROR_CHECK(bme680_init_sensor(&sensor));
   // Changes the oversampling rates to 4x oversampling for temperature
   // and 2x oversampling for humidity and Pressure measurement.
   bme680_set_oversampling_rates(&sensor, BME680_OSR_4X,
BME680_OSR_2X, BME680_OSR_2X);
   // Change the IIR filter size for temperature and pressure to 7.
   bme680_set_filter_size(&sensor, BME680_IIR_SIZE_7);
   // Change the heater profile 0 to 200 degree Celsius for 100 ms.
   bme680_set_heater_profile(&sensor, 0, 200, 100);
   bme680_use_heater_profile(&sensor, 0);
   // Set ambient temperature to 10 degree Celsius
   bme680_set_ambient_temperature(&sensor, 10);
```

```
    // as long as sensor configuration isn't changed, duration is constant
    uint32_t duration;
    bme680_get_measurement_duration(&sensor, &duration);
    TickType_t last_wakeup = xTaskGetTickCount();
    bme680_values_float_t values;
    while (1)
    {
        // trigger the sensor to start one TPHG measurement cycle
        if (bme680_force_measurement(&sensor) == ESP_OK)
        {
            // passive waiting until measurement results are available
            vTaskDelay(duration);
            // get the results and do something with them
            if (bme680_get_results_float(&sensor, &values) == ESP_OK)
                printf("BME680 Sensor: %.2f C, %.2f %%, %.2f hPa, %.2f
Ohm\n",
                        values.temperature, values.humidity, values.pressure,
values.gas_resistance);
        }
        // passive waiting until 1 second is over
        vTaskDelayUntil(&last_wakeup, pdMS_TO_TICKS(1000));
    }
}
void BME680_task_start(void)
{

    ESP_ERROR_CHECK(i2cdev_init());
    xTaskCreatePinnedToCore(bme680_test, "bme680_test",
BME680_TASK_STACK_SIZE, NULL, BME680_TASK_PRIORITY,
NULL, BME680_TASK_CORE_ID);
}
```

18.3 CODE OF BME680_SENSOR MAIN.C:

```
/**
```

```
*Application entry point
*/
#include<stdio.h>
#include<stdbool.h>
#include<unistd.h>
#include "bme680_sensor.h"
void app_main(void)
{
	//Start the BME680 task
	BME680_task_start();
}
```

18.4 CODE OF BME680_SENSOR TASKS_COMMON.H:

```
/*
 * tasks_common.h
 *
 *  Created on: Jul 10, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_TASKS_COMMON_H_
#define MAIN_TASKS_COMMON_H_
//BME680 TASK
#define BME680_TASK_STACK_SIZE      6144
#define BME680_TASK_PRIORTIY        3
#define BME680_TASK_CORE_ID         1
#endif /* MAIN_TASKS_COMMON_H_ */
```

CHAPTER#19 GIT REPOSITORY

19.1 Setting Up a Git Repository for ESP32 Development with ESP-IDF:

Using a Git repository to manage your ESP32 project ensures version control, collaboration, and organized code management. Here's a concise guide on setting up and using a Git repository for your ESP32 development:

19.1.1 INSTALL GIT:

Windows: Download and install Git from the official website.

macOS: Install via Homebrew:

sh Copy code brew install git
Linux: Install via package manager:

sh Copy code sudo apt-get install git

19.1.2 Initialize a Git Repository:

1: **Navigate to Your Project Directory**:

Copy code cd path/to/your/esp32-project

2: Initialize the Repository:
git init

3: Add ESP-IDF as a Submodule (if not already included):

git submodule add https://github.com/espressif/esp-idf.git

cd esp-idf

git submodule update --init --recursive

cd ..

19.1.3 Add and Commit Files:

1: Add Files to Staging Area:

git add.

2: Commit Files with a Message:

git commit -m "Initial commit"

19.1.4 CREATE A REMOTE REPOSITORY:

Create a new repository on GitHub, GitLab, or another Git hosting service.
Copy the repository URL.

CHAPTER#20 RGB APPLICATION STATUS LED

20.1 RGB LED STATUS COLORS:

Create a function for initializing the RGB LED settings per channel (R, G, and B), GPIO for each channel and timer configuration. Create function for setting the colors based on duty cycle for each channel. Create specific color functions to use as application status indicators. We will then test these functions in the main.c file, but we will use them later to indicate the WLAN application status (WiFi application started, HTTP server started, WiFi connected).

20.2 LED Control (LEDC) ESP-IDF:

Timer Configuration →Each timer counts upwards, and the bits defined, determines the count number before it resets, and frequency determines the amount of time it takes to count to that number. We will set members

of the ledc_timer_config_t structure and pass the configuration to, ledc_timer_config.

Channel Configuration→When defined, the GPIO pin that the PWM output signal appears on, is specified along with the timer that is associated with that Channel. We will set members of the ledc_channel_config_t struct and pass the configuration to, ledc_channel_config.

Set/Update Duty Cycle →The time duration within a period that the PWM output signal will be high before it goes low. We'll manipulate the duty cycle to create colors, by calling ledc_set_duty, then ledc_update_duty.

20.3 COLOR FUNCTIONS:

20.3.1 RGB Set Color Function:

We'll create a function that sets the color by updating the duty cycle per channel (R, G, B).

20.3.2 The Status Functions Call the Set Color Function at Specified Duty Cycles:

WiFi Started → RGB input values to our rgb_set_color function (255, 102, 255).
HTTP Server Started →RGB input values to rgb_set_color (204, 255, 51).
WiFi Connected →RGB input values to rgb_set_color (0, 255, 153).

20.4 RGB LEDS:

RGB LEDs come in different types e.g.
Check the datasheet for your RGB LED to verify the required resistor values.
Make connections with your jumper wires from the defined GPIOs (R, G, B) from the DevKit to the RGB LED.
Common Anode/Cathode → Make the connections based on the diagram (note the long-leg difference – ground vs. positive voltage.

CHAPTER#21 CODE OF RGB APPLICATION STATUS LED

21.1 CODE OF RGB_LED.H:

```
/*
 * rgb_led.h
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_RGB_LED_H_
#define MAIN_RGB_LED_H_
// RGB LED GPIOs
#define RGB_LED_RED_GPIO        21
#define RGB_LED_GREEN_GPIO          22
#define RGB_LED_BLUE_GPIO           23
// RGB LED color mix channels
#define RGB_LED_CHANNEL_NUM             3
// RGB LED configuration
typedef struct
{
        int channel;
        int gpio;
        int mode;
        int timer_index;
} ledc_info_t;
// ledc_info_t ledc_ch[RGB_LED_CHANNEL_NUM]; Move this
declaration to the top of rgb_led.c to avoid linker errors
/**
 * Color to indicate WiFi application has started.
 */
void rgb_led_wifi_app_started(void);
/**
```

```
 * Color to indicate HTTP server has started.
 */
void rgb_led_http_server_started(void);
/**
 * Color to indicate that the ESP32 is connected to an access point.
 */
void rgb_led_wifi_connected(void);
#endif /* MAIN_RGB_LED_H_ */
```

21.2 CODE OF RGB_LED.C:

```
/*
 * rgb_led.c
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include <stdbool.h>
#include "driver/ledc.h"
#include "rgb_led.h"
// RGB LED Configuration Array
ledc_info_t ledc_ch[RGB_LED_CHANNEL_NUM];
// handle for rgb_led_pwm_init
bool g_pwm_init_handle = false;
/**
 * Initializes the RGB LED settings per channel, including
 * the GPIO for each color, mode and timer configuration.
 */
static void rgb_led_pwm_init(void)
{
        int rgb_ch;
        // Red
        ledc_ch[0].channel          = LEDC_CHANNEL_0;
        ledc_ch[0].gpio             = RGB_LED_RED_GPIO;
        ledc_ch[0].mode             = LEDC_HIGH_SPEED_MODE;
```

```
	ledc_ch[0].timer_index	= LEDC_TIMER_0;
	// Green
	ledc_ch[1].channel		= LEDC_CHANNEL_1;
	ledc_ch[1].gpio		= RGB_LED_GREEN_GPIO;
	ledc_ch[1].mode		= LEDC_HIGH_SPEED_MODE;
	ledc_ch[1].timer_index	= LEDC_TIMER_0;
	// Blue
	ledc_ch[2].channel		= LEDC_CHANNEL_2;
	ledc_ch[2].gpio		= RGB_LED_BLUE_GPIO;
	ledc_ch[2].mode		= LEDC_HIGH_SPEED_MODE;
	ledc_ch[2].timer_index	= LEDC_TIMER_0;
	// Configure timer zero
	ledc_timer_config_t ledc_timer =
	{
		.duty_resolution = LEDC_TIMER_8_BIT,
		.freq_hz		= 100,
		.speed_mode		=
LEDC_HIGH_SPEED_MODE,
		.timer_num		= LEDC_TIMER_0
	};
	ledc_timer_config(&ledc_timer);
	// Configure channels
	for (rgb_ch = 0; rgb_ch < RGB_LED_CHANNEL_NUM;
rgb_ch++)
	{
		ledc_channel_config_t ledc_channel =
		{
			.channel	= ledc_ch[rgb_ch].channel,
			.duty		= 0,
			.hpoint		= 0,
			.gpio_num = ledc_ch[rgb_ch].gpio,
			.intr_type	= LEDC_INTR_DISABLE,
			.speed_mode = ledc_ch[rgb_ch].mode,
			.timer_sel	= ledc_ch[rgb_ch].timer_index,
		};
```

```
		ledc_channel_config(&ledc_channel);
	}
	g_pwm_init_handle = true;
}
/**
 * Sets the RGB color.
 */
static void rgb_led_set_color(uint8_t red, uint8_t green, uint8_t blue)
{
	// Value should be 0 - 255 for 8 bit number
	ledc_set_duty(ledc_ch[0].mode, ledc_ch[0].channel, red);
	ledc_update_duty(ledc_ch[0].mode, ledc_ch[0].channel);
	ledc_set_duty(ledc_ch[1].mode, ledc_ch[1].channel, green);
	ledc_update_duty(ledc_ch[1].mode, ledc_ch[1].channel);
	ledc_set_duty(ledc_ch[2].mode, ledc_ch[2].channel, blue);
	ledc_update_duty(ledc_ch[2].mode, ledc_ch[2].channel);
}
void rgb_led_wifi_app_started(void)
{
	if (g_pwm_init_handle == false)
	{
		rgb_led_pwm_init();
	}
	rgb_led_set_color(255, 102, 255);
}
void rgb_led_http_server_started(void)
{
	if (g_pwm_init_handle == false)
	{
		rgb_led_pwm_init();
	}
	rgb_led_set_color(204, 255, 51);
}

void rgb_led_wifi_connected(void)
```

```
{
        if (g_pwm_init_handle == false)
        {
                rgb_led_pwm_init();
        }
        rgb_led_set_color(0, 255, 153);
}
```

21.3 CODE OF MAIN.C:

```
#include "freertos/FreeRTOS.h"
#include "esp_wifi.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "nvs_flash.h"
#include "driver/gpio.h"
#include "rgb_led.h"
void app_main(void)
{
    while (true)
    {
         rgb_led_wifi_app_started();
        vTaskDelay(1000 / portTICK_PERIOD_MS);
        rgb_led_http_server_started();
        vTaskDelay(1000 / portTICK_PERIOD_MS);
        rgb_led_wifi_connected();
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

CHAPTER#22 WIFI APPLICATION

22.1 WIFI IMPLEMENTATION:

The ESP32 should start its Access Point so that other devices can connect to it. This enables users to access information e.g., sensor data, device info., connection status/information, user option to connect to and disconnect from an AP, display local time, etc. The WiFi application will start an HTTP Server (created in the next section), which will support a web page (to do all the stuff mentioned above). The application will contain a FreeRTOS task that accepts FreeRTOS Queue messages (xQueueCreate, xQueueSend and xQueueReceive) for event coordination. To be Implemented Later in the Course →connect the ESP32 using previously used (saved) credentials.

22.2 SIMPLIFEID OVERVIEW OF WIFI APPLICATION:

The simplified overview of WiFi application are:

22.2.1 Connecting Devices:

Becomes "station" of ESP32's SoftAP when connected. DHCP service from the ESP32's SoftAP will dynamically assign an IP to your device. Interact with the ESP32 via the web page.

Figure 22.1.7:CONNECTING DEVICES OF WIFI APPLICATION TO ESP32

22.2.2 ESP32 SOFTAP/STATION:

AP/STA combination mode. We assign an IP to the SOFTAP; the interface of the ESP32 is statically configured. DHCP server dynamically assigns an IP for connecting stations. We set a maximum number of stations allowed to connect.

Figure 22.2.8:ESP32 SOFTAP/STATION OF WIFI APPLICATIONS

22.2.3 ACCESS POINT:

When the ESP32 connects to an AP, local time can be obtained utilizing SNTP (if the AP is connected to the internet). DHCP service from the AP, will dynamically assign an IP to the ESP32 in our application.

Figure 22.2.9:ACCESS POINT OF CONNECTING DEVICES OF WIFI APPLICATION

22.3 Notable ESP-IDF Functions Used in This Section:

Define WiFi settings →Header file with SSID, Password, IP, Gateway, Netmask. etc.
Define WiFi FreeRTOS task →Use xTaskCreatePinnedToCore or xTaskCreate.
Create an event handler →Call esp_event_handler_instance_register.
Implement default configuration →Initialize TCP/IP stack using esp_netif_init and WiFi configuration by calling esp_wifi_init, esp_wifi_set_storage, and default configurations esp_netif_create_default_wifi_ap esp_netif_create_default_wifi_sta.
Define ESP32 SoftAP configuration →Define AP Settings wifi_config_t struct and static IP (in our case). Functions Used esp_netif_set_ip_info, esp_netif_dhcps_start, esp_wifi_set_mode, esp_wifi_set_config, esp_wifi_set_bandwidth, esp_wifi_set_ps.
Start WiFi (esp_wifi_start).

CHAPTER#23 CODE OF WIFI APP

23.1 CODE OF WIFI_APP.H:

```
/*
 * wifi_app.h
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_WIFI_APP_H_
#define MAIN_WIFI_APP_H_
#include "esp_netif.h"
// WiFi application settings
#define WIFI_AP_SSID                    "HUZAIFASHAHBAZ"
                // AP name
```

```
#define WIFI_AP_PASSWORD                "huzaf12345"
        // AP password
#define WIFI_AP_CHANNEL                 1
        // AP channel
#define WIFI_AP_SSID_HIDDEN             0
        // AP visibility
#define WIFI_AP_MAX_CONNECTIONS         5
        // AP max clients
#define WIFI_AP_BEACON_INTERVAL         100
        // AP beacon: 100 milliseconds recommended
#define WIFI_AP_IP                      "192.168.0.1"           // AP
default IP
#define WIFI_AP_GATEWAY                 "192.168.0.1"
        // AP default Gateway (should be the same as the IP)
#define WIFI_AP_NETMASK                 "255.255.255.0"
        // AP netmask
#define WIFI_AP_BANDWIDTH               WIFI_BW_HT20
        // AP bandwidth 20 MHz (40 MHz is the other option)
#define WIFI_STA_POWER_SAVE             WIFI_PS_NONE
        // Power save not used
#define MAX_SSID_LENGTH                 32
        // IEEE standard maximum
#define MAX_PASSWORD_LENGTH                     64
                // IEEE standard maximum
#define MAX_CONNECTION_RETRIES          5
        // Retry number on disconnect
// netif object for the Station and Access Point
extern esp_netif_t* esp_netif_sta;
extern esp_netif_t* esp_netif_ap;
/**
 * Message IDs for the WiFi application task
 * @note Expand this based on your application requirements.
 */
typedef enum wifi_app_message
{
```

```
	WIFI_APP_MSG_START_HTTP_SERVER = 0,
	WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER,
	WIFI_APP_MSG_STA_CONNECTED_GOT_IP,
} wifi_app_message_e;

/**
 * Structure for the message queue
 * @note Expand this based on application requirements e.g. add another
type and parameter as required
 */
typedef struct wifi_app_queue_message
{
	wifi_app_message_e msgID;
} wifi_app_queue_message_t;
/**
 * Sends a message to the queue
 * @param msgID message ID from the wifi_app_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue,
otherwise pdFALSE.
 * @note Expand the parameter list based on your requirements e.g. how
you've expanded the wifi_app_queue_message_t.
 */
BaseType_t wifi_app_send_message(wifi_app_message_e msgID);
/**
 * Starts the WiFi RTOS task
 */
void wifi_app_start(void);
#endif /* MAIN_WIFI_APP_H_ */
```

23.2 CODE OF WIFI_APP.C:

```
/*
 * wifi_app.c
 *
 *  Created on: Jul 11, 2024
```

```
 *      Author: HUZAIFASHAHBAZ
 */
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "freertos/task.h"
#include "esp_err.h"
#include "esp_log.h"
#include "esp_wifi.h"
#include "lwip/netdb.h"
#include "rgb_led.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG [] = "wifi_app";
// Queue handle used to manipulate the main queue of events
static QueueHandle_t wifi_app_queue_handle;
// netif objects for the station and access point
esp_netif_t* esp_netif_sta = NULL;
esp_netif_t* esp_netif_ap  = NULL;
/**
 * WiFi application event handler
 * @param arg data, aside from event data, that is passed to the handler
when it is called
 * @param event_base the base id of the event to register the handler for
 * @param event_id the id fo the event to register the handler for
 * @param event_data event data
 */
static void wifi_app_event_handler(void *arg, esp_event_base_t
event_base, int32_t event_id, void *event_data)
{
        if (event_base == WIFI_EVENT)
        {
                switch (event_id)
                {
                        case WIFI_EVENT_AP_START:
```

```
                        ESP_LOGI(TAG,
"WIFI_EVENT_AP_START");
                        break;
                case WIFI_EVENT_AP_STOP:
                        ESP_LOGI(TAG, "WIFI_EVENT_AP_STOP");
                        break;
                case WIFI_EVENT_AP_STACONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_AP_STACONNECTED");
                        break;

                case WIFI_EVENT_AP_STADISCONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_AP_STADISCONNECTED");
                        break;
                case WIFI_EVENT_STA_START:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_START");
                        break;

                case WIFI_EVENT_STA_CONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_CONNECTED");
                        break;
                case WIFI_EVENT_STA_DISCONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_DISCONNECTED");
                        break;
            }
        }
        else if (event_base == IP_EVENT)
        {
            switch (event_id)
            {
                case IP_EVENT_STA_GOT_IP:
```

```
                        ESP_LOGI(TAG, "IP_EVENT_STA_GOT_IP");
                        break;
                }
        }
}

/**
 * Initializes the WiFi application event handler for WiFi and IP events.
 */
static void wifi_app_event_handler_init(void)
{
        // Event loop for the WiFi driver
        ESP_ERROR_CHECK(esp_event_loop_create_default());

        // Event handler for the connection
        esp_event_handler_instance_t instance_wifi_event;
        esp_event_handler_instance_t instance_ip_event;
        ESP_ERROR_CHECK(esp_event_handler_instance_register(WIF
I_EVENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_wifi_event));
        ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_E
VENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_ip_event));
}

/**
 * Initializes the TCP stack and default WiFi configuration.
 */
static void wifi_app_default_wifi_init(void)
{
        // Initialize the TCP stack
        ESP_ERROR_CHECK(esp_netif_init());
        // Default WiFi config - operations must be in this order!
        wifi_init_config_t wifi_init_config =
WIFI_INIT_CONFIG_DEFAULT();
```

```
	ESP_ERROR_CHECK(esp_wifi_init(&wifi_init_config));
	ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_R
AM));
	esp_netif_sta = esp_netif_create_default_wifi_sta();
	esp_netif_ap = esp_netif_create_default_wifi_ap();
}
/**
 * Configures the WiFi access point settings and assigns the static IP to
the SoftAP.
 */
static void wifi_app_soft_ap_config(void)
{
	// SoftAP - WiFi access point configuration
	wifi_config_t ap_config =
	{
		.ap = {
				.ssid = WIFI_AP_SSID,
				.ssid_len = strlen(WIFI_AP_SSID),
				.password = WIFI_AP_PASSWORD,
				.channel = WIFI_AP_CHANNEL,
				.ssid_hidden = WIFI_AP_SSID_HIDDEN,
				.authmode = WIFI_AUTH_WPA2_PSK,
				.max_connection =
WIFI_AP_MAX_CONNECTIONS,
				.beacon_interval =
WIFI_AP_BEACON_INTERVAL,
		},
	};
	// Configure DHCP for the AP
	esp_netif_ip_info_t ap_ip_info;
	memset(&ap_ip_info, 0x00, sizeof(ap_ip_info));
	esp_netif_dhcps_stop(esp_netif_ap);				///>
must call this first
	inet_pton(AF_INET, WIFI_AP_IP, &ap_ip_info.ip);		///>
Assign access point's static IP, GW, and netmask
```

```
	inet_pton(AF_INET, WIFI_AP_GATEWAY, &ap_ip_info.gw);
	inet_pton(AF_INET, WIFI_AP_NETMASK,
&ap_ip_info.netmask);
	ESP_ERROR_CHECK(esp_netif_set_ip_info(esp_netif_ap,
&ap_ip_info));		///> Statically configure the network
interface
	ESP_ERROR_CHECK(esp_netif_dhcps_start(esp_netif_ap));
				///> Start the AP DHCP server (for
connecting stations e.g. your mobile device)
	ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_APST
A));				///> Setting the mode as Access Point /
Station Mode
	ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP,
&ap_config));		///> Set our configuration
	ESP_ERROR_CHECK(esp_wifi_set_bandwidth(WIFI_IF_AP,
WIFI_AP_BANDWIDTH));		///> Our default bandwidth 20
MHz
	ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_STA_POWER_SA
VE));				///> Power save set to "NONE"
}
/**
 * Main task for the WiFi application
 * @param pvParameters parameter which can be passed to the task
 */
static void wifi_app_task(void *pvParameters)
{
	wifi_app_queue_message_t msg;
	// Initialize the event handler
	wifi_app_event_handler_init();
	// Initialize the TCP/IP stack and WiFi config
	wifi_app_default_wifi_init();
	// SoftAP config
	wifi_app_soft_ap_config();
	// Start WiFi
	ESP_ERROR_CHECK(esp_wifi_start());
```

```
	// Send first event message
	wifi_app_send_message(WIFI_APP_MSG_START_HTTP_SERV
ER);
	for (;;)
	{
		if (xQueueReceive(wifi_app_queue_handle, &msg,
portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case
WIFI_APP_MSG_START_HTTP_SERVER:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_START_HTTP_SERVER");
//					http_server_start();
					rgb_led_http_server_started();
					break;
				case
WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER");
					break;
				case
WIFI_APP_MSG_STA_CONNECTED_GOT_IP:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_CONNECTED_GOT_IP");
					rgb_led_wifi_connected();
					break;
				default:
					break;
			}
		}
	}
}
BaseType_t wifi_app_send_message(wifi_app_message_e msgID)
```

```
{
	wifi_app_queue_message_t msg;
	msg.msgID = msgID;
	return xQueueSend(wifi_app_queue_handle, &msg,
portMAX_DELAY);
}
void wifi_app_start(void)
{
	ESP_LOGI(TAG, "STARTING WIFI APPLICATION");
	// Start WiFi started LED
	rgb_led_wifi_app_started();
	// Disable default WiFi logging messages
	esp_log_level_set("wifi", ESP_LOG_NONE);
	// Create message queue
	wifi_app_queue_handle = xQueueCreate(3,
sizeof(wifi_app_queue_message_t));
	// Start the WiFi application task
	xTaskCreatePinnedToCore(&wifi_app_task, "wifi_app_task",
WIFI_APP_TASK_STACK_SIZE, NULL,
WIFI_APP_TASK_PRIORITY, NULL,
WIFI_APP_TASK_CORE_ID);
}
```

23.3 CODE OF MAIN.C:

```
/**
 * Application entry point.
 */
#include "nvs_flash.h"
#include "wifi_app.h"
void app_main(void)
{
   // Initialize NVS
	esp_err_t ret = nvs_flash_init();
```

```
	if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND)
	{
		ESP_ERROR_CHECK(nvs_flash_erase());
		ret = nvs_flash_init();
	}
	ESP_ERROR_CHECK(ret);
	// Start Wifi
	wifi_app_start();
}
```

23.4 CODE OF TASKSCOMMON.H:

```
/*
 * tasks_common.h
 *
 *  Created on: Jul 11, 2024
 *      Author:HUZAIFASHAHBAZ
 */
#ifndef MAIN_TASKS_COMMON_H_
#define MAIN_TASKS_COMMON_H_
// WiFi application task
#define WIFI_APP_TASK_STACK_SIZE		4096
#define WIFI_APP_TASK_PRIORITY		5
#define WIFI_APP_TASK_CORE_ID		0
#endif /* MAIN_TASKS_COMMON_H_ */
```

CHAPTER#24 HTTP SERVER

24.1 HTTP SERVER REQUIREMENTS:

The HTTP Server will support the web page files (.html, .css and .js).
It will also support OTA (Over the Air) Firmware Updates.
Additionally, support for DHT22 Temperature and Humidity Sensor

readings for display on the web page will be added.
The HTTP server will be able to respond to Connection and Disconnection buttons on the web page e.g., by entering SSID & Password into text fields and clicking connect and disconnect for removing a connection.
The web server will also handle sending connection information (SSID and IP, Gateway, Netmask) about the active connection to the web page.
We will also send the ESP32's assigned SSID to the web page.

24.2 WEB PAGES FILES:

index.html →HTML markup that we will expand upon (it already includes OTA
Update markup).
app.css →Used to style for the index.html document that we will expand (already includes OTA Update styling).
app.js →Javascript file that we will spend time expanding throughout the course.
The template provided already includes the OTA Update functions.
favicon.ico →Icon that gets displayed in the address bar of the browser.
jquery-3.3.1.min.js →JavaScript library.

24.3 CREATING AN HTTP SERVER USING THE ESP-IDF:

Embed Binary Data (index.html, app.css and code.js) →Embedding Binary Data.
Create HTTP Server start and stop functions →create a wrappers around these.
Create the default HTTP server configuration and adjust to our needs →Create the struct httpd_config_t and call httpd_start.
Register the URI handlers →The first lesson, this will include the .html, .css and.js files) and call httpd_register_uri_handler for each.

CHAPTER#25 CODES OF HTTP SERVER

25.1 CODE OF HTTP SERVER.C:

```
/*
 * http_server.c
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include "esp_http_server.h"
#include "esp_log.h"
#include "http_server.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG[] = "http_server";
// HTTP server task handle
static httpd_handle_t http_server_handle = NULL;
// HTTP server monitor task handle
static TaskHandle_t task_http_server_monitor = NULL;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t http_server_monitor_queue_handle;
// Embedded files: JQuery, index.html, app.css, app.js and favicon.ico
files
extern const uint8_t jquery_3_3_1_min_js_start[]
        asm("_binary_jquery_3_3_1_min_js_start");
extern const uint8_t jquery_3_3_1_min_js_end[]
        asm("_binary_jquery_3_3_1_min_js_end");
extern const uint8_t index_html_start[]
        asm("_binary_index_html_start");
extern const uint8_t index_html_end[]
        asm("_binary_index_html_end");
extern const uint8_t app_css_start[]
        asm("_binary_app_css_start");
```

```
extern const uint8_t app_css_end[]
	asm("_binary_app_css_end");
extern const uint8_t app_js_start[]
	asm("_binary_app_js_start");
extern const uint8_t app_js_end[]
	asm("_binary_app_js_end");
extern const uint8_t favicon_ico_start[]
	asm("_binary_favicon_ico_start");
extern const uint8_t favicon_ico_end[]
	asm("_binary_favicon_ico_end");
/**
 * HTTP server monitor task used to track events of the HTTP server
 * @param pvParameters parameter which can be passed to the task.
 */
static void http_server_monitor(void *parameter)
{
	http_server_queue_message_t msg;
	for (;;)
	{
		if (xQueueReceive(http_server_monitor_queue_handle,
&msg, portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case HTTP_MSG_WIFI_CONNECT_INIT:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_INIT");
					break;
				case HTTP_MSG_WIFI_CONNECT_SUCCESS:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_SUCCESS");
					break;
				case HTTP_MSG_WIFI_CONNECT_FAIL:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_FAIL");
```

```
                                break;
                        case
HTTP_MSG_OTA_UPDATE_SUCCESSFUL:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_SUCCESSFUL");
                                break;
                        case HTTP_MSG_OTA_UPDATE_FAILED:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_FAILED");
                                break;
                        case HTTP_MSG_OTA_UPATE_INITIALIZED:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPATE_INITIALIZED");
                                break;
                        default:
                                break;
                }
            }
        }
}
/**
 * Jquery get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_jquery_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "Jquery requested");
        httpd_resp_set_type(req, "application/javascript");
        httpd_resp_send(req, (const char *)jquery_3_3_1_min_js_start,
jquery_3_3_1_min_js_end - jquery_3_3_1_min_js_start);
        return ESP_OK;
}
/**
 * Sends the index.html page.
```

```
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_index_html_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "index.html requested");
	httpd_resp_set_type(req, "text/html");
	httpd_resp_send(req, (const char *)index_html_start,
index_html_end - index_html_start);
	return ESP_OK;
}
/**
 * app.css get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_css_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.css requested");
	httpd_resp_set_type(req, "text/css");
	httpd_resp_send(req, (const char *)app_css_start, app_css_end -
app_css_start);
	return ESP_OK;
}
/**
 * app.js get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_js_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.js requested");

	httpd_resp_set_type(req, "application/javascript");
```

```
	httpd_resp_send(req, (const char *)app_js_start, app_js_end -
app_js_start);

	return ESP_OK;
}
/**
 * Sends the .ico (icon) file when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_favicon_ico_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "favicon.ico requested");
	httpd_resp_set_type(req, "image/x-icon");
	httpd_resp_send(req, (const char *)favicon_ico_start,
favicon_ico_end - favicon_ico_start);
	return ESP_OK;
}
/**
 * Sets up the default httpd server configuration.
 * @return http server instance handle if successful, NULL otherwise.
 */
static httpd_handle_t http_server_configure(void)
{
	// Generate the default configuration
	httpd_config_t config = HTTPD_DEFAULT_CONFIG();
	// Create HTTP server monitor task
	xTaskCreatePinnedToCore(&http_server_monitor,
"http_server_monitor", HTTP_SERVER_MONITOR_STACK_SIZE,
NULL, HTTP_SERVER_MONITOR_PRIORITY,
&task_http_server_monitor, HTTP_SERVER_MONITOR_CORE_ID);
	// Create the message queue
	http_server_monitor_queue_handle = xQueueCreate(3,
sizeof(http_server_queue_message_t));
	// The core that the HTTP server will run on
```

```
	config.core_id = HTTP_SERVER_TASK_CORE_ID;
	// Adjust the default priority to 1 less than the wifi application task
	config.task_priority = HTTP_SERVER_TASK_PRIORITY;
	// Bump up the stack size (default is 4096)
	config.stack_size = HTTP_SERVER_TASK_STACK_SIZE;
	// Increase uri handlers
	config.max_uri_handlers = 20;
	// Increase the timeout limits
	config.recv_wait_timeout = 10;
	config.send_wait_timeout = 10;
	ESP_LOGI(TAG,
			"http_server_configure: Starting server on port: '%d'
with task priority: '%d'",
			config.server_port,
			config.task_priority);
	// Start the httpd server
	if (httpd_start(&http_server_handle, &config) == ESP_OK)
	{
		ESP_LOGI(TAG, "http_server_configure: Registering URI
handlers");
		// register query handler
		httpd_uri_t jquery_js = {
				.uri = "/jquery-3.3.1.min.js",
				.method = HTTP_GET,
				.handler = http_server_jquery_handler,
				.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &jquery_js);
		// register index.html handler
		httpd_uri_t index_html = {
				.uri = "/",
				.method = HTTP_GET,
				.handler = http_server_index_html_handler,
				.user_ctx = NULL
		};
```

```
		httpd_register_uri_handler(http_server_handle,
&index_html);
		// register app.css handler
		httpd_uri_t app_css = {
			.uri = "/app.css",
			.method = HTTP_GET,
			.handler = http_server_app_css_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &app_css);
		// register app.js handler
		httpd_uri_t app_js = {
			.uri = "/app.js",
			.method = HTTP_GET,
			.handler = http_server_app_js_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &app_js);
		// register favicon.ico handler
		httpd_uri_t favicon_ico = {
			.uri = "/favicon.ico",
			.method = HTTP_GET,
			.handler = http_server_favicon_ico_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&favicon_ico);

		return http_server_handle;
	}

	return NULL;
}
void http_server_start(void)
{
```

```
	if (http_server_handle == NULL)
	{
		http_server_handle = http_server_configure();
	}
}
void http_server_stop(void)
{
	if (http_server_handle)
	{
		httpd_stop(http_server_handle);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP
server");
		http_server_handle = NULL;
	}
	if (task_http_server_monitor)
	{
		vTaskDelete(task_http_server_monitor);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP server
monitor");
		task_http_server_monitor = NULL;
	}
}
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID)
{
	http_server_queue_message_t msg;
	msg.msgID = msgID;
	return xQueueSend(http_server_monitor_queue_handle, &msg,
portMAX_DELAY);
}
```

25.2 CODE OF HTTP SERVER.H:

```
/*
 * http_server.h
```

```
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_HTTP_SERVER_H_
#define MAIN_HTTP_SERVER_H_
/**
 * Messages for the HTTP monitor
 */
typedef enum http_server_message
{
        HTTP_MSG_WIFI_CONNECT_INIT = 0,
        HTTP_MSG_WIFI_CONNECT_SUCCESS,
        HTTP_MSG_WIFI_CONNECT_FAIL,
        HTTP_MSG_OTA_UPDATE_SUCCESSFUL,
        HTTP_MSG_OTA_UPDATE_FAILED,
        HTTP_MSG_OTA_UPATE_INITIALIZED,
} http_server_message_e;
/**
 * Structure for the message queue
 */
typedef struct http_server_queue_message
{
        http_server_message_e msgID;
} http_server_queue_message_t;
/**
 * Sends a message to the queue
 * @param msgID message ID from the http_server_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue,
otherwise pdFALSE.
 * @note Expand the parameter list based on your requirements e.g. how
you've expanded the http_server_queue_message_t.
 */
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID);
```

```
/**
 * Starts the HTTP server.
 */
void http_server_start(void);
/**
 * Stops the HTTP server.
 */
void http_server_stop(void);
#endif /* MAIN_HTTP_SERVER_H_ */
```

25.3 CODE OF TASK_COMMON.H:

```
/*
 * tasks_common.h
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_TASKS_COMMON_H_
#define MAIN_TASKS_COMMON_H_
// WiFi application task
#define WIFI_APP_TASK_STACK_SIZE                    4096
#define WIFI_APP_TASK_PRIORITY                      5
#define WIFI_APP_TASK_CORE_ID                       0
// HTTP Server task
#define HTTP_SERVER_TASK_STACK_SIZE                 8192
#define HTTP_SERVER_TASK_PRIORITY                   4
#define HTTP_SERVER_TASK_CORE_ID                    0
// HTTP Server Monitor task
#define HTTP_SERVER_MONITOR_STACK_SIZE              4096
#define HTTP_SERVER_MONITOR_PRIORITY                3
#define HTTP_SERVER_MONITOR_CORE_ID                 0
#endif /* MAIN_TASKS_COMMON_H_ */
```

25.3 CODE OF WIFI_APP.C:

```
/*
```

```
 * tasks_common.h
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "freertos/task.h"
#include "esp_err.h"
#include "esp_log.h"
#include "esp_wifi.h"
#include "lwip/netdb.h"
#include "http_server.h"
#include "rgb_led.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG [] = "wifi_app";
// Queue handle used to manipulate the main queue of events
static QueueHandle_t wifi_app_queue_handle;
// netif objects for the station and access point
esp_netif_t* esp_netif_sta = NULL;
esp_netif_t* esp_netif_ap  = NULL;
/**
 * WiFi application event handler
 * @param arg data, aside from event data, that is passed to the handler
when it is called
 * @param event_base the base id of the event to register the handler for
 * @param event_id the id fo the event to register the handler for
 * @param event_data event data
 */
static void wifi_app_event_handler(void *arg, esp_event_base_t
event_base, int32_t event_id, void *event_data)
{
        if (event_base == WIFI_EVENT)
```

```
        {
                switch (event_id)
                {
                        case WIFI_EVENT_AP_START:
                                ESP_LOGI(TAG,
"WIFI_EVENT_AP_START");
                                break;
                        case WIFI_EVENT_AP_STOP:
                                ESP_LOGI(TAG, "WIFI_EVENT_AP_STOP");
                                break;
                        case WIFI_EVENT_AP_STACONNECTED:
                                ESP_LOGI(TAG,
"WIFI_EVENT_AP_STACONNECTED");
                                break;
                        case WIFI_EVENT_AP_STADISCONNECTED:
                                ESP_LOGI(TAG,
"WIFI_EVENT_AP_STADISCONNECTED");
                                break;
                        case WIFI_EVENT_STA_START:
                                ESP_LOGI(TAG,
"WIFI_EVENT_STA_START");
                                break;
                        case WIFI_EVENT_STA_CONNECTED:
                                ESP_LOGI(TAG,
"WIFI_EVENT_STA_CONNECTED");
                                break;
                        case WIFI_EVENT_STA_DISCONNECTED:
                                ESP_LOGI(TAG,
"WIFI_EVENT_STA_DISCONNECTED");
                                break;
                }
        }
        else if (event_base == IP_EVENT)
        {
                switch (event_id)
```

```
            {
                case IP_EVENT_STA_GOT_IP:
                    ESP_LOGI(TAG, "IP_EVENT_STA_GOT_IP");
                    break;
            }
        }
}
/**
 * Initializes the WiFi application event handler for WiFi and IP events.
 */
static void wifi_app_event_handler_init(void)
{
        // Event loop for the WiFi driver
        ESP_ERROR_CHECK(esp_event_loop_create_default());
        // Event handler for the connection
        esp_event_handler_instance_t instance_wifi_event;
        esp_event_handler_instance_t instance_ip_event;
        ESP_ERROR_CHECK(esp_event_handler_instance_register(WIF
I_EVENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_wifi_event));
        ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_E
VENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_ip_event));
}
/**
 * Initializes the TCP stack and default WiFi configuration.
 */
static void wifi_app_default_wifi_init(void)
{
        // Initialize the TCP stack
        ESP_ERROR_CHECK(esp_netif_init());
        // Default WiFi config - operations must be in this order!
        wifi_init_config_t wifi_init_config =
WIFI_INIT_CONFIG_DEFAULT();
        ESP_ERROR_CHECK(esp_wifi_init(&wifi_init_config));
```

```
	ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_R
AM));
	esp_netif_sta = esp_netif_create_default_wifi_sta();
	esp_netif_ap = esp_netif_create_default_wifi_ap();
}
/**
 * Configures the WiFi access point settings and assigns the static IP to
the SoftAP.
 */
static void wifi_app_soft_ap_config(void)
{
	// SoftAP - WiFi access point configuration
	wifi_config_t ap_config =
	{
		.ap = {
				.ssid = WIFI_AP_SSID,
				.ssid_len = strlen(WIFI_AP_SSID),
				.password = WIFI_AP_PASSWORD,
				.channel = WIFI_AP_CHANNEL,
				.ssid_hidden = WIFI_AP_SSID_HIDDEN,
				.authmode = WIFI_AUTH_WPA2_PSK,
				.max_connection =
WIFI_AP_MAX_CONNECTIONS,
				.beacon_interval =
WIFI_AP_BEACON_INTERVAL,
		},
	};
	// Configure DHCP for the AP
	esp_netif_ip_info_t ap_ip_info;
	memset(&ap_ip_info, 0x00, sizeof(ap_ip_info));
	esp_netif_dhcps_stop(esp_netif_ap);				///>
must call this first
	inet_pton(AF_INET, WIFI_AP_IP, &ap_ip_info.ip);		///>
Assign access point's static IP, GW, and netmask
	inet_pton(AF_INET, WIFI_AP_GATEWAY, &ap_ip_info.gw);
```

```
	inet_pton(AF_INET, WIFI_AP_NETMASK,
&ap_ip_info.netmask);
	ESP_ERROR_CHECK(esp_netif_set_ip_info(esp_netif_ap,
&ap_ip_info));		///> Statically configure the network
interface
	ESP_ERROR_CHECK(esp_netif_dhcps_start(esp_netif_ap));
				///> Start the AP DHCP server (for
connecting stations e.g. your mobile device)
	ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_APST
A));				///> Setting the mode as Access Point /
Station Mode
	ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP,
&ap_config));		///> Set our configuration
	ESP_ERROR_CHECK(esp_wifi_set_bandwidth(WIFI_IF_AP,
WIFI_AP_BANDWIDTH));		///> Our default bandwidth 20
MHz
	ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_STA_POWER_SA
VE));				///> Power save set to "NONE"
}
/**
 * Main task for the WiFi application
 * @param pvParameters parameter which can be passed to the task
 */
static void wifi_app_task(void *pvParameters)
{
	wifi_app_queue_message_t msg;
	// Initialize the event handler
	wifi_app_event_handler_init();
	// Initialize the TCP/IP stack and WiFi config
	wifi_app_default_wifi_init();
	// SoftAP config
	wifi_app_soft_ap_config();
	// Start WiFi
	ESP_ERROR_CHECK(esp_wifi_start());
	// Send first event message
```

```
	wifi_app_send_message(WIFI_APP_MSG_START_HTTP_SERV
ER);
	for (;;)
	{
		if (xQueueReceive(wifi_app_queue_handle, &msg,
portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case
WIFI_APP_MSG_START_HTTP_SERVER:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_START_HTTP_SERVER");
					http_server_start();
					rgb_led_http_server_started();
					break;
				case
WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER");
					break;
				case
WIFI_APP_MSG_STA_CONNECTED_GOT_IP:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_CONNECTED_GOT_IP");
					rgb_led_wifi_connected();
					break;
				default:
					break;

			}
		}
	}
}
BaseType_t wifi_app_send_message(wifi_app_message_e msgID)
```

```
{
	wifi_app_queue_message_t msg;
	msg.msgID = msgID;
	return xQueueSend(wifi_app_queue_handle, &msg,
portMAX_DELAY);
}
void wifi_app_start(void)
{
	ESP_LOGI(TAG, "STARTING WIFI APPLICATION");
	// Start WiFi started LED
	rgb_led_wifi_app_started();
	// Disable default WiFi logging messages
	esp_log_level_set("wifi", ESP_LOG_NONE);
	// Create message queue
	wifi_app_queue_handle = xQueueCreate(3,
sizeof(wifi_app_queue_message_t));
	// Start the WiFi application task
	xTaskCreatePinnedToCore(&wifi_app_task, "wifi_app_task",
WIFI_APP_TASK_STACK_SIZE, NULL,
WIFI_APP_TASK_PRIORITY, NULL,
WIFI_APP_TASK_CORE_ID);
}
```

25.4 CODE OF APP.CSS:

```
body {
   background-color: #f1f1f1;
   font-family: HelveticaNeueRegular, HelveticaNeue-Regular,
"Helvetica Neue Regular", HelveticaNeue, "Helvetica Neue", Helvetica,
Arial;
   color: #0272B7;
}

h1 {
   display: block;
```

```
    text-align: center;
    margin: 0;
    padding: 10px;
    font-size: 1.7em;
    color: #1d3557;
}

h2 {
    font-size: 1.4em;
    color: #1d3557;
}

h3 {
    font-size: 1.2em;
    color: #1d3557;
}

h4 {
    font-size: 1em;
    color: #1d3557;
}

#latest_firmware, #latest_firmware_label {
    display:inline;
}
#latest_firmware {
    color: #1d3557;
}

.buttons {
    padding: 5px;
}

input[type="button"] {
    font-size: 12px;
```

```
    border: 0;
    line-height: 2;
    padding: 0 5px;
    text-align: center;
    color: #ffffff;
    text-shadow: 1px 1px 1px #000;
    border-radius: 10px;
    background-color: RGBA(106,125,144,0.35);
    background-image: linear-gradient(to top left,
                                      rgba(0, 0, 0, .2),
                                      rgba(0, 0, 0, .2) 30%,
                                      rgba(0, 0, 0, 0));
    box-shadow: inset 2px 2px 3px rgba(255, 255, 255, .6),
                inset -2px -2px 3px rgba(0, 0, 0, .6);
}

input[type="button"]:hover {
    background-color: RGBA(112,164,222,0.68);
}

input[type="button"]:active {
    box-shadow: inset -2px -2px 3px rgba(255, 255, 255, .6),
                inset 2px 2px 3px rgba(0, 0, 0, .6);
}
```

25.4 CODE OF APP.JS:

```
/**
 * Add gobals here
 */
var seconds     = null;
var otaTimerVar =  null;

/**
```

```
 * Initialize functions here.
 */
$(document).ready(function(){
   getUpdateStatus();
});

/**
 * Gets file name and size for display on the web page.
 */
function getFileInfo()
{
   var x = document.getElementById("selected_file");
   var file = x.files[0];

   document.getElementById("file_info").innerHTML = "<h4>File: " +
file.name + "<br>" + "Size: " + file.size + " bytes</h4>";
}

/**
 * Handles the firmware update.
 */
function updateFirmware()
{
   // Form Data
   var formData = new FormData();
   var fileSelect = document.getElementById("selected_file");

   if (fileSelect.files && fileSelect.files.length == 1)
   {
      var file = fileSelect.files[0];
      formData.set("file", file, file.name);
      document.getElementById("ota_update_status").innerHTML =
"Uploading " + file.name + ", Firmware Update in Progress...";

      // Http Request
```

```
        var request = new XMLHttpRequest();

        request.upload.addEventListener("progress", updateProgress);
        request.open('POST', "/OTAupdate");
        request.responseType = "blob";
        request.send(formData);
    }
    else
    {
        window.alert('Select A File First')
    }
}

/**
 * Progress on transfers from the server to the client (downloads).
 */
function updateProgress(oEvent)
{
    if (oEvent.lengthComputable)
    {
        getUpdateStatus();
    }
    else
    {
        window.alert('total size is unknown')
    }
}

/**
 * Posts the firmware udpate status.
 */
function getUpdateStatus()
{
    var xhr = new XMLHttpRequest();
    var requestURL = "/OTAstatus";
```

```
    xhr.open('POST', requestURL, false);
    xhr.send('ota_update_status');

    if (xhr.readyState == 4 && xhr.status == 200)
    {
        var response = JSON.parse(xhr.responseText);

        document.getElementById("latest_firmware").innerHTML =
response.compile_date + " - " + response.compile_time

        // If flashing was complete it will return a 1, else -1
        // A return of 0 is just for information on the Latest Firmware
request
        if (response.ota_update_status == 1)
        {
            // Set the countdown timer time
            seconds = 10;
            // Start the countdown timer
            otaRebootTimer();
        }
        else if (response.ota_update_status == -1)
        {
            document.getElementById("ota_update_status").innerHTML =
"!!! Upload Error !!!";
        }
    }
}

/**
 * Displays the reboot countdown.
 */
function otaRebootTimer()
{
```

```
    document.getElementById("ota_update_status").innerHTML = "OTA
Firmware Update Complete. This page will close shortly, Rebooting in:
" + seconds;

    if (--seconds == 0)
    {
        clearTimeout(otaTimerVar);
        window.location.reload();
    }
    else
    {
        otaTimerVar = setTimeout(otaRebootTimer, 1000);
    }
}
```

25.4 CODE OF INDEX.HTML:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8"/>
        <meta name="viewport" content="width=device-width, initial-
scale=1.0, user-scalable=no">
        <meta name="apple-mobile-web-app-capable" content="yes" />
        <script src='jquery-3.3.1.min.js'></script>
        <link rel="stylesheet" href="app.css">
        <script async src="app.js"></script>
        <title>ESP32 Udemy Course</title>
    </head>
    <body>
    <header>
        <h1>ESP32 Application Development</h1>
    </header>
    <div id="OTA">
    <h2>ESP32 Firmware Update</h2>
```

```
        <label id="latest_firmware_label">Latest Firmware: </label>
        <div id="latest_firmware"></div>
        <input type="file" id="selected_file" accept=".bin" style="display:
none;" onchange="getFileInfo()" />
        <div class="buttons">
            <input type="button" value="Select File"
onclick="document.getElementById('selected_file').click();" />
            <input type="button" value="Update Firmware"
onclick="updateFirmware()" />
        </div>
        <h4 id="file_info"></h4>
        <h4 id="ota_update_status"></h4>
    </div>
    <hr>
    </body>
<html>
```

CHAPTER#26 OTA FIRMWARE UPDATES

26.1 ABOUT OTA UPDATES:

Allows a device to update itself based on data received (for example, over WiFi) while the normal firmware is running.
Requires configurating a Partition Table of the device with at least two "OTA app slots" (i.e., ota_0 and ota_1) and an "OTA Data Partition".
The OTA operation functions write a new app firmware image to whichever OTA app slot that is currently not selected for booting...
Once the image is verified, the OTA Data partition is updated to specify that this image should be used for the next boot.

26.2 ABOUT PARTITION TABLES:

A single ESP32's flash can contain multiple apps, as well as many kinds of data (calibration data, filesystems, parameter storage, etc.), for this reason a partition table is flashed to 0x8000 (default offset) in the flash.

Each entry in the partition table has a name (label), type (app, data, etc.),subtype and the offset in flash where the partition is loaded.
The simplest way to use the partition table is to open the project configuration (sdkconfig in Eclipse) and choose “Factory app, two OTA definitions” (as we have already done in the project configuration.
The table is located: esp-idf\components\partition_table.

26.3 MORE ABOUT PARTITION TABLES:

```
# ESP-IDF Partition Table
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x9000,  0x4000,
otadata,  data, ota,     0xd000,  0x2000,
phy_init, data, phy,     0xf000,  0x1000,
factory,  app,  factory, 0x10000,  1M,
ota_0,    app,  ota_0,   0x110000, 1M,
ota_1,    app,  ota_1,   0x210000, 1M,
```

Figure 26.3.10: Description of “Factory app, two OTA definitions” configuration

There are now three app partition definitions. The type of the factory app (at 0x10000) and the next two “OTA” apps are all set to “app”, but their subtypes are different. There is also a new “otadata” slot, which holds the data for OTA updates. The bootloader consults this data in order to know which app to execute. If “otadata” is empty, it will execute the factory app.
• Note: you can create custom partition tables to suite your need

26.4 OTA UPDATE REQUIREMENTS:

The User starts the OTA update by uploading a .bin file over the web page.
The OTA Firmware Update is performed, and the web page displays the status.
Once updated the web page is no longer available and the ESP32 will restart.
When we test the update, we’ll verify the above by uploading a build file (.bin file) with a different SoftAP SSID and different color web page background.

26.5 UTILIZING ESP-IDF FOR OTA UPDATES:

Receive the file from the web page →via the web server, calling httpd_req_recv.
Identify where the .bin file starts, then →call esp_ota_begin.
Write the first part of the data →call esp_ota_write.
Continue to receiving the file calling httpd_req_recv and esp_ota_write until all content is received.
Finish OTA update and validate newly written app image →call esp_ota_end.
Configure OTA data for a new boot partition →call esp_ota_set_boot_partition.
Restart the ESP32 →call esp_restart.

CHAPTER#27 CODES OF OTA FIRMWARE UPDATES

27.1 CODE OF HTTP_SERVER.H:

```
/*
 * http_server.h
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_HTTP_SERVER_H_
#define MAIN_HTTP_SERVER_H_
#define OTA_UPDATE_PENDING          0
#define OTA_UPDATE_SUCCESSFUL       1
#define OTA_UPDATE_FAILED           -1
/**
 * Messages for the HTTP monitor
 */
typedef enum http_server_message
```

```
{
	HTTP_MSG_WIFI_CONNECT_INIT = 0,
	HTTP_MSG_WIFI_CONNECT_SUCCESS,
	HTTP_MSG_WIFI_CONNECT_FAIL,
	HTTP_MSG_OTA_UPDATE_SUCCESSFUL,
	HTTP_MSG_OTA_UPDATE_FAILED,
} http_server_message_e;
/**
 * Structure for the message queue
 */
typedef struct http_server_queue_message
{
	http_server_message_e msgID;
} http_server_queue_message_t;
/**
 * Sends a message to the queue
 * @param msgID message ID from the http_server_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue,
otherwise pdFALSE.
 * @note Expand the parameter list based on your requirements e.g. how
you've expanded the http_server_queue_message_t.
 */
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID);
/**
 * Starts the HTTP server.
 */
void http_server_start(void);
/**
 * Stops the HTTP server.
 */
void http_server_stop(void);
/**
 * Timer callback function which calls esp_restart upon successful
firmware update.
```

```
 */
void http_server_fw_update_reset_callback(void *arg);
#endif /* MAIN_HTTP_SERVER_H_ */
```

27.2 CODE OF HTTP_SERVER.C:

```
/*
 * http_server.c
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include "esp_http_server.h"
#include "esp_log.h"
#include "esp_ota_ops.h"
#include "sys/param.h"
#include "http_server.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG[] = "http_server";
// Firmware update status
static int g_fw_update_status = OTA_UPDATE_PENDING;
// HTTP server task handle
static httpd_handle_t http_server_handle = NULL;
// HTTP server monitor task handle
static TaskHandle_t task_http_server_monitor = NULL;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t http_server_monitor_queue_handle;
/**
 * ESP32 timer configuration passed to esp_timer_create.
 */
const esp_timer_create_args_t fw_update_reset_args = {
		.callback = &http_server_fw_update_reset_callback,
		.arg = NULL,
```

```
		.dispatch_method = ESP_TIMER_TASK,
		.name = "fw_update_reset"
};
esp_timer_handle_t fw_update_reset;
// Embedded files: JQuery, index.html, app.css, app.js and favicon.ico
files
extern const uint8_t jquery_3_3_1_min_js_start[]
	asm("_binary_jquery_3_3_1_min_js_start");
extern const uint8_t jquery_3_3_1_min_js_end[]
	asm("_binary_jquery_3_3_1_min_js_end");
extern const uint8_t index_html_start[]
	asm("_binary_index_html_start");
extern const uint8_t index_html_end[]
	asm("_binary_index_html_end");
extern const uint8_t app_css_start[]
	asm("_binary_app_css_start");
extern const uint8_t app_css_end[]
	asm("_binary_app_css_end");
extern const uint8_t app_js_start[]
	asm("_binary_app_js_start");
extern const uint8_t app_js_end[]
	asm("_binary_app_js_end");
extern const uint8_t favicon_ico_start[]
	asm("_binary_favicon_ico_start");
extern const uint8_t favicon_ico_end[]
	asm("_binary_favicon_ico_end");
/**
 * Checks the g_fw_update_status and creates the fw_update_reset timer
if g_fw_update_status is true.
 */
static void http_server_fw_update_reset_timer(void)
{
	if (g_fw_update_status == OTA_UPDATE_SUCCESSFUL)
	{
```

```
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
updated successful starting FW update reset timer");
		// Give the web page a chance to receive an acknowledge
back and initialize the timer

	ESP_ERROR_CHECK(esp_timer_create(&fw_update_reset_args,
&fw_update_reset));

	ESP_ERROR_CHECK(esp_timer_start_once(fw_update_reset,
8000000));
	}
	else
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
update unsuccessful");
	}
}
/**
 * HTTP server monitor task used to track events of the HTTP server
 * @param pvParameters parameter which can be passed to the task.
 */
static void http_server_monitor(void *parameter)
{
	http_server_queue_message_t msg;
	for (;;)
	{
		if (xQueueReceive(http_server_monitor_queue_handle,
&msg, portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case HTTP_MSG_WIFI_CONNECT_INIT:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_INIT");
					break;
```

```
                        case HTTP_MSG_WIFI_CONNECT_SUCCESS:
                                ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_SUCCESS");
                                break;
                        case HTTP_MSG_WIFI_CONNECT_FAIL:
                                ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_FAIL");
                                break;
                        case
HTTP_MSG_OTA_UPDATE_SUCCESSFUL:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_SUCCESSFUL");
                                g_fw_update_status =
OTA_UPDATE_SUCCESSFUL;
                                http_server_fw_update_reset_timer();
                                break;
                        case HTTP_MSG_OTA_UPDATE_FAILED:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_FAILED");
                                g_fw_update_status =
OTA_UPDATE_FAILED;
                                break;
                        default:
                                break;
                }
            }
        }
}
/**
 * Jquery get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_jquery_handler(httpd_req_t *req)
{
```

```
	ESP_LOGI(TAG, "Jquery requested");
	httpd_resp_set_type(req, "application/javascript");
	httpd_resp_send(req, (const char *)jquery_3_3_1_min_js_start,
jquery_3_3_1_min_js_end - jquery_3_3_1_min_js_start);
	return ESP_OK;
}
/**
 * Sends the index.html page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_index_html_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "index.html requested");
	httpd_resp_set_type(req, "text/html");
	httpd_resp_send(req, (const char *)index_html_start,
index_html_end - index_html_start);
	return ESP_OK;
}
/**
 * app.css get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_css_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.css requested");
	httpd_resp_set_type(req, "text/css");
	httpd_resp_send(req, (const char *)app_css_start, app_css_end -
app_css_start);
	return ESP_OK;
}
/**
 * app.js get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
```

```
 * @return ESP_OK
 */
static esp_err_t http_server_app_js_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.js requested");

	httpd_resp_set_type(req, "application/javascript");
	httpd_resp_send(req, (const char *)app_js_start, app_js_end -
app_js_start);

	return ESP_OK;
}
/**
 * Sends the .ico (icon) file when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_favicon_ico_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "favicon.ico requested");
	httpd_resp_set_type(req, "image/x-icon");
	httpd_resp_send(req, (const char *)favicon_ico_start,
favicon_ico_end - favicon_ico_start);
	return ESP_OK;
}
/**
 * Receives the .bin file fia the web page and handles the firmware
update
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK, otherwise ESP_FAIL if timeout occurs and the
update cannot be started.
 */
esp_err_t http_server_OTA_update_handler(httpd_req_t *req)
{
	esp_ota_handle_t ota_handle;
```

```
	char ota_buff[1024];
	int content_length = req->content_len;
	int content_received = 0;
	int recv_len;
	bool is_req_body_started = false;
	bool flash_successful = false;
	const esp_partition_t *update_partition =
esp_ota_get_next_update_partition(NULL);
	do
	{
		// Read the data for the request
		if ((recv_len = httpd_req_recv(req, ota_buff,
MIN(content_length, sizeof(ota_buff)))) < 0)
		{
			// Check if timeout occurred
			if (recv_len == HTTPD_SOCK_ERR_TIMEOUT)
			{
				ESP_LOGI(TAG,
"http_server_OTA_update_handler: Socket Timeout");
				continue; ///> Retry receiving if timeout occurred
			}
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
OTA other Error %d", recv_len);
			return ESP_FAIL;
		}
		printf("http_server_OTA_update_handler: OTA RX: %d of
%d\r", content_received, content_length);

		// Is this the first data we are receiving
		// If so, it will have the information in the header that we
need.
		if (!is_req_body_started)
		{
			is_req_body_started = true;
```

```
                    // Get the location of the .bin file content (remove the
web form data)
                    char *body_start_p = strstr(ota_buff, "\r\n\r\n") + 4;
                    int body_part_len = recv_len - (body_start_p -
ota_buff);
                    printf("http_server_OTA_update_handler: OTA file
size: %d\r\n", content_length);
                    esp_err_t err = esp_ota_begin(update_partition,
OTA_SIZE_UNKNOWN, &ota_handle);
                    if (err != ESP_OK)
                    {
                        printf("http_server_OTA_update_handler: Error
with OTA begin, cancelling OTA\r\n");
                        return ESP_FAIL;
                    }
                    else
                    {
                        printf("http_server_OTA_update_handler:
Writing to partition subtype %d at offset 0x%x\r\n", update_partition-
>subtype, update_partition->address);
                    }
                    // Write this first part of the data
                    esp_ota_write(ota_handle, body_start_p,
body_part_len);
                    content_received += body_part_len;
                }
                else
                {
                    // Write OTA data
                    esp_ota_write(ota_handle, ota_buff, recv_len);
                    content_received += recv_len;
                }
        } while (recv_len > 0 && content_received < content_length);
        if (esp_ota_end(ota_handle) == ESP_OK)
        {
```

```
		// Lets update the partition
		if (esp_ota_set_boot_partition(update_partition) ==
ESP_OK)
		{
			const esp_partition_t *boot_partition =
esp_ota_get_boot_partition();
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
Next boot partition subtype %d at offset 0x%x", boot_partition-
>subtype, boot_partition->address);
			flash_successful = true;
		}
		else
		{
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
FLASHED ERROR!!!");
		}
	}
	else
	{
		ESP_LOGI(TAG, "http_server_OTA_update_handler:
esp_ota_end ERROR!!!");
	}

	// We won't update the global variables throughout the file, so send
the message about the status
	if (flash_successful) {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_SU
CCESSFUL); } else {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_FAI
LED); }
	return ESP_OK;
}
/**
 * OTA status handler responds with the firmware update status after the
OTA update is started
```

```
 * and responds with the compile time/date when the page is first
requested
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
esp_err_t http_server_OTA_status_handler(httpd_req_t *req)
{
	char otaJSON[100];
	ESP_LOGI(TAG, "OTAstatus requested");
	sprintf(otaJSON,
"{\"ota_update_status\":%d,\"compile_time\":\"%s\",\"compile_date\":\"
%s\"}", g_fw_update_status, __TIME__, __DATE__);

	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, otaJSON, strlen(otaJSON));
	return ESP_OK;
}
/**
 * Sets up the default httpd server configuration.
 * @return http server instance handle if successful, NULL otherwise.
 */
static httpd_handle_t http_server_configure(void)
{
	// Generate the default configuration
	httpd_config_t config = HTTPD_DEFAULT_CONFIG();
	// Create HTTP server monitor task
	xTaskCreatePinnedToCore(&http_server_monitor,
"http_server_monitor", HTTP_SERVER_MONITOR_STACK_SIZE,
NULL, HTTP_SERVER_MONITOR_PRIORITY,
&task_http_server_monitor, HTTP_SERVER_MONITOR_CORE_ID);
	// Create the message queue
	http_server_monitor_queue_handle = xQueueCreate(3,
sizeof(http_server_queue_message_t));
	// The core that the HTTP server will run on
	config.core_id = HTTP_SERVER_TASK_CORE_ID;
```

```
	// Adjust the default priority to 1 less than the wifi application task
	config.task_priority = HTTP_SERVER_TASK_PRIORITY;
	// Bump up the stack size (default is 4096)
	config.stack_size = HTTP_SERVER_TASK_STACK_SIZE;
	// Increase uri handlers
	config.max_uri_handlers = 20;
	// Increase the timeout limits
	config.recv_wait_timeout = 10;
	config.send_wait_timeout = 10;
	ESP_LOGI(TAG,
			"http_server_configure: Starting server on port: '%d'
with task priority: '%d'",
			config.server_port,
			config.task_priority);
	// Start the httpd server
	if (httpd_start(&http_server_handle, &config) == ESP_OK)
	{
		ESP_LOGI(TAG, "http_server_configure: Registering URI
handlers");
		// register query handler
		httpd_uri_t jquery_js = {
				.uri = "/jquery-3.3.1.min.js",
				.method = HTTP_GET,
				.handler = http_server_jquery_handler,
				.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &jquery_js);

		// register index.html handler
		httpd_uri_t index_html = {
				.uri = "/",
				.method = HTTP_GET,
				.handler = http_server_index_html_handler,
				.user_ctx = NULL
		};
```

```
        httpd_register_uri_handler(http_server_handle,
&index_html);

        // register app.css handler
        httpd_uri_t app_css = {
                .uri = "/app.css",
                .method = HTTP_GET,
                .handler = http_server_app_css_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_css);

        // register app.js handler
        httpd_uri_t app_js = {
                .uri = "/app.js",
                .method = HTTP_GET,
                .handler = http_server_app_js_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_js);

        // register favicon.ico handler
        httpd_uri_t favicon_ico = {
                .uri = "/favicon.ico",
                .method = HTTP_GET,
                .handler = http_server_favicon_ico_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&favicon_ico);
        // register OTAupdate handler
        httpd_uri_t OTA_update = {
                .uri = "/OTAupdate",
                .method = HTTP_POST,
                .handler = http_server_OTA_update_handler,
```

```
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_update);
        // register OTAstatus handler
        httpd_uri_t OTA_status = {
                .uri = "/OTAstatus",
                .method = HTTP_POST,
                .handler = http_server_OTA_status_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_status);
        return http_server_handle;
    }
    return NULL;
}
void http_server_start(void)
{
    if (http_server_handle == NULL)
    {
        http_server_handle = http_server_configure();
    }
}
void http_server_stop(void)
{
    if (http_server_handle)
    {
        httpd_stop(http_server_handle);
        ESP_LOGI(TAG, "http_server_stop: stopping HTTP
server");
        http_server_handle = NULL;
    }
    if (task_http_server_monitor)
    {
```

```
                vTaskDelete(task_http_server_monitor);
                ESP_LOGI(TAG, "http_server_stop: stopping HTTP server
monitor");
                task_http_server_monitor = NULL;
        }
}
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID)
{
        http_server_queue_message_t msg;
        msg.msgID = msgID;
        return xQueueSend(http_server_monitor_queue_handle, &msg,
portMAX_DELAY);
}
void http_server_fw_update_reset_callback(void *arg)
{
        ESP_LOGI(TAG, "http_server_fw_update_reset_callback: Timer
timed-out, restarting the device");
        esp_restart();
}
```

CHAPTER#28 DHT22 SENSOR

28.1 AM2302 DHT22 SENSOR TEMOERATURE AND HUMDITITY SENSOR MODULE:

AM2302 Humicap digital temperature and humidity module outputs a digital signal and contains a calibrated temperature and humidity sensor. Only three connections; data, VCC and ground.It uses a dedicated digital modules capture technology and the temperature and humidity sensor technology and provides higher accuracy than the DHT11.
Temperature range is -40~80 Degrees Celsius and Humidity Range 20-90%.

28.2 PIN CONNECTION: VCC, DATA (GPIO 25) AND GROUND:

Figure 28.2.11:PIN CONNECTION:VCC ,DATA(GPIO 25) AND GROUND

28.3 About the DHT22 Sensor Implementation:

We will use an existing library for the DHT22.c and DHT22.h files and will attach to the resources in the next sections –these files will be included under main.
Create a FreeRTOS task to read the DHT22 data at a specified interval and test it.
Update the index.html, app.css and app.js web page files in order to display the data.
Update the web server to respond to GET requests with Temperature and Humidity data from the DHT22 sensor.

CHAPTER#29 CODES OF DHT22 SENSOR

29.1 CODE OF DHT22.C:

```
/*
 * DHT22.C
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#define LOG_LOCAL_LEVEL ESP_LOG_VERBOSE
#include <stdio.h>
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#include "driver/gpio.h"
#include "DHT22.h"
// == global defines
=============================================
static const char* TAG = "DHT";
int DHTgpio = 4;                        // my default DHT pin = 4
float humidity = 0.;
float temperature = 0.;
```

```
// == set the DHT used
pin==========================================
void setDHTgpio( int gpio )
{
	DHTgpio = gpio;
}
// == get temp & hum
=================================================
float getHumidity() { return humidity; }
float getTemperature() { return temperature; }
// == error handler
==================================================
void errorHandler(int response)
{
	switch(response) {
		case DHT_TIMEOUT_ERROR :
			ESP_LOGE( TAG, "Sensor Timeout\n" );
			break;
		case DHT_CHECKSUM_ERROR:
			ESP_LOGE( TAG, "CheckSum error\n" );
			break;
		case DHT_OK:
			break;
		default :
			ESP_LOGE( TAG, "Unknown error\n" );
	}
}
/*-----------------------------------------------------------------------------
;
;	get next state
;
;	I don't like this logic. It needs some interrupt blocking / priority
;	to ensure it runs in realtime.
;
;-----------------------------------------------------------------------------*/
```

```
int getSignalLevel( int usTimeOut, bool state )
{
	int uSec = 0;
	while( gpio_get_level(DHTgpio)==state ) {

		if( uSec > usTimeOut )
			return -1;
		++uSec;
		ets_delay_us(1);	// uSec delay
	}
	return uSec;
}
/*-----------------------------------------------------------------------------
;
;	read DHT22 sensor
```

copy/paste from AM2302/DHT22 Docu:
DATA: Hum = 16 bits, Temp = 16 Bits, check-sum = 8 Bits
Example: MCU has received 40 bits data from AM2302 as
0000 0010 1000 1100 0000 0001 0101 1111 1110 1110
16 bits RH data + 16 bits T data + check sum
1) we convert 16 bits RH data from binary system to decimal system,
0000 0010 1000 1100 → 652
Binary system Decimal system: RH=652/10=65.2%RH
2) we convert 16 bits T data from binary system to decimal system, 0000 0001 0101 1111 → 351
Binary system Decimal system: T=351/10=35.1°C
When highest bit of temperature is 1, it means the temperature is below 0 degree Celsius.
Example: 1000 0000 0110 0101, T= minus 10.1°C: 16 bits T data
3) Check Sum=0000 0010+1000 1100+0000 0001+0101 1111=1110 1110 Check-sum=the last 8 bits of Sum=11101110
Signal & Timings:
The interval of whole process must be beyond 2 seconds.
To request data from DHT:
1) Sent low pulse for > 1~10 ms (MILI SEC)

2) Sent high pulse for > 20~40 us (Micros).
3) When DHT detects the start signal, it will pull low the bus 80us as response signal,
  then the DHT pulls up 80us for preparation to send data.
4) When DHT is sending data to MCU, every bit's transmission begin with low-voltage-level that last 50us,
  the following high-voltage-level signal's length decide the bit is "1" or "0".

```
        0: 26~28 us
        1: 70 us
;-----------------------------------------------------------------------------*/
#define MAXdhtData 5       // to complete 40 = 5*8 Bits
int readDHT()
{
int uSec = 0;
uint8_t dhtData[MAXdhtData];
uint8_t byteInx = 0;
uint8_t bitInx = 7;
        for (int k = 0; k<MAXdhtData; k++)
                dhtData[k] = 0;
        // == Send start signal to DHT sensor ===========
        gpio_set_direction( DHTgpio, GPIO_MODE_OUTPUT );
        // pull down for 3 ms for a smooth and nice wake up
        gpio_set_level( DHTgpio, 0 );
        ets_delay_us( 3000 );
        // pull up for 25 us for a gentile asking for data
        gpio_set_level( DHTgpio, 1 );
        ets_delay_us( 25 );
        gpio_set_direction( DHTgpio, GPIO_MODE_INPUT );        //
change to input mode
        // == DHT will keep the line low for 80 us and then high for 80us
====
        uSec = getSignalLevel( 85, 0 );
//      ESP_LOGI( TAG, "Response = %d", uSec );
        if( uSec<0 ) return DHT_TIMEOUT_ERROR;
```

```
        // -- 80us up ----------------------
        uSec = getSignalLevel( 85, 1 );
//      ESP_LOGI( TAG, "Response = %d", uSec );
        if( uSec<0 ) return DHT_TIMEOUT_ERROR;
        // == No errors, read the 40 data bits ================
        for( int k = 0; k < 40; k++ ) {
                // -- starts new data transmission with >50us low signal
                uSec = getSignalLevel( 56, 0 );
                if( uSec<0 ) return DHT_TIMEOUT_ERROR;
                // -- check to see if after >70us rx data is a 0 or a 1
                uSec = getSignalLevel( 75, 1 );
                if( uSec<0 ) return DHT_TIMEOUT_ERROR;
                // add the current read to the output data
                // since all dhtData array where set to 0 at the start,
                // only look for "1" (>28us us)
                if (uSec > 40) {
                        dhtData[ byteInx ] |= (1 << bitInx);
                        }
                // index to next byte
                if (bitInx == 0) { bitInx = 7; ++byteInx; }
                else bitInx--;
        }
        // == get humidity from Data[0] and Data[1]
==========================
        humidity = dhtData[0];
        humidity *= 0x100;                              // >> 8
        humidity += dhtData[1];
        humidity /= 10;                                 // get the decimal
        // == get temp from Data[2] and Data[3]
        temperature = dhtData[2] & 0x7F;
        temperature *= 0x100;                           // >> 8
        temperature += dhtData[3];
        temperature /= 10;
        if( dhtData[2] & 0x80 )                 // negative temp, brrr it's
freezing
```

```
		temperature *= -1;
	// == verify if checksum is ok
==============================================
	// Checksum is the sum of Data 8 bits masked out 0xFF
	if (dhtData[4] == ((dhtData[0] + dhtData[1] + dhtData[2] +
dhtData[3]) & 0xFF))
		return DHT_OK;
	else
		return DHT_CHECKSUM_ERROR;
}
```

29.2 CODE OF DHT22.H:

```
/*
 * DHT22.h
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef DHT22_H_
#define DHT22_H_
#define DHT_OK 0
#define DHT_CHECKSUM_ERROR -1
#define DHT_TIMEOUT_ERROR -2
// == function prototypes
==========================================
void  setDHTgpio(int gpio);
void  errorHandler(int response);
int     readDHT();
float  getHumidity();
float  getTemperature();
int     getSignalLevel( int usTimeOut, bool state );
#endif
```

29.3 CODE OF HTTP_SERVER.C:

```
/*
 * http_server.c
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include "esp_http_server.h"
#include "esp_log.h"
#include "esp_ota_ops.h"
#include "sys/param.h"
#include "DHT22.h"
#include "http_server.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG[] = "http_server";
// Firmware update status
static int g_fw_update_status = OTA_UPDATE_PENDING;
// HTTP server task handle
static httpd_handle_t http_server_handle = NULL;
// HTTP server monitor task handle
static TaskHandle_t task_http_server_monitor = NULL;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t http_server_monitor_queue_handle;
/**
 * ESP32 timer configuration passed to esp_timer_create.
 */
const esp_timer_create_args_t fw_update_reset_args = {
		.callback = &http_server_fw_update_reset_callback,
		.arg = NULL,
		.dispatch_method = ESP_TIMER_TASK,
		.name = "fw_update_reset"
};
```

```
esp_timer_handle_t fw_update_reset;
// Embedded files: JQuery, index.html, app.css, app.js and favicon.ico
files
extern const uint8_t jquery_3_3_1_min_js_start[]
	asm("_binary_jquery_3_3_1_min_js_start");
extern const uint8_t jquery_3_3_1_min_js_end[]
	asm("_binary_jquery_3_3_1_min_js_end");
extern const uint8_t index_html_start[]
	asm("_binary_index_html_start");
extern const uint8_t index_html_end[]
	asm("_binary_index_html_end");
extern const uint8_t app_css_start[]
	asm("_binary_app_css_start");
extern const uint8_t app_css_end[]
	asm("_binary_app_css_end");
extern const uint8_t app_js_start[]
	asm("_binary_app_js_start");
extern const uint8_t app_js_end[]
	asm("_binary_app_js_end");
extern const uint8_t favicon_ico_start[]
	asm("_binary_favicon_ico_start");
extern const uint8_t favicon_ico_end[]
	asm("_binary_favicon_ico_end");
/**
 * Checks the g_fw_update_status and creates the fw_update_reset timer
if g_fw_update_status is true.
 */
static void http_server_fw_update_reset_timer(void)
{
	if (g_fw_update_status == OTA_UPDATE_SUCCESSFUL)
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
updated successful starting FW update reset timer");
		// Give the web page a chance to receive an acknowledge
back and initialize the timer
```

```
	ESP_ERROR_CHECK(esp_timer_create(&fw_update_reset_args,
&fw_update_reset));

	ESP_ERROR_CHECK(esp_timer_start_once(fw_update_reset,
8000000));
	}
	else
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
update unsuccessful");
	}
}
/**
 * HTTP server monitor task used to track events of the HTTP server
 * @param pvParameters parameter which can be passed to the task.
 */
static void http_server_monitor(void *parameter)
{
	http_server_queue_message_t msg;
	for (;;)
	{
		if (xQueueReceive(http_server_monitor_queue_handle,
&msg, portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case HTTP_MSG_WIFI_CONNECT_INIT:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_INIT");
					break;
				case HTTP_MSG_WIFI_CONNECT_SUCCESS:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_SUCCESS");
					break;
```

```
                        case HTTP_MSG_WIFI_CONNECT_FAIL:
                                ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_FAIL");
                                break;
                        case
HTTP_MSG_OTA_UPDATE_SUCCESSFUL:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_SUCCESSFUL");
                                g_fw_update_status =
OTA_UPDATE_SUCCESSFUL;
                                http_server_fw_update_reset_timer();
                                break;
                        case HTTP_MSG_OTA_UPDATE_FAILED:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_FAILED");
                                g_fw_update_status =
OTA_UPDATE_FAILED;
                                break;
                        default:
                                break;
                }
            }
        }
}
/**
 * Jquery get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_jquery_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "Jquery requested");
        httpd_resp_set_type(req, "application/javascript");
        httpd_resp_send(req, (const char *)jquery_3_3_1_min_js_start,
jquery_3_3_1_min_js_end - jquery_3_3_1_min_js_start);
```

```
	return ESP_OK;
}
/**
 * Sends the index.html page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_index_html_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "index.html requested");
	httpd_resp_set_type(req, "text/html");
	httpd_resp_send(req, (const char *)index_html_start,
index_html_end - index_html_start);
	return ESP_OK;
}
/**
 * app.css get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_css_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.css requested");
	httpd_resp_set_type(req, "text/css");
	httpd_resp_send(req, (const char *)app_css_start, app_css_end -
app_css_start);
	return ESP_OK;
}
/**
 * app.js get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_js_handler(httpd_req_t *req)
{
```

```
	ESP_LOGI(TAG, "app.js requested");

	httpd_resp_set_type(req, "application/javascript");
	httpd_resp_send(req, (const char *)app_js_start, app_js_end -
app_js_start);
	return ESP_OK;
}
/**
 * Sends the .ico (icon) file when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_favicon_ico_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "favicon.ico requested");
	httpd_resp_set_type(req, "image/x-icon");
	httpd_resp_send(req, (const char *)favicon_ico_start,
favicon_ico_end - favicon_ico_start);
	return ESP_OK;
}
/**
 * Receives the .bin file fia the web page and handles the firmware
update
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK, otherwise ESP_FAIL if timeout occurs and the
update cannot be started.
 */
esp_err_t http_server_OTA_update_handler(httpd_req_t *req)
{
	esp_ota_handle_t ota_handle;
	char ota_buff[1024];
	int content_length = req->content_len;
	int content_received = 0;
	int recv_len;
	bool is_req_body_started = false;
```

```
	bool flash_successful = false;
	const esp_partition_t *update_partition =
esp_ota_get_next_update_partition(NULL);
	do
	{
		// Read the data for the request
		if ((recv_len = httpd_req_recv(req, ota_buff,
MIN(content_length, sizeof(ota_buff)))) < 0)
		{
			// Check if timeout occurred
			if (recv_len == HTTPD_SOCK_ERR_TIMEOUT)
			{
				ESP_LOGI(TAG,
"http_server_OTA_update_handler: Socket Timeout");
				continue; ///> Retry receiving if timeout occurred
			}
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
OTA other Error %d", recv_len);
			return ESP_FAIL;
		}
		printf("http_server_OTA_update_handler: OTA RX: %d of
%d\r", content_received, content_length);

		// Is this the first data we are receiving
		// If so, it will have the information in the header that we
need.
		if (!is_req_body_started)
		{
			is_req_body_started = true;

			// Get the location of the .bin file content (remove the
web form data)
			char *body_start_p = strstr(ota_buff, "\r\n\r\n") + 4;
			int body_part_len = recv_len - (body_start_p -
ota_buff);
```

```
                printf("http_server_OTA_update_handler: OTA file
size: %d\r\n", content_length);

                esp_err_t err = esp_ota_begin(update_partition,
OTA_SIZE_UNKNOWN, &ota_handle);
                if (err != ESP_OK)
                {
                        printf("http_server_OTA_update_handler: Error
with OTA begin, cancelling OTA\r\n");
                        return ESP_FAIL;
                }
                else
                {
                        printf("http_server_OTA_update_handler:
Writing to partition subtype %d at offset 0x%x\r\n", update_partition-
>subtype, update_partition->address);
                }

                // Write this first part of the data
                esp_ota_write(ota_handle, body_start_p,
body_part_len);
                content_received += body_part_len;
        }
        else
        {
                // Write OTA data
                esp_ota_write(ota_handle, ota_buff, recv_len);
                content_received += recv_len;
        }
    } while (recv_len > 0 && content_received < content_length);
    if (esp_ota_end(ota_handle) == ESP_OK)
    {
        // Lets update the partition
```

```
		if (esp_ota_set_boot_partition(update_partition) ==
ESP_OK)
		{
			const esp_partition_t *boot_partition =
esp_ota_get_boot_partition();
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
Next boot partition subtype %d at offset 0x%x", boot_partition-
>subtype, boot_partition->address);
			flash_successful = true;
		}
		else
		{
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
FLASHED ERROR!!!");
		}
	}
	else
	{
		ESP_LOGI(TAG, "http_server_OTA_update_handler:
esp_ota_end ERROR!!!");
	}
	// We won't update the global variables throughout the file, so send
the message about the status
	if (flash_successful) {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_SU
CCESSFUL); } else {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_FAI
LED); }
	return ESP_OK;
}
/**
 * OTA status handler responds with the firmware update status after the
OTA update is started
 * and responds with the compile time/date when the page is first
requested
```

```
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
esp_err_t http_server_OTA_status_handler(httpd_req_t *req)
{
	char otaJSON[100];
	ESP_LOGI(TAG, "OTAstatus requested");
	sprintf(otaJSON,
"{\"ota_update_status\":%d,\"compile_time\":\"%s\",\"compile_date\":\"
%s\"}", g_fw_update_status, __TIME__, __DATE__);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, otaJSON, strlen(otaJSON));

	return ESP_OK;
}
/**
 * DHT sensor readings JSON handler responds with DHT22 sensor data
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
static esp_err_t
http_server_get_dht_sensor_readings_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/dhtSensor.json requested");
	char dhtSensorJSON[100];
	sprintf(dhtSensorJSON,
"{\"temp\":\"%.1f\",\"humidity\":\"%.1f\"}", getTemperature(),
getHumidity());
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, dhtSensorJSON, strlen(dhtSensorJSON));
	return ESP_OK;
}
/**
 * Sets up the default httpd server configuration.
 * @return http server instance handle if successful, NULL otherwise.
```

```
 */
static httpd_handle_t http_server_configure(void)
{
	// Generate the default configuration
	httpd_config_t config = HTTPD_DEFAULT_CONFIG();
	// Create HTTP server monitor task
	xTaskCreatePinnedToCore(&http_server_monitor,
"http_server_monitor", HTTP_SERVER_MONITOR_STACK_SIZE,
NULL, HTTP_SERVER_MONITOR_PRIORITY,
&task_http_server_monitor, HTTP_SERVER_MONITOR_CORE_ID);
	// Create the message queue
	http_server_monitor_queue_handle = xQueueCreate(3,
sizeof(http_server_queue_message_t));
	// The core that the HTTP server will run on
	config.core_id = HTTP_SERVER_TASK_CORE_ID;
	// Adjust the default priority to 1 less than the wifi application task
	config.task_priority = HTTP_SERVER_TASK_PRIORITY;
	// Bump up the stack size (default is 4096)
	config.stack_size = HTTP_SERVER_TASK_STACK_SIZE;
	// Increase uri handlers
	config.max_uri_handlers = 20;
	// Increase the timeout limits
	config.recv_wait_timeout = 10;
	config.send_wait_timeout = 10;
	ESP_LOGI(TAG,
			"http_server_configure: Starting server on port: '%d'
with task priority: '%d'",
			config.server_port,
			config.task_priority);
	// Start the httpd server
	if (httpd_start(&http_server_handle, &config) == ESP_OK)
	{
		ESP_LOGI(TAG, "http_server_configure: Registering URI
handlers");
		// register query handler
```

```
		httpd_uri_t jquery_js = {
			.uri = "/jquery-3.3.1.min.js",
			.method = HTTP_GET,
			.handler = http_server_jquery_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &jquery_js);
		// register index.html handler
		httpd_uri_t index_html = {
			.uri = "/",
			.method = HTTP_GET,
			.handler = http_server_index_html_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&index_html);
		// register app.css handler
		httpd_uri_t app_css = {
			.uri = "/app.css",
			.method = HTTP_GET,
			.handler = http_server_app_css_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &app_css);
		// register app.js handler
		httpd_uri_t app_js = {
			.uri = "/app.js",
			.method = HTTP_GET,
			.handler = http_server_app_js_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &app_js);
		// register favicon.ico handler
		httpd_uri_t favicon_ico = {
			.uri = "/favicon.ico",
```

```
                .method = HTTP_GET,
                .handler = http_server_favicon_ico_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&favicon_ico);
        // register OTAupdate handler
        httpd_uri_t OTA_update = {
                .uri = "/OTAupdate",
                .method = HTTP_POST,
                .handler = http_server_OTA_update_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_update);
        // register OTAstatus handler
        httpd_uri_t OTA_status = {
                .uri = "/OTAstatus",
                .method = HTTP_POST,
                .handler = http_server_OTA_status_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_status);
        // register dhtSensor.json handler
        httpd_uri_t dht_sensor_json = {
                .uri = "/dhtSensor.json",
                .method = HTTP_GET,
                .handler =
http_server_get_dht_sensor_readings_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&dht_sensor_json);
        return http_server_handle;
```

```
	}
	return NULL;
}
void http_server_start(void)
{
	if (http_server_handle == NULL)
	{
		http_server_handle = http_server_configure();
	}
}
void http_server_stop(void)
{
	if (http_server_handle)
	{
		httpd_stop(http_server_handle);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP
server");
		http_server_handle = NULL;
	}
	if (task_http_server_monitor)
	{
		vTaskDelete(task_http_server_monitor);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP server
monitor");
		task_http_server_monitor = NULL;
	}
}
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID)
{
	http_server_queue_message_t msg;
	msg.msgID = msgID;
	return xQueueSend(http_server_monitor_queue_handle, &msg,
portMAX_DELAY);
}
```

```
void http_server_fw_update_reset_callback(void *arg)
{
	ESP_LOGI(TAG, "http_server_fw_update_reset_callback: Timer
timed-out, restarting the device");
	esp_restart();
}
```

29.4 CODE OF MAIN.C:

```
/**
 * Application entry point.
 */
#include "nvs_flash.h"
#include "DHT22.h"
#include "wifi_app.h"
void app_main(void)
{
   // Initialize NVS
	esp_err_t ret = nvs_flash_init();
	if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND)
	{
		ESP_ERROR_CHECK(nvs_flash_erase());
		ret = nvs_flash_init();
	}
	ESP_ERROR_CHECK(ret);
	// Start Wifi
	wifi_app_start();
	// Start DHT22 Sensor task
	DHT22_task_start();
}
```

29.5 CODE OF APP.CSS:

```
body {
   background-color: #f1f1f1;
   font-family: HelveticaNeueRegular, HelveticaNeue-Regular,
"Helvetica Neue Regular", HelveticaNeue, "Helvetica Neue", Helvetica,
Arial;
   color: #0272B7;
}
h1 {
   display: block;
   text-align: center;
   margin: 0;
   padding: 10px;
   font-size: 1.7em;
   color: #1d3557;
}
h2 {
      font-size: 1.4em;
   color: #1d3557;
}
h3 {
      font-size: 1.2em;
   color: #1d3557;
}
h4 {
      font-size: 1em;
   color: #1d3557;
}
#latest_firmware, #latest_firmware_label {
      display:inline;
}
#latest_firmware {
      color: #1d3557;
}
```

```
#temperature_reading {
        display:inline;
        color: #1d3557;
}
#humidity_reading {
        display:inline;
        color: #1d3557;
}
.buttons {
   padding: 5px;
}
input[type="button"] {
        font-size: 12px;
        border: 0;
   line-height: 2;
   padding: 0 5px;
   text-align: center;
   color: #ffffff;
   text-shadow: 1px 1px 1px #000;
   border-radius: 10px;
   background-color: RGBA(106,125,144,0.35);
   background-image: linear-gradient(to top left,
                              rgba(0, 0, 0, .2),
                              rgba(0, 0, 0, .2) 30%,
                              rgba(0, 0, 0, 0));
   box-shadow: inset 2px 2px 3px rgba(255, 255, 255, .6),
             inset -2px -2px 3px rgba(0, 0, 0, .6);
}
input[type="button"]:hover {
   background-color: RGBA(112,164,222,0.68);
}
input[type="button"]:active {
   box-shadow: inset -2px -2px 3px rgba(255, 255, 255, .6),
             inset 2px 2px 3px rgba(0, 0, 0, .6);
}
```

29.5 CODE OF APP.JS:

```
/**
 * Add gobals here
 */
var seconds        = null;
var otaTimerVar =  null;
/**
 * Initialize functions here.
 */
$(document).ready(function(){
        getUpdateStatus();
        startDHTSensorInterval();
});
/**
 * Gets file name and size for display on the web page.
 */
function getFileInfo()
{
   var x = document.getElementById("selected_file");
   var file = x.files[0];

   document.getElementById("file_info").innerHTML = "<h4>File: " +
file.name + "<br>" + "Size: " + file.size + " bytes</h4>";
}
/**
 * Handles the firmware update.
 */
function updateFirmware()
{
   // Form Data
   var formData = new FormData();
   var fileSelect = document.getElementById("selected_file");
   if (fileSelect.files && fileSelect.files.length == 1)
        {
```

```
        var file = fileSelect.files[0];
        formData.set("file", file, file.name);
        document.getElementById("ota_update_status").innerHTML =
"Uploading " + file.name + ", Firmware Update in Progress...";
        // Http Request
        var request = new XMLHttpRequest();
        request.upload.addEventListener("progress", updateProgress);
        request.open('POST', "/OTAupdate");
        request.responseType = "blob";
        request.send(formData);
    }
         else
         {
        window.alert('Select A File First')
    }
}
/**
 * Progress on transfers from the server to the client (downloads).
 */
function updateProgress(oEvent)
{
    if (oEvent.lengthComputable)
         {
        getUpdateStatus();
    }
         else
         {
        window.alert('total size is unknown')
    }
/**
 * Posts the firmware udpate status.
 */
function getUpdateStatus()
{
    var xhr = new XMLHttpRequest();
```

```
    var requestURL = "/OTAstatus";
    xhr.open('POST', requestURL, false);
    xhr.send('ota_update_status');

    if (xhr.readyState == 4 && xhr.status == 200)
        {
        var response = JSON.parse(xhr.responseText);

                document.getElementById("latest_firmware").innerHTML =
response.compile_date + " - " + response.compile_time

                // If flashing was complete it will return a 1, else -1
                // A return of 0 is just for information on the Latest Firmware
request
        if (response.ota_update_status == 1)
                {
                // Set the countdown timer time
            seconds = 10;
            // Start the countdown timer
            otaRebootTimer();
        }
        else if (response.ota_update_status == -1)
                {
            document.getElementById("ota_update_status").innerHTML =
"!!! Upload Error !!!";
        }
    }
}
/**
 * Displays the reboot countdown.
 */
function otaRebootTimer()
{
```

```
    document.getElementById("ota_update_status").innerHTML = "OTA
Firmware Update Complete. This page will close shortly, Rebooting in:
" + seconds;

    if (--seconds == 0)
         {
        clearTimeout(otaTimerVar);
        window.location.reload();
    }
         else
         {
        otaTimerVar = setTimeout(otaRebootTimer, 1000);
    }
}
/**
 * Gets DHT22 sensor temperature and humidity values for display on
the web page.
 */
function getDHTSensorValues()
{
        $.getJSON('/dhtSensor.json', function(data) {
                $("#temperature_reading").text(data["temp"]);
                $("#humidity_reading").text(data["humidity"]);
        });
}
/**
 * Sets the interval for getting the updated DHT22 sensor values.
 */
function startDHTSensorInterval()
{
        setInterval(getDHTSensorValues, 5000);
}
```

29.6 CODE OF INDEX.HTML:

```
<!DOCTYPE html>
<html lang="en">
	<head>
		<meta charset="utf-8"/>
		<meta name="viewport" content="width=device-width,
initial-scale=1.0, user-scalable=no">
		<meta name="apple-mobile-web-app-capable" content="yes"
/>
		<script src='jquery-3.3.1.min.js'></script>
		<link rel="stylesheet" href="app.css">
		<script async src="app.js"></script>
		<title>ESP32 Udemy Course</title>
	</head>
	<body>
	<header>
		<h1>ESP32 Application Development</h1>
	</header>
	<div id="OTA">
	<h2>ESP32 Firmware Update</h2>
		<label id="latest_firmware_label">Latest Firmware: </label>
		<div id="latest_firmware"></div>
		<input type="file" id="selected_file" accept=".bin"
style="display: none;" onchange="getFileInfo()" />
		<div class="buttons">
			<input type="button" value="Select File"
onclick="document.getElementById('selected_file').click();" />
			<input type="button" value="Update Firmware"
onclick="updateFirmware()" />
		</div>
		<h4 id="file_info"></h4>
		<h4 id="ota_update_status"></h4>
	</div>
	<hr>
```

```
        <div id="DHT22Sensor">
                <h2>DHT22 Sensor Readings</h2>
                <label for="temperature_reading">Temperature: </label>
                <div id="temperature_reading"></div>
                <label for="humidity_reading">Humidity: </label>
                <div id="humidity_reading"></div>
        </div>
        <hr>
        </body>
<html>
```

CHAPTER#30 WIFI CONNECT ESP

30.1 WIFI CONNECTION:

Credentials should be entered into text fields of the web page, where a "connect" button is available to kick off the connection process.
The web server will handle receiving the credentials and the application will update the WiFi configuration and then attempt the connection.
The application will let the user know by displaying the connection result on the web page; success for failure.
Connection details (AP name and IP information) will also be displayed on the web page (when there is a connection).
A "disconnect" button will also be available to the user for disconnecting from an active connection.
The RGB LED should indicate a connected status.

30.2 MAKING A WIFI CONNECTION USING THE ESP-IDF:

Within the URI Handler for receiving credentials →Get header value length, using http_req_get_hdr_value_len.
Next, get the value string of a field from the request headers →

httpd_req_get_hdr_value_str.
Next, we need to update the WiFi configuration →Update the station configuration data in the union wifi_config_t.
Set the configuration →For station mode esp_wifi_set_config the interface should be specified as ESP_IF_WIFI_STA.
Attempt the connection →Using esp_wifi_connect.
Disconnect from an active connection →Using esp_wifi_disconnect.

30.3 ESP IDF APIs USED (DISPLAYING CONECTION INFORMATION TO THE WEB PAGE…):

Get the access point information (SSID) for display on the page →Using wifi_ap_record_t and pass it to esp_wifi_sta_get_ap_info.
Get the IP connection information (assigned to the ESP32) for display on the page →Using esp_netif_ip_info_t and pass it to esp_netif_get_ip_info.
Convert the numeric IP address into dotted decimal ASCII →Using esp_ip4addr_ntoa.

30.4 WIFI APPLICATION:

30.4.1 CONNECTING DEVICES:

Becomes "station" of ESP32's SoftAP when connected. DHCP service from the ESP32's SoftAP will dynamically assign an IP to your device. Interact with the ESP32 via the web page.

Figure 30.4.12: CONNECTING DEVICES OF WIFI APPLICATION TO ESP32

30.4.2 ESP32 SOFTAP:

AP/STA combination mode. We assign an IP to the SOFTAP; the interface of the ESP32 is statically configured. DHCP server dynamically assigns an IP for connecting stations. We set a maximum number of stations allowed to connect.

Figure 30.4.13: ESP32 SOFTAP/STATION OF WIFI APPLICATIONS

30.4.3 ACCESS POINT:

When the ESP32 connects to an AP, local time can be obtained utilizing SNTP (if the AP is connected to the internet). DHCP service from the AP, will dynamically assign an IP to the ESP32 in our application.

Figure 30.4.14: ACCESS POINT OF CONNECTING DEVICES OF WIFI APPLICATION

CHAPTER#31 CODES OF WIFI CONNECT ESP

31.1 CODE OF WIFI_APP.H:

```
/*
 * wifi_app.h
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFA SHAHBAZ
 */
#ifndef MAIN_WIFI_APP_H_
#define MAIN_WIFI_APP_H_
#include "esp_netif.h"
// WiFi application settings
#define WIFI_AP_SSID                    "huzaifashahbaz"
        // AP name
#define WIFI_AP_PASSWORD                "huzaifa12345"
        // AP password
#define WIFI_AP_CHANNEL                 1
        // AP channel
#define WIFI_AP_SSID_HIDDEN             0
        // AP visibility
#define WIFI_AP_MAX_CONNECTIONS         5
        // AP max clients
#define WIFI_AP_BEACON_INTERVAL         100
        // AP beacon: 100 milliseconds recommended
```

```
#define WIFI_AP_IP                          "192.168.0.1"           // AP
default IP
#define WIFI_AP_GATEWAY                     "192.168.0.1"
        // AP default Gateway (should be the same as the IP)
#define WIFI_AP_NETMASK                     "255.255.255.0"
        // AP netmask
#define WIFI_AP_BANDWIDTH                   WIFI_BW_HT20
        // AP bandwidth 20 MHz (40 MHz is the other option)
#define WIFI_STA_POWER_SAVE                 WIFI_PS_NONE
        // Power save not used
#define MAX_SSID_LENGTH                     32
        // IEEE standard maximum
#define MAX_PASSWORD_LENGTH                     64
                // IEEE standard maximum
#define MAX_CONNECTION_RETRIES              5
        // Retry number on disconnect
// netif object for the Station and Access Point
extern esp_netif_t* esp_netif_sta;
extern esp_netif_t* esp_netif_ap;
/**
 * Message IDs for the WiFi application task
 * @note Expand this based on your application requirements.
 */
typedef enum wifi_app_message
{
        WIFI_APP_MSG_START_HTTP_SERVER = 0,
        WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER,
        WIFI_APP_MSG_STA_CONNECTED_GOT_IP,
        WIFI_APP_MSG_STA_DISCONNECTED,
} wifi_app_message_e;
/**
 * Structure for the message queue
 * @note Expand this based on application requirements e.g. add another
type and parameter as required
 */
```

```
typedef struct wifi_app_queue_message
{
	wifi_app_message_e msgID;
} wifi_app_queue_message_t;
/**
 * Sends a message to the queue
 * @param msgID message ID from the wifi_app_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue,
otherwise pdFALSE.
 * @note Expand the parameter list based on your requirements e.g. how
you've expanded the wifi_app_queue_message_t.
 */
BaseType_t wifi_app_send_message(wifi_app_message_e msgID);
/**
 * Starts the WiFi RTOS task
 */
void wifi_app_start(void);
/**
 * Gets the wifi configuration
 */
wifi_config_t* wifi_app_get_wifi_config(void);
#endif /* MAIN_WIFI_APP_H_ */
```

31.2 CODE OF WIFI_APP.C:

```
/*
 * wifi_app.c
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "freertos/task.h"
#include "esp_err.h"
```

```
#include "esp_log.h"
#include "esp_wifi.h"
#include "lwip/netdb.h"
#include "http_server.h"
#include "rgb_led.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG [] = "wifi_app";
// Used for returning the WiFi configuration
wifi_config_t *wifi_config = NULL;
// Used to track the number for retries when a connection attempt fails
static int g_retry_number;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t wifi_app_queue_handle;
// netif objects for the station and access point
esp_netif_t* esp_netif_sta = NULL;
esp_netif_t* esp_netif_ap  = NULL;
/**
 * WiFi application event handler
 * @param arg data, aside from event data, that is passed to the handler
when it is called
 * @param event_base the base id of the event to register the handler for
 * @param event_id the id fo the event to register the handler for
 * @param event_data event data
 */
static void wifi_app_event_handler(void *arg, esp_event_base_t
event_base, int32_t event_id, void *event_data)
{
	if (event_base == WIFI_EVENT)
	{
		switch (event_id)
		{
			case WIFI_EVENT_AP_START:
```

```
                        ESP_LOGI(TAG,
"WIFI_EVENT_AP_START");
                        break;
                case WIFI_EVENT_AP_STOP:
                        ESP_LOGI(TAG, "WIFI_EVENT_AP_STOP");
                        break;
                case WIFI_EVENT_AP_STACONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_AP_STACONNECTED");
                        break;
                case WIFI_EVENT_AP_STADISCONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_AP_STADISCONNECTED");
                        break;
                case WIFI_EVENT_STA_START:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_START");
                        break;
                case WIFI_EVENT_STA_CONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_CONNECTED");
                        break;
                case WIFI_EVENT_STA_DISCONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_DISCONNECTED");
                        wifi_event_sta_disconnected_t
*wifi_event_sta_disconnected =
(wifi_event_sta_disconnected_t*)malloc(sizeof(wifi_event_sta_disconn
ected_t));
                        *wifi_event_sta_disconnected =
*((wifi_event_sta_disconnected_t*)event_data);
                        printf("WIFI_EVENT_STA_DISCONNECTED,
reason code %d\n", wifi_event_sta_disconnected->reason);
                        if (g_retry_number <
MAX_CONNECTION_RETRIES)
```

```
                        {
                                esp_wifi_connect();
                                g_retry_number ++;
                        }
                        else
                        {

        wifi_app_send_message(WIFI_APP_MSG_STA_DISCONNECT
ED);
                        }
                        break;
                }
        }
        else if (event_base == IP_EVENT)
        {
                switch (event_id)
                {
                        case IP_EVENT_STA_GOT_IP:
                                ESP_LOGI(TAG, "IP_EVENT_STA_GOT_IP");

        wifi_app_send_message(WIFI_APP_MSG_STA_CONNECTED_
GOT_IP);
                                break;
                }
        }
}
/**
 * Initializes the WiFi application event handler for WiFi and IP events.
 */
static void wifi_app_event_handler_init(void)
{
        // Event loop for the WiFi driver
        ESP_ERROR_CHECK(esp_event_loop_create_default());
        // Event handler for the connection
        esp_event_handler_instance_t instance_wifi_event;
```

```
	esp_event_handler_instance_t instance_ip_event;
	ESP_ERROR_CHECK(esp_event_handler_instance_register(WIF
I_EVENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_wifi_event));
	ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_E
VENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_ip_event));
}
/**
 * Initializes the TCP stack and default WiFi configuration.
 */
static void wifi_app_default_wifi_init(void)
{
	// Initialize the TCP stack
	ESP_ERROR_CHECK(esp_netif_init());
	// Default WiFi config - operations must be in this order!
	wifi_init_config_t wifi_init_config =
WIFI_INIT_CONFIG_DEFAULT();
	ESP_ERROR_CHECK(esp_wifi_init(&wifi_init_config));
	ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_R
AM));
	esp_netif_sta = esp_netif_create_default_wifi_sta();
	esp_netif_ap = esp_netif_create_default_wifi_ap();
}

/**
 * Configures the WiFi access point settings and assigns the static IP to
the SoftAP.
 */
static void wifi_app_soft_ap_config(void)
{
	// SoftAP - WiFi access point configuration
	wifi_config_t ap_config =
	{
		.ap = {
```

```
                        .ssid = WIFI_AP_SSID,
                        .ssid_len = strlen(WIFI_AP_SSID),
                        .password = WIFI_AP_PASSWORD,
                        .channel = WIFI_AP_CHANNEL,
                        .ssid_hidden = WIFI_AP_SSID_HIDDEN,
                        .authmode = WIFI_AUTH_WPA2_PSK,
                        .max_connection =
WIFI_AP_MAX_CONNECTIONS,
                        .beacon_interval =
WIFI_AP_BEACON_INTERVAL,
            },
        };
        // Configure DHCP for the AP
        esp_netif_ip_info_t ap_ip_info;
        memset(&ap_ip_info, 0x00, sizeof(ap_ip_info));
        esp_netif_dhcps_stop(esp_netif_ap);                         ///>
must call this first
        inet_pton(AF_INET, WIFI_AP_IP, &ap_ip_info.ip);            ///>
Assign access point's static IP, GW, and netmask
        inet_pton(AF_INET, WIFI_AP_GATEWAY, &ap_ip_info.gw);
        inet_pton(AF_INET, WIFI_AP_NETMASK,
&ap_ip_info.netmask);
        ESP_ERROR_CHECK(esp_netif_set_ip_info(esp_netif_ap,
&ap_ip_info));                  ///> Statically configure the network
interface
        ESP_ERROR_CHECK(esp_netif_dhcps_start(esp_netif_ap));
                                ///> Start the AP DHCP server (for
connecting stations e.g. your mobile device)
        ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_APST
A));                                ///> Setting the mode as Access Point /
Station Mode
        ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP,
&ap_config));                   ///> Set our configuration
```

```
	ESP_ERROR_CHECK(esp_wifi_set_bandwidth(WIFI_IF_AP,
WIFI_AP_BANDWIDTH));		///> Our default bandwidth 20
MHz
	ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_STA_POWER_SA
VE));		///> Power save set to "NONE"
}
/**
 * Connects the ESP32 to an external AP using the updated station
configuration
 */
static void wifi_app_connect_sta(void)
{
	ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA,
wifi_app_get_wifi_config()));
	ESP_ERROR_CHECK(esp_wifi_connect());
}
/**
 * Main task for the WiFi application
 * @param pvParameters parameter which can be passed to the task
 */
static void wifi_app_task(void *pvParameters)
{
	wifi_app_queue_message_t msg;
	// Initialize the event handler
	wifi_app_event_handler_init();
	// Initialize the TCP/IP stack and WiFi config
	wifi_app_default_wifi_init();
	// SoftAP config
	wifi_app_soft_ap_config();
	// Start WiFi
	ESP_ERROR_CHECK(esp_wifi_start());
	// Send first event message
	wifi_app_send_message(WIFI_APP_MSG_START_HTTP_SERV
ER);
	for (;;)
```

```
	{
		if (xQueueReceive(wifi_app_queue_handle, &msg,
portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case
WIFI_APP_MSG_START_HTTP_SERVER:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_START_HTTP_SERVER");
					http_server_start();
					rgb_led_http_server_started();
					break;
				case
WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER");
					// Attempt a connection
					wifi_app_connect_sta();
					// Set current number of retries to zero
					g_retry_number = 0;
					// Let the HTTP server know about the
connection attempt

	http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_INIT);
					break;
				case
WIFI_APP_MSG_STA_CONNECTED_GOT_IP:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_CONNECTED_GOT_IP");
					rgb_led_wifi_connected();

	http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_SUCCESS);
```

```
                                break;
                        case WIFI_APP_MSG_STA_DISCONNECTED:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED");

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_FAIL);
                                break;
                        default:
                                break;
                }
            }
        }
}
BaseType_t wifi_app_send_message(wifi_app_message_e msgID)
{
        wifi_app_queue_message_t msg;
        msg.msgID = msgID;
        return xQueueSend(wifi_app_queue_handle, &msg,
portMAX_DELAY);
}
wifi_config_t* wifi_app_get_wifi_config(void)
{
        return wifi_config;
}
void wifi_app_start(void)
{
        ESP_LOGI(TAG, "STARTING WIFI APPLICATION");
        // Start WiFi started LED
        rgb_led_wifi_app_started();
        // Disable default WiFi logging messages
        esp_log_level_set("wifi", ESP_LOG_NONE);
        // Allocate memory for the wifi configuration
        wifi_config = (wifi_config_t*)malloc(sizeof(wifi_config_t));
        memset(wifi_config, 0x00, sizeof(wifi_config_t));
```

```
	// Create message queue
	wifi_app_queue_handle = xQueueCreate(3,
sizeof(wifi_app_queue_message_t));
	// Start the WiFi application task
	xTaskCreatePinnedToCore(&wifi_app_task, "wifi_app_task",
WIFI_APP_TASK_STACK_SIZE, NULL,
WIFI_APP_TASK_PRIORITY, NULL,
WIFI_APP_TASK_CORE_ID);
}
```

31.3 CODE OF HTTP_SERVER.H:

```
/*
 * http_server.h
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_HTTP_SERVER_H_
#define MAIN_HTTP_SERVER_H_
#define OTA_UPDATE_PENDING		0
#define OTA_UPDATE_SUCCESSFUL	1
#define OTA_UPDATE_FAILED		-1
/**
 * Connection status for Wifi
 */
typedef enum http_server_wifi_connect_status
{
	NONE = 0,
	HTTP_WIFI_STATUS_CONNECTING,
	HTTP_WIFI_STATUS_CONNECT_FAILED,
	HTTP_WIFI_STATUS_CONNECT_SUCCESS,
} http_server_wifi_connect_status_e;
/**
 * Messages for the HTTP monitor
```

```
 */
typedef enum http_server_message
{
	HTTP_MSG_WIFI_CONNECT_INIT = 0,
	HTTP_MSG_WIFI_CONNECT_SUCCESS,
	HTTP_MSG_WIFI_CONNECT_FAIL,
	HTTP_MSG_OTA_UPDATE_SUCCESSFUL,
	HTTP_MSG_OTA_UPDATE_FAILED,
} http_server_message_e;
/**
 * Structure for the message queue
 */
typedef struct http_server_queue_message
{
	http_server_message_e msgID;
} http_server_queue_message_t;
/**
 * Sends a message to the queue
 * @param msgID message ID from the http_server_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue,
otherwise pdFALSE.
 * @note Expand the parameter list based on your requirements e.g. how
you've expanded the http_server_queue_message_t.
 */
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID);
/**
 * Starts the HTTP server.
 */
void http_server_start(void);
/**
 * Stops the HTTP server.
 */
void http_server_stop(void);
/**
```

```
 * Timer callback function which calls esp_restart upon successful
firmware update.
 */
void http_server_fw_update_reset_callback(void *arg);
#endif /* MAIN_HTTP_SERVER_H_ */
```

31.4 CODE OF HTTP_SERVER.C:

```
/*
 * http_server.c
 *
 *  Created on: Jul 11, 2024
 *      Author:HUZAIFASHAHBAZ
 */
 #include "esp_http_server.h"
#include "esp_log.h"
#include "esp_ota_ops.h"
#include "sys/param.h"
#include "DHT22.h"
#include "http_server.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG[] = "http_server";
// Wifi connect status
static int g_wifi_connect_status = NONE;
// Firmware update status
static int g_fw_update_status = OTA_UPDATE_PENDING;
// HTTP server task handle
static httpd_handle_t http_server_handle = NULL;
// HTTP server monitor task handle
static TaskHandle_t task_http_server_monitor = NULL;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t http_server_monitor_queue_handle;
/**
```

```
 * ESP32 timer configuration passed to esp_timer_create.
 */
const esp_timer_create_args_t fw_update_reset_args = {
            .callback = &http_server_fw_update_reset_callback,
            .arg = NULL,
            .dispatch_method = ESP_TIMER_TASK,
            .name = "fw_update_reset"
};
esp_timer_handle_t fw_update_reset;
// Embedded files: JQuery, index.html, app.css, app.js and favicon.ico
files
extern const uint8_t jquery_3_3_1_min_js_start[]
        asm("_binary_jquery_3_3_1_min_js_start");
extern const uint8_t jquery_3_3_1_min_js_end[]
        asm("_binary_jquery_3_3_1_min_js_end");
extern const uint8_t index_html_start[]
        asm("_binary_index_html_start");
extern const uint8_t index_html_end[]
        asm("_binary_index_html_end");
extern const uint8_t app_css_start[]
        asm("_binary_app_css_start");
extern const uint8_t app_css_end[]
        asm("_binary_app_css_end");
extern const uint8_t app_js_start[]
        asm("_binary_app_js_start");
extern const uint8_t app_js_end[]
        asm("_binary_app_js_end");
extern const uint8_t favicon_ico_start[]
        asm("_binary_favicon_ico_start");
extern const uint8_t favicon_ico_end[]
        asm("_binary_favicon_ico_end");
/**
 * Checks the g_fw_update_status and creates the fw_update_reset timer
if g_fw_update_status is true.
 */
```

```
static void http_server_fw_update_reset_timer(void)
{
	if (g_fw_update_status == OTA_UPDATE_SUCCESSFUL)
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
updated successful starting FW update reset timer");
		// Give the web page a chance to receive an acknowledge
back and initialize the timer

	ESP_ERROR_CHECK(esp_timer_create(&fw_update_reset_args,
&fw_update_reset));

	ESP_ERROR_CHECK(esp_timer_start_once(fw_update_reset,
8000000));
	}
	else
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
update unsuccessful");
	}
}
/**
 * HTTP server monitor task used to track events of the HTTP server
 * @param pvParameters parameter which can be passed to the task.
 */
static void http_server_monitor(void *parameter)
{
	http_server_queue_message_t msg;
	for (;;)
	{
		if (xQueueReceive(http_server_monitor_queue_handle,
&msg, portMAX_DELAY))
		{
			switch (msg.msgID)
			{
```

```
                        case HTTP_MSG_WIFI_CONNECT_INIT:
                                ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_INIT");
                                g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECTING;
                                break;
                        case HTTP_MSG_WIFI_CONNECT_SUCCESS:
                                ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_SUCCESS");
                                g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_SUCCESS;
                                break;
                        case HTTP_MSG_WIFI_CONNECT_FAIL:
                                ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_FAIL");
                                g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_FAILED;
                                break;
                        case
HTTP_MSG_OTA_UPDATE_SUCCESSFUL:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_SUCCESSFUL");
                                g_fw_update_status =
OTA_UPDATE_SUCCESSFUL;
                                http_server_fw_update_reset_timer();
                                break;
                        case HTTP_MSG_OTA_UPDATE_FAILED:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_FAILED");
                                g_fw_update_status =
OTA_UPDATE_FAILED;
                                break;
                        default:
                                break;
                }
```

```
            }
        }
}
/**
 * Jquery get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_jquery_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "Jquery requested");
        httpd_resp_set_type(req, "application/javascript");
        httpd_resp_send(req, (const char *)jquery_3_3_1_min_js_start,
jquery_3_3_1_min_js_end - jquery_3_3_1_min_js_start);
        return ESP_OK;
}
/**
 * Sends the index.html page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_index_html_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "index.html requested");
        httpd_resp_set_type(req, "text/html");
        httpd_resp_send(req, (const char *)index_html_start,
index_html_end - index_html_start);
        return ESP_OK;
}
/**
 * app.css get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_css_handler(httpd_req_t *req)
```

```
{
	ESP_LOGI(TAG, "app.css requested");
	httpd_resp_set_type(req, "text/css");
	httpd_resp_send(req, (const char *)app_css_start, app_css_end -
app_css_start);
	return ESP_OK;
}
/**
 * app.js get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_js_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.js requested");
	httpd_resp_set_type(req, "application/javascript");
	httpd_resp_send(req, (const char *)app_js_start, app_js_end -
app_js_start);
	return ESP_OK;
}
/**
 * Sends the .ico (icon) file when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_favicon_ico_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "favicon.ico requested");
	httpd_resp_set_type(req, "image/x-icon");
	httpd_resp_send(req, (const char *)favicon_ico_start,
favicon_ico_end - favicon_ico_start);
	return ESP_OK;
}
/**
```

```
 * Receives the .bin file fia the web page and handles the firmware
update
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK, otherwise ESP_FAIL if timeout occurs and the
update cannot be started.
 */
esp_err_t http_server_OTA_update_handler(httpd_req_t *req)
{
	esp_ota_handle_t ota_handle;
	char ota_buff[1024];
	int content_length = req->content_len;
	int content_received = 0;
	int recv_len;
	bool is_req_body_started = false;
	bool flash_successful = false;
	const esp_partition_t *update_partition =
esp_ota_get_next_update_partition(NULL);
	do
	{
		// Read the data for the request
		if ((recv_len = httpd_req_recv(req, ota_buff,
MIN(content_length, sizeof(ota_buff)))) < 0)
		{
			// Check if timeout occurred
			if (recv_len == HTTPD_SOCK_ERR_TIMEOUT)
			{
				ESP_LOGI(TAG,
"http_server_OTA_update_handler: Socket Timeout");
				continue; ///> Retry receiving if timeout occurred
			}
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
OTA other Error %d", recv_len);
			return ESP_FAIL;
		}
```

```
		printf("http_server_OTA_update_handler: OTA RX: %d of
%d\r", content_received, content_length);
		// Is this the first data we are receiving
		// If so, it will have the information in the header that we
need.
		if (!is_req_body_started)
		{
			is_req_body_started = true;
			// Get the location of the .bin file content (remove the
web form data)
			char *body_start_p = strstr(ota_buff, "\r\n\r\n") + 4;
			int body_part_len = recv_len - (body_start_p -
ota_buff);
			printf("http_server_OTA_update_handler: OTA file
size: %d\r\n", content_length);
			esp_err_t err = esp_ota_begin(update_partition,
OTA_SIZE_UNKNOWN, &ota_handle);
			if (err != ESP_OK)
			{
				printf("http_server_OTA_update_handler: Error
with OTA begin, cancelling OTA\r\n");
				return ESP_FAIL;
			}
			else
			{
				printf("http_server_OTA_update_handler:
Writing to partition subtype %d at offset 0x%x\r\n", update_partition-
>subtype, update_partition->address);
			}
			// Write this first part of the data
			esp_ota_write(ota_handle, body_start_p,
body_part_len);
			content_received += body_part_len;
		}
		else
```

```
		{
			// Write OTA data
			esp_ota_write(ota_handle, ota_buff, recv_len);
			content_received += recv_len;
		}
	} while (recv_len > 0 && content_received < content_length);
	if (esp_ota_end(ota_handle) == ESP_OK)
	{
		// Lets update the partition
		if (esp_ota_set_boot_partition(update_partition) ==
ESP_OK)
		{
			const esp_partition_t *boot_partition =
esp_ota_get_boot_partition();
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
Next boot partition subtype %d at offset 0x%x", boot_partition-
>subtype, boot_partition->address);
			flash_successful = true;
		}
		else
		{
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
FLASHED ERROR!!!");
		}
	}
	else
	{
		ESP_LOGI(TAG, "http_server_OTA_update_handler:
esp_ota_end ERROR!!!");
	}
	// We won't update the global variables throughout the file, so send
the message about the status
	if (flash_successful) {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_SU
CCESSFUL); } else {
```

```
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_FAI
LED); }
	return ESP_OK;
}
/**
 * OTA status handler responds with the firmware update status after the
OTA update is started
 * and responds with the compile time/date when the page is first
requested
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
esp_err_t http_server_OTA_status_handler(httpd_req_t *req)
{
	char otaJSON[100];
	ESP_LOGI(TAG, "OTAstatus requested");
	sprintf(otaJSON,
"{\"ota_update_status\":%d,\"compile_time\":\"%s\",\"compile_date\":\"
%s\"}", g_fw_update_status, __TIME__, __DATE__);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, otaJSON, strlen(otaJSON));

	return ESP_OK;
}
/**
 * DHT sensor readings JSON handler responds with DHT22 sensor data
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
static esp_err_t
http_server_get_dht_sensor_readings_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/dhtSensor.json requested");
	char dhtSensorJSON[100];
```

```
	sprintf(dhtSensorJSON,
"{\"temp\":\"%.1f\",\"humidity\":\"%.1f\"}", getTemperature(),
getHumidity());
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, dhtSensorJSON, strlen(dhtSensorJSON));

	return ESP_OK;
}
/**
 * wifiConnect.json handler is invoked after the connect button is
pressed
 * and handles receiving the SSID and password entered by the user
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_wifi_connect_json_handler(httpd_req_t
*req)
{
	ESP_LOGI(TAG, "/wifiConnect.json requested");
	size_t len_ssid = 0, len_pass = 0;
	char *ssid_str = NULL, *pass_str = NULL;
	// Get SSID header
	len_ssid = httpd_req_get_hdr_value_len(req, "my-connect-ssid") +
1;
	if (len_ssid > 1)
	{
		ssid_str = malloc(len_ssid);
		if (httpd_req_get_hdr_value_str(req, "my-connect-ssid",
ssid_str, len_ssid) == ESP_OK)
		{
			ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
ssid: %s", ssid_str);
		}
	}
```

```
	// Get Password header
	len_pass = httpd_req_get_hdr_value_len(req, "my-connect-pwd")
+ 1;
	if (len_pass > 1)
	{
		pass_str = malloc(len_pass);
		if (httpd_req_get_hdr_value_str(req, "my-connect-pwd",
pass_str, len_pass) == ESP_OK)
		{
			ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
pwd: %s", pass_str);
		}
	}
	// Update the Wifi networks configuration and let the wifi
application know
	wifi_config_t* wifi_config = wifi_app_get_wifi_config();
	memset(wifi_config, 0x00, sizeof(wifi_config_t));
	memcpy(wifi_config->sta.ssid, ssid_str, len_ssid);
	memcpy(wifi_config->sta.password, pass_str, len_pass);
	wifi_app_send_message(WIFI_APP_MSG_CONNECTING_FRO
M_HTTP_SERVER);
	free(ssid_str);
	free(pass_str);
	return ESP_OK;
}
/**
 * wifiConnectStatus handler updates the connection status for the web
page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t
http_server_wifi_connect_status_json_handler(httpd_req_t *req)
{
```

```
	ESP_LOGI(TAG, "/wifiConnectStatus requested");
	char statusJSON[100];
	sprintf(statusJSON, "{\"wifi_connect_status\":%d}",
g_wifi_connect_status);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, statusJSON, strlen(statusJSON));
	return ESP_OK;
}
/**
 * Sets up the default httpd server configuration.
 * @return http server instance handle if successful, NULL otherwise.
 */
static httpd_handle_t http_server_configure(void)
{
	// Generate the default configuration
	httpd_config_t config = HTTPD_DEFAULT_CONFIG();
	// Create HTTP server monitor task
	xTaskCreatePinnedToCore(&http_server_monitor,
"http_server_monitor", HTTP_SERVER_MONITOR_STACK_SIZE,
NULL, HTTP_SERVER_MONITOR_PRIORITY,
&task_http_server_monitor, HTTP_SERVER_MONITOR_CORE_ID);
	// Create the message queue
	http_server_monitor_queue_handle = xQueueCreate(3,
sizeof(http_server_queue_message_t));
	// The core that the HTTP server will run on
	config.core_id = HTTP_SERVER_TASK_CORE_ID;
	// Adjust the default priority to 1 less than the wifi application task
	config.task_priority = HTTP_SERVER_TASK_PRIORITY;
	// Bump up the stack size (default is 4096)
	config.stack_size = HTTP_SERVER_TASK_STACK_SIZE;
	// Increase uri handlers
	config.max_uri_handlers = 20;
	// Increase the timeout limits
	config.recv_wait_timeout = 10;
	config.send_wait_timeout = 10;
```

```
	ESP_LOGI(TAG,
		"http_server_configure: Starting server on port: '%d'
with task priority: '%d'",
		config.server_port,
		config.task_priority);
	// Start the httpd server
	if (httpd_start(&http_server_handle, &config) == ESP_OK)
	{
		ESP_LOGI(TAG, "http_server_configure: Registering URI
handlers");
		// register query handler
		httpd_uri_t jquery_js = {
			.uri = "/jquery-3.3.1.min.js",
			.method = HTTP_GET,
			.handler = http_server_jquery_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &jquery_js);
		// register index.html handler
		httpd_uri_t index_html = {
			.uri = "/",
			.method = HTTP_GET,
			.handler = http_server_index_html_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&index_html);
		// register app.css handler
		httpd_uri_t app_css = {
			.uri = "/app.css",
			.method = HTTP_GET,
			.handler = http_server_app_css_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &app_css);
```

```
        // register app.js handler
        httpd_uri_t app_js = {
                .uri = "/app.js",
                .method = HTTP_GET,
                .handler = http_server_app_js_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_js);
        // register favicon.ico handler
        httpd_uri_t favicon_ico = {
                .uri = "/favicon.ico",
                .method = HTTP_GET,
                .handler = http_server_favicon_ico_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&favicon_ico);
        // register OTAupdate handler
        httpd_uri_t OTA_update = {
                .uri = "/OTAupdate",
                .method = HTTP_POST,
                .handler = http_server_OTA_update_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_update);
        // register OTAstatus handler
        httpd_uri_t OTA_status = {
                .uri = "/OTAstatus",
                .method = HTTP_POST,
                .handler = http_server_OTA_status_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_status);
```

```
		// register dhtSensor.json handler
		httpd_uri_t dht_sensor_json = {
			.uri = "/dhtSensor.json",
			.method = HTTP_GET,
			.handler =
http_server_get_dht_sensor_readings_json_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&dht_sensor_json);
		// register wifiConnect.json handler
		httpd_uri_t wifi_connect_json = {
			.uri = "/wifiConnect.json",
			.method = HTTP_POST,
			.handler =
http_server_wifi_connect_json_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&wifi_connect_json);
		// register wifiConnectStatus.json handler
		httpd_uri_t wifi_connect_status_json = {
			.uri = "/wifiConnectStatus",
			.method = HTTP_POST,
			.handler =
http_server_wifi_connect_status_json_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&wifi_connect_status_json);
		return http_server_handle;
	}
	return NULL;
}
void http_server_start(void)
```

```
{
	if (http_server_handle == NULL)
	{
		http_server_handle = http_server_configure();
	}
}
void http_server_stop(void)
{
	if (http_server_handle)
	{
		httpd_stop(http_server_handle);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP
server");
		http_server_handle = NULL;
	}
	if (task_http_server_monitor)
	{
		vTaskDelete(task_http_server_monitor);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP server
monitor");
		task_http_server_monitor = NULL;
	}
}
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID)
{
	http_server_queue_message_t msg;
	msg.msgID = msgID;
	return xQueueSend(http_server_monitor_queue_handle, &msg,
portMAX_DELAY);
}
void http_server_fw_update_reset_callback(void *arg)
{
	ESP_LOGI(TAG, "http_server_fw_update_reset_callback: Timer
timed-out, restarting the device");
```

```
	esp_restart();
}
```

31.5 CODE OF INDEX_HTML:

```
<!DOCTYPE html>
<html lang="en">
	<head>
		<meta charset="utf-8"/>
		<meta name="viewport" content="width=device-width,
initial-scale=1.0, user-scalable=no">
		<meta name="apple-mobile-web-app-capable" content="yes"
/>
		<script src='jquery-3.3.1.min.js'></script>
		<link rel="stylesheet" href="app.css">
		<script async src="app.js"></script>
		<title>ESP32 Udemy Course</title>
	</head>
	<body>
	<header>
		<h1>ESP32 Application Development</h1>
	</header>
	<div id="OTA">
	<h2>ESP32 Firmware Update</h2>
		<label id="latest_firmware_label">Latest Firmware: </label>
		<div id="latest_firmware"></div>
		<input type="file" id="selected_file" accept=".bin"
style="display: none;" onchange="getFileInfo()" />
		<div class="buttons">
			<input type="button" value="Select File"
onclick="document.getElementById('selected_file').click();" />
			<input type="button" value="Update Firmware"
onclick="updateFirmware()" />
		</div>
```

```
            <h4 id="file_info"></h4>
            <h4 id="ota_update_status"></h4>
        </div>
        <hr>
        <div id="DHT22Sensor">
            <h2>DHT22 Sensor Readings</h2>
            <label for="temperature_reading">Temperature: </label>
            <div id="temperature_reading"></div>
            <label for="humidity_reading">Humidity: </label>
            <div id="humidity_reading"></div>
        </div>
        <hr>

        <div id="WiFiConnect">
            <h2>ESP32 WiFi Connect</h2>
            <section>
                <input id="connect_ssid" type="text" maxlength="32"
placeholder="SSID" value="">
                <input id="connect_pass" type="password"
maxlength="64" placeholder="Password" value="">
                <input type="checkbox"
onclick="showPassword()">Show Password
            </section>
            <div class="buttons">
                <input id="connect_wifi" type="button"
value="Connect" />
            </div>
            <div id="wifi_connect_credentials_errors"></div>
            <h4 id="wifi_connect_status"></h4>
        </div>

        </body>
<html>
```

31.6 CODE OF APP.CSS:

```
body {
   background-color: #f1f1f1;
   font-family: HelveticaNeueRegular, HelveticaNeue-Regular,
"Helvetica Neue Regular", HelveticaNeue, "Helvetica Neue", Helvetica,
Arial;
   color: #0272B7;
}
h1 {
   display: block;
   text-align: center;
   margin: 0;
   padding: 10px;
   font-size: 1.7em;
   color: #1d3557;
}
h2 {
      font-size: 1.4em;
   color: #1d3557;
}
h3 {
      font-size: 1.2em;
   color: #1d3557;
}
h4 {
      font-size: 1em;
   color: #1d3557;
}
.gr {
      color: green;
}
.rd {
      color: red;
}
```

```
#latest_firmware, #latest_firmware_label {
        display:inline;
}
#latest_firmware {
        color: #1d3557;
}
#temperature_reading {
        display:inline;
        color: #1d3557;
}
#humidity_reading {
        display:inline;
        color: #1d3557;
}
.buttons {
   padding: 5px;
}
input[type="button"] {
        font-size: 12px;
        border: 0;
   line-height: 2;
   padding: 0 5px;
   text-align: center;
   color: #ffffff;
   text-shadow: 1px 1px 1px #000;
   border-radius: 10px;
   background-color: RGBA(106,125,144,0.35);
   background-image: linear-gradient(to top left,
                                 rgba(0, 0, 0, .2),
                                 rgba(0, 0, 0, .2) 30%,
                                 rgba(0, 0, 0, 0));
   box-shadow: inset 2px 2px 3px rgba(255, 255, 255, .6),
             inset -2px -2px 3px rgba(0, 0, 0, .6);
}
```

```
input[type="button"]:hover {
   background-color: RGBA(112,164,222,0.68);
}

input[type="button"]:active {
   box-shadow: inset -2px -2px 3px rgba(255, 255, 255, .6),
               inset 2px 2px 3px rgba(0, 0, 0, .6);
}
```

31.7 CODE OF APP.JS:

```
/**
 * Add gobals here
 */
var seconds        = null;
var otaTimerVar =  null;
var wifiConnectInterval = null;
/**
 * Initialize functions here.
 */
$(document).ready(function(){
        getUpdateStatus();
        startDHTSensorInterval();
        $("#connect_wifi").on("click", function(){
                checkCredentials();
        });
});
/**
 * Gets file name and size for display on the web page.
 */
function getFileInfo()
{
   var x = document.getElementById("selected_file");
   var file = x.files[0];
```

```
   document.getElementById("file_info").innerHTML = "<h4>File: " +
file.name + "<br>" + "Size: " + file.size + " bytes</h4>";
}
/**
 * Handles the firmware update.
 */
function updateFirmware()
{
   // Form Data
   var formData = new FormData();
   var fileSelect = document.getElementById("selected_file");
   if (fileSelect.files && fileSelect.files.length == 1)
       {
       var file = fileSelect.files[0];
       formData.set("file", file, file.name);
       document.getElementById("ota_update_status").innerHTML =
"Uploading " + file.name + ", Firmware Update in Progress...";
       // Http Request
       var request = new XMLHttpRequest();
       request.upload.addEventListener("progress", updateProgress);
       request.open('POST', "/OTAupdate");
       request.responseType = "blob";
       request.send(formData);
   }
        else
       {
       window.alert('Select A File First')
   }
}
/**
 * Progress on transfers from the server to the client (downloads).
 */
function updateProgress(oEvent)
{
   if (oEvent.lengthComputable)
```

```
        {
        getUpdateStatus();
    }
         else
        {
        window.alert('total size is unknown')
    }
}
/**
 * Posts the firmware udpate status.
 */
function getUpdateStatus()
{
    var xhr = new XMLHttpRequest();
    var requestURL = "/OTAstatus";
    xhr.open('POST', requestURL, false);
    xhr.send('ota_update_status');
    if (xhr.readyState == 4 && xhr.status == 200)
        {
        var response = JSON.parse(xhr.responseText);
                document.getElementById("latest_firmware").innerHTML =
response.compile_date + " - " + response.compile_time
                // If flashing was complete it will return a 1, else -1
                // A return of 0 is just for information on the Latest Firmware
request
        if (response.ota_update_status == 1)
                {
                // Set the countdown timer time
            seconds = 10;
            // Start the countdown timer
            otaRebootTimer();
        }
        else if (response.ota_update_status == -1)
                {
```

```
            document.getElementById("ota_update_status").innerHTML =
"!!! Upload Error !!!";
        }
    }
}
/**
 * Displays the reboot countdown.
 */
function otaRebootTimer()
{
    document.getElementById("ota_update_status").innerHTML = "OTA
Firmware Update Complete. This page will close shortly, Rebooting in:
" + seconds;
    if (--seconds == 0)
         {
        clearTimeout(otaTimerVar);
        window.location.reload();
    }
         else
         {
        otaTimerVar = setTimeout(otaRebootTimer, 1000);
    }
}

/**
 * Gets DHT22 sensor temperature and humidity values for display on
the web page.
 */
function getDHTSensorValues()
{
        $.getJSON('/dhtSensor.json', function(data) {
                $("#temperature_reading").text(data["temp"]);
                $("#humidity_reading").text(data["humidity"]);
        });
}
```

```
/**
 * Sets the interval for getting the updated DHT22 sensor values.
 */
function startDHTSensorInterval()
{
	setInterval(getDHTSensorValues, 5000);
}
/**
 * Clears the connection status interval.
 */
function stopWifiConnectStatusInterval()
{
	if (wifiConnectInterval != null)
	{
		clearInterval(wifiConnectInterval);
		wifiConnectInterval = null;
	}
}
/**
 * Gets the WiFi connection status.
 */
function getWifiConnectStatus()
{
	var xhr = new XMLHttpRequest();
	var requestURL = "/wifiConnectStatus";
	xhr.open('POST', requestURL, false);
	xhr.send('wifi_connect_status');
	if (xhr.readyState == 4 && xhr.status == 200)
	{
		var response = JSON.parse(xhr.responseText);

	document.getElementById("wifi_connect_status").innerHTML =
"Connecting...";
		if (response.wifi_connect_status == 2)
```

```
		{

	document.getElementById("wifi_connect_status").innerHTML =
"<h4 class='rd'>Failed to Connect. Please check your AP credentials and
compatibility</h4>";
			stopWifiConnectStatusInterval();
		}
		else if (response.wifi_connect_status == 3)
		{

	document.getElementById("wifi_connect_status").innerHTML =
"<h4 class='gr'>Connection Success!</h4>";
			stopWifiConnectStatusInterval();
		}
	}
}
/**
 * Starts the interval for checking the connection status.
 */
function startWifiConnectStatusInterval()
{
	wifiConnectInterval = setInterval(getWifiConnectStatus, 2800);
}
/**
 * Connect WiFi function called using the SSID and password entered
into the text fields.
 */
function connectWifi()
{
	// Get the SSID and password
	selectedSSID = $("#connect_ssid").val();
	pwd = $("#connect_pass").val();
	$.ajax({
		url: '/wifiConnect.json',
		dataType: 'json',
```

```
		method: 'POST',
		cache: false,
		headers: {'my-connect-ssid': selectedSSID, 'my-connect-
pwd': pwd},
		data: {'timestamp': Date.now()}
	});
	startWifiConnectStatusInterval();
}
/**
 * Checks credentials on connect_wifi button click.
 */
function checkCredentials()
{
	errorList = "";
	credsOk = true;

	selectedSSID = $("#connect_ssid").val();
	pwd = $("#connect_pass").val();

	if (selectedSSID == "")
	{
		errorList += "<h4 class='rd'>SSID cannot be empty!</h4>";
		credsOk = false;
	}
	if (pwd == "")
	{
		errorList += "<h4 class='rd'>Password cannot be
empty!</h4>";
		credsOk = false;
	}

	if (credsOk == false)
	{
		$("#wifi_connect_credentials_errors").html(errorList);
	}
```

```
        else
        {
                $("#wifi_connect_credentials_errors").html("");
                connectWifi();
        }
}
/**
 * Shows the WiFi password if the box is checked.
 */
function showPassword()
{
        var x = document.getElementById("connect_pass");
        if (x.type === "password")
        {
                x.type = "text";
        }
        else
        {
                x.type = "password";
        }
}
```

CHAPTER#32 DISPLAYING WIFI CONNECTION INFORMATION ON THE WEB PAGE

32.1 DISPLAYING WIFI CONNECTION INFORMATION ON THE WEB PAGE:

To display WiFi connection information on an ESP32 web page, set up the ESPAsyncWebServer to handle HTTP GET requests. Connect the ESP32 to WiFi using `WiFi.begin()`. Define a route "/wifi" that responds with an HTML page showing the SSID and IP address. Retrieve WiFi information using `WiFi.SSID()` and `WiFi.localIP().toString()`. Serve this data dynamically in the response using `request->send()`.

CHAPTER#33 CODE OF WIFI CONNECTION INFORMATION ON THE WEB PAGE:

33.1 CODE OF HTTP_SERVER.C:

```
/*
 * http_server.c
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include "esp_http_server.h"
#include "esp_log.h"
#include "esp_ota_ops.h"
#include "esp_wifi.h"
#include "sys/param.h"
#include "DHT22.h"
#include "http_server.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG[] = "http_server";
// Wifi connect status
static int g_wifi_connect_status = NONE;
// Firmware update status
static int g_fw_update_status = OTA_UPDATE_PENDING;
// HTTP server task handle
static httpd_handle_t http_server_handle = NULL;
// HTTP server monitor task handle
static TaskHandle_t task_http_server_monitor = NULL;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t http_server_monitor_queue_handle;
/**
 * ESP32 timer configuration passed to esp_timer_create.
```

```
 */
const esp_timer_create_args_t fw_update_reset_args = {
		.callback = &http_server_fw_update_reset_callback,
		.arg = NULL,
		.dispatch_method = ESP_TIMER_TASK,
		.name = "fw_update_reset"
};
esp_timer_handle_t fw_update_reset;
// Embedded files: JQuery, index.html, app.css, app.js and favicon.ico
files
extern const uint8_t jquery_3_3_1_min_js_start[]
	asm("_binary_jquery_3_3_1_min_js_start");
extern const uint8_t jquery_3_3_1_min_js_end[]
	asm("_binary_jquery_3_3_1_min_js_end");
extern const uint8_t index_html_start[]
	asm("_binary_index_html_start");
extern const uint8_t index_html_end[]
	asm("_binary_index_html_end");
extern const uint8_t app_css_start[]
	asm("_binary_app_css_start");
extern const uint8_t app_css_end[]
	asm("_binary_app_css_end");
extern const uint8_t app_js_start[]
	asm("_binary_app_js_start");
extern const uint8_t app_js_end[]
	asm("_binary_app_js_end");
extern const uint8_t favicon_ico_start[]
	asm("_binary_favicon_ico_start");
extern const uint8_t favicon_ico_end[]
	asm("_binary_favicon_ico_end");
/**
 * Checks the g_fw_update_status and creates the fw_update_reset timer
if g_fw_update_status is true.
 */
static void http_server_fw_update_reset_timer(void)
```

```
{
	if (g_fw_update_status == OTA_UPDATE_SUCCESSFUL)
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
updated successful starting FW update reset timer");
		// Give the web page a chance to receive an acknowledge
back and initialize the timer

	ESP_ERROR_CHECK(esp_timer_create(&fw_update_reset_args,
&fw_update_reset));

	ESP_ERROR_CHECK(esp_timer_start_once(fw_update_reset,
8000000));
	}
	else
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
update unsuccessful");
	}
}
/**
 * HTTP server monitor task used to track events of the HTTP server
 * @param pvParameters parameter which can be passed to the task.
 */
static void http_server_monitor(void *parameter)
{
	http_server_queue_message_t msg;
	for (;;)
	{
		if (xQueueReceive(http_server_monitor_queue_handle,
&msg, portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case HTTP_MSG_WIFI_CONNECT_INIT:
```

```
                        ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_INIT");
                        g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECTING;
                        break;
                    case HTTP_MSG_WIFI_CONNECT_SUCCESS:
                        ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_SUCCESS");
                        g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_SUCCESS;
                        break;
                    case HTTP_MSG_WIFI_CONNECT_FAIL:
                        ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_FAIL");
                        g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_FAILED;
                        break;
                    case
HTTP_MSG_OTA_UPDATE_SUCCESSFUL:
                        ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_SUCCESSFUL");
                        g_fw_update_status =
OTA_UPDATE_SUCCESSFUL;
                        http_server_fw_update_reset_timer();
                        break;
                    case HTTP_MSG_OTA_UPDATE_FAILED:
                        ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_FAILED");
                        g_fw_update_status =
OTA_UPDATE_FAILED;
                        break;
                    default:
                        break;
                }
            }
```

```
        }
}
/**
 * Jquery get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_jquery_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "Jquery requested");
        httpd_resp_set_type(req, "application/javascript");
        httpd_resp_send(req, (const char *)jquery_3_3_1_min_js_start,
jquery_3_3_1_min_js_end - jquery_3_3_1_min_js_start);
        return ESP_OK;
}
/**
 * Sends the index.html page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_index_html_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "index.html requested");
        httpd_resp_set_type(req, "text/html");
        httpd_resp_send(req, (const char *)index_html_start,
index_html_end - index_html_start);
        return ESP_OK;
}
/**
 * app.css get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_css_handler(httpd_req_t *req)
{
```

```
	ESP_LOGI(TAG, "app.css requested");
	httpd_resp_set_type(req, "text/css");
	httpd_resp_send(req, (const char *)app_css_start, app_css_end -
app_css_start);

	return ESP_OK;
}
/**
 * app.js get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_js_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.js requested");
	httpd_resp_set_type(req, "application/javascript");
	httpd_resp_send(req, (const char *)app_js_start, app_js_end -
app_js_start);
	return ESP_OK;
}
/**
 * Sends the .ico (icon) file when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_favicon_ico_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "favicon.ico requested");
	httpd_resp_set_type(req, "image/x-icon");
	httpd_resp_send(req, (const char *)favicon_ico_start,
favicon_ico_end - favicon_ico_start);
	return ESP_OK;
}
/**
```

```
 * Receives the .bin file fia the web page and handles the firmware
update
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK, otherwise ESP_FAIL if timeout occurs and the
update cannot be started.
 */
esp_err_t http_server_OTA_update_handler(httpd_req_t *req)
{
	esp_ota_handle_t ota_handle;
	char ota_buff[1024];
	int content_length = req->content_len;
	int content_received = 0;
	int recv_len;
	bool is_req_body_started = false;
	bool flash_successful = false;
	const esp_partition_t *update_partition =
esp_ota_get_next_update_partition(NULL);
	do
	{
		// Read the data for the request
		if ((recv_len = httpd_req_recv(req, ota_buff,
MIN(content_length, sizeof(ota_buff)))) < 0)
		{
			// Check if timeout occurred
			if (recv_len == HTTPD_SOCK_ERR_TIMEOUT)
			{
				ESP_LOGI(TAG,
"http_server_OTA_update_handler: Socket Timeout");
				continue; ///> Retry receiving if timeout occurred
			}
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
OTA other Error %d", recv_len);
			return ESP_FAIL;
		}
```

```
		printf("http_server_OTA_update_handler: OTA RX: %d of
%d\r", content_received, content_length);
		// Is this the first data we are receiving
		// If so, it will have the information in the header that we
need.
		if (!is_req_body_started)
		{
			is_req_body_started = true;
			// Get the location of the .bin file content (remove the
web form data)
			char *body_start_p = strstr(ota_buff, "\r\n\r\n") + 4;
			int body_part_len = recv_len - (body_start_p -
ota_buff);
			printf("http_server_OTA_update_handler: OTA file
size: %d\r\n", content_length);
			esp_err_t err = esp_ota_begin(update_partition,
OTA_SIZE_UNKNOWN, &ota_handle);
			if (err != ESP_OK)
			{
				printf("http_server_OTA_update_handler: Error
with OTA begin, cancelling OTA\r\n");
				return ESP_FAIL;
			}
			else
			{
				printf("http_server_OTA_update_handler:
Writing to partition subtype %d at offset 0x%x\r\n", update_partition-
>subtype, update_partition->address);
			}
			// Write this first part of the data
			esp_ota_write(ota_handle, body_start_p,
body_part_len);
			content_received += body_part_len;
		}
		else
```

```
		{
			// Write OTA data
			esp_ota_write(ota_handle, ota_buff, recv_len);
			content_received += recv_len;
		}

	} while (recv_len > 0 && content_received < content_length);

	if (esp_ota_end(ota_handle) == ESP_OK)
	{
		// Lets update the partition
		if (esp_ota_set_boot_partition(update_partition) ==
ESP_OK)
		{
			const esp_partition_t *boot_partition =
esp_ota_get_boot_partition();
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
Next boot partition subtype %d at offset 0x%x", boot_partition-
>subtype, boot_partition->address);
			flash_successful = true;
		}
		else
		{
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
FLASHED ERROR!!!");
		}
	}
	else
	{
		ESP_LOGI(TAG, "http_server_OTA_update_handler:
esp_ota_end ERROR!!!");
	}
	// We won't update the global variables throughout the file, so send
the message about the status
```

```
	if (flash_successful) {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_SU
CCESSFUL); } else {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_FAI
LED); }
	return ESP_OK;
}
/**
 * OTA status handler responds with the firmware update status after the
OTA update is started
 * and responds with the compile time/date when the page is first
requested
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
esp_err_t http_server_OTA_status_handler(httpd_req_t *req)
{
	char otaJSON[100];
	ESP_LOGI(TAG, "OTAstatus requested");
	sprintf(otaJSON,
"{\"ota_update_status\":%d,\"compile_time\":\"%s\",\"compile_date\":\"
%s\"}", g_fw_update_status, __TIME__, __DATE__);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, otaJSON, strlen(otaJSON));
	return ESP_OK;
}
/**
 * DHT sensor readings JSON handler responds with DHT22 sensor data
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
static esp_err_t
http_server_get_dht_sensor_readings_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/dhtSensor.json requested");
```

```
	char dhtSensorJSON[100];
	sprintf(dhtSensorJSON,
"{\"temp\":\"%.1f\",\"humidity\":\"%.1f\"}", getTemperature(),
getHumidity());
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, dhtSensorJSON, strlen(dhtSensorJSON));
	return ESP_OK;
}
/**
 * wifiConnect.json handler is invoked after the connect button is
pressed
 * and handles receiving the SSID and password entered by the user
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_wifi_connect_json_handler(httpd_req_t
*req)
{
	ESP_LOGI(TAG, "/wifiConnect.json requested");
	size_t len_ssid = 0, len_pass = 0;
	char *ssid_str = NULL, *pass_str = NULL;
	// Get SSID header
	len_ssid = httpd_req_get_hdr_value_len(req, "my-connect-ssid") +
1;
	if (len_ssid > 1)
	{
		ssid_str = malloc(len_ssid);
		if (httpd_req_get_hdr_value_str(req, "my-connect-ssid",
ssid_str, len_ssid) == ESP_OK)
		{
			ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
ssid: %s", ssid_str);
		}
	}
```

```
	// Get Password header
	len_pass = httpd_req_get_hdr_value_len(req, "my-connect-pwd")
+ 1;
	if (len_pass > 1)
	{
		pass_str = malloc(len_pass);
		if (httpd_req_get_hdr_value_str(req, "my-connect-pwd",
pass_str, len_pass) == ESP_OK)
		{
			ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
pwd: %s", pass_str);
		}
	}
	// Update the Wifi networks configuration and let the wifi
application know
	wifi_config_t* wifi_config = wifi_app_get_wifi_config();
	memset(wifi_config, 0x00, sizeof(wifi_config_t));
	memcpy(wifi_config->sta.ssid, ssid_str, len_ssid);
	memcpy(wifi_config->sta.password, pass_str, len_pass);
	wifi_app_send_message(WIFI_APP_MSG_CONNECTING_FRO
M_HTTP_SERVER);
	free(ssid_str);
	free(pass_str);
	return ESP_OK;
}
/**
 * wifiConnectStatus handler updates the connection status for the web
page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t
http_server_wifi_connect_status_json_handler(httpd_req_t *req)
{
```

```
	ESP_LOGI(TAG, "/wifiConnectStatus requested");
	char statusJSON[100];
	sprintf(statusJSON, "{\"wifi_connect_status\":%d}",
g_wifi_connect_status);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, statusJSON, strlen(statusJSON));
	return ESP_OK;
}
/**
 * wifiConnectInfo.json handler updates the web page with connection
information.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t
http_server_get_wifi_connect_info_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/wifiConnectInfo.json requested");
	char ipInfoJSON[200];
	memset(ipInfoJSON, 0, sizeof(ipInfoJSON));
	char ip[IP4ADDR_STRLEN_MAX];
	char netmask[IP4ADDR_STRLEN_MAX];
	char gw[IP4ADDR_STRLEN_MAX];
	if (g_wifi_connect_status ==
HTTP_WIFI_STATUS_CONNECT_SUCCESS)
	{
		wifi_ap_record_t wifi_data;

	ESP_ERROR_CHECK(esp_wifi_sta_get_ap_info(&wifi_data));
		char *ssid = (char*)wifi_data.ssid;
		esp_netif_ip_info_t ip_info;
		ESP_ERROR_CHECK(esp_netif_get_ip_info(esp_netif_sta,
&ip_info));
		esp_ip4addr_ntoa(&ip_info.ip, ip,
IP4ADDR_STRLEN_MAX);
```

```
		esp_ip4addr_ntoa(&ip_info.netmask, netmask,
IP4ADDR_STRLEN_MAX);
		esp_ip4addr_ntoa(&ip_info.gw, gw,
IP4ADDR_STRLEN_MAX);
		sprintf(ipInfoJSON,
"{\"ip\":\"%s\",\"netmask\":\"%s\",\"gw\":\"%s\",\"ap\":\"%s\"}", ip,
netmask, gw, ssid);
	}
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, ipInfoJSON, strlen(ipInfoJSON));
	return ESP_OK;
}
/**
 * Sets up the default httpd server configuration.
 * @return http server instance handle if successful, NULL otherwise.
 */
static httpd_handle_t http_server_configure(void)
{
	// Generate the default configuration
	httpd_config_t config = HTTPD_DEFAULT_CONFIG();
	// Create HTTP server monitor task
	xTaskCreatePinnedToCore(&http_server_monitor,
"http_server_monitor", HTTP_SERVER_MONITOR_STACK_SIZE,
NULL, HTTP_SERVER_MONITOR_PRIORITY,
&task_http_server_monitor, HTTP_SERVER_MONITOR_CORE_ID);
	// Create the message queue
	http_server_monitor_queue_handle = xQueueCreate(3,
sizeof(http_server_queue_message_t));
	// The core that the HTTP server will run on
	config.core_id = HTTP_SERVER_TASK_CORE_ID;
	// Adjust the default priority to 1 less than the wifi application task
	config.task_priority = HTTP_SERVER_TASK_PRIORITY;
	// Bump up the stack size (default is 4096)
	config.stack_size = HTTP_SERVER_TASK_STACK_SIZE;
	// Increase uri handlers
```

```
	config.max_uri_handlers = 20;
	// Increase the timeout limits
	config.recv_wait_timeout = 10;
	config.send_wait_timeout = 10;
	ESP_LOGI(TAG,
			"http_server_configure: Starting server on port: '%d'
with task priority: '%d'",
			config.server_port,
			config.task_priority);
	// Start the httpd server
	if (httpd_start(&http_server_handle, &config) == ESP_OK)
	{
		ESP_LOGI(TAG, "http_server_configure: Registering URI
handlers");
		// register query handler
		httpd_uri_t jquery_js = {
				.uri = "/jquery-3.3.1.min.js",
				.method = HTTP_GET,
				.handler = http_server_jquery_handler,
				.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &jquery_js);
		// register index.html handler
		httpd_uri_t index_html = {
				.uri = "/",
				.method = HTTP_GET,
				.handler = http_server_index_html_handler,
				.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&index_html);
		// register app.css handler
		httpd_uri_t app_css = {
				.uri = "/app.css",
				.method = HTTP_GET,
```

```
                .handler = http_server_app_css_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_css);
        // register app.js handler
        httpd_uri_t app_js = {
                .uri = "/app.js",
                .method = HTTP_GET,
                .handler = http_server_app_js_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_js);
        // register favicon.ico handler
        httpd_uri_t favicon_ico = {
                .uri = "/favicon.ico",
                .method = HTTP_GET,
                .handler = http_server_favicon_ico_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&favicon_ico);
        // register OTAupdate handler
        httpd_uri_t OTA_update = {
                .uri = "/OTAupdate",
                .method = HTTP_POST,
                .handler = http_server_OTA_update_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_update);
        // register OTAstatus handler
        httpd_uri_t OTA_status = {
                .uri = "/OTAstatus",
                .method = HTTP_POST,
                .handler = http_server_OTA_status_handler,
```

```
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_status);
        // register dhtSensor.json handler
        httpd_uri_t dht_sensor_json = {
                .uri = "/dhtSensor.json",
                .method = HTTP_GET,
                .handler =
http_server_get_dht_sensor_readings_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&dht_sensor_json);
        // register wifiConnect.json handler
        httpd_uri_t wifi_connect_json = {
                .uri = "/wifiConnect.json",
                .method = HTTP_POST,
                .handler =
http_server_wifi_connect_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_connect_json);
        // register wifiConnectStatus.json handler
        httpd_uri_t wifi_connect_status_json = {
                .uri = "/wifiConnectStatus",
                .method = HTTP_POST,
                .handler =
http_server_wifi_connect_status_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_connect_status_json);
        // register wifiConnectInfo.json handler
```

```
		httpd_uri_t wifi_connect_info_json = {
			.uri = "/wifiConnectInfo.json",
			.method = HTTP_GET,
			.handler =
http_server_get_wifi_connect_info_json_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&wifi_connect_info_json);
		return http_server_handle;
	}
	return NULL;
}
void http_server_start(void)
{
	if (http_server_handle == NULL)
	{
		http_server_handle = http_server_configure();
	}
}
void http_server_stop(void)
{
	if (http_server_handle)
	{
		httpd_stop(http_server_handle);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP
server");
		http_server_handle = NULL;
	}
	if (task_http_server_monitor)
	{
		vTaskDelete(task_http_server_monitor);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP server
monitor");
		task_http_server_monitor = NULL;
```

```
        }
}
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID)
{
        http_server_queue_message_t msg;
        msg.msgID = msgID;
        return xQueueSend(http_server_monitor_queue_handle, &msg,
portMAX_DELAY);
}
void http_server_fw_update_reset_callback(void *arg)
{
        ESP_LOGI(TAG, "http_server_fw_update_reset_callback: Timer
timed-out, restarting the device");
        esp_restart();
}
```

33.2 CODE OF APP.CSS:

```
body {
   background-color: #f1f1f1;
   font-family: HelveticaNeueRegular, HelveticaNeue-Regular,
"Helvetica Neue Regular", HelveticaNeue, "Helvetica Neue", Helvetica,
Arial;
   color: #0272B7;
}
h1 {
   display: block;
   text-align: center;
   margin: 0;
   padding: 10px;
   font-size: 1.7em;
   color: #1d3557;
}
h2 {
```

```
        font-size: 1.4em;
    color: #1d3557;
}
h3 {
        font-size: 1.2em;
    color: #1d3557;
}
h4 {
        font-size: 1em;
    color: #1d3557;
}
.gr {
        color: green;
}
.rd {
        color: red;
}
#latest_firmware, #latest_firmware_label {
        display:inline;
}
#latest_firmware {
        color: #1d3557;
}
#temperature_reading {
        display:inline;
        color: #1d3557;
}
#humidity_reading {
        display:inline;
        color: #1d3557;
}
#connected_ap_label, #connected_ap {
        display:inline;
}
#ip_address_label, #netmask_label, #gateway_label {
```

```
        display:inline;
}
#wifi_connect_ip, #wifi_connect_netmask, #wifi_connect_gw {
        display:inline;
}
#connected_ap {
        color: #1d3557;
}
#wifi_connect_ip {
        color: #1d3557;
}
#wifi_connect_netmask {
        color: #1d3557;
}
#wifi_connect_gw {
        color: #1d3557;
}
#disconnect_wifi {
        display: none;
}
.buttons {
   padding: 5px;
}
input[type="button"] {
        font-size: 12px;
        border: 0;
   line-height: 2;
   padding: 0 5px;
   text-align: center;
   color: #ffffff;
   text-shadow: 1px 1px 1px #000;
   border-radius: 10px;
   background-color: RGBA(106,125,144,0.35);
   background-image: linear-gradient(to top left,
                                     rgba(0, 0, 0, .2),
```

```
                                rgba(0, 0, 0, .2) 30%,
                                rgba(0, 0, 0, 0));
    box-shadow: inset 2px 2px 3px rgba(255, 255, 255, .6),
                inset -2px -2px 3px rgba(0, 0, 0, .6);
}
input[type="button"]:hover {
    background-color: RGBA(112,164,222,0.68);
}

input[type="button"]:active {
    box-shadow: inset -2px -2px 3px rgba(255, 255, 255, .6),
                inset 2px 2px 3px rgba(0, 0, 0, .6);
}
```

33.3 CODE OF APP.JS:

```
/**
 * Add gobals here
 */
var seconds        = null;
var otaTimerVar =  null;
var wifiConnectInterval = null;
/**
 * Initialize functions here.
 */
$(document).ready(function(){
        getUpdateStatus();
        startDHTSensorInterval();
        getConnectInfo();
        $("#connect_wifi").on("click", function(){
                checkCredentials();
        });
});
/**
 * Gets file name and size for display on the web page.
```

```
 */
function getFileInfo()
{
   var x = document.getElementById("selected_file");
   var file = x.files[0];
   document.getElementById("file_info").innerHTML = "<h4>File: " +
file.name + "<br>" + "Size: " + file.size + " bytes</h4>";
}
/**
 * Handles the firmware update.
 */
function updateFirmware()
{
   // Form Data
   var formData = new FormData();
   var fileSelect = document.getElementById("selected_file");
   if (fileSelect.files && fileSelect.files.length == 1)
        {
       var file = fileSelect.files[0];
       formData.set("file", file, file.name);
       document.getElementById("ota_update_status").innerHTML =
"Uploading " + file.name + ", Firmware Update in Progress...";
       // Http Request
       var request = new XMLHttpRequest();
       request.upload.addEventListener("progress", updateProgress);
       request.open('POST', "/OTAupdate");
       request.responseType = "blob";
       request.send(formData);
   }
        else
        {
       window.alert('Select A File First')
   }
}
/**
```

```
 * Progress on transfers from the server to the client (downloads).
 */
function updateProgress(oEvent)
{
   if (oEvent.lengthComputable)
        {
       getUpdateStatus();
   }
        else
        {
       window.alert('total size is unknown')
   }
}
/**
 * Posts the firmware udpate status.
 */
function getUpdateStatus()
{
   var xhr = new XMLHttpRequest();
   var requestURL = "/OTAstatus";
   xhr.open('POST', requestURL, false);
   xhr.send('ota_update_status');
   if (xhr.readyState == 4 && xhr.status == 200)
        {
       var response = JSON.parse(xhr.responseText);
               document.getElementById("latest_firmware").innerHTML =
response.compile_date + " - " + response.compile_time
               // If flashing was complete it will return a 1, else -1
               // A return of 0 is just for information on the Latest Firmware
request
       if (response.ota_update_status == 1)
               {
               // Set the countdown timer time
           seconds = 10;
           // Start the countdown timer
```

```
            otaRebootTimer();
        }
        else if (response.ota_update_status == -1)
                {
            document.getElementById("ota_update_status").innerHTML =
"!!! Upload Error !!!";
        }
    }
}
/**
 * Displays the reboot countdown.
 */
function otaRebootTimer()
{
    document.getElementById("ota_update_status").innerHTML = "OTA
Firmware Update Complete. This page will close shortly, Rebooting in:
" + seconds;
    if (--seconds == 0)
         {
        clearTimeout(otaTimerVar);
        window.location.reload();
    }
         else
         {
        otaTimerVar = setTimeout(otaRebootTimer, 1000);
    }
}
/**
 * Gets DHT22 sensor temperature and humidity values for display on
the web page.
 */
function getDHTSensorValues()
{
         $.getJSON('/dhtSensor.json', function(data) {
                $("#temperature_reading").text(data["temp"]);
```

```
		$("#humidity_reading").text(data["humidity"]);
	});
}
/**
 * Sets the interval for getting the updated DHT22 sensor values.
 */
function startDHTSensorInterval()
{
	setInterval(getDHTSensorValues, 5000);
}
/**
 * Clears the connection status interval.
 */
function stopWifiConnectStatusInterval()
{
	if (wifiConnectInterval != null)
	{
		clearInterval(wifiConnectInterval);
		wifiConnectInterval = null;
	}
}
/**
 * Gets the WiFi connection status.
 */
function getWifiConnectStatus()
{
	var xhr = new XMLHttpRequest();
	var requestURL = "/wifiConnectStatus";
	xhr.open('POST', requestURL, false);
	xhr.send('wifi_connect_status');
	if (xhr.readyState == 4 && xhr.status == 200)
	{
		var response = JSON.parse(xhr.responseText);
```

```
	document.getElementById("wifi_connect_status").innerHTML =
"Connecting...";
		if (response.wifi_connect_status == 2)
		{

	document.getElementById("wifi_connect_status").innerHTML =
"<h4 class='rd'>Failed to Connect. Please check your AP credentials and
compatibility</h4>";
			stopWifiConnectStatusInterval();
		}
		else if (response.wifi_connect_status == 3)
		{

	document.getElementById("wifi_connect_status").innerHTML =
"<h4 class='gr'>Connection Success!</h4>";
			stopWifiConnectStatusInterval();
			getConnectInfo();
		}
	}
}
/**
 * Starts the interval for checking the connection status.
 */
function startWifiConnectStatusInterval()
{
	wifiConnectInterval = setInterval(getWifiConnectStatus, 2800);
}
/**
 * Connect WiFi function called using the SSID and password entered
into the text fields.
 */
function connectWifi()
{
	// Get the SSID and password
```

```
	selectedSSID = $("#connect_ssid").val();
	pwd = $("#connect_pass").val();
	$.ajax({
		url: '/wifiConnect.json',
		dataType: 'json',
		method: 'POST',
		cache: false,
		headers: {'my-connect-ssid': selectedSSID, 'my-connect-
pwd': pwd},
		data: {'timestamp': Date.now()}
	});
	startWifiConnectStatusInterval();
}
/**
 * Checks credentials on connect_wifi button click.
 */
function checkCredentials()
{
	errorList = "";
	credsOk = true;
	selectedSSID = $("#connect_ssid").val();
	pwd = $("#connect_pass").val();
	if (selectedSSID == "")
	{
		errorList += "<h4 class='rd'>SSID cannot be empty!</h4>";
		credsOk = false;
	}
	if (pwd == "")
	{
		errorList += "<h4 class='rd'>Password cannot be
empty!</h4>";
		credsOk = false;
	}
	if (credsOk == false)
	{
```

```
		$("#wifi_connect_credentials_errors").html(errorList);
	}
	else
	{
		$("#wifi_connect_credentials_errors").html("");
		connectWifi();
	}
}
/**
 * Shows the WiFi password if the box is checked.
 */
function showPassword()
{
	var x = document.getElementById("connect_pass");
	if (x.type === "password")
	{
		x.type = "text";
	}
	else
	{
		x.type = "password";
	}
}
/**
 * Gets the connection information for displaying on the web page.
 */
function getConnectInfo()
{
	$.getJSON('/wifiConnectInfo.json', function(data)
	{
		$("#connected_ap_label").html("Connected to: ");
		$("#connected_ap").text(data["ap"]);
		$("#ip_address_label").html("IP Address: ");
		$("#wifi_connect_ip").text(data["ip"]);
		$("#netmask_label").html("Netmask: ");
```

```
		$("#wifi_connect_netmask").text(data["netmask"]);
		$("#gateway_label").html("Gateway: ");
		$("#wifi_connect_gw").text(data["gw"]);
		document.getElementById('disconnect_wifi').style.display =
'block';
	});
}
```

33.4 CODE OF INDEX.HTML:

```
<!DOCTYPE html>
<html lang="en">
	<head>
		<meta charset="utf-8"/>
		<meta name="viewport" content="width=device-width,
initial-scale=1.0, user-scalable=no">
		<meta name="apple-mobile-web-app-capable" content="yes"
/>
		<script src='jquery-3.3.1.min.js'></script>
		<link rel="stylesheet" href="app.css">
		<script async src="app.js"></script>
		<title>ESP32 Udemy Course</title>
	</head>
	<body>
	<header>
		<h1>ESP32 Application Development</h1>
	</header>
	<div id="OTA">
	<h2>ESP32 Firmware Update</h2>
		<label id="latest_firmware_label">Latest Firmware: </label>
		<div id="latest_firmware"></div>
		<input type="file" id="selected_file" accept=".bin"
style="display: none;" onchange="getFileInfo()" />
		<div class="buttons">
```

```
                <input type="button" value="Select File"
onclick="document.getElementById('selected_file').click();" />
                <input type="button" value="Update Firmware"
onclick="updateFirmware()" />
            </div>
            <h4 id="file_info"></h4>
            <h4 id="ota_update_status"></h4>
        </div>
        <hr>
        <div id="DHT22Sensor">
            <h2>DHT22 Sensor Readings</h2>
            <label for="temperature_reading">Temperature: </label>
            <div id="temperature_reading"></div>
            <label for="humidity_reading">Humidity: </label>
            <div id="humidity_reading"></div>
        </div>
        <hr>
        <div id="WiFiConnect">
            <h2>ESP32 WiFi Connect</h2>
            <section>
                <input id="connect_ssid" type="text" maxlength="32"
placeholder="SSID" value="">
                <input id="connect_pass" type="password"
maxlength="64" placeholder="Password" value="">
                <input type="checkbox"
onclick="showPassword()">Show Password
            </section>
            <div class="buttons">
                <input id="connect_wifi" type="button"
value="Connect" />
            </div>
            <div id="wifi_connect_credentials_errors"></div>
            <h4 id="wifi_connect_status"></h4>
        </div>
        <div id="ConnectInfo">
```

```
            <section>
                <div id="connected_ap_label"></div> <div
id="connected_ap"></div>
            </section>
            <div id="ip_address_label"></div> <div
id="wifi_connect_ip"></div>
            <div id="netmask_label"></div> <div
id="wifi_connect_netmask"></div>
            <div id="gateway_label"></div> <div
id="wifi_connect_gw"></div>
            <div class="buttons">
                <input id="disconnect_wifi" type="button"
value="Disconnect" />
            </div>
        </div>
        <hr>
        </body>
<html>
```

CHAPTER#34 WIFI DISCONNECT BUTTON

34.1 WIFI DISCONNECT BUTTON:

To implement a Wi-Fi disconnect button on the ESP32 using ESP-IDF, configure a GPIO pin as an input for the button. Monitor the button state in a loop, and call `esp_wifi_disconnect()` when the button is pressed. This setup allows the ESP32 to disconnect from the Wi-Fi network upon detecting a button press, providing a simple and effective method for user-initiated Wi-Fi disconnection.

CHAPTER#35 CODES OF WIFI DISCONNECT BUTTON

35.1 CODE OF HTTP_SERVER.C:

```
/*
 * http_server.c
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFA SHAHBAZ
 */
#include "esp_http_server.h"
#include "esp_log.h"
#include "esp_ota_ops.h"
#include "esp_wifi.h"
#include "sys/param.h"
#include "DHT22.h"
#include "http_server.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG[] = "http_server";
// Wifi connect status
static int g_wifi_connect_status = NONE;
// Firmware update status
static int g_fw_update_status = OTA_UPDATE_PENDING;
// HTTP server task handle
static httpd_handle_t http_server_handle = NULL;
// HTTP server monitor task handle
static TaskHandle_t task_http_server_monitor = NULL;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t http_server_monitor_queue_handle;
/**
 * ESP32 timer configuration passed to esp_timer_create.
```

```
 */
const esp_timer_create_args_t fw_update_reset_args = {
		.callback = &http_server_fw_update_reset_callback,
		.arg = NULL,
		.dispatch_method = ESP_TIMER_TASK,
		.name = "fw_update_reset"
};
esp_timer_handle_t fw_update_reset;
// Embedded files: JQuery, index.html, app.css, app.js and favicon.ico
files
extern const uint8_t jquery_3_3_1_min_js_start[]
	asm("_binary_jquery_3_3_1_min_js_start");
extern const uint8_t jquery_3_3_1_min_js_end[]
	asm("_binary_jquery_3_3_1_min_js_end");
extern const uint8_t index_html_start[]
	asm("_binary_index_html_start");
extern const uint8_t index_html_end[]
	asm("_binary_index_html_end");
extern const uint8_t app_css_start[]
	asm("_binary_app_css_start");
extern const uint8_t app_css_end[]
	asm("_binary_app_css_end");
extern const uint8_t app_js_start[]
	asm("_binary_app_js_start");
extern const uint8_t app_js_end[]
	asm("_binary_app_js_end");
extern const uint8_t favicon_ico_start[]
	asm("_binary_favicon_ico_start");
extern const uint8_t favicon_ico_end[]
	asm("_binary_favicon_ico_end");
/**
 * Checks the g_fw_update_status and creates the fw_update_reset timer
if g_fw_update_status is true.
 */
static void http_server_fw_update_reset_timer(void)
```

```
{
	if (g_fw_update_status == OTA_UPDATE_SUCCESSFUL)
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
updated successful starting FW update reset timer");
		// Give the web page a chance to receive an acknowledge
back and initialize the timer

	ESP_ERROR_CHECK(esp_timer_create(&fw_update_reset_args,
&fw_update_reset));

	ESP_ERROR_CHECK(esp_timer_start_once(fw_update_reset,
8000000));
	}
	else
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
update unsuccessful");
	}
}
/**
 * HTTP server monitor task used to track events of the HTTP server
 * @param pvParameters parameter which can be passed to the task.
 */
static void http_server_monitor(void *parameter)
{
	http_server_queue_message_t msg;
	for (;;)
	{
		if (xQueueReceive(http_server_monitor_queue_handle,
&msg, portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case HTTP_MSG_WIFI_CONNECT_INIT:
```

```
                                ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_INIT");
                                g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECTING;
                                break;
                        case HTTP_MSG_WIFI_CONNECT_SUCCESS:
                                ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_SUCCESS");
                                g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_SUCCESS;
                                break;
                        case HTTP_MSG_WIFI_CONNECT_FAIL:
                                ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_FAIL");
                                g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_FAILED;
                                break;
                        case
HTTP_MSG_OTA_UPDATE_SUCCESSFUL:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_SUCCESSFUL");
                                g_fw_update_status =
OTA_UPDATE_SUCCESSFUL;
                                http_server_fw_update_reset_timer();
                                break;
                        case HTTP_MSG_OTA_UPDATE_FAILED:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_FAILED");
                                g_fw_update_status =
OTA_UPDATE_FAILED;
                                break;
                        default:
                                break;
                }
        }
```

```
        }
}
/**
 * Jquery get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_jquery_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "Jquery requested");
        httpd_resp_set_type(req, "application/javascript");
        httpd_resp_send(req, (const char *)jquery_3_3_1_min_js_start,
jquery_3_3_1_min_js_end - jquery_3_3_1_min_js_start);
        return ESP_OK;
}
/**
 * Sends the index.html page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_index_html_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "index.html requested");
        httpd_resp_set_type(req, "text/html");
        httpd_resp_send(req, (const char *)index_html_start,
index_html_end - index_html_start);
        return ESP_OK;
}
/**
 * app.css get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_css_handler(httpd_req_t *req)
{
```

```
	ESP_LOGI(TAG, "app.css requested");
	httpd_resp_set_type(req, "text/css");
	httpd_resp_send(req, (const char *)app_css_start, app_css_end -
app_css_start);
	return ESP_OK;
}
/**
 * app.js get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_js_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.js requested");
	httpd_resp_set_type(req, "application/javascript");
	httpd_resp_send(req, (const char *)app_js_start, app_js_end -
app_js_start);
	return ESP_OK;
}
/**
 * Sends the .ico (icon) file when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_favicon_ico_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "favicon.ico requested");
	httpd_resp_set_type(req, "image/x-icon");
	httpd_resp_send(req, (const char *)favicon_ico_start,
favicon_ico_end - favicon_ico_start);
	return ESP_OK;
}
/**
 * Receives the .bin file fia the web page and handles the firmware
update
```

```
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK, otherwise ESP_FAIL if timeout occurs and the
update cannot be started.
 */
esp_err_t http_server_OTA_update_handler(httpd_req_t *req)
{
	esp_ota_handle_t ota_handle;
	char ota_buff[1024];
	int content_length = req->content_len;
	int content_received = 0;
	int recv_len;
	bool is_req_body_started = false;
	bool flash_successful = false;
	const esp_partition_t *update_partition =
esp_ota_get_next_update_partition(NULL);
	do
	{
		// Read the data for the request
		if ((recv_len = httpd_req_recv(req, ota_buff,
MIN(content_length, sizeof(ota_buff)))) < 0)
		{
			// Check if timeout occurred
			if (recv_len == HTTPD_SOCK_ERR_TIMEOUT)
			{
				ESP_LOGI(TAG,
"http_server_OTA_update_handler: Socket Timeout");
				continue; ///> Retry receiving if timeout occurred
			}
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
OTA other Error %d", recv_len);
			return ESP_FAIL;
		}
		printf("http_server_OTA_update_handler: OTA RX: %d of
%d\r", content_received, content_length);
		// Is this the first data we are receiving
```

```
		// If so, it will have the information in the header that we
need.
		if (!is_req_body_started)
		{
			is_req_body_started = true;
			// Get the location of the .bin file content (remove the
web form data)
			char *body_start_p = strstr(ota_buff, "\r\n\r\n") + 4;
			int body_part_len = recv_len - (body_start_p -
ota_buff);
			printf("http_server_OTA_update_handler: OTA file
size: %d\r\n", content_length);
			esp_err_t err = esp_ota_begin(update_partition,
OTA_SIZE_UNKNOWN, &ota_handle);
			if (err != ESP_OK)
			{
				printf("http_server_OTA_update_handler: Error
with OTA begin, cancelling OTA\r\n");
				return ESP_FAIL;
			}
			else
			{
				printf("http_server_OTA_update_handler:
Writing to partition subtype %d at offset 0x%x\r\n", update_partition-
>subtype, update_partition->address);
			}
			// Write this first part of the data
			esp_ota_write(ota_handle, body_start_p,
body_part_len);
			content_received += body_part_len;
		}
		else
		{
			// Write OTA data
			esp_ota_write(ota_handle, ota_buff, recv_len);
```

```
                content_received += recv_len;
            }
        } while (recv_len > 0 && content_received < content_length);
        if (esp_ota_end(ota_handle) == ESP_OK)
        {
            // Lets update the partition
            if (esp_ota_set_boot_partition(update_partition) ==
ESP_OK)
            {
                const esp_partition_t *boot_partition =
esp_ota_get_boot_partition();
                ESP_LOGI(TAG, "http_server_OTA_update_handler:
Next boot partition subtype %d at offset 0x%x", boot_partition-
>subtype, boot_partition->address);
                flash_successful = true;
            }
            else
            {
                ESP_LOGI(TAG, "http_server_OTA_update_handler:
FLASHED ERROR!!!");
            }
        }
        else
        {
            ESP_LOGI(TAG, "http_server_OTA_update_handler:
esp_ota_end ERROR!!!");
        }
        // We won't update the global variables throughout the file, so send
the message about the status
        if (flash_successful) {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_SU
CCESSFUL); } else {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_FAI
LED); }
        return ESP_OK;
```

```
}
/**
 * OTA status handler responds with the firmware update status after the
OTA update is started
 * and responds with the compile time/date when the page is first
requested
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
esp_err_t http_server_OTA_status_handler(httpd_req_t *req)
{
	char otaJSON[100];
	ESP_LOGI(TAG, "OTAstatus requested");
	sprintf(otaJSON,
"{\"ota_update_status\":%d,\"compile_time\":\"%s\",\"compile_date\":\"
%s\"}", g_fw_update_status, __TIME__, __DATE__);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, otaJSON, strlen(otaJSON));
	return ESP_OK;
}
/**
 * DHT sensor readings JSON handler responds with DHT22 sensor data
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
static esp_err_t
http_server_get_dht_sensor_readings_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/dhtSensor.json requested");
	char dhtSensorJSON[100];
	sprintf(dhtSensorJSON,
"{\"temp\":\"%.1f\",\"humidity\":\"%.1f\"}", getTemperature(),
getHumidity());
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, dhtSensorJSON, strlen(dhtSensorJSON));
```

```
	return ESP_OK;
}
/**
 * wifiConnect.json handler is invoked after the connect button is
pressed
 * and handles receiving the SSID and password entered by the user
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_wifi_connect_json_handler(httpd_req_t
*req)
{
	ESP_LOGI(TAG, "/wifiConnect.json requested");
	size_t len_ssid = 0, len_pass = 0;
	char *ssid_str = NULL, *pass_str = NULL;
	// Get SSID header
	len_ssid = httpd_req_get_hdr_value_len(req, "my-connect-ssid") +
1;
	if (len_ssid > 1)
	{
		ssid_str = malloc(len_ssid);
		if (httpd_req_get_hdr_value_str(req, "my-connect-ssid",
ssid_str, len_ssid) == ESP_OK)
		{
			ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
ssid: %s", ssid_str);
		}
	}
	// Get Password header
	len_pass = httpd_req_get_hdr_value_len(req, "my-connect-pwd")
+ 1;
	if (len_pass > 1)
	{
		pass_str = malloc(len_pass);
```

```
		if (httpd_req_get_hdr_value_str(req, "my-connect-pwd",
pass_str, len_pass) == ESP_OK)
		{
			ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
pwd: %s", pass_str);
		}
	}
	// Update the Wifi networks configuration and let the wifi
application know
	wifi_config_t* wifi_config = wifi_app_get_wifi_config();
	memset(wifi_config, 0x00, sizeof(wifi_config_t));
	memcpy(wifi_config->sta.ssid, ssid_str, len_ssid);
	memcpy(wifi_config->sta.password, pass_str, len_pass);
	wifi_app_send_message(WIFI_APP_MSG_CONNECTING_FRO
M_HTTP_SERVER);
	free(ssid_str);
	free(pass_str);
	return ESP_OK;
}
/**
 * wifiConnectStatus handler updates the connection status for the web
page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t
http_server_wifi_connect_status_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/wifiConnectStatus requested");
	char statusJSON[100];
	sprintf(statusJSON, "{\"wifi_connect_status\":%d}",
g_wifi_connect_status);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, statusJSON, strlen(statusJSON));
```

```
	return ESP_OK;
}
/**
 * wifiConnectInfo.json handler updates the web page with connection
information.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t
http_server_get_wifi_connect_info_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/wifiConnectInfo.json requested");
	char ipInfoJSON[200];
	memset(ipInfoJSON, 0, sizeof(ipInfoJSON));
	char ip[IP4ADDR_STRLEN_MAX];
	char netmask[IP4ADDR_STRLEN_MAX];
	char gw[IP4ADDR_STRLEN_MAX];
	if (g_wifi_connect_status ==
HTTP_WIFI_STATUS_CONNECT_SUCCESS)
	{
		wifi_ap_record_t wifi_data;

	ESP_ERROR_CHECK(esp_wifi_sta_get_ap_info(&wifi_data));
		char *ssid = (char*)wifi_data.ssid;
		esp_netif_ip_info_t ip_info;
		ESP_ERROR_CHECK(esp_netif_get_ip_info(esp_netif_sta,
&ip_info));
		esp_ip4addr_ntoa(&ip_info.ip, ip,
IP4ADDR_STRLEN_MAX);
		esp_ip4addr_ntoa(&ip_info.netmask, netmask,
IP4ADDR_STRLEN_MAX);
		esp_ip4addr_ntoa(&ip_info.gw, gw,
IP4ADDR_STRLEN_MAX);
```

```
                sprintf(ipInfoJSON,
"{\"ip\":\"%s\",\"netmask\":\"%s\",\"gw\":\"%s\",\"ap\":\"%s\"}", ip,
netmask, gw, ssid);
        }
        httpd_resp_set_type(req, "application/json");
        httpd_resp_send(req, ipInfoJSON, strlen(ipInfoJSON));
        return ESP_OK;
}
/**
 * wifiDisconnect.json handler responds by sending a message to the
Wifi application to disconnect.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_wifi_disconnect_json_handler(httpd_req_t
*req)
{
        ESP_LOGI(TAG, "wifiDisconect.json requested");
        wifi_app_send_message(WIFI_APP_MSG_USER_REQUESTED
_STA_DISCONNECT);
        return ESP_OK;
}
/**
 * Sets up the default httpd server configuration.
 * @return http server instance handle if successful, NULL otherwise.
 */
static httpd_handle_t http_server_configure(void)
{
        // Generate the default configuration
        httpd_config_t config = HTTPD_DEFAULT_CONFIG();
        // Create HTTP server monitor task
        xTaskCreatePinnedToCore(&http_server_monitor,
"http_server_monitor", HTTP_SERVER_MONITOR_STACK_SIZE,
NULL, HTTP_SERVER_MONITOR_PRIORITY,
&task_http_server_monitor, HTTP_SERVER_MONITOR_CORE_ID);
```

```
	// Create the message queue
	http_server_monitor_queue_handle = xQueueCreate(3,
sizeof(http_server_queue_message_t));
	// The core that the HTTP server will run on
	config.core_id = HTTP_SERVER_TASK_CORE_ID;
	// Adjust the default priority to 1 less than the wifi application task
	config.task_priority = HTTP_SERVER_TASK_PRIORITY;
	// Bump up the stack size (default is 4096)
	config.stack_size = HTTP_SERVER_TASK_STACK_SIZE;
	// Increase uri handlers
	config.max_uri_handlers = 20;
	// Increase the timeout limits
	config.recv_wait_timeout = 10;
	config.send_wait_timeout = 10;
	ESP_LOGI(TAG,
			"http_server_configure: Starting server on port: '%d'
with task priority: '%d'",
			config.server_port,
			config.task_priority);
	// Start the httpd server
	if (httpd_start(&http_server_handle, &config) == ESP_OK)
	{
		ESP_LOGI(TAG, "http_server_configure: Registering URI
handlers");
		// register query handler
		httpd_uri_t jquery_js = {
				.uri = "/jquery-3.3.1.min.js",
				.method = HTTP_GET,
				.handler = http_server_jquery_handler,
				.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &jquery_js);
		// register index.html handler
		httpd_uri_t index_html = {
				.uri = "/",
```

```
                .method = HTTP_GET,
                .handler = http_server_index_html_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&index_html);
        // register app.css handler
        httpd_uri_t app_css = {
                .uri = "/app.css",
                .method = HTTP_GET,
                .handler = http_server_app_css_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_css);
        // register app.js handler
        httpd_uri_t app_js = {
                .uri = "/app.js",
                .method = HTTP_GET,
                .handler = http_server_app_js_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_js);
        // register favicon.ico handler
        httpd_uri_t favicon_ico = {
                .uri = "/favicon.ico",
                .method = HTTP_GET,
                .handler = http_server_favicon_ico_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&favicon_ico);
        // register OTAupdate handler
        httpd_uri_t OTA_update = {
                .uri = "/OTAupdate",
                .method = HTTP_POST,
```

```
                .handler = http_server_OTA_update_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_update);
        // register OTAstatus handler
        httpd_uri_t OTA_status = {
                .uri = "/OTAstatus",
                .method = HTTP_POST,
                .handler = http_server_OTA_status_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_status);
        // register dhtSensor.json handler
        httpd_uri_t dht_sensor_json = {
                .uri = "/dhtSensor.json",
                .method = HTTP_GET,
                .handler =
http_server_get_dht_sensor_readings_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&dht_sensor_json);
        // register wifiConnect.json handler
        httpd_uri_t wifi_connect_json = {
                .uri = "/wifiConnect.json",
                .method = HTTP_POST,
                .handler =
http_server_wifi_connect_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_connect_json);
        // register wifiConnectStatus.json handler
```

```
		httpd_uri_t wifi_connect_status_json = {
			.uri = "/wifiConnectStatus",
			.method = HTTP_POST,
			.handler =
http_server_wifi_connect_status_json_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&wifi_connect_status_json);
		// register wifiConnectInfo.json handler
		httpd_uri_t wifi_connect_info_json = {
			.uri = "/wifiConnectInfo.json",
			.method = HTTP_GET,
			.handler =
http_server_get_wifi_connect_info_json_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&wifi_connect_info_json);
		// register wifiDisconnect.json handler
		httpd_uri_t wifi_disconnect_json = {
			.uri = "/wifiDisconnect.json",
			.method = HTTP_DELETE,
			.handler =
http_server_wifi_disconnect_json_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&wifi_disconnect_json);
		return http_server_handle;
	}
	return NULL;
}
void http_server_start(void)
{
```

```
	if (http_server_handle == NULL)
	{
		http_server_handle = http_server_configure();
	}
}
void http_server_stop(void)
{
	if (http_server_handle)
	{
		httpd_stop(http_server_handle);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP
server");
		http_server_handle = NULL;
	}
	if (task_http_server_monitor)
	{
		vTaskDelete(task_http_server_monitor);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP server
monitor");
		task_http_server_monitor = NULL;
	}
}
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID)
{
	http_server_queue_message_t msg;
	msg.msgID = msgID;
	return xQueueSend(http_server_monitor_queue_handle, &msg,
portMAX_DELAY);
}
void http_server_fw_update_reset_callback(void *arg)
{
	ESP_LOGI(TAG, "http_server_fw_update_reset_callback: Timer
timed-out, restarting the device");
	esp_restart();
```

```
}
```

35.2 CODE OF WIFI_APP.C:

```
/*
 * wifi_app.c
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
 #include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "freertos/task.h"
#include "esp_err.h"
#include "esp_log.h"
#include "esp_wifi.h"
#include "lwip/netdb.h"
#include "http_server.h"
#include "rgb_led.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG [] = "wifi_app";
// Used for returning the WiFi configuration
wifi_config_t *wifi_config = NULL;
// Used to track the number for retries when a connection attempt fails
static int g_retry_number;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t wifi_app_queue_handle;
// netif objects for the station and access point
esp_netif_t* esp_netif_sta = NULL;
esp_netif_t* esp_netif_ap  = NULL;
/**
 * WiFi application event handler
```

```
 * @param arg data, aside from event data, that is passed to the handler
when it is called
 * @param event_base the base id of the event to register the handler for
 * @param event_id the id fo the event to register the handler for
 * @param event_data event data
 */
static void wifi_app_event_handler(void *arg, esp_event_base_t
event_base, int32_t event_id, void *event_data)
{
	if (event_base == WIFI_EVENT)
	{
		switch (event_id)
		{
			case WIFI_EVENT_AP_START:
				ESP_LOGI(TAG,
"WIFI_EVENT_AP_START");
				break;
			case WIFI_EVENT_AP_STOP:
				ESP_LOGI(TAG, "WIFI_EVENT_AP_STOP");
				break;
			case WIFI_EVENT_AP_STACONNECTED:
				ESP_LOGI(TAG,
"WIFI_EVENT_AP_STACONNECTED");
				break;
			case WIFI_EVENT_AP_STADISCONNECTED:
				ESP_LOGI(TAG,
"WIFI_EVENT_AP_STADISCONNECTED");
				break;
			case WIFI_EVENT_STA_START:
				ESP_LOGI(TAG,
"WIFI_EVENT_STA_START");
				break;
			case WIFI_EVENT_STA_CONNECTED:
				ESP_LOGI(TAG,
"WIFI_EVENT_STA_CONNECTED");
```

```
                        break;
                case WIFI_EVENT_STA_DISCONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_DISCONNECTED");
                        wifi_event_sta_disconnected_t
*wifi_event_sta_disconnected =
(wifi_event_sta_disconnected_t*)malloc(sizeof(wifi_event_sta_disconn
ected_t));
                        *wifi_event_sta_disconnected =
*((wifi_event_sta_disconnected_t*)event_data);
                        printf("WIFI_EVENT_STA_DISCONNECTED,
reason code %d\n", wifi_event_sta_disconnected->reason);
                        if (g_retry_number <
MAX_CONNECTION_RETRIES)
                        {
                                esp_wifi_connect();
                                g_retry_number ++;
                        }
                        else
                        {

        wifi_app_send_message(WIFI_APP_MSG_STA_DISCONNECT
ED);
                        }
                        break;
                }
        }
        else if (event_base == IP_EVENT)
        {
                switch (event_id)
                {
                        case IP_EVENT_STA_GOT_IP:
                                ESP_LOGI(TAG, "IP_EVENT_STA_GOT_IP");
```

```
        wifi_app_send_message(WIFI_APP_MSG_STA_CONNECTED_
GOT_IP);
                        break;
            }
        }
}
/**
 * Initializes the WiFi application event handler for WiFi and IP events.
 */
static void wifi_app_event_handler_init(void)
{
        // Event loop for the WiFi driver
        ESP_ERROR_CHECK(esp_event_loop_create_default());
        // Event handler for the connection
        esp_event_handler_instance_t instance_wifi_event;
        esp_event_handler_instance_t instance_ip_event;
        ESP_ERROR_CHECK(esp_event_handler_instance_register(WIF
I_EVENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_wifi_event));
        ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_E
VENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_ip_event));
}
/**
 * Initializes the TCP stack and default WiFi configuration.
 */
static void wifi_app_default_wifi_init(void)
{
        // Initialize the TCP stack
        ESP_ERROR_CHECK(esp_netif_init());
        // Default WiFi config - operations must be in this order!
        wifi_init_config_t wifi_init_config =
WIFI_INIT_CONFIG_DEFAULT();
        ESP_ERROR_CHECK(esp_wifi_init(&wifi_init_config));
```

```
	ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_R
AM));
	esp_netif_sta = esp_netif_create_default_wifi_sta();
	esp_netif_ap = esp_netif_create_default_wifi_ap();
}
/**
 * Configures the WiFi access point settings and assigns the static IP to
the SoftAP.
 */
static void wifi_app_soft_ap_config(void)
{
	// SoftAP - WiFi access point configuration
	wifi_config_t ap_config =
	{
		.ap = {
				.ssid = WIFI_AP_SSID,
				.ssid_len = strlen(WIFI_AP_SSID),
				.password = WIFI_AP_PASSWORD,
				.channel = WIFI_AP_CHANNEL,
				.ssid_hidden = WIFI_AP_SSID_HIDDEN,
				.authmode = WIFI_AUTH_WPA2_PSK,
				.max_connection =
WIFI_AP_MAX_CONNECTIONS,
				.beacon_interval =
WIFI_AP_BEACON_INTERVAL,
			},
	};
	// Configure DHCP for the AP
	esp_netif_ip_info_t ap_ip_info;
	memset(&ap_ip_info, 0x00, sizeof(ap_ip_info));
	esp_netif_dhcps_stop(esp_netif_ap);				///>
must call this first
	inet_pton(AF_INET, WIFI_AP_IP, &ap_ip_info.ip);		///>
Assign access point's static IP, GW, and netmask
	inet_pton(AF_INET, WIFI_AP_GATEWAY, &ap_ip_info.gw);
```

```
	inet_pton(AF_INET, WIFI_AP_NETMASK,
&ap_ip_info.netmask);
	ESP_ERROR_CHECK(esp_netif_set_ip_info(esp_netif_ap,
&ap_ip_info));		///> Statically configure the network
interface
	ESP_ERROR_CHECK(esp_netif_dhcps_start(esp_netif_ap));
				///> Start the AP DHCP server (for
connecting stations e.g. your mobile device)
	ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_APST
A));				///> Setting the mode as Access Point /
Station Mode
	ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP,
&ap_config));		///> Set our configuration
	ESP_ERROR_CHECK(esp_wifi_set_bandwidth(WIFI_IF_AP,
WIFI_AP_BANDWIDTH));		///> Our default bandwidth 20
MHz
	ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_STA_POWER_SA
VE));				///> Power save set to "NONE"
}
/**
 * Connects the ESP32 to an external AP using the updated station
configuration
 */
static void wifi_app_connect_sta(void)
{
	ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA,
wifi_app_get_wifi_config()));
	ESP_ERROR_CHECK(esp_wifi_connect());
}
/**
 * Main task for the WiFi application
 * @param pvParameters parameter which can be passed to the task
 */
static void wifi_app_task(void *pvParameters)
{
```

```
	wifi_app_queue_message_t msg;
	// Initialize the event handler
	wifi_app_event_handler_init();
	// Initialize the TCP/IP stack and WiFi config
	wifi_app_default_wifi_init();
	// SoftAP config
	wifi_app_soft_ap_config();
	// Start WiFi
	ESP_ERROR_CHECK(esp_wifi_start());
	// Send first event message
	wifi_app_send_message(WIFI_APP_MSG_START_HTTP_SERV
ER);
	for (;;)
	{
		if (xQueueReceive(wifi_app_queue_handle, &msg,
portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case
WIFI_APP_MSG_START_HTTP_SERVER:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_START_HTTP_SERVER");
					http_server_start();
					rgb_led_http_server_started();
					break;
				case
WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER");
					// Attempt a connection
					wifi_app_connect_sta();
					// Set current number of retries to zero
					g_retry_number = 0;
```

```
                                // Let the HTTP server know about the
connection attempt

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_INIT);
                                break;
                        case
WIFI_APP_MSG_STA_CONNECTED_GOT_IP:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_CONNECTED_GOT_IP");
                                rgb_led_wifi_connected();

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_SUCCESS);
                                break;
                        case
WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT");
                                g_retry_number =
MAX_CONNECTION_RETRIES;

        ESP_ERROR_CHECK(esp_wifi_disconnect());
                                rgb_led_http_server_started(); ///> todo:
rename this status LED to a name more meaningful (to your liking)...
                                break;
                        case WIFI_APP_MSG_STA_DISCONNECTED:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED");

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_FAIL);
                                break;
                        default:
                                break;
```

```
                }
            }
        }
}
BaseType_t wifi_app_send_message(wifi_app_message_e msgID)
{
        wifi_app_queue_message_t msg;
        msg.msgID = msgID;
        return xQueueSend(wifi_app_queue_handle, &msg,
portMAX_DELAY);
}
wifi_config_t* wifi_app_get_wifi_config(void)
{
        return wifi_config;
}
void wifi_app_start(void)
{
        ESP_LOGI(TAG, "STARTING WIFI APPLICATION");
        // Start WiFi started LED
        rgb_led_wifi_app_started();
        // Disable default WiFi logging messages
        esp_log_level_set("wifi", ESP_LOG_NONE);
        // Allocate memory for the wifi configuration
        wifi_config = (wifi_config_t*)malloc(sizeof(wifi_config_t));
        memset(wifi_config, 0x00, sizeof(wifi_config_t));
        // Create message queue
        wifi_app_queue_handle = xQueueCreate(3,
sizeof(wifi_app_queue_message_t));
        // Start the WiFi application task
        xTaskCreatePinnedToCore(&wifi_app_task, "wifi_app_task",
WIFI_APP_TASK_STACK_SIZE, NULL,
WIFI_APP_TASK_PRIORITY, NULL,
WIFI_APP_TASK_CORE_ID);
}
```

35.3 CODE OF WIFI_APP.H:

```
/*
 * wifi_app.h
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_WIFI_APP_H_
#define MAIN_WIFI_APP_H_
#include "esp_netif.h"
// WiFi application settings
#define WIFI_AP_SSID                    "ESP32_AP"
        // AP name
#define WIFI_AP_PASSWORD                "password"
        // AP password
#define WIFI_AP_CHANNEL                 1
        // AP channel
#define WIFI_AP_SSID_HIDDEN             0
        // AP visibility
#define WIFI_AP_MAX_CONNECTIONS         5
        // AP max clients
#define WIFI_AP_BEACON_INTERVAL         100
        // AP beacon: 100 milliseconds recommended
#define WIFI_AP_IP                      "192.168.0.1"           // AP
default IP
#define WIFI_AP_GATEWAY                 "192.168.0.1"
        // AP default Gateway (should be the same as the IP)
#define WIFI_AP_NETMASK                 "255.255.255.0"
        // AP netmask
#define WIFI_AP_BANDWIDTH               WIFI_BW_HT20
        // AP bandwidth 20 MHz (40 MHz is the other option)
#define WIFI_STA_POWER_SAVE             WIFI_PS_NONE
        // Power save not used
```

```
#define MAX_SSID_LENGTH                    32
        // IEEE standard maximum
#define MAX_PASSWORD_LENGTH                64
                // IEEE standard maximum
#define MAX_CONNECTION_RETRIES             5
        // Retry number on disconnect
// netif object for the Station and Access Point
extern esp_netif_t* esp_netif_sta;
extern esp_netif_t* esp_netif_ap;
/**
 * Message IDs for the WiFi application task
 * @note Expand this based on your application requirements.
 */
typedef enum wifi_app_message
{
        WIFI_APP_MSG_START_HTTP_SERVER = 0,
        WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER,
        WIFI_APP_MSG_STA_CONNECTED_GOT_IP,
        WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT,
        WIFI_APP_MSG_STA_DISCONNECTED,
} wifi_app_message_e;
/**
 * Structure for the message queue
 * @note Expand this based on application requirements e.g. add another
type and parameter as required
 */
typedef struct wifi_app_queue_message
{
        wifi_app_message_e msgID;
} wifi_app_queue_message_t;
/**
 * Sends a message to the queue
 * @param msgID message ID from the wifi_app_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue,
otherwise pdFALSE.
```

```
 * @note Expand the parameter list based on your requirements e.g. how
you've expanded the wifi_app_queue_message_t.
 */
BaseType_t wifi_app_send_message(wifi_app_message_e msgID);
/**
 * Starts the WiFi RTOS task
 */
void wifi_app_start(void);
/**
 * Gets the wifi configuration
 */
wifi_config_t* wifi_app_get_wifi_config(void);
#endif /* MAIN_WIFI_APP_H_ */
```

35.4 CODE OF APP.JS:

```
/**
 * Add gobals here
 */
var seconds        = null;
var otaTimerVar =  null;
var wifiConnectInterval = null;
/**
 * Initialize functions here.
 */
$(document).ready(function(){
        getUpdateStatus();
        startDHTSensorInterval();
        getConnectInfo();
        $("#connect_wifi").on("click", function(){
                checkCredentials();
        });
        $("#disconnect_wifi").on("click", function(){
                disconnectWifi();
        });
```

```
});
/**
 * Gets file name and size for display on the web page.
 */
function getFileInfo()
{
   var x = document.getElementById("selected_file");
   var file = x.files[0];
   document.getElementById("file_info").innerHTML = "<h4>File: " +
file.name + "<br>" + "Size: " + file.size + " bytes</h4>";
}
/**
 * Handles the firmware update.
 */
function updateFirmware()
{
   // Form Data
   var formData = new FormData();
   var fileSelect = document.getElementById("selected_file");
   if (fileSelect.files && fileSelect.files.length == 1)
        {
       var file = fileSelect.files[0];
       formData.set("file", file, file.name);
       document.getElementById("ota_update_status").innerHTML =
"Uploading " + file.name + ", Firmware Update in Progress...";
       // Http Request
       var request = new XMLHttpRequest();
       request.upload.addEventListener("progress", updateProgress);
       request.open('POST', "/OTAupdate");
       request.responseType = "blob";
       request.send(formData);
   }
        else
        {
       window.alert('Select A File First')
```

```
    }
}
/**
 * Progress on transfers from the server to the client (downloads).
 */
function updateProgress(oEvent)
{
    if (oEvent.lengthComputable)
        {
        getUpdateStatus();
    }
        else
        {
        window.alert('total size is unknown')
    }
}
/**
 * Posts the firmware udpate status.
 */
function getUpdateStatus()
{
    var xhr = new XMLHttpRequest();
    var requestURL = "/OTAstatus";
    xhr.open('POST', requestURL, false);
    xhr.send('ota_update_status');
    if (xhr.readyState == 4 && xhr.status == 200)
        {
        var response = JSON.parse(xhr.responseText);
                document.getElementById("latest_firmware").innerHTML =
response.compile_date + " - " + response.compile_time
                // If flashing was complete it will return a 1, else -1
                // A return of 0 is just for information on the Latest Firmware
request
        if (response.ota_update_status == 1)
                {
```

```
                // Set the countdown timer time
            seconds = 10;
            // Start the countdown timer
            otaRebootTimer();
        }
        else if (response.ota_update_status == -1)
                {
            document.getElementById("ota_update_status").innerHTML =
"!!! Upload Error !!!";
        }
    }
}
/**
 * Displays the reboot countdown.
 */
function otaRebootTimer()
{
    document.getElementById("ota_update_status").innerHTML = "OTA
Firmware Update Complete. This page will close shortly, Rebooting in:
" + seconds;
    if (--seconds == 0)
         {
        clearTimeout(otaTimerVar);
        window.location.reload();
    }
         else
         {
        otaTimerVar = setTimeout(otaRebootTimer, 1000);
    }
}
/**
 * Gets DHT22 sensor temperature and humidity values for display on
the web page.
 */
function getDHTSensorValues()
```

```
{
	$.getJSON('/dhtSensor.json', function(data) {
		$("#temperature_reading").text(data["temp"]);
		$("#humidity_reading").text(data["humidity"]);
	});
}
/**
 * Sets the interval for getting the updated DHT22 sensor values.
 */
function startDHTSensorInterval()
{
	setInterval(getDHTSensorValues, 5000);
}
/**
 * Clears the connection status interval.
 */
function stopWifiConnectStatusInterval()
{
	if (wifiConnectInterval != null)
	{
		clearInterval(wifiConnectInterval);
		wifiConnectInterval = null;
	}
}
/**
 * Gets the WiFi connection status.
 */
function getWifiConnectStatus()
{
	var xhr = new XMLHttpRequest();
	var requestURL = "/wifiConnectStatus";
	xhr.open('POST', requestURL, false);
	xhr.send('wifi_connect_status');
	if (xhr.readyState == 4 && xhr.status == 200)
	{
```

```
		var response = JSON.parse(xhr.responseText);

	document.getElementById("wifi_connect_status").innerHTML =
"Connecting...";
		if (response.wifi_connect_status == 2)
		{

	document.getElementById("wifi_connect_status").innerHTML =
"<h4 class='rd'>Failed to Connect. Please check your AP credentials and
compatibility</h4>";
			stopWifiConnectStatusInterval();
		}
		else if (response.wifi_connect_status == 3)
		{

	document.getElementById("wifi_connect_status").innerHTML =
"<h4 class='gr'>Connection Success!</h4>";
			stopWifiConnectStatusInterval();
			getConnectInfo();
		}
	}
}
/**
 * Starts the interval for checking the connection status.
 */
function startWifiConnectStatusInterval()
{
	wifiConnectInterval = setInterval(getWifiConnectStatus, 2800);
}
/**
 * Connect WiFi function called using the SSID and password entered
into the text fields.
 */
function connectWifi()
{
```

```
	// Get the SSID and password
	selectedSSID = $("#connect_ssid").val();
	pwd = $("#connect_pass").val();
	$.ajax({
		url: '/wifiConnect.json',
		dataType: 'json',
		method: 'POST',
		cache: false,
		headers: {'my-connect-ssid': selectedSSID, 'my-connect-
pwd': pwd},
		data: {'timestamp': Date.now()}
	});
	startWifiConnectStatusInterval();
}
/**
 * Checks credentials on connect_wifi button click.
 */
function checkCredentials()
{
	errorList = "";
	credsOk = true;
	selectedSSID = $("#connect_ssid").val();
	pwd = $("#connect_pass").val();
	if (selectedSSID == "")
	{
		errorList += "<h4 class='rd'>SSID cannot be empty!</h4>";
		credsOk = false;
	}
	if (pwd == "")
	{
		errorList += "<h4 class='rd'>Password cannot be
empty!</h4>";
		credsOk = false;
	}
	if (credsOk == false)
```

```
        {
                $("#wifi_connect_credentials_errors").html(errorList);
        }
        else
        {
                $("#wifi_connect_credentials_errors").html("");
                connectWifi();
        }
}
/**
 * Shows the WiFi password if the box is checked.
 */
function showPassword()
{
        var x = document.getElementById("connect_pass");
        if (x.type === "password")
        {
                x.type = "text";
        }
        else
        {
                x.type = "password";
        }
}
/**
 * Gets the connection information for displaying on the web page.
 */
function getConnectInfo()
{
        $.getJSON('/wifiConnectInfo.json', function(data)
        {
                $("#connected_ap_label").html("Connected to: ");
                $("#connected_ap").text(data["ap"]);
                $("#ip_address_label").html("IP Address: ");
                $("#wifi_connect_ip").text(data["ip"]);
```

```
		$("#netmask_label").html("Netmask: ");
		$("#wifi_connect_netmask").text(data["netmask"]);
		$("#gateway_label").html("Gateway: ");
		$("#wifi_connect_gw").text(data["gw"]);
		document.getElementById('disconnect_wifi').style.display =
'block';
	});
}
/**
 * Disconnects Wifi once the disconnect button is pressed and reloads
the web page.
 */
function disconnectWifi()
{
	$.ajax({
		url: '/wifiDisconnect.json',
		dataType: 'json',
		method: 'DELETE',
		cache: false,
		data: { 'timestamp': Date.now() }
	});
	// Update the web page
	setTimeout("location.reload(true);", 2000);
}
```

CHAPTER#36 NON-VOLATILE STORAGE (NVS)

36.1 SAVING, LOADING, CLEARING WIFI CREDENTIALS:

After successfully connecting the ESP32 to an access point via the web page, the SSID and Password used to connect will be saved to flash. Upon startup, the ESP32 will check the flash for any saved credentials, if there are any, they will be used to attempt a connection immediately.

Upon startup, if after the MAX_CONNECTION_RETRIES is reached and a connection cannot be established, we will clear the flash.
If the “Disconnect Button” on the web page is pressed, the credentials will be cleared from the flash.

36.2 NVS FOR WIFI CREDENTIALS USING THE ESP-IDF:

36.2.1 SAVING THE CREDENTIALS:

Open storage area with a given namespace ->Call nvs_openand specify the namespace name, open in Read/Write (in our case) and we need to pass a handle as well which will be used for subsequent calls to nvs_set_*, and nvs_commit functions.
Set variable length binary value (SSID & Password) ->Using nvs_set_blob.
After setting the values, we need to write the changes to NVS ->Using nvs_commit.

36.2.2 GETTING THE CREDENTIALS:

Similarly, making changes, when retrieving information from the flash, we must open using nvs_open, then, in our case ->Call nvs_get_blob.

36.2.3 CLEARING THE CREDENTIALS:

Similarly, we open storage area with a given namespace ->Call nvs_openand specify the namespace name, open in Read/Write and we need to pass the handle.
Now, we erase the key-value pairs in the namespace->Using nvs_erase_all.
After erasing, we need to commit the changes to NVS ->Using nvs_commit.

CHAPTER#37 CODES OF NON-VOLATILE STORAGE

37.1 CODE OF APP_NVS.C:

```
/*
 * app_nvs.c
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include "esp_log.h"
#include "nvs_flash.h"
#include "app_nvs.h"
#include "wifi_app.h"
// Tag for logging to the monitor
static const char TAG[] = "nvs";
// NVS name space used for station mode credentials
const char app_nvs_sta_creds_namespace[] = "stacreds";
esp_err_t app_nvs_save_sta_creds(void)
{
	nvs_handle handle;
	esp_err_t esp_err;
	ESP_LOGI(TAG, "app_nvs_save_sta_creds: Saving station mode
credentials to flash");
	wifi_config_t *wifi_sta_config = wifi_app_get_wifi_config();
	if (wifi_sta_config)
	{
		esp_err = nvs_open(app_nvs_sta_creds_namespace,
NVS_READWRITE, &handle);
		if (esp_err != ESP_OK)
```

```
		{
			printf("app_nvs_save_sta_creds: Error (%s) opening
NVS handle!\n", esp_err_to_name(esp_err));
			return esp_err;
		}
		// Set SSID
		esp_err = nvs_set_blob(handle, "ssid", wifi_sta_config-
>sta.ssid, MAX_SSID_LENGTH);
		if (esp_err != ESP_OK)
		{
			printf("app_nvs_save_sta_creds: Error (%s) setting
SSID to NVS!\n", esp_err_to_name(esp_err));
			return esp_err;
		}
		// Set Password
		esp_err = nvs_set_blob(handle, "password", wifi_sta_config-
>sta.password, MAX_PASSWORD_LENGTH);
		if (esp_err != ESP_OK)
		{
			printf("app_nvs_save_sta_creds: Error (%s) setting
Password to NVS!\n", esp_err_to_name(esp_err));
			return esp_err;
		}
		// Commit credentials to NVS
		esp_err = nvs_commit(handle);
		if (esp_err != ESP_OK)
		{
			printf("app_nvs_save_sta_creds: Error (%s) comitting
credentials to NVS!\n", esp_err_to_name(esp_err));
			return esp_err;
		}
		nvs_close(handle);
		ESP_LOGI(TAG, "app_nvs_save_sta_creds: wrote
wifi_sta_config: Station SSID: %s Password: %s", wifi_sta_config-
>sta.ssid, wifi_sta_config->sta.password);
```

```
	}
	printf("app_nvs_save_sta_creds: returned ESP_OK\n");
	return ESP_OK;
}
bool app_nvs_load_sta_creds(void)
{
	nvs_handle handle;
	esp_err_t esp_err;
	ESP_LOGI(TAG, "app_nvs_load_sta_creds: Loading Wifi
credentials from flash");
	if (nvs_open(app_nvs_sta_creds_namespace, NVS_READONLY,
&handle) == ESP_OK)
	{
		wifi_config_t *wifi_sta_config =
wifi_app_get_wifi_config();
		if (wifi_sta_config == NULL)
		{
			wifi_sta_config =
(wifi_config_t*)malloc(sizeof(wifi_config_t));
		}
		memset(wifi_sta_config, 0x00, sizeof(wifi_config_t));
		// Allocate buffer
		size_t wifi_config_size = sizeof(wifi_config_t);
		uint8_t *wifi_config_buff = (uint8_t*)malloc(sizeof(uint8_t)
* wifi_config_size);
		memset(wifi_config_buff, 0x00, sizeof(wifi_config_size));
		// Load SSID
		wifi_config_size = sizeof(wifi_sta_config->sta.ssid);
		esp_err = nvs_get_blob(handle, "ssid", wifi_config_buff,
&wifi_config_size);
		if (esp_err != ESP_OK)
		{
			free(wifi_config_buff);
			printf("app_nvs_load_sta_creds: (%s) no station SSID
found in NVS\n", esp_err_to_name(esp_err));
```

```
                return false;
            }
            memcpy(wifi_sta_config->sta.ssid, wifi_config_buff,
wifi_config_size);
            // Load Password
            wifi_config_size = sizeof(wifi_sta_config->sta.password);
            esp_err = nvs_get_blob(handle, "password",
wifi_config_buff, &wifi_config_size);
            if (esp_err != ESP_OK)
            {
                free(wifi_config_buff);
                printf("app_nvs_load_sta_creds: (%s) retrieving
password!\n", esp_err_to_name(esp_err));
                return false;
            }
            memcpy(wifi_sta_config->sta.password, wifi_config_buff,
wifi_config_size);
            free(wifi_config_buff);
            nvs_close(handle);
            printf("app_nvs_load_sta_creds: SSID: %s Password: %s\n",
wifi_sta_config->sta.ssid, wifi_sta_config->sta.password);
            return wifi_sta_config->sta.ssid[0] != '\0';
        }
        else
        {
            return false;
        }
}
esp_err_t app_nvs_clear_sta_creds(void)
{
        nvs_handle handle;
        esp_err_t esp_err;
        ESP_LOGI(TAG, "app_nvs_clear_sta_creds: Clearing Wifi station
mode credentials from flash");
```

```
	esp_err = nvs_open(app_nvs_sta_creds_namespace,
NVS_READWRITE, &handle);
	if (esp_err != ESP_OK)
	{
		printf("app_nvs_clear_sta_creds: Error (%s) opening NVS
handle!\n", esp_err_to_name(esp_err));
		return esp_err;
	}
	// Erase credentials
	esp_err = nvs_erase_all(handle);
	if (esp_err != ESP_OK)
	{
		printf("app_nvs_clear_sta_creds: Error (%s) erasing station
mode credentials!\n", esp_err_to_name(esp_err));
		return esp_err;
	}
	// Commit clearing credentials from NVS
	esp_err = nvs_commit(handle);
	if (esp_err != ESP_OK)
	{
		printf("app_nvs_clear_sta_creds: Error (%s) NVS
commit!\n", esp_err_to_name(esp_err));
		return esp_err;
	}
	nvs_close(handle);
	printf("app_nvs_clear_sta_creds: returned ESP_OK\n");
	return ESP_OK;
}
```

37.2 CODE OF APP_NVS.H:

```
/*
 * app_nvs.h
 *
```

```
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_APP_NVS_H_
#define MAIN_APP_NVS_H_
/**
 * Saves station mode Wifi credentials to NVS
 * @return ESP_OK if successful.
 */
esp_err_t app_nvs_save_sta_creds(void);
/**
 * Loads the previously saved credentials from NVS.
 * @return true if previously saved credentials were found.
 */
bool app_nvs_load_sta_creds(void);
/**
 * Clears station mode credentials from NVS
 * @return ESP_OK if successful.
 */
esp_err_t app_nvs_clear_sta_creds(void);
#endif /* MAIN_APP_NVS_H_ */
```

37.3 CODE OF HTTP_SERVER.C:

```
/*
 * http_server.c
 *
 *  Created on: Jul 11, 2024
 *      Author: HUZAIFASHAHBAZ
 */
 #include "esp_http_server.h"
#include "esp_log.h"
#include "esp_ota_ops.h"
#include "esp_wifi.h"
#include "sys/param.h"
```

```
#include "DHT22.h"
#include "http_server.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG[] = "http_server";
// Wifi connect status
static int g_wifi_connect_status = NONE;
// Firmware update status
static int g_fw_update_status = OTA_UPDATE_PENDING;
// HTTP server task handle
static httpd_handle_t http_server_handle = NULL;
// HTTP server monitor task handle
static TaskHandle_t task_http_server_monitor = NULL;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t http_server_monitor_queue_handle;
/**
 * ESP32 timer configuration passed to esp_timer_create.
 */
const esp_timer_create_args_t fw_update_reset_args = {
                .callback = &http_server_fw_update_reset_callback,
                .arg = NULL,
                .dispatch_method = ESP_TIMER_TASK,
                .name = "fw_update_reset"
};
esp_timer_handle_t fw_update_reset;
// Embedded files: JQuery, index.html, app.css, app.js and favicon.ico
files
extern const uint8_t jquery_3_3_1_min_js_start[]
        asm("_binary_jquery_3_3_1_min_js_start");
extern const uint8_t jquery_3_3_1_min_js_end[]
        asm("_binary_jquery_3_3_1_min_js_end");
extern const uint8_t index_html_start[]
        asm("_binary_index_html_start");
```

```
extern const uint8_t index_html_end[]
	asm("_binary_index_html_end");
extern const uint8_t app_css_start[]
	asm("_binary_app_css_start");
extern const uint8_t app_css_end[]
	asm("_binary_app_css_end");
extern const uint8_t app_js_start[]
	asm("_binary_app_js_start");
extern const uint8_t app_js_end[]
	asm("_binary_app_js_end");
extern const uint8_t favicon_ico_start[]
	asm("_binary_favicon_ico_start");
extern const uint8_t favicon_ico_end[]
	asm("_binary_favicon_ico_end");
/**
 * Checks the g_fw_update_status and creates the fw_update_reset timer
if g_fw_update_status is true.
 */
static void http_server_fw_update_reset_timer(void)
{
	if (g_fw_update_status == OTA_UPDATE_SUCCESSFUL)
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
updated successful starting FW update reset timer");
		// Give the web page a chance to receive an acknowledge
back and initialize the timer

	ESP_ERROR_CHECK(esp_timer_create(&fw_update_reset_args,
&fw_update_reset));

	ESP_ERROR_CHECK(esp_timer_start_once(fw_update_reset,
8000000));
	}
	else
	{
```

```
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
update unsuccessful");
	}
}
/**
 * HTTP server monitor task used to track events of the HTTP server
 * @param pvParameters parameter which can be passed to the task.
 */
static void http_server_monitor(void *parameter)
{
	http_server_queue_message_t msg;
	for (;;)
	{
		if (xQueueReceive(http_server_monitor_queue_handle,
&msg, portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case HTTP_MSG_WIFI_CONNECT_INIT:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_INIT");
					g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECTING;
					break;
				case HTTP_MSG_WIFI_CONNECT_SUCCESS:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_SUCCESS");
					g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_SUCCESS;
					break;
				case HTTP_MSG_WIFI_CONNECT_FAIL:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_FAIL");
					g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_FAILED;
```

```
                                break;
                        case HTTP_MSG_WIFI_USER_DISCONNECT:
                                ESP_LOGI(TAG,
"HTTP_MSG_WIFI_USER_DISCONNECT");
                                g_wifi_connect_status =
HTTP_WIFI_STATUS_DISCONNECTED;
                                break;
                        case
HTTP_MSG_OTA_UPDATE_SUCCESSFUL:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_SUCCESSFUL");
                                g_fw_update_status =
OTA_UPDATE_SUCCESSFUL;
                                http_server_fw_update_reset_timer();
                                break;
                        case HTTP_MSG_OTA_UPDATE_FAILED:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_FAILED");
                                g_fw_update_status =
OTA_UPDATE_FAILED;
                                break;
                        default:
                                break;
                }
            }
        }
}
/**
 * Jquery get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_jquery_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "Jquery requested");
```

```
	httpd_resp_set_type(req, "application/javascript");
	httpd_resp_send(req, (const char *)jquery_3_3_1_min_js_start,
jquery_3_3_1_min_js_end - jquery_3_3_1_min_js_start);
	return ESP_OK;
}
/**
 * Sends the index.html page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_index_html_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "index.html requested");
	httpd_resp_set_type(req, "text/html");
	httpd_resp_send(req, (const char *)index_html_start,
index_html_end - index_html_start);
	return ESP_OK;
}
/**
 * app.css get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_css_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.css requested");
	httpd_resp_set_type(req, "text/css");
	httpd_resp_send(req, (const char *)app_css_start, app_css_end -
app_css_start);
	return ESP_OK;
}
/**
 * app.js get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
```

```
 */
static esp_err_t http_server_app_js_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.js requested");
	httpd_resp_set_type(req, "application/javascript");
	httpd_resp_send(req, (const char *)app_js_start, app_js_end -
app_js_start);
	return ESP_OK;
}
/**
 * Sends the .ico (icon) file when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_favicon_ico_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "favicon.ico requested");
	httpd_resp_set_type(req, "image/x-icon");
	httpd_resp_send(req, (const char *)favicon_ico_start,
favicon_ico_end - favicon_ico_start);
	return ESP_OK;
}
/**
 * Receives the .bin file fia the web page and handles the firmware
update
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK, otherwise ESP_FAIL if timeout occurs and the
update cannot be started.
 */
esp_err_t http_server_OTA_update_handler(httpd_req_t *req)
{
	esp_ota_handle_t ota_handle;
	char ota_buff[1024];
	int content_length = req->content_len;
	int content_received = 0;
```

```
	int recv_len;
	bool is_req_body_started = false;
	bool flash_successful = false;
	const esp_partition_t *update_partition =
esp_ota_get_next_update_partition(NULL);
	do
	{
		// Read the data for the request
		if ((recv_len = httpd_req_recv(req, ota_buff,
MIN(content_length, sizeof(ota_buff)))) < 0)
		{
			// Check if timeout occurred
			if (recv_len == HTTPD_SOCK_ERR_TIMEOUT)
			{
				ESP_LOGI(TAG,
"http_server_OTA_update_handler: Socket Timeout");
				continue; ///> Retry receiving if timeout occurred
			}
			ESP_LOGI(TAG, "http_server_OTA_update_handler:
OTA other Error %d", recv_len);
			return ESP_FAIL;
		}
		printf("http_server_OTA_update_handler: OTA RX: %d of
%d\r", content_received, content_length);
		// Is this the first data we are receiving
		// If so, it will have the information in the header that we
need.
		if (!is_req_body_started)
		{
			is_req_body_started = true;
			// Get the location of the .bin file content (remove the
web form data)
			char *body_start_p = strstr(ota_buff, "\r\n\r\n") + 4;
			int body_part_len = recv_len - (body_start_p -
ota_buff);
```

```
                printf("http_server_OTA_update_handler: OTA file
size: %d\r\n", content_length);
                esp_err_t err = esp_ota_begin(update_partition,
OTA_SIZE_UNKNOWN, &ota_handle);
                if (err != ESP_OK)
                {
                        printf("http_server_OTA_update_handler: Error
with OTA begin, cancelling OTA\r\n");
                        return ESP_FAIL;
                }
                else
                {
                        printf("http_server_OTA_update_handler:
Writing to partition subtype %d at offset 0x%x\r\n", update_partition-
>subtype, update_partition->address);
                }
                // Write this first part of the data
                esp_ota_write(ota_handle, body_start_p,
body_part_len);
                content_received += body_part_len;
        }
        else
        {
                // Write OTA data
                esp_ota_write(ota_handle, ota_buff, recv_len);
                content_received += recv_len;
        }
    } while (recv_len > 0 && content_received < content_length);
    if (esp_ota_end(ota_handle) == ESP_OK)
    {
        // Lets update the partition
        if (esp_ota_set_boot_partition(update_partition) ==
ESP_OK)
        {
```

```
                        const esp_partition_t *boot_partition =
esp_ota_get_boot_partition();
                        ESP_LOGI(TAG, "http_server_OTA_update_handler:
Next boot partition subtype %d at offset 0x%x", boot_partition-
>subtype, boot_partition->address);
                        flash_successful = true;
                }
                else
                {
                        ESP_LOGI(TAG, "http_server_OTA_update_handler:
FLASHED ERROR!!!");
                }
        }
        else
        {
                ESP_LOGI(TAG, "http_server_OTA_update_handler:
esp_ota_end ERROR!!!");
        }
        // We won't update the global variables throughout the file, so send
the message about the status
        if (flash_successful) {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_SU
CCESSFUL); } else {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_FAI
LED); }
        return ESP_OK;
}
/**
 * OTA status handler responds with the firmware update status after the
OTA update is started
 * and responds with the compile time/date when the page is first
requested
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
```

```
esp_err_t http_server_OTA_status_handler(httpd_req_t *req)
{
	char otaJSON[100];
	ESP_LOGI(TAG, "OTAstatus requested");
	sprintf(otaJSON,
"{\"ota_update_status\":%d,\"compile_time\":\"%s\",\"compile_date\":\"
%s\"}", g_fw_update_status, __TIME__, __DATE__);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, otaJSON, strlen(otaJSON));
	return ESP_OK;
}
/**
 * DHT sensor readings JSON handler responds with DHT22 sensor data
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
static esp_err_t
http_server_get_dht_sensor_readings_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/dhtSensor.json requested");
	char dhtSensorJSON[100];
	sprintf(dhtSensorJSON,
"{\"temp\":\"%.1f\",\"humidity\":\"%.1f\"}", getTemperature(),
getHumidity());
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, dhtSensorJSON, strlen(dhtSensorJSON));
	return ESP_OK;
}
/**
 * wifiConnect.json handler is invoked after the connect button is
pressed
 * and handles receiving the SSID and password entered by the user
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
```

```
static esp_err_t http_server_wifi_connect_json_handler(httpd_req_t
*req)
{
	ESP_LOGI(TAG, "/wifiConnect.json requested");
	size_t len_ssid = 0, len_pass = 0;
	char *ssid_str = NULL, *pass_str = NULL;
	// Get SSID header
	len_ssid = httpd_req_get_hdr_value_len(req, "my-connect-ssid") +
1;
	if (len_ssid > 1)
	{
		ssid_str = malloc(len_ssid);
		if (httpd_req_get_hdr_value_str(req, "my-connect-ssid",
ssid_str, len_ssid) == ESP_OK)
		{
			ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
ssid: %s", ssid_str);
		}
	}
	// Get Password header
	len_pass = httpd_req_get_hdr_value_len(req, "my-connect-pwd")
+ 1;
	if (len_pass > 1)
	{
		pass_str = malloc(len_pass);
		if (httpd_req_get_hdr_value_str(req, "my-connect-pwd",
pass_str, len_pass) == ESP_OK)
		{
			ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
pwd: %s", pass_str);
		}
	}
```

```
	// Update the Wifi networks configuration and let the wifi
application know
	wifi_config_t* wifi_config = wifi_app_get_wifi_config();
	memset(wifi_config, 0x00, sizeof(wifi_config_t));
	memcpy(wifi_config->sta.ssid, ssid_str, len_ssid);
	memcpy(wifi_config->sta.password, pass_str, len_pass);
	wifi_app_send_message(WIFI_APP_MSG_CONNECTING_FRO
M_HTTP_SERVER);
	free(ssid_str);
	free(pass_str);
	return ESP_OK;
}
/**
 * wifiConnectStatus handler updates the connection status for the web
page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t
http_server_wifi_connect_status_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/wifiConnectStatus requested");
	char statusJSON[100];
	sprintf(statusJSON, "{\"wifi_connect_status\":%d}",
g_wifi_connect_status);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, statusJSON, strlen(statusJSON));
	return ESP_OK;
}
/**
 * wifiConnectInfo.json handler updates the web page with connection
information.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
```

```
static esp_err_t
http_server_get_wifi_connect_info_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/wifiConnectInfo.json requested");
	char ipInfoJSON[200];
	memset(ipInfoJSON, 0, sizeof(ipInfoJSON));
	char ip[IP4ADDR_STRLEN_MAX];
	char netmask[IP4ADDR_STRLEN_MAX];
	char gw[IP4ADDR_STRLEN_MAX];
	if (g_wifi_connect_status ==
HTTP_WIFI_STATUS_CONNECT_SUCCESS)
	{
		wifi_ap_record_t wifi_data;

	ESP_ERROR_CHECK(esp_wifi_sta_get_ap_info(&wifi_data));
		char *ssid = (char*)wifi_data.ssid;
		esp_netif_ip_info_t ip_info;
		ESP_ERROR_CHECK(esp_netif_get_ip_info(esp_netif_sta,
&ip_info));
		esp_ip4addr_ntoa(&ip_info.ip, ip,
IP4ADDR_STRLEN_MAX);
		esp_ip4addr_ntoa(&ip_info.netmask, netmask,
IP4ADDR_STRLEN_MAX);
		esp_ip4addr_ntoa(&ip_info.gw, gw,
IP4ADDR_STRLEN_MAX);
		sprintf(ipInfoJSON,
"{\"ip\":\"%s\",\"netmask\":\"%s\",\"gw\":\"%s\",\"ap\":\"%s\"}", ip,
netmask, gw, ssid);
	}
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, ipInfoJSON, strlen(ipInfoJSON));
	return ESP_OK;
}
/**
```

```
 * wifiDisconnect.json handler responds by sending a message to the
Wifi application to disconnect.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_wifi_disconnect_json_handler(httpd_req_t
*req)
{
	ESP_LOGI(TAG, "wifiDisconect.json requested");
	wifi_app_send_message(WIFI_APP_MSG_USER_REQUESTED
_STA_DISCONNECT);
	return ESP_OK;
}
/**
 * Sets up the default httpd server configuration.
 * @return http server instance handle if successful, NULL otherwise.
 */
static httpd_handle_t http_server_configure(void)
{
	// Generate the default configuration
	httpd_config_t config = HTTPD_DEFAULT_CONFIG();
	// Create HTTP server monitor task
	xTaskCreatePinnedToCore(&http_server_monitor,
"http_server_monitor", HTTP_SERVER_MONITOR_STACK_SIZE,
NULL, HTTP_SERVER_MONITOR_PRIORITY,
&task_http_server_monitor, HTTP_SERVER_MONITOR_CORE_ID);
	// Create the message queue
	http_server_monitor_queue_handle = xQueueCreate(3,
sizeof(http_server_queue_message_t));
	// The core that the HTTP server will run on
	config.core_id = HTTP_SERVER_TASK_CORE_ID;
	// Adjust the default priority to 1 less than the wifi application task
	config.task_priority = HTTP_SERVER_TASK_PRIORITY;
	// Bump up the stack size (default is 4096)
	config.stack_size = HTTP_SERVER_TASK_STACK_SIZE;
```

```
	// Increase uri handlers
	config.max_uri_handlers = 20;
	// Increase the timeout limits
	config.recv_wait_timeout = 10;
	config.send_wait_timeout = 10;
	ESP_LOGI(TAG,
			"http_server_configure: Starting server on port: '%d'
with task priority: '%d'",
			config.server_port,
			config.task_priority);
	// Start the httpd server
	if (httpd_start(&http_server_handle, &config) == ESP_OK)
	{
		ESP_LOGI(TAG, "http_server_configure: Registering URI
handlers");
		// register query handler
		httpd_uri_t jquery_js = {
				.uri = "/jquery-3.3.1.min.js",
				.method = HTTP_GET,
				.handler = http_server_jquery_handler,
				.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &jquery_js);
		// register index.html handler
		httpd_uri_t index_html = {
				.uri = "/",
				.method = HTTP_GET,
				.handler = http_server_index_html_handler,
				.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&index_html);
		// register app.css handler
		httpd_uri_t app_css = {
				.uri = "/app.css",
```

```
                .method = HTTP_GET,
                .handler = http_server_app_css_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_css);
        // register app.js handler
        httpd_uri_t app_js = {
                .uri = "/app.js",
                .method = HTTP_GET,
                .handler = http_server_app_js_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_js);
        // register favicon.ico handler
        httpd_uri_t favicon_ico = {
                .uri = "/favicon.ico",
                .method = HTTP_GET,
                .handler = http_server_favicon_ico_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&favicon_ico);
        // register OTAupdate handler
        httpd_uri_t OTA_update = {
                .uri = "/OTAupdate",
                .method = HTTP_POST,
                .handler = http_server_OTA_update_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_update);
        // register OTAstatus handler
        httpd_uri_t OTA_status = {
                .uri = "/OTAstatus",
                .method = HTTP_POST,
```

```
                        .handler = http_server_OTA_status_handler,
                        .user_ctx = NULL
            };
            httpd_register_uri_handler(http_server_handle,
&OTA_status);
            // register dhtSensor.json handler
            httpd_uri_t dht_sensor_json = {
                        .uri = "/dhtSensor.json",
                        .method = HTTP_GET,
                        .handler =
http_server_get_dht_sensor_readings_json_handler,
                        .user_ctx = NULL
            };
            httpd_register_uri_handler(http_server_handle,
&dht_sensor_json);
            // register wifiConnect.json handler
            httpd_uri_t wifi_connect_json = {
                        .uri = "/wifiConnect.json",
                        .method = HTTP_POST,
                        .handler =
http_server_wifi_connect_json_handler,
                        .user_ctx = NULL
            };
            httpd_register_uri_handler(http_server_handle,
&wifi_connect_json);
            // register wifiConnectStatus.json handler
            httpd_uri_t wifi_connect_status_json = {
                        .uri = "/wifiConnectStatus",
                        .method = HTTP_POST,
                        .handler =
http_server_wifi_connect_status_json_handler,
                        .user_ctx = NULL
            };
            httpd_register_uri_handler(http_server_handle,
&wifi_connect_status_json);
```

```
		// register wifiConnectInfo.json handler
		httpd_uri_t wifi_connect_info_json = {
				.uri = "/wifiConnectInfo.json",
				.method = HTTP_GET,
				.handler =
http_server_get_wifi_connect_info_json_handler,
				.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&wifi_connect_info_json);
		// register wifiDisconnect.json handler
		httpd_uri_t wifi_disconnect_json = {
				.uri = "/wifiDisconnect.json",
				.method = HTTP_DELETE,
				.handler =
http_server_wifi_disconnect_json_handler,
				.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&wifi_disconnect_json);
		return http_server_handle;
	}
	return NULL;
}
void http_server_start(void)
{
	if (http_server_handle == NULL)
	{
		http_server_handle = http_server_configure();
	}
}
void http_server_stop(void)
{
	if (http_server_handle)
	{
```

```
		httpd_stop(http_server_handle);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP
server");
		http_server_handle = NULL;
	}
	if (task_http_server_monitor)
	{
		vTaskDelete(task_http_server_monitor);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP server
monitor");
		task_http_server_monitor = NULL;
	}
}
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID)
{
	http_server_queue_message_t msg;
	msg.msgID = msgID;
	return xQueueSend(http_server_monitor_queue_handle, &msg,
portMAX_DELAY);
}
void http_server_fw_update_reset_callback(void *arg)
{
	ESP_LOGI(TAG, "http_server_fw_update_reset_callback: Timer
timed-out, restarting the device");
	esp_restart();
}
```

37.4 CODE OF HTTP_SERVER.H:

```
/*
 * http_server.h
 *
 *  Created on: Jul 12, 2024
```

```
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_HTTP_SERVER_H_
#define MAIN_HTTP_SERVER_H_
#define OTA_UPDATE_PENDING          0
#define OTA_UPDATE_SUCCESSFUL       1
#define OTA_UPDATE_FAILED           -1
/**
 * Connection status for Wifi
 */
typedef enum http_server_wifi_connect_status
{
        NONE = 0,
        HTTP_WIFI_STATUS_CONNECTING,
        HTTP_WIFI_STATUS_CONNECT_FAILED,
        HTTP_WIFI_STATUS_CONNECT_SUCCESS,
        HTTP_WIFI_STATUS_DISCONNECTED,
} http_server_wifi_connect_status_e;
/**
 * Messages for the HTTP monitor
 */
typedef enum http_server_message
{
        HTTP_MSG_WIFI_CONNECT_INIT = 0,
        HTTP_MSG_WIFI_CONNECT_SUCCESS,
        HTTP_MSG_WIFI_CONNECT_FAIL,
        HTTP_MSG_WIFI_USER_DISCONNECT,
        HTTP_MSG_OTA_UPDATE_SUCCESSFUL,
        HTTP_MSG_OTA_UPDATE_FAILED,
} http_server_message_e;
/**
 * Structure for the message queue
 */
typedef struct http_server_queue_message
{
```

```
        http_server_message_e msgID;
} http_server_queue_message_t;
/**
 * Sends a message to the queue
 * @param msgID message ID from the http_server_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue,
otherwise pdFALSE.
 * @note Expand the parameter list based on your requirements e.g. how
you've expanded the http_server_queue_message_t.
 */
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID);
/**
 * Starts the HTTP server.
 */
void http_server_start(void);
/**
 * Stops the HTTP server.
 */
void http_server_stop(void);
/**
 * Timer callback function which calls esp_restart upon successful
firmware update.
 */
void http_server_fw_update_reset_callback(void *arg);
#endif /* MAIN_HTTP_SERVER_H_ */
```

37.5 CODE OF WIFI_APP.C:

```
/*
 * wifi_app.c
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
```

```
 *
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "freertos/task.h"
#include "esp_err.h"
#include "esp_log.h"
#include "esp_wifi.h"
#include "lwip/netdb.h"
#include "app_nvs.h"
#include "http_server.h"
#include "rgb_led.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG [] = "wifi_app";
// Used for returning the WiFi configuration
wifi_config_t *wifi_config = NULL;
// Used to track the number for retries when a connection attempt fails
static int g_retry_number;
/**
 * Wifi application event group handle and status bits
 */
static EventGroupHandle_t wifi_app_event_group;
const int WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT
                = BIT0;
const int WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT
                = BIT1;
const int WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT
        = BIT2;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t wifi_app_queue_handle;
// netif objects for the station and access point
esp_netif_t* esp_netif_sta = NULL;
esp_netif_t* esp_netif_ap  = NULL;
/**
```

```
 * WiFi application event handler
 * @param arg data, aside from event data, that is passed to the handler
when it is called
 * @param event_base the base id of the event to register the handler for
 * @param event_id the id fo the event to register the handler for
 * @param event_data event data
 */
static void wifi_app_event_handler(void *arg, esp_event_base_t
event_base, int32_t event_id, void *event_data)
{
	if (event_base == WIFI_EVENT)
	{
		switch (event_id)
		{
			case WIFI_EVENT_AP_START:
				ESP_LOGI(TAG,
"WIFI_EVENT_AP_START");
				break;
			case WIFI_EVENT_AP_STOP:
				ESP_LOGI(TAG, "WIFI_EVENT_AP_STOP");
				break;
			case WIFI_EVENT_AP_STACONNECTED:
				ESP_LOGI(TAG,
"WIFI_EVENT_AP_STACONNECTED");
				break;
			case WIFI_EVENT_AP_STADISCONNECTED:
				ESP_LOGI(TAG,
"WIFI_EVENT_AP_STADISCONNECTED");
				break;
			case WIFI_EVENT_STA_START:
				ESP_LOGI(TAG,
"WIFI_EVENT_STA_START");
				break;
			case WIFI_EVENT_STA_CONNECTED:
```

```
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_CONNECTED");
                        break;
                case WIFI_EVENT_STA_DISCONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_DISCONNECTED");
                        wifi_event_sta_disconnected_t
*wifi_event_sta_disconnected =
(wifi_event_sta_disconnected_t*)malloc(sizeof(wifi_event_sta_disconn
ected_t));
                        *wifi_event_sta_disconnected =
*((wifi_event_sta_disconnected_t*)event_data);
                        printf("WIFI_EVENT_STA_DISCONNECTED,
reason code %d\n", wifi_event_sta_disconnected->reason);
                        if (g_retry_number <
MAX_CONNECTION_RETRIES)
                        {
                                esp_wifi_connect();
                                g_retry_number ++;
                        }
                        else
                        {

        wifi_app_send_message(WIFI_APP_MSG_STA_DISCONNECT
ED);
                        }
                        break;
                }
        }
        else if (event_base == IP_EVENT)
        {
                switch (event_id)
                {
                        case IP_EVENT_STA_GOT_IP:
                                ESP_LOGI(TAG, "IP_EVENT_STA_GOT_IP");
```

```
        wifi_app_send_message(WIFI_APP_MSG_STA_CONNECTED_
GOT_IP);
                        break;
            }
        }
}
/**
 * Initializes the WiFi application event handler for WiFi and IP events.
 */
static void wifi_app_event_handler_init(void)
{
        // Event loop for the WiFi driver
        ESP_ERROR_CHECK(esp_event_loop_create_default());
        // Event handler for the connection
        esp_event_handler_instance_t instance_wifi_event;
        esp_event_handler_instance_t instance_ip_event;
        ESP_ERROR_CHECK(esp_event_handler_instance_register(WIF
I_EVENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_wifi_event));
        ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_E
VENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_ip_event));
}
/**
 * Initializes the TCP stack and default WiFi configuration.
 */
static void wifi_app_default_wifi_init(void)
{
        // Initialize the TCP stack
        ESP_ERROR_CHECK(esp_netif_init());
        // Default WiFi config - operations must be in this order!
        wifi_init_config_t wifi_init_config =
WIFI_INIT_CONFIG_DEFAULT();
        ESP_ERROR_CHECK(esp_wifi_init(&wifi_init_config));
```

```
	ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_R
AM));
	esp_netif_sta = esp_netif_create_default_wifi_sta();
	esp_netif_ap = esp_netif_create_default_wifi_ap();
}
/**
 * Configures the WiFi access point settings and assigns the static IP to
the SoftAP.
 */
static void wifi_app_soft_ap_config(void)
{
	// SoftAP - WiFi access point configuration
	wifi_config_t ap_config =
	{
		.ap = {
				.ssid = WIFI_AP_SSID,
				.ssid_len = strlen(WIFI_AP_SSID),
				.password = WIFI_AP_PASSWORD,
				.channel = WIFI_AP_CHANNEL,
				.ssid_hidden = WIFI_AP_SSID_HIDDEN,
				.authmode = WIFI_AUTH_WPA2_PSK,
				.max_connection =
WIFI_AP_MAX_CONNECTIONS,
				.beacon_interval =
WIFI_AP_BEACON_INTERVAL,
			},
	};
	// Configure DHCP for the AP
	esp_netif_ip_info_t ap_ip_info;
	memset(&ap_ip_info, 0x00, sizeof(ap_ip_info));
	esp_netif_dhcps_stop(esp_netif_ap);				///>
must call this first
	inet_pton(AF_INET, WIFI_AP_IP, &ap_ip_info.ip);		///>
Assign access point's static IP, GW, and netmask
	inet_pton(AF_INET, WIFI_AP_GATEWAY, &ap_ip_info.gw);
```

```
	inet_pton(AF_INET, WIFI_AP_NETMASK,
&ap_ip_info.netmask);
	ESP_ERROR_CHECK(esp_netif_set_ip_info(esp_netif_ap,
&ap_ip_info));		///> Statically configure the network
interface
	ESP_ERROR_CHECK(esp_netif_dhcps_start(esp_netif_ap));
				///> Start the AP DHCP server (for
connecting stations e.g. your mobile device)
	ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_APST
A));				///> Setting the mode as Access Point /
Station Mode
	ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP,
&ap_config));		///> Set our configuration
	ESP_ERROR_CHECK(esp_wifi_set_bandwidth(WIFI_IF_AP,
WIFI_AP_BANDWIDTH));		///> Our default bandwidth 20
MHz
	ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_STA_POWER_SA
VE));				///> Power save set to "NONE"
}
/**
 * Connects the ESP32 to an external AP using the updated station
configuration
 */
static void wifi_app_connect_sta(void)
{
	ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA,
wifi_app_get_wifi_config()));
	ESP_ERROR_CHECK(esp_wifi_connect());
}
/**
 * Main task for the WiFi application
 * @param pvParameters parameter which can be passed to the task
 */
static void wifi_app_task(void *pvParameters)
{
```

```
	wifi_app_queue_message_t msg;
	EventBits_t eventBits;
	// Initialize the event handler
	wifi_app_event_handler_init();
	// Initialize the TCP/IP stack and WiFi config
	wifi_app_default_wifi_init();
	// SoftAP config
	wifi_app_soft_ap_config();
	// Start WiFi
	ESP_ERROR_CHECK(esp_wifi_start());
	// Send first event message
	wifi_app_send_message(WIFI_APP_MSG_LOAD_SAVED_CRE
DENTIALS);
	for (;;)
	{
		if (xQueueReceive(wifi_app_queue_handle, &msg,
portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case
WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS");
					if (app_nvs_load_sta_creds())
					{
						ESP_LOGI(TAG, "Loaded station
configuration");
						wifi_app_connect_sta();

	xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT);
					}
					else
					{
```

```
                                        ESP_LOGI(TAG, "Unable to load
station configuration");
                                }
                                // Next, start the web server

        wifi_app_send_message(WIFI_APP_MSG_START_HTTP_SERV
ER);
                                break;
                        case
WIFI_APP_MSG_START_HTTP_SERVER:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_START_HTTP_SERVER");
                                http_server_start();
                                rgb_led_http_server_started();
                                break;
                        case
WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER");
                                xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
                                // Attempt a connection
                                wifi_app_connect_sta();
                                // Set current number of retries to zero
                                g_retry_number = 0;
                                // Let the HTTP server know about the
connection attempt

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_INIT);
                                break;
                        case
WIFI_APP_MSG_STA_CONNECTED_GOT_IP:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_CONNECTED_GOT_IP");
```

```
                                rgb_led_wifi_connected();

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_SUCCESS);
                                eventBits =
xEventGroupGetBits(wifi_app_event_group);
                                if (eventBits &
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT) ///> Save
STA creds only if connecting from the http server (not loaded from
NVS)
                                {

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT); ///>
Clear the bits, in case we want to disconnect and reconnect, then start
again
                                }
                                else
                                {
                                        app_nvs_save_sta_creds();
                                }
                                if (eventBits &
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT)
                                {

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
                                }
                                break;
                        case
WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT");
                                xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT);
```

```
                                g_retry_number =
MAX_CONNECTION_RETRIES;

        ESP_ERROR_CHECK(esp_wifi_disconnect());
                                app_nvs_clear_sta_creds();
                                rgb_led_http_server_started(); ///> todo:
rename this status LED to a name more meaningful (to your liking)...
                                break;
                        case WIFI_APP_MSG_STA_DISCONNECTED:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED");
                                eventBits =
xEventGroupGetBits(wifi_app_event_group);
                                if (eventBits &
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT)
                                {
                                        ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT USING
SAVED CREDENTIALS");

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT);
                                        app_nvs_clear_sta_creds();
                                }
                                else if (eventBits &
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT)
                                {
                                        ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT FROM THE
HTTP SERVER");

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
```

```
	http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_FAIL);
				}
				else if (eventBits &
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT)
				{
					ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: USER REQUESTED
DISCONNECTION");

	xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT);

	http_server_monitor_send_message(HTTP_MSG_WIFI_USER_D
ISCONNECT);
				}
				else
				{
					ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT FAILED,
CHECK WIFI ACCESS POINT AVAILABILITY");
					// Adjust this case to your needs -
maybe you want to keep trying to connect...
				}
				break;
			default:
				break;
			}
		}
	}
}
BaseType_t wifi_app_send_message(wifi_app_message_e msgID)
{
	wifi_app_queue_message_t msg;
```

```
	msg.msgID = msgID;
	return xQueueSend(wifi_app_queue_handle, &msg,
portMAX_DELAY);
}
wifi_config_t* wifi_app_get_wifi_config(void)
{
	return wifi_config;
}
void wifi_app_start(void)
{
	ESP_LOGI(TAG, "STARTING WIFI APPLICATION");
	// Start WiFi started LED
	rgb_led_wifi_app_started();
	// Disable default WiFi logging messages
	esp_log_level_set("wifi", ESP_LOG_NONE);
	// Allocate memory for the wifi configuration
	wifi_config = (wifi_config_t*)malloc(sizeof(wifi_config_t));
	memset(wifi_config, 0x00, sizeof(wifi_config_t));
	// Create message queue
	wifi_app_queue_handle = xQueueCreate(3,
sizeof(wifi_app_queue_message_t));
	// Create Wifi application event group
	wifi_app_event_group = xEventGroupCreate();
	// Start the WiFi application task
	xTaskCreatePinnedToCore(&wifi_app_task, "wifi_app_task",
WIFI_APP_TASK_STACK_SIZE, NULL,
WIFI_APP_TASK_PRIORITY, NULL,
WIFI_APP_TASK_CORE_ID);
}
```

37.5 CODE OF WIFI_APP.H:

```
/*
 * wifi_app.h
 *
```

```
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_WIFI_APP_H_
#define MAIN_WIFI_APP_H_
#include "esp_netif.h"
// WiFi application settings
#define WIFI_AP_SSID                    "HUZAIFASHAHBAZ"
            // AP name
#define WIFI_AP_PASSWORD                "huzaf12345"
        // AP password
#define WIFI_AP_CHANNEL                 1
        // AP channel
#define WIFI_AP_SSID_HIDDEN             0
        // AP visibility
#define WIFI_AP_MAX_CONNECTIONS         5
        // AP max clients
#define WIFI_AP_BEACON_INTERVAL         100
        // AP beacon: 100 milliseconds recommended
#define WIFI_AP_IP                      "192.168.0.1"           // AP
default IP
#define WIFI_AP_GATEWAY                 "192.168.0.1"
        // AP default Gateway (should be the same as the IP)
#define WIFI_AP_NETMASK                 "255.255.255.0"
        // AP netmask
#define WIFI_AP_BANDWIDTH               WIFI_BW_HT20
        // AP bandwidth 20 MHz (40 MHz is the other option)
#define WIFI_STA_POWER_SAVE             WIFI_PS_NONE
        // Power save not used
#define MAX_SSID_LENGTH                 32
        // IEEE standard maximum
#define MAX_PASSWORD_LENGTH                     64
            // IEEE standard maximum
#define MAX_CONNECTION_RETRIES          5
        // Retry number on disconnect
```

```
// netif object for the Station and Access Point
extern esp_netif_t* esp_netif_sta;
extern esp_netif_t* esp_netif_ap;
/**
 * Message IDs for the WiFi application task
 * @note Expand this based on your application requirements.
 */
typedef enum wifi_app_message
{
	WIFI_APP_MSG_START_HTTP_SERVER = 0,
	WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER,
	WIFI_APP_MSG_STA_CONNECTED_GOT_IP,
	WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT,
	WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS,
	WIFI_APP_MSG_STA_DISCONNECTED,
} wifi_app_message_e;
/**
 * Structure for the message queue
 * @note Expand this based on application requirements e.g. add another
type and parameter as required
 */
typedef struct wifi_app_queue_message
{
	wifi_app_message_e msgID;
} wifi_app_queue_message_t;
/**
 * Sends a message to the queue
 * @param msgID message ID from the wifi_app_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue,
otherwise pdFALSE.
 * @note Expand the parameter list based on your requirements e.g. how
you've expanded the wifi_app_queue_message_t.
 */
BaseType_t wifi_app_send_message(wifi_app_message_e msgID);
/**
```

```
 * Starts the WiFi RTOS task
 */
void wifi_app_start(void);
/**
 * Gets the wifi configuration
 */
wifi_config_t* wifi_app_get_wifi_config(void);
#endif /* MAIN_WIFI_APP_H_ */
```

CHAPTER#38 BUTTON WITH INTERRUPT AND SEAMAPHORE

38.1 DISCONNECTING WIFI AND CLEARING CREDENTIALS USING THE BOOT BUTTON ON THE DEVKIT:

The BOOT button will be configured to generate an interrupt on IO0. When the interrupt occurs, a message will be sent to the WiFi Application about the user request to disconnect/clear credentials. Upon receiving the message, the WiFi Application will check if there really is an active connection prior to disconnecting and clearing credentials.

38.2 GPIO INTERRUPT FOR WIFI RESET BUTTON:

Background info: an interrupt is a signal that indicates the occurrence of a specific event that requires immediate attention and blocks the normal program execution to run an interrupt service routine (ISR) –which reacts to the event that occurred.

•Note: ISRs must have a short execution

The GPIO interrupt in our case, we are configuring the interrupt to trigger on a falling edge signal, because when you press the BOOT

button, the pin connects to ground as shown here in the ESP32 DevKit-C schematic:

Figure 38.2.15:GPIO INTERRUPT FOR WIFI RESET BUTTON

Due to the short execution time of ISRs; a binary semaphore will be used to notify a FreeRTOS task, which will handle the actions performed when the button is pressed.

CHAPTER#39 CODES OF BUTTON WITH INTERRUPT AND SEAMAPHORE

39.1 CODE OF WIFI_APP.C:

```
/*
 * wifi_app.c
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "freertos/task.h"
#include "esp_err.h"
#include "esp_log.h"
#include "esp_wifi.h"
#include "lwip/netdb.h"
#include "app_nvs.h"
#include "http_server.h"
```

```
#include "rgb_led.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG [] = "wifi_app";
// Used for returning the WiFi configuration
wifi_config_t *wifi_config = NULL;
// Used to track the number for retries when a connection attempt fails
static int g_retry_number;
/**
 * Wifi application event group handle and status bits
 */
static EventGroupHandle_t wifi_app_event_group;
const int WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT
                = BIT0;
const int WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT
                = BIT1;
const int WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT
        = BIT2;
const int WIFI_APP_STA_CONNECTED_GOT_IP_BIT
                = BIT3;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t wifi_app_queue_handle;
// netif objects for the station and access point
esp_netif_t* esp_netif_sta = NULL;
esp_netif_t* esp_netif_ap  = NULL;
/**
 * WiFi application event handler
 * @param arg data, aside from event data, that is passed to the handler
when it is called
 * @param event_base the base id of the event to register the handler for
 * @param event_id the id fo the event to register the handler for
 * @param event_data event data
 */
```

```
static void wifi_app_event_handler(void *arg, esp_event_base_t
event_base, int32_t event_id, void *event_data)
{
	if (event_base == WIFI_EVENT)
	{
		switch (event_id)
		{
			case WIFI_EVENT_AP_START:
				ESP_LOGI(TAG,
"WIFI_EVENT_AP_START");
				break;
			case WIFI_EVENT_AP_STOP:
				ESP_LOGI(TAG, "WIFI_EVENT_AP_STOP");
				break;
			case WIFI_EVENT_AP_STACONNECTED:
				ESP_LOGI(TAG,
"WIFI_EVENT_AP_STACONNECTED");
				break;
			case WIFI_EVENT_AP_STADISCONNECTED:
				ESP_LOGI(TAG,
"WIFI_EVENT_AP_STADISCONNECTED");
				break;
			case WIFI_EVENT_STA_START:
				ESP_LOGI(TAG,
"WIFI_EVENT_STA_START");
				break;
			case WIFI_EVENT_STA_CONNECTED:
				ESP_LOGI(TAG,
"WIFI_EVENT_STA_CONNECTED");
				break;
			case WIFI_EVENT_STA_DISCONNECTED:
				ESP_LOGI(TAG,
"WIFI_EVENT_STA_DISCONNECTED");
				wifi_event_sta_disconnected_t
*wifi_event_sta_disconnected =
```

```
(wifi_event_sta_disconnected_t*)malloc(sizeof(wifi_event_sta_disconn
ected_t));
                        *wifi_event_sta_disconnected =
*((wifi_event_sta_disconnected_t*)event_data);
                        printf("WIFI_EVENT_STA_DISCONNECTED,
reason code %d\n", wifi_event_sta_disconnected->reason);
                        if (g_retry_number <
MAX_CONNECTION_RETRIES)
                        {
                                esp_wifi_connect();
                                g_retry_number ++;
                        }
                        else
                        {

        wifi_app_send_message(WIFI_APP_MSG_STA_DISCONNECT
ED);
                        }
                        break;
                }
        }
        else if (event_base == IP_EVENT)
        {
                switch (event_id)
                {
                        case IP_EVENT_STA_GOT_IP:
                                ESP_LOGI(TAG, "IP_EVENT_STA_GOT_IP");

        wifi_app_send_message(WIFI_APP_MSG_STA_CONNECTED_
GOT_IP);
                        break;
                }
        }
}
/**
```

```
 * Initializes the WiFi application event handler for WiFi and IP events.
 */
static void wifi_app_event_handler_init(void)
{
	// Event loop for the WiFi driver
	ESP_ERROR_CHECK(esp_event_loop_create_default());
	// Event handler for the connection
	esp_event_handler_instance_t instance_wifi_event;
	esp_event_handler_instance_t instance_ip_event;
	ESP_ERROR_CHECK(esp_event_handler_instance_register(WIF
I_EVENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_wifi_event));
	ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_E
VENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_ip_event));
}
/**
 * Initializes the TCP stack and default WiFi configuration.
 */
static void wifi_app_default_wifi_init(void)
{
	// Initialize the TCP stack
	ESP_ERROR_CHECK(esp_netif_init());
	// Default WiFi config - operations must be in this order!
	wifi_init_config_t wifi_init_config =
WIFI_INIT_CONFIG_DEFAULT();
	ESP_ERROR_CHECK(esp_wifi_init(&wifi_init_config));
	ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_R
AM));
	esp_netif_sta = esp_netif_create_default_wifi_sta();
	esp_netif_ap = esp_netif_create_default_wifi_ap();
}
/**
 * Configures the WiFi access point settings and assigns the static IP to
the SoftAP.
```

```
 */
static void wifi_app_soft_ap_config(void)
{
	// SoftAP - WiFi access point configuration
	wifi_config_t ap_config =
	{
		.ap = {
				.ssid = WIFI_AP_SSID,
				.ssid_len = strlen(WIFI_AP_SSID),
				.password = WIFI_AP_PASSWORD,
				.channel = WIFI_AP_CHANNEL,
				.ssid_hidden = WIFI_AP_SSID_HIDDEN,
				.authmode = WIFI_AUTH_WPA2_PSK,
				.max_connection =
WIFI_AP_MAX_CONNECTIONS,
				.beacon_interval =
WIFI_AP_BEACON_INTERVAL,
		},
	};
	// Configure DHCP for the AP
	esp_netif_ip_info_t ap_ip_info;
	memset(&ap_ip_info, 0x00, sizeof(ap_ip_info));
	esp_netif_dhcps_stop(esp_netif_ap);				///>
must call this first
	inet_pton(AF_INET, WIFI_AP_IP, &ap_ip_info.ip);		///>
Assign access point's static IP, GW, and netmask
	inet_pton(AF_INET, WIFI_AP_GATEWAY, &ap_ip_info.gw);
	inet_pton(AF_INET, WIFI_AP_NETMASK,
&ap_ip_info.netmask);
	ESP_ERROR_CHECK(esp_netif_set_ip_info(esp_netif_ap,
&ap_ip_info));		///> Statically configure the network
interface
	ESP_ERROR_CHECK(esp_netif_dhcps_start(esp_netif_ap));
				///> Start the AP DHCP server (for
connecting stations e.g. your mobile device)
```

```
	ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_APST
A));                                    ///> Setting the mode as Access Point /
Station Mode
	ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP,
&ap_config));                   ///> Set our configuration
	ESP_ERROR_CHECK(esp_wifi_set_bandwidth(WIFI_IF_AP,
WIFI_AP_BANDWIDTH));                ///> Our default bandwidth 20
MHz
	ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_STA_POWER_SA
VE));                                   ///> Power save set to "NONE"
}
/**
 * Connects the ESP32 to an external AP using the updated station
configuration
 */
static void wifi_app_connect_sta(void)
{
	ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA,
wifi_app_get_wifi_config()));
	ESP_ERROR_CHECK(esp_wifi_connect());
}
/**
 * Main task for the WiFi application
 * @param pvParameters parameter which can be passed to the task
 */
static void wifi_app_task(void *pvParameters)
{
	wifi_app_queue_message_t msg;
	EventBits_t eventBits;
	// Initialize the event handler
	wifi_app_event_handler_init();
	// Initialize the TCP/IP stack and WiFi config
	wifi_app_default_wifi_init();
	// SoftAP config
	wifi_app_soft_ap_config();
```

```
	// Start WiFi
	ESP_ERROR_CHECK(esp_wifi_start());
	// Send first event message
	wifi_app_send_message(WIFI_APP_MSG_LOAD_SAVED_CRE
DENTIALS);
	for (;;)
	{
		if (xQueueReceive(wifi_app_queue_handle, &msg,
portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case
WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS");
					if (app_nvs_load_sta_creds())
					{
						ESP_LOGI(TAG, "Loaded station
configuration");
						wifi_app_connect_sta();

	xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT);
					}
					else
					{
						ESP_LOGI(TAG, "Unable to load
station configuration");
					}
					// Next, start the web server

	wifi_app_send_message(WIFI_APP_MSG_START_HTTP_SERV
ER);
					break;
```

```
                        case
WIFI_APP_MSG_START_HTTP_SERVER:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_START_HTTP_SERVER");
                                http_server_start();
                                rgb_led_http_server_started();
                                break;
                        case
WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER");
                                xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
                                // Attempt a connection
                                wifi_app_connect_sta();
                                // Set current number of retries to zero
                                g_retry_number = 0;
                                // Let the HTTP server know about the
connection attempt

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_INIT);
                                break;
                        case
WIFI_APP_MSG_STA_CONNECTED_GOT_IP:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_CONNECTED_GOT_IP");
                                xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_STA_CONNECTED_GOT_IP_BIT);
                                rgb_led_wifi_connected();

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_SUCCESS);
                                eventBits =
xEventGroupGetBits(wifi_app_event_group);
```

```
                                if (eventBits &
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT) ///> Save
STA creds only if connecting from the http server (not loaded from
NVS)
                                {

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT); ///>
Clear the bits, in case we want to disconnect and reconnect, then start
again
                                }
                                else
                                {
                                        app_nvs_save_sta_creds();
                                }
                                if (eventBits &
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT)
                                {

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
                                }
                                break;
                        case
WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT");
                                eventBits =
xEventGroupGetBits(wifi_app_event_group);
                                if (eventBits &
WIFI_APP_STA_CONNECTED_GOT_IP_BIT)
                                {

        xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT);
```

```
                                        g_retry_number =
MAX_CONNECTION_RETRIES;

        ESP_ERROR_CHECK(esp_wifi_disconnect());
                                        app_nvs_clear_sta_creds();
                                        rgb_led_http_server_started(); ///>
todo: rename this status LED to a name more meaningful (to your
liking)...
                                }
                                break;
                        case WIFI_APP_MSG_STA_DISCONNECTED:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED");
                                eventBits =
xEventGroupGetBits(wifi_app_event_group);
                                if (eventBits &
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT)
                                {
                                        ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT USING
SAVED CREDENTIALS");

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT);
                                        app_nvs_clear_sta_creds();
                                }
                                else if (eventBits &
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT)
                                {
                                        ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT FROM THE
HTTP SERVER");

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
```

```
	http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_FAIL);
				}
				else if (eventBits &
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT)
				{
					ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: USER REQUESTED
DISCONNECTION");

	xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT);

	http_server_monitor_send_message(HTTP_MSG_WIFI_USER_D
ISCONNECT);
				}
				else
				{
					ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT FAILED,
CHECK WIFI ACCESS POINT AVAILABILITY");
					// Adjust this case to your needs -
maybe you want to keep trying to connect...
				}
				if (eventBits &
WIFI_APP_STA_CONNECTED_GOT_IP_BIT)
				{

	xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_STA_CONNECTED_GOT_IP_BIT);
				}
				break;
			default:
				break;
```

```
                }
            }
        }
}
BaseType_t wifi_app_send_message(wifi_app_message_e msgID)
{
        wifi_app_queue_message_t msg;
        msg.msgID = msgID;
        return xQueueSend(wifi_app_queue_handle, &msg,
portMAX_DELAY);
}
wifi_config_t* wifi_app_get_wifi_config(void)
{
        return wifi_config;
}
void wifi_app_start(void)
{
        ESP_LOGI(TAG, "STARTING WIFI APPLICATION");
        // Start WiFi started LED
        rgb_led_wifi_app_started();
        // Disable default WiFi logging messages
        esp_log_level_set("wifi", ESP_LOG_NONE);
        // Allocate memory for the wifi configuration
        wifi_config = (wifi_config_t*)malloc(sizeof(wifi_config_t));
        memset(wifi_config, 0x00, sizeof(wifi_config_t));
        // Create message queue
        wifi_app_queue_handle = xQueueCreate(3,
sizeof(wifi_app_queue_message_t));
        // Create Wifi application event group
        wifi_app_event_group = xEventGroupCreate();
        // Start the WiFi application task
        xTaskCreatePinnedToCore(&wifi_app_task, "wifi_app_task",
WIFI_APP_TASK_STACK_SIZE, NULL,
WIFI_APP_TASK_PRIORITY, NULL,
WIFI_APP_TASK_CORE_ID);
```

```
}
```

39.2 CODE OF WIFI_RESET_BUTTON.C:

```
/*
 * wifi_reset_button.c
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
 #include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "driver/gpio.h"
#include "esp_log.h"
#include "tasks_common.h"
#include "wifi_app.h"
#include "wifi_reset_button.h"
static const char TAG[] = "wifi_reset_button";
// Semaphore handle
SemaphoreHandle_t wifi_reset_semphore = NULL;
/**
 * ISR handler for the Wifi reset (BOOT) button
 * @param arg parameter which can be passed to the ISR handler.
 */
void IRAM_ATTR wifi_reset_button_isr_handler(void *arg)
{
        // Notify the button task
        xSemaphoreGiveFromISR(wifi_reset_semphore, NULL);
}
/**
 * Wifi reset button task reacts to a BOOT button event by sending a
message
 * to the Wifi application to disconnect from Wifi and clear the saved
credentials.
```

```
 * @param pvParam parameter which can be passed to the task.
 */
void wifi_reset_button_task(void *pvParam)
{
	for (;;)
	{
		if (xSemaphoreTake(wifi_reset_semphore,
portMAX_DELAY) ==  pdTRUE)
		{
			ESP_LOGI(TAG, "WIFI RESET BUTTON
INTERRUPT OCCURRED");
			// Send a message to disconnect Wifi and clear
credentials

	wifi_app_send_message(WIFI_APP_MSG_USER_REQUESTED
_STA_DISCONNECT);
			vTaskDelay(2000 / portTICK_PERIOD_MS);
		}
	}
}
void wifi_reset_button_config(void)
{
	// Create the binary semaphore
	wifi_reset_semphore = xSemaphoreCreateBinary();
	// Configure the button and set the direction
	gpio_pad_select_gpio(WIFI_RESET_BUTTON);
	gpio_set_direction(WIFI_RESET_BUTTON,
GPIO_MODE_INPUT);
	// Enable interrupt on the negative edge
	gpio_set_intr_type(WIFI_RESET_BUTTON,
GPIO_INTR_NEGEDGE);
	// Create the Wifi reset button task
	xTaskCreatePinnedToCore(&wifi_reset_button_task,
"wifi_reset_button", WIFI_RESET_BUTTON_TASK_STACK_SIZE,
```

```
NULL, WIFI_RESET_BUTTON_TASK_PRIORITY, NULL,
WIFI_RESET_BUTTON_TASK_CORE_ID);
	// Install gpio isr service
	gpio_install_isr_service(ESP_INTR_FLAG_DEFAULT);
	// Attach the interrupt service routine
	gpio_isr_handler_add(WIFI_RESET_BUTTON,
wifi_reset_button_isr_handler, NULL);
}
```

39.3 CODE OF WIFI_RESET_BUTTON.H:

```
/*
 * wifi_reset_button.h
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_WIFI_RESET_BUTTON_H_
#define MAIN_WIFI_RESET_BUTTON_H_
// Default interrupt flag
#define ESP_INTR_FLAG_DEFAULT        0
// Wifi reset button is the BOOT button on the DevKit
#define WIFI_RESET_BUTTON            0
/**
 * Configures Wifi reset button and interrupt configuration
 */
void wifi_reset_button_config(void);
#endif /* MAIN_WIFI_RESET_BUTTON_H_ */
```

CHAPTER#40 SNTP TIME SYNCHRONIZATION

40.1 SNTP TIME SYNCHRONIZATION:

Once the ESP32 has an internet connection (connected to an AP/Router), the SNTP task start function will be called.

The task start function will set off the FreeRTOS time synchronization task, which will call a function to obtain the updated time ->In this implementation, the task will keep synchronizing/checking that the time is up-to-date.
The obtain time function will initialize the SNTP service to query an SNTP server for the universal time –the obtain time function will reinitialize in the case the time is not up-to-date.
The local time zone will be set after SNTP service is initialized.
The web server will respond with the updated time once time service is initialized.

40.2 SNTP:

SNTP (Simple Network Time Protocol) is a protocol designed to synchronize the clock of devices connected to the internet.
Basic operation is as follows:
-The client device connects to the server using the UDP protocol on port 123.
-The client transmits a request packet to the server.
-The server responds with a time stamp packet.
-The client can then parse out the current date and time values.
•If the ESP32 is connected to the internet, it can get the date and time using SNTP.
Note: The Simple Network Time Protocol (SNTP) in ESP-IDF is supported by LWIP functions

40.3 Utilizing ESP-IDF for SNTP Time Synchronization:

After the FreeRTOS task kicks off and the obtain time function is called, the initialize SNTP function is called, where the first SNTP function used, configures the client in *poll* mode to query the server every n seconds->sntp_setoperatingmode(SNTP_OPMODE_POLL).
Also, within the initialize SNTP function, we will tell the client which server to use. A common choice is a cluster of servers from pool.ntp.org

->sntp_setservername(0, “pool.ntp.org).
Then we’ll initialize service->Using sntp_init().
Set the TZ variable and initialize the time zone conversion set the time zone e.g., setenv(“TZ”, “CET-1”, 1); and tzset(); to initialize the timezone conversion routine.

CHAPTER#41 TIME ZONES

41.1 AUSTRAILA TIME ZONE:

Region	Time Zone String
Melbourne,Canberra,Sydney	AEST-10AEDT-11,M10.5.0/02:00:00,M4.1.0/03:00:00
Perth	AWST-8AWDT-9,M10.5.0,M3.5.0/03:00:00
Brisbane	AEST-10
Adelaide	ACST-9:30ACDT-10:30,M10.5.0/02:00:00,M4.1.0/03:00:00
Darwin	ACST-9:30
Hobart	AEST-10AEDT-11,M10.1.0/02:00:00,M4.1.0/03:00:00

Figure 41.1.16AUSTAILA TIME ZONES

41.2 EUROPE TIME ZONE:

Amsterdam, Netherlands	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Athens, Greece	EET-2EEST-3,M3.5.0/03:00:00,M10.5.0/04:00:00
Barcelona, Spain	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Berlin, Germany	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Brussels, Belgium	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Budapest, Hungary	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Copenhagen, Denmark	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Dublin, Ireland	GMT+0IST-1,M3.5.0/01:00:00,M10.5.0/02:00:00
Geneva, Switzerland	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Helsinki, Finland	EET-2EEST-3,M3.5.0/03:00:00,M10.5.0/04:00:00
Kyiv, Ukraine	EET-2EEST,M3.5.0/3,M10.5.0/4
Lisbon, Portugal	WET-0WEST-1,M3.5.0/01:00:00,M10.5.0/02:00:00
London, Great Britain	GMT+0BST-1,M3.5.0/01:00:00,M10.5.0/02:00:00
Madrid, Spain	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Oslo, Norway	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Paris, France	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Prague, Czech Republic	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Roma, Italy	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Moscow, Russia	MSK-3MSD,M3.5.0/2,M10.5.0/3
Sofia, Bulgaria	EET-2EEST-3,M3.5.0/03:00:00,M10.5.0/04:00:00
St.Petersburg, Russia	MST-3MDT,M3.5.0/2,M10.5.0/3
Stockholm, Sweden	CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
Tallinn, Estonia	EET-2EEST-3,M3.5.0/03:00:00,M10.5.0/04:00:00
Warsaw, Poland	CET-1CEST,M3.5.0,M10.5.0/3

Figure 41.2.17:EUROPE TIME ZONE

41.3 NEW ZELAND TIME ZONE:

Region	Time Zone String
Auckland, Wellington	NZST-12NZDT-13,M10.1.0/02:00:00,M3.3.0/03:00:00

Figure 41.3.18:NEW ZELAND TIME ZONE

41.4 USA AND CANADA:

Region	Time Zone String
Hawaii Time	HAW10
Alaska Time	AKST9AKDT,M3.2.0,M11.1.0
Pacific Time	PST8PDT,M3.2.0,M11.1.0
Mountain Time	MST7MDT,M3.2.0,M11.1.0
Mountain Time (Arizona, no DST)	MST7
Central Time	CST6CDT,M3.2.0,M11.1.0
Eastern Time	EST5EDT,M3.2.0,M11.1.0
Atlantic Time	AST4ADT
Atlantic Time (New Brunswick)	AST4ADT,M4.1.0/00:01:00,M10.5.0/00:01:00
Newfoundland Time	NST+3:30NDT+2:30,M3.2.0/00:01:00,M11.1.0/00:01:00

Figure 41.4.19:USA AND CANADA TIME ZONE

41.5 ASIA TIME ZONE:

Region	Time Zone String
Jakarta	WIB-7
Singapore	SGT-8
Hong Kong	HKT-8
Ulaanbaatar, Mongolia	ULAT-8ULAST,M3.5.0/2,M9.5.0/2

Figure 41.5.20:ASIA TIME ZONE

41.6 CENTRAL AND SOUTH AMERICA:

Region	Time Zone String
Brazil, São Paulo	BRST+3BRDT+2,M10.3.0,M2.3.0
Colombia	UTC+5
Argentina	UTC+3
Central America	CST+6

Figure 41.6.21:CENTRAL AND SOUTH AMERICA TIME ZONES

CHAPTER#42 CODES OF SNTP TIME SYNCHRONIZATION

42.1 CODE OF MAIN.C:

```
/**
 * Application entry point.
 */
#include "esp_log.h"
#include "nvs_flash.h"
#include "DHT22.h"
#include "sntp_time_sync.h"
#include "wifi_app.h"
#include "wifi_reset_button.h"
```

```
static const char TAG[] = "main";
void wifi_application_connected_events(void)
{
	ESP_LOGI(TAG, "WiFi Application Connected!!");
	sntp_time_sync_task_start();
}
void app_main(void)
{
   // Initialize NVS
	esp_err_t ret = nvs_flash_init();
	if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND)
	{
		ESP_ERROR_CHECK(nvs_flash_erase());
		ret = nvs_flash_init();
	}
	ESP_ERROR_CHECK(ret);
	// Start Wifi
	wifi_app_start();
	// Configure Wifi reset button
	wifi_reset_button_config();
	// Start DHT22 Sensor task
	DHT22_task_start();
	// Set connected event callback
	wifi_app_set_callback(&wifi_application_connected_events);
}
```

42.2 CODE OF SNTP_TIME_SYNC.C:

```
/*
 * sntp_time_sync.c
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
```

```
 */
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "lwip/apps/sntp.h"
#include "tasks_common.h"
#include "http_server.h"
#include "sntp_time_sync.h"
#include "wifi_app.h"
static const char TAG[] = "sntp_time_sync";
// SNTP operating mode set status
static bool sntp_op_mode_set = false;
/**
 * Initialize SNTP service using SNTP_OPMODE_POLL mode.
 */
static void sntp_time_sync_init_sntp(void)
{
	ESP_LOGI(TAG, "Initializing the SNTP service");
	if (!sntp_op_mode_set)
	{
		// Set the operating mode
		sntp_setoperatingmode(SNTP_OPMODE_POLL);
		sntp_op_mode_set = true;
	}
	sntp_setservername(0, "pool.ntp.org");
	// Initialize the servers
	sntp_init();
	// Let the http_server know service is initialized
	http_server_monitor_send_message(HTTP_MSG_TIME_SERVIC
E_INITIALIZED);
}
/**
 * Gets the current time and if the current time is not up to date,
 * the sntp_time_synch_init_sntp function is called.
 */
```

```
static void sntp_time_sync_obtain_time(void)
{
	time_t now = 0;
	struct tm time_info = {0};
	time(&now);
	localtime_r(&now, &time_info);
	// Check the time, in case we need to initialize/reinitialize
	if (time_info.tm_year < (2016 - 1900))
	{
		sntp_time_sync_init_sntp();
		// Set the local time zone
		setenv("TZ", "CET-1CEST,M3.5.0,M10.5.0/3", 1);
		tzset();
	}
}
/**
 * The SNTP time synchronization task.
 * @param arg pvParam.
 */
static void sntp_time_sync(void *pvParam)
{
	while (1)
	{
		sntp_time_sync_obtain_time();
		vTaskDelay(10000 / portTICK_PERIOD_MS);
	}
	vTaskDelete(NULL);
}
char* sntp_time_sync_get_time(void)
{
	static char time_buffer[100] = {0};
	time_t now = 0;
	struct tm time_info = {0};
	time(&now);
	localtime_r(&now, &time_info);
```

```
	if (time_info.tm_year < (2016 - 1900))
	{
		ESP_LOGI(TAG, "Time is not set yet");
	}
	else
	{
		strftime(time_buffer, sizeof(time_buffer), "%d.%m.%Y
%H:%M:%S", &time_info);
		ESP_LOGI(TAG, "Current time info: %s", time_buffer);
	}
	return time_buffer;
}
void sntp_time_sync_task_start(void)
{
	xTaskCreatePinnedToCore(&sntp_time_sync, "sntp_time_sync",
SNTP_TIME_SYNC_TASK_STACK_SIZE, NULL,
SNTP_TIME_SYNC_TASK_PRIORITY, NULL,
SNTP_TIME_SYNC_TASK_CORE_ID);
}
```

42.3 CODE OF SNTP_TIME_SYNC.H:

```
/*
 * sntp_time_synch.h
 *
 *  Created on: Jul 12, 2024
 *	Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_SNTP_TIME_SYNC_H_
#define MAIN_SNTP_TIME_SYNC_H_
/**
 * Starts the NTP server synchronization task.
 */
void sntp_time_sync_task_start(void);
/**
```

```
 * Returns local time if set.
 * @return local time buffer.
 */
char* sntp_time_sync_get_time(void);
#endif /* MAIN_SNTP_TIME_SYNC_H_ */
```

42.4 CODE OF TASKS_COMMON.H:

```
/*
 * tasks_common.h
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_TASKS_COMMON_H_
#define MAIN_TASKS_COMMON_H_
// WiFi application task
#define WIFI_APP_TASK_STACK_SIZE            4096
#define WIFI_APP_TASK_PRIORITY              5
#define WIFI_APP_TASK_CORE_ID               0
// HTTP Server task
#define HTTP_SERVER_TASK_STACK_SIZE         8192
#define HTTP_SERVER_TASK_PRIORITY           4
#define HTTP_SERVER_TASK_CORE_ID            0
// HTTP Server Monitor task
#define HTTP_SERVER_MONITOR_STACK_SIZE      4096
#define HTTP_SERVER_MONITOR_PRIORITY        3
#define HTTP_SERVER_MONITOR_CORE_ID         0
// Wifi Reset Button task
#define WIFI_RESET_BUTTON_TASK_STACK_SIZE   2048
#define WIFI_RESET_BUTTON_TASK_PRIORITY     6
#define WIFI_RESET_BUTTON_TASK_CORE_ID      0
// DHT22 Sensor task
#define DHT22_TASK_STACK_SIZE               4096
#define DHT22_TASK_PRIORITY                 5
```

```
#define DHT22_TASK_CORE_ID                          1
// SNTP Time Sync task
#define SNTP_TIME_SYNC_TASK_STACK_SIZE              4096
#define SNTP_TIME_SYNC_TASK_PRIORITY        4
#define SNTP_TIME_SYNC_TASK_CORE_ID                 1
#endif /* MAIN_TASKS_COMMON_H_ */
```

42.4 CODE OF WIFI_APP.C:

```
/*
 * wifi_app.c
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "freertos/task.h"
#include "esp_err.h"
#include "esp_log.h"
#include "esp_wifi.h"
#include "lwip/netdb.h"
#include "app_nvs.h"
#include "http_server.h"
#include "rgb_led.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG [] = "wifi_app";
// WiFi application callback
static wifi_connected_event_callback_t wifi_connected_event_cb;
// Used for returning the WiFi configuration
wifi_config_t *wifi_config = NULL;
// Used to track the number for retries when a connection attempt fails
static int g_retry_number;
```

```
/**
 * Wifi application event group handle and status bits
 */
static EventGroupHandle_t wifi_app_event_group;
const int WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT
                = BIT0;
const int WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT
                = BIT1;
const int WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT
        = BIT2;
const int WIFI_APP_STA_CONNECTED_GOT_IP_BIT
                = BIT3;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t wifi_app_queue_handle;
// netif objects for the station and access point
esp_netif_t* esp_netif_sta = NULL;
esp_netif_t* esp_netif_ap  = NULL;
/**
 * WiFi application event handler
 * @param arg data, aside from event data, that is passed to the handler
when it is called
 * @param event_base the base id of the event to register the handler for
 * @param event_id the id fo the event to register the handler for
 * @param event_data event data
 */
static void wifi_app_event_handler(void *arg, esp_event_base_t
event_base, int32_t event_id, void *event_data)
{
        if (event_base == WIFI_EVENT)
        {
                switch (event_id)
                {
                        case WIFI_EVENT_AP_START:
                                ESP_LOGI(TAG,
"WIFI_EVENT_AP_START");
```

```
                        break;
                    case WIFI_EVENT_AP_STOP:
                        ESP_LOGI(TAG, "WIFI_EVENT_AP_STOP");
                        break;
                    case WIFI_EVENT_AP_STACONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_AP_STACONNECTED");
                        break;
                    case WIFI_EVENT_AP_STADISCONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_AP_STADISCONNECTED");
                        break;
                    case WIFI_EVENT_STA_START:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_START");
                        break;
                    case WIFI_EVENT_STA_CONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_CONNECTED");
                        break;
                    case WIFI_EVENT_STA_DISCONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_DISCONNECTED");
                        wifi_event_sta_disconnected_t
*wifi_event_sta_disconnected =
(wifi_event_sta_disconnected_t*)malloc(sizeof(wifi_event_sta_disconn
ected_t));
                        *wifi_event_sta_disconnected =
*((wifi_event_sta_disconnected_t*)event_data);
                        printf("WIFI_EVENT_STA_DISCONNECTED,
reason code %d\n", wifi_event_sta_disconnected->reason);
                        if (g_retry_number <
MAX_CONNECTION_RETRIES)
                        {
                            esp_wifi_connect();
```

```
                        g_retry_number ++;
                    }
                    else
                    {

        wifi_app_send_message(WIFI_APP_MSG_STA_DISCONNECT
ED);
                    }
                    break;
            }
        }
        else if (event_base == IP_EVENT)
        {
            switch (event_id)
            {
                case IP_EVENT_STA_GOT_IP:
                    ESP_LOGI(TAG, "IP_EVENT_STA_GOT_IP");

        wifi_app_send_message(WIFI_APP_MSG_STA_CONNECTED_
GOT_IP);
                    break;
            }
        }
}
/**
 * Initializes the WiFi application event handler for WiFi and IP events.
 */
static void wifi_app_event_handler_init(void)
{
        // Event loop for the WiFi driver
        ESP_ERROR_CHECK(esp_event_loop_create_default());
        // Event handler for the connection
        esp_event_handler_instance_t instance_wifi_event;
        esp_event_handler_instance_t instance_ip_event;
```

```
	ESP_ERROR_CHECK(esp_event_handler_instance_register(WIF
I_EVENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_wifi_event));
	ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_E
VENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_ip_event));
}
/**
 * Initializes the TCP stack and default WiFi configuration.
 */
static void wifi_app_default_wifi_init(void)
{
	// Initialize the TCP stack
	ESP_ERROR_CHECK(esp_netif_init());
	// Default WiFi config - operations must be in this order!
	wifi_init_config_t wifi_init_config =
WIFI_INIT_CONFIG_DEFAULT();
	ESP_ERROR_CHECK(esp_wifi_init(&wifi_init_config));
	ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_R
AM));
	esp_netif_sta = esp_netif_create_default_wifi_sta();
	esp_netif_ap = esp_netif_create_default_wifi_ap();
}
/**
 * Configures the WiFi access point settings and assigns the static IP to
the SoftAP.
 */
static void wifi_app_soft_ap_config(void)
{
	// SoftAP - WiFi access point configuration
	wifi_config_t ap_config =
	{
		.ap = {
				.ssid = WIFI_AP_SSID,
				.ssid_len = strlen(WIFI_AP_SSID),
```

```
                        .password = WIFI_AP_PASSWORD,
                        .channel = WIFI_AP_CHANNEL,
                        .ssid_hidden = WIFI_AP_SSID_HIDDEN,
                        .authmode = WIFI_AUTH_WPA2_PSK,
                        .max_connection =
WIFI_AP_MAX_CONNECTIONS,
                        .beacon_interval =
WIFI_AP_BEACON_INTERVAL,
                },
        };
        // Configure DHCP for the AP
        esp_netif_ip_info_t ap_ip_info;
        memset(&ap_ip_info, 0x00, sizeof(ap_ip_info));
        esp_netif_dhcps_stop(esp_netif_ap);                     ///>
must call this first
        inet_pton(AF_INET, WIFI_AP_IP, &ap_ip_info.ip);         ///>
Assign access point's static IP, GW, and netmask
        inet_pton(AF_INET, WIFI_AP_GATEWAY, &ap_ip_info.gw);
        inet_pton(AF_INET, WIFI_AP_NETMASK,
&ap_ip_info.netmask);
        ESP_ERROR_CHECK(esp_netif_set_ip_info(esp_netif_ap,
&ap_ip_info));                  ///> Statically configure the network
interface
        ESP_ERROR_CHECK(esp_netif_dhcps_start(esp_netif_ap));
                                ///> Start the AP DHCP server (for
connecting stations e.g. your mobile device)
        ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_APST
A));                            ///> Setting the mode as Access Point /
Station Mode
        ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP,
&ap_config));                   ///> Set our configuration
        ESP_ERROR_CHECK(esp_wifi_set_bandwidth(WIFI_IF_AP,
WIFI_AP_BANDWIDTH));            ///> Our default bandwidth 20
MHz
```

```
	ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_STA_POWER_SA
VE));	///> Power save set to "NONE"
}
/**
 * Connects the ESP32 to an external AP using the updated station
configuration
 */
static void wifi_app_connect_sta(void)
{
	ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA,
wifi_app_get_wifi_config()));
	ESP_ERROR_CHECK(esp_wifi_connect());
}
/**
 * Main task for the WiFi application
 * @param pvParameters parameter which can be passed to the task
 */
static void wifi_app_task(void *pvParameters)
{
	wifi_app_queue_message_t msg;
	EventBits_t eventBits;
	// Initialize the event handler
	wifi_app_event_handler_init();
	// Initialize the TCP/IP stack and WiFi config
	wifi_app_default_wifi_init();
	// SoftAP config
	wifi_app_soft_ap_config();
	// Start WiFi
	ESP_ERROR_CHECK(esp_wifi_start());
	// Send first event message
	wifi_app_send_message(WIFI_APP_MSG_LOAD_SAVED_CRE
DENTIALS);
	for (;;)
	{
```

```
		if (xQueueReceive(wifi_app_queue_handle, &msg,
portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case
WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS");
					if (app_nvs_load_sta_creds())
					{
						ESP_LOGI(TAG, "Loaded station
configuration");
						wifi_app_connect_sta();

	xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT);
					}
					else
					{
						ESP_LOGI(TAG, "Unable to load
station configuration");
					}
					// Next, start the web server

	wifi_app_send_message(WIFI_APP_MSG_START_HTTP_SERV
ER);
					break;
				case
WIFI_APP_MSG_START_HTTP_SERVER:
					ESP_LOGI(TAG,
"WIFI_APP_MSG_START_HTTP_SERVER");
					http_server_start();
					rgb_led_http_server_started();
					break;
```

```
                        case
WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER");
                                xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
                                // Attempt a connection
                                wifi_app_connect_sta();
                                // Set current number of retries to zero
                                g_retry_number = 0;
                                // Let the HTTP server know about the
connection attempt

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_INIT);
                                break;
                        case
WIFI_APP_MSG_STA_CONNECTED_GOT_IP:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_CONNECTED_GOT_IP");
                                xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_STA_CONNECTED_GOT_IP_BIT);
                                rgb_led_wifi_connected();

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_SUCCESS);
                                eventBits =
xEventGroupGetBits(wifi_app_event_group);
                                if (eventBits &
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT) ///> Save
STA creds only if connecting from the http server (not loaded from
NVS)
                                {

        xEventGroupClearBits(wifi_app_event_group,
```

```
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT); ///>
Clear the bits, in case we want to disconnect and reconnect, then start
again
                                }
                                else
                                {
                                        app_nvs_save_sta_creds();
                                }
                                if (eventBits &
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT)
                                {

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
                                }
                                // Check for connection callback
                                if (wifi_connected_event_cb)
                                {
                                        wifi_app_call_callback();
                                }
                                break;
                        case
WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT");
                                eventBits =
xEventGroupGetBits(wifi_app_event_group);
                                if (eventBits &
WIFI_APP_STA_CONNECTED_GOT_IP_BIT)
                                {

        xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT);
                                        g_retry_number =
MAX_CONNECTION_RETRIES;
```

```
        ESP_ERROR_CHECK(esp_wifi_disconnect());
                                        app_nvs_clear_sta_creds();
                                        rgb_led_http_server_started(); ///>
todo: rename this status LED to a name more meaningful (to your
liking)...
                                }
                                break;
                        case WIFI_APP_MSG_STA_DISCONNECTED:
                                ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED");
                                eventBits =
xEventGroupGetBits(wifi_app_event_group);
                                if (eventBits &
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT)
                                {
                                        ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT USING
SAVED CREDENTIALS");

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT);
                                        app_nvs_clear_sta_creds();
                                }
                                else if (eventBits &
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT)
                                {
                                        ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT FROM THE
HTTP SERVER");

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
```

```
	http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_FAIL);
		}
		else if (eventBits &
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT)
		{
			ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: USER REQUESTED
DISCONNECTION");

	xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT);

	http_server_monitor_send_message(HTTP_MSG_WIFI_USER_D
ISCONNECT);
		}
		else
		{
			ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT FAILED,
CHECK WIFI ACCESS POINT AVAILABILITY");
			// Adjust this case to your needs -
maybe you want to keep trying to connect...
		}
		if (eventBits &
WIFI_APP_STA_CONNECTED_GOT_IP_BIT)
		{

	xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_STA_CONNECTED_GOT_IP_BIT);
		}
		break;
	default:
		break;
```

```
                }
            }
        }
}
BaseType_t wifi_app_send_message(wifi_app_message_e msgID)
{
        wifi_app_queue_message_t msg;
        msg.msgID = msgID;
        return xQueueSend(wifi_app_queue_handle, &msg,
portMAX_DELAY);
}
wifi_config_t* wifi_app_get_wifi_config(void)
{
        return wifi_config;
}
void wifi_app_set_callback(wifi_connected_event_callback_t cb)
{
        wifi_connected_event_cb = cb;
}
void wifi_app_call_callback(void)
{
        wifi_connected_event_cb();
}
void wifi_app_start(void)
{
        ESP_LOGI(TAG, "STARTING WIFI APPLICATION");
        // Start WiFi started LED
        rgb_led_wifi_app_started();
        // Disable default WiFi logging messages
        esp_log_level_set("wifi", ESP_LOG_NONE);
        // Allocate memory for the wifi configuration
        wifi_config = (wifi_config_t*)malloc(sizeof(wifi_config_t));
        memset(wifi_config, 0x00, sizeof(wifi_config_t));
        // Create message queue
```

```
	wifi_app_queue_handle = xQueueCreate(3,
sizeof(wifi_app_queue_message_t));
	// Create Wifi application event group
	wifi_app_event_group = xEventGroupCreate();
	// Start the WiFi application task
	xTaskCreatePinnedToCore(&wifi_app_task, "wifi_app_task",
WIFI_APP_TASK_STACK_SIZE, NULL,
WIFI_APP_TASK_PRIORITY, NULL,
WIFI_APP_TASK_CORE_ID);
}
```

42.5 CODE OF WIFI_APP.H:

```
/*
 * wifi_app.h
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_WIFI_APP_H_
#define MAIN_WIFI_APP_H_
#include "esp_netif.h"
// Callback typedef
typedef void (*wifi_connected_event_callback_t)(void);
// WiFi application settings
#define WIFI_AP_SSID                    "HUZAIFASHAHBAZ"
                // AP name
#define WIFI_AP_PASSWORD                "huzaf12345"
	// AP password
#define WIFI_AP_CHANNEL                 1
	// AP channel
#define WIFI_AP_SSID_HIDDEN             0
	// AP visibility
#define WIFI_AP_MAX_CONNECTIONS         5
	// AP max clients
```

```
#define WIFI_AP_BEACON_INTERVAL		100
	// AP beacon: 100 milliseconds recommended
#define WIFI_AP_IP		"192.168.0.1"		// AP
default IP
#define WIFI_AP_GATEWAY		"192.168.0.1"
	// AP default Gateway (should be the same as the IP)
#define WIFI_AP_NETMASK		"255.255.255.0"
	// AP netmask
#define WIFI_AP_BANDWIDTH		WIFI_BW_HT20
	// AP bandwidth 20 MHz (40 MHz is the other option)
#define WIFI_STA_POWER_SAVE		WIFI_PS_NONE
	// Power save not used
#define MAX_SSID_LENGTH		32
	// IEEE standard maximum
#define MAX_PASSWORD_LENGTH		64
		// IEEE standard maximum
#define MAX_CONNECTION_RETRIES		5
	// Retry number on disconnect
// netif object for the Station and Access Point
extern esp_netif_t* esp_netif_sta;
extern esp_netif_t* esp_netif_ap;
/**
 * Message IDs for the WiFi application task
 * @note Expand this based on your application requirements.
 */
typedef enum wifi_app_message
{
	WIFI_APP_MSG_START_HTTP_SERVER = 0,
	WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER,
	WIFI_APP_MSG_STA_CONNECTED_GOT_IP,
	WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT,
	WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS,
	WIFI_APP_MSG_STA_DISCONNECTED,
} wifi_app_message_e;
/**
```

```
 * Structure for the message queue
 * @note Expand this based on application requirements e.g. add another
type and parameter as required
 */
typedef struct wifi_app_queue_message
{
        wifi_app_message_e msgID;
} wifi_app_queue_message_t;
/**
 * Sends a message to the queue
 * @param msgID message ID from the wifi_app_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue,
otherwise pdFALSE.
 * @note Expand the parameter list based on your requirements e.g. how
you've expanded the wifi_app_queue_message_t.
 */
BaseType_t wifi_app_send_message(wifi_app_message_e msgID);
/**
 * Starts the WiFi RTOS task
 */
void wifi_app_start(void);
/**
 * Gets the wifi configuration
 */
wifi_config_t* wifi_app_get_wifi_config(void);
/**
 * Sets the callback function.
 */
void wifi_app_set_callback(wifi_connected_event_callback_t cb);
/**
 * Calls the callback function.
 */
void wifi_app_call_callback(void);
#endif /* MAIN_WIFI_APP_H_ */
```

42.6 CODE OF APP.CSS:

```
body {
    background-color: #f1f1f1;
    font-family: HelveticaNeueRegular, HelveticaNeue-Regular,
"Helvetica Neue Regular", HelveticaNeue, "Helvetica Neue", Helvetica,
Arial;
    color: #0272B7;
}
h1 {
    display: block;
    text-align: center;
    margin: 0;
    padding: 10px;
    font-size: 1.7em;
    color: #1d3557;
}
h2 {
        font-size: 1.4em;
    color: #1d3557;
}
h3 {
        font-size: 1.2em;
    color: #1d3557;
}
```

```
h4 {
        font-size: 1em;
    color: #1d3557;
}
.gr {
        color: green;
}
.rd {
        color: red;
}
#local_time {
        display:inline;
        color: #1d3557;
}
#latest_firmware, #latest_firmware_label {
        display:inline;
}
#latest_firmware {
        color: #1d3557;
}
#temperature_reading {
        display:inline;
        color: #1d3557;
```

```
}
#humidity_reading {
	display:inline;
	color: #1d3557;
}
#connected_ap_label, #connected_ap {
	display:inline;
}
#ip_address_label, #netmask_label, #gateway_label {
	display:inline;
}
#wifi_connect_ip, #wifi_connect_netmask, #wifi_connect_gw {
	display:inline;
}
#connected_ap {
	color: #1d3557;
}
#wifi_connect_ip {
	color: #1d3557;
}
#wifi_connect_netmask {
	color: #1d3557;
}
```

```
#wifi_connect_gw {
        color: #1d3557;
}
#disconnect_wifi {
        display: none;
}
.buttons {
   padding: 5px;
}
input[type="button"] {
        font-size: 12px;
        border: 0;
   line-height: 2;
   padding: 0 5px;
   text-align: center;
   color: #ffffff;
   text-shadow: 1px 1px 1px #000;
   border-radius: 10px;
   background-color: RGBA(106,125,144,0.35);
   background-image: linear-gradient(to top left,
                                     rgba(0, 0, 0, .2),
                                     rgba(0, 0, 0, .2) 30%,
                                     rgba(0, 0, 0, 0));
```

```
    box-shadow: inset 2px 2px 3px rgba(255, 255, 255, .6),
                inset -2px -2px 3px rgba(0, 0, 0, .6);
}
input[type="button"]:hover {
    background-color: RGBA(112,164,222,0.68);
}
input[type="button"]:active {
    box-shadow: inset -2px -2px 3px rgba(255, 255, 255, .6),
                inset 2px 2px 3px rgba(0, 0, 0, .6);
}
```

42.7 CODE OF APP.JS:

```
/**
 * Add gobals here
 */
var seconds        = null;
var otaTimerVar =  null;
var wifiConnectInterval = null;
/**
 * Initialize functions here.
 */
$(document).ready(function(){
        getUpdateStatus();
        startDHTSensorInterval();
        startLocalTimeInterval();
        getConnectInfo();
        $("#connect_wifi").on("click", function(){
                checkCredentials();
        });
```

```
        $("#disconnect_wifi").on("click", function(){
                disconnectWifi();
        });
});
/**
 * Gets file name and size for display on the web page.
 */
function getFileInfo()
{
   var x = document.getElementById("selected_file");
   var file = x.files[0];
   document.getElementById("file_info").innerHTML = "<h4>File: " +
file.name + "<br>" + "Size: " + file.size + " bytes</h4>";
}
/**
 * Handles the firmware update.
 */
function updateFirmware()
{
   // Form Data
   var formData = new FormData();
   var fileSelect = document.getElementById("selected_file");
   if (fileSelect.files && fileSelect.files.length == 1)
       {
       var file = fileSelect.files[0];
       formData.set("file", file, file.name);
       document.getElementById("ota_update_status").innerHTML =
"Uploading " + file.name + ", Firmware Update in Progress...";
       // Http Request
       var request = new XMLHttpRequest();
       request.upload.addEventListener("progress", updateProgress);
       request.open('POST', "/OTAupdate");
       request.responseType = "blob";
       request.send(formData);
   }
```

```
        else
        {
       window.alert('Select A File First')
    }
}
/**
 * Progress on transfers from the server to the client (downloads).
 */
function updateProgress(oEvent)
{
    if (oEvent.lengthComputable)
        {
       getUpdateStatus();
    }
        else
        {
       window.alert('total size is unknown')
    }
}
/**
 * Posts the firmware udpate status.
 */
function getUpdateStatus()
{
    var xhr = new XMLHttpRequest();
    var requestURL = "/OTAstatus";
    xhr.open('POST', requestURL, false);
    xhr.send('ota_update_status');
    if (xhr.readyState == 4 && xhr.status == 200)
        {
       var response = JSON.parse(xhr.responseText);
                document.getElementById("latest_firmware").innerHTML =
response.compile_date + " - " + response.compile_time
                // If flashing was complete it will return a 1, else -1
```

```
                // A return of 0 is just for information on the Latest Firmware
request
        if (response.ota_update_status == 1)
                {
                // Set the countdown timer time
            seconds = 10;
            // Start the countdown timer
            otaRebootTimer();
        }
        else if (response.ota_update_status == -1)
                {
            document.getElementById("ota_update_status").innerHTML =
"!!! Upload Error !!!";
        }
    }
}
/**
 * Displays the reboot countdown.
 */
function otaRebootTimer()
{
    document.getElementById("ota_update_status").innerHTML = "OTA
Firmware Update Complete. This page will close shortly, Rebooting in:
" + seconds;
    if (--seconds == 0)
         {
        clearTimeout(otaTimerVar);
        window.location.reload();
    }
         else
         {
        otaTimerVar = setTimeout(otaRebootTimer, 1000);
    }
}
/**
```

```
 * Gets DHT22 sensor temperature and humidity values for display on
the web page.
 */
function getDHTSensorValues()
{
	$.getJSON('/dhtSensor.json', function(data) {
		$("#temperature_reading").text(data["temp"]);
		$("#humidity_reading").text(data["humidity"]);
	});
}
/**
 * Sets the interval for getting the updated DHT22 sensor values.
 */
function startDHTSensorInterval()
{
	setInterval(getDHTSensorValues, 5000);
}
/**
 * Clears the connection status interval.
 */
function stopWifiConnectStatusInterval()
{
	if (wifiConnectInterval != null)
	{
		clearInterval(wifiConnectInterval);
		wifiConnectInterval = null;
	}
}
/**
 * Gets the WiFi connection status.
 */
function getWifiConnectStatus()
{
	var xhr = new XMLHttpRequest();
	var requestURL = "/wifiConnectStatus";
```

```
	xhr.open('POST', requestURL, false);
	xhr.send('wifi_connect_status');
	if (xhr.readyState == 4 && xhr.status == 200)
	{
		var response = JSON.parse(xhr.responseText);

	document.getElementById("wifi_connect_status").innerHTML =
"Connecting...";
		if (response.wifi_connect_status == 2)
		{

	document.getElementById("wifi_connect_status").innerHTML =
"<h4 class='rd'>Failed to Connect. Please check your AP credentials and
compatibility</h4>";
			stopWifiConnectStatusInterval();
		}
		else if (response.wifi_connect_status == 3)
		{

	document.getElementById("wifi_connect_status").innerHTML =
"<h4 class='gr'>Connection Success!</h4>";
			stopWifiConnectStatusInterval();
			getConnectInfo();
		}
	}
}
/**
 * Starts the interval for checking the connection status.
 */
function startWifiConnectStatusInterval()
{
	wifiConnectInterval = setInterval(getWifiConnectStatus, 2800);
}
/**
```

```
 * Connect WiFi function called using the SSID and password entered
into the text fields.
 */
function connectWifi()
{
	// Get the SSID and password
	selectedSSID = $("#connect_ssid").val();
	pwd = $("#connect_pass").val();
	$.ajax({
		url: '/wifiConnect.json',
		dataType: 'json',
		method: 'POST',
		cache: false,
		headers: {'my-connect-ssid': selectedSSID, 'my-connect-
pwd': pwd},
		data: {'timestamp': Date.now()}
	});
	startWifiConnectStatusInterval();
}
/**
 * Checks credentials on connect_wifi button click.
 */
function checkCredentials()
{
	errorList = "";
	credsOk = true;
	selectedSSID = $("#connect_ssid").val();
	pwd = $("#connect_pass").val();
	if (selectedSSID == "")
	{
		errorList += "<h4 class='rd'>SSID cannot be empty!</h4>";
		credsOk = false;
	}
	if (pwd == "")
	{
```

```
		errorList += "<h4 class='rd'>Password cannot be
empty!</h4>";
		credsOk = false;
	}
	if (credsOk == false)
	{
		$("#wifi_connect_credentials_errors").html(errorList);
	}
	else
	{
		$("#wifi_connect_credentials_errors").html("");
		connectWifi();
	}
}
/**
 * Shows the WiFi password if the box is checked.
 */
function showPassword()
{
	var x = document.getElementById("connect_pass");
	if (x.type === "password")
	{
		x.type = "text";
	}
	else
	{
		x.type = "password";
	}
}
/**
 * Gets the connection information for displaying on the web page.
 */
function getConnectInfo()
{
	$.getJSON('/wifiConnectInfo.json', function(data)
```

```
	{
		$("#connected_ap_label").html("Connected to: ");
		$("#connected_ap").text(data["ap"]);
		$("#ip_address_label").html("IP Address: ");
		$("#wifi_connect_ip").text(data["ip"]);
		$("#netmask_label").html("Netmask: ");
		$("#wifi_connect_netmask").text(data["netmask"]);
		$("#gateway_label").html("Gateway: ");
		$("#wifi_connect_gw").text(data["gw"]);
		document.getElementById('disconnect_wifi').style.display =
'block';
	});
}
/**
 * Disconnects Wifi once the disconnect button is pressed and reloads
the web page.
 */
function disconnectWifi()
{
	$.ajax({
		url: '/wifiDisconnect.json',
		dataType: 'json',
		method: 'DELETE',
		cache: false,
		data: { 'timestamp': Date.now() }
	});
	// Update the web page
	setTimeout("location.reload(true);", 2000);
}
/**
 * Sets the interval for displaying local time.
 */
function startLocalTimeInterval()
{
	setInterval(getLocalTime, 10000);
```

```
}
/**
 * Gets the local time.
 * @note connect the ESP32 to the internet and the time will be updated.
 */
function getLocalTime()
{
	$.getJSON('/localTime.json', function(data) {
		$("#local_time").text(data["time"]);
	});
}
```

42.8 CODE OF INDEX.HTML:

```
<!DOCTYPE html>
<html lang="en">
	<head>
		<meta charset="utf-8"/>
		<meta name="viewport" content="width=device-width,
initial-scale=1.0, user-scalable=no">
		<meta name="apple-mobile-web-app-capable" content="yes"
/>
		<script src='jquery-3.3.1.min.js'></script>
		<link rel="stylesheet" href="app.css">
		<script async src="app.js"></script>
		<title>ESP32 Udemy Course</title>
	</head>
	<body>
	<header>
		<h1>ESP32 Application Development</h1>
	</header>
	<div id="LocalTime">
		<h2>SNTP Time Synchronization</h2>
		<label for="local_time">Connect to WiFi for Local Time:
</label>
```

```
            <div id="local_time"></div>
        </div>
        <hr>
        <div id="OTA">
        <h2>ESP32 Firmware Update</h2>
            <label id="latest_firmware_label">Latest Firmware: </label>
            <div id="latest_firmware"></div>
            <input type="file" id="selected_file" accept=".bin"
style="display: none;" onchange="getFileInfo()" />
            <div class="buttons">
                <input type="button" value="Select File"
onclick="document.getElementById('selected_file').click();" />
                <input type="button" value="Update Firmware"
onclick="updateFirmware()" />
            </div>
            <h4 id="file_info"></h4>
            <h4 id="ota_update_status"></h4>
        </div>
        <hr>
        <div id="DHT22Sensor">
            <h2>DHT22 Sensor Readings</h2>
            <label for="temperature_reading">Temperature: </label>
            <div id="temperature_reading"></div>
            <label for="humidity_reading">Humidity: </label>
            <div id="humidity_reading"></div>
        </div>
        <hr>
        <div id="WiFiConnect">
            <h2>ESP32 WiFi Connect</h2>
            <section>
                <input id="connect_ssid" type="text" maxlength="32"
placeholder="SSID" value="">
                <input id="connect_pass" type="password"
maxlength="64" placeholder="Password" value="">
```

```
                        <input type="checkbox"
onclick="showPassword()">Show Password
                </section>
                <div class="buttons">
                        <input id="connect_wifi" type="button"
value="Connect" />
                </div>
                <div id="wifi_connect_credentials_errors"></div>
                <h4 id="wifi_connect_status"></h4>
        </div>
        <div id="ConnectInfo">
                <section>
                        <div id="connected_ap_label"></div> <div
id="connected_ap"></div>
                </section>
                <div id="ip_address_label"></div> <div
id="wifi_connect_ip"></div>
                <div id="netmask_label"></div> <div
id="wifi_connect_netmask"></div>
                <div id="gateway_label"></div> <div
id="wifi_connect_gw"></div>
                <div class="buttons">
                        <input id="disconnect_wifi" type="button"
value="Disconnect" />
                </div>
        </div>
        <hr>

        </body>
<html>
```

42.9 CODE OF HTTP_SERVER.H:

```
/*

 * http_server.h
```

```
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_HTTP_SERVER_H_
#define MAIN_HTTP_SERVER_H_
#define OTA_UPDATE_PENDING              0
#define OTA_UPDATE_SUCCESSFUL           1
#define OTA_UPDATE_FAILED               -1
/**
 * Connection status for Wifi
 */
typedef enum http_server_wifi_connect_status
{
        NONE = 0,
        HTTP_WIFI_STATUS_CONNECTING,
        HTTP_WIFI_STATUS_CONNECT_FAILED,
        HTTP_WIFI_STATUS_CONNECT_SUCCESS,
        HTTP_WIFI_STATUS_DISCONNECTED,
} http_server_wifi_connect_status_e;
/**
 * Messages for the HTTP monitor
 */
```

```
typedef enum http_server_message
{
	HTTP_MSG_WIFI_CONNECT_INIT = 0,
	HTTP_MSG_WIFI_CONNECT_SUCCESS,
	HTTP_MSG_WIFI_CONNECT_FAIL,
	HTTP_MSG_WIFI_USER_DISCONNECT,
	HTTP_MSG_OTA_UPDATE_SUCCESSFUL,
	HTTP_MSG_OTA_UPDATE_FAILED,
	HTTP_MSG_TIME_SERVICE_INITIALIZED,
} http_server_message_e;
/**
 * Structure for the message queue
 */
typedef struct http_server_queue_message
{
	http_server_message_e msgID;
} http_server_queue_message_t;
/**
 * Sends a message to the queue
 * @param msgID message ID from the http_server_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue, otherwise pdFALSE.
 * @note Expand the parameter list based on your requirements e.g. how you've expanded the http_server_queue_message_t.
```

```
 */
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID);
/**
 * Starts the HTTP server.
 */
void http_server_start(void);
/**
 * Stops the HTTP server.
 */
void http_server_stop(void);
/**
 * Timer callback function which calls esp_restart upon successful
firmware update.
 */
void http_server_fw_update_reset_callback(void *arg);
#endif /* MAIN_HTTP_SERVER_H_ */
```

42.10 CODE OF HTTP_SERVER.C:

```
/*
 * http_server.c
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
```

```
 */
#include "esp_http_server.h"
#include "esp_log.h"
#include "esp_ota_ops.h"
#include "esp_wifi.h"
#include "sys/param.h"
#include "DHT22.h"
#include "http_server.h"
#include "sntp_time_sync.h"
#include "tasks_common.h"
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG[] = "http_server";
// Wifi connect status
static int g_wifi_connect_status = NONE;
// Firmware update status
static int g_fw_update_status = OTA_UPDATE_PENDING;
// Local Time status
static bool g_is_local_time_set = false;
// HTTP server task handle
static httpd_handle_t http_server_handle = NULL;
// HTTP server monitor task handle
static TaskHandle_t task_http_server_monitor = NULL;
```

```
// Queue handle used to manipulate the main queue of events
static QueueHandle_t http_server_monitor_queue_handle;
/**
 * ESP32 timer configuration passed to esp_timer_create.
 */
const esp_timer_create_args_t fw_update_reset_args = {
            .callback = &http_server_fw_update_reset_callback,
            .arg = NULL,
            .dispatch_method = ESP_TIMER_TASK,
            .name = "fw_update_reset"
};
esp_timer_handle_t fw_update_reset;
// Embedded files: JQuery, index.html, app.css, app.js and favicon.ico
files
extern const uint8_t jquery_3_3_1_min_js_start[]
        asm("_binary_jquery_3_3_1_min_js_start");
extern const uint8_t jquery_3_3_1_min_js_end[]
        asm("_binary_jquery_3_3_1_min_js_end");
extern const uint8_t index_html_start[]
        asm("_binary_index_html_start");
extern const uint8_t index_html_end[]
        asm("_binary_index_html_end");
extern const uint8_t app_css_start[]
        asm("_binary_app_css_start");
```

```
extern const uint8_t app_css_end[]
	asm("_binary_app_css_end");

extern const uint8_t app_js_start[]
	asm("_binary_app_js_start");

extern const uint8_t app_js_end[]
	asm("_binary_app_js_end");

extern const uint8_t favicon_ico_start[]
	asm("_binary_favicon_ico_start");

extern const uint8_t favicon_ico_end[]
	asm("_binary_favicon_ico_end");

/**

 * Checks the g_fw_update_status and creates the fw_update_reset timer
if g_fw_update_status is true.

 */

static void http_server_fw_update_reset_timer(void)

{

	if (g_fw_update_status == OTA_UPDATE_SUCCESSFUL)

	{

		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
updated successful starting FW update reset timer");

		// Give the web page a chance to receive an acknowledge
back and initialize the timer


	ESP_ERROR_CHECK(esp_timer_create(&fw_update_reset_args,
&fw_update_reset));
```

```
	ESP_ERROR_CHECK(esp_timer_start_once(fw_update_reset,
8000000));
	}
	else
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
update unsuccessful");
	}
}
/**
 * HTTP server monitor task used to track events of the HTTP server
 * @param pvParameters parameter which can be passed to the task.
 */
static void http_server_monitor(void *parameter)
{
	http_server_queue_message_t msg;
	for (;;)
	{
		if (xQueueReceive(http_server_monitor_queue_handle,
&msg, portMAX_DELAY))
		{
			switch (msg.msgID)
			{
```

```
                    case HTTP_MSG_WIFI_CONNECT_INIT:
                        ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_INIT");
                        g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECTING;
                        break;
                    case HTTP_MSG_WIFI_CONNECT_SUCCESS:
                        ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_SUCCESS");
                        g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_SUCCESS;
                        break;
                    case HTTP_MSG_WIFI_CONNECT_FAIL:
                        ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_FAIL");
                        g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_FAILED;
                        break;
                    case HTTP_MSG_WIFI_USER_DISCONNECT:
                        ESP_LOGI(TAG,
"HTTP_MSG_WIFI_USER_DISCONNECT");
                        g_wifi_connect_status =
HTTP_WIFI_STATUS_DISCONNECTED;
                        break;
                    case
HTTP_MSG_OTA_UPDATE_SUCCESSFUL:
```

```
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_SUCCESSFUL");
                                g_fw_update_status =
OTA_UPDATE_SUCCESSFUL;
                                http_server_fw_update_reset_timer();
                                break;
                        case HTTP_MSG_OTA_UPDATE_FAILED:
                                ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_FAILED");
                                g_fw_update_status =
OTA_UPDATE_FAILED;
                                break;
                        case
HTTP_MSG_TIME_SERVICE_INITIALIZED:
                                ESP_LOGI(TAG,
"HTTP_MSG_TIME_SERVICE_INITIALIZED");
                                g_is_local_time_set = true;
                                break;
                        default:
                                break;
                }
            }
        }
}
/**
```

```
 * Jquery get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_jquery_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "Jquery requested");
	httpd_resp_set_type(req, "application/javascript");
	httpd_resp_send(req, (const char *)jquery_3_3_1_min_js_start,
jquery_3_3_1_min_js_end - jquery_3_3_1_min_js_start);
	return ESP_OK;
}
/**
 * Sends the index.html page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_index_html_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "index.html requested");
	httpd_resp_set_type(req, "text/html");
	httpd_resp_send(req, (const char *)index_html_start,
index_html_end - index_html_start);
	return ESP_OK;
```

```
}
/**
 * app.css get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_css_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.css requested");
	httpd_resp_set_type(req, "text/css");
	httpd_resp_send(req, (const char *)app_css_start, app_css_end -
app_css_start);
	return ESP_OK;
}
/**
 * app.js get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_js_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.js requested");
	httpd_resp_set_type(req, "application/javascript");
```

```
	httpd_resp_send(req, (const char *)app_js_start, app_js_end - app_js_start);
	return ESP_OK;
}
/**
 * Sends the .ico (icon) file when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_favicon_ico_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "favicon.ico requested");
	httpd_resp_set_type(req, "image/x-icon");
	httpd_resp_send(req, (const char *)favicon_ico_start, favicon_ico_end - favicon_ico_start);
	return ESP_OK;
}
/**
 * Receives the .bin file fia the web page and handles the firmware update
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK, otherwise ESP_FAIL if timeout occurs and the update cannot be started.
 */
```

```
esp_err_t http_server_OTA_update_handler(httpd_req_t *req)
{
	esp_ota_handle_t ota_handle;
	char ota_buff[1024];
	int content_length = req->content_len;
	int content_received = 0;
	int recv_len;
	bool is_req_body_started = false;
	bool flash_successful = false;
	const esp_partition_t *update_partition =
esp_ota_get_next_update_partition(NULL);
	do
	{
		// Read the data for the request
		if ((recv_len = httpd_req_recv(req, ota_buff,
MIN(content_length, sizeof(ota_buff)))) < 0)
		{
			// Check if timeout occurred
			if (recv_len == HTTPD_SOCK_ERR_TIMEOUT)
			{
				ESP_LOGI(TAG,
"http_server_OTA_update_handler: Socket Timeout");
				continue; ///> Retry receiving if timeout occurred
			}
```

```
                ESP_LOGI(TAG, "http_server_OTA_update_handler:
OTA other Error %d", recv_len);
                return ESP_FAIL;
        }
        printf("http_server_OTA_update_handler: OTA RX: %d of
%d\r", content_received, content_length);
        // Is this the first data we are receiving
        // If so, it will have the information in the header that we
need.
        if (!is_req_body_started)
        {
                is_req_body_started = true;
                // Get the location of the .bin file content (remove the
web form data)
                char *body_start_p = strstr(ota_buff, "\r\n\r\n") + 4;
                int body_part_len = recv_len - (body_start_p -
ota_buff);
                printf("http_server_OTA_update_handler: OTA file
size: %d\r\n", content_length);
                esp_err_t err = esp_ota_begin(update_partition,
OTA_SIZE_UNKNOWN, &ota_handle);
                if (err != ESP_OK)
                {
                        printf("http_server_OTA_update_handler: Error
with OTA begin, cancelling OTA\r\n");
```

```
                return ESP_FAIL;
            }
            else
            {
                printf("http_server_OTA_update_handler: Writing
to partition subtype %d at offset 0x%x\r\n", update_partition->subtype,
update_partition->address);
            }
            // Write this first part of the data
            esp_ota_write(ota_handle, body_start_p,
body_part_len);
            content_received += body_part_len;
        }
        else
        {
            // Write OTA data
            esp_ota_write(ota_handle, ota_buff, recv_len);
            content_received += recv_len;
        }
    } while (recv_len > 0 && content_received < content_length);
    if (esp_ota_end(ota_handle) == ESP_OK)
    {
        // Lets update the partition
```

```
            if (esp_ota_set_boot_partition(update_partition) ==
ESP_OK)
            {
                const esp_partition_t *boot_partition =
esp_ota_get_boot_partition();
                ESP_LOGI(TAG, "http_server_OTA_update_handler:
Next boot partition subtype %d at offset 0x%x", boot_partition-
>subtype, boot_partition->address);
                flash_successful = true;
            }
            else
            {
                ESP_LOGI(TAG, "http_server_OTA_update_handler:
FLASHED ERROR!!!");
            }
        }
        else
        {
            ESP_LOGI(TAG, "http_server_OTA_update_handler:
esp_ota_end ERROR!!!");
        }
        // We won't update the global variables throughout the file, so send
the message about the status
        if (flash_successful) {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_SUC
CESSFUL); } else {
```

```
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_FAI
LED); }
        return ESP_OK;
}
/**
 * OTA status handler responds with the firmware update status after the
OTA update is started
 * and responds with the compile time/date when the page is first
requested
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
esp_err_t http_server_OTA_status_handler(httpd_req_t *req)
{
        char otaJSON[100];
        ESP_LOGI(TAG, "OTAstatus requested");
        sprintf(otaJSON,
"{\"ota_update_status\":%d,\"compile_time\":\"%s\",\"compile_date\":\"
%s\"}", g_fw_update_status, __TIME__, __DATE__);
        httpd_resp_set_type(req, "application/json");
        httpd_resp_send(req, otaJSON, strlen(otaJSON));
        return ESP_OK;
}
/**
 * DHT sensor readings JSON handler responds with DHT22 sensor data
```

```
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
static esp_err_t
http_server_get_dht_sensor_readings_json_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "/dhtSensor.json requested");
        char dhtSensorJSON[100];
        sprintf(dhtSensorJSON,
"{\"temp\":\"%.1f\",\"humidity\":\"%.1f\"}", getTemperature(),
getHumidity());
        httpd_resp_set_type(req, "application/json");
        httpd_resp_send(req, dhtSensorJSON, strlen(dhtSensorJSON));
        return ESP_OK;
}
/**
 * wifiConnect.json handler is invoked after the connect button is
pressed
 * and handles receiving the SSID and password entered by the user
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_wifi_connect_json_handler(httpd_req_t
*req)
{
```

```
ESP_LOGI(TAG, "/wifiConnect.json requested");
size_t len_ssid = 0, len_pass = 0;
char *ssid_str = NULL, *pass_str = NULL;
// Get SSID header
len_ssid = httpd_req_get_hdr_value_len(req, "my-connect-ssid") +
1;
if (len_ssid > 1)
{
        ssid_str = malloc(len_ssid);
        if (httpd_req_get_hdr_value_str(req, "my-connect-ssid",
ssid_str, len_ssid) == ESP_OK)
        {
                ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
ssid: %s", ssid_str);
        }
}
// Get Password header
len_pass = httpd_req_get_hdr_value_len(req, "my-connect-pwd")
+ 1;
if (len_pass > 1)
{
        pass_str = malloc(len_pass);
        if (httpd_req_get_hdr_value_str(req, "my-connect-pwd",
pass_str, len_pass) == ESP_OK)
```

```
            {
                ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
pwd: %s", pass_str);
            }
        }
        // Update the Wifi networks configuration and let the wifi
application know
        wifi_config_t* wifi_config = wifi_app_get_wifi_config();
        memset(wifi_config, 0x00, sizeof(wifi_config_t));
        memcpy(wifi_config->sta.ssid, ssid_str, len_ssid);
        memcpy(wifi_config->sta.password, pass_str, len_pass);
        wifi_app_send_message(WIFI_APP_MSG_CONNECTING_FRO
M_HTTP_SERVER);
        free(ssid_str);
        free(pass_str);
        return ESP_OK;
}
/**
 * wifiConnectStatus handler updates the connection status for the web
page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
```

```
static esp_err_t
http_server_wifi_connect_status_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/wifiConnectStatus requested");
	char statusJSON[100];
	sprintf(statusJSON, "{\"wifi_connect_status\":%d}",
g_wifi_connect_status);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, statusJSON, strlen(statusJSON));
	return ESP_OK;
}
/**
 * wifiConnectInfo.json handler updates the web page with connection
information.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t
http_server_get_wifi_connect_info_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/wifiConnectInfo.json requested");
	char ipInfoJSON[200];
	memset(ipInfoJSON, 0, sizeof(ipInfoJSON));
	char ip[IP4ADDR_STRLEN_MAX];
```

```
	char netmask[IP4ADDR_STRLEN_MAX];
	char gw[IP4ADDR_STRLEN_MAX];
	if (g_wifi_connect_status ==
HTTP_WIFI_STATUS_CONNECT_SUCCESS)
	{
		wifi_ap_record_t wifi_data;

	ESP_ERROR_CHECK(esp_wifi_sta_get_ap_info(&wifi_data));
		char *ssid = (char*)wifi_data.ssid;
		esp_netif_ip_info_t ip_info;
		ESP_ERROR_CHECK(esp_netif_get_ip_info(esp_netif_sta,
&ip_info));
		esp_ip4addr_ntoa(&ip_info.ip, ip,
IP4ADDR_STRLEN_MAX);
		esp_ip4addr_ntoa(&ip_info.netmask, netmask,
IP4ADDR_STRLEN_MAX);
		esp_ip4addr_ntoa(&ip_info.gw, gw,
IP4ADDR_STRLEN_MAX);
		sprintf(ipInfoJSON,
"{\"ip\":\"%s\",\"netmask\":\"%s\",\"gw\":\"%s\",\"ap\":\"%s\"}", ip,
netmask, gw, ssid);
	}
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, ipInfoJSON, strlen(ipInfoJSON));
	return ESP_OK;
```

```
}
/**
 * wifiDisconnect.json handler responds by sending a message to the
Wifi application to disconnect.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_wifi_disconnect_json_handler(httpd_req_t
*req)
{
	ESP_LOGI(TAG, "wifiDisconect.json requested");
	wifi_app_send_message(WIFI_APP_MSG_USER_REQUESTED
_STA_DISCONNECT);
	return ESP_OK;
}
/**
 * localTime.json handler responds by sending the local time.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_get_local_time_json_handler(httpd_req_t
*req)
{
	ESP_LOGI(TAG, "/localTime.json requested");
```

```
	char localTimeJSON[100] = {0};
	if (g_is_local_time_set)
	{
		sprintf(localTimeJSON, "{\"time\":\"%s\"}",
sntp_time_sync_get_time());
	}
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, localTimeJSON, strlen(localTimeJSON));
	return ESP_OK;
}
/**
 * Sets up the default httpd server configuration.
 * @return http server instance handle if successful, NULL otherwise.
 */
static httpd_handle_t http_server_configure(void)
{
	// Generate the default configuration
	httpd_config_t config = HTTPD_DEFAULT_CONFIG();
	// Create HTTP server monitor task
	xTaskCreatePinnedToCore(&http_server_monitor,
"http_server_monitor", HTTP_SERVER_MONITOR_STACK_SIZE,
NULL, HTTP_SERVER_MONITOR_PRIORITY,
&task_http_server_monitor, HTTP_SERVER_MONITOR_CORE_ID);
	// Create the message queue
```

```
	http_server_monitor_queue_handle = xQueueCreate(3,
sizeof(http_server_queue_message_t));
	// The core that the HTTP server will run on
	config.core_id = HTTP_SERVER_TASK_CORE_ID;
	// Adjust the default priority to 1 less than the wifi application task
	config.task_priority = HTTP_SERVER_TASK_PRIORITY;
	// Bump up the stack size (default is 4096)
	config.stack_size = HTTP_SERVER_TASK_STACK_SIZE;
	// Increase uri handlers
	config.max_uri_handlers = 20;
	// Increase the timeout limits
	config.recv_wait_timeout = 10;
	config.send_wait_timeout = 10;
	ESP_LOGI(TAG,
			"http_server_configure: Starting server on port: '%d'
with task priority: '%d'",
			config.server_port,
			config.task_priority);
	// Start the httpd server
	if (httpd_start(&http_server_handle, &config) == ESP_OK)
	{
		ESP_LOGI(TAG, "http_server_configure: Registering URI
handlers");
		// register query handler
```

```
        httpd_uri_t jquery_js = {
                .uri = "/jquery-3.3.1.min.js",
                .method = HTTP_GET,
                .handler = http_server_jquery_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &jquery_js);
        // register index.html handler
        httpd_uri_t index_html = {
                .uri = "/",
                .method = HTTP_GET,
                .handler = http_server_index_html_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&index_html);
        // register app.css handler
        httpd_uri_t app_css = {
                .uri = "/app.css",
                .method = HTTP_GET,
                .handler = http_server_app_css_handler,
                .user_ctx = NULL
        };
```

```
        httpd_register_uri_handler(http_server_handle, &app_css);
        // register app.js handler
        httpd_uri_t app_js = {
                .uri = "/app.js",
                .method = HTTP_GET,
                .handler = http_server_app_js_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_js);
        // register favicon.ico handler
        httpd_uri_t favicon_ico = {
                .uri = "/favicon.ico",
                .method = HTTP_GET,
                .handler = http_server_favicon_ico_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&favicon_ico);
        // register OTAupdate handler
        httpd_uri_t OTA_update = {
                .uri = "/OTAupdate",
                .method = HTTP_POST,
                .handler = http_server_OTA_update_handler,
```

```
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_update);
        // register OTAstatus handler
        httpd_uri_t OTA_status = {
                .uri = "/OTAstatus",
                .method = HTTP_POST,
                .handler = http_server_OTA_status_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&OTA_status);
        // register dhtSensor.json handler
        httpd_uri_t dht_sensor_json = {
                .uri = "/dhtSensor.json",
                .method = HTTP_GET,
                .handler =
http_server_get_dht_sensor_readings_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&dht_sensor_json);
        // register wifiConnect.json handler
```

```
        httpd_uri_t wifi_connect_json = {
                .uri = "/wifiConnect.json",
                .method = HTTP_POST,
                .handler =
http_server_wifi_connect_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_connect_json);
        // register wifiConnectStatus.json handler
        httpd_uri_t wifi_connect_status_json = {
                .uri = "/wifiConnectStatus",
                .method = HTTP_POST,
                .handler =
http_server_wifi_connect_status_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_connect_status_json);
        // register wifiConnectInfo.json handler
        httpd_uri_t wifi_connect_info_json = {
                .uri = "/wifiConnectInfo.json",
                .method = HTTP_GET,
                .handler =
http_server_get_wifi_connect_info_json_handler,
```

```
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_connect_info_json);
        // register wifiDisconnect.json handler
        httpd_uri_t wifi_disconnect_json = {
                .uri = "/wifiDisconnect.json",
                .method = HTTP_DELETE,
                .handler =
http_server_wifi_disconnect_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_disconnect_json);
        // register localTime.json handler
        httpd_uri_t local_time_json = {
                .uri = "/localTime.json",
                .method = HTTP_GET,
                .handler =
http_server_get_local_time_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&local_time_json);
        return http_server_handle;
```

```
        }
        return NULL;
}
void http_server_start(void)
{
        if (http_server_handle == NULL)
        {
                http_server_handle = http_server_configure();
        }
}
void http_server_stop(void)
{
        if (http_server_handle)
        {
                httpd_stop(http_server_handle);
                ESP_LOGI(TAG, "http_server_stop: stopping HTTP
server");
                http_server_handle = NULL;
        }
        if (task_http_server_monitor)
        {
                vTaskDelete(task_http_server_monitor);
                ESP_LOGI(TAG, "http_server_stop: stopping HTTP server
monitor");
```

```
            task_http_server_monitor = NULL;
        }
}
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID)
{
        http_server_queue_message_t msg;
        msg.msgID = msgID;
        return xQueueSend(http_server_monitor_queue_handle, &msg,
portMAX_DELAY);
}
void http_server_fw_update_reset_callback(void *arg)
{
        ESP_LOGI(TAG, "http_server_fw_update_reset_callback: Timer
timed-out, restarting the device");
        esp_restart();
}
```

CHAPTER#43 DISPLAYING THE ESP32'S ACCESS POINT SSID ON THE WEBPAGE

43.1 DISPLAYING THE ESP32'S ACCESS POINT SSID ON THE WEBPAGE:

To display the ESP32's Access Point (AP) SSID on a webpage, first configure the ESP32 to run as a web server. Serve an HTML page that includes a placeholder for the SSID. In the web server's handler

function, dynamically retrieve the AP SSID using `esp_wifi_get_config()` and insert it into the HTML content before sending it to the client. This approach allows users to see the AP SSID directly on the webpage hosted by the ESP32, providing a user-friendly way to confirm network details.

CHAPTER#44 CODES OF DISPLAYING THE ESP32'S ACCESS POINT SSID ON THE WEBPAGE

44.1 CODE OF HTTP_SERVER.C:

```
/*
 * http_server.c
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include "esp_http_server.h"
#include "esp_log.h"
#include "esp_ota_ops.h"
#include "esp_wifi.h"
#include "sys/param.h"
#include "DHT22.h"
#include "http_server.h"
#include "sntp_time_sync.h"
#include "tasks_common.h"
```

```
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG[] = "http_server";
// Wifi connect status
static int g_wifi_connect_status = NONE;
// Firmware update status
static int g_fw_update_status = OTA_UPDATE_PENDING;
// Local Time status
static bool g_is_local_time_set = false;
// HTTP server task handle
static httpd_handle_t http_server_handle = NULL;
// HTTP server monitor task handle
static TaskHandle_t task_http_server_monitor = NULL;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t http_server_monitor_queue_handle;
/**
 * ESP32 timer configuration passed to esp_timer_create.
 */
const esp_timer_create_args_t fw_update_reset_args = {
        .callback = &http_server_fw_update_reset_callback,
        .arg = NULL,
        .dispatch_method = ESP_TIMER_TASK,
        .name = "fw_update_reset"
```

```
};
esp_timer_handle_t fw_update_reset;
// Embedded files: JQuery, index.html, app.css, app.js and favicon.ico
files
extern const uint8_t jquery_3_3_1_min_js_start[]
        asm("_binary_jquery_3_3_1_min_js_start");
extern const uint8_t jquery_3_3_1_min_js_end[]
        asm("_binary_jquery_3_3_1_min_js_end");
extern const uint8_t index_html_start[]
        asm("_binary_index_html_start");
extern const uint8_t index_html_end[]
        asm("_binary_index_html_end");
extern const uint8_t app_css_start[]
        asm("_binary_app_css_start");
extern const uint8_t app_css_end[]
        asm("_binary_app_css_end");
extern const uint8_t app_js_start[]
        asm("_binary_app_js_start");
extern const uint8_t app_js_end[]
        asm("_binary_app_js_end");
extern const uint8_t favicon_ico_start[]
        asm("_binary_favicon_ico_start");
extern const uint8_t favicon_ico_end[]
        asm("_binary_favicon_ico_end");
/**
 * Checks the g_fw_update_status and creates the fw_update_reset timer
if g_fw_update_status is true.
```

```
 */
static void http_server_fw_update_reset_timer(void)
{
	if (g_fw_update_status == OTA_UPDATE_SUCCESSFUL)
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
updated successful starting FW update reset timer");
		// Give the web page a chance to receive an acknowledge
back and initialize the timer

	ESP_ERROR_CHECK(esp_timer_create(&fw_update_reset_args,
&fw_update_reset));

	ESP_ERROR_CHECK(esp_timer_start_once(fw_update_reset,
8000000));
	}
	else
	{
		ESP_LOGI(TAG, "http_server_fw_update_reset_timer: FW
update unsuccessful");
	}
}
/**
 * HTTP server monitor task used to track events of the HTTP server
 * @param pvParameters parameter which can be passed to the task.
```

```
 */
static void http_server_monitor(void *parameter)
{
	http_server_queue_message_t msg;
	for (;;)
	{
		if (xQueueReceive(http_server_monitor_queue_handle,
&msg, portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case HTTP_MSG_WIFI_CONNECT_INIT:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_INIT");
					g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECTING;
					break;
				case HTTP_MSG_WIFI_CONNECT_SUCCESS:
					ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_SUCCESS");
					g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_SUCCESS;
					break;
				case HTTP_MSG_WIFI_CONNECT_FAIL:
```

```
                    ESP_LOGI(TAG,
"HTTP_MSG_WIFI_CONNECT_FAIL");
                    g_wifi_connect_status =
HTTP_WIFI_STATUS_CONNECT_FAILED;
                    break;
                case HTTP_MSG_WIFI_USER_DISCONNECT:
                    ESP_LOGI(TAG,
"HTTP_MSG_WIFI_USER_DISCONNECT");
                    g_wifi_connect_status =
HTTP_WIFI_STATUS_DISCONNECTED;
                    break;
                case
HTTP_MSG_OTA_UPDATE_SUCCESSFUL:
                    ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_SUCCESSFUL");
                    g_fw_update_status =
OTA_UPDATE_SUCCESSFUL;
                    http_server_fw_update_reset_timer();
                    break;
                case HTTP_MSG_OTA_UPDATE_FAILED:
                    ESP_LOGI(TAG,
"HTTP_MSG_OTA_UPDATE_FAILED");
                    g_fw_update_status =
OTA_UPDATE_FAILED;
                    break;
```

```
                        case
HTTP_MSG_TIME_SERVICE_INITIALIZED:
                                ESP_LOGI(TAG,
"HTTP_MSG_TIME_SERVICE_INITIALIZED");
                                g_is_local_time_set = true;
                                break;
                        default:
                                break;
                }
            }
        }
}
/**
 * Jquery get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_jquery_handler(httpd_req_t *req)
{
        ESP_LOGI(TAG, "Jquery requested");
        httpd_resp_set_type(req, "application/javascript");
        httpd_resp_send(req, (const char *)jquery_3_3_1_min_js_start,
jquery_3_3_1_min_js_end - jquery_3_3_1_min_js_start);
        return ESP_OK;
```

```
}
/**
 * Sends the index.html page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_index_html_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "index.html requested");
	httpd_resp_set_type(req, "text/html");
	httpd_resp_send(req, (const char *)index_html_start,
index_html_end - index_html_start);
	return ESP_OK;
}
/**
 * app.css get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_css_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.css requested");
	httpd_resp_set_type(req, "text/css");
```

```
	httpd_resp_send(req, (const char *)app_css_start, app_css_end -
app_css_start);
	return ESP_OK;
}
/**
 * app.js get handler is requested when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_app_js_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "app.js requested");
	httpd_resp_set_type(req, "application/javascript");
	httpd_resp_send(req, (const char *)app_js_start, app_js_end -
app_js_start);
	return ESP_OK;
}
/**
 * Sends the .ico (icon) file when accessing the web page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_favicon_ico_handler(httpd_req_t *req)
{
```

```
	ESP_LOGI(TAG, "favicon.ico requested");
	httpd_resp_set_type(req, "image/x-icon");
	httpd_resp_send(req, (const char *)favicon_ico_start,
favicon_ico_end - favicon_ico_start);
	return ESP_OK;
}
/**
 * Receives the .bin file fia the web page and handles the firmware
update
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK, otherwise ESP_FAIL if timeout occurs and the
update cannot be started.
 */
esp_err_t http_server_OTA_update_handler(httpd_req_t *req)
{
	esp_ota_handle_t ota_handle;
	char ota_buff[1024];
	int content_length = req->content_len;
	int content_received = 0;
	int recv_len;
	bool is_req_body_started = false;
	bool flash_successful = false;
	con st esp_partition_t *update_partition =
esp_ota_get_next_update_partition(NULL);
```

```
do
{
        // Read the data for the request
        if ((recv_len = httpd_req_recv(req, ota_buff,
MIN(content_length, sizeof(ota_buff)))) < 0)
        {
                // Check if timeout occurred
                if (recv_len == HTTPD_SOCK_ERR_TIMEOUT)
                {
                        ESP_LOGI(TAG,
"http_server_OTA_update_handler: Socket Timeout");
                        continue; ///> Retry receiving if timeout occurred
                }
                ESP_LOGI(TAG, "http_server_OTA_update_handler:
OTA other Error %d", recv_len);
                return ESP_FAIL;
        }
        printf("http_server_OTA_update_handler: OTA RX: %d of
%d\r", content_received, content_length);
        // Is this the first data we are receiving
        // If so, it will have the information in the header that we
need.
        if (!is_req_body_started)
        {
                is_req_body_started = true;
```

```
// Get the location of the .bin file content (remove the
web form data)
char *body_start_p = strstr(ota_buff, "\r\n\r\n") + 4;
int body_part_len = recv_len - (body_start_p -
ota_buff);
printf("http_server_OTA_update_handler: OTA file
size: %d\r\n", content_length);
esp_err_t err = esp_ota_begin(update_partition,
OTA_SIZE_UNKNOWN, &ota_handle);
if (err != ESP_OK)
{
    printf("http_server_OTA_update_handler: Error
with OTA begin, cancelling OTA\r\n");
    return ESP_FAIL;
}
else
{
    printf("http_server_OTA_update_handler: Writing
to partition subtype %d at offset 0x%x\r\n", update_partition->subtype,
update_partition->address);
}
// Write this first part of the data
esp_ota_write(ota_handle, body_start_p,
body_part_len);
content_received += body_part_len;
```

```
            }
            else
            {
                // Write OTA data
                esp_ota_write(ota_handle, ota_buff, recv_len);
                content_received += recv_len;
            }
        } while (recv_len > 0 && content_received < content_length);
        if (esp_ota_end(ota_handle) == ESP_OK)
        {
            // Lets update the partition
            if (esp_ota_set_boot_partition(update_partition) ==
ESP_OK)
            {
                const esp_partition_t *boot_partition =
esp_ota_get_boot_partition();
                ESP_LOGI(TAG, "http_server_OTA_update_handler:
Next boot partition subtype %d at offset 0x%x", boot_partition-
>subtype, boot_partition->address);
                flash_successful = true;
            }
            else
            {
                ESP_LOGI(TAG, "http_server_OTA_update_handler:
FLASHED ERROR!!!");
```

```
            }
        }
        else
        {
            ESP_LOGI(TAG, "http_server_OTA_update_handler:
esp_ota_end ERROR!!!");
        }
        // We won't update the global variables throughout the file, so send
the message about the status
        if (flash_successful) {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_SUC
CESSFUL); } else {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_FAI
LED); }
        return ESP_OK;
}
/**
 * OTA status handler responds with the firmware update status after the
OTA update is started
 * and responds with the compile time/date when the page is first
requested
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
esp_err_t http_server_OTA_status_handler(httpd_req_t *req)
```

```
{
	char otaJSON[100];
	ESP_LOGI(TAG, "OTAstatus requested");
	sprintf(otaJSON,
"{\"ota_update_status\":%d,\"compile_time\":\"%s\",\"compile_date\":\"
%s\"}", g_fw_update_status, __TIME__, __DATE__);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, otaJSON, strlen(otaJSON));
	return ESP_OK;
}
/**
 * DHT sensor readings JSON handler responds with DHT22 sensor data
 * @param req HTTP request for which the uri needs to be handled
 * @return ESP_OK
 */
static esp_err_t
http_server_get_dht_sensor_readings_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/dhtSensor.json requested");
	char dhtSensorJSON[100];
	sprintf(dhtSensorJSON,
"{\"temp\":\"%.1f\",\"humidity\":\"%.1f\"}", getTemperature(),
getHumidity());
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, dhtSensorJSON, strlen(dhtSensorJSON));
```

```
        return ESP_OK;
}
/**
 * wifiConnect.json handler is invoked after the connect button is
pressed
 * and handles receiving the SSID and password entered by the user
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_wifi_connect_json_handler(httpd_req_t
*req)
{
        ESP_LOGI(TAG, "/wifiConnect.json requested");
        size_t len_ssid = 0, len_pass = 0;
        char *ssid_str = NULL, *pass_str = NULL;
        // Get SSID header
        len_ssid = httpd_req_get_hdr_value_len(req, "my-connect-ssid") +
1;
        if (len_ssid > 1)
        {
                ssid_str = malloc(len_ssid);
                if (httpd_req_get_hdr_value_str(req, "my-connect-ssid",
ssid_str, len_ssid) == ESP_OK)
                {
```

```
                ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
ssid: %s", ssid_str);
            }
        }
        // Get Password header
        len_pass = httpd_req_get_hdr_value_len(req, "my-connect-pwd")
+ 1;
        if (len_pass > 1)
        {
            pass_str = malloc(len_pass);
            if (httpd_req_get_hdr_value_str(req, "my-connect-pwd",
pass_str, len_pass) == ESP_OK)
            {
                ESP_LOGI(TAG,
"http_server_wifi_connect_json_handler: Found header => my-connect-
pwd: %s", pass_str);
            }
        }
        // Update the Wifi networks configuration and let the wifi
application know
        wifi_config_t* wifi_config = wifi_app_get_wifi_config();
        memset(wifi_config, 0x00, sizeof(wifi_config_t));
        memcpy(wifi_config->sta.ssid, ssid_str, len_ssid);
        memcpy(wifi_config->sta.password, pass_str, len_pass);
```

```
	wifi_app_send_message(WIFI_APP_MSG_CONNECTING_FRO
M_HTTP_SERVER);
	free(ssid_str);
	free(pass_str);
	return ESP_OK;
}
/**
 * wifiConnectStatus handler updates the connection status for the web
page.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t
http_server_wifi_connect_status_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/wifiConnectStatus requested");
	char statusJSON[100];
	sprintf(statusJSON, "{\"wifi_connect_status\":%d}",
g_wifi_connect_status);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, statusJSON, strlen(statusJSON));
	return ESP_OK;
}
/**
```

```
 * wifiConnectInfo.json handler updates the web page with connection
information.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t
http_server_get_wifi_connect_info_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/wifiConnectInfo.json requested");
	char ipInfoJSON[200];
	memset(ipInfoJSON, 0, sizeof(ipInfoJSON));
	char ip[IP4ADDR_STRLEN_MAX];
	char netmask[IP4ADDR_STRLEN_MAX];
	char gw[IP4ADDR_STRLEN_MAX];
	if (g_wifi_connect_status ==
HTTP_WIFI_STATUS_CONNECT_SUCCESS)
	{
		wifi_ap_record_t wifi_data;

	ESP_ERROR_CHECK(esp_wifi_sta_get_ap_info(&wifi_data));
		char *ssid = (char*)wifi_data.ssid;
		esp_netif_ip_info_t ip_info;
		ESP_ERROR_CHECK(esp_netif_get_ip_info(esp_netif_sta,
&ip_info));
```

```
            esp_ip4addr_ntoa(&ip_info.ip, ip,
IP4ADDR_STRLEN_MAX);
            esp_ip4addr_ntoa(&ip_info.netmask, netmask,
IP4ADDR_STRLEN_MAX);
            esp_ip4addr_ntoa(&ip_info.gw, gw,
IP4ADDR_STRLEN_MAX);
            sprintf(ipInfoJSON,
"{\"ip\":\"%s\",\"netmask\":\"%s\",\"gw\":\"%s\",\"ap\":\"%s\"}", ip,
netmask, gw, ssid);
        }
        httpd_resp_set_type(req, "application/json");
        httpd_resp_send(req, ipInfoJSON, strlen(ipInfoJSON));
        return ESP_OK;
}
/**
 * wifiDisconnect.json handler responds by sending a message to the
Wifi application to disconnect.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_wifi_disconnect_json_handler(httpd_req_t
*req)
{
        ESP_LOGI(TAG, "wifiDisconect.json requested");
```

```
	wifi_app_send_message(WIFI_APP_MSG_USER_REQUESTED
_STA_DISCONNECT);
	return ESP_OK;
}
/**
 * localTime.json handler responds by sending the local time.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_get_local_time_json_handler(httpd_req_t
*req)
{
	ESP_LOGI(TAG, "/localTime.json requested");
	char localTimeJSON[100] = {0};
	if (g_is_local_time_set)
	{
		sprintf(localTimeJSON, "{\"time\":\"%s\"}",
sntp_time_sync_get_time());
	}
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, localTimeJSON, strlen(localTimeJSON));
	return ESP_OK;
}
/**
```

```
 * apSSID.json handler responds by sending the AP SSID.
 * @param req HTTP request for which the uri needs to be handled.
 * @return ESP_OK
 */
static esp_err_t http_server_get_ap_ssid_json_handler(httpd_req_t *req)
{
	ESP_LOGI(TAG, "/apSSID.json requested");
	char ssidJSON[50];
	wifi_config_t *wifi_config = wifi_app_get_wifi_config();
	esp_wifi_get_config(ESP_IF_WIFI_AP, wifi_config);
	char *ssid = (char*)wifi_config->ap.ssid;
	sprintf(ssidJSON, "{\"ssid\":\"%s\"}", ssid);
	httpd_resp_set_type(req, "application/json");
	httpd_resp_send(req, ssidJSON, strlen(ssidJSON));
	return ESP_OK;
}
/**
 * Sets up the default httpd server configuration.
 * @return http server instance handle if successful, NULL otherwise.
 */
static httpd_handle_t http_server_configure(void)
{
	// Generate the default configuration
```

```
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();
    // Create HTTP server monitor task
    xTaskCreatePinnedToCore(&http_server_monitor,
"http_server_monitor", HTTP_SERVER_MONITOR_STACK_SIZE,
NULL, HTTP_SERVER_MONITOR_PRIORITY,
&task_http_server_monitor, HTTP_SERVER_MONITOR_CORE_ID);
    // Create the message queue
    http_server_monitor_queue_handle = xQueueCreate(3,
sizeof(http_server_queue_message_t));
    // The core that the HTTP server will run on
    config.core_id = HTTP_SERVER_TASK_CORE_ID;
    // Adjust the default priority to 1 less than the wifi application task
    config.task_priority = HTTP_SERVER_TASK_PRIORITY;
    // Bump up the stack size (default is 4096)
    config.stack_size = HTTP_SERVER_TASK_STACK_SIZE;
    // Increase uri handlers
    config.max_uri_handlers = 20;
    // Increase the timeout limits
    config.recv_wait_timeout = 10;
    config.send_wait_timeout = 10;
    ESP_LOGI(TAG,
            "http_server_configure: Starting server on port: '%d'
with task priority: '%d'",
            config.server_port,
            config.task_priority);
```

```
	// Start the httpd server
	if (httpd_start(&http_server_handle, &config) == ESP_OK)
	{
		ESP_LOGI(TAG, "http_server_configure: Registering URI
handlers");
		// register query handler
		httpd_uri_t jquery_js = {
			.uri = "/jquery-3.3.1.min.js",
			.method = HTTP_GET,
			.handler = http_server_jquery_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle, &jquery_js);
		// register index.html handler
		httpd_uri_t index_html = {
			.uri = "/",
			.method = HTTP_GET,
			.handler = http_server_index_html_handler,
			.user_ctx = NULL
		};
		httpd_register_uri_handler(http_server_handle,
&index_html);
		// register app.css handler
		httpd_uri_t app_css = {
```

```
                .uri = "/app.css",
                .method = HTTP_GET,
                .handler = http_server_app_css_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_css);
        // register app.js handler
        httpd_uri_t app_js = {
                .uri = "/app.js",
                .method = HTTP_GET,
                .handler = http_server_app_js_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle, &app_js);
        // register favicon.ico handler
        httpd_uri_t favicon_ico = {
                .uri = "/favicon.ico",
                .method = HTTP_GET,
                .handler = http_server_favicon_ico_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&favicon_ico);
```

```
// register OTAupdate handler
httpd_uri_t OTA_update = {
        .uri = "/OTAupdate",
        .method = HTTP_POST,
        .handler = http_server_OTA_update_handler,
        .user_ctx = NULL
};
httpd_register_uri_handler(http_server_handle,
&OTA_update);
// register OTAstatus handler
httpd_uri_t OTA_status = {
        .uri = "/OTAstatus",
        .method = HTTP_POST,
        .handler = http_server_OTA_status_handler,
        .user_ctx = NULL
};
httpd_register_uri_handler(http_server_handle,
&OTA_status);
// register dhtSensor.json handler
httpd_uri_t dht_sensor_json = {
        .uri = "/dhtSensor.json",
        .method = HTTP_GET,
        .handler =
http_server_get_dht_sensor_readings_json_handler,
```

```
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&dht_sensor_json);
        // register wifiConnect.json handler
        httpd_uri_t wifi_connect_json = {
                .uri = "/wifiConnect.json",
                .method = HTTP_POST,
                .handler =
http_server_wifi_connect_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_connect_json);
        // register wifiConnectStatus.json handler
        httpd_uri_t wifi_connect_status_json = {
                .uri = "/wifiConnectStatus",
                .method = HTTP_POST,
                .handler =
http_server_wifi_connect_status_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_connect_status_json);
        // register wifiConnectInfo.json handler
```

```
        httpd_uri_t wifi_connect_info_json = {
                .uri = "/wifiConnectInfo.json",
                .method = HTTP_GET,
                .handler =
http_server_get_wifi_connect_info_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_connect_info_json);
        // register wifiDisconnect.json handler
        httpd_uri_t wifi_disconnect_json = {
                .uri = "/wifiDisconnect.json",
                .method = HTTP_DELETE,
                .handler =
http_server_wifi_disconnect_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&wifi_disconnect_json);
        // register localTime.json handler
        httpd_uri_t local_time_json = {
                .uri = "/localTime.json",
                .method = HTTP_GET,
                .handler =
http_server_get_local_time_json_handler,
```

```
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&local_time_json);
        // register apSSID.json handler
        httpd_uri_t ap_ssid_json = {
                .uri = "/apSSID.json",
                .method = HTTP_GET,
                .handler = http_server_get_ap_ssid_json_handler,
                .user_ctx = NULL
        };
        httpd_register_uri_handler(http_server_handle,
&ap_ssid_json);
        return http_server_handle;
    }
    return NULL;
}
void http_server_start(void)
{
    if (http_server_handle == NULL)
    {
        http_server_handle = http_server_configure();
    }
}
```

```
void http_server_stop(void)
{
	if (http_server_handle)
	{
		httpd_stop(http_server_handle);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP
server");
		http_server_handle = NULL;
	}
	if (task_http_server_monitor)
	{
		vTaskDelete(task_http_server_monitor);
		ESP_LOGI(TAG, "http_server_stop: stopping HTTP server
monitor");
		task_http_server_monitor = NULL;
	}
}
BaseType_t http_server_monitor_send_message(http_server_message_e
msgID)
{
	http_server_queue_message_t msg;
	msg.msgID = msgID;
	return xQueueSend(http_server_monitor_queue_handle, &msg,
portMAX_DELAY);
```

```
}
void http_server_fw_update_reset_callback(void *arg)
{
        ESP_LOGI(TAG, "http_server_fw_update_reset_callback: Timer timed-out, restarting the device");
        esp_restart();
}
```

44.2 CODE OF APP.CSS:

```
body {
   background-color: #f1f1f1;
   font-family: HelveticaNeueRegular, HelveticaNeue-Regular, "Helvetica Neue Regular", HelveticaNeue, "Helvetica Neue", Helvetica, Arial;
   color: #0272B7;
}
h1 {
   display: block;
   text-align: center;
   margin: 0;
   padding: 10px;
   font-size: 1.7em;
   color: #1d3557;
}
h2 {
```

```
        font-size: 1.4em;
    color: #1d3557;
}
h3 {
        font-size: 1.2em;
    color: #1d3557;
}
h4 {
        font-size: 1em;
    color: #1d3557;
}
.gr {
        color: green;
}
.rd {
        color: red;
}
#ap_ssid {
        display:inline;
        color: #1d3557;
}
#local_time {
        display:inline;
```

```
        color: #1d3557;
}
#latest_firmware, #latest_firmware_label {
        display:inline;
}
#latest_firmware {
        color: #1d3557;
}
#temperature_reading {
        display:inline;
        color: #1d3557;
}
#humidity_reading {
        display:inline;
        color: #1d3557;
}
#connected_ap_label, #connected_ap {
        display:inline;
}
#ip_address_label, #netmask_label, #gateway_label {
        display:inline;
}
#wifi_connect_ip, #wifi_connect_netmask, #wifi_connect_gw {
```

```
        display:inline;
}
#connected_ap {
        color: #1d3557;
}
#wifi_connect_ip {
        color: #1d3557;
}
#wifi_connect_netmask {
        color: #1d3557;
}
#wifi_connect_gw {
        color: #1d3557;
}
#disconnect_wifi {
        display: none;
}
.buttons {
   padding: 5px;
}
input[type="button"] {
        font-size: 12px;
        border: 0;
```

```
    line-height: 2;
    padding: 0 5px;
    text-align: center;
    color: #ffffff;
    text-shadow: 1px 1px 1px #000;
    border-radius: 10px;
    background-color: RGBA(106,125,144,0.35);
    background-image: linear-gradient(to top left,
                                rgba(0, 0, 0, .2),
                                rgba(0, 0, 0, .2) 30%,
                                rgba(0, 0, 0, 0));
    box-shadow: inset 2px 2px 3px rgba(255, 255, 255, .6),
                inset -2px -2px 3px rgba(0, 0, 0, .6);
}
input[type="button"]:hover {
    background-color: RGBA(112,164,222,0.68);
}
input[type="button"]:active {
    box-shadow: inset -2px -2px 3px rgba(255, 255, 255, .6),
                inset 2px 2px 3px rgba(0, 0, 0, .6);
}
```

44.3 CODE OF APP.JS:

```
/**
```

```
 * Add gobals here
 */
var seconds        = null;
var otaTimerVar =  null;
var wifiConnectInterval = null;
/**
 * Initialize functions here.
 */
$(document).ready(function(){
        getSSID();
        getUpdateStatus();
        startDHTSensorInterval();
        startLocalTimeInterval();
        getConnectInfo();
        $("#connect_wifi").on("click", function(){
                checkCredentials();
        });
        $("#disconnect_wifi").on("click", function(){
                disconnectWifi();
        });
});
/**
 * Gets file name and size for display on the web page.
```

```
 */
function getFileInfo()
{
   var x = document.getElementById("selected_file");
   var file = x.files[0];
   document.getElementById("file_info").innerHTML = "<h4>File: " +
file.name + "<br>" + "Size: " + file.size + " bytes</h4>";
}
/**
 * Handles the firmware update.
 */
function updateFirmware()
{
   // Form Data
   var formData = new FormData();
   var fileSelect = document.getElementById("selected_file");
   if (fileSelect.files && fileSelect.files.length == 1)
       {
       var file = fileSelect.files[0];
       formData.set("file", file, file.name);
       document.getElementById("ota_update_status").innerHTML =
"Uploading " + file.name + ", Firmware Update in Progress...";
       // Http Request
       var request = new XMLHttpRequest();
```

```
        request.upload.addEventListener("progress", updateProgress);
        request.open('POST', "/OTAupdate");
        request.responseType = "blob";
        request.send(formData);
    }
         else
         {
        window.alert('Select A File First')
    }
}
/**
 * Progress on transfers from the server to the client (downloads).
 */
function updateProgress(oEvent)
{
    if (oEvent.lengthComputable)
         {
        getUpdateStatus();
    }
         else
         {
        window.alert('total size is unknown')
    }
```

```
}
/**
 * Posts the firmware udpate status.
 */
function getUpdateStatus()
{
   var xhr = new XMLHttpRequest();
   var requestURL = "/OTAstatus";
   xhr.open('POST', requestURL, false);
   xhr.send('ota_update_status');
   if (xhr.readyState == 4 && xhr.status == 200)
       {
      var response = JSON.parse(xhr.responseText);
            document.getElementById("latest_firmware").innerHTML =
response.compile_date + " - " + response.compile_time
            // If flashing was complete it will return a 1, else -1
            // A return of 0 is just for information on the Latest Firmware
request
      if (response.ota_update_status == 1)
            {
            // Set the countdown timer time
         seconds = 10;
         // Start the countdown timer
         otaRebootTimer();
```

```
        }
        else if (response.ota_update_status == -1)
                {
            document.getElementById("ota_update_status").innerHTML =
"!!! Upload Error !!!";
        }
    }
}
/**
 * Displays the reboot countdown.
 */
function otaRebootTimer()
{
    document.getElementById("ota_update_status").innerHTML = "OTA
Firmware Update Complete. This page will close shortly, Rebooting in:
" + seconds;
    if (--seconds == 0)
         {
        clearTimeout(otaTimerVar);
        window.location.reload();
    }
         else
         {
        otaTimerVar = setTimeout(otaRebootTimer, 1000);
```

```
    }
}
/**
 * Gets DHT22 sensor temperature and humidity values for display on
the web page.
 */
function getDHTSensorValues()
{
        $.getJSON('/dhtSensor.json', function(data) {
                $("#temperature_reading").text(data["temp"]);
                $("#humidity_reading").text(data["humidity"]);
        });
}
/**
 * Sets the interval for getting the updated DHT22 sensor values.
 */
function startDHTSensorInterval()
{
        setInterval(getDHTSensorValues, 5000);
}
/**
 * Clears the connection status interval.
 */
```

```
function stopWifiConnectStatusInterval()
{
	if (wifiConnectInterval != null)
	{
		clearInterval(wifiConnectInterval);
		wifiConnectInterval = null;
	}
}
/**
 * Gets the WiFi connection status.
 */
function getWifiConnectStatus()
{
	var xhr = new XMLHttpRequest();
	var requestURL = "/wifiConnectStatus";
	xhr.open('POST', requestURL, false);
	xhr.send('wifi_connect_status');
	if (xhr.readyState == 4 && xhr.status == 200)
	{
		var response = JSON.parse(xhr.responseText);

	document.getElementById("wifi_connect_status").innerHTML =
"Connecting...";
		if (response.wifi_connect_status == 2)
```

```
            {

        document.getElementById("wifi_connect_status").innerHTML =
"<h4 class='rd'>Failed to Connect. Please check your AP credentials and
compatibility</h4>";
                stopWifiConnectStatusInterval();
            }
            else if (response.wifi_connect_status == 3)
            {

        document.getElementById("wifi_connect_status").innerHTML =
"<h4 class='gr'>Connection Success!</h4>";
                stopWifiConnectStatusInterval();
                getConnectInfo();
            }
        }
}
/**
 * Starts the interval for checking the connection status.
 */
function startWifiConnectStatusInterval()
{
        wifiConnectInterval = setInterval(getWifiConnectStatus, 2800);
}
/**
```

```
 * Connect WiFi function called using the SSID and password entered
into the text fields.
 */
function connectWifi()
{
	// Get the SSID and password
	selectedSSID = $("#connect_ssid").val();
	pwd = $("#connect_pass").val();
	$.ajax({
		url: '/wifiConnect.json',
		dataType: 'json',
		method: 'POST',
		cache: false,
		headers: {'my-connect-ssid': selectedSSID, 'my-connect-
pwd': pwd},
		data: {'timestamp': Date.now()}
	});
	startWifiConnectStatusInterval();
}
/**
 * Checks credentials on connect_wifi button click.
 */
function checkCredentials()
{
```

```
errorList = "";
credsOk = true;
selectedSSID = $("#connect_ssid").val();
pwd = $("#connect_pass").val();
if (selectedSSID == "")
{
	errorList += "<h4 class='rd'>SSID cannot be empty!</h4>";
	credsOk = false;
}
if (pwd == "")
{
	errorList += "<h4 class='rd'>Password cannot be
empty!</h4>";
	credsOk = false;
}
if (credsOk == false)
{
	$("#wifi_connect_credentials_errors").html(errorList);
}
else
{
	$("#wifi_connect_credentials_errors").html("");
	connectWifi();
```

```
        }
}
/**
 * Shows the WiFi password if the box is checked.
 */
function showPassword()
{
        var x = document.getElementById("connect_pass");
        if (x.type === "password")
        {
                x.type = "text";
        }
        else
        {
                x.type = "password";
        }
}
/**
 * Gets the connection information for displaying on the web page.
 */
function getConnectInfo()
{
        $.getJSON('/wifiConnectInfo.json', function(data)
```

```
	{
		$("#connected_ap_label").html("Connected to: ");
		$("#connected_ap").text(data["ap"]);
		$("#ip_address_label").html("IP Address: ");
		$("#wifi_connect_ip").text(data["ip"]);
		$("#netmask_label").html("Netmask: ");
		$("#wifi_connect_netmask").text(data["netmask"]);
		$("#gateway_label").html("Gateway: ");
		$("#wifi_connect_gw").text(data["gw"]);
		document.getElementById('disconnect_wifi').style.display =
'block';
	});
}
/**
 * Disconnects Wifi once the disconnect button is pressed and reloads
the web page.
 */
function disconnectWifi()
{
	$.ajax({
		url: '/wifiDisconnect.json',
		dataType: 'json',
		method: 'DELETE',
		cache: false,
```

```
            data: { 'timestamp': Date.now() }
        });
        // Update the web page
        setTimeout("location.reload(true);", 2000);
}
/**
 * Sets the interval for displaying local time.
 */
function startLocalTimeInterval()
{
        setInterval(getLocalTime, 10000);
}
/**
 * Gets the local time.
 * @note connect the ESP32 to the internet and the time will be updated.
 */
function getLocalTime()
{
        $.getJSON('/localTime.json', function(data) {
            $("#local_time").text(data["time"]);
        });
}
/**
```

```
 * Gets the ESP32's access point SSID for displaying on the web page.
 */
function getSSID()
{
	$.getJSON('/apSSID.json', function(data) {
		$("#ap_ssid").text(data["ssid"]);
	});
}
```

44.4 CODE OF INDEX.HTML:

```
<!DOCTYPE html>
<html lang="en">
	<head>
		<meta charset="utf-8"/>
		<meta name="viewport" content="width=device-width,
initial-scale=1.0, user-scalable=no">
		<meta name="apple-mobile-web-app-capable" content="yes"
/>
		<script src='jquery-3.3.1.min.js'></script>
		<link rel="stylesheet" href="app.css">
		<script async src="app.js"></script>
		<title>ESP32 Udemy Course</title>
	</head>
	<body>
```

```
<header>
    <h1>ESP32 Application Development</h1>
</header>
<div id="EspSSID">
    <h2>ESP32 SSID</h2>
    <label for="ap_ssid">Access Point SSID: </label>
    <div id="ap_ssid"></div>
</div>
<hr>
<div id="LocalTime">
    <h2>SNTP Time Synchronization</h2>
    <label for="local_time">Connect to WiFi for Local Time:
</label>
    <div id="local_time"></div>
</div>
<hr>
<div id="OTA">
<h2>ESP32 Firmware Update</h2>
    <label id="latest_firmware_label">Latest Firmware: </label>
    <div id="latest_firmware"></div>
    <input type="file" id="selected_file" accept=".bin"
style="display: none;" onchange="getFileInfo()" />
    <div class="buttons">
```

```
                <input type="button" value="Select File"
onclick="document.getElementById('selected_file').click();" />
                <input type="button" value="Update Firmware"
onclick="updateFirmware()" />
            </div>
            <h4 id="file_info"></h4>
            <h4 id="ota_update_status"></h4>
        </div>
        <hr>
        <div id="DHT22Sensor">
            <h2>DHT22 Sensor Readings</h2>
            <label for="temperature_reading">Temperature: </label>
            <div id="temperature_reading"></div>
            <label for="humidity_reading">Humidity: </label>
            <div id="humidity_reading"></div>
        </div>
        <hr>
        <div id="WiFiConnect">
            <h2>ESP32 WiFi Connect</h2>
            <section>
                <input id="connect_ssid" type="text" maxlength="32"
placeholder="SSID" value="">
                <input id="connect_pass" type="password"
maxlength="64" placeholder="Password" value="">
```

```
                <input type="checkbox"
onclick="showPassword()">Show Password
            </section>
            <div class="buttons">
                <input id="connect_wifi" type="button"
value="Connect" />
            </div>
            <div id="wifi_connect_credentials_errors"></div>
            <h4 id="wifi_connect_status"></h4>
        </div>
        <div id="ConnectInfo">
            <section>
                <div id="connected_ap_label"></div> <div
id="connected_ap"></div>
            </section>
            <div id="ip_address_label"></div> <div
id="wifi_connect_ip"></div>
            <div id="netmask_label"></div> <div
id="wifi_connect_netmask"></div>
            <div id="gateway_label"></div> <div
id="wifi_connect_gw"></div>
            <div class="buttons">
                <input id="disconnect_wifi" type="button"
value="Disconnect" />
            </div>
```

```
    </div>
    <hr>
    </body>
<html>
```

CHAPTER#45 AWS IOT IMPLEMENTATION

45.1 AWS IOT CORE RESOURCES:

45.1.1 AWS IOT:

The AWS IoT Core is a platform that connects IoT devices to AWS IoT services.AWS IoT Getting Started and Learn More About AWS IoT. AWS IoT Core includes the device gateway, which hosts the message broker, which connects and processes messages between your IoT devices and the cloud.

45.1.2 ABOUT MQTT AND AWS IOT:

MQTT: is a lightweight, publish-subscribe network protocol that transports messages between devices. AWS IoT Device Communication Protocols.

Figure 45.1.22:AWS IOT CORE RESOURCES

45.2 ACCOUNT SETUP:

Give your email address, account name, confirm it, create a password, enter your name, phone number and address, credit card information (for identity verification purposes) and verify your identity with a phone number as well.

Figure 45.2.23:AWS ACCOUNT SETUP

Choose Basic Support – Free Plan

Figure 45.5.24:BASIC SUPPORT PLAN

CHAPTER#46 AWS IOT TECHNICAL SUPPLEMENT

46.1 MQTT:

MQTT stands for MQ Telemetry Transport. It is a simple and lightweight messaging protocol, designed for resource constrained devices and low-bandwidth, high-latency or unreliable network.

46.2 MQTT IOT CORE:

MQTT is used because it requires minimal resources, and network bandwidth, while ensuring reliability and some degree of delivery assurance.This makes MQTT ideal in “machine to machine” (M2M) or

“Internet of Things”connected devices, where bandwidth and battery power are at a premium.

46.3 HOW MQTT WORKS:

MQTT uses a client/server model where every IoT device is a client and is connected to a server, called an MQTT broker (AWS IoT). The clients send messages to an address, called a topic. The MQTT broker will forward that message to all the clients subscribed to that topic.

Figure 45.3.25:WORKING OF MQTT

46.4 AWS IOT DEVICE AUTHENTICATION:

AWS IoT devices are authenticated using mutual TLS (mTLS) authentication with X.509 certificates. Once a certificate is provisioned and activated it can be installed on a device. The device will then use that certificate for all requests to AWS IoT.

Figure 46.4.26:AWS IOT DEVICE AUTHENTICATION

46.5 MUTUAL TLS AUTHENTICATION:

mTLS is used to establish trust between two parties. Each party verifies the certificate provided by the other. Certificate Authorities (CA) like Verisign are an important part of the mutual authentication.

46.6 AWS SECURITY OVERVIEW:

AWS provides the device certificate and keys (AWS is the CA). The certificates and keys are installed on the device. The device will then use that certificate and key to authenticate itself and send all requests to AWS IoT. To perform AWS IoT operations with your device, you must create an AWS IoT policy and attach it to your device certificate.

Figure 46.6.27:AWS SECURITY OVERVIEW

46.7 AWS IOT POLICIES:

AWS IoT policies are JSON documents that authorize your device for performing AWS IoT operations.AWS IoT defines a set of policy actions describing the operations and resources for which you can grant or deny access. For example:

iot:Connect represents permission to connect to the AWS IoT message broker.

iot:Subscribe represents permission to subscribe to an MQTT topic or topic filter.

CHAPTER#47 CODES OF AWS IOT

47.1 CODE OF AWS_IOT.C:

```
/*
 * aws_iot.c
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
 #include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <limits.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "esp_system.h"
#include "esp_log.h"
#include "esp_vfs_fat.h"
```

```
#include "driver/sdmmc_host.h"
#include "aws_iot.h"
#include "DHT22.h"
#include "nvs.h"
#include "nvs_flash.h"
#include "tasks_common.h"
#include "wifi_app.h"
#include "aws_iot_config.h"
#include "aws_iot_log.h"
#include "aws_iot_version.h"
#include "aws_iot_mqtt_client_interface.h"
static const char *TAG = "aws_iot";
// AWS IoT task handle
static TaskHandle_t task_aws_iot = NULL;
/**
 * CA Root certificate, device ("Thing") certificate and device ("Thing")
key.
 * "Embedded Certs" are loaded from files in "certs/" and embedded into
the app binary.
 */
extern const uint8_t aws_root_ca_pem_start[]
asm("_binary_aws_root_ca_pem_start");
extern const uint8_t certificate_pem_crt_start[]
asm("_binary_certificate_pem_crt_start");
```

```
extern const uint8_t private_pem_key_start[]
asm("_binary_private_pem_key_start");
/**
 * @brief Default MQTT HOST URL is pulled from the
aws_iot_config.h
 */
char HostAddress[255] = AWS_IOT_MQTT_HOST;
/**
 * @brief Default MQTT port is pulled from the aws_iot_config.h
 */
uint32_t port = AWS_IOT_MQTT_PORT;
void iot_subscribe_callback_handler(AWS_IoT_Client *pClient, char
*topicName, uint16_t topicNameLen,
                                IoT_Publish_Message_Params *params, void
*pData) {
    ESP_LOGI(TAG, "Subscribe callback Test: %.*s\t%.*s",
topicNameLen, topicName, (int) params->payloadLen, (char *)params-
>payload);
}
void disconnectCallbackHandler(AWS_IoT_Client *pClient, void *data)
{
    ESP_LOGW(TAG, "MQTT Disconnect");
    IoT_Error_t rc = FAILURE;
    if(NULL == pClient) {
        return;
```

```
    }
    if(aws_iot_is_autoreconnect_enabled(pClient)) {
        ESP_LOGI(TAG, "Auto Reconnect is enabled, Reconnecting
attempt will start now");
    } else {
        ESP_LOGW(TAG, "Auto Reconnect not enabled. Starting manual
reconnect...");
        rc = aws_iot_mqtt_attempt_reconnect(pClient);
        if(NETWORK_RECONNECTED == rc) {
            ESP_LOGW(TAG, "Manual Reconnect Successful");
        } else {
            ESP_LOGW(TAG, "Manual Reconnect Failed - %d", rc);
        }
    }
}
void aws_iot_task(void *param) {
    char cPayload[100];
    int32_t i = 0;
    IoT_Error_t rc = FAILURE;
    AWS_IoT_Client client;
    IoT_Client_Init_Params mqttInitParams =
iotClientInitParamsDefault;
    IoT_Client_Connect_Params connectParams =
iotClientConnectParamsDefault;
```

```
    IoT_Publish_Message_Params paramsQOS0;
    IoT_Publish_Message_Params paramsQOS1;
    ESP_LOGI(TAG, "AWS IoT SDK Version %d.%d.%d-%s",
VERSION_MAJOR, VERSION_MINOR, VERSION_PATCH,
VERSION_TAG);
    mqttInitParams.enableAutoReconnect = false; // We enable this later
below
    mqttInitParams.pHostURL = HostAddress;
    mqttInitParams.port = port;
    mqttInitParams.pRootCALocation = (const char
*)aws_root_ca_pem_start;
    mqttInitParams.pDeviceCertLocation = (const char
*)certificate_pem_crt_start;
    mqttInitParams.pDevicePrivateKeyLocation = (const char
*)private_pem_key_start;
    mqttInitParams.mqttCommandTimeout_ms = 20000;
    mqttInitParams.tlsHandshakeTimeout_ms = 5000;
    mqttInitParams.isSSLHostnameVerify = true;
    mqttInitParams.disconnectHandler = disconnectCallbackHandler;
    mqttInitParams.disconnectHandlerData = NULL;
    rc = aws_iot_mqtt_init(&client, &mqttInitParams);
    if(SUCCESS != rc) {
        ESP_LOGE(TAG, "aws_iot_mqtt_init returned error : %d ", rc);
        abort();
    }
```

```
    connectParams.keepAliveIntervalInSec = 10;
    connectParams.isCleanSession = true;
    connectParams.MQTTVersion = MQTT_3_1_1;
    /* Client ID is set in aws_iot.h and AKA your Thing's Name in AWS
IoT */
    connectParams.pClientID =
CONFIG_AWS_EXAMPLE_CLIENT_ID;
    connectParams.clientIDLen = (uint16_t)
strlen(CONFIG_AWS_EXAMPLE_CLIENT_ID);
    connectParams.isWillMsgPresent = false;
    ESP_LOGI(TAG, "Connecting to AWS...");
    do {
        rc = aws_iot_mqtt_connect(&client, &connectParams);
        if(SUCCESS != rc) {
            ESP_LOGE(TAG, "Error(%d) connecting to %s:%d", rc,
mqttInitParams.pHostURL, mqttInitParams.port);
            vTaskDelay(1000 / portTICK_RATE_MS);
        }
    } while(SUCCESS != rc);
    /*
     * Enable Auto Reconnect functionality. Minimum and Maximum
time of Exponential backoff are set in aws_iot_config.h
     *  #AWS_IOT_MQTT_MIN_RECONNECT_WAIT_INTERVAL
     *  #AWS_IOT_MQTT_MAX_RECONNECT_WAIT_INTERVAL
     */
```

```
    rc = aws_iot_mqtt_autoreconnect_set_status(&client, true);
    if(SUCCESS != rc) {
        ESP_LOGE(TAG, "Unable to set Auto Reconnect to true - %d",
rc);
        abort();
    }
    const char *TOPIC = "test_topic/esp32";
    const int TOPIC_LEN = strlen(TOPIC);
    ESP_LOGI(TAG, "Subscribing...");
    rc = aws_iot_mqtt_subscribe(&client, TOPIC, TOPIC_LEN, QOS0,
iot_subscribe_callback_handler, NULL);
    if(SUCCESS != rc) {
        ESP_LOGE(TAG, "Error subscribing : %d ", rc);
        abort();
    }
    sprintf(cPayload, "%s : %d ", "hello from SDK", i);
    paramsQOS0.qos = QOS0;
    paramsQOS0.payload = (void *) cPayload;
    paramsQOS0.isRetained = 0;
    paramsQOS1.qos = QOS1;
    paramsQOS1.payload = (void *) cPayload;
    paramsQOS1.isRetained = 0;
    while((NETWORK_ATTEMPTING_RECONNECT == rc ||
NETWORK_RECONNECTED == rc || SUCCESS == rc)) {
```

```
        //Max time the yield function will wait for read messages
        rc = aws_iot_mqtt_yield(&client, 100);
        if(NETWORK_ATTEMPTING_RECONNECT == rc) {
            // If the client is attempting to reconnect we will skip the rest of
the loop.
            continue;
        }
        ESP_LOGI(TAG, "Stack remaining for task '%s' is %d bytes",
pcTaskGetTaskName(NULL),
uxTaskGetStackHighWaterMark(NULL));
        vTaskDelay(3000 / portTICK_RATE_MS);
        sprintf(cPayload, "%s : %d ", "WiFi RSSI", wifi_app_get_rssi());
        paramsQOS0.payloadLen = strlen(cPayload);
        rc = aws_iot_mqtt_publish(&client, TOPIC, TOPIC_LEN,
&paramsQOS0);
        sprintf(cPayload, "%s : %.1f, %s : %.1f", "Temperature",
getTemperature(), "Humidity", getHumidity());
        paramsQOS1.payloadLen = strlen(cPayload);
        rc = aws_iot_mqtt_publish(&client, TOPIC, TOPIC_LEN,
&paramsQOS1);
        if (rc == MQTT_REQUEST_TIMEOUT_ERROR) {
            ESP_LOGW(TAG, "QOS1 publish ack not received.");
            rc = SUCCESS;
        }
    }
```

```
    ESP_LOGE(TAG, "An error occurred in the main loop.");
    abort();
}
void aws_iot_start(void)
{
        if (task_aws_iot == NULL)
        {
                xTaskCreatePinnedToCore(&aws_iot_task, "aws_iot_task",
AWS_IOT_TASK_STACK_SIZE, NULL,
AWS_IOT_TASK_PRIORITY, &task_aws_iot,
AWS_IOT_TASK_CORE_ID);
        }
}
```

47.2 CODE OF AWS_IOT.H:

```
/*
 * aws_iot.h
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_AWS_IOT_H_
#define MAIN_AWS_IOT_H_
#define CONFIG_AWS_EXAMPLE_CLIENT_ID
"Udemy_ESP32_Test"
```

```
/**
 * Starts AWS IoT task.
 */
void aws_iot_start(void);
#endif /* MAIN_AWS_IOT_H_ */
```

47.3 CODE OF WIFI_APP.H:

```
/*
 * wifi_app.h
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef MAIN_WIFI_APP_H_
#define MAIN_WIFI_APP_H_
#include "esp_netif.h"
// Callback typedef
typedef void (*wifi_connected_event_callback_t)(void);
// WiFi application settings
#define WIFI_AP_SSID                    "HUZAIFASHAHBAZ"
                // AP name
#define WIFI_AP_PASSWORD                "huzaf12345"
        // AP password
#define WIFI_AP_CHANNEL                 1
        // AP channel
```

```
#define WIFI_AP_SSID_HIDDEN                    0
        // AP visibility

#define WIFI_AP_MAX_CONNECTIONS          5
        // AP max clients

#define WIFI_AP_BEACON_INTERVAL            100
        // AP beacon: 100 milliseconds recommended

#define WIFI_AP_IP                              "192.168.0.1"            // AP
default IP

#define WIFI_AP_GATEWAY                        "192.168.0.1"
        // AP default Gateway (should be the same as the IP)

#define WIFI_AP_NETMASK                        "255.255.255.0"
        // AP netmask

#define WIFI_AP_BANDWIDTH                      WIFI_BW_HT20
        // AP bandwidth 20 MHz (40 MHz is the other option)

#define WIFI_STA_POWER_SAVE                    WIFI_PS_NONE
        // Power save not used

#define MAX_SSID_LENGTH                        32
        // IEEE standard maximum

#define MAX_PASSWORD_LENGTH                          64
                // IEEE standard maximum

#define MAX_CONNECTION_RETRIES           5
        // Retry number on disconnect

// netif object for the Station and Access Point

extern esp_netif_t* esp_netif_sta;

extern esp_netif_t* esp_netif_ap;

/**
```

```
 * Message IDs for the WiFi application task
 * @note Expand this based on your application requirements.
 */
typedef enum wifi_app_message
{
        WIFI_APP_MSG_START_HTTP_SERVER = 0,
        WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER,
        WIFI_APP_MSG_STA_CONNECTED_GOT_IP,
        WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT,
        WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS,
        WIFI_APP_MSG_STA_DISCONNECTED,
} wifi_app_message_e;
/**
 * Structure for the message queue
 * @note Expand this based on application requirements e.g. add another
type and parameter as required
 */
typedef struct wifi_app_queue_message
{
        wifi_app_message_e msgID;
} wifi_app_queue_message_t;
/**
 * Sends a message to the queue
```

```
 * @param msgID message ID from the wifi_app_message_e enum.
 * @return pdTRUE if an item was successfully sent to the queue, otherwise pdFALSE.
 * @note Expand the parameter list based on your requirements e.g. how you've expanded the wifi_app_queue_message_t.
 */
BaseType_t wifi_app_send_message(wifi_app_message_e msgID);
/**
 * Starts the WiFi RTOS task
 */
void wifi_app_start(void);
/**
 * Gets the wifi configuration
 */
wifi_config_t* wifi_app_get_wifi_config(void);
/**
 * Sets the callback function.
 */
void wifi_app_set_callback(wifi_connected_event_callback_t cb);
/**
 * Calls the callback function.
 */
void wifi_app_call_callback(void);
/**
```

```
 * Gets the RSSI value of the Wifi connection.
 * @return current RSSI level.
 */
int8_t wifi_app_get_rssi(void);
#endif /* MAIN_WIFI_APP_H_ */
```

47.4 CODE OF WIFI_APP.C:

```
/*
 * wifi_app.c
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "freertos/task.h"
#include "esp_err.h"
#include "esp_log.h"
#include "esp_wifi.h"
#include "lwip/netdb.h"
#include "app_nvs.h"
#include "http_server.h"
#include "rgb_led.h"
#include "tasks_common.h"
```

```
#include "wifi_app.h"
// Tag used for ESP serial console messages
static const char TAG [] = "wifi_app";
// WiFi application callback
static wifi_connected_event_callback_t wifi_connected_event_cb;
// Used for returning the WiFi configuration
wifi_config_t *wifi_config = NULL;
// Used to track the number for retries when a connection attempt fails
static int g_retry_number;
/**
 * Wifi application event group handle and status bits
 */
static EventGroupHandle_t wifi_app_event_group;
const int WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT
                = BIT0;
const int WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT
                = BIT1;
const int WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT
        = BIT2;
const int WIFI_APP_STA_CONNECTED_GOT_IP_BIT
                = BIT3;
// Queue handle used to manipulate the main queue of events
static QueueHandle_t wifi_app_queue_handle;
// netif objects for the station and access point
```

```
esp_netif_t* esp_netif_sta = NULL;
esp_netif_t* esp_netif_ap  = NULL;
/**
 * WiFi application event handler
 * @param arg data, aside from event data, that is passed to the handler
when it is called
 * @param event_base the base id of the event to register the handler for
 * @param event_id the id fo the event to register the handler for
 * @param event_data event data
 */
static void wifi_app_event_handler(void *arg, esp_event_base_t
event_base, int32_t event_id, void *event_data)
{
	if (event_base == WIFI_EVENT)
	{
		switch (event_id)
		{
			case WIFI_EVENT_AP_START:
				ESP_LOGI(TAG, "WIFI_EVENT_AP_START");
				break;
			case WIFI_EVENT_AP_STOP:
				ESP_LOGI(TAG, "WIFI_EVENT_AP_STOP");
				break;
			case WIFI_EVENT_AP_STACONNECTED:
```

```
                        ESP_LOGI(TAG,
"WIFI_EVENT_AP_STACONNECTED");
                        break;
                    case WIFI_EVENT_AP_STADISCONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_AP_STADISCONNECTED");
                        break;
                    case WIFI_EVENT_STA_START:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_START");
                        break;
                    case WIFI_EVENT_STA_CONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_CONNECTED");
                        break;
                    case WIFI_EVENT_STA_DISCONNECTED:
                        ESP_LOGI(TAG,
"WIFI_EVENT_STA_DISCONNECTED");
                        wifi_event_sta_disconnected_t
*wifi_event_sta_disconnected =
(wifi_event_sta_disconnected_t*)malloc(sizeof(wifi_event_sta_disconn
ected_t));
                        *wifi_event_sta_disconnected =
*((wifi_event_sta_disconnected_t*)event_data);
                        printf("WIFI_EVENT_STA_DISCONNECTED,
reason code %d\n", wifi_event_sta_disconnected->reason);
```

```
                        if (g_retry_number <
MAX_CONNECTION_RETRIES)
                        {
                                esp_wifi_connect();
                                g_retry_number ++;
                        }
                        else
                        {

        wifi_app_send_message(WIFI_APP_MSG_STA_DISCONNECT
ED);
                        }
                        break;
                }
        }
        else if (event_base == IP_EVENT)
        {
                switch (event_id)
                {
                        case IP_EVENT_STA_GOT_IP:
                                ESP_LOGI(TAG, "IP_EVENT_STA_GOT_IP");

        wifi_app_send_message(WIFI_APP_MSG_STA_CONNECTED_
GOT_IP);
                        break;
```

```
            }
        }
}
/**
 * Initializes the WiFi application event handler for WiFi and IP events.
 */
static void wifi_app_event_handler_init(void)
{
        // Event loop for the WiFi driver
        ESP_ERROR_CHECK(esp_event_loop_create_default());
        // Event handler for the connection
        esp_event_handler_instance_t instance_wifi_event;
        esp_event_handler_instance_t instance_ip_event;
        ESP_ERROR_CHECK(esp_event_handler_instance_register(WIF
I_EVENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_wifi_event));
        ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_E
VENT, ESP_EVENT_ANY_ID, &wifi_app_event_handler, NULL,
&instance_ip_event));
}
/**
 * Initializes the TCP stack and default WiFi configuration.
 */
static void wifi_app_default_wifi_init(void)
```

```
{
	// Initialize the TCP stack
	ESP_ERROR_CHECK(esp_netif_init());
	// Default WiFi config - operations must be in this order!
	wifi_init_config_t wifi_init_config =
WIFI_INIT_CONFIG_DEFAULT();
	ESP_ERROR_CHECK(esp_wifi_init(&wifi_init_config));
	ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_R
AM));
	esp_netif_sta = esp_netif_create_default_wifi_sta();
	esp_netif_ap = esp_netif_create_default_wifi_ap();
}
/**
 * Configures the WiFi access point settings and assigns the static IP to
the SoftAP.
 */
static void wifi_app_soft_ap_config(void)
{
	// SoftAP - WiFi access point configuration
	wifi_config_t ap_config =
	{
		.ap = {
				.ssid = WIFI_AP_SSID,
				.ssid_len = strlen(WIFI_AP_SSID),
```

```
                        .password = WIFI_AP_PASSWORD,
                        .channel = WIFI_AP_CHANNEL,
                        .ssid_hidden = WIFI_AP_SSID_HIDDEN,
                        .authmode = WIFI_AUTH_WPA2_PSK,
                        .max_connection =
WIFI_AP_MAX_CONNECTIONS,
                        .beacon_interval =
WIFI_AP_BEACON_INTERVAL,
            },
        };
        // Configure DHCP for the AP
        esp_netif_ip_info_t ap_ip_info;
        memset(&ap_ip_info, 0x00, sizeof(ap_ip_info));
        esp_netif_dhcps_stop(esp_netif_ap);                    ///>
must call this first
        inet_pton(AF_INET, WIFI_AP_IP, &ap_ip_info.ip);        ///>
Assign access point's static IP, GW, and netmask
        inet_pton(AF_INET, WIFI_AP_GATEWAY, &ap_ip_info.gw);
        inet_pton(AF_INET, WIFI_AP_NETMASK,
&ap_ip_info.netmask);
        ESP_ERROR_CHECK(esp_netif_set_ip_info(esp_netif_ap,
&ap_ip_info));              ///> Statically configure the network
interface
        ESP_ERROR_CHECK(esp_netif_dhcps_start(esp_netif_ap));
                            ///> Start the AP DHCP server (for
connecting stations e.g. your mobile device)
```

```
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_APSTA));                    ///> Setting the mode as Access Point / Station Mode
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP, &ap_config));       ///> Set our configuration
    ESP_ERROR_CHECK(esp_wifi_set_bandwidth(WIFI_IF_AP, WIFI_AP_BANDWIDTH)); ///> Our default bandwidth 20 MHz
    ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_STA_POWER_SAVE));                  ///> Power save set to "NONE"
}
/**
 * Connects the ESP32 to an external AP using the updated station configuration
 */
static void wifi_app_connect_sta(void)
{
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA, wifi_app_get_wifi_config()));
    ESP_ERROR_CHECK(esp_wifi_connect());
}
/**
 * Main task for the WiFi application
 * @param pvParameters parameter which can be passed to the task
 */
```

```
static void wifi_app_task(void *pvParameters)
{
	wifi_app_queue_message_t msg;
	EventBits_t eventBits;
	// Initialize the event handler
	wifi_app_event_handler_init();
	// Initialize the TCP/IP stack and WiFi config
	wifi_app_default_wifi_init();
	// SoftAP config
	wifi_app_soft_ap_config();
	// Start WiFi
	ESP_ERROR_CHECK(esp_wifi_start());
	// Send first event message
	wifi_app_send_message(WIFI_APP_MSG_LOAD_SAVED_CRE
DENTIALS);
	for (;;)
	{
		if (xQueueReceive(wifi_app_queue_handle, &msg,
portMAX_DELAY))
		{
			switch (msg.msgID)
			{
				case
WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS:
```

```
                ESP_LOGI(TAG,
"WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS");
                if (app_nvs_load_sta_creds())
                {
                        ESP_LOGI(TAG, "Loaded station
configuration");
                        wifi_app_connect_sta();

        xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT);
                }
                else
                {
                        ESP_LOGI(TAG, "Unable to load
station configuration");
                }
                // Next, start the web server

        wifi_app_send_message(WIFI_APP_MSG_START_HTTP_SERV
ER);
                break;
            case WIFI_APP_MSG_START_HTTP_SERVER:
                ESP_LOGI(TAG,
"WIFI_APP_MSG_START_HTTP_SERVER");
                http_server_start();
```

```
                    rgb_led_http_server_started();
                    break;
                case
WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER:
                    ESP_LOGI(TAG,
"WIFI_APP_MSG_CONNECTING_FROM_HTTP_SERVER");
                    xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
                    // Attempt a connection
                    wifi_app_connect_sta();
                    // Set current number of retries to zero
                    g_retry_number = 0;
                    // Let the HTTP server know about the
connection attempt

        http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_INIT);
                    break;
                case
WIFI_APP_MSG_STA_CONNECTED_GOT_IP:
                    ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_CONNECTED_GOT_IP");
                    xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_STA_CONNECTED_GOT_IP_BIT);
                    rgb_led_wifi_connected();
```

```
http_server_monitor_send_message(HTTP_MSG_WIFI_CONNECT_SUCCESS);
                    eventBits = xEventGroupGetBits(wifi_app_event_group);
                    if (eventBits & WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT) ///> Save STA creds only if connecting from the http server (not loaded from NVS)
                    {

xEventGroupClearBits(wifi_app_event_group, WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT); ///> Clear the bits, in case we want to disconnect and reconnect, then start again
                    }
                    else
                    {
                        app_nvs_save_sta_creds();
                    }
                    if (eventBits & WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT)
                    {

xEventGroupClearBits(wifi_app_event_group, WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
                    }
```

```
                    // Check for connection callback
                    if (wifi_connected_event_cb)
                    {
                        wifi_app_call_callback();
                    }
                    break;
                case
WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT:
                    ESP_LOGI(TAG,
"WIFI_APP_MSG_USER_REQUESTED_STA_DISCONNECT");
                    eventBits =
xEventGroupGetBits(wifi_app_event_group);
                    if (eventBits &
WIFI_APP_STA_CONNECTED_GOT_IP_BIT)
                    {

    xEventGroupSetBits(wifi_app_event_group,
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT);
                        g_retry_number =
MAX_CONNECTION_RETRIES;

    ESP_ERROR_CHECK(esp_wifi_disconnect());
                        app_nvs_clear_sta_creds();
                        rgb_led_http_server_started(); ///>
todo: rename this status LED to a name more meaningful (to your
liking)...
```

```
                }
                break;
            case WIFI_APP_MSG_STA_DISCONNECTED:
                ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED");
                eventBits =
xEventGroupGetBits(wifi_app_event_group);
                if (eventBits &
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT)
                {
                    ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT USING
SAVED CREDENTIALS");

    xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_USING_SAVED_CREDS_BIT);
                    app_nvs_clear_sta_creds();
                }
                else if (eventBits &
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT)
                {
                    ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT FROM THE
HTTP SERVER");

    xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_CONNECTING_FROM_HTTP_SERVER_BIT);
```

```
http_server_monitor_send_message(HTTP_MSG_WIFI_CONNE
CT_FAIL);
                    }
                    else if (eventBits &
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT)
                    {
                        ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: USER REQUESTED
DISCONNECTION");

xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_USER_REQUESTED_STA_DISCONNECT_BIT);

http_server_monitor_send_message(HTTP_MSG_WIFI_USER_D
ISCONNECT);
                    }
                    else
                    {
                        ESP_LOGI(TAG,
"WIFI_APP_MSG_STA_DISCONNECTED: ATTEMPT FAILED,
CHECK WIFI ACCESS POINT AVAILABILITY");
                        // Adjust this case to your needs -
maybe you want to keep trying to connect...
                    }
                    if (eventBits &
WIFI_APP_STA_CONNECTED_GOT_IP_BIT)
```

```
                        {

        xEventGroupClearBits(wifi_app_event_group,
WIFI_APP_STA_CONNECTED_GOT_IP_BIT);
                        }
                        break;
                    default:
                        break;
                }
            }
        }
}
BaseType_t wifi_app_send_message(wifi_app_message_e msgID)
{
        wifi_app_queue_message_t msg;
        msg.msgID = msgID;
        return xQueueSend(wifi_app_queue_handle, &msg,
portMAX_DELAY);
}
wifi_config_t* wifi_app_get_wifi_config(void)
{
        return wifi_config;
}
void wifi_app_set_callback(wifi_connected_event_callback_t cb)
```

```
{
	wifi_connected_event_cb = cb;
}
void wifi_app_call_callback(void)
{
	wifi_connected_event_cb();
}
int8_t wifi_app_get_rssi(void)
{
	wifi_ap_record_t wifi_data;
	ESP_ERROR_CHECK(esp_wifi_sta_get_ap_info(&wifi_data));
	return wifi_data.rssi;
}
void wifi_app_start(void)
{
	ESP_LOGI(TAG, "STARTING WIFI APPLICATION");
	// Start WiFi started LED
	rgb_led_wifi_app_started();
	// Disable default WiFi logging messages
	esp_log_level_set("wifi", ESP_LOG_NONE);
	// Allocate memory for the wifi configuration
	wifi_config = (wifi_config_t*)malloc(sizeof(wifi_config_t));
	memset(wifi_config, 0x00, sizeof(wifi_config_t));
```

```
	// Create message queue
	wifi_app_queue_handle = xQueueCreate(3,
sizeof(wifi_app_queue_message_t));
	// Create Wifi application event group
	wifi_app_event_group = xEventGroupCreate();
	// Start the WiFi application task
	xTaskCreatePinnedToCore(&wifi_app_task, "wifi_app_task",
WIFI_APP_TASK_STACK_SIZE, NULL,
WIFI_APP_TASK_PRIORITY, NULL, WIFI_APP_TASK_CORE_ID);
}
```

47.5 CODE OF DEMO_CONFIG.H:

```
/*
 * demo_config.h
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
#ifndef DEMO_CONFIG_H_
#define DEMO_CONFIG_H_
/*************************************************/
/******* DO NOT CHANGE the following order ********/
/*************************************************/
/* Logging related header files are required to be included in the
following order:
```

```
 * 1. Include the header file "logging_levels.h".
 * 2. Define LIBRARY_LOG_NAME and  LIBRARY_LOG_LEVEL.
 * 3. Include the header file "logging_stack.h".
 */
/* Include header that defines log levels. */
#include "logging_levels.h"
/* Logging configuration for the Demo. */
#ifndef LIBRARY_LOG_NAME
   #define LIBRARY_LOG_NAME     "MQTT_DEMO"
#endif
#ifndef LIBRARY_LOG_LEVEL
   #define LIBRARY_LOG_LEVEL    LOG_INFO
#endif
#include "logging_stack.h"
/************ End of logging configuration ****************/
/**
 * @brief Details of the MQTT broker to connect to.
 *
 * @note Your AWS IoT Core endpoint can be found in the AWS IoT
console under
 * Settings/Custom Endpoint, or using the describe-endpoint API.
 *
 */
```

```
#ifndef AWS_IOT_ENDPOINT
    #define AWS_IOT_ENDPOINT
CONFIG_MQTT_BROKER_ENDPOINT
#endif
/**
 * @brief AWS IoT MQTT broker port number.
 *
 * In general, port 8883 is for secured MQTT connections.
 *
 * @note Port 443 requires use of the ALPN TLS extension with the
ALPN protocol
 * name. When using port 8883, ALPN is not required.
 */
#ifndef AWS_MQTT_PORT
    #define AWS_MQTT_PORT    ( CONFIG_MQTT_BROKER_PORT
)
#endif
/**
 * @brief The username value for authenticating client to MQTT broker
when
 * username/password based client authentication is used.
 *
 * Refer to the AWS IoT documentation below for details regarding
client
 * authentication with a username and password.
```

```
 * https://docs.aws.amazon.com/iot/latest/developerguide/custom-authentication.html
 * As mentioned in the link above, an authorizer setup needs to be done to use
 * username/password based client authentication.
 *
 * @note AWS IoT message broker requires either a set of client certificate/private key
 * or username/password to authenticate the client. If this config is defined,
 * the username and password will be used instead of the client certificate and
 * private key for client authentication.
 *
 * #define CLIENT_USERNAME    "...insert here..."
 */
/**
 * @brief The password value for authenticating client to MQTT broker when
 * username/password based client authentication is used.
 *
 * Refer to the AWS IoT documentation below for details regarding client
 * authentication with a username and password.
```

```
 * https://docs.aws.amazon.com/iot/latest/developerguide/custom-authentication.html
 * As mentioned in the link above, an authorizer setup needs to be done to use
 * username/password based client authentication.
 *
 * @note AWS IoT message broker requires either a set of client certificate/private key
 * or username/password to authenticate the client.
 *
 * #define CLIENT_PASSWORD    "...insert here..."
 */
/**
 * @brief MQTT client identifier.
 *
 * No two clients may use the same client identifier simultaneously.
 */
#ifndef CLIENT_IDENTIFIER
   #define CLIENT_IDENTIFIER CONFIG_MQTT_CLIENT_IDENTIFIER
#endif
/**
 * @brief Size of the network buffer for MQTT packets.
 */
```

```
#define NETWORK_BUFFER_SIZE       (
CONFIG_MQTT_NETWORK_BUFFER_SIZE )
/**
 * @brief The name of the operating system that the application is
running on.
 * The current value is given as an example. Please update for your
specific
 * operating system.
 */
#define OS_NAME                   "FreeRTOS"
/**
 * @brief The version of the operating system that the application is
running
 * on. The current value is given as an example. Please update for your
specific
 * operating system version.
 */
#define OS_VERSION                tskKERNEL_VERSION_NUMBER
/**
 * @brief The name of the hardware platform the application is running
on. The
 * current value is given as an example. Please update for your specific
 * hardware platform.
 */
```

```
#define HARDWARE_PLATFORM_NAME
CONFIG_HARDWARE_PLATFORM_NAME
/**
 * @brief The name of the MQTT library used and its version, following
an "@"
 * symbol.
 */
#include "core_mqtt.h"
#define MQTT_LIB    "core-mqtt@" MQTT_LIBRARY_VERSION
#endif /* ifndef DEMO_CONFIG_H_ */
```

47.6 CODE OF MAIN.C:

```
/**
 * Application entry point.
 */
#include "esp_log.h"
#include "nvs_flash.h"
//#include "aws_iot.h"
#include "DHT22.h"
#include "sntp_time_sync.h"
#include "wifi_app.h"
#include "wifi_reset_button.h"
static const char TAG[] = "main";
int aws_iot_demo_main(int argc, char ** argv);
```

```
void wifi_application_connected_events(void)
{
        ESP_LOGI(TAG, "WiFi Application Connected!!");
        sntp_time_sync_task_start();
//      aws_iot_start();
        aws_iot_demo_main(0, NULL);
}
void app_main(void)
{
   // Initialize NVS
        esp_err_t ret = nvs_flash_init();
        if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND)
        {
                ESP_ERROR_CHECK(nvs_flash_erase());
                ret = nvs_flash_init();
        }
        ESP_ERROR_CHECK(ret);
        // Start Wifi
        wifi_app_start();
        // Configure Wifi reset button
        wifi_reset_button_config();
        // Start DHT22 Sensor task
```

```
        DHT22_task_start();
        // Set connected event callback
        wifi_app_set_callback(&wifi_application_connected_events);
}
```

47.7 CODE OF MQTT_DEMO_MUTUAL_AUTH.C:

```
/*
 * mqtt_demo_mutual_auth.c
 *
 *  Created on: Jul 12, 2024
 *      Author: HUZAIFASHAHBAZ
 */
/* Standard includes. */
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
/* POSIX includes. */
#include <unistd.h>
/* Include Demo Config as the first non-system header. */
#include "demo_config.h"
/* MQTT API headers. */
#include "core_mqtt.h"
#include "core_mqtt_state.h"
```

```
/* OpenSSL sockets transport implementation. */
#include "network_transport.h"
/*Include backoff algorithm header for retry logic.*/
#include "backoff_algorithm.h"
/* Clock for timer. */
#include "clock.h"
#include "DHT22.h"
#include "wifi_app.h"
/**
 * These configuration settings are required to run the mutual auth demo.
 * Throw compilation error if the below configs are not defined.
 */
#ifndef ROOT_CA_PEM
   #if CONFIG_BROKER_CERTIFICATE_OVERRIDDEN == 1
   static const char root_cert_auth_pem_start[]  = "-----BEGIN
CERTIFICATE-----\n"
CONFIG_BROKER_CERTIFICATE_OVERRIDE "\n-----END
CERTIFICATE-----";
   #else
   extern const char root_cert_auth_pem_start[]
asm("_binary_aws_root_ca_pem_start");
   #endif
//    extern const char root_cert_auth_pem_end[]
asm("_binary_aws_root_ca_pem_end");
#endif
```

```
#ifndef CLIENT_IDENTIFIER
   #error "Please define a unique client identifier,
CLIENT_IDENTIFIER, in menuconfig"
#endif
/* The AWS IoT message broker requires either a set of client
certificate/private key
 * or username/password to authenticate the client. */
#ifndef CLIENT_USERNAME
/*
 *!!! Please note democonfigCLIENT_PRIVATE_KEY_PEM in used for
 *!!! convenience of demonstration only.  Production devices should
 *!!! store keys securely, such as within a secure element.
 */
   #ifndef CLIENT_CERTIFICATE_PEM
       extern const char client_cert_pem_start[]
asm("_binary_certificate_pem_crt_start");
//       extern const char client_cert_pem_end[]
asm("_binary_certificate_pem_crt_end");
   #endif
   #ifndef CLIENT_PRIVATE_KEY_PEM
       extern const char client_key_pem_start[]
asm("_binary_private_pem_key_start");
//       extern const char client_key_pem_end[]
asm("_binary_private_pem_key_end");
   #endif
```

```
#else
/* If a username is defined, a client password also would need to be
defined for
 * client authentication. */
   #ifndef CLIENT_PASSWORD
       #error "Please define client password(CLIENT_PASSWORD) in
demo_config.h for client authentication based on username/password."
   #endif
/* AWS IoT MQTT broker port needs to be 443 for client authentication
based on
 * username/password. */
   #if AWS_MQTT_PORT != 443
       #error "Broker port, AWS_MQTT_PORT, should be defined as 443
in demo_config.h for client authentication based on
username/password."
   #endif
#endif /* ifndef CLIENT_USERNAME */
/**
 * @brief Length of MQTT server host name.
 */
#define AWS_IOT_ENDPOINT_LENGTH         ( ( uint16_t ) ( sizeof(
AWS_IOT_ENDPOINT ) - 1 ) )
/**
 * @brief Length of client identifier.
 */
```

```
#define CLIENT_IDENTIFIER_LENGTH        ( ( uint16_t ) ( sizeof( CLIENT_IDENTIFIER ) - 1 ) )
/**
 * @brief ALPN (Application-Layer Protocol Negotiation) protocol name for AWS IoT MQTT.
 *
 * This will be used if the AWS_MQTT_PORT is configured as 443 for AWS IoT MQTT broker.
 * Please see more details about the ALPN protocol for AWS IoT MQTT endpoint
 * in the link below.
 * https://aws.amazon.com/blogs/iot/mqtt-with-tls-client-authentication-on-port-443-why-it-is-useful-and-how-it-works/
 */
#define AWS_IOT_MQTT_ALPN               "x-amzn-mqtt-ca"
/**
 * @brief Length of ALPN protocol name.
 */
#define AWS_IOT_MQTT_ALPN_LENGTH        ( ( uint16_t ) ( sizeof( AWS_IOT_MQTT_ALPN ) - 1 ) )
/**
 * @brief This is the ALPN (Application-Layer Protocol Negotiation) string
 * required by AWS IoT for password-based authentication using TCP port 443.
```

```
 */
#define AWS_IOT_PASSWORD_ALPN           "mqtt"
/**
 * @brief Length of password ALPN.
 */
#define AWS_IOT_PASSWORD_ALPN_LENGTH    ( ( uint16_t ) (
sizeof( AWS_IOT_PASSWORD_ALPN ) - 1 ) )
/**
 * @brief The maximum number of retries for connecting to server.
 */
#define CONNECTION_RETRY_MAX_ATTEMPTS           ( 5U )
/**
 * @brief The maximum back-off delay (in milliseconds) for retrying
connection to server.
 */
#define CONNECTION_RETRY_MAX_BACKOFF_DELAY_MS    (
5000U )
/**
 * @brief The base back-off delay (in milliseconds) to use for
connection retry attempts.
 */
#define CONNECTION_RETRY_BACKOFF_BASE_MS        ( 500U )
/**
 * @brief Timeout for receiving CONNACK packet in milli seconds.
```

```
 */
#define CONNACK_RECV_TIMEOUT_MS                 ( 1000U )
/**
 * @brief The topic to subscribe and publish to in the example.
 *
 * The topic name starts with the client identifier to ensure that each
demo
 * interacts with a unique topic name.
 */
#define MQTT_EXAMPLE_TOPIC                  "test_topic/esp32"
/**
 * @brief Length of client MQTT topic.
 */
#define MQTT_EXAMPLE_TOPIC_LENGTH           ( ( uint16_t ) (
sizeof( MQTT_EXAMPLE_TOPIC ) - 1 ) )
/**
 * @brief The MQTT message published in this example.
 */
#define MQTT_EXAMPLE_MESSAGE                "Hello World!"
/**
 * @brief The length of the MQTT message published in this example.
 */
#define MQTT_EXAMPLE_MESSAGE_LENGTH         ( ( uint16_t ) (
sizeof( MQTT_EXAMPLE_MESSAGE ) - 1 ) )
```

```
/**
 * @brief Maximum number of outgoing publishes maintained in the application
 * until an ack is received from the broker.
 */
#define MAX_OUTGOING_PUBLISHES            ( 5U )
/**
 * @brief Invalid packet identifier for the MQTT packets. Zero is always an
 * invalid packet identifier as per MQTT 3.1.1 spec.
 */
#define MQTT_PACKET_ID_INVALID            ( ( uint16_t ) 0U )
/**
 * @brief Timeout for MQTT_ProcessLoop function in milliseconds.
 */
#define MQTT_PROCESS_LOOP_TIMEOUT_MS      ( 1500U )
/**
 * @brief The maximum time interval in seconds which is allowed to elapse
 *  between two Control Packets.
 *
 *  It is the responsibility of the Client to ensure that the interval between
 *  Control Packets being sent does not exceed the this Keep Alive value. In the
```

```
 *  absence of sending any other Control Packets, the Client MUST send
a
 *  PINGREQ Packet.
 */
#define MQTT_KEEP_ALIVE_INTERVAL_SECONDS    ( 60U )
/**
 * @brief Delay between MQTT publishes in seconds.
 */
#define DELAY_BETWEEN_PUBLISHES_SECONDS     ( 1U )
/**
 * @brief Number of PUBLISH messages sent per iteration.
 */
#define MQTT_PUBLISH_COUNT_PER_LOOP         ( 5U )
/**
 * @brief Delay in seconds between two iterations of
subscribePublishLoop().
 */
#define MQTT_SUBPUB_LOOP_DELAY_SECONDS      ( 5U )
/**
 * @brief Transport timeout in milliseconds for transport send and
receive.
 */
#define TRANSPORT_SEND_RECV_TIMEOUT_MS      ( 1500U )
/**
```

```
 * @brief The MQTT metrics string expected by AWS IoT.
 */
#define METRICS_STRING                    "?SDK=" OS_NAME
"&Version=" OS_VERSION "&Platform="
HARDWARE_PLATFORM_NAME "&MQTTLib=" MQTT_LIB
/**
 * @brief The length of the MQTT metrics string expected by AWS IoT.
 */
#define METRICS_STRING_LENGTH             ( ( uint16_t ) ( sizeof(
METRICS_STRING ) - 1 ) )
#ifdef CLIENT_USERNAME
/**
 * @brief Append the username with the metrics string if
#CLIENT_USERNAME is defined.
 *
 * This is to support both metrics reporting and username/password
based client
 * authentication by AWS IoT.
 */
   #define CLIENT_USERNAME_WITH_METRICS
CLIENT_USERNAME METRICS_STRING
#endif
/*-----------------------------------------------------------*/
/**
```

```
 * @brief Structure to keep the MQTT publish packets until an ack is
received
 * for QoS1 publishes.
 */
typedef struct PublishPackets
{
    /**
     * @brief Packet identifier of the publish packet.
     */
    uint16_t packetId;
    /**
     * @brief Publish info of the publish packet.
     */
    MQTTPublishInfo_t pubInfo;
} PublishPackets_t;
/*-----------------------------------------------------------*/
/**
 * @brief Packet Identifier generated when Subscribe request was sent
to the broker;
 * it is used to match received Subscribe ACK to the transmitted
subscribe.
 */
static uint16_t globalSubscribePacketIdentifier = 0U;
/**
```

```
 * @brief Packet Identifier generated when Unsubscribe request was
sent to the broker;
 * it is used to match received Unsubscribe ACK to the transmitted
unsubscribe
 * request.
 */
static uint16_t globalUnsubscribePacketIdentifier = 0U;
/**
 * @brief Array to keep the outgoing publish messages.
 * These stored outgoing publish messages are kept until a successful
ack
 * is received.
 */
static PublishPackets_t outgoingPublishPackets[
MAX_OUTGOING_PUBLISHES ] = { 0 };
/**
 * @brief Array to keep subscription topics.
 * Used to re-subscribe to topics that failed initial subscription attempts.
 */
static MQTTSubscribeInfo_t pGlobalSubscriptionList[ 1 ];
/**
 * @brief The network buffer must remain valid for the lifetime of the
MQTT context.
 */
static uint8_t buffer[ NETWORK_BUFFER_SIZE ];
```

```
/**
 * @brief Status of latest Subscribe ACK;
 * it is updated every time the callback function processes a Subscribe
ACK
 * and accounts for subscription to a single topic.
 */
static MQTTSubAckStatus_t globalSubAckStatus =
MQTTSubAckFailure;
/**
 * @brief Static buffer for TLS Context Semaphore.
 */
static StaticSemaphore_t xTlsContextSemaphoreBuffer;
/*-----------------------------------------------------------*/
int aws_iot_demo_main( int argc, char ** argv );
/**
 * @brief The random number generator to use for exponential backoff
with
 * jitter retry logic.
 *
 * @return The generated random number.
 */
static uint32_t generateRandomNumber();
/**
 * @brief Connect to MQTT broker with reconnection retries.
```

```
 *
 * If connection fails, retry is attempted after a timeout.
 * Timeout value will exponentially increase until maximum
 * timeout value is reached or the number of attempts are exhausted.
 *
 * @param[out] pNetworkContext The output parameter to return the
created network context.
 *
 * @return EXIT_FAILURE on failure; EXIT_SUCCESS on successful
connection.
 */
static int connectToServerWithBackoffRetries( NetworkContext_t *
pNetworkContext );
/**
 * @brief A function that connects to MQTT broker,
 * subscribes a topic, publishes to the same
 * topic MQTT_PUBLISH_COUNT_PER_LOOP number of times, and
verifies if it
 * receives the Publish message back.
 *
 * @param[in] pMqttContext MQTT context pointer.
 * @param[in,out] pClientSessionPresent Pointer to flag indicating if an
 * MQTT session is present in the client.
 *
```

```
 * @return EXIT_FAILURE on failure; EXIT_SUCCESS on success.
 */
static int subscribePublishLoop( MQTTContext_t * pMqttContext,
                                 bool * pClientSessionPresent );
/**
 * @brief The function to handle the incoming publishes.
 *
 * @param[in] pPublishInfo Pointer to publish info of the incoming publish.
 * @param[in] packetIdentifier Packet identifier of the incoming publish.
 */
static void handleIncomingPublish( MQTTPublishInfo_t * pPublishInfo,
                                   uint16_t packetIdentifier );
/**
 * @brief The application callback function for getting the incoming publish
 * and incoming acks reported from MQTT library.
 *
 * @param[in] pMqttContext MQTT context pointer.
 * @param[in] pPacketInfo Packet Info pointer for the incoming packet.
 * @param[in] pDeserializedInfo Deserialized information from the incoming packet.
 */
```

```
static void eventCallback( MQTTContext_t * pMqttContext,
                           MQTTPacketInfo_t * pPacketInfo,
                           MQTTDeserializedInfo_t * pDeserializedInfo );
/**
 * @brief Initializes the MQTT library.
 *
 * @param[in] pMqttContext MQTT context pointer.
 * @param[in] pNetworkContext The network context pointer.
 *
 * @return EXIT_SUCCESS if the MQTT library is initialized;
 * EXIT_FAILURE otherwise.
 */
static int initializeMqtt( MQTTContext_t * pMqttContext,
                           NetworkContext_t * pNetworkContext );
/**
 * @brief Sends an MQTT CONNECT packet over the already
connected TCP socket.
 *
 * @param[in] pMqttContext MQTT context pointer.
 * @param[in] createCleanSession Creates a new MQTT session if true.
 * If false, tries to establish the existing session if there was session
 * already present in broker.
 * @param[out] pSessionPresent Session was already present in the
broker or not.
```

```
 * Session present response is obtained from the CONNACK from
broker.
 *
 * @return EXIT_SUCCESS if an MQTT session is established;
 * EXIT_FAILURE otherwise.
 */
static int establishMqttSession( MQTTContext_t * pMqttContext,
                                 bool createCleanSession,
                                 bool * pSessionPresent );
/**
 * @brief Close an MQTT session by sending MQTT DISCONNECT.
 *
 * @param[in] pMqttContext MQTT context pointer.
 *
 * @return EXIT_SUCCESS if DISCONNECT was successfully sent;
 * EXIT_FAILURE otherwise.
 */
static int disconnectMqttSession( MQTTContext_t * pMqttContext );
/**
 * @brief Sends an MQTT SUBSCRIBE to subscribe to
#MQTT_EXAMPLE_TOPIC
 * defined at the top of the file.
 *
 * @param[in] pMqttContext MQTT context pointer.
```

```
 *
 * @return EXIT_SUCCESS if SUBSCRIBE was successfully sent;
 * EXIT_FAILURE otherwise.
 */
static int subscribeToTopic( MQTTContext_t * pMqttContext );
/**
 * @brief Sends an MQTT UNSUBSCRIBE to unsubscribe from
 * #MQTT_EXAMPLE_TOPIC defined at the top of the file.
 *
 * @param[in] pMqttContext MQTT context pointer.
 *
 * @return EXIT_SUCCESS if UNSUBSCRIBE was successfully sent;
 * EXIT_FAILURE otherwise.
 */
static int unsubscribeFromTopic( MQTTContext_t * pMqttContext );
/**
 * @brief Sends an MQTT PUBLISH to #MQTT_EXAMPLE_TOPIC
defined at
 * the top of the file.
 *
 * @param[in] pMqttContext MQTT context pointer.
 *
 * @return EXIT_SUCCESS if PUBLISH was successfully sent;
```

```
 * EXIT_FAILURE otherwise.
 */
static int publishToTopic( MQTTContext_t * pMqttContext );
/**
 * @brief Function to get the free index at which an outgoing publish
 * can be stored.
 *
 * @param[out] pIndex The output parameter to return the index at which an
 * outgoing publish message can be stored.
 *
 * @return EXIT_FAILURE if no more publishes can be stored;
 * EXIT_SUCCESS if an index to store the next outgoing publish is obtained.
 */
static int getNextFreeIndexForOutgoingPublishes( uint8_t * pIndex );
/**
 * @brief Function to clean up an outgoing publish at given index from the
 * #outgoingPublishPackets array.
 *
 * @param[in] index The index at which a publish message has to be cleaned up.
 */
```

```
static void cleanupOutgoingPublishAt( uint8_t index );
/**
 * @brief Function to clean up all the outgoing publishes maintained in the
 * array.
 */
static void cleanupOutgoingPublishes( void );
/**
 * @brief Function to clean up the publish packet with the given packet id.
 *
 * @param[in] packetId Packet identifier of the packet to be cleaned up from
 * the array.
 */
static void cleanupOutgoingPublishWithPacketID( uint16_t packetId );
/**
 * @brief Function to resend the publishes if a session is re-established with
 * the broker. This function handles the resending of the QoS1 publish packets,
 * which are maintained locally.
 *
 * @param[in] pMqttContext MQTT context pointer.
 */
```

```
static int handlePublishResend( MQTTContext_t * pMqttContext );
/**
 * @brief Function to update variable globalSubAckStatus with status
 * information from Subscribe ACK. Called by eventCallback after
processing
 * incoming subscribe echo.
 *
 * @param[in] Server response to the subscription request.
 */
static void updateSubAckStatus( MQTTPacketInfo_t * pPacketInfo );
/**
 * @brief Function to handle resubscription of topics on Subscribe
 * ACK failure. Uses an exponential backoff strategy with jitter.
 *
 * @param[in] pMqttContext MQTT context pointer.
 */
static int handleResubscribe( MQTTContext_t * pMqttContext );
/*-----------------------------------------------------------*/
static uint32_t generateRandomNumber()
{
    return( rand() );
}
/*-----------------------------------------------------------*/
```

```
static int connectToServerWithBackoffRetries( NetworkContext_t *
pNetworkContext )
{
    int returnStatus = EXIT_SUCCESS;
    BackoffAlgorithmStatus_t backoffAlgStatus =
BackoffAlgorithmSuccess;
    TlsTransportStatus_t tlsStatus = TLS_TRANSPORT_SUCCESS;
    BackoffAlgorithmContext_t reconnectParams;
    pNetworkContext->pcHostname = AWS_IOT_ENDPOINT;
    pNetworkContext->xPort = AWS_MQTT_PORT;
    pNetworkContext->pxTls = NULL;
    pNetworkContext->xTlsContextSemaphore =
xSemaphoreCreateMutexStatic(&xTlsContextSemaphoreBuffer);
    pNetworkContext->disableSni = 0;
    uint16_t nextRetryBackOff;
    /* Initialize credentials for establishing TLS session. */
    pNetworkContext->pcServerRootCAPem =
root_cert_auth_pem_start;
    /* If #CLIENT_USERNAME is defined, username/password is used
for authenticating
     * the client. */
#ifdef CONFIG_EXAMPLE_USE_SECURE_ELEMENT
    pNetworkContext->pcClientCertPem = NULL;
    pNetworkContext->pcClientKeyPem = NULL;
    pNetworkContext->use_secure_element = true;
```

```
#elif CONFIG_EXAMPLE_USE_DS_PERIPHERAL
    pNetworkContext->pcClientCertPem = client_cert_pem_start;
    pNetworkContext->pcClientKeyPem = NULL;
#error "Populate the ds_data structure and remove this line"
    /* pNetworkContext->ds_data = DS_DATA; */
    /* The ds_data can be populated using the API's provided by
esp_secure_cert_mgr */
#else
    #ifndef CLIENT_USERNAME
        pNetworkContext->pcClientCertPem = client_cert_pem_start;
        pNetworkContext->pcClientKeyPem = client_key_pem_start;
    #endif
#endif
    /* AWS IoT requires devices to send the Server Name Indication
(SNI)
     * extension to the Transport Layer Security (TLS) protocol and
provide
     * the complete endpoint address in the host_name field. Details about
     * SNI for AWS IoT can be found in the link below.
     * https://docs.aws.amazon.com/iot/latest/developerguide/transport-
security.html */
    if( AWS_MQTT_PORT == 443 )
    {
        /* Pass the ALPN protocol name depending on the port being used.
```

```
        * Please see more details about the ALPN protocol for the AWS
IoT MQTT
        * endpoint in the link below.
        * https://aws.amazon.com/blogs/iot/mqtt-with-tls-client-
authentication-on-port-443-why-it-is-useful-and-how-it-works/
        *
        * For username and password based authentication in AWS IoT,
        * #AWS_IOT_PASSWORD_ALPN is used. More details can be
found in the
        * link below.
        * https://docs.aws.amazon.com/iot/latest/developerguide/custom-
authentication.html
        */
        static const char * pcAlpnProtocols[] = { NULL, NULL };
        #ifdef CLIENT_USERNAME
            pcAlpnProtocols[0] = AWS_IOT_PASSWORD_ALPN;
        #else
            pcAlpnProtocols[0] = AWS_IOT_MQTT_ALPN;
        #endif
        pNetworkContext->pAlpnProtos = pcAlpnProtocols;
    } else {
        pNetworkContext->pAlpnProtos = NULL;
    }
    /* Initialize reconnect attempts and interval */
```

```
    BackoffAlgorithm_InitializeParams( &reconnectParams,

CONNECTION_RETRY_BACKOFF_BASE_MS,

CONNECTION_RETRY_MAX_BACKOFF_DELAY_MS,
                                CONNECTION_RETRY_MAX_ATTEMPTS );
    /* Attempt to connect to MQTT broker. If connection fails, retry after
     * a timeout. Timeout value will exponentially increase until
maximum
     * attempts are reached.
     */
    do
    {
        /* Establish a TLS session with the MQTT broker. This example
connects
         * to the MQTT broker as specified in AWS_IOT_ENDPOINT and
AWS_MQTT_PORT
         * at the demo config header. */
        LogInfo( ( "Establishing a TLS session to %.*s:%d.",
                   AWS_IOT_ENDPOINT_LENGTH,
                   AWS_IOT_ENDPOINT,
                   AWS_MQTT_PORT ) );
        tlsStatus = xTlsConnect ( pNetworkContext );
        if( tlsStatus != TLS_TRANSPORT_SUCCESS )
        {
```

```
        /* Generate a random number and get back-off value (in
milliseconds) for the next connection retry. */
        backoffAlgStatus = BackoffAlgorithm_GetNextBackoff(
&reconnectParams, generateRandomNumber(), &nextRetryBackOff );
        if( backoffAlgStatus == BackoffAlgorithmRetriesExhausted )
        {
            LogError( ( "Connection to the broker failed, all attempts
exhausted." ) );
            returnStatus = EXIT_FAILURE;
        }
        else if( backoffAlgStatus == BackoffAlgorithmSuccess )
        {
            LogWarn( ( "Connection to the broker failed. Retrying
connection "
                       "after %hu ms backoff.",
                       ( unsigned short ) nextRetryBackOff ) );
            Clock_SleepMs( nextRetryBackOff );
        }
    }
  } while( ( tlsStatus != TLS_TRANSPORT_SUCCESS ) && (
backoffAlgStatus == BackoffAlgorithmSuccess ) );
  return returnStatus;
}
/*-----------------------------------------------------------*/
static int getNextFreeIndexForOutgoingPublishes( uint8_t * pIndex )
```

```
{
    int returnStatus = EXIT_FAILURE;
    uint8_t index = 0;
    assert( outgoingPublishPackets != NULL );
    assert( pIndex != NULL );
    for( index = 0; index < MAX_OUTGOING_PUBLISHES; index++ )
    {
        /* A free index is marked by invalid packet id.
         * Check if the the index has a free slot. */
        if( outgoingPublishPackets[ index ].packetId ==
MQTT_PACKET_ID_INVALID )
        {
            returnStatus = EXIT_SUCCESS;
            break;
        }
    }
    /* Copy the available index into the output param. */
    *pIndex = index;
    return returnStatus;
}
/*-----------------------------------------------------------*/
static void cleanupOutgoingPublishAt( uint8_t index )
{
```

```
    assert( outgoingPublishPackets != NULL );
    assert( index < MAX_OUTGOING_PUBLISHES );
    /* Clear the outgoing publish packet. */
    ( void ) memset( &( outgoingPublishPackets[ index ] ),
                     0x00,
                     sizeof( outgoingPublishPackets[ index ] ) );
}
/*-----------------------------------------------------------*/
static void cleanupOutgoingPublishes( void )
{
    assert( outgoingPublishPackets != NULL );
    /* Clean up all the outgoing publish packets. */
    ( void ) memset( outgoingPublishPackets, 0x00, sizeof(
outgoingPublishPackets ) );
}
/*-----------------------------------------------------------*/
static void cleanupOutgoingPublishWithPacketID( uint16_t packetId )
{
    uint8_t index = 0;
    assert( outgoingPublishPackets != NULL );
    assert( packetId != MQTT_PACKET_ID_INVALID );
    /* Clean up all the saved outgoing publishes. */
    for( ; index < MAX_OUTGOING_PUBLISHES; index++ )
```

```
    {
        if( outgoingPublishPackets[ index ].packetId == packetId )
        {
            cleanupOutgoingPublishAt( index );
            LogInfo( ( "Cleaned up outgoing publish packet with packet id
%u.\n\n",
                       packetId ) );
            break;
        }
    }
}
/*-----------------------------------------------------------*/
static int handlePublishResend( MQTTContext_t * pMqttContext )
{
    int returnStatus = EXIT_SUCCESS;
    MQTTStatus_t mqttStatus = MQTTSuccess;
    uint8_t index = 0U;
    MQTTStateCursor_t cursor =
MQTT_STATE_CURSOR_INITIALIZER;
    uint16_t packetIdToResend = MQTT_PACKET_ID_INVALID;
    bool foundPacketId = false;
    assert( pMqttContext != NULL );
    assert( outgoingPublishPackets != NULL );
```

```
    /* MQTT_PublishToResend() provides a packet ID of the next
PUBLISH packet
     * that should be resent. In accordance with the MQTT v3.1.1 spec,
     * MQTT_PublishToResend() preserves the ordering of when the
original
     * PUBLISH packets were sent. The outgoingPublishPackets array is
searched
     * through for the associated packet ID. If the application requires
     * increased efficiency in the look up of the packet ID, then a hashmap
of
     * packetId key and PublishPacket_t values may be used instead. */
    packetIdToResend = MQTT_PublishToResend( pMqttContext,
&cursor );
    while( packetIdToResend != MQTT_PACKET_ID_INVALID )
    {
        foundPacketId = false;
        for( index = 0U; index < MAX_OUTGOING_PUBLISHES;
index++ )
        {
            if( outgoingPublishPackets[ index ].packetId ==
packetIdToResend )
            {
                foundPacketId = true;
                outgoingPublishPackets[ index ].pubInfo.dup = true;
                LogInfo( ( "Sending duplicate PUBLISH with packet id %u.",
```

```
                outgoingPublishPackets[ index ].packetId ) );
            mqttStatus = MQTT_Publish( pMqttContext,
                                       &outgoingPublishPackets[ index ].pubInfo,
                                       outgoingPublishPackets[ index ].packetId );
            if( mqttStatus != MQTTSuccess )
            {
                LogError( ( "Sending duplicate PUBLISH for packet id %u
"
                            " failed with status %s.",
                            outgoingPublishPackets[ index ].packetId,
                            MQTT_Status_strerror( mqttStatus ) ) );
                returnStatus = EXIT_FAILURE;
                break;
            }
            else
            {
                LogInfo( ( "Sent duplicate PUBLISH successfully for
packet id %u.\n\n",
                           outgoingPublishPackets[ index ].packetId ) );
            }
        }
    }
    if( foundPacketId == false )
    {
```

```
            LogError( ( "Packet id %u requires resend, but was not found in
"
                        "outgoingPublishPackets.",
                        packetIdToResend ) );
            returnStatus = EXIT_FAILURE;
            break;
        }
        else
        {
            /* Get the next packetID to be resent. */
            packetIdToResend = MQTT_PublishToResend( pMqttContext,
&cursor );
        }
    }
    return returnStatus;
}
/*-----------------------------------------------------------*/
static void handleIncomingPublish( MQTTPublishInfo_t * pPublishInfo,
                                   uint16_t packetIdentifier )
{
    assert( pPublishInfo != NULL );
    /* Process incoming Publish. */
    LogInfo( ( "Incoming QOS : %d.", pPublishInfo->qos ) );
```

```
    /* Verify the received publish is for the topic we have subscribed to.
*/
    if( ( pPublishInfo->topicNameLength ==
MQTT_EXAMPLE_TOPIC_LENGTH ) &&
        ( 0 == strncmp( MQTT_EXAMPLE_TOPIC,
                        pPublishInfo->pTopicName,
                        pPublishInfo->topicNameLength ) ) )
    {
        LogInfo( ( "Incoming Publish Topic Name: %.*s matches
subscribed topic.\n"
                   "Incoming Publish message Packet Id is %u.\n"
                   "Incoming Publish Message : %.*s.\n\n",
                   pPublishInfo->topicNameLength,
                   pPublishInfo->pTopicName,
                   packetIdentifier,
                   ( int ) pPublishInfo->payloadLength,
                   ( const char * ) pPublishInfo->pPayload ) );
    }
    else
    {
        LogInfo( ( "Incoming Publish Topic Name: %.*s does not match
subscribed topic.",
                   pPublishInfo->topicNameLength,
                   pPublishInfo->pTopicName ) );
```

```
    }
}
/*-----------------------------------------------------------*/
static void updateSubAckStatus( MQTTPacketInfo_t * pPacketInfo )
{
    uint8_t * pPayload = NULL;
    size_t pSize = 0;
    MQTTStatus_t mqttStatus = MQTT_GetSubAckStatusCodes(
pPacketInfo, &pPayload, &pSize );
    /* MQTT_GetSubAckStatusCodes always returns success if called
with packet info
     * from the event callback and non-NULL parameters. */
    assert( mqttStatus == MQTTSuccess );
    /* Suppress unused variable warning when asserts are disabled in
build. */
    ( void ) mqttStatus;
    /* Demo only subscribes to one topic, so only one status code is
returned. */
    globalSubAckStatus = pPayload[ 0 ];
}
/*-----------------------------------------------------------*/
static int handleResubscribe( MQTTContext_t * pMqttContext )
{
    int returnStatus = EXIT_SUCCESS;
```

```
    MQTTStatus_t mqttStatus = MQTTSuccess;
    BackoffAlgorithmStatus_t backoffAlgStatus =
BackoffAlgorithmSuccess;
    BackoffAlgorithmContext_t retryParams;
    uint16_t nextRetryBackOff = 0U;
    assert( pMqttContext != NULL );
    /* Initialize retry attempts and interval. */
    BackoffAlgorithm_InitializeParams( &retryParams,

CONNECTION_RETRY_BACKOFF_BASE_MS,

CONNECTION_RETRY_MAX_BACKOFF_DELAY_MS,
                                CONNECTION_RETRY_MAX_ATTEMPTS );
    do
    {
        /* Send SUBSCRIBE packet.
         * Note: reusing the value specified in
globalSubscribePacketIdentifier is acceptable here
         * because this function is entered only after the receipt of a
SUBACK, at which point
         * its associated packet id is free to use. */
        mqttStatus = MQTT_Subscribe( pMqttContext,
                                     pGlobalSubscriptionList,
                                     sizeof( pGlobalSubscriptionList ) / sizeof(
MQTTSubscribeInfo_t ),
```

```
                                globalSubscribePacketIdentifier );
        if( mqttStatus != MQTTSuccess )
        {
            LogError( ( "Failed to send SUBSCRIBE packet to broker with
error = %s.",
                        MQTT_Status_strerror( mqttStatus ) ) );
            returnStatus = EXIT_FAILURE;
            break;
        }
        LogInfo( ( "SUBSCRIBE sent for topic %.*s to broker.\n\n",
                   MQTT_EXAMPLE_TOPIC_LENGTH,
                   MQTT_EXAMPLE_TOPIC ) );
        /* Process incoming packet. */
        mqttStatus = MQTT_ProcessLoop( pMqttContext,
MQTT_PROCESS_LOOP_TIMEOUT_MS );
        if( mqttStatus != MQTTSuccess )
        {
            LogError( ( "MQTT_ProcessLoop returned with status = %s.",
                        MQTT_Status_strerror( mqttStatus ) ) );
            returnStatus = EXIT_FAILURE;
            break;
        }
        /* Check if recent subscription request has been rejected.
globalSubAckStatus is updated
```

```
 * in eventCallback to reflect the status of the SUBACK sent by the
broker. It represents
 * either the QoS level granted by the server upon subscription, or
acknowledgement of
 * server rejection of the subscription request. */
if( globalSubAckStatus == MQTTSubAckFailure )
{
    /* Generate a random number and get back-off value (in
milliseconds) for the next re-subscribe attempt. */
    backoffAlgStatus = BackoffAlgorithm_GetNextBackoff(
&retryParams, generateRandomNumber(), &nextRetryBackOff );
    if( backoffAlgStatus == BackoffAlgorithmRetriesExhausted )
    {
        LogError( ( "Subscription to topic failed, all attempts
exhausted." ) );
        returnStatus = EXIT_FAILURE;
    }
    else if( backoffAlgStatus == BackoffAlgorithmSuccess )
    {
        LogWarn( ( "Server rejected subscription request. Retrying "
                   "connection after %hu ms backoff.",
                   ( unsigned short ) nextRetryBackOff ) );
        Clock_SleepMs( nextRetryBackOff );
    }
}
```

```
    } while( ( globalSubAckStatus == MQTTSubAckFailure ) && (
backoffAlgStatus == BackoffAlgorithmSuccess ) );
    return returnStatus;
}
/*-----------------------------------------------------------*/
static void eventCallback( MQTTContext_t * pMqttContext,
                           MQTTPacketInfo_t * pPacketInfo,
                           MQTTDeserializedInfo_t * pDeserializedInfo )
{
    uint16_t packetIdentifier;
    assert( pMqttContext != NULL );
    assert( pPacketInfo != NULL );
    assert( pDeserializedInfo != NULL );
    /* Suppress unused parameter warning when asserts are disabled in
build. */
    ( void ) pMqttContext;
    packetIdentifier = pDeserializedInfo->packetIdentifier;
    /* Handle incoming publish. The lower 4 bits of the publish packet
     * type is used for the dup, QoS, and retain flags. Hence masking
     * out the lower bits to check if the packet is publish. */
    if( ( pPacketInfo->type & 0xF0U ) ==
MQTT_PACKET_TYPE_PUBLISH )
    {
        assert( pDeserializedInfo->pPublishInfo != NULL );
```

```
        /* Handle incoming publish. */
        handleIncomingPublish( pDeserializedInfo->pPublishInfo,
packetIdentifier );
    }
    else
    {
        /* Handle other packets. */
        switch( pPacketInfo->type )
        {
            case MQTT_PACKET_TYPE_SUBACK:
                /* A SUBACK from the broker, containing the server response
to our subscription request, has been received.
                 * It contains the status code indicating server
approval/rejection for the subscription to the single topic
                 * requested. The SUBACK will be parsed to obtain the status
code, and this status code will be stored in global
                 * variable globalSubAckStatus. */
                updateSubAckStatus( pPacketInfo );
                /* Check status of the subscription request. If
globalSubAckStatus does not indicate
                 * server refusal of the request (MQTTSubAckFailure), it
contains the QoS level granted
                 * by the server, indicating a successful subscription attempt.
*/
                if( globalSubAckStatus != MQTTSubAckFailure )
```

```
            {
                LogInfo( ( "Subscribed to the topic %.*s. with maximum
QoS %u.\n\n",
                        MQTT_EXAMPLE_TOPIC_LENGTH,
                        MQTT_EXAMPLE_TOPIC,
                        globalSubAckStatus ) );
            }
            /* Make sure ACK packet identifier matches with Request
packet identifier. */
            assert( globalSubscribePacketIdentifier == packetIdentifier );
            break;
        case MQTT_PACKET_TYPE_UNSUBACK:
            LogInfo( ( "Unsubscribed from the topic %.*s.\n\n",
                    MQTT_EXAMPLE_TOPIC_LENGTH,
                    MQTT_EXAMPLE_TOPIC ) );
            /* Make sure ACK packet identifier matches with Request
packet identifier. */
            assert( globalUnsubscribePacketIdentifier == packetIdentifier
);
            break;
        case MQTT_PACKET_TYPE_PINGRESP:
            /* Nothing to be done from application as library handles
             * PINGRESP. */
            LogWarn( ( "PINGRESP should not be handled by the
application "
```

```
                "callback when using MQTT_ProcessLoop.\n\n" ) );
            break;
        case MQTT_PACKET_TYPE_PUBACK:
            LogInfo( ( "PUBACK received for packet id %u.\n\n",
                    packetIdentifier ) );
            /* Cleanup publish packet when a PUBACK is received. */
            cleanupOutgoingPublishWithPacketID( packetIdentifier );
            break;
        /* Any other packet type is invalid. */
        default:
            LogError( ( "Unknown packet type received:(%02x).\n\n",
                     pPacketInfo->type ) );
    }
  }
}
/*-----------------------------------------------------------*/
static int establishMqttSession( MQTTContext_t * pMqttContext,
                                 bool createCleanSession,
                                 bool * pSessionPresent )
{
  int returnStatus = EXIT_SUCCESS;
  MQTTStatus_t mqttStatus;
  MQTTConnectInfo_t connectInfo = { 0 };
```

```
    assert( pMqttContext != NULL );
    assert( pSessionPresent != NULL );
    /* Establish MQTT session by sending a CONNECT packet. */
    /* If #createCleanSession is true, start with a clean session
     * i.e. direct the MQTT broker to discard any previous session data.
     * If #createCleanSession is false, directs the broker to attempt to
     * reestablish a session which was already present. */
    connectInfo.cleanSession = createCleanSession;
    /* The client identifier is used to uniquely identify this MQTT client
to
     * the MQTT broker. In a production device the identifier can be
something
     * unique, such as a device serial number. */
    connectInfo.pClientIdentifier = CLIENT_IDENTIFIER;
    connectInfo.clientIdentifierLength =
CLIENT_IDENTIFIER_LENGTH;
    /* The maximum time interval in seconds which is allowed to elapse
     * between two Control Packets.
     * It is the responsibility of the Client to ensure that the interval
between
     * Control Packets being sent does not exceed the this Keep Alive
value. In the
     * absence of sending any other Control Packets, the Client MUST
send a
     * PINGREQ Packet. */
```

```
    connectInfo.keepAliveSeconds =
MQTT_KEEP_ALIVE_INTERVAL_SECONDS;
    /* Use the username and password for authentication, if they are
defined.
     * Refer to the AWS IoT documentation below for details regarding
client
     * authentication with a username and password.
     * https://docs.aws.amazon.com/iot/latest/developerguide/custom-
authentication.html
     * An authorizer setup needs to be done, as mentioned in the above
link, to use
     * username/password based client authentication.
     *
     * The username field is populated with voluntary metrics to AWS
IoT.
     * The metrics collected by AWS IoT are the operating system, the
operating
     * system's version, the hardware platform, and the MQTT Client
library
     * information. These metrics help AWS IoT improve security and
provide
     * better technical support.
     *
     * If client authentication is based on username/password in AWS IoT,
     * the metrics string is appended to the username to support both
client
```

```
 * authentication and metrics collection. */
#ifdef CLIENT_USERNAME
    connectInfo.pUserName =
CLIENT_USERNAME_WITH_METRICS;
    connectInfo.userNameLength = strlen(
CLIENT_USERNAME_WITH_METRICS );
    connectInfo.pPassword = CLIENT_PASSWORD;
    connectInfo.passwordLength = strlen( CLIENT_PASSWORD );
#else
    connectInfo.pUserName = METRICS_STRING;
    connectInfo.userNameLength = METRICS_STRING_LENGTH;
    /* Password for authentication is not used. */
    connectInfo.pPassword = NULL;
    connectInfo.passwordLength = 0U;
#endif /* ifdef CLIENT_USERNAME */
/* Send MQTT CONNECT packet to broker. */
mqttStatus = MQTT_Connect( pMqttContext, &connectInfo, NULL,
CONNACK_RECV_TIMEOUT_MS, pSessionPresent );
if( mqttStatus != MQTTSuccess )
{
    returnStatus = EXIT_FAILURE;
    LogError( ( "Connection with MQTT broker failed with status
%s.",
            MQTT_Status_strerror( mqttStatus ) ) );
```

```
    }
    else
    {
        LogInfo( ( "MQTT connection successfully established with
broker.\n\n" ) );
    }
    return returnStatus;
}
/*-----------------------------------------------------------*/
static int disconnectMqttSession( MQTTContext_t * pMqttContext )
{
    MQTTStatus_t mqttStatus = MQTTSuccess;
    int returnStatus = EXIT_SUCCESS;
    assert( pMqttContext != NULL );
    /* Send DISCONNECT. */
    mqttStatus = MQTT_Disconnect( pMqttContext );
    if( mqttStatus != MQTTSuccess )
    {
        LogError( ( "Sending MQTT DISCONNECT failed with
status=%s.",
                    MQTT_Status_strerror( mqttStatus ) ) );
        returnStatus = EXIT_FAILURE;
    }
    return returnStatus;
```

```
}
/*-----------------------------------------------------------*/
static int subscribeToTopic( MQTTContext_t * pMqttContext )
{
    int returnStatus = EXIT_SUCCESS;
    MQTTStatus_t mqttStatus;
    assert( pMqttContext != NULL );
    /* Start with everything at 0. */
    ( void ) memset( ( void * ) pGlobalSubscriptionList, 0x00, sizeof(
pGlobalSubscriptionList ) );
    /* This example subscribes to only one topic and uses QOS1. */
    pGlobalSubscriptionList[ 0 ].qos = MQTTQoS1;
    pGlobalSubscriptionList[ 0 ].pTopicFilter =
MQTT_EXAMPLE_TOPIC;
    pGlobalSubscriptionList[ 0 ].topicFilterLength =
MQTT_EXAMPLE_TOPIC_LENGTH;
    /* Generate packet identifier for the SUBSCRIBE packet. */
    globalSubscribePacketIdentifier = MQTT_GetPacketId(
pMqttContext );
    /* Send SUBSCRIBE packet. */
    mqttStatus = MQTT_Subscribe( pMqttContext,
                                 pGlobalSubscriptionList,
                                 sizeof( pGlobalSubscriptionList ) / sizeof(
MQTTSubscribeInfo_t ),
                                 globalSubscribePacketIdentifier );
```

```
    if( mqttStatus != MQTTSuccess )
    {
        LogError( ( "Failed to send SUBSCRIBE packet to broker with
error = %s.",
                    MQTT_Status_strerror( mqttStatus ) ) );
        returnStatus = EXIT_FAILURE;
    }
    else
    {
        LogInfo( ( "SUBSCRIBE sent for topic %.*s to broker.\n\n",
                   MQTT_EXAMPLE_TOPIC_LENGTH,
                   MQTT_EXAMPLE_TOPIC ) );
    }
    return returnStatus;
}
/*-----------------------------------------------------------*/
static int unsubscribeFromTopic( MQTTContext_t * pMqttContext )
{
    int returnStatus = EXIT_SUCCESS;
    MQTTStatus_t mqttStatus;
    assert( pMqttContext != NULL );
    /* Start with everything at 0. */
    ( void ) memset( ( void * ) pGlobalSubscriptionList, 0x00, sizeof(
pGlobalSubscriptionList ) );
```

```
    /* This example subscribes to and unsubscribes from only one topic
     * and uses QOS1. */
    pGlobalSubscriptionList[ 0 ].qos = MQTTQoS1;
    pGlobalSubscriptionList[ 0 ].pTopicFilter =
MQTT_EXAMPLE_TOPIC;
    pGlobalSubscriptionList[ 0 ].topicFilterLength =
MQTT_EXAMPLE_TOPIC_LENGTH;
    /* Generate packet identifier for the UNSUBSCRIBE packet. */
    globalUnsubscribePacketIdentifier = MQTT_GetPacketId(
pMqttContext );
    /* Send UNSUBSCRIBE packet. */
    mqttStatus = MQTT_Unsubscribe( pMqttContext,
                                   pGlobalSubscriptionList,
                                   sizeof( pGlobalSubscriptionList ) / sizeof(
MQTTSubscribeInfo_t ),
                                   globalUnsubscribePacketIdentifier );
    if( mqttStatus != MQTTSuccess )
    {
        LogError( ( "Failed to send UNSUBSCRIBE packet to broker with
error = %s.",
                    MQTT_Status_strerror( mqttStatus ) ) );
        returnStatus = EXIT_FAILURE;
    }
    else
    {
```

```
        LogInfo( ( "UNSUBSCRIBE sent for topic %.*s to broker.\n\n",
                   MQTT_EXAMPLE_TOPIC_LENGTH,
                   MQTT_EXAMPLE_TOPIC ) );
    }
    return returnStatus;
}
/*-----------------------------------------------------------*/
static int publishToTopic( MQTTContext_t * pMqttContext )
{
        char cPayload[100];
        sprintf(cPayload, "%s : %d, %s : %.1f, %s : %.1f", "WiFi RSSI",
wifi_app_get_rssi(), "Temperature", getTemperature(), "Humidity",
getHumidity());
    int returnStatus = EXIT_SUCCESS;
    MQTTStatus_t mqttStatus = MQTTSuccess;
    uint8_t publishIndex = MAX_OUTGOING_PUBLISHES;
    assert( pMqttContext != NULL );
    /* Get the next free index for the outgoing publish. All QoS1 outgoing
     * publishes are stored until a PUBACK is received. These messages
are
     * stored for supporting a resend if a network connection is broken
before
     * receiving a PUBACK. */
    returnStatus = getNextFreeIndexForOutgoingPublishes(
&publishIndex );
```

```
    if( returnStatus == EXIT_FAILURE )
    {
        LogError( ( "Unable to find a free spot for outgoing PUBLISH
message.\n\n" ) );
    }
    else
    {
        /* This example publishes to only one topic and uses QOS1. */
        outgoingPublishPackets[ publishIndex ].pubInfo.qos =
MQTTQoS1;
        outgoingPublishPackets[ publishIndex ].pubInfo.pTopicName =
MQTT_EXAMPLE_TOPIC;
        outgoingPublishPackets[ publishIndex ].pubInfo.topicNameLength
= MQTT_EXAMPLE_TOPIC_LENGTH;
        outgoingPublishPackets[ publishIndex ].pubInfo.pPayload =
cPayload;
        outgoingPublishPackets[ publishIndex ].pubInfo.payloadLength =
strlen(cPayload);
        /* Get a new packet id. */
        outgoingPublishPackets[ publishIndex ].packetId =
MQTT_GetPacketId( pMqttContext );
        /* Send PUBLISH packet. */
        mqttStatus = MQTT_Publish( pMqttContext,
                                   &outgoingPublishPackets[ publishIndex ].pubInfo,
```

```
                        outgoingPublishPackets[ publishIndex ].packetId
);
        if( mqttStatus != MQTTSuccess )
        {
            LogError( ( "Failed to send PUBLISH packet to broker with
error = %s.",
                        MQTT_Status_strerror( mqttStatus ) ) );
            cleanupOutgoingPublishAt( publishIndex );
            returnStatus = EXIT_FAILURE;
        }
        else
        {
            LogInfo( ( "PUBLISH sent for topic %.*s to broker with packet
ID %u.\n\n",
                       MQTT_EXAMPLE_TOPIC_LENGTH,
                       MQTT_EXAMPLE_TOPIC,
                       outgoingPublishPackets[ publishIndex ].packetId ) );
        }
    }
    return returnStatus;
}
/*-----------------------------------------------------------*/
static int initializeMqtt( MQTTContext_t * pMqttContext,
                           NetworkContext_t * pNetworkContext )
```

```
{
    int returnStatus = EXIT_SUCCESS;
    MQTTStatus_t mqttStatus;
    MQTTFixedBuffer_t networkBuffer;
    TransportInterface_t transport;
    assert( pMqttContext != NULL );
    assert( pNetworkContext != NULL );
    /* Fill in TransportInterface send and receive function pointers.
     * For this demo, TCP sockets are used to send and receive data
     * from network. Network context is SSL context for OpenSSL.*/
    transport.pNetworkContext = pNetworkContext;
    transport.send = espTlsTransportSend;
    transport.recv = espTlsTransportRecv;
    /* Fill the values for network buffer. */
    networkBuffer.pBuffer = buffer;
    networkBuffer.size = NETWORK_BUFFER_SIZE;
    /* Initialize MQTT library. */
    mqttStatus = MQTT_Init( pMqttContext,
                            &transport,
                            Clock_GetTimeMs,
                            eventCallback,
                            &networkBuffer );
    if( mqttStatus != MQTTSuccess )
```

```
    {
        returnStatus = EXIT_FAILURE;
        LogError( ( "MQTT init failed: Status = %s.",
MQTT_Status_strerror( mqttStatus ) ) );
    }
    return returnStatus;
}
/*-----------------------------------------------------------*/
static int subscribePublishLoop( MQTTContext_t * pMqttContext,
                                 bool * pClientSessionPresent )
{
    int returnStatus = EXIT_SUCCESS;
    bool mqttSessionEstablished = false, brokerSessionPresent;
    MQTTStatus_t mqttStatus = MQTTSuccess;
    uint32_t publishCount = 0;
    const uint32_t maxPublishCount =
MQTT_PUBLISH_COUNT_PER_LOOP;
    bool createCleanSession = false;
    assert( pMqttContext != NULL );
    assert( pClientSessionPresent != NULL );
    /* A clean MQTT session needs to be created, if there is no session
saved
     * in this MQTT client. */
    createCleanSession = ( *pClientSessionPresent == true ) ? false : true;
```

```
    /* Establish MQTT session on top of TCP+TLS connection. */
    LogInfo( ( "Creating an MQTT connection to %.*s.",
               AWS_IOT_ENDPOINT_LENGTH,
               AWS_IOT_ENDPOINT ) );
    /* Sends an MQTT Connect packet using the established TLS session,
     * then waits for connection acknowledgment (CONNACK) packet. */
    returnStatus = establishMqttSession( pMqttContext,
createCleanSession, &brokerSessionPresent );
    if( returnStatus == EXIT_SUCCESS )
    {
        /* Keep a flag for indicating if MQTT session is established. This
         * flag will mark that an MQTT DISCONNECT has to be sent at
the end
         * of the demo, even if there are intermediate failures. */
        mqttSessionEstablished = true;
        /* Update the flag to indicate that an MQTT client session is saved.
         * Once this flag is set, MQTT connect in the following iterations
of
         * this demo will be attempted without requesting for a clean
session. */
        *pClientSessionPresent = true;
        /* Check if session is present and if there are any outgoing
publishes
         * that need to resend. This is only valid if the broker is
         * re-establishing a session which was already present. */
```

```
        if( brokerSessionPresent == true )
        {
            LogInfo( ( "An MQTT session with broker is re-established. "
                       "Resending unacked publishes." ) );
            /* Handle all the resend of publish messages. */
            returnStatus = handlePublishResend( pMqttContext );
        }
        else
        {
            LogInfo( ( "A clean MQTT connection is established."
                       " Cleaning up all the stored outgoing publishes.\n\n" ) );
            /* Clean up the outgoing publishes waiting for ack as this new
             * connection doesn't re-establish an existing session. */
            cleanupOutgoingPublishes();
        }
    }
    if( returnStatus == EXIT_SUCCESS )
    {
        /* The client is now connected to the broker. Subscribe to the topic
         * as specified in MQTT_EXAMPLE_TOPIC at the top of this file by sending a
         * subscribe packet. This client will then publish to the same topic it
         * subscribed to, so it will expect all the messages it sends to the broker
```

```
        * to be sent back to it from the broker. This demo uses QOS1 in
Subscribe,
        * therefore, the Publish messages received from the broker will
have QOS1. */
        LogInfo( ( "Subscribing to the MQTT topic %.*s.",
                MQTT_EXAMPLE_TOPIC_LENGTH,
                MQTT_EXAMPLE_TOPIC ) );
        returnStatus = subscribeToTopic( pMqttContext );
    }
    if( returnStatus == EXIT_SUCCESS )
    {
        /* Process incoming packet from the broker. Acknowledgment for
subscription
        * ( SUBACK ) will be received here. However after sending the
subscribe, the
        * client may receive a publish before it receives a subscribe ack.
Since this
        * demo is subscribing to the topic to which no one is publishing,
probability
        * of receiving publish message before subscribe ack is zero; but
application
        * must be ready to receive any packet. This demo uses
MQTT_ProcessLoop to
        * receive packet from network. */
        mqttStatus = MQTT_ProcessLoop( pMqttContext,
MQTT_PROCESS_LOOP_TIMEOUT_MS );
```

```
        if( mqttStatus != MQTTSuccess )
        {
            returnStatus = EXIT_FAILURE;
            LogError( ( "MQTT_ProcessLoop returned with status = %s.",
                        MQTT_Status_strerror( mqttStatus ) ) );
        }
    }
    /* Check if recent subscription request has been rejected. globalSubAckStatus is updated
     * in eventCallback to reflect the status of the SUBACK sent by the broker. */
    if( ( returnStatus == EXIT_SUCCESS ) && ( globalSubAckStatus == MQTTSubAckFailure ) )
    {
        /* If server rejected the subscription request, attempt to resubscribe to topic.
         * Attempts are made according to the exponential backoff retry strategy
         * implemented in retryUtils. */
        LogInfo( ( "Server rejected initial subscription request. Attempting to re-subscribe to topic %.*s.",
                   MQTT_EXAMPLE_TOPIC_LENGTH,
                   MQTT_EXAMPLE_TOPIC ) );
        returnStatus = handleResubscribe( pMqttContext );
    }
```

```
    if( returnStatus == EXIT_SUCCESS )
    {
        /* Publish messages with QOS1, receive incoming messages and
         * send keep alive messages. */
        for( publishCount = 0; publishCount < maxPublishCount;
publishCount++ )
        {
            LogInfo( ( "Sending Publish to the MQTT topic %.*s.",
                       MQTT_EXAMPLE_TOPIC_LENGTH,
                       MQTT_EXAMPLE_TOPIC ) );
            returnStatus = publishToTopic( pMqttContext );
            /* Calling MQTT_ProcessLoop to process incoming publish
echo, since
             * application subscribed to the same topic the broker will send
             * publish message back to the application. This function also
             * sends ping request to broker if
MQTT_KEEP_ALIVE_INTERVAL_SECONDS
             * has expired since the last MQTT packet sent and receive
             * ping responses. */
            mqttStatus = MQTT_ProcessLoop( pMqttContext,
MQTT_PROCESS_LOOP_TIMEOUT_MS );
            /* For any error in #MQTT_ProcessLoop, exit the loop and
disconnect
             * from the broker. */
            if( mqttStatus != MQTTSuccess )
```

```
        {
            LogError( ( "MQTT_ProcessLoop returned with status = %s.",
                        MQTT_Status_strerror( mqttStatus ) ) );
            returnStatus = EXIT_FAILURE;
            break;
        }
        LogInfo( ( "Delay before continuing to next iteration.\n\n" ) );
        /* Leave connection idle for some time. */
        sleep( DELAY_BETWEEN_PUBLISHES_SECONDS );
    }
}
if( returnStatus == EXIT_SUCCESS )
{
    /* Unsubscribe from the topic. */
    LogInfo( ( "Unsubscribing from the MQTT topic %.*s.",
               MQTT_EXAMPLE_TOPIC_LENGTH,
               MQTT_EXAMPLE_TOPIC ) );
    returnStatus = unsubscribeFromTopic( pMqttContext );
}
if( returnStatus == EXIT_SUCCESS )
{
    /* Process Incoming UNSUBACK packet from the broker. */
```

```
        mqttStatus = MQTT_ProcessLoop( pMqttContext,
MQTT_PROCESS_LOOP_TIMEOUT_MS );
        if( mqttStatus != MQTTSuccess )
        {
            returnStatus = EXIT_FAILURE;
            LogError( ( "MQTT_ProcessLoop returned with status = %s.",
                        MQTT_Status_strerror( mqttStatus ) ) );
        }
    }
    /* Send an MQTT Disconnect packet over the already connected TCP
socket.
     * There is no corresponding response for the disconnect packet. After
sending
     * disconnect, client must close the network connection. */
    if( mqttSessionEstablished == true )
    {
        LogInfo( ( "Disconnecting the MQTT connection with %.*s.",
                   AWS_IOT_ENDPOINT_LENGTH,
                   AWS_IOT_ENDPOINT ) );
        if( returnStatus == EXIT_FAILURE )
        {
            /* Returned status is not used to update the local status as there
             * were failures in demo execution. */
            ( void ) disconnectMqttSession( pMqttContext );
```

```
        }
        else
        {
            returnStatus = disconnectMqttSession( pMqttContext );
        }
    }
    /* Reset global SUBACK status variable after completion of
subscription request cycle. */
    globalSubAckStatus = MQTTSubAckFailure;
    return returnStatus;
}
/*-----------------------------------------------------------*/
/**
 * @brief Entry point of demo.
 *
 * The example shown below uses MQTT APIs to send and receive
MQTT packets
 * over the TLS connection established using OpenSSL.
 *
 * The example is single threaded and uses statically allocated memory;
 * it uses QOS1 and therefore implements a retransmission mechanism
 * for Publish messages. Retransmission of publish messages are
attempted
```

```
 * when a MQTT connection is established with a session that was already
 * present. All the outgoing publish messages waiting to receive PUBACK
 * are resent in this demo. In order to support retransmission all the outgoing
 * publishes are stored until a PUBACK is received.
 */
int aws_iot_demo_main( int argc,
        char ** argv )
{
    int returnStatus = EXIT_SUCCESS;
    MQTTContext_t mqttContext = { 0 };
    NetworkContext_t xNetworkContext = { 0 };
    bool clientSessionPresent = false;
    struct timespec tp;
    ( void ) argc;
    ( void ) argv;
    /* Seed pseudo random number generator (provided by ISO C standard library) for
     * use by retry utils library when retrying failed network operations. */
    /* Get current time to seed pseudo random number generator. */
    ( void ) clock_gettime( CLOCK_REALTIME, &tp );
    /* Seed pseudo random number generator with nanoseconds. */
```

```
    srand( tp.tv_nsec );
    /* Initialize MQTT library. Initialization of the MQTT library needs to be
     * done only once in this demo. */
    returnStatus = initializeMqtt( &mqttContext, &xNetworkContext );
    if( returnStatus == EXIT_SUCCESS )
    {
        for( ; ; )
        {
            /* Attempt to connect to the MQTT broker. If connection fails, retry after
             * a timeout. Timeout value will be exponentially increased till the maximum
             * attempts are reached or maximum timeout value is reached. The function
             * returns EXIT_FAILURE if the TCP connection cannot be established to
             * broker after configured number of attempts. */
            returnStatus = connectToServerWithBackoffRetries( &xNetworkContext );
            if( returnStatus == EXIT_FAILURE )
            {
                /* Log error to indicate connection failure after all
                 * reconnect attempts are over. */
                LogError( ( "Failed to connect to MQTT broker %.*s.",
```

```
                    AWS_IOT_ENDPOINT_LENGTH,
                    AWS_IOT_ENDPOINT ) );
        }
        else
        {
            /* If TLS session is established, execute Subscribe/Publish
loop. */
            returnStatus = subscribePublishLoop( &mqttContext,
&clientSessionPresent );
        }
        if( returnStatus == EXIT_SUCCESS )
        {
            /* Log message indicating an iteration completed successfully.
*/
            LogInfo( ( "Demo completed successfully." ) );
        }
        /* End TLS session, then close TCP connection. */
        ( void ) xTlsDisconnect( &xNetworkContext );
        LogInfo( ( "Short delay before starting the next iteration....\n" ) );
        sleep( MQTT_SUBPUB_LOOP_DELAY_SECONDS );
    }
  }
  return returnStatus;
}
```

/*----------*/

CHAPTER#48 CONCLUSION

48.1 CONCLUSION:

Developing IoT applications with the ESP32 using the ESP-IDF provides a powerful platform for creating scalable solutions. Setting up the development environment with ESP-IDF and CMake is essential. Integrating components and sensors like the DHT22 extends functionality, while implementing RGB status LEDs enhances user interaction. WiFi capabilities allow network connectivity, with HTTP servers enabling web-based interfaces. OTA updates ensure remote firmware deployment, and non-volatile storage retains critical data. Handling button interrupts, SNTP time synchronization, and displaying network info on web pages improve usability. Finally, AWS IoT integration enables robust cloud-based interactions, making the ESP32 a versatile choice for IoT development.

REFERENCES:

1: GIT HUB

2: AWS IOT AMAZON

3:ESPRESSIF-IDF

4:ECLIPSE IDE