ESP-Jumpstart

Feb 07, 2023

Contents

1 Introduction **3**
- 1.1 ESP-Jumpstart: Build ESP32 Products Fast . . . 3
- 1.2 For the Restless . . . 5

2 Getting Started **7**
- 2.1 Development Overview . . . 7
- 2.2 ESP-IDF . . . 7
- 2.3 Getting ESP-Jumpstart . . . 8
- 2.4 The Code . . . 11
- 2.5 Progress so far . . . 12

3 The Driver **13**
- 3.1 The Push Button . . . 13
- 3.2 The Output . . . 14
- 3.3 Progress so far . . . 15

4 Wi-Fi Connection **17**
- 4.1 The Code . . . 17
- 4.2 Progress so far . . . 18

5 Network Configuration **19**
- 5.1 Overview . . . 19
- 5.2 Demo . . . 20
- 5.3 Unified Provisioning . . . 21
- 5.4 NVS: Persistent key-value store . . . 24
- 5.5 Reset to Factory . . . 25
- 5.6 Progress so far . . . 26

6 Remote Control (Cloud) **27**
- 6.1 Security First . . . 28
- 6.2 Embedding Files in the Firmware . . . 28
- 6.3 AWS IoT . . . 28
- 6.4 Progress so far . . . 31

7 Firmware Upgrades **33**
- 7.1 Flash Partitions . . . 33
- 7.2 OTA Mechanism . . . 34

7.3 The Code . . . 36
7.4 Send Firmware Upgrade URL . . . 36
7.5 Progress So Far . . . 37

8 Manufacturing **39**
8.1 Multiple NVS Partitions . . . 39
8.2 The Code . . . 40
8.3 Generating the Factory Data . . . 40
8.4 Progress So Far . . . 41

9 Security Considerations **43**
9.1 Securing Remote Communication . . . 43
9.2 Securing Physical Accesses . . . 44

[]

Building Products with ESP32 fast: Jumpstart from concept to production

Jumpstart Now

CHAPTER 1

Introduction

[]

1.1 ESP-Jumpstart: Build ESP32 Products Fast

Building production-ready firmware can be hard. It involves multiple questions and decisions about the best ways of doing things. It involves building phone applications, and integrating cloud agents to get all the features done. What if there was a ready reference, a known set of best steps, gathered from previous experience of others, that you could jumpstart with?

ESP-Jumpstart is focused on building *products* on ESP32. It is a quick-way to get started into your product development process. ESP-Jumpstart builds a fully functional, ready to deploy "Smart Power Outlet" in a sequence of incremental tutorial steps. Each step addresses either a user-workflow or a developer workflow. Each step is an application built with ESP-IDF, ESP32's software development framework.

The ESP-Jumpstart's Smart Power Outlet firmware assumes the device has one input push-button, and one GPIO output. It implements the following commonly required functionality.

- Allows and end-user to configure their home Wi-Fi network through phone applications (iOS/Android)
- Switch on or off the GPIO output
- Use a push-button to physically toggle this output
- Allow remote control of this output through a cloud
- Implement over-the-air (OTA) firmware upgrade
- Perform *Reset to Factory* settings on long-press of the push-button

Once you are familiar with ESP-Jumpstart, building your production firmware, is a matter of replacing the power-outlet's device driver, with your device driver (bulb, washing machine).

You will require the following to get started:

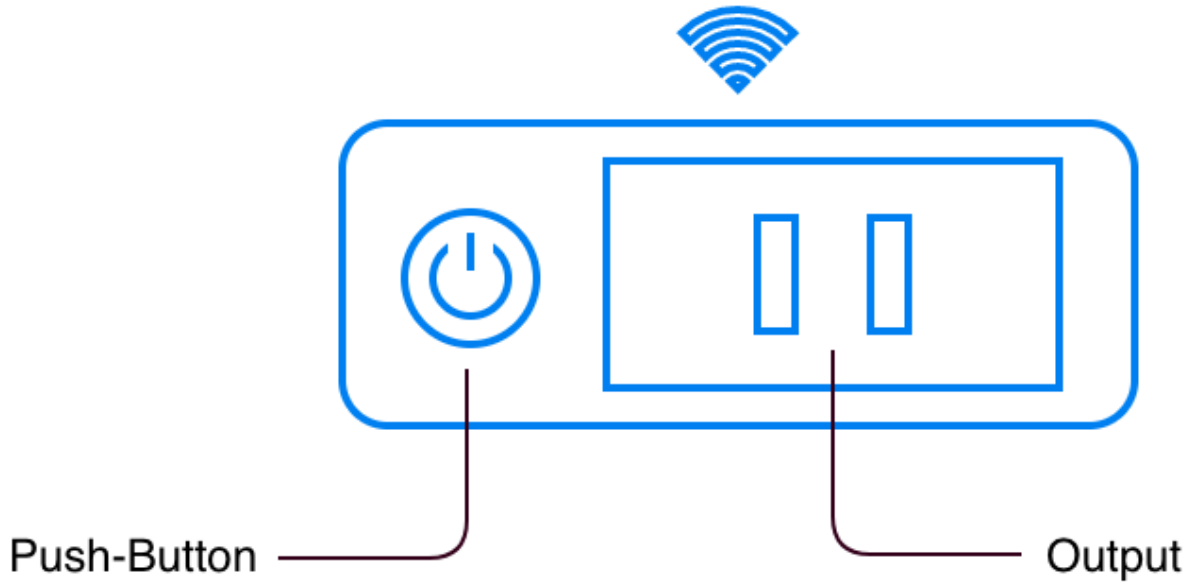

Fig. 1: Smart Power Outlet

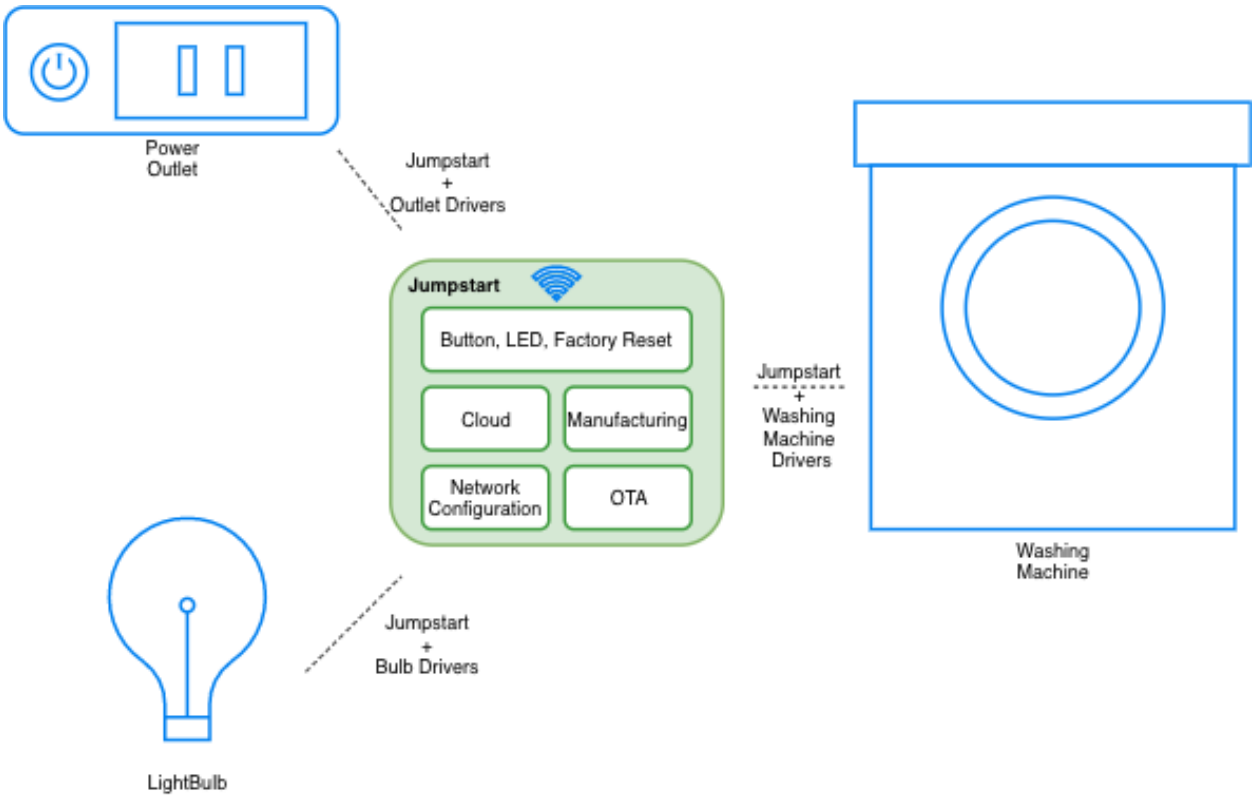

Fig. 2: Jumpstart Applicability

- An ESP32 development kit: ESP32-DevKit-C https://www.espressif.com/en/products/hardware/esp32-devkitc/overview available through your closest store. You could also use any other ESP32 development board if you already have one.
- A Development host setup (Windows, Linux or Mac) that will be used for development.

1.1.1 For ESP8266 Users

You will require the following to get started:

- An ESP8266 development kit: ESP8266-DevKit-C https://www.espressif.com/products/hardware/esp8266ex/overview/ available through your closest store. You could also use any other ESP8266 development board if you already have one.
- ESP8266_RTOS_SDK is Espressif's IoT Development Framework for ESP8266 (https://docs.espressif.com/projects/esp8266-rtos-sdk/en/latest). All references to IDF, ESP-IDF are to be mapped to ESP8266_RTOS_SDK in the context of ESP8266.
- The instructions for ESP32 and ESP8266 are common unless specified under the **For ESP8266 Users** section.

1.2 For the Restless

If you are familiar with Espressif's hardware and/or embedded systems, and are looking for a production-reference without the incremental steps, you can do the following:

1. Directly use the final application in ESP-Jumpstart
2. If you don't have a cloud account, configure your AWS IoT Cloud configuration as mentioned in Section *AWS IoT*
3. Create the manufacturing configuration file for your device's unique cloud credentials, based on the instructions provided in Section *Generating the Factory Data* and flash it at the appropriate location
4. Build, flash and boot up the firmware image as you usually do
5. Use the reference phone-app (iOS/Android) libraries for building your phone applications. Or use the reference application to try things out as discussed in Section *Unified Provisioning*
6. Use the commands discussed in Section *AWS IoT* for remote control
7. Now that you have this functional, modify to work with your driver

CHAPTER 2

Getting Started

[]

In this chapter, our aim would be to get our development setup functional, and also to get an understanding for the development tools and repositories available around ESP32.

2.1 Development Overview

The following diagram depicts the typical developer setup for development with ESP32.

The PC, or the Development Host can be any of Linux, Windows or Mac. The ESP32 based development board is connected to the Development Host over a USB cable. The Development Host has the ESP-IDF (Espressif's SDK), the compiler toolchain and the code for your project. The development host builds this code and generates the executable firmware image. The tools on the Development Host then download the generated firmware image on to the development board. As the firmware executes on the development board, the logs from the firmware can be monitored from the Development Host.

2.2 ESP-IDF

ESP-IDF is Espressif's IoT Development Framework for ESP32.

- ESP-IDF is a collection of libraries and header files that provides the core software components that are required to build any software projects on ESP32.
- ESP-IDF also provides tools and utilities that are required for typical developer and production usecases, like build, flash, debug and measure.

2.2.1 Setting up IDF

Please follow the steps in this documentation for setting up IDF: https://docs.espressif.com/projects/esp-idf/en/release-v4.4/get-started/index.html. Please complete all the steps on this page.

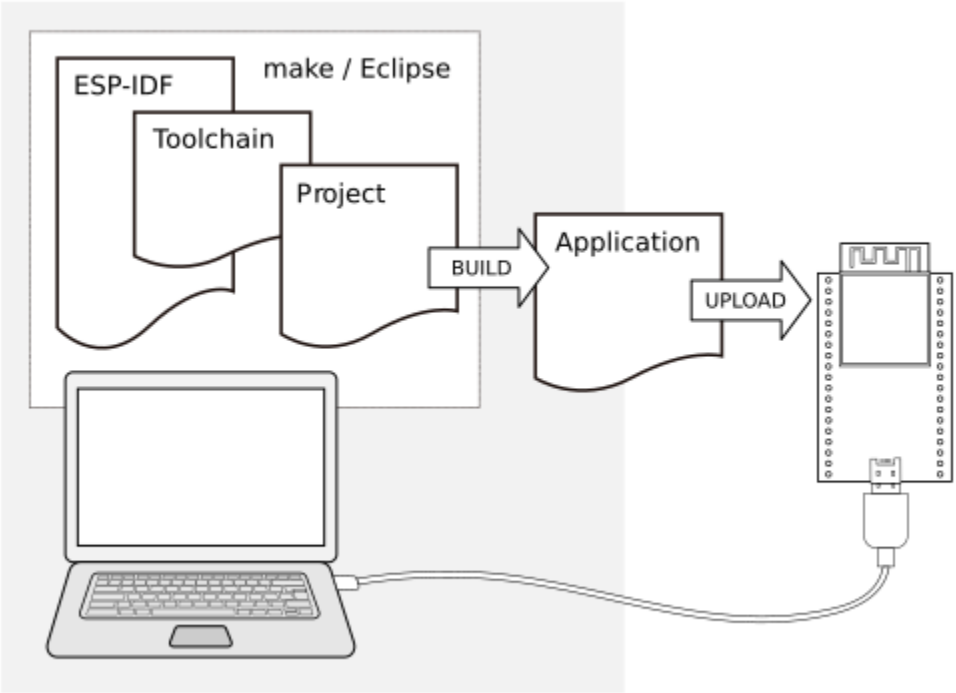

Fig. 1: Typical Developer Setup

Before proceeding, please ensure that you have setup your development host, and have built the first application as indicated in this page. Now that you have done that, let's look at some additional details about IDF.

2.2.2 IDF Details

The IDF has a component based design.

All the software in the IDF is available as components. The Operating System, the network stack, Wi-Fi drivers, middleware modules like the HTTP Server are all components within IDF.

This design allows you to use your own or third-party components that are built for ESP-IDF.

A developer typically builds *applications* against the IDF. The applications contain the business logic, any drivers for externally interfaced peripherals and the SDK configuration.

An application must contain one *main* component. This is the primary component that holds the application logic. The application may additionally include other components as may be desired. The application's *CMakeLists.txt/Makefile* defines the build instructions for the application. Additionally, an optional *sdkconfig.defaults* may be placed that picks up the default SDK configuration that should be selected for this application.

2.3 Getting ESP-Jumpstart

The ESP-Jumpstart repository contains a sequence of *applications* that we will use for this exercise. These applications build with the ESP-IDF that you have setup before. Let's get started by cloning the ESP-Jumpstart git repository https://github.com/espressif/esp-jumpstart.

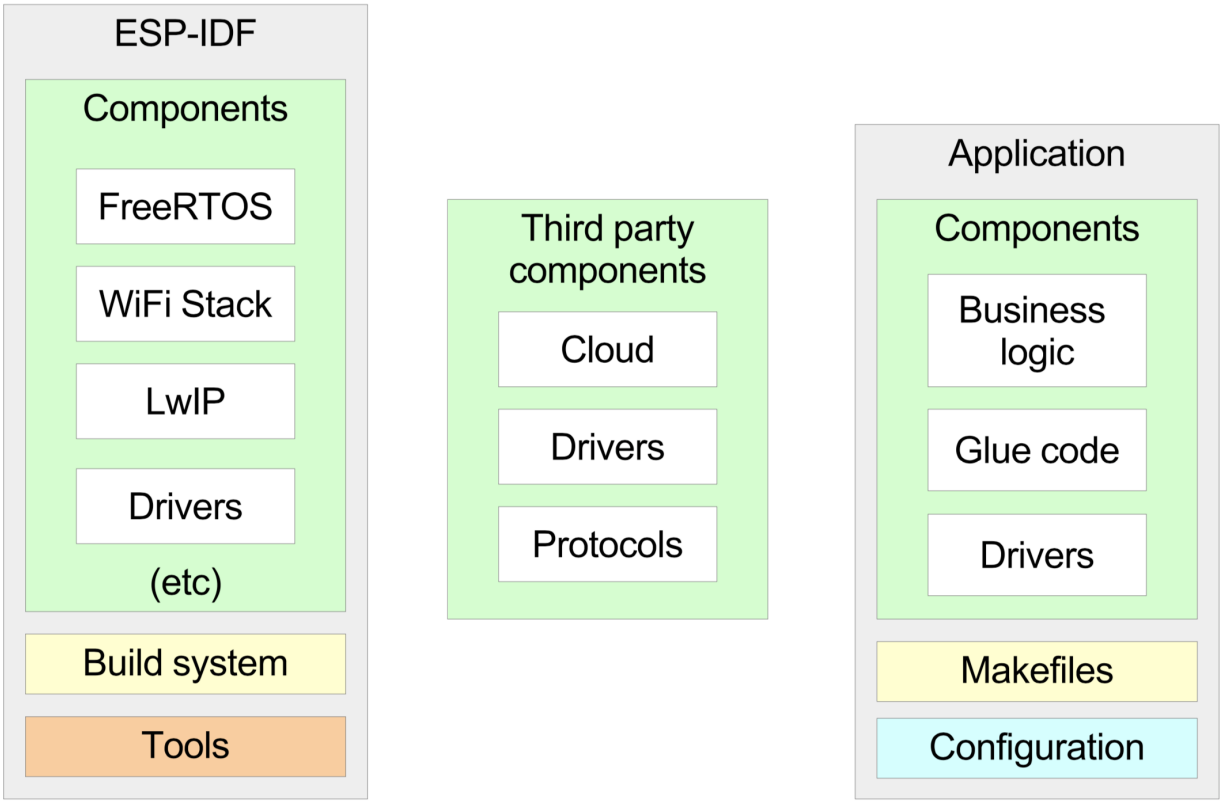

Fig. 2: Component Based Design

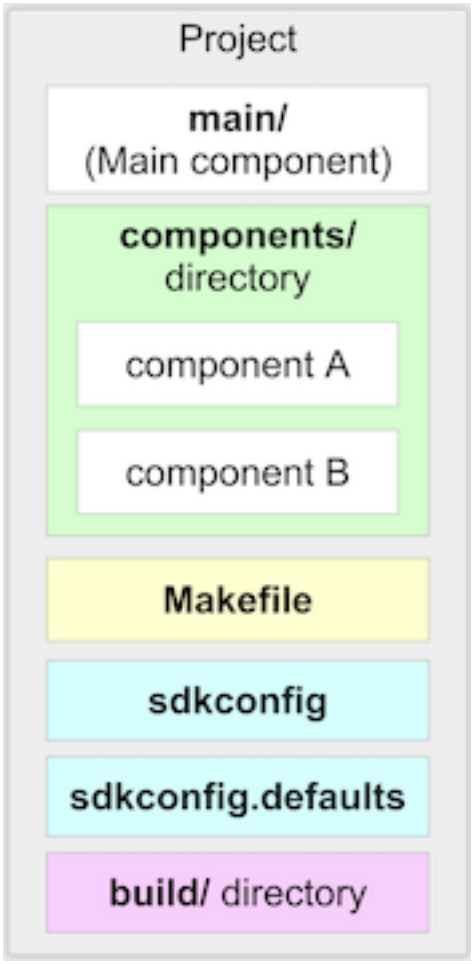

Fig. 3: Application's Structure

```
$ git clone --recursive https://github.com/espressif/esp-jumpstart
```

Since we are building a production-ready firmware here, we would want to base our development on a stable release of IDF. Currently, ESP-Jumpstart uses the stable version 4.4 of ESP-IDF. Let us first switch to that version of ESP-IDF.

```
$ cd esp-idf
$ git checkout -b release/v4.4 remotes/origin/release/v4.4
$ git submodule update --recursive
```

Now we build our first, *Hello World*, application from ESP-Jumpstart and flash it on to our development board. You should be already familiar with most of the steps below.

```
$ cd esp-jumpstart/1_hello_world
$ export ESPPORT=/dev/cu.SLAB_USBTOUART   # Or the correct device name for your setup
$ export ESPBAUD=921600
$ idf.py menuconfig
$ idf.py flash monitor
```

This will then build the entire SDK and the application. Once the build is successful, it will write the generated firmware to the device.

Once the flashing is successful, the device will reset and you will see the console output from this firmware.

2.3.1 For ESP8266 Users

Please make sure that the IDF_PATH is set to the path of ESP8266_RTOS_SDK. ESP8266 uses the branch platform/esp8266 of esp-jumpstart. Switch to this branch using the following command.

```
$ cd esp-jumpstart
$ git checkout -b platform/esp8266 origin/platform/esp8266
```

2.4 The Code

Now let's look at the code of the Hello World Application. It is only a few lines of code as shown below:

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"


void app_main()
{
    int i = 0;
    while (1) {
        printf("[%d] Hello world!\n", i);
        i++;
        vTaskDelay(5000 / portTICK_PERIOD_MS);
    }
}
```

The code is fairly simple. A few takeaways:

- The app_main() function is the application entry point. All applications begin execution at this point. This function gets called after the FreeRTOS kernel is already executing on both the cores of the ESP32. Once FreeRTOS

is initialised, it forks an application thread, called the main thread, on one of the cores. The app_main() function is called in this thread's context. The stack of the application thread can be configured through the SDK configuration.

- C library functions like printf(), strlen(), time() can be directly called. The IDF uses the newlib C library, which is a low-footprint implementation of the C library. Most of the category of functions of the C library like stdio, stdlib, string operations, math, time/timezones, file/directory operations are supported. Support for signals, locales, wchrs is not available. In our example above, we use the printf() function for printing to the console.
- FreeRTOS is the operating system powering both the cores. FreeRTOS (https://www.freertos.org) is a tiny kernel that provides mechanisms for task creation, inter-task communication (semaphores, message queues, mutexes), interrupts and timers. In our example above, we use the vTaskDelay function for putting the thread to sleep for 5 seconds. Details of the FreeRTOS APIs are available at: https://www.freertos.org/a00106.html

2.5 Progress so far

Now we have the basic development setup and process in place. We can build the code into executable firmware images. We can flash these images to a connected development board, and we can monitor the console to look at debug logs and messages generated by the firmware.

Let's now build a simple power outlet with ESP32.

CHAPTER 3

The Driver

[]

In this Chapter we will create a basic power outlet using the driver APIs of the ESP32. The power outlet will do the following:

- Provide a button that the user can press
- Toggle an output GPIO on every button press

For the scope of this chapter, we won't worry about 'connectivity' of this power outlet. That will follow in subsequent chapters. Here we will only focus on implementing the outlet functionality. You may refer to the *2_drivers/* directory of esp-jumpstart for looking at this code.

The code for the driver has been neatly isolated in the file *app_driver.c*. This way, later whenever you have to modify this application to adapt to your product, you could simply change the contents of this file to talk to your peripheral.

3.1 The Push Button

Let's first create a push-button. The Devkit-C development board has a button called 'boot' which is connected to GPIO 0. We will configure this button to be used to toggle the outlet's state.

3.1.1 The Code

The code for enabling this is shown as below:

```
#include <iot_button.h>

button_handle_t btn_handle=iot_button_create(JUMPSTART_BOARD_BUTTON_GPIO,
                                JUMPSTART_BOARD_BUTTON_ACTIVE_LEVEL);
iot_button_set_evt_cb(btn_handle, BUTTON_CB_RELEASE,
                            push_btn_cb, "RELEASE");
```

We use the *iot_button* module for implementing the button. First off we create the iot_button object. We specify the GPIO number and the active level of the GPIO to detect the button press. In the case of DevKit-C the *BUTTON_GPIO* is set to GPIO 0.

Then we register an event callback for the button, whenever the button is *released* the ***push_btn_cb*** function will be called. This function is called in the esp-timer thread's context. So do make sure that the default stack configured for the esp-timer thread is sufficient for your callback function.

The *push_btn_cb* code then is simply as shown below:

```
static void push_btn_cb(void* arg)
{
    static uint64_t previous;
    uint64_t current = xTaskGetTickCount();
    if ((current - previous) > DEBOUNCE_TIME) {
        previous = current;
        app_driver_set_state(!g_output_state);
    }
}
```

The *xTaskGetTickCount()* is a FreeRTOS function that provides the current tick counts. In the callback function, we make sure that the button press doesn't accidentally generate multiple events in a short duration of time. This is generally not what the end-user wants. (In the current case, we absorb all events generated within a 300 millisecond span, and call it a single event.) Finally, we call the function *app_driver_toggle_state()* which is responsible for toggling the output on or off.

3.2 The Output

Now we will configure a GPIO to act as the output of the power outlet. We will assert this GPIO on or off which would ideally trigger a relay to switch the output on or off.

3.2.1 The Code

First off we initialize the GPIO with the correct configuration as shown below:

```
gpio_config_t io_conf;
io_conf.mode = GPIO_MODE_OUTPUT;
io_conf.pull_up_en = 1;
io_conf.pin_bit_mask = ((uint64_t)1 << JUMPSTART_BOARD_OUTPUT_GPIO);

/* Configure the GPIO */
gpio_config(&io_conf);
```

In this example, we have chosen GPIO 27 to act as the output. We initialize the *gpio_config_t* structure with the settings to set this as a GPIO output with internal pull-up enabled.

```
/* Assert GPIO */
gpio_set_level(JUMPSTART_BOARD_OUTPUT_GPIO, target);
```

Finally, the state of the GPIO is set using the *gpio_set_level()* call.

3.3 Progress so far

With this, now we have a power outlet functionality enabled. Once you build and flash this firmware, every time the user presses the push-button the output from the ESP32 toggles on and off. As of now, this is not a connected outlet though.

As our next step, let's add Wi-Fi connectivity to this firmware.

CHAPTER 4

Wi-Fi Connection

[]

Let's now get this power outlet on a Wi-Fi network. In this Chapter we will connect to a hard-coded Wi-Fi network that is embedded within the device's firmware executable image. You may refer to the *3_wifi_connection/* directory of esp-jumpstart for looking at this code.

4.1 The Code

```
#include <esp_wifi.h>
#include <esp_event_loop.h>

tcpip_adapter_init();
esp_event_loop_init(event_handler, NULL);

wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
esp_wifi_init(&cfg);
esp_wifi_set_mode(WIFI_MODE_STA);

wifi_config_t wifi_config = {
    .sta = {
        .ssid = EXAMPLE_ESP_WIFI_SSID,
        .password = EXAMPLE_ESP_WIFI_PASS,
    },
 };
esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config);
esp_wifi_start();
```

In the above code:

- We initialize the TCP/IP stack with the *tcpip_adapter_init()* call
- Similarly, the Wi-Fi subsystem and its station interface is initialized with the calls to *esp_wifi_init()* and *esp_wifi_set_mode()*

- Finally, the hard-coded SSID and passphrase configuration of the target Wi-Fi network are configured and we start the station using a call to *esp_wifi_start()*

Wi-Fi is a protocol that can generate asynchronous events like connectivity lost, connection established, DHCP Address received etc. The event loop collects events from the TCP/IP Stack and the Wi-Fi subsystem. The call to *esp_event_loop_init()* initialises the event loop. The event loop delivers these events to the callback that is registered through the first parameter.

The asynchronous event handler that is registered with the event loop can be implemented as:

```
esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch(event->event_id) {
    case SYSTEM_EVENT_STA_START:
        esp_wifi_connect();
        break;
    case SYSTEM_EVENT_STA_GOT_IP:
        ESP_LOGI(TAG, "Connected with IP Address:%s",
             ip4addr_ntoa(&event->event_info.got_ip.ip_info.ip));
        break;
    case SYSTEM_EVENT_STA_DISCONNECTED:
        esp_wifi_connect();
        break;
    return ESP_OK;
}
```

The event handler current handles 3 events. When it receives an event *SYSTEM_EVENT_STA_START*, it asks the station interface to connect using the *esp_wifi_connect()* call. The same action is taken even when we receive a Wi-Fi disconnect event.

The event *SYSTEM_EVENT_STA_GOT_IP* is received when a DHCP IP address is obtained by ESP32. In this particular case, we only print the IP address on the console.

4.2 Progress so far

You can now modify the application to enter your Wi-Fi network's SSID and the passphrase. When you compile and flash this code on your development board, the ESP32 should connect to your Wi-Fi network and print the IP address on the console. The outlet's functionality of toggling the GPIO on pressing the push-button is, of course, also retained.

One problem with this approach is that the Wi-Fi settings are hard-coded into the firmware image. While this is ok for a hobby project, a product will require the end-user to dynamically configure this device with their settings. This is what we will look at in the next chapter.

CHAPTER 5

Network Configuration

[]

In the previous example, we had hard-coded the Wi-Fi credentials into the firmware. This obviously doesn't work for an end-user product.

In this step we will build a firmware such that the end-user can configure her Wi-Fi network's credentials into the device at run-time. Since a user's network credentials will be stored persistently on the device, we will also provide a *Reset to Factory* action where a user's configurations can be erased from the device. You may refer to the *4_network_config/* directory of esp-jumpstart for looking at this code.

5.1 Overview

As can be seen in this figure, in the network configuration stage, the end-user typically uses her smart-phone to *securely* configure her Wi-Fi credentials into your device. Once the devices acquires these credentials, it can then connect to her home Wi-Fi network.

There can be multiple channels through which your device can receive the Wi-Fi credentials. ESP-Jumpstart supports the following mechanisms:

- SoftAP
- Bluetooth Low Energy (BLE)

Each of these have their own pros and cons. There is no single way of doing this, some developers may pick one way, and some the other, depending upon what they value more.

5.1.1 SoftAP

In the SoftAP mechanism your outlet will launch its own temporary Wi-Fi Access Point. The user can then connect their smart-phones to this temporary Wi-Fi network. And then use this connection to transfer the Home Wi-Fi's credentials to the outlet. Many connected devices in the market today use this kind of mechanism. In this network configuration workflow, the user has to

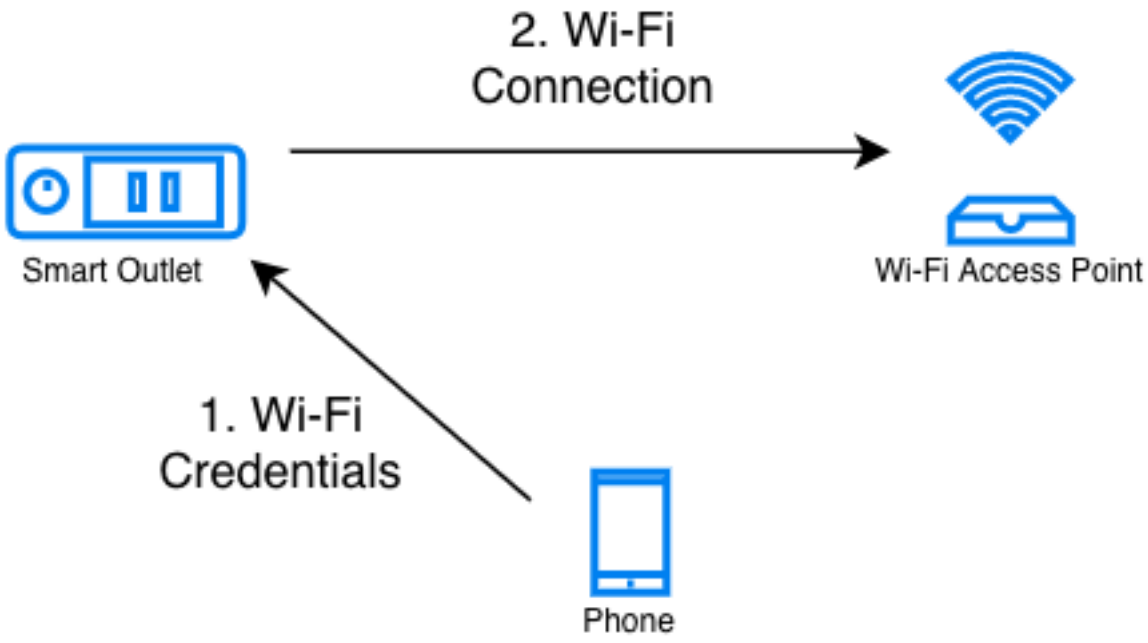

Fig. 1: Network Configuration Process

- switch their phone's Wi-Fi network to your outlet's temporary Wi-Fi network
- launch your phone application
- enter her home Wi-Fi credentials that will be then transferred to the outlet over the SoftAP connection

From a user experience perspective, the first step of this requires the user to change their phone's Wi-Fi network. This may be confusing to some users. Changing the Wi-Fi network programmatically through the phone application may not always be possible (iOS and some variants of Android don't allow phone apps to do this). But the advantage of this method is that it is very reliable (SoftAP being just Wi-Fi, is an established mechanism), and doesn't require a lot of additional code (footprint) in the device firmware.

5.1.2 BLE

In the Bluetooth Low Energy (BLE) method, your outlet will be doing a BLE advertisement. Phones in the vicinity can see this advertisement, and ask the user to do a BLE connection with your device. Then this network is used to transfer the credentials to the outlet. In this network configuration workflow, the user doesn't have to do the hard task of switching between Wi-Fi networks. Additionally, both iOS and Android allow phone application to scan for BLE devices in the vicinity and also connect to them through the app. This means a much smoother end-user experience.

One side-effect, though, of using the BLE based network configuration is that it also pulls in the code for Bluetooth. This means your flash requirement may be affected since your firmware size will increase. During the network configuration mode, BLE will also consume memory until the network configuration is complete.

5.2 Demo

Before getting into the details of the network configuration workflow, let us get a feel for how an end-user will configure the network using the provided application. You may refer to the *4_network_config/* directory of esp-jumpstart for trying this out.

- Go to the *4_network_config* application.
- Build, flash and load the application.
- By default, the firmware is launched in BLE provisioning mode.
- Install the companion phone application for network configuration. Android: https://play.google.com/store/apps/details?id=com.espressif.provble, iOS: https://apps.apple.com/in/app/esp-ble-provisioning/id1473590141.
- Launch the application and follow the wizard.
- If all goes well, your device would be connected to your Home Wi-Fi network.
- If you now reboot the device, it will NOT enter the network-configuration mode. Instead it will go and connect to the Wi-Fi network that is configured. This is the end product experience that we want.

5.2.1 For ESP8266 Users

ESP8266 supports only SoftAP based provisioning as it does not have Bluetooth. Use **ESP SoftAP Provisioning** app for provisioning.

5.3 Unified Provisioning

Espressif provides a **Unified Provisioning** module for assisting you with your network configuration. When this module is invoked from your firmware executable, the module takes care of managing all the state transitions (like starting/stopping the softAP/BLE interface, exchanging the credentials securely, storing them for subsequent use etc).

- Extensible Protocol: The protocol is completely flexible and it offers the ability for the developers to send custom configuration in the provisioning process. The data representation too is left to the application to decide.
- Transport Flexibility: The protocol can work on Wi-Fi (SoftAP + HTTP server) or on BLE as a transport protocol. The framework provides an ability to add support for any other transport easily as long as command-response behaviour can be supported on the transport.
- Security Scheme Flexibility: It's understood that each use-case may require different security scheme to secure the data that is exchanged in the provisioning process. Some applications may work with SoftAP that's WPA2 protected or BLE with "just-works" security. Or the applications may consider the transport to be insecure and may want application level security. The unified provisioning framework allows application to choose the security as deemed suitable.
- Compact Data Representation: The protocol uses Google Protocol Buffers as a data representation for session setup and Wi-Fi provisioning. They provide a compact data representation and ability to parse the data in multiple programming languages in native format. Please note that this data representation is not forced on application specific data and the developers may choose the representation of their choice.

The following components are offered as part of the provisioning infrastructure:

- **Unified Provisioning Specification:** A specification to *securely* transfer Wi-Fi credentials to the device, independent of the transport (SoftAP, BLE). More details can be found here: https://docs.espressif.com/projects/esp-idf/en/release-v4.4/api-reference/provisioning/provisioning.html.
- **IDF Components:** Software modules that implement this specification in the device firmware, available through ESP-IDF. More details can be found here: https://docs.espressif.com/projects/esp-idf/en/release-v4.4/api-reference/provisioning/wifi_provisioning.html.
- **Phone apps:** Android: BLE Provisioning(https://play.google.com/store/apps/details?id=com.espressif.provble), SoftAP Provisioning(https://play.google.com/store/apps/details?id=com.espressif.provsoftap). iOS:

BLE Provisioning(https://apps.apple.com/in/app/esp-ble-provisioning/id1473590141), SoftAP Provisioning(https://apps.apple.com/in/app/esp-softap-provisioning/id1474040630)

- **Phone App sources: Sources for the phone apps for** Android (https://github.com/espressif/esp-idf-provisioning-android) and iOS (https://github.com/espressif/esp-idf-provisioning-ios) are available for testing during your development, or for skinning with your brand's elements.

5.3.1 The Code

The code for invoking the unified provisioning through your firmware is shown below:

```
wifi_prov_mgr_init(config);
if (wifi_prov_mgr_is_provisioned(&provisioned) != ESP_OK) {
    return;
}

if (provisioned != true) {
    /* Start provisioning service */
    wifi_prov_mgr_start_provisioning(security, pop,
                service_name, service_key);
} else {
    /* Start the station */
    wifi_init_sta();
}
```

The *wifi_provisioning* component provides a wrapper over the unified provisioning interface. Some notes about the code above:

- The *wifi_prov_mgr_init* API initialises the Wi-Fi provisioning manager. This should be the first API call before invoking any other Wi-Fi provisioning APIs.
- The *wifi_prov_mgr_is_provisionined()* API checks whether Wi-Fi network credentials have already been configured or not. These are typically stored in a flash partition called the *NVS*. More about NVS later in this Chapter.
- If no Wi-Fi network credentials are available, the firmware launches the unified provisioning using the call *wifi_prov_mgr_start_provisioning()*. This API will take care of everything, specifically:
    1. It will start the SoftAP or BLE transport as configured
    2. It will enable the necessary advertisements using the Wi-Fi or BLE standards
    3. It will *securely* accept any network credentials from a phone application
    4. It will store these credentials, for future use, in the NVS
    5. Finally, it will deinitialise any components (SoftAP, BLE, HTTP Server etc) that were required by the unified provisioning mechanism. This ensures once provisioning is complete there is almost no memory overhead from the unified provisioning module.
- If a Wi-Fi network configuration was found in NVS, we directly start the Wi-Fi station interface using *wifi_init_sta()*.

These steps ensure that the firmware launches the unified provisioning module when no configuration is found, and if a configuration is available, then starts the Wi-Fi station interface.

The unified provisioning module also needs to know the state transitions of the Wi-Fi interface. Hence an additional call needs to be made from the event handler for taking care of this:

```
esp_err_t event_handler(void *ctx, system_event_t *event)
{
     wifi_prov_mgr_event_handler(ctx, event);

     switch(event->event_id) {
     case SYSTEM_EVENT_STA_START:
...
...
...
```

Configurable Options

In the code above, we first initialise the Wi-Fi Provisioning manager with a config structure, an example of which is as below:

```
/* Configuration for the provisioning manager */
wifi_prov_mgr_config_t config = {
    .scheme = wifi_prov_scheme_ble,
    .scheme_event_handler = WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM,
    .app_event_handler = {
        .event_cb = prov_event_handler,
        .user_data = NULL
    }
};
wifi_prov_mgr_init(config);
```

The paramters are as follows:

1. **Scheme:** What is the Provisioning Scheme that we want? SoftAP (wifi_prov_scheme_softap) or BLE (wifi_prov_scheme_ble)?
2. **Scheme Event Handler: Any default scheme specific event handler that you would like to choose.**
   Normally, this is used just to reclaim some memory after provisioning is done.
   - **WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BTDM - Free both classic BT and BLE (BTDM) m**
     Used when main application doesn't require Bluetooth at all
   - **WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BLE - Free only BLE memory.**
     Used when main application requires classic BT.
   - **WIFI_PROV_SCHEME_BLE_EVENT_HANDLER_FREE_BT - Free only classic BT.** Used when main application requires BLE. In this case freeing happens right when the manager is initialized.
   - **WIFI_PROV_EVENT_HANDLER_NONE Don't use any scheme specific handler.** Used when provisioning scheme is not BLE (i.e. SoftAP or Console), or when main application wants to handle the memory reclaiming on its own, or needs both BLE and classic BT to function.
3. **Application Event Handler:** Applications may want to use the provisioning events. A handler for that can be registered here. Any application specific user data can also be indicated here, which will be passed to the event handler.

After initialising, we have used the following call for starting the provisioning:

```
/* Start provisioning service */
wifi_prov_mgr_start_provisioning(security, pop, service_name, service_key);
```

Let us now look at the parameters, or the configuration options of this API:

1. **Security:** The unified provisioning module currently supports two security methods for transferring the credentials: *security0* and *security1*. Security0 uses no security for exchanging the credentials. This is primarily used for development purposes. Security1 uses secure communication which consists of secure handshake using *X25519* key exchange and proof of possession (pop) and *AES-CTR* for encryption/decryption of messages.
2. **Proof of Possession:** When a user brings in a new smart device, the device launches its provisioning network (BLE, SoftAP) for configuration. How do you make sure that only the owner of the device configures the device and not their neighbours? This configurable option is for that. Please read the following subsection for more details about this option.
3. **Service Name:** When the user launches the network configuration app, the user will be presented with a list of unconfigured devices, in her vicinity. The service name is this name that will be visible to the user. You may choose a name that identifies your device conveniently (abc-thermostat). It is common practice to have some element in the service name that is unique or random. This helps in scenarios when there could be multiple unconfigured devices that the user is configuring at the same time. When the provisioning mode is SoftAP, the service name appears as the SSID of the temporary Wi-Fi access point. When the provisioning mode is BLE, this appears as the BLE device name.
4. **Service Key:** Service Key is an optional parameter, which, if used serves as a password to protect the transport from being accessed by unauthorized users. This is useful when the mode of transport is SoftAP and you want the temporary Wi-Fi access point to be password protected. When the provisioning mode is BLE, this option is ignored altogether as BLE uses "just-works" pairing method.

Proof of Possession

When a user brings in a new smart device, the device launches its provisioning network (BLE, SoftAP) for configuration. How do you make sure that only the owner of the device configures the device and not their neighbours?

Some products expect the user configuring the device to provide a proof that they really own (or posses) the device that they are configuring. The proof of possession can be provided by taking some physical action on the device, or by entering some unique random key that is pasted on the device's packaging box, or by displaying on a screen, if the device is equipped with one.

At manufacturing, every device can be programmed with a unique random key. This key could then be provided to the unified provisioning module as a proof of possession option. When the user configures the device using the phone application, the phone application transfers the proof of possession to the device. The unified provisioning module then validates that the proof of possession matches and then confirms the configuration.

5.3.2 Additional Details

More details about Unified provisioning and the Wi-Fi provisioning abstraction layer are available at: https://docs.espressif.com/projects/esp-idf/en/release-v4.4/api-reference/provisioning/provisioning.html and https://docs.espressif.com/projects/esp-idf/en/release-v4.4/api-reference/provisioning/wifi_provisioning.html

5.4 NVS: Persistent key-value store

In the Unified Provisioning section above, we mentioned in passing that the Wi-Fi credentials are stored in the NVS. The NVS is a software component that maintains a persistent storage of key-value pairs. Since the storage is persistent this information is available even across reboots and power shutdowns. The NVS uses a dedicated section of the flash to store this information.

The NVS is designed in such a manner so as to be resilient to metadata corruption across power loss events. It also takes care of wear-levelling of the flash by distributing the writes throughout the NVS partition.

Application developers can also use the NVS to store any additional data that you wish to maintain as part of your application firmware. Data types like integers, NULL-terminated strings and binary blobs can be stored in the NVS. This can be used to maintain any user configurations for your product. Simple APIs like the following can be used to read and write values to the NVS.

```
/* Store the value of key 'my_key' to NVS */
nvs_set_u32(nvs_handle, "my_key", chosen_value);

/* Read the value of key 'my_key' from NVS */
nvs_get_u32(nvs_handle, "my_key", &chosen_value);
```

5.4.1 Additional Details

More details about NVS are available at: https://docs.espressif.com/projects/esp-idf/en/release-v4.4/api-reference/storage/nvs_flash.html

5.5 Reset to Factory

Another common behaviour that is expected of products is *Reset to Factory Settings*. Once the user configuration is stored into the NVS as discussed above, reset to factory behaviour can be achieved by simply erasing the NVS partition.

Generally, this action is triggered by long-pressing a button available on the product. This can easily be configured using the *iot_button_()* functions

5.5.1 The Code

In the *4_network_config/* application, we use a long-press action of the same toggle push-button to configure the reset to factory behaviour.

```
/* Register 3 second press callback */
iot_button_add_on_press_cb(btn_handle, 3, button_press_3sec_cb, NULL);
```

This function makes the configuration such that the *button_press_3sec_cb()* function gets called whenever the button associated with the *btn_handle* is pressed and released for longer than 3 seconds. Remember we had initialised the *btn_handle* in Section *The Code*

The callback function can then be written as follows:

```
static void button_press_3sec_cb(void *arg)
{
    nvs_flash_erase();
    esp_restart();
}
```

This code basically erases all the contents of the NVS, and then triggers a restart. Since the NVS is now wiped, the next time the device boots-up it will go back into the unconfigured mode.

If you have loaded and configured the device with the *4_network_config/* application, you can see this in action and by pressing the toggle button for more than 3 seconds and then releasing it.

5.6 Progress so far

Now we have a smart outlet that the user can configure, through a phone app, to their home Wi-Fi network. Once configured, the outlet will keep connecting to this configured network. We also have the ability to erase these settings on a long-press of a push-button.

As of now, the outlet functionality and the connectivity functionality are separate. As our next step, let's control and monitor the state of the outlet (on/off) remotely.

CHAPTER 6

Remote Control (Cloud)

[]

The true potential for smart connected devices can be realised when the connectivity is used to control or monitor the device remotely, or through integration with other services. This is where the cloud communication comes into picture. In this Chapter, we will get the smart outlet connected to a cloud platform, and enable remote control and monitoring of the device.

Typically, this is achieved through either of the scenarios as shown in the figure.

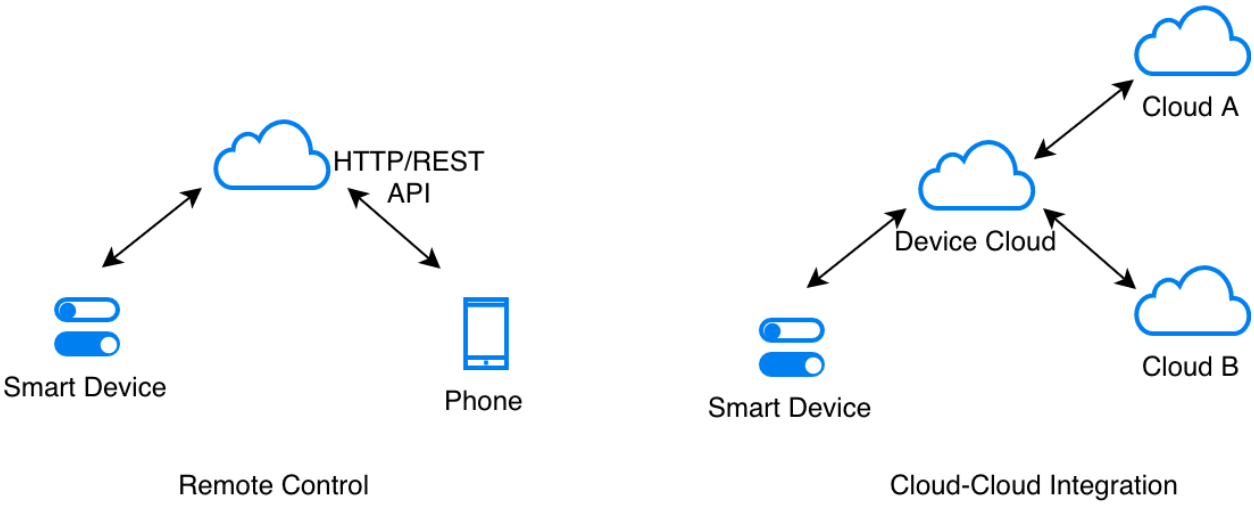

Fig. 1: Value of Cloud Connectivity

In most cases, once a device is connected to the cloud, the Device Cloud platforms expose the device control and monitoring through a RESTful web API. Authenticated clients, like smartphone apps can use these APIs to access the device remotely.

Additionally, integration with other clouds also helps in realising valuable use cases. For example, the device can be linked with a weather information system to automatically tune itself, or it can be linked to voice-assistant cloud

interfaces (like Alexa or Google Voice Assistant) to expose control through voice.

6.1 Security First

Before we get into the details about cloud connectivity, a few important words about security.

Connecting with any remote cloud infrastructure must always happen using TLS (Transport Layer Security). It is a standard and it takes care of ensuring that the communication stays secure. This is a transport layer protocol. Any higher-level protocols like HTTP, or MQTT can use TLS as the underlying transport. All reputable cloud vendors provide device services over TLS.

6.1.1 CA Certificates

One aspect of TLS is server validation using CA certificates. The TLS layer uses the CA certificate to validate that you are really talking to the server that you are supposed to talk to. For this validation to happen, your device must be pre-programmed with one or more valid and trusted CA certificates. The TLS layer will use these as trusted certificates and then validate the server based on these trusted certificates. Please refer to Section *Embedding Files in the Firmware* for more details about how the CA certificate can be made part of your firmware.

6.2 Embedding Files in the Firmware

At times, the firmware has to use certain files directly. Most commonly, in the case of CA certificates, that need to be embedded within the firmware for server validation.

The question is how do you make the entire contents of these files be part of your firmware image, and how do you access them within your firmware?

ESP-IDF provides a great mechanism for enabling this. The *CMakeLists.txt* file can be used to inform the build system that the contents of certain files should be embedded within firmware image. This can be enabled by adding the following line into your application's *CMakeLists.txt* file.

```
target_add_binary_data(${COMPONENT_TARGET} "cloud_cfg/server.cert" TEXT)
```

If you are using Legacy GNU make based build system instead of cmake, the same can be enabled by adding the following line into your application's *component.mk* file.

```
COMPONENT_EMBED_TXTFILES := cloud_cfg/server.cert
```

In the above example, the build system will make the file *cloud_cfg/server.cert* be part of the firmware. The contents of this file are in the firmware's address space and can be directly accessed as follows:

```
extern const uint8_t certificate_pem_crt_start[] asm("_binary_server_cert_start");
extern const uint8_t certificate_pem_crt_end[] asm("_binary_server_cert_end");
```

The file can then be accessed using these start and end pointers.

6.3 AWS IoT

In this section we will take AWS IoT as an example and connect the device to this cloud.

6.3.1 Quick Setup

Feel free to skip this sub-section, if you already have a registered account with a cloud platform.

As a way for you to try this functionality out, we have created a web-page that allows you to quickly connect a device to the AWS IoT cloud platform. This page will create a set of credentials for your device, that your device can use to authenticate with the cloud. The credentials will stay valid for a duration of 14 days, which gives you enough time to experiment with the remote control and OTA upgrades feature that are demonstrated in this and subsequent chapters. Beyond this duration, you can register with AWS IoT for a cloud account yourself and then use that cloud in your production.

You can create credentials for your device by:

1. Visit the following URL: https://espressif.github.io/esp-jumpstart/
2. Enter your email address to which the device credentials should be mailed
3. You will receive an email that contains the device credentials that should be programmed on your device

6.3.2 Demo

By now, you should have the following items ready to get your device to start talking with AWS IoT:

1. A Device Private Key (a file)
2. A Device Certificate (a file)
3. A Device ID (a file)
4. A CA Certificate for the AWS-IoT service's domain name (a file)
5. An endpoint URL (a file)

Before getting into the details of the code, let us actually try to use the remote control for our device. You may refer to the *5_cloud/* directory of esp-jumpstart for trying this out.

To setup your AWS IoT example,

1. Go to the *5_cloud/* application
2. Copy the files (overwriting any previous files) as mentioned below: (Note that some email clients will rename the files and add a .txt extension to them. Please make sure that the downloaded files have names as expected below.)
    - The AWS CA Certificate to **5_cloud/main/cloud_cfg/server.cert**
    - The Device Private Key to **5_cloud/main/cloud_cfg/device.key**
    - The Device Certificate to **5_cloud/main/cloud_cfg/device.cert**
    - The Device ID to **5_cloud/main/cloud_cfg/deviceid.txt**
    - The Endpoint to **5_cloud/main/cloud_cfg/endpoint.txt**
3. Build, flash and load the firmware on your device

The device will now connect to the AWS IoT cloud platform and will notify the cloud of any state changes. The firmware will also fetch any updates to the state from the cloud and apply them locally.

6.3.3 Remote Control

For remote control, AWS IoT exposes a RESTful web API for all devices that connect to it. Phone applications can interact with this Web API to control and monitor the device. We will use cURL, a command-line utility, that can be used to simulate this phone app.

Using curl, we can then read the current state of the device by executing the following command on your Linux/Windows/Mac console:

```
curl --tlsv1.2 --cert cloud_cfg/device.cert \
      --key cloud_cfg/device.key    \
      https://a3orti3lw2padm-ats.iot.us-east-1.amazonaws.com:8443/things/<contents-
↪of-deviceid.txt-file>/shadow \
      | python -mjson.tool
```

In the above command, please copy paste the contents of the deviceid.txt file between *things* and *shadow*.

Note: AWS expects that access to a device state is only granted to entities that are authorised to do so. Hence in the command above, we use the *device.cert* and *device.key*, which are the same files that we have configured to be in the firmware. This ensures that we are authorised to access the device's state. In the production scenario, you must create separate authentication keys in the cloud for clients like this curl instance or phone applications, to access/modify the device state.

The device state can be modified as:

```
curl -d '{"state":{"desired":{"output":false}}}' \
     --tlsv1.2 --cert cloud_cfg/device.cert \
     --key cloud_cfg/device.key \
     https://a3orti3lw2padm-ats.iot.us-east-1.amazonaws.com:8443/things/<contents-of-
↪deviceid.txt-file>/shadow \
     | python -mjson.tool
```

This cURL command will generate an HTTP POST request, and sends the JSON data, as shown above, in the POST's body. This JSON data instructs AWS IoT to update the state of the device to false.

You can observe the corresponding change of state on the device whenever you change the state from cURL to true or false.

So that's how remote control is achieved. Let's now quickly talk about the code.

6.3.4 The Code

All the code for the cloud communication has been consolidated in the *cloud_aws.c* file. The structure of this file is similar to what the standard AWS IoT SDK expects.

The file uses our output driver's APIs, *app_driver_get_state()* and *app_driver_toggle_state()*, to fetch and modify the device state respectively.

The AWS IoT requires 3 files to be embedded within your firmware:

- The AWS CA Certificate **5_cloud/main/cloud_cfg/server.cert**
- The Device Private Key **5_cloud/main/cloud_cfg/device.key**
- The Device Certificate **5_cloud/main/cloud_cfg/device.cert**

The application uses the mechanism as shown in Section *Embedding Files in the Firmware* for embedding this within the firmware.

6.4 Progress so far

With this application we finally tie the functionality of the device (outlet power toggle) to network connectivity. Connecting it to the cloud makes it now accessible to be controlled and monitored over the network. We also looked at the security aspects that we must consider before connecting to any remote/cloud service.

As our next step, let's look at one of the most common requirements of a connected device, the over-the-air (OTA) firmware upgrade.

CHAPTER 7

Firmware Upgrades

[]

Before we discuss firmware upgrades, one pertinent topic that needs to be discussed is the flash partitions.

7.1 Flash Partitions

The ESP-IDF framework divides the flash into multiple logical partitions for storing various components. The typical way this is done is shown in the figure.

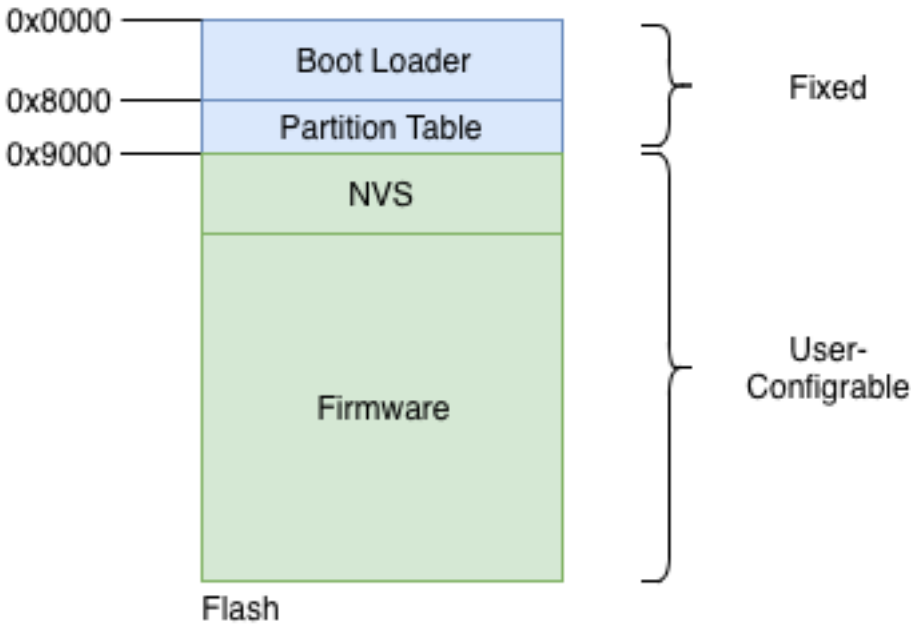

Fig. 1: Flash Partitions Structure

As can be seen, the structure is static upto flash address 0x9000. The first part of the flash contains the second-stage bootloader, which is immediately followed by the partition table. The partition table then stores how the rest of the flash should be interpreted. Typically an installation will have at-least 1 NVS partition and 1 firmware partition.

7.2 OTA Mechanism

For firmware upgrades, an active-passive partition scheme is used. Two flash partitions are reserved for the 'firmware' component, as shown in the figure. The OTA Data partition remembers which of these is the active partition.

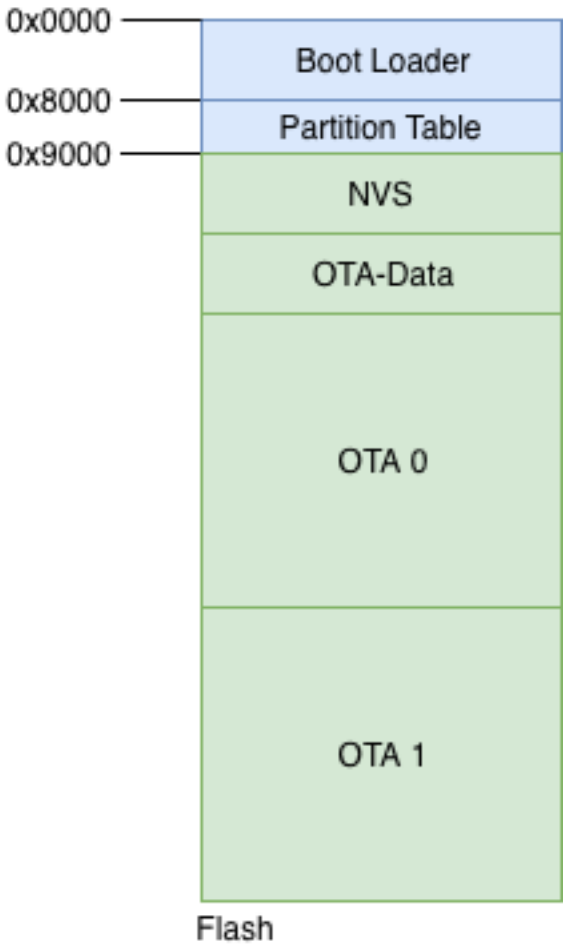

Fig. 2: OTA Flash Partitions

The typical state changes that happen across the OTA firmware upgrade workflow are as shown in the figure.

- Step 0: OTA 0 is the active firmware. The OTA data partition stores this information as can be seen.
- Step 1: The firmware upgrade process begins. The passive partition is identified, erased and new firmware is being written to the OTA 1 partition.
- Step 2: The firmware upgrade is completely written and verification is in-progress.
- Step 3: The firmware upgrade is successful, the OTA data partition is updated to indicate that OTA 1 is now the active partition. On the next boot-up the firmware from this partition will boot.

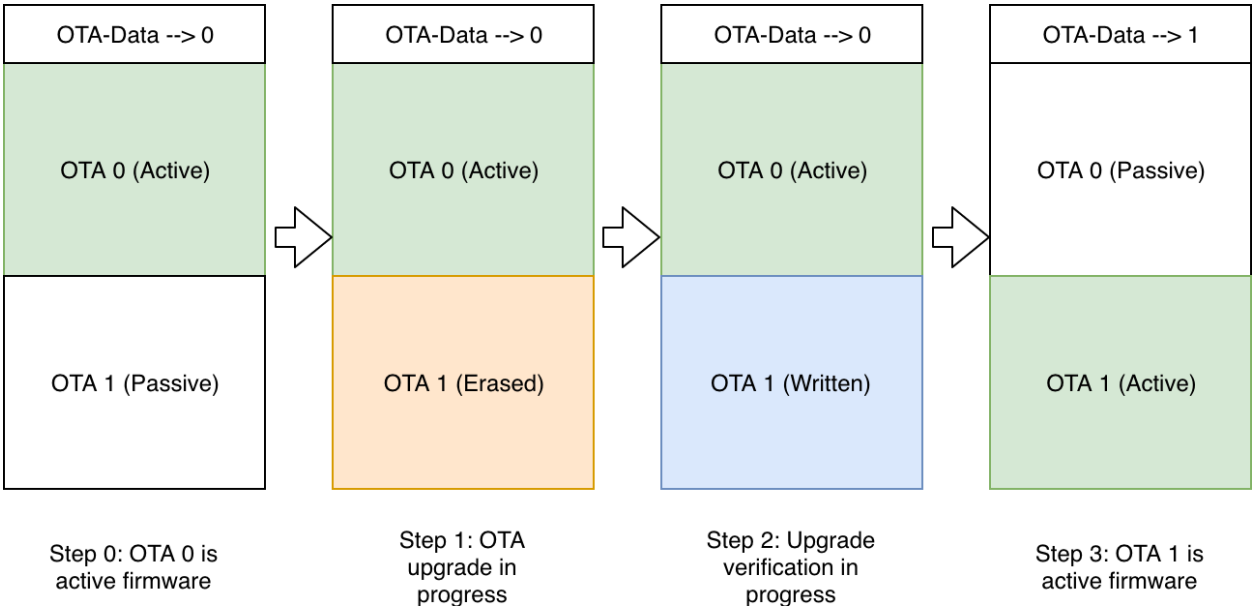

Fig. 3: Firmware Upgrade Flow

7.2.1 Updating the Flash Partitions

So how exactly do we instruct the IDF to create a partition table that has this OTA-Data partition and the 2 partitions for storing the firmware?

This can be achieved by creating a partitions file. This is a simple CSV (Comma Separated Values) file that instructs IDF what are the partitions that we want, what should be their size and how should they be placed.

The partitions file that is used for this example is shown below:

```
# Name,   Type, SubType, Offset,  Size, Flags
# Note: if you change the phy_init or app partition offset
# make sure to change the offset in Kconfig.projbuild
nvs,      data, nvs,     ,        0x6000,
otadata,  data, ota,     ,        0x2000,
phy_init, data, phy,     ,        0x1000,
ota_0,    app,  ota_0,   ,        1600K,
ota_1,    app,  ota_1,   ,        1600K,
```

The above partitions file instructs the IDF to create partitions: NVS, OTA-Data, OTA 0 and OTA 1, and it also specifies the sizes for each of these.

Once we create this partition file, we should instruct IDF to use this custom partitioning mechanism, over its default mechanism. This can be enabled by updating the SDK configuration. In the case of our application right now, this setting has already been activated in the *6_ota/sdkconfig.defaults* file. Hence you don't have to do any extra step for activating this.

But should you wish to use a different partitions file, or update the offset of the primary firmware, you should modify this setting. This can be done by executing the *idf.py menuconfig* command, and then configuring correct options in *menuconfig -> Partition Table*.

7.2.2 For ESP8266 Users

If your ESP8266 board has a smaller 2MB flash, please use this partition table.

```
# Name,   Type, SubType, Offset,  Size, Flags
# Note: if you change the phy_init or app partition offset, make sure to change the␣
↪offset in Kconfig.projbuild
nvs,      data, nvs,     0x9000,   0x4000,
otadata,  data, ota,     0xd000,   0x2000,
phy_init, data, phy,     0xf000,   0x1000,
ota_0,    app,  ota_0,   0x10000,  0xC5000,
ota_1,    app,  ota_1,   0x110000, 0xC5000,
```

7.3 The Code

Now let's check the code for actually performing the firmware upgrade.

```
esp_http_client_config_t config = {
    .url = url,
    .cert_pem = (char *)upgrade_server_cert_pem_start,
};
esp_err_t ret = esp_https_ota(&config);
```

- The *esp_http_client_config_t* structure is used to define the OTA upgrade source. This includes the URL that should be upgraded from, and also the CA certificate for validating the server from which the upgrade should be fetched. Please note that it is quite critical to ensure the validation of the CA certificate as mentioned in the Section *Security First*.
- The API *esp_https_ota()* is then executed which initiates the firmware upgrade. When the firmware upgrade process is successful (or fails), this API returns with the appropriate error code.
- By default, we have added the GitHub's CA certificate for the firmware upgrade URL. This makes it easy for you to host your upgrade image on GitHub and try out the upgrades. Ideally, you will install the CA certificate of the appropriate server from where you will download the upgrade image.

7.4 Send Firmware Upgrade URL

The open question is how does the device receive the upgrade URL. The firmware upgrade command is typically different from the remote-control commands discussed in the earlier section. This is because the firmware upgrade is generally triggered by the device manufacturer for a batch or group of devices based on certain criteria.

For the sake of simplicity, we will use the same remote control infrastructure to pass the firmware upgrade URL command to the device. But note that in your production scenario, you will send this firmware upgrade URL using some other mechanism controlled through the cloud.

For quickly trying out firmware upgrades, we have a sample firmware image (of the 1_hello_world application) uploaded on GitHub. We can try to upgrade to this firmware image as follows:

```
curl -d '{"state":{"desired":{"ota_url":"https://raw.githubusercontent.com/wiki/
↪espressif/esp-jumpstart/images/hello-world.bin"}}}' \
        --tlsv1.2 --cert cloud_cfg/device.cert \
        --key cloud_cfg/device.key \
        https://a3orti3lw2padm-ats.iot.us-east-1.amazonaws.com:8443/things/<contents-
↪of-deviceid.txt-file>/shadow | python -mjson.tool
```

If you are using an ESP32C3 DevKit, change hello-world.bin to hello-world-c3m-idf5.bin.

After the firmware upgrade is successful, the device will now execute the Hello World firmware.

7.5 Progress So Far

With this firmware we enable a key feature of any smart connected device, the over-the-air firmware upgrade.

Our product firmware is almost ready to be go, but for the final considerations for maintaining unique device data. Let's wrap that up in the upcoming Chapter.

CHAPTER 8

Manufacturing

[]

While building IoT products, it is often the case that some unique information needs to be stored in each device.

For example, in our journey so far, you might have realised that some cloud platforms have certificate based authentication, and we have embedded the device certificate within the firmware itself. This is fine when we are developing with a single device, but what do we do when we have to create hundreds of thousands of such devices? In this section this is what we will look at.

You might remember the NVS partitions that we discussed in Section *NVS: Persistent key-value store*. This was used to store key-value pairs persistently into the flash. Because this is stored in the flash, this information was accessible even across device reboot events. Also remember that we implemented the *Reset to Factory* action in Section *The Code* by erasing the contents of this NVS.

We can use the similar NVS partition for storing per-device unique key-value pairs at manufacturing time. But we want that this unique information should not be erased across the *Reset to Factory* events. This can be facilitated by creating another NVS partition that is primarily used for storing such unique factory-programmed information. Since this partition is programmed at the factory, we will use this NVS partition as a read-only partition, only referring to it to read the unique values that were configured for us.

Thus we can reuse the same concept to store factory unique information.

8.1 Multiple NVS Partitions

We had looked at *Flash Partitions* in Section *Flash Partitions* while discussing firmware upgrades. In Section *Updating the Flash Partitions* we also looked at how the flash partitions can be modified. For this example, we will add this extra NVS partition that will store the unique factory settings, and call it *fctry*.

You can check this by looking at the file *7_mfg/partitions.csv*.

8.2 The Code

Now that this NVS partition is present, we can access it using the standard NVS APIs. The only thing though is that you need to instruct NVS to use this other NVS partition while performing its NVS operations. This can be done by initialising the NVS handle as follows:

```
#define MFG_PARTITION_NAME "fctry"
/* Error checks removed for brevity */
nvs_handle fctry_handle;
nvs_flash_init_partition(MFG_PARTITION_NAME);
nvs_open_from_partition(MFG_PARTITION_NAME, "mfg_ns",
                NVS_READWRITE, &fctry_handle);
```

Now the NVS get operations that are performed with the *fctry_handle* NVS handle will result in reading data from this factory NVS partition. For example,

```
nvs_get_str(fctry_handle, "serial_no", buf, &buflen);
```

So, we can now disable the code that embeds any certificates in the firmware itself, and instead, read them from the unique factory partition that is flashed for this device.

8.3 Generating the Factory Data

Now we are good to go from the firmware perspective. But we still need to identify some mechanism for generating the factory data that will be written to the *fctry* partition.

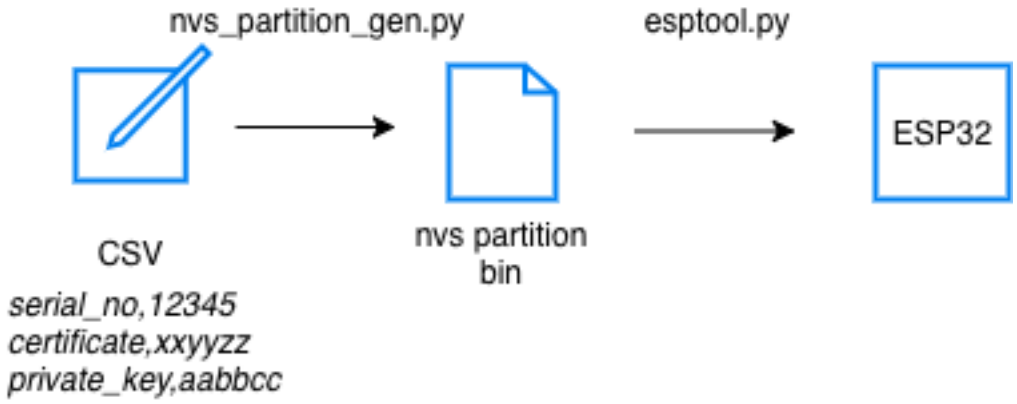

Fig. 1: Generating Factory Partition

The utility *components/nvs_flash/nvs_partition_generator/nvs_partition_gen.py* is used to generate an NVS image on the development host. This image can then be written to the flash into the location of the *fctry* partition.

This utility accepts a CSV file, and generates the image of an NVS partition from it. This CSV file stores the information about the key-value pairs that will be part of the generated NVS partition. At a factory, hundreds of thousands of these NVS partition images will be generated, one per device being manufactured, and then written to the respective devices uniquely.

A sample CSV file, called *mfg_config.csv* is available in the app. Each of its lines contains the values for the variables that are unique at the factory. Update them such that your unique settings are part of this CSV file.

The NVS partition can then be generated as:

```
$ python $IDF_PATH/components/nvs_flash/nvs_partition_generator/nvs_partition_gen.py␣
↪generate mfg_config.csv my_mfg.bin 0x6000
```

The my_mfg.bin file is the NVS partition data that can now be programmed into the device. You can use the following command to write this NVS partition to flash:

```
$ $IDF_PATH/components/esptool_py/esptool/esptool.py --port $ESPPORT write_flash␣
↪0x340000 my_mfg.bin
```

8.3.1 For ESP8266 Users

If your ESP8266 board has a smaller 2MB flash, use this command to flash my_mfg.bin.

```
$ $IDF_PATH/components/esptool_py/esptool/esptool.py --port $ESPPORT write_flash␣
↪0x1D5000 my_mfg.bin
```

Now if you boot up your firmware, it will work exactly as the firmware in the previous Chapter. But in this case, the firmware image itself is independent of the unique settings per device.

This allows you to create as many unique images as you want, and then flash them on the respective boards.

For more details about the unique factory partitions please refer to this link https://medium.com/the-esp-journal/building-products-creating-unique-factory-data-images-3f642832a7a3

8.4 Progress So Far

In this Chapter we looked at creating unique factory images per device, for contents that typically change across devices.

With this, we now have a fully functional, production-ready device firmware ready to ship out!

CHAPTER 9

Security Considerations

[]

Any discussion about connected devices is incomplete without discussion about the security considerations. Let us look at some of the security considerations that should be taken into account.

9.1 Securing Remote Communication

All communication with any entity outside of the device must be secured. Instead of reinventing the wheel, we recommend using the standard TLS for securing this communication. The ESP-IDF supports *mbedtls* that implements all the features of the TLS protocol.

All the code in the ESP-Jumpstart already includes this for remote communication. This section is applicable for any other remote connections that you wish to make from your firmware. You can skip to the next section if you are not using any other remote connections.

9.1.1 CA Certificates

The TLS layer uses trusted CA certificates to validate that the remote endpoint/server is really who it claims to be.

The *esp_tls* API accepts a CA certificate for performing server validation.

```
esp_tls_cfg_t cfg = {
    .cacert_pem_buf  = server_root_cert_pem_start,
    .cacert_pem_bytes = server_root_cert_pem_end - server_root_cert_pem_start,
};

struct esp_tls *tls = esp_tls_conn_http_new("https://www.example.com", &cfg);
```

If this parameter is not present, then the server validation check is skipped. It is strongly recommended that for all your TLS connections you specify the trusted CA certificate that can be used for server validation.

9.1.2 Obtaining CA Certificates

As can be seen from the code above, the trusted CA certificate that can validate your server must be programmed into your firmware. You can obtain the trusted CA certificates by using the following command:

```
$ openssl s_client -showcerts -connect www.example.com:443 < /dev/null
```

This command prints out a list of certificates. The last certificate from this list can be embedded in your device's firmware. Please refer to the Section *Embedding Files in the Firmware* for embedding files in your firmware.

9.2 Securing Physical Accesses

Now let us look at some of the features of ESP32 that protect from physically tampering with the device.

9.2.1 For ESP8266 Users

ESP8266 does not support the following features and thus cannot be secured from physical access.

9.2.2 Secure Boot

The secure boot support ensures that when the ESP32 executes any software from flash, that software is trusted and signed by a known entity. If even a single bit in the software bootloader and application firmware is modified, the firmware is not trusted, and the device will refuse to execute this untrusted code.

This is achieved by building a chain of trust from the hardware, to the software bootloader to the application firmware.

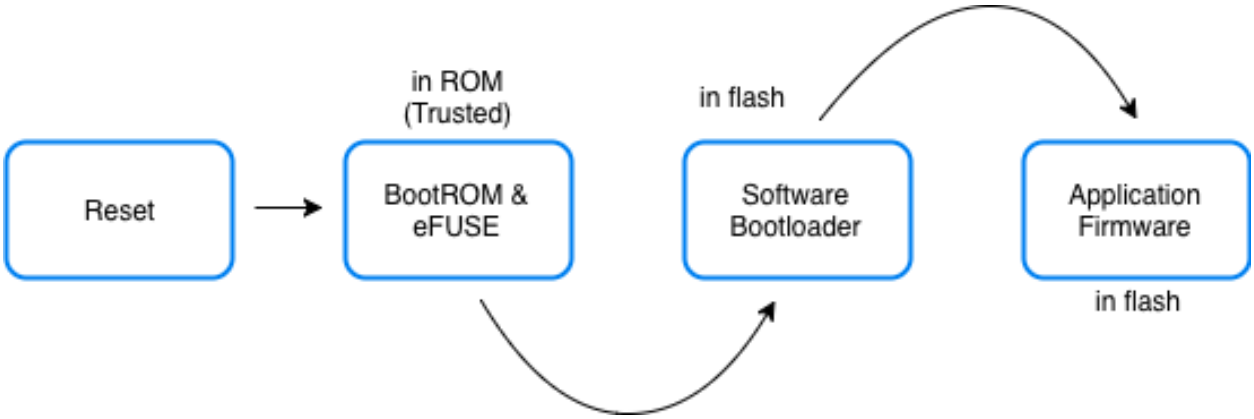

Fig. 1: Secure Boot

The process works as follows:

- During manufacturing
  - A secret key is programmed into the ESP32's eFUSE. Once programmed this key is protected from software read-out or writes
  - The software bootloader and the application firmware are signed with the correct keys and the signatures are appended to their images
  - The signed versions of the bootloader and firmware images are programmed into the ESP32's flash

- On power on reset of ESP32
  - The BootROM uses the secure key in the eFUSE to verify the software bootloader
  - Once the software bootload is verified, the BootROM loads and executes the software bootloader
  - The software bootloader verifies the signature of the application firmware
  - Once the application firmware is verified, the software bootloader loads and executes the application firmware

As you might have noticed, you will have to perform some additional steps for enabling secure boot on your devices. Please head over to the detailed information about Secure Boot (https://docs.espressif.com/projects/esp-idf/en/release-v4.4/security/secure-boot.htmlSecure Boot) to understand further.

9.2.3 Encrypted Flash

The flash encryption support ensures that any application firmware, that is stored in the flash of the ESP32, stays encrypted. This allows manufacturers to ship encrypted firmware in their devices.

When flash encryption is enabled, all memory-mapped read accesses to flash are transparently, and at-runtime, decrypted. The flash controller uses the AES key stored in the eFUSE to perform the AES decryption. This encryption key (in the eFUSE) is separate from the secure boot key mentioned above. This key can also be protected from software read-out and writes. Hence only the hardware can perform decryption of the flash contents.

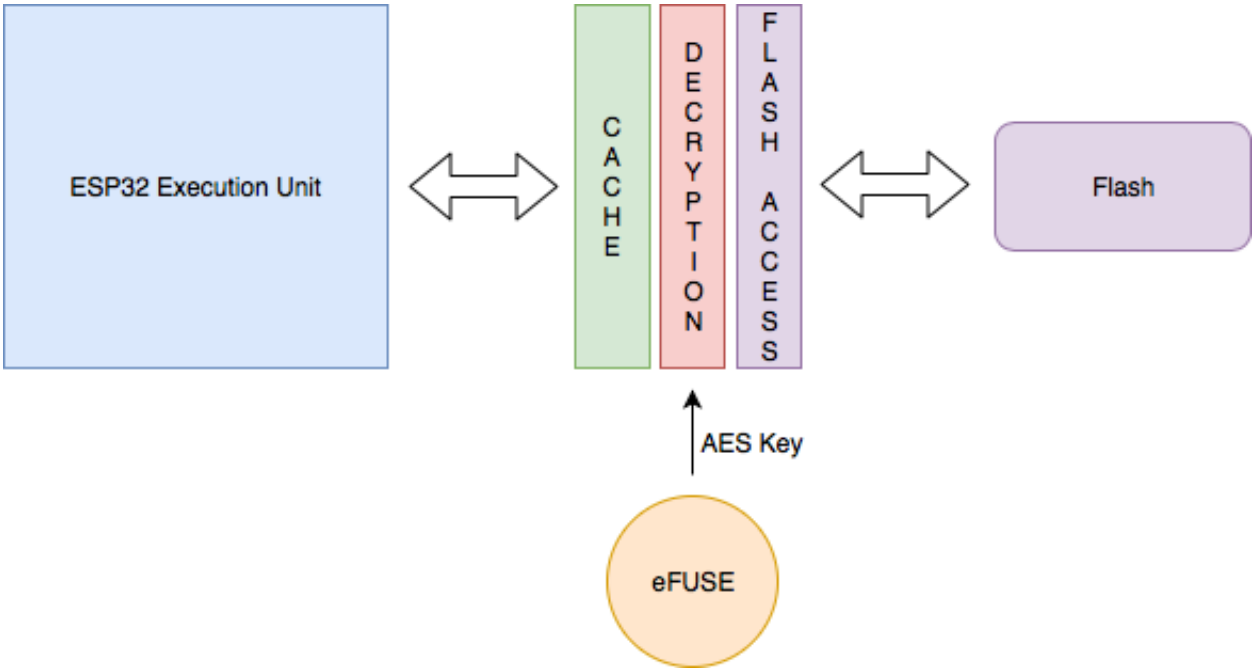

Fig. 2: Flash Encryption

For more information about enabling flash encryption, you can head over to additional documentation of Flash Encryption (https://docs.espressif.com/projects/esp-idf/en/release-v4.4/security/flash-encryption.html).

9.2.4 Encrypting NVS

The NVS partition has a different access pattern than the application firmware with more frequent writes, and with contents that depend on the user's preferences. Using the same encryption technique that is applicable for application

firmware isn't the best option for this scenario. Hence, the ESP-IDF provides a separate encryption mechanism for the NVS partition. This uses the industry-standard AES-XTS encryption that is recommended for protecting data at rest.

The process works as follows: The process works as follows:

- During manufacturing
  - Create a separate flash partition to store the encryption keys that will be used for NVS encryption
  - Mark this partition for flash-encryption
  - Use the *nvs_partition_gen.py* tool to generate the partition with random keys
  - Write this generated partition file into the newly created partition
- In the firmware
  - Call *nvs_flash_read_security_cfg()* API to read the encryption keys from the above partition and populate them in *nvs_sec_cfg_t*
  - Initialize the NVS flash partition using the APIs *nvs_flash_secure_init()* or *nvs_flash_secure_init_partition()*
  - Perform rest of the NVS operations as you normally would

For more information about using NVS encryption, you can head over to the additional documentation at https://docs.espressif.com/projects/esp-idf/en/release-v4.4/api-reference/storage/nvs_flash.html#nvs-encryption.