Antonio Lago Araújo Seixas
Vanderson da Silva dos Santos

Embedded systems simulator for robotics applications

São Paulo, Brazil

2025

Antonio Lago Araújo Seixas
Vanderson da Silva dos Santos

Embedded systems simulator for robotics applications

Work presented to Escola Politécnica da Universidade de São Paulo for undergraduate conclusion.

São Paulo, Brazil
2025

Antonio Lago Araújo Seixas
Vanderson da Silva dos Santos

Embedded systems simulator for robotics applications

Work presented to Escola Politécnica da Universidade de São Paulo for undergraduate conclusion.

Universidade de São Paulo – USP

Escola Politénica

Undergraduate Program

Supervisor: Prof. Dr. Bruno de Carvalho Albertini

São Paulo, Brazil
2025

Antonio Lago Araújo Seixas
Vanderson da Silva dos Santos

Embedded systems simulator for robotics applications

Work presented to Escola Politécnica da Universidade de São Paulo for undergraduate conclusion.

Prof. Dr. Bruno de Carvalho Albertini
Supervisor

São Paulo, Brazil
2025

Antonio Lago Araújo Seixas
Vanderson da Silva dos Santos
Embedded systems simulator
for robotics applications/ Antonio Lago Araújo Seixas
Vanderson da Silva dos Santos. – São Paulo, Brazil, 2025-
74p. : il. (algumas color.) ; 30 cm.

Supervisor: Prof. Dr. Bruno de Carvalho Albertini

Final Paper (Undergraduate) – Universidade de São Paulo – USP
Escola Politénica
Undergraduate Program, 2025.

1. Palavra-chave1. 2. Palavra-chave2. 2. Palavra-chave3. I. Orientador. II. Universidade xxx. III. Faculdade de xxx. IV. Título

This work is dedicated to all robotics enthusiasts and all young people who aspire at least once to contribute to a better world.

Agradecimentos

Ao nosso professor orientador, Bruno Albertini, por seus conselhos e por sua paciência ao nos orientar durante todo o trabalho de conclusão de curso.

Aos nossos amigos Lucas Haug e Lucas Schneider, por terem sido nossos padrinhos no mundo do desenvolvimento de firmware e pelo companheirismo ao longo dos últimos anos.

À Universidade de São Paulo pelas oportunidades e a equipe de robótica Thunderatz, não apenas nos proporcionou uma base teórica e prática incrível durante os nossos primeiros cinco anos de graduação, mas também tornou nossa jornada acadêmica muito mais prazerosa e únicas.

- Vanderson:

Primeiramente, à minha família, que me deram suporte, tanto emocional quanto financeiro durante toda a minha vida e graduação. Agradeço também por sempre acreditarem no meu potencial muito mais do que eu mesmo customo acreditar.

À minha dupla de trabalho de conclusão de curso, Antonio, pelas noites mal dormidas estudando e trabalhando, tanto no TCC quanto em outras atividades acadêmicas. Agradeço também pela companhia nesses últimos 5 anos de graduação e por sempre me lembrar dos compromissos que eu nem lembrava que tinha.

- Antonio:

Começo agradecendo aos meus pais João e Márcia, que sempre estiveram comigo, me incentivaram, acreditaram e me deram todo o suporte para que eu pudesse ir atrás dos meus sonhos. Ao meu irmão João, por todos os conselhos e guiamentos que sempre pude ter em qualquer que seja a situação, sendo peça fundamental para eu me tornar a pessoa que sou hoje. Ao meu irmão Pedro, o qual eu sei que posso contar com sua ajuda, não importa qual seja a situação. Agradeço aos meus avós, Jonas e Valdete, que sempre vibraram e continuam a vibrar por cada conquista que realizo.

Agradeço à minha dupla deste projeto, Vanderson, por compartilhar todo esse árduo processo de formação em Engenharia Elétrica. Obrigado por toda a parceria nesse período de graduação, todas as madrugadas realizando trabalhos, todas as competições de robótica partilhadas e por saber que sempre posso contar com você, seja para os momentos bons ou ruins.

"Para que serve a utopia?
Serve para isso:
Para que eu não deixe de caminhar"

- Eduardo Galeano

Resumo

A utilização de sistemas autônomos está aumentando tanto em número de dispositivos quanto em número de aplicações. Esses sistemas, quando projetados com um propósito específico e capazes de realizar processamento de hardware, são classificados como sistemas embarcados. Nesse âmbito, a realização de testes para averiguar a execução, segurança e confiabilidade do sistema, de forma a garantir a exata integração entre o software implementado e o sistema físico, é de extrema importância. Para evitar a dependência da eletrônica do dispositivo, a implementação de um sistema simulado para validação do software é útil e necessária. Utilizando o QEMU para emulação de hardware e o Gazebo para simulação do ambiente real, este projeto busca desenvolver a possibilidade de testar e validar o software de um sistema embarcado sem a necessidade de qualquer componente eletrônico.

Palavras-chave: sistemas embarcados. robótica. QEMU. Gazebo. simulação.

Abstract

The usage of autonomous devices is growing in number of devices and applications. For having a specific purpose with hardware processing execution, it is classified as an embedded system. In this scope, the realization of all tests to determine the execution, safety and reliability of the system, to guarantee the exact integration between the physical environment and the implemented software, are of extreme importance. To avoid the dependency to the electronics, the implementation of a simulated system to validate the software is useful and necessary.With the using of QEMU for hardware emulation and Gazebo to simulate the environment, the project looks for a way to develop and have a full first trial test of an embedded system without needing the presence of any electronic component.

Keywords: embedded systems. robotics. QEMU. Gazebo. simulation.

List of Figures

Figure 1 – General concept 21
Figure 2 – Microcontroller basic structure 22
Figure 3 – GPIO addresses from microcontroller STM32F103C8T6 23
Figure 4 – Protocol Buffer workflow 24
Figure 5 – Publisher-Subscriber Pattern 25
Figure 6 – KVM Architecture 25
Figure 7 – Pioneer 2 26
Figure 8 – P2OS Connection 28
Figure 9 – ESP32 D1 Mini 38
Figure 10 – Environment model 44
Figure 11 – ROS Implementation within STM32 QEMU 45
Figure 12 – QEMU with ROS socket initial idea 45
Figure 13 – Emulation and Simulation complete integration 46
Figure 14 – General topics schema 47
Figure 15 – Callback memory access example 48
Figure 16 – Test Setup - General 49
Figure 17 – Test Setup - Hardware 50
Figure 18 – Test Setup - ESP32 Connections 51
Figure 19 – Real Robot With Prototype Hardware 52
Figure 20 – Pioneer 2DX - Batteries Informations 52
Figure 21 – Test Setup - Pioneer 2dx Interface 53
Figure 22 – Test Setup - Encoder Data as Bits Example 54
Figure 23 – Test Setup - PS5 Controller Commands 55
Figure 24 – Test Setup - P2OS Communication Code Organization - Simplified . . 57
Figure 25 – Pioneer 2 Interface PCB - 3D 71
Figure 26 – Pioneer 2 Interface PCB - Routes 71
Figure 27 – Pioneer 2DX Interface - P2OS Communication Code Organization . . . 73
Figure 28 – Pioneer 2DX Interface - Complete Code Organization 74

List of Tables

Table 1 – P2OS Internal Serial Port Connections ("HOST" JP8) 27
Table 2 – P2OS client command packet 29
Table 3 – Connection between Esp32 and Bluepill 50
Table 4 – Project main components 51

Contents

Contents **13**
1 Introduction **17**
1.1 Motivation 17
1.1.1 Micro-controllers and Embedded Systems 17
1.1.2 Simulations and Robots 18
1.1.3 Mobile Autonomous Robots 18
1.2 Objective 19
1.3 Justification 19
1.4 Project Organization 19
2 Conceptual Aspects **21**
2.1 Microcontroller Emulation 21
2.1.1 What is a Microcontroller 22
2.1.1.1 General Purpose Input/Output (GPIO) 23
2.2 Environment Simulation 23
2.2.1 Modeling the environment 23
2.2.2 Running the environment 24
2.3 Integration between the simulation and the emulation 24
2.3.1 Protocol Buffers 24
2.3.2 Publishers and Subscribers 24
2.4 Kernel Virtual Machine and Emulation 25
2.4.1 Kernel Virtual Machine 25
2.4.2 Emulation 26
2.5 Real Robot hardware and communication 26
2.5.1 Pioneer 2DX 26
2.5.1.1 Communication 27
2.6 P2OS - Communication protocol 28
2.6.0.1 P2OS - Commands Packet 29
3 Methodology **31**
3.1 Development Planning 31
3.1.1 Choose and emulate the micro-controller 31
3.1.2 Model the environment and the robot 31
3.1.3 Run the environment and the robot 31
3.1.4 Interconnect the emulation and simulation 31
3.1.5 Prototype an electronic interface between the micro-controller and the robot 32
3.1.6 Manufacture and test the electronic interface 32

3.1.7 Test development code with the real robot . . . 32
3.1.8 Compare the results with a real device . . . 32
4 Requirements Specifications . . . 33
4.0.1 Functional . . . 33
4.0.2 Non-Functional . . . 33
5 Project Development . . . 35
5.1 Tools . . . 35
5.1.1 Main program development . . . 35
5.1.1.1 C and C++ programming languages . . . 35
5.1.1.2 CubeMX . . . 35
5.1.1.3 PlatformIO . . . 35
5.1.2 Microcontroller Emulation . . . 36
5.1.2.1 QEMU . . . 36
5.1.2.2 ROS . . . 36
5.1.3 Simulation . . . 36
5.1.3.1 Gazebo . . . 36
5.1.3.2 Blender . . . 36
5.1.4 Hardware development . . . 37
5.1.4.1 Altium desing . . . 37
5.1.4.2 Multisim . . . 37
5.1.5 Hardware technologies . . . 37
5.1.5.1 Robot Pionner 2 . . . 37
5.1.5.2 Bluepill developement board . . . 37
5.1.5.3 Esp32 D1 Mini . . . 38
5.1.6 Others . . . 38
5.1.6.1 Max3232 . . . 38
5.1.6.2 LM317 . . . 38
5.2 Project and implementation . . . 39
5.2.1 Microcontroller emulation . . . 39
5.2.1.1 Choosing the hardware . . . 39
5.2.1.2 Emulating with QEMU . . . 39
5.2.2 ROS/Gazebo Simulation . . . 40
5.2.2.1 Robot model . . . 40
5.2.2.2 Peripherals plugins . . . 40
5.2.2.2.1 Digital distance sensor plugin . . . 40
5.2.2.2.2 Encoder plugin . . . 41
5.2.2.2.3 Joint motor plugin . . . 42
5.2.2.3 Environment model . . . 43
5.2.3 Integration . . . 43

5.2.3.1 Adding ROS to QEMU . . . 44
5.2.3.2 Complete integration . . . 46
5.3 Test Setup . . . 49
5.3.1 Hardware . . . 49
5.3.2 Real robot . . . 52
5.3.2.1 Batteries . . . 52
5.3.3 Pioneer 2DX Interface Firmware . . . 52
5.3.3.1 Bluepill Connection . . . 53
5.3.3.2 Ps5 Controller Connection . . . 54
5.3.3.3 Pioneer 2DX communication . . . 56
5.4 Results . . . 58
5.4.1 Motors command validation . . . 59
5.4.2 Distance sensors validation . . . 59
5.4.3 Encoders validation . . . 60
5.4.4 Dodge obstacles validation . . . 60
6 Final considerations . . . 61
6.1 Conclusion . . . 61
6.2 Contributions . . . 61
6.3 Prospects for Continuity . . . 62

Bibliography . . . 63
APPENDIX A Pioneer 2DX Interface Hardware . . . 67
A.1 Schematics . . . 67
A.2 PCB . . . 71
A.3 Components . . . 71
APPENDIX B Pioneer 2DX Interface Code Organization . . . 73

1 Introduction

With the exponential increase in memory and processing capacity of embedded devices, the complexity of algorithms and applications has significantly grown. However, as this complexity increases, so does the demand for testing. Unfortunately, the methods used to test these devices have not kept pace with this exponential growth.

In recent years, we have witnessed serious flaws in embedded software that have resulted in significant accidents, such as plane crashes involving Boeing aircraft [1] and pedestrian accidents caused by Uber autonomous vehicles [2]. These incidents highlight the fact that even market leaders in standalone embedded devices still suffer from firmware failures due to a lack of proper testing.

To address this problem, the most common method used is to utilize hardware prototyping boards, such as Arduino [3], which are becoming increasingly popular. However, only a small portion of professional micro-controller development at a lower level of abstraction can be considered valid and tested. This approach does not provide an efficient solution to the problem.

While not yet common, there are tools available today that allow for the simulation of micro-controllers without the need for physical hardware. However, these tools are quite specific and lack adequate integration with the external environment to simulate thermal events, for example. Conversely, there are excellent tools available for simulating the physical world and its numerous natural phenomena.

The biggest challenge lies in the lack of a tool that integrates these important simulation tools. If such tools were more readily available, we could explore a world of possibilities to innovate and create a new way of simulating embedded devices. These simulations would require testing and validation with stimuli from the physical world and natural phenomena to ensure they can be used in production without the risk of causing accidents.

1.1 Motivation

1.1.1 Micro-controllers and Embedded Systems

The usage of micro-controllers to develop applications for embedded systems is handled by a wide range of companies. In this scope, one of the most famous producer for those hardware is STMicroelectronics [4].

With a wide variety of devices and different families, the STM32 microcontrollers

from STMicroelectronics, or ST, are used in many applications of embedded systems. Examples of these applications include robot control, Internet of Things devices, and automation.

STM32 microcontrollers generally work with the ARM architecture and include a series of functionalities provided by the Hardware Abstraction Layer (HAL), which optimizes software development for these devices. Combined with its widespread market usability, this makes it easier and more convenient to implement the desired functionalities and utilities for embedded systems.

Despite their high presence in this environment, ST microcontrollers lack a way to test all implemented software without using the physical device itself, which poses a barrier to development.

1.1.2 Simulations and Robots

In the realm of modern robotics, Gazebo applications are quite common. Gazebo [5] is a widely used open-source, multi-robot simulation environment that provides a 3D simulated world where robots, objects like sensors, and entire environments can be modeled and simulated with native plugins.

In the hardware simulation scenario, the open-source software QEMU [6] stands out. As a Kernel-based Virtual Machine (KVM), it serves as a generic machine emulator, leveraging Linux's kernel to assist in the virtualization of different types of hardware. This usage allows for the emulation of microprocessor functionalities, enabling the simulation of embedded software execution within the emulated machine.

1.1.3 Mobile Autonomous Robots

With advances in technology and artificial intelligence, mobile autonomous robots are becoming increasingly common across various fields, from simple household tasks to space exploration. These robots can move independently without direct human intervention, enhancing efficiency and safety in many processes.

An example of a mobile autonomous robot application is the 'robô hospitalar', developed by the Escola Politécnica da Universidade de São Paulo. It assists in delivering medicines and necessary tools to hospital staff, improving logistics, efficiency, and people management. This field holds significant interest in robotics and the development of solutions for the benefit of humanity.

1.2 Objective

The main goal of this work is to develop a simulator for software testing of embedded systems using STMicroelectronics microcontrollers applied to mobile robotics, enabling comparisons with real projects. The primary tools used will be Gazebo for simulating the environment and data acquisition, and QEMU for hardware emulation.

Using the simulator, it is expected to enable code validation before bench testing with physical devices.

1.3 Justification

Nowadays, there are not so many options to simulate the functionality of an embedded system. Most of them are useful to test hardware features, one example is to use circuit emulators like Proteus Design Suite. Although it is possible to simulate some micro-controllers with real source code, it is limited to the elements and devices of the software itself, not being able to stipulate the data flow with a real device.

Other option is the use of Hardware-in-the-loop, which is a validation technique used to create a realistic testing environment. It is a good way to test complex systems applied to robotics, giving a safety and risk mitigation before the deploy of the application. On the other hand, although it is not necessary to have the complete physical setup, to use this technique it is still necessary to have hardware components to connect to the simulation environment. Therefore, it is impossible to test only the software that would be used itself.

Accordingly to this scope, in the current moment it is not possible to test the software for the whole embedded system with all expected functionalities without a physical device. This brings a barrier to the development of an application as it brings the dependency of having a prototype and certifying that it is working correctly.

In this way, the implementation of this project will provide another possibility for embedded systems software testing, enabling code validation before bench testing through a complete simulation of both the processor and connected devices. As a result, it is expected to achieve: reduced development costs, increased safety and reliability in tests, and fewer evaluations on real-world equipment.

1.4 Project Organization

The chapter 2 introduces the main concepts utilized in this Bachelor thesis. Subsequently, chapter 3 presents the methodology applied during the development of this work, while chapter 4 lists the functional and non-functional requirements of the project.

Following this, chapter 5 discusses the tools used, provides descriptions, and explains how the entire simulation and test setup was implemented. Additionally, appendix A contains information on the hardware developed for the project, and appendix B provides details on the structure of the Pioneer 2DX interface code.

2 Conceptual Aspects

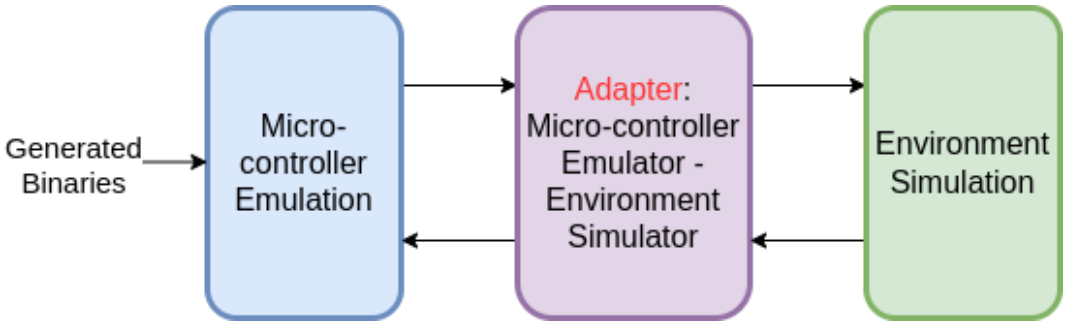
Dealing with a complete simulation of an embedded system there are three main problems: microcontroller emulation, environment simulation and the interconnection between both.

Microcontroller Emulation: The main aspects of the microcontroller must be emulated, including: microprocessing, General Purpose Input Output, RAM memory, flash memory and others peripherals present in the device.

Environment Simulation: The embedded device is always immersed in an environment, therefore it's simulation must be able to replicate a variety of natural physical phenomena, in addition to being easily manipulated and adaptable.

Integration: The integration between them should capture the phenomena obtained from the environment simulation and send the data to the microcontroller emulator as if they were electrical signals it would receive in reality. Conversely, the integration must also capture the values calculated by the emulation and send them back to the virtual environment. This cycle should occur continuously.

Figure 1 – General concept



In this undergraduate final project, those three main problems will be addressed. To achieve this, industry-standard tools will be used to implement microcontroller emulation, environment simulation, and design. Additionally, there is currently no available tool capable of effectively connecting these features.

2.1 Microcontroller Emulation

Embedded devices are designed primarily to execute specific functions or to address particular applications, with inherent data processing capabilities. Microcontrollers are widely used to achieve this goal. However, it's important to have a clear understanding of their concepts and functioning due to the complexity and diversity of these chips.

2.1.1 What is a Microcontroller

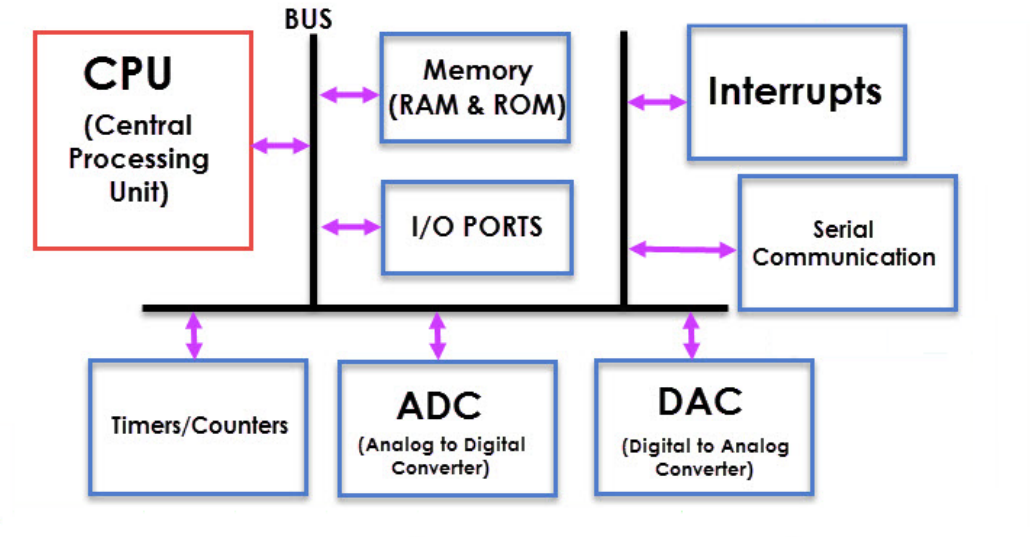
A microcontroller is a programmable embedded electronic device that integrates a processor, memory, and peripherals into a single integrated circuit. These devices have specific architectures that vary depending on the manufacturer and include a range of peripherals designed to perform specific functions.

Microcontrollers typically feature 8-bit, 16-bit, or 32-bit architectures. Each architecture organizes information in memory differently, which affects aspects such as storage capacity and address management, among other factors.

The microcontroller's peripherals are responsible for communication with external environment. They play a role in receiving sensor information, managing communication interfaces, and sending signals, which translates into actions in the external world.

Initially, the peripherals receive and send a series of electrical signals originating from various sensors and actuators, which access the internal circuits of the microcontroller. These signals modify the microcontroller's memory. When the microcontroller reads the information stored in memory, it detects any changes and, if needed, adjusts its output behavior accordingly.

Figure 2 – Microcontroller basic structure



Source: ElectronicsHub [7]

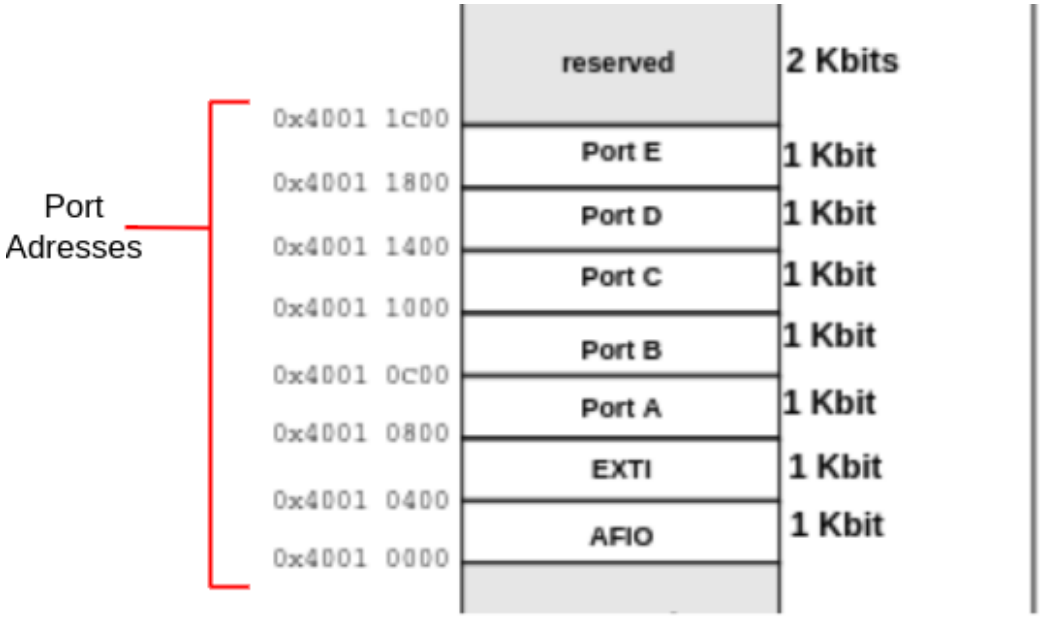
In the context of simulation, it is essential to replicate these external world functionalities, which are the responsibilities of the peripherals. Due to the specific way each peripheral accesses reserved positions in the microcontroller's memory and organizes information, it is important to explain in greater detail how this process is carried out for the peripherals that are the focus of emulation in this project.

2.1.1.1 General Purpose Input/Output (GPIO)

The GPIOs (General Purpose Input/Output) are fundamental peripherals in microcontrollers, responsible for handling information in a binary form, represented as 0 or 1. In the context of microcontrollers, it is necessary to specify which PORT and PIN will be used to configure a particular GPIO. This selection ensures that the appropriate pin is allocated within the circuit in which the microcontroller is integrated.

GPIOs function as MMOI (Memory Mapped Input/Output), which means they map a specific memory location to store information related to the PORT and PIN. Each port corresponds to a specific memory address, while each pin acts as a mask that indicates the memory position where the information is being stored.

Figure 3 – GPIO addresses from microcontroller STM32F103C8T6



Fonte: STM32F103C8T6 Datasheet [8]

Additionally, the GPIO configuration—such as whether it will be used as an input or output or for other functions—also influences the memory location that will be accessed to retrieve the necessary information.

2.2 Environment Simulation

2.2.1 Modeling the environment

The scope of modeling refers to how the simulation will be visualized. The two main objects that need to be modeled are the room and the embedded system itself. To achieve this, there is a variety of open-source software that provides modeling tools and can export the created objects to the simulation environment.

2.2.2 Running the environment

The running environment integrates the modeled objects, including all their physical characteristics and interactions. It will also be responsible for exporting and receiving data to and from an external source.

2.3 Integration between the simulation and the emulation

Nowadays there are many ways to make the interconnection and data exchange between between two different software.

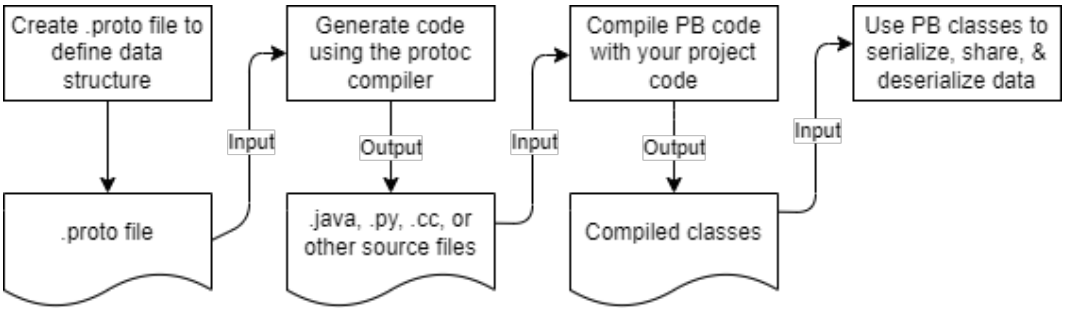
For this project, it is necessary a non-stop data exchange of both sides between the emulation and the simulation. To accomplish that, it is interesting to use used communication protocols that is shown as it follows:

2.3.1 Protocol Buffers

Protocol buffers [9], or protobufs, are mechanism developed by Google used to serialize different kind of data. Using a language and platform neutral schema definition, protobufs are really useful to parse and serialize data between software that uses different types of languages and formats.

To define how the data is structured, it is used the `.proto` files. These files needs to be compiled by the proto compiler to generate the source code structuring the data format in different types of languages. In that way, the same `.proto` file can used by different software.

Figure 4 – Protocol Buffer workflow



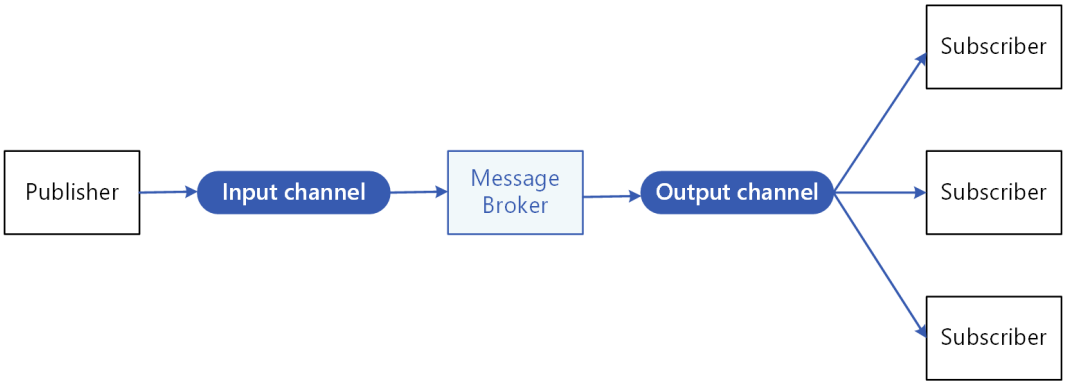
An important feature of this protocol is how its messages are compact. Using a compact binary format reduces the network load on the application.

2.3.2 Publishers and Subscribers

This messaging pattern [10] is used to facilitate communication between different parts of a system or different software components. This pattern is based on what is known

as a "topic," which acts as a channel that allows two possible actions: publishing and subscribing.

Figure 5 – Publisher-Subscriber Pattern



Publishing means sending a message to a topic, making that message visible to anyone with reading access to the topic. It allows data to be shared via the topic.

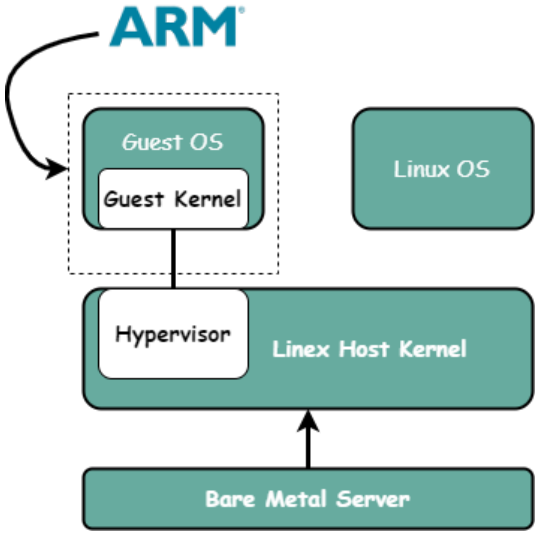
On the other hand, subscribing means 'listening' to what is published on a topic. When a topic is published, the subscriber receives a callback notifying them of the message's arrival and retrieving the data.

2.4 Kernel Virtual Machine and Emulation

2.4.1 Kernel Virtual Machine

System virtualization is the process of hosting one operating system (the guest system) within another (the host system). Through virtualization, the guest system can replicate its functionalities by leveraging the host system's processing power. This allows the host to run functionalities exclusive to the guest system.

Figure 6 – KVM Architecture



Virtualization is made possible by the use of a hypervisor, a type of operating system-level software responsible for managing the host system's computing resources and allocating them to hosted virtual machines.

In this context, the Kernel Virtual Machine (KVM) [11] is a technology used for virtualization on Linux systems. It transforms the Linux kernel into a hypervisor, enabling part of

the host hardware to be directly utilized for guest system functionalities. For instance, KVM maps the virtualized guest memory directly to the host memory and allows the host processor to function as a virtualized CPU.

This approach enables the guest system to run nearly as if it were native, significantly improving performance, security, and reliability.

For our project, KVM is particularly useful for emulating all the specific components of a microcontroller (such as an ARM microcontroller [12]), including the processor, memory, peripherals, and more.

2.4.2 Emulation

In computing, emulation refers to the process of replicating the behavior of one computer system or software on another system.

In this context, hardware emulation involves replicating the functionalities of hardware components such as memory management, I/O operations, CPU processing, and interrupt handling within another system. Unlike simulation, which models how a system behaves in theory, emulation strives to replicate the original system's actual operation. This enables the emulated system to use the same configurations, load identical software, and adhere to the same standards as the original hardware.

An ideal hardware emulation accurately reproduces every functionality of the original system, maintaining full compatibility.

2.5 Real Robot hardware and communication

2.5.1 Pioneer 2DX

The Pioneer 2 is a programmable mobile robot platform widely used in research and educational settings for the development and testing of robotics algorithms and systems worldwide. The Pioneer 2 supports a variety of sensors and actuators, such as motors, ultrasonic sensors, encoders, cameras, and more. This enables it to perform tasks like navigation, mapping, and object manipulation in diverse environments.

Figure 7 – Pioneer 2

Designed for research and education, the Pioneer 2 serves as a tool for exploring conceptual aspects of robotics, including kinematics, path planning, and sensor fusion.

As a commercial robot, there is an operations manual available for users [13], which includes all hardware specifications and detailed instructions on how to use the robot correctly.

2.5.1.1 Communication

For communication with the robot, there is a Serial RS232 port available. The pins for this serial port are listed in Table 1. The TX and RX pins allow external devices to communicate with the Pioneer 2DX through the P2OS protocol, which is further described in Section 2.6.

Table 1 – P2OS Internal Serial Port Connections ("HOST" JP8)

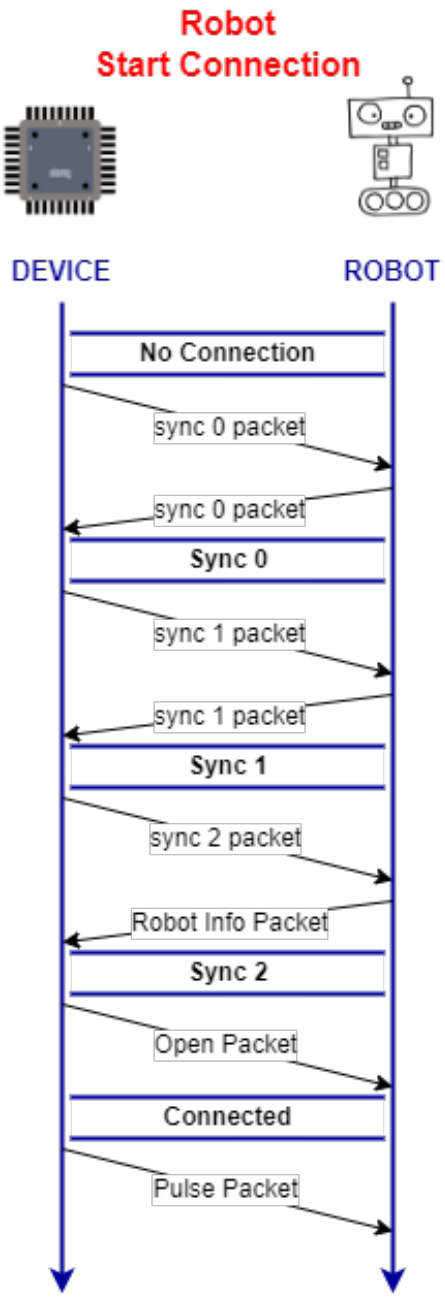
Pin #	**Connection**	**Pin #**	**Connection**
1	Gnd	2	P3_12
3	TxD1	4	12 VDC (switched)
5	RxD1	6	Gnd
7	P3_15	8	P3_14
9	Gnd	10	5 VDC (switched)

Source: Pioneer2 Operations Manual [13]

2.6 P2OS - Communication protocol

The P2OS (Pioneer 2 Operating System) protocol is used to establish a connection between the Pioneer 2DX robot and your device. This protocol enables you to send commands to the actuators and receive data from the sensors. Additionally, it also allows you to modify the robot's configuration through the protocol.

Figure 8 – P2OS Connection



At first, the client device must first establish a connection with the robot server. After it, the client can send commands to the server and receive information from the robot back. In the case of the Pioneer 2, the communication is done through the Host RS-232 serial port.

Initially, P2OS starts in an initial state **NO-CONN**. To establish this connection, the client application must send three synchronization packets consecutively to the robot: the SYNC0, SYNC1, and SYNC2 commands packets. For each command, the device must receive the corresponding responses from the robot. After the SYNC0 command, the robot transitions to the **SYNC0** state; after SYNC1, it moves to the **SYNC1** state; and after SYNC2, it enters the **SYNC2** state.

The feedback from the robot before reaching the **SYNC2** state is a special package containing information about the robot. This package includes the name, type, and subtype of the currently connected robot.

After reaching the **SYNC2** state, it is possible to open the connection using the Open command packet. Once the open command is sent, the robot transitions to the **CONNECTED** state.

After the connection is established, the robot begins sending information about all available sensors and receives data regarding the actuators. To maintain the connection with the robot, it is necessary to send a pulse command at least every 2 seconds.

2.6.0.1 P2OS - Commands Packet

The P2OS protocol has a structured command format for receiving and responding to instructions from a device. It is possible to observe the whole command packet structure in the table 2

Table 2 – P2OS client command packet

Component	**Bytes**	**Value**	**Description**
Header	2	0xFA, 0xFB	Packet header; same for client and server
Byte Count	1	N + 2	Number of following command bytes plus Checksum's two bytes, but not including Byte Count. Maximum of 200.
Command Number	1	0 - 255	Client command number; see Table 4-4
Argument Type (depends on the command)	1	0x3B or 0x1B or 0x2B	Required data type of command argument: positive integer (sfARGINT), negative integer or absolute value (sfARGINT), string (sfARGSTR)
Argument (depends on the command)	n	data	Command argument; integer or string
Checksum	2	computed	Packet integrity checksum

Source: Pioneer2 Operations Manual [13]

Following that structure, if it is necessary to send a command "0" (which corresponds to the SYNC0 command), the complete packet will be "0xFA 0xFB 0x03 0x00 0x00 0x00". If it is necessary to send a command "1" (which corresponds to the SYNC1 command), the complete packet will be "0xFA 0xFB 0x03 0x01 0x00 0x01".

3 Methodology

3.1 Development Planning

Using the concepts mentioned on the previous chapter, this project has as a goal the building an embedded system simulator. To achieve this objective, the project follows the methodology composed by the following steps:

3.1.1 Choose and emulate the micro-controller

The first step was chose the micro-controller that will be used. Each kind of micro-controller has it own specificities and characteristics, and it is essential to chose one with the demanded features by the simulated device. After this, using an open source framework, it is fundamental to emulate all the micro-controller essentials functionalities.

3.1.2 Model the environment and the robot

After emulating the micro-controller that will be used, the whole simulated environment and the robot will be made and tested. This step is important to determine how the environment will be and which physical properties it will have. It is important that this environment and robot to be able to be exported to others software and keep the same properties.

3.1.3 Run the environment and the robot

With the modelled environment in hands, it is time to run it. In this stage, it will test if the simulated robot and the environment are working as they should. At this part, the topic structure will be defined, and will be tested if the simulated robot is publishing and subscribing to them.

3.1.4 Interconnect the emulation and simulation

With both the emulation and the simulation made, it is time to connect them. To do this, all the concepts of interconnection mentioned on the last chapter will be used. Using TCP/IP sockets, the emulated hardware and the simulated environment will share topics to published and subscribed from both sides.

3.1.5 Prototype an electronic interface between the micro-controller and the robot

After choosing the microcontroller and selecting which pin from the microcontroller to use, it is essential to create a simple electronic case to gather all the electronic signal from the microcontroller and send them correctly to the robot.

3.1.6 Manufacture and test the electronic interface

After creating the electronic case, it is essential to find a manufacture, send them the project, buy the components and assemble the case. After that, it is important to test everything and check whether it is all working as it should.

3.1.7 Test development code with the real robot

After the electronic case is working, it is time to upload the code to the microcontroller and run the program, which was created with the help of the simulation, on the real robot.

3.1.8 Compare the results with a real device

After all the project implementation, it is time to compare the results of the simulation with how the real device works. At this stage, it will see how good the simulation approached reality and in which points the project should be improved.

4 Requirements Specifications

This chapter presents the, functional and non-functional, requirements specifications of the project to be developed.

4.0.1 Functional

Micro-controller Emulation: The project must be able to run the binary of an embedded system software the same way it would run in a real hardware. Therefore, the emulation of the functionalities and peripherals of the chosen micro-controller is essential.

Environment and Robot simulation: To reproduce how the physical system would respond in reality, it is necessary to have a whole simulated environment where the simulated robot will actuate.

Connection Between the Emulation and the Simulation: Since it is the simulation of a entire embedded system, it is necessary the connection between the hardware emulation and the environment simulation.

Embedded System software binary: To test and validate the project, it is necessary to have a functional software binary to be run by the hardware emulator. **Physical embedded system**: To validate the project, it is essential to have a physical device. With that, it will be possible to see how close from a real system the simulation will get.

4.0.2 Non-Functional

Simulation Performance: It is important that the simulation has a good performance, reducing it's delay as maximum as possible.

5 Project Development

5.1 Tools

5.1.1 Main program development

5.1.1.1 C and C++ programming languages

C and C++ are essential programming languages in embedded systems development due to their efficiency and control over hardware resources.

C was used not only for developing the firmware for the STM32 microcontroller (BluePill) but also for the hardware emulation. Its low-level capabilities allow precise control over microcontroller peripherals and were crucial for simulating the hardware's behavior.

C++ extends C with object-oriented features, making it ideal for more complex system architectures. In this project, C++ was utilized in the robotic simulation, especially within the ROS environment. Its object-oriented approach helped manage the complexity of the system, allowing better modularity and code organization, especially when dealing with ROS nodes and topics.

Thus, C was key in both the firmware development and hardware emulation, while C++ was primarily used for the simulation environment, leveraging its additional features for managing higher-level structures.

5.1.1.2 CubeMX

STM32CubeMX is a microcontroller configuration tool provided by STMicroelectronics. It was used for the initial configuration of the STM32F103C8T6 microcontroller, including generating the base project code and setting up peripherals such as UART, SPI, and GPIO. The tool simplified pin mapping and system parameter configuration, facilitating the integration between hardware and software.

5.1.1.3 PlatformIO

PlatformIO [14] is an open-source IDE (integrated development environment). It simplifies the process of building, debugging, and deploying code to microcontrollers and other embedded systems.

5.1.2 Microcontroller Emulation

5.1.2.1 QEMU

QEMU [6] was used as the hardware emulator for the STM32F103C8T6 microcontroller. The specialized version for STM32 microcontrollers allowed for precise simulation of the microcontroller's functionalities, particularly in emulating GPIO interfaces, which are essential for controlling external devices. QEMU enabled testing the firmware behavior without the need for physical hardware.

5.1.2.2 ROS

ROS [15] was used as middleware for communication between the microcontroller emulation and the robotic environment simulation. Through ROS topics, the emulated GPIO input and output signals of the STM32 were sent and received, allowing the emulated microcontroller to interact seamlessly with the simulation environment.

5.1.3 Simulation

5.1.3.1 Gazebo

Gazebo was the main tool used for simulating the robotic environment. It allowed for testing the interactions of the microcontroller with the simulation in a visual and interactive manner, particularly in controlling actuators and reading sensor data. The integration between Gazebo, ROS, and the emulated microcontroller enabled a more realistic and efficient simulation of the overall robotic system.

5.1.3.2 Blender

Blender is an open-source 3D modeling and animation software widely used in various fields, including game development, visual effects, and architectural visualization. It provides a comprehensive suite of tools for modeling, sculpting, texturing, and rendering, making it a versatile choice for creating detailed 3D environments.

In this project, Blender was utilized to model the test environment for the robotic simulation. The detailed 3D models created in Blender were essential for accurately representing the physical environment in which the robot would operate. Once the modeling was completed, the environment was exported in a compatible format for integration with Gazebo, allowing for realistic simulation and interaction with the robot in the virtual world.

5.1.4 Hardware development

5.1.4.1 Altium desing

Altium Designer is a professional PCB (Printed Circuit Board) design software that was used to design a custom circuit board for the project. The PCB was designed to adapt the system for use in the real robot. Altium Designer offers a complete set of tools for schematic capture, PCB layout, and component management, making it ideal for creating custom hardware solutions.

5.1.4.2 Multisim

Multisim is an industry-standard software used for simulating electronic circuits. It allows for the creation and testing of circuit designs in a virtual environment, offering a wide range of tools for analyzing circuit behavior under different conditions. In this project, Multisim was utilized for initial testing of the hardware that was designed using Altium Designer. Before physically assembling the printed circuit board (PCB), simulations in Multisim helped validate the planned design, ensuring that key components and circuit configurations would function as expected when implemented in the real-world system.

5.1.5 Hardware technologies

5.1.5.1 Robot Pionner 2

The Pioneer 2 is a mobile robot platform developed by Adept MobileRobots (formerly ActivMedia Robotics). It is widely used in academic research and robotics education due to its robustness and flexibility in various applications such as navigation, mapping, and autonomous robotics. The Pioneer 2 is equipped with multiple sensors and interfaces, making it ideal for testing algorithms and embedded systems in real-world scenarios.

In this project, the Pioneer 2 was chosen as the physical robot for system validation. It was made available by POLI for use as the real robot platform.

5.1.5.2 Bluepill developement board

The Blue Pill is a small, low-cost development board featuring the STM32F103C8T6 microcontroller, part of the STM32 family from STMicroelectronics. The board is highly popular in embedded systems projects due to its affordability, versatility, and relatively powerful ARM Cortex-M3 core. It comes equipped with 64 KB of flash memory, 20 KB of RAM, and operates at a frequency of 72 MHz, providing sufficient performance for a wide range of applications.

5.1.5.3 Esp32 D1 Mini

The ESP32 [16] D1 Mini is a development board built around the ESP32 microcontroller from Espressif Systems. The board features dual-core processing with a 32-bit Tensilica LX6 microprocessor running at up to 240 MHz. It comes equipped with 4 MB of flash memory and 520 KB of SRAM, supporting some connectivity options, like Wi-Fi, classic Bluetooth, and BLE Bluetooth. These capabilities make the ESP32 D1 Mini an ideal choice for IoT applications, sensor networks, and embedded projects in general.

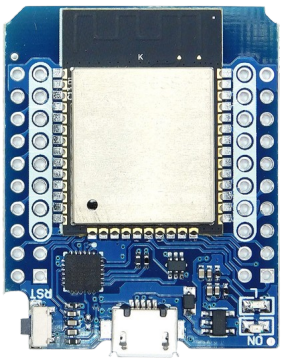


Figure 9 – ESP32 D1 Mini

5.1.6 Others

5.1.6.1 Max3232

There are some components on the market capable of transforming a TTL serial signal into RS232. Among the most reliable components is the MAX232, which is well known for this signal translation.

5.1.6.2 LM317

There are some components on the market capable of control voltage with high precision and efficiency. Among the most reliable components is the LM317, which is well known for its adjustable voltage regulation capabilities and versatility in power supply designs.

5.2 Project and implementation

5.2.1 Microcontroller emulation

5.2.1.1 Choosing the hardware

For the system development, a microcontroller from the STM32 family, manufactured by STMicroelectronics [4], was chosen. The decision to use this line of microcontrollers was driven by several factors, including their versatility, a wide range of integrated features, and strong support provided by both the community and STMicroelectronics itself. Additionally, the STM32 family is well-known for offering development tools, such as STM32CubeMX [17], which simplifies hardware configuration and code generation, making the development process more efficient and accessible.

Among the available models, the Blue Pill [18] development board was selected, which integrates the STM32F103C8T6 microcontroller [8]. This choice was guided by its low cost, versatility and previous familiarity.

5.2.1.2 Emulating with QEMU

After selecting the STM32 microcontroller, the QEMU hardware emulator was used to reproduce the microcontroller's operation. A specialized version of QEMU [19], tailored for STM32 microcontroller emulation, was employed to replicate the functionality of the STM32F103C8T6.

The primary focus of the emulation was on simulating the GPIO (General-Purpose Input/Output) functions, which is used for interfacing with the sensors and actuators in the robotic system. Initially, efforts were made to emulate other functionalities, such as Pulse Width Modulation (PWM) generation and timers. However, these features did not function as expected during testing.

Despite these challenges, the emulator allowed for the first round of binary testing, enabling verification of GPIO pin value changes, the relationship between memory positions, and overall emulator functionality.

To streamline the execution of QEMU for hardware emulation, a custom bash script, run.sh, was developed. This script gathered parameters to be able to run the emulation such as:

- **Link to the embedded software binary**: The used software to be executes by the emulator;
- **Machine reference**: This version of QEMU has the capability to reproduce different kinds of machines, so it is necessary to specify which one is going be emulated;

- **TCP/IP Socket**: Initialize and configure a TCP/IP socket.

5.2.2 ROS/Gazebo Simulation

The development of the simulation is structured around three main components: the model of the Pioneer 2 robot [13], the peripheral plugins (sensors, encoders, and motors) and the implementation of the environment where the simulated robot will be put. Each of these components plays a distinct role in the overall simulation setup.

5.2.2.1 Robot model

For this part, a search was conducted in the developer community for potential CAD models [20] that could be exported to Gazebo [5] and used as the project's robot model. Although an exact version of the Pioneer 2 robot was not found, a model of the Pioneer 3 robot was available. Since the differences between these models are primarily related to hardware, the Pioneer 3 [21] CAD model was deemed suitable for use.

This model [22] encapsulates all the necessary information about the robot, including its physics (such as weight, momentum, dimensions, and degrees of freedom), mechanical details (such as its wheels and chassis), and overall design.

5.2.2.2 Peripherals plugins

The peripheral plugins were responsible for simulating the operation of the robot's sensors and actuators. Through the use of ROS Topics [23] for communication with the emulation, each plugin acted as an intermediary, either publishing or subscribing to its respective node.

For sensors, a distance sensor plugin and an encoder plugin were developed and utilized. For actuators, the joint_motor ROS actuator was employed. Each of these plugins will be explained in detail in the following sections.

5.2.2.2.1 Digital distance sensor plugin

The digital distance sensor plugin was created to replicate the behavior of a basic digital proximity sensor. This sensor operates by detecting nearby objects and providing a binary output based on the presence of an object. It outputs a signal of 0 (low) when no object is within range and a signal of 1 (high) when an object is detected. This simple binary response enables effective object detection and basic environmental awareness, making it a useful tool in applications where obstacle detection is crucial.

The implementation was based on the standard ROS ray sensor [24]. To simulate the operation of a digital distance sensor, a visibility threshold was set to determine when

an object is considered detected or not, outputting either 1 or 0. The sensor then publishes this output to a ROS topic, which sends the information to QEMU's emulation.

Listing 5.1 – Digital distance sensor simulation with ray plugin

```
void DigitalDistanceSensor::OnNewLaserScans() {
    this->range_to_pub = this->range_msg;

    std_msgs::Int32 digital_value;
    if (float(this->range_msg.range) < float(this->threshold)
       ) {
        range_to_pub.range = 1;
        digital_value.data = 1;
    } else {
        range_to_pub.range = 0;
        digital_value.data = 0;
    }

    this->pub.publish(digital_value);

    if (this->topic_name != "") {
        common::Time cur_time = this->world->SimTime();

        if (cur_time - this->last_update_time >= this->
           update_period) {
            this->UpdateRangeData();
            this->last_update_time = cur_time;
        }
    } else {
        gzthrow("digital distance sensor topic name not set")
           ;
    }
}
```

5.2.2.2.2 Encoder plugin

The encoder plugin was designed to simulate an absolute encoder. For this simulation, a 3-bit resolution was chosen, which allows for 8 distinct positions over a full 360-degree rotation. Unlike incremental encoders, which only track movement by counting pulses, an absolute encoder provides a unique position value for each angle. This ensures

that the system always knows the exact angular position, regardless of any previous movements or resets.

To implement this, the current position of the wheel joint element, which is responsible for rotation, was used. The angle position of the wheel joint is then assigned to one of the eight discrete values. For instance, if the wheel joint's position is between 0 and 1/8 of 360 degrees, the output would be 0, and so on. The output is then published to a ROS topic to provide the updated position to the system.

Listing 5.2 – Encoder plugin position update function

```
void EncoderPlugin::OnUpdate() {
        float wheel_angle = this->wheel_joint->Position(0);

        int encoder_value = static_cast<int32_t>(fmod(
           wheel_angle,
        2 * M_PI) / (M_PI / 4));
        encoder_value = encoder_value % 8;

        this->encoder_msg.data = encoder_value;
        this->encoder_pub.publish(this->encoder_msg);
    }
```

5.2.2.2.3 Joint motor plugin

The Joint Motor plugin [25], also sourced from the developer community, was used to represent the robot's motors. This plugin was adapted to manage the motor commands with a 3-bit resolution, allowing control over the robot's speed.

To implement this, the plugin subscribes to a command node that is published by the emulation, receiving speed commands to control the robot's movement. Once the command is received, the plugin processes the speed value and publishes it to the nodes responsible for controlling the motors. These nodes then drive the motors accordingly, ensuring the robot's movements align with the speed commands provided by the emulation.

Listing 5.3 – Joint motor update function

```
void JointMotor::UpdateChild() {
    joint_->SetParam("fmax", 0, ode_joint_motor_fmax_);
    joint_ -> SetParam("vel", 0, input_); // input_ is
       updated
                                          // by node
                                             subscription
```

```
5
6       common::Time current_time = parent->GetWorld()->SimTime()
            ;
7       double seconds_since_last_update =
8           ( current_time - last_update_time_ ).Double();
9       double current_speed =
10          joint_->GetVelocity( 0u )*encoder_to_shaft_ratio_;
11      ignition::math::Vector3d current_torque =
12          this->link_->RelativeTorque();
13      if ( seconds_since_last_update > update_period_ ) {
14          publishWheelJointState( current_speed, current_torque
                .Z() );
15          publishRotorVelocity( current_speed );
16          last_update_time_+= common::Time ( update_period_ );
17      }
18  }
```

5.2.2.3 Environment model

To create a realistic testing environment, the third-floor ramp of the Electrical Engineering building at the Polytechnic School of the University of São Paulo was selected as the basis for replication. This location was chosen not only because of our relation to the building but also because it offers an ideal environment for testing the robot's performance.

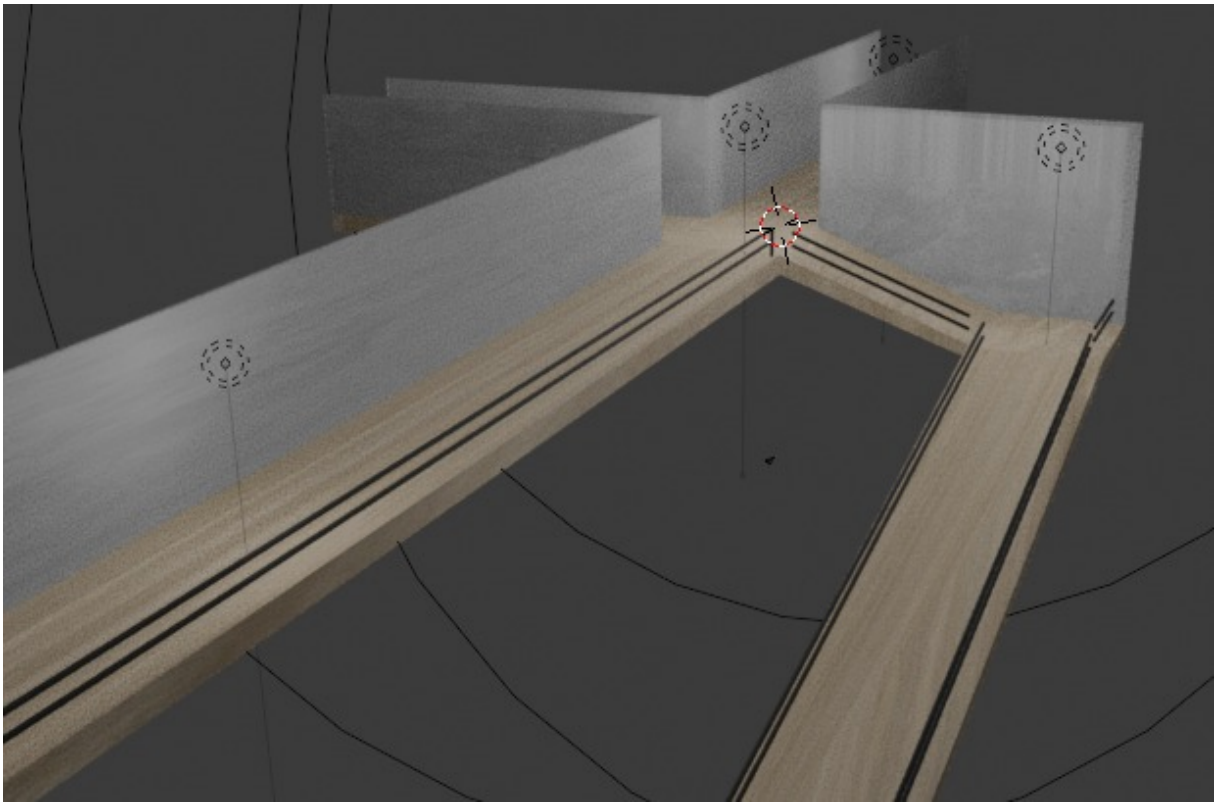
The modeling of this environment 10 was done using Blender, which provided the tools necessary to create a detailed representation of the physical structure. After the environment was modeled, it was exported for use in Gazebo, ensuring an accurate and functional setting for testing and validating the robot within the simulation.

5.2.3 Integration

One of the primary challenges of this project was establishing effective integration between hardware emulation and the Gazebo simulation environment. To achieve this, ROS topics were chosen as the communication medium between the two systems, a decision driven by their modularity and compatibility with the Gazebo simulation.

On the other hand, there was no inherent support for ROS on the QEMU side, which required efforts to overcome this compatibility barrier.

Figure 10 – Environment model



5.2.3.1 Adding ROS to QEMU

One of the initial tasks in integrating ROS with QEMU was the difference in their native programming languages: ROS libraries are primarily written in C++ or Python, whereas QEMU is implemented in C.

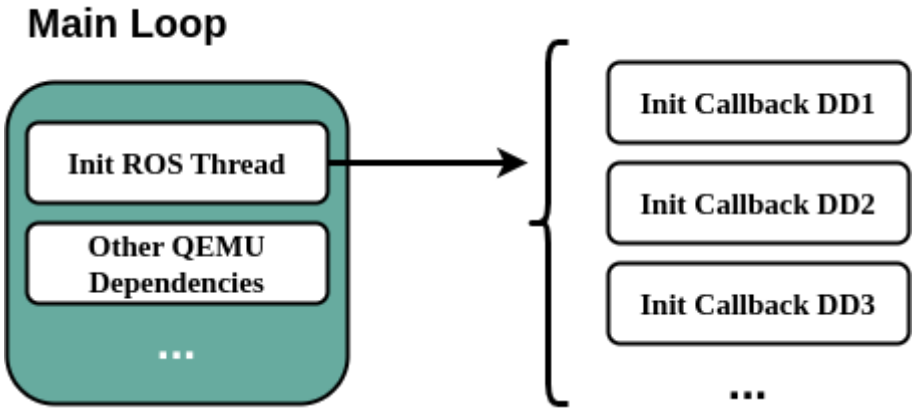
To address this, it was necessary to find a C-based library capable of implementing a ROS node. After conducting research, the cROS library [26] was identified as a suitable solution. This library provided the functionality needed to create ROS nodes in a C environment. However, integrating this library into QEMU's codebase presented an additional challenge.

To integrate the ROS library in C, all the library files were added to the repository and imported into the main QEMU file. This file, `vl.c`, contains the implementation of QEMU's main loop. That was made by including the header file `qom/cros.h`, which provides the implementation of the library and handles all the necessary internal includes. For more information about how the cROS works, it is possible to check the official manual [27] or the source code [26].

The integration was achieved by introducing a dedicated thread within QEMU's main loop. This thread was responsible for running the ROS main loop, which linked to a TCP/IP socket to stablish communication with other nodes. Each node was assigned a

callback function, responsible for publishing or subscribing to a topic, enabling interaction between QEMU and the broader ROS ecosystem. In the figure 11, it can be visualized.

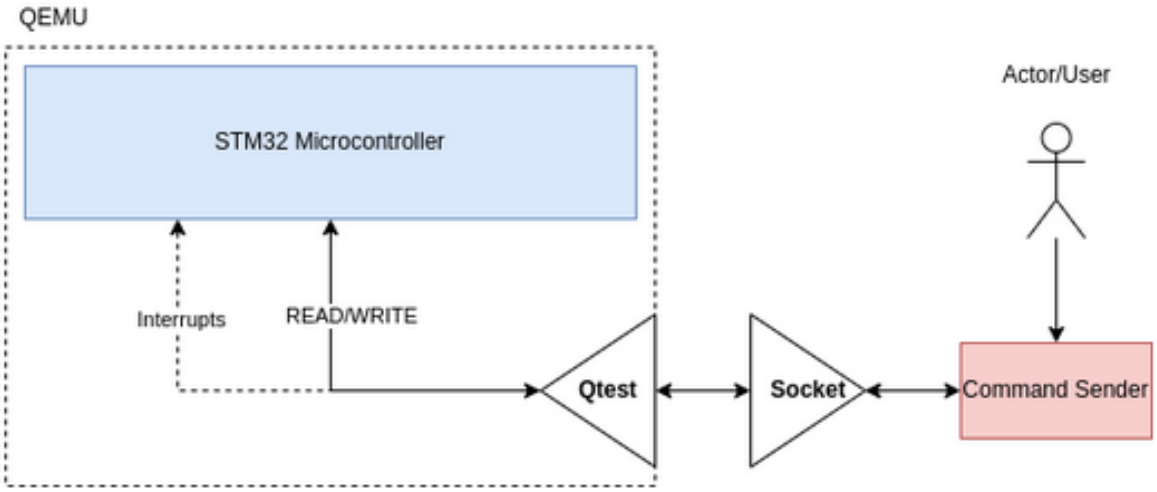
Figure 11 – ROS Implementation within STM32 QEMU



Each callback from a sensor is responsible for reading a value from a ROS topic and writing it in the correct place in memory. For the topic from the motors, the value is read from the memory and it is written in the ROS topics. There is more detailed information about it in the topic 5.2.3.2

The TCP/IP socket is important for making the correct link with the simulation because it connects the ROS provider from outside the QEMU with the ROS provider from inside QEMU. The initial idea of the TCP/IP socket can be visualized in the figure 12

Figure 12 – QEMU with ROS socket initial idea

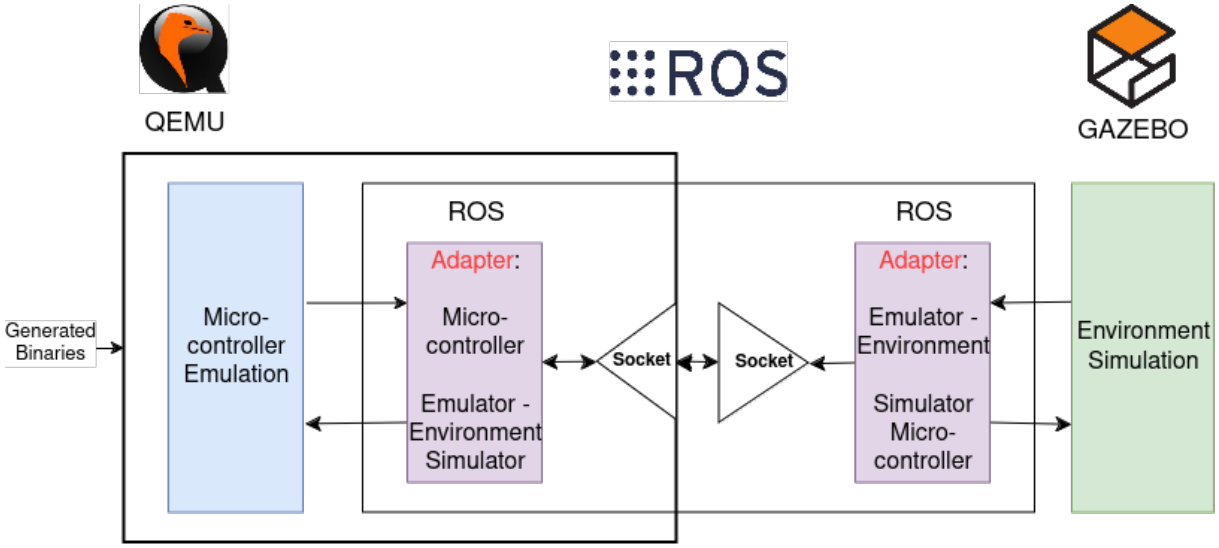


In addition to the ROS main loop, it was necessary to add to the `run.sh` script, the TCP/IP socket configuration. With that, this script is able to compile the stm32_qemu

code, initialize the socket, select the binary code from the bluepill folder and execute the whole simulation.

5.2.3.2 Complete integration

Figure 13 – Emulation and Simulation complete integration



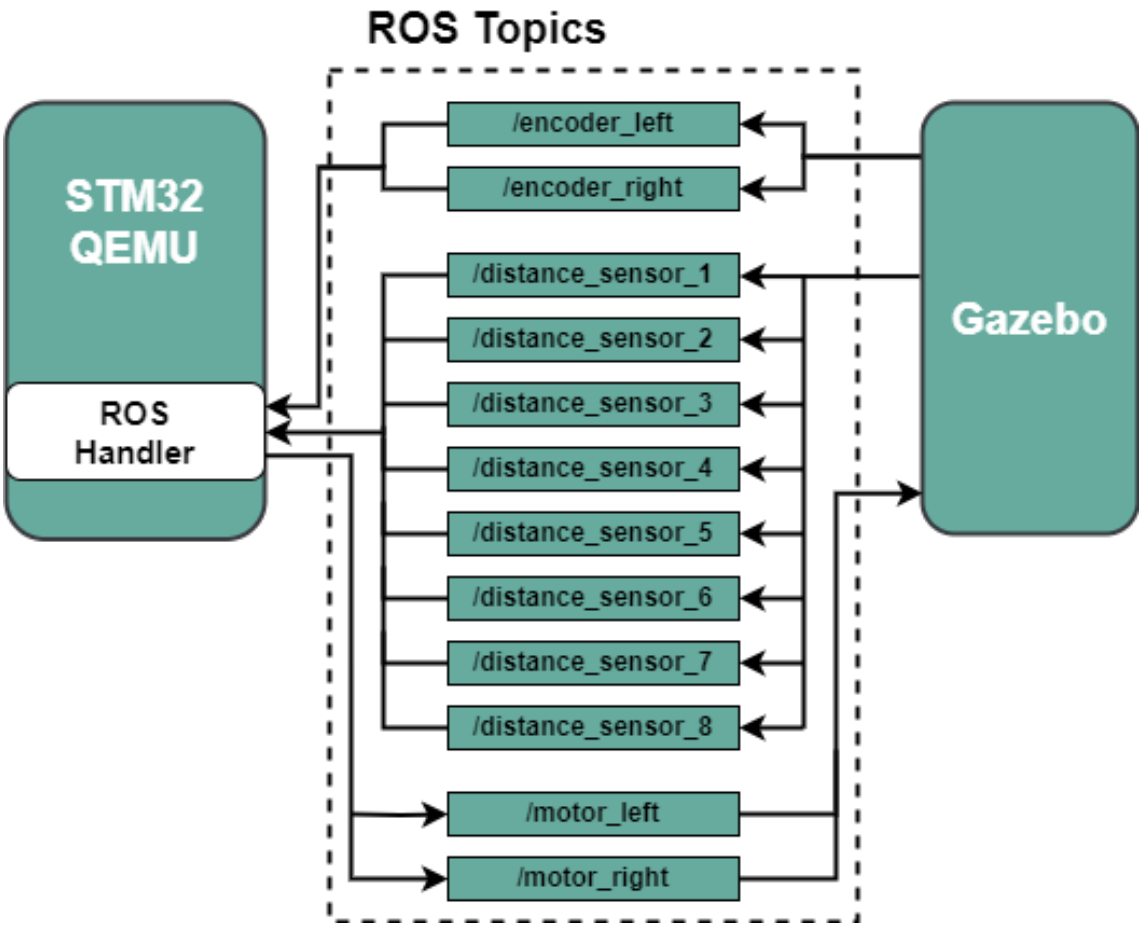
After integrating ROS into QEMU's emulation, it was possible to establish full communication between both systems. To achieve this, the following topics were defined:

- **distance_sensor_(1-8)**: Topics responsible for sending the output of the distance sensors. These are published by the Gazebo simulation and subscribed to on the QEMU side.
- **encoder_(left/right)**: Topics responsible for sending the encoder output. These are published by the Gazebo simulation and subscribed to on the QEMU side.
- **motor_(left/right)**: Topics responsible for sending motor commands. These are published by QEMU and subscribed to on the Gazebo simulation side.

By running both environments with these topic references and establishing connections between the ROS nodes, the systems were able to communicate effectively—emulation sending commands to the simulation and the simulation providing sensor information, completing, finally, the system implementation.

In the figure 14 it is possible to visualize all topics that have been published and subscribed to between the QEMU emulation and the Gazebo simulation. The arrows that are coming out of QEMU mean that it is the one that is publishing data there, and the arrows that are coming in QEMU mean that it has been reading. The same logic is applied to Gazebo.

Figure 14 – General topics schema



Within the STM32 QEMU environment, each **distance_sensor_(1-8)** and **encoder_(left/right)** has a topic subscription linked to a callback function. Whenever new data are received from the Gazebo simulation, the callback function is triggered, updating the memory location responsible for maintaining the GPIO input state.

For **motor_(left/right)**, there are also callback functions. However, instead of writing data to the microcontroller's memory, it reads data from the location responsible for maintaining the GPIO output state. This data is then published on the respective motor topic. The code example 5.4 aims to illustrate the process in detail. The function is automatically executed every 100 milliseconds.

Listing 5.4 – Callback function for motor 1 - From vl.c file

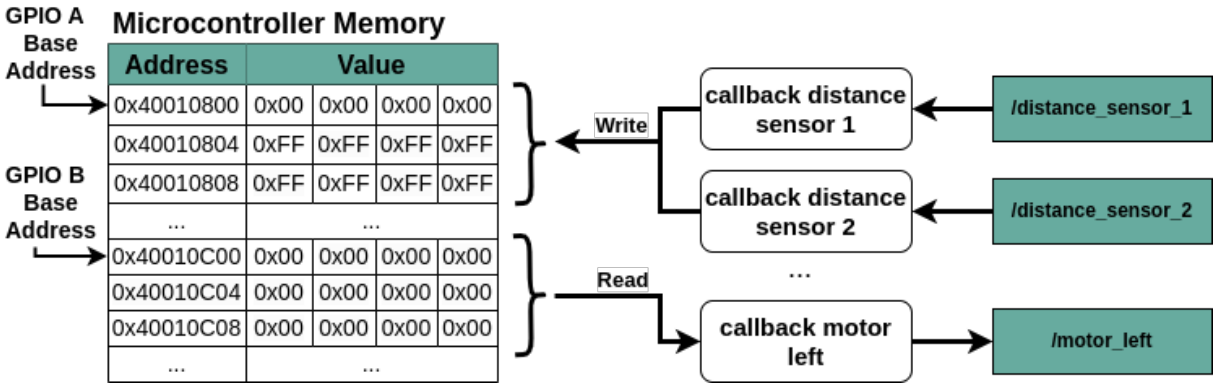
```
static CallbackResponse callback_motor_1(cRosMessage *message
    , void* data_context){
    char buf[1024];
    cRosMessageField *data_field;
    data_field = cRosMessageGetField(message, "data");
    uint32_t value, data;
    cpu_physical_memory_read(
        GPIOA_BASE_ADDR + 0x0c,
        &data, 4
    );
    value = (data >> 1) & 0b111;
    ROS_INFO(
        node, "Right motor publish value: [%d]\n",
        value
    );
    data_field->data.as_uint32 = value;

    cRosMessageSetFieldValueString(&data_field, buf);

    return 0;
}
```

In Figure 15, the flow of information can be visualized as it moves from the ROS topic to the microcontroller's memory for the **distance_sensor_(1-8)** and **encoder_(left/right)** and, subsequently, from the microcontroller's memory back to the ROS topic for the **motor_(left/right)**.

Figure 15 – Callback memory access example



5.3 Test Setup

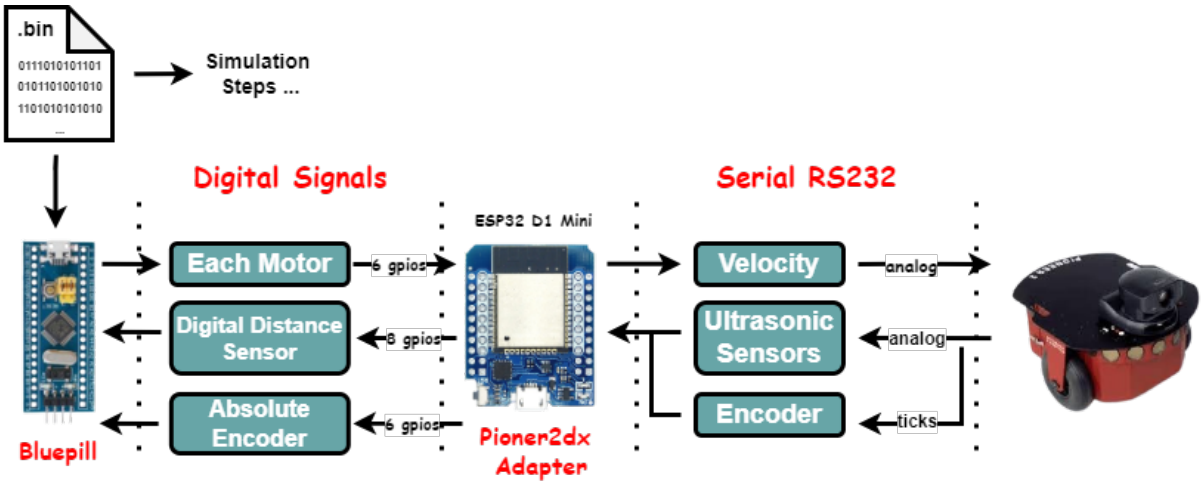
As the main project is a simulation, the best way to validate the project was to create the same situation that was created in the simulation in real life. With that, it could compare the performance in the simulated environment with a similar real-life situation.

For it, we choose to use the Pioneer 2 DX [13], the same robot that we used in the simulation, but as your simulation has not good support for communication protocols and PWM, we had to add one adapter between the information from the real robot and the bluepill.

In general, the Pioneer 2 DX adapter is responsible for getting all the information from the robot by communication protocol and sending it to the bluepill in the form of digital values with GPIOS. In contrast, the Pioneer 2 DX adapter is also responsible for getting velocity from each motor that the bluepill has sent and sending it back to pioneer 2DX.

It is possible to visualize the general schema in the figure 16.

Figure 16 – Test Setup - General



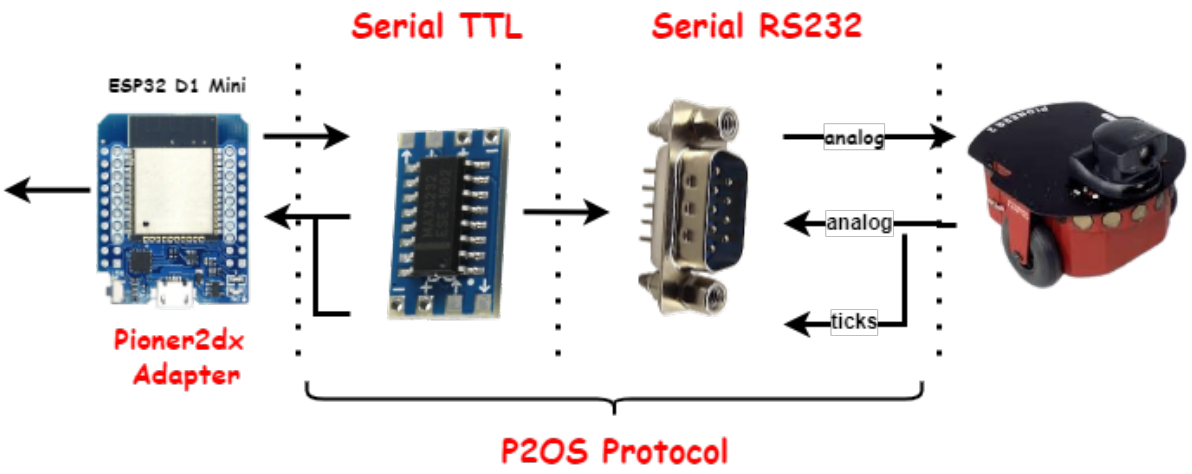
5.3.1 Hardware

The hardware of the project has two main functionality:

- Connect the ESP32 D1 Mini with the bluepill
- Connect the ESP32 D1 Mini with the pioneer 2DX

The connection between the Bluepill and the ESP32 is detailed in Table 3. A transistor is placed between the rst_signal from the Bluepill and the ESP32 D1 Mini. This

Figure 17 – Test Setup - Hardware



enables the ESP32 to be reset manually and ensures the signal stays high even when the Bluepill's signal is low.

It is possible to check the whole developed pcb for this interface in our github repository in our organization [28].

Table 3 – Connection between Esp32 and Bluepill

Label	Bluepill Pin	ESP32 Pin
DD1	PA4	SD3
DD2	PA5	TCK
DD3	PA6	IO5
DD4	PA7	IO25
DD5	PB0	IO19
DD6	PB1	IO18
DD7	PB10	IO26
DD8	PB11	SVP
rst_signal	PB13	RST
p2dx_con	PB12	SD0

Label	Bluepill Pin	ESP32 Pin
encoder1_s1	PB4	CLK
encoder1_s2	PB3	SD1
encoder1_s3	PB15	IO2
encoder2_s1	PC13	NC
encoder2_s2	PC14	SD2
encoder2_s3	PC15	CMD
motor1_comm1	PB7	TDI
motor1_comm2	PB6	IO4
motor1_comm3	PB5	IO0
motor2_comm1	PA1	IO27
motor2_comm2	PA2	IO25
motor2_comm3	PA3	IO32

To connect the ESP32 D1 Mini to the Pioneer 2DX, we used specific components for communication. The communication flow is illustrated in Figure 17, and the components' names and functions are detailed in Table 4.

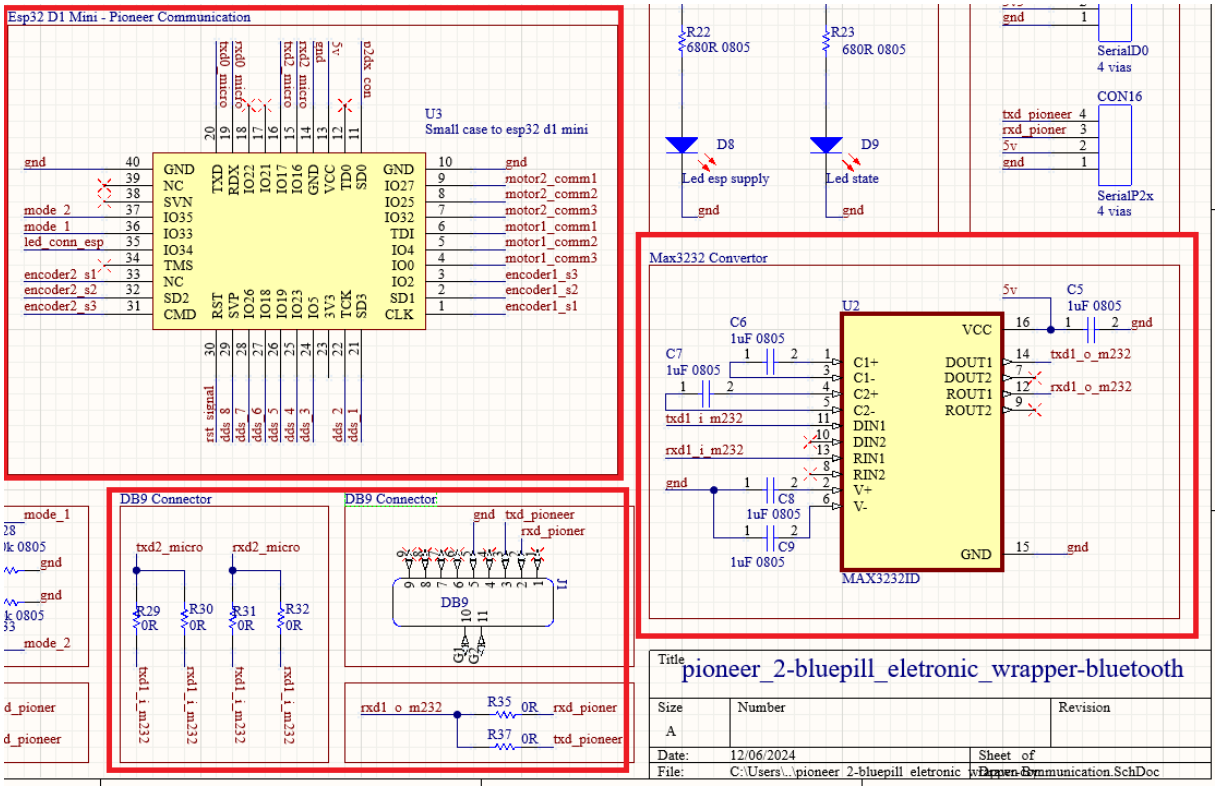
It is also possible to visualize the whole connection in the figure 18. The whole schematic can be visualized in Appendix A

Unfortunately, it was not possible to get the Printed Circuit Board (PCB) in time for this bachelor's thesis deadline, but to test and demonstrate the whole project, all the

Table 4 – Project main components

Main Component	Function
Bluepill	Main microcontroller of the project, it is resposible to receive the same code that is received in the simulation
Esp32 D1 Mini	Microcontroller used to create a interface between bluepill and pioner 2 DX
Max3232	Conversor between Serial TTL and Serial RS232
DB9 Connector	Connector type DB9, widely used for serial RS232
2 LM317 Linear regulator	Linear regulator for 3.3V and 5V

Figure 18 – Test Setup - ESP32 Connections



circuits that were planned to be used, except the voltage regulators, in the schematic were built on a protoboard.

Instead of voltage regulators, we are using a small 10,000 mAh Ugreen power bank [29] to supply 5V and the internal 3.3V regulator of the ESP32 D1 Mini to supply 3.3V to the circuit.

5.3.2 Real robot

As the main PCB of the project could not arrive in Brazil in time for manufacturing, it was necessary to build the biggest part of circuit on a protoboard.

On top of the Pioneer 2DX, as shown in Figure 19, there is a Ugreen power bank [29] placed above the visible wheel. A protoboard is positioned at the back of the robot. An RS232 cable extends from the protoboard and connects to the robot's serial port.

On the top of the robot, there are also three cables connected to the RS232 serial connector, entering the robot PCB on the top of the Pioneer 2DX, and being fixed in a better way. This setup exists to reduce intermittent contact between the communication pins.

Figure 19 – Real Robot With Prototype Hardware

5.3.2.1 Batteries

For the power supply of the entire robot, we are using the Intelbras 12V 7A [30]. The Pioneer 2DX supports up to three of these batteries. For the robot test, we used only one of them.

Figure 20 – Pioneer 2DX - Batteries Informations

5.3.3 Pioneer 2DX Interface Firmware

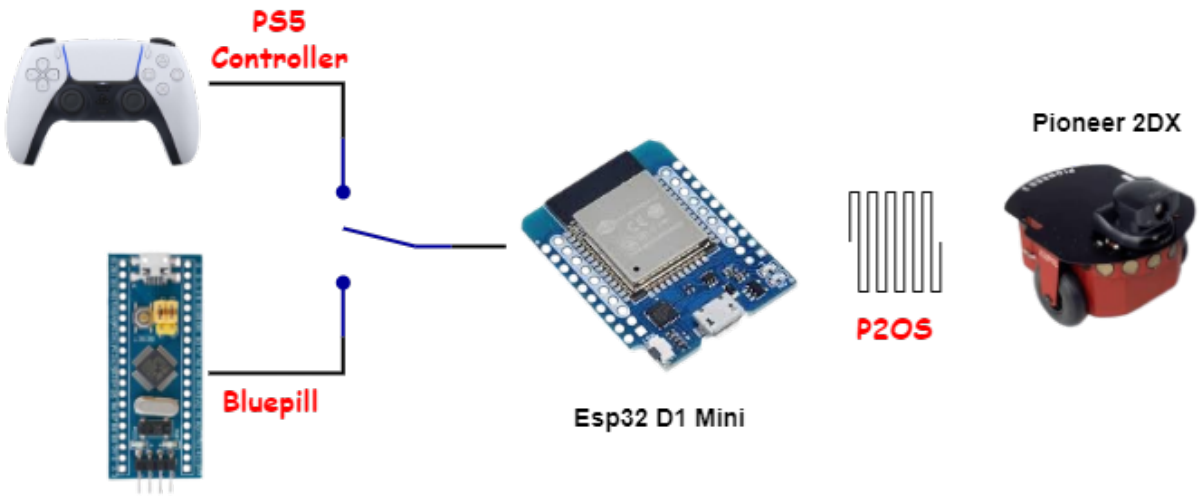
The Pioneer 2DX Interface has the mission to perform the computational logic behind the communication with the Pioneer 2DX and enable for an outside electronic device to send and receive data from the robot. Being more specific for this project, the Pioneer 2DX interface firmware has three main tasks:

- Connect the ESP32 D1 Mini with the bluepill
- Connect the ESP32 D1 Mini with a PS5 controller [31]
- Connect the ESP32 D1 Mini with the pioneer 2DX

The Pioneer 2DX interface allows either a PS5 controller [31] or the Bluepill device to send data to the ESP32, but never simultaneously, as shown in Figure 21. The complete code developed for this interface is available in the GitHub repository of our organization [32].

The PS5 Controller was added to the project to make it easier to validate the robot's mechanical and electrical flow without needing to use the autonomous code. This approach allows for identifying issues without suspecting the bluepill code.

Figure 21 – Test Setup - Pioneer 2dx Interface



5.3.3.1 Bluepill Connection

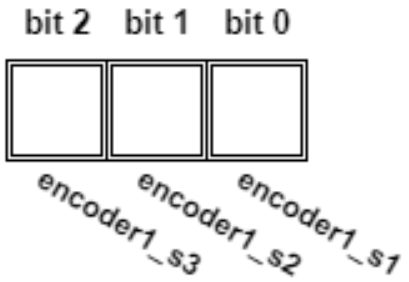
In general, the connection between the Bluepill and the ESP32 D1 Mini is performed through a digital signal. As there are values that need to be represented as decimal values to increase the precision, like a velocity of a motor, it is used as a concatenation of bits, in complement of 2, to send values through the Bluepill to the ESP32 D1 Mini.

For the bit concatenation, for example, there are the pins **motor2_comm1**, **motor2_comm2**, and **motor2_comm3**. The pin **motor2_comm3** represents the most significant bit, while **motor2_comm1** represents the least significant bit. Check Figure 22 to visualize an example for the encoder 1 data. It is also possible to check all the connection pins between the Bluepill and the ESP32 D1 Mini in Table 3. The code logic applied to convert the digital signal into a decimal value is described in the code example 5.5. It is possible to read the whole code in the project repository [32].

This bit transformation is applied for **encoder2_s1**, **encoder2_s2**, **encoder2_s3**, **encoder1_s1**, **encoder1_s2**, **encoder1_s3**, **motor2_comm1**, **motor2_comm2**, **motor2_comm3**, **motor1_comm1**, **motor1_comm2**, **motor1_comm3**.

For other values, like all the digital distance sensor values (DDx) and **p2dx_con** (responsible for initializing the communication with the robot), the information is passed through a normal GPIO value.

Figure 22 – Test Setup - Encoder Data as Bits Example



Listing 5.5 – BluepillCommunication::loop() function

```
int16_t linear_vel = (
    (this->motor\_gpio\_state[0] << 2)
    | (this->motor_gpio_state[1] << 1)
    | (this->motor_gpio_state[2])
);

int16_t angular_vel = (
    (this->motor_gpio_state[3] << 2)
    | (this->motor_gpio_state[4] << 1)
    | this->motor_gpio_state[5]
);
```

Regarding the digital distance sensor data, the ESP32 D1 Mini retrieves the data from the ultrasonic sensor on the Pioneer 2DX. Verify if the sensor value is less than half of the range (this can be easily adjusted in the main code). If it is, set the digital distance sensor pin to high; otherwise, set it to low.

Regarding the pin **p2dx_con**, in general, if the ESP32 D1 Mini receives a high signal on this pin for more than 10 ms, it initiates the connection with the Pioneer 2DX.

5.3.3.2 Ps5 Controller Connection

The connection with the PS5 Controller with the ESP32 D1 Mini is performed by the Bluetooth BLE [33] and the help of the Arduino library ps5-esp32 [34]. The library in

general allows you to quickly connect with a PS5 controller; it is only necessary to have the MAC address of the controller.

With the intention of making it easy to control the robot, the buttons to control the robot were chosen to work similarly to a car video game, which is possible to visualize in the figure 23.

Figure 23 – Test Setup - PS5 Controller Commands

The logic behind the entire code is possible to be read in the ESP32 D1 Mini project repository [32], however, the main logic can be checked in the code 5.6, what is a small part of the whole code.

Listing 5.6 – Part of the main loop() function - PS5

```
while (ps5.isConnected() == true) {
    if (ps5.Up() && (is_connected < 1)) {
        is_connected = !(p2os->setup());
    }

    if (ps5.Down() && (is_connected > 0)) {
        is_connected = p2os->shutdown();
    }

    if (is_connected) {
        p2os->loop();
        current_r2_val=scale(ps5.R2Value(), 0, 255, 0, 400);
        current_l2_val=scale(ps5.L2Value(), 0, 255, 0, 400);
        current_rs_x_val =
            (-1)*scale(ps5.RStickX(), -128, 128, -170, 170);
        msg_vel.linear.x = double(
            double(current_r2_val - current_l2_val) / 1000
        );
```

```
19          msg_vel.angular.z = double(
20              double(current_rs_x_val) / 100
21          );
22          p2os->set_vel(&msg_vel);
23          if (millis() - last_time_motor_state > 100) {
24              msg_motor_state.state = 1;
25              p2os->set_motor_state(&msg_motor_state);
26              last_time_motor_state = millis();
27          }
28      }
```

5.3.3.3 Pioneer 2DX communication

The Pioneer 2 DX has the mission of establishing a two-way, trustworthy communication with the robot. For the main emulation of the project, we are using encoders and digital distance sensors as sensors, and two motors as actuators. The communication aims to obtain the current position of the robot (which will be converted into the position of each wheel in the ESP32 logic for the Bluepill), the values from the ultrasonic sensor (which will be used as digital distance sensor values through a threshold for the Bluepill), and, simultaneously, transmit specific linear and angular velocities for the robot (derived from the speed of the left and right wheels for the Bluepill). For a better understanding of how this transformation occurs for the Bluepill, read the Bluepill connection topic 5.3.3.1.

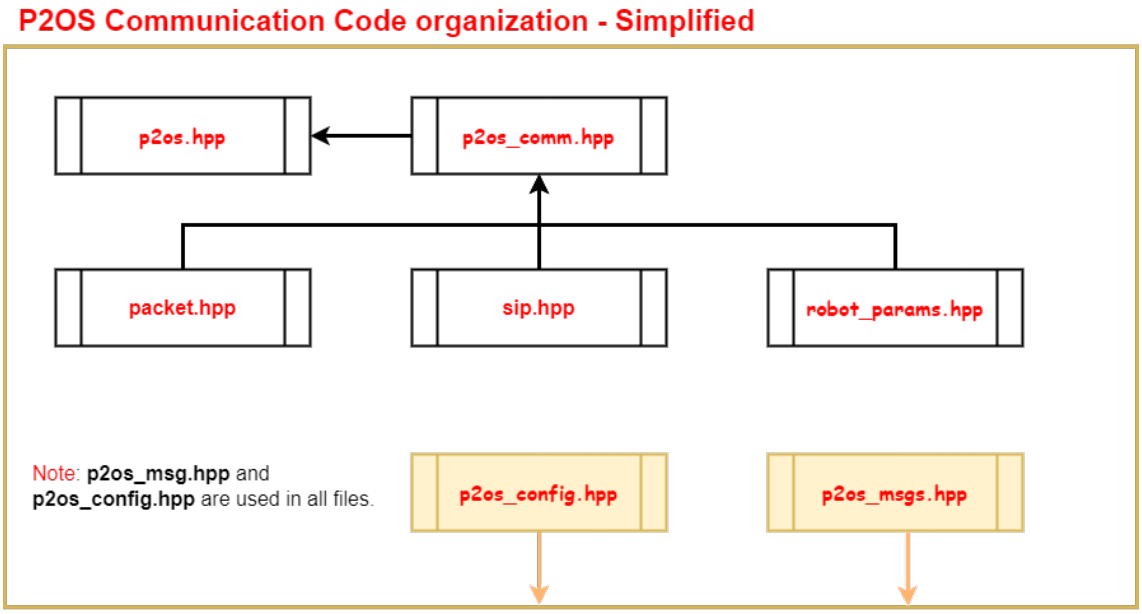
The Pioneer 2DX between the ESP32 D1 Mini is done by the P2OS Communication, which was described in the topic 2.6.

For the creation of the base code for this communication protocol, as inspiration, was used the ROS package P2OS [35], which is available on GitHub in the P2OS repository [36], specifically in the folder **p2os_driver**. Unfortunately, this ROS package is designed to be used on a computer with ROS [15], so for this project, it was necessary to translate most of the driver to make it compatible with a microcontroller using the Arduino library.

The simplified diagram of the P2OS-developed code is shown in Figure 24. For the complete diagram, read the appendix B. The explanation of what each code file is doing can be found below:

- **p2os**: Class responsible for performing the entire P2OS communication. It includes methods for startup, shutdown, handling main loop operations, retrieving robot data, and sending velocity and motor state formatted for the robot..
- **p2os_comm**: Class responsible for encapsulating parameters, settings, and helper methods to perform the entire P2OS communication protocol correctly. This class is

Figure 24 – Test Setup - P2OS Communication Code Organization - Simplified



also responsible for managing the entire infinite state machine required to connect with the robot, as described in the topic 2.6.

- **packet**: Class responsible for building messages (adding headers and checksums), verifying received messages, receiving messages from the Pioneer 2DX, and sending messages to the Pioneer 2DX.
- **sip**: Class responsible for processing and managing data exchanged with a robot, such as separating the data into different types of variables and filtering out incorrect data.
- **robot_params**: File responsible for managing all Pioneer robot type configurations, like the distance between wheels, available commands, number of ultrasonic sensors etc.
- **p2os_config**: File responsible for maintaining all the P2OS protocol configurations.
- **p2os_msgs**: File responsible for defining all message types used throughout the P2OS protocol.

As the **p2os.hpp** class describes the higher abstraction level of the communication, it is possible to observe in the main loop of this class how the algorithm works. Basically, every time the velocity input changes and it is a valid input value, it is sent to the robot, and the current data from the available sensors is retrieved. The same logic is applied to the motor state input. Additionally, at a predefined time interval specified in the **p2os_config** file, a pulse is sent to the robot to maintain the connection. Furthermore, even if nothing

is being sent, the SendReceive data function is called to continue receiving and saving data from the robot's sensors.

The main loop is described here in the code example 5.7.

Listing 5.7 – Part of the P2OS::loop() function

```
1  void P2OS::loop() {
2      this->current_time = millis();
3
4      this->p2os_comm->check_and_set_vel();
5      this->p2os_comm->check_and_set_motor_state();
6
7      if (this->p2os_comm->get_pulse() > 0) {
8          if (this->p2os_comm->millis2Sec(this->current_time -
               this->last_time_pulse) > this->p2os_comm->
               get_pulse()) {
9  #ifdef P2OS_DEBUG_PRINT
10             this->debug_serial->println("sending pulse");
11 #endif
12             this->p2os_comm->SendPulse();
13             this->last_time_pulse = this->current_time;
14         }
15     }
16
17     this->p2os_comm->SendReceive(NULL, true);
18     this->p2os_comm->updateDiagnostics();
19 }
```

5.4 Results

After completing the project development and documentation, it was possible to validate the project's main goal: ensuring that the simulator behaves exactly like the real robot while running the exact same binaries.

The proof of concept was established through the following tests:

- **Motor command data verification**
- **Distance sensor data acquisition verification**
- **Encoder data verification**

- **Obstacle avoidance software testing**: A software implementation that enables the robot to avoid obstacles.

Each component of the emulation-simulation connection was tested individually to ensure proper functionality and accuracy. For security reasons, the values published on the simulation's ROS topics were compared with the values transmitted via the ESP32. After these comparisons, the binaries were tested by running them in both the real setup and the simulation, ensuring consistent behavior in both environments.

5.4.1 Motors command validation

The first test aimed to verify the commands sent to the motors. As detailed in Bluepill Connection5.3.3.1, the command signal was a 3-bit two's complement value. To move forward, the robot could receive values between 1 and 3, while for backward motion, values ranged from -1 to -4.

To validate the motor commands, both the simulation and the real robot were tasked with executing two simple movements: moving forward and backward. The binary test initialized the command at 0, then transitioned to 3, back to -4, and finally returned to 0.

As a first step, the commands published on the ROS topics were compared with those sent by the ESP32 D1 Mini. Subsequently, the simulation's behavior was compared to that of the real robot.

The results showed consistency between the simulation and the real robot. However, due to the limited precision of the 3-bit command signal, the motor transitions were somewhat abrupt and lacked smoothness. For improved performance, a higher precision command range would be recommended to enable smoother motor acceleration and deceleration.

5.4.2 Distance sensors validation

To verify the functioning of the distance sensors, each sensor was tested individually, followed by a test of all sensors operating simultaneously. Initially, the values read by the ROS topics were compared with those obtained from the ESP to ensure consistency.

Next, a more dynamic test was conducted where the robot was programmed to move forward only when a sensor detected an obstacle. This test was designed to evaluate the responsiveness of the system.

Finally, the simulation results were compared with the real robot's behavior. The real robot responded as expected, validating the accuracy and functionality of the distance sensors.

5.4.3 Encoders validation

The encoder validation primarily focused on comparing the data received from the simulation with the data obtained from the ESP32. Once the values matched, the encoder functionality was satisfactory. However, it was not possible to perform a comprehensive test of the encoders due to precision differences between the simulation and the physical encoders on the Pioneer 2 robot.

5.4.4 Dodge obstacles validation

After completing the individual component validations, a full robot application was tested: obstacle dodging. Widely used in mobile robotics, this task combines robot control and sensor responsiveness, making it an effective demonstration of the simulator's comprehensive validation.

The developed software employs a finite state machine (FSM) to guide the robot's navigation—a common approach in the field of robotics.

With this software, the project was able to demonstrate practical application in a real-world scenario. Moreover, the simulator served as a valuable tool for testing the obstacle-dodging software before deploying it on the physical robot, further reinforcing the project's goals and purpose.

As anticipated, the simulation produced satisfactory results, with the real robot successfully performing the task as intended.

6 Final considerations

6.1 Conclusion

In conclusion, this project represents a important advancement in robotics simulation by successfully integrating a hardware emulator with a fully simulated environment. This achievement not only validated the project's primary objectives, as shown in the section 1.2, but also underscored its utility and potential applications. During its implementation and testing, the simulator proved its value as a self-validating tool and debugger, particularly when integrated with the Pioneer 2DX system, as described in Section 5.3.

Working on this project has been a gratifying experience, culminating in the delivery of an innovative solution. The integration of hardware emulation and simulation stands as a pioneering effort, opening doors for further exploration in this domain. By successfully bridging the gap between these two technologies, this project lays the groundwork for future advancements, encouraging others to delve into the field of robotics simulation.

As the first initiative of its kind, this project serves as inspiration for others to collaborate, innovate, and contribute to the development of new simulators. Such efforts have the potential to drive even greater progress in robotics, fostering creativity and accelerating advancements in the field.

6.2 Contributions

As the conclusion of this work, we successfully integrated QEMU with Gazebo, two completely different technologies. This results demonstrates that it is entirely possible to fully integrate a microcontroller within an emulation environment and opens for diffent types of implementation in the future.

Besides covering the entire scope of the project, it was one of the first open-source initiatives to emulate a wide range of microcontroller peripherals simultaneously, using ROS to make the GPIOs easily readable and writable outside the emulation, like an interface. With this strategy, it has the potential to enable much more scalable projects in the future.

Regarding the test setup of the robot, as described in Section 5.3, it was necessary to translate the entire ROS P2OS library, originally designed for use on a computer, to be compatible with a standard microcontroller and arduino IDE. Since no other library is available to perform the same function, this enables the possibility of making it open source and adding future contributions to support the work of other students who wish to

use the Pioneer 2DX.

6.3 Prospects for Continuity

As unconvencional and multi area project, there are a plenty of possible improvement in different aspects.

Regarding the microcontroller emulation, it would be nice to add the possibility to emulate peripherals derived from timers, such as PWM, ADC, and several communication protocols. It would also be interesting to build a better architecture in the STM32 QEMU project.

Regarding the environment simulation, it would be interesting to try different types of simulators, such as the CARLA simulator [37], which offers alternative ways to model and visualize sensors and actuators.

Regarding the integration between the emulation and the simulation, it would be better to use a ROS2 provider instead of ROS Noetic (the last version of ROS1), as ROS1 will reach its end of life around the beginning of 2024 [38]. Furthermore, Gazebo Classic, also used in the project, will also reach its end of life next year [39], making it necessary to update. Updating the software versions is extremely important to ensure the use of the safest and most stable version of the product.

Regarding the test setup, it would be beneficial to manufacture the entire PCB using the current available schematic and improve the battery support in the robot, as it is not properly secured at the moment.

Bibliography

1 Natalie Rosa. Boeing admite que sabia de falhas de software antes de dois acidentes fatais. *CanalTech*, Maio 2019. Disponível em: <https://canaltech.com.br/software/boeing-admite-que-sabia-de-falhas-de-software-antes-de-dois-acidentes-fatais-138563/l>. Citado na página 17.

2 Caroline Sassatelli e Guilherme Blanco Muniz. Falha em software pode ter motivado acidente fatal com autônomo da Uber. *autoesporte - globo*, Maio 2018. Disponível em: <https://autoesporte.globo.com/carros/noticia/2018/05/falha-em-software-pode-ter-motivado-acidente-fatal-com-autonomo-da-uber.ghtml>. Citado na página 17.

3 Arduino. *Arduino CC*. 2023. New York, New York, United States. Organization - For Profit. Disponível em: <https://www.arduino.cc>. Citado na página 17.

4 STMicroelectronics. *STMicroelectronics*. 2023. Geneva, Switzerland. . Disponível em: <https://www.st.com/content/st_com/en.html>. Citado 2 vezes nas páginas 17 and 39.

5 The Gazebo Team. *Robot simulation made easy*. 2021. Disponível em: <http://gazebosim.org/>. Citado 2 vezes nas páginas 18 and 40.

6 QEMU. *QEMU A generic and open source machine emulator and virtualizers*. [S.l.], 2024. Disponível em: <https://www.qemu.org>. Citado 2 vezes nas páginas 18 and 36.

7 ELECTRONICHUB. *Basics of Microcontrollers – History, Structure and Applications*. Disponível em: <https://www.electronicshub.org/microcontrollers-basics-structure-applications/>. Citado na página 22.

8 STMICROELECTRONICS. *Medium-density performance line Arm®-based 32-bit MCU with 64 or 128 KB Flash, USB, CAN, 7 timers, 2 ADCs, 9 com. interfaces*. [S.l.], 2022. Disponível em: <https://www.st.com/resource/en/datasheet/stm32f103c8.pdf>. Citado 2 vezes nas páginas 23 and 39.

9 Google. *Google Protocol Buffers*. 2023. Mountain View, California, United States. Organization - For Profit. Disponível em: <https://protobuf.dev/>. Citado na página 24.

10 Microsoft. *Publisher-Subscriber pattern*. 2023. Redmond, Washington, United States. Organization - For Profit. Disponível em: <https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber>. Citado na página 24.

11 Linux KVM Project. *Linux Kernel-based Virtual Machine (KVM)*. 2023. Official Website. An open-source virtualization technology integrated into the Linux kernel. Disponível em: <https://linux-kvm.org/page/Main_Page>. Citado na página 25.

12 Wikipedia contributors. *ARM architecture family – Wikipedia*. 2024. [Online; accessed 10-December-2024]. Disponível em: <https://en.wikipedia.org/wiki/ARM_architecture_family>. Citado na página 26.

13 ROBOTICS, A. *Pioneer 2 DX Operations Manual*. [S.l.], 2001. Accessed: 2024-12-05. Disponível em: <https://www.iri.upc.edu/groups/lrobots/private/Pioneer2/AT_DISK1/DOCUMENTS/p2opman9.pdf>. Citado 4 vezes nas páginas 27, 29, 40, and 49.

14 LABS, P. *PlatformIO: Professional Collaborative Platform for Embedded Development*. 2024. Accessed: 2024-12-10. Disponível em: <https://platformio.org>. Citado na página 35.

15 ROS FOUNDATION INC. *ROS - Robot Operating System*. [S.l.], 2024. Disponível em: <https://www.ros.org>. Citado 2 vezes nas páginas 36 and 56.

16 SYSTEMS, E. *ESP32*. 2024. Accessed: 2024-12-05. Disponível em: <https://www.espressif.com/en/products/socs/esp32>. Citado na página 38.

17 STMicroelectronics. *STM32CubeMX: Initialization Code Generator*. 2023. Geneva, Switzerland. Software tool for STM32 microcontroller configuration and code generation. Disponível em: <https://www.st.com/en/development-tools/stm32cubemx.html>. Citado na página 39.

18 Generic Manufacturers. *Blue Pill Development Board: STM32F103-based Microcontroller Board*. 2023. Widely used open-source development board. Compact, affordable development board featuring the STM32F103C8 microcontroller. Disponível em: <https://stm32-base.org/boards/STM32F103C8T6-Blue-Pill.html>. Citado na página 39.

19 Beckus and Contributors. *QEMU STM32: STM32 Microcontroller Emulator for QEMU*. 2023. GitHub Repository. QEMU-based emulator for STM32 microcontrollers, enabling simulation of STM32 applications without hardware. Disponível em: <https://github.com/beckus/qemu_stm32>. Citado na página 39.

20 Autodesk and Other CAD Software Developers. *Computer-Aided Design (CAD)*. 2023. Digital Design and Modeling Software Tools. Software used for precision design, simulation, and modeling in various industries. Disponível em: <https://en.wikipedia.org/wiki/Computer-aided_design>. Citado na página 40.

21 Adept MobileRobots. *Pioneer 3 Robot Platform*. 2023. Official Mobile Robotics Development Platform. Versatile robot platform used in research and education, featuring modular design and advanced sensors. Disponível em: <https://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx>. Citado na página 40.

22 Mario Serna and Contributors. *Pioneer P3-DX CAD Model Repository*. 2023. GitHub Repository. Open-source CAD model for the Pioneer P3-DX robot. Disponível em: <https://github.com/mario-serna/pioneer_p3dx_model/tree/master>. Citado na página 40.

23 Open Source Robotics Foundation (OSRF). *ROS Topics: Communication Mechanism in Robot Operating System*. 2023. Online Documentation and Tutorials. Core concept in ROS enabling publish/subscribe communication between nodes. Disponível em: <https://wiki.ros.org/Topics>. Citado na página 40.

24 ROS and Ray Project Developers. *ROS Ray Sensor: Integration of Ray Sensors with Robot Operating System*. 2023. Online Documentation and GitHub Repository. ROS package for integrating Ray sensors in robotic applications, enabling sensor data processing and communication. Disponível em: <https://github.com/ray-project/ray/tree/master/python/ray/experimental/ros_sensor>. Citado na página 40.

25 Nils Europa and Contributors. *Gazebo ROS Motors Plugin: Integration of Motors in Gazebo with ROS*. 2023. GitHub Repository. Plugin for integrating motor control in Gazebo simulations with ROS, enabling realistic motor dynamics and control. Disponível em: <https://github.com/nilseuropa/gazebo_ros_motors>. Citado na página 42.

26 ROS Industrial Attic and Contributors. *CROS: ROS-based Control Software for Industrial Robots*. 2023. GitHub Repository. CROS is a collection of software for controlling industrial robots in a ROS environment. Disponível em: <https://github.com/ros-industrial-attic/cros>. Citado na página 44.

27 ROS-INDUSTRIAL. *cROS Manual*. [S.l.], n.d. Version available at GitHub. Disponível em: <https://github.com/ros-industrial-attic/cros/blob/master/docs/cROS_manual.pdf>. Citado na página 44.

28 TEAM, Q.-G. S. *Pioneer 2DX Interface - PCB*. 2024. Accessed: 2024-12-06. Disponível em: <https://github.com/qemu-gazebo-sim/pioneer_2DX-Interface/tree/master/PCB_Project>. Citado na página 50.

29 Ugreen Brasil. *Carregador Sem Fio Ugreen 10000mAh 20W Power Bank Cinza*. 2024. Accessed: 2024-12-11. Disponível em: <https://ugreendobrasil.com.br/produto/carregador-sem-fio-ugreen-10000mah-20w-power-bank-cinza/?srsltid=AfmBOopIYC9qEaEK-_pi07Ne5WbIGNc5jrGfm3ZDJlUxDqeqGEk3A3U3>. Citado 2 vezes nas páginas 51 and 52.

30 Intelbras. *Bateria de Chumbo Ácido 12V XB 1270*. 2024. Acessado em: 11 de dezembro de 2024. Disponível em: <https://www.intelbras.com/pt-br/bateria-de-chumbo-acido-12v-xb-1270>. Citado na página 52.

31 ENTERTAINMENT, S. I. *DualSense™ Wireless Controller*. 2024. Acesso em: 7 de dezembro de 2024. Disponível em: <https://www.playstation.com/pt-br/accessories/dualsense-wireless-controller/>. Citado na página 53.

32 TEAM, Q.-G. S. *Pioneer 2DX Interface - ESP32 Code*. 2024. Accessed: 2024-12-06. Disponível em: <https://github.com/qemu-gazebo-sim/pioneer_2DX-Interface/tree/master/code-esp32_d1>. Citado 2 vezes nas páginas 53 and 55.

33 Wikipedia contributors. *Bluetooth Low Energy*. 2024. Accessed: 2024-12-07. Disponível em: <https://en.wikipedia.org/wiki/Bluetooth_Low_Energy>. Citado na página 54.

34 BAKISKAN, R. *PS5 ESP32*. 2023. <https://github.com/rodneybakiskan/ps5-esp32>. Acesso em: 7 de dezembro de 2024. Disponível em: <https://github.com/rodneybakiskan/ps5-esp32>. Citado na página 54.

35 H., A. *P2OS*. 2024. Accessed: 2024-12-08. Disponível em: <https://wiki.ros.org/p2os>. Citado na página 56.

36 H., A. *p2os*. 2024. <https://github.com/allenh1/p2os>. Accessed: 2024-12-08. Citado na página 56.

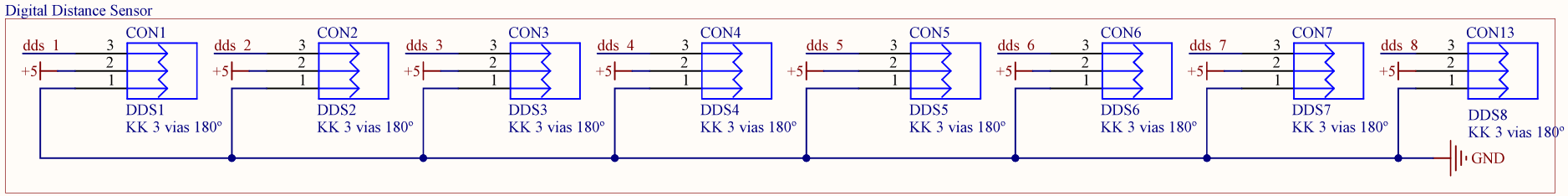
37 CARLA. *CARLA Simulator*. 2024. Accessed: 2024-12-11. Disponível em: <https://carla.org>. Citado na página 62.

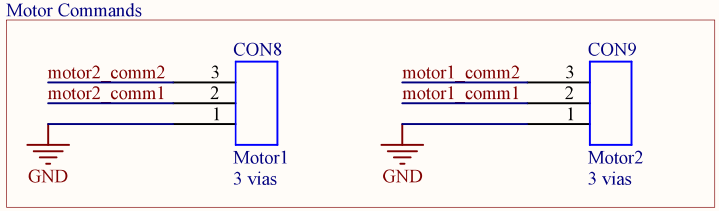
38 DISCOURSE, R. *ROS News for the Week of December 2nd, 2024*. 2024. <https://discourse.ros.org/t/ros-news-for-the-week-of-december-2nd-2024/40997>. Accessed: 2024-12-11. Citado na página 62.

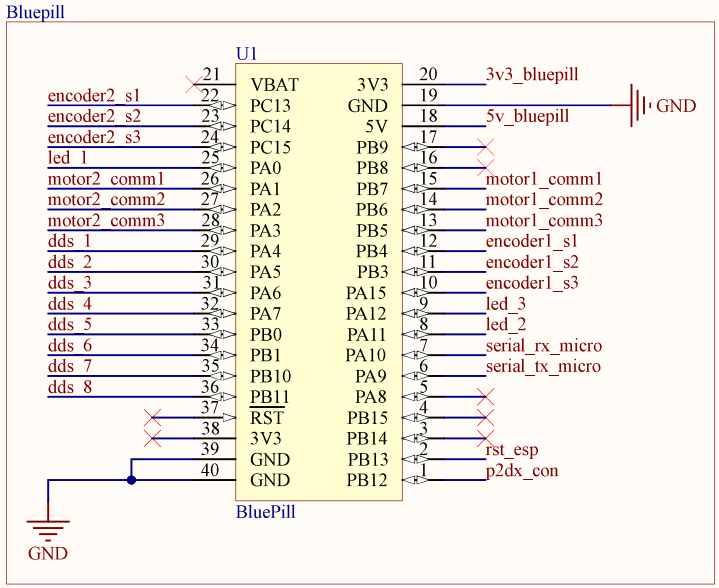
39 SCOTT, K. *Gazebo Classic and Citadel End of Life [x-post Gazebo Sim Community]*. 2024. Accessed: 2024-12-11. Disponível em: <https://discourse.ros.org/t/gazebo-classic-and-citadel-end-of-life-x-post-gazebo-sim-community/40931>. Citado na página 62.

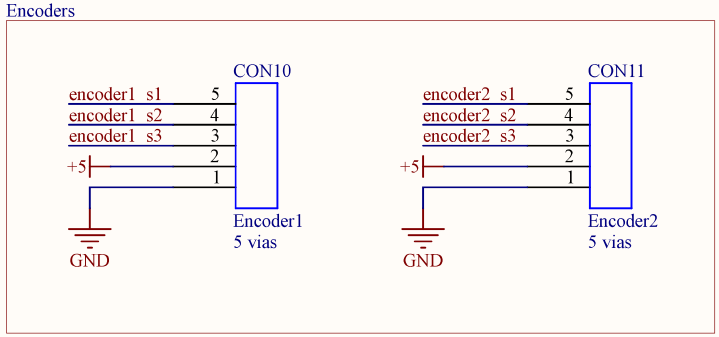
APPENDIX A – Pioneer 2DX Interface Hardware

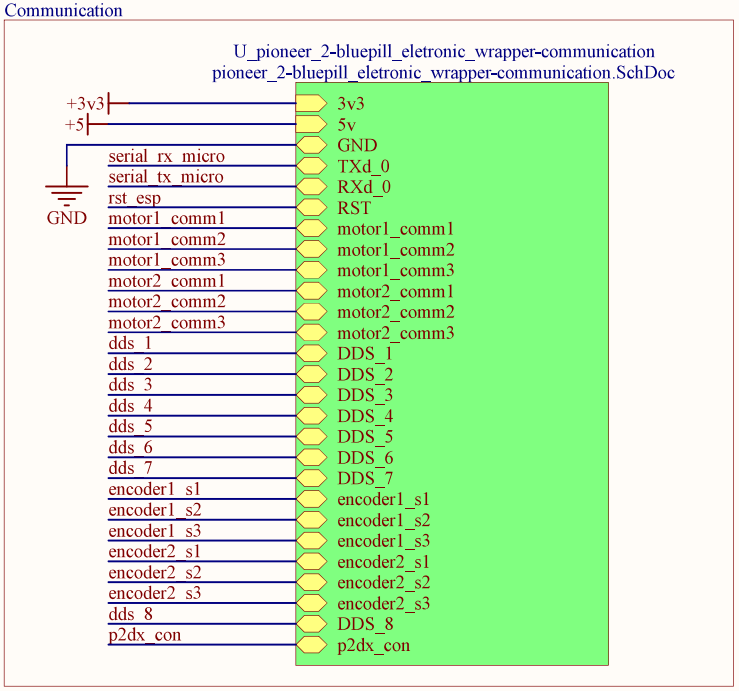
A.1 Schematics

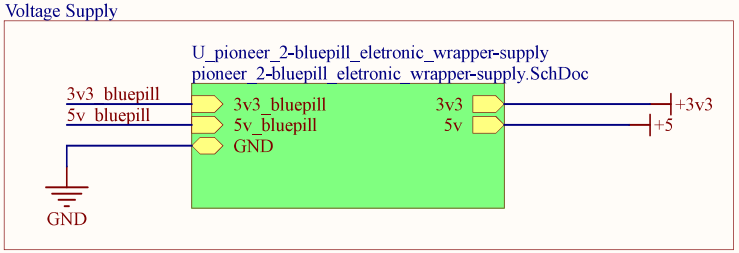


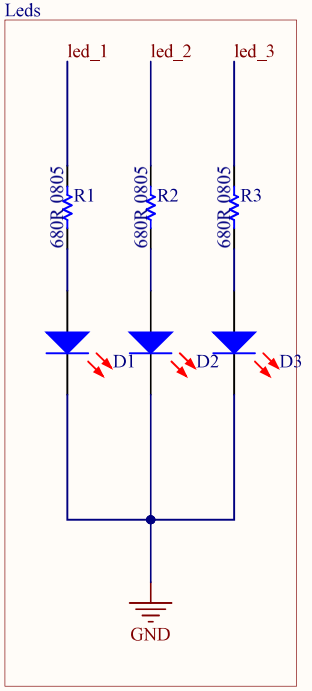






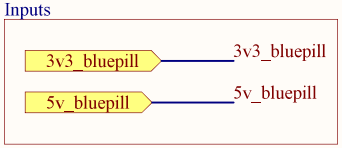




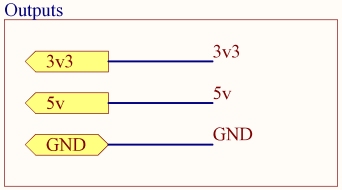


Title	pioneer_2-bluepill_eletronic_wrapper	
Size: A	Number	Revision
Date: 12/05/2024	Sheet of	
File: C:\Users\...\pioneer_2-bluepill_eletronic_wrapper.SchDoc	Drawn By:	

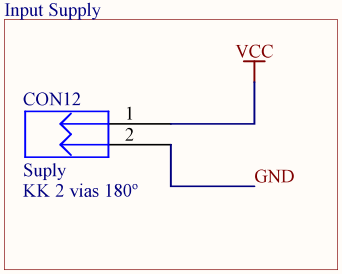
Inputs



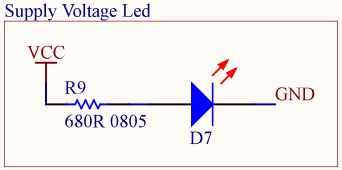
Outputs



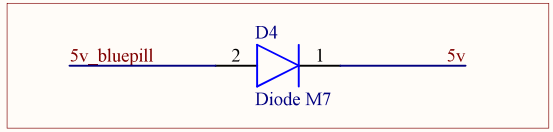
Input Supply



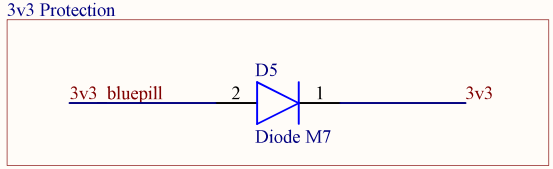
Supply Voltage Led



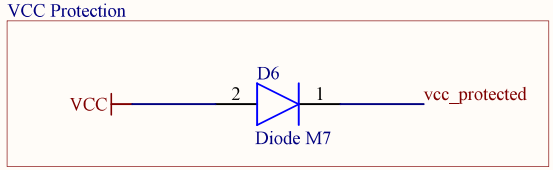
5v Protection



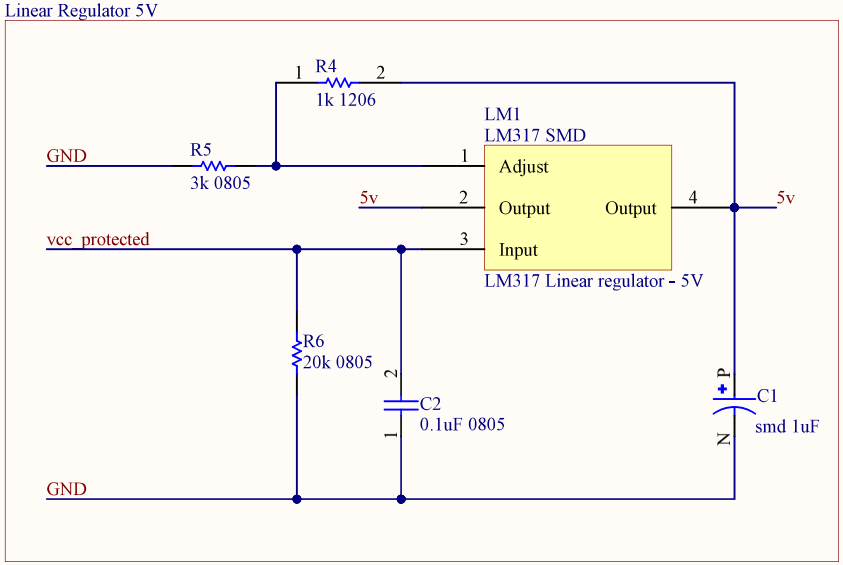
3v3 Protection



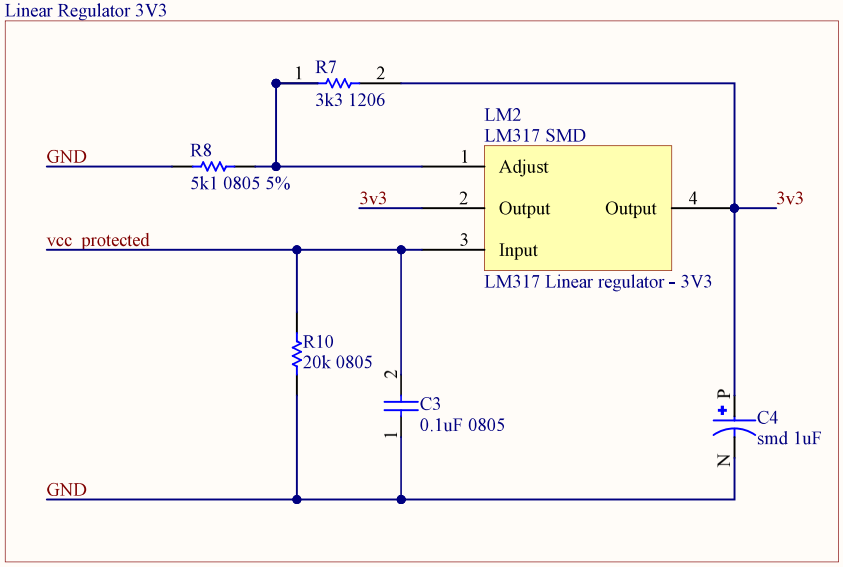
VCC Protection



Linear Regulator 5V

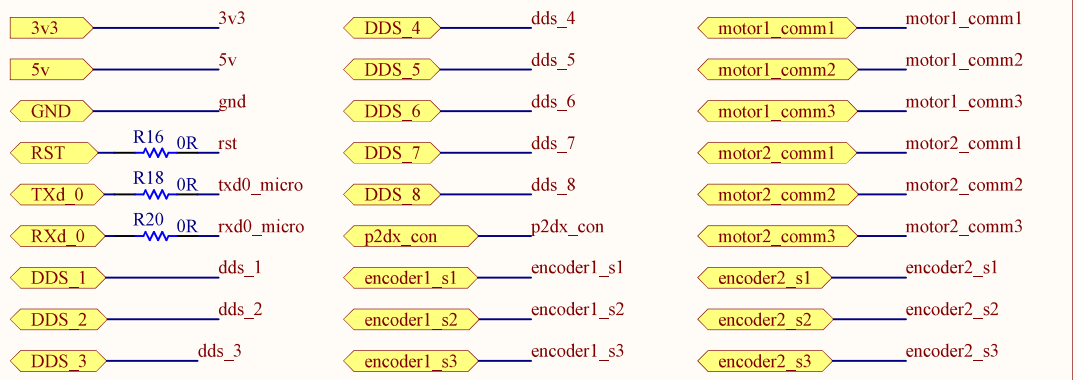


Linear Regulator 3V3

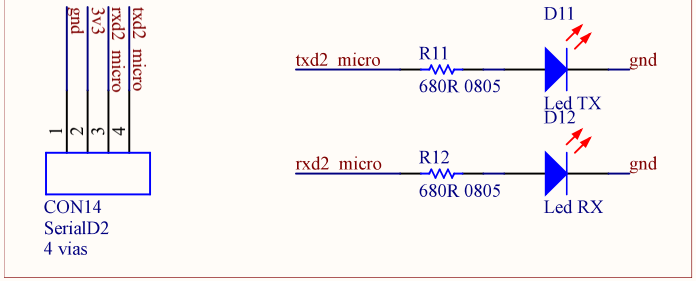


Title	pioneer_2-bluepill_eletronic_wrapper-supply	
Size: A	Number:	Revision:
Date: 12/05/2024	Sheet of	
File: C:\Users\...\pioneer_2-bluepill_eletronic_wrapper-supply.SchDoc	Drawn By:	

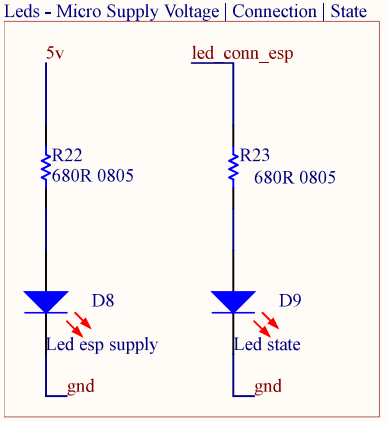
Inputs & Outputs



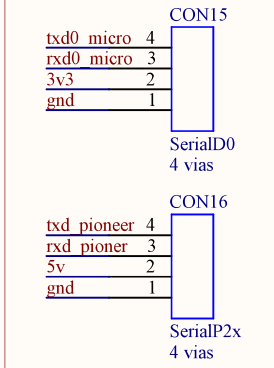
Serial



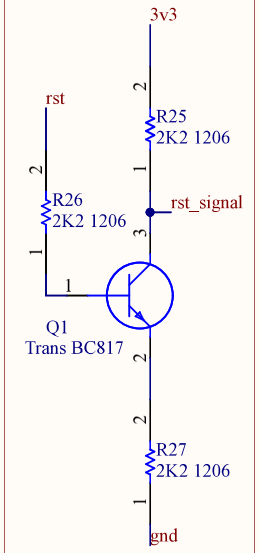
Leds - Micro Supply Voltage | Connection | State



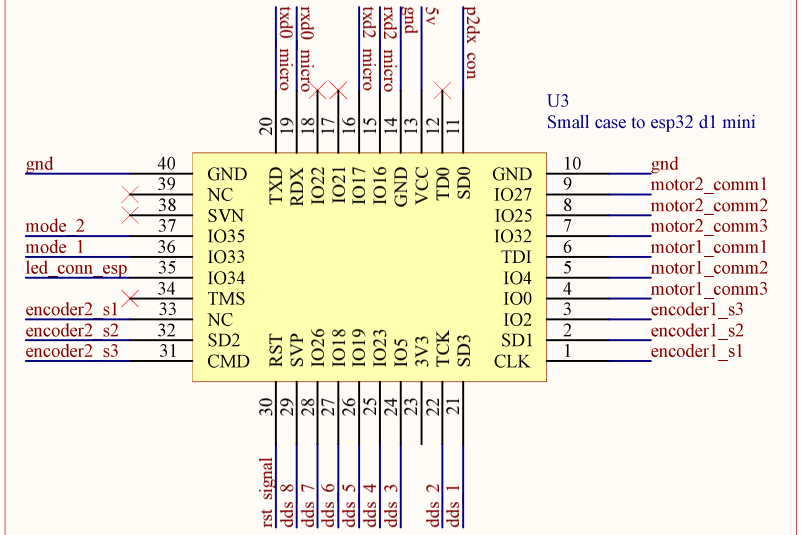
Serial



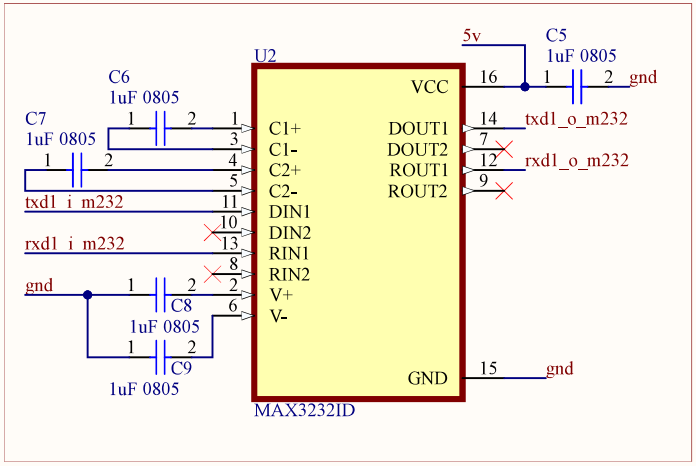
Reset



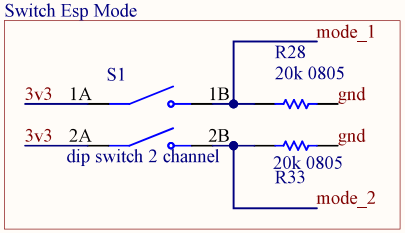
Esp32 D1 Mini - Pioneer Communication



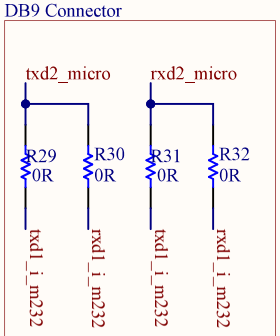
Max3232 Convertor



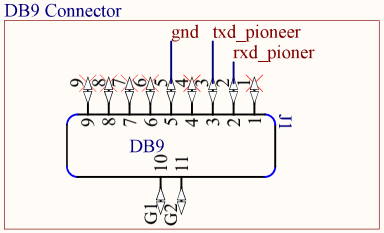
Switch Esp Mode

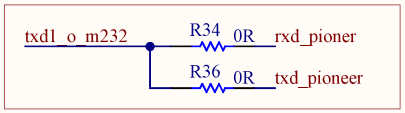


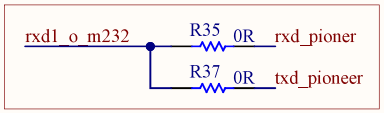
DB9 Connector



DB9 Connector







Title	pioneer_2-bluepill_eletronic_wrapper-bluetooth	
Size A	Number	Revision
Date:	12/05/2024	Sheet of
File:	C:\Users\...\pioneer_2-bluepill_eletronic_wrapper-Bluetooth.SchDoc	

A.2 PCB

Figure 25 – Pioneer 2 Interface PCB - 3D

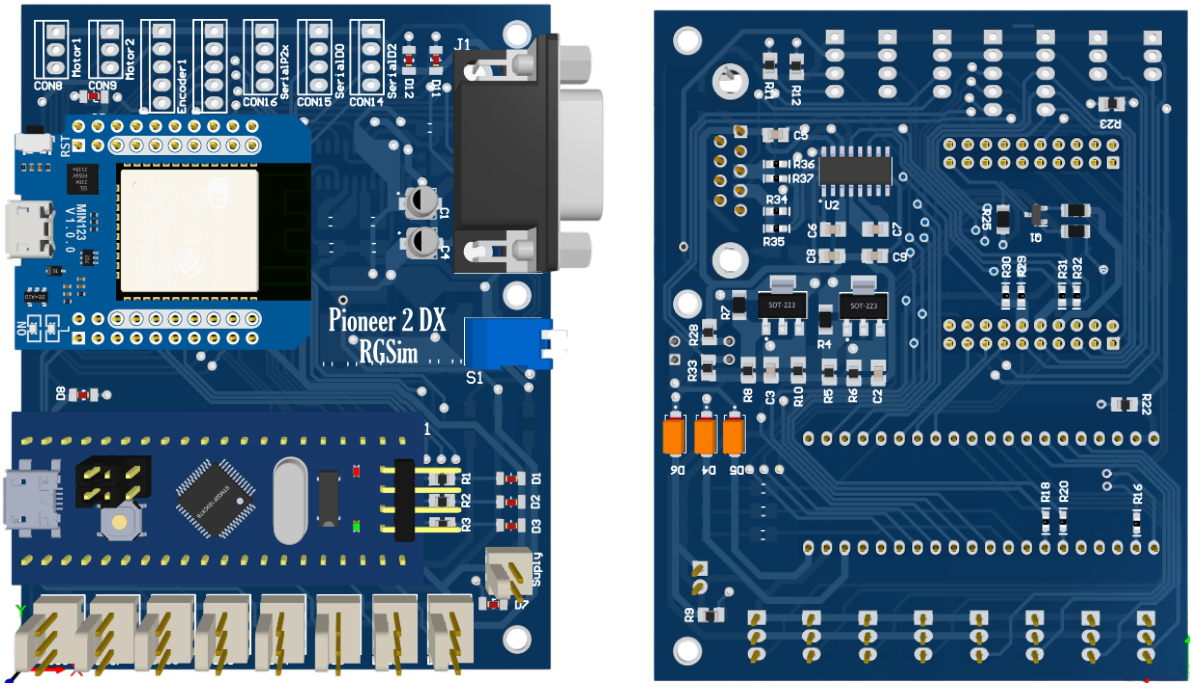
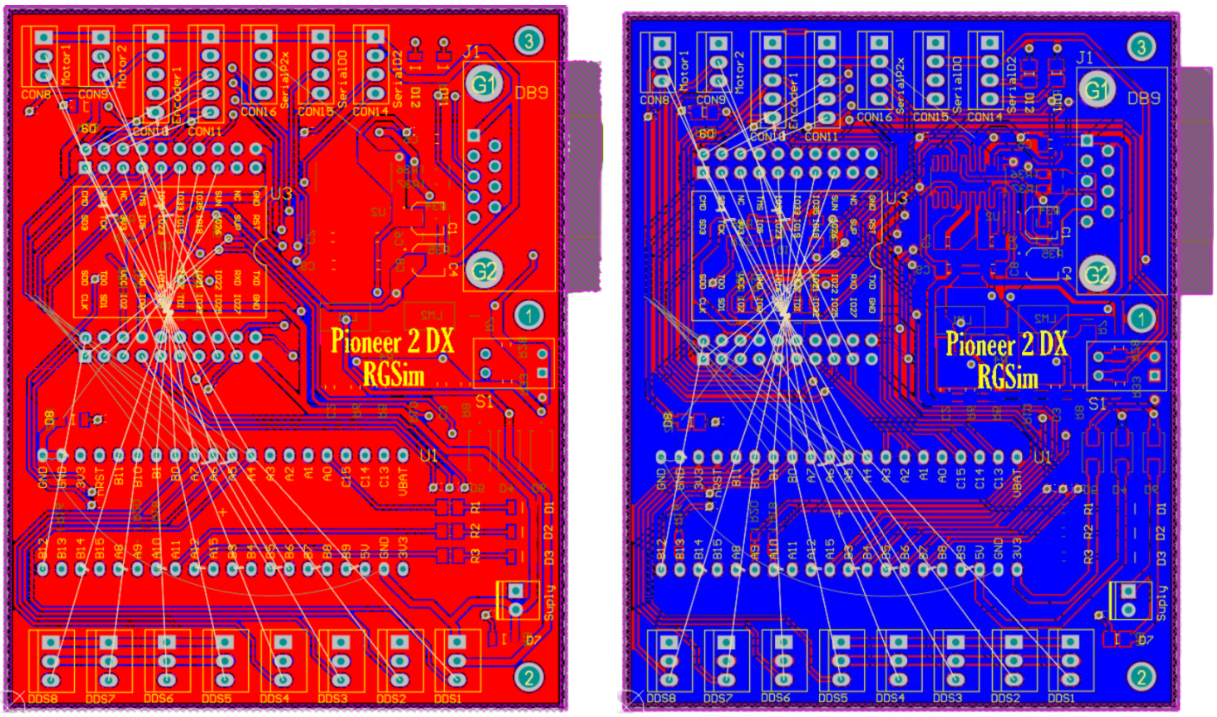


Figure 26 – Pioneer 2 Interface PCB - Routes

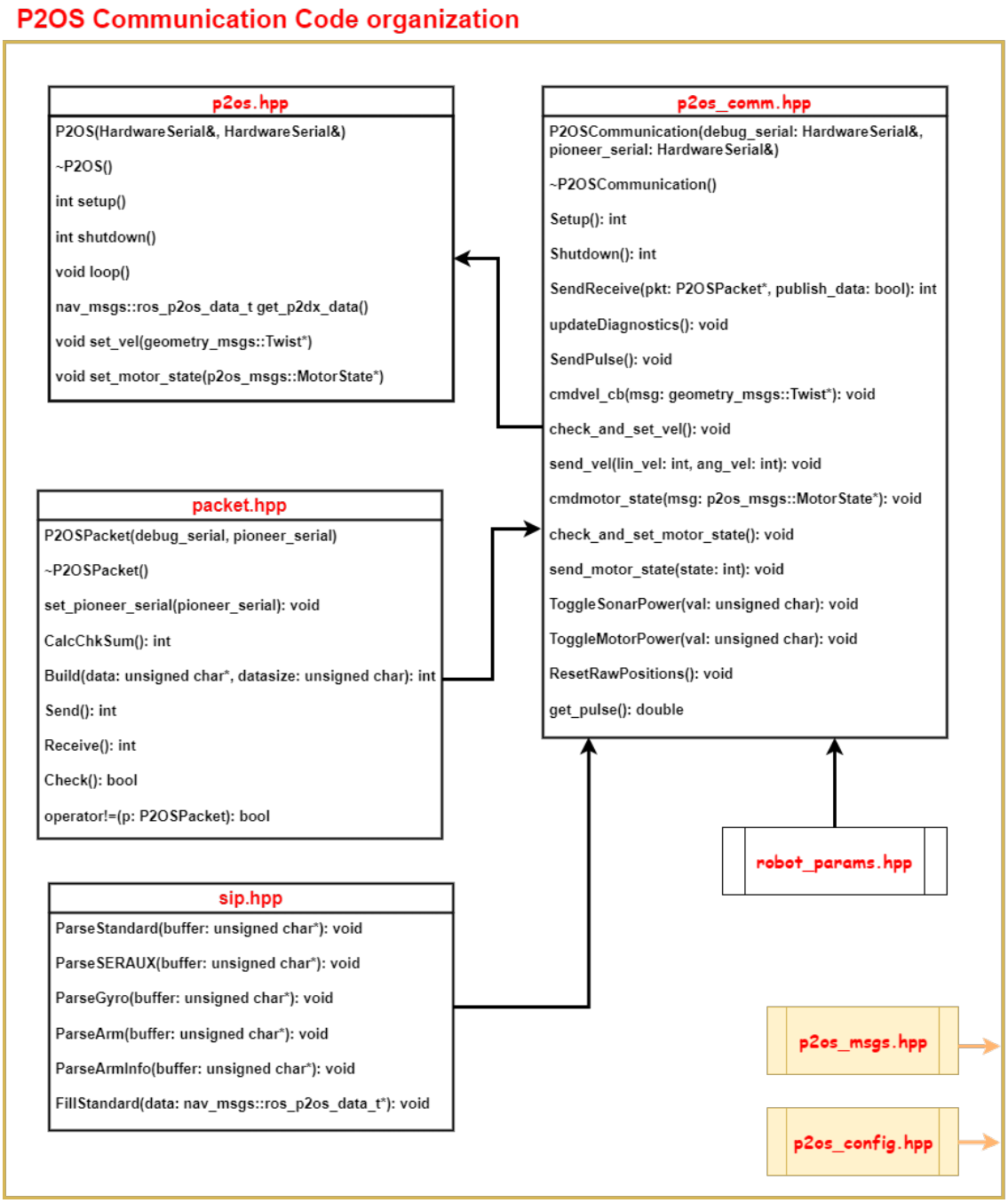


A.3 Components

Comment	Description	Designator	Footprint	LibRef	Quantity
smd 1uF	Aluminum Organic Polymer Capacitors smd 50volts 1uF	C1, C4	CAP_EEE1HA2R2SR	Cap. Elet. smd 1uF 50V	2
0.1uF 0805	capacitor 0805	C2, C3	CAPACITOR 0805	CAP 0805	2
1uF 0805	capacitor 0805	C5, C6, C7, C8, C9	CAPACITOR 0805	CAP 0805	5
DDS1	Conector KK 2.54mm 3 vias	CON1	KK_3vias_180º	KK_2.54_3vias	1
DDS2	Conector KK 2.54mm 3 vias	CON2	KK_3vias_180º	KK_2.54_3vias	1
DDS3	Conector KK 2.54mm 3 vias	CON3	KK_3vias_180º	KK_2.54_3vias	1
DDS4	Conector KK 2.54mm 3 vias	CON4	KK_3vias_180º	KK_2.54_3vias	1
DDS5	Conector KK 2.54mm 3 vias	CON5	KK_3vias_180º	KK_2.54_3vias	1
DDS6	Conector KK 2.54mm 3 vias	CON6	KK_3vias_180º	KK_2.54_3vias	1
DDS7	Conector KK 2.54mm 3 vias	CON7	KK_3vias_180º	KK_2.54_3vias	1
Motor1	holes 2.54mm 3 vias	CON8	3vias	2.54_3vias	1
Motor2	holes 2.54mm 3 vias	CON9	3vias	2.54_3vias	1
Encoder1	holes 2.54mm 5 vias	CON10	5vias	2.54_5vias	1
Encoder2	holes 2.54mm 5 vias	CON11	5vias	2.54_5vias	1
Suply	Conector KK 2.54mm 2 vias	CON12	KK_2VIAS_180º	KK_2.54_2vias	1
DDS8	Conector KK 2.54mm 3 vias	CON13	KK_3vias_180º	KK_2.54_3vias	1
SerialD2	holes 2.54mm 4 vias	CON14	4vias	2.54_4vias	1
SerialD0	holes 2.54mm 4 vias	CON15	4vias	2.54_4vias	1
SerialP2x	holes 2.54mm 4 vias	CON16	4vias	2.54_4vias	1
LED 3MM RED	LED 3MM RED	D1, D2, D3, D7	LED RED	LED 3MM RED	4
Diode M7	Diode, SMA, 1000V, 1A, 150°C	D4, D5, D6	DIOM505270X240	Diode M7	3
Led esp supply	LED 3MM RED	D8	LED RED	LED 3MM RED	1
Led state	LED 3MM RED	D9	LED RED	LED 3MM RED	1
Led TX	LED 3MM RED	D11	LED RED	LED 3MM RED	1
Led RX	LED 3MM RED	D12	LED RED	LED 3MM RED	1
DB9	Cable Grip 0.38 ~ 0.50 (9.5mm ~ 12.7mm) Aluminum - Silver	J1	*DB9_DB9(Primary)	DB9	1
LM317 Linear regulator - 5V	SOT223	LM1	SOT-223-2	lm317 smd	1
LM317 Linear regulator - 3V3	SOT223	LM2	SOT-223-2	lm317 smd	1
Trans BC817	Transistor BJT NPN BC817-25-7-F	Q1	SOT96P240X110-3N	Trans BC817	1
680R 0805	0805	R1, R2, R3, R9, R11, R12, R22, R23	RESISTOR 0805	RES 0805 5%	8
1k 1206	RES 1206 5%	R4	RESC3216X60N	RES 1206	1
3k 0805	0805	R5	RESISTOR 0805	RES 0805 5%	1
20k 0805	0805	R6, R10, R28, R33	RESISTOR 0805	RES 0805 5%	4
3k3 1206	RES 1206 5%	R7	RESC3216X60N	RES 1206	1
5k1 0805 5%	0805	R8	RESISTOR 0805	RES 0805 5%	1
0R	0603	R16, R18, R20, R29, R30, R31, R32, R34, R35, R36, R37	RESISTOR 0603	RES 0603 5%	11
2K2 1206	RES 1206 5%	R25, R26, R27	RESC3216X60N	RES 1206	3
dip switch 2 channel	DS02-254-2L-02BE	S1	SW_DS02-254-2L-02BE	dip switch 2 channel	1
BluePill	BluePill - STMDuino (STM32F103)	U1	BluePill	CMP-010-000000-1	1
MAX3232ID	3-V to 5.5-V multichannel RS-232 line driver & receiver 16-SOIC -40 to 85	U2	SOIC127P600X175-16N	MAX3232ID	1
Small case to esp32 d1 mini	ESP32 D1 Mini	U3	ESP32 D1 MINI	ESP32 D1 Mini	1

APPENDIX B – Pioneer 2DX Interface Code Organization

Figure 27 – Pioneer 2DX Interface - P2OS Communication Code Organization



Note: **p2os_msg.hpp** and **p2os_config.hpp** are used in all files.

Figure 28 – Pioneer 2DX Interface - Complete Code Organization