Yet Another Example of ESPx Over the Air (OTA) Firmware Upload and Related Details

Fernando G. Tinetti*

Technical Report TR-RT-02-2022
III-LIDI, Fac. de Informática, UNLP
CIC, Prov. de Buenos Aires
Argentina
contact e-mail: fernando@info.unlp.edu.ar
September 2022

Abstract. This document is intended to define a step-by-step guide for successful programming of the ESPx (ESP8266/ESP32) devices "Over The Air" or OTA, as referred to in most of the available documentation. OTA provides a simple yet highly useful functionality: update the ESP runtime code at runtime without the requirement of having a physical-serial direct communication with the device, i.e., via WiFi. While there are many official and unofficial documents about the ESP OTA, this document/report is expected to provide one simplest example of ESP OTA, including some details on ESP functionality seen in the process of experimentation with available code and examples. Security issues will not be discussed in this report, mostly because they are highly dependent on the specific applications and runtime environment in production systems.

1.- Introduction

Development environments for computing/control systems based on microcontrollers (or, for short, MCU: MicroController Units) are mostly based on personal computers cross-compilers and related tools. Furthermore, microcontrollers-based computing and control systems are heavily dependent on software, mostly due to the increasing MCU computing power and hardware reliability. In these environments, IDE (Integrated Development Environments) including simulators, cross-compilers and loading firmware software are the most usual tools, even in low-cost Arduino development boards, and boards handled from the Arduino IDE. The standard workflow/s in these environments is schematically shown in Fig. 1, where

- The *initial* (a)-cycle can be considered "software only":
  - o Write → (Cross)Compile → Simulate → (again to Write) …

  is made in a standard computer/computing (source code) development environment. Simulation sometimes is made previous to any real hardware test on the MCU i.e. a purely software simulation of the MCU.
- The (b)-cycle includes specific hardware
  - o Write → (Cross)Compile → Load → Test → (again to Write) …

  is usually driven from the same computer/computing (source code) development environment used in the (a)-cycle, and implies firmware uploading on the MCU, as well as using the complete application MCU hardware-attached devices (e. g. sensor/s and actuator/s).

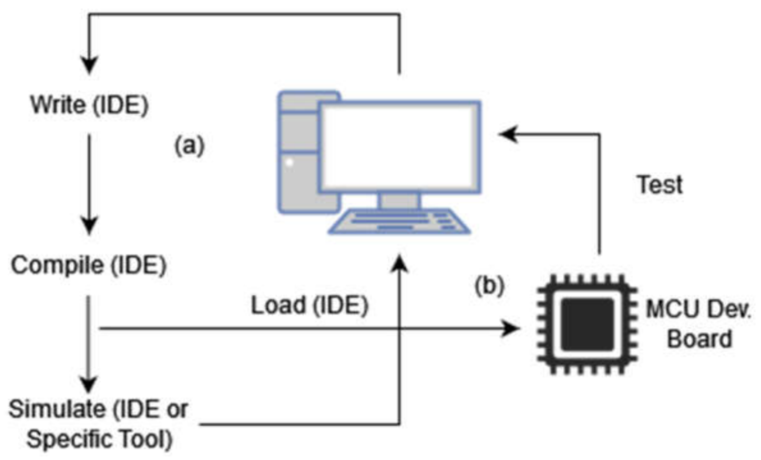


Figure 1: Development Workflow/s for MCU-Based Environments.

Traditionally, setting the firmware/loading the MCU program has been a manual task, mostly via a direct cable interconnection between a (standard) computer and a MCU development board. Since many years, a USB serial interconnection is the most likely method for firmware uploading on MCU-based development boards, including the popular Arduino development boards, and boards handled from the Arduino IDE. Espressif SoCs (System on Chips) [4] ESP8266 and ESP32 (ESPx, for short in this technical report) have been integrated in several development boards capable of being programmed from the Arduino IDE. The whole set of capabilities of ESPx are not going to be discussed in this report, but the firmware loading system/options. More specifically, there are a whole set of libraries, cross compiler and firmware loader tools for programming ESPx from a standard computer with a USB port and cable. The next section will discuss the ESPx firmware upload options.

2.- ESPx Uploading Firmware Options

As Fig. 2 schematically shows, the initial and standard way of firmware uploading in ESPx involves a very simple hardware and software configuration:

a) A computer running an IDE such as the Arduino IDE [2], PlatformIO [8] or ESP-IDF provided by Espressif for the ESP32 SoCs [5].
b) A USB cable for connecting the computer with an ESPx. (take into account that the USB cable should have the proper data communication lines as well as power lines).
c) A development board based on some model of ESPx (e.g., ESP8266-01, ESP32, ESP32-S, etc.), many of them popularly known in the Arduino environment and users.

In these environment/configuration, ESPx firmware uploading is almost transparent to the application developer, because it is “reduced to” using the corresponding command IDE interface. Furthermore, the ESPx code does not include any detail about how firmware is loaded, it is just assumed to be ran from the corresponding ESPx flash memory.

Small ESPx systems are easily developed and handled with an environment as that shown in Fig. 2, provided there is physical access to the hardware. Thus, having the first final application can be easily managed “in house” by the corresponding developers. However, IoT (Internet of Things) systems and MCU (and ESPx in particular) applications in general are highly heterogeneous not only in terms of functionality but in terms of environments in which they operate once they are deployed (or reach the “in production state”). As it is very well known in the *traditional* application software/software engineering areas, new versions of software are likely to appear sometimes because of bug fixing and most of the times for adding/changing system functionality. In this scenario, changing the software (firmware) of systems based on ESPx involve some drawbacks usually not found in massively used computing systems. Furthermore, updating software systems running in data centers (public/private clouds and on-premise data centers) or those running in mobile devices (cellphones, tablets, etc.) have their own set of facilities and tools: DevOps [7] [3] [6] and/or tailored tools and methodologies are successfully used in both environments since several years.

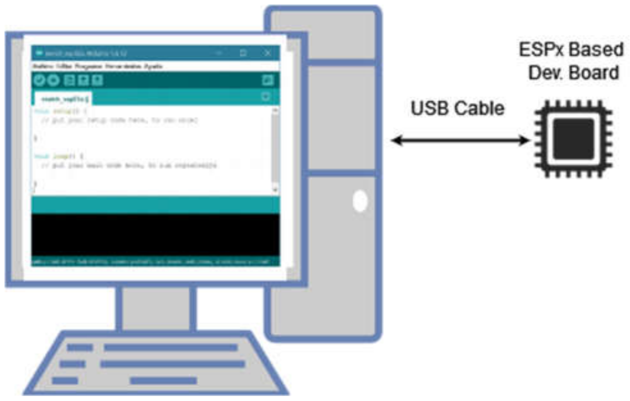


Figure 2: ESPx Upload via USB.

ESPx based systems have unique and different characteristics as compared to aforementioned software systems. Beyond the strongly different hardware capabilities, ESPx based systems have at least two problems for software update, which would imply firmware upload to each ESPx:

1. Many ESPx are tightly integrated in other systems and/or in hard to physically reach, i.e. making too difficult and therefore increasing the cost of physical access:
    a. Having an ESPx controlling a robot/UV (Unmanned Vehicle) would imply disassembly the robot/UV for plugging in the USB cable.
    b. Several distributed sensor and control systems are installed in distant and hard to reach areas.
2. ESPx systems are installed in physically distributed areas and/or in mobile devices (e.g., for tracking purposes), so the individual/ one-by-one upgrading is time-consuming and prone to errors.

For those cases mentioned above, as well as for simplifying firmware upgrade/update, ESPx systems have the so-called OTA (Over The Air) update facility (ESPx OTA will be used for short from now on in this report). Fig. 3 schematically shows the ESPx OTA, which (initially) takes advantage of the WiFi ESPx capabilities for receiving the firmware from the development source/environment. Clearly, the ESPx firmware binary is generated in the standard PC development environment (as that explained before), but the way in which the firmware binary is received and started to run in the ESPx is strongly different. In particular, OTA implies losing firmware upload transparency for providing a very flexible way of changing the code to be run in the ESPx. More specifically, this report will be focused on the ways of receiving the firmware in the ESPx via WiFi, but (at least conceptually), the firmware is just a binary file received in the ESPx to be "installed" and "started to run" once it is available. Furthermore, there are several options for the process identified in Fig. 3 as "ESPx Binary/Firmware(*)" to reach the ESPx WiFi range. This technical report will be focused in two specific ESPx OTA configurations, both involving ESPx WiFi communications, as shown in Fig. 4:

a) The development computer is connected to a WiFi Access Point/router which, in turn, transfers the binary data to the ESPx. In this case, the ESPx is configured as a "Station" (STA) in ESPx terminology, i.e., the ESPx is a device connected to the WiFi network provided by a WiFi router.

b) The development computers is connected to a WiFi network provided by the ESPx itself. In this case, the ESPx is configured as an “AP” (Access Point) in ESPx terminology.

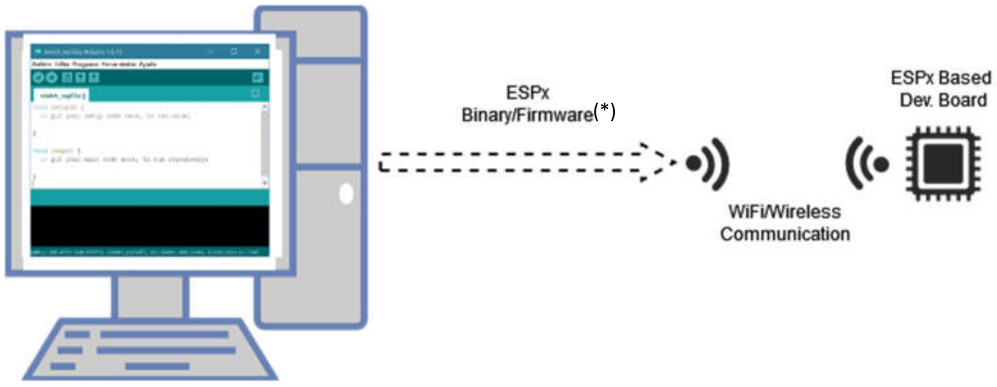


Figure 3: ESPx OTA.

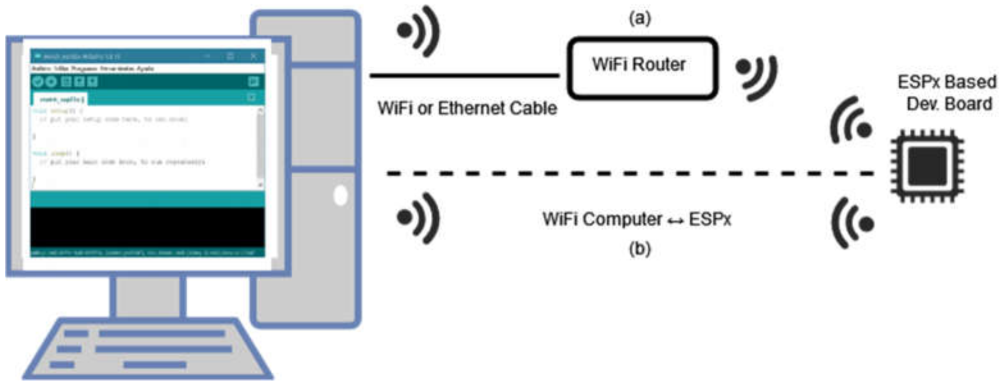


Figure 4: ESPx OTA Basic WiFi Options.

Both firmware uploading options depicted in Fig. 4 could be considered almost “Internet-free” (the router can be maintained isolated from Internet access) and, thus, almost secure in terms of Internet privacy and security, i.e., unwanted accesses and security violations. It is highly arguable the pros and cons of both configurations as compared to Internet access to the ESPx, but the objective of this report is focused in OTA, not privacy, security, and/or any other Internet implications. There are lots of web sites and long discussions about different approaches, and many of them seem to be relevant in the corresponding specific context/s. Furthermore, if ESPx OTA is expected to be made via Internet, option (a) could be used, adding that the router provides such Internet access.

Related Details: Options other than using WiFi for ESPx OTA firmware upload include some ESPx wireless access/communication without an Internet router or even WiFi at all. At least conceptually, the only requirement for ESPx OTA firmware upload is that the corresponding firmware in binary form (i.e., a binary file) is available at the ESPx. Clearly, WiFi is the most discussed and documented way, including several libraries available, some of them directly included in the Arduino IDE environment, which will be used in this report for the sake of simplicity and general availability. As a *rule of thumb*: “if something is available in the Arduino IDE then it is available in every other development environment”.

3.- Learning from OTA Examples

There are several OTA examples for ESPx development boards, some of them are already included in the Arduino IDE (once ESPx development boards are installed), and many examples available in web sites. Every example and the OTA usage in itself implied that the application source code has to be modified for including OTA functionality.

Related Details: OTA firmware upload implies that the application itself if modified for including OTA, i.e., firmware upload is no longer "transparent" to the developer. Conceptually, in terms of the application itself, a new functionality is included: OTA. On one hand, it is part of the software functions, so it is reasonable it has its own part of the application (source) code. On the other hand, now the application is not only "solving" the application-specific functionality but "administrative" or operating system-like functionality: firmware (binary software) upload. The latter is radically different from classical software engineering and software development, where operational tasks are rather isolated from application software development (statement that now maybe argued against in the context of current DevOps trends).

Arduino IDE OTA examples will be discussed in this section, mostly because they are almost ubiquitously available in every IDE/library used for ESPx OTA. Besides, Arduino OTA examples have specific characteristics it is interesting to describe in this section. Arduino OTA examples described in this report correspond to Arduino IDE version 1.8.13 and ESP32 1.0.6 (it is expected to work in almost every ESPx version, though) already installed. There are two Arduino OTA examples, explicitly identified in the Arduino IDE (available under Examples ==> ESP32 Dev Module ==> Arduino OTA):

- BasicOTA.ino
- OTAWebUpdater.ino

There are some common underlying design and implementation decisions in both examples:

- Given they are focused on ESPx, they do not explain how to use them at runtime. Almost every example in the web shows the usage from Arduino IDE, which makes the whole system too similar to have a computer physically connected to the ESPx with a USB cable.
- They include host/service resolution (via the Espressif ESPmDNS library) and/or some security and/or data integrity control (user-password, MD5), which is reasonable for a final implementation but, at the same time
  - Arguable about final utility (long discussions on this issue in the web, not tackled in this report).
  - Implies a code mixture: OTA concepts + Security + Data integrity control + …, a mixture avoided in this report.
- There is some reference to low(er)-level details which will be avoided in this report. More specifically, this report focuses on a "plain" .bin file containing a new ESPx firmware version to run in the ESPx, avoiding to make reference to details such as "sketch/filesystem" or "flash/spiffs" binaries.
- At least the most documented examples are shown with interactive/manual updates, which make them interesting to learn, but hard to adapt to more automated firmware uploads (e.g., via a command line to be executed under a software development process/trigger event).

Related Details: Arduino OTA examples as well as many OTA examples found in the Internet avoid using the Arduino delay() function and include in the sketch loop function a call to an OTA handling function. In the case of the two Arduino OTA examples (found in the Arduino IDE), Fig. 5 shows the sketch loop function, where: a) "ArduinoOTA" has been configured/instantiated (in the sketch setup function) to have every necessary instantiation for handling the OTA according to the ArduinoOTA library definitions, and b) "server" is basically an HTTP server with one more *endpoint* added, "/update" for receiving and starting a new firmware, i.e., for handling the ESPx OTA. Many examples in the Internet include a “delay” function or “delay” code which does not actually suspends the microcontroller (as the standard delay function does), which in fact is a useful design and implementation decision for almost every Arduino and ESPx application. Other libraries are directly asynchronous, i.e., they do require to call an OTA handling function in the sketch loop function [1].

```
void loop() {
  ArduinoOTA.handle();
}
```

a) BasicOTA.ino

```
void loop(void) {
  server.handleClient();
  delay(1);
}
```

b) OTAWebUpdater.ino

Figure 5: Arduino OTA Examples loop() Function.

In this report, another Arduino IDE example/sketch will be used, WebUpdater.ino, available in the Arduino IDE under Examples ==> ESP32 Dev Module ==> HTTPUpdateServer ==> WebUpdater. It seems rather strange it is not explicitly identified as OTA or related to OTA but it actually is. The WebUpdater example is almost the same code as that of the OTAWebUpdater, but contains several interesting and useful characteristics:

- There is a single comment line at the beginning of the source code that provides some insight about how to use OTA without any IDE, with a plain command line:

  “To upload through terminal you can use: curl -F "image=@firmware.bin" esp32-webupdate.local/update”

  Unfortunately, the specific curl command line given in the example/comment was found not to work, but the command line idea (i.e., independently of any IDE and IDE interface) will be used in the approach in this document.
- The complete HTML code provided by the HTTP server is “hidden” (or, maybe, contained…) in a separate file, HTTPUpdateServer.h and, thus, the OTA *server* (functionality) is directly used from the source code of the application without any reference to HTML/HTTP.

Fig. 6 shows the source code of the WebUpdater.ino example as it is provided by the Arduino IDE, that includes several source code lines that will not be used in the rest of this report: those related to ESPmDNS usage/configuration library. As a starting point, the ESPx configuration as STA will be maintained i.e., the ESPx will be a WiFi station, connected to a WiFi access point (a priori, isolated from the public Internet). Later, it is possible to configure the ESPx so that an external WiFi AP will not be necessary. Thus, the initial source code for OTA experimentation and usage will be that of Fig. 7, which could be described as one of the simplest source codes to start with, named WebUpdater_r1.ino:

- Every reference to ESPmDNS is deleted.
- ESPx connection as STA is simplified.

```
/*
  To upload through terminal you can use: curl -F "image=@firmware.bin" esp32-webupdate.local/update
*/

#include <WiFi.h>
#include <WiFiClient.h>
#include <WebServer.h>
#include <ESPmDNS.h>
#include <HTTPUpdateServer.h>

#ifndef  STASSID
#define STASSID "your-ssid"
#define STAPSK  "your-password"
#endif

const char* host = "esp32-webupdate";
const char* ssid = STASSID;
const char* password = STAPSK;

WebServer httpServer(80);
HTTPUpdateServer httpUpdater;

void setup(void) {
  Serial.begin(115200);
  Serial.println();
  Serial.println("Booting Sketch...");
  WiFi.mode(WIFI_AP_STA);
  WiFi.begin(ssid, password);

  while (WiFi.waitForConnectResult() != WL_CONNECTED) {
    WiFi.begin(ssid, password);
    Serial.println("WiFi failed, retrying.");
  }

  MDNS.begin(host);
  if (MDNS.begin("esp32")) {
    Serial.println("mDNS responder started");
  }

  httpUpdater.setup(&httpServer);
  httpServer.begin();

  MDNS.addService("http", "tcp", 80);
  Serial.printf("HTTPUpdateServer ready! Open http://%s.local/update in your browser\n", host);
}

void loop(void) {
  httpServer.handleClient();
}
```

Figure 6: WebUpdater.ino (Arduino IDE) Example.

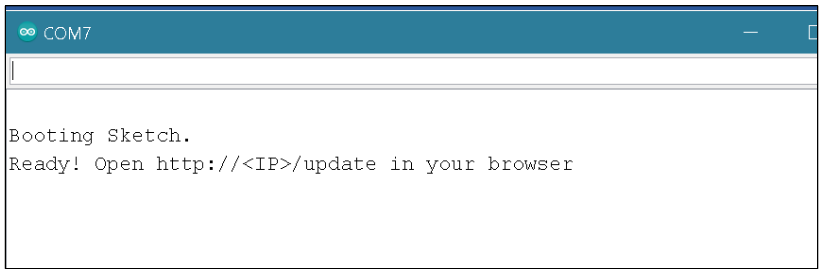
It could be argued that WebUpdater_r1.ino (in Fig. 7) is too complicated to be used, because it is not necessarily simple to know the IP given by the WiFi AP to which the ESPx is connected to. Besides, the IP assignment to the ESPx configured as STA is usually dynamic, and setting a fixed IP is possible but rather specific/manual to do in the WiFi AP as well as to maintain in the long term. Nevertheless, this report is going to start with this source code version and following WebUpdater0x versions will be enhanced not only for access but for including other interesting ways of firmware uploading. The next section, though, will show the WebUpdater_r1 "web interface" and usage from a web browser.

```
#include <WiFi.h>
#include <WiFiClient.h>
#include <WebServer.h>
#include <HTTPUpdateServer.h>

// WiFi to connect to, as station WiFi client/STA
#define STASSID "<SSID_here>"
#define STAPASS "<password_here>"

WebServer httpServer(80);
HTTPUpdateServer httpUpdater;

/****************************************************************/
void setup(void) {
  Serial.begin(115200);
  Serial.println();
  Serial.print("Booting Sketch");
  WiFi.mode(WIFI_AP_STA);

  do {
    WiFi.begin(STASSID, STAPASS);
    Serial.print(".");
  } while (WiFi.waitForConnectResult() != WL_CONNECTED);

  httpUpdater.setup(&httpServer);
  httpServer.begin();

  Serial.printf("\nReady! Open http://<IP>/update in your browser\n");}

/****************************************************************/
void loop(void) {
  httpServer.handleClient();
}
```

Figure 7: WebUpdater_r1.ino, First WebUpdater Modification.

4.- First Example Usage

Fig. 8 shows the Arduino IDE Serial Monitor once the sketch WebUpdater_r1 is loaded and run in the ESP (in this case, the ESP is connected to computer's COM7 USB.



```
Booting Sketch.
Ready! Open http://<IP>/update in your browser
```

Figure 8: WebUpdater_r1.ino, Arduino IDE Serial Monitor.

The IP assigned to the ESP is defined by the WiFi Access Point (AP), because the ESP is configured as a WiFi station. In this example, the IP assigned by the AP is 192.168.1.64, so a browser could access a web page provided by the ESP. Fig. 9 shows the web page sent by the ESP, where:

- The html is *hidden* from the programmer, it is neither in the WebUpdater_r1.ino nor in the WebUpdater.ino, it is defined by the WebUpdater library.
- There seems to be two options for upload: “Update Firmware” and “Update FileSystem”. It is not very clear at this point similarities and differences among both options. However, at least the “Update Firmware” option imply to look for a file in the browser local filesystem to send the binary file with the new firmware to be loaded and run in the ESP.

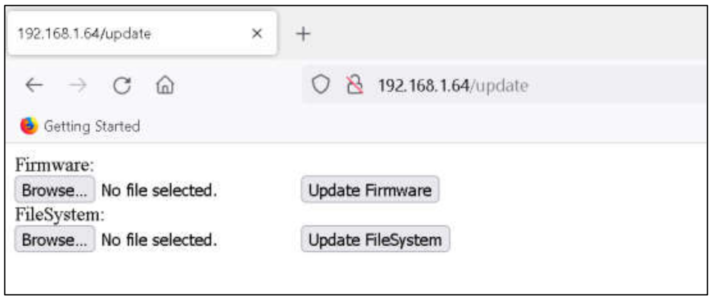


Figure 9: Web page Provided by WebUpdater_r1.ino.

Obtaining a .bin file for firmware uploading is easily obtained in general and in the Arduino IDE in particular (note that the ESP development environment usually implies to generate a binary file to be uploaded in the ESP). Fig. 10 shows the sequence of using the web page, in which the binary file containing a firmware to be uploaded has been already selected (Simple.ino.ESP32.bin) and previous to select/click the “Update Firmware” web page button.

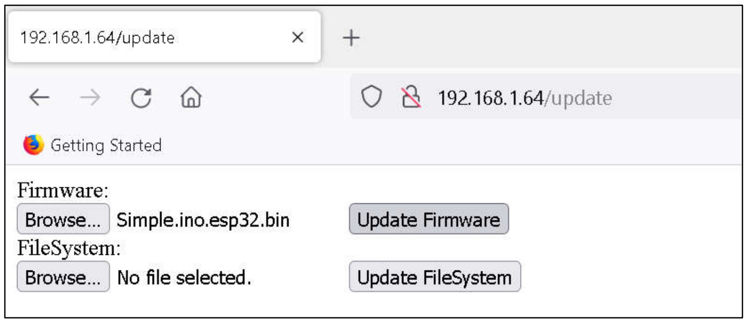
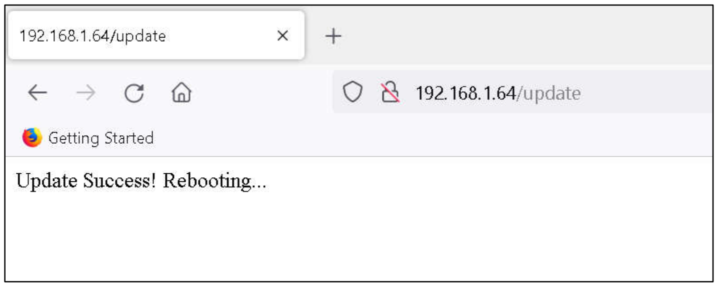


Figure 10: Selecting and Using a Binary File for Firmware Uploading.

The last step is the specific “firmware upload” via the web page, which implies sending the binary file from the web browser and receiving the binary file at the ESP and using that file as a new firmware to be run in the ESP. Fig. 11 shows the browser result once the “Update Firmware” is used in the web page. After that, the ESP is running the newly uploaded firmware (the file sent by the browser). If OTA needs to be maintained in the new application run in the ESP, then the new firmware should contain some similar way of firmware uploading as that explained up to this point.



Update Success! Rebooting...

Figure 11: Firmware Upload Page after Sending the Binary File.

The process shown in this section is proven to work and is clear in terms of interactive usage. It has disadvantages, too, mostly in terms of automatization beyond those frequently discussed in terms of security exposure. In this context, automatization is related at least with two important technical software update tasks:

1. Distributed system software upgrade: when more than one ESP (or microcontroller, in general) is involved, manual/interactive software (firmware, in this case) update is prone to introduce human errors. Error likelihood is proportional to the number of ESP to be upgraded. In case of errors, the whole (distributed) system may also introduce errors/problems/inconsistencies for the inter-operation of different software/firmware versions.
2. When ESPs are physically distributed, interactive update impose an operator to keep track each ESP individually, identifying and handling details such as DNS/IP details. All of these details are better managed by specific programs/automatic processes, which are also able to keep track and identify specific individual errors such as Internet/network connectivity.

Thus, the next section will be focused on using the WebUpdater example in a programmatic way, thus avoiding the interactive upload via a web browser.

5.- Technical Issues for Programmatic OTA Usage

The interactive usage makes transparent the real exchange of information and, more specifically, the tasks a browser carries out in order to send a binary file for being uploaded in the ESP, allowed by the ESP OTA firmware upload. Thus, it is interesting to identify the HTML the browser receives from the ESP32, shown in Fig. 12. Beyond that of the browser layout of HTML forms, the important technical information is given by the line

```
form method='POST' action='' enctype='multipart/form-data'
```

i.e. the browser generates at runtime an HTTP post request containing the binary file with encoding type defined as 'multipart/form-data'. Thus, an HTTP client has to generate/replicate this browser behavior. The example code includes a comment line making reference to using the command curl:

```
/*
To upload through terminal you can use: curl -F "image=@firmware.bin" esp8266-webupdate.local/update
*/
```

Actually, that line does not work, at least at it is given in the example source code comment line (probably the curl command does not properly format the enctype='multipart/form-data', but it was not examined/debugged in full detail).

```
<body>
    <form method='POST' action='' enctype='multipart/form-data'>
      Firmware:<br>
      <input type='file' accept='.bin,.bin.gz' name='firmware'>
      <input type='submit' value='Update Firmware'>
    </form>
    <form method='POST' action='' enctype='multipart/form-data'>
      FileSystem:<br>
      <input type='file' accept='.bin,.bin.gz,.image' name='filesystem'>
      <input type='submit' value='Update FileSystem'>
    </form>
    </body>
```

Figure 12: WebUpdater HTML.

Summarizing, the requirements in this scenario for uploading firmware OTA to the ESP are

- IP connectivity to the ESP
- The binary file containing the firmware to be uploaded
- HTTP post generation with the firmware, specifically encoded as 'multipart/form-data'

The HTTP post generation can be made in any language, and there are several examples in the Internet (not too easily found, however). Fig. 13 shows a and adapted a python example for this report, including several comments.

```
<#!Python
# pip install requests (if not already installed...)
# Works fine with ESP IP

import requests

upldfiledict = {"file": open(r"<FirmwareFilename.bin>", "rb")}

post_resp = requests.post("http://< ESP_IP>/update", files=upldfiledict)
print(post_resp)
```

Figure 13: Python Code for OTA Usage.

The Python code shown in Fig. 13 uses the Python requests package, which could be expected, and generates the proper HTTP post request via a dictionary which is referenced by the variable upldfiledict, the specific file is read as a binary file, as expected.

Related Details: the “client” program using the OTA upgrade, either a web browser, or the process generated or using the Python code in Fig. 13 (or any other similar process, for that matter) should be an authorized one. More specifically, once the OTA is enabled via an IP, the security issues have to be discussed. Security issues could be “contained” with either a “non-public” IP (e.g., 192.168.x.y) or HTTPS access or similar design and implementation decisions. In all the cases, each decision has it own technical implications and characteristics.

Using OTA via an HTTP POST request could be replaced by some other OTA method/implementation, such as modifying OTA examples available in the Arduino IDE, or more specific libraries available for the OTA ESPx. This report will not analyze any other of such alternative methods mostly because the basic functionality is already obtained with the method explained so far. Also, there is no “best” method, as commented above.

6.- Conclusions and Further Work

OTA provides a simple yet effective way of firmware upload/upgrade, particularly in ESPx, where no extra hardware is needed due to the ESPx WiFi facility. Even when most OTA examples and tutorials are shown via interactive tasks (e.g., using an IDE or a web browser) it has been shown that it is possible to automate firmware uploading via an external process. Uploading automatization includes advantages such as uploading multiple ESPx without manual intervention and other (interactive) activities prone to errors. A simple example such as the WebUpdater (directly available once the ESPx is installed in the Arduino IDE) can be directly used, without any change in order to have ESPx OTA update. A simple Python program has been proven to take advantage of the WebUpdater code running in the ESPx for OTA update. Furthermore, the specific WebUpdater code implementing OTA update implies a few source code lines to be included in every application requiring OTA update.

Most of the further work can be related to security issues in OTA updates, since a new firmware is now enabled to be uploaded and run in a production environment. So far, there is no single solution and/or configuration universally accepted due to the extremely high heterogeneity of applications. Maybe a large number of security problems would be avoided if OTA is made independent from a single access, such as that of the HTTP server receiving and starting the new firmware as the case of the WebUpdater example. Making OTA independent of an HTTP server (or any other communication-related task, for that matter) implies an explicit separation between receiving the firmware binary from starting to run that new firmware, which does not seem to be the case so far.

References

[1] Arduino, “AsyncElegantOTA - Arduino Reference”
https://www.arduino.cc/reference/en/libraries/asyncelegantota/

[2] Arduino, “Software | Arduino”
https://www.arduino.cc/en/software

[3] J. Davis and R. Daniels, Effective DevOps, O'Reilly Media, July 2016.

[4] Espressif Systems (Shanghai) Co., Ltd., “Chipsets | Espressif Systems”,
https://www.espressif.com/en/products/socs/

[5] Espressif Systems (Shanghai) Co., Ltd., “Official IoT Development Framework”
https://www.espressif.com/en/products/sdks/esp-idf

[6] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Professional, July 2010.

[7] G. Kim, J. Humble, P. Debois, J. Willis, N. Forsgren, The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations, IT Revolution Press, 2nd Ed., November 2021.

[8] PlatformIO, “A professional collaborative platform for embedded development”
https://platformio.org/platformio-ide