Internet of Things

Laboratory 3

Over The Air Programming (OTAP)
(2024-2025)

”Gheorghe Asachi” Technical University of Iaşi
Faculty of Automatic Control and Computer Engineering
Computer Engineering Department

Contents

List of Tables

List of Figures

1. Objectives

- Familiarize yourself with the concept of updating firmware using a generic communication interface;
- Develop an application that employs a Wi-Fi connection to update the code.

2. OTAP (Over the Air Programming)

2.1 Overview

As its name suggests, an OTAP (Over-The-Air Programming) method updates the firmware via a wireless communication interface. The main advantage of this approach is that a system can be updated after it has already been deployed (e.g., installed in a location that is difficult to access). The primary drawback, however, is the additional complexity it adds to the application's operating logic.

An OTAP process typically involves the following steps:

- Retrieving the new binary file through the communication channel (Wi-Fi, Bluetooth, or other proprietary methods);
- Verifying the binary file's integrity (optional);
- Writing the new code to the Flash memory;
- Modifying the boot sequence so that the new code is used when the system restarts.

2.2 *esp-idf* implementation

esp-idf provides two components for implementing an OTAP (Over-The-Air Programming) solution:

- A flash memory partitioning mechanism;
- A library for writing to flash memory (and managing partitions).

An ESP32 SoC's attached flash memory can contain up to 95 partitions, which can be used for storing code or data. The partition table is written at offset 0x8000 and occupies 3072 bytes. At the framework level, this partition table is described by CSV files with a specific structure.

The default partition table used by the applications in the previous labs has the following structure (and is defined in the file `partitions_singleapp.csv` located in `$ESP-IDF-DIR$/components/partition_table`):

```
# Espressif ESP32 Partition Table
# Name, Type, SubType, Offset, Size, Flags
nvs,      data, nvs,     0x9000,  0x6000,
phy_init, data, phy,     0xf000,  0x1000,
factory,  app,  factory, 0x10000, 1M,
```

Listing 1. `partitions_singleapp.csv` contents

The *nvs* partition is used by more complex libraries (e.g., Wi-Fi, Bluetooth, lwIP) to store initialization and state information. The *phy_init* partition stores calibration data for the radio peripheral, while the *factory* partition contains the main application that is uploaded via the bootloader.

When using the OTA mechanism, there will be two or more partitions of type *app* (subtype *ota_x*, where x ranges from 0 to 15), as well as a *data* partition (subtype *ota*) that holds the boot information.

```
# Espressif ESP32 Partition Table
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x9000,  0x4000,
otadata,  data, ota,     0xd000,  0x2000,
phy_init, data, phy,     0xf000,  0x1000,
factory,  0,    0,       0x10000, 1M,
ota_0,    0,    ota_0,  0x110000, 1M,
ota_1,    0,    ota_1,  0x210000, 1M,
```

Listing 2. Partition table with two ota partitions

Info

- If the *otadata* partition does not contain any data (initially filled with 0xFF), the bootloader will load the default application from the *factory* partition;
- If you want to revert to the default application (in the *factory* partition) after performing an OTAP update, you need to erase the contents of the *otadata* partition.

If the *otadata* partition indicates that at least one application exists in an *ota_x* partition, the bootloader will load the newest application (where “newest” is determined by a counter variable).

As shown in Figure 1, the *ota_x* partitions are written alternately, ensuring the previous version is always retained in flash memory. This way, if the latest version contains bugs, you can revert to the earlier code and still have access to the OTAP mechanism.

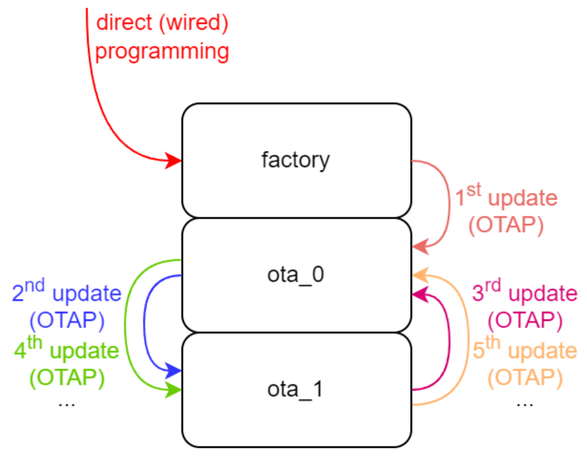

Fig. 1. Usage sequence for the application partitions

2.3 Interaction flow

Figure 2 illustrates a set of OTAP interactions. First, the initial application with OTAP support is loaded onto the platform (step 1), and then update images are made available on a web server (step 2). When an update event occurs (e.g., a button press or a periodic check) (step 3), the ESP32 platform runs an HTTP client and retrieves the new image from the server (step 4). After writing the image to the next available *ota_x* partition, the platform resets and runs the newly loaded code (step 5).

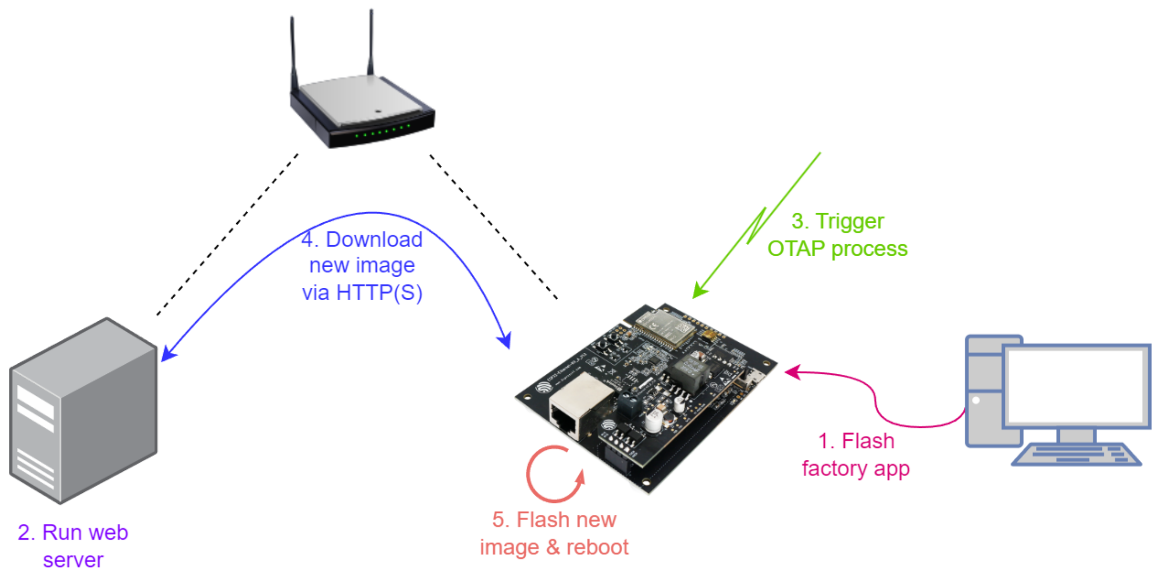
4. Download new image via HTTP(S)
3. Trigger OTAP process
1. Flash factory app
2. Run web server
5. Flash new image & reboot

Fig. 2. Interaction steps for an OTAP via HTTP application

3. Tasks

1. Follow the steps outlined below to run a basic OTAP scenario:

    - Create a new project and add the files available on the Moodle platform:
        - add `main.c` in `src` subfolder;
        - add `ca_cert.pem`, `ca_key.pem` and `server.py` to the root folder.
    - Add extra configuration directives in *platformio.ini* as instructed in Lab 1;
    - In order to use the partition table presented in Listing 2, add the line `board_build.partitions = partitions_two_ota.csv` to *platformio.ini*.
    - The ESP32 HTTP client needs the self-signed Certificate used by the server during the TLS handshake. In order to load it in the flash, add the directive `board_build.embed_txtfiles = ca_cert.pem` to *platformio.ini*. Also, at the end of *CMakeLists.txt* from the `src` subfolder, append `target_add_binary_data(${COMPONENT_TARGET} "../ca_cert.pem" TEXT)`
    - Upload the code and start the serial monitor;
    - Run the *server.py* script on the PC. It implements a HTTPS server that exposes the binary of the project as a web resource;
    - Press the button on the add-on board of the ESP32 platform and monitor the serial output. The application executes steps 4 and 5, as described above.

2. Modify the application from task 1 to implement a conditional update mechanism based on versioning.

    - Add the *versioning.py* script to the project root folder. In *platformio.ini* add the directive `extra_scripts = pre:versioning.py`

      Each time the **Build** command is executed, before the compilation process starts, the *versioning.py* script is run. This script keeps track of the total number of builds and generates a *version.h* header file. That file contains three `#define` directives, which can be used to identify the current version.
    - Modify the *server.py* script. The server must provide a resource named */version* (or similar), that will return the current version of the build (hint: you must read one of the files *version.h* or *versioning*)
    - Modify the code that runs on ESP32. Prior to step 4, verify if the version available on the server is newer than the one on the board. Use the *esp_http_client* library. API docs are available here, code examples are available here.

Support documents

- ESP32 datasheet and manual
- esp-idf API reference
- FreeRTOS API reference