



# Development of a Bootloader for the STM32F3 Platform

---

Gardier Simon

Personal project for PROJ0011 as part of the curriculum in:  
**Master of Science in Computer Science**

Academic year: **2024 - 2025**

# Acknowledgements

I would first like to thank Pr. Boigelot for sharing his incredible knowledge and passion for embedded systems and robotics, as well as for the many anecdotes about past projects! His help has greatly influenced the results and reach of the project, pushing me to go deeper into the details of the architecture of the Cortex M4.

I am grateful to the students of the RoboCup team at Montefiore, who have always been there when needed and have always been enthusiastic about new challenges!

Most importantly, I would like to thank Mai-Phi, my love, for her continuous support.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The competition . . . . .	1
1.2	The RoboCup Montefiore team . . . . .	3
1.3	The robot of the Robocup Montefiore team - Bobby . . . . .	4
<b>2</b>	<b>Objective</b>	<b>5</b>
<b>3</b>	<b>Introduction to the Arm® Cortex®-M4 and STM32F303</b>	<b>7</b>
3.1	Introduction . . . . .	7
3.2	Processor vs Microcontroller . . . . .	7
3.3	Architecture . . . . .	9
3.3.1	Instruction Set Architecture (ISA) . . . . .	9
3.3.2	Exceptions . . . . .	9
3.3.3	Vector table . . . . .	10
3.3.4	Operation modes . . . . .	10
3.3.5	Clock tree . . . . .	10
3.4	Memory . . . . .	11
3.4.1	Memory map . . . . .	11
3.4.2	Flash memory . . . . .	12
3.4.3	Cyclic Redundancy Check (CRC) . . . . .	12
<b>4</b>	<b>ARM executables</b>	<b>13</b>
4.1	Introduction to Executable and Linkable Format (ELF) . . . . .	13
4.2	Inspection of the executable “blinky” . . . . .	14
<b>5</b>	<b>Bootloader</b>	<b>16</b>
5.1	Bootloader - Version 0 . . . . .	16
5.1.1	Introduction . . . . .	16
5.1.2	Illustrations . . . . .	16
5.1.3	Start bootloader . . . . .	16
5.1.4	Main bootloader . . . . .	16
5.1.5	Recovery bootloader . . . . .	17
5.2	Bootloader - Version 1 . . . . .	18
5.2.1	Introduction . . . . .	18
5.2.2	Illustrations . . . . .	18
5.2.3	Stage 1 bootloader . . . . .	18
5.2.4	Stage 2 bootloader . . . . .	19
5.2.5	Networking . . . . .	19
5.2.6	Test bench . . . . .	19
5.2.7	Implementation of the jump mechanism . . . . .	19

## TABLE OF CONTENTS

---

5.2.8	Usage . . . . .	20
5.2.9	Limitations . . . . .	20
<b>6</b>	<b>Conclusions</b>	<b>21</b>
<b>7</b>	<b>Future work</b>	<b>22</b>
	<b>Appendices</b>	<b>23</b>
7.1	Robocup-info Git Repository . . . . .	23
7.2	Access to resources . . . . .	24
7.3	Installation guide . . . . .	24
7.4	Hands-on with libopencm3 . . . . .	24
7.5	Guide to GDB for debugging on the STM32 platform . . . . .	25
7.5.1	Introduction . . . . .	25
7.5.2	Getting started . . . . .	25
7.5.3	Start a debugging session . . . . .	25
7.5.4	Debugging . . . . .	26
7.5.5	More readings . . . . .	26
7.6	Hands-on with FreeRTOS . . . . .	27
7.6.1	Introduction . . . . .	27
7.6.2	Description . . . . .	27
7.7	Jump mechanism implementation with libopencm3 . . . . .	28
7.8	Bootloader illustrations (version 0) . . . . .	29
7.9	Bootloader illustrations (version 1) . . . . .	33
	<b>Bibliography</b>	<b>39</b>

# Chapter 1

## Introduction

### 1.1 The competition

RoboCup [1] is an international robotics competition created in 1996. The initiative was created to promote robotics and AI. Originally, the goal was to beat the best team of football players with a team of fully autonomous robots by the middle of the 21st century. For this purpose, the *RobocupSoccer* category has been created. It is the most popular category of Robocup.

However, with the competition reaching new audiences, many more categories were created:

- **RoboCupRescue** aims at advancing the research in rescue solutions using robots.
- **RoboCup@Home** aims at advancing the research in home solutions for people with disabilities or the elderly.
- **RoboCupIndustrial** is the newest category, which targets work-related robotics in industrial scenarios.
- **RoboCupJunior** aims at promoting robotics to the younger generation with simplified robotics challenges.

The RobocupCupSoccer category is composed of the following leagues:

- **Standard Platform League:** a league composed exclusively of Nao robots [2]. This league focuses only on the programming of the robots, as the hardware is provided by the Nao robots. The other leagues require the teams to develop their hardware platform.
- **Humanoid League:** a league of autonomous robots with human-like bodies and human-like sensors, “non-human” sensors (e.g. cameras behind the head, lidar,...) are forbidden. The league is composed of 3 sub-leagues: KidSize, TeenSize, and AdultSize. In the KidSize league, the games are played by two teams of 4 robots, one of whom must be designated as the goalkeeper.
- **Middle League:** a league of fully autonomous robots on wheels playing with a standard FIFA soccer ball. The robots compete in teams of 6.
- **Small League:** a league of smaller robots competing in teams of 11.
- **Simulation League:** The oldest league of RoboCup, which is focused on AI and team strategy. It is composed of 2 sub-leagues: Simulation 2D and Simulation 3D.

Finally, the RoboCup organisation hosts a research conference [3] in which researchers from around the world can submit their work about Robotics. The conference is taking place at the same time and place as the competition, which is hosted on a different continent each year.

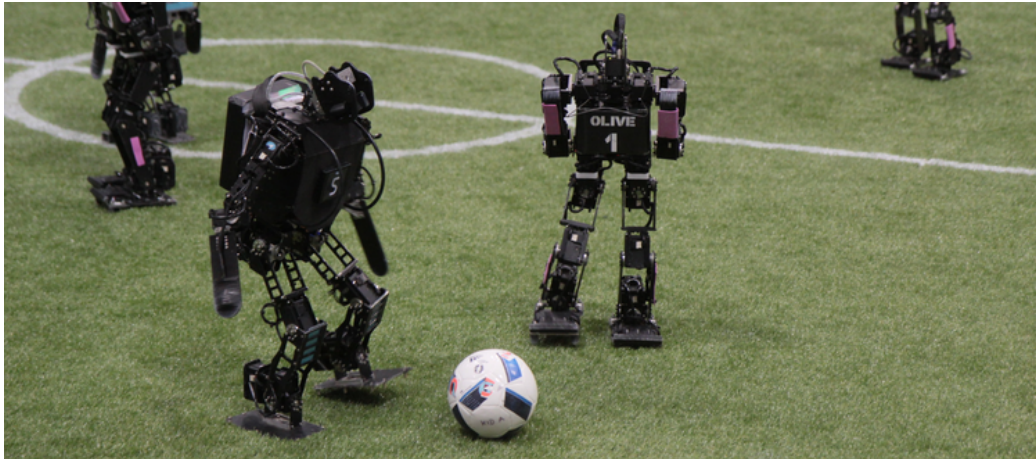


Figure 1.1: KidSize robots from other teams in action! (Source: [Robocup](https://www.robocup.org/))

## 1.2 The RoboCup Montefiore team

After several successes in the Eurobot competition, the Montefiore team, led by Pr. Boigelot, decided to tackle new challenges in the **RobocupSoccer KidSize** League.

The project is composed of three teams:

- Electronic team: responsible for the design and production of the electrical and electronic parts of the robots (battery pack, multifunction board, embedded board for servomotors,...).
- Mechanic team: responsible for the design of the robot body.
- Computer science team: responsible for the design and development of the robot software (control, planning, networking, vision,...)



Figure 1.2: The team's robot, Bobby, uncapping a beer!

Throughout the years, the following achievements have been accomplished: Vision-based 2D pose estimation (2016) [4], Development of a embedded controller for Dynamixel MX-28 servomotors (2016) [5], Development of a simulation environment for the robot platform using V-Rep (2016) [6], Stereoscopy to estimate distance to the soccer ball (2017) [7], Locomotion control system (2017) [8], Building of a simulation environment using Blender (2017) [9], Integration of Libopencm3 in FreeRTOS for the actuators (2019) [10], V1 Design of the robot body (2025) [11], Development of a C library for communication over CAN bus (2024) [12], Development of a Software architecture for the motherboard of the robot (2024) [13], Development of a embedded multifunction board (2025) [14], Animation of the robot body with Blender and FreeRTOS (2025) [15].

The next problems the team will tackle are the design of a second version of the embedded controller for the servomotors, the design of a second version of the robot body, the development of a software suite for bipedal walking in the real world and the development of a software suite for self-localisation in the real world.

### 1.3 The robot of the Robocup Montefiore team - Bobby

The robot platform, called Bobby, is composed of the following:

- A rigid body made of tough PLA.
- A LattePanda 3 Delta [16] main board running Fedora 39 with a Preempt-RT kernel and acting as the Central Unit of the robot.
- A Homemade Multifunction board [17]. This card is connected to the main board via USB. The role of the multifunction board is to provide a link between the motherboard and the robot's peripherals (mainly the servomotors), manage the power supply of the entire robot, and provide an emergency stop button.
- 22 Dyamixel MX28 servomotors [18] present in each articulation of the robot. These servomotors are connected to the multifunction board through a CAN bus (A detailed report on the CAN communication in the robot can be found in [12]). The board of the servomotors will be replaced by a custom board [5] with a more powerful MCU and more sensors.
- Two [USB 3.0 cameras](#) connected to the main board via USB [7].

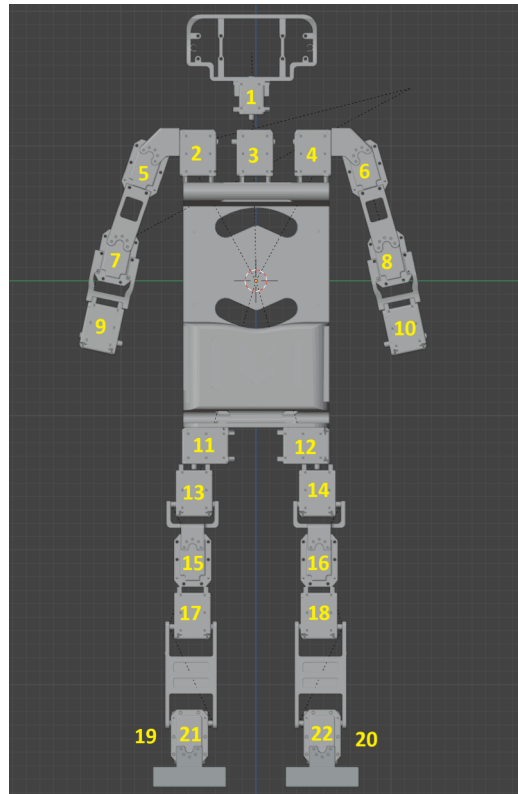


Figure 1.3: 3D model of Bobby, with the 22 servomotors, acting as articulation at each joint. (Author: Damiano Pantalone)



## Chapter 2

# Objective

*“A bootloader, or boot loader, is a program that runs before the main program, called the application, and is responsible for locating and starting the application. Its capabilities may vary, but it often provides facilities for the programmer to inspect the MCU memory and receive updates over a communication bus for the application code and/or for itself.”*

— Simon Gardier

The Microcontroller unit (MCU) embedded in each servomotor already provides firmware with boot-loading and application update facilities. However, this firmware does not support updates over CAN bus, which is the only means for the servomotors in our architecture.

Furthermore, the firmware present in the servomotor MCU requires the use of the MCU debugging connector to update the application present on it. The solution is not viable for fast development iterations, as the debugging connector is only accessible when the protection casing of the servomotors is open. With this firmware, updating the 22 servos would imply removing the 22 MCUs from their casing and soldering connections to the debugging connector, **each time**.

Finally, the firmware is closed-source, making it impossible to modify it to our needs. Thus, the need for a custom bootloader became apparent.

The bootloader must provide the following features:

- Receive updates from a master node (Multifunction card) over a communication bus (CAN bus) and update the application code (Compiled program with FreeRTOS).
- Receive updates from a master node over a communication bus and update itself.
- Start the application code.

These features must be developed with absolute robustness in mind. Indeed, after flashing the bootloader the first time, the MCU will be encased in the protection casing of the servomotors. If the bootloader stops responding afterwards, the user would have to re-open **every** servomotor to flash a new bootloader.

In a complete setup, an update for the application code of a servomotor would be as follows:

1. The programmer sends his program to the LattePanda motherboard of the robot over WIFI with SSH.
2. The programmer starts a script on the motherboard to send his program to the multifunction card over USB, selecting the servomotor to target and the software to update (bootloader or application).
3. The multifunction card initiates the update procedure.
4. The MCU processes the update.
5. The MCU is updated!

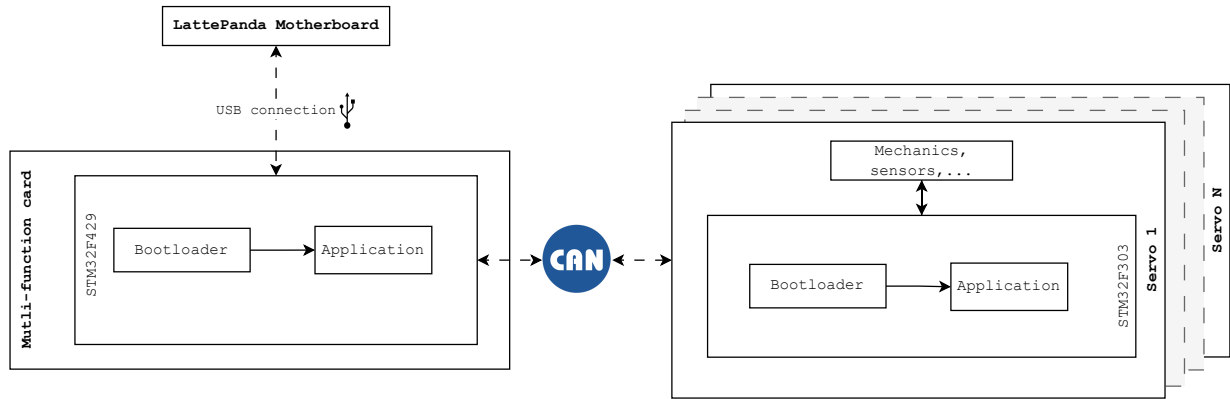


Figure 2.1: Organisation of the communication in the robot body.

Given these objectives, the personal project has been divided into the following sections:

1. Creation of an installation guide (macOS, WSL and Linux versions) for the embedded tool suite (arm-none-eabi [19], git, ssh, libopenm3, OpenOCD [20]).
2. Creation of hands-on C code with libopenm3 and FreeRTOS to learn how STM32F3 embedded cards work. These examples are documented as a reference for the computer science team. This part allowed the author to familiarise himself with libopenm3 and the STM32F3 MCU.
3. Creation of a debugging guide for the STM32F3 using OpenOCD and GDB. This part allowed the student to debug his code.
4. Review of the STM32F3 MCU and Cortex-M4 core functionalities as a reference for the computer science team. This part allowed the student to understand the inner working process of these complex systems, necessary to develop the bootloader.
5. Design and documentation of a bootloader with recovery mechanisms.
6. Development of an MVP<sup>1</sup> for the bootloader.

<sup>1</sup>Minimum Viable Product

## Chapter 3

# Introduction to the Arm® Cortex®-M4 and STM32F303

### 3.1 Introduction

The microcontroller used to control the servomotors and communicate with the rest of the robot is the [STM32F303CC](#). This MCU is equipped with an ARM Cortex-M4 core, which is a feature-rich, high-performance core.

### 3.2 Processor vs Microcontroller

The Cortex-M4 processor (interchangeable with *core* in this document) is the processor of the STM32F303CC MCU. Here are the main characteristics of the Cortex-M4:

- Release year: 2010
- Instruction pipeline: Three-stage pipeline. This allows the processor to speed up the execution by fetching a new instruction while others are being decoded and executed.

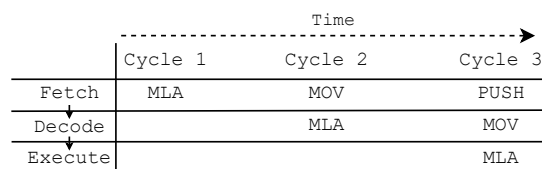


Figure 3.1: Three-stage pipeline illustration

- The processor relies on the Harvard architecture, i.e. the instructions and the data live in the same memory space **but** can be accessed simultaneously through parallel buses, reducing latency. For example, when an exception occurs, the processor can take advantage of its bus architecture to push the state of the registers to the stack while simultaneously fetching the interrupt vector and the exception handler instructions.
- 32 bits addressing, allowing up to  $2^{32} \text{ Bytes} = 4 \text{ GB}$  of memory space.
- A Nested Vector Interrupt Controller (NVIC) allowing up to 240 interrupt requests, with 8 to 256 interrupt priority levels.
- Floating Point Unit (FPU) for single precision floating point operations.
- Thumb-2 instruction set: an ARM instruction set composed of 16-bit and 32-bit instructions.
- Instructions for advanced features like: function calls, system control, OS support, single instruction multiple data (SIMD), fast multiply and accumulate (MAC).

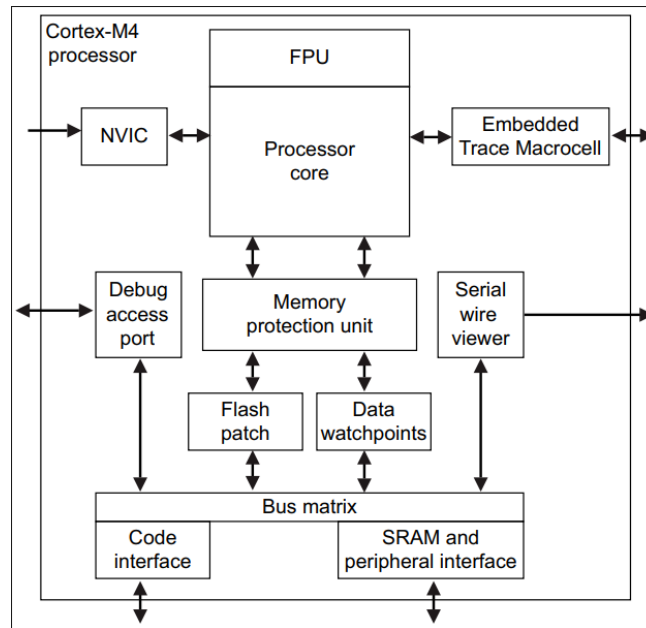


Figure 3.2: Architecture of the Cortex-M4 (Source: ST [21])

A microcontroller is composed of a processor and many vendor-dependent peripherals (support for communication protocols, flash memory size, clocks, GPIO purpose,...). Here are the main characteristics of the STM32F303CC:

- Operating frequency: 72MHz
- Flash memory size: 256 KBytes
- RAM size: 48KB
- Dimensions: 7x7mm 48-pin package. The STM32F3 is the most powerful MCU available in this form factor
- Communication support: CAN 2.0, I2C, SPI, I2S, USART, UART, USB
- Other peripherals: Op-amps (4), 16-bit and 32-bit timers, Watchdog timers, A/D - D/A converters (6)

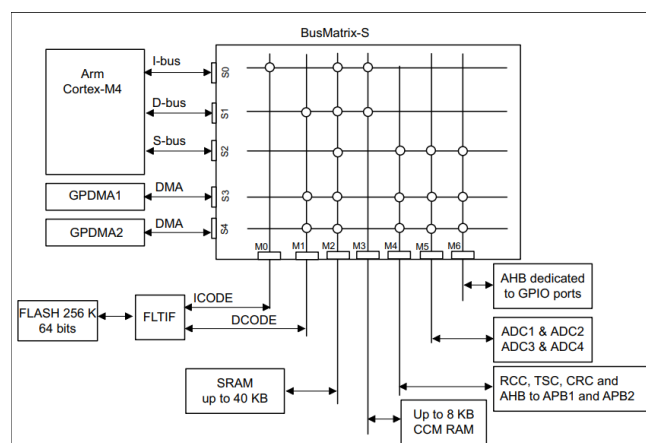


Figure 3.3: Architecture of the STM32F303, the Cortex-M4 is only a fraction of the architecture of the MCU. (Source: ST [22])

## 3.3 Architecture

### 3.3.1 Instruction Set Architecture (ISA)

The Instruction Set Architecture (ISA) is the set of instructions designed for a specific CPU architecture.

The Cortex-M4 processor is based on the ARMv7-M architecture. The ISA of ARMv7-M is special in the way that it contains both 16 and 32-bit instructions that can be mixed; this ISA is based on Thumb2-technology which allows the mixing of instructions. Mixing the two types of instructions allows for increasing code density with the 32-bit instructions while taking advantage of 16-bit instructions that are easier to decode, and that are sufficient for simple instructions.

Earlier Cortex architecture already worked with a set of 16 bits (Thumb set) for small instructions and another set of 32 bits (ARM set) for complex instructions. In this case, the programmer had to switch the operation state of the processor to go from one to the other, which added overhead. For example, with a JMP instruction, the programmer had to specify how the CPU would have to decode instructions at the new location using the least significant bit (LSB) of the instruction address. The LSB not being used for addressing as an instruction is at least 2 Bytes. Thus, the programmer could set it to 0 to switch to ARM mode (32-bit instructions) or 1 to switch to thumb mode (16-bit).

In the ARMv7-M architecture, the Thumb set has been extended to include 32-bit instructions, and it does not support the 32-bit ARM set anymore, along with its operation state. Using a branching instruction on an odd address would cause an exception.

#### Barrier instructions

On Cortex-M4, a write to memory is slower than an instruction cycle; thus, a write buffer is used to temporarily store the data and copy it to memory while the following instructions are being executed. This functionality, combined with the multi-stage pipeline, can lead to undesired behaviours; thus, barrier instructions are provided.

- Data Memory Barrier (DMB): ensures all memory accesses are completed before the next one following the DMB is processed.
- Data Synchronisation Barrier (DSB): ensures all memory accesses are completed before the next instruction following the DSB is executed.
- Instruction Synchronisation Barrier (ISB): flushes the instruction pipeline and ensures previous instructions are executed before the next instruction following the ISB.

### 3.3.2 Exceptions

“Exception” is a general term that encompasses all the events that interrupt the normal flow of a program. These exceptions can be generated by peripherals (e.g. message received on the CAN bus, a timer reaching 0,...), in which case they are called interrupts. When an exception is raised, an exception handler processes the exception and then returns control to the main program.

#### Nested Vector Interrupt Controller (NVIC)

The NVIC is the component responsible for handling interrupt requests. It takes as input an Interrupt Request (IRQ), identifies the request and starts the routine responsible for processing the interrupt. Traditionally, the identification of the interrupt type and the localisation of the routine are done in software. However, on the Cortex-M4 processor, this is done in hardware. The exceptions are all identified by an exception type number. This number identifies the routine to call to handle the interrupt. The addresses of the routines are stored in order in a Vector Table. Thus, at interrupt time, the processor can fetch the first instruction of the interrupt routine by reading the address stored at: `vector_table_address + 4 * exception_type`. This results in a faster entry in the interrupt routine.

The NVIC unit also contains the System Control Block (SCB). This block contains registers for system control, notably:

- CPUID: ID identifying the processor type and revision.
- Interrupt Control and State Register ICSR: can be used to determine exception/interrupt status.
- Vector Table Offset Register (VTOR): location of the vector table (default at 0x8000000).
- Application Interrupt and Reset Control Register (AIRCR): can be used to request a reset.

### Reset interrupt requests

The Cortex M4 supports three reset sequences for the MCU:

- Power on reset: Reset the processor, the peripherals and the debug component.
- System reset: Reset the processor and the peripherals.
- Processor reset: Reset the processor.

After a reset, the VTOR register is restored to its default value (0x8000000), the interrupts are disabled, the interrupt pending statuses are cleared, the Master Stack Pointer (MSP) is initialised with the word at the VTOR address and a jump to  $VTOR + 4$ , the address of the reset handler, is performed.

To initialize a system request one can call the [`scb\_system\_reset\(\)`](#) routine from `libopenm3` which sets the SYSRESETREQ bit (bit 2) of the AIRCR register.

### 3.3.3 Vector table

The vector table is a table containing the initial address of the stack pointer, the address of the reset handler and the addresses of all the other interrupt routines, sorted by interrupt type. These addresses are called interrupt vectors. The vector table is usually placed at the start of the program, but can be relocated using the VTOR register. In all circumstances, an initial vector table must be provided at address 0x8000000 as it is used by the hardware upon reset to initialise the MSP and localise the reset handler.

### 3.3.4 Operation modes

Two modes are defined for code execution:

- Handler mode: to execute ISRs.
- Thread mode: to execute application code.

In Thread mode, two access levels are defined:

- Privileged: Default starting mode for the MCU. In this mode, the Stack Pointer (SP) used is the Main Stack Pointer (MSP).
- Unprivileged: To be used by an RTOS to control user code access to some functionalities of the MCU (e.g. NVIC registers access is disabled). SP is switched from MSP to Process Stack Pointer (PSP).

This level has to be selected from the CONTROL register. Only a switch from privileged to unprivileged is possible. Unprivileged code can not set access level to privileged; instead, the system can rely on interrupts like SysTick to switch to privileged access to run privileged code (e.g. Operating System). If unprivileged code generates an exception, the privileged code (OS) can recover and continue its execution along with other unprivileged tasks.

### 3.3.5 Clock tree

To orchestrate the operations in an MCU, the system relies on clocks. These clocks can be seen as a heart, rhythmically timing the MCU operations. However, all components (peripherals, CPU) do not run at the same frequency. Furthermore, not all components need to be powered by a clock all the time; powering all the components all the time increases power consumption. To deal with these issues, the

STM32F303CC implements a complex clock tree connecting all components of the MCU to multiple clock sources through buses and switch mechanisms.

The clocks available are:

- High-Speed External (HSE): usually used as input for the System Clock (SYSCLK) to power the CPU and reach its maximum operation speed of 72MHz. This clock is based on an external crystal oscillator. The HSE is directed through a Phase-Locked Loop (PLL) to multiply its original frequency and multiply it to reach 72MHz.
- High-Speed Internal (HSI): default high speed clock reaching 8MHz. Usually replaced by HSE to reach maximum speed.
- Low-Speed External/Internal (LSE, LSI): used in low-power mode.

The complete clock tree diagram can be found in [22], p.126.

## 3.4 Memory

### 3.4.1 Memory map

On the STM32F303, all MCU functionalities (FLASH, RAM, Core registers, Peripherals) are mapped together to the memory in the following manner:

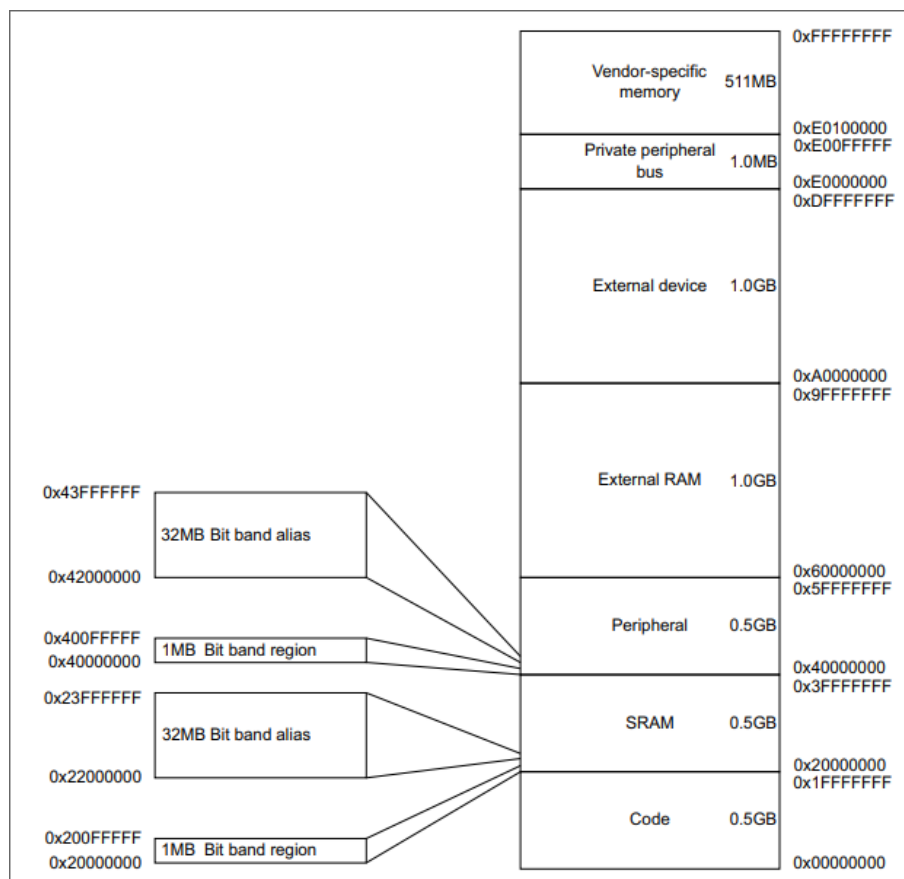


Figure 3.4: Memory map of the STM32F303CC (Source: [23] p.28)

This makes the access to any functionality very convenient, as one can use a simple C pointer to control any part of the MCU.

### 3.4.2 Flash memory

Although it is possible to run code from RAM in the SRAM region, the programs are usually executed from FLASH memory. On the STM32F303, the start of the flash memory is mapped to address 0x00000000 (alias to 0x80000000). The flash memory is composed of 128 2KB pages, making it 256KB. Flash memory can not be written to as easily and as fast as RAM. The following sequence has to be respected:

1. Unlock flash memory
2. Identify the page on which data has to be written
3. Copy existing content from the page (If to be preserved)
4. Copy data to the page and restore existing content
5. Verify integrity
6. Lock flash memory

An implementation of this procedure in a test called “flash-rw” is developed in Section [7.4](#).

#### Write protection

On the STM32F303cc, the FLASH module allows for protecting pages in write access, with a granularity of 2 pages. The write protection is managed by the Write Protection Register (FLASH\_WRP). More details can be found in [22], p.79.

### 3.4.3 Cyclic Redundancy Check (CRC)

CRC is an error-detecting technique used in networking and storage systems. It allows the programmer to efficiently compute an error-detecting code that can be used to detect corrupted packets received on a network or corrupted data on a disk. Depending on the nature of the network, packets can be altered. Thus, one may want to compute a CRC code of the data to be sent, send the CRC along with the data, and check data integrity on reception using the sent CRC code. On storage devices, the FLASH memory has a write limit and eventually becomes unwritable. To ensure that data has been correctly written to the FLASH memory, one can compute a CRC code for the data to be written, write the data and check data integrity using the CRC.

To efficiently compute these CRC codes, the STM32F303CC provide a CRC peripheral that uses the CRC-32 (Ethernet) polynomial: 0x4C11DB7. A routine, `generate_checksum()`, implementing the CRC code generation with the CRC peripheral using `libopencm3` can be found in the file `code/servos/shared/flash.c` on the `robocup-info` repository. The instructions on how to access it are given in Section [7.2](#).



## Chapter 4

# ARM executables

### 4.1 Introduction to Executable and Linkable Format (ELF)

If we compile a program with `arm-none-eabi-gcc` and inspect it with `file executable_name`, we get the following result: `executable_name: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, with debug_info, not stripped`

The ELF format defines a standard for executable files, among other types of files (object code, shared libraries, core dumps). These files may contain much information. In the scope of this project, we are interested in the following:

- ELF header: contains file information, target architecture, entry point address
- A Section header table describing sections of the executable
- The sections containing the program itself (code + data):
  - `.text` section: contains the program code.
  - `.data` section: contains the initialised read-write global and static variables. These variables live in RAM at execution time.
  - `.bss` section: contains the uninitialized read-write global and static variables. These variables live in RAM at execution time.
  - `.rodata` section: contains read-only data (constants, string literals).

When the programmer flashes a program to the MCU, the `.data` section is stored to flash along with the size of the `.bss` section. It is part of the role of the reset handler, which is the first piece of code executed on the microcontroller, to initialise these sections in RAM. In `libopencm3`, it is the [`reset\_handler\(\)`](#) routine that implements this initialization.

## 4.2 Inspection of the executable “blink”

The following sections use the arm-none-eabi tools suite to inspect the content of the executable “blink”, the source code of which is provided in Section 7.2.

If we inspect blinky with arm-none-eabi-objdump -x blinky, which displays the content of the headers, we get:

```

1 > arm-none-eabi-objdump -x blinky
2 ...
3 start address 0x08000531
4
5 Program Header:
6   LOAD off 0x00001000 vaddr 0x08000000 paddr 0x08000000 align 2**12
7     filesz 0x0000060c memsz 0x0000060c flags r-x
8   LOAD off 0x00002000 vaddr 0x20000000 paddr 0x0800060c align 2**12
9     filesz 0x0000000c memsz 0x0000000c flags rw-
10 private flags = 0x5000400: [Version5 EABI] [hard-float ABI]
11
12 Sections:
13 Idx Name          Size      VMA      LMA      File off  Algn
14  0 .text          0000060c  08000000  08000000  00001000  2**2
15                  CONTENTS, ALLOC, LOAD, READONLY, CODE
16  ...
17  4 .data          0000000c  20000000  0800060c  00002000  2**2
18                  CONTENTS, ALLOC, LOAD, DATA
19  5 .bss           00000000  2000000c  08000618  00000000  2**0
20                  ALLOC
21  ...

```

Listing 4.1: Headers content

The Start Address of the program is detected as **0x08000531**. The size of the .text section is 1548Bytes (0x60c) and will be flashed at address 0x8000000. The size of the .data section is 12Bytes (0xc) and will be flashed at address 0x800060c (thus, right after the .text section). The Virtual Memory Address (VMA) differs from the Load Memory Address (LMA). When the program is flashed, the .data section will be stored at the LMA address. When the program is started, the reset handler will copy this section to RAM (Starting at 0x20000000). The size of the .bss section is 0 (because there are no uninitialized variables in the code) and will be flashed at 0x8000618 (thus, right after the .data section).

If we inspect blinky with arm-none-eabi-objdump -d blinky, which disassemble the executables sections, we get:

```

1 > arm-none-eabi-objdump -d blinky | head -n20
2
3 blinky:      file format elf32-littlearm
4
5
6 Disassembly of section .text:
7
8 08000000 <vector_table>:
9 80000000:      00 a0 00 20 31 05 00 08 2d 05 00 08 2b 05 00 08      ... 1...-...+...
10 80000100:      2b 05 00 08 2b 05 00 08 2b 05 00 08 00 00 00 00      +...+...+.....
11 ...

```

Listing 4.2: Text content. Values are stored in Little Endian.

The first section to be disassembled is the vector table, as already discussed in Section 3.3.3. As explained in 3.3.3, the first value is the initial SP value: 0x2000a000. The stack grows downward, meaning we have:

$0x20000000$ (RAM start address) –  $0x2000a000 = 40KB$   
of space for the stack.

The second value is the absolute address of the reset handler, the entry point of the program:  $0x8000531$ . Which corresponds to the value we found in the ELF header with `arm-none-eabi-objdump -x blinky`.

The reason why the address is odd is explained in Section 3.3.1.

If we inspect the output, starting at address  $0x8000530$ , we find the reset handler:

```

1 08000530 <reset_handler>:
2 8000530:      b538          push    {r3, r4, r5, lr}
3 ...
4 8000568:      4c15          ldr     r4, [pc, #84] @ (80005c0 <reset_handler+0x90>)
5 800056a:      4d16          ldr     r5, [pc, #88] @ (80005c4 <reset_handler+0x94>)
6 800056c:      42ac          cmp     r4, r5
7 800056e:      d312          bcc.n   8000596 <reset_handler+0x66>
8 8000570:      f7ff fe10      bl      8000194 <main>
9 ...

```

Listing 4.3: Reset handler

As expected, we can see that after initialisation of the `.data` and `.bss` segments, the reset handler branches to the main function with a `bl` instruction (branch to label, save return address in Link Register (LR)).

# Chapter 5

## Bootloader

### 5.1 Bootloader - Version 0

#### 5.1.1 Introduction

Here, the author presents version 0 of the bootloader. Version 1 is a second iteration of this design of the bootloader. The version 0 design and diagrams are provided as a reference. To provide the features presented in Section 2, the bootloaders in the sections below have been proposed. Each of these bootloaders is an executable compiled separately and flashed to its dedicated location in flash memory.

#### 5.1.2 Illustrations

Flow diagrams of the bootloaders and an illustration of the memory organisation can be found in Section 7.8.

#### 5.1.3 Start bootloader

The start bootloader is the executable started on Power On by the MCU. This bootloader acts as a switch mechanism, choosing the next executable to run (main bootloader, recovery bootloader or the application). As shown in Section 7.8, the decision is based on a FLASH flag, `ERROR_MODE`, and a RAM flag, `BOOT_MODE`. The FLASH flags are stored in a page right after the start bootloader executable as illustrated in 7.8.

#### FLAGS values

- `ERROR_MODE` can be set to 1 to start the recovery bootloader (Initially set to 0).
- `BOOT_MODE` can be set to `BOOT_MODE_START_APP` (0xDEADCAFE) to start the application (Initially set to 0).
- By default, the main bootloader is started.

#### 5.1.4 Main bootloader

The main bootloader is the executable responsible for receiving and applying the updates for the application code and itself. A timer is used because the update requests come from the main board, which sends them to the multifunction card. The main board takes dozens of seconds to start on Power Up before being able to send an update request. Thus, on power-up of the robot, the multifunction card starts by broadcasting a **Wait** message to the bootloader of the servos.

If a **Wait** message has been received before the timer runs out, the bootloader will wait indefinitely for an Update, Boot or Reset request.

Else, the `BOOT_MODE` is set to `BOOT_MODE_START_APP` to start the application, a System reset is sent, and the start bootloader will start the application after the reset.

### Start the application

When the work of the main bootloader is finished, it waits for a reset request to launch the start bootloader instead of directly jumping to the application. It is the start bootloader that will start the application. It is done this way so that the application executable can initialise as it wants the clock tree and the peripherals of the MCU.

Another approach would be to start the application from the main bootloader. In this case, one would have to reset manually all the peripherals and the clock tree. This approach is prone to bugs if one does not reset a peripheral correctly. For example, if the clock tree is not correctly reset and if the application code starts with `rcc_clock_setup_pll(&rcc_hse8mhz_configs[RCC_CLOCK_HSE8_72MHZ]);` (which initialises the clock tree to reach the maximum speed of 72MHz), the application will be stuck in `rcc_clock_setup_pll()` indefinitely.

#### 5.1.5 Recovery bootloader

The recovery bootloader acts as a last resort bootloader in case an issue has been detected. This bootloader is not upgradable. The recovery bootloader can be used to update the bootloader, write data (e.g. modify the flags in the flags FLASH page) or read data. To start it, the `ERROR_MODE FLASH` flag must be set to 1. The flag is set to 1 if:

- The main bootloader had an issue during an update (Robot lost power during the update, CRC of the update written in memory does not match the one received).
- The main bootloader received a Boot request with its Error Mode (EM) parameter set to 1. This mechanism can be useful if a main bootloader with a faulty update algorithm has been flashed. In this case, the main bootloader can not be used to fix itself using its update mechanism, and the recovery bootloader must be used instead.

## 5.2 Bootloader - Version 1

### 5.2.1 Introduction

Version 0 provided good foundations, but the following points led the author to provide an improved version, version 1.

1. The start bootloader and the recovery bootloader can be merged. Indeed, separating the two executables wastes space as they are flashed in separate pages, leading to potentially more internal fragmentation.
2. The bootloader having to update itself complicates the entire update flow. First, given that the bootloader has to update itself, it implies that it must be moved around in memory to not overwrite itself while being executed, as illustrated in Figure 7.5 and Figure 7.6.  
In addition to the manipulation in memory, it also causes issues with sending an update. In this case, the programmer must send 2 executables for an update (of the main bootloader or the application): one compiled to be flashed after the flags page (0x8001800) and one compiled for the end of the flash memory. Even more, because updating the main bootloader implies writing the new bootloader at the location of the application, the application is erased and must be sent back after the update of the bootloader. This constraint complicates again the update sequence of the main bootloader.
3. The memory organisation does not facilitate the addition of an intermediate bootloader between the main bootloader and the application.

Thus, a new approach has been proposed: a multi-stage bootloader. In this version, a bootloader at stage  $x$  is responsible for updating the stage  $x+1$  (which can be the application), starting the stage  $x+1$  and possibly additional functionalities. The bootloaders are simplified and always placed in a fixed location in flash. Sending updates is easier, and the update procedures are simplified.

### 5.2.2 Illustrations

Flow diagrams of the bootloaders, an illustration of the memory organisation, and examples of network exchange sequences can be found in Section 7.9.

### 5.2.3 Stage 1 bootloader

The stage 1 bootloader is the executable started on Power On by the MCU. The stage 1 bootloader is kept as simple as possible, as it is a critical part of the bootloading process. This bootloader is only responsible for starting the next stage (which can be the application) or updating it. When updating the stage 2, a blue LED is lit up.

#### Recovery mode

When the bootloader is in recovery mode, a red LED is lit up and the stage 1 bootloader waits indefinitely for an update from the multifunction board.

#### Initialization of the BOOT\_MODE ram flag

To know if the bootloader should start the stage 2 bootloader or wait for an update, the stage 1 bootloader relies on a RAM flag called "BOOT\_MODE". When a bootloader at stage  $x$  receives a Boot message, it sets BOOT\_MODE to  $x+1$ . This flag must be initialised at the power-up of the MCU. The following initialisation is done with the following code at the beginning of the stage 1 bootloader:

**IF** word at specific address is ram != MAGIC\_NUMBER (0xDEADCAFE)  
**THEN** BOOT\_MODE = 0

### 5.2.4 Stage 2 bootloader

The generic read/write feature, which was provided by the recovery bootloader in version 0, might be useful to have during the bootloading process to debug issues/write data at a specific location. Thus, this bootloader will integrate this functionality, along with the update of the stage 3 (the application). The stage 2 bootloader will be able to support updated versions of the CAN bus protocol, as one could update this executable using the stage 1 bootloader.

**Note:** This stage 2 bootloader has not been developed yet, but when it is, one will be able to flash it to memory by sending it as an update to the stage 1 bootloader. This bootloader will overwrite the application present at its location. Finally, the programmer will be able to send an update to the stage 2 bootloader to write the application.

### FLASH memory flags

To store persistent information related to the servo, 1 page (2048B, 512 words) of FLASH memory is allocated to "flags" variables. This page is located right after the stage 1 bootloader. The flags are stored in this order:

1. Offset 0: CAN\_ID. The ID used by the servo to identify itself on the CAN bus. (Initially set to 1).
2. Offset 4: ERROR\_MODE. If an error arises during an update, this flag will be set to x, where x is the stage in which the error occurred. (Initially set to 0).
3. Offset 8: STAGE\_1\_VERSION. The version of the stage 1 software. Even though this software is not meant to be updated, new versions could be developed in the future. Thus, one may want to check which version is installed on a given servomotor. (Initially set to 0).
4. Offset 12: STAGE\_2\_ADDR. The location of the start of the stage 2 executable.
5. Offset 16: STAGE\_2\_VERSION. The version of the stage 2 software. (Initially set to 0).
6. Offset 20: STAGE\_3\_ADDR. The location of the start of the application.
7. Offset 24: STAGE\_3\_VERSION. The version of the application. (Initially set to 0).

As more stages are added, more flags can be added to this page, with a limit of 512 32-bit flags.

### 5.2.5 Networking

To communicate between the bootloader of the servomotors and the multifunction card, a protocol over CAN bus has been designed and developed by Maxime Léonard [12]. The documentation of the messages available can be found in Section 3 of [12]. The source code can be found in the `code/networking` folder of the robocup-info repository.

### 5.2.6 Test bench

As the multifunction card and the modified servos were not available at the time of this work, the test bench described in Section 5 of [12] has been used to test the update mechanism of the bootloader. The multifunction card is replaced by an STM32F429I-DISCOVERY, and the servomotor by an STM32F3DISCOVERY.

### 5.2.7 Implementation of the jump mechanism

The main functionality the bootloader must provide is the ability to start an executable from another one. To do so, one can follow the next procedure:

1. Localise the start address (which we will call `application_vector_table_address`) of the executable in memory.
2. Initialise the VTOR register with the start address.
3. Set access level to privileged.
4. Jump to the address stored at `application_vector_table_address + 4`.

An implementation of this procedure using `libopenm3` is provided in Section 7.7.

### 5.2.8 Usage

The bootloader code is located in the folder `code/servos/bootloader-v1/` on the `robocup-info` repository. To install the bootloader **version 1** along with a first application (Blinky with FreeRTOS) on the STM32F303, one must:

1. Follow instructions provided in 7.2 to install the required tools and to access the repository.
2. Open a terminal in the folder `code/servos/bootloader-v1/`
3. Connect a STM32F303CC to their computer.
4. Type `make flash` to compile the bootloader, the application and flash them to memory.
5. The bootloader is now ready to receive updates!

To try the update mechanism, one must:

1. Connect an STM32F429I-DISC1 and an STM32F3DISCOVERY on the electrical assembly described in Section 5 of [12].
2. Flash the `code/multifunction/bootloader-v1-master/` project to the STM32F429I-DISC1 using the `make flash` rule. This program will send a new Blinky program (Anti-clockwise Blinky) to the STM32F3DISCOVERY.
3. Power on the electrical assembly.
4. The update is now being processed!

### 5.2.9 Limitations

- The flash memory is not write-protected by the bootloader; thus, if a bug in the application writes data to flash memory in unexpected ways, it may alter the bootloader and lead to unrecoverable loss of the bootloading process. In this case, one would have to manually flash the bootloader again with a debugger connector. However, this is unlikely to happen unexpectedly, as one has to unlock the flash memory using the FLASH registers and then erase it to alter the bootloader.
- Update a bootloader at stage  $x$  may require updating the next bootloaders (from  $x+1$  to  $x_t$ , with  $x_t$  being the application) as they may get overwritten by the update at stage  $x$ . This is not that much of an issue, as the update of a bootloader is rare. When the necessity of an update arises, one could write a new bootloader-master code for the multifunction board to update in sequence the different bootloaders.
- The stage 1 bootloader is not capable of updating itself. However, in the case where an update of this bootloader is **really** necessary, one could temporarily update the stage 2 bootloader with an executable capable of updating the stage 1 bootloader, let it update the stage 1 bootloader and finally use the new stage 1 bootloader to update the stage 2 bootloader. **Warning**, this operation must be executed with caution, as if the update for the stage 1 is faulty, one will have to manually restore it using the debugger connector of the servomotor.



## Chapter 6

# Conclusions

In this work, the author has studied and documented the architecture of the STM32F303CC MCU and Cortex-M4 core. Based on this knowledge, he then implemented a series of small learning projects to experiment with the MCU. The author proposed two iterations of the design of a bootloader, version 0 and version 1, for the STM32F303CC platform. Finally, he described the usage of the current implementation and its limitations.

The bootloader allows the programmers of the Robocup Montefiore Team to install software updates on the servomotors from their computer, without the need to connect to the servomotors with a debugger connector. This crucial piece of software can be used to program an application or a bootloader with more elaborate capabilities, and it will unlock new developments in the next year for the Robocup team.

## Chapter 7

### Future work

In the continuity of the work done in this project, the Robocup Montefiore Team will continue with:

- Adaptation and testing of the bootloader for the new servomotor board. Indeed, the bootloader presented in this work was developed for the STM32F3DISCOVERY board, not the servomotors board, whose board is under development. Thus, some tweaks (modification of the initialisation of the CAN bus, modification of the LEDs pins,...) might be needed to make it work on the servomotors board. Furthermore, only one STM32F3 was connected to the CAN bus during the tests; more testing must be done with multiple MCUs connected to the CAN bus.
- Development of the stage 2 bootloader with a generic read/write operations functionality.
- Installation of the bootloader on the 22 servomotors of the robot.
- Development of the software suite to allow a programmer to place an executable on the robot's main board and have it sent automatically to the servomotors via the multifunction card.
- Adaptation and testing of the bootloader for the multifunction board. This bootloader will require further development as the updates will not be received from the CAN bus but from the main board on the USB port. However, most of the heavy lifting of the bootloading algorithm has already been developed in this project, and this new bootloader should be seen as an extension of the one presented in this work, with USB capabilities.

# Appendices

## 7.1 Robocup-info Git Repository

To facilitate the collaboration, the robocup-info repository has been organised in the following way:

- **code/**
  - **freertos/**: folder with the configuration of FreeRTOS.
  - **FreeRTOS-LTS/**: submodule folder for the FreeRTOS repository.
  - **libopencm3/**: submodule folder for the libopencm3 repository.
  - **mainboard/**: folder for the LattePanda motherboard projects.
  - **multifunction/**: folder for the multifunction board projects
    - **template/**: template project with up-to-date Makefile to start a new project with libopencm3.
    - **template-freertos/**: template project with up-to-date Makefile to start a new project with libopencm3 + FreeRTOS.
    - **stm32f4/**: folder with general-purpose code (control of the LCD, dynamixel library...) to be shared between projects.
    - **bootloader-v1-master/**: folder with the implementation of a master node sending an application update to a bootloader on an STM32F303CC MCU.
    - **Makefile.include**: Makefile with rules related to the STM32F4 platform.
    - **open.cfg**: OpenOCD config file for the STM32F4 platform.
    - **stm32f4.ld**: Link script for the STM32F4 platform.
  - **stm32/**: folder with the CAN library source code, dummy syscalls, utility functions, executable startup code,...
  - **servos/**: folder for the servos projects.
    - **template/**: template project with up-to-date Makefile to start a new project with libopencm3.
    - **template-freertos/**: template project with up-to-date Makefile to start a new project with libopencm3 + FreeRTOS.
    - **stm32f3**: folder with general-purpose code (flash routine, gpio configuration, clocks manipulation) to be shared between projects.
    - **bootloader-v1/**: folder with the implementation of the bootloader.
    - **hands-on**: folder with the hands-on (A description of the hands-on available is described in Section 7.4).
    - **Makefile.include**: Makefile with rules related to the STM32F3 platform.
    - **open.cfg**: OpenOCD config file for the STM32F4 platform.
    - **stm32f4.ld**: Link script for the STM32F3 platform.
  - **rules.mk**: Makefile with the rules to be used by all the projects.

All the projects and Makefile are documented with a `README.md` file located in their respective folder and/or directly in the source code/Makefile.

## 7.2 Access to resources

Most of the resources are not available to the public. A request to Pr. Boigelot ([bernard.boigelot@uliege.be](mailto:bernard.boigelot@uliege.be)) can be made to access the project's Wiki (<https://people.montefiore.uliege.be/robocup/wiki/>) and the git repositories.

The hands-on are available on the robocup-info git repository (`robot.montefiore.ulg.ac.be:robocup-info`).

## 7.3 Installation guide

The installation guide for macOS, WSL and Linux can be found on its page on the Wiki ([url](#)). The reader is invited to follow the instructions on this page before starting development.

## 7.4 Hands-on with libopencm3

The libopencm3 hands-on can be found on the robocup-info git repository (`robot.montefiore.ulg.ac.be:robocup-info`) under `code/servos/hands-on`.

- `flash-rw-example`: In this hands-on, the Flash memory module of the microcontroller is introduced. The program flashes a string to memory and then reads it to compare it with the initial string. An explanation of the code is available in `flash-rw-example/README.md`.
- `mini-boot`: In this hands-on, a small executable (`blinky`) is flashed from the `mini-boot.c` programs and jumps to it. The executable of `blinky` has been converted to an `uint8_t` array (`application_text_section`), making it easy to flash it programmatically. An explanation of the code is available in `mini-boot/README.md`.
- `medium-boot`: In this hands-on, the same result is obtained, but the `blinky` executable is extracted programmatically with a Python script, `extract_elf_data.py`, which reads and extracts the content of the ELF file `blinky`. An explanation of the code is available in `medium-boot /README.md`.
- `reset-on-ram-flag`: In this hands-on, a branching mechanism based on a flag stored in RAM is implemented. An explanation of the code is available in `reset-on-ram-flag/README.md`.

## 7.5 Guide to GDB for debugging on the STM32 platform

### 7.5.1 Introduction

Sometimes things go wrong while programming. We can use a debugger like GDB [24] to help us find out what. The GNU Debugger (GDB) is a software tool capable of inspecting code, variables, registers, tracing, examining and altering the execution of a program running on the hardware.

A hands-on code can be found in `code/servos/hands-on/blink-debug` with explanations and commands in its `README.md`. The code is the official blinky code from `libopenm3` [25]. The makefile has been modified to compile blinky in debug mode (`-O0` option to disable optimisation and `-g` option to generate debug information for GDB) and to add a debug rule to start the GDB server.

The following steps assume the reader has followed the Installation guide 7.3, downloaded the `blink-debug` project and has a terminal open in the project folder.

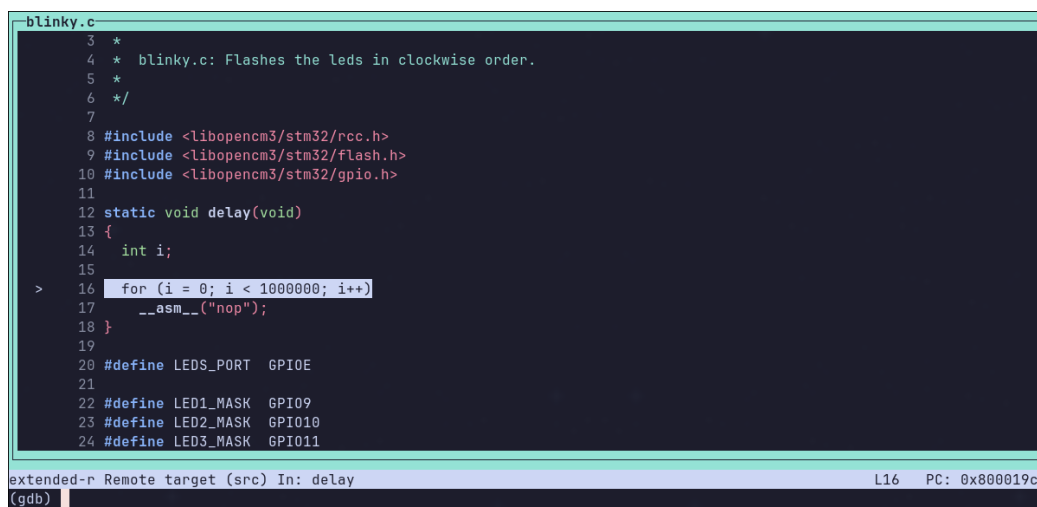
### 7.5.2 Getting started

Today, most UNIX-based OSes come with GDB installed. However, the STM32F3 MCU is based on ARM, which prompts the need for cross-compiling tools. We recommend installing `arm-none-eabi-gdb`, a version of GDB that supports debugging on the ARM architecture. `arm-none-eabi-gdb`, which is part of the `arm-none-eabi` tools suite, can be installed by following the steps in the installation guide 7.3.

### 7.5.3 Start a debugging session

Debugging a program requires the following steps:

1. Connect your MCU to your computer. In our case, it will be an STM32F3DISCOVERY via its USB port “USB ST-LINK”.
2. Flash `blink` to the MCU with the command `make debug`. The LEDs on the board should start blinking clockwise. This rule will also start the GDB server. The GDB server allows a GDB client to connect to the MCU running `blink`. In the case of `blink-debug`, it is `OpenOCD` that will provide this server.
3. Start the GDB client from another terminal with the command `arm-none-eabi-gdb blink`. It will start the client and load the executable “`blink`” in its memory. From GDB, use `target extended-remote :4242` to connect to the MCU, then enable the GDB layout with `tui enable`.



```

blinky.c
3  *
4  * blinky.c: Flashes the leds in clockwise order.
5  *
6  */
7
8 #include <libopenm3/stm32/rcc.h>
9 #include <libopenm3/stm32/flash.h>
10 #include <libopenm3/stm32/gpio.h>
11
12 static void delay(void)
13 {
14     int i;
15
16     for (i = 0; i < 1000000; i++)
17         __asm__("nop");
18 }
19
20 #define LEDS_PORT  GPIOE
21
22 #define LED1_MASK  GPIO9
23 #define LED2_MASK  GPIO10
24 #define LED3_MASK  GPIO11

```

extended-r Remote target (src) In: delay L16 PC: 0x800019c  
(gdb)

Figure 7.1: GDB with TUI enabled.

### 7.5.4 Debugging

Now that the GDB is connected, one can interact with the program, run it step by step, inspect variables,... Here is a list of useful commands:

Command	Abbrev.	Description
help	h	Display help information.
layout <content>	∅	Add content in the GDB window (asm to show assembly code, src for the C code, regs for the registers of the CPU).
break <location>	b	Set a breakpoint at the given location (function name, file:line, or address).
continue	c	Continue the execution to the next breakpoint.
next	n	Execute the current instruction, without entering the function call if any.
step	s	Execute the current instruction (the one preceded by >) and enter the function call (if the instruction is a function call).
finish	∅	Jump to the end of the current function
print <expr>	p	Print the value of an expression / variable.
info <subject>	∅	Display info about a subject (breakpoints, locals, target, mem)
delete <num>	d	Delete a breakpoint by number.
monitor reset halt	∅	Reset the execution of the program.
x/<nb><format><size>	∅	Examine memory. E.g. x/12xw 0x8000000 will display memory content, starting at address 0x8000000, of 12 words (32 bits) in hexadecimal.
focus <content>	fs	Set focus to content (src, cmd, asm,...). E.g. fs cmd will focus on the command line, allowing to use of tabs for autocompletion and arrow-up/down to navigate command history.

Table 7.1: GDB commands. The commands can be replaced by their abbreviation.

### 7.5.5 More readings

The README.md located in the `blinky-debug` project can be consulted for help with GDB. An overview of GDB for STM32 can be found here: [ST Wiki](#). A recommended reading is chapter 21, “Troubleshooting”, from “Beginning STM32” [26]. Finally, a GDB manual can be found here: [Sourceware](#).

## 7.6 Hands-on with FreeRTOS

### 7.6.1 Introduction

FreeRTOS [27] is the operating system that has been chosen to support the servomotors' control software. This operating system is small and simple. Written in C, it provides facilities like tasks, mutexes, semaphores and timers.

### 7.6.2 Description

The project for this hands-on has been developed for the STM32F4DISCOVERY board. The code can be found in `code/multifunction/animatronic`. The program `animatronic` implements a master program to control the robot body to perform an animation stored in a lookup table. First, `animatronic` initialises the connection with the servomotors. Then, it starts a single task that sends a goal position to each servomotor every 20 milliseconds using the RS-485 protocol.

The task is created with the function `xTaskCreate`, from FreeRTOS, and it is awakened every 20 milliseconds using the function `xTaskDelayUntil`.

The `README.md` located in the `animatronic` project can be consulted for more details.

## 7.7 Jump mechanism implementation with libopencm3

```

1 #include <libopencm3/cm3/cortex.h>
2 #include <libopencm3/cm3/nvic.h>
3 #include <libopencm3/cm3/scb.h>
4 #include <stdint.h>
5 void start_application(uint32_t application_vector_table_address)
6 {
7     // Clear all pending interrupt requests in NVIC.
8     cm_disable_interrupts();
9     for (uint8_t i = 0; i < NVIC_IRQ_COUNT; i++)
10     {
11         nvic_clear_pending_irq(i);
12     }
13
14     // Load the vector table address into VTOR.
15     SCB_VTOR = application_vector_table_address;
16     __asm__ volatile("DSB");
17
18     // Set the MSP with the value provided by the application vector table.
19     __asm__ volatile("MSR MSP, %0" :: "r" (*(uint32_t *)SCB_VTOR));
20     __asm__ volatile("ISB");
21
22     // Set privileged access
23     uint32_t control_value = 0;
24     __asm__ volatile("MSR CONTROL, %0" :: "r" (control_value));
25     __asm__ volatile("ISB");
26
27     // Jump to application reset handler, will never return.
28     cm_enable_interrupts();
29     void (*application)(void) = (void (*)(void))((*(volatile uint32_t*)(
30 application_vector_table_address + 4))+ offset);
31     application();
32
33     while(1);
34 }

```

Listing 7.1: Start code ( Adapted from: [ARM documentation](#) )



## 7.8 Bootloader illustrations (version 0)

Legend:

**BLUE** text = flag stored in FLASH memory

**ORANGE** text = flag stored in RAM memory

**SOFTWARE\_VERSION** and **SOFTWARE\_ADDR** denote either the version and flash address of the main bootloader or the application.

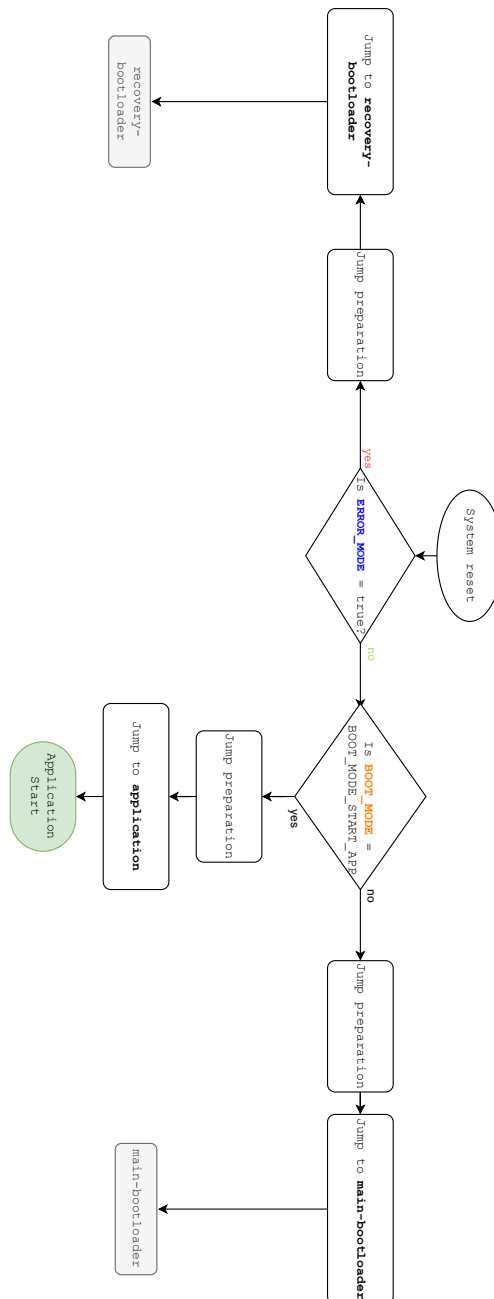


Figure 7.2: Flowchart of the start bootloader

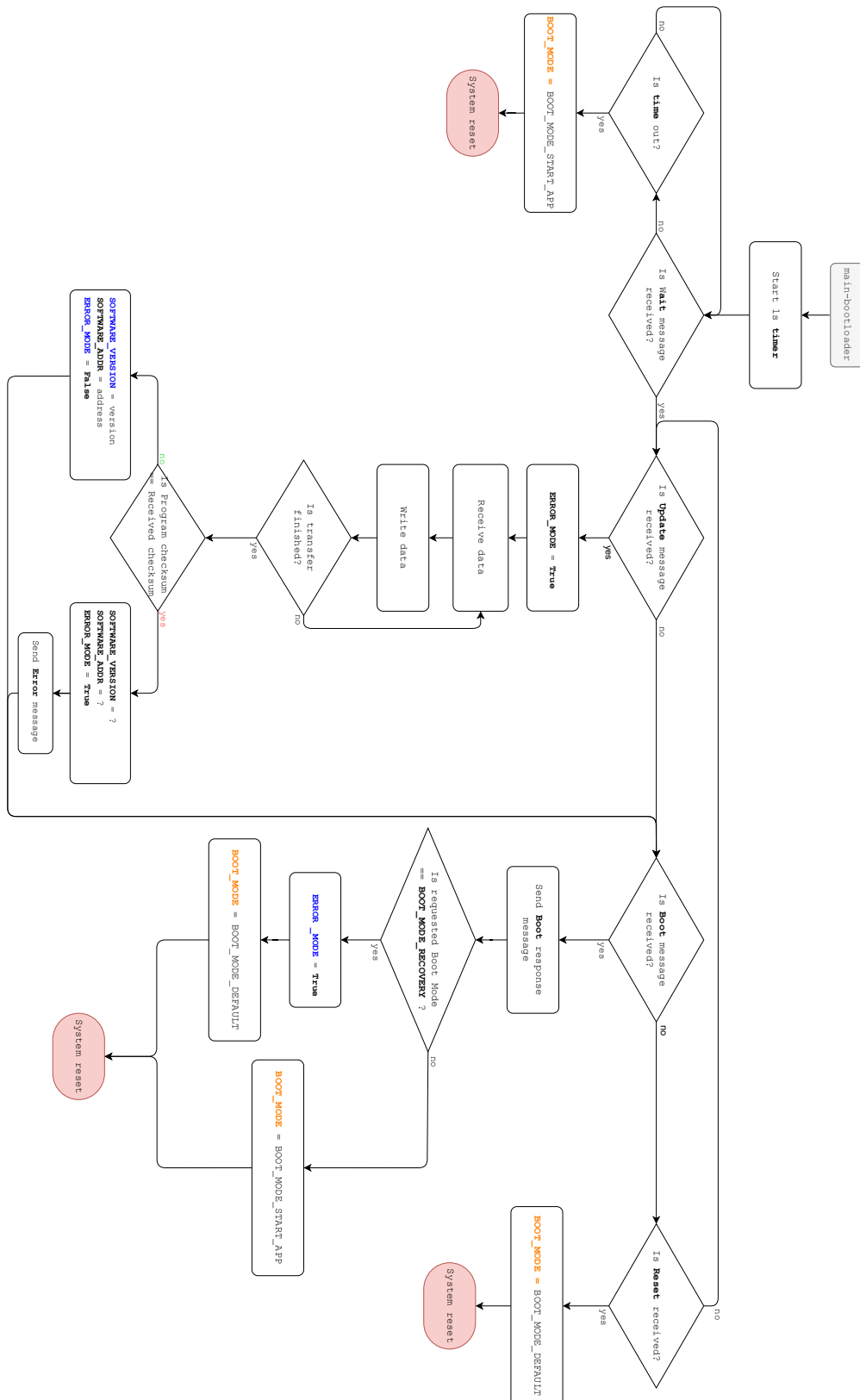
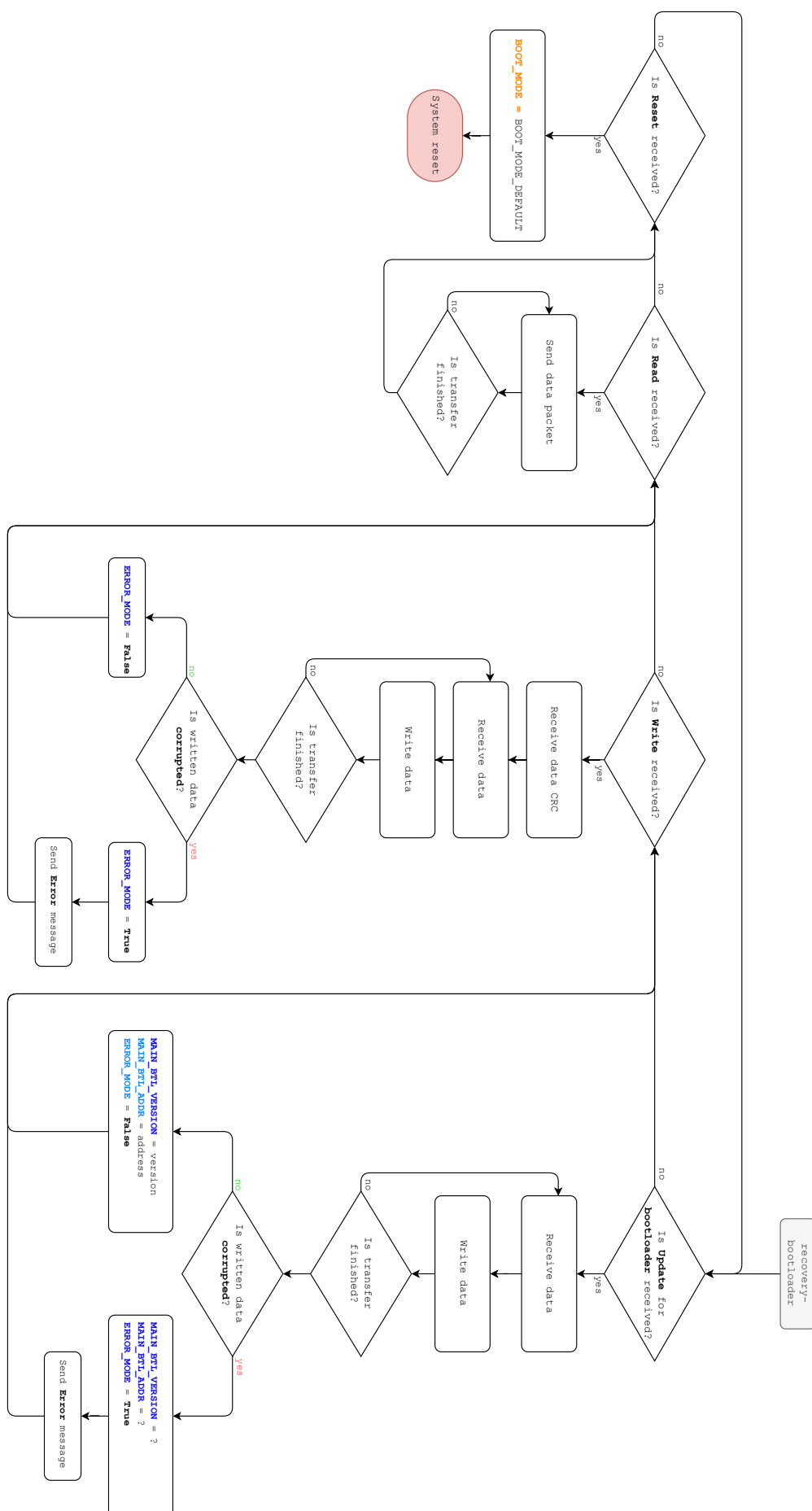


Figure 7.3: Flowchart of the main bootloader



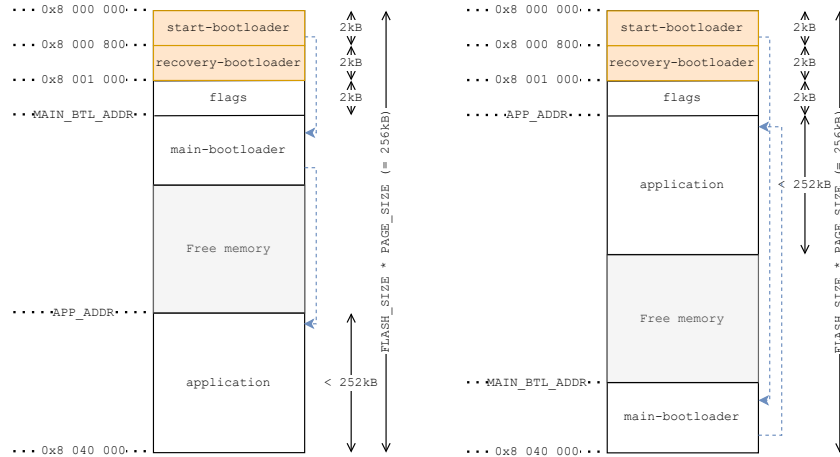


Figure 7.5: The 2 possible memory configurations in version 0.

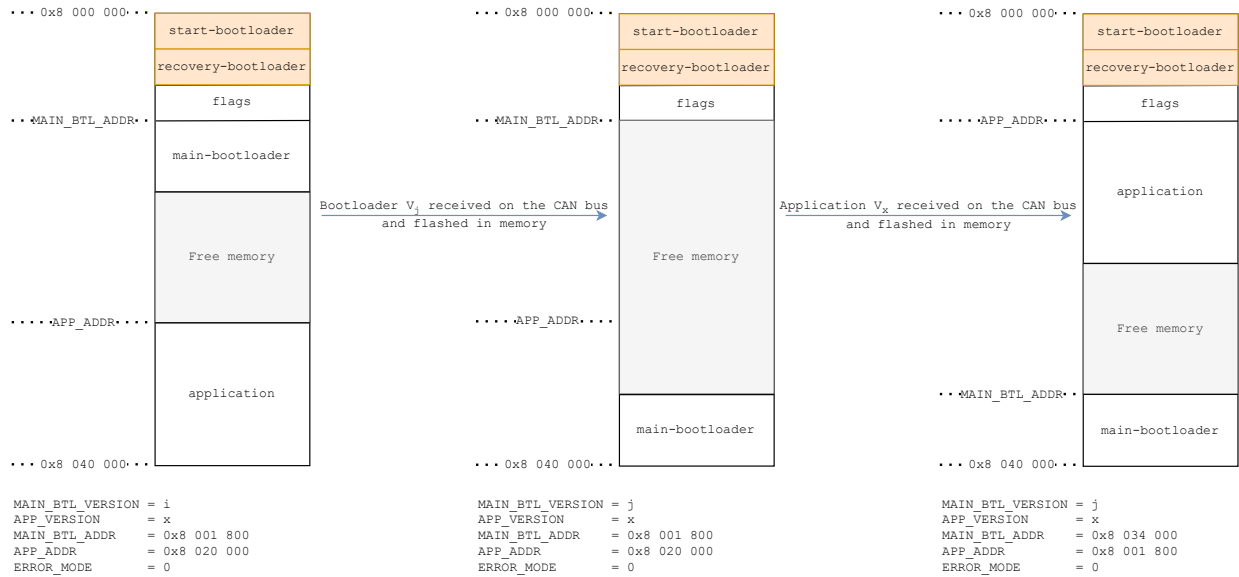


Figure 7.6: Main bootloader update sequence in version 0.

## 7.9 Bootloader illustrations (version 1)

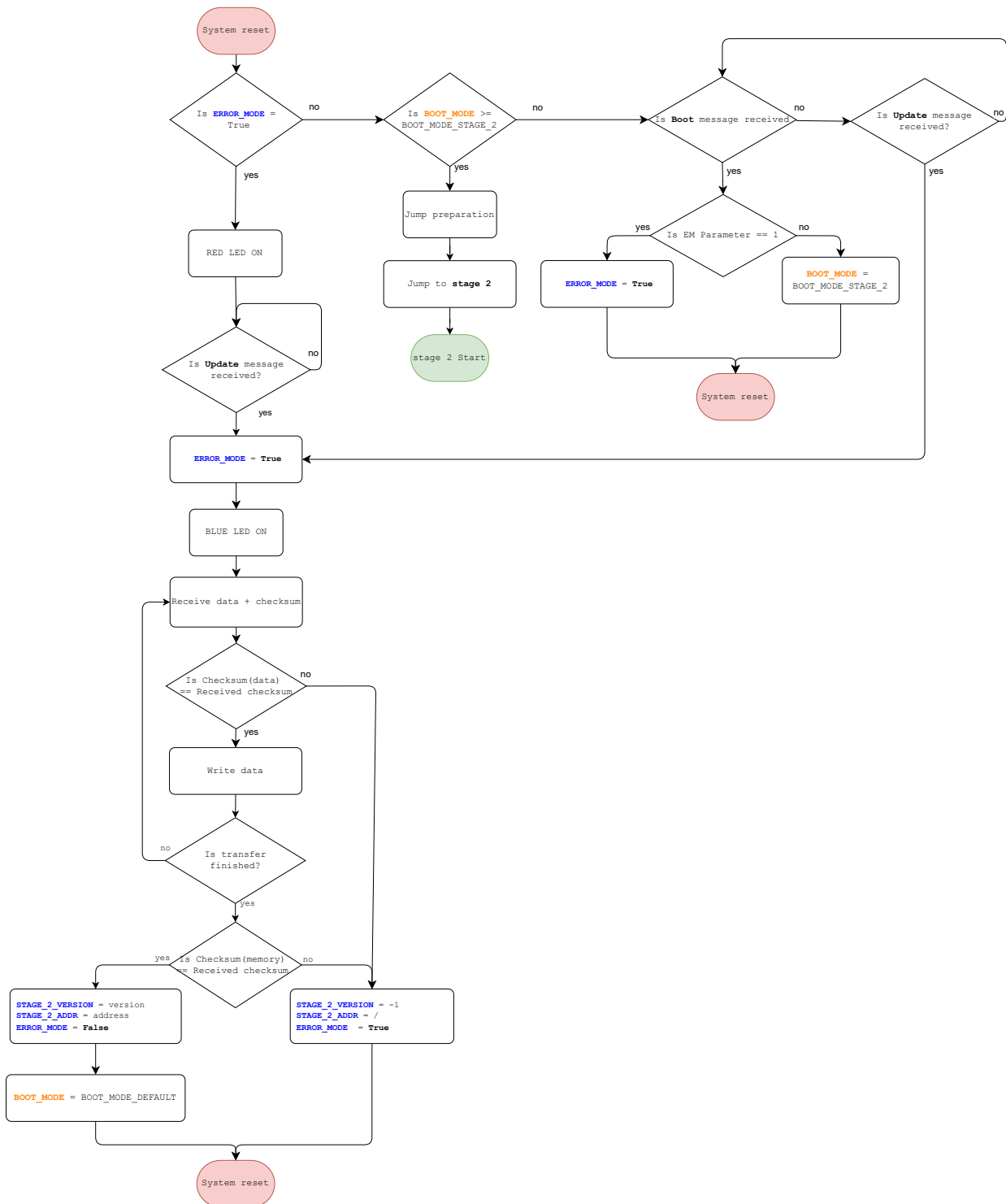


Figure 7.7: Flowchart of the Stage 1 bootloader.

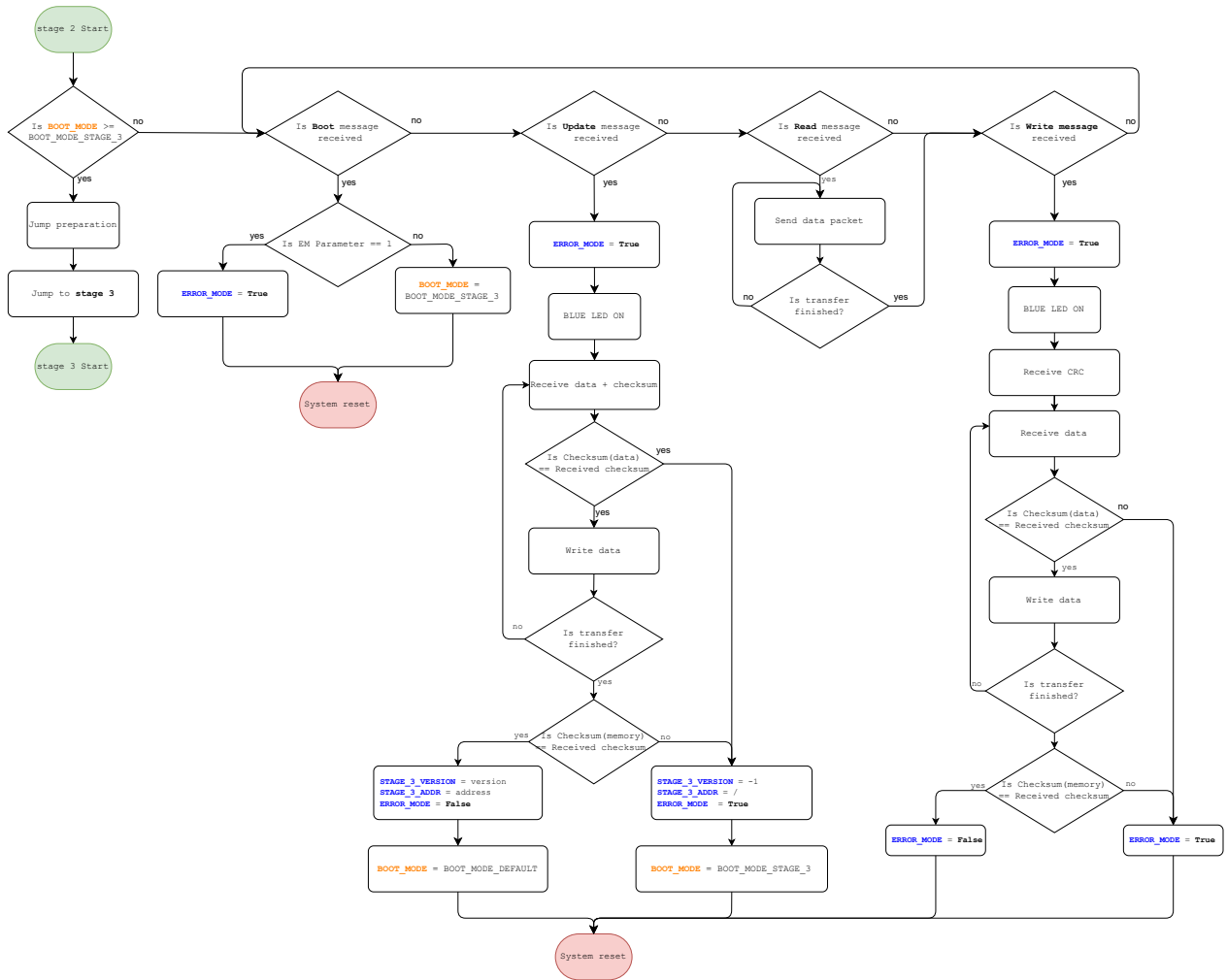
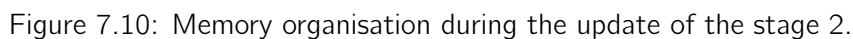
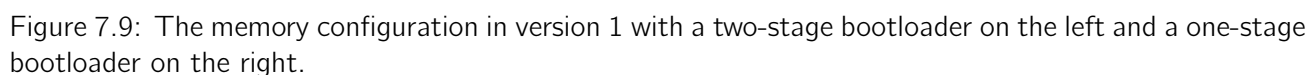


Figure 7.8: Flowchart of the Stage 2 bootloader.



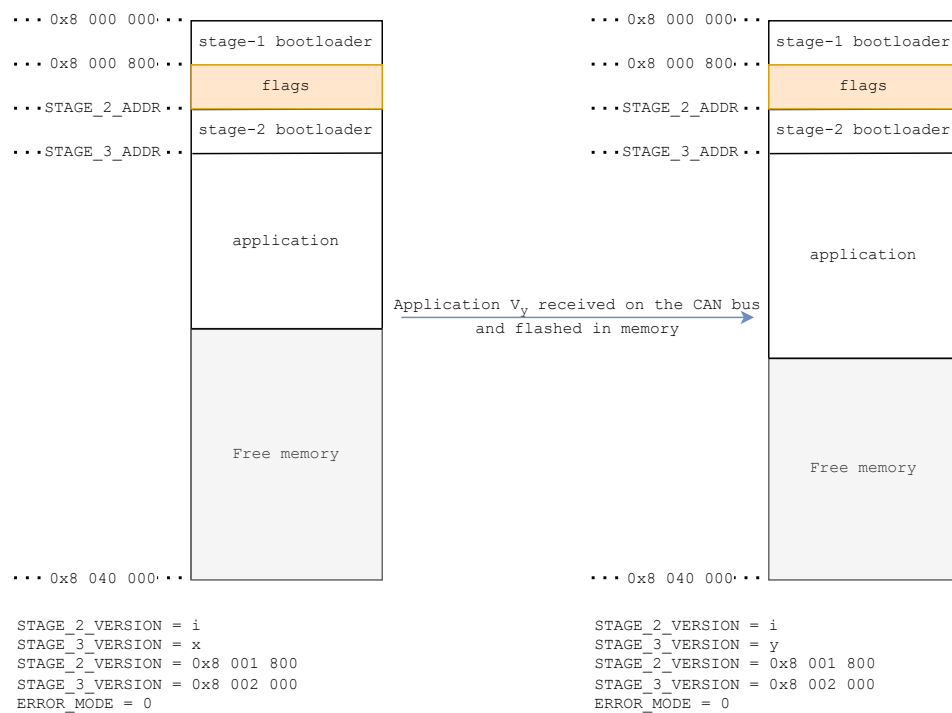


Figure 7.11: Memory organisation during the update of the stage 3.



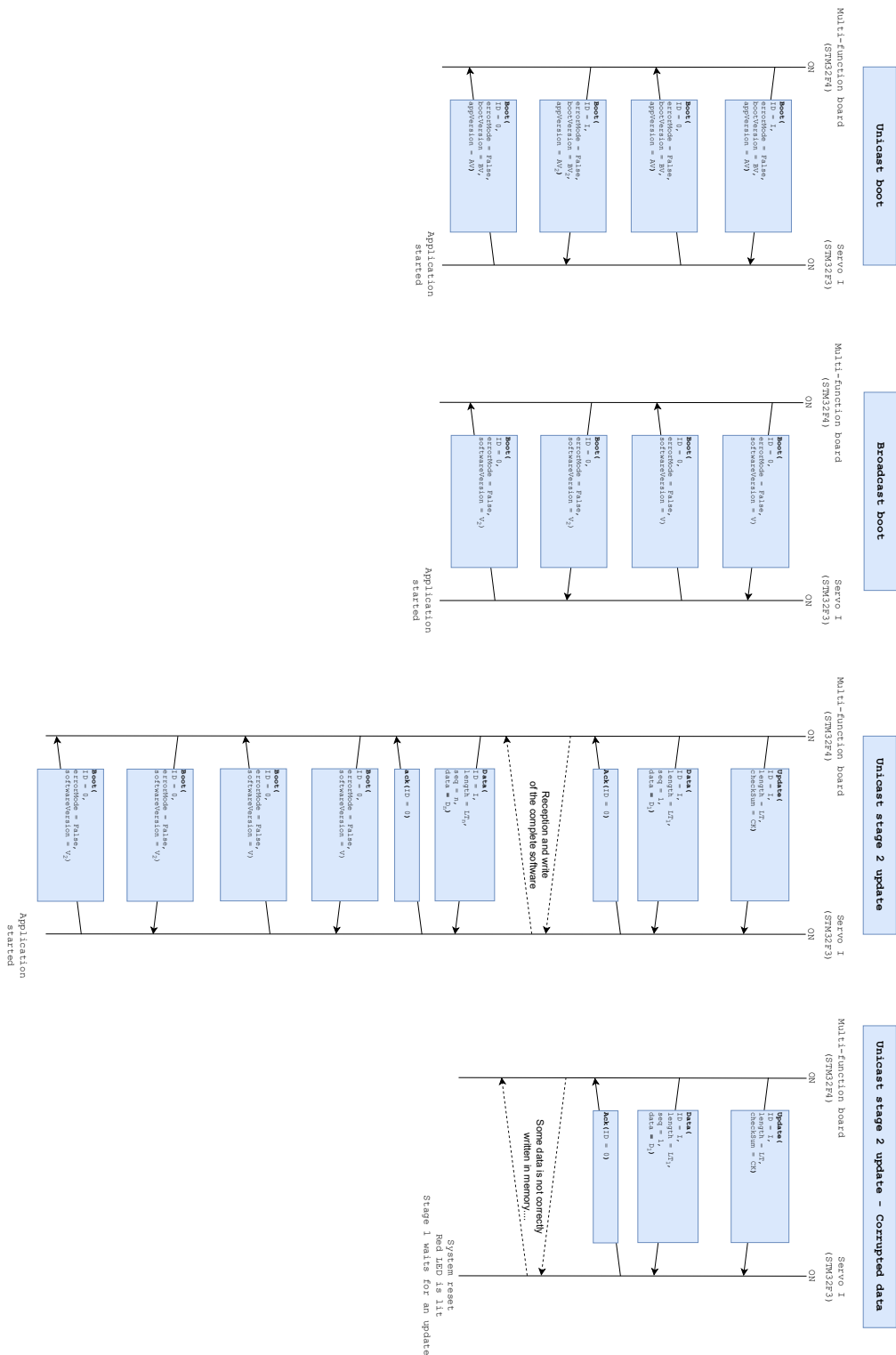


Figure 7.12: Example of communication scenarios.

# Bibliography

- [1] RoboCup Federation : Robocup official website. [https : //www.robocup.org/](https://www.robocup.org/). Accessed: 2025-08-10.
- [2] Wikipedia : Nao (robot). [https : //en.wikipedia.org/wiki/Nao\\_\(robot\)](https://en.wikipedia.org/wiki/Nao_(robot)). Accessed: 2025-08-10.
- [3] RoboCup Federation : Robocup symposium. [https : //www.robocup.org/symposium](https://www.robocup.org/symposium). Accessed: 2025-08-10.
- [4] Gregory Di Carlo : Vision-based robot position estimation. [http : //hdl.handle.net/2268.2/1393](http://hdl.handle.net/2268.2/1393). Accessed: 2025-08-10.
- [5] Guillaume Lempereur : Development of an embedded servomotor controller. [http : //hdl.handle.net/2268.2/1641](http://hdl.handle.net/2268.2/1641). Accessed: 2025-08-10.
- [6] Hubert Woszczyk : Simulation of complex actuators. [http : //hdl.handle.net/2268.2/1446](http://hdl.handle.net/2268.2/1446). Accessed: 2025-08-10.
- [7] Tom Ewbank : Efficient and precise stereoscopic vision for humanoid robots. [http : //hdl.handle.net/2268.2/3144](http://hdl.handle.net/2268.2/3144). Accessed: 2025-08-10.
- [8] Odile Wauquaire : Locomotion control system of a humanoid robot. a biologically inspired model. [http : //hdl.handle.net/2268.2/3169](http://hdl.handle.net/2268.2/3169). Accessed: 2025-08-10.
- [9] Quentin Boileau : Building a simulation environment for biped walking robots using blender. Rapport, Accessed: 2025-08-10.
- [10] Pierre Nicolay : Integration of libopencm3 in freertos for stm32f4 and stm32f3 mcu. Rapport, Accessed: 2025-08-10.
- [11] RoboCup Team, Université de Liège : Structure, 2024. [https : //people.montefiore.uliege.be/robocup/wiki/index.php?n = M%c3%a9canique.Structure](https://people.montefiore.uliege.be/robocup/wiki/index.php?n=M%c3%a9canique.Structure). Accessed: 2025-08-10.
- [12] Maxime Leonard : Design of an internal communication protocol over can bus for humanoid robots. Rapport, Accessed: 2025-08-10.
- [13] Nadir Bounar : Development of the software architecture of a mobile robot. [http : //hdl.handle.net/2268.2/20386](http://hdl.handle.net/2268.2/20386). Accessed: 2025-08-10.
- [14] Aurelien Roekens : Development of a multifunction board for a humanoid robot. Rapport, Accessed: 2025-08-10.
- [15] Simon Gardier, Damiano Pantalone, Balian Franquet et Camille Trinh : Humanoid robot animation generation and execution with blender and freertos, juillet 2025. [https : //people.montefiore.uliege.be/robocup/wiki/files/informatique/animation/animation.pdf](https://people.montefiore.uliege.be/robocup/wiki/files/informatique/animation/animation.pdf). Accessed: 2025-08-10.

- [16] LattePanda Team : LattePanda 3 delta specification. [https : //docs.lattepanda.com/content/3rd\\_delta\\_edition/specification/](https://docs.lattepanda.com/content/3rd_delta_edition/specification/), 2025. Accessed: 2025-08-19.
- [17] Aurélien Roekens : Development of a multifunction board for a humanoid robot. Personal student project (proj0011-1), Université de Liège, 2025. Accessed: 2025-08-19.
- [18] ROBOTIS : Dynamixel mx-28 manual. [https : //emanual.robotis.com/docs/en/dxl/mx/mx – 28/](https://emanual.robotis.com/docs/en/dxl/mx/mx-28/), 2025. Accessed: 2025-08-19.
- [19] Arm Limited : Arm GNU toolchain. [https : //developer.arm.com/downloads/ – /gnu – rm](https://developer.arm.com/downloads/-/gnu-rm), 2025. Accessed: 2025-08-10.
- [20] OpenOCD Developers : Open on-chip debugger. [https : //openocd.org/](https://openocd.org/), 2023. Version 0.12.0 released on 3 March 2023. Accessed: 2025-08-10.
- [21] ARM Limited : *Cortex-M4 Technical Reference Manual, Revision r0p0*, 2010. ARM DDI 0439B.
- [22] STMicroelectronics : *RM0316: STM32F303xB/C/D/E, STM32F303x6/8, STM32F328x8, STM32F358xC, STM32F398xE Advanced Arm®-based MCUs Reference Manual*, 2024. Rev 6, 19-Feb-2024.
- [23] STMicroelectronics : *PM0214: STM32 Cortex®-M4 MCUs and MPUs Programming Manual*, 2024. Rev 7, 18-Mar-2024.
- [24] GDB Developers (Free Software Foundation) : Gdb: The gnu project debugger. [https : //www.sourceware.org/gdb/](https://www.sourceware.org/gdb/), 2025. Latest release (version 16.3) announced April 20, 2025. Accessed: 2025-08-10.
- [25] libopencm3 Project : libopencm3 examples. [https : //github.com/libopencm3/libopencm3 – examples](https://github.com/libopencm3/libopencm3-examples), 2025. Collection of example projects for the libopencm3 firmware library for ARM Cortex-M microcontrollers. Accessed: 2025-08-10.
- [26] Warren Gay : *Beginning STM32: Developing with FreeRTOS, libopencm3 and GCC*. Technology in Action. Apress, New York, NY, USA, 2018.
- [27] Amazon Web Services : Freertos™: Real-time operating system for microcontrollers and small microprocessors. [https : //www.freertos.org/](https://www.freertos.org/), 2025. Accessed: 2025-08-11.