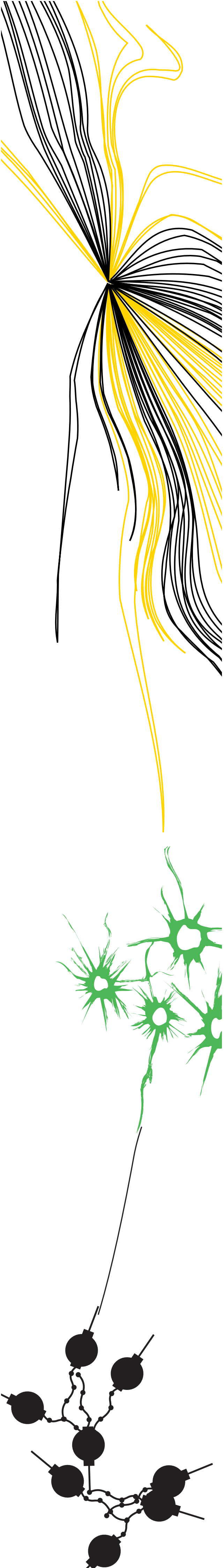


RAM ROBOTICS AND MECHATRONICS

ROS2 – XENOMAI4 REAL-TIME FRAMEWORK ON RASPBERRY PI

I. (Ilyas) Raoudi

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
dr. ir. G. van Oort
dr. ing. K.H. Chen

December, 2024

078RaM2024
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE

UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Summary

With the ever-growing robotics field, an increased need for experts in this field is needed. These future engineers have to be trained for research or development. The RaM group of the University of Twente is one of the institutes that train engineers.

Previously a framework has been developed to set up a real-time embedded system as this takes up a lot of time. However, new technologies and tools have been made available since the development of this framework making the old framework outdated. In this thesis a new framework has been developed to take on this role.

The framework developed in this project provides a firm real-time loop for computation and has a ROS-Xenomai-bridge to communicate between the soft and firm real-time parts of the system. Additional functionalities/ properties added to the framework are easy to use in education, flexible for different use cases, monitoring and logging on the Xenomai kernel.

An analysis was done using the system engineering approach to come to the design requirements. In the analysis, the target groups, use cases and the project constraints for the framework were analysed.

The design of the framework is split up into five components. The first component is the real-time loop which contains the firm real-time loop and the computation which has to be implemented by the users. The second component is the state machine which implements the coordination. The third component is the communication which consists of a bridge between ROS2 and Xenomai and a communication between Xenomai and the FPGA. The last component is the logging and monitoring which provide instrumentation for logging and monitoring of the Xenomai kernel.

The first test is a characterisation test to get the important characteristics of a real-time system using the framework. The results of the test show that the framework has achieved the requirements set for the communication overhead and jitter by at least 99 % of the time. Although there is some interference in the communications and execution, these do not hinder the working of the framework too much. There is a jitter build-up in the communication latency between ROS2 and Xenomai. This is caused by a 155 ns constant negative jitter in the ROS2 node.

The result of the first test showed a possible relation between the CPU usage and the performance of the framework. Therefore, a second test was done in which the first test was redone but with added stress to the CPU. The results show an overall decrease in the average execution time of all components but an increase in the maximum. In contrast, the sampling jitter of the firm real-time loop reacted negatively to the stress with a small decrease in the achieved deadlines.

The last test was an implementation test, showing the working of the framework by implementing one of the use cases. The results show that the framework works but this is only verified by the developer of the framework. To truly test whether the framework works, user testing has to be done with the different target groups of the framework.

Contents

1 Introduction **1**
1.1 Context 1
1.2 Design objectives 1
1.3 Plan of approach 3
1.4 Project constraints 3
1.5 Report structure 3

2 Analysis **4**
2.1 Target groups 4
2.2 Use cases 6
2.3 Project constraints 9
2.4 Requirements 10

3 Design **12**
3.1 Real-time loop 12
3.2 Communication 17
3.3 State machine 22
3.4 Logging & monitoring 25
3.5 Complete design 27
3.6 Coding style & file structure 28

4 Testing **31**
4.1 Characterisation test 31
4.2 Stress test 42
4.3 Implementation test 44

5 Conclusions and Recommendations **46**
5.1 Conclusions 46
5.2 Recommendations 46

A 5Cs approach **48**

B Missed cycles, method 2 **49**

C OPCODE commands IcoIo class **50**

D Instructions for implementation and use of XenoFrtLogger **51**

E Educators check list for student works **54**

F Instructions for implementation and use of XenoFrtRosIco or XenoFrt20Sim 55
F.1 Structure of the Framework 55
F.2 Use of the Framework 56

G Results individual components 60

References 63

1 Introduction

1.1 Context

The robotics field is ever-growing. Robots are increasingly becoming a common sight within all parts of life, seen and unseen. The robotics field encompasses a broad number of other fields including but not limited to medical, production, drones and self-driving cars. This growing need for robotic products is accompanied by an increasing need for experts in the field of robotics.

These future experts have to be trained to help research and develop new robotic systems. The University of Twente is one of the institutes that helps train and develop future engineers in robotics. Within the university, the Robotics and Mechatronics (RaM) research group does robotics research and contributes to education in robotics. For some of the courses and projects of the RaM group, students have to develop a real-time embedded system to control robots.

Setting up a real-time embedded system from scratch takes a lot of time and courses and projects have a time limit set. Therefore, a framework was developed (Meijer, 2021) to help students set up the real-time embedded system on a Raspberry Pi and FPGA setup. The setup provided for students is shown in Figure 1.1.

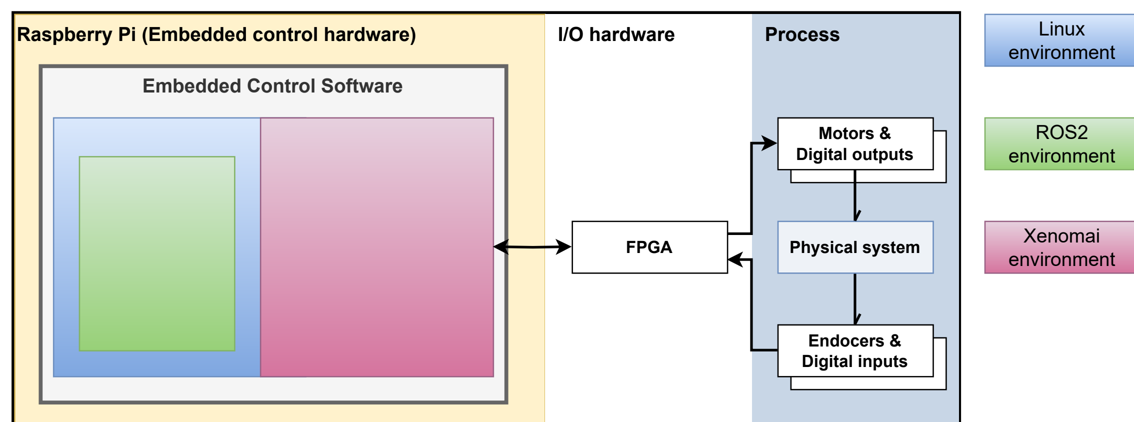


Figure 1.1: Diagram of the setup

The Raspberry Pi is responsible for running real-time embedded software. This is based on a Linux and Xenomai dual-kernel architecture. The Linux kernel runs the ROS2 (Macenski et al., 2022) framework. The FPGA is used as a device driver. The FPGA has been configured to be able to control four motors, four encoders, eight digital inputs and four digital outputs. This means that the embedded control system can at most control a robot with four degrees of freedom (DoF).

The developed framework has been in use for the last couple of years. During these years major software updates have been released for Linux, Xenomai, and ROS2 making the framework outdated. To keep up with the latest developments, steps have to be taken to create a new framework.

1.2 Design objectives

The goal of the project is to make a new framework. Figure 1.2 shows the placement of the framework within the kernel architecture. The placement shows that the framework has to take a similar role as the ROS2 framework, but on the firm real-time side within the real-time embedded system.

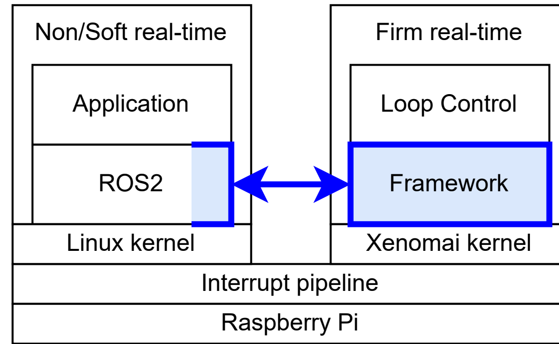


Figure 1.2: Diagram showing the framework's role within dual kernel architecture in blue.

The Linux kernel uses the ROS2 framework but it can only provide soft real-time capabilities. This is not sufficient for time-sensitive loop control in which control algorithms need a steady cycle time with at least firm real-time specifications.

The Xenomai kernel can guarantee a degree of firm real-time capabilities. Xenomai uses a dual kernel architecture (Xenomai 4 project, 2024c) alongside Linux to be able to execute real-time tasks with stringent requirements. Xenomai guarantees a higher degree of precision, which satisfies the firm real-time requirements needed for the use cases of this framework. The framework has to transform these firm real-time capabilities into an infrastructure in which the end users can easily implement their real-time projects. One of the most important functionality is the implementation of the firm real-time loop.

Figure 1.2 shows a division between the two kernels. Parts of the real-time embedded systems are implemented on both kernels following the embedded control software stack (Broenink and Ni, 2012). Therefore, the framework has to implement a communication infrastructure in which communication can be established with ROS2 from the firm real-time loop.

Sub design objectives for the framework are being able to provide logging and monitoring of the Xenomai kernel. The framework further has to be easily implemented in education as most of its target base and use cases are related to education. To deal with the different use cases, the framework also needs to be flexible.

Additional functionalities that the framework needs to provide are logging and monitoring of the Xenomai environment. The framework further has to be easily implemented in education as most of its target base and use cases are related to education. To deal with the different use cases, the framework also needs to be flexible.

To determine whether the framework meets the requirements for a project, it must be tested to determine its characteristics. The characteristics can help the end user to determine whether the framework is suitable. Some of the important characteristics of this framework are communication latency and real-timeness between ROS and Xenomai and the overhead of the real-time loop.

The design objectives of this project can be summarised as:

1. The creation of a framework that allows developers to implement a firm real-time task on Xenomai while being able to establish communication with ROS2.
   (a) Easy to use in education.
   (b) Flexible to use for different use cases.
   (c) Have an easy-to-use way to monitor and log data in the Xenomai kernel.
2. The characterisation of the framework concerning:
   (a) the communication latency between ROS and Xenomai
   (b) real-timeness of the communication between ROS and Xenomai.

(c) overhead of the firm real-time loop.

1.3 Plan of approach

The system engineering approach was used for the analysis in which target groups, use cases and project limitations were analysed to come to the requirements for the design. User testing has been done on an alpha version of the framework. The test was done with students of the course Advanced Software Development for Robotics. The final design was tested for its technical specifications. An implementation test was done to prove its working.

1.4 Project constraints

The project constraints are the boundaries for this project. Below a summary of the constraints is given.

1. The hardware setup consists of the Raspberry Pi 4 and the ico-board FPGA.
2. The software setup consists of the following:
   (a) Ubuntu version 22.04
   (b) Xenomai 4, kernel v5.15
   (c) ROS2 Humble
3. The configuration/code of the FPGA.

1.5 Report structure

The rest of the report is structured as follows:

- Chapter 2, Analysis, discusses the analysis of the target groups, use cases and, project constraints and ends with the requirements for the design.
- Chapter 3, Design, discusses the solution/designs to achieve the goals while taking into account the requirements.
- Chapter 4, Testing, discusses the tests that have been done, the results and their discussion.
- Chapter 5, Conclusion and Recommendations, discusses the conclusion of this project with regards to the goals and requirements and the recommendations for further research or projects.

2 Analysis

This chapter discusses the target groups, use cases and influence of the project constraints on the framework. These topics in combination with the goals result in the requirements for the design. This follows the system engineering approach (Blanchard and Fabrycky, 2011). User tests have been done with an early alpha version of the framework. The findings of this test are used to elaborate the findings of the analysis. In the following sections, the target groups will be discussed first. This is followed by the use cases and project constraints and ends with the requirements.

2.1 Target groups

From the context, we can determine the target group for the framework is educational institutions. Within educational institutions, there are four different sub-target groups. These sub-target groups each have their priorities for the framework. These target groups are stated below and each group will be discussed in the following sections. From this list target groups one to three are users of the framework and the last sub-target group is a developer. The last sub-section will discuss general points that are important regardless of the target group.

1. Learners
2. Educators
3. Researches
4. Framework developers

2.1.1 Learners

Learners consist of students who use the framework for a practical of a course will be the majority. This group will be the majority of the user base of the framework. For them, the following four points are important:

1. Ease of implementation
2. Ease of understanding
3. Fast use of the framework
4. 20-sim integration

Ease of implementation means that for learners there are no high programming-skill hurdles in trying to implement the framework. The reason for this is that the skill level of the learners varies as the study background of the learners varies. This does not mean that the implementation of the framework should be overly simplified, as a certain standard and basic understanding can be expected from the learners.

Ease of understanding is about the ease with which the learners understand what is happening in the code. When the code is too complicated to understand, the learners will just follow the instructions on how to implement and use the framework without learning about what is happening in the code. This is contradictory towards the goal of the learners to learn about real-time embedded systems.

Fast use of the framework is important for learners because it is not efficient for them if they have to take too long to use the framework. This is because all courses have a set amount of time the learners have to invest in the course. Considering this limit, the time the learners invest in the course should be used efficiently. Furthermore, if it takes more time to use the framework than to make it from scratch, then the framework would be useless.

The last point is 20-sim integration. 20-sim is a program that is part of the toolchain used by the learners during the practicals. Auto-generating code from a model is one of the main uses

of the 20-sim program. The learners use 20-sim to simulate the loop control and afterwards generate code for the loop control. This code will then be used in the framework. This needs to be facilitated in an effective manner.

2.1.2 Educators

The term 'educators' refers to teachers and teaching assistants (TAs). Educators are mostly not personally using the framework but they will help with and grade the learners' implementation, that uses the framework. For educators, the following points are important:

1. Tools being provided to check system validity
2. A clear boundary between framework and learners' implementation/code

To help educators with debugging the code, especially when learners come to the educators with questions or statements regarding whether they have done something wrong or the given embedded system is at fault, additional tools can provide help. By running these tools the educators can distinguish whether a part of the embedded system or the learners' code is at fault. An example is the communication with the FPGA. A small program is run to check whether the FPGA is responsive and also if the actuators and sensors are working.

To help educators grade, a clear indication/boundary must be established to determine which parts have been implemented by the learner and which parts by the framework. This makes it also easy for the learners to implement their solution as they can easily differentiate what is theirs and what has been given.

2.1.3 Robot developers

This group consists of people who will use the framework to develop a robot. This group will mainly consist of students who are doing their graduation thesis/project but can also include technicians or other employees at RaM. For them, the following points are important:

1. Fast use of the framework
2. Flexibility to implement for a wide range of use cases

For rapid prototyping, the framework can be used to reduce the time required for implementation. Therefore, if the framework takes too much time to be used, the developer would rather write their own code, which would make the framework useless.

There is a wide range of RaM projects that need a real-time system. To not limit the user's ability to use the framework for different kinds of projects, it has to be flexible. This entails that the framework of this project should be configurable to suit different robots.

2.1.4 Framework developers

Framework developers will continue the development of the framework after this project. For them the following point is important:

1. Modularity

Modularity is the most important point for future developers. For them, it would be inconvenient to have to rewrite multiple parts of the code again and again. Therefore, the software has to be written in a modular way to be able to reuse parts of the code. This has to be done logically, as some parts of the code should not be split, as this will increase the complexity without giving any advantages. Modularity and complexity must be balanced so as not to inconvenience future developers.

2.1.5 Common to all target groups

Some functionality is important regardless of the target group. These are:

1. Ease of use
2. Logging
3. Monitoring

Ease of use in the context of setting up an embedded control system is for the user to be able to focus on implementing the control algorithm. For the given setup, this means that the framework has to provide the communication infrastructure between ROS2 on the Linux kernel, a real-time program running on the Xenomai kernel, and the FPGA.

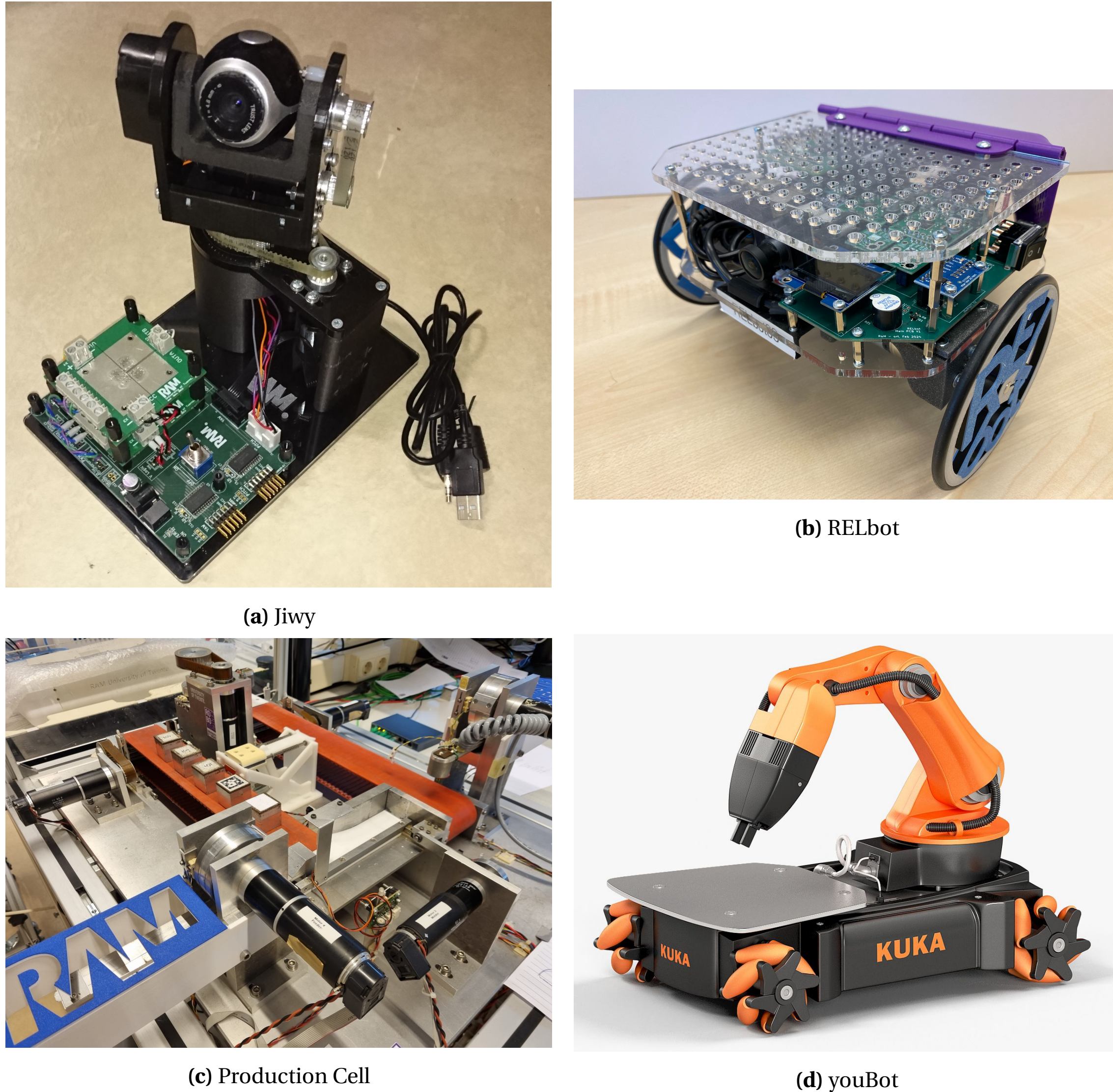
Logging is important to be able to get insight into the situation by analysing the gathered data after running the system. Furthermore, with big systems where a lot of data is monitored, the user cannot watch all data at the same time.

Monitoring is important for the users so that they can see changes to the data during run time. This gives the developer information about the current state of the robot. It may give the user time to intervene when the system makes a mistake or goes into an error. It also gives insight into why the error might have happened in the first place.

2.2 Use cases

There are already a few known use cases that will or might use the framework. These use cases are shown in Figure 2.1 with a list below with a short description of these use-cases.

1. **The Jiwy** is a robotic setup consisting of a camera that can pan and tilt. It is used as course material for the practical of the course Embedded System Laboratory.
2. **The RELbot** is a differential-drive wheeled robot on which new sensors can be added or removed. It is mostly used as course material for practicals of the courses of the master Robotics, most notably Software Development for Robotics (SDfR) and Advanced Software Development for Robotics (ASDfR).
3. **Producton Cell** is based on an injection moulding machine and uses multiple embedded systems to collaborative control the setup. It is used for demonstrations showing how to control the setup using different control-software design approaches. Implementing these different design approaches are thesis for master and bachelor students.
4. **youBot** is a robotic arm that has been placed on a mobile platform that has four omnidirectional wheels. Implementing a control-software design approach can be done for a master's or bachelor's thesis.



(a) Jiwy

(b) RELbot

(c) Production Cell

(d) youBot

Figure 2.1: Use cases

From these use cases, two different categories of robotic setups can be distinguished. These categories are stated below and in the following sections, each will be discussed. Afterwards, the general influence of real-time robotic systems on the framework is discussed, ending with an overview.

1. Robotic manipulators
2. Mobile robots

2.2.1 Robotic manipulators

Robotic manipulators are robots in which each actuator controls a DoF. An interesting point about robotic manipulators for the framework is how they will maintain their current pose. These can be divided into two types. The first type will automatically maintain its current pose, depending on the actuator itself or the mechanical design of the manipulator. When power to the actuators is shut off, it will still maintain its current pose. Examples of this are the Production Cell and the Jiwy shown in Figure 2.1a and d.

The second type does not automatically maintain its current pose and, when no power is given to the actuator, the robot will collapse because of gravity. This means that for these robots to

maintain their pose they need gravity compensation. To implement this, a real-time loop is needed. For the framework, this means that in all operational states of the robot, a real-time loop has to be implemented. An example of this is the robotic arm of the youBot as shown in Figure 2.1d.

Another consequence of not being able to maintain the current pose automatically is the need for a closing sequence. The reason for this is that when the robotic arm shuts down, the arm collapses. If this happens, it might damage the robotic arm or its environment. To prevent this, the framework should be able to implement a closing sequence. This sequence can then be used to bring the robotic arm to a state in which there is no chance of damage.

The last important point for the framework is the need for the robots to go to a ready state before being able to run their main functionality. This can mean different things for different robots. For the robotics setup like the youBot in Figure 2.1d this means first setting up the robotic arm before allowing movement from the mobile part of the robot. For robots like the Production Cell in Figure 2.1c, it means a homing sequence to be able to identify the position of all its actuators.

2.2.2 Mobile robots

Mobile robots are robots that can individually move through their environment. The aspect that must be considered with these types of robots is that no physical action should be taken when starting up. With mobile robots, there is no inherent limit to the robot's range of operation. This means accidents can happen like riding off the table or suddenly accelerating towards a physical obstacle. To prevent this, after starting up the programme, the user or higher-level system has to give the command to start the movement.

The argument can be made that this is true for all robotic setups, but with stationary robots, the movement limit is already known beforehand. The robot is also not capable of any movements outside its scope. This means that people can move outside its operating range before starting the program to avoid any possible danger. This is not the case with mobile robots.

2.2.3 Common for all use cases

For the real-time systems that are developed at RaM, the embedded control software stack (Broenink and Ni, 2012) is followed for the implementation of the software. The different components in the diagram have different needs for real-time depending on the application. For the use cases discussed, the soft real-time cycle frequency falls within 1 to 100 Hz. For the firm real-time this is between 100 to 10,000 Hz. The ROS2 framework provides a way to create a soft real-time loop on the Linux kernel. The framework of this project has to provide an easy way to create a firm real-time loop on the Xenomai kernel with an adjustable cycle frequency.

With the high cycle frequencies on the Xenomai side comes a lower cycle time. Within this cycle time, the system has to execute the control algorithm. To implement compute-heavy control algorithms, a low overhead is needed, leaving as much time available as possible for the control algorithm.

In Raoudi (2023) a limit was set for 3 % of the cycle time for the SPI communication to the FPGA. For the communication between ROS2 and Xenomai a limit of 1.5 % of the cycle time is chosen as the ROS2 communication does not involve any external communications to components outside of the CPU. This comes down to a limit of 4.5 % of the cycle time for the total communication overhead of the firm real-time loop.

The last point for all use cases is having a stable sampling frequency. This is because if the sampling frequency has fluctuations it causes the control-law parameters that contain the step size to become inaccurate. This leads to deviations in the control of the robot. Small deviations

are acceptable but many and larger deviations can lead to an unstable system. This means that the jitter of the sample time has to stay below 1% as motivated in Marwedel (2011).

2.2.4 Coordination of execution parts using a FSM

In the Section 2.2 multiple requirements for the states or interactions with the system have been discussed. These requirements all belong to the coordination aspect of the 5Cs approach (Klotzbücher et al., 2013). The coordination part of the loop control takes the form of a finite state machine.

The idea of implementing a state machine on the Xenomai side of the framework has taken inspiration from the work of Hoogendijk (2013). In his work, there is also a division caused by having a synchronous and non-synchronous part. States were added to the computation part to be able to control it from the non-synchronous coordination part (Hoogendijk, 2013, p. 50).

Combining all the requirements for the states results in Table 2.1.

State	Description
Idle	The system is not operational and can only go to the next state with instructions of a higher-level component.
Initialising	This state sets up the system to go to a ready state.
Run	This state for the main control algorithm to be executed.
Stopping	This state sets up the physical system to be powered off.

Table 2.1: Description of state for the state machine.

In Table 2.2 an overview is given of the use cases with respect to the needed states. This shows that the youBot is currently the most complex use case.

Use case / State	Initial	Initialising	Run	Stopping
Jiwy		x	x	
RELbot	x		x	
Production Cell		x	x	
youBot	x	x	x	x

Table 2.2: Overview use cases against the needed states

2.3 Project constraints

Of the project constraints, two pertain to the hardware and software setup (Figure 1.1) for which the framework has to be developed. Combining this with the embedded control software stack, results in Figure 2.2. The placement of the components of the embedded control software is based on their requirements for real-time. In this figure, the sequence controller can be in either the soft or firm real-time side.

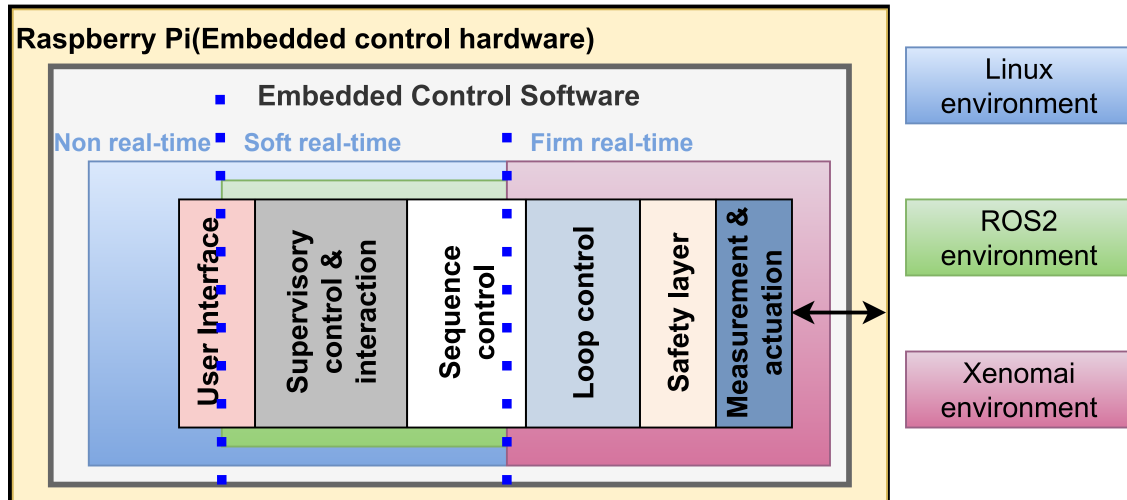


Figure 2.2: The embedded setup combining both the hardware and software diagrams.

As data has to flow between all adjacent components, a bridge has to be made to be able to communicate between ROS2 and Xenomai. Both sides have different real-time capabilities and as discussed in Section 2.2.3 both sides have different ranges of cycle frequencies. With Xenomai having generally a higher frequency, every ROS2 cycle will receive a packet that contains data of multiple Xenomai cycles. For the soft real-time part of the system, only the latest data is used. Therefore, the framework has to filter the data from the bridge on the ROS2 side to only include the latest data.

An exception to this requirement is data sent to ROS2 for logging from the Xenomai side. This is because all data has to be logged and data cannot be missing. Additionally, ROS2 has multiple internal and third-party tools to log and monitor data on and between the nodes. However, this is not an option due to the soft real-time limitations of the ROS2 side. This could lead to a loss of data, which would be unacceptable. Furthermore adding the log and monitor data to the communication will contaminate it will extra data not needed for control. Therefore, the logging of the data from the Xenomai kernel has to happen on the Xenomai kernel.

The division between Linux and Xenomai causes the communication latency to be of importance. The communication latency between ROS2 and Xenomai determines the reaction time of the soft real-time part of the system to events in the firm real-time part.

2.4 Requirements

The requirements are based on the findings of the previous sections and the design objective. Below is an overview of all the requirements for the design, which will be discussed in the next chapter. The MoSCoW method (Bradner, 1997) is used to prioritise the requirements. The prioritising is done based on how much influence the requirement has on the working of the framework.

Must

1. The framework must provide a firm real-time loop on the Xenomai kernel with a programmable cycle frequency.
2. Sample time should have a jitter smaller than 1 % of the cycle time. With the most used cycle time being 1 ms, this corresponds to a maximum of 10 μs of jitter.
3. The communication overhead for the real-time loop on the Xenomai kernel must be limited to 4.5 %. With the most used cycle time being 1 ms, this corresponds to a maximum of 45 μs of communication overhead.

4. The framework must contain the following functionality for the real-time loop:
   (a) A communication bridge between the ROS2 framework and the Xenomai kernel.
   (b) When in operation a real-time loop should always be running.
   (c) A state machine implementing the states in Table 2.1.
5. The framework must filter only the latest data on the ROS2 side of the bridge.
6. The framework must have the following properties:
   (a) Flexible to use in different use cases.
   (b) Fast to use and implement for the user.
7. The framework must have instrumentation to be tested on the following aspects:
   (a) Communication latency
   (b) Real-timeness of the communication
   (c) Overhead of the real-time loop

Should

1. The framework should contain the following functionality for the real-time loop:
   (a) One way to monitor the real-time loop of the Xenomai kernel.
   (b) One way to log data in the real-time loop of the Xenomai kernel.
2. The framework should have the following properties:
   (a) Modular
   (b) Easy to use for users
   (c) Easy to understand
3. The framework should have a well-defined boundary between its built-in and user-created components.
4. 20-sim integration into the framework.

Could

1. The framework could provide tools to help debug the system.

Will not

1. Will not have any changes to the hardware setup.
2. Will not have any changes to the FPGA code.

3 Design

Based on the requirements, a general design has been made of all the modules needed for the framework. This general design is shown in Figure 3.1. In the following sections, each module will be discussed. This is followed by the complete design and ending with the coding style and file structure.

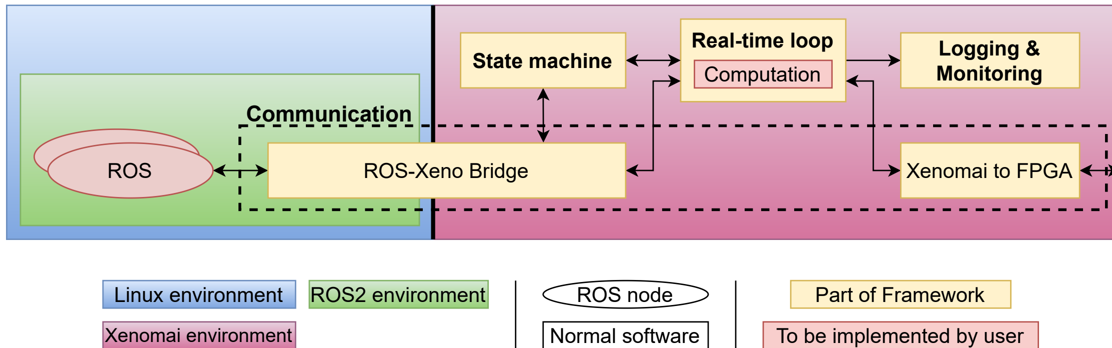


Figure 3.1: General design based on requirements

1. **Real-time loop**
   This module implements the firm real-time loop design and its structure in which the other modules are called. This part also includes the computation that the users have to implement.
2. **Communication**
   The communication consists of two modules:
   (a) **ROS-Xenomai Bridge**
   This part of the communication infrastructure is responsible for the communication between the user made ROS nodes and the Xenomai kernel.
   (b) **Xenomai to FPGA**
   This module implements the communication between the Xenomai kernel and the FPGA.
3. **State machine**
   This is the coordination part and implements the state machine and its control.
4. **Logging & Monitoring**
   The module represents both the logging and monitoring modules on the Xenomai side.

3.1 Real-time loop

The main component of the framework is the firm real-time loop. This loop functions as the basis of the framework in which all other components are called. The loop has an adjustable cycle time, which the user can set without altering the framework.

The loop can be implemented using various methods, each distinguished by how it handles missed deadlines. Four methods are shown in Figure 3.2. Below the figure, each method is described along with its respective advantages and disadvantages.

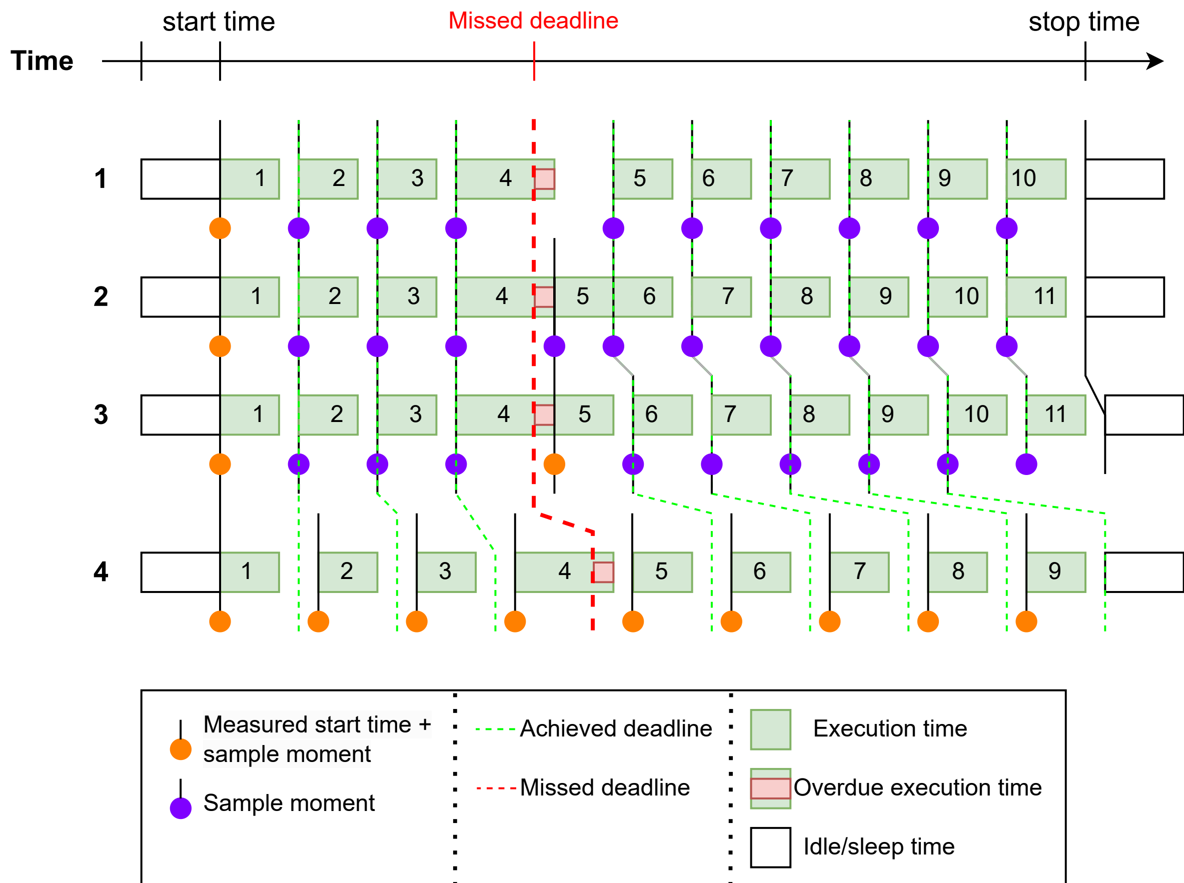


Figure 3.2: Diagram of different methods to resolve missed deadlines.

Method 1
In the first method, the cycle directly after the missed deadline will be skipped and nothing will be done during that cycle.

- **Advantage**
  - Easy synchronisation between multiple threads.
  - No long-term jitter build-up.
- **Disadvantages**
  - The loop controller expects each time step to be a certain duration. By not doing anything for one cycle the amount of change that the sensors pick up the next cycle is double which will cause the loop controller to think that the speed was double, which would be incorrect.
  - The program would do nothing for the rest of the cycle.

Method 2
The second method is to directly start the next cycle after the missed deadline while keeping the next deadline the same. This method wants to ensure that the cycle frequency stays constant. This method concurs that a missed deadline only happens as an anomaly. Appendix B shows through calculations the number of cycles needed to recover from missed deadlines based on the cycle time, the amount of time overdue, and the average execution time.

- **Advantages**
  - Has a constant cycle frequency.
  - Easy synchronisation between multiple threads.
  - Long-term predictable behaviour even with missed deadlines.

- **Disadvantages**
  - Reduces the time available for the next cycle.
  - The controller will use the wrong time step duration for all the cycles needed to compensate for the missed deadline.
  - All cycles after the missed deadline that are used to compensate will also miss their deadlines, see Appendix B.
  - Increased cycle-to-cycle jitter when a deadline is missed.

Method 3
The third method is to directly start the next cycle after the missed deadline while moving the start time to the current start time of the cycle. This method aims to maintain a consistent cycle time.

- **Advantage**
  - A constant cycle time
- **Disadvantages**
  - The cycle after the missed deadline has to measure a new start time.
  - With multiple missed deadlines the amount of lag will increase. This might be a problem when multiple real-time threads have to run synchronously.
  - The average cycle frequency of a run will never be correct to the inputted cycle frequency after a missed deadline.
  - In the same time as the other methods this method will not be able to run as many cycles.
  - Increased long-term jitter with each missed deadline.

Method 4
The fourth method involves measuring a new start time at the beginning of each cycle. This approach ensures that if a deadline is missed, it won't affect the following cycle, as each cycle calculates its deadline based on its own start time. This method aims to maintain a constant cycle time.

- **Advantage**
  - A constant cycle time
- **Disadvantages**
  - Each cycle has to measure a new start time.
  - Some lag will be slowly accumulated during the run as there is some time between each cycle. This will negatively affect the jitter. Compensation can be implemented but this will never be perfect.
  - The average cycle frequency of a run will never be equal to the given inputted cycle frequency.
  - In the same time as the other methods this method will not be able to run as many cycles.

Conclusion
For the choice of which method to implement, the first and fourth methods are directly dismissed because of their apparent disadvantages. From a technical perspective, it does not matter which solution is chosen between the second and third methods. The reason for this is, that the range used for the cycle time is between 100 and 10,000 Hz. With the most used cycle time being 1000 Hz. Take this in combination with the actuators used and the speeds they operate at, the amount of movement the actuators can make within a cycle is severely limited. This limitation means it does not matter which method is chosen.

The second method was chosen as this method is the most predictable in the long term. This is because there is only one moment in time when time is measured. From that moment there is a constant cycle time between each deadline. Even with a mist deadline the loop will eventually

fall back to the predictable behaviour after a few cycles and there is no long-term build-up of jitter. With the third method, the cycles will have shifted because of the missed deadlines. Furthermore, this method is the easiest to synchronise between multiple threads.

3.1.1 Loop structure

The real-time loop is built up of multiple components that will be called sequential in the same order. All these components together have to finish their execution within the cycle time. The structure of the loop has an impact on the behaviour and stability of the real-time system.

The implementation of the loop is split into two parts using C++ class inheritance. The `XenoFrtLoop` class handles the loop with deadline checks. The `XenoFrtRosIco` class inherits from the `XenoFrtLoop` class and overrides the `step` method to define the control loop. The user has to create a class that will inherit from the `XenoFrtRosIco`. The user-made class serves as the basis of the users' implementation as will be further discussed in this chapter. Figure 3.3 shows the UML class diagram of this structure.

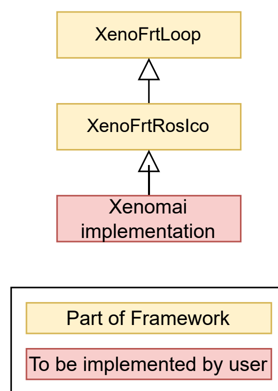


Figure 3.3: UML class diagram of `XenoFrtLoop` and `XenoFrtRosIco`.

Figure 3.4 shows the structure of the real-time loop with all the components. The diagram is split into two colours depending on in which class it is implemented. For each component in the control-loop step, an indication is given as to which aspect of the 5Cs they belong to (See Appendix A for a short description of 5Cs). Below the figure, a description of each component is given.

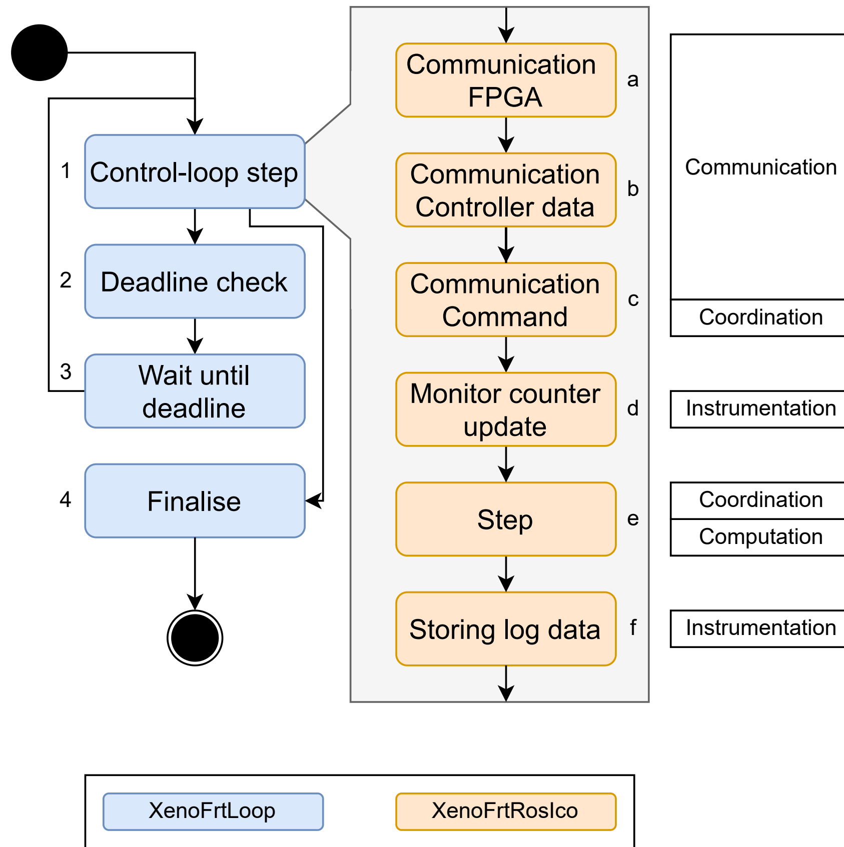


Figure 3.4: Diagram of the components making up the real-time loop.

1. **Control-loop step**
   The control loop step consists of the different steps needed to implement the control algorithm. Each of the current steps is explained next:
   (a) **Communication FPGA**
   The FPGA communication is responsible for communicating with the Ico Board and getting the sample data of the physical system. This is called first to ensure a stable sampling frequency and minimise any potential source of jitter.
   (b) **Communication Controller data**
   The data communication is responsible for sending and receiving the data from ROS. The data type of this is defined by the user. The received data is most likely the setpoints for the loop controllers and is used in the next components.
   (c) **Communication Command**
   The command communication is responsible for receiving commands for state changes and for sending the current state to the ROS2 side. It is placed third as it has to be called before the state machine step component as this might change the current state.
   (d) **Monitor update**
   This step updates the counter of the monitor object. When the `monitor.printf` function is called the counter determines whether monitor data can be sent or not.
   (e) **Step**
   The step calls the current state of the state machine as discussed in Section 3.3. The implementation of these states is done by the user and must include all the control algorithms. This step comes after all communication has been done to get the newest data for the control algorithms to do their calculations with. The state machine uses the return value of this step to determine the next state.

   (f) **Storing log data**
   This step calls the logger to log the registered data. This is the last step as this has to come after the state machine step where the registers have been updated.
2. **Deadline check**
   Check if the execution time did not surpass the deadline. If the deadline is surpassed, the counter for the deadline misses is increased.
3. **Wait until deadline**
   Sleep until the current deadline is passed.
4. **Finalise**
   This step is only reached if the current state of the state machine is quit. The firm real-time loop will stop running and the following variables will be printed on the terminal:
   (a) Runtime (seconds + nanoseconds)
   (b) Cycle count
   (c) Missed cycle count

3.1.2 Using 20-sim generated code

To integrate the 20-sim generated code, a new class has been made called `XenoFrt20Sim`. This class inherits from the `XenoFrtRosIco` class. The `XenoFrt20Sim` class automatically adjusts the cycle time of the framework to the generated code. The user can choose to use the base implementation of the firm real-time loop or the 20-sim variant.

The `XenoFrt20Sim` class uses the .xml file which is included with the generated code to get the cycle step size in seconds. This file contains different tokens that can be used to get information about the system. This information is gathered before the code is compiled and given to the compiler as a definition.

For the user to use the 20-sim variant, the class implementation has to inherit the `XenoFrt20Sim`. In this implementation, he can create an object of the controller given in the generated code of 20-sim. This controller class has a parent class called `Submodel20sim`. Figure 3.5 shows the UML class diagram of this structure.

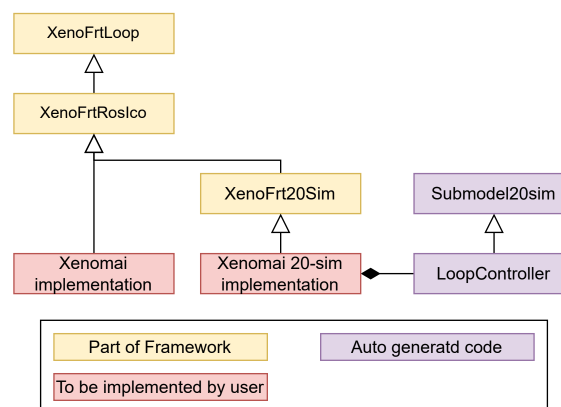


Figure 3.5: UML class diagram with both base and the 20-sim variant.

3.2 Communication

The framework has to facilitate the following communication infrastructure:

1. **Ros-Xenomai Bridge**: Between the user made ROS nodes on the Linux kernel and the firm real-time loop ruining on a thread on the Xenomai kernel.
2. **Xenomai to FPGA**: Between the firm real-time loop ruining on a thread on the Xenomai kernel and the FPGA.

In the following sections, these are further elaborated.

3.2.1 Ros-Xenomai Bridge

For the Ros-Xenomai Bridge two communication channels are needed. The first one is for the data transfer, for which the user determines the data type. The second channel is for controlling the state machine. The state machine has to receive commands from the users' ROS nodes and has to send the current state to the ROS nodes. The command channel is predefined and cannot be changed.

The distinction between the command and data channel is made to always be able to compile the code and control the state machine. If these channels were to be combined, it would mean that the user has to add pre-specified information to the combined channel. If this information was not added the framework would not work. Therefore, combining the two channels would create an extra point of failure. This would cause unnecessary complications for the users.

The communication between the Linux and Xenomai kernel must go through the cross buffer (X-buffer) (Xenomai 4 project, 2024a), see Figure 3.6.

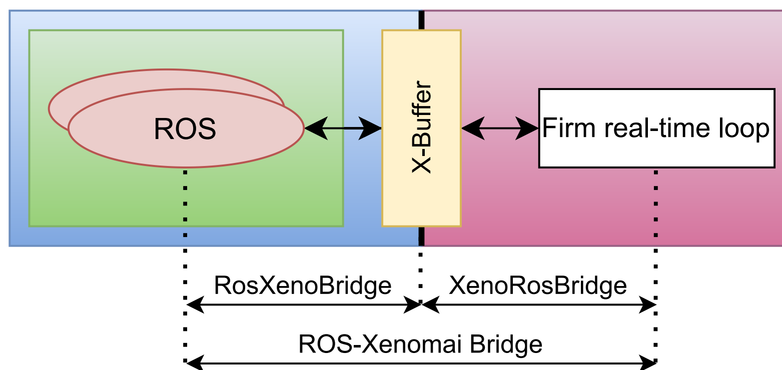


Figure 3.6: Components that need to be connected for the ROS Xenomai communication.

It is important to take into account the following points about the working of the cross buffers:

1. After the data is read once, it disappears from the buffer.
2. The cross buffer works like a first-in, first-out buffer.
3. After creation the size is fixed and cannot be changed.
4. When the buffer is full no data can be added.
5. Both the Linux and Xenomai kernel have to individually set the cross buffers to NON-BLOCK, to avoid blocking the program when trying to read an empty buffer or write to a full buffer from one side.
6. Data in the buffer cannot be overwritten.

RosXenoBridge

For the RosXenoBridge there are two options. Figure 3.7 visualises the two options. Below the figure, the two options are explained with their advantages and disadvantages.

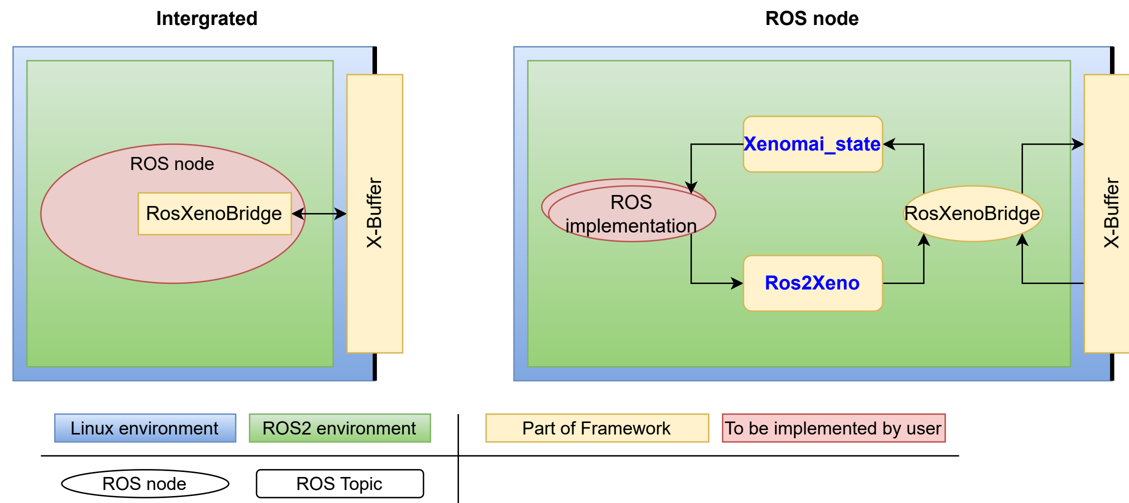


Figure 3.7: Two options for the RosXenoBridge implementation.

1. **Integrated**
   The first option is to let the users integrate the communication directly in their own ROS implementation. The framework would provide the functionality in the form of a library that has the functionality needed for easy integration.
   - **Advantages**:
     (a) The latency is lower.
     (b) More freedom to implement how the user wants it.
   - **Disadvantages**:
     (a) The end-user has to possibly drastically change their implementation to use the framework.
     (b) Only one node can have access to the data as the data disappears after having been read from the buffer. This node can publish the data to make it available to other nodes.
2. **Dedicated ROS2 Node**
   The second option is a dedicated ROS node to communicate between the cross buffers and the users' ROS nodes. The node uses topics to communicate with the users' ROS nodes. The node takes the data that is published on a specific topic and writes this data to the cross buffers. When data is available on the cross buffer, the node reads the data and publishes this data on another topic.
   - **Advantages**:
     (a) Data is directly available to all ROS nodes.
     (b) Uses the ROS method of communication.
     (c) Decreases the implementation time for the user.
   - **Disadvantages**:
     (a) Increased latency as there are more components between users' ROS nodes and cross buffer.

The choice was made to implement the second option as this increases the user-friendliness and decreases the implementation time for the user more in comparison with the harm that the increased latency could cause. Furthermore, if the latency is too big for a system, the users can always change their ROS nodes to implement the first option without having to change the Xenomai side of the framework.

XenoRosBridge

The second connection that has to be made is between the cross buffers and the firm real-time loop running on the Xenomai kernel. For this, the same equivalent solutions are available as with the RosXenobridge. The first option is to have the communication integrated into the firm real-time loop. This means that time will be taken to read and write data to the cross buffer.

The second solution is to have an intermediate in the form of a thread to handle the communication with the cross buffers. This option has one big drawback which is its resource usage. This can happen in two forms. The first is starting a new thread on the same core as the thread used for the firm real-time loop. This reduces the time available for the firm real-time loop. The second option is adding a dedicated Xenomai core for communication. This will reduce the resources for the Linux side, which means that fewer ROS nodes and topics can be run.

Both variants of the second choice are not viable, which makes the first solution the preferred choice.

Message between ROS and Xenomai

Between ROS2 and Xenomai, a message has to be defined for each direction. To define a message the following points have to be considered:

1. Message name
2. Message structure
3. Location of the definition/ implementation of the message in the file system

For the message, the decision has to be made to choose between having a single prepared message in which the user has to determine the message structure or leaving everything to the user to implement. Below, the advantages and disadvantages of each option are shown.

Single message

- **Advantages**:
  1. The user only has to define the message structure once for each direction.
  2. Setting up the communication channel for the message can be automated.
- **Disadvantages**:
  1. Less flexibility as the message name and location within the file system have been chosen beforehand.

User implemented message

- **Advantages**:
  1. Greater freedom and flexibility for the user to implement solutions as they see fit
- **Disadvantages**:
  1. Takes extra time to implement for the user.
  2. There are more points of failure.
  3. Both sides must have the same data structure or the user has to implement conversion between data types. If this goes wrong the communication will fail.

The choice was made to go with a single data type as that was the most user-friendly in terms of implementation speed and reliability. With a single data type for each direction, the user only has to change two types to set up the desired messages. Furthermore, having no data type adds a point of failure to the framework if the conversion is wrongly implemented.

The ROS custom messages were chosen for the data type as they can be used in both the ROS2 framework and this project framework. It can be imported as a library and can easily be adjusted by the user.

3.2.2 Xenomai to FPGA

The communication between the Xenomai and FPGA (Ico board) is fixed. The `IcoIo` class has been previously developed for Xenomai 3. For this project, there are no functional changes to this class. The only change is to separate the communication method from the implementation. This increases modularity by making it so that if in the future, for example, the user decides to switch from SPI to UART, this does not require changing the `IcoIo` class.

The `IcoIo` library works with different codes through which the FPGA can determine what data it has received and what to send back. These codes are called OPCODEs. The current class is made for the four PMOD connectors attached to the FPGA. Each PMOD connector has the physical connections for one motor, one encoder, one digital output and two digital inputs. The current list of OPCODEs is shown in Appendix C.

3.2.3 Implementation

In Figure 3.8, a visualisation of the complete communication infrastructure is shown. Below the figure, an explanation is given of the structure and functionality of the different parts.

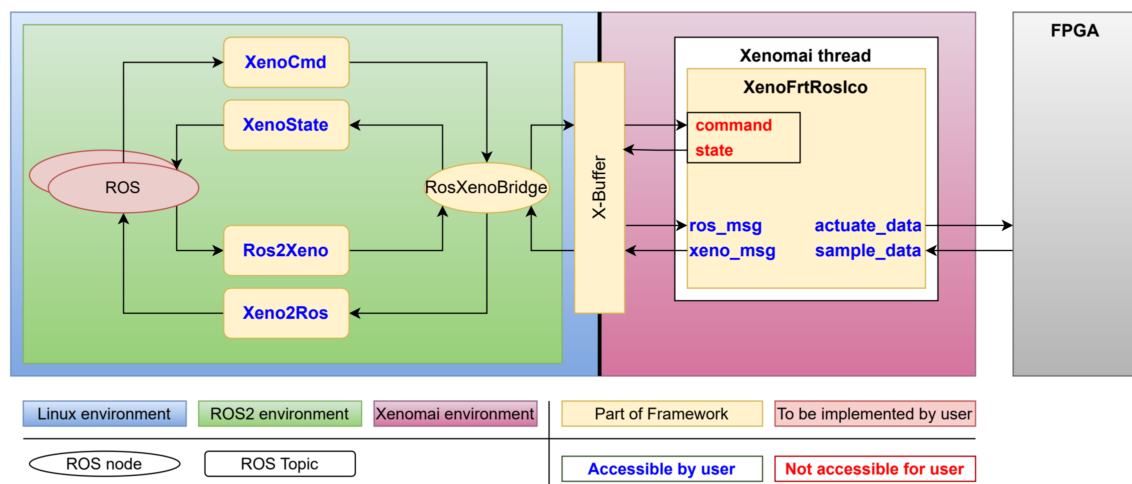


Figure 3.8: Overview of all communication within the framework

The ROS side of the communication infrastructure has four topics made available for the user. Through these topics, the communication between ROS and the cross buffers is facilitated. The topics and their use are as follows:

1. **XenoCmd**: The state-transition commands for the state machine of the firm real-time loop. Is of the type 'int32' (integer) of the ROS standards messages.
2. **XenoState**: The current state of the firm real-time loop. Is of the type 'int32' (integer) of the ROS standards messages.
3. **Ros2Xeno**: The user-defined communication channel to send data to the firm real-time loop.
4. **Xeno2Ros**: The user-defined communication channel to receive data from the firm real-time loop.

The Xenomai side has two classes for the communication with the ROS side. One for each channel. These classes are `XenoRosDataBridge` for the data channel and `XenoRosCommandBridge` for the command channel. The following points explain the implementation for both classes.

1. The framework has been designed to communicate with the cross buffer through two pre-selected variables. These variables function as a single-element buffer. The vari-

ables for the command channel are not accessible to the user. This was done through the inheritance structure by making them private instead of protected. These variables are called `command` for receiving commands from the ROS2 side and `state` for sending the current state to ROS2. The variables for the data channel are `ros_msg` for receiving data and `xeno_msg` for sending data to ROS. Both variables are made available through the `XenoFrtRosIco` class through the inheritance structure.

2. The classes will only begin with sending data when the first message has been received. This is to prevent the buffers from becoming full and preventing the Xenomai side from writing to them. This is because it may take a few seconds to start up the ROS side, in which multiple hundreds or thousands of cycles on the Xenomai side might run. If data is sent during this time, it might cause the cross buffer to become full which is undesirable.
3. The classes include a counter to write the values to ROS with the same cycle frequency as on which the ROS side runs. This will reduce the amount of unnecessary data that has to be communicated because the ROS side of the implementation generally runs at a lower frequency. Furthermore, this also reduces the chances that the cross buffer becomes full.

The communication with the FPGA happens through two variables that function as a single-element buffer. These variables are `sample_data` for incoming data and `actuate_data` for outgoing data. These two variables are updated every cycle with the latest data from the FPGA. These variables are made available through the `XenoFrtRosIco` class. The `IcoIo` class is responsible for handling the communication.

Figure 3.9 shows a UML class diagram structure of the framework including the communication classes.

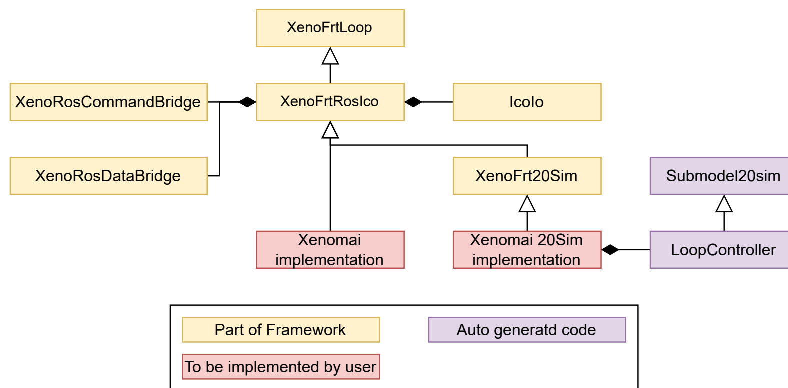


Figure 3.9: UML class diagram including the 20-sim variant.

3.3 State machine

For the coordination part of the 5Cs, a finite state machine has been chosen. The initial states have been determined in Table 2.1. For the design of the state machine, the required states have been implemented into the GAC life-cycle (Bezemer, 2013, Page 45) state machine. This was then extended with inspiration from the PackML state machine (OMAC - Organization for Machine Automation and Control, 2009).

The first point taken from the PackML state machine is that most states are split into waiting and acting states. This separation gives a clear distinction between when actions are still underway or finished.

The second point taken from the PackML state machine is two halting commands that can be given to stop the system but not go towards the stopped state. Being able to pause during the run state offers opportunities to configure the system without having to stop. These states have been combined into the pausing and pause states.

The resulting state machine is shown in Figure 3.10.

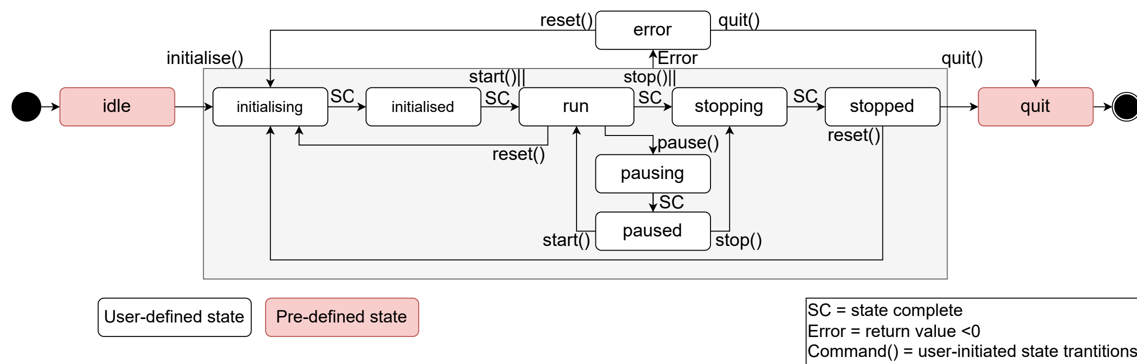


Figure 3.10: Diagram of state machine implemented.

The state transitions in this state machine can happen in two ways. The first is a command received through the command channel. If the conditions for the command are achieved, the state transition will go through. The second method is when a state is 'state complete' (SC). This means that the state has accomplished its task and is ready to transition to the next state.

This distinction is made so that the user can determine which of these options best suits the needs of the use case. An example is the transition between the initialised and run state. Some use cases might want to use a start button to give the start command others want to directly start after initialising.

If the user does not want to implement certain states, state complete can be used to go to the next state by only returning the value 1. This makes it easy for the user to skip unneeded states. This also avoids the need to use the commands if not wanted.

3.3.1 States

Below, a description for all states is given. The state descriptions are suggestions of the implementation for the states and the user can always deviate from it.

- **idle**
  The idle state is the start state. This state is already pre-implemented. In this state, the system is not operational and is waiting for the 'initialise' commando to be received.
- **initialising**
  The initialising state is for initialising the system and bringing the physical system to a ready pose. This can mean calibrating or homing the actuators or starting up other parts of the systems if needed. When this state is state complete it goes to the initialised state.
- **Initialised**
  The initialised state is for when the system is ready to execute its main operation. In this state, the physical system maintains its pose waiting for the 'start' command or when the state is state complete. It will then go to the run state.
- **run**
  The run state is where the system does its main operation which is implementing a control algorithm. When the 'stop' command is received or the state is state complete, the system goes to the stopping state. If the 'pause' command is received it goes to the pausing state. During the run state, the 'reset' command can be given to reset the physical system to a ready pose by going to the initialising state.
- **stopping**
  The stopping state is for bringing the physical system to a pose in which the system can

deactivate without damaging itself or the surroundings. When this pose is achieved it is state complete and goes to the stopped state.

- **stopped**
  The stopped state is for when the physical system maintains a pose in which the power can be shut down without damaging itself or the surroundings. The 'reset' command can be given to bring the physical system back to a ready state. If the 'quit' command is received the state will go to the quit state.
- **pausing**
  The pausing state is for bringing the physical system to a halt without damaging itself or the surroundings. When this state is achieved it is state complete and should go to the paused state.
- **paused**
  The paused state is when the physical system is stopped therefore maintaining its current pose but not ready to be deactivated without damaging itself or the surroundings. The system can resume its main operations when a 'start' command is received or stop when a 'stop' command is received and then go towards the stopping state.
- **error**
  In the error state, the physical system should directly stop any movement regardless of any possible damage to itself but do take into consideration the surroundings. This state can be reached by any user-defined state when it returns a value lower than zero at the end of its current cycle. To leave this state a command has to be received. This can be the 'reset' command to resume normal activity after re-initialising or the 'quit' command to complete stop operations.
- **quit**
  When the quit state is achieved the real-time loop will shut down. This state indicates that the system is done with all its operations. Afterwards, the following information is printed with respect towards the complete run:
  1. Total run time in seconds and nanoseconds
  2. Total cycle count
  3. Amount of missed cycles

3.3.2 State transition commands

Below the description of all commands as shown in Figure 3.10 is given. These commands can only be sent through the ROS command channel. See Section 3.2.3 on how to give the commands to the state machine.

1. **initialise**
   Go from idle to initialising state. Before any movement can happen this command has to be given by a higher system. This is to comply with the starting requirement of having a higher system give the okay before the physical movement of the robot can happen.
2. **start**
   Go from the initialised or paused state to the run state.
3. **stop**
   Go from the run or paused state to the stopping state.
4. **reset**
   Go from the run, stop, error or pause state to the initialising state.
5. **pause**
   Go from the run state to the pausing state.
6. **quit**
   Go from the error or stopped state to the quit state. After this command is called, the Xenomai side will shut down.

3.3.3 Implementation

For the implementation of the state machine, the choice had to be made between the conventional way that is widely used and an unconventional method.

1. The conventional method uses classes for both the state machine and the states.
   Advantages
   - This keeps the code clear and organised.

   Disadvantages
   - Have many .cpp and .hpp files, as each state needs one. It is possible to combine multiple states within one file but this is contradictory with the goals of education in which the coding style has to be followed.
   - The many files and their underlying structure might make it daunting for students without programming backgrounds.
2. The unconventional method is to implement the states in functions and keep the state machine as a class. These functions can then be given to the state machine class through a `bind` function.
   Advantages
   - Faster implementation speed for users as all functions/states can be defined in one class and then given to the state machine class.
   - By using the inheritance structure binding the functions and state machine can be done beforehand for the user and the user only has to implement the states. This increases ease of use.

   Disadvantages
   - Is unconventional.

The choice was made to go with the unconventional method as the implementation speed and ease of use are of greater importance than being organised.

The state machine is implemented in the `FullStateMachine` class. The states/ functions are implemented in the `XenoFrtRosIco` class and can be implemented in any child class of it. The UML class diagram is shown in Figure 3.11.

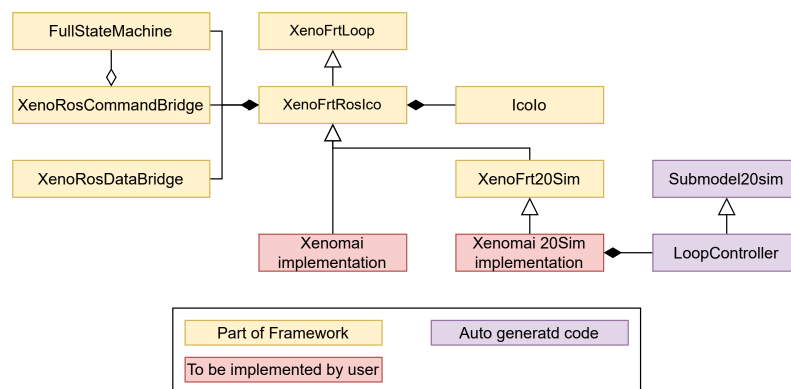


Figure 3.11: UML class diagram including the state machine.

3.4 Logging & monitoring

3.4.1 Logging

For logging of the data residing in the Xenomai kernel, there are two decisions to be made. The first decision is to decide when the data to be stored is sent. The second decision is whether the data is stored in a text- or binary-based file type. The decisions for both are connected. Depending on when the data is stored, different file types are preferred. Below, two moments for storage are discussed with their advantages and disadvantages:

1. **Stored when operational**
   When the data is stored when the system is operational, each cycle, time will be taken to send the data to be stored. On the Xenomai kernel data cannot be directly stored and has to go through a proxy (Xenomai 4 project, 2024b). This proxy will pass on the data to the file on the Linux kernel.
   Advantages
   - Even if the program or system crashes, all the previously logged data is retained. Some data that at the moment of the crash is located in the proxy might be lost.
   - The amount of data that can be logged is only limited by the available memory space.

   Disadvantage
   - Have to reserve more time to log data in the firm real-time loop.
2. **Stored when non-operational**
   When stored after operation the data will be stored in a buffer until the system is non-operational. The system will then write all the data in the buffer to a file.
   Advantage
   - The variables to be logged only have to be copied to a buffer, this is really fast and therefore uses little time in the firm real-time loop.

   Disadvantages
   - The amount of data that can be logged is limited by the amount set in the code. No variable buffer sizes can be used as this will compromise the real-timeness of the system.
   - Takes up more compute memory from the computer when operational as the space needed for the buffer has to be reserved.
   - If the program or system crashes when operational, all data to be logged is lost.

The decision is made to go with logging when operational. The reason for this is that when a crash occurs, the data is still saved. This data can then help to possibly determine the fault or error in the system. Furthermore, the amount of data that can be logged is not limited by the program as with logging when non-operational.

For the file type to use there are two types to choose from. The first is text-based file types like '.csv'. The second type is the binary file type '.bin'. Below, the advantages and disadvantages are stated for both file types.

1. **Text-based file types**
   Advantages
   - Directly usable in other programs.
   - File is easily human-readable.

   Disadvantages
   - Take processor time to format data.
   - Bigger file size.
2. **Binary based file type**
   Advantages
   - Fast logging as data only has to be copied from place A to B.

   Disadvantages
   - The file is not easily human-readable.
   - Needs info or header to determine data types used. If this is not kept decoding the file later might be difficult.

The choice was made to go with the binary-based file type as formatting data takes a lot of processing time for the processor. As the requirements state, we need to minimise the amount of overhead to leave as much time for the main calculation. Post-processing can take the '.bin' to a usable file type for other programs if needed.

To help the users, the post-processing/decoder program has been included with the framework. To make this possible, the logger has to add a header at the start of the log file to make it known to the decoder what kind of variable types are used. Furthermore, to help the user remember which variables were logged, the header includes the names of the variables. An example of the header is shown in Figure 3.12. Line 1 contains all the names and line 2 is the identifier of the variables.

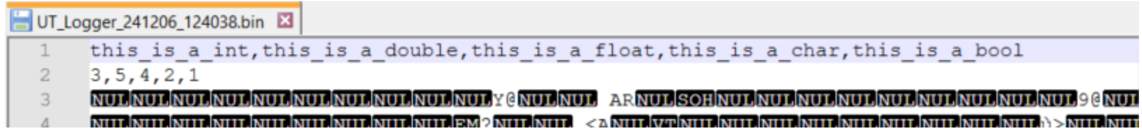


Figure 3.12: Header of a log file

To make the logger more user-friendly, a start and stop mechanism is added to the logger, to control when data is logged. This has the following advantages:

1. Some users would not like to log data in all states, as data from these states might be irrelevant.
2. The total amount of data gathered is reduced. This is beneficial for the data analysis as data from states that are irrelevant are not included.
3. In some states, the system has to wait for a signal from the ROS2 side. During this time a lot of useless data would be gathered. By stopping the logger this can be avoided.
4. The user might only want to gather data or a certain amount of cycles.

The logger is implemented in the `XenoFrtLogger` class. Appendix D gives instructions on how to implement and use the logger.

3.4.2 Monitoring

For direct monitoring on the Xenomai kernel, the options are limited to the Xenomai `evl_printf` function. There are no other viable options to monitor programs on the Xenomai kernel.

To make monitoring with the `evl_printf` statement more user-friendly, a configurable monitoring frequency can be set. This frequency dictates how often the `monitor.printf` function prints the specified statement, adjusting the output rate to the nearest divisor of the cycle time. For example, if the monitoring frequency is set to 30 Hz and the cycle time is 1000 Hz, the function will print the statement every 33 cycles.

If the monitoring frequency is set to 0 Hz, no statements will be printed. This feature allows users to deactivate monitoring with a single code adjustment, eliminating the need to manually remove all `monitor.printf` statements. This approach saves development time and frees up computational resources for critical calculations, as `printf` statements can introduce delays, especially when they involve data conversion to text.

The monitor has been implemented in the `XenoFrtMonitor` class.

3.5 Complete design

Figure 3.13 shows the complete object diagram of the framework. This shows the inheritance structure for the implementation of the framework. The user can choose between two types of implementation. A normal implementation using the `XenoFrtRosIco` class or a 20-sim implementation using the `XenoFrt20Sim` class.

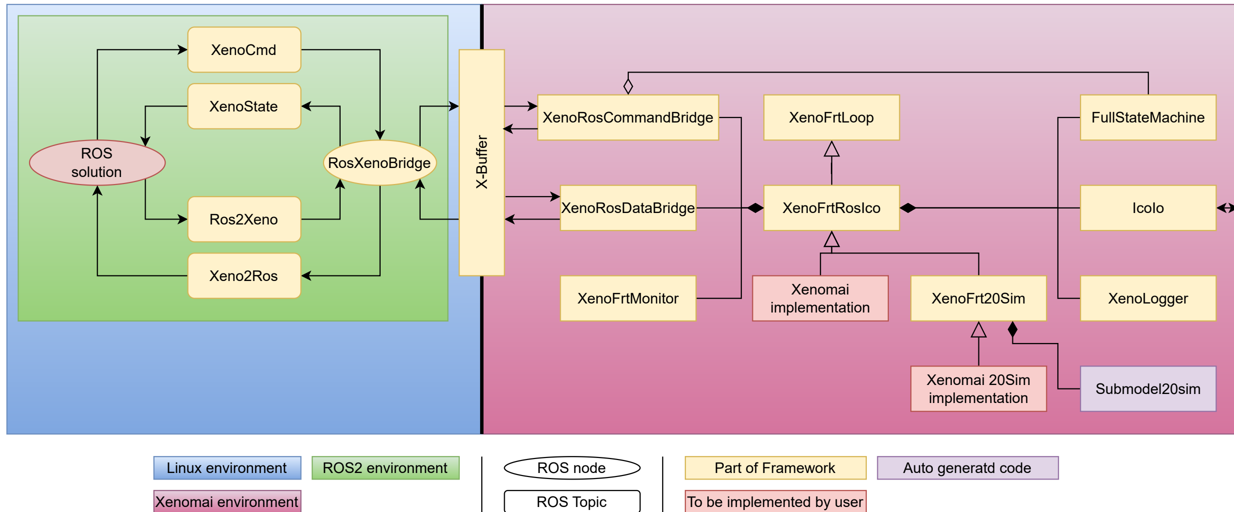


Figure 3.13: Complete design of the code for the whole setup on the Raspberry Pi.

3.6 Coding style & file structure

3.6.1 Coding style

The code style of this project is based on the coding style of ROS (ROS2, 2024a) with a few exceptions. The exceptions are shown in the list below:

1. Names of classes use UpperCamelCase.
2. Names of functions use lowerCamelCase.
3. Names of variables use snake_case.
4. All classes that make use of the Xenomai library start with 'Xeno'.
5. All classes that need a firm real-time loop to function use 'Frt' in their name.
6. Communication classes will be split into communication interface and communication protocol classes.

Points 1 to 5 help make the code easier to understand by being able to look at a name and knowing what kind of code element it is.

Point 6 helps with modularity as now any communication protocol can be swapped with each other. An example would be being able to change the SPI communication with the Ico-board with UART without having to change the interface of the data. Point six further helps with increasing the implementation speed as when another type of communication protocol has to be used not everything has to change or be implemented again.

3.6.2 File structure

The file structure can be split into two parts. The first is the structure of the framework and the second is the structure of how to use the framework. In the following sections, the design for both will be discussed.

Framework

The framework repository contains two folders. The first folder contains the framework and is named 'xenomai4-framework'. The second folder contains templates for the framework implementations and is called 'template'. Figure 3.14 shows the structure for the repository.

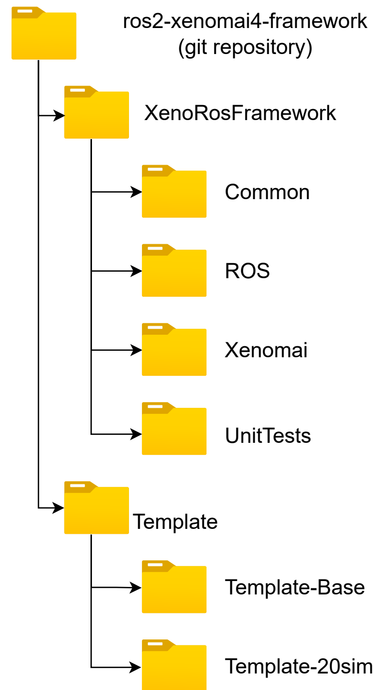


Figure 3.14: File structure framework repository

The file structure of the framework (xenomai4-framework) contains all the core files needed to set up the framework. The amount of different components/files it contains causes the need for a logically organised structure. This is to prevent confusion by future developers who are going to develop further based on this structure. To do this the files are separated into four categories, which are stated below.

1. **Common**
   This category contains files that are used in both ROS and Xenomai or instrumentation. The only files that the end-user has to change within the framework fall within this category. These are the custom message types `Ros2Xeno` and `Xeno2Ros`.
   This folder also contains the files needed for debugging the framework with debug messages. These debug messages are normally deactivated. They can be activated by changing the debug value from 0 to 1 in the 'debug_settings.h' file. These use precompile options to activate, this means that the code has to be re-compiled after each change.
2. **ROS**
   This category contains any file meant for compiling ROS-related programs, including nodes and messages. The files in this category can on their own be compiled into a program.
3. **Xenomai**
   This category contains all files related to classes that have to be implemented on a Xenomai kernel or the firm real-time thread. This category does not contain any files that on their own can be compiled into a program.
4. **UnitTests**
   This category contains different small programs for unit testing. Through these unit tests the end-user can determine whether the framework or their implementation is at fault when running into an error. Below, the currently available unit tests are given with the class or classes for which they are meant:
   (a) IcoIo - IcoIo & XenoSpiControllerHandler
   (b) Logger - XenoFrtLogger

(c) Monitor - XenoFrtMonitor

The template folder contains templates that can help the user with their coding. The templates on their own can be compiled. As of writing this report, there are two templates. One base template and one 20-sim variant. The templates have the following advantages:

1. Increases implementation speed, as the initial work is done.
2. Makes it easier to implement the framework, as more complex parts have been implemented.
3. Increase understanding of the framework as the templates can be seen as an example.
4. As the templates can be compiled they can be seen as a unit test for the state machines.

To minimise the chance of students just copying and using the framework/templates without trying to understand the code, some precautions have been taken. The educators can also use these to check whether the student truly has gone through the code and made smart and thoughtful decisions. These precautions are:

1. Weird naming of variables within the template classes. Appendix E gives a list of points for educators to check on when grading students.
2. All states have a simple implementation.
3. Names of the topic variables `Ros2Xeno` and `Xeno2Ros` are set to 'example_' in combination with 'a' to 'd'.

Usage of framework

The structure for using the framework follows the ROS2 approach (ROS2, 2024b). This file structure is chosen for the following reasons:

1. Clear separation between framework and implementation.
2. The code can be compiled with one simple commando `colcon build`. This builds everything in the folders be it ROS-related or not.
3. To skip building a component only an empty file called 'COLCON_IGNORE' has to be added. This tells the compiler to ignore the folder.
4. Projects can easily be added by adding them to the 'src' folder, see Figure 3.15.

Figure 3.15 shows an example of the file structure of an implementation. In this example, next to the framework, two projects are added. These projects are a ROS package and a base template that is provided. In Appendix F, instructions are given for implementing the framework, compiling and running the code.

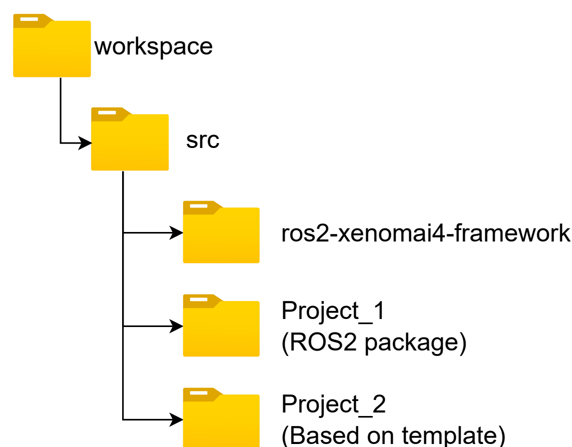


Figure 3.15: File structure for using the framework.

4 Testing

This chapter is about testing the framework. The following three tests have been done:

1. Characterisation test
   (a) Timing test
   (b) Communication test
2. Stress test
3. Implementation test

The first test gives insight into the framework's characteristics/specifications. Based on the results of the first test, the second test was conducted. The last test was done to prove the working of the framework.

All data shown in a plot is from one variable. For the line plots, the data points are coloured black, and to show chronology, orange lines are drawn between each subsequent point. Differences in colour within the figures result from the density of data points within that region.

4.1 Characterisation test

The characterisation test is implemented to get the important characteristics of the real-time system using the framework. These characteristics are:

1. Overhead of the firm real-time loop
2. Sampling jitter
3. Communication latency
4. Real-timeness of the communication

All the data on these characteristics are gathered in one test using one setup.

The reason why these tests are combined is that to get a representative value for the communication overhead, actual communication has to happen. This would mean that a test node had to be made to write data to the Xenomai kernel. This is the same for the latency and real-timeness tests. Therefore, it would be inefficient to not combine these tests.

In the following sections, the following points are discussed:

1. Test setup: This section discusses how data is gathered.
2. Timings test: This section shows and discusses the results of the overhead and jitter of the real-time loop.
3. Latency test: This section shows and discusses the results of latency and real-timeness of the communication between ROS and Xenomai

4.1.1 Test setup

To determine the overhead and the jitter of the firm real-time loop between each step in the firm real-time loop a timestamp is taken. Figure 4.1 shows the moments in the real-time loop when timestamps are taken. By taking the delta between each timestamp, the duration of each step can be determined.

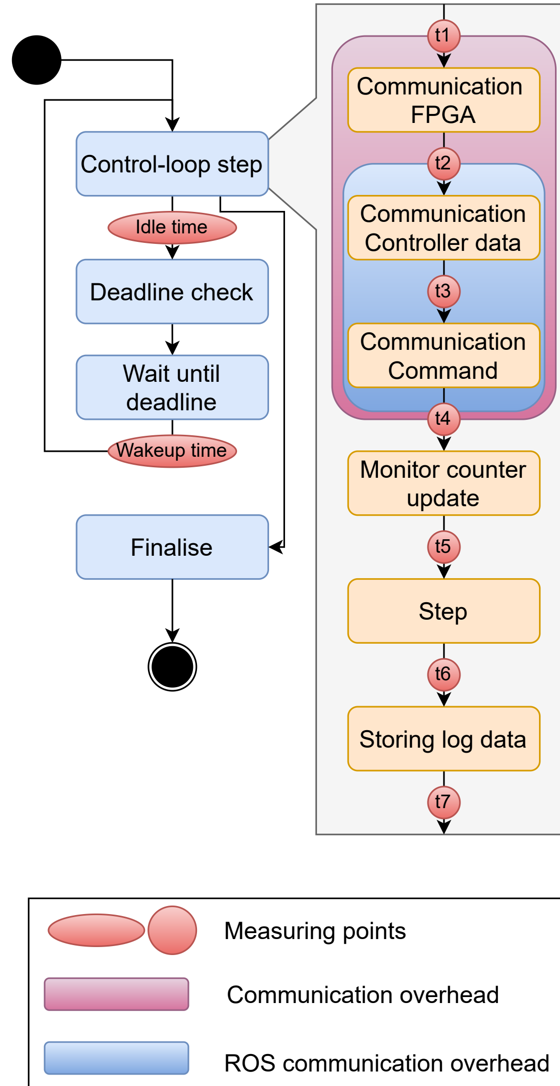


Figure 4.1: Timing test diagram

To determine the jitter for each time step (k), the formula below is used:

$$\text{jitter} = (t_{1,k+1} - t_{1,k}) - \text{cycle_time}$$

To determine the communication related characteristics messages are sent between ROS2 and Xenomai. One test node is made to be both the sender and receiver on the ROS2 side. The messages contain a timestamp from when they were and a sequence number that increments with each cycle. When the messages are received another timestamp is taken.

Figure 4.2 shows the setup. The setup shows the communication route between the test node and the Xenomai thread, where the framework is run.

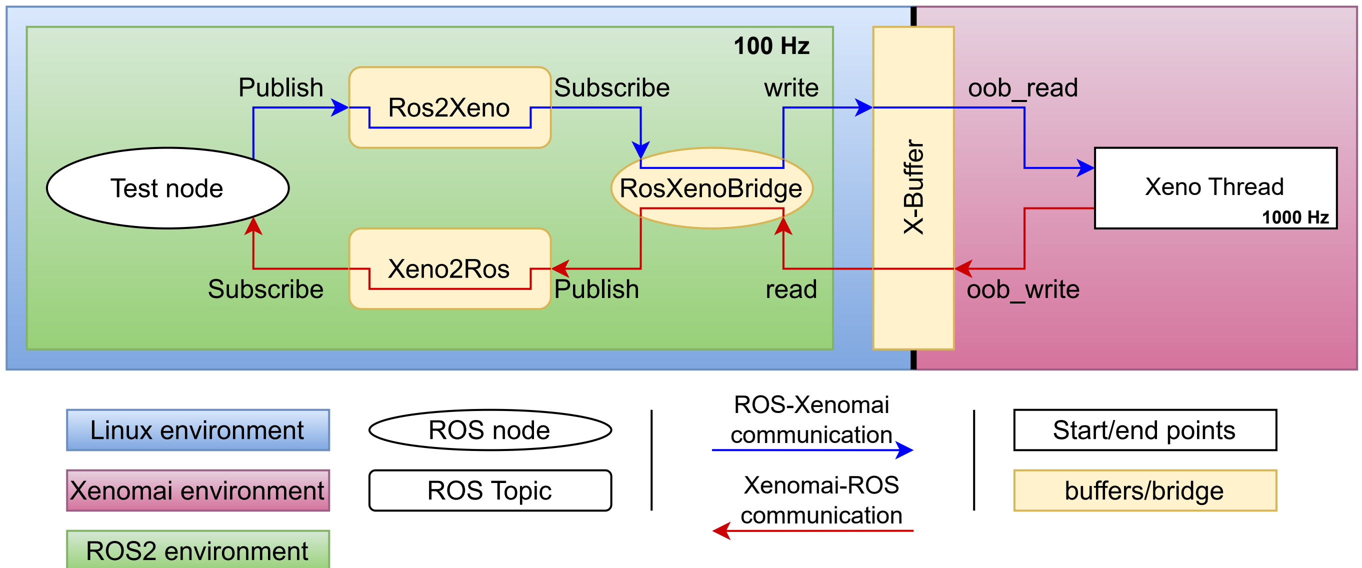


Figure 4.2: Test setup for characterisation

The communication latency is determined by calculating the time difference between the timestamps contained in the messages and the timestamp of when the messages are received.

The real-timeness of the communication is determined based on two approaches. The first approach checks the continuity of the numbers sent with the messages, which should increase linearly. Any lapses in the series indicate message loss and the point at which it occurred.

The second approach is checking the time intervals between when each message was received. This should remain consistent throughout the test. Any substantial increase in these intervals suggests a delay in the communication pipeline.

The test is run for 1 million cycles at a cycle frequency of 1000 Hz on the Xenomai side. This is equal to 1000 seconds. This results in only 999999 samples of data for the Xenomai side. The reduction of data is because of how part of a data set for a cycle is saved in the next cycle. The results on the Xenomai side are logged by using the logger as described in Section 3.4.1. The logged data is then decoded by using the decoder written in Python. Because of project-time constraints, the decoder is made using OpenAI ChatGPT (OpenAI, 2024).

The ROS side runs at 100 Hz which equals to 100,000 cycles and should result in 100,000 data points if all messages are received. The ROS side logs the data to a .csv file.

4.1.2 Results

In this section, the results of the test will be shown and discussed. To clearly show the findings, the results in this section are split up into the following subsections:

1. Overall statistics overhead and jitter
2. Total overhead
3. Individual components
4. ROS to Xenomai latency
5. ROS to Xenomai real-timeness
6. Xenomai to ROS latency
7. Xenomai to ROS real-timeness

Overall statistics overhead and jitter

The data gathered from the overhead and jitter from the firm real-time loop was analysed and the most important statistics are shown in Table 4.1. The 'total overhead' is all overhead combined. 'Communication' is the sum of all communication and the 'Communication ROS' is the sum of the command and data communication.

	Mean (μs)	**Median (μs)**	**SD (μs)**	**Min (μs)**	**Max (μs)**	**DL (μs)**	**On time (%)**
Total overhead	35.40	34.68	3.96	16.06	68.81	-	-
Communication	31.08	30.24	3.64	14.57	62.20	45	99.94
Comm FPGA	24.53	24.15	2.43	12.63	51.98	30	99.18
Comm ROS	6.54	5.91	2.00	1.89	19.67	15	99.99
Comm Command	3.41	3.11	1.07	0.96	12.93	-	-
Comm Data	3.13	2.81	0.96	0.89	7.72	-	-
Monitor Update	0.19	0.20	0.03	0.06	2.28	-	-
Storing log data	3.92	4.02	0.46	1.19	9.45	-	-
Jitter sample Freq	0.00	0.002	0.44	-18.81	15.30	10	99.99

Table 4.1: Statistics timing test (SD = standard deviation, DL = deadline, Comm = Communication)

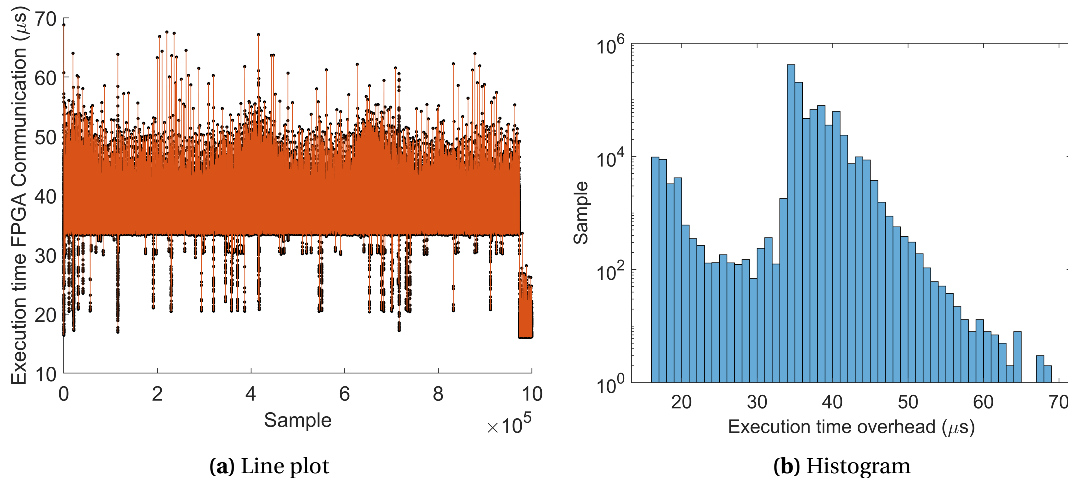
Table 4.1 shows that all the deadline requirements as stated in Section 2.4 have been met with at least 99 %. This is good enough to be seen as having achieved the requirements for the communication and jitter.

Through calculation shown in Equation 4.1, it can be determined that should the FPGA communication have no interruptions, it had to take around 3.1 μs. Currently, the average execution time for FPGA communication is 24.53 μs. This means that currently the SPI communication does not happen optimally and might have room for improvement.

$$\text{Communication Latency} = \frac{\text{Total Bit Count}}{\text{Baud Rate}} = \frac{96\,\text{bits}}{31 \times 10^6\,\text{MHz}} \approx 3.1\,\mu\text{s} \tag{4.1}$$

Total overhead

The results of the test for the total overhead are shown in Figure 4.3. Figure 4.3a shows the total overhead for each subsequent cycle. Figure 4.3b shows the histogram of the total overhead.



(a) Line plot

(b) Histogram

Figure 4.3: Execution time for overhead

As shown in Figure 4.3a the graph suddenly goes down at multiple points. At the end of the graph, it even stays down for a considerable time. The hypothesis was made that high CPU usage causes increased performance. To prove this a new test had to be done. The results of this are shown in Section 4.2.

Individual components

In Figure 4.4, a zoomed-in version of the line plots of a few of the individual components is shown. A zoomed-in version is shown because showing the whole plot would be pointless as there are too many data points to differentiate anything of importance. In Figure 4.5, the histograms of the individual components are shown. In Appendix G the line plots and histograms of all individual components are shown.

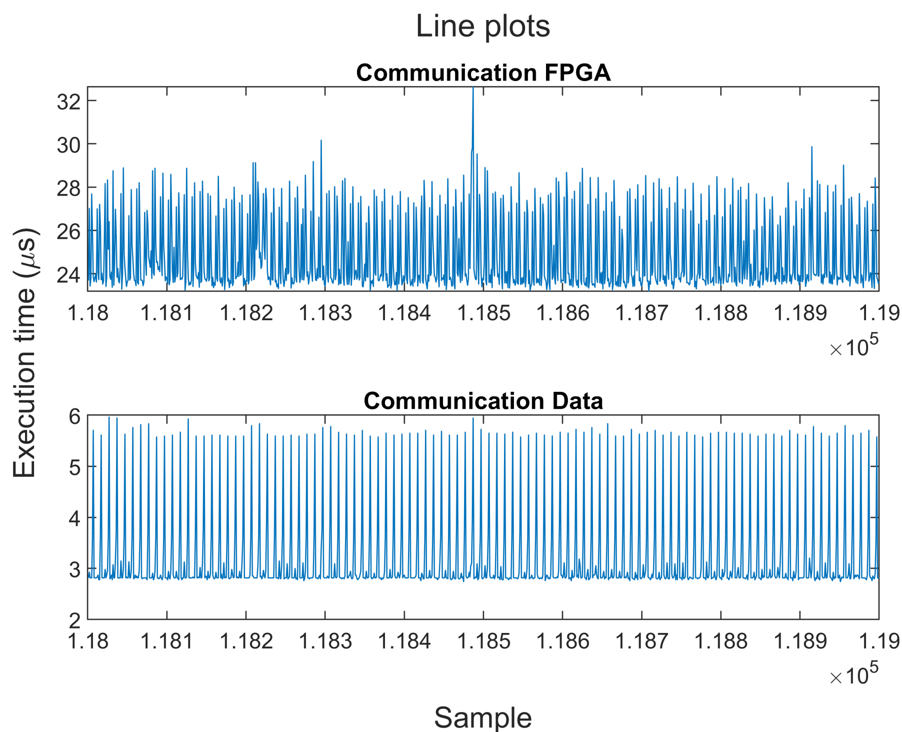


Figure 4.4: Line plots of individual components (Sample=118001:119000)

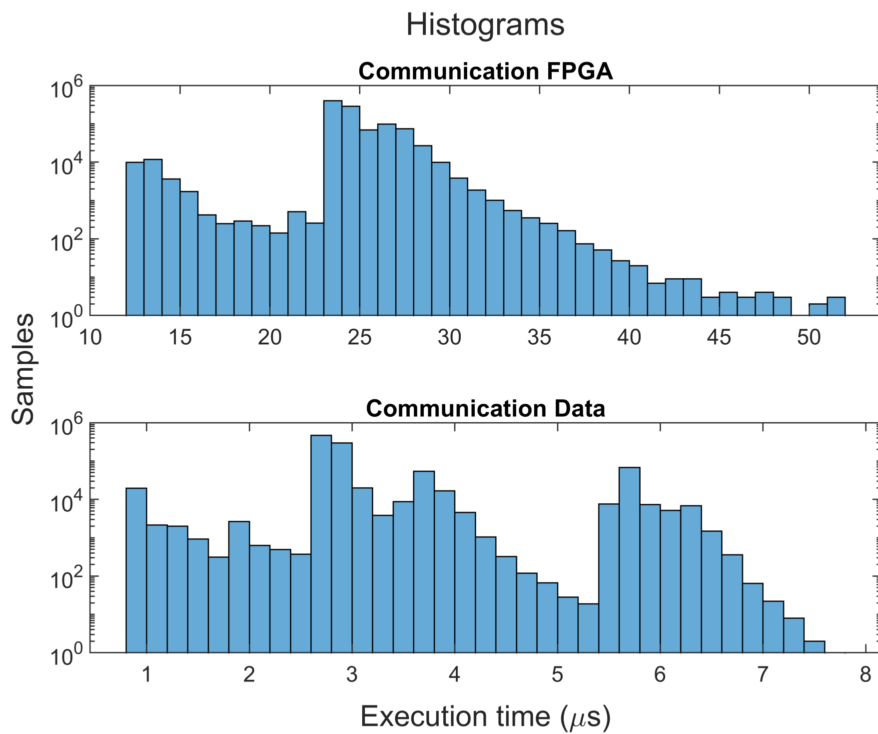


Figure 4.5: Histograms of individual components

Figure 4.4 shows clearly that the communication lines have some kind of pattern. To confirm this the autocovariance is calculated. The autocovariance can show if there is periodic behaviour. The results of this are shown in Figure 4.6. The autocovariance of all components are shown in Figure G.3

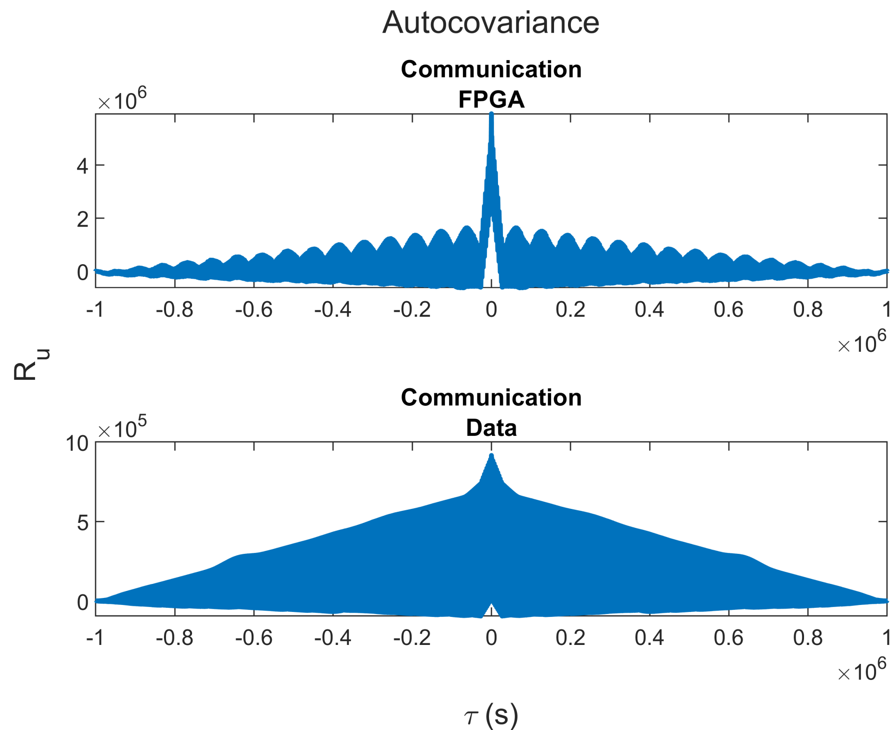


Figure 4.6: Autocovariance of separate components

The results show that for the communications there is a high degree of periodic behaviour. By analysing the execution time data of the 'Communication Command' and 'Communication Data' it was confirmed that the period between each peak is 10 ms. This exactly matches the communication time as only once every 10 cycles data is sent or received on the Xenomai kernel. This would mean that when sending or receiving data, extra execution time is needed.

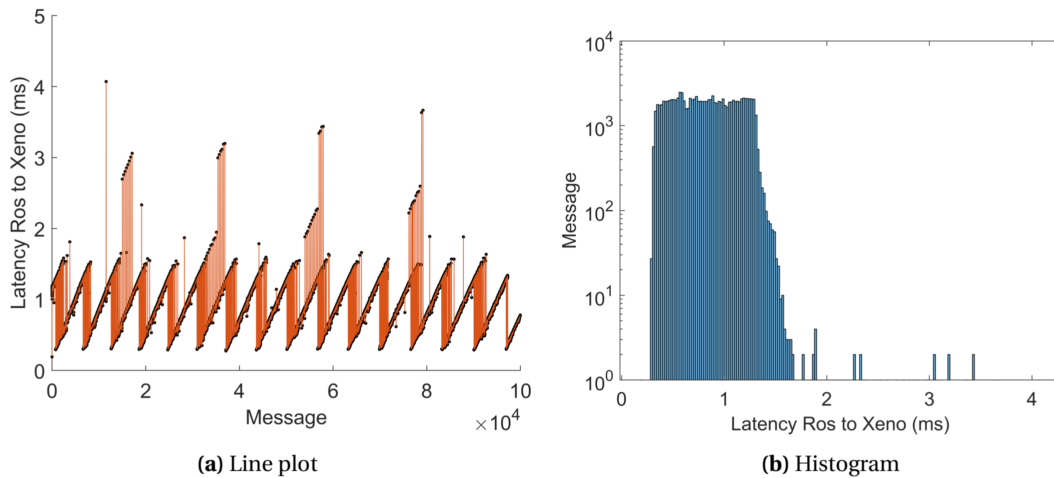
By using the message numbers received, it can determined when messages were received. This in combination with deducing which cycle data is sent from the Xenomai kernel, it can be determined whether sending or receiving data takes more time. If the average cycle time for the sending is higher, that means that the peaks are the result of sending data and the opposite is true for receiving data.

The result of the analysis using 'Communication Data' is an average execution time for receiving data of 3.17 μs and for sending data of 5.65 μs. This proves that sending data from ROS to Xenomai takes more time.

The FPGA communication also shows periodic behaviour, though no clear explanation has been identified for this. Since the FPGA communication is external, it must pass through the CPU's peripheral interface of the BCM2711 (Broadcom Inc., 2020). This interface could introduce interference, potentially affecting the communication. However, due to time constraints, no further investigation into this potential source of interference was conducted.

ROS to Xenomai latency

The results of the test for the ROS to Xenomai latency are shown in Figure 4.7. Figure 4.7a shows the latency for each subsequent message. Figure 4.7b shows the histogram of the latency data. In Table 4.2, an overview of different statistics of the data is shown.



(a) Line plot

(b) Histogram

Figure 4.7: ROS to Xenomai latency

	Mean (ms)	Median (ms)	Mode (ms)	SD	Min (ms)	Max (ms)	Data points
ROS to Xenomai latency	0.828	0.824	0.434	0.296	0.195	4.069	99974

Table 4.2: ROS to Xenomai latency

The data shows consistent behaviour with some interference. To explain this consistent behaviour a part of the plot in Figure 4.7a is shown zoomed in, in Figure 4.8. Starting from the lower left corner at point 1, it shows that the jitter starts building up from 0.3 ms. This build-up continues until approximately 1.3 ms at point 2. From this point onwards the latency jumps between 1.3 to 1.5 ms and 0.3 to 0.5 ms until point 3.

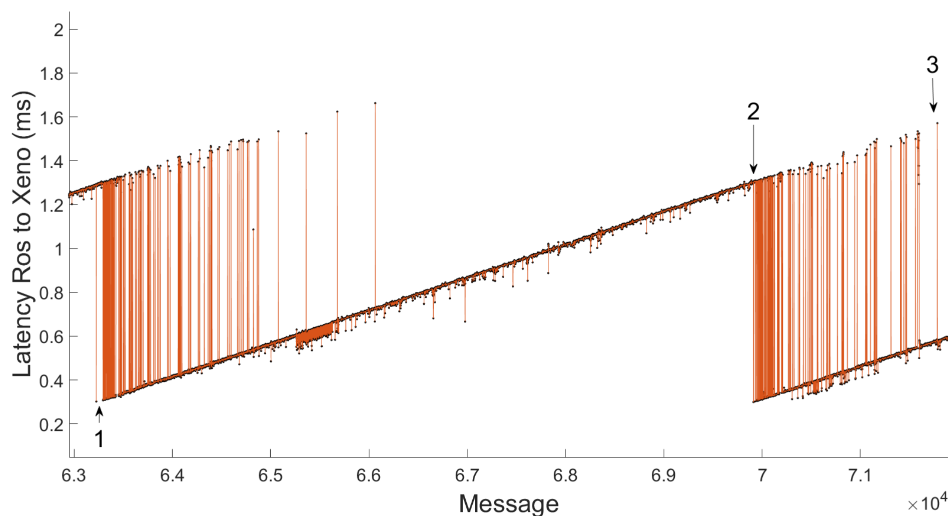


Figure 4.8: Zoomed in version of Figure 4.7a

The graph shows that a minimum of 0.3 to 0.5 ms is needed for sending data from a ROS node until it arrives at the Xenomai thread. This can be deduced from the jumping behaviour between points 2 and 3. This behaviour shows that the message arrives at the cross buffer at a

moment when the Xenomai thread might or might not read it on time. When not read on time, this will cause a delay of 1 ms which causes the latency to jump to 1.3 to 1.5 ms.

Through calculations as shown in Equation 4.2, it can be concluded that each succeeding message (N) has an extra 150 ns latency (Δt_{R2X}). This means that one or multiple objects within the communication create a compounding delay. From looking at the interference it can also be concluded that this extra delay is constant throughout the run. Figure 4.9 shows multiple blue lines which are the expected trend extended from the normal behaviour. These blue lines show that regardless of interference, the extra delay to the communication is constant throughout the run as even with interference the data points fall in proximity to the blue lines.

$$\Delta(\Delta t_{\text{R2X}}) = \frac{\Delta t_{\text{R2X}}}{\Delta N} = \frac{1\text{ms}}{6500} = 150\text{ns/Sample} \tag{4.2}$$

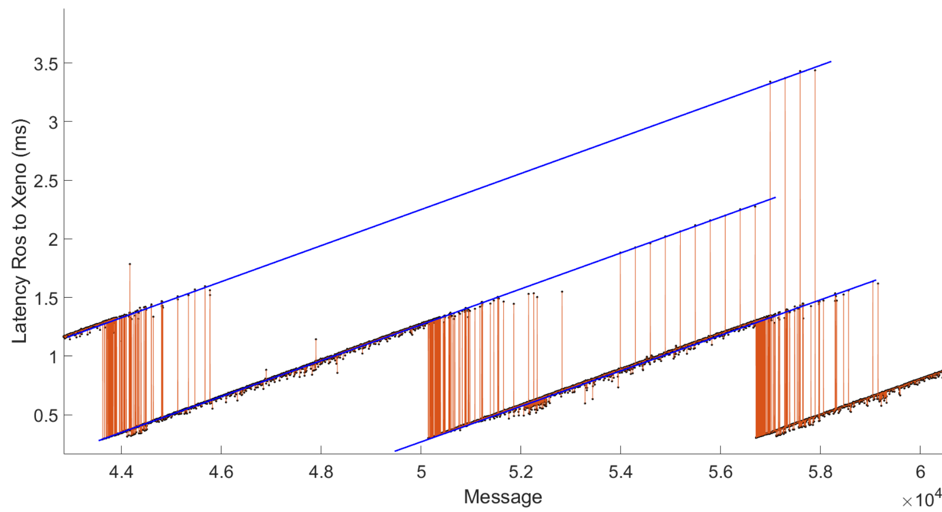
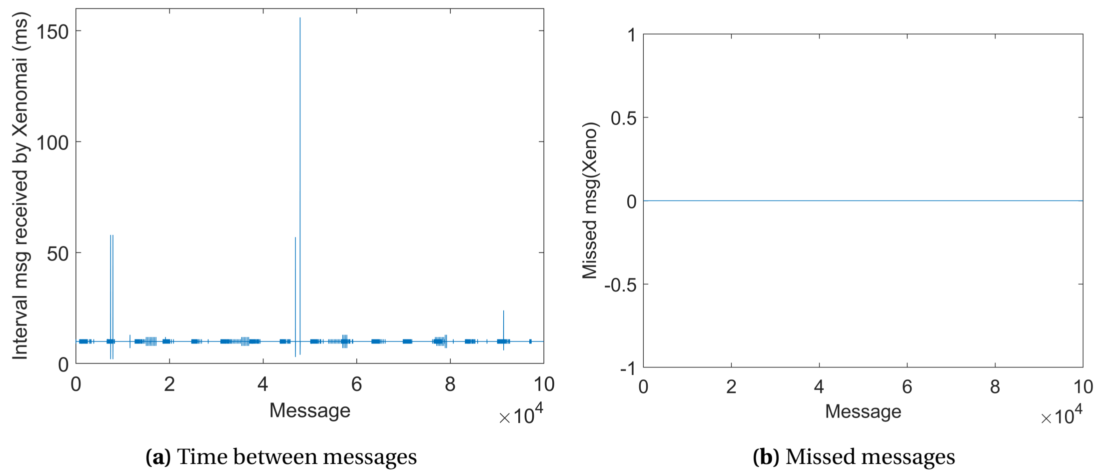


Figure 4.9: Zoomed in of Figure 4.7a in which blue lines are the expected trend of time extended.

Table 4.1 shows that the sampling jitter of the firm real-time loop on the Xenomai side has no build-up of jitter as the average jitter is 0.0 μs. To determine whether the ROS2 test node or one of the intermediary objects creates the delay, the jitter of the ROS2 test node was analysed. This was done using Equation 4.2 but replacing t_1 with the sent timestamp of the ROS to Xenomai messages. The data analysis excluded all jitter bigger than 1 ms as these were caused by other factors. Through the analysis, the average jitter of the ROS node was determined to be -155 ns. This matches the calculations and explains the behaviour of the increased latency.

ROS to Xenomai real-timeness

In Figure 4.10, the data representing the real-timeness of the communication is shown. Figure 4.10a shows the time between each message received. This should be 10 ms with small variations because of jitter. Figure 4.10a shows the missed message based on the message number sent with each message. This number should go up with one for each message sent. If the number between each message is bigger than one, it means a message has been missed.



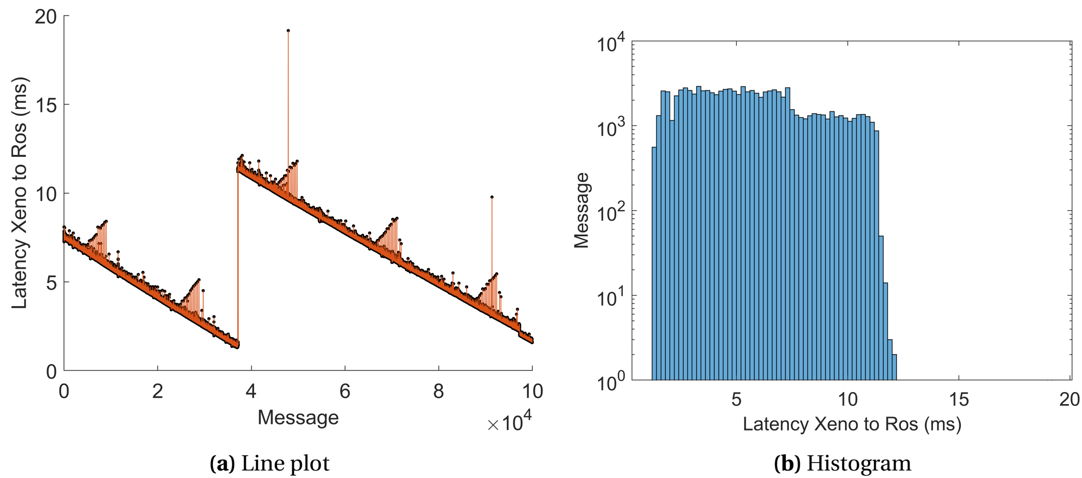
(a) Time between messages **(b)** Missed messages

Figure 4.10: Real-timeness of ROS to Xenomai communication

Figure 4.10b shows that all messages sent from ROS to Xenomai have been received. Although all messages have been received, the expected time between messages is not always met. This shows that the ROS Test node was only active when the RosXenoBridge was active. If this would not have been the case, Figure 4.10b would have shown that a message has been missed.

Xenomai to ROS latency

The results of the test for the Xenomai to ROS latency are shown in Figure 4.11. Figure 4.11a shows the latency for each subsequent message. Figure 4.11b shows the histogram of the latency data. In Table 4.3, an overview of different statistics of the data is shown.



(a) Line plot **(b)** Histogram

Figure 4.11: ROS to Xenomai latency

	Mean (ms)	Median (ms)	Mode (ms)	SD	Min (ms)	Max (ms)	Data points
Xenomai to ROS latency	5.707	5.449	3.272	2.650	1.297	19.149	99953

Table 4.3: Xenomai to ROS latency

The graphs show consistent behaviour with some interference. The consistent behaviour matches the behaviour shown in Section 4.1.2. The difference is that the behaviour is mirrored

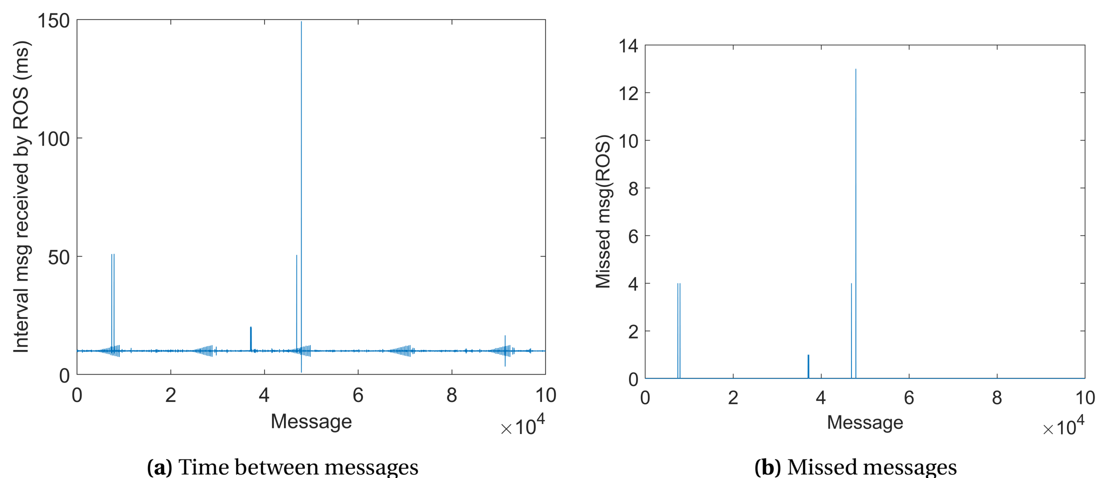
on the x-axis, the cycle of increased delay build-up is bigger and in the negative direction. The mirroring on the x-axis and the delay build-up in the negative direction can be explained by the switch in the direction of the communication as the data is now read earlier with each message because of the -155 ns jitter.

The increased cycle time for the build-up can be explained by the switch in the end destination. On the Xenomai side, the cycle time is 1 ms, on the ROS side, it is 10 ms. This means that only every 10 ms, the cross buffer is checked for data. The increased cycle time also ensures that the interference does not significantly influence the latency as there is a greater buffer time between each read on the ROS2 node.

Table 4.3 shows that a minimal 1.3 ms is needed for the for the communication from Xenomai to ROS2. This is an increase in minimum latency of 1 ms for this direction in contrast with the reverse direction. This comes from the way communication is handled in ROS2. When data is published to the `Ros2Xeno` topic, a callback function is called which publishes the data directly to the cross buffer. For data that is going to ROS2, it first has to be written to the cross buffer. Only when the `RosXenoBridge` node reads the data is it published to the `Xeno2Ros` topic. This increases the time needed based on the way the code of the `RosXenoBridge` is structured.

Xenomai to ROS real-timeness

In Figure 4.12, the data representing the real-timeness of the communication is shown. Figure 4.12a shows the time between each message received. This should be 10 ms with small variations because of jitter. Figure 4.12a shows the missed message based on the message number sent with each message. This number should go up with 10 for each message sent. If the number between each message is bigger than 10, it means a message has been missed.



(a) Time between messages **(b)** Missed messages

Figure 4.12: Real-timeness of Xenomai to ROS communication

From Figure 4.12 it can be determined that not all data that is sent from firm real-time loop arrives at the ROS topics. Figure 4.12 shows that up to a total of 13 consecutive messages can be missed. This is equal to 130 ms as every 10 ms, one message has to be sent. This means for that duration the Linux scheduler does not give any time to the RosXenoBridge node, ROS topics and/or the Test node. This proves the unreliability of the Linux kernel concerning being able to handle firm real-time tasks.

Figure 4.12a looks allot like Figure 4.10a. This is normal as both communications go through the RosXenoBridge node.

4.1.3 Conclusion of the characterisation test

The conclusion can be drawn that the framework has achieved the requirements set for the communication overhead and jitter. Although there is some interference in the communications and execution, these do not hinder the working of the framework too much. The jitter build-up in the communication latency is caused by the 155 ns constant negative jitter of the ROS2 test node. This has little to no effect on the framework but might influence the system as a whole as it increases the latency between the ROS and Xenomai part.

4.2 Stress test

Figure 4.3a shows in some points an increase in efficiency as the total overhead decreases to below 30 μs. This is especially shown towards the end of the test in the last 25,000 cycles. A hypothesis was made to explain the improvement. This hypothesis is:

- The Raspberry Pi has increased performance with high CPU usage.

To prove this hypothesis a test had to be done. This test is the same as the test discussed in Section 4.1 but with an added CPU usage. This was done through the following function:

```
stress -c 4
```

This function increases the CPU usage to 100 % of all but the Xenomai core. Figure 4.13 shows the result of this using the Linux `htop` function which shows the CPU usage. All but core 1 has added stress as this is the Xenomai core.

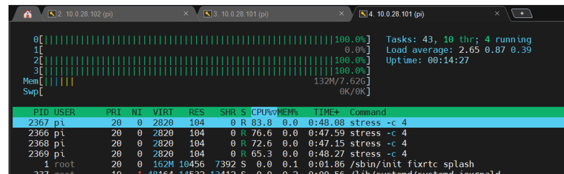


Figure 4.13: Result of Linux `htop` function

4.2.1 Results

Table 4.4 shows the statistics gathered from the test. The green cells show a decrease with respect to the initial test. A red cell shows an increase. A white cell means the value did not change. The results show an overall decrease in the timing statistics except for the maximum value. The total overhead average decreased by 49.6 %. The jitter is the only part that was completely negatively affected by the stress, as all statistics have increased.

	Mean (μs)	Median (μs)	SD (μs)	Min (μs)	Max (μs)	DL (μs)	On time (%)
Total overhead	17.84	17.09	1.59	15.93	102.26	-	-
Change	-49.6 %	-17.59 μs	-59.8 %	-0.8%	+48.6%	-	-
Communication	16.28	15.57	1.46	14.63	97.74	45	99.99
Change	-47.6 %	-14.67 μs	-60.0 %	+0.4 %	+57.1 %	-	+0.05 %pt
Comm FPGA	13.74	13.46	0.87	12.65	87.96	30	99.99
Change	-44.0 %	-10.69 μs	-64.2 %	+0.2 %	+69.2 %	-	+0.81 %pt
Comm ROS	2.54	2.07	0.91	1.85	19.24	15	99.99
Change	-61.2 %	-3.84 μs	-54.5 %	-2.0 %	-2.2 %	-	<0.01 %pt
Comm Command	1.45	1.15	0.55	0.963	13.20	-	-
Change	-57.5 %	-1.96 μs	-48.6 %	0.0 %	+2.2 %	-	
Comm Data	1.10	0.93	0.39	0.85	7.48	-	-
Change	-65.0 %	-1.88 μs	-59.4 %	-4.2 %	-3.1 %	-	
Monitor Update	0.08	0.06	0.07	0.055	1.56	-	-
Change	-58.0 %	-0.14 μs	+133.3 %	0.0 %	-31.7 %	-	
Storing log data	1.39	1.35	0.16	1.06	14.04	-	-
Change	-64.5 %	-2.67 μs	-65.2 %	-11.0 %	+48.6 %	-	
Jitter sample Freq	0.00	-0.01	0.694	-34.42	34.80	10	99.99
Change	0.0 %	-0.012 μs	+56.5 %	+83.0 %	+127.5 %	-	>-0.01 %pt

Table 4.4: Statistics stress timing test and the change with respect to Table 4.1 in either %, absolute values or percentage points (%pt) (Office for National Statistics, 2024). For mean, median, SD, Min and, Max, the closer to zero the better. For 'On time' the bigger the better. (SD = standard deviation, DL = deadline, Comm = Communication)

Table 4.5 shows the statistics of the latency results. There is an overall decrease but this is minimal and not as significant as the timing results. Looking at the standard deviation the overall spreading of the results has increased.

	Mean (ms)	Median (ms)	SD (ms)	Min (ms)	Max (ms)	Data points
ROS to Xenomai latency	0.702	0.698	0.304	0.153	10.022	99975
Change	-15.2 %	-15.3 %	+2.7 %	-21.5 %	+146.3 %	+1
Xenomai to ROS latency	5.398	4.900	2.874	1.175	14.391	99954
Change	-5.4 %	-10.1 %	+8.5 %	-9.4 %	-24.8 %	+1

Table 4.5: Statistics latency specification under stress and the change with respect to Table 4.2 and Table 4.3 in either % or absolute values. For mean, median, SD, Min and, Max, the closer to zero the better. For 'Data points' the bigger the better. (SD = standard deviation)

One of the interesting results is the ROS to Xenomai real-timeness results. In the normal test, no send message went missing but as shown in Figure 4.14 some went missing in the stress test.

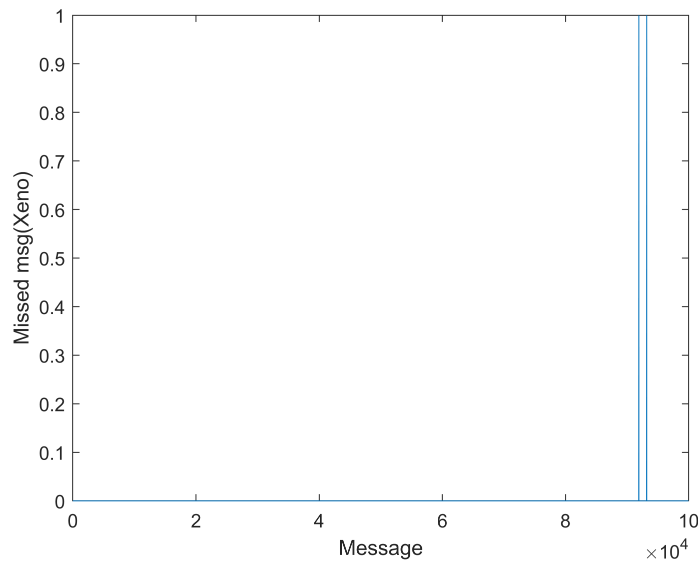


Figure 4.14: Missed messages of ROS to Xenomai communication under stress

The hypothesis is proven true. Some information has been acquired that may indicate that this is the result of the Raspberry idling at a lower frequency and only increasing the frequency when needed (Linux Kernel Documentation, 2023). This is called CPU frequency scaling and is used to prevent overheating and save power.

4.3 Implementation test

For the implementation test, one of the use cases is implemented.

The RELbot was chosen despite Table 2.2 showing that the YouBot would have been better to show the coordination implementation. The reason for this is that the youBot is not up and running and making it ready is a project on its own.

For the implementation test, one of the assignments of the ASDfR course has been implemented. For this assignment, the RELbot had to follow a green ball. It used a camera video input that was accessible through the Linux kernel as input to detect the green ball. From there, the setpoint of the green ball had to be detected and a setpoint had to be generated. The setpoint is then sent to Xenomai in which the loop control calculates the speeds for each wheel. Figure 4.15 shows the setup of the implementation for the RELbot.

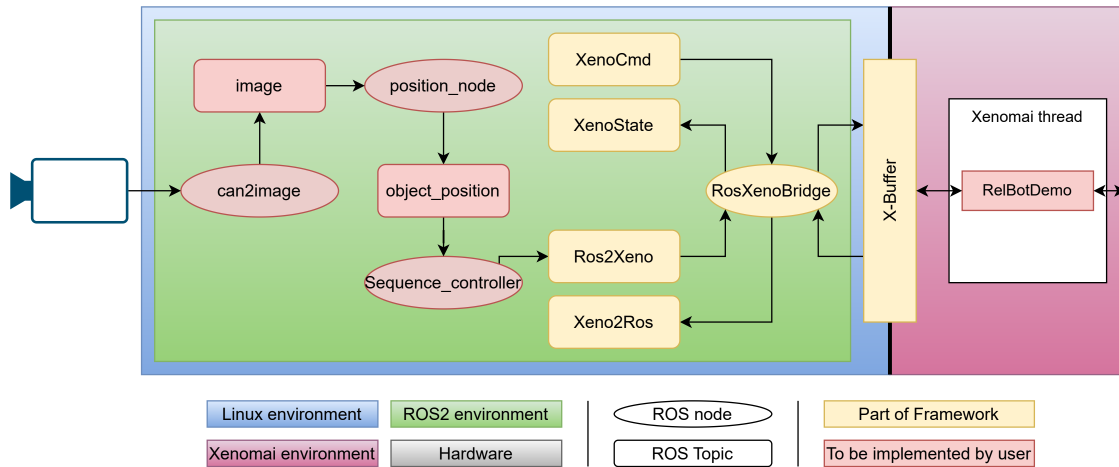


Figure 4.15: Setup for implementation test for RELbot

The result of this test is a video clip of the RELbot following the green ball. The video clip is shown in this link. This test shows the working of the framework but only limited as this implementation has been done by the maker of the framework. To truly test whether the framework works, user testing has to be done with different target groups. Limited user testing has been done with an early alpha version of the framework, but no user testing has been done with the latest version because of the limited project time.

5 Conclusions and Recommendations

5.1 Conclusions

The first main design objective of this project is to create a framework that allows the end users to implement a firm real-time task on Xenomai while being able to communicate with ROS2. This goal has been achieved. The implementation test shows that the framework works and can communicate with ROS2.

The first sub-design objective is for the framework to be easy to use in education. This objective is achieved by taking the needs of all the different target groups within education into consideration in the analysis. Furthermore, multiple instructions and tips for both educators and learners on how to use the framework are provided. These instructions are added in the appendix.

The second sub-design objective is for the framework to be flexible for different use cases. This was achieved by considering the use cases in the analysis. Only one of the use cases was implemented. Therefore, to verify the flexibility of the framework more testing is required.

The third sub-design objective was to monitor and log data in the Xenomai kernel. This goal is achieved. It was used to gather the data on the Xenomai side for the characterisation tests, which proves that it is working. Appendix D shows how to use the logger. Appendix F shows how the monitor is used in the framework.

The second main design objective is the characterisation of the framework concerning communication latency and real-timeness of the communication between ROS and Xenomai and the overhead of the firm real-time loop. Through the characterisation test in Chapter 4 the results were obtained. The latency for the communication from ROS to Xenomai is on average 0.828 μs. The real-timeness is depended on the Linux kernel's performance and deadline misses can go up to at least 15 cycles. But all messages that were sent were received, except when under load some messages were lost. For the communication from Xenomai to ROS an average latency of 5.7 μs was measured, with up to 13 cycles of missed messages.

For the overhead, an average of 35 μs was used with the communication overhead being 31 μs. All communication requirements were met with at least 99 % of the deadlines having been met. The jitter requirement was 99.999 % met, but with added stress, this was less although not significant.

5.2 Recommendations

The first recommendation is to do user testing. User testing is needed to validate whether the framework has any shortcomings. These user tests have to be done on learners as they are the majority of the user base of the framework.

The second recommendation is to implement the framework for the youBot. The youBot is the most complex use case for the framework. Implementing the framework for the youBot will validate the framework and might bring to light some shortcomings.

The third recommendation is to look into ways to reduce the unnecessary programs on the Linux side to improve performance. These programs take time from the processor which leaves less time for the ROS programs. By reducing these programs more calculation intensify programs can be run.

The fourth recommendation is to look into how to permanently run the Raspberry Pi at the highest possible clock speed without having to add load. This will greatly increase the efficiency

of the real-time loop. If overheating is a problem, solutions like passive cooling through a heat sink and active cooling through a fan might help.

The fifth recommendation is to remodel the 20-sim class structure to fit the needs of RaM. There are two ways to do this. The first is the 20-sim auto code generator, which has an built-in function to remodel the generated code. The second option is to use the token XML file more. These solutions can increase the 20-sim integration into the framework to a higher degree which will increase the ease of use for the user.

The sixth recommendation is to look into the SPI communication. The SPI communication towards the FPGA is not as efficient as it probably can be. The current overhead is ± 24 μs. When under load it is around ± 14 μs. Technically the communication can take around 3.1 μs. Furthermore, there is currently still interference in the communication that has not been solved. In previous research (Raoudi, 2023), WiFi interference increased the execution time. This can currently also be the case.

The seventh recommendation is to keep documentation of the Xenomai kernel within the RaM group. The Xenomai website is constantly updating. The Xenomai website (Xenomai 4 project, 2024c) does not keep documentation separate for older versions. Therefore, documentation on the website is not always a good representation of the code used, especially for older versions of the Xenomai kernel.

The last recommendation is to upgrade the software versions of both Linux, ROS and, Xenomai. Newer LTS versions have been released during this project for Linux and ROS. There should be no major problems upgrading to the newest versions of these with the framework, except certain Xenomai functions might have changed.

A 5Cs approach

The 5Cs approach (Klotzbücher et al., 2013) separates a component into 5 concerns (5Cs). Below, the concerns are explained:

1. The Computation is about the execution of the control algorithm of a component.
2. The Coordination is about the state machine of a component.
3. Configuration is about configurable parameters that can change the behaviour of a component
4. The Communication takes care of the communication to the environment of a component.
5. The Composition is about how the above four aspects interact with each other in a component.

B Missed cycles, method 2

Table B.1 shows an example of how many cycles are needed to catch up with a missed deadline using the second method described in Section 3.1. For this example, the deadline is missed by 1 ms with a cycle time of 1 ms. The first row shows the amount of execution needed for each cycle assuming it stays relatively constant. Each column shows the time after the execution of each cycle. For example the

The green-coloured cells are on time. In the red-coloured cells, the deadline still has not been caught up with. In the yellow-coloured cells, the time is equal to the deadline.

Table B.1 shows that for lower execution times after one cycle the loop has already caught up to the deadline. For higher execution times more cycles are needed.

Cycle \ Execution time (ms)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
1	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9
2	2.2	2.4	2.6	2.8	3.0	3.2	3.4	3.6	3.8
3	3.1	3.2	3.3	3.4	3.5	3.8	4.1	4.4	4.7
4	4.1	4.2	4.3	4.4	4.5	4.6	4.8	5.2	5.6
5	5.1	5.2	5.3	5.4	5.5	5.6	5.7	6.0	6.5
6	6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.8	7.4
7	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	8.3
8	8.1	8.2	8.3	8.4	8.5	8.6	8.7	8.8	9.2
9	9.1	9.2	9.3	9.4	9.5	9.6	9.7	9.8	10.1
10	10.1	10.2	10.3	10.4	10.5	10.6	10.7	10.8	11.0
11	11.1	11.2	11.3	11.4	11.5	11.6	11.7	11.8	11.9

Table B.1: Cycle compensation calculation for missed deadline of 1 ms on a 1ms cycle time.

C OPCODE commands IcoIo class

OPCODE	IcoIo function	Value	Description
NOOP	-	0	Does nothing
INIT	init	1	Checks connection
RESET	reset	2	Resets the encoder position and outputs to zero.
ALL	update_io	3	Set the new PWM and output data for all the ports and send back all the encoder and input data.
PWM_1	setOutput	4	Set a new PWM and output data for port 1
PWM_2	setOutput	5	Send the encoder position and input data for port 1
PWM_3	setOutput	6	Set a new PWM and output data for port 2
PWM_4	setOutput	7	Send the encoder position and input data for port 2
ENC_1	getInput	8	Set a new PWM and output data for port 3
ENC_2	getInput	9	Send the encoder position and input data for port 3
ENC_3	getInput	10	Set a new PWM and output data for port 4
ENC_4	getInput	11	Send the encoder position and input data for port 4
TEST	test	12	Send an array of test values and and a correct array has to be received back

Table C.1: Function + OPCODE's of the IcoIo class/library

D Instructions for implementation and use of XenoFrtLogger

XenoFrtLogger provides methods to firm real-time log data on Xenomai when the control loop is running. This is done through a proxy that communicates the data from the Xenomai to the Linux kernel to log the data on a file. This class works on both the Xenomai and the Linux kernel.

The logger works by having one variable to log data from. This variable has to be defined by the user. Each element in the variable has to be registered. Start and stop commands can be given to determine when the logger has to log the data.

Below, step-by-step instructions are given on how to implement and use the XenoFrtLogger. Under each step, a code example is shown. This code used matches the unit test for the XenoFrtLogger.

1. Create a variable that is going to be logged. This variable can consist of multiple elements like an array or struct. When using structs make sure to use "#pragma pack (1)" before the struct and "#pragma pack (0)" after the struct (cppreference, 2024). This will remove the space within the memory between different elements. In the definition of the struct, default values can be given.

```
// Definition of the struct ThisIsAStruct
// A normal struct name is PosistionData
#pragma pack (1)
struct LoggedVariable
{
    int this_is_a_int;
    double this_is_a_double;
    float this_is_a_float;
    char this_is_a_char;
    bool this_is_a_bool;
};
#pragma pack(0)

// Variable that is going to be logged
struct LoggedVariable data_to_be_logged;
```

2. Go into the `main` function and create a XenoFileHandler instance which represents the log file.

```
file =
XenoFileHandlerfile(1,"/home/pi/Logger_Test","bin");
```

3. Create the XenoFrtLogger instance.

```
logger = XenoFrtLogger(&file,
&data_to_be_logged);
```

4. Register all elements in the variable to the logger. Table D.1 shows all the variable types that can be registered. The registration of the elements has to happen in the same order as the allocation in the variable.

Name	Size	Type
id_bool	1	bool
id_char	1	char
id_float	4	float
id_int	4	int
id_double	8	double
id_long_int	8	int64, timespec.tv_sec, timespec.tv_nsec

Table D.1: Different constants of the enum VariableTypes.

```
    logger.addVariable("this_is_a_int", id_int);
    logger.addVariable("this_is_a_double",
id_double);
    logger.addVariable("this_is_a_float", id_float);
    logger.addVariable("this_is_a_char", id_char);
    logger.addVariable("this_is_a_bool", id_bool);
```

5. After having added all elements, initialise the logger once. If done twice, the program will crash. A check can be done to avoid this. This call might take a while therefore it is advised not to do this within a control loop.

```
if(!logger.isInitialised())
    logger.initialise();
```

6. Start the logger by calling:

```
logger.start();
```

7. The data has to be allocated to the variable given by the logger before it can be logged. Afterwards, the `log` function has to be called, to sent the data to be logged. This has to happen each cycle the data has to be logged. In the example below 100 cycles are simulated in a `for` loop.

```
    // Setting initial values for data_to_be_logged
    data_to_be_logged.this_is_a_bool = 0;
    data_to_be_logged.this_is_a_int = 0;
    data_to_be_logged.this_is_a_char = 'R';
    data_to_be_logged.this_is_a_float = 2.0;
    data_to_be_logged.this_is_a_double = 4.0;

    //Cycling through 100 cycles
    // And changing the data each cycle
    for (size_t i = 0; i < 99; i++)
    {
        // Allocating data to data_to_be_logged to be logged
        data_to_be_logged.this_is_a_bool =
!data_to_be_logged.this_is_a_bool;
        data_to_be_logged.this_is_a_int++;
        if(data_to_be_logged.this_is_a_char == 'R')
            data_to_be_logged.this_is_a_char = 'A';
        else
            data_to_be_logged.this_is_a_char = 'R';
```

```
            data_to_be_logged.this_is_a_float =
data_to_be_logged.this_is_a_float/2;
            data_to_be_logged.this_is_a_double =
data_to_be_logged.this_is_a_double/4;

            // Sent data to the log file
            logger.log();
      }
```

8. Stop the logger when not more needed, by calling:

```
logger.stop();
```

9. After being done with logging, the user can decode the log file into a .mat or .csv file. This can be done by going to ’ros2-xenomai4-framework/XenoRosFramework/Common/XenoFrtLogger_decoder’ folder in the framework. In this folder, two python scripts reside for decoding the log file to the preferred file type. This is done by calling one of the scripts in the terminal and following the given instructions.

```
python binairy_to_matlab.py
```

E Educators check list for student works

Below, a list of points is given on which the work of students can be check on by educators.

1. Check if all the names are appropriate. This includes:
    (a) Classes, both the file name and class name should be the same.
    (b) Variables
    (c) Functions
    (d) Project names in CMakeList.txt files
2. The naming convention should be consistent. The current naming convention is:
    (a) Names of classes use UpperCamelCase.
    (b) Names of functions use lowerCamelCase.
    (c) Names of variables use snake_case.
    (d) All classes that make use of the Xenomai library start with 'Xeno'.
    (e) All classes that need a firm real-time loop to function use 'Frt' in their name.
3. If deviated from the naming convention, the deviation has to be implemented consistently throughout the code.
4. Unused or useless code should be removed.
5. Functions and variables within classes must have the correct access level (`PRIVATE PUBLIC PROTECTD`).

F Instructions for implementation and use of XenoFrtRosIco or XenoFrt20Sim

The ROS 2 – Xenomai 4 framework provides a skeleton to set up a communication infrastructure between ROS2 and an FRT loop running on EVL / Xenomai 4. The framework provides the FRT methods to monitor and log data in the FRT loop on Xenomai.

Note: The controller function must be named `LoopController`. Indeed, hardcoded. Sorry for that. We did not have time to use the submodel name of the code generation output of 20-sim. This will of course be taken care of in a later version.

F.1 Structure of the Framework

The file structure of the framework is shown in Figure F.1.

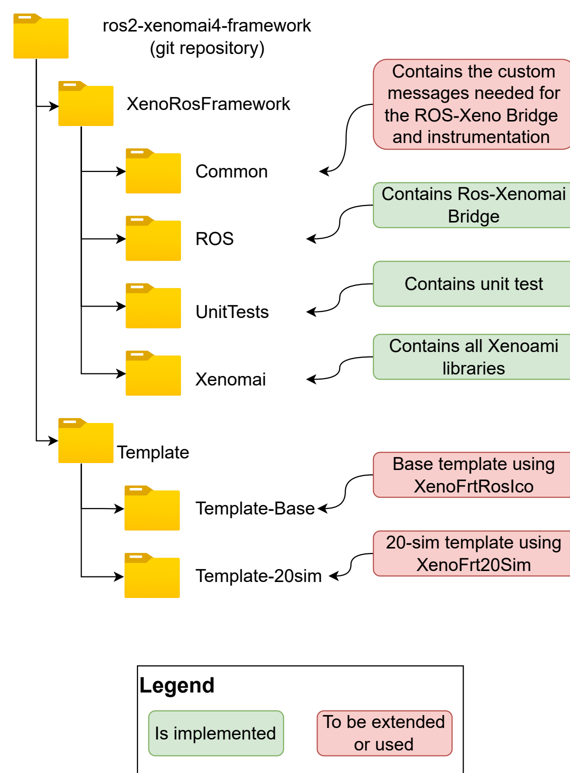


Figure F.1: File structure of the framework. Files in green annotated folders must be extended.

A general structure of typical use of the framework is shown in Figure F.2.

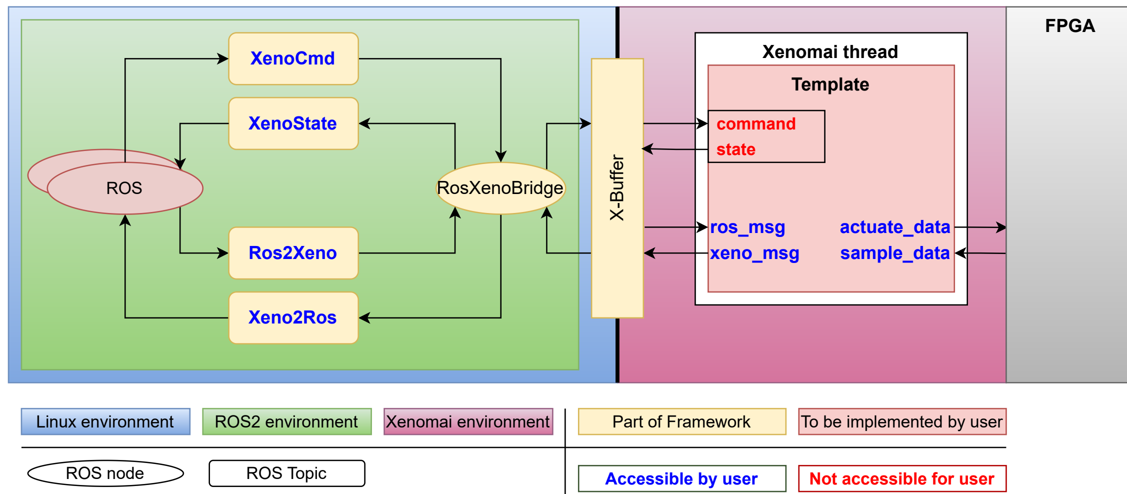


Figure F.2: Typical use of the framework.

F.2 Use of the Framework

Below a step-by-step instruction is given on how to extend a template using the ROS2-Xenomai4 framework with your code. It also includes the steps needed for building and running your code.

1. If you already have a ROS2 workspace go to step 4.
2. Create a folder for the workspace.
3. Create a folder in the [WORKSPACE] folder called src.
4. Download the 'ros2-xenomai4-framework' framework from Canvas, unpack it and put the framework in the src folder.
5. Copy one of the templates in the [WORKSPACE]/src/ros2-xenomai4-framework/Template folder into the [WORKSPACE]/src folder. Copy the complete folder of the selected template and not only the files inside the folder.
6. Change the name of the copied template folder to suit the project. Your folder tree should look like Figure F.3.

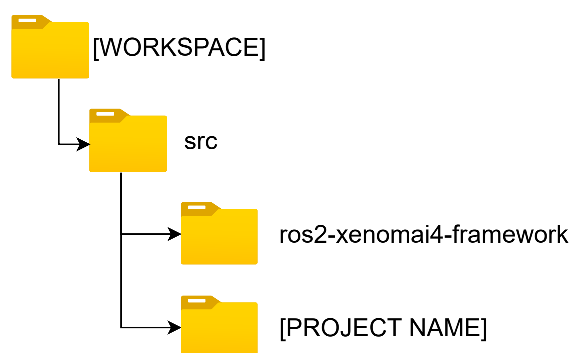


Figure F.3: Folder structure of workspace.

7. Go to [WORKSPACE]/src/ros2-xenomai4-framework/ XenoRosFramework/Common/-custom_msgs/msg folder.
8. In this folder the definitions of the message types are stored that you can use to communicate between ROS2 and Xenomai. The Ros2Xeno is the message type for communication from ROS2 to Xenomai. The Xeno2Ros is the message type for the communication from Xenomai to ROS2. Change the two message types to the needs of your project and

give each element in the message a suitable name. These names will be used in your code.

9. Go to [WORKSPACE]/src/[PROJECT NAME]
   - **20-sim variant**
     (a) Go into the /controller folder and remove all files within it.
     (b) Generate 20-sim code from your model and copy it to this folder. Take note, currently the class of the generated code has to be named **LoopController**.
     (c) Go back to [WORKSPACE]/src/[PROJECT NAME].
10. Open the CMakeList.txt file and rename the project name on line 2 to suit the use case.
11. Go into the /include folder and rename `Template.hpp` to suit your project. This file contains the definition of the class responsible for implementing the computation components of your system.
12. Open the .hpp file. Rename the class and (de)constructor to the file name.
   - If the logger is used see Appendix D about how to use it. If not used remove the the struct `GenericLogStruct`.
   - **20-sim variant**: Change the sizeof the variables `u` and `y` to the correct size specified by your 20-sim model.
13. Go back to the [WORKSPACE]/src/[PROJECT NAME] folder and open the /src folder.
14. Rename `Template.cpp` to same name as the .hpp file.
15. Open the .cpp file.
16. Each function except the (de)constructor in the class represents a state. The state diagram is shown in Figure F.4. The state machine can be controlled by two ROS2 topics. The states are published in the `XenomaiState` topic. The commands can be given by publishing the correct integer value to the `command` topic. Each command and its corresponding value is shown in Table F.1.

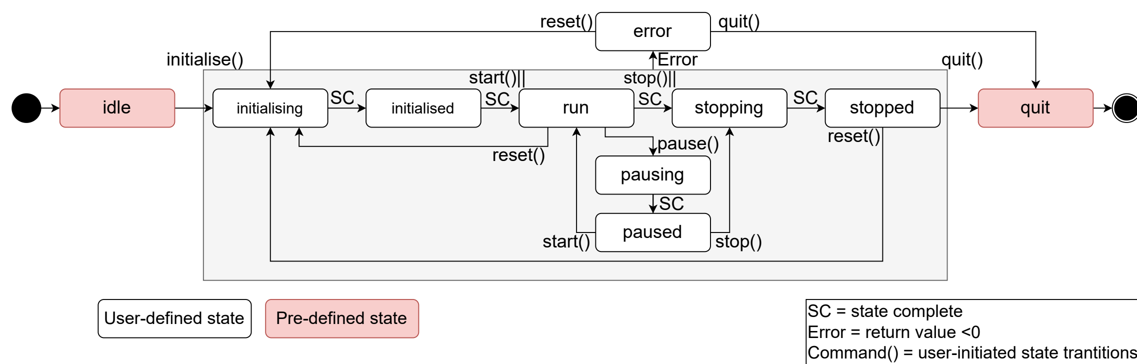


Figure F.4: Diagram of state machine implemented.

Command	Value	Discription
initialise	1	Go from idle to initialising state. Before any movement can happen this command has to be given from the ROS side of the communication.
start	2	Go from the initialised or paused state to the run state.
stop	3	Go from the run or paused state to the stopping state.
reset	4	Go from the run, stop, error or pause state to the initialising state.
pause	5	Go from the run state to the pausing state.
quit	6	Go from the error or stopped state to the quit state. After this command is called, the Xenomai side will shut down.

Table F.1: Commands that can be given through the ROS2 topic `XenoCmd` to the FSM on the Xenomai4 side.

17. Remove all code lines in the functions that you are *not* going to use. Do *not* remove the return statements.
18. The variable `sample_data` provides all the sampling data from the FPGA and has the following structure:

```
struct IcoRead
{
    int channel1;       // Encoder value from channel 1
    int channel2;       // Encoder value from channel 2
    int channel3;       // Encoder value from channel 3
    int channel4;       // Encoder value from channel 4
    bool channel1_1;    // Digital input 1 from channel 1
    bool channel1_2;    // Digital input 2 from channel 1
    bool channel2_1;    // Digital input 1 from channel 2
    bool channel2_2;    // Digital input 2 from channel 2
    bool channel3_1;    // Digital input 1 from channel 3
    bool channel3_2;    // Digital input 2 from channel 3
    bool channel4_1;    // Digital input 1 from channel 4
    bool channel4_2;    // Digital input 2 from channel 4
};
```

The variable for controlling the FPGA output is called `actuate_data` and its structure is:

```
struct IcoWrite
{
    int16_t pwm1;       // PWM value for channel 1
    bool val1;          // Digital output 1 of channel 1
    int16_t pwm2;       // PWM value for channel 2
    bool val2;          // Digital output 1 of channel 2
    int16_t pwm3;       // PWM value for channel 3
    bool val3;          // Digital output 1 of channel 3
    int16_t pwm4;       // PWM value for channel 4
    bool val4;          // Digital output 1 of channel 4
};
```

The data sent from ROS2 to Xenomai is called `ros_msg` and has the structure of the Ros2Xeno message. The data to be sent from Xenomai to ROS2 is set in the variable called `xeno_msg` and has the structure of the Xeno2Ros message.
19. Implement the computation.
    - **20-sim variant**
        (a) The 20-sim model is already imported and is called `controller`. Through the `controller.calculate` function one computation step is performed. The `controller.finished` function tells when the simulation is finished. Explore the controller class for other functionality provided by the 20-sim generated code.
20. Tip: Use `monitor.printf()` instead of `evl_printf()` for debug messaging to reduce the output flow of the printf statements to a lower frequency.
21. Open `main.cpp` and replace the template class with your own class.
    (a) On line 13 the object of the class is made. The parameters of the class are in the following sequence:
        i. Cycle time frequency (Hz)

   ii. Communication frequency to ROS (Hz)
   iii. Monitor frequency (Hz) of the `monitor.printf()`

22. If not already done add all your ROS2 projects to the [WORKSPACE]/src/ folder.
23. Go to the [WORKSPACE] folder.
24. To compile all the projects listed in /src call `colcon build`. To exclude a project from being compiled add a file called COLCON_IGNORE to the source folder of that project. This will exclude it from being compiled by `colcon build`.
25. After compiling three directories will be added to the [WORKSPACE] folder. In the build folder, you will find different folders for all the projects built including both the ROS2 and Xenomai projects.
26. Call in the terminal:
    `source install/setup.bash`
27. To run the Xenomai project call:
    `sudo ./build/`*`[PROJECT NAME]`*`/`*`[PROJECT NAME]`*
28. To start the ROS-Xenomai bridge call in a different terminal:
    `ros2 run ros_xeno_bridge RosXenoBridge` Note: A Xenomai4 project has to be started first or it will not work.
29. To run other ROS2 nodes call in separate terminals:
    `ros2 run` *`[PACKAGE NAME] [EXECUTABLE NAME]`*
30. Tip: Create a ROS2 launch file to make it easy to start all needed ROS2 nodes and topics. This reduces the amount of terminals needed. Commands to publish data to certain topics can also be added to launch files.
31. Do not forget to publish the initialise command to the`XenoCmd` topic to start the computation (see Table F.1).

G Results individual components

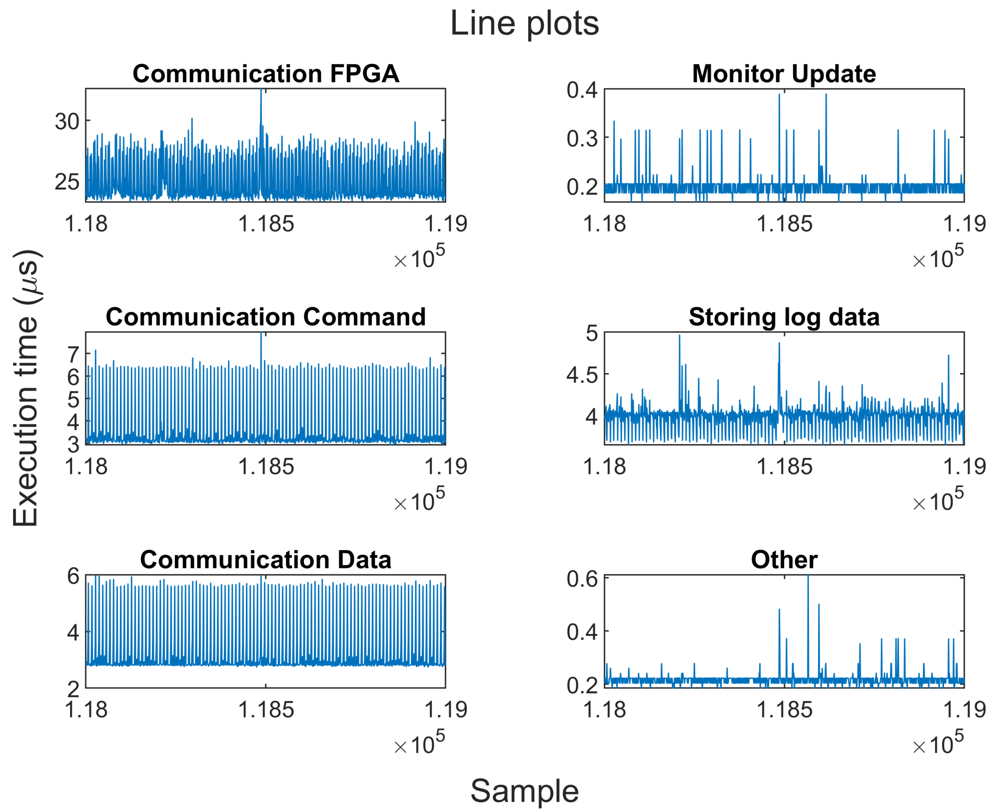


Figure G.1: Line plots, overhead results of individual components (Sample=118001:119000)

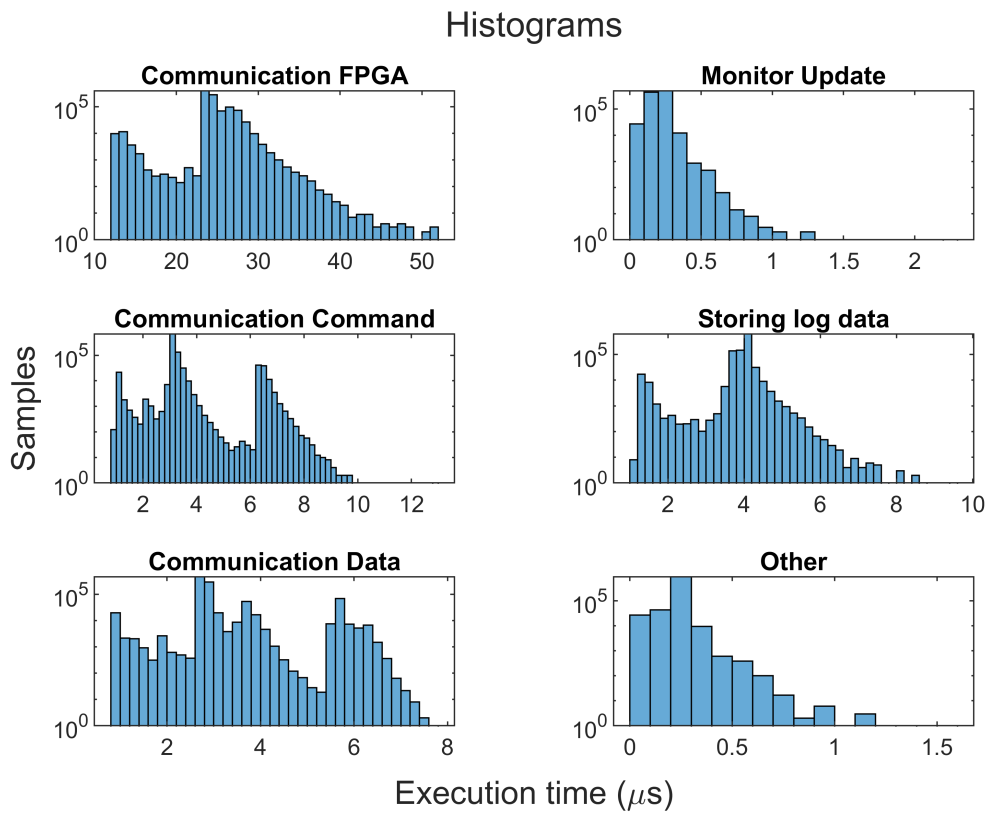


Figure G.2: Histograms, overhead results of individual components

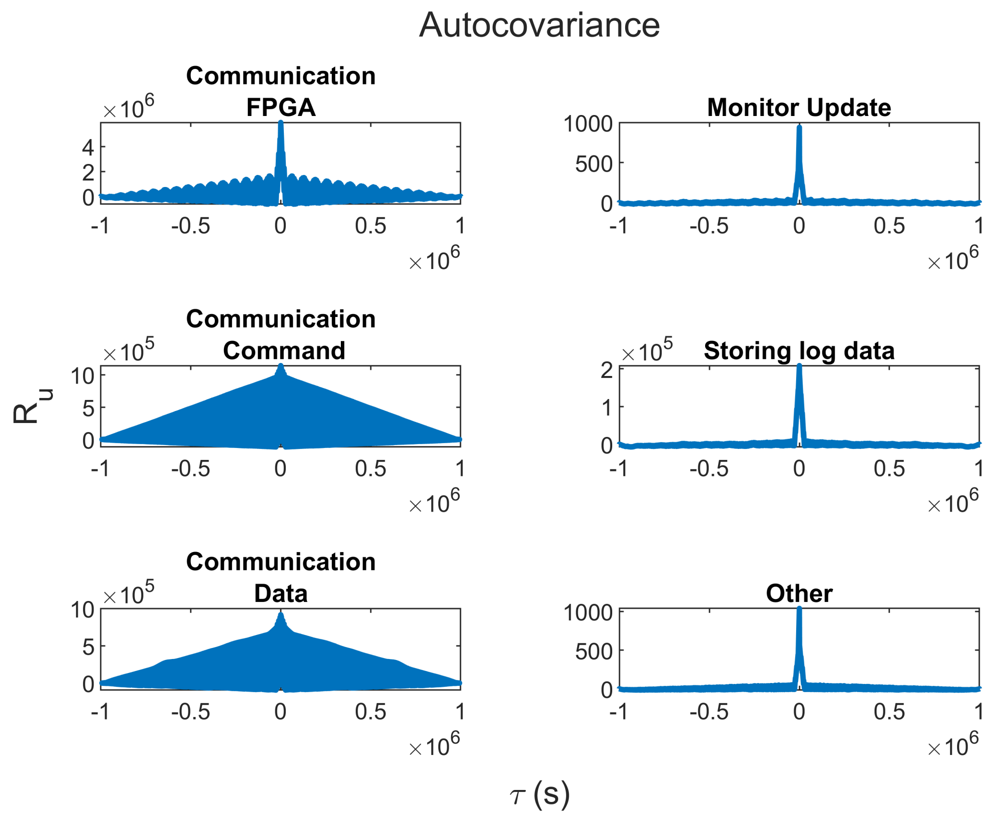


Figure G.3: Autocovariance, overhead results of individual components

References

Bezemer, Maarten M. (2013). 'Cyber-Physical Systems Software Development: Way of Working and Tool Suite'. Doctoral dissertation. PhD thesis. Netherlands: University of Twente. URL: https://doi.org/10.3990/1.9789036518796.

Blanchard, Benjamin S. and Wolter J. Fabrycky (1st Jan. 2011). *Systems engineering and analysis*. Prentice Hall.

Bradner, Scott (Mar. 1997). *Key words for use in RFCs to Indicate Requirement Levels*. Request for Comments: 2119. [Online; accessed 12-September-2024]. URL: https://www.ietf.org/rfc/rfc2119.txt.

Broadcom Inc. (Feb. 2020). *BCM2711 ARM Peripherals*. Accessed: 2024-11-08. URL: https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf.

Broenink, Jan F. and Yunyun Ni (2012). 'Model-driven robot-software design using integrated models and co-simulation'. In: *2012 International Conference on Embedded Computer Systems (SAMOS)*, pp. 339–344. DOI: 10.1109/SAMOS.2012.6404197.

cppreference (2024). *Implementation defined behavior control*. URL: https://en.cppreference.com/w/cpp/preprocessor/impl (visited on 30/10/2024).

Hoogendijk, T. A. (2013). *Design of a generic software component for embedded control software using CSP*. University of Twente: Robotics and Mechatronics. URL: https://cloud.ram.eemcs.utwente.nl/index.php/s/q5FoaBACB5qPZJ5.

Klotzbücher, Markus et al. (2013). 'The BRICS Component Model: A Model-Based Development Paradigm For Complex Robotics Software Systems'. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*. ACM, pp. 1758–1764. DOI: 10.1145/2480362.2480695.

Linux Kernel Documentation (2023). *CPU Frequency Scaling*. Accessed: 2024-11-08. URL: https://docs.kernel.org/admin-guide/pm/cpufreq.html.

Macenski, Steven et al. (2022). 'Robot Operating System 2: Design, architecture, and uses in the wild'. In: *Science Robotics* 7.66, eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

Marwedel, Peter (2011). *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Accessed: 2024-11-08. Springer. ISBN: 978-94-007-0257-8. DOI: 10.1007/978-94-007-0257-8. URL: https://link.springer.com/book/10.1007/978-94-007-0257-8.

Meijer, Bram (Oct. 2021). *REAL-TIME ROBOT SOFTWARE FRAMEWORK ON RASPBERRY PI USING XENOMAI AND ROS2*.

Office for National Statistics (2024). *Numbers and percentages*. Accessed: 2024-11-08. URL: https://service-manual.ons.gov.uk/content/numbers/percentages.

OMAC - Organization for Machine Automation and Control (2009). *OMAC PackML State Machine*. Accessed: 2024-11-08. URL: https://us.store.codesys.com/media/n98_media_assets/files/000060-F/0/OMAC%20PackML%20State%20Machine_en.pdf.

OpenAI (2024). *ChatGPT*. Accessed: October 14, 2024. URL: https://chatgpt.com/.

Raoudi, Ilyas (Apr. 2023). *Latency Reduction and Modularity Improvement of a Raspberry Pi-FPGA Control System*. Technical Report. Individual Assignment. Enschede, The Netherlands: University of Twente, Robotics and Mechatronics Group. URL: https://www.utwente.nl.

ROS2 (2024a). *Code style and language versions — ROS 2 Documentation: Humble documentation*. URL: https://docs.ros.org/en/humble/The-ROS2-

Project/Contributing/Code-Style-Language-Versions.html (visited on 09/09/2024).

ROS2 (2024b). *Creating a workspace — ROS 2 Documentation: Humble documentation*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html (visited on 17/09/2024).

Xenomai 4 project (2024a). *Bi-directional out-of-band in-band messaging :: Xenomai 4*. URL: https://v4.xenomai.org/core/user-api/xbuf/ (visited on 29/10/2024).

— (2024b). *File proxy :: Xenomai 4*. URL: https://v4.xenomai.org/core/user-api/proxy/ (visited on 29/10/2024).

— (2024c). *Overview :: Xenomai 4*. URL: https://v4.xenomai.org/overview/ (visited on 27/11/2024).