第一章 ESP32 的空中升级（OTA）

1. 学习目的及目标

- 掌握 OTA 工作过程
- 掌握 ESP32 的 OTA 程序设计

2. OTA 工作过程讲解

在实际产品开发过程中，在线升级（OTA）可以远程解决产品软件开发引入的问题，更好地满足用户需求。

2.1. ESP32 的 OTA 简介（原文）

OTA（空中）更新是使用 WiFi 连接而不是串行端口将固件加载到 ESP 模块的过程。

2.2. ESP32 的 OTA 升级有三种方式：

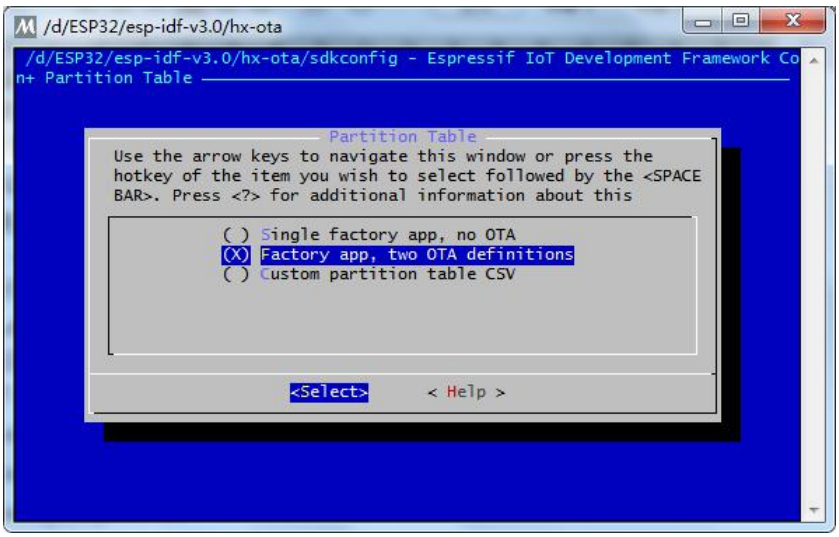
- Arduino IDE：主要用于软件开发阶段，实现不接线固件烧写
- Web Browser：通过 Web 浏览器手动提供应用程序更新模块
- HTTP Server：自动使用 http 服务器 - 针对产品应用

在三种升级情况下，必须通过串行端口完成第一个固件上传。

OTA 进程没有强加的安全性，需要确保开发人员只能从合法/受信任的来源获得更新。更新完成后，模块将重新启动，并执行新的代码。开发人员应确保在模块上运行的应用程序以安全的方式关闭并重新启动。

2.3. ESP32 Flash 空间分区配置

目前使用的 ESP-WROOM-32 集成 4MB SPI Flash。在编译 esp32 程序时，通过 make menuconfig -> PartitionTable 可以有三种分区选择：工厂程序（无 OTA 分区）/工厂程序（双 OTA 分区）/用户自定义分区。如下图：

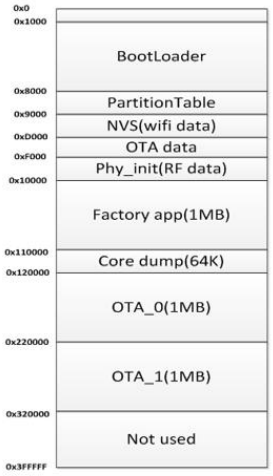


menuconfig 中的配置只是修改配置文件中的宏，实际上 ESP32 SDK 对应 Flash 分区配置的源码路径是：\esp-idf-v3.0\components\partition_table 该路径下有以下.csv 文件都是用来对 Flash 分区进行配置的。

无 OTA 分区：partitions_singleapp.csv、partitions_singleapp_coredump.csv

双 OTA 分区：partitions_two_ota.csv、partitions_two_ota_coredump.csv

双 OTA 分区时，4M SPI Flash 的分区情况：



SPI Flash分区示意
（two_ota_coredump）

2.4. OTA 升级策略(HTTP)

ESP32 连接 HTTP 服务器（可以使本地也可以是云，OTA demo 使用本地服务器），发送请求 Get 升级固件；每次读取 1KB 固件数据，写入 Flash。

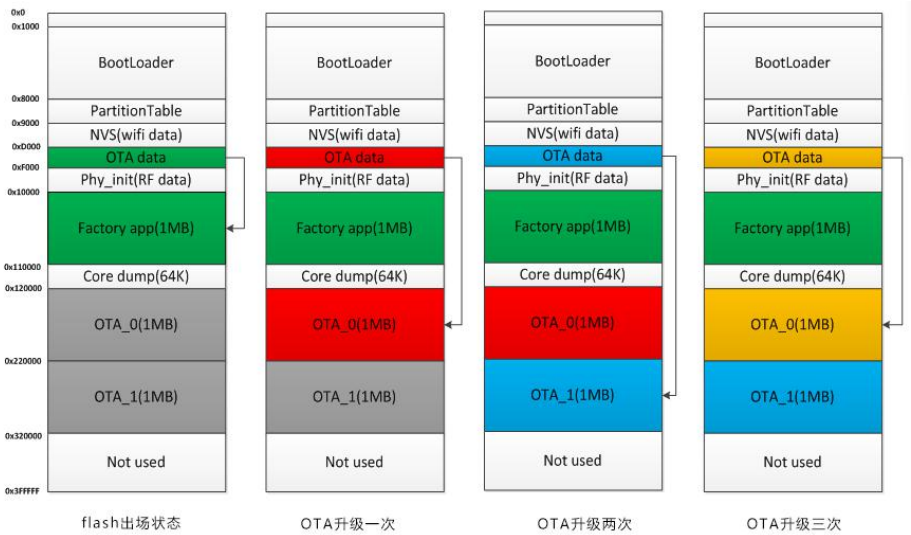
ESP32 SPI Flash 内有与升级相关的（至少）四个分区：

- OTA data 区：决定运行哪个区的 App
- Factory App 区：有出厂时的默认 App
- OTA_0 区：OTA_0 App
- OTA_1 区：OTA_1 App

首次进行 OTA 升级时，OTA Demo 向 OTA_0 分区烧录目标 App，并在烧录完成后，更新 OTA data 分区数据并重启。

系统重启时获取 OTA data 分区数据进行计算，决定此后加载 OTA_0 分区的 App 执行（而不是默认的 Factory App 分区内的 App），从而实现升级。

同理，若某次升级后 ESP32 已经在执行 OTA_0 内的 App，此时再升级时，OTA Demo 就会向 OTA_1 分区写入目标 App。再次启动后，执行 OTA_1 分区实现升级。以此类推，升级的目标 App 始终在 OTA_0、OTA_1 两个分区之间交互烧录，不会影响到出厂时的 Factory App 固件，如下图状态。



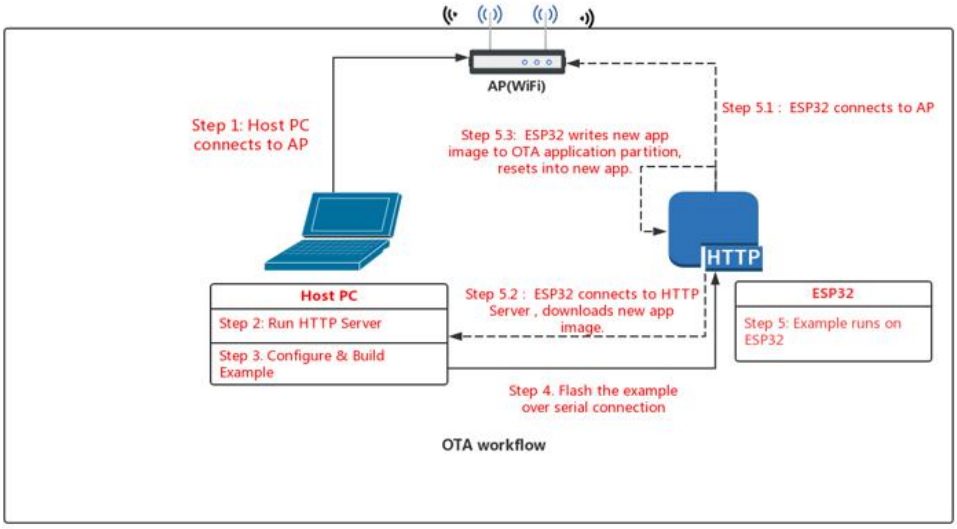
2.5. 保密性 Security

模块必须以无线方式联网获取新的固件， 这使得模块被强行入侵并加载了其他代码。 为了减少被黑客入侵的可能性，请考虑使用密码保护您的上传，选择某些 OTA 端口，也可以给 bin 文件加密等。

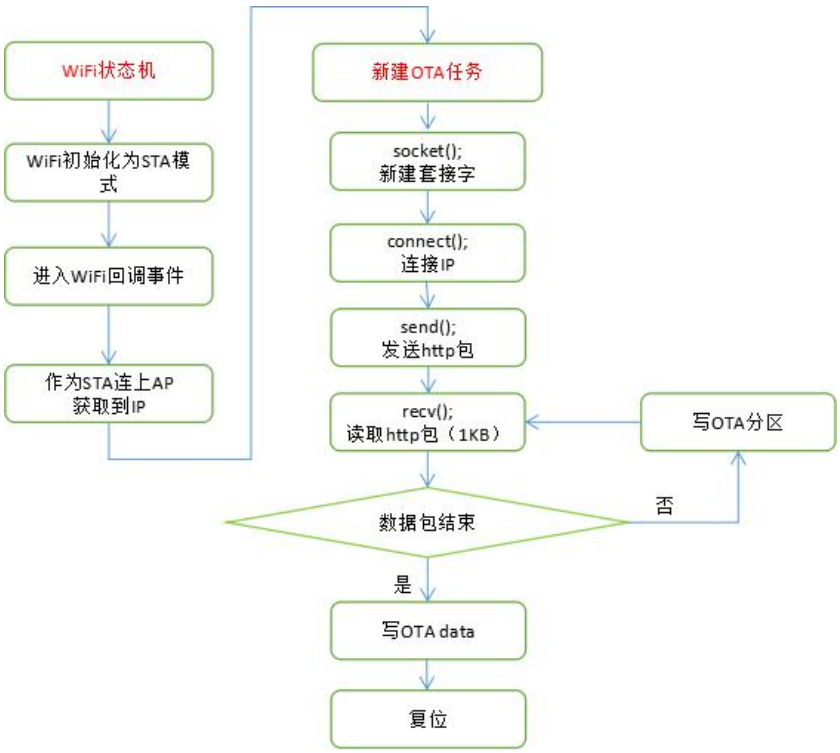
3. ESP32 OTA Demo 升级流程和说明

3.1. 升级流程

- 电脑连上路由器（AP）
- 电脑运行 HTTP 服务器（本地）
- 下载 OTA Demo 到 ESP32 开发板
- ESP32 连上路由器（AP）后就会访问 HTTP 下载新的 APP 到 OTA 区



3.2. ESP32 的 OTA 详细过程逻辑



3.3. ESP32 OTA 接口简略说明

- ota 源码路径：\esp-idf-v3.0\examples\system\ota\main\ota_example_main.c

1	esp_ota_get_boot_partition	boot
2	esp_ota_get_running_partition	获取当前系统执行的固件所在的 Flash 分区
3	esp_ota_get_next_update_partition	获取当前系统下一个（紧邻当前使用的 OTA_X 分区）可用于烧录升级固件的 Flash 分区
4	esp_ota_begin esp_ota_write esp_ota_end	向可用的 Flash 分区（一般是 OTA_X 分区）刷入升级目标固件
5	esp_ota_set_boot_partition	升级完成更新 OTA data 区数据，重启时根据 OTA data 区数据到 Flash 分区加载执行目标（新）固件

- boot 源码路径：

\esp-idf-v3.0\components\bootloader\subproject\main\bootloader_start.c

1	load_partition_table	加载 Flash 分区表（从分区表找到 OTA data 区地址）
2	get_selected_boot_partition	获取 Flash 启动分区（计算 OTA data 区数据得到）
3	load_boot_imageunpack_load_app:	从 Flash 启动分区加载解压固件并执行

- OTA 分区操作流程

4. 软件设计

4.1. ESP32 的 HTTP 接口介绍，同 TCP 接口

4.2. OTA 任务编写

和 HTTP 获取城市温度基本相同，这里是将 HTTP 应答包中的数据存放到 OTA 分区。

```
static void ota_example_task(void *pvParameter)
{
    esp_err_t err;
    /* update handle : set by esp_ota_begin(), must be freed via esp_ota_end() */
    esp_ota_handle_t update_handle = 0 ;
    const esp_partition_t *update_partition = NULL;

    ESP_LOGI(TAG, "Starting OTA example...");
    //获取 OTA app 存放的位置
    const esp_partition_t *configured = esp_ota_get_boot_partition();
    //获取当前系统执行的固件所在的 Flash 分区
    const esp_partition_t *running = esp_ota_get_running_partition();

    if (configured != running) {
        ESP_LOGW(TAG, "Configured OTA boot partition at offset 0x%08x, but running from offset 0x%08x",
                 configured->address, running->address);
        ESP_LOGW(TAG, "(This can happen if either the OTA boot data or preferred boot image become
corrupted somehow.)");
    }
    ESP_LOGI(TAG, "Running partition type %d subtype %d (offset 0x%08x)",
             running->type, running->subtype, running->address);

    //等待 wifi 连上后进行 OTA，项目中可以使升级命令进入 OTA
    xEventGroupWaitBits(wifi_event_group, CONNECTED_BIT,
                        false, true, portMAX_DELAY);
    ESP_LOGI(TAG, "Connect to Wifi ! Start to Connect to Server....");

    //连 http 服务器
    if (connect_to_http_server()) {
        ESP_LOGI(TAG, "Connected to http server");
    } else {
        ESP_LOGE(TAG, "Connect to http server failed!");
        task_fatal_error();
    }

    //组 http 包发送
    const char *GET_FORMAT =
        "GET %s HTTP/1.0\r\n"
        "Host: %s:%s\r\n"
```

```
        "User-Agent: esp-idf/1.0 esp32\r\n\r\n";

    char *http_request = NULL;
    int get_len = asprintf(&http_request, GET_FORMAT, EXAMPLE_FILENAME, EXAMPLE_SERVER_IP,
EXAMPLE_SERVER_PORT);
    if (get_len < 0) {
        ESP_LOGE(TAG, "Failed to allocate memory for GET request buffer");
        task_fatal_error();
    }
    int res = send(socket_id, http_request, get_len, 0);
    free(http_request);
    if (res < 0) {
        ESP_LOGE(TAG, "Send GET request to server failed");
        task_fatal_error();
    } else {
        ESP_LOGI(TAG, "Send GET request to server succeeded");
    }

    //获取当前系统下一个（紧邻当前使用的 OTA_X 分区）可用于烧录升级固件的 Flash 分区
    update_partition = esp_ota_get_next_update_partition(NULL);
    ESP_LOGI(TAG, "Writing to partition subtype %d at offset 0x%x",
            update_partition->subtype, update_partition->address);
    assert(update_partition != NULL);
    //OTA 写开始
    err = esp_ota_begin(update_partition, OTA_SIZE_UNKNOWN, &update_handle);
    if (err != ESP_OK) {
        ESP_LOGE(TAG, "esp_ota_begin failed, error=%d", err);
        task_fatal_error();
    }
    ESP_LOGI(TAG, "esp_ota_begin succeeded");

    bool resp_body_start = false, flag = true;
    //接收完成
    while (flag) {
        memset(text, 0, TEXT_BUFFSIZE);
        memset(ota_write_data, 0, BUFFSIZE);
        //接收 http 包
        int buff_len = recv(socket_id, text, TEXT_BUFFSIZE, 0);
        if (buff_len < 0) { //包异常
            ESP_LOGE(TAG, "Error: receive data error! errno=%d", errno);
            task_fatal_error();
        } else if (buff_len > 0 && !resp_body_start) { //包头
            memcpy(ota_write_data, text, buff_len);
            resp_body_start = read_past_http_header(text, buff_len, update_handle);
        } else if (buff_len > 0 && resp_body_start) { //数据段包
            memcpy(ota_write_data, text, buff_len);
```

```
            //写 flash
            err = esp_ota_write( update_handle, (const void *)ota_write_data, buff_len);
            if (err != ESP_OK) {
                ESP_LOGE(TAG, "Error: esp_ota_write failed! err=0x%x", err);
                task_fatal_error();
            }
            binary_file_length += buff_len;
            ESP_LOGI(TAG, "Have written image length %d", binary_file_length);
        } else if (buff_len == 0) {  //结束包
            flag = false;
            ESP_LOGI(TAG, "Connection closed, all packets received");
            close(socket_id);
        } else {//未知错误
            ESP_LOGE(TAG, "Unexpected recv result");
        }
    }

    ESP_LOGI(TAG, "Total Write binary data length : %d", binary_file_length);
    //OTA 写结束
    if (esp_ota_end(update_handle) != ESP_OK) {
        ESP_LOGE(TAG, "esp_ota_end failed!");
        task_fatal_error();
    }
    //升级完成更新 OTA data 区数据，重启时根据 OTA data 区数据到 Flash 分区加载执行目标（新）固件
    err = esp_ota_set_boot_partition(update_partition);
    if (err != ESP_OK) {
        ESP_LOGE(TAG, "esp_ota_set_boot_partition failed! err=0x%x", err);
        task_fatal_error();
    }
    ESP_LOGI(TAG, "Prepare to restart system!");
    esp_restart();
    return ;
}
```

5. 效果展示

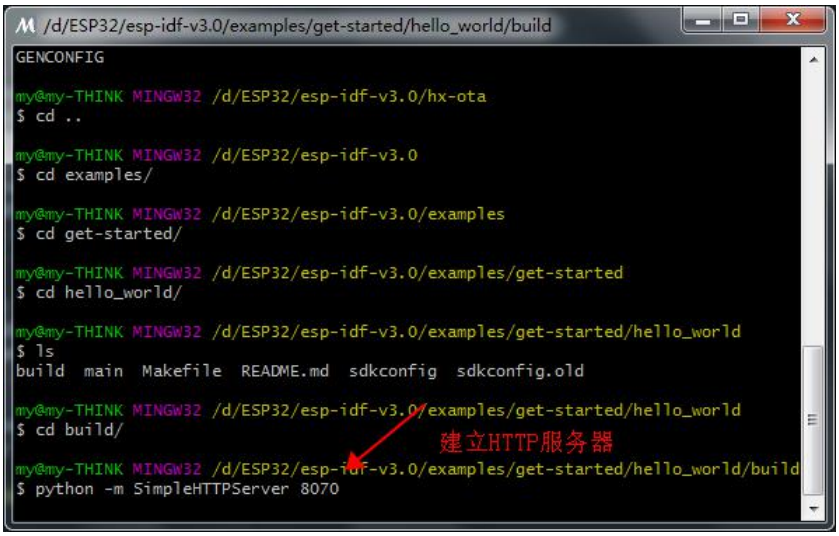
5.1. 测试流程（原文）

➢ 启动服务器

编译链内 Python 有一个内置的 HTTP 服务器，我们这里可以直接使用它。我们将会使用示例 get-started/hello_world 作为需要更新的固件。

打开一个终端，输入如下的命令来编译示例并启动服务器：

```
cd $IDF_PATH/examples/get-started/hello_world    //进入 helloworld 路径
make                                             //编译
cd build                                         //进入编译文件.bin 目录
python -m SimpleHTTPServer 8070                  //运行 http 服务器（本地）
```



服务器运行后，构建目录的内容可以通过网址 http://localhost:8070/ 浏览到。

NB：在某些系统中，命令可能是 python2 -m SimpleHTTPServer。

NB：你可能已经注意到，用于更新的“hello world”没有任何特殊之处，这是因为由 esp-idf 编译的任何 .bin 应用程序都可以作为 OTA 的应用程序。唯一的区别是它是会被写到工厂分区还是 OTA 分区。

如果你的防火墙阻止了对端口 8070 的访问，请在本示例运行期间打开它。

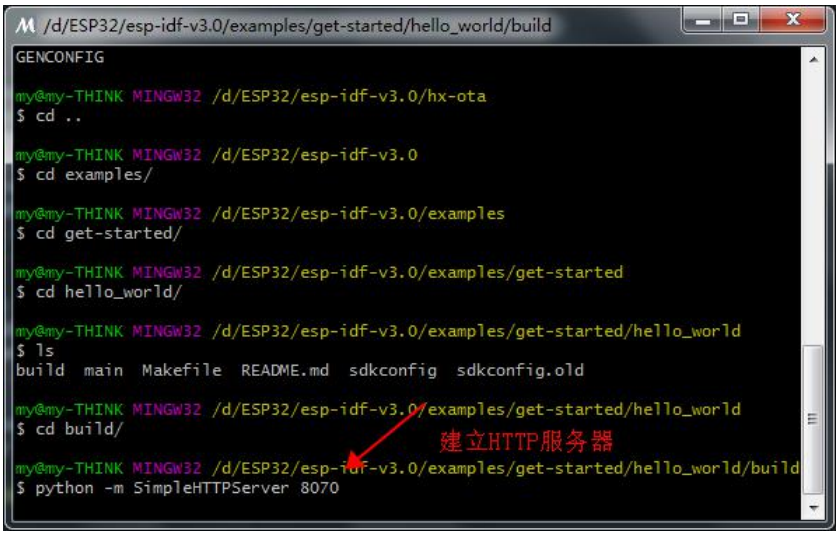
If you have any firewall software running that will block incoming access to port 8070, configure it to allow access while running the example.

- 在烧写时，需要先用目标 erase_flash 来擦除整个 flash（这会删除之前在 ota_data 分区留下的所有数据），然后通过串口烧写工厂进行：

```
1 make erase_flash flash    //擦除整个 flash
```

- 修改 WiFi 账号密码，HTTP 服务器的 IP（电脑 IP）和 Port，编译、下载 OTA Demo 代码到 ESP32
- ESP32 连上 WiFi 后自动 OTA，成功后运行 HelloWorld 程序

5.2. 效果展示



```
my@my-THINK MINGW32 /d/ESP32/esp-idf-v3.0/hx-ota
$ make erase_flash
Erasing entire flash...
esptool.py v2.1
Connecting....
Chip is ESP32D0WDQ6 (revision 1)
Uploading stub...
Running stub...
Stub running...
Erasing flash (this may take a while)...
Chip erase completed successfully in 3.5s
Hard resetting...
```

全擦除

```
my@my-THINK MINGW32 /d/ESP32/esp-idf-v3.0/hx-ota
$ make flash monitor
Flashing binaries to serial port com7 (app at offset 0x10000)...
esptool.py v2.1
Connecting....
Chip is ESP32D0WDQ6 (revision 1)
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Auto-detected Flash size: 4MB
Compressed 19392 bytes to 11435...
Wrote 19392 bytes (11435 compressed) at 0x00001000 in 1.0 seconds (effective 152.1 kbit/s)...
Hash of data verified.
Compressed 493216 bytes to 306686...
Wrote 493216 bytes (306686 compressed) at 0x00010000 in 27.1 seconds (effective 145.8 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 117...
Wrote 3072 bytes (117 compressed) at 0x00008000 in 0.0 seconds (effective 1024.0 kbit/s)...
Hash of data verified.
```

下载并监测串口

```
I (2491) wifi: state: auth -> assoc (0)
I (2491) wifi: state: assoc -> run (10)
I (2511) wifi: connected with stop, channel 1
I (3111) event: sta ip: 192.168.2.105, mask: 255.255.255.0, gw: 192.168.2.1
I (3111) ota: Connect to Wifi ! Start to Connect to Server....
I (3111) ota: Server IP: 192.168.2.103 Server Port:8070
I (3131) ota: Connected to server
I (3131) ota: Connected to http server
I (3131) ota: Send GET request to server succeeded
I (3131) ota: Writing to partition subtype 16 at offset 0x110000
I (3211) ota: esp_ota_begin succeeded
I (3211) ota: esp_ota_write header OK
I (3211) ota: Have written image length 1249
I (3221) ota: Have written image length 2273
I (3221) ota: Have written image length 3297
I (3221) ota: Have written image length 4321
I (3231) ota: Have written image length 5345
I (3231) ota: Have written image length 6369
```

连上wifi

发http请求

获取http数据写到ota分区

```
I (3651) ota: Have written image length 144225
I (3651) ota: Have written image length 144272
I (3651) ota: Connection closed, all packets received
I (3651) ota: Total Write binary data length : 144272
I (3661) esp_image: segment 0: paddr=0x00110020 vaddr=0x3f400020 size=0x04e80 ( 20096) map
I (3681) esp_image: segment 1: paddr=0x00114ea8 vaddr=0x3ffb0000 size=0x02170 (  8560)
I (3691) esp_image: segment 2: paddr=0x00117020 vaddr=0x40080000 size=0x00400 (  1024)
0x40080000: _iram_start at D:/ESP32/esp-idf-v3.0/components/freertos/xtensa_vectors.S:1685

I (3691) esp_image: segment 3: paddr=0x00117428 vaddr=0x40080400 size=0x08598 ( 34200)
I (3721) esp_image: segment 4: paddr=0x0011f9c8 vaddr=0x400c0000 size=0x00000 (     0)
I (3721) esp_image: segment 5: paddr=0x0011f9d0 vaddr=0x00000000 size=0x00640 (  1600)
I (3731) esp_image: segment 6: paddr=0x00120018 vaddr=0x400d0018 size=0x1334c ( 78668) map
0x400d0018: _stext at ??:?

I (3801) esp_image: segment 0: paddr=0x00110020 vaddr=0x3f400020 size=0x04e80 ( 20096) map
I (3821) esp_image: segment 1: paddr=0x00114ea8 vaddr=0x3ffb0000 size=0x02170 (  8560)
I (3821) esp_image: segment 2: paddr=0x00117020 vaddr=0x40080000 size=0x00400 (  1024)
0x40080000: _iram_start at D:/ESP32/esp-idf-v3.0/components/freertos/xtensa_vectors.S:1685

I (3831) esp_image: segment 3: paddr=0x00117428 vaddr=0x40080400 size=0x08598 ( 34200)
I (3861) esp_image: segment 4: paddr=0x0011f9c8 vaddr=0x400c0000 size=0x00000 (     0)
I (3861) esp_image: segment 5: paddr=0x0011f9d0 vaddr=0x00000000 size=0x00640 (  1600)
I (3871) esp_image: segment 6: paddr=0x00120018 vaddr=0x400d0018 size=0x1334c ( 78668) map
0x400d0018: _stext at ??:?

I (3941) ota: Prepare to restart system!
I (3951) wifi: state: run -> init (0)
```

接收完成

重启

```
I (0) cpu_start: Starting scheduler on APP CPU.
Hello world!
This is ESP32 chip with 2 CPU cores, WiFi/BT/BLE, silicon revision 1, 4MB external flash
Restarting in 10 seconds...
Restarting in 9 seconds...
Restarting in 8 seconds...
Restarting in 7 seconds...
```

运行新固件

6. OTA 总结

- 此处 OTA 方式的优点与可能的问题

从 ESP32 SDKOTA Demo 升级策略看，应该是比较稳妥的，无论升级期间出现任何异常，只要 OTA data 区数据未被修改，设备还可以加载原有的固件执行。

目前看来可能需要考虑的地方有：

1. 是否存在可能，OTA data 数据指向了一个升级失败的区，导致设备加载损坏的固件；
2. 因为 OTA 需要三个升级相关区，因此固件大小被限制在小于 SPI Flash Size/3
3. 获取升级目标固件还应当加入防错/重传/校验的机制；出现异常时也应当有相应处理。
4. 留出后台控制接口，用于修改 OTA data 区，便于远程控制程序运行。

- OTA 完全使用了官方的源码，项目中建议使用 HTTPS。
- 源码地址：https://github.com/xiaolongba/wireless-tech