ESP32 Wi-Fi: Connection, Scanning, Web Server, and IoT Applications

1. Introduction to Wi-Fi

1.1. What is Wi-Fi?

Wi-Fi (Wireless Fidelity) is a revolutionary wireless networking technology that enables electronic devices such as computers, smartphones, tablets, IoT sensors, and embedded systems to establish internet connectivity and inter-device communication without the constraints of physical cables. This technology fundamentally transforms how devices interact by utilizing radio wave transmission over local area networks (LANs), commonly referred to as Wireless Local Area Networks (WLANs).The essence of Wi-Fi lies in its ability to provide seamless connectivity and mobility, allowing users and devices to access network resources and internet services from virtually anywhere within the wireless coverage area. This mobility aspect is particularly crucial for modern IoT applications, smart home systems, and embedded projects where flexible connectivity is essential.Wi-Fi operates through a sophisticated infrastructure where wireless access points or routers serve as intermediary bridges between wired Ethernet networks and wireless clients. These access points perform multiple critical functions including managing wireless traffic, coordinating communication between connected devices, and implementing security features such as encryption and authentication to protect data transmission.

1.2. IEEE 802.11 Standards Overview

The entire Wi-Fi ecosystem is built upon the IEEE 802.11 family of standards, which establishes the technical specifications and protocols governing wireless communication between devices. These standards ensure reliability, interoperability, and consistent performance across different manufacturers and device types.Over the years, multiple versions of Wi-Fi standards have been developed and deployed, each introducing significant improvements in various aspects of wireless communication:

Standard	Frequency Band(s)	Maximum Speed	Key Features	Year Introduced
802.11b	2.4 GHz	Up to 11 Mbps	Early Wi-Fi standard, longer range, slower speeds	1999
802.11a	5 GHz	Up to 54 Mbps	Less interference than 2.4 GHz, shorter range	1999
802.11g	2.4 GHz	Up to 54 Mbps	Backwards compatible with 802.11b	2003
802.11n	2.4 GHz & 5 GHz	Up to 600 Mbps	MIMO technology, dual-band support	2009
802.11ac	5 GHz	Up to 3.46 Gbps	Improved MIMO, wider channels, beamforming	2013
802.11ax (Wi-Fi 6)	2.4 GHz & 5 GHz	Up to 9.6 Gbps	OFDMA, improved efficiency, better performance in dense environments	2019
802.11be (Wi-Fi 7)	2.4 GHz, 5 GHz & 6 GHz	Up to 30 Gbps (projected)	Enhanced throughput, lower latency, multi-link operation	Upcoming

These standards progressively improve data transmission speeds, coverage, and network efficiency, while addressing interference and security.

1.3. Core Principles of Wi-Fi Communication

Each successive Wi-Fi standard has introduced substantial improvements in multiple domains:

- Data Transmission Speeds: Earlier standards like 802.11b operated primarily on the 2.4 GHz frequency band offering speeds up to 11 Mbps, while modern standards such as 802.11ac and 802.11ax utilize the 5 GHz band (and in some cases the 6 GHz band) to provide significantly faster data rates and enhanced performance in crowded environments.
- Frequency Band Utilization: The progression from single-band to dual-band and now tri-band support allows devices to optimize performance by selecting the most appropriate frequency band based on current network conditions.

- Coverage Range and Network Efficiency: Advanced techniques like beamforming, MIMO (Multiple Input Multiple Output) technology, and improved antenna designs have enhanced both signal coverage and overall network efficiency.
- Dense Environment Performance: Modern standards like Wi-Fi 6 introduce technologies such as OFDMA (Orthogonal Frequency Division Multiple Access) that significantly improve performance in environments with many connected devices.

1.4. Applications of Wi-Fi in IoT

Wi-Fi plays a crucial role in the Internet of Things (IoT) ecosystem by providing reliable wireless connectivity for diverse smart devices and systems:

- Smart Home Devices: Wi-Fi enables home automation gadgets, sensors, cameras, and appliances to communicate and be controlled remotely.
- Industrial Automation: Manufacturing and industrial environments use Wi-Fi to connect sensors, machinery, and monitoring systems for improved operational efficiency and safety.
- Healthcare: Medical devices use Wi-Fi for real-time patient monitoring, data collection, and remote diagnostics.
- Enterprise and Education: Wi-Fi supports flexible wireless networks, facilitating mobile computing, online learning, and collaboration.
- Retail and Transportation: Retailers use Wi-Fi for customer engagement and analytics, while transportation hubs offer Wi-Fi for passengers.
- Media Streaming: Devices stream high-definition audio and video over Wi-Fi supporting rich multimedia experiences.
- Emerging Applications: Advanced uses include Wi-Fi sensing for motion/gesture recognition and AI-driven Wi-Fi management for optimized performance.

Within IoT projects using ESP32 modules, Wi-Fi connectivity typically operates on the 2.4 GHz band supporting the 802.11 b/g/n protocols, allowing devices to send sensor data, control actuators, and host web servers for remote monitoring and control.

2. ESP32 and Wi-Fi

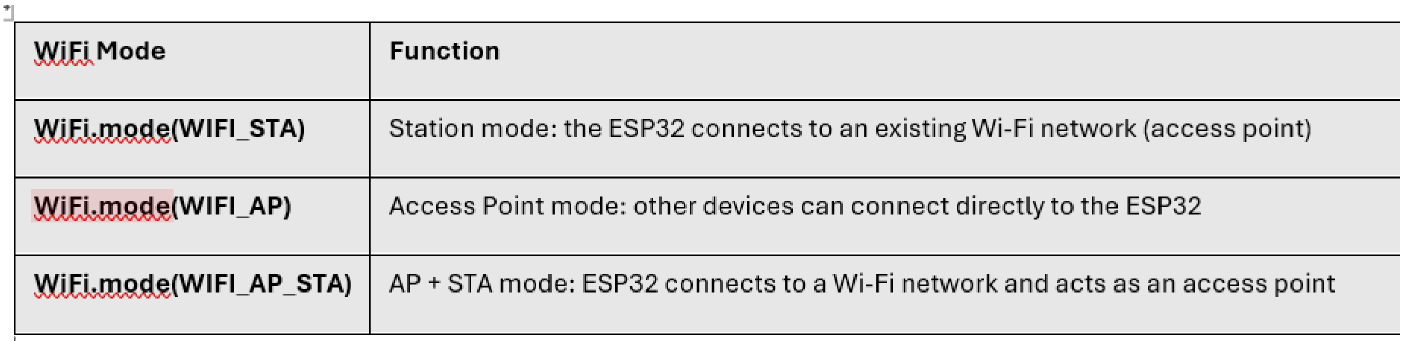
2.1. ESP32 Wi-Fi Capabilities

The ESP32 integrates a powerful dual-core microcontroller with a full IEEE 802.11 b/g/n Wi-Fi radio, operating exclusively in the 2.4 GHz band. It delivers up to 150 Mbps of throughput, enabling responsive data exchange for sensor telemetry, control commands, and lightweight web-server tasks. The on-chip radio supports adjustable transmit power up to 20.5 dBm, allowing developers to trade off range and energy consumption—critical for battery-powered deployments. Antenna diversity support via an external RF switch further enhances link reliability by dynamically selecting the optimal antenna path in environments with multipath interference.

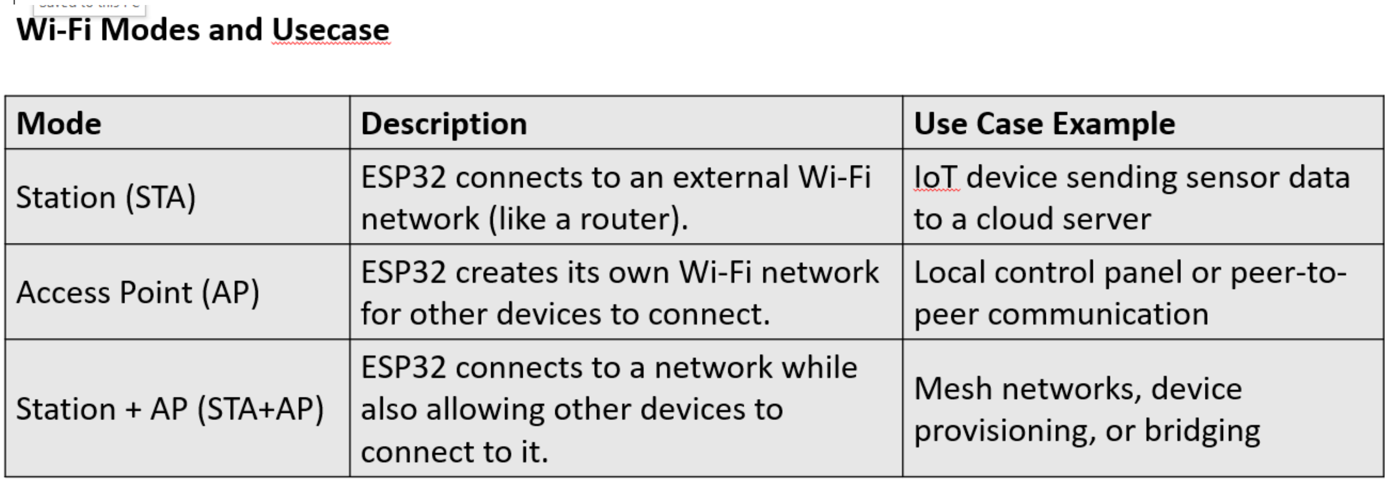
2.2. Wi-Fi Modes

The ESP32 supports three distinct operational modes, each tailored to specific networking scenarios:

- Station Mode (WIFI_STA) Acts as a Wi-Fi client connecting to an existing access point. The ESP32 obtains an IP address via DHCP, enabling full internet access for cloud integration, OTA updates, and API interactions.
- Access Point Mode (WIFI_AP) Creates a soft-AP network that other devices join directly. Useful for local provisioning and peer-to-peer control, though without internet connectivity since the ESP32 is not linked to an external router.
- Combined Mode (WIFI_AP_STA) Operates simultaneously as client and access point. Ideal for onboarding new devices via a captive-portal AP while maintaining a WAN connection for cloud services or bridging local devices to the internet.

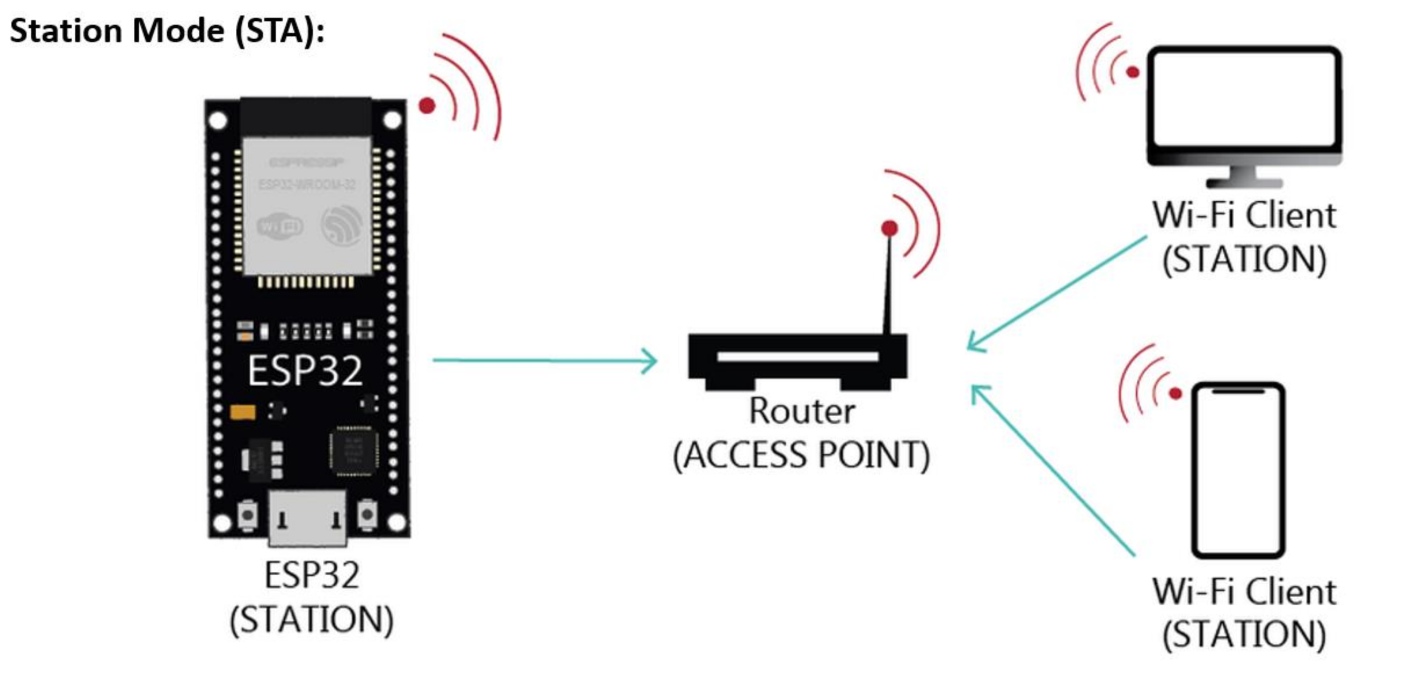


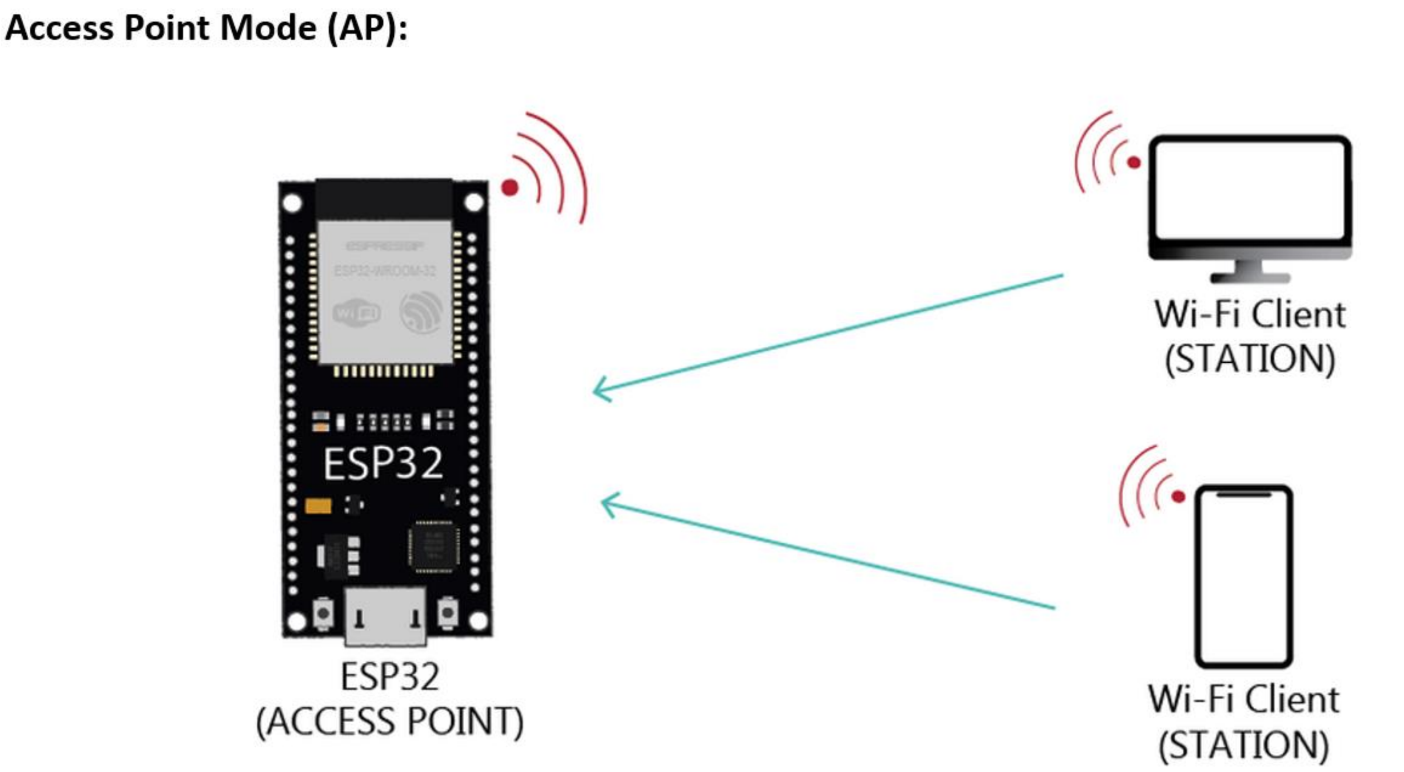
WiFi Mode	Function
WiFi.mode(WIFI_STA)	Station mode: the ESP32 connects to an existing Wi-Fi network (access point)
WiFi.mode(WIFI_AP)	Access Point mode: other devices can connect directly to the ESP32
WiFi.mode(WIFI_AP_STA)	AP + STA mode: ESP32 connects to a Wi-Fi network and acts as an access point



Wi-Fi Modes and Usecase

Mode	Description	Use Case Example
Station (STA)	ESP32 connects to an external Wi-Fi network (like a router).	IoT device sending sensor data to a cloud server
Access Point (AP)	ESP32 creates its own Wi-Fi network for other devices to connect.	Local control panel or peer-to-peer communication
Station + AP (STA+AP)	ESP32 connects to a network while also allowing other devices to connect to it.	Mesh networks, device provisioning, or bridging





2.3. Wi-Fi Connection Status and Events

Robust applications require real-time awareness of network state. The ESP32's WiFi.status() API returns status codes such as WL_CONNECTED, WL_DISCONNECTED, and WL_CONNECT_FAILED, enabling conditional logic for reconnection or fallback routines. An event-driven framework signals granular lifecycle events:

Event	**Description**
ARDUINO_EVENT_WIFI_READY	Wi-Fi driver initialized

ARDUINO_EVENT_WIFI_SCAN_DONE	Network scan completed
ARDUINO_EVENT_WIFI_STA_CONNECTED	Station linked to AP
ARDUINO_EVENT_WIFI_STA_DISCONNECTED	Station lost AP link
ARDUINO_EVENT_WIFI_STA_GOT_IP	Station obtained IP address
ARDUINO_EVENT_WIFI_AP_STACONNECTED	Client joined soft-AP
ARDUINO_EVENT_WIFI_AP_STADISCONNECTED	Client left soft-AP

By handling these events, firmware can automatically retry connections, switch modes, or signal status to application layers for seamless reliability.

2.4. Security Considerations for Wi-Fi IoT Projects

Secure Wi-Fi deployment is paramount in IoT environments:

- Encryption: Use WPA2-PSK or WPA3-SAE where supported to protect data in transit.
- Authentication: Implement unique credentials for each device and disable default SSIDs.
- Network Segmentation: Isolate IoT devices on a dedicated VLAN or guest network to limit lateral movement if compromised.
- Firmware Updates: Enable secure OTA updates with code signing to patch vulnerabilities promptly.
- Certificate Management: Where possible, leverage TLS certificates for cloud and peer-to-peer communication instead of unencrypted channels.

Adhering to these best practices prevents unauthorized access, eavesdropping, and mitigates attack surfaces in Wi-Fi-enabled IoT systems.

3. Wi-Fi Connection with ESP32

3.1. Steps for Establishing Wi-Fi Connection

- Initialize the Wi-Fi Interface
  Use WiFi.begin(ssid, password) to start the station. This triggers the ESP32's internal Wi-Fi driver to scan for the specified SSID and begin association.
- Monitor Connection Progress
  Poll WiFi.status() in a loop until it equals WL_CONNECTED. Implement a timeout (for example, 10 seconds) to avoid blocking indefinitely:

```
unsigned long start = millis();
while (WiFi.status() != WL_CONNECTED && millis() - start < 10000) {
  delay(500);
  Serial.print(".");
}
```

- Confirm DHCP Assignment
  After WL_CONNECTED, call WiFi.localIP(). A valid non-zero IP confirms successful DHCP lease from the router.
- Validate Internet Reachability
  Perform a lightweight HTTP GET or ICMP ping to a reliable external host (e.g., “www.google.com”) to ensure packets traverse beyond the LAN.
- Handle Connection Failures
  If WiFi.status() returns an error such as WL_CONNECT_FAILED or the timeout expires, invoke WiFi.disconnect(), wait a few seconds, then retry WiFi.begin().

3.2. Debugging with WiFi.status

The WiFi.status() function provides granular connection states:

- WL_IDLE_STATUS: No active connection attempts
- WL_NO_SSID_AVAIL: Target SSID not found in scan results
- WL_SCAN_COMPLETED: Network scan finished without connect call
- WL_CONNECTED: Successfully associated and authenticated
- WL_CONNECT_FAILED: Authentication, encryption, or DHCP error
- WL_CONNECTION_LOST: Link dropped after initial success
- WL_DISCONNECTED: Explicit disconnect or network outage

Use a switch statement on these codes to log precise diagnostics, trigger reconnection logic, and update UI indicators (LED blink patterns or status displays).

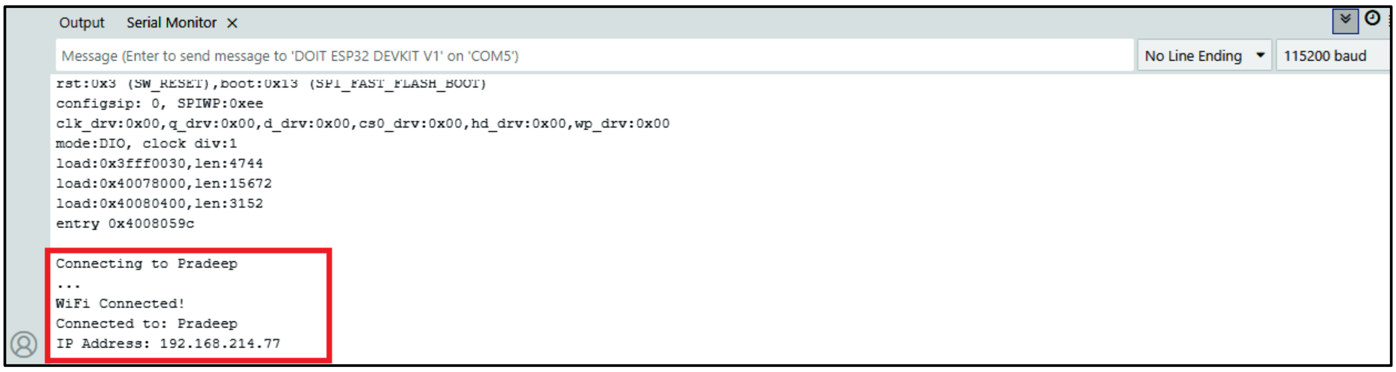
3.3. Code and Output Explanation

Complete Code:

```
#include <WiFi.h>  // Include the WiFi library for ESP32

// Replace with your WiFi credentials
const char* ssid = "Your SSID";          // Your WiFi network name
const char* password = "Your Password"; // Your WiFi password

void setup() {
  Serial.begin(115200);  // Initialize serial communication at 115200 baud
  delay(1000);           // Wait for serial to initialize

  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);  // Start connecting to WiFi

  // Wait until the ESP32 is connected
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");  // Print dots on serial monitor while waiting
  }
  Serial.println();
  Serial.println("WiFi Connected!");          // Connected successfully
  Serial.print("Connected to: ");
  Serial.println(WiFi.SSID());                  // Print connected network name
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());               // Print ESP32 IP address
}

void loop() {
  // Empty loop - Add repeated code here if needed
}
```

Line-by-Line Explanation:

- #include <WiFi.h>: Imports the ESP32 Wi-Fi library providing network functions.
- const char* ssid = "Your SSID";: Stores the Wi-Fi network name.
- const char* password = "Your Password";: Stores the Wi-Fi password.
- Serial.begin(115200);: Starts serial communication at 115200 baud rate for debugging output on the serial monitor.
- delay(1000);: Waits one second to ensure the serial port is ready.
- Serial.print and Serial.println: Display connection status messages on the serial monitor for user feedback.
- WiFi.begin(ssid, password);: Initiates the connection process to the specified Wi-Fi network.
- while (WiFi.status() != WL_CONNECTED): Loops until the ESP32 successfully connects to the network.
- delay(500); Serial.print(".");: Waits half a second and prints a dot each time to show connection progress.
- Upon successful connection, Serial.println("WiFi Connected!") confirms device connection.
- WiFi.SSID() retrieves and prints the name of the connected Wi-Fi network.
- WiFi.localIP() retrieves and prints the IP address assigned to the ESP32.
- The loop() function is empty here but is a placeholder for repeated code after connection.


Output
Serial Monitor
Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on 'COM5')
No Line Ending
115200 baud

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:4744
load:0x40078000,len:15672
load:0x40080400,len:3152
entry 0x4008059c

Connecting to Pradeep
...
WiFi Connected!
Connected to: Pradeep
IP Address: 192.168.214.77
```

The output on the Serial Monitor indicates the step-by-step progress of the ESP32 attempting to connect to the specified Wi-Fi network. Initially, it prints the name of the network it is trying to join. During the connection process, dots are printed repeatedly, each representing a short wait period and showing that ESP32 is actively trying to establish the connection. Once the ESP32 successfully connects, it prints a confirmation message "WiFi Connected!", followed by the name (SSID) of the connected Wi-Fi network. Finally, it displays the IP address assigned to the ESP32 by the router, which is essential for network communication and accessing other services. This output helps users verify both the connection attempt and success, and confirms that the device is now ready for network-based operations.

4. Wi-Fi Scanning on ESP32

4.1. Concept of Wi-Fi Scanning

Wi-Fi scanning is the process by which the ESP32's radio actively searches for available wireless networks within range. During a scan, the module cycles through all supported channels in the 2.4 GHz band, listening for beacon frames broadcast by access points. Each beacon contains essential network information—SSID, supported data rates, encryption methods, and signal strength indicators—that enables the ESP32 to build a list of candidate networks for user selection or automated connection routines.

4.2. Functions for Scanning Networks

The Arduino-style API offers two primary functions:

- WiFi.scanNetworks()
  Initiates a full active scan across all channels. The call blocks until the scan completes and returns the total number of networks found.
- WiFi.scanComplete()
  Nonblocking alternative that returns the number of networks found if a prior WiFi.scanNetworks() call has finished, or –2 if still in progress and –1 on error.

These functions provide flexibility for both synchronous and asynchronous scanning workflows, accommodating applications that require continuous network monitoring without stalling the main loop.

4.3. Displaying SSID RSSI and Encryption Type

Once the scan results are available, iterate over each network index to retrieve:

- SSID: Network name as a String via WiFi.SSID(i).
- RSSI: Received Signal Strength Indicator in dBm via WiFi.RSSI(i), representing link quality.
- Encryption Type: Security mode via WiFi.encryptionType(i), returning an enum (WEP, WPA_PSK, WPA2_PSK, WPA_WPA2_PSK, or OPEN).

By formatting these values into a table or console output, developers can visualize network landscape and make informed decisions—choosing the strongest open network or a known secure SSID for reliable connectivity.

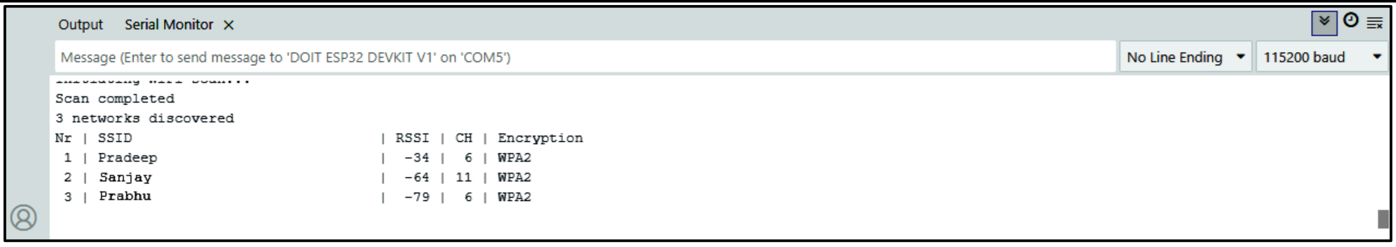
4.4. Code and Output Explanation

Complete Code:

```
#include "WiFi.h"

void setup() {
  Serial.begin(115200);

  // Set WiFi to station mode and disconnect from any previous connections
  WiFi.mode(WIFI_STA);
  WiFi.disconnect();

  delay(100);
  Serial.println("Setup completed");
}

void loop() {
  Serial.println("Scanning for networks...");

  // Perform WiFi scan
  int networkCount = WiFi.scanNetworks();

  Serial.println("Scan completed");

  if (networkCount == 0) {
    Serial.println("No networks found");
  } else {
    Serial.print(networkCount);
    Serial.println(" networks found:");

    // Display network details in table format
    Serial.println("Nr | SSID                              | RSSI | CH | Encryption");

    for (int i = 0; i < networkCount; i++) {
      Serial.printf("%2d", i + 1);
      Serial.print(" | ");
      Serial.printf("%-32.32s", WiFi.SSID(i).c_str());
      Serial.print(" | ");
      Serial.printf("%4d", WiFi.RSSI(i));
      Serial.print(" | ");
      Serial.printf("%2d", WiFi.channel(i));
      Serial.print(" | ");

      // Print encryption type
      switch (WiFi.encryptionType(i)) {
        case WIFI_AUTH_OPEN: Serial.print("Open"); break;
        case WIFI_AUTH_WEP: Serial.print("WEP"); break;
        case WIFI_AUTH_WPA_PSK: Serial.print("WPA"); break;
        case WIFI_AUTH_WPA2_PSK: Serial.print("WPA2"); break;
        case WIFI_AUTH_WPA_WPA2_PSK: Serial.print("WPA+WPA2"); break;
        case WIFI_AUTH_WPA2_ENTERPRISE: Serial.print("WPA2-EAP"); break;
        case WIFI_AUTH_WPA3_PSK: Serial.print("WPA3"); break;
        case WIFI_AUTH_WPA2_WPA3_PSK: Serial.print("WPA2+WPA3"); break;
        case WIFI_AUTH_WAPI_PSK: Serial.print("WAPI"); break;
        default: Serial.print("Unknown");
      }
      Serial.println();
    }
  }

  // Clear scan results to free memory
  WiFi.scanDelete();

  // Wait before scanning again
  delay(5000);
}
```

Line-by-Line Explanation:

- #include "WiFi.h": Imports the Wi-Fi library essential for network functions.
- Serial.begin(115200);: Starts serial communication for output on the Serial Monitor at 115200 baud rate.
- WiFi.mode(WIFI_STA);: Sets the ESP32 to Station mode for scanning networks.
- WiFi.disconnect();: Ensures disconnection from any previous Wi-Fi connection before scanning.
- delay(100);: Short delay to stabilize the Wi-Fi stack before scanning.
- Inside loop(), WiFi.scanNetworks(); starts scanning and returns the count of detected networks.
- If no networks are found, a message is printed, otherwise total networks found is shown.
- Network details are shown in a tabulated format with columns: Number, SSID (network name), RSSI (signal strength), Channel, and Encryption type.
- Serial.printf is used for formatted output ensuring data aligns neatly in columns for readability.
- The switch statement interprets encryption codes and prints human-readable security types.
- WiFi.scanDelete(); frees memory used by the scan, preventing memory leaks during repeated scans.
- delay(5000); pauses for 5 seconds before rescanning to avoid continuous network flooding.



Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on 'COM5')

No Line Ending

115200 baud

```
Scan completed
3 networks discovered
Nr | SSID                             | RSSI | CH | Encryption
 1 | Pradeep                          |  -34 |  6 | WPA2
 2 | Sanjay                           |  -64 | 11 | WPA2
 3 | Prabhu                           |  -79 |  6 | WPA2
```

The output on the Serial Monitor displays the process and results of the ESP32 scanning for available Wi-Fi networks. It starts by confirming that the setup is complete, then indicates that a scan is underway. Once the scan finishes, it shows the total number of networks detected. If no networks are found, a relevant message is displayed. Otherwise, the output presents a neatly formatted table listing each detected network with its assigned number, SSID (network name), signal strength indicated by RSSI (Received Signal Strength Indicator), the Wi-Fi channel in use, and the type of encryption securing the network, such as Open, WEP, WPA, WPA2, or WPA3. This detailed output allows users to visually assess the wireless environment, making it easier to select an appropriate network for connection based on signal quality and security considerations. The information also helps in debugging network connectivity issues and optimizing IoT device setups using the ESP32.

5. ESP32 Web Server

5.1. Working Principle of a Web Server on ESP32

An ESP32 web server runs an HTTP listener on the board, using its TCP/IP stack to accept incoming connections on port 80 (or a custom port). When a client (browser or HTTP client) issues a request, the ESP32 parses the request line and headers, maps the requested URI to handler functions in firmware, and generates an HTTP response. Static resources (HTML, CSS, JavaScript) can be stored in SPIFFS or PROGMEM, while dynamic content (sensor readings, actuator states) is generated at runtime. The lightweight nature of the ESP32's dual-core architecture ensures that network handling and application logic run concurrently without blocking.

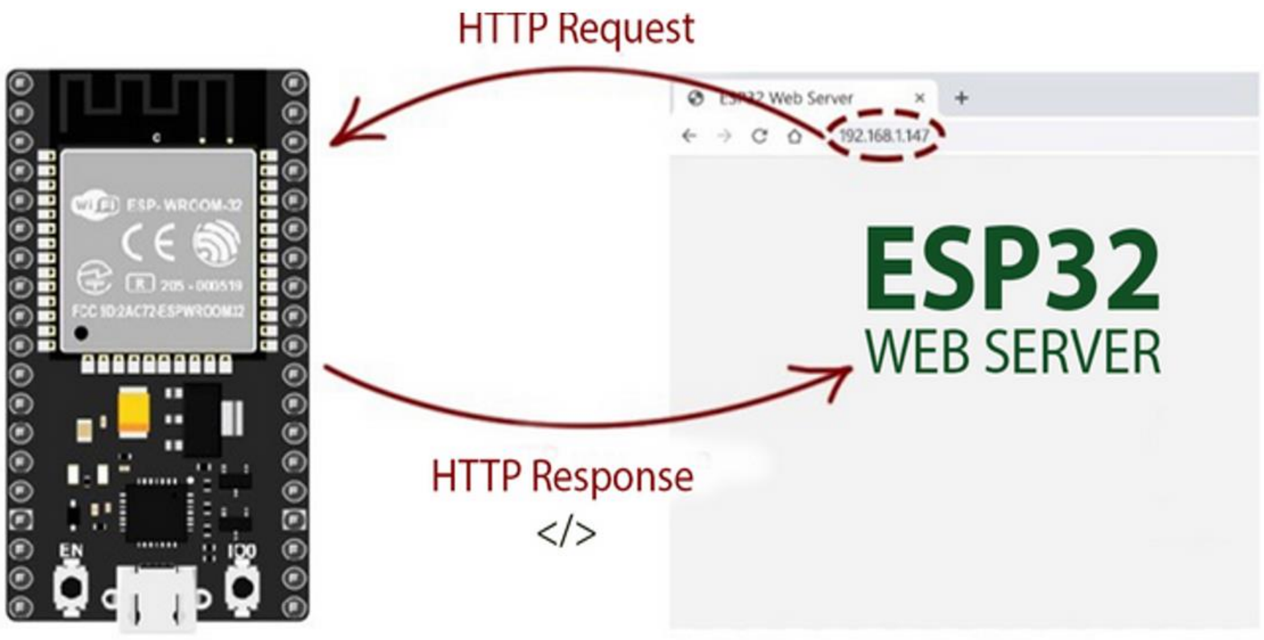
5.2. HTTP Communication

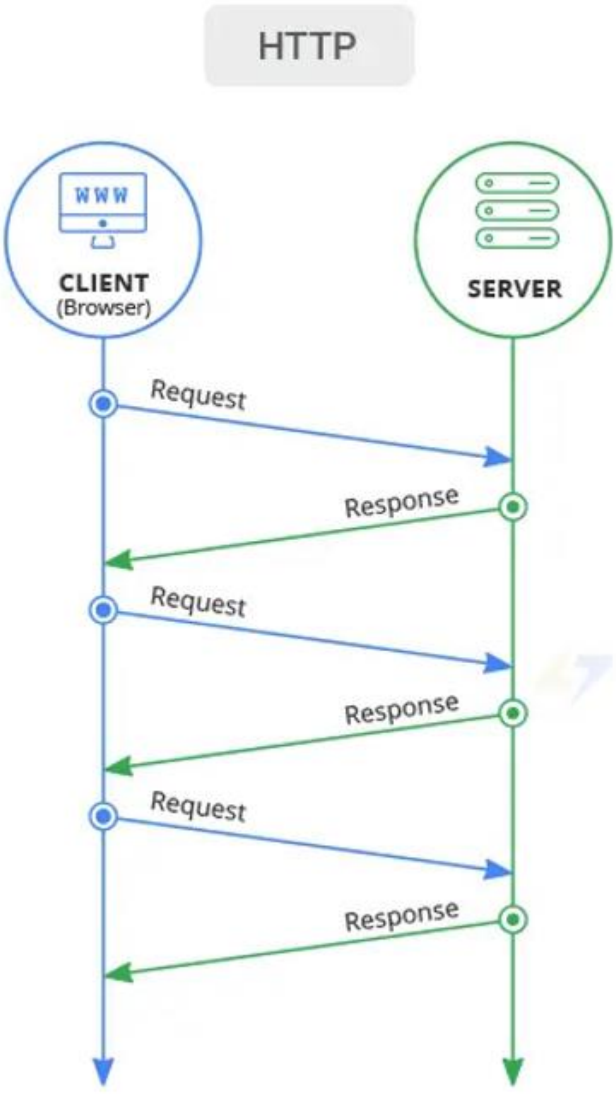
HTTP on the ESP32 uses simple text-based requests and responses over TCP:

- Request: A client sends a request line (GET /status HTTP/1.1), headers (Host:, User-Agent:), and an optional body for POST.
- Response: The server replies with a status line (HTTP/1.1 200 OK), headers (Content-Type: text/html; charset=utf-8, Connection: close), and the payload.
- Connection Handling: By default, connections are closed after each request (Connection: close), though keep-alive can be enabled for multiple requests per session.

ESPAsyncWebServer or WebServer libraries abstract this exchange, providing handler registration for URI patterns and automatic parsing of query parameters or JSON bodies.

Web Server


HTTP Request
ESP32 Web Server
192.168.1.147
ESP32
WEB SERVER
HTTP Response
</>


HTTP
WWW
CLIENT
(Browser)
SERVER
Request
Response
Request
Response
Request
Response

5.3. Handling Requests and Sending Responses

- Register Handlers
  Use server.on(path, method, handler) to bind URIs to callbacks. For example:

```
server.on("/toggle", HTTP_POST, handleToggle);
server.on("/status", HTTP_GET, [](){
  server.send(200, "application/json", getStatusJson());
});
```

- Parse Inputs
  Within handlers, access server.arg("param") for URL query strings or server.arg(i) for POST body fields. JSON payloads can be read via server.arg(0) and parsed with ArduinoJson.
- Generate Output
  Send plain text, JSON, or full HTML using server.send(code, contentType, payload). For file-based resources, use server.serveStatic("/app.js", SPIFFS, "/app.js").
- Start Server
  After setup, call server.begin(). Incoming requests spawn tasks on the internal HTTP thread, invoking handlers asynchronously.

5.4. Required Libraries

- WiFi.h: Core Wi-Fi connectivity and TCP/IP stack.
- WebServer.h (or ESPAsyncWebServer.h): HTTP server framework for handler registration, request parsing, and response generation.
- SPIFFS.h (optional): File system for serving static assets.
- ArduinoJson.h (optional): Efficient JSON parsing and serialization for REST-style APIs.

5.5. Code and Output Explanation

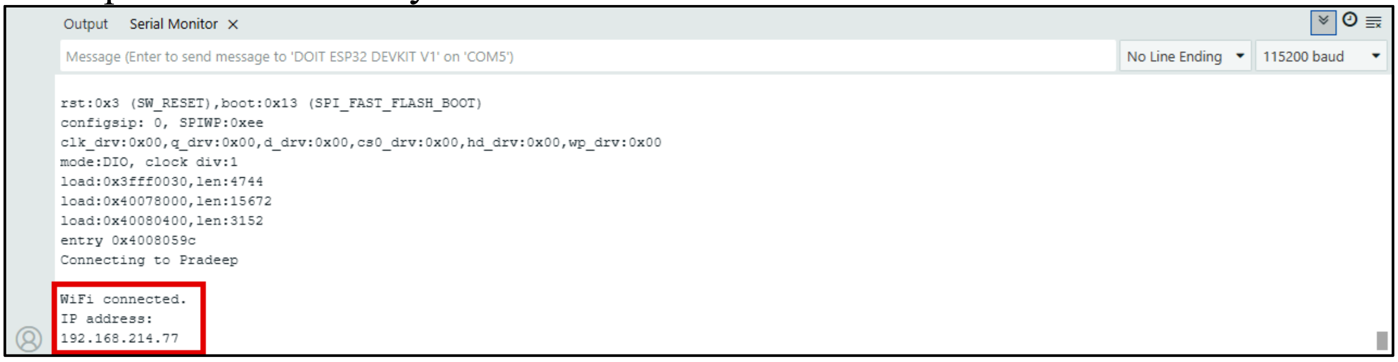
Complete Code:

```
#include <WiFi.h>          // Library for Wi-Fi functions
#include <WebServer.h>     // Library for web server

// Replace with your network credentials
const char* ssid = "Your_SSID";
const char* password = "Your_PASSWORD";

WebServer server(80);    // Create a web server object that listens on port 80

// HTML web page to send to clients
const char webPage[] PROGMEM = R"rawliteral(
<!DOCTYPE html>
<html>
  <head>
    <title>ESP32 Web Server</title>
  </head>
  <body>
    <h1>Welcome to ESP32 Web Server</h1>
    <p>This is a simple web server running on ESP32.</p>
  </body>
</html>
)rawliteral";

// Handler function to respond to root URL "/"
void handleRoot() {
  server.send(200, "text/html", webPage);
}

void setup() {
  Serial.begin(115200);          // Start serial communication for debugging
  WiFi.begin(ssid, password);    // Connect to Wi-Fi network

  Serial.print("Connecting to Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println();
  Serial.print("Connected! IP address: ");
  Serial.println(WiFi.localIP());   // Print the ESP32 IP address

  server.on("/", handleRoot);    // Define handler for root URL
  server.begin();                // Start the web server
  Serial.println("HTTP server started");
}

void loop() {
  server.handleClient();         // Handle client requests
}
```

Line-by-Line Explanation

- #include <WiFi.h>
  Includes the Wi-Fi library enabling network functions.
- #include <WebServer.h>
  Includes the library for running an HTTP web server on ESP32.
- const char* ssid = "Your_SSID";
  Stores the Wi-Fi network name to connect to.
- const char* password = "Your_PASSWORD";
  Stores the Wi-Fi password.
- WebServer server(80);
  Creates a server object listening on HTTP port 80.
- const char webPage[] PROGMEM = …
  Stores the HTML content of the web page to be served.
- void handleRoot()
  Defines a function to handle requests to the root URL (/), sending the stored HTML page with HTTP status 200 and MIME type text/html.

- void setup()
  Runs once on startup.
- Serial.begin(115200);
  Starts serial communication for debugging output at 115200 baud.
- WiFi.begin(ssid, password);
  Initiates connection to the specified Wi-Fi network.
- The while loop waits until ESP32 connects to Wi-Fi, printing dots periodically.
- Serial.println(WiFi.localIP());
  Once connected, prints the ESP32 IP address to Serial Monitor for accessing the web server.
- server.on("/", handleRoot);
  Sets up the handler function for root URL requests.
- server.begin();
  Starts the web server listening for incoming requests.
- void loop()
  Runs repeatedly; calls server.handleClient() to process incoming client requests continuously.


Output
Serial Monitor
Message (Enter to send message to 'DOIT ESP32 DEVKIT V1' on 'COM5')
No Line Ending
115200 baud

```
rst:0x3 (SW_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:4744
load:0x40078000,len:15672
load:0x40080400,len:3152
entry 0x4008059c
Connecting to Pradeep

WiFi connected.
IP address:
192.168.214.77
```

Once the above code is uploaded to the ESP32 and connected to a Wi-Fi network, the Serial Monitor will display the assigned IP address. Opening a web browser on any device connected to the same network and entering this IP address will load the hosted web page. The page will display a heading "Welcome to ESP32 Web Server" and a paragraph describing it as a simple web server running on ESP32. This confirms successful setup and communication, demonstrating how the ESP32 can serve web pages accessible on local networks for interactive IoT applications.

6. Controlling LED via Web Server

6.1. Circuit Connection LED with ESP32

To control an LED from the ESP32 web server, connect the LED’s anode (long leg) to a GPIO pin (e.g., GPIO 33) through a current-limiting resistor (330 Ω). Connect the LED’s cathode (short leg) to the ESP32 GND. If controlling two

LEDs, repeat on a second pin (e.g., GPIO 4). Ensure the resistor is placed in series to protect both the LED and the microcontroller.

6.2. Code and Output Explanation

Complete Code:

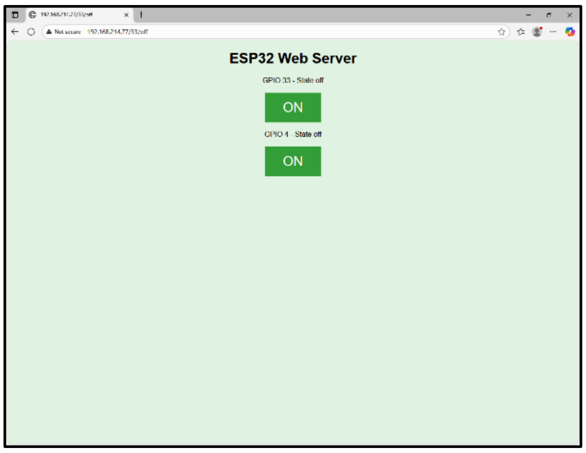
```
#include <WiFi.h>

// Replace with your network credentials
const char* ssid = "Your_SSID";
const char* password = "Your_PASSWORD";

WiFiServer server(80);

// Define GPIO pins for LEDs
const int ledPin33 = 33;
const int ledPin4 = 4;

String output33State = "off";
String output4State = "off";

void setup() {
  Serial.begin(115200);
  pinMode(ledPin33, OUTPUT);
  pinMode(ledPin4, OUTPUT);
  digitalWrite(ledPin33, LOW);
  digitalWrite(ledPin4, LOW);

  WiFi.begin(ssid, password);
  Serial.print("Connecting to Wi-Fi");

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println();
  Serial.print("Connected. IP address: ");
  Serial.println(WiFi.localIP());

  server.begin();
}

void loop() {
  WiFiClient client = server.available();
  if (!client) return;

  Serial.println("New Client Connected");
  String currentLine = "";
  String header = "";

  while (client.connected()) {
    if (client.available()) {
      char c = client.read();
      header += c;

      if (c == '\n') {
        // Check the HTTP request
        if (header.indexOf("GET /33/on") >= 0) {
          digitalWrite(ledPin33, HIGH);
          output33State = "on";
        }
        else if (header.indexOf("GET /33/off") >= 0) {
          digitalWrite(ledPin33, LOW);
          output33State = "off";
        }
        else if (header.indexOf("GET /4/on") >= 0) {
          digitalWrite(ledPin4, HIGH);
          output4State = "on";
        }
        else if (header.indexOf("GET /4/off") >= 0) {
          digitalWrite(ledPin4, LOW);
          output4State = "off";
        }

        // Send HTTP response
        client.println("HTTP/1.1 200 OK");
        client.println("Content-type:text/html");
        client.println();

        client.println("<!DOCTYPE html><html>");
        client.println("<head><title>ESP32 LED Control</title></head>");
        client.println("<body><h1>ESP32 Web Server</h1>");

        // Display LED 33 control buttons and state
        client.println("<p>GPIO 33 - State " + output33State + "</p>");
        if (output33State == "off") {
          client.println("<a href=\"/33/on\"><button>ON</button></a>");
        } else {
          client.println("<a href=\"/33/off\"><button>OFF</button></a>");
        }

        // Display LED 4 control buttons and state
        client.println("<p>GPIO 4 - State " + output4State + "</p>");
        if (output4State == "off") {
          client.println("<a href=\"/4/on\"><button>ON</button></a>");
        } else {
          client.println("<a href=\"/4/off\"><button>OFF</button></a>");
        }

        client.println("</body></html>");
        client.println();

        break;
      } else {
        currentLine = "";
      }
```

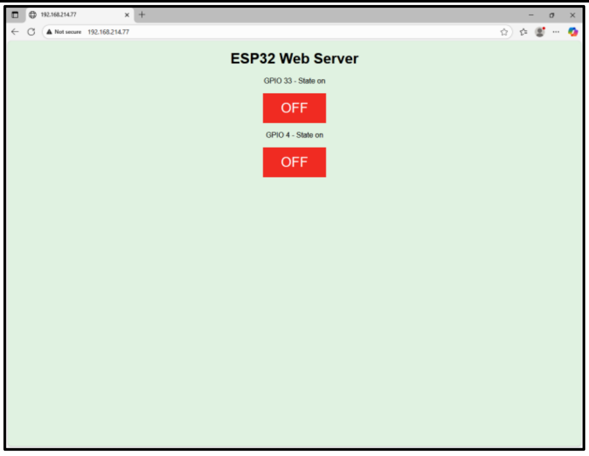
1

```
    }
  }

  // Clear header variable and stop client
  header = "";
  client.stop();
  Serial.println("Client disconnected");
  Serial.println();
}
```

Line-by-Line Explanation

- #include <WiFi.h>
  Includes ESP32 Wi-Fi library.
- const char* ssid and password
  Credentials for connecting ESP32 to Wi-Fi.
- WiFiServer server(80);
  Creates TCP server listening on HTTP port 80.
- const int ledPin33 = 33; and const int ledPin4 = 4;
  Assigns GPIO pins for LEDs.
- String output33State and output4State
  Store the current state of each LED ("on" or "off").
- void setup()
  Initializes serial communication, sets GPIO pins as output and turns LEDs off; connects to Wi-Fi; starts the server.
- void loop()
  Waits for clients to connect and handles their requests.
- WiFiClient client = server.available();
  Checks for incoming client connections.
- The inner while loop reads the HTTP request line-by-line.
- if (header.indexOf("GET /33/on") >= 0) and similar statements
  Detect user clicks on ON/OFF buttons and toggle the corresponding LED state.
- The server sends an HTML response:
  Shows the current state of each LED.
  Provides ON/OFF buttons as clickable links that trigger the GPIO state changes.
- client.stop();
  Ends client connection after the response.





After uploading the LED control web server code to the ESP32 and establishing a Wi-Fi connection, the Serial Monitor displays the IP address assigned to the ESP32. This IP address is used to access the web server from any device connected to the same Wi-Fi network, including mobile phones and PCs.
When the user enters the ESP32's IP address in a browser, a web page is displayed showing the current state (ON or OFF) of the LEDs connected to GPIO pins 33 and 4. The page provides clear ON and OFF buttons for each LED, allowing the user to remotely toggle their states by clicking the respective buttons. Clicking an ON or OFF button sends an HTTP GET request to the ESP32 web server which interprets the request and switches the corresponding GPIO pin HIGH or LOW, turning the LED ON or OFF physically. The page then reloads, instantly reflecting the updated LED state, providing real-time feedback on the web interface. This mechanism enables remote control of physical hardware via a simple, user-friendly web interface accessible on any device with a browser over a Wi-Fi network. It demonstrates how ESP32 can serve as a standalone IoT device controller, bridging web technologies and embedded systems.
This output approach is intuitive for students, showing how HTTP requests translate into physical device actions and how web servers can facilitate interactive remote hardware control in IoT projects.

6.3. Designing Web Interface

The interface is a minimal HTML page with two buttons and an embedded JavaScript function:

- Buttons trigger the JavaScript toggle(id) function.
- fetch() performs an asynchronous GET request to /toggle?led=id.
- Console Logging displays server responses in browser dev tools.

This approach avoids page reloads and provides real-time control.

6.4. Testing with Browser or Mobile

- Ensure your computer or mobile device is on the same Wi-Fi network as the ESP32.
- Open a browser and enter the ESP32's IP address (e.g., http://192.168.1.42).

- The page loads with two buttons labeled “Toggle LED 1” and “Toggle LED 2.”
- Tap a button to send a toggle command; observe the LED change state instantly.
- Inspect the browser console (F12 or dev tools) to view the server’s plain-text responses.

This setup demonstrates a robust, responsive mechanism for remotely controlling hardware via a lightweight embedded web server on the ESP32.

7. Displaying Analog Value on Dashboard

7.1. Introduction to ADC on ESP32

The ESP32 features two 12-bit successive-approximation ADCs (ADC1 and ADC2) with up to 18 input channels. Each ADC measures voltages between 0 V and 3.3 V, converting them into integer values from 0 to 4095. This enables precise sampling of analog sensors such as potentiometers, temperature sensors, or light-dependent resistors (LDR).

7.2. Reading Analog Values

To read an analog input, use:

```
int raw = analogRead(pin);
float voltage = raw * (3.3f / 4095.0f);
```

- analogRead(pin) returns a 0–4095 integer proportional to the measured voltage.
- Multiplying by 3.3 V/4095 converts the raw value into a floating-point voltage.

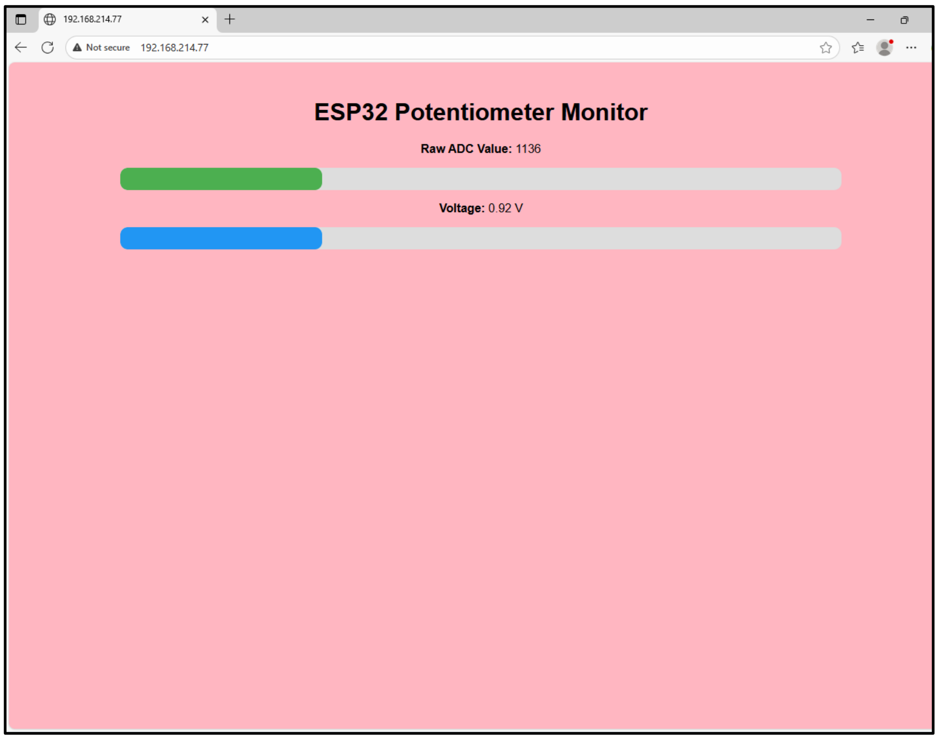
7.3. Code and Output Explanation

```
#include <WiFi.h>

// Replace with your network credentials
const char* ssid = "Your_SSID";
const char* password = "Your_PASSWORD";

WiFiServer server(80);

// Analog pin connected to the sensor (e.g., potentiometer)
const int analogPin = 34;

void setup() {
  Serial.begin(115200);

  pinMode(analogPin, INPUT);

  WiFi.begin(ssid, password);
  Serial.print("Connecting to Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println();
  Serial.print("Connected! IP address: ");
  Serial.println(WiFi.localIP());

  server.begin();
}

void loop() {
  WiFiClient client = server.available();

  if (!client) return;

  String currentLine = "";
  while (client.connected()) {
    if (client.available()) {
      char c = client.read();
      currentLine += c;

      if (c == '\n') {
        if (currentLine.length() == 1) {
          int analogValue = analogRead(analogPin);

          client.println("HTTP/1.1 200 OK");
          client.println("Content-type:text/html");
          client.println();

          client.println("<!DOCTYPE html><html>");
          client.println("<head><title>ESP32 Analog Read</title></head>");
          client.println("<body>");
          client.println("<h1>ESP32 Analog Sensor Reading</h1>");
          client.print("<p>Analog Value: ");
          client.print(analogValue);
          client.println("</p>");
          client.println("</body></html>");
          client.println();

          break;
        } else {
          currentLine = "";
        }
      }
    }
  }

  client.stop();
  Serial.println("Client disconnected");
}
```

Line-by-Line Explanation:

- #include <WiFi.h>
  Includes the library to manage Wi-Fi connectivity.

- const char* ssid and password
  Store the SSID and password of the Wi-Fi network to connect.
- WiFiServer server(80);
  Creates a server listening on port 80 (HTTP).
- const int analogPin = 34;
  Defines the ADC pin where the analog sensor is connected (GPIO 34 is an input-only pin).
- void setup()
  Initializes serial communication, sets the analog pin mode, connects to Wi-Fi, and starts the server.
- void loop()
  Continuously checks for incoming client connections.
- if (!client) return;
  If no client connects, the function exits early.
- The inner loop reads the incoming HTTP request line by line until an empty line signals the end of the request.
- Upon request end, it reads the analog value from the specified pin.
- Sends the HTTP response headers and constructs a simple HTML web page showing the current analog sensor value.
- client.stop();
  Stops the client connection and frees resources.
- Serial.println("Client disconnected");
  Logs client disconnection status.



ESP32 Potentiometer Monitor

Raw ADC Value: 1136

Voltage: 0.92 V

After uploading the code to the ESP32 and connecting it to a Wi-Fi network, the device will print its IP address in the Serial Monitor. When this IP address is entered into a browser on any device connected to the same Wi-Fi network (such as a smartphone or PC), a simple web page loads displaying the current analog value read from the connected sensor pin (e.g., potentiometer on GPIO 34). This value represents the sensor's real-time analog reading in digital form from 0 to 4095. As the analog sensor input varies, refreshing or reloading the web page will update the displayed value accordingly, giving a live view of sensor output remotely through the web interface. This output confirms the ESP32's ability to read analog data and serve this information dynamically via HTTP, effectively creating an accessible IoT dashboard for monitoring analog sensor values in real time.

7.4. Viewing Data on Mobile Browser

- Ensure the mobile device is on the same Wi-Fi network as the ESP32.
- Open a web browser and navigate to the ESP32's IP address.
- The page refreshes on each load, showing the latest analog reading.
- For live updates, add a meta-refresh tag or use JavaScript setInterval() to poll every second:

```
<meta http-equiv="refresh" content="1">
```

This provides a simple IoT dashboard accessible from any browser-equipped device.

8. Summary and Applications

8.1. IoT Use Cases with Wi-Fi and ESP32

Wi-Fi–enabled ESP32 modules excel in a wide array of IoT deployments. In smart homes, they power connected lighting, climate control, and security cameras by hosting local dashboards and bridging to cloud services. Within industrial automation, ESP32 boards monitor equipment health via vibration or temperature sensors, transmitting real-time metrics to on-premise servers for predictive maintenance. In agricultural IoT, soil moisture and environmental sensors leverage ESP32 web interfaces for remote irrigation control and data logging. Wearable health monitors also utilize the ESP32's Wi-Fi connectivity to stream biometric data to mobile apps for on-the-fly analytics.

8.2. Advantages of Wi-Fi Based IoT Projects

Wi-Fi integration on the ESP32 provides several key benefits:

- Ubiquitous Infrastructure: Leverages existing wireless LANs without additional gateways, reducing deployment complexity.
- High Throughput: Supports data rates up to 150 Mbps for responsive dashboards and multimedia streams.
- Rich Protocol Support: Full TCP/IP stack enables MQTT, HTTP, WebSocket, and secure TLS/SSL connections for versatile application protocols.
- Local and Cloud Connectivity: Dual-mode operation (AP + STA) allows simultaneous edge-local control and Internet-based services.
- Power Management: Advanced sleep modes and adjustable transmit power extend battery life in portable sensors and wearables.

8.3. Limitations and Improvements

Despite its strengths, Wi-Fi on the ESP32 has constraints:

- 2.4 GHz Only: Lacks 5 GHz support, which can limit performance in congested environments.
- Power Consumption: Active Wi-Fi draws significantly more current than low-power LPWAN technologies, impacting battery-powered designs.
- Range and Interference: 2.4 GHz signals face interference from Bluetooth and household devices, reducing effective range.

Potential improvements include:

- External Antenna: Adding an external RF antenna or diversity network can boost range and reliability.
- Mesh Networking: Employ protocols like ESP-Mesh to extend coverage beyond single-AP limits.
- Adaptive Power Control: Dynamically adjust transmit power based on RSSI to balance connectivity and energy usage.
- Hybrid Architectures: Integrate Wi-Fi with BLE or LoRaWAN to optimize for long-range, low-power scenarios when full throughput is unnecessary.