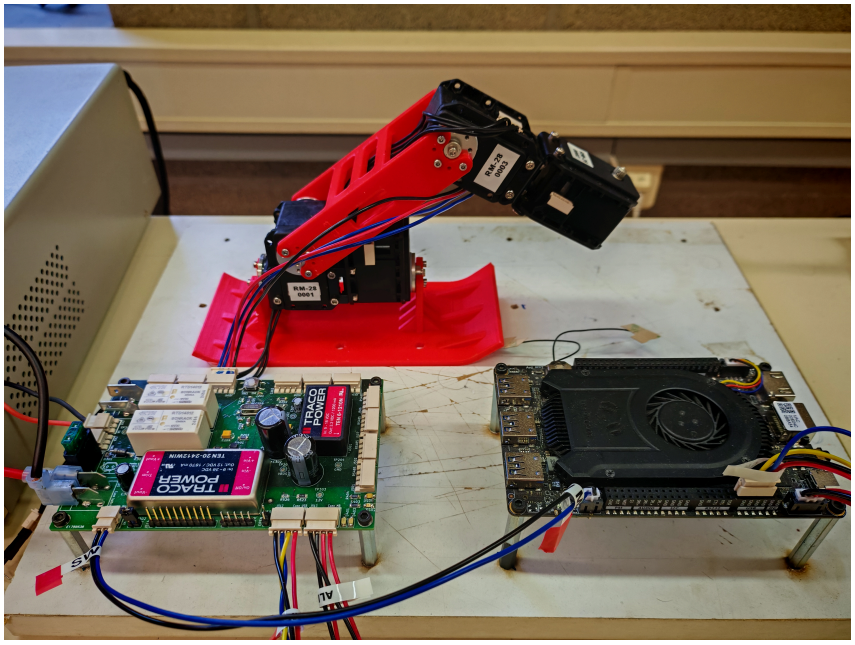
Master thesis : Biologically-inspired architecture for a humanoid robot: from low-level hardware to elementary control

Auteur : Roekens, Aurélien
Promoteur(s) : Boigelot, Bernard
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"
Année académique : 2025-2026
URI/URL : http://hdl.handle.net/2268.2/26152

LIÈGE université



Biologically-inspired architecture for a humanoid robot: from low-level hardware to elementary control

Author:
ROEKENS Aurélien

Faculty of Applied Sciences

Thesis presented to obtain the degree of :
MSc. in Computer Science and Engineering

Thesis supervisor :
BOIGELOT Bernard

Academic year: **2025 – 2026**

Abstract

The Montefiore Team aims to participate in the KidSize category of the RoboCup competition, in which humanoid robots play soccer matches. Achieving this challenge requires a robust and fully integrated platform capable of human-inspired locomotion.

This thesis focuses on the development of every layer of a custom-made robot, following a bottom-up approach. At the hardware-level, we completed two custom electronic boards: the RM-28 servomotor board integrating motor control, various sensors and a CAN channel, and the multifunction board distributing the power throughout the robot and orchestrating a periodic CAN cycle for the communication with the servomotors, as well as a USB connection with the motherboard.

The software embedded in the servomotors includes three elementary PID controllers to drive the motors in position, velocity, and torque, which were validated in a real environment demonstrating good reference tracking.

The logic of the motherboard is organised into software units that handle state management, data processing, and reflex execution. A bio-inspired locomotion system is developed with control units, similar to human neurons, which are stimulated and inhibited to produce human-like motion.

Based on an experiment in a real physical environment, the elementary control of the servomotors effectively reproduced the human adaptation to the ground. This validates that all developed layers operate correctly together as an integrated system.

This thesis provides a solid and operational foundation on which future contributors can build toward the long-term goal of having a custom-made robot play soccer matches.

Acknowledgments

First of all, I would like to express my deep gratitude to Professor Boigelot for his precious help, multiple proofreading, and constant availability during all my participation in the Montefiore Team. I am sincerely grateful for that. This thesis represented a great opportunity to work together. It greatly inspired me and I will carry all the valuable knowledge acquired with me in the future.

I would like to thank the whole Montefiore Team, as the project has been a significant part of my life for three years, through weekly meetings and many hours spent in our laboratory. It was a pleasure to share these three years with you and I am looking forward to seeing you in the future, for the project or not.

Thanks also to my father, who kindly agreed to proofread this thesis and gave me valuable writing advice.

Finally, I want to thank all the people close to my heart who share my life. My studies and daily life would have been less pleasant without their presence and support.

Use of generative Artificial Intelligence

I hereby state that the use of AI tools was strictly restricted to the permitted use following the "ULiège Charter for the use of generative artificial intelligence in academic work". In other words, they helped me during the writing process of this thesis for the formulation, formatting and translation of texts initially written by me. Moreover, AI tools like Claude were used as search assistants to provide support for LaTeX formatting.

The technical contributions of this paper, as well as the analyses, interpretations and conclusions presented are my sole responsibility. All external sources and contributions are properly cited when applicable.

Contents

List of Figures and Tables

List of Algorithms

List of Acronyms

ADC	**A**nalog **D**igital **C**onverter
CAN	**C**ontroller **A**rea **N**etwork
CCW	**C**ounter-**C**lock**w**ise
CDC	**C**ommunications **D**evice **C**lass
CM	**C**ommunication **M**ode
CPU	**C**entral **P**rocessing **U**nit
CRC	**C**yclic **R**edundancy **C**heck
CW	**C**lock**w**ise
DB	**D**aughter **B**oard
FSM	**F**inite-**S**tate **M**achine
HAL	**H**ardware **A**bstraction **L**ayer
I²C	**I**nter-**I**ntegrated **C**ircuit
IPC	**I**nter**p**rocess **C**ommunication
IMU	**I**nertial **M**easurement **U**nit
LED	**L**ight-**E**mitting **D**iode
LSB	**L**east **S**ignificant **B**it
MB	**M**other**b**oard
MCU	**M**icro**C**ontroller **U**nit
MF	**M**ulti**f**unction
MT	**M**essage **T**ype
NTC	**N**egative**T**emperature **C**oefficient
PCB	**P**rinted **C**ircuit **B**oard
PID	**P**roportional **I**ntegral **D**erivative
PWM	**P**ulse-**W**idth **M**odulation
RTOS	**R**eal-**T**ime **O**perating **S**ystem
SPG	**S**pinal **P**attern **G**enerator
SRAM	**S**tatic **R**andom **A**ccess **M**emory
USB	**U**niversal **S**erial **B**us
ZMP	**Z**ero **M**oment **P**oint

Introduction

RoboCup [7] is an international competition whose objective is to promote robotics and research, in which teams of robots face each other in soccer matches. The ultimate goal of the RoboCup initiative is the following: "*By the middle of the 21st century, a team of fully autonomous humanoid robot players shall win a soccer game, complying with the official rules of FIFA, against the winner of the most recent World Cup*". The competition constrains the robots to the same restrictions as humans. Hence, robots must walk using two legs, see with eyes, and perform human-like actions. The objective of the Montefiore Team is to participate in the KidSize category, involving small robots. Unlike many other teams, our project is developed from scratch by students of the university through master's theses, personal projects, or voluntary participation.

The Montefiore Team, responsible for developing our robotic platform, is divided into three sub teams:

- Mechanics, responsible for the physical structure, mainly 3D-printed, of the robot.
- Electronics, designing and building electronic boards.
- Informatics, developing low-level code running on electronic boards and high-level logic such as strategy, vision, and motion planning.

The goal of this thesis is to design a first version of the platform capable of simulating human reflex based on elementary control of actuators by integrating, adapting, enhancing and correcting, if needed, past contributions.

State of the art

In the end of the 20th century, the first human-like walking robots emerged, such as the P2 robot developed by Honda [8] in 1996. Honda concluded at that time that robots should have an orientation sensor and a ground reaction force sensor [8]. They also achieved dynamic walking using the Zero Moment Point (ZMP) method [9]. Takubo et al. then improved ZMP to solve rough terrain walking [10]. Kajita et al. developed a single linear inverted pendulum with ZMP delay [11]. More recently, in 2025, Cho et al. [12] tackled motion generation, robust to disturbances by minimising angular momentum. Their solution is based on the mathematical models developed over the past few years. We can see the trend of the robotics community, oriented toward model-based control methods, which are mostly mathematically driven, rather than bio-inspired.

Towards a more bio-inspired point of view

These model-based methods have the drawback of being strongly model dependent and not robust enough to handle unknowns. Indeed, a robot's control depends on the dynamic models of the robot and the

environment, and thus on the accuracy of those models. For instance, if the modelled environment supposes a flat surface but the robot is walking on uneven terrain, it can lead to instability. Moreover, adaptation to external forces is a difficult problem for model-based methods [6]. In our case, we want our control to be really robust as a soccer game is a tough environment. For instance, the robot can be pushed or can collide with an opponent. The terrain is supposed to be flat, but the robot can fall on another robot. It is impossible to model such unpredictable configurations, thus we need methods robust to these unplanned situations.

Part of the scientific community developed biologically inspired bipedal locomotion to improve model-based methods. Their main hypothesis was that human and bird locomotion has been optimised by natural selection to be robust, fast, and energy efficient. Their approaches can be divided into three types [13]:

- Human Biped Walking Robots (HBWR),
- Bird Biped Walking Robots (BBWR),
- Synthetic Biped Walking Robots (SBWR), based on heuristic and synthetic ideas.

We will focus on the HBWR methods due to the RoboCup scope. The human musculoskeletal system was analysed in a 300 degree of freedom model [14]. Thus, this system is really hard to fully model accurately, and the researchers simplified the musculoskeletal system into more elementary models [15], [16]. The development of control methods relying on human control locomotion has driven the work of Tobias Luksch [6]. In his PhD's thesis, he designed a control system based on local feedforward and feedback units. The coordination is handled by sensor-based events, and torque commands are applied resulting in a highly robust system against unknowns and disturbances.

However, many problems arise from biologically inspired methods, as summarised by Mikolajczyk et al. [13]:

- Maintaining a stable and controlled bipedal gait across diverse environments,
- Designing transitions between different locomotion modes (standing, walking, running, etc.), accurate enough so that the robot does not fall [17],
- Processing low-level signals into high-level orders for joint activation or other actions, and
- Adaptation to non static surfaces, such as moving ground or slippery surfaces.

No one-size-fits-all solution to these problems is currently known.

Machine Learning methods

The breakthroughs of artificial intelligence (AI) during the last decade provide many more possibilities for robotics development. We can cite some research papers related to the scope of this master's thesis. Shahna et al. [18] integrated a collision-free trajectory planner based on Deep Reinforcement Learning by auto-tuning the low-level control. Li et al. [19] introduced a hierarchical reinforcement learning framework to achieve optimal motion for quadruped robots by providing optimal parameters to the low-level controller. These two papers provide good examples of the use of machine learning models as parameter optimisers. Machine learning is also useful for more high-level tasks. Nguyen et al. [20] provided a comprehensive overview of recent deep reinforcement learning applied to path planning tasks.

Our project does not rely on AI for the moment. The goal of this thesis is to design and implement an architecture which, among other things, includes low-level software for the electronic boards to process

data. Thus, one can see this work as the design of a tool box which can be used by a machine learning model to enhance the control by tuning the parameters or to develop higher-level tasks.

State of the Bobby platform

The contribution of multiple students over the years led to the ongoing Bobby platform. Its mechanical structure is home-made and mainly 3D printed by our mechanical team. The motherboard of an embedded computer, roughly equivalent in performance to a small laptop, performs the high-level computer operations. One can visualise the motherboard as the brain of the robot. Bobby is equipped with 22 servomotors that act as the joints of the robot. A custom board, called multifunction board (MF), is responsible for exchanging messages between the motherboard and the servomotors. It acts as the spine of the robot. The MF board is also responsible for distributing power throughout the robot, which is powered with a removable, custom five-cell lithium-polymer battery pack [21].

Over time, the Montefiore Team's members wrote multiple documents, starting in the academic year 2015-2016 with three master's theses:

- Grégory Di Carlo : *Vision-based robot position estimation* [22],
- Guillaume Lempereur : *Development of an embedded servomotor controller* [3],
- and Hubert Woszczyk : *Simulation of complex actuators* [23].

To fully understand this work, it is important to detail the master's thesis of Guillaume Lempereur. Before his work, the Montefiore team was using servomotors manufactured by Robotis: the dynamixel MX-28 [24]. A servomotor is composed of a controllable motor, multiple sensors to measure various quantities such as the motor's angular position for instance, and a communication channel used to exchange information with the external world. Finally, a microcontroller (MCU) manages each component of the servomotor.

At some point, the need for more powerful and modular servomotors arose, resulting in the development of a new internal electronic board for them. The most important modifications were the add of an efficient torque measurement sensor, the replacement of the communication protocol by a more robust protocol, and the incorporation of new sensors such as an accelerometer and a gyroscope. The new servomotors are equipped with an improved microcontroller, which performs powerful local computations and controls the servomotors in position (i.e. reach a specific angular position), in velocity (i.e. rotate with a defined angular speed), and in torque (i.e. maintain a given force). Some improvements were, however, still possible and Lisa Infantino tackled this challenge with the specific goal of designing a better torque measurement tool. She works on this issue as part of her personal research project, in parallel to this thesis [25].

Following these three master's theses, other documents have followed:

- Tom Ewbank's master's thesis in 2016-2017: *Efficient and precise stereoscopic vision for humanoid robots* [26],
- Odile Wauquaire's master's thesis in 2016-2017: *Locomotion control system of a humanoid robot - A biologically inspired model* [27],
- Quentin Boileau's personal research project in 2016-2017: *Building a simulation environment for biped walking robots using Blender* [28], and
- Pierre Nicolay's personal research project in 2018-2019: *Integration of Libopencm3 in FreeRTOS for STM32F4 and STM32F3 MCU* [29].

Odile Wauquaire was looking for a robust and efficient locomotion system for a robot. She challenged multiple locomotion methods before choosing the ones of Tobias Luksch [6]. After that, she simulated the locomotion system using Blender in order to evaluate, compare, and validate these methods.

The personal research project of Pierre Nicolay was focused on the embedded architecture of the microcontrollers used in the project. In our project, the MCU manages multiple tasks in parallel. For instance, one task can be responsible for computing the next action to take, while another for measuring some data. To ensure good reactivity, we want to minimise the latency, i.e. the delay between the decision to take an action, and the execution of this action. A real-time operating system (RTOS) minimises latency by dividing tasks by priority. High-priority tasks execute first, so that low-priority tasks do not delay more important processes. Pierre Nicolay chose FreeRTOS as the RTOS for our custom electronic boards. He also chose to use the LibOpenCM3 library [30] for the Hardware Abstraction Layer (HAL) on the STM32 microcontrollers.

After a temporary pause due to the Covid-19 pandemic, the Montefiore RoboCup project resumed, leading in 2023-2024 to the thesis of Nadir Bounar: *Development of the software architecture of a mobile robot* [31], and the personal research project of Maxime Léonard: *Design of an internal communication protocol over can bus for humanoid robots* [32]. The thesis of Nadir Bounar was mostly focused on the development of a real-time platform which minimises the latency between the robot's motherboard (MB) and the multifunction board. The multifunction board, also called MF board is, among other things, responsible for:

- Powering the motherboard, and communicating with the MB using a USB connection, and
- Exchanging data with the daughter boards of the robot, the servomotors for instance, through a communication channel using the protocol of Maxime Léonard [32].

Multiple participants of the Montefiore team developed the MF board, among those I was the one who finalised it in 2024-2025 during my personal research project: *Development of a multifunction board for a humanoid robot* [33]. In the same year, Simon Gardier developed a software to program the servomotors' microcontroller remotely during his personal research project: *Development of a Bootloader for the STM32F3 Plateform* [34]. Finally, Loris Degueldre is currently working, as part of his master's thesis, on the development of a new electronic board responsible for the charge and discharge of the battery [35]. His ultimate goal is to integrate this board with the current multifunction board.

Comparison with the state of the art

Since the RoboCup competition's intention is to promote research, the documentation of participating robots has to be fully accessible.

When taking a look at the winner of the 2025 international RoboCup competition: Boosted-HTWK [36] which uses the industrial Booster-K1 robot [37], or the one of the Germany Major: ZJUDancer [38], we can see that both teams use only one inertial measurement unit for the whole robot.

An inertial measurement unit (IMU) is composed of an accelerometer and a gyroscope, which respectively measures the linear acceleration and angular velocity. This sensor is used, among other things, to estimate the proprioception of the robot. However, having a single IMU assumes that the structure remains constant while moving. This hypothesis is usually not fulfilled while walking, let alone running. Indeed, the mechanical structure of the robot is not fully rigid and the external forces applied on the moving robot deform the body limbs. Each bent limb induces a small offset between its expected position and the actual one. When the offsets accumulate, across a whole leg for instance, the end position can significantly deviate from its expected location.

Unlike their work, each servomotor of our robot is equipped with an individual IMU, enabling accurate knowledge of the orientation and acceleration of each joint of the robot. Combined with the multi-IMU state estimation developed by Xavier et al. [39], our robot's proprioception will be robust to structural deformations. This example highlights the usual assumption in robotics: joints are not able to measure accurate data, nor perform powerful computation.

To continue the comparison with former winners of RoboCup:

- Unlike Boosted-HTWK, our team fully designed our robot, thus we have a very high control on all hardware and software aspects. A concrete example is that their team depends on industrial manufacturers in case of material damage. On the other hand, we are way more independent: if the arm of our robot breaks, we can easily 3D-print another copy.
- ZJUDancer is also mechanically custom made, based on unmodified DYNAMIXEL servomotors. Since these servomotors are designed to be used for various types of applications, ZJUDancer's servomotors are less tailored for the specific use of RoboCup. As presented earlier, our custom servomotors give us more flexibility and possibilities.

Our team also developed its own framework and architecture that does not rely on the dominant framework in robotic: ROS2 [40], which is less tailored for our needs.

The analysis of the state of the art by the Montefiore Team drew two conclusions:

1. As presented earlier, model-based control methods, which are mostly mathematically driven, are too dependent on the model and not robust enough to unknown events. The biological inspired methods present a bigger potential, even though they bring a series of problems which need to be tackled.
2. Most of the research does not go into detail of the low-level hardware and focuses on the high-level solutions. Many proposed methods are tested using simulations, that never fully represent the real world. In case of physical robots, its hardware specifications are rarely fully detailed, or the robot is commercially made, which lowers the flexibility and modularity.

To conclude, the Bobby platform, by its fully home-made design, from the low-level hardware to mechanical structure and high-level processing, provides us bigger flexibility, modularity, and greater potential compared to the other competitors of RoboCup.

Objectives of this work

The long-term objective of the Montefiore Team is to build a robot that is capable of playing soccer. The RoboCup project at Montefiore is now at a key point: over the years, many students have contributed to the project and achieved significant milestones. Each contribution was done with the project's long term goal in mind, but remains independent from the others, with few interactions between all parts.

Therefore, the first challenge addressed by this work is the integration of all existing contributions in order to build a first complete version of the Bobby platform. This challenge is tackled in collaboration with the Montefiore Team and this thesis focuses on my contributions.

As explained above, O. Wauquaire incorporated the locomotion system of T. Luksch in a simulated environment, which we needed to connect to the physical Bobby platform. Multiple problems are encountered when transposing a theoretical or simulated environment into a real robot. Some questions are harder to answer in real situations, like "How to reproduce anatomical constraints of the human body?", "How to determine the direction of rotation of the servomotors?" or "How to determine if the robot has touched the ground ?". We identified open questions, such as how to compute the orientation of the servomotors, and proposed solutions for them. Addressing these questions constitutes the second challenge for this work.

The milestone of this thesis is the development of a Bobby platform capable of reproducing human reflexes through elementary controllers implemented on the servomotors. It requires multiple intermediate steps, detailed in each chapter of this thesis:

- Conception of robust and reliable hardware to fit the current requirement of the project (cf. Chapter 1).
- Development of low-level software running on the servomotors in order to interface the sensors and communicate with the MF board (cf. Chapter 2).
- Design of a global communication logic across the robot, from the high-level motherboard to the low-level daughter boards. The conductor of this logic is the multifunction board, which has its own intermediate-level software (cf. Chapter 3).
- Design of a high-level software layer using the lower-level layers. This layer provides future developers with a solid base on which more complex algorithms can be created (cf. Chapter 4).
- Implementation of efficient controllers, based on the low-level software, to control the servomotors in position, velocity, and torque. This is crucial in order to build a robust system capable of standing, walking, and running (cf. Chapter 5).
- Transposition of bio-inspired locomotion methods into our own robotic platform, based on each developed layer and reproduction of a human-like reflex (cf. Chapter 6).

This report thus follows a bottom-up approach where we build the robot's software layer by layer, from the hardware-electronic to the high-level logic. Besides, what we refer with low- and high-level in this document is highly dependent on the perspective. From the point of view of the hardware design, everything that we implement is high-level, even interfacing the sensors. On the other hand, the highest-level layer developed in this project corresponds to the lowest-layer of the software running the motherboard, as our goal is to build a platform upon which developers can create more complex algorithms.

Chapter 1

Hardware

This chapter details the low-level hardware composing the robot. First, we present an overview of the robot before detailing the functionalities of the custom electronic boards, namely the servomotors' board and the multifunction board, and their software architecture used. Finally, we describe the test bench created for the project.

1.1 Overview

Bobby is equipped with a motherboard (MB) performing all the high-level computing such as image processing, using two cameras connected by USB, motion planning, and robot strategy. Multiple custom servomotors are used to make the robot move. Finally, a custom multifunction board distributes the power throughout the robot, starts the MB remotely and creates a link between the MB's high-level decisions and the low-level actuators of the daughter boards. For now, the only DBs used are the servomotors, but later, we could add others such as a pressure sensor board in the feet. The MF board itself is started using a push button located on the head of the robot.

The Universal Serial Bus (USB) connects a device to a host. There are several classes of devices, among which we use CDC that provides a bidirectional data stream. We chose USB in the project due to its convenience and because it is present on every computer nowadays, including our motherboard. If a better and more convenient serial bus emerges in the next years, we would potentially switch to it.

The multifunction board communicates through two different channels with external devices: USB for the motherboard and CAN for the daughter boards. We give more details about CAN in Chapter 2.2.

The structure of Bobby is illustrated in Figure 1.1.

1.2 Motherboard

The motherboard is the central element of the robot, managing the robot's data, running complex algorithms and taking decisions. It is also the intermediate between the robot and a remote operator. It acts as an embedded computer on which we develop all the code running on the robotic platform.

Based on the requirement of the project, our motherboard is the LattePanda Delta 3 [41]. It features a x86-64 architecture with a processor which supports multithreading. WiFi and Ethernet interfaces are available, as well as three USB ports: one USB 3 and two USB 2.

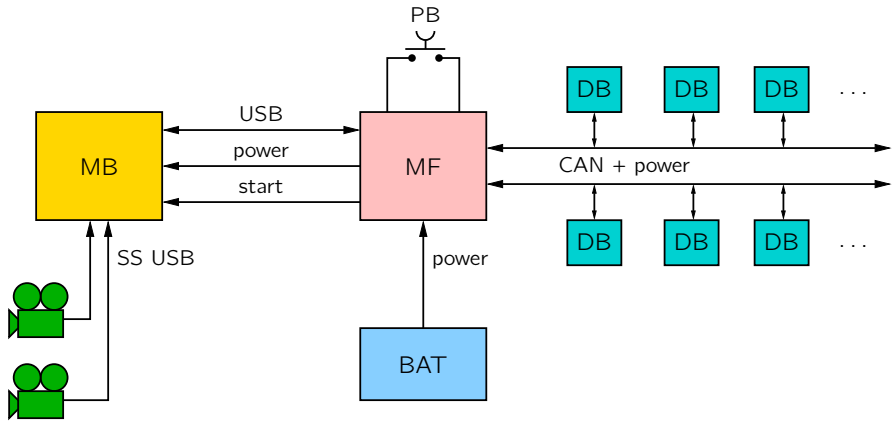


Figure 1.1: The Bobby platform (Source: Montefiore Team [1]).

Once the cameras are operational, we will use the USB 3 port for connecting them, due to its higher data transmission rate. Finally, one of the USB 2 port is used to exchange data with the multifunction board. This data channel is set to be in Full-Speed operating mode which provides a transmission rate of 12 Mbit/s.

1.2.1 Real-Time Operating System

In order to perform the high-level operations presented in Chapter 1.1, the motherboard manages multiple tasks in parallel using multithreading. Since the MB acts as the brain of our robot, it is critical for the MB to be reactive. We use PREEMPT_RT for this purpose.

PREEMPT_RT is an extension of the Linux kernel to transform it into a real-time kernel [42]. It minimises the latency by scheduling tasks based on their priority.

1.3 Custom electronic boards

We developed two custom electronic boards in our project: the multifunction board and the servomotors' electronics.

To do so, we first decide the functionalities of each board, which act as a list of requirements that the board must fulfil. Based on this, we designed the electronics, i.e. reading and understanding the datasheet of the components, selecting the most appropriate one for our needs and integrating each component on the board.

Once this is done, we developed the Printed Circuit Board (PCB) which must fulfil another set of requirements. For instance, the board can have some mechanical constraints to be mounted on the robot. The PCB is then printed by industrial manufacturers. Once the PCB was received, we populated all the components on the board. After that, we performed hardware testing to ensure that the soldering process was done correctly and none of the components were damaged.

We present now the concrete conception of the electronic boards of the project.

1.3.1 Servomotors

The servomotors act as the joints of the robot. They consist in a motor and an electronic board, which controls the motor thanks to a microcontroller, power electronics, and various sensors.

The custom servomotors modify the MX-28 of Dynamixel [24] in order to improve controllability and reliability, as well as to enhance their functionalities, for instance by implementing custom control algorithms.

We named our new version of the board RM-28, for **R**ococup **M**ontefiore, which major functionalities are the following :

- The replacement of the RS-485 communication protocol by the more reliable and robust CAN. Chapter 2.2 provides more details on the CAN protocol and its use in the project.
- The replacement of the microcontroller by a more powerful one, the STM32F303CC [43].
- An inertial measurement unit (IMU), the LSM6DS3H [5], which combines an accelerometer and a gyroscope, and measures the linear acceleration and the angular speed. This improvement is crucial for an efficient estimation the position of the servomotors.
- A temperature sensor, close to the motor, providing a better temperature information than relying on the temperature sensor embedded in the microcontroller.
- A torque estimation, based on the current consumption of the motor, measured by a current amplifier sensor.
- A multicolour LED used to give various information during the working of the robot, the SMLP34 [44]. For instance, the LED blinks green to indicate that the servomotor is ready to start or magenta to indicate that the servomotor has reached his goal.

These modifications are added to the angular position measurement, kept from the MX-28 design. The MX-28's magnetometer determines the angular position of the servomotor using a magnetic sensor which measures the magnetic field induced by a magnet glued to the shaft of the servomotor. This solution has two main advantages:

- We can measure the angular position over a full turn of 360 degrees, i.e. there is no dead zone, and
- The magnetometer is not physically in contact with the shaft and the sensor does not apply any friction force on the motor's axis. Therefore, the shaft wears out less quickly. The magnetometer is illustrated in Figure 1.2.

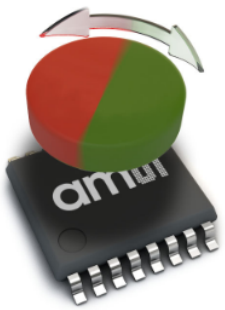


Figure 1.2: Magnetic rotary encoder (Source: AS5045's datasheet [2]).

The RM-28 electronic schematics are shown in Appendix A, in Figures A.1 and A.2

H-Bridge

Controlling the Maxon RE-max 17 DC motor [45] requires understanding the relation between current and torque. Since the torque generated by the motor is proportional to the current flowing through it, controlling the current directly controls the produced torque.

A H-bridge consists of four transistors and can apply the supply voltage across the motor in either direction [3]. It enables bidirectional current flow through the motor, and thus the control of both the direction and magnitude of torque produced. The circuit of a H-bridge is illustrated in Figure 1.3.

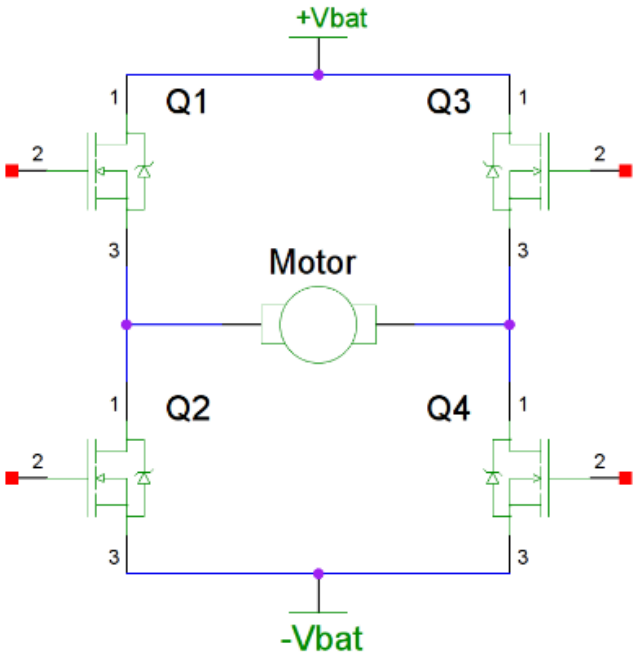


Figure 1.3: Circuit of a H-bridge (Source: [3]).

We describe three possible configurations:

- Q3 and Q2 are switched on. This corresponds to the forward configuration where the high potential of the battery is connected to the right terminal of the motor and the ground to the left terminal.
- Q1 and Q4 are switched on. This corresponds to the reverse configuration, which is the opposite of the forward state.
- Q1 and Q3 (or Q2 and Q4) are switched on. This corresponds to a braking state.

Since the DC motor is an inductive load, it is critical to always provide a path for the current to flow through it. Indeed, during the forward configuration, a large current flows through the motor and if the current path is then disconnected, the inductor tries to maintain the current flow. This increases the voltage's difference at the transistors' terminal, which can damage the components. To avoid this issue, we use the sign-magnitude method which alternates between the forward/reverse state and the braking state. During the braking state, the current circulates through the motor which safely dissipates the energy stored.

The switching of the H-bridge's transistors is controlled by pulse-width modulation (PWM) signals, alternating between high and low at a fixed frequency. The duty cycle, i.e. the percentage of time during which the PWM signal is high, regulates the current flowing through the motor. In our implementation,

the MCU generates two centred PWM signals with a phase shift to the gate drivers of the H-bridge. which doubles the PWM frequency seen by the motor. The situation is illustrated in Figure 1.4.

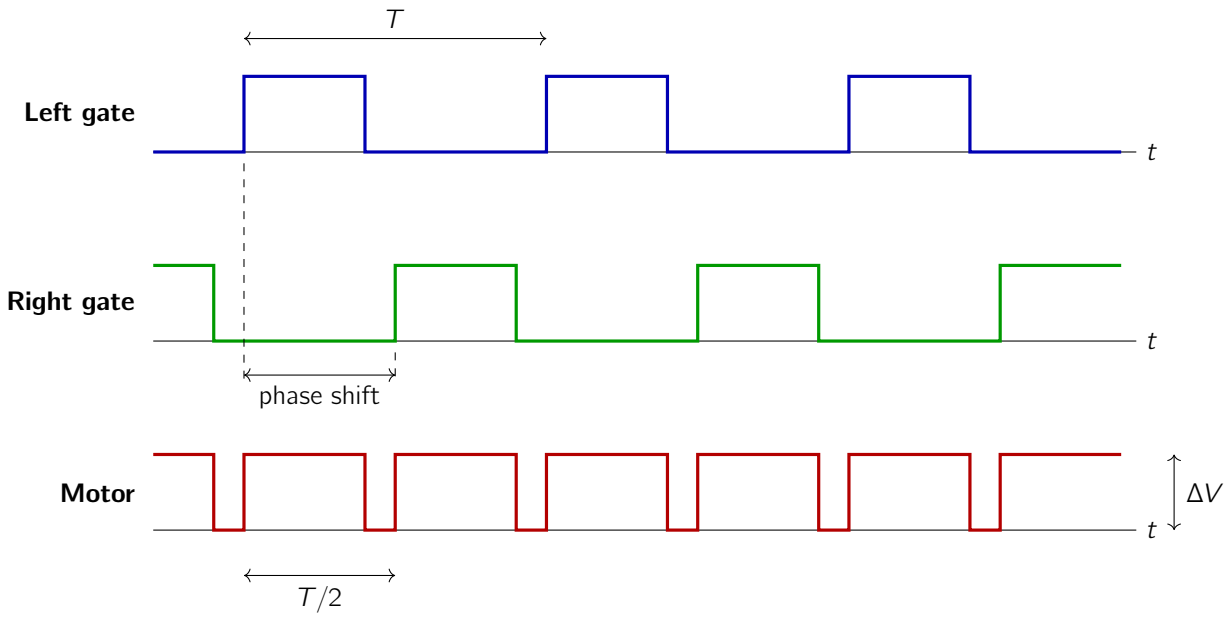


Figure 1.4: Phase-shifted PWM signals on the gate drivers and effective switching frequency of the motor.

The sign-magnitude method has two advantages. First, since the dissipation of the current is evenly distributed across the transistors, the transistors heat up uniformly. Second, we generate a lower frequency PWM signal to switch on the transistors, which reduces the switching losses. By phase-shifting the centred PWM signals, the frequency of the PWM signal applied on the motor is twice the one generated on each transistor, which smooths the current.

However, this method has two drawbacks. Due to the braking period, power is dissipated in the transistors, even when the motor does not produce any torque, which represents a constant current consumption. Moreover, one advantage of the H-bridge is that the battery can be recharged when the motor is driven by an external torque. Due to the sign-magnitude method, however, regeneration is only possible when the H-bridge is not in a braking state therefore limiting the energy recovery.

RM-28's hardware sensors

Chapter 2.5 describes how we obtain the measurement of each sensor, as well as the low-level software developed on RM-28. For now, we give an exhaustive list of the data measured by the sensors:

- The angular position of the motor measured by the magnetometer (AS5045) [2], from a zero reference which is arbitrary defined.
- The angular velocity of the servomotor sensed by the gyroscope embedded in the IMU.
- The servomotor's linear acceleration, including the gravity's influence, measured by the accelerometer embedded in the IMU.
- The torque generated by the motor, estimated using a current amplifier sensor, the INA240 [4].
- The temperature of the microcontroller measured by the integrated temperature sensor of the MCU.

- The temperature of the motor sensed using a negative temperature coefficient (NTC) resistor, whose resistivity decreases with increasing temperature.
- The power supply's voltage measured using analogue to digital converter of the MCU.

PCB design

We completely designed the PCB layout for RM-28, accounting for various constraints:

- The board has to fit mechanically into MX-28 enclosure, which limits the available space. Moreover, we must include two mounting holes of 2 mm of diameter, which allow to screw the board to the plastic enclosure of the servomotor.
- The magnetometer, measuring the angular position of the motor, needs to be precisely aligned with the motor's shaft. A misalignment greater than 250 μm may cause the magnetometer to be unusable. Also, to be usable, this component is placed on the side of the board facing the shaft.
- We must position the multicolour LED in the middle and at the right-most corner of the board, on the side facing the servomotor cover, to be visible from the outside.
- We must centre the IMU on the board for accurate measurements. Any offset could introduce errors due to rotational or translational effects.
- In order to have a good stability of the clock signal, we need to place the resonator very close to the microcontroller.
- We must use a Kelvin connection for the rooting between the current amplifier and the current-sense resistor as illustrated in Figure 1.5. This provides a more accurate current measure.

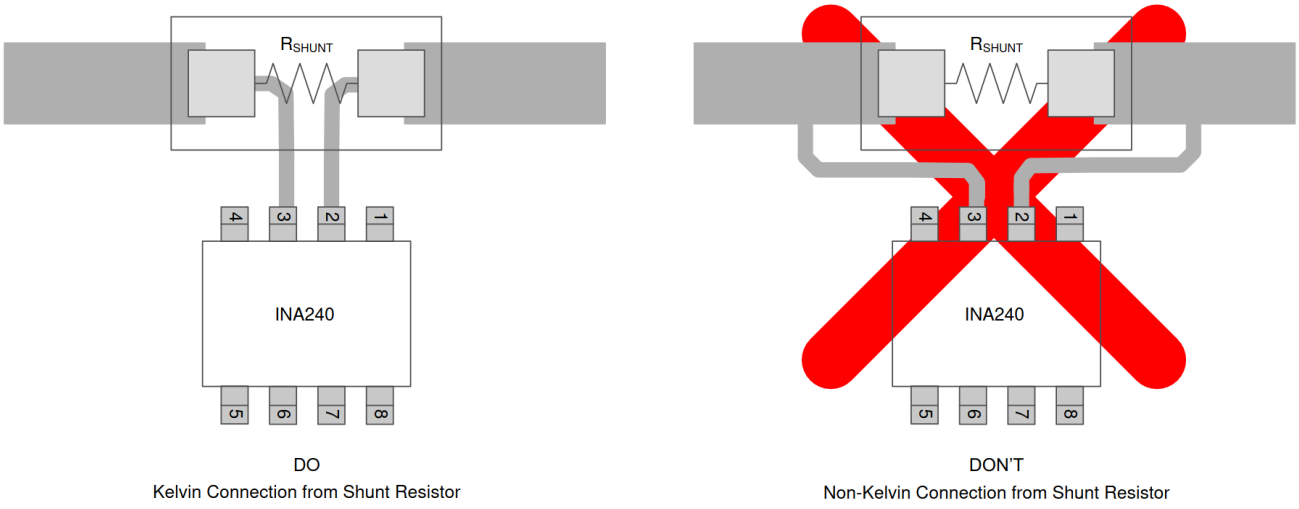


Figure 1.5: Shunt connections to the INA240 (Source: INA240 datasheet [4]).

The PCB is divided into 6 layers, illustrated in Figures A.3, A.4 and A.5. We can only place components on top and bottom layers. The density of components on board is really high, and we thus need multiple internal layers, used for routing, to connect all components with each other. We chose to dedicate one internal layer for the ground plane, improving the signal integrity and reducing electromagnetic interference.

Specific soldering methods are required for two particular components:

- The microcontroller has a large number of closely spaced pins and solder each pads individually would be impractical. Hence, we use a particular tip which can hold a small amount of solder. We applied soldering flux on the pads to ensure connections with no short-circuit. We then manipulate the soldering iron slowly and steadily over the pads, depositing solder to create the connections.
- The IMU has its pads located underneath the components, as we can see in Figure 1.6, which prevents the use of a soldering iron. Hence, we used a hot air soldering station. We describe below the complete process performed with pictures to efficiently visualise the process and to make it reproducible by an external person. We first apply soldering paste on the board (see Figure 1.7a). The paste is a mixed of solder powder and flux to create efficient electrical connection. We place the IMU as precisely as possible, using the silk layer as position reference (see Figure 1.7b). Then, we parametrise the hot air station with a temperature of 275°C with a percentage of air of 15%. A minuter of one minute is also set on the station to limit the time exposure of the IMU to high temperature. Finally, we place the head of the air blower on top of the IMU in the most vertical position possible. The soldered IMU is shown in Figure 1.7c.

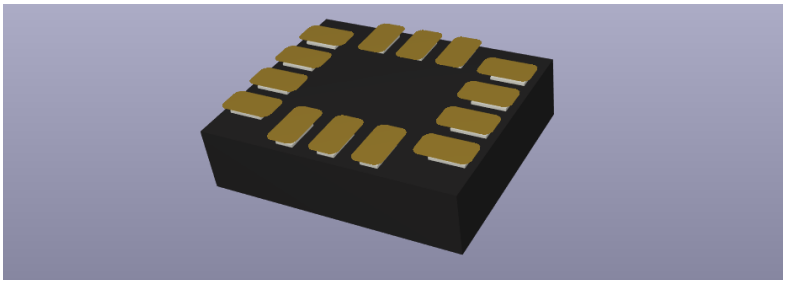
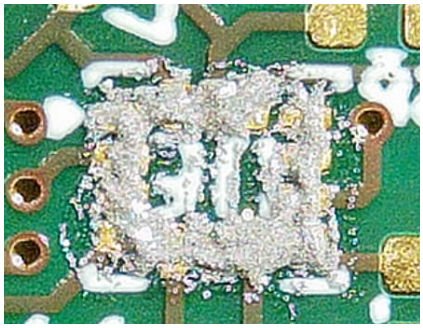
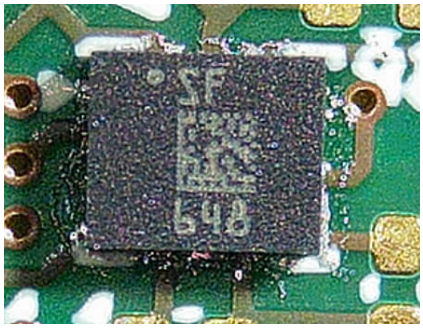


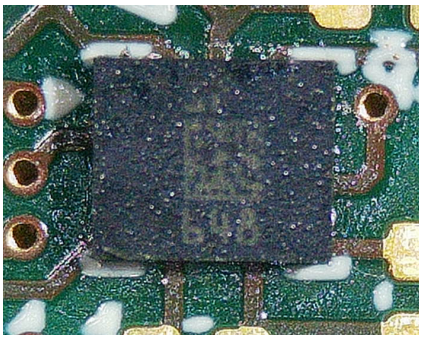
Figure 1.6: 3D model of the IMU.



(a) Soldering paste placed.



(b) IMU placed.



(c) IMU soldered.

Figure 1.7: Steps of soldering the IMU using a hot air soldering station.

The printed RM-28 in its mechanical enclosure is visible in Figure 1.8.

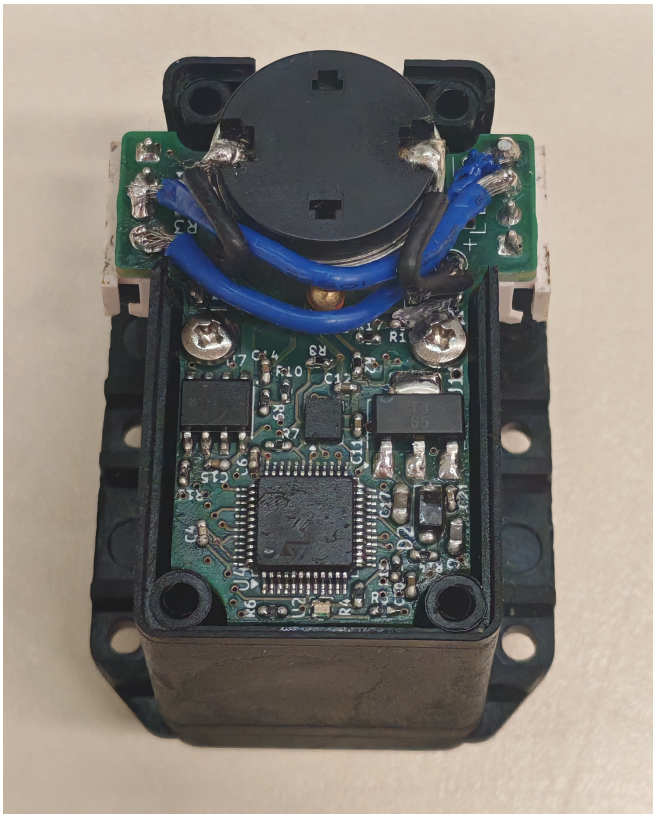


Figure 1.8: RM-28.

1.3.2 Multifunction board

State of the art

Before developing a robot for the RoboCup competition, the Montefiore Team participated during multiple years to another competition called Eurobot. The strategy of the Montefiore Team was to develop robots composed of several electronic boards dedicated to each functionality. For RoboCup, however, we chose to design a single board incorporating everything needed. This led to the development of the multifunction board on which multiple people have worked. The first version of the board was finalised by myself during my personal research project [33], but some issues remained. For instance, the MCU provides two CAN busses but only one was working. Another mistake is the emergency shutdown circuit, which was not working due to a design error. This circuit should power off the robot if the battery's voltage goes below a certain threshold, in order to protect the battery.

During the development of the second version of the board, we corrected the mistakes and enhanced the new version by clever components positioning. Moreover, we decided to remove the emergency shutdown circuit. In parallel of this thesis, Loris Degueldre is working on the development of a new electronic board responsible for managing battery charge and discharge. It was estimated that his board will measure the voltage with a higher accuracy. Hence, the new board of L. Degueldre will now be responsible for a better battery protection.

We do not fully detail the circuit design of the MF board in this work, as it was already done in my personal

research project [33]. The electronic schematics and PCB are available in Appendix B.

Electronic design

The multifunction board provides a link between the high-level motherboard and the low-level daughter boards. It acts as an intermediate to transmit information and power across the robot. Therefore, the MF board has various functionalities:

- Galvanic isolation of the battery from the board to avoid leakage current,
- Control of the MF board power switching and relay current reduction,
- Communication with RM-28 using the CAN bus, corresponding to two logical branches to double the available bandwidth, i.e. the data transfer rate,
- Communication with unmodified servomotors of Dynamixel using the RS-485 protocol,
- Manage the motherboard's start using a virtual start button,
- Power the MB with 12 V and the MF board itself with 3.3 V,
- Exchange of data with the MB using a two-way USB 2.0 data connection,
- Measurement of linear acceleration and angular speed using an IMU, the LSM6DS3H [5],
- Computation of complex operations using a powerful microcontroller (MCU), the STM32F429 [46], and
- Software emergency stop to protect the battery.

To structure the MF board, we organise the board into four main functional sections:

- Microcontroller Unit (MCU): This is the core of the board, managing all logic and communications.
- Power Management: This section is responsible for voltage regulation and protection. Using DC/DC converters, it generates both 3.3 V, used to power the MCU and all the other embedded components, and 12 V, used to power the MB.
- Peripherals: This part includes key components such as the IMU and the communication transceivers for CAN and RS-485.
- External Connectors: These connectors interface the board with the rest of the robot. They include ports for the CAN bus, RS-485, and USB, enabling communication between the board and external devices. In addition, general-purpose input/output (GPIO) pins are available for custom extensions or debugging. The board also offers a 12 V power output that can be used to supply other modules if needed.

The electronic schematics of the MF are shown in Appendix B, in Figures B.1 and B.2. The organisation of the MF board is illustrated in Figure 1.9.

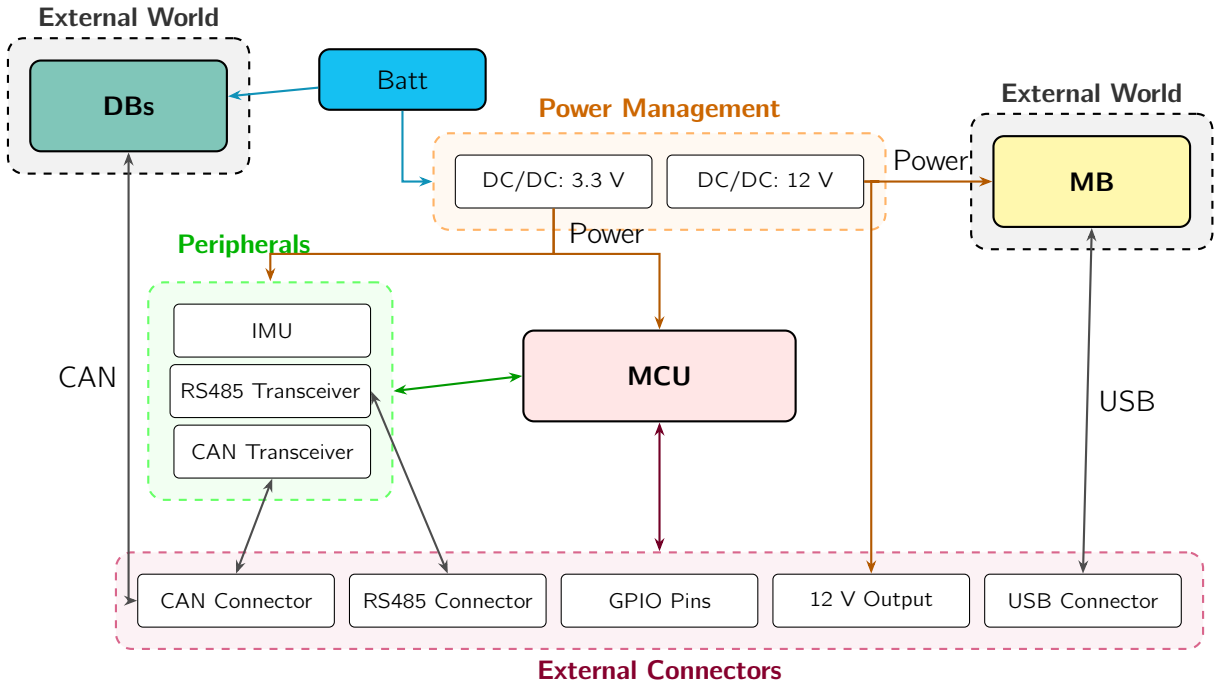


Figure 1.9: The multifunction board's diagram representation.

PCB design

The PCB of the MF board is composed of 4 layers, among those one is used for the ground plane. We also designed a power plane on the top layer to efficiently distribute power to all components that require battery voltage. The plane is segmented into two areas only connected through one bigger trace to avoid current loop. This division separates the relays from the DC/DC converters. This choice of layout simplifies routing and minimises voltage drops across the board. Similarly as for RM-28, we place the IMU in the geometric centre of the board for accurate measures. The PCB layers are available in Figures B.3 and B.4, in Appendix B.

The printed multifunction board is shown in Figure 1.10.

Figure 1.10: Multifunction board.

1.3.3 Hardware testing

For each custom board, we developed simple codes to test the critical sections. Their purpose is twofold. First, these codes ensure that we correctly soldered each component and that none of them are damaged. Second, we provide to future developers simple examples of the way we can setup and control the components.

These programs first initiate the devices used, i.e. we configure the clock's frequency and the pins of the MCU. Then, we test the component to verify that the device performs what is expected. For instance, we communicate with the IMU to retrieve the linear acceleration on the Z-axis. If the output is about 9.81 m/s^2 at steady state, then the test is successful.

1.3.4 Architecture

FreeRTOS

Both the MF board and RM-28 require to perform multiple tasks simultaneously. Similarly to what we presented for the motherboard in Chapter 1.2, we use a RTOS for our custom boards, FreeRTOS [47], to program and manage different tasks running in parallel.

FreeRTOS has the advantages to be stable and of small size which is important in our embedded configuration where memory is limited. Moreover FreeRTOS supports the STM32F303 and STM32F429 microcontrollers, respectively used for RM-28 and the MF board.

LibOpenCM3

The STM32 microcontrollers are complex devices which provides various functionalities, accessible by the physical hardware of the component. To facilitate the use of the MCU, we use a hardware abstraction layer (HAL), LibOpenCM3 [30], abstracting the hardware-level operations into usable functions.

1.4 Test bench

We built a test bench on which the multifunction board and the motherboard are fixed. This bench represents a portion of Bobby. When we develop algorithms, it is convenient to have a smaller platform on which we make our tests for multiple reasons:

1. The boards are often manipulated while developing, to connect the cables or to simply transport them. These manipulations involve risk of fall or shortcircuit. The boards are thus fixed to the bench in order to transport them easily and to facilitate the wiring.

2. On the test bench, we have fewer mechanical restrictions than on the full robot. We can thus test some algorithms by minimising the risk of damaging the mechanical structure of the robot, or injuring the testers. On the bench, we can for example perform controller tests with 2-joint arm configuration, before building the full arm if the tests perform well. If the control is poorly designed, testing on the 2-joint configuration is safer than performing the tests on the full robot directly.

3. The test bench is powered using an external power supply. Hence, we do not need to use the battery of the robot. On the one hand, it prevents the battery from draining, implying that we would need

to wait for the battery to be full again. On the other hand, minimising the use of the battery extends its lifetime.

4. Only a part of the robot's structure can be used on the bench. We can thus easily make located tests, on a single leg for instance. In our case, it was particularly useful since not all the custom servomotors were completed.

The annotated test bench is shown in Figure 1.11.

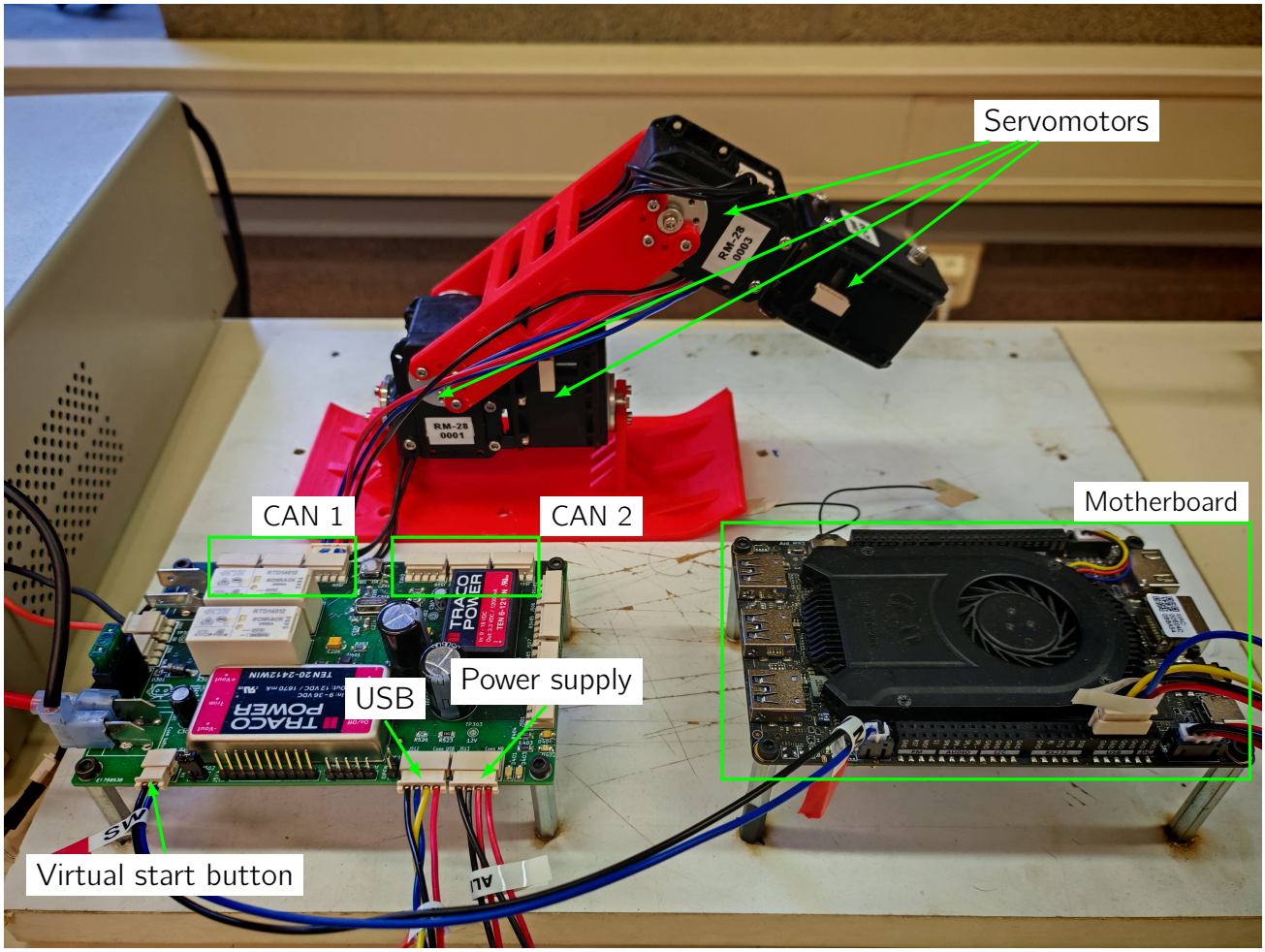


Figure 1.11: Annotated test bench setup.

Chapter 2

Low-level software aspects

In this chapter, we present the low-level software developed for the daughter boards (DBs), responsible for collecting the data measured by sensors and communicating over the CAN bus.

The DBs are only responsible for their own system by measuring physical quantities such as angular position, temperature or current consumption, performing computation based primarily on their sensor data and driving their actuators in response to MB commands. We designed that they do not actively notify external devices through the communication channel.

This follows a master and slave strategy where the motherboard acts as the master by requesting sensor data, combining information from multiple DBs and making high-level decisions. The DBs act as slaves executing the commands received, without taking much initiatives. Multiple reasons led us to this choice:

- For the sake of simplicity. It is more convenient to perform complex computations on the motherboard, even though the servomotors are equipped with a powerful microcontroller. The goal of this master thesis is to design an elementary layer of control. Future works will enhance this system to be more robust by introducing more local responsibilities, for instance.
- As mentioned in Chapter 1, the motherboard has high computation capabilities and full speed communication rate (12 Mbps). The MB is thus sufficiently powerful and fast to be the main computation unit.
- The motherboard has the highest view over the current state of the robot. Since some computations can be situation dependent, we let the MB compute complex operations to allow the DBs to be unaware of the global situation. This avoids numerous messages to notify the DBs on the current state.
- Finally, to simplify the collision-free management of the bus, the DBs should not send any message to other DBs or to the MF board. They are only responsible for answering messages.

A drawback of this method is the higher latency. Indeed, the information is first sent to the MB, then processed by the MB, and only after, transmitted back to the DB.
According to the official Book of Rules of the World Athletics organisation [48], a false start occurs when "the reaction time is less than 0.100 seconds". We suppose thus that the human absolute limit of reaction time is 0.1 seconds. Chapter 3.2 gives more details on the latency, but we can already assert that the delay of our system remains small and much less than the actual human reaction time.

2.1 DBs identification

To exchange data with the daughter boards, we need a way to address each DB, in other words, to uniquely identify each DB from the others. We want this identification system to be consistent and modular. Consistence means here that physically close DB should have a similar ID. One can visualise this constraint with a neighbourhood, where neighbouring houses have similar numbering. Modularity implies that if a new DB is added, it must be possible to assign it an ID similar to its neighbours. Moreover, this must not lead to a reidentification of all DBs.

To solve this issue, we chose to assign an unique identifier (ID) to each DB. We divide the robot into 2 parts, which one can visualise as a community. Each part is itself divided into 4 subparts, representing neighbourhoods. Inside the subparts, we assign a number to identify each DB among its neighbours.

In our project, we have two CAN channels and we divide the robot into two parts: the upper and lower body, each connected to one bus. For the neighbourhood, we take inspiration from the FDI (World Dental Federation) notation. The FDI divides the dentition in 4 quadrants and assigns an ID of 2 digits to each tooth. The first indicates the quadrant and the second, the tooth position from the midline. Therefore, a superfluous tooth is added to the quadrant without breaking overall tooth ordering. Following their scheme, we split each body part into 4 quadrants similarly to what is done by the FDI. Inside each quadrant, we attribute to each DB an integer value from 0 to 7.

This system implies that we have two different ID formats: 3-digit (xyz) for the high-, and intermediate-level perspective (motherboard - multifunction board) and 2-digit (yz) for the low-level (daughter boards), where

- x is a one bit identifier that indicates on which bus is the DB located (0 for the upper body and 1 for the lower one).
- y is a two bits identifier representing one of the four quadrants.
- z is a three bits identifier representing the position within the quadrant.

The low-level DBs are identified by the 2-digit format since they communicate on their own bus, and not with the other. From their perspective, the DBs do not need to be identified by their communication bus. By using 2 bits for the quadrant and 3 bits for the position, we use 5 bits to identify the DBs on the low-level.

Using this system, we can distribute DBs on both body parts and identify them efficiently. Since the MF board communicates with all DBs, the board is addressable from each bus. Within each bus, we assign the MF board to quadrant 0 with ID 0 within this neighbourhood. Therefore, the MF takes an available ID which leaves only seven available DBs on quadrant 0 instead of eight.

Thanks to the 2 bits of quadrant numbering and the 3 bits of quadrant positioning, we have $2^2 \times 2^3 - 1 = 31$ DBs per body part where we account for the MF slot. This capacity is sufficient for the project.

If we add a new DB on the robot, we would assign it to a community, depending on its body location. Then, we would attribute a neighbourhood to this DB, where neighbours share the same quadrant ID, making their identifier similar. This satisfies our consistency requirement.

A reidentification of a quadrant does not affect the whole community, satisfying our modularity constraint.

A first sketch of the ID assignment is shown in Figure 2.1. We can see that quadrant 0 is located in the centre of Bobby, since the MF board belongs to the two communities. The remaining quadrants correspond to anatomical limbs such as legs, arms, hip, and head.

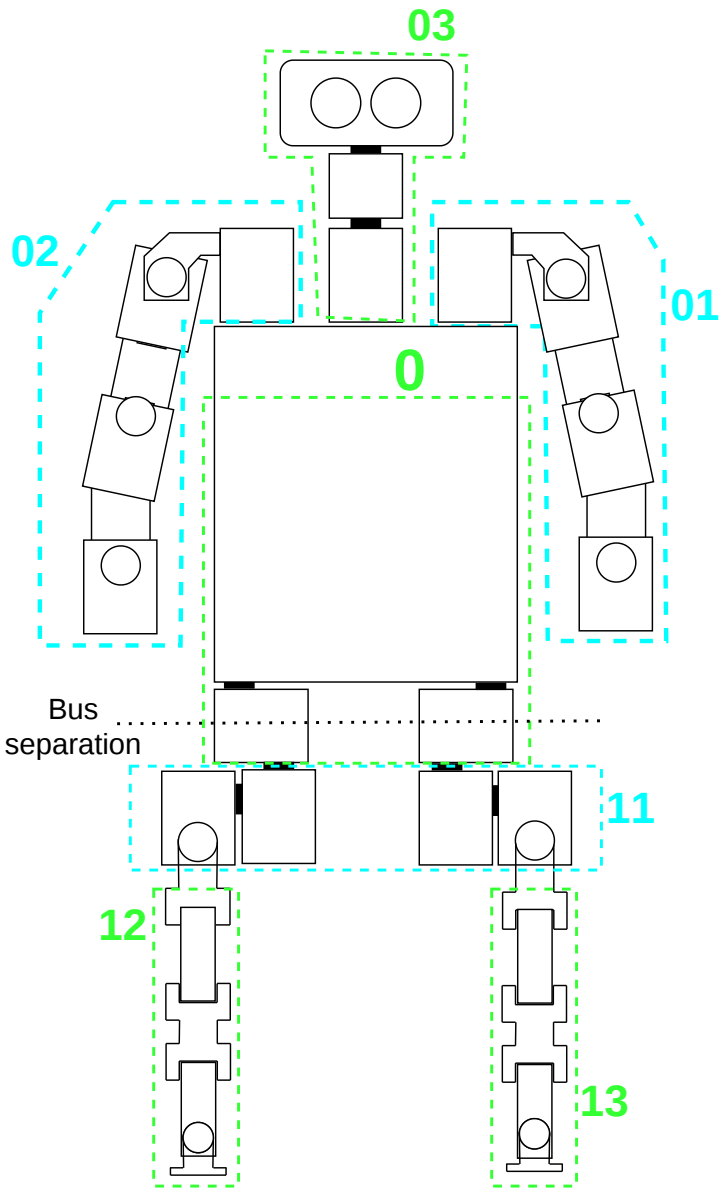


Figure 2.1: First sketch of the ID configuration.

2.2 CAN protocol

The multifunction board communicates with the daughter boards using two CAN communication channels as presented in Chapter 1. The CAN protocol [49] is a differential signalling protocol that transmits data through a physical bus. The peripherals on the bus are called nodes, sending and receiving CAN frames. This protocol provides very good noise immunity, which is very important in robotics because the DBs' power circuits can generate electrical interference on the bus.

The format of the message exchanged on the CAN bus is not specified by the protocol. Instead the developer is responsible for designing the frames. The CAN messages have an identifier of fixed size and a data field of variable size. We want three communication modes for messages transmission: broadcast which sends to every node, unicast which sends to a specific node, and multicast to multiple specific nodes. Using these three modes, we can reduce the number of messages on the bus, having thus more

time slots available for other messages. For instance, instead of sending an identical message to each node of a neighbourhood, we send a single message to notify all of them.

2.2.1 CAN frame

To identify each node on the bus, we use the low-level DB identification presented in Chapter 2.1. In the standard format, the CAN identifier is defined on 11 bits and we divide these bits into 5 for the sender identifier, 1 for the communication mode (CM) and the last 5 for the target identifier [1]. The CM is either 0 to indicate a broadcast mode and 1 otherwise. The structure of the CAN identifier is illustrated on Table 2.1.

The data field contains the message type (MT), used to identify the message among the others, and type-dependent parameters. By knowing the MT, the receiver knows the number and meaning of the parameters given in the data field.

The developed message's format is predefined and so, less flexible. However, it allows the messages to be very short as the sender does not need to insert any additional information to identify each parameter.

10	9	8	7	6	5	4	3	2	1	0
Sender ID					CM	Target ID				

Table 2.1: Structure of the CAN identifier.

2.2.2 Hardware filters

Knowing the CAN identifier format, we now explain the implementation of each communication mode. We use the embedded hardware filters of the MCU to filter messages based on their CAN identifier.

The CAN peripheral of the STM32F4 and STM32F3 microcontrollers [46], [43] provides hardware filters for receiving CAN messages. A message matching a filter is copied into a reception FIFO, which triggers an interrupt. The others are simply discarded. This filtering saves CPU resources since the software would perform the filtering otherwise. Indeed, having no hardware filter imposes that every node on the bus receives each message sent, whatever the communication mode.

Using the integrated CAN filter, we implement each communication mode:

- Broadcast: accepts each message whose *CM* is equal to 0.
- Unicast: accepts messages whose *Target ID* corresponds to the low-level ID of the node.
- Multicast: accepts messages whose *Target ID* matches the quadrant ID of the node.

The drawback of this method is that we can only multicast predefined neighbourhoods. For instance, we cannot send a message to a single arm in one situation and to both shoulder in another since they are part of different neighbourhoods.

The CAN peripheral provides 28 hardware filters shared among the two busses. We only use three filters for the moment, one for each communication mode. Since 25 filters are left, future work can address the flexibility's issue of the multicast mode by developing more complex filters.

2.3 Application messages

With the CAN frame just presented, we can define the frames used during the nominal operation of the robot, which we call the application messages. We have a limited bandwidth so we need to analyse the trade-off between long messages sent at lower rate or short ones sent frequently.

Each message is characterised by a message type from 0x00 to 0xff. Depending on the message, additional 0 to 7 bytes of payload data are included. Regarding the low-level logic explained in the beginning of this chapter, we developed the system around GET and WRITE messages. The MB updates the parameters of the DBs using WRITE messages and collects all the needed information using GET messages.

We chose to send small messages at a higher rate so that each message is as useful as possible. On the one hand, a message should no be delayed because its data field is not yet full. On the other hand, we do not want to fill the data field with useless information.

With this system, only the needed messages are exchanged and thus, we optimise the bus utilisation. Moreover, we chose to attribute an integer identifier to each information exchanged on the bus. We call this identifier the Information ID. For example, to request the motor's angular position, the MB send a GET message with the Information ID associated with the angular position. This system compacts the message while being sufficiently flexible as more Information IDs can be added without affecting other IDs.

We list all application messages developed in Appendix C with the message type (MT), any other potential data sent, and returned frame.

2.4 Event reporting system

During software development and the robot's operations, we want the servomotors to report errors and status information. For instance, the angular position measurement provides 6 status bits, indicating the integrity of the data sensed. The MB and remote operators must be notified whether a problem occurred, but the daughter boards cannot actively send messages on the bus.

We use two systems to report information: one to the MB using flags, and the other to the operators, using a LED.

2.4.1 LED

The multicolour LED signals events by blinking with a specific colour and frequency, where, for instance, a fast red blinking signal corresponds to a critical error. The LED emits a signal whose colour depends on the intensity of each primary colour (red, green, and blue). This system has the advantage to be straightforward as the operator easily sees the LED indications. However, it remains simple and quite limited to report detailed information. Using the LED for various events requires the operator to know the meaning of each colour and blinking frequency pairs. Moreover, we cannot efficiently record these signal reports and it is hard to know when the event exactly occurred.

2.4.2 Flags

The second system associates a flag, i.e. a binary values, to each event, which the servomotor raises to signal that the event has occurred and clears at each cycle. It is the responsibility of the MB to collect

them on time. Once received, the MB stores their reception timing, which is crucial for debugging or tracking servomotors' state across time, and displays a detailed message to the operator.

Unlike the LED method, the events are understandable without a priori knowledge, nor the inner working of the report system. Finally, this method is modular since tracking more events only requires implementing more flags. However, the main limitation is that the MB must actively request and process the flags. Thus, this reporting cannot operate on its own and is less suited for fast debugging.

In practice, we combine the flags into a byte where each bit represents a single event. The MB can request this byte using GET message. Currently, only one flag byte is implemented. We give the meaning of each bit in Appendix D.10.

2.4.3 Combination

By combining the two systems, we obtain a straightforward but general report using the LED and a more informative but complex one using flags.

A critical error situation illustrates the complementary of both systems: the LED starts blinking fast in red, the operator sees this indication and knows that a problem occurred. He then checks what the MB has received and can understand the exact cause of the issue. This combination ensures rapid detection and deep analysis.

The `flags.c` and `diag.c` files are respectively responsible for handling the flags and LEDs logic and are both located in the `stm32/src/utils/` folder.

2.5 RM-28's sensors interface software

RM-28 are equipped with various sensors as presented in Chapter 1.3.1. We now describe how the MCU interfaces each sensor to collect the measured data.

We use the SI convention [50] to express the unit of each quantity. For instance, speed is expressed in m/s and force in newtons, with the only exception for angles, represented in degrees instead of radians. We decided that this is more intuitive for the Montefiore Team to work with an angular velocity in dps rather than in rad/s.

In practical situations, we cannot continuously exchange messages with the servomotors since multiple daughter boards share the same CAN bus, and simultaneous transmissions would cause collisions. Therefore, we defined a cycle of 10 ms, on which the servomotors are synchronised. Every 10 ms, the servomotors update their output, which controls the motor, based on the computations and sensor data from the previous cycle. Then, a new cycle of operations begins: the servomotors receive commands from the MB, read their sensors, and compute the next output.

2.5.1 Motor's angular position

The MB commands the motor in position to achieve high-level objectives. For instance, it determines the position in space of the feet, knowing the robot's mechanical structure and angular position of each servomotor of the leg.

As presented in Chapter 1.3.1, the angular position is determined using a magnetometer, the AS5045, which measures the magnetic field induced by a magnet glued to the shaft of the servomotor. This component outputs the current angular position with respect to a zero reference. Depending on the

zero's position, the angle measured is different as illustrated in Figure 2.2: with the same angular position, two distinct α and α_{bis} are measured.

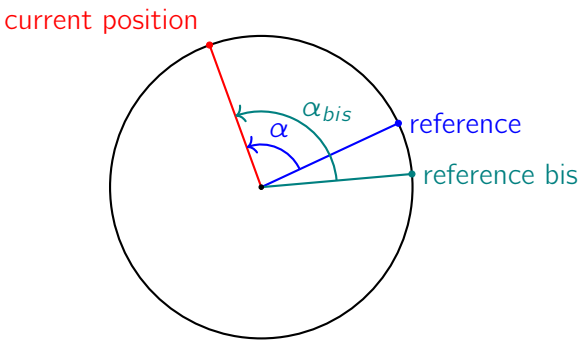


Figure 2.2: Angular position's measurement with different references.

Following the component's datasheet [2], the measured data is available as a serial bit stream and as a PWM signal. In our case, we chose the bit stream, i.e. the data is transmitted one bit at a time. The angular information is encoded on 12 bits, followed by 6 status bits, used to detect an error. To read each bit of information, we manually generate a clock cycle on the clock pin of the component. Finally, AS5045 provides zero position programming so that the software matches the mechanical and logical zero positions. This is more convenient than manually adjusting the magnet to the zero position.

We process the status bits by checking that each value corresponds to the one expected. If the value is incorrect, we raise the magnetometer flag. We use a single flag to represent each potential error as the measurement is robust and we should never get an error during normal usage. Hence, if an error occurs, regardless its meaning, it is sufficient for the motherboard to know it. The remote operator would then perform further analysis to understand the cause.

The magnetometer provides a measure of motor's angle with a resolution of 0.0879°, i.e. 4096 positions per revolution. We thus have to multiply the 12 bits of data by 0.0879 to convert the data in degrees. The measured values range from 0° to 359.9505°. Therefore the 360° and 0° positions overlap and the magnetometer cannot differentiate them.

2.5.2 Motor's speed

Similarly as for the angular position, the MB commands the servomotor to keep a specified velocity.

By motor's speed, we refer to the angular distance travelled by the motor during an unit time. It is different from the servomotor's speed which is the velocity of the whole servomotor, i.e. of the mechanical enclosure of the servomotor. The motor can rotate clockwise (CW) or counter-clockwise (CCW) as illustrated in Figure 2.3, so the software must be able to differentiate the rotation's direction.

Direction of rotation

We represent the motor's speed by a signed value, whose sign determines whether the motor turns CW (positive) or CCW (negative).

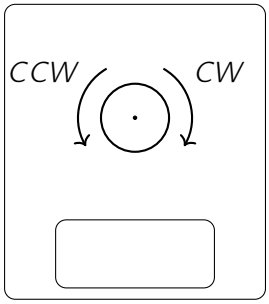


Figure 2.3: Rotation convention (view facing the motor shaft).

Angle difference

The magnetometer used only provides the angular position of the motor and not its speed. We chose to not populate RM-28 with a motor's speed sensor as we evaluated that the density of components is too high for another device. Thus, we compute the motor's velocity by taking the derivative of the angular position, which requires us to determine the difference between two angles.

To compute an angular difference, we cannot simply subtract them due to the continuous angular representation: the 0° position is physically closed to 359°, while being logically distant.

If the servomotor's position goes from 20° to 330° in one second and turning CCW, the computed velocity by simply taking the angular difference is equal to:

$$\omega = \frac{330 - 20}{1} = 310 \text{ [dps]},$$

while the actual speed is of -50 dps, taking into account the direction of rotation convention described before. The situation is illustrated in Figure 2.4.

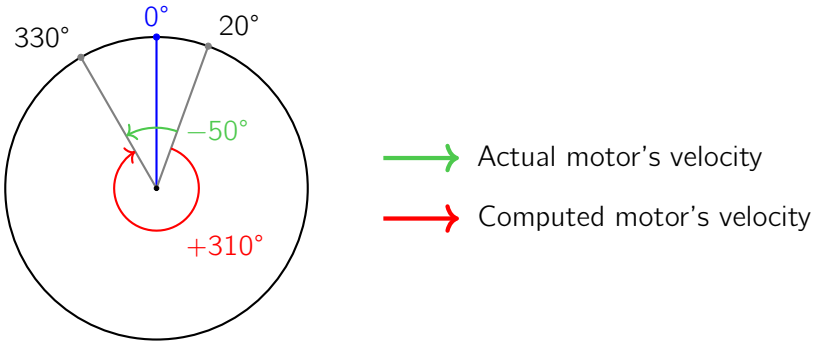


Figure 2.4: Example of angular speed computation's issue.

Therefore, we developed a function to properly compute the difference between two angles. We compare, in absolute value, $\alpha_t - \alpha_{t-1}$ and its coterminal to determine which is the absolute smallest one. The function then returns the small and big angle differences.

If we take back the previous example, the function computes that the difference is equal to 310 and its coterminal equals -50. By comparing them in absolute value, we get that:

$$|310| > |-50| \tag{2.1}$$

The function concludes that the angular difference is equal to -50, and thus an angular velocity of -50 dps.

We must note that our system assumes that the motor goes from 20° to 330° while taking the shortest path. In practice, we computes the motor's speed every 10 ms, thus taking the longest path would imply that the motor's speed is of

$$\omega = \frac{310}{0.01} = 31000 \text{ [dps]},$$

which is impossible, thus our assumption is realistic.

2.5.3 IMU

While the motor's angular position provides the joint angle used to determine the servomotor's 3D position, the IMU measures the orientation of the servomotor in space, which is crucial for walking and shooting.

The IMU combines two sensors into one physical device, measuring 3-axis linear acceleration using an accelerometer and 3-axis angular velocity with a gyroscope.

Before describing the interface developed, it is important to understand that the accelerometer actually measures the proper acceleration applied to a body, which includes the gravitational force of the earth. The accelerometer, in steady-state, measures 1 *g* on the Z-axis, which we assume is aligned with gravity here, while both the X-axis and Y-axis measure 0 *g*. If we want to measure the actual force applied to the body without the gravitational effect, we need to post-process the data by subtracting the gravitational component.

The LSM6DS3H [5] provides two communication interfaces: the serial peripheral interface (SPI) and the inter-integrated circuit (I^2C) control interface, which is the one we chose to use. I^2C is a bidirectional 2-wire bus with a serial data line (SDA), used for sending and receiving the data, and a serial clock line (SCL) [51]. We communicate with the IMU by sending I^2C messages requesting the data measured, individually for each dimensional axis. The axis conventions of the IMU are visible in Figure 2.5.

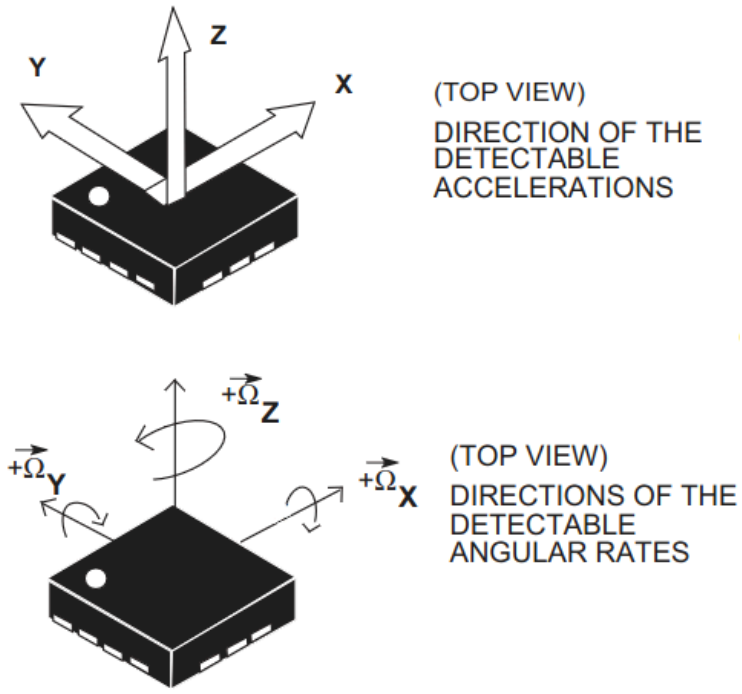


Figure 2.5: Accelerometer and gyroscope axis conventions (Source: LSM6DS3H's datasheet [5]).

First, we set output data rate of the accelerometer and the gyroscope, using respectively the angular rate sensor control register (CTRL2_G) and the linear acceleration sensor control register (CTRL1_XL). For both sensor control, we set the output data rate to 1.66 kHz, such that the data rate is approximatively 16 times higher than the cycle's frequency:

$$\frac{1}{10\ [\text{ms}]} = \frac{1}{0.01\ [\text{s}]} = 100\ [\text{Hz}] \tag{2.2}$$

For both control registers, we can select the scale of the data output. The smaller the scale, the lower the sensitivity. There is thus a trade-off to discuss: finding the appropriate scale which can measure the range of possible values while keeping the sensitivity as low as possible.

Accelerometer

We set the accelerometer scale to $\pm 2\ g$[1] since the linear acceleration of the servomotors should not exceed $2 \times 9.81 = 19.62$ [m/s²] in normal usage. The sensitivity of the accelerometer is equal to 0.061 mg. Thus, to convert the raw data into m/s², we multiply the linear acceleration data by $0.061 \times 9.81 \times 10^{-3}$.

Gyroscope

We chose to configure the gyroscope's scale to 500 dps as it is a good trade-off between large range of possible values and the corresponding sensitivity, 17.50 mdps, which is sufficiently low. To get the angular speed in dps, we multiply the measured data by 17.5×10^{-3}.

Data measurement

Both the accelerometer and the gyroscope provide 3-axis measurements (X, Y and Z). Each value is a 16-bit signed integer, encoded in two's complement representation. Since our IMU uses 8-bit registers to encode the data, each measure is split into two registers. Therefore, to read a measure of the IMU, two I²C messages are required. We can then reconstruct the 16-bit information by combining the two registers.

The servomotor angular velocity follows its movement in time, but the sensor is sensitive to drifts, i.e. over time small errors accumulate resulting to an incorrect orientation estimation. Using linear acceleration, the gravitational force gives an absolute reference when the servomotor is immobile and corrects the drift of the gyroscope. By merging these measurements, we get a clean orientation estimation.

2.5.4 Motor's temperature

It is important to have a good overview over the actual temperature of the motor, which can become sufficiently high to damage the servomotor. If the temperature raises too much, we must shutdown the robot to ensure material safety.

The MCU has an integrated temperature sensor but we decided that an external sensor located near the motor would give a more accurate information. We chose to use a NTC resistor which has a decreasing resistivity, and thus a decreasing resistance, with increasing temperature. Using a voltage divider, we determine the value of the NTC resistance.

[1] Note: g is acceleration induced by the force of gravity and is approximate to 9.81 m/s² in this work.

Based on the resistance found r_{th} and using the formula given in the NTC resistor's datasheet[2], we compute the estimated temperature of the motor using Equation 2.3:

$$T = \Big(3.354016 \times 10^{-3} + 2.569850 \times 10^{-4} \ln\big(\frac{r_{th}}{r_{ref}}\big) + 2.620131 \times 10^{-6} \ln^2\big(\frac{r_{th}}{r_{ref}}\big) + 6.383091 \times 10^{-8} \ln^3\big(\frac{r_{th}}{r_{ref}}\big)\Big)^{-1} \text{ [K]} \tag{2.3}$$

where r_{ref} is given in the datasheet and is equal to 10 kΩ. To respect the SI convention, we convert the computed temperature in °C.

2.5.5 Power supply's voltage

The servomotors are not responsible for managing the battery voltage, but knowing this information is still useful as the power supply is transported across the robot, from the MF board. At the end of the supply chain, the servomotors could observe a voltage drop due to resistive losses along the power path and receive a lower voltage than what is given by the source. The servomotor measures its input's voltage to detect these losses so that remote operators resolve this issue by adding other power injections along the power path.

To read the power supply voltage, we use the ADC function of the microcontroller to determine the output's voltage of a potential divider with the battery voltage as input.

2.5.6 Torque estimation from the current consumption

We explained in Chapter 1.3.1 the use of a H-Bridge to control the DC motor. The torque produced is estimated from the measure of a current amplifier sensor, the INA240A1 [4]. Similarly to the motor's speed reference, the software must differentiate the direction of the torque produced. Moreover, we should also be able to know when the motor actively produces a torque or is driven by an external force.

The INA240 measures the voltage drop across a shunt resistor by amplifying it with a fixed gain. The MCU then reads the measured voltage drop outputted by the sensor using a 12-bit ADC. Knowing the value of the resistance used and the voltage measured, we compute the current consumption using Ohm's law.

The gain of the INA240, G_{INA}, equals 20 and using a shunt resistor of 33mΩ, we compute the voltage drop:

$$V_{drop} = 3.3 \times \frac{V_{out}}{4095} \times \frac{1}{G_{INA}} \tag{2.4}$$

where V_{out} is the raw value read by the ADC.

Using Equation 2.4 and Ohm's Law, we compute:

$$I = \frac{V_{drop}}{R_{shunt}}$$
$$I = 3.3 \times \frac{V_{out}}{4095} \times \frac{1}{20 \times 0.033} \text{ [A]} \tag{2.5}$$

[2]Note: the coefficients used in the formula are the ones given in the datasheet, with 6 decimal digits.

The torque is then computed as follows:

$$\tau_{measured} = K \times N \times I \tag{2.6}$$

where K is a physical torque constant of the motor which equals to 10.7 mNm/A [45] and N, the gear ratio, equals 193:1 [24]. We estimate that each servomotor consumes up to 1 A, so the maximal torque produced is about 2065.1 mNm.

However, the measured value does not directly represent the torque produced because of two reasons. First, to represent the current as a signed value, the INA240 outputs a value centred around $V_{DD}/2$: if smaller than its midpoint, the current is negative and positive otherwise. Second, the presence of friction forces implies that the measured torque does not exactly correspond to the useful torque developed by the motor.

As already explained in the beginning of this section, the servomotors communicate their sensor's value once each 10 ms. Across this cycle, we compute the mean of V_{out} to sharpen the precision of Equation 2.6.

Similarly to the motor's speed, we represent the torque as a signed value, using the same convention: the torque produced in the CW direction is positive, and negative otherwise. Nevertheless, using only the torque sign, we cannot conclude whether the motor is actively producing torque or driven by an external force. To disambiguate the two cases, we combine the torque direction with the motor's velocity direction:

$$\begin{cases} \tau \times \omega > 0 & \Rightarrow \text{Motor} \\ \tau \times \omega < 0 & \Rightarrow \text{External force} \end{cases} \tag{2.7}$$

2.5.7 Software developed

Pseudocodes of the algorithms described in this section are given in Appendix E.1. The actual codes are located in the `stm32/src/sensors/` folder of the project repository.

2.6 Control of the motor

In Chapter 1.3.1, we presented the H-Bridge circuit and the logic to control the motor using phase-shifted centred PWMs. We now describe its practical implementation and how we use it from a higher-level perspective.

We define two control spaces:

- The actuator space refers to the hardware PWM implementation, where the MCU controls the motor by generating two PWM signals, one per gate driver. In this space, the MCU uses two unsigned variables to configure the duty cycle of the signals.
- The abstract control space represents both the intensity and direction of the torque produced in a single signed variable, called PWM command. This abstraction makes the control more intuitive and more modular. Indeed, modifications to either the hardware or the motor control method would not affect the high-level command logic.
- The interface between the two spaces maps the PWM command to two PWM duty cycles of the actuator space.

2.6.1 Actuator space

The MCU's timer peripheral operates as counter incremented at a configured frequency and provides additional PWM generation functions. The duty cycles of the PWM are configurable using an output compare value. Whenever the timer counter matches this value, the corresponding output signal changes state.

We took a period counter of 1024 with a centre-aligned counting mode. With our 72 MHz clock frequency, each PWM signal has a frequency of

$$\frac{72000\ [\text{kHz}]}{2 \times 1024} \approx 35\ \text{kHz} \tag{2.8}$$

where the factor 2 in the denominator comes from the centre-aligned counting mode: the counter goes up then decreases within one period (0 -> 1024 -> 0).

Since the phase-shifted PWMs doubles the frequency seen by the motor, as explained in Chapter 1.3.1, the motor operates at approximatively 70 kHz.

To control the motor, we set the compare value of each PWM signal. The absolute difference between the two values determines the torque's intensity, while the direction depends on which gate receives the higher value. If the right gate receives the higher PWM, the torque produced is CW and CCW otherwise. For instance, a symmetric configuration, where both compare values are set to 512, corresponds to the zero produced torque. In contrast, if one compare value is set to 0 and the other to 1024, the motor produces its maximal torque.

2.6.2 Abstract control space

In the abstract control space, we represent the PWM command by a single signed value within a symmetric range from -512 to 512. A value of 0 corresponds to the zero torque produced, while the sign of the command indicates the torque's direction, consistently with the convention introduced in Chapter 2.5.6. This abstraction simplifies motor control by hiding the actuator-level control behind a single input.

2.6.3 Control space mapping

To map the single PWM command to the actuator space, we use the following method:

```
Input:
    pwm_command

compare_value <- pwm_command + 512 # Put input into the [0, 1024]
   range

# Set left gate compare value
left_cv <- 1024 - compare_value

# Set right gate compare value
right_cv <- compare_value
```

If the PWM command is positive, the right gate value is always greater than the left one, which generates a positive torque. In contrast, if the command is negative, the motor produces a negative torque. Finally, if the command is null, both compare values are equal to 512 which corresponds to zero torque produced.

The code controlling the motor is the `pwm.c` located in the `stm32/src/controllers/` folder of the project repository.

2.7 Concurrent programming

We previously mentioned in Chapter 1.3.4 that we want RM-28 to perform multiple tasks simultaneously. Using FreeRTOS, we can specify the priority of execution of each task. The challenges of the developer are multiple. First, he needs to precisely divide each task by importance to prevent critical tasks from being delayed because of the execution of less important tasks. Moreover, if tasks run periodically, it is crucial to avoid busy waiting, explained in Chapter 2.7.2. Finally, he must avoid data races' issues, detailed in Chapter 2.7.1.

2.7.1 Data races

A program has data races if multiple tasks access shared memory concurrently and at least one is a write operation. When data races occur, the program is not sequentially consistent and may lead to inconsistent or corrupted results.

To avoid this issue, we use synchronisation mechanisms such as locks or binary semaphores. When a task enters a critical section, it acquires the lock. Another one trying to reach the same section is blocked until the lock is released.

The use of the RM-28 sensors can lead to data races because some measures imply a write operation. For instance, retrieving the angular position requires to first initiate the measurement and then read the value bit by bit. If a task tries to read the angular position while another is still using the sensor, the second read would start a new measurement sequence, which corrupts the first read.

For this reason, we use locks to protect the data measure.

2.7.2 Busy waiting

Using multithreading, it is quite usual for a task to wait, for some data or for other reasons. For instance, a task can wait for the angular position of the motor to become available to resume its computation. Another example is if a task is periodic and does not need to be executed continuously.

A thread performs busy waiting when it waits while doing nothing useful. CPU resources are then consumed instead of being used for something relevant.

The solution is to use an operating system service which suspends the current running thread and another which awakes the suspended thread. FreeRTOS provides these facilities with TaskNotifyTake(), waiting for a task notification in a way similar to taking a binary semaphore, and TaskNotifyGive(), sending the notification [47].

2.7.3 Implemented tasks

Currently, the software running on RM-28 consists in two tasks:

- Task *orders* responsible for receiving and handling messages on the CAN bus. Once the frame is received, this task checks its type and calls the appropriate function to handle the message. To be

efficient, this thread waits for an incoming frame to execute. To notify this task, the MCU uses the interruption raised when a message is received on the CAN bus. This task has the highest priority as new incoming commands are always more important than what is currently done. If the MF board sends an emergency stop, we want the servomotor to immediately receive and handle it.

- Task *main-loop* performing the main operations such as updating sensors' measure and computing the next output. This thread is periodic and runs every 10 ms. His priority is the second highest because it performs the main operations of the software.

Other tasks can be implemented later, such as a low priority task *diagnostics* similar to the one presented in Chapter 3.1.

Regarding the current logic, an attentive reader could question the utility of locks for data measurements since only one thread performs these measures. Indeed, if only one task enters the sensors critical section, then there is no need for lock protection. This is done because the current threads' responsibilities are not fixed and future work can decide that it would be better to divide the *main-loop* responsibility into subtasks. Taking this into account, we chose to already include the use of locks to prevent possible future data races.

Chapter 3

Intermediate-level software aspects

The intermediate-level is dedicated to the multifunction board. This board has various responsibilities as presented in Chapter 1, and we focus this chapter on the communication logic.

As a reminder, the MF board communicates with the motherboard through a USB 2.0 channel, and with the daughter boards using CAN busses. We can simplify the communication logic as the MB transmitting to the MF board, a set of messages to send to the DBs or intended to the MF. These messages command the DBs to run an algorithm, update some variables, or request for data. In this chapter, we call these messages the daughter messages for simplicity.

Similarly as for RM-28, the MF board uses multithreading to execute different tasks in parallel (cf. Chapter 2.7). We begin this chapter by describing the current state of the multifunction software.

3.1 State of the MF software

The interface software of the MF board is still in development. Chapter 1.3.2 presents the different functionalities of the board which are successfully tested individually. However, we have not yet implemented a single software incorporating these functions. For instance, the MF board is equipped with an IMU, which is the same used by RM-28. We can thus implement the interfacing similarly, but the MB cannot request these data yet. Another example is the virtual start of the motherboard. We decided that it would be impractical to start the MB using the physical button when the board is mounted on the robot. We designed a circuit to virtually start the MB, but the current software running on the MF does not use this functionality.

Based on the work of Nadir Bounar [31], the Montefiore Team developed three tasks for the MF board, which compose the current software of the MF board:

- One task is responsible for USB communication. It has the highest priority, as we do not want the communication to be delayed.
- The second one is the *main-loop* task which, as the name suggests, is responsible for the execution of the main task of the MF. It includes, for instance, the communication on CAN busses. Its priority is intermediate.
- The last one is the *diagnostics* task which lights up the LEDs with a dedicated colour and period. This is used for debugging or notifying the remote operator. The priority of *diagnostics* is low as no useful computation is done for the robot.

3.2 Communication cycle

We defined in Chapter 2 that the DBs are passive on the bus, i.e. they do not actively communicate on the CAN channel but only reply to messages sent by the MF. We have to develop a strategy to communicate with the daughter messages without any collision, occurring when two nodes use the bus simultaneously. Moreover, we must define the use of the USB communication.

We developed a communication cycle handled by the MF board, which one can see as the metronome of the robot. The responsibility of the MF is then twofold. On one hand, it synchronises the USB channel by sending the set of answers, called the *uplink block*, to the MB commands of the previous cycle. Then, the MF board receives the new set of commands, called the *downlink block*. The MB associates the 3 digits receiver's ID to each command, introduced in Chapter 2.1, and a timing at which the MF must send the message. On the other hand, the MF synchronises the CAN channel by broadcasting a SYNC message to the DBs. Following this, the MF sends each daughter message on the appropriate bus and at the low-level address. The multifunction board tries to respect the timing imposed by the MB.

The response of each DB is collected and stored in the next *uplink block* by the MF board in order to transmit them to the MB at the beginning of the next cycle. If a message of the MB is intended to the MF board, this message is then simply processed by the MF and its reply is also stored in the next *uplink block*. The communication strategy is illustrated in Figure 3.1.

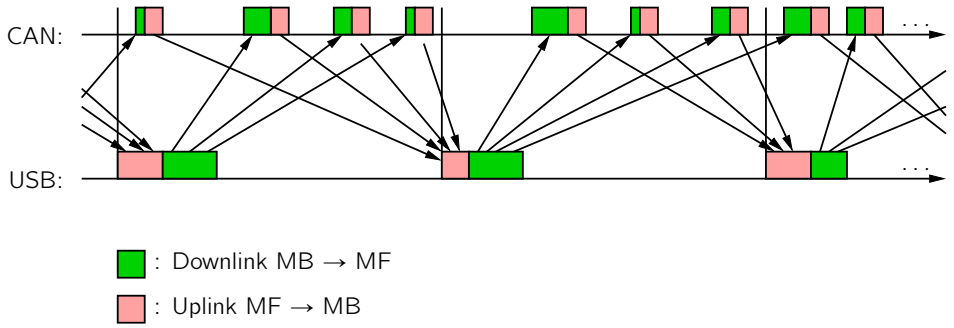


Figure 3.1: Communication strategy (Source: Montefiore Team [1]).

Each CAN bus operates at 1 Mbit/s with 500 kbit/s of useful throughput. In the current version of the project, we consider a period of 10 ms, i.e. there are exactly 100 periods per second. Considering the payload, we can exchange 625 bytes of payload per period:

$$\frac{500000 \cdot 0.01}{8} = 625 \text{ bytes per period} \tag{3.1}$$

which is high enough to communicate with each DB of the bus.

The chosen period implies a typical latency of 10 ms for the DB, with a worst case of 20 ms, which occurs if the DB sends its data at the beginning of the cycle and receives the response at the end of the following cycle. This latency is at least 10 times lower than the human reaction time limit, presented in Chapter 2.

3.2.1 Timing description

The broadcast SYNC message gives a common time reference t_0 to all DBs. The MF board then sends a series of messages to the DBs at specific times, i.e. $t_0 + \delta_i$ where the δ_i values are given by the MB. We explain the computation of these delays in Chapter 4.2. The CAN sequencing is illustrated in Figure 3.2.

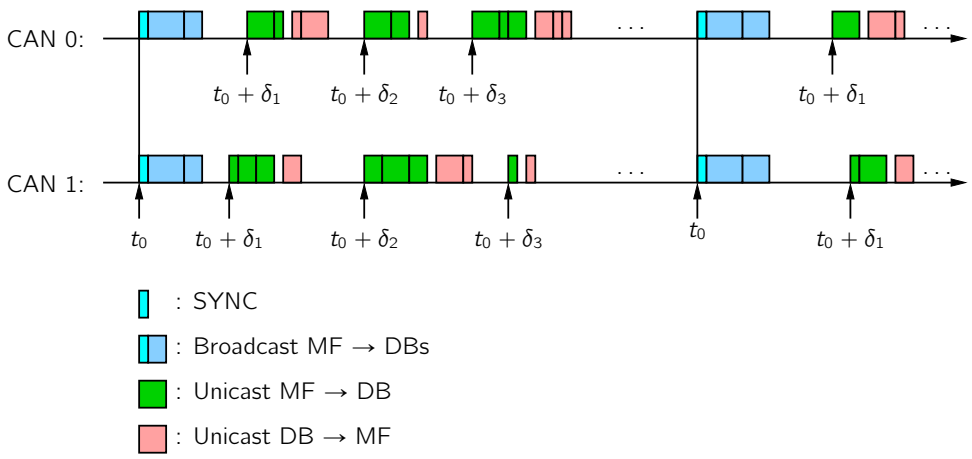


Figure 3.2: CAN sequencing (Source: Montefiore Team [1]).

The MF board processes each daughter message sequentially, waits until the specified timing δ_i is reached, using the integrated timer of the MCU, and transmits the command on the CAN bus. It requires the MB to send the messages in chronological order because otherwise some scheduled transmission times would already have elapsed when the MF board processes them. The MF would then discard these messages.

The MF board tries to respect the delays specified by the MB but if it misses the timing, the message is still sent. One can see these timings are as wish list of the MB, so the motherboard cannot assume that they are always respected.

Finally, the servomotors update their output and data measurements when receiving the SYNC message. This confirms the statement given in Chapter 2.5, where we briefly mentioned the servomotors' update each 10 ms, which corresponds directly to the cycle period.

Chapter 4

High-level operations

The high-level layer developed in this project provides a platform interfacing the low- and intermediate-level software, introduced in the previous chapters, in order to achieve elementary control of the servomotors.

The developed software should be sufficiently extensible to integrate future daughter boards. This platform should also be robust enough so that future work can build over it to implement more complex algorithms such as computer vision, motion planning, and global strategy.

4.1 State of the art

Before RoboCup, the Montefiore Team participated for many years in the Eurobot contest for which they developed robots to achieve different objectives each year. The internal components of these robots communicated over a 100 kbit/s serial bus, which is not fast enough for real-time continuous communication. They were thus unable to command all their actuators continuously and instead, performed preprogrammed sequences of operations. Finally, the Montefiore Team developed, for Eurobot, a high-level software which divides the responsibility of each task [52], which inspired our units system.

For RoboCup, the team decided that a more robust control system should be used which led to the development of RM-28 using the CAN communication protocol, providing higher bandwidth and improved reliability through its built-in error detection mechanism. Thanks to this change, the robot can transmit more information between its components, enabling more powerful and complex strategies.

As presented before, the motherboard runs multiple threads, each responsible for a specific functionality, using Linux as a real-time operating system thanks to PREEMPT_RT (cf. Chapter 1.2.1). The tasks may require information from the daughter boards or from other tasks to perform their operations. During his master's thesis [31], Nadir Bounar developed an interface that provides communication logic between tasks. Each of them subscribes to the messages that interest the task. The thread then waits for the data and is notified once the data become available. Nevertheless, we found a logical bug in the implementation of his system which results in O(n) complexity, where O(1) was expected.

Nadir Bounar also developed a log system, incorporated into the Bobby platform, particularly useful for debugging and tracking the state of the robot across time. With his log system, we attribute a severity level to all entries. Each of them also includes the timestamp at which the entry is recorded and the location in the code where the event occurred which provides accurate state tracking. This system writes all the logs in a text file stored on the MB.

The Montefiore Team [1] developed the USB interface to communicate between the MB and the MF board and defined the format of the exchanged messages (uplink and downlink blocks).

Finally, part of our team is currently working on a dashboard which will provide real-time information on the robot in use. By combining it with the log system, the dashboard will also enable filtering and analysis of the log file to more easily track the robot's state, for instance, by providing severity filters to show only part of the log entries.

4.1.1 Objectives of the high-level software

Based on prior works, the developed high-level software must fulfil multiple objectives.

First, the developed platform is deployed on the motherboard of the robot. It communicates through a USB channel with the multifunction board, which orchestrates the communication cycle, as introduced in Chapter 3.2. Thus, the MB must compute the timestamp of each daughter message, i.e. the delay δ_i between the start of the cycle at time t_0, and the moment the i-th message is transmitted at $t_0 + \delta_i$.

Second, this software layer must include functions to implement task-specific functionalities, which we call units in this work. These units compute various commands to be sent the daughter boards of the robot. These commands are processed into USB messages and sent to the MF board with their associated timestamps.

Third, the communication architecture initially developed by Nadir Bounar lacks efficiency due to the logical bug found. We must reworked it to provide an efficient and reliable way for tasks to exchange information among them.

We present here the services provided by the developed software, while the following chapters detail its use to design elementary controllers and to simulate human-like reflexes.

4.2 Downlink buffer management

In Chapter 3.2, we presented the cycle communication logic orchestrated by the MF board. As a reminder, the MB fills a downlink buffer at each cycle with messages to send to the DB, or intended to the MF board. The MB then transmits this buffer to the MF board. To avoid collisions on the bus, the MB associates, with each daughter message, the timestamp at which the MF must transmit the message. Obviously, the messages targeted to the MF does not require any timestamp.

To command the DBs, each task inserts its message in the downlink buffer using a dedicated function which also computes the associated timestamp. Two timing possibilities are available:

- the message is sent as soon as possible by the MF board, which results in collisions on the bus if multiple messages are sent in one cycle, and
- the message is sent at a specified timing.

The first method is useful for fast debugging where we do not need to pay attention to collisions. However, in practice, the second is always preferred, but requires the MB to determine the timestamp precisely. For this reason, we focus on this method in the next sections.

4.2.1 Data blocks format

Each data block in the downlink buffer consists of

- the size of this block in bytes,
- the high-level address of the receiver board as introduced in Chapter 2.1,
- the timestamp at which the message should be sent, expressed in tens of microseconds, measured from the transmission of the SYNC frame at t_0. A reserved value (0xffffffff) indicates that the MF must transmit the message as soon as possible, i.e. immediately after the SYNC message,
- the data to be sent to the DB, i.e. the application message type and its associated parameters as introduced in Chapter 2.3, and
- a Cyclic Redundancy Check (CRC) value of this block used to ensure the integrity of the transmitted data.

4.2.2 Timing computation

In Chapter 3.2.1, we presented the use of timestamps by the MF board to synchronise the transmission of daughter messages. Due to its high-level view on the robot, the MB computes these timestamps and associates them with the corresponding message. The MB must ensure that the delay between each consecutive message is sufficient to avoid collisions on the bus, in other words δ_i should be greater than the transmission time of the daughter message and its response from the DB while taking into account its precessing latency, i.e. the time elapsed from the DB receiving the message to the beginning of its response. The delay constraint is thus represented by:

$$\delta_i - \delta_{i-1} = t_{transmission} + t_{latency} + t_{response} + \epsilon \tag{4.1}$$

where ϵ represents a safety margin.

There is a trade-off on the value of ϵ: a high value ensures message transmission but reduces the number of messages than can be sent in one cycle, while a low value assumes that our expected time ($t_{transmission} + t_{latency} + t_{response}$) is close to the actual time, which could result in collisions on the bus but optimises the use of the bandwidth.

Implementation

We decided to leave the responsibility for computing the timestamps to the downlink buffering function. Centralising this computation in a single function simplifies thread synchronisation, as each thread only needs to provide its message without paying attention to the timestamps.

A variable tracks the timestamp to associate with the next message. It is initialised with a non-zero value to account for the SYNC message, which is always the first transmission of each cycle. We decided to separate each message from the next by a constant time (t_{const}), computed as the sum of the maximum possible timings of Equation 4.1, so that the delay constraint becomes:

$$t_{const} = \delta_i - \delta_{i-1} = max(t_{transmission}) + max(t_{latency}) + max(t_{response}) + \epsilon \tag{4.2}$$

This choice is taken for simplicity but has the drawback of being suboptimal as in practice the delay between each message is excessive.

The MB assigns the current timestamp to all daughter messages and then updates it by adding t_{const}. Whenever the timestamp to be assigned to the next message exceeds the cycle period, currently set to 10 ms, all the next ones of this cycle are discarded. The operator is then notified by a critical log entry. Therefore, the MB has the responsibility of ensuring that all messages can be sent during a single cycle period.

Since our CAN busses do not interfere with each other, we can schedule messages independently. Hence, each of them maintains their own timestamp variable.

To prevent concurrency issues, we use a binary semaphore whenever a thread updates the buffer ensuring mutual exclusion and preventing race condition on the shared buffer.

4.2.3 Perspectives

We intentionally designed the management of the downlink buffer to be quite simple to have a first operational system which enables communication between the MF board and MB. Multiple improvements are possible, but would require significant additional work.

First, rather than discarding messages whose timestamps exceed the period, the MB could keep them in memory and send them during the next cycle.

Second, the delay between each message is excessive and our use of the available bandwidth is suboptimal. The computation method could be improved by computing a message-dependent delay.

Third, instead of considering the messages independently of each other, the MB could group all the messages intended for the same DB, so that the MF board would send a series of frames which are answered by a the same DB. This would reduce the delays between each message, which is useful for maximising use of the bandwidth.

4.3 Units

As presented in the beginning of this chapter, the units represent functionalities of high-level software running on the MB. We divide them into categories, implemented as single or multiple threads to achieve their goal.

The organisation of the units is illustrated in Figure 4.1.

4.3.1 World

World is the robot's database, representing all the knowledge of the MB about the state of the environment and of the robot, from its low-level actuators, such as the angular position of a servomotor, to the high-level logic, such as the next action to perform to maintain a walking path. DB data are updated whenever the MB receives a daughter response, while high-level data are updated in real-time. Each servomotor is represented by a custom structure composed of its state variables, given in Appendix E.4.

We encapsulate the world database, so that it is only accessible through dedicated read and update functions.

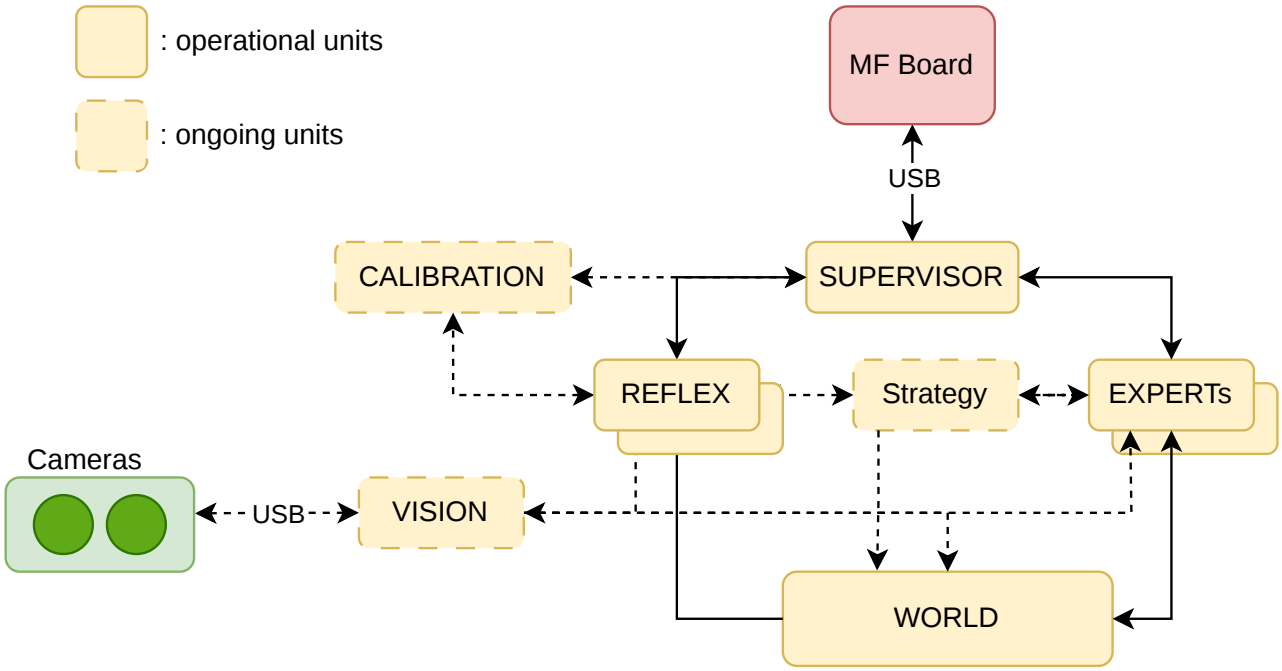


Figure 4.1: Organisation of the motherboard's units.

4.3.2 Supervisor

To orchestrate the USB communication channel, the *supervisor* receives and handles the uplink buffer from the MF board and distributes the incoming messages to each interested task. The distribution is performed using the inter-task communication interface described later in Chapter 4.4.

This task is also the one which initiates the other units, starts the ongoing dashboard and transmits the robot's state to it in real-time. It is also the one managing the downlink buffer presented in Chapter 4.2.

4.3.3 Experts

The *experts* are responsible for updating a specific piece the world knowledge. An expert receives information from the daughter boards as well as from other tasks thanks to the supervisor. For instance, the *expert-collisions* determines for each servomotor whether it collides with something using the linear acceleration data provided by the IMU of RM-28.

4.3.4 Reflex units

The *reflex* units reproduce human behaviours to achieve human-inspired locomotion. In practice, they execute algorithms to determine the state parameters of the actuators. For instance, to stand up, one reflex is responsible for maintaining the current angular position, while others gradually adjust the position of the robot to minimise the energy consumption by reducing the required torque.

Chapter 6 provides more information on the reflexes theorised and the one concretely implemented in this work.

4.3.5 Perspectives

The aforementioned units provide a solid basis on which complex functionalities can be developed. The next chapters present concrete examples using these operational units.

Still, we already planned other units for future developments which we present now.

Calibration

The state parameters of the DBs are configurable by the MB, but some of them must not be updated, they are constant during the operation of the robot. For instance, the angular limits of the servomotors are stable and could cause mechanical damage if incorrectly modified.

For this reason, constant parameters are hardcoded in the daughter boards' software to prevent external modifications, but this is not scalable as it implies that each servomotor has a different software.

A *calibration* unit would be responsible for setting those parameters once by using experts and reflexes to determine their value. For instance, by performing specific movements, the calibration could determine the mechanical structure of the robot and update accordingly the mechanically dependent configuration parameters. This unit would prevent us from manually determining constant parameters by setting them automatically.

Strategy

The current strategy followed by the robot is quite simple and does not require a dedicated unit to handle its logic, but one will become necessary as the robot's movement become more complex, requiring coordinated motion planning and decision-making.

The tasks within this unit would manage the locomotion of the robot by determining whether it should walk or run, and decomposing this decision into actions, such as raising the robot's foot to take a step.

Vision

The robot is currently blind as its cameras are not yet interfaced to the USB communication channel. Once this is done, a *vision* unit responsible for handling the information coming from the robot's cameras would be critical. For instance, this unit would determine whether there are obstacles in front of the robot or localising the ball on the field.

This unit will provide its information to the *experts* and *reflex* units which require them.

4.4 Inter-task communication architecture

To briefly remind the reader, the purpose of this architecture is to provide communication between MB tasks, which often require data produced by other tasks or received from the DBs to complete their operations, as presented in the units chapter (cf. Chapter 4.3).

Information is communicated using messages that correspond to an event or data availability, and tasks subscribe to the messages that interest them. For instance, MSG-COLLISIONS indicates that the *expert-collisions* has detected a shock, or MSG-POSITION transmits the angular position of a specific servomotor.

One crucial requirement is that the architecture must determine, for a given message type, which tasks it must notify. Iterating over all tasks to check their subscriptions results in O(n) complexity which is not scalable as the number of threads grows. For this purpose, we link the message type with the interested tasks using a hashtable.

Finally, the architecture also provides an efficient way to block the tasks until required data become available to avoid wasting CPU resources by busy-waiting as explained in Chapter 2.7.

4.4.1 Hashtable

Unlike arrays which access data using only indices, a hashtable maps keys to indices and the lookup search time complexity of the hashtable is O(1). Therefore, given a key, we retrieve the stored value in constant time, regardless of size of the hashtable.

We use a hash function to map a key to its corresponding index in the table by multiplying the key by a prime number modulo the hashtable size.

In our case, the hashtable maps each message, used as key, to the list of tasks subscribed to that message to efficiently retrieve all interested tasks.

4.4.2 Task data structure

The tasks are represented by a custom structure. It comprises:

- the identifier of the task,
- a message buffer, used to store incoming data,
- a condition variable, which manages the blocking and notification of the task,
- a mutual exclusion semaphore, which ensures exclusive access to the structure and thus prevents race conditions, and
- a waiting counter that tracks the number of messages that the task is still expecting.

The complete structure is given in Appendix E.4.

4.4.3 Message transmission

To wait for an incoming message, tasks add themselves to the array associated with that key so that they can be notified whenever data become available. They also increment their waiting counter so that the task knows how many messages it is still waiting before it can resume its execution.

To optimise the use of the motherboard CPU, the threads are blocked using RTOS facilities, as introduced in Chapter 2.7.2, until all required data are received.

To send a message, we look up the list of interested tasks in the table. The lookup achieves O(1) average complexity, while iterating over k subscribers is O(k). We then insert this message into each task buffer, unblock the threads, and decrement their waiting counter.

Chapter 5

Control

In this chapter, we introduce the control of a dynamical system using feedback. We start with general concepts before going into details of the controller designed for our actuators, the servomotors.

5.1 Theoretical concepts

In their book [53], Åström et al. define a dynamical system as: "A system whose behaviour changes over time, often in response to external stimulation or forcing". They also define the concept of feedback by a situation in which two or more dynamical systems are connected together such that each system influences the others. Their dynamics are thus strongly coupled.

We refer by closed-loop, multiple systems linked by a feedback loop, and by open-loop otherwise. Closed-loop systems provide good controllability, defined as the possibility to reach a specified target state, and high robustness to external disturbances and variations in their individual elements [53]. To give an example of such variations, consider the grease placed in the reductor of the motor to minimise friction forces. During the initial stage of the motor's operation, the grease spreads inside the gears before efficiently reducing friction forces. The motor's efficiency is thus stage-dependent, but the closed-loop system is robust to this.

However, the feedback loop can introduce instabilities due to sensor noises and oscillations if the system is not well designed. In the worst case, the system becomes uncontrollable.

The common controller senses its current state and compares it to the desired target, called the error. Based on this, it computes corrective actions to reach the objective.

In our project, we use PID controllers.

5.1.1 PID controller

A PID, Proportional-Integral-Derivative, controller is composed of three terms:

- the proportional term taking into account the current error,
- the integral term depending on past errors, and
- the derivative term anticipating future errors.

Using only the proportional term, the controller produces an action proportional to the current error. In steady state, with constant error, the computed action may be insufficient to drive the actuator to the reference. The integral term solves this issue by accumulating the error over time. As long as the error is not null, the integral grows which increases the corrective action until the reference is reached. It helps converging toward the reference even with steady-state error. The derivative term is useful to speedup convergence, by producing a corrective action proportional to the error's speed. To illustrate this concept, let us take the example of driving. If I want to go from 90 km/h to 100 km/h and I am currently accelerating, I anticipate and release the accelerator before reaching the target speed. It prevents me from going over 100 km/h, then slowing down to the target.
We want the controller to reduce this peak over the desired velocity, what is called an overshoot. The derivative term reduces the overshoot: by quickly approaching the reference, the derivative term produces a braking action to smoothly reach the reference.

Each term is weighted by a gain (K_p, K_i or K_d), which controls their influence on the corrective action. Increasing K_p makes the controller more aggressive in response to the current error, while K_i determines the speed of error reduction in steady-state, but can introduce oscillations if wrongly tuned. Finally, K_d influences the reactivity to error's variation, but also the sensitivity to noisy measurements.

Figure 5.1 illustrates a closed-loop system with PID controller.

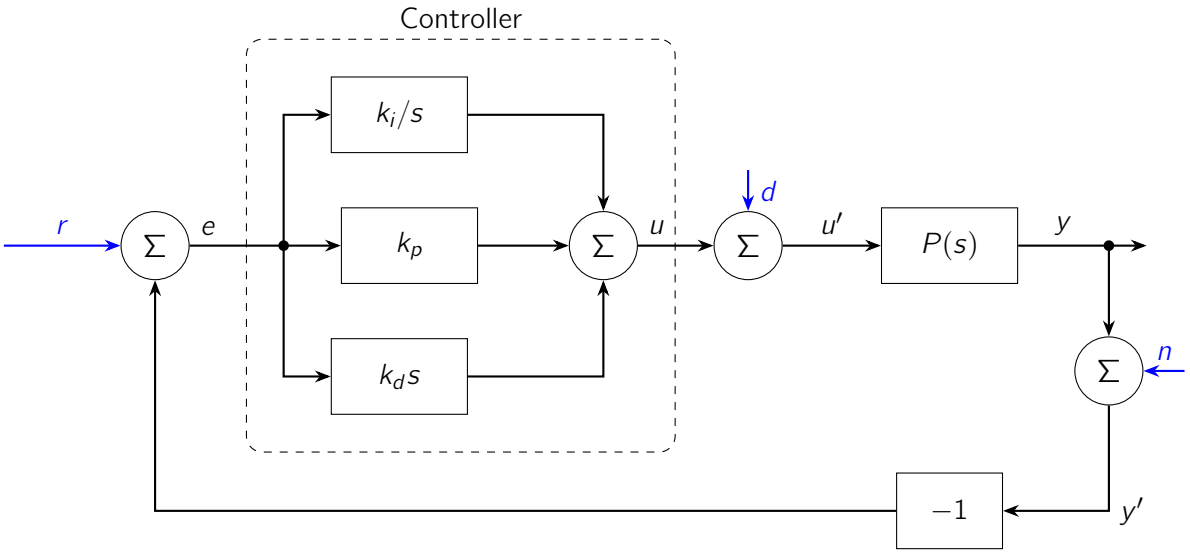


Figure 5.1: Block diagram of closed-loop system with PID controller.

The system has three inputs:

- r the reference, corresponding to the target tracked by the controller,
- d the disturbances, representing the external perturbations of the environment, and
- n the measurement noise, corrupting the information of the sensors.

The control error, e, is the difference between the reference and the measured output y'. Using it, the controller computes the corrective action u following Equation 5.1.

$$u(t) = K_p\, e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \tag{5.1}$$

After being affected by the disturbances, the action is given to the actuators P of the system. The output y is measured by sensors, but due to measurement noise, there is a difference between the actual output and the measured one y'.

In practice, disturbances are usually low frequency, and the system usually attenuates them thanks to its mechanical and electrical inertia, acting as a natural low-pass filter in robotic. For instance, an external push on the arm of the robot is attenuated by friction forces generated in the motor's gears. In contrast, measurement noise is high frequency as it affects every measured sample. Moreover, the noise directly influences the error and can introduce oscillations due to the derivative term. Indeed, the derivative response is larger when the error's slope is steep, which is typically the case with high frequency noise. This term tries to counterbalance the noise's influence, resulting in oscillations. We say that the derivative term amplifies high frequencies. For this reason, we do not usually use the derivative factor in the presence of noisy measurements.

5.1.2 Integrator windup

A system typically operates within a defined range of acceptable values. For example, a motor has limited, achievable speed and torque. If the control variable reaches the actuator limits: the system saturates. The error being typically nonzero in this situation, the integral term may become very large and the action remains saturated, even when the error reduces. In that case, the system becomes less reactive, or even unstable and oscillates.

This situation is called integrator windup and can be solved using multiple methods. In our project, we decided to use a simple but efficient solution: limiting the accumulated integral value. First, it prevents the integral term from saturating the actuator output. Second, the integral limit becomes a tunable parameter that influences the reactivity of the controller. For instance, if the error becomes very large but the corrective action does not saturate the controller yet, it can slow down the reactivity upon a reference change. By limiting the integral term, this undesirable behaviour is mitigated.

5.2 Bobby's PID controllers

In our project, we use controllers to drive servomotors to achieve higher-level objectives, for instance, moving a leg joint to take a walking step. The corrective action computed by each controller corresponds to the PWM command sent to the H-Bridge, as introduced in Chapter 2.6.

Different motion objectives require controlling different state variables. For instance, by controlling the position of each servomotor of the arm, we can move the robot's hand to reach a location in space. Moreover, some situations require the joints to apply a controlled force to maintain a position while being compliant to external disturbances. Finally, some tasks require controlling the velocity of motion, such as shooting the ball with the desired impact speed.

These practical cases motivate the use of three different PID controllers operating in angular position, angular velocity, and torque. Using a single controller or by combining them, as discussed in Chapter 5.2.7, we can achieve high-level objectives.

Our servomotors operate in a periodic cycle as explained in Chapters 2.5 and 3.2. The PID controllers compute a single corrective action per cycle by executing the following sequence:

1. Wait for SYNC message,
2. Update output using corrective action of previous period,
3. Update sensors' measure,
4. Compute corrective action,

5. Repeat.

The servomotors update their output simultaneously, which is crucial for robot's synchronisation needed by the locomotion modes covered in Chapter 6.

5.2.1 PID tuning

The performance of the PID controller depends highly on the gain values, which are fixed during the robot's operation. The gains depend on the mechanical constraints and the environment in which the actuators operate and thus correspond to the constant parameters introduced in Chapter 4.3.

To set the gains, we first gradually increased the proportional gain until we achieved a stable reference tracking. However, the controller demonstrates non zero steady-state error. Hence, we added an integral term to eliminate the steady-state error. Finally, we set the derivative term based on the tracking response analysis presented below, in Chapter 5.2.10. Nevertheless, we have not fine-tuned these parameters in this thesis for three reasons.

Firstly, manual tuning is feasible given the small number of parameters and can produce acceptable performance. Secondly, not all custom servomotors have been built yet. Only some of them were operational, and we tested our platform on a single leg as illustrated on the test bench (see Figure 1.11). The constraints imposed on these servomotors are thus different from those on the completed robot, which require us to retune the gains once the robot is fully assembled. Thirdly, the goal of this thesis is to design a platform that can simulate human-like reflexes. We concentrate our work on strengthening each developed layer, while the fine tuning of the gains is left for a future step.

5.2.2 Limits

When using a closed-loop system, it is crucial to define limits to constrain the controller. These limits can be mechanical, i.e. preventing the robot from damaging its mechanical structure, anatomical, the feet of the robot cannot rotate beyond their natural amplitude for example, preventive, such as a maximal motor's temperature to avoid overheating, or also motivated by control performance, like antiwindup limits.

We differentiate two types of limits. On the one hand, *soft* limits constrain the actuator to stay within operational range. The controller is designed to respect them, but, if the actuator reaches its boundaries, its software restricts the corrective action without interrupting the working of the robot. Positional range or antiwindup limits are part of this category. On the other hand, *hard* limits, such as the maximal motor's temperature, trigger a safety mode or a full stop of the robot, if exceeded.

Our servomotors have one *hard* limit: the motor's temperature to prevent material damage in case of overheating, and several *soft* restrictions:

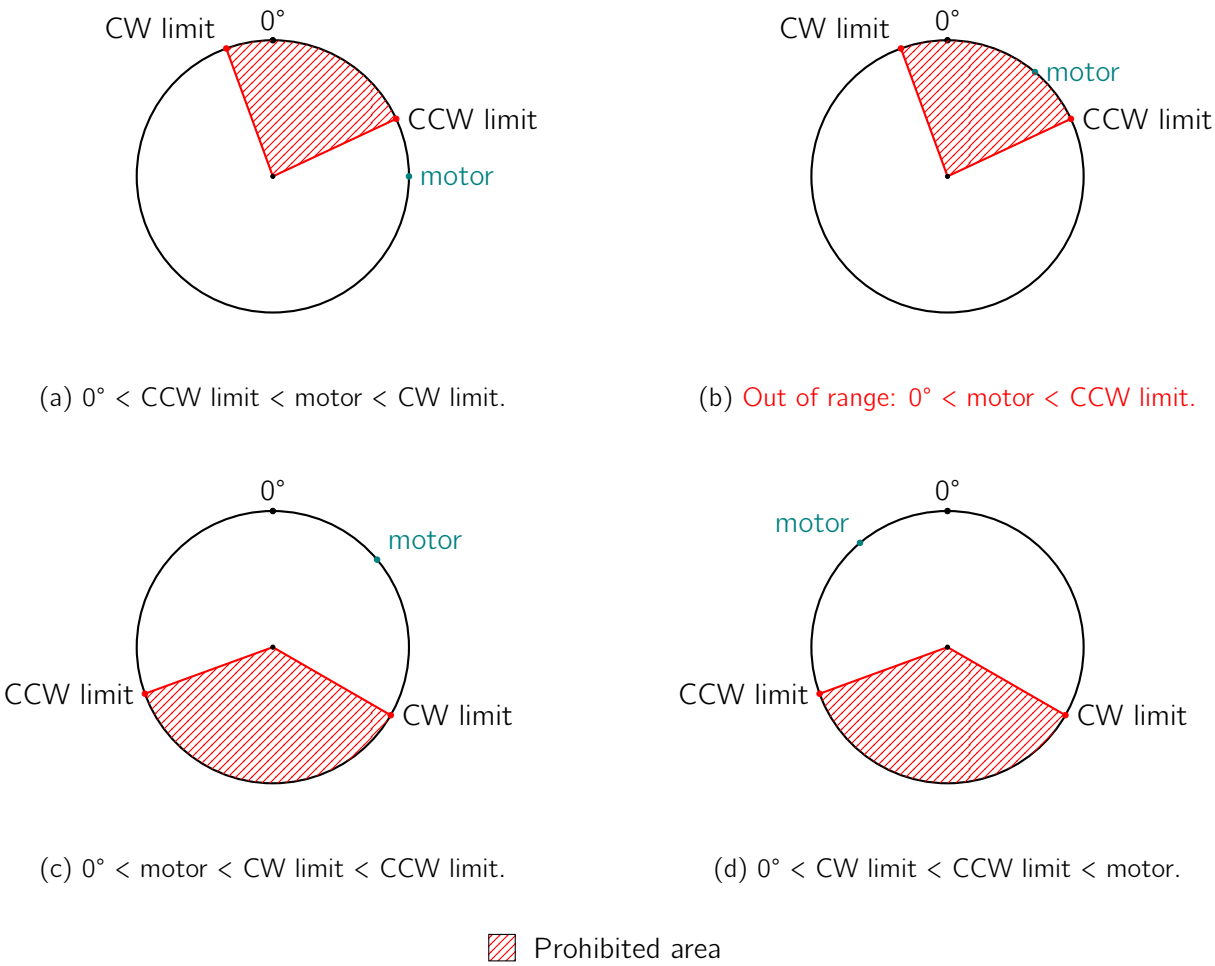
- Angular position, to prevent reaching mechanically inaccessible or anatomically impossible positions,
- Velocity, to bound the motor's speed in either direction,
- Torque, to restrict the force of the motor,
- Integral term, to prevent this term from accumulating too much error. This is an antiwindup solution as explained in Chapter 5.1.2.

We raise a flag whenever a limit is reached to inform the MB, using the reporting system introduced in Chapter 2.4.

5.2.3 Angular position limits

The purpose of angular limits is twofold. First, servomotors can generate a high torque, sufficient to damage the mechanical structure of the robot. Hence, we want to prevent the actuator from reaching a physically inaccessible area to protect the robot. Second, the robot's servomotors are capable of 360° rotation, but the RoboCup rules require the robots' joints to have the human natural amplitude. Therefore, the robot's joints cannot rotate beyond this anatomical limit.

We restrict the angular position by introducing a counter-clockwise and clockwise angle limit, beyond which the motor should not operate. Nevertheless, it is not trivial for the controller to determine whether the motor is within the acceptable range, which depends on the relative position of the limits and the motor with respect to the 0° reference. Figure 5.2 illustrates configurations in which the acceptable range is located differently relative to the motor's position.



(a) 0° < CCW limit < motor < CW limit.

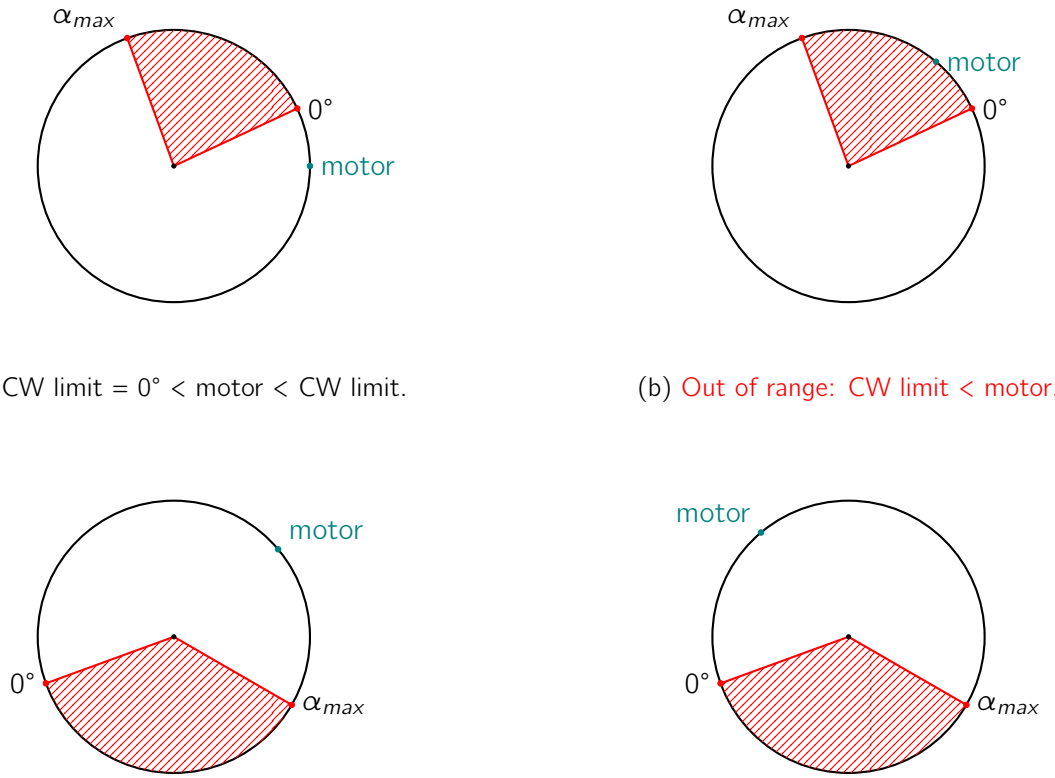
(b) Out of range: 0° < motor < CCW limit.

(c) 0° < motor < CW limit < CCW limit.

(d) 0° < CW limit < CCW limit < motor.

Figure 5.2: Position of the angular limits relative to the motor's position.

As previously mentioned in Chapter 2.5.2, the presented problem is due to the representation of angles that wrap around. Therefore, we introduce a new reference frame so that the prohibited area is located consistently relative to the motor. To do so, we represent the position of the CCW angular limit as the zero angle and the CW limit as the upper limit of the acceptable area.Figure 5.3 illustrates the same configurations as before with the new coordinate convention, and we see that the acceptable area is always located within the $[0; \alpha_{max}[$ range, regardless of the physical position of the limits.



(a) CCW limit = 0° < motor < CW limit.

(b) Out of range: CW limit < motor.

(c) CCW limit = 0° < motor < CW limit.

(d) CCW limit = 0° < motor < CW limit.

Figure 5.3: Position of the angular limits relative to the motor's position with updated reference frame.

To prevent the motor from reaching the prohibited area, we introduce a detection mechanism based on critical areas, physically close to the limits but within the acceptable range. If the motor's position is close to one of these areas and actively moves toward it, the motor is put in braking state, i.e. it generates a torque of zero. The controller then pushes the actuator back into the acceptable range, as explained in Chapter 5.2.9. The drawback of this method is that the critical zone is not fully reachable, so we further limit the accessible area. The logic is illustrated in Figure 5.4, where we enlarge the critical areas for better visualisation.

To obtain information on the torque direction, we use the PWM value applied to each gate, as stated in Chapter 2.6. Then, we compare the current position of the motor to its limits in the new reference frame. If we determine that the motor actively goes out of range, we set the PWM command to zero.

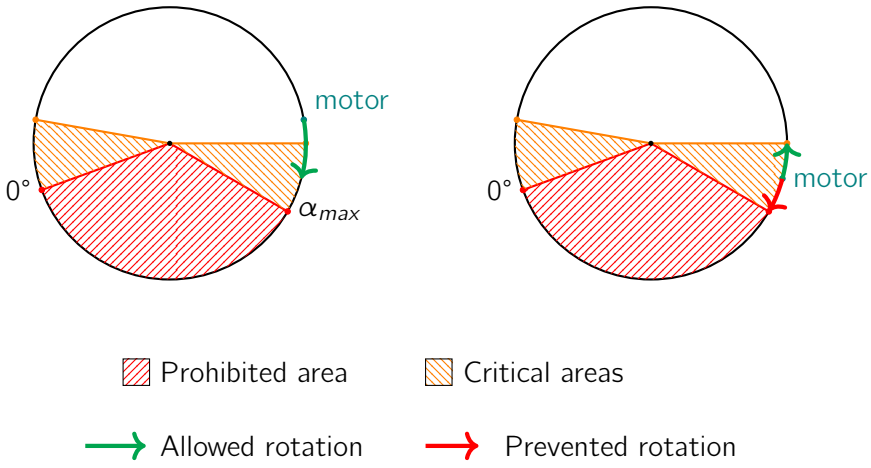


Figure 5.4: Visualisation of the critical areas and the allowed and prevented rotations.

To define critical areas, we use a constant tolerance whose value must be tuned. If too high, the accessible range is too restricted, but if too low, the motor can go over the critical zone and the detection mechanism does not activate. This is a trade-off between maximising the usable angular range and ensuring reliable limit detection.

The full angular range is thus divided as follows:

$$\begin{cases} [0^\circ;\ \text{tolerance}[\equiv \text{critical area} \\ [\text{tolerance};\ \alpha_{max} - \text{tolerance}[\equiv \text{accepted area} \\ [\alpha_{max} - \text{tolerance};\ \alpha_{max}[\equiv \text{critical area} \\ [\alpha_{max};\ 360^\circ[\equiv \text{prohibited area} \end{cases}$$

5.2.4 Velocity limits

To illustrate the utility of velocity limits, we take a concrete example outside the scope of a soccer match. Imagine the robot is carrying a glass filled with water and wants to place it on the table. Without velocity limits, the robot's hand could move too fast. The impact once putting the glass on the table could cause it to spill its contents. The velocity limit introduced in this section directly addresses such situations by bounding the maximum joint speed to ensure smooth motion.

As presented in Chapter 2.5.2, the motor rotates in CW and CCW directions. We prevent the motor from reaching a certain speed by setting a velocity limit, identical in both direction. Whenever the motor goes faster that its constraint, we stop the motor.

The solution is similar to the angular position limits where an operational range is defined with critical areas as follows:

$$\begin{cases}]-\omega_{max};\ -\omega_{limit}] \equiv \text{prohibited area} \\ ]-\omega_{limit};\ -\omega_{limit} + \text{tolerance}] \equiv \text{critical area} \\ ]-\omega_{limit} + \text{tolerance};\ \omega_{limit} - \text{tolerance}[\equiv \text{accepted area} \\ [\omega_{limit} - \text{tolerance};\ \omega_{limit}[\equiv \text{critical area} \\ [\omega_{limit};\ \omega_{max}[\equiv \text{prohibited area} \end{cases}$$

where the maximal speed is about 420 dps in either direction, at no load configuration.

Similarly as before, we use the torque direction to determine whether the motor rotates toward the prohibited area while being in the critical zone. If we detect that the motor will exceed the velocity limits, we set the PWM command to zero.

The velocity range is illustrated in Figure 5.5.

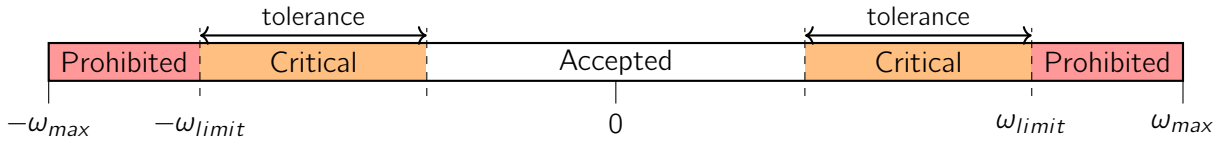


Figure 5.5: Visualisation of the velocity range.

Perspectives

The use of critical areas has a clear drawback: stopping the motor abruptly when reaching them, which increases the range of the non reachable zones. A future solution is to decrease the torque generated proportionally to the motor's position in the critical zone. This solution avoids the trade-off where the critical zones cannot be fully access by the motor since it can operate within the critical zones, but with a gradually reduced action instead of being fully blocked.

5.2.5 Torque limit

The torque limit is particularly interesting in our context because it enables human-like behaviour. Nobody always uses the maximum force of his joints. Instead, our muscles naturally limit their force to move smoothly and absorb external impact. We replicate this behaviour to allow our control to be smoother and compliant to external disturbances.

We express the torque limit by a percentage: 100% defines that the motor can produce its maximal torque and the actuator is the least compliant, while at 0% it cannot produce any torque and the actuator is fully compliant.

As presented in Chapter 2.6.2, we control the motor using a PWM command, which we restrict to limit the torque produced. The maximal absolute value of this command is 512. Hence, we compute the maximal torque by multiplying 512 by the torque percentage.

5.2.6 Temperature limit

In contrast to the previous *soft* limits of the actuator, the motor's temperature is a *hard* one: unlike the position, velocity or torque, which can be commanded, heat is produced consistently while the motor is in use. Moreover, heat dissipation is dependant on the air flow and ambient temperature, which is not controllable. If the temperature rises above a threshold, we say that the motor is overheating, and it can damage the motor and the electronic board itself.

The servomotor software periodically checks whether the measured motor temperature exceeds its maximal value, and disables the motor if the threshold is exceeded.

The motor maximal temperature is 85°C, as stated by the datasheet of the motor [45], thus we set the temperature limit to 80°C to add a safety margin of 5°C in order to let time to the software to react

and properly disable the motor by turning off the PWM signals, defined in Chapter 2.6. For the moment, there is no condition to enable the motor again. Therefore, a full restart must be performed.

5.2.7 Heterogeneous control actions

Our servomotor can activate its three PID controllers simultaneously, each computing a corrective action in a different control space (position, velocity and torque), which we refer to as heterogeneous control actions. For instance, to shoot the ball, a high-level reflex determines the angular position to reach the ball, while another computes the speed with which the robot hits the ball. However, since we control the actuator with a single PWM command's input, the servomotor must fuse the heterogeneous actions into a single value.

In this thesis, we considered two fusion methods initially proposed by T. Luksch [6]: maximum fusion, outputting the action with the highest absolute value, and normalised weighted average. The first method considers that the controller with the largest corrective action is the one the actuator follows. The second one weights the corrective actions by a modulation factor, such that the fused action depends on each controller and not only the dominant one.

We decided to use the normalised weighted average because we want each controller to matter. To take the same example as before, if the MB determines that it is more important to hit the ball fast, regardless of the precision of the shoot, the MB would attribute a higher weight on the velocity controller. The actuator is still driven by the positional goal, but considers it as a secondary objective. The weighted average method is illustrated in Figure 5.6.

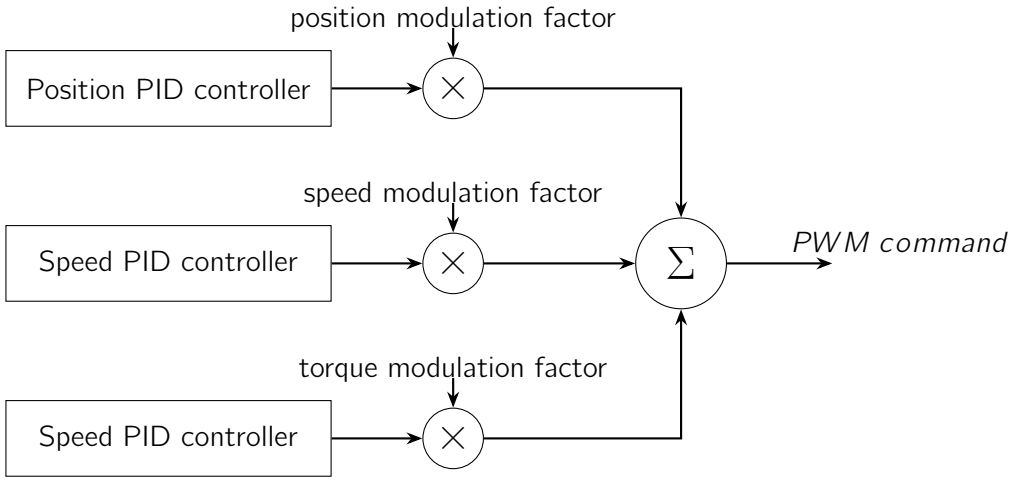


Figure 5.6: Block diagram of normalised weighted average of PID controllers.

5.2.8 Homogeneous reference fusion

While the heterogeneous fusion systematically merges different actions using weighted average, another fusion problem arises for the PID reference. Each high-level reflex computes the reference for one or more controllers. The servomotor, acting as a passive component, expects a single reference per controller. The last reference received is then always considered as the one to follow. Hence, the motherboard must fuse different references into a single value. We refer to this situation as homogeneous reference fusion.

Both the maximum fusion and normalised weighted average methods introduced in the previous section are applicable here. Unlike heterogeneous fusion, where the normalised weighted method is systematically

applied, the motherboard's choice is dependent on which reflexes are active and their relative priorities. For instance, if the robot is about to collide with an opponent, its safety reflexes determine the joints positions to protect itself. Other reflexes are responsible for the arms positions in order to maintain balance and compute other target positions. In this case, the MB uses the maximum fusion method and prioritises the safety reflexes which likely generates larger corrective actions. In another context, when the robot is standing, multiple reflexes compute different torque references for each joint in order to optimise their energy consumption. To account for all reflexes, the MB fuses the computed references with a normalised weighted average.

5.2.9 PID controllers implementation

We base our implementation of the PID controllers on the servomotors' responsibilities introduced in Chapter 2, and the communication cycle described in Chapter 3.2.

Using WRITE application messages covered in Chapter 2.3, the MB configures the controller by setting the reference, modulation factor, gains, and limit values. Then, to efficiently enable and disable PID operations, we developed three dedicated application messages: PID_POS, PID_SPE, and PID_TOR, taking as argument a Boolean value that enables or disables the corresponding controller.

Following the commands received from the MB, the servomotor configures its controllers. For this purpose, we developed an interface providing functions to initiate and manage a custom PID structure which comprises all parameters configurable by the MB. Using this structure, we implemented a corrective action computation, applicable to each controller. For the velocity and torque controllers, this general algorithm is sufficient. The angular position, however, requires additional handling due to the circular nature of the angles representation and the limit mechanism:

- Using the coordinate convention defined in Chapter 5.2.3, the actuator unambiguously determines its position relative to its angular limits. Within the valid range, the controller follows the shortest path to the reference, to minimise energy consumption and travel time. However, following this path without additional checks may lead the actuator to reach a critical area and become blocked.
- The angular limit mechanism only constraints the actions of the controller: the software does not prevent the motor to go out of range due to external forces, such as gravity. In case the actuator enters the prohibited area, the controller must drive the actuator back into the valid range, without causing mechanical damage.

These two issues are solved with distinct methods presented below.

Shortest path issue

It is important to recall the meaning of the PID error, which is defined as the difference between the reference and the measured output. Thus, the error is an angle, in this case, and must be computed following the angular difference algorithm introduced in Chapter 2.5.2. This algorithm returns the shortest signed distance between the motor's position α and the target. The error's sign and magnitude provide information on whether the shortest path crosses the prohibited area, as illustrated in Figure 5.7. Whenever the path planned crosses the prohibited area, the controller takes the longer path instead. This is done by replacing the error by its coterminal angle.

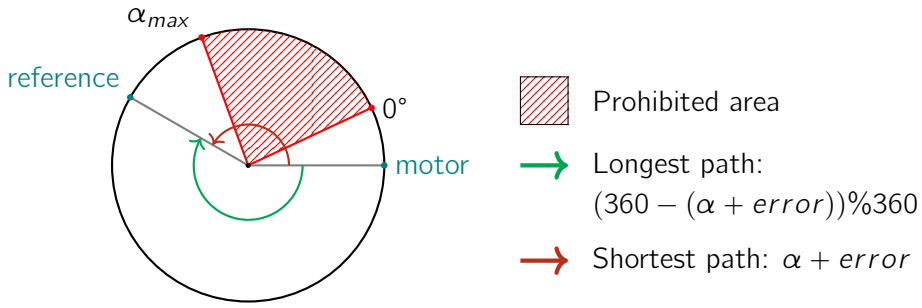


Figure 5.7: Short and long path representation.

Out of range recovery

In case the motor enters the prohibited zone, the controller stops following the reference provided by the MB and enters a recovery mode until the motor returns into the valid range. Instead, it sets its closest angular limit as target. We assume that the path from the actuator's position to the closest limit is free from mechanical obstacles. This is reasonable since the motor encounters the angular limits before the hard mechanical stop of the structure. Therefore, a valid path is always available, as illustrated in Figure 5.8.

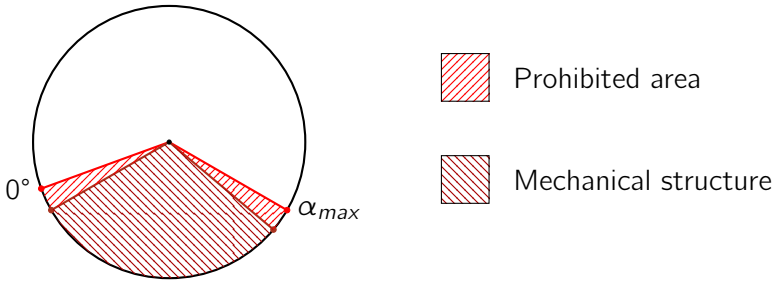


Figure 5.8: Mechanical structure's range representation.

5.2.10 Elementary controllers

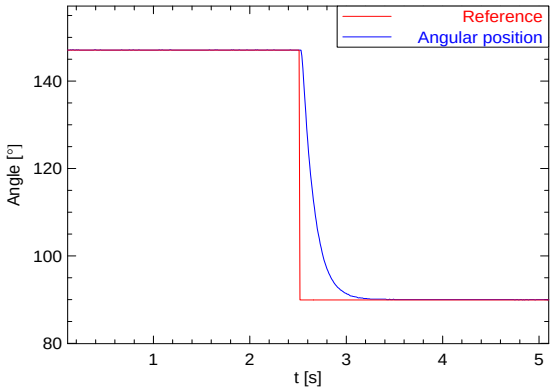
We developed and tested all the concepts and methods introduced in this section, resulting in elementary controllers able to track position, velocity, and torque references. We evaluate the controllers' performance by analysing their step response and the position reference tracking, including gradual and abrupt reference changes.

We tested the PID controllers on the servomotors of a robot leg assembled on the test bench (cf. Figure 1.11 in Chapter 1). The actuators are thus subjected to a real environment, which is harsher than a simulated environment. Moreover, since the servomotors are all mounted on a single mechanical structure, the motors must generate a significant torque to reach the PID target due to the weight of the structure and joints. This configuration therefore demands more from the controller than a no-load case.

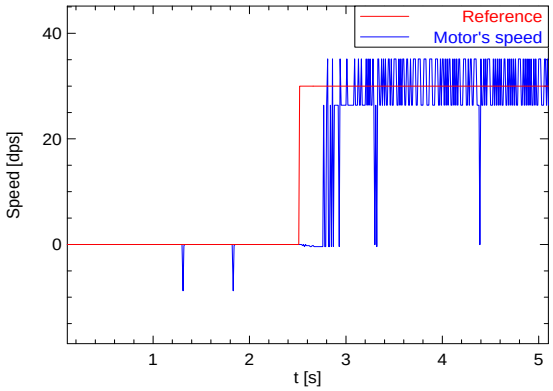
Step response

Figure 5.9 illustrates the step response, i.e. the response to an instantaneous reference change, of the developed PID controllers. We observe that each of them has a short rise time, i.e. the time to reach the reference, and almost no overshoot. However, while the settling time, i.e. the time to stabilise, is relatively short for the position and velocity controller, it is longer for the torque.

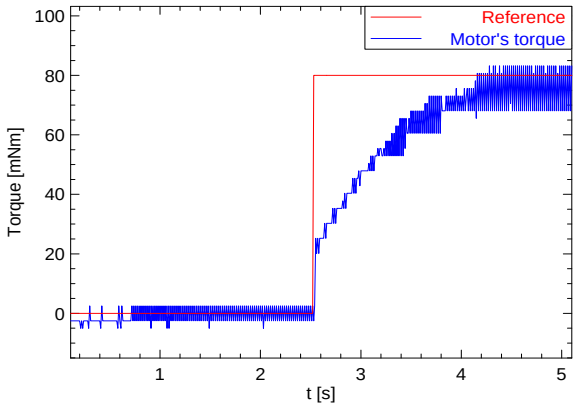
Moreover, the response of the velocity and torque PID has higher oscillations than the position controller, due to the measurement noise as discussed in the theoretical concepts of this chapter. This noise affects more the velocity and torque than the position controller, since they rely on indirect measurements. On the one hand, velocity is derived from the angular position measurement, and the derivation amplifies the high frequency noise. The torque is estimated from the current consumption, which cannot fully represent the actual torque produced as explained in Chapter 2.5.6. Nevertheless, both are sufficiently accurate for the controller to drive the motor reliably for the purpose of this platform. On the other hand, the angular position of the motor is directly measured by the magnetometer, providing a less noisy signal. The resulting controller shows less oscillation, and therefore better performance than the other two.



(a) Position controller: $K_p = 10$, $K_i = 5$ and $K_d = 0.3$.



(b) Velocity controller: $K_p = 1$, $K_i = 10$ and $K_d = 0$.



(c) Torque controller: $K_p = 0.15$, $K_i = 0.4$ and $K_d = 0$.

Figure 5.9: PID controllers' response to step input.

To remind the reader, in Chapter 5.1.1, we discussed the influence of each gain and, in Chapter 5.2.1 we presented the importance of gain values on controllers' performance. We observe that the resulting controllers confirm the theoretical statements. Unlike the others, the position controller performs better with a derivative term, since its measure is less noisy than the speed or torque. For comparison, Figures F.1

and F.2, in Appendix F, illustrate the step response of the position and velocity controller with suboptimal tuning.

We also observe the diversity of gain values used between the controllers, where each of them must be tuned independently of the others. This is expected as the gains determine the influence of the error on the action and error's range is highly dependent of the measure itself. For instance, the angular position goes from 0° to approximatively 360°, the maximal error value is thus $\pm$360°, while the maximal torque is about $\pm$2065.1 mNm (cf. Chapter 2.5.6), thus the maximal error is $\pm$4130.2 mNm. The torque error range is greater than the position range by a factor 10. Therefore, it is expected that the gains are smaller by a factor 10 as well.

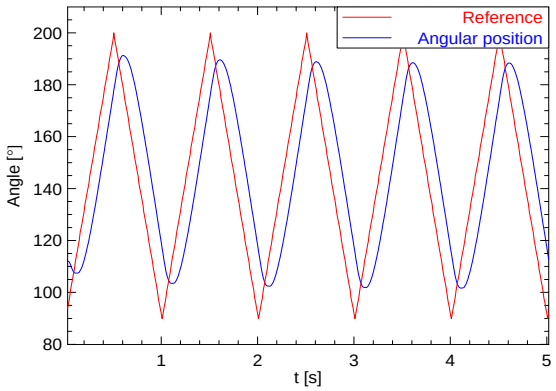
Reference update

Due to noisy measurements, the velocity and torque controllers do not perform well enough to achieve good reference tracking under gradual and abrupt reference update. However, it is worth exploring the position controller's capabilities.

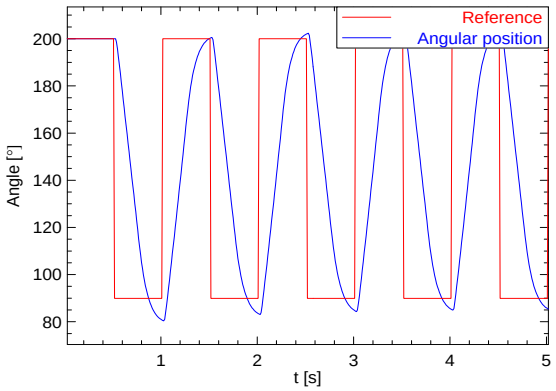
In Figure 5.10, we observe that the position controller follows the reference accurately considering the manual PID tuning performed. Unlike the step input configuration, the controller does not use the derivative term. The performances with and without this term, while keeping the rest identical, are similar but the derivative term induces a more aggressive response at the peaks of the reference. It results in higher mechanical stress with no major improvements, therefore we prefer omitting the derivative term.

We can observe that with an abrupt reference change, the controller has a higher overshoot when reaching 90° than 200° (cf. Figure 5.10b). It is expected as the 90° position is oriented toward the ground and thus the controller has to compensate for the force of gravity when returning to 200° which explains the large overshoot observed at 90°.

This two observations illustrate the trade-off in PID tuning: gains tuned for a step input may be suboptimal in reference tracking due to the nonlinearity of the environment, for instance of the gravity influence.



(a) Position PID controller: $K_p = 10$, $K_i = 5$ and $K_d = 0$ with gradual reference update.



(b) Position PID controller: $K_p = 10$, $K_i = 5$ and $K_d = 0$ with abrupt reference update.

Figure 5.10: PID controllers' response to reference update.

Conclusions and perspectives

Based on the results obtained, we conclude that the objective of developing elementary controllers is fulfilled. Each controller is able to reach a reference in a real environment with a short rise time and reasonable settling time, demonstrating their suitability for the platform. However, due to their noisy measurement, the velocity and torque controllers have limited performance. The effect of noise could be mitigated by filtering the measurement signal, but could degrade the performance of the controllers [54]. Future work could investigate whether signal filtering could enhance velocity and torque controllers.

The angular controller demonstrates strong tracking capabilities by being able to follow gradual and abrupt reference changes, allowing precise angular control for the future.

The limit mechanisms and controller logic presented in this chapter are detailed as pseudocode, respectively, in the Appendices E.2 and E.3. Moreover, the implementation of the PID controllers can be found in the `stm32/src/controllers/` folder of the project repository.

Chapter 6

Biologically inspired locomotion system

The biologically inspired control system proposed by Tobias Luksch [6] was tested and validated in a simulation environment ([27]) that abstracts hardware and logical constraints inherent to a physical platform. Deploying this system on a physical robot is more complex than in a simulated environment and requires significant adaptation to transfer the biologically inspired methods to our platform. We must take into account the restrictions of Bobby, such as a limited bandwidth and finite computing resources. Thus, the computing units of the robot cannot be aware of the global situation.

These restrictions lead to multiple trade-offs that we addressed while developing the platform. For instance, measuring data requires us to select and integrate dedicated sensors, which have limited accuracy and need to fit within the electronic boards. For this reason, some quantities are not directly measured but are derived or estimated from others such as the motor's speed (cf. Chapter 2.5.2) and torque (cf. Chapter 2.5.6).

This chapter contains the adaptation of the bio-inspired locomotion methods developed by T. Luksch to the Bobby platform. Through the reproduction of a human reflex, this chapter demonstrates that all layers developed, from the hardware layer to the PID controllers, operate correctly together.

6.1 Control units

In this chapter, we propose an abstraction of the bio-inspired control system, which we refer to as control units. Unlike the software units introduced in Chapter 4.3, which represent software threads of the motherboard, the control units are similar to neurons, the fundamental control units of the human nervous system.

A neuron is only active if stimulated, and its action can be inhibited to reduce its influence. In our project, the MB stimulates (enables) the control units and assigns them a weight used to modulate their action.

6.1.1 Locomotion Mode

We define modes representing locomotion states of the robot, such as standing, walking, running, and falling. These modes determine the generated pattern described in the next section, but also influence the weight attributed to the reflexes, i.e. the neuron inhibition introduced above.

We introduced methods in Chapter 5.2.8 to fuse multiple references targeted to the same controller where one solution is to compute the weighted average of the target values. These weights are initially attributed

depending on the locomotion mode which can then be modified depending on the situation. However, these default weights are not fixed and the MB can update them dynamically within a given locomotion mode depending on the situation. For instance, if a reflex is important while standing still, a default higher weight is attributed in this mode, which can be decreased in some particular cases. If this reflex is less important for walking, a lower default weight is assigned instead.

The strategy unit, introduced in Chapter 4.3.5, takes the decisions to transition between locomotion modes and to update the weights based on information provided by the experts.

6.1.2 Spinal Patterns generator

To achieve the chosen locomotion mode, the robot's joints are coordinated using spinal pattern generators (SPGs), each represented by a finite-state machine (FSM). The state transitions are triggered by sensory events and each state represents a locomotion phase, called a motion phase.

SPGs and motion phases are managed by the strategy unit that determines which experts and reflexes are stimulated, as well as the actuator limits based on the current motion phase. For instance, depending on the walking phase, the strategy unit updates the torque limit of the servomotors to adjust the compliance to external disturbances, as presented in Chapter 5.2.5.

6.1.3 Reflexes

Human reflexes provide feedback mechanisms for motion control. They range from very localised actions affecting a single DB to simultaneous actions triggered by distributed sensory events.

In our project, each bio-inspired reflex is implemented as *reflex* units (cf. Chapter 4.3.4) and controls the DBs using PID controllers (cf. Chapter 5). We divide them into three categories based on two criteria: (1) the sensory event triggering the action and (2) the DBs executing this action:

- Local reflexes in which both the sensory event and the resulting action concern the same DB, for instance a servomotors reacting to disturbances measured by its IMU,
- Intermediate reflexes involving DBs within the same quadrant (cf. Chapter 2.1): either the sensory event comes from a neighbouring DB, or the resulting action is performed by multiple DBs in that quadrant, and
- Postural reflexes triggered by global sensory events that result in actions executed by any daughter boards.

This division of reflexes has currently a low impact on their implementation, as the DBs act as slaves of the motherboard: even for a local reflex, the MB first receives the signal and then determines the action to take. This means that the local reflexes are not faster than the others.

Only intermediate reflexes have facilities that impact their implementation: a single multicast command can target multiple DBs simultaneously, introduced in Chapter 2.2.

Nevertheless, as mentioned in Chapter 2, future work will enhance the system by delegating more responsibilities to the DBs. Local reflexes will then represent actions that the DBs would perform autonomously using their sensors without notifying the motherboard beforehand.

6.2 Hierarchical organisation

The presented locomotion system is illustrated in Figure 6.1, which, for clarity, focuses on the control units. We observe the cascading stimulation of each element of the *strategy* unit that ends with the *reflex* units and *experts* computing corrective actions and commands. These commands are inserted into the downlink buffer by the *supervisor*, which then transmits the buffer to the DBs through the MF board at the scheduled timestamps, as explained in Chapter 3.2. Finally, the DBs execute the command and transmit their response to the MB through the uplink buffer, which provides feedback on which the *strategy*, *reflex* units, and *experts* based their next computation.

It is important to note that although only one motion phase is stimulated simultaneously for each SPG, multiple locomotion modes can be active at the same time. For instance, at the beginning of the walking phase, the standing mode is still stimulated but becomes progressively inhibited.

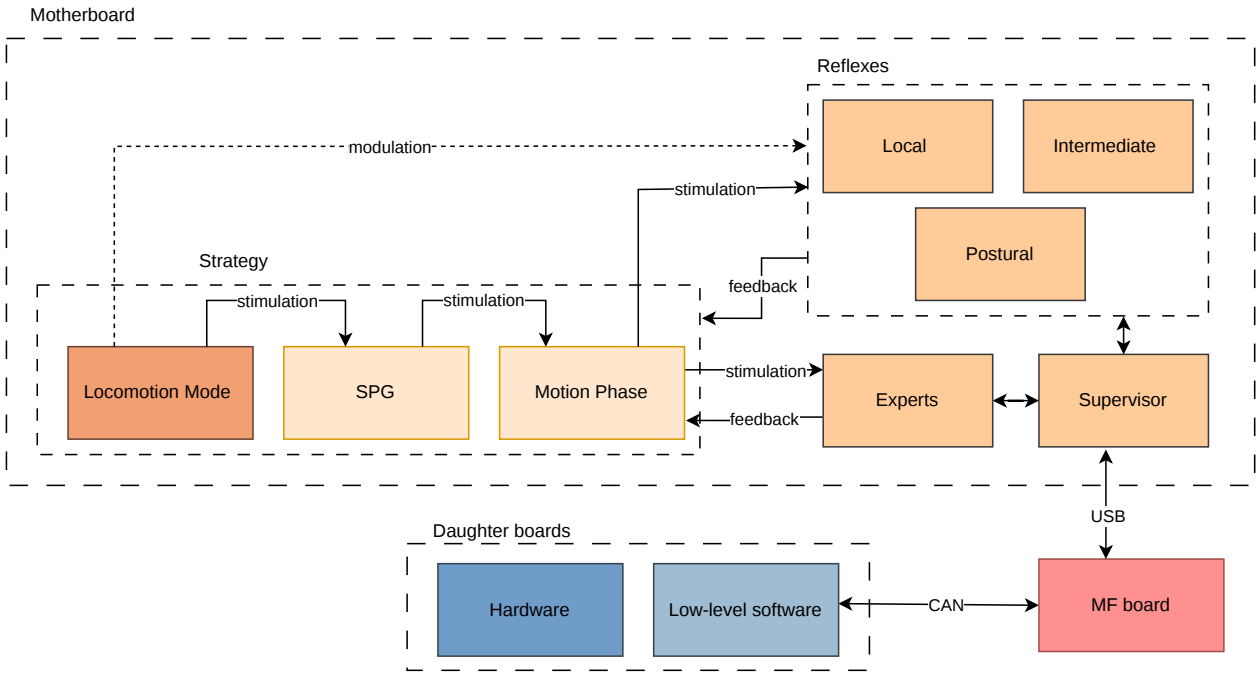


Figure 6.1: Hierarchical organisation of the locomotion system.

6.3 Stable Standing

Before introducing dynamic locomotion modes, it is important to define how our robot achieves a stable standing posture before initiating its walking. The FSM representing the *stable standing* mode contains four phases. Before touching the ground, the *Ground adaptation* initiates the robot into a position suitable for accepting the terrain regardless of its geometry. The *Stabilisation* phase compensates the external disturbances to stop the motion of the robot and thus stabilise it. This aforementioned mode does not take into account the energy cost as it is the responsibility of the *Optimisation* phase. It determines the joint angles required to reduce the necessary torque and lower the energy consumption. Finally, the robot enters and stays in the *Relaxed posture* phase until external disturbances are applied to the robot.

6.3.1 State transitions

The robot always starts its operations by adapting its posture until both feet touch the ground. Since the robot is not equipped with foot pressure sensors, we use the IMU of the ankle servomotors to detect that a collision occurred. This information is then interpreted as a contact with the ground or as a external chock depending on the cases.

The robot enters the stabilisation state when it touches the ground or when disturbances are sensed. The robot maintains its posture by increasing the stiffness of the servomotors. Using servomotors' gyroscope, embedded in the IMU, we determine that a quiet stance is achieved by measuring the angular velocity of each joint. If the angular velocity stays below a certain threshold, the robot transitions to the next state.

To optimise its posture, multiple reflexes determine the next angular position to reach. Using the current consumption sensor of each servomotor, we determine the robot's energy consumption and once this consumption becomes sufficiently low, the final state is reached.

Finally, the robot relaxes its joints and remains in this state until the *expert-collisions* detects a disturbance. In that case, the robot returns to the *Stabilisation* phase.

The finite-state machine is illustrated in Figure 6.2.

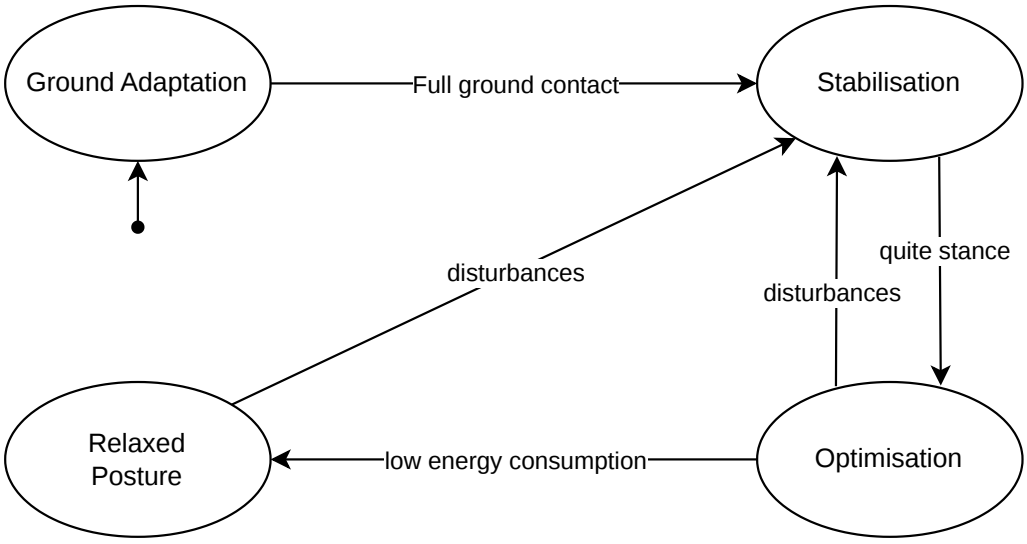


Figure 6.2: Stable Standing FSM inspired from [6].

6.3.2 State description

Each standing state is implemented using reflexes for all desired behaviours.

Ground adaptation

The *Ground adaptation* controls the joints compliance depending on whether the robot touches the ground. The logic of this state is executed by the local *Hold* reflex, stimulated across all servomotors.

Initially, this reflex sets a low torque limit to all servomotors, so that the joints can passively adjust to external forces. Moreover, a PID control in position (cf. Chapter 5) is activated with a reference set to the motor's current position. As the motor moves passively, the reference thus tracks the current position, preventing the controller from generating any active corrective torque. The use of this controller is crucial as the motor requires an active controller to simulate compliance. Therefore, disabling it would make the torque limit ineffective.

Once one foot touches the ground, the torque limit of this leg's joints is set to an intermediate value to prevent the robot from collapsing on the ground. Moreover, their position references are fixed to their current positions to maintain the robot's posture.

Finally, the torque limits of all servomotors are raised to a high value once both feet touch the ground to stop the motion the robot.

The pseudocode of the *Hold* reflex is illustrated by Algorithm 6.1.

```
1: compliance ← high compliance

2: /* Initiate position controller */

3: if no ground contact then
4:     α_ref ← α_motor // Update angular position reference
5: else if single ground contact then
6:     compliance ← intermediate compliance
7: else if full ground contact then
8:     compliance ← low compliance
9: end if
```

Algorithm 6.1: *Hold* reflex pseudocode.

As introduced above, the *supervisor* stimulates the *expert-collisions* presented in Chapter 4.3.3 to detect contact with the ground. This expert uses the IMU's data of each ankle and computes the difference between the 3-axis current linear acceleration with its previous value. It determines that a collision occurred if the difference is greater than a tolerance value.

The *expert-collisions* is illustrated by Algorithm 6.2.

Require: tol

1: $a_{x,prev}, a_{y,prev}, a_{z,prev} \leftarrow$ get_linear_acc()
2: **while** True **do**
3: $a_x, a_y, a_z \leftarrow$ get_linear_acc() // Update linear acceleration
4: **if** $(|a_x - a_{x,prev}| \geqslant tol)$ || $(|a_y - a_{y,prev}| \geqslant tol)$ || $(|a_z - a_{z,prev}| \geqslant tol)$ **then**
5: /* *Collision detected* */
6: **end if**
7: $a_{x,prev}, a_{y,prev}, a_{z,prev} \leftarrow a_x, a_y, a_z$
8: **end while**

Algorithm 6.2: *expert-collisions* pseudocode.

A dedicated pressure sensor would provide a more accurate signal by measuring the ground reaction force. Indeed, using the IMU, the expert cannot differentiate the robot touching the ground from being hit by an opponent. However, it remains a reliable information, as it is unexpected that the robot collides with something while being in the air.

Stabilisation

The *Stabilisation* phase starts whenever both feet touch the ground or if external forces disturb the standing of the robot, such as being pushed by an opponent, for instance. As for the *Ground adaptation* phase, the *Hold* reflex is also stimulated, sets all torque limits to a high value and enables a position PI controller. The robot tries to maintain its current posture by being stiff against external disturbances.

Optimisation

We consider that the robot is in a quiet stance, and thus transitions the *Optimisation* phase, once the position controller of each servomotor has reached its target. The MB determines this by comparing the reference α_{ref} with the measured motor's position α_{motor}:

$$|\alpha_{ref} - \alpha_{motor}| \leqslant tolerance_{pid} \Rightarrow \text{reference reached} \tag{6.1}$$

In addition to the *Hold* reflex still stimulated, two types of postural reflexes compute the next position of the joints to lower the total energy consumption:

- Reflexes relaxing the joints by reducing the torque generated, and
- reflexes releasing mechanical tension due to opposing target positions between paired joints.

Relax joint reflexes use the motor's position and the corresponding torque of each servomotor to compute a change rate of their position proportional to the torque generated.

Releasing tension reflexes adjust the position of two paired joints, both ankles for instance, to minimise the torque discrepancy between the two. Similarly as before, the reflexes compute a change rate of the angular position, here proportional to the difference between one joint's torque and the average torque of both joints.

The pseudocodes of the relax joint and release tension reflexes are given by Algorithms 6.3 and 6.4.

Require: $\alpha_{motor}, torque_{motor}, torque_{max}, \beta$

1: $\Delta\alpha \leftarrow \beta \times \alpha_{motor} \times \dfrac{torque_{motor}}{torque_{max}}$
2: $\alpha_{ref} \leftarrow \alpha_{ref} - \Delta\alpha$

Algorithm 6.3: *Relax joint* reflex pseudocode.

Require: $\alpha_{L,motor}, torque_{L,motor}, torque_{L,max}, \alpha_{R,motor}, torque_{R,motor}, torque_{R,max}, \beta$

1: $torque_{avg} \leftarrow \dfrac{torque_{L,motor} + torque_{R,motor}}{2}$
2: $\Delta\alpha_L \leftarrow \beta \times \dfrac{|torque_{L,motor} - torque_{avg}|}{torque_{L,max}}$
3: $\Delta\alpha_R \leftarrow \beta \times \dfrac{|torque_{R,motor} - torque_{avg}|}{torque_{R,max}}$
4: $\alpha_{L,ref} \leftarrow \alpha_{L,ref} - \Delta\alpha_L$
5: $\alpha_{R,ref} \leftarrow \alpha_{R,ref} - \Delta\alpha_R$

Algorithm 6.4: *Release tension* reflex pseudocode.

where β is a modulation constant and determines the velocity of adaptation. The higher β, the faster the optimisation but it can lead to dynamic effects disturbing the stance.

If multiple reflexes compute a new reference position for the same servomotor, the MB must fuse these references using the methods detailed in Chapter 5.2.8. For instance, using the normalised weighted average, we have:

$$\alpha_{ref} = \frac{\gamma_{hold} \times \alpha_{hold,ref} + \gamma_{relax} \times \alpha_{relax,ref} + \gamma_{release} \times \alpha_{release,ref}}{1} \tag{6.2}$$

Relaxed posture

The last phase of the stable standing mode is the *Relaxed posture*, reached when the robot achieves a low energy posture. The energy threshold is fixed and determined experimentally. The robot is manually placed in an optimised stance, and we record the energy consumption as the threshold value.

This phase stimulates only the *Hold* reflex and slowly reduces the servomotors torque limit to take a relaxed posture. If the stance is not stable or disturbed by external forces, which is determined by the *expert-collisions*, the phase transitions back to *Stabilisation*.

6.4 Platform validation by reproduction of a human reflex

To demonstrate that the Bobby platform, with all the developed layers, is capable of reproducing human reflexes and thus of achieving human inspired locomotion, we implemented the *Ground adaptation* and *Stabilisation* phases described in Chapter 6.3.1. Both phases stimulate the *Hold* reflex, which requires the coordinated operation of all layers of the Bobby platform.

We conducted an experiment with a single leg of Bobby released at a small height in the air, where only the knee joint and one servomotor of the ankle are controlled as shown in Figure 6.3.

This section follows the execution of the experiment by showing log entries (cf. Chapter 4.1) to illustrate the decisions and actions taken by each component.

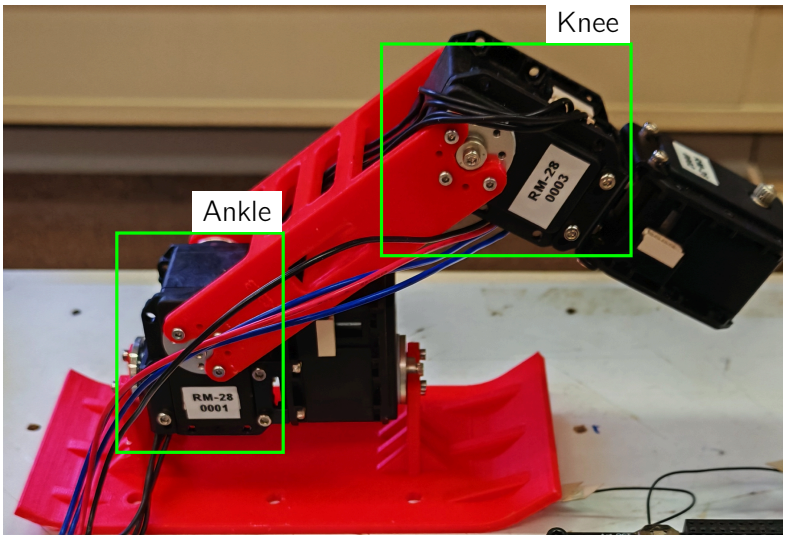


Figure 6.3: Servomotors used for the reproduction of a human reflex.

6.4.1 Initialisation

First, the MF board initiates the cycle by broadcasting a SYNC message to all DBs and by sending an uplink block to the motherboard, which demonstrates that the USB and CAN communication channels are correctly implemented. Upon receiving the uplink block, the MB starts a new cycle of operation and notifies the *expert-collisions* and the *Hold* reflex, both executed by threads running in parallel on the MB:

```
[18:35:42.984487799 27/05/2026] [src/reflex/hold_reflex.c:427] [DEBUG] Hold reflex started
[18:35:42.993893805 27/05/2026] [src/expert/expert-collisions.c:93] [DEBUG] Expert collisions started
```

6.4.2 Ground contact detection

The *expert-collisions* (cf. Algorithm 6.2) is stimulated on the ankle to determine whether a collision occurred.

The operation of this *expert* requires the linear acceleration data of the ankle, and the *Hold* reflex is responsible for sending the GET message to retrieve this quantity (cf. Chapter 6.4.3). Once an external disturbance is detected:

```
[18:35:44.033952073 27/05/2026] [src/expert/expert-collisions.c:59] [NORMAL] Expert detects a collision for servomotor 1
```

the *expert* transmits this information to the *Hold* reflex which interprets it as the foot having touched the ground:

```
[18:35:44.044046792 27/05/2026] [src/reflex/hold_reflex.c:313] [NORMAL] Hold reflex determines that the robot touched the ground
```

By analysing the log entries, we observe that the *Hold* reflex interprets the collision information approximatively 10 ms later, i.e. one cycle later (cf. Chapter 3.2). This latency is a known limitation of the current implementation as the inter-task communication, presented in Chapter 4.4, is not yet operational and its integration to the Bobby platform is left as a future work. Instead, a shared variable *touched_ground*

is used. Therefore, if the *Hold* thread checks the *touched_ground* value before being updated by the *expert*, there is a delay of one cycle to detect the ground.

6.4.3 Hold reflex

Following the pseudocode presented by Algorithm 6.1, the *Hold* reflex initiates the position controller of each servomotor by setting their PID parameters, namely the modulation factor, gains, integral limit, and torque limit (cf. Chapter 5.2). The reflex configures each controller using WRITE messages (cf. Chapter 2.3). It is important to note that each servomotor is tuned differently as the gains are dependent of the actuator's constraints and environment as stated in Chapter 5.2.1.

The log entries confirm that each command was received successfully by the servomotors:

```
[18:35:43.014098257 27/05/2026] [src/supervisor/operations.c:341] [NORMAL] WRITE:
    Servo ID 2 received position modulation factor = 100 %

[18:35:43.014153513 27/05/2026] [src/supervisor/operations.c:341] [NORMAL] WRITE:
    Servo ID 1 received position modulation factor = 100 %

[18:35:43.014167208 27/05/2026] [src/supervisor/operations.c:312] [NORMAL] WRITE:
    Servo ID 2 received position proportional gain = 30.000000

[18:35:43.014216569 27/05/2026] [src/supervisor/operations.c:312] [NORMAL] WRITE:
    Servo ID 1 received position proportional gain = 10.000000

[18:35:43.014228113 27/05/2026] [src/supervisor/operations.c:322] [NORMAL] WRITE:
    Servo ID 2 received position integral gain = 30.000000

[18:35:43.014358710 27/05/2026] [src/supervisor/operations.c:322] [NORMAL] WRITE:
    Servo ID 1 received position integral gain = 5.000000

[18:35:43.014459696 27/05/2026] [src/supervisor/operations.c:350] [NORMAL] WRITE:
    Servo ID 2 received position integral limit = 10

[18:35:43.014476671 27/05/2026] [src/supervisor/operations.c:350] [NORMAL] WRITE:
    Servo ID 1 received position integral limit = 10

[18:35:43.014567679 27/05/2026] [src/supervisor/operations.c:302] [NORMAL] WRITE:
    Servo ID 2 received max torque = 25 mNm

[18:35:43.014651780 27/05/2026] [src/supervisor/operations.c:302] [NORMAL] WRITE:
    Servo ID 1 received max torque = 25 mNm
```

For the experiment, we use PI and not PID controllers according to the results obtained in Chapter 5.2.10, where the derivative gain induces a more aggressive response.

After the configuration messages, the reflex queues GET messages to retrieve the angular positions and linear accelerations of the servomotors[1]:

```
[18:35:43.014730329 27/05/2026] [src/supervisor/operations.c:125] [NORMAL] GET: MB
    received angle = 3.603900° from servo ID 2

[18:35:43.014811305 27/05/2026] [src/supervisor/operations.c:125] [NORMAL] GET: MB
    received angle = 95.723099° from servo ID 1

[18:35:43.014903882 27/05/2026] [src/supervisor/operations.c:142] [NORMAL] GET: MB
    received lin_acc x = -9.702621 m/s² from servo ID 2
```

[1] Note that only the ankle's linear acceleration is used for the moment.

```
6
7 [18:35:43.014917593 27/05/2026] [src/supervisor/operations.c:143] [NORMAL] GET: MB
    received lin_acc y = 0.257316 m/s^2 from servo ID 2
8
9 [18:35:43.014928383 27/05/2026] [src/supervisor/operations.c:144] [NORMAL] GET: MB
    received lin_acc z = -0.758784 m/s^2 from servo ID 2
10
11 [18:35:43.014984977 27/05/2026] [src/supervisor/operations.c:142] [NORMAL] GET: MB
    received lin_acc x = 6.060098 m/s^2 from servo ID 1
12
13 [18:35:43.014992768 27/05/2026] [src/supervisor/operations.c:143] [NORMAL] GET: MB
    received lin_acc y = -7.757189 m/s^2 from servo ID 1
14
15 [18:35:43.014999115 27/05/2026] [src/supervisor/operations.c:144] [NORMAL] GET: MB
    received lin_acc z = -0.516428 m/s^2 from servo ID 1
```

Sending all configuration messages and GET requests in one cycle requires the *supervisor* unit to appropriately assign timestamps to each daughter message and insert them into the downlink buffer as introduced in Chapter 4.2.

The reflex then suspends its operations until the next cycle, to wait for the WRITE and GET frames to be received. Then, the *Hold* reflex sets the angular reference using WRITE and enables the servomotors with the dedicated POS_PID message (cf Chapter 5.2.9):

```
1 [18:35:43.044048179 27/05/2026] [src/supervisor/operations.c:277] [NORMAL] WRITE:
    Servo ID 2 received the angle reference = 3.603900°
2
3 [18:35:43.044183482 27/05/2026] [src/supervisor/operations.c:277] [NORMAL] WRITE:
    Servo ID 1 received the angle reference = 95.723098°
```

Then, at each new cycle, the *Hold* reflex executes the following operations:

- sending GET messages to both servomotors to retrieve their current motor's position and linear acceleration,
- checking for the $touched\_ground$ variable value,
- sending WRITE messages to update each position reference with the position received if the robot has not yet touched the ground.

Once the *expert-collisions* updates $touched\_ground$ to true, *Hold* executes its *full ground contact*[2] phase (cf. Algorithm 6.1) and increases all torque limits to their maximal value:

```
1 [18:35:44.054458432 27/05/2026] [src/supervisor/operations.c:302] [NORMAL] WRITE:
    Servo ID 2 received max torque = 512 mNm
2
3 [18:35:44.054468917 27/05/2026] [src/supervisor/operations.c:302] [NORMAL] WRITE:
    Servo ID 1 received max torque = 512 mNm
```

6.4.4 Lower level implications

Using the USB connection, the MF board receives the downlink buffer containing all the daughter messages to transmit during this cycle (cf. Chapter 3.2).

[2]Since we carried out the experiment on a single leg, the single ground contact phase is skipped.

On the servomotor side, RM-28 receives each message using its *orders* thread, i.e. updating its parameters and responding to the GET requests, then the *main_loop* thread handles the main operations of the servomotors (cf. Chapter 2.7.3). This is a key validation of the Bobby platform that each layer operates correctly in a coordinate way: from high-level MB's reflex logic and PID controllers, through the USB and CAN communication protocol, to the MF board and RM-28 software and electronic.

6.4.5 Results

Figure 6.4 illustrates the controller's response of the ankle servomotor.

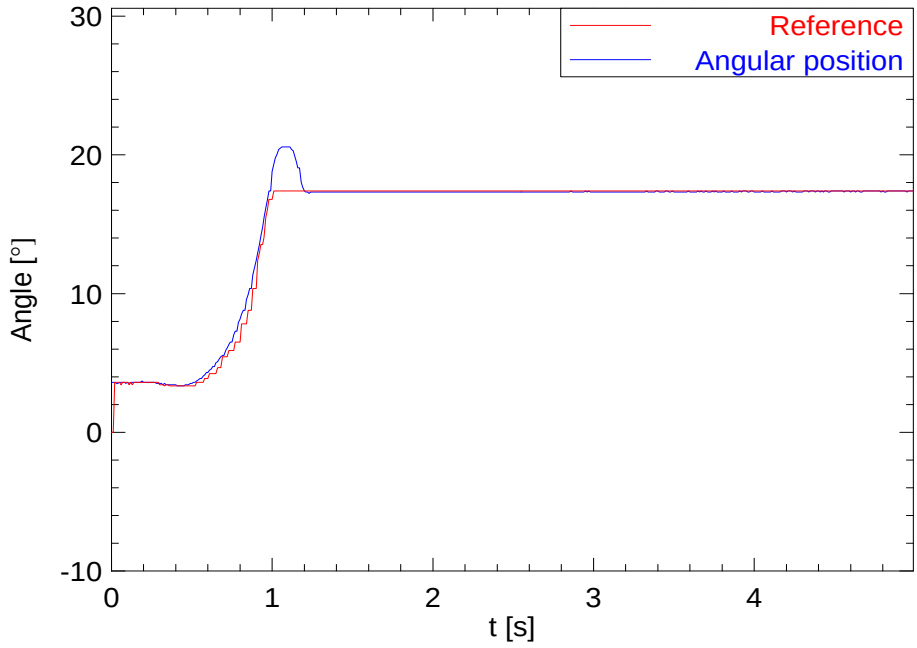


Figure 6.4: Position PI controller of the ankle servomotor during the *Hold* reflex.

Initially, the *Hold* reflex is in its first phase and the motor adapts to external forces with high compliance. Its reference position is updated at each cycle, following its current position.

The force of gravity drives the motor, which increases its angular position until the foot touches the ground. At $t \simeq 1$ s, the reference stops updating, which indicates ground contact. It corresponds to the transition from the *Ground adaptation* to the *Stabilisation* phase.

Due to the acceleration caused by gravity, the controller overshoots before reaching its reference after about 0.15 seconds, showing good reactivity after the collision. Because of the increased stiffness in the *Stabilisation* phase, we observe that the actuator is no longer driven by the force of gravity.

The convergence of the controller's response toward the reference corresponds to the transition state to the next motion phase of the *Standing* mode.

Chapter 7

Conclusions and perspectives

The goal of this work was to build a first complete version of the Bobby platform capable of reproducing human-inspired reflexes. Building on past contributions, this thesis addresses the development of all layers of the system, from the hardware design to the bio-inspired locomotion control, leading to a working platform on which future work can proceed further.

At the hardware level, we completed and fully tested two custom electronic boards:

- the multifunction board, distributing the power throughout the robot and orchestrating the downlink communication with the MB through USB, and with all daughter boards through a periodic CAN cycle,
- the RM-28 servomotor board integrating two CAN transceivers, a motor driver and various sensors such as an IMU, a magnetometer, and a current sense amplifier.

The embedded RM-28 software interfaces the hardware to the higher-level logic. Through a CAN communication channel, the servomotors receive commands to retrieve state information or to update some parameters such as PID references and gains values. We implemented three PID controllers, in position, velocity, and torque, controlling the servomotors and we validated them in a real environment. We designed control limits for safety, performance and anatomical reasons, and an antiwindup mechanism prevents the saturation in the integral term. Each controller tracks the reference with good performance: short rise time, small overshoot and acceptable settling time.

The multifunction board is the intermediary between the MB's commands and the daughter boards. We avoid collisions on the busses by clever scheduling using timestamps associated with each message. Each CAN frame follows the custom application message protocol defined for the platform.

At the high-level, the MB's threads are organised into software units depending on their responsibilities: *world* for state management, *supervisor* for communication orchestration, *experts* for data processing, and *reflex* for locomotion control. We developed fusion methods to resolve the problem when multiple references simultaneously target the same controller or when different controllers produce corrective actions together.

Using all the developed layers, we developed a bio-inspired locomotion system relying on control units similar to human neurons which are stimulated and coordinated through finite-state machines to produce human-like motion. The reproduction of the *Ground adaptation* and *Stabilisation* motion phases of the *Stable standing* mode validates the developed platform since the custom hardware boards, CAN and USB channels, embedded software, PID controllers, and reflex logic operate all correctly together in a real physical environment.

Before this thesis, multiple contributors produced valuable but independent work on Bobby. The RoboCup project at Montefiore can be visualised as multiple parallel paths of past contributions converging to this thesis. Integrating these contributions into a single and operational platform serves as a checkpoint from which future contributors can work toward the project's long-term goal of playing soccer matches.

Software contributions

A dedicated GitHub repository (`https://github.com/Kyhura/tfe`) contains the files to which I actively contributed. Therefore, it does not represent the full project repository of the Montefiore Team and cannot operate on its own.

The repository is divided as follows:

```
repository/
|--- common/
|--- stm32/
|   |--- servo/
|       |--- src/
|           |--- controllers/
|           |--- sensors/
|           |--- utils/
|           |--- main.c
|--- x86-64/
|   |--- mb/
|       |--- src/
|           |--- expert/
|           |--- reflex/
|           |--- supervisor/
|           |--- utils/
|           |--- world/
|           |--- main.c
```

where

- `stm32/` contains the embedded software running on the STM32 microcontrollers of the MF board[1] and servomotors,
- `x86-64/` contains the high-level software of the motherboard, and
- `common/` contains shared header files defining the application message protocol, the information IDs, and constants used across the robot.

We counted source code lines, excluding blank lines and comments, using the *cloc* tool [55]:

- `stm32/` includes 14 headers and 13 source files for a total of 1251 lines,
- `x86-64/` includes 7 headers and 6 source files for a total of 1028 lines, and

[1] Note that the code of the MF board is not present in the repository as my contribution to this component was focus on bug identification and suggesting improvements rather than developing new functionalities.

- `common/` includes 3 headers files for a total of 86 lines.

Therefore, this thesis led to the development of 43 files and approximately 2400 lines of code, all in the C language.

Perspectives

The contributions of this thesis open multiple perspectives for future work.

First of all, the current platform could be strengthened. We voluntary simplified some solutions to focus this thesis on its primary goal. For instance, the MB and DBs follow a master and slave strategy, and we could instead delegate more local responsibilities to the DBs. Similarly, the integrated CAN filters are currently underused and could be enhanced to implement flexible multicast. This two improvements of the platform would reduce the number of messages on the communication channels. Therefore, more bandwidth would be available to incorporate more daughter boards or exchange more higher-priority messages.

In addition, future work could investigate the possibility of direct communication among daughter boards to reduce latency for the transmission of critical messages. However, ensuring collision-free communication would be significantly more challenging, since the MB no longer guarantee the centralised CAN coordination and the scheduling would instead be distributed across multiple nodes.

On the MB side, we could enhance the management of the downlink buffer to maximise the number of daughter messages per cycle. For example, by grouping all frames that are intended to the same DB, we could minimise the delay between consecutive messages. To avoid collisions, the MF would transmit the number of commands that the DB should receive before transmitting its response. Moreover, the MB would perform additional processing of the downlink buffer to group the daughter messages. This would represent an extension of the current responsibilities of the *supervisor* units.

The PID controllers implemented in this work are elementary, and further research could investigate whether filtering velocity and torque measurements would significantly improve the controllers' performance. Furthermore, the integration of direct velocity and torque sensors would represent a significant improvement to the RM-28 design. Such sensors would likely require a redesign of the PCB layout, which could result in the need of a new mechanical enclosure, as the RM-28 is currently restricted to the MX-28 housing.

Now that the Bobby platform demonstrates the capabilities to achieve human-like locomotion, future contributors could implement the rest of the motion phases of the *Stable Standing* mode and other locomotion modes like *Walking* and *Running* as presented by Tobias Luksch [6]. Moreover, determining the orientation of the robot is a critical but tough challenge. Additional work could tackle this issue and implement an efficient proprioceptive system using all IMUs of the robot following the method proposed by Xavier et al. [39].

Bobby's mechanical improvements could include the addition of intrinsic compliance to its structure, similar to humans, whose bodies are not fully rigid. First, through software simulation of elastic return forces, and later by adding electrical series elastic actuators to achieve soft motion control as presented by Vonwirth et al. [56].

About the intelligence of the robot, we have not incorporated any AI into the robot at the moment, as stated in the introduction of this thesis. However, since the platform is currently operational and provides a set of functions interfacing the lower layers, we could train a machine learning model to tune DBs parameters or to solve path planning tasks for instance.

Appendices

A RM-28

A.1 Electronic schematics

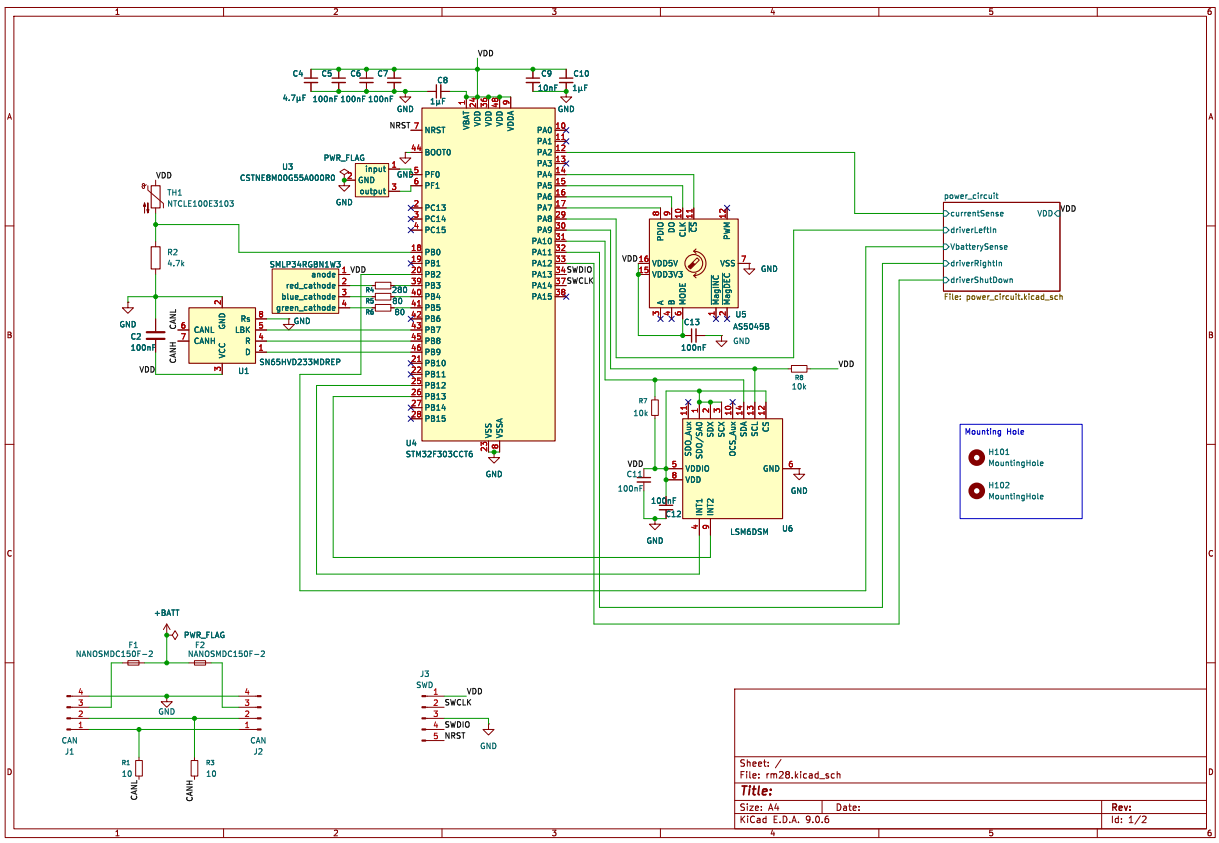


Figure A.1: RM-28 schematics: Microcontroller and Sensors.

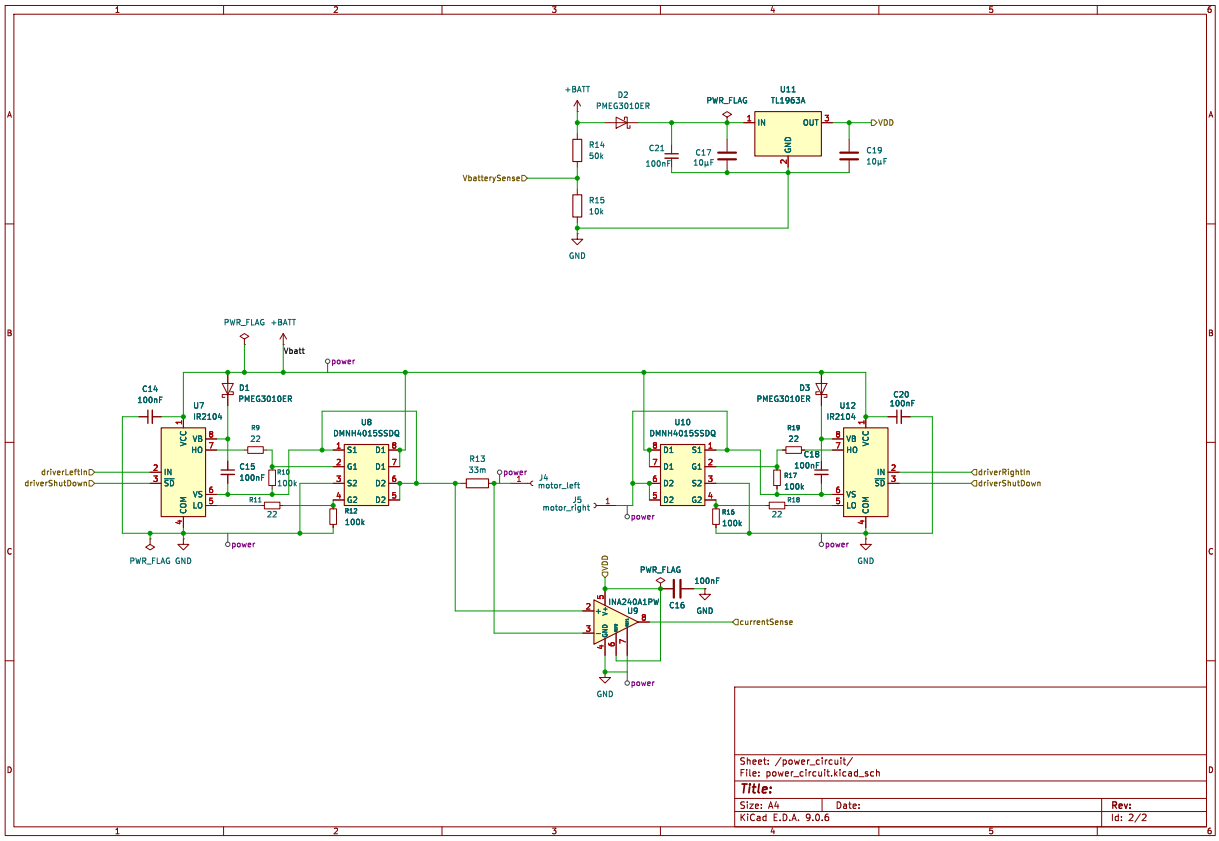
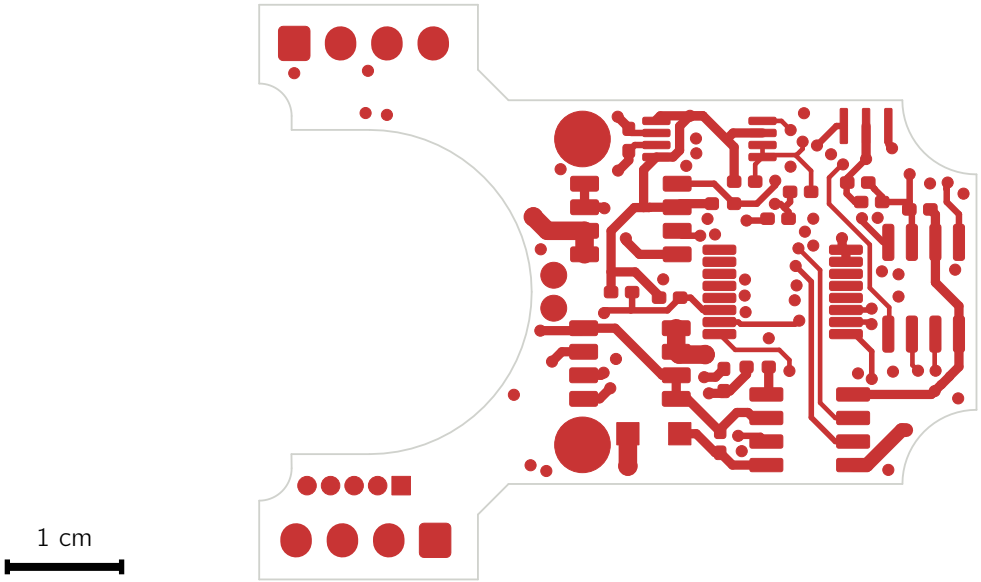
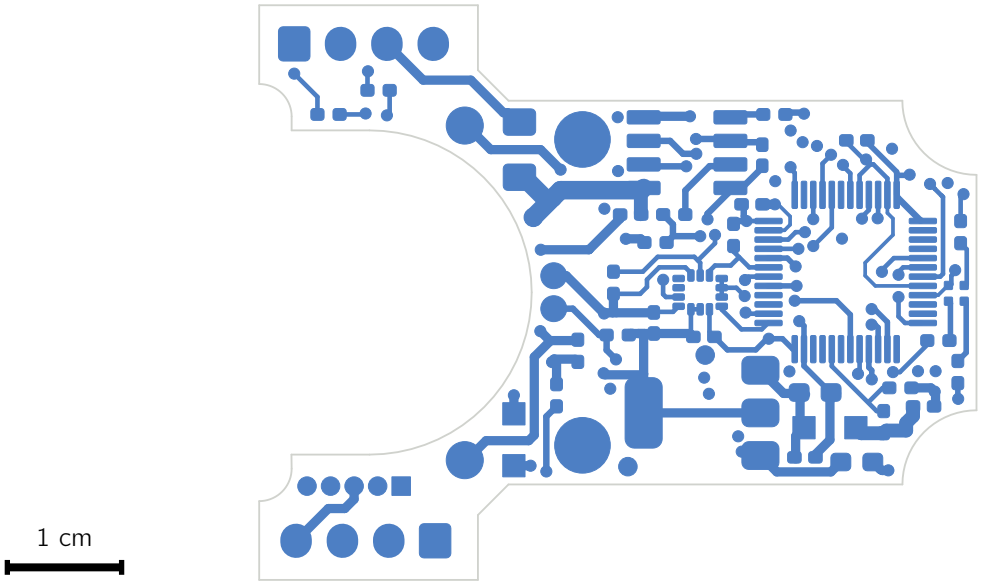


Figure A.2: RM-28 schematics: Power circuit.

A.2 PCB layers

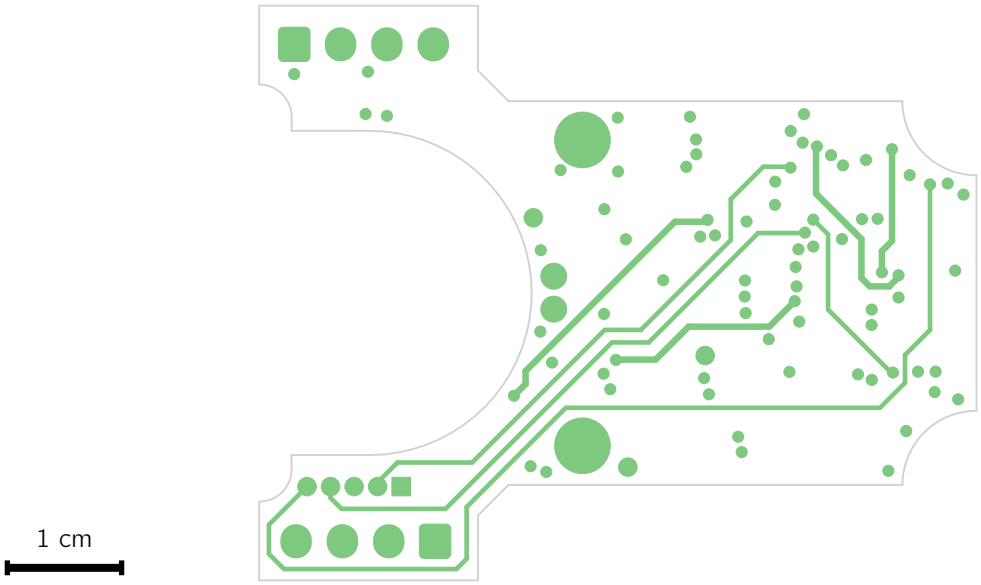


(a) Top layer.

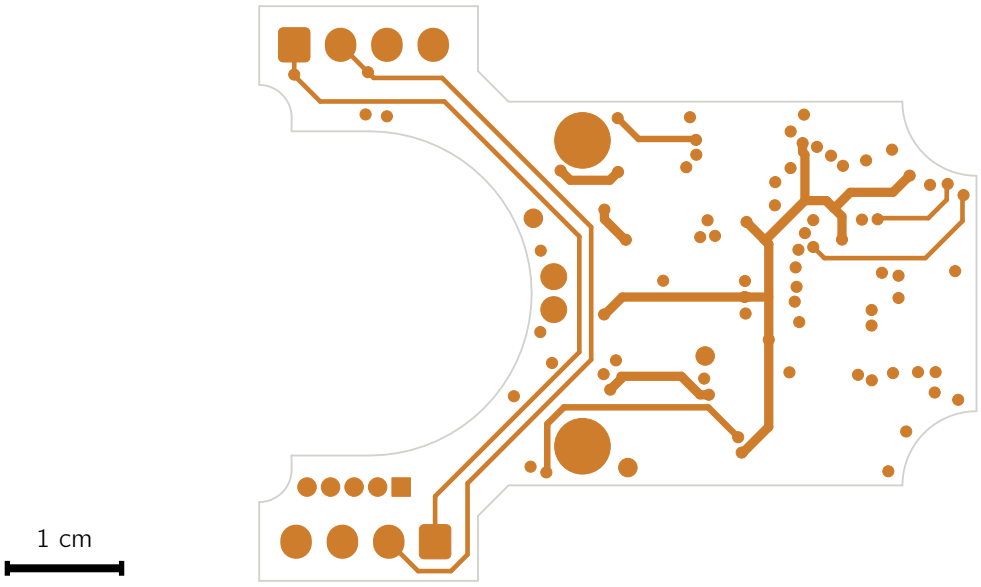


(b) Bottom layer.

Figure A.3: RM-28 PCB layers: Top and Bottom layers.

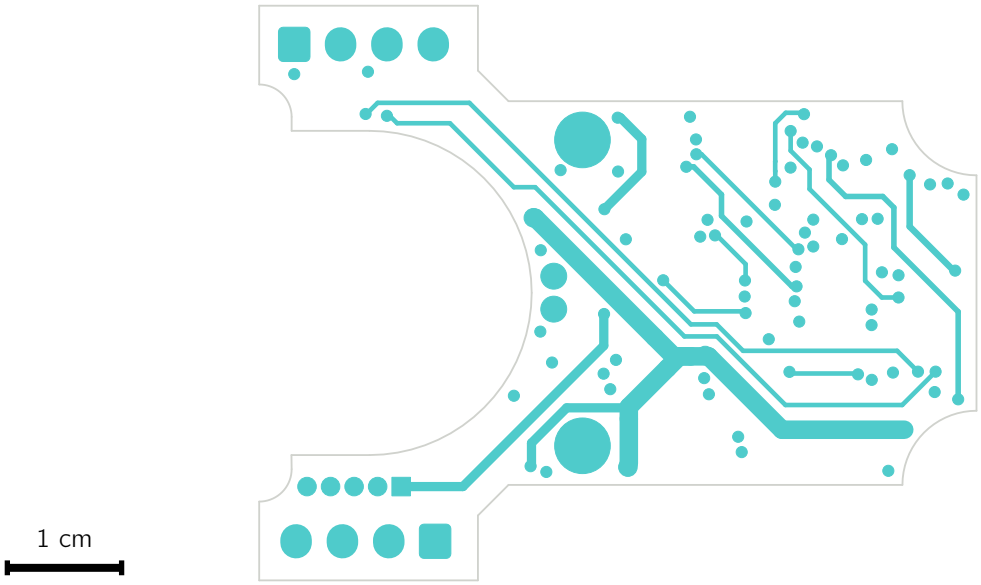


(a) Inner layer 1.

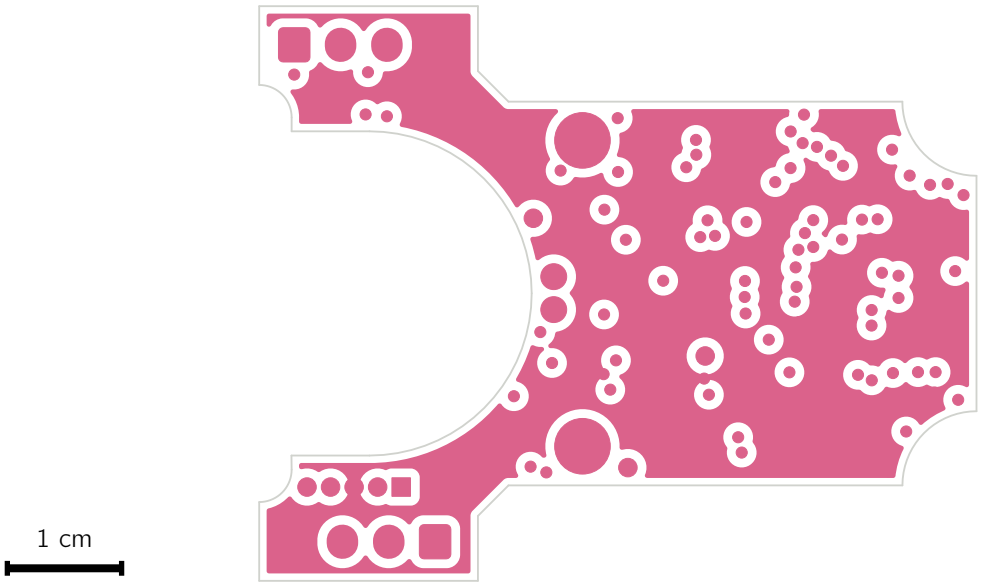


(b) Inner layer 2.

Figure A.4: RM-28 PCB layers: Inner layers 1 and 2.



(a) Inner layer 3.

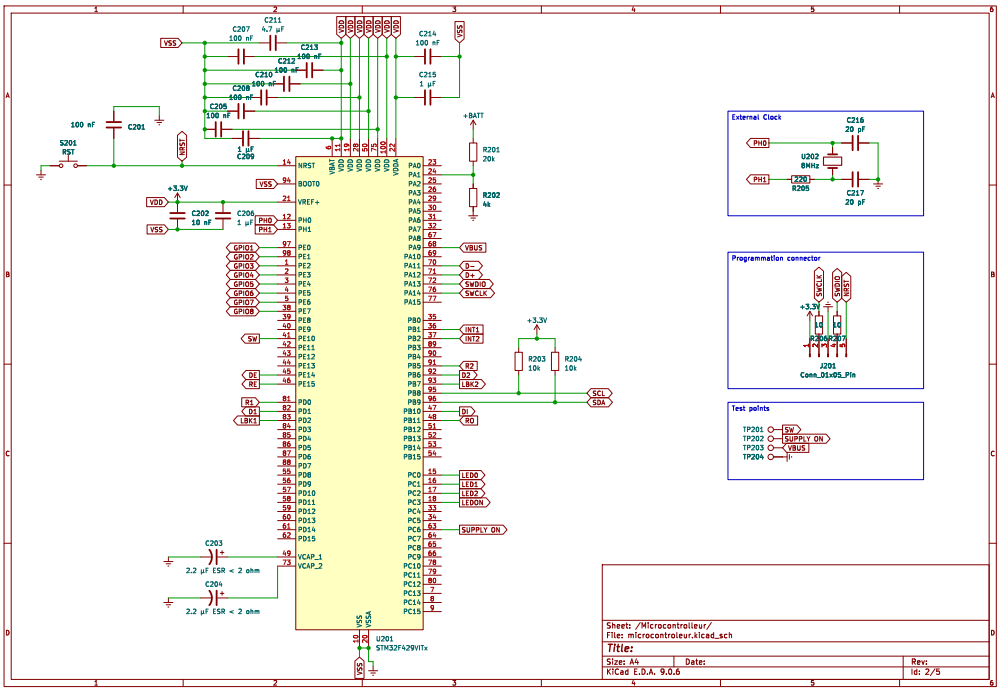


(b) Ground plane.

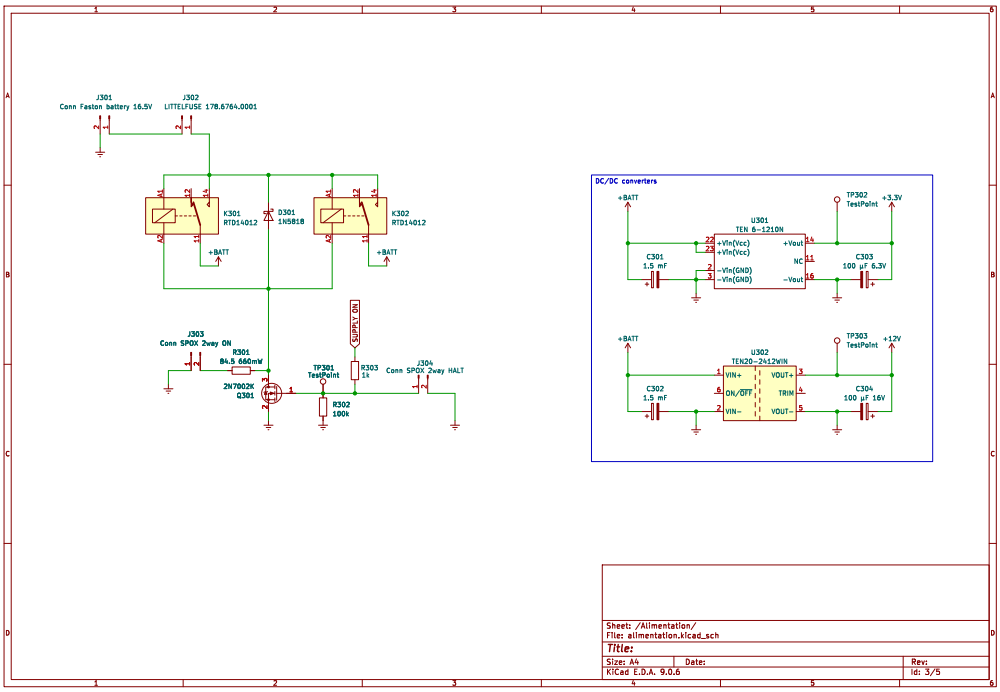
Figure A.5: RM-28 PCB layers: Inner layer 3 and Ground plane.

B Multifunction board

B.1 Electronic schematics

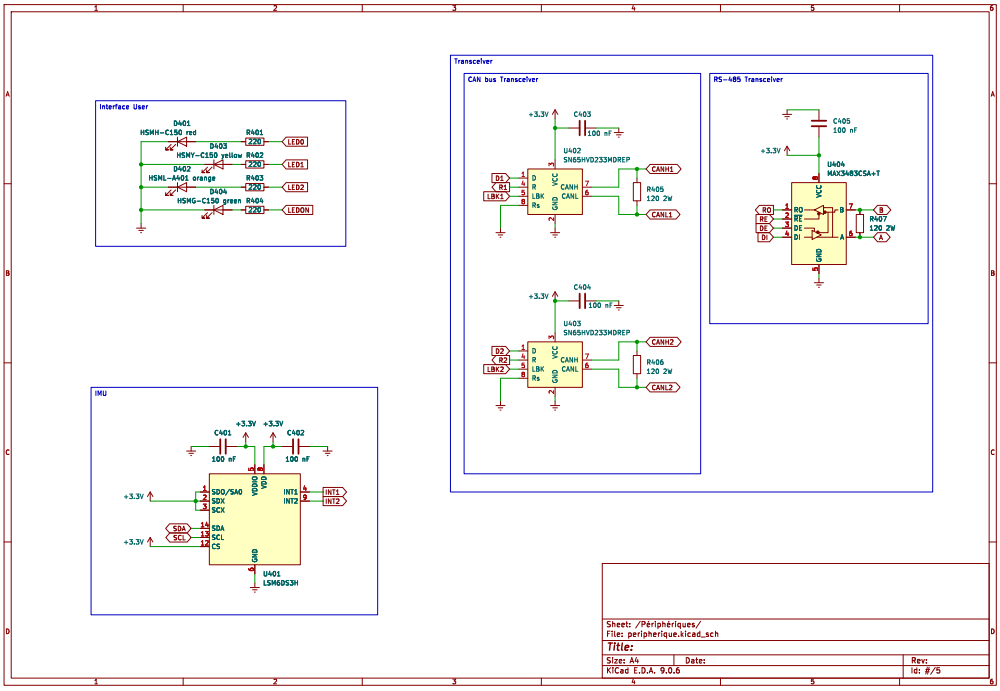


(a) Microcontroller.

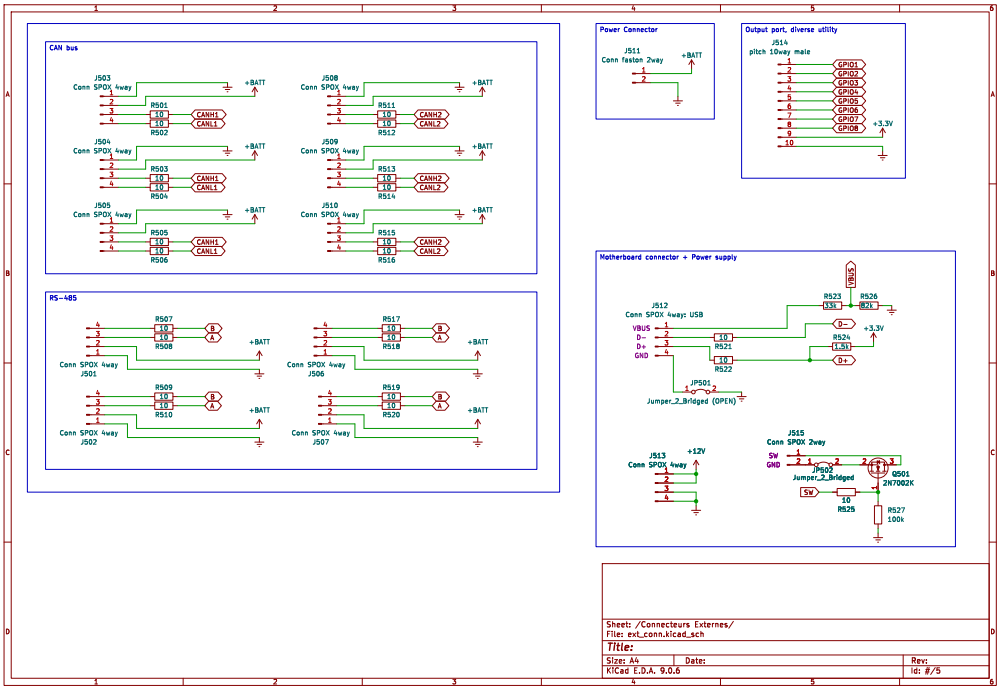


(b) Power supply.

Figure B.1: MF board schematics: Microcontroller and Power Supply.



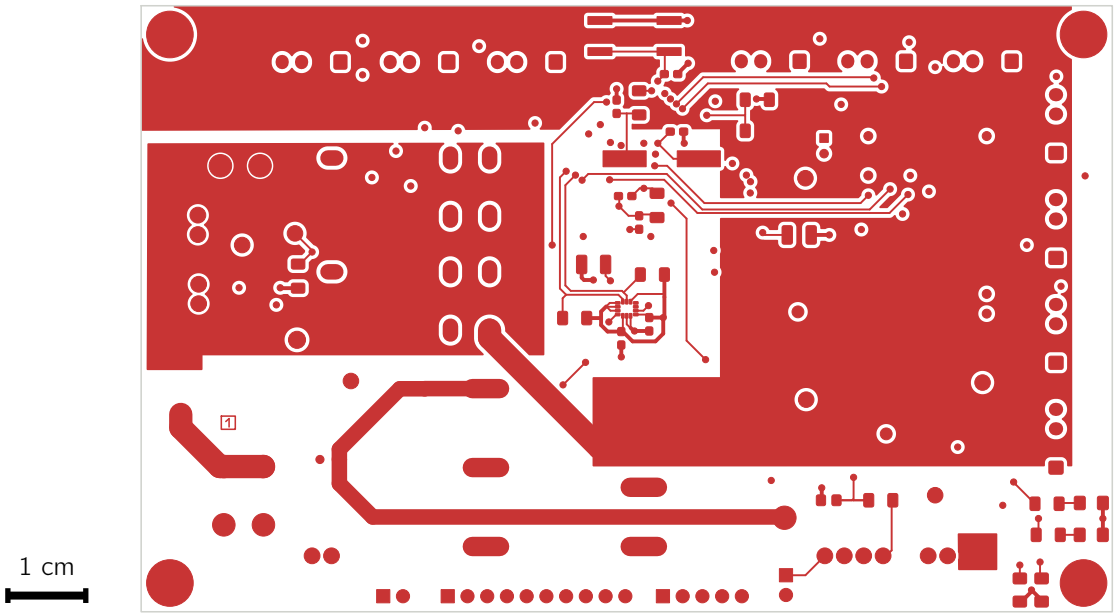
(a) Peripherals.



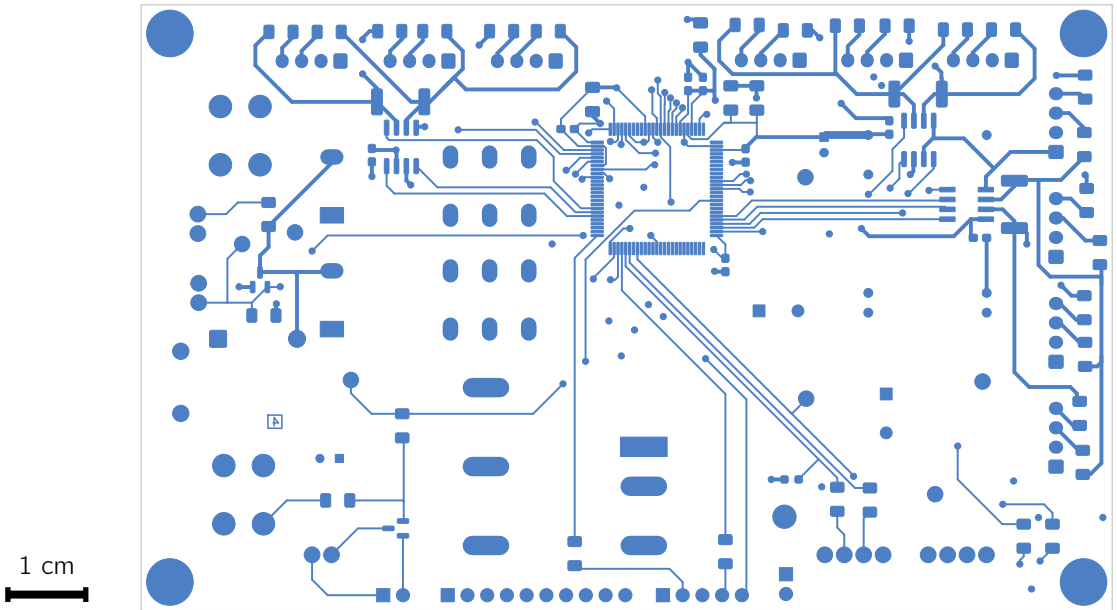
(b) External Connectors.

Figure B.2: MF board schematics: Peripherals and External Connectors.

B.2 PCB layers

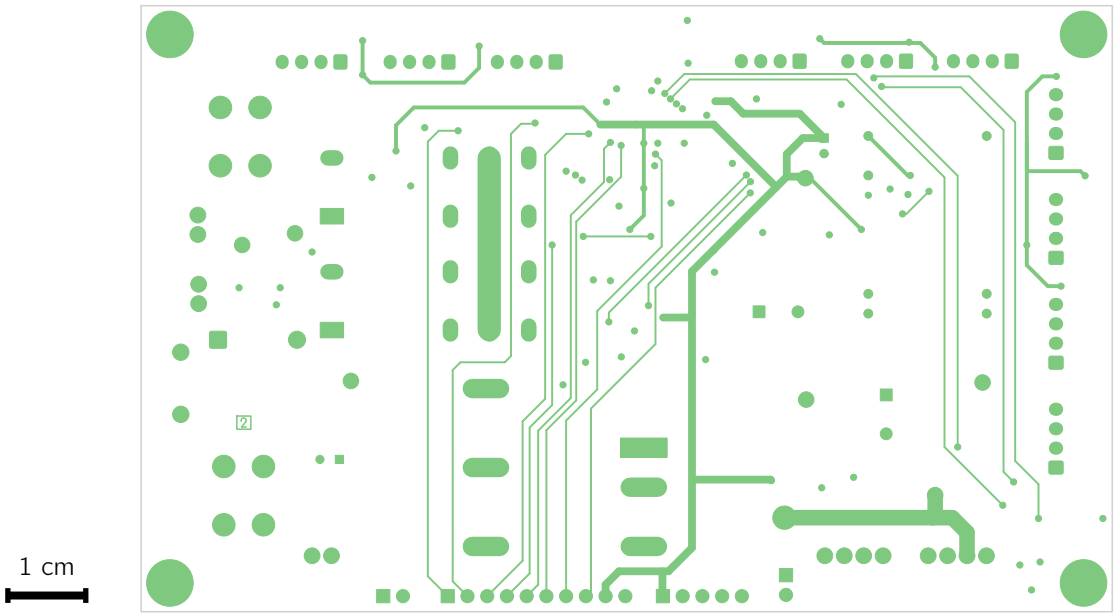


(a) Top layer.

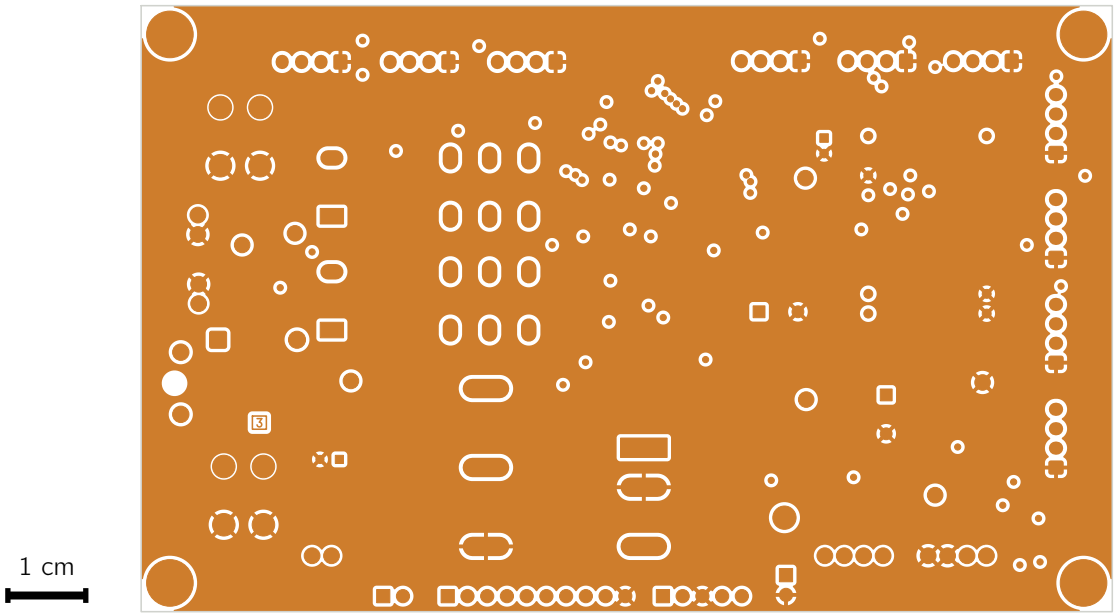


(b) Bottom layer.

Figure B.3: MF board PCB layers: Top and Bottom layers.



(a) Inner layer.



(b) Ground plane.

Figure B.4: MF board PCB layers: Inner layer and Ground plane.

C Application messages

GET

- MT = 0x07
- *Information ID* (1 byte)

Returns:

- MT = 0x07
- *Information ID*: 1 byte
- Value of information requested : the size depends on the requested information.

This message requests for particular data (e.g., current acceleration). We detail the concept of *Information ID* in Appendix D.

Note that the *Information ID* is returned so that the sender can verify that the response corresponds to the expected message. Imagine a situation where two *GET* messages are sent but one is lost and thus the response to the second one could be misinterpreted as the response to the first. The *Information ID* in the response therefore allows the sender to detect such mismatches, even though in normal operation, no messages are expected to be lost.

PID_POS

- MT = 0x08
- *PID_state*: Boolean (1 byte)

PID_SPE

- MT = 0x09
- *PID_state*: Boolean (1 byte)

PID_TOR

- MT = 0x0a
- *PID_state*: Boolean (1 byte)

These messages enable or disable a PID controller as introduced in Chapter 5. Its boolean argument, *PID_state*, enables the controller when set to true and disables it otherwise. This argument returned is used to avoid putting the system into an inconsistent state: if the MB sends a *PID_POS* message that is lost in transmission and does not expect any response, the MB would consider the PID controller as enabled, while the servomotor's controller remains disabled. Similarly to the *Information ID* returned, *PID_state* in the response allows the MB to verify that the controller state was correctly applied.

WRITE0

- MT = 0x0b
- *Information ID* (1 byte)
- Value of the data sent: size depends on the information transmitted (cf. section D).

cf. WRITE1

WRITE1

- MT = 0x0c
- *Information ID* (1 byte)
- Value of the data sent: size depends on the information transmitted (cf. section D).

Returns:

- MT = 0x0c
- *Information ID* (1 byte)
- Value of the data sent: size depends on the information transmitted (cf. section D).

WRITE is used to update the data stored in the DB, for instance, to change the reference of a PID controller. We use alternating *WRITE0* and *WRITE1* similarly to *DATA0* and *DATA1* mechanism used in bootloader messages [1]. This is a protection ensuring that all messages are correctly received by the DB. If two consecutive *WRITE0* messages are received, the DB knows that a message was lost.

The DB return the receive data back to the sender, confirming that the message was correctly received. The MB then updates its knowledge of the *World* accordingly (cf. Chapter 4.3.1).

D Information ID

We decided to associate with each information a 1 byte identifier providing 256 distinct data types which is sufficient for the current platform but could be increased in the future if needed. For each entry, we detail the data associated type, its size in bytes, its units and its accessibility. The parameters that cannot be modified externally (cf. Chapter 4.3.5) are marked as read-only (R) and the others as read-write (RW).

Information ID	Associated data	Size in byte	Unit	Access
0x00	ID	1	/	R
0x01	Serial number	1	/	R
0x02	Motor's position (cf. Chapter 2.5.1)	2 (0-4095)	0.0879[°]	R
0x03	Angular velocity (cf. D.1)	6 (3×2)	0.0175 [dps]	R
0x04	Linear acceleration (cf. D.2)	6 (3×2)	5.9841×10^{-4} [m/s^2]	R
0x05	Torque (cf. D.3)	2	0.1 [%]	R
0x06	Motor Velocity	4	cf. D.4	R
0x07	Voltage (cf. D.5)	1 (0-255)	0.1 [V]	R
0x08	Temperature of motor	1 (0-150)	1[°C]	R
0x09	Temperature of MCU	1 (0-150)	1[°C]	R
0x0a	LEDs	1 (0-7)	cf. D.6	RW
0x0b	PWM command	2	cf. D.7	RW
0x0c	CW Position Limit (cf. Chapter 5.2)	2	0.0879[°]	RW
0x0d	CCW Position Limit (cf. Chapter 5.2)	2	0.0879[°]	RW
0x0e	Max motor's velocity	4	cf. D.4	RW
0x0f	Max Torque (cf. D.3)	2	0.1 [%]	RW
0x10	Motor temperature Limit	1 (0-150)	1[°C]	RW
0x11	MCU temperature Limit	1 (0-150)	1[°C]	RW
0x12	Min Voltage Limit	1 (0-255)	0.1 [V]	RW
0x13	Max Voltage Limit	1 (0-255)	0.1 [V]	RW
0x14	Position reference	2 (0-4095)	0.0879[°]	RW
0x15	Velocity reference	4	cf. D.4	RW
0x16	Torque reference (cf. D.3)	2	0.1 [%]	RW
0x17	Position proportional gain kp	4	cf. D.8	RW
0x18	Position integral gain ki	4	cf. D.8	RW
0x19	Position derivative gain kd	4	cf. D.8	RW
0x1a	Velocity proportional gain kp	4	cf. D.8	RW
0x1b	Velocity integral gain ki	4	cf. D.8	RW
0x1c	Velocity derivative gain kd	4	cf. D.8	RW
0x1d	Torque proportional gain kp	4	cf. D.8	RW
0x1e	Torque integral gain ki	4	cf. D.8	RW
0x1f	Torque derivative gain kd	4	cf. D.8	RW
0x20	PID position output generated	2	cf. D.7	R
0x21	PID velocity output generated	2	cf. D.7	R
0x22	PID torque output generated	2	cf. D.7	R
0x23	Position integral limit	1	1	RW
0x24	Velocity integral limit	1	1	RW
0x25	Torque integral limit	1	1	RW
0x26	Position modulation factor (cf. 5.2.7)	2 (0-1000)	0.1 [%]	RW
0x27	Velocity modulation factor (cf. 5.2.7)	2 (0-1000)	0.1 [%]	RW
0x28	Torque modulation factor (cf. 5.2.7)	2 (0-1000)	0.1 [%]	RW
0x29	Position tolerance	2	cf. D.9	RW

0x2a	Velocity tolerance	2	cf. D.9	RW
0x2b	Torque tolerance	2	cf. D.9	RW
0x2c	Flags	1	cf. D.10	R

Table D.1: Table describing the meaning size, unit and accessibility of each Information ID

D.1 Angular velocity

As presented in Chapter 2.5.3, the gyroscope has a full-scale range of 500 dps and a sensitivity of 17.5 mdps/LSB. Each axis is encoded as a 16-bit signed integer and all three axes are returned, resulting in $3 \times 2 = 6$ bytes in total.

D.2 Linear Acceleration

As presented in Chapter 2.5.3, the accelerometer has a full-scale range of $2g$ and a sensitivity of 0.061 mg/LSB. We decided to express the acceleration in m/s^2, thus each bit represents $0.061 \times 9.81 \times 10^{-3} = 5.9841 \times 10^{-4}$. Each axis is encoded as a 16-bit signed integer and all three axes are returned, resulting in $3 \times 2 = 6$ bytes in total.

D.3 Torque

Similarly to the torque limit introduced in Chapter 5.2.5, we express the torque as a percentage of its maximal value. Each LSB corresponds to 0.1 % so we need two bytes to encode this information.

D.4 Motor's velocity

The motor's velocity (cf. Chapter 2.5.2) is expressed by a 16.16-fixed-point number stored in four bytes where 16 bits represent the integer part and 16 bits the decimal part. The transmitted raw value is divided by 2^{16} to recover the actual velocity.

D.5 Voltage

This value represents the power's supply voltage presented in Chapter 2.5.5. The byte value ranges from 0 to 255, where each LSB corresponds to 0.1 V. We can thus measure input voltage from 0 V to 25.5 V which is sufficient as the power supply should not exceed 20 V in normal operating conditions.

D.6 LEDs

The LEDs are used to report event as presented in Chapter 2.4.1.

Each value represents a different case:

1. Off
2. Red
3. Green

4. Blue
5. Yellow (Red-Green)
6. Magenta (Red-Blue)
7. Cyan (Green-Blue)
8. White (Red-Green-Blue)

D.7 PWM command

This value is a signed integer whose range goes from 0 to 512. It corresponds to the PWM command used in the abstract control space introduced in Chapter 2.6.2.

Note that the PWM command information ID (0x0b) is only used for debugging reasons to externally control the motor.

D.8 Gains

All the PID gains (cf. Chapter 5.2.1) are expressed by a 16.16-fixed-point number stored in four bytes. The transmitted raw value is divided by 2^{16} to recover the actual gain.

Note that for the moment, the use of four bytes is generous but it is not a major issue. First, as gains are never modified every cycle by design, thus using too much bytes does not significantly impact the communication bandwidth. Second, it helps determining the range of values needed in practice. The number of bytes can be reduced in the future to better fit the required range.

D.9 Tolerances

This value represents the limits tolerance introduced in Chapter 5.2.2 which delimits the critical areas range.

We represents this value by a 8.8-fixed-point number. The transmitted raw value is divided by 2^8 to recover the actual tolerance.

D.10 Flags

This byte represents all the flag bits that the servomotor can raise (cf. Chapter 2.4.2).

The bit description is as follows:

1. Error in magnetic sensor detected
2. PID position windup detected
3. PID velocity windup detected
4. PID torque windup detected
5. Maximum torque reached

6. Actuator out of range
7. Maximum temperature reached
8. *Unused*

E Pseudocode and custom structure

E.1 RM-28's sensor interface software

Magnetometer sensor

```
    // Block to prevent simultaneous multiple read
 1: Acquire lock
 2: Set CS_pin to low // Initiate communication
 3: i ← 0
 4: v ← 0

    // Read 18 bits
 5: for i = 0 to 17 do
 6:     Pulse CLK_pin // Generate one clock cycle
 7:     Shift v left by 1
 8:     Set LSB of v to current bit value
 9: end for
10: Set CS_pin to high // End communication
11: Release lock

    // End of critical section
12: status ← v[5 : 0]
13: data ← v[17 : 6]

    // Status checks
14: if status[5] = 0 then
15:     Set magnetometer flag // Offset compensation not finished
16: end if
17: if status[4] = 1 then
18:     Set magnetometer flag // CORDIC overflow
19: end if
20: if status[3] = 1 then
21:     Set magnetometer flag // Linearity alarm
22: end if

    // Parity check
23: if parity(v[17 : 1]) ≠ status[0] then
24:     Set magnetometer flag // Transmission error
25: end if
26: return data
```

Algorithm E.1: Read of magnetometer sensor.

Angular differences

Require: α_1, α_2

```
1: α_diff ← α_1 − α_2
2: // Compute coterminal angle
3: α_diff,cot ← angle_coterminal(α_diff)
4: if |α_diff| > |α_diff,cot| then
```

5: $difference_{big} \leftarrow \alpha_{diff}$
6: $difference_{small} \leftarrow \alpha_{diff,cot}$
7: **else**
8: $difference_{small} \leftarrow \alpha_{diff}$
9: $difference_{big} \leftarrow \alpha_{diff,cot}$
10: **end if**
11: **return** $difference_{small}, difference_{big}$

Algorithm E.2: Angular differences.

E.2 RM-28 controller limit mechanism

Angular position limit

Require: $pwm\_command$

1: $tol \leftarrow 5$
// Angle reference frame conversion
2: $\alpha_{motor,conv} \leftarrow$ angle_frame_conversion($\alpha motor$)
3: $\theta_{max} \leftarrow$ angle_frame_conversion(α_{CW})

4: **if** $pwm\_command < 0$ **then**
5: $ccw \leftarrow True$
6: **else**
7: $ccw \leftarrow False$
8: **end if**
// If actively tries to go outside the limit
9: **if** (ccw && $\alpha_{motor,conv} \leqslant$ tol) || (!ccw && $\alpha_{motor,conv} \geqslant \theta_{max}$ - tol) **then**
10: $pwm\_command \leftarrow 0$
11: **end if**

Algorithm E.3: Angular position limit implementation.

Torque limit

Require: $pwm\_command$, $torque\_percentage$

1: $max\_torque \leftarrow torque\_percentage \times 512$

2: **if** $|pwm\_command| > max\_torque$ **then**
3: **if** $pwm\_command < 0$ **then**
4: $pwm\_command \leftarrow -max\_torque$
5: **else**
6: $pwm\_command \leftarrow max\_torque$
7: **end if**
8: **end if**

Algorithm E.4: Torque limit implementation.

Antiwindup integral limit

Require: $integral\_term$, $integral\_limit$

if $|integral\_term| > |integral\_limit|$ **then**
  if $integral\_term < 0$ **then**
    $integral\_term \leftarrow -integral\_limit$
  else
    $integral\_term \leftarrow integral\_limit$
  end if
end if

Algorithm E.5: Antiwindup implementation.

E.3 RM-28 PID controller logic

Angular convention

Require: α, α_{CCW}, α_{CW}

$\alpha_{conv} \leftarrow \alpha - \alpha_{CCW}$

if $\alpha_{conv} < 0$ **then**
  // Convert back into [0; 360[range
  $\alpha_{conv} \leftarrow \alpha_{conv} + 360$
end if

return α_{conv}

Algorithm E.6: Conversion to the angular convention.

PID

```
struct pid {
    double kp;
    double ki;
    double kd;
    uint16_t integral_limit;
    double integral_term;
    uint8_t mod_factor;
    double prev_error;
};
```

Code E.1: PID structure

Require: pid, $error$

$cycle\_period \leftarrow 0.01$ // 10 ms
pid.$integral\_term \leftarrow error \times cycle\_period +$ pid.$integral\_term$

/* *Antiwindup* (cf. Algorithm E.5) */

4: $deriv\_term \leftarrow \dfrac{error - \text{pid}.previous\_error}{cycle\_period}$
5: $\text{pid}.previous\_error \leftarrow error$

6: // Corrective action
7: $u \leftarrow \text{pid}.k_p \times error + \text{pid}.k_i \times pid.intergral\_term + \text{pid}.k_d \times deriv\_term$

8: $u\_modulated \leftarrow \dfrac{\text{pid}.mod\_factor \times u}{100}$

9: **return** $u\_modulated$

Algorithm E.7: Corrective action computation.

Require: α_{motor}, α_{ref}

1: // Angle frame conversion
2: $\alpha_{motor,conv} \leftarrow angle\_frame\_conversion(\alpha_{motor})$
3: $\alpha_{ref,conv} \leftarrow angle\_frame\_conversion(\alpha_{ref})$
4: $\theta_{max} \leftarrow angle\_frame\_conversion(\alpha_{CW})$

5: // Out of range check
6: $oor \leftarrow false$
7: **if** $\alpha_{motor,conv} > \theta_{max}$ **then**
8: $oor \leftarrow true$
9: /* *Update reference with the closest angular limit* */
10: **end if**

// Compute error
11: $difference_{small}, difference_{big} \leftarrow$ angle_diff($\alpha_{ref,conv}$, $\alpha_{motor,conv}$) // cf. Algorithm E.2

12: **if** not oor **then**
13: **if** /* *smaller path goes out of range* */ **then**
14: // Take the longest path
15: $error \leftarrow difference_{big}$
16: **end if**
17: **end if**

18: compute_corrective_action(position_pid, $error$) // cf. Algorithm E.7

Algorithm E.8: Position PID controller.

E.4 Motherboard high-level software

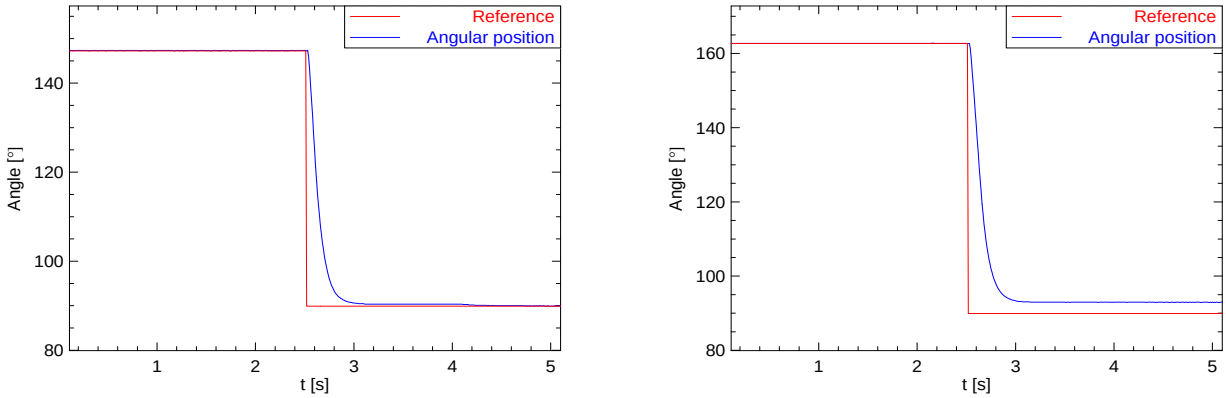
```
struct servo {
    uint8_t polarity;

    double angle;
    double ref_angle;
    double pos_kp;
    double pos_ki;
    double pos_kd;
    uint16_t pos_integral_limit;
    uint8_t pos_mod;

    double speed;
    double ref_speed;
    double spe_kp;
    double spe_ki;
    double spe_kd;
    uint16_t spe_integral_limit;
    uint8_t spe_mod;

    double torque;
    double ref_torque;
    double tor_kp;
    double tor_ki;
    double tor_kd;
    uint16_t tor_integral_limit;
    uint8_t tor_mod;

    uint16_t max_torque;

    double lin_acc_x, lin_acc_y, lin_acc_z;
};
```

Code E.2: Servomotor structure

```
struct Task {
    unsigned int task_id;
    char message_buffer[QUEUE_SIZE];
    pthread_cond_t cond;
    pthread_mutex_t mutex;
    int waiting_message;
};
```

Code E.3: Task structure

F PID controllers results



(a) $K_p = 10$, $K_i = 5$ and $K_d = 0$.

(b) $K_p = 10$, $K_i = 0$ and $K_d = 0$.

Figure F.1: Position PID controller's response to step input with suboptimal tuning

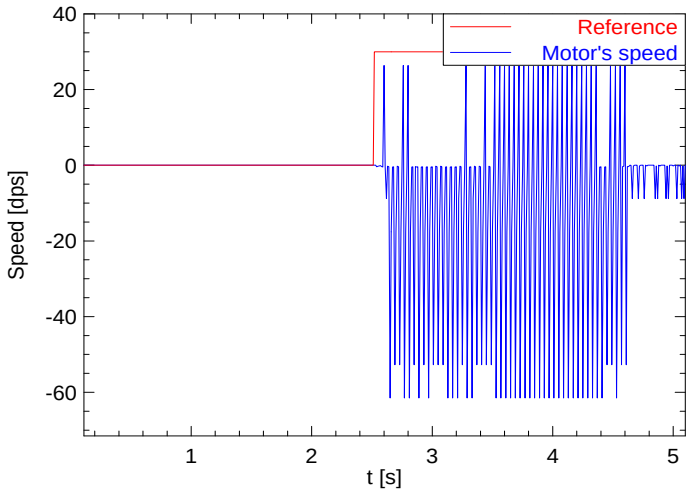


Figure F.2: Velocity controller's response to step input with small derivative term ($K_p = 1$, $K_i = 10$ and $K_d = 1$)

Bibliography

[1] Montefiore Team and B. Boigelot. Structure of embedded hardware v0.1, 2026. Revision 13 March, 2026.

[2] ams OSRAM Group. *AS5045 : Electronic Component Datasheet (Document 4092588)*. Accessed: 2026-05-05.

[3] G. Lempereur. Development of an embedded servomotor controller. Master's thesis, University of Liège, Liège, Belgium, 2016.

[4] Texas Instruments. *INA240: Bidirectional, Zero-Drift, Current-Sense Amplifier with Enhanced PWM Rejection*, 2021. Accessed: 2026-05-05.

[5] STMicroelectronics. *LSM6DS3H: iNEMO Inertial Module – Always-on 3D Accelerometer and 3D Gyroscope*. STMicroelectronics, September 2017. Datasheet – production data.

[6] T. Luksch. *Human-like Control of Dynamically Walking Bipedal Robots*. 01 2010.

[7] RoboCup Federation. Robocup official website. https://www.robocup.org/. Accessed: 2026-05-01.

[8] K. Hirai, M. Hirose, Y. Haikawa, and T. Takenaka. The development of honda humanoid robot. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, volume 2, pages 1321–1326 vol.2, 1998.

[9] MIOMIR VUKOBRATOVIĆ and BRANISLAV BOROVAC. Zero-moment point — thirty five years of its life. volume 01, pages 157–173, 2004.

[10] Tomohito Takubo, Yoshinori Imada, Kenichi Ohara, Yasushi Mae, and Tatsuo Arai. Rough terrain walking for bipedal robot by using zmp criteria map. In *2009 IEEE International Conference on Robotics and Automation*, pages 788–793, 2009.

[11] Shuuji Kajita, Mitsuharu Morisawa, Kanako Miura, Shin'ichiro Nakaoka, Kensuke Harada, Kenji Kaneko, Fumio Kanehiro, and Kazuhito Yokoi. Biped walking stabilization based on linear inverted pendulum tracking. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4489–4496, 2010.

[12] Yonghee Cho and Jong Hyeon Park. Whole-body motion generation for balancing of biped robot. volume 15, 2025.

[13] Tadeusz Mikolajczyk, Emilia Mikołajewska, Hayder F. N. Al-Shuka, Tomasz Malinowski, Adam Kłodowski, Danil Yurievich Pimenov, Tomasz Paczkowski, Fuwen Hu, Khaled Giasin, Dariusz Mikołajewski, and Marek Macko. Recent advances in bipedal walking robots: Review of gait, drive, sensors and control systems. volume 22, pages 4440–, Basel, 2022. MDPI AG.

[14] A. Seireg and R.J. Arvikar. A mathematical model for evaluation of forces in lower extremeties of the musculo-skeletal system. volume 6, pages 313–326, 1973.

[15] Vivek Sangwan and Sunil K. Agrawal. Differentially flat design of bipeds ensuring limit cycles. volume 14, pages 647–657, 2009.

[16] Christine Chevallereau, Guy Bessonnet, Gabriel Abba, and Yannick Aoustin. *Bipedal robots: Modeling, design and walking synthesis*. John Wiley & Sons, 2013.

[17] John G. Milton. Introduction to focus issue: Bipedal locomotion—from robots to humans. volume 19, 06 2009.

[18] Mehdi Heydari Shahna, Seyed Adel Alizadeh Kolagar, and Jouni Mattila. Integrating deeprl with robust low-level control in robotic manipulators for non-repetitive reaching tasks. In *2024 IEEE International Conference on Mechatronics and Automation (ICMA)*, page 329–336. IEEE, August 2024.

[19] Y. Li, Z. Chen, C. Wu, H. Mao, and P. Sun. A hierarchical framework for quadruped robots gait planning based on DDPG. volume 8, page 382, Basel, Switzerland, 2023. MDPI.

[20] Thanh Thi Nguyen, Saeid Nahavandi, Imran Razzak, Dung Nguyen, Nhat Truong Pham, and Quoc Viet Hung Nguyen. The emergence of deep reinforcement learning for path planning, 2025.

[21] A123 Systems. Anr26650m1-b nanophosphate lithium ion cell datasheet. Technical report, A123 Systems, 2012. Accessed: 2026-05-01.

[22] G. Di Carlo. Vision-based robot position estimation. Master's thesis, University of Liège, Liège, Belgium, 2016.

[23] H. Woszczyk. Simulation of complex actuators. Master's thesis, University of Liège, Liège, Belgium, 2016.

[24] ROBOTIS. Dynamixel mx-28 e-manual. https://emanual.robotis.com/docs/en/dxl/mx/mx-28/, 2026. Accessed: 2026-05-01.

[25] L. Infantino. Current measurement for a servomotor, 2026.

[26] T. Ewbank. Efficient and precise stereoscopic vision for humanoid robots. Master's thesis, University of Liège, Liège, Belgium, 2017.

[27] O. Wauquaire. Locomotion control system of a humanoid robot - a biologically inspired model. Master's thesis, University of Liège, Liège, Belgium, 2017.

[28] Q Boileau. Building a simulation environment for biped walking robots using blender, 2017.

[29] P Nicolay. Integration of libopencm3 in freertos for stm32f4 and stm32f3 mcu, 2019.

[30] libopencm3 Project. libopencm3 wiki. https://github.com/libopencm3/libopencm3/wiki, 2026. Accessed: 2026-05-10.

[31] N. Bounar. Development of the software architecture of a mobile robot. Master's thesis, University of Liège, Liège, Belgium, 2024.

[32] M. Léonard. Design of an internal communication protocol over can bus for humanoid robots, 2024.

[33] A. Roekens. Development of a multifunction board for a humanoid robot, 2025.

[34] S. Gardier. Development of a bootloader for the stm32f3 plateform, 2025.

[35] L. Degueldre. Development of a battery management system for a mobile robot. Master's thesis, University of Liège, Liège, Belgium, 2026.

[36] Rico Tilgner, Stefan Seering, Tobias Kalbitz, Thomas Reinhardt, Michael Wünsch, Florian Mewes, Tobias Jagla, Marvin Jenkel, Felix Loos, Benedikt Wismeth, Lea Kunz, Eric Behrendt, Max Polter, Johann Straube, Lennart Konstantin Peters, Freijdis Jurkat, Leah Herrfurth, David Schulte, Johannes Walter, Jurek Max Engelmann, Pascal Setzer, Yushi Wang, Penghui Chen, and Hao Dong. Boosted htwk - extended abstract. RoboCup Humanoid League Team Description Paper, 2025. Accessed: 2026-05-02.

[37] Booster Robotics Technology Co., Ltd. Booster k1 humanoid robot. https://www.aparobot.com/robots/booster-k1, 2025. Accessed: 2026-05-02.

[38] ZJUDancer Team, Zhejiang University. Zjudancer robot specifications. Robocup humanoid league specification document, Zhejiang University, 2026. Accessed: 2026-05-02.

[39] Fabio Elnecave Xavier, Guillaume Burger, Marine Pétriaux, Jean-Emmanuel Deschaud, and François Goulette. Multi-imu proprioceptive state estimator for humanoid robots. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10880–10887, 2023.

[40] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. volume 7. American Association for the Advancement of Science (AAAS), May 2022.

[41] LattePanda. Lattepanda 3 delta specification. https://docs.lattepanda.com/content/3rd_delta_edition/specification/. Accessed: 2026-05-05.

[42] Sebastian Andrzej Siewior. Theory of operation - preempt_rt. https://docs.kernel.org/core-api/real-time/theory.html, 2026. Linux Kernel Documentation, accessed 2026-05-28.

[43] STMicroelectronics. Stm32f303xb/c/d/e advanced arm-based 32-bit mcus datasheet. Technical report, STMicroelectronics, 2016. Accessed: 2026-05-01.

[44] ROHM Semiconductors. *NTCLESMLP34RGBN1W3 100E3*. Accessed: 2026-05-18.

[45] Maxon. *RE-max 17, diam 17 mm, Precious Metal Brushes CLL, 4 Watt*, 2014. Accessed: 2026-05-22.

[46] STMicroelectronics. *RM0090 Reference Manual: STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 Advanced Arm-based 32-bit MCUs*. STMicroelectronics, 2024. Revision 21 June, 2024.

[47] FreeRTOS. Freertos kernel book. https://github.com/FreeRTOS/FreeRTOS-Kernel-Book, 2026. Accessed: 2026-05-17.

[48] World Athletics. *C1.1 C2.1 - Competition Rules Technical Rules*. World Athletics, 2024. Accessed: 2026-04-30.

[49] Steve Corrigan. Introduction to the controller area network (can). Application Report SLOA101B, Texas Instruments, May 2016. Revised from original publication in August 2002.

[50] Bureau International des Poids et Mesures. *The International System of Units (SI)*. BIPM, 9 edition, 2019.

[51] NXP Semiconductors. *I2C-bu-s Specification and User Manual*. NXP Semiconductors, 7.0 edition, October 2021. Rev. 7.0 — 1 October 2021.

[52] Montefiore Team and B. Boigelot. Frida athos : Notes sur l'informatique embarquée, 2009. Revision 15 October, 2009.

[53] Karl J. Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2 edition, 2008.

[54] V. Romero Segovia, T. Hägglund, and K.J. Åström. Measurement noise filtering for pid controllers. volume 24, pages 299–313, 2014.

[55] AlDanial. cloc. https://github.com/AlDanial/cloc, 2026. Accessed: 2026-06-08.

[56] Patrick Vonwirth and Karsten Berns. Muscular damping distribution strategy for bio-inspired, soft motion control at variable precision. volume 23, page 2428. MDPI, 2023.