Guide to Setting Up Wi-Fi Provisioning on ESP32 Using ESP-IDF on Linux

Table of Contents

---

1. Introduction

What is Wi-Fi Provisioning?

Wi-Fi provisioning is the process of **remotely** providing **Wi-Fi credentials** (SSID and password) to an **ESP32** device using a **Bluetooth Low Energy (BLE)** connection. This process eliminates the need to hardcode the network credentials directly into the firmware, offering **flexibility** and **ease of deployment**, especially when you need to configure many devices in a network.

In this guide, we will cover **every step** to set up **Wi-Fi provisioning** on your **ESP32** using the **ESP-IDF framework**.

---

2. Setting Up Your Development Environment

2.1 Installing Prerequisites

Before you can begin development work on the **ESP32** using **ESP-IDF**, it's important to install the necessary dependencies required for **building**, **flashing**, and **running firmware** on the ESP32 device. These tools will enable a seamless development environment, allowing you to interact with the ESP32 hardware efficiently.

Commands and Explanations:

1. Update Package Lists:

```
sudo apt update
```

- **`sudo`**: The `sudo` (superuser do) command grants **root privileges** to the user, allowing them to run administrative tasks like software installation. It's necessary when modifying system files or installing software packages.
- **`apt`**: This is the **Advanced Package Tool**, which is Ubuntu's package management system used to handle the installation and removal of software. `apt` simplifies package management tasks such as searching for packages, installing, upgrading, and removing them.
- **`update`**: The `apt update` command updates the list of **available packages** and their versions from the repositories configured on the system. This ensures that your system has the latest list of available software and its versions. This step is critical because it ensures that you're installing the **latest stable versions** of the required dependencies.

2. Install the Required Packages:

```
sudo apt install -y git wget flex bison gperf python3 python3-pip
python3-venv cmake ninja-build ccache libffi-dev libssl-dev dfu-util
```

This command installs some **essential packages** for working with the **ESP32**. Let's break down each of these packages:

- **`git`**:
  `git` is a **version control system**. It is crucial for **downloading code** from repositories (e.g., GitHub). ESP-IDF and many other software libraries are hosted on GitHub, and `git` allows you to clone the repository and download the source code. With `git`, you can track and manage different versions of code during development.
- **`wget`**:
  `wget` is a **network downloader**. It is used for **downloading files** from the internet via HTTP or FTP protocols. In the ESP-IDF setup process, `wget` is often used to fetch dependencies or additional libraries that the system needs for building the ESP32 firmware.
- **`flex` and `bison`**:
  `flex` and **`bison`** are tools for generating **lexical analyzers** and **parsers**. These tools are often used in compiling source code and are necessary for **parsing** and **compiling** complex code structures in the ESP-IDF framework. They enable the handling of input code that needs to be broken down into recognizable components.

- **`gperf`**:
  `gperf` is used to generate **perfect hash functions**. This is particularly useful when you need to convert a list of items (such as strings) into a hash, enabling fast lookup and minimal collision. In the context of ESP-IDF, `gperf` helps optimize lookups during code execution.
- **`python3`, `python3-pip`, and `python3-venv`**:

  `python3`: Python is required for various **scripts** and **tools** in ESP-IDF (such as setup scripts and build tools). Python 3.x is needed as some ESP-IDF tools are written in Python.

  `python3-pip`: `pip` is the **Python package manager**. It's used to install Python libraries that are necessary for building and working with the ESP-IDF environment.

  `python3-venv`: This package allows the creation of **isolated Python environments**. It prevents dependency conflicts across different projects by providing isolated environments where you can install specific versions of libraries without affecting the rest of the system.

- **`cmake`**:
  `cmake` is a **build system generator**. It is used to configure the build process for software projects. ESP-IDF uses CMake to handle the **build configuration**, ensuring that the right toolchains and settings are applied to compile the firmware for ESP32.
- **`ninja-build`**:
  `ninja` is a **build system** that is more **efficient** and **faster** than traditional build tools like `make`. It is used by ESP-IDF for **incremental builds**, which means it only rebuilds parts of the project that have changed, speeding up the development cycle.
- **`ccache`**:
  `ccache` is a **compiler cache**. It speeds up the build process by caching the compiled object files. If the same source files are compiled again, `ccache` fetches the results from the cache, avoiding recompilation and reducing build time.
- **`libffi-dev` and `libssl-dev`**:
  These libraries provide **cryptographic functions** that are needed for secure communication between the ESP32 and other devices (e.g., when using SSL/TLS protocols).
- **`dfu-util`**:
  `dfu-util` is a tool used to **flash firmware** onto devices like the ESP32. It communicates with the ESP32 to upload compiled binaries to the device.

2.2 Installing ESP-IDF

Now that all the necessary dependencies are installed, the next step is to **install ESP-IDF** (the development framework for ESP32). ESP-IDF contains all the tools, libraries, and APIs needed to develop applications on the ESP32.

Cloning the ESP-IDF Repository

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

- **`cd ~/esp`**: This changes the directory to `~/esp`, which is where we will install ESP-IDF. You can choose a different directory if needed.
- **`git clone --recursive`**: This command clones the entire ESP-IDF repository from GitHub, which contains the official codebase for the ESP32. The `--recursive` flag ensures that **all submodules** (dependencies) required by ESP-IDF are also cloned, ensuring a complete installation.

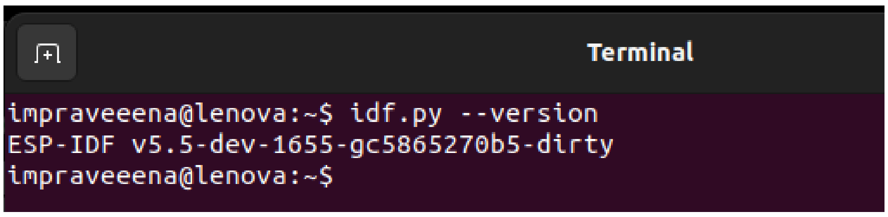
Running the Setup Script

```
cd ~/esp/esp-idf
./install.sh esp32
```

- **`cd ~/esp/esp-idf`**: This navigates into the `esp-idf` directory where the ESP-IDF code has been cloned.
- **`./install.sh esp32`**: The `install.sh` script is designed to install the required **toolchain** and other **dependencies** for compiling code for the ESP32. The `esp32` argument specifies that the installation is targeted for the **ESP32 architecture**. This script configures all the necessary components for building and flashing ESP32 firmware, including downloading compilers, building tools, and other platform-specific dependencies.

Once the script finishes running, the **ESP-IDF environment** will be fully set up, and you'll be ready to build projects for the ESP32.

To check the installation process follow the below steps



```
impraveeena@lenova:~$ idf.py --version
ESP-IDF v5.5-dev-1655-gc5865270b5-dirty
impraveeena@lenova:~$
```

3. Exploring the Wi-Fi Provisioning Example

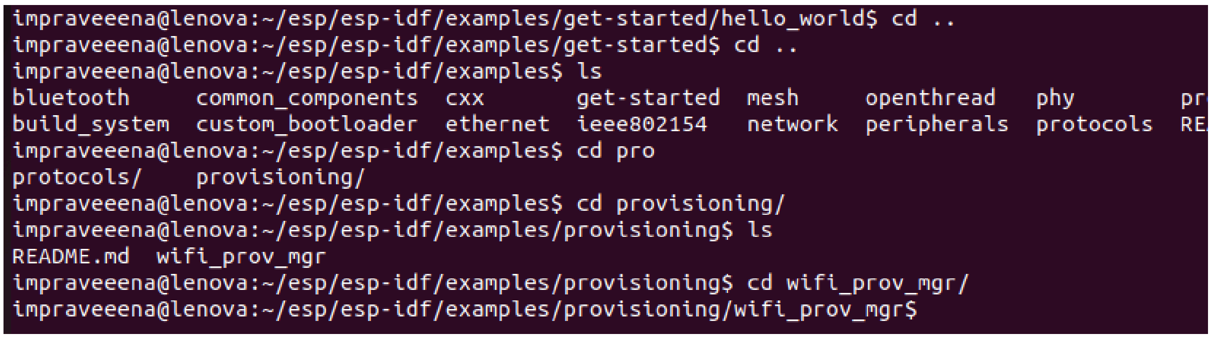
Once you've set up **ESP-IDF** and installed all prerequisites, you're ready to explore and work with an example project. In this case, the **Wi-Fi provisioning** example demonstrates how to use **Bluetooth Low Energy (BLE)** to send **Wi-Fi credentials** to an ESP32 device.

Navigating to the Example Directory

To access the **Wi-Fi provisioning example**, you'll need to navigate to the specific folder within the ESP-IDF repository. Here's how you can do that:

```
cd ~/esp/esp-idf/examples/provisioning/wifi_prov_mgr
```

- **`cd ~/esp/esp-idf`**: This command changes your current directory to the `esp-idf` folder where the ESP-IDF framework has been cloned. The ~ represents your **home directory**. The ESP-IDF should have been cloned into this directory if you followed the installation steps earlier.
- **`cd examples/provisioning/wifi_prov_mgr`**: This part of the command changes your directory to the **Wi-Fi provisioning example** folder, specifically the **`wifi_prov_mgr`** folder, which contains the code for managing the BLE provisioning process. This folder contains the core example that demonstrates how to use BLE to provision Wi-Fi credentials.



```
impraveeena@lenova:~/esp/esp-idf/examples/get-started/hello_world$ cd ..
impraveeena@lenova:~/esp/esp-idf/examples/get-started$ cd ..
impraveeena@lenova:~/esp/esp-idf/examples$ ls
bluetooth     common_components  cxx        get-started  mesh     openthread   phy        pr
build_system  custom_bootloader  ethernet   ieee802154   network  peripherals  protocols  RE
impraveeena@lenova:~/esp/esp-idf/examples$ cd pro
protocols/    provisioning/
impraveeena@lenova:~/esp/esp-idf/examples$ cd provisioning/
impraveeena@lenova:~/esp/esp-idf/examples/provisioning$ ls
README.md  wifi_prov_mgr
impraveeena@lenova:~/esp/esp-idf/examples/provisioning$ cd wifi_prov_mgr/
impraveeena@lenova:~/esp/esp-idf/examples/provisioning/wifi_prov_mgr$
```

Understanding the Code Structure

The **Wi-Fi provisioning example** demonstrates how to set up the **Wi-Fi credentials** on an **ESP32** using **Bluetooth Low Energy (BLE)**. This is done by having the ESP32 advertise its presence over BLE, allowing a **mobile app** (like the **ESP BLE Provisioning App**) to connect, send the Wi-Fi SSID and password, and have the ESP32 connect to that network.

The **main logic** of this example resides in the `main/app_main.c` file. Here's a quick breakdown of its role:

- **`app_main.c`** contains the entry point for the application and houses the main logic for:

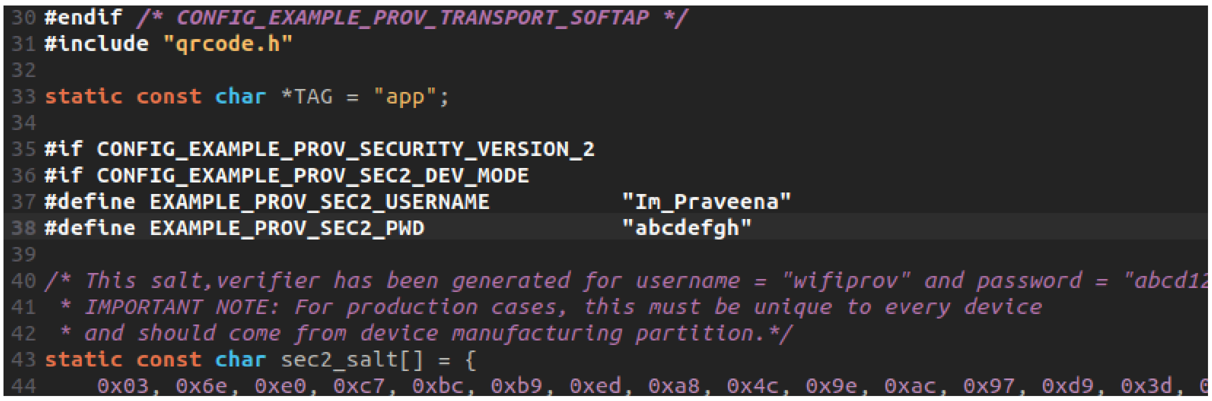
BLE advertising: The ESP32 begins advertising itself over Bluetooth to be discovered by the **ESP BLE Provisioning App**.

Handling Wi-Fi provisioning: Once the app sends the credentials (SSID and password), the ESP32 stores them and uses them to connect to the Wi-Fi network.

---

4. Building and Flashing the Code

After exploring the example and possibly modifying it (e.g., changing the BLE device name or altering the default Wi-Fi credentials), the next step is to **build** and **flash** the code onto the **ESP32**.



```
#endif /* CONFIG_EXAMPLE_PROV_TRANSPORT_SOFTAP */
#include "qrcode.h"

static const char *TAG = "app";

#if CONFIG_EXAMPLE_PROV_SECURITY_VERSION_2
#if CONFIG_EXAMPLE_PROV_SEC2_DEV_MODE
#define EXAMPLE_PROV_SEC2_USERNAME          "Im_Praveena"
#define EXAMPLE_PROV_SEC2_PWD               "abcdefgh"

/* This salt,verifier has been generated for username = "wifiprov" and password = "abcd12
 * IMPORTANT NOTE: For production cases, this must be unique to every device
 * and should come from device manufacturing partition.*/
static const char sec2_salt[] = {
    0x03, 0x6e, 0xe0, 0xc7, 0xbc, 0xb9, 0xed, 0xa8, 0x4c, 0x9e, 0xac, 0x97, 0xd9, 0x3d, 0
```

4.1 Setting the Target Device

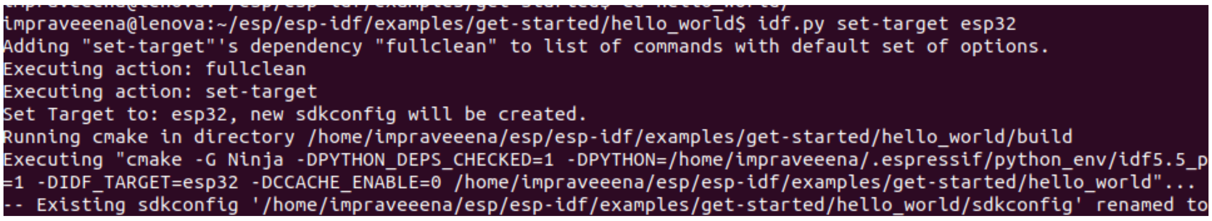
When you work with ESP-IDF, you need to specify which **target device** (in this case, the ESP32) you're building for. This ensures that the build system compiles the code correctly for the architecture of the target device.

To set the **target device** as **ESP32**, run the following command:

```
idf.py set-target esp32
```

- **`idf.py`**: This is the primary tool used in **ESP-IDF** for managing the build and flash process. It's responsible for compiling the code, flashing it onto the ESP32, and managing various other tasks like monitoring serial output or cleaning up the build.
- **`set-target esp32`**: This argument tells `idf.py` that the firmware should be **compiled for the ESP32 architecture**. ESP32 is the target device for this example, so setting the target ensures that the build system uses the correct toolchain, libraries, and configurations specific to the ESP32.

By setting the target device, you're ensuring that the toolchain will correctly build the firmware for the ESP32 architecture. If you were using a different ESP chip (like **ESP32-S2** or **ESP32-C3**), you would specify the respective chip name.



```
impraveeena@lenova:~/esp/esp-idf/examples/get-started/hello_world$ idf.py set-target esp32
Adding "set-target"'s dependency "fullclean" to list of commands with default set of options.
Executing action: fullclean
Executing action: set-target
Set Target to: esp32, new sdkconfig will be created.
Running cmake in directory /home/impraveeena/esp/esp-idf/examples/get-started/hello_world/build
Executing "cmake -G Ninja -DPYTHON_DEPS_CHECKED=1 -DPYTHON=/home/impraveeena/.espressif/python_env/idf5.5_p
=1 -DIDF_TARGET=esp32 -DCCACHE_ENABLE=0 /home/impraveeena/esp/esp-idf/examples/get-started/hello_world"...
-- Existing sdkconfig '/home/impraveeena/esp/esp-idf/examples/get-started/hello_world/sdkconfig' renamed to
```

4.2 Building the Project

After setting the target, the next step is to **build the project**. This is the process of **compiling** the source code into executable binary files that can be flashed onto the ESP32.

To build the project, run:

```
idf.py build
```

- **`build`**: This command tells **`idf.py`** to **compile** the source code. During this process:

    Preprocessing: The source code files are checked for dependencies, and headers are included as needed.

    Compilation: The code is compiled into machine language, which the ESP32 can understand.

    Linking: After compiling individual source files, they are linked together to form a single binary image.

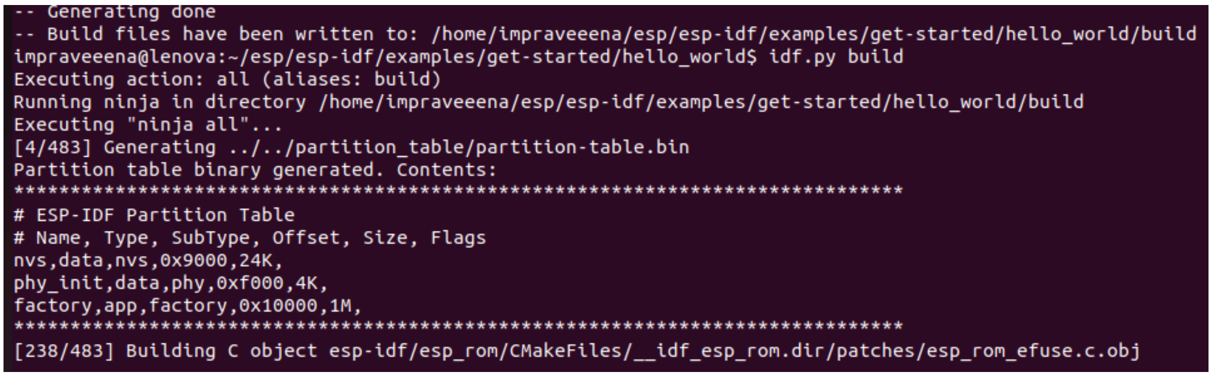
Once the build completes, **binary files** are generated in the `build/` folder. These files include the application code and other necessary components that the ESP32 requires to run.

- The **final binaries** include:

    Application Binary: The compiled application code for the ESP32.

    Partition Table: Defines how memory is organized on the ESP32.

    Bootloader: The initial program that runs on the ESP32 when it powers up, which loads the application code.



```
-- Generating done
-- Build files have been written to: /home/impraveeena/esp/esp-idf/examples/get-started/hello_world/build
impraveeena@lenova:~/esp/esp-idf/examples/get-started/hello_world$ idf.py build
Executing action: all (aliases: build)
Running ninja in directory /home/impraveeena/esp/esp-idf/examples/get-started/hello_world/build
Executing "ninja all"...
[4/483] Generating ../../partition_table/partition-table.bin
Partition table binary generated. Contents:
*******************************************************************************
# ESP-IDF Partition Table
# Name, Type, SubType, Offset, Size, Flags
nvs,data,nvs,0x9000,24K,
phy_init,data,phy,0xf000,4K,
factory,app,factory,0x10000,1M,
*******************************************************************************
[238/483] Building C object esp-idf/esp_rom/CMakeFiles/__idf_esp_rom.dir/patches/esp_rom_efuse.c.obj
```

4.3 Flashing the Firmware

After successfully building the project, you need to **flash** the compiled firmware onto the ESP32. Flashing transfers the compiled binaries to the ESP32's **flash memory** so it can execute them.

To flash the firmware to the ESP32, run:

```
idf.py flash
```

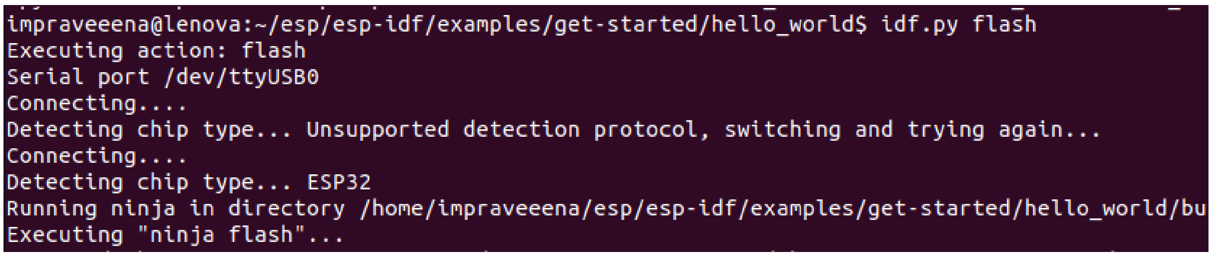
- **`flash`**: This command instructs **`idf.py`** to upload the compiled binaries to the **ESP32** via the **USB** connection.

  The command will **automatically detect** the connected device and upload the **application firmware**, **bootloader**, and **partition table**.

  It communicates with the ESP32 over the USB interface and programs the device's flash memory.

During the flashing process, you should see output like the following:

```
Flashing binaries to serial port /dev/ttyUSB0 (app, bootloader,
partition table)...
Hash of data verified.
Compressed 123456 bytes at 0x1000...
...
```



```
impraveeena@lenova:~/esp/esp-idf/examples/get-started/hello_world$ idf.py flash
Executing action: flash
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... Unsupported detection protocol, switching and trying again...
Connecting....
Detecting chip type... ESP32
Running ninja in directory /home/impraveeena/esp/esp-idf/examples/get-started/hello_world/bu
Executing "ninja flash"...
```

Once the flash operation completes, the ESP32 will be ready to execute the newly uploaded firmware.

---

5. Wi-Fi Provisioning Process

Once the firmware is flashed, the **ESP32** will start the **Wi-Fi provisioning** process. This involves advertising itself over **BLE** and waiting for the **ESP BLE Provisioning App** to send Wi-Fi credentials.

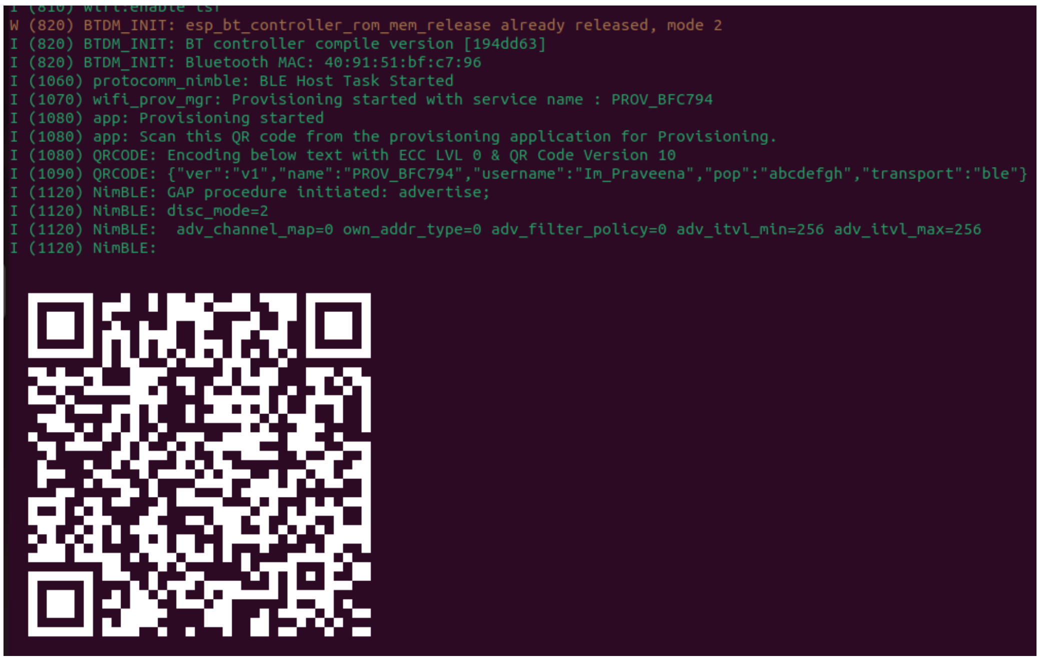
Monitor the Output

You can monitor the output using the `idf.py monitor` command:

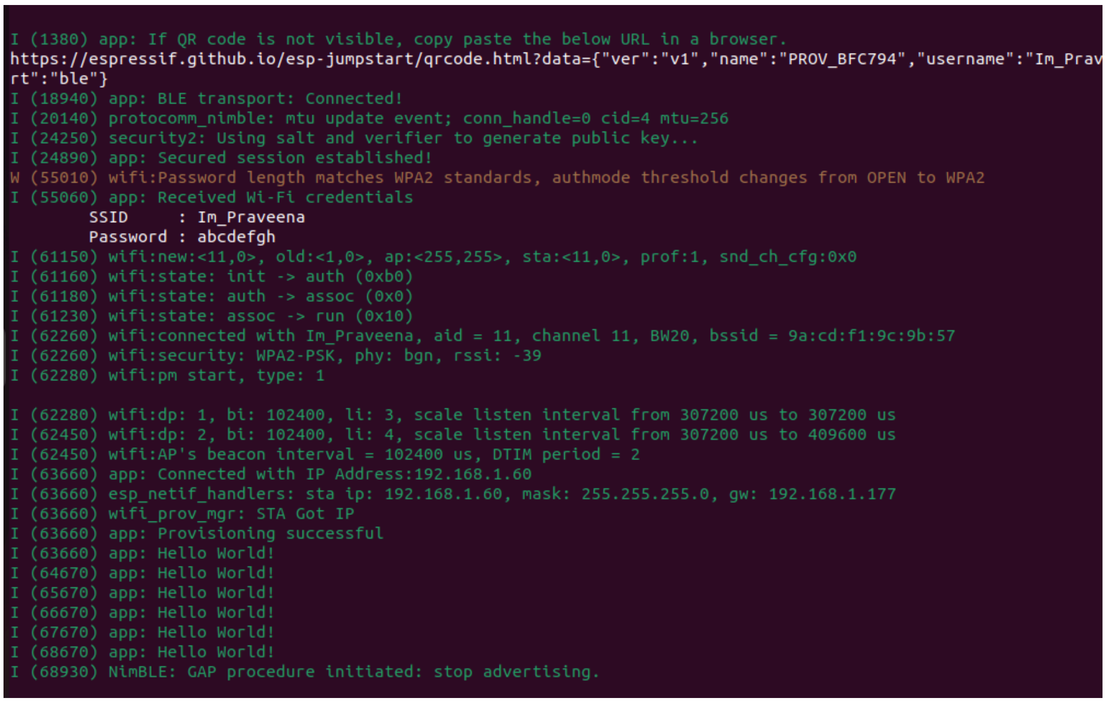
```
idf.py monitor
```

- **`monitor`**: This command opens a serial monitor that allows you to view the real-time logs from the ESP32. You can see status messages like:

```
I (30000) app: Waiting for provisioning...
I (50000) app: Provisioning success!
```



```
I (810) wifi:enable tsf
W (820) BTDM_INIT: esp_bt_controller_rom_mem_release already released, mode 2
I (820) BTDM_INIT: BT controller compile version [194dd63]
I (820) BTDM_INIT: Bluetooth MAC: 40:91:51:bf:c7:96
I (1060) protocomm_nimble: BLE Host Task Started
I (1070) wifi_prov_mgr: Provisioning started with service name : PROV_BFC794
I (1080) app: Provisioning started
I (1080) app: Scan this QR code from the provisioning application for Provisioning.
I (1080) QRCODE: Encoding below text with ECC LVL 0 & QR Code Version 10
I (1090) QRCODE: {"ver":"v1","name":"PROV_BFC794","username":"Im_Praveena","pop":"abcdefgh","transport":"ble"}
I (1120) NimBLE: GAP procedure initiated: advertise;
I (1120) NimBLE: disc_mode=2
I (1120) NimBLE:  adv_channel_map=0 own_addr_type=0 adv_filter_policy=0 adv_itvl_min=256 adv_itvl_max=256
I (1120) NimBLE:
```



```
I (1380) app: If QR code is not visible, copy paste the below URL in a browser.
https://espressif.github.io/esp-jumpstart/qrcode.html?data={"ver":"v1","name":"PROV_BFC794","username":"Im_Prav
rt":"ble"}
I (18940) app: BLE transport: Connected!
I (20140) protocomm_nimble: mtu update event; conn_handle=0 cid=4 mtu=256
I (24250) security2: Using salt and verifier to generate public key...
I (24890) app: Secured session established!
W (55010) wifi:Password length matches WPA2 standards, authmode threshold changes from OPEN to WPA2
I (55060) app: Received Wi-Fi credentials
        SSID     : Im_Praveena
        Password : abcdefgh
I (61150) wifi:new:<11,0>, old:<1,0>, ap:<255,255>, sta:<11,0>, prof:1, snd_ch_cfg:0x0
I (61160) wifi:state: init -> auth (0xb0)
I (61180) wifi:state: auth -> assoc (0x0)
I (61230) wifi:state: assoc -> run (0x10)
I (62260) wifi:connected with Im_Praveena, aid = 11, channel 11, BW20, bssid = 9a:cd:f1:9c:9b:57
I (62260) wifi:security: WPA2-PSK, phy: bgn, rssi: -39
I (62280) wifi:pm start, type: 1

I (62280) wifi:dp: 1, bi: 102400, li: 3, scale listen interval from 307200 us to 307200 us
I (62450) wifi:dp: 2, bi: 102400, li: 4, scale listen interval from 307200 us to 409600 us
I (62450) wifi:AP's beacon interval = 102400 us, DTIM period = 2
I (63660) app: Connected with IP Address:192.168.1.60
I (63660) esp_netif_handlers: sta ip: 192.168.1.60, mask: 255.255.255.0, gw: 192.168.1.177
I (63660) wifi_prov_mgr: STA Got IP
I (63660) app: Provisioning successful
I (63660) app: Hello World!
I (64670) app: Hello World!
I (65670) app: Hello World!
I (66670) app: Hello World!
I (67670) app: Hello World!
I (68670) app: Hello World!
I (68930) NimBLE: GAP procedure initiated: stop advertising.
```

---

6. Using the ESP BLE Provisioning App

The **ESP BLE Provisioning App** is used to send Wi-Fi credentials to the ESP32.

6.1 Downloading the App

- You need to install the **ESP BLE Provisioning App** on your Android phone from the Google Play Store.

6.2 Provision Wi-Fi Credentials

1. Open the **ESP BLE Provisioning App**.
2. Tap on **"Provision New Device"**.
3. Select your ESP32 device (it will appear as `"MY_ESP_WIFI_SETUP"` or whatever name you have set).
4. Enter your **Wi-Fi SSID** and **Password**.
5. Tap **"Provision"** to send the credentials to the ESP32.

---

7. Final Notes and Troubleshooting

7.1 Common Issues

- **ESP32 not showing up**: Make sure Bluetooth is enabled on your phone. If issues persist, try restarting the ESP32 and phone.
- **Wi-Fi connection fails**: Recheck the Wi-Fi credentials. Ensure the Wi-Fi network is stable and within range.

7.2 Erasing Flash

If the ESP32 fails to provision or you want to start over, erase the flash:

```
idf.py erase-flash
idf.py flash
```

This command erases the existing firmware and prepares the device for fresh provisioning.

Conclusion

This detailed guide has walked you through the entire process of setting up **Wi-Fi provisioning** on your **ESP32** using **ESP-IDF**. We've covered every command and explained its purpose in depth. With this, you should be able to:

Set up your development environment.

Work with the Wi-Fi provisioning example.

Build, flash, and troubleshoot the code.

🎥 Demo Videos

WATCH THE DEMO VIDEOS HERE

Terminal screen recording: 🎬 **Terminal_SR.webm**
App demo: 🎬 **ESP-IDF.mp4**

Description

In this video, I demonstrate how the ESP32 connects to a Wi-Fi network using BLE (Bluetooth Low Energy) provisioning. The video shows the following steps:

1. BLE advertising by the ESP32, where it becomes discoverable by the ESP BLE Provisioning App.
2. Provisioning process using the mobile app, where the user inputs their Wi-Fi SSID and password.
3. The ESP32 successfully connects to the Wi-Fi and shows the IP address in the serial monitor.

This video provides a step-by-step visual walkthrough of how the Wi-Fi provisioning process works and verifies the successful connection to the Wi-Fi network.