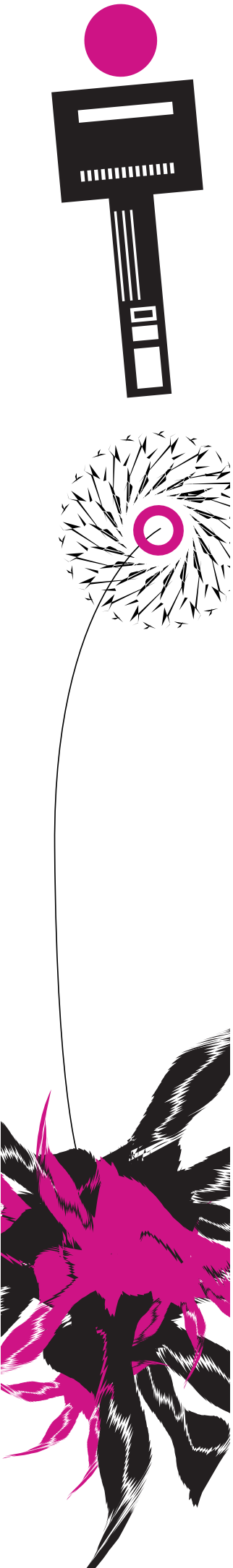


RAM ROBOTICS AND MECHATRONICS

EPG: AUTOMATING MODEL-TO-CODE TRANSLATION FOR MICRO-ROS AND ROS 2

D.V. (Daniël) Huiskes

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
dr. ir. G. van Oort
dr. ir. P.T. de Boer

March, 2025

015RaM2025
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

Robotics, as a multidisciplinary field, requires expertise in control design, system modelling, software development, and mechatronics, and often lacks integrated tools that combine these domains efficiently. Existing tools address specific aspects, but no comprehensive solution exists.

To bridge this gap, this thesis introduces the Embedded-Project Generator (EPG), a model-driven development (MDD) software tool designed to automate the creation of microcontroller-based projects. These EPG-generated projects are configured for real-time management of model-based mechatronic control software, and enable network connectivity between a robot, controlled by a microcontroller that uses micro-ROS to facilitate communication with ROS 2 on a coordinating single-board computer.

The goals of this thesis are to design a method to automate the translation from a robot-controller model to code that uses the ROS 2 ecosystem and supports real-time control; design and implement a software tool based on the automation method; test the performance of code, generated by the software tool, with a robot; and test the performance and stability of a distributed network with multiple robots using code generated by the software tool.

The designed automation method is implemented in the EPG software tool with a GUI that guides users through selecting a target configuration and connecting model ports to target ports. The EPG combines model-generated code, user inputs, and the target configuration to create a microcontroller project.

Six target configurations are implemented and tested for the EPG. The targets include bare-metal, FreeRTOS, and Zephyr on the Raspberry Pi Pico; bare-metal and FreeRTOS on the Raspberry Pi Pico 2; and Zephyr on the STM32 Nucleo-H743ZI. All targets are tested in a ping-pong round-trip message passing application that is used to determine the performance of the micro-ROS communication. Subsequently, EPG-target implementations are tested on JIWY, a 2-DoF robot. JIWY is used to perform micro-ROS communication and control loop performance measurements. Besides that, a motion profile is used to steer both the simulation model of the robot as the actual robot, enabling a comparison of their behaviour. The tests confirm that the EPG-generated projects function as intended and demonstrate the EPG's ability to generate projects that meet the 1 ms firm real-time control loop and 33 ms soft real-time communication requirement.

Additionally, network tests are conducted with a ping-pong test confirming stable communication between nodes with consistent round-trip times. Furthermore, the performance and stability of a distributed ROS 2 network incorporating four JIWYs, controlled by EPG-targets, as ROS 2 nodes is tested. The conducted tests demonstrate the effectiveness of a distributed system using EPG-generated projects deployed on EPG-targets, in achieving stable motion tracking, real-time synchronisation, and teleoperation.

In addition to testing on the JIWY robot, an EPG-target implementation is tested on the RELbot, a 2-DoF robot, demonstrating the EPG's compatibility with a different mechatronic system. Three RELbots are tested in a distributed network, performing motion profile tracking and teleoperation, with the setup developed through rapid prototyping in one day.

The goals are achieved with the implementation of the designed automation method in the EPG software tool. Testing confirms the EPG's effectiveness, ensuring stable communication, control loop performance, and enabling robots to operate within a distributed network.

For future work, it is recommended to test the EPG on various robots, to identify possible limitations and optimisation opportunities. Additionally, evaluating it in a distributed system, like the Production Cell, is also recommended to assess adaptability and robustness.

Contents

1 Introduction

1.1 Context

Robotics is a multidisciplinary field requiring expertise in control design, system modelling, software development, and mechatronics. Developers often specialize in one area, complicating the integration of various system components. The wide range of available tools and techniques makes it difficult for engineers to assess compatibility and find optimal combinations across system levels.
Developing robot software with requirements such as real-time performance, distributed control, and networking can be complex and time-consuming. Transitioning from model development to implementation in a robotic system is challenging, as there is currently no comprehensive software tool that seamlessly integrates model development, networking, real-time execution, and direct deployment for robotic systems.
There are software tools that address some of these challenges. For example, OROCOS (Soetens, 2024) supports real-time control, but lacks model-driven development (MDD) integration. Papyrus for robotics (The Eclipse Foundation, 2023), a high-level modelling tool, SMARTMDSD (Stampfer et al., 2016), a system composition tool, and ReApp (Wenger et al., 2016), a ROS-based framework with tools for modelling, all focus on software modelling instead of physical-system modelling. They offer tools for component assembly, but lack direct robotic system deployment, and have limited support for real-time performance. 20-sim 4C (Controllab, 2025) enables rapid prototyping but lacks networking support for ROS 2 and is incompatible with the latest real-time Linux versions needed for real-time execution. Additional examples are shown in Appendix D.
This thesis presents the Embedded-Project Generator (EPG), a MDD software tool developed in this project to integrate solutions addressing these challenges. The EPG generates fully configured directly deployable microcontroller-software projects. The generated project includes real-time management of model-based mechatronic control software and establishes network connectivity with a coordinating single-board computer to steer and monitor a robot. Figure 1.1 illustrates the functional context of the EPG. In this report, the colour green is used to show the elements developed in this thesis when presented alongside existing ones.

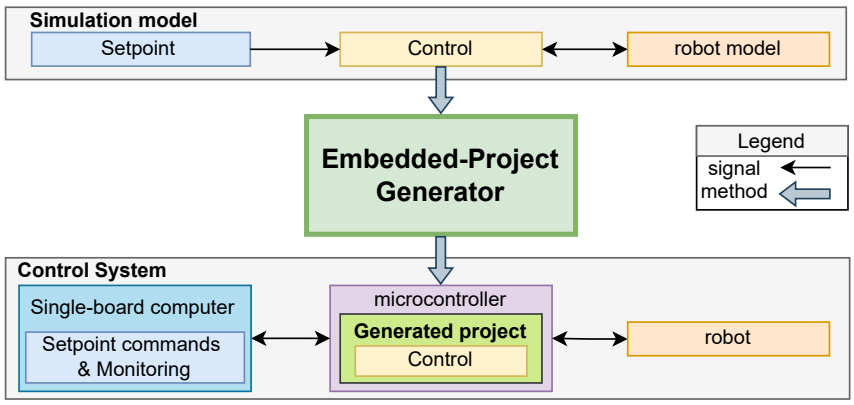


Figure 1.1: Embedded-Project Generator functional context. Green blocks show thesis work.

To provide a preview of the outcomes of this thesis, this **video** presents a demonstration of the EPG in action.

1.2 Goals

The main goal of this thesis is to develop an MDD software tool to automate the translation from a robot-controller model to code for an embedded device, enabling both real-time control of a mechatronic system and integration within the ROS 2 ecosystem for networking.

The goals of this thesis are therefore as follows:

- Design a method to automate the translation from a robot-controller model to code that uses the ROS 2 ecosystem and supports real-time control.
- Design and implement a software tool based on the automation method.
- Test the performance of code, generated by the software tool, with a robot.
- Test the performance and stability of a distributed network with multiple robots using code generated by the software tool.

The project constraints of this thesis are as follows:

- The code, generated by the software tool, uses the ROS 2 ecosystem for networking.
- The code, generated by the software tool, must support a robot setup that uses a Raspberry Pi single-board computer.
- The software tool must support direct implementation of 20-sim model-generated code.

1.3 Report outline

Chapter 2, presents essential background information for this thesis. Subsequently, Chapter 3 presents an analysis in which design requirements are derived from a use case to realise the goals stated in Section 1.2. The requirements lead to a design-space exploration systematically showing the trade-off between design options to substantiate the chosen setup. Chapter 4 shows the design of the EPG, a component-based software tool for networked robot systems that incorporates all analysis outcomes. In Chapter 5 the performance of different test setups is evaluated. The thesis is concluded with Chapter 6, which combines conclusions and recommendations.

2 Background

2.1 Introduction

This chapter provides background information needed to understand the concepts that are used in this thesis. As part of this thesis an extensive literature study is conducted on software development methodologies, software tools, and software architectures relevant for this thesis. This chapter discusses the directly relevant aspects of this study, while the complete literature study, including all results, is added to Appendix D.

2.2 Component-based software

Component-based software methodologies facilitate the re-use of code and aim to provide convenient system assembly by combining several components.
According to Szyperski (2002) the definition of a software component is as follows: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.". In other words, this implies that a component functions as a black box that incorporates all the functionality, referring to the internal operations, processes, and algorithms that define the component's behaviour, within the box. The box can have multiple inputs and outputs with pre-defined data formats. Figure 2.1 shows a general high-level component description with multiple inputs and outputs. The functionality within the component can operate stand-alone.
The components that result from a component-based methodology can therefore be combined just like building blocks to quickly build a software architecture. Since the components can operate independently, they can also be conveniently re-used in different software structures.

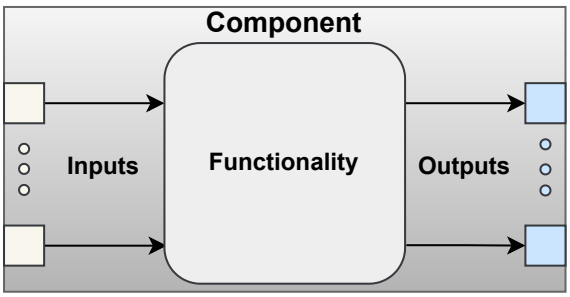


Figure 2.1: Diagram of a component

Brugali and Scandurra (2009) state that "a robot–software architecture describes the decomposition of the robot control system into a collection of software components, the encapsulation of functionality and control activities into components, and the flow of data and control information among components.". The advantage of describing a robot-software architecture in this component-based way is that the functional and non-functional requirements of the robot system can be realised by mapping the requirements on parts of the component. The functional requirements of the robotic system can for example be implemented within the functionality of the component. Non-functional requirements, such as how communication within the system should take place can be implemented by specifying the input and output data formats of the component. More literature on this topic is discussed in Section D.2.6.

2.3 Robotic software architectures

2.3.1 ROS 2

The Robot Operating System (ROS 2) (Macenski et al., 2022) is a set of software libraries and tools that can be used to build robot applications.
ROS 2 uses the Data Distribution Service (DDS). This is an open standard for communication and enables security, embedded and real-time support, multi-robot communication, and operations in non-ideal networking environments (Macenski et al., 2022).
ROS 2 uses nodes, which can be considered as components, as discussed in Section 2.2. A node has functionality that performs a certain task and uses a publisher-subscriber communication model, where publishers send data to a topic, making the data available to all nodes with subscribers to the topic.
Figure 2.2 shows an example in which node Coordinator publishes setpoints on topic /Setpoint to steer node Robot, which subscribes to topic /Setpoint. Similarly, the node Robot publishes position data on topic /Position for monitoring, received by the subscriber of node Coordinator.

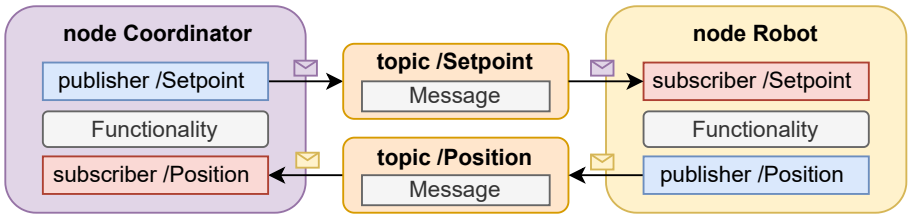


Figure 2.2: ROS 2 node communication example - Node Coordinator publishes setpoints on topic Setpoint to steer node Robot, which subscribes to topic Setpoint. The node Robot subsequently publishes position data on topic /Position for monitoring, received by the subscriber of node Coordinator.

Although ROS 2 claims to have real-time support, achieving real-time performance requires the use of additional real-time middleware. While ROS 2 provides the necessary infrastructure and features for real-time communication, the integration of dedicated real-time middleware is essential to fully achieve deterministic and predictable behaviour in real-time applications.

2.3.2 Micro-ROS

Micro-ROS (Belsare et al., 2023) is a real-time framework designed to bring ROS 2 functionalities to resource-constraint microcontroller devices.

The most important aspects of micro-ROS are:

- Microcontroller-optimised client API supporting all major ROS 2 concepts.
- Extremely resource-constrained but flexible middleware.
- Seamless integration with ROS 2.
- Multi-RTOS support.

Micro-ROS makes use of the DDS-XRCE protocol, which stands for DDS For Extremely Resource Constrained Environments. This resource constraint variant of the DDS protocol can directly be used within the ROS 2 architecture and seamlessly integrates with DDS.
Micro-ROS can be used in combination with a real-time operating system (RTOS), such as FreeRTOS or Zephyr, or with a bare-metal implementation. The advantage of using micro-ROS in combination with an RTOS is that different scheduling algorithms can be used to obtain real-time performance.
Micro-ROS follows the same node-based architecture of ROS 2 that makes use of message passing between the nodes by subscribing and publishing on topics.

The architecture overview of micro-ROS is shown in Figure 2.3. This figure shows that micro-ROS runs on a microcontroller on top of an RTOS. The Micro XRCE-DDS client integrated in micro-ROS can connect to a ROS 2 agent that runs on another device such as a Raspberry Pi. By doing this, a connection can be established over which messages can be sent between nodes. The ROS 2 agent connects micro-ROS nodes (i.e. components) on MCUs seamlessly with standard ROS 2 systems. This allows accessing micro-ROS nodes with the known ROS 2 tools and APIs just as normal ROS nodes.

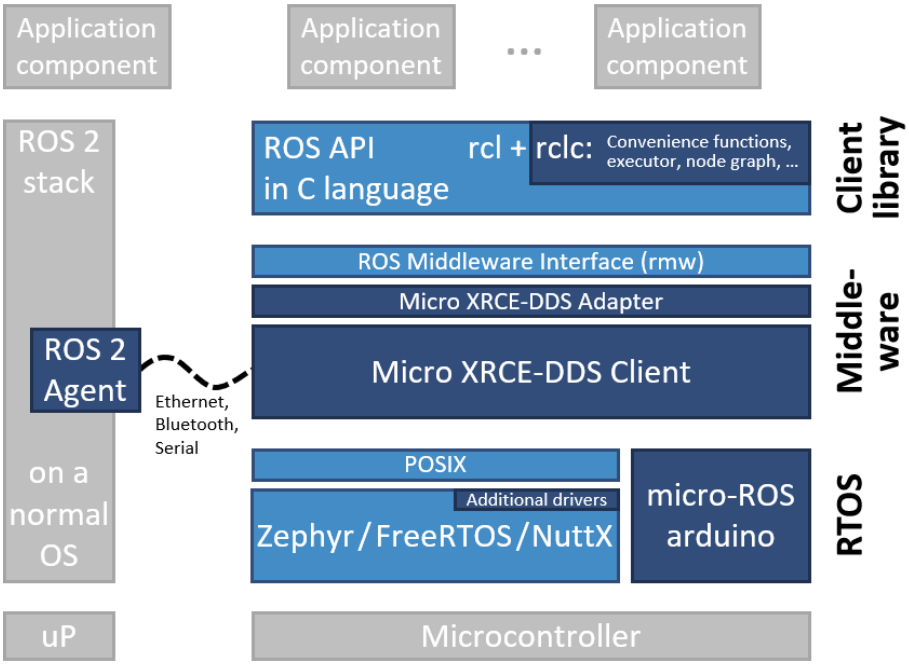


Figure 2.3: Architecture of the micro-ROS stack. Adapted from Belsare et al. (2023)

The seamless integration of micro-ROS with ROS 2 enables microcontrollers to fully participate in a distributed network. Figure 2.4 shows the operation of micro-ROS within the OSI network stack. The figure depicts a multi-device network configuration in which two single-board computers (SBCs) are interconnected via an Ethernet switch and communicate using ROS 2 Fast DDS, which operates over UDP/IP. Each single-board computer is connected to a microcontroller unit (MCU) running micro-ROS, allowing the microcontrollers to function as integral components of the ROS 2 ecosystem.
Micro-ROS communication between a micro-ROS client on the Raspberry Pi Pico, Pico 2, and Nucleo H743ZI, and a micro-ROS agent on a single-board computer uses serial USB 2.0, but is limited to speeds of 12 Mbps. The process of sending data involves serialising the data, applying HDLC framing, and packaging into a USB 2.0 package before transmission, while receiving data follows the reverse order. The package being transmitted follows a layered structure where the USB 2.0 protocol handles communication with a 7-byte overhead, the HDLC frame encapsulates the XRCE-DDS message with an additional 7-byte overhead, and the XRCE-DDS message itself carries micro-ROS topic data with a 12-byte overhead. The transmitted data consists of an 8-byte double-precision floating-point number, and the total overhead is 26 bytes, resulting in a final package size of 34 bytes and an estimated transmission time of about 23 μs. It should be noted that this is only the transmission time, without taking into account additional overheads, such as serialisation and deserialisation of the packages. More details about the micro-ROS communication are provided in Appendix B.

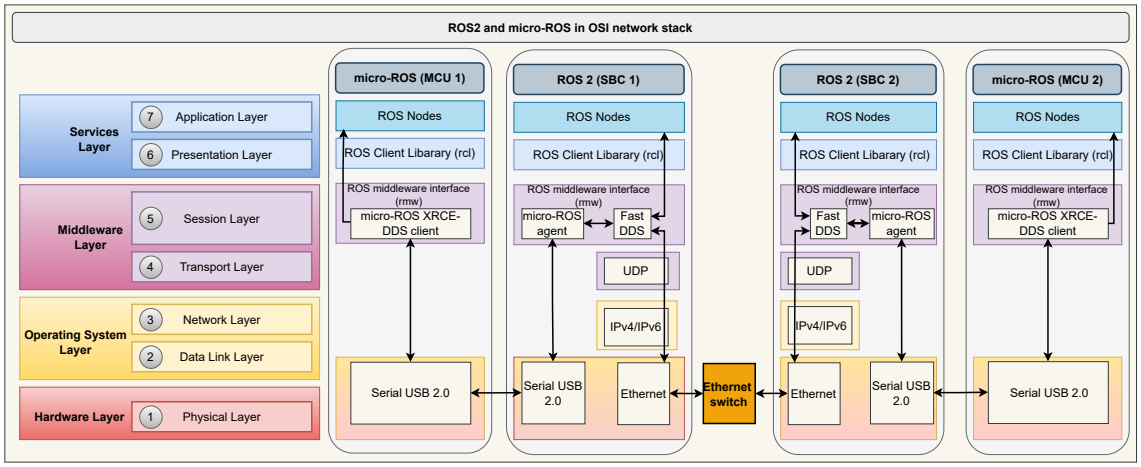


Figure 2.4: Micro-ROS and ROS 2 in OSI network stack

2.3.3 Xenomai/EVL

Xenomai adds real-time functionalities to non-real-time operating systems, such as Linux. Xenomai/EVL is a companion core that can be used in combination with a Linux kernel forming a dual kernel architecture (EVL Project, 2023).
It is possible to communicate with the real-time kernel from the Linux kernel by using the Dovetail interface. Figure 2.6 shows how the different layers stack on each other.
Communication between the real-time EVL core and the non-real-time general purpose kernel is possible by means of a cross-buffer as shown in Figure 2.6. This buffer connects the in-band and out-of-band contexts with each other and makes it possible to use the general read() and write() functions on the inbound side and similar oob_write() and oob_read() functions on the outbound side.

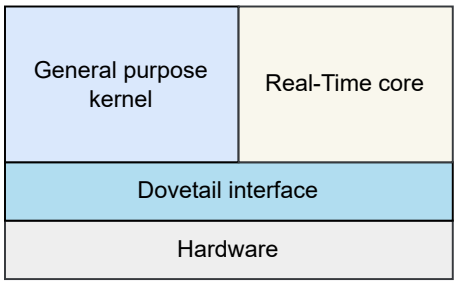


Figure 2.5: Xenomai Linux kernel

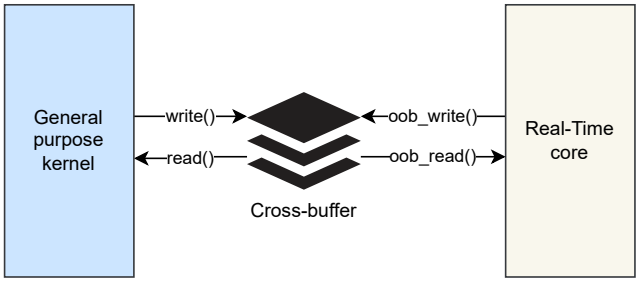


Figure 2.6: Cross-buffer

2.4 Modelling and simulation tool 20-sim

20-sim is a modelling and simulation tool for mechatronic systems (Controllab, 2025). It enables multi-domain modelling using energy-based bond graphs, supports simulation, and has a C-Code generation functionality to generate C or C++ code from model elements that can be run on microcontrollers or single-board computers.
The Targets.ini file is the starting point of the 20-sim code generation process. In 20-sim you can create a new target for code generation by defining a Targets.ini configuration file. It specifies which template files should be used for code generation. Within the template files, 20-sim tokens act as placeholders for information about the models such as variable names, equations, and inputs/outputs. During the 20-sim code generation step, the tokens are replaced by the corresponding model information, and the resulting generated code is placed in a destination folder, which is specified before the code generation process starts.

Besides template files, additional commands can be executed using preCommand and post-Command, which make it possible to run a terminal command in the target directory before or after the C-Code is generated. This option adds flexibility to the 20-sim code generation process by enabling the execution of pre- and post-processing software.

2.5 Device tree

Device trees are a concept for representing hardware configurations. They are particularly used in the context of Linux-based operating systems (Linux Foundation, 2025), but are also essential in Real-Time operating systems such as Zephyr (Zephyr Project, 2025). Device trees are used to allow for flexible and modular hardware configurations without having to hardcode the configuration for each individual device.
According to Gibson (2006) “The device tree consists of nodes representing devices or buses. Each node contains properties, name-value pairs that give information about the device. The values are arbitrary byte strings, and for some properties, they contain tables or other structured information.”. This means the device tree is a hierarchical data structure, where each node holds configuration information of a hardware element, such as hardware settings and data ports.

3 Analysis

3.1 Introduction

This chapter presents an analysis of the requirements needed to reach the goals of this thesis and determines the best software/hardware configuration to be used for the EPG.
First a use case is formulated from which functional and non-functional requirements are derived. Subsequently the requirements are used in a design-space exploration that evaluates two software/hardware configurations. The configurations are Xenomai on a single-board computer with an FPGA for robot communication and bare-metal/RTOS software on a microcontroller using micro-ROS for communication with a single-board computer, while hardware I/O handles communication with the robot. After the best configuration is determined several hardware options are evaluated to determine the optimal setup.

3.2 Requirements

3.2.1 Use case

The software tool is aimed towards students, researchers, and engineers that want to create real-time networked robotic systems following a model-driven development methodology, while minimizing the complexities involved with deploying the model on an embedded device. An example scenario is a student tasked to get a robot in the RaM laboratory operational using ROS 2 for command and monitoring.
The robot is a 2-DoF manipulator that receives PWM input signals for motor control and has encoder output signals to determine the robot position. The robot hardware and its I/O are known, but robot code should be developed and implemented for the robot functionality.
The student starts with creating a model of the robot in 20-sim. The model incorporates the control software, hardware I/O, and the dynamic system behaviour.
After simulations to verify model correctness, the student uses C-Code generation to generate code for the control software. Transitioning from model to deployment reveals several challenges:

1. **Hardware interfaces:** The 20-sim control software must be used to control the robot. Since the robot only has hardware I/O interfaces, the student needs an embedded device that can implement the control software and has functionality to interact with the hardware I/O. The student integrates the hardware I/O with the control software by configuring device GPIO pins with implementations for PWM signals to steer the robot and quadrature encoders to get the current position information. Because the student wants to minimize delays between the control software and hardware I/O they are tightly integrated.
2. **Firm real-time control:** For correct functioning the control software is bound to firm real-time constraints (occasional deadline misses are acceptable, but frequent misses degrade performance and can lead to system failure) and performs calculations using double-precision floating-point numbers. Consequently, the student must ensure that the embedded device supports real-time processing and double-precision floating-point calculations.
3. **Soft real-time ROS 2 communication:** Furthermore, the robot must use ROS 2 for command and monitoring. Therefore, the student must create a link between the control software and ROS 2 to establish soft real-time (occasional deadline misses are allowed without causing critical failure, but reduce the usefulness of the result) communication.
4. **Testing and debugging:** After implementing the hardware interfaces, firm real-time control, and ROS 2 communication, while ensuring each functions without interfering with

the others the student can start with testing and debugging.
The student makes sure that the code running on the embedded device can be debugged in a development environment. Setpoint commands are sent using ROS 2 publishers and ROS 2 subscribers are used to monitor the current robot position to verify correct behaviour.

5. **Refinement:** If the robot does not behave as expected, the student must refine the code. This refinement can include remodelling parts of the robot system or improving the embedded device operations. Through multiple development iterations, the student ultimately gets the robot working as intended.

The goals in Section 1.2 form the basis for the use case, which outlines a scenario addressing these goals. From the complexities identified in the use case, as well as additional requirements not directly covered in the scenario but necessary for the development of the EPG, the functional and non-functional requirements of the software tool are derived. These requirements, explained in Section 3.2.2 and Section 3.2.3, are structured to support the achievement of the goals and are prioritized using the MoSCoW convention, starting with musts, followed by shoulds, and concluding with coulds.

3.2.2 Functional requirements

1. **The EPG-generated project *must* use the ROS 2 ecosystem.**
ROS 2 is currently one of the most popular open-source robotic frameworks. There are already numerous ROS 2 software packages, offering functionality that can be directly used to support development. To ensure a seamless integration, the EPG-generated project must be designed to use the ROS 2 ecosystem.
2. **The EPG-generated project *must* support a robotic setup that uses a Raspberry Pi single-board computer.**
The Raspberry Pi is a small but powerful single-board computer making use of an ARM processor. This computer is widely used and supported within the robotics industry. Besides that, the RaM robotics laboratory makes use of the Raspberry Pi. The EPG-generated project must support a robotic setup that uses a Raspberry Pi single-board computer.
3. **20-sim model-generated code of embedded control software *must* be directly implementable on mechatronic systems.**
The EPG directly implements mechatronic control software following from a modelling tool. Because of the intensive use of 20-sim at the University of Twente in education and research along with its functionality for multi-domain modelling, simulation, and C-Code generation from model elements, 20-sim must be supported by the EPG.
4. **The EPG-generated project *must* have real-time capabilities.**
Both soft real-time and firm real-time constraints must be supported. The EPG-generated project must support the robots in the RaM laboratory. Some of the robots use a webcam that provides images at 30 fps (33 ms), with the images used for setpoint generation and system monitoring at the same rate. Therefore, soft real-time communication for transmitting setpoint and monitoring data is required with a 33 ms time interval. The robot control software must run in a loop with a 1 ms time interval, which is a typical rate used by the robots that balances responsiveness and computational efficiency. Because of that, firm real-time performance is required with a 1 ms time interval.
5. **There *should* be compatibility with different mechatronic systems.**
The EPG should be functional for different mechatronic systems, which makes it important to make the system general and not to design it specifically for a single application.
6. **Support for networking between robotic components *should* be implemented.**
Support for networking between robotic systems should be implemented. Robotic systems often consist of components that work together to achieve the desired system beha-

viour. Multiple robotic components should be able to collaborate within this distributed control system.

7. **There *should* be support for double-precision floating-point calculations.**
   Double-precision floating-point calculations are often used in control software. When available, hardware support for double-precision floating-point numbers should be used. Otherwise, floating-point calculations should be performed in software, which is less efficient.

3.2.3 Non-functional requirements

1. **Minimal user input *should* be required.**
   Minimal user input should be required, to enable a smooth transition from model to deployment with as much automation as possible.
2. **The EPG *should* have good maintainability.**
   The EPG should be easy to maintain. When software is difficult to maintain or when certain parts are quickly outdated, this will result in a short lifespan for the EPG. Therefore, the following aspects should be taken into account:
   - **Limit EPG complexity**
     When the EPG is complex, this will make it difficult to get a good overview of how it can be maintained.
   - **Limit custom software**
     Custom software solutions often make it difficult to maintain a software tool in the long term. These solutions do not have an active developer community and a small number of users, which makes the developer solely responsible for maintenance and further developments. Therefore, it is preferred to use active open-source initiatives. This involves several developers who actively help develop and improve the software.
3. **There *should* be support for the use of software components.**
   There is a trend within robotics in which software components are increasingly used. The advantage of using software components is that they can be reused by the end user within different projects. It is therefore desirable that reusable software can be used.
4. **The EPG and EPG-generated project *could* use common programming languages.**
   Using uncommon programming languages complicates the simplicity with which the EPG and EPG-generated projects can be used. The 20-sim generated software is written in C/C++, which could be used for EPG-generated projects to ensure consistency. The EPG user primarily interacts with the EPG-generated project, however for convenience in maintainability the EPG could also use a common programming language.

3.3 Selection of the software/hardware configuration

3.3.1 Introduction

A design-space exploration is used to select the best software/hardware configuration for the EPG. Best configuration implies that the configuration has the most potential to meet the previously mentioned requirements.
The design-space exploration is performed by creating decision matrices. The scores in the decision matrices are given by using the symbols $--$, $-$, $-/+$, $+$ and $++$, which represent a linear range between -2 and 2. For each alternative the scores of the criteria are multiplied with a weight representing the importance of the criteria. These multiplications are subsequently summed to get a final score for each of the alternatives in the decision matrix.

3.3.2 Configurations

For the design of the EPG both software and hardware should be taken into consideration, since a choice for one constraints the options for the other. The most common software/hardware combinations supporting real-time performance are:

- Custom VHDL/Verilog software running on an FPGA.
- Real-time patched Linux software with C/C++ code running on single-board computer hardware optionally in combination with an FPGA for hardware I/O.
- Bare-metal/RTOS software with C/C++ code running on microcontroller hardware.

Only the last two combinations are considered in this analysis. The first option is not considered, because of the following complications:

- 20-sim must be used for generating the embedded control software. 20-sim only has options for generating C/C++ code and not VHDL/Verilog code. A workaround could be to create a High-Level Synthesis (HLS) solution that converts C/C++ to FPGA code. This would however result in combining multiple programming languages, creating a vendor-specific solution, and shifting from sequential software thinking to parallel hardware, which adds complexity.
- Using an FPGA for networking with ROS 2 is not straightforward and would require an FPGA-specific bridge between the FPGA logic and ROS 2 implementation.
- FPGA development requires special expertise, making an FPGA-based software tool harder to maintain.

The result would be a complex, difficult-to-maintain software tool.

Single-board computer with real-time Linux and FPGA hardware I/O

Previous projects at RaM supporting real-time networked robotic software used Xenomai patched Linux with custom ROS 2 networking solutions, running on a Raspberry Pi 4B single-board computer with an IcoBoard FPGA for hardware I/O (Meijer, 2021; Raoudi, 2024). Meijer (2021) showed that Xenomai is the preferred real-time Linux patch for single-board computer implementations, as it offers better timing performance compared to other patches, while using identical programming interfaces.
Figure 3.1 shows a preliminary mapping of the embedded control system layers within this setup. All tasks are performed on the Raspberry Pi using a combination of ROS 2 and Xenomai. The soft real-time ROS 2 tasks use the normal Linux kernel on three separate cores. The firm real-time tasks are executed using the Xenomai patched kernel on the remaining core. Since the kernels run on separate processing cores a cross-buffer is needed to enable communication between both execution environments. For hardware I/O the Raspberry Pi communicates with an Ico-Board FPGA over SPI to steer actuators and read sensors. A user interface can be added to the system to monitor the system state by communicating with ROS 2.

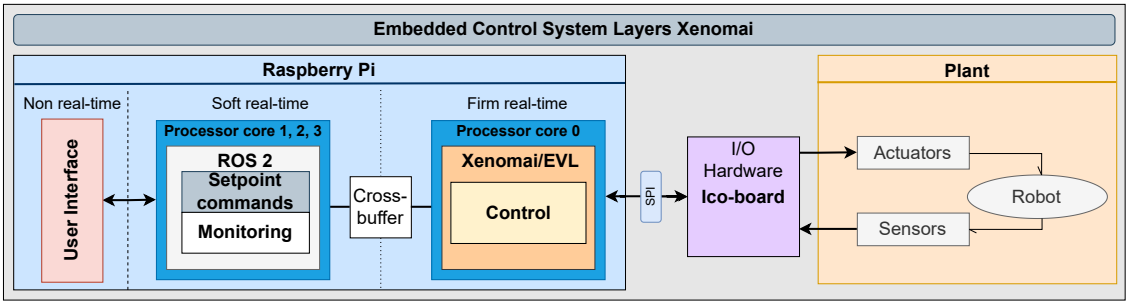


Figure 3.1: Embedded Control System Layers Xenomai

Single-board computer and microcontroller with bare-metal/RTOS

The alternative proposed in this thesis is to use micro-ROS on a microcontroller with a bare-metal or RTOS implementation for real-time performance to control a robot and using a Raspberry Pi 4B single-board computer for non-real-time processing. Micro-ROS is established as the standard for ROS 2 based microcontroller communication and extends the ROS 2 concepts to resource constraint microcontroller devices while ensuring compatibility with all ROS 2 concepts, making it the most suitable option for a microcontroller based software tool.
Figure 3.2 shows a preliminary mapping of the embedded control system layers using a micro-ROS setup. This figure shows the same elements as Figure 3.1, but mapped at different locations and hardware. The non real-time and soft real-time processing are still present at the Raspberry Pi. The firm real-time and I/O are however mapped on the microcontroller. Micro-ROS communication between the Raspberry Pi and the microcontroller takes place within the ROS framework. The microcontroller can be connected to the Raspberry Pi making use of the XRCE-DDS protocol. The microcontroller has functionalities to create PWM signals and read encoders. Real-time performance is guaranteed by making use of an RTOS or a bare-metal solution on the microcontroller.

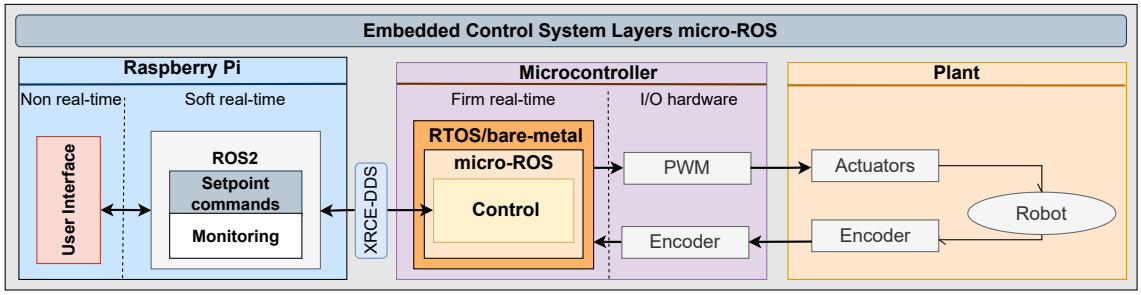


Figure 3.2: Embedded Control System Layers micro-ROS

Evaluation of Xenomai and micro-ROS

This exploration compares a setup using Xenomai with a setup using micro-ROS running on a microcontroller to discover the strengths and limitations of each.
Xenomai and micro-ROS are both software frameworks that greatly impact the design of a software tool. They are not directly comparable. Xenomai is a real-time patch for Linux, while micro-ROS is a framework for enabling ROS 2 communication on microcontrollers. They do however have in common that using them creates constraints on real-time behaviour, the availability of system resources, and the way in which robotic applications can be designed. A robotic application is built on the functionality the software framework provides, and solutions must be developed for any shortcomings. Therefore, it is important to evaluate the influence of the functionality and imposed constraints of these frameworks on the design requirements. The software frameworks are evaluated on the following points:

- **ROS 2 compatibility:** Xenomai does not have ROS 2 support for software that runs real-time. Therefore, a custom solution is needed to enable communication from tasks running real-time to non-real-time tasks running ROS 2. Micro-ROS is developed to be fully compatible with all ROS 2 concepts and is tightly integrated.
- **Modularity:** System modularity makes it possible to flexibly adjust and scale a robotic system. When the setup is represented as a component that contains all functionality, multiple independent components can be composed to create a distributed control system.
  Xenomai performs all processing on the Raspberry Pi, with different tasks mapped to the cores of the Raspberry Pi. There are however restrictions when creating a distributed system. Preferably one processing core is used for each control task to have a clear sep-

aration of concerns.
Besides that, the IcoBoard has a limited amount of I/O coverage. The work of In 't Veld (2023) implemented a Xenomai approach for a production cell application containing six units, each equipped with a motor and encoder controlled by a control loop. The research showed that the ideal mapping required one Raspberry Pi 4B with IcoBoard for every two units resulting in a total of three Raspberry Pis with Icoboards. While this setup can be effective for smaller systems, it becomes increasingly complex for larger systems, requiring multiple Raspberry Pis with IcoBoards.
Using micro-ROS all functionality to control a plant runs on a microcontroller performing the computations. Multiple microcontrollers controlling plants can be connected to the Raspberry Pi, creating a multicomponent distributed computing system with a separation of concerns. The Raspberry Pi only has to steer and monitor the system components by publishing and subscribing on ROS 2 topics, while most computational power can be used to perform other tasks. In this setup the Raspberry Pi handles intensive computations, while microcontrollers manage the control tasks making efficient use of powerful hardware.

- **Real-time performance:** Xenomai is designed for real-time performance with low latencies making it suitable for real-time applications. A micro-ROS setup, on the other hand, uses bare-metal interrupt timers or a micro-ROS compatible RTOS to obtain real-time performance. The expectation is that both solutions will have firm real-time performance meeting the requirements. Xenomai uses a co-kernel architecture, where the Linux kernel might interfere with real-time tasks in heavy load situations, though the higher processing speed at which Xenomai processes real-time tasks might reduce jitter. The RTOS implementation is more minimalistic and deterministic, operates without interference from non-real-time tasks, and offers simplified management of real-time tasks.
- **Hardware I/O support:** The Xenomai setup uses the IcoBoard FPGA for hardware I/O. The Raspberry Pi communicates over SPI to set the PWM and read encoder values. The SPI communication can however be a performance bottleneck and is problematic with some Linux kernels patched with Xenomai that cripple USB and SPI functionality.
Using micro-ROS all hardware I/O on the microcontroller and single-board computer is available without restrictions. A plant is directly controlled by the microcontroller which immediately executes the control software.
- **Processing power:** Both Xenomai and micro-ROS setups make use of a Raspberry Pi 4B. The Xenomai setup performs all computation on the Raspberry Pi 4B. On the other hand, the micro-ROS setup includes a microcontroller to control plants, distributing the workload over different devices. Therefore, while the Xenomai setup has more processing power available to control the plant, the micro-ROS setup better distributes the power where it is needed.
- **Independence from custom software:** Xenomai offers a real-time framework, but needs custom solutions for enabling networking in ROS and communicating with the FPGA for hardware I/O. Therefore, multiple custom solutions are needed. Using a micro-ROS implementation requires no dependence on custom software. All elements are present for networking and real-time performance. The challenge is to efficiently use resource constraint hardware and optimise performance.
- **Development complexity and maintainability:** Using Xenomai is not straightforward. Xenomai is implemented by building a Linux kernel enabling the Xenomai core. Since Xenomai is not compatible with all Linux kernels and the provided kernels are often not up-to-date with the latest Linux kernel, older Linux kernel versions are used limiting the implementation of new features, security patches, and operating system updates. When a Xenomai patch is available for a new version of the Linux kernel, problems can arise, because not all functionalities provided by the kernel work yet, requiring additional patches

and solutions. Micro-ROS can be used by adding the micro-ROS module to a microcontroller project using CMake. Micro-ROS is up-to-date with the latest ROS 2 version and uses the same concepts as ROS 2. Additionally, a bare-metal or RTOS implementation is needed for firm real-time performance, with multiple up-to-date solutions available.

- **Documentation:** Having a good documentation of the software framework is essential for being able to develop applications. The documentation of Xenomai is limited and does not keep a good separation between different Xenomai versions. Micro-ROS documentation is up-to-date, contains all information needed to create applications, and builds forward on the extensive ROS 2 documentation by using the same concepts.
- **Cost:** Both Xenomai and micro-ROS setups use a Raspberry Pi 4B. Using Xenomai requires an IcoBoard, while micro-ROS needs a microcontroller. Table 3.1 lists example hardware and costs. Using these costs, Table 3.2 lists the cost of a base configuration showing that the Xenomai base setup costs about twice as much as the micro-ROS setup, and about five times as much for the production Cell case. Overall, the cost of a Xenomai setup will be significantly more compared to a setup using micro-ROS.

Table 3.1: Hardware prices at 17-11-2024

	Raspberry Pi 4B 8GB	**IcoBoard**	**Raspberry Pi Pico (microcontroller)**
Price (€)	82,27	100,39	3,95
Vendor	Kiwi electronics	DigiKey	Kiwi electronics

Table 3.2: Cost comparison base setup

Case	**Setup**	**#Raspberry Pi 4B**	**#IcoBoard**	**#Raspberry Pi Pico**	**Total Cost (€)**
Base	Xenomai	1	1	0	183
Base	micro-ROS	1	0	1	86
Production Cell	Xenomai	3	3	0	548
Production Cell	micro-ROS	1	0	6	106

All evaluation points are summarised in Table 3.3.

Table 3.3: Comparison setups

	Xenomai/Evl	**micro-ROS**
ROS 2 compatibility	Custom solutions	Tightly integrated
Modularity	Increasingly complex for larger systems, requiring multiple Raspberry Pis with IcoBoards	Raspberry Pi handles intensive computations, while microcontrollers manage control tasks efficiently using powerful hardware
Real-time performance	Firm real-time co-kernel	Firm real-time RTOS/bare-metal
Hardware I/O support	I/O using IcoBoard FPGA over SPI	All microcontroller I/O available
Processing power	More processing power for robotic plant control	Better distribution of power where it is needed
Independence from custom software	Custom solutions for ROS 2 and FPGA communication	No dependence on custom solutions
Development complexity and maintainability	Complex patched Linux kernel	Add micro-ROS and firm real-time software to microcontroller project
Documentation	Limited	Adequate
Cost	High	Low

In order to qualitatively compare the setups, a decision matrix is created as shown in Table 3.4. The matrix covers the evaluation points shown in Table 3.3. The weights are assigned to the table such that the elements that are most important for meeting the design requirements have the highest weight. This gives an estimate on how the elements are represented in the setups.

Table 3.4: Qualitative Comparison setups

	Weight	**Xenomai/EVL**	**micro-ROS**
ROS 2 compatibility	**3**	-	++
Modularity	**3**	-/+	++
Real-time performance	**3**	++	+
Hardware I/O support	**2**	+	++
Processing power	**2**	+	+
Independence from custom software	**2**	- -	++
Development complexity and maintainability	**2**	-	++
Documentation	**1**	-	+
Cost	**1**	- -	++
Total:		**-2**	**32**

Conclusion evaluation

The results in Table 3.4 show that using micro-ROS for the development of the EPG has benefits over using Xenomai/EVL. Therefore, it can be concluded that using micro-ROS on a microcontroller in combination with a Raspberry Pi creates the best setup to reach the design requirements. However, it is not yet clear which microcontroller is most convenient to use within this setup.

micro-ROS compatible hardware

Several microcontrollers are micro-ROS compatible. Therefore, it is investigated which microcontroller is most suited to meet the design requirements by comparing the different options and investigate their strengths and weaknesses. The microcontrollers that are taken into consideration are shown in Table 3.5.

Table 3.5: Comparison micro-ROS compatible hardware

	STM32-F767ZI	**STM32-H743ZI**	**RPi Pico**	**RPi Pico 2**	**Teensy 4.0/4.1**	**Portenta H7**
Clock speed	216 MHz	480 MHz	133 MHz	150 MHz	600 MHz	480 MHz + 240 MHz
Double-precision FPU	Yes	Yes	No	No	Yes	Yes
Dual core	No	No	Yes	Yes	Yes	Yes
Processor (Arm Cortex)	M7	M7	2X M0+	2X M33	M7	M7 + M4
Flash memory	2MB	2MB	2MB	4MB	8MB	16MB
(S)RAM	512 Kbyte	1 Mbyte	264 Kbyte	520 Kbyte	1 Mbyte	8MB
Operating system	FreeRTOS, Zephyr	bare-metal, FreeRTOS, Zephyr	bare-metal, FreeRTOS, Zephyr	bare-metal, FreeRTOS	Arduino OS	Arduino OS
Approximate price (€)	30	30	4	6	40	100
Programming language	C/C++	C/C++	C/C++	C/C++	Arduino C	Arduino C
Debugging	ST-LINK	ST-LINK	Serial Wire	Serial Wire	No debugging	Serial Wire
Extensive Development community	Active	Active	Very active	Very active	Niche	Niche

When considering the hardware in Table 3.5 the following considerations can be made for each of the microcontroller types:

- **STM32:** The STM32-F767ZI and STM32-H743ZI are boards from the STM32 microcontroller family, both featuring a double-precision FPU, powerful processor, and I/O cap-

abilities, making them interchangeable for many applications. However, the STM32-H743ZI offers more powerful hardware. The boards are compatible with both FreeRTOS and Zephyr RTOS, and are priced at approximately €30,-.

- **Raspberry Pi Pico:** The Raspberry Pi Pico supports micro-ROS and can be used bare-metal, with FreeRTOS and Zephyr RTOS. The Pico lacks a double-precision FPU, so double-precision calculations happen in software optimised by the compiler. It does however have two ARM Cortex-M0+ cores clocked at 133MHz, enabling efficient multitasking. The Raspberry Pi Pico, priced at about €4, is one of the most powerful cheap microcontrollers.
- **Raspberry Pi Pico 2:** At the beginning of this thesis the Raspberry Pi Pico 2 was not yet available, but because of its interesting properties it has been added to the analysis. The Raspberry Pi Pico 2 is the more powerful successor of the Raspberry Pi Pico. It retains the same compatibility as the Pico, with a more powerful dual-core processor running at 150 MHz and a single precision FPU. The Pico 2 has a similar price as the Pico of about €6,-.
- **Teensy 4.0/4.1:** This is a powerful microcontroller board featuring an ARM Cortex-M7 at 600 MHz and supports double-floating precision. The price is about €40,-. This board makes use of the Arduino framework. Therefore, it is uncertain if the 20-sim generated C/C++ code can run on the microcontroller without modifications. Teensy is an interesting project, but there are some downsides. Debugging is for example difficult, because there is no option to directly connect a debugger to the device. There are some options to debug using commercial software on Windows such as Visual Micro, but this is inconvenient. Also, there is little documentation about running an RTOS such as FreeRTOS on the Teensy. These limitations make development difficult. The Teensy is a niche product and therefore does not have a large development community, which can complicate finding solutions to issues.
- **Arduino Portenta H7:** This board uses a dual-core version of the STM32H747. It is quite expensive at €100,- and there are not a lot of advantages compared to the regular STM32 boards. The disadvantages are comparable to the Teensy. The Portenta is also an Arduino product, which raises uncertainty about whether the 20-sim generated C/C++ code will work without modifications. It is an Arduino product and therefore enjoys support from that community, but within the Arduino world it is a niche product.

Taking these aspects and the design requirements into account a comparison matrix has been created, which is shown in Table 3.6. In this comparison the following evaluation criteria are taken into consideration:

- **Clock speed:** The microcontroller needs to have sufficient speed to run control loops, while simultaneously performing other tasks, such as communication.
- **Double-precision FPU:** A double-precision floating-point unit performs double-precision calculations in specialized hardware instead of using processing cycles accelerating these types of calculations.
- **Dual-core processor:** Having two processing cores enables multitasking by dividing workload over both processor cores. Therefore, even with a lower clock speed of each core the same performance can be reached as a single-core microcontroller running at a higher clock speed.
- **RTOS support:** RTOS support ensures firm real-time performance by task scheduling and resource management, providing advantages in multi-task situations.
- **Programming language support:** Programming language support is important, since the programming language has a large influence on how convenient the framework is to work with. The STM32 and Pico are known for their good support of C/C++. The other microcontrollers should also support this, but because they are niche products more complications can arise.

- **Approximate price:** The price is taken into account to create a trade-off between performance and price. Since this setup can conveniently use multiple microcontrollers to create a distributed system it is desirable that adding additional microcontrollers is not too expensive.
- **Debugging:** Debugging capabilities are needed to quickly resolve issues during code development.
- **Flash memory and (S)RAM:** Sufficient flash memory and (S)RAM are needed to flash the compiled program, for micro-ROS to function, and to smoothly run code without running out of memory.
- **Extensive development community:** An extensive development community provides a platform to discuss hardware specific problems and quickly find solutions. In an active community, many users work with the same hardware improving its support and functionality. Joining an active community directly provides benefits from the collective effort and expertise.

This qualitative comparison shows that the STM32 Nucleo-H743ZI and Raspberry Pi Pico 2 are the best options considering the requirements, followed by the STM32 Nucleo-F767ZI and Raspberry Pi Pico. Both the Pico 2 and STM32 Nucleo-H743ZI are interesting boards that can be used for the EPG. The STM32 Nucleo-F767ZI is also good, but has little added value compared to the STM32 Nucleo-H743ZI. The Raspberry Pi Pico 2 has no double-precision FPU, which decreases its quality in the comparison, but does on the other hand have a powerful dual-core processor allowing for multitasking designs.

Table 3.6: micro-ROS compatible microcontroller qualitative comparison

	Weight	STM32-F767ZI	STM32-H743ZI	RPi Pico	RPi Pico 2	Teensy 4.0/4.1	Portenta H7
Clock speed	**3**	-/+	+	-/+	-/+	+	+
Double-precision FPU	**3**	+	+	-	-	+	+
Dual core	**3**	-	-	+	+	+	+
RTOS support	**3**	++	++	++	++	-/+	-/+
Programming language	**3**	+	+	+	+	-/+	-/+
Approximate price	**2**	+	+	++	++	-	- -
Debugging	**2**	++	++	+	+	- -	-/+
Flash memory	**1**	-	-	-	-/+	+	++
(S)RAM	**1**	-/+	+	-	-/+	-	++
Extensive development community	**1**	+	+	++	++	-	-/+
Total:		15	19	15	17	2	9

4 Design

4.1 Introduction

This chapter presents the design and design considerations of the Embedded-Project Generator (EPG). The EPG is a software tool for the transition from 20-sim model-generated code to a deployable EPG-generated project.
The diagram of the design context in Figure 4.1 provides insight into the outcomes of this chapter. From an EPG user perspective, this can be summarised as follows:

- In the 20-sim C-Code generator the user selects the Embedded-Project Generator. By doing so, 20-sim generates C-Code from the control component of the 20-sim model and starts the EPG.
- The EPG takes the 20-sim generated C-Code as input and follows the EPG steps:
  1. Select target: The user selects a target configuration, which defines the microcontroller and its I/O target ports.
  2. Connect ports: The user connects 20-sim model ports with target ports.
  3. Code generation: The user starts EPG code generation resulting in an EPG-generated project.
- After EPG code generation, the user compiles the EPG-generated project and flashes the resulting binary directly on a microcontroller target.
- The microcontroller target is ready to be used with command & monitoring from a single-board computer running ROS 2 to control the robot.

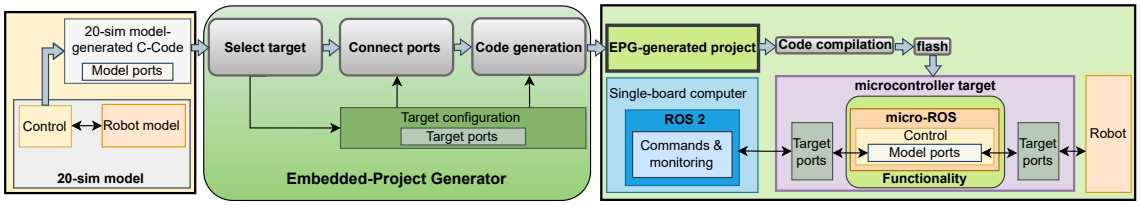


Figure 4.1: Embedded-Project Generator design context

The chapter begins by addressing the first thesis goal: Design a method to automate the translation from a robot-controller model to code that uses the ROS 2 ecosystem and supports real-time control. To provide insight into the expected output of the EPG, the structure of an EPG-generated project is explained. In addition to the code generated by 20-sim, the EPG needs to know how to connect the 20-sim model ports to the corresponding microcontroller target ports. The information about the target ports is described in the target configuration file. The microcontroller embedded I/O target ports are connected with the robot and the network I/O target ports are, via micro-ROS, connected with a single-board computer.
With all necessary information now available, the EPG steps are discussed, outlining the process from EPG input to output, including selecting a target configuration, connecting model ports with target ports, and generating the code.
With the first goal elaborated, the second thesis goal is addressed: Design and implement a software tool based on the automation method. The implementation of the automation method in the software tool involves determining the appropriate programming language, defining how the EPG steps are executed, structuring an EPG software tool package, and specifying how different parts of the automation method will be realised.
Subsequently, the EPG implementation of targets selected in Analysis Chapter 3 is discussed. The EPG software tool is already addressed, but needs target configurations to demonstrate that it functions as expected. This discussion highlights how the targets can be configured to

align with the system requirements for real-time processing, micro-ROS communication with a single-board computer, and hardware I/O for robot control.
The chapter concludes with a verification of the functional and non-functional requirements.
In this chapter topics are used to present design considerations or motivations. For each design consideration topic, the pros and cons are evaluated alongside a decision and realisation. When a decision is made, the pros and cons guide the choice, and the realisation describes its implementation. Similarly, for each design motivation topic, a motivation explains why a particular approach is used, and the realisation outlines how it is implemented.

4.2 Automation method

The automation method for translating a robot-controller model to code using the EPG is structured by starting with the final output: the EPG-generated project. To ensure compatibility with 20-sim control software, an RTOS, and micro-ROS, first the design of the EPG-generated project is established. Subsequently, the necessary steps to achieve this output are outlined. The target configuration has to be constructed and saved, leading to the decision whether this configuration is selected within 20-sim or the EPG. Finally, the EPG steps, a sequence for target selection, port connection and project generation are defined.

4.2.1 EPG-generated project

Design motivation topic: Design the structure of the EPG-generated project so that 20-sim control software operates together with an RTOS and micro-ROS to achieve the desired functionality of the target.
Motivation: Since support for software components is required, the EPG-generated project is organised as a component that implements all target functionality and I/O interfaces. This implies implementing control software with an RTOS in the component functionality and using micro-ROS network I/O and embedded I/O interfaces to interact with the component functionality. There are no other alternatives that meet this requirement.
Realisation: The EPG-generated project deployed on a microcontroller, enables the microcontroller to function as a component that can interact with a single-board computer and a robot using component I/O. This component integrates all functionality to work stand-alone and uses both networking and embedded interfaces to handle component I/O, simplifying system assembly and maintenance.
After the EPG-generated project is compiled, the resulting binary is directly flashed on a microcontroller as shown in Figure 4.2. The microcontroller uses embedded I/O for hardware-level interaction with a robot, while network I/O using micro-ROS integrates the component into the ROS 2 ecosystem for command and monitoring.
Real-time management of the 20-sim control software is realised by implementing support for micro-ROS compatible open-source RTOSes FreeRTOS, Zephyr, and bare-metal implementations.
The microcontroller is now a component that integrates functionality, including real-time management, micro-ROS for resource constrained communication, and 20-sim control software for controlling the robot. Seen from ROS 2 the component is a node with all system complexities abstracted away, leaving only the node's subscriber and publisher topics for steering and monitoring the robot.

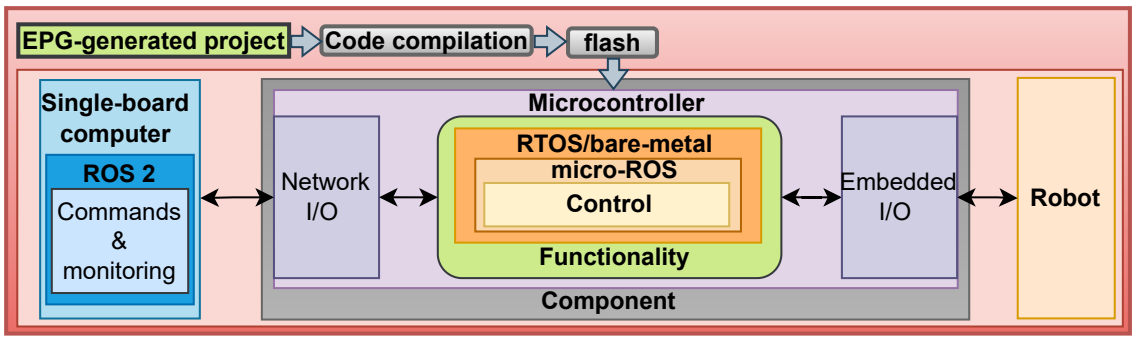


Figure 4.2: EPG project deployment

4.2.2 Target configuration

The target configuration is the process of defining and storing the setup of a target, which describes how the 20-sim control software on the microcontroller communicates via embedded I/O ports with the robot and network I/O ports with a single-board computer.
Before configuring the EPG target, it is essential to determine whether the EPG target is selected within 20-sim or in the EPG itself.

Design consideration topic: Determine how the target configuration is constructed and stored.
Options:

- Create a user interface to handle the entire configuration and store user choices.
  Pros: The user does not have to prepare the configuration in advance but is assisted by the GUI.
  Cons: Requires user expertise of the configurable parameters of the target. Also, it is difficult to account for all possible configuration choices during the development of the GUI, which may require the GUI to be expanded and redesigned at a later stage for a new target.
- Use a device-tree approach where all configuration takes place in a target configuration file.
  Pros: Predefined configurations can be created for targets requiring less user expertise. A predefined configuration structure makes it easier to add new targets, because it can be flexibly extended.
  Cons: A device tree requires manual adjustments when adding or updating a target.

Decision: Users with minimal knowledge should be able to use the tool, which is why a device-tree approach is chosen. Pre-configured target configuration files can be used without any knowledge of mechatronics.
Realisation: A target is defined by a target configuration file (.tcf), which is structured like a device tree (see Section 2.5). Each .tcf file describes, for a specific microcontroller, the embedded I/O ports used to interact with a particular robot and the network I/O ports for communication with a single-board computer.
Figure 4.3 shows an example, where two components are used to pass a setpoint from a ROS 2 topic to the motor of a robot. The yellow/blue squares in the figure represent input/output ports. A micro-ROS subscriber target component, subscribed to a ROS 2 topic, receives setpoints. The setpoints are provided via the output port of this component to the input port of the model-based control software. Via the output port an updated PWM value is set to an input port of a PWM target component, resulting in signals controlling the motor of the robot.
In this way, target functionality can be composed by combining multiple target components. Each target component instance can be distinguished using customisation settings in the .tcf file. In the .tcf file, each component instance is defined by its name, port name, EPG-module, and EPG-module parameters. The EPG-module is a code template file with tokens that serve as placeholders. During code generation, the placeholders are replaced with component-specific information, such as names and settings derived from EPG-module parameters in the .tcf file. This process produces component-specific code defining the component's functionality.

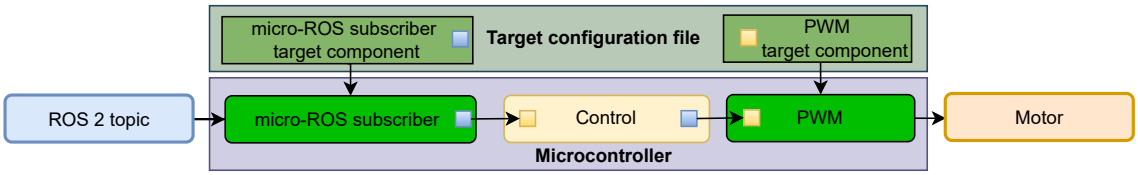


Figure 4.3: Example target component functionality

Design consideration topic: Determine where the EPG-target configuration is selected: in 20-sim or in the EPG.
Options:

- Select the EPG-target configuration in the 20-sim C-Code generation menu.
  Pros: There is no need to make 2 choices, because choosing the EPG is combined with selecting the EPG-target configuration.
  Cons: By choosing the EPG-target configuration in 20-sim, the clear separation between modelling and target-specific implementation is broken.
- In 20-sim, the EPG is selected, and after code generation, the EPG starts automatically, allowing the EPG-target configuration to be selected.
  Pros: Maintains a clear separation of concerns. 20-sim focuses on modelling, while the EPG handles target-specific functionality. This aligns with 20-sim's existing approach, where general, non-target-specific C-Code is generated and imported into 20-sim 4C for target-specific execution.
  Cons: In 20-sim, the EPG is selected, followed by selecting the target in EPG, making it a two-step process.

Decision: It is chosen to select the EPG in 20-sim and the EPG-target configuration in the EPG. This maintains a clear separation of concerns between the tasks. 20-sim remains focused on modelling and simulation, and the EPG is responsible for generating the functionality required for a target.

4.2.3 EPG steps

The EPG takes 20-sim model-generated code and a .tcf file as input. The output is an EPG-generated project, which after compilation is deployed on a microcontroller. The microcontroller interacts with the robot through embedded I/O ports and communicates with a single-board computer via network I/O ports.
To achieve the desired output from the given input, the following EPG steps must be executed:

- **Select target - Identify the target by selecting the appropriate .tcf file:** Define the target system, including the microcontroller, RTOS, and available I/O ports.
- **Connect ports - Connect the 20-sim model ports with the target ports:** Establish connections between the 20-sim model ports and the microcontroller's embedded and network I/O ports.
- **Code generation - Combine all information and generate an EPG project:** Integrate the control code, port connections, and target configuration into a complete project ready for deployment.

Design motivation topic: Ensure an efficient sequence of EPG steps for select target, connect ports, and code generation.
Motivation: The order of the EPG steps has been chosen to ensure a logical and efficient sequence for configuring and generating the EPG project. Figure 4.4 shows a schematic of the sequence of steps, which is motivated as follows:

1. **Select target:** This step is performed first because the .tcf file defines the specific microcontroller configuration for a given robot. Without this, it is not possible to proceed with connecting ports or generating a project, as the microcontroller's functionality and hardware interactions depend on the target definition.
2. **Connect ports:** After the target is identified, the next logical step is to connect the 20-sim model ports to the microcontroller's I/O ports. This connecting ensures that the control software can interact with the robot through embedded I/O ports and communicate with the single-board computer via network I/O ports. This step depends on the information in the .tcf file, as it defines the available ports and their functionalities.
3. **Code generation:** Once the target is identified and the ports are connected, the final step is to combine the port connection information, control code, and the target configuration to generate the EPG project. This ensures that the generated project is complete and ready for deployment on the specific target system.

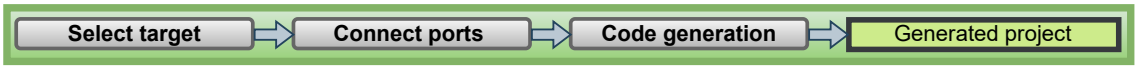


Figure 4.4: EPG steps

4.3 Software tool implementation

The implementation of the EPG software tool is based on the automation method. To realise the translation from a robot-controller model to deployable code, first a programming language is selected for developing the EPG. Subsequently, the implementation of the EPG steps, which include target selection, port connection, and code generation, is outlined. Then, the structure of the EPG package is addressed, which is designed to reflect the EPG steps while maintaining modularity and clarity. Finally, the code generator used in the EPG's code generation step is discussed. It generates a project that, after compilation, can be directly flashed on a microcontroller.

4.3.1 EPG programming language

Design consideration topic: Determine the programming language for developing the EPG.
Options:

- A C/C++ program implementing the code generation method.
  Pros: The C language is used in both 20-sim generated code and the microcontroller project code. Developers only need knowledge of C/C++.
  Cons: Developing a software tool in C/C++ has a high development time. It is difficult to manage cross-platform compatibility.
- A Python package implementing the code generation method.
  Pros: Python is a cross-platform programming language that enables structured development of the EPG. Python standard libraries allow building the EPG without external dependencies. Python is easy to use, allowing for rapid development.
  Cons: By using Python an additional programming language is used.

Decision: Python is used to develop the EPG. Although it involves the use of an additional programming language, the benefits of cross-platform support, flexible standard Python libraries, and rapid development outweigh this drawback. By structuring the EPG as a Python package, it can be conveniently installed with one command and requires minimal user actions to get everything up and running.

4.3.2 EPG steps

Design consideration topic: Determine how to select the target and connect the 20-sim model ports to the target ports.
Options:

- Text file with target selection and port connections.
  Pros: Low development overhead.
  Cons: Error-prone and not user-friendly.
- Command-line tool.
  Pros: Low development overhead.
  Cons: Error-prone and not user-friendly.
- GUI.
  Pros: User friendly and intuitive.
  Cons: Requires more development effort.

Decision: A GUI is used to select a target and connect the ports, because it is user friendly.
Realisation: The EPG steps result in a generated project for a target. An overview of the EPG steps is shown in Figure 4.5:

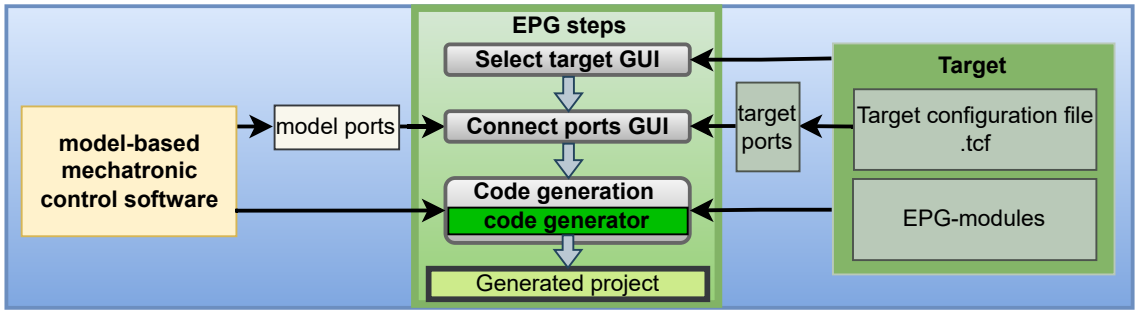


Figure 4.5: EPG steps overview

The following sequence of actions is executed in the EPG steps:

1. **Select target GUI:** The EPG steps start with a select target step using EPG GUI, a graphical user interface designed to facilitate the project generation process. The GUI displays all available .tcf files, allowing the user to select the desired configuration. This is shown in Figure 4.6.

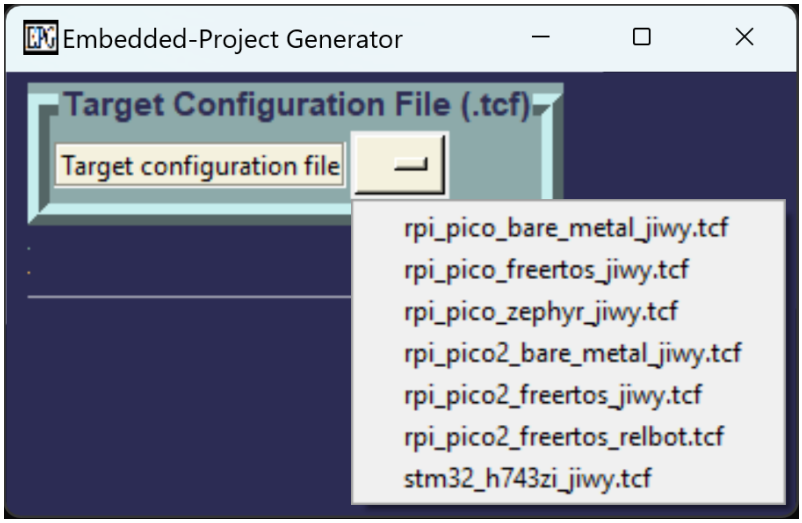
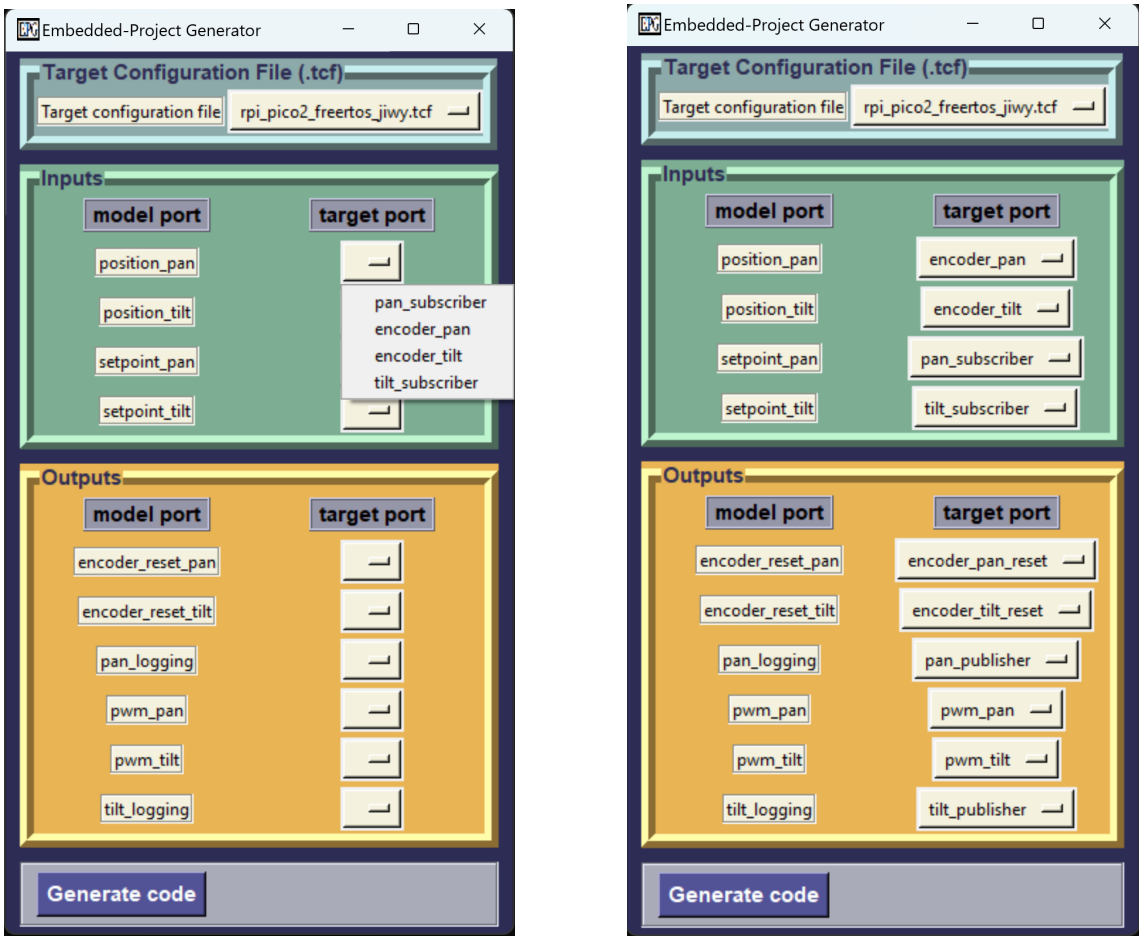


Figure 4.6: Target selection EPG GUI

2. **Connect ports GUI:**
   After target selection, the EPG GUI shows all model ports, obtained from the metadata of the model-based mechatronic control software, along with all target ports defined in the .tcf file for each target component, in order to connect the ports together. This is shown in Figure 4.7, where Figure 4.7a shows the empty GUI with a dropdown menu for selecting target ports, and Figure 4.7b shows the GUI with all ports connected.



(a) Select target port from dropdown menu

(b) All model ports connected with target ports

Figure 4.7: Connect model ports with target ports by selecting target ports from EPG GUI dropdown menu

3. **Code generation:** In the final EPG step, the port connection information, control code, and the target configuration are provided to the EPG code generator (discussed in Section 4.3.4). This step combines all the information to generate a complete project, which is ready for compilation.

4.3.3 EPG software package

Design motivation topic: Develop an EPG package structure.
Motivation: The EPG package structure is designed to closely follow the previously discussed EPG steps to ensure organised and maintainable code. To achieve this, the EPG steps are implemented in separate Python files. The main.py file coordinates the EPG steps by using connect_gui.py to handle the GUI for the select target and connect ports steps, and code_generation.py to manage the final code generation step. Communication between the separate Python files takes place using config.py, which functions as a centralised database to store and retrieve data.

All EPG targets, which are stored in the Target folder, follow the target description from Section 4.2.3. Each target includes a .tcf file, Python EPG-modules, and a pre-configured base project. Generated code is integrated into the base project, resulting in the EPG-generated project. To enable 20-sim to create code for use by the EPG, the EPG must be registered as a target in the 20-sim C-Code Generation Target List. This is done by providing a file called Targets.ini. It configures the 20-sim C-Code Generation process and specifies the main.py file, which is triggered after 20-sim generates the C-Code. It is important to note that the Targets.ini file is not related to the EPG targets themselves, but rather serves as a configuration file for integrating the EPG with 20-sim.

Realisation: The structure of the EPG Python package is shown in Figure 4.8. The following elements are part of the package:

- **main.py**: is the starting point of the Python package, which coordinates the EPG. The file contains 20-sim tokens, which are replaced by the 20-sim C-Code generation process with 20-sim metadata of the model name, all model inputs/outputs, and the destination folder for the generated code.
- **connect_gui.py**: The connect_gui.py is started by main.py to manage the EPG GUI as explained in Section 4.2.3 to connect the model with the target. It uses a Connection Python class, whose objects are used by connect_gui.py to create and connect the model/target ports in the GUI and store the user selections in config.py.
- **code_generation.py**: After connect_gui.py, main.py starts code_generation.py to combine the GUI user selections with the target information to create a generated project. A detailed explanation of this process can be found in Section 4.3.4.
- **config.py**: The config.py is used by the EPG as a centralised database. It is used by all other Python files to store and retrieve EPG GUI choices, model information, and the selected .tcf file.
- **Target**: All EPG targets are stored in the Target folder. The targets use the target concept as explained in Section 4.2.2. Each target has a .tcf file, Python EPG-modules for the functionality of the target components, and a base project which is a pre-configured target project to which the generated code is added.
- **Targets.ini**: A Targets.ini file is implemented in the EPG directory, enabling that the EPG can be added as a target in the 20-sim C-Code Generation Target List and is automatically started by the 20-sim C-Code Generation process when the EPG target is selected from the Target List. The Targets.ini file, specifies the 20-sim C-Code template files that are used to generate 20-sim C-Code files before the EPG is started. Additionally, the Targets.ini file specifies that the EPG main.py Python file is called using a postCommand after 20-sim generates the C-Code files, which starts the EPG. The generated 20-sim C-Code files serve as input to the EPG.

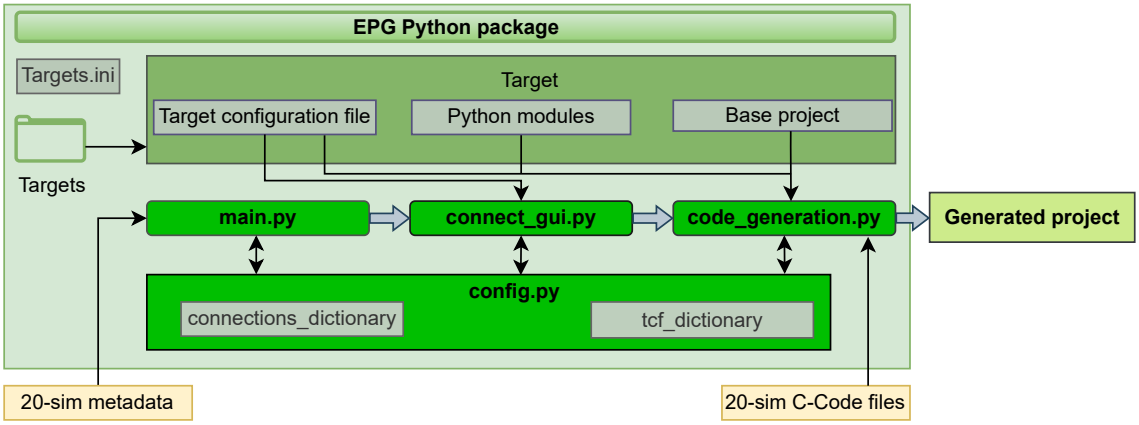


Figure 4.8: EPG Python package

4.3.4 Code generator

Design consideration topic: Determine how microcontroller code is generated based on the target configuration file.
Options:

- Create a separate C-Code template with tokens for each target component. The tokens are placeholders that are filled with component-specific information from the .tcf file during code generation, resulting in C-Code snippets being placed in the appropriate sections of the main.c file.
  Pros: Each component has its own template, making it easy to implement modifications and to add additional components. Templates with similar functionality can be re-used.
  Cons: Since all templates are separated it can be difficult to keep an overview of the resulting main.c file.
- Create one C-Code template with tokens for all target components that are filled with component specific information from the .tcf during code generation. After filling the tokens, the template functions as a main.c file.
  Pros: The entire template is in one file making it easier to keep an overview of the resulting main.c file.
  Cons: When the template implements a lot of target components it becomes difficult to maintain. Adding new target components to the template is difficult, since the right location for the target component has to be manually determined.

Decision: A separate C-Code template with tokens for each target component is used. This provides a flexible approach allowing for easy modifications and a separation of concerns between different target components.
Realisation: The code_generation.py script manages the code generator, which follows a series of steps with the goal of creating a main.c file with all the target functionality. The code generator follows the following steps:

- First, a list is created of all target components in the .tcf file of the chosen target. Each Python EPG module specified in a target component contains multiple functions with C-Code that together represent the functionality of the target component. The functions return a structured string with this code. The code resulting from all target components is combined in a main.c file.
- Code is generated taking into account the structure of a main.c file. The main.c file structure includes libraries and headers, global declarations, function prototypes, a main function for initialization, and looping. Therefore, a code generation order has been created. The code generation order defines function names that can be used by the Python EPG modules. The function name determines the location in the main.c file where the generated code will be placed.
  The code is generated based on the pseudocode shown in Algorithm 1. This process is illustrated in Figure 4.9 showing both the pseudocode and the information used in each step of the pseudocode. The illustration uses the same example situation as discussed in Section 4.2.2.
  The algorithm shows two nested loops. The outer loop iterates over the function names defined in code generation order. For each function name the inner loop iterates over all target components defined in the .tcf file and checks if the Python module belonging to the target component has the specific function. If the module has the function, it is executed and the returned string containing C-Code is placed at the corresponding location in the main.c file, otherwise the function is skipped.
- The base project is first copied to the target directory. A base project is a preconfigured project that serves as the foundation for the generated project. It contains a CMakeLists.txt file for compilation and when needed additional files required for the

execution of a specific target. Each target has a setup Python module to finalize the configuration of the generated project. This setup Python module adds the generated main.c file, along with all additional required files, to the base project, creating the final project.

Algorithm 1 Code generator

```
for function in code_generation_order do
    for target_component in target_configuration_file do
        if function in target_component.python_module then
            code_string ← run(function)
            main_c.write(code_string)
        end if
    end for
end for
```

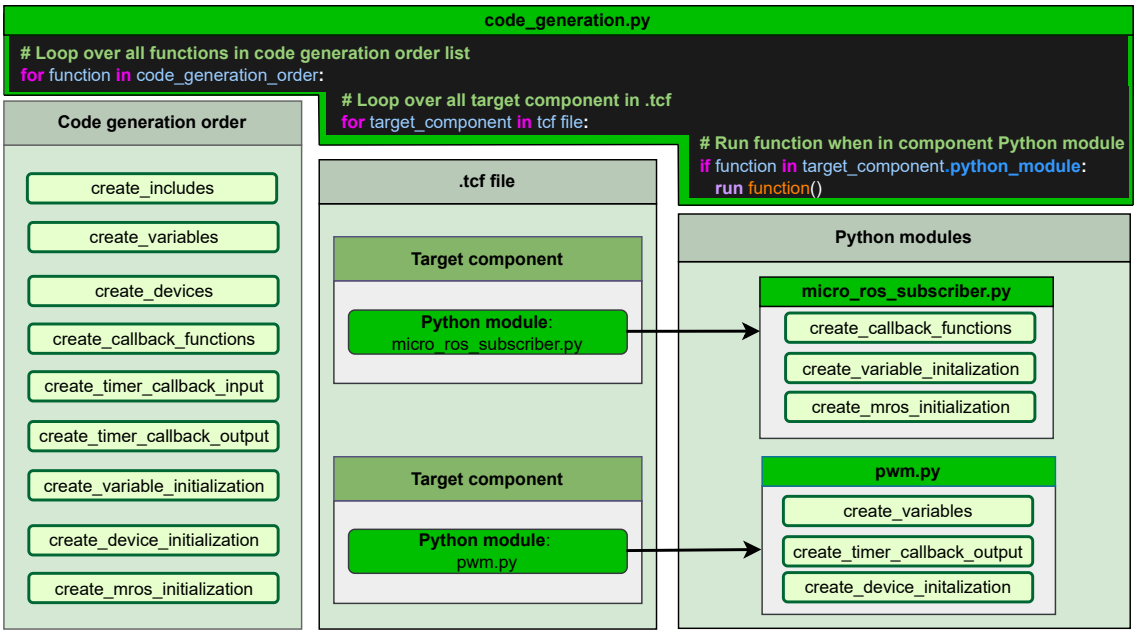


Figure 4.9: Code generation

4.4 EPG-target configurations

Target configurations have been created for the Raspberry Pi Pico, Raspberry Pi Pico 2, and STM32 Nucleo H743ZI as shown in Table 4.1. The Raspberry Pi Pico is compatible with bare-metal, FreeRTOS, and Zephyr. The Raspberry Pi Pico 2 is compatible with bare-metal and FreeRTOS. However, it is a new platform and has not yet been implemented in Zephyr, so it is currently not supported. The STM32 Nucleo H743ZI lacks a bare-metal implementation compatible with micro-ROS. While it should be FreeRTOS compatible, the existing implementation appears to be non-functional. The issue has been reported to micro-ROS, which is discussed in more detail in Appendix C. This Appendix discusses all active engagement with open-source developers that took place during this thesis. The Zephyr implementation is fully functional and consequently used.

Table 4.1: Target configurations

	Raspberry Pi Pico	Raspberry Pi Pico 2	STM32 Nucleo H743ZI
Bare-metal	Implemented & tested	Implemented & tested	not available
FreeRTOS	Implemented & tested	Implemented & tested	not working
Zephyr	Implemented & tested	not (yet) available	Implemented & tested

4.4.1 Bare-metal

The bare-metal implementation is used by the Raspberry Pi Pico and the Raspberry Pi Pico 2. A diagram of this mapping is shown in Figure 4.10.

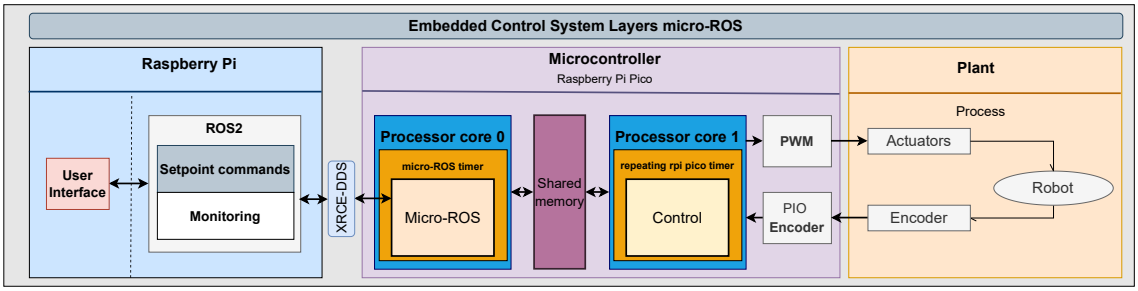


Figure 4.10: Bare-metal target mapping

Dual-core processing

Both microcontrollers have two cores that can be used to perform processing. The Raspberry Pi Pico Software Development Kit (SDK) has an option to run a task on a specific core.

Design consideration topic: Efficiently use both cores for processing.
Options:

- Create a producer-consumer separation, where one core produces data by retrieving sensor data and micro-ROS subscriber information and use another core for processing the data by running the control loop and driving outputs.
  Pros: There is a clear distinction between a core producing data and handling all system inputs and a core consuming the produced data resulting in system outputs.
  Cons: Retrieving and processing sensor input data happens on separate cores resulting in delays. This design does not consider the real-time constraints of different tasks.
- Separate the soft real-time and firm-real time tasks on two different cores. One core takes care of the soft real-time communication, and the other core handles the firm real-time control loop.
  Pros: Real-time constraints are considered. The soft real-time core manages all external communication mitigating the impact of communication delays. The firm real-time core only runs time critical code to guarantee the timing constraints are met.
  Cons: Because on one core only the firm real-time task is executed, this core may be underutilized.

Decision: The approach with soft real-time and firm-real time tasks running on two different cores is used. This approach is chosen since firm-real time performance is essential for the correct functioning of the target. Having a strict separation of input/output data is less of a concern.
Realisation: The core running the soft real-time communication is responsible for receiving information from ROS 2 subscribers and publishing information for monitoring using micro-ROS publishers. Publishing data happens using a micro-ROS timer that publishes data at a

fixed rate.
Micro-ROS is not thread-safe by default and can therefore only run on one core, otherwise memory may become corrupted. Since micro-ROS is needed for communication, it is used on the soft real-time core.
The firm real-time core uses the Pico SDK repeated timer, which is an interface to use the hardware interrupt timer on the Pico. The repeated timer is configured such that the specified timer interval determines the time between the starts of the timer callback function.

Inter-task communication

Design consideration topic: The target needs inter-task communication for communication between the soft real-time and firm real-time tasks running on two different microcontroller cores.
Options:

- Atomic operations guarantee that setting/getting data happens safely by taking care of this in low level instructions.
  Pros: Atomic operations are efficient for handling memory protection.
  Cons: The data communicated in shared memory uses double precision for setpoint and position data. Atomic operations are however not available on the Pico for double precision. Therefore, this option is not viable.
- The Pico has FIFOs that can be used for communication between cores.
  Pros: Simple ordered inter-task communication method.
  Cons: The FIFOs available on the Pico can only use single-precision numbers. Because of that, the double-precision data cannot directly be communicated using FIFOs. Therefore, this option is not viable.
- Semaphore mechanisms can be used to protect access to a single shared memory resource. Mutex is the most suitable semaphore due to its inherent ownership model, ensuring that only the task that locks the mutex can unlock it. This prevents accidental resource release by other tasks.
  Pros: Mutexes can handle the protection of double-precision data.
  Cons: Mutexes are less efficient to alternatives such as atomic operations and FIFOs.

Decision: Considering all options, only using shared memory protected by mutexes seems to be a viable option for inter-task communication. It might be possible to use atomic operations or FIFOs, but this would introduce additional complexity and overhead in code by having to perform transformations for the double-precision data to realise compatibility. Using mutexes does not require any custom implementations, which is preferred.

Design motivation topic: Synchronise access to shared memory between tasks with different timing constraints.
Motivation: The shared memory is implemented as a single-slot buffer to ensure that only the most recent data is shared between both tasks. When new data becomes available the buffer is overwritten. The buffer is protected by mutexes.
The firm real-time loop is completely non-blocking to ensure firm real-time loop performance. It attempts to lock the buffer once per loop iteration, and if unsuccessful retries the next loop iteration.
The soft real-time task accesses the data at a lower rate compared to the firm real-time task. Because of the timing differences, it is likely that the data is already blocked when the soft real-time task tries to access it. To handle this, the mutex tries to access the memory for a maximum time of 10 ms, before continuing other operations.
Since the firm real-time control loop, locking and releasing the same data, runs with 1 ms intervals, the release of memory is guaranteed to happen within that time frame. The 10 ms timeout serves as a safety margin for unexpected delays.

Hardware I/O

The Raspberry Pi Pico microcontrollers have hardware timers that can be used for generating PWM signals to control the speed of a motor via a H-bridge. The advantage of using hardware timers is that they operate independently from the processor, allowing the signal to switch at the right moment without consuming processing time.
The microcontrollers do however not have hardware timers that can be used to keep the count of quadrature encoders. Quadrature encoders are used to determine the position of robot components.
Design consideration topic: Implement quadrature encoder counting.
Options:

- The quadrature encoder count can be tracked in software, by using hardware interrupts that adjust a counter variable upon changing signals.
  Pros: Simple approach for keeping track of quadrature encoder signals.
  Cons: Using hardware interrupts would lead to disturbances on the processor. Especially for the firm real-time task it is desired to run with minimal disturbances.
- The Raspberry Pi Pico has PIO (programmable Input/Output) units. The PIO units can be programmed with state machines that function as small processors with a limited instruction set. The state machines run in parallel with the main processor, executing instructions at the same speed, without disturbing its operations.
  Pros: The PIO units can be programmed to read quadrature encoders and update the encoder count in a register without disturbing the processor. When needed the current encoder value can be read from that register. The PIO's run at the same clock speed as the main processor ensuring high performance tracking of high frequency quadrature encoder signals.
  Cons: Requires using the PIO assembly instruction set.

Decision: The PIO units are used to track quadrature encoder signals. This allows to offload this task from the processor, enabling parallel processing. The control loop can directly read the current encoder count from the PIO register.

Realisation: The control loop directly interacts with the hardware I/O to steer the system to certain positions. By using hardware solutions for all hardware I/O, the control loop can run without any disturbances caused by managing hardware I/O. The control loop is only responsible for setting PWM values and reading encoder counts from PIO registers, while the hardware handles all functionality.

Deployment and debugging

Design motivation topic: Enable easy deployment and debugging for EPG-generated projects.
Motivation: EPG-generated projects should be easy to deploy. Therefore, each base project, the pre-configured project that serves as the foundation for the generated project, for the Pico and Pico 2 has a CMakelists.txt file that fully configures the project. The CMakelists.txt file fetches all external dependencies, such as micro-ROS libraries, and contains compilation instructions for the project. The CMakelists.txt is configured such that EPG-generated projects are directly compatible with the Raspberry Pi Pico Visual Studio Code extension with the functionality to compile, deploy, and debug EPG-generated projects.

4.4.2 FreeRTOS

A FreeRTOS implementation has been created for the Raspberry Pi Pico and the Raspberry Pi Pico 2 using FreeRTOS-Kernel V11.1.0. The implementation is similar to the bare-metal implementation in terms of inter-task communication, hardware I/O, and deployment and debugging. The dual-core processing is however different. A diagram of this mapping is shown in Figure 4.11.

Dual-core processing

With FreeRTOS it is also possible to use both cores of the microcontroller. The difference is however that the FreeRTOS scheduler determines on which core a task is going to run. This is implemented using the Symmetric Multiprocessing (SMP) implementation in FreeRTOS that schedules tasks over multiple identical processor cores that share the same memory.
The setup is similar as in the bare-metal case where one task is used for soft real-time micro-ROS communication, and another task is used for firm real-time control loop.
In contrast to the bare-metal case it is possible to create more than 2 tasks. FreeRTOS uses preemption, where tasks with a higher priority can interrupt lower priority tasks and use Round-robin for equal priority tasks. Since only two tasks are created and two cores are available for processing this will not be a problem.
The firm real-time control loop is managed by a FreeRTOS timer. The Timer Task is set to run with maximum priority, ensuring it always gets precedence over other tasks.

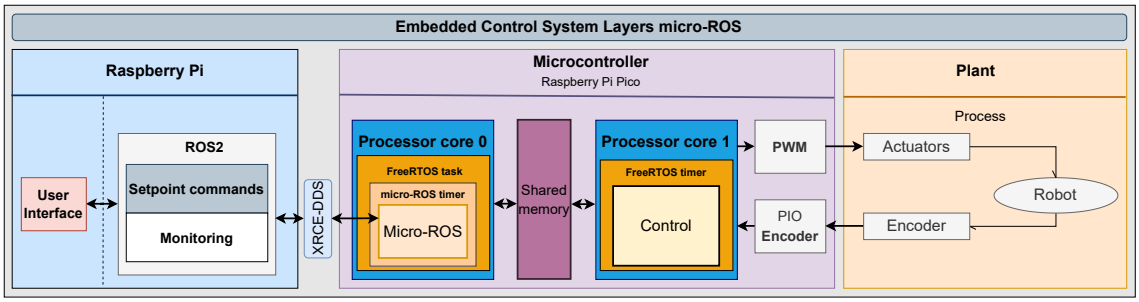


Figure 4.11: FreeRTOS target mapping

4.4.3 Zephyr

The Zephyr implementation is used by the Raspberry Pi Pico and the Nucleo H743ZI. A diagram of this mapping is shown in Figure 4.12.

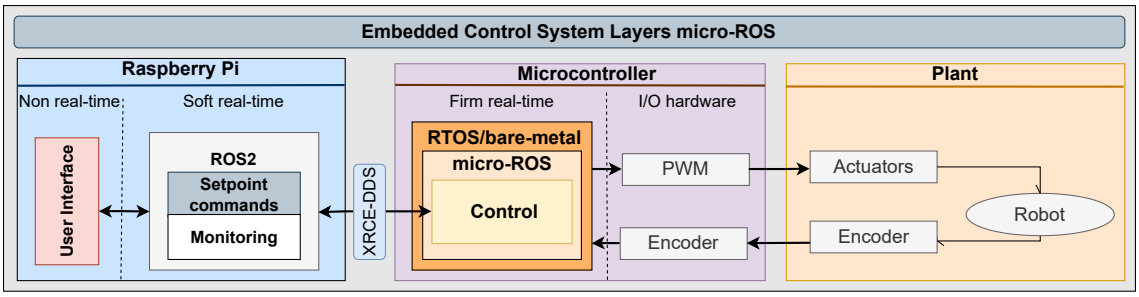


Figure 4.12: Embedded Control System Layers micro-ROS (repetition Figure 3.2)

Single-core processing

Unlike the bare-metal and FreeRTOS implementations Zephyr does not support SMP for the targets. Therefore, the Pico and Nucleo use a single core to perform both the soft real-time micro-ROS communication and the firm real-time control loop, leaving the other core idle.
Design consideration topic: Process all tasks using the single core.
Options:

- In the Zephyr implementation all execution can take place in one task. Because of that, the micro-ROS timer can be used to perform the firm real-time control loop management. Another micro-ROS timer can be used for publishing monitoring data.
  Pros: All processing can take place in one Zephyr task.

Cons: Using micro-ROS timers can have higher latency compared to alternatives such as a Zephyr timer.

- Use two Zephyr tasks, where the firm real time task makes use of a Zephyr timer.
  Pros: More control of the timing and priorities of the tasks.
  Cons: Potentially increases overhead caused by context switches between the tasks. Using multiple tasks would introduce additional complexity, as both tasks need sufficient execution time on a single-core system without starving the other task.

Decision: The micro-ROS timers are used, because all processing takes place in a single task limiting complexity and task switching overhead.

Hardware I/O

In Zephyr, hardware I/O functionality is defined in device drivers, and has to be registered in the Zephyr device tree. This makes it possible to use device drivers modular and add devices for the functionality of the target.
Both the Raspberry Pi Pico and the Nucleo H743ZI have hardware timers to create PWM signals and have Zephyr device drivers supporting this functionality. This is however not the case for reading quadrature encoder signals. As previously mentioned, the Raspberry Pi Pico does not have hardware timers that can read quadrature encoders and needs a PIO implementation for this functionality. The Nucleo H743ZI does have hardware timers capable of reading quadrature encoder signals and a Zephyr device driver for this purpose, but there is an error in the driver code making it non-functional.
Design consideration topic: Implementing quadrature encoder functionality in Zephyr.
Options:

- Develop a Zephyr quadrature encoder device driver supporting both the Raspberry Pi Pico and the Nucleo H743ZI. On the Raspberry Pi Pico, the device driver employs PIO logic to read quadrature encoders, while on the Nucleo H743ZI, it includes functionality needed to configure the hardware timers to read quadrature encoder signals.
  Pros: Zephyr typically uses drivers for adding functionalities to a device. All other functionalities are registered in device drivers. Creating a device driver for the quadrature encoder makes it possible to keep this consistency and integrate it with Zephyr interfaces. Using device drivers enables easy reusability of code.
  Cons: Creating a device driver is time consuming and requires expertise in how Zephyr device drivers work.
- Directly implement quadrature encoder code in the main code.
  Pros: Simple solution that is easy to deploy.
  Cons: A direct implementation would not be consistent with the way other device functionalities are handled in Zephyr.

Decision: It was chosen to create a Zephyr device driver for implementing the functionality of the quadrature encoder. Although this takes more development effort the result is a solution that fits well within the Zephyr ecosystem.

Deployment and debugging

Design motivation topic: Enable easy deployment and debugging for EPG-generated projects.
Motivation: EPG-generated projects should be easy to deploy. For Zephyr the base project is configured according to the Zephyr project layout. A west.yml is configured to directly import EPG-generated projects in the Zephyr development environment that can be used to compile, deploy, and debug EPG-generated projects using Zephyr.

4.5 Verification of the requirements

This section verifies the functional and non-functional requirements outlined in Chapter 3. These requirements are summarised as:

- **Functional Requirements:** Use ROS 2, use a Raspberry Pi 4B, direct implementation of 20-sim model-generated code on mechatronic systems, firm real-time capabilities, compatibility with different mechatronic systems, networking support robotic components, double-precision floating-point calculations.
- **Non-Functional Requirements:** Minimal user input, good maintainability, support for software components, and use of common programming languages.

The verification of functional requirements is discussed in Section 4.5.1 and the verification of non-functional requirements is discussed in Section 4.5.2.

4.5.1 Functional requirements

1. **The EPG-generated project *must* use the ROS 2 ecosystem.** The EPG implements microcontroller targets with micro-ROS Jazzy support for communication with ROS 2 Jazzy on Ubuntu server 24.04 running on both the Raspberry Pi 4B and Raspberry Pi 5. This implementation is verified by the ping-pong performance test in Section 5.2.2.
2. **The EPG-generated project *must* support a robotic setup that uses a Raspberry Pi single-board computer.**
   Verified. See item 1 - The EPG-generated project must use the ROS 2 ecosystem.
3. **20-sim model-generated code of embedded control software *must* be directly implementable on mechatronic systems.**
   A 20-sim model of JIWY is created and simulated for the JIWY performance test. The JIWY_Control implementation of the 20-sim model is deployed on microcontroller targets using the EPG. The motion tracking test in Section 5.2.3 demonstrates consistency between the simulation and real-world results, verifying the direct implementation of 20-sim generated code in a real-world application.
4. **The EPG-generated project *must* have real-time capabilities.**
   EPG-target must fulfil the requirement for soft real-time micro-ROS communication with a 33 ms time interval and firm real-time control loop execution with a 1 ms time interval. Both are verified in tests using the JIWY robot in Section 5.2.3.
5. **There *should* be compatibility with different mechatronic systems.**
   The design and implementation of the EPG ensures compatibility with different mechatronic systems. The implemented EPG-target components enable flexible configuration of PWM components to control motor drivers and quadrature encoders to read encoder signals. Tests have been conducted with both the JIWY robot (Section 5.2; Section 5.3) and the RELbot robot (Section 5.4) present in the RaM laboratory verifying compatibility with different mechatronic systems.
6. **Support for networking between robotic components *should* be implemented.**
   The networking capabilities of the microcontroller implementations are tested, demonstrating reliable communication using micro-ROS. The microcontroller is implemented as a node in ROS 2, verifying the ability of the EPG-target to participate in a robot network. The tests confirmed effective command execution and monitoring in a multi-node environment. Because the networking capabilities of ROS 2 make it possible to flexibly add multiple microcontrollers and single-board computers and the EPG-target is integrated and tested in ROS 2, support for networking between robotic components is implemented. This is verified by the networking tests using four robots in Section 5.3.
7. **There *should* be support for double-precision floating-point calculations.**
   The JIWY tests in Section 5.2.3 uses JIWY_Control software on all targets. This software implements 20-sim model-generated C-Code that performs double-precision floating-

point calculations. The correct functioning of this software is shown in the control loop and motion profile tracking tests, verifying that double-precision floating-point calculations are supported.

4.5.2 Non-functional requirements

1. **Minimal user input *should* be required.**
   The EPG automatically collects 20-sim metadata and displays it in the GUI. The user selects the EPG-target and connects model ports with target ports, providing the necessary data for project generation. The GUI is tested by generating the EPG-generated project used for the JIWY tests in Section 5.2.3, which supports the verification of this requirement.
2. **The EPG *should* have good maintainability.**
   According to the IEEE Software Engineering Glossary "Software maintainability is defined as the ease with which a software system or a component can be modified, to correct faults, improve performance or other attributes, or adapt to a changed environment." (Chen et al., 2017).
   To optimise maintainability, the following strategies are used in the implementation of the EPG:
   - Using a component-based implementation ensures easier modification of software components and modularity for extending the tool.
   - Using open-source software when possible, reducing individual maintenance effort by benefiting from community support.
   - Minimizing custom software implementations, which increase the complexity.

   The Maintainability Index (MI) quantifies code maintainability based on complexity, volume, and size, with higher scores indicating easier maintenance. Radon, a Python tool for computing code metrics, is used to assess the MI score of the EPG Python package (Radon Developers, 2025). The result shows that the EPG package is "very high maintainable" with an average MI score of 94.
3. **There *should* be support for the use of software components.**
   Support for software components is implemented in the EPG. The EPG-target uses EPG-target components to configure the target configuration file. The use of the EPG-target components is verified in the JIWY tests discussed in Section 5.2.3.
   Furthermore, the EPG-generated project deployed on a microcontroller is implemented such that it can be used as a ROS 2 node component within a ROS 2 system, which is also verified in Section 5.2.3.
4. **The EPG and EPG-generated project *could* use common programming languages.**
   The EPG implementation uses Python, while the EPG-generated projects use C, both of which are common programming languages. The functioning of the EPG and EPG-generated projects is verified in Section 5.2.3.

5 Testing

5.1 Introduction

This chapter evaluates the performance and network stability of micro-ROS communication and distributed control using tests with varying EPG-targets and multiple robots connected in a network, to demonstrate the effectiveness of EPG-generated projects in real-world applications. While the EPG itself is not explicitly tested, its extensive use in generating EPG-projects for numerous tests inherently serves as a validation of its functionality.
The chapter begins with performance testing in Section 5.2, which addresses the third thesis goal: Test the performance of code, generated by the software tool, with a robot. The tests evaluate micro-ROS communication and real-time execution of control loops. Two test setups are used to explore different hardware and RTOS configurations and their impact on system performance. One is a ping-pong round-trip measurement to assess micro-ROS communication without interference from other tasks, and the other is a robot test to analyse real-time control and communication in a real-world application, as well as the ability to match the behaviour of a simulated model.
Network testing in Section 5.3 addresses the fourth thesis goal: Test the performance and stability of a distributed network with multiple robots using code generated by the software tool. The tests investigate the integration of multiple EPG-generated projects as ROS 2 nodes in a distributed network. Round-trip time measurements assess communication performance, while motion profile tracking tests are used to evaluate the scalability and stability of distributed control. Additionally, the tests explore the feasibility of teleoperation with mirrored robots.
Test compatibility with a different mechatronic system in Section 5.4, discusses a test that further evaluates the applicability of the EPG on a robot different from the one used in the previous tests. This test has three objectives: First, it aims to demonstrate that the EPG can be applied to a mechatronic system other than the one used in the other tests. Second, it shows the feasibility of rapid prototyping by using the EPG to generate an EPG project for another robot within one day. Third, by accomplishing this, the test implicitly demonstrates the ease of use of the EPG.
Together, these sections provide insight into the applicability of EPG-generated projects in networked real-time robot systems. The objective of these tests is to demonstrate that the generated projects meet the requirements and can be effectively used for robot setups, rather than pushing the limits of the test setup.

5.2 Performance testing

5.2.1 Introduction

In this section tests are discussed that provide insight into the performance of EPG-generated projects deployed on multiple EPG-targets. Performance implies in this context the responsiveness of micro-ROS communication, the real-time execution of a control loop running concurrently on the same system as micro-ROS, and the ability of a robot, controlled by an EPG-target, to match the behaviour of a simulated model.
Since it is unclear in advance which targets meet the requirement of a 1 ms firm real-time control loop and 33 ms soft real-time communication, a wide variety of performance tests are conducted to gain insight into the strengths and weaknesses of the different targets.
The first test setup is a ping-pong test used to obtain micro-ROS communication performance metrics with no other tasks running on the system. In Section 2.3.2 the data transmission between the micro-ROS agent and micro-ROS client is estimated to be 23 μs. However, this estimate does not account for overheads, such as serialisation and deserialisation. To obtain a realistic estimate, measurements are necessary to verify the actual communication performance of EPG-targets.

The second test setup uses a robot for both micro-ROS communication and control loop performance measurements. This provides insight into a realistic case and the applicability of micro-ROS in real-time robot systems. Since the performance is also dependent on the used hardware and RTOS, different hardware and software combinations are used for the tests. The section concludes with a robot motion profile tracking test, where an EPG-generated project is deployed on an EPG-target to follow a motion profile. This test aims to compare simulated performance with real-world results, which evaluates the functionality of the EPG-generated project.

5.2.2 Ping pong

Setup

The ping-pong measurement setup is a round-trip message passing application that is used to determine the performance of the micro-ROS communication and to determine which EPG-targets can perform soft real-time communication within the required 33 ms time interval with no other tasks running on the system.
The performance variables of interest are:

- **Round-trip time (RTT):** The time a package takes to be transmitted round-trip through the setup.
- **Latency distribution:** The distribution of latency across different stages of the round-trip.
- **Jitter:** The variability in round-trip times.

A diagram of the setup is shown in Figure 5.1. The figure shows a Raspberry Pi single-board computer running ROS 2 Jazzy on Ubuntu server 24.04 LTS and a microcontroller running micro-ROS Jazzy, which communicate via a serial USB 2.0 connection using micro-ROS XRCE-DDS. The Raspberry Pi and the microcontroller have separate clocks and the microcontroller has a limited amount of memory to log data. Therefore, a logic analyser is used that logs data from the GPIO pins of both devices, enabling synchronised measurements. Each measurement consists of $500{\cdot}10^6$ samples, taken at a rate of 1 MHz, which corresponds to a measurement duration of 8.33 min. The use of $500 \cdot 10^6$ samples enables the observation of long-term behaviour in the logged data. Sampling at a frequency of 1 MHz provides sufficient accuracy to reliably detect all signal changes. The sequence of actions and measurements is shown in Table 5.1.

Table 5.1: Ping pong measurement steps

GPIO #	Platform	Action	GPIO toggle
1	Raspberry Pi	Start of ping-pong measurement	Before publish function is called.
2	Raspberry Pi	ROS2 publisher sends message on /topic ping	Directly after publish function completes.
3	Microcontroller	micro-ROS subscriber callback receives message on /topic ping	Directly at the beginning of the callback function.
4	Microcontroller	micro-ROS subscriber callback publishes message on /topic pong	Directly after the publish function is finished.
5	Raspberry Pi	ROS2 subscriber callback receives message on /topic pong	Directly at the beginning of the callback function.

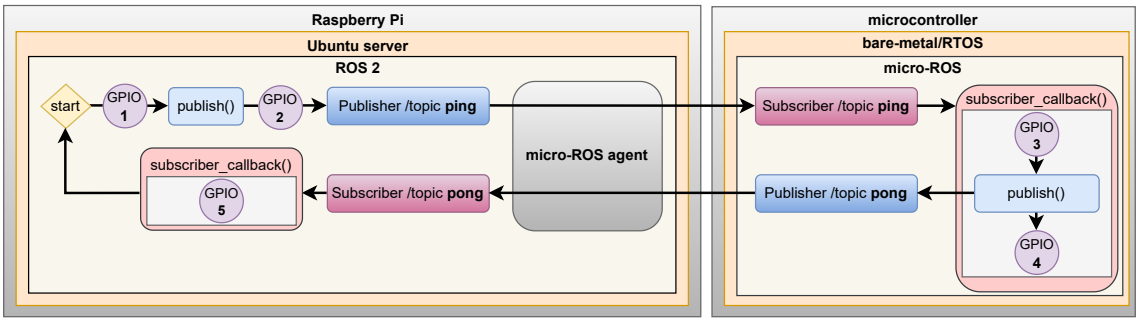


Figure 5.1: Ping pong setup

The transmitted message is a double-precision floating-point number, which is increased with each iteration. Doubles are used because robot controllers often require double-precision set-points, and monitoring data is usually represented in double precision as well.
The callback function at the Raspberry Pi checks if the received message content is equal to the previously sent message content. If not, the logic analyser measurements at the different GPIO toggle points will be out of sequence corrupting the measurements. Only when the check is successful a new message is published starting a new round-trip.
Micro-ROS offers best effort and default quality of service settings for the subscribers and publishers. The quality of service (QoS) settings give a guarantee for the reliability of the communication. The default setting has reliable delivery and provides checking mechanisms to guarantee that messages are delivered. The best effort setting does not have a checking mechanism, and therefore if something goes wrong during transmission the message is lost. The configurations shown in Table 5.2 are used during testing. Four QoS configurations are tested to determine the impact on performance.

Table 5.2: Quality of service configurations

QoS config	Subscriber	Publisher
1	best effort	best effort
2	best effort	default (reliable delivery)
3	default (reliable delivery)	best effort
4	default (reliable delivery)	default (reliable delivery)

The ping-pong test is conducted for each target in Table 4.1. The micro-ROS agent and ROS 2 run on a Raspberry Pi 4B or Raspberry Pi 5 single-board computer. The Raspberry Pi 5 is more powerful than the Raspberry Pi 4B. Since the micro-ROS agent and ROS 2 also affect the round-trip time, both single-board computers are used to investigate their impact.

Result round-trip time

Figure 5.2 shows the mean round-trip times, along with the standard deviation showing the variability. Each target is represented by four bars, corresponding to the QoS configurations listed in the same order as in Table 5.2. The bars are coloured following a heatmap colour scheme, with dark blue representing tests with a low RTT and dark red indicating those with a high RTT.

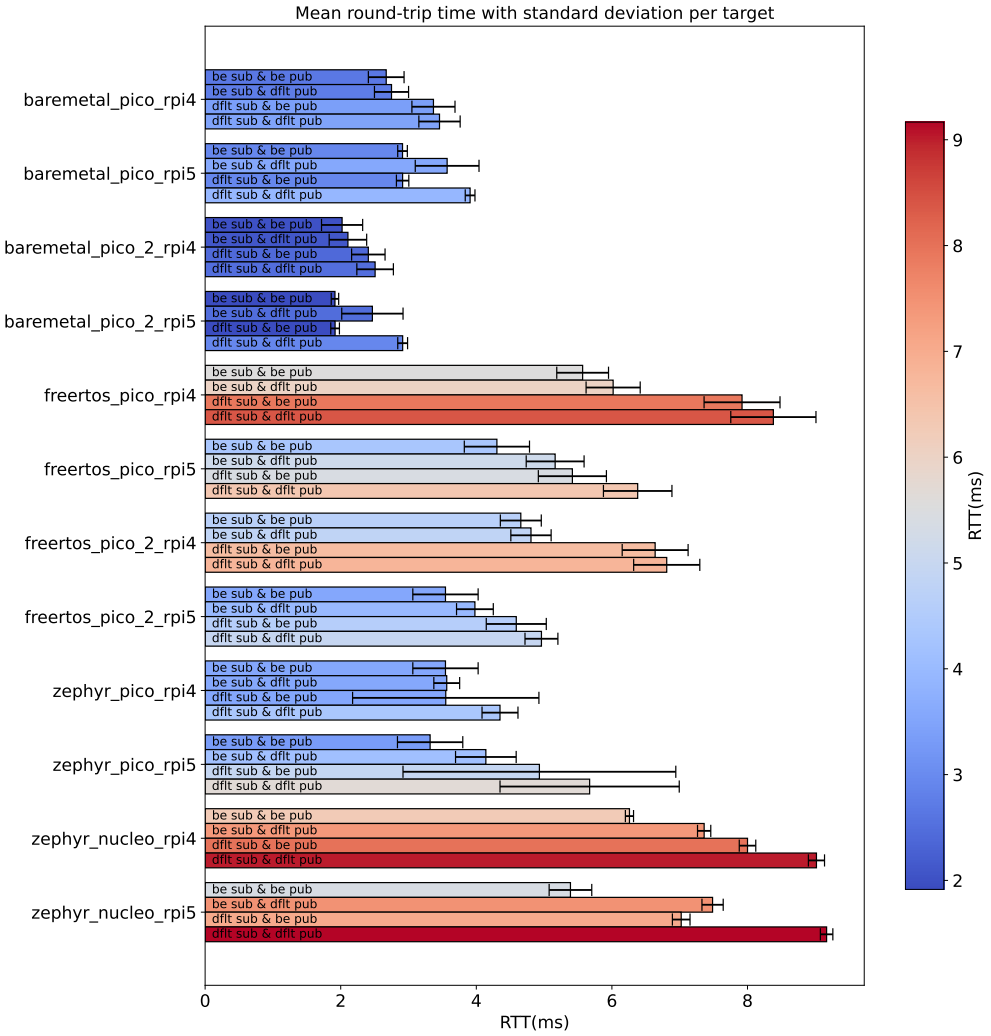


Figure 5.2: Mean round-trip time with standard deviation per target

Discussion round-trip time

Figure 5.2 shows that the bare-metal implementations overall have the lowest RTT. This is likely because these implementations have the least amount of overhead. The FreeRTOS and Zephyr implementations depend on RTOS task management using a scheduler to allocate time to the micro-ROS task. Although no other tasks were running during the test, meaning the scheduler should give all available time to the micro-ROS task, the overhead of the RTOS is still noticeable.

When looking at the QoS configurations, Figure 5.2 shows that for all cases best effort for both subscribing and publishing results in the lowest RTTs. This is expected, since this configuration has the lowest amount of overhead for verifying successful message delivery. When considering the other QoS results, it often seems that especially using the default subscriber setting causes a significant increase in RTT. This increase can probably be explained by the message acknowledgement process of the micro-ROS subscriber. Using default QoS for the micro-ROS publisher also results in a higher RTT.

Considering the different EPG-targets, the Raspberry Pi Pico 2 has the lowest RTTs, followed by the Raspberry Pi Pico. This is surprising, since the specifications indicate that the Nucleo H743ZI has a more powerful single-core processor and only a single core is used for communication.

The best performing combination is bare-metal Raspberry Pi Pico 2 with the Raspberry Pi 5. To

demonstrate the extent to which the mean roundtrip time has been improved by new hardware, the comparison begins with the Raspberry Pi Pico and Raspberry Pi 4. Replacing the Pico with the Pico 2 results in a gain of 24%. Subsequently, replacing the Raspberry Pi 4 with the Raspberry Pi 5 provides an improvement of 5%. Overall, the mean roundtrip time has decreased from 2.67 ms to 1.91 ms. All round-trip times stay within 10 ms. Therefore, from this test it follows that all EPG-targets can perform soft real-time communication within the required 33 ms time interval with no other tasks running on the system.

Result latency distribution and jitter

The Raspberry Pi Pico 2 bare-metal implementation using a best-effort publisher and subscriber and a Raspberry Pi 5 showed the best RTT results. Therefore, the RTT for this configuration is investigated in more detail with respect to the latency distribution and jitter. Detailed results of the other situations are also available in Appendix E.
Figure 5.3 shows a histogram of the RTTs, from which the jitter can be observed. The 100 worst RTTs are shown in Figure 5.4 along with the latency distribution showing what stages of the round-trip transmission take most of the time.

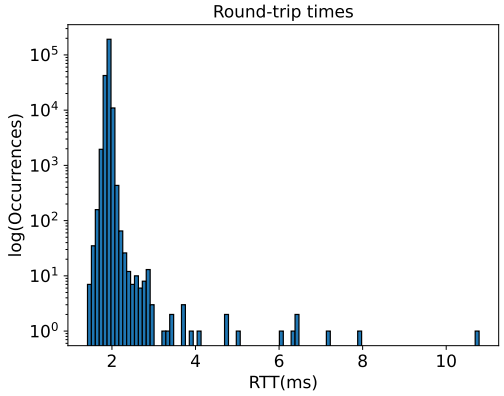


Figure 5.3: RTT jitter

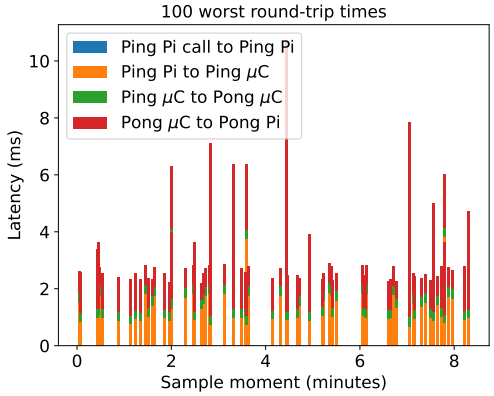


Figure 5.4: 100 worst RTTs

Figure 5.5 shows latency histograms for each of the different stages of a round-trip. The mean round-trip latency distribution is shown in Figure 5.6.

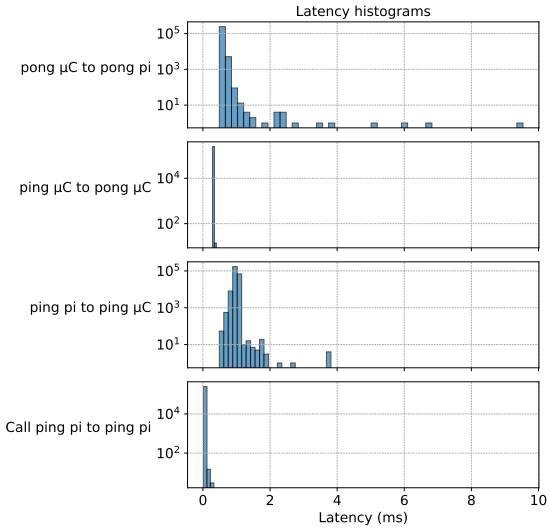


Figure 5.5: Latency histograms

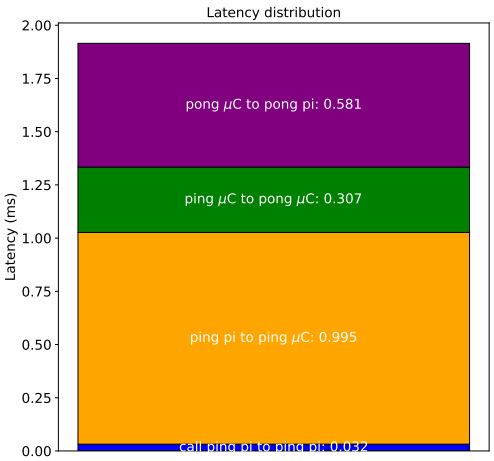


Figure 5.6: Mean latency distribution

Discussion latency distribution and jitter

The measurements were conducted to gain insight into the communication process. The histogram in Figure 5.3 shows that the majority of all RTTs are about 2 ms. There are some incidental outliers of which the 100 worst RTTs are highlighted in Figure 5.4. The outliers do not happen at a particular moment, but seem to be randomly distributed over the entire measurement. The largest outliers are in most cases caused by the communication from the Pico 2 to the Raspberry Pi 5. In Figure 5.5 the histograms of the different communication stages are shown. The histograms show that most of the outliers are from ping pi to ping μC and pong μC to pong pi. The mean latency distribution in Figure 5.6 shows that the majority of the round-trip time is spent in the communication from the Raspberry Pi 5 (GPIO 2) to the Pico 2 (GPIO 3). This can be explained, because this latency contains both serialization of the message by the micro-ROS agent as deserialization on the Pico 2. The latency from ping Pico 2 (GPIO 3) to pong Pico 2 (GPIO 4) only contains a serialization step and the transmission from Pico 2 (GPIO 4) to Pi 5 (GPIO 5) only contains a deserialisation step by the micro-ROS agent. Therefore to compare transmission to and from the Pico 2 it is more accurate to compare the sum of the times from Pi 5 GPIO 1 to Pico 2 GPIO 3 resulting in 1.027 ms and from Pico 2 GPIO 3 to Pi 5 GPIO 5 resulting in 0.888 ms, which has a difference of about 14%. Therefore, the latency distribution shows that the transmission from the Raspberry Pi 5 to the Raspberry Pi Pico 2 takes about the same time as the reverse path. Compared to the previously mentioned estimated theoretical transmission time of 23 μs between the micro-ROS agent and the micro-ROS client, these times are much higher. As already mentioned, the theoretical estimation did not account for overhead, such as serialisation, deserialisation, and scheduling overhead, but only gives an estimate of the time spend on the physical transmission.
The results of the measurements provide no indication that micro-ROS usage could lead to system instability.

5.2.3 JIWY robot

Setup

JIWY, a robot used in the RaM laboratory for educational purposes, is used for performance measurements of micro-ROS communication, the control-loop, and motion profile tracking. JIWY is a 2-DoF pan/tilt robot with a 30 fps webcam, as shown in Figure 5.8. JIWY contains typical elements of a robot system, such as PWM-driven motors and quadrature encoders, making it well-suitable for demonstrating an EPG-generated project.
The performance variables of interest are:

- **Control-loop cycle times:** The variation of measured control-loop cycle times from the set cycle time of 1 ms.
- **Control-loop computation time:** The time the control loop spends on calculations and setting/getting signals.
- **Round-trip communication:** The communication time a ROS package takes to be transmitted round-trip through the setup.
- **Latency distribution:** Latency at different stages of the communication process.
- **Motion profile tracking:** Position monitoring of tracking motion profile setpoints.

A 20-sim model of JIWY is created and simulated. The model overview is shown in Figure 5.7. The model contains motion profiles providing setpoints, an embedded controller steering the system, I/O hardware, and a robot plant.

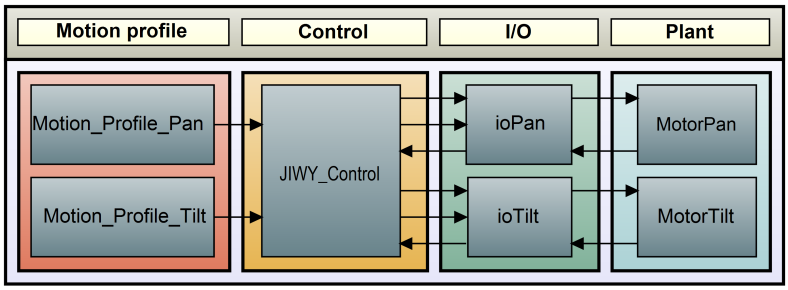


Figure 5.7: JIWY 20-sim model

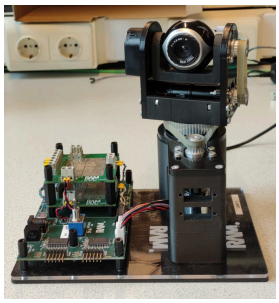
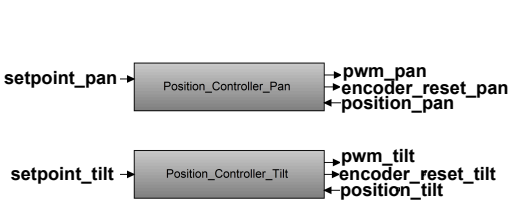
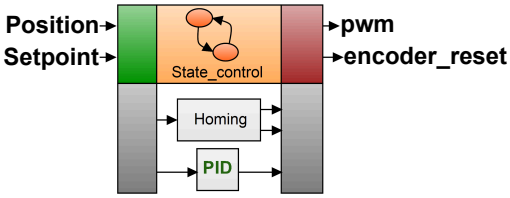


Figure 5.8: JIWY plant

The 20-sim JIWY_Control block is the embedded controller used for code generation. The implementation is shown in Figure 5.9. JIWY_Control includes two position controllers for the pan and tilt, as shown in Figure 5.9a. The details of these position controllers are shown in Figure 5.9b. The position controller uses a modified implementation of the continuous simple state machine provided by 20-sim to create a state controller that can switch between a homing sequence and PID control. Before JIWY can be operated, a homing sequence must be performed, during which JIWY is directed to a known reference position. Once initialized, PID controllers control the pan and tilt movements.



(a) JIWY position controllers



(b) Position controller implementation

Figure 5.9: JIWY_Control implementation

For this test the JIWY_Control implementation is deployed on all configurations shown in Table 5.2. Each measurement consists of $500 \cdot 10^6$ samples, taken at a rate of 1 MHz.
An example of such an implementation applied to JIWY is shown in Figure 5.10. This example uses a target containing the Raspberry Pi Pico 2, running FreeRTOS. The entire setup is configured in a .tcf file.
A Raspberry Pi 5 single-board computer is used to run ROS 2 Jazzy with a micro-ROS agent.
The Raspberry Pi Pico 2 distributes the workload of soft real-time communication and firm real-time control of the system using FreeRTOS SMP, with both tasks being allocated across the two cores to optimise performance.
On the Raspberry Pi 5 the setpoints are published on the ROS 2 topics /pan_subscriber and /tilt_subscriber, which are received by the microcontroller task running micro-ROS. Communication takes place via a serial USB 2.0 connection using micro-ROS XRCE-DDS.
On the Raspberry Pi Pico 2 a FreeRTOS task is dedicated to handling all micro-ROS communication, receiving setpoints and publishing monitoring data at a fixed rate. The FreeRTOS timer task ensures firm real-time execution of JIWY_Control at 1 ms, which is a typical rate used for robots because it balances responsiveness and computational efficiency. Shared memory is used for communication between the tasks. Using shared memory, the setpoints received by the communication task are provided to the FreeRTOS timer task running the JIWY_control control loop. After the controller calculations are completed the setpoint data is set as monitoring data by the control loop. Again, using shared memory, the monitoring data is provided to the task running micro-ROS. The microcontroller uses topics /pan_publisher

and /tilt_publisher to publish the monitoring data. The data is subsequently received on those topics by the Raspberry Pi 5 to monitor the current position completing a round-trip of the data. The PWM channels on the Pico 2 steer the motors and a PIO encoder implementation reads the encoder values. The sequence of actions and measurements is shown in Table 5.3.

Table 5.3: Ping pong measurements

GPIO #	Platform	Action	GPIO toggle
1 → 2	From Raspberry Pi 5 to Raspberry Pi Pico 2	Publish message on topic /pan_subscriber	Before publishing message (GPIO 1) and in callback function (GPIO 2).
3 → 4	From Raspberry Pi Pico 2 to Raspberry Pi 5	Publish message on topic /pan_publisher	Before publishing message (GPIO 3) and in callback function (GPIO 4).
5 → 6	From Raspberry Pi 5 to Raspberry Pi Pico 2	Publish message on topic /tilt_subscriber	Before publishing message (GPIO 5) and in callback function (GPIO 6).
7 → 8	From Raspberry Pi Pico 2 to Raspberry Pi 5	Publish message on topic /tilt_publisher	Before publishing message (GPIO 7) and in callback (GPIO 8).
A → A	Raspberry Pi Pico 2	Control loop cycles are measured by comparing the time between consecutive toggles of GPIO A	At the beginning of the control loop.
A → B	Raspberry Pi Pico 2	Get input signals (Encoders and setpoints)	Before controller calculations (GPIO B) and directly after controller calculations (GPIO C).
B → C	Raspberry Pi Pico 2	Controller calculations	Directly at the beginning of the callback function.
C → D	Raspberry Pi Pico 2	Set output signals (PWM and monitoring)	Before setting output signals (GPIO C) and directly after setting output signals (GPIO D).
1 → 4	Measurements on Raspberry Pi 5. Message crosses GPIOS: 1 → 2 → A → B → C → D → 3 → 4	Roundtrip message pan	Before publishing message on topic /pan_subscriber (GPIO 1) and in callback of topic /pan_publisher (GPIO 4). At the measurement points messages are also logged to a .CSV file.
5 → 8	Measurements on Raspberry Pi 5. The message crosses GPIOS: 5 → 6 → A → B → C → D → 7 → 8	Roundtrip message tilt	Before publishing message on topic /tilt_subscriber (GPIO 5) and in callback of topic /tilt_publisher (GPIO 8). At the measurement points messages are also logged to a .CSV file.

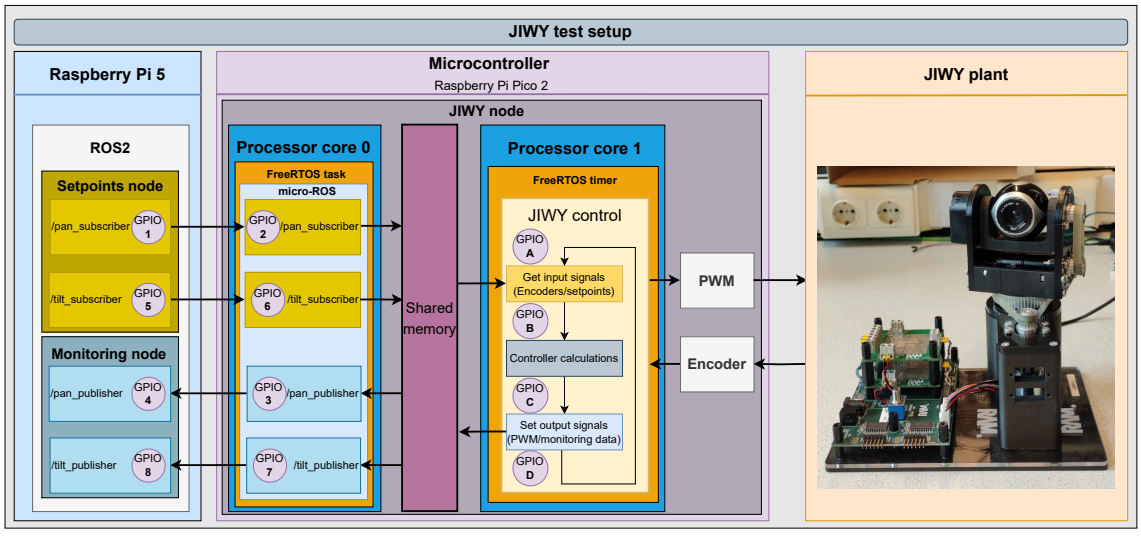
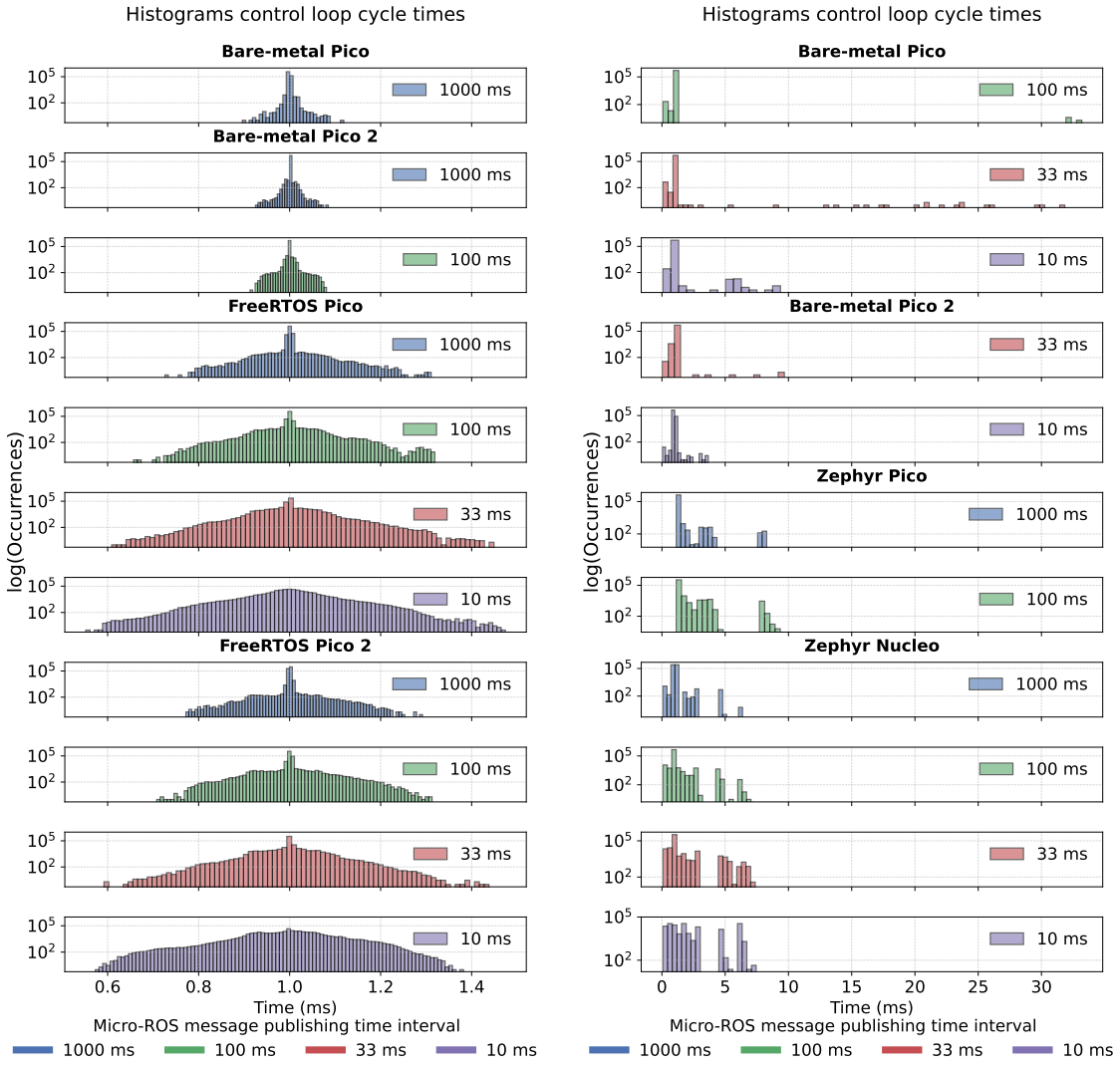


Figure 5.10: JIWY test setup

Result control loop cycle times

Figure 5.11 shows histograms of control loop cycle times for targets running JIWY_control code at 1 ms while communicating with a Raspberry Pi 5 single-board computer as shown in Figure 5.10. The time in the figures indicates the communication time interval used by the ROS publishers. This is therefore the interval used by both the Raspberry Pi 5 for publishing setpoint data on subscriber topics and the microcontroller target for publishing monitoring data on publisher topics.
Measurements are taken with publishing time intervals of 1000 ms (1 Hz), 100 ms (10 Hz), 10 ms (100 Hz), and 33 ms (30 Hz). The first three time intervals evaluate performance across different time scales, while 33 ms matches the setpoint generation rate of the JIWY webcam. Figure 5.11a shows all measurements with outliers lower than 2 ms. These are the situations that have most practical use case in a robot setup. Figure 5.11b shows all targets with outliers larger than 2 ms.



(a) Histograms control loop cycle times < 2 ms

(b) Histograms control loop cycle times > 2 ms

Figure 5.11: Control loop cycle times at 1 ms with varying ROS publishing time intervals

Discussion control loop cycle times

The results show that the bare-metal implementations overall have the best firm real-time performance for high micro-ROS communication time intervals of 1000 ms for the Pico, and 1000

ms and 100 ms for the Pico 2.
However, when the communication time interval is lowered, occasional jitter spikes occur. Before this happens, the control loop completes successfully, and micro-ROS on the other core remains functional, indicating the Pico operates normally except for the repeated interrupt timer. So, the jitter spikes are caused by the Raspberry Pi SDK repeated interrupt timer that temporarily stops operating. The jitter spikes, occurring at high communication rates, are likely caused by interference between the interrupt timer and the Raspberry Pi Pico's USB interface, which relies on an interrupt-driven task that, under high load, consumes more processing time, disrupting the control loop's interrupt timer.
All targets using FreeRTOS are stable under all communication load conditions without any spikes. When the communication time interval is lowered the jitter increases and the outliers, become slightly larger. The performance of the Pico is comparable to the Pico 2.
The Zephyr targets all perform worse compared to the bare-metal and FreeRTOS targets. Using the Pico, it was only possible to perform measurements with rates of 1000 ms and 100 ms. On the Nucleo, measurements could be done with all communication time intervals, but these lower time intervals significantly affected the cycle time.
Only the EPG-targets with FreeRTOS on the Pico and Pico 2 fulfil the 1 ms firm real-time control loop and 33 ms soft real-time communication requirement.

Result control loop cycle times Pico 2: 100 ms bare-metal and 33 ms FreeRTOS

The 100 ms Pico 2 bare-metal implementation does not meet the 33 ms soft real-time requirement, but offers strong firm real-time performance with low jitter. Therefore, this target is investigated alongside the 33 ms Pico 2 FreeRTOS. The control loop cycle time, 400 worst control loop cycles, and control loop computation times are investigated in more detail for those situations. Detailed results of the other situations are also available in Appendix F.
The Pico 2 bare-metal situation with 100 ms micro-ROS communication is shown in Figure 5.12, Figure 5.13, and Figure 5.14. The Pico 2 FreeRTOS situation with 33 ms micro-ROS communication is shown in Figure 5.15, Figure 5.16, and Figure 5.17.

Bare-metal Pico 2 - 100 ms

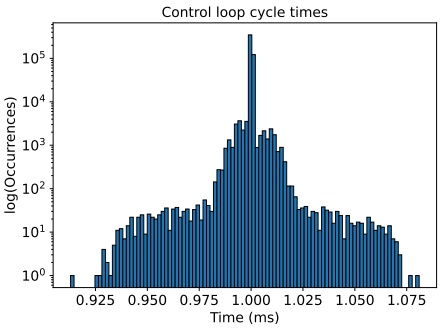


Figure 5.12: Control loop cycle times

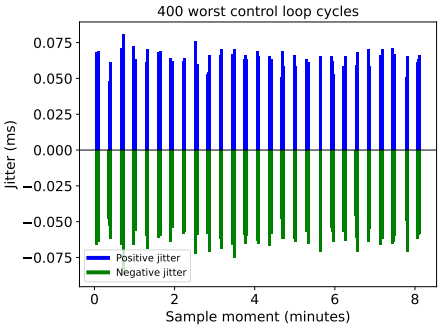


Figure 5.13: 400 worst control loop cycles

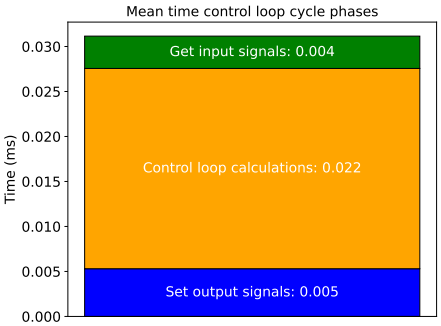


Figure 5.14: Control loop computation time

FreeRTOS Pico 2 - 33 ms

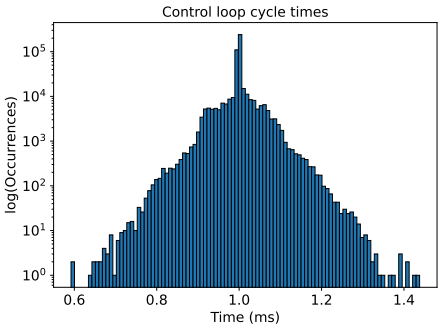


Figure 5.15: Control loop cycle times

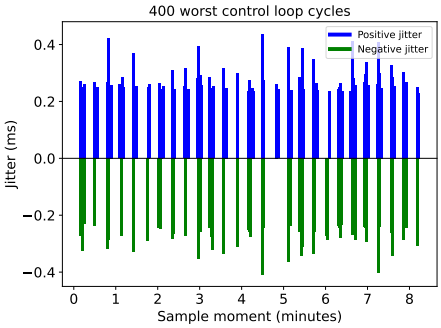


Figure 5.16: 400 worst control loop cycles

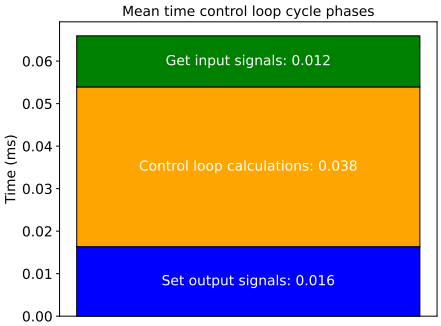


Figure 5.17: Control loop computation time

Discussion control loop cycle times Pico 2: 100 ms bare-metal and 33 ms FreeRTOS

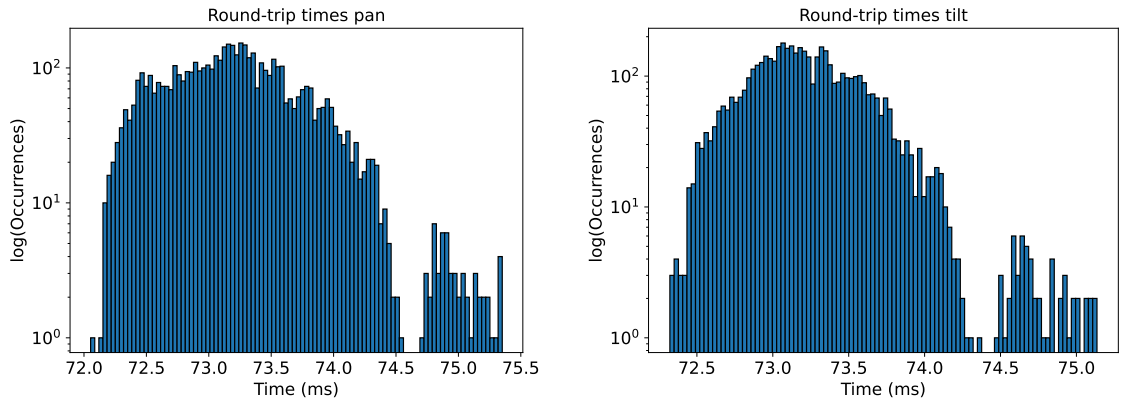
Figure 5.12 shows the control-loop cycle times for the Pico 2 bare-metal with micro-ROS communication with a time interval of 100 ms. The figure shows a minimal amount of jitter at the cycle time of 1 ms with maximum fluctuations of about 0.075 ms. The repeated Pico timer works with hardware interrupts that enable strict timing. Figure 5.15 shows the cycle times for the FreeRTOS situation with micro-ROS communication at 33 ms. This figure shows more jitter with maximum fluctuations of about 0.4 ms. The histogram has a bell-shape, which suggests a normal distribution with occasional fluctuations. This situation uses the FreeRTOS scheduler with a FreeRTOS timer for handling the cycles. Since this runs in software, it is less strict compared to hardware interrupt-based timing, as it can be affected by scheduling delays and context switches.

Figure 5.13 shows the 400 worst cycle times for the bare-metal situation. Both the positive and negative jitter are plotted. This shows that a positive jitter is directly followed by a negative

jitter. For the bare-metal situation this is caused by the behaviour of the repeated Pico timer. The timer constantly tries to keep the control loop time at 1 ms. If in one cycle the control loop started too late this is compensated by starting the next cycle earlier. The same behaviour is shown in Figure 5.16 for the FreeRTOS situation. In this situation the FreeRTOS scheduler is responsible for the real-time loop and compensates for delays. Figure 5.14 shows the control loop computation time for the bare-metal situation, which takes about 0.031 ms. In the FreeRTOS situation shown in Figure 5.17 the computation takes about 0.066 ms. As expected, most time of the computation is spent on doing the control loop calculations. Both situations provide sufficient safety margin for the control loop calculations, which consume about 3% of the cycle time using bare-metal and about 7% of the cycle time using FreeRTOS, which is required for high-speed loops. Although the 100 ms Pico 2 bare-metal implementation provides strong firm real-time performance with low jitter and may be a good candidate for scenarios where 100 ms communication is sufficient, it does not meet the 33 ms soft real-time requirement. The 33 ms Pico 2 FreeRTOS implementation fully meets the requirements for the control loop.

Results round-trip communication Pico 2: 100 ms bare-metal and 33 ms FreeRTOS

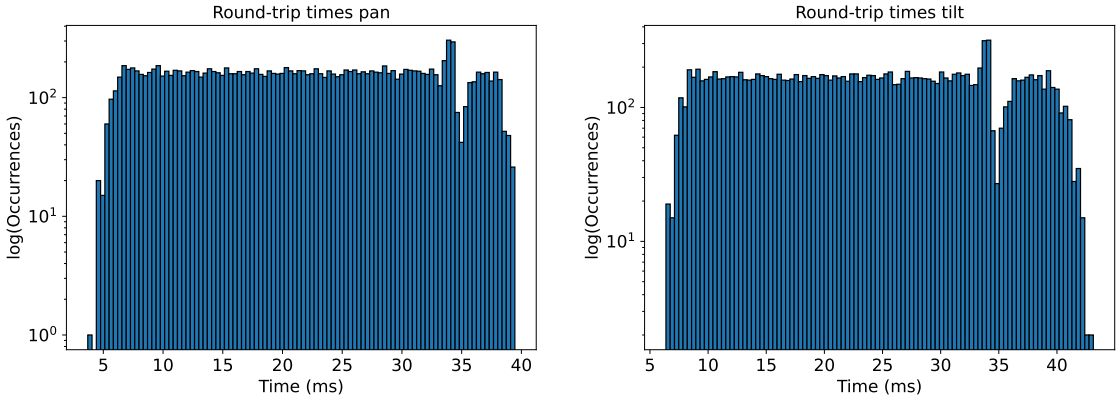
The round-trip communication times for the bare-metal Pico 2 with a publishing time interval of 100 ms are shown for the pan in Figure 5.18a and tilt in Figure 5.18b. The same plots for the FreeRTOS Pico 2 situation with a publishing time interval of 33 ms are shown in Figure 5.19a and Figure 5.19b. Additional measurement results are shown in Appendix G.



(a) Round-trip times pan

(b) Round-trip times tilt

Figure 5.18: Bare-metal Raspberry Pi Pico 2 at 100 ms



(a) Round-trip times pan

(b) Round-trip times tilt

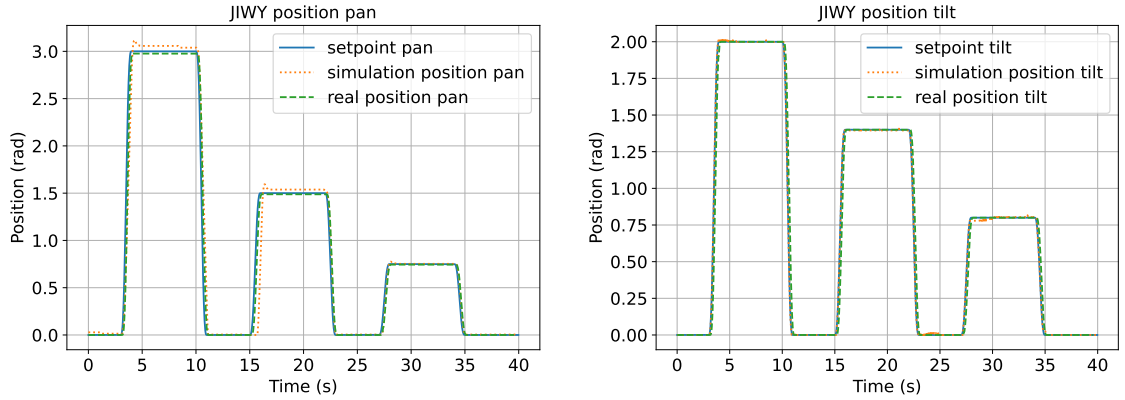
Figure 5.19: FreeRTOS Raspberry Pi Pico 2 at 33 ms

Discussion round-trip communication Pico 2: 100 ms bare-metal and 33 ms FreeRTOS

The histograms in Figure 5.18a and Figure 5.18b show the round-trip times for bare-metal Pico 2 communication, where both the Raspberry Pi 5 and microcontroller publish data every 100 ms. The measurement begins when the Raspberry Pi 5 publishes the message. There is some overhead as the message is received by micro-ROS, passed via shared memory to the control loop, and passed back before being published by the Pico 2. The figure shows that all messages have a round-trip time between 72 ms and 76 ms. This implies that there are no significant delays within the round-trip and new data is available within the 100 ms timeslot the micro-controller uses to publish data.
The same histograms are shown in Figure 5.19a and Figure 5.19b for the FreeRTOS Pico 2, where data is published every 33 ms. This situation shows that it takes between 3 ms and 45 ms for the round-trip to complete. If a message is not passed back within the 33 ms timeframe, it will be passed in the next one. However, this is not an issue, as it is not expected that the message will always be passed back within the Pico 2 publisher's time interval, due to the overhead of the round-trip communication. The timing between the message's arrival and when the next timeframe is ready to send it back, along with the overhead, determines the overall round-trip time. As the communication time intervals lowers and the timeframes shorten, small delays in the round-trip communication become more noticeable. When a timeframe is missed, the message is passed in the following cycle.
In both situations micro-ROS communication and control loop computation takes place. The RTT plots show no large outliers, which indicates communication is stable within the set time interval. The results therefore provide no indication that micro-ROS usage could lead to system instability.

Result motion profile tracking

To compare the behaviour of the JIWY model and the real-world JIWY target deployment, a 20-sim simulation was conducted. In the simulation JIWY receives setpoints from a motion profile, which has also been implemented in a ROS 2 Setpoints node on the Raspberry Pi 5 for real-world comparison. The node publishes setpoints simultaneously on the pan and tilt subscriber topics with a time interval of 33 ms.
The Pico 2 publishes messages simultaneously on both publisher topics at a time interval of 33 ms, which are logged by a separate Monitoring node on the Raspberry Pi 5.
Figure 5.20 shows motion profile setpoints along with the simulation and real position for the pan in Figure 5.20a and tilt in Figure 5.20b.



(a) Motion profile tracking pan

(b) Motion profile tracking tilt

Figure 5.20: JIWY following motion profile

Discussion motion profile tracking

The motion profile tracking test, using the FreeRTOS with Pico 2 EPG-target, combines both micro-ROS communication and control loop performance to execute the tracking task. Figure 5.20 shows motion profile setpoints for the pan and tilt, along with the simulation and real position. The figure shows consistency between the simulation and real-world results for both the pan and the tilt demonstrating the applicability of the EPG-generated project in a real-world application.

5.2.4 Conclusion

For the use of the EPG-generated project with the robots in the RaM laboratory, the requirements of 1 ms firm real-time control loop and 33 ms soft real-time communication must be met.
The bare-metal EPG-targets using the Raspberry Pi Pico and Raspberry Pi Pico 2 offer the best performance for micro-ROS communication in the ping-pong test. In the JIWY control-loop measurements it appears that the bare-metal implementations have the best firm real-time performance for high micro-ROS communication time intervals of 1000 ms for the Pico, and 1000 ms and 100 ms for the Pico 2. However, the bare-metal EPG-targets do not meet the 33 ms soft real-time communication requirement, because when the communication time interval is lowered, occasional jitter spikes occur.
The FreeRTOS EPG-targets using the Raspberry Pi Pico and Raspberry Pi Pico 2 had slightly lower performance compared to the bare-metal implementations, but in all cases the performance was good. The FreeRTOS implementations remain stable with low micro-ROS communication time intervals. With lower time intervals the control loop jitter increases, but no high jitter spikes were measured.
The Zephyr EPG-targets using the Raspberry Pi Pico and STM32 Nucleo-H743ZI performed overall worse compared to the bare-metal and FreeRTOS EPG-targets. The Zephyr EPG-targets do not fulfil the 1 ms firm real-time control loop and 33 ms soft real-time communication requirements.
Overall, FreeRTOS on the Pico and Pico 2 are the only EPG-targets that fully meet the 1 ms firm real-time control loop and 33 ms soft real-time communication requirements.

5.3 Network testing

5.3.1 Introduction

This section conducts tests to evaluate the use of multiple EPG-generated projects as ROS 2 nodes in a distributed network. A round-trip time test and a motion profile tracking test are conducted to assess whether performance remains stable in a distributed network, while also exploring the flexibility of adding multiple EPG-targets in a ROS 2 system.

5.3.2 Ping pong

Setup

The ping-pong measurement setup is a round-trip message-passing application that is used to determine the performance of the communication. This setup is an extension of the ping-pong application discussed in Section 5.2.2. The performance variables of interest are again the round-trip time, latency distribution, and the jitter. A diagram of the setup is shown in Figure 5.21. The figure shows two Raspberry Pi single-board computers (Pis) and two Raspberry Pi Pico microcontrollers (Picos). A Pico 2 is connected to a Pi 5 and a Pico is connected to a Pi 4. The Pi 5 and Pi 4 are connected to each other via an Ethernet switch. The communication between the Pi 5 and Pi 4 happens over Ethernet using Fast DDS, which operates over UDP/IP. The communication between the Pis and Picos happens serial over USB 2.0 using XRCE-DDS (explained in Section 2.3.2), with a micro-ROS agent running on the Pis to facilitate communication with the Pico micro-ROS clients. All communication happens with best-effort QoS.
The round-trip consists of three segments. A message is sent from the Pi 5 to the Pico 2, followed by the Pico 2 sending a message to the Pico, and the Pico sends a message back to the Pi 5, completing the round-trip. The callback function at the Pi 5 checks if the received message content is equal to the previously sent message content. Only when the check is successful a new message is directly published starting a new round-trip.
The sequence of actions and measurements is shown in Table 5.4. Each measurement consists of $500 \cdot 10^6$ samples, taken at a rate of 1 MHz. The ping-pong tests are conducted in two separate scenarios, one where both Picos run bare-metal and another where they operate using FreeRTOS. From performance testing Section 5.2 it appeared that EPG-targets with FreeRTOS on the Pico and Pico 2 fully meet the requirements. Furthermore, the bare-metal implementation performed best for 100 ms soft real-time micro-ROS communication. It did not meet the requirement of 33 ms, but it can still be useful for situations with less demanding communication making it also interesting to investigate.

Table 5.4: Ping pong measurement steps

GPIO #	Platform	Action	GPIO toggle
1	Raspberry Pi 5	Start of ping-pong measurement	Before publish function is called.
2	Raspberry Pi 5	ROS2 publisher sends message on /topic ping_pico_2	Directly after publish function completes.
3	Pico 2 microcontroller	micro-ROS subscriber callback receives message on /topic ping_pico_2	Directly at the beginning of the callback function.
4	Pico 2 microcontroller	micro-ROS subscriber callback publishes message on /topic ping_pico. The message goes via the Pi 5, Ethernet switch, and Pi 4 to the Pico.	Directly after the publish function is finished.
5	Pico microcontroller	micro-ROS subscriber callback receives message on /topic ping_pico	Directly at the beginning of the callback function.
6	Pico microcontroller	micro-ROS subscriber callback publishes message on /topic pong_pico. The message goes via the Pi 4 and Ethernet switch to the Pi 5.	Directly after the publish function is finished.
7	Raspberry Pi 5	ROS2 subscriber callback receives message on /topic pong_pico.	Directly at the beginning of the callback function.

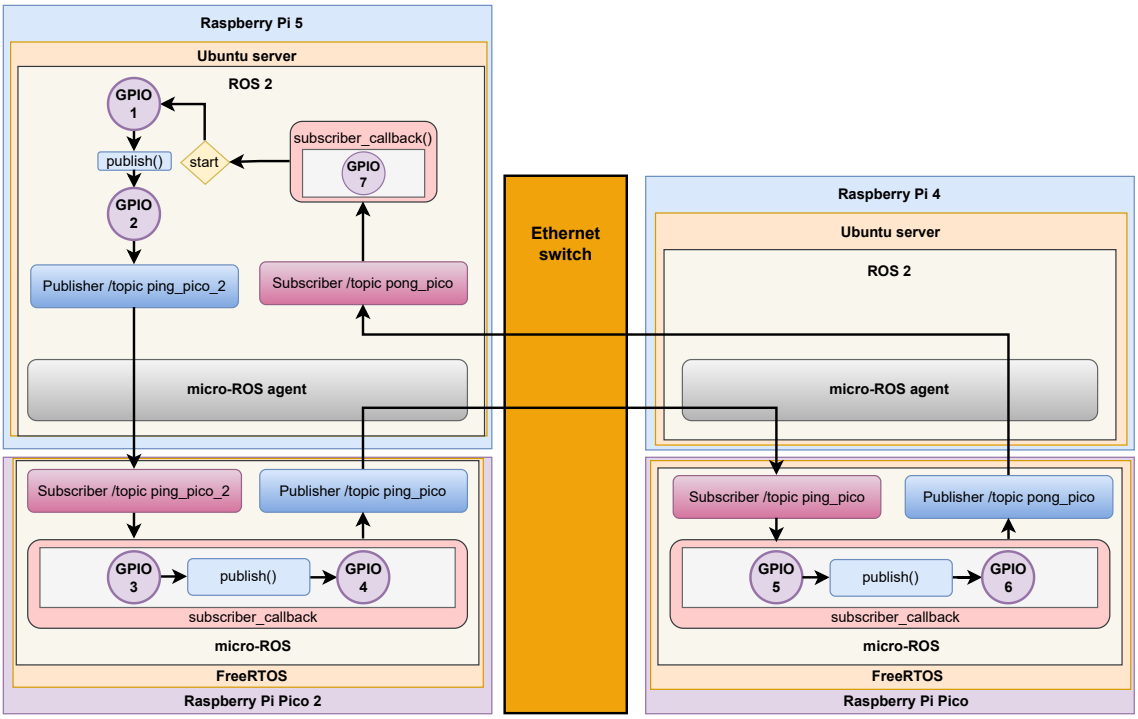


Figure 5.21: Ping-pong setup

Results

The ping-pong tests are conducted in two separate scenarios, one where both Picos run bare-metal and another where they operate using FreeRTOS. The round-trip times, 100 worst round-trips, mean latency distribution, and latency histograms are investigated for those scenarios. The bare-metal results are shown in Figure 5.22, Figure 5.23, Figure 5.24, and Figure 5.25. The FreeRTOS situation is shown in Figure 5.26, Figure 5.27, Figure 5.28, and Figure 5.29.

Bare-metal

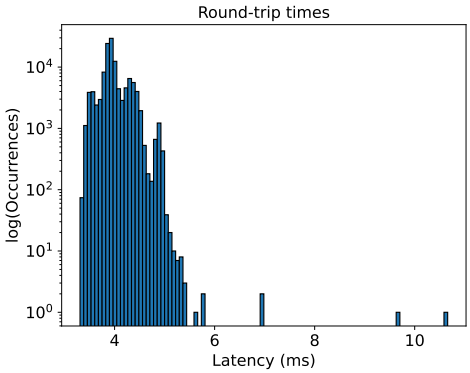


Figure 5.22: RTT

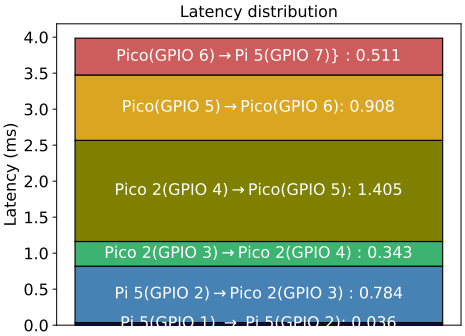


Figure 5.23: Mean latency distribution

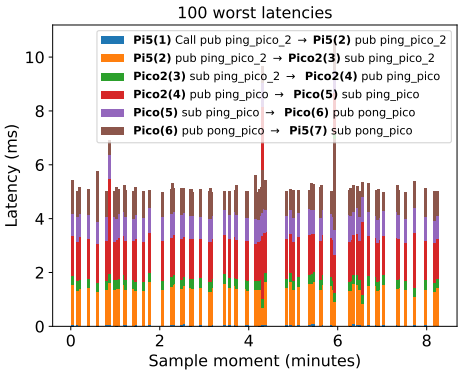


Figure 5.24: 100 worst round-trips

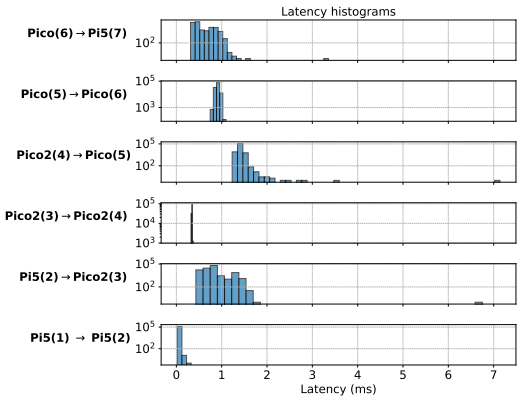


Figure 5.25: Latency histograms

FreeRTOS

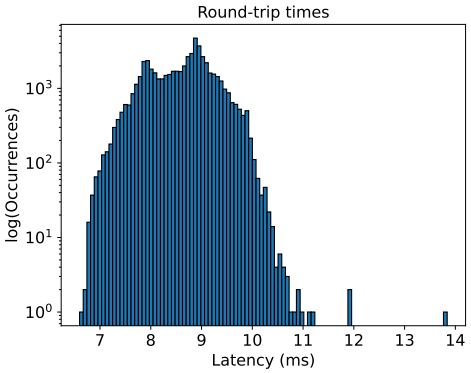


Figure 5.26: RTT

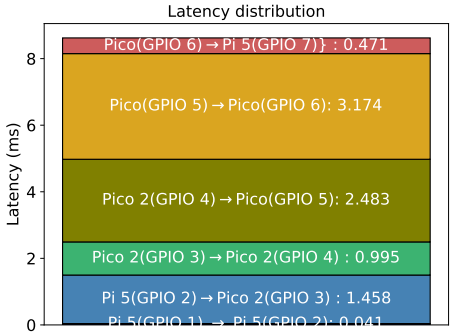


Figure 5.27: Mean latency distribution

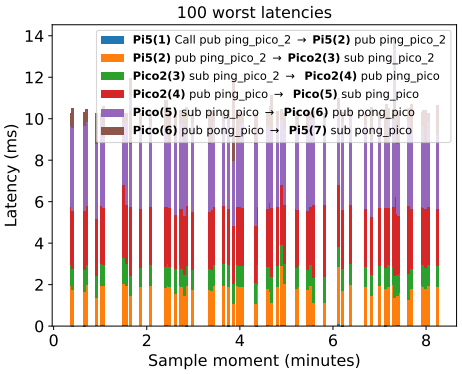


Figure 5.28: 100 worst control loop cycles

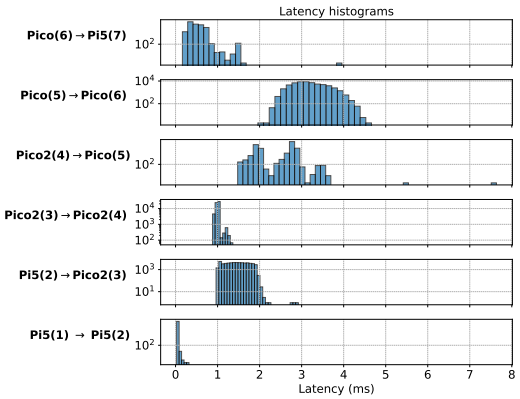


Figure 5.29: Latency histograms

Discussion

Figure 5.22 and Figure 5.26 show histograms of the round-trip times. The histograms show that with bare-metal the round-trips take about 4 ms to complete and with FreeRTOS about 9 ms to complete. According to the soft real-time communication requirement, transmitting setpoint and monitoring data should occur within a 33 ms time interval. Since the round-trip includes multiple communication paths for which this requirement applies, and as the measured round-trip times are below this threshold, no issues are expected.
The results of the latency distribution in Figure 5.23 and Figure 5.27 are consistent with the corresponding segments of the latency distribution discussed in Section 5.2.2. Additionally in this setup data is also sent from the Pico 2 (GPIO4) to the Pico (GPIO 5) via the Pis, which takes 1.4 ms with bare-metal and 2.5 ms with FreeRTOS. This is comparable to the communication times obtained in the performance tests discussed in Section 5.2. The 100 worst latencies in Figure 5.24 and Figure 5.28 also show that incidental outliers do not appear at a particular moment, but randomly. The histograms of Figure 5.25 and Figure 5.29 show the different communication stages. The histograms show that there are few outliers, and the outliers that are present are not high.
The test results provide no indication to expect that introducing multiple nodes in a distributed network will cause any limitations.

5.3.3 Motion profile tracking

Motion profile tracking tests are performed to assess whether performance remains stable in a distributed network, while also exploring the flexibility of adding multiple EPG targets as nodes in a ROS 2 system.
In accordance with the requirements, motion profile tracking is conducted with a soft real-time communication time interval of 33 ms. Therefore, EPG-targets with FreeRTOS on the Pico and Pico 2 are used.
The setup includes two Pis, a Pi 4 and a Pi 5, connected through an Ethernet switch, with both running Ubuntu server 24.04 with ROS 2 Jazzy. Each Pi is also connected to a Pico and a Pico 2, which communicate via a serial USB 2.0 connection using micro-ROS XRCE-DDS. The Picos are connected to JIWY robots, enabling distributed control and coordination of the robots through ROS 2 and micro-ROS. Similar as in Section 5.2.3, motion profiles from 20-sim are used as setpoints to steer four JIWY robots. This setup is tested in three different situations to provide insight in how multiple robots can be managed in networked conditions.

Situation 1 - Motion profile tracking with a shared motion profile

A diagram of situation 1 is shown in Figure 5.31. In this situation the Pi 5 sends four identical motion profiles to all connected Picos, causing the JIWY robots to perform the same movements simultaneously. This test is conducted to investigate the system's capability to synchronise multiple robots through distributed control. It demonstrates that the communication between the Raspberry Pi and the Picos via ROS 2 and micro-ROS is effective, enabling coordinated actions across all robots. The Setpoints node on the Pi 5 publishes the setpoints with a time interval of 33 ms, which mimics the setpoint generation time interval of the JIWY webcam. The Monitoring node is subscribed to topics that receive the current position of all JIWY robots. The Picos publish monitoring data at a time interval of 33 ms. All communication happens with best-effort QoS. Since all Picos are nodes, together with the Setpoints and Monitoring node, the ROS 2 system has 6 nodes in total.

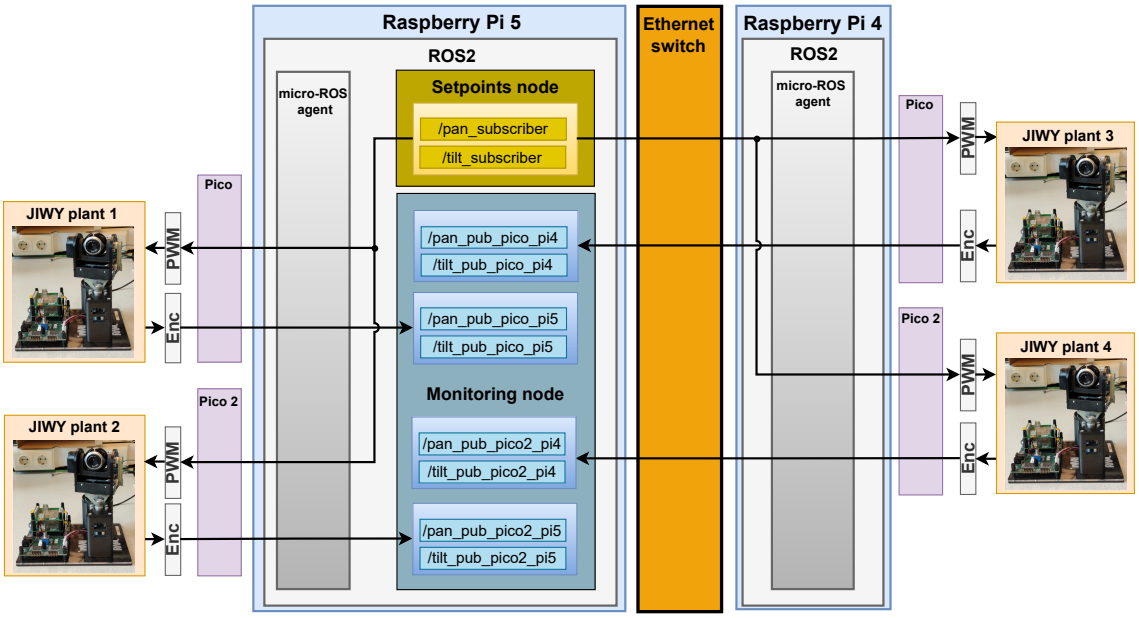
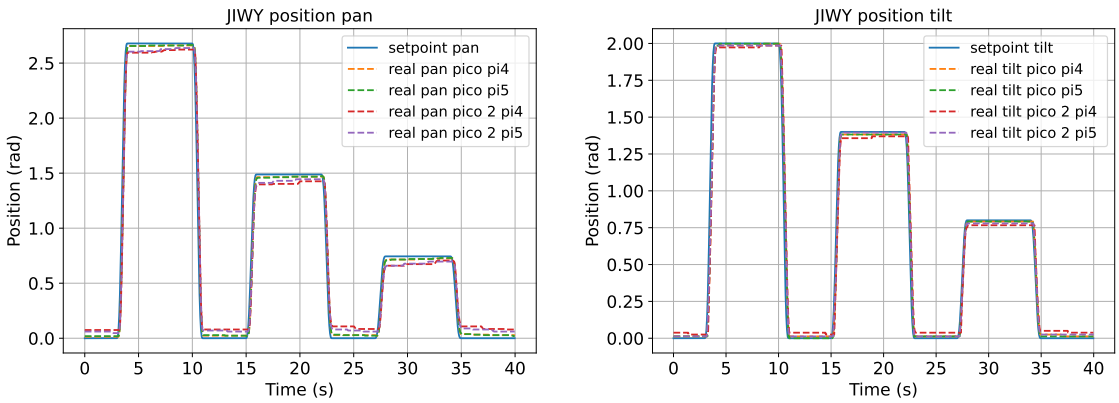


Figure 5.30: Situation 1 - Motion profile tracking with a shared motion profile

Results situation 1 - Motion profile tracking with a shared motion profile

The results of the motion profile tracking test are shown in Figure 5.31, where Figure 5.31a shows motion profile tracking for the pan movement and Figure 5.31b motion profile tracking for the tilt movement. A demo showing the setup in action is provided in the following **video of situation 1**.



(a) Motion profile tracking pan

(b) Motion profile tracking tilt

Figure 5.31: Motion profile tracking

Discussion situation 1 - Motion profile tracking with a shared motion profile

The results in Figure 5.31 show consistency between the motion profile and real-world results for both the pan and the tilt.
This test uses ROS 2 for collecting measurement data. The data is logged asynchronously by the Setpoint node and Monitoring node at the moment data becomes available, which is approximately every 33 ms. The exact time between the publishing of a Setpoint by the Pi 5 and the movement of the JIWYs belonging to the different Picos and Pico 2s can therefore not be determined. It is however possible to estimate this time by first interpolating the target dataset (the Picos and Pico 2s following the setpoints) to align its time scale with the reference dataset

(the provided setpoints), followed by cross-correlation to determine the lag between the datasets. The calculated cross-correlations are shown in Table 5.5. It should be noted that these cross-correlations include both the networking time needed to transmit setpoint and monitoring data between nodes, as well as the settling time needed for the JIWY to adjust to the given setpoint. The cross-correlation is only a match when the setpoint and JIWY positions fully align. The results indicate that the lags are approximately multiples of the communication time interval of 33 ms. This is expected, as this time interval was used to collect the data and therefore determined the resolution of the lags in the cross-correlation. The Pico Pi 4 and Pico Pi 5 have approximately the same position for the pan as the provided setpoint after about 66 ms. The tilt of the Pico Pi 4 has the same position after about 66 ms, while the Pico Pi 5 takes 99 ms, probably because of mechanical differences between JIWYs. The determined values of the Pico 2s are overall higher, which can also be explained by mechanical differences between the JIWY robots. Under steady-state conditions, there is a small error between the setpoint and the actual position. These errors are larger and less consistent for the Pico 2s, influencing the accuracy of the cross-correlation and making those results less reliable. Small issues, such as a slipping gear belt, can cause such errors. These errors are further investigated in situation 2.

Table 5.5: Cross-correlation

	Pan (lag in ms)	**Tilt (lag in ms)**
Pico Pi 4	66	99
Pico Pi 5	66	66
Pico 2 Pi 4	99	133
Pico 2 Pi 5	99	99

The results suggest that communication within the distributed network does not cause significant delays in motion profile tracking. The test shows that the system is capable to synchronise multiple robots through distributed control.

Situation 2 - Split motion profile tracking using two independent motion profiles

This test is conducted to investigate whether, in addition to synchronised control of JIWYs, it is also possible to flexibly control each JIWY node separately while maintaining synchronisation properties. From the results in situation 1 it appeared that the mechanical deviations occurred at the JIWYs connected to Pico 2. Situation 2 is almost similar to situation 1. The difference is that the setpoint node uses two distinct motion profiles to separately control the JIWYs connected to the Pico microcontroller and those connected to the Pico 2 microcontroller. The JIWYs connected to the Pico 2 were then swapped with those connected to the Pico to investigate whether the errors in motion profile tracking are caused by the JIWYs. A diagram of situation 2 is shown in Figure 5.32.

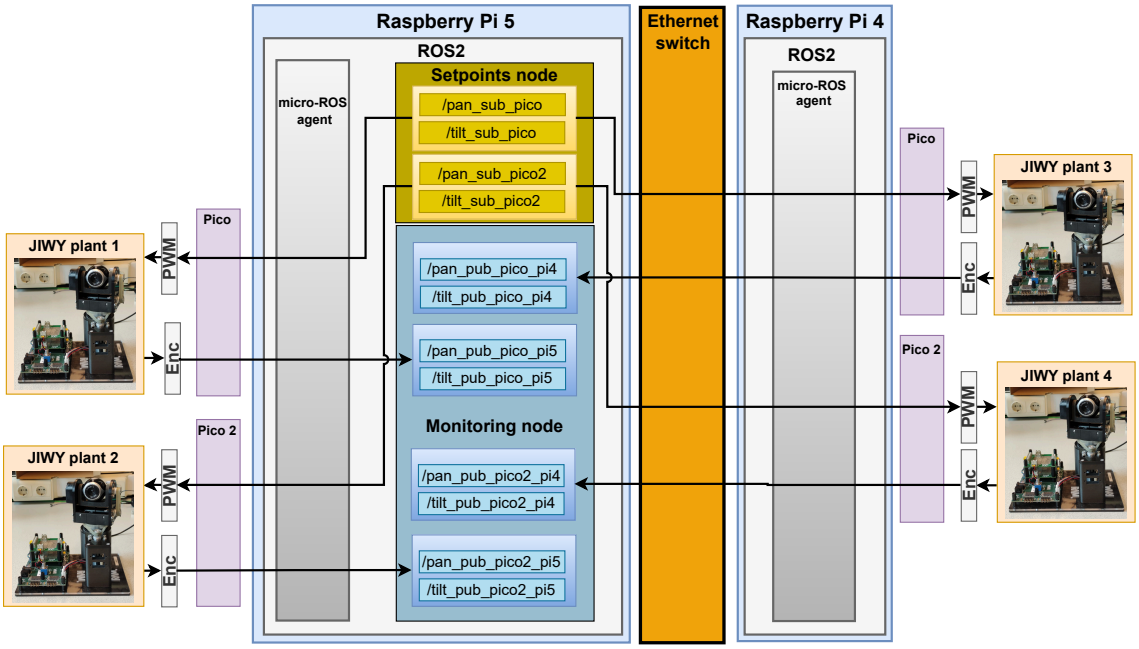
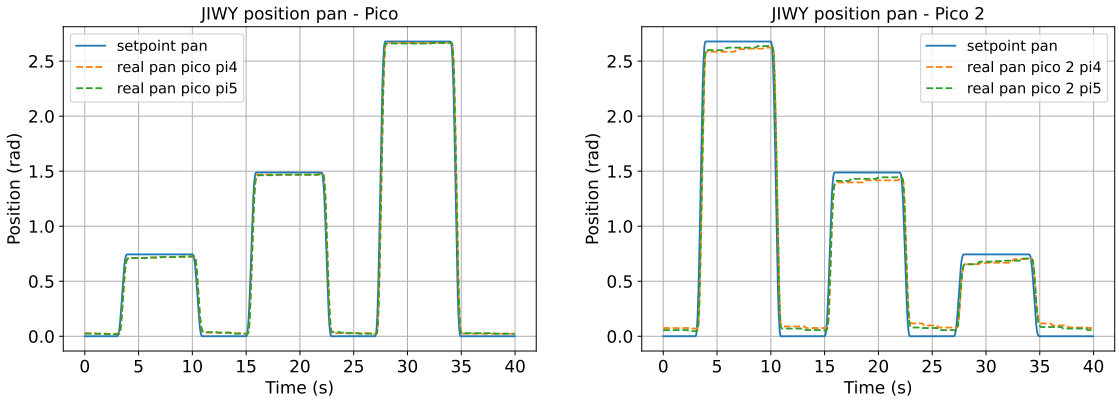


Figure 5.32: Situation 2 - Split motion profile tracking using two independent motion profiles

Results situation 2 - Split motion profile tracking using two independent motion profiles

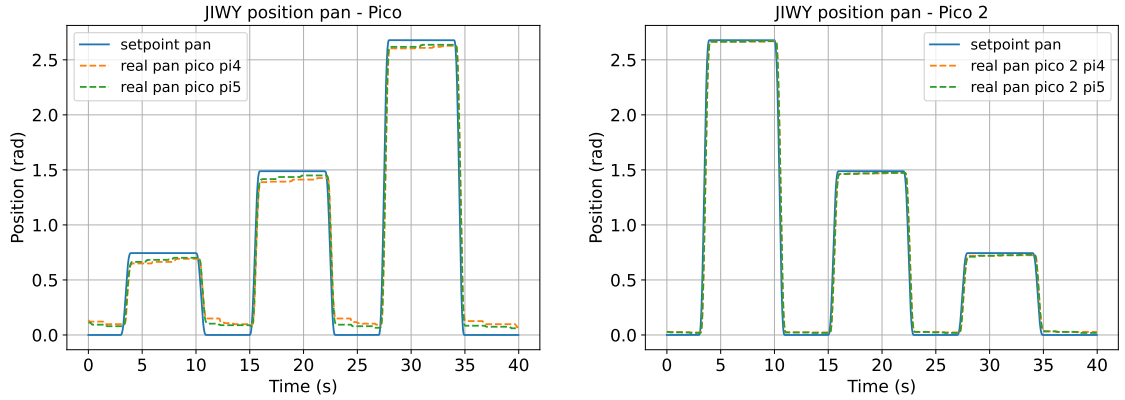
The results of the motion profile tracking for pan are shown in Figure 5.33, with Figure 5.33a showing the motion profile tracking for the JIWYs connected to the Pico and Figure 5.33b for JIWYs connected to the Pico 2. Similarly, Figure 5.34 shows the pan motion profile tracking for the swapped JIWYs, where Figure 5.33a shows JIWYs connected to the Pico and Figure 5.33b shows JIWYs connected to the Pico 2. Similar measurements are shown for the tilt in Appendix H. A demo showing the setup in action is provided in the following **video of situation 2**.



(a) Motion profile tracking Pico

(b) Motion profile tracking Pico 2

Figure 5.33: Motion profile tracking pan



(a) Motion profile tracking Pico swapped JIWYs

(b) Motion profile tracking Pico 2 swapped JIWYs

Figure 5.34: Motion profile tracking pan swapped JIWYs

Discussion situation 2 - Split motion profile tracking using two independent motion profiles

This test shows that in addition to synchronised control of JIWYs, it is also possible to flexibly control each JIWY node separately while maintaining synchronisation properties.
Furthermore, Figure 5.33 shows that the motion profile tracking with JIWYs connected to the Pico 2, shown in Figure 5.33b, has noticeable steady-state errors compared to the JIWYs connected to the Pico in Figure 5.33a. In Figure 5.34 the JIWYs are swapped, which shows that the errors now occur with JIWYs connected to the Pico in Figure 5.34a. These test results show that the steady-state errors are caused by the mechanical errors in the JIWYs and the distributed control functions as expected.

Situation 3 - Teleoperation with mirrored robots

This situation investigates whether teleoperation with motion replication, a demanding networking application, can be achieved using the networking capabilities of EPG-generated projects. Teleoperation involves real-time control of a remote robot by a human operator. This often involves the operator performing physical movements on a passive device that are replicated by an active robot. Teleoperation requires a low-latency network to enable real-time manipulation. If teleoperation works well, this demonstrates that an EPG-target can control another ROS 2 node within the distributed network, which increases the usability of these projects in distributed systems.
A diagram of situation 3 is shown in Figure 5.35. Two JIWYs are used to create a teleoperation setup. One JIWY acts as a passive robot that is used by the operator to provide physical inputs by manually adjusting the pan and tilt. Another JIWY acts as an active robot that replicates the exact movements of the passive JIWY. To demonstrate the scalability of EPG-generated projects, two pairs of JIWYs are used instead of one.
In this setup, the Pico connected to the Pi 4 publishes the current position of the connected JIWY robot as monitoring data. The Pico connected to the Pi 5 uses this monitoring data as setpoint, causing the connected JIWY to mirror the same position. This also applies to the JIWYs with a Pico 2.
To obtain clear measurements, without disturbances from manual movements, a motion profile is used. The Pi 5 Setpoints node publishes a motion profile to the Picos connected to the Pi 4 that steers the JIWYs to the desired position, which mimics the manipulation of passive JIWYs by a remote operator. These Picos publish their current position as monitoring data. The Picos connected to the Pi 5 use this monitoring data as setpoints, causing the JIWYs connected to them to mirror the same position.
Teleoperation, by manual manipulation of the passive JIWY, is shown in the demo video linked

in the results. The motors of the passive JIWYs connected to the Pi 4 were turned off, while the encoders stayed functional for sending position monitoring data to the active JIWYs connected to the Pi 5. Manually manipulating the passive JIWY results in the active JIWY mirroring its movements.

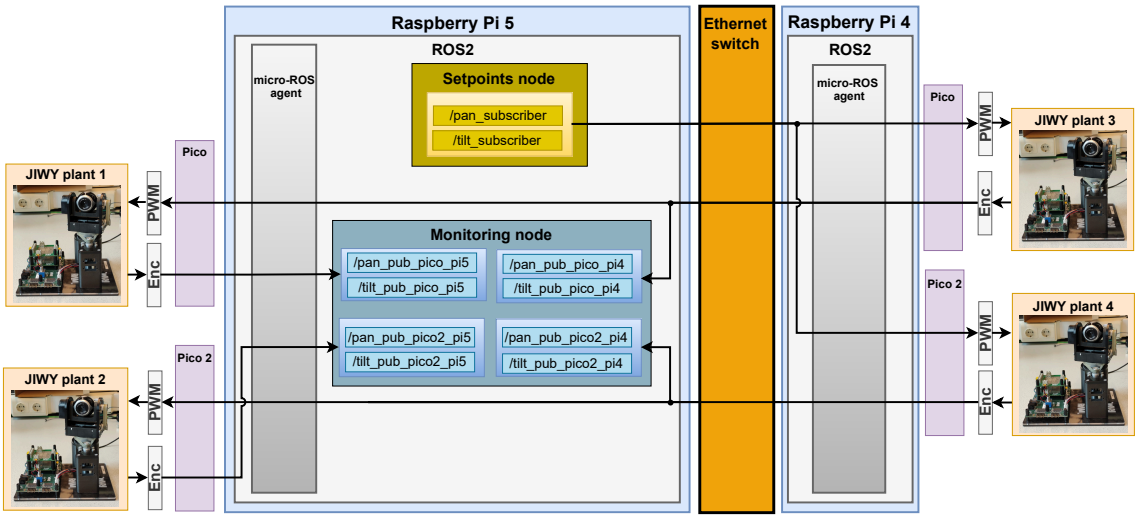
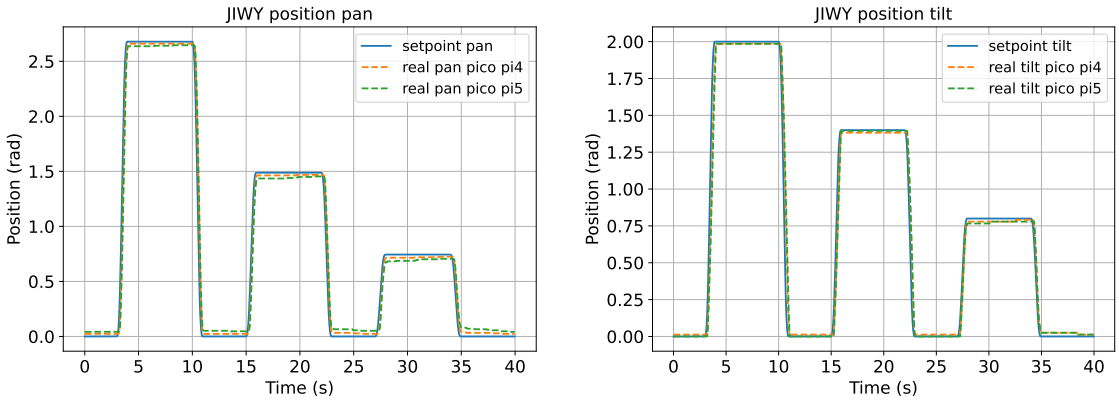


Figure 5.35: Situation 3 - Motion profile tracking with mirrored robot pairs for teleoperation

Results situation 3 - Teleoperation with mirrored robots

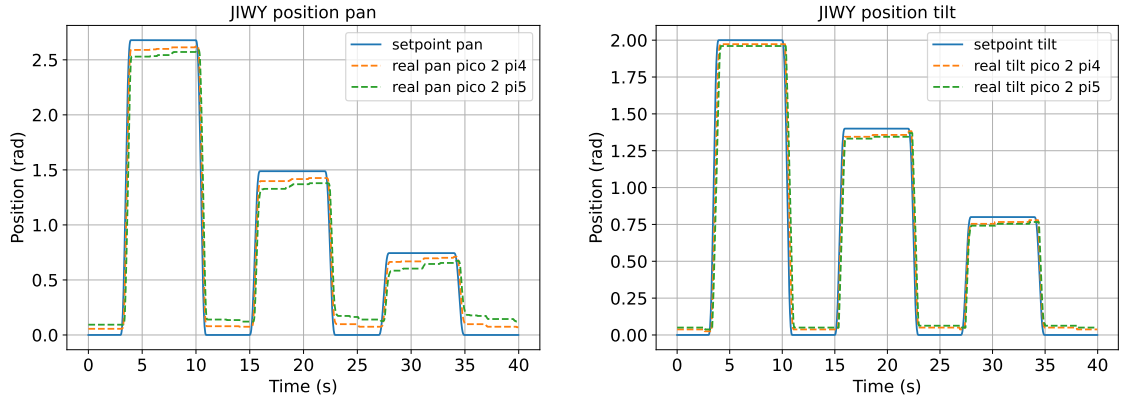
For the Pico the results are shown in Figure 5.36, where Figure 5.36a shows motion profile tracking for the pan movement and Figure 5.36b motion profile tracking for the tilt movement. The figures show how the JIWY with Pico connected to the Pi 5 mirrors the JIWY with Pico connected to the Pi 4. The results for the Pico 2 are shown in Figure 5.37, where Figure 5.37a shows motion profile tracking for the pan movement and Figure 5.37b motion profile tracking for the tilt movement. Similarly to the Pico, these figures show how the JIWY with Pico 2 connected to the Pi 5 mirrors the JIWY with Pico 2 connected to the Pi 4. The JIWYs with mechanical errors are connected to the Pico 2.
A demo showing the setup in action is provided in the **video of situation 3**. This demonstrates that the setup allows one robot to control another and is an example of teleoperation with motion replication.



(a) Motion profile tracking pan

(b) Motion profile tracking tilt

Figure 5.36: Motion profile tracking Pico



(a) Motion profile tracking pan

(b) Motion profile tracking tilt

Figure 5.37: Motion profile tracking Pico 2

Discussion situation 3 - Teleoperation with mirrored robots

The results are shown in Figure 5.36 and Figure 5.37. Figure 5.36 shows how the JIWY with a Pico connected to the Pi 5 mirrors the JIWY with a Pico connected to the Pi 4.
This discussion considers both the deviations in steady state as in transient state.
When considering the steady-state in Figure 5.36, the JIWY connected to the Pi 5 shows a larger deviation from the setpoint, as it not only accounts for its own error but also mirrors the error of the JIWY connected to the Pi 4. Similarly, Figure 5.37 shows how the JIWY with Pico 2 connected to the Pi 5 mirrors the JIWY with Pico 2 connected to the Pi 4. In this figure, due to the aforementioned mechanical errors, the steady-state errors are larger compared to the Pico situation.
When considering the transient state, the delay between the Picos is investigated. To determine the delay between the movement of the JIWY belonging to the Pico connected to the Pi 4 and the movement of the JIWY belonging to the Pico 2 connected to the Pi 5, the 3 measurement points shown in Figure 5.38 can be used. Figure 5.38 is a simplified representation of Figure 5.35 with measurement points:

1. Setpoint node, which logs setpoint data published by the Pi 5.
2. Monitoring node, which logs position data published by the Pico connected to the Pi 4.
3. Monitoring node, which logs position data published by the Pico connected to the Pi 5.

This test uses ROS 2 for collecting measurement data. The data is logged asynchronously by the Setpoint node and Monitoring node at the moment data becomes available, which is approximately every 33 ms. The exact time between the movement of the JIWY belonging to the Pico connected to the Pi 4 and the movement of the JIWY belonging to the Pico 2 connected to the Pi 5 can therefore not be determined. Similarly to situation 1, it is however possible to estimate this time by first interpolating the target dataset (the Pico following the setpoints) to align its time scale with the reference dataset (the provided setpoints), followed by cross-correlation to determine the lag between the datasets. The result of this cross-correlation is shown in Table 5.6. The results indicate that the lags are approximately multiples of the communication time interval of 33 ms. This is expected, as this time interval was used to collect the data and therefore determined the resolution of the lags in the cross-correlation. The cross-correlation lag of measurement points 2 → 3 for both the pan and tilt is 0. Since this lag is 0, it can be estimated that the delay between JIWYs following each other is at most 33ms. This is supported by subtracting cross-correlation 1 → 2 from 1 → 3 to approximate the cross-correlation 2 → 3, which results in 33ms. So, the delay between the movement of the JIWY belonging to the Pico connected to the Pi 4 and the movement of the JIWY belonging to the Pico connected to the Pi

5 is at most 33 ms. The cross-correlation is only determined for the Picos, because the measurements using the Pico 2s have deviations caused by mechanical errors of the JIWYs, which makes the cross-correlation method unsuitable for these measurements.

Table 5.6: Cross-correlation

	Pan (lag in ms)	**Tilt (lag in ms)**
1 → 2	66	100
2 → 3	0	0
1 → 3	99	133

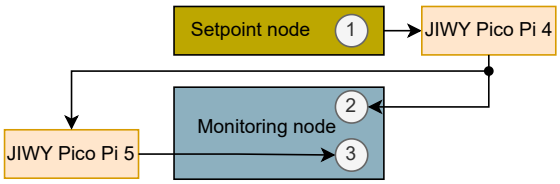


Figure 5.38: Situation 3 simplified

This test demonstrates that each node using an EPG-generated project can function as an independent component within the ROS 2 system, capable of being controlled by another node, while also controlling other nodes, enhancing the system's flexibility.

5.3.4 Conclusion

The conducted tests demonstrate the effectiveness of a distributed system using EPG-generated projects deployed on EPG-targets, which function as ROS 2 nodes, in achieving stable motion tracking, real-time synchronisation, and teleoperation. The results indicate that network delays are small, with round-trip time measurements demonstrating consistent communication performance using bare-metal and FreeRTOS EPG-targets. The motion tracking tests confirm that the system can reliably coordinate multiple robots using a shared motion profile. The teleoperation tests established that the system supports real-time motion replication with low-latency communication, enabling robots to mirror movements synchronously without significant delays.

5.4 Test compatibility with a different mechatronic system

5.4.1 Introduction

In the previous sections all tests are executed using the JIWY robot. One of the requirements of this thesis is 'There should be compatibility with different mechatronic systems.', which is investigated by testing the EPG compatibility with the RELbot, a differential-drive robot shown in Figure 5.39. The RELbot is intensively used in assignments of courses of the MSc Robotics programme.

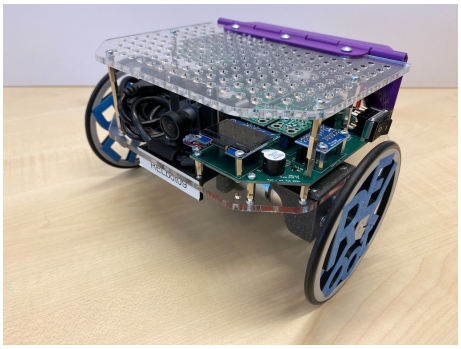


Figure 5.39: RELbot plant

Based on a 20-sim model of the RELbot an EPG-project is generated, which is made operational for the RELbot within a single day to demonstrate the flexibility of the EPG and explore what functionality could be achieved within this limited timeframe.

As this robot has not been previously used, this represents a complete walkthrough of the EPG. The EPG-project is flashed on a Raspberry Pi Pico 2, which controls the RELbot, demonstrating the EPG as a software tool for rapid prototyping. The aim is not to evaluate performance, but to validate the EPG's compatibility with another mechatronic system.
In addition to the three networking situations in Section 5.3, two additional distributed networking situations are discussed that were achieved within the timeframe. A motion profile situation in which three RELbots are synchronously steered by a common motion profile and a teleoperation situation where two RELbots are directly steered by another RELbot.

5.4.2 Situation 4: Motion profile tracking with shared motion profile

A diagram of situation 4 is shown in Figure 5.40. The setup includes two Pis, a Pi 4 and a Pi 5, connected through an Ethernet switch, with both running ROS 2 Jazzy. The Pi 5 is connected to a Pico 2, which communicates via a serial USB 2.0 connection using micro-ROS XRCE-DDS. The Pi 4 is connected to two Pico 2s. The Pico 2s are connected to RELbot robots, enabling distributed control and coordination of the robots through ROS 2 and micro-ROS. All Pico 2s are flashed with an EPG-generated project.
In this situation the Pi 5 sends three identical motion profiles to all connected Pico 2s, causing the RELbot robots to perform the same movements simultaneously. This test is conducted to investigate the system's capability to synchronise multiple robots through distributed control. The Setpoints node on the Pi 5 publishes the setpoints with a time interval of 33 ms, which mimics the setpoint generation time interval of the RELbot webcam. All communication happens with best-effort QoS. Since all Pico 2s are nodes, together with the Setpoints node, the ROS 2 system has 4 nodes in total.

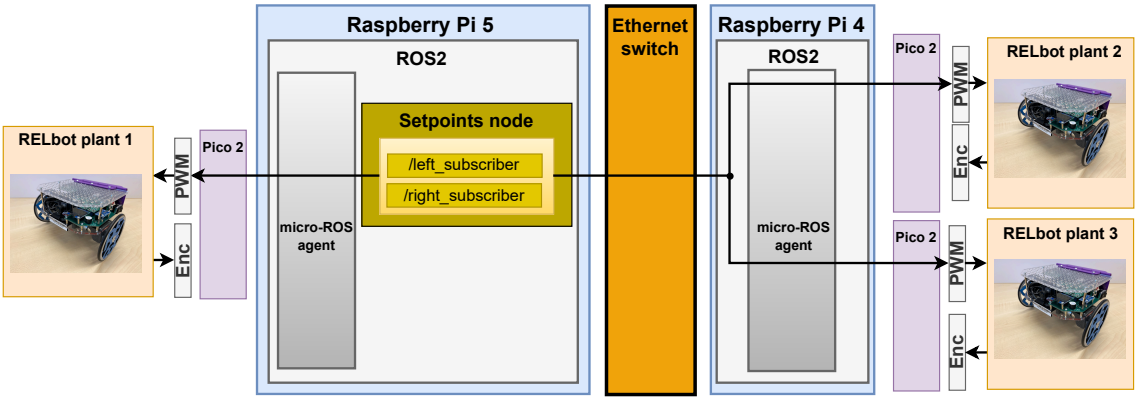


Figure 5.40: Situation 4 - Motion profile tracking with shared motion profile

5.4.3 Situation 5: Mirrored robots by teleoperation

This situation uses almost the same setup as in situation 4. The difference is however that the setpoints are not from the Setpoints node, but from the RELbot with Pico 2 connected to Pi 5. The PWM signals have been disabled, leaving only the encoders functional. The encoder data is provided as setpoints to the RELbots with Pico 2s connected to the Pi 4. This setup is a 3 node ROS 2 system, where each node is present on a Pico 2. The setup is therefore an example of a Pico 2 node steering other Pico 2s.

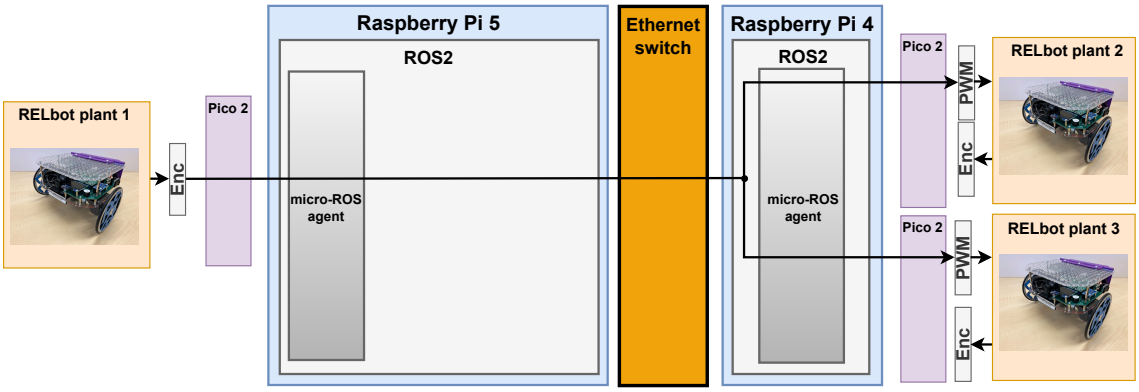


Figure 5.41: Situation 5 - Mirrored robots by teleoperation

5.4.4 Results and discussion situation 4 and situation 5

The implementation of the RELbot is completed within a single day, demonstrating the EPG's flexibility of integration with another mechatronic system. The EPG-project is successfully flashed onto multiple Pico 2s, enabling the RELbot robots to operate as intended. This confirms the compatibility of the EPG with another robotic platform beyond JIWY.
In Situation 4, a distributed control setup is tested, where a Pi 5 sends identical motion profiles to all connected Pico 2s, causing the three RELbot robots to move synchronised. This is shown in **video of situation 4**.
In Situation 5, a single RELbot provides encoder data as setpoints to other RELbots, demonstrating a Pico 2 node steering other Pico 2 nodes, showing teleoperation with RELbots. This is shown in **video of situation 5**.
Both situations are successfully executed, validating the compatibility of the EPG with different mechatronic systems.

6 Conclusions and Recommendations

6.1 Conclusion

The first goal of this thesis is: Design a method to automate the translation from a robot-controller model to code that uses the ROS 2 ecosystem and supports real-time control. This goal is achieved by designing an automation method based on micro-ROS running on a microcontroller with a Raspberry Pi single-board computer using ROS 2. The EPG uses 20-sim model-based C-Code as input and follows the EPG steps: select target, connect ports, and code generation, which results in an EPG-generated project that, when compiled, can be flashed onto a microcontroller, ensuring seamless deployment and real-time control.
The second goal is: Design and implement a software tool based on the automation method. This goal is achieved by implementing the automation method in an EPG software tool with a GUI that guides users through the EPG steps. The EPG has been extensively used for generating EPG projects for numerous tests, thereby providing a validation of its functionality.
The third goal is: Test the performance of code, generated by the software tool, with a robot. This goal is achieved by testing the EPG on six target configurations on the Raspberry Pi Pico, Raspberry Pi Pico 2, and STM32 Nucleo-H743ZI. Tests determine the performance of micro-ROS communication using a ping-pong round-trip message passing application. Configurations are deployed on the JIWY robot, to assess communication, control loop performance, and motion tracking behaviour in both simulation and real deployment. Overall, FreeRTOS on the Pico and Pico 2 are the EPG-targets that fully meet the 1 ms firm real-time control loop and 33 ms soft real-time communication requirements.
The fourth goal is: Test the performance and stability of a distributed network with multiple robots using code generated by the software tool. This goal is achieved by testing the performance and stability of a distributed ROS 2 network incorporating four JIWY robots, controlled by EPG-targets, as ROS 2 nodes. The conducted tests demonstrate the effectiveness of a distributed system using EPG-generated projects deployed on EPG-targets, in achieving stable motion tracking, real-time synchronisation, and teleoperation.
An EPG-target implementation is also tested on three RELbot robots, performing motion profile tracking and teleoperation, with the setup developed through rapid prototyping in one day.
All thesis requirements to achieve the goals are verified, as summarised in Table 6.1. The requirements in the table include a hyperlink to the corresponding verification.

Table 6.1: Verification of requirements

Requirement	**Verified**
The EPG-generated project *must* use the ROS 2 ecosystem.	✓
The EPG-generated project *must* support a robotic setup that uses a Raspberry Pi single-board computer.	✓
20-sim model-generated code of embedded control software *must* be directly implementable on mechatronic systems.	✓
The EPG-generated project *must* have real-time capabilities.	✓
There *should* be compatibility with different mechatronic systems.	✓
Support for networking between robotic components *should* be implemented.	✓
There *should* be support for double-precision floating-point calculations.	✓
Minimal user input *should* be required.	✓
The EPG *should* have good maintainability.	✓
There *should* be support for the use of software components.	✓
The EPG and EPG-generated project *could* use common programming languages.	✓

The main goal of this thesis is to develop an MDD software tool to automate the translation from a robot-controller model to code for an embedded device, enabling both real-time control of a mechatronic system and integration within the ROS 2 ecosystem for networking. In conclusion, the main goal is achieved by the EPG, which enables seamless integration of robots within the ROS 2 ecosystem by deploying EPG-generated projects on EPG-targets that control the robots. These EPG-targets function as ROS 2 nodes, abstracting system complexity while ensuring soft real-time communication (33 ms) using micro-ROS, and firm real-time control loop management (1 ms) using 20-sim model-based control software.

6.2 Recommendations

Testing the EPG for improvements

To further evaluate the usability of the EPG, it is recommended that students test it with various robot platforms in different operational scenarios. This can help identify potential limitations and opportunities for optimisation. Based on the results, improvements that enhance the EPG's functionality can be implemented.

Test the EPG on a distributed system with inhomogeneous robots

Test the EPG in a distributed scenario involving multiple inhomogeneous robots to evaluate its adaptability and robustness in real-world scenarios. For example, the Production Cell available in the RaM laboratory. The Production Cell is a setup that simulates a plastic injection moulding machine. It has six automated units that transport blocks.

Investigate control loop jitter spikes in bare-metal implementations

The Pico and Pico 2 bare-metal implementations do not meet the 33 ms soft real-time requirement, but offer strong firm real-time performance with low jitter for 1000 ms and 100 ms micro-ROS communication time intervals.
Tests with the bare-metal Raspberry Pi Pico and Raspberry Pi Pico 2 implementation showed spikes in the control loop times with low micro-ROS communication time intervals.
The suspected cause is a problem in the way the Raspberry Pi SDK repeated interrupt timer is implemented in the SDK. The cause of these outliers under high loads could be investigated in future research.

A EPG user guide

This chapter provides instructions on how the EPG can be used. The EPG is a Python package that enables a rapid prototyping process from modelling and simulation in 20-sim to direct target deployment for a networked real-time mechatronic setup with full support for micro-ROS and ROS2. The project directory can by found at: https://git.ram.eemcs.utwente.nl/huiskesdv/embedded_project_generator, which includes README.md files with instructions for setting up the EPG and using the different targets. This appendix provides an overview of those instructions.
This package provides an architecture to efficiently generate embedded code projects targetting the Raspberry Pi Pico (bare-metal, FreeRTOS and Zephyr), Raspberry Pi Pico 2 (bare-metal, FreeRTOS) and Nucleo H743ZI (Zephyr). The package is designed to be easily extendable for new targets. This user guide primarily focuses on the bare-metal and FreeRTOS targets, as they are considered most practical. Additional instructions for using the Zephyr targets are provided in the README.md files located in the EPG directory.

A.1 Software installation

A.1.1 Installing the EPG

These instructions have been tested on a Windows 11 machine with Python 3.12.4 installed. The EPG Python package has no special dependencies related to this Python version and is therefore usable with equal or higher versions of Python. For use of the EPG Python package, Python has to be installed globally on the system or be added to the system PATH.
In order to use the EPG the entire repository has to be cloned and implemented within 20-sim. The target is shown directly in 20-sim when the directory is cloned in the `C:\Program Files (x86)\20-sim xx\Ccode` folder. Another option is to add the path of the folder by clicking in 20-sim on "Tools ⇒ Options ⇒ Folders⇒ C-Code Folders" and adding the path to the cloned folder. The EPG directory can be cloned by running:

```
git clone https://git.ram.eemcs.utwente.nl/huiskesdv/embedded_project_generator.git
```

After cloning the repository an "Embedded-Project generator" target is listed in the 20-sim C-Code Generation Target List. The EPG is implemented in a Python package to avoid any path-related issues, allowing the repository to be placed at each desired location, enabling easy installation. The EPG can be installed using pip by running the following command:

```
pip install -e /path/to/package
```

This can for example be done as follows when the terminal is opened in the directory where the EPG was cloned:

```
pip install -e .\embedded_project_generator
```

By using the -e option, changes made to the package code are directly available without having to re-install the package, which is convenient for developement. After following these steps the EPG is ready for use.

A.1.2 Installing micro-ROS agent on single-board computer

In order to use micro-ROS a micro-ROS agent should be installed in ROS 2. These steps have been tested with ROS 2 Jazzy on a Raspberry Pi 4 (Ubuntu server 24.04), Raspberry Pi 5 (Ubuntu server 24.04) and a laptop (Ubuntu 24.04 desktop). These steps assume that ROS 2 is already installed. The build steps can be found on the following page: https://micro.ros.org/docs/tutorials/core/first_application_linux/
Summary of the build steps:

- Source the ROS 2 installation:

```
source /opt/ros/$ROS_DISTRO/setup.bash
```

- Create a workspace and download the micro-ROS tools:

```
mkdir microros_ws
cd microros_ws
git clone -b $ROS_DISTRO https://github.com/micro-ROS/micro_ros_setup.git
src/micro_ros_setup
```

- Update dependencies using rosdep:

```
sudo apt update && rosdep update
rosdep install --from-paths src --ignore-src -y
```

- Install pip:

```
sudo apt-get install python3-pip
```

- Build micro-ROS tools and source them:

```
colcon build
source install/local_setup.bash
```

- Download micro-ROS-Agent packages

```
ros2 run micro_ros_setup create_agent_ws.sh
```

- Build step

```
ros2 run micro_ros_setup build_agent.sh
source install/local_setup.bash
```

A.1.3 Running micro-ROS agent on single-board computer

After installation the micro-ROS agent can be run with the following commands:

- First source ROS2 within the terminal:

```
source /opt/ros/$ROS_DISTRO/setup.bash
```

- Within the micro_ROS workspace source local_setup.bash to source the micro-ROS agent:

```
source install/local_setup.bash
```

- Start the micro-ROS agent (note that the baudrate specified after the -b option has no influence on the actual data transmission speeds. The targets always operate at USB 2.0 speeds. However, this command aligns with the micro-ROS documentation. When leaving it out default settings are used.):

```
ros2 run micro_ros_agent micro_ros_agent serial -b 115200 --dev /dev/ttyACM0
```

In this command /dev/ttyACM0 defines the USB port to which the microcontroller is connected. On most devices this is by default /dev/ttyACM0, but when the microcontroller appears on another port this should be adjusted accordingly.

A.2 Using the EPG

A.2.1 EPG-project - generation, compilation and flashing

Generate an EPG-project from a 20-sim model (General explanation with JIWY example).

These instructions explain how the EPG can be generate an EPG-generated project for a 20-sim model using JIWY as example. Also a **demo video** is available showing these steps.

- The EPG enables the translation from model-to-code. Therefore, a JIWY 20-sim model is needed to generate code. In the embedded_project_generator folder there is a DEMO/JIWY folder with a JIWY.emx file. The JIWY.emx file contains a 20-sim model, which has been tested in 20-sim 5.1. After opening the file in 20-sim the model shown in Figure A.1 is opened. The JIWY_Control part of this model is going to be used for C-Code generation. Open the 20-sim simulator (click on the icon with a red square shown in Figure A.2) and open C-Code Generation in the simulator (click on icon with red square shown in Figure A.3). When the EPG is installed it is shown in the target list as Embedded-Project Generator (as shown in Figure A.4). Select the Embedded-Project Generator in the target list, the JIWY_control submodel and an output directory where the resulting project should be created. Then click on OK, which opens the EPG GUI.

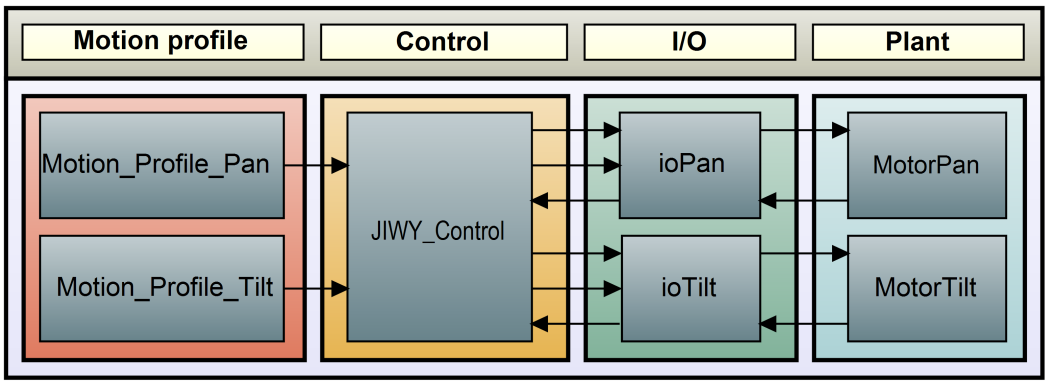


Figure A.1: JIWY 20-sim model

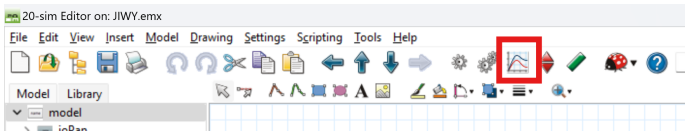


Figure A.2: Open 20-sim simluation window

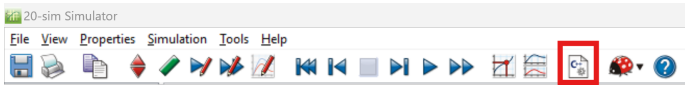


Figure A.3: Open C-Code generation window

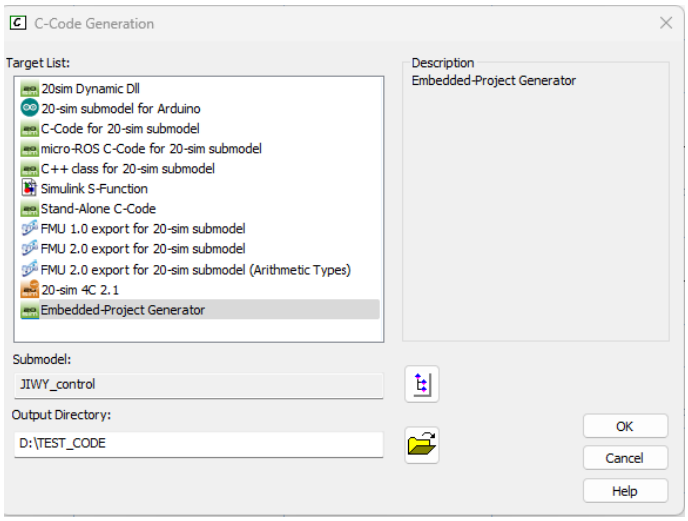


Figure A.4: C-Code generation window

- The EPG GUI shown in Figure A.5 shows a list with all available targets. From this list select the desired target.

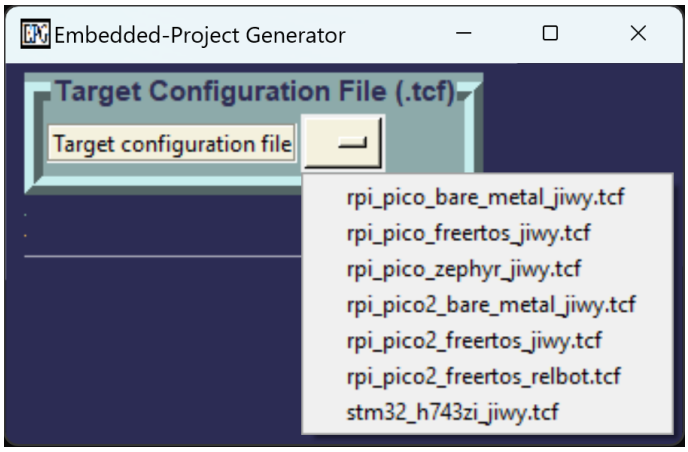
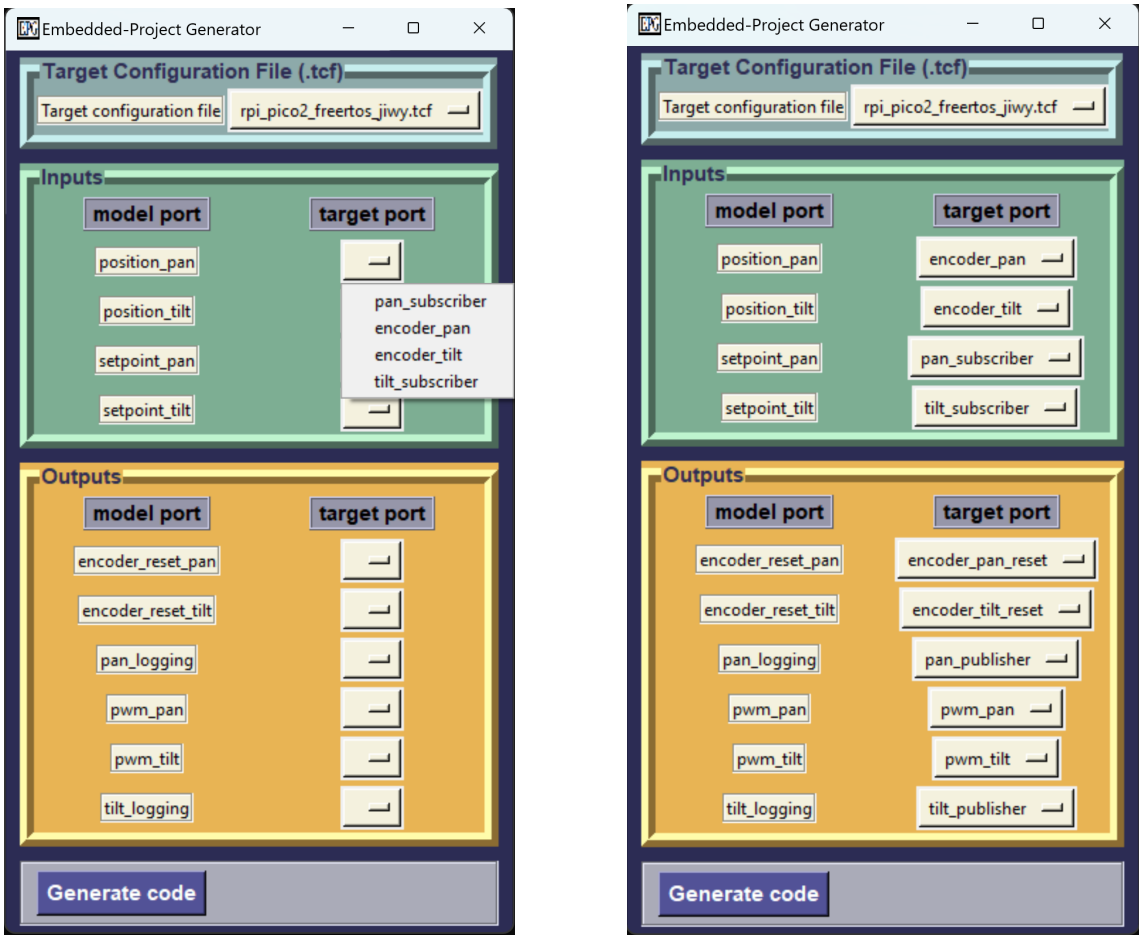


Figure A.5: Target selection EPG GUI

- After selecting the target the GUI shows all model ports along with the target ports available in the target, which is shown in Figure A.6a. Connect all the model ports to the corresponding target ports as shown in Figure A.6b. After having connected all ports click on Generate code. This generates an EPG-project in the previously selected output directory.



(a) Select target port from dropdown menu

(b) All model ports connected with target ports

Figure A.6: Connect model ports with target ports by selecting target ports from EPG GUI dropdown options

Generate an EPG-project from a RELbot 20-sim model.

Generating an EPG-project for the RELbot follows the same steps as generating an EPG-generated project for JIWY. However, now the RELbot.emx 20-sim model in the DEMO/RELbot folder has to be used instead of the JIWY.emx model. The RELbot 20-sim model is shown in Figure A.7.

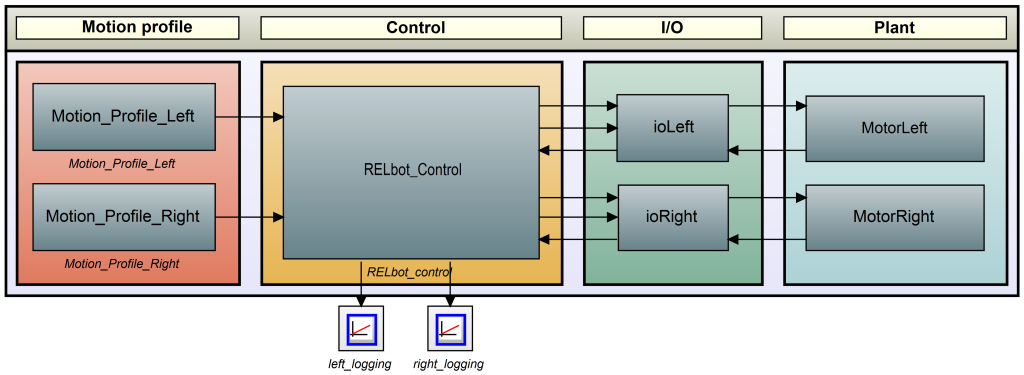


Figure A.7: RELbot 20-sim model

In the 20-sim C-Code Generation Target list select the Embedded-Project Generator and as submodel select RELbot control as shown in Figure A.8

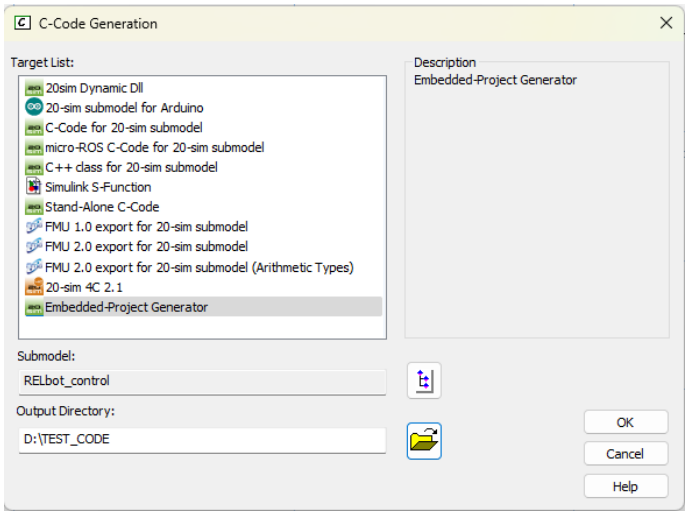


Figure A.8: C-Code generation window RELbot

The RELbot has been implemented for the EPG as an EPG target for the Raspberry Pi Pico 2 using FreeRTOS. Therefore, select the rpi_pico2_freertos_relbot.tcf file in the EPG and connect all model ports to the target ports as shown in Figure A.9.

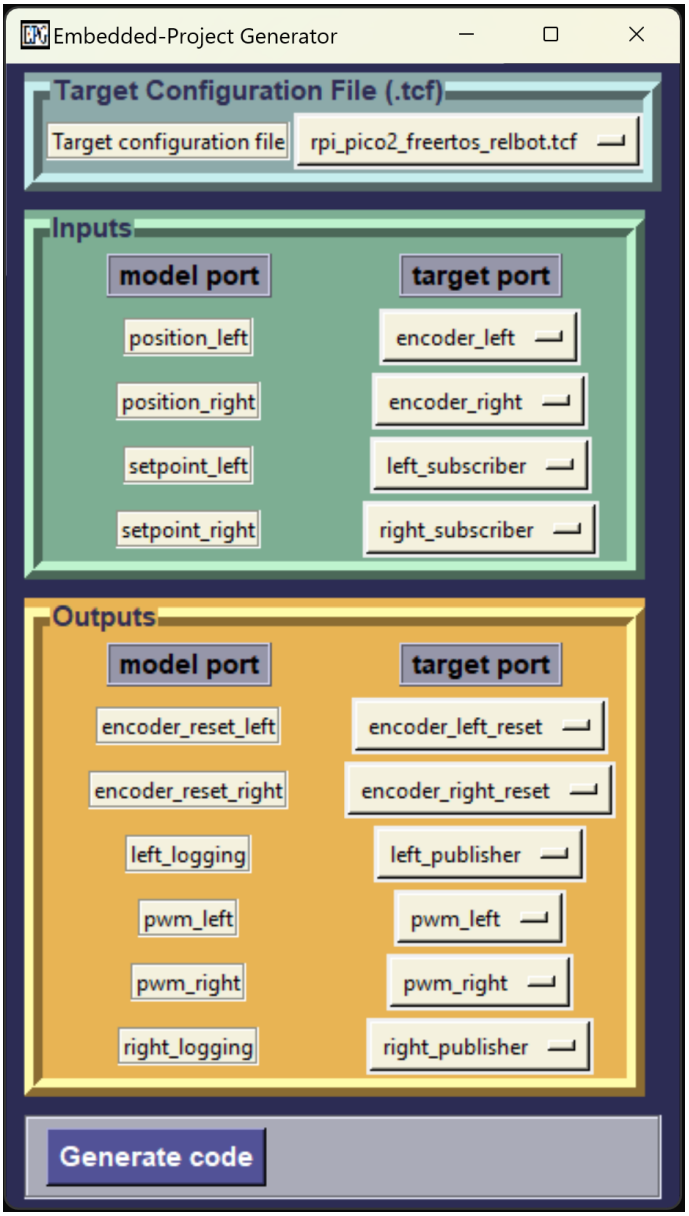


Figure A.9: EPG GUI RELbot

Compile and flash an EPG-generated project (bare-metal and FreeRTOS)

The EPG-generated project is a fully configured project that is ready for compilation. These instructions are for the rpi_pico_bare_metal_jiwy.tcf, rpi_pico_freertos_jiwy.tcf, rpi_pico2_bare_metal_jiwy.tcf, and rpi_pico2_freertos_jiwy.tcf EPG-targets.
The generated project is created such that it can be conveniently used within the Visual Studio Code text editor. The projects are fully compatible with the free official Raspberry Pi Pico Visual Studio Code extension. After installing and opening the extension it is possible to import a project by clicking on "Import Project" as shown in Figure A.10. This opens the menu in Figure A.11, which makes it possible to select a project and import it. Subsequently the project can be compiled, flashed and debugged by clicking on the corresponding options in the extension. The Visual studio code extension automatically fetches all dependencies, but on Windows PC's it might be needed to install GCC for compilation.
It is possible to run the project on the Pico with or without debugger. After the compilation step a .uf2 executable is generated in the build folder that has to run on the Pico. It is now

possible to drag and drop this file on the Pico. Note that without debugger it is not possible to use the flash project and debug project options in the VS Code extension.

- **Flash without debugger:**
  - Hold the bootsel button on the Pico for 2 seconds while plugging in the usb cable in your pc. The Pico will appear as a thumbdrive on your pc.
  - After the compilation step a .uf2 executable is generated in the build folder that has to run on the Pico. It is now possible to drag and drop this file on the Pico.
  - The Pico project is now ready to be used!
- **Flash with debugger:** The advised debugger is the official Raspberry Pi Debug Probe. It is however also possible to flash another Pico to function as a debugger by flashing the pico with the debugprobe .uf2 or configuring pins on a Raspberry Pi 4B/5 to function as SWD interface.
  - Connect the debugger to the SWD interface of the Pico.
  - Use "Flash Project (SWD)" in the VS extension to only flash the project and "Debug project" to both flash and debug the project.

The Pico project is now ready to be used! Start the micro-ROS agent on your device and connect the USB cable of the flashed Pico. The Pico is now a node within the ROS 2 system.

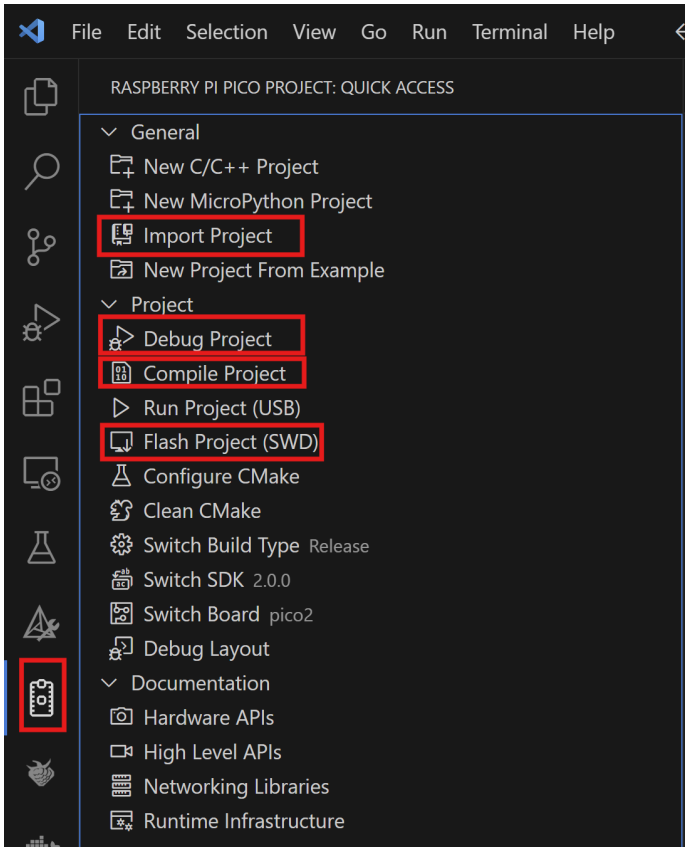


Figure A.10: VScode Pico extension

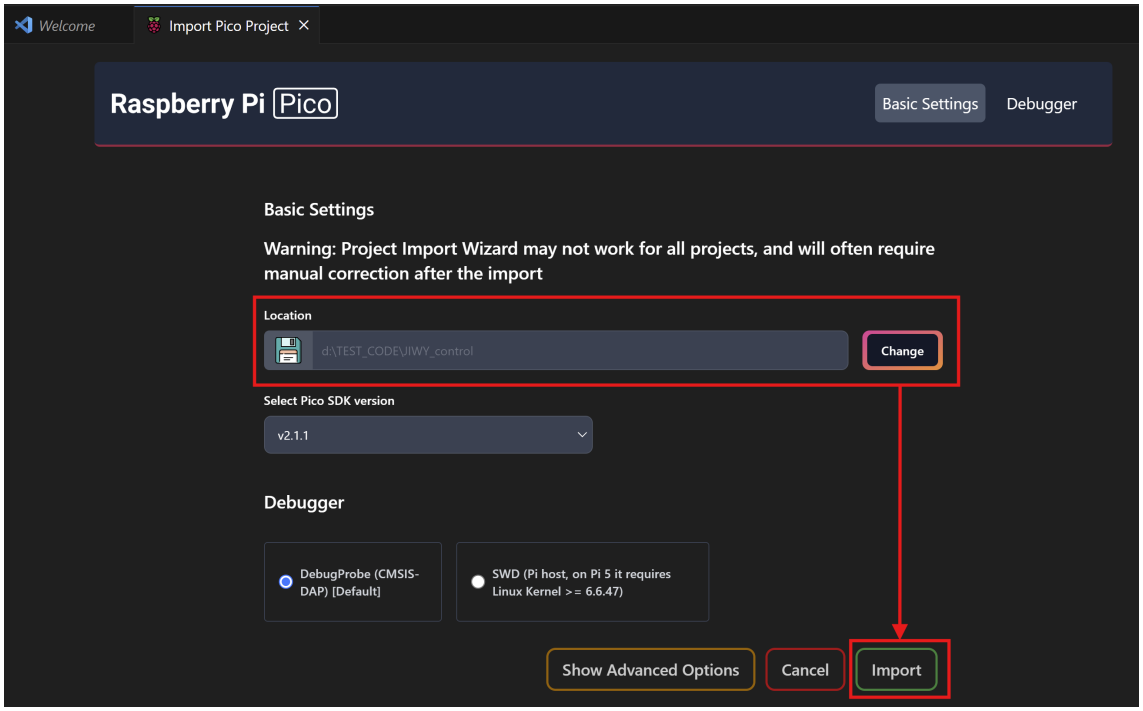


Figure A.11: Import EPG project

A.2.2 Updating software versions

Each EPG-generated project is configured with a CMakeLists.txt which is used to import and configure the project in VSCode. It also contains the compile instructions of the project. This file also automatically fetches micro-ROS code, aswell as the FreeRTOS-Kernel code for FreeRTOS targets, from remote repositories. The projects are now coded to always fetch micro-ROS Jazzy and a specific version of FreeRTOS, which is the latest version at the time of writing this thesis. The same versions are always fetched to ensure the EPG-generated projects remain functional, even when changes are made to the remote repositories that may not directly be compatible.
When it is desired to use newer versions of micro-ROS or FreeRTOS this can be accomplished by looking for the FetchContent_Declare statement in the CMakeLists.txt file and update the GIT_TAG to the desired version. By setting the GIT_tag to "main" the latest version of the remote repository is automatically fetched.

A.2.3 EPG-target wiring

Connecting the Pico/Pico 2 to JIWY

All Picos follow the same wiring scheme to connect to the JIWY. The pins present on the JIWY are shown in Figure A.12. Table A.1 shows the connections from the JIWY to the Pico/Pico 2 for the Yaw/Pan and Table A.2 shows those connections for the Pitch/Tilt. The Pico has multiple GNDs and both the 3V3_EN and 3V3(OUT) can be used for powering the JIWY. The 3V3_EN pin is a pin that is by default 3.3V, but when pulled to ground disables the Pico. It can therefore be used as voltage source, but it is better to use 3V3(OUT). In total four 3V3 pins of JIWY need to be connected and only two are by default available. Therefore using a breadboard is recommended.

Pins Yaw/Pan					
1 PWM DIR A	**3** PWM VAL	**5** ENC A	**7** ENC B	GND	3V3
2 PWM DIR B	**4**	**6**	**8**	GND	3V3

Pins Pitch/Tilt					
1 PWM DIR A	**3** PWM VAL	**5** ENC A	**7** ENC B	GND	3V3
2 PWM DIR B	**4**	**6**	**8**	GND	3V3

Figure A.12: Pins on JIWY robot

Table A.1: Connection wiring JIWY to Pico Yaw/Pan

JIWY PIN	Pico/Pico 2 GPIO pin
ENC A	10
ENC B	11
PWM VAL	2
PWM DIR A	7
PWM DIR B	8
GND	GND
GND	GND
3V3	3V3(OUT)
3V3	3V3(OUT)

Table A.2: Connection wiring JIWY to Pico Pitch/Tilt

JIWY PIN	Pico/Pico 2 GPIO pin
ENC A	12
ENC B	13
PWM VAL	4
PWM DIR A	5
PWM DIR B	6
GND	GND
GND	GND
3V3	3V3(OUT)
3V3	3V3(OUT)

Connecting Pico 2 to RELbot

The Pico 2 can directly be connected to the RELbot via the PMOD cables on the RELbot. Figure A.13 shows a schematic of the PMOD cable header layout. Table A.4 shows the connections from RELbot PMOD P1 to the Pico 2 and Table A.3 shows the connections from RELbot PMOD P2 to the Pico 2.

PMOD1					
PWM A	PWM VAL	ENC A	ENC B	GND	3.3V
PWM B				GND	3.3V

PMOD2					
PWM A	PWM VAL	ENC A	ENC B	GND	3.3V
PWM B				GND	3.3V

Figure A.13: Pins on RELbot robot

Table A.3: Connection wiring RELbot PMOD P2 to Pico

RELbot PIN	Pico/Pico 2 GPIO pin
ENC A	10
ENC B	11
PWM VAL	2
PWM DIR A	7
PWM DIR B	8
GND	GND
GND	GND
3V3	3V3(OUT)
3V3	3V3(OUT)

Table A.4: Connection wiring RELbot PMOD P1 to Pico

RELbot PIN	Pico/Pico 2 GPIO pin
ENC A	12
ENC B	13
PWM VAL	4
PWM DIR A	5
PWM DIR B	6
GND	GND
GND	GND
3V3	3V3(OUT)
3V3	3V3(OUT)

A.2.4 Use EPG-target deployments in ROS 2

JIWY EPG-target deployment

After having:

- Installed micro-ROS
- Flashed the Pico or Pico 2 with an EPG-generated project.
- Connected the JIWY to the Pico or Pico 2

JIWY can be directly controlled from ROS 2. Using micro-ROS JIWY is completely integrated in ROS 2 as a ROS 2 node, just like other ROS 2 nodes.
First make sure that the micro-ROS agent is running on the single-board computer (i.e. Raspberry Pi 4). This is done by running:

- First source ROS2 within the terminal:

```
source /opt/ros/$ROS_DISTRO/setup.bash
```

- Within the micro_ROS workspace source local_setup.bash to source the micro-ROS agent:

```
source install/local_setup.bash
```

- Start the micro-ROS agent:

```
ros2 run micro_ros_agent micro_ros_agent serial -b 115200 --dev /dev/ttyACM0
```

Then the Pico or Pico 2 can be connected to the single-board computer via USB. The EPG-target is automatically recognised by the micro-ROS agent and ready for use.
At the moment no EPG-target is connected the micro-ROS agent will repeatedly show "Serial port not found". After connection the micro-ROS client (EPG-target) is initialised as shown in Figure A.14.

Figure A.14: Establishing connection

By running "ROS 2 node list" it can be verified that the JIWY_node is now available in ROS 2. And by subsequently running "ROS 2 topic list" all topics of JIWY_node are displayed as shown in Figure A.15.

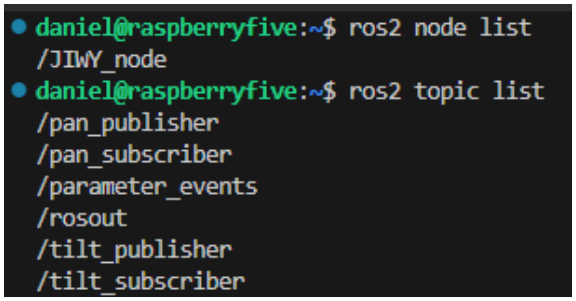


Figure A.15: JIWY_node and topics

JIWY has 4 topics. Two subscriber topics (/pan_subscriber and /tilt_subscriber) that can be used to steer the pan and tilt of JIWY by providing setpoints. Setpoints are given in radians. There are also two publisher topics (/pan_publisher and /tilt_publisher) that publish monitoring data. For bare-metal targets monitoring data is published every 1000 ms and for FreeRTOS targets every 33 ms. This is specified in the target configuration file and can be adjusted when desired. Published data can be seen by running for example "ros2 topic echo /pan_publisher", which will show the current pan position in radians.
The JIWY can also be directly steered from the terminal by running a topic pub command to steer to a certain angle. For example steer the pan to 2 radians:

```
ros2 topic pub /pan_subscriber std_msgs/msg/Float64 "{data:"2"}"
```

RELbot EPG-target deployment

The RELbot target deployment follows the same steps as the JIWY target deployment. First make sure the micro-ROS agent is running:

- First source ROS2 within the terminal:

  ```
  source /opt/ros/$ROS_DISTRO/setup.bash
  ```

- Within the micro_ROS workspace source local_setup.bash to source the micro-ROS agent:

```
source install/local_setup.bash
```

- Start the micro-ROS agent:

```
ros2 run micro_ros_agent micro_ros_agent serial -b 115200 --dev /dev/ttyACM0
```

Then after connecting the Pico 2 USB cable to the single-board computer the "RELbot_node" will be available in the ROS 2.
RELbot has 4 topics. Two subscriber topics (/left_subscriber and /right_subscriber) that can be used to steer the left and right of RELbot wheel by providing setpoints. Position setpoints are given in radians. There are also two publisher topics (/left_publisher and /right_publisher) that publish monitoring data every 33 ms. This is specified in the target configuration file and can be adjusted when desired.

A.3 Demos

A.3.1 Motion profiles

The motion profiles used for the networking tests in this thesis are all present in: https://git.ram.eemcs.utwente.nl/huiskesdv/epg_demos. For both the JIWY and RELbot folders are present for each situation. Each folder contains a ROS 2 package that includes the motion profile.

The repository can be cloned by running:

```
git clone https://git.ram.eemcs.utwente.nl/huiskesdv/epg_demos
```

All ROS 2 packages can be separately compiled and run using the same steps:

- First source ROS 2

```
source /opt/ros/$ROS_DISTRO/setup.bash
```

- Use colcon build for building the package

```
colcon build --packages-select cpp_pubsub
```

- Within the micro_ROS workspace source local_setup.bash to source the micro-ROS agent:

```
source install/setup.bash
```

- Each package has a launch file that starts the package.

```
ros2 launch networking.launch
```

A.3.2 EPG target configurations

In the DEMO folder within the EPG directory there are different subfolders for each test situation. Each subfolder contains target configuration files for the EPG-targets used in the test.
The .tcf files are not present in the EPG menu by default to avoid clouding the GUI target overview, but they can be added to the menu by going to the target directory in the EPG and copying

the file in the right target directory. For example the "rpi_pico_freertos_jiwy_PLANT_1.tcf" file can be used by copying it to the "/target/rpi_pico/freertos folder". After copying, the .tcf file is automatically picked up by the EPG and shown in the GUI.
The target configuration files assume the same setups for EPG-targets as discussed in the report. Therefore when for example PLANT_1 is mentioned in the target configuration file, this represents the EPG-target used for PLANT 1 in the test. The figures of the different network tests can be used as schematic for the setup.
When desired it it possible to manually adjust the .tcf file to use for example 4 Pico 2s. It is advised to look at the structure of a Pico 2 .tcf file and adjust the Pico .tcf file accordingly.
Note that for the teleoperation demos the device that is used as reference device should have its motors disabled, while the encoders stay active. For the JIWY this can be done by turning off the power switch, which leaves the encoders enabled. For the RELbot this can be achieved by disconnecting PWM_VAL, PWM A, and PWM B or by putting the PWM value to 0 in the EPG-generated project.

A.3.3 ROS 2 agents

Most setups require ROS 2 agents running on the single-board computers. Often the USB ports "/dev/ttyACM0" and "/dev/ttyACM1" are used for this. Two micro-ROS agents can be started by running the micro-ROS agent command twice in different terminals or togehter with an &:
Agent 1:

```
ros2 run micro_ros_agent micro_ros_agent serial -b 115200 --dev /dev/ttyACM0
```

Agent 2:

```
ros2 run micro_ros_agent micro_ros_agent serial -b 115200 --dev /dev/ttyACM1
```

A.3.4 Picotool

The Raspberry Pi Pico and Raspberry Pi Pico 2 do not have a reset button. Therefore, when it is desired to reset a Pico this can be done by unplugging the USB cable from the single-board computer. The Picos are however also configured with the functionality to be reset over the USB connection. This can conveniently be managed using the official Raspberry Pi Pico Picotool.
With a single Pico or Pico 2 connected this can conveniently be done by running:

```
sudo picotool reboot -f
```

This does however not work with multiple Picos connected. The Picos then need to be uniquely addressable. This can best be done using the iSerial number of the Pico, since it stays the same over reboots.
When the Pico is connected to the single-board computer the "lsusb -v" command can be used to get information about the Pico as a USB device. In this information the iSerial number is present. Using the iSerial number a Pico can be rebooted by running:

```
sudo picotool reboot -f --ser iSerial_number
```

B Micro-ROS communication

The micro-ROS communication on the Raspberry Pi Pico and Raspberry Pi Pico 2 is via USB. Both devices have an integrated USB 1.1 PHY (Physical layer), handling the electrical transmission of data, and a USB 2.0 controller that manages the communication protocol. While the USB 2.0 protocol is used for the communication, the data transfer speed is limited to USB 1.1 speeds, which are limited to a maximum of 12 Mbps Full-Speed. Similarly, the Nucleo H743ZI also features a USB 2.0 controller with at maximum speeds of 12 Mbps.
When sending or receiving micro-ROS topic data, the data is packaged in multiple layers according to the USB 2.0 (USB Implementers Forum, 2000) and XRCE-DDS (eProsima, 2024; Object Management Group, 2020) standards as shown in Figure B.1. The different elements of this package are:

- **USB 2.0 layer:** The communication between a micro-ROS client running on a microcontroller and a micro-ROS agent running on a single-board computer happens using the USB 2.0 protocol. This protocol forms the outermost layer of the micro-ROS data packet structure.
  When sending a micro-ROS package the USB 2.0 protocol first sends a token packet to announce a USB transaction followed by a data packet with a PID(packet identifier), HDLC frame payload, CRC(Cyclic Redundancy Code check) field, finalised by a handshake packet. The USB 2.0 protocol overhead is 7 bytes.
- **HDLC frame:** The payload of a USB 2.0 data packet is an HDLC frame. XRCE-DDS uses the HDLC stream framing protocol to ensure compatibility with all general stream-oriented transports, including those that do not support packet-based communication. The HDLC frame starts with a flag indicating the start of the frame, followed by a source address of the device sending the message and a remote address of the device receiving the message. The LEN field specifies the length of the payload, which contains the XRCE-DDS message. A CRC is used to ensure message integrity. The HDLC frame overhead is 7 bytes.
- **XRCE-DDS message:** The payload of the HDLC frame is the XRCE-DDS message. This message contains the actual micro-ROS topic data that is transmitted. The message consists of a header and a submessage. The header specifies metadata about the communication, while the submessage carries the actual data. The submessage ID specifies the payload type of the submessage. A write data message is used when data is sent from a micro-ROS client to a micro-ROS agent and a read data message is received when data is sent from a micro-ROS agent to a micro-ROS client. Both data messages carry serialized data that, in the tests performed in this report, consists of an 8-byte double-precision floating-point number. The additional XRCE-DDS message overhead is 12 bytes.

By adding the overhead from USB 2.0, the HDLC frame, and the XRCE-DDS message, the total estimated overhead for sending and receiving a double-precision floating-point topic message is 26 bytes, resulting in a total packet size of 34 bytes. This implies that the transmission time for a single message between a micro-ROS client and a micro-ROS agent is about 23 μs.
There is however additional protocol overhead. The different steps needed for sending a data message are shown in Figure B.2. The figure shows the process of packaging the topic data in the previously discussed layers. The client library first serialises the topic data into an XRCE-DDS message, which is placed in an output stream buffer. The client library then applies the HDLC framing on the XRCE-DDS message, after which the resulting HDLC frame is put in a framing buffer. The framing buffer is eventually written into the device buffer, and USB 2.0 framing is applied to package the data in a USB 2.0 package. Finally, the data is transferred to

the micro-ROS agent via USB. Figure B.3 shows the steps of receiving data, which are similar to the steps of sending data, but in reverse order.

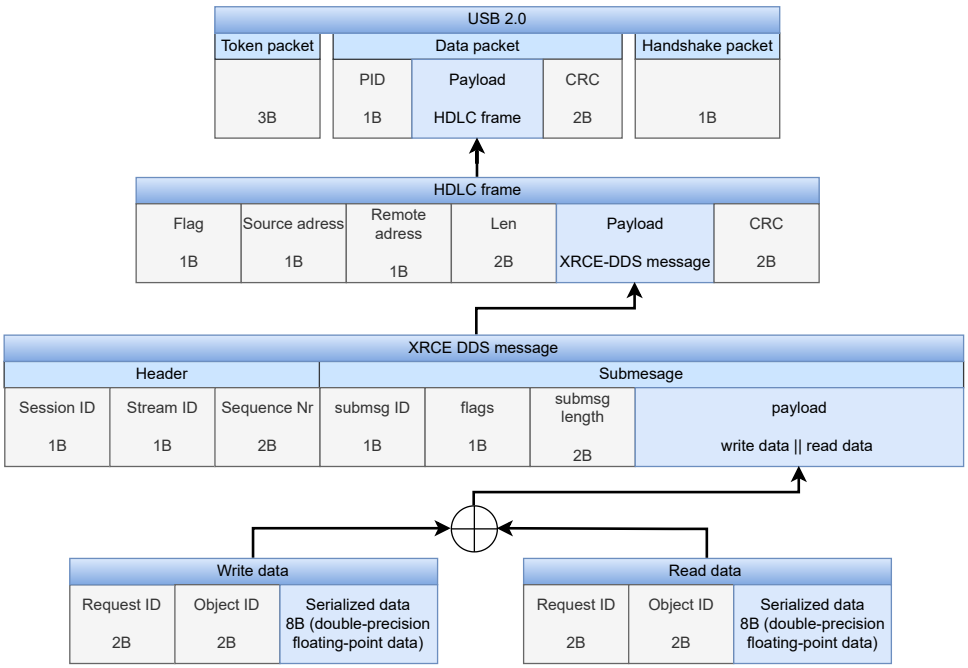


Figure B.1: Micro-ROS data package

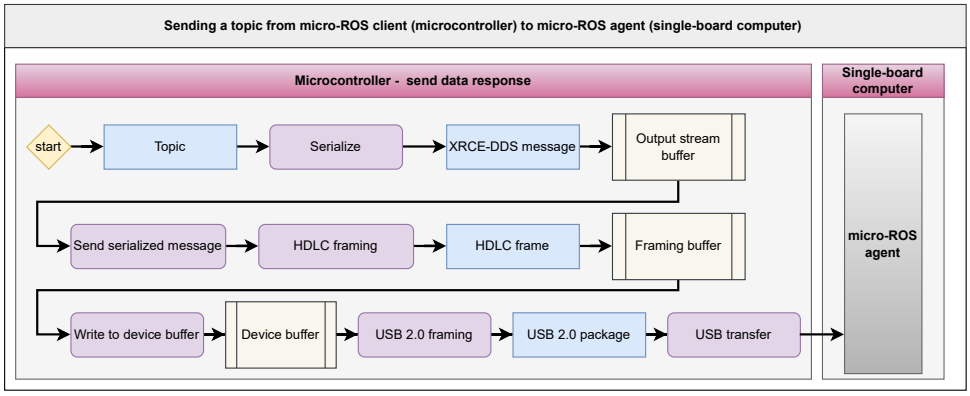


Figure B.2: Sending a micro-ROS topic message

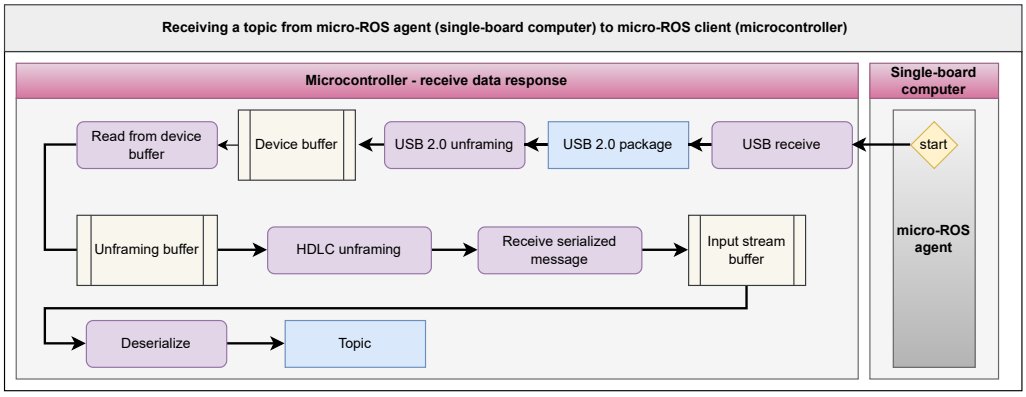


Figure B.3: Receiving a micro-ROS topic message

C Active engagement with open-source developers

Throughout this thesis, developers from various open-source projects were contacted on multiple occasions to resolve issues or contribute to solutions. This appendix provides an overview of those interactions.

Contributing to micro-ROS Zephyr (10-1-2024 closed issue)

During this thesis Zephyr made an update to version 3.5.99, which had adjustments that made micro-ROS incompatible with the latest version of Zephyr. The micro-ROS developers were informed about this issue but reported back that only an old version (v3.1.0) is supported, but are open to suggestions on how to solve the issue. Since it is desired to work with recent software the issue was investigated, and a solution was created in the form of a pull request (PR). The micro-ROS developers accepted the proposed solution and it is now implemented in the micro-ROS Zephyr source code. Therefore, this thesis directly contributed to the development micro-ROS. (https://github.com/micro-ROS/micro_ros_zephyr_module/issues/134)

Resolving Quadrature Encoder issue on Raspberry Pi Pico 2 (18-9-2024 closed issue)

On the Raspberry Pi Pico the PIO unit is used to function as a quadrature encoder. By doing this the hardware on the Raspberry Pi Pico is optimally used, because no processing time of the main processor is spent by handling quadrature encoders. The quadrature encoder implementation did however not function on the Raspberry Pi Pico 2. Since the Raspberry Pi Pico 2 was just released nobody had yet experienced this issue and no information could be found about it. Not solving this issue would make the Pico 2 unusable as an EPG-target, because the alternative solution of handling quadrature encoder interrupts would disturb the processor cores too much. After thoroughly investigating the issue the Raspberry Pi development community was contacted to resolve the issue. Eventually the issue was resolved, which made it possible to use the Raspberry Pi Pico 2 as EPG-target in this project. (https://github.com/raspberrypi/pico-examples/issues/550)

Using create_firmware_ws.sh to flash Nucleo F767ZI (28-3-2023 open issue)

In an early stage of this thesis it was attempted to use the create_firmware_ws.sh package to flash a Nucleo F767ZI microcontroller, which was used for early tests, but offers no technical advantages over the Nucleo H743ZI. This did however not work and the micro-ROS developers recommended an alternative approach. (https://github.com/micro-ROS/micro_ros_setup/issues/623)

Testing EPG-Target with FreeRTOS on Nucleo H743ZI (8-9-2023 open issue)

It was desired to test an EPG-target using FreeRTOS on the Nucleo H743ZI. The FreeRTOS version was however non-functional. Therefore, this issue was reported to the micro-ROS developers to resolve it. Multiple people have experienced the same problem and have pinpointed in the direction of a solution, but the issue does still not appear to be fully resolved. (https://github.com/micro-ROS/micro_ros_stm32cubemx_utils/issues/119)

Attempt to Enable SMP on Raspberry Pi Pico in Zephyr (10-4-2024 open issue)

It was desired to use both cores of the Raspberry Pi Pico in the Zephyr implementation using SMP. This was however not supported. On 28-6-2023 another developer already opened an issue to implement this support, but there was no solution and the discussion became

unactive. Therefore, the issue was revamped by involving several Zephyr developers. Eventually the outcome was that SMP is not yet implemented, because the processor of the Pico lacks a needed monitor implementation. Therefore, this issue was not resolved. (https://github.com/zephyrproject-rtos/zephyr/issues/59826)

Long micro-ROS publishing times with Zephyr on Nucleo H743ZI (19-6-2024 open issue)

While using Zephyr with the Nucleo H743ZI, long micro-ROS publishing times were experienced. The issue is still open. https://github.com/micro-ROS/micro_ros_zephyr_module/issues/140

Rebuilding micro-ROS static library for Raspberry Pi Pico (2-10-2024 open issue)

It was desired to rebuild the pre-build micro-ROS static library that is used for the Raspberry Pi Pico. This would enable the functionality of some custom functions that are not enabled by default. An example is multithreading support. Micro-ROS has a function to enable multi-threading support. In an initial version of the bare-metal Pico EPG-target it was planned to manage the timing of the soft real-time and firm real-time task with the micro-ROS timer implementation and giving priority to the firm real-time task. This would however require micro-ROS to operate on two separate cores simultaneously, which by default corrupts memory and crashes the system. The static library builder did however not work. Attempts on Linux, Windows, and Docker all crashed. Therefore, the micro-ROS developers were contacted to report the issue an find a solution. (https://github.com/micro-ROS/micro_ros_raspberrypi_pico_sdk/issues/1274)
The issue has still not been fully fixed. Eventually for this thesis it was chosen to only use the micro-ROS timer for soft real-time communication and the rpi pico interrupt timer for the firm real-time control loop, running micro-ROS only on a single core. This is also the preferred solution after all, as the interrupt timer of the RPI pico has strict interrupt based timing with low overhead compared to the micro-ROS timer.

D Literature study

This literature study appendix consists of literature on software development methodologies, software tools and software architectures relevant for this thesis. The background information is ordered by collecting literature that forms an answer on the following questions:

- What are the key aspects of a component-based design methodology?
- What is the state-of-the-art in robotic software architectures?
- What solutions are available for real-time robotic software?
- How can networking be applied within a robotic software architecture?

D.1 Key aspects of a component-based design methodology

D.1.1 Component-based software

Component-based software methodologies facilitate the re-use of code and aim to provide convenient system assembly by combining several components.

According to Szyperski (2002) the definition of a software component is as follows: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.". In other words this implies that a component functions as a black box that incorporates all the functionality within the box. The box can have multiple inputs and outputs that receive a pre-defined data format. A schematic that illustrates this description is shown in Figure D.1. The components that result from a component-based methodology can therefore be combined just like building blocks to quickly build a software architecture. Since the components can operate independently, they can also be conveniently re-used in different software structures.

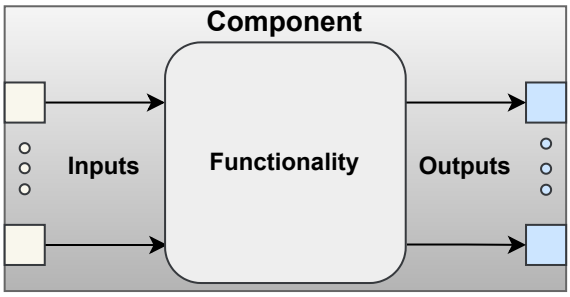


Figure D.1: Schematic of a component - This figure shows a general high-level general description of a component that can have multiple inputs and outputs. Within the component there is functionality that can operate stand-alone.

Brugali and Scandurra (2009) state that "a robot–software architecture describes the decomposition of the robot control system into a collection of software components, the encapsulation of functionality and control activities into components, and the flow of data and control information among components.". The advantage of describing a robot-software architecture in this component-based way is that there is a convenient way in which the functional and non-functional requirements of the robotic system can be realized by mapping the requirements on parts of the component. The functional requirements of the robotic system can for example be implemented within the functionality of the component. Non-functional requirements, such as how communication within the system should take place can be implemented by specifying the input and output data formats of the component.

D.2 State-of-the-art in robotic software architectures

D.2.1 Xenomai/EVL

The purpose of Xenomai is to add real-time functionalities to non-real-time operating systems, such as Linux. Linux itself does not have a real-time capable kernel. As a consequence real-time performance of applications is not guaranteed. Xenomai/EVL is a companion core that can be used in combination with a Linux kernel forming a dual kernel architecture. (EVL Project, 2023). It becomes more common to use devices, such as a Raspberry Pi 4B, within robotic projects. Often these devices make use of a Linux based operating system. Xenomai makes it therefore possible to also execute real-time tasks on those devices. It is possible to communicate with the real-time kernel from the Linux kernel by using the Dovetail interface. Figure D.3 shows how the different layers stack on each other.

Communication between the EVL core and the general kernel is possible by means of a cross-buffer as shown in Figure D.3. This buffer connects the in-band and out-of-band contexts with each other and makes it possible to use the general read() and write() functions on the inbound side and similar oob_write() and oob_read() functions on the outbound side.

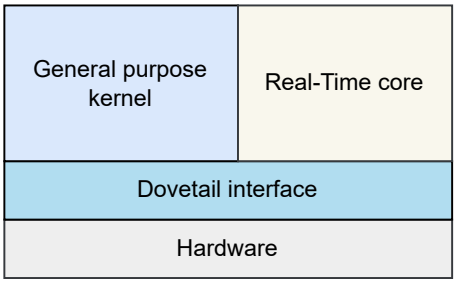


Figure D.2: Xenomai Linux kernel

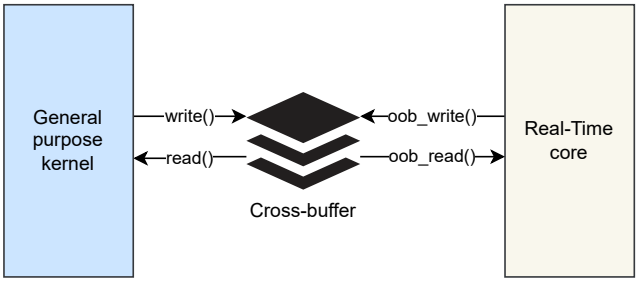


Figure D.3: Cross-buffer

D.2.2 ROS 2

The Robot Operating System (ROS) is a set of software libraries and tools that can be used to build robot applications. The latest version of ROS is ROS 2, which has improvements in Communication Middleware, real-time support and scalability compared to its predecessor.

Some of the key ROS 2 design principles are distribution, abstraction, asynchrony, and modularity (Macenski et al., 2022).

ROS is not an actual operating system, but an abstraction layer that runs on top of a host operating system and has a set of tools and services for building robotic applications.

ROS 2 is based on the Data Distribution Service (DDS). This is an open standard for communication and enables security, embedded and realtime support, multi-robot communication, and operations in non-ideal networking environments (Macenski et al., 2022).

ROS 2 has different communication methods that can be used. These communication patterns are shown in Figure D.4. The communication patterns are topics, services and actions. These communication patterns can be used within the context of a node. The node is an entity that can be created within ROS to perform certain tasks and communicate with other nodes.

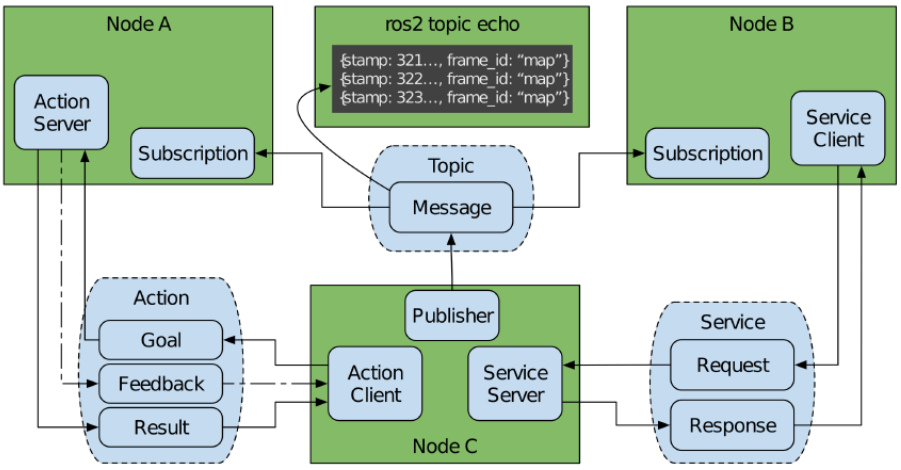


Figure D.4: ROS 2 node interfaces: topics, services, and actions.(Macenski et al., 2022)

An overview of the client library API stack is shown in Figure D.5. The client library forms an abstraction layer to make use of the different ROS 2 functionality. Therefore, to have access to the communication API's the client library can be used. There are different ros client libraries (rcl) for different operating systems that all depend on a rcl that is written in C. Also there is a middleware abstraction layer rmw (ROS Middleware) that provides the communication interfaces.

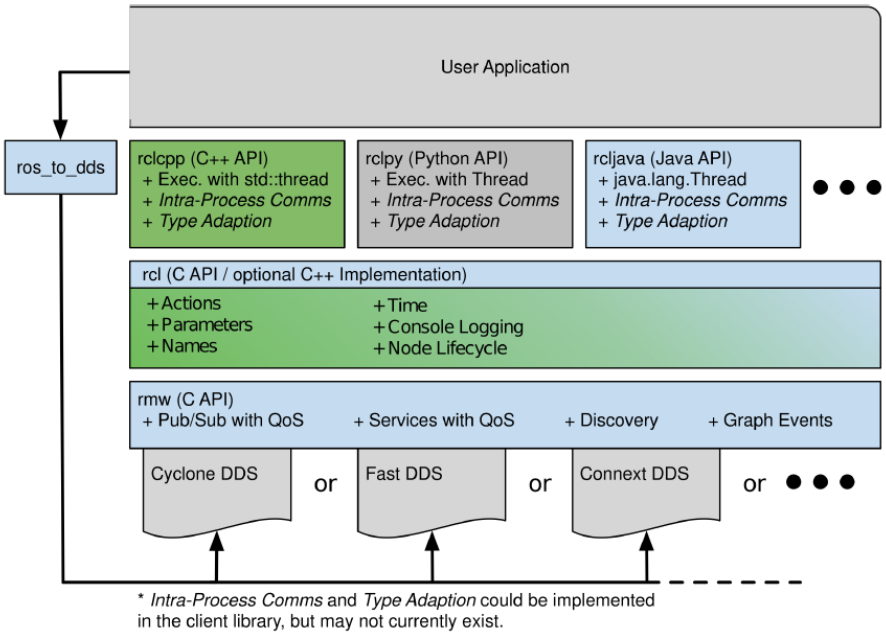


Figure D.5: ROS 2 Client Library API Stack. (Macenski et al., 2022)

Although ROS 2 claims to have real-time support, achieving real-time performance requires the use of additional real-time middleware. While ROS 2 provides the necessary infrastructure and features for real-time communication, the integration of dedicated real-time middleware is essential to fully achieve deterministic and predictable behaviour in real-time applications.

D.2.3 micro-ROS

Micro-ROS is a real-time framework designed to bring functionalities of the Robot Operating System (ROS) to microcontrollers (Belsare et al., 2023). ROS is often used on more powerful hardware such as the Raspberry Pi. By using micro-ROS it is therefore possible to extend the existing ROS framework to microcontrollers.

The most important aspects of micro-ROS are :

- Microcontroller-optimized client API supporting all major ROS concepts
- Extremely resource-constrained but flexible middleware
- Seamless integration with ROS 2
- Multi-RTOS support

Micro-ROS makes use of DDS-XRCE protocol, which stands for DDS For Extremely Resource Constrained Environments. This resource constraint variant of the DDS protocol can directly be used within the ROS 2 architecture and seamlessly integrates with DDS.

Micro-ROS can be used in combination with a real-time Operating System (RTOS), such a FreeRTOS, or bare-metal. The advantage of using micro-ROS in combination with a RTOS is that different scheduling algorithms can be used to obtain real-time performance.

Micro-ROS follows the same node-based architecture of ROS 2 that makes use of message passing between the nodes by subscribing and publishing on topics.

The architecture overview of micro-ROS is shown in Figure D.6. This figure shows that micro-ROS runs on a microcontroller on top of a RTOS. The Micro XRCE-DDS client integrated in micro-ROS can connect to a ROS 2 agent that runs on another device such as a Raspberry Pi. By doing this, a connection can be established over which messages can be sent between nodes.

The micro-ROS agent connects micro-ROS nodes (i.e. components) on MCUs seamlessly with standard ROS 2 systems. This allows accessing micro-ROS nodes with the known ROS 2 tools and APIs just as normal ROS nodes.

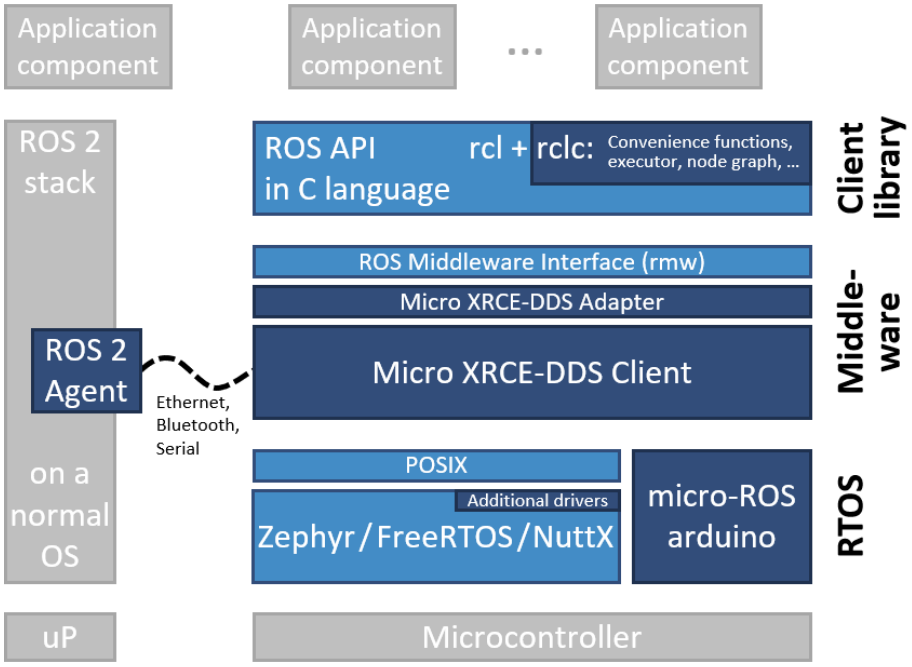


Figure D.6: Architecture of the micro-ROS stack (Belsare et al., 2023)

D.2.4 20-sim

20-sim is a modelling and simulation software package for mechatronic systems (Controllab, 2025). It can be used to create graphic models. The behaviour of the dynamic systems represented by these models can be analyzed, and control systems can be designed. Using the C-Code generation function in 20-sim models it is possible to export 20-sim models to C or C++ code such that it can be run on microcontrollers or microprocessor devices.

D.2.5 Development of an Onboard Robotic Platform for Embedded Programming Education

In the research of Lee and Yi (2021) a robotic platform is proposed that can be used to teach embedded programming. This platform makes use of STM32CubeIDE, OpenCV, and MATLAB for the development environment. A STM32F micro-controller in combination with a Raspberry Pi was used as the cyber part of the mechatronic system.
In Figure D.7 the communication architecture of the robotic platform is shown. The setup shows a connection between the microcontroller and the Raspberry Pi. The robotic platform makes use of a vision application, which is executed on the Raspberry Pi. Subsequently, information from the vision system is communicated to the microcontroller using serial communication. The microcontroller is in control of the physical system and controls the different motors and reads the sensors.
The platform offers both soft real-time programming for basic level programming courses and firm real-time programming for advanced level courses.

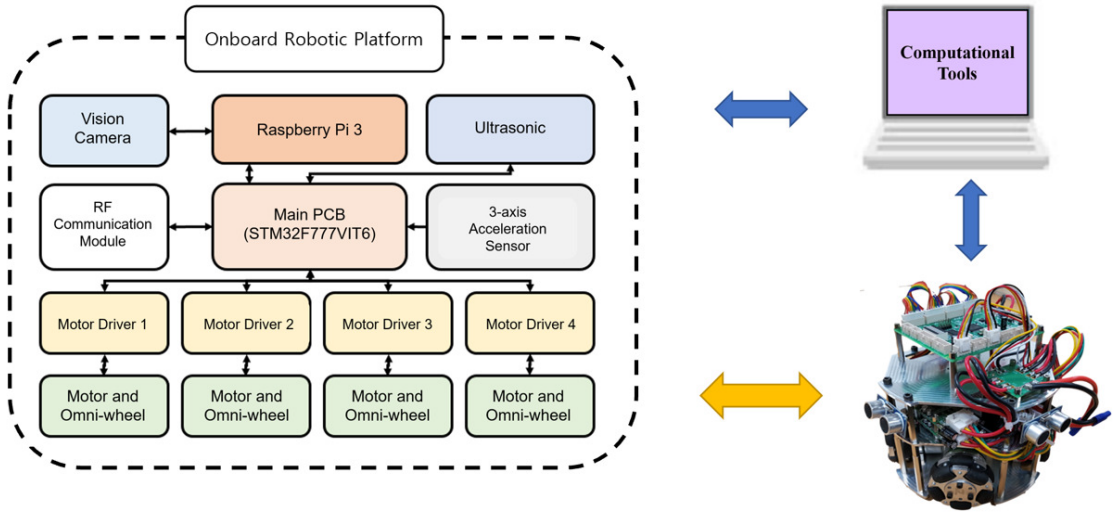


Figure D.7: Entire communication architecture of the developed platform. (Lee and Yi, 2021)

D.2.6 A Model Based Engineering Tool for ROS Component Compositioning, Configuration and Generation of Deployment Information

The ReApp project is a workbench created by Wenger et al. (2016). The workbench is based on ROS. The central part of this framework is the skill and solution modelling tool. This tool can be used for model-based design of robot applications composed of reusable components. The entire workbench consists of five parts:

- **Component Modelling Tool** (CMT)Software- and Hardware-Access-Components are created using the CMT
- **Skill and Solution Modelling Tool** (SSMT) The SSMT is used to assemble applications. These applications are obtained from a repository where they are stored. The SSMT

forms an abstraction layer for ROS, such that familiarity with ROS is not needed. It also has a graphical tool that can be used for the composition and configuration ROS nodes.
- **Integration Platform & Development Environment** (IPDE) The IPDE contains the executables needed to run the ReApp App
- **Cloud-based semantic repository** The cloud store is used to share work created by the SSMT.

Figure D.8 shows the interconnection of the different components that together form the framework.

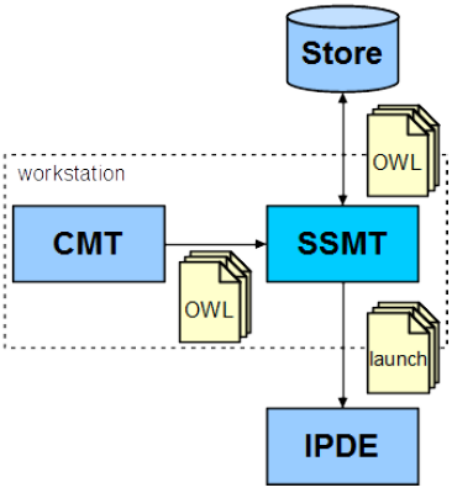


Figure D.8: The Skill and Solution Modelling Tool (SSMT) and its connections to the Component Modelling Tool (CMT), the ReApp Store and the Integration Platform & Development Environment (IPDE) (Wenger et al., 2016)

ReApp is an interesting framework. The framework is designed specifically for ROS 1. ReApp makes use of a graphical editor for the SSMT tool. Such a graphical editor can be convenient to connect different components. The framework currently creates locations were code can be implemented at specific locations, but is not able to create code itself. Therefore, an additional tool such as 20-sim must be used to create models.

The example discussed in the framework was directed towards a robot arm that can be used by car manufacturer BMW, one of the sponsors of this project. The project does not seem to be active.

D.2.7 Microcontroller for 20-sim-generated C code

The work of Visser (2020) investigates the Arduino Due as a bare-metal platform for C-Code generated with 20-sim. The Arduino Due board makes use of an ARM-M3 microcontroller.
In this work it was found that the microcontroller could run a PID controller making use of a 10th-order low-pass filter at a loop frequency of 100 Hz and a maximum combined latency of 0.6ms. The limiting factor for the sampling frequency is that there is no double-precision floating-point unit present on the microcontroller.
Because of that, the floating-point calculations have to be performed in software, which reduces performance. The software framework that is used for this implementation is shown in Figure D.9.
This research shows interesting results for implementing 20-sim models on a relatively low-power microcontroller. Making use of more powerful hardware and better optimized software protocols might significantly improve the results.

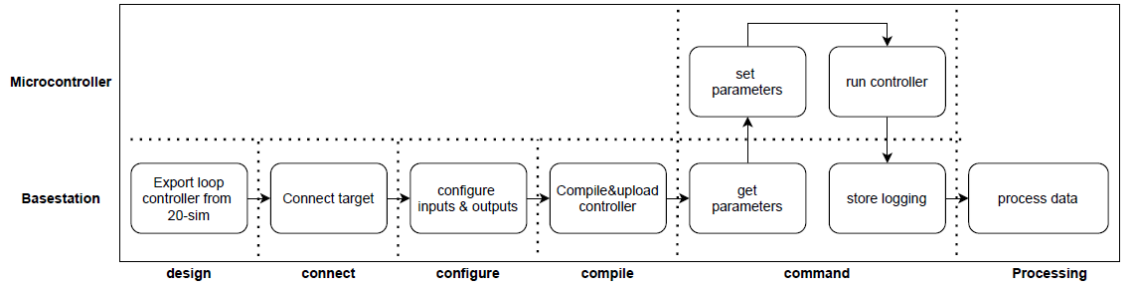


Figure D.9: Overview of functions the software framework has to perform. (Visser, 2020)

D.3 Real-time robotic software

D.3.1 Robot software framework using Xenomai and ROS 2

The research conducted by Meijer (2021) considers creating a real-time robot software framework that makes use of Xenomai and ROS 2. The goal of the thesis was to develop a framework that integrates real-time tasks and ROS 2 to control physical robots. Therefore, this thesis had similar objectives as discussed in this project plan.

When deciding on what hardware to use a decision had to be made between a dual device and single device. It was argued that the use of a microcontroller such as the ESP32 or an Arduino would only be possible in combination with a companion device, because of the limited processing power. From a weighted decision table it was eventually concluded that making use of the Raspberry Pi would be the best option. The consequence is however that the choice of software is limited when real-time operation is desired. There has been chosen to make use of Xenomai. In this research Xenomai 3 was used, which makes use of the XDDP protocol for cross-kernel connection for communication between a non-real-time and a real-time task.

The layout that follows from this combination of choices is shown in Figure D.10. This figure shows that there is a separation between real-time and non-real-time tasks by the Linux kernel and the regular Xenomai kernel. In practice this does not only have to be a separation in software, but it is also possible to make this a separation in hardware, which is shown by the example layout in Figure D.11. In this example layout the real-time tasks run on 2 separate cores of the Raspberry Pi quad-core processing unit. The non-real-time tasks run on the other 2 cores.

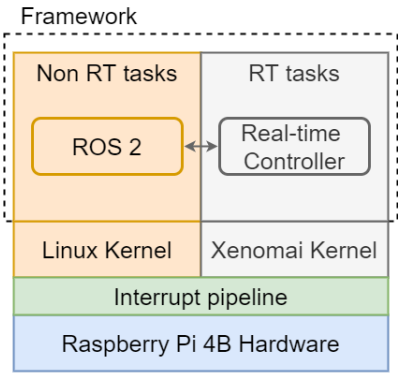


Figure D.10: Hardware and software layout. (Meijer, 2021)

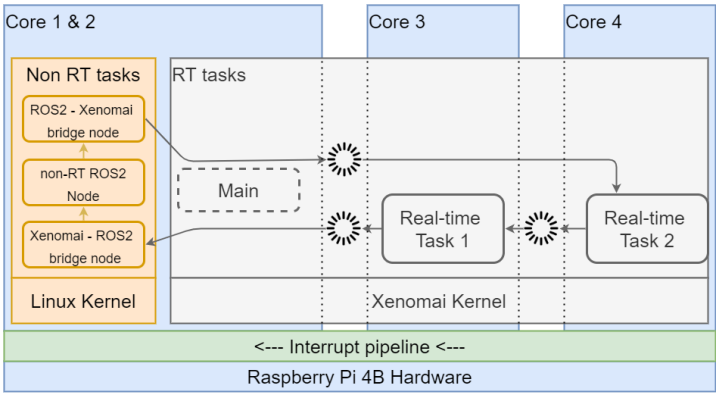


Figure D.11: Example setup of the complete framework architecture. (Meijer, 2021)

In this research the choice has been made to implement the framework functionality in a set of C++ classes.

The functionality of the framework was at the time of this research on a base level. It was possible to do some in software experiments, but the framework is not mature enough yet to be used with a real plant. Also, an additional device in the form of an FPGA is used to be able to communicate from the Raspberry Pi with a setup making it effectively a dual device setup after all. The FPGA produces PWM signals and reads encoders. Communication between the FPGA and Raspberry Pi takes place using the SPI protocol.

The framework has been applied in another thesis by In 't Veld (2023) to control a production cell. In this setup the six different production cell units are mapped to three Raspberry Pi 4Bs. Each Raspberry Pi controls two units. Communication between the boards takes place using ROS 2.

D.3.2 Embedded software architecture for a mobile education robot with Real-Time control

The work of Vinkenvleugel (2022) investigates the use of a Raspberry Pi 4 and an icoBoard for a mobile education robot. This work follows partly from the work of Figure D.11 and also makes use of Xenomai. In total there were three bachelor student projects involved in working on elements of the mobile education robot, but this projects main focus is the embedded software architecture, which is relevant for this thesis.

The objective of this thesis is "to design an embedded software architecture for an education robot that is expandable in both software and hardware features, provides open access to the complete software stack and utilizes a Raspberry Pi 4 and an icoBoard."(Vinkenvleugel, 2022).

There has been chosen to use Xenomai, because it has a lower latency compared to alternatives such as PREEMPT-RT and there is already an SPI driver that can be used with Xenomai to communicate with the icoBoard. The indicated disadvantages for Xenomai are however that Xenomai does not have good support for different hardware platforms and a custom kernel build is required for the Raspberry Pi 4. Also, a concern with using Xenomai was that drivers needed to be rewritten to make use of the real-time capabilities of the kernel. Beside this Xenomai, code has to be adapted to make use of POSIX like functions that are created by Xenomai to schedule tasks in a high-priority execution stage. This was not seen as a disadvantage, since POSIX functions are well documented, so their use is known. It does however imply that code has to be adjusted using these functions.

A potential bottleneck for performance within this work seemed to be the communication between the Raspberry Pi and the icoBoard making use of the SPI protocol. Vinkenvleugel (2022) and Raoudi (2023) conducted an investigation that resulted in valuable insights into the factors that affect communication latency and how this latency can be optimized.

D.3.3 20-sim Template for Raspberry Pi 3

It is possible to use 20-sim-4C for implementation of 20-sim code on a target device. An implementation of 20-sim-4C targeted for the Raspberry Pi 3 has been developed by Dokter (2016). It was desired to make use of Xenomai in order to have real-time guarantees, but implementing this was not successful. Xenomai was only supported by distinct Linux kernels. Therefore, a match had to be found between a Linux kernel that is both supported by Xenomai and the Raspberry Pi. The problem was however that for the version were there was a match no Linux headers were available for the kernel version. Beside that Xenomai missed functions that were needed to run models on Xenomai. Because of these complications it has been concluded that using Xenomai is not feasible.

Experiments showed that it is possible to keep up with a 1 kHz sample rate for simple control applications developed in 20-sim. The Raspberry Pi has a powerful processor and therefore also performs good without having real-time guarantees. Therefore, the platform can be used for a

lot of control applications that do not have Hard real-time guarantees. There were still some problems with the 20-sim-4C implementation of the Raspberry Pi such as a communication problem with logging. This work does however clearly show the power of the Raspberry Pi for running control applications.

D.3.4 OROCOS

The work of Bruyninckx (2002) describes the design of the OROCOS framework. OROCOS (Open Robot Control Software) is an open-source software framework designed for real-time control of robotic systems. It provides a modular and flexible platform for developing control algorithms and managing complex robotic systems. The OROCOS framework has grown over the years and now consists of a set of portable C++ libraries for advanced machine and robot control. The Ocoros real-time Toolkit (RTT) and the Orocos Component Library (OCL)are the most important parts of this framework.

The idea behind the framework is that it can be used as an application generating code base from which robotic systems can be constructed. The code base consists of components that can be used for this construction.

The RTT is created for the implementation of (realtime and non-realtime) control systems (Soetens, 2024). The general principle behind the RTT is shown in Figure D.12. From the real-time toolkit different components can be build. The control components that follow from this can be used to build control applications that can run on common-of-the-shelf hardware.

The OROCOS components are built upon the real-time Toolkit Application Stack as shown in Figure D.13. The different elements of the RTT in combination with native OS libraries can be used to build components.

The OCL contains the necessary components to start an application and interact with it at run-time.

Initially the OROCOS framework only made use of the CORBA standard for communication functionality. However, since the popularity of the ROS framework increased also functionality for ROS has been introduces into the OROCOS framework.

To make use of the real-time capabilities of the OROCOS framework it is important that the underlying operating system makes use of a real-time extension such as Xenomai to guarantee Real-Time performance.

The idea behind the OROCOS framework is interesting, but the framework has multiple dependencies on other frameworks. Therefore, OROCOS is not up-to-date with the latest developments. For example the latest version of Xenomai that is supported by the OROCOS framework is Xenomai 3.

This implies that if it is desired to use OROCOS with Xenomai the implementation would be dependent on an old version of Xenomai that is no longer supported. The OROCOS framework is still in use by different parties, but the development of the framework does not seem to be active at the moment.

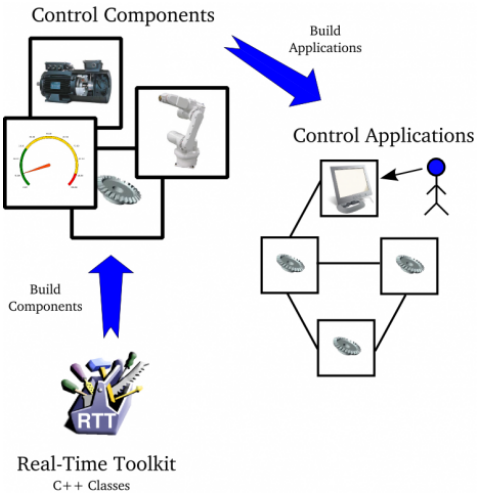


Figure D.12: Real Time Toolkit (Soetens, 2024)

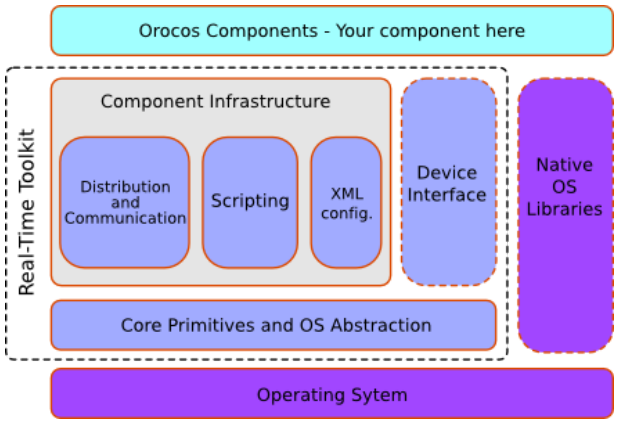


Figure D.13: real-time toolkit application stack. (Soetens, 2024)

D.4 Networking within a robotic software architectures

D.4.1 Zoro: A robotic middleware combining high performance and high reliability

Zoro is a robotic middleware proposed by Liu et al. (2022). The aim of Zoro is to provide a data transmission method that has both high performance and high reliability. The proposed method brings an improvement of up to 41% in communication latency compared to similar middleware such as ROS 2. A combination of shared memory for performance improvement and socket-based communication for improved reliability is used for the communication mechanism. Robotic middleware often makes use of socket communication, because it is reliable. However, there is a linear relation between an increase in message size and communication latency. Shared memory usage can therefore be used to improve the performance, since by using shared memory the data can be accessed from the same location by multiple users. Making use of shared memory can however often lead to safety problems. Some examples of these problems are:

- **Crash Safety Issue** - A crash of a process that holds a read-write lock can have as a result that the shared memory is destroyed unexpectedly. This can result in other processes being blocked that make use of the same shared memory.
- **Data Reliability Issue** - It is not ensured that all processes that make use of the shared data can read the data before it is replaced.
- **Memory Protection Issue** - Shared memory can be directly accessed and modified by multiple processes, which can lead to invalid memory access and data corruption.

Service discovery is important for high performance and high reliability. ROS 2 makes use of a decentralized method. This implies that all nodes receive the messages, which is good for safety. When there is a crash somewhere in the system this does not directly result in a total system crash. The disadvantage is however that at the moment that the number of nodes that send messages increases also the number of resources that is used within the system increases rapidly.

To solve the previously mentioned problems socket based communication control algorithm is proposed. The socket is used to sent a control message with for example the data location, while shared memory is used to transport the real data. In this method it is assumed that the control message has a smaller data size then the real data. When this is not the case the socket based communication can still be used to transport the real data.

A weak centralized service discovery mechanism is proposed. This is a hybrid between centralized and decentralized approaches. A centralized node stores information of the nodes, but the node lifecycle is managed by the nodes themselves.

Figure D.14 shows an overview of how Zoro can be used. In this figure there is a publisher and a subscriber. Both are connected to a network where a notifier of the publisher sends the location of the data to the notifier of a subscriber. The shared memory is used by the publisher to store the data and by the subscriber to read the data.

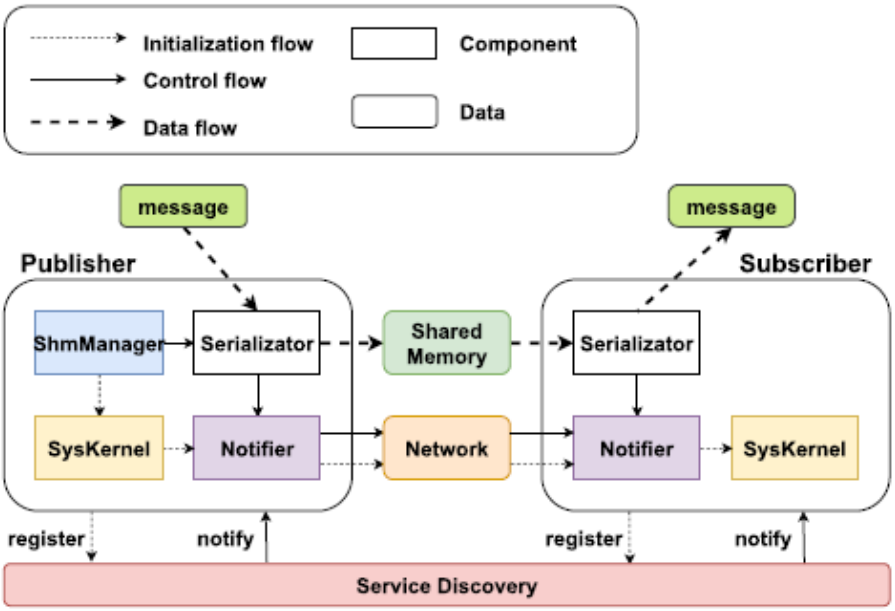


Figure D.14: The overview of Zoro. (Liu et al., 2022)

The work of Zoro tries to create the best of both worlds. It does this by combining shared memory with sockets, but also by combining aspects of the centralized and decentralized approaches. It is interesting that the methods are decoupled just enough to get rid of disadvantages. An example is that in cases that a small amount of data is send there is chosen for a socket approach instead of using the shared memory. The work is extensive, and the developers have thoroughly thought about different problems that might hinder performance and reliability. There is a trade-off between hardware and performance. More hardware (in this case the shared memory) is used in order to achieve the lower latency's and less processor usage. Since shared memory is used this approach might be less effective in multi-machine use cases and distributed systems since there is no directly shared memory between those distributed systems.

D.4.2 Distributed Robotic Systems

The increasing popularity of cloud services and IoT has also had its influence on robotics. The work of Zhang et al. (2022) shows a comprehensive overview of the latest developments within the field of distributed robotic systems. These technologies enable for direct communication with each other using a communication protocol such as DDS, but also for containerized robotic applications to run on the edge or in the cloud. For these applications to be relevant it is important to have a tight integration with other popular robotic frameworks such as ROS. Common way in which distribution can take place are as follows:

- Using the **DDS protocol** middleware it is possible for different devices that run ROS to communicate with each other.

- **Containers** create an environment where applications, such as ROS, can run with all required dependencies. Containers make use of a lightweight form of virtualization. Containers can be build with a tool like Docker and orchestrated with tools such as Kubernetes.
- **Computational offloading** can be used to perform remote teleoperation of robots, run a part of different computations in the cloud or do both of these tasks. Important for offloading orchestration are real-time latency and bandwidth.
  Edge computing solutions are gaining popularity as a means to reduce communication latency, in addition to the widespread popularity of cloud computing.
  By using computational offloading it is for example possible to create ROS 2 nodes on the cloud or the edge and perform computational extensive tasks in those nodes. By doing this it is still possible to make use of the ROS ecosystem for communication, but system resources can be extended as desired.

Vulcanexus is an example of a tool set that can be used as an extension to the ROS 2 environment to add additional convenient features (Vulcanexus, 2023). Examples of these features are

- ROS 2 monitor : graphical desktop application to monitor ROS 2 communications
- ROS 2 Router: application that enables the connection of distributed ROS 2 environments.
- Webots: Robot simulator

A docker image of Vulcanexus is available that contains the Vulcanexus tool with all the dependencies needed to run Vulcanexus.

It is interesting to note that also a docker image of ROS 2 is available. Therefore, it is possible to conveniently run ROS 2 in a distributed way. When a device has a docker installation it is possible to run the application on that device and further extend the ROS 2 environment over a networked system.

D.4.3 Designing a communication component

The work of Kempenaar (2014) is involved with designing a communication component that can be used for connecting different computing platforms. Algorithms that run on the different platforms should be able to exchange data using this communication component. The communication is realized by means of a communication server running on a computing platform. Using Inter Process Communication the server can interact with an application. Communication between Communication Servers takes place over Ethernet. A schematic overview of this system is shown in Figure D.15. The communication component was implemented in LUNA (Bezemer, Wilterdink and Broenink, 2011) and XML was used for configuration information.

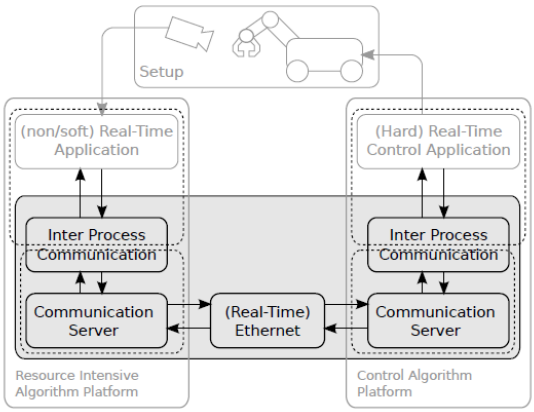


Figure D.15: Abstract system overview, for the Communication Component. (Kempenaar, 2014)

The idea behind the communication component is interesting. It also has some resemblance with the principle behind micro-ROS to create communication between different devices by means of a server that handles communication. In the report it was however also mentioned that it is difficult to use the Communication Component together with other frameworks and to get a tool-chain up and running. It is suggested that interfaces should be created within the OROCOS framework or for 20-sim. By doing this the component should become more flexible to be used within other frameworks. Since the implementation of the Communication Component is custom work it does not have support of a large developer community, which makes integration into different components and development more difficult. It is the sole responsibility of the maintainers.

E Ping-pong measurements

E.1 Data distribution round-trip times

Figure E.1 shows boxplots of the different targets. In this figure, only targets with outliers lower than 25ms have been plotted for readability. Figure E.2 shows a boxplot of all targets.

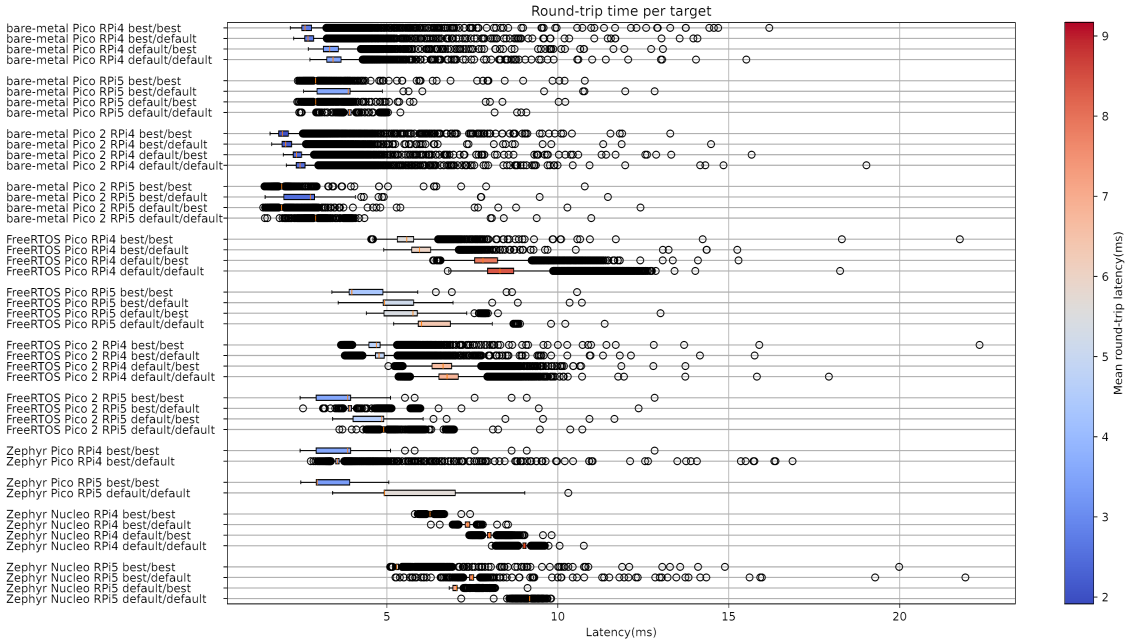


Figure E.1: Latency boxplots per target

Comparing the different boxplots in Figure E.1 shows that most targets have outliers of at least 10 ms. The outliers are concentrated on the right side of the boxplot, indicating a right skew in the data distribution. The bare-metal implementations using the Raspberry Pi Pico 5 show comparatively the least amount of outliers.

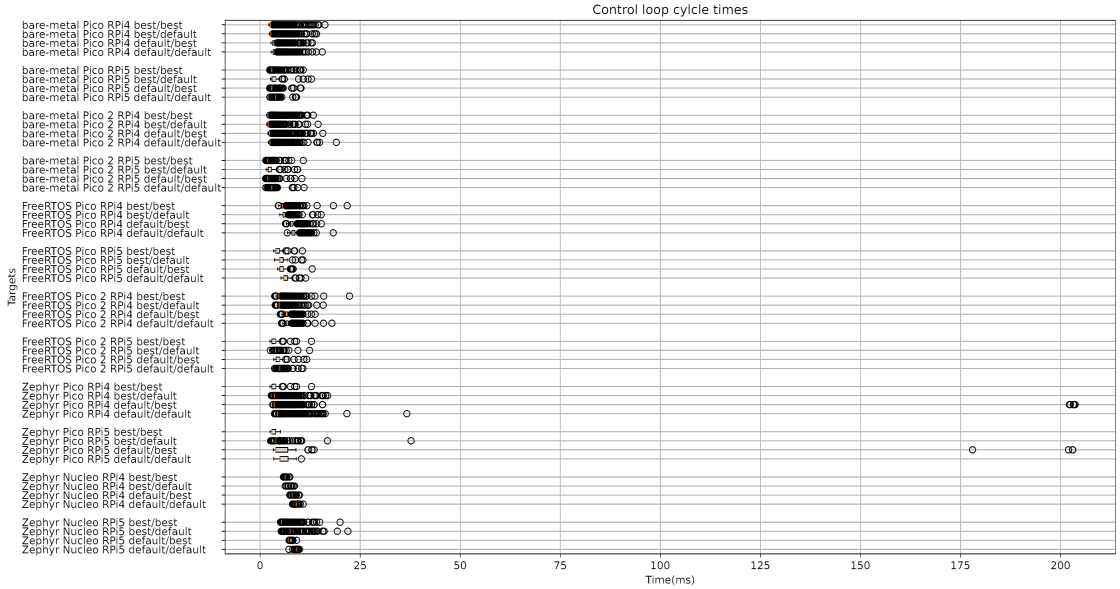


Figure E.2: Latency boxplots complete

Table E.1 shows a table with statistics of the round-trip communication within the ping-pong setup. The table presents the mean, standard deviation, percentual deviation, minimum, maximum, and median. The percentual deviation represents the percentage of difference in the mean round-trip time of all targets compared to the target with the lowest round-trip time. In this case, the baseline is the bare-metal Pico 2 with an RPi5 using a best-effort publisher and subscriber.

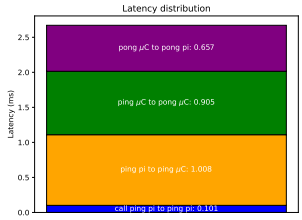
Table E.1: Round-trip time statistics

Target	Mean (ms)	Standard Deviation (ms)	Percentual Deviation (%)	Minimum(ms)	Maximum (ms)	Median (ms)
bare-metal Pico RPi4 best/best	2.67	0.26	39.48	2.17	16.19	2.60
bare-metal Pico RPi4 best/default	2.75	0.25	43.56	2.27	14.08	2.69
bare-metal Pico RPi4 default/best	3.37	0.32	75.86	2.70	13.08	3.32
bare-metal Pico RPi4 default/default	3.46	0.30	80.52	2.75	15.51	3.42
bare-metal Pico RPi5 best/best	2.91	0.07	52.04	2.39	10.79	2.91
bare-metal Pico RPi5 best/default	3.57	0.47	86.40	2.56	12.83	3.87
bare-metal Pico RPi5 default/best	2.91	0.09	52.11	2.37	10.22	2.91
bare-metal Pico RPi5 default/default	3.91	0.07	104.09	2.43	9.08	3.90
bare-metal Pico 2 RPi4 best/best	2.02	0.30	5.49	1.58	13.29	1.94
bare-metal Pico 2 RPi4 best/default	2.11	0.28	9.99	1.63	14.48	2.04
bare-metal Pico 2 RPi4 default/best	2.41	0.25	25.71	1.97	15.67	2.35
bare-metal Pico 2 RPi4 default/default	2.51	0.27	30.92	2.06	19.03	2.46
bare-metal Pico 2 RPi5 best/best	1.92	0.05	0.00	1.42	10.79	1.92
bare-metal Pico 2 RPi5 best/default	2.47	0.45	28.83	1.44	11.46	2.76
bare-metal Pico 2 RPi5 default/best	1.92	0.06	0.10	1.38	12.42	1.92
bare-metal Pico 2 RPi5 default/default	2.91	0.07	52.15	1.43	10.98	2.92
FreeRTOS Pico RPi4 best/best	5.57	0.38	190.76	4.54	21.76	5.58
FreeRTOS Pico RPi4 best/default	6.02	0.40	214.26	4.91	15.25	5.95
FreeRTOS Pico RPi4 default/best	7.92	0.56	313.53	6.35	15.29	7.81
FreeRTOS Pico RPi4 default/default	8.38	0.63	337.66	6.78	18.26	8.31
FreeRTOS Pico RPi5 best/best	4.30	0.48	124.73	3.39	10.56	3.97
FreeRTOS Pico RPi5 best/default	5.16	0.43	169.59	3.58	10.70	4.93
FreeRTOS Pico RPi5 default/best	5.42	0.50	182.86	4.40	13.00	5.76
FreeRTOS Pico RPi5 default/default	6.38	0.51	233.16	5.19	11.37	6.01
FreeRTOS Pico 2 RPi4 best/best	4.66	0.30	143.14	3.65	22.34	4.69
FreeRTOS Pico 2 RPi4 best/default	4.81	0.30	150.98	3.76	15.76	4.77
FreeRTOS Pico 2 RPi4 default/best	6.64	0.49	246.61	5.04	13.73	6.64
FreeRTOS Pico 2 RPi4 default/default	6.81	0.49	255.53	5.34	17.93	6.76
FreeRTOS Pico 2 RPi5 best/best	3.55	0.48	85.12	2.46	12.83	3.85
FreeRTOS Pico 2 RPi5 best/default	3.98	0.27	107.79	2.54	12.36	3.91
FreeRTOS Pico 2 RPi5 default/best	4.59	0.44	139.65	3.41	11.66	4.85
FreeRTOS Pico 2 RPi5 default/default	4.96	0.24	159.04	3.61	10.72	4.90
Zephyr Pico RPi4 best/best	3.55	0.48	85.12	2.46	12.83	3.85
Zephyr Pico RPi4 best/default	3.56	0.19	86.10	2.77	16.87	3.55
Zephyr Pico RPi4 default/best	3.55	1.37	85.34	3.17	203.63	3.52
Zephyr Pico RPi4 default/default	4.35	0.27	127.11	3.57	36.66	4.33
Zephyr Pico RPi5 best/best	3.32	0.48	73.29	2.48	5.05	2.96
Zephyr Pico RPi5 best/default	4.14	0.45	116.22	2.63	37.71	3.94
Zephyr Pico RPi5 default/best	4.93	2.01	157.46	3.28	203.03	3.94
Zephyr Pico RPi5 default/default	5.67	1.32	196.15	3.41	10.30	4.93
Zephyr Nucleo RPi4 best/best	6.26	0.06	226.82	5.81	7.43	6.26
Zephyr Nucleo RPi4 best/default	7.36	0.10	284.35	6.28	8.54	7.37
Zephyr Nucleo RPi4 default/best	8.00	0.12	317.75	7.41	9.82	8.00
Zephyr Nucleo RPi4 default/default	9.02	0.12	370.85	8.07	10.76	9.01
Zephyr Nucleo RPi5 best/best	5.39	0.31	181.38	5.09	19.99	5.29
Zephyr Nucleo RPi5 best/default	7.48	0.16	290.84	5.25	21.92	7.50
Zephyr Nucleo RPi5 default/best	7.02	0.13	266.68	6.82	9.11	7.04
Zephyr Nucleo RPi5 default/default	9.17	0.09	378.67	7.17	9.80	9.17

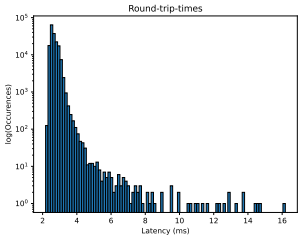
E.2 Bare-metal control loop measurements

E.2.1 Raspberry Pi 4B

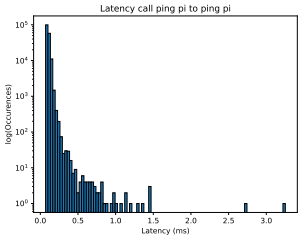
Bare-metal Rasberry Pi Pico μC - best effort publisher and best effort subscriber



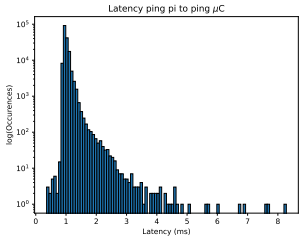
(a) Latency distribution



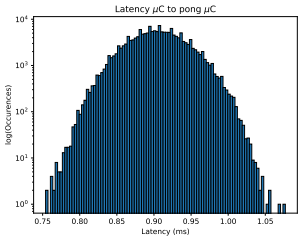
(b) Round-trip times



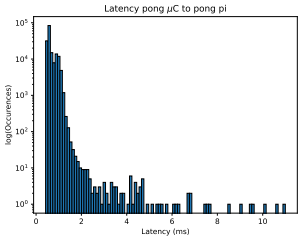
(c) Call ping pi to ping pi (1 → 2)



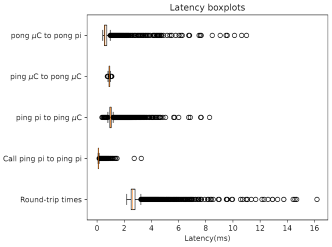
(d) ping pi to ping μC (2 → 3)



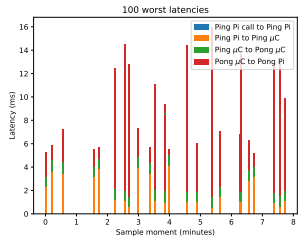
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



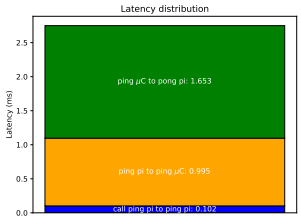
(g) Latency boxplots



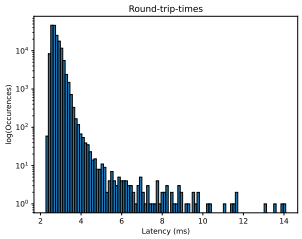
(h) 100 worst latencies

Figure E.3: Raspberry Pi 4B with bare-metal Rasberry Pi Pico μC with - best effort publisher and best effort subscriber

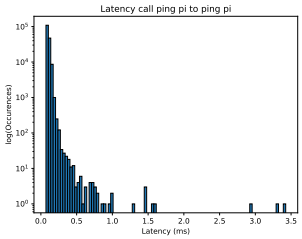
Bare-metal Rasberry Pi Pico μC - default publisher and best effort subscriber



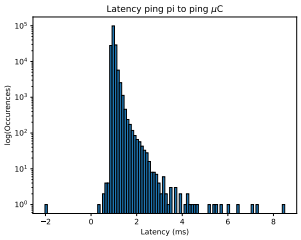
(a) Latency distribution



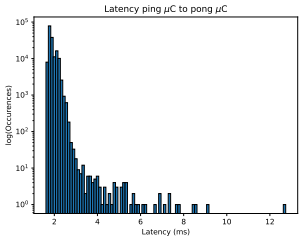
(b) Round-trip times



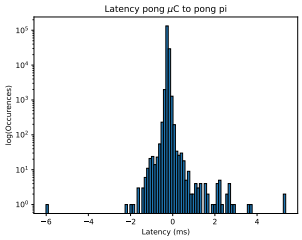
(c) Call ping pi to ping pi (1 → 2)



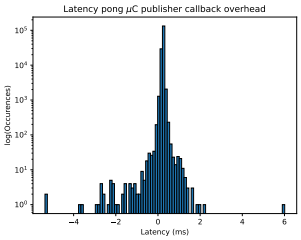
(d) ping pi to ping μC (2 → 3)



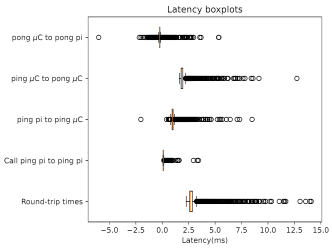
(e) ping μC to pong μC (3 → 4)



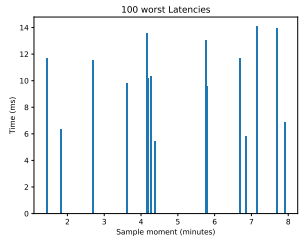
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



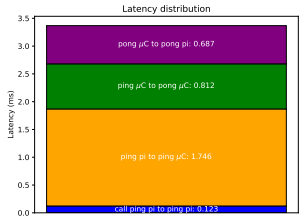
(h) Latency boxplots



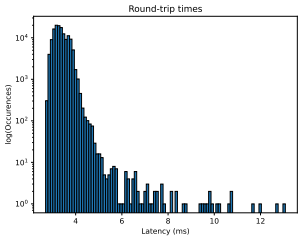
(i) 100 worst latencies

Figure E.4: Raspberry Pi 4B with bare-metal Rasberry Pi Pico μC - default publisher and best effort subscriber

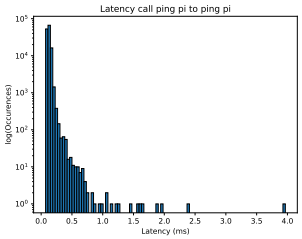
Bare-metal Rasberry Pi Pico μC - best effort publisher and default subscriber



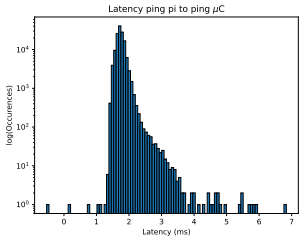
(a) Latency distribution



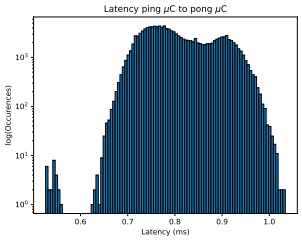
(b) Round-trip times



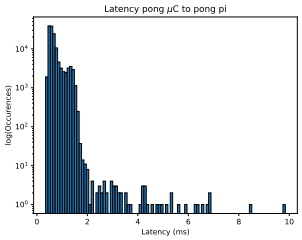
(c) Call ping pi to ping pi (1 → 2)



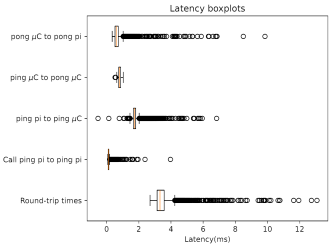
(d) ping pi to ping μC (2 → 3)



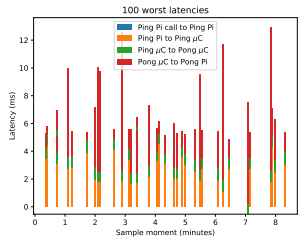
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



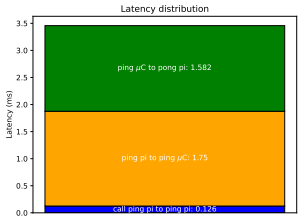
(g) Latency boxplots



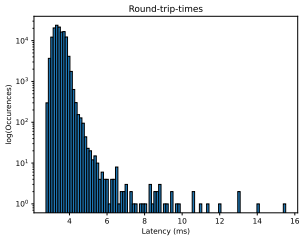
(h) 100 worst latencies

Figure E.5: Raspberry Pi 4B with bare-metal Rasberry Pi Pico μC - best effort publisher and default subscriber

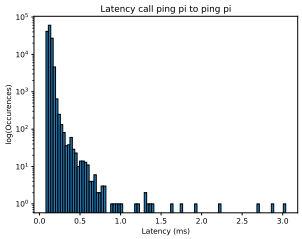
Bare-metal Rasberry Pi Pico μC - default publisher and default subscriber



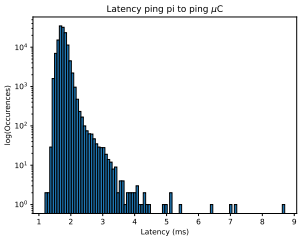
(a) Latency distribution



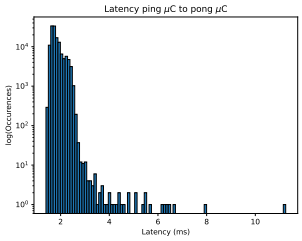
(b) Round-trip times



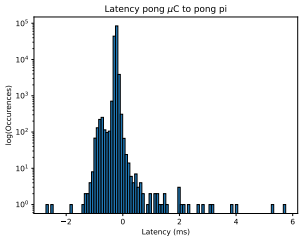
(c) Call ping pi to ping pi ($1 \rightarrow 2$)



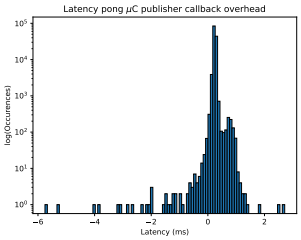
(d) ping pi to ping μC ($2 \rightarrow 3$)



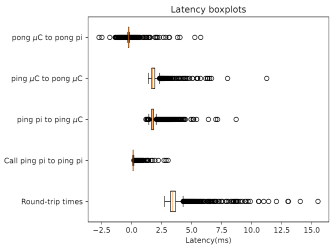
(e) ping μC to pong μC ($3 \rightarrow 4$)



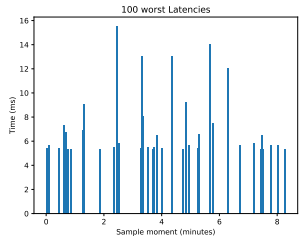
(f) pong μC to pong pi ($4 \rightarrow 5$)



(g) Callback overhead



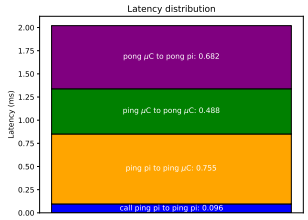
(h) Latency boxplots



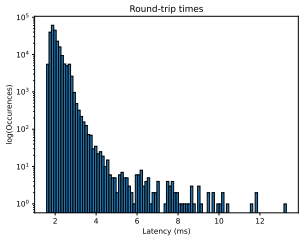
(i) 100 worst latencies

Figure E.6: Raspberry Pi 4B with bare-metal Rasberry Pi Pico μC - default publisher and default subscriber

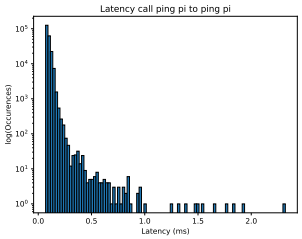
Bare-metal Rasberry Pi Pico 2 μC - best effort publisher and best effort subscriber



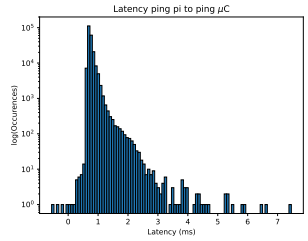
(a) Latency distribution



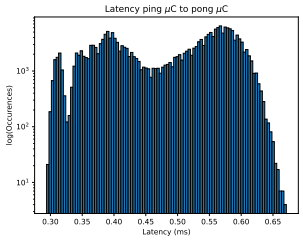
(b) Round-trip times



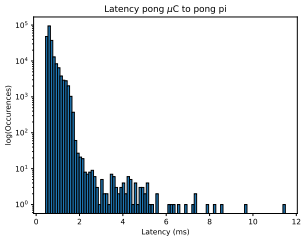
(c) Call ping pi to ping pi (1 → 2)



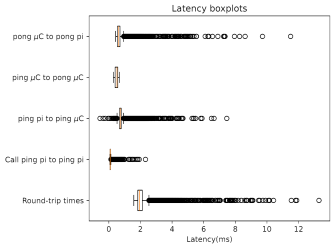
(d) ping pi to ping μC (2 → 3)



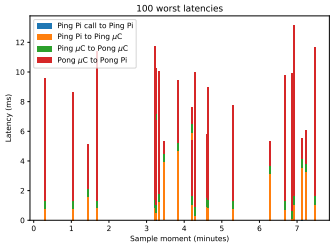
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



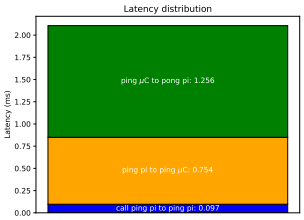
(g) Latency boxplots



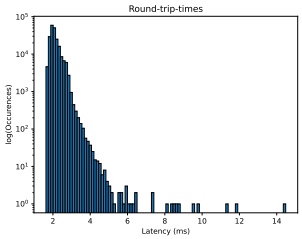
(h) 100 worst latencies

Figure E.7: Raspberry Pi 4B with bare-metal Rasberry Pi Pico 2 μC with - best effort publisher and best effort subscriber

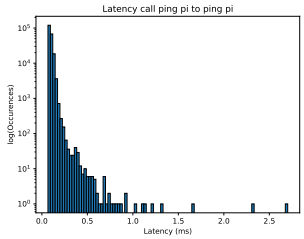
Bare-metal Rasberry Pi Pico 2 μC - default publisher and best effort subscriber



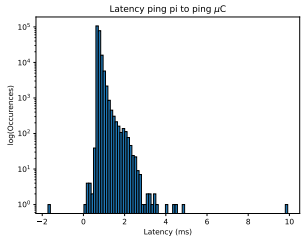
(a) Latency distribution



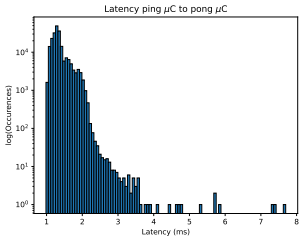
(b) Round-trip times



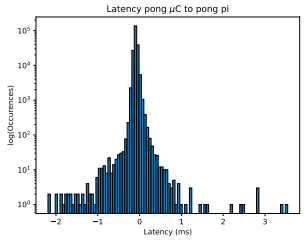
(c) Call ping pi to ping pi (1 → 2)



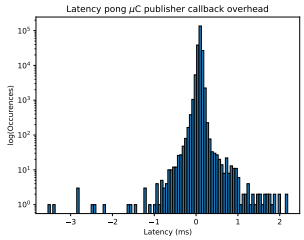
(d) ping pi to ping μC (2 → 3)



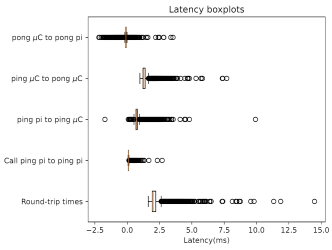
(e) ping μC to pong μC (3 → 4)



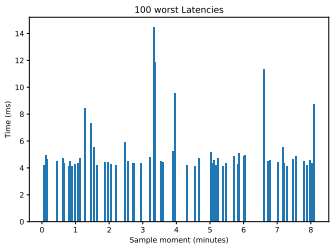
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



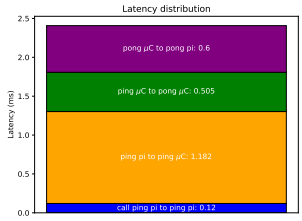
(h) Latency boxplots



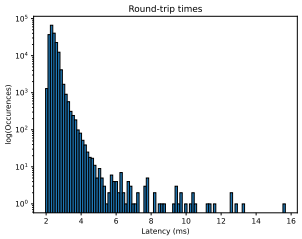
(i) 100 worst latencies

Figure E.8: Raspberry Pi 4B with bare-metal Rasberry Pi Pico 2 μC - default publisher and best effort subscriber

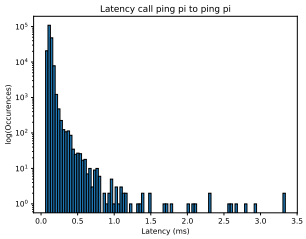
Bare-metal Rasberry Pi Pico 2 μC - best effort publisher and default subscriber



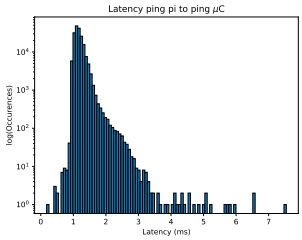
(a) Latency distribution



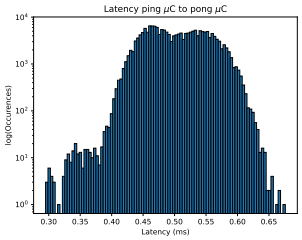
(b) Round-trip times



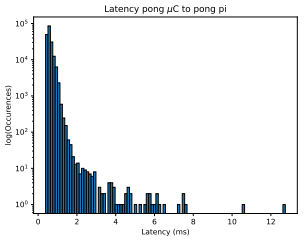
(c) Call ping pi to ping pi (1 → 2)



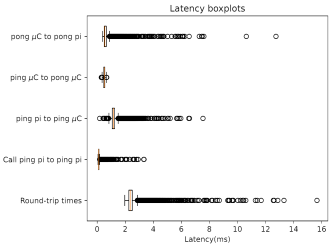
(d) ping pi to ping μC (2 → 3)



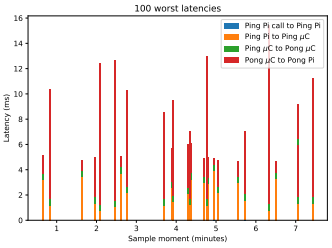
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



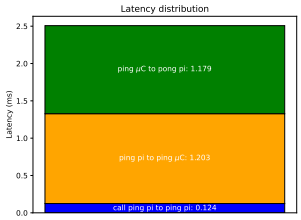
(g) Latency boxplots



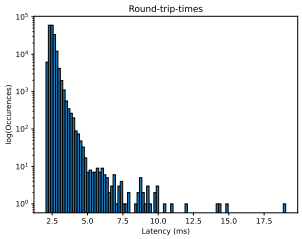
(h) 100 worst latencies

Figure E.9: Raspberry Pi 4B with bare-metal Rasberry Pi Pico 2 μC - best effort publisher and default subscriber

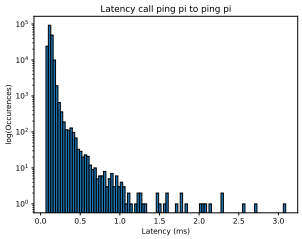
Bare-metal Rasberry Pi Pico 2 μC - default publisher and default subscriber



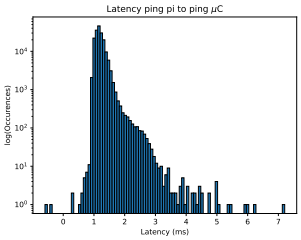
(a) Latency distribution



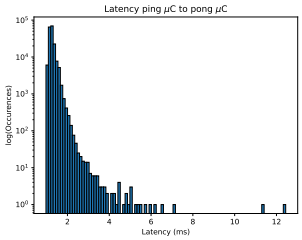
(b) Round-trip times



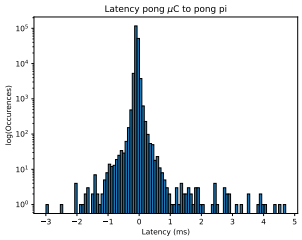
(c) Call ping pi to ping pi (1 → 2)



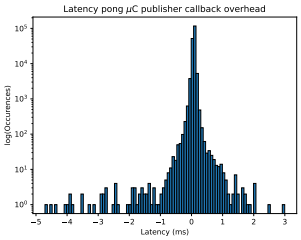
(d) ping pi to ping μC (2 → 3)



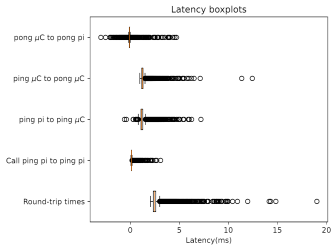
(e) ping μC to pong μC (3 → 4)



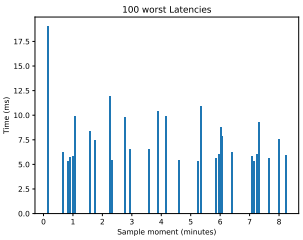
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



(h) Latency boxplots

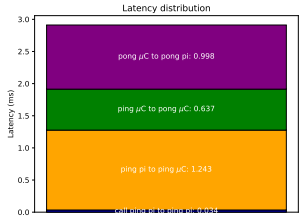


(i) 100 worst latencies

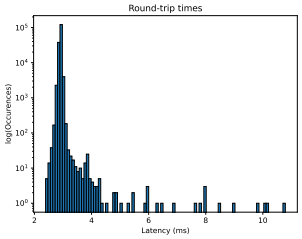
Figure E.10: Raspberry Pi 4B with bare-metal Rasberry Pi Pico 2 μC - default publisher and default subscriber

E.2.2 Raspberry Pi 5

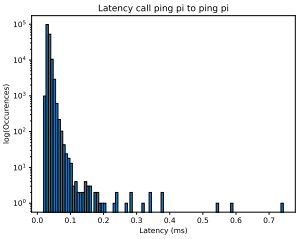
Bare-metal Rasberry Pi Pico μC - best effort publisher and best effort subscriber



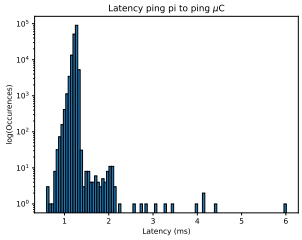
(a) Latency distribution



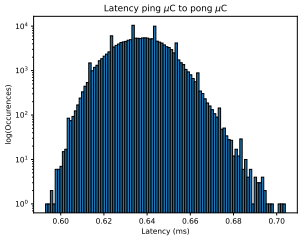
(b) Round-trip times



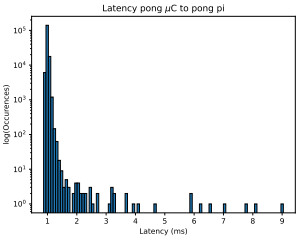
(c) Call ping pi to ping pi (1 → 2)



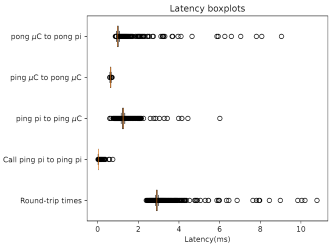
(d) ping pi to ping μC (2 → 3)



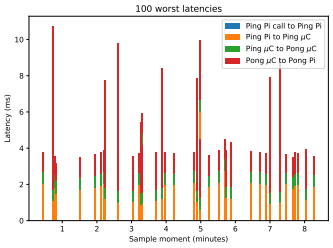
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



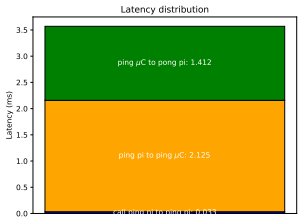
(g) Latency boxplots



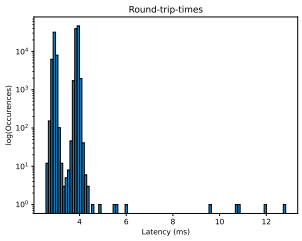
(h) 100 worst latencies

Figure E.11: Raspberry Pi 5 with bare-metal Rasberry Pi Pico μC with - best effort publisher and best effort subscriber

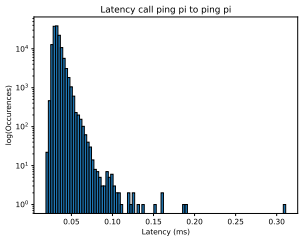
Raspberry Pi 5 with bare-metal Rasberry Pi Pico μC - default publisher and best effort subscriber



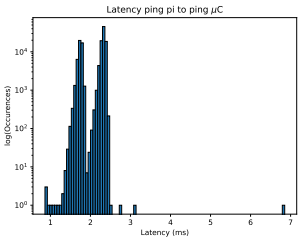
(a) Latency distribution



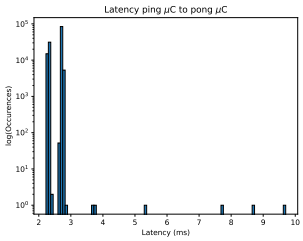
(b) Round-trip times



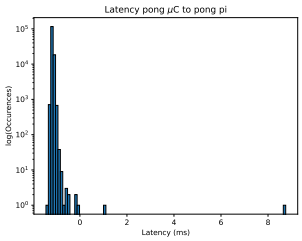
(c) Call ping pi to ping pi (1 → 2)



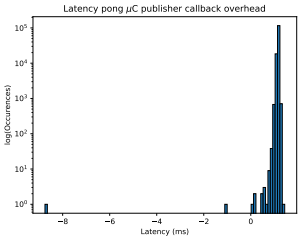
(d) ping pi to ping μC (2 → 3)



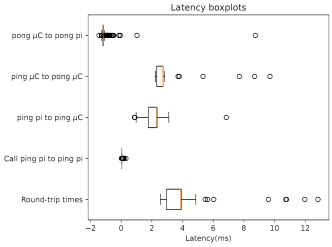
(e) ping μC to pong μC (3 → 4)



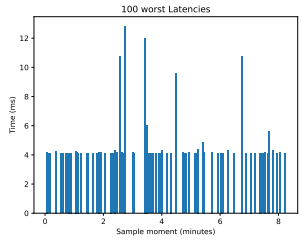
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



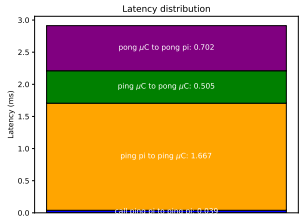
(h) Latency boxplots



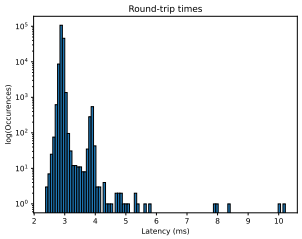
(i) 100 worst latencies

Figure E.12: Raspberry Pi 5 with bare-metal Rasberry Pi Pico μC - default publisher and best effort subscriber

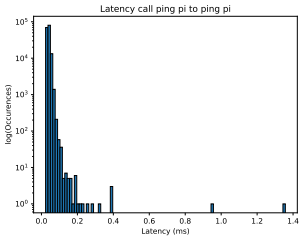
Bare-metal Rasberry Pi Pico μC - best effort publisher and default subscriber



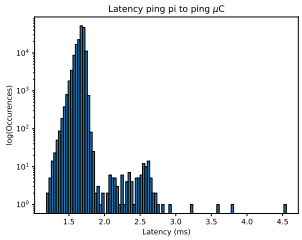
(a) Latency distribution



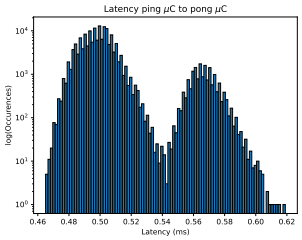
(b) Round-trip times



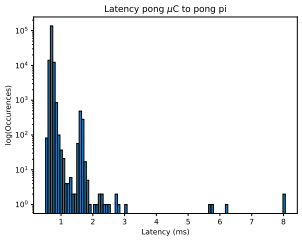
(c) Call ping pi to ping pi (1 → 2)



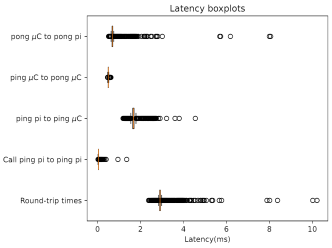
(d) ping pi to ping μC (2 → 3)



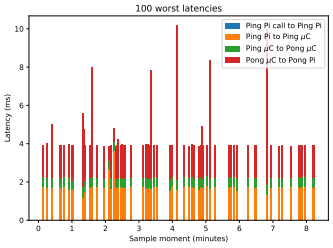
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



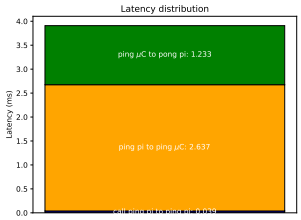
(g) Latency boxplots



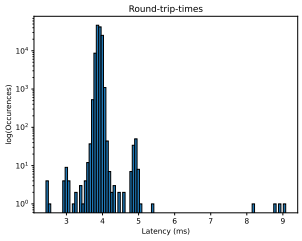
(h) 100 worst latencies

Figure E.13: Raspberry Pi 5 with bare-metal Rasberry Pi Pico μC - best effort publisher and default subscriber

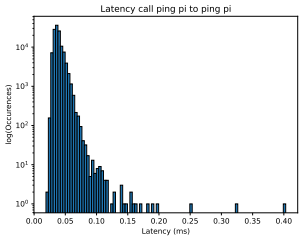
Bare-metal Rasberry Pi Pico μC - default publisher and default subscriber



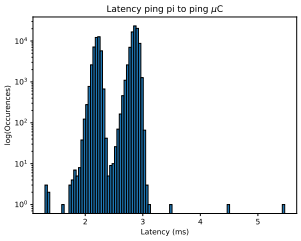
(a) Latency distribution



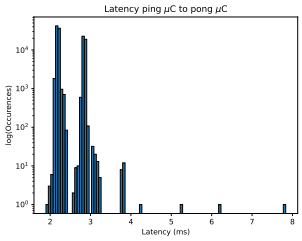
(b) Round-trip times



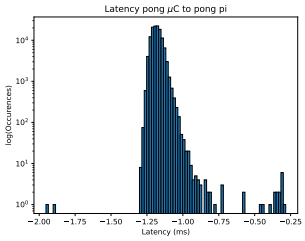
(c) Call ping pi to ping pi (1 → 2)



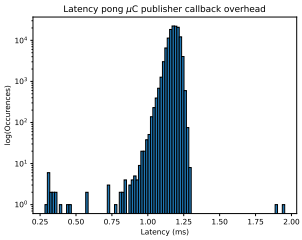
(d) ping pi to ping μC (2 → 3)



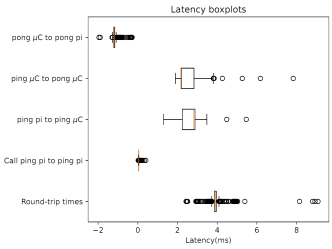
(e) ping μC to pong μC (3 → 4)



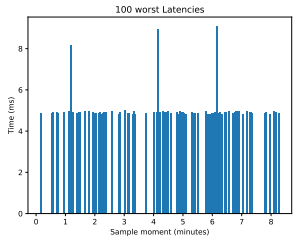
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



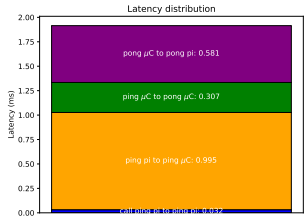
(h) Latency boxplots



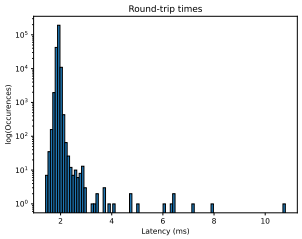
(i) 100 worst latencies

Figure E.14: Raspberry Pi 5 with bare-metal Rasberry Pi Pico μC - default publisher and default subscriber

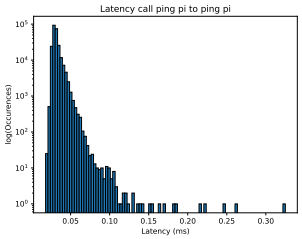
Bare-metal Rasberry Pi Pico 2 μC - best effort publisher and best effort subscriber



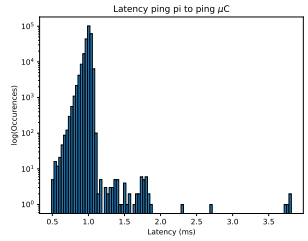
(a) Latency distribution



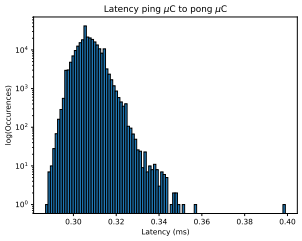
(b) Round-trip times



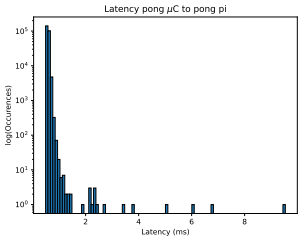
(c) Call ping pi to ping pi (1 → 2)



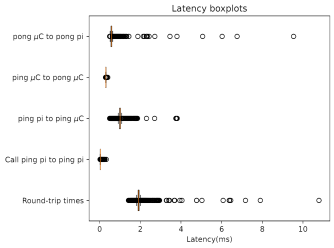
(d) ping pi to ping μC (2 → 3)



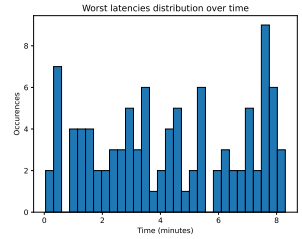
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



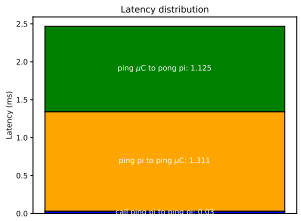
(g) Latency boxplots



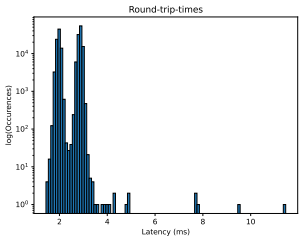
(h) 100 worst latencies

Figure E.15: Raspberry Pi 5 with bare-metal Rasberry Pi Pico 2 μC with - best effort publisher and best effort subscriber

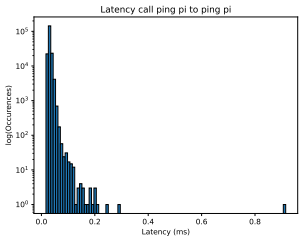
Raspberry Pi 5 with bare-metal Rasberry Pi Pico 2 μC - default publisher and best effort subscriber



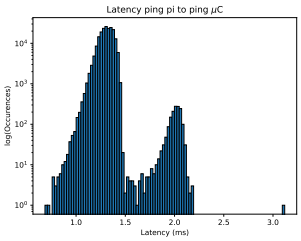
(a) Latency distribution



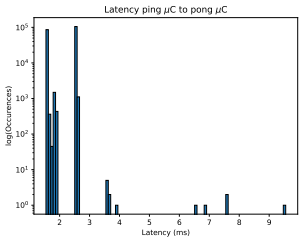
(b) Round-trip times



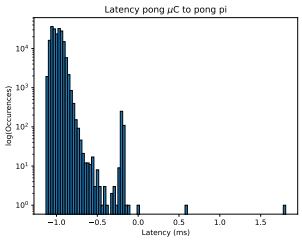
(c) Call ping pi to ping pi (1 → 2)



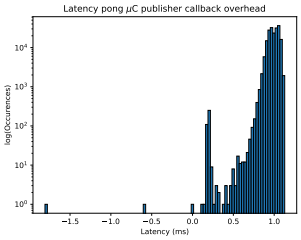
(d) ping pi to ping μC (2 → 3)



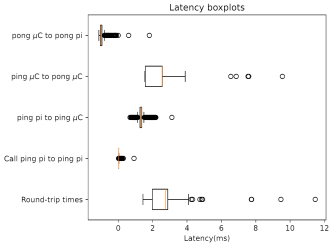
(e) ping μC to pong μC (3 → 4)



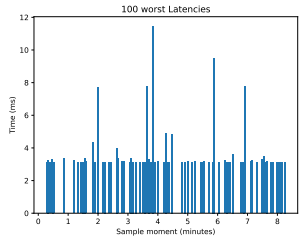
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



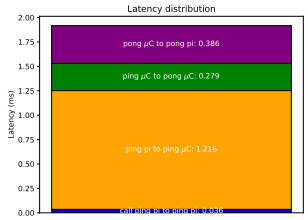
(h) Latency boxplots



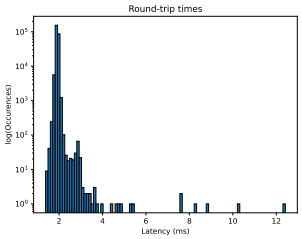
(i) 100 worst latencies

Figure E.16: Raspberry Pi 5 with bare-metal Rasberry Pi Pico 2 μC - default publisher and best effort subscriber

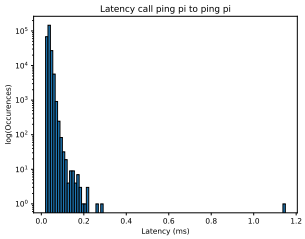
Bare-metal Rasberry Pi Pico 2 μC - best effort publisher and default subscriber



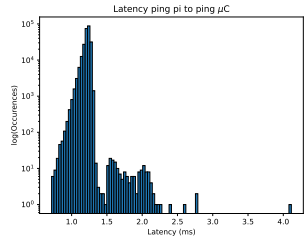
(a) Latency distribution



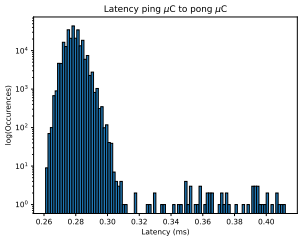
(b) Round-trip times



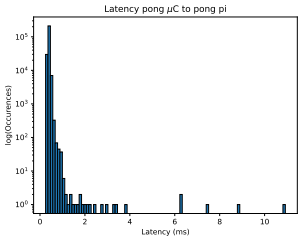
(c) Call ping pi to ping pi (1 → 2)



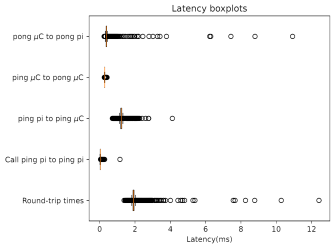
(d) ping pi to ping μC (2 → 3)



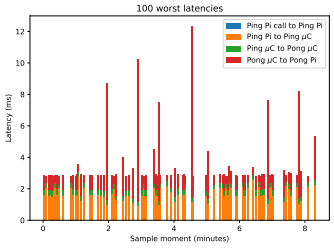
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



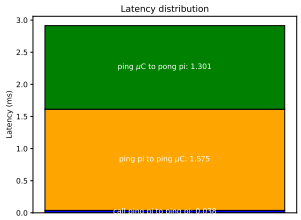
(g) Latency boxplots



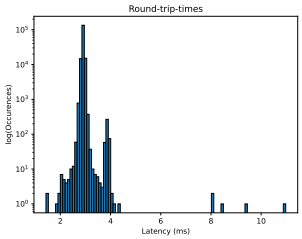
(h) 100 worst latencies

Figure E.17: Raspberry Pi 5 with bare-metal Rasberry Pi Pico 2 μC - best effort publisher and default subscriber

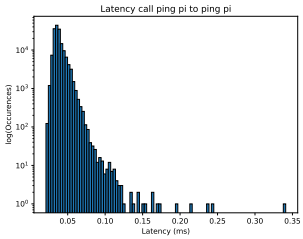
Bare-metal Rasberry Pi Pico 2 μC - default publisher and default subscriber



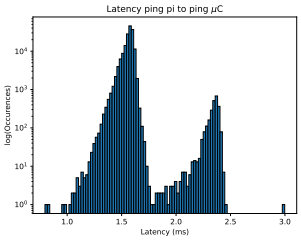
(a) Latency distribution



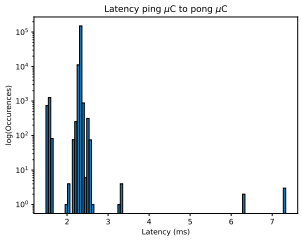
(b) Round-trip times



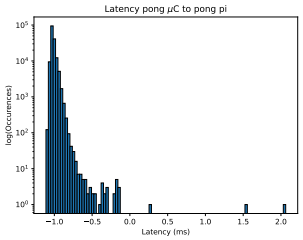
(c) Call ping pi to ping pi (1 → 2)



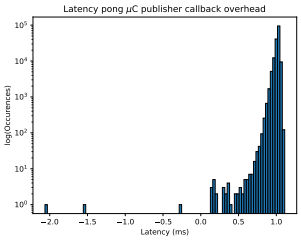
(d) ping pi to ping μC (2 → 3)



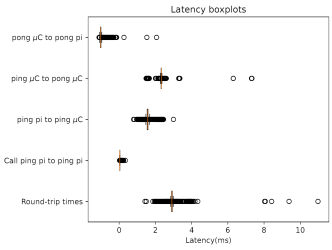
(e) ping μC to pong μC (3 → 4)



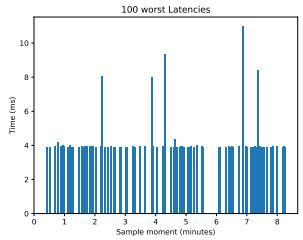
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



(h) Latency boxplots



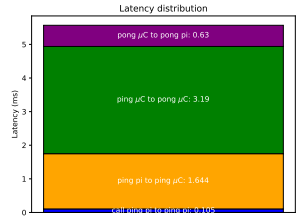
(i) 100 worst latencies

Figure E.18: Raspberry Pi 5 with bare-metal Rasberry Pi Pico 2 μC - default publisher and default subscriber

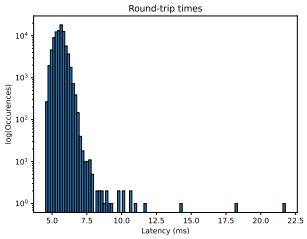
E.3 FreeRTOS control loop measurements

E.3.1 Raspberry Pi 4B

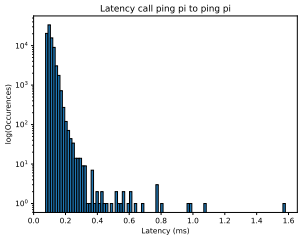
FreeRTOS Rasberry Pi Pico μC - best effort publisher and best effort subscriber



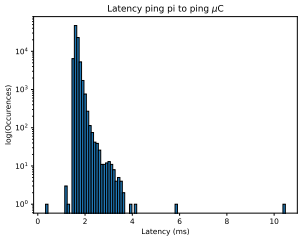
(a) Latency distribution



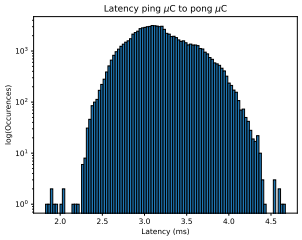
(b) Round-trip times



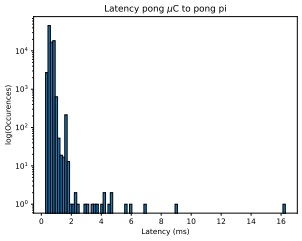
(c) Call ping pi to ping pi (1 → 2)



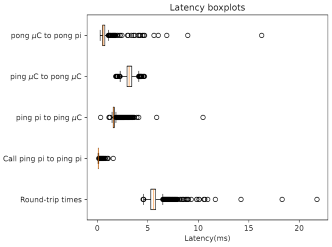
(d) ping pi to ping μC (2 → 3)



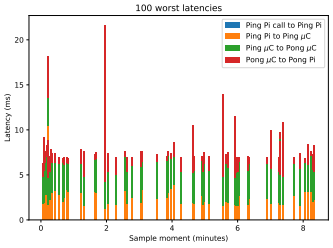
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



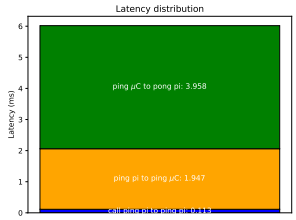
(g) Latency boxplots



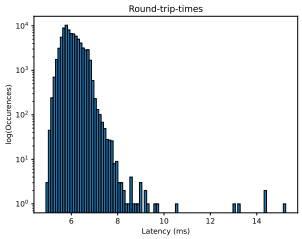
(h) 100 worst latencies

Figure E.19: Raspberry Pi 4B with FreeRTOS Rasberry Pi Pico μC with - best effort publisher and best effort subscriber

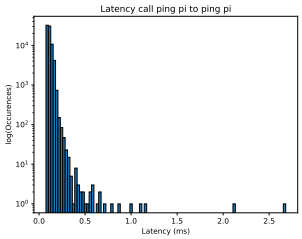
FreeRTOS Rasberry Pi Pico μC - default publisher and best effort subscriber



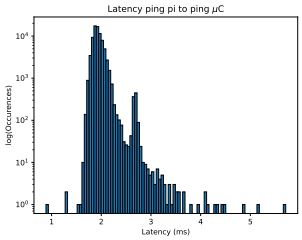
(a) Latency distribution



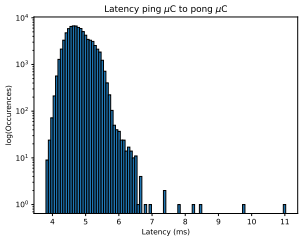
(b) Round-trip times



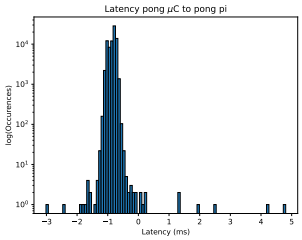
(c) Call ping pi to ping pi (1 → 2)



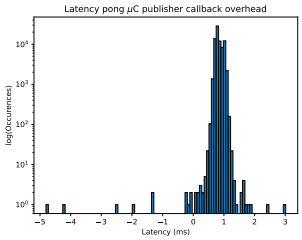
(d) ping pi to ping μC (2 → 3)



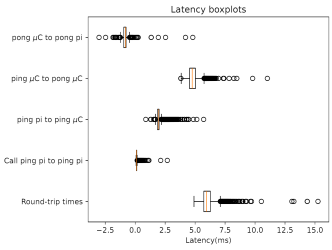
(e) ping μC to pong μC (3 → 4)



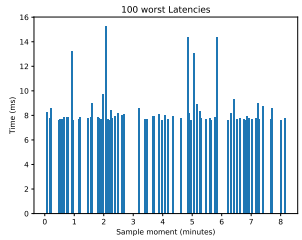
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



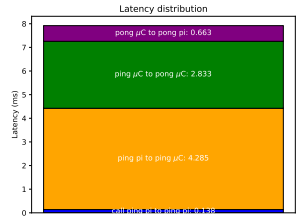
(h) Latency boxplots



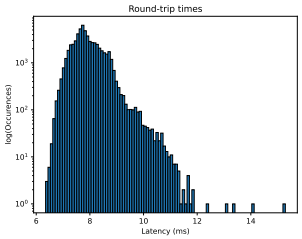
(i) 100 worst latencies

Figure E.20: Raspberry Pi 4B with FreeRTOS Rasberry Pi Pico μC - default publisher and best effort subscriber

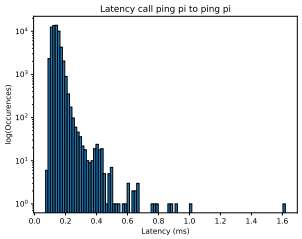
FreeRTOS Rasberry Pi Pico μC - best effort publisher and default subscriber



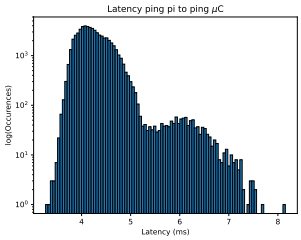
(a) Latency distribution



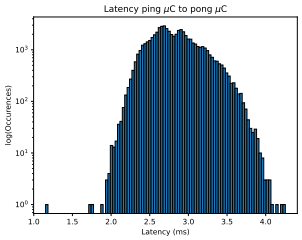
(b) Round-trip times



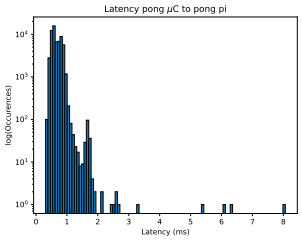
(c) Call ping pi to ping pi (1 → 2)



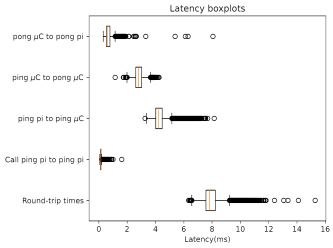
(d) ping pi to ping μC (2 → 3)



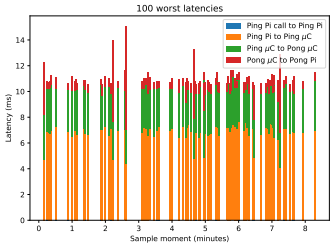
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



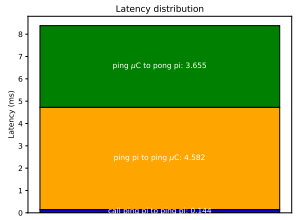
(g) Latency boxplots



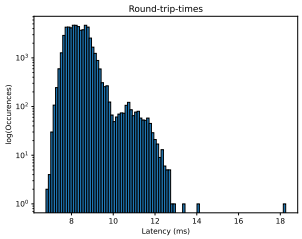
(h) 100 worst latencies

Figure E.21: Raspberry Pi 4B with FreeRTOS Rasberry Pi Pico μC - best effort publisher and default subscriber

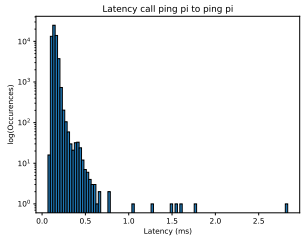
FreeRTOS Rasberry Pi Pico μC - default publisher and default subscriber



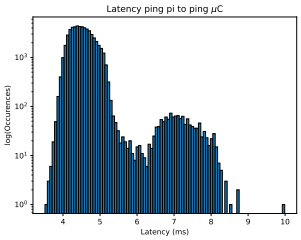
(a) Latency distribution



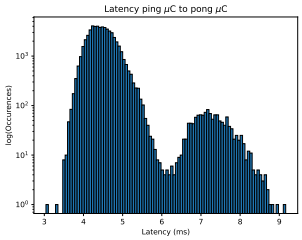
(b) Round-trip times



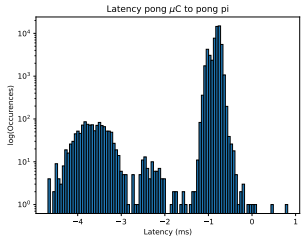
(c) Call ping pi to ping pi (1 → 2)



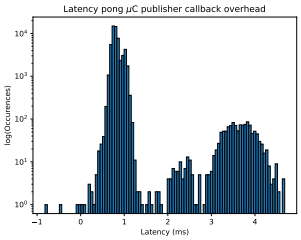
(d) ping pi to ping μC (2 → 3)



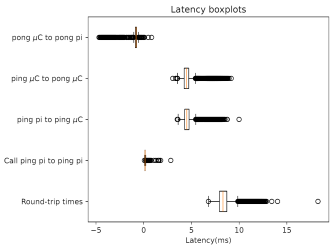
(e) ping μC to pong μC (3 → 4)



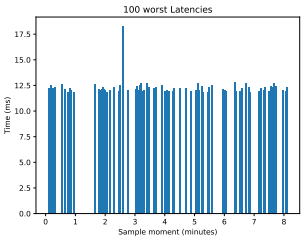
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



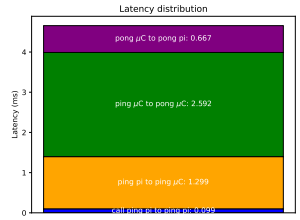
(h) Latency boxplots



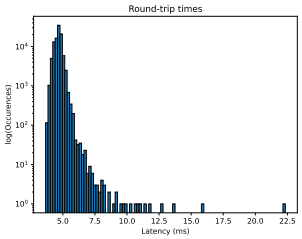
(i) 100 worst latencies

Figure E.22: Raspberry Pi 4B with FreeRTOS Rasberry Pi Pico μC - default publisher and default subscriber

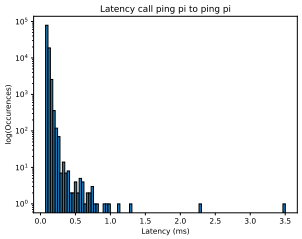
FreeRTOS Rasberry Pi Pico 2 μC - best effort publisher and best effort subscriber



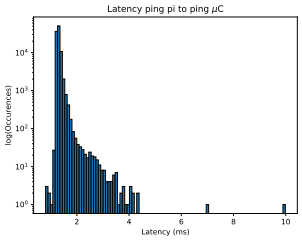
(a) Latency distribution



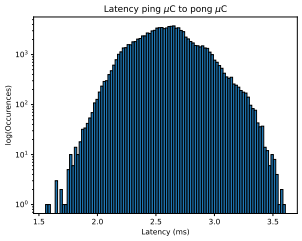
(b) Round-trip times



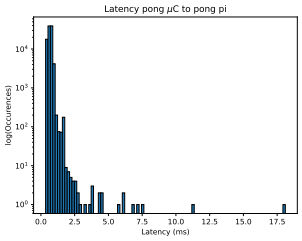
(c) Call ping pi to ping pi (1 → 2)



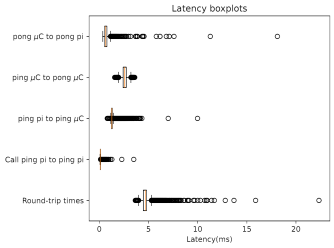
(d) ping pi to ping μC (2 → 3)



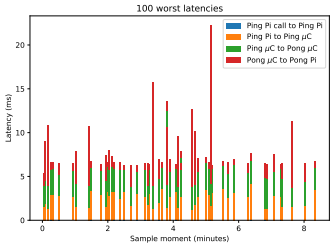
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



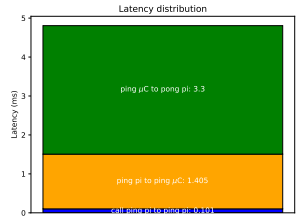
(g) Latency boxplots



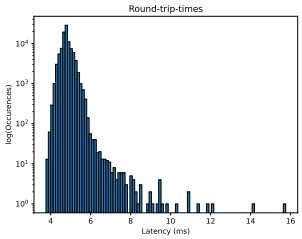
(h) 100 worst latencies

Figure E.23: Raspberry Pi 4B with FreeRTOS Rasberry Pi Pico 2 μC with - best effort publisher and best effort subscriber

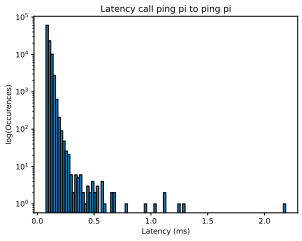
FreeRTOS Rasberry Pi Pico 2 μC - default publisher and best effort subscriber



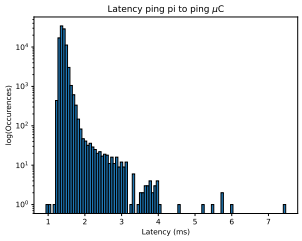
(a) Latency distribution



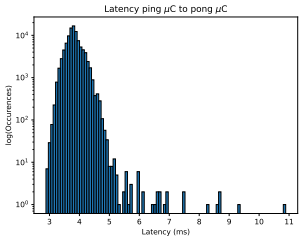
(b) Round-trip times



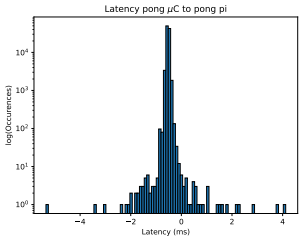
(c) Call ping pi to ping pi (1 → 2)



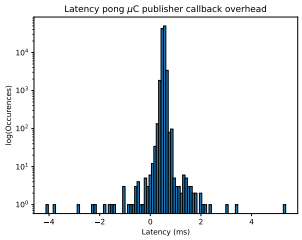
(d) ping pi to ping μC (2 → 3)



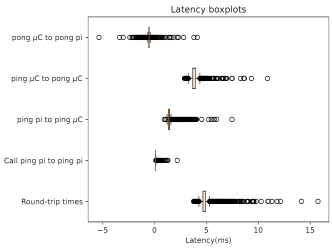
(e) ping μC to pong μC (3 → 4)



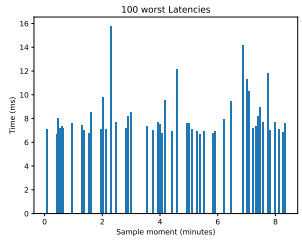
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



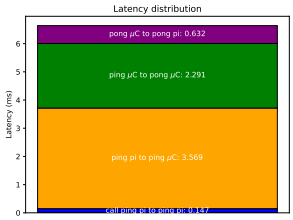
(h) Latency boxplots



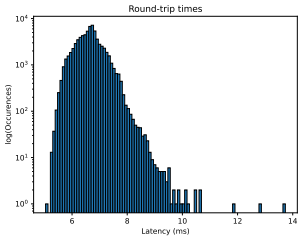
(i) 100 worst latencies

Figure E.24: Raspberry Pi 4B with FreeRTOS Rasberry Pi Pico 2 μC - default publisher and best effort subscriber

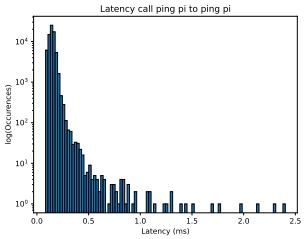
FreeRTOS Rasberry Pi Pico 2 μC - best effort publisher and default subscriber



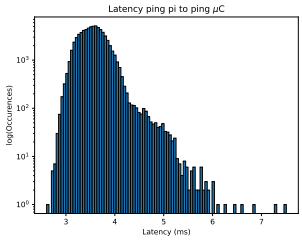
(a) Latency distribution



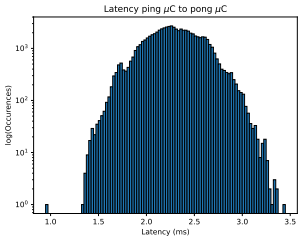
(b) Round-trip times



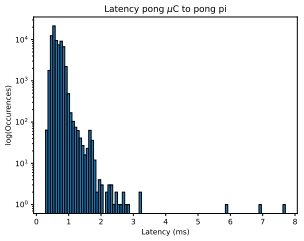
(c) Call ping pi to ping pi (1 → 2)



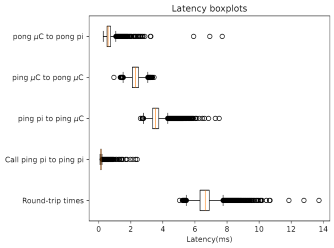
(d) ping pi to ping μC (2 → 3)



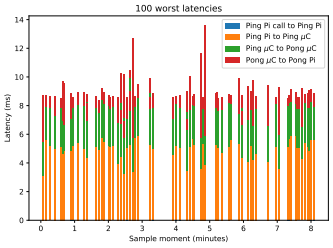
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



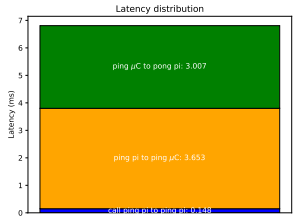
(g) Latency boxplots



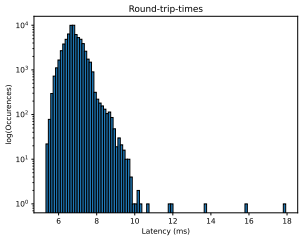
(h) 100 worst latencies

Figure E.25: Raspberry Pi 4B with FreeRTOS Rasberry Pi Pico 2 μC - best effort publisher and default subscriber

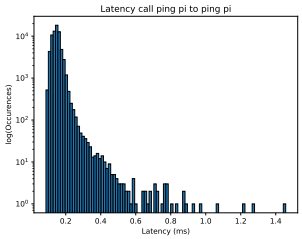
FreeRTOS Rasberry Pi Pico 2 μC - default publisher and default subscriber



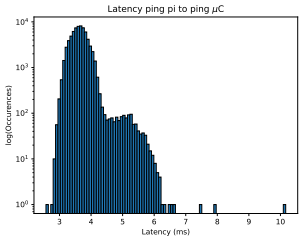
(a) Latency distribution



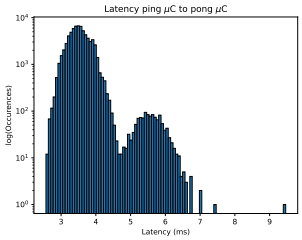
(b) Round-trip times



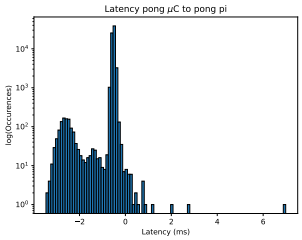
(c) Call ping pi to ping pi (1 → 2)



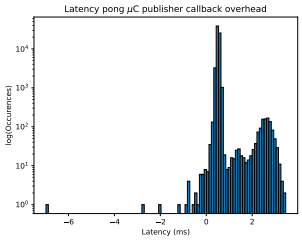
(d) ping pi to ping μC (2 → 3)



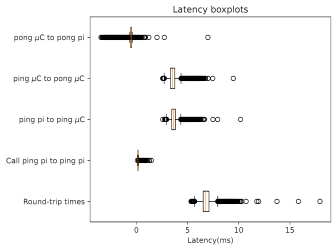
(e) ping μC to pong μC (3 → 4)



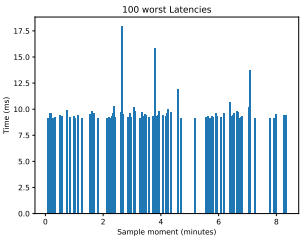
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



(h) Latency boxplots

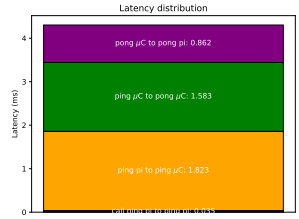


(i) 100 worst latencies

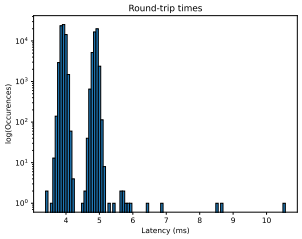
Figure E.26: Raspberry Pi 4B with FreeRTOS Rasberry Pi Pico 2 μC - default publisher and default subscriber

E.3.2 Raspberry Pi 5

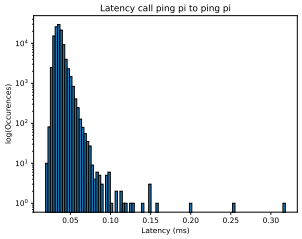
FreeRTOS Rasberry Pi Pico μC - best effort publisher and best effort subscriber



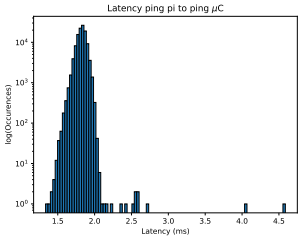
(a) Latency distribution



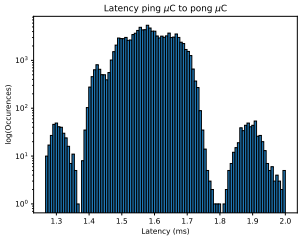
(b) Round-trip times



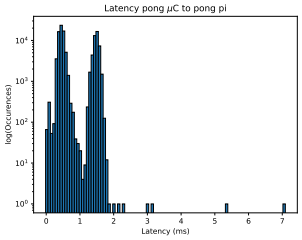
(c) Call ping pi to ping pi (1 → 2)



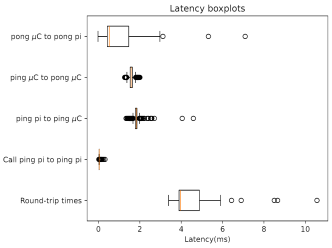
(d) ping pi to ping μC (2 → 3)



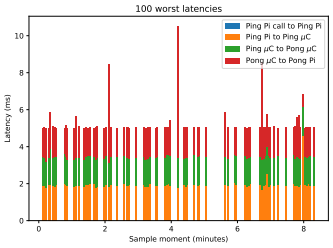
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



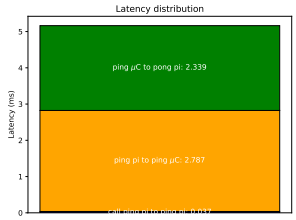
(g) Latency boxplots



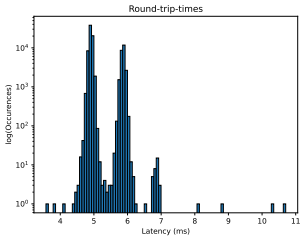
(h) 100 worst latencies

Figure E.27: Raspberry Pi 5 with FreeRTOS Rasberry Pi Pico μC with - best effort publisher and best effort subscriber

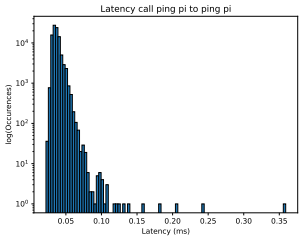
FreeRTOS Rasberry Pi Pico μC - default publisher and best effort subscriber



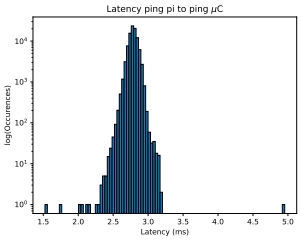
(a) Latency distribution



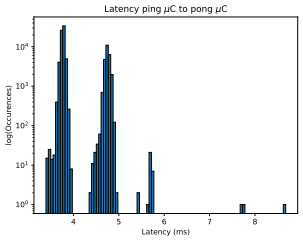
(b) Round-trip times



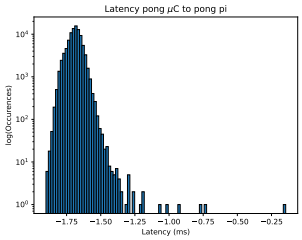
(c) Call ping pi to ping pi (1 → 2)



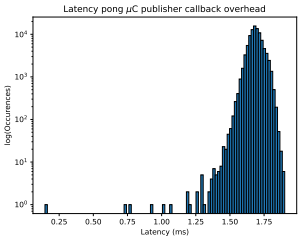
(d) ping pi to ping μC (2 → 3)



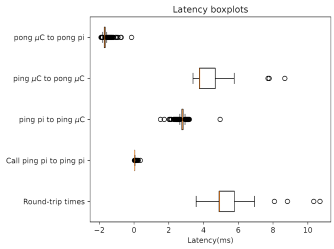
(e) ping μC to pong μC (3 → 4)



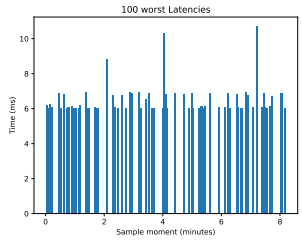
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



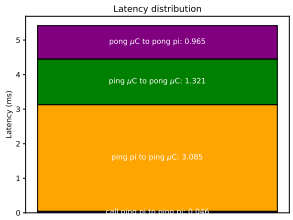
(h) Latency boxplots



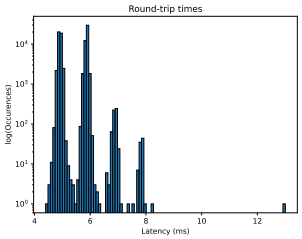
(i) 100 worst latencies

Figure E.28: Raspberry Pi 5 with FreeRTOS Rasberry Pi Pico μC - default publisher and best effort subscriber

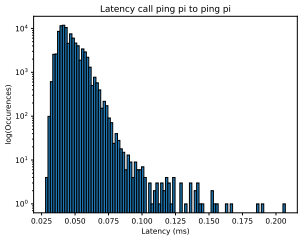
FreeRTOS Rasberry Pi Pico μC - best effort publisher and default subscriber



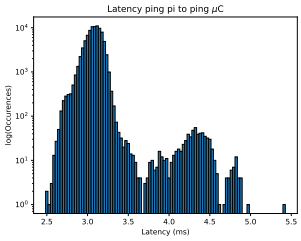
(a) Latency distribution



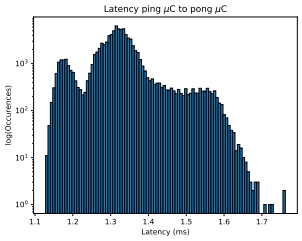
(b) Round-trip times



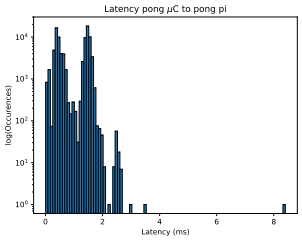
(c) Call ping pi to ping pi (1 → 2)



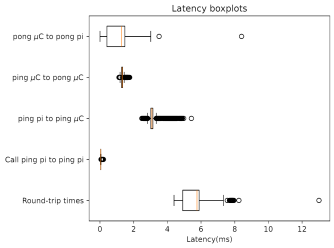
(d) ping pi to ping μC (2 → 3)



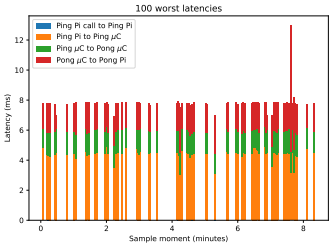
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



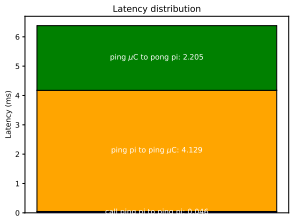
(g) Latency boxplots



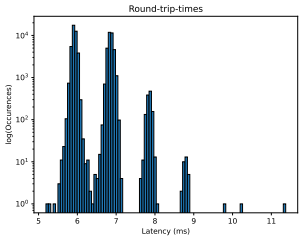
(h) 100 worst latencies

Figure E.29: Raspberry Pi 5 with FreeRTOS Rasberry Pi Pico μC - best effort publisher and default subscriber

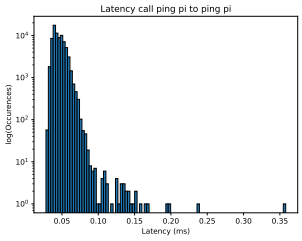
FreeRTOS Rasberry Pi Pico μC - default publisher and default subscriber



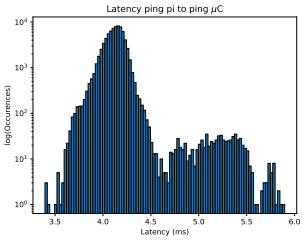
(a) Latency distribution



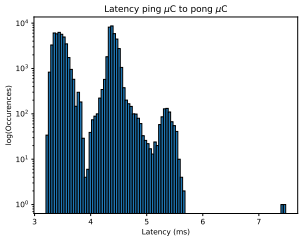
(b) Round-trip times



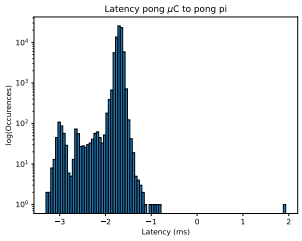
(c) Call ping pi to ping pi (1 → 2)



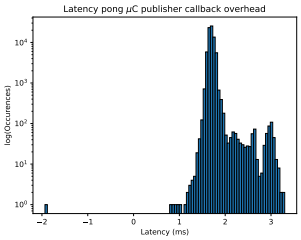
(d) ping pi to ping μC (2 → 3)



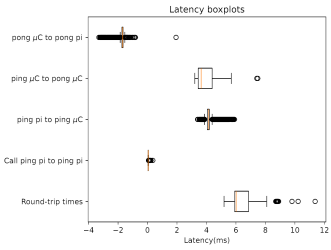
(e) ping μC to pong μC (3 → 4)



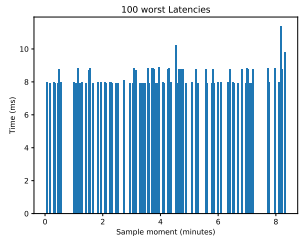
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



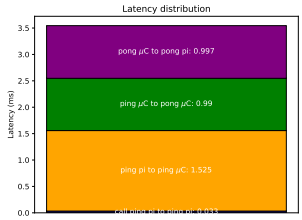
(h) Latency boxplots



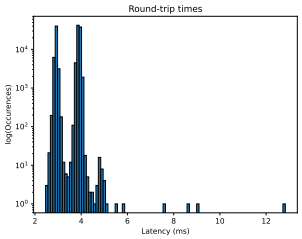
(i) 100 worst latencies

Figure E.30: Raspberry Pi 5 with FreeRTOS Rasberry Pi Pico μC - default publisher and default subscriber

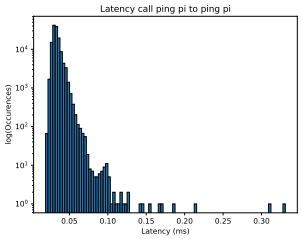
FreeRTOS Rasberry Pi Pico 2 μC - best effort publisher and best effort subscriber



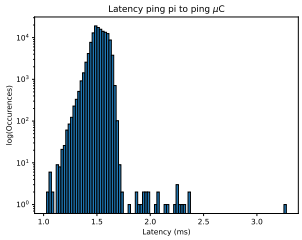
(a) Latency distribution



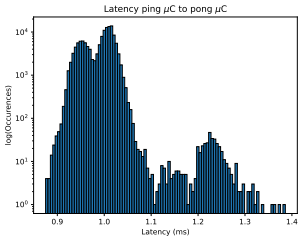
(b) Round-trip times



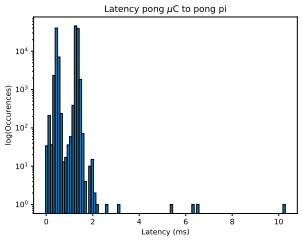
(c) Call ping pi to ping pi (1 → 2)



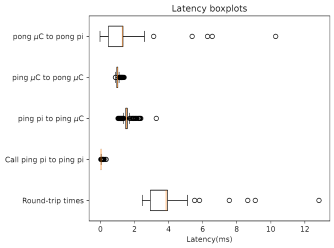
(d) ping pi to ping μC (2 → 3)



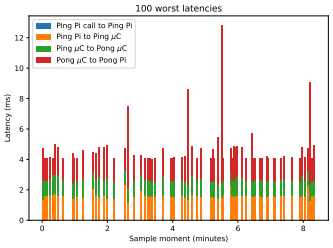
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



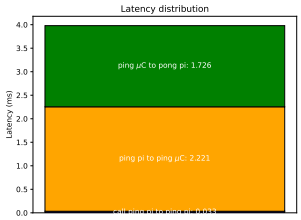
(g) Latency boxplots



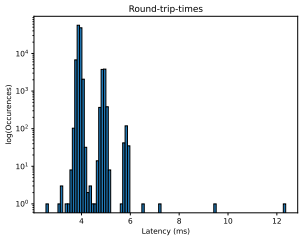
(h) 100 worst latencies

Figure E.31: Raspberry Pi 5 with FreeRTOS Rasberry Pi Pico 2 μC with - best effort publisher and best effort subscriber

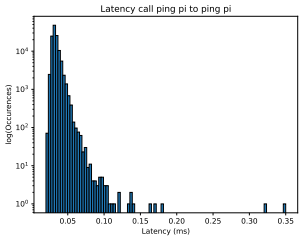
FreeRTOS Rasberry Pi Pico 2 μC - default publisher and best effort subscriber



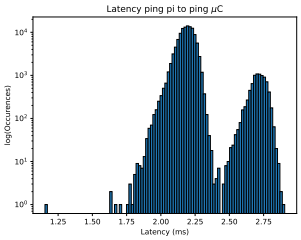
(a) Latency distribution



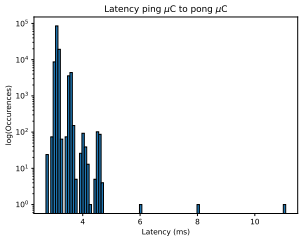
(b) Round-trip times



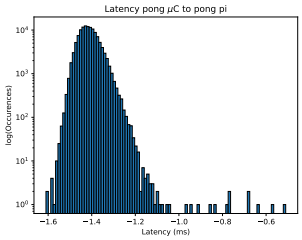
(c) Call ping pi to ping pi (1 → 2)



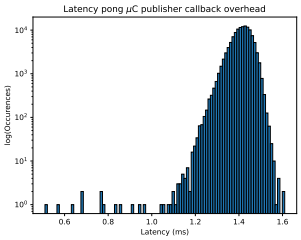
(d) ping pi to ping μC (2 → 3)



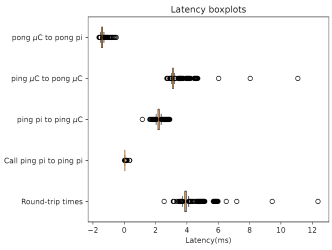
(e) ping μC to pong μC (3 → 4)



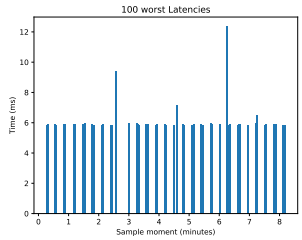
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



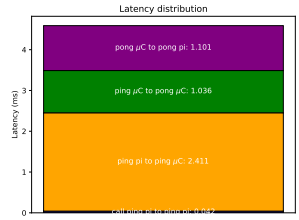
(h) Latency boxplots



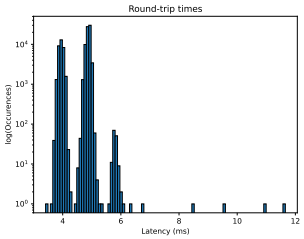
(i) 100 worst latencies

Figure E.32: Raspberry Pi 5 with FreeRTOS Rasberry Pi Pico 2 μC - default publisher and best effort subscriber

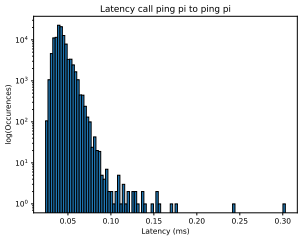
FreeRTOS Rasberry Pi Pico 2μC - best effort publisher and default subscriber



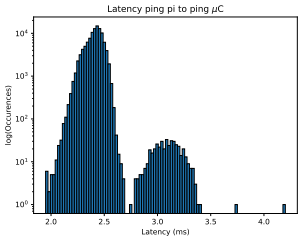
(a) Latency distribution



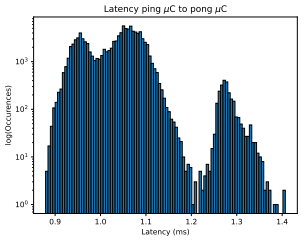
(b) Round-trip times



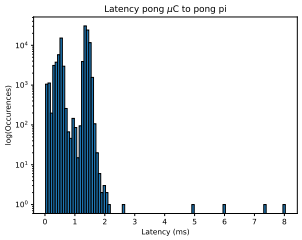
(c) Call ping pi to ping pi (1 → 2)



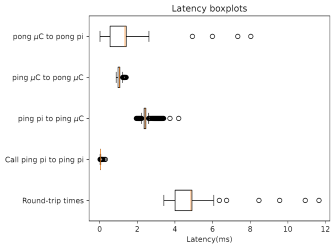
(d) ping pi to ping μC (2 → 3)



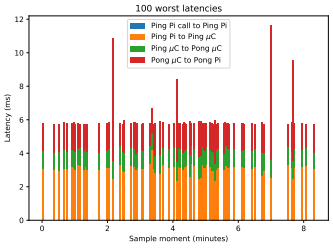
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



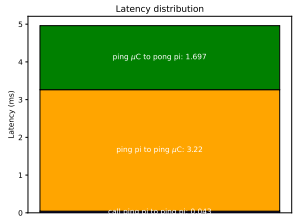
(g) Latency boxplots



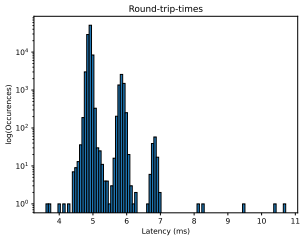
(h) 100 worst latencies

Figure E.33: Raspberry Pi 5 with FreeRTOS Rasberry Pi Pico 2 μC - best effort publisher and default subscriber

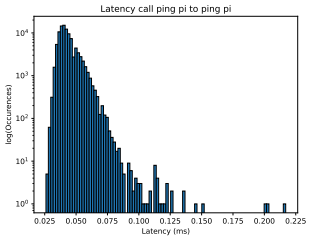
FreeRTOS Rasberry Pi Pico 2 μC - default publisher and default subscriber



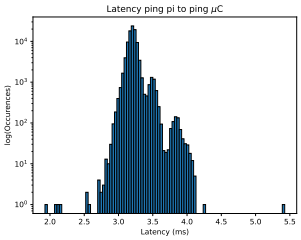
(a) Latency distribution



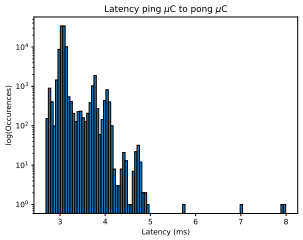
(b) Round-trip times



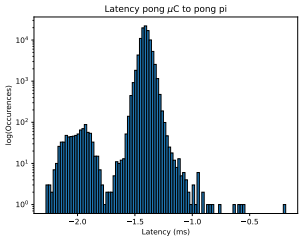
(c) Call ping pi to ping pi (1 → 2)



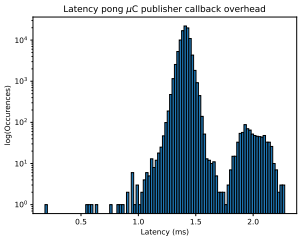
(d) ping pi to ping μC (2 → 3)



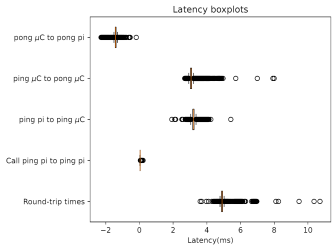
(e) ping μC to pong μC (3 → 4)



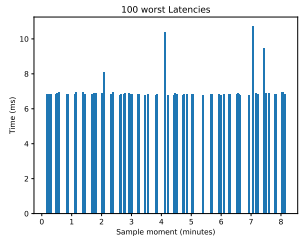
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



(h) Latency boxplots



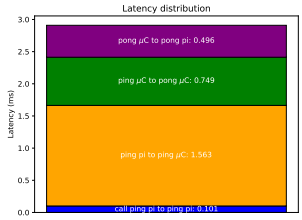
(i) 100 worst latencies

Figure E.34: Raspberry Pi 5 with FreeRTOS Rasberry Pi Pico 2 μC - default publisher and default subscriber

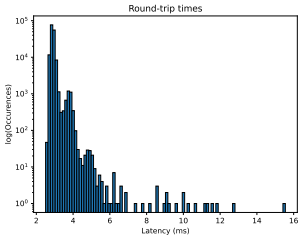
E.4 Zephyr control loop measurements

E.4.1 Raspberry Pi 4B

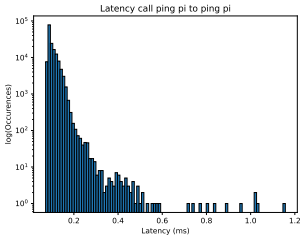
Zephyr Rasberry Pi Pico μC - best effort publisher and best effort subscriber



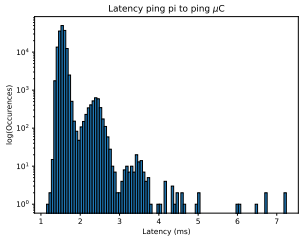
(a) Latency distribution



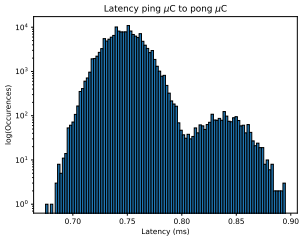
(b) Round-trip times



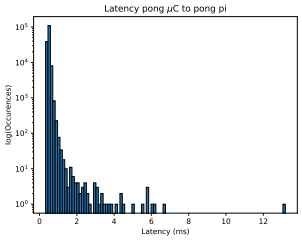
(c) Call ping pi to ping pi (1 → 2)



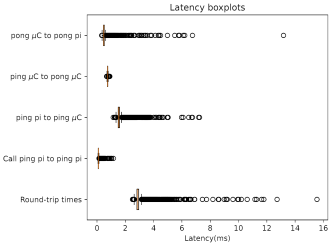
(d) ping pi to ping μC (2 → 3)



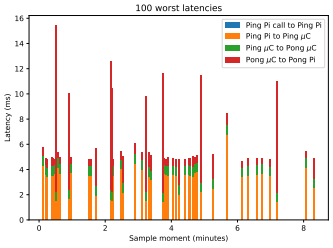
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



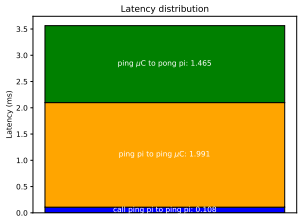
(g) Latency boxplots



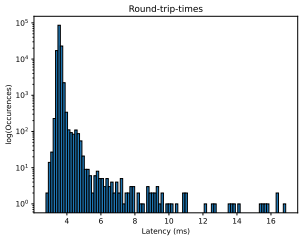
(h) 100 worst latencies

Figure E.35: Raspberry Pi 4B with Zephyr Rasberry Pi Pico μC with - best effort publisher and best effort subscriber

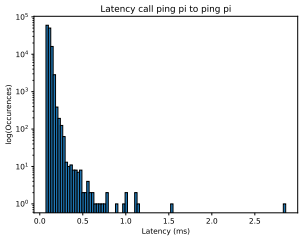
Zephyr Rasberry Pi Pico μC - default publisher and best effort subscriber



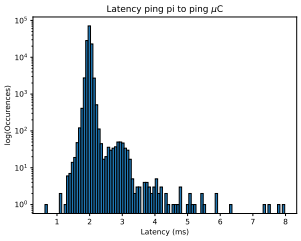
(a) Latency distribution



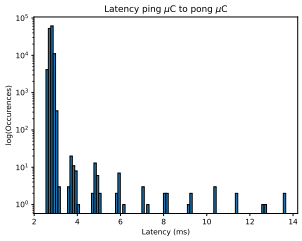
(b) Round-trip times



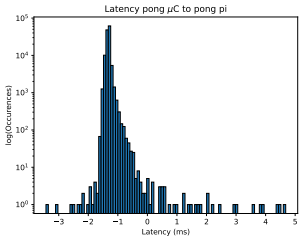
(c) Call ping pi to ping pi (1 → 2)



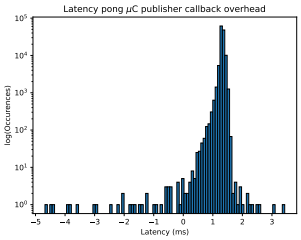
(d) ping pi to ping μC (2 → 3)



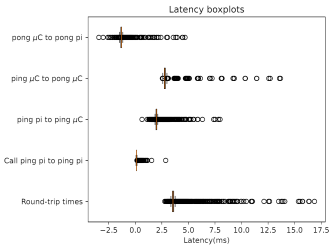
(e) ping μC to pong μC (3 → 4)



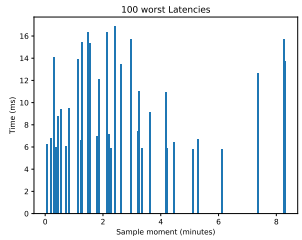
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



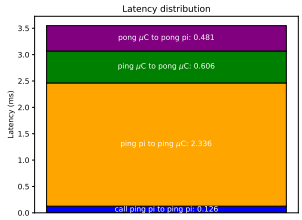
(h) Latency boxplots



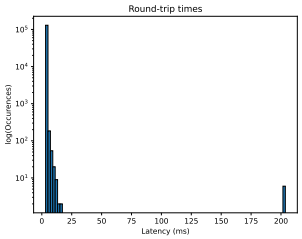
(i) 100 worst latencies

Figure E.36: Raspberry Pi 4B with Zephyr Rasberry Pi Pico μC - default publisher and best effort subscriber

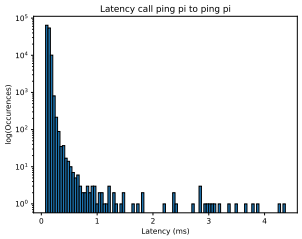
Zephyr Rasberry Pi Pico μC - best effort publisher and default subscriber



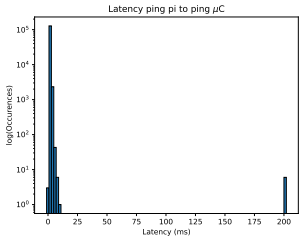
(a) Latency distribution



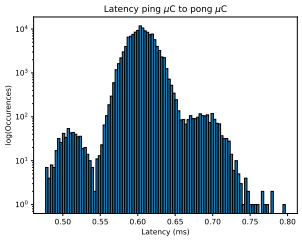
(b) Round-trip times



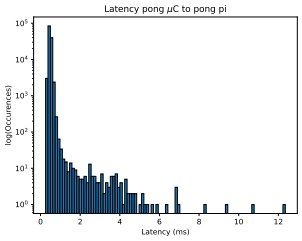
(c) Call ping pi to ping pi (1 → 2)



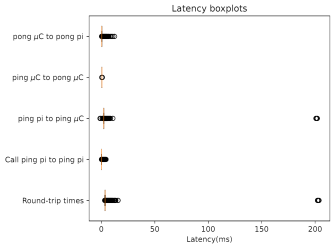
(d) ping pi to ping μC (2 → 3)



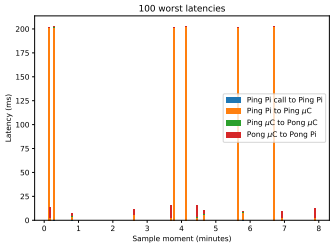
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



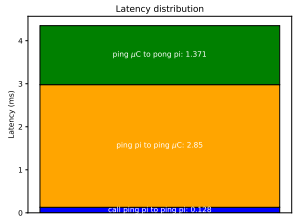
(g) Latency boxplots



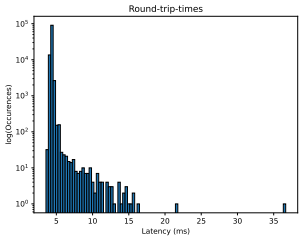
(h) 100 worst latencies

Figure E.37: Raspberry Pi 4B with Zephyr Rasberry Pi Pico μC - best effort publisher and default subscriber

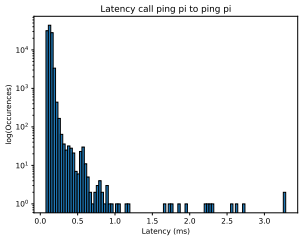
Zephyr Rasberry Pi Pico μC - default publisher and default subscriber



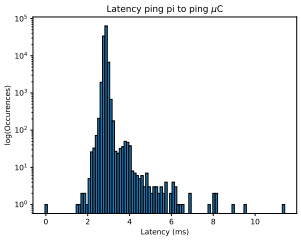
(a) Latency distribution



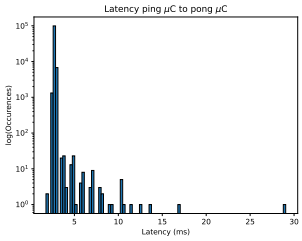
(b) Round-trip times



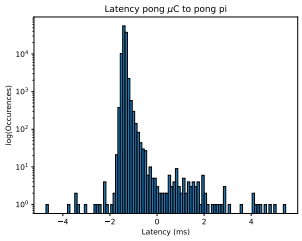
(c) Call ping pi to ping pi (1 → 2)



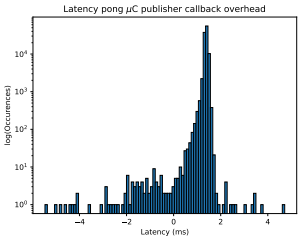
(d) ping pi to ping μC (2 → 3)



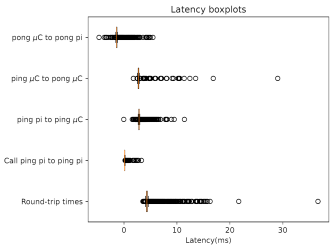
(e) ping μC to pong μC (3 → 4)



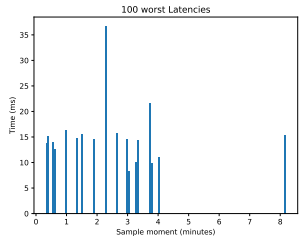
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



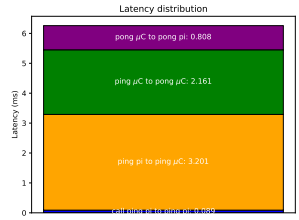
(h) Latency boxplots



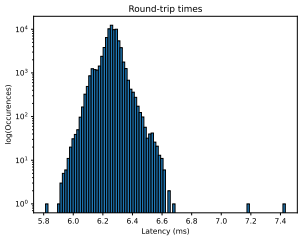
(i) 100 worst latencies

Figure E.38: Raspberry Pi 4B with Zephyr Rasberry Pi Pico μC - default publisher and default subscriber

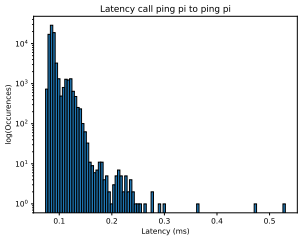
Zephyr Rasberry Pi Nucleo H743ZI μC - best effort publisher and best effort subscriber



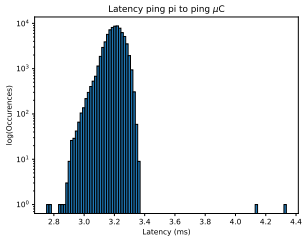
(a) Latency distribution



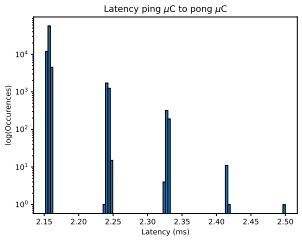
(b) Round-trip times



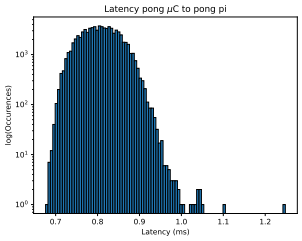
(c) Call ping pi to ping pi (1 → 2)



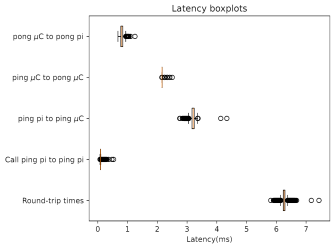
(d) ping pi to ping μC (2 → 3)



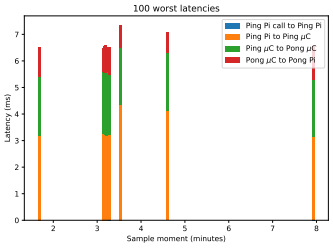
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



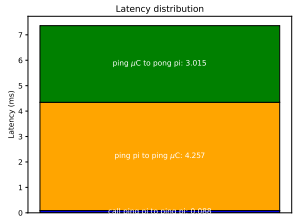
(g) Latency boxplots



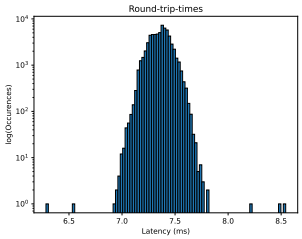
(h) 100 worst latencies

Figure E.39: Raspberry Pi 4B with Zephyr Nucleo H743ZI μC with - best effort publisher and best effort subscriber

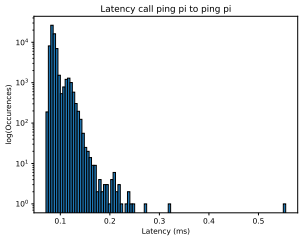
Zephyr Nucleo H743ZI μC - default publisher and best effort subscriber



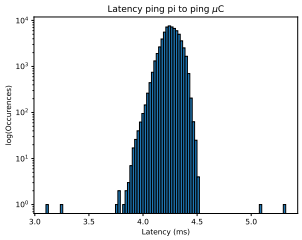
(a) Latency distribution



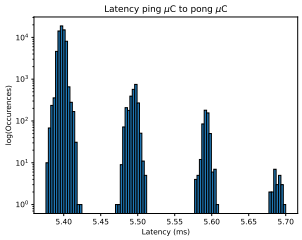
(b) Round-trip times



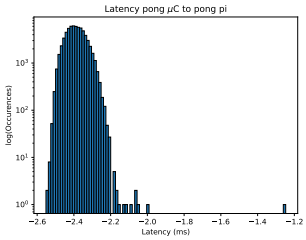
(c) Call ping pi to ping pi (1 → 2)



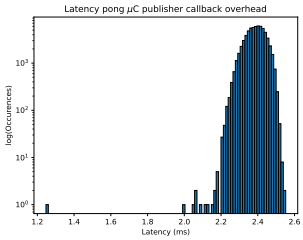
(d) ping pi to ping μC (2 → 3)



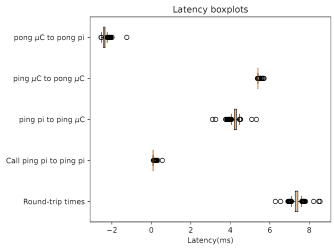
(e) ping μC to pong μC (3 → 4)



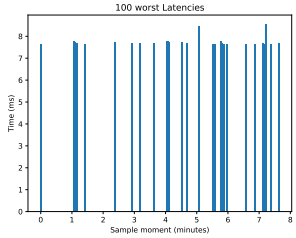
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



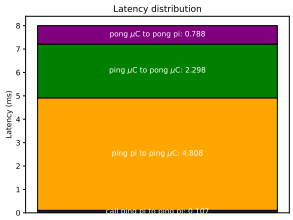
(h) Latency boxplots



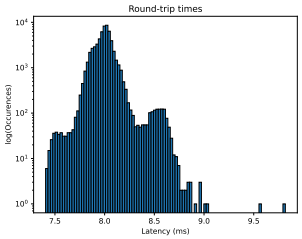
(i) 100 worst latencies

Figure E.40: Raspberry Pi 4B with Zephyr Nucleo H743ZI μC - default publisher and best effort subscriber

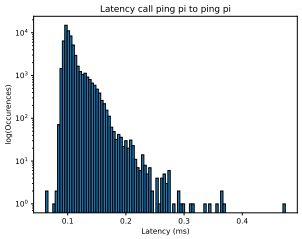
Zephyr Nucleo H743ZI μC - best effort publisher and default subscriber



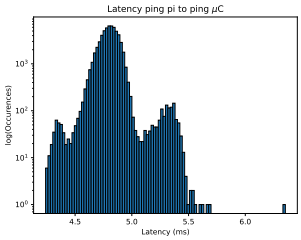
(a) Latency distribution



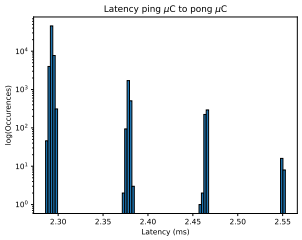
(b) Round-trip times



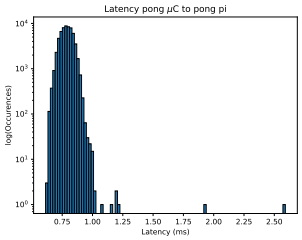
(c) Call ping pi to ping pi (1 → 2)



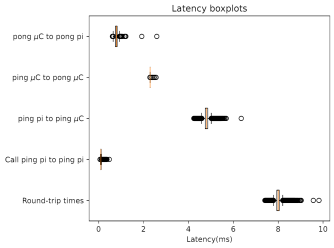
(d) ping pi to ping μC (2 → 3)



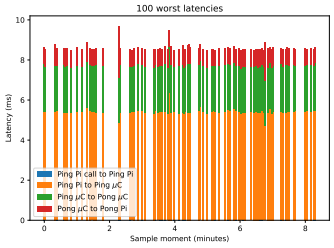
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



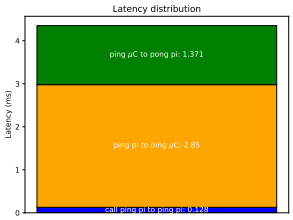
(g) Latency boxplots



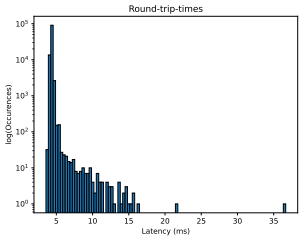
(h) 100 worst latencies

Figure E.41: Raspberry Pi 4B with Zephyr Nucleo H743ZI μC - best effort publisher and default subscriber

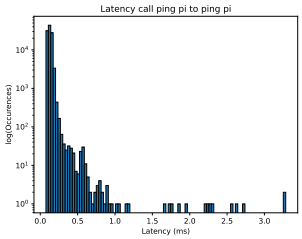
Zephyr Rasberry Pi Pico μC - default publisher and default subscriber



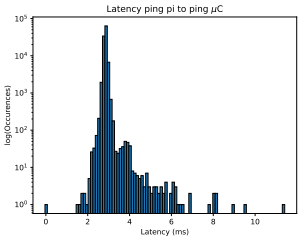
(a) Latency distribution



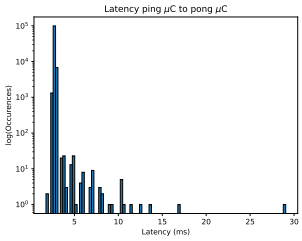
(b) Round-trip times



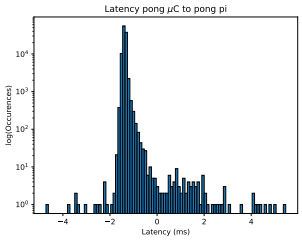
(c) Call ping pi to ping pi (1 → 2)



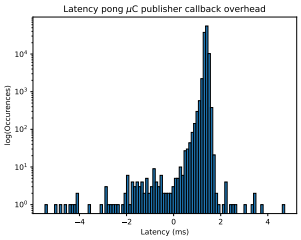
(d) ping pi to ping μC (2 → 3)



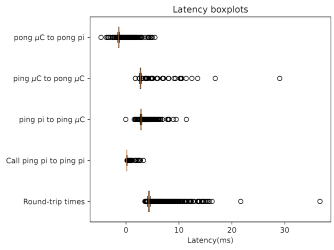
(e) ping μC to pong μC (3 → 4)



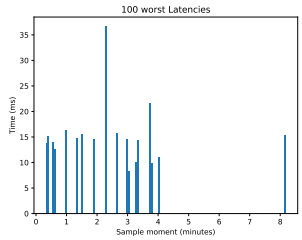
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



(h) Latency boxplots

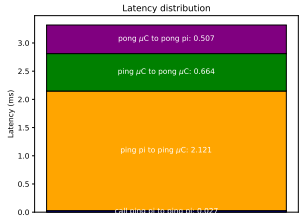


(i) 100 worst latencies

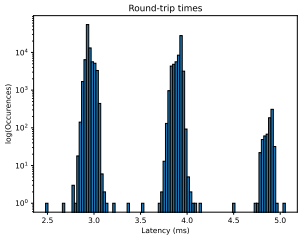
Figure E.42: Raspberry Pi 4B with Zephyr Rasberry Pi Pico μC - default publisher and default subscriber

E.4.2 Raspberry Pi 5

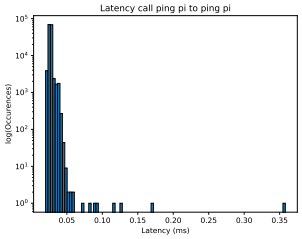
Zephyr Rasberry Pi Pico μC - best effort publisher and best effort subscriber



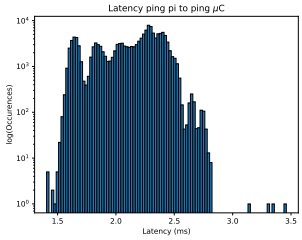
(a) Latency distribution



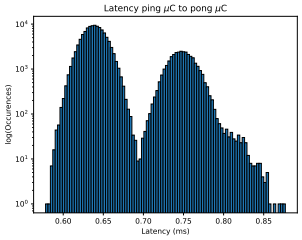
(b) Round-trip times



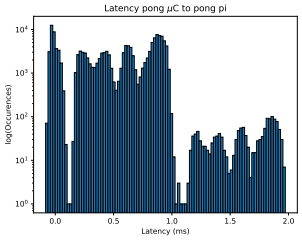
(c) Call ping pi to ping pi (1 → 2)



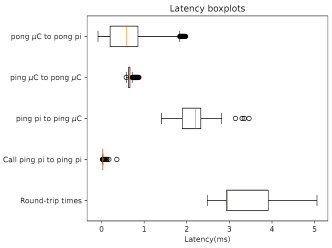
(d) ping pi to ping μC (2 → 3)



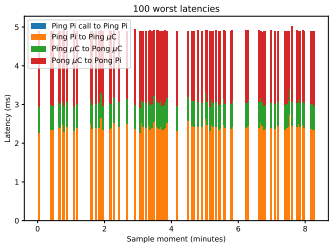
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



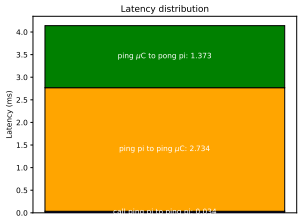
(g) Latency boxplots



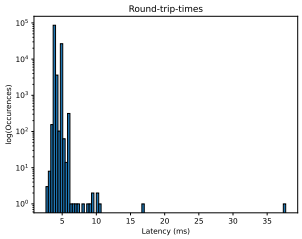
(h) 100 worst latencies

Figure E.43: Raspberry Pi 5 with Zephyr Rasberry Pi Pico μC with - best effort publisher and best effort subscriber

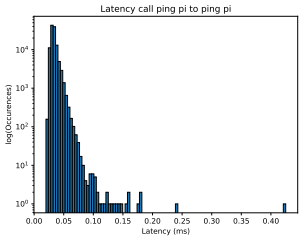
Raspberry Pi 5 with Zephyr Rasberry Pi Pico μC - default publisher and best effort subscriber



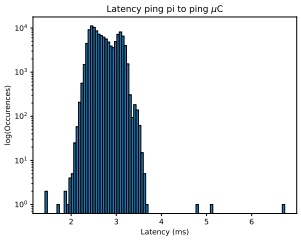
(a) Latency distribution



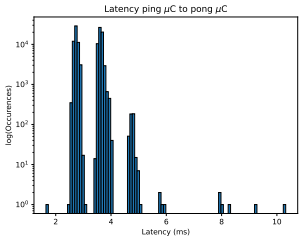
(b) Round-trip times



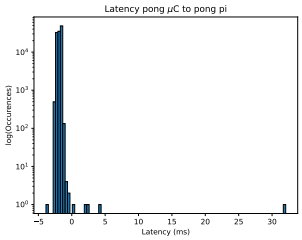
(c) Call ping pi to ping pi (1 → 2)



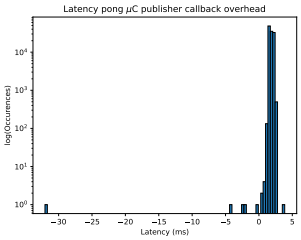
(d) ping pi to ping μC (2 → 3)



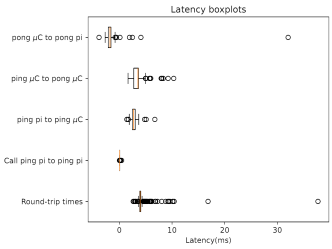
(e) ping μC to pong μC (3 → 4)



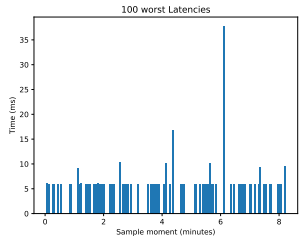
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



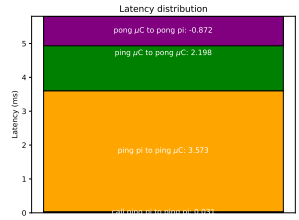
(h) Latency boxplots



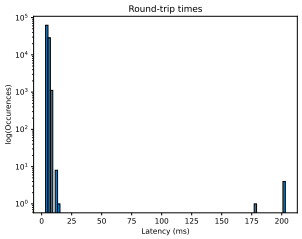
(i) 100 worst latencies

Figure E.44: Raspberry Pi 5 with Zephyr Rasberry Pi Pico μC - default publisher and best effort subscriber

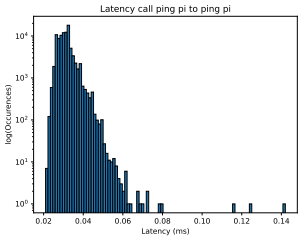
Zephyr Rasberry Pi Pico μC - best effort publisher and default subscriber



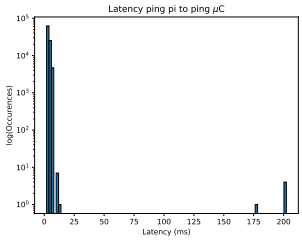
(a) Latency distribution



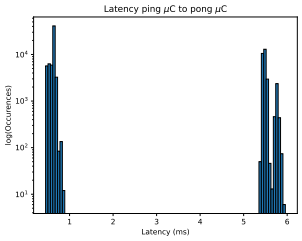
(b) Round-trip times



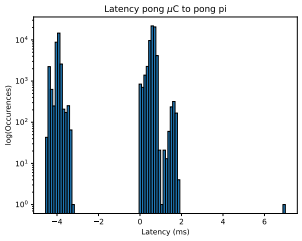
(c) Call ping pi to ping pi (1 → 2)



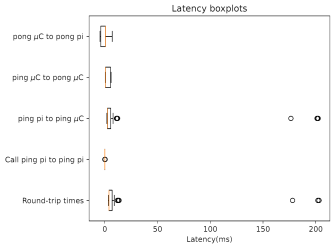
(d) ping pi to ping μC (2 → 3)



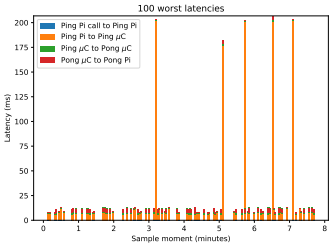
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



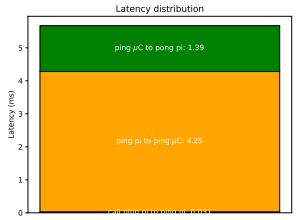
(g) Latency boxplots



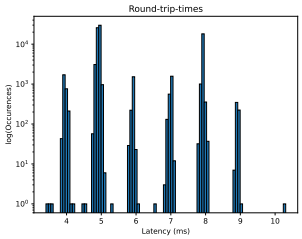
(h) 100 worst latencies

Figure E.45: Raspberry Pi 5 with Zephyr Rasberry Pi Pico μC - best effort publisher and default subscriber

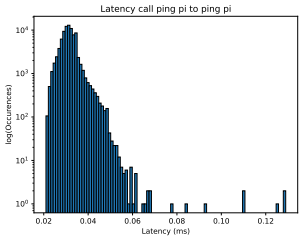
Zephyr Rasberry Pi Pico μC - default publisher and default subscriber



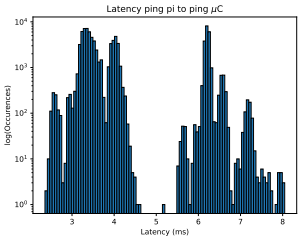
(a) Latency distribution



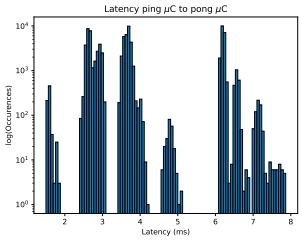
(b) Round-trip times



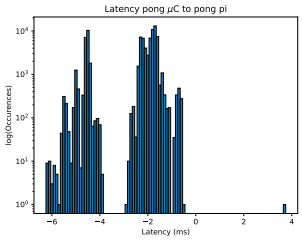
(c) Call ping pi to ping pi (1 → 2)



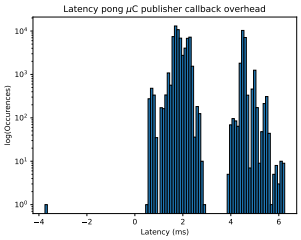
(d) ping pi to ping μC (2 → 3)



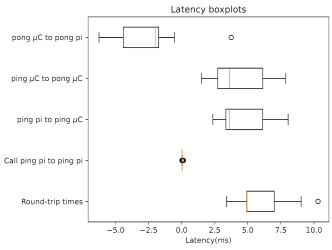
(e) ping μC to pong μC (3 → 4)



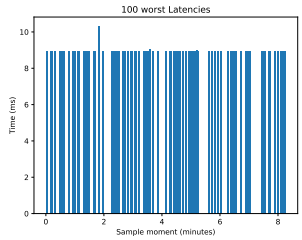
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



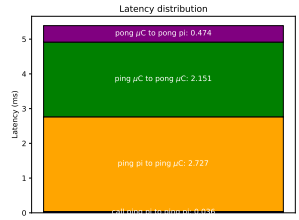
(h) Latency boxplots



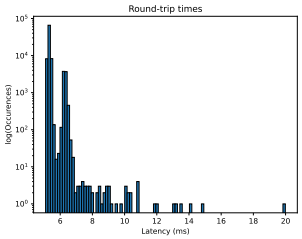
(i) 100 worst latencies

Figure E.46: Raspberry Pi 5 with Zephyr Rasberry Pi Pico μC - default publisher and default subscriber

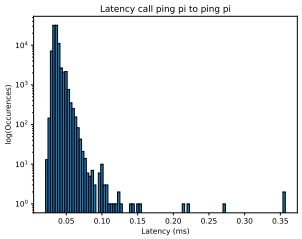
Zephyr Nucleo H743ZI μC - best effort publisher and best effort subscriber



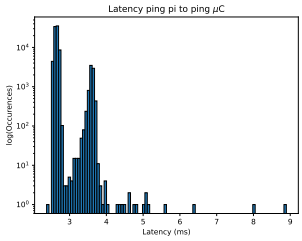
(a) Latency distribution



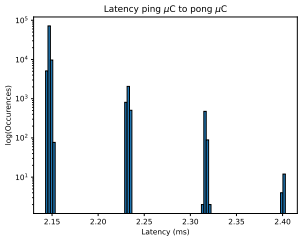
(b) Round-trip times



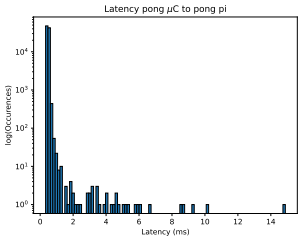
(c) Call ping pi to ping pi (1 → 2)



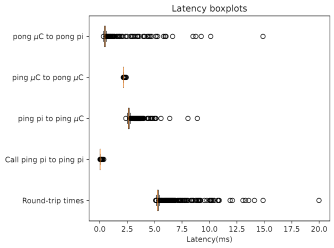
(d) ping pi to ping μC (2 → 3)



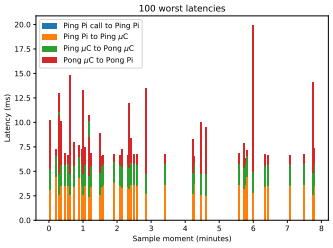
(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)



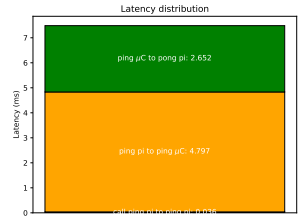
(g) Latency boxplots



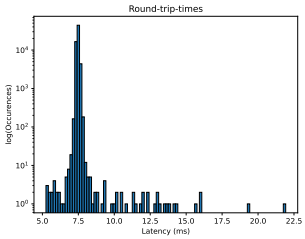
(h) 100 worst latencies

Figure E.47: Raspberry Pi 5 with Zephyr Nucleo H743ZI μC with - best effort publisher and best effort subscriber

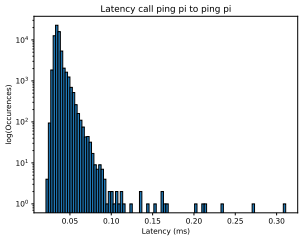
Raspberry Pi 5 with Zephyr Nucleo H743ZI μC - default publisher and best effort subscriber



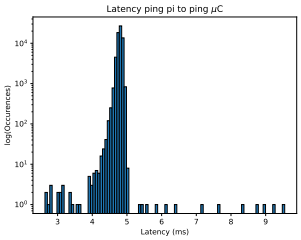
(a) Latency distribution



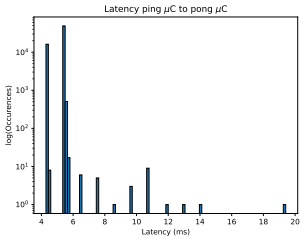
(b) Round-trip times



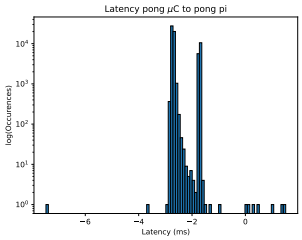
(c) Call ping pi to ping pi (1 → 2)



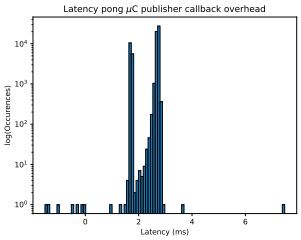
(d) ping pi to ping μC (2 → 3)



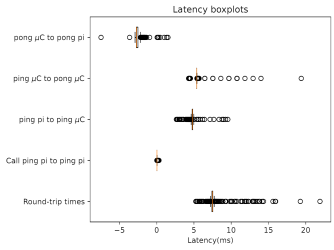
(e) ping μC to pong μC (3 → 4)



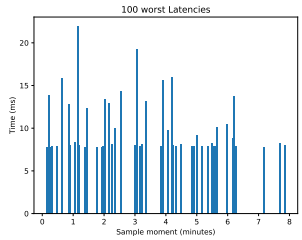
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



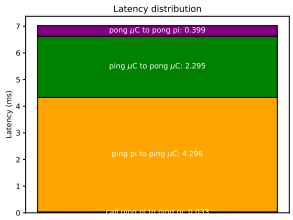
(h) Latency boxplots



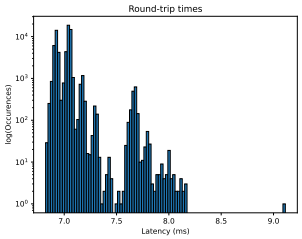
(i) 100 worst latencies

Figure E.48: Raspberry Pi 5 with Zephyr Nucleo H743ZI μC - default publisher and best effort subscriber

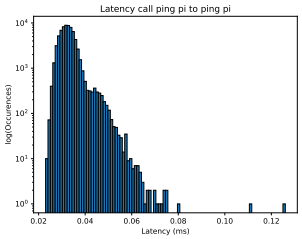
Zephyr Nucleo H743ZI μC - best effort publisher and default subscriber



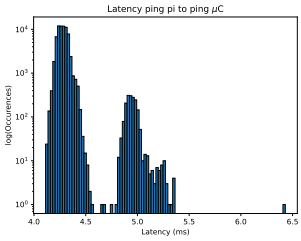
(a) Latency distribution



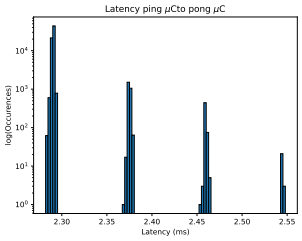
(b) Round-trip times



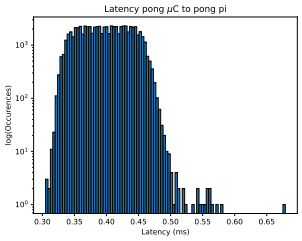
(c) Call ping pi to ping pi (1 → 2)



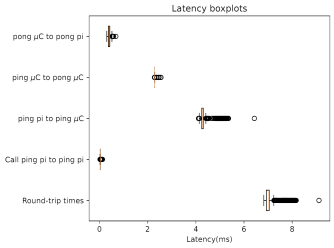
(d) ping pi to ping μC (2 → 3)



(e) ping μC to pong μC (3 → 4)



(f) pong μC to pong pi (4 → 5)

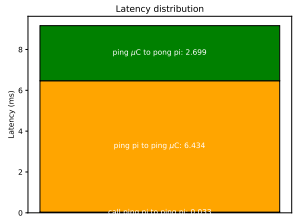


(g) Latency boxplots

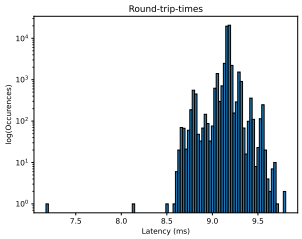
(h) 100 worst latencies

Figure E.49: Raspberry Pi 5 with Zephyr Nucleo H743ZI μC - best effort publisher and default subscriber

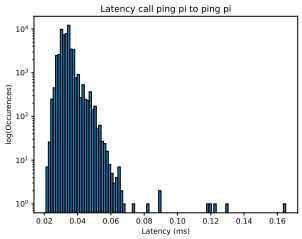
Zephyr Nucleo H743ZI μC - default publisher and default subscriber



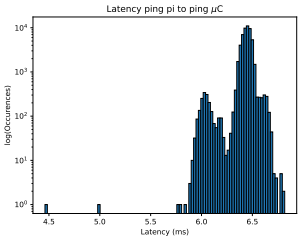
(a) Latency distribution



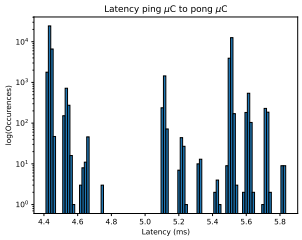
(b) Round-trip times



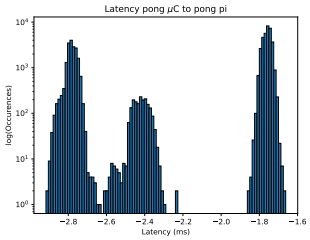
(c) Call ping pi to ping pi (1 → 2)



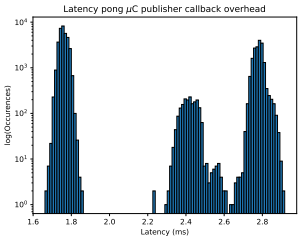
(d) ping pi to ping μC (2 → 3)



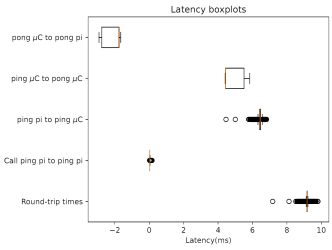
(e) ping μC to pong μC (3 → 4)



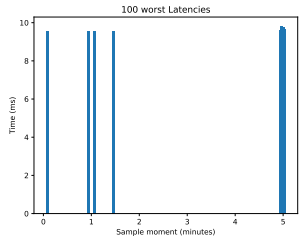
(f) pong μC to pong pi (4 → 5)



(g) Callback overhead



(h) Latency boxplots



(i) 100 worst latencies

Figure E.50: Raspberry Pi 5 with Zephyr Nucleo H743ZI μC - default publisher and default subscriber

F JIWY control loop measurements

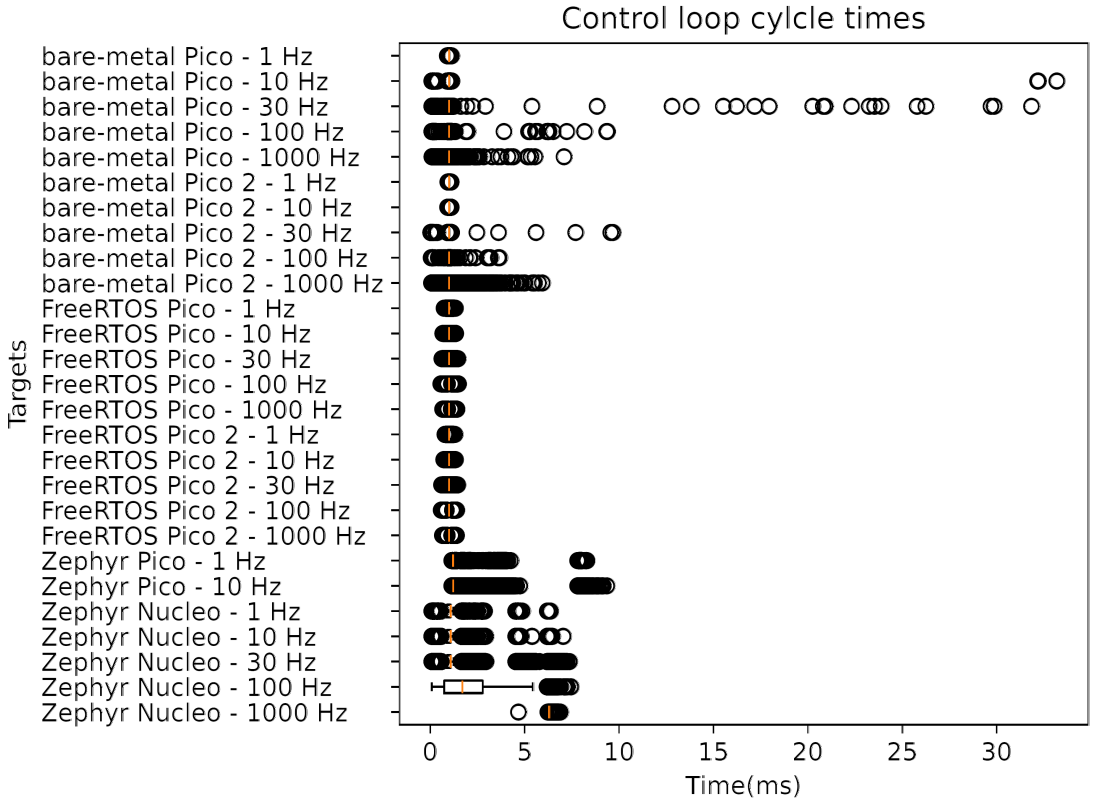


Figure F.1: Control loop cycle times complete

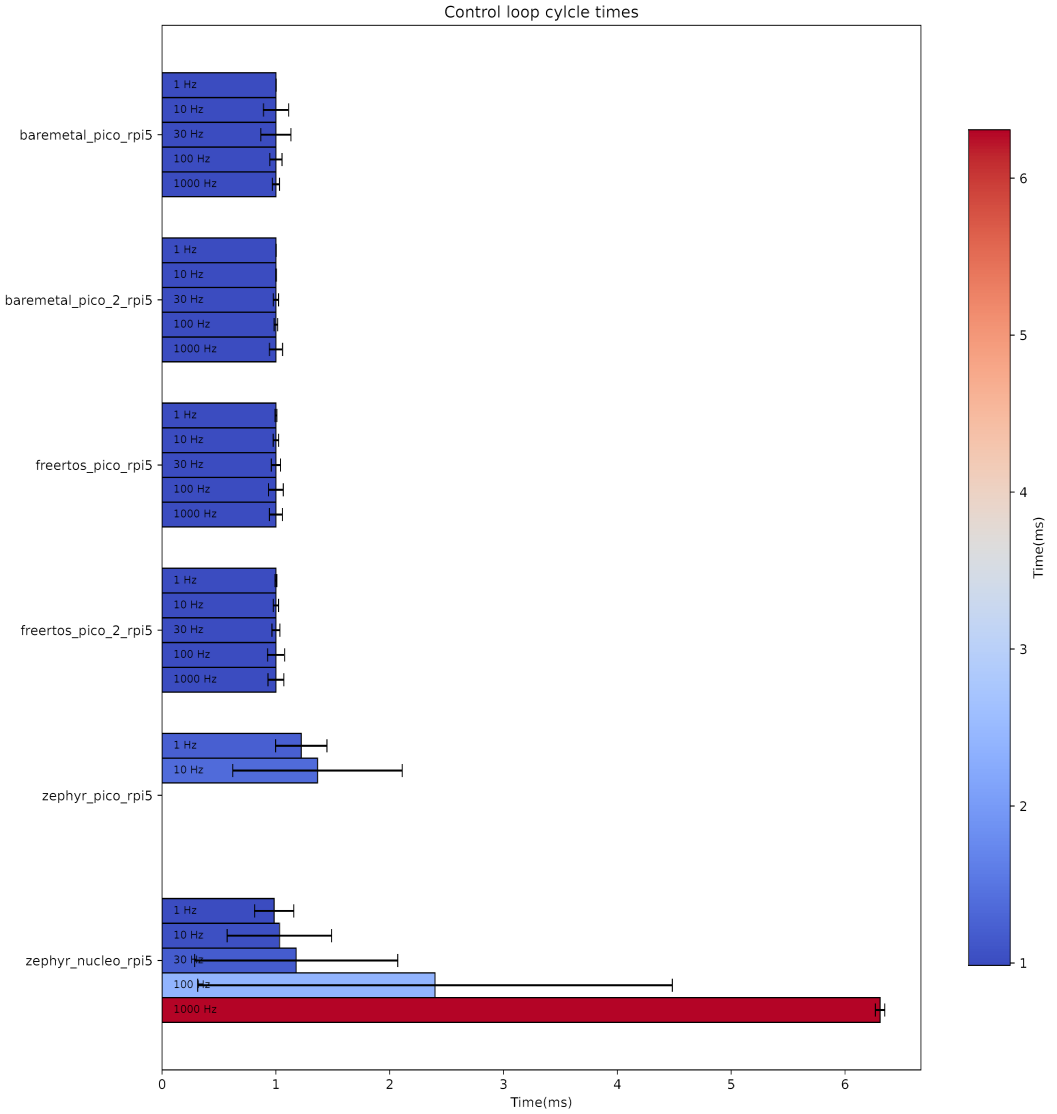
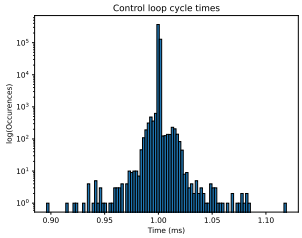


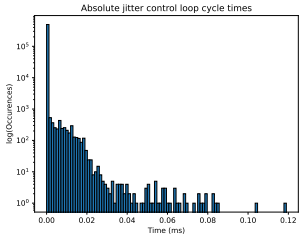
Figure F.2: Mean Control loop cycle times

F.1 Bare-metal control loop measurements

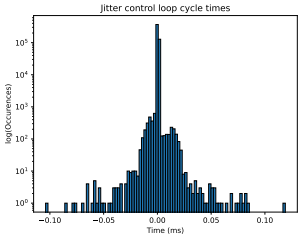
F.1.1 Control loop measurements bare-metal Rasberry Pi Pico at 1 Hz



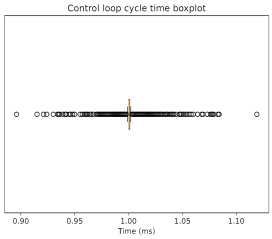
(a) Control loop cycle times



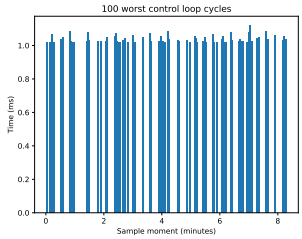
(b) Abs jitter cycle times



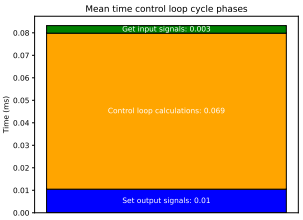
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



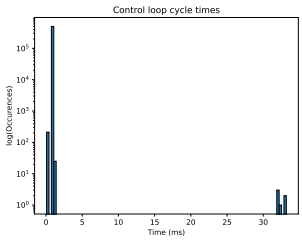
(e) 100 worst control loop cycles



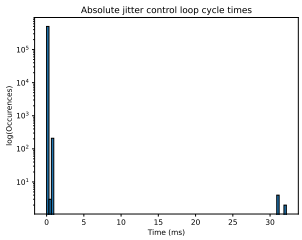
(f) Mean time control loop phases

Figure F.3: Bare-metal Rasberry Pi Pico at 1 Hz

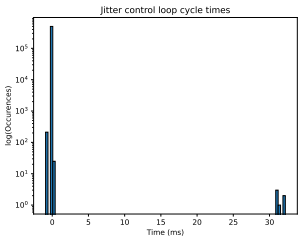
F.1.2 Control loop measurements bare-metal Rasberry Pi Pico at 10 Hz



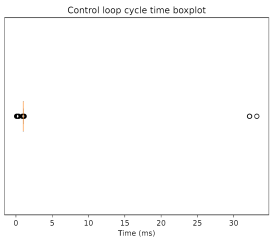
(a) Control loop cycle times



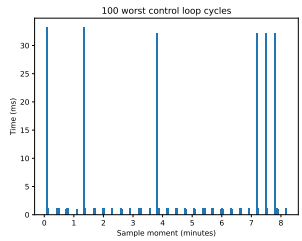
(b) Abs jitter cycle times



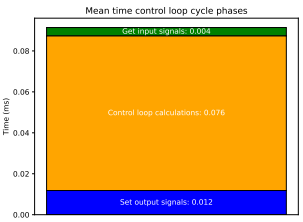
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



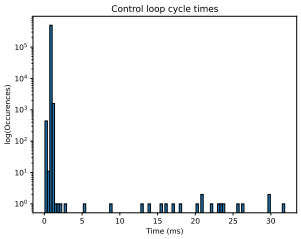
(e) 100 worst control loop cycles



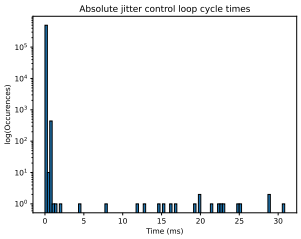
(f) Mean time control loop phases

Figure F.4: Bare-metal Rasberry Pi Pico at 10 Hz

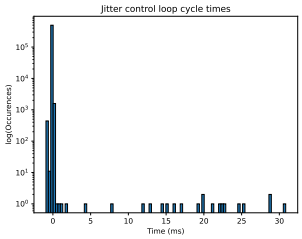
F.1.3 Control loop measurements bare-metal Rasberry Pi Pico at 30 Hz



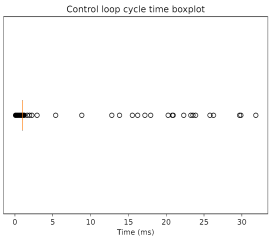
(a) Control loop cycle times



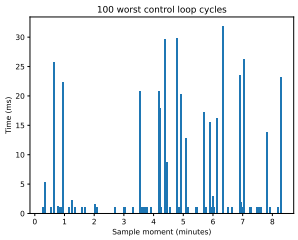
(b) Abs jitter cycle times



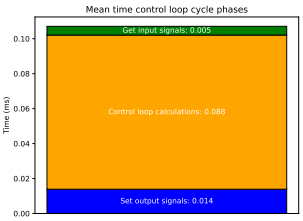
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



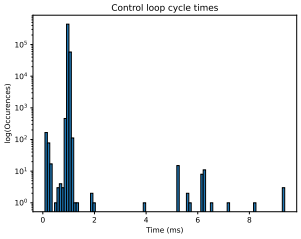
(e) 100 worst control loop cycles



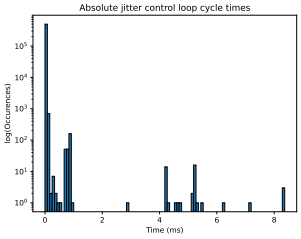
(f) Mean time control loop phases

Figure F.5: Bare-metal Rasberry Pi Pico at 30 Hz

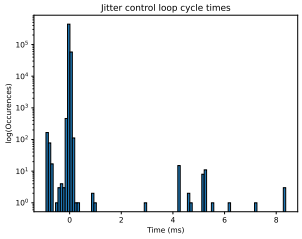
F.1.4 Control loop measurements bare-metal Rasberry Pi Pico at 100 Hz



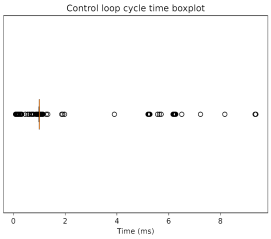
(a) Control loop cycle times



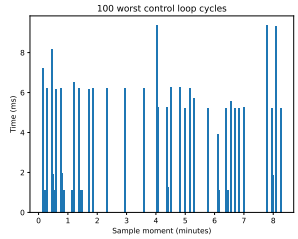
(b) Abs jitter cycle times



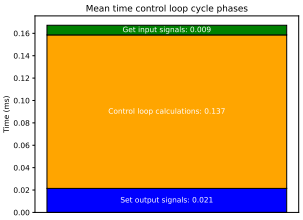
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



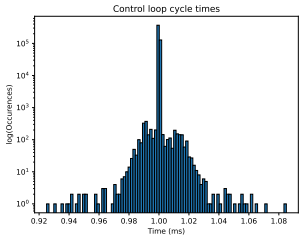
(e) 100 worst control loop cycles



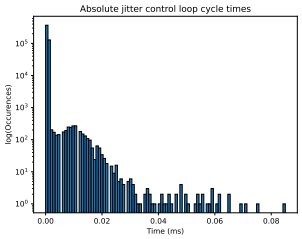
(f) Mean time control loop phases

Figure F.6: Bare-metal Rasberry Pi Pico at 100 Hz

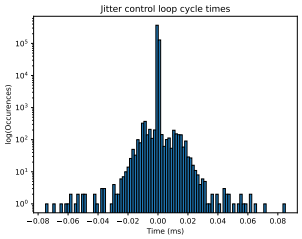
F.1.5 Control loop measurements bare-metal Rasberry Pi Pico 2 at 1 Hz



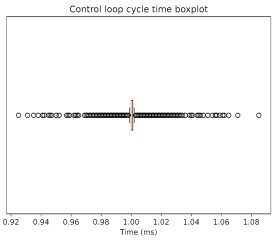
(a) Control loop cycle times



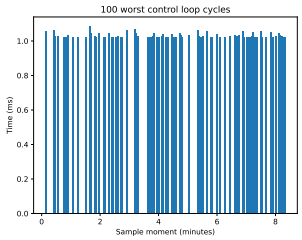
(b) Abs jitter cycle times



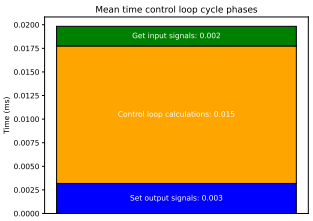
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



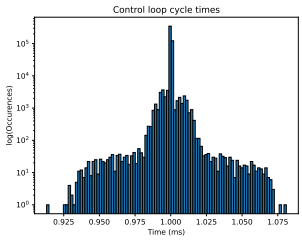
(e) 100 worst control loop cycles



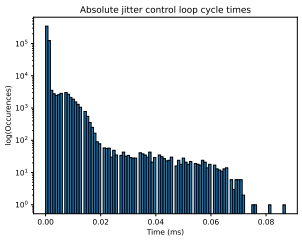
(f) Mean time control loop phases

Figure F.7: Bare-metal Rasberry Pi Pico 2 at 1 Hz

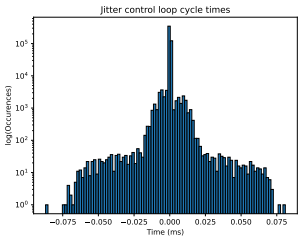
F.1.6 Control loop measurements bare-metal Rasberry Pi Pico 2 at 10 Hz



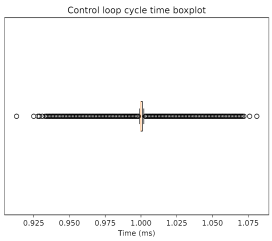
(a) Control loop cycle times



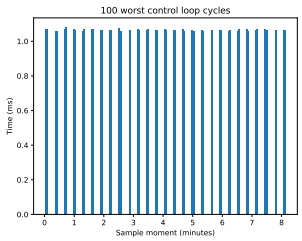
(b) Abs jitter cycle times



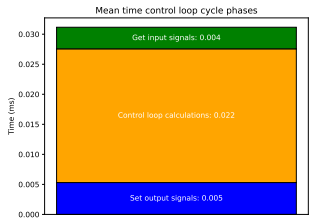
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



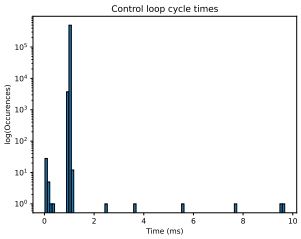
(e) 100 worst control loop cycles



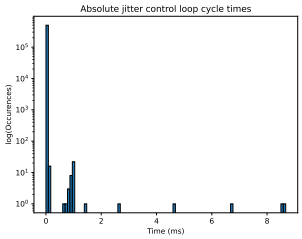
(f) Mean time control loop phases

Figure F.8: Bare-metal Rasberry Pi Pico 2 at 10 Hz

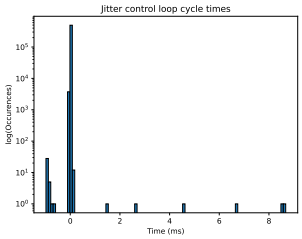
F.1.7 Control loop measurements bare-metal Rasberry Pi Pico 2 at 30 Hz



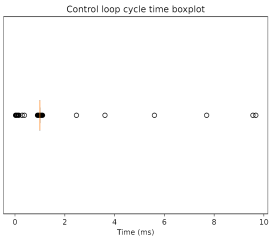
(a) Control loop cycle times



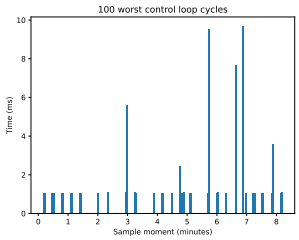
(b) Abs jitter cycle times



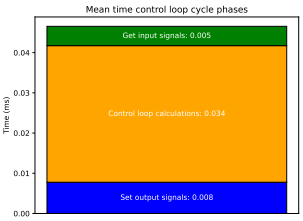
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



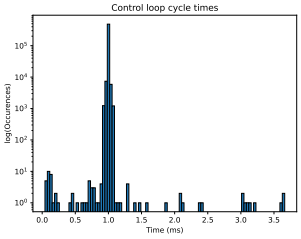
(e) 100 worst control loop cycles



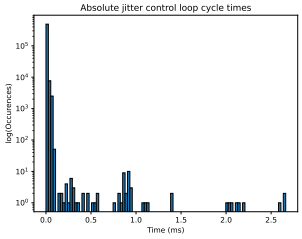
(f) Mean time control loop phases

Figure F.9: Bare-metal Rasberry Pi Pico 2 at 30 Hz

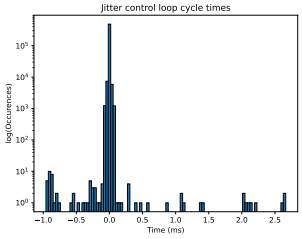
F.1.8 Control loop measurements bare-metal Rasberry Pi Pico 2 at 100 Hz



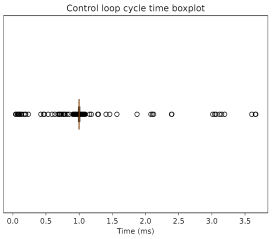
(a) Control loop cycle times



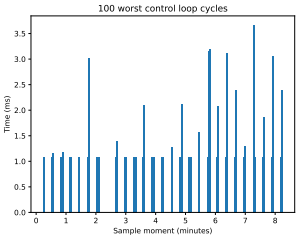
(b) Abs jitter cycle times



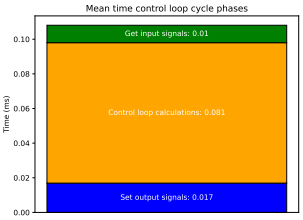
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



(e) 100 worst control loop cycles

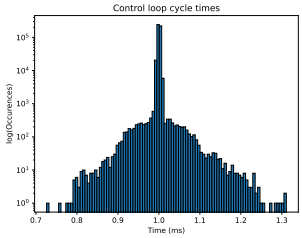


(f) Mean time control loop phases

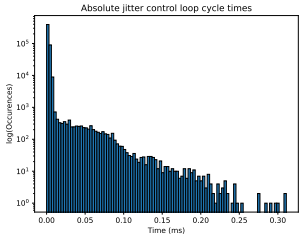
Figure F.10: Bare-metal Rasberry Pi Pico 2 at 100 Hz

F.2 FreeRTOS control loop measurements

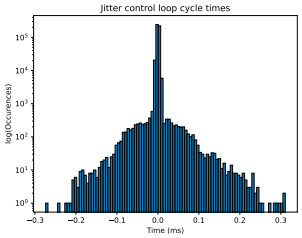
F.2.1 Control loop measurements FreeRTOS Rasberry Pi Pico at 1 Hz



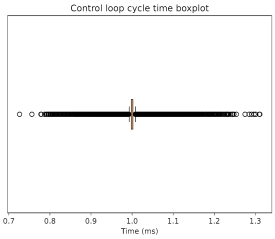
(a) Control loop cycle times



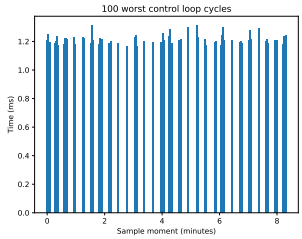
(b) Abs jitter cycle times



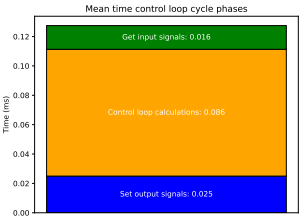
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



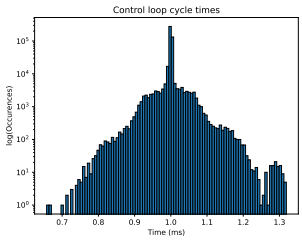
(e) 100 worst control loop cycles



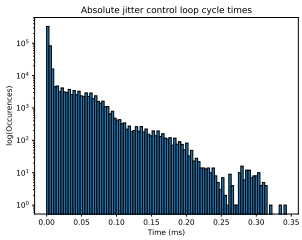
(f) Mean time control loop phases

Figure F.11: FreeRTOS Rasberry Pi Pico at 1 Hz

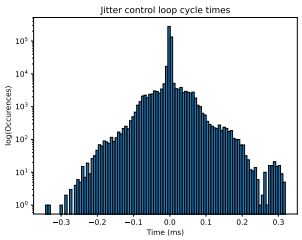
F.2.2 Control loop measurements FreeRTOS Rasberry Pi Pico at 10 Hz



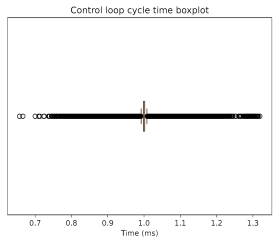
(a) Control loop cycle times



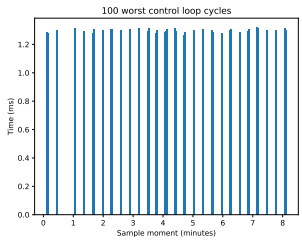
(b) Abs jitter cycle times



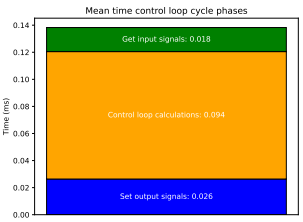
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



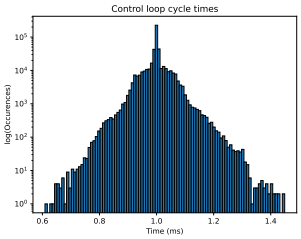
(e) 100 worst control loop cycles



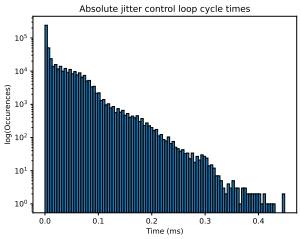
(f) Mean time control loop phases

Figure F.12: FreeRTOS Rasberry Pi Pico at 10 Hz

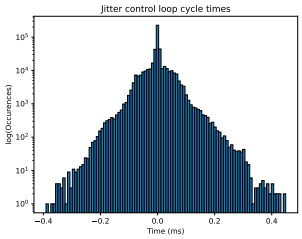
F.2.3 Control loop measurements FreeRTOS Rasberry Pi Pico at 30 Hz



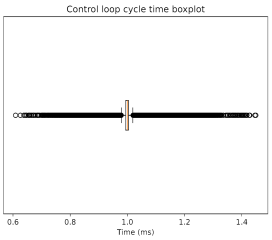
(a) Control loop cycle times



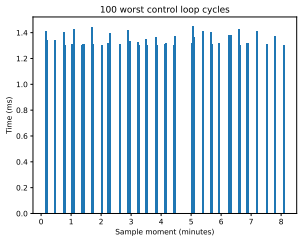
(b) Abs jitter cycle times



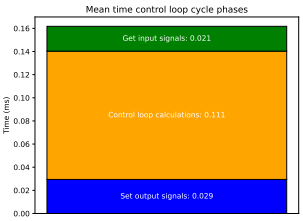
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



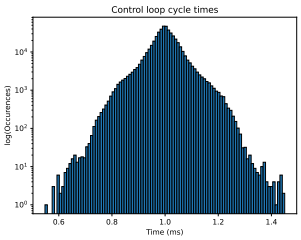
(e) 100 worst control loop cycles



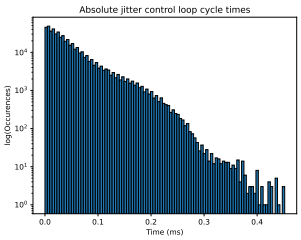
(f) Mean time control loop phases

Figure F.13: FreeRTOS Rasberry Pi Pico at 30 Hz

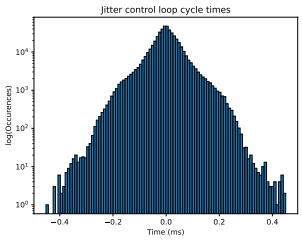
F.2.4 Control loop measurements FreeRTOS Rasberry Pi Pico at 100 Hz



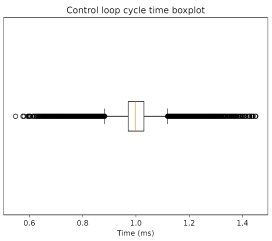
(a) Control loop cycle times



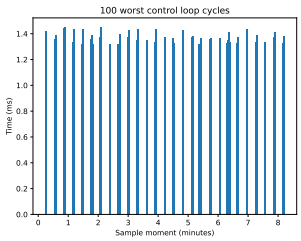
(b) Abs jitter cycle times



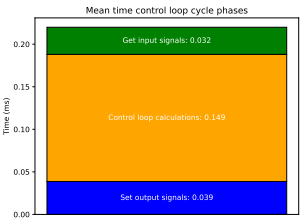
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



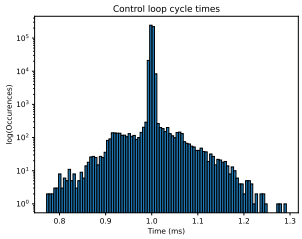
(e) 100 worst control loop cycles



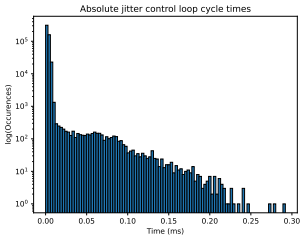
(f) Mean time control loop phases

Figure F.14: FreeRTOS Rasberry Pi Pico at 100 Hz

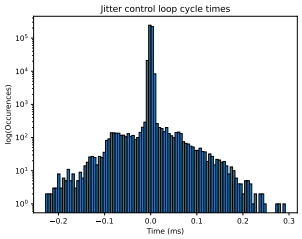
F.2.5 Control loop measurements FreeRTOS Rasberry Pi Pico 2 at 1 Hz



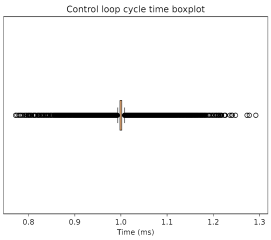
(a) Control loop cycle times



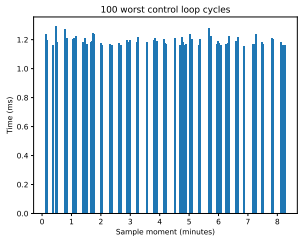
(b) Abs jitter cycle times



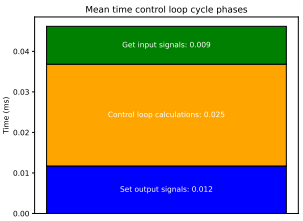
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



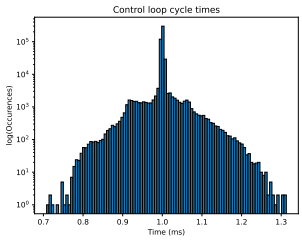
(e) 100 worst control loop cycles



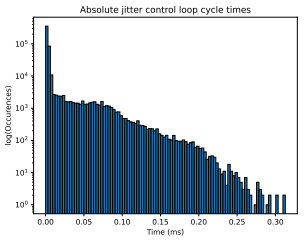
(f) Mean time control loop phases

Figure F.15: FreeRTOS Rasberry Pi Pico 2 at 1 Hz

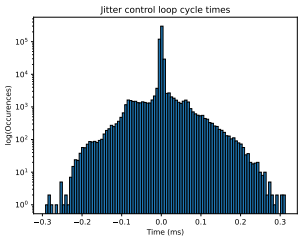
F.2.6 Control loop measurements FreeRTOS Rasberry Pi Pico 2 at 10 Hz



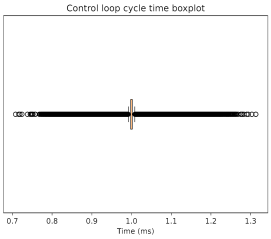
(a) Control loop cycle times



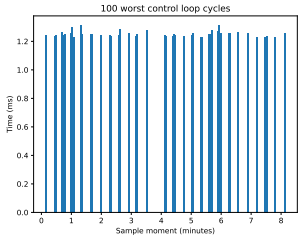
(b) Abs jitter cycle times



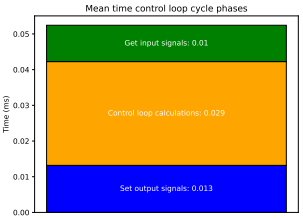
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



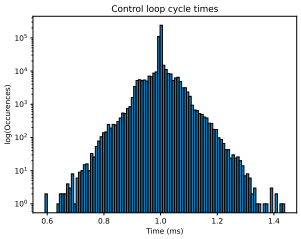
(e) 100 worst control loop cycles



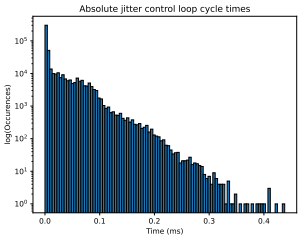
(f) Mean time control loop phases

Figure F.16: FreeRTOS Rasberry Pi Pico 2 at 10 Hz

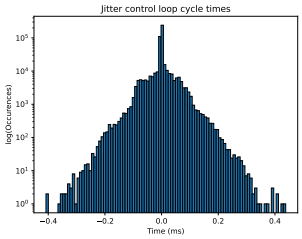
F.2.7 Control loop measurements FreeRTOS Rasberry Pi Pico 2 at 30 Hz



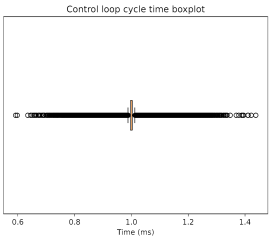
(a) Control loop cycle times



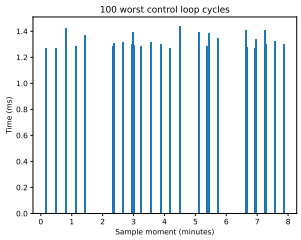
(b) Abs jitter cycle times



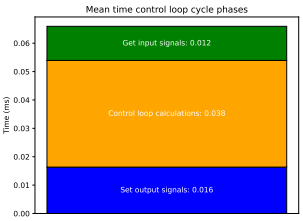
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



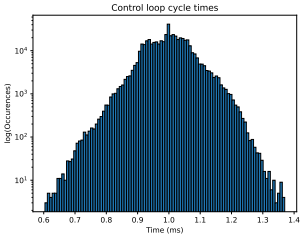
(e) 100 worst control loop cycles



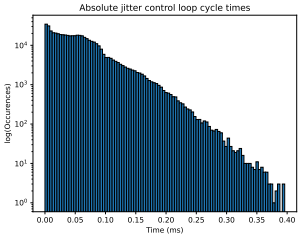
(f) Mean time control loop phases

Figure F.17: FreeRTOS Rasberry Pi Pico 2 at 30 Hz

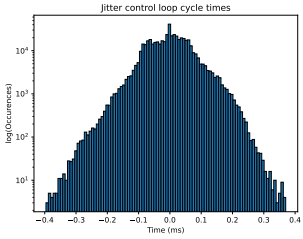
F.2.8 Control loop measurements FreeRTOS Rasberry Pi Pico 2 at 100 Hz



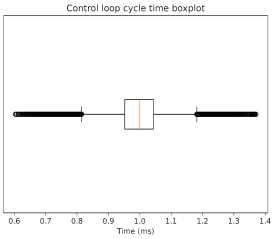
(a) Control loop cycle times



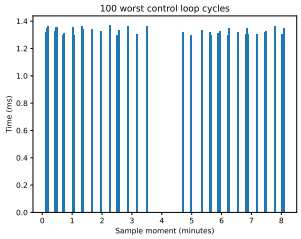
(b) Abs jitter cycle times



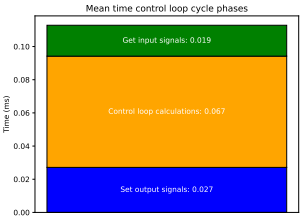
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



(e) 100 worst control loop cycles

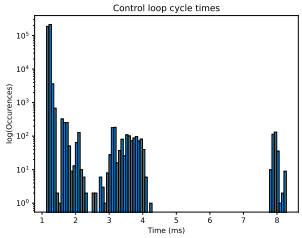


(f) Mean time control loop phases

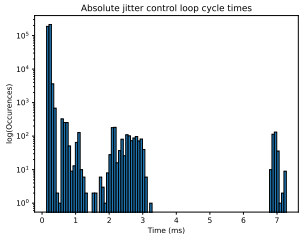
Figure F.18: FreeRTOS Rasberry Pi Pico 2 at 100 Hz

F.3 Zephyr control loop measurements

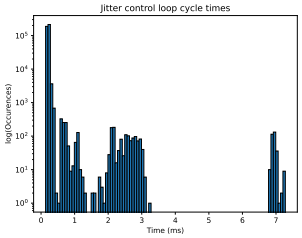
F.3.1 Control loop measurements Zephyr Rasberry Pi Pico at 1 Hz



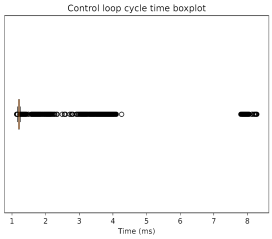
(a) Control loop cycle times



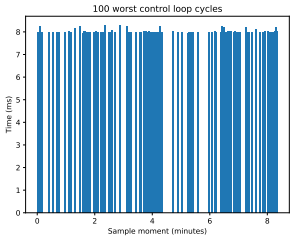
(b) Abs jitter cycle times



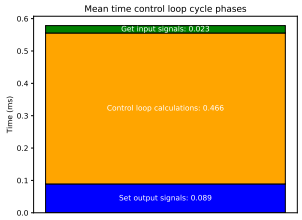
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



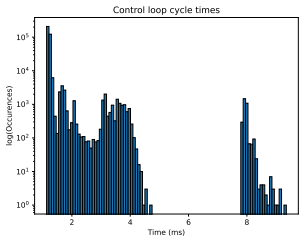
(e) 100 worst control loop cycles



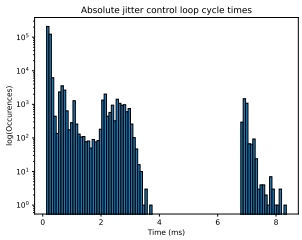
(f) Mean time control loop phases

Figure F.19: Control loop measurements Zephyr Rasberry Pi Pico at 1 Hz

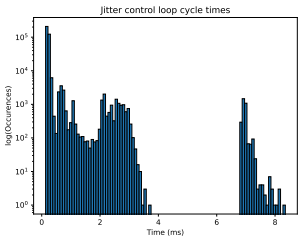
F.3.2 Control loop measurements Zephyr Rasberry Pi Pico at 10 Hz



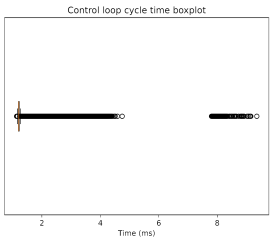
(a) Control loop cycle times



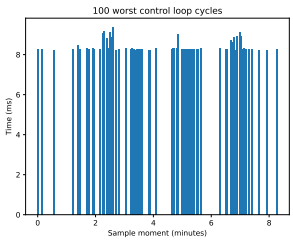
(b) Abs jitter cycle times



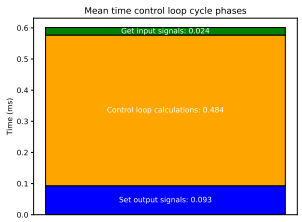
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



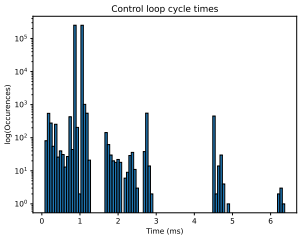
(e) 100 worst control loop cycles



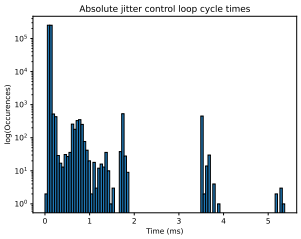
(f) Mean time control loop phases

Figure F.20: Control loop measurements Zephyr Rasberry Pi Pico at 10 Hz

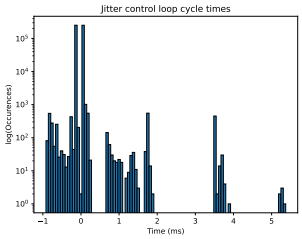
F.3.3 Control loop measurements Zephyr Nucleo H743ZI at 1 Hz



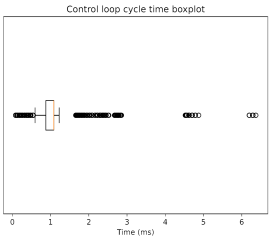
(a) Control loop cycle times



(b) Abs jitter cycle times

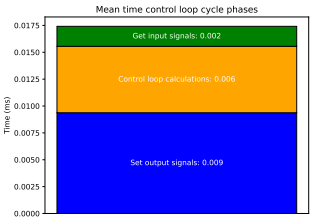


(c) Jitter control loop cycle times



(d) Control loop cycle boxplot

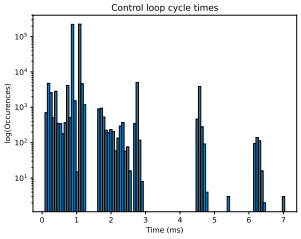
(e) 100 worst control loop cycles



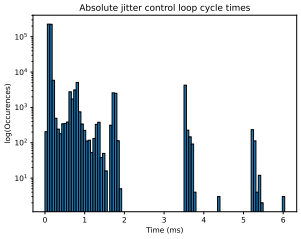
(f) Mean time control loop phases

Figure F.21: Control loop measurements Zephyr Nucleo H743ZI at 1 Hz

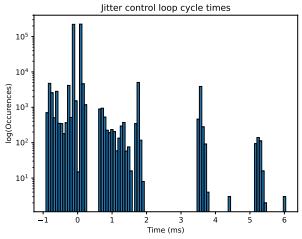
F.3.4 Control loop measurements Zephyr Nucleo H743ZI at 10 Hz



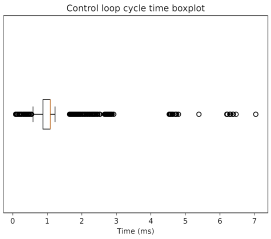
(a) Control loop cycle times



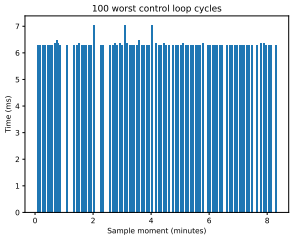
(b) Abs jitter cycle times



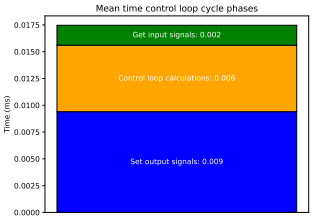
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



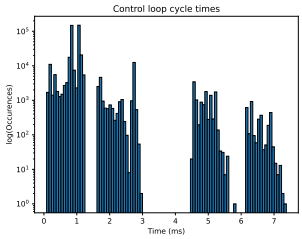
(e) 100 worst control loop cycles



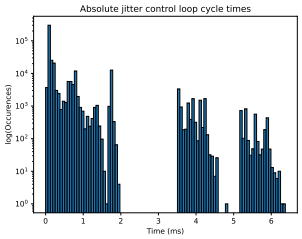
(f) Mean time control loop phases

Figure F.22: Control loop measurements Zephyr Nucleo H743ZI at 10 Hz

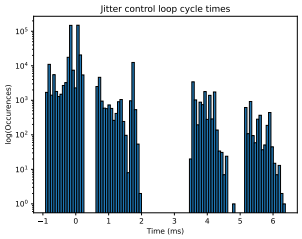
F.3.5 Control loop measurements Zephyr Nucleo H743ZI at 30 Hz



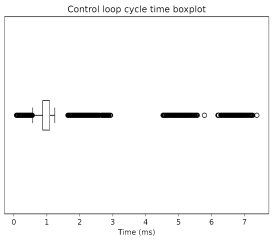
(a) Control loop cycle times



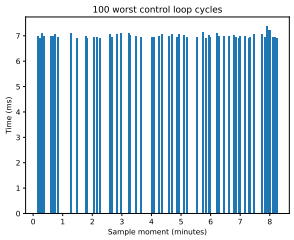
(b) Abs jitter cycle times



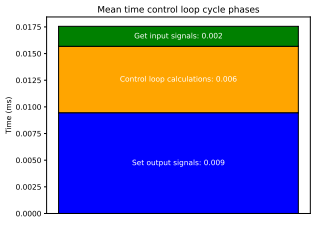
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



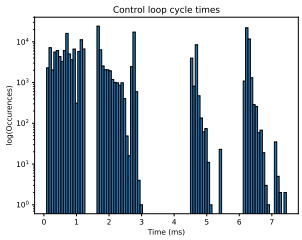
(e) 100 worst control loop cycles



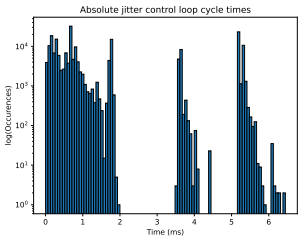
(f) Mean time control loop phases

Figure F.23: Control loop measurements Zephyr Nucleo H743ZI at 30 Hz

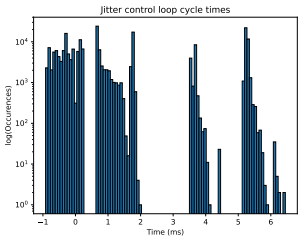
F.3.6 Control loop measurements Zephyr Nucleo H743ZI at 100 Hz



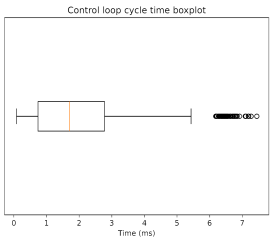
(a) Control loop cycle times



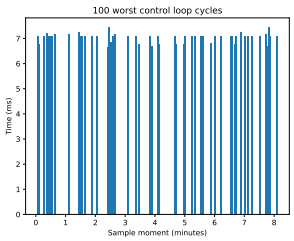
(b) Abs jitter cycle times



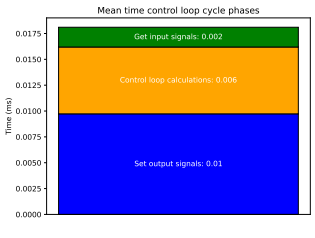
(c) Jitter control loop cycle times



(d) Control loop cycle boxplot



(e) 100 worst control loop cycles



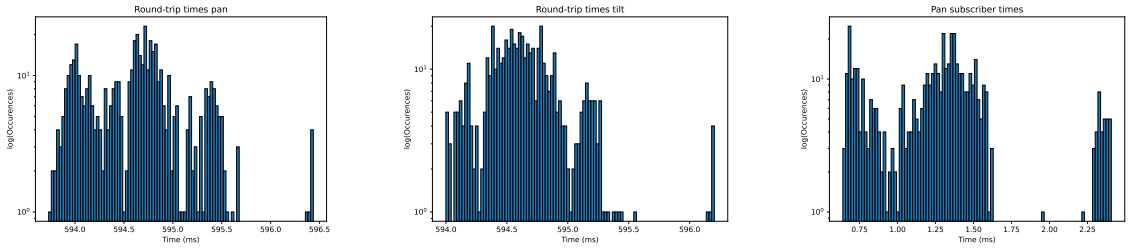
(f) Mean time control loop phases

Figure F.24: Control loop measurements Zephyr Nucleo H743ZI at 100 Hz

G JIWY communication measurements

G.1 Bare-metal

G.1.1 JIWY communication measurements bare-metal Rasberry Pi Pico 2 at 1 Hz

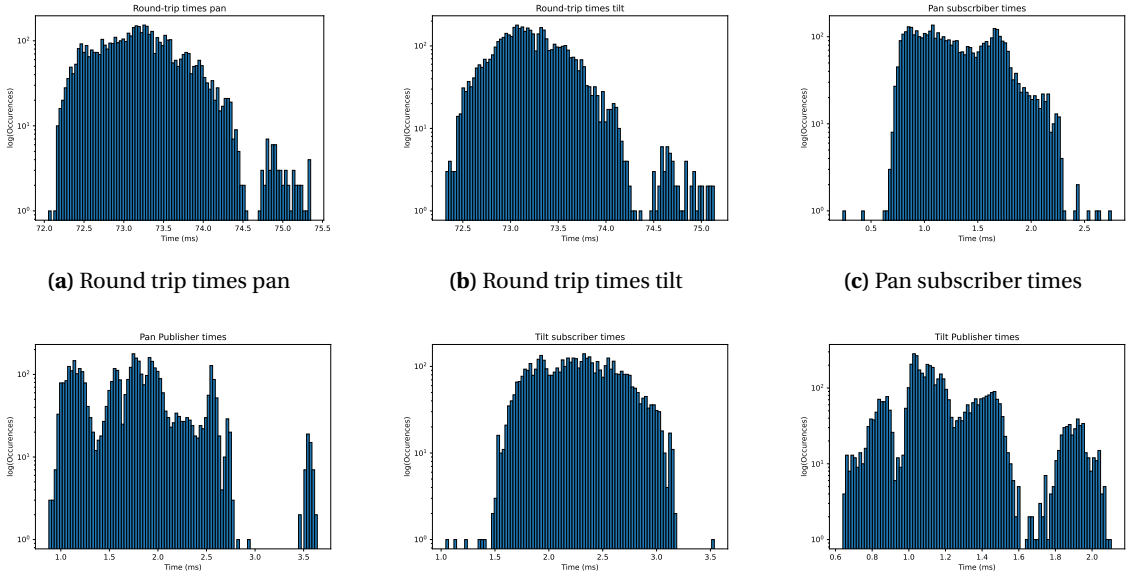


(a) Round trip times pan **(b)** Round trip times tilt **(c)** Pan subscriber times

(d) Pan publisher times **(e)** Tilt subscriber times **(f)** Tilt publisher times

Figure G.1: Bare-metal Rasberry Pi Pico 2 at 1 Hz

G.1.2 JIWY communication measurements bare-metal Rasberry Pi Pico 2 at 10 Hz

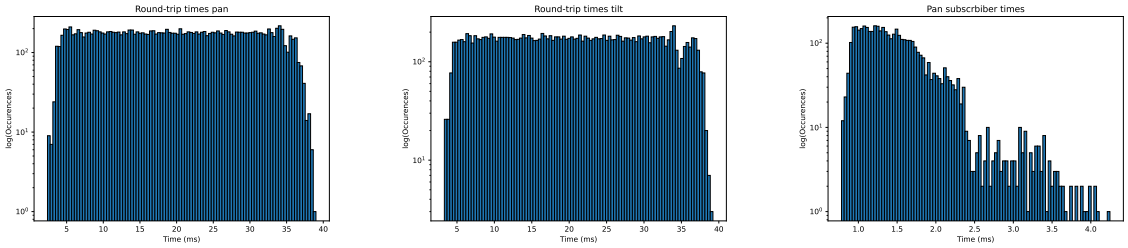


(a) Round trip times pan **(b)** Round trip times tilt **(c)** Pan subscriber times

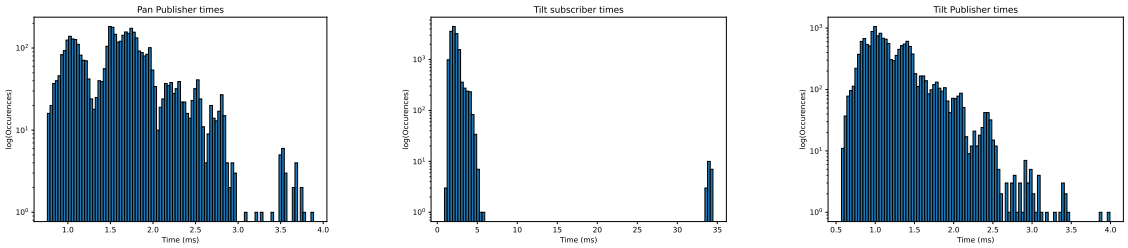
(d) Pan publisher times **(e)** Tilt subscriber times **(f)** Tilt publisher times

Figure G.2: Bare-metal Rasberry Pi Pico 2 at 10 Hz

G.1.3 JIWY communication measurements bare-metal Rasberry Pi Pico 2 at 30 Hz



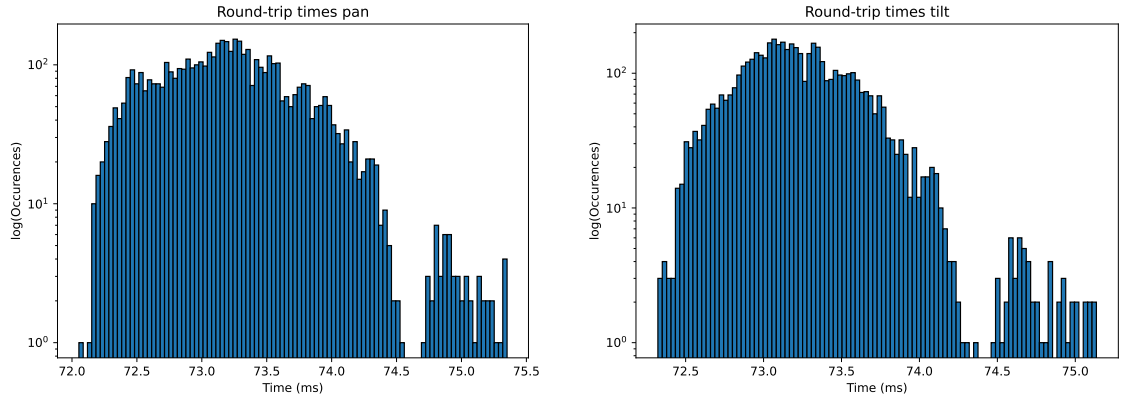
(a) Round trip times pan **(b)** Round trip times tilt **(c)** Pan subscriber times



(d) Pan publisher times **(e)** Tilt subscriber times **(f)** Tilt publisher times

Figure G.3: Bare-metal Rasberry Pi Pico 2 at 30 Hz

G.1.4 JIWY communication measurements bare-metal Rasberry Pi Pico 2 at 100 Hz

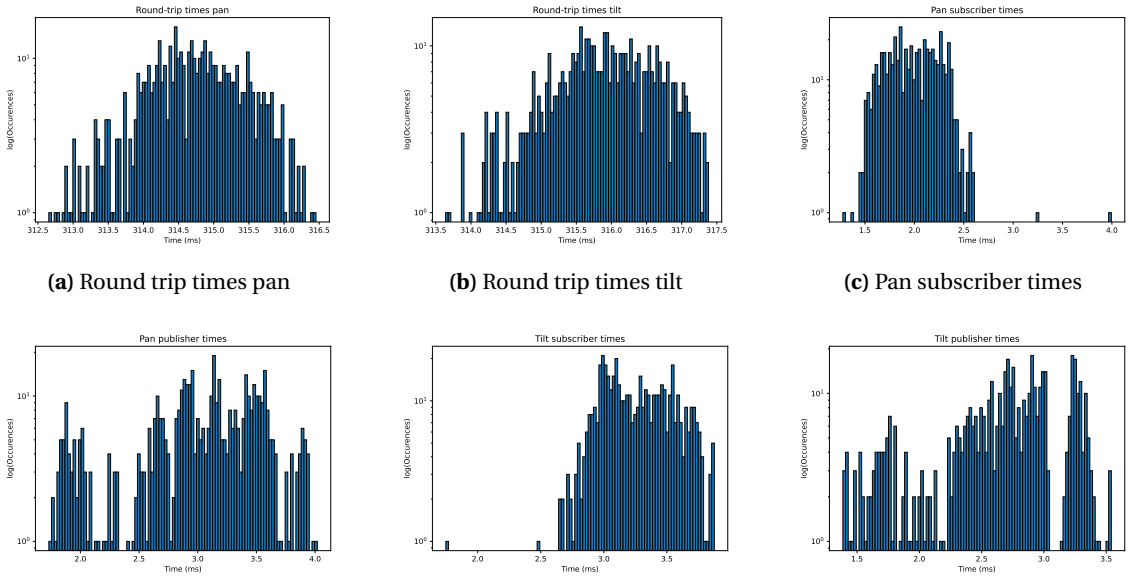


(a) Round trip times pan **(b)** Round trip times tilt

Figure G.4: Bare-metal Rasberry Pi Pico 2 at 100 Hz

G.2 FreeRTOS

G.2.1 JIWY communication measurements FreeRTOS Rasberry Pi Pico 2 at 1 Hz

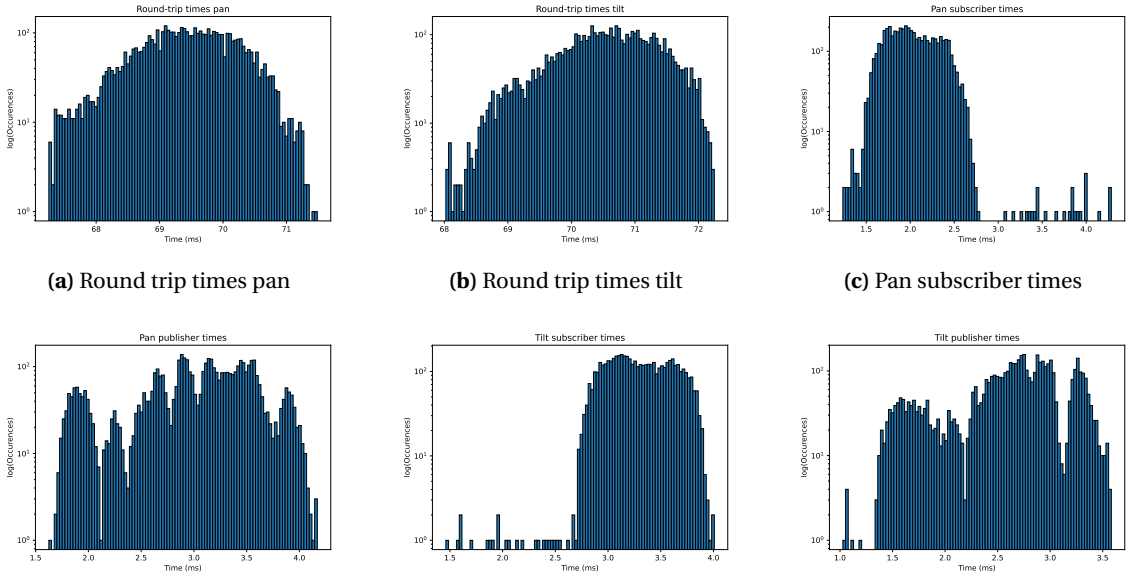


(a) Round trip times pan **(b)** Round trip times tilt **(c)** Pan subscriber times

(d) Pan publisher times **(e)** Tilt subscriber times **(f)** Tilt publisher times

Figure G.5: FreeRTOS Rasberry Pi Pico 2 at 1 Hz

G.2.2 JIWY communication measurements FreeRTOS Rasberry Pi Pico 2 at 10 Hz

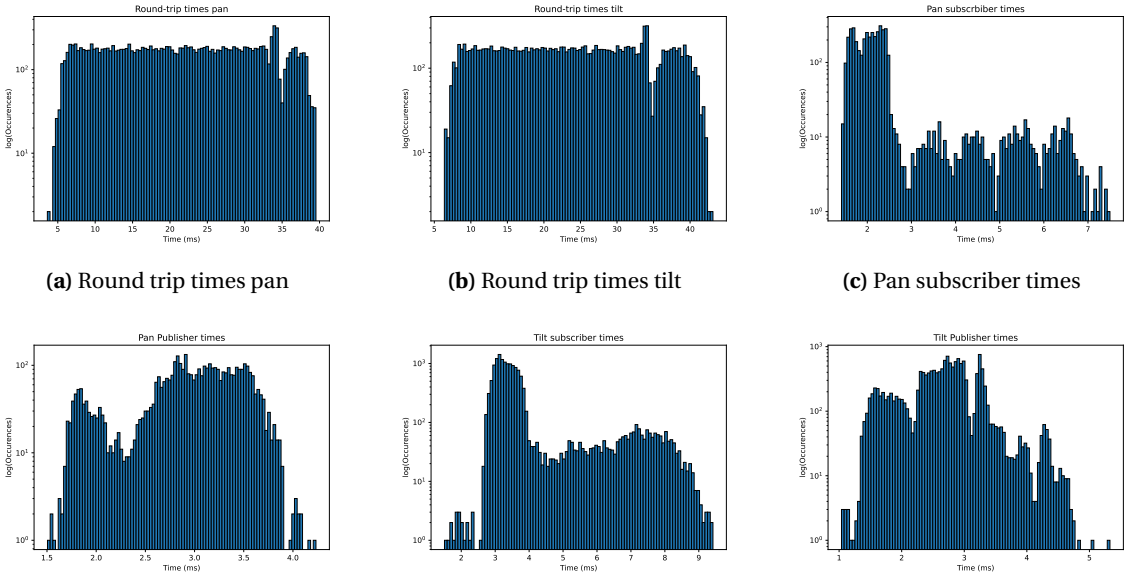


(a) Round trip times pan **(b)** Round trip times tilt **(c)** Pan subscriber times

(d) Pan publisher times **(e)** Tilt subscriber times **(f)** Tilt publisher times

Figure G.6: FreeRTOS Rasberry Pi Pico 2 at 10 Hz

G.2.3 JIWY communication measurements FreeRTOS Rasberry Pi Pico 2 at 30 Hz

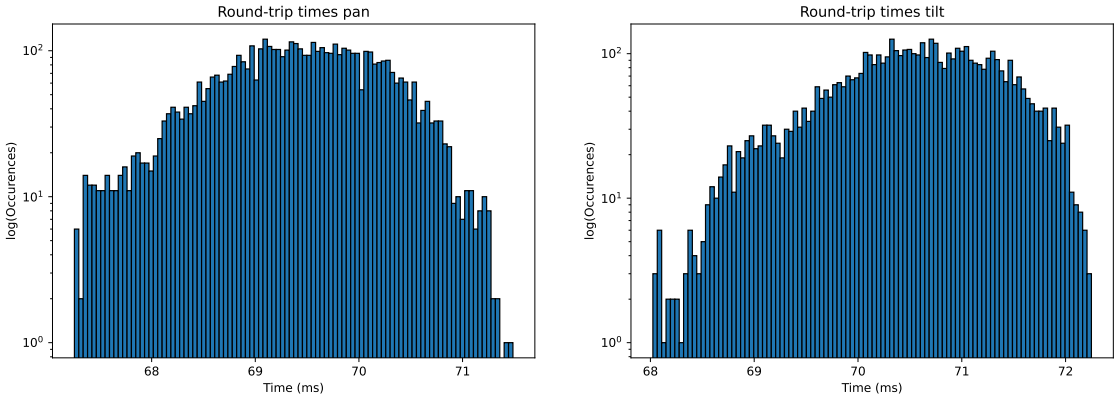


(a) Round trip times pan **(b)** Round trip times tilt **(c)** Pan subscriber times

(d) Pan publisher times **(e)** Tilt subscriber times **(f)** Tilt publisher times

Figure G.7: FreeRTOS Rasberry Pi Pico 2 at 30 Hz

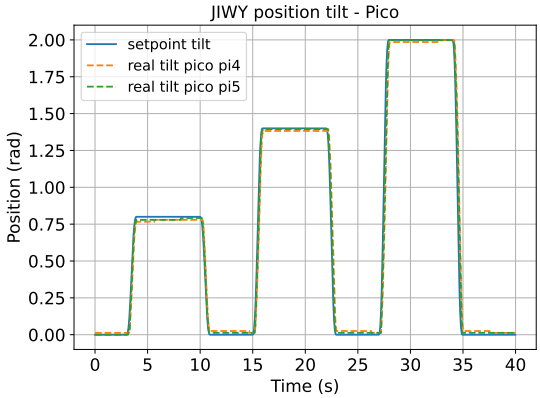
G.2.4 JIWY communication measurements FreeRTOS Rasberry Pi Pico 2 at 100 Hz



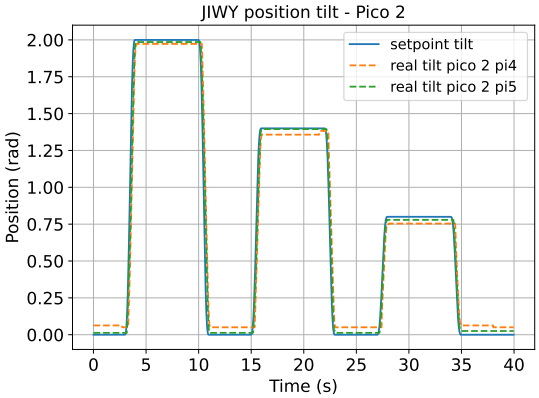
(a) Round trip times pan **(b)** Round trip times tilt

Figure G.8: FreeRTOS Rasberry Pi Pico 2 at 100 Hz

H JIWY motion tracking

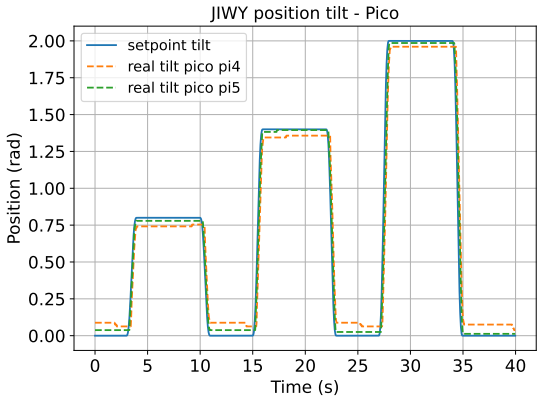


(a) Motion profile Pico

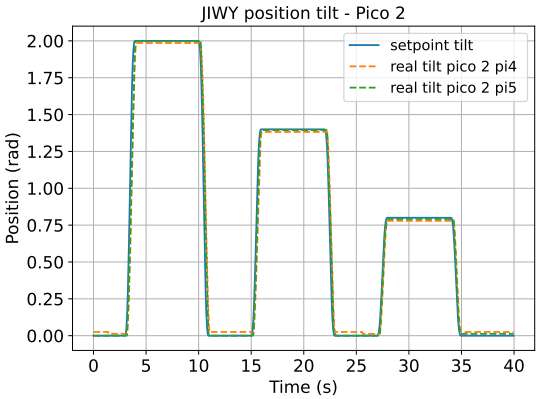


(b) Motion profile Pico 2

Figure H.1: Motion profile tracking



(a) Motion profile Pico swapped JIWYs



(b) Motion profile Pico 2 swapped JIWYs

Figure H.2: Motion profile tracking swapped JIWYs

References

Belsare, Kaiwalya et al. (2023). 'Micro-ROS'. In: *Robot Operating System (ROS): The Complete Reference (Volume 7)*. Cham: Springer International Publishing, pp. 3–55. ISBN: 978-3-031-09062-2. DOI: 10.1007/978-3-031-09062-2_2. URL: https://doi.org/10.1007/978-3-031-09062-2_2.

Bezemer, Maarten, Robert Wilterdink and J.F. Broenink (June 2011). 'LUNA: Hard real-time, multi-threaded, CSP-capable execution framework'. In: *Communicating Process Architectures 2011*. Vol. 68. Concurrent System Engineering Series WoTUG-33. IOS, pp. 157–175. ISBN: 978-1-60750-773-4. DOI: 10.3233/978-1-60750-774-1-157.

Brugali, Davide and Patrizia Scandurra (2009). 'Component-based robotic engineering (Part I) [Tutorial]'. In: vol. 16. 4, pp. 84–96. DOI: 10.1109/MRA.2009.934837.

Bruyninckx, Herman (2002). 'OROCOS: design and implementation of a robot control software framework'. In: *Proc. IEEE RAS EMBS Int. Conf. Biomed. Robot. Biomechatron.* Citeseer.

Chen, Celia et al. (2017). *Why Is It Important to Measure Maintainability and What Are the Best Ways to Do It?* DOI: 10.1109/ICSE-C.2017.75.

Controllab (2025). *20-sim - Bond Graph-based Multidomain Simulation Software*. Accessed: February 7, 2025. Controllab Products B.V. URL: https://www.20sim.com/.

Dokter, J. (July 2016). *20-sim Template for Raspberry Pi 3*. BSc Thesis 019RaM2016. University of Twente. URL: https://cloud.ram.eemcs.utwente.nl/index.php/s/SydM7FSnCiBiaYw.

eProsima (2024). *eProsima Fast DDS Documentation*. Accessed: 25 February 2025. URL: https://fast-dds.docs.eprosima.com/en/latest/.

EVL Project (2023). *EVL Project*. https://evlproject.org/. Website.

Gibson, David (2006). *Device Tree Compiler*. https://ozlabs.org/~dgibson/papers/dtc-paper.pdf. Accessed: 2025-01-28.

In 't Veld, N.E.D. (Dec. 2023). *Control of the Production Cell on Raspberry Pi using a real-time robot-software framework*. MSc Thesis 056RaM2023. University of Twente. URL: https://cloud.ram.eemcs.utwente.nl/index.php/s/BWBn4TEbQSik668.

Kempenaar, J.J. (Jan. 2014). *Communication Component for Multiplatform Distribution of Control Algorithms*. MSc Thesis 001RaM2014. University of Twente. URL: https://cloud.ram.eemcs.utwente.nl/index.php/s/PNS7YzGrMbp6zgc.

Lee, Hyun-Jae and Hak Yi (2021). *Development of an Onboard Robotic Platform for Embedded Programming Education*. DOI: 10.3390/s21113916.

Linux Foundation (2025). *Linux and the Devicetree*. Accessed: 2025-01-20. URL: https://www.kernel.org/doc/html/latest/devicetree/usage-model.html.

Liu, Wei et al. (Aug. 2022). 'Zoro: A robotic middleware combining high performance and high reliability'. en. In: *Journal of Parallel and Distributed Computing* 166, pp. 126–138. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2022.04.010. URL: https://www.sciencedirect.com/science/article/pii/S0743731522000879 (visited on 12/03/2023).

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.

Meijer, A. (Oct. 2021). *Real-time robot software framework on Raspberry Pi using Xenomai and ROS2*. MSc Thesis 070RaM2021. University of Twente. URL: https://cloud.ram.eemcs.utwente.nl/index.php/s/TPtt2yjixfN5DCc.

Object Management Group (Feb. 2020). *DDS-XRCE: DDS for Extremely Resource Constrained Environments Specification*. Version 1.0. Accessed: 19 February 2025. URL: https://www.omg.org/spec/DDS-XRCE/.

Radon Developers (2025). *Radon Documentation*. Accessed: 2025-02-08. URL: https://radon.readthedocs.io/en/latest/commandline.html.

Raoudi, I. (Apr. 2023). *Latency reduction and modularity improvement of a Raspberry Pi - FPGA control system*. Pre-MSc Thesis 011RaM2023. University of Twente. URL: https://cloud.ram.eemcs.utwente.nl/index.php/s/yCTxXcqHC8arBD6.

— (Dec. 2024). *ROS2 – Xenomai4 Real-time Framework on Raspberry Pi*. MSc Thesis 078RaM2024. msc. University of Twente. URL: https://cloud.ram.eemcs.utwente.nl/index.php/s/g6SijsJ4P8igszo.

Soetens, P. (2024). *RTT: Real-Time Toolkit*. http://www.orocos.org/rtt.

Stampfer, Dennis et al. (Aug. 2016). 'The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software'. In: *Journal of Software Engineering for Robotics (JOSER)* 7, pp. 3–19.

Szyperski, Clemens (2002). *Component Software: Beyond Object-Oriented Programming*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201745720.

The Eclipse Foundation (2023). *Papyrus for robotics*. https://www.eclipse.org/papyrus/. Website.

USB Implementers Forum (Apr. 2000). *Universal Serial Bus Specification, Revision 2.0*. Accessed: 20 February 2025. URL: https://www.usb.org/document-library/usb-20-specification.

Vinkenvleugel, J.T. (July 2022). *Designing an embedded software architecture for a mobile education robot with real-time control on a Raspberry Pi 4 with FPGA-based I/O*. BSc Thesis 025RaM2022. University of Twente. URL: https://cloud.ram.eemcs.utwente.nl/index.php/s/aKqPCq4jppWonec.

Visser, B. (Aug. 2020). *A bare-metal microcontroller as a target for 20-sim-generated C code*. BSc Thesis 043RaM2020. University of Twente. URL: https://cloud.ram.eemcs.utwente.nl/index.php/s/CZg3SRs7yEf9Wen.

Vulcanexus (2023). URL: https://docs.vulcanexus.org/en/latest/rst/tutorials/cloud/kubernetes/kubernetes.html.

Wenger, Monika et al. (Sept. 2016). 'A model based engineering tool for ROS component compositioning, configuration and generation of deployment information'. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8. DOI: 10.1109/ETFA.2016.7733559.

Zephyr Project (2025). *Zephyr Project*. https://zephyrproject.org.

Zhang, Jiaqiang et al. (Dec. 2022). 'Distributed Robotic Systems in the Edge-Cloud Continuum with ROS 2: a Review on Novel Architectures and Technology Readiness'. In: *2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 1–8. DOI: 10.1109/FMEC57183.2022.10062523.