ESP-Jumpstart

2023 年 06 月 05 日

Contents

[English]

快速构建 ESP32 产品：从概念到生产的飞跃

Jumpstart Now

CHAPTER 1

简介

[English]

1.1 ESP-Jumpstart：快速构建 ESP32 产品

众所周知，固件开发并非易事，特别是用于量产的固件。开发人员不但需要面临各种决策上的问题并权衡多种选择，而且还需开发手机 app，并接入各家云服务商。现在，我们盛情推出 ESP-Jumpstart 示例项目，内含产品开发的完整步骤、最佳做法，并结合其他产品开发经验，助您快速启动基于 ESP32 的产品开发！

ESP-Jumpstart 项目专注于在 ESP32 上构建产品，展示了基于 ESP32 的完整产品开发流程。该项目分步介绍了一款真实产品的完整开发流程，即一款功能齐全、随时可推广的 “智能电源插座”。其中，每个步骤均为用户/开发人员的工作流提供指南/参考，并且都是基于 ESP32 的软件开发框架 ESP-IDF 进行开发。

本项目中的智能电源插座拥有一个实体按钮和一个 GPIO 输出端，可实现以下常见功能：

- 允许用户在家庭 Wi-Fi 网络环境中，通过手机 app（苹果/安卓）进行配置；
- 允许通过手机 app，打开/关闭 GPIO 输出端；
- 允许通过实体按钮，打开/关闭 GPIO 输出端；
- 允许通过云端，远程打开/关闭 GPIO 输出端；
- 支持 OTA 固件升级；
- 长按实体按钮，恢复出厂设置。

在实际开发中，您仅需将本项目中的智能电源插座替换为您的设备驱动程序（灯泡、洗衣机）即可。

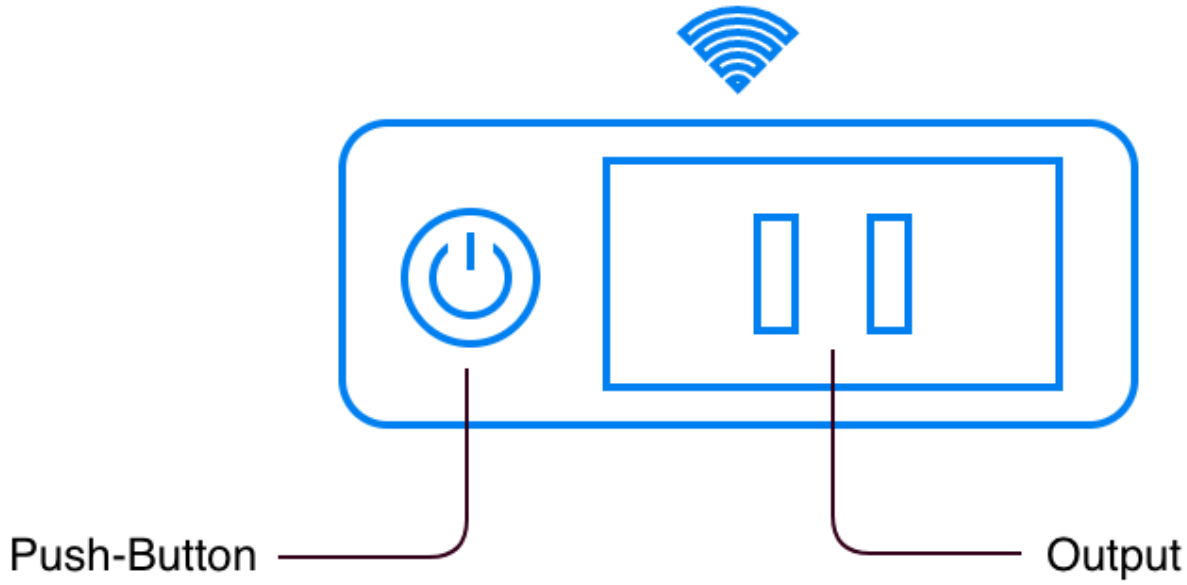

图 1: 智能电源插座

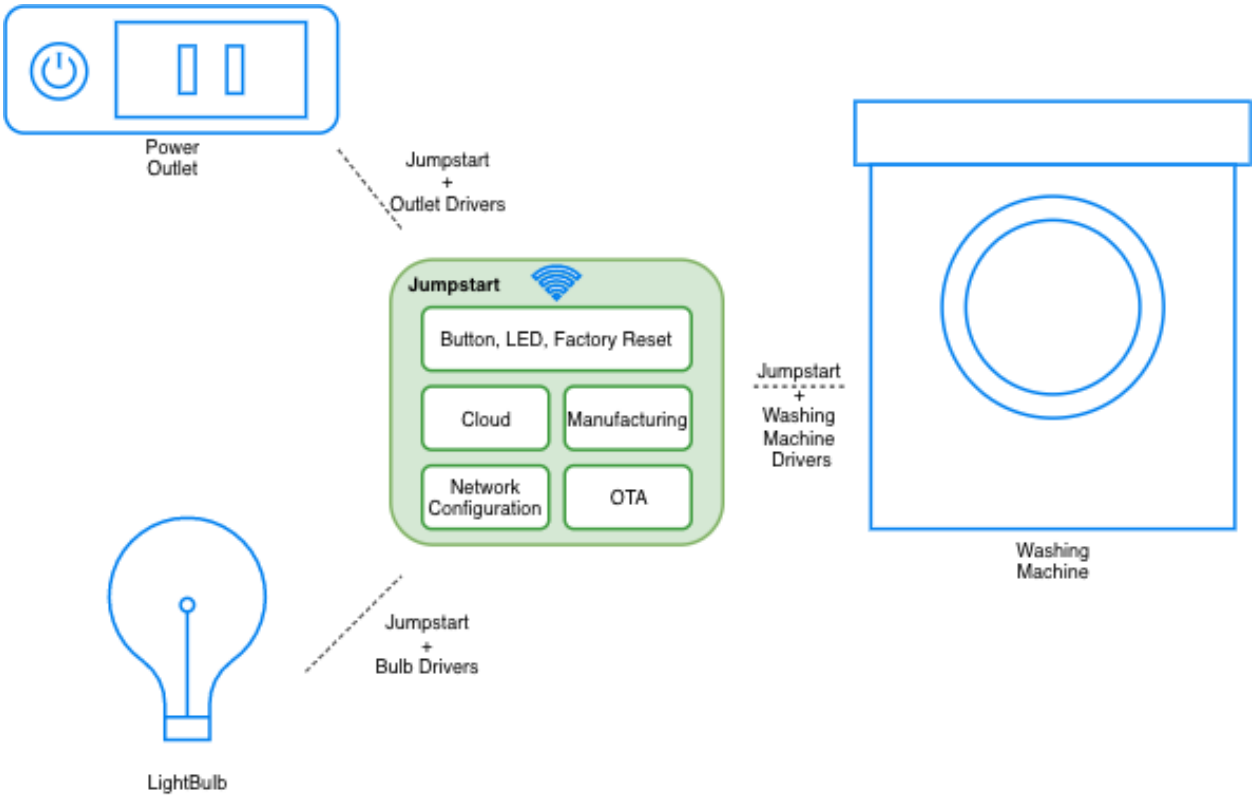

图 2: ESP-Jumpstart 适用性

准备工作：

- ESP32 开发板：ESP32-DevKitC，您也可以使用其他 ESP32 开发板；
- 用于开发的 PC（Windows、Linux 或 Mac OS）。

1.1.1 ESP8266 用户

准备工作：

- ESP8266 开发板：ESP8266-DevKitC，您也可以使用其他 ESP8266 开发板。
- ESP8266_RTOS_SDK 是乐鑫 ESP8266 的软件开发框架，ESP8266 开发过程中出现的 IDF 路径均指向 ESP8266_RTOS_SDK。
- 除了上面专门针对 ESP8266 用户的说明，其他部分 ESP32 和 ESP8266 通用。

1.2 其他

如果您已经熟悉乐鑫硬件和/或嵌入式系统，想寻找一些产品参考，而不需要量产环节，那您可以：

1. 直接使用 ESP-Jumpstart 中的最终应用程序；
2. 如果您没有云帐户，请按照亚马逊 *AWS IoT* 章节配置 AWS IOT 云；
3. 按照生成工厂数据 章节为设备特有的云证书创建量产配置文件，并将文件烧录至适当位置；
4. 正常构建、烧录并启动固件；
5. 使用 app（苹果/安卓）参考库构建自己的手机 app，或者直接参考统一配置 章节尝试这种解决方案；
6. 使用亚马逊 *AWS IoT* 章节中的指令进行远程控制；
7. 具备这些功能后，您就可以将这些功能适配到您的驱动程序上运行。

CHAPTER 2

入门指南

[English]

在本章中，我们将介绍 ESP32 的开发环境，并帮助您了解 ESP32 可用的开发工具和代码仓库。

2.1 开发过程概述

使用 ESP32 开发产品时，常用的开发环境如下图所示：

上图电脑，即开发主机可以是 Linux、Windows 或 MacOS 操作系统。ESP32 开发板通过 USB 连接到开发主机，开发主机上有 ESP-IDF (乐鑫 SDK)、编译器工具链和项目代码。主机先编译代码生成可执行文件，然后电脑上的工具把生成的文件烧到板子上，然后板子开始执行文件。最后你可以从主机查看日志。

2.2 ESP-IDF 介绍

ESP-IDF 是乐鑫为 ESP32 提供的物联网开发框架。

- ESP-IDF 包含一系列库及头文件，提供了基于 ESP32 构建软件项目所需的核心组件。
- ESP-IDF 还提供了开发和量产过程中最常用的工具及功能，例如：构建、烧录、调试和测量等。

2.2.1 设置 ESP-IDF

请参照 ESP-IDF 入门指南，按照步骤设置 ESP-IDF。注：请完成链接页面的所有步骤。

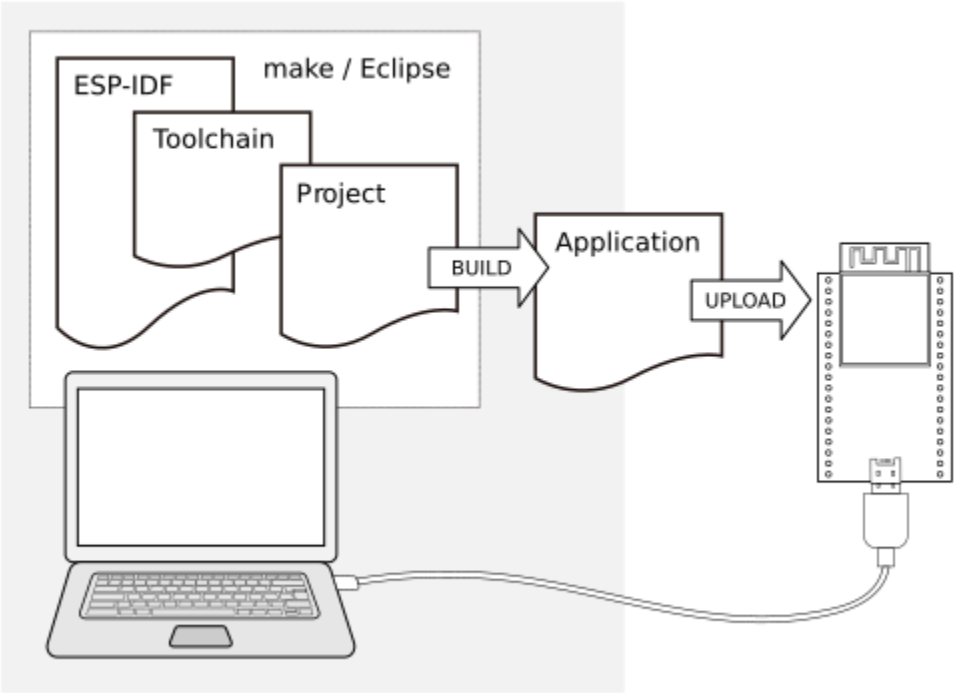

图 1: ESP32 产品开发过程

在进行下面步骤之前，请确认您已经正确设置了开发主机，并按照上面链接中的步骤构建了第一个应用程序。如果上面步骤已经完成，那让我们继续探索 ESP-IDF。

2.2.2 ESP-IDF 详解

ESP-IDF 采用了一种基于组件的架构：

ESP-IDF 中的所有软件均以“组件”的形式提供，比如操作系统、网络协议栈、Wi-Fi 驱动程序、以及 HTTP 服务器等中间件等等。

在这种基于“组件”的架构下，你可以轻松使用更多自己研发或第三方提供的组件。

开发人员通常借助 ESP-IDF 构建 *应用程序*，包含业务逻辑、外设驱动程序和 SDK 配置。

应用程序必须包含一个 *main* 组件，这是保存应用程序逻辑的主要组件。除此之外，应用程序根据自身需求还可以包含其他组件。应用程序的 *CMakeLists.txt/Makefile* 文件定义了应用程序的构建指令，此外，应用程序还包含一个可选的 *sdkconfig.defaults* 文件，用于存放应用程序默认的 SDK 配置。

2.3 获取 ESP-Jumpstart 库

ESP-Jumpstart 库包含了一系列由 ESP-IDF 构建的 *应用程序*，我们将在本次练习中使用这些应用程序。首先克隆 ESP-Jumpstart 库：

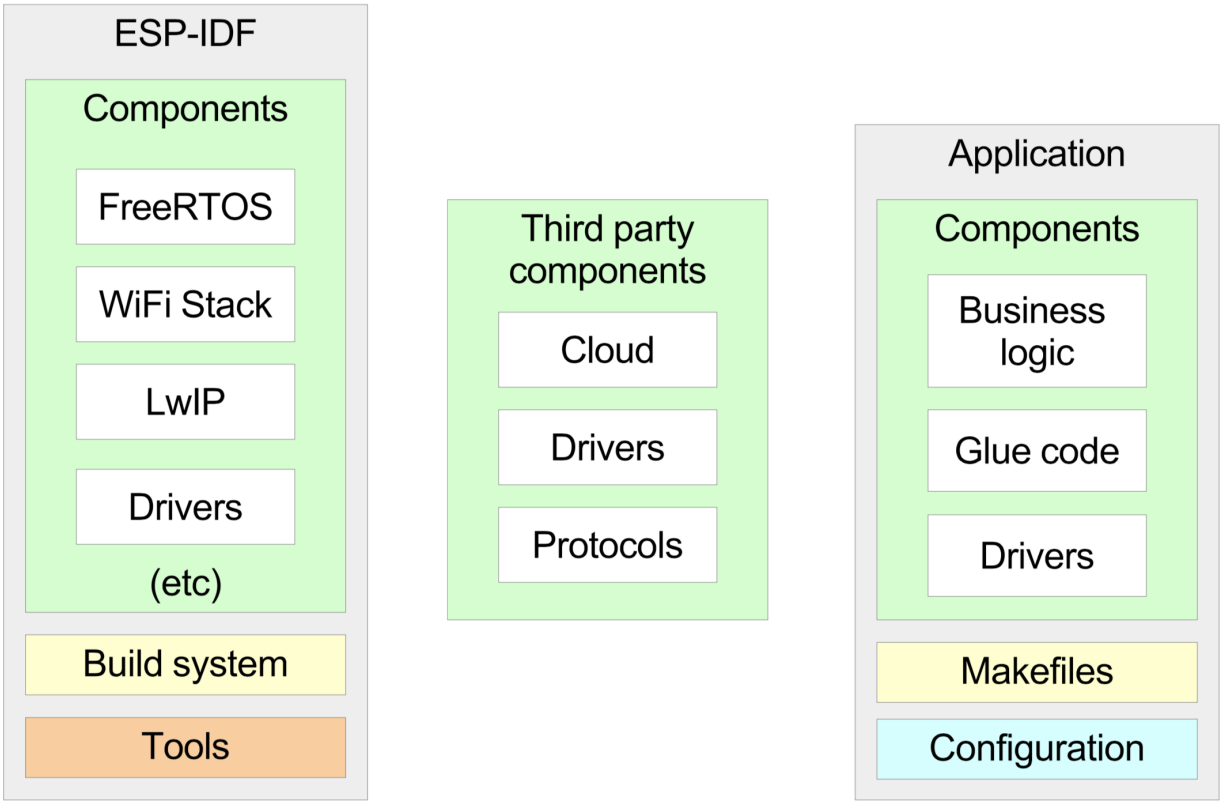
ESP-IDF
Components
FreeRTOS
WiFi Stack
LwIP
Drivers
(etc)
Build system
Tools
Third party components
Cloud
Drivers
Protocols
Application
Components
Business logic
Glue code
Drivers
Makefiles
Configuration

图 2: 组件设计

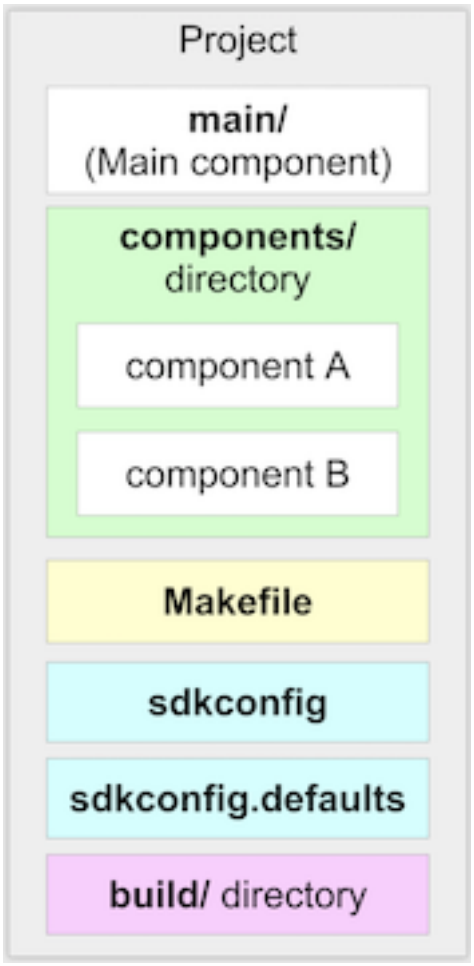
Project
main/
(Main component)
components/
directory
component A
component B
Makefile
sdkconfig
sdkconfig.defaults
build/ directory

图 3: 应用程序架构

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

我们将构建一个可用于量产的固件，因此选择使用 ESP-IDF 稳定版本进行开发。目前 ESP-Jumpstart 使用的是 ESP-IDF v4.4 稳定版本，请切换到这一版本。

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

现在，我们构建 ESP-Jumpstart 中的第一个应用程序 *Hello World*，并将其烧录到开发板上，具体步骤如下，相信您已经熟悉这些步骤：

```
$ 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
```

上面的步骤将构建整个 SDK 和应用程序。构建成功后，将会把生成的固件烧录到开发板。

烧录成功后，设备将重启。同时，你还可以在控制台看到该固件的输出。

2.3.1 ESP8266 用户

请确保已将 IDF_PATH 设置为 ESP8266_RTOS_SDK 的路径，并使用 esp-jumpstart 的 platform/esp8266 分支。请使用以下命令切换到该分支：

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

2.4 代码

现在，让我们研究一下 Hello World 应用程序的代码，只有如下几行：

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



void app_main()
{
```

(下页继续)

(续上页)

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

这组代码非常简单，下面是一些要点：

- app_main() 函数是应用程序入口点，所有应用程序都从这里开始执行。FreeRTOS 内核在 ESP32 双核上运行之后将调用此函数，FreeRTOS 一旦完成初始化，即将在 ESP32 的其中一个核上新建一个应用程序线程，称为主线程，并在这一线程中调用 app_main() 函数。应用程序线程的堆栈可以通过 SDK 的配置信息进行配置。
- printf()、strlen()、time() 等 C 库函数可以直接调用。IDF 使用 newlib C 标准库，newlib 是一个占用空间较低的 C 标准库，支持 stdio、stdlib、字符串操作、数学、时间/时区、文件/目录操作等 C 库中的大多数函数，不支持 signal、locale、wchr 等。在上面示例中，我们使用 printf() 函数将数据输出打印到控制台。
- FreeRTOS 是驱动 ESP32 双核的操作系统。FreeRTOS 是一个很小的内核，提供了任务创建、任务间通信（信号量、信息队列、互斥量）、中断和定时器等机制。在上面示例中，我们使用 vTaskDelay 函数让线程休眠 5 秒。有关 FreeRTOS API 的详细信息，请查看 FreeRTOS 文档。

2.5 未完待续

到现在为止，我们已经具备了基本的开发能力，可以进行编译代码、烧录固件、查看固件日志和消息等基本开发操作。

下一步，让我们用 ESP32 构建一个简单的电源插座。

CHAPTER 3

驱动程序

[English]

在本章中，我们将使用 ESP32 驱动程序 API 创建一个简单的电源插座应用。该电源插座：

- 包含一个实体按钮
- 按下按钮即可打开/关闭 GPIO 输出端

本章节只关注实现插座自身的功能，后续章节会具体介绍插座的连网功能。请参考 esp-jumpstart 项下的 *2_drivers/* 目录，查看相关代码。

此驱动程序的代码已被单独放入 *app_driver.c* 文件，因此如果后续需要修改此应用程序用于产品，您只需更改此文件内容，即可与外设通信。

3.1 实体按钮

首先，我们需要创建一个实体按钮。在 DevkitC 开发板上设有一个名为 Boot 的按钮，并连接至 GPIO 0。我们将配置此按钮，用于打开/关闭插座。

3.1.1 代码

实现此功能的代码如下：

```
#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");
```

我们使用 *iot_button* 模块来实现按钮功能。首先，创建 iot_button 对象，指定 GPIO 输出端及其有效电平用于检测按钮动作。DevKitC 开发板的 *BUTTON_GPIO* 设置为 GPIO 0。

然后我们为按钮注册事件回调函数，松开按钮时，就会在 esp-timer 线程中调用 *push_btn_cb* 函数。请确保为 esp-timer 线程配置的默认堆栈足以满足回调函数需求。

push_btn_cb 代码如下：

```
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);
    }
}
```

xTaskGetTickCount() 属于 FreeRTOS 函数，提供当前节拍事件计数。在回调函数中，需要确保按钮操作短时间内不会生成多个事件，否则会影响终端用户体验。在本示例中，所有 300 毫秒内生成的事件均视为一次有效事件。最后，调用 *app_driver_toggle_state()* 函数，用于打开或关闭输出端。

3.2 输出端

现在我们将配置 GPIO 作为电源插座的输出端。在理想情况下，打开或关闭此 GPIO 将触发继电器打开或关闭输出端。

3.2.1 代码

首先，使用以下配置初始化 GPIO。

```
gpio_config_t io_conf;
io_conf.mode = GPIO_MODE_OUTPUT;
io_conf.pull_up_en = 1;
```

(下页继续)

(续上页)

```
io_conf.pin_bit_mask = ((uint64_t)1 << JUMPSTART_BOARD_OUTPUT_GPIO);

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

在本示例中，选择 GPIO 27 用作输出端。使用上述设置初始化 *gpio_config_t* 结构，将其设置为 GPIO 输出端，内部上拉。

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

最后，使用 *gpio_set_level()* 设置 GPIO 状态。

3.3 未完待续

现在，我们已经实现了电源插座本身的插座功能。将此固件构建并烧录至设备后，用户每次按下按钮，ESP32 就会打开或关闭输出端。当然，目前该插座还无法连网。

下一步，我们将为固件增加 Wi-Fi 连接功能。

CHAPTER 4

Wi-Fi 连接

[English]

在本章中，我们要把这个电源插座连接到 Wi-Fi 网络。此 Wi-Fi 网络信息已嵌入到设备固件中，源代码可在 *3_wifi_connection/* 目录中查看。

4.1 代码

```
#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,
    },
```

(下页继续)

(续上页)

```
 };
esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config);
esp_wifi_start();
```

在上述代码中：

- 我们调用 *tcpip_adapter_init()* 来初始化 TCP/IP 堆栈；
- 调用 *esp_wifi_init()* 和 *esp_wifi_set_mode()* 来初始化 Wi-Fi 子系统及其 station 接口；
- 最后，使用嵌入的 SSID 和密码配置 Wi-Fi 网络，调用 *esp_wifi_start()* 启动 station 接口。

Wi-Fi 协议栈会发出断开连接、建立连接、获取到 IP 地址等异步事件。事件循环 (event loop) 从 TCP/IP 堆栈和 Wi-Fi 子系统收集事件，调用 *esp_event_loop_init()* 初始化 event loop，event loop 将这些事件传递给通过第一参数注册的回调函数。

异步事件处理程序在 Event Loop 注册，其实现方式如下：

```
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;
}
```

事件处理程序当前处理 3 个事件，当接收到 *SYSTEM_EVENT_STA_START* 事件后，要求 station 接口调用 *esp_wifi_connect()* 进行网络连接。收到 Wi-Fi 断开事件，也会要求 station 接口调用 *esp_wifi_connect()* 重新进行网络连接。

ESP32 接收到获取 IP 地址即相当于 *SYSTEM_EVENT_STA_GOT_IP* 事件发生，在这种情况下，我们只在控制台打印 IP 地址。

4.2 未完待续

现在您可以修改应用程序，输入 Wi-Fi 网络的 SSID 和密码。如果您已将该代码编译并烧录至开发板，ESP32 将连接到您设置的 Wi-Fi 网络，并在控制台上打印 IP 地址。当然，我们还保留了插座按下按钮触发 GPIO 的功能。

但这种方法有个弊端：Wi-Fi 配置被写死到了固件中。虽然这种方法对业余开发项目而言没有问题，但是如果用于量产，终端用户则希望自定义配置设备。这就是我们下一章要讨论的问题。

CHAPTER 5

SoftAP 配网和 BLE 配网

[English]

在上个示例中，我们把 Wi-Fi 信息（SSID 和 PASSWORD）直接嵌入到了固件中，但这显然不适用于终端产品。

因此在这一章节，我们将构建一个适用于终端用户的固件，允许用户在设备运行时，将其 Wi-Fi 信息配置到设备中。由于用户的网络信息将永久储存在设备中，所以我们另外提供了 恢复出厂设置功能，可擦除设备中储存的用户配置信息。如需查看相关代码，请前往 esp-jumpstart 项下的 *4_network_config/* 目录。

5.1 概述

如下图所示，在配网阶段，终端用户通常使用智能手机将 Wi-Fi 信息安全地配置到你的设备中。设备获取 Wi-Fi 信息后，就会连接到终端用户的家庭 Wi-Fi 网络中。

设备可通过多种途径接收 Wi-Fi 信息。ESP-Jumpstart 支持以下方式：

- SoftAP
- 低功耗蓝牙 (BLE)

这两种方式各有千秋，一些开发人员会选择这种，有些开发人员则会选择那种，这主要取决于自己的侧重点。

5.1.1 SoftAP 配网

在 SoftAP 模式下，插座会充当临时的 Wi-Fi 接入点。之后，用户可将智能手机连接到这个临时 Wi-Fi 网络中。连接创建完成后，即可将用户的家庭 Wi-Fi 信息传送到插座。当今市场上的多数连网设备均使用这一模

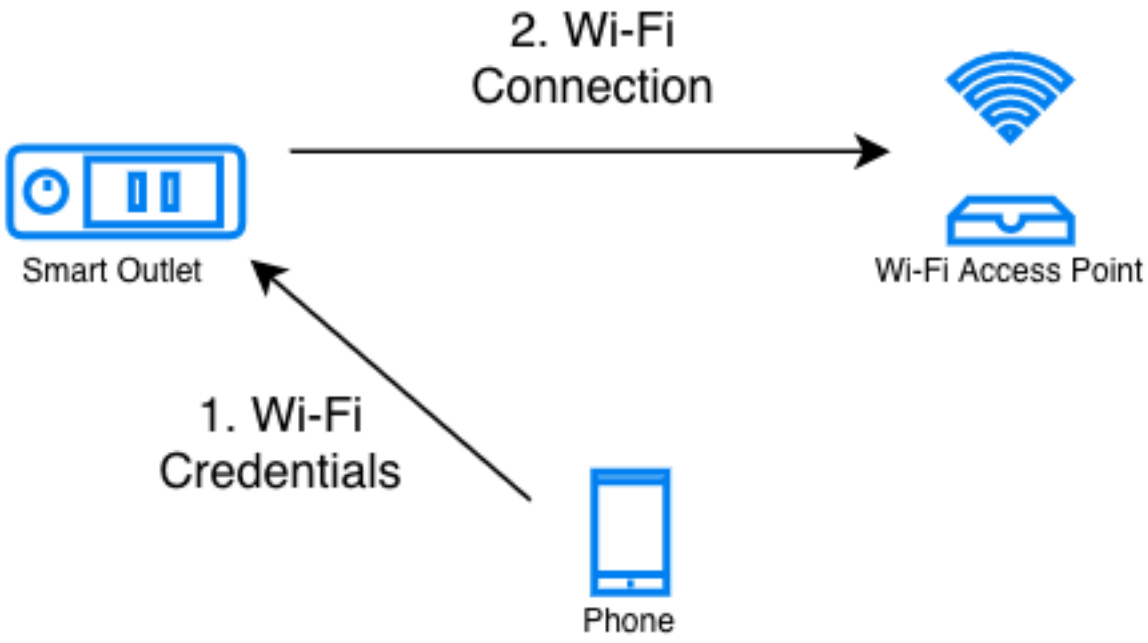

图 1: 配网过程

式，在这种配网过程中，用户需要：

- 将手机的 Wi-Fi 网络切换到插座创建的临时 Wi-Fi 网络；
- 使用你所提供的手机 app；
- 输入家庭 Wi-Fi 信息，并将该信息通过 SoftAP 连接传输到插座。

这种模式一开始就要求客户手动切换手机连接的 Wi-Fi 网络，这可能会让部分用户感到困惑，因此用户体验并不友好。而且，我们通常也很难直接将这个“切换网络”的过程写入代码，交由 app 自动完成（全部 iOS 和部分 Android 版本均并不允许手机 app 进行上述操作）。不过，这种模式的优势在于：第一，非常可靠（SoftAP 只作为 Wi-Fi 接入点，这已经是很成熟的技术了）；第二在于简洁（无需在设备固件中增加额外代码）。

5.1.2 BLE 配网

在 BLE 配网模式下，插座会首先进行 BLE 广播，而后附近的手机会收到该广播，并询问手机用户是否与该插座进行 BLE 连接。如选择创建 BLE 连接，手机即可将网络信息传输到插座。在这种配网过程中，用户无需切换 Wi-Fi，而且 iOS 和 Android 系统都支持手机 app 扫描并连接到周围的 BLE 设备。这样就可以大大提升终端用户的体验。

但是，使用 BLE 进行配网时，有一个缺点：此过程需要加入蓝牙相关代码。这就意味着你的固件大小会增加，因而会对 flash 提出更高要求。此外，在这种配网模式下，BLE 在配网结束前还会一直占用内存。

5.2 演示

在详细介绍网络配置流程之前，让我们先了解一下终端用户会如何使用我们提供的 APP 进行网络配置。请前往 esp-jumpstart 项下 *4_network_config/* 目录，查看详情：

- 进入 *4_network_config* 目录；
- 构建、烧录并载入上面的程序；
- 固件默认以 BLE 配网模式启动；
- 点击 应用下载页面，下载配套的手机 app，进行网络配置。请安装名为 **ble-sec1** 的最新版 app；
- 启动该 app，并按照 app 提供的操作向导进行操作；
- 如果一切顺利，设备将连接到家庭 Wi-Fi 网络；
- 如果此时重启设备，设备将不会进入网络配置模式，而是去连接已配置的 Wi-Fi 网络。这就是我们想要的终端产品体验。

5.2.1 ESP8266 用户

ESP8266 不具备蓝牙功能，因此现阶段仅支持 SoftAP 配网模式。ESP8266 用户请使用 **wifi-sec1** app 进行配置。

5.3 统一配置

乐鑫提供一种 **统一配置**模块，协助进行网络配置。当从可执行固件调用此模块时，模块将负责管理所有状态转换（如启动/停止 softAP/BLE 接口，安全交换网络信息，储存网络信息以备后续之用等）。统一配置模块具备以下优势：

- 协议可扩展：协议灵活，开发人员可在配置阶段发送自定义配置，数据含义由应用程序决定。
- 传输灵活：此协议作为一种传输协议，既支持 Wi-Fi （SoftAP + HTTP 服务器），又支持 BLE。该统一配置框架还可以轻松增加对其他传输形式的支持（该传输形式必须支持“命令-响应”行为）。
- 安全方案灵活：在配网阶段，为确保数据传输安全，每个用例对安全方案的需求可能不一样。有些应用需要使用 WPA2 加密的 SoftAP 传输方式或“just-works”加密的 BLE 传输模式；而有些应用为了确保数据传输安全，会选择应用级别安全机制。统一配置框架允许应用程序选择自己合适的安全模式。
- 紧凑的数据表示：本协议使用谷歌 Protocol Buffers 作为数据表示方式，用于会话设置和 Wi-Fi 配网。谷歌 Protocol Buffers 数据表示紧凑，并能够以原生格式解析多种编程语言中的数据。注意，这种数据表示并不专门用于应用程序特有的数据，开发人员可以选择自己的数据表示方式。

配网底层结构包含以下组件：

- **统一配置规范**：用于将 Wi-Fi 信息安全地传输到设备，与传输模式（SoftAP 或 BLE 模式）无关。请参考 统一配置相关文档，查看详细信息。

- **IDF 组件**：在设备固件中实现此规范的软件模块，可通过 ESP-IDF 获取。
- **手机库**：在 iOS 和 Android 系统中的参考实现，可以直接集成到现有的手机应用程序中。
- **手机应用程序参考**：乐鑫提供功能齐全的 Android 和 iOS 手机应用程序，可在开发时用于测试或排除你的手机品牌影响。

5.3.1 代码

通过固件调用统一配置的代码如下所示：

```
if (conn_mgr_prov_is_provisioned(&provisioned) != ESP_OK) {
    return;
}

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

conn_mgr_prov 组件是在统一配置接口上的一层封装，请注意：

- *conn_mgr_prov_is_provisionined()* API 用于检查 Wi-Fi 网络信息是否已经配置。网络信息通常储存在名为 *NVS* 的 flash 分区内，本章节后续部分会详细介绍 NVS（Non-volatile storage 非易失性存储器）。
- 如果没有可用的 Wi-Fi 网络信息，固件将使用 *conn_mgr_prov_start_provisioning()* API 启动统一配置。此 API 可以处理以下任务：
    1. 按照配置启动 SoftAP 或 BLE 传输；
    2. 使用 Wi-Fi 或 BLE 标准进行必要的广播；
    3. 安全接收手机应用程序传输过来的任意网络信息；
    4. 将上述网络信息储存在 NVS 中，以备后续之用；
    5. 最后，还可以对统一配置所需的所有组件（SoftAP、BLE、HTTP 等）撤销初始化，确保配网结束后，统一配置模块几乎不占用内存。
- 如果在 NVS 中发现 Wi-Fi 配网信息，即可使用 *wifi_init_sta()* API 直接启动 Wi-Fi station 接口。

上述步骤确保了在没有发现任何配网信息后，固件即可启用统一配置模块，如果有可用的配网信息，则会启动 Wi-Fi station 接口。

统一配置模块还需知道 Wi-Fi 接口的状态转换情况。因此，需要事件处理程序 (event handler) 发出调用请求，来处理这一问题：

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

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

配置选项

在上述代码中，我们用到了下面的 API 来调用统一配置接口：

```
/* Starting unified provisioning */
conn_mgr_prov_start_provisioning(prov_type,
           security, pop, service_name, service_key);
```

该 API 用到的参数，即该 API 的配置选项如下：

1. **安全性 (Security)**：统一配置模块当前支持两种用于传输网络信息的安全模式：*security0* 模式和 *security1* 模式。Security0 交换网络信息时，未采用任何安全措施，主要用于开发目的。Security1 使用椭圆曲线 *curve25519* 对密钥交换进行加密，然后使用 *AES-CTR* 对信道上交换的数据进行加密。

2. **传输机制 (Transport)**：开发人员可自主选择传输机制，用于网络配置。可供选择的传输机制有：SoftAP 和 BLE。

   - 此模块编写方式特殊，可根据开发人员的选择，仅将相关软件库添加到最终可执行映像。
   - 统一配置模块同时还将管理配网所需的状态转换和其他服务。

3. **所有权证明 (Proof of Possession)**：当用户拿来一个新的智能设备时，该设备将启动其配网功能（BLE 或 SoftAP）进行网络配置。如何才能确保只有设备所有者才能对该设备进行配置？而不是周围邻居也能轻易进行配置呢？此配置选项就是为了解决这个问题的。有关此选项的详细信息，请阅读以下小节。

4. **服务名称 (Service Name)**：用户在启动配网 app 后，将看到周围未配置设备的列表。这里的 **服务名称**就是客户将在列表中看到的你的设备的名称。因此，你在为设备命名时应注意取一个便于用户理解区别的名字，比如 “某某恒温器”。通常而言，您的 **服务名称**中应能够部分体现所提供服务（通过唯一或随机元素），方便用户在配网时轻松找到需要配置的设备。当采用 SoftAP 配网时，**服务名称**显示为临时 Wi-Fi 接入点的 SSID；当采用 BLE 配网时，服务名称则显示为 BLE 设备的名称。

5. **服务秘钥 (Service Key)**：服务秘钥为可选参数，可以用作密码，防止未经授权的用户访问传输数据。该参数一般仅在 SoftAP 配网模式下使用，可用于为临时 Wi-Fi 接入点提供密码保护。当采用 BLE 配

网时，BLE 会使用“just-works”配对方式，此选项可忽略。

所有权证明

当用户拿到一个新的智能设备，并启动设备进行配网时（无论是 SoftAP 配网或 BLE 配网），如何确保仅有设备所有者才能对该设备进行配置，而不是随便一个附近的邻居呢?

为了实现这个目的，有些产品会要求用户完成某些设置，以证明自己是此设备的所有者，比如要求用户对设备进行一些物理操作，比如按下某个按键；或输入设备包装盒或显示屏（如果配备显示屏的情况下）上的某些特有随机秘钥等。

在生产过程中，每个设备均将具备一个唯一的随机秘钥。接着，该秘钥将被提供给统一配网模块，用于所有权证明过程。之后，用户在使用手机 app 对设备进行配置时，app 会将该所有权证明传送给设备，然后由统一配网模块验证所有权证明是否匹配，进而确认是否进行配网操作。

5.3.2 补充信息

请参考 统一配置相关文档，查看详细信息。

5.4 NVS：永久储存键值对

在上文有关“统一配置模块”的介绍中，我们曾介绍说数据传输时的 Wi-Fi 网络信息是储存在 NVS 中的。NVS 是一种软件组件，用于永久储存键值对。由于 NVS 存储是永久性的，因此即便设备重启或断电，这些信息也不会丢失。NVS 在 flash 中有一个专门的分区来储存这些信息。

NVS 经过专门设计，不但可以防止设备断电带来的数据损坏影响，而且还可以通过将写入的内容分布到整个 NVS 分中以处理 flash 磨损的问题。

开发人员还可以使用 NVS 储存任何你希望与应用程序固件一起维护的数据，比如产品的用户配置信息。NVS 支持存储多种数据类型，比如整型、以 NULL 结尾的字符串和二进制大对象（BLOB）等。此外，NVS 的操作简便，仅通过以下两个 API 即可完成读写操作。

```
/* 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 补充信息

请参考 NVS 相关文档，查看详细信息.

5.5 恢复出厂设置

恢复出厂设置是产品另一个常见功能。如上述所述，只要将用户配置储存到 NVS 后，后续只需擦除 NVS 分区内的信息即可将设备恢复为出厂设置。通常而言，长按设备上的某个按钮即可恢复出厂设置。配置按钮功能也很简单，通过 *iot_button_()* 函数即可实现。

5.5.1 代码

在 *4_network_config/* 应用程序中，我们同样通过长按按钮动作来恢复出厂设置。

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

具体实现过程为：一旦与 *btn_handle* 关联的按钮被按下超过 3 秒，就会回调 *button_press_3sec_cb()* 函数。请注意，我们在代码 章节中对 *btn_handle* 进行了初始化。

回调函数示例如下：

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

这段代码的作用是擦除 NVS 的所有内容，然后触发设备重启。由于 NVS 内容已被清除，设备下次启动时将回到未配置状态。

这里，如果你已经通过 *4_network_config/* 加载并配置了你的设备，则可以尝试长按（3 秒以上）相关按钮，亲自查看恢复出厂设置的整个过程。

5.6 未完待续

截止目前，我们已经拥有了这样一款允许用户通过手机 app 连入家庭 Wi-Fi 网络的智能插座。一旦完成配置，该智能插座将总是尝试连接这个 Wi-Fi 网络。当然了，我们也可以通过长按按钮擦除现有配网信息，恢复出厂设置。

然而，到目前为止，插座自身功能与连网功能还是分开的。下一步，我们会将这两个功能结合起来，实现远程控制与监控插座状态，即打开/关闭。

CHAPTER 6

远程控制（云端）

[English]

智能硬件的真正优势在于通过联网对设备进行远程控制或监测，或者将联网功能集成于其他云服务。当集成联网功能与其他云服务时，需要用到云平台。在本章中，我们将智能插座连接到云平台，来实现对设备的远程控制和监测。

通常，我们可以通过图中所示的任一场景来实现对设备的远程控制和监测。

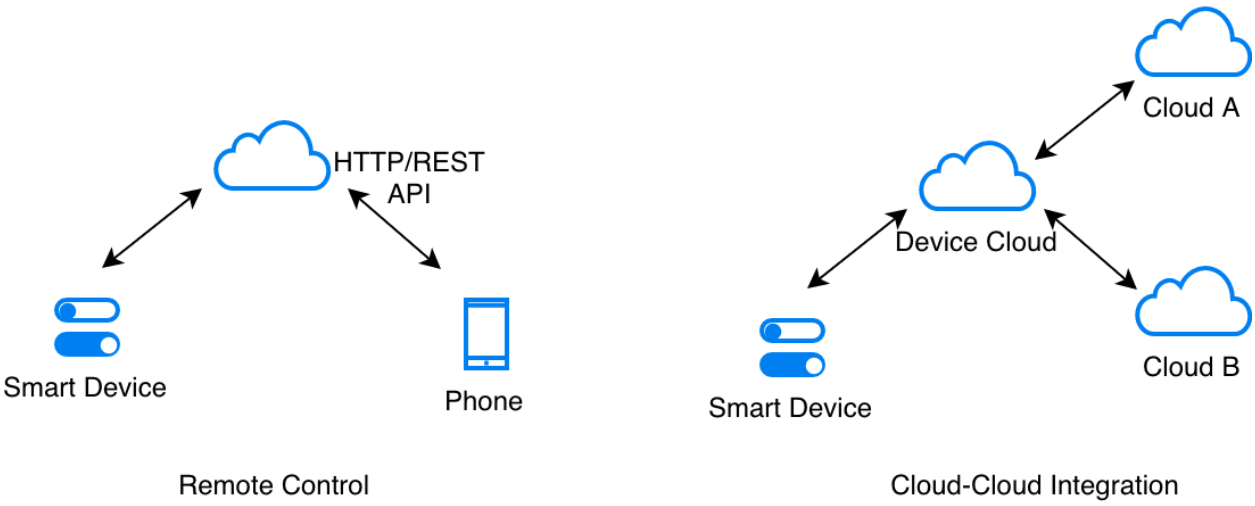

图 1: 云连接价值所在

在大多数情况下，一旦设备连接到云，云平台就会通过 Web API 公开设备的控制和监控权限。通过身份验证的客户端（如智能手机应用）可以使用这些 API 远程访问设备。

此外，通过集成其他云服务也能够实现许多有价值的用途。例如，设备可以与天气信息系统连接，根据天气情况进行自动调节，或者连接到语音助理云界面（如 Alexa 或谷歌语音助手），通过语音来进行控制。

6.1 安全性第一

在详细介绍云连接之前，让我们先了解一下安全性的重要知识。

连接远程云基础架构时，我们必须使用传输层安全性协议 (Transport Layer Security, TLS)。TLS 是一种安全标准，负责确保通信安全。同时，TLS 作为一种传输层协议，任何更高级别的协议（如 HTTP 或 MQTT）都可以将该协议用于底层传输。大多数云供应商都通过 TLS 提供设备服务。

6.1.1 CA 证书

TLS 协议内容之一是使用 CA 证书进行服务器验证，因此 TLS 层将使用 CA 证书来验证您是否在跟正确的服务器进行通信。要进行此验证，设备必须预先嵌入一个或多个有效且受信任的 CA 证书。TLS 层将这些证书视为可信证书，之后根据可信证书验证服务器。请参考在固件中嵌入文件 章节，查看如何将 CA 证书嵌入固件。

6.2 在固件中嵌入文件

固件有时需要直接使用某些文件。比如，我们经常需要将 CA 证书嵌入固件中以进行服务器验证。

但问题是如何将这些文件的内容全部嵌入固件中并进行访问？

ESP-IDF 提供了一个很好的实现方法，即使用 *CMakeLists.txt* 文件告诉构建系统哪些文件的内容需要嵌入到固件映像中。只要将下面的代码添加到 *CMakeLists.txt* 文件中即可。

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

如果你使用的是 Legacy GNU 构建系统，而非 CMake 构建系统，向您应用中的 component.mk 文件加入如下行同样可以使能该功能。

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

在上面的示例中，构建系统将 *cloud_cfg/server.cert* 文件嵌入到固件中，文件内容存放在固件地址空间内，可通过如下方式直接访问：

```
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");
```

然后我们可以使用开始和结束指针访问该文件。

6.3 亚马逊 AWS IoT

在本节中，我们将以亚马逊 AWS IoT 为例，将设备连接到该云端。

6.3.1 快速设置

如果您已经拥有云平台的注册帐户，您可以跳过此部分。

为了方便您试用该功能，我们创建了一个网页，该网页允许您快速将设备连接到 AWS IoT 云平台，这个网页为您的设备创建一组证书，设备可以使用这些证书进行身份验证。证书有效期为 14 天，您将有足够的时间来尝试本章和后续章节中演示的远程控制和 OTA 升级功能。试用期限过后，请自行在 AWS IoT 注册云帐户即可。

您可以通过以下方式为设备创建证书：

1. 点击 证书创建页面
2. 输入接收证书的电子邮件地址
3. 您会收到一封电子邮件，里面包含您将用到的设备证书。

6.3.2 演示

现在，您应该已经准备好了以下文件，可以将设备连接到 AWS IoT 平台了：

1. 设备私钥（文件）
2. 设备证书（文件）
3. 设备 ID（文件）
4. AWS IoT 域名的 CA 证书（文件）
5. 端点 URL（文件）

在详细了解代码之前，让我们先尝试一下设备远程控制。您可以参考 esp-jumpstart 项下 *5_cloud/* 目录。

请按照以下步骤，设置 AWS IoT 应用示例：

1. 进入 *5_cloud/* 程序
2. 复制如下文件，覆盖以前的所有文件。请注意，有些电子邮件客户端会将这些文件自动重命名，并为其添加.txt 扩展名。请确保下载文件的名称与下列一致：
   - 复制 AWS CA 到 **5_cloud/main/cloud_cfg/server.cert**
   - 复制设备私钥到 **5_cloud/main/cloud_cfg/device.key**
   - 复制设备证书到 **5_cloud/main/cloud_cfg/device.cert**
   - 复制设备 ID 到 **5_cloud/main/cloud_cfg/deviceid.txt**

- 复制端点文件到 **5_cloud/main/cloud_cfg/endpoint.txt**

3. 构建、烧录、上载固件至设备

现在，设备已经连接到 AWS IoT 云平台，并会在状态更改时通知云端。固件也将从云端获取所有状态更新，并应用到本地。

6.3.3 远程控制

AWS IoT 为连接到它的所有设备提供了 Web API，用以实现远程控制。手机应用程序可以与此 Web API 交互以控制和监测设备。在这里我们使用命令行工具 cURL 来模拟手机应用程序。

使用 cURL，然后在 Linux/Windows/Mac 控制台执行以下命令，就可以读取设备的当前状态：

```
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
```

在上面的命令中，请复制粘贴 deviceid.txt 的文件内容替换 *things* 和 *shadow* 之间的 *<contents-of-deviceid.txt-file>*。

注意：AWS 仅允许获得授权的实体访问设备状态。因此，在上面的命令中，我们用到了 *device.cert* 和 *device.key*，与我们在固件中配置的文件相同，可以确保我们有权访问设备状态。但在生产中，必须要在云端为客户端（如该 cURL 和手机应用程序）创建单独的身份验证密钥，以访问/修改设备状态。

设备状态可以修改为：

```
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
```

此 cURL 命令将生成 HTTP POST 请求，并在 POST 主体中发送 JSON 数据（如上所示）。此 JSON 数据指导 AWS IoT 将设备状态更新为 *false*。

每当您将设备状态从 cURL 端更改为 true 或 false 时，您都可以观察设备上相应的状态更改。

这就是远程控制的实现方式。现在让我们来快速探讨一下代码。

6.3.4 代码

所有云通信的代码都已整合到 *cloud_aws.c* 文件中。此文件的结构与 AWS IoT SDK 所要求的结构标准相似。

该文件使用我们的驱动程序 API：*app_driver_get_state()* 和 *app_driver_toggle_state()*，分别用于获取设备状态和反转设备状态。

AWS IoT 需要在您的固件中嵌入以下 3 个文件：

- AWS CA 证书文件：**5_cloud/main/cloud_cfg/server.cert**
- 设备私钥文件：**5_cloud/main/cloud_cfg/device.key**
- 设备证书文件：**5_cloud/main/cloud_cfg/device.cert**

应用程序使用在固件中嵌入文件 章节中所描述的机制将以上文件嵌入到固件中。

6.4 未完待续

通过这个应用程序，我们终于将插座自身的功能（插座电源的切换）与网络连接功能结合到了一起。设备连接到云端后，我们现在可以通过网络对其进行控制和监控。我们还提到了在连接到任何远程/云服务之前必须考虑的安全性问题。

下一章，我们会探讨连网设备的一个常见需求：空中（OTA）固件升级。

CHAPTER 7

固件升级

[English]

在讨论固件升级之前，让我们先了解一下 flash 分区。

7.1 Flash 分区

ESP-IDF 框架将 flash 划分为多个逻辑分区，用于储存各个组件。具体结构如下：

从上图可以看出，flash 地址在 0x9000 之前的结构是固定的，第一部分包括二级 Bootloader，后面紧接着就是分区表，分区表则用来储存 flash 剩余区域的分布信息。通常，至少包含 1 个 NVS 分区和 1 个固件分区。

7.2 空中升级 (OTA)

固件升级使用活动-非活动分区方案，并预留两个 flash 分区（如下图所示），OTA Data 分区将记录哪个是活动分区。

OTA 固件升级过程中，状态变更如图所示：

- 步骤 0：OTA 0 为活动固件，该信息储存在 OTA Data 分区（如图所示）。
- 步骤 1：固件升级开始，识别并擦除非活动分区，新的固件将写入 OTA 1 分区。
- 步骤 2：固件写入完毕，开始进行验证。
- 步骤 3：固件升级成功，OTA Data 分区已更新，并指示 OTA 1 现在是活动分区。下次启动时，固件将从此分区启动。

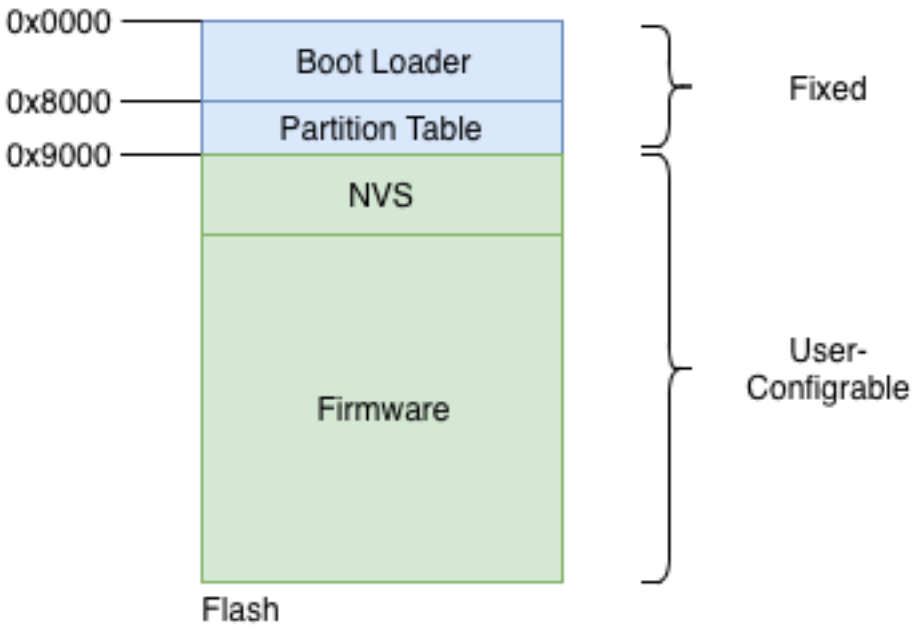

图 1: Flash 分区结构

7.2.1 更新 Flash 分区

那么，我们如何让 IDF 创建一个分区表，既包含 OTA Data 分区又包含 2 个储存固件的分区?

我们可以创建一个分区文件来实现，即 CSV 文件（Comma Separated Values，逗号分隔值），此文件会指示 IDF 我们想要的分区是什么，大小应该是多少，以及如何放置等问题。

本示例所用的分区文件如下图所示：

```
# 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,
```

上述分区文件指导 IDF 创建 NVS、OTA Data、OTA 0 及 OTA 1 分区，同时指定分区大小。

创建此分区文件后，我们应指示 IDF 使用该自定义分区，而非默认分区，可以通过更新 SDK 配置来启用自定义分区。本应用程序示例中，此项设置已经在 *6_ota/sdkconfig.defaults* 文件中激活，因此无需再进行其他激活操作。

但如果希望使用不同的分区文件，或更新主固件的偏移量，请修改此设置。可以通过执行 *idf.py menuconfig* 命令来实现，然后在 *menuconfig -> Partition Table* 中配置正确的选项。

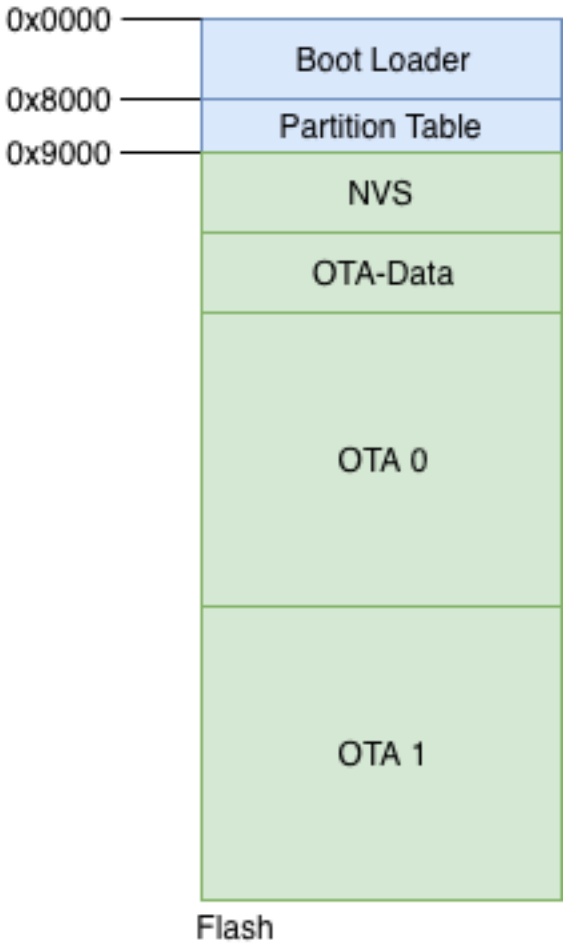

图 2: OTA Flash 分区

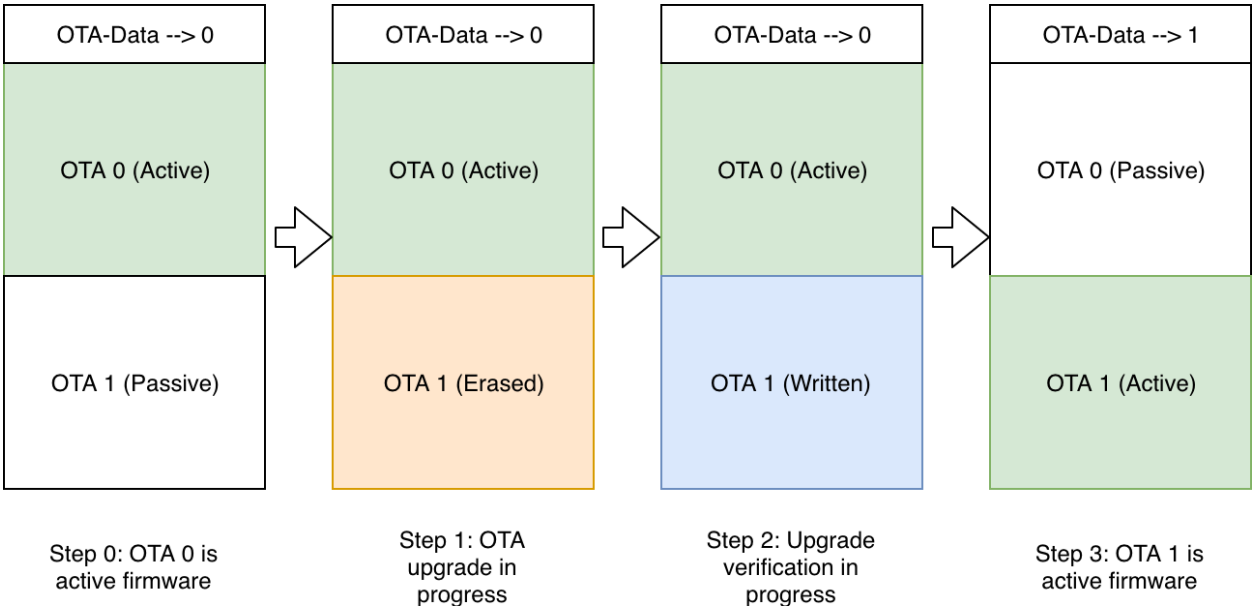

图 3: 固件升级步骤

7.2.2 ESP8266 用户

对使用 ESP8266 的用户而言，如果开发板仅有 2 MB flash，则请使用下面的分区表：

```
# 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 代码

现在我们来看一下实际执行固件升级的代码：

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

- 使用 *esp_http_client_config_t* 定义 OTA 升级源，包括标记升级地址的 URL，用于验证服务器的 CA 证书（升级从此服务器处获取）。注意，请确保按照安全性第一 章节进行 CA 证书验证，这一步非常重要。
- 然后执行 *esp_https_ota()* API 启动固件升级，固件升级成功（或失败）后，此 API 将返回相应的代码（或错误代码）。
- 默认情况下，我们为固件升级 URL 添加了 GitHub 的 CA 证书，这样您就可以轻松地在 GitHub 上存放升级固件并进行升级。理想情况下，您将安装相应服务器的 CA 证书，并从该服务器下载升级固件。

7.4 发送固件升级 URL

现在我们的问题是设备如何接收升级 URL。固件升级指令与前面讨论的远程控制指令不同，固件升级通常由设备生产商根据特定的标准为一批或一组设备进行的升级。

为了简便起见，我们使用相同的远程控制基础架构将固件升级 URL 指令传递给设备。请注意，在批量生产时，您将使用其他的云控制机制发送固件升级 URL。

为了快速进行固件升级，我们在 GitHub 上传了一个固件示例（1_hello_world 应用程序），可以按照下述方式快速升级该固件映像：

```
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
```

如果您使用 ESP32C3 系列开发板，请把 hello-world.bin 更改为 hello-world-c3m-idf5.bin。

固件升级成功后，设备将执行 Hello World 固件。

7.5 未完待续

有了这个固件，我们就实现了智能连网设备的一大关键功能，即固件升级功能。

到现在为止，产品固件马上准备就绪，最后就是维护设备特有数据，我们将在下一章中讨论这一问题。

CHAPTER 8

量产

[English]

构建物联网产品时，我们通常需要在每个设备中储存一些唯一性信息。

例如，我们在前面步骤中提到了云平台使用证书进行身份验证，我们还把需要用到的设备证书嵌入了固件中，这些步骤都用到了这种唯一性信息（证书）。开发单个设备时，可以使用上述方法，将这种唯一性信息直接嵌入到固件中，但如果需要开发大量设备呢？在本章节中，我们将讨论这一问题。

在*NVS: 永久储存键值对* 章节中，我们讨论了 NVS 分区，用于将键值对永久储存在 flash 中。这样即使设备重启，这部分信息也不会丢失。我们还在*代码* 章节中提到了擦除 NVS 分区内容即可将设备恢复出厂设置。

量产时，我们也可以使用类似的 NVS 分区储存每个设备唯一的键值对，但不希望设备恢复出厂设置时擦除这部分信息。要达到这一目的，我们可以创建另一个 NVS 分区，用于储存工厂嵌入的唯一性信息。由于此分区是在工厂预嵌入的，因此我们将此 NVS 分区用作只读分区，仅用来读取设备配置的唯一性信息。

因此，我们就使用这种方法来存储工厂的唯一性信息。

8.1 多个 NVS 分区

在讨论固件升级时，我们在*Flash 分区* 章节中研究了 *flash* 分区，还在*更新 Flash 分区* 章节中研究了如何修改 flash 分区。在本示例中，我们将新添加一个名为 *fctry* 的 NVS 分区，用于存储唯一的工厂配置信息。

您可以在 *7_mfg/partitions.csv* 文件中查看相关信息。

8.2 代码

有了这个 *fctry* NVS 分区，我们就可以使用标准 NVS API 进行访问。唯一的问题是，我们在执行 NVS 操作时，需要指示 NVS 使用该分区。这可以通过初始化 NVS 句柄来完成，如下所示：

```
#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);
```

现在，您可以使用 *fctry_handle* NVS 句柄执行 NVS 操作，从该 *fctry* NVS 分区读取数据。例如:

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

现在，我们就可以从烧录至设备的 *fctry* NVS 分区中读取证书信息，而不再需要使用代码将证书嵌入到固件中。

8.3 生成工厂数据

现在我们可以从固件方面讨论这个问题了。但在此之前，我们仍然需要指定工厂数据生成的机制，这些工厂数据将写入 *fctry* 分区。

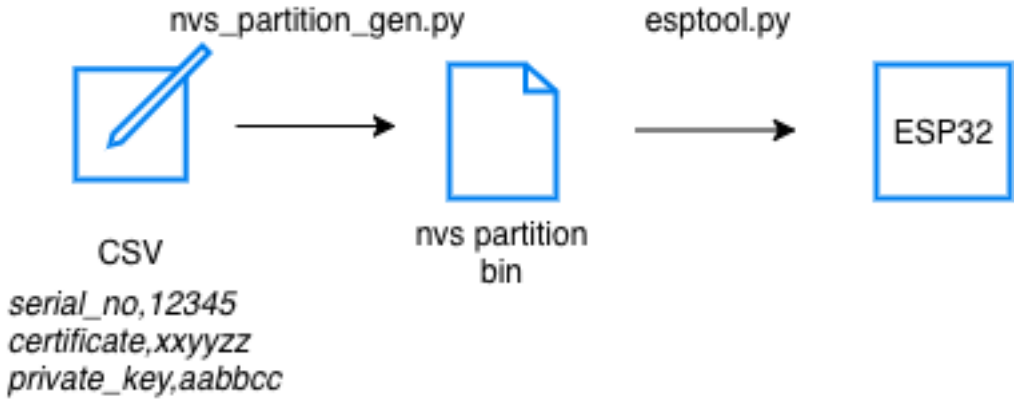

图 1: 生成工厂分区

使用 *components/nvs_flash/nvs_partition_generator/nvs_partition_gen.py* 程序在开发主机上生成 NVS 镜像，然后我们需要将镜像写入 flash 中的 *fctry* 分区。

此程序接收一个 CSV 文件，并依据 CSV 文件生成一个 NVS 分区镜像。CSV 文件储存键值对信息，这部分

信息将被加入到生成的 NVS 分区。工厂将生成海量的此类 NVS 分区镜像，每生产一台设备，就将一个唯一的分区镜像写入设备。

应用程序中还提供了一个名为 *mfg_config.csv* 的 CSV 示例文件。文件中每一行都包含有在工厂设置的唯一变量值，更新变量值后，您唯一的设置信息就被加入到 CSV 文件。

接下来，您可以使用下面的指令生成 NVS 分区 bin 文件：

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

my_mfg.bin 文件就是 NVS 分区数据，现在可以嵌入到设备中。您可以使用以下命令将此 NVS 分区 bin 文件写入 flash：

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

8.3.1 ESP8266 用户

对使用 ESP8266 的用户而言，如果开发板仅有 2 MB flash，请使用下面的指令烧录 my_mfg.bin：

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

现在，如果启动固件，固件将会像上一章中的固件一样进行工作。但在这种情况下，固件映像的功能与设备内置的唯一配置信息无关。但会根据这些唯一的配置信息表象为不同设备。

这样，您就可以根据需要创建任意数量的唯一性镜像，然后将这些镜像烧录到相应的开发板上。

请参考 工厂分区文档，查看更多详细信息。

8.4 未完待续

在本章中，我们研究了如何为每个设备创建具备唯一性的工厂镜像，设备不同，镜像内容也不同。

现在，我们就有了一个功能齐全，可以量产的设备固件！

CHAPTER 9

安全性

[English]

在上一章节中，我们实现了能够进行量产的设备固件，但还不能完美收官，因为我们还没有考虑到设备的安全性问题。在本章节中，我们将介绍有哪些安全性问题需要考虑。

9.1 远程通信安全

设备与外部实体通信时，必须确保通信安全，建议使用已有的 TLS 协议标准来保护通信安全，而不推荐使用其他新方法。ESP-IDF 支持 *mbedtls*，*mbedtls* 实现了 TLS 协议的全部功能。

ESP-Jumpstart 项目中所有代码均采用了上述安全机制，确保远程通信安全。您也可以采用这种机制来保护设备固件其他类型的远程通信，如果您尚未用到远程通信，请跳过本章节。

9.1.1 CA 证书

TLS 层使用受信任的 CA 证书验证远程端点/服务器是否正确。

esp_tls API 则接收 CA 证书用于服务器验证。

```
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,
};
```

(下页继续)

(续上页)

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

如果没有此参数，则会跳过服务器验证。强烈建议对所有 TLS 通信，指定受信任的 CA 证书用于服务器验证。

9.1.2 获取 CA 证书

从上面的代码可以看出，必须将受信任的 CA 证书嵌入固件，才能用于验证服务器。您可以使用以下命令获取受信任的 CA 证书：

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

运行此命令将输出证书列表，请将列表中最后一个证书嵌入到设备固件中，具体步骤请查看在固件中嵌入文件 章节。

9.2 物理访问安全

ESP32 可以保护设备免受物理篡改，保障设备物理访问安全。本章后续内容将详细介绍这一功能。

9.2.1 ESP8266 用户

ESP8266 暂不支持此功能，不能防止对设备进行物理篡改。

9.2.2 Secure Boot

Secure Boot 可以确保 ESP32 从 flash 运行任何软件时，软件受信任且由已知实体签名。如果软件 Bootloader 和应用程序固件中有任何改动，此固件则被视为不受信任，设备将拒绝执行此不受信任的代码。

这一功能是通过构建信任链来实现的，包括从硬件到 Bootloader，再到固件。

具体工作步骤如下：

- 量产时：
  - 将密钥嵌入到 ESP32 eFUSE，密钥嵌入后禁止软件读出或写入；
  - Bootloader 和固件使用正确的密钥进行签名，并将签名添加到映像中；
  - 将签名版的 Bootloader 和固件映像嵌入 ESP32 flash。
- ESP32 上电复位时：
  - BootROM 使用 eFUSE 中嵌入的安全密钥，验证 Bootloader；

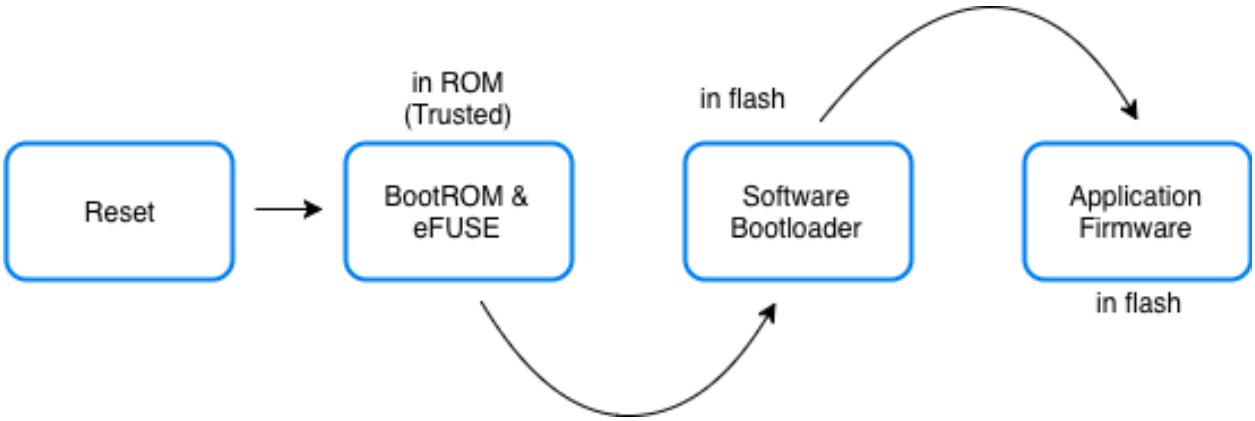

图 1: Secure Boot

- Bootloader 验证通过后，BootROM 开始加载并执行 Bootloader；
- Bootloader 开始验证固件签名；
- 固件签名验证通过后，Bootloader 加载并执行固件。

如上所述，启动设备 Secure Boot 之前，我们需要先进行其他一些操作。请参考 Secure Boot 相关文档，查看更多详细信息。

9.2.3 Flash 加密

flash 加密可以确保储存在 ESP32 flash 中的应用程序固件保持加密状态，从而允许制造商在设备中传输加密固件。

启用 flash 加密时，所有经内存映射对 flash 进行的读操作，均在运行时进行透明解密。flash 控制器使用储存在 eFUSE 中的 AES 密钥来执行 AES 解密。这个存储在 eFUSE 中的加密密钥与上面的 Secure Boot 密钥是分开存放的，该密钥还可以防止软件读出和写入。因此，只有硬件有权对 flash 内容解密。

请参考 flash 加密文档，查看如何启用 flash 加密功能。

9.2.4 NVS 加密

与应用程序固件相比，NVS 分区具有不同的访问模式，写操作更频繁，且内容依赖用户偏好。适用于应用程序固件的加密技术，并不是 NVS 加密的最佳选择。因此，ESP-IDF 为 NVS 分区提供了专门的加密机制，即行业标准 AES-XTS 加密，推荐使用这种加密方式保护静态数据。

具体工作步骤如下：

- 量产时：
  - 创建一个单独的 flash 分区，专门储存用于 NVS 加密的密钥；
  - 将此分区标记为 flash 加密；

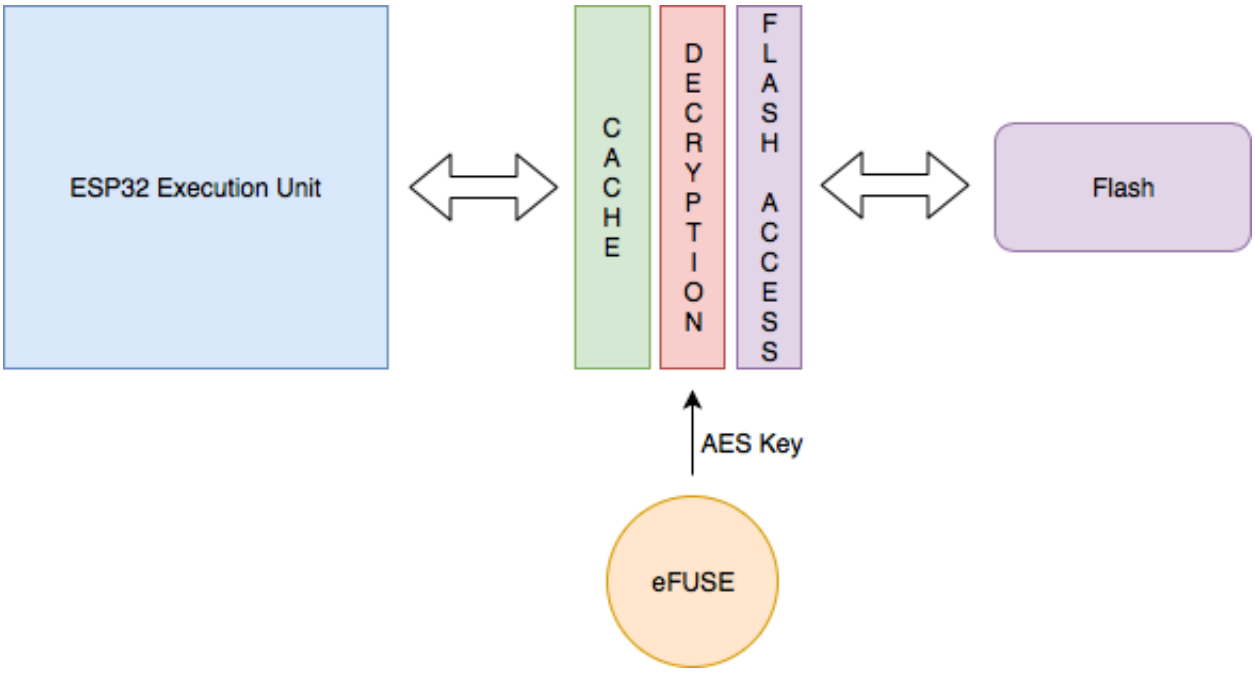

图 2: flash 加密

  - 使用 *nvs_partition_gen.py* 工具生成随机密钥分区文件；
  - 将生成的分区文件写入新建的分区。

- 在固件中：
  - 调用 *nvs_flash_read_security_cfg()* API 从上述分区读取加密密钥，并将密钥填充到 *nvs_sec_cfg_t* 中；
  - 使用 *nvs_flash_secure_init()* API 或 *nvs_flash_secure_init_partition()* API 初始化 NVS flash 分区；
  - 正常执行其他的 NVS 操作。

请参考 NVS 加密相关文档，查看更多详细信息。