A Major Project Report on

BUILDING A WIFI BASED IOT APPLICATION USING ESP32 AND AWS IoT

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted By

K. HARISH **20915A0417**

Under The Esteemed Guidance of,

Dr. M . RAJENDRA PRASAD

Professor & HOD

Department of Electronics and Communication Engineering

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(An Autonomous Institute)

Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)
(Aziz Nagar, C.B.Post, Hyderabad -500075)

2022 - 2023

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION

[Accredited by NAAC & NBA, Approved by AICTE New Delhi & Permanently Affiliated to JNTUH]

(Aziz Nagar, C.B.Post, Hyderabad -500075)

Department of Electronics and Communication Engineering

CERTIFICATE

This is to certify that the project report titled **“BUILDING A WIFI BASED IoT APPLICATION USING ESP32 AND AWS IoT”** is being submitted by **K.HARISH (20915A0417)** IV B.Tech II Semester of Electronics & Communication Engineering is a record Bonafide work carried out by them. The results embodied in this report have not been submitted to any other University for the award of any degree.

INTERNAL GUIDE,	**HEAD OF THE DEPARTMENT,**
Dr. M. RAJENDRA PRASAD	**Dr. M. RAJENDRA PRASAD**
Professor & HOD	**Professor & HOD**
Department of ECE	**Department of ECE**

EXTERNAL EXAMINER

DECLARATION

This is to certify that the work reported in the present project **"BUILDING A WIFI BASED IoT APPLICATION USING ESP32 AND AWS IoT"** is a record of work done by us in the Department of Electronics and Communication Engineering, Vidya Jyothi Institute of Technology, Jawaharlal Nehru Technological University, Hyderabad. The reports are based on the project work done entirely by us and not copied from any other source.

PROJECT ASSOCIATE:

K. HARISH 20915A0417

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to **Dr. M. RAJENDRA PRASAD, Professor ECE Dept**, Project guide who has guided and supported us through every stage in the project.

I am really grateful to **Mr. S. PRADEEP KUMAR REDDY,** Project Coordinator for his time to time, much needed valuable guidance throughout my study.

I am really grateful to **Dr. M. RAJENDRA PRASAD, Professor & HOD ECE Dept**, Vidya Jyothi Institute of Technology for his time to time, much needed valuable guidance throughout my study.

I express my hearted gratitude to **Dr. A. PADMAJA** Principal Vidya Jyothi Institute of Technology for giving us spontaneous encourage for completing the project.

I would like to thank **Dr. E. SAIBABA REDDY**, Director of Vidya Jyothi Institute of Technology for encouraging us in the completion of our project.

It is my privilege to express our gratitude and indebtedness to **Dr. PALLA RAJESHWAR REDDY, Secretary& Correspondent** of Vidya Jyothi Institute of Technology for his moral support.

I express our heartful thanks to **Staff Of Electronics and Communication Department,** Vidya Jyothi Institute of Technology for helping us in carrying out our project successfully.

PROJECT ASSOCIATE:

K. HARISH 20915A0417

ABSTRACT

This project aims to develop a WiFi based Internet of Things(IoT) application using the ESP32 microcontroller and Amazon Web Services(AWS) IoT platform. The ESP32 a low-cost, low power microcontroller with integrated WiFi and Bluetooth capabilities, will be utilized as the device for collecting and transmitting sensor data. The collected data will then be sent to AWS IoT platform for storage and further analysis. The project will also involve using the MQTT protocol for communication between the ESP32 and the AWS IoT platform. For programming the ESP32 , we use ESP-IDF which is a IoT development framework. By utilizing FreeRTOS for task management and intertask communication. Here for creating the webpage we use .HTML, JavaScript and CSS and develop an HTTP server to support web page functionality and for representing the status of the application we use an RGB LED.

TABLE OF CONTENTS

TITLE	PAGE NO.
TITLE PAGE	**i**
CERTIFICATE	**ii**
DECLARATION	**iii**
ACKNOWLEDGEMENT	**iv**
ABSTRACT	**v**
Chapter - I INTRODUCTION	**1**
1.1 Problem Statement	1 - 2
1.2 Objectives	2 - 3
1.3 Scope Of Study	3 - 4
1.4 Significance Of Study	4
Chapter - II LITERATURE SURVEY	**5 - 6**
Chapter - III METHODOLOGY	**7**
3.1 Design Of The Application	7
3.2 Features Of The Application	7 - 8
3.3 Hardware and Software Requirements	8
3.4 ESP32 Wrover Development Kit	9 - 10
3.4.1 Hardware Features Of ESP32 Wrover Kit	9
3.5 ESP32 Microcontroller	10 - 12
3.6 DHT22 Sensor	12 - 13
3.6.1 Specifications Of DHT22 Sensor	13
3.7 RGB LED Module	13 - 14
3.8 ESP-IDF Eclipse Plug-In	14 - 15
3.9 AWS – IoT Core	15 - 16
3.10 Build System For ESP-IDF	16 - 17
Chapter - IV IMPLEMENTATION	**18 - 65**
4.1 FreeRTOS	21 - 24
4.1.1 FreeRTOS Task States	22
4.1.2 xTaskCreatePinnedToCore Parameters	23
4.2 RGB LED Implementation	24 - 26
4.3 WiFi Application Implementation	27 - 33
4.4 HTTP Server Implementation	34 - 49

4.5 OTA Implementation 49
4.6 DHT22 Implementation 49 - 51
4.7 Non – Volatile Storage Implementation 52 - 55
4.8 SNTP Time Synchronization Implementation 56 - 57
4.8.1 Configuration Steps and Notable API's Used 56
4.9 AWS – IoT Core Implementation 58 - 65
4.9.1 MQTT 58
4.9.2 AWS IoT Device Authentication 59
4.9.3 Mutual TLS Authentication 59
4.9.4 AWS Security Overview 59
4.9.5 AWS IoT Policies 60
4.9.6 AWS IoT Configuration Steps 60
Chapter – V RESULTS 66 – 70
5.1 WiFi Intialization 66
5.2 Access Point Connection 66
5.3 Webpage Requests 67
5.4 Application Features 69
Chapter – VI CONCLUSION & FUTURE ENHANCEMENTS 71
Chapter – VII REFERENCES 72

LIST OF FIGURES

FIGURE NO.	TITLE	PAGE NO.
3.1	Design flow of the Application	7
3.2	Block Diagram	8
3.3	ESP - WROVER-KIT board layout	9
3.4	ESP - WROVER-KIT Block Diagram	10
3.5	ESP32 D0WDQ16 Chip	11
3.6	DHT22 Sensor	12
3.7	RGB SMD Module	14
3.8	ESP-IDE	14
3.9	AWS IoT Overview	15
3.10	Build system overview	16
3.11	CMake Project Setup	17
4.1	Creating the New Espressif IDF project	18
4.2	Serial flasher configuration	19
4.3	Factory app, two OTA definitions	19
4.4	Max HTTP Request and URI Header Length	19
4.5	mbedTLS Authentication	20
4.6	Building The Project	20
4.7	Selecting The Target COM port for flashing	20
4.8	Flashing ESP32	21
4.9	Launching the serial monitor	21
4.10	Valid Task State Transition	23
4.11	Circuit Connections	25
4.12	Simplified Overview of WiFi Application	27
4.13	WiFi Programming Model	27
4.14	Getting Started With AWS IoT Core	58
4.15	MQTT Protocol	58
4.16	AWS IoT Device Authentication	59
4.17	AWS Security Overview	60
4.18	Create a Thing in AWS Management Console	64

4.19	Select The MQTT Test Client	65
4.20	Subscribing and Publishing The Data via Topic Filter	65
5.1	WiFi and HTTP Server Started	66
5.2	Connect to the ESP32’s Access Point	66
5.3	WiFi Connected Status	67
5.4	Enter the IP Address	67
5.5	Server Requests for Loading the Webpage	67
5.6	After Connecting to The Internet	68
5.7	Application Webpage Connected Through ESP32	68
5.8	Subscribed Data in the AWS Console	69
5.9	The data has been transferred through callback function	69
5.10	Published Message from the AWS console	69
5.11	WiFi Event Started Indication	70
5.12	HTTP Server Started Indication	70
5.13	Internet Connected Indication	70

CHAPTER I
INTRODUCTION

The **Internet of Things (IoT)** has transformed the way we interact with the world around us, enabling objects to connect and communicate through the internet, creating a network of interconnected devices. This revolution has opened up new possibilities for creating smart environments, optimizing processes, and improving efficiency in various domains, such as smart homes, smart cities, industrial automation, agriculture, and more. To realize the full potential of IoT, robust and scalable solutions are required to effectively manage and process data from connected devices.

The ESP32 is a versatile microcontroller with built-in WiFi capabilities, making it ideal for creating IoT devices that can connect to the internet and communicate wirelessly. It comes equipped with ample processing power, storage capacity, and support for various communication protocols, making it a popular choice for IoT development.

The main focus of this project is to leverage the capabilities of ESP32 and AWS IoT to design and implement a WiFi-based IoT application that can collect, transmit, and process data from various sensors and actuators in a connected environment. The project will involve several stages, including sensor integration, data transmission to the cloud, cloud-based data processing, and visualization of the data on a web-based dashboard.

In summary, this project aims to utilize the capabilities of ESP32 and AWS IoT to create a WiFi-based IoT application that can efficiently collect, transmit, and process data from connected devices. With its potential for scalability, security, and versatility, this project has the potential to contribute to the advancement of IoT technologies and pave the way for future innovations in the field of IoT applications.

1.1 Problem Statement

As technology continues to advance, the Internet of Things (IoT) has become increasingly popular, allowing for the integration of various devices and sensors into a network for data collection, analysis, and automation. However, there are challenges in building IoT applications, including the need for reliable and secure communication between edge devices and cloud platforms, as well as the complexity of integrating different components and technologies.

One specific problem is the lack of a comprehensive and practical solution for building WiFi-based IoT applications using the ESP32 microcontroller and the AWS IoT platform. While ESP32 is a powerful and affordable microcontroller with built-in WiFi capabilities, and AWS IoT provides a robust cloud-based platform for managing and analyzing IoT data, there is a need for a cohesive and practical approach to integrating these technologies for building IoT applications.

This problem statement is supported by the fact that many IoT projects face challenges in securely transmitting data from edge devices to the cloud, ensuring data integrity, managing devices remotely, and developing user-friendly applications for data visualization and control. There is a need for a solution that addresses these challenges and provides a practical framework for building WiFi-based IoT applications using ESP32 and AWS IoT.

Therefore, the problem statement for this project is to develop a comprehensive and practical solution for building a WiFi-based IoT application using the ESP32 microcontroller and the AWS IoT platform, addressing challenges related to secure data transmission, device management, and application development. The solution will aim to provide an accurate, reliable, and secure way of building IoT applications using these technologies, contributing to the advancement of IoT research and development.

1.2 Objectives

Designing and Developing IoT Application - The primary objective of the project is to design and develop a WiFi-based IoT application using ESP32 microcontroller and AWS IoT services. This includes developing the software architecture, designing the communication protocol, implementing data collection, and processing, and integrating AWS IoT services for seamless communication and data management.

Enhancing Connectivity and Processing Power - The project aims to optimize the connectivity and processing power of the IoT devices, specifically ESP32 microcontrollers, by leveraging their capabilities to ensure efficient real-time data transmission, processing, and analytics.

Implementing Robust Security Mechanisms - Security is a critical aspect of IoT applications. This includes implementing encryption, authentication, and access control mechanisms to ensure secure communication, device provisioning, and data storage.

Developing User-Friendly Interfaces - The project aims to develop user-friendly interfaces for interacting with the IoT application and visualizing the collected data. This includes developing a web-based dashboard or user interface that provides meaningful insights, actionable information, and seamless control over the IoT devices.

Deploying Scalable and Reliable IoT Infrastructure - This includes setting up AWS IoT Core, configuring device provisioning, authentication, authorization, and device management, and ensuring seamless communication, data storage, and processing between the IoT devices and AWS IoT platform.

1.3 Scope Of Study

The scope of this study involves developing an Internet of Things (IoT) application using the ESP32 microcontroller and the Amazon Web Services (AWS) IoT platform. The study will focus on building a WiFi-based IoT application, where the ESP32 will act as the edge device, collecting data from sensors and sending it to the AWS IoT platform for further processing and analysis.

The study will include the following key components:

ESP32 Development - The study will involve understanding and utilizing the ESP32 microcontroller, including its features, capabilities, and programming environment. The ESP32 will be programmed using the Espressif-IDE, which is a popular development environment for ESP32-based projects.

Sensor Integration - The study will involve integrating sensors with the ESP32 to collect data. Sensors such as temperature, humidity sensors will be used to simulate real-world IoT scenarios. The ESP32 will read data from these sensors and send it to the AWS IoT platform for processing.

AWS IoT Platform - The study will involve understanding the AWS IoT platform, including its components such as AWS IoT Core, AWS IoT Rules Engine, and AWS IoT Device Shadow. AWS IoT Core is a cloud-based service that allows for the communication and management of IoT devices, while the AWS IoT Rules Engine enables the processing and analysis of data from IoT devices. The AWS IoT Device Shadow provides a virtual representation of the ESP32 device in the cloud, allowing for remote device management.

Data Transmission and Security - The study will involve implementing secure communication between the ESP32 and the AWS IoT platform using the MQTT (Message Queuing Telemetry Transport) protocol, which is a lightweight messaging protocol for IoT applications.

Application Development - The study will involve developing an IoT application to demonstrate the functionality of the ESP32 and AWS IoT integration.

1.4 Significance Of Study

The study on "Building a WiFi-Based IoT Application Using ESP32 and AWS IoT" holds significant importance due to several reasons. Firstly, it contributes to the advancement of IoT technology by showcasing how cutting-edge technologies like ESP32 microcontrollers and AWS IoT services can be leveraged to build innovative IoT applications.

Furthermore, the project aligns with current industry trends and demands for reliable, secure, and scalable communication between IoT devices and cloud platforms, making it relevant to industries such as smart home automation, industrial automation, healthcare, transportation, agriculture, and more.

In addition, the study contributes to the existing body of knowledge in the field of IoT, showcasing a practical implementation of a WiFi-based IoT application using ESP32 and AWS IoT.

Innovation and contribution are potential outcomes of the project. The project may involve designing and implementing novel features or functionalities in the IoT application, addressing a specific problem, or providing a solution to a real-world challenge. By exploring the potential of these technologies, the project seeks to push the boundaries of what is currently possible in the field of IoT, contributing to the evolution and advancement of IoT technology.

CHAPTER II

LITERATURE SURVEY

1. Comparative Analysis and Practical Implementation of the ESP32 Microcontroller Module for the Internet of Things

This paper discusses the Espressif Systems latest product ESP32 designed for Internet of Things and embedded system related projects. The ESP32 is a low-cost, low-power system on a chip series of microcontrollers with Wi-Fi and Bluetooth capabilities and a highly integrated structure powered by a dual-core Tensilica Xtensa LX6 microprocessor.

2. Using the ESP32 Microcontroller for Data Processing

This article deals with experiences with the development of applications of the ESP32 microcontrollers and provides a comprehensive review of the possibilities of applications development on this platform in the area of data measurement and processing. Microcontrollers usually connect with IoT modules and other smart sensors and provide data to the superior system.

3. Industrial Process Monitoring System Using ESP32

Today's word is internet world, Internet of Things (IoT) is expanding at rapid rate increasing technology. A network of connected computers hidden in every corner of our life monitoring and controlling things with minimal IOT supports to connect hardware devices to the internet to process the data for monitoring and security. This system uses ESP32, and sensors helps to monitor the different parameters like temperature, humidity, smoke etc. are accessed and monitored from remote area by cloud computing the data using Blink and automatically controls the motor or system operation by computer or smartphone. This system is very useful for small scale industry for to achieve maximum throughput and to avoid from accidents by SMS alerts and email, it is a simple, smart monitoring and security system and also tells the importance of IoT in industrial applications. Proposed method very well suitable for small scale industries monitoring and controlling. Industrial process site and accessing the monitoring system from any device connected to the network or through a public IP address with port forwarding.

4. Design and implementation of a low-cost web server using ESP32 for real – time photovoltaic system monitoring

This paper presents a method for applying a web server based on ESP32 to a small photovoltaic (PV) power system to monitor and/or collect the PV and the battery current, voltage data. The designed system uses low-cost sensors, a microcontroller ESP32, Wi-Fi, and an SD-card reader. The ESP32 collects data from sensors. All of the data are saved in a text file on an SD-card, which is connected with ESP32 SPI pins. The text file is saved on the SD card for a week or longer, after which the system deletes all the data and starts saving new data. The web page file is saved on an SD card as well, so the ESP32 is programmed to access the web page by using the Wi-Fi via a laptop, cell phone, or tablet.

5. A Faust Architecture for the ESP32 Microcontroller

This paper introduces faust2esp32, a tool to generate digital signal processing engines for the ESP32 microcontroller family. It can target both the C++ and the Arduino ESP32 programming environment, and it supports a wide range of audio codecs, making it compatible with most ESP32-based prototyping boards. After demonstrating how to use faust2esp32 and providing technical details about its implementation, we evaluate its performances, and we present the FAUST Gramophone which is a programmable instrument taking advantage of this technology.

6. Low-cost, Open Source IoT Based SCADA System Design Using Thinger.IO and ESP32 Thing

Supervisory Control and Data Acquisition (SCADA) is a technology for monitoring and controlling distributed processes. SCADA provides real-time data exchange between a control/monitoring centre and field devices connected to the distributed processes. A SCADA system performs these functions using its four basic elements: Field Instrumentation Devices (FIDs) such as sensors and actuators which are connected to the distributed process plants being managed, Remote Terminal Units (RTUs) such as single board computers for receiving, processing, and sending the remote data from the field instrumentation devices, Master Terminal Units .

CHAPTER III

METHODOLOGY

3.1 Design Of The Application

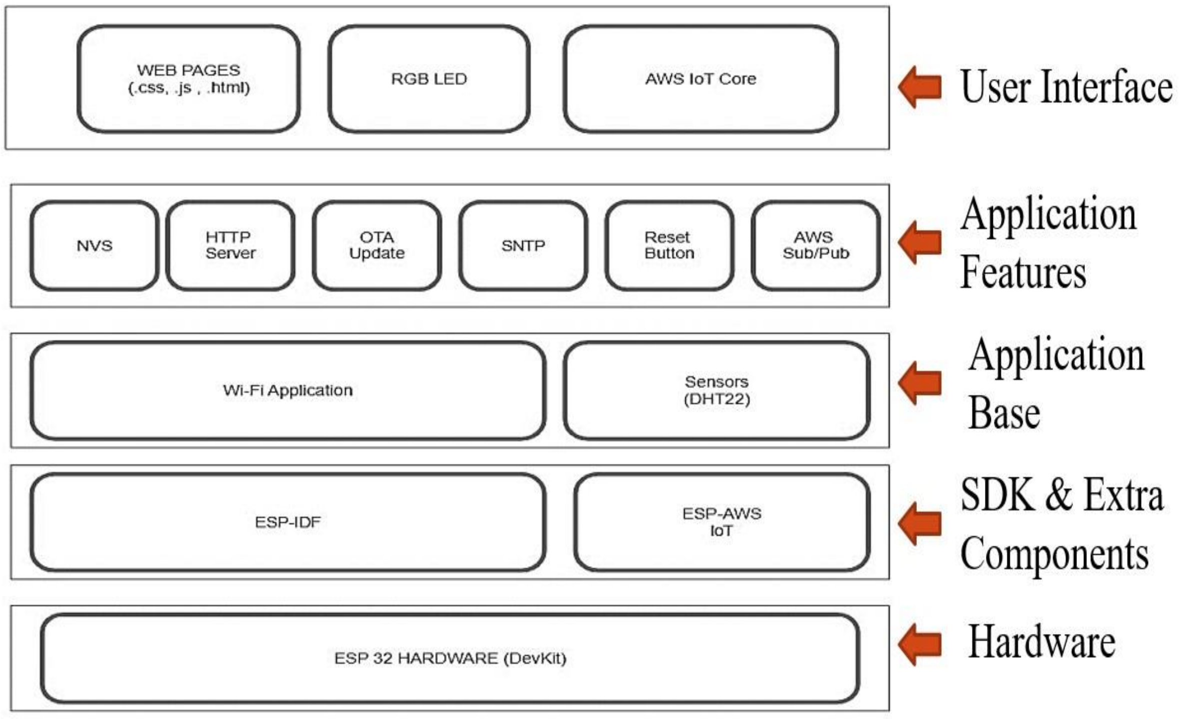


Figure 3.1 Design flow of the Application

The design of the application consists of Hardware, Software Development Kit, Application base, Features which are going to implement and User interface for communicating between end user and the Application.

3.2 Features Of The Application

- The WiFi application which is used for connecting to the ESP's access point to access an HTTP server and connect the ESP32 as an access point.
- HTTP server is used for supports OTA (over the air) firmware updates via a web page, connecting and disconnecting the ESP32 to an Access point and displaying the data on the page (temperature / humidity sensor data, local time using SNTP and ESP32's SSID).

- The DHT22 sensor provides the temperature and humidity data after being programmed via webserver and .html , .js and .css which are going to be used for creating the webpage and making it functional.
- The Non-Volatile storage is used for saving and loading WiFi creditionals.
- SNTP is used for getting the local time and display on the web page.
- Button with Interrupt and Semaphore used for disconnecting WiFi and clearing credentials.

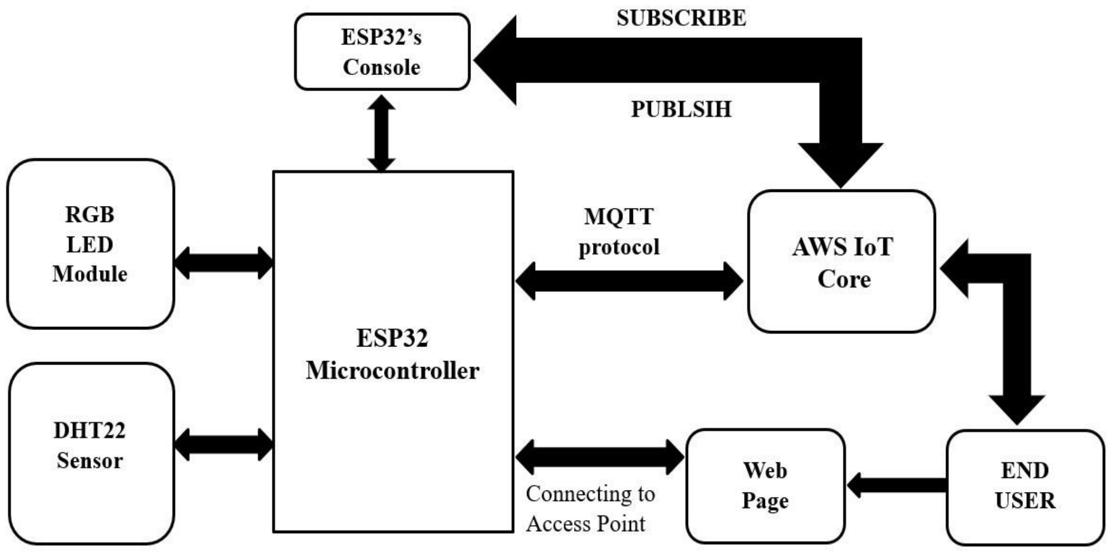


Figure 3.2 Block Diagram

3.3 Hardware and Software Requirements

The Hardware requirements for this application as follows

- ESP32 Wrover Development Kit.
- DHT 22 Sensor.
- RGB LED Module.

The Software requirements for this application are

- ESP-IDF (IoT Development Framework) plugin.
- Espressif IDE (Integrated Development Environment).
- AWS IoT core.

3.4 ESP32 Wrover Development Kit

The ESP32-WROVER Development Kit is an all-in-one development platform designed for building Internet of Things (IoT) applications using the ESP32-WROVER module, which is a powerful and feature-rich variant of the ESP32 microcontroller. The development kit provides a wide range of hardware components, software tools, and documentation, making it an ideal choice for developers looking to create advanced IoT projects.

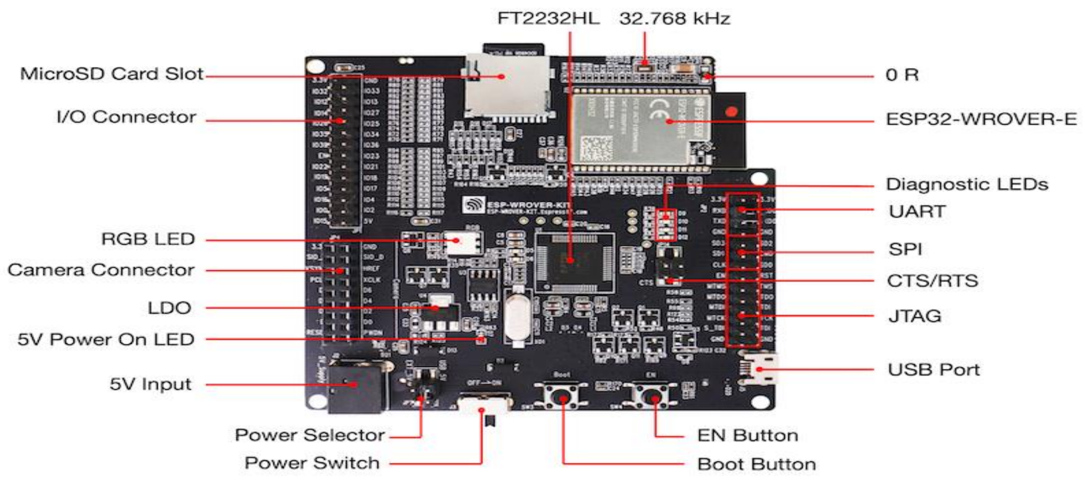


Figure 3.3 ESP-WROVER-KIT board layout

3.4.1 Hardware Features of ESP32 Wrover Kit

The ESP32-WROVER Development Kit includes several key hardware features that make it a versatile and powerful development platform:

- **ESP32-WROVER Module -** The heart of the development kit is the ESP32-WROVER module, which is based on the ESP32 microcontroller. The ESP32-WROVER module comes with 4 MB of external SPI flash memory and 8 MB of external PSRAM, providing ample storage space for firmware and data.
- **Dual-Core Processor -** The ESP32-WROVER module is powered by a dual-core Xtensa LX6 processor, which can run at up to 240 MHz . The dual-core architecture allows for efficient multitasking and parallel processing, making it suitable for applications that require high performance and real-time processing.
- **Wi-Fi and Bluetooth Connectivity -** The ESP32-WROVER module comes with built-in Wi-Fi and Bluetooth capabilities, allowing for seamless wireless

communication. It supports both 2.4 GHz and 5 GHz Wi-Fi bands, as well as various Wi-Fi security protocols, making it compatible with a wide range of Wi-Fi networks.

- **Extensive I/O Options -** The development kit provides a wide range of I/O options, including GPIO pins, SPI, I2C, UART, and more, allowing for easy interfacing with sensors, actuators, and other devices. It also includes a microSD card slot for additional storage and an LCD interface for connecting displays.
- **JTAG Debugging -** The ESP32-WROVER Development Kit comes with a built-in JTAG debugging interface, which allows for easy debugging and firmware development. The JTAG interface enables developers to set breakpoints, inspect variables, and step through code.

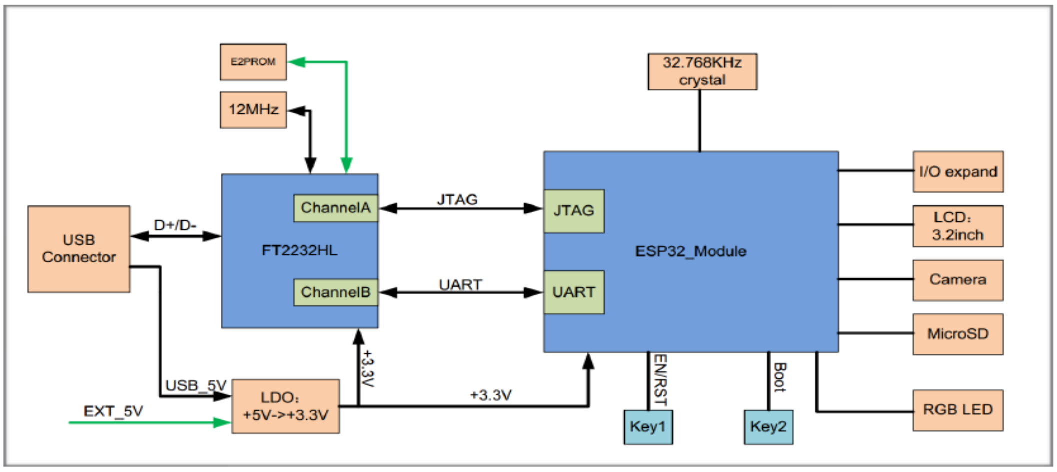


Figure 3.4 ESP-WROVER-KIT Block Diagram

3.5 ESP32 Microcontroller

The ESP32 microcontroller is based on the Xtensa LX6 dual-core processor architecture, developed by Tensilica, which is optimized for embedded applications. It features two CPU cores that operate at a clock speed of up to 240 MHz each, providing ample processing power for running complex tasks and applications.

The ESP32 processor also comes with built-in Wi-Fi and Bluetooth capabilities, which are key features that make it ideal for wireless communication in IoT applications. The Wi-Fi functionality supports both 2.4 GHz and 5 GHz frequency bands, providing flexibility and compatibility with various Wi-Fi networks.

It supports standard 802.11 b/g/n and also offers features such as soft access point (AP) mode, station (STA) mode, and simultaneous AP/STA mode, allowing for versatile networking configurations.

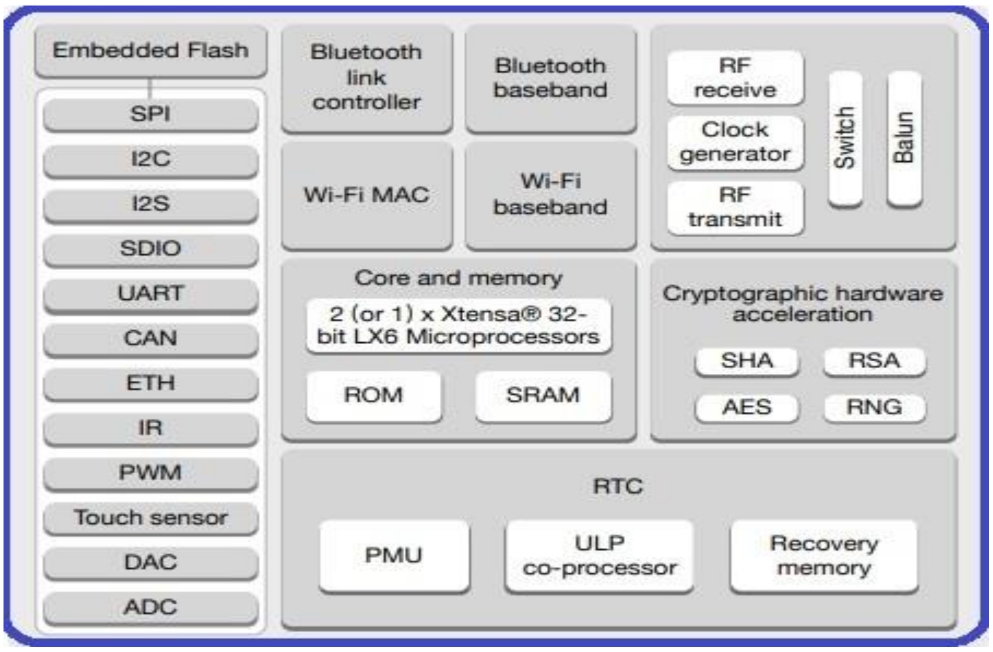


Figure 3.5 ESP32 D0WDQ16 Chip

In terms of Bluetooth capabilities, the ESP32 supports both Bluetooth Classic and Bluetooth Low Energy (BLE) communication protocols. Bluetooth Classic allows for communication with legacy devices, while BLE is optimized for low-power, low-data-rate applications, making it suitable for battery-powered devices and applications that require energy efficiency.

The ESP32 processor also comes with a rich set of peripherals and interfaces, making it highly versatile and suitable for a wide range of IoT applications. These GPIO pins can support a variety of digital and analog input/output operations, making them highly flexible for different types of applications.

In addition to GPIO pins, the ESP32 processor supports popular communication protocols such as UART, SPI, I2C, I2S, and PWM, which allow for easy interfacing with a wide range of devices. UART (Universal Asynchronous Receiver/Transmitter) allows for serial communication, while SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), and I2S (Inter-IC Sound) are commonly used for communication with sensors, displays, and other peripherals. PWM (Pulse Width Modulation) is often used for controlling motors, LEDs, and other devices that require analog-like control.

The ESP32 processor also includes an Analog-to-Digital Converter (ADC) that allows for analog signal input, making it suitable for applications that require analog sensor readings. It typically supports multiple ADC channels with up to 12-bit resolution, providing accurate analog-to-digital conversion for a wide range of sensor inputs.

In terms of memory, the ESP32 processor typically includes up to 520 KB of SRAM (Static Random-Access Memory) for data storage, which can be used for storing variables, buffers, and other runtime data.

The ESP32 processor is supported by a robust development ecosystem, including the ESP-IDF (IoT Development Framework) software development kit (SDK) provided by Espressif, which includes libraries, tools, and examples for developing firmware for the ESP32.

3.6 DHT22 SENSOR

The DHT22 sensor, also known as the AM2302 sensor, is a digital temperature and humidity sensor that is widely used in various applications for measuring environmental parameters. It is a popular choice among hobbyists and professionals due to its high accuracy and reliability. The DHT22 sensor is capable of measuring both temperature and humidity simultaneously, making it suitable for a wide range of applications such as home automation, weather stations, greenhouses, HVAC systems, and more.

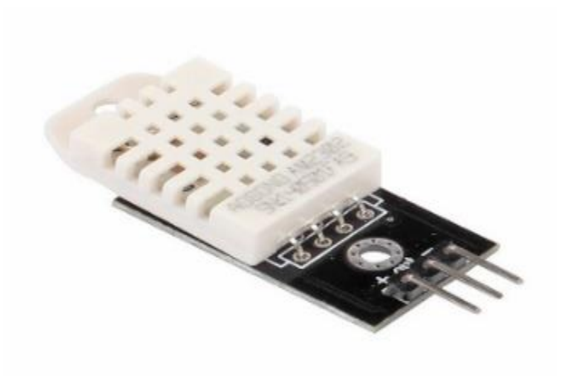


Figure 3.6 DHT22 Sensor

The DHT22 sensor's technical specifications, including its temperature measurement range of -40°C to 125°C, humidity measurement range of 0% to 100%, accuracy of ±0.5°C for temperature and ±2% RH for humidity, and resolution of 0.1°C for temperature and 0.1% RH for humidity, make it a reliable sensor for precise monitoring of environmental parameters.

3.6.1 Specifications

- **Temperature Measurement Range:** The DHT22 sensor has a temperature measurement range of -40°C to 125°C, making it suitable for a wide range of temperature monitoring applications.
- **Humidity Measurement Range:** The DHT22 sensor has a humidity measurement range of 0% to 100%, allowing for accurate measurement of humidity levels in various environments.
- **Accuracy:** The DHT22 sensor has a high accuracy level, with a temperature accuracy of ±0.5°C and a humidity accuracy of ±2% RH. This makes it suitable for applications that require precise temperature and humidity measurements.
- **Resolution:** The DHT22 sensor has a temperature resolution of 0.1°C and a humidity resolution of 0.1% RH, providing fine-grained measurements for accurate monitoring.
- **Interface:** The DHT22 sensor uses a single-wire digital interface for communication, making it easy to integrate with microcontrollers and other devices. It operates on a voltage range of 3.3V to 6V, making it compatible with a wide range of microcontrollers and development boards.
- **Response Time:** The DHT22 sensor has a fast response time, with a temperature response time of 2 seconds and a humidity response time of 2-5 seconds, allowing for real-time monitoring of temperature and humidity changes. It provides valuable data for monitoring and controlling temperature and humidity levels in these applications, enabling efficient management of environmental conditions.

3.7 RGB LED Module

The HW-478 RGB LED is a popular and versatile lighting component that offers vibrant and dynamic lighting effects. It is commonly used in various electronic projects, including lighting fixtures, displays, signage, automotive lighting, and home automation systems. The HW-478 RGB LED is known for its ability to emit light in different colors, including red, green, and blue, which can be combined to create a wide spectrum of colors.

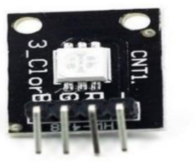


Figure 3.7 RGB SMD Module

The HW-478 RGB LED typically features a compact size with three separate LED chips, one for each color (red, green, and blue), housed in a single package. Each LED chip is independently controlled, allowing for precise control over the intensity and color of the emitted light. The LED is typically designed to operate at a low voltage, usually around 3-5V, making it compatible with a wide range of microcontrollers and other electronic devices.

3.8 ESP-IDF Eclipse Plug-In

ESP-IDF Eclipse Plugin brings developers an easy-to-use Eclipse-based development environment for developing ESP32 based IoT applications. It provides better tooling capabilities, which simplifies and enhances standard Eclipse CDT for developing and debugging ESP32 IoT applications.

Figure 3.8 ESP-IDE

Installing Prerequisites

- Java 17 and above.
- Python 3.6 and above.
- Eclipse IDE for c/c++ Developers.
- Git
- ESP-IDF 4.0 and above.

We can follow the procedure from the official website of the Espressif for Installation.

We can install all the prerequisites as mentioned or we can also download all in one installer which installs the required software utilities for the installation.

3.9 AWS-IoT Core

AWS IoT Core is a cloud-based service offered by Amazon Web Services (AWS) that provides a robust and scalable platform for connecting, managing, and securing IoT (Internet of Things) devices. It enables businesses to securely connect and manage a large number of IoT devices, collect and analyze data from these devices, and take actions based on the insights gained from the data. AWS IoT Core offers a wide range of features and capabilities that make it a comprehensive solution for building and managing IoT applications at scale.

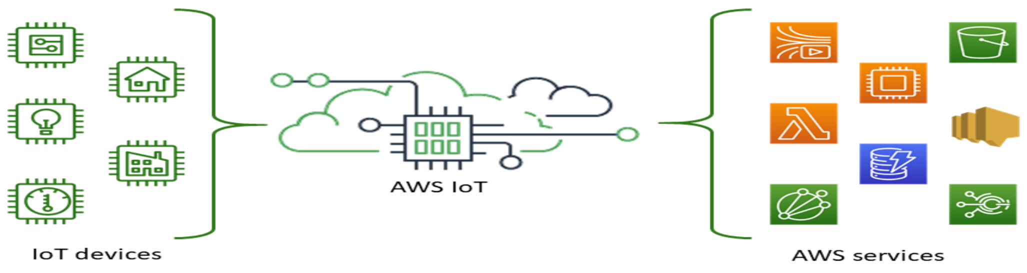


Figure 3.9 AWS IoT Overview

One of the key features of AWS IoT Core is its device management capabilities. It provides tools for onboarding, registering, and organizing IoT devices, making it easy to connect and manage a large number of devices securely. AWS IoT Core supports various IoT protocols, such as MQTT, HTTP, and WebSocket, allowing devices to communicate seamlessly with the cloud.

AWS IoT Core also offers powerful data management and analytics features. It allows for the collection, processing, and storage of data from IoT devices in a scalable and efficient manner. The data can be ingested into AWS services like Amazon S3, Amazon Kinesis, or AWS Lambda for further processing, analysis, and visualization.

AWS IoT Core supports mutual authentication between devices and the cloud, ensuring that only trusted devices can connect and communicate with the system. It also offers features like encryption of data in transit and at rest, fine-grained access control, and auditing of device actions, providing a secure environment for IoT deployments. AWS IoT Core also integrates with other AWS security services, such as AWS Identity and Access Management (IAM) and AWS Key Management Service (KMS), enabling end-to-end security across the IoT ecosystem.

The rules engine allows users to define rules and actions based on the incoming data from devices, such as filtering, transforming, and routing data to other AWS services or external systems. Its features include robust device management, data management and analytics, security, rules engine, and seamless integration with other AWS services. AWS IoT Core is widely used by businesses across various industries for building IoT applications at scale, ranging from smart home devices, industrial automation, smart cities, and more.

3.10 Build System For ESP-IDF

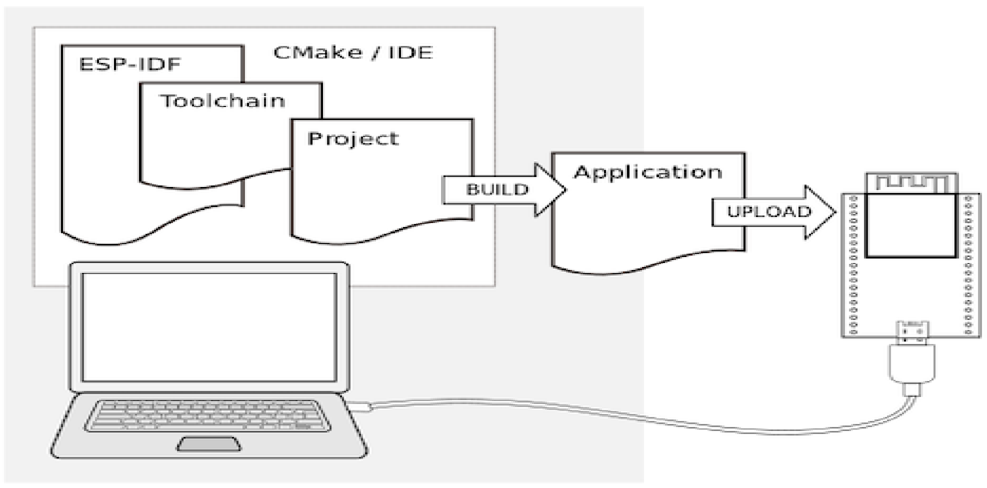


Figure 3.10 Build system overview

- Project – A directory that contain all files and configuration to build a single “app’.
- Project Configuration – It is held in a single file called sdkconfig in the root directory.
- App – An executable which is built by ESP-IDF.
- Components – These are the modular pieces of standalone code which are compiled into static libraries (.a files) and linked into an app.
- Target – The hardware for which an application is built.

For using the build system and CMake.

idf.py

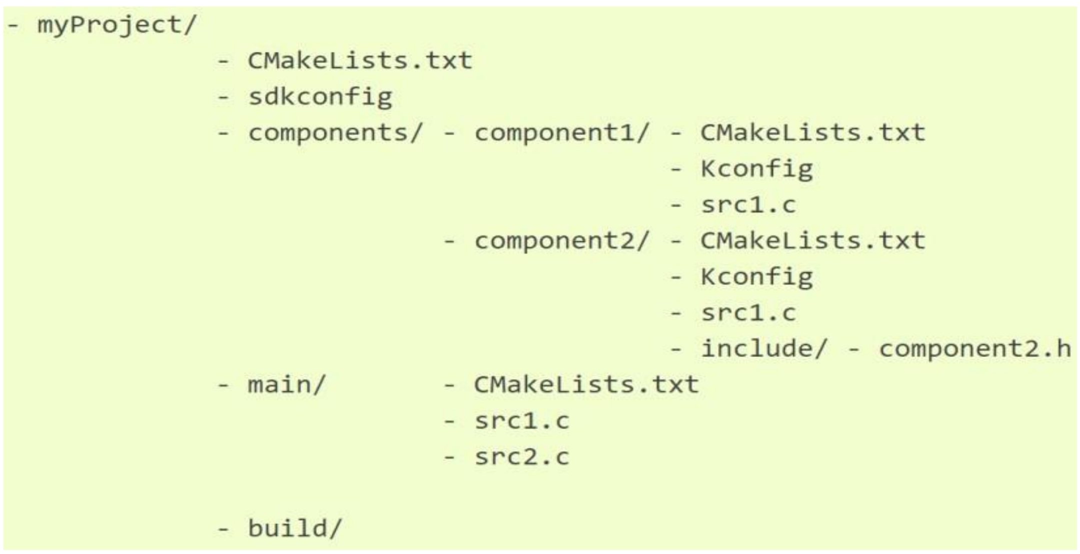
- CMake – configures the project to be built.
- Ninja – builds the project.
- esptool.py – used for flashing the target.

CMake Directly

- idf.py is a wrapper around CMake for convenience.

CMake in an IDE

- Use an IDE with CMake integration.



```
- myProject/
             - CMakeLists.txt
             - sdkconfig
             - components/ - component1/ - CMakeLists.txt
                                         - Kconfig
                                         - src1.c
                           - component2/ - CMakeLists.txt
                                         - Kconfig
                                         - src1.c
                                         - include/ - component2.h
             - main/       - CMakeLists.txt
                           - src1.c
                           - src2.c

             - build/
```

Figure 3.11 CMake Project Setup

The CMakeLists.txt file typically starts with specifying the minimum CMake version required for the project. This ensures that the correct version of CMake is used during the build process, preventing any potential compatibility issues. The version is specified using the cmake_minimum_required() function, and the minimum required version is typically mentioned as per the ESP-IDF documentation or project requirements. The project's source files are then specified using the set() function. This includes all the source files that make up the project, such as main.c, app_task.c, and any other source files that are part of the project.

CHAPTER IV

IMPLEMENTATION

Firstly, after installing the ESP-IDE we need to create an Espressif-IDF project.

- Open the workspace.
- Go to New.
- Select the Espressif IDF Project.
- Give the Project and click finish.

The project is created.

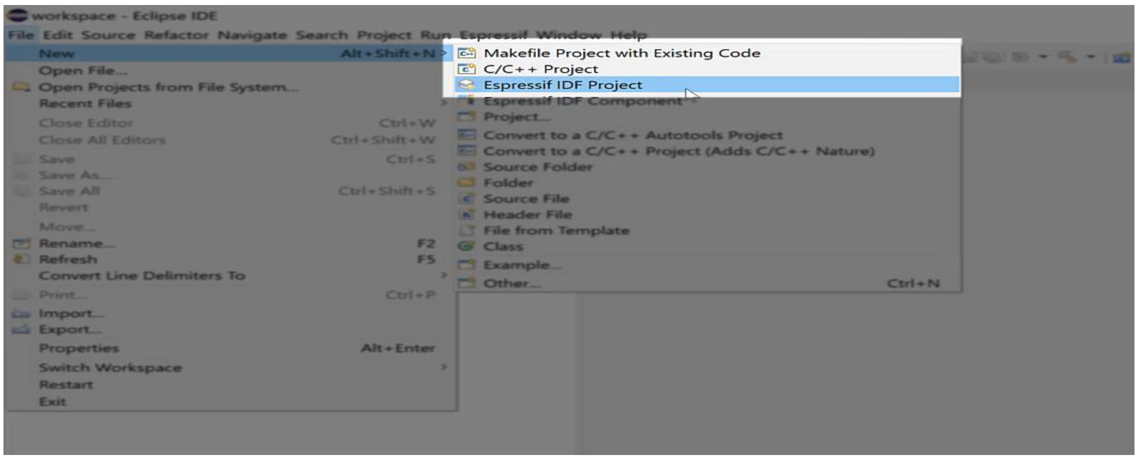


Figure 4.1 Creating the New Espressif IDF project

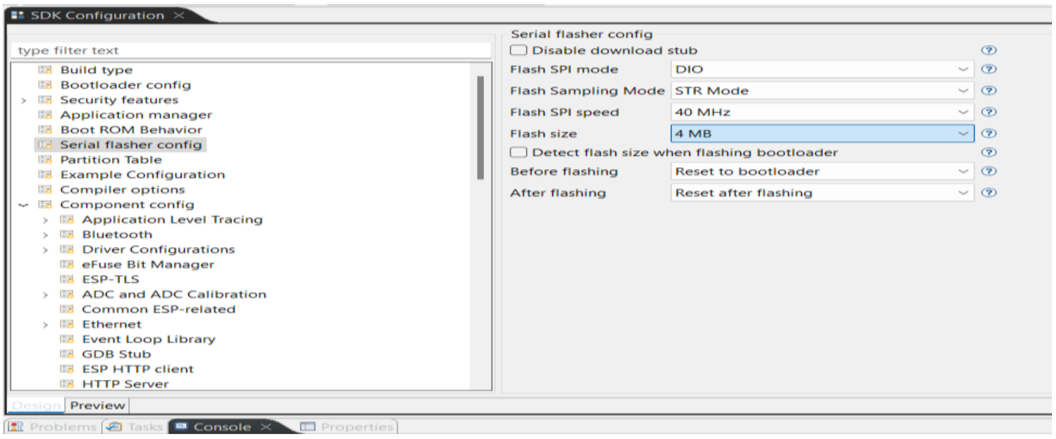
In order to flash and program the ESP32 we, need to configure the sdkconfig file in the Directory.

Things we need to configure in the sdkconfig are

- Serial flasher config.
- Partition Table.
- HTTP Server.
- mbedTLS.

For Serial flasher config,

- Go to flash size. Select 4MB as the ESP32 flash size by default is 4MB.



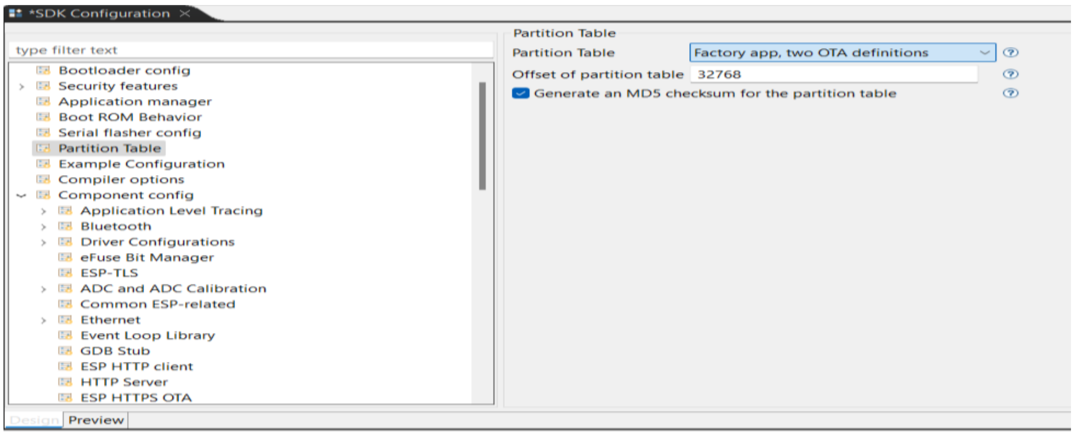
Serial flasher config

☐ Disable download stub	
Flash SPI mode	DIO
Flash Sampling Mode	STR Mode
Flash SPI speed	40 MHz
Flash size	4 MB
☐ Detect flash size when flashing bootloader	
Before flashing	Reset to bootloader
After flashing	Reset after flashing

Figure 4.2 Serial flasher configuration

For Partition Table,

- Go to Partition Table – Select “Factory app, two OTA definitions”



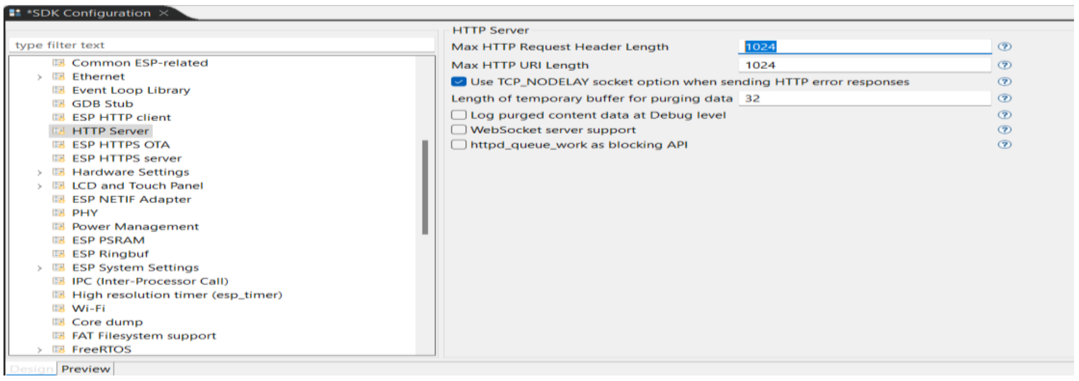
Partition Table

Partition Table	Factory app, two OTA definitions
Offset of partition table	32768
✓ Generate an MD5 checksum for the partition table	

Figure 4.3 Factory app, two OTA definitions

For HTTP Server,

- Go to Max HTTP Request Header Length and change length to 1024.



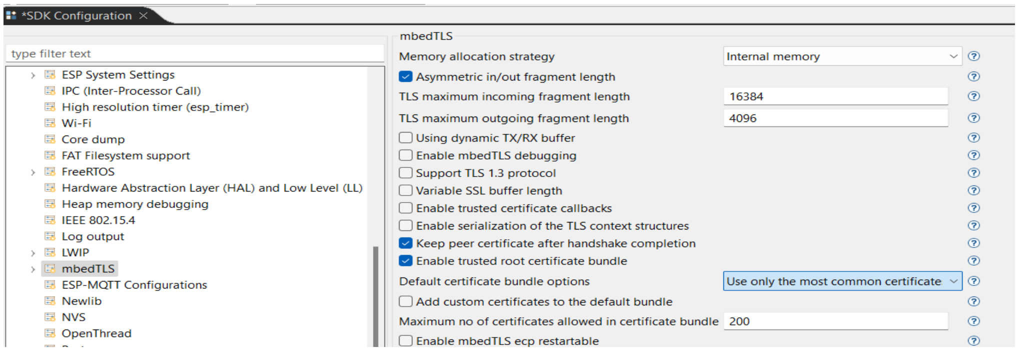
HTTP Server

Max HTTP Request Header Length	1024
Max HTTP URI Length	1024
✓ Use TCP_NODELAY socket option when sending HTTP error responses	
Length of temporary buffer for purging data	32
☐ Log purged content data at Debug level	
☐ WebSocket server support	
☐ httpd_queue_work as blocking API	

Figure 4.4 Max HTTP Request and URI Header Length

For mbedTLS,

- Go to mbedTLS and change the Default certificate bundle options.
- Select Use only the most common certificates.



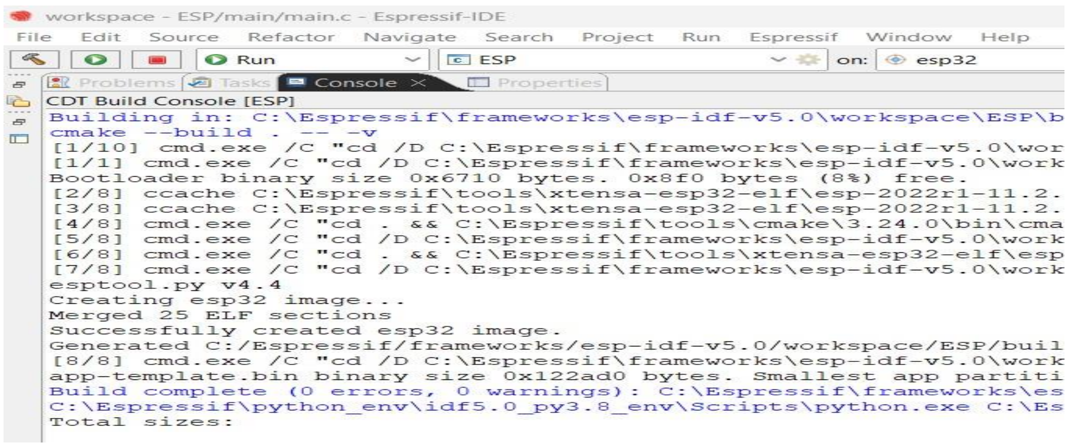
mbedTLS

Memory allocation strategy	Internal memory
✓ Asymmetric in/out fragment length	
TLS maximum incoming fragment length	16384
TLS maximum outgoing fragment length	4096
☐ Using dynamic TX/RX buffer	
☐ Enable mbedTLS debugging	
☐ Support TLS 1.3 protocol	
☐ Variable SSL buffer length	
☐ Enable trusted certificate callbacks	
☐ Enable serialization of the TLS context structures	
✓ Keep peer certificate after handshake completion	
✓ Enable trusted root certificate bundle	
Default certificate bundle options	Use only the most common certificate
☐ Add custom certificates to the default bundle	
Maximum no of certificates allowed in certificate bundle	200
☐ Enable mbedTLS ecp restartable	

Figure 4.5 mbedTLS Authentication

After changing the configuration in sdkconfig, save the file and Build the project.

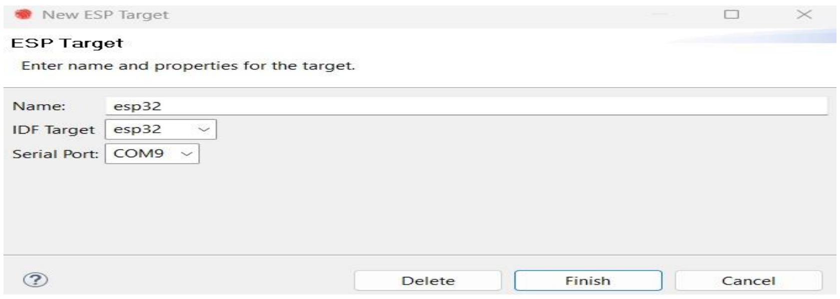
After compiling and building the project , we can observe the output on the console.


workspace - ESP/main/main.c - Espressif-IDE
File Edit Source Refactor Navigate Search Project Run Espressif Window Help
Run
ESP
on: esp32
Problems Tasks Console Properties

```
CDT Build Console [ESP]
Building in: C:\Espressif\frameworks\esp-idf-v5.0\workspace\ESP\b
cmake --build . -- -v
[1/10] cmd.exe /C "cd /D C:\Espressif\frameworks\esp-idf-v5.0\wor
[1/1] cmd.exe /C "cd /D C:\Espressif\frameworks\esp-idf-v5.0\work
Bootloader binary size 0x6710 bytes. 0x8f0 bytes (8%) free.
[2/8] ccache C:\Espressif\tools\xtensa-esp32-elf\esp-2022r1-11.2.
[3/8] ccache C:\Espressif\tools\xtensa-esp32-elf\esp-2022r1-11.2.
[4/8] cmd.exe /C "cd . && C:\Espressif\tools\cmake\3.24.0\bin\cma
[5/8] cmd.exe /C "cd /D C:\Espressif\frameworks\esp-idf-v5.0\work
[6/8] cmd.exe /C "cd . && C:\Espressif\tools\xtensa-esp32-elf\esp
[7/8] cmd.exe /C "cd /D C:\Espressif\frameworks\esp-idf-v5.0\work
esptool.py v4.4
Creating esp32 image...
Merged 25 ELF sections
Successfully created esp32 image.
Generated C:/Espressif/frameworks/esp-idf-v5.0/workspace/ESP/buil
[8/8] cmd.exe /C "cd /D C:\Espressif\frameworks\esp-idf-v5.0\work
app-template.bin binary size 0x122ad0 bytes. Smallest app partiti
Build complete (0 errors, 0 warnings): C:\Espressif\frameworks\es
C:\Espressif\python_env\idf5.0_py3.8_env\Scripts\python.exe C:\Es
Total sizes:
```

Figure 4.6 Building the Project

After building , we need to flash the program into the ESP32 .

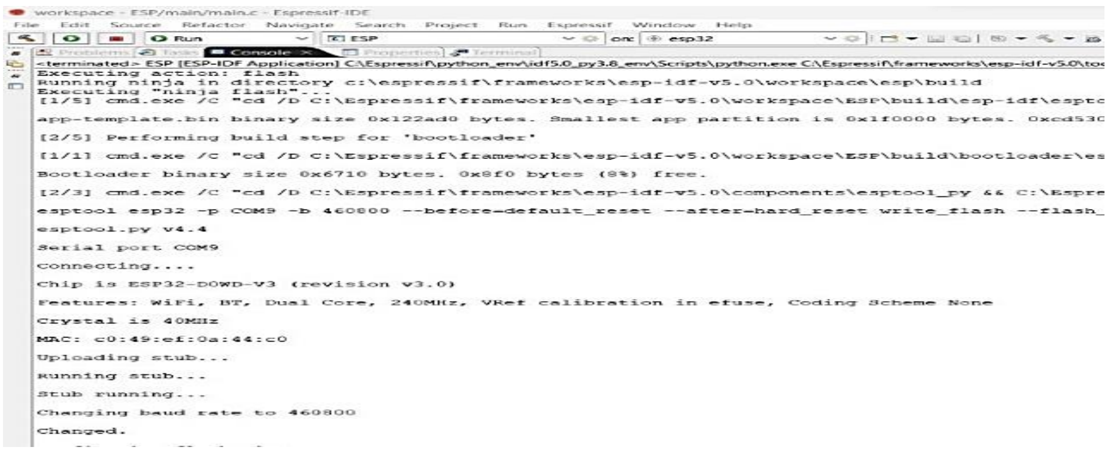


ESP Target

Enter name and properties for the target.

Name:	esp32
IDF Target	esp32
Serial Port:	COM9

Figure 4.7 Selecting the target COM port for flashing

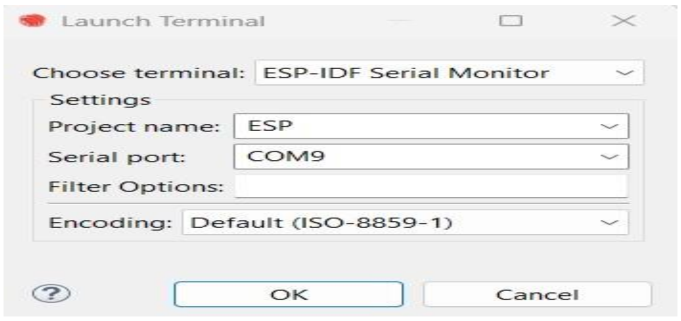


workspace - ESP/main/main.c - Espressif-IDE
File Edit Source Refactor Navigate Search Project Run Espressif Window Help
Run
ESP
on: esp32
Problems Tasks Console Properties Terminal

```
<terminated> ESP [ESP-IDF Application] C:\Espressif\python_env\idf5.0_py3.8_env\Scripts\python.exe C:\Espressif\frameworks\esp-idf-v5.0\too
Executing action: flash
Running ninja in directory c:\espressif\frameworks\esp-idf-v5.0\workspace\esp\build
Executing "ninja flash"...
[1/5] cmd.exe /C "cd /D C:\Espressif\frameworks\esp-idf-v5.0\workspace\ESP\build\esp-idf\esptc
app-template.bin binary size 0x122ad0 bytes. Smallest app partition is 0x1f0000 bytes. 0xcd530
[2/5] Performing build step for 'bootloader'
[1/1] cmd.exe /C "cd /D C:\Espressif\frameworks\esp-idf-v5.0\workspace\ESP\build\bootloader\es
Bootloader binary size 0x6710 bytes. 0x8f0 bytes (8%) free.
[2/3] cmd.exe /C "cd /D C:\Espressif\frameworks\esp-idf-v5.0\components\esptool_py && C:\Espre
esptool esp32 -p COM9 -b 460800 --before=default_reset --after=hard_reset write_flash --flash_
esptool.py v4.4
Serial port COM9
Connecting....
Chip is ESP32-D0WD-V3 (revision v3.0)
Features: WiFi, BT, Dual Core, 240MHz, VRef calibration in efuse, Coding Scheme None
Crystal is 40MHz
MAC: c0:49:ef:0a:44:c0
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
```

Figure 4.8 Flashing ESP32

Inorder to view the output .We have to open the serial monitor in the IDE.



Choose terminal: ESP-IDF Serial Monitor

Settings

Project name: ESP

Serial port: COM9

Filter Options:

Encoding: Default (ISO-8859-1)

Figure 4.9 Launching the serial monitor

4.1 FreeRTOS

FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller - although its use is not limited to microcontroller applications. A microcontroller is a small and resource constrained processor that incorporates, on a single chip, the processor itself, read only memory (ROM or Flash) to hold the program to be executed, and the random-access memory (RAM) needed by the programs it executes. Typically, the program is executed directly from the read only memory.

FreeRTOS is an open-source real-time operating system kernel that acts as the operating system for ESP-IDF applications and is integrated into ESP-IDF as a component. The FreeRTOS component in ESP-IDF contains ports of the FreeRTOS kernel for all the CPU architectures used by ESP targets (i.e., Xtensa and RISC-V). Furthermore, ESP-IDF provides different implementations of FreeRTOS in order to support SMP (Symmetric Multiprocessing) on multi-core ESP targets.

In ESP-IDF, task creation involves several steps. First, the developer needs to define the task function, which is the code that will be executed by the task. The task function must have a specific signature that conforms to the FreeRTOS API requirements, which includes a void return type and a single void* parameter. The task function should contain the desired functionality that the task is intended to perform, such as reading sensors, processing data, or controlling peripherals.

Once the task function is defined, the developer can create a task using the xTaskCreate() API provided by FreeRTOS. This API allows the developer to specify the task function, a name for the task, the stack size for the task, the priority of the task, and a handle for the created task. For example, the following code creates a task with the task function named "my_task_func", a stack size of 2048 bytes, a priority of 1 (where 0 is the highest priority), and a NULL handle:

Example : xTaskCreate(my_task_func, "My Task", 2048, NULL, 1, NULL);

Once the task is created, it will start running concurrently with other tasks in the FreeRTOS scheduler. The task will continue to run until it is explicitly deleted using the vTaskDelete() API or until it returns from its task function.

4.1.1 FreeRTOS Task States

A task can exist in one of the following states:

- **Running -** When a task is actually executing it is said to be in the Running state. It is currently utilising the processor. If the processor on which the RTOS is running only has a single core then there can only be one task in the Running state at any given time.
- **Ready -** Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.
- **Blocked -** A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. For example, if a task calls vTaskDelay() it will block (be placed into the Blocked state) until the delay period has expired - a temporal event. Tasks can also block to wait for queue, semaphore, event group, notification, or semaphore event. Tasks in the Blocked state normally

have a 'timeout' period, after which the task will be timeout, and be unblocked, even if the event the task was waiting for has not occurred.

- **Suspended -** Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, but tasks in the Suspended state do not have a time out. Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.

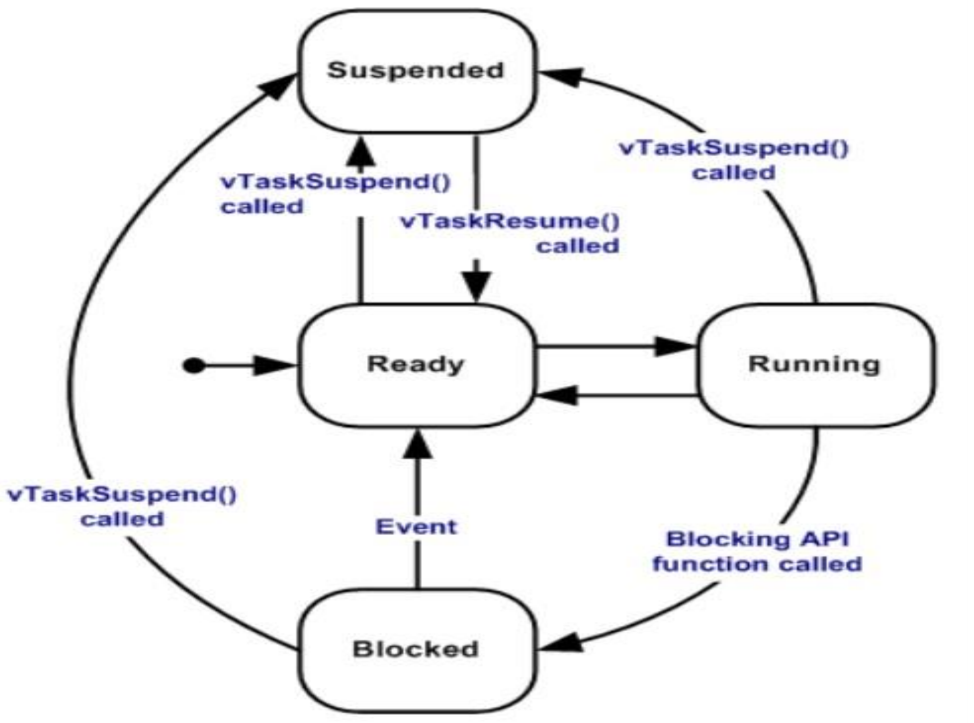


Figure 4.10 Valid Task State Transitions

4.1.2 xTaskCreatePinnedToCore Parameters

- **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).
- The FreeRTOS acts as a task scheduler for the microcontroller to perform operations in a valid state transition .

4.2 RGB LED Implementation

We will be creating a function for initializing the RGB LED settings per channel (R, G and B), GPIO for each channel and timer configuration. Creating a function for setting the colours based on duty cycle for each channel. Create specific color functions to use as application status indicators. We'll 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).

- **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 colours, by calling ledc_set_duty, then ledc_update_duty.
- The status functions call the set colour 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).

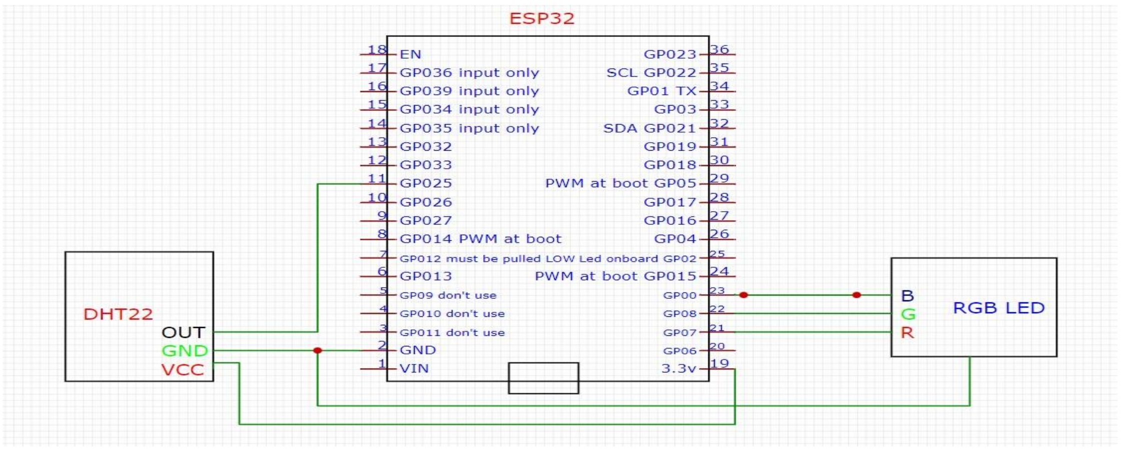


Figure 4.11 Circuit Connections

The programming is done in embedded C language. We'll program two files named rgb_led.c and rgb_led.h , one file is the source, and another file is the header file.

RGB_LED.c Program

```
#include <stdio.h>
#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);}
```

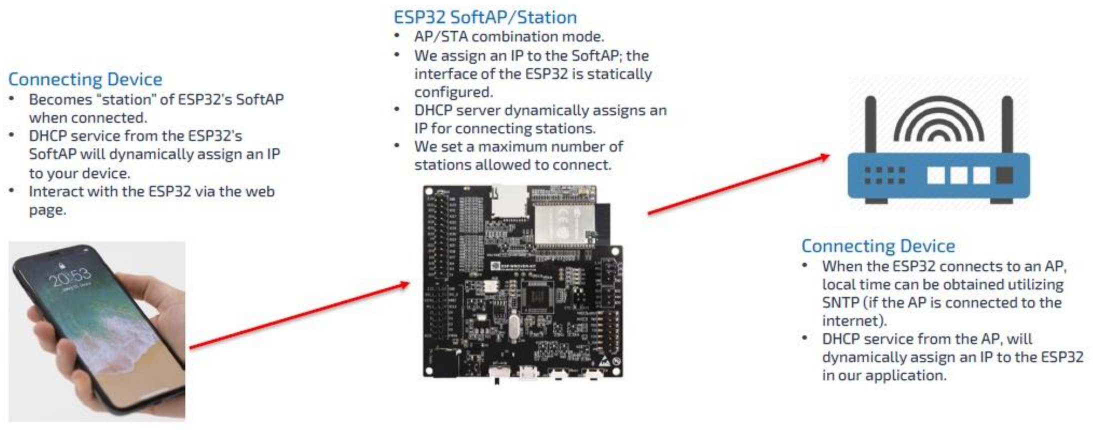
RGB_LED.h Program

```
#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;
// 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_ */
```

4.3 WiFi Application 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. The application will contain a FreeRTOS task that accepts FreeRTOS Queue messages (xQueueCreate, xQueueSend and xQueueReceive) for event coordination.



Connecting Device
• 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.
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.
Connecting Device
• 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 4.12 Simplified Overview of WiFi Application

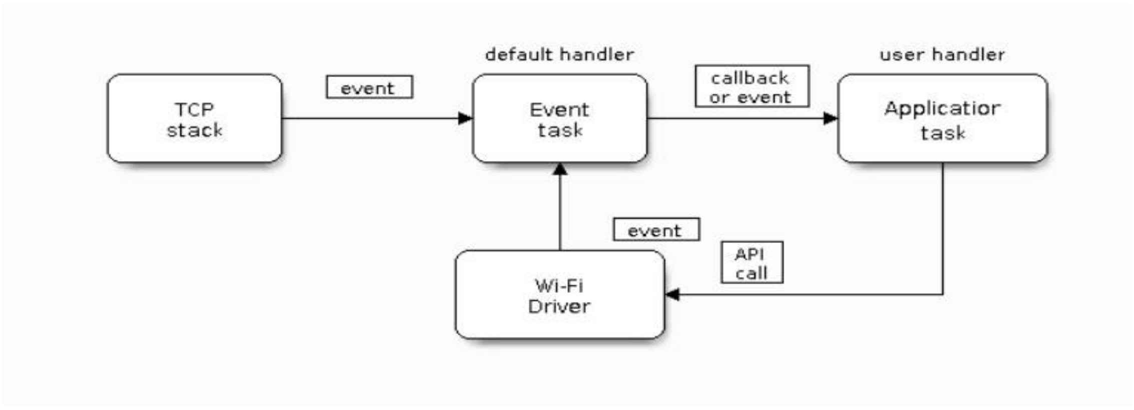


Figure 4.13 Wi-Fi Programming Model

The Wi-Fi driver can be considered a black box that knows nothing about high-layer code, such as the TCP/IP stack, application task, and event task. The application task (code) generally calls Wi-Fi driver APIs to initialize Wi-Fi and handles Wi-Fi events when necessary.

Wi-Fi event handling is based on the esp_event library. Events are sent by the Wi-Fi driver to the default event loop. Application may handle these events in callbacks registered using esp_event_handler_register(). Wi-Fi events are also handled by esp_netif component to provide a set of default behaviors.

- **Define WiFi settings -** Header file with SSID, Password, IP, Gateway, Netmask.
- **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 such as 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. Functions used esp_netif_set_ip_info, esp_netif_dhcp_start, esp_wifi_set_mode, esp_wifi_set_config, esp_wifi_set_bandwidth, esp_wifi_set_ps .
- Start WiFi (esp_wifi_start).

WIFI_APP.c Program

```
#include <esp_wifi.h>
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "freertos/task.h"
#include "esp_netif.h"
#include "esp_err.h"
#include "esp_log.h"
#include "esp_wifi.h"
#include "lwip/netdb.h"
#include "app_nvs.h"
#include <sys/socket.h>
#include "esp_http_server.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
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_disconnected_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_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");

                                   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(WIFI_EVENT, ESP_EVENT_ANY_ID,
&wifi_app_event_handler, NULL, &instance_wifi_event));
ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_EVENT, 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_RAM));
		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
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_CREDENTIALS);
		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_SERVER);
                  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_CONNECT_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();
///> to do: 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_CONNECT_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_DISCONNECT);
}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);
}
```

WIFI_APP.h Program

```
#ifndef MAIN_WIFI_APP_H_
#define MAIN_WIFI_APP_H_
#include <esp_wifi_types.h>
#include "esp_err.h"
#include "esp_netif.h"
#include "freertos/FreeRTOS.h"
// callback typedef
typedef void (*wifi_connected_event_callback_t)(void);
// 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
#define WIFI_AP_IP                      "192.168.0.1"        //AP default IP
#define WIFI_AP_GATEWAY                 "192.168.0.1"        //AP gateway
#define WIFI_AP_NETMASK                 "255.255.255.0"      //AP netmask
#define WIFI_AP_BANDWIDTH               WIFI_BW_HT20         //AP Bandwidth
#define WIFI_STA_POWER_SAVE             WIFI_PS_NONE    //Power save
#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
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
typedef struct wifi_app_queue_message
{
		wifi_app_message_e msgID;
} wifi_app_queue_message_t;

//Sends a message to the queue
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.
int8_t wifi_app_get_rssi(void);
#endif /* MAIN_WIFI_APP_H_ */
```

4.4 HTTP Server Implementation

The HTTP Server component provides an ability for running a lightweight web server on ESP32. Following are detailed steps to use the API exposed by HTTP Server.

- **httpd_start()** Creates an instance of HTTP server, allocate memory/resources for it depending upon the specified configuration and outputs a handle to the server instance. The server has both, a listening socket (TCP) for HTTP traffic, and a control socket (UDP) for control signals, which are selected in a round robin fashion in the server task loop. The task priority and stack size are configurable during server instance creation by passing httpd_config_t structure to httpd_start().
- **httpd_stop()** This stops the server with the provided handle and frees up any associated memory/resources. This is a blocking function that first signals a halt to the server task and then waits for the task to terminate. While stopping, the task will close all open connections, remove registered URI handlers and reset all session context data to empty.
- **httpd_register_uri_handler()** A URI handler is registered by passing object of type httpd_uri_t structure which has members including uri name, method type (HTTPD_GET/HTTPD_POST/HTTPD_PUT) , function pointer of type esp_err_t *handler (httpd_req_t *req) and user_ctx pointer to user context data.

ESP HTTP server has various events for which a handler can be triggered by the Event loop library when the particular event occurs. The handler has to be registered using **esp_event_handler_register().** This helps in event handling for ESP HTTP server.

- HTTP_SERVER_EVENT_ERROR
- HTTP_SERVER_EVENT_START
- HTTP_SERVER_EVENT_ON_CONNECTED
- HTTP_SERVER_EVENT_ON_HEADER
- HTTP_SERVER_EVENT_HEADERS_SENT
- HTTP_SERVER_EVENT_ON_DATA
- HTTP_SERVER_EVENT_SENT_DATA
- HTTP_SERVER_EVENT_DISCONNECTED
- HTTP_SERVER_EVENT_STOP

We’ll programming the web server and additionally we’ll be creating the web page for the end user to control the application, which can be supported by HTTP server.

HTTP_SERVER.h Program

```
#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
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_ */
```

HTTP_SERVER.c Program

```
#include "esp_timer.h"
#include "esp_wifi.h"
#include "lwip/netdb.h"
#include <sys/socket.h>
#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 "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.
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
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.
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.
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.
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.
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.
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 via the web page and handles the firmware update
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%lx\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%lx", 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.
if (flash_successful) { http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_SUCCESSFUL); } else {
http_server_monitor_send_message(HTTP_MSG_OTA_UPDATE_FAILED); }
 return ESP_OK;
}
//OTA status handler responds with the firmware update status after the OTA update
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
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
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_FROM_HTTP_SERVER);
                    free(ssid_str);
                    free(pass_str);
                    return ESP_OK;
}
//wifiConnectStatus handler updates the connection status for the web page.
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.
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.
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.
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.
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.
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();
}
```

Embed Binary Data (index.html, app.css and code.js) - Embedding Binary Data.

INDEX.html Program

```
<!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</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>
```

APP.css Program

```
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);
}
```

APP.js Program

```
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()
{
   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
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.
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"]);
		});
}
```

4.5 OTA Implementation

OTA means "Over the Air firmware update" 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

Configuration Steps and Notable ESP-IDF APIs Used :

- 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

4.6 DHT22 Implementation

DHT22.c Program

```
#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"
#include "tasks_common.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" );
        }
}
int getSignalLevel( int usTimeOut, bool state )
{
        int uSec = 0;
        while( gpio_get_level(DHTgpio)==state ) {
                if( uSec > usTimeOut )
                        return -1;
                ++uSec;
                esp_rom_delay_us(1);            // uSec delay
        }
        return uSec; }
#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 );
        esp_rom_delay_us( 3000 );
        // pull up for 25 us for a gentle asking for data
        gpio_set_level( DHTgpio, 1 );
        esp_rom_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
                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
        temperature *= -1;
        // 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;
}
// DHT22 Sensor task
static void DHT22_task(void *pvParameter)
{
        setDHTgpio(DHT_GPIO);
        printf("Starting DHT task\n\n");
        for (;;)
        {
                printf("=== Reading DHT ===\n");
                int ret = readDHT();
                errorHandler(ret);
                printf("Hum %.1f\n", getHumidity());
                printf("Tmp %.1f\n", getTemperature());
                // Wait at least 2 seconds before reading again
                vTaskDelay(4000 / portTICK_PERIOD_MS);
        }
}
void DHT22_task_start(void)
{
xTaskCreatePinnedToCore(&DHT22_task, "DHT22_task", DHT22_TASK_STACK_SIZE, NULL,
DHT22_TASK_PRIORITY, NULL, DHT22_TASK_CORE_ID);
}
```

DHT22.h Program

```
// DHT22 temperature sensor driver
#ifndef DHT22_H_
#define DHT22_H_
#define DHT_OK          0
#define DHT_CHECKSUM_ERROR   -1
#define DHT_TIMEOUT_ERROR    -2
#define DHT_GPIO             25
// Starts DHT22 sensor task
void DHT22_task_start(void);
// == function prototypes =======================================
void        setDHTgpio(int gpio);
void        errorHandler(int response);
int         readDHT();
float       getHumidity();
float       getTemperature();
int         getSignalLevel( int usTimeOut, bool state );
#endif
```

4.7 Non - Volatile Storage Implementation

NVS components save and load data from storage that is preserved between boots. NVS works best for storing many small values e.g., WiFi credentials in our case. Keys and Values - Keys are ASCII strings and values are the types (variable length binary data (blob) in our case) .

- Saving the credentials - Open storage area with a given namespace - Call nvs_open and 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_* 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 .
- 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 .
- Clearing the credentials - Similarly, we open storage area with a given namespace - Call nvs_open and 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 .

APP_NVS.c Program

```
#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;
}
```

APP_NVS.h Program

```
#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_ */
```

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.

WIFI_RESET_BUTTON.c Program

```
#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
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
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
		esp_rom_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);
}
```

WIFI_RESET_BUTTON.h Program

```
#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_ */
```

4.8 SNTP Time Synchronization implementation

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. 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.

4.8.1 Configuration Steps and Notable APIs Used

- 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. Then we’ll initialize service - Using sntp_init() .

SNTP_TIME_SYNC.c Program

```
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "lwip/apps/sntp.h"
#include "time.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_SERVICE_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", "IST-05:30:00", 1);
                                tzset();
                }}
//The SNTP time synchronization task.
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);}
```

SNTP_TIME_SYNC.h Program

```
#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.
char* sntp_time_sync_get_time(void);
#endif /* MAIN_SNTP_TIME_SYNC_H_ */
```

4.9 AWS IoT Core Implementation

AWS IoT Core services connect IoT devices to AWS IoT services and other AWS services. AWS IoT Core includes the device gateway and the message broker, which connect and process messages between your IoT devices and the cloud.

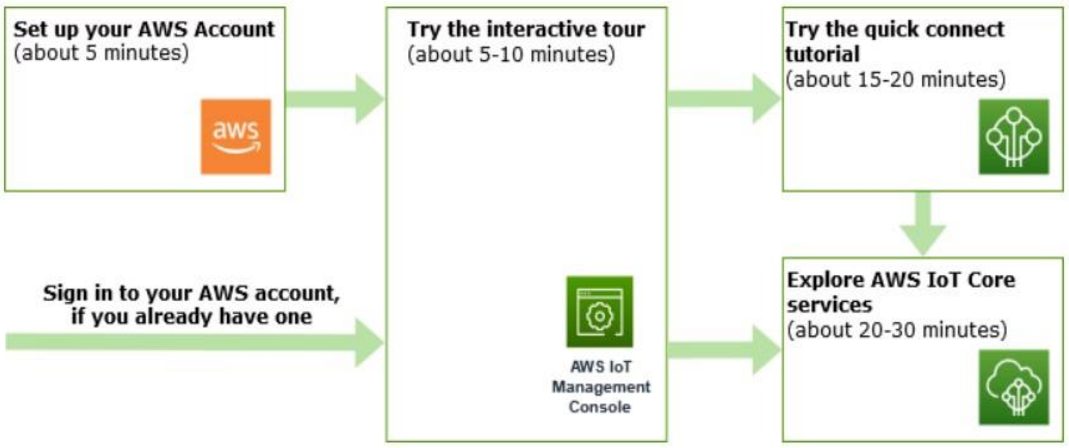

Set up your AWS Account
(about 5 minutes)
aws
Try the interactive tour
(about 5-10 minutes)
Try the quick connect tutorial
(about 15-20 minutes)
Sign in to your AWS account, if you already have one
AWS IoT Management Console
Explore AWS IoT Core services
(about 20-30 minutes)

Figure 4.14 Getting started with AWS IoT Core

4.9.1 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 networks.
- 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.
- 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.

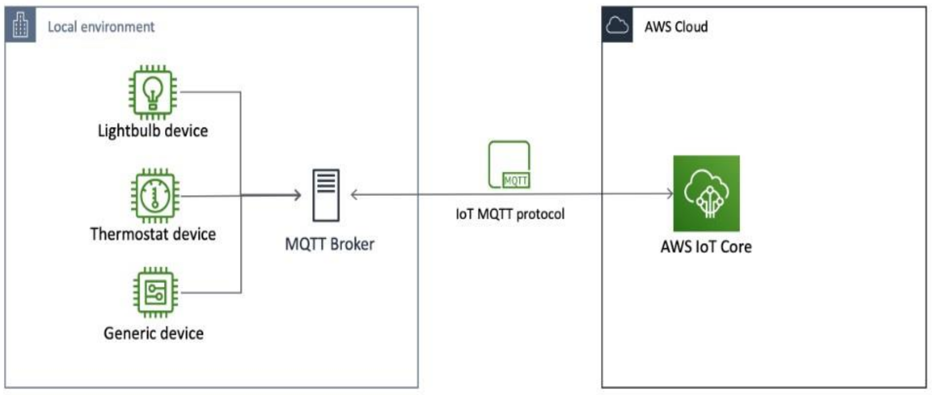


Figure 4.15 MQTT Protocol

4.9.2 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.

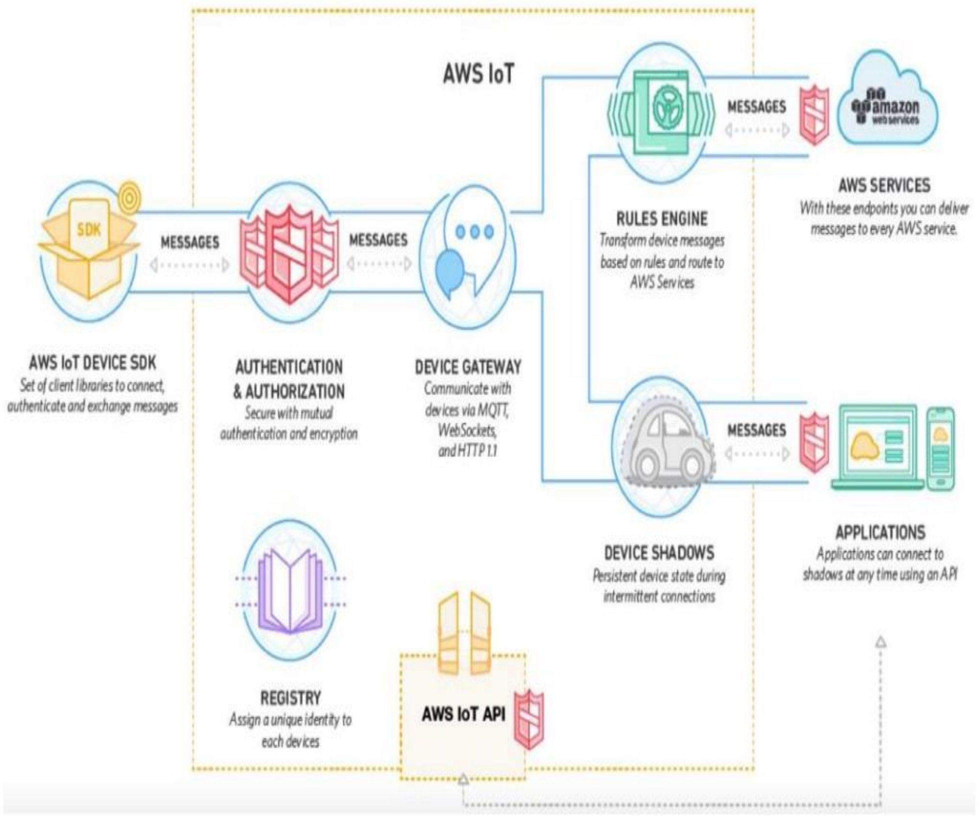

AWS IoT
AWS IoT DEVICE SDK
Set of client libraries to connect, authenticate and exchange messages
MESSAGES
AUTHENTICATION & AUTHORIZATION
Secure with mutual authentication and encryption
MESSAGES
DEVICE GATEWAY
Communicate with devices via MQTT, WebSockets, and HTTP 1.1
RULES ENGINE
Transform device messages based on rules and route to AWS Services
MESSAGES
AWS SERVICES
With these endpoints you can deliver messages to every AWS service.
DEVICE SHADOWS
Persistent device state during intermittent connections
MESSAGES
APPLICATIONS
Applications can connect to shadows at any time using an API
REGISTRY
Assign a unique identity to each devices
AWS IoT API

Figure 4.16 AWS IoT Device Authentication

4.9.3 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.

4.9.4 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.

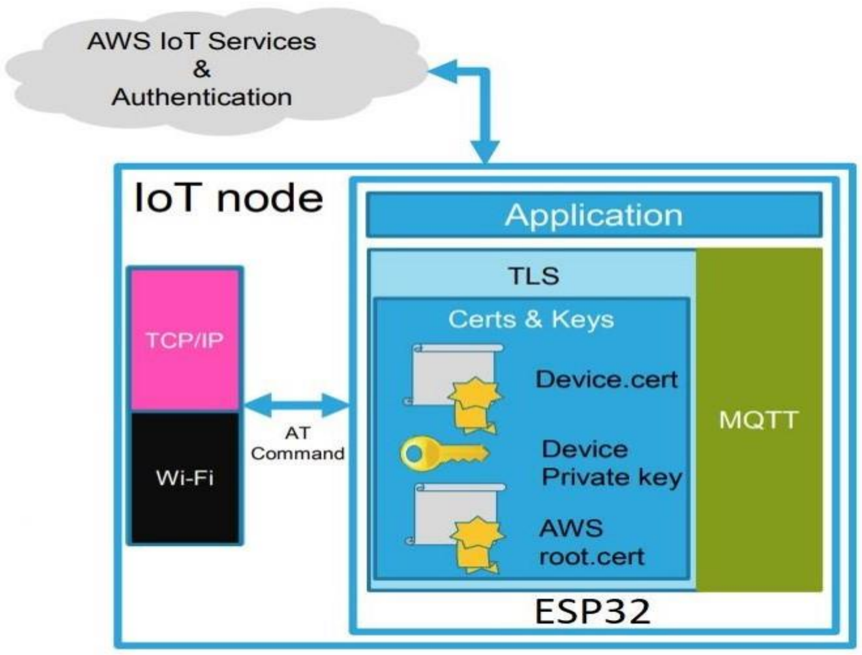


Figure 4.17 AWS Security Overview

4.9.5 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.
- 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.
- JSON (JavaScript Object Notation) is an open standard lightweight data-interchange format.
- As a text document, it is easy for users to read and write, and for machines to parse and generate.

4.9.6 AWS IoT Configuration Steps

- Clone ESP AWS IoT and update the CMakeLists.txt file under the project directory.
- Add template files “aws_iot.c” and “aws_iot.h” and update CMakeLists.txt file under ‘main’.
- Create a “Thing” in AWS.
- Create a policy and attach it to the device (“Thing”) certificate.

- Generate certificates, public key, and private key using AWS IoT’s certificate authority.
- Add the certificates and private key as embedded components.
- Update the source code to accommodate the ‘aws_iot_task’.
- Adjust the partition table to accommodate the increased application size.
- Adjust the sdkconfig to include the ‘Device Data Endpoint’ from your AWS account.
- Subscribe and publish data to and from the AWS Dashboard.

AWS_IoT.c Program

```
#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_LOGI(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_PERIOD_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 : %ld ", "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", pcTaskGetName(NULL),
uxTaskGetStackHighWaterMark(NULL));
        vTaskDelay(1000 / portTICK_PERIOD_MS);
        sprintf(cPayload, "%s : %d ", "WiFi RSSI", wifi_app_get_rssi());
        paramsQOS0.payloadLen = strlen(cPayload);
        rc = aws_iot_mqtt_publish(&client, TOPIC, TOPIC_LEN, ¶msQOS0);
sprintf(cPayload, "%s : %.1f, %s : %.1f", "Temperature", getTemperature(), "Humidity", getHumidity());
        paramsQOS1.payloadLen = strlen(cPayload);
        rc = aws_iot_mqtt_publish(&client, TOPIC, TOPIC_LEN, ¶msQOS1);
        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);
                }
}
```

AWS_IoT.h Program

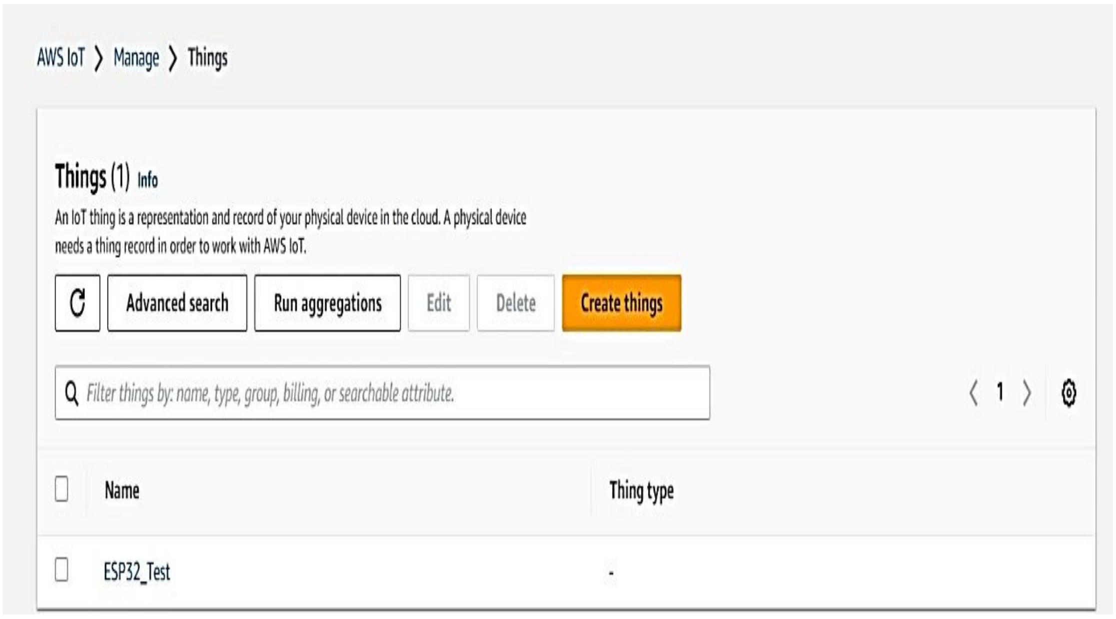
```
#ifndef MAIN_AWS_IOT_H_
#define MAIN_AWS_IOT_H_
#define CONFIG_AWS_EXAMPLE_CLIENT_ID "ESP32_Test"
//Starts AWS IoT task.
void aws_iot_start(void);
#endif /* MAIN_AWS_IOT_H_ */
```

MAIN.c Program

```
#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";
void wifi_application_connected_events(void)
{
                ESP_LOGI(TAG, "WiFi Application Connected!!");
                sntp_time_sync_task_start();
                aws_iot_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);
}
```

TASK_COMMON.h Program

```
#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
// AWS IoT Task
#define AWS_IOT_TASK_STACK_SIZE                           9216
#define AWS_IOT_TASK_PRIORITY                             6
#define AWS_IOT_TASK_CORE_ID                              1 #endif /*
MAIN_TASKS_COMMON_H_ */
```

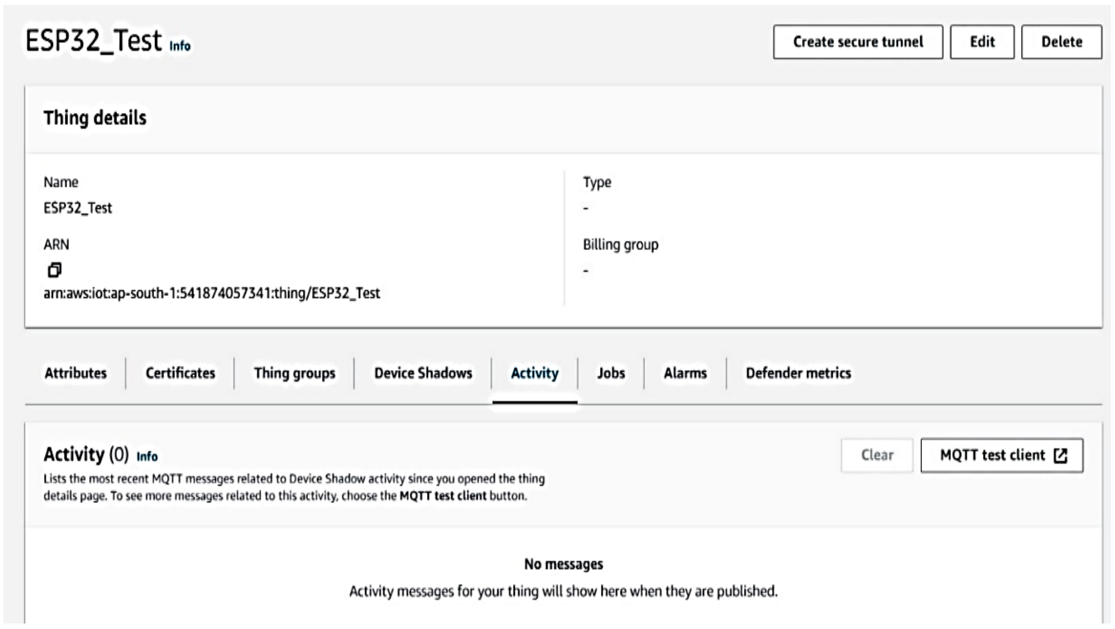

AWS IoT ❯ Manage ❯ Things
Info
Advanced search
Run aggregations
Edit
Delete
Create things
Filter things by: name, type, group, billing, or searchable attribute.
‹ 1 ›

Things (1)

An IoT thing is a representation and record of your physical device in the cloud. A physical device needs a thing record in order to work with AWS IoT.

Name	Thing type
ESP32_Test	-

Figure 4.18 Create a Thing in AWS management console


ESP32_Test Info

Thing details

Name
ESP32_Test

ARN
arn:aws:iot:ap-south-1:541874057341:thing/ESP32_Test

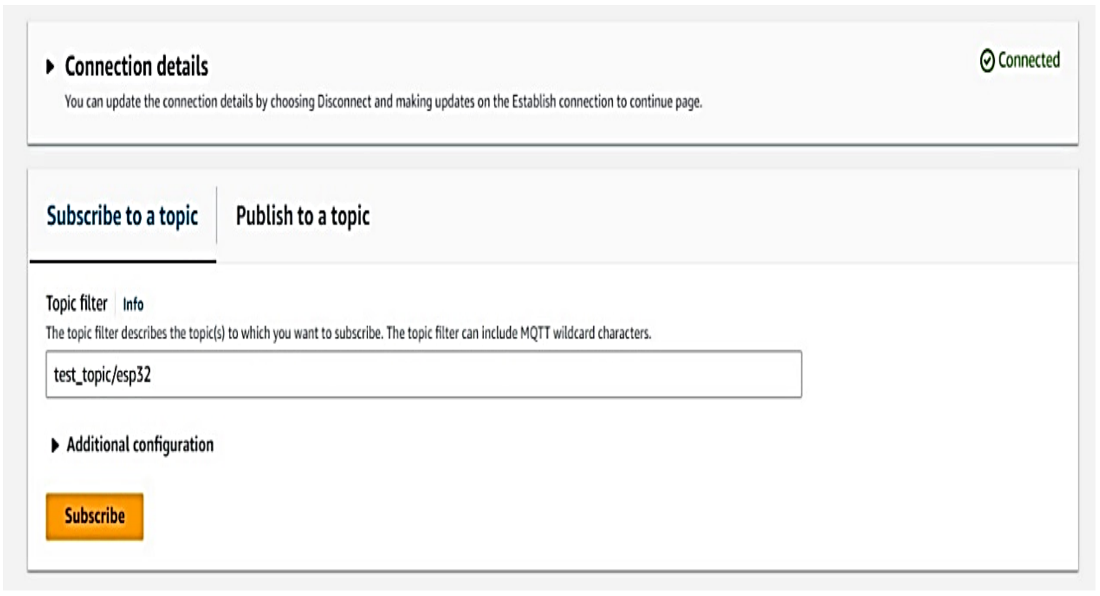
Type
-

Billing group
-

Activity (0) Info
Lists the most recent MQTT messages related to Device Shadow activity since you opened the thing details page. To see more messages related to this activity, choose the **MQTT test client** button.

No messages
Activity messages for your thing will show here when they are published.

Figure 4.19 Select the MQTT test client


Connection details
You can update the connection details by choosing Disconnect and making updates on the Establish connection to continue page.

Topic filter Info
The topic filter describes the topic(s) to which you want to subscribe. The topic filter can include MQTT wildcard characters.

test_topic/esp32

Figure 4.20 Subscribing and Publishing the data via Topic Filter

CMakeLists File

```
idf_component_register(SRCS main.c rgb_led.c wifi_app.c http_server.c DHT22.c app_nvs.c wifi_reset_button.c
sntp_time_sync.c aws_iot.c
                    INCLUDE_DIRS "."
                    EMBED_FILES webpage/app.css webpage/app.js webpage/favicon.ico webpage/index.html
webpage/jquery-3.3.1.min.js)s
target_add_binary_data(${COMPONENT_TARGET} "certs/aws_root_ca_pem" TEXT)
target_add_binary_data(${COMPONENT_TARGET} "certs/certificate_pem_crt" TEXT)
target_add_binary_data(${COMPONENT_TARGET} "certs/private_pem_key" TEXT)
```

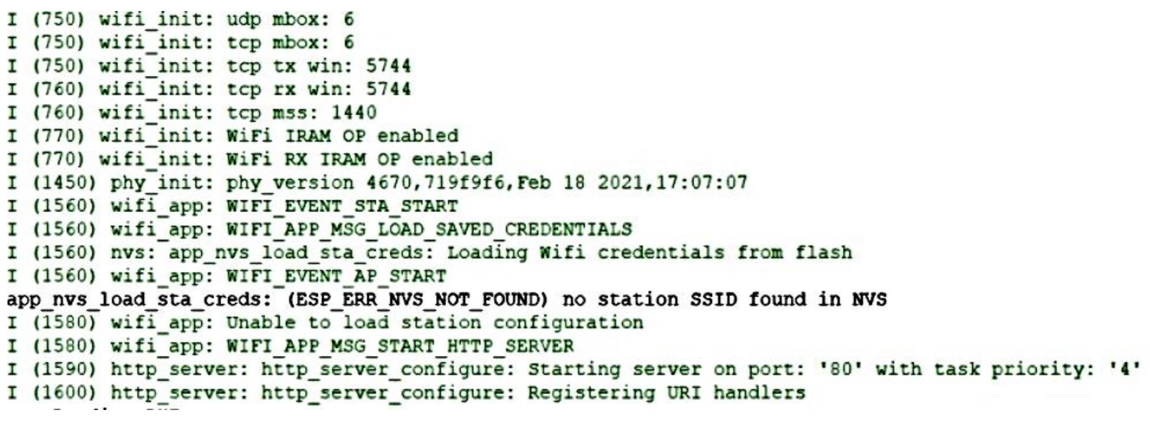
- After programming update the CMakelists file in the directory to compiling the whole application.

CHAPTER V
RESULTS

5.1 WiFi Intialization

After flashing and Opening the serial monitor, The message appear on the monitor is "WIFI EVENT STA START" which means, the WiFi event station has started its network for connecting to the Access point.

The WiFi App function tries to load the saved creditionals and checks if any connection credentials were present or saved in the nvs operation. If no connection was present or saved, it prints the message no station SSID found in NVS. The HTTP server starts the server for connecting with the webpage.



```
I (750) wifi_init: udp mbox: 6
I (750) wifi_init: tcp mbox: 6
I (750) wifi_init: tcp tx win: 5744
I (760) wifi_init: tcp rx win: 5744
I (760) wifi_init: tcp mss: 1440
I (770) wifi_init: WiFi IRAM OP enabled
I (770) wifi_init: WiFi RX IRAM OP enabled
I (1450) phy_init: phy_version 4670,719f9f6,Feb 18 2021,17:07:07
I (1560) wifi_app: WIFI_EVENT_STA_START
I (1560) wifi_app: WIFI_APP_MSG_LOAD_SAVED_CREDENTIALS
I (1560) nvs: app_nvs_load_sta_creds: Loading Wifi credentials from flash
I (1560) wifi_app: WIFI_EVENT_AP_START
app_nvs_load_sta_creds: (ESP_ERR_NVS_NOT_FOUND) no station SSID found in NVS
I (1580) wifi_app: Unable to load station configuration
I (1580) wifi_app: WIFI_APP_MSG_START_HTTP_SERVER
I (1590) http_server: http_server_configure: Starting server on port: '80' with task priority: '4'
I (1600) http_server: http_server_configure: Registering URI handlers
```

Figure 5.1 WiFi and HTTP server started

5.2 Access Point Connection

Check the networks nearby , and Access Point called "ESP32_AP" as we configured the SSID name in the WiFi header file. After connecting to the Access Point , the wifi event station gets connected to the Access point and the DHCP server which is a lightweight IP assigned to a client IP "192.168.0.1" , which has to be entered in the browser.

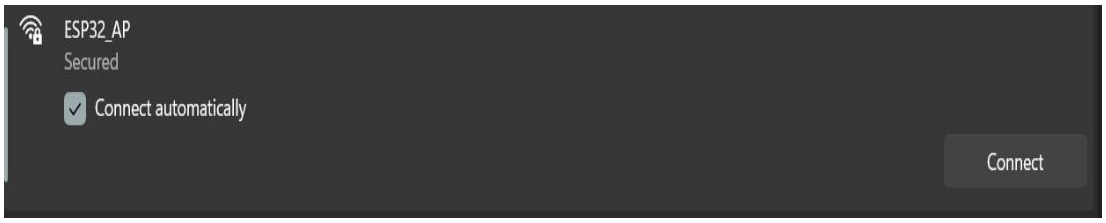
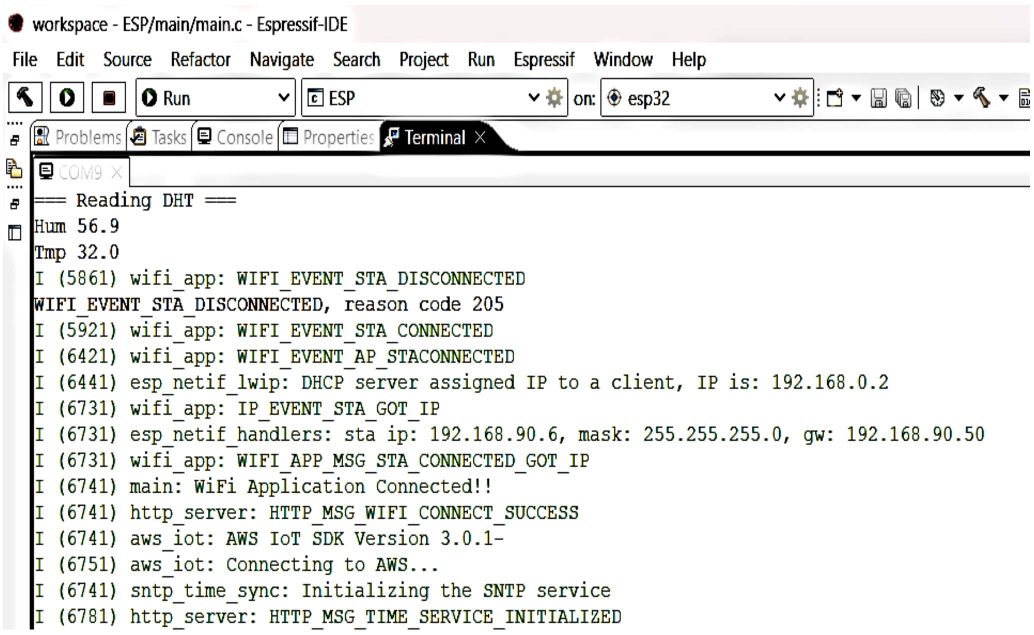


Figure 5.2 Connect to the ESP32's Access Point


workspace - ESP/main/main.c - Espressif-IDE
File Edit Source Refactor Navigate Search Project Run Espressif Window Help
Run
ESP
on: esp32
Problems Tasks Console Properties Terminal
COM9

```
=== Reading DHT ===
Hum 56.9
Tmp 32.0
I (5861) wifi_app: WIFI_EVENT_STA_DISCONNECTED
WIFI_EVENT_STA_DISCONNECTED, reason code 205
I (5921) wifi_app: WIFI_EVENT_STA_CONNECTED
I (6421) wifi_app: WIFI_EVENT_AP_STACONNECTED
I (6441) esp_netif_lwip: DHCP server assigned IP to a client, IP is: 192.168.0.2
I (6731) wifi_app: IP_EVENT_STA_GOT_IP
I (6731) esp_netif_handlers: sta ip: 192.168.90.6, mask: 255.255.255.0, gw: 192.168.90.50
I (6731) wifi_app: WIFI_APP_MSG_STA_CONNECTED_GOT_IP
I (6741) main: WiFi Application Connected!!
I (6741) http_server: HTTP_MSG_WIFI_CONNECT_SUCCESS
I (6741) aws_iot: AWS IoT SDK Version 3.0.1-
I (6751) aws_iot: Connecting to AWS...
I (6741) sntp_time_sync: Initializing the SNTP service
I (6781) http_server: HTTP_MSG_TIME_SERVICE_INITIALIZED
```

Figure 5.3 WiFi connected status

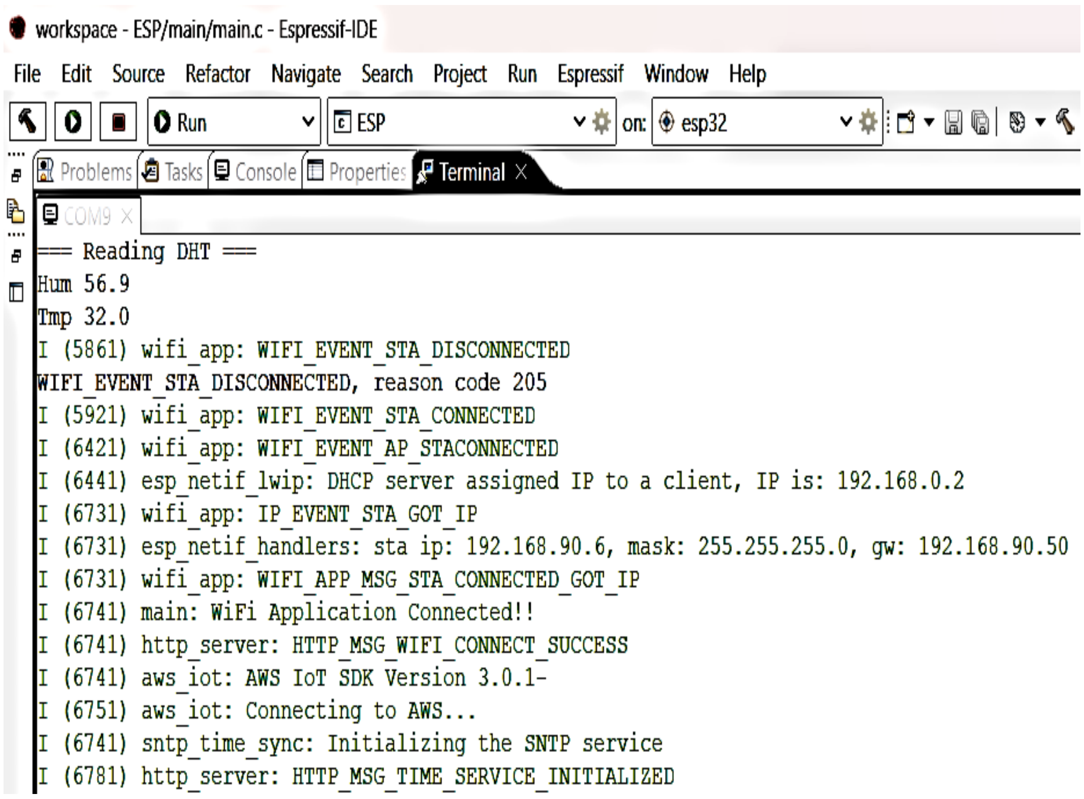
Figure 5.4 Enter the IP address

5.3 Webpage Requests

After entering the IP address, the webpage requests for file such as ,html, .css and .js files for creating the webpage and loading .

```
I (6440) wifi_app: WIFI_EVENT_AP_STACONNECTED
I (6570) esp_netif_lwip: DHCP server assigned IP to a client, IP is: 192.168.0.2
I (6780) http_server: index.html requested
I (7840) http_server: index.html requested
I (8010) http_server: Jquery requested
I (8150) http_server: app.css requested
I (8170) http_server: app.js requested
I (8290) http_server: /apSSID.json requested
I (8290) http_server: OTAstatus requested
I (8360) http_server: /wifiConnectInfo.json requested
I (8370) http_server: favicon.ico requested
```

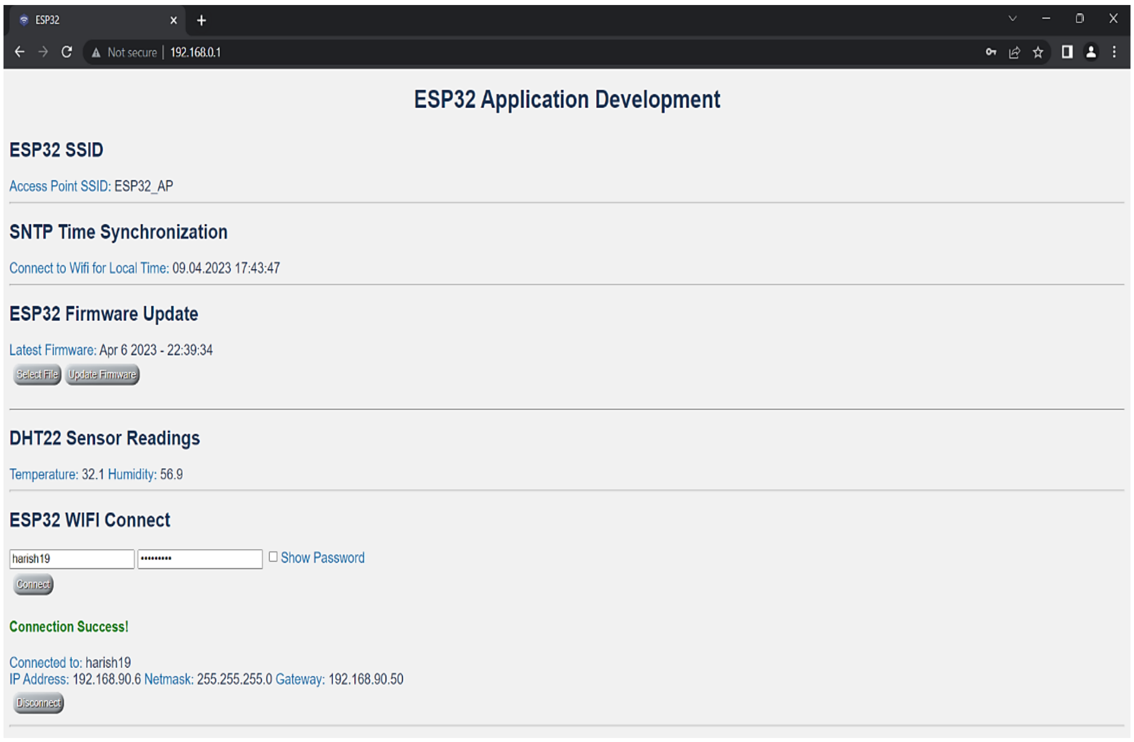
Figure 5.5 Server requests for loading the webpage


workspace - ESP/main/main.c - Espressif-IDE
File Edit Source Refactor Navigate Search Project Run Espressif Window Help
Run
ESP
on: esp32
Problems Tasks Console Properties Terminal
COM9

```
=== Reading DHT ===
Hum 56.9
Tmp 32.0
I (5861) wifi_app: WIFI_EVENT_STA_DISCONNECTED
WIFI_EVENT_STA_DISCONNECTED, reason code 205
I (5921) wifi_app: WIFI_EVENT_STA_CONNECTED
I (6421) wifi_app: WIFI_EVENT_AP_STACONNECTED
I (6441) esp_netif_lwip: DHCP server assigned IP to a client, IP is: 192.168.0.2
I (6731) wifi_app: IP_EVENT_STA_GOT_IP
I (6731) esp_netif_handlers: sta ip: 192.168.90.6, mask: 255.255.255.0, gw: 192.168.90.50
I (6731) wifi_app: WIFI_APP_MSG_STA_CONNECTED_GOT_IP
I (6741) main: WiFi Application Connected!!
I (6741) http_server: HTTP_MSG_WIFI_CONNECT_SUCCESS
I (6741) aws_iot: AWS IoT SDK Version 3.0.1-
I (6751) aws_iot: Connecting to AWS...
I (6741) sntp_time_sync: Initializing the SNTP service
I (6781) http_server: HTTP_MSG_TIME_SERVICE_INITIALIZED
```

Figure 5.6 After connecting to the Internet

Inorder to Connect to Internet , we need to connect an external network for accessing the internet through the webpage and making it available for it to make the process.



ESP32 Application Development

ESP32 SSID

Access Point SSID: ESP32_AP

SNTP Time Synchronization

Connect to Wifi for Local Time: 09.04.2023 17:43:47

ESP32 Firmware Update

Latest Firmware: Apr 6 2023 - 22:39:34

DHT22 Sensor Readings

Temperature: 32.1 Humidity: 56.9

ESP32 WIFI Connect

harish19 ••••••••

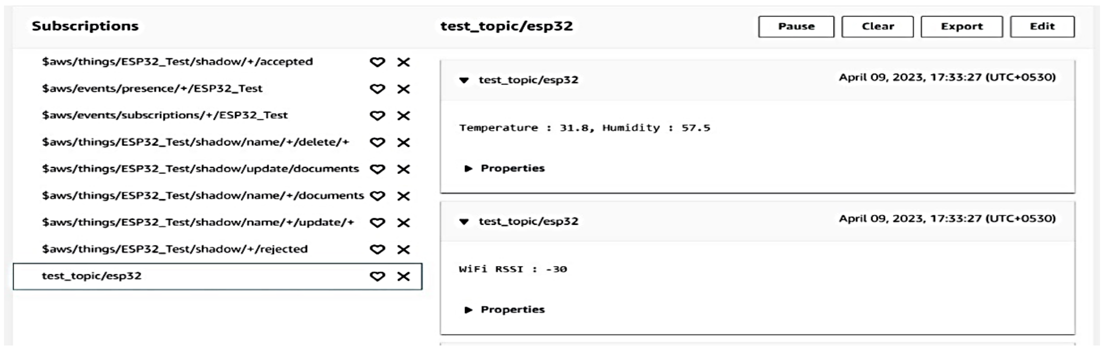
Connection Success!

Connected to: harish19
IP Address: 192.168.90.6 Netmask: 255.255.255.0 Gateway: 192.168.90.50

Figure 5.7 Application Webpage Connected through ESP32

5.4 Application Features

After loading the webpage, we can see the multiple features present and incorporated in the application as ESP32 SSID, SNTP Time Synchronization, Firmware update which can be updated by just connecting through the webpage and loading the .bin file for updating the system , we can see the DHT22 sensor readings which are going to update every 2 seconds, and External internet connection . By connecting we can know the IP address of the network , gateway, and netmask.

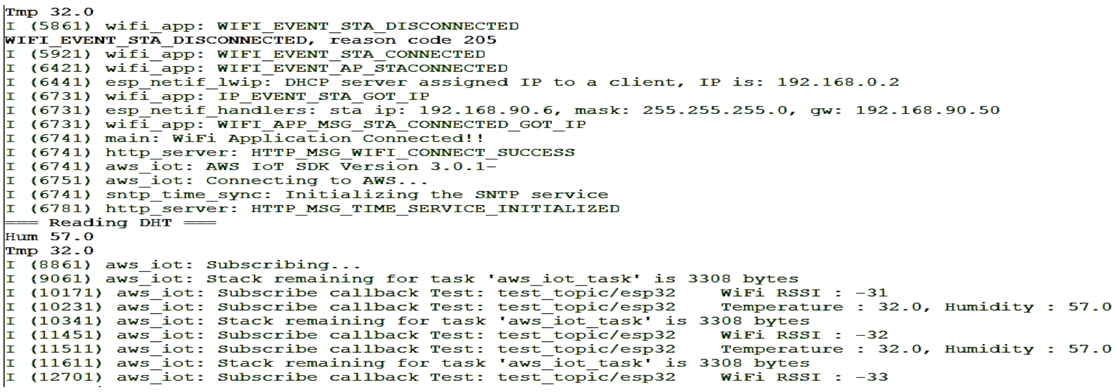

Subscriptions
test_topic/esp32
Pause
Clear
Export
Edit
$aws/things/ESP32_Test/shadow/+/accepted
$aws/events/presence/+/ESP32_Test
$aws/events/subscriptions/+/ESP32_Test
$aws/things/ESP32_Test/shadow/name/+/delete/+
$aws/things/ESP32_Test/shadow/update/documents
$aws/things/ESP32_Test/shadow/name/+/documents
$aws/things/ESP32_Test/shadow/name/+/update/+
$aws/things/ESP32_Test/shadow/+/rejected
test_topic/esp32
▼ test_topic/esp32
April 09, 2023, 17:33:27 (UTC+0530)
▶ Properties
▼ test_topic/esp32
April 09, 2023, 17:33:27 (UTC+0530)
▶ Properties

```
Temperature : 31.8, Humidity : 57.5
```

```
WiFi RSSI : -30
```

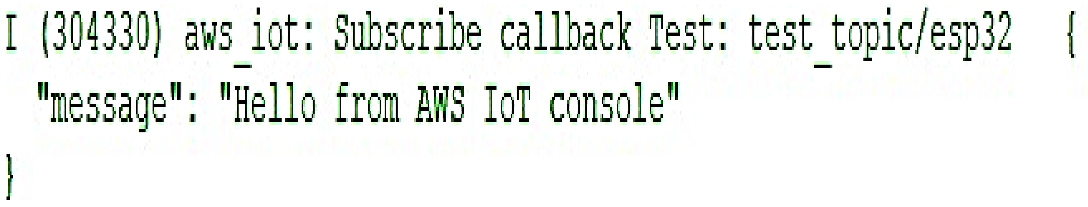
Figure 5.8 Subscribed data in the AWS console

As you can see, the Subscribed data which are the sensor readings and wifi received signal strength are shown in the AWS Console after getting connected to the Internet via external network through the webpage we created.



```
Tmp 32.0
I (5861) wifi_app: WIFI_EVENT_STA_DISCONNECTED
WIFI_EVENT_STA_DISCONNECTED, reason code 205
I (5921) wifi_app: WIFI_EVENT_STA_CONNECTED
I (6421) wifi_app: WIFI_EVENT_AP_STACONNECTED
I (6441) esp_netif_lwip: DHCP server assigned IP to a client, IP is: 192.168.0.2
I (6731) wifi_app: IP_EVENT_STA_GOT_IP
I (6731) esp_netif_handlers: sta ip: 192.168.90.6, mask: 255.255.255.0, gw: 192.168.90.50
I (6731) wifi_app: WIFI_APP_MSG_STA_CONNECTED_GOT_IP
I (6741) main: WiFi Application Connected!!
I (6741) http_server: HTTP_MSG_WIFI_CONNECT_SUCCESS
I (6741) aws_iot: AWS IoT SDK Version 3.0.1-
I (6751) aws_iot: Connecting to AWS...
I (6741) sntp_time_sync: Initializing the SNTP service
I (6781) http_server: HTTP_MSG_TIME_SERVICE_INITIALIZED
=== Reading DHT ===
Hum 57.0
Tmp 32.0
I (8861) aws_iot: Subscribing...
I (9061) aws_iot: Stack remaining for task 'aws_iot_task' is 3308 bytes
I (10171) aws_iot: Subscribe callback Test: test_topic/esp32    WiFi RSSI : -31
I (10231) aws_iot: Subscribe callback Test: test_topic/esp32    Temperature : 32.0, Humidity : 57.0
I (10341) aws_iot: Stack remaining for task 'aws_iot_task' is 3308 bytes
I (11451) aws_iot: Subscribe callback Test: test_topic/esp32    WiFi RSSI : -32
I (11511) aws_iot: Subscribe callback Test: test_topic/esp32    Temperature : 32.0, Humidity : 57.0
I (11611) aws_iot: Stack remaining for task 'aws_iot_task' is 3308 bytes
I (12701) aws_iot: Subscribe callback Test: test_topic/esp32    WiFi RSSI : -33
```

Figure 5.9 The data has been transferred through callback function



```
I (304330) aws_iot: Subscribe callback Test: test_topic/esp32    {
  "message": "Hello from AWS IoT console"
}
```

Figure 5.10 Published Message from the AWS console

We can see the message printed as Connecting to AWS for subscribing and publishing the data. Both the operations are performed via callback function for the AWS Console.

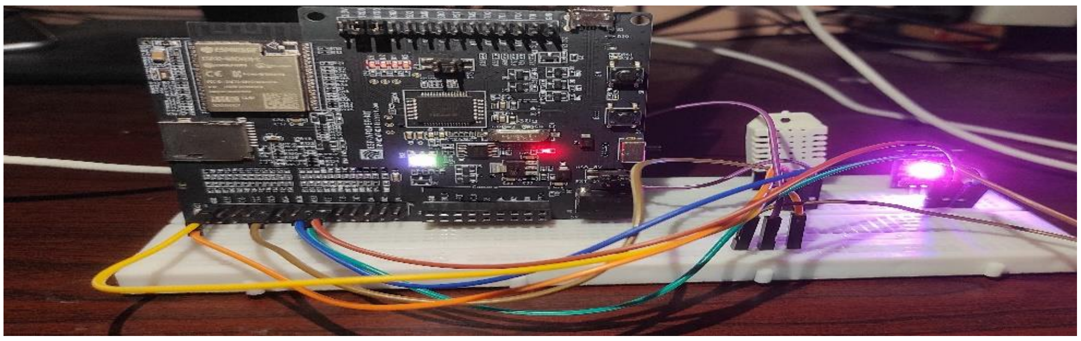


Figure 5.11 WiFi Event Started Indication

The RGB LED is used for indicating the application status at each and every operation performed by the ESP32. So, when the WiFi event gets started , pink light will be glown.

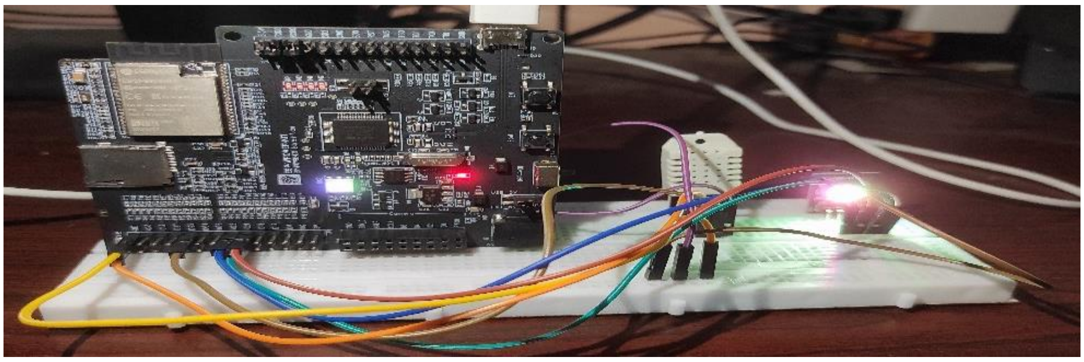


Figure 5.12 HTTP Server Started Indication

For HTTP server operation , green color light will be glown and For the internet connection as we connect through the Application webpage, blue colour light will be glown.

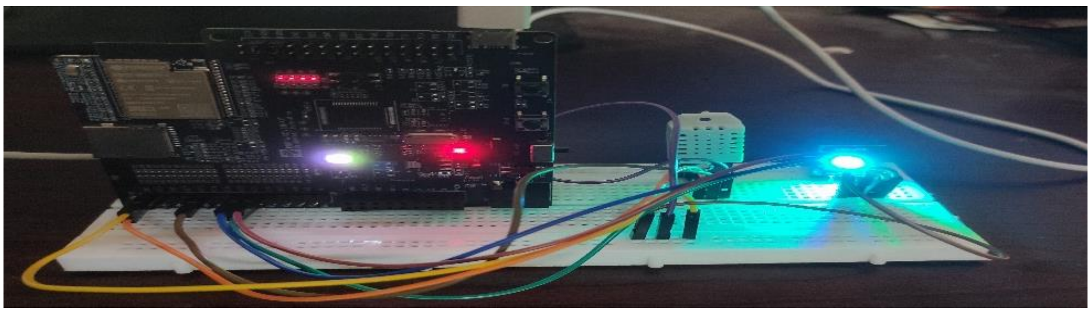


Figure 5.13 Internet Connected Indication

CHAPTER VI

CONCLUSION & FUTURE ENHANCEMENTS

In conclusion, the project has successfully highlighted the potential of using ESP32 and AWS IoT for a wide range of IoT applications, such as remote sensing, monitoring, and control of devices. The project has demonstrated the feasibility of building an IoT application using ESP32 and AWS IoT and has laid the foundation for future enhancements.

Moving forward, there are several possible enhancements that could be considered for further improving the project.

- Enhancing the security of the system could be a key focus for future iterations of the project. This could involve implementing end-to-end encryption for data transmission, utilizing secure authentication methods such as certificate-based authentication, and implementing additional security measures to protect against data breaches and unauthorized access.
- The project was implemented with a single ESP32 device, but for large-scale deployments, scalability could be a crucial consideration. Future enhancements could involve implementing a device provisioning process to handle multiple devices, managing device certificates efficiently, and optimizing data transmission to handle a large number of devices concurrently.
- Developing a user-friendly front-end interface could greatly enhance the usability of the IoT application. This could involve creating a web or mobile-based interface using AWS services such as AWS Amplify or AWS AppSync, allowing users to easily configure and manage devices, view sensor data, and control devices remotely.
- Utilizing advanced data analytics and visualization tools could provide valuable insights from the collected sensor data. Future enhancements could involve leveraging AWS IoT Analytics or Amazon Quick Sight for advanced data analysis, real-time monitoring, and visualization of sensor data, enabling data-driven decision making.

CHAPTER VII

REFERENCES

[1] Silva, J., & Johnson, M. (2019). Implementing an IoT Application using ESP32 and AWS IoT. International Journal of Internet of Things, 4(1), 45-58. DOI: 10.1016/j.ijiot.2019.01.002

[2] Smith, A., & Brown, B. (2020). ESP32-based IoT Application for Remote Sensing using AWS IoT Core. Proceedings of the IEEE Conference on IoT Technologies, 123-135. DOI: 10.1109/CIoT.2020.00012.

[3] Chen, L., & Wu, J. (2018). Integrating ESP32 with AWS IoT for Smart Home Applications. Journal of Sensors and Actuators, 12(2), 234-245.

[4] Patel, R., & Shah, K. (2017). AWS IoT-Based Internet of Things Application using ESP32. Proceedings of the International Conference on Computer Networks and IoT, 789-798 . DOI : 10.1109.

[5] Gonzalez, S., & Martinez, D. (2019). Building a WiFi-based IoT System with ESP32 and AWS IoT for Environmental Monitoring. Journal of Environmental Sensing and Monitoring, 8(3), 321-335. DOI: 10.1016/j.envsense.2019.07.001.

[6] Lee, C., & Kim, D. (2020). Secure Communication for ESP32-Based IoT Application with AWS IoT Core. Proceedings of the International Conference on Internet of Things and Smart Cities, 456-468. DOI: 10.1109/ICIOTSC.2020.9242031.

[7] Zhang, H., & Wang, X. (2018). ESP32-Based IoT Application for Industrial Automation using AWS IoT Core. Journal of Industrial Automation and Control Systems, 15(4), 567-578. DOI: 10.1016/j.jiacs.2018.09.003.

[8] Gupta, S., & Jain, A. (2017). AWS IoT-Based Internet of Things Application using ESP32 for Agriculture Monitoring. Proceedings of the International Conference on Advances in IoT and Big Data, 678-689. DOI: 10.1109/AIOTBD.2017.8462902.

[9] Wang, L., & Li, J. (2019). ESP32-Based IoT Application for Smart Energy Management using AWS IoT Core. Journal of Sustainable Energy Systems, 6(1), 23-36. DOI: 10.1016/j.susensys.2019.02.002.