CustomBoot-32
Project Report

Varun Adinath Patil
Omkar Nanajkar
Archit More

August 2025

Acknowledgments

Overview

PCB design is not merely a skill—it is a form of creative freedom. It empowers an individual to bring abstract concepts to life, unbound by limitations, transforming ideas into tangible reality.

In the same way, bootloaders stand as silent architects of modern computing, crafting the vital run-time environments upon which machines awaken and operate. They are the unseen bridge between hardware and software, the very first whisper that gives a system its voice.

Through this project, we venture into the world of embedded systems—where engineering meets imagination. We dive deep into the art of PCB design and the intricate workings of bootloaders, not just to understand their functions, but to appreciate their importance. In doing so, we seek not only knowledge, but also the inspiration to build systems that are both purposeful and enduring.

TABLE OF CONTENTS

1. Introduction

1.1 About

CustomBoot-32 is a project rooted in the domains of embedded systems, PCB design, and bare-metal programming. The primary objective of this project is to gain an in-depth understanding of concepts such as **bootloaders, OTA (Over-The-Air) updates, UART communication, file transfer protocols, and PCB development**, ultimately leading to the creation of a custom **development board**.

1.2 Problem Statement

STM32F103C8T6 also called as 'Blue Pill', has limited Flash memory and lacks an integrated **USB-to-UART** converter, which necessitates the use of either an external USB-to-UART adapter or an **ST-Link**. Additionally, flashing the device requires outdated tools such as the **STM CubeProgrammer**, making the process inefficient. Furthermore, switching between two firmwares currently requires reflashing each time, which is repetitive and unproductive. To overcome these limitations, we aimed to develop a solution that minimizes these efforts and enables effortless firmware updates.

1.3 Solution

Creating a custom PCB that integrates **Blue Pill** with **ESP32 WROOM-32D**, which uses OTA feature of ESP to receive two

firmwares from a **Website** hosted by ESP itself, create **memory partitions** in ESP and initialize a **File System** in one of the partitions. Further on the STM side, we created a **Dual-Image Bootloader** that requests Firmware according to the user input, verifies it, allocates it in the flash memory and executes it. The firmware transfer is carried out via **UART.**

2. Custom PCB Layout

2.1 Introduction

This chapter describes the design and layout of the custom PCB of the project **CustomBoot-32**.The goal of the layout process was to integrate the **STM32** and **ESP32** on a compact, single-board solution while ensuring proper routing, reliable power distribution, and proper connection of the MCU's. All the connections that are made were made by referring to the datasheets.

2.2 List of Components

The following components are used:

- **ESP32-WROOM32D:** Wi-Fi and Bluetooth enabled MCU module used in this project to provide wireless connectivity and Over-The-Air (OTA) functionality.

- **STM32F103C8T6:** ARM Cortex-M3 based microcontroller that serves as the main application processor.

- **Type-C USB Module:** It feeds a 5 V supply which is then regulated down to 3.3 V for the microcontrollers.

- **CP2102N:** USB-to-UART converter used to interface the board with a computer for serial communication.

- **AMS1117-3.3V:** To convert 5V from the TYPE-C module to 3.3V

2.3 Circuit Connections

2.3.1 Type-C USB Module Connections

The main power supply is the USB TYPE-C power module which supplies 5V to the voltage regulator. In addition, the USB is also connected to the CP2102N module for debugging and serial monitoring of the ESP32.
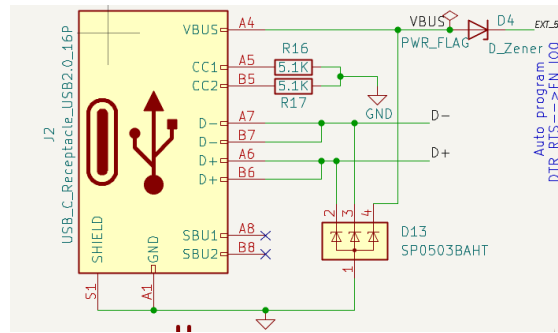
Figure 2.1: USB-C schematic connection

Figure 2.2: CP2102N schematic connection
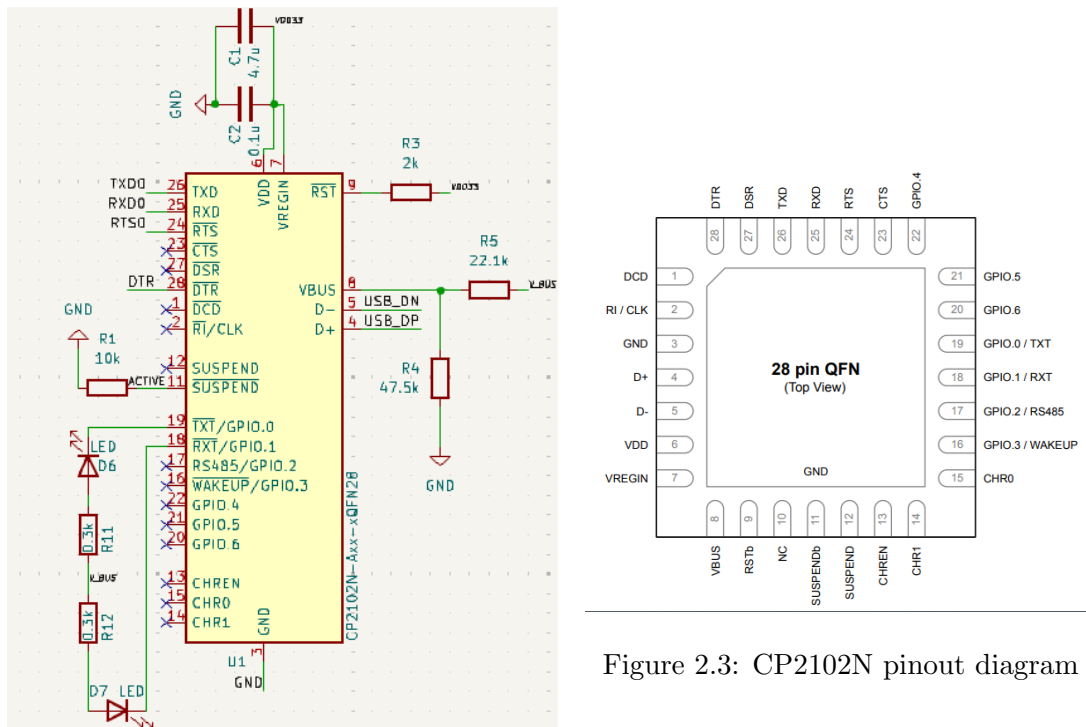
Figure 2.3: CP2102N pinout diagram

2.3.2 CP2102N Connections

The **CP2102N** is connected to the **UART0** (GPIO1,GPIO3) of the ESP32 because this is the default boot and console interface, which allows serial debugging and initial firmware flashing when the device is blank. It is also connected to the USB module via the **VBUS** pin for power and **D+** and **D-** lines for data transmission. The USB protocol sends data in differential form, **which means the**

logical '1' and '0' are represented by the **voltage difference** between the D+ and D- lines instead of a single signal level. This **improves noise immunity** and **allows high-speed communication.**

2.3.3 AMS1117 Connections

The **AMS1117** is connected directly to the **USB port**. The **Vin** pin of AMS is connected to the **VBUS** pin of USB which gives an input voltage of **5V**. This voltage is converted to **3.3V**, which is required to power the microcontrollers. It works by **dissipating the excess voltage as heat**, thereby regulating the input voltage 5V down to a stable 3.3V output.
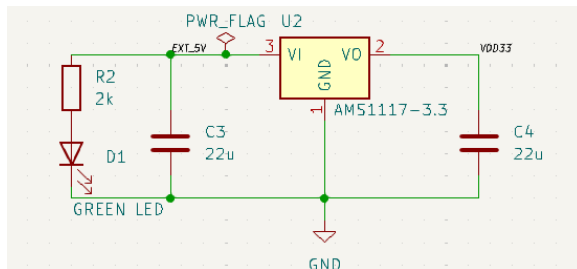

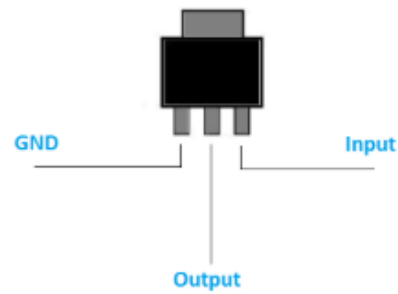Figure 2.4: AMS schematic connection

Figure 2.5: AMS pinout diagram

2.3.4 ESP-STM Connections

The **UART1** interface of the ESP32 (GPIO16 and GPIO17) is connected to **UART1** of the STM32 (PA9 and PA10), which enables the ESP32 to stream the firmware data directly to the STM32. Both microcontrollers are provided with individual **RESET** and **BOOT** switches to allow manual control and testing of the firmware loading process. All **VDD** pins of the STM32 are internally shorted and connected to small decoupling capacitors placed close to the pins. Decoupling capacitors act as **local energy reservoirs** and **filter high-frequency noise**; they supply instantaneous current during switching events and **prevent voltage fluctuations** on the power rail. In addition, most of the remaining I/O pins of both the ESP32 and the STM32 are routed to pin headers, so the board can also be used for **general-purpose prototyping** and is not restricted to this specific project.

2.3.5 STM Power Circuitry

The VBAT and VDD pins of the STM32 are internally shorted and connected to a group of decoupling capacitors placed close to the device. These capacitors stabilise

the supply voltage and suppress high-frequency noise caused by the fast switching of the internal digital logic. The 100 nF capacitors provide high-frequency decoupling, while the larger 10 µF capacitor acts as a local energy reservoir for lower-frequency transients.

In addition, the VDDA pin (analogue supply) is separated from the main digital 3.3 V rail by a ferrite bead and filtered using small capacitors (100 nF and 1 µF). This prevents digital noise from coupling into the analogue supply and ensures a clean and stable voltage for the ADC and other analogue circuitry.

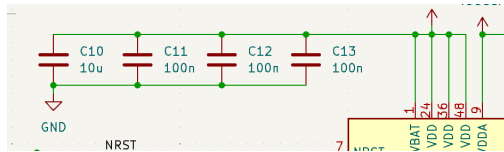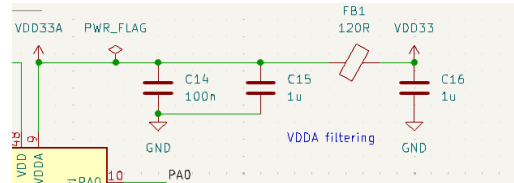Figure 2.6: Decoupling capacitors schematic

Figure 2.7: Analog circuit schematic

2.4 Routing

After finalizing the circuit design, the next step is PCB routing. In this step, we place the components according to our ease of connecting the components with each other according to our schematic. Proper routing is essential to ensure signal integrity, minimize noise and maintain reliable power distribution.

3. OTA Firmware Handling

3.1 Introduction

Over-The-Air (OTA) update refers to the process of wirelessly delivering new firmware or software to a device, thereby eliminating the dependency on physical connections such as USB interfaces or external programmers. This approach directly addresses our **problem statement** of repetitive hardware connections. By leveraging OTA, we can seamlessly transfer **updated firmware** to our development board, enabling faster, more efficient, and even remote updates with minimal effort.

3.2 Workflow

- User uploads firmware via a web page.
- ESP32 receives the file over Wi-Fi.
- Firmware is stored in SPIFFS partitions.

3.3 Memory Partition

The **4 MB** flash of the ESP32 is divided to store the main OTA firmware as well as to allocate memory for the **File System** that manages files received during OTA.

3.3.1 partitions.csv

In the root directory of the project, we create a **partitions.csv** file that defines the custom partition layout.

- **Name** – Label for the partition (e.g., `nvs`, `app0`, `storage`).
- **Type** – Defines the partition type:
  - `app` – Application (firmware).

Figure 3.1: Custom partitions defined in partitions.csv

- **data** – Data storage (e.g., NVS, SPIFFS, FAT).
- **SubType** – Further classification of the type:
  - For `app`: `factory`, `ota_0`, `ota_1`, etc.
  - For `data`: `nvs`, `ota`, `SPIFFS`, etc.
- **Offset** – Starting address of the partition in flash (hexadecimal).
- **Size** – Size of the partition (can be written in hex, KB, or MB, e.g., `0x10000`, `64K`, `1M`).

The last partition labeled **storage** is used to initialize the **File System**.

3.4   File System - SPIFFS

SPIFFS (SPI Flash File System) is a lightweight filesystem designed for embedded systems. It enables storing, reading, writing, and deleting files in flash, effectively making it act as a small disk. Within ESP-IDF, it is widely used for storing configuration files, web assets (`HTML`, `CSS`, `JS`), or other essential application data.

- **Optimized for small devices:** Efficient operation with limited RAM and flash.
- **Wear leveling:** Extends flash lifetime by evenly distributing writes.
- **Flat structure:** Supports only a flat namespace (no directories).
- **Ideal for embedded use:** Suitable for lightweight data and resources.

3.4.1 SPIFFS Initialization

The memory partition labeled `storage` is used to initialize SPIFFS. Below is its integration in both the `CMakeLists.txt` file and the project source code:

```cmake
cmake_minimum_required(VERSION 3.5)
set(SPIFFS_IMG_DIR "${CMAKE_SOURCE_DIR}/spiffs")

# Initialize the ESP-IDF build system
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
include(/home/varun/esp/esp-idf/components/spiffs/project_include.cmake)

project(OTA_ESP)
spiffs_create_partition_image(storage spiffs FLASH_IN_PROJECT)
```

Figure 3.2: CMakeLists.txt: specifying `storage` partition label

```c
static esp_err_t spiffs_init(void)
{
    esp_vfs_spiffs_conf_t conf = {
        .base_path = "/spiffs",
        .partition_label = "storage",
        .max_files = 5,
        .format_if_mount_failed = true};

    esp_err_t ret = esp_vfs_spiffs_register(&conf);
    if (ret != ESP_OK)
    {
        ESP_LOGE(TAG, "Failed to mount SPIFFS (%s)", esp_err_to_name(ret));
        return ret;
    }

    size_t total = 0, used = 0;
    ret = esp_spiffs_info("storage", &total, &used);
    if (ret != ESP_OK)
    {
        ESP_LOGE(TAG, "Failed to get SPIFFS info (%s)", esp_err_to_name(ret));
        return ret;
    }

    ESP_LOGI(TAG, "SPIFFS mounted. Total: %d bytes, Used: %d bytes", total, used);
    return ESP_OK;
}
```
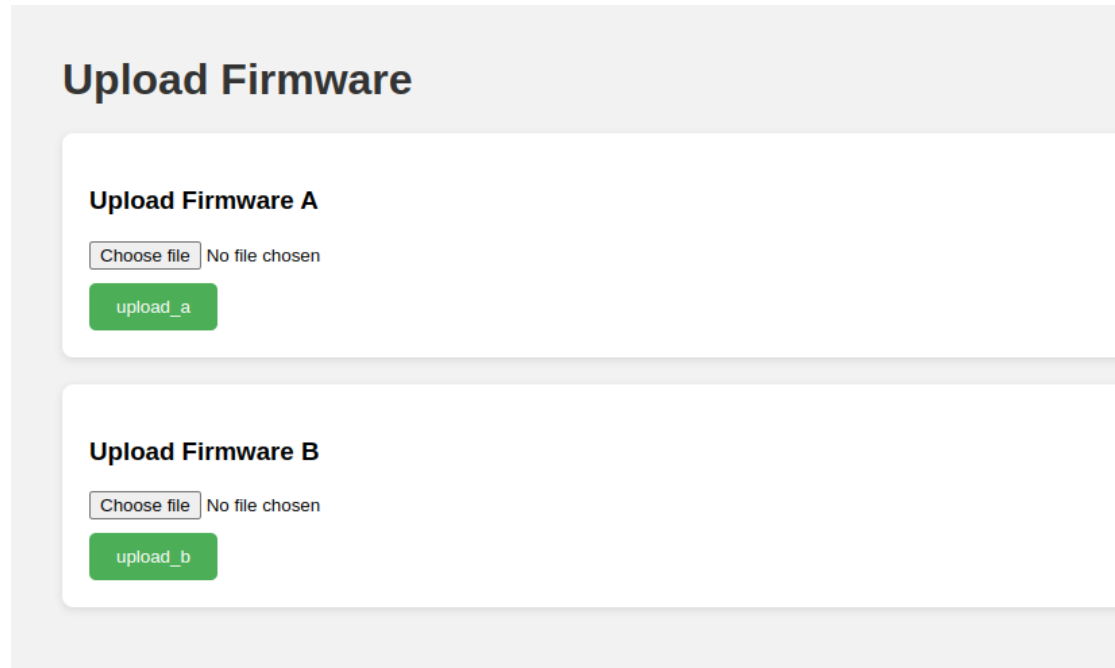
Figure 3.3: SPIFFS initialization in source code

3.5 Firmware Upload via Web Interface

We host a **web page** that provides a form for uploading firmware files. The **index.html** file is placed inside a folder in the project's root directory and served by the ESP32 using its assigned **IP address**.

The user must connect to the 2.4 GHz Wi-Fi network(because 5G isn't supported in ESP) and open the ESP32's IP address in a browser. The upload page appears as shown below:

Figure 3.4: Web interface for firmware upload

All web requests are processed by their respective handlers in the OTA `main.c` file.

3.6 Firmware transfer and extraction

To upload **.bin firmware files**, we use the **multipart/form-data** content type because:

- **Standard for file uploads:** Widely accepted method for binary file transfer via web forms.

- **Supports binary data:** Unlike *application/x-www-form-urlencoded*, it handles large binary files (firmware images) without corruption.

- **Data segmentation:** Each request is divided into multiple

parts, containing both metadata (filename, content type) and the actual data.

- **Reliable transmission:** Ensures that the ESP32 server correctly receives and stores firmware files.

3.6.1 File Extraction

A drawback of **multipart/form-data** is that it adds **headers and footers** around the file payload. Therefore, it is necessary to strip this additional data before saving the actual binary file.

```
POST /upload HTTP/1.1
Host: 192.168.4.1
Content-Type: multipart/form-data; boundary=----
    WebKitFormBoundary7MA4

------WebKitFormBoundary7MA4
Content-Disposition: form-data; name="file"; filename="firmware.
    bin"
Content-Type: application/octet-stream

...........<binary firmware data here>.............

------WebKitFormBoundary7MA4--
```

Listing 3.1: Multipart/Form-Data Example

You can refer the entire function here to study how the data is filtered in depth.

4. Bootloader

4.1 Introduction

Our project aims to handle two firmwares at a time, store it in the development board and execute them according to user input.

Obviously the default bootloader provided by STElectronics in the **SOC** isn't going to satisfy our needs. Hence we are creating a custom **Dual-Image Bootloader** that:

- Reads the state of PA0 to decide which firmware is requested.

- Sends a firmware request to the ESP32 over UART.

- Erases the flash memory region allocated for the application before writing new data.

- Receives firmware in multiple chunks via UART.

- Verifies the final bytes to ensure complete firmware reception.

- Integrity check (CRC) to confirm that the firmware is valid.

- Writes each received chunk into the erased flash region.

- Jumps to the application entry point to execute the new firmware.

- If no firmware is available from ESP32, the bootloader executes the default application stored in flash.

4.1.1 Libraries

HAL

Hardware Abstraction Layer (HAL) libraries provide a standardized software interface that abstracts the low-level hardware details of microcontrollers, allowing developers to control peripherals such as GPIO, UART, I^2C, SPI, ADC, and timers without direct register programming. This library is provided by STMicroelectronics. However it has its own disadvantages:

- Introduces extra code overhead, increasing memory usage.

- Slower execution compared to direct register-level programming.

- Less suitable for strict real-time or resource-constrained applications.

- Provides limited fine-grained control over peripherals.

- Can restrict optimization for specific hardware features.

Our STM32F103C8T6 has **64 kB flash**. Hence it is extremely necessary that our bootloader is well optimized and small in size. Hence, though HAL is easy to implement, it isn't the best option.

libopencm3

libopencm3 is an open-source, community-driven firmware library for ARM Cortex-M microcontrollers. Unlike vendor-specific libraries (such as HAL), it provides a lightweight, consistent, and low-level API to directly access hardware peripherals across multiple vendors. It is widely used for bare-metal development where efficiency and fine-grained control are required.

Advantages:

- Open-source and community-maintained, with active contributions.

- Lightweight and efficient compared to vendor HAL libraries.

- Provides low-level access to registers while keeping code readable.

- Does not introduce extra code overhead, reducing firmware size, hence perfect for us.

Both have their own advantages. Since HAL was beginner friendly and the documentation was well-structured, we decided to start implementation of our bootloader with HAL libraries.

4.1.2 STM32CubeIDE

STM32CubeIDE is STMicroelectronics' official IDE for STM32 microcontrollers. It integrates code editing, peripheral configuration (via CubeMX), compiling, and debugging in one platform.

Hence, we use STM32CubeIDE for easier peripheral setup, rapid firmware development, and debugging support during our project. Refer this for more information.
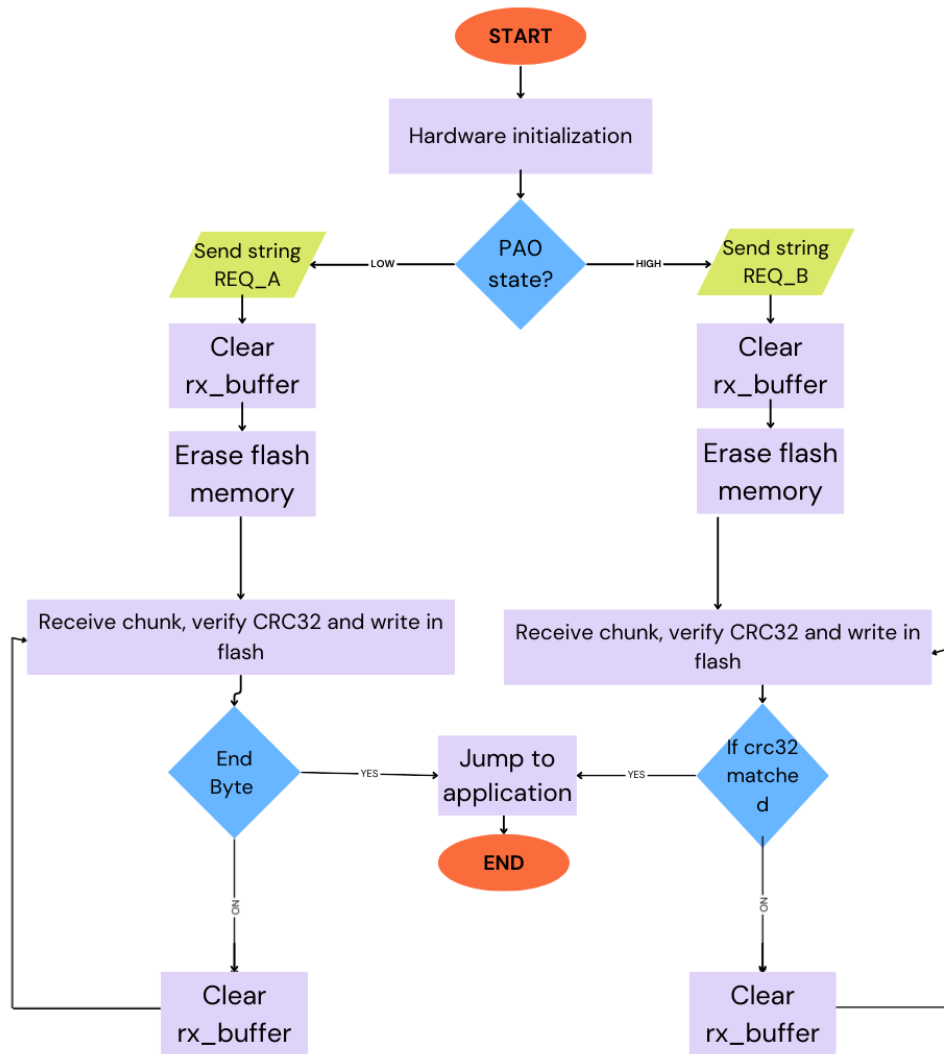
4.2 Flowchart

Figure 4.1: Bootloader Flowchart

4.3 Functions

4.3.1 Hardware initialization

In an STM32 bootloader, **hardware initialization functions** prepare the system before firmware loading. They configure the system clock, initialize Flash interface, set up GPIOs, enable communication, Following links provide the hardware initialization functions used in our bootloader:

- **Clock Initialization (External Crystal Oscillator)**
  Configures system clocks based on the chosen oscillator settings. You can configure according to your own settings.

- **GPIO Initialization**
  Only PA0 and PC13 are configured, as they are used by the bootloader. Anyways, all peripherals are deinitialized before handing control to the application.

- **UART initialization**
  This function initializes the UART used for receiving firmware from ESP according to the necessary configurations.

- **Flash Erase**
  Flash memory in STM32 is organized in pages (1 KB each on the Blue Pill). Individual bytes cannot be erased; instead, entire pages must be cleared. This function erases the specified number of pages, which can be adjusted as needed on line 171 of code.

- **Flash Write**
  Writes a data chunk from memory into flash at the specified address, applying the correct offset from the base address as provided by function parameters.

- **Chunk receiving**
  Doesn't have default application handling and CRC32, because the complete function is written in LibOpenCM3, which is properly explained in **chapter 5.**

4.4 LibOpenCM3 implementation

Till this point we were using HAL libraries. However we noticed that the firmware was already above 13 KB in size and we haven't even implemented **CRC32**. Hence we decided to switch into LibOpenCM3 from this point. Following is the entire bootloader written in LibOpenCM3 with same function names and with hardware CRC32 implementation as well:

- LibOpenCM3 Bootloader

4.4.1 Hardware CRC32

CRC32 (Cyclic Redundancy Check, 32-bit) is an error-detection algorithm that produces a 32-bit checksum for a block of data. It operates by dividing the data using a fixed generator polynomial:

$$G(x) = 0x04C11DB7$$

The remainder of this division is used as the checksum. At the receiver side, the same computation is repeated to verify data integrity and detect errors during transmission or storage.

The **STM32F103C8T6** provides a hardware CRC unit that simplifies this process. When a 32-bit word is written into the CRC data register, the hardware automatically updates the CRC value. Feeding additional words without resetting the unit causes the CRC engine to accumulate results, combining the new data with the previous calculation.

This feature enables efficient validation of large data streams: for example, chunks of firmware received from the ESP32 can be written word by word into the CRC unit, and the final CRC value corresponds to the checksum of the entire block.

- CRC32 hardware function(STM)

Figure 3. Step-by-step CRC computing example

bindex	Execution step	Binary format	Hex
	Crc = Initial_Crc ^ Input_Data	1 1 1 1 1 1 1 1 (*Initial_Crc*)	0xFF
		^ 1 1 0 0 0 0 0 1 (*Input_Data*)	0xC1
0	Crc << 1	[0] 0 1 1 1 1 1 0	0x3E
1	Crc << 1	[0] 1 1 1 1 1 0 0	0x7C
2	Crc << 1	[1] 1 1 1 1 0 0 0	0xF8
	Crc = Crc ^ POLY	[1] 1 1 1 0 0 0 0	0xF0
		1 1 0 0 1 0 1 1 (*POLY*)	0xCB
3	Crc << 1	[0] 0 1 1 1 0 1 1	0x3B
4	Crc << 1	0 1 1 1 0 1 1 0	0x76
5	Crc << 1	[1] 1 1 0 1 1 0 0	0xEC
	Crc = Crc ^ POLY	[1] 1 0 1 1 0 0 0	0xD8
		^ 1 1 0 0 1 0 1 1 (*POLY*)	0xCB
6	Crc << 1	[0] 0 0 1 0 0 1 1	0x13
7	Crc << 1	[0] 0 1 0 0 1 1 0	0x26
	Crc (Returned value)	0 1 0 0 1 1 0 0	0x4C

1. The returned CRC value (0x4C) has been verified by the CRC peripheral on STM32F37x products, with the above mentioned configurations.

2. This routine has been implemented under CrcSoftwareFunc function in the CRC_usage example.

Figure 4.2:

However there is no such hardware registsers fot CRC32 in ESP, hence we implemented software CRC32 calculation in ESP that virtually mimics the CRC32 calculation done by STM's hardware. This is the exact function that calculates CRC32 in ESP.

4.4.2 Default App

In case ESP doesn't have any firmware to send to STM, the entire development board will be in a hang condition and user will be clueless about the state of board. Hence, it is necessary to implement a default application so that user knows something is wrong.

In this case ESP sends a chunk of 520 bytes, all bytes being 0xAA. This function in STM bootloader checks wheather all the bytes are 0xAA, denoting no firware available in ESP. If the condition is true, STM jumps to default application, which is just a rapidly blinking LED.

5. Binary File Transfer

5.1 Introduction

This chapter explains how we achieved binary file transfer from ESP32 to STM32 and how we implemented a lightweight yet reliable **Custom File Protocol** to accomplish this task.

5.2 File Transfer Protocols

File transfer protocols are standardized communication methods that enable computers or embedded devices to exchange files reliably over networks or serial links. They define rules for how data is structured, transmitted, acknowledged, and verified to ensure error-free delivery.

In the domain of embedded systems, **XMODEM, YMODEM, and ZMODEM** are widely used over serial connections. These protocols employ block-based transmission, error detection, retransmission on failure, and acknowledgment mechanisms. While effective, they differ in complexity and efficiency:

- **XMODEM**: Simple and lightweight, uses 128-byte blocks with CRC16 for error detection.

- **YMODEM**: Extension of XMODEM, supports larger block sizes (1 KB), batch file transfers, and additional metadata.

- **ZMODEM**: More advanced, with streaming support, resume capability, and better efficiency over noisy channels.

Each protocol has its trade-offs between simplicity, reliability, and overhead. For our use case, XMODEM was too limited, while YMODEM and ZMODEM introduced unnecessary complexity. Thus, inspired by these well-established protocols, we designed a minimal **Custom File Protocol** optimized for binary firmware transfer over UART.

5.2.1 Custom File Protocol

Our objective is to transfer binary firmware images from ESP32 to STM32. Since we only handle two firmware files at a time and operate over a simple UART interface, features such as parallel transfers or batch metadata (offered by YMODEM/ZMODEM) were unnecessary. Instead, we focused on a balance between simplicity and reliability.

The protocol is based on fixed-size **512-byte chunks**, chosen as an optimal size for flash programming and UART transfer efficiency. Each transmission packet follows a well-defined structure:

1. **Start Byte (1 byte)**: Marks the beginning of a packet, ensuring synchronization.

2. **Length Field (2 bytes)**: Indicates the size of the payload. While this is always 512 bytes in our implementation, it provides flexibility for future scalability.

3. **Payload (512 bytes)**: The actual firmware data block being transmitted.

4. **CRC32 Checksum (4 bytes)**: Ensures data integrity. The STM32 verifies each payload using hardware-accelerated CRC32 before writing it to flash.

5. **Stop Byte (1 byte)**: Marks the end of a packet. This also helps the receiver detect whether the firmware has been fully received.

Compared to XMODEM, our protocol introduces larger block sizes and CRC32 (instead of CRC16) for improved reliability. The use of start/stop delimiters further enhances packet framing and error recovery.

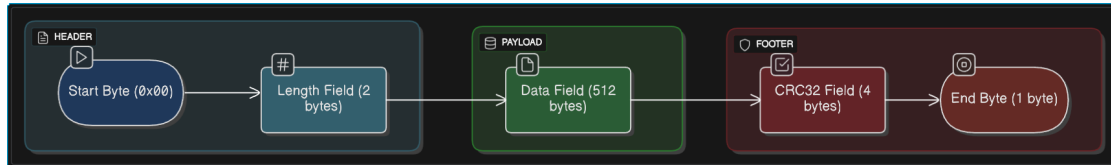The final packet structure of the protocol is illustrated in Figure 5.1.

Figure 5.1: Custom File Protocol

5.3 ESP implementation

We searched for examples of file protocol implementation in C and found this GitHub repository. We found the approach of initializing a struct to define the protocol format particularly effective, as it provides a unified structure capable of storing multiple data types (for example, a 32-bit CRC field, an 8-bit payload, start and end bytes, and a 16-bit length field). Based on this idea, we created a struct for our file protocol as follows:

```
struct Chunk_file
{
    uint8_t start;
    uint16_t length;
    uint8_t data[DATA_SIZE]; // DATA_SIZE = 512
    uint8_t crc32[4];
    uint8_t end;
};
```

Listing 5.1: File Protocol Struct

Further we arranged the chunk to be sent in steps. Initially we set start byte as 0x00. As you can see below, initially the 512 bytes of the firmware are loaded in **Firmware.data** and the actual bytes loaded in it are saved in **data-read**.

If data-read is less than 512 bytes, it will denote end of firmware. Now, in STM, 0xFF is considered as empty flash. Hence when data-read is less than 512 bytes, ESP appends 0xFFin the remaining bytes to make the chunk of 512 bytes(line 7, figure 5.2). This is the reason our **Firmware.Length** always stays constant. Also the end byte is set to **0xAF**, denoting end of transmission.

```c
size_t data_read = fread(Firmware.data, 1, DATA_SIZE, f);

        if (data_read == 0) // EOF
            break;

        // Check if last chunk
        if (data_read < DATA_SIZE)
        {
            memset(&Firmware.data[data_read], 0xFF, DATA_SIZE -
                data_read);
            Firmware.end = 0xAF; // Final chunk
        }
        else
        {
            // Peek ahead: if no more data, this is final chunk
            int c = fgetc(f);
            if (c == EOF)
                Firmware.end = 0xAF;
            else
            {
                ungetc(c, f);
                Firmware.end = 0x00;
            }
        }
```

Listing 5.2: End chunk condition

If data-read is equal to 512, the end byte is 0x00. Further, CRC32 is calculated using the function provided in section 4.4.1.

```
1  // Fill uint8_t Packet [520]
2          Packet [1] = (data_read >> 8) & 0xFF;
3          Packet [2] = data_read & 0xFF;
4          memcpy(&Packet [3], Firmware.data, DATA_SIZE);
5          Packet [515 + 0] = (crc_ESP >> 24) & 0xFF; // MSB
6          Packet [515 + 1] = (crc_ESP >> 16) & 0xFF;
7          Packet [515 + 2] = (crc_ESP >> 8) & 0xFF;
8          Packet [515 + 3] = (crc_ESP >> 0) & 0xFF; // LSB
9          Packet [7 + DATA_SIZE] = Firmware.end;
10
11         int sent = uart_write_bytes(UART_NUM, (const char *)Packet
            , 8 + DATA_SIZE);
```

Listing 5.3: Preparing Packet to send

As per the file protocol, we define an array named Packet consisting of 520 elements. This array is filled in a structured manner, the first and last bytes represent the start and end markers, respectively, as per the implemented conditions. The following two bytes store the 16-bit length field, transmitted in the form of two separate bytes. After this, the array holds a 512-byte payload, which is followed by the last four bytes contain the CRC32 checksum, also transmitted in the form of four bytes. Visit here for complete function.

On the STM32 side, the CRC32 checksum is recalculated and verified. If the calculated value matches the received checksum, the STM32 sends an acknowledgement string back to the ESP. On the ESP side, the firmware runs within a continuous while loop, which terminates only upon receiving this acknowledgement string, as provided below:

```
while (1)
        {
            memset(rx_buffer, 0, BUF_SIZE);
            uart_read_bytes(UART_NUM, rx_buffer, BUF_SIZE - 1,
                pdMS_TO_TICKS(1000));

            if (strncmp((char *)rx_buffer, "crc", 3) == 0)
            {
                ESP_LOGI(TAG, "CRC of ESP: %lx\r", crc_ESP);
                ESP_LOGI(TAG, "CRC of STM: %s\r", (char*)&
                    rx_buffer[15]);
                break;
            }
            if (Firmware.end == 0xAF)
            {
                ESP_LOGI(TAG, "Final Chunk Sent.");
                receiveAck();
            }
        }
```

Listing 5.4: Receiving acknowledgement from STM

5.4 STM implementation

Our Bootloader written using LibOpenCM3 libraries accordingly receives the data in the same format as defined in the ESP implementation.

The initial part is sending a request to ESP. **PA0 GPIO pin's** state is verified and accordingly firmware is requested. If state is low, a string **SEND-A** is sent and if state is high, then a string **SEND-B** is sent and the ESP accordingly sends the firmware. Before sending request, EraseUserApplication() is called that clears flash to write the firmware. A variable **offest** is declared which makes sure the next chunk is written above the previous chunk and not overwritten.

```
1
2  void ReceiveChunkOverUART(uint32_t addr, const char *str) // str
       is passed as argument according to firmware needed
3  {
4
5      EraseUserApplication(addr);
6      uint32_t offset = 0;
7      uart_transmit(USART1, str, 8);
```

Listing 5.5: Firmware request

Further, the function receives the chunk and remains in the receiving state until all the 520 bytes are received. Then, isAllBool() function's value is checked. If it is true that means there is no data inside ESP to transmit. In this case, a default application is executed.

```
1
2      while (1)
3        {
4            // memset(rx_buffer, 0, CHUNK_SIZE);
5
6            uart_receive_blocking(USART1, rx_buffer, CHUNK_SIZE);
7
8            if (is_all_AA(rx_buffer, 520))
9            {
10                JumpToAddress((uint32_t)DEFAULT_APP_ADDRESS);
11            }
```

Listing 5.6: Default application condition

Further, CRC32 values are received in 4 bytes, which are recombined and CRC32 is calculated using the hardware of CRC32 in STM by crc32() function. If the calculated CRC32 matches the received CRC32, then only the chunk of data is written in the flash by WriteUserApplication() and offset is increased by 512 to make sure next chunk is written above the previous chunk. Also an acknowledgement string is sent to ESP so that ESP sends next chunk. Acknowledgement is sent by uart-print-hex() function.

```
if (rx_buffer[CHUNK_SIZE - 1] == 0xAF)
        {
                crc_STM = 0x00;
                crc_ESP = 0x00;
                crc_STM = crc32_libopencm3_style(&rx_buffer[3], 512);
                crc_ESP = ((uint8_t)rx_buffer[515] << 24) |
                          ((uint8_t)rx_buffer[516] << 16) |
                          ((uint8_t)rx_buffer[517] << 8) |
                          ((uint8_t)rx_buffer[518]);

                if (crc_STM == crc_ESP)
                {
                    WriteUserApplication((uint32_t)addr, (uint32_t *)&
                        rx_buffer[3], 128, offset);

                    JumpToAddress((uint32_t)addr);
                }
        }
        else if (rx_buffer[CHUNK_SIZE - 1] == 0x00)
        {
                crc_STM = 0x00;
                crc_ESP = 0x00;
                crc_STM = crc32_libopencm3_style(&rx_buffer[3], 512);
                crc_ESP = ((uint8_t)rx_buffer[515] << 24) |
                          ((uint8_t)rx_buffer[516] << 16) |
                          ((uint8_t)rx_buffer[517] << 8) |
                          ((uint8_t)rx_buffer[518]);

                if (crc_STM == crc_ESP)
                {
                    WriteUserApplication((uint32_t)addr, (uint32_t *)&
                        rx_buffer[3], 128, offset);

                    offset += 512;


                        uart_print_hex(crc_STM);
                }
        }
    }
```

Listing 5.7: Data handling

Also, end byte is checked everytime before doing alll the above operations. If end byte is **0xAF**, it means it is the last chunk of data. So that chunk is written in flash and we jump to execute the app using JumpToApplication() function.

In this way, we successfully implement our custom File Protocol in ESP and send binary files from ESP to STM, receive it in STM, verify its integrity using CRC32, erase flash, write the data in flash and efficiently handle file transfer between ESP and STM.

Appendix

Glossary

ACK (Acknowledgement) Signal confirming successful data receipt.

Address A memory location used for storing or fetching data.

Application / Default Application The main user program executed after boot.

Block Fixed-size data unit (128, 512, or 1024 bytes).

Bootloader A small program that initializes hardware and loads applications.

BUF_SIZE Constant defining buffer memory size.

Chunk A structured 520-byte packet with 512-byte payload.

CRC32 32-bit cyclic redundancy check for stronger integrity validation.

Checksum Calculated 32-bit value verifying data integrity.

Data Integrity Assurance that transmitted or stored data remains unaltered.

DATA_SIZE Constant defining payload size (512 bytes).

EOF (End of File) Marker indicating completion of file transmission.

End Byte Special byte marking packet termination.

ESP_LOGI Logging function for ESP32 firmware.

fgetc / fread / ungetc Standard C I/O functions for file reading.

Firmware Software programmed into embedded system flash memory.

Flash / Flash Programming Process of writing data into non-volatile flash.

Function Defined block of code for specific tasks (e.g., UART handling).

GPIO (PA0 pin) General-purpose input/output pin on STM32.

Hardware CRC CRC engine implemented in microcontroller hardware.

Hexadecimal values Data represented in base-16 format (e.g., 0xAA).

Length Field 16-bit field defining payload size in a packet.

LibOpenCM3 Lightweight, bare-metal, open-source peripheral library for ARM Cortex-M.

Metadata Additional info about file.

MSB / LSB Most/least significant byte of a multi-byte value.

Offset Variable controlling flash write position from starting address.

Packet 520-byte structured transmission unit.

PA0 STM32 pin used as GPIO input/output for decision making of firmware.

Payload Actual transmitted data (512 bytes).

pdMS_TO_TICKS FreeRTOS macro converting milliseconds to ticks.

Protocol Set of rules for communication (e.g., XMODEM, YMODEM).

Retransmission Re-sending lost or corrupted packets.

rx_buffer Buffer holding received UART data temporarily.

SEND-A / SEND-B ESP firmware request commands for STM.

Start Byte Special marker (0x00) for packet initiation.

Streaming Support Continuous data transfer (ZMODEM feature).

Struct User-defined grouping of related variables of same or different data types.

Synchronization Alignment of sender and receiver communication.

TAG Identifier string used in ESP logging.

Transmission Sending of data over a medium.

uart_transmit() Function transmitting UART data.

UART / USART1 Serial communication peripherals in STM32.

uart_print_hex() Function printing bytes in hex format.

uart_read_bytes() Function reading data from UART buffer.

uart_receive_blocking() Blocking UART receive function.

uart_write_bytes() Function writing data to UART.

References

Understanding GIT
ESP32 WROOM-32D datasheet
STM32F103C8T6 datasheet
CP2102 datasheet
AMS1117 datasheet
Learning KiCad
Designing a full STM blue pill in KiCad
Bootloader basics (EmbeTronicx)
Getting Started with bootloader(EmbeddedInventor)
Another blog on bootloader
Video on bootloader
Getting started with STM32
Bare-metal UART STM32
Programming STM32 using USB-to-TTL
CRC32 for STM
LibOpenCM3 documentation
Git repo to refer LibOpenCM3 implementation
ESP32 devkit schematic(V4)
ESP-IDF programming basics
ESP32 UART
File systems in ESP32
SPIFFS
FreeRTOS
Web server handling in ESP File transfer protocols
C implementation of file protocols at sender's side