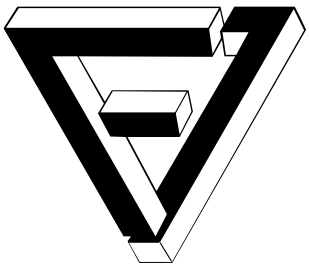
Masaryk University
Faculty of Informatics



Remote firmware flashing for STM32 boards via CAN bus

Bachelor's Thesis

Adam Valt

Brno, 2025

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools:

- Grammarly for grammar check and to improve my writing style.
- ChatGPT to improve my writing style and code troubleshooting.

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Adam Valt

Advisor: Mgr. Ján Labuda

Acknowledgements

I would like to thank everyone who helped me make this thesis possible. You know your names. Finally, I would like to thank my family and friends for their support and patience.

Abstract

This thesis presents the design and implementation of a system for remote firmware flashing and updating of STM32-based embedded boards integrated into a Formula Student vehicle. These embedded systems, interconnected via a shared CAN bus, form a critical part of the vehicle. The implemented solution addresses the limitations of traditional flashing methods that require physical access to each individual board, which is often impractical during rapid development cycles and testing phases. The system enables over-the-air firmware updates utilizing the CAN network, with support for selective flashing of specific nodes. The work considers the constraints and architecture of the current vehicle hardware and software. The final outcome includes a bootloader capable of using CANOpen protocol and firmware responsible for the UI and flashing process.

Keywords

Embedded Systems, STM32, Formula Student, CAN bus, Firmware update, CANopen, Controller Area Network, Mongoose, OTA

Contents

Abbreviations

CAN Controller Area Network 4

ECU Electronic Control Unit 4

FOTA Firmware Over The Air 6

HAL Hardware abstraction layer 5

LwIP Lightweight IP 26

MCU microcontroller 4

NMT Network Management protocol 15

OD Object Dictionary 16

SBL secondary bootloader 15

SDO Service Data Object 15

URB Universal Reversal Board 15

VCU Vehicle Control Unit 5

XCP Universal Measurement and Calibration Protocol 10

I Introduction

A challenging aspect of building a Formula Student vehicle is the updating of vehicle software during development. Like any modern vehicle, it relies heavily on a sophisticated network of sensors, electronic systems, and control units, all of which require frequent improvements and bug fixes by the members of our software team. The process of updating our Formula Student vehicle was traditionally tedious and time-consuming. The manual process involved dismantling a section of a vehicle, removing the necessary hardware device, and connecting it to the computer. Our software team wanted an easier way to update the functionality of the vehicle, which led to the system outlined in this work.

This thesis designs and implements a system that updates our Formula Student vehicle software distributed on multiple control units without physical access. The system improves development speed and efficiency by eliminating the need for manual disassembly and flashing of ECUs, allowing remote firmware updates via a web interface. There are some limitations imposed by existing solutions in the vehicle, mainly in terms of how communication works between embedded boards. The emphasis is on open-source software, as no comparable solution currently exists.

The work started by prototyping individual functional units and getting feedback from the team. Initially, I developed individual functional units and then combined them into one functional system.

The Chapter 3 presents limitations of the current vehicle state. Next, Chapter 4 is exploring possible solutions to the problem, and working around presented limitations. The Section 4.6 explores methods of sending large amounts of data using a Controller Area Network bus. Section 4.5 offers a deep dive into development of a custom bootloader. Section 5.5 describes the process of developing a user interface.

The system has been successfully used directly on our Formula Student vehicle. The Chapter 6 discusses the impact on time savings, development efficiency, and the user experience from the perspective of our software team.

1.1 Formula Student

Formula Student (FS) is an international design competition for students. It is the european version of the original *Formula SAE*[1] competition which originated in the USA. The goal is to build a monopost type vehicle for one person capable of competing with other teams in static and dynamic disciplines. Points are awarded in each of these categories and the overall winner is determined by the combined score across all disciplines [1]. One of the static disciplines is *Engineering Design Event* where the contents of this work might be presented and discussed with the competition judges. The competitions are held since 1981 and they are currently active in 24 countries all over the world [2].

My work takes place in a *TU BRNO Racing* team at the Brno University of Technology. The team has participated in Formula Student competitions since 2010. In the past four years, the focus has shifted from combustion vehicles exclusively on electric vehicles. Over its history, the team has designed and built 10 internal combustion engine and 4 electric monoposts [3]. The importance of software features has become prominent only with the transition to electric monoposts. It has opened a lot of opportunities to improve the monopost with advanced software algorithms such as [4]. The team currently has a section entirely focused on software development. The work presented in this thesis aims to support the development process of our team members.

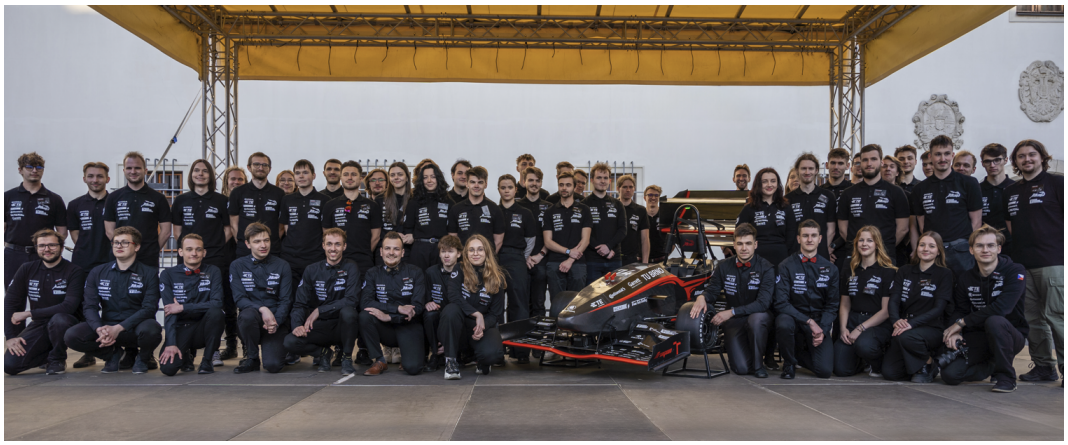


Figure 1.1: Team photo with eD4 monopost

[1] https://www.fsaeonline.com/

2 Preliminaries

The world of embedded programming might be often unfamiliar to many software developers. This chapter introduces essential concepts in embedded programming relevant to this thesis, which will serve as a foundation in the following chapters.

2.1 Embedded system

According to Barr and Massa [5, p. 1], an embedded system is a combination of computer hardware and software designed to perform a specific function and it is frequently a component within a large system. Modern cars can integrate as many as 150 Electronic Control Unit (ECU)s [6]. In the case of our monopost vehicle this number is apprximately 15 ECUs. Each embedded system is responsible for specific functions within the vehicle and they usually communicate with each other via data buses.

2.2 Microcontroller

microcontroller (MCU) is an semiconductor integrating CPU, memory and I/O auxiliaries in a single chip [7]. The ECUs in our vehicle use MCUs from the manufacturer *STMicroelectronics*. The product family name is *STM32*[1]. Many ECUs in our system share the same processor model, although some variations exist. These microcontrollers were chosen by our electrical team for their integrated memory and peripheral support. Selected list of microcontrollers in our vehicle can be found in table 4.1.

2.3 Electronic Control Unit

This represents the core module in the already defined embedded system. According to Schmidgall [8, p. 21], this term is also established as a synonym for an embedded system.

An ECU generally consists of an microcontroller, optional external flash memory and/or SDRAM memory [9]. These units typically include power delivery for the microcontroller, communication bus interfaces and connections to sensors or external devices. The microcontroller handles

[1] Product page: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

computation and additional memory is used when internal memory is insufficient. Regarding the communication interfaces the primary focus of this thesis is on the ethernet interface and the Controller Area Network (CAN) interface, which is later discussed in section 2.9. Other sensors and specific external devices are not discussed as they fall outside the scope of this thesis.

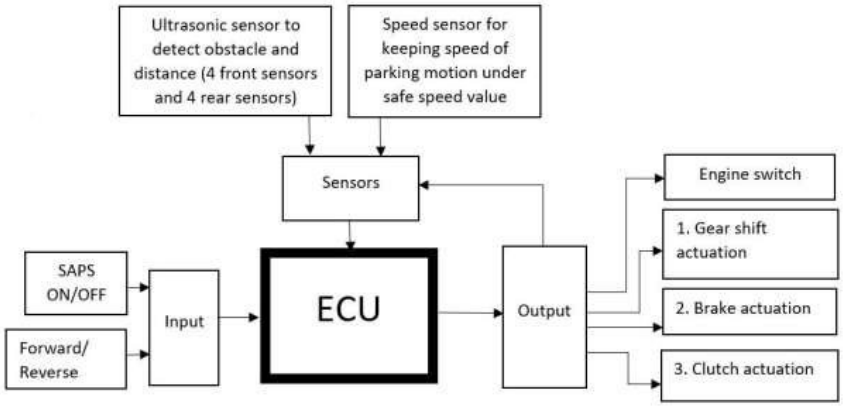


Figure 2.1: Block diagram of the ECU system from Ohol and Kaware [10, p. 2]

The main purpose of ECUs is to take information from large number of sensors and send commands to connected devices and peripherals. In our TU Brno Racing eD4 vehicle, one of the ECUs, the Vehicle Control Unit (VCU) plays a central role in the decision making of the vehicle. It is responsible maintaining the overall state of the vehicle, coordinating the safe operation of the vehicle, and issuing control commands to other ECUs. The remaining ECUs mostly act as a separate independent units that execute control commands from the VCU and handle their specific tasks without affecting the broader vehicle state.

2.4 Firmware

The term is defined in an international standard [11, Chapter 4.14] as a "combination of a hardware device and computer instructions or computer data that reside as read-only software on the hardware device". Firmware is persistent in the flash memory and remains there even after the device loses power [12, p. 1]. In the context of this work, the firmware consists of:

- The software responsible for the core functionality of the hardware device.
- Hardware abstraction layer (HAL) drivers provided from the manufacturer. They are responsible for the correct initialization and interacton with the hardware and peripheral devices [13]. They serve as a foundation for the rest of the firmware.

- Libraries that provide specialized high-level functionalities such as *CANopenlibrary* and *Mongoose*. This is discussed in the section 3.2 and section 5.6 respectively.

The firmware is not a static piece of code that does not ever change. It usually recieves numerous updates from the developers during the development phase. This process is described in detail in the next section.

2.5 Firmware flashing

Firmware flashing refers to the process of loading a firmware into the memory of the hardware device [14]. This term is often used interchangeably with other terms, such as firmware updating [15] or upgrading [16] or OTA update [17]. Specific variant of this mechanism is also known as Firmware Over The Air (FOTA) or Over The Air (OTA) [17]. Flashing is necessary for implementing new features or fixing bugs in the device firmware.

2.5.1 Flashing the device manually

Microcontrollers are typically equipped with a flashing mechanism provided by the manufacturer [18]. We have used that mechanism to flash the ECUs until now. The manual process is slow and requires disassembling the vehicle to access the physical units, making it a considerable inefficiency in the development process. It also requires an external programming device and a specific connector suitable for the programmed ECU. The computer has to be connected to the ECU with external programmer device as seen in fig. 2.2. We are using *STLINK-V3SET* [2] as a programmer device. After succesful connection the device is flashed using one of the supported applications, for example `OpenOCD` [3] or `STM32CubeProgrammer`[4]. This work will cover them in section 5.1.2.

There are both advantages and disadvantages to this approach. The primary advantage of the STLINK-V3SET programmer device is the provided additional functionality besides flashing of the device, such as debugging functionality. However, the requirement for physical connection makes this method less desirable during rapid development.

It is important to note that the custom system developed in this thesis does not include debugging capabilities. This was a deliberate design choice, as debugging falls outside the intended scope of the functionality.

[2] Device manual available from: https://www.st.com/resource/en/user_manual/um2448-stlinkv3set-debuggerprogrammer-for-stm8-and-stm32-stmicroelectronics.pdf
[3] Open On-Chip Debugger available from: https://openocd.org/
[4] Software manual available from: https://www.st.com/resource/en/user_manual/um2237-stm32cubeprogrammer-software-description-stmicroelectronics.pdf

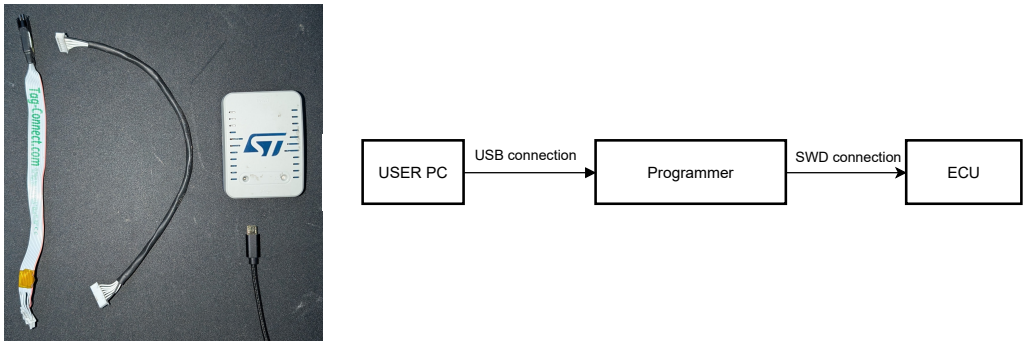


Figure 2.2: Required equipment (on the left) and diagram of connection (on the right) for manual flashing

2.6 Flash memory

STM32 microcontrollers feature onboard flash memory. This memory is persistent and can be accessed for both read and write, based on the device configuration. It is important to note that the flash memory is organized into sectors. Writing into memory sector usually requires erasing the whole memory sector which contains the desired memory address, otherwise the write is not succesful. According to [19, pp. 90–98], "Programming in a previously programmed address is not allowed except if the data to write are all 0." This somewhat complicates the writing process.

This memory is usualy limited due to cost reasons which requires efficient memory managment. A linker file determines how memory is allocated within the microcontroller, ensuring the firmware is correctly placed and executable.

2.7 Communication bus

A comunication bus provides the physical layer for communication in the ISO/OSI model. In our system, the CAN bus acts as the primary communication bus, connecting all ECUs within the vehicle.

2.8 Communication protocol

While communication bus refers to the physical system present in the vehicle, the communication protocol is a set of rules that defines how data should be structured and transmitted over the physical bus [20]. These rules are typically implemented through software or using peripheral devices for offloading the functionality to the hardware.

2.9 Controller Area Network

The Controller Area Network (CAN) bus is a robust, standardized communication protocol developed by Bosch in the 1980s [21, p. XV]. It facilitates communication among ECUs over a two-wire differential interface. The protocol is widely adopted in the automotive industry due to its cost-effective wiring, high fault tolerance, and proven reliability under real-time constraints [22]. Although enhanced variants such as CAN FD (Flexible Data-rate) offer improved bandwidth and efficiency, our system adheres to the standard CAN protocol for its simplicity and compatibility with existing hardware.

2.10 Bootloader

This is the firmware responsible for loading the application firmware into memory. The reasoning behind bootloader is presented here [23].

2.11 Embedded HTTP Server

An HTTP server running on the embedded device allows remote interaction with the ECU, providing a user-friendly interface for firmware management. This server eliminates the need for specialized software and enables communication with the ECU through any standard web browser or any software capable of HTTP requests.

3 Limitations

The primary challenges in developing the desired system involve dealing with hardware limitations, such as the configurations of the data bus already used by the boards for communication and the protocol implemented over the data bus. The following sections provide an in-depth examination of the specific issues. This thesis operates within the constraints of the current vehicle state.

3.1 Communication bus in the vehicle

The communication architecture in the vehicle relies heavily on the CAN bus as the only viable communication backbone for the system. The CAN bus has been chosen by our electrical team as an exclusive method of communication between ECUs due to the simplicity of wiring. For this reason, we have to utilize the CAN bus for firmware flashing, as it is the only communication bus that connects all ECUs within the vehicle, thus allowing a unified solution to update each board on the vehicle. An alternative approach would involve connecting each ECU via other bus systems or ethernet. Doing this would significantly increase costs, and our team would not stay within a strict budget for building the vehicle. Utilizing the CAN bus reduces the cost of physical wiring and hardware complexity in exchange for increased software complexity [24]. Consequently, the implementation of this thesis relies exclusively on the CAN bus interface.

3.2 Communication protocol in the vehicle

CANopen protocol, specifically *CANopenNode* [25], is implemented over the CAN bus in the current software stack. The automotive electronics manufacturing company *Vector* describes it as an *Layer 7* in the *ISO/OSI* mode [26]. Following internal discussions in our team, we decided to rely only on CANopen protocol in our software stack, as introducing an additional protocol for a singular purpose of device flashing would introduce unnecessary system complexity and increase long-term maintenance overhead. Using a bootloader without CANopen support is technically feasible, although leveraging the CANopen protocol makes it possible to utilize the entire ecosys-

tem of the protocol, notably configuration and monitoring tools. [1]
This constraint restricts the use of bootloaders which support only standard CAN communication or other communication protocols.

3.3 Internal bootloader

As mentioned in section 2.2, the MCUs used in the vehicle are manufactured by *STMicroelectronics*. According to the manufacturer, stock internal bootloader in STM32 MCUs does not support firmware updates via the CAN bus in network mode [27, p. 3, note 1], making it unsuitable for this use case. For unknown reasons, it supports only point-to-point firmware updates. Other supported protocol interfaces mentioned in the document, such as USART, I2C, USB DFU, and SPI, can not be used due to the hardware limitations described in the section 3.1 as they are simply not hardware compatible. This thesis has to use interface that works with CAN bus.

3.3.1 Alternative

As one of the potential alternatives, the OpenBLT bootloader was considered. However, as noted in [28], it "does not implement a CANopen protocol stack. Instead, it contains the XCP communication protocol for firmware data exchange." The Universal Measurement and Calibration Protocol (XCP) protocol is incompatible with the CANopen protocol stack. Although the article has mentioned that protocol offers advantages such as reduced overhead and lower memory usage, the objective is to leverage the existing CANopen protocol within the vehicle for the reasons discussed in section 3.2. Therefore, the OpenBLT bootloader was excluded from further consideration.

3.3.2 Commercial products

Commercial alternatives, for example, the *Vector Flash Bootloader* [29] or *MicroControl CANopen Bootloader* [30], were disregarded due to their cost and closed-source nature, which would complicate future modifications.

3.3.3 Conclusion

To the best of my knowledge, no open-source bootloader currently exists for STM32 microcontrollers that supports the CANopen protocol. The lack of open-source implementations of boot-

[1] Example software tool: https://www.vector.com/gb/en/products/products-a-z/software/canoe/option-canopen/#c33313

loaders supporting CANopen protocol necessitates the creation of a custom bootloader that suits the project needs.

4 System design

This chapter details the system design within the imposed limitations outlined in chapter 3.

4.1 The problem

The existing manual firmware flashing method has proved to be inefficient for the iterative development cycles of a Formula Student vehicle. Personal time of embedded developers in our team is valuable. It does not make sense wasting it on manual dismouning of the ECUs from the vehicle just to flash them once and put them back in the vehicle for testing.
There is no available and easily deployable open-source solution at the moment which meets our requirements and adresses the mentioned issue. Brief overview of commercial options that would satisfy at least part of the solution is in section 3.3.2. This thesis implements the solution by considering the current state of the vehicle software and hardware. It replaces the manual way of flashing ECUs, which requires an individual to manually remove the ECU board from the vehicle, connect it to the flashing device, and execute the flashing process.
In contrast, a new simplified workflow should allow updating the selected ECU remotely using only web browser or even command line interface. This approach also simplifies planned future system improvments such as tracking of firmware versions for the currently flashed firmware.

The general process of flashing via CAN bus is already well described, for example, in [31]. However, due to reasons described in chapter 3 it is necessary to create a custom solution that meets our requirements.

4.2 High level view

The custom solution implemented in this thesis should simplify firmware updates for the ECUs in the Formula Student vehicle by enabling remote updates via the CAN bus. The main steps of the process are as follows:

- The user connects to the local WIFI network of the vehicle and opens the web-based interface.

- The user selects the target board and uploads the firmware file to an ECU responsible for the flashing process.
- The targeted ECU is set into bootloader mode using a reset command.
- The firmware file is then transferred to the targeted ECU via the CAN bus.
- The bootloader begins accepting the firmware file from the CAN bus and saves it into internal flash memory.
- After finishing the upload, the targeted ECU verifies the data and starts executing the new firmware.

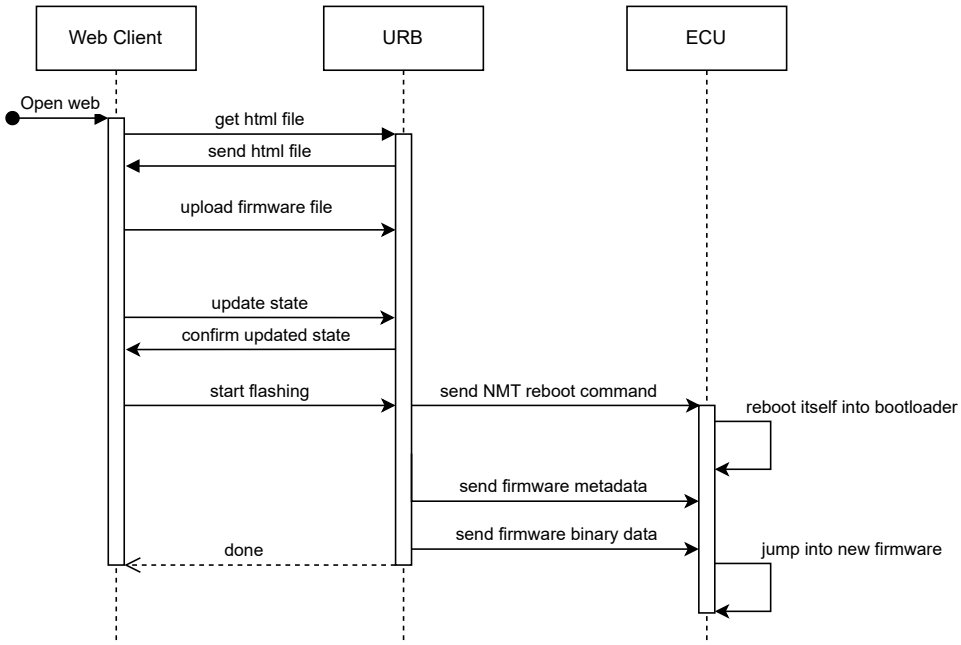


Figure 4.1: Overview of the flashing procedure

4.3 Formula eD4 architecture

As discussed in section 3.1, the vehicle communication system is built around the CAN bus. Pair of them serves as the primary communication backbone of the system. CAN Critical is used for the majority of the messages such as commands for ECUs and critical error messages. CAN General

is used for less important messages, mainly diagnostic data. The system as a whole is designed to work even if CAN General malfunctions.

Figure 4.2 describes a simplified model of the system. Parts beyond the scope of this thesis have been ommited. The electrical team was responsible for the design and development of this system and their decisions have significantly impacted the work presented in the next chapters.

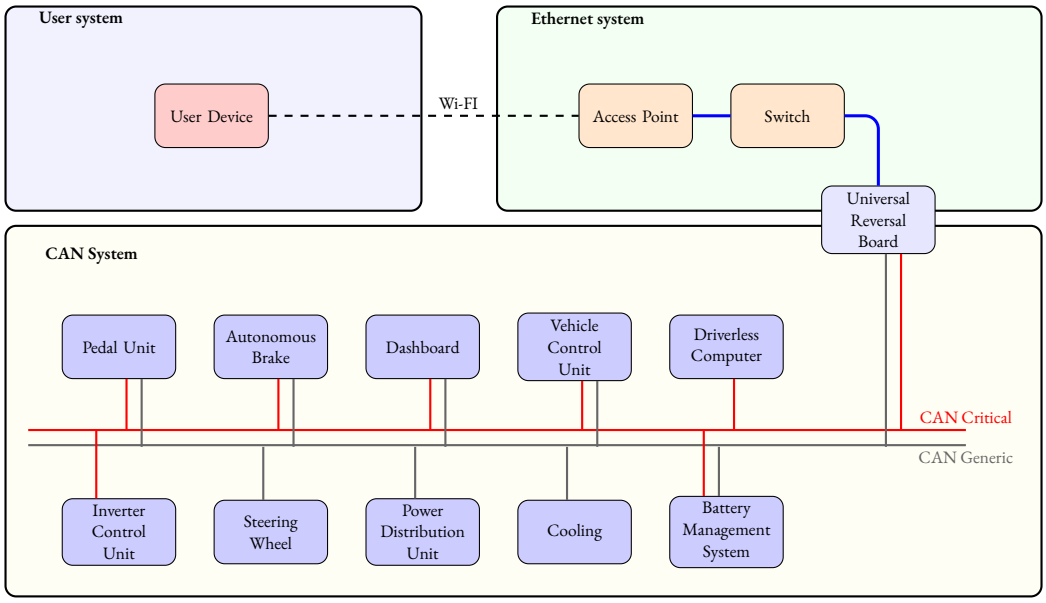


Figure 4.2: Monopost eD4 simplified communication system diagram. Blue rectangle represents a vehicle ECU, orange rectangle is our networking equipment in the vehicle, and red rectangle is the user device used for uploading the firmware binary.

Full name	Abbreviation	STM32 Model	CAN Critical	CAN General
Universal Reversal Board	URB	STM32H723	yes	yes
Pedal Unit	PDB	STM32G473	yes	yes
Vehicle Control Unit	VCU	STM32H745	yes	yes
Battery Managment System	BMS	STM32G473	yes	no
Inverter Control Unit	SCM	STM32G473	yes	no
Steering wheel	Steering	STM32F429	no	yes
Power Distribution Unit	PDU	STM32G473	no	yes
Cooling Unit	Cooling	STM32G473	no	yes
Dashboard Unit	Dashboard	STM32G473	yes	no

Table 4.1: Overview of ECU hardware and abbreviations

4.4 Universal Reversal Board

This ECU, based on an STM32H723 microcontroller, serves a special role in this system. It is the central coordinator responsible for the firmware flashing process. It is also responsible for hosting the embedded HTTP server and facilitation of the interaction with user. As can be seen from fig. 4.2 it is the only ECU equipped with dual CAN interfaces and an ethernet connection, allowing it to communicate with both the vehicle network and the user interface.

I have assigned the Universal Reversal Board (URB) to act as a bridge, managing firmware binary transfers and issuing reboot commands to target ECUs. Its memory layout is split into three parts, the first part is reserved for the URB firmware. Second smaller part for persisting the data of flashing attempts. Third part of the flash is reserved for storage of the currently uploaded firmware binary. This section is designed to store only one firmware binary at a time. Uploading new file will erase this section and save the new file at this address. This design decision determines maximum allowed size of uploaded file, which is calculated as following.

```
end_user_file_sector_addr − start_user_file_sector_addr = size
```

```
0x8100000 − 0x8060000 = 0xA0000
```

If there is need for bigger maximum allowed size, it can be achieved by using an MCU with higher capacity of flash memory as a URB.

It is important to note, that the URB is not designed to support self updating. It is possible to update only other nodes on the network.

4.5 Secondary bootloader design

As mentioned in section 3.3 the original internal bootloader in STM32 microcontrollers does not support firmware updates via the CAN bus in network mode. The goal of secondary bootloader is to recieve data from URB and save it into memory.

The requriments for the custom bootloader are very simple. It has to support features from the higher level communication protocol CANopennode - specifically Service Data Object (SDO) transfer and Network Management protocol (NMT) and the ability to save the recieved data into internal flash memory. It functions as a secondary bootloader (SBL), while the internal bootloader remains in the MCUs for practical considerations [32]. See the following memory map

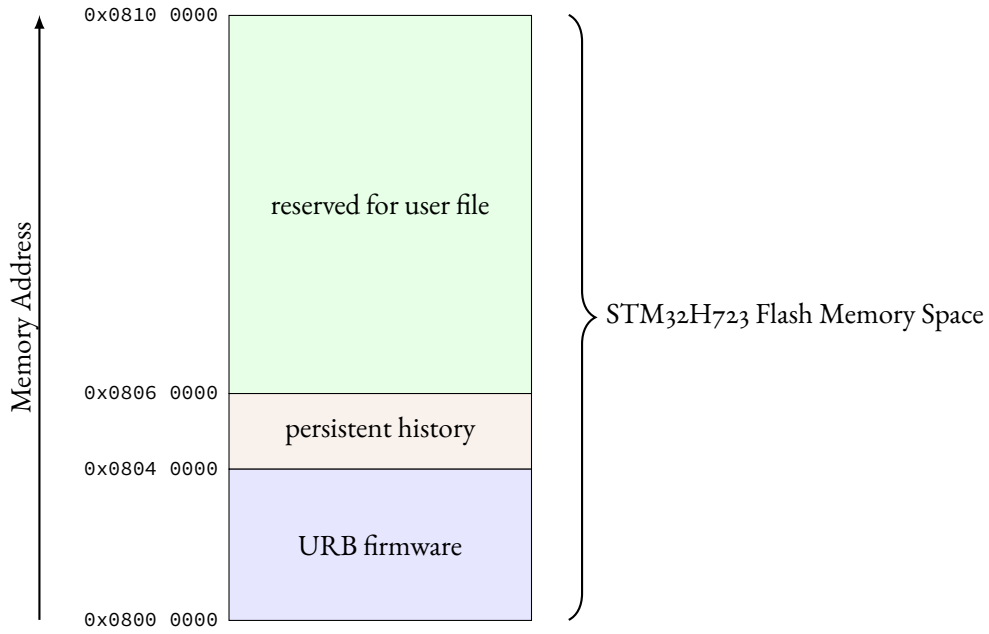


Figure 4.3: URB Flash memory layout

schema fig. 4.5.

The internal bootloader from the vendor is accessed only during manual flashing. This way it is possible to keep the original functionality and only add parts that suit the project. This approach allows for flexibility in the development process and improves the robustness of the system. If something does not work out and the ECU can not be reset into the SBL, there is always the option of removing it from vehicle and performing manual flash which will be handled by the bootloader from manufacturer.

4.6 How to transfer large ammount of data over CAN bus

The Object Dictionary (OD) stores the metadata needed for proper functioning of the CANopen protocol and metadata of the firmware which are necessary for functioning of the remote flashing system.

To transmit data into the OD of bootloader, this work uses the SDO (Service Data Object) protocol as defined by the CANopen specification. SDO is a mechanism for a direct peer to peer communication between two CANopen devices which enables direct access to the OD of an ECU. The specification of SDO also describes multiple variants of the protocol, namely *Expedited transfer*, *Segmented transfer* and *Block Transfer*. This thesis uses primarily *Block Transfer* mode, as this

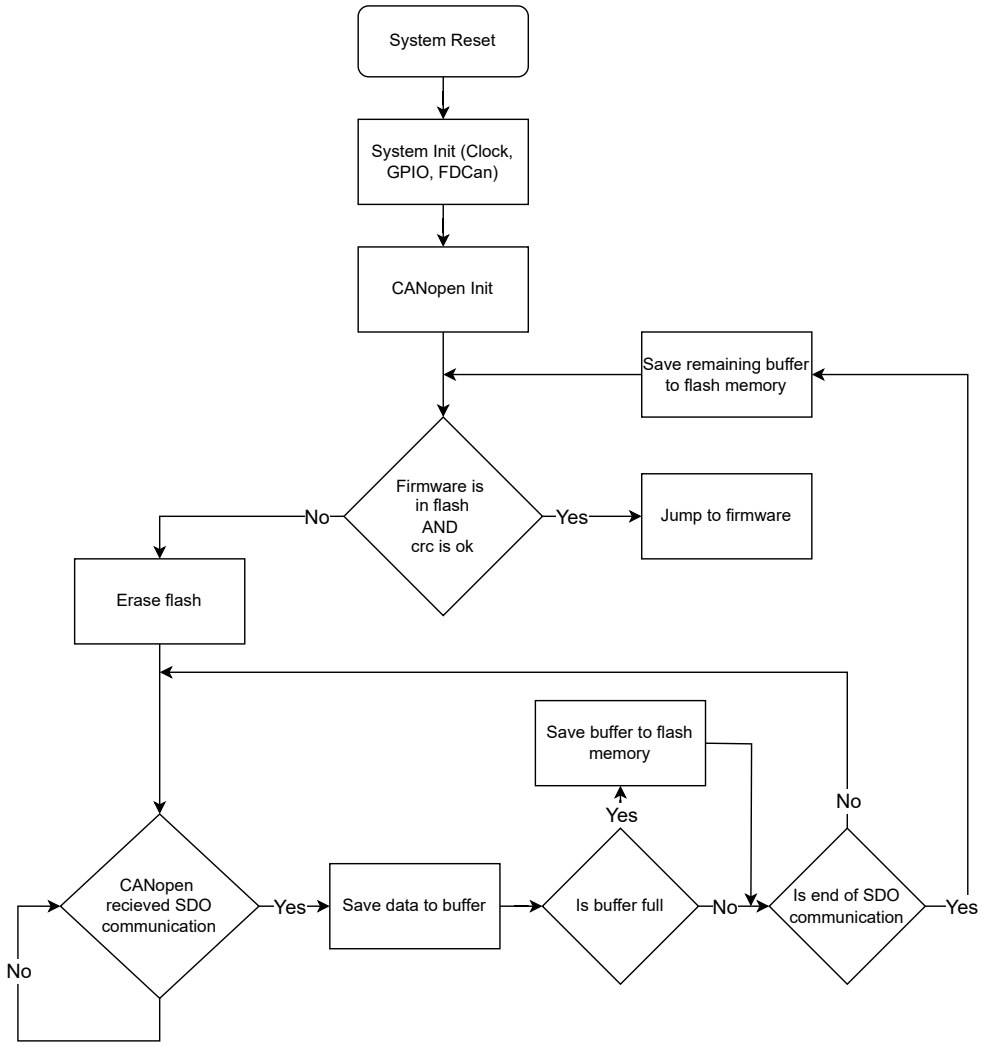


Figure 4.4: Secondary bootloader basic principle diagram

variant is well suited for transfering large amounts of data such as firmware binaries. In the *Block Transfer* mode, data is divided into blocks, with each block containing up to 127 segments. [21, p. 61]

The naming in the SDO protocol is the following. In the CANopennode terminology, the 'client' can access the OD of other nodes that are acting as a server. The default setting of each CANopen node on our network is acting as a 'server' that handles read and write requests to and from the desired OD index. The process of transmitting data from a client (which is in this

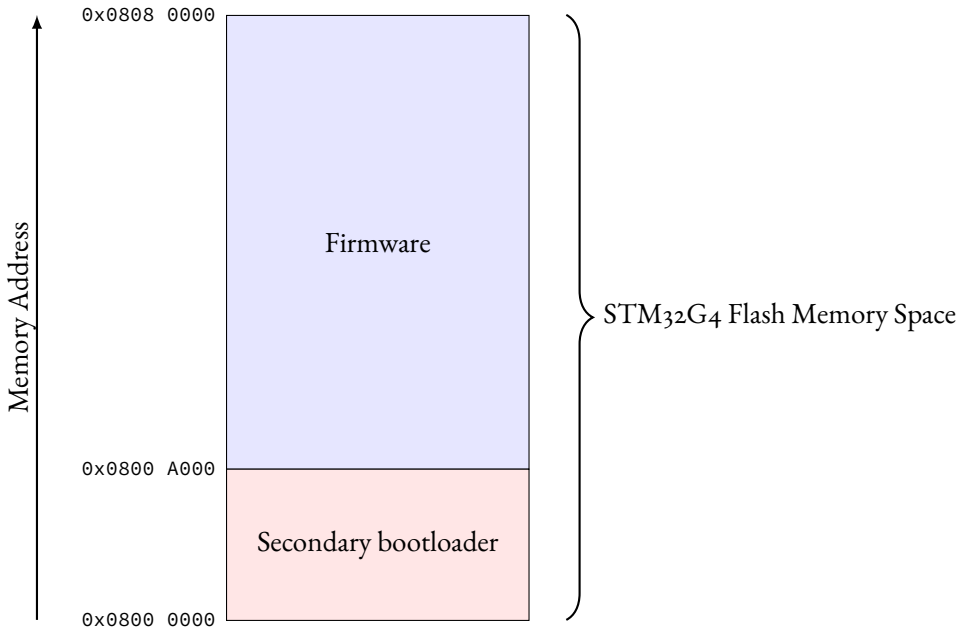


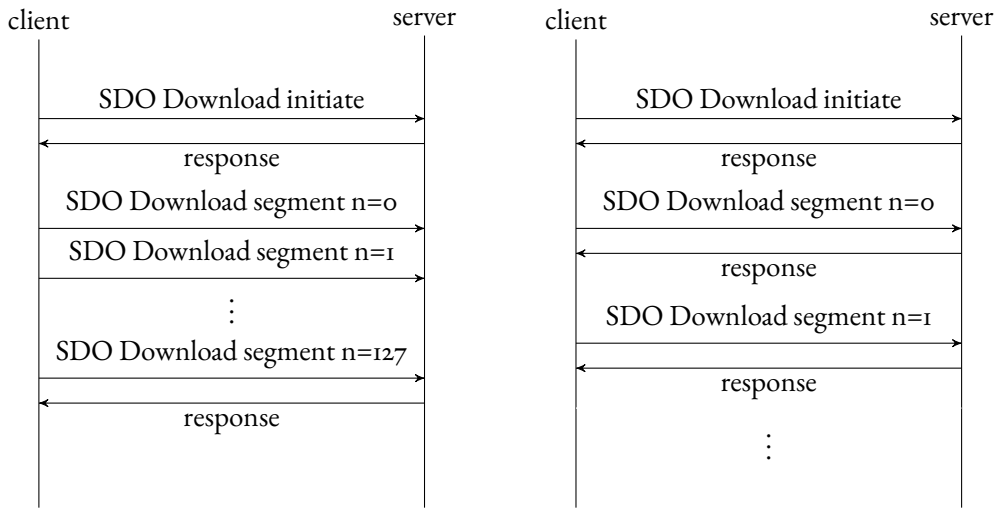
Figure 4.5: SBL Flash memory layout

case URB) to a server (any other ECU) is referred to as downloading a segment [33, pp. 42–43]. The terminology is from the point of OD owner, in our case the ECU with bootloader. While this terminology differs from the typical mental model where the server *serves* the data and client *uploads* the data to the server, the terms align with those used in the specification and will therefore be adopted throughout this thesis.

The client initiates the transfer, and the server acknowledges receipt of an entire block rather than individual segments. If the *Block Transfer* mode had not been used, the server would respond after each CAN frame, which quickly fills the network bandwidth. This transfer mechanism reduces the mentioned communication overhead and subsequently improves data transmission efficiency. The difference between block and non block transfer is displayed in the following fig. 4.6.

4.7 How to get ECU into secondary bootloader

The simple naive way to get into SBL is to perform an MCU reset. This action causes the device to transition through all its stages: first the primary bootloader, then the secondary bootloader, and finally, it may proceed to the application if one exists in the memory. In practice, this reset process is triggered whenever the device needs to be flashed via CAN bus, as the SBL is responsible for handling data reception during this operation.



(a) SDO with *Block Transfer* (b) SDO without *Block Transfer*

Figure 4.6: Comparision of SDO transfer variants

One of the mechanisms used to remotely initiate such a reset is the NMT protocol, which is defined in the CANopen specification [21, p. 83]. It provides remote control of ECUs, including issuing reset commands. This protocol follows a master-slave structure. On the entire CAN network, there should be precisely one NMT manager responsible for executing commands. The URB fulfills this purpose and thus it is the only board that can issue reset commands. In the future this responsibility will be probably be moved to VCU in order to maintain its designated function.

In the system described in this thesis, NMT commands are used to reboot ECUs and thus place them into bootloader mode. The MCU always starts at the bootloader after reset. This behavior follows from the structure of the flash memory and the fact that the MCU begins execution from address `0x08000000`, where the bootloader is always located, see fig. 4.4.

4.8 Safety concerns

Basic safety measures are implemented to prevent unauthorized access. The connection to vehicle internet access point is password protected and the web UI also implements password protection. Further security enhancements, such as encryption and authentication protocols [34], can be implemented in the future if the system is deployed beyond the development stage, although they are not necessary for our purpose at the moment.

5 Implementation

This chapter provides a detailed examination of the implemented remote firmware flashing system for STM32 boards via CAN bus. The goal is to explain critical parts of the technical solution in detail and thoroughly demonstrate specific components of the system.

5.1 Environment

Embedded programming works with real world hardware, thus it often uses specialized tools. The development process relied on several tools to simplify configuration and coding tasks. This section serves as a brief introduction to these tools.

5.1.1 Programming language

The primary programming language chosen for this work was language C, the de facto standard in embedded programming. All libraries used in this work are also written in the C language. Although the Mongoose library also generates Javascript, HTML, and CSS for the web interface, these files were automatically converted into a binary format, requiring no direct interaction with them during development. This is later explained in section 5.6.

5.1.2 Code environment and tools

A STM32CubeMX is a graphical configuration tool provided by the manufacturer of our MCUs, STMicroelectronics. It was used to set up MCU peripherals and generate initialization code. It has simplified the project by offering an intuitive interface for configuring the hardware peripherals.

These tools facilitated the development process:

- Clion: Integrated development environment with support for embedded programming[1].
- STM32CubeMX: A graphical configuration tool used for setting up MCU peripherals and generating initialization code.
- CMake: Managed the build process, streamlining the integration of project components.

[1] https://www.jetbrains.com/help/clion/embedded-overview.html

- Arm GNU Toolchain[2]: Handled the compilation of the code for ARM architecture of STM32 MCUs.
- STLINK-V3SET: Debugging and programming capabilities using manual flashing.
- OpenOCD: Communication with the debugging probe.
- Serial console: Debug UART output from the ECUs at the baudrate 115200.
- PCAN-View and PCAN-USB: Monitored and diagnosed CAN bus communication.
- CANopen Architect Professional: A graphical tool for generating Object Dictionary files compatible with the CANopen standard.
- STM32CubeProgrammer: Inspected device memory, critical for debugging data transfer issues. This was crucial for transferring large files via the CAN bus, as errors concerning incorrect writes into the flash memory were often visible only through this tool.

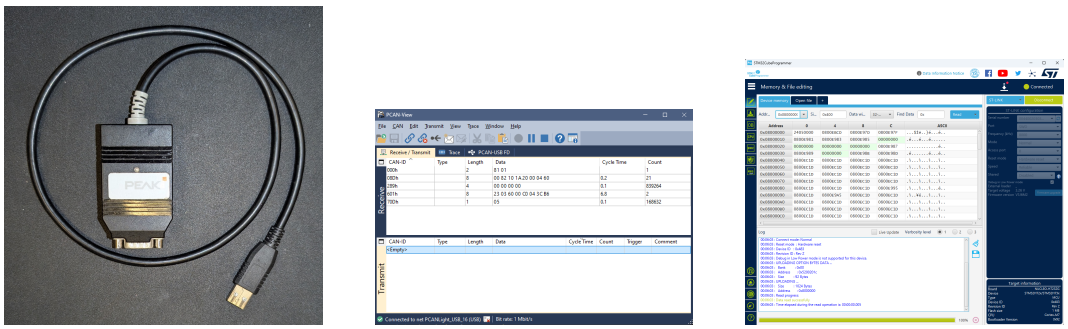


Figure 5.1: Development tools, from left to right - PCAN-USB, PCAN-View, STM32CubeProgrammer

5.1.3 Development setup

The usual development setup shown in the fig. 5.2 included the URB development board and one connected ECUs. The CAN bus connections were terminated with 120Ω resistors at both ends to maintain proper bus signaling. This is required by the CAN specification [21, p. 217]. A STLINK-V3SET probe connected to the flashed ECU provided debugging access and a PCAN-USB adapter was analyzing the CAN network.

[2] https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain

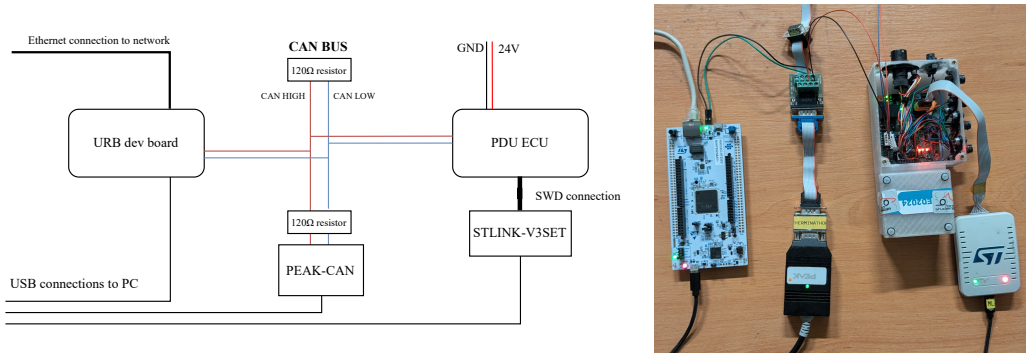


Figure 5.2: Development setup

5.2 Configuring CANopenNode

As discussed in section 3.2, the implementation is based on the `CANopenNode` library. To enable Service Data Object (SDO) communication, the library must be configured appropriately, depending on the node's role in the CANopen network.

5.2.1 SDO Server Configuration

In the bootloader project, the node operates as an SDO server. The relevant configuration is defined in the `CO_driver_config_Slave.h` file. It is necessary to enable both segmented and block transfer modes, as well as to allocate sufficient buffer space for incoming data.

```
#define CO_CONFIG_SDO_SRV (CO_CONFIG_SDO_SRV_SEGMENTED | \
                           CO_CONFIG_GLOBAL_FLAG_CALLBACK_PRE | \
                           CO_CONFIG_SDO_SRV_BLOCK)
#define CO_CONFIG_SDO_SRV_BUFFER_SIZE 900
```

5.2.2 SDO Client Configuration

In the URB project, the node functions as an SDO client. The configuration must reflect this role by enabling the appropriate client mode. In addition to segmented and block transfer modes, FIFO and CRC functionalities must also be activated:

```
#define CO_CONFIG_SDO_CLI (CO_CONFIG_SDO_CLI_ENABLE | \
                           CO_CONFIG_SDO_CLI_SEGMENTED | \
                           CO_CONFIG_SDO_CLI_BLOCK)
#define CO_CONFIG_FIFO (CO_CONFIG_FIFO_ENABLE | \
                        CO_CONFIG_FIFO_ALT_READ | \
```

```
                        CO_CONFIG_FIFO_CRC16_CCITT)
#define CO_CONFIG_CRC16 (CO_CONFIG_CRC16_ENABLE)
```

5.2.3 NMT Configuration

A minimal Network Management (NMT) implementation is required in both the bootloader and URB projects. The following configuration macro enables basic NMT master functionality.

```
#define CO_CONFIG_NMT (CO_CONFIG_NMT_CALLBACK_CHANGE | \
                       CO_CONFIG_NMT_MASTER | \
                       CO_CONFIG_GLOBAL_FLAG_CALLBACK_PRE)
```

5.3 Jump from secondary bootloader into application binary

After successfully recieving and saving the firmware, the bootloader must transfer control to the newly flashed firmware.

The function `jump_to_firmware()` is responsible for this crucial part of the bootloader

First, all interrupts are disabled in order to prevent any unexpected behavior during the transition. Since no further interrupt handling is expected at this point, this does not pose any risk. The system is now entirely dedicated to setting up execution of the new firmware. Next, all initialized peripherals are deinitialized, and the system clock configuration is restored to its default state. The configuration of the system is exactly the same as immediately after the ECU is powered on. This ensures that the new firmware starts in a well-defined environment without residual configuration from the bootloader which could otherwise persist in the peripherals. As the last step in the peripheral deinitialization process, the *Reset and Clock Control* peripheral is reset. This peripheral is responsible for managing clock sources and distribution across the microcontroller and must be returned to its default state to avoid conflicts with the initialization sequence of new firmware.

Following this, all interrupts are manually reset. This is achieved by writing the value `0xFFFFFFFF` to both the *Interrupt Clear Enable Register* (ICER) and the *Interrupt Clear Pending Register* (ICPR). According to the reference manual [35, pp. 211, 213], this operation guarantees that all interrupt enables and pending flags are cleared. Immediately before the control is passed to the application, the correct *Main Stack Pointer* value is set. This is followed by a direct jump to the firmware reset handler, which is usually located at the base address of the firmware offset by length of stack pointer. There are the positions that the bootloader is expecting, the actual locations can be changed in the linker file of the flashed project if needed. This low-level jump finalizes the transition and begins execution of the main firmware.

After disassembling the crucial part which performs the jump into application code, it shows only two simple ARM instructions. the first line `ldr r3, [r3, #4]` loads into register `r3` a 32 bit value from register `r3` offset by the length of stack pointer which is 4 bytes from the start. The `struct boot_vectable_` was already loaded into the register `r3` and the offset ensures loading of the start application address. The second line `blx r3` branches to the address in register `r3` and changes instruction set according to the ARM documentation [36].

```
127     // Jump to app code                    243     ldr r3, [r3, #4]
128     BOOTVTAB->app();                       244     blx r3
```

Figure 5.3: Disassembled jump into application address

This completes the procedure. The new executable starts from the `BOOT_ADDR` defined in the `canopen_bootloader.h`.

This procedure maintains compatibility across different STM32 microcontroller architectures, including Cortex-M4 (stm32g4...) and Cortex-M7 (stm32h7...).

The following source code is taken from [37].

```
1  // line 95 in bootloader\Core\canopen_bootloader.c
2  _Noreturn void jump_to_firmware(void)
3  {
4      __disable_irq();                        /* Disable all interrupts */
5      HAL_DeInit();                           /* Deinitialize all HALs */
6
7      SysTick->CTRL = 0;                      /* Disable Systick timer */
8      SysTick->LOAD = 0;
9      SysTick->VAL = 0;
10     HAL_RCC_DeInit();                       /* Set the clock to the default state */
11     /* Reset all interrupts */
12     for (int i = 0; i < sizeof(NVIC->ICER) / sizeof(NVIC->ICER[0]); i++){
13         NVIC->ICER[i] = 0xFFFFFFFF;
14         NVIC->ICPR[i] = 0xFFFFFFFF;
15     }
16     __enable_irq();                         /* Re-enable all interrupts */
17     __set_MSP(BOOTVTAB->initial_SP);        /* Set the Main Stack Pointer */
18     BOOTVTAB->firmware();                   /* Jump to firmware code */
19
20     while (true) { __NOP(); }
21 }
```

5.4 Selective board targeting

A key aspect of the system and one of the goals of this work is the ability to selectively target individual ECUs for firmware updates. Each ECU in our network has assigned a unique node ID within the CANopen protocol. The SBL of each ECU uses the same node ID, which is set with the `CAN_NODE_ID` definition while compiling bootloader. The selective targeting mechanism allows specific nodes to be addressed without interrupting the functionality of other devices on the network.

The targeting process involves leveraging the SDO protocol provided by the CANopen standard. By specifying the node ID in the communication command, the system can direct firmware updates exclusively to the desired ECU.

User is able to select the desired ECU in the web UI. See fig. 5.4 This selection is at the Firmware Update screen in the form of dropdown menu with ECU names. Most of the boards are connected to CAN Critical bus, however few of them are connected only to CAN General. This information is included in the `s_boards` array and the URB uses the information to select correct CAN bus for sending the data.

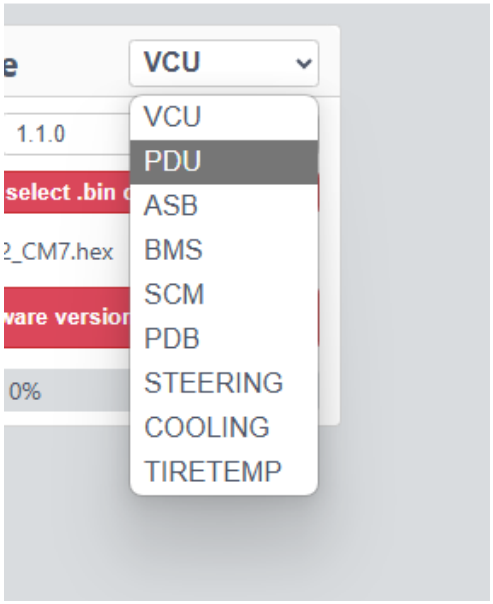


Figure 5.4: Selection of board to flash in UI

5.5 User interface

During the design phase, I considered multiple approaches for implementing the user interface (UI). Initially, two promising solutions were explored. //

First option was UDP/TCP Server with a custom transfer protocol. This approach involved creating a UDP or TCP server on the URB, which would use a custom protocol to communicate with an external application running on the user device. While functional, this solution had several disadvantages:

- Development of a new protocol adds complexity.
- Dependency on an external application, which could become outdated on the user machine.
- Creating functional TCP or UDP server in embedded environment was not trivial.

Second option experimented with HTTP server using Lightweight IP (LwIP) Library. This option used the already existing HTTP daemon in the LwIP stack, although it was still a lot of work to implement all necessary functions. Both approaches have reached a prototype stage but neither was sufficiently robust for practical use.

The final user interface was implemented as a web-based application, chosen for its accessibility and simplicity. This solution embeds a lightweight HTTP server on the URB using the Mongoose library. This library is designed to developing web interfaces on embedded devices. While Mongoose can operate atop an existing TCP/IP stack like LwIP (which is supported by STM32 MCU configuration tools), I opted to use their built-in TCP/IP stack due to the availability of an example tailored for the reference board used in this project. Building on this example was a lot easier than creating blank project and implementing the LwIP support.

The web interface enables users to upload firmware binaries, initiate firmware updates and monitor the progress of the flashing process. Morover, there is a dashboard which makes it easy to see current state of uploaded firmware in the vehicle and also list of flashing history with metadata such as Board name, Time of flash, Version, Filename, Author and wether the flash was successful.

5.6 Mongoose library

The file ‘mongoose_glue.c‘ contained unfinished functions called by the Mongoose library, which needed integration into the existing system. These functions act as the API for interacting with the library. Embedded devices generally lack native filesystem support, which would otherwise be necessary for serving web pages via HTTP. However, web pages require HTML, CSS, and

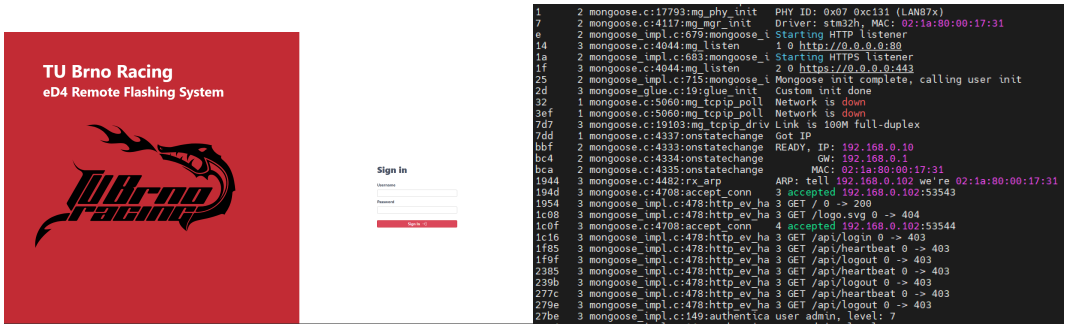


Figure 5.5: Login page and log from the URB

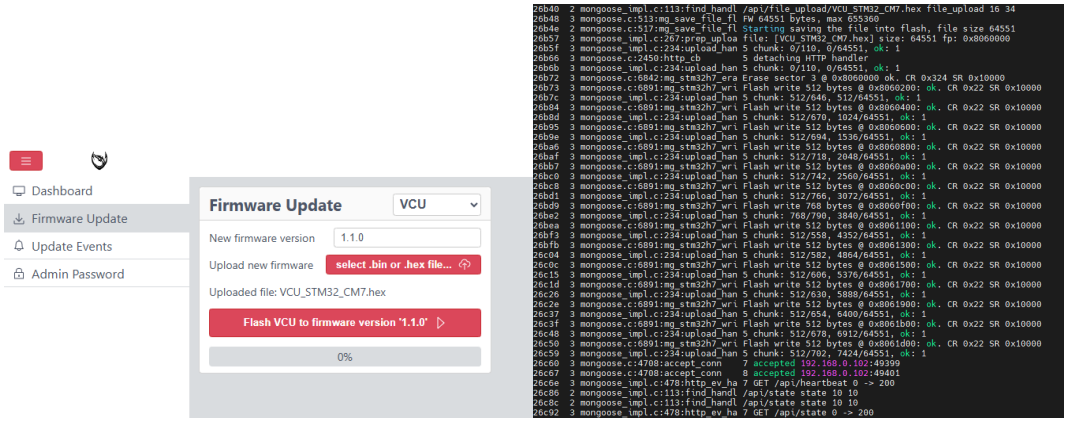


Figure 5.6: UI and log from the saving process

JavaScript files to be accessible. Mongoose addresses this limitation with a clever solution. It provides a utility that converts the required files into a binary "pack", which can be embedded directly into the C code. This binary representation is generated in the file ‘mongoose_fs.c‘ and stored in a variable named ‘v1‘. During runtime, Mongoose uses its file system virtualization layer to extract the requested file contents from this binary representation. The details of this feature are described in the Mongoose documentation [38]. To achieve seamless integration with command-line tools, HTTP appears to be the most universal solution. It makes it possible to interact with the URB by sending simple POST and GET requests.

Using the configuration wizard provided by Mongoose[3], I generated a basic UI that runs on the URB. Below is a table describing the generated files and their roles in the project.

File name	
mongoose_config.h	main configuration file of the mongoose library
mongoose.c/.h	amalgamation of the core mongoose source files required for the library
mongoose_glue.c/h	generated API which was integrated with codebase
mongoose_fs.c	generated HTML, Javascript and CSS files packed into binary format
mongoose_impl.c	generated and modified implementation of defined web API functions

Table 5.1: Overview of mongoose files in the URB project

5.6.1 License

Mongoose is available under a dual-license model: GPLv2 for open-source use and a commercial license for proprietary projects, allowing flexibility based on project requirements. The source code for this thesis is open source and complies with the Mongoose GPLv2 license terms.

5.7 Ensuring data integrity

CRC checksums are used to ensure reliable data transmission on all parts of the transfer. If mismatched, the transmission halts, and an error message is logged in the console. Mongoose implements ethernet-based transmission safeguards by verifying each packet with checksums as can be seen in the file `mongoose.c`. Prior to transferring a file over CAN from URB, a checksum is calculated using the `mg_crc32()` function from mongoose.c file. This checksum is sent via an SDO transfer to the variable in the OD at index `0x6000` on the target device. After completing the SDO transfer and storing the data in flash memory, the target device calculates a checksum directly from the stored data using `mg_crc32()` and compares it to the expected value. In case of a mis-

[3] https://mongoose.ws/wizard/

match, firmware execution is aborted, the uploaded data is erased from memory, and the device remains in bootloader mode.

6 Conclusion

This thesis has successfully designed and implemented a system for remote firmware flashing of STM32-based embedded boards within a Formula Student vehicle, leveraging the established CAN bus infrastructure and the CANopen protocol. Over the air (OTA) firmware updates work via a user-friendly web interface hosted on a central Universal Reversal Board (URB). This allowed developers to update specific control units (ECUs) without physical access which already significantly accelerated the development and testing phases.

Bibliography

[1] *Disciplines*. https://www.formulastudent.de/about/disciplines/. 2024. (Visited on 12/06/2024).

[2] *FSG: Competitions*. https://www.formulastudent.de/world/competitions/. 2024. (Visited on 12/04/2024).

[3] *TU Brno Racing Achievements*. URL: https://tubrnoracing.cz/en/about-us/achievements/ (visited on 12/12/2024).

[4] Sebastian Šimanský. "Enhancement of Vehicle Dynamics Through Adaptive Torque Vectoring Control with PMSM Powertrain". In: *Proceedings II of the 30st Conference STUDENT EEICT 2024: Selected papers* (2024), pp. 48–52. DOI: 10.13164/eeict.2024.48. URL: https://doi.org/10.13164/eeict.2024.48.

[5] M. Barr and A. Massa. *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Series. O'Reilly Media, 2006. ISBN: 9780596009830. URL: https://books.google.cz/books?id=NzqbAgAAQBAJ.

[6] Christoph Hammerschmidt. *Number of automotive ECUs continues to rise*. https://www.eenewseurope.com/en/number-of-automotive-ecus-continues-to-rise/. 2019. (Visited on 05/03/2025).

[7] Ian Smalley Josh Schneider. *What is a microcontroller?* https://www.ibm.com/think/topics/microcontroller. Accessed: 2025-05-22. 2024.

[8] Ralf Schmidgall. "Automotive embedded systems software reprogramming". PhD thesis. Brunel University School of Engineering and Design PhD Theses, 2012.

[9] Rimac Technology. *Next-gen ECU: Pioneering the Future of Electrical Architecture*. https://www.rimac-technology.com/case-study/pioneering-the-future-of-electrical-architecture. 2025. (Visited on 05/20/2025).

[10] Shantipal Ohol and Samruddhi Kaware. "Semi-Autonomous Parking system for Automatic Transmission vehicles". In: *IOP Conference Series: Materials Science and Engineering* 1012 (Jan. 2021), p. 012051. DOI: 10.1088/1757-899X/1012/1/012051.

[11] "ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes". In: *IEEE STD 12207-2008* (2008), pp. 1–138. DOI: 10.1109/IEEESTD.2008.4475826.

[12] Jonathan T Moore, Michael Hicks, and Scott Nettles. *General-purpose persistence using flash memory*. Tech. rep. Citeseer, 1997.

[13] ST Microelectronics. *Description of STM32F4 HAL and low-layer drivers*. 2023. URL: https://www.st.com/resource/en/user_manual/um1725-description-of-stm32f4-hal-and-lowlayer-drivers-stmicroelectronics.pdf (visited on 12/01/2024).

[14] Tech-FAQ. *Flashing Firmware*. https://www.tech-faq.com/flashing-firmware.html. Accessed: 2025-05-22. n.d.

[15] Luigi Catuogno and Clemente Galdi. "Secure Firmware Update: Challenges and Solutions". In: *Cryptography* 7.2 (2023). ISSN: 2410-387X. DOI: 10.3390/cryptography7020030. URL: https://www.mdpi.com/2410-387X/7/2/30.

[16] Wei Dong. "Research on Online Upgrade of DSP Application Program Based on Serial Communication". In: *2023 China Automation Congress (CAC)* (Nov. 2023), pp. 1435–1440. DOI: 10.1109/cac59555.2023.10451910. URL: https://doi.org/10.1109/cac59555.2023.10451910.

[17] Wael Badawy et al. "On Flashing Over The Air "FOTA" for IoT Appliances – An ATMEL Prototype". In: *2020 IEEE 10th International Conference on Consumer Electronics (ICCE-Berlin)*. 2020, pp. 1–5. DOI: 10.1109/ICCE-Berlin50680.2020.9352203.

[18] STM32 base contributors. *Guide: Flashing*. https://stm32-base.org/guides/flashing.html. 2022. (Visited on 05/11/2025).

[19] ST Microelectronics. *STM32G4 series advanced Arm®-based 32-bit MCUs*. 2025. URL: https://www.st.com/resource/en/reference_manual/dm00355726-stm32g4-series-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf (visited on 04/16/2025).

[20] PubNub Team. *What is a Communication Protocol?* https://www.pubnub.com/learn/glossary/communication-protocols/. 2025. (Visited on 05/11/2025).

[21] Olaf Pfeiffer, Andrew Ayre, and Christian Keydel. *Embedded Networking with CAN and CANopen*. 1st. Greenfield, MA, USA: Copperhill Media Corporation, 2008. ISBN: 0976511622.

[22] Nicolas Navet et al. "Trends in automotive communication systems". In: *Proceedings of the IEEE* 93.6 (2005), pp. 1204–1223.

[23] B. Montanari. "Why do we need a bootloader in a microcontroller?" In: (2022). URL: \url{https://community.st.com/t5/stm32-mcus/why-do-we-need-a-bootloader-in-a-microcontroller/ta-p/49446} (visited on 05/21/2025).

[24] ME-Meßsysteme GmbH. *Basics of the CAN Bus*. https://www.me-systeme.de/docs/grundlagen/canbus/kb-canbus-en.pdf. 2016. (Visited on 11/01/2024).

[25] CANopenNode. *CANopenNode*. https://github.com/CANopenNode/CANopenNode. 2024.

[26] Vector Informatik GmbH. *CANopen - Higher-Layer CAN Protocol*. https://www.vector.com/gb/en/know-how/protocols/canopen/#. (n.d.) (Visited on 05/20/2025).

[27] STMicroelectronics. *How to Use FDCAN Bootloader Protocol on STM32 MCUs*. https://www.st.com/resource/en/application_note/an5405-how-to-use-fdcan-bootloader-protocol-on-stm32-mcus-stmicroelectronics.pdf. 2024. (Visited on 10/28/2024).

[28] Feaser. *CANopen bootloader based on OpenBLT*. https://www.feaser.com/en/blog/2020/06/canopen-bootloader-based-on-openblt/. 2020. (Visited on 11/08/2024).

[29] Vector Informatik GmbH. *Flash Bootloader*. https://www.vector.com/in/en/products/products-a-z/embedded-software/flash-bootloader/. (n.d.) (Visited on 11/04/2024).

[30] MicroControl. *CANopen bootloader protocol stack*. https://www.microcontrol.net/en/portfolio/protocol-stacks/canopen/canopen-bootloader/. (Visited on 11/04/2024).

[31] Renesas. *Flash Over CAN*. https://www.renesas.cn/zh/document/apn/rx600-series-flash-over-can-rev121. 2012. (Visited on 12/08/2024).

[32] PiEmbSysTech. *Secondary Bootloader (SBL) in Two-Stage Bootloader*. https://piembsystech.com/secondary-bootloader-sbl-in-two-stage-bootloader/. 2017. (Visited on 11/02/2024).

[33] CiA CiA. "301 V4. 2.0–CANopen application layer and communication profile". In: *CAN in Automation eV* (2011).

[34] Dennis K. Nilsson, Ulf E. Larson, and Erland Jonsson. "Creating a Secure Infrastructure for Wireless Diagnostics and Software Updates in Vehicles". In: *Computer Safety, Reliability, and Security*. Ed. by Michael D. Harrison and Mark-Alexander Sujan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 207–220. ISBN: 978-3-540-87698-4.

[35] ST Microelectronics. *STM32 Cortex®-M4 MCUs and MPUs programming manual*. 2020. URL: https://www.st.com/resource/en/programming_manual/pm0214-stm32-cortexm4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf (visited on 04/16/2025).

[36] ARM Limited. *BLX*. https://developer.arm.com/documentation/dui0489/i/arm-and-thumb-instructions/blx?lang=en. 2024. (Visited on 11/18/2024).

[37] gbm. *How to jump to system bootloader from application code on STM32 microcontrollers*. 2023. URL: \url{https://community.st.com/t5/stm32-mcus/how-to-jump-to-system-bootloader-from-application-code-on-stm32/tac-p/568644/highlight/true\#M475} (visited on 05/21/2025).

[38] Cesanta. *Embedded Filesystem*. https://mongoose.ws/documentation/tutorials/core/embedded-filesystem/. 2024. (Visited on 11/18/2024).

Appendix

All source code created as a result of this thesis is available in the attached ZIP archive. The archive contains the following directories:

- urb - firmware responsible for the flashing process and the HTTP server with UI
- bootloader - secondary bootloader with CANOpen protocol support

The source code is also available on GitHub:
`https://github.com/adamvalt/remote-firmware-flashing-for-STM32-boards-via-CAN-bus`