Ecole Polytechnique
Fédérale de Lausanne

Firmware development for a sensorized Pleurobot

Semester Project

Author

GRILLO Andrea

371099

Professor

IJSPEERT Auke Jan

Supervisor

FU Qiyuan

Fall 2024

BIOROB - EPFL

Contents

1 Introduction

The Pleurobot is a bio-inspired robot designed to mimic the movement and behavior of the salamander, specifically the Pleurodeles waltl. Developed by the Biorobotics Laboratory (BIOROB) at EPFL, the Pleurobot aims to provide insights into the control of locomotion and to advance the field of robotics by leveraging biological principles.
The robot features a sophisticated design that includes 27 degrees of freedom, each actuated by a Dynamixel servo motors. This allows for the study of control algorithms aiming to replicate the complex movements of its biological counterpart.
This project focuses on the development of the software stack for the third version of the Pleurobot.
The tasks that have to be handled are the following. It must control the motors to ensure precise and coordinated movement, gather data from various sensors to provide real-time feedback and analysis, and manage communication with a ground station for remote real time monitoring and control.

Figure 1: Pleurobot - Sensorized Robot

1.1 Objectives

The objectives given at the beginning of this project are as follows:

1. Improve the **sampling speed** and **robustness** of the microcontrollers that collect data from multiple sensors.
2. Increase the **bandwidth** of and reduce the **latency** in the communication between the onboard computer and multiple microcontrollers.
3. **Wireless communication** between the onboard computer and the user's laptop for remote control.
4. (opt) **GUI** for interaction with the user.

1.2 System overview

The software stack has to be built on top and tailored to the hardware architecture of the Pleurobot. The components of the system are as follows:

- **Raspberry Pi 5** - main onboard computer
- **Sensor Modules** - featuring a Raspberry Pi Pico and an AD7124-8 ADC, connected to five sensors
- **Dynamixel Servo Motors** - actuation system for the robot

The Raspberry Pi 5 is responsible for the high-level control of the robot, while the Raspberry Pi Picos are responsible for the data acquisition from the sensors.
The Raspberry Pi 5 is connected to the motors by means of a RS-485 serial bus. A Dynamixel U2D2 adapter is used as a USB to RS-485 converter. The Dynamixel Protocol 2.0 [2] is used for communication with the motors.
The communication between the Raspberry Pi 5 and the Raspberry Pi Picos is done through a RS-485 communication bus.

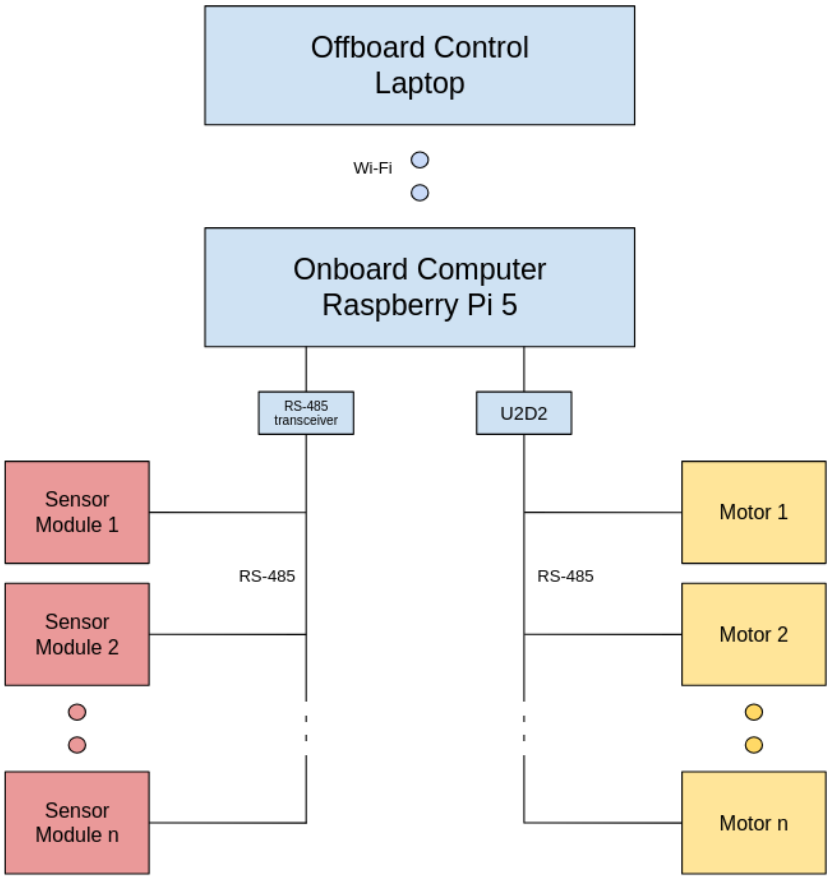


Figure 2: System Overview

1.3 Main Axes of Development

As seen in the previous sections, the system is complex and involves multiple components that need to work together seamlessly. For these components to be integrated effectively, the development of the software stack must be approached in a structured and systematic manner. The work has been therefore divided into the following main axes:

- **Sensor Modules** - firmware and communication protocol design and implementation
- **Development and Deployment Environment** - standardization and optimization of the development and deployment environment
- **ROS2 Integration** - integration of the sensor modules with the ROS2 framework

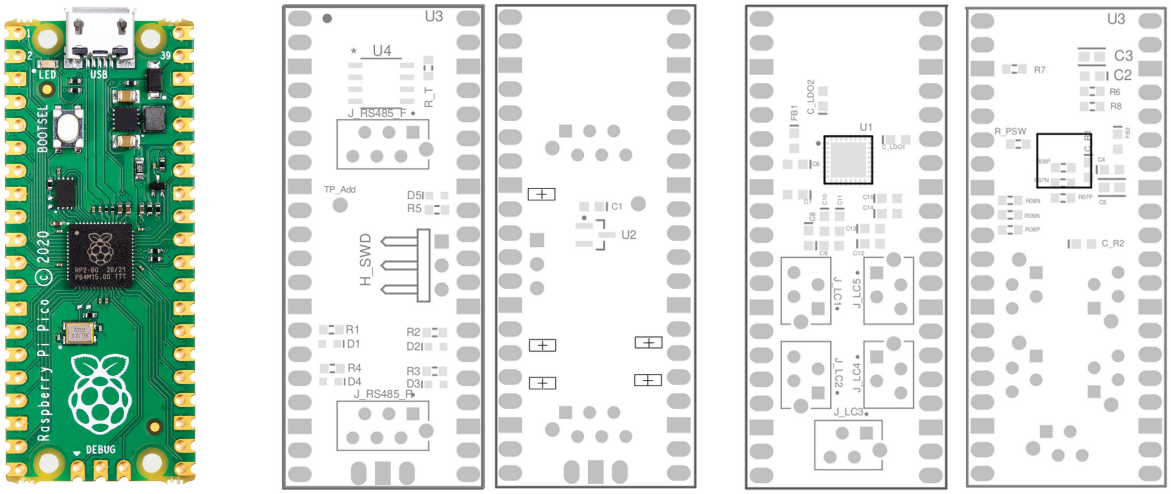
Each of these axes will be detailed in the following sections, outlining the objectives, methodologies, and results of the work carried out in each area.

2 Sensor Modules

2.1 Modules Description

Each sensor module consist of 3 parts, each one corresponding to a different PCB layer:

- **Top Layer:** Communication board, including RS-485 transceiver and connectors.
- **Middle Layer:** Raspberry Pi Pico microcontroller.
- **Bottom Layer:** AD7124-8 ADC and connectors.



(a) Raspberry Pi Pico (b) Communication PCB (c) ADC PCB

Figure 3: Sensor Modules Components

The AD7124 ADC chip is connected to the RP2040 using an SPI serial connection.
The RP2040 is also connected to the RS-485 transceiver, which allows communication with the Raspberry Pi 5. The transceiver is used in a half-duplex configuration, allowing the RP2040 to both send and receive data over the bus. The communication is managed by the RP2040 using the UART peripheral, and the direction of the communication is controlled by a GPIO pin connected to the transceiver, called RTS (Request To Send).
Together with the bus transceiver, the RP2040 is also connected to a Sync Trig line, that is shared between all the slaves and the master device.

2.2 Specifications

The main challenge of the given sensor system is the communication with the main onboard computer. The communication has to be reliable, ensuring fast and low-latency communication to allow fast feedback control loops.
For this reason, at the beginning of the project the specifications for the communication system were given. The specifications to be matched for the communication system are as follows:

Parameter	Value	Notes
Number of slaves	20	14 on the trunk + 1 for each leg + 2 backup
Data frame size	>= 40 bytes	Double precision reading values and timings of 5 channels each module. Not including ID, starting, ending, and error checking bytes
Sampling rate (min)	100 Hz	Ideally 200 Hz
Sampling rate (max)	1000 Hz	Potentially useful for the sensors on the feet
Latency (max)	3 ms	From one force sensor changes reading to the RPi 5 receives the filtered data. 10% of the intersegmental lag (12 joints, 1.5 cycles along the trunk) in the shortest swimming period (0.5s)
Data loss rate	< 1%	/

Figure 4: Communication Specifications

In the following sections an analysis of the original communication system will be provided, following the proposal of a new communication system. The performance of the two systems will be compared and the improvements of the new system will be analyzed.

2.3 Original Communication System Description

The original communication system involves a bus controller that manages the communication between the Raspberry Pi 5 and the Raspberry Pi Picos (slaves).

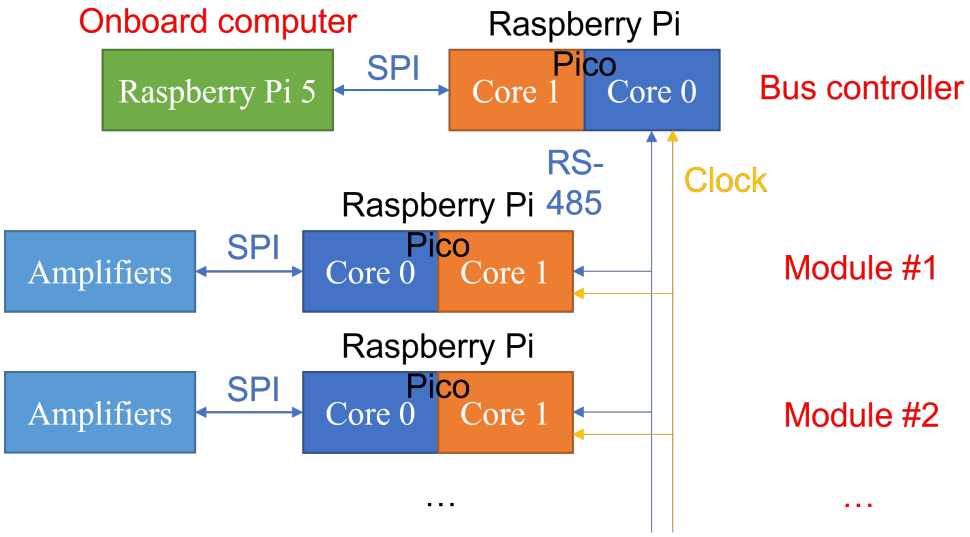


Figure 5: AD7124 Schematic

The bus controller polls each slave Pico for its data in a circular manner. As soon as the data from all the slaves is received, the data is then sent to the Raspberry Pi 5.

2.3.1 Data Protocol

The protocol for the data to be communicated is defined as follows:

Bus controller → Slave

Byte	Data
0	i (Start)
1	Slave ID
3	L (Command)
4	F (End)

Slave → Bus controller

Byte	Data
0	i (Start)
1	Slave ID
2-21	Channel raw readings (5 x 4 bytes)
22-41	Channel reading time (5 x 4 bytes)
42	Error checking (8-bit CRC)
43	f (End)

Figure 6: Slave to Bus Master communication

Bus controller → RPi 5

Byte	Data
0	i (Start)
1~4	Time (ms)
5~8	Bus communication success rate
9~664	ADC readings (41 x 16 = 656 bytes): ID + 40-byte data, from module 0 to 17
665	Error checking (8-bit CRC)
666	f (End)

Figure 7: Bus Master to Raspberry Pi 5 communication

2.3.2 Performance analysis

An analysis of the performance of the system will be conducted. First an estimation of the communication times will be done, then the best case and worst case scenarios for latency will be analyzed.
In both analysis we will make the following assumptions:

- No communication errors. Each data packet is sent and received correctly.
- No delays introduced by computation. As soon as the data is received from the bus controller, it is ready to be sent to the Raspberry Pi 5.
- Maximum theoretical bus speed of 1 Mbit/s.
- ADC filter word of 5, therefore 3840Hz sampling frequency.
- 20 slaves connected to the bus, allowing for a total of 100 sensors.

For each communication round, the bus controller has to query each Pico for its data, and then send the data to the Raspberry Pi 5. The following analysis provides an overview of the data exchanged and the time taken for a full communication cycle.
Data exchanged:

- 4 bytes are sent as a request from the master to each slave, and 44 bytes are sent from each slave back to the master.
- 666 bytes are sent from the master to the Raspberry Pi 5.

Time analysis:

- Total number of bytes per full communication round (from bus master to all slaves and back):

$$48 \text{ bytes} \cdot 20 \text{ slaves} = 7680 \text{ bits}$$

- Assuming a maximum teoretical bus speed of 1 Mbit/second:

$$\text{Total time per communication round} = \frac{7680 \text{ bits}}{1,000,000 \text{ bits/second}} = 7.68 \text{ ms}$$

Therefore we can assume that in absence of communication errors and not taking into account any other delays introduced by processing times and communication response delays, the data from each slave will be updated in the master every 8 ms.

- Total number of bytes per communication round:

$$666 \text{ bytes} = 5328 \text{ bits}$$

- Assuming a maximum teoretical bus speed of 1 Mbit/second also for the communication between the bus controller and the Raspberry Pi 5:

$$\text{Total time per communication round} = \frac{5328 \text{ bits}}{1,000,000 \text{ bits/second}} = 5.33 \text{ ms}$$

Best case scenario In the best possible scenario, the data has been just acquired from the ADC when the slave sends it to the bus controller, and when the the data from the slave is received by the bus controller, it is immediately sent to the Raspberry Pi 5. In this case, the total latency would be the sum of a single communication between a slave and the bus controller and the communication between the bus controller and the Raspberry Pi 5.

$$48 \text{ bytes} + 666 \text{ bytes} = 5712 \text{ bits}$$

$$\text{Minimum latency} = \frac{5712 \text{ bits}}{1,000,000 \text{ bits/second}} = 5.712 \text{ ms}$$

This assume that the data from the sensor is sent to the bus controller as soon as it is acquired and it is then sent to the Raspberry Pi 5 as soon as it is received by the bus controller, which is not realistic.

Worst case scenario In the worst case scenario, when the Raspberry Pi 5 needs the data, it has not yet received the new data from the bus controller. When the data from the bus controller is sent to the Raspberry Pi 5, the data from a specific slave is about to be received, so the available data is the oldest possible. The same applied to the ADC reading, which can be at most 1/3840 s old.

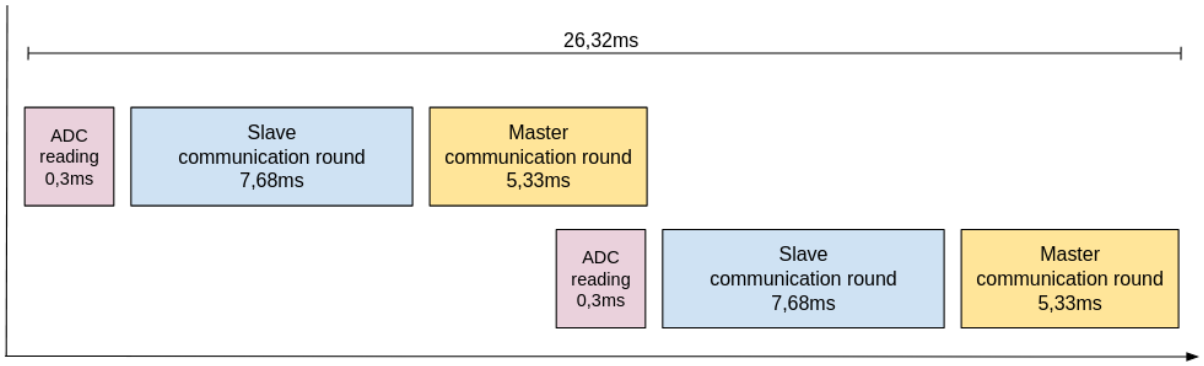


Figure 8: Maximum Latency

In this case, the total latency would be the sum of the time taken for an ADC reading, plus 2 times the sum of a full communication cycle and the time taken for the communication between the bus controller and the Raspberry Pi 5.

$$\text{Maximum latency} = 0.3 \text{ ms} + 2 \cdot (7.68 \text{ ms} + 5.33 \text{ ms}) = 26.32 \text{ ms}$$

Remark: this worst case scenario analysis is still optimistically irrealistic. This assumes that the devices involved (Pi Pico slave, bus controller, Raspberry Pi 5) are able to process data instantaneously (buffer storing, retrieving from memory, packet serialization, CRC computation, UART TX buffers), which is not the case. It also assumes no transmisson errors. In reality, the maximum latency will be greater than 26.31 ms.

2.3.3 Performance Measurements of the existing system

The existing system was already tested and benchmarked in terms of throughput.
The bus master was able to achieve a data publishing rate of 4000 data frames/second, which is enough for reading 20 modules at 200Hz.
However, this kind of test does not provide information about the latency of the system, which is crucial for the functioning of a real-time feedback control loop. As shown in the theoretical computation, the latency of this system is too high for the requirements of the project. Therefore, a new communication system has to be designed to meet the specifications given at the beginning of the project.

2.4 New Communication System

The proposed system changes aim to reduce complexity and improve efficiency by simplifying the communication protocol structure and eliminating the need for a bus controller.
The new system involves direct communication between the Raspberry Pi 5 and the Raspberry Pi Picos (slaves), with the Pi5 managing the communication and data exchange with the Picos.
The main sources of latency of the system are the following:

- The need for an **master/slave** communication protocol. The bus controller has to poll each slave for its data. This introduces a delay in the communication.
- The presence of a **bus master** that act as a bottleneck for the communication. The bus master has the function of a store and forward device, which introduces a delay in the communication.

2.4.1 Key Changes

1. **Remove Bus Controller:** The system eliminates the need for a bus controller, relying instead on Pi5 to directly read from the bus and manage communication.
2. **Simplified Communication Protocol:** The new protocol removes the need for the master/slave communication structure, optimizing the data exchange overhead by removing the need for polling.
    - **Communication initialization:** The data exchange begins with Pi5 sending a configuration message to all Picos, indicating the number of slaves and requesting the start of the communication.
    - **Circular data exchange:** Each Pico sends its data to Pi5 in sequence, with the next Pico in line waiting for its turn to transmit.

2.4.2 Data Protocol

The protocol for the data to be communicated is defined as follows:

- **Pi5 sends configuration message:**
    - **First byte:** first 3 bits are 1 and indicate the start of the configuration message. The next 5 bits represent the number of slaves.
    - **End byte:** all 8 bits are 1. It indicates the end of the configuration message.
- **Picos send sensor data:**
    - **First byte:** it has a value of 0x55, to indicate the start of the message.
    - **Second byte:** first 3 bits are 0, the other 5 are the Pico ID.
    - **Data bytes:** 15 bytes - 24 bits for each of the 5 sensors (ADC readings).
    - **Next call byte:** it encodes the id of the next Pico in line.

– **End byte:** it has a value of 0xAA. It indicates the end of the data message.

Note that in this new system, there is no CRC byte. The reason for this is the strict requirement on the latency performance. The RP2040 CPU runs at 125MHz, which means that only 125 CPU cycles are executed per each bit sent. This means that if the firmware is not carefully designed to avoid costly operations executed during the reading or writing to the serial communication, important delays or communication errors would be generated. The CRC computation is one of these costly operations, and it has been decided to remove it to reduce the latency of the system.
Considering the high robustness of a RS485 differential signalling, the CRC check is not strictly necessary. If an error is produced in the transmission, two things can happen:

- The error is in the signature of the message. In this case the circuar communication stops and the Pi5 will send a new configuration message to restart the communication.
- The error is in the data bytes. In this case the error would not be detected and the data would be corrupted. However, the data is sent at a high frequency, so the corrupted data would be overwritten by the new data in the next communication cycle.

As the former has never been observed in pratice, we assume that the error rate is negligible in our setup up to 4 sensor modules.

Pi5 Algorithm

1. **Initialization:** Pi5 begins by sending the configuration message containing the number of slaves.
2. **Retry Mechanism:** If no response is received from the slaves within a set timeout period, Pi5 resends the configuration message to reset and restart the communication cycle.

Pico Algorithm

1. **Receive Configuration Message:** Upon receiving a configuration message from Pi5, the Pico resets its configuration and restarts its communication routine. The configuration message from the Pi5 is signalled by a Rising interrupt of the Sync Trig line.
2. **Transmit Data:** Each Pico waits for its turn to send data. If the Pico detects that the previous Pico in sequence has sent its message, it transmits its own message.

2.4.3 Performance Analysis

The performance of the proposed communication system is analyzed in the following section, in the same way as the original system, and making the same assumptions.
As the start message is sent only once at the beginning of the communication, the only delay to be taken into account is the sending of the data from the Picos to the Raspberry Pi 5.

- Total number of bytes per full communication cycle:

$$19 \text{ bytes} \cdot 20 \text{ slaves} = 3040 \text{ bits}$$

- Assuming again a maximum theoretical bus speed of 1 Mbit/second:

$$\text{Total time per communication round} = \frac{3040 \text{ bits}}{1,000,000 \text{ bits/second}} = 3.04 \text{ ms}$$

Best case scenario In the best case scenario, when the data is needed on the Raspberry Pi 5, it has just been received from the slave. The data has been just acquired from the ADC when the slave sends it to the Raspberry Pi 5. In this case, the total latency would be the time taken for the communication between the Picos and the Raspberry Pi 5.

$$\text{Minimum latency} = \frac{152 \text{ bits}}{1,000,000 \text{ bits/second}} = 0.152 \text{ ms}$$

Worst case scenario In the worst case scenario, when the the data is needed on the Raspberry Pi 5, we get the oldest possible data, that arrived at most 3.04 ms before. To this we have to sum the maximum time between the ADC readings.

$$\text{Maximum latency} = 3.04 \text{ ms} + 0.3 \text{ ms} = 3.34 \text{ ms}$$

Remark: As in the analysis of the worst case scenario for the original system, the obtained value is still optimistic. However in this case, given the absence of any delays introduced by the bus controller (CRC computation, in memory buffer store and load) and the master/slave communication protocol, the latency will be closer to the calculated value.

Possible improvement As per the analysis, the proposed system is expected to have a lower latency compared to the original system. However, it just meets the specifications given. To further improve the system, the following changes can be made:

- **Reduce precision of the ADC:** the ADC can be set to a lower precision, which will reduce the number of bits sent by each Pico, and therefore the time taken for the communication. Instead of sending the whole 24 bits of the ADC reading, only the 16 most significant bits can be sent. This will reduce the number of bytes sent by each Pico from 19 to 14. This would lead to an improvement in the performances of the system of **26%**.

2.5 Comparison of Communication Systems

The following table provides a comparison between the original and the new communication systems, highlighting the difference in latency performance.

Feature	**Original System**	**New System**	**New with reduced ADC prec.**
Best Case Latency	5.712ms	0.152ms	0.112ms
Worst Case Latency	26.32ms	3.34ms	2.46 ms

Table 1: Comparison of Original and New Communication Systems

2.6 Firmware Implementation

2.6.1 Raspberry Pi Pico

The firmware has been written using the PlatformIO system, which allows for easy development and deployment of the firmware on the Raspberry Pi Pico. The firmware is written in C++. It uses the Arduino Pico libraries developed by Earle F. Philhower [6], on top of the RP2040 SDK provided by Raspberry Pi.
To keep the system as maintainable as possible, and modular, the firmware for the Raspberry Pi Pico is divided into several libraries, each handling a specific part of the functionality. This approach allows for easier debugging, testing, and future modifications.
By organizing the firmware into these modular libraries, the system becomes easier to maintain and extend. Each library can be developed and tested independently, reducing the complexity of the overall system and making it easier to identify and fix issues.
Each library contains its *README* file containing the documentation of its functionalities and how to use them.

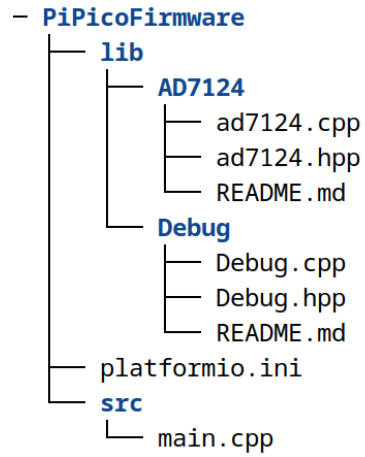


Figure 9: Firmware Structure for Raspberry Pi Pico

Debug Library The Debug Library provides functions for logging and debugging information. It includes methods to send debug messages over a serial connection, which can be useful for monitoring the system's behavior during development and troubleshooting.

AD7124 Library The AD7124 Library interfaces with the AD7124-8 ADC. It includes functions for configuring the ADC and reading sensor data. This library abstracts the complexity of the ADC, providing a simple interface for the main firmware to use.

Main Code The Main Code integrates the functionalities provided by the libraries. It initializes the system, configures the peripherals, and manages the main communication loop. The main code is responsible for coordinating the data acquisition from the sensors, processing the data, and transmitting it to the Raspberry Pi 5.

2.6.2 Raspberry Pi 5

The implementation for the Onboard Computer will be explained in detail in the next section, as it is part of the ROS2 integration.

3 ROS2

In the second axe of development, an integration system for all the software components of the Pleurobot was designed.
The following principles have been pursued in the design and choices for this integration system:

- **Modularity** - the system should be modular, splitting features and functionalities into different components that can be run independently.
- **Configurability** - the system should be configurable, allowing for easy modification at runtime of the parameters of the system.
- **Scalability** - the system should be scalable, allowing for the addition of new components and functionalities without the need of modifying the existing ones.
- **Usability and Maintainability** - the system should be easy to use and maintain, with clear interfaces and documentation.

3.1 ROS2 Overview

Based on said principles, the Robot Operating System 2 (ROS2 [1]) has been chosen as the integration system for the Pleurobot.
ROS2 is a set of software libraries and tools that help building robot applications.
A brief overview of what ROS provides:

- **Modularity** - ROS2 is designed to be modular, with a set of libraries that can be used to build robot applications. This allows for the reuse of existing components and the easy addition of new ones.
- **Already Existing Tools** - ROS2 provides a set of tools that help in the development of robot applications, such as visualization tools, simulation tools, and debugging tools.
- **Communication** - ROS2 provides a communication system that allows for the exchange of messages between different components, both on the same machine and on different machines.
- **Visualization** - ROS2 provides tools for visualization, such as RViz, that allow for real time feedback of the robot and its state.
- **Community** - ROS2 has a large community of users and developers that can provide support and help in the development of the software.
- **Scalability** - ROS2 is designed to be scalable, allowing for the development of complex robot applications.
- **Open Source** - ROS2 is open source, which means that it is free to use and can be modified and distributed by anyone.

Choice of ROS version The goal is to have a stable system with support that lasts as long as possible, so the *Humble Hawksbill* distribution has been chosen. It is a LTS (Long Term Support) version and its support will end in 2027, which seems a reasonable timeframe for the development of the system.

3.2 ROS2 in depth

ROS2 is a complex system, it offers a wide range of methods to implement the system, so a preliminary phase of the development has been about evaluating and selecting ROS features. [3] [4]

3.2.1 ROS components

- **Packages** are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), libraries, datasets, configuration files. Packages are the most atomic build item and release item in ROS.
- **Nodes** are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a control system usually comprises many nodes.
- **Topics**, a transport system with publish / subscribe semantics for data exchange between nodes. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.
- **Bags** are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

3.2.2 ROS interfaces

The ROS Interfaces are the way data can be exchanged between the components of the system. There are three primary styles of interfaces [5].
Below a brief description of those:

- **Topics**
  - Should be used for continuous data streams (sensor data, . . .).
  - Are for continuous data flow. Data might be published and subscribed at any time independent of any senders/receivers. Many to many connection. Callbacks receive data once it is available. The publisher decides when data is sent.
  - The definition of a message is done in a *.msg* file.
- **Services**
  - Should be used for remote procedure calls that terminate quickly. They should never be used for longer running processes, in particular processes that might be required to preempt if exceptional situations occur and they should never change or depend on state to avoid unwanted side effects for other nodes.
  - Simple blocking call. Mostly used for comparably fast tasks as requesting specific data. Semantically for processing requests.
  - The definition is done in a *.srv* file, which is composed by: Request, Result.
- **Actions**
  - Should be used for any discrete behavior that runs for a longer time but provides feedback during execution.
  - The most important property of actions is that they can be preempted.
  - More complex non-blocking background processing, used for longer tasks. Semantically for real-world actions.
  - The definition is done in a *.action* file, which is composed by: Goal, Result, Feedback.

3.2.3 ROS2 configuration system

The system to be developed carries complexity, and each component has multiple parameters that need to be configurable.
ROS2 uses YAML files for its configuration system to make managing settings simple, modular, and reusable. The features of the ROS2 configuration system are:

- **Human-Readable and Flexible** - YAML files are easy to understand and edit, making them great for defining node parameters like thresholds, modes, or paths.

- **Separation of Code and Configurations** - Configuration files allow developers to adjust a robot's behavior without changing the source code. This makes the system more modular and easier to maintain. This also removes the need to recompile the source code to change the parameters.
- **Scalability for Complex Systems** - YAML supports namespaces and hierarchical structures, which help manage configurations for multiple nodes in large systems.

3.2.4 Launch files

The system designed is quite complex, as all the components need to be started at the same time, with different configuration and on different machines.
In ROS2, this problem is solved using launch files.They can be written in Python and they are used to start up and configure multiple nodes and other components of a ROS2 system. They provide a flexible and powerful way to manage the startup process, allowing for complex configurations and conditional logic.
They allow for a modular building of the launch system, as each file can include other launch files, making it easy to reuse and organize launch configurations.

3.2.5 Nodes running on different machines

To enable the system to run nodes on different machines and still communicate effectively, ROS2's middleware, DDS (Data Distribution Service), is utilized. DDS allows for seamless communication between nodes regardless of their physical location, whether on the same machine or across a network.
As per the specifications, a part of the system will be run on an external laptop. For example, the system is designed to run the visualization node on a laptop, while other nodes are run on on the embedded computer on the robot.
The communication between nodes on different machines is facilitated by ROS2's discovery mechanism, which automatically detects and connects nodes within the same network. This allows for real-time data exchange and visualization, ensuring that the state of the robot and its environment can be monitored effectively from the laptop.
Therefore, as long as the laptop is connected to the same network as the robot, the nodes can communicate seamlessly, enabling remote monitoring and control of the robot.

3.2.6 ROS2 improvements over ROS1

ROS2 is an evolution of ROS1, and it brings several improvements over the previous version regarding the network features and performances. Some of the key improvements are:

- **Decentralized Communication** - In ROS1, a central process called `roscore` was required to facilitate node communication, creating a potential bottleneck and single point of failure. ROS2 eliminates the need for a central tracker by adopting a decentralized approach based on DDS (Data Distribution Service). Nodes communicate directly using a discovery mechanism, which enhances robustness and scalability. However, this discovery process may introduce a slight delay during node startup as nodes identify and connect to each other.
- **Configurable Quality of Service (QoS)** - ROS2 provides advanced communication control through DDS QoS policies, enabling fine-tuning of parameters like reliability, latency, and durability. This flexibility allows developers to optimize network performance for specific applications, ensuring efficient data exchange even in resource-constrained environments.
- **Improved Transport Protocols** - ROS1 primarily relied on TCP for communication, which, while reliable, incurs higher latency due to connection overhead. In contrast, ROS2 adopts UDP as the default transport protocol. UDP's lightweight nature reduces latency and is particularly well-suited for high-frequency, time-sensitive communication, such as sensor data streams or real-time control signals.
- **Reduced Latency and Overhead** - By leveraging DDS middleware, ROS2 minimizes communication latency and overhead, offering significant improvements in throughput and efficiency. The middleware ensures optimized data serialization, delivery, and management, catering to the demands of real-time robotic systems.

The performances of ROS2 have been tested and benchmarked in [11].

3.3 Implemented System

The system implemented, according to all the principles and features of ROS2, is complex, and will be described in the following sections.
The extract of the files created is the following:

```
pleurobot_ros2
├── latency_measurement
│   ├── CMakeLists.txt
│   ├── include
│   ├── package.xml
│   ├── README.md
│   └── src
│       ├── latency_subscriber.cpp
│       └── timestamp_publisher.cpp
├── pico_com
│   ├── CMakeLists.txt
│   ├── config
│   │   └── pico_com.yaml
│   ├── launch
│   │   └── pico_com.launch.py
│   ├── package.xml
│   ├── README.md
│   └── src
│       ├── pico_com.cpp
│       ├── pico_com.hpp
│       ├── rs485.cpp
│       └── rs485.hpp
├── pleurobot3_utils
│   ├── CMakeLists.txt
│   ├── gui
│   │   ├── pleurobot_bin_red.stl
│   │   ├── pleurobot.urdf
│   │   └── rvizConfig.rviz
│   ├── launch
│   │   └── visualization.launch.py
│   ├── package.xml
│   ├── README.md
│   └── src
│       ├── rviz_interface_node.cpp
│       └── rviz_interface_node.hpp
├── pleuro_msgs
│   ├── CMakeLists.txt
│   ├── msg
│   │   ├── ForceSensor.msg
│   │   └── LatencyMeas.msg
│   ├── package.xml
│   └── README.md
└── README.md

└── robot
    ├── CMakeLists.txt
    ├── config
    │   ├── motor_config.yaml
    │   └── test_motor_config.yaml
    ├── demo
    │   ├── demo2
    │   │   └── demo2.cpp
    │   └── transparent_FR_to_HR
    │       └── transparent_FR_to_HR.cpp
    ├── include
    │   ├── pleurobot_config.hpp
    │   ├── robot.hpp
    │   └── utils
    │       ├── misc_math.hpp
    │       ├── struct_defs.hpp
    │       └── utils.hpp
    ├── launch
    │   ├── demo2.launch.py
    │   ├── DXL_test.launch.py
    │   ├── sensor_motor_test.launch.py
    │   └── transparent_demo.launch.py
    ├── package.xml
    ├── README.md
    ├── src
    │   ├── pleurobot_config.cpp
    │   ├── robot.cpp
    │   └── utils
    │       ├── misc_math.cpp
    │       └── utils.cpp
    └── test
        ├── DXL_test
        │   └── DXL_test.cpp
        └── sensor_motor_test
            └── sensor_motor_test.cpp
```

Figure 10: ROS2 System Design

The system includes 5 packages:

- **robot** - package that includes the nodes that manage the high-level control of the robot.
- **pico_com** - package that includes the nodes that communicate with the sensor modules.
- **pleurobot3_utils** - package that includes the utility tools, such as the visualization files.
- **pleuro_msg** - package that includes the custom messages used in the system.
- **latency_measurement** - package that includes the nodes that measure the latency of the system.

3.3.1 Robot Package

The robot package is the core of the system, as it includes the nodes that manage the high-level control of the robot. The package is composed of the following components:

- **Robot class** - the robot class has taken inspiration from previous implentation of the Pleurobot. It extends the `KMR_dxl::BaseRobot` class. The class abstract the communication with the motors and provides a set of methods to control the robot. It includes methods for controlling single and multiple motors, and for getting feedback about the state of the motors, and hence the whole robot. The class is used by the controller nodes.
- **Utils** - a set of utility functions that are used by the code base.
- **Tests** - a set of tests that are used to validate the code base and the hardware. Two tests have been written so far:
  - *DXL_test* - a test that checks the communication with the motors.
  - *sensor_motor_test* - a test that makes a single motor move according to the data read from a single force sensor. Shown in the midterm presentation.
- **Demos** - a ROS2 node that manages the high-level control of the robot. The node subscribes to the data from the sensor modules and sends the commands to the robot.
- **Configurations** - Configuration files to set parameters for each component of the system.

  Considerable work has been done to improve the configuration setup. Originally, there were 2 configuration files, one to be used by the `KMR_dxl` library and one to be used by the controller. Both files included data about the motors configuration. Therefore the data was redundant and difficult to maintain, as it was stored in two different formats. Following the principles of modularity and maintainability, the idea was to have a single configuration file that could be used by both components. And split the configuration of the controller part in a separate file. Therefore the two files have been merge into one, that integrates now data for both components, and is read both by the ROS2 node and the `KMR_dxl` library. An excerpt of it is shown below:

```yaml
nbr_motors: 27
baudrate: 1000000
max_torque: 2.0
max_speed: 5.0

path_to_KMR_dxl: "/root/KMR_dxl"
motor_serial_port: "/dev/ttyUSB0"

motors:
  - ID: 1
    model: XM430_W350
    multiturn: 0

    enabled       : true
    axis_flip     : true
    home_position : 0.0
    angle_min     : -7854.0
    angle_max     :  7854.0

  - ID: 2
    model: XH540_W150
    multiturn: 0

    enabled       : true
    axis_flip     : true
    home_position : 0.0
    angle_min     : -7854.0
    angle_max     :  7854.0
```

Figure 11: Motor Config Excerpt

The parameters set for each motor include:

- *ID* - the ID of the motor, used by the Dynamixel Protocol.
- *Model* - the model of the Dynamixel motor.
- *Multiturn* - indicates if the motor is a multiturn motor.
- *Enabled* - allows for the motors to be disabled, in order to test the controller for single parts of the robot.

  - *Axis Flip* - allows for the motor to be flipped, in case the motor is mounted in the opposite direction.
  - *Home Position* - the position of the motor when it is in the starting position.
  - *Angle Min/Max* - the minimum and maximum angle that the motor reaches in normal operation of the robot.

  Any other component of the system (e.g. the controller that will be developed in the future) will have its own configuration file for its own parameters. The component will include the robot class, and the configuration file will be read by the robot class to set the parameters of the motors.

- **Launch files** - launch file to start each demo and test. They include the configuration files and the parameters for the nodes, as well as all the other components needed to run the specific demo or test (e.g. `pico_com`).

3.3.2 Pico Com Package

This package handles the communication with the sensor modules. The sensor modules are connected to the Raspberry Pi 5 via the RS-485 bus, as described in depth in the previous sections. The package is composed of the following components:

- **Pico Com Node** - a ROS2 node that communicates with the sensor modules via the RS-485 bus. The node reads the data from the sensor modules and publishes it on a topic.
- **RS-485 library** - a C++ library that handles the communication with the sensor modules. The library is used by the Pico Com Node to send and receive data from the sensor modules. It is a class that exposes the following methods:
  - *RS485(std::string port, int baud_rate, int rts_pin, int sync_pin)* - constructor that initializes the serial port, the baud rate, and the pins of the *RTS* and *SYNC_TRIG* lines. The *RTS* line is used to handle the half-duplex setup of the communication, as explained in 2.1. The *SYNC_TRIG* line is used to synchronize the sending of the configuration message at the beginning of the communication, as explained in 2.4.2.
  - *void sendStartMessage(int num_picos)* - sends the configuration message to all the Picos connected to the bus.
  - *bool readByte(uint8_t *buffer)* - reads a byte from the serial port and stores it in the buffer.
  - *void closeSerial()* - closes the serial port.
- **Configuration file** - a YAML file that contains the configuration parameters for the node. It includes:
  - the serial port to which the sensors are connected
  - the baud rate of the communication
  - the pins of the *RTS* and *SYNC_TRIG* lines
  - the number of sensor modules connected
  - the timeout setting in milliseconds. After no data is received for this time, the node will restard the communication bus by sending the configuration message. Set to -1 to disable the timeout
- **Launch File** - a launch file that starts the Pico Com Node with the specified configuration parameters.

The command to start the Pico Com system is the following:

```
ros2 launch pico_com pico_com.launch.py
```

However, this launch file is included in other launch files, as the Pico Com system is a fundamental part of the whole system. Therefore, it will not be necessary to explicit use this command, but it will rather be included in the launch files of the other packages.

3.3.3 Pleurobot3 Utils Package

The Utils package is meant to include various generic tools, such as the one for the visualization. To be able to both visualize what is happening during both simulation and real execution, and have a real-time feedback of the system, the visualization was a necessary and valuable tool to be developed.
The RViz, included in the ROS suite of tools, offers the possibility to easily create visualizations based on markers that are published on specific topics.
The RViz tool allows for visualization of both static and dynamic data, and it is a powerful tool for debugging and monitoring the system.
In this first prototype of the system, the visualization is quite simple, as it includes only the static robot and the visualization of data coming from the four 3-axis load cells on each leg.
The visualization subsystem is composed of the following components:

- **RViz Interface Node** - a ROS2 node that subscribes to the data from the force sensors and publishes the visualization markers for RViz.
- **STL file** - the 3D model of the Pleurobot. The file has been manipulated to be shrinked in size and simplified for a faster loading of the RViz tool.
- **URDF file** - the URDF file that describes the robot model. In this prototype it only includes the main STL file of the whole robot, but in the future it will include all the links and joints of the robot.
- **RViz Configuration File** - a configuration file that defines the layout and appearance of the visualization in RViz. It includes the path of the URDF file and the topics to subscribe to.
- **Launch file** - a launch file that starts the RViz tool, the RViz Interface Node, and the static transform publisher.

The static transform publisher is a node that publishes the static transforms between the world frame and the robot frame, in order to visualize the robot in the right position. This position is fixed in the launch file.

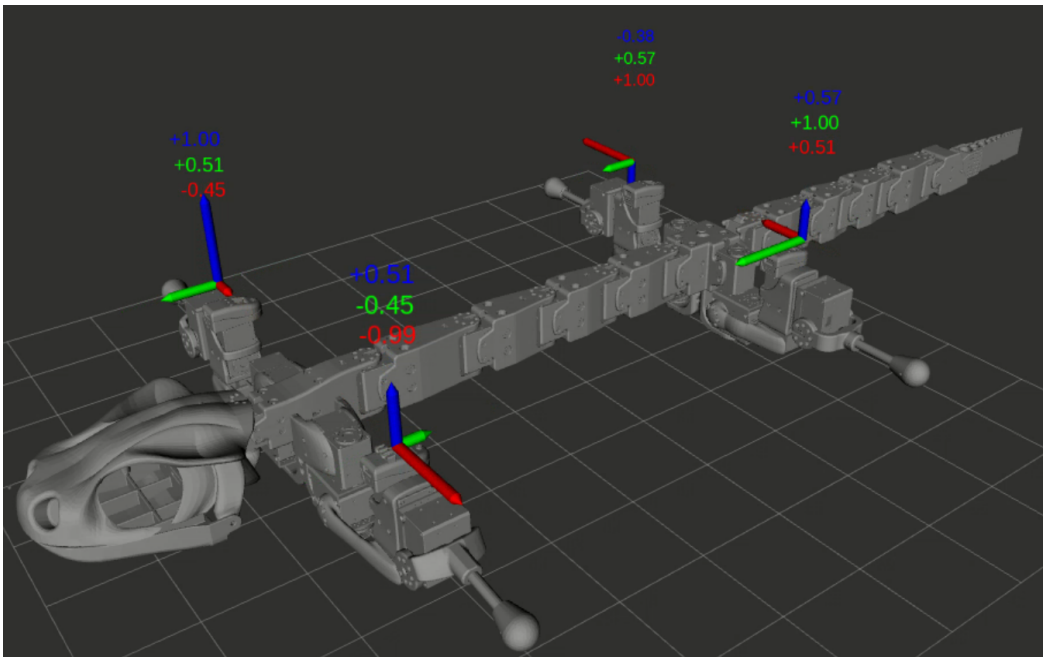


Figure 12: Visualization of the Pleurobot3 in RViz

To visualize the data from the four 3-axis load cells, the RViz Interface Node subscribes to the topics where the data is published and creates the visualization markers for RViz. The markers are then published on a specific topic that RViz listens to, and the data is visualized in the RViz tool.
The command to start the visualization system is the following:

```
ros2 launch pleurobot3_utils visualization.launch.py
```

3.3.4 Pleuro Msg Package

ROS2 allows for the definition of custom messages, that can be used to exchange data between nodes. The custom messages are defined in a *.msg* file, and then compiled to generate the necessary code to use the messages in the system.
The messages defined so far are the following 2:

- **ForceSensor.msg** - a message that is published by the *pico_com* node, includes the data coming from the sensor modules.
- **LatencyMeas.msg** - a message that is used as a dummy payload for the latency measurement system. It includes a timestamp and 100 32bits integers.

3.3.5 Latency Measurement Package

The latency measurement package is designed to measure the latency of the message passing system of ROS2. This is crucial to ensure that the system meets the real-time requirements of the Pleurobot.
The latency measurement subsystem is composed of the following components:

- **Latency Publisher Node** - a ROS2 node that publishes the *LatencyMeas.msg* message at a fixed rate.
- **Latency Subscriber Node** - a ROS2 node that subscribes to the *LatencyMeas.msg* message and measures the time difference between the timestamp in the message and the current time.

To commands are used to start the latency measurement system. On two shells of the same machine, or on two different machines, the following commands are used:

```
ros2 run latency_measurement latency_subscriber
ros2 run latency_measurement timestamp_publisher
```

The two commands start the publisher and the subscriber nodes, and the latency measurement system is started, as shown in the following graph:

Figure 13: Latency Measurement System

The results of the latency measurement executed on the Raspberry Pi 5, with a payload emulating data from 100 sensors, messages published at 1000Hz rate, are visualized in the following figure:

```
Messages received: 50000, Frequency: 1000.31 Hz, Average latency: 0.046 ms
```

Figure 14: Latency Measurement Results

As shown in the figure, the latency of the system is around 50 us, which is more than enough for the real-time requirements of the Pleurobot.

Latency over Network The same system can be used to measure latency over a network. To do so, the two nodes must be run on different machines. In this case, the latency is largerly dependendent on the network. In an ethernet network, the latency would be sub-millisecond, whereas in a common Wi-Fi network it would reach numbers around 6/7 ms. Therefore, the ROS2 induced latency is considerably lower than the network latency.

Remark To be able to measure the latency between messages on two different machines, it is important to ensure that their clock is synchronized. To do so, the NTP (Network Time Protocol) can be used to synchronize the clocks of the machines. Install a service such as Chrony [7] to synchronize the clocks of the machines.

4 Work Environment Standardization

ROS is quite a complex system to install and maintain. It has strict requirements on the operating system and the libraries that have to be installed.
For this project, a few libraries has also specific requirements, such as the `KMR_dxl` library, that requires specific modifications to the standard `Dynamixel SDK` source code. (according to the library documentation [8]).
In the past, the installation of all those libraries on a new machine was a time-consuming task, and it was easy to make mistakes. Therefore, setting up a new machine for development, or updating a component, quickly became a tedious and complex task.
This kind of system is not easily scalable, and it is difficult to share the same environment across multiple machines.
To solve this problem, a new standardized environment was designed. This environment is based on a Docker image that contains all the necessary libraries and tools to develop the project.

4.1 Docker

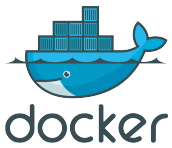


Figure 15: Docker Logo

Docker was chosen as the solution for standardizing the work environment for several reasons:

- **Consistency Across Machines**: Docker encapsulates an entire software environment, including libraries, dependencies, and tools, into a single container. This ensures that every machine running the Docker container operates in an identical environment, eliminating discrepancies caused by differences in installed software or configurations.
- **Ease of Deployment**: Setting up a development environment becomes as simple as running a single command to build and execute the pre-configured Docker image. This drastically reduces the time required for onboarding new machines or developers and minimizes the risk of errors during installation.
- **Portability**: Docker containers are highly portable and can run on any system that supports Docker, regardless of the underlying operating system or architecture. This is particularly important given the project's need to support different platforms, including Raspberry Pi and personal computers.
- **Scalability and Maintainability**: Docker simplifies scaling the system to additional machines or environments. Updates to the environment, such as adding new libraries or modifying existing ones, can be made to the Docker image and propagated by simply rebuilding and redeploying the image across machines.
- **Isolation of Dependencies**: Docker isolates the project's dependencies from the host operating system. This ensures that updates or changes to the host system (e.g., upgrading the OS or installing new software) do not interfere with the development environment. Conversely, changes within the Docker container do not impact the host system.
- **Support for Complex Dependencies**: Certain libraries, such as the `KMR_dxl` library, require specific modifications and configurations. Docker allows these customizations to be baked into the image, ensuring they are correctly applied every time and eliminating the need for manual intervention.

By leveraging Docker, the project benefits from a robust, scalable, and developer-friendly solution that addresses the challenges of managing complex software environments.

4.1.1 Docker Performance

One concern with using Docker is the potential performance overhead introduced by running applications within containers.
However, Docker containers have been chosen instead of virtual machines. Containers share the host system's kernel and do not require a separate guest OS.
This means that every call to a system function is not translated by a hypervisor, as in the case of virtual machines, but is directly executed by the host system. This excludes any overhead that could be introduced by the virtualization layer.
Also, Docker is executed in our setup using the host network, which means that the network calls are not translated by the Docker network stack, but are directly executed by the host system, to avoid the only overhead that could be introduced by Docker.
The performance of Docker containers has been benchmarked in [10]. The results confirm that the overhead introduced is negligible.

4.2 Repositories Structure

The following repositories make up the code base for the Pleurobot3 robot:

- **Pleurobot ROS2** - contains the ROS2 packages. This code will be run on the Raspberry Pi 5 inside a Docker container.
- **Pleurobot ROS2 Docker** - contains the scripts for the generation and deployment of the Docker image.
- **Pleurobot firmware** - contains the code to be run on the Raspberry Pi Picos to read the data from the force sensors and send it to the Raspberry Pi 5.

The **Pleurobot ROS2** and **Pleurobot firmware** repositories are the main code bases for the project, already discussed in previous sections.
The Docker image is built using the `pleurobot3_ros2_docker` repository. The repository structure is shown in Figure 16.

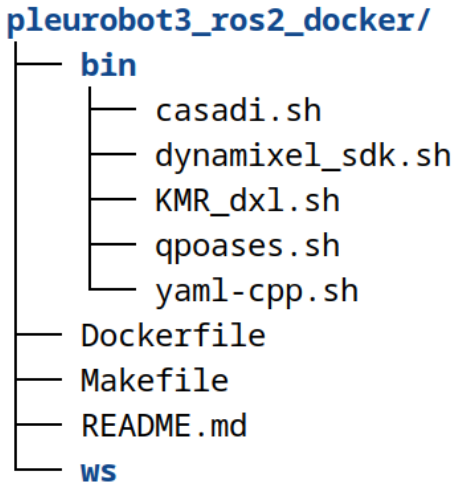


Figure 16: Docker environment repository structure

The main files are the following:

- `Makefile` - contains the commands to be run to manage the image building and deployment.
- `Dockerfile` - contains the image definition.
- `bin` directory - contains the scripts that are run during the image building to install all the dependencies.

4.3 Dependencies Installed

The scripts in the `bin` directory are used to install the dependencies.
Each dependency must be enabled in the `Dockerfile`.
Most of the dependencies follow a standard download and CMake installation process. However, a few require specific modifications or hacks, which are detailed below:

- **Casadi** - standard installation, with the possibility to set the version in the script.
- **KMR_dxl library** - includes a fix for an error in the Makefile to allow compilation on the `aarch64` (Raspberry Pi 5) architecture.
- **Dynamixel SDK** - modified according to the `KMR_dxl` library documentation [8]. Changes include fixing errors in the Makefile and enabling low-latency mode for serial communication.
- **qpOASES** - standard installation.
- **yaml-cpp** - standard installation.

4.4 Setting Up a New Machine

4.4.1 Initial setup

The first step to setup a new machine is the operating system. For the onboard computer, Raspberry Pi OS has been used, but the system would also work on Ubuntu and other operating systems. The machine has to be connected to the network for the installation of the necessary software.
In new versions of Ubuntu and Raspberry Pi OS, the NetworkManager is installed to manage the network connections. Below, a quick guide to set the Wi-Fi connection and static IP, needed to easily reach the machine.
To list the available networks run the following command:

```
nmcli dev wifi list
```

To connect to the desired WiFi network, use:

```
nmcli dev wifi connect "<SSID>" password "<password>"
```

Replace `<SSID>` with the network name and `<password>` with the WiFi password.
To check the connection status:

```
nmcli connection show
```

To set a static IP, first identify the connection name and then run the following:

```
nmcli connection modify "<connection_name>" \
    ipv4.addresses <static_ip>/<subnet_mask> \
    ipv4.gateway <gateway> ipv4.method manual
```

The data used for the Raspberry Pi 5 are the following:

- **IPv4** - `192.168.21.160`
- **Subnet Mask** - `255.255.255.0`
- **Gateway** - `192.168.21.1`

Then restart the connection to apply changes:

```
nmcli connection up "<connection_name>"
```

4.4.2 Docker Installation

Follow the guide for your operating system from the official website [9].

4.4.3 Clone Repositories

The following commands will let you clone this repository and the Pleurobot ROS2 repository into the correct directory configuration:

```
git clone git@ponyo.epfl.ch:qiyuan.fu/pleurobot3_ros2_docker.git
cd pleurobot3_ros2_docker
mkdir -p ws/src && cd ws/src
git clone git@ponyo.epfl.ch:qiyuan.fu/pleurobot_ros2.git
```

4.4.4 Build and Deploy Docker Image

The following commands will let you manage the Docker image. These commands must be run in the `pleurobot3_ros2_docker` directory.

- To build the image:

  ```
  make build
  ```

- To run the container:

  ```
  make run
  ```

- To get a shell from a running container:

  ```
  make attach
  ```

4.4.5 ROS2 Workspace Build

To build the ROS2 workspace inside the Docker container, run the following commands:

```
colcon build --symlink-install
```

The first time you run this command, it will take some time to build all the packages. Subsequent runs will be faster due to already built packages. **Note:** Any changes made inside the container are limited to that execution. The only folder that is mounted during execution is `/root/ros2_ws`. Changes to that directory are permanent. To make a change permanent for future container runs, you must adapt the Dockerfile and rebuild the image using the `make build` command.

4.5 Operating Systems

The Docker image has been successfully tested on the following systems:

- **Raspberry Pi 5**:
  - OS: Raspberry Pi OS GNU/Linux 12 (bookworm)
  - Kernel: `Linux raspberrypi 6.6.51+rpt-rpi-2712 #1 SMP PREEMPT Debian 1:6.6.51-1+rpt2 (2024-10-01) aarch64 GNU/Linux`
- **Personal Computer**:
  - OS: Debian GNU/Linux 12 (bookworm)
  - Kernel: `Linux 6.1.0-25-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.106-3 (2024-08-26) x86_64 GNU/Linux`

The information was retrieved using the `uname -a` command. The Docker image is expected to work on any Docker-supported system, including Windows via WSL (Windows Subsystem for Linux).

5 Testing and Validation

Ensuring the reliability and performance of the Pleurobot's system was a crucial part of this project. Throughout the development process, tests were conducted to validate the system's functionality under various conditions. These tests ensured that both the communication system and firmware were robust and capable of meeting real-time and performance requirements.
At the conclusion of the project, demonstrations were created to showcase the capabilities of the system. These demos highlighted its features, providing a practical evaluation of the system's effectiveness as a base framework for future development.

5.1 First Test: Validation of Communication System

To validate the new circular communication system, a series of tests were performed to measure its performance and reliability during data exchange between the Raspberry Pi 5 (master device) and the Raspberry Pi Picos (slave devices).

5.1.1 Test Setup

The setup for the first test was as follows:

- **Master Device:** Raspberry Pi 5, managing the communication and data flow.
- **Slave Devices:** Three Raspberry Pi Picos simulating sensor modules by sending dummy sensor data.

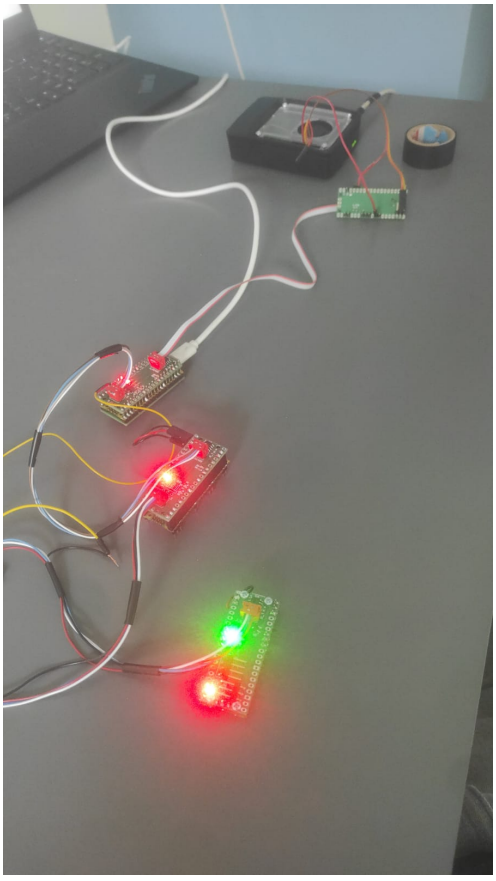


Figure 17: Communication Test Setup

5.1.2 Test Results

Two main aspects were evaluated during the test: communication reliability and system throughput.

- **Reliability:** A potential issue identified was the possibility of system stops if a slave device failed to respond. To address this, the Raspberry Pi 5 was configured to resend the configuration message in case of non-response. The system was run continuously for 3 hours, during which **no stops or interruptions were observed**, validating the system's robustness.
- **Throughput:** The communication frequency was measured by monitoring data transfer from the slave devices to the master device. The ROS2 node achieved a data publishing rate of **3000 Hz** for three slaves, equivalent to **1000 Hz per slave**. This performance meets the requirements for real-time data acquisition and processing.

5.2 Demonstrations

To further evaluate and showcase the system's capabilities, demonstrations were conducted. These demos utilized the integrated sensor system and control algorithms to highlight the Pleurobot's functionalities.

5.2.1 Sensors Mounted

For the demonstrations, the four legs of the Pleurobot were equipped with three-axis force sensors, one for each leg.

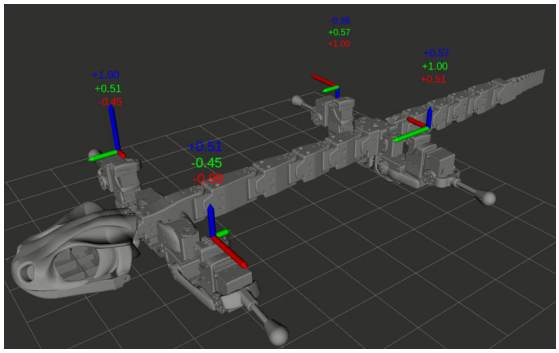


Figure 18: Sensors Mounted on Pleurobot's Legs

In this setup with 4 sensors, we obtained a performance in terms of sampling time of over 1 kHz for all the sensors.

Comparison of performances with original system Having 1kHz of sampling time for 4 sensors means that a setup with 20 sensors would have a sampling time of 200 Hz. This value is the same as the original system, but the crucial difference is about the latency of the communication system. In this case the latency is much lower than the original system, as shown in the previous sections.

5.2.2 Demo Planning

The demonstrations were planned to showcase specific features of the system, such as force-based feedback and safe movement control.
Unfortunately, due to time constraints, the sensors were mounted on all four legs only on the last day of the semester. Earlier sensor integration would have allowed for more comprehensive demos.
Despite this, two meaningful demonstrations were executed:

- **Demo 1:** Transparent Force Reproduction (FR to HR)
- **Demo 2:** Safe Movement with Force Threshold

Those demos leverage the use of a single 3-axis force sensor, mounted on the front-right leg, as shown in the picture below:

Figure 19: Single 3-axis Force Sensor Mounted on Front-Right Leg

5.3 Demo 1: Transparent Force Reproduction

This demonstration showcased the system's ability to reproduce forces in real-time using proportional control. The following process was implemented:

- Data from the three-axis force sensors on the **FR (front-right) leg** was read and filtered using an Exponential Moving Average (EMA) filter to reduce noise.
- The filtered force data was mapped to a corresponding position using a proportional controller.
- The computed position commands were sent to the **HR (hind-right) leg** motors, enabling it to mimic the force applied to the FR leg in real time.

This demo demonstrated effective sensor integration and precise control loop functionality.

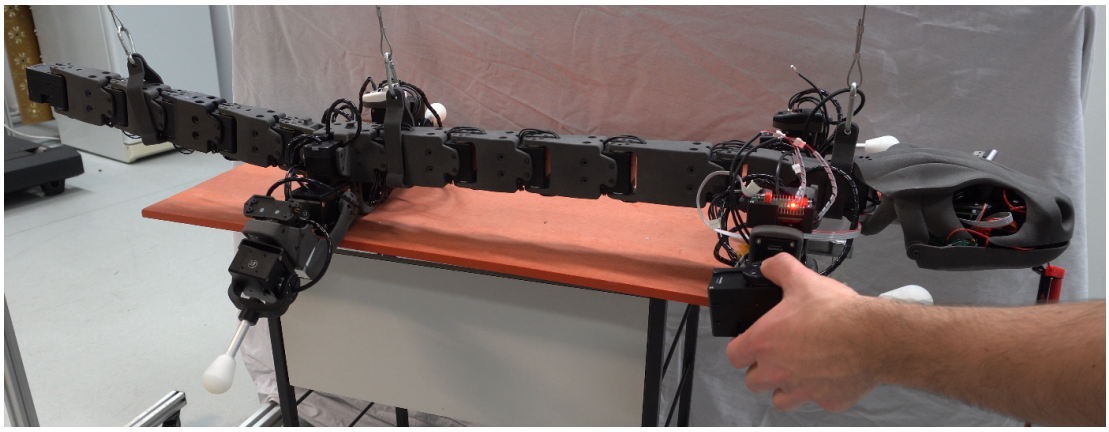


Figure 20: Transparent Demo

5.4 Demo 2: Safe Movement with Force Threshold

The second demonstration highlighted the safety features that could be implemented thanks to the real-time data acquisition, designed to stop movement when excessive force was detected. The process was as follows:

- The legs were moved using a sinusoidal setpoint function to simulate walking or rhythmic motion.
- If the force measured by the sensors exceeded a predefined threshold, the movement was stopped immediately to prevent potential damage.
- A hysteresis mechanism was implemented to avoid frequent triggering of stop commands due to small fluctuations around the threshold.

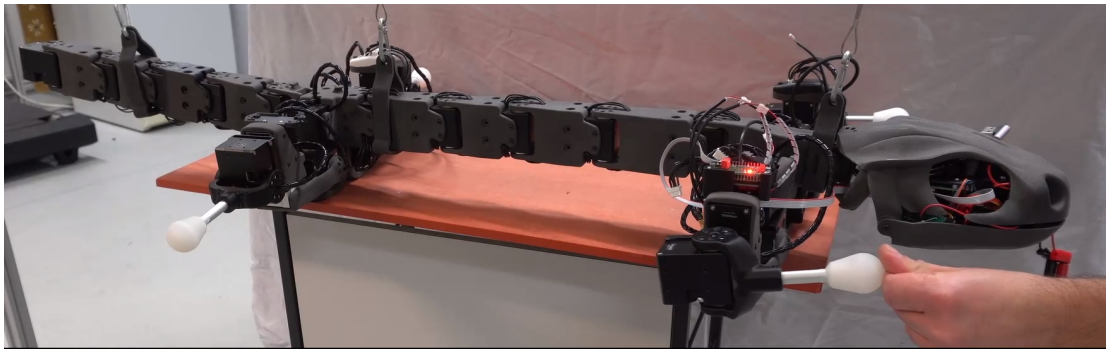


Figure 21: Safety Demo

6 Future work

The current project lays a strong foundation for the continued development of the Pleurobot, with several opportunities for further improvement and expansion. Below are the key areas identified for future work:

6.1 Robot Controller

The work presented in this report focused on the design of the software framework needed to control the Pleurobot.
Future work should focus on the development of control algorithms on top of this framework, that leverage its features and capabilities, such as the real-time sensor data acquisition.

6.2 PCB for Raspberry Pi 5

In the current setup, the connection to the Raspberry Pi 5 relies on an adapted PCB and jumper wires, which can be prone to signal degradation and interference.
The current setup is shown in the picture below:

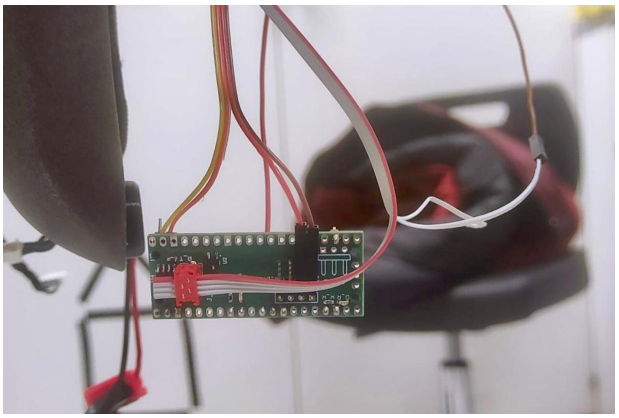


Figure 22: Current PCB setup

This setup can be improved by designing a custom PCB that connects directly to the Raspberry Pi 5 40-pin GPIO connector, in order to improve the stability of the connection and tidy up the wiring.

6.3 Enhanced Visualization

The visualization tool currently in place provide basic real-time feedback on just sensor data from the 4 legs sensors. Future developments could include:

- Comprehensive integration of the Pleurobot's kinematic model into the visualization system using URDF files, enabling the visualization of the robot more detailed and interactive simulations.
- Adding monitoring features to the visualization tool, such as motor communication delay, CPU and memory usage, to help diagnose performance issues.

6.4 Microcontroller Optimization

In the current version of the sensor modules, we are close to the limit of what the microcontroller can do. The baudrate of the RS-485 has been set to 1Mbps, and the main system clock of the RP2040 is 125MHz. This means that only 125 CPU cycles are executed per each bit sent. This means that if the firmware is not carefully designed to avoid costly operations executed during the reading or writing to the serial communication, important delays or communication errors would be generated.
To overcome this limitation and further improve the performance of the sensor modules, two main directions are suggested:

- **Overclocking:** it has been shown that the RP2040 chip (the Pi Pico CPU) is quite reliable to overclocking, which could be used to increase the main system clock and reduce the time needed to execute operations.

- **Alternative Microcontroller Models:** Exploring more powerful microcontroller options that offer higher computational capabilities and lower latency without compromising energy efficiency. Options such as the RP2350 (the second generation of the Raspberry Pi Pico, following the RP2040), or ESP32 (also offering Wi-Fi communication, useful for OTA updates and configuration) or STM32 (very energy efficient if the correct model is chosen) could be considered.

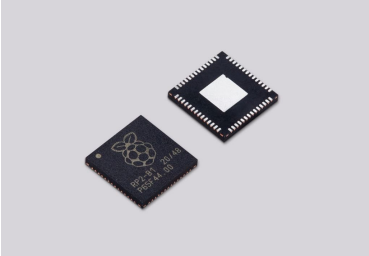


Figure 23: RP2350 Microcontroller

6.5 New Sensor Modules

As explained in previous sections, the sensor modules are currently composed by a 3 layer stack of PCBs. The components of this stack are:

- A custom PCB with the force sensors and the ADC.
- A Raspberry Pi Pico.
- A custom PCB with the RS-485 communication module.

The stack is compact and functional, but its size could be lowered by designing a new PCB that integrates all the components. On a PCB of a size similar to the Raspberry Pi Pico (21.4mm x 51mm), it would be possible to integrate the microcontroller, the ADC and the transceiver chip, as well as all the connectors for both the sensor and the communication bus.
This would optimize the space used by the sensor modules, allowing for more modules to be placed on the robot, and would also reduce the complexity of the system, as there would be only one PCB to manage, instead of 3.

6.6 Kernel Module Development

In the current version of the `pico_com` ROS2 node, a thread is continuosly actively polling the serial device. This ensures that no latency is introduced by buffers from the operating system, but it is not the most efficient way to manage the communication, as it introduces a lot of overhead CPU usage.
This means that a full core of the CPU is used to manage the communication, which could be used for other tasks, such as control algorithms or visualization.
To have a similar latency performance without compromising the CPU usage, it would be possible to exploit the operating system kernel functionalities to handle the communication. By developing a custom kernel module, the communication could be managed at a lower level, leveraging CPU interrupts and freeing the CPU from the polling task.

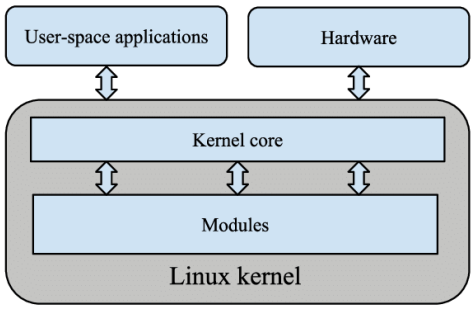


Figure 24: Linux Kernel Structure

7 Conclusion

This project focused on the development of a robust firmware and communication system for the sensorized Pleurobot, leveraging modular and scalable technologies to achieve significant improvements in latency, sampling speed, and reliability.
By integrating ROS2 and Docker, the work also established a standardized and efficient environment for development and deployment, ensuring consistency across platforms and simplifying future enhancements.
Key achievements include the design and implementation of a new communication protocol that eliminated the bottlenecks introduced by the original bus controller, reducing latency and enhancing system robustness.
The firmware's modular architecture and the inclusion of real-time visualization tools further contributed to a system that is both efficient and user-friendly.
Testing validated the system's performance, demonstrating its reliability in real-world conditions.
Looking forward, several avenues for improvement and expansion have been identified. These include further optimization of the communication system, development of custom PCBs, and enhanced visualization capabilities.
Overall, this project has provided a solid foundation for the continued evolution of the Pleurobot, paving the way for new software, algorithms and research to be carried on on this robotic platform.
The project's success was made possible by the support Supervisor Qiyuan Fu and Professor Auke Ijspeert of the Biorobotics Laboratory at EPFL.

8 References

References

[1] ROS2 - Robot Operating System https://index.ros.org/doc/ros2/

[2] Dynamixel Communication Protocol 2.0 https://emanual.robotis.com/docs/en/dxl/protocol2/

[3] ROS2 Documentation - Introduction http://wiki.ros.org/ROS/Introduction

[4] ROS2 Documentation - Concepts http://wiki.ros.org/ROS/Concepts

[5] ROS2 Documentation - Topics - Services - Actions https://docs.ros.org/en/humble/How-To-Guides/Topics-Services-Actions.html

[6] *Earle F. Philhower* - Raspberry Pi Pico Arduino core https://github.com/earlephilhower/arduino-pico

[7] Chrony - NTP client/server https://chrony-project.org

[8] KMR_dxl installation documentation https://github.com/KM-RoBoTa/KMR_dxl/blob/main/docs/markdown_sources/setup_git.md

[9] Docker installation guide https://docs.docker.com/engine/install/

[10] Benchmarking Container Performance https://blog.hathora.dev/benchmarking-container-performance/

[11] Latency Analysis of ROS2 Multi-Node Systems https://www.barkhauseninstitut.org/fileadmin/user_upload/Publikationen/2021/2021_Kronauer_Latency.pdf

Note: All the links included in the Reference have been accessed on January 4, 2025.