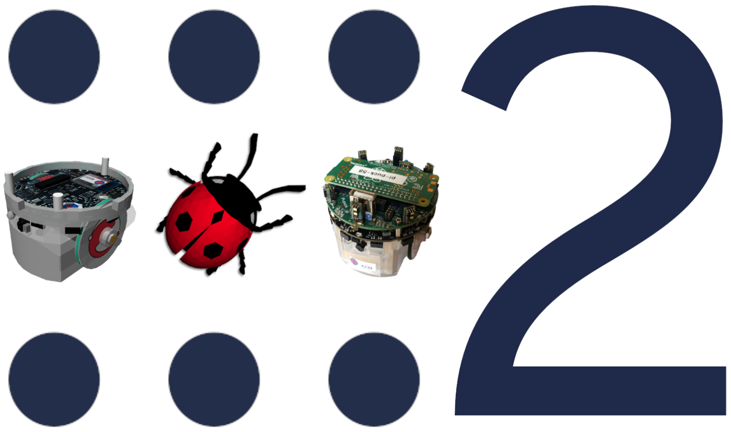
SS 2020, MT-S, No. DISAL-MP44	Start: 17.02.2020
	Finish: 14.08.2020

ROS2 Programming Interface for the E-puck2 Robot

Darko Lukic



Professor: Alcherio Martinoli
Assistant: David Mansolino, Cyrill Baumann and Olivier Michel

Abstract

Robotics simulations have been proven to be a powerful tool for developing a robot controller as they are easy to set up, cheap, fast, and convenient to use. However, the final objective is usually to deploy the controller on the real robots or even to run the controller on an arbitrary robot. This thesis presents a ROS2 driver for e-puck2 physical robots and a generalized ROS2 driver for Webots simulated robots. The ROS2 drivers expose a nearly identical ROS2 interface that allows a controller to interact in the same way with the physical e-puck2 and the simulated robots without changes. Effectively, it allows the controller developers a seamless transition between simulated and physical e-puck2 robots or other simulated robots. The ROS2 drivers are validated in multiple scenarios, like navigation and mapping. The results prove that researchers can quickly validate their ROS2 controllers on the e-puck2 physical or simulated robot and other Webots simulated robots.

***Keywords*—** robotics, simulation, ROS, ROS2, e-puck2, Webots

Acknowledgments

I am incredibly grateful to all fantastic people from Cyberbotics and Distributed Intelligent Systems and Algorithms Laboratory (DISAL) as this project would not be possible without their help.

Therefore, I would like to thank David Mansolino, my assistant from Cyberbotics, for all time he put in reviewing my code, to all valuable suggestions and comments, and quick responses to my problems. I learned a lot of practical skills from him. Nevertheless, I would like to thank Cyrill Baumann, my assistant from DISAL, for his patience, plenty of advice, and efforts to structure the project.

I want to thank Olivier Michel from Cyberbotics for his general support, and especially for his critical problem-solving suggestions. I would like to sincerely thank my supervisor, professor Alcherio Martinoli, from DISAL, for the sincere and valuable encouragement extended to me.

Finally, I thank my family, my friends, and Jelena Banjac for unceasing encouragement, support, and attention.

Contents

Acronyms

API Application Programming Interface

CI Continuous Integration

CPU Central Processing Unit

CSI Camera Serial Interface

DAC Digital-to-Analog Converter

DDS Data Distribution Service

DISAL Distributed Intelligent Systems and Algorithms Laboratory

DMA Direct Memory Access

DSP Digital Signal Processor

EPFL École Polytechnique Fédérale de Lausanne

FoV Field of View

FPGA Field-Programmable Gate Array

FPS Frames Per Second

FPU Floating-Point Unit

GPU Graphics Processing Unit

I^{2}C Inter-Integrated Circuit

ID Identifier

IMU Inertial Measurement Unit

IoU Intersection over Union

JPEG Joint Photographic Experts Group

LED Light-Emitting Diode

LiDAR Light Detection And Ranging sensor

LTS Long-Term Support

MCU Microcontroller Unit

MIPS Million Instructions Per Second

MMAL Multi-Media Abstraction Layer

ODE	Open Dynamics Engine
OpenMAX	Open Media Acceleration
OS	Operating System
PC	Personal Computer
PCB	Printed Circuit Board
QoS	Quality of Service
RGB	Red Green Blue
ROS	Robotic Operating System
ROS1	Robotic Operating System1
ROS2	Robotic Operating System2
RTOS	Real-time Operating System
SD card	Secure Digital card
SLAM	Simultaneous Localization And Mapping
SoC	System On a Chip
TCP	Transmission Control Protocol
TCPROS	Transmission Control Protocol for Robotic Operating System
ToF	Time of Flight
UDP	User Datagram Protocol
UDPROS	User Datagram Protocol for Robotic Operating System
URDF	Unified Robot Description Format
USB	Universal Serial Bus
V4L2	Video4Linux 2
VCHI	VideoCore Host Interface
VCOS	VideoCore Operating System
VRML	Virtual Reality Modeling Language
X3D	Extensible 3D Graphics
XML	Extensible Markup Language
YUV	Luminance-Bandwidth-Chrominance

List of Figures

List of Tables

Chapter 1 Introduction

This project's motivation is to close the loop between the simulation and the physical world in a robotics application. Furthermore, the motivation is also to allow an effortless controller transfer to different simulated robots. The project aims to utilize ROS2, the second iteration of a popular robotics framework, to develop a standard interface for a range of different robots including physical and simulated e-puck2 robots.

1.1 Problem Statement

Robotics simulations have been proven to be a powerful tool for research and development as they are easy to set up, cheap, fast, and convenient to use [6]. Usually, the final objective is to perform the experiments on real robots. Therefore, the problem roboticists are facing is a transition from the simulated to the physical world, also called bridging the reality gap. The aim is to make this process simpler and faster, effectively minimizing this gap. The solution should be easy to integrate into the simulation and not cause a significant computational overhead for the physical robot.

Another challenge roboticists are facing is code reuse. In the world of diverse hardware solutions for robots, it is hard to create a modular software solution that can be reused on different robotics platforms. For example, navigation, Simultaneous Localization And Mapping (SLAM), and localization are only a few algorithms widely used in mobile robotics, and many mobile robots could reuse that. The researchers need modular, reusable software to share robot controllers among colleagues. Then, the researchers can evaluate and compare the controller on different robots [7]. As for the previous challenge, the solution must be easily integrated into the simulation.

In order to demonstrate the flexibility of the proposed solution, it will be implemented for the e-puck2 physical and simulated robot. The solution also has to be scaled to the Khepera IV simulated robot and potentially other simulated robots.

1.2 Project Objective

As described in the previous section, two challenges in robotics have to be tackled in this thesis. The first is closing the loop between the simulation and the physical world, and the second is software reuse between different robots.

From the software point of view, solving those problems can be addressed by defining a common API for the physical and simulated robot. ROS is often used as a meta-operating system for this purpose as the common API can be defined in ROS. The newest version of ROS, ROS2, includes useful improvements relevant to the project, and it is about to replace the old version completely. Therefore, the proposed solution is based on developing ROS2 nodes that closely interact with the hardware and the Webots[1] simulation - ROS2 driver. As a result, a user should have a development workflow, as shown in Figure 1.1.

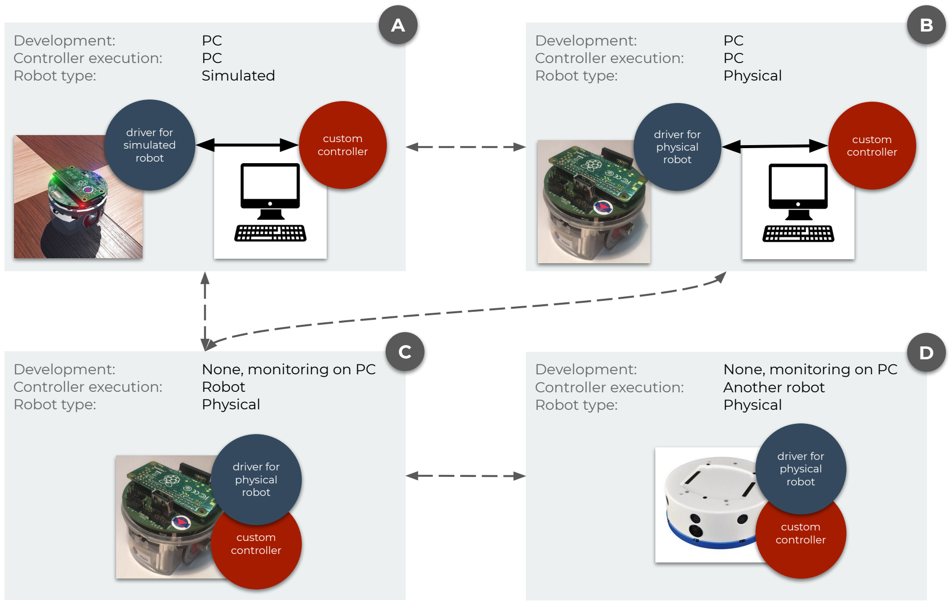

A
Development: PC
Controller execution: PC
Robot type: Simulated
driver for simulated robot
custom controller
B
Development: PC
Controller execution: PC
Robot type: Physical
driver for physical robot
custom controller
C
Development: None, monitoring on PC
Controller execution: Robot
Robot type: Physical
driver for physical robot
custom controller
D
Development: None, monitoring on PC
Controller execution: Another robot
Robot type: Physical
driver for physical robot
custom controller

Figure 1.1: A development workflow (in robotics) that has to be achieved with the proposed solution. The circle represents a piece of software that the user wants to develop to control the robot, and it is the same in each scenario. The dashed arrows are transitions in the development workflow, while the solid arrows represent a network protocol.

In Figure 1.1, the user develops a robot controller called `custom con-`

[1]Webots is a desktop application, developed at École Polytechnique Fédérale de Lausanne (EPFL), used to simulate robots. It will be more closely introduced in the following chapter.

`troller`. From the user's perspective, the controller is supposed to be the same in each scenario. First, the user should start by developing the controller on a PC and testing it in the simulation (scenario A). After the user is satisfied with the robot's behavior in simulation, the controller can still be executed on the PC, but now it can control the physical robot (scenario B). If the robot's behavior is not desired, the user can improve the simulation model and test it again in the simulation (scenario A), reducing the reality gap. Once everything works as expected, the user should move the controller to the robot (scenario C). Here, it is possible to, for example, hit the computational limit of the on-board computer. In this case, the controller should then be edited and tested in simulation again (scenario B), moving back to the on-board computer once it is ready. Finally, the controller can be shared with the other researchers to evaluate it on different robots or to further improve it (scenario D).

The project has to be done in three phases. First, specific ROS2 nodes for an e-puck2 simulated and physical robot have to be developed. Second, examples that utilize the nodes have to be created. The purpose of this phase is to evaluate the nodes and to give the users usage examples. Finally, in the third phase, the specific ROS2 node for simulated e-puck2 robot has to be generalized to support other robots, focusing on Khepera IV and TurtleBot3 robots. The final software has to be peer-reviewed, united tested, code quality tested, automated with CI[2], user friendly, and well documented with comprehensive tutorials.

1.3 Document Structure

The structure of the thesis roughly follows the phases described in the previous section. It should allow readers to skip information, but also to reduce the chance of missing important details. Therefore, the project is presented as follows:

- **Chapter 2: Background and Related Work** gives the theoretical background on common concepts, and software and hardware technologies, utilized in the project. Also, it clarifies relations with relevant projects.

- **Chapter 3: E-puck2 Simulation and ROS2 Interface** explains a process used to create a ROS2 node that exposes access to simulated the e-puck2 robot's sensors and actuators through ROS2 interface.

[2]The CI automation has to be done with Industrial CI. This CI is created by ROS-industrial, an organization committed to close a gap between research and industry by bringing industry standards to ROS2.

- **Chapter 4: ROS2 Interface for Physical E-puck2** has a goal to explain implementation of the same ROS2 interface for e-puck2 physical robot.
- **Chapter 5: E-puck2 Demos** shows tools, existing packages and custom created controllers that utilize the e-puck2 ROS2 interface.
- **Chapter 6: The Generalization of ROS2 Interface for Webots** gives implementation overview on generalized ROS2 driver for simulated robots. It aims at creating universal ROS2 driver to support e-puck2, Khepera IV, TurtleBot3 Burger and potentially the other robots as well.
- **Chapter 7: Results and Interpretation** quantifies difference between the ROS2 driver for the simulated and the physical robot, difference between e-puck2 and Khepera IV ROS2 interfaces and it shows simplification resulted by generalization described in the previous chapter.
- **Chapter 8: Conclusion and Future Work** presents the project summary, limitations, impact and potential improvements.

Chapter 2 Background and Related Work

This chapter has two purposes. First, to give a reader a theoretical background about the concepts and technologies needed for understanding the project. Second, it shows the related work - how this project is different and what improvements it brings in comparison to the existing solutions.

2.1 ROS

The initial founders of ROS define it as an open-source operating system which, instead of process management and scheduling, provides a communication layer on top of the host operating systems of a heterogeneous compute cluster [8].

Figure 2.1: ROS described through a picture, available at the official ROS website

One also can say that ROS is a collection of tools, libraries, and conventions that simplify developing a complex robot behavior. In one interview, Roger Barga, leader of Amazon's Web Service (AWS RoboMaker), emphasized the importance of ROS in robotics by saying *"We think that ROS is becoming the Linux for the robots of the future"*[1]. Defining ROS in one sentence is hard, but in further text, two crucial concepts of ROS will be covered.

The first is the concept of a node. A task of the node is to perform a computation. The nodes utilize a publisher-subscriber communication model to exchange messages with each other. In that way, the nodes are

[1]Roger Barga is an interview by Ricardo Tellez (from The Construct) in ROS Developers podcast - `https://www.theconstructsim.com/aws-robomaker-with-roger-barga/`.

combined into a graph that can perform more complex tasks. There are a few important characteristics of each node:

- The node can be implemented in various programming languages, without affecting the other nodes. Being a programming language agnostic allows developers to optimize the nodes for different behaviors. For example, if the node has to be efficient or interact closely with the hardware, C programming language may be more suitable. Otherwise, if fast prototyping is important, Python may be a better fit. As a result, community nodes are implemented in a programming language that fits best the node's purpose.
- The nodes can be run on different computers. Each ROS node includes a mechanism to communicate locally or over the network with the other nodes. A complex robotics system often includes multiple powerful computers and dozens of MCUs or Field-Programmable Gate Arrays (FPGAs) specialized for various tasks, and therefore, this capability of ROS nodes is significant. In this project, for example:
  - MCU is used to execute real-time tasks, like motor control,
  - on-board computer is used for heavy tasks, like perception,
  - while workstation (PC) is used for data visualization, monitoring, and debugging.
- The nodes can run on virtually any Operating System (OS). Since ROS2, Linux, Mac, and Windows are officially supported. A stripped version of ROS2, micro-ROS, runs even on NuttX, FreeRTOS, and Zephyr, and the community has been porting to other OSs as well.
- Since ROS2, the nodes are fully distributed. It means that there is no single message broker to orchestrate the communication. Being distributed significantly increases robustness as there is no single point of failure; if a node crashes the rest of the system will continue to function.

In Figure 2.2, a graph of ROS nodes is given, representing the node concepts given before.

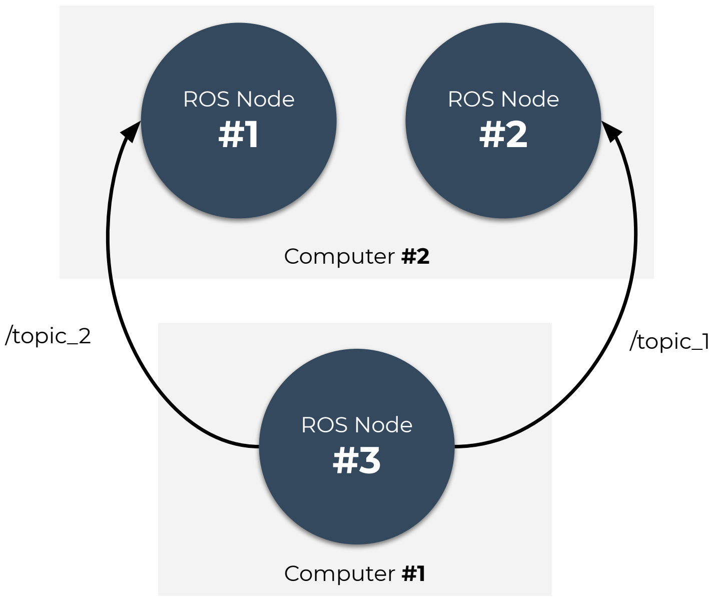


Figure 2.2: General example of three nodes exchange messages through two topics

The second important concept in ROS is the means of communication between the nodes. Mostly, communication between nodes is done through topics, but there other types of communication:

- Topics use publisher-subscriber communication model [9]. It means that a node's message will be received by all nodes that are subscribed to the corresponding topic. Quality of Service (QoS) was important to ROS team and therefore, the following QoS can be defined:
  - history, keep last (keep only the last N messages) or keep all,
  - reliability, best effort (may lose messages if the network is not stable) or reliable (guarantees message delivery) and
  - durability, transient local (nodes that got subscribed to a topic will receive the last message even though the message is published a long time before) or volatile (messages will not be preserved for late joiners).
- Services are used to get a response from the other nodes. The concept is very similar to functions with return values. A client node sends a service request with data to a node that provides service, and after the result is ready, the client node gets the response. A typical request can be, *"Is the motor turned on?"* or *"Retrieve the full map now"*.

- Parameters allow nodes to be configured by the user or other nodes, on startup or during run-time. The underlying implementation is based on the previously explained services. It means the nodes can be configured over the network.
- Actions are available in ROS core since ROS2, and they are similar to the services. A difference is that the actions are optimized for requests that generally take longer to execute and need continuous feedback. A typical action can be, *"Move the robot to the new position and while doing it, keep sending me a position"*.

The nodes and means of communication between the nodes, are two core concepts of ROS around which all ROS tools are built. Depending on the area of robotics, new concepts may also emerge, but those two are the foundation of the ROS.

2.1.1 ROS Messages

One of the subobjectives of this master project is to provide ROS2 interface that is as compatible as possible with the existing packages. That will allow users to simply integrate existing ROS2 packages and significantly increase their productivity. To achieve the objective, ROS2 defines a vast list of existing message, service, and action types. In the scope of the project, only message types will be further described.

Except for the standard types available in most programming languages, such as bool, integer, float, and string, some messages are more specific to the robotics such as odometry (for describing odometry data), twist (for defining angular and linear velocity), and range (for data from distance sensors). These messages are exchanged through the previously described topics. Therefore, integrating a community ROS2 package is a matter of launching (and sometimes configuring it), the package should automatically start publishing and subscribing to the messages as common message types are used.

There are a few groups of message types, some of which are:

- `std_msgs` is a wrapper around primitive types such as `Bool`, `Int32`, `Float64`, `String`, and `Float64MultiArray`. These message types are usually used to compose more complex message types and in general should be avoidable if there is more suitable type from the other groups.
- `geometry_msgs` contains geometry primitives. For example, `Twist` is used to describe linear and angular velocity (usually used to control robot's velocity), rotations are described with `Quaternion` message types, robot's pose `Pose` (e.g. the navigation stack uses this message

type to describe the robot's goal position and orientation) and `Transform` to describe relative orientation and translation of two frames.

- `nav_msgs` group contains message types used to interact with the navigation stack and the related packages. It defines message types such as `OccupancyGrid` that represent a map and `Odometry` that contains odometry data.

- `sensor_msgs` contains message types that are commonly used to describe data from sensor such as distance sensors (`Range`), LiDARs (`LaserScan`), cameras (`CameraInfo` and `Image`), light sensors (`Illuminance`) and similar.

There are many more message group types, but those are the most relevant ones for this project.

2.1.2 ROS Distributions

ROS distributions are another vital aspect of the project. The newest ROS distribution (as of the time of writing the thesis) is deliberately chosen. The choice will be described later, but first, ROS versioning has to be described.

ROS team releases a new ROS distribution every six months, following Ubuntu's release cycle. Every two years, a new ROS Long-Term Support (LTS) is released, a few weeks after Ubuntu's LTS release, as ROS relies on the latest Ubuntu LTS.

In December 2017, ROS2 is introduced, and since then ROS team has been publishing ROS and ROS releases every six months. ROS2 should fully replace the ROS1 once ROS2 mature[2].

[2]Dirk Thomas, principle software engineer working on ROS core, wrote about ROS future at ROS Discourse - `https://discourse.ros.org/t/planning-future-ros-1-distribution-s/6538`. He stated ROS1 will be most probably supported until 2025 and then it should be completely replaced by ROS2

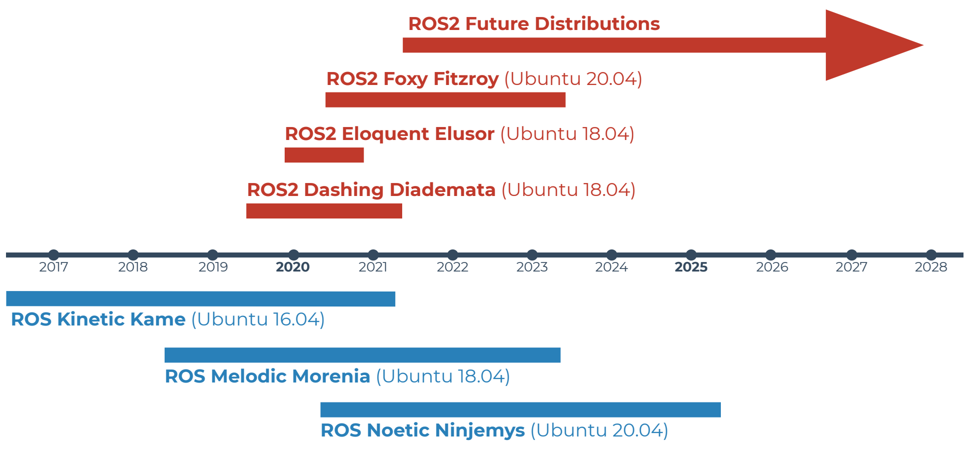


Figure 2.3: Timeline of ROS1 and ROS2 distributions

In Figure 2.3, a few latest ROS distributions are shown. Note that they all rely on Ubuntu LTS distributions and that ROS2 Foxy Fitzroy is the first ROS2 distribution with three years of support.

In this project, the importance of ROS2 is recognized, and therefore it is chosen instead of ROS1. Furthermore, the master project had started before ROS2 Foxy Fitzroy was released, and therefore the initial code is written for ROS2 Eloquent Elusor with a plan to adopt ROS2 Foxy Fitzroy. Finally, the master project is compatible with ROS2 Eloquent Elusor and ROS2 Foxy Fitzroy.

2.2 Robotics Platforms

The project's main objective is to introduce ROS2 support for the e-puck2 physical and simulated robot, and other simulated robots while focusing on Khepera IV. Therefore, those two robots are will be described in more detail.

2.2.1 E-puck2

E-puck2 is the second generation of the e-puck robot[10]. E-pucks are small differential wheeled robots (e-puck and e-puck2 have the same footprint) with a radius of 35mm. As a miniature robot, it is the perfect candidate for education and multi-robot research. It is originally designed for micro-engineering education by Michael Bonani and Francesco Mondada at EPFL. The robot is open hardware, and the software is open-source.

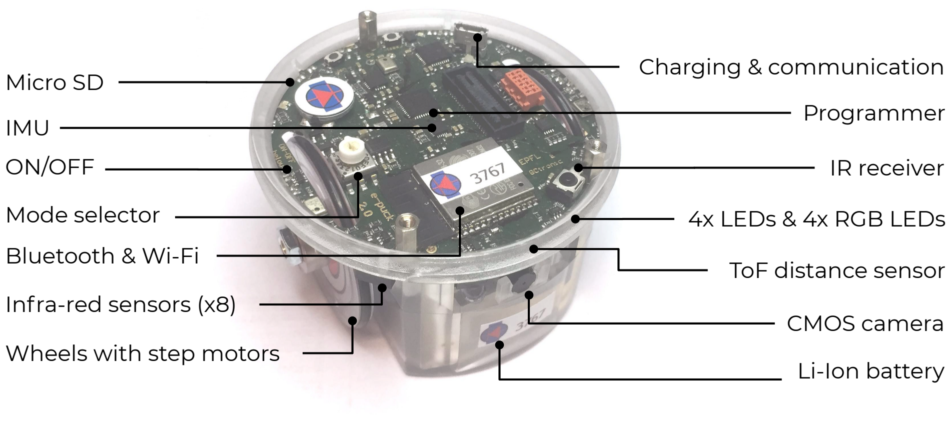


Figure 2.4: E-puck2 robot with list of main components

The robot is shown in Figure 2.4 and its main specifications are given in Table. 2.1.

Size, weight	70mm diameter, 130g
MCU	32-bit STM32F407 @ 168 MHz (210 MIPS), DSP and FPU, DMA
Motors	2 stepper motors, 50:1 reduction gear and 20 steps per revolution
Max velocity	0.154m/s
Distance sensor	8 infra-red sensors (up to 0.06m) and one ToF (up to 2m)
Camera	640x480 at 15FPS
IMU	3D accelerometer, 3D gyro, 3D magnetometer
LEDs	4 red LEDs and 4 RGB LEDs

Table 2.1: Relevant specifications of e-puck2 robot

Pi-puck

E-puck2 robot allows extensions to be added providing different features such as additional autonomy, ground sensors, or additional processing power. The pi-puck extension, for example, consists of a Raspberry Pi Zero W and adapter PCB, and interacts directly with on-board MCU and sensors [1]. Since Raspberry Pi Zero W is Linux base board, it can run ROS with full Data Distribution Service (DDS) implementation.

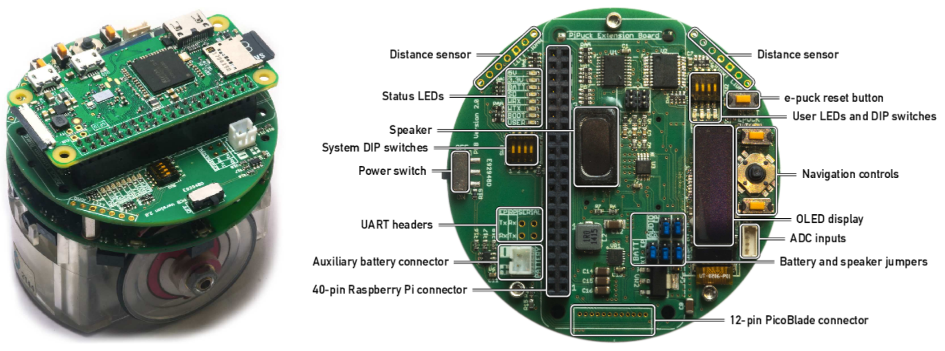


Figure 2.5: Pi-puck extension consist of two parts, PCB and Raspberry Pi Zero W [1]

In the project, Raspberry Pi Zero W allows us to run ROS2 nodes on the robot and execute complex operations such as JPEG image compression that otherwise would not be possible.

2.2.2 Khepera IV

Khepera IV is a similar robot to e-puck2, but is more prominent in size (with a radius of 70mm), and has more powerful sensors and computational units [2]. The robot is designed by K-Team[3] to fit any indoor lab application.

Figure 2.6: Khepera IV robot [2]

The robot is shown in Figure 2.6 and its main specifications are given in Table. 2.2.

[3]Official website of K-Team is available at `https://www.k-team.com/`.

Size, weight	140mm diameter, 540g
Processor	800MHz ARM Cortex-A8 Processor and MCU
Motors	2 DC brushed motors with incremental encoders
Max velocity	0.8m/s
Distance sensor	8 infra-red (up to 0.25m) and 5 ultrasonic (up to 2m)
Camera	752x480 at 30FPS
IMU	3D accelerometer, 3D gyro
LEDs	3 RGB LEDs

Table 2.2: Relevant specifications of Khepera IV robot

2.3 Webots

Webots is an open-source robot simulator developed by Cyberbotics[4], initially designed at EPFL. The simulator provides a development environment to model, program, and simulate robots [6, 11, 12].

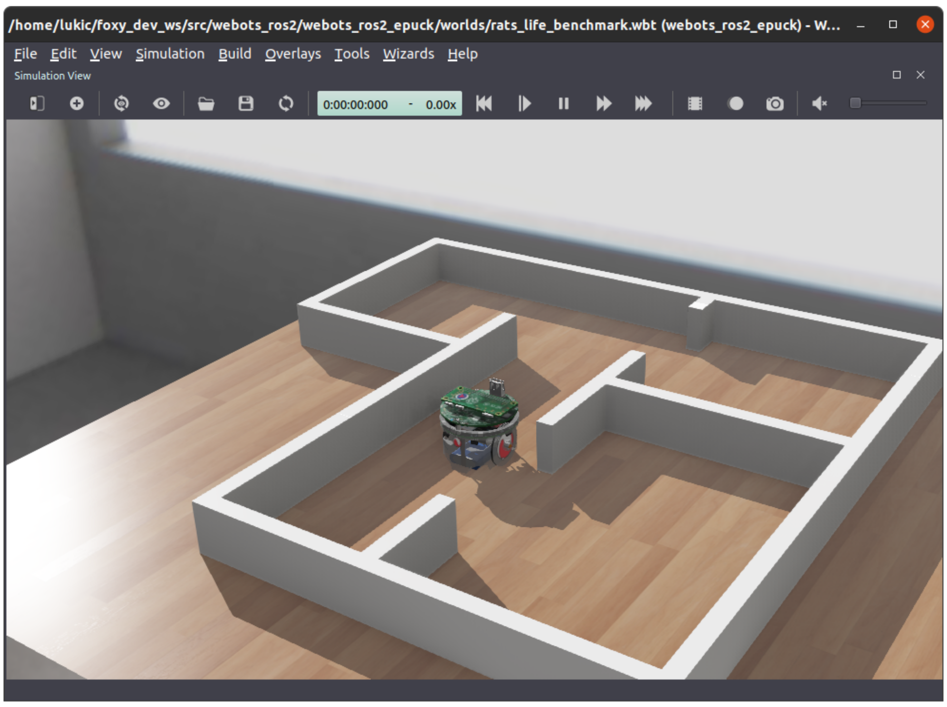

/home/lukic/foxy_dev_ws/src/webots_ros2/webots_ros2_epuck/worlds/rats_life_benchmark.wbt (webots_ros2_epuck) - W...
File Edit View Simulation Build Overlays Tools Wizards Help
Simulation View
0:00:00:000
0.00x

Figure 2.7: Webots robot simulator: e-puck2 robot on a table

The main Webots features are:

[4]Official website of Cyberbotics is available at `https://cyberbotics.com/`.

- Robot/world editor: The models are stored in VRML97 format, allowing users to either change the document manually or through the Webots user interface. The Webots world editor allows the users to see changes in the world immediately. It is possible to include prebuilt models, and the users can choose among a substantial library of models.

- Realistic simulation: The simulations produced in Webots are realistic compared to the similar products. The physics simulation is based on a modified version of Open Dynamics Engine (ODE). The models shipped with the Webots are visually very detailed, although still optimized for high performance.

- Programming interface: An API for programming robots is officially available in the most popular programming languages, C, C++, Python, MATLAB, and Java, while community-contributed packages further extend this list[5]. The API allows simple access to the sensors and actuators available in the robots.

- Deterministic simulations: Webots guarantees a simulation to output the same behavior every time it runs (the user can also choose non-deterministic simulation). Although a non-deterministic simulation is preferable before transferring to a real robot, deterministic simulation is beneficial when doing initial algorithm tests. Additionally, this feature can be exploited to simplify automated testing in CI.

Those are the main points that justify the usage of Webots over similar products.

2.4 Related Work

The existing ROS support for the e-puck2 physical robot and Webots will be analyzed. Drawbacks in current implementations of ROS support are the motivation for this project; therefore, they will be explained in this section.

2.4.1 ROS Support for E-puck2

GCtronic, a company behind e-puck robots, has two types of ROS drivers available. The first is made for e-puck robots without pi-puck extension[6]. In that case, ROS driver runs on a workstation while communicating with the

[5]An example of API in Haskell programming language, created by the community, is available at `https://github.com/cyberbotics/HsWebots`.

[6]ROS driver that doesn't run on pi-puck extension is available at `https://github.com/gctronic/epuck_driver_cpp/tree/e-puck2`.

e-puck over Bluetooth. The second ROS driver implementation runs on pi-puck extension[7] and it is similar to the ROS driver we aim to develop. The main drawbacks are that it is not available for ROS2, it is not unit tested nor code quality tested, it does not include a camera driver, and there is no comprehensive documentation.

	GCtronic's #1	**GCtronic's #2**	**Proposed solution**
ROS2 support	No	No	Yes
Communication	Bluetooth	UDPROS/TCPROS	DDS
Camera	160x120@4	No	640x480@10
Battery autonomy	Long	Short	Short
Unit tests	No	No	Yes, with CI
Code quality tests	No	No	Yes, with CI
Cross-compilation	No	No	Yes, tools are given
Independent from PC	No	Yes	Yes

Table 2.3: Comparison of the proposed ROS2 driver with the existing implementations (aspects in which the proposed solution is better are highlighted).

As showed in Table 2.3, the proposed solution offers better implementation in many aspects. The proposed solution's main drawback is the battery autonomy as the pi-puck extension requires a significant amount of power (around 0.7 watts, it can vary depending on usage).

2.4.2 ROS Support in Webots

A major part of the master project is the improvement of ROS2 support for Webots. Webots supports both, ROS1 and ROS2, but with certain limitations.

Webots ROS1 implementation automatically exposes all Webots API functions as ROS topics and services. Automatically exposing ROS API sounds as a reasonable solution, but it is uncanny for the ROS ecosystem, and therefore, much effort is required from users to adapt the exposed API to fit other ROS packages. For example, if one wants to publish odometry data, one needs to create a ROS node that subscribes to topics published by encoders and then publishes the corresponding odometry and transform messages. This work is time consuming, complex, and consumes unnecessary processing power by republishing all the messages.

Therefore, Cyberbotics has taken another approach in ROS2 support. Instead of creating ROS2 topics and services as done for ROS1 it provides facilities to users to design and implement their own ROS2 interface using Webots API functions. Creating a specific ROS2 driver simplifies usage,

[7]ROS driver that uses pi-puck extension is available at `https://github.com/gctronic/epuck_driver_cpp/tree/pi-puck`.

but it is still time-consuming as the ROS2 driver has to be created for each robot.

This project extends Webots' support for ROS2 by adding modules that can automatically create ROS2 interface, compatible with other ROS2 packages, based on a robot description. For the previously mentioned example, in which odometry has to be published, the proposed improvement will allow automatic publishing of odometry and transform messages, including support for ROS2 parameters and velocity control of the robots.

This improvement should allow users to integrate Webots simulations faster in their ROS2 applications, and it should effectively lead to a greater adoption of robot simulations among the ROS2 community that uses Webots.

Chapter 3 E-puck2 Simulation and ROS2 Interface

As previously explained, one of the project's objectives is to create a ROS2 interface for simulated e-puck2 robots. Even though creating ROS2 interface for Webots robots is automated later in the project, specific ROS2 driver for the e-puck2 in Webots is created first. For us, as developers, creating the specific ROS2 driver was necessary to understand better building blocks that can be generalized. To readers, this chapter will help better understand improvements brought by generalization (see Chapter 6). It will also show that some devices cannot be generalized, e.g., distance sensors cannot feed the `LaserScan` topic.

3.1 Introduction

Before going to implementation details, it is important to understand the concept of a controller in Webots and how it fits in ROS2 driver for Webots simulated robots. The Webots controller controls actuators and reads data from sensors available in a robot. The controller is a "brain" of the robot; it makes the robot move and senses the environment. It consists of two parts, the Webots API, which communicates with the simulation and user-defined code that uses Webots API to control the robot. The Webots API communicates with the Webots simulation through pipes and shared memory, a type of inter-process communication [13]. Therefore, there are two processes that run independently, Webots controller and Webots simulation (see Figure 3.1).

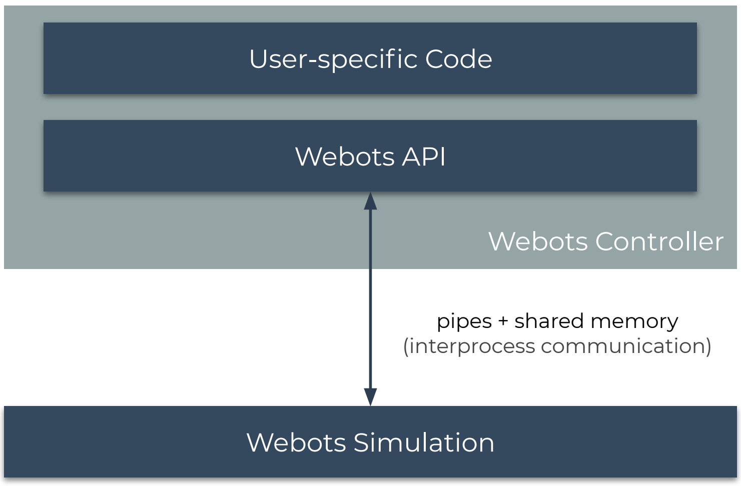


Figure 3.1: User specific code, API and simulation within Webots

In this chapter, a logic that performs translation from Webots API to ROS2 API will be implemented as a Webots controller (block *User-specific Code* in Figure 3.1). Therefore, a block *Webots Controller* from the figure will be referenced as ROS2 driver in the further text.

3.2 Webots within ROS2

As the ROS2 driver is defined, we should clarify how the ROS2 driver and Webots simulation can be launched within a ROS2 application. The straightforward way to launch the ROS2 driver and Webots simulation is the following:

- put ROS2 libraries to the environment variable `PATH`,
- start the Webots simulation,
- execute the ROS2 driver and
- start the rest of the ROS2 application (ROS2 nodes).

However, to better integrate Webots into ROS2, launch files[1] are used. The launch files in ROS2 allow user to execute multiple processes (often ROS2 nodes) at once. The launch files are described with Python scripts, and there are three important concepts:

[1]Launch files in scope of ROS2 are poorly documented.Good documentation to understand the core concepts (although not completely accurate) can be found in ROS2 design specification at `https://design.ros2.org/articles/roslaunch.html`. A superficial explanation, hiding core concepts, on the usage of the launch files, is given in ROS2 tutorials at `https://index.ros.org/doc/ros2/Tutorials/Launch-system/`.

- Actions: They represent an intention to do something, like start a process (usually ROS2 nodes), set a parameter, or push a namespace.
- Substitutions: Define a transformable expression. It means that the expression contains a placeholder that can be replaced. For example, the substitution can be a path to a file in which filename is again a substitution:
  `PathJoinSubstitution([package_path, LaunchConfiguration('world')])`
- Events: Actions can produce subscribable events. For example, when a process is closed, it will emit an event on which we can close the whole ROS2 application.

Using those three concepts a minimal launch file containing Webots and ROS2 driver is created (see Figure 3.2).

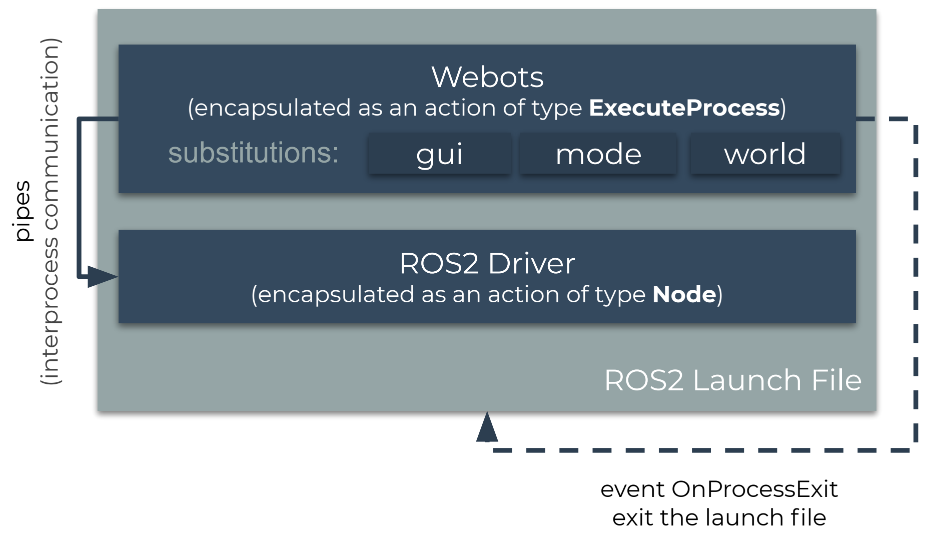

pipes
(interprocess communication)
Webots
(encapsulated as an action of type **ExecuteProcess**)
substitutions:
gui
mode
world
ROS2 Driver
(encapsulated as an action of type **Node**)
ROS2 Launch File
event OnProcessExit
exit the launch file

Figure 3.2: Webots and ROS2 driver within the launch file

It will start Webots with three substitutions, `gui`, `mode` and `world` which are used to configure Webots. These substitutions are taken from arguments (as `LaunchConfiguration`). Also it defines an action which will kill the launch file if the user exit the simulation.

This whole implementation is later completely hidden from the user (introduced in Chapter 6).

3.3 Covered Sensors and Actuators

In this section, the implementation of ROS2 driver for the simulated e-puck2 robot is explained. Please note that only the first section (Section

3.3.1) will provide the comprehensive explanation of the respective topic. The other sections will only briefly cover the implementation details as the ROS2 support does not differ significantly from sensor to sensor.

3.3.1 Differential Drive

Using data from sensors such as wheel encoders, camera, or IMU, or fusing them, one can estimate the change in robot's position overtime [14, 15]. With the dead reckoning method, the change in position can be accumulated, and, in that way, the robot's position in the local frame (frame relative to the robot's start position) can be estimated [16, 17]. In ROS2, each sensor used for odometry should publish messages of type `nav_msgs/Odometry`, and messages from different sensors later can be fused to increase accuracy.

In the scope of this project, only wheel encoders are used for odometry. Even though the e-puck2 does not have encoders but step motors, we control steps precisely, and therefore, that information we can use to calculate odometry.

```
nav_msgs/Odometry
├── (std_msgs/Header) header
├── (string) child_frame_id
├── (geometry_msgs/PoseWithCovariance) pose
│   ├── (geometry_msgs/Pose) pose
│   │   ├── (geometry_msgs/Point) position
│   │   └── (geometry_msgs/Quaternion) orientation
│   └── (float64[36]) covariance
└── (geometry_msgs/TwistWithCovariance) twist
    ├── (geometry_msgs/Twist) twist
    │   ├── (geometry_msgs/Vector3) linear
    │   └── (geometry_msgs/Vector3) angular
    └── (float64[36]) covariance
```

Figure 3.3: `nav_msgs/Odometry` message type definition in ROS2

In Figure 3.3, odometry format proposed by ROS2 and used by the community packages is given. It requires `geometry_msgs/Pose` (position and orientation) of the robot to be specified, as well as `geometry_msgs/Twist` (linear and angular velocity).

First we express velocity of left (v_{left}) and right (v_{right}) wheel by multiplying wheel radius (R) with angular velocity:

$$\begin{aligned} v_{left} &= R\frac{\gamma_{left}(n) - \gamma_{left}(n-1)}{\Delta t} \\ v_{right} &= R\frac{\gamma_{right}(n) - \gamma_{right}(n-1)}{\Delta t} \end{aligned} \tag{3.1}$$

in which $\gamma_{left}(n)$ and $\gamma_{right}(n)$ are angular positions of left and right wheel respectively at the sample n.

For differential drive robots we can simply express linear (v) and angular (ω) velocity as:

$$\begin{aligned} v &= \frac{v_{left} + v_{right}}{2} \\ \omega &= \frac{v_{right} - v_{left}}{L} \end{aligned} \tag{3.2}$$

where L is axle length (distance between the left and the right wheel). The velocity of the robot in odometry frame is given by:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ \omega \end{bmatrix} \tag{3.3}$$

Knowing the angular and linear velocity, we can integrate it to obtain a position.

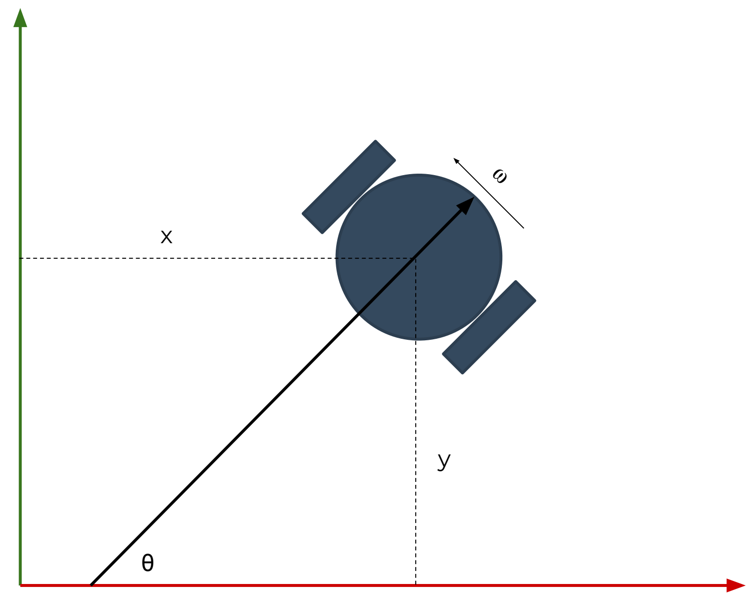


Figure 3.4: Robot in local frame [3]

As it is a non-linear system of differential equations we can integrate it using a numeric integration. It can simply be integrated using Euler method expressed in general terms as:

$$y(t+h) \approx y(t) + hy'(t) \tag{3.4}$$

This method would be computationally cheap, but not as accurate as for example fourth order of Runge-Kutta [18, p. 40]. The choice of the method for the numerical integration is mainly based on sample rate we perform with the pi-puck extension. In the case of e-puck2 physical robot, communication with the on-board MCU is configured exchange data with pi-puck extension at around 20Hz. Considering the slow sample rate and the computational power of the Raspberry Pi Zero W to handle floating-point operations, it is reasonable to choose the fourth order of Runge-Kutta over Euler method and trade computation resources for better accuracy. Therefore, we express the numerical integration as follows:

$$
\begin{aligned}
k_{00} &= v\cos\theta_{n-1}\\
k_{01} &= v\sin\theta_{n-1}\\
k_{02} &= \omega\\
\\
k_{10} &= v\cos\theta_{n-1} + \frac{t}{2}k_{02}\\
k_{11} &= v\sin\theta_{n-1} + \frac{t}{2}k_{02}\\
k_{12} &= \omega\\
\\
k_{20} &= v\cos\theta_{n-1} + \frac{t}{2}k_{12}\\
k_{21} &= v\sin\theta_{n-1} + \frac{t}{2}k_{12}\\
k_{22} &= \omega\\
\\
k_{30} &= v\cos\theta_{n-1} + tk_{22}\\
k_{31} &= v\sin\theta_{n-1} + tk_{22}\\
k_{32} &= \omega
\end{aligned}
\tag{3.5}
$$

$$
\begin{bmatrix} x_n \\ y_n \\ \theta_n \end{bmatrix} = \begin{bmatrix} x_{n-1} \\ y_{n-1} \\ \theta_{n-1} \end{bmatrix} + \frac{t}{6}\begin{bmatrix} k_{00} + 2(k_{10} + k_{20}) + k_{30} \\ k_{01} + 2(k_{11} + k_{21}) + k_{31} \\ k_{02} + 2(k_{12} + k_{22}) + k_{32} \end{bmatrix} \tag{3.6}
$$

Notice in Figure 3.3 that the orientation is represented as a quaternion while our orientation is represented in Euler angles. To convert the Euler angles to quaternions we reference to [19, p. 12]:

$$
\boldsymbol{q}(\alpha_x, \alpha_y, \theta) = \begin{bmatrix} \cos\frac{\alpha_x}{2}\cos\frac{\alpha_y}{2}\cos\frac{\theta}{2} + \sin\frac{\alpha_x}{2}\sin\frac{\alpha_y}{2}\sin\frac{\theta}{2} \\ -\cos\frac{\alpha_x}{2}\sin\frac{\alpha_y}{2}\sin\frac{\theta}{2} + \cos\frac{\alpha_y}{2}\cos\frac{\alpha_y}{2}\sin\frac{\theta}{2} \\ \cos\frac{\alpha_x}{2}\cos\frac{\theta}{2}\cos\frac{\alpha_y}{2} + \sin\frac{\alpha_x}{2}\cos\frac{\alpha_y}{2}\sin\frac{\theta}{2} \\ \cos\frac{\alpha_x}{2}\cos\frac{\alpha_y}{2}\sin\frac{\theta}{2} - \sin\frac{\alpha_x}{2}\sin\frac{\theta}{2}\sin\frac{\alpha_y}{2} \end{bmatrix} \tag{3.7}
$$

in which we can neglect α_x and α_y as those two elements are always 0 for differentially-wheeled robots.

At this point, the obtained $x, y, \boldsymbol{q}, \dot{x}, \dot{y}$ and $\dot{\theta}$ are packed in `nav_msgs/Odometry` (see Figure 3.3) and published periodically at 20Hz.

With `nav_msgs/Odometry` messages the rest of the ROS2 is aware of robot's odometry data. However, in addition to odometry data, the odometry frame has to be defined as well to explain the robot's position with respect to the odometry frame. For that purpose ROS2 defines transform messages of type `geometry_msgs/TransformStamped` (see Figure 3.5). In short, transform messages are used to create a transform tree to keep track of multiple coordinate frames over time. Keeping track of the coordinate frames is an essential aspect of ROS in general, and it will be properly explained in Chapter 6 in which it will be extensively utilized.

```
geometry_msgs/TransformStamped
├── (std_msgs/Header) header
├── (string) child_frame_id
└── (geometry_msgs/Transform) transform
    ├── (geometry_msgs/Vector3) translation
    └── (geometry_msgs/Quaternion) rotation
```

Figure 3.5: `geometry_msgs/TransformStamped` message type definition in ROS2

However, to control the robot's velocity a message of type `geometry_msgs/Twist` (see Figure 3.6) has to be utilized. Therefore, the node has to subscribe to the topic and set the wheels' angular speed accordingly.

```
geometry_msgs/Twist
├── (std_msgs/Header) header
├── (geometry_msgs/Vector3) linear
└── (geometry_msgs/Vector3) angular
```

Figure 3.6: `geometry_msgs/Twist` message type definition in ROS2

The target velocity of the left (v_{left}) and right (v_{right}) can be obtained as:

$$
\begin{aligned}
v_{left} &= v_{ref} + L\frac{\omega_{ref}}{2} \\
v_{right} &= v_{ref} - L\frac{\omega_{ref}}{2}
\end{aligned}
\tag{3.8}
$$

where v_{ref} is a reference linear velocity (available in the ROS2 message `.linear.x`) and ω_{ref} angular reference velocity (available in the ROS2 message `.angular.z`). Webots expect the velocity to be given in radians per second (rad/s), therefore:

$$\begin{aligned} \omega_{left} &= \frac{v_{left}}{R} \\ \omega_{right} &= \frac{v_{ref}}{R} \end{aligned} \tag{3.9}$$

Within this section, a minimal implementation of velocity control and odometry is given. It allows the e-puck2 to use the standard ROS2 interface to receive velocity control commands and to publish it's position and other relevant information within the odometry frame. Messages of type `geometry_msgs/TransformStamped` are published to topic name `/tf`, messages of type `nav_msgs/Odometry` are published to a topic name `/odom` and messages of type `geometry_msgs/Twist` are received from topic name `/cmd_vel`. Those topic names follow ROS2 conventions for topic naming and can be changed using ROS2 remapping[2].

3.3.2 Distance Sensors

E-puck2 is equipped with 8 infra-red sensors and one ToF sensor (see Figure 3.7). These sensors are modeled as `DistanceSensor`[3] in Webots.

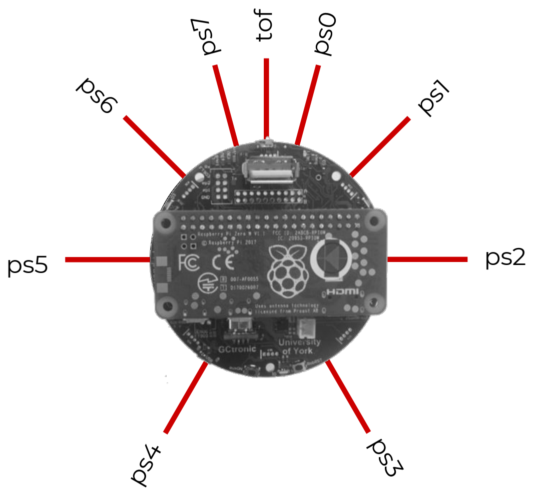


Figure 3.7: Distance sensors available on e-puck2

All details about the sensors are obtained from the Webots and published to a topic of type `sensor_msgs/Range`[4].

[2]Tutorial on ROS2 remapping can be found at `https://index.ros.org/doc/ros2/Tutorials/Node-arguments/#id1`.

[3]More information about `DistanceSensor` nodes is available `https://cyberbotics.com/doc/reference/distancesensor`.

[4]Definition of `sensor_msgs/Range` message type is available at `https://github.com/`

Laser Scanner

ROS2 defines `sensor_msgs/LaserScan`[5] for LiDARs and other types of planar laser range-finders. Those messages are commonly used by ROS2 community packages like `navigation2` and `slam_toolbox`. The message type requires an array of measurements at angles that are equally distanced from each other.

Therefore, even though there is no LiDAR available on the e-puck2, it is possible to emulate it using available distance sensors. Since the distance sensors are not evenly distributed, virtual distance sensors are added to fill space between the actual distance sensors (see Figure 3.8). These virtual distance sensors always give measurements of 0 meters which corresponds to invalid measurement (as minimum valid range in `sensor_msgs/LaserScan` message is defined to be greater than 0 meters).

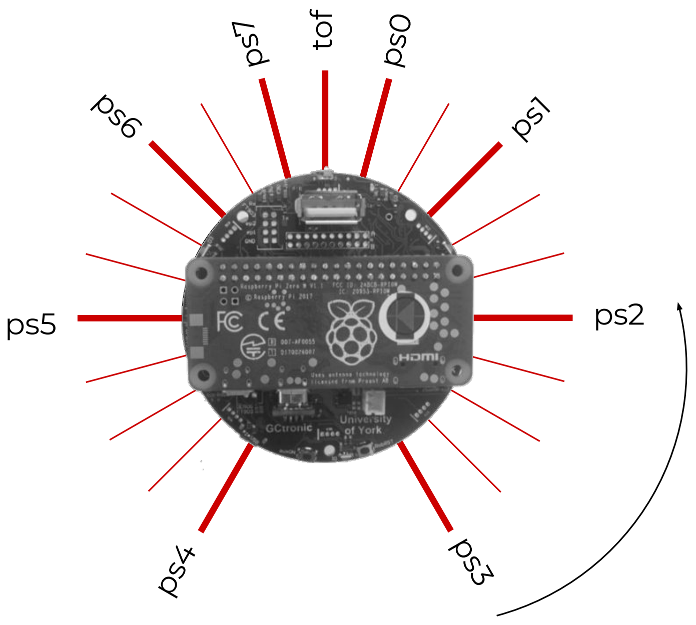


Figure 3.8: Virtual distance sensors are added to emulate LiDAR, making the angle difference between the rays constant (15°)

3.3.3 Light Sensors

E-puck2 has eight infra-red sensors which, besides proximity, can measure light intensity as well. ROS2 interface for light sensors uses messages of type

`ros2/common_interfaces/blob/master/sensor_msgs/msg/Range.msg`.

[5]Definition of `sensor_msgs/LaserScan` message type is available at `https://github.com/ros2/common_interfaces/blob/master/sensor_msgs/msg/LaserScan.msg`.

`sensor_msgs/Illuminance`[6]. The message type requires the measurements to be in Lux units (illuminance) while measurements acquired in Webots[7] are expressed in watts per square meter $[W/m^2]$ (irradiance). The conversion is done according to [20].

3.3.4 Inertial Measurement Unit

IMU in Webots modeled as two nodes, `Accelerometer` and `Gyro`. Measurements from those two nodes are combined and packed into messages of type `sensor_msgs/Imu`.

3.3.5 Camera

Typically, cameras in ROS2 use two topics, one for data (images) and another one for intrinsic camera parameters. Images from Webots camera node are sampled, packed in ROS2 messages of type `sensor_msgs/Image` and published repeatedly. The intrinsic parameters are defined by ROS2 has the following form:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

Webots doesn't provide such a matrix, but it is straightforward to create one from the existing parameters. Since Webots camera doesn't provide distortions that move the focal length then $f_x = f_y$ which is equal to the focal length that can be obtained in Webots (e.g. `getFocalLength()`). The principal point also doesn't have offset, but it is in the center of the image c_x is equal to $\frac{\texttt{image\_width}}{2}$ and c_y is equal to $\frac{\texttt{image\_height}}{2}$.

3.3.6 LEDs

There are eight LEDs available in the robot and they are controlled with messages of type `std_msgs/Int32`. The last three bytes of the value are used to set three RGB components in case of RGB LEDs and for the regular LEDs the value is used to set the intensity. Arguably here, message type `std_msgs/ColorRGBA` may be more suitable, but `std_msgs/Int32` is chosen to be more consistent with Webots API.

[6]Definition of `sensor_msgs/Illuminance` message type is available at `https://github.com/ros2/common_interfaces/blob/master/sensor_msgs/msg/Illuminance.msg`.

[7]Light sensors in Webots are represented as `LightSensor` node - `https://cyberbotics.com/doc/reference/lightsensor`.

3.3.7 Final Interface

In the table bellow (see Table 3.1) the final interface is shown.

Topic name	**Message type**	**Description**
`/cmd_vel`	`geometry_msgs/Twist`	Controls robot's velocity
`/odom`	`nav_msgs/Odometry`	Odometry measurements from wheels
`/ps[0-7]`	`sensor_msgs/Range`	Measurements from infra-red sensors
`/tof`	`sensor_msgs/Range`	Measurements from ToF sensor
`/scan`	`sensor_msgs/LaserScan`	Emulated LiDAR measurements
`/ls[0-7]`	`sensor_msgs/Illuminance`	Light measurements from infra-red sensors
`/imu`	`sensor_msgs/Imu`	Measurements from IMU
`/led[0-7]`	`std_msgs/Int32`	Controls LEDs
`/gs[0-2]`	`sensor_msgs/Range`	Measurements from ground sensors
`/image_raw`	`sensor_msgs/Image`	Camera images
`/camera_info`	`sensor_msgs/CameraInfo`	Camera intrinsic parameters
`/tf`	`tf2_msgs/TFMessage`	Dynamic transforms
`/tf_static`	`tf2_msgs/TFMessage`	Static transforms

Table 3.1: Complete ROS2 interface for e-puck2 robot

In addition to the previously mentioned topics, there are topics with name `/gs[0-2]`, and those topics publish data from ground sensors. The ground sensors can be bought as a separate e-puck2 module. Therefore, this part of the ROS2 interface will be automatically created if the module is present in the e-puck.

Another topic not mentioned before is `/tf_static`. It is used to describe transformations between different coordinate frames that do not typically change (for example, a transformation between the robot's base and the LiDAR). Messages published to this topic have specific QoS configured, durability is set to be transient local. The transient local QoS means that the nodes joined to the system will receive the messages even though the messages are published much earlier and avoiding periodic publishing reduces the load on the ROS2 driver as the messages have to be published only once.

Another performance improvement is made by not publishing the messages all the time. Messages are published only if the subscribers are available. This significantly improves performance, especially in the camera's case, as it is very CPU/GPU intensive task. For example, on the same computer and in the same Webots simulation, with camera of resolution 640x480, the simulation can run almost 2 times faster (1.81 times faster in e-puck2 default word). The difference can be much bigger for robots with multiple camera and many other sensors.

Chapter 4 ROS2 Interface for Physical E-puck2

Details about ROS2 interface implementation on the e-puck2 physical robot will be given in this chapter. Even though the objective is to create the same ROS2 interface as the one explained in the previous chapter, the implementation is very different. The difference mostly comes from the physical interface to the sensors and actuators, performance limitations, and CPU architecture. Therefore, these differences will be emphasized in this chapter.

4.1 Introduction

Pi-puck extension uses Inter-Integrated Circuit (I^2C) and Universal Serial Bus (USB) to communicate with sensors and actuators available on the e-puck2 robot (see Figure 4.1).

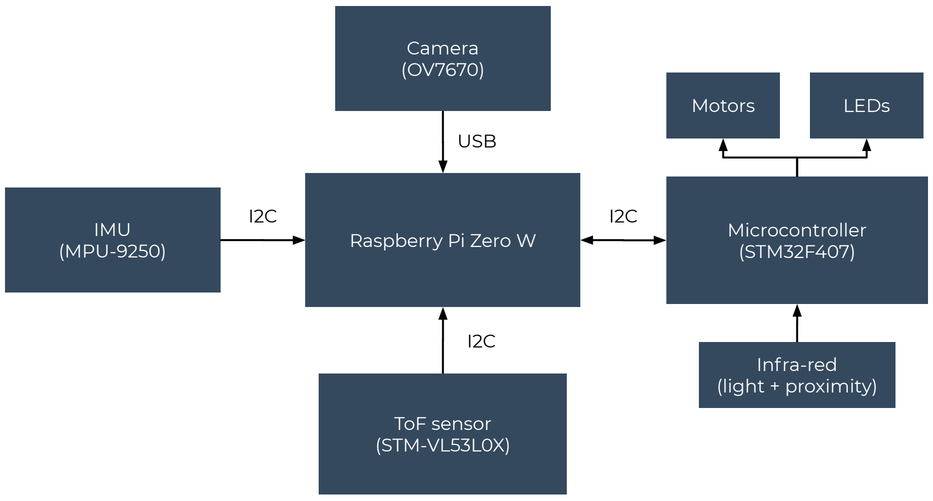


Figure 4.1: Components utilized in ROS2 driver

Therefore, different communication channels are adopted for various sensors and actuators. LEDs, motors, and infrared sensors are connected to the on-board MCU, allowing the Raspberry Pi Zero W to access the devices over I^{2}C. For the infrared sensors, this is necessary as Raspberry Pi Zero W does not have any Digital-to-Analog Converter (DAC) module. Therefore, the MCU acts as a slave which stands between analog sensors and actuators, or actuators that require deterministic updates (motors). With IMU and the ToF sensor, Raspberry Pi Zero W communicates directly over I^{2}C, while with the camera, the communication is done over USB.

Sensor function	**Sensor model**
Camera	Omnivision OV7670 CMOS[1]
Distance and light sensors	TCRT1000
IMU	InvenSense MPU-9250
ToF	STM-VL53L0X [21]

Table 4.1: List of the relevant sensors available on e-puck2 robot shown in Figure 4.1

The protocol used to communicate with the sensors is explained in the corresponding sensor's documentation, while the communication with the MCU is specified by the format shown in Table 4.2 and Table 4.3.

Left speed (2)	Right speed (2)	Speaker (1)	LED[1,3,5,7] (1)
LED2 (3)	LED4 (3)	LED6 (3)	LED8 (3)
Settings (1)	**Checksum (1)**		

Table 4.2: Message format sent from the Raspberry Pi Zero W to the MCU. Bolded fields highlight the most and the least significant bytes in the packet.

8 x Prox (16)	8 x Ambient (16)	4 x Mic (8)	Selector + button (1)
Left steps (2)	Right steps (2)	TV remote (1)	**Checksum (1)**

Table 4.3: Message format sent from the MCU to the Raspberry Pi Zero W. Bolded fields highlight the most and the least significant bytes in the packet.

4.2 ROS2 on Raspberry Pi OS

Before the ROS2 driver is created, ROS2 has to be installed on the Raspberry Pi Zero W. The Raspberry Pi Zero W board contains CPU with Arm32 architecture and Raspberry Pi OS. The ROS2 does not officially support this configuration, and the standard installation procedure using OS' package manager doesn't work. The closest official support is a source compilation for Debian Buster placed as a tier 3 support[2]. This means that ROS2 has to be cross-compiled and that potential incompatibilities have to be manually resolved.

Since the ROS2 driver is intended for a wide range of users, the installation procedure has to be user-friendly. For that purpose, three installation procedures are created to fit different use cases, using preconfigured Secure Digital card (SD card), compilation on Raspberry Pi Zero W, and cross-compilation from the user's PC. A comparison of these methods is given by Table 4.4.

	Using image	**Compilation on the board**	**Cross-compilation**
Compilation speed	++	-	+
Easy to use	++	-	−
Flexibility	−	++	+

Table 4.4: Comparison of different installation methods provided in the scope of the project

Using the existing image with ROS2 and other tools configured is the most accessible approach for the users. The problem appears once the user has to upgrade the ROS2 version, install a new package, or to develop a custom package that has a lot of dependencies as compilation time is slow. For those use cases, cross-compilation tools are provided.

4.2.1 ROS2 Cross-compilation

In the scope of the project, tools are built to help with the process of ROS2 cross-compilation. All cross-compilation dependencies and configurations are packaged into a Docker container[3]. This means that the user does not need to worry about host OS and tools compatibility.

[2]ROS2 supported platforms are defined by REP 2000 available at `https://www.ros.org/reps/rep-2000.html#foxy-fitzroy-may-2020-may-2023`.

[3]An in-depth guide about the ROS2 cross-compilation is available at `https://github.com/cyberbotics/epuck_ros2/tree/master/installation/cross_compile`.

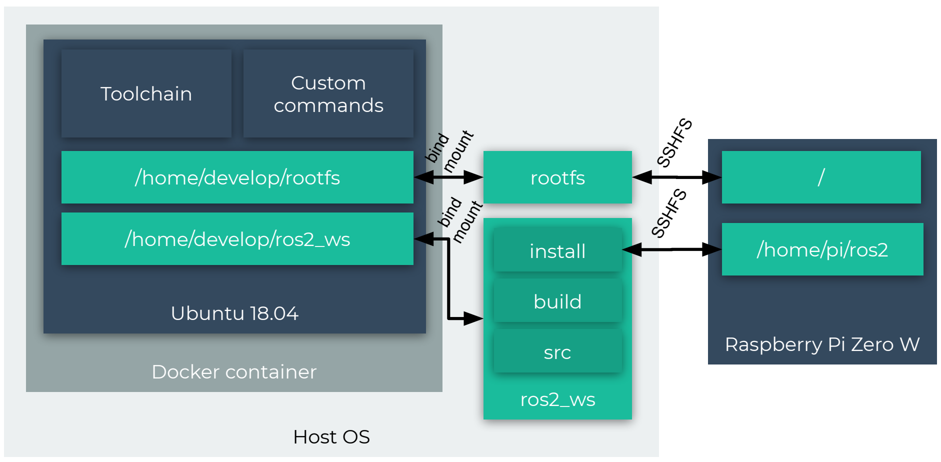


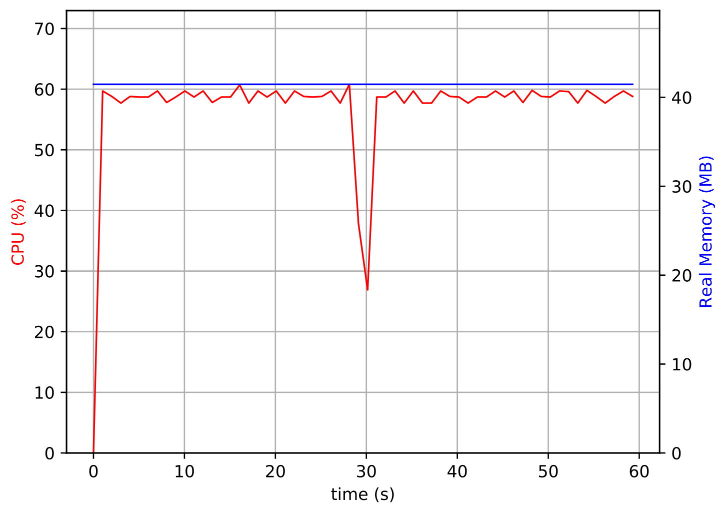
Figure 4.2: ROS2 cross-compilation setup

The typical setup of using the cross-compilation tools created in the scope of this project is shown in Figure 4.2. The boxes in green represent directories while the others are software tools. `ros2_ws` is the ROS2 workspace, it contains source files, temporary build files, and compiled output (libraries and executables). The source code available in this directory is compiled with cross-compilation tools in the Docker container while the result stays present on the host OS. Both directories are available in the Docker container as well as on the host OS.

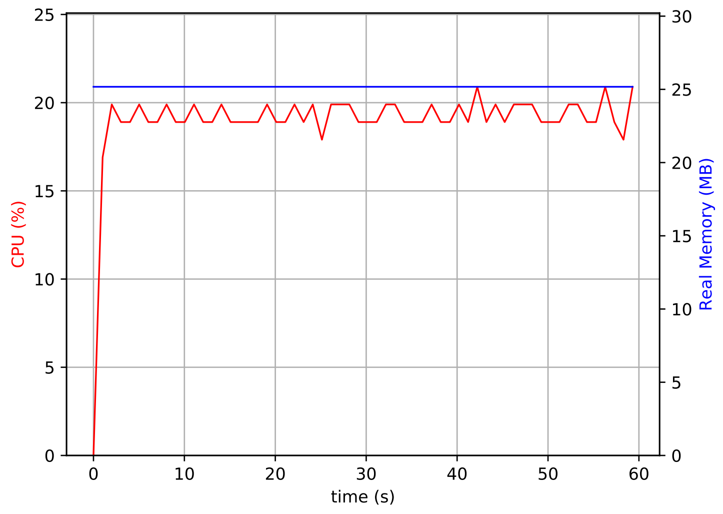
Once the ROS2 packages are compiled, they can be used by copying them into the `ros_ws/install` directory or by mounting them onto the Raspberry Pi Zero W. However, it is worth noting that mounting the directory can significantly increase productivity as copying the files is avoided.

4.3 Programming Language

Initially, a ROS2 driver with a few basic features for the physical e-puck2 is implemented in Python. However, as the ROS2 driver included more sensors, the CPU load was too high to handle other functionalities. Therefore, the implementation has been upgraded to C++ for the computational speedup.



(a) CPU and RAM load with Python



(b) CPU and RAM load with C++

Figure 4.3: Performance comparison of the similar node ROS2 node implemented in Python and C++. The CPU utilization and RAM usage are measured using `psrecord` tool[4]

Figure 4.3 shows performances of two ROS2 nodes with the same set of features, but implemented with different programming languages, Python

[4]The exact command to measure the load is `psrecord $(pgrep epuck2_driver) --interval 1 --plot plot.pdf --duration 60`

and C++. More precisely, they both publish odometry data, measurements from eight distance sensors (including the virtual laser scanner), and they are both subscribed to the velocity control topic. In the figure, we can observe that CPU usage is three times lower, while RAM is almost two times lower for C++. This analysis in the early stage of implementation steered the development towards C++[5].

4.4 Camera

As mentioned before, the camera available on e-puck2 (Omnivision OV7670 CMOS) captures images at 15 FPS in a resolution of 640x480. Compared to the simulation, data produced by the camera has to be efficiently processed, and intrinsic camera parameters have to be determined.

4.4.1 Camera Optimization

Our preliminary investigation showed that the ROS2 driver wasn't able to publish images more frequently than 3-4 FPS (a more thorough analysis is available in Chapter 7). Publishing raw (uncompressed) images would cause the network to become a bottleneck while compressing the images before transmitting would put a high CPU load, effectively limiting FPS. Fortunately, there is GPU available on Raspberry Pi Zero W on which specific image manipulation tasks can be offloaded.

Introduction

A brief overview on GPU on Raspberry Pi Zero W will be given first. The typical architecture of the GPU and the related components is given in the following figure (Figure 4.4):

[5]One of the bottlenecks was in the ROS2 client library for Python (`rclpy`) related to timer implementation - `https://github.com/ros2/rclpy/issues/520`. Fixing it improved the performances, but C++ implementation was still more efficient.

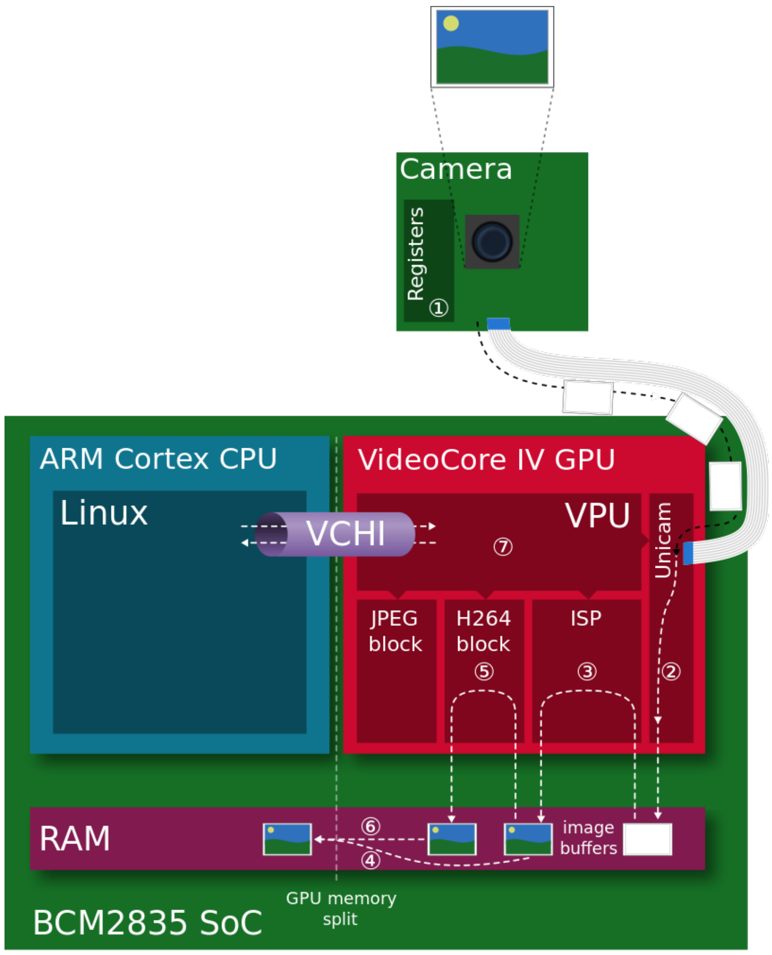


Figure 4.4: GPU architecture [4]

In the figure, there are three main components on the System On a Chip (SoC): CPU, GPU, and RAM. The GPU has its own computational components dedicated to different image processing tasks. These components are orchestrated by VideoCore Operating System (VCOS) (an abstraction layer on top of an Real-time Operating System (RTOS)), which also reads images from the camera. Signaling between CPU and GPU is done through VideoCore Host Interface (VCHI) while the images are shared by storing them on the RAM. From a software point of view, the Multi-Media Abstraction Layer (MMAL) library is used to interact with the GPU. The library is based on Open Media Acceleration (OpenMAX), and the goal is to ensure consistent interface across all GPUs available in different models of Raspberry Pi. For a comprehensive explanation about hardware blocks, please refer to [4], and for software [22], both are excellent sources of information.

ROS2 Camera Driver Implementation

There are a few available ROS1 and ROS2 packages that implement GPU accelerated image acquisition and processing. Unfortunately, these packages expect the camera to be connected via Camera Serial Interface (CSI), which is not the case in e-puck2. Therefore, the whole ROS2 camera node is implemented from scratch, including image acquisition based on V4L2 and image processing (compression and resizing) based on MMAL.

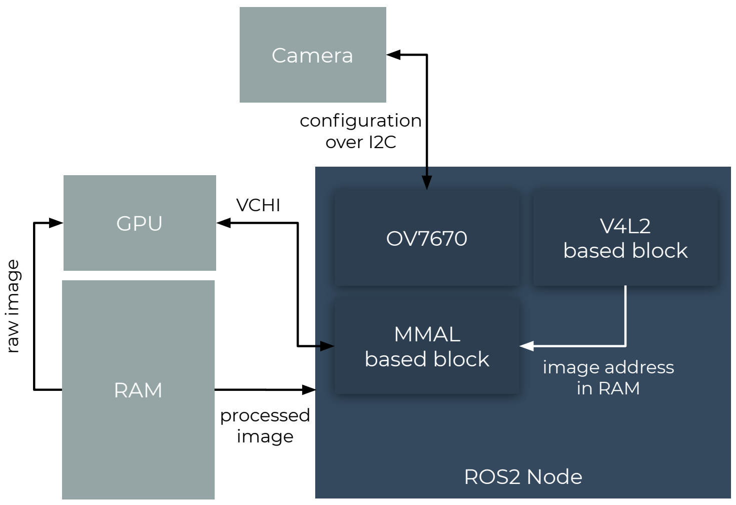


Figure 4.5: Camera software architecture

Figure 4.5 shows a software architecture of the camera node. Initially, the camera is configured over I^{2}C (block `OV7670`) and then V4L2 is used to initiate an image capture and store it to the RAM. The pointer to the image is passed through MMAL block, which interacts with the GPU to compress and resize it. Finally, the GPU stores the processed image to the RAM, and the pointer the processed image is available to the ROS2 camera node. The node packs it into the corresponding messages type and publishes it.

ROS2 Camera Driver Usage

Since the camera transmits compressed images through the network, they have to be uncompressed once the target computer is reached.

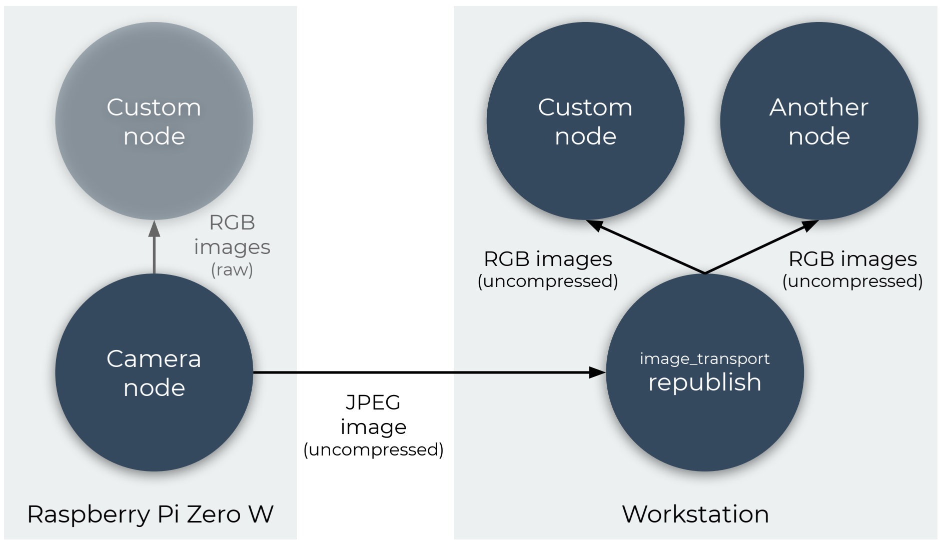

Custom node
RGB images (raw)
Camera node
JPEG image (uncompressed)
Raspberry Pi Zero W
Custom node
Another node
RGB images (uncompressed)
RGB images (uncompressed)
image_transport republish
Workstation

Figure 4.6: Image flow within ROS2 application

Figure 4.6 shows a typical flow of raw, compressed, and uncompressed images within ROS2 application. The general idea is to make a compromise between network utilization (less bandwidth is used if compressed images are transferred) and CPU load (it is less CPU intensive to avoid image compression). Therefore, the camera node advertises two topics, one with RGB (raw) images and the other with JPEG (compressed) images. A user can subscribe to any of those two, and the node will start publishing corresponding images to it. Usually, the optimal approach is to publish RGB images locally and JPEG images over the network. The RGB images are consumed normally, but JPEG images have to be received by an intermediate node (usually performed by ROS2 node called `image_transport/republish`), uncompressed and published locally. The `image_transport/republish` node supplies all nodes available on the workstation with uncompressed images. It is an especially convenient solution when there are many nodes on the workstation as the images are transferred only once through the network (nodes on the workstation consume images from the `image_transport/republish` node).

Camera Calibration

The intrinsic camera parameters of the Omnivision OV7670 camera (available on the e-puck2 physical robot) are not known. These parameters can be found through a camera calibration process. To perform the camera calibration, we have to capture multiple images of 3D points that have known positions in the real world. The camera projects the points to the camera frame (2D) that can be modeled as equidistance projection ($r = f\theta$, where f is focal length, r radial distance, and θ angle) and parametrized with a projection matrix (a matrix that contains intrinsic parameters). The 3D points from the real world and the corresponding 2D points projected on the camera frame represent a dataset used for the camera calibration. Then, we use nonlinear optimization to find the projection matrix's values that minimize a position difference of obtained and expected 2D points in the camera frame [23]. This process is automated by creating a custom ROS2 node that relies on OpenCV's calibration module.

4.4.2 Differential Drive

Besides the performance limitations of the Raspberry Pi Zero W that cause issues, there are other limitations as well. In the case of odometry, in Table 4.2, you can notice that the number of ticks for the left and the right wheel is limited to 2 bytes. Taking into account that it is a signed number (can take a value from -32767 to 32767) and that there are 1000 ticks per revolution, it can make around 32 revolutions before overflow (or $2\pi R \frac{32767}{1000} = 4.11m$). This means that the robot's odometry will become completely wrong after around 4 meters. To avoid this, the following simple approach is applied:

Algorithm 1: Overflow protection algorithm

```
  input : $N_{overflow}$ – Number of overflows (can be negative)
          $N_{grace}$ – Constant which represents the maximum
          difference in number of ticks between two sequential
          samples
          $P_{ticks}$ – Number of ticks in the previous sample
          $C_{ticks}$ – Number of ticks received from the sensor
  output: $T_{ticks}$ – Total (corrected) number of ticks
1 if $|P_{ticks} - C_{ticks}| > 2^{15} - N_{grace}$ then
2     if $P_{ticks} > 0 \land C_{ticks} < 0$ then
3         increment $N_{overflow}$;
4     else
5         decrement $N_{overflow}$;
6 $T_{ticks} \leftarrow 2^{16} N_{overflow} + C_{ticks}$;
```

By applying Algorithm 1, in T_{ticks} a total number of ticks will be stored, taking overflows into the consideration. It is important to set N_{grace} reasonably big such that $|P_{ticks} - C_{ticks}| > N_{grace}$ is only true when the overflow occurs.

4.5 Software Quality Assurance

As a part of ROSIN[6], this master project has to incorporate software engineering best practices such as continuous integration [24], unit testing and code reviews. Those practices are fully utilized throughout the whole project, and they will be briefly explained here.

4.5.1 ROS2 Node Unit Tests

The unit tests are implemented according to the standard procedure recommended by ROS2. Two ROS2 nodes are executed, one node is the actual node we want to test while the other simulates the rest of the application and runs tests.

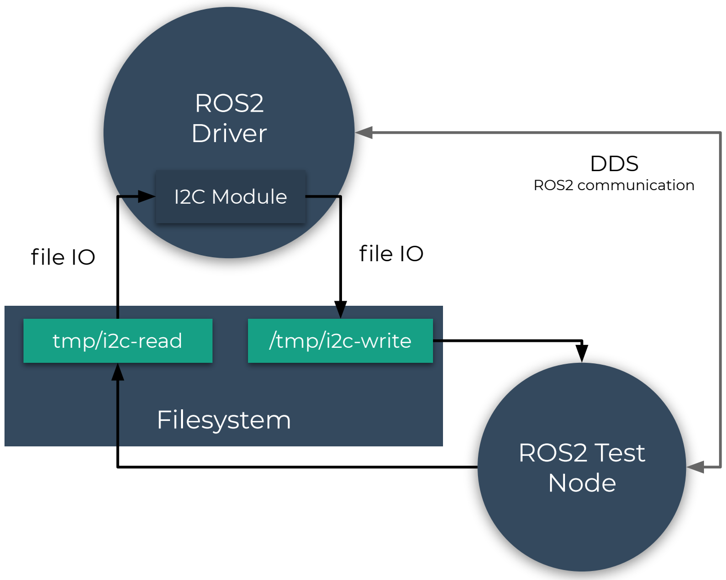


Figure 4.7: Testing communication with the microcontroller

A concrete example for testing of the ROS2 driver is given in Figure 4.7. In the figure, ROS2 test node is used to emulate the rest of the ROS2 application. For example, it can publish a message to `/cmd_vel` and set angular velocity. The ROS2 driver should receive the message and send the corresponding motor velocity through I^2C. To verify whether proper commands are sent over I^2C, a special I^2C module is created. It implements two modes; in the regular mode, it interacts with I^2C directly, while in the testing mode, it writes everything to the filesystem. This way, ROS2 test node has an opportunity to test commands addressed to the I^2C.

[6]The Software Quality Assurance propositions of the ROSIN project can be fount at `https://www.rosin-project.eu/software-quality-assurance`.

All unit tests are based on Python's standard unit testing framework, `unittest`, and ROS' `launch_test`.

4.5.2 Code Quality Tests

Besides the unit tests, many tests for static code analysis are used as well. The following tests are used to perform code quality analysis:

- `cppcheck` detects undefined behavior (e.g., dead pointers, division by zero or integer overflows) and security issues (e.g., buffer errors and information leaks) in C++ code.
- `cpplint` verifies whether the user follows C++ best practices and it detects syntax errors.
- `clang_format` recommends how the C++ code should be formatted and it fails if the code is not formatted properly.
- `lint_cmake` verifies whether CMake files follow the best practices.
- `flake8` enforces Python code to follow a style guide.
- `pep257` verifies if docstrings in Python code are properly formatted.
- `xmllint` verifies whether XML files follow the best practices.
- `copyright` checks if the copyright header is present in the files.

4.5.3 Continuous Integration

Unit tests, code quality tests, and more are executed in the scope of CI. It means that the source code is stored in Git repositories, and that a series of tests are executed every time a new change is pushed. In particular, every time a new commit is pushed, a Docker container is created, a ROS2 environment is configured, static code analysis is done, the packages are built, and unit tested. If any of these actions fail, the whole test is considered a failure, and a developer is forced to fix the error.

Chapter 5 E-puck2 Demos

Once a robot has running ROS2 driver, ROS2 nodes can be built on top of it, or community ROS2 nodes can be integrated. Therefore, in this chapter, various examples that utilize the e-puck2's ROS2 interface will be shown while focusing on custom-built nodes. All demos presented in this chapter work with both the physical and simulated e-puck2 robots. Some demos are also successfully tested with a few other simulated robots.

5.1 Visualizations

RViz2 acts as a ROS2 node, and it allows users to visualize the robot's state and perception of the environment in 3D. This tool is officially supported and developed by the ROS2 team; it is commonly used in ROS2 applications for visualizations and is extensively utilized throughout this project. Therefore, a few use cases will be given here.

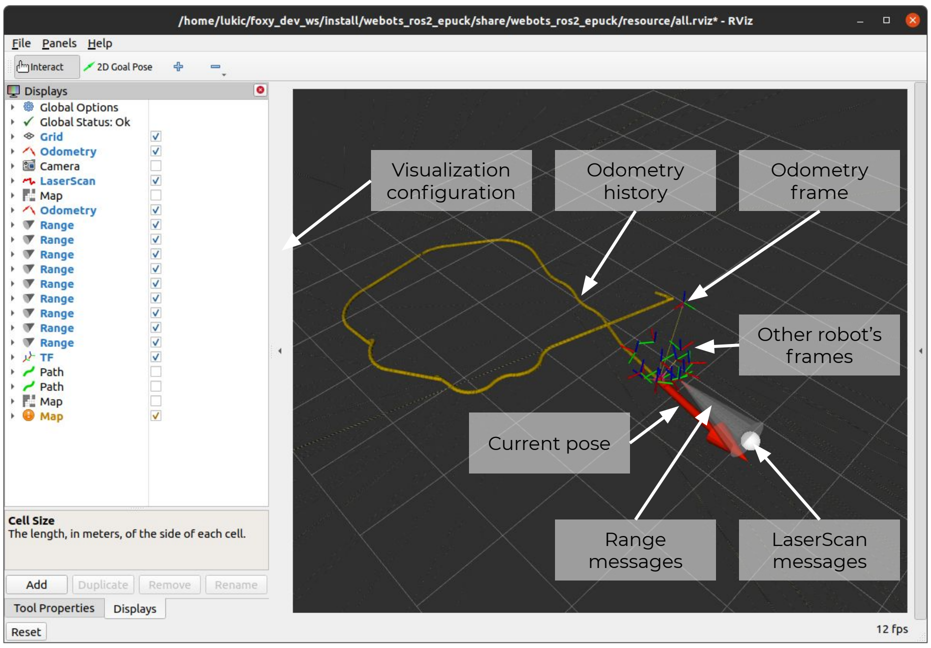


Figure 5.1: Typical visualization of e-puck2 in RViz2

RViz2 is very customizable, meaning it can visualize different aspects of the robot depending on the user's needs. Once the user is satisfied with the visualization, the view can be saved to a file for later reuse. In this master project, there are many RViz2 configurations provided, optimized for different scenarios like sensor inspection, mapping, and navigation. These configurations will be automatically loaded, depending on the launch file that is used.

In Figure 5.1 a default view of RViz2 for the e-puck2 robot is shown. It visualizes the robot's pose obtained from the odometry, history of odometry readings, different coordinate systems (like odometry, robot's base, distance sensors, and similar), range, and laser scan measurements.

5.2 Drive Calibration

Two constants are essential to have accurate odometry: wheelbase (distance between the contact points of the two wheels), and wheel radius. These constants can be measured, but even with the perfect measurements, they are subject to systematic odometry errors, caused by imperfections in the design and mechanical implementation of a mobile robot. Typical systematic error is uncertainty about the wheelbase and means that the rubber tires contact the floor not in one point, but rather in a contact area [25].

Therefore, we use `/odom` and `/cmd_vel` topics that are previously created to calibrate the two important constants for odometry. The technique is inspired by the one presented in [25]; the robot should move linearly, we can compare the anticipated distance (self-reported distance) with the actual distance. The linear movements allow us to adjust the wheel radius. For the wheelbase, the robot is rotated for a predefined number of rotations; then, it is compared to the actual number of rotations, and the wheelbase is adjusted accordingly (see Figure 5.2).

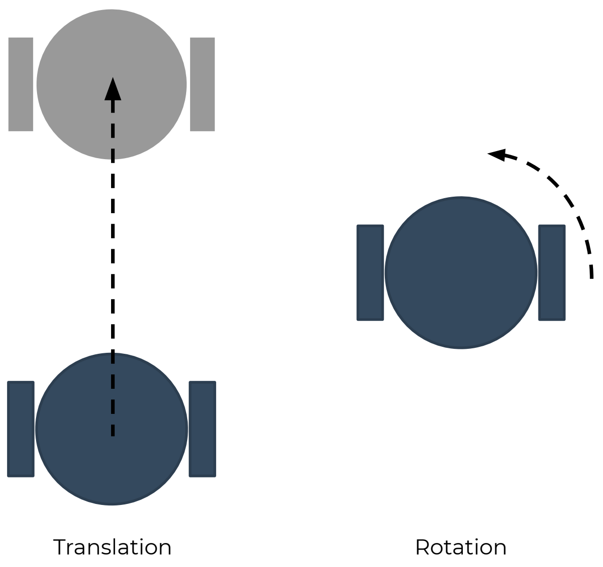


Figure 5.2: Differential drive robot calibration process

The custom-created node continuously receives the readings from the `/odom` topic to stop the robot exactly when the robot reaches the goal position. It also sets appropriate linear or angular velocity depending on the type of the test. Finally, the calibration node is successfully utilized for odometry calibration of the e-puck2 robot.

5.3 Custom Mapper Node

The goal of the next example is to utilize a larger portion of the created ROS2 interface. The goal is to map the environment using an e-puck2 robot and its distance sensors. A few approaches are considered to meet the goal:

- Utilize the existing SLAM solutions available for ROS2 (`slam_toolbox` or `cartographer`).
- Create a simple custom solution.

- Make a bridge to ROS1 workspace and use SLAM solutions available for ROS1 (e.g., `gmapping` is only available for ROS1).
- Port `gmapping` SLAM solution (that is available only for ROS1) to ROS2.

In tests, `slam_toolbox` and `cartographer` were not able to provide accurate mapping due to the scarcity of the distance measurements[1]. Using `gmapping` through ROS1 bridge indeed gave good results. However, complex installation and usage is something we tried to avoid as it is not user-friendly. Porting `gmapping` to ROS2 is feasible, but maintenance of a such complex software would be time intensive.

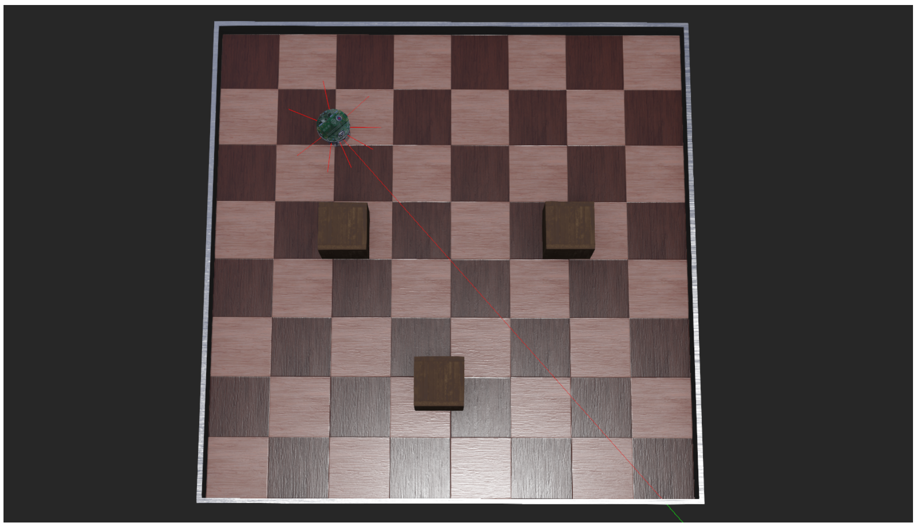
Finally, considering the scope of this master project and the high precision of e-puck2 odometry, the decision was to create a simple mapping node. The node uses odometry exclusively for localization. In general, this is not the right solution knowing that the odometry accumulates the error, but it provides satisfying results for small maps.

Algorithm 2: Mapping process

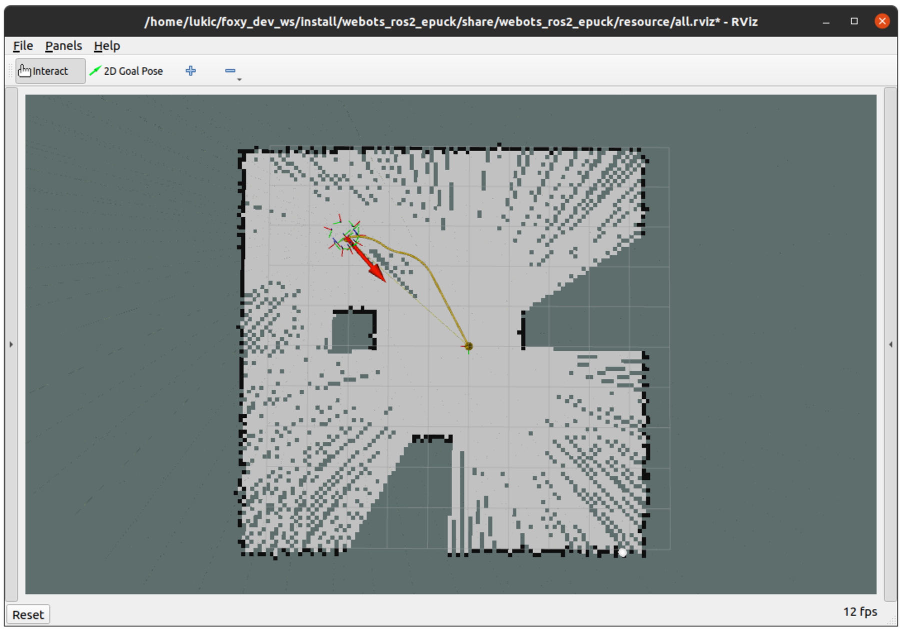
```
input  : $L_{scans}$ – Laser scan readings
         $P_{odom}$ – Position of the robot obtained from the odometry
output: $M_{map}$ – Map of type nav_msgs/OccupancyGrid
1 $O_{world} = [\ ]$ // List of coordinates of obstacles
2 use tf2 to get position of laser scanner $P_{scan}$;
3 for each $L_{scan}$ in $L_{scans}$ do
4     determine angle of ray ($\alpha$) from index;
5     $O_x \leftarrow P_{scan} + L_{scan} cos(\alpha)$;
6     $O_y \leftarrow P_{scan} + L_{scan} sin(\alpha)$;
7     add coordinates ($O_x$, $O_y$) to $O_{world}$;
8 write coordinates ($O_x$, $O_y$) to $M_{map}$;
9 use Bresenham's line algorithm to write empty space;
```

The algorithm (see Algorithm 2) uses a power of `tf2` package which listens for messages of `/TfMessage` to determine the position of the virtual laser scanner in respect to the odometry frame. With a few transformations, it is straightforward to calculate the position of the obstacles on the map. To fill the empty space between the robot and the obstacle (white pixels in Figure 5.3), Bresenham's line algorithm is implemented [25].

[1]The author of `slam_toolbox`, Steve Macenski, explained that modern graph-based SLAM solutions do not provide good results when used with e-puck2 robot (or similar robots with a few distance sensors) like old particle filter based SLAM solutions - `https://github.com/SteveMacenski/slam_toolbox/issues/192`.



(a) Webots simulation with e-puck2



/home/lukic/foxy_dev_ws/install/webots_ros2_epuck/share/webots_ros2_epuck/resource/all.rviz* - RViz
File
Panels
Help
Interact
2D Goal Pose
Reset
12 fps

(b) Map visualized in RViz2

Figure 5.3: Mapping results shown in RViz2

Finally, the result can be observed in the figure above (Figure 5.3).

5.4 Navigation Integration

Another example is navigation. For this demo, the `navigation2` community package is integrated.

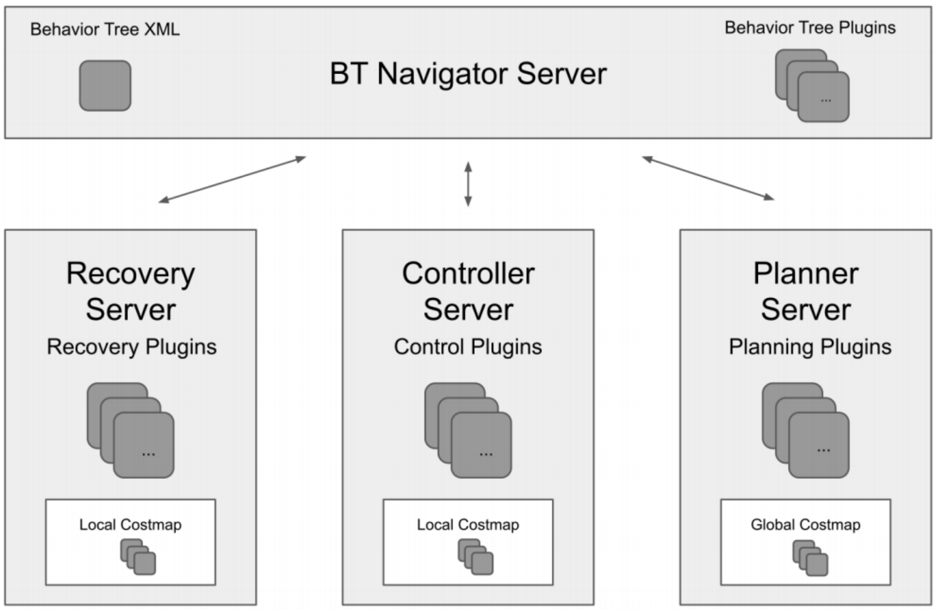


Figure 5.4: Architecture of the `navigation2` package[2]

It uses transforms, maps, and measurements from range finders to control the robot's velocity, effectively avoiding obstacles and converging towards the destination (see Figure 5.4). Based on the provided map, it builds a global cost map used by a planner to find a global path (e.g., using an A star algorithm) to the destination. Based on the data from the range finders, it builds a local cost map that is used by the controller server to avoid obstacles locally. The controller server is also used to follow the path generated by the planner server, and it usually uses an implementation called DWB local planner. The planner considers the robot's maximum rotational and linear velocity and various critics (e.g., goal align and path align), to issue velocity commands [26].

[2]The image is taken from the official `navigation2` GitHub repository available at `https://github.com/ros-planning/navigation2`.

Chapter 6 The Generalization of ROS2 Interface for Webots

6.1 Introduction

Up to now, the ROS2 interface for the e-puck2 robot in Webots has been created manually. It means that the Webots controller has to be written to expose ROS2 interfaces. Although this method gives a developer much flexibility, it is time demanding, prone to errors, and requires changes every time the Webots robot model is alternated. Since the Webots model is accessible from the Webots API, we saw an opportunity to automate the process of creating ROS2 driver for Webots.

The primary objective is to create a universal launch file that is supposed to read the Webots robot model and create ROS2 interface accordingly. This process has to be fully automated by default, highly configurable, and it has to work in conjunction with the user's custom code. The universal launch file allows users to bootstrap the project and benefit from reusable blocks quickly, furthermore, it also offers a possibility to configure the ROS2 interface and extend the driver if needed.

This system is supposed to bring a few major advantages for the users, the most important of which is the reduction of development time. Instead of writing a custom Webots controller, which exposes the ROS2 interface for each robot, the process of creating a ROS2 interface can be completely avoided, reducing the development time significantly. The second major advantage of the universal driver is that it is less prone to errors. Since the universal driver is supposed to be maintained by the Webots team and community, the bugs should be discovered and fixed quickly. However, the universal driver and launch introduce a level of abstraction, hiding implementation details and customization possibilities. Therefore, the building blocks must be carefully designed to allow a high level of customization if needed.

As mentioned, the universal driver needs access to the Webots robot model to generate a proper ROS2 interface. This will present multiple challenges: if a Webots robot is not configured to work as Supervisor, many aspects of Webots robot model are hidden. Therefore, the Webots controller API has to be extended to provide relevant endpoints that can be utilized for publishing of ROS2 transforms and resolving actual sensor readings. The other missing function in the Webots API is, for example, access to a lookup table of `DistanceSensor`. The access to the lookup table is needed because the `sensor_msgs/Range` topic requires real values to be published instead of raw values.

6.2 Design

In this section, more details on the design will be given. Nevertheless, we need to review the approach that has taken place up to now (see Figure 6.1).

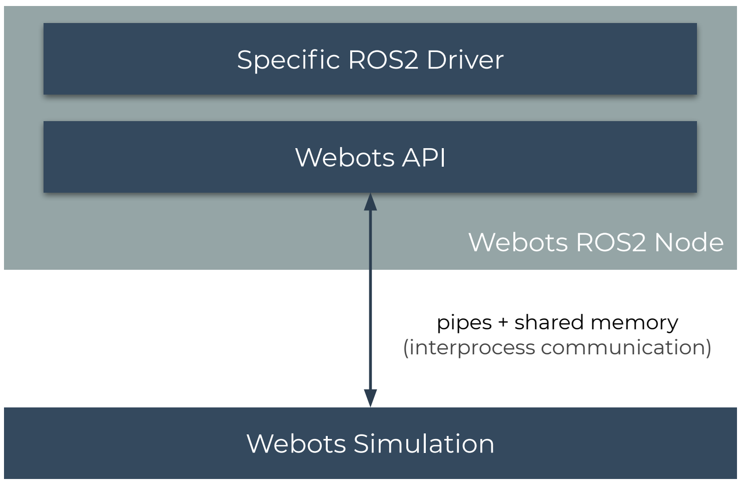


Figure 6.1: Technique of creating ROS2 interface within Webots before universal driver and launch file are introduced

The user had to write a layer that sits in between low-level API (Webots API in this case) and ROS2 interface. Although this is a necessary step for the real robots and allows users a lot of customization (or even the ability to optimize), the objective is to avoid the step for the reasons given in the previous section (see Section 6.1).

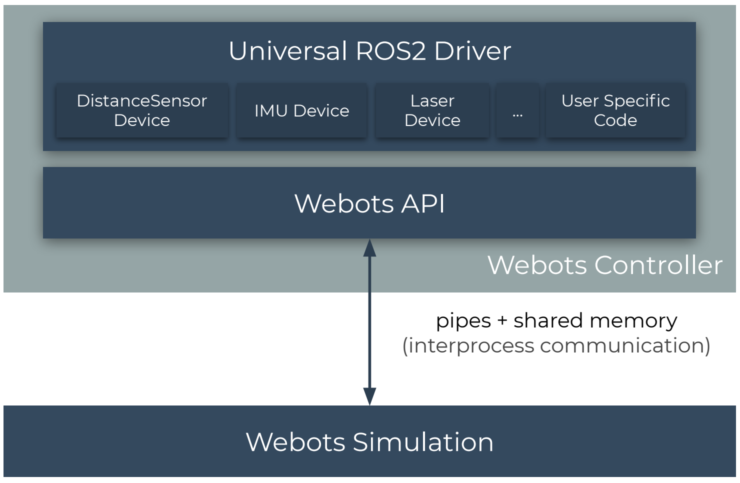


Figure 6.2: Webots and ROS2 interface using the new universal driver

With the universal driver, a notion of a device is introduced. A device[1], in this context, represents a module which transforms data from one or more Webots devices to one or more ROS2 topics and services. By default, the universal driver will go through all Webots devices available in a robot and try to match them with a suitable device. If the match is found, a new device is instantiated, a Webots device is assigned to it, and the device will start publishing or subscribing to the ROS2 messages.

All devices are configurable, meaning that they will use default parameters if custom parameters are not supplied. Since a device can be parameterized from multiple sources, the priority is the following:

- A parameter value obtained through ROS2 parameters has the highest priority, and it will override parameter values obtained through any other source.

- Python dictionary consisted of parameters passed to the device manager is the second in the priority list.

- Default or autogenerated values have the least priority. The autogenerated values are usually based on Webots' device names.

If a user is not satisfied with the customization achieved by using the parameters, the user can write custom code for it. A good example is distance sensors in e-puck2. The e-puck2 has nine distance sensors positioned around the robot that could emulate LiDAR and publish data to the `LaserScan` topic. Allowing something like this in the universal driver would be

[1]Note that “device” is related to a module in the universal driver and ”Webots device” is a node that represents a robot device in Webots.

too much work, and little gain due to the small number of users who would benefit from it. Therefore, users can disable the ROSification of Webots distance sensors and implement a custom code that will publish measurements to `LaserScan` topic[2].

6.3 ROS2 Transformations

In context of ROS2, transforms represent translation and rotation between two coordinate frames (see Figure 6.3). Keeping track of the transforms is important as it can provide a relative translation and rotation of two arbitrary coordinate frames in a transforms tree at any point in time [5]. ROS2 considers the transforms as a vital aspect of robotics applications. Therefore, many ROS2 packages rely on the ROS2 transformations. For example, the `slam_toolbox` uses the transforms to find translation and rotation between `base_link` and a link that publishes to a topic of type `LaserScan`. This is necessary as otherwise, the `slam_toolbox` could not determine the robot's position in the map.

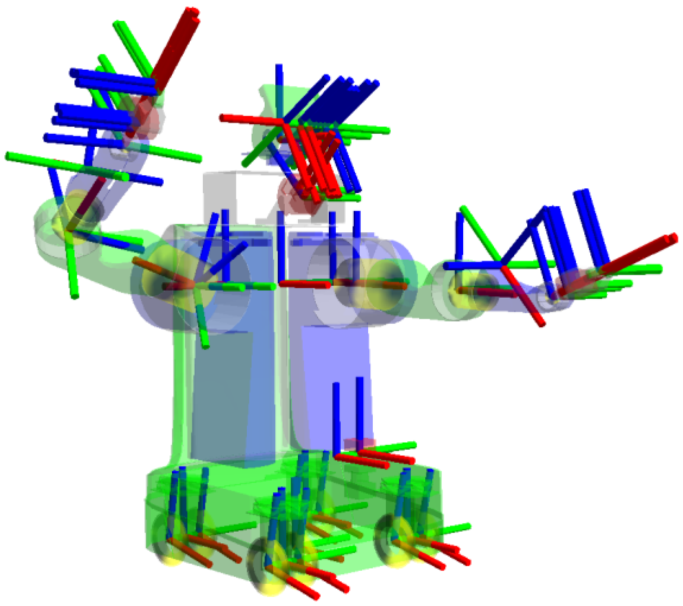


Figure 6.3: An example of coordinate frames visualized in RViz2 [5]

Considering the importance of transforms in ROS2, the universal driver has to offer functionality to publish those transforms automatically. To

[2]The described example is implemented, explained in more details and it is publicly available at `https://github.com/cyberbotics/webots_ros2/tree/master/webots_ros2_core#custom-launcher-file-and-driver`

publish transforms automatically, three approaches are considered, two of which are implemented in this project's scope:

- Method #1. The absolute position of each solid node is sampled periodically and published as a dynamic transform. The whole method can be implemented in the universal driver.
- Method #2. A new API function is added to Webots, which retrieves a tree of important links and joints. The tree is parsed in the universal driver, and relative transforms are published based on corresponding encoder readings (`PositionSensor` node in Webots).
- Method #3. A new API function is added to Webots which retrieves URDF as a string. URDF string is then passed to `robot_state_publisher` as ROS2 parameter. Furthermore, from the universal driver, encoder readings are published periodically as messages of type `sensor_msgs/JointState`. These messages are consumed by `robot_state_publisher`, and corresponding transforms are published.

The following table compares the most important aspects of these methods.

	Method #1	**Method #2**	**Method #3**
Preserves noise	No	Yes	Yes
Supervisor mode is necessary	Yes	No	No
Recognizes static transforms	No	Yes	Yes
Implemented in the project scope	No	Yes	Yes

Table 6.1: Comparison of different methods considered for publishing ROS2 transforms

As shown in Table 6.1, there are two significant reasons why Method #1 is not used: it does not preserve noise that is coming from the encoders, and the robot has to work in supervisor mode. Even though the implementation does not require changes to Webots core, a considerable disadvantage is a fact that it does not preserve a realistic behavior of the robot supported in Webots.

6.3.1 Transforms from URDF

This section describes Method #3, which is based on exporting an URDF document from Webots.

URDF vs. Webots' Robot Model Representation

However, it is important to compare URDF format and Webots' robot model representation first. URDF is an Extensible Markup Language (XML) format and it uses only two primitives to describe robot model, links and joints.

The URDF document has to contain at least one link, usually named as `base_link`, and its children can be only joints [27] (see Figure 6.4b). Therefore, a link cannot be encapsulated inside another link. Thus, a link can be connected to the other link with a common joint only. This is a main difference to Webots' robot representation as the root node in Webots encapsulates other nodes. In addition to it, Webots has richer specter of nodes. Besides the links (`Solid` node in Webots) and joints there are also nodes like `Group`, `Transform`, `Geometry`, and `Shape` (see Figure 6.4b).

```
Robot {
  children [
    Solid {
      children [
        Camera {
          name "camera"
        }
      ]
    }
  ]
}
```

(a) Webots' model representation

```
<?xml version="1.0"?>
<robot name="robot"
  xmlns:xacro="http://ros.org/wiki/xacro">
  <link name="base_link"/>
  <link name="solid"/>
  <link name="camera"/>
  <joint name="base_link_solid_joint"
    type="fixed">
    <parent link="base_link"/>
    <child link="solid"/>
    <origin xyz="0 0 0" rpy="0 0 0" />
  </joint>
  <joint name="solid_camera_joint"
    type="fixed">
    <parent link="solid"/>
    <child link="camera"/>
    <origin xyz="0 0 0" rpy="0 0 0" />
  </joint>
</robot>
```

(b) Model representation in URDF

Figure 6.4: Typical robot representation in URDF and in Webots of the same model

The described two main differences in a robot model, representation, hierarchy, and node diversity, make URDF export feature implementation rather complicated.

URDF Export

In this section, the main points of URDF export feature will be described. It is vital to know that this feature is implemented in Webots core utilizing the existing exporting robot models' mechanisms. The mechanism already supports a robot model to be exported to VRML, X3D and PROTO documents. Even though the mechanism provides useful features, flexibility to implement URDF export is limited, and therefore the implementation may differ from what is expected in general (nodes in URDF are related by ID, while nodes in Webots are encapsulated).

The Webots's mechanism to handle exports depends on a few methods declared in `WbNode`. In Webots, each node is derived from `WbNode` class and in the context of URDF export it has the following prototype:

```cpp
class WbNode {
protected:
  // Calls `.writeExport()` to continue the export if needed
  virtual void write(WbVrmlWriter &writer) const;

  // Uses `WbVrmlWriter` to write node details a document
  virtual void writeExport(WbVrmlWriter &writer) const;

  // Uses `WbVrmlWriter` to recursively continue export (export children)
  virtual void exportNodeSubNodes(WbVrmlWriter &writer) const;

  // Other declarations
};
```

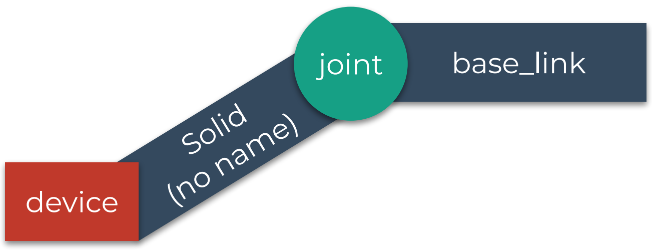
Those methods have to be defined for each Webots node to write URDF content using `WbVrmlWriter` class. Then, the methods are recursively called from the tree root (`Robot` node) to the leaves. The algorithm in Figure 6.5 defines a strategy to export nodes that are related by ID instead of encapsulating them.

```
Procedure write(this, writter)
  ;  /* $N_{current}$ and $N_{queue}$ are global variables.  $N_{current}$ represent a
     reference on instance of WbNode, while $N_{queue}$ is a queue of the
     references */
  if $N_{current} \neq$ this $\wedge$ this is not joint $\wedge$ this is not in $N_{queue}$ then
      add this to $N_{queue}$ ;
  if $N_{current}$ is not set then
      $N_{current}$ = this ;
  writeExport(this) ;
  if $N_{current}$ = this then
      if $N_{queue}$ not empty then
          $N_{current}$ = dequeue the last node from $N_{queue}$ ;
          write(this, writter) ;
      else
          $N_{current}$ = NULL ;
```

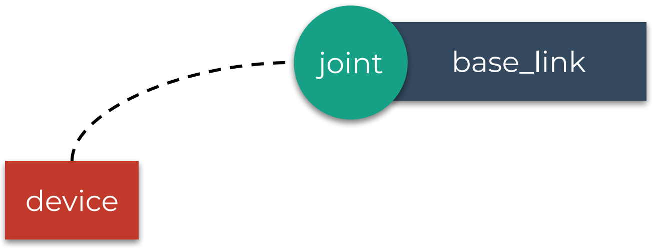
```
Procedure writeExport(this)
  if $N_{current}$ = this then
      write URDF of link ;
```

Figure 6.5: Approach used to handle URDF's flat nature in object-oriented architecture

Properly implementing `writeExport()` to Webots' nodes that represent joints and links will produce a desired URDF document. Although, the URDF document is accurate, it contains unnecessary links (see Figure 6.6a).



(a) Exported model



(b) Webots' model representation

Figure 6.6: Typical URDF and Webots robot representation

For example, in Figure 6.6a, there is Webots `Solid` node which got converted to URDF link tagged as `Solid (no name)`. Although this may be desirable in some cases, it usually makes ROS2 transform tree complex, which leads to unnecessary computations. In order to avoid the computations, a squashing is performed. It means that only relevant Webots nodes are exported into URDF (as link nodes) while the others are neglected. The relevant nodes are all nodes that have a `name` field defined. The field `name` is always present for devices, and users can explicitly define `name` fields if it should be exported as a URDF link.

As multiple intermediate Webots nodes can be neglected, the translation of child node (I_0) in respect to its parent node (I_N) is defined as:

$$\boldsymbol{T}_{I_0}^{I_N} = \boldsymbol{T}_{I_0} + \sum_{i=1}^{N-1} \boldsymbol{R}_{I_{i-1}}^{I_i} \boldsymbol{T}_{I_i}^{I_{i+1}} \tag{6.1}$$

whereas the translation of the intermediate node (I_m) in respect to its parent (I_n) is represented by $\boldsymbol{T}_{I_m}^{I_n}$ and consists of a three-dimensional vector.

In order to determine a rotation between two relevant links we use:

$$\boldsymbol{R}_{I_0}^{I_N} = \prod_{i=0}^{N-1} \boldsymbol{R}_{I_i}^{I_{i+1}} \tag{6.2}$$

Note that $\boldsymbol{R}_{I_0}^{I_N}$ is a 3×3 rotation matrix and that the matrix multiplication has to be started from the parent. In the implementation, a node recursively tries to find a relevant parent (visiting nodes towards the tree's root) while adding the visited nodes to the list. Once the relevant node is reached, it calculates the translation from the first element in the list.

The rotation matrix has to be converted to Euler angles around a fixed axis (extrinsic) roll-pitch-yaw to match URDF's specification. This is done according to [28, p. 9].

Finally, the interprocess communication (from Webots simulation to Webots API) is extended to support a new function:

```
const char *wb_robot_get_urdf(const char *prefix);
```

The API function is exposed to C++, Python, MATLAB, Java, and ROS client libraries.

Utilization of robot_state_publisher

Once the Webots' API is capable of exporting URDF as a string, it can be utilized in the ROS2 driver to publish ROS2 transforms. The approach to publish the transform is given in Figure 6.7.

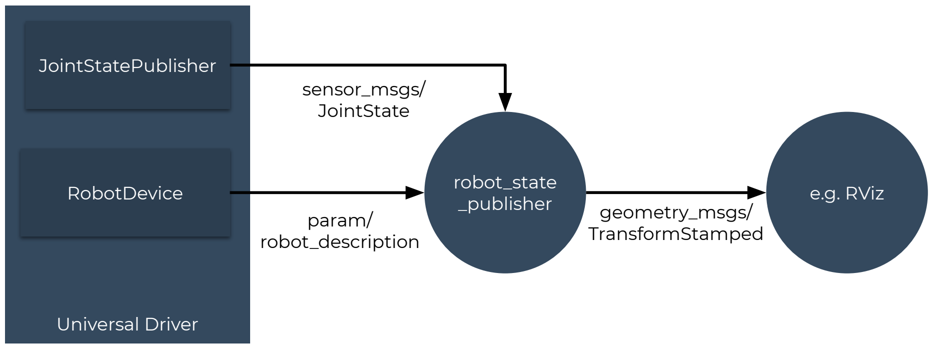


Figure 6.7: Publishing ROS2 transforms using URDF and `robot_state_publisher`

In the figure, the universal driver has to provide the parameter `robot_description` (URDF string) and a topic with messages of type `sensor_msgs/JointState` which contains readings from Webots `PositionSensors`. The parameter and the topic are consumed by `robot_state_publisher` (ROS2 community node) which publishes ROS2 transformations.

6.4 ROS2 Wrapped Devices

As mentioned before, the universal driver is supposed to create ROS2 automatically interface for each Webots device. In the scope of this master project, modules that perform the conversion from Webots to ROS2 interface are created for all Webots devices available on e-puck2, Khepera IV, and TurtleBot3 Burger. The code is designed to be easily expandable to other Webots devices as well.

In this section, a brief explanation will be given for a few Webots devices.

6.4.1 Differential Drive

The differential drive module interacts with two motors and two position sensors. The implementation is the same as explained in Section 3.3.1, but it is decoupled from the rest of the driver to work as an independent component. Additionally, it provides a high level of configurability (see table below, Table 6.2).

Name	**Description**
`left_encoder`	Name of position sensor mounted on the left wheel
`right_encoder`	Name of position sensor mounted on the right wheel
`left_joint`	Name of motor mounted on the left wheel
`right_joint`	Name of motor mounted on the right wheel
`wheel_distance`	Distance between left and right wheel in meters
`wheel_radius`	Radius of the wheels
`command_topic`	Topic name on which it should receive velocity commands
`odometry_topic`	Topic name to which it should publish odometry data
`odometry_frame`	Odometry frame name used in ROS2 transform messages
`robot_base_frame`	Robot base frame name used in ROS2 transform messages

Table 6.2: Parameters available for differential drive module

6.4.2 Range

Webots returns values from distance sensors according to specified lookup table[3].

```
lookupTable [ 0     1000  0,
              0.1   200   0.1 ]
```

It means it will not necessarily return a real distance to the closest object, but what Webots considers a raw value (linearly interpolated value based on the lookup table). Therefore, to obtain the actual distance, the lookup table has to be read. The values are then interpolated according to the table.

[3]In the example of lookup table first column represents the actual measurement, the second is the raw measurement, and the third column is the noise.

However, Webots API does not provide function to retrieve lookup table. Therefore, the function is implemented[4] similarly to the one explained in Section 6.3.1.

Then, in the universal controller, the actual values are returned, as shown by Algorithm 6.8.

```
Procedure interpolate($x_{value}$, $x_{start}$, $y_{start}$, $x_{end}$, $y_{end}$)
1 return $\frac{y_{end}-y_{start}}{x_{end}-x_{start}}(x_{value} - x_{start}) + y_{start}$ ;

Procedure interpolateTable($x_{value}$, $T$)
1 for $i \leftarrow 0$ to size of table $T$ - 1 do
2     if $(x_{value} T_{raw}[i] \wedge x_{value} \geq T_{raw}[i+1]) \vee (x_{value} > T_{raw}[i] \wedge x_{value} \leq T_{raw}[i+1])$ then
3         return interpolate($x_{value}$, $T_{raw}[i]$, $T_{actual}[i]$, $T_{raw}[i+1]$, $T_{actual}[i+1]$) ;
  ; /* Extrapolation, assumes the table is sorted in descending order */
4 if $x_{value} > T_{raw}[0]$ then
5     return interpolate($x_{value}$, $T_{raw}[0]$, $T_{actual}[0]$, $T_{raw}[1]$, $T_{actual}[1]$) ;
6 else
7     return interpolate($x_{value}$, $T_{raw}[-2]$, $T_{actual}[-2]$, $T_{raw}[-1]$, $T_{actual}[-1]$) ;
```

Figure 6.8: Procedure used to interpolate table

Once actual values are obtained, the values are packed into messages of type `sensor_msgs/Range` and published.

Name	**Description**
`topic_name`	ROS2 topic name
`timestep`	Publish period in ms
`disable`	Whether to create ROS2 interface for this sensor
`always_publish`	Publish even if there are no subscribers
`frame_id`	Value for `header.frame_id` field

Table 6.3: Parameters available for distance sensor device

[4]The API function to obtained lookup table is also added for `Accelerometer`, `Compass`, `Gyro`, `InertialUnit`, `LightSensor` and `TouchSensor`. All API functions are also added to C, C++, Python, MATLAB, Java and ROS.

Chapter 7 Results and Interpretation

This chapter analyzes whether the ROS2 driver for the e-puck2 physical robot and the ROS2 universal driver provide expected behavior. The analysis will be shown through three pillars:

- Verification whether the e-puck2 physical and simulated robots have a similar behavior with the same controller.
- Verification whether simulated e-puck2 and Khepera IV robots have a similar behavior with the same controller.
- Verification whether the ROS2 universal driver works properly with the other robots such as TIAGo++ and TurtleBot3 Burger.

7.1 Comparison of Physical and Simulated E-puck2

In this section, we want to verify whether the ROS2 interface is the same for the physical and simulated e-puck2 robots. The difference of the ROS2 interfaces will then be quantified with ROS2 navigation and mapping controllers.

7.1.1 ROS2 Interface Endpoints Comparison

The ROS2 interfaces for physical and simulated robots are very similar, image-related topics being the only difference. The physical e-puck2 robot advertises additional `/image_raw/compressed` topic of type `sensor_msgs/CompressedImage` and the robot uses it to transfer compressed images over the network.

7.1.2 Camera Performance Comparison

In this section, we want to compare different ROS2 camera node implementations on the physical e-puck2 robot, analyzing at which resolution and FPS it can operate. The analysis aims to determine a suitable way to transport the images from Raspberry Pi Zero W and identify bottlenecks.

In terms of compression, the images are transported as raw, JPEG compressed and as a Theora compressed video stream[1]. In terms of color encoding, RGB and YUV are tested. And in terms of nodes separation, the testing node was located on-board or on workstation while communicating to the Raspberry Pi Zero W over Wi-Fi.

In the Table 7.1, performances are measured in FPS and the measurements are given for different camera implementations.

	32x24 [FPS (σ)]	160x120 [FPS (σ)]	640x480 [FPS (σ)]
RAW over Wi-Fi	13.95 (0.009s)	10.08 (0.013s)	1.62 (0.096s)
RAW on-board	X	X	3.80 (0.064s)
JPEG over Wi-Fi	X	X	2.97 (0.105s)
JPEG over Wi-Fi with white-noise	X	X	0.95 (0.998s)
JPEG on-board	X	X	2.10 (0.016s)
RAW on-board without YUV42RGB	X	X	5.04 (0.037s)
Theora over Wi-Fi	X	X	1.25 (0.054s)
Theora on-board	X	X	1.03 (0.026s)
Custom over Wi-Fi	10.079 (0.073s)	9.981 (0.076s)	9.904 (0.079s)

Table 7.1: FPS measurements in different configurations within ROS2 environment

Please note that the experiments have been done under the following conditions:

- Package `v4l2_camera` is used to read and transport images. The package works as following:
  - the images are read directly from memory using `mmap()` in YUV422_-YUY2 format (native camera format),
  - the images are converted to RGB color encoding using `cv_bridge` package,
  - the images are transported using `image_transport`, `image_transport_plugins` (equipped with `compressed_image_transport` and `theora_image_transport`) with default configuration,
  - the package is alternated to accommodate image resize for this experiment and the image resizing is done just before YUV422_-YUY2 to RGB conversion,
  - the package is implemented in C++ with attention to memory management (the image is cloned only when necessary).

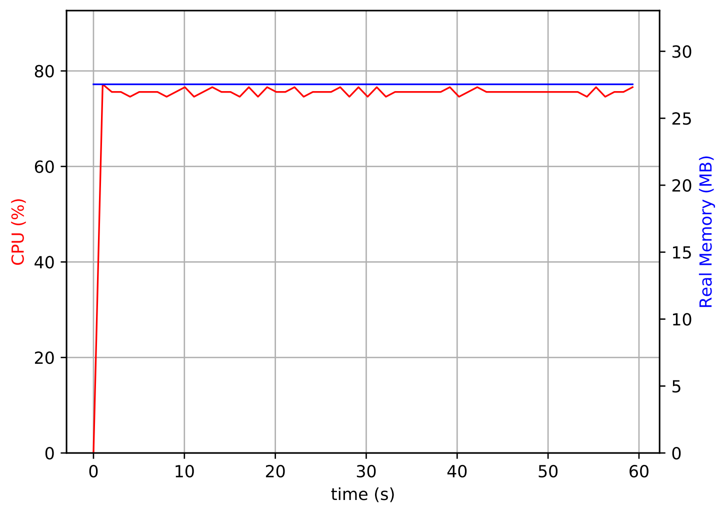
[1]Description of the Theora technology is available at the official website - `https://www.theora.org/`

- Camera is configured to 15 FPS.
- FPS measurements are done using `ros2 topic hz`.
- The Wi-Fi network performance measurements are performed using `iperf3` and the following results acquired:
  - 16.4 Mbits/sec for transfer from PC to Raspberry Pi Zero W and
  - 13.8 Mbits/sec for transfer from Raspberry Pi Zero W to PC.
- White noise is simulated by putting finger on the camera. The assumption is that the low light condition produces a lot of white noise.

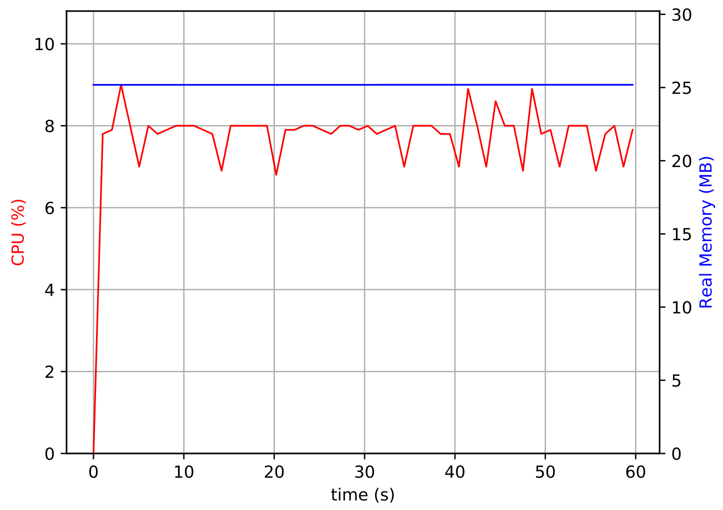
The measured data transfer between the Raspberry Pi Zero W and the PC during the publishing of the raw images is 12.8Mb/s. Since every image is sent in RGB format, that means 7Mbits per image ($8 \times \frac{3 \times 640 \times 480}{1024 \times 1024}$), or at 1.62 FPS, it is 11.4Mbits/s. Therefore, by sending raw images, we expected to encounter the limitations of the Wi-Fi network.

In all other methods that require compression (JPEG or Theora), the CPU on Raspberry Pi Zero W was hitting 100% of workload. Therefore, in those cases, the CPU is the bottleneck.

It may look strange that a slightly higher FPS is achieved with the images that are transferred over the network ("JPEG over Wi-Fi") than the images transferred locally ("JPEG on-board"). It is worth noting that the Raspberry Pi Zero W has a relatively slow CPU and that the tool used to measure performance has to allocate a certain amount of CPU time. Since the network is not the bottleneck here, but CPU, the effect of another process using the CPU is noticeable.



(a) CPU usage while publishing raw RGB images



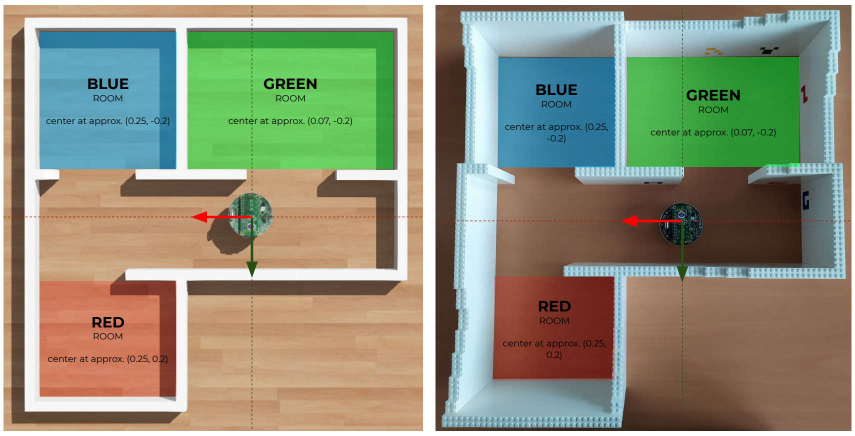
(b) CPU usage while publishing compressed (on the GPU) JPEG images

Figure 7.1: Comparison of CPU usage for two transfer modes, raw RGB (named as "RAW over Wi-Fi" in Table 7.1) and JPEG compressed on GPU (named as "Custom over Wi-Fi" in Table 7.1)

To improve JPEG compression performance and effectively increase the FPS, the compression is offloaded to GPU. Comparison of JPEG image compression performed on CPU and GPU is depicted in Figure 7.1. It shows that even though there is no compression involved, just image publishing, the images are too large for Raspberry Pi Zero W to be transmitted efficiently, and the process allocates a lot of CPU time. In contrast, the images compressed with GPU are the small and it is much easier for the CPU to deal with the images (pack them to ROS2 messages and transfer them to the other nodes).

7.1.3 Performance Comparison in Mapping

In Section 5.3, a custom ROS2 node for mapping is described. Here, the custom ROS2 mapping node will be used by the e-puck2 physical and simulated robot for performance comparison. In that purpose, a physical map is created, as well as its digital copy in Webots (see Figure 7.2).


BLUE
ROOM
center at approx. (0.25, -0.2)
GREEN
ROOM
center at approx. (0.07, -0.2)
RED
ROOM
center at approx. (0.25, 0.2)
BLUE
ROOM
center at approx. (0.25, -0.2)
GREEN
ROOM
center at approx. (0.07, -0.2)
RED
ROOM
center at approx. (0.25, 0.2)

(a) Webots map (b) Real-world map

Figure 7.2: Webots and physical map used for the mapping benchmark with room names and coordinate systems marked

To create the maps, both robots, physical and simulate, are programmed to follow the same path (see Table A.1). This is done to minimize the difference between the generated maps caused by differences in paths. All maps are saved as pictures, and the comparison of the maps is shown in Figure 7.3.

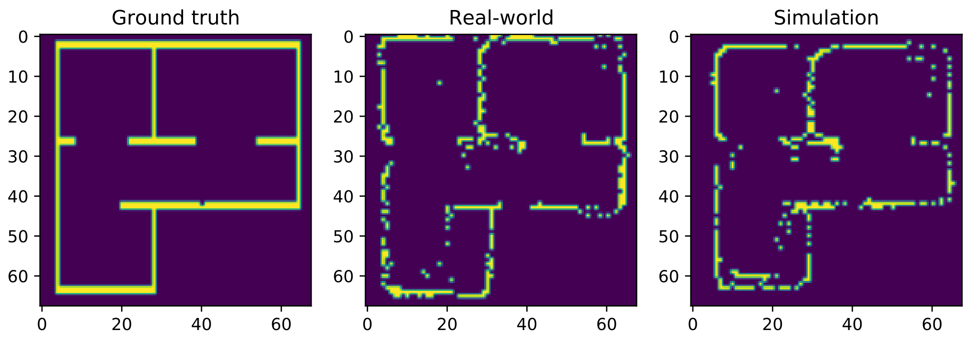


Figure 7.3: Comparison of the ground truth map, map created by physical robot (real-world) and map created by simulated robot (simulation)

One notices, that the quality of the maps generated in the simulation and the physical robot are similar. However, to quantitatively compare the quality of the maps we use IoU:

$$\texttt{IoU} = \frac{\texttt{Area of Overlap}}{\texttt{Area of Union}} \tag{7.1}$$

In which `Area of Overlap` is a number of pixels that represent a wall on both maps, while the `Area of Union` is a number of pixels represent the wall in at least one of the maps.

The IoU gives us the results shown in Table 7.2.

Map	**IoU with ground truth map**
Real-world #1	0.2792208
Real-world #2	0.2596006
Real-world #3	0.29588607
Simulated	0.2993197

Table 7.2: IoU of ground truth and other maps

The table shows that the quality of the maps is very similar. Furthermore, to compare whether the types of errors are similar in both maps, you can check Figure 7.4.

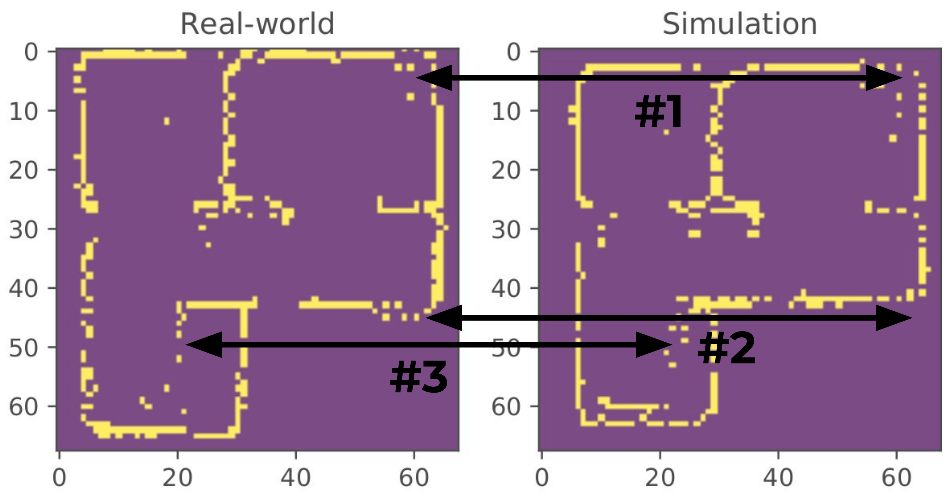


Figure 7.4: Error comparison of map obtained in simulation and real-world

The image shows three types of errors produced by mapping: sparse walls, rounded corners, and invisible walls. Those three types of errors are very similar on both maps.

- Error #1: Sparse walls are a result of a low sampling rate. The robot rotates too fast to map all obstacles.
- Error #2: Rounded corners are result of ToF' sensor wide Field of View (FoV). The sensor doesn't measure the distance to the point, but rather it averages distance to all obstacles in its FoV.
- Error #3: The invisible walls are also caused by ToF' sensor wide FoV. While the robot rotates, the distance measurements to the obstacles get averaged.

This experiment shows that even in more complex scenarios such as mapping, the same controller gives very similar results. The simulated robot not only replicates the same quality of mapping but also captures the same defects. The observed behavior is exactly what we wanted to achieve as it proves that the same controller works correctly with the simulated and physical e-puck2 robots.

7.1.4 Performance Comparison in Navigation

Similar to the mapping, a navigation utilizes a lot of sensors. However, for navigation ROS2 `navigation2` package is used to verify if the ROS2 drivers provide satisfying results with the community packages. The navigation is configured to use a particle filter to fix the transformation between map and odometry frame (although the weight of particle filter is much smaller in comparison to odometry). In addition, it uses the previously obtained map in the mapping experiment.

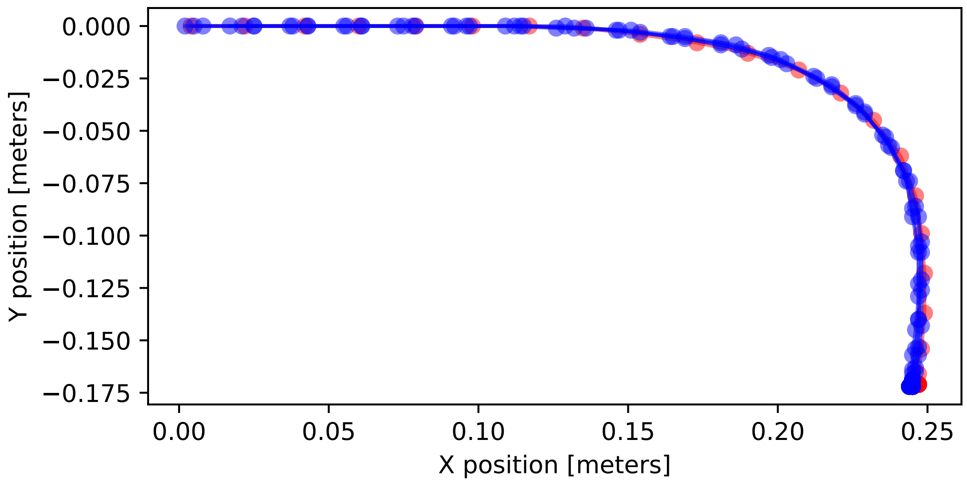


Figure 7.5: Path chosen by the simulated (red) e-puck2 robot and paths chosen by the physical (blue) e-puck2 robot

In Figure 7.5, the comparison in navigation between physical and simulated e-puck2 robots is given. The figure is obtained by sampling robot's location reported by odometry at fixed intervals (sampling period is 0.5s). The robots start from position (0, 0) and they have to move to the blue room (see Figure 7.3). Therefore, in the figure you can see that the paths chosen by the physical and simulated e-puck2 robots are very similar.

7.2 ROS2 Interface for E-puck2 vs Khepera IV

In this experiment, the goal is to verify whether the same ROS2 controller can be used with different robots. As before, there are two tests. The first is with a custom mapping node and the second is with the ROS2 community navigation packages. A map for this experiment had to be bigger to accommodate Khepera IV robot (see Figure 7.6).

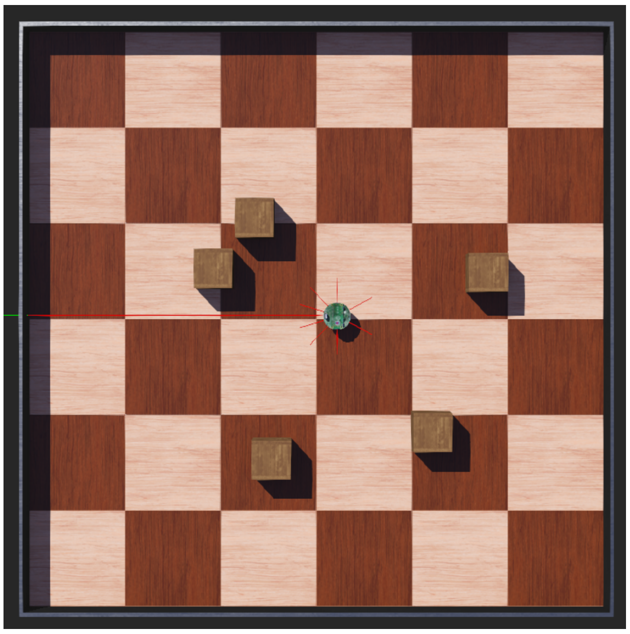


Figure 7.6: Map used to compare the behavior of e-puck2 and Khepera IV with the same ROS2 controller

7.2.1 Performance Comparison in Mapping

In Figure 7.7 we can notice that both robots managed to map the environment.

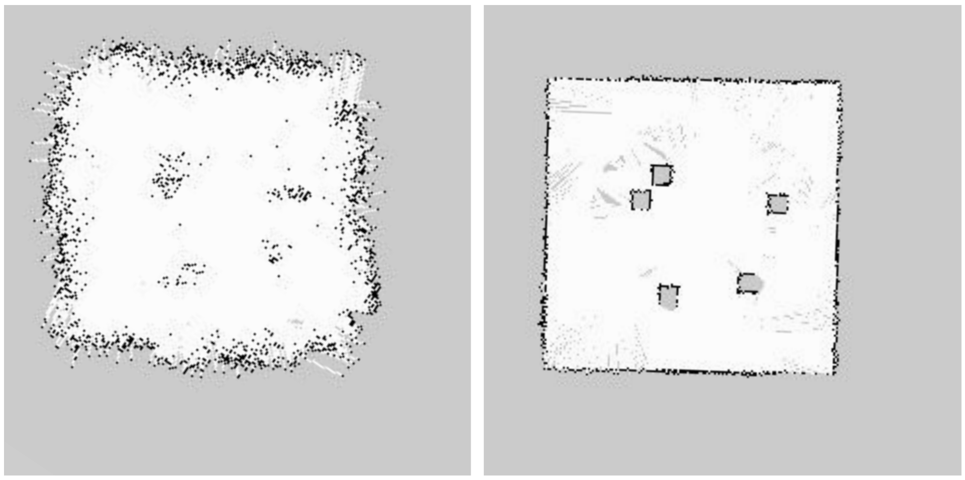


Figure 7.7: Map produced by Khepera IV (left) and e-puck2 (right)

It proofs it is possible to use different robots with same ROS2 controller. However, ToF sensor available on the e-puck2 robot provides visibly better performance in mapping compared to Khepera IV with ultrasonic sensors. The main reason in poor map quality of produced by Khepera IV robot is high noise in ultrasonic sensors.

7.2.2 Performance Comparison in Navigation

Navigation analysis is done in same way as in Section 7.1.4, but with a different map (see Figure 7.8).

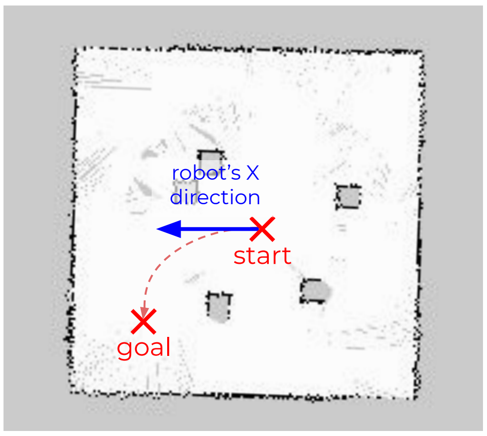


Figure 7.8: Navigation goal on the map

Similarly to the previous navigation comparison the result is shown in Figure 7.9.

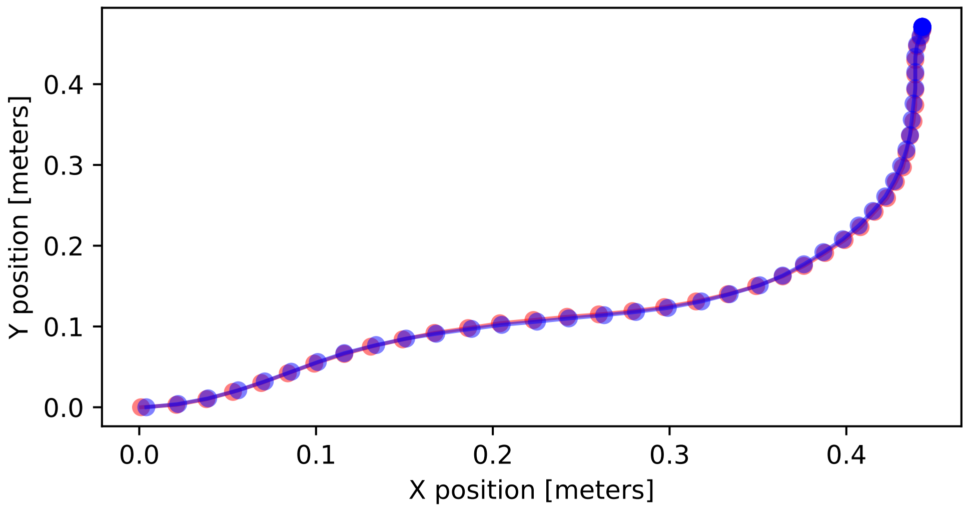


Figure 7.9: Path chosen by e-puck2 (red) and path chosen by Khepera IV (blue)

7.3 Benefits of Generalized ROS2 Interface for Webots

Generalized ROS2 driver on e-puck2 robot produces the same ROS2 interface as a specific ROS2 driver given in Chapter 3. Therefore, in this section, a brief overview on using the generalized ROS2 driver with other robots will be given.

7.3.1 Khepera IV Driver Analysis

In the first figure (Figure 7.10), a result of URDF export is shown. The figure shows a different coordinate frames exported with the new Webots API function.

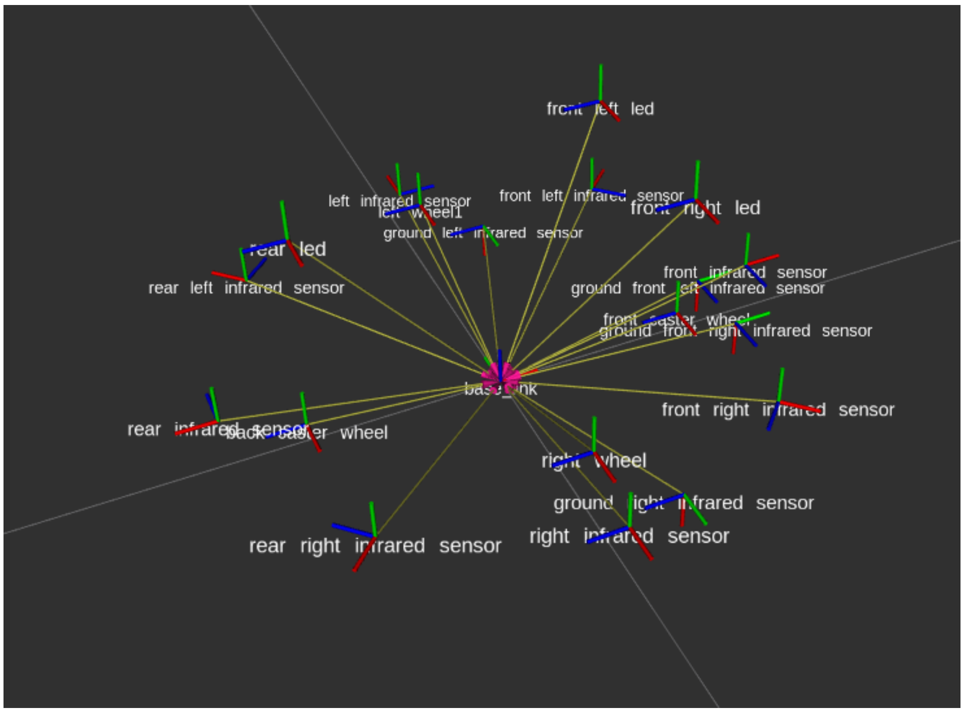

front left led
left infrared sensor
left wheel1
front left infrared sensor
front right led
ground left infrared sensor
rear led
rear left infrared sensor
front infrared sensor
ground front left infrared sensor
front caster wheel
ground front right infrared sensor
base_link
front right infrared sensor
rear infrared sensor
back caster wheel
right wheel
ground right infrared sensor
right infrared sensor
rear right infrared sensor

Figure 7.10: Coordinate frames generated by URDF exporter

Another aspect is ROS2 API available in Table 7.3.

Topic name	**Message type**	**Description**
/camera/camera_info	sensor_msgs/CameraInfo	Camera intrinsic parameters
/camera/image_raw	sensor_msgs/Image	Images from camera
/cmd_vel	geometry_msgs/Twist	Velocity control
/[position]_infrared_sensor (x12)	sensor_msgs/Range	Infrared measurements
/[position]_led (x3)	std_msgs/Int32	LED control
/[position]_ultrasonic_sensor (x5)	sensor_msgs/Range	Ultrasonic measurements
/imu	sensor_msgs/Imu	IMU measurements
/joint_states	sensor_msgs/JointState	Encoder measurements
/odom	nav_msgs/Odometry	Odometry
/robot_description	std_msgs/String	URDF as a string
/tf	tf2_msgs/TFMessage	Dynamic transforms
/tf_static	tf2_msgs/TFMessage	Static transforms

Table 7.3: ROS2 API for Khepera IV robot

The table shows that the ROS2 generalized driver managed to expose devices to the ROS2 system.

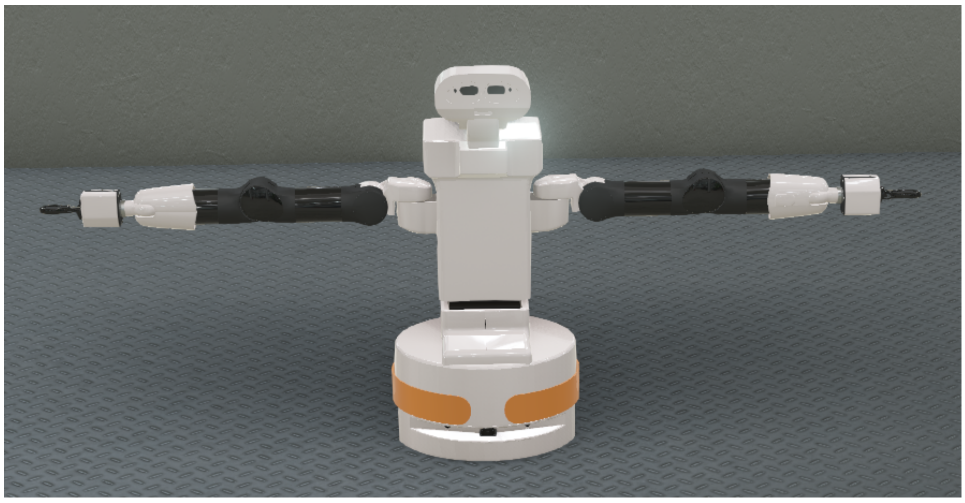
7.3.2 Going Beyond Khepera IV and E-puck2

The ROS2 generalized driver is tested on many robots. In further text, a two examples will be given, TIAGo++ and TurtleBot3 Burger.

TIAGo++ Robot

TIAGo++ robot is designed by PAL Robotics to work in indoor environments. Typically, the platform is used in research or light industry. To us, the robot is interesting because it has a lot of joints[2]. Therefore, it is a good test for the URDF exporter.

[2]A specific ROS2 driver was already developed, but the universal driver presented in this thesis greatly extends its capabilities


(a) TIAGo++ model in Webots

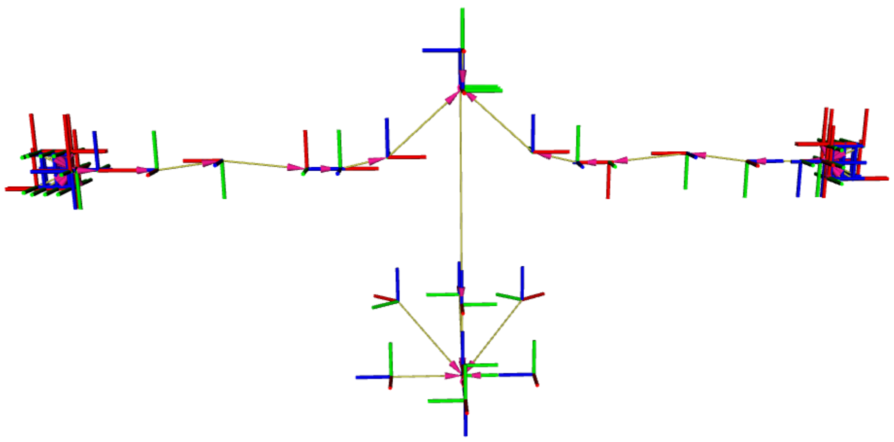

(b) TIAGo++ frames in RViz2

Figure 7.11: Result of URDF exporter on TIAGo++ robot

In Figure 7.11, TIAGo++ model is shown in Webots as well its coordinates frames produced by the URDF exporter.

TurtleBot3 Burger Robot

TurtleBot3 Burger is a small and affordable, often used by ROS team as a reference robot. For us, it was important to verify whether the generalized ROS2 driver is compatible with the official ROS2 package[3] provided by a

[3]The ROS2 TurtleBot3 package is available at `https://github.com/ROBOTIS-GIT/turtlebot3`

team behind TurtleBot3 Burger.

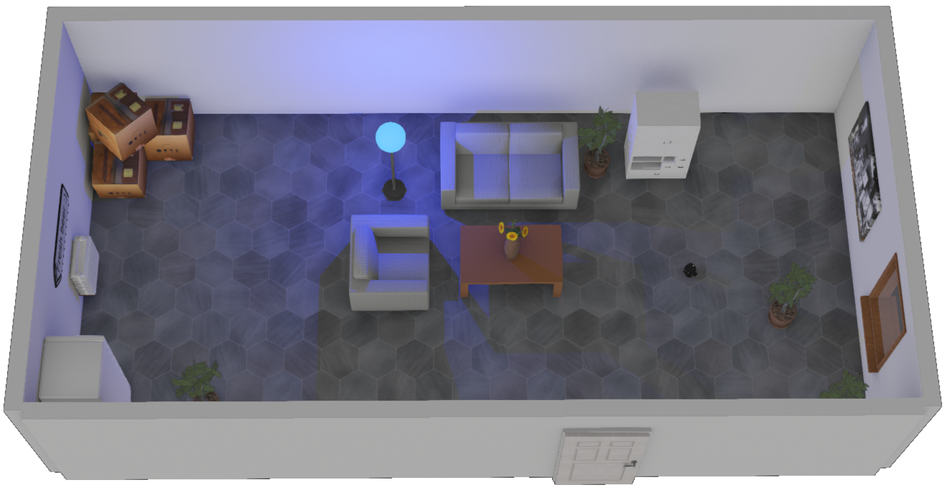


Figure 7.12: Webots world used to verify TurtleBot3 Burger mapping and navigation capabilities

In the figures bellow a RViz2 view for SLAM (see Figure 7.13) and navigation (see Figure 7.14) are given.

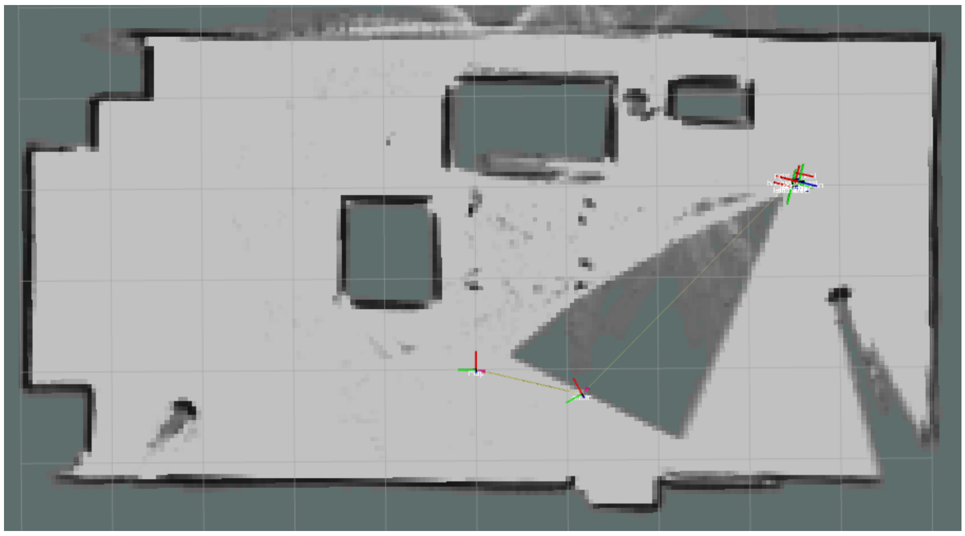


Figure 7.13: TurtleBot3 Burger mapping view in RViz2 (during the mapping process)

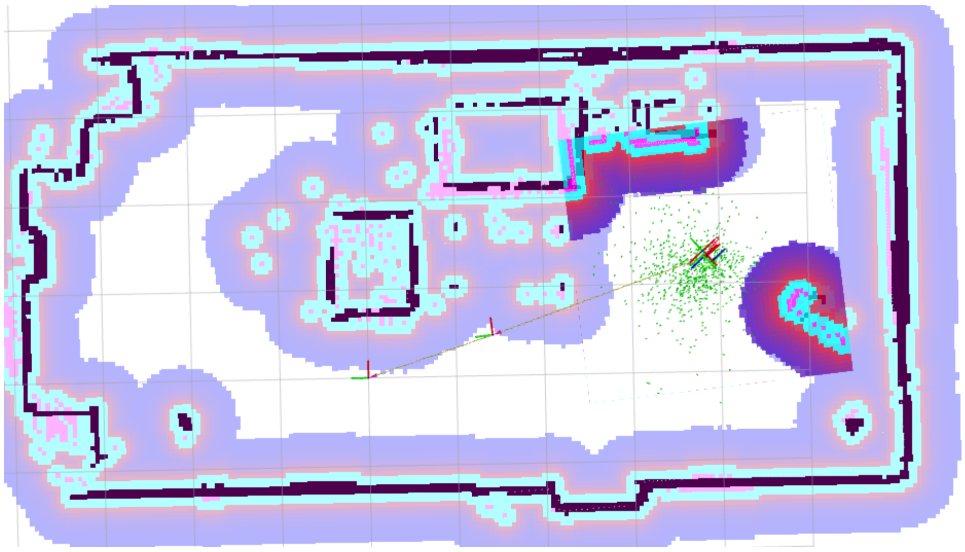


Figure 7.14: TurtleBot3 Burger navigation view in RViz2

The RViz2 views show that the TurtleBot3 package works as expected without any code modifications. Overall, these experiments show that the generalized ROS2 driver scales well to all tested scenarios.

Chapter 8 Conclusion and Future Work

ROS2 interfaces are developed for the e-puck2 physical robot and a variety of simulated robots. A Webots support for ROS2 is improved, providing facilities for automatic creation of a ROS2 interface for various robot models. The ROS2 interfaces provide a firm abstraction over the robot's simulated and physical hardware. Thus, the ROS2 interface allows a ROS2 controller to work with the physical or simulate e-puck2 robot, or with any other simulated robot, without changes needed to the ROS2 controller. The results prove that researchers can quickly validate their ROS2 controllers on the e-puck2 physical or simulated robot and other Webots simulated robots. Effectively closing the loop between the simulation and the physical world.

The thesis summarizes the implementation details of a ROS2 driver for Webots. It shows how Webots distance sensors, IMU related sensors, LEDs, cameras, motors, and encoders are mapped into ROS2 interface. It introduces how odometry, velocity control, and coordinate frames are generated from basic Webots devices, such as motors and encoders.

The ROS2 driver is also developed for the e-puck2 physical robot, and major challenges in its implementation are given. It provides a solution to various problems originated from a low-performance computer with an Armv6 architecture. Most notably, the tackled challenges are cross-compilation, offloading image compression to GPU, unit testing in CI (with x86 architecture), and overall performance issues.

A possibility to automate the creation of ROS2 interface for Webots is observed. Therefore, a universal ROS2 driver for Webots is developed. New features are introduced to Webots core to allow automatic creation of the ROS2 interface, most notably URDF export. On top of the Webots, a modular software layer is implemented that performs API conversion from Webots to ROS2.

The whole project is publicly available on GitHub, allowing users and us to further improve it:

- Webots to ROS2 conversion layer covers the following Webots nodes: `Camera`, `DistanceSensor`, `Accelerometer`, `Gyro`, `InertialUnit`, `LED`, `Lidar`, `LightSensor`, `Robot`, `Motor` and `PositionSensor`. The list of the covered nodes is not exhaustive and more of them should be covered.

- URDF export does not support visual elements, although the visual elements are useful in RViz2 visualizations. Moreover, it does not support `Hinge2Joint`, nor `BallJoint`.
- In certain use cases pi-puck extension is not necessary and ROS2 could be deployed directly to the MCU. This will reduce battery usage and overall system complexity, but it will also put limitations such as a dynamic discovering of communication entities within a network.
- Additional effort should be put in implementing ROS2 package with SLAM for e-puck2 and similar robots (robots with few distance sensors which have a very limited range and wide FoV).

Bibliography

[1] A. G. Millard, R. Joyce, J. A. Hilder, C. Fleşeriu, L. Newbrook, W. Li, L. J. McDaid, and D. M. Halliday, "The Pi-puck extension board: A raspberry Pi interface for the e-puck robot platform," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2017, pp. 741–748, iSSN: 2153-0866.

[2] J. M. Soares, I. Navarro, and A. Martinoli, "The Khepera IV Mobile Robot: Performance Evaluation, Sensory Data and Software Toolbox," in *Robot 2015: Second Iberian Robotics Conference*, L. P. Reis, A. P. Moreira, P. U. Lima, L. Montano, and V. Muñoz-Martinez, Eds. Cham: Springer International Publishing, 2016, vol. 417, pp. 767–781, series Title: Advances in Intelligent Systems and Computing.

[3] K. Viswanathan, "Introduction to Robotics - Dead Reckoning." [Online]. Available: https://www.cs.cmu.edu/~16311/s07/labs/NXTLabs/Lab%203.html

[4] D. Jones, "VideoCore IV - MMAL." [Online]. Available: https://picamera.readthedocs.io/en/release-1.13/api_mmalobj.html

[5] T. Foote, "tf: The transform library," in *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*. Woburn, MA, USA: IEEE, Apr. 2013, pp. 1–6.

[6] O. Michel, "Cyberbotics Ltd. Webots$^{\text{TM}}$: Professional Mobile Robot Simulation," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, p. 5, Mar. 2004, publisher: SAGE Publications.

[7] R. T. Vaughan and B. P. Gerkey, "Really Reusable Robot Code and the Player/Stage Project," *Tracts on Advanced Robotics*, 2007.

[8] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," *ICRA Workshop on Open Source Software*, p. 6, 2009.

[9] C. Chen, Y. Tock, and S. Girdzijauskas, "BeaConvey: Co-Design of Overlay and Routing for Topic-based Publish/Subscribe on Small-

World Networks," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems.* Hamilton New Zealand: ACM, Jun. 2018, pp. 64–75.

[10] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli, "The e-puck, a Robot Designed for Education in Engineering," *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, p. 7, 2009.

[11] O. Michel, "Webots: Symbiosis Between Virtual and Real Mobile Robots," in *Virtual Worlds*, ser. Lecture Notes in Computer Science, J.-C. Heudin, Ed. Berlin, Heidelberg: Springer, 1998, pp. 254–263.

[12] O. Michel, F. Rohrer, and N. Heiniger, "Cyberbotics' Robot Curriculum," Mar. 2014, accepted: 2014-03-13T17:07:57Z Publisher: Wikibooks. [Online]. Available: http://doer.col.org/handle/123456789/4117

[13] M. Kashyian, S. L. Mirtaheri, and E. M. Khaneghah, "Portable Inter Process Communication Programming," in *2008 The Second International Conference on Advanced Engineering Computing and Applications in Sciences*, Sep. 2008, pp. 181–186.

[14] J. Shen, D. Tick, and N. Gans, "Localization through fusion of discrete and continuous epipolar geometry with wheel and IMU odometry," in *Proceedings of the 2011 American Control Conference*, Jun. 2011, pp. 1292–1298, iSSN: 2378-5861.

[15] D. Nister, O. Naroditsky, and J. Bergen, "Visual odometry," in *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, vol. 1, Jun. 2004, pp. I–I, iSSN: 1063-6919.

[16] M. Ben-Ari and F. Mondada, *Elements of Robotics.* Cham: Springer International Publishing, 2018. [Online]. Available: http://link.springer.com/10.1007/978-3-319-62533-1

[17] A. Astolfi, "Exponential Stabilization of a Wheeled Mobile Robot Via Discontinuous Control," *Journal of Dynamic Systems, Measurement, and Control*, vol. 121, no. 1, pp. 121–126, Mar. 1999.

[18] E. Hairer, S. Norsett, and G. Wanner, *Solving Ordinary Differential Equations I*, ser. Springer Series in Computational Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, vol. 8.

[19] J. Diebel, "Representing attitude: Euler angles, unit quaternions, and rotation vectors," *Matrix*, vol. 58, 01 2006.

[20] P. Michael, "A conversion guide: Solar irradiance and lux illuminance," 2019. [Online]. Available: http://dx.doi.org/10.21227/mxr7-p365

[21] N. Laković, M. Brkić, B. Batinić, J. Bajić, V. Rajs, and N. Kulundžić, "Application of low-cost vl53l0x tof sensor for robot environment detection," in *2019 18th International Symposium INFOTEH-JAHORINA (INFOTEH)*, 2019, pp. 1–4.

[22] D. Jones, "VideoCore IV - Hardware." [Online]. Available: https://picamera.readthedocs.io/en/release-1.13/fov.html

[23] D. Lukic, "Dual Fisheye Camera Calibration," Swiss Federal Institute of Technology in Lausanne (EPFL), Tech. Rep., 2019. [Online]. Available: https://lukic.io/files/Dual_Fisheye_Camera_Calibration.pdf

[24] M. Meyer, "Continuous Integration and Its Tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, May 2014, conference Name: IEEE Software.

[25] J. Borenstein and Liqiang Feng, "Measurement and correction of systematic odometry errors in mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 6, pp. 869–880, Dec. 1996.

[26] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The Marathon 2: A Navigation System," *arXiv:2003.00368 [cs]*, Jul. 2020, arXiv: 2003.00368.

[27] "URDF - ROS Wiki." [Online]. Available: http://wiki.ros.org/urdf

[28] D. Eberly, "Euler Angle Formulas." [Online]. Available: https://www.geometrictools.com/Documentation/EulerAngles.pdf

[29] K. Zheng, "ROS Navigation Tuning Guide," *arXiv preprint arXiv:1706.09068*, p. 23, 2017.

[30] O. Michel and F. Rohrer, "The Rat's Life benchmark: competing cognitive robots," in *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems - PerMIS '08*. Gaithersburg, Maryland: ACM Press, 2008, p. 43.

[31] F. Mondada, E. Franzi, and A. Guignard, "The Development of Khepera," in *Experiments with the Mini-Robot Khepera, Proceedings of the First International Khepera Workshop*, no. CONF, 1999, pp. 7–14.

[32] "ros-industrial/industrial_ci," Jul. 2020, original-date: 2015-11-18T15:15:44Z. [Online]. Available: https://github.com/ros-industrial/industrial_ci

[33] "ROS.org | About ROS," library Catalog: www.ros.org. [Online]. Available: https://www.ros.org/about-ros/

[34] J. M. O'Kane, *A gentle introduction to ROS.* Leipzig: Amazon, 2014, oCLC: 935415054.

[35] N. Correll, N. Correll, and Open Textbook Library, *Introduction to Autonomous Robots.* MIT press, 2016, oCLC: 1136484039.

[36] J. E. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965, conference Name: IBM Systems Journal.

Appendix A Mapping Comparison

Command	**Description**
position=[0, 0]	Initial pose
orientation=$-\frac{\pi}{2}$	Explore the hall, rotate
orientation=0	Explore the hall, rotate
position=[0.28, 0], orientation=0	Move near RED entrance
orientation=$\frac{\pi}{2}$	Rotate towards RED
position=[0.28, 0.2]	Move inside RED
orientation=0	Explore RED, rotate
orientation=$-/pi$	Explore RED, rotate
orientation=$-\frac{\pi}{2}$	Rotate towards BLUE
position=[0.23, -0.2]	Move inside BLUE
orientation=0	Explore BLUE, rotate
orientation=$\frac{\pi}{3}$	Explore BLUE, rotate
orientation=$-\frac{\pi}{2}$	Explore BLUE, rotate
orientation=-2.7*pi	Explore BLUE, rotate
orientation=$\frac{\pi}{2}$	Rotate towards the hall
position=[0.23, 0]	Go to the hall
orientation=$\frac{\pi}{3}$	Explore the hall, rotate
orientation=π	Rotate towards back
position=[-0.1, 0]	Go to the GREEN entrance
orientation=$\frac{\pi}{2}$	Explore hall, rotate
orientation=$\frac{\pi}{4}$	Explore hall, rotate
orientation=$\frac{2}{3}\pi$	Explore hall, rotate
orientation=$-\frac{\pi}{2}$	Rotate towards the GREEN
position=[-0.1, -0.2]	Move inside GREEN
orientation=0	Explore GREEN, rotate
orientation=$\frac{\pi}{3}$	Explore GREEN, rotate
orientation=$-\frac{\pi}{2}$	Explore GREEN, rotate
orientation=$\frac{3}{4}\pi$	Explore GREEN, rotate

Table A.1: List of poses used to map the environment