FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Integration of the safety certified PXROS-HR real-time operating system in ROS2 robotic system.
Student:	Bc. Jakub Zahradník
Supervisor:	Ing. Martin Daňhel, Ph.D.
Study program:	Informatics
Branch / specialization:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	until the end of summer semester 2023/2024

Instructions

Instructions:
1. Get familiar with ROS2 system, its real-time concept, and collect set of requirements expected from the underlying real-time operating system.
2. Analyze the existing feature set of the PXROS-HR operating system and map them to collected set of requirements.
3. Propose concepts and solutions for identified gaps in existing feature set of PXROS-HR with respect to ROS2 real-time needs. The specific scope of work will be determined by the supervisor.
4. Implement proposed solutions either as service drivers running above the kernel space or by modifying the kernel itself adding essential capabilities into it.
5. To prove the concept of a successful integration, try to create a minimalistic ROS2 system based on PXROS-HR real-time operating system.

Electronically approved by prof. Ing. Hana Kubátová, CSc. on 16 February 2023 in Prague.

Master's thesis

INTEGRATION OF THE SAFETY CERTIFIED PXROS-HR REAL-TIME OPERATING SYSTEM IN ROS2 ROBOTIC SYSTEM

Bc. Jakub Zahradník

Faculty of Information Technology
Department of Digital Design
Supervisor: Ing. Martin Daňhel, Ph.D.
May 3, 2023

Citation of this thesis: Zahradník Jakub. *Integration of the safety certified PXROS-HR real-time operating system in ROS2 robotic system.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I sincerely thank everyone who took the time to help me write this master's thesis. Firstly, I thank my supervisor Ing. Martin Daňhel, for his professional guidance, patience, and valuable advice that helped me complete the work. I would also like to sincerely thank my family for their support throughout my studies. I would also like to thank Roman Knížek – the product owner of the PXROS-HR, Radek Olexa – my boss; and all the other colleagues of the HighTec company for the discussions regarding the proposed implementation, sharing ideas about this thesis and sharing their knowledge. Special thanks also go to all my friends who mentally supported and motivated me, especially Lucie Hartmanová, Jakub Horáček and Dominik Resl.

Declaration

In Prague on May 3, 2023 . . . .

Abstract

This master's thesis provides a comprehensive analysis of the Robot Operating System (ROS) 2, including its architecture, communication patterns, concepts, and use of the Data Distribution Service (DDS) as a middleware for data sharing. Additionally, the thesis explores Micro-ROS, a lightweight version of ROS 2 designed to run on microcontrollers with limited resources. This work focuses on analyzing the Micro-ROS's architecture, features, and suitability for embedded systems use. Additionally, the thesis explores using PXROS-HR, a real-time operating system (RTOS), in the proposed solution.

The proposed solution involves building a custom static library for Micro-ROS and implementing a mutex task for thread-safe data access. The project structure is presented, including configurations and linker files, and describes the implementation of custom allocators and custom transport for Micro-ROS. The thesis also includes demonstrations of multithread publisher-subscriber and multicore publisher-subscriber for Micro-ROS, showcasing the proposed solution's feasibility and effectiveness.

Furthermore, the proposed solution is evaluated by conducting one test for each demo, including publishing and subscribing to a topic, creating a service server for remote procedures, and distributing work to multiple tasks or cores. The results demonstrate that the proposed solution achieves thread-safe data access and enables efficient communication in resource-constrained environments.

Keywords Aurix, HighTec, Infineon, Micro-ROS, PXROS-HR, PXROS, Real-time, ROS, ROS 2, Robotic Operating System, RTOS, TriCore

Abstrakt

Tato diplomová práce poskytuje komplexní analýzu robotického operačního systému (ROS) 2, včetně jeho architektury, komunikačních vzorů a konceptů, a také jeho využití Data Distribution Service (DDS) jako middlewaru pro sdílení dat. Dále se práce zabývá Micro-ROS, odlehčenou verzí ROS 2 navrženou pro provoz na mikrokontrolérech s omezenými zdroji. Práce se věnuje architektuře, vlastnostem a vhodnosti Micro-ROS pro použití ve vestavných systémech. Následně práce rozebírá využití PXROS-HR, operačního systému reálného času (RTOS), v navrhovaném řešení.

Navrhované řešení zahrnuje vytvoření vlastní statické knihovny pro Micro-ROS a implementaci mutex tasku pro bezpečný přístup k datům. Práce představuje strukturu projektu včetně konfigurací, linker souborů a popisuje implementaci vlastních alokátorů a vlastního transportu pro Micro-ROS. Práce také obsahuje ukázky vícevláknových publisher-subscriber a vícejádrových publisher-subscriber aplikací pro Micro-ROS, které ukazují proveditelnost a efektivitu navrhovaného řešení.

Dále je vyhodnoceno navržené řešení provedením jednoho testu pro každé demo, který zahrnuje publishing a subscribing k topicu, vytvoření servisního serveru pro vzdálené procedury a distribuci práce na více úloh nebo jader. Výsledky ukazují, že navrhované řešení dosahuje bezpečného přístupu k datům a umožňuje efektivní komunikaci v prostředích s omezenými zdroji.

Klíčová slova Aurix, HighTec, Infineon, Micro-ROS, PXROS-HR, PXROS, Real-time, ROS, ROS 2, Robotický operační systém, RTOS, TriCore

List of abbreviations

AMR	Autonomous Mobile Robot
API	Application Programming Interface
ARP	Address Resolution Protocol
ASIL	Automotive Safety Integrity Level
ASIO	Audio Stream Input Output
BSD	Berkeley Software Distribution
CIoT	Consumer Internet of Things
CLI	Command-Line Interface
DDS	Data Distribution Service
GUI	Graphical User Interface
IDE	Integrated Development Environment
IDL	Interface Definition Language
IEC	International Electrotechnical Commission
IIoT	Industrial Internet of Things
IoT	Internet of Things
IP	Internet Protocol
ISO	International Organization for Standardization
lwIP	Light-Weight Internet Protocol
MCU	Cicrocontroller Unit
MIT	Massachusetts Institute of Technology
OMG	Object Management Group
OS	Operating System
PC	Personal Computer
POSIX	Portable Operating System Interface
PXROS-HR	Portable eXtendible Real-time Operating System – High Realiability
QoS	Quality of Service
RAM	Random Access Memory
ROS	Robotic Operating System
RPC	Remote Procedure Call
SLAM	Simultaneous Localization and Mapping
SW	Software
RTOS	Real-Time Operating System
RTPS	Real-Time Publish-Subscribe
SIL	Safety Integrity Level
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
USB	Universal Serial Bus
Wi-Fi	Wireless Fidelity
XRCE	eXtremely Resource-Constrained Environment

Chapter 1 Introduction

The focus of this master's thesis is to explore the possibility of integrating ROS (Robot Operating System) with PXROS-HR (Portable eXtendible Real-time Operating System – High Reliability), a certified real-time operating system. PXROS-HR is a highly reliable and portable real-time operating system that has obtained certifications for TriCore, validating its suitability for safety-critical applications up to SIL 3 (IEC 61508) and ASIL D (ISO 26262). The thesis aims to explore the potential benefits and drawbacks of such an integration and provide insights into the implementation process.

The thesis starts with an analysis of ROS and its features, including the design principles, communication patterns, architecture, and security features. It also covers the latest version of ROS, ROS 2, and its key features, such as support for real-time systems and the Data Distribution Service (DDS) protocol. Additionally, the thesis introduces Micro-ROS, a lightweight implementation of ROS that can run on microcontrollers with limited resources, and discusses its architecture, features, and applications.

Furthermore, the thesis explains DDS, a standard protocol for data-centric communication, and its role in ROS 2 and Micro-ROS. It describes the DDS model, architecture, and key features, including the Publish-Subscribe pattern and the Real-Time Publish-Subscribe (RTPS) protocol. The thesis also discusses PXROS-HR, an RTOS from HighTec, and its unique features.

The proposed solution and implementation section details the steps taken to integrate ROS with PXROS-HR using Micro-ROS. It covers building the Micro-ROS library and making code modifications to support mutex implementation, project structure, and testing. The section also includes examples of implementing custom allocators, custom transport, multithreaded and multicore publisher-subscriber communication, and other Micro-ROS demos.

Finally, the thesis concludes with testing results. The testing section evaluates the proper functioning of the proposed solution by separately testing each demo, which includes publishing, subscribing to a topic, creating a service server for remote procedures, and the ability to distribute the work to multiple tasks (threads) or multiple cores. The results of the testing section demonstrate the feasibility and efficacy of the proposed solution.

1.1 Motivation

The field of robotics and automation is rapidly expanding, and there is a growing need for safe and reliable real-time operating systems to support increasingly complex and sophisticated robotic systems. PXROS-HR is a certified real-time operating system with high reliability, designed to meet the safety requirements of safety-critical applications in various domains, including automotive, aerospace, and industrial automation. On the other hand, the Robot Operating System

(ROS) has become the de-facto standard in robotics research and development due to its flexibility, versatility, and ease of use.

However, integrating PXROS-HR with ROS is not a straightforward task due to differences in their underlying architectures and design philosophies. This integration can bring many benefits, including real-time performance, safety, and reliability of robotic systems, and opens up new possibilities for building more sophisticated robotic applications. Therefore, this thesis aims to investigate the feasibility of integrating PXROS-HR with ROS and to provide a proof-of-concept implementation. The results of this thesis can help researchers and practitioners in the field of robotics to build safer, more reliable, and more performant robotic systems.

1.2 Goals

The main goal of this thesis is to assess the feasibility of using PXROS-HR as the real-time operating system for micro-ROS. The thesis aims to identify the essential requirements of the underlying operating system and evaluate PXROS-HR's ability to meet those requirements. If any requirements are unmet, the thesis proposes potential solutions to enable the use of PXROS-HR in micro-ROS projects. Additionally, the thesis seeks to develop a prototype example that showcases the effectiveness of PXROS-HR as a real-time operating system for robotics applications in conjunction with micro-ROS.

1.3 Current State

Micro-ROS currently supports three real-time operating systems: FreeRTOS, Zephyr, and NuttX. FreeRTOS has been developed over 18 years in collaboration with leading chip companies and is now distributed under the MIT open-source license. It is known for its simplicity and micro-ROS uses the POSIX extension provided by FreeRTOS. Zephyr is an open-source, scalable project that aims to obtain functional safety certification and is developed within a Linux Foundation Project, making it the first open-source RTOS with such a certification. NuttX, on the other hand, emphasizes compliance with standards and has a UNIX API mimic that simplifies development for Linux users. [1]

Each of these RTOSes supports multiple architectures and offers various features to users, with different board support and licensing. However, RTOS-based support remains the primary entry point to micro-ROS, despite the fact that real-time operating systems are commonly used in resource-constrained environments. It is possible to use micro-ROS directly on bare metal with an integration into the Arduino IDE.

Additionally, the experimental Arm Mbed OS is a new real-time operating system supported by the micro-ROS project, specifically designed for creating IoT applications using an Arm Cortex-M microcontroller.

Micro-ROS does not directly connect to the ROS 2 world, and instead relies on a micro-ROS agent to facilitate communication between resource-constrained environments and standard ROS 2 systems. The agent acts as a node, following the client-server paradigm, serving as a server between the DDS Network and Micro-ROS nodes inside the MCU, and interacting with DDS Global Data Space on behalf of the Micro-ROS nodes.

Chapter 2

Analysis

Robot Operating System (ROS) is a powerful tool for the development of robotic systems that has gained popularity in recent years. As such, it has become the focus of numerous research studies exploring its functionality, architecture, and applications. In this master's thesis, the analysis chapter provides an in-depth exploration of ROS, including its latest version ROS 2, and a related project, Micro-ROS. Additionally, this chapter discusses the Data Distribution Service (DDS) and how it relates to ROS 2. The purpose of this chapter is to evaluate the design principles, communication patterns, architecture, security, and features of these technologies. The analysis also examines how ROS 2 and Micro-ROS address the limitations of the previous ROS version and how DDS enhances communication and data distribution in ROS 2. The chapter concludes with PXROS-HR, a Real-Time Operating System (RTOS) developed by HighTec for embedded systems. Finally, RTOS characteristics and PXROS-HR special features will be discussed in more detail in the last section of this chapter.

2.1 Robot Operating System

This section will provide an overview of ROS, starting with a discussion of what ROS is and its features. It will then provide a brief historical background of the platform. The next subsection will explore why ROS has become such a popular choice among robotics developers, highlighting the key advantages it offers. Finally, the section will provide an overview of the different versions of ROS that have been released to date, detailing the new features and improvements in each version.

2.1.1 What Is ROS?

Robot Operating System (ROS) is an open-source, meta-operating system for robots. It provides a collection of software libraries and tools to help software developers create robot applications. ROS offers a modular and distributed architecture, which allows developers to build complex robot behaviors by integrating different software components.

ROS was initially developed by Willow Garage in 2007 for use in their personal robotics research projects, and it has since grown into a widely used platform for robotics research, education, and industry. ROS has a large and active community of users and contributors who have developed a wide range of packages, libraries, and tools to extend its capabilities.

ROS supports a variety of programming languages, including C++, Python, and Java, and provides a standard communication infrastructure for different parts of a robot system to communicate with each other. This communication infrastructure is based on a publish-subscribe messaging model and enables developers to easily integrate sensors, actuators, and other hardware components into their robot systems.

Overall, ROS provides a flexible and powerful platform for developing robot applications, allowing developers to focus on higher-level tasks such as perception, planning, and control, while abstracting away the underlying hardware and communication infrastructure. [2, 3]

2.1.2 Why ROS?

ROS is a popular open-source middleware widely used for developing robotic applications. ”It’s the fastest way to build a robot!” says [4], providing some reasons why ROS is a good choice for robotics development:

- Global community of millions of developers and users contributing to and improving the software
- Proven in use across the robotics industry, teaching, research, and large-scale competitions
- Helps to create billions of dollars in value in the autonomous mobile robot (AMR) industry
- Shortens time to market by providing the tools, libraries, and capabilities needed for robotics applications
- Provides flexibility and freedom to customize ROS according to business needs
- Multi-domain ready for use across a wide array of robotics applications
- Multi-platform supported and tested on Linux, Windows, macOS, and various embedded platforms via micro-ROS
- 100% open-source ensuring free and unfettered access to a high-quality, fully featured robotics SDK
- Commercial-friendly distribution under permissive open-source licenses, such as Apache 2.0
- Strong industry support demonstrated by the membership of the ROS 2 Technical Steering Committee

Overall, ROS is a powerful and versatile platform that enables the rapid development and prototyping of complex robotics and automation systems. Its modular architecture, large community, open-source nature, interoperability, visualization and simulation tools, robust communication infrastructure, and flexibility make it a popular choice among roboticists and researchers. [4]

Benefits

In the past, robot designers and researchers needed to spend significant amounts of time designing both the hardware and embedded software for each robot project. This process required specialized skills in mechanical engineering, electronics, and embedded programming. The resulting programs were more akin to embedded programming rather than robotics as we know it today. Programs were closely tied to the underlying hardware, leading to significant program reuse.

Robot operating systems were developed to address these challenges by offering standardized functionalities that abstract hardware, similar to conventional operating systems for personal computers. ROS acts as a facilitator for robotics projects, allowing researchers and engineers to avoid continuously reinventing the wheel for each project and reducing financial costs.

ROS has a positive impact on research and development by reducing costs and time to market, making it an attractive option for those who need to quickly launch a new prototype or reduce a technological gap. Additionally, ROS allows experts from various disciplines to combine their knowledge and skills to design and program robots, as robotics projects require a diverse set of skills, often beyond those of a single individual.

ROS lowers the technical level required for working on robotics projects, making it easier for companies to get started in robotics or design complex systems more quickly. In summary, ROS offers a standardized and efficient approach to robotics that streamlines the design and programming process, reduces costs, and facilitates collaboration across disciplines. [5]

2.1.3 Versions

This subsection will present ROS versions and give additional information about the reasons behind the changes.

ROS 1

ROS 1 provides a comprehensive set of libraries that facilitate the development of various types of robots, including tools for monitoring processes, communication introspection, and time-series transformations, among others. Moreover, it features a large collection of sensor, control, and algorithmic packages contributed by the community that enables even small teams to create complex robotics applications. Despite these benefits, ROS 1 faces several challenges such as inconsistency in data delivery over lossy links, absence of built-in security mechanisms, and a single point of failure. Although the ROS 1 community has made attempts to address these issues, such efforts often involve compromises due to architectural and engineering limitations. For instance, addressing the single point of failure required bespoke solutions for each existing client library. Another attempt involved extending ROS 1 for security via the SROS project, but it was difficult to maintain and needed further development to align with security trends. These efforts extended ROS 1's lifespan but failed to address its core limitations. [2]

Reason to Make ROS 2

Robot Operating System began as a software development environment for the Willow Garage PR2 robot, aiming to provide software tools for research and development projects while also being useful on other robots. ROS 1 was guided by the PR2 use case, which was a single robot with workstation-class computational resources, no real-time requirements, excellent network connectivity, and applications in research, mostly academia. ROS 1 overshot its goal and became useful on a wide variety of robots, even being adopted by industries like manufacturing and agriculture. However, with new use cases arising, ROS 1 was stretched in unexpected ways. ROS 2 was then developed to meet the needs of a broader community by tackling new use cases head-on, such as teams of multiple robots, small embedded platforms, real-time systems, non-ideal networks, production environments, and prescribed patterns for building and structuring systems. At the core of ROS is an anonymous publish-subscribe middleware system, which is now built using off-the-shelf open-source libraries that benefit from ongoing improvements and are already used in production systems. ROS 2 also aims to improve user-facing APIs by designing new APIs, while still maintaining the key concepts of distributed processing, anonymous publish/subscribe messaging, Remote Procedure Call (RPC) with feedback, language neutrality, and system introspectability. While ROS 2 will not be API-compatible with existing ROS code, mechanisms will be in place to allow both to coexist. [6]

ROS 2

ROS 2 is the successor to the original ROS. It was designed to overcome the limitations of ROS 1 (such as performance issues and lacked support for real-time systems) and to address the evolving needs of the robotics industry. It also offers better support for multiple operating systems, programming languages, and hardware architectures, making it easier for developers to build and deploy robot applications. ROS 2 uses the Data Distribution Service (DDS) standard for communication, which provides better performance and scalability than the previous ROS 1 communication protocol. ROS 2 also has a more modular architecture, which allows developers to use only the components they need and to add new components as necessary.

One of the key goals of ROS 2 is to enable the development of more complex and robust robotic systems. It achieves this by providing better support for multi-robot systems and more advanced hardware, as well as a wider range of programming languages and tools. Moreover, ROS 2 incorporates new features that make it easier to integrate with other software systems, including cloud services and machine learning frameworks. [7, 8]

Comparison of ROS 1 with ROS 2

A comparison of significant distinctions between ROS 1 and ROS 2 is presented in 2.1, taken from [2].

Category	**ROS 1**	**ROS 2**
Network transport	Bespoke protocol built on TCP/UDP	Existing standard (DDS), with abstraction supporting addition of others
Network architecture	Central name server (roscore)	Peer-to-peer discovery
Platform support	Linux	Linux, Windows, and macOS
Client libraries	Written independently in each language	Sharing a common underlying C library (rcl)
Node versus process	Single node per process	Multiple nodes per process
Threading model	Callback queues and handlers	Swappable executor
Node state management	None	Lifecycle nodes
Embedded systems	Minimal experimental support (rosserial)	Commercially supported implementation (micro-ROS)
Parameter access	Auxilliary protocol built on XMLRPC	Implemented using service calls
Parameter types	Type inferred when assigned	Type declared and enforced

Table 2.1 Summary of ROS 2 features compared with ROS 1. [2]

2.2 ROS 2

This section provides an overview of ROS 2, including its design principles and requirements that guided its development. It discusses the various communication patterns supported in ROS 2, as well as the architecture of the system. The section describes the internal interfaces used by ROS 2, including the ROS middleware interface (*rmw* API) and the ROS client library interface (*rcl* API). In addition, it offers conceptual overviews that offer broad and high-level introductory details on important elements of ROS 2. A thorough analysis of the system's performance and reliability aspects is presented, along with a discussion of the security considerations that were taken into account during its development. Lastly, the section provides a brief overview of related projects that are based on or related to ROS 2.

Scope

The Robot Operating System 2 is a versatile platform that caters to a wide range of robotics applications, including education, research, product development, and deployment. ROS 2 is composed of several interrelated software components, categorized into three areas (taken from [2]):

1. **Middleware**

   The middleware component, which handles communication among components, is considered the foundation of ROS 2 and includes network APIs and message parsers.

2. **Algorithms**

   ROS 2 also provides commonly used algorithms in robotics applications, such as perception, Simultaneous Localization and Mapping (SLAM), and planning.

3. **Developer tools**

   ROS 2 includes a suite of developer tools for configuration, debugging, simulation, and logging.

2.2.1 Design Principles & Requirements

The design of the ROS 2 has been guided by a set of design principles, including:

- **Distribution** – involves separating requirements into functionally independent components that have their own execution context and share data via explicit communication in a decentralized and secure manner
- **Abstraction** – is used to govern communication through interface specifications that define the semantics of the data exchanged, leading to an ecosystem of interoperable components abstracted away from specific vendors of hardware or software components
- **Asynchrony** – enables communication among components asynchronously, creating an event-based system that allows an application to work across multiple time domains
- **Modularity** – is enforced at multiple levels, and the ecosystem is organized into a large number of federated packages rather than a single codebase

These design principles have trade-offs, such as making it more difficult to achieve deterministic execution, but adherence to them generally leads to better outcomes, including code reuse, software testing, fault isolation, collaboration within interdisciplinary project teams, and cooperation at a global scale. [2]

To meet the needs of robotics developers, a system has been designed with the following requirements:

- Security
- Embedded systems support
- Diverse networking environments support
- Real-time computing
- Product readiness

ROS 2 has an integrated security system that comprises authentication, encryption, and access control to secure its communication channels. Additionally, Micro-ROS helps to extend the reach of ROS 2 to embedded systems. ROS 2's quality of service is designed to adapt to the constraints of a network, while APIs enable real-time computing. Furthermore, Apex.AI[1] has achieved functional safety certification for their ROS 2-based autonomous vehicle software. This certification enables the use of ROS 2 in safety-critical systems, such as autonomous vehicles and heavy machinery.[2]

2.2.2 Communication Patterns

The ROS 2 documentation includes engaging animations and clear explanations that demonstrate how these patterns work, and they can be found in the tutorial[2] section.

ROS 2 Graph

The ROS graph is a network of ROS 2 elements processing data together at one time. It encompasses all executables and the connections between them if you were to map them all out and visualize them. [10]

Node

Node is a software component in the ROS 2 framework that performs a specific task or set of tasks within a robot application. It is recommended that every node has a single specific function, such as controlling a particular component of the robot (e.g., one node for wheel motors and another for a laser range-finder), and they can communicate with each other using topics, services, actions, or parameters. Multiple nodes work together to form a complete robotic system. In ROS 2, a single executable, such as a C++ or Python program, can include one or more nodes. [10]

Parameters

ROS 2 parameters are a way to store and retrieve data that affects the behavior of a node. Parameters can be thought of as configuration variables that can be set and modified at runtime. Unlike ROS 1, ROS 2 parameters are type-safe, meaning that the parameter's data type is specified and enforced by the system. This ensures that nodes receive valid parameter values and eliminates potential bugs caused by type mismatches.

[1]"Apex.Grace is a fork of ROS 2 that has been made so robust and reliable that it can be used for the development and production of highly-safety critical systems such as autonomous vehicles, robots, and aerospace applications. Apex.Grace is API-compatible to ROS 2. In a nutshell, Apex.Grace is an SDK for autonomous driving software and other safety-critical mobility applications." [9]

[2]ROS 2 tutorials are available from https://docs.ros.org/en/humble/Tutorials.html.

ROS 2 parameters can be set via command-line arguments, configuration files, or dynamically at runtime. They can be used to tune the behavior of a node or to specify information about the node, such as its name or namespace. Nodes can also use parameters to share information with other nodes in the system.

ROS 2 parameters are organized under a node, and nodes can expose their parameters to the rest of the system, allowing other nodes to access and modify them. This makes parameters a useful tool for configuring and coordinating the behavior of a system of ROS nodes. [11]

Message

ROS 2 messages are the basic unit of data exchange in ROS 2. They are a structured way of exchanging information between ROS 2 nodes using topics, services, and actions. Each message is defined as a simple data structure in a language- and platform-independent manner, allowing nodes written in different programming languages to communicate with each other seamlessly.

ROS 2 messages are defined in ROS interface definition language (IDL) files, which describe the fields and data types that make up the message. The IDL files are used to generate language-specific code that can be compiled and linked into the nodes that use them. [12]

Topic

In ROS 2, the communication between nodes is often achieved through topics, which are a type of asynchronous message-passing framework. Similar to other asynchronous frameworks such as ASIO (Audio Stream Input Output), topics allow nodes to communicate with one another through publish-subscribe (described in 2.4.2) functionality. This approach is focused on organizing a system using strongly typed interfaces by arranging end points in a computational graph under the concept of a node.

One of the advantages of using topics is the ability to have many-to-many communication, which is useful for system introspection. This allows developers to observe messages passing on a topic by simply creating a subscription to that topic without making any changes to the system. This feature is achieved through the use of an anonymous publish-subscribe architecture. [2, 13]

Service

ROS 2 provides different communication patterns to accommodate different use cases. While asynchronous message-passing frameworks like topics can be advantageous for system introspection, they may not always be the right tool. In such cases, ROS 2 provides request-response communication through services. These services enable easy data association between a request and a response, which is useful for ensuring that a task was completed or received. Unlike traditional request-response patterns that may block a client's process, ROS 2 allows non-blocking service calls. Services are organized under a node for organization and introspection, allowing a subsystem's interfaces to appear together in system diagnostics.

A service consists of two parts:

- **Request** – a message that contains the data the client node wants to send to the service node
- **Response** – a message that contains the data the service node sends back to the client node

ROS 2 services are typically used when a node needs to perform a specific action, like setting a parameter or triggering a specific behavior in another node. [2, 14, 15]

Action

ROS 2 introduces the concept of actions which are goal-oriented, asynchronous communication interfaces. An action is defined by an action interface, which consists of a goal, feedback, and result message. Unlike services, actions are designed to manage complex and long-running operations, such as autonomous navigation or manipulation that can be preempted or canceled. Additionally, actions are designed to provide feedback to the client about the status of the requested operation, allowing clients to monitor and react to progress updates. As with services, actions are nonblocking and organized under the node, allowing for easy organization and introspection of a subsystem's interfaces. [2, 16, 17]

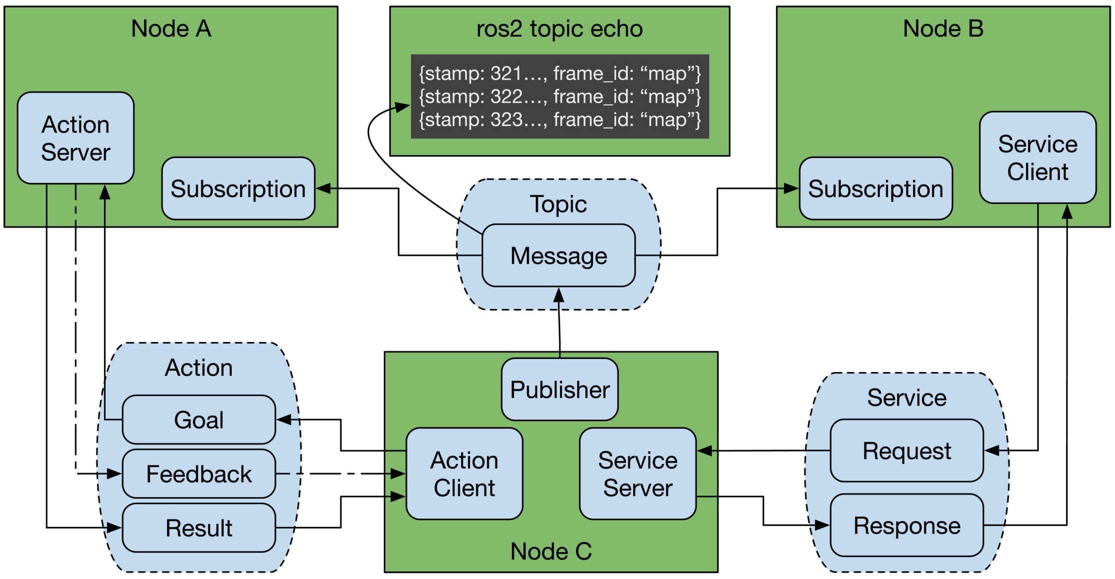


Figure 2.1 The picture shows the different communication patterns available in ROS 2 – topics, services, and actions. The arrows between nodes indicate the flow of information, with publishers and subscribers using topics, clients and services servers using services, and clients and actions servers using actions. [2]

2.2.3 Architecture

The middleware architecture of ROS 2 consists of several abstraction layers, which are generally hidden behind the client library during development. The client libraries provide access to the core communication APIs and are tailored to each programming language to make them more idiomatic and take advantage of language-specific features. Communication is agnostic to how the system is distributed across compute resources, whether they are in the same process, a different process, or even a different computer.

The middleware abstraction layer called ROS Middleware (*rmw*) provides the essential communication interfaces, and the vendors for each middleware implements the *rmw* interface and are made interchangeable without code changes. Users may choose different *rmw* implementations, and thereby different middleware technologies, based on performance, software license, or supported platforms. Although all of the supported *rmws* are based on DDS, a few community-supported *rmws* exist for other communication methods.

The network interfaces, such as topics, services, and actions, are defined with message types using an interface description language (IDL). ROS 2 defines these types using the ROS IDL format (*.msg* files) or the OMG IDL standard (*.idl* files), and user-provided interface definitions are generated at compile time and create code required for communication in any client library language.

ROS 2 provides a pattern for managing the lifecycle of nodes that transition through a state machine with states like unconfigured, inactive, active, and finalized. This is an important tool for coordinating various parts of the distributed asynchronous system.

Developers can use additional architectural patterns to structure their programs. ROS 2 offers a pattern that manages the lifecycle of nodes, moving through a state machine with different states, such as unconfigured, inactive, active, and finalized. These states enable system integrators to control when specific nodes are active, which is essential for coordinating various parts of the distributed asynchronous system.

As previously mentioned, communication is not tied to the location of endpoints within machines and processes. However, the decision of which machine or process to allocate each node should not be made during node development. Instead, it should be based on how the node is used in the larger system. Nodes developed as components can be allocated to any process as a configuration, which is an important feature for systems under development. It allows developers to rearrange where nodes are running based on various circumstances. For example, several nodes can share a process to conserve system resources or reduce latency. [2]

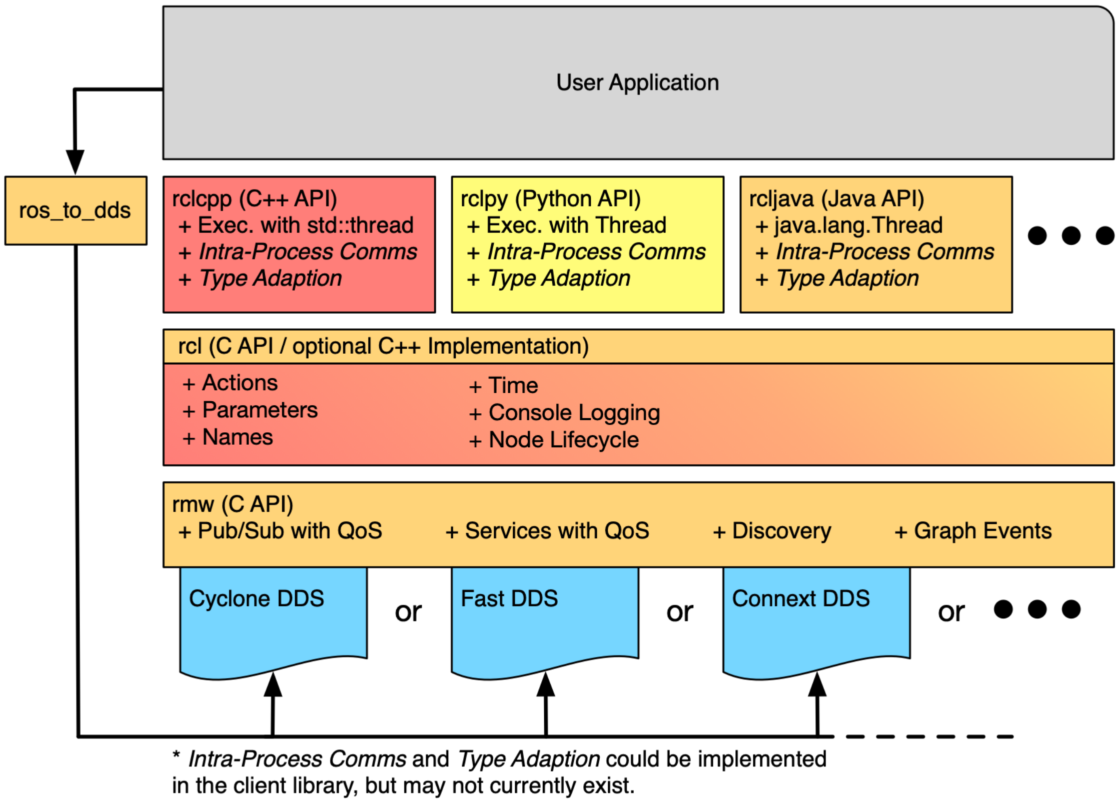


Figure 2.2 The picture shows the architecture of ROS 2, divided into two main internal interfaces – the *rmw* (ROS Middleware) layer, the *rcl* (ROS Client Library) layer; and the client libraries layer. The *rmw* layer provides an abstraction of the communication layer, allowing ROS 2 to be used with different middleware implementations. The *rcl* layer provides the basic building blocks for ROS 2 applications. The client libraries layer provides the high-level APIs for creating and running ROS 2 applications, such as the C++ and Python libraries. [18]

2.2.4 Internal Interfaces

The internal interfaces in ROS 2 consist of the ROS middleware interface (*rmw* API) and the ROS client library interface (*rcl* API). The *rmw* API acts as an intermediary between the ROS 2 software stack and the underlying middleware implementation, which could either be a DDS or RTPS implementation. It facilitates discovery, publish and subscribe mechanics, request-reply mechanics for services, and serialization of message types.

On the other hand, the *rcl* API is a slightly higher level API that provides an abstraction layer between the middleware implementation and the client libraries, such as *rclcpp*. The *rcl* interface gives access to the ROS graph and graph events and is used by the client libraries. In contrast, the *rmw* API provides the minimum middleware functionality required to support ROS's client libraries. The *rmw* API implementation is provided by middleware implementation-specific packages, like *rmw_fastrtps_cpp*, which are compiled against vendor-specific DDS interfaces and types.

The goal of the interface abstraction is to isolate the ROS user space code from the underlying middleware implementation, allowing users to switch DDS vendors or even middleware technology with minimal impact on their code. The *ros_to_dds* (shown in 2.2) package category represents the packages that allow users to access DDS vendor-specific objects and settings using the ROS equivalents, without exposing vendor-specific symbols and headers in the normal interface. Despite this, it may be necessary on occasion to manually adjust settings, and these packages make it easy to identify any potential violations of vendor portability by inspecting the package's dependencies. [18]

2.2.5 ROS 2 Concepts

Conceptual overviews provides general background information about key aspects of the system. This includes high-level information on the architecture, key concepts, and features of ROS 2. The aim is to provide a foundational understanding of the system without diving into more technical details.

Executors

In ROS 2, an executor is responsible for executing callbacks for subscriptions and timers. It is essentially a thread or set of threads that continuously checks for new events and runs the corresponding callback functions. Executors help manage the asynchronous nature of ROS 2, ensuring that callbacks are executed in a timely and efficient manner.

ROS 2 provides several different types of executors, each with a specific behavior and use case, including single-threaded and multi-threaded executors. Single-threaded executors run all callbacks in a single thread, while multi-threaded executors execute callbacks in multiple threads, allowing for parallel processing of callbacks resulting in improved performance. In addition, ROS 2 provides executors that are optimized for specific use cases. For example, the *StaticSingleThreadedExecutor* is designed for use in systems where the number of subscriptions and timers is fixed and known in advance. The *StaticSingleThreadedExecutor* can pre-allocate memory for subscriptions and timers, resulting in more efficient callback execution.

Overall, executors are an important part of ROS 2, as they help manage the flow of data and events within a ROS 2 system. By providing a way to handle asynchronous events in a structured and efficient manner, executors help ensure that ROS 2 systems can operate effectively and reliably. [19]

ROS 2 Client Libraries

Client libraries are application programming interfaces that allow developers to write their code for the ROS. These libraries provide access to the fundamental concepts of ROS, including nodes, topics, services, and others. They are available in different programming languages so that developers can write their ROS code in a language that best suits their requirements. This enables developers to build their applications using Python for rapid prototyping and C++ for computationally intensive tasks.

Sharing of messages between nodes written using different client libraries is possible because all client libraries have code generators that allow users to interact with ROS interface files in their respective programming languages.

Moreover, client libraries expose the core functionality of ROS, such as names and namespaces, time, parameters, console logging, threading model, and intra-process communication, to developers. This functionality is essential in building ROS applications and is made available to developers through client libraries.

The ROS 2 client libraries, *clcpp* and *rclpy*, offer C++ and Python interfaces respectively, both of which use the common functionalities of the ROS Client Library (*rcl*).

- rclcpp

  The *rclcpp* package provides a C++ interface to create nodes, publishers, and subscribers, and is designed to be used with C++ messages generated by *rosidl_generator_cpp*. *Rclcpp* uses all the features of C++ and C++17 features to provide a user-friendly interface, while also maintaining consistent behavior with other client libraries that use the *rcl* API.

- rclpy

  Similarly, *rclpy* builds on top of the *rcl* C API and provides an idiomatic Python experience using native Python types and patterns. The Python client library generates custom Python code for each ROS message and converts the native Python message object into the C version of the message when needed. The *rclpy* also takes care of the execution model using threading.Thread or similar to run the functions in the *rcl* API. All operations happen on the Python version of the messages until they need to be passed into the *rcl* layer, at which point they are converted into the plain C version of the message so it can be passed into the *rcl* C API.

- Community-maintained

  Additionally, there are other community-maintained client libraries for languages such as Ada, C, JVM, .NET Core, Node.js, and Rust.

These client libraries make use of the common core *rcl* interface to implement logic and behavior of ROS concepts that are not language-specific, such as parameters and namespaces. By making use of the *rcl*, client libraries only need to wrap common functionalities with foreign function interfaces, which keeps them thin and easier to develop. Having a common core also makes the behavior between languages more consistent and requires less maintenance when it comes to bug fixes. [20]

The ROS_DOMAIN_ID

In ROS, a domain is a logical grouping of nodes that are able to communicate with each other. It is defined by a unique domain ID, which is an integer that is set by the user. Nodes that belong to the same domain can communicate with each other directly, without the need for extra configuration. Nodes in different domains can still communicate, but they need to explicitly specify the domain ID of the other nodes. This can be useful for separating different parts of a larger system or for isolating test environments. The ROS domain ID can be set using the *ROS_DOMAIN_ID* environment variable or using command-line arguments. [21]

QoS

ROS 2 provides a Quality of Service (QoS) settings framework to manage how data is exchanged between nodes in the system. The QoS settings define the reliability, durability, and communication speed of the data being sent. By configuring the QoS settings, developers can optimize the system to meet the requirements of their specific application. There are different types of QoS settings available in ROS 2, such as reliability, durability, history, liveliness, and deadline. Each type has different options that can be set to achieve the desired behavior. By understanding and appropriately setting the QoS parameters, developers can create a more efficient and reliable ROS 2 system. [22]

2.2.6 Security

In modern commercial robotics, security is an essential aspect that needs to be considered. ROS 2 is designed with security in mind and implements the DDS security standard along with a suite of tools called SROS2 that simplifies the management of security infrastructure. DDS security is based on three main concepts, described in [2]:

- Authentication

  Verifies the identity of a message or participant in the network. In ROS 2, digital signatures are utilized for authentication through public key cryptography. SROS2 includes command-line tools for generating and storing digital signatures.

- Access control

  Allows fine-grained policies to be applied to authorized network participants. It allows participants to communicate only with approved network interfaces and to discover approved participants. SROS2 includes command-line tools for generating these configurations.

- Encryption

  Guarantees that third parties cannot access or replay data into the network. Encryption is accomplished using the Advanced Encryption Standard Galois/Counter Mode (AES-GCM) symmetric-key cryptography. The key material is derived from the shared secret obtained during authentication. The following section explores the details of these security concepts in DDS security and SROS2, and the tools available for configuring them.

2.2.7 Related Projects

There are several related projects in the ROS 2 ecosystem. Among the most significant are:

- **Gazebo** – physics simulation tool for ROS-based robots
- **Ros2_control** – flexible framework for real-time control of robots implemented with ROS 2
- **Navigation2** – comprehensive and flexible navigation stack for mobile robots using ROS 2
- **MoveIt** – rich platform for building manipulation applications featuring advanced kinematics, motion planning, control, collision checking, and more
- **Micro-ROS** – platform for putting ROS 2 onto microcontrollers, starting at less than 100 kB of RAM (detailed information in 2.3)

There are also hundreds of further community projects that are developed and maintained by the global ROS community. Developers can maintain a ”README.md” file in the root of their package folder to document their package, which is then rendered into the overview page of the package at https://index.ros.org/. In addition, there are also company-driven projects from Intel and NVIDIA. [23]

2.3 Micro-ROS

Robot Operating System 2 is an open-source robotics middleware widely used in research and industry. It provides a wide range of libraries and tools to ease the development of complex robotic systems. However, the majority of ROS 2 users target high-end computing platforms such as laptops and servers, neglecting the potential of deploying ROS 2 on low-cost and low-power microcontrollers.

Micro-ROS aims to address this issue by bringing ROS 2 to microcontrollers, enabling the development of first-class ROS 2 entities in the embedded world. The project provides a client library optimized for microcontrollers, which includes all major ROS 2 concepts such as nodes, publish/subscribe, client/service, node graph, and lifecycle management. The client library is based on the standard ROS 2 Client Support Library (*rcl*) and a set of extensions and convenience functions (*rclc*), which are optimized for microcontrollers. Micro-ROS mission is defined as follows: ”Bridging the gap between resource-constrained microcontrollers and larger processors in robotic applications that are based on the Robot Operating System.” [24]

One of the main challenges of running ROS 2 on microcontrollers is memory usage, as these devices usually have limited resources. To address this challenge, Micro-ROS provides a complete article and tutorial on how to tune the memory consumption of the middleware. Additionally, the project uses Micro XRCE-DDS, a middleware implementation optimized for deeply embedded systems, which provides built-in support for various communication transports such as serial, Ethernet, Wi-Fi, Bluetooth, and 6LoWPAN.

Micro-ROS supports popular open-source real-time operating systems (RTOS) such as Zephyr, FreeRTOS, and NuttX, and can be ported to any RTOS that comes with a POSIX interface. The RTOS-specific build systems are integrated into a few generic setup scripts, which are provided as a ROS 2 package. [24]

Why Microcontrollers?

Microcontrollers are used extensively in robotic applications due to their hardware access, hard, low-latency real-time processing capabilities, and power-saving features. Additionally, microcontrollers are preferred for safety-critical applications, but it is important to note that micro-ROS is not developed according to any safety standard. [24]

2.3.1 Features and Architecture

The micro-ROS system allows for the implementation of major core concepts such as nodes, publish/subscribe, client/service, node graph, and lifecycle on microcontrollers. The micro-ROS client API, which is based on the standard ROS 2 Client Support Library (*rcl*) and a set of extensions and convenience functions (*rclc*), is optimized for MCUs and does not require dynamic memory allocations. The micro-ROS agent seamlessly connects micro-ROS nodes to standard ROS 2 systems, providing access to micro-ROS nodes with known ROS 2 tools and APIs. The extremely resource-constrained middleware, Micro XRCE-DDS, is integrated with the ROS middleware interface (*rmw*) in the micro-ROS stack and supports various transports. Micro-ROS also supports popular open-source real-time operating systems, and its build systems are integrated into generic setup scripts that allow ROS developers to use their usual command line tools. The architecture scheme is shown in 2.3. [25]

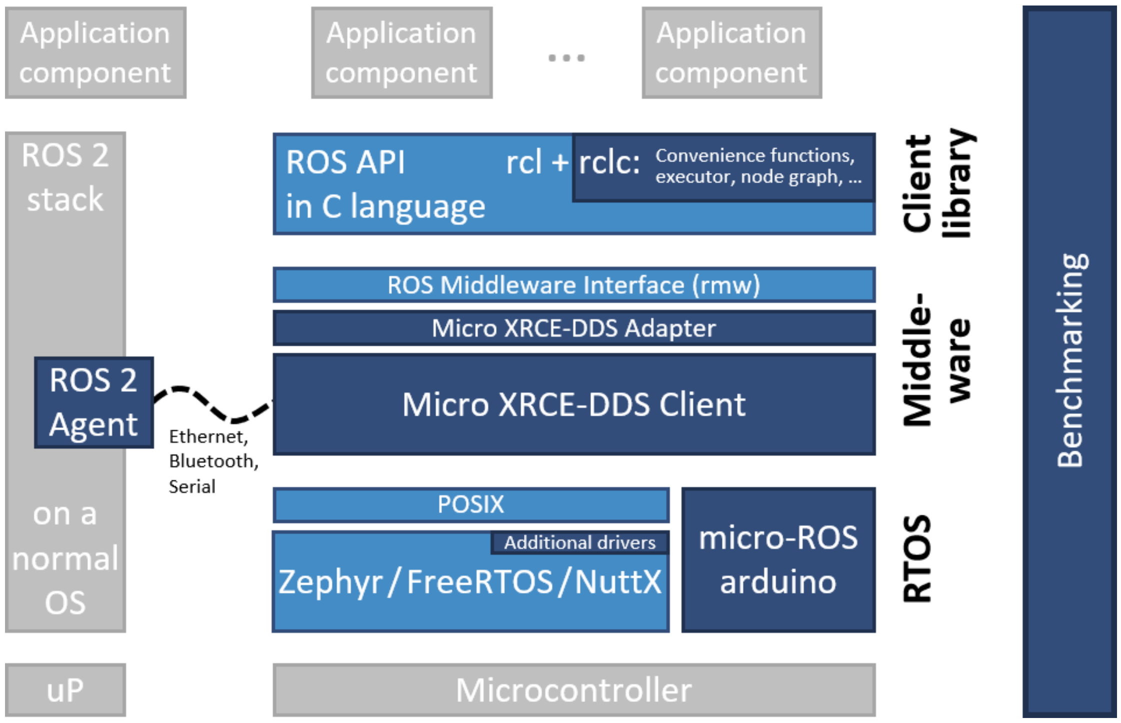


Figure 2.3 The image shows the architecture of Micro-ROS, a lightweight implementation of the Robot Operating System 2 (ROS 2) designed for microcontrollers and microprocessors. The architecture includes multiple layers, including the RTOS layer, middleware layer, and the micro-ROS client library layer. The image also depicts the different components of Micro-ROS, including the Micro XRCE-DDS agent, which handles communication between the microcontroller and the larger ROS 2 system, and the Micro-ROS client library, which provides the functionality needed to develop and deploy applications on the microcontroller. [25]

2.3.2 Supported RTOSes

The use of RTOSes is widespread due to the benefits they offer (more about what is RTOS is described in 2.5.2). RTOSes typically provide hardware abstraction layers that ease the use of hardware resources, such as timers and communication buses, and offer threads and tasks entities which, together with the use of schedulers, provide the necessary tools to implement determinism in the applications. They also offer stack management, which helps in the correct memory usage of the MCU resources, a valuable good in embedded systems.

In addition, the use of RTOSes allows the development of reusable code, as they offer different algorithms for scheduling and hardware abstraction layers that can be leveraged by different applications. The exchange of software entities is expected and desired at all levels, including the RTOS layer.

Micro-ROS supports three RTOSes – FreeRTOS, Zephyr, and NuttX – all of which are integrated into the micro-ROS build system. Here are the key features of each RTOS, taken from [1].

FreeRTOS

FreeRTOS is licensed under the MIT license and is well-known for its simplicity and the extension provided by Amazon called a:FreeRTOS. In the case of micro-ROS, the POSIX extension is utilized.

- Extremely small footprint
- POSIX extension available
- Memory management tools
- Standard and idle tasks available with assignable priorities
- Transport resources: TCP/IP and lwIP

Zephyr

Zephyr is a recent open-source real-time operating system that was created under the Linux Foundation Project, and is supported by various reputable semiconductor companies. The main objective of the project is to obtain a certification for functional safety, which would make Zephyr the first open-source RTOS to acquire such a certification.

- Small footprint
- Native POSIX port
- Cross Architecture: Huge collection of supported boards
- Extensive suite of Kernel services
- Multiple Scheduling Algorithms
- Highly configurable/Modular for flexibility
- Native Linux, macOS, and Windows Development

NuttX

NuttX focuses on its adherence to standards such as POSIX and its ability to fit on microcontrollers with 8- to 32-bit capacity. It has APIs that are similar to UNIX, which makes it convenient for developers familiar with Linux. NuttX operates under the BSD license and is developed using the GNU toolchain. However, the uClib++ library used with NuttX comes under the more restrictive GNU LGPL Version 3 license.

- POSIX compliant interface to a high degree
- Rich Feature OS Set
- Highly scalable
- Real-Time behavior: fully pre-emptible; fixed priority, round-robin, and "sporadic" scheduling

There is one more RTOS, yet still only experimentally supported – Arm Mbed OS, an open-source operating system for Arm Cortex-M microcontrollers. Micro-ROS can also be used as a bare-metal application, thanks to its release as a standalone library with header files, and its support for the Arduino IDE.

RTOS Requirements

The ROS 2 stack has been developed for operating systems according to the Portable Operating System Interface (POSIX) standards. In fact, ROS 2 only depends on a small subset of POSIX for process identification, clock, time, and filesystem operations. The use of POSIX is hidden by a thin abstraction layer implemented by the rcutils package and repository. This package also implements support for Windows and streamlines tiny differences between different Unix-like operating systems. Furthermore, rcutils implements an abstraction for atomic operations.

Micro-ROS adopts the assumption of POSIX but makes smaller changes to rcutils, using the same changeset mechanism as explained above for *rcl*: filesystem operations are disabled, dynamic memory allocations (in the context of error reporting) are prevented, and a user-level implementation of 64-bit atomic operations is provided.

The use of POSIX standards is essential for the porting or reuse of code of ROS 2 that was natively coded in Linux, as it ensures compatibility between different RTOSes. Both NuttX and Zephyr comply to a good degree with POSIX standards, making the porting effort minimal, whereas FreeRTOS provides a plugin, FreeRTOS+POSIX, thanks to which an existing POSIX compliant application can be easily ported to FreeRTOS ecosystem, and therefore leverage all its functionality. [26]

2.3.3 Supported Platforms

Micro-ROS has the objective of making ROS 2 accessible to a broad range of microcontrollers, enabling first-class ROS 2 components in the embedded domain. The primary focus of micro-ROS is mid-range 32-bit microcontroller families, which usually have limited memory resources. Typically, micro-ROS requires MCUs with at least tens of kilobytes of RAM and communication peripherals that support communication between the micro-ROS Client and Agent. Table 2.2 shows supported hardware, RTOSes, and transports together with the size of RAM and Flash memories.

Board	RTOSes	Supported transports	Support	RAM	Flash
Renesas EK RA6M5	FreeRTOS, ThreadX, Bare-metal	UDP, UART, USB-CDC	Official	512 kB	$\leq$ 2 MB
Espressif ESP32	FreeRTOS	UART, WiFi UDP, Ethernet UDP	Official	520 kB	4 MB
Arduino Portenta H7	—	USB, WiFi UDP	Official	8 MB	16 MB
Raspberry Pi Pico RP2040	—	USB, UART	Official	264 kB	$\leq$ 16 MB
ROBOTIS OpenCR 1.0	—	USB, UART	Official	320 kB	1024 kB
Teensy 3.2	—	USB, UART	Official	64 kB	256 kB
Teensy 4.0/4.1	—	USB, UART, Ethernet UDP (4.1)	Official	1024 kB	2048 kB
Crazyflie 2.1 Drone	FreeRTOS	Custom Radio Link	Official	192 kB	1 MB
STM32L4 Discovery kit IoT	Zephyr	USB, UART, Ethernet UDP	Official	128 kB	1 MB
Olimex LTD STM32-E407	Zephyr, FreeRTOS, NuttX	USB (Z, N), UART (Z, F, N), Ethernet UDP (F, N)	Official	196 kB	1 MB
Arduino Due	—	USB, UART	Community	96 kB	512 kB
Arduino Zero	—	USB, UART	Community	32 kB	256 kB
ST NUCLEO-F446ZE	FreeRTOS	UART	Community	128 kB	512 kB
ST NUCLEO-F746ZG	FreeRTOS	UART	Community	320 kB	1 MB
ST NUCLEO-H743ZI	FreeRTOS	UART	Community	1 MB	2 MB

Table 2.2 Supported platforms [27].

Micro-ROS also allows users to build their applications for Linux, which can be useful for testing and debugging purposes. Moreover, users can build a static library of Micro-ROS, which can be integrated into custom development tools. Finally, Micro-ROS also allows users to build a generic library for Android, expanding its compatibility with different platforms. [28, 27]

2.3.4 Build System

Micro-ROS provides two approaches to develop applications for embedded platforms. The first approach is a ROS-specific build system that includes modules to integrate software for cross-compiling applications on supported platforms in terms of both hardware and firmware. The second approach generates standalone modules and components that allow integration of micro-ROS into external or custom development frameworks. A dedicated tool is available to compile micro-ROS as a standalone library, enabling developers to use micro-ROS without the ROS build system. [27]

1. **micro_ros_setup**

   The micro_ros_setup is a ROS 2 package that provides a standalone build system suitable for any typical ROS 2 workspace. It enables the compilation and generation of micro-ROS apps in images for various supported hardware boards and RTOSes.

   Using micro_ros_setup is straightforward as it can be installed similar to any other ROS 2 package, and its functionalities can be accessed through the ROS 2 CLI tool. It requires only four ROS 2 commands to compile, generate, and flash an image onto a board.

2. **External build systems**

   The second approach to building micro-ROS involves the integration[3] of the micro-ROS build system into the build system of a larger project, followed by building micro-ROS as a component of the larger project. This approach offers increased flexibility and control over the build process; however, it may require a greater amount of effort to effectively integrate the two build systems.

2.3.5 Applications

Micro-ROS demos showcase the capabilities and functionalities of Micro-ROS in various use cases and scenarios. Here is the list of the examples taken from [27], where are provided details. However, there does not exist any ”readme” files or any documentation that would describe the demo.

addtwoints_client	fibonacci_action_server	multithread_publisher_subscriber
addtwoints_server	fragmented_publication	parameter_server
autodiscover_agent	fragmented_subscription	ping_pong
complex_msg_publisher	graph_introspection	ping_uros_agent
complex_msg_subscriber	int32_multinode	static_type_handling
configuration_example	int32_publisher	string_publisher
epoch_synchronization	int32_publisher_subscriber	string_subscriber
fibonacci_action_client	int32_subscriber	timer

Table 2.3 Micro-ROS demos

Some of these applications are commonly used with RTOSes; however, not all the supported RTOSes have all the demos integrated. Different RTOSes only have samples and some extra demos to show the integration with more realistic use cases, such as reading data from sensors and publishing it.

[3]Micro-ROS has been integrated with several platform build tools that are listed at https://github.com/micro-ROS/micro_ros_setup.

2.3.6 Multithread Support

As of the time of writing this thesis, there is no official support for multithreaded executors in micro-ROS, unlike ROS 2's *rclcpp* and *rclpy*, which offer multithreading at the executor level. Micro-ROS currently supports only multithreading at the publish/subscribe level, which allows two publishers to publish at the same time to the same session. Micro XRCE-DDS handles synchronization using mutexes. However, dispatching subscribers to their own threads is not yet supported, meaning that only one thread can be dispatched at a time.

Some Micro-ROS's GitHub branches offer support for specific RTOSes, such as NuttX, which is fully POSIX-compliant and thus uses POSIX API for multithreading. However, multithreading is crucial for optimizing performance on multicore embedded devices. Although Micro-ROS's *rclc* GitHub page is used for development, there is currently no multithread support in the official "ros2/rclc" repository. Nevertheless, developers are working on implementing multithreaded executors[4]. Initially, multithreading will be available for POSIX-compliant RTOSes, followed by a solution for non-POSIX operating systems.

[4]There already exists one pull request https://github.com/ros2/rclc/pull/339 related to this issue https://github.com/ros2/rclc/issues/340 that is trying to bring multithreaded executor to micro-ROS (firstly targetted to NuttX).

2.4 Data Distribution Service

In recent times, the use of the Internet of Things has become increasingly popular. As such, there is a need for reliable communication middleware to support the development of distributed systems in IoT. One such protocol is the Data Distribution Service (DDS), which has become an integral part of industrial development today.

This chapter focuses on DDS, its basic concepts and features, how it works, and its advantages. It will also examine the DDS model, which is based on the Publish-Subscribe pattern, and explain how it differs from other messaging patterns. Additionally, the chapter will describe the DDS architecture and its modular design, highlighting the different entities created by the DDS and their functions. Furthermore, it will explore how DDS is involved in the ROS and its significance to the robotics industry. Finally, the chapter will examine micro-ROS, which uses the eXtremely Resource-Constrained Environment XRCE-DDS, and compare it to the standard DDS protocol.

2.4.1 What Is DDS?

DDS is a key data protocol that offers low-latency data connectivity, extreme reliability, and scalability for mission-critical IoT applications. It is a specification of the API of a Publish-Subscribe Communication Middleware for distributed systems that is maintained by the Object Management Group (OMG). DDS is used in a wide range of industries, including telecommunications, defense, data centers, IIoT, robotics, virtualization and cloud computing, energy, healthcare, public and private transportation, and mining. DDS is particularly important for robotics, mainly because this protocol has been selected as the default middleware of the ROS 2, which means to robotics similar what Ubuntu and Linux to computing. [29, 30]

Middleware

In a distributed system, middleware is a type of software that provides services between the application layer and the operating system. For better idea there is an image 2.4 interpreting middleware's position and its content. Furthermore, it simplifies the communication and data exchange and let software developers to focus on the specific purpose of the apllication. [29]

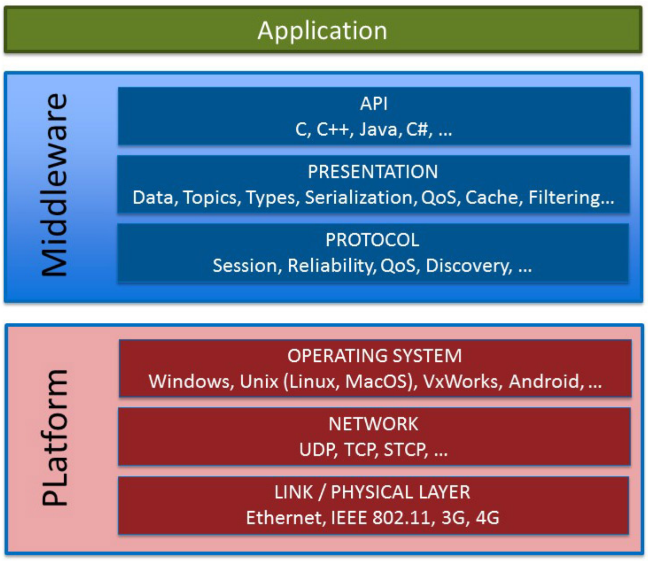

Application
Middleware
API
C, C++, Java, C#, ...
PRESENTATION
Data, Topics, Types, Serialization, QoS, Cache, Filtering...
PROTOCOL
Session, Reliability, QoS, Discovery, ...
PLatform
OPERATING SYSTEM
Windows, Unix (Linux, MacOS), VxWorks, Android, ...
NETWORK
UDP, TCP, STCP, ...
LINK / PHYSICAL LAYER
Ethernet, IEEE 802.11, 3G, 4G

Figure 2.4 The DDS middleware is a software layer that lies between the application layer and operating system, abstracting the application from the details of OS, network transport and low-level data formats. The APIs are provided in different programming languages making it independent from operating system, language used and processor architectures. [29]

2.4.2 Publish-Subscribe Pattern

The Publish-Subscribe (Pub/Sub) pattern is a messaging pattern used in software architecture to facilitate communication between components of a system. In Pub/Sub, publishers broadcast messages to multiple subscribers without needing to know which specific subscribers are interested in the message.

Pub/Sub has a few key benefits, such as decoupling components and improving scalability. By using this pattern, components are not directly dependent on each other, which makes it easier to modify or replace them without impacting other parts of the system. Pub/Sub also allows for a high degree of flexibility in scaling, as new subscribers or publishers can be added as needed without impacting the rest of the system.

Pub/Sub is used in many different types of applications, including real-time data streaming, event-driven architectures, and Internet of Things applications. It provides a way for components to communicate with each other in a flexible and scalable way, making it an important pattern to understand for software developers. [31]

2.4.3 DDS Model

The basic idea of DDS is straightforward: a ”topic” is a name and description for a type of data that we want to share across a distributed system. These topics exist in an abstract space called the ”Global Data Space” in DDS, which is similar to the real implementation because there are usually no intermediate brokers involved.

Entities created by the DDS factory are called publishers and subscribers. Typically, an application creates one participant that includes either a publisher or subscriber, but DDS also defines a third level of entities called Datawriters and Datareaders that are responsible for reading and writing data from the global data space (described in 2.4.5). To simplify the explanation, the focus will be on publishers and subscribers.

DDS automatically discovers remote participants in the system by having each participant send out a multicast announcement or a unicast message to a pre-defined list of peers. DDS then maintains a list of remote destinations for each participant.

This model is decoupled in several dimensions that are taken from [32]:

- **Space** – automatic discovery makes the distributed system independent of the network topology, removing the need to change an application accordingly
- **Time** – publisher can publish asynchronously without checking for subscribers, and late-joiner subscribers receive data if persistence QoS parameters are set to persistent
- **Redundancy** – multiple publishers can share a topic and assigns the publisher with the highest ”strength” parameter as the owner; failover and takeover mechanisms are available if the owner fails, and an unlimited number of subscribers can subscribe to a single topic
- **Platform & Language** – DDS is compatible with a wide range of platforms (including Windows, Linux, Solaris, Aix, Mac Os, Integrity, LynxOS, QNX, VxWorks) and programming languages (such as C, C++, Java, C#, Ada), allowing for heterogeneous distributed systems and relieving developers of concerns regarding platform or language compatibility
- **Implementation** – The DDS specification includes an interoperability protocol named RTPS (details in 2.4.6), allowing different applications within a distributed system to use different DDS implementations and still work together

2.4.4 DDS Architecture

DDS has a straightforward modular design where the DDS infrastructure can be linked to the application as a library without the need for installation of any service or daemon, and the interoperability protocol is implemented on top of the transport layer, making it possible to be implemented over any underlying transport. Typical transports used with DDS include UDP, TCP, and Shared Memory, but users can also add their own transports. Figure 2.5 provides a visual representation for better understanding. [32]

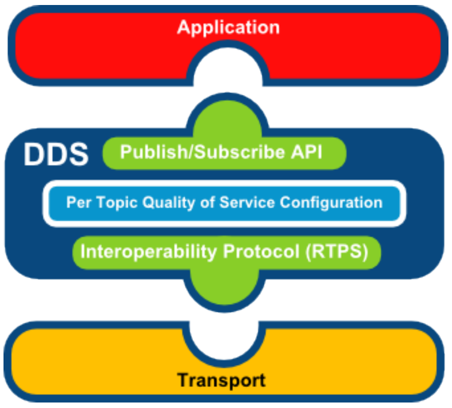

Application
DDS
Publish/Subscribe API
Per Topic Quality of Service Configuration
Interoperability Protocol (RTPS)
Transport

Figure 2.5 DDS architecture provides a simple and modular design. For most implementations, the DDS infrastructure is just a library that can be linked to an application. The interoperability protocol sits on top of the transport layer allowing it to be implemented over any underlying transport. [32]

2.4.5 DDS Key Features

This section examines the essential characteristics of the Data Distribution Service, encompassing data-centricity, a global data space, dynamic discovery, quality of service, and security. DDS stands out among other middleware solutions due to its data-centric approach, which eliminates the need for developers to manage data sharing complexity in their own code. The global data space in DDS serves as a local repository of data and facilitates sharing of data between applications running on different systems and in different languages with low latency. DDS also provides flexible Quality of Service specifications, including reliability, system health, and security. Dynamic discovery enables automatic discovery of publishers and subscribers, while the scalable architecture of DDS allows for its use from small devices to the cloud and very large systems. Finally, DDS provides a range of security options to ensure that sensitive and confidential data remains secure and confidential during transmission without the loss of real-time performance.

Data Centricity

DDS's unique focus on data-centricity makes it particularly well-suited for the Industrial IoT. Unlike other middleware solutions, DDS ensures that each message contains the relevant contextual information needed to properly interpret the data received. With DDS's data-centric approach, programmers can specify how and when data should be shared, with the data values being shared directly. This eliminates the need for developers to manage data sharing complexity in their own code, as DDS implements controlled, managed, and secure data sharing on their behalf.

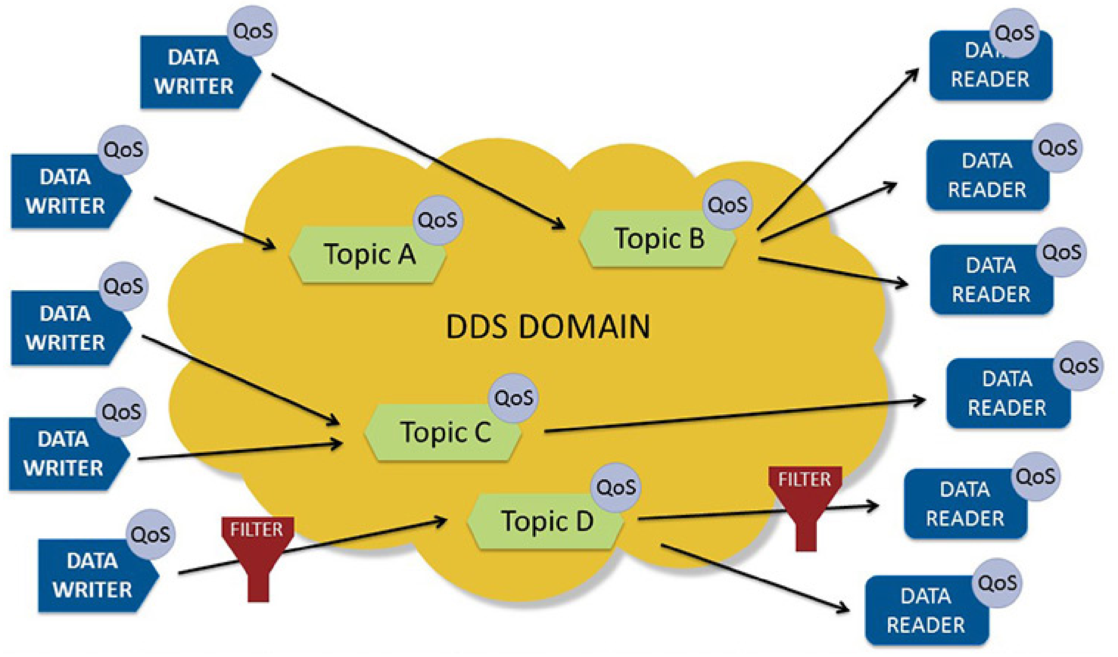
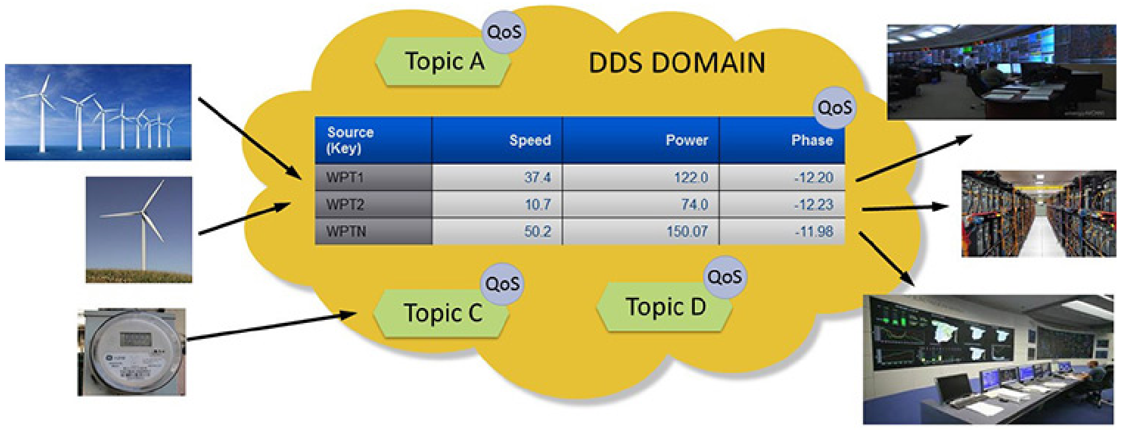


Figure 2.6 DDS enables QoS-controlled data-sharing through topic-based communication, where applications publish and subscribe to topics by name. Subscribers can apply filters to receive only specific data subsets. DDS domains are entirely independent and do not share data across domains. [29]

Global Data Space

DDS uses a ”global data space” as a local repository of data. Applications access the global data space through an API, and DDS sends messages to update relevant stores on remote nodes. This gives the impression that the application has access to the entire global data space, but in reality, each application only stores locally what it needs for as long as it needs it. DDS is designed to manage data in motion and facilitates the sharing of data between applications running on different systems and in different languages with extremely low latency.



Source (Key)	Speed	Power	Phase
WPT1	37.4	122.0	-12.20
WPT2	10.7	74.0	-12.23
WPTN	50.2	150.07	-11.98

Figure 2.7 In DDS, data sharing occurs within Topics and data-objects identified by their Key attributes. This is similar to identifying records in a database. DDS is peer-to-peer and doesn’t rely on servers or cloud to broker data. [29]

Quality of Service

DDS enables flexible data sharing with Quality of Service specifications for reliability, system health, and security. It only sends necessary data to each endpoint and has reliability measures to ensure messages reach their intended destination. DDS can dynamically determine where to send data and efficiently handles large data sizes by filtering and sending only relevant data. It uses multicast messages for fast updates and tracks data versions to translate between them. DDS offers security features such as access control, data flow path enforcement, and on-the-fly data encryption. DDS is most effective when all of these capabilities are used together in demanding and unpredictable environments, with high-speed data transfer.

Dynamic Discovery

Dynamic discovery in DDS allows for automatic discovery of publishers and subscribers without manual configuration, enabling ”plug-and-play” capabilities and extensibility. This feature also discovers the communication characteristics and endpoint attributes, allowing for easy addition of communication participants across different operating systems or hardware platforms without needing to configure IP addresses or machine architectures.

Scalable Architecture

The architecture of OMG DDS is devised to be adaptable for small and large systems, ranging from small devices to the cloud. It has the capability to enable the Internet of Things by scaling across a massive number of participants, delivering data at exceptionally high speeds, managing numerous data objects, and ensuring maximum availability and security. DDS minimizes the intricacy of distributed system development by consolidating much of the complexity into a single, standardized communication layer.

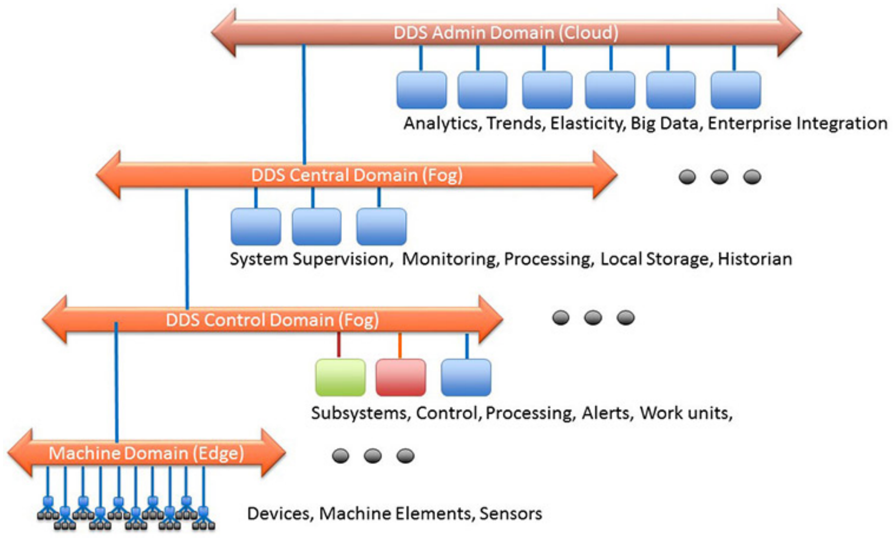

DDS Admin Domain (Cloud)
Analytics, Trends, Elasticity, Big Data, Enterprise Integration
DDS Central Domain (Fog)
System Supervision, Monitoring, Processing, Local Storage, Historian
DDS Control Domain (Fog)
Subsystems, Control, Processing, Alerts, Work units,
Machine Domain (Edge)
Devices, Machine Elements, Sensors

Figure 2.8 DDS spans Edge to Cloud, supporting high-speed machine-to-machine communications at the edge, robust QoS in intermediary systems, and scalable access to information in the cloud. [29]

Security

Security is a critical feature in DDS, particularly for systems that handle sensitive or confidential data. DDS provides a range of security options, including access control, authentication, encryption, and data integrity. These security measures ensure that only authorized nodes can access and modify data in the system, and that the data remains secure and confidential during transmission all without the loss of real-time performance. [29]

2.4.6 What Is RTPS?

The Real-Time Publish-Subscribe protocol is a communication protocol used for publishing and subscribing to information in a best effort and reliable manner. It can function over unreliable transports like UDP, and can be used for both unicast and multicast communication. RTPS provides an open standard for the exchange of real-time data between distributed systems, supporting a wide range of application domains such as robotics, aerospace, transportation, and the Industrial Internet of Things (IIoT).

Built on top of the Data Distribution Service standard, RTPS defines the application-level publish-subscribe communication model and provides a standardized wire protocol[5]. This protocol facilitates interoperability between different vendor implementations of DDS, enabling seamless communication between systems.

RTPS supports the discovery of new participants, the management of data flow between participants, and the exchange of real-time data with a range of Quality of Service (QoS) levels. It provides mechanisms for reliability, scalability, and fault tolerance, making it suitable for use in strict real-time systems. [34]

RTPS Advantages

The Real-Time Publish-Subscribe protocol has several advantages according to [34]:

- **Scalability** – is designed to scale to large systems with many participants, allowing efficient communication between them
- **Interoperability** – different DDS vendors can use RTPS as their interoperability protocol, ensuring that systems built with different DDS implementations can communicate with each other
- **Real-time communication** – provides reliable and timely delivery of data, even in distributed real-time systems with strict timing requirements
- **Extensibility** – allows for custom extensions to be added to the protocol, enabling users to tailor the communication to their specific needs
- **Platform independence** – is independent of the underlying transport and can be implemented on a variety of platforms, including embedded systems, desktop computers, and cloud servers
- **QoS support** – provides a rich set of QoS policies that allow users to control the reliability, latency, and bandwidth of the communication

2.4.7 RTPS Architecture

The architecture consists of four modules: **Structure**, **Message**, **Behavior**, and **Discovery**. Each module has a specific role in controlling the exchange of information.

[5]There is no exact definition of a wire protocol. However, there exists one that quite precisely explain what does it means. The encyclopedia of PCMAG (available from [33]) says: "In a network, a wire protocol is the mechanism for transmitting data from point a to point b. The term is a bit confusing, because it sounds like layer 1 of the network, which physically places the bits 'onto the wire.' In some cases, it may refer to layer 1; however, it generally refers to higher layers, including Ethernet and ATM (layer 2) and even higher layer distributed object protocols such as SOAP, CORBA or RMI."

Structure Module

The Structure module defines the communication endpoints and maps them to their DDS counterparts. It establishes a separate communication plane for each RTPS domain and contains a set of Participants. Each Participant can contain multiple local Endpoints of two different kinds: Writers and Readers. These endpoints exchange information in the RTPS network by sending RTPS messages. The interface between the RTPS endpoints and their corresponding DDS entities is the HistoryCache. The information exchanged between the endpoints is usually stored in a CacheChange. To illustrate, when a write operation occurs, it creates a CacheChange in the Writer History. Afterward, the RTPS Writer transmits an RTPS message to all matching Readers. When the RTPS Reader receives the message, it adds the CacheChange to its corresponding HistoryCache and informs the DDS entity that new data is now accessible.

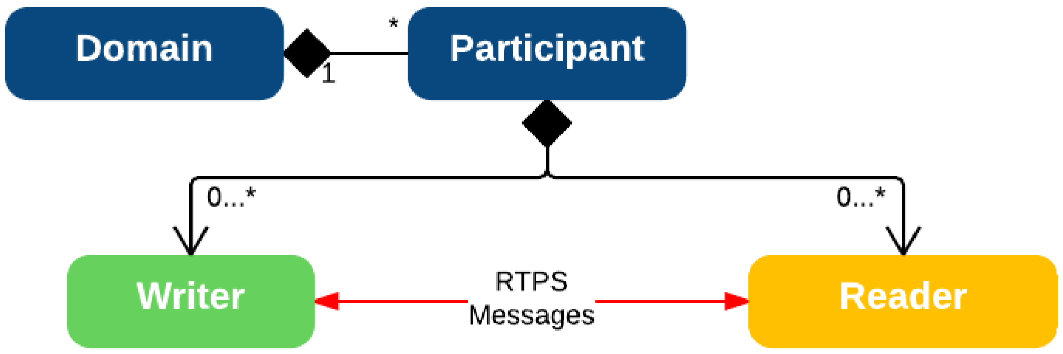


Figure 2.9 RTPS Participants can have multiple Writers and Readers which exchange information through RTPS messages. [34]

Message Module

The Message module defines the content of the atomic information exchanges between RTPS Writers and Readers. RTPS Messages are composed of a header followed by a number of submessages. The submessages allow the vocabulary of messages to be extended while maintaining backward compatibility. The three most important messages are DATA, HEARTBEAT, and ACKNACK.

Message	Direction	Description
DATA	Writer → Reader	Conveys details about a change in a data object associated with the Writer, such as new information being added or an update in the data object's life cycle.
HEARTBEAT	Writer → Reader	Indicating the CacheChanges that are available at the current moment
ACKNACK	Reader → Writer	Permits the Reader to notify the Writer about the received and missing changes. It supports both positive and negative acknowledgements.

Table 2.4 The most important RTPS messages

Behavior Module

The Behavior module describes the valid messages exchanges that can occur between a Writer and a Reader. It also defines the changes in the state of the Writer and Reader depending on each message. These rules ensure interoperability between different implementations.

Discovery Module

The Discovery module describes the protocol that enables Participants to obtain information about the existence and attributes of all the other Participants and Endpoints in the Domain. This information exchange is called metatraffic. The discovery protocol is divided into two layers: Participant Discovery Protocol (PDP) and Endpoint Discovery Protocol (EDP). The PDP specifies how the Participants discover each other. Upon discovery, the Participants exchange information about their endpoints using the EDP. Different vendors may implement multiple discovery protocols, however to ensure interoperability one PDP and one EDP must be implemented by all vendors. The discovery mechanism allows simple plug and play connectivity without requiring any configuration by the user. [34]

2.4.8 ROS 2 & DDS

Data Distribution Service is the underlying communication protocol used by ROS 2's middleware layer to facilitate communication between nodes. DDS is a popular middleware protocol for real-time, distributed systems, and ROS 2's adoption of DDS provides a number of benefits.

ROS 2's middleware layer, called the ROS MiddleWare (*rmw*), provides an abstraction layer for communication between nodes. The *rmw* layer is responsible for managing the communication interfaces, while the DDS protocol is responsible for managing the actual data distribution.

ROS 2 provides support for multiple DDS vendors, allowing users to choose the vendor that best suits their needs. ROS 2's vendor-independent architecture allows for easy integration with a wide range of DDS-based systems and supports the development of interoperable systems.

Overall, the use of DDS in ROS 2 enables real-time, scalable, and interoperable communication between nodes, making it a key component of the ROS 2 ecosystem.

ROS 2 DDS/RTPS Vendors

DDS is an industry standard that is implemented by several vendors, including RTI's Connext DDS, eProsima's Fast DDS, Eclipse's Cyclone DDS, and GurumNetworks's GurumDDS. RTPS is the wire protocol used by DDS to communicate over the network.

ROS 2 supports multiple DDS/RTPS implementations, as choosing the right vendor or implementation depends on several factors, such as the license, platform availability, or computation footprint. Vendors may offer different DDS or RTPS implementations for specific purposes, like RTI's Connext implementation for microcontrollers or for applications requiring special safety certifications.

To use a DDS/RTPS implementation with ROS 2, a "ROS Middleware interface" (*rmw* interface) package must be created to implement the abstract ROS middleware interface using the DDS or RTPS implementation's API and tools. Creating and maintaining *rmw* packages for supporting DDS implementations is a lot of work, but it is essential to support at least a few implementations to ensure that the ROS 2 codebase is not tied to any one particular implementation. Users may want to switch out implementations depending on their project's needs. [35]

Product name	License	*Rmw* implementation	Status
eProsima Fast DDS	Apache 2	rmw_fastrtps_cpp	Full support. Default *rmw*. Packaged with binary releases.
Eclipse Cyclone DDS	Eclipse Public License v2.0	rmw_cyclonedds_cpp	Full support. Packaged with binary releases.
RTI Connext DDS	commercial, research	rmw_connextdds	Full support. Support included in binaries, but Connext installed separately.
GurumNetworks GurumDDS	commercial	rmw_gurumdds_cpp	Community support. Support included in binaries, but GurumDDS installed separately.

Table 2.5 Supported RMW implementations [35]

Multiple *rmw* Implementations

The default *rmw* implementation in ROS 2 binary releases is Fast DDS, which comes pre-installed. However, other *rmw* implementations like Cyclone DDS, Connext DDS, and GurumDDS can be enabled by installing additional packages, without requiring any rebuilds or replacement of existing packages. Multiple *rmw* implementations can also be built and installed simultaneously in a ROS 2 workspace that is built from source, as long as the relevant DDS/RTPS implementation is installed properly and relevant environment variables are configured. If the *rmw* package for a specific DDS implementation is present in the workspace, it will be built during the compilation of the ROS 2 code if the corresponding DDS implementation is also installed.

Many instances demonstrate that nodes utilizing various *rmw* implementations can communicate with each other, although this isn't universally valid. According to the documentation [35], the following inter-vendor communication configurations are not supported:

Rmw implementations	Limitation
Fast DDS ↔ Connext	WString published by Fast DDS can not be received correctly by Connext on macOS
Connext ↔ Cyclone DDS	does not support pub/sub communication for WString

Table 2.6 *Rmw* inter-vendor communication configurations that are not supported

2.4.9 Micro-ROS & DDS

The use of microcontrollers in robotics and IoT has rapidly grown in recent years, and with it comes the need for a lightweight middleware solution that can support publisher/subscriber communication architectures. Micro XRCE-DDS (described below in this subsection) has emerged as a promising option for such applications, as it is designed to work within the constraints of microcontrollers, providing low resource consumption and multi-transport support. This has led to its adoption by companies such as Renesas and ROBOTIS. Additionally, Micro XRCE-DDS has been selected as the default middleware layer for the micro-ROS project, which aims to bring the capabilities of the Robot Operating System 2 to microcontrollers. This section examines the integration of Micro XRCE-DDS with micro-ROS and the benefits of this combination in the development of robotic and IoT applications on microcontrollers. [36]

XRCE

The e**X**tremely **R**esource-**C**onstrained **E**nvironment is a lightweight communication protocol, which provides a means of communicating data from resource-constrained environments to larger systems that use the DDS standards. The DDS-XRCE wire protocol has been standardized by the OMG.

The XRCE architecture is designed to enable resource-constrained devices to communicate with a DDS network through an XRCE Agent. Image 2.10 illustrates the communication protocols used and the architecture scheme that is composed of three main components:

- **XRCE Client** is responsible for generating requests and receiving responses to communicate with the XRCE Agent. It is implemented in the application running on the resource-constrained device.
- **XRCE Agent** is an intermediate service that sits between the XRCE Client and the DDS network. It is responsible for translating the XRCE Client's requests into DDS protocol messages and vice versa. The XRCE Agent enables the XRCE Client to publish and subscribe to topics in the DDS network, as well as manage its DDS entities, such as topics, publishers, and subscribers.
- **DDS network** is the final destination for the data exchanged between the XRCE Client and other DDS entities. It is responsible for routing data to the appropriate subscribers and maintaining the quality of service required by the system. [37]

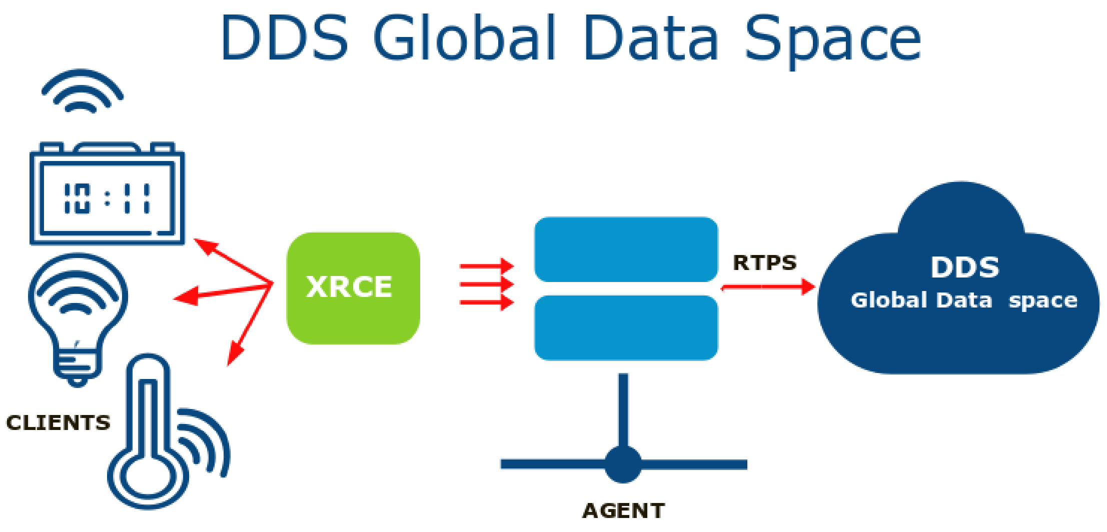


Figure 2.10 The DDS-XRCE protocol allows for the seamless integration of resource-constrained devices into DDS-based systems, enabling them to communicate data efficiently with larger systems. Its architecture is divided into three parts – clients, agent and DDS network [37]

eProsima Micro XRCE-DDS

Micro XRCE-DDS is a small-footprint implementation of the DDS-XRCE protocol that targets resource-constrained devices and microcontrollers. It is designed to provide a lightweight and efficient communication framework for distributed systems that require real-time performance and reliable data exchange.

Micro XRCE-DDS is developed and maintained by eProsima, a company that specializes in middleware solutions for distributed systems. The implementation is based on the eProsima's Fast DDS library, which provides a full-featured DDS implementation for more powerful devices.

Micro XRCE-DDS provides a client-server architecture for connecting resource-constrained devices to a larger DDS network. The client library is designed to run on resource-constrained devices, such as microcontrollers or embedded systems, and provides a simplified API for publishing and subscribing to DDS topics. The server component runs on a more powerful device and acts as a bridge between the DDS network and the client devices.

One of the main features of Micro XRCE-DDS is its small memory footprint, which makes it suitable for deployment on devices with limited resources. The implementation is highly configurable, allowing developers to fine-tune the performance and memory usage of the library to suit their specific needs.

Micro XRCE-DDS is compatible with the ROS 2 ecosystem and can be used as a middleware implementation for ROS 2 nodes. It provides a ROS 2 interface that allows ROS 2 applications to communicate with resource-constrained devices using DDS. This makes it possible to build distributed robotic systems that incorporate both high-performance computing platforms and low-power, embedded devices.

Key Features

Micro XRCE-DDS is a middleware, designed for microcontroller applications, that offers several key features according to [36]:

- **Low resource consumption**

  Micro XRCE-DDS is optimized for microcontroller applications with limited memory, making it highly efficient in terms of memory consumption. The latest version of this library has a memory consumption of less than 75 KB of Flash memory and around 3 KB of RAM, which is ideal for low-power devices. The middleware is also highly configurable, allowing users to customize the library's size and features based on their application requirements.

- **Multi-transport support**

  Micro XRCE-DDS supports multiple transport protocols natively, including UDP, TCP, and a custom Serial transport protocol. This feature makes it highly versatile and suitable for use in a wide range of applications.

- **Multi-platform support**

  The XRCE Client supports FreeRTOS, Zephyr, and NuttX as embedded RTOS, as well as Windows and Linux. The XRCE Agent, on the other hand, supports Windows and Linux. This feature allows for the middleware's use in a wide range of platforms and applications.

- **QoS support**

  Micro XRCE-DDS allows users to create DDS entities in the XRCE Agent either by XML or reference. The XML approach enables users to create entities in Reliable or Best-Effort mode, while the reference approach reduces the memory consumption of the Client inside the MCU. The ability to write custom XML QoS as in DDS enables users to tailor the middleware to their specific requirements.

2.5 PXROS-HR

This section focuses on the exploration of PXROS-HR, a commercial real-time operating system that is designed to work with advanced multi-core MCUs. The section begins with an overview of real-time operating systems, outlining their key characteristics. Following that, the section proceeds to introduce PXROS-HR, where the main features are discussed, and how it differs from other real-time operating systems. Finally, the section delves into the special features of PXROS-HR, which include hardware-based memory protection and object-based system architecture. The objective of this section is to offer a comprehensive understanding of PXROS-HR, emphasizing its strengths and capabilities for use in safety-critical applications.

2.5.1 HighTec

HighTec EDV Systeme is a privately owned company that was established in 1982 and is now the world's recognized commercial open source compiler vendor. With offices in Germany (headquarter), Czech Republic, Hungary and China, HighTec offers reliable and secure tools for embedded software development, ensuring independence for the future. The HighTec C/C++ compiler is portable and always available for the latest chip revisions for our supported architectures ahead of general release.

HighTec has been assigned as the Preferred Design House[6] by Infineon and the Preferred Compiler Partner by STMicroelectronics. Cooperation agreements with semiconductor vendors ensure the long-term availability of HighTec's development toolsets and guarantee automotive-grade support, including frozen version support and bug scanning compilers. [39]

HTC IDE

The Hightec IDE is an Eclipse-based software development environment designed for embedded systems. It provides GUI toolchain configuration, including a compiler, assembler, linker, and debugger, along with project management capabilities and support for programming languages like C and C++. The IDE supports a variety of microcontroller architectures, such as ARM, PowerPC, and TriCore, among others, making it a versatile and valuable tool for embedded software developers working with HighTec's products.

PxNet

PxNet is a module that enables TCP/IP communication by implementing a task with a specific API primarily used for TCP and UDP communication. SEVENSTAX company provides the TCP/IP stack, which HighTec adapted for use in PXROS-HR applications. The PxNet STX package includes the PxNet API module, which implements the application programming interface. Application tasks can access PxNet services by calling functions from the PxNet API. [40]

2.5.2 What Is RTOS

A Real-time Operating System is a specialized operating system designed to ensure real-time applications meet specific deadlines. They are critical for applications with timing-specific requirements and other critical systems. RTOS has similar functions as general-purpose operating systems like Windows, macOS, or Linux, but with a scheduler that meets specific deadlines for different tasks. They are typically used in embedded systems for real-time environments. RTOS

[6] As PDH, HighTec provides Basic Support Services free of charge, operating as a generic technical help-desk to the customers for AURIX related topics. Design services or topics requiring substantial design or R&D effort are also delivered by HighTec, as part of HighTec Premium Services. [38]

handles multiple processes and ensures events are responded to within predictable time limits, providing multitasking functionality, prioritization of process threads, and sufficient interrupt levels. These operating systems are used in various industries, including air traffic control systems, anti-lock brakes, medical systems, and cameras. RTOS has unique opportunities and challenges for research in computer science, electrical engineering, and related fields, with a modular and scalable architecture that allows for customization to meet specific application requirements. [41]

Real-time operating systems are designed to not only accurately perform a calculation process but also deliver the result within a predictable timeframe. Typically, two categories of real-time are identified (taken from [42]):

- **Hard real-time** – demands that all time limits be strictly met, as a single delay could result in an unusable outcome
- **Soft real-time** – more lenient and allows for a few time limits to be exceeded in specific instances without rendering the result unusable

Key Characteristics

Real-time operating systems generally have the following characteristics (taken from [41]):

- Small footprint

  Real-time operating systems are designed to have a small footprint, making them lightweight in comparison to general OSes.
- High performance

  RTOSes are known for their high performance, being fast and responsive.
- Determinism

  Determinism is a key characteristic of RTOSes, ensuring that repeating inputs result in the same output.
- Safety and security

  Safety and security are the highest priorities in RTOS development, as they are frequently used in critical systems.
- Priority-based scheduling

  RTOSes use priority-based scheduling to ensure high-priority tasks are executed before lower-priority ones.
- Timing information

  RTOSes are responsible for timing and providing an API.

2.5.3 What Is PXROS-HR

PXROS-HR (Portable eXtendible Real-time Operating System – High Reliability) is a real-time operating system designed for advanced multi-core MCUs. It features a modern micro-kernel and hardware memory protection mechanisms (Memory Protection Unit – MPU) for improved encapsulation and robustness. The latest version of PXROS-HR supports fine-grained hardware memory protection mechanisms (MPU) available in modern microcontrollers like the AURIX.

PXROS-HR is officially safety approved for safety-critical applications up to SIL 3 (IEC61508) and ASIL D (ISO 26262), making it ideal for industrial and automotive applications where safety is a top priority. The certification was issued by TÜV-Nord Systems GmbH & Co. KG.

Developed with the HighTec C/C++ compiler for TriCore/AURIX, PXROS-HR integrates with Infineon's MCAL and SafeTlib software frameworks. It is a non-AUTOSAR based and highly optimized for the TriCore architecture, providing multi-core support for the AURIX family. [43]

2.5.4 PXROS-HR Special Features

In comparison to other Real-time Operating Systems, PXROS-HR has some special features. These features are mentioned in [42].

Hardware-based Memory Protection

PXROS-HR implements a hardware-based memory protection mechanism to restrict memory access between different tasks. Each task is allocated a specific amount of memory, and the hardware's MPU detects any unauthorized access attempts outside of this memory area, triggering a trap that transfers control to the operating system.

Permission Concepts for Tasks

PXROS-HR manages the permissions of tasks by granting or denying specific rights, such as access to operating system or hardware resources. Furthermore, the data exchanged via messages are secured by the MPU to ensure task protection.

Object-based System Architecture

In PXROS-HR, all managed elements are treated as objects, including message objects and task objects.

No Interrupt Locks

Unlike other operating systems, PXROS-HR doesn't use interrupt locks during scheduling. This allows for immediate reaction to interrupts without latency, significantly improving system predictability.

Message-based Data Interchange

Task data is exchanged through message objects and mailboxes, providing implicit synchronization and restricting data access to the current message user. This eliminates the need for explicit synchronization mechanisms, such as semaphores.

Encapsulation

In PXROS-HR, the object-based approach and hardware-assisted memory protection facilitate the concept of encapsulation. A task is considered as an independent capsule that is isolated from the outside world. Interaction with the external world is possible only through a well-defined and narrow interface using message objects. Hardware prevents accidental interference by restricting access operations. The system consists of several capsules, allowing for manageable connections within the system while ensuring unobtrusive behavior.

Reloading Tasks

With the concept of encapsulation, PXROS-HR enables the loading and unloading of new tasks during system runtime dynamically.

Debugging The Running System

PXROS-HR allows individual tasks to be stopped within a running system and debugged, while the overall system continues to operate.

Difference Between System Tasks and Application Tasks

The permission concept in PXROS-HR allows distinguishing between system tasks and application tasks. System tasks have comprehensive privileges and offer services to application tasks that require a higher privilege level, thus forming a system platform that provides additional functionality to applications. Application tasks, which typically implement application-specific functionalities, have lower privileges and can be reloaded as needed.

Micro-kernel

PXROS-HR is implemented as a micro-kernel, meaning that only the essential functionalities of the operating system are implemented in the kernel. Additional functionalities, such as the TCP/IP stack or the file system, are realized as separate modules running as application tasks.

Chapter 3

Proposed Solution and Implementation

This thesis aims to develop a prototype that integrates PXROS-HR into micro-ROS. The current distributions supported for micro-ROS include Foxy Fitzroy and Humble Hawksbill. While Foxy Fitzroy offers more stability with code fixes, it will reach end-of-life soon, and therefore, the prototype will be based on Humble Hawksbill, which has a longer support duration of five years.

The integration will be achieved by creating a prototype based on the HighTec PxNet package that encapsulates the TCP/IP stack. The UDP communication protocol will facilitate data exchange between the micro-ROS agent and the client. The HighTec *TC39x PxNet base example*, available in the HighTec content manager[1], will serve as the foundation for the prototype. It will be aligned with the HighTec project structure, and irrelevant elements will be removed. The Application Kit TC3X7 version 2.0 with TC397 B-Step will fulfill the requirements of the base example. The HighTec IDE and toolchain for TriCore version 4.9.4.1 will be utilized for managing, configuring, and building the prototype. The micro-ROS library will be created with the same toolchain as a custom static library and added to the prototype with the set of headers.

The project will use the hardware debugger from PLS and the Universal Debug Engine (UDE) GUI for programming and debugging. Additionally, the debug logs from the micro-ROS demos will be displayed in a terminal by connecting the development board to a computer via UART communication.

The chapter begins by explaining the process of building the micro-ROS library, which involves creating a custom static library and modifying the ROS 2 and micro-ROS code. Subsequently, the chapter outlines the implementation of multicore mutexes (based on PXROS-HR API) in micro-ROS, which involves the development of mutex API functions to ensure basic mutex functionality, configuration and creation of mutex task. The project structure is then described, which includes the HighTec project structure, project configurations, linker file modifications required for the hardware MPU configuration, and task description, configuration and distribution. Furthermore, the chapter covers the implementation of various micro-ROS demos, including implementations of custom allocators and custom transport functions. There is a particular focus on multithread publisher-subscriber, and a newly created multicore publisher-subscriber that highlights the potential of PXROS-HR as a multicore RTOS.

Overall, this chapter is an essential resource for anyone interested in integrating PXROS-HR into micro-ROS.

[1]"The HighTec Content Manager provides you with easy access to the HighTec's resource cloud repository. You find there project examples, templates and other documents that help you to get up to the speed quickly."[44]

3.1 Building Micro-ROS Library

As far as the external build system is used for building the thesis example, a custom static library will be built. There is an official tutorial on how to create a custom static library. Before building the library, the tutorial needs some prerequisites to be accomplished. All of them are described quite in detail on the official micro-ROS pages or in ROS 2 documentation. Here are the following steps how to successfully get to creating custom static library:

1. Download and install the ROS 2 sources for the Humble version [45]
2. Download and install the micro-ROS build system for the Humble version [46]
3. Follow the tutorial for creating a custom static library [47]

3.1.1 Creating Custom Static Library

The tutorial instructs users to create two files that specify the cross-compilation requirements for building a custom static library and a set of header files:

1. CMake toolchain file – contains the toolchain configuration
2. Colcon[2] meta file – contains the micro-ROS library configuration

The tutorial provides example versions of both files, which are modified as required for this thesis. The toolchain file used for the HighTec TriCore toolchain version 4.9.4.1 is provided below:

```
set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_SYSTEM_PROCESSOR tricore)
set(CMAKE_CROSSCOMPILING 1)
set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)
set(PLATFORM_NAME "PXROS")
set(PXROS 1)

set(CMAKE_SYSROOT ~/ros2_humble/microros_ws)

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)

# Makefile flags
set(CROSSDEV /mnt/hgfs/HighTec/toolchains/tricore/v4.9.4.1-linux/bin/tricore-)
set(ARCH_CPU_FLAGS "-mcpu=tc39xx -DAPPKIT_TC3X7_V2_0 -O0 -Wall -fdata-sections -
    ffunction-sections -g2")
set(ARCH_OPT_FLAGS "")

# Compiler tools
foreach(tool gcc ld ar)
    string(TOUPPER ${tool} TOOL)
    find_program(${TOOL} ${CROSSDEV}${tool})
    if(NOT ${TOOL})
        message(FATAL_ERROR "could not find ${CROSSDEV}${tool}")
    endif()
endforeach()

set(CMAKE_C_COMPILER ${CROSSDEV}gcc)
set(CMAKE_CXX_COMPILER ${CROSSDEV}g++)
```

[2]"Colcon – collective construction. Colcon is a command line tool to improve the workflow of building, testing and using multiple software packages. It automates the process, handles the ordering and sets up the environment to use the packages." [48]

```

set(CMAKE_C_FLAGS_INIT "-std=c99 ${ARCH_CPU_FLAGS} ${ARCH_OPT_FLAGS}" CACHE
    STRING "" FORCE)
set(CMAKE_CXX_FLAGS_INIT "${ARCH_CPU_FLAGS} ${ARCH_OPT_FLAGS} " CACHE STRING ""
    FORCE)

include_directories(SYSTEM
    /mnt/hgfs/HighTec/pxros-hr/tricore/v8.2.0/kernel/include
    /mnt/hgfs/MutexServer
    )
```

Code listing 3.1 CMAKE toolchain configuration file needed to build a custom static micro-ROS library in *my_custom_toolchain.cmake.*

The example colcon meta file was modified in the following ways:

- The option ”-DUCDR_PIC=OFF”, which controls position independent code in the Micro-CDR project, was set to *OFF*. By default, this option is set to *ON* in Micro-CDR's *CMakeLists.txt* file:

  ```
  option(UCDR_PIC "Control Position Independent Code." ON)
  ```

- The option ”-DUCLIENT_PROFILE_STREAM_FRAMING=OFF” needed to be set to *ON* only when non-packet communication is used. In this thesis, UDP communication is used, which is packet-oriented. However, setting this flag to *ON* should not cause any problems, as the custom transport can later be set to use framing or not. Disabling this option can save some memory, as the framing functions will not be included. This is one way to reduce the memory needed for the entire library.
- The option ”-DUCLIENT_PROFILE_MULTITHREAD=ON” must be set to true for further multithread and multicore demo examples.

The modified projects and their configurations in the example colcon meta file are:

```
"microcdr": {
    "cmake-args": [
        "-DUCDR_PIC=OFF"
    ]
},
"microxrcedds_client": {
    "cmake-args": [
        "-DUCLIENT_PIC=OFF",
        "-DUCLIENT_PROFILE_UDP=OFF",
        "-DUCLIENT_PROFILE_TCP=OFF",
        "-DUCLIENT_PROFILE_DISCOVERY=OFF",
        "-DUCLIENT_PROFILE_SERIAL=OFF",
        "-DUCLIENT_PROFILE_STREAM_FRAMING=OFF",
        "-DUCLIENT_PROFILE_CUSTOM_TRANSPORT=ON",
        "-DUCLIENT_PROFILE_MULTITHREAD=ON"
    ]
},
```

Code listing 3.2 Modified projects and their configuration in the custom colcon meta file in *my_custom_colcon.meta.*

3.1.2 ROS 2 and Micro-ROS Code Modifications

PXROS-HR is not POSIX-compliant operating system, which means that several functions that micro-ROS or ROS 2 relies on will not work, such as the time function. This section describes the modifications and additions that were made to the code to enable micro-ROS and ROS 2 to function on PXROS-HR.

- Time function

  To address the issue with the time function, modifications were made to the library. The library contained two time function implementations – one for WIN32 systems and the other for the rest. To add support for PXROS-HR, an implementation of the time functions needed in the library was added, named *time_pxros.c* to align with the naming convention of the other implementations.

  The added *time_pxros.c* file contains implementations for the *rcutils_system_time_now* and *rcutils_steady_time_now* functions. These functions retrieve the current system time and the current steady time, respectively, using the *PxTickGetTimeInMilliSeconds* function provided by the PXROS-HR operating system.

  The time function implementation selection was modified in *uros/rcutils/CMakeLists.txt* as follows:

```
if(WIN32)
  set(time_impl_c src/time_win32.c)
elseif(PXROS)
  set(time_impl_c src/time_pxros.c)
else()
  set(time_impl_c src/time_unix.c)
endif()
```

Code listing 3.3 Selection of time functions implementation based on the underlying operating system in *uros/rcutils/CMakeLists.txt.*

Here is the content of the added *time_pxros.c* implementation of the time functions in *uros/rcutils/src/time_pxros.c* file:

```
#ifdef __cplusplus
extern "C"
{
#endif

#include "pxdef.h"
#include "rcutils/time.h"
#include <unistd.h>

#include "./common.h"
#include "rcutils/allocator.h"
#include "rcutils/error_handling.h"

rcutils_ret_t
rcutils_system_time_now(rcutils_time_point_value_t * now)
{
  RCUTILS_CHECK_ARGUMENT_FOR_NULL(now, RCUTILS_RET_INVALID_ARGUMENT);

  *now = RCUTILS_MS_TO_NS(PxTickGetTimeInMilliSeconds());

  return RCUTILS_RET_OK;
}

rcutils_ret_t
rcutils_steady_time_now(rcutils_time_point_value_t * now)
{
  RCUTILS_CHECK_ARGUMENT_FOR_NULL(now, RCUTILS_RET_INVALID_ARGUMENT);

  *now = RCUTILS_MS_TO_NS(PxTickGetTimeInMilliSeconds());

  return RCUTILS_RET_OK;
}

#ifdef __cplusplus
}
#endif
```

Code listing 3.4 PXROS-HR time functions implementation in *uros/rcutils/src/time_pxros.c.*

- Unused and not provided header file

  Additionally, the include statements in the *process.c* file were modified by adding a conditional statement to exclude the unused *libgen.h* header file, which is not provided, when building with PXROS-HR. The modified conditional statement is shown below:

```
#if defined _WIN32 || defined __CYGWIN__
// When building with MSVC 19.28.29333.0 on Windows 10 (as of 2020-11-11),
// there appears to be a problem with winbase.h (which is included by
// Windows.h).  In particular, warnings of the form:
//
// warning C5105: macro expansion producing 'defined' has undefined behavior
//
// See https://developercommunity.visualstudio.com/content/problem/695656/wdk
    -and-sdk-are-not-compatible-with-experimentalpr.html
// for more information.  For now disable that warning when including windows
    .h
#pragma warning(push)
#pragma warning(disable : 5105)
#include <Windows.h>
#pragma warning(pop)
#else
#  if !defined __HIGHTEC__
#    include <libgen.h>
#  endif
#include <unistd.h>
#endif
```

Code listing 3.5 Conditional include of unused header (when using HighTec compiler) in *uros/rcutils/src/process.c*.

- eProsima

  The integration of Micro XRCE-DDS into the PXROS-HR operating system required significant modifications, including the implementation of time functions, the addition of PXROS to operating systems, and the implementation of mutexes for the multithreading profile.

  Initially, PXROS-HR was added to the platforms in the *eProsima/Micro-XRCE-DDS-Client /CMakeLists.txt* file as shown below:

```
# Check platform.
if(CMAKE_SYSTEM_NAME STREQUAL "Linux" OR CMAKE_SYSTEM_NAME STREQUAL "Android"
    )
    set(UCLIENT_PLATFORM_LINUX ON)
elseif(CMAKE_SYSTEM_NAME STREQUAL "Windows")
    set(UCLIENT_PLATFORM_WINDOWS ON)
elseif(CMAKE_SYSTEM_NAME STREQUAL "Darwin")
    set(UCLIENT_PLATFORM_MACOS ON)
elseif(CMAKE_SYSTEM_NAME STREQUAL "Generic")
    ...
    elseif(PLATFORM_NAME STREQUAL "PXROS")
        set(UCLIENT_PLATFORM_PXROS ON)
    endif()
endif()
```

Code listing 3.6 Simplified code snippet showing the addition of PXROS-HR to platforms in *eProsima/Micro-XRCE-DDS-Client/CMakeLists.txt*.

  The CMake define was added to the header of the template configuration file located at *eProsima/Micro-XRCE-DDS-Client/include/uxr/client/config.h.in*:

```
#cmakedefine UCLIENT_PLATFORM_POSIX
#cmakedefine UCLIENT_PLATFORM_POSIX_NOPOLL
#cmakedefine UCLIENT_PLATFORM_WINDOWS
#cmakedefine UCLIENT_PLATFORM_FREERTOS_PLUS_TCP
```

```
#cmakedefine UCLIENT_PLATFORM_RTEMS_BSD_NET
#cmakedefine UCLIENT_PLATFORM_ZEPHYR
#cmakedefine UCLIENT_PLATFORM_PXROS
```

Code listing 3.7 Add CMAKE define of the platform for PXROS-HR in configuration header template in *eProsima/Micro-XRCE-DDS-Client/include/uxr/client/config.h.in*.

To implement the time function in the *eProsima/Micro-XRCE-DDS-Client/src/c/util/time.c*, the *pxdef.h* header file is included:

```
#ifdef WIN32
#include <Windows.h>
#elif defined(UCLIENT_PLATFORM_FREERTOS_PLUS_TCP)
#include "FreeRTOS.h"
#include "task.h"
#elif defined(UCLIENT_PLATFORM_PXROS)
#include "pxdef.h"
#endif /* ifdef WIN32 */
```

Code listing 3.8 *pxdef.h* include for time function implementation in *eProsima/Micro-XRCE-DDS-Client/src/c/util/time.c*.

Inside the function *uxr_nanos*, the PXROS-HR API function *PxTickGetTimeInMilliSeconds* was called to obtain the current time, which is returned in milliseconds and then transformed to nanoseconds:

```
...
#elif defined(UCLIENT_PLATFORM_PXROS)
    return (int64_t) PxTickGetTimeInMilliSeconds() * 1000000;
#else
...
```

Code listing 3.9 Time function *uxr_nanos* implementation using PXROS-HR API in *eProsima/Micro-XRCE-DDS-Client/src/c/util/time.c*.

eProsima's Micro XRCE-DDS also provides a multithreading profile that includes APIs for mutex implementation (implementation details are in section 3.2). These APIs enable users to implement mutexes according to their specific operating systems or platforms.

Below are simplified code samples that demonstrate the implementation of eProsima's mutexes:

```
void uxr_init_lock(
        uxrMutex* mutex)
{
    ...
#elif defined(UCLIENT_PLATFORM_PXROS)
    PxInitLock(&mutex->impl);
#else
    ...
}
```

```
void uxr_lock(
        uxrMutex* mutex)
{
    ...
#elif defined(UCLIENT_PLATFORM_PXROS)
    PxLock(&mutex->impl);
#else
    ...
}
```

```c
void uxr_unlock(
        uxrMutex* mutex)
{
    ...
#elif defined(UCLIENT_PLATFORM_PXROS)
    PxUnlock(&mutex->impl);
#else
    ...
}
```

Code listing 3.10 eProsima Micro XRCE-DDS mutex functions defined in *eProsima/Micro-XRCE-DDS-Client/src/c/profile/multithread/multithread.c*.

To use the multithreading profile provided by eProsima's Micro XRCE-DDS, a user needs to add a specific mutex structure definition to the uxrMutex. This is a general structure used in Micro XRCE-DDS. For example, in this case, the *PxMutex_t* structure is used, which is described in 3.2.

```c
// Micro XRCE-DDS Client mutex implementation

typedef struct uxrMutex
{
#ifdef WIN32
#elif defined(PLATFORM_NAME_FREERTOS)
    SemaphoreHandle_t impl;
    StaticSemaphore_t xMutexBuffer;
#elif defined(UCLIENT_PLATFORM_ZEPHYR)
    struct k_mutex impl;
#elif defined(UCLIENT_PLATFORM_POSIX)
    pthread_mutex_t impl;
#elif defined(UCLIENT_PLATFORM_PXROS)
    PxMutex_t impl;
#endif // ifdef WIN32
} uxrMutex;
```

Code listing 3.11 Micro XRCE-DDS Client mutex implementation structure in *eProsima/Micro-XRCE-DDS-Client/include/uxr/client/profile/multithread/multithread.h*.

- ”RMW_DECLARE_DEPRECATED” macro definition

  It is important to note that Hightec's compiler for TriCore version 4.9.4.1 v does not support the attribute deprecated with an optional argument inside the enumerate definition. Thus, the *RMW_DECLARE_DEPRECATED* macro does not warn about the deprecation in this compiler version. The macro is then defined as follows:

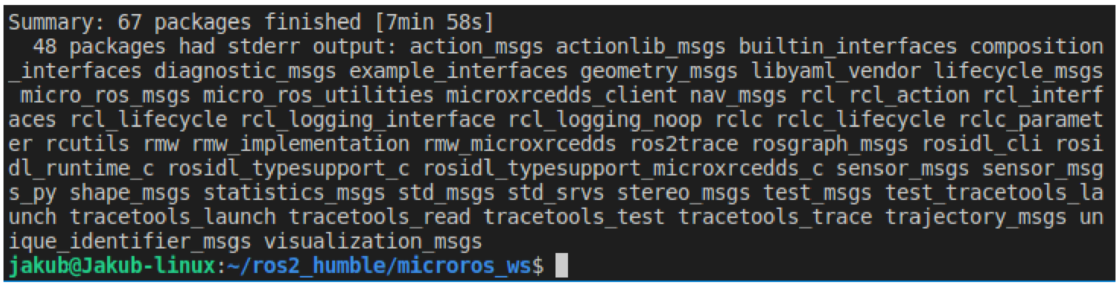
```c
#ifndef _WIN32
# ifdef __HIGHTEC__
#  define RMW_DECLARE_DEPRECATED(name, msg) name
# else
#  define RMW_DECLARE_DEPRECATED(name, msg) name __attribute__((deprecated(
    msg)))
# endif
#else
# define RMW_DECLARE_DEPRECATED(name, msg) name __pragma(deprecated(name))
#endif
```

Code listing 3.12 *RMW_DECLARE_DEPRECATED* macro definition in *eProsima/Micro-XRCE-DDS-Client/include/uxr/client/profile/multithread/multithread.h*.

3.1.3 Library Build

After making the necessary changes to the source code, the micro-ROS library can be built using the command provided in the tutorial:

```
$ ros2 run micro_ros_setup build_firmware.sh $(pwd)/my_custom_toolchain.cmake $(
    pwd)/my_custom_colcon.meta
```



```
Summary: 67 packages finished [7min 58s]
  48 packages had stderr output: action_msgs actionlib_msgs builtin_interfaces composition
_interfaces diagnostic_msgs example_interfaces geometry_msgs libyaml_vendor lifecycle_msgs
 micro_ros_msgs micro_ros_utilities microxrcedds_client nav_msgs rcl rcl_action rcl_interf
aces rcl_lifecycle rcl_logging_interface rcl_logging_noop rclc rclc_lifecycle rclc_paramet
er rcutils rmw rmw_implementation rmw_microxrcedds ros2trace rosgraph_msgs rosidl_cli rosi
dl_runtime_c rosidl_typesupport_c rosidl_typesupport_microxrcedds_c sensor_msgs sensor_msg
s_py shape_msgs statistics_msgs std_msgs std_srvs stereo_msgs test_msgs test_tracetools_la
unch tracetools_launch tracetools_read tracetools_test tracetools_trace trajectory_msgs un
ique_identifier_msgs visualization_msgs
jakub@Jakub-linux:~/ros2_humble/microros_ws$
```

Figure 3.1 Successful custom static micro-ROS library build.

Despite the fact that the build process produced 48 packages with stderr output, the library was successfully built. It is worth noting that the stderr output only contained warnings, as the build process would have failed with the first error encountered.

The resulting include folder structure after building a custom static micro-ROS library is not suitable for adding just one include path. This is due to the fact that the required header files are located in different directories. A comment on a GitHub issue[3] talking about this problem suggests: "As far as some platforms are easier just to include one folder and have all the headers inside, what we do is iterate the folder structure and copy 'back' folders that are repeated" [49].

To address the issue of the resulting include folder structure in the micro-ROS workspace not being suitable for adding a single include path, a bash script has been provided in a GitHub issue. [49] This script iterates through the folder structure and copies back folders that are repeated to make it easier to include a single folder with all the necessary headers. Running this script fixes the include paths in the micro-ROS workspace:

```
#!/bin/bash

LIBRARY_PATH=$(pwd)/firmware/build

######## Fix include paths  ########
pushd firmware/mcu_ws > /dev/null
    INCLUDE_ROS2_PACKAGES=$(colcon list | awk '{print $1}' | awk -v d=" " '{s=(
    NR==1?s:s d)$0}END{print s}')
popd > /dev/null

for var in ${INCLUDE_ROS2_PACKAGES}; do
    if [ -d "$LIBRARY_PATH/include/${var}/${var}" ]; then
        rsync -r $LIBRARY_PATH/include/${var}/${var}/* $LIBRARY_PATH/include/${
    var}
        rm -rf $LIBRARY_PATH/include/${var}/${var}
    fi
done
```

Code listing 3.13 Script to fix include paths folder structure taken from [49].

[3]The issue is available from https://github.com/micro-ROS/micro_ros_setup/issues/530

3.2 Mutex Implementation

The implementation of mutexes presented in this thesis does not follow any standard, such as POSIX. Instead, it was created to meet the specific requirements of eProsima's Micro XRCE-DDS multithreading profile implementation.

There are two ways to implement mutexes with PXROS-HR. However, since PXROS-HR is a multicore RTOS, the simpler implementation that can only work within a single core is not suitable for our purposes. As the goal of this thesis is to provide a multicore example, the simpler implementation will not be further examined in this work. To achieve multicore mutexes, a serializing entity, in the form of a task, is required. The Mutex Server is the task that will take care of providing mutex functionality across multiple cores. The micro-ROS implementation requires mutexes to be recursive which means that the task that already has the lock can lock it again without any waiting and continue in the execution. The same number of lock and unlock operations must be ensured.

In order to implement the Mutex Server, several components are needed. Firstly, the mutex structure has to be defined:

- Mutex structure – visible from the application
- Mutex service data structure – visible from Mutex Server
- Message metadata structure – to pass necessary information from an application to the Mutex Server

```
/* ==================================================================
 * Mutex structures
 * ==================================================================*/

/* Task mutex structure */
typedef struct PxMutex_t
{
  int mutex_id;           /* Mutex ID to identify the mutex service data */
  PxMbx_t mutex_server_mbx; /* Mutex server mailbox - avoiding to get it with
                                                      each request again */
} PxMutex_t;

/* Structure for inter task communication via PXROS messages metadata */
typedef union PxMutexMsg_t
{
  PxMsgMetadata_t pxmd; /* original metadata representation */
  struct itc            /* InterTask Communication representation of metadata */
  {
    int mutex_id;       /* Mutex ID to identify the mutex service data */
    MutexOperation_t mutex_operation;  /* Mutex request operation */
  } itc;
} PxMutexMsg_t;

/* Mutex server service data */
typedef struct PxMutexData_t
{
  PxMbx_t mutex_mbx;     /* Mutex mailbox (FIFO) for incoming lock requests */
  MutexLockState_t mutex_locked; /* State flag, MUTEX_LOCKED or MUTEX_UNLOCKED*/
  PxTask_t mutex_task;  /* Task ID of the task that has the active lock */
  int mutex_counter;     /* Recursive counter */
} PxMutexData_t;
```

Code listing 3.14 PXROS-HR mutex structures for application tasks, PXROS-HR message metadata for passing data between tasks and internal structure of the Mutex Server in *MutexServer.h*.

PXROS-HR objects necessary for the implementation:

- **Mailbox** – queue for message objects
- **Message** – object for communication within tasks

The following features of PXROS-HR are utilized in the implementation of mutexes:

- Name query – tasks can register data with the Name Server under a 32-bit tag, which makes the data ”public” and accessible to other tasks that request it by the tag
- Request a message – a message object is created to request init/lock/unlock operations
- Use metadata of the message – PXROS-HR offers 8 bytes of metadata that can be used to speed up message sending. If this size is insufficient, a buffer of the required size can be obtained
- Send message – messages are sent to the requested mailbox
- Release message – a message object can be returned to the object pool, or if a task is waiting for the release, the message can be returned to that task
- Wait for the release of the message – a task that has sent a message can wait for the receiver to release it. This blocks the task in execution but does not block the core, allowing other tasks to run.

In PXROS-HR, messages can be sent and waited for release, providing a mechanism for request-response data exchange. However, mutexes are considered a core mechanism for ensuring synchronization between data accessed by multiple threads or tasks. Typically, mutexes do not have return values and are not directly checked by the application. It is assumed that the mutex mechanism is functioning correctly. As such, any error in the mutex implementation is considered fatal and will result in a *PxPanic* (a panic routine triggered in case of a fatal error, with exact behavior depending on the processor, typically involving a breakpoint or illegal instruction) when a standard error occurs, or a *PxTrapAbort* when an unexpected error is encountered.

3.2.1 Mutex API Functions

- Mutex API functions have few things in common

  Firstly, they all request a message to communicate with the Mutex Server. This message is created without any extra buffer allocation, as the 8 bytes that can be stored in the message metadata are sufficient for passing information to the Mutex Server. The message is then set to await release, which allows the task to later call the *PxMsgAwaitRel* function and block itself without blocking the execution on the core for other tasks. Once a task calls *PxMsgRelease* on this message, the task that was waiting will receive the message and can continue its execution. This feature is also useful for passing return values, which can be stored in message metadata. After setting the message to await release, metadata from the message are obtained and filled with specific values defined by the metadata structure (3.14), including the mutex operation (init/lock/unlock) and mutex ID (obtained during initialization). The message is then sent to the Mutex Server, and a task waits for the response with the *PxMsgAwaitRel* function. This call is blocking, which gives us the desired functionality of the mutex – waiting until the caller can continue with having a lock. Finally, the message is released, which means that the message object is returned to the object pool and can be used again.

- The initialization call takes care of two additional things

  1. Firstly, it obtains the mailbox ID of the Mutex Server task, which has previously been registered to the Name Server by the Mutex Server. This ID is written to the application mutex structure, which has minimal memory overhead but improves functionality speed. While it would be possible to make a request to the Name Server to obtain the mailbox

ID, this would add more time overhead than necessary. Since mutexes are a core feature that always comes with non-negligible overhead, it's important to keep this overhead as small as possible.

2. Secondly, after initialization is done by the Mutex Server, the mutex ID that has been initialized is stored in the metadata of the message and needs to be copied to the application mutex structure so it can be used later.

The code of the API functions can be seen in appendix B.1.

3.2.2 Mutex Task Implementation

To minimize the probability of mutex failures during runtime, it is important to limit the maximum number of mutexes and ensure that there is always sufficient memory available for them. In the implementation described below, the mutex service data is stored on the stack of the Mutex Server task. However, determining the appropriate size for the task stack can be challenging, as it requires either static analysis of the code to identify worst-case memory consumption or experimental methods such as filling the stack with a pattern and observing the amount of memory used.

To minimize memory overhead and ensure that the mutex service data is always available and errors can be detected during compile time, a dedicated section can be created for this data and access can be allowed in extended memory regions. This approach provides greater control over the mutex service data and can help reduce the risk of runtime failures.

```
/* ======================================================================
 * Defines
 * ====================================================================*/

#define MAX_MUTEX_NUMBER    40
```

In the implementation mentioned in this thesis, mutexes are provided as a service task through a set of API functions. The service task maintains a simple array as its service data structure to handle mutexes. Since there is no requirement to destroy or deallocate the mutexes, the array serves as a sufficient data structure for this use case. Initialization of mutexes is performed by utilizing the *mutexes_used* index, which represents the next available index in the service data array. However, it is important to note that this index must not exceed the maximum number of mutexes, which is defined by *MAX_MUTEX_NUMBER*.

```
/* Mutex service data */
PxMutexData_t mutex_data[MAX_MUTEX_NUMBER];

int mutexes_used = 0;
```

At the start of its execution, the Mutex Server task registers its mailbox ID with the Name Server. This allows application tasks to retrieve the mailbox ID by sending a request to the Name Server, enabling them to send requests to the Mutex Server task.

```
/* Register Mutex server mailbox in NameServer that other tasks can query it */
PxError_t regErr = PxNameRegister (MutexServer_MID_NAMESERVERID,
                                   sizeof(PxTask_t),
                                   &myMailbox );
if (regErr != PXERR_NOERROR)
    PxPanic();
```

The Mutex Server task's main loop consists of the following steps:

1. Wait for messages from clients (the Mutex Server task does not block other tasks on the same core from executing their code)
2. Get metadata from the request message
3. Process the request based on its type
4. Release the request message

Requested operations:

- MUTEX_INIT

  One of the requested operations is *MUTEX_INIT*. When the Mutex Server receives a request with this operation, it first checks if there is any free mutex available. If there is, the Mutex Server proceeds to set the initial values to the service data that corresponds to the first free mutex. These initial values include the mutex's unlocked state, an invalid task ID, a counter set to zero, and the ID of the newly created mailbox.

  Suppose there are no errors during the creation of the mailbox. In that case, the Mutex Server fills the metadata with the initialized mutex ID (which corresponds to the array index) and increments the counter of used mutexes (setting it to the next free mutex index).

```
case MUTEX_INIT:

  /* Check if there is any free mutex */
  if (mutexes_used >= MAX_MUTEX_NUMBER)
    PxPanic(); // No free mutex

  /* Initialize mutex service data */
  mutex_data[mutexes_used].mutex_locked = MUTEX_UNLOCKED;
  mutex_data[mutexes_used].mutex_task = PxTaskIdInvalidate();
  mutex_data[mutexes_used].mutex_mbx = PxMbxRequest(PXOpoolTaskdefault);
  mutex_data[mutexes_used].mutex_counter = 0;

  /* Error while initializing the mutex? */
  if (!PxMbxIdIsValid(mutex_data[mutexes_used].mutex_mbx))
    PxPanic();

  /* Set metadata as an answer to application task */
  mutex_msg_metadata.itc.mutex_id = mutexes_used;
  PxMsgSetMetadata(taskMsg, mutex_msg_metadata.pxmd);

  mutexes_used++;

  break;
```

Code listing 3.15 Mutex Server process of *MUTEX_INIT* request in *MutexServer.c*.

- MUTEX_LOCK

  The locking process starts with the Mutex Server checking if the requested mutex ID is correct and has been previously initialized. If the check is successful, there are two possible situations – the mutex is either *LOCKED* or *UNLOCKED*. In the case of an unlocked mutex, the Mutex Server sets the mutex state to *LOCKED*, assigns the requester's task ID to the service data, and increments the recursive counter.

  However, if the mutex is already locked, the Mutex Server checks whether the requester is the owner of the lock. If the requester is indeed the owner, the recursive counter is simply incremented. If the requester is not the owner of the lock, a request message is sent to the mutex mailbox, which behaves like a FIFO for such requests. This ensures that the requests are served in the order in which they arrived. Once the current lock owner unlocks the mutex, the first waiting message in the mailbox will take the lock.

```c
case MUTEX_LOCK:

  /* Check if the ID is valid */
  if (mutex_id < 0 || mutex_id >= mutexes_used)
    PxPanic(); // Invalid ID

  /* Check if the mutex is unlocked */
  if (mutex_data[mutex_id].mutex_locked == MUTEX_UNLOCKED)
  {
    /* Mutex is free, claim it */
    mutex_data[mutex_id].mutex_locked = MUTEX_LOCKED;
    mutex_data[mutex_id].mutex_task = PxMsgGetOwner(taskMsg);
    mutex_data[mutex_id].mutex_counter++;
    break;
  }

  /* Owner of the lock locks it again */
  else if (PxTaskIdGet(mutex_data[mutex_id].mutex_task) == PxTaskIdGet(
    PxMsgGetOwner(taskMsg)))
  {
    mutex_data[mutex_id].mutex_counter++;
    break;
  }

  /* Mutex is locked, send msg to waiting list (mailbox FIFO) for the mutex
    */
  PxMsgSend(taskMsg, mutex_data[mutex_id].mutex_mbx);

  break;
```

Code listing 3.16 Mutex Server process of *MUTEX_LOCK* request in *MutexServer.c*.

- MUTEX_UNLOCK

  The unlocking process also starts with checking if the requested mutex ID is correct and previously initialized. If the mutex is locked and the requester is the owner of the lock, the unlocking process can continue. Otherwise, there are two possible situations – unlocking an unlocked mutex or unlocking without having a lock. In this implementation, these situations do not generate errors, but they should never happen in a well-designed application. Therefore, it is debatable whether this behavior should be considered an error or not.

  If the unlocking process continues, the recursive counter is decremented. If the recursive counter reaches zero, which means that the lock was the ”last” one, the Mutex Server tries to get a request message from the mutex mailbox. If there is no waiting request (no message in the mailbox), the mutex is unlocked (set to *UNLOCKED* and invalidate the task ID). If there is any task waiting for the lock (request message was sent to the mailbox because the lock was already locked), the mutex service data are updated according to the waiting request, and the request message is released so that the waiting task can continue its execution with the lock.

```c
case MUTEX_UNLOCK:

  /* Check if the ID is valid */
  if (mutex_id < 0 || mutex_id >= mutexes_used)
    PxPanic(); // Invalid ID

  /* Check if the task that is trying to unlock is the owner of the lock */
  if (mutex_data[mutex_id].mutex_locked == MUTEX_LOCKED
      && PxTaskIdGet(mutex_data[mutex_id].mutex_task) == PxTaskIdGet(
    PxMsgGetOwner(taskMsg)))
  {
    /* Firstly decrement mutex counter and then if it can be freed */
    if (--mutex_data[mutex_id].mutex_counter == 0)
    {

      /* Check if any other task waits for the lock */
      PxMsg_t waiting_mutex_msg = PxMsgReceive_NoWait(mutex_data[mutex_id].
    mutex_mbx);
      if (PxMsgIdIsValid(waiting_mutex_msg))
      {
        /* A task is waiting - claim the mutex for first waiting task */
        mutex_data[mutex_id].mutex_task = PxMsgGetOwner(waiting_mutex_msg);
        mutex_data[mutex_id].mutex_counter++;
        PxMsgRelease(waiting_mutex_msg);
        break;
      }

      /* Set mutex to free */
      mutex_data[mutex_id].mutex_locked = MUTEX_UNLOCKED;
      mutex_data[mutex_id].mutex_task = PxTaskIdInvalidate();
    }
  }

  break;
```

Code listing 3.17 Mutex Server process of *MUTEX_UNLOCK* request in *MutexServer.c*.

In order to ensure proper functioning of mutexes, users must follow a strict order of operations as outlined in 3.2. Firstly, the Mutex Server task must be executed to register its mailbox ID with the Name Server, which should already be running. Secondly, each individual mutex must be initialized before any locking or unlocking operations can be performed. By following this order, users can ensure that mutexes will function correctly and efficiently within their application.

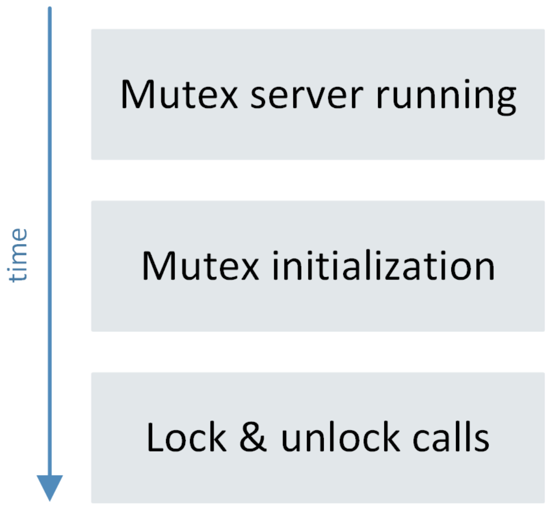


Figure 3.2 Mutex timeline that the user must follow the order to ensure mutexes will work properly.

3.2.3 Mutex Task Creation

The Mutex Server task is created as a service task before any other task in the system. This is achieved by calling the Mutex Server create function on the Mutex Server core. However, since the Init task has a higher priority, the Mutex Server does not get any execution time initially. To allow the Mutex Server to execute its code, the Init task lowers its priority. The Mutex Server then executes its initialization code and waits for incoming requests.

Now the Init task can continue with its execution and registers the Mutex Server mailbox as a service mailbox, ensuring that no task can continue execution until the Mutex Server is registered. This is achieved by calling the *HtcWaitForService* function for all tasks. The function waits for the Mutex Server mailbox to be registered and returns an error if the wait time exceeds a set duration. By default, the wait time is set in the *px_utils.h* file.

```
/* Mutex server initialization */
if (coreId == MUTEXSERVER_CORE)
{
    /* Create Mutex server task */
    PxTask_t taskId = MutexServer_Create(MUTEXSRV_PRIO, 0, PXMcTaskdefault,
    PXOpoolTaskdefault);

    /* Enable the execution of service tasks during the user tasks deployment by
     setting priority
     * of InitTask lower than the lowest service priority (lower prio = higher
    number!)
     */
    PxTaskSetPrio( myID, MUTEXSRV_PRIO + 1 );

    /* Register service to a free request ID */
    PxMbxRegisterMbx(_PxSrv5_ReqMbxId, PxTaskGetMbx(taskId));
}

/* Wait for the Mutex server service task on MUTEXSERVER_CORE to get initialized
    .*/
errRes = HtcWaitForService (MUTEXSERVER_CORE, _PxSrv5_ReqMbxId, 0, 0,
    INITTASK_EVENT_WAIT);
if (errRes != PXERR_NOERROR)
    PxPanic();
```

Code listing 3.18 Mutex Server task creation and initialization in *InitTask.c*.

The *HtcWaitForService* is an essential part of the initialization process for the Mutex Server and Name Server tasks, as it ensures that no task will continue execution until the task mailboxes are registered as service mailboxes.

3.3 Project Structure

The example developed in this thesis is based on HighTec's example that demonstrates the use of PxNet – *TC39x PxNet base example* (most simple PxNet project with only PxNetTask, EthDrvTask, PxNetMonitor and LedServer). To understand how the example is structured, the HighTec project structure will be briefly introduced, and then it will be explained what changes and additions were made to this example to make it work with the custom static micro-ROS library built earlier.

3.3.1 HighTec Project Structure

Only brief description will be provided. For more detailed information please refer to HighTec TC39x PXROS-HR BSP example's documentation [50].

The HighTec project structure is composed of several logically separated elements, each having its own folder within the project. These elements include bsp, crt0, ld, src, and pxros (see folder structure in Figure 3.5).

- **bsp** – a BSP component represents microcontroller and evaluation board dependent SW tailored to a particular microcontroller derivative
- **crt0** – a toolchain and a microcontroller-dependent startup code that initializes a 'C' runtime environment
- **ld** – linker files prescribing placement of the final application code and data
- **src** – it contains one file 'shared_main.c' implemented as a shared code executed on each core; the PXROS-HR operating system starts here by the execution of the 'PxInit' API function
- **pxros/config** – PXROS-HR system configuration. Here the user specifies parameters for each of the instantiated kernels, like the number of objects, tasks, and size of the default user memory
- **pxros/hal** – An abstraction of the underlying microcontroller for PXROS-HR time tick functionality
- **pxros/tasks** – A folder with the implementation of example tasks
- **pxros/utils** – Common utilities used across user tasks

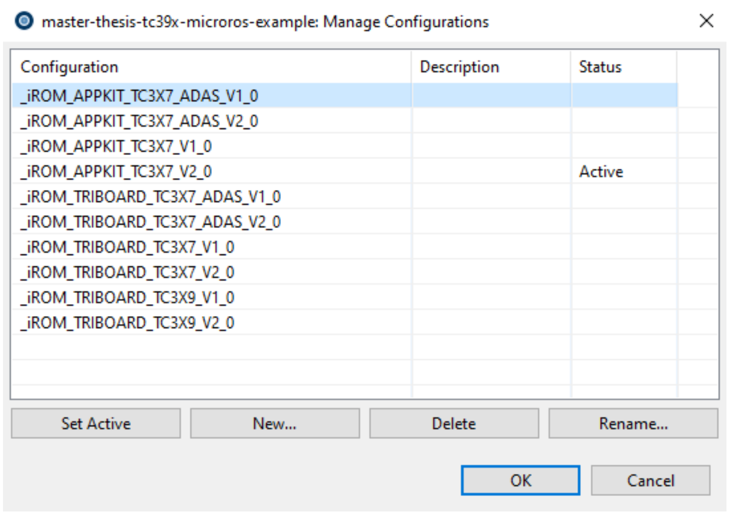
3.3.2 Configurations

The *tc39x_pxnet_base_example* includes pre-defined configurations (3.3) for different boards. For the purpose of this master's thesis, all configurations except *_iROM_APPKIT_TC3X7_V2_0* – which is used for the Application Kit TC37X version 2.0 board and serves as a base for creating new configurations – can be deleted. Each configuration will represent a demo and should follow the naming convention: "_⟨*demo name*⟩".

The *Create New Configuration* window will appear where you can set the desired settings for the new configuration (as shown in 3.4). The *Copy settings from – Existing configuration* option is used, with *_iROM_APPKIT_TC3X7_V2_0* selected as the base configuration. Once a configuration has been created for each demo, the original *_iROM_APPKIT_TC3X7_V2_0* configuration can be deleted.

For each configuration, the symbol *DEPLOY_⟨DEMO NAME⟩=1* is defined in both the C and C++ Compilers. Additionally, the include paths to the micro-ROS library headers must be set in the *Project properties* for both compilers. Finally, the micro-ROS library itself needs to be added to the C++ Linker with the appropriate library search path.

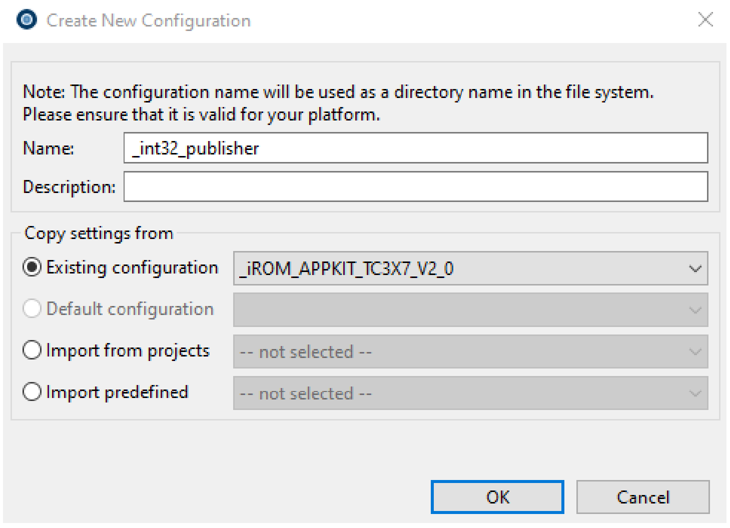
By following these steps, multiple demos can be implemented and tested on the Application Kit TC37X version 2.0 board with only changing the configuration.



master-thesis-tc39x-microros-example: Manage Configurations

Configuration	Description	Status
_iROM_APPKIT_TC3X7_ADAS_V1_0		
_iROM_APPKIT_TC3X7_ADAS_V2_0		
_iROM_APPKIT_TC3X7_V1_0		
_iROM_APPKIT_TC3X7_V2_0		Active
_iROM_TRIBOARD_TC3X7_ADAS_V1_0		
_iROM_TRIBOARD_TC3X7_ADAS_V2_0		
_iROM_TRIBOARD_TC3X7_V1_0		
_iROM_TRIBOARD_TC3X7_V2_0		
_iROM_TRIBOARD_TC3X9_V1_0		
_iROM_TRIBOARD_TC3X9_V2_0		

Figure 3.3 PxNet base example configurations for different boards.



Create New Configuration

Note: The configuration name will be used as a directory name in the file system.
Please ensure that it is valid for your platform.

Name: _int32_publisher

Description:

Copy settings from

[✓ Existing configuration] _iROM_APPKIT_TC3X7_V2_0

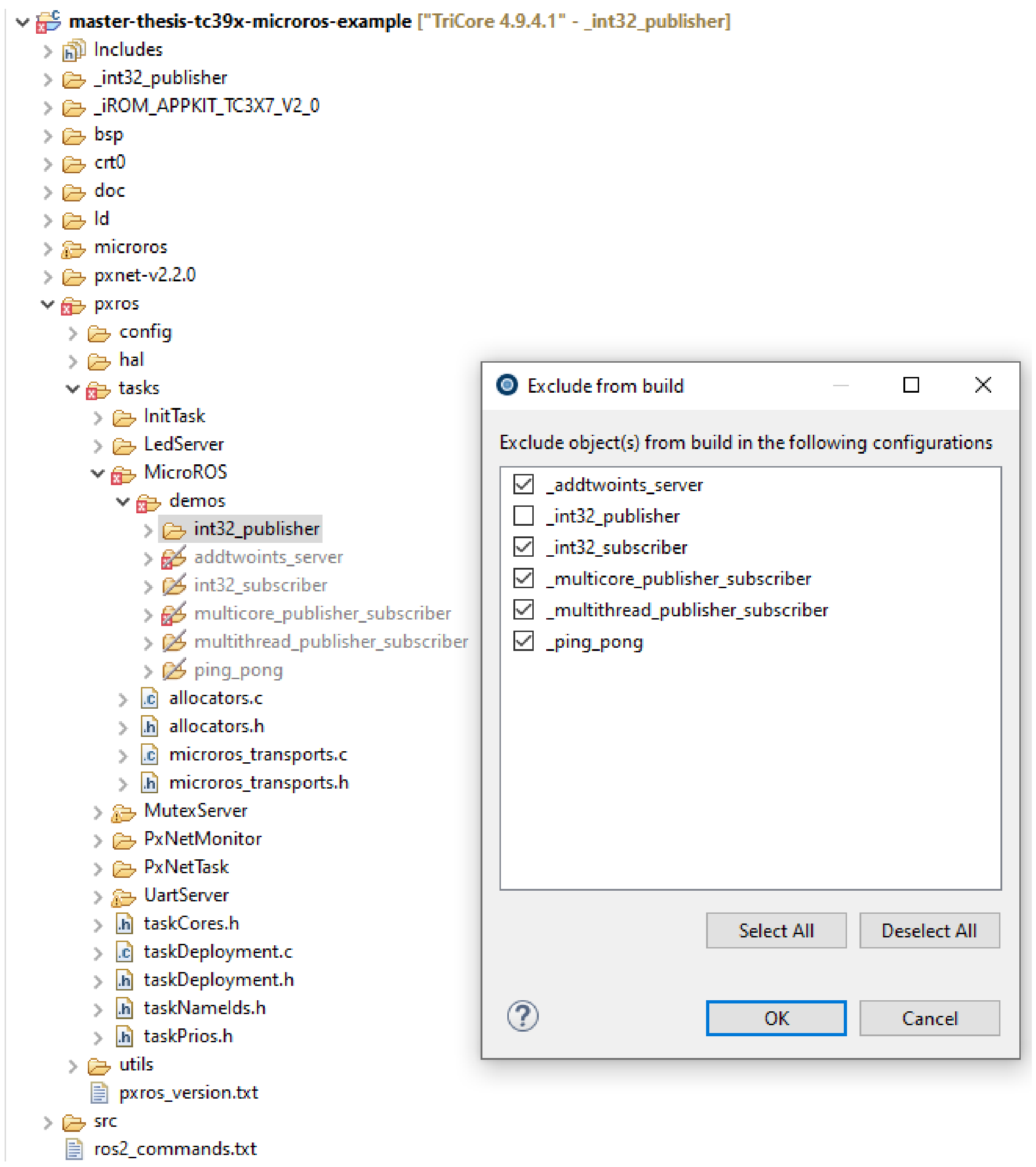
Default configuration

Import from projects -- not selected --

Import predefined -- not selected --

Figure 3.4 Creating new configuration by copying the settings from the PxNet base example configuration for Application kit TC3X7 version 2.0.

To ensure that the demos do not define the same variables or structures, etc., the symbol *DEPLOY_⟨DEMO NAME⟩=1* is used to include only the header files of the currently selected demo by the configuration. Similarly, this can be done for the sources as well. In the HTC IDE, there is an option to exclude sources from the build, which requires manual exclusion of the demo from all non-corresponding configurations. An example of this exclusion can be seen with *_int32_publisher* in Figure 3.5.



```
master-thesis-tc39x-microros-example ["TriCore 4.9.4.1" - _int32_publisher]
  Includes
  _int32_publisher
  _iROM_APPKIT_TC3X7_V2_0
  bsp
  crt0
  doc
  ld
  microros
  pxnet-v2.2.0
  pxros
    config
    hal
    tasks
      InitTask
      LedServer
      MicroROS
        demos
          int32_publisher
          addtwoints_server
          int32_subscriber
          multicore_publisher_subscriber
          multithread_publisher_subscriber
          ping_pong
        allocators.c
        allocators.h
        microros_transports.c
        microros_transports.h
      MutexServer
      PxNetMonitor
      PxNetTask
      UartServer
      taskCores.h
      taskDeployment.c
      taskDeployment.h
      taskNameIds.h
      taskPrios.h
    utils
    pxros_version.txt
  src
  ros2_commands.txt
```

Exclude from build

Exclude object(s) from build in the following configurations

- ☑ _addtwoints_server
- ☐ _int32_publisher
- ☑ _int32_subscriber
- ☑ _multicore_publisher_subscriber
- ☑ _multithread_publisher_subscriber
- ☑ _ping_pong

Figure 3.5 Selecting configurations in which to exclude sources inside the demo folder from the build.

3.3.3 Linker File

Each micro-ROS demo example requires two sections: the micro-ROS library section, which provides access to the library functions and global data, and a demo-specific section that contains the global data required for that particular demo (e.g. publisher data). All of these sections are enclosed within two symbols that define the beginning and end of the section and are named using the following convention:

- ⟨*NAME OF THE SECTION*⟩_BASE
- ⟨*NAME OF THE SECTION*⟩_END

These symbols allow the user to add the section either to the task context or to the extended memory region and give the tasks access to the section. Additionally, all symbols are aligned to 8 bytes to match the MPU region's granularity. Micro-ROS demos section are defined as follows:

```
/* ======================================================================
 * Micro-ROS demos sections
 * ====================================================================*/

SECTIONS
{
    /* MicroROS data and bss section */
    .microROS.data :
    {
      . = ALIGN(8);
        MICROROS_BASE = .;
      *libmicroros.a:(*.data*)
    } > DATA AT > RODATA

    .microROS.bss :
    {
        *libmicroros.a:(*.bss*)
        . = ALIGN(8);
        MICROROS_END = .;
    } > DATA

    /* MicroROS applications data */
    .microROS_demos.bss :
    {
        /* int32_publisher */
        . = ALIGN(8);
        INT32_PUBLISHER_BASE = .;
        *(.int32_publisher)
        . = ALIGN(8);
        INT32_PUBLISHER_END = .;

        ...

    } > DATA
}

```

Code listing 3.19 Micro-ROS demo sections in linker file *tc39x_pxnet_base_example.ld*

To maintain independence between the library and demos, three output sections are created. The library section includes both *bss* (global data) and *data* (initialized values/constants). The startup code in HighTec examples includes a configuration table that informs the application on how to manage user's memory. Global data requires no special initialization, but is assumed to be initialized with zeroes. During startup code execution, HighTec sets the RAM memory to zeroes for the ranges defined in the *clear table*. On the other hand, data requiring value initialization are placed into a *copy table* and are copied from FLASH to RAM memory by the startup code. The definition of *clear* and *copy* tables for core 0 and core 1:

```
/* ==========================================================================
 * CLEAR & COPY TABLES with END delimiter to support crt0 init
 * Each core has its own table to process during its init to allow multicore
 * execution.
 * Shared resources are inserted to Core[0] tables (the RESET core)
 * clear_sec:
 *    data memory ranges to clear to zero
 * copy_sec:
 *    data memory ranges that needs to be value initialized
 *    (init values are stored in FLASH and copied to RAM)
 * ========================================================================*/

```

```
SECTIONS
{
  /* ---- CORE 0 ---- */

  .CPU0.clear_sec :
  {
    LONG(ADDR(.microROS.bss)); LONG(SIZEOF(.microROS.bss));
    LONG(ADDR(.library.bss)); LONG(SIZEOF(.library.bss));
    LONG(ADDR(.microROS_demos.bss)); LONG(SIZEOF(.microROS_demos.bss));
    LONG(ADDR(.bss)); LONG(SIZEOF(.bss));
    LONG(ADDR(.heap)); LONG(SIZEOF(.heap));
    LONG(-1); LONG(-1);
  } > RODATA_CPU0_

  .CPU0.copy_sec :
  {
    LONG(LOADADDR(.microROS.data)); LONG(ADDR(.microROS.data)); LONG(SIZEOF(.
    microROS.data));
    LONG(LOADADDR(.library.data)); LONG(ADDR(.library.data)); LONG(SIZEOF(.
    library.data));
    LONG(LOADADDR(.data)); LONG(ADDR(.data)); LONG(SIZEOF(.data));
    LONG(-1); LONG(-1); LONG(-1);
  } > RODATA_CPU0_

  /* ---- CORE 1 ---- */

  .CPU1.clear_sec :
  {
    LONG(-1); LONG(-1);
  } > RODATA_CPU1_

  .CPU1.copy_sec :
  {
    LONG(-1); LONG(-1); LONG(-1);
  } > RODATA_CPU1_

  ...
}

```

Code listing 3.20 Beginning of linker *clear* and *copy* tables in *tc39x_pxnet_base_example.ld*.

3.3.4 Tasks

In the HighTec example structure, each task typically has its own folder containing a source file, header file, and configuration file. In addition, there are some support files that are related to all tasks:

- **taskCores** – configuration file that provides the task-to-core assignment
- **taskDeployment** – file containing create functions for user task deployment
- **taskNameIds** – names definition to query entry specific content using NameServer
- **taskPrios** – file that specifies the priorities for tasks, services, and hardware interrupts

The unused tasks from the original example (*tc39x_pxnet_base_example*) can be deleted – PxNetMonitor and LedServer.

Mutex Server

The Mutex Server is a service task responsible for the multicore mutex functionality. More information about the Mutex Server can be found in section 3.2.

PxNet Task

The PxNet Task is responsible for configuring PxNet and allowing users to use the TCP/IP stack.

Uart Server

The UART Server is utilized for debugging output, although the conventional *printf* function is not applicable. Instead, a buffer for output is defined, and the text is formatted to the buffer using the *sprintf* function. The buffer is then sent to the UART Server via the *ioWrite* function, ensuring that the buffer is forwarded to the output to be displayed on terminals on a PC.

In order to use macros defined in the standard micro-ROS demos, the buffer must be global (more about this is mentioned in 3.4.3). As a result, a special section must be created to provide access to the task that will be using the buffer for output. To enable PXROS to set the MPU with appropriate access rights for the task, the user must add an extended region or task context that includes the buffer.

To ensure that the buffer is printed correctly, it should not be altered by other tasks while a request is made to forward the buffer to the output. As a result, each task should have its own buffer.

The API and intertask communication to the UART task are defined in the *utils* folder and provide functions for read, write, and select:

- **ioWrite** – for sending a buffer to the output
- **ioRead** – waits (blocking) until input is received and then reads the data
- **ioSelect** – waits without blocking for input, without reading the data

The function declarations for these API functions are:

```
int ioWrite(PxMbx_t mbx, char *buf, int length);
int ioRead(PxMbx_t mbx, char *buf, int length);
int ioSelect(PxMbx_t mbx, PxEvents_t ev);
```

Micro-ROS Demos

Demos are created as pluggable tasks. Each demo has its own folder and configuration which excludes all other demos from build. All the main task create functions are placed to *taskDeployment.c* to the deployment table. *Init task* calls the *TaskDeploy* function which iterates through this table and when the task is supposed to run on the core where the function is executed, it will call the task create function.

3.4 Micro-ROS Demos

All the demos presented in this thesis are equivalent to the demos available in the official micro-ROS GitHub repository [51]. Although there are numerous examples that showcase the basic functionalities, due to the limitations of this thesis, it was not feasible to implement all of them. To achieve the main goal of this thesis, which is to investigate the feasibility of integrating micro-ROS with PXROS-HR, inspiration was taken from other RTOSes that are officially supported by micro-ROS, such as FreeRTOS. Each supported RTOS has its own repository, which contains subsets of the original micro-ROS demos. For this thesis, the following demos were chosen:

- **int32_publisher** – demonstrates the basic functionality of publishing to a topic
- **int32_subscriber** – demonstrates the basic functionality of subscribing to a topic
- **ping_pong** – demonstrates more advanced functionality by combining two publishers and two subscribers
- **addtwoints_server** – demonstrates the implementation of a service server that provides a response for the addition of two given integers
- **multithread_publisher_subscriber** – demonstrates running multiple tasks (threads)

To establish a connection with the ROS 2 agent, a transport must be defined. Although it is possible to use the default transports provided by eProsima's Micro XRCE-DDS, the HighTec PxNet module provides TCP/IP functionality, which necessitates the creation of a custom transport. In this case, mapping the PxNet API functions to the micro-ROS API functions is required.

Furthermore, micro-ROS allows for the definition of custom allocators, which can be useful when the user requires more control over the managed memory. On TriCore with PXROS-HR, the hardware memory protection unit is used to grant each task access only to the necessary memory areas for the task's functionality. This restricts unwanted accesses, which improves safety and security.

3.4.1 Custom Allocators

According to [52], the default allocator wraps the following methods:

```
- allocate = wraps malloc()
- deallocate = wraps free()
- reallocate = wraps realloc()
- zero_allocate = wraps calloc()
- state = ‘NULL‘
```

Code listing 3.21 The default allocator functions from [52].

which are functions of the standard library *stdlib.h*. To use these functions, a user must define a heap and grant access rights to the standard library for the tasks that will use them. However, this means that tasks will have access to the entire heap, which is not necessary and compromises safety.

PXROS-HR provides its own memory management functions, such as *PxMcTakeBlk* for allocating a block of memory of a desired size from a requested memory class and *PxMcReturnBlk* for freeing or deallocating the block. Realloc and zero allocate can be easily implemented with these two functions and the standard library. It is important to note that the *realloc* function in PXROS-HR will not extend the actual buffer size, but it will always try to allocate a new, larger buffer.

The advantage of using PXROS-HR memory management functions is that access rights are handled by the operating system, and users do not need to add any access rights to the task's configuration. Additionally, if there is no need to use any other standard library function that requires global data, access rights to that area can be omitted.

In a single-task environment, all demos will use PXROS-HR functions for memory management. However, this is not a suitable solution for multithreading, multitasking, or multicore environments, as access rights cannot be shared easily. For example, consider a main task that creates buffers and initializes the system. After initialization, the main task creates a new task (for example, a publisher). The main thread passes the pointer to the allocated publisher structure, but the publisher task does not have access rights to the memory because it was allocated by a different task, and access rights are set only for the task that took the block from the memory class. Any approach that uses PXROS-HR memory management functions in a multithreaded environment would be too complicated or would result in excessive overhead. Thus, setting both tasks to have access to the heap area means that the first one can allocate the necessary data and the new task can access it. Therefore, all demos that run within multiple tasks will use the heap and default standard allocators approach.

The prototypes for Micro-ROS functions for custom allocators are presented below:

```
void * (*allocate)(size_t size, void * state);
void   (*deallocate)(void * pointer, void * state);
void * (*reallocate)(void * pointer, size_t size, void * state);
void * (*zero_allocate)(size_t number_of_elements, size_t size_of_element,
                        void * state);
```

Here are the PXROS-HR functions fulfilling the Micro-ROS prototypes:

```
void * PXROS_allocate(size_t size, void * state);
void   PXROS_deallocate(void * pointer, void * state);
void * PXROS_reallocate(void * pointer, size_t size, void * state);
void * PXROS_zero_allocate(size_t number_of_elements, size_t size_of_element,
                           void * state);
```

The implementation of custom allocators based on PXROS-HR memory management functions and standard library functions is provided in Appendix C.

3.4.2 Custom Transport

The official micro-ROS website provides a tutorial ([53]) on implementing custom micro-ROS transports, which require adherence to specified function prototypes. Two communication modes are available for implementation:

- Stream-oriented mode
- Packet-oriented mode

For the present application, which employs the PxNet module for encapsulating the TCP/IP stack and uses UDP communication for message exchange, packet-oriented mode is selected by disabling the *framing* flag, setting it to 0.
Here are the micro-ROS function prototypes for custom transport callbacks:

```
bool   (*custom_transport_open)(uxrCustomTransport* transport);
bool   (*custom_transport_close)(uxrCustomTransport* transport);
size_t (*custom_transport_write)(
            uxrCustomTransport* transport,
            const uint8_t* buffer,
            size_t length,
            uint8_t* errcode);
size_t (*custom_transport_read)(
            uxrCustomTransport* transport,
            uint8_t* buffer,
            size_t length,
            int timeout,
            uint8_t* errcode);
```

Below is a list of the PxNet custom transport structure and the functions that meet the Micro-ROS prototypes requirements:

```
/* Structure for PxNet communication */
typedef struct PxnTransport
{
  int sock;
  int port;
  struct sockaddr_in addr;
} PxnTransport;

bool pxn_udp_open(uxrCustomTransport * transport);
bool pxn_udp_close(uxrCustomTransport * transport);
size_t pxn_udp_write(uxrCustomTransport* transport, const uint8_t * buf, size_t
    len, uint8_t * err);
size_t pxn_udp_read(uxrCustomTransport* transport, uint8_t* buf, size_t len, int
     timeout, uint8_t* err);
```

A detailed description of the implementation of each function:

- pxn_udp_open

  This function allocates a UDP socket instance using the *Pxn_Socket* function from the PxNet API. If the allocation is successful, the function fills the custom transport structure (*PxnTransport*) with the IP address of the ROS 2 agent.

- pxn_udp_close

  This function closes a socket instance by calling *Pxn_Close*.

- pxn_udp_write

  This function for sending data is more complicated than the previous. Firstly, the buffers used in micro-ROS are not aligned, which does not meet the requirements of PxNet. Defining the buffer with the "aligned" attribute would not be effective, as the buffer is used incrementally, meaning that it will inevitably become misaligned at some point. This means that extra memory block has to be obtained from the memory class using *PxMcTakeBlk*. This block is aligned and the original buffer containing data to be sent must be copied to this temporary aligned buffer. The PxNet function *Pxn_Sendto*, used for sending a UDP packet, can return an error code *EAGAIN*, which means "try again later". This usually happens only before the PxNet knows where to send the packet (before initial ARP requests). Therefore, a timeout has been added specifically for the *EAGAIN* error code case. The timeout is based on the PXROS-HR timeout object, where a task sets the time in the number of ticks and event that would be used to generate a wake-up event. The default values are 5 tries with a 2-millisecond timeout between each try, resulting in a total of 10 ms for "try again later" time. During the 2 milliseconds, the task waits without blocking the execution on its core for other tasks. If any other error occurs during the sending or data are successfully sent, the timeout object is stopped, released, and the temporary aligned buffer is released as well.

- pxn_udp_read

  This function for receiving data is similar to the previously mentioned *pxn_udp_write*. However, the timeout is obtained as a parameter of the function and is implemented not to block the execution on its core for other tasks. If the timeout is higher than 10 ms, the number of retries is set to 10, and the retry timeout is divided by 10 so that the final timeout number corresponds. This limits the application to use multiples of 10, so as not to shorten the timeout (the maximum cut from the original timeout is 9 ms). If the timeout is lower than 10 ms, one millisecond is used as the retry timeout, and the number of retries corresponds to the given timeout value. As the *Pxn_Recvfrom* function also expects the buffer to be aligned, the aligned temporary buffer must be created for receiving the data as well. After the successful receive, the data are copied into the given buffer. The number of bytes received or an error code is returned after stopping the timeout object, releasing it, and the temporary aligned buffer.

The timeout for sending brings almost zero overhead since the *Pxn_Sendto* function is typically called only once. However, the overhead resulting from copying data to aligned buffers is not negligible, which is a limitation of PxNet.

The code for the custom transport functions can be found in Appendix D.

3.4.3 Example Implementation

Each demo has a section named after the example, containing global variables such as:

1. uart_buff – an aligned buffer used for debug prints via UART
2. uart_mbx – the mailbox of the UART Server (where to send requests)

Here is an example for the *int32_publisher*:

```
#define STRING_BUFFER_LEN 100

#pragma section ".int32_publisher" 8 awB
char uart_buff[STRING_BUFFER_LEN] PXMEM_ALIGNED;
PxMbx_t uart_mbx;
rcl_publisher_t publisher;
std_msgs__msg__Int32 msg;
#pragma section
```

The synchronization with the PxNet (TCP/IP stack) is the first step in every demo. Each PXROS-HR kernel has a local table of service request mailboxes on each core. After creating the PxNet task, its mailbox is registered to the table on the core where it runs. In order to make this service visible from a different call, the user must call *PxGetGlobalServerMbx* with the core where it was registered. This function copies the entry to the local table so that it can be later accessed with *PxMbxRequestMbx*, which looks only to the local table. This is necessary when a task that wants to use the PxNet runs on a different core.

PxNet does not currently have any synchronization point, meaning that there is no exact point at which a user knows that PxNet and the Ethernet driver are fully initialized. When a task starts executing micro-ROS code without synchronization, there are timeouts that could cause the application to fail. Although a synchronization point would require intervention in the PxNet implementation, which is beyond the scope of this thesis. As a simple workaround, a waiting period of a few seconds is used to ensure the full initialization of PxNet before starting the application. Although this solution is not appropriate for a production environment, it suffices to test the feasibility of the prototype in this thesis. Based on experimentation, a waiting time of ten seconds has been found to be sufficient in most cases.

```
/* Synchronization point
 * ---------------------
 * The demo task needs to wait for PxNet to get initialized and running.
 * It will be done as soon as the _PxTcpAccessReqMbxId global server mailbox
 * became available.
 */

```

```c
errRes = PxGetGlobalServerMbx (PXNETTASK_CORE, _PxTcpAccessReqMbxId);
if (errRes != PXERR_NOERROR)
    PxPanic();

/* Get UART Server mailbox */
uart_mbx = UartServer_getMbx(1 << 25);

/* Wait for 10 sec (complete initialization of PxNet Task and Ethernet driver)
    */
HtcSleep(PxTickGetTicksFromMilliSeconds(10000), 1);
```

In the case of the demos that are executed within a single task, the custom transport structure (*transport*) is allocated on the task's stack. Conversely, for the examples that run across multiple tasks, the transport structure is allocated on the global data section of the demo (thus allowing access to be shared across the multiple tasks).

The code for the single task demos is provided below:

```c
/* Set custom transport */
PxnTransport transport;
transport.sock = -1;

if (rmw_uros_set_custom_transport(
    false,
    (void *) &transport,
    pxn_udp_open,
    pxn_udp_close,
    pxn_udp_write,
    pxn_udp_read) != RMW_RET_OK)
{
    PxPanic();
}

```

In single task demos, the process of custom transport configuration is followed by the setting of custom allocators:

```c
/* Set custom allocators */
rcl_allocator_t PXROS_allocator = rcutils_get_zero_initialized_allocator();
PXROS_allocator.allocate = PXROS_allocate;
PXROS_allocator.deallocate = PXROS_deallocate;
PXROS_allocator.reallocate = PXROS_reallocate;
PXROS_allocator.zero_allocate = PXROS_zero_allocate;

if (!rcutils_set_default_allocator(&PXROS_allocator))
{
    PxPanic();
}
```

With the help of the UART Server and this initialization step, it becomes possible to replace the original *printf* function with *sprintf* and *ioWrite* function calls. This can be observed in the modified *RCCHECK* and *RCSOFTCHECK* macros which have been changed from the original version:

```c
#define RCCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc != RCL_RET_OK)){printf
    ("Failed status on line %d: %d. Aborting.\n",__LINE__,(int)temp_rc);}}
#define RCSOFTCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc != RCL_RET_OK)){
    printf("Failed status on line %d: %d. Continuing.\n",__LINE__,(int)temp_rc)
    ;}}
```

Code listing 3.22 Original *RCCHECK* and *RCSOFTCHECK* macros.

to:

```
#define RCCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc != RCL_RET_OK)){ \
    sprintf(uart_buff, "Failed status on line %d: %d. Aborting.\n",__LINE__,(int
    )temp_rc); \
    ioWrite(uart_mbx, uart_buff, strlen(uart_buff)); \
    PxPanic();}}
#define RCSOFTCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc != RCL_RET_OK)){ \
    sprintf(uart_buff, "Failed status on line %d: %d. Continuing.\n",__LINE__,(
    int)temp_rc); \
    ioWrite(uart_mbx, uart_buff, strlen(uart_buff));}}
```

Code listing 3.23 Modified *RCCHECK* and *RCSOFTCHECK* functions using UART Server.

The header file associated with each demo only declares the create function. For instance, the *int32_publisher* demo has the following declaration in its corresponding header file:

```
/* ======================================================================
 * TASK API INTERFACE
 * =====================================================================*/

extern PxTask_t int32_publisher_Create(PxPrio_t prio, PxEvents_t events,
                                       PxMc_t memClass, PxOpool_t objPool);
```

Task configuration

The configuration of the task is stored in a dedicated file with the suffix "_cfg.c". All the configuration parameters remain the same as in the other application tasks from the original *PxNet base example*, except for a few things described below. For a more in-depth understanding of the configuration, please refer to HighTec's PXROS-HR example manuals.

Regarding the stack size, it should be noted that it is defined separately for each task. As mentioned previously, the stack size is set to be much larger than what was revealed in memory profiling [54] of micro-ROS. This ensures that the integration is not burdened by memory failure issues. The stack size definition for the task is shown below:

```
/* TASK STACKs (in Bytes) */
#define TASK_STACKSIZE                4000
#define TASK_INTR_STACKSIZE           64
```

After defining the stack size, the symbols from the linker file that create the boundaries of the sections need to be declared so that they can later be used to define memory regions for the MPU. Tasks inherit the data context from their parent, and the extended regions are set based on the needs of the application. The specific demo section, micro-ROS library, and standard library global data are sections where tasks have access to read and write operations. The extended memory regions have a special entry that indicates that there are no other extended memory regions. This configuration helps ensure that tasks are properly isolated and have access only to the memory regions they require.

```
extern PxUInt_t INT32_PUBLISHER_BASE[];
extern PxUInt_t INT32_PUBLISHER_END[];

extern PxUInt_t MICROROS_BASE[];
extern PxUInt_t MICROROS_END[];

extern PxUInt_t LIBRARY_DATA_BASE[];
extern PxUInt_t LIBRARY_DATA_END[];

/* TASK DATA CONTEXT
 * Data regions that stay permanently programmed in Task MPU regions
 * Notes:
 * .lowerbound = 0 : region inherited from the parent
 * .lowerbound = .uppebound : no valid region
 */
static const PxTaskContext_T task_Context =
{
    .protection[0] =
    {
        0,
        0,
        NoAccessProtection
    },
    .protection[1] =
    {
        0,
        0,
        NoAccessProtection
    }
};

/* TASK EXTENDED MEMORY REGIONS
 * Any memory region the  task needs to access outside task's stack and global
    data region
 * stated in Task Data Context
 */
static const PxProtectRegion_T  taskAPRegions[] =
{
  { (unsigned int) INT32_PUBLISHER_BASE,  (unsigned int) INT32_PUBLISHER_END,
    WRProtection },
  { (unsigned int) MICROROS_BASE, (unsigned int) MICROROS_END, WRProtection },
  { (unsigned int) LIBRARY_DATA_BASE, (unsigned int) LIBRARY_DATA_END,
    WRProtection },
    {0, 0, NoAccessProtection}
};
```

Code listing 3.24 Task memory access definition for correct MPU settings.

To configure the task specification structure, the task's name and function must be specified. Below is an example showing the configuration for *int32_publisher*:

```
/* Configure Task specification structure */
task_Spec.ts_name = (const PxChar_t *)"Task int32_publisher";
task_Spec.ts_fun = int32_publisher_taskFunc;
```

3.4.4 Multithread_publisher_subscriber

The *multithread_publisher_subscriber* demo consists of two task functions, one for the main task where subscribers run, and the other for the publisher tasks that run separately. A configuration file for the publisher tasks is added to the task files. Following the micro-ROS initialization (before calling the executor), the publisher tasks are created with lower priority. Once created, the publisher tasks wait for the initial message from the main task that contains the pointer to the publisher structure. This message is sent immediately after calling the publishers task create function. The distribution of the publishers and subscribers running on a single core is shown in 3.6.

3.4.5 Multicore_publisher_subscriber

The *multicore_publisher_subscriber* example is designed to demonstrate the capability of PXROS-HR as a multicore RTOS for distributing the load of tasks across multiple cores. This example consists of two task functions, one for the main task where the subscribers run, and one for the publisher tasks that run on different cores.

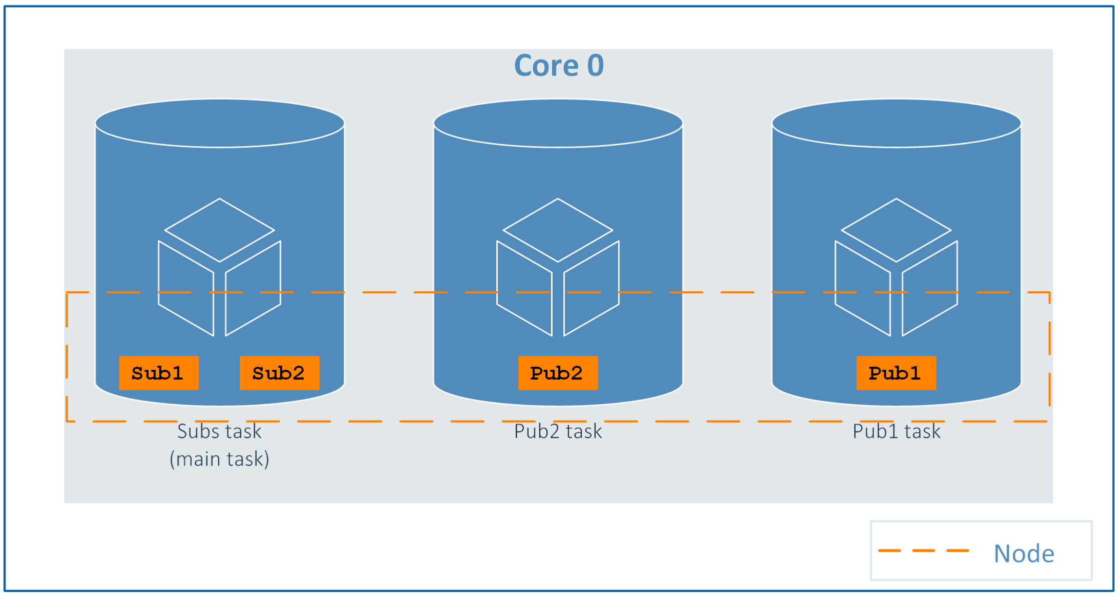


Figure 3.6 Multithread demo distribution of publishers and subscribers.

In this example, the create function of the publisher tasks is assigned to the task deployment table and they are created by the Init task, because PXROS-HR does not allow a task to create a new task on a different core. When the publisher tasks start their execution, they register their task IDs to the Name Server so that they can be accessed by the main task. Then, they wait for the initial message from the main task.

Meanwhile, the main task initializes the micro-ROS and gets the IDs of the publisher tasks from the Name Server before calling the executor. The main task then sends the pointer to the publisher structure to each publisher task. The publisher tasks use the synchronization with the PxNet described above and copy the service mailbox entry to their local table of service mailboxes.

The figure 3.7 displays the distribution of publishers and subscribers running on multiple cores but within one node.

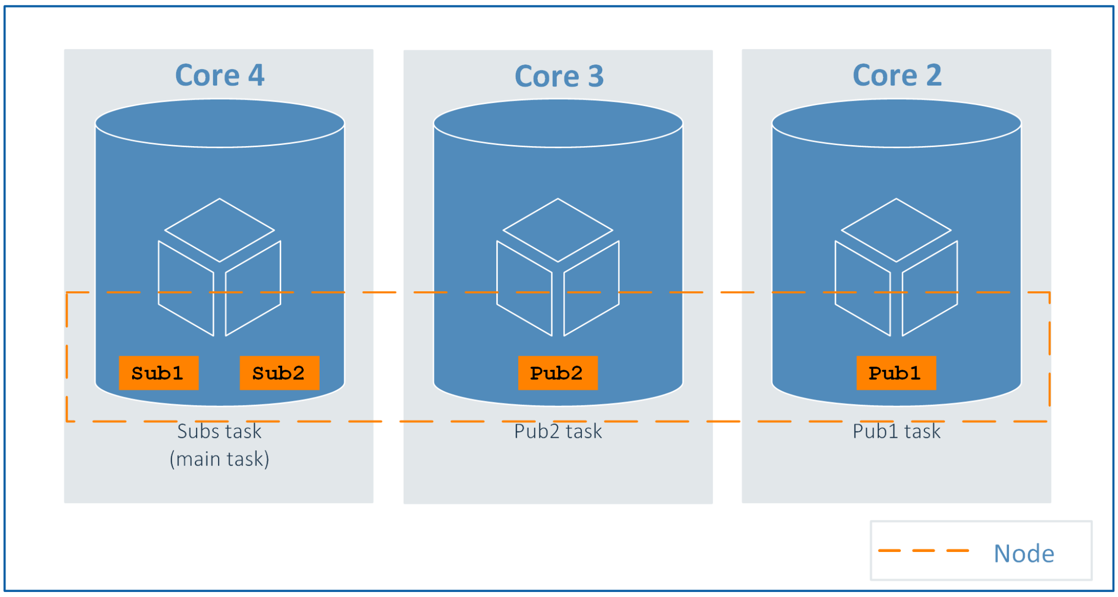


Figure 3.7 Multicore demo distribution of publishers and subscribers.

Chapter 4

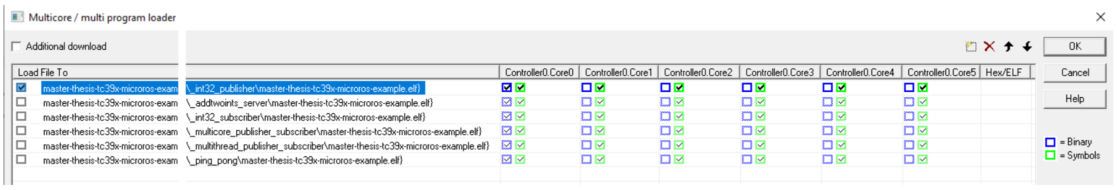
Testing

To validate the proposed implementation, the focus was on testing the micro-ROS basic concept of communication. The testing environment used was based on the Application Kit with TC397 B-Step development board. The TriCore chip was flashed and debugged using a PLS hardware debugger connected to a Windows PC running the UDE (Universal Debug Engine) debugger GUI. The development board was connected via USB cable to the Windows PC to print UART debug output and via Ethernet cable to a laptop running native Ubuntu 22.04.2 LTS (Jammy Jellyfish). On this laptop, the micro-ROS agent version Humble was installed according to the documentation ([27]).

The testing process involved:

1. Flash the demo ".elf" file to the MCU
2. Run the micro-ROS agent
3. Run the demo in the MCU
4. Run the corresponding ROS 2 commands for each demo
5. Observe the behavior based on the outputs of the commands, UART debug output from the MCU, and Wireshark logs

The custom transports file had to be updated with the IP address of the machine where the micro-ROS agent runs, along with the port specified by the user when running the micro-ROS agent command. The UDE was configured for the Application Kit with TC39x B-Step (DAP). Testing was done for each demo separately.



Multicore / multi program loader

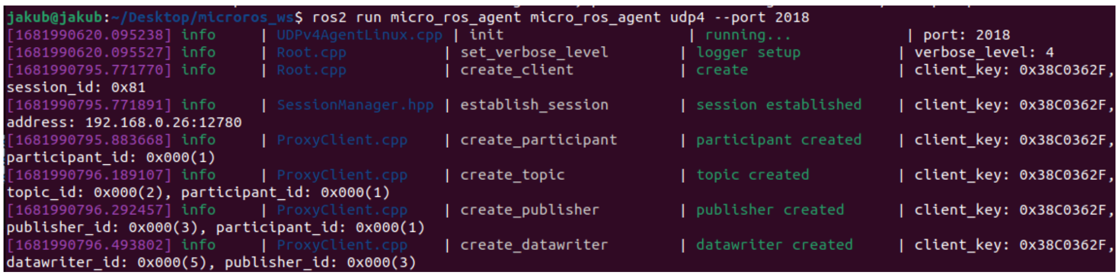
☐ Additional download

	Load File To		Controller0.Core0	Controller0.Core1	Controller0.Core2	Controller0.Core3	Controller0.Core4	Controller0.Core5	Hex/ELF
☑	master-thesis-tc39x-microros-exam	_int32_publisher\master-thesis-tc39x-microros-example.elf}	☑ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	
☐	master-thesis-tc39x-microros-exam	_addtwoints_server\master-thesis-tc39x-microros-example.elf}	☑ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	
☐	master-thesis-tc39x-microros-exam	_int32_subscriber\master-thesis-tc39x-microros-example.elf}	☑ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	
☐	master-thesis-tc39x-microros-exam	_multicore_publisher_subscriber\master-thesis-tc39x-microros-example.elf}	☑ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	
☐	master-thesis-tc39x-microros-exam	_multithread_publisher_subscriber\master-thesis-tc39x-microros-example.elf}	☑ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	
☐	master-thesis-tc39x-microros-exam	_ping_pong\master-thesis-tc39x-microros-example.elf}	☑ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	☐ ☑	

Figure 4.1 UDE program loader with all the demo ELF files – binary is loaded only to core 0, symbols are loaded for each core.

4.1 Int32_publisher

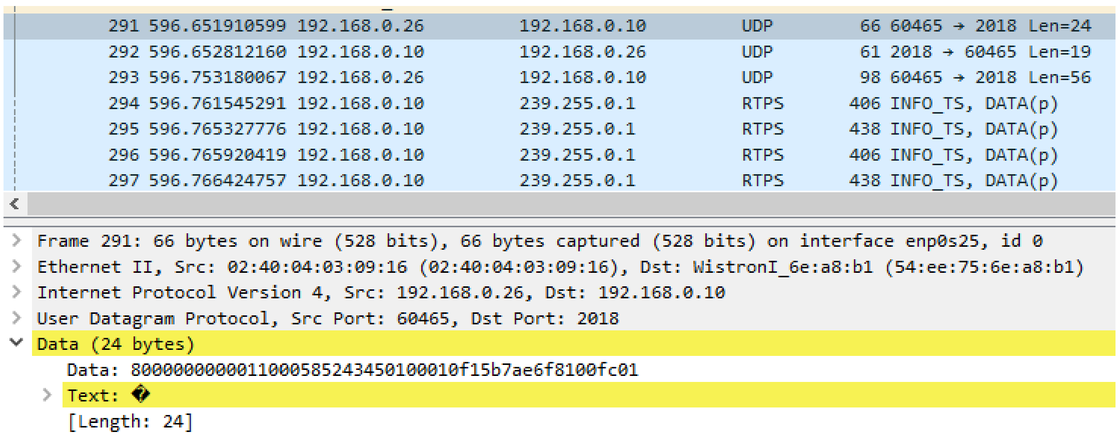
The *int32_publisher* example can be considered the simplest among the tested demos. The micro-ROS agent output (as shown in Figure 4.2, although it does not match the verbose output due to different sessions) confirms that the micro-ROS client was successfully created and the session was established. The log messages indicating that the ”topic created” and ”publisher created” were the most important, signifying the successful creation of the publisher.



```
jakub@jakub:~/Desktop/microros_ws$ ros2 run micro_ros_agent micro_ros_agent udp4 --port 2018
[1681990620.095238] info     | UDPv4AgentLinux.cpp | init                   | running...             | port: 2018
[1681990620.095527] info     | Root.cpp           | set_verbose_level      | logger setup           | verbose_level: 4
[1681990795.771770] info     | Root.cpp           | create_client          | create                 | client_key: 0x38C0362F,
session_id: 0x81
[1681990795.771891] info     | SessionManager.hpp | establish_session      | session established    | client_key: 0x38C0362F,
address: 192.168.0.26:12780
[1681990795.883668] info     | ProxyClient.cpp    | create_participant     | participant created    | client_key: 0x38C0362F,
participant_id: 0x000(1)
[1681990796.189107] info     | ProxyClient.cpp    | create_topic           | topic created          | client_key: 0x38C0362F,
topic_id: 0x000(2), participant_id: 0x000(1)
[1681990796.292457] info     | ProxyClient.cpp    | create_publisher       | publisher created      | client_key: 0x38C0362F,
publisher_id: 0x000(3), participant_id: 0x000(1)
[1681990796.493802] info     | ProxyClient.cpp    | create_datawriter      | datawriter created     | client_key: 0x38C0362F,
datawriter_id: 0x000(5), publisher_id: 0x000(3)
```

Figure 4.2 Micro-ROS agent output of *int32_publisher* demo.

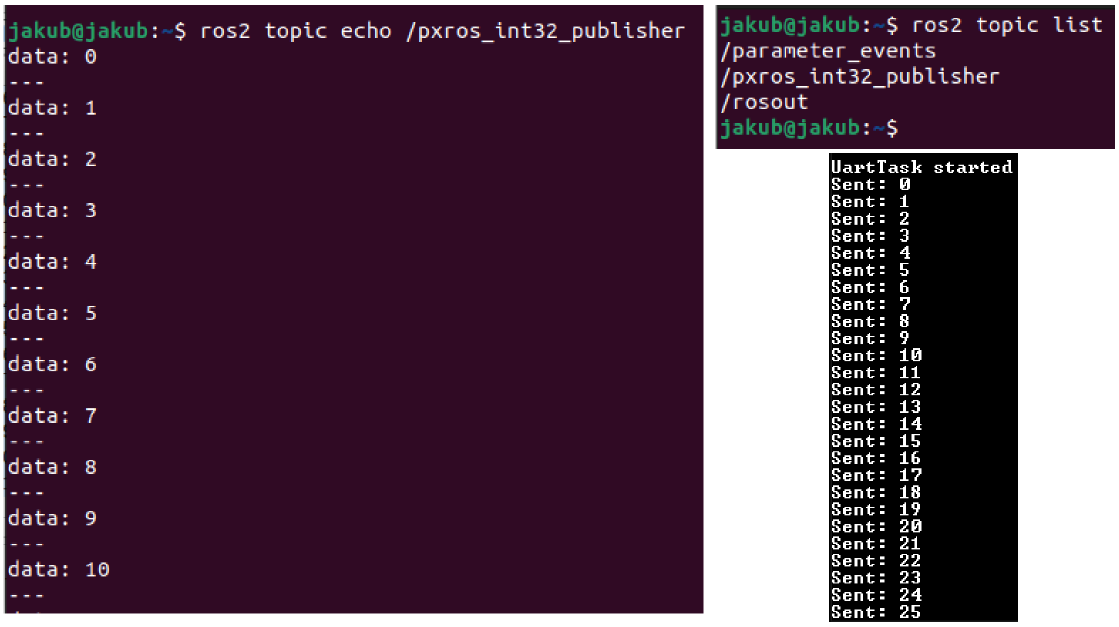
The Wireshark log in 4.3 shows the UDP packet sent from the development board to the laptop during the testing of *int32_publisher*. The data in this packet matches the data from the micro-ROS agent verbose output shown in 4.6, indicating that the communication is functioning properly and that the implementation of custom transports is successful.



```
291 596.651910599 192.168.0.26    192.168.0.10    UDP     66 60465 → 2018 Len=24
292 596.652812160 192.168.0.10    192.168.0.26    UDP     61 2018 → 60465 Len=19
293 596.753180067 192.168.0.26    192.168.0.10    UDP     98 60465 → 2018 Len=56
294 596.761545291 192.168.0.10    239.255.0.1     RTPS   406 INFO_TS, DATA(p)
295 596.765327776 192.168.0.10    239.255.0.1     RTPS   438 INFO_TS, DATA(p)
296 596.765920419 192.168.0.10    239.255.0.1     RTPS   406 INFO_TS, DATA(p)
297 596.766424757 192.168.0.10    239.255.0.1     RTPS   438 INFO_TS, DATA(p)

> Frame 291: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface enp0s25, id 0
> Ethernet II, Src: 02:40:04:03:09:16 (02:40:04:03:09:16), Dst: WistronI_6e:a8:b1 (54:ee:75:6e:a8:b1)
> Internet Protocol Version 4, Src: 192.168.0.26, Dst: 192.168.0.10
> User Datagram Protocol, Src Port: 60465, Dst Port: 2018
v Data (24 bytes)
    Data: 800000000000110005852434501000010f15b7ae6f8100fc01
  > Text: �
    [Length: 24]
```

Figure 4.3 Wireshark log showing the creation of the micro-ROS client. The *Data* matches the data of the verbose output of the micro-ROS agent.

In the top right corner of Figure 4.4, the available topics can be seen, with the topic */pxros_int32_publisher* created by the MCU. The echo of this topic shows the data published to it, and the bottom right corner shows the UART debug output, confirming that the timer, timer callback, and executor are functioning properly.



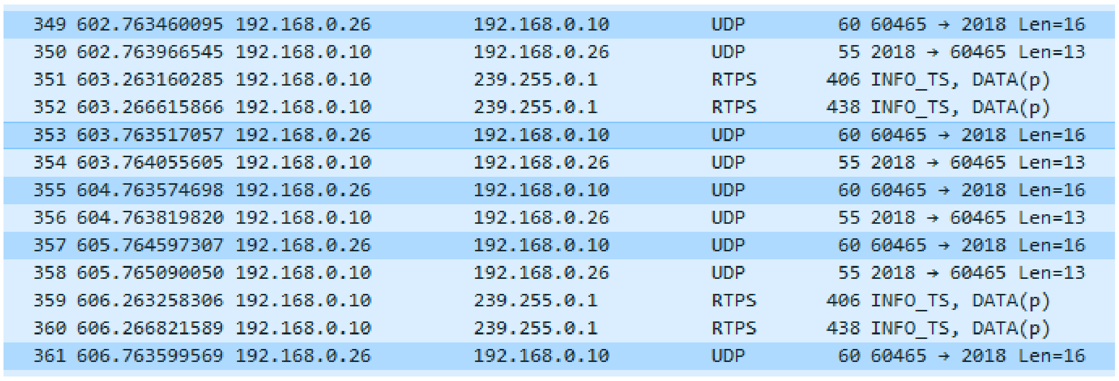
```
jakub@jakub:~$ ros2 topic echo /pxros_int32_publisher
data: 0
---
data: 1
---
data: 2
---
data: 3
---
data: 4
---
data: 5
---
data: 6
---
data: 7
---
data: 8
---
data: 9
---
data: 10
---
```

```
jakub@jakub:~$ ros2 topic list
/parameter_events
/pxros_int32_publisher
/rosout
jakub@jakub:~$
```

```
UartTask started
Sent: 0
Sent: 1
Sent: 2
Sent: 3
Sent: 4
Sent: 5
Sent: 6
Sent: 7
Sent: 8
Sent: 9
Sent: 10
Sent: 11
Sent: 12
Sent: 13
Sent: 14
Sent: 15
Sent: 16
Sent: 17
Sent: 18
Sent: 19
Sent: 20
Sent: 21
Sent: 22
Sent: 23
Sent: 24
Sent: 25
```

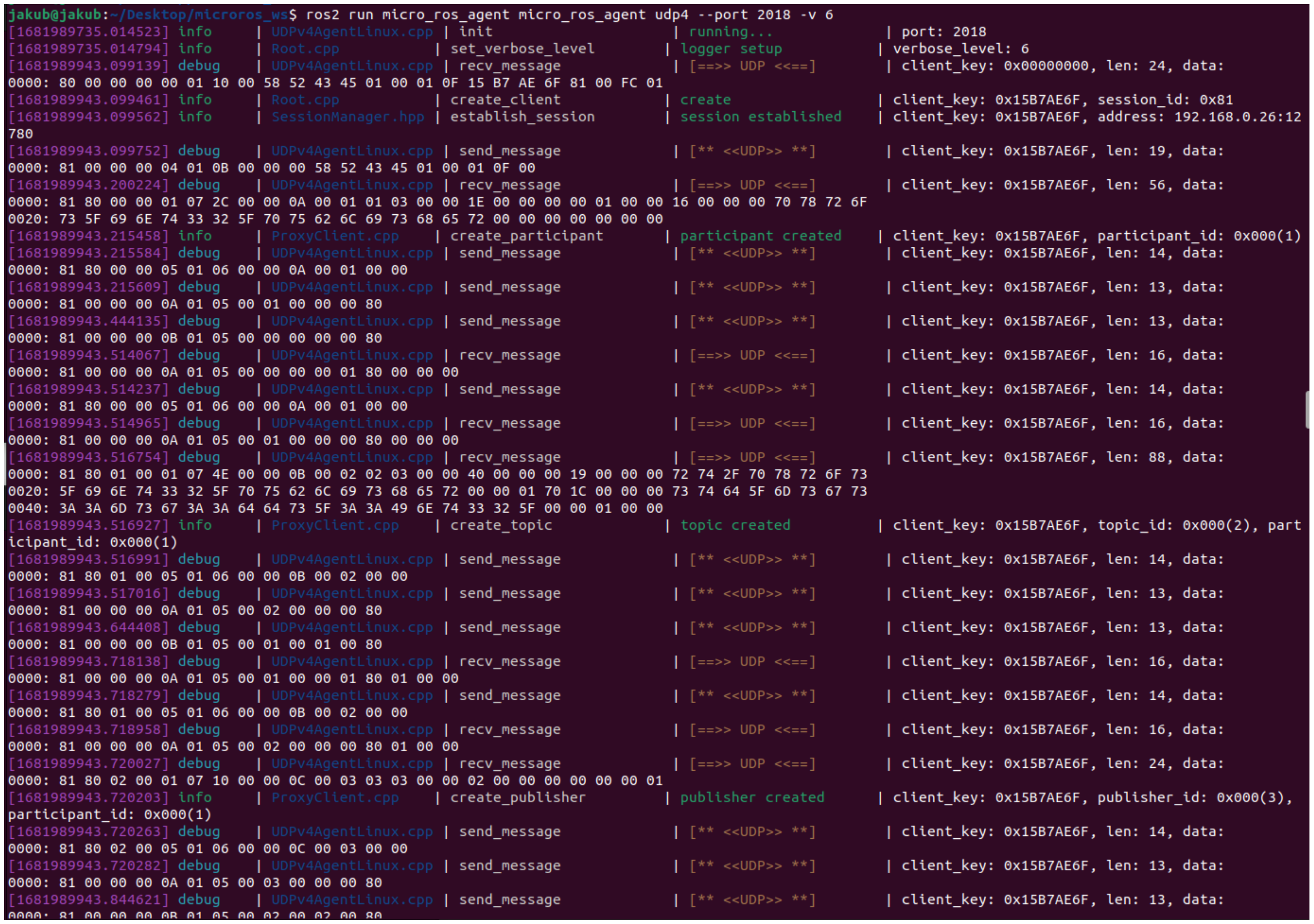
Figure 4.4 Publisher topic registration check, the echo of the publisher topic and UART debug output from the MCU.

From the Wireshark log (4.5), it can be observed that the UDP packet is sent every second, which corresponds to the timer settings defined in the demo.



```
349 602.763460095 192.168.0.26     192.168.0.10     UDP      60 60465 → 2018 Len=16
350 602.763966545 192.168.0.10     192.168.0.26     UDP      55 2018 → 60465 Len=13
351 603.263160285 192.168.0.10     239.255.0.1      RTPS    406 INFO_TS, DATA(p)
352 603.266615866 192.168.0.10     239.255.0.1      RTPS    438 INFO_TS, DATA(p)
353 603.763517057 192.168.0.26     192.168.0.10     UDP      60 60465 → 2018 Len=16
354 603.764055605 192.168.0.10     192.168.0.26     UDP      55 2018 → 60465 Len=13
355 604.763574698 192.168.0.26     192.168.0.10     UDP      60 60465 → 2018 Len=16
356 604.763819820 192.168.0.10     192.168.0.26     UDP      55 2018 → 60465 Len=13
357 605.764597307 192.168.0.26     192.168.0.10     UDP      60 60465 → 2018 Len=16
358 605.765090050 192.168.0.10     192.168.0.26     UDP      55 2018 → 60465 Len=13
359 606.263258306 192.168.0.10     239.255.0.1      RTPS    406 INFO_TS, DATA(p)
360 606.266821589 192.168.0.10     239.255.0.1      RTPS    438 INFO_TS, DATA(p)
361 606.763599569 192.168.0.26     192.168.0.10     UDP      60 60465 → 2018 Len=16
```

Figure 4.5 Wireshark log approving that the publish is done each second.

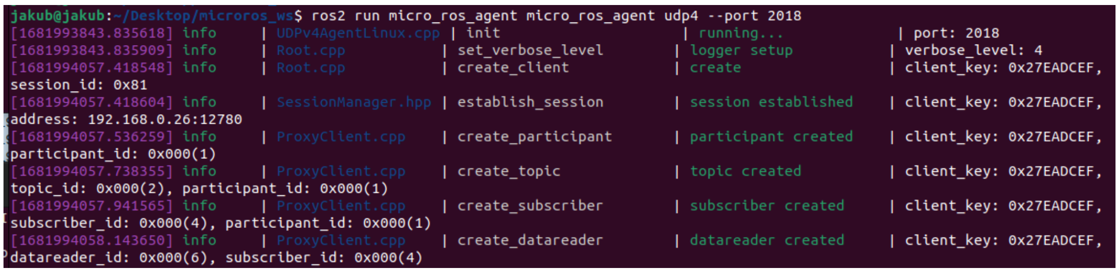


```
jakub@jakub:~/Desktop/microros_ws$ ros2 run micro_ros_agent micro_ros_agent udp4 --port 2018 -v 6
[1681989735.014523] info     | UDPv4AgentLinux.cpp | init                     | running...            | port: 2018
[1681989735.014794] info     | Root.cpp            | set_verbose_level        | logger setup          | verbose_level: 6
[1681989943.099139] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]       | client_key: 0x00000000, len: 24, data:
0000: 80 00 00 00 00 01 10 00 58 52 43 45 01 00 01 0F 15 B7 AE 6F 81 00 FC 01
[1681989943.099461] info     | Root.cpp            | create_client            | create                | client_key: 0x15B7AE6F, session_id: 0x81
[1681989943.099562] info     | SessionManager.hpp  | establish_session        | session established   | client_key: 0x15B7AE6F, address: 192.168.0.26:12
780
[1681989943.099752] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 19, data:
0000: 81 00 00 00 04 01 0B 00 00 00 58 52 43 45 01 00 01 0F 00
[1681989943.200224] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]       | client_key: 0x15B7AE6F, len: 56, data:
0000: 81 80 00 00 01 07 2C 00 00 0A 00 01 01 03 00 00 1E 00 00 00 01 00 00 16 00 00 00 70 78 72 6F
0020: 73 5F 69 6E 74 33 32 5F 70 75 62 6C 69 73 68 65 72 00 00 00 00 00 00 00
[1681989943.215458] info     | ProxyClient.cpp     | create_participant       | participant created   | client_key: 0x15B7AE6F, participant_id: 0x000(1)
[1681989943.215584] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 14, data:
0000: 81 80 00 00 05 01 06 00 00 0A 00 01 00 00
[1681989943.215609] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
[1681989943.444135] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1681989943.514067] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]       | client_key: 0x15B7AE6F, len: 16, data:
0000: 81 00 00 00 0A 01 05 00 00 00 01 80 00 00 00
[1681989943.514237] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 14, data:
0000: 81 80 00 00 05 01 06 00 00 0A 00 01 00 00
[1681989943.514965] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]       | client_key: 0x15B7AE6F, len: 16, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80 00 00 00
[1681989943.516754] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]       | client_key: 0x15B7AE6F, len: 88, data:
0000: 81 80 01 00 01 07 4E 00 00 0B 00 02 02 03 00 00 40 00 00 00 19 00 00 00 72 74 2F 70 78 72 6F 73
0020: 5F 69 6E 74 33 32 5F 70 75 62 6C 69 73 68 65 72 00 00 01 70 1C 00 00 00 73 74 64 5F 6D 73 67 73
0040: 3A 3A 6D 73 67 3A 3A 64 64 73 5F 3A 3A 49 6E 74 33 32 5F 00 00 01 00 00
[1681989943.516927] info     | ProxyClient.cpp     | create_topic             | topic created         | client_key: 0x15B7AE6F, topic_id: 0x000(2), part
icipant_id: 0x000(1)
[1681989943.516991] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 14, data:
0000: 81 80 01 00 05 01 06 00 00 0B 00 02 00 00
[1681989943.517016] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 02 00 00 00 80
[1681989943.644408] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 01 00 01 00 80
[1681989943.718138] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]       | client_key: 0x15B7AE6F, len: 16, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 01 80 01 00 00
[1681989943.718279] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 14, data:
0000: 81 80 01 00 05 01 06 00 00 0B 00 02 00 00
[1681989943.718958] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]       | client_key: 0x15B7AE6F, len: 16, data:
0000: 81 00 00 00 0A 01 05 00 02 00 00 00 80 01 00 00
[1681989943.720027] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <<==]       | client_key: 0x15B7AE6F, len: 24, data:
0000: 81 80 02 00 01 07 10 00 00 0C 00 03 03 03 00 00 02 00 00 00 00 00 00 01
[1681989943.720203] info     | ProxyClient.cpp     | create_publisher         | publisher created     | client_key: 0x15B7AE6F, publisher_id: 0x000(3),
participant_id: 0x000(1)
[1681989943.720263] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 14, data:
0000: 81 80 02 00 05 01 06 00 00 0C 00 03 00 00
[1681989943.720282] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 03 00 00 00 80
[1681989943.844621] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>> **]       | client_key: 0x15B7AE6F, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 02 00 02 00 80
```

Figure 4.6 Micro-ROS agent verbose output of *int32_publisher* demo.

4.2 Int32_subscriber

Similarly to the publisher, the successful creation of the subscriber can be seen in the micro-ROS agent output (4.7).



```
jakub@jakub:~/Desktop/microros_ws$ ros2 run micro_ros_agent micro_ros_agent udp4 --port 2018
[1681993843.835618] info     | UDPv4AgentLinux.cpp | init                  | running...             | port: 2018
[1681993843.835909] info     | Root.cpp           | set_verbose_level      | logger setup           | verbose_level: 4
[1681994057.418548] info     | Root.cpp           | create_client          | create                 | client_key: 0x27EADCEF,
session_id: 0x81
[1681994057.418604] info     | SessionManager.hpp | establish_session      | session established    | client_key: 0x27EADCEF,
address: 192.168.0.26:12780
[1681994057.536259] info     | ProxyClient.cpp    | create_participant     | participant created    | client_key: 0x27EADCEF,
participant_id: 0x000(1)
[1681994057.738355] info     | ProxyClient.cpp    | create_topic           | topic created          | client_key: 0x27EADCEF,
topic_id: 0x000(2), participant_id: 0x000(1)
[1681994057.941565] info     | ProxyClient.cpp    | create_subscriber      | subscriber created     | client_key: 0x27EADCEF,
subscriber_id: 0x000(4), participant_id: 0x000(1)
[1681994058.143650] info     | ProxyClient.cpp    | create_datareader      | datareader created     | client_key: 0x27EADCEF,
datareader_id: 0x000(6), subscriber_id: 0x000(4)
```

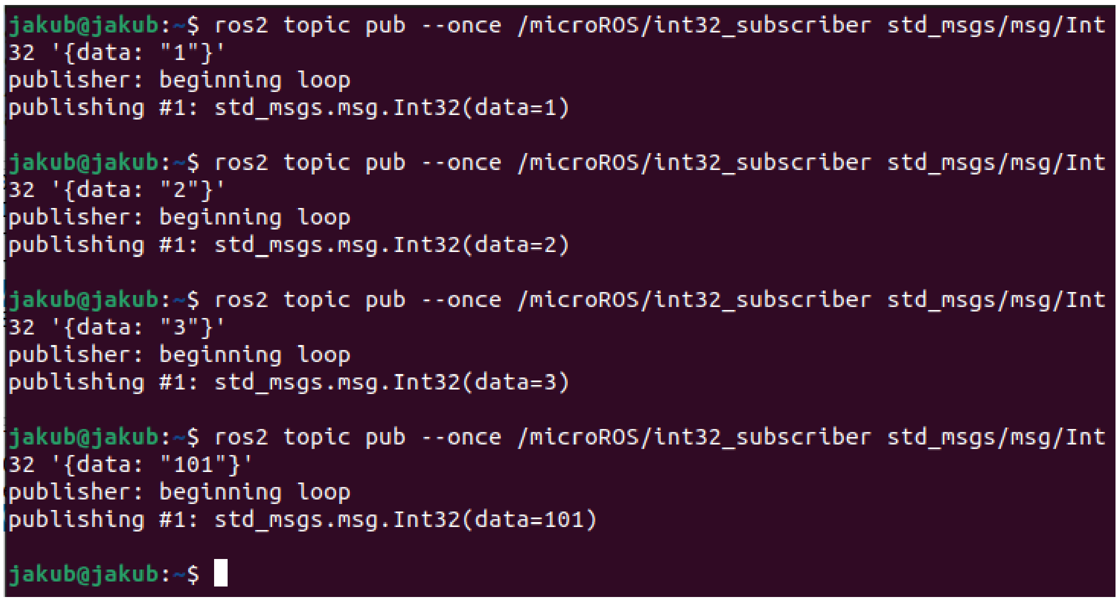
Figure 4.7 Micro-ROS agent output of *int32_subscriber* demo.

As shown in the top-right corner of 4.9, the subscriber topic *MicroROS/int32_subscriber* is available, indicating that the subscriber was successfully created. To verify that the subscriber is receiving messages, the following command can be executed:

```
$ ros2 topic pub --once /microROS/int32_subscriber std_msgs/msg/Int32 '{data:
    "1"}'
```

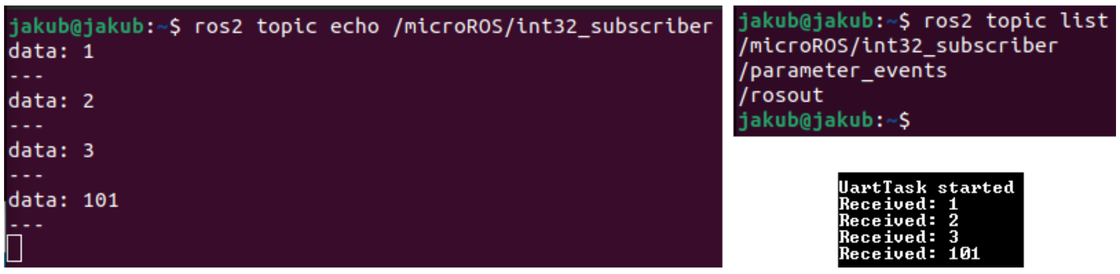
To verify that the subscriber is working properly, commands to publish messages containing the 32-bit integers "1," "2," "3," and "101" to the topic */microROS/int32_subscriber* were executed. This test aimed to check for the successful publishing of messages, which was confirmed by observing the echo of the topic. All four messages were displayed in the echo, indicating that they were published correctly (shown in the left image of 4.9).

Furthermore, the UART debug output was checked, and all the messages were printed with the correct values (shown in the right low corner of 4.9). This indicates that the subscriber was successfully receiving and processing published messages.



```
jakub@jakub:~$ ros2 topic pub --once /microROS/int32_subscriber std_msgs/msg/Int
32 '{data: "1"}'
publisher: beginning loop
publishing #1: std_msgs.msg.Int32(data=1)

jakub@jakub:~$ ros2 topic pub --once /microROS/int32_subscriber std_msgs/msg/Int
32 '{data: "2"}'
publisher: beginning loop
publishing #1: std_msgs.msg.Int32(data=2)

jakub@jakub:~$ ros2 topic pub --once /microROS/int32_subscriber std_msgs/msg/Int
32 '{data: "3"}'
publisher: beginning loop
publishing #1: std_msgs.msg.Int32(data=3)

jakub@jakub:~$ ros2 topic pub --once /microROS/int32_subscriber std_msgs/msg/Int
32 '{data: "101"}'
publisher: beginning loop
publishing #1: std_msgs.msg.Int32(data=101)

jakub@jakub:~$ █
```

Figure 4.8 Publishing a message containing 32-bit integer value for a subscriber.



```
jakub@jakub:~$ ros2 topic echo /microROS/int32_subscriber
data: 1
---
data: 2
---
data: 3
---
data: 101
---
```

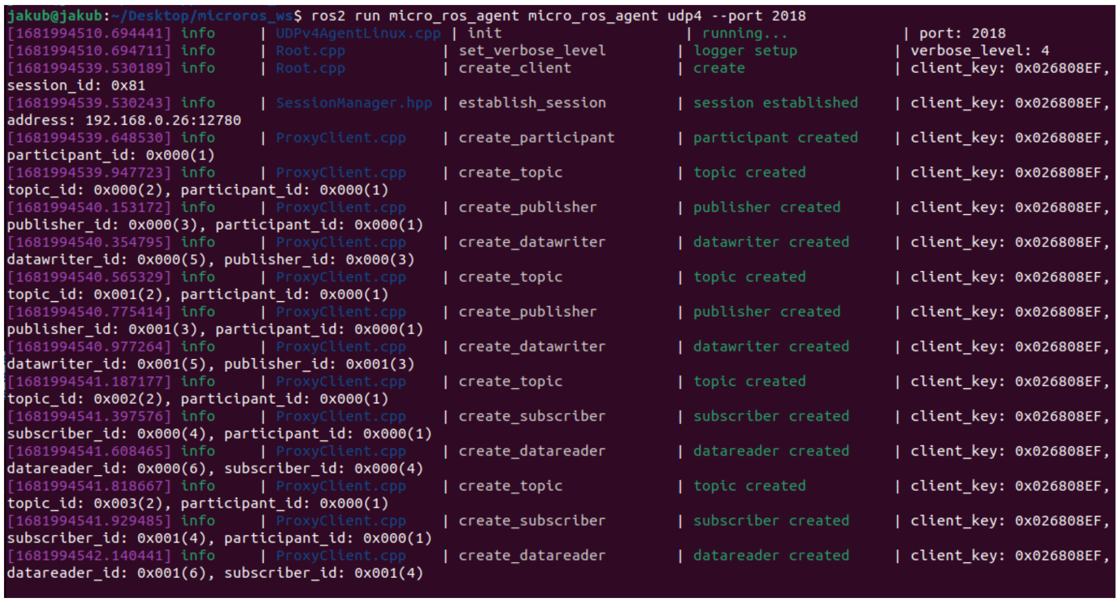
```
jakub@jakub:~$ ros2 topic list
/microROS/int32_subscriber
/parameter_events
/rosout
jakub@jakub:~$
```

```
UartTask started
Received: 1
Received: 2
Received: 3
Received: 101
```

Figure 4.9 Subscriber topic registration check, the echo of the subscriber topic and UART debug output from the MCU.

4.3 Ping_pong

The ping-pong demo involves two publishers and two subscribers, making it a more advanced demo. All publishers and subscribers with the two topics used in this demo can be found in the micro-ROS agent output (4.10), which indicates that their creation was successful.



```
jakub@jakub:~/Desktop/microros_ws$ ros2 run micro_ros_agent micro_ros_agent udp4 --port 2018
[1681994510.694441] info     | UDPv4AgentLinux.cpp | init                   | running...            | port: 2018
[1681994510.694711] info     | Root.cpp           | set_verbose_level      | logger setup          | verbose_level: 4
[1681994539.530189] info     | Root.cpp           | create_client          | create                | client_key: 0x026808EF,
session_id: 0x81
[1681994539.530243] info     | SessionManager.hpp | establish_session      | session established   | client_key: 0x026808EF,
address: 192.168.0.26:12780
[1681994539.648530] info     | ProxyClient.cpp    | create_participant     | participant created   | client_key: 0x026808EF,
participant_id: 0x000(1)
[1681994539.947723] info     | ProxyClient.cpp    | create_topic           | topic created         | client_key: 0x026808EF,
topic_id: 0x000(2), participant_id: 0x000(1)
[1681994540.153172] info     | ProxyClient.cpp    | create_publisher       | publisher created     | client_key: 0x026808EF,
publisher_id: 0x000(3), participant_id: 0x000(1)
[1681994540.354795] info     | ProxyClient.cpp    | create_datawriter      | datawriter created    | client_key: 0x026808EF,
datawriter_id: 0x000(5), publisher_id: 0x000(3)
[1681994540.565329] info     | ProxyClient.cpp    | create_topic           | topic created         | client_key: 0x026808EF,
topic_id: 0x001(2), participant_id: 0x000(1)
[1681994540.775414] info     | ProxyClient.cpp    | create_publisher       | publisher created     | client_key: 0x026808EF,
publisher_id: 0x001(3), participant_id: 0x000(1)
[1681994540.977264] info     | ProxyClient.cpp    | create_datawriter      | datawriter created    | client_key: 0x026808EF,
datawriter_id: 0x001(5), publisher_id: 0x001(3)
[1681994541.187177] info     | ProxyClient.cpp    | create_topic           | topic created         | client_key: 0x026808EF,
topic_id: 0x002(2), participant_id: 0x000(1)
[1681994541.397576] info     | ProxyClient.cpp    | create_subscriber      | subscriber created    | client_key: 0x026808EF,
subscriber_id: 0x000(4), participant_id: 0x000(1)
[1681994541.608465] info     | ProxyClient.cpp    | create_datareader      | datareader created    | client_key: 0x026808EF,
datareader_id: 0x000(6), subscriber_id: 0x000(4)
[1681994541.818667] info     | ProxyClient.cpp    | create_topic           | topic created         | client_key: 0x026808EF,
topic_id: 0x003(2), participant_id: 0x000(1)
[1681994541.929485] info     | ProxyClient.cpp    | create_subscriber      | subscriber created    | client_key: 0x026808EF,
subscriber_id: 0x001(4), participant_id: 0x000(1)
[1681994542.140441] info     | ProxyClient.cpp    | create_datareader      | datareader created    | client_key: 0x026808EF,
datareader_id: 0x001(6), subscriber_id: 0x001(4)
```

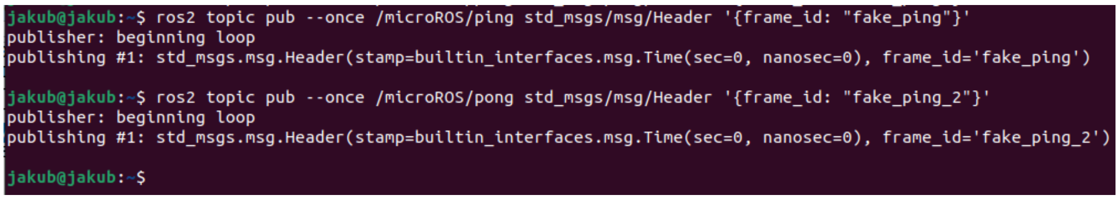
Figure 4.10 Micro-ROS agent output of *ping_pong* demo.

In the demo, pings are published to the */microROS/ping* topic every two seconds, as observed in the left image of 4.12, as well as in the UART output (right low corner of 4.12), where they have matching *frame_id*.

To demonstrate that the application can respond to custom pings, a fake ping with the *frame_id* called *fake_ping* was published to */microROS/ping*. The following command was used to achieve this:

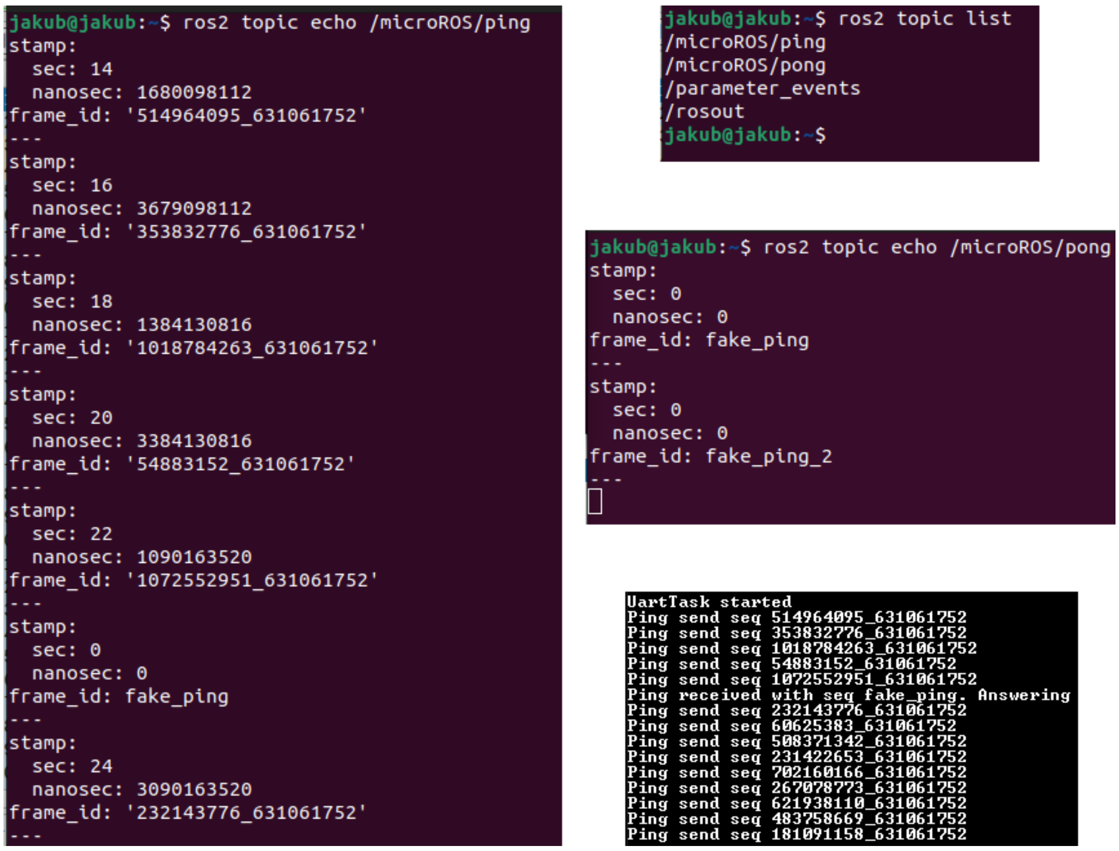
```
$ ros2 topic pub --once /microROS/ping/ std_msgs/msg/Header '{frame_id: "
    fake_ping"}'
```

The ping should be visible in the echo of */microROS/ping*, the UART debug output, and the echo of */microROS/pong* if the subscriber recognizes that the ping was not created by the demo in MCU. As observed in 4.12, the subscriber for the */microROS/ping* topic correctly recognized the external pings and responded with a pong to the */microROS/pong* topic, indicating that the application was working properly.



```
jakub@jakub:~$ ros2 topic pub --once /microROS/ping std_msgs/msg/Header '{frame_id: "fake_ping"}'
publisher: beginning loop
publishing #1: std_msgs.msg.Header(stamp=builtin_interfaces.msg.Time(sec=0, nanosec=0), frame_id='fake_ping')

jakub@jakub:~$ ros2 topic pub --once /microROS/pong std_msgs/msg/Header '{frame_id: "fake_ping_2"}'
publisher: beginning loop
publishing #1: std_msgs.msg.Header(stamp=builtin_interfaces.msg.Time(sec=0, nanosec=0), frame_id='fake_ping_2')

jakub@jakub:~$
```

Figure 4.11 Publishing a message containing *fake_ping* for a ping subscriber.



```
jakub@jakub:~$ ros2 topic echo /microROS/ping
stamp:
  sec: 14
  nanosec: 1680098112
frame_id: '514964095_631061752'
---
stamp:
  sec: 16
  nanosec: 3679098112
frame_id: '353832776_631061752'
---
stamp:
  sec: 18
  nanosec: 1384130816
frame_id: '1018784263_631061752'
---
stamp:
  sec: 20
  nanosec: 3384130816
frame_id: '54883152_631061752'
---
stamp:
  sec: 22
  nanosec: 1090163520
frame_id: '1072552951_631061752'
---
stamp:
  sec: 0
  nanosec: 0
frame_id: fake_ping
---
stamp:
  sec: 24
  nanosec: 3090163520
frame_id: '232143776_631061752'
---
```

```
jakub@jakub:~$ ros2 topic list
/microROS/ping
/microROS/pong
/parameter_events
/rosout
jakub@jakub:~$
```

```
jakub@jakub:~$ ros2 topic echo /microROS/pong
stamp:
  sec: 0
  nanosec: 0
frame_id: fake_ping
---
stamp:
  sec: 0
  nanosec: 0
frame_id: fake_ping_2
---
```

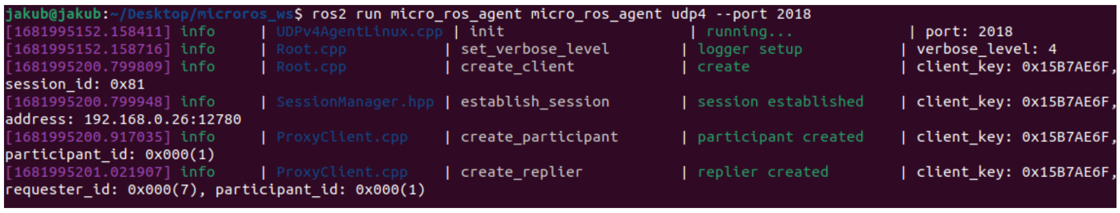
```
UartTask started
Ping send seq 514964095_631061752
Ping send seq 353832776_631061752
Ping send seq 1018784263_631061752
Ping send seq 54883152_631061752
Ping send seq 1072552951_631061752
Ping received with seq fake_ping. Answering
Ping send seq 232143776_631061752
Ping send seq 60625383_631061752
Ping send seq 508371342_631061752
Ping send seq 231422653_631061752
Ping send seq 702160166_631061752
Ping send seq 267078773_631061752
Ping send seq 621938110_631061752
Ping send seq 483758669_631061752
Ping send seq 181091158_631061752
```

Figure 4.12 Ping and pong topics registration check, their echo and UART debug output from the MCU showing that the external *fake_ping* was answered with a pong.

4.4 Addtwoints_server

The *Add two ints* demo in micro-ROS showcases the request-response communication pattern that can be utilized in ROS 2 systems. This demo allows the user to send a request to the MCU to add two integers, and then waits for the MCU to process the request. After processing the request, the MCU returns the desired output as the response. This pattern is useful for delegating high-computing power demanding tasks to more powerful devices.

In the micro-ROS agent log, the service server is created as a ”replier” (see 4.13). This indicates that it is ready to receive requests and provide a response based on those requests.



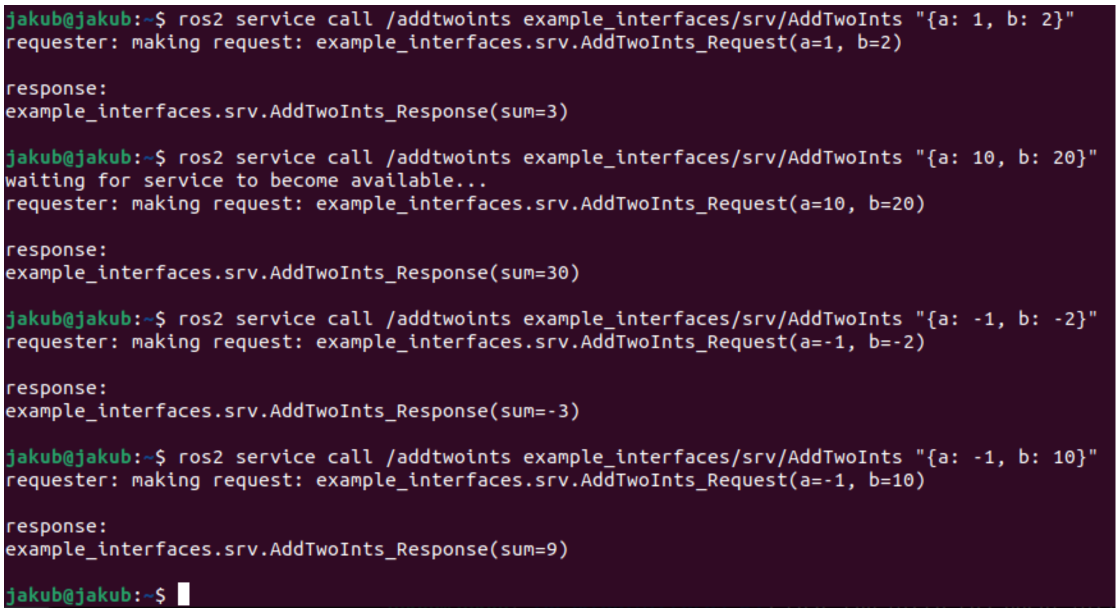
```
jakub@jakub:~/Desktop/microros_ws$ ros2 run micro_ros_agent micro_ros_agent udp4 --port 2018
[1681995152.158411] info     | UDPv4AgentLinux.cpp | init                  | running...             | port: 2018
[1681995152.158716] info     | Root.cpp            | set_verbose_level     | logger setup           | verbose_level: 4
[1681995200.799809] info     | Root.cpp            | create_client         | create                 | client_key: 0x15B7AE6F,
session_id: 0x81
[1681995200.799948] info     | SessionManager.hpp  | establish_session     | session established    | client_key: 0x15B7AE6F,
address: 192.168.0.26:12780
[1681995200.917035] info     | ProxyClient.cpp     | create_participant    | participant created    | client_key: 0x15B7AE6F,
participant_id: 0x000(1)
[1681995201.021907] info     | ProxyClient.cpp     | create_replier        | replier created        | client_key: 0x15B7AE6F,
requester_id: 0x000(7), participant_id: 0x000(1)
```

Figure 4.13 Micro-ROS agent output of *addtwoints_server* demo.

In order to test the Add Two Ints service server, a service call with appropriate arguments needs to be created. The service is available under the name */addtwoints*, which can be confirmed by running the command ”$ ros2 service list”. The service call requires two arguments, *a* and *b*, which represent the two integers to be added. In order to evaluate the functionality of the service server, various scenarios with different arguments (both positive and negative integers) can be tested by executing the following command:

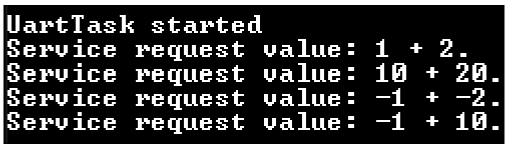
```
$ ros2 service call /addtwoints example_interfaces/srv/AddTwoInts "{a: 1, b: 2}"
```

The service response was displayed directly in the terminal (as shown in 4.14), and the response was correct. Additionally, the UART output printed the request with the passed arguments.



```
jakub@jakub:~$ ros2 service call /addtwoints example_interfaces/srv/AddTwoInts "{a: 1, b: 2}"
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=1, b=2)

response:
example_interfaces.srv.AddTwoInts_Response(sum=3)

jakub@jakub:~$ ros2 service call /addtwoints example_interfaces/srv/AddTwoInts "{a: 10, b: 20}"
waiting for service to become available...
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=10, b=20)

response:
example_interfaces.srv.AddTwoInts_Response(sum=30)

jakub@jakub:~$ ros2 service call /addtwoints example_interfaces/srv/AddTwoInts "{a: -1, b: -2}"
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=-1, b=-2)

response:
example_interfaces.srv.AddTwoInts_Response(sum=-3)

jakub@jakub:~$ ros2 service call /addtwoints example_interfaces/srv/AddTwoInts "{a: -1, b: 10}"
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=-1, b=10)

response:
example_interfaces.srv.AddTwoInts_Response(sum=9)

jakub@jakub:~$
```

Figure 4.14 Service call of the *addtwoints* to add two integers and return the sum.

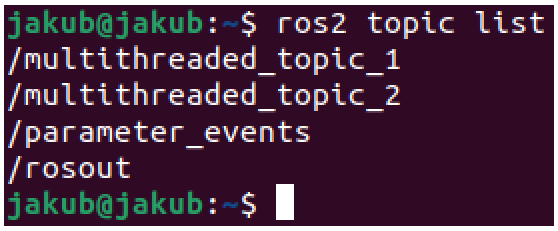


```
UartTask started
Service request value: 1 + 2.
Service request value: 10 + 20.
Service request value: -1 + -2.
Service request value: -1 + 10.
```

Figure 4.15 UART debug output of *addtwoints_server* demo.

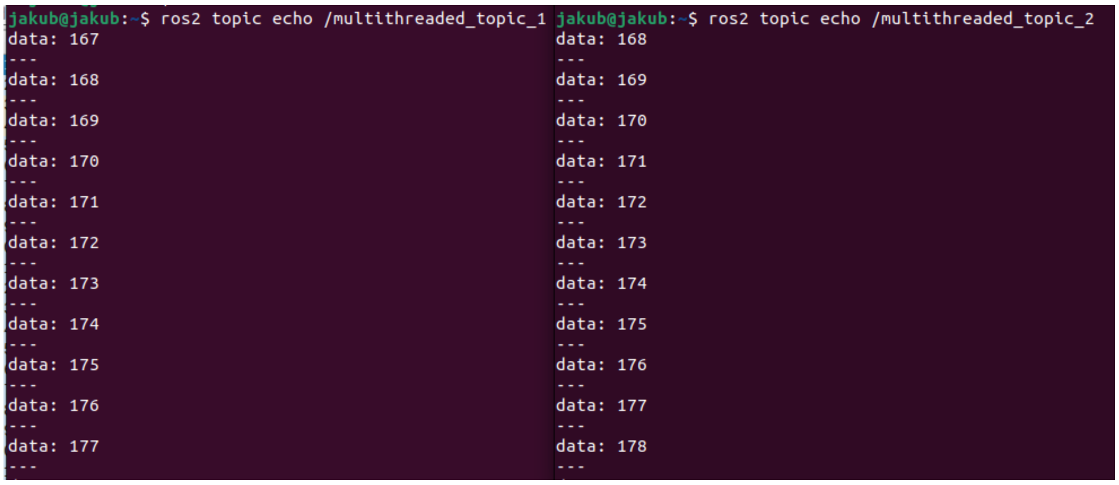
4.5 Multithread_publisher_subscriber

In the multithread example, two publishers and two subscribers are created, which is similar to the *ping_pong* example, so the micro-ROS agent output is not attached. The availability of the */multithreaded_topic_1* and */multithreaded_topic_2* topics can be observed in 4.16. The correct publishing to these topics is confirmed by the output shown in 4.17.



```
jakub@jakub:~$ ros2 topic list
/multithreaded_topic_1
/multithreaded_topic_2
/parameter_events
/rosout
jakub@jakub:~$
```

Figure 4.16 Multithreaded topics registration check.



```
jakub@jakub:~$ ros2 topic echo /multithreaded_topic_1
data: 167
---
data: 168
---
data: 169
---
data: 170
---
data: 171
---
data: 172
---
data: 173
---
data: 174
---
data: 175
---
data: 176
---
data: 177
---
```

```
jakub@jakub:~$ ros2 topic echo /multithreaded_topic_2
data: 168
---
data: 169
---
data: 170
---
data: 171
---
data: 172
---
data: 173
---
data: 174
---
data: 175
---
data: 176
---
data: 177
---
data: 178
---
```

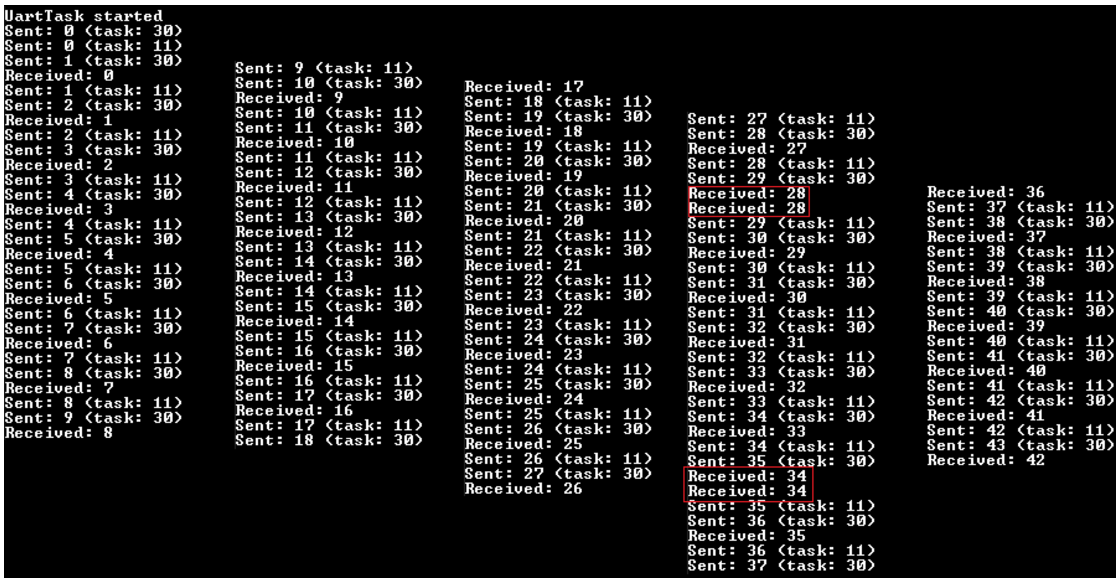
Figure 4.17 Multithreaded topics echo showing that the application running on MCU publishes correctly to the two topics.

The UART output of the publishers can be observed in 4.18 along with their respective task IDs. However, the behavior of the subscribers is inconsistent. One publisher-subscriber pair is set as *default*, which uses reliable communication with acknowledgments, while the other is set as *best effort*, which does not use acknowledgments. As a result, some missed messages may be acceptable. However, further investigation is needed to determine what steps can be taken to ensure proper triggering of the subscribers. As shown in figure 4.18, it is evident that initially, only one subscriber was active. However, later in the process, the second subscriber was triggered, which can be observed from the appearance of red rectangles in the figure.

Due to the difference in approaches used by other RTOSes for running the executor compared to the original demos, tests were conducted to investigate whether the issue with the second subscriber not being executed was caused by a problem with the executor call or due to a high execution load on the core.

The official demos utilize the *rclc_executor_spin* function:

```
rclc_executor_spin(&executor);
```



```
UartTask started
Sent: 0 (task: 30)
Sent: 0 (task: 11)
Sent: 1 (task: 30)
Received: 0
Sent: 1 (task: 11)
Sent: 2 (task: 30)
Received: 1
Sent: 2 (task: 11)
Sent: 3 (task: 30)
Received: 2
Sent: 3 (task: 11)
Sent: 4 (task: 30)
Received: 3
Sent: 4 (task: 11)
Sent: 5 (task: 30)
Received: 4
Sent: 5 (task: 11)
Sent: 6 (task: 30)
Received: 5
Sent: 6 (task: 11)
Sent: 7 (task: 30)
Received: 6
Sent: 7 (task: 11)
Sent: 8 (task: 30)
Received: 7
Sent: 8 (task: 11)
Sent: 9 (task: 30)
Received: 8
```

```
Sent: 9 (task: 11)
Sent: 10 (task: 30)
Received: 9
Sent: 10 (task: 11)
Sent: 11 (task: 30)
Received: 10
Sent: 11 (task: 11)
Sent: 12 (task: 30)
Received: 11
Sent: 12 (task: 11)
Sent: 13 (task: 30)
Received: 12
Sent: 13 (task: 11)
Sent: 14 (task: 30)
Received: 13
Sent: 14 (task: 11)
Sent: 15 (task: 30)
Received: 14
Sent: 15 (task: 11)
Sent: 16 (task: 30)
Received: 15
Sent: 16 (task: 11)
Sent: 17 (task: 30)
Received: 16
Sent: 17 (task: 11)
Sent: 18 (task: 30)
```

```
Received: 17
Sent: 18 (task: 11)
Sent: 19 (task: 30)
Received: 18
Sent: 19 (task: 11)
Sent: 20 (task: 30)
Received: 19
Sent: 20 (task: 11)
Sent: 21 (task: 30)
Received: 20
Sent: 21 (task: 11)
Sent: 22 (task: 30)
Received: 21
Sent: 22 (task: 11)
Sent: 23 (task: 30)
Received: 22
Sent: 23 (task: 11)
Sent: 24 (task: 30)
Received: 23
Sent: 24 (task: 11)
Sent: 25 (task: 30)
Received: 24
Sent: 25 (task: 11)
Sent: 26 (task: 30)
Received: 25
Sent: 26 (task: 11)
Sent: 27 (task: 30)
Received: 26
```

```
Sent: 27 (task: 11)
Sent: 28 (task: 30)
Received: 27
Sent: 28 (task: 11)
Sent: 29 (task: 30)
Received: 28
Received: 28
Sent: 29 (task: 11)
Sent: 30 (task: 30)
Received: 29
Sent: 30 (task: 11)
Sent: 31 (task: 30)
Received: 30
Sent: 31 (task: 11)
Sent: 32 (task: 30)
Received: 31
Sent: 32 (task: 11)
Sent: 33 (task: 30)
Received: 32
Sent: 33 (task: 11)
Sent: 34 (task: 30)
Received: 33
Sent: 34 (task: 11)
Sent: 35 (task: 30)
Received: 34
Received: 34
Sent: 35 (task: 11)
Sent: 36 (task: 30)
Received: 35
Sent: 36 (task: 11)
Sent: 37 (task: 30)
```

```
Received: 36
Sent: 37 (task: 11)
Sent: 38 (task: 30)
Received: 37
Sent: 38 (task: 11)
Sent: 39 (task: 30)
Received: 38
Sent: 39 (task: 11)
Sent: 40 (task: 30)
Received: 39
Sent: 40 (task: 11)
Sent: 41 (task: 30)
Received: 40
Sent: 41 (task: 11)
Sent: 42 (task: 30)
Received: 41
Sent: 42 (task: 11)
Sent: 43 (task: 30)
Received: 42
```

Figure 4.18 UART debug output of *multithread_publisher_subscriber* demo where the *best effort* subscriber does not work properly. The red rectangles show that both subscribers received a message.

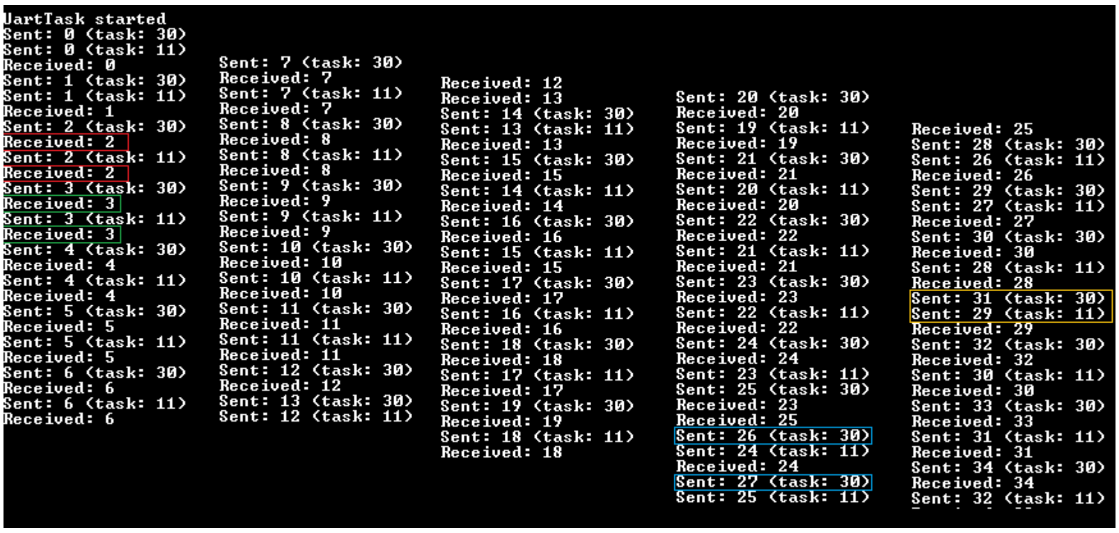
An example of a different approach is demonstrated in the FreeRTOS demos, where the *rclc_executor_spin_some* function is periodically called, giving the executor a specific amount of time to run before the task goes to sleep. The following code was used with PXROS-HR:

```
while(1)
{
    rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100));
    HtcSleep(PxTickGetTicksFromMilliSeconds(100), EV_WAIT_TIMEOUT);
}
```

The change in approach for running the executor helped in triggering both subscribers as shown in 4.19 with the red and green rectangles indicating activation of both subscribers. However, it can still be observed that the subscriber is not triggered consistently, as shown by the blue rectangles in 4.24 where the numbers 26 and 27 were published by the task with ID 30 but the corresponding subscriber was not triggered. Based on the observations, it seems that the handling of *rclc_executor_spin* might be the reason why the subscriber callbacks are not being triggered.

The difference in publishing speed between the two publishers is also worth noting. Even though they both wait for one second between publishing, after some time, one publisher becomes ahead of the other. This is because the implementation does not trigger a publisher every second to synchronize the publishing but instead triggers it one second after the previous publish. Furthermore, the amount of time needed for processing the publishing differs between the two publishers. The publisher that is defined as *default* needs to wait for acknowledgment, which causes its execution to take longer than the *best effort* publisher that does not use acknowledgments. This can be seen in 4.24 where in the yellow rectangle, both publishers have just published, but the difference in the numbers they published is greater than one.

To achieve synchronization and ensure reliable delivery of messages, the *best effort* publisher can be modified to become reliable. By doing this, the output is synchronized as both the publisher and subscriber take similar time to execute, and the subscribers are guaranteed to be triggered as the pub-sub pair is defined as reliable. The output can be seen in Figure 4.20.



```
UartTask started
Sent: 0 (task: 30)
Sent: 0 (task: 11)
Received: 0
Sent: 1 (task: 30)
Sent: 1 (task: 11)
Received: 1
Sent: 2 (task: 30)
Received: 2
Sent: 2 (task: 11)
Received: 2
Sent: 3 (task: 30)
Received: 3
Sent: 3 (task: 11)
Received: 3
Sent: 4 (task: 30)
Received: 4
Sent: 4 (task: 11)
Received: 4
Sent: 5 (task: 30)
Received: 5
Sent: 5 (task: 11)
Received: 5
Sent: 6 (task: 30)
Received: 6
Sent: 6 (task: 11)
Received: 6
```

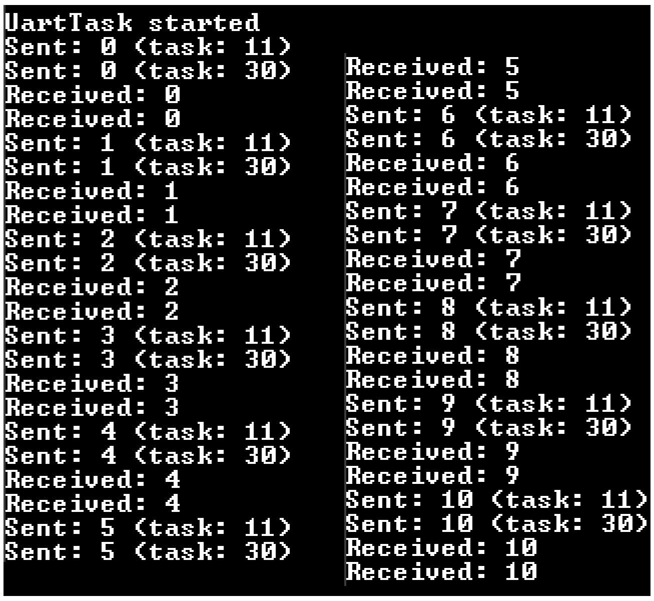
```
Sent: 7 (task: 30)
Received: 7
Sent: 7 (task: 11)
Received: 7
Sent: 8 (task: 30)
Received: 8
Sent: 8 (task: 11)
Received: 8
Sent: 9 (task: 30)
Received: 9
Sent: 9 (task: 11)
Received: 9
Sent: 10 (task: 30)
Received: 10
Sent: 10 (task: 11)
Received: 10
Sent: 11 (task: 30)
Received: 11
Sent: 11 (task: 11)
Received: 11
Sent: 12 (task: 30)
Received: 12
Sent: 13 (task: 30)
Sent: 12 (task: 11)
```

```
Received: 12
Received: 13
Sent: 14 (task: 30)
Sent: 13 (task: 11)
Received: 13
Sent: 15 (task: 30)
Received: 15
Sent: 14 (task: 11)
Received: 14
Sent: 16 (task: 30)
Received: 16
Sent: 15 (task: 11)
Received: 15
Sent: 17 (task: 30)
Received: 17
Sent: 16 (task: 11)
Received: 16
Sent: 18 (task: 30)
Received: 18
Sent: 17 (task: 11)
Received: 17
Sent: 19 (task: 30)
Received: 19
Sent: 18 (task: 11)
Received: 18
```

```
Sent: 20 (task: 30)
Received: 20
Sent: 19 (task: 11)
Received: 19
Sent: 21 (task: 30)
Received: 21
Sent: 20 (task: 11)
Received: 20
Sent: 22 (task: 30)
Received: 22
Sent: 21 (task: 11)
Received: 21
Sent: 23 (task: 30)
Received: 23
Sent: 22 (task: 11)
Received: 22
Sent: 24 (task: 30)
Received: 24
Sent: 23 (task: 11)
Sent: 25 (task: 30)
Received: 23
Received: 25
Sent: 26 (task: 30)
Sent: 24 (task: 11)
Received: 24
Sent: 27 (task: 30)
Sent: 25 (task: 11)
```

```
Received: 25
Sent: 28 (task: 30)
Sent: 26 (task: 11)
Received: 26
Sent: 29 (task: 30)
Sent: 27 (task: 11)
Received: 27
Sent: 30 (task: 30)
Received: 30
Sent: 28 (task: 11)
Received: 28
Sent: 31 (task: 30)
Sent: 29 (task: 11)
Received: 29
Sent: 32 (task: 30)
Received: 32
Sent: 30 (task: 11)
Received: 30
Sent: 33 (task: 30)
Received: 33
Sent: 31 (task: 11)
Received: 31
Sent: 34 (task: 30)
Received: 34
Sent: 32 (task: 11)
```

Figure 4.19 UART debug output of multithread demo with *rclc_executor_spin_some* function proving that it helps with the receiving of the *best effort* subscriber. However, it still misses some messages. In the yellow rectangle, it is possible to observe that the publishing time differs between *best effort* and *reliable* publishers.



```
UartTask started
Sent: 0 (task: 11)
Sent: 0 (task: 30)
Received: 0
Received: 0
Sent: 1 (task: 11)
Sent: 1 (task: 30)
Received: 1
Received: 1
Sent: 2 (task: 11)
Sent: 2 (task: 30)
Received: 2
Received: 2
Sent: 3 (task: 11)
Sent: 3 (task: 30)
Received: 3
Received: 3
Sent: 4 (task: 11)
Sent: 4 (task: 30)
Received: 4
Received: 4
Sent: 5 (task: 11)
Sent: 5 (task: 30)
```

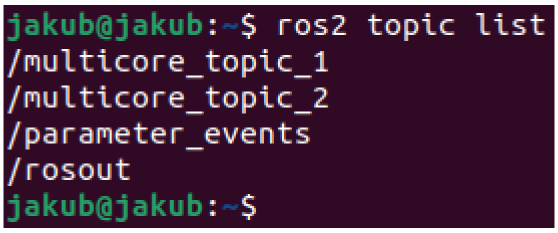
```
Received: 5
Received: 5
Sent: 6 (task: 11)
Sent: 6 (task: 30)
Received: 6
Received: 6
Sent: 7 (task: 11)
Sent: 7 (task: 30)
Received: 7
Received: 7
Sent: 8 (task: 11)
Sent: 8 (task: 30)
Received: 8
Received: 8
Sent: 9 (task: 11)
Sent: 9 (task: 30)
Received: 9
Received: 9
Sent: 10 (task: 11)
Sent: 10 (task: 30)
Received: 10
Received: 10
```

Figure 4.20 UART debug output of multithread demo with reliable Pub-Sub pairs.

4.6 Multicore_publisher_subscriber

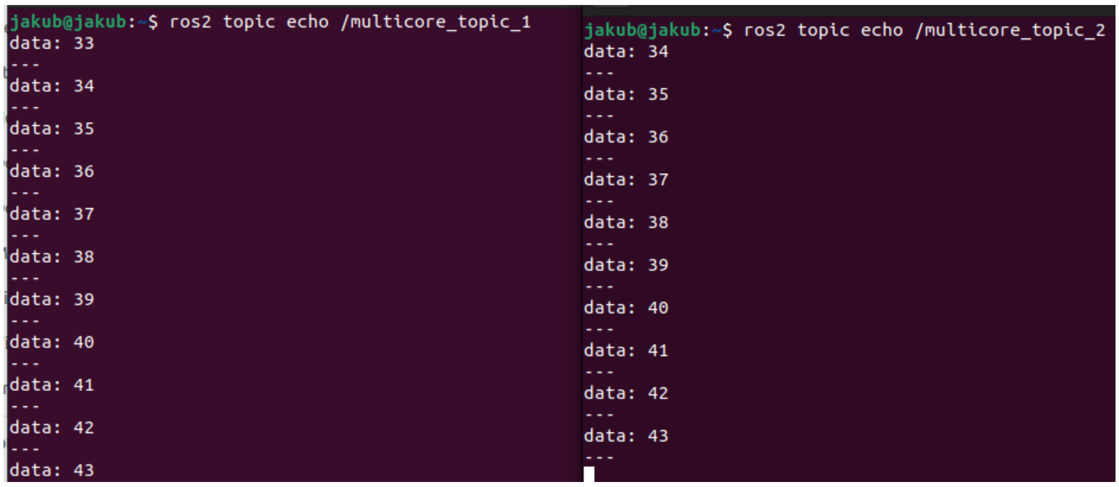
The multicore example builds upon the multithread example, with the main difference being that tasks are executed on separate cores to distribute the processing load.

The micro-ROS agent output remains the same as the ping_pong example, with two publishers, two subscribers, and two topics created. In figures 4.21 and 4.22, both multicore topics are available, with the publishers running on the MCU correctly publishing data to their respective topics.



```
jakub@jakub:~$ ros2 topic list
/multicore_topic_1
/multicore_topic_2
/parameter_events
/rosout
jakub@jakub:~$
```

Figure 4.21 Multicore topics registration check.



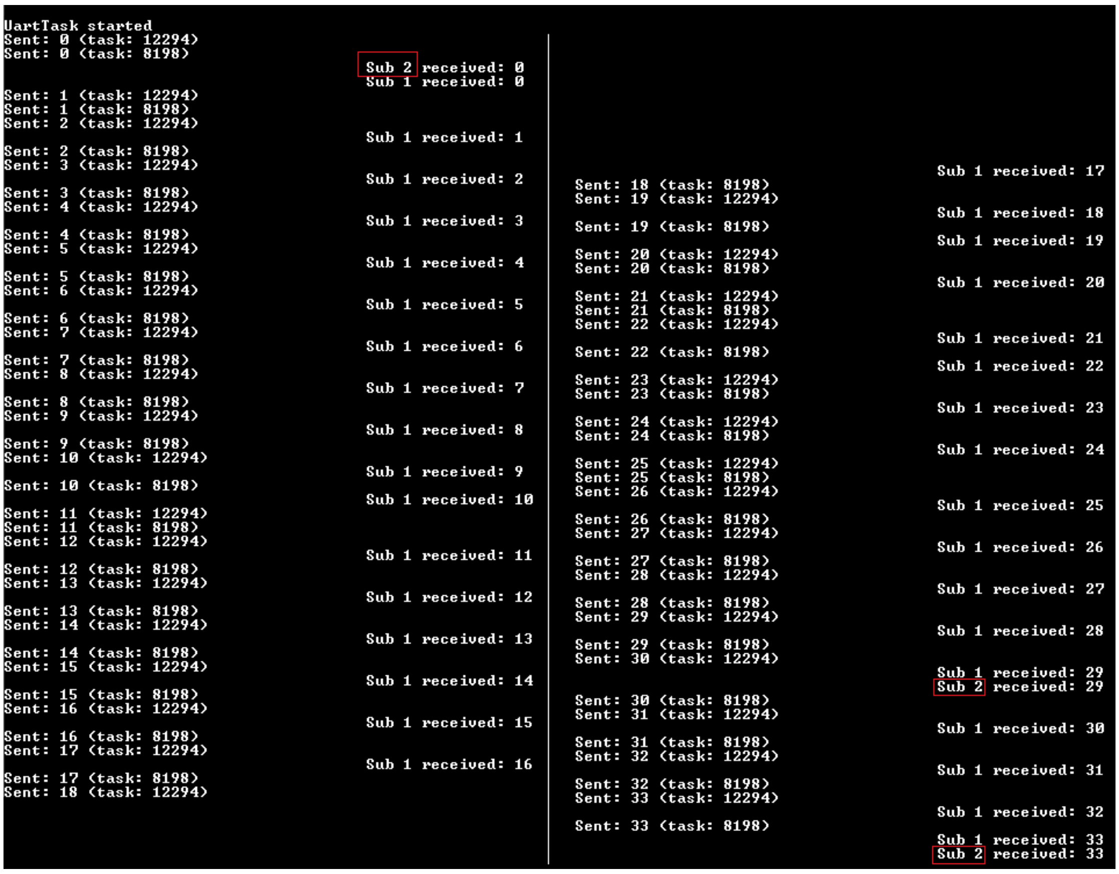
```
jakub@jakub:~$ ros2 topic echo /multicore_topic_1
data: 33
---
data: 34
---
data: 35
---
data: 36
---
data: 37
---
data: 38
---
data: 39
---
data: 40
---
data: 41
---
data: 42
---
data: 43
```

```
jakub@jakub:~$ ros2 topic echo /multicore_topic_2
data: 34
---
data: 35
---
data: 36
---
data: 37
---
data: 38
---
data: 39
---
data: 40
---
data: 41
---
data: 42
---
data: 43
---
```

Figure 4.22 Multicore topics echo showing that the application running on MCU publishes correctly to the two topics.

In the multicore example, both subscribers are running on the same core and encounter a similar issue as in the multithread example, where the use of the *rclc_executor_spin* function causes the *best effort* subscriber not to be triggered as shown in 4.23. However, changing the approach to periodic calls of *rclc_executor_spin_some* was sufficient to resolve the issue and ensure both subscribers are triggered correctly, as seen in 4.24. This suggests that distributing the execution load across different cores can improve the dependability of the example's execution.

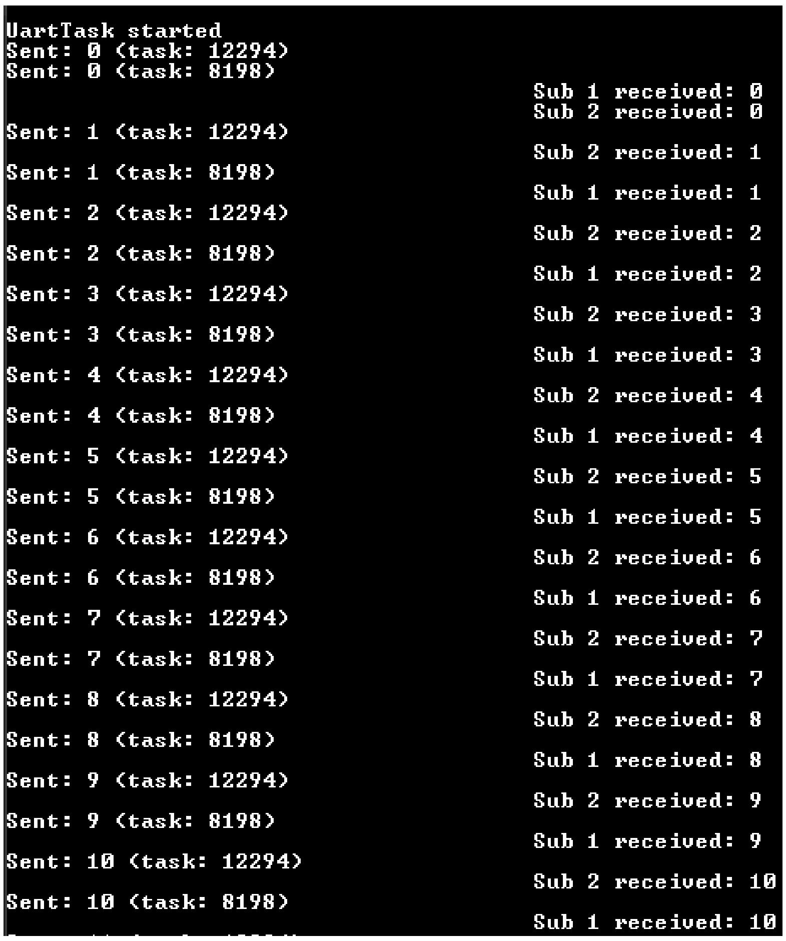
The execution time difference between the *default* and *best effort* publishing functions is significant enough to cause a noticeable discrepancy in the published numbers. After approximately 3 minutes, the *best effort* publisher is ten numbers ahead of the *default* publisher, as illustrated in 4.25.



```
UartTask started
Sent: 0 (task: 12294)
Sent: 0 (task: 8198)
                                    Sub 2 received: 0
                                    Sub 1 received: 0

Sent: 1 (task: 12294)
Sent: 1 (task: 8198)
Sent: 2 (task: 12294)
                                    Sub 1 received: 1
Sent: 2 (task: 8198)
Sent: 3 (task: 12294)
                                    Sub 1 received: 2
Sent: 3 (task: 8198)
Sent: 4 (task: 12294)
                                    Sub 1 received: 3
Sent: 4 (task: 8198)
Sent: 5 (task: 12294)
                                    Sub 1 received: 4
Sent: 5 (task: 8198)
Sent: 6 (task: 12294)
                                    Sub 1 received: 5
Sent: 6 (task: 8198)
Sent: 7 (task: 12294)
                                    Sub 1 received: 6
Sent: 7 (task: 8198)
Sent: 8 (task: 12294)
                                    Sub 1 received: 7
Sent: 8 (task: 8198)
Sent: 9 (task: 12294)
                                    Sub 1 received: 8
Sent: 9 (task: 8198)
Sent: 10 (task: 12294)
                                    Sub 1 received: 9
Sent: 10 (task: 8198)
                                    Sub 1 received: 10
Sent: 11 (task: 12294)
Sent: 11 (task: 8198)
Sent: 12 (task: 12294)
                                    Sub 1 received: 11
Sent: 12 (task: 8198)
Sent: 13 (task: 12294)
                                    Sub 1 received: 12
Sent: 13 (task: 8198)
Sent: 14 (task: 12294)
                                    Sub 1 received: 13
Sent: 14 (task: 8198)
Sent: 15 (task: 12294)
                                    Sub 1 received: 14
Sent: 15 (task: 8198)
Sent: 16 (task: 12294)
                                    Sub 1 received: 15
Sent: 16 (task: 8198)
Sent: 17 (task: 12294)
                                    Sub 1 received: 16
Sent: 17 (task: 8198)
Sent: 18 (task: 12294)
```

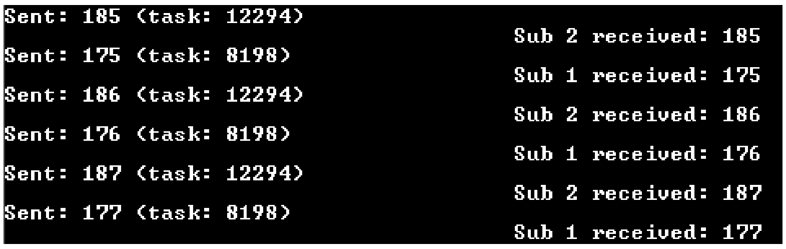
```
                                    Sub 1 received: 17
Sent: 18 (task: 8198)
Sent: 19 (task: 12294)
                                    Sub 1 received: 18
Sent: 19 (task: 8198)
                                    Sub 1 received: 19
Sent: 20 (task: 12294)
Sent: 20 (task: 8198)
                                    Sub 1 received: 20
Sent: 21 (task: 12294)
Sent: 21 (task: 8198)
Sent: 22 (task: 12294)
                                    Sub 1 received: 21
Sent: 22 (task: 8198)
                                    Sub 1 received: 22
Sent: 23 (task: 12294)
Sent: 23 (task: 8198)
                                    Sub 1 received: 23
Sent: 24 (task: 12294)
Sent: 24 (task: 8198)
                                    Sub 1 received: 24
Sent: 25 (task: 12294)
Sent: 25 (task: 8198)
Sent: 26 (task: 12294)
                                    Sub 1 received: 25
Sent: 26 (task: 8198)
Sent: 27 (task: 12294)
                                    Sub 1 received: 26
Sent: 27 (task: 8198)
Sent: 28 (task: 12294)
                                    Sub 1 received: 27
Sent: 28 (task: 8198)
Sent: 29 (task: 12294)
                                    Sub 1 received: 28
Sent: 29 (task: 8198)
Sent: 30 (task: 12294)
                                    Sub 1 received: 29
                                    Sub 2 received: 29
Sent: 30 (task: 8198)
Sent: 31 (task: 12294)
                                    Sub 1 received: 30
Sent: 31 (task: 8198)
Sent: 32 (task: 12294)
                                    Sub 1 received: 31
Sent: 32 (task: 8198)
Sent: 33 (task: 12294)
                                    Sub 1 received: 32
Sent: 33 (task: 8198)
                                    Sub 1 received: 33
                                    Sub 2 received: 33
```

Figure 4.23 UART debug output of *multicore_publisher_subscriber* demo where the *best effort* subscriber does not work properly. The red rectangles show when the *best effort* subscribers were triggered.



```
UartTask started
Sent: 0 (task: 12294)
Sent: 0 (task: 8198)
                                       Sub 1 received: 0
                                       Sub 2 received: 0

Sent: 1 (task: 12294)
                                       Sub 2 received: 1
Sent: 1 (task: 8198)
                                       Sub 1 received: 1
Sent: 2 (task: 12294)
                                       Sub 2 received: 2
Sent: 2 (task: 8198)
                                       Sub 1 received: 2
Sent: 3 (task: 12294)
                                       Sub 2 received: 3
Sent: 3 (task: 8198)
                                       Sub 1 received: 3
Sent: 4 (task: 12294)
                                       Sub 2 received: 4
Sent: 4 (task: 8198)
                                       Sub 1 received: 4
Sent: 5 (task: 12294)
                                       Sub 2 received: 5
Sent: 5 (task: 8198)
                                       Sub 1 received: 5
Sent: 6 (task: 12294)
                                       Sub 2 received: 6
Sent: 6 (task: 8198)
                                       Sub 1 received: 6
Sent: 7 (task: 12294)
                                       Sub 2 received: 7
Sent: 7 (task: 8198)
                                       Sub 1 received: 7
Sent: 8 (task: 12294)
                                       Sub 2 received: 8
Sent: 8 (task: 8198)
                                       Sub 1 received: 8
Sent: 9 (task: 12294)
                                       Sub 2 received: 9
Sent: 9 (task: 8198)
                                       Sub 1 received: 9
Sent: 10 (task: 12294)
                                       Sub 2 received: 10
Sent: 10 (task: 8198)
                                       Sub 1 received: 10
```

Figure 4.24 UART debug output of multicore demo with *rclc_executor_spin_some* function proving that it helps with the receiving of the *best effort* subscriber.



```
Sent: 185 (task: 12294)
                                     Sub 2 received: 185
Sent: 175 (task: 8198)
                                     Sub 1 received: 175
Sent: 186 (task: 12294)
                                     Sub 2 received: 186
Sent: 176 (task: 8198)
                                     Sub 1 received: 176
Sent: 187 (task: 12294)
                                     Sub 2 received: 187
Sent: 177 (task: 8198)
                                     Sub 1 received: 177
```

Figure 4.25 Time difference in the publishing functions of the *best effort* and *reliable* publishers. After approximately three minutes, the difference is about ten seconds.

Chapter 5
Conclusion

In conclusion, this thesis has provided an analysis of Robot Operating System (ROS) and its successor, ROS 2, as well as the Micro-ROS framework, Data Distribution Service (DDS), and PXROS-HR. For a Real-Time Operating System (RTOS) to be compatible with micro-ROS, it needs to be POSIX compliant. For RTOSes that are not POSIX compliant, they need to align with the POSIX standard to some degree. In the case of PXROS-HR, this meant implementing time functions and mutexes to achieve compatibility with micro-ROS. The PXROS-HR features were introduced and used to map the requirements for time functions and mutex functionality. These requirements were then implemented using the relevant PXROS-HR features and API functions. The proposed solution and implementation involve building a custom static library for micro-ROS, implementing Mutex Server, configuring the project structure, and testing different micro-ROS demos. The implementation was done at the application level, so there was no need to modify the kernel itself. The results showed that the prototype successfully integrates PXROS-HR into Micro-ROS using the UDP communication protocol for data exchange between the Micro-ROS agent and the client.

The necessary components required to build this project are as follows:

- HighTec compiler for TriCore version 4.9.4.1
- HighTec safety certified RTOS for TriCore – PXROS-HR
- PxNet – HighTec module encapsulating TCP/IP stack developed by SEVENSTAX company
- SEVENSTAX TCP/IP stack sources
- Micro-ROS library
- HighTec PxNet base example
- iLLD (Infineon Low-Level Driver) sources to build the Ethernet driver

The above-mentioned list highlights the complexity of the prototype, which necessitated the use of various commercial tools, libraries, and software.

The development of this thesis was conducted in collaboration with HighTec, and any code generated to investigate the feasibility of integrating PXROS-HR into micro-ROS is the property of HighTec.

To enhance the readability of the text, captions were omitted in some of the code listings.

Known Limitations

The *stdbool.h* header from the standard library is utilized by the Micro-ROS library for boolean data types. However, HighTec defines its own boolean type and overwrites the previous definition in *pxdef.h*, which contains all the public definitions, API, and structures for PXROS-HR. This becomes problematic because the boolean type defined in the standard library has a size of 1 byte while the boolean type in PXROS-HR is defined as *PxUInt32_t* with a size of 4 bytes. Although *pxdef.h* is included in several places, the issue arises when the PXROS-HR mutex implementation in the Micro XRCE-DDS Client project includes *MutexServer.h*, which requires the PXROS-HR API for its implementation. As a result, the size of function parameters (or return values) in function declarations and definitions did not match. To address this problem, the redefinition in *pxdef.h* was commented out during the build of the Micro-ROS library to prevent any interference with code relying on the standard library. The original *pxdef.h* was used without modification during the build of the prototype.

During testing, it was observed that the micro-ROS application failed to start and encountered errors in some of the PxNet API functions. This issue is likely due to a lack of synchronization in the PxNet module, which means that there is no exact point where a task can access information that the PxNet module is fully initialized and ready for use. Although waiting for 10 seconds appeared to solve the problem during experimentation, later testing showed that this simple waiting time was not sufficient. This issue indicates the possibility of a bug in the PxNet module, but detailed testing and root cause investigation are beyond the scope of this thesis. However, it is important to note that this issue did not affect the feasibility of the integration.

As the integration of PXROS-HR into micro-ROS was not straightforward and PXROS-HR does not provide mutexes by default, the development of the Mutex Server was necessary. The implementation of this task was tested internally at HighTec. However, like any other use of mutexes, this approach introduces some overhead. Although this is not within the scope of this thesis, it would be valuable to evaluate the size of the overhead and compare it with standard POSIX mutexes on Linux, for example.

The use of PxNet requires an aligned buffer, which can also cause overhead in micro-ROS. This is because the buffer used in micro-ROS is incrementally filled, and may not always be aligned. As a result, message copying is necessary both during sending and receiving functions to ensure alignment with PxNet requirements.

Results

A functional prototype was developed to demonstrate the feasibility of integrating PXROS-HR into the micro-ROS system, with a focus on the key concept of micro-ROS communication patterns. The prototype includes several micro-ROS demo applications that were adapted to use PXROS-HR as their underlying real-time operating system. Additionally, an extra demo was created to showcase the power of PXROS-HR as a multicore RTOS, with multiple tasks assigned to different cores. Overall, the prototype proves that the integration of PXROS-HR into micro-ROS is possible and offers benefits in terms of real-time capabilities and multicore processing.

The prototype is currently designed for the Application Kit TC3X7 version 2.0 with the TC397 B-Step, but it can be easily adapted for other boards and processors supported by the PxNet base example. Only the general files that are not directly related to micro-ROS need to be added or modified.

Appendix A

History of ROS

ROS originated as a personal project initiated by Keenan Wyrobek and Eric Berger while studying at Stanford University aiming to address the recurring issue of reinventing the wheel in robotics (comic of the robotics research process is shown in A.1). Keenan Wyrobek and Eric Berger were concerned about the prevalent issue in robotics at the time, meaning mainly:

- Too much time was spent re-creating the software infrastructure to develop complex robotics algorithms (this included creating drivers for sensors and actuators, as well as enabling communication between different programs within the same robot)
- Lack of time devoted to building intelligent robotics programs

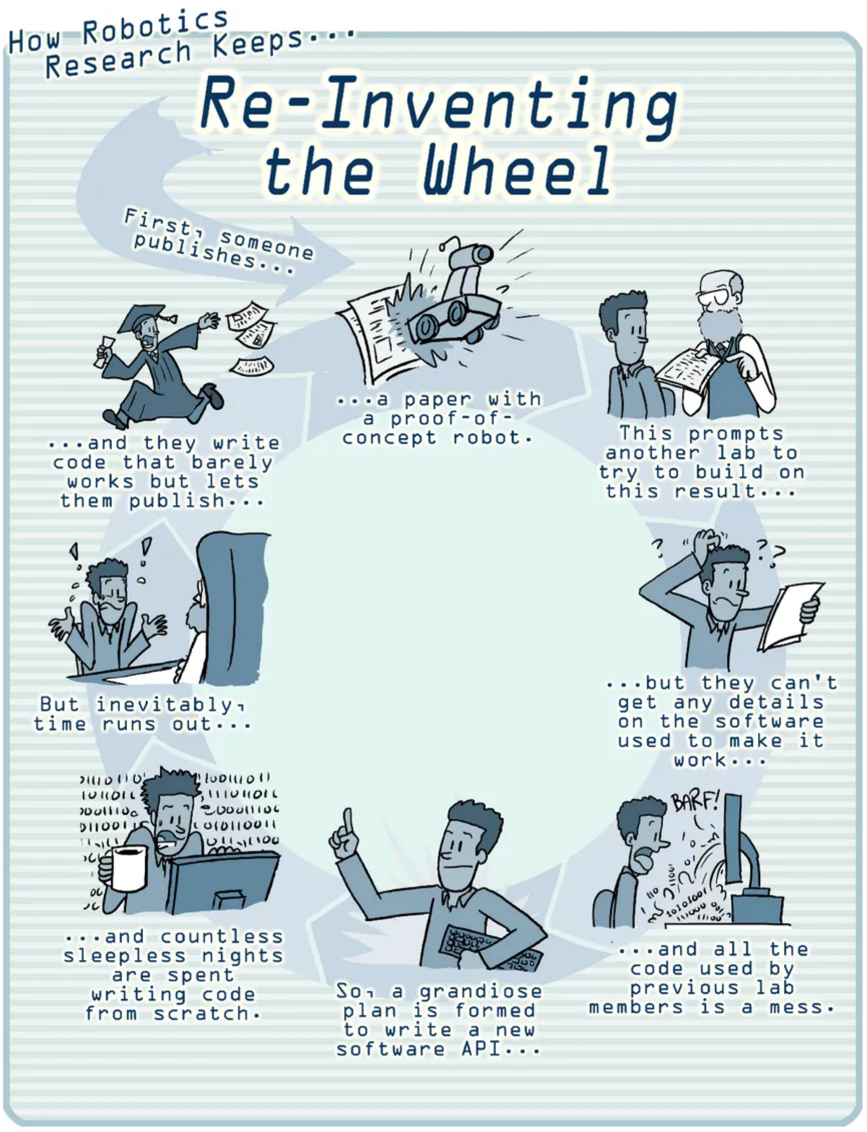
Keenan and Eric highlighted the issue of the reinvention of drivers and communication systems being repeated for every new project, even within the same organization. They beautifully expressed the situation in one of their pitch deck[1] slides that they used to present to potential investors. Figure A.2 shows these ratios between reimplementing and bringing new solutions.

Eric and Keenan established the Stanford Personal Robotics Program in 2006 with the objective of addressing the issue of reinventing the wheel in robotics. They aimed to develop a framework that enables seamless communication among processes, along with tools to facilitate code creation. The goal was to raise a total of $4 million, which was estimated to cover the expenses of hiring software engineers for ROS support and the construction of ten robot copies. Eventually, Keenan and Eric received only $50k in funding. The funds, along with additional financial support from Stanford Deans, were used to develop PR1 (Personal Robot 1 – visible in A.3). This robot was then used to garner support for the project from leading robotics software R&D teams, as well as the Stanford AI Robot team, which provided valuable insights into the high standards required for a robotics software development platform to be truly effective. However, perhaps the most significant use of PR1 was in creating engaging videos[2] that showcased its capabilities, which were then used to attract further funding for the project.

Despite receiving feedback that their goal of creating the ”Linux of Robotics” was too ambitious, the creators of ROS persisted in pursuing this vision. Their persistence paid off when they met Scott Hassan, an investor and the founder of Willow Garage, a research center specializing in robotics products, who was impressed by their vision and saw its potential to enable future entrepreneurs in the robotics industry with an open-source foundation. As a successful entrepreneur himself who had used open-source software to build companies like Google and

[1]Pitch decks in a simplified way are: ”Short presentations to help someone else learn about your business quickly.” Taken from [56]

[2]One of the videos is about Teleoperated PR1 cleaning a room in 2006, which is accessible at https://www.youtube.com/watch?v=oyHWkQcin7I.


How Robotics Research Keeps...
Re-Inventing the Wheel
First, someone publishes...
...a paper with a proof-of-concept robot.
This prompts another lab to try to build on this result...
...but they can't get any details on the software used to make it work...
BARF!
...and all the code used by previous lab members is a mess.
So, a grandiose plan is formed to write a new software API...
...and countless sleepless nights are spent writing code from scratch.
But inevitably, time runs out...
...and they write code that barely works but lets them publish...

■ **Figure A.1** Comic commissioned at Willow Garage, from Jorge Cham, to illustrate the wasted time in robotics R&D. [55]

eGroups, Hassan was passionate about ”paying it forward” and supporting the development of ROS.

Scott was so intrigued by their idea that he chose to provide funding and establish a Personal Robotics Program with Keenan and Eric at Willow Garage, which was the third program alongside autonomous car and boat programs. Thanks to Hassan’s support, the development of ROS at Willow Garage received a funding that exceeded the initial estimate of $4 million. This is where the Robot Operating System was created, along with the PR2 (Personal Robot 2) robot. The ROS project soon became so significant that Willow Garage abandoned all of its other projects and solely focused on the development and dissemination of ROS.

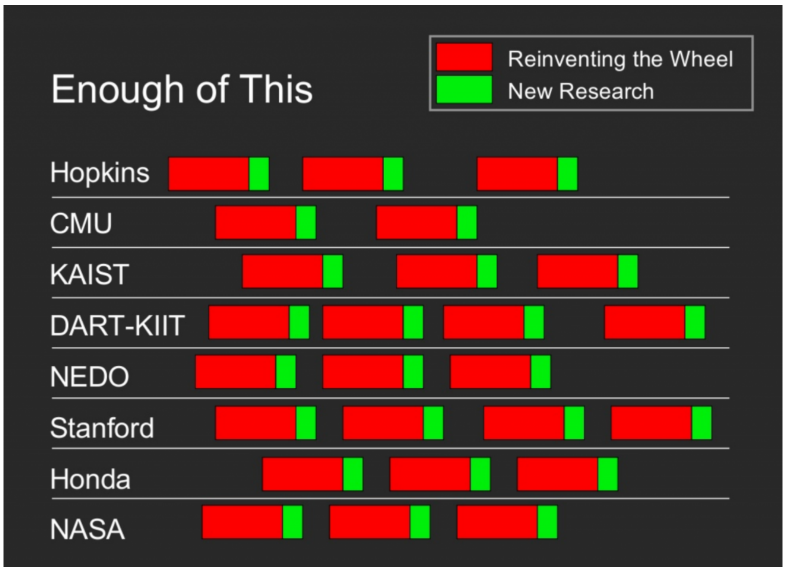


Figure A.2 The Eric and Keenan pitch deck slide shows the ratio of time spent between reinventing the wheel and new research in robotics. It points out that most of the time is wasted by re-implementing already existing software. The image also indicates that this issue also happened in the same organizations. [55]

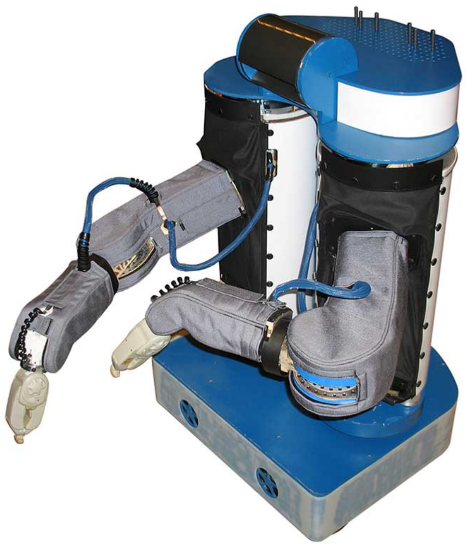


Figure A.3 PR1 (Personal Robot 1) was utilized to garner support from the foremost research and development teams in the field of robotics software. [57]

A.1 Getting to ROS 1.0

At Willow Garage, a multitude of factors contributed to the eventual success of the Robot Operating System (ROS) in becoming the "Linux of Robotics." One key factor was the early recruitment of renowned leaders, engineers, and researchers. These individuals brought valuable expertise and experience to the team. Another critical aspect was the team's unwavering focus on building ROS as the "Linux of Robotics." They achieved this by incorporating leaders from previous open-source robotics initiatives into the ROS community, investing in user-friendly and

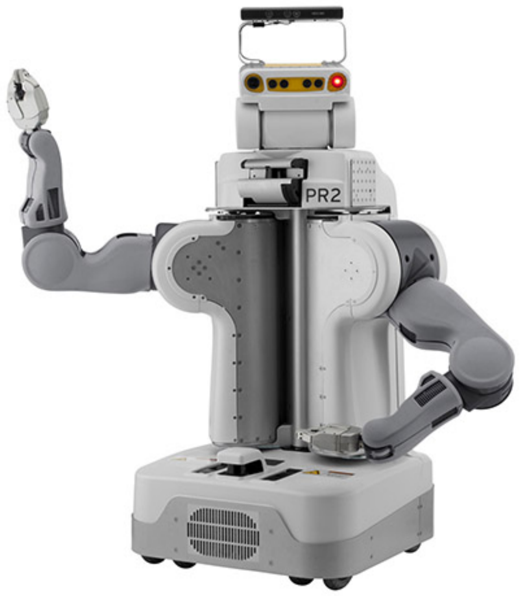


Figure A.4 The PR2 (Personal Robot 2) robot is considered to be among the most advanced research robots ever developed. Its cutting-edge hardware and software capabilities enable it to perform complex tasks. PR2 robot is entirely integrated with ROS, providing access to all the ROS developer tools and built-in capabilities for tasks ranging from complete system calibration to manipulation. [58]

powerful software, and persuading major companies like Bosch to adopt open-source libraries.

To foster the growth of the ROS community, the team employed various tactics (e.g. The two-day workshop, The intern program). For example, they encouraged applicants for a free PR2 through their Beta Program to describe how they would use the hardware platform to benefit the entire ROS community. In addition, they incentivized leaders at academic institutions and companies to open-source their robotics work by awarding them prizes, rather than offering traditional academic discounts for the PR2s.

A.2 Willow Garage Time

ROS was developed at Willow Garage for approximately six years, until the company shut down in 2014. During this time, the project underwent many advancements which contributed to its rise in popularity.

In 2009, ROS Mango Tango, also known as ROS 0.4, was released as the first distribution of ROS. Interestingly, its name had no connection to the current naming convention. The release of ROS 1.0 followed almost a year later in 2010. From this point on, the ROS team adopted a naming convention that used turtle species for their distributions. The subsequent distributions and their release dates are as follows in table A.1.

During that time, several other significant events occurred. In 2009, Willow Garage developed the second version of the Personal Robot, known as the PR2. In an effort to address technical queries about ROS, the team launched ROS Answers. The first ROSCON was organized in 2012 and has since become the official yearly conference for ROS developers. Willow Garage provided

Distribution	Release date
Box Turtle	2010
ROS C-Turtle	2010
Diamond Back	2011
ROS Electric Emys	2011
ROS Fuerte Turtle	2012
ROS Groovy Galapagos	2012

Table A.1 First ROS distributions with release dates.

11 PR2 robots to 11 universities in 2010 to facilitate robotics software development using ROS. This move was in line with the original idea of Eric and Keenan.

Moreover, during this period, simulation became crucial, specifically 3D simulation. Consequently, the team incorporated Gazebo, the 3D robotics simulator from the Player/Stage project, into ROS. As a result, Gazebo became the default 3D simulator for ROS. The growth of ROS was remarkable as it progressed. The number of repositories, packages offered, and the adoption rate by universities and corporations increased rapidly. This massive increase can be seen in A.5, showing the number of repositories and packages in the early days.

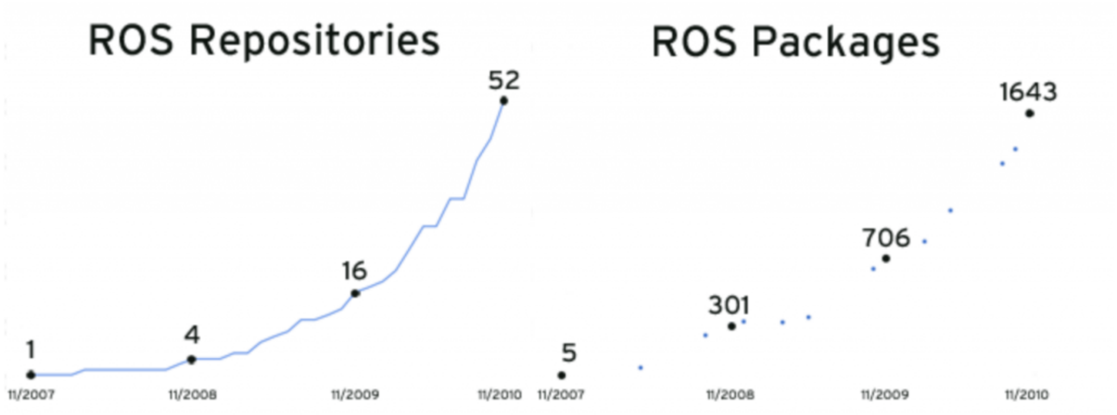


Figure A.5 In the early days, the development of ROS was evolving and brought a massive increase in the number of repositories and packages offered. [58]

Another crucial event that contributed to the expansion of the ROS community was the introduction of the Turtlebot robot by Willow Garage in 2011, which became a widely recognized robot for ROS developers. While the PR2 was originally intended for testing and development with ROS, its complexity and high cost made it unfeasible for most researchers. In contrast, the TurtleBot was an affordable and simple robot that allowed for basic experimentation with ROS and robotics. It quickly gained widespread popularity and is still in use today, with updated versions such as the TurtleBot2, TurtleBot3, and the most current TurtleBot4, released in May 2022. The whole TurtleBot family is shown in A.6.

In 2013, an announcement was made by Willow Garage that the company would be dissolved within that year, which sparked questions about the fate of ROS. It was proposed that the Open Source Robotics Foundation, which was newly established at the time, would take over the leadership of ROS development. Additionally, many of the former Willow Garage employees were hired by Suitable Technologies, one of the spin-offs created from Willow Garage, despite not using ROS for their products. The responsibility of providing customer support for all the

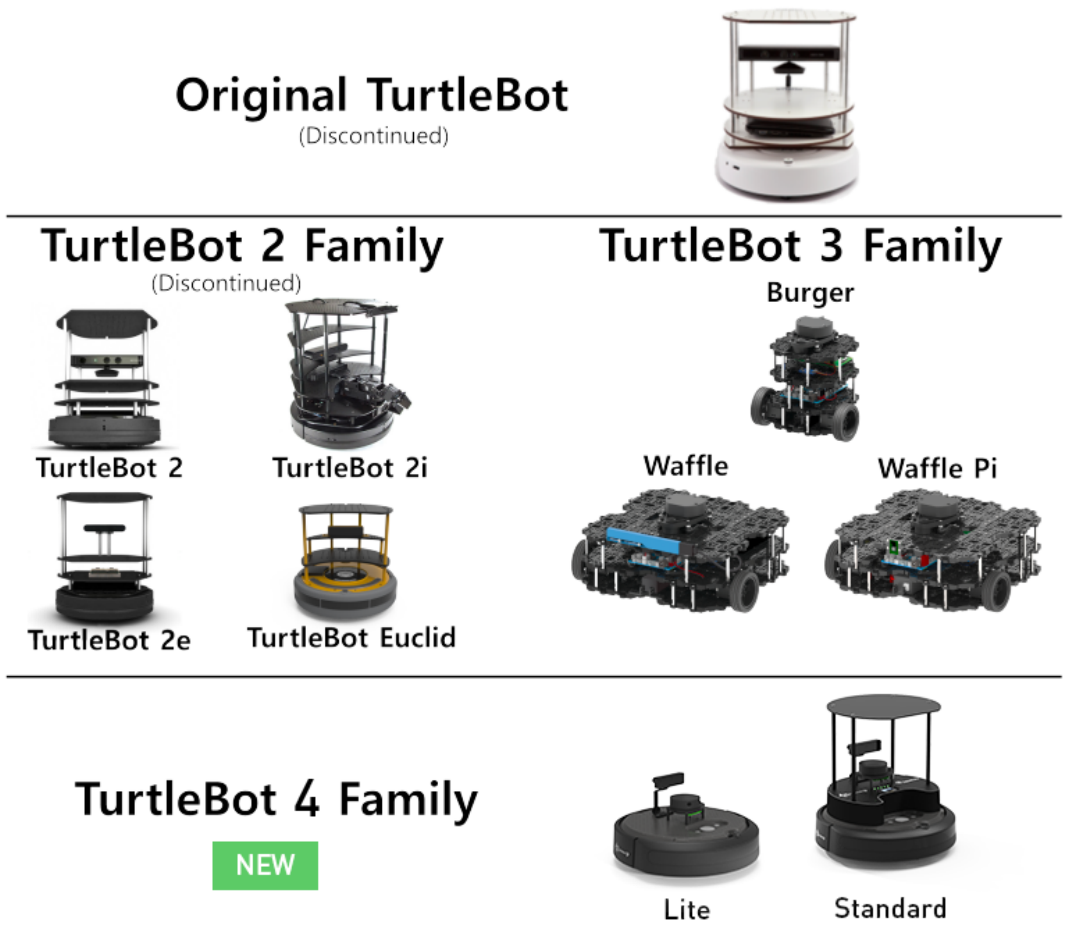


Figure A.6 The TurtleBot is an open-source personal robot kit developed by Melonee Wise and Tully Foote at Willow Garage in November 2010. It was an inexpensive and straightforward robot facilitating elementary ROS and robotics experimentation. There are now four generations of TurtleBots. The image shows the TurtleBot Family. [59]

PR2 robots was taken up by Clearpath Robotics, another significant company in the robotics industry.

A.3 Open Source Robotics Foundation Time

ROS kept progressing and introducing new versions under the fresh legal structure of the Open Source Robotics Foundation. Table A.2 lists the following distributions with their release date.

At the time of writing this thesis, there are only two active ROS 1 distributions – ROS Melodic Morenia and ROS Noetic Ninjemys. It should be noted that the latest ROS distribution, ROS Noetic, is the final version ever released. It uses Python 3, unlike all previous versions that used Python 2. No additional ROS 1 distributions were made after this release, and the focus was shifted to developing ROS 2.

Distribution	Release date
ROS Hydro Medusa	2013
ROS Indigo Igloo	2014
ROS Jade Turtle	2015
ROS Kinetic Kame	2016
ROS Lunar Loggerhead	2017
ROS Melodic Morenia	2018
ROS Noetic Ninjemys	2020

Table A.2 Following ROS distributions created under Open Source Robotics Foundation with release dates.

A.4 ROS 2.0, The New Generation

In approximately 2015, the limitations of ROS for commercial use became more evident. Companies were hesitant to adopt ROS in their products due to issues such as a single point of failure (the roscore), lack of security, and no support for real-time applications. However, ROS needed a stronger presence in the industrial sector to become the standard for robotics. To address this challenge, the Open Source Robotics Foundation undertook the development of ROS 2.0. This foundation underwent a name change in 2017 to become Open Robotics. The rebranding was done to give the organization a more corporate identity rather than being perceived as a foundation, even though the foundation aspect of the organization still remains. More details about ROS 2 will be provided in the following sections. [58, 55]

Appendix B

Mutex Implementation

This appendix provides additional information on the mutex implementation, including the mutex API functions that application tasks can use, as well as visual representations to aid in understanding the Mutex Server implementation.

B.1 Mutex API Functions

The following functions provide the mutex API that should be called from the application tasks. It covers three basic functionalities – initialization, locking, and unlocking of the mutex.

```c
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * FUNCTION: PxInitLock
 *     Initialize the mutex structure with valid mutex ID on success, otherwise
 *     error code
 * IN:
 *     mutex     : pointer to a mutex structure
 * NOTES:
 * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

void PxInitLock(PxMutex_t* mutex)
{
  /* Get Mutex server mailbox */
  PxError_t errRes = HtcPxNameQuery(MutexServer_MID_NAMESERVERID,
                                    sizeof(PxTask_t),
                                    &mutex->mutex_server_mbx,
                                    0,
                                    0,
                                    MUTEXSERVER_EV_WAIT_TIMEOUT);
  if (errRes != PXERR_NOERROR)
    PxPanic();

  /* Get message to communicate with Mutex server, metadata are sufficient */
  PxMsg_t msgHnd = PxMsgRequest (0, PXMcTaskdefault, PXOpoolTaskdefault);

  /* Set message to be send back */
  PxMsgSetToAwaitRel(msgHnd);

  /* Get message metadata */
  PxMutexMsg_t mutex_msg_metadata;
  mutex_msg_metadata.pxmd = PxMsgGetMetadata(msgHnd);

  /* Set metadata for mutex init */
  mutex_msg_metadata.itc.mutex_operation = MUTEX_INIT;
  mutex->mutex_id = -1; // uninitialized
```

```
  mutex_msg_metadata.itc.mutex_id = mutex->mutex_id;
  PxMsgSetMetadata(msgHnd, mutex_msg_metadata.pxmd);

  /* Send message to Mutex server, blocking call with await release */
  PxMsgSend(msgHnd, mutex->mutex_server_mbx);
  PxMsgAwaitRel(msgHnd); // wait for response

  /* Set mutex ID */
  mutex_msg_metadata.pxmd = PxMsgGetMetadata(msgHnd);
  mutex->mutex_id = mutex_msg_metadata.itc.mutex_id;

  PxMsgRelease(msgHnd);
}
```

Code listing B.1 Mutex initialization function in *MutexServer.c*.

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * FUNCTION: PxLock
 *     Get the lock of the mutex if it is free
 *     otherwise wait for the mutex to be free and then get the lock.
 * IN:
 *     mutex     : pointer to a mutex structure
 * NOTES:
 *      When the message is released by the Mutex server, lock was successfully
 *      obtained
 * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

void PxLock(PxMutex_t* mutex)
{
  /* Get message to communicate with Mutex server, metadata are sufficient */
  PxMsg_t msgHnd = PxMsgRequest (0, PXMcTaskdefault, PXOpoolTaskdefault);

  /* Set message to be send back */
  PxMsgSetToAwaitRel(msgHnd);

  /* Get message metadata */
  PxMutexMsg_t mutex_msg_metadata;
  mutex_msg_metadata.pxmd  = PxMsgGetMetadata(msgHnd);

  /* Set metadata for mutex init */
  mutex_msg_metadata.itc.mutex_operation = MUTEX_LOCK;
  mutex_msg_metadata.itc.mutex_id = mutex->mutex_id;
  PxMsgSetMetadata(msgHnd, mutex_msg_metadata.pxmd);

  /* Send message to Mutex server, blocking call with await release */
  PxMsgSend(msgHnd, mutex->mutex_server_mbx);
  PxMsgAwaitRel(msgHnd); // wait for response

  PxMsgRelease(msgHnd);

  /* Successfully claimed the lock */
}
```

Code listing B.2 Mutex lock function in *MutexServer.c*.

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * FUNCTION: PxUnlock
 *     If the requesting task has the lock:
 *        If there is any task waiting for the lock, pass it the lock; if no
   task
 *        waits, unlock it
 *     otherwise:
 *        do nothing
 * IN:
 *     mutex     : pointer to a mutex structure
```

```
 * NOTES:
 *     Unlocking always passes the lock to the first task waiting for the lock,
 *     if there is any.
 * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

void PxUnlock(PxMutex_t* mutex)
{
  /* Get message to communicate with Mutex server, metadata are sufficient */
  PxMsg_t msgHnd = PxMsgRequest (0, PXMcTaskdefault, PXOpoolTaskdefault);

  /* Set message to be send back */
  PxMsgSetToAwaitRel(msgHnd);

  /* Get message metadata */
  PxMutexMsg_t mutex_msg_metadata;
  mutex_msg_metadata.pxmd  = PxMsgGetMetadata(msgHnd);

  /* Set metadata for mutex init */
  mutex_msg_metadata.itc.mutex_operation = MUTEX_UNLOCK;
  mutex_msg_metadata.itc.mutex_id = mutex->mutex_id;
  PxMsgSetMetadata(msgHnd, mutex_msg_metadata.pxmd);

  /* Send message to Mutex server, blocking call with await release */
  PxMsgSend(msgHnd, mutex->mutex_server_mbx);
  PxMsgAwaitRel(msgHnd); // wait for response

  PxMsgRelease(msgHnd);

  /* Successfully unlocked */
}
```

Code listing B.3 Mutex unlock function in *MutexServer.c*.

B.2 Mutex Scheme

This section provides detailed visualizations of the implementation of Mutex Server. Figure B.1 depicts the state after initialization and provides an overview of how communication between application tasks and Mutex Server works, as well as how the internal structure of the mutexes is organized. Figure B.2 demonstrates the scenario where TaskA already has the lock, and TaskB tries to lock the same mutex. TaskB's request goes to the mutex mailbox (FIFO) and TaskB is blocked until the mutex is unlocked.

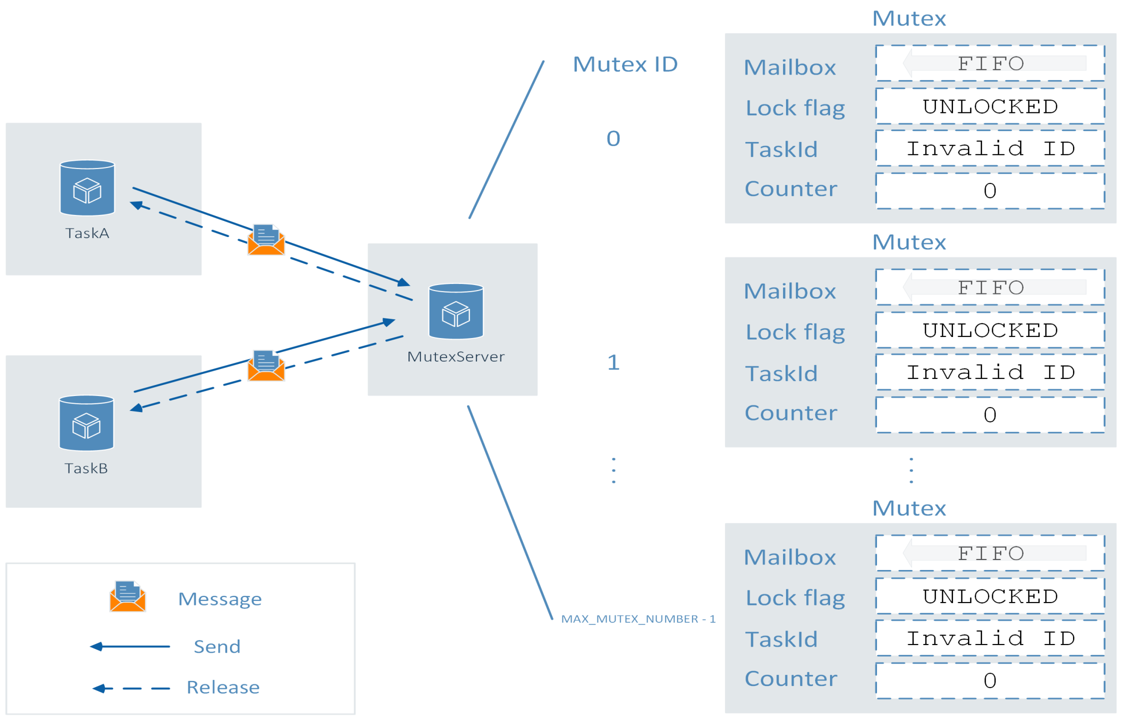


Figure B.1 Example scheme after initialization with two application tasks – TaskA, TaskB.

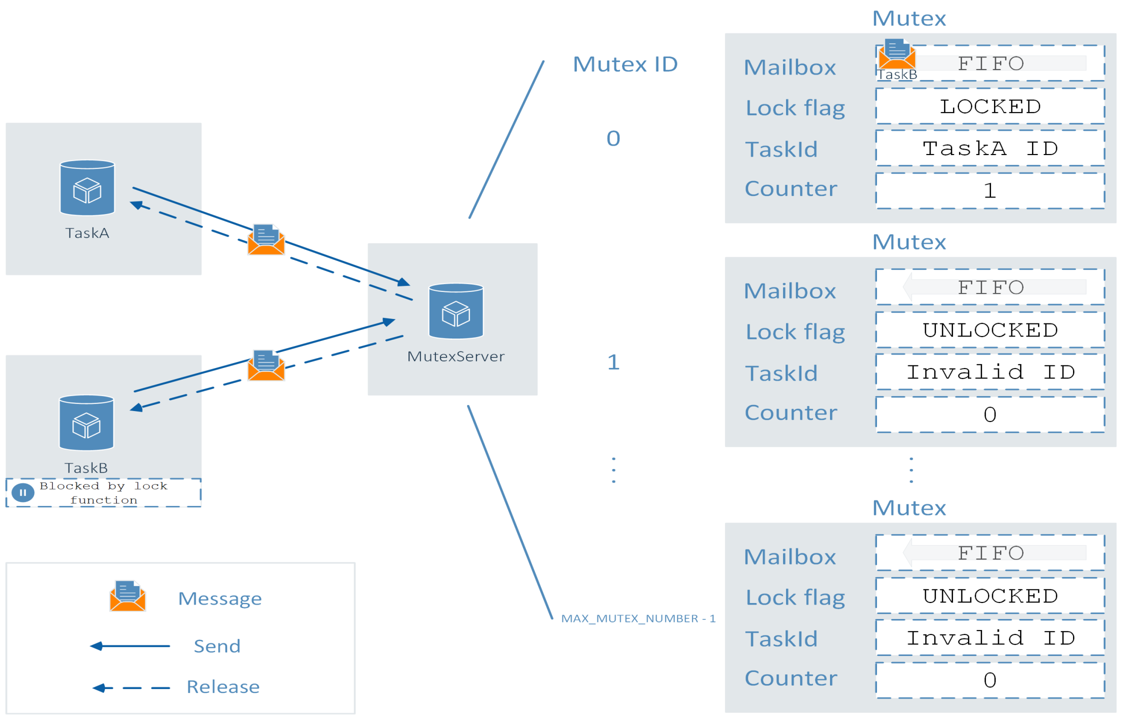


Figure B.2 Example scheme where TaskA has the lock, and TaskB sends the request to lock the same mutex. TaskB's request goes to the mutex mailbox, and TaskB waits for the release of the request message (obtaining the lock).

Appendix C
Custom Allocators Implementation

The TriCore microcontrollers come with a hardware Memory Protection Unit (MPU) and PXROS-HR handles the correct configuration of this MPU. Each task is required to specify its access rights to the memory areas it needs to work with. To ensure proper access rights are set, custom allocators are utilized for dynamic memory allocation. The following implementations were employed for the single-core demos:

```
void * PXROS_allocate(size_t size, void * state){
  (void) state;

  return PxMcTakeBlk(PXMcTaskdefault, PXMEM_ADJUST(size));
}


void PXROS_deallocate(void * pointer, void * state){
  (void) state;

  if (pointer)
    (void) PxMcReturnBlk(PXMcTaskdefault, (PxMptr_t) pointer);
}


void * PXROS_reallocate(void * pointer, size_t size, void * state){
  (void) state;

  if (NULL == pointer)
  {
    return PxMcTakeBlk(PXMcTaskdefault, PXMEM_ADJUST(size));
  }
  else
  {
    PxMptr_t tmp = PxMcTakeBlk(PXMcTaskdefault, PXMEM_ADJUST(size));
    if (tmp)
    {
      memcpy(tmp, pointer, size);
      (void) PxMcReturnBlk(PXMcTaskdefault, (PxMptr_t) pointer);
    }
    return tmp;
  }
}



```

```
void * PXROS_zero_allocate(size_t number_of_elements, size_t size_of_element,
    void * state){
  (void) state;

  PxMptr_t ptr = PxMcTakeBlk(PXMcTaskdefault, PXMEM_ADJUST(number_of_elements *
    size_of_element));
  memset(ptr, 0, number_of_elements * size_of_element);
  return ptr;
}
```

Code listing C.1 PXROS-HR custom allocators implementation in *allocators.c*.

Appendix D

Custom Transports Implementation

This appendix includes the custom transport function implementations that enable the communication between the micro-ROS agent and client through the use of PxNet and PXROS-HR APIs.

```
bool pxn_udp_open(uxrCustomTransport * transport)
{
  PxnTransport * PxnTransport_data = (PxnTransport*) transport->args;

   bool rv = false;

  /* Socket initialization */
  PxnTransport_data->sock = Pxn_Socket(AF_INET, SOCK_DGRAM, 0);
  if (PxnTransport_data->sock != -1)
  {
    memset(&PxnTransport_data->addr, 0, sizeof(PxnTransport_data->addr));

    /* Remote IP setup. */
    PxnTransport_data->addr.sin_family = AF_INET;
    PxnTransport_data->addr.sin_port = htons(2018);
    PxnTransport_data->addr.sin_addr.s_addr = htonl(IP_ADDR_TO_INT(192, 168, 0,
    10));  // Host IP address

    rv = true;
  }

  return rv;
}
```

Code listing D.1 Custom open function using PxNet API in *microros_transports.c*.

```
bool pxn_udp_close(uxrCustomTransport * transport){
  PxnTransport * PxnTransport_data = (PxnTransport*) transport->args;

  return Pxn_Close(PxnTransport_data->sock) == 0;
}
```

Code listing D.2 Custom close function using PxNet API in *microros_transports.c*.

```c
size_t pxn_udp_write(uxrCustomTransport* transport, const uint8_t * buf, size_t len, uint8_t * err){
  PxnTransport * PxnTransport_data = (PxnTransport*) transport->args;

  size_t rv = 0;
  PxMptr_t tmp_buf = PxMcTakeBlk(PXMcTaskdefault, PXMEM_ADJUST(len));

  if (!tmp_buf)
    PxPanic();

  /* Ensure that the buffer is aligned */
  memcpy(tmp_buf, buf, len);

  /* Prepare timeout for PxNet response "try again later" */
  int retryCount = SEND_RETRY_COUNT;
  int retryTimeout = PxTickGetTicksFromMilliSeconds(SEND_RETRY_TIMEOUT);

  /* Ask for a timeout object from the task default object pool to generate wake-up event */
  PxTo_t to = PxToRequest(PXOpoolTaskdefault, retryTimeout, EV_WAIT_TIMEOUT);

  if (PxToIdError(to) != PXERR_NOERROR)
  {
    PxPanic();
    *err = 1;
    return rv;
  }

  /* Retry loop with waiting sleep in between send retries */
  do
  {
    int32_t bytes_sent = Pxn_Sendto(PxnTransport_data->sock,
                                    (void*)tmp_buf,
                                    PXMEM_ADJUST(len),
                                    0,
                                    (struct sockaddr*)&(PxnTransport_data->addr),
                                    sizeof(PxnTransport_data->addr));

    /* Successfully sent */
    if (bytes_sent > 0)
    {
      rv = (size_t)bytes_sent;
      *err = 0;
      break;
    }
    /* Try again later returned from PxNet */
    else if (bytes_sent == EAGAIN)
    {
      PxToStart(to);
      PxAwaitEvents(EV_WAIT_TIMEOUT);
    }
    /* Error */
    else
    {
      retryCount = 0;
      break;
    }
  } while (--retryCount > 0);

  /* Entry not found after number of trials
   * Something must be wrong in the complete application
   */
  if (retryCount == 0)
    *err = 1;

  /* Stop and release the temporary timeout object */
  PxToStop(to);
  PxToRelease(to);

  /* Return aligned temporary buffer */
  PxMcReturnBlk(PXMcTaskdefault, tmp_buf);

  return rv;
}
```

Code listing D.3 Custom write function using PxNet API in *microros_transports.c*.

```
size_t pxn_udp_read(uxrCustomTransport* transport, uint8_t* buf, size_t len, int
     timeout, uint8_t* err){
  PxnTransport * PxnTransport_data = (PxnTransport*) transport->args;

  size_t rv = 0;

  /* Timeout is in ms, divide by 10 and do it ten times so the other tasks can
    run too.
   * Total timeout = retryCount * retryTimeout
   * Time for running other tasks while waiting for next try: timeout/retryCount
   * If the timeout is lower than 10 ms, wait 1 ms within each try
   */
  int retryCount;
  int retryTimeout;
  if (timeout >= 10)
  {
    retryCount = 10;
    retryTimeout = PxTickGetTicksFromMilliSeconds(timeout)/retryCount;
  }
  else
  {
    retryCount = timeout;
    retryTimeout = PxTickGetTicksFromMilliSeconds(1);
  }

  /* Get aligned temporary buffer */
  PxMptr_t tmp_buf = PxMcTakeBlk(PXMcTaskdefault, PXMEM_ADJUST(len));

  if (!tmp_buf)
    PxPanic();

  /* Receive address structure */
  struct sockaddr recv_addr;
  unsigned int recv_addr_size = sizeof(recv_addr);

  /* Ask for a timeout object from the task default object pool to generate wake
    -up event */
  PxTo_t to = PxToRequest(PXOpoolTaskdefault, retryTimeout, EV_WAIT_TIMEOUT);

  if (PxToIdError(to) != PXERR_NOERROR)
  {
    PxPanic();
    *err = 1;
    return rv;
  }

  /* Retry loop with waiting sleep in between send retries */
  do
  {
    int32_t bytes_received = Pxn_Recvfrom(PxnTransport_data->sock,
                                          (void*)tmp_buf,
                                          PXMEM_ADJUST(len),
                                          MSG_DONTWAIT,
                                            (struct sockaddr*) &recv_addr,
                                            &recv_addr_size);

    /* data successfully received */
    if (bytes_received >= 0)
    {
      /* Copy received data from temporary buffer to receive buffer */
      memcpy(buf, tmp_buf, len);
      rv = (size_t)bytes_received;
      *err = 0;
      break;
    }
    /* Error */
    else
    {
      PxToStart(to);
      PxAwaitEvents(EV_WAIT_TIMEOUT);
    }
  } while (--retryCount > 0);

  /* Entry not found after number of trials
   * Something must be wrong in the complete application
   */
  if (retryCount == 0)
    *err = 1;

  /* Stop and release the temporary timeout object */
  PxToStop(to);
  PxToRelease(to);

  /* Return aligned temporary buffer */
  PxMcReturnBlk(PXMcTaskdefault, tmp_buf);

  return rv;
}
```

Code listing D.4 Custom read function using PxNet API in *microros_transports.c*.

Bibliography

1. MICRO-ROS. *Supported RTOSes* [online]. 2023. [visited on 2023-04-25]. Available from: https://micro.ros.org/docs/overview/rtos/.
2. MACENSKI, Steven; FOOTE, Tully; GERKEY, Brian; LALANCETTE, Chris; WOODALL, William. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics* [online]. 2022, vol. 7, no. 66, eabm6074 [visited on 2023-04-25]. Available from DOI: 10.1126/scirobotics.abm6074.
3. OPEN ROBOTICS. *ROS - Robot Operating System* [online]. 2022-07. [visited on 2023-04-25]. Available from: https://www.ros.org/.
4. OPEN ROBOTICS. *Why ROS?* [online]. [visited on 2023-04-25]. Available from: https://www.ros.org/blog/why-ros/.
5. MAZZARI, Vanessa. *ROS – Robot Operating System* [online]. 2016-03. [visited on 2023-04-25]. Available from: https://www.generationrobots.com/blog/en/ros-robot-operating-system-2/.
6. GERKEY, Brian. *Why ROS 2?* [online]. 2014-06. [visited on 2023-04-25]. Available from: https://design.ros2.org/articles/why_ros2.html.
7. THOMAS, Dirk. *Changes between ROS 1 and ROS 2* [online]. 2015-09. [visited on 2023-04-25]. Available from: https://design.ros2.org/articles/changes.html.
8. OPEN ROBOTICS. *ROS 2 Documentation* [online]. 2023-02. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/index.html.
9. APEX.AI. *FAQ* [online]. 2023. [visited on 2023-04-25]. Available from: https://www.apex.ai/faq.
10. OPEN ROBOTICS. *Understanding nodes* [online]. 2022-06. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html.
11. OPEN ROBOTICS. *Understanding parameters* [online]. 2022-06. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Parameters/Understanding-ROS2-Parameters.html.
12. OPEN ROBOTICS. *About ROS 2 interfaces* [online]. 2021-08. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Concepts/About-ROS-Interfaces.html.
13. OPEN ROBOTICS. *Understanding topics* [online]. 2022-06. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html.
14. MILLÁN, José L. *Creating ROS 2 Services* [online]. 2022-08. [visited on 2023-04-25]. Available from: https://foxglove.dev/blog/creating-ros2-services.

15. OPEN ROBOTICS. *Understanding services* [online]. 2022-06. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html.
16. WEON, Esther. *Creating ROS 2 Actions* [online]. 2022-08. [visited on 2023-04-25]. Available from: https://foxglove.dev/blog/creating-ros2-actions.
17. OPEN ROBOTICS. *Understanding actions* [online]. 2022-07. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html.
18. OPEN ROBOTICS. *About internal ROS 2 interfaces* [online]. 2021-03. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Concepts/About-Internal-Interfaces.html.
19. OPEN ROBOTICS. *Executors* [online]. 2022-12. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Concepts/About-Executors.html.
20. OPEN ROBOTICS. *About ROS 2 client libraries* [online]. 2022-08. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Concepts/About-ROS-2-Client-Libraries.html.
21. OPEN ROBOTICS. *The ROS_DOMAIN_ID* [online]. 2021-09. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Concepts/About-Domain-ID.html.
22. OPEN ROBOTICS. *About Quality of Service settings* [online]. 2022-07. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Concepts/About-Quality-of-Service-Settings.html.
23. OPEN ROBOTICS. *Related Projects* [online]. 2022-07. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Related-Projects.html.
24. MICRO-ROS. *Micro-ROS* [online]. 2023. [visited on 2023-04-25]. Available from: https://micro.ros.org/.
25. MICRO-ROS. *Features and Architecture* [online]. 2023. [visited on 2023-04-25]. Available from: https://micro.ros.org/docs/overview/features/.
26. BELSARE, Kaiwalya; RODRIGUEZ, Antonio Cuadros; SÁNCHEZ, Pablo Garrido; HIERRO, Juanjo; KOŁCON, Tomasz; LANGE, Ralph; LÜTKEBOHLE, Ingo; MALKI, Alexandre; LOSA, Jaime Martin; MELENDEZ, Francisco; RODRIGUEZ, Maria Merlan; NORDMANN, Arne; STASCHULAT, Jan; MENDEL, Julian von. Micro-ROS. In: *Robot Operating System (ROS): The Complete Reference (Volume 7)*. Ed. by KOUBAA, Anis. Springer, 2023, pp. 3–55. Available from DOI: 10.1007/978-3-031-09062-2_2.
27. MICRO-ROS. *Micro_ros_setup* [online]. 2023-02. [visited on 2023-04-25]. Available from: https://github.com/micro-ROS/micro_ros_setup.
28. MICRO-ROS. *Supported Hardware* [online]. 2023. [visited on 2023-04-25]. Available from: https://micro.ros.org/docs/overview/hardware/.
29. OBJECT MANAGEMENT GROUP. *What is DDS?* [online]. [visited on 2023-04-25]. Available from: https://www.dds-foundation.org/what-is-dds-3/.
30. RAVAL, Khushbu. *Datatechvibe Explains: Data Distribution Service (DDS) Protocol* [online]. 2022-07. [visited on 2023-04-25]. Available from: https://datatechvibe.com/data/datatechvibe-explains-data-distribution-service-dds-protocol/.
31. O'RIORDAN, Matthew. *Everything You Need To Know About Publish/Subscribe* [online]. 2020-07. [visited on 2023-04-25]. Available from: https://ably.com/topic/pub-sub.
32. EPROSIMA. *Introduction to DDS* [online]. [visited on 2023-04-25]. Available from: https://www.eprosima.com/index.php/resources-all/whitepapers/dds.

33. PCMAG. *Wire protocol* [online]. [visited on 2023-04-25]. Available from: https://www.pcmag.com/encyclopedia/term/wire-protocol.

34. EPROSIMA. *RTPS Introduction* [online]. [visited on 2023-04-25]. Available from: https://www.eprosima.com/index.php/resources-all/whitepapers/rtps.

35. OPEN ROBOTICS. *About different ROS 2 DDS/RTPS vendors* [online]. 2022-03. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Concepts/About-Different-Middleware-Vendors.html.

36. MICRO-ROS. *Micro XRCE-DDS* [online]. 2021-03. [visited on 2023-04-25]. Available from: https://micro.ros.org/docs/concepts/middleware/Micro_XRCE-DDS/.

37. EPROSIMA. *Introduction to XRCE* [online]. [visited on 2023-04-25]. Available from: https://www.eprosima.com/index.php/resources-all/whitepapers/xrce.

38. HIGHTEC EDV-SYSTEME GMBH. *Infineon Preferred Design House* [online]. 2022. [visited on 2023-04-25]. Available from: https://hightec-rt.com/en/company/pdh.

39. HIGHTEC EDV-SYSTEME GMBH. *About us* [online]. 2022. [visited on 2023-04-25]. Available from: https://hightec-rt.com/en/company/about-us.

40. HIGHTEC EDV-SYSTEME GMBH. *PxNet STX: Quick Guide*. 2022-10.

41. GILLIS, Alexander S. *real-time operating system (RTOS)* [online]. 2022-02. [visited on 2023-04-25]. Available from: https://www.techtarget.com/searchdatacenter/definition/real-time-operating-system.

42. HIGHTEC EDV-SYSTEME GMBH. *PXROS-HR Kernel v8.2.0: User's Guide*. 2020.

43. HIGHTEC EDV-SYSTEME GMBH. *PXROS - Real-time OS for TriCore and AURIX* [online]. 2022. [visited on 2023-04-25]. Available from: https://hightec-rt.com/en/products/real-time-os.

44. HIGHTEC EDV-SYSTEME GMBH. *HighTec Content Manager* [online]. 2023. [visited on 2023-04-25]. Available from: https://docs.hightec-rt.com/component-ide-qsg/3.1.0/chapter/chapter-content-manager.html.

45. OPEN ROBOTICS. *Ubuntu (source)* [online]. 2023-03. [visited on 2023-04-25]. Available from: https://docs.ros.org/en/humble/Installation/Alternatives/Ubuntu-Development-Setup.html.

46. MICRO-ROS. *First micro-ROS Application on Linux* [online]. 2023-03. [visited on 2023-04-25]. Available from: https://micro.ros.org/docs/tutorials/core/first_application_linux/.

47. MICRO-ROS. *Creating custom static micro-ROS library* [online]. 2021-07. [visited on 2023-04-25]. Available from: https://micro.ros.org/docs/tutorials/advanced/create_custom_static_library/.

48. THOMAS, Dirk. *colcon - collective construction* [online]. 2022-06. [visited on 2023-04-25]. Available from: https://colcon.readthedocs.io/en/released/index.html.

49. DR-MH; GARRIDO, Pablo. *Static library creation: Wrong folder structure in firmware/build /include* [online]. 2022-05. [visited on 2023-04-25]. Available from: https://github.com/micro-ROS/micro_ros_setup/issues/530.

50. HIGHTEC EDV-SYSTEME GMBH. *TC39x PXROS-HR BSP example: Quick Guide*. 2020-11.

51. MICRO-ROS. *Micro_ros_demos* [online]. 2022-05. [visited on 2023-04-25]. Available from: https://github.com/micro-ROS/micro-ROS-demos.

52. MICRO-ROS. *micro-ROS utilities* [online]. 2023. [visited on 2023-04-25]. Available from: https://micro.ros.org/docs/tutorials/programming_rcl_rclc/micro-ROS/.

53. MICRO-ROS. *Creating custom micro-ROS transports* [online]. 2021-07. [visited on 2023-04-25]. Available from: `https://micro.ros.org/docs/tutorials/advanced/create_custom_transports/`.

54. [MICRO-ROS]. *Memory profiling* [online]. 2021-04. [visited on 2023-04-25]. Available from: `https://micro.ros.org/docs/concepts/benchmarking/memo_prof/`.

55. WYROBEK, Keenan. *The Origin Story of ROS, the Linux of Robotics* [online]. 2017-10. [visited on 2023-04-25]. Available from: `https://spectrum.ieee.org/the-origin-story-of-ros-the-linux-of-robotics`.

56. MAILCHIMP. *What is a Pitch Deck?* [online]. [visited on 2023-04-25]. Available from: `https://mailchimp.com/resources/what-is-pitch-deck/`.

57. GREENLEIGH, John. *PR2* [online]. [visited on 2021-04-25]. Available from: `https://robots.ieee.org/robots/pr2/`.

58. TELLEZ, Ricardo. *A History of ROS (Robot Operating System)* [online]. 2019-07. [visited on 2023-04-25]. Available from: `https://www.theconstructsim.com/history-ros/`.

59. OPEN ROBOTICS. *What is a TurtleBot?* [online]. [visited on 2021-04-25]. Available from: `https://www.turtlebot.com/`.