

Degree Programme
Systems Engineering
Major Infotronics

BACHELOR'S THESIS

DIPLOMA 2022

Samy Francelet

BLE2ROS

Professor
Prof. Medard Rieder

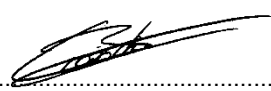

Expert
Thierry Hischier

Submission date of the report
19.08.2022

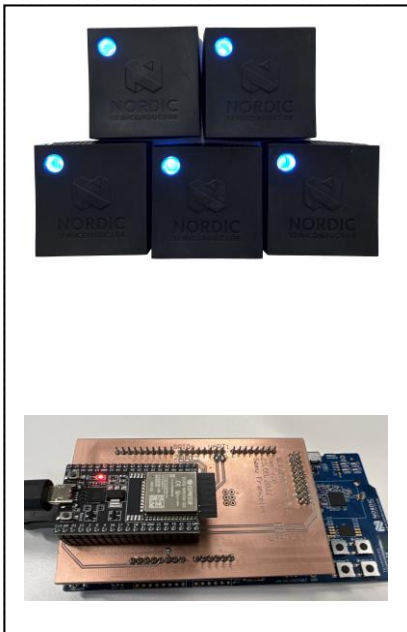


Filière / Studiengang SYND	Année académique / Studienjahr 2021-22	No TB / Nr. BA IT/2022/76
Mandant / Auftraggeber <input type="checkbox"/> HES—SO Valais <input checked="" type="checkbox"/> Industrie <input type="checkbox"/> Etablissement partenaire <i>Partnerinstitution</i>	Etudiant / Student Samy Francelet <hr/> Professeur / Dozent Medard Rieder	Lieu d'exécution / Ausführungsort <input checked="" type="checkbox"/> HES—SO Valais <input type="checkbox"/> Industrie <input type="checkbox"/> Etablissement partenaire <i>Partnerinstitution</i>
Travail confidentiel / vertrauliche Arbeit <input type="checkbox"/> oui / ja <input checked="" type="checkbox"/> non / nein	Expert / Experte (données complètes) Thierry Hischier - thierry.hischier@thalesgroup.com Thales S&T AG, Stauffacherstrasse 65, 3000 Bern 22	

Titre / Titel	BLE2ROS Gateway
Description <p>The goal of the thesis is a proof of concept to integrate BLE mesh Nodes over a BLE2ROS gateway into a typical ROS network. A specific ROS Node with a BLE2ROS Gateway provides specific sensor topics coming from sensors in the BLE Mesh network. This specific ROS node can publish this sensor data to the ROS network. It must also be possible, that the specific ROS node can subscribe for specific data in the ROS network and publish this data in a reduced defined periodicity into the BLE mesh network.</p> <p>Goals</p> <ul style="list-style-type: none"> Evaluate and put to work a hardware platform for the gateway. An NRF53xxx platform is proposed. A serially connected ESP32 with a microROS node will make the connection to the ROS system. Some sensors, such as temperature, gyro and some actuators such as a led or other has to be integrated too. Sensor may be NRF52xxx devices. The thesis should examine the boundaries of such a solution including the investigation of the number of possible BLE Mesh nodes, the maximum possible data rate, and data size. Implement and test the mesh stack on the BLE side. Put to work a small network with a couple (2-3) nodes. Implement and test the BLE2ROS gateway software on the NRF53 and the microROS node to publish the sensor data topics and subscribe for specific ROS data. <p>Deliverables</p> <ul style="list-style-type: none"> Entire source code with UML diagrams Working demonstrator Technical report and presentation slides. 	

Signature ou visa / Unterschrift oder Visum Responsable de l'orientation / <i>Leiter der Vertiefungsrichtung:</i>  ¹ Etudiant / Student : 	Délais / Termine Attribution du thème / Ausgabe des Auftrags: 16.05.2022 Présentation intermédiaire / Zwischenpräsentation: 20-21.06.2022 Remise du rapport final / Abgabe des Schlussberichts: 19.08.22, 12:00 Expositions / Ausstellungen der Diplomarbeiten: 24-26.08.2022 Défense orale / Mündliche Verfechtung: Semaine/Woche 36 (5-9.09.2022)
--	--

¹ Par sa signature, l'étudiant-e s'engage à respecter strictement la directive DI.1.2.02.07 liée au travail de diplôme.
 Durch seine Unterschrift verpflichtet sich der/die Student/in, sich an die Richtlinie DI.1.2.02.07 der Diplomarbeit zu halten.



Bachelor's Thesis | 2022 |

Degree programme
Systems Engineering

Field of application
Major Infotronics

Supervising professor
Prof. Medard Rieder
medard.rieder@hevs.ch

Partner
Thales S&T AG
Thierry Hischier
thierry.hischier@thalesgroup.com

BLE2ROS

Graduate

Samy Francelet

Objectives

BLE2ROS is a gateway integrating Bluetooth Mesh Networking into the Robot Operating System (ROS)

Methods | Experiences | Results

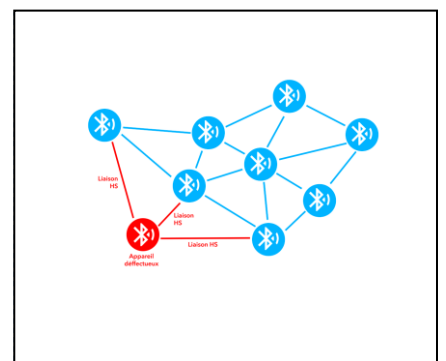
The goal of this thesis is a proof of concept of a gateway, linking ROS and Bluetooth Mesh Networking together. This gateway consists of two, embedded systems: a microROS Node and a BT Mesh Provisioner node. The microROS node, implemented on an ESP32, will serve as a bridge between ROS and the Mesh Network. The BT Mesh Provisioner, implemented on a nRF53, will receive commands from ROS to manage the network and will transfer data between ROS and the Mesh network. Multiple BT Mesh nodes are also implemented, using Thingy:52, to test the data rate limits of this solution.

Is it possible to link BT Mesh to ROS? Yes. Is it efficient? Maybe: the current bugs of the microROS library makes this uncertain. The current result is a gateway, capable of transmitting a joystick's data from Mesh to ROS, controlling lighting of the whole network with ROS, and receive a feedback.

The data rate is limited, due to BT Mesh network underlying physical layer being the Bluetooth Low Energy, and to the specific use case we have.



The Robot Operating System is a set of libraries and tools used to develop robot applications. It achieves great interoperability using node, communicating via Topics. Services and Actions



Bluetooth Mesh Networking is a computer mesh networking standard, based on Bluetooth Low Energy, that allows a full-stack, decentralized, efficient and reliable mesh network

Information about this report

Contact information

Author: Samy Francelet
Bachelor Student
HES-SO//Valais Wallis
Switzerland
Email: samy.francelet@ik.me

Declaration of honor

I, undersigned, Samy Francelet, hereby declare that the work submitted is the result of a personal work. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and the author quotes were clearly mentioned.

Place, date: Sion, August 19, 2022

Signature: _____

Acknowledgments

Thanks to **pablogs9**, maintainer of the microROS ESP_IDF component, for all the help given while working with this component and the patches made when I found bugs in the library.

Thanks to the Jass players from the I6 class for providing fun breaks during the thesis.

Thanks to **Vincent Savioz** for the coffee machine, which gave me sufficient caffeine to keep working.

Thanks to **Pascal Sartoretti** and **Patrick Rudaz** who gave me the materials needed for the thesis.

Special thanks to **Medard Rieder** and *Nordic semiconductors* for providing the nRF dev boards and Thingies:52 that allowed me to set up a Mesh Network quickly with more than two nodes.

Abstract

With the beginning of the fourth industrial revolution, large-scale machine-to-machine communication and IoT become more and more a focus of the industry. From building hundreds of robots, all are communicating together, for warehouse automation, to using thousands of IoT sensors, ultra-low power, to record every data needed in a factory. However, this raises a problem: How to effectively interface complex machines with external, simpler devices wirelessly? You need very complex, high-level communication protocols to manage a fleet of robots, while also needing a lower power, lower data rate, IoT protocol.

The goal of this thesis is a proof of concept of a gateway, linking ROS and Bluetooth Mesh Networking together. This gateway consists of two embedded systems: a microROS Node and a BT Mesh Provisioner node. The microROS node, implemented on an ESP32, will serve as a bridge between ROS and the Mesh Network. The BT Mesh Provisioner, implemented on an nRF53, will receive commands from ROS to manage the network and will transfer data between ROS and the Mesh network. Multiple BT Mesh nodes are also implemented, using Thingy:52, to test the data rate limits of this solution.

Is it possible to link BT Mesh to ROS? Yes. Is it efficient? Maybe: the current bugs of the microROS library make this uncertain. The current result is a gateway capable of transmitting a joystick's data from Mesh to ROS, controlling the lighting of the whole network with ROS, and receiving feedback. The data rate is limited due to the BT Mesh network underlying physical layer being the Bluetooth Low Energy, and to the specific use case we have.

Key words: ROS, microROS, Bluetooth Mesh, gateway, industry 4.0

Contents

Acknowledgements	vii
Abstract	ix
Contents	x
List of Figures	xii
List of Tables	xii
List of Listings	xiii
1 Introduction	1
1.1 Context / Problem	1
1.2 Objectives	2
1.3 Structure of this report	3
2 Analysis	5
2.1 ROS	6
2.2 BT Mesh	11
2.3 Conclusion	15
3 Design	17
3.1 BLE2ROS gateway	18
3.2 BT Mesh nodes	22
3.3 ROS Node	23
4 Implementation	25
4.1 UART Communication	26
4.2 BT Mesh nodes	30
4.3 BT Mesh Provisioner	32
4.4 microROS node	34
4.5 ROS node with UI	35
5 Validation	37
5.1 Data through ROS and BT Mesh	38
5.2 Solution's boundaries	39
5.3 Data rate between BT Mesh Network and ROS	41
6 Conclusions	43

6.1	Project summary	43
6.2	Comparison with the initial objectives	43
6.3	Encountered difficulties	44
6.4	Future perspectives	45
A	UART MAC - Zephyr and ESP-IDF	47
A.1	UART MAC TX - Zephyr	48
A.2	UART MAC RX - Zephyr	51
A.3	UART MAC TX - ESP-IDF	55
A.4	UART MAC RX - ESP-IDF	58
B	UART Physical - Zephyr	63
B.1	Header	64
B.2	Source	66
C	UART Physical - ESP-IDF	71
C.1	Header	72
C.2	Source	74
D	UART Message emulator	77
E	BLE2ROS Test Data	81
	Bibliography	83
	Acronyms	85
	Glossary	87

List of Figures

1.1	Ocado warehouse robots	1
1.2	BLE2ROS Network	2
2.1	ROS2 nodes	6
2.2	ROS2 topics	7
2.3	ROS2 services	8
2.4	ROS2 actions	9
2.5	microROS architecture	10
2.6	Examples of network topologies	11
2.7	BT Mesh nodes	12
2.8	BT Mesh messages	13
2.9	BT Mesh provisioning: node lifecycle	14
2.10	Gazebo simulator for ROS	15
3.1	BLE2ROS Overview	17
3.2	Shield layout	18
3.3	BT Mesh provisioner firmware design	19
3.4	microROS node firmware design	20
3.5	Thingy:52	22
3.6	BLE2ROS UI Design	24
4.1	UART Layers	26
4.2	Mesh Node firmware architecture	30
4.3	First mesh network, using multiple Thingy:52	30
4.4	Sensor node hardware, using a nRF52DK-nRF52832	31
4.5	Basic provisioning sequence diagram	32
4.6	BLE2ROS provisioning sequence diagram	33
4.7	BLE2ROS UI Implementation with PyQt	35
5.1	BT Mesh nodes, provisioned from ROS	38
5.2	Minimum period between messages vs data size	41

List of Tables

3.1	BLE2ROS UART Protocol frame	21
3.2	UART MIDs	21
3.3	Ping service	23
3.4	ROS messages	23
6.1	Comparisons with initial objectives	43

List of Listings

2.1	ROS2 joystick interface example	7
2.2	ROS2 service interface example	8
4.1	UART message structure	26
4.2	UART MAC TX Thread pseudocode	27
4.3	UART MAC RX Thread pseudocode	27
4.4	UART Message emulator Ping test results	29
4.5	colcon.meta configuration for Micro XRCE-DDS	34

1 | Introduction

1.1 Context / Problem

With the arrival of the fourth industrial revolution, bringing large-scale [machine-to-machine communication \(M2M\)](#) and [IoT](#) to modern smart technologies and automation, the need for interconnectivity keeps increasing. Rising usage of robotics combined with vast networks of sensors brings a new connectivity problem: **How to effectively interface complex machines with external, simpler sensors, wirelessly?**

Say you are managing a warehouse using robots. In the beginning, you have 10 robots, each receiving specific tasks like: *"go to shelf B7 and pick item G2"*. When you receive an order on your website, a central computer will dispatch the tasks to your 10 robots, and the order will quickly be made. As time passes, you have more customers and quickly realize that you need a bigger warehouse and more robots.



Figure 1.1 Warehouse robots operating goods logistics in an Ocado warehouse.¹

Now, you have **70** robots and quickly realize the central computer is overwhelmed and cannot properly manage every robot. At this point, you want to make the robots *smarter*, so the central computer can just send the task: *"I need 2 keyboards, 12 bottles of beer, ..."* and the robots will autonomously assemble the order. However, there is still a problem: **How do the robots know where the items are?** They need to synchronize with the inventory on the central computer, which also overwhelms it.

What if each shelf knows what items it contains? You equip every item with a tag detected by a sensor on the shelf. Those shelf's sensors can all be linked wirelessly and connected to the same network as the robots, giving full inventory access to every bot. This setup hugely increases the effectiveness of your warehouse and, most importantly, makes upscaling much easier.

¹Image from the 4th Industrial Revolution Wikipedia page: https://en.wikipedia.org/wiki/Fourth_Industrial_Revolution

Chapter 1. Introduction

To achieve this setup while staying cost-efficient, you'll most likely use two different communication protocols that can manage vast networks:

- **A powerful one** for the robots, so they can communicate with each other (e.g. [Robot Operating System \(ROS\)](#))
- **A lighter one** for the shelves (e.g. [Bluetooth Mesh Networking \(BT Mesh\)](#))

Which brings us to the theme of this thesis: **How to make ROS and BT Mesh work together? Is it reliable? What are the limitations? Is it beneficial to use both protocols together?**

1.2 Objectives

The goal of the thesis is a proof of concept of a gateway integrating [BT Mesh](#) nodes into a typical [ROS](#) network. The gateway will consist of two systems, a [microROS](#) Node and a [BT Mesh Provisioner](#) node. The [Provisioner](#) will receive commands from the [microROS](#) node to add or remove [BT Mesh](#) nodes, and manage the [BT Mesh](#) network. The [BT Mesh](#) network will be composed of multiple nodes, from simple LED and buttons nodes to sensor nodes. The [Provisioner](#) node will transmit sensor data from the network to the [microROS](#) node, whose job is to relay the data to the [ROS](#) network.

It must also be possible that the [microROS](#) node can subscribe for specific data in the [ROS](#) network and publish this data in a reduced defined periodicity into the [BT Mesh](#) network.

An ESP32 from Espressif Systems will take the role of the [microROS](#) node. A nRF53, from Nordic Semiconductors, will be the [BT Mesh Provisioner](#). Multiple nRF52 (mostly Thingy:52), will be used as low-power, [BT Mesh](#) nodes.

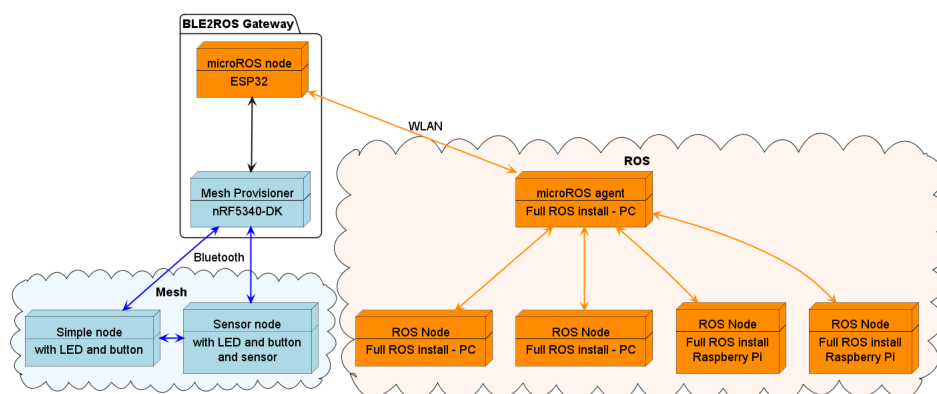


Figure 1.2 BLE2ROS Network

1.2.1 Requirements

- **Evaluate** a hardware platform for the gateway. Using an nRF53 as the [BT Mesh Provisioner](#) and a serially connected ESP32 with a [microROS](#) node will make the connection to [ROS](#).
- **Examine the boundaries** of such a solution, including the maximal number of possible [BT Mesh](#) nodes, the maximum possible data rate, and data size.
- **Create Mesh nodes** with some sensors, such as gyro, and some actuators, such as LEDs.
- **Implement the Mesh stack** and put a small network with a couple of nodes to work.
- **Implement the BLE2ROS gateway** firmware on the nRF53 and [microROS](#) node to to publish the sensors' data to [ROS](#) and drive the [BT Mesh](#) Network from [ROS](#).

1.3 Structure of this report

First, a *State of the Art* analysis [2](#) will be made, describing both [ROS](#) and [BT Mesh](#), and the relevance of such a gateway.

Then, the *Design* [3](#) and *Implementation* [4](#) of the BLE2ROS gateway will show how the link between both protocols is made.

Finally, a *Validation* [5](#), showing capabilities and limitations of the platform, and a *Conclusion* [6](#) summarizing the whole project.

2 | Analysis

In this chapter, a so-called *State of the Art* analysis is done. It will go through details about both the [Robot Operating System \(ROS\)](#) and [Bluetooth Mesh Networking \(BT Mesh\)](#), about why one or another isn't sufficient for [M2M](#) of both complex machines, such as robots, and low-power [IoT](#) equipment.

Contents

2.1	ROS	6
2.1.1	ROS nodes	6
2.1.2	ROS topics	7
2.1.3	ROS services	8
2.1.4	ROS actions	9
2.1.5	microROS	10
2.2	BT Mesh	11
2.2.1	Mesh Networking	11
2.2.2	Managed flooding	11
2.2.3	Node types	12
2.2.4	BT Mesh models	13
2.2.5	BT Mesh messages	13
2.2.6	Provisioning	14
2.3	Conclusion	15
2.3.1	ROS	15
2.3.2	Bluetooth Mesh	15
2.3.3	Solution chosed	15

2.1 ROS

"The [Robot Operating System \(ROS\)](#) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source."[1]

Here we will mostly talk about the more recent [ROS](#) version, ROS2. Every aspect of covered in this section comes from the *ROS Documentation*[2].

2.1.1 ROS nodes

[ROS](#) achieves great interoperability using nodes. Each node is responsible for a single function (e.g., one node controlling motors, another for detecting obstacles with data from another node controlling a LiDAR, etc.). Each node can communicate with other nodes via topics, services, actions, or parameters.

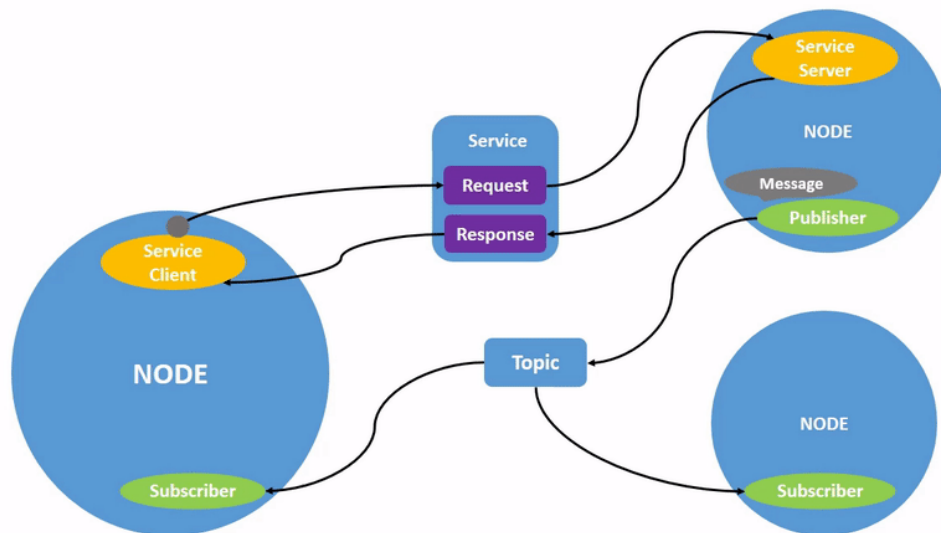


Figure 2.1 ROS2 nodes, with topics and services [2]

2.1.2 ROS topics

In ROS, topics are used as *buses* to exchange messages. The act of transmitting data to a topic is called *publishing* while listening for data on a topic is called *subscribing*. A node can publish messages to many topics and at the same time, subscribe to many other topics.

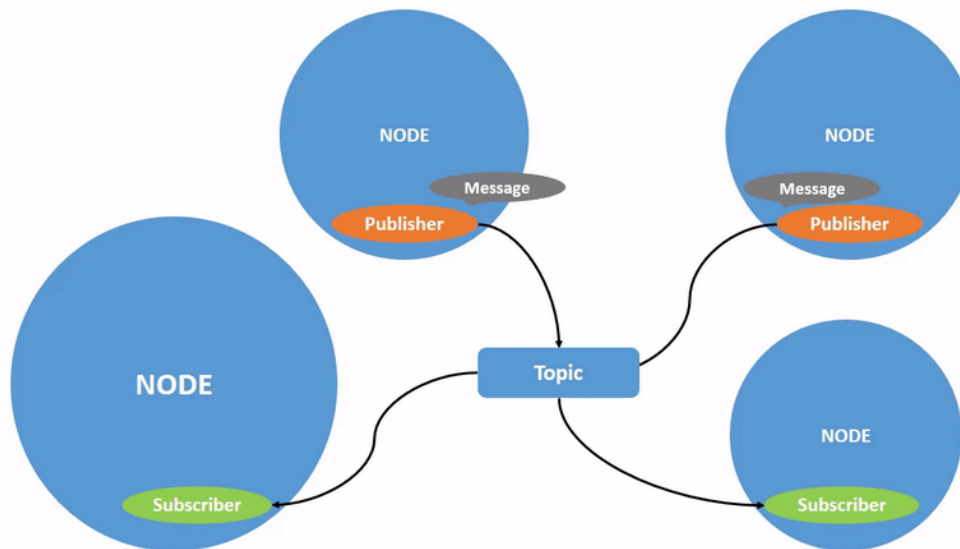


Figure 2.2 ROS2 topics [2]

Each topic is defined with an interface whose job is to ensure that every node can understand the messages transmitted through those topics. For example, a topic *joystick* will have the interface:

```
1 # joystick axis data
2 int16 x
3 int16 y
```

Listing 2.1 ROS2 joystick interface example

This means that every message sent to the joystick topic will contain two 16-bit integers: *x* and *y*. Those interfaces can be more complex, containing arrays of data or even other interfaces as a data type.

2.1.3 ROS services

Services in ROS function like every computer service. A client sends a request to a server and receives a response from the server.

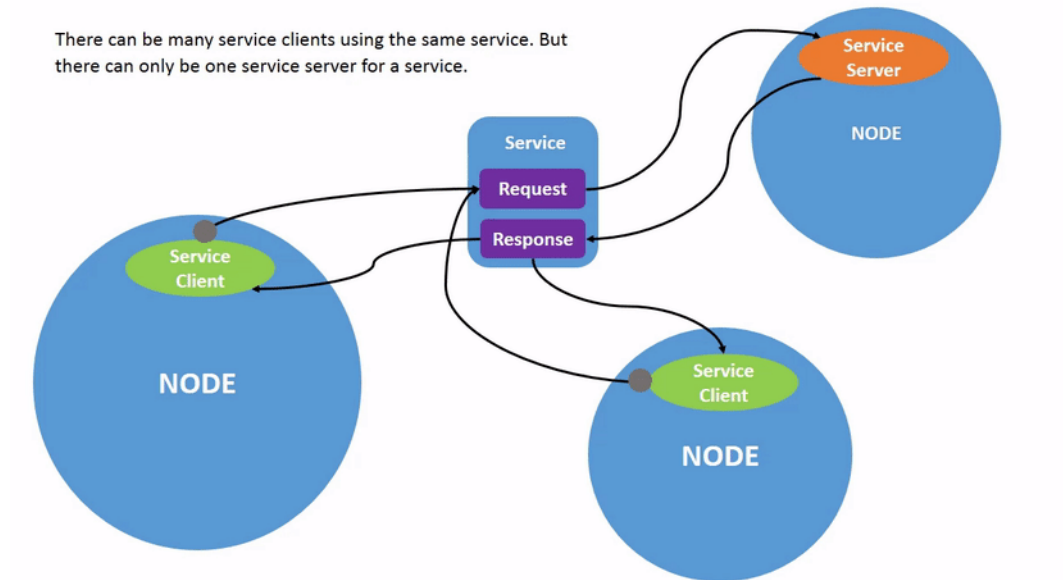


Figure 2.3 ROS2 services [2]

Like the topics, an interface must be defined for each service.

```
1 # Sends a ping with a text
2 # Receives a pong with same text
3 string ping
4 ---
5 string pong
```

Listing 2.2 ROS2 service interface example

The request interface is separated from the response by the three —.

2.1.4 ROS actions

ROS is made for robots, and robots executes *actions*. A controller might want to start a movement on the wheels and get feedback while the robot is moving.

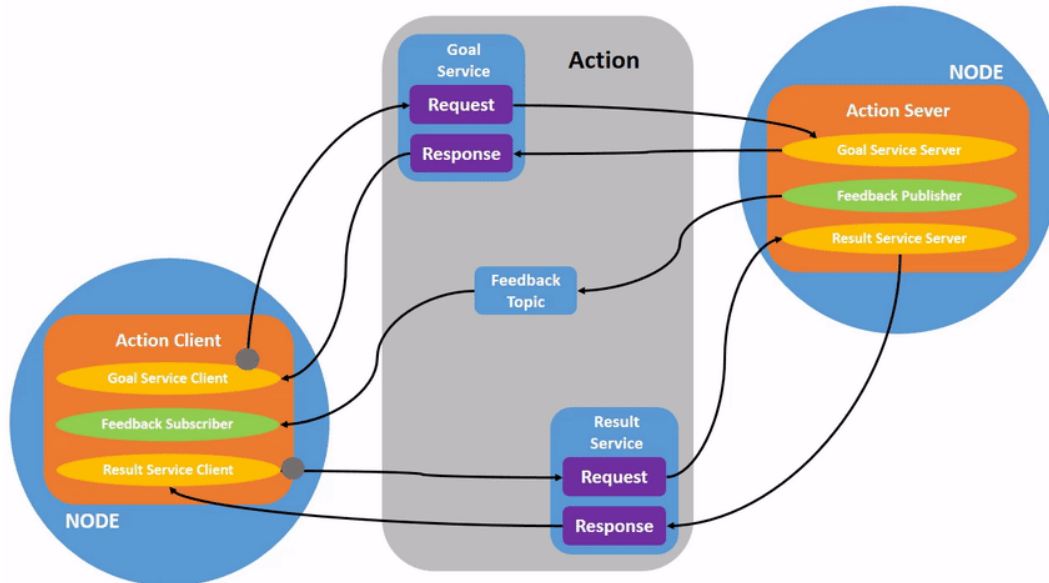


Figure 2.4 ROS2 actions [2]

An action implements two services, one for the goal (e.g., moves 10 meters) and one for the result. And a topic for continuous feedback on the task. The action client will send the goal to the server, which acknowledges the goal with the goal service, then the client will send a result request, starting the action. The action server will provide feedback on the topic, and the client can decide to send another goal or cancel the current goal at any time during the action.

2.1.5 microROS

microROS is a feature complete port of **ROS** for micro-controllers, backed by many company like *Bosch*, *Amazon*, *Canonical* and *Wind River*.¹

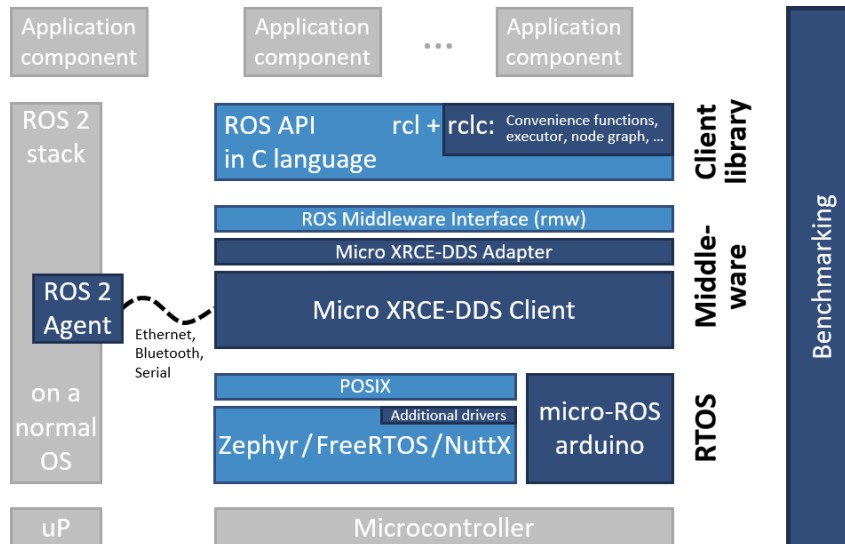


Figure 2.5 **microROS** architecture [3]

microROS uses the *Micro XRCE-DDS Middleware* to communicate with the **ROS** Middleware. It can exchange messages via a serial cable, ethernet cable, or WiFi (other transport protocols can also be implemented manually [4]). As a consequence, a **microROS** node must be connected to a **ROS Agent**, running on a full **ROS** install on a computer to communicate with the **ROS** network. This means that **microROS** currently cannot work independently from computers using **ROS**. It is compatible with many **RTOS**s, like *Zephyr* or *FreeRTOS*, and even works on simple *Arduinos*.

¹Partners cited from microROS webpage: https://micro.ros.org/docs/overview/users_and_clients/

2.2 BT Mesh

Bluetooth Mesh Networking (BT Mesh) is a computer mesh networking standard, based on the Bluetooth Low Energy (BLE) protocol, that allows a full-stack, decentralized, efficient and reliable mesh network[5]. Every detail described in this section comes from the *BT Mesh Glossary*[6].

2.2.1 Mesh Networking

A Mesh network is a network topology where nodes are linked in a direct, dynamic and non-hierarchical way to each other, typically wirelessly.

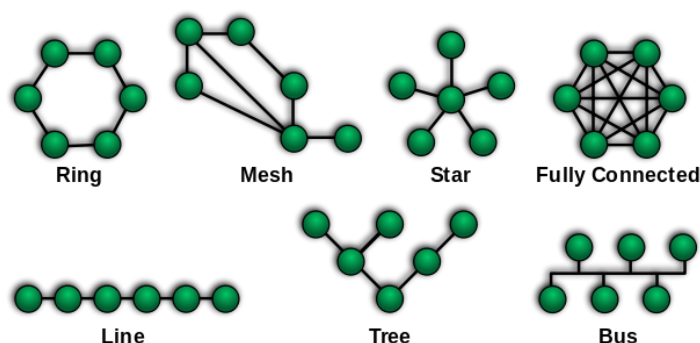


Figure 2.6 Examples of network topologies.¹

Non-hierarchical means that there is not any required specific node that the network needs, unlike WiFi with its WLAN routers.

2.2.2 Managed flooding

There are two ways to relay messages in a mesh network: Flooding it over every node in the net or having a complex routing mechanism to ensure the shortest path to the destination. BT Mesh uses flooding, but with some optimizations:

- All packets include a Time To Live (TTL) field. This field contains a number decreased every time a node relays the message. When it comes to 0, the message will not be relayed. BT Mesh optimizes the flooding by setting the TTL as low as possible. Each node has a general idea of the distance to other nodes because they all send regular *heartbeat* messages.
- Every node contains a cache to ensure it is not relaying a message it just relayed.

¹Wikipedia source: https://en.wikipedia.org/wiki/Network_topology

2.2.3 Node types

Another optimization of the **BT Mesh** is the different node types available for custom usage.

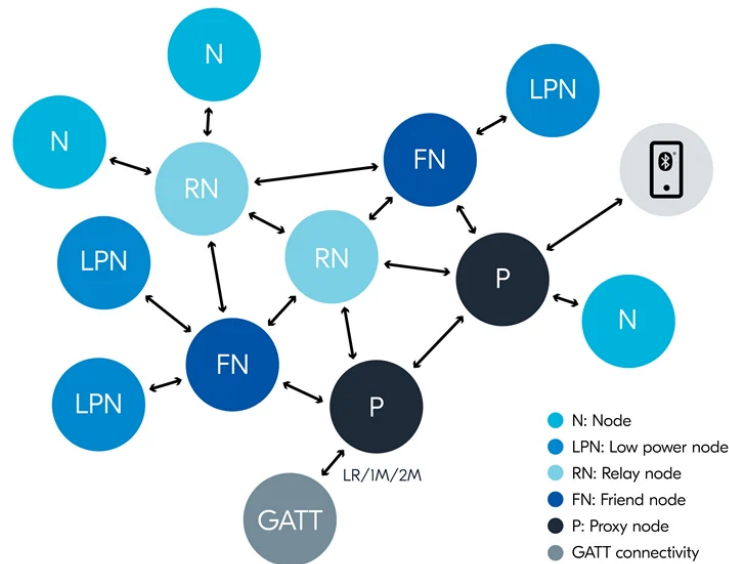


Figure 2.7 **BT Mesh** nodes.¹

Relay node: typical **BT Mesh** node, can receive and transmit messages

Proxy nodes: can receive and transmit data between **BT Mesh** and **BLE**

Low power nodes: power-constrained devices can be designated as "low power nodes" and will work in conjunction with one or more devices, designated "friends".

Friends nodes: friends are not power constrained and keep messages addressed to the low power node while its power saving.

¹Image source: <https://www.nordicsemi.com/Products/Bluetooth-mesh/What-is-Bluetooth-mesh>

2.2.4 BT Mesh models

A BT Mesh device composition is made of models. Those models represents clients/servers that can publish/subscribe to BT Mesh groups. Many SIG-defined models, called *Root Models* or *SIG Models*, are available for basic mesh networks, like lighting management. It is also possible to define custom models, called *Vendor Models*.

The typical models seen in a *lighting* node are the following:

- A Configuration Client/Server, to configure other models' publish/subscribe states.
- A Health Client/Server, to collect data about transmission errors,...
- A Generic OnOff Client/Server, typically linked to buttons and lights. A button will publish on/off to a group (e.g., Kitchen Lights), and lights subscribed to this group will be turned on/off by the button message.

2.2.5 BT Mesh messages

BT Mesh messages contains the typical *source address*, *destination address*, *payload*, a message identifier check, and the TTL used for the *managed flooding*. It uses BLE as its underlying physical layer.

	1		1	3	2	2	12 or 16	4 or 8
≧	Network ID	⌈	TTL	Sequence Number	Source Address	Dest Address	Packet Payload	NWK MIC

Figure 2.8 BT Mesh messages.¹

Every message is encrypted with at least a Network Key, distributed when nodes are provisioned into the network. On top of that, Applications Keys can be added to add a layer of security inside the network. With minimal payload, it is expected that the BT Mesh communication will be a huge bottleneck.

¹Figure from the Silicon Labs' BT Mesh performance analysis[7]

2.2.6 Provisioning

In **BT Mesh**, when a node is not part of any network, it is said to be *unprovisioned*. The act of adding the node to a network is called *provisioning*. Initially, when a node is unprovisioned, it will send *Beaconing* messages through the Advertise and GATT. A special node, called **Provisioner** will see receive those beacons and can *provision* nodes into the net.

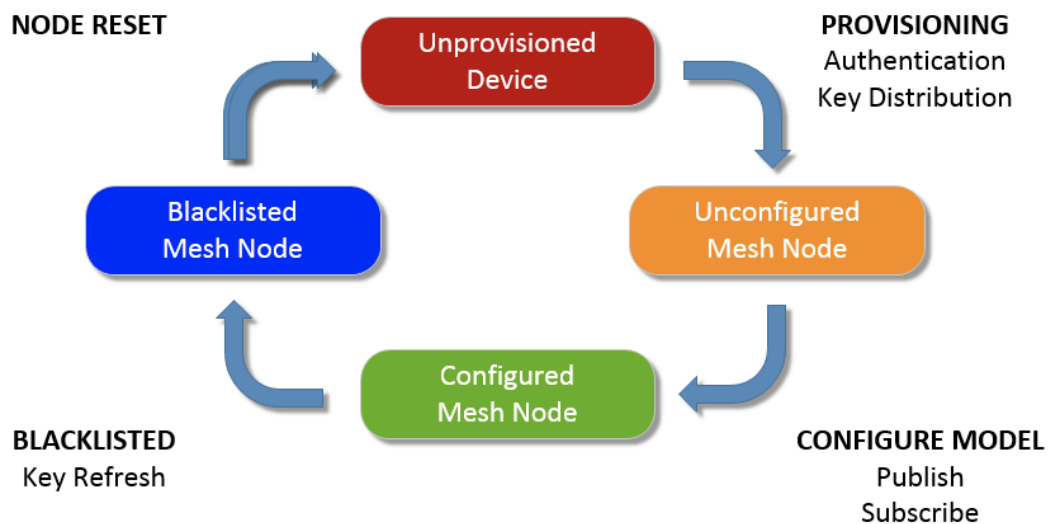


Figure 2.9 **BT Mesh** provisioning: node lifecycle.¹

First, the **Provisioner** will invite the node into the network. An optional authentication called **Out Of Band (OOB)** can be made to ensure we provision the correct node. There are four types of **OOB**:

Ouput OOB where the node generates a number and outputs it to a LED or through a terminal, and the **Provisioner** must confirm this number.

Input OOB, similar to the Output OOB, but with roles reversed. The user must input the number on the node, e.g., by pressing a button multiple times.

Static OOB where both devices generate a random number and proceed to check the confirmation value operation.

No OOB where no authentication is done.

Then the **Provisioner** will send the Net Keys to cipher the communication and can configure the node models.

¹Image source: <https://blog.rtone.fr/bluetooth-mesh>

2.3 Conclusion

2.3.1 ROS

ROS topics, services, and actions are great to separate systems on a big project. For example, when developing a robot with complex navigation, you might want to simulate the robot before destroying the hardware. The *navigation* node does not know that the *LiDAR* data comes from a real *LiDAR* or from a simulation, because it does not need to. Likewise, when sending commands to the wheels, the node can either be a real motor driver or a simulation.



Figure 2.10 Gazebo simulator for **ROS**.¹

Nevertheless, all those systems are cumbersome and unsuitable for **IoT** systems. Even **microROS** is limited, due to its requirement of the **ROS Agent**, which must run on a full **ROS** install and must be always up and running. This requirement also means that every **microROS** node will be connected to this Agent, and this brings us back to the problem described in the introduction (1).

2.3.2 Bluetooth Mesh

BT Mesh is perfect for implementing robust, large, low-power, **IoT** networks. The publish/subscribe pattern can easily be used with **ROS** topics. The mass availability of Bluetooth-capable processors (such as *Nordic Semiconductor's*) makes the making of those Mesh Networks fast and cost-efficient. But using only **BT Mesh** for **M2M** of complex machines, like robots, isn't suitable, as the Mesh Network is limited by its reduced message payload and high latency [8].

2.3.3 Solution chosed

To bring **IoT** sensors to a **ROS** network, a gateway implementing a **microROS** node and a **BT Mesh Provisioner** is a solid choice. This way, the Mesh Network can be fully driven directly through **ROS**, enabling a perfect combination of the high-level capabilities of **ROS**, and the low-power design of **BT Mesh**.

¹Source: <https://gazebo-sim.org/showcase>

3 | Design

This chapter will cover the design concepts of the BLE2ROS gateway, some BT Mesh nodes and the ROS2 node controlling the Mesh provisioner.

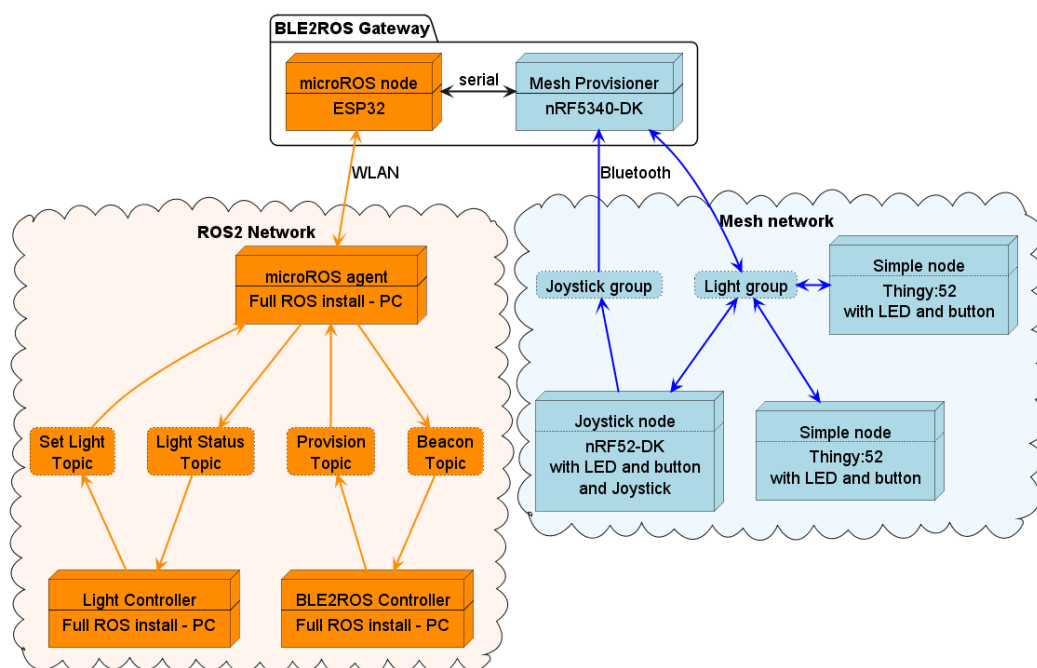


Figure 3.1 BLE2ROS Overview, with the BLE2ROS gateway linking ROS2 and BT Mesh

Figure 3.1 shows how a typical BLE2ROS Network is structured.

Contents

3.1	BLE2ROS gateway	18
3.1.1	Hardware	18
3.1.2	BT Mesh Provisioner	19
3.1.3	microROS node	20
3.1.4	BLE2ROS UART Protocol	21
3.1.5	UART Message emulator	21
3.2	BT Mesh nodes	22
3.2.1	Simple node	22
3.2.2	Sensor node	22
3.3	ROS Node	23
3.3.1	ROS interfaces	23
3.3.2	ROS Node with UI	24

3.1 BLE2ROS gateway

To quickly put to work a portable gateway with simple hardware, the choice was made to use two devices :

- An ESP32 from *Espressif Systems*, with a [microROS](#) implementation, serves as a bridge between ROS2 and the BT Mesh network
- A nRF53 from *Nordic Semiconductor* will be the [Provisioner](#) of the BT Mesh network, receiving commands by the ESP32 to operate the Mesh.

Those two devices will communicate over a serial interface. They both have multiple UARTs to allow this communication.

3.1.1 Hardware

The gateway's hardware is pretty straightforward. An ESP32 serialy connected to an nRF53. *DevBoards* are used to ease the prototyping.

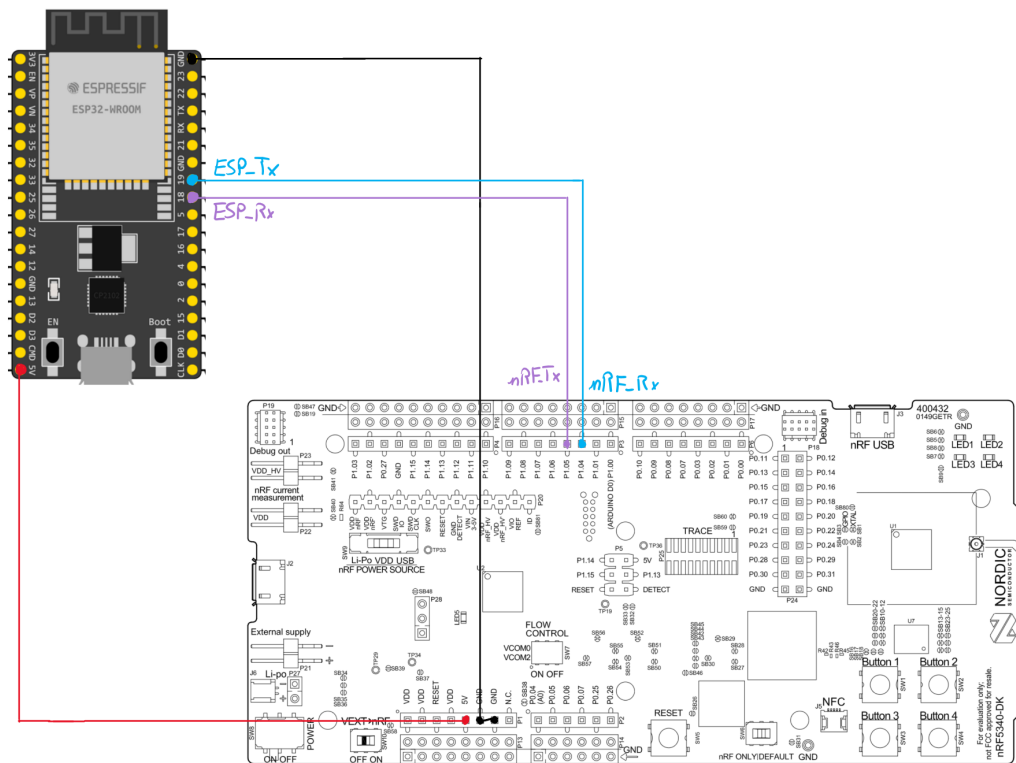


Figure 3.2 Shield layout

A shield for the nRF53 DevBoard will be made. Current comes from the nRF53 programming USB connector and powers both the nRF53 and ESP32.

3.1.2 BT Mesh Provisioner

The [Provisioner](#), running on the nRF53 from *Nordic Semiconductor*, must be able to receive commands through UART and apply them to the Mesh network.

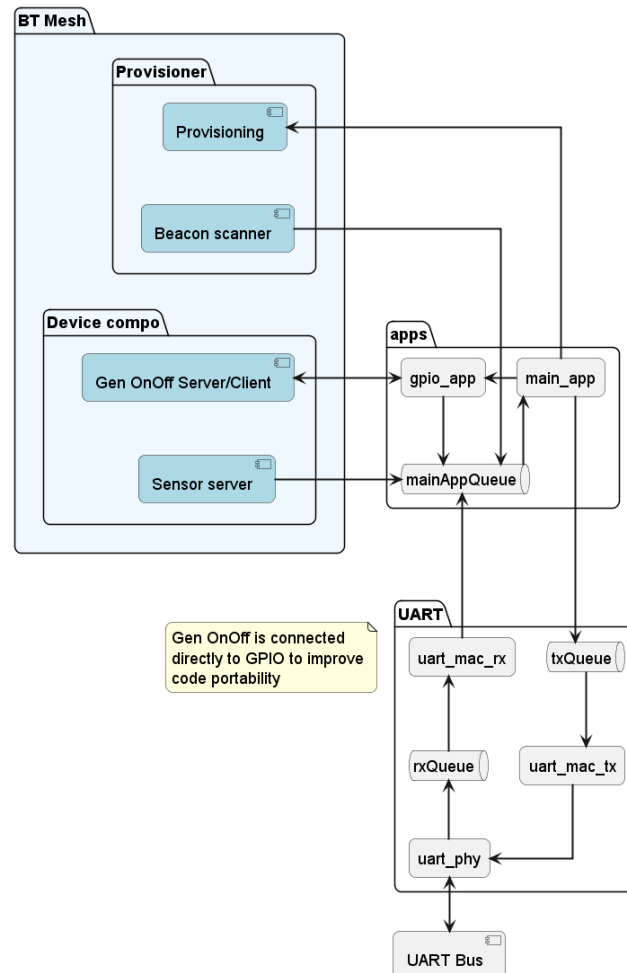


Figure 3.3 BT Mesh provisioner firmware design

Commands received on the UART from the [microROS](#) node (running on the ESP32) are sent to the mainAppQueue. This main app directly accesses the provisioning interface of *Zephyr*. The Generic OnOff server and client are linked to a GPIO app that can control the LED and get signals from the button. The main app can also send the same signal as the button to the GPIO app to command lights on the network. The main app also sends feedback on the current state of the Mesh network to the [microROS](#) node by putting messages in the txQueue. The UART package is composed of three threads:

UART Phy that receives bytes from the UART Bus and sends messages to UART MAC RX

UART MAC RX which receives messages from its queue and dispatch them accordingly

UART MAC TX that gets messages to send from its queue and transmits them using UART Phy interfaces when it is available.

3.1.3 microROS node

The role of the microROS node, implemented on an ESP32, is to bridge data from the [Provisioner](#) (running on the nRF53), coming through the UART Bus and ROS2. By design, it must remain stateless.

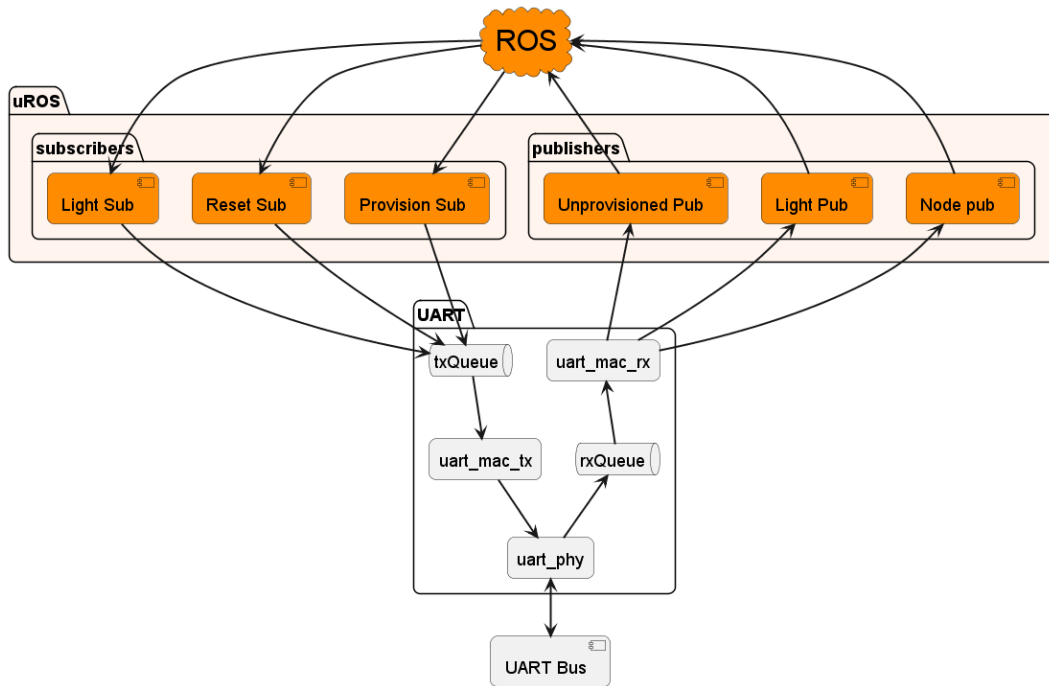


Figure 3.4 *microROS node firmware design*

A [microROS](#) thread with the [RCL Executor](#) runs the publishers and subscribers. Every subscriber has a *callback method* that generates messages and sends them to the [Provisioner](#), using the *txQueue* of the UART MAC TX thread. When a message is received from the [Provisioner](#) via the UART interface, it will get dispatched to the correct publisher with the correct ROS2 message. The UART package is composed of the same three threads as the BT Mesh provisioner ([3.1.2](#)).

To enable complete control of the [BT Mesh](#) network from [ROS](#), this node must implement, at least, the following publishers/subscribers:

- **Unprovisioned Pub** that published UUID of unprovisioned nodes detected by the [Provisioner](#).
- **Provision Sub** which receives a UUID of a node that the [Provisioner](#) must provision.
- **Node Pub** that publish addresses and info of the currently provisioned nodes on the network.
- **Remove Sub** which receives addresses of provisioned nodes that must be removed from the network.
- **Reset Sub** to remove every currently provisioned node on the network.

3.1.4 BLE2ROS UART Protocol

The protocol between the [Provisioner](#) (running on the nRF53) and the [microROS](#) node (running on the ESP32) will exchange the following frames:

Name	Function	Size
MID	Message identifier	1 byte
Length	Length of the payload	1 byte
Payload	Message payload	'Length' bytes
CRC	Checksum of the message	1 byte

Table 3.1 BLE2ROS UART Protocol frame

The *MID* contains an identifier about the type of the message, it can be one of the following (Non-exhaustive list):

Name	Function	ID
CMD_PING	Ping with a payload	0x01
RSP_PING	Ping response with same payload	0x02
CMD_RESTART	Restarts the Provisioner	0x0A
CMD_RESET	Resets the BT Mesh network and Provisioner	0x0B
INVALID_CRC	Message received had an invalid CRC	0xFB
INVALID_MID	Message received had an invalid MID	0xFD
INVALID_LEN	Message received had an invalid length	0xFF
UNPROV	Unprovisioned beacon UUID	0x10
PROV	Command to provision node with UUID	0x11
NODE	Provisioned node UUID	0x12
ONOFF	Command to the OnOff client or OnOff server status	0x20
JOYSTICK_DATA	Joystick data from the BT Mesh network (X and Y)	0x30

Table 3.2 UART MIDs

3.1.5 UART Message emulator

To quickly test both UARTs of the [microROS](#) node and [BT Mesh Provisioner](#), a simple program to emulate message . Python and Jupyter were chosen for this tester to allow quick prototyping of UART functionalities.

3.2 BT Mesh nodes

In order to examine the boundaries of the BLE2ROS network, such as the maximum data rate, different BT Mesh nodes must be designed.

3.2.1 Simple node

A simple node is composed of at least one button and one LED. It will provide two BT Mesh models :

- **A Generic On/Off client**, linked with the button
- **A Generic On/Off server**, linked with the LED

Hardware

To quickly put to work, multiple simple nodes, *Nordic's Thingy:52* were chosen.

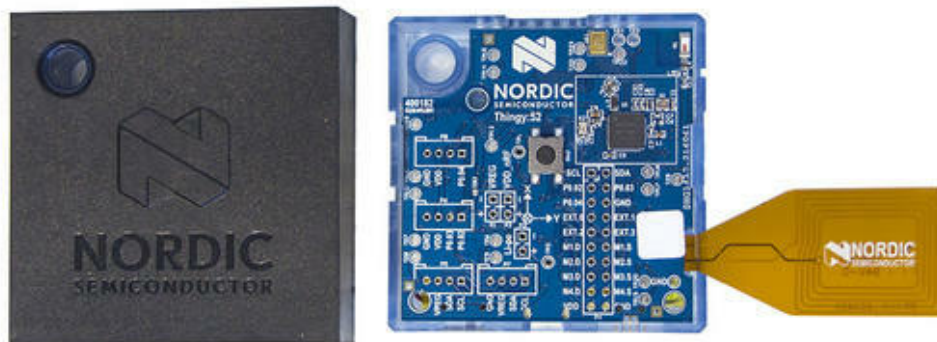


Figure 3.5 *Nordic Thingy:52 Prototyping platform*

Nordic Semiconductor provides firmware examples to evaluate their BT Mesh stack. This will avoid the extra cost of writing firmware during first tests.

3.2.2 Sensor node

A sensor node implements the same models as a simple node but with at least one sensor, publishing data periodically over the BT Mesh network. The typical composition of such a node is the following :

- **A Generic On/Off client**, linked with a button
- **A Generic On/Off server**, linked with a LED
- **A custom Sensor client Vendor model**, publishing a joystick's or accelerometer's X and Y positions

This node can either be a *Thingy:52*, transmitting its accelerometer data, or another *Nordic* board with a joystick.

3.3 ROS Node

A [ROS](#) node, controlling the Mesh network from [ROS](#) will be made. It will help for demonstrations and testing of the BLE2ROS gateway.

3.3.1 ROS interfaces

To communicate with the [microROS](#) node, we need to define custom [ROS](#) interfaces (messages and services).

Services

The only service will be a ping service to test the full communication [ROS](#) -> [microROS](#) -> [Provisioner](#) -> [microROS](#) -> [ROS](#)

Req/Rsp	Name	Type
Request	Ping	string
Response	Pong	string

Table 3.3 Ping service

This service sends a ping request to the [microROS](#) node, which will transmit a *UART* message with *MID* CMD_PING and the ping string as payload. Then when the [microROS](#) node receives the RSP_PING, it will transmit the response to [ROS](#) with the payload as the pong string.

Topics

[ROS](#) topics will be used to send and receive data from the [BT Mesh](#) network. The [ROS](#) interfaces for these topics will carry data such as sensor data, OnOff data and UUIDs from provisioned and unprovisioned nodes.

Msg Name	Data Name	Data Type
joystick	x	int16
	y	int16
lighting	on_off	bool
mesh_uuid	uuid	uint8[16]

Table 3.4 [ROS](#) messages

Here, the sensor will be a joystick, transmitting his X and Y data. A lighting message is used to control lights in the [BT Mesh](#) network and to receive the current state of the lights. [BT Mesh](#) UUID is used to receive unprovisioned beacons detected by the [Provisioner](#), to order provisioning of those beacons, and to receive currently provisioned nodes of the network.

3.3.2 ROS Node with UI

An UI is planned to ease the usage of the [ROS](#) node. It will give an overview of the current state of the [BT Mesh](#) network, and implements buttons to control the network.

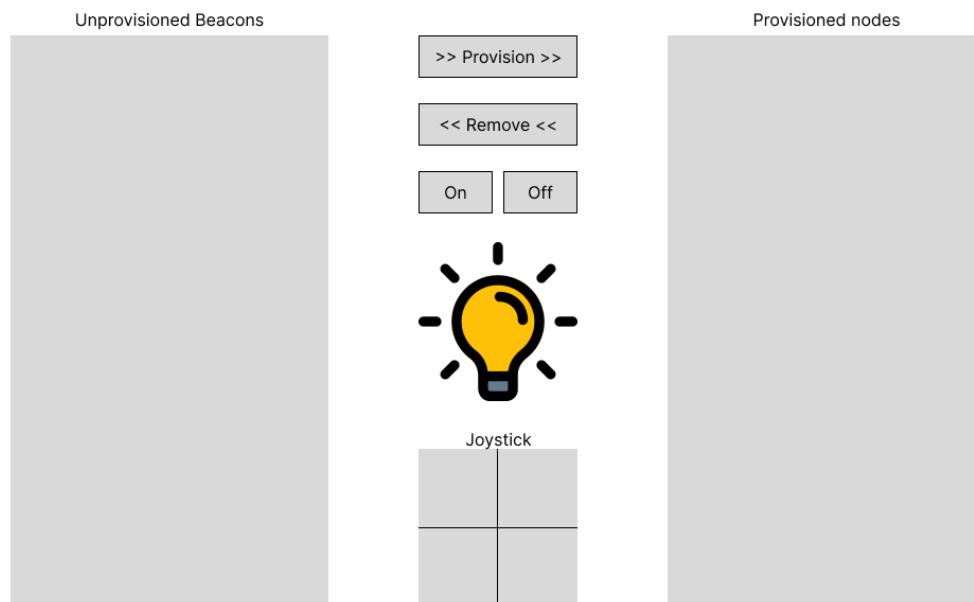


Figure 3.6 BLE2ROS UI Design

It will be composed of two lists: one for the unprovisioned beacons, detected by the [Provisioner](#), and another for the currently provisioned nodes.

4 | Implementation

This chapter will cover details about the hardware, firmware, and software implementations of every system made during the thesis. Those are the following:

- UART Communication
- ROS Communication
- BT Mesh nodes
- BT Mesh provisioning
- ROS node

Contents

4.1	UART Communication	26
4.1.1	UART MAC Threads	26
4.1.2	UART MAC TX	27
4.1.3	UART MAC RX	27
4.1.4	UART PHY	28
4.1.5	UART Message emulator	29
4.2	BT Mesh nodes	30
4.2.1	Simple nodes	30
4.2.2	Sensor node	31
4.3	BT Mesh Provisioner	32
4.3.1	BLE2ROS provisioning	33
4.4	microROS node	34
4.5	ROS node with UI	35

4.1 UART Communication

The first part implemented was the UART communication. It is the basis on which the gateway will be built. As a reminder, here is the design of the *UART* layers implemented for both the [microROS](#) (ESP32) and the [Provisioner](#) (nRF53 DevBoard):

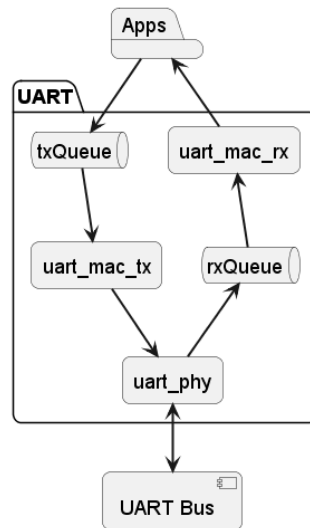


Figure 4.1 UART Layers

4.1.1 UART MAC Threads

Messages exchanged to and from the **UART MAC Threads** are the following:

```

typedef struct _st_ble2ros_uart_msg {
    uint8_t mid;
    uint8_t length;
    uint8_t payload[BLE2ROS_UART_PAYLOAD_BUFF_SIZE];
    uint8_t crc;
} st_ble2ros_uart_msg;
  
```

Listing 4.1 UART message structure

This structure is a direct implementation of the definition [3.1](#).

4.1.2 UART MAC TX

The **UART MAC TX** thread is a simple loop, waiting for messages in his queue, and sending them to **UART PHY**.

```
void uart_tx_thread(void* arg) {
    int err = 0;
    st_ble2ros_uart_msg txMsg;
    txQueue = createQueue(MAX_MSG_QUEUE_ELEMENTS, sizeof(st_ble2ros_uart_msg));

    while (1) {
        if (queueReceive(txQueue, &txMsg, WaitForever)) {
            err = uart_send(txMsg);
            if (err == UART_BUSY) {
                WARNING("uart busy, trying later...");
                to_mac_tx(txMsg); // Puts the message back into the queue
            } else if (err) {
                ERROR("Failed to transmit message");
            }
        }
    }
    taskDelete();
}
```

Listing 4.2 UART MAC TX Thread pseudocode

4.1.3 UART MAC RX

When the **UART PHY** receives a full message, it will send it to the **UART MAC RX** thread via its *rxQueue*. The thread will then dispatch the message to the apps.

```
void uart_rx_thread(void* arg) {
    st_ble2ros_uart_msg rxMsg;
    rxQueue = createQueue(MAX_MSG_QUEUE_ELEMENTS, sizeof(st_ble2ros_uart_msg));
    while (1) {
        if (queueReceive(rxQueue, &rxMsg, WaitForever)) {
            if (rxMsg.crc == calc_crc(rxMsg)) {
                switch (rxMsg.mid) {
                    case BLE2ROS_UART_GENERAL_MID_CMD_PING:
                        cmd_ping(rxMsg);
                        break;
                    case BLE2ROS_UART_GENERAL_MID_RSP_PING:
                        to_ping_srv(rxMsg);
                        break;
                    ...
                }
            } else {
                WARNING("Bad CRC! Expected: %d, Received: %d", calc_crc(rxMsg),
                    ↪ rxMsg.crc);
                send_bad_crc();
            }
        }
    }
    taskDelete();
}
```

Listing 4.3 UART MAC RX Thread pseudocode

Both implementations are pretty similar, except from the **RTOS** functions. Specific code and **RTOS** functions can be viewed at appendix A.

4.1.4 UART PHY

The **UART PHY thread** must implement the specific interface from each system. Both drivers will use 115200 baud rate, 8 data bits, 1 stop bit, no parity, and no flow control.

BT Mesh Provisioner

On the nRF53, the *UART Asynchronous API*[\[9\]](#) was chosen, because it doesn't rely on *interrupts* and never stops code execution. It uses the processor *DMA*[\[10\]](#) to allow a higher level, uninterrupting interface for the *UART Peripheral*. For the reception, a timeout must be defined. When data starts incoming, the timeout starts ticking. Every time a new byte is received, the timeout is reset. This is useful to isolate UART messages, there will always be some time between two complete frames, and the timeout can be adjusted accordingly. This *API* uses a *callback method* running on the main thread, which is called when an event happens:

UART_TX_DONE: the whole TX buffer was transmitted.

UART_TX_ABORTED: transmission aborted due to timeout or abort the call.

UART_RX_RDY: Some data was received, and the RX timeout occurred, or the buffer is full. At this moment, the *RX* must be disabled and will emit the *UART_RX_DISABLED* event. If *RX* is not disabled, the *API* will keep appending data at the end of the TX buffer, and eventually, it will overflow. Disabling and re-enabling it will force the *API* to always store data at the beginning of the buffer.

UART_RX_BUF_REQUEST: *Driver* requests a second buffer for continuous reception.

UART_RX_BUF_RELEASED: Buffer is no longer used by *UART driver*.

UART_RX_DISABLED: Event generated when the receiver was disabled or finished its operation and can be enabled again.

UART_RX_STOPPED: Receiver was stopped due to an external event.

Complete implementation of this *API* can be found at [appendix B](#).

microROS node

On the ESP32, the *UART API* from *ESP-IDF* [\[11\]](#) was used. This *API* is well documented, and everything can be found on the [ESP-IDF Programming Guide](#). Full code implementation can be found at [appendix C](#).

4.1.5 UART Message emulator

This emulator's full code implementation can be found in appendix [D](#). The test made was a simple Ping-Pong; the tester sends a Ping frame with a text payload and must receive a Pong frame with the same text payload. When testing both the ESP32 and nRF53, the result was as expected:

```
1  -----
2  BLE2ROS Frame
3  MID: 0x01 CMD_PING
4  Length: 14
5  Payload: Hello friend !
6  CRC: 220
7  -----
8  -----
9  BLE2ROS Frame
10 MID: 0x02 RSP_PING
11 Length: 14
12 Payload: Hello friend !
13 CRC: 221
14 -----
```

Listing 4.4 *UART Message emulator Ping test results*

It validates that the serial interface is working and ready to use with the applications.

4.2 BT Mesh nodes

The **BT Mesh** node implementation is pretty straight forward, every model is linked with an application.

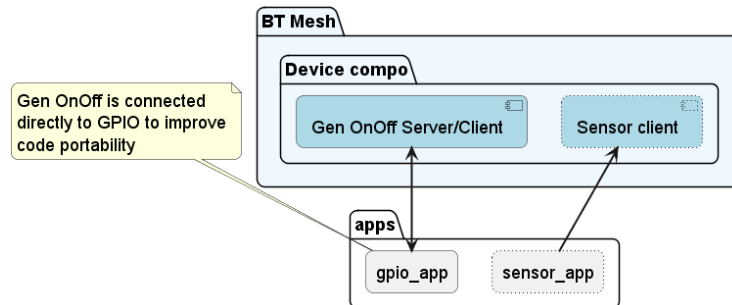


Figure 4.2 Mesh Node firmware architecture

Both node types share the same codebase; the sensor model and sensor app are built if the compiler detects that the device is not a *Thingy:52*. The full code for the nodes can be found in the GitHub repo: <https://github.com/SamyFrancelet/ble2ros>.

4.2.1 Simple nodes

Multiple simple nodes were programmed and provisioned by the *nRF Mesh smartphone app*.



Figure 4.3 First mesh network, using multiple Thingy:52

Every node was provisioned and configured properly, their Gen OnOff clients published data to the *Lights* group, and the Gen OnOff servers subscribed to that group. This demo also showed that, when provisioned, nodes can perfectly work without the provisioner up and running on the network.

4.2.2 Sensor node

The sensor used was a simple video game controller, joystick. Both the X and Y axis are connected to ADCs, and the ADC driver publishes data to a group.

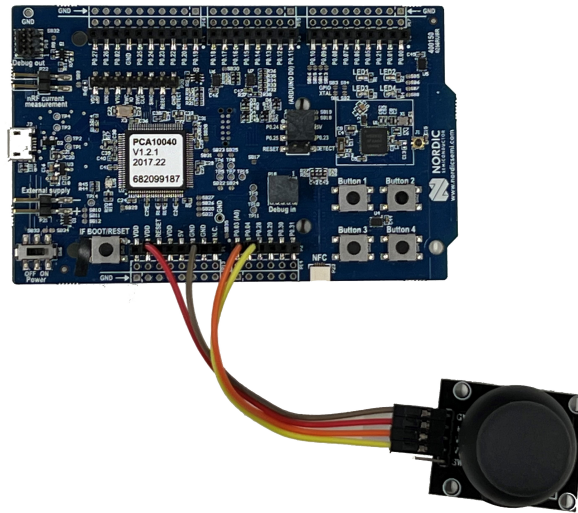


Figure 4.4 Sensor node hardware, using a nRF52DK-nRF52832

The joystick application is an infinite loop, with a delay between each ADC conversion. This allows a periodic, manageable, published data rate. The four buttons on this DevBoard can be used to increase/decrease the current data rate and data size to test the maximum capabilities of a single BT Mesh node.

4.3 BT Mesh Provisioner

The provisioning process is the following:

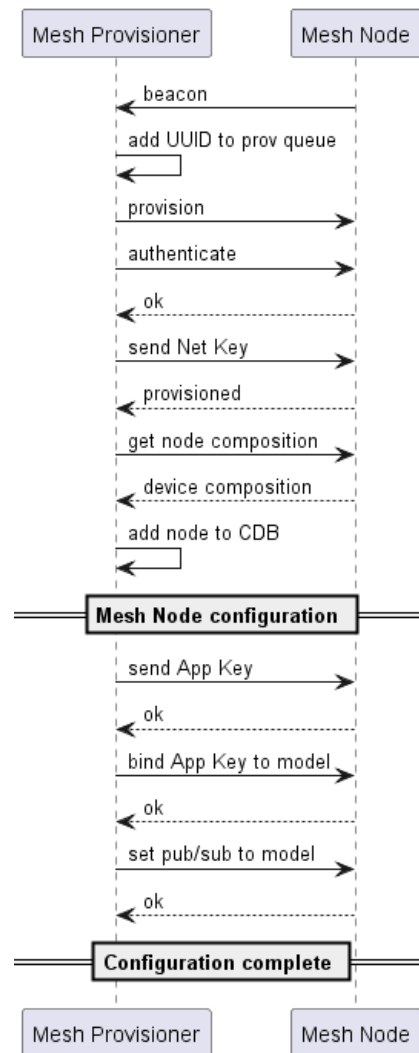


Figure 4.5 Basic provisioning sequence diagram from the Zephyr *BT Mesh* provisioning example¹

The provisioner will receive *Beacons* from the unprovisioned nodes and start the provisioning process. First, the provisioner will authenticate the node if there is an *OOB*. Then it will send the network key to the node and add the node to its *Configuration DataBase* (CDB). The *BT Mesh* provisioner thread will configure every new node at each iteration, and mark them as configured when successful.

¹Zephyr *BT Mesh* provisioning example: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/2.0.0/zephyr/samples/bluetooth/mesh_provisioner/README.html

4.3.1 BLE2ROS provisioning

The implementation of the provisioning process from ROS is the following:

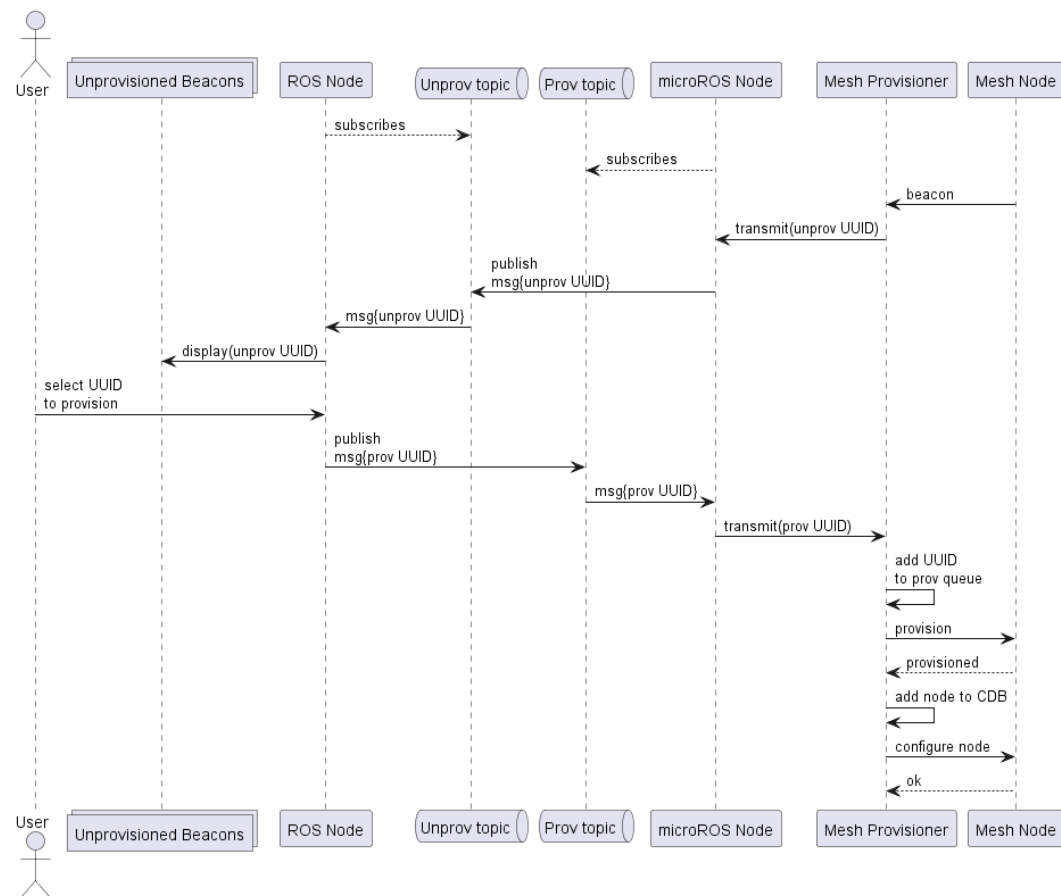


Figure 4.6 BLE2ROS provisioning sequence diagram

When a *Beacon* is received, the node UUID is sent to the **microROS** node with the serial interface. The UUID will be published to **ROS** and the implemented ROS Node will display it to the user. The user will have a list of every detected unprovisioned node and can select one for provisioning. The to-be provisioned node UUID will be transferred to the provisioner, starting the provisioning process described at 4.5. The full code implementation of this provisioner can be found in the GitHub repo: <https://github.com/SamyFrancelet/ble2ros>.

4.4 microROS node

The `microROS` node is implemented as described in the design chapter (3.1.3), and the code can be found in the GitHub repo: <https://github.com/SamyFrancelet/ble2ros..>

The `microROS Component for ESP-IDF`[12] was used to implement the `microROS` library on the ESP32. It should be kept in mind that by default, this component only allows one client/service, Two publishers and two subscribers. This can be modified in the `colcon.meta` file of the component:

```
1  "rmw_microxrcedds": {
2      "cmake-args": [
3          "-DRMW_UXRCE_XML_BUFFER_LENGTH=400",
4          "-DRMW_UXRCE_TRANSPORT=udp",
5          "-DRMW_UXRCE_MAX_NODES=1",
6          "-DRMW_UXRCE_MAX_PUBLISHERS=2",
7          "-DRMW_UXRCE_MAX_SUBSCRIPTIONS=2",
8          "-DRMW_UXRCE_MAX_SERVICES=1",
9          "-DRMW_UXRCE_MAX_CLIENTS=1",
10         "-DRMW_UXRCE_MAX_HISTORY=1"
11     ]
12 }
```

Listing 4.5 `colcon.meta` configuration for Micro XRCE-DDS

4.5 ROS node with UI

The ROS node with UI, and ROS interfaces are implemented as described in the design chapter (3.3) and the full code can be found in the GitHub repo: <https://github.com/SamyFrancelet/ble2ros>.

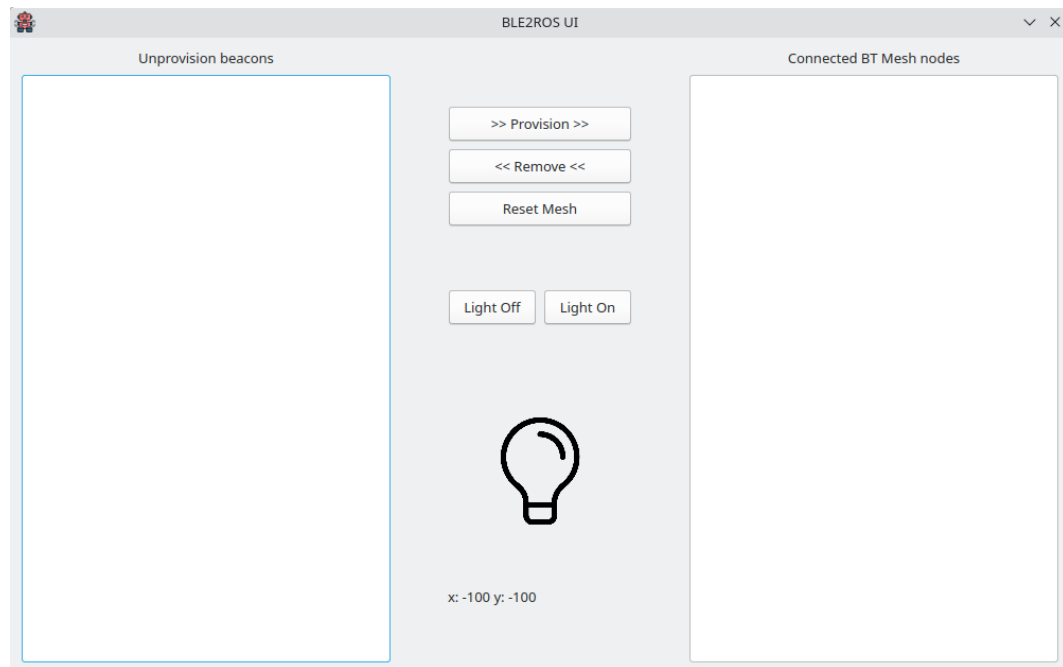


Figure 4.7 BLE2ROS UI Implementation with PyQt

Qt, with its implementation PyQt, was used to make the UI. The ROS Node is part of this UI and linked using a subject/observer pattern.

5 | Validation

This chapter will provide proof of the functionalities implemented, and evaluate the boundaries of this solution.

Contents

5.1	Data through ROS and BT Mesh	38
5.2	Solution's boundaries	39
5.2.1	Goals and Methods	39
5.2.2	Experiments results	41
5.3	Data rate between BT Mesh Network and ROS	41

5.1 Data through ROS and BT Mesh

The BLE2ROS gateway makes provision from the ROS node UI possible.

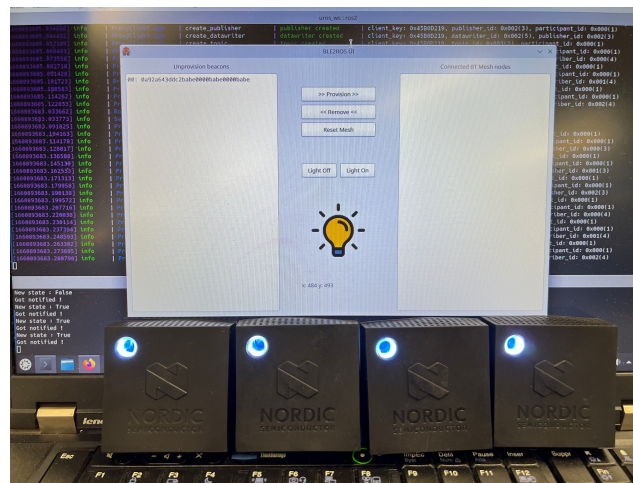


Figure 5.1 BT Mesh nodes, provisioned from ROS, with joystick data and lighting feedback

When provisioned, the following messages can be exchanged between BT Mesh and ROS:

- Lighting messages, controlling all lights in the BLE2ROS network
- Joystick messages, coming from BT Mesh to ROS, illustrating the capabilities of sensors in the Mesh

The UI light icon is synchronised with the Gen OnOff server from the Provisioner.

5.2 Solution's boundaries

5.2.1 Goals and Methods

The main concern of this thesis is the maximum data rate and data size that can be exchanged between the [BT Mesh](#) Network and [ROS](#). [BT Mesh](#) is the main bottleneck here because of two factors. The first is that, compared to other Mesh Networks, [BT Mesh](#) is limited in terms of *throughput (i.e. data rate)*, around 1000[*bits/s*][8]. Second, every data must pass through the [Provisioner](#) to be sent to [ROS](#). This means that the maximum data rate received by the [Provisioner](#) must be evaluated.

To evaluate this maximal data rate, two experiments have been made:

- Data rate vs data size, with a single publisher
- Data rate vs number of node, with a fixed data size

Data rate vs data size

The setup to evaluate the data rate vs data size is the following:

- The [Provisioner](#) has a service vendor model, ready to accept messages from 0 bytes to 8 bytes
- A node, with a client vendor model, will publish periodically messages from 0 to 8 bytes:
 - Two buttons will be used to increment/decrement the periodicity of the messages published
 - Two other buttons used to increment/decrement the message data size

This will show the limits of a continuous data rate coming from a sensor.

The influence of two other parameters will also be evaluated:

- The relaying of messages by the provisioner
- The acknowledgment of messages

Data rate vs number of nodes

To evaluate the continuous data rate boundary coming from multiple nodes, the nodes will periodically publish a 1-byte message, using the same vendor model and a synchronized periodicity.

The nodes used to evaluate this boundary are the following:

- A nRF52 DevBoard will drive the periodicity of the messages, using the Gen OnOff client
- Multiple Thingy:52 will publish the same message periodically, dictated by the nRF52, until the [Provisioner](#) is overwhelmed

This will show the limits of a continuous data rate coming from multiple sensors.

The influence of two other parameters will also be evaluated:

- The relaying of messages by the provisioner
- The acknowledgment of messages

Review of other analysis

Other evaluation of the performance of [Bluetooth Mesh Networking \(BT Mesh\)](#) focus on the data latency over different network size and data size [7][13] and none evaluate the maximum data rate received by one node. Our application is really specific, and the relative novelty of [BT Mesh](#) makes data sparse.

5.2.2 Experiments results

Data used in this report can be found in appendix E

Data rate vs data size

Using a relay buffer size of 30 (adjusted with KConfig `CONFIG_BT_MESH_ADV_BUF_COUNT`) and a maximal message segmentation of 32, messages were published until either the publisher or the [Provisioner](#) outputs an error.

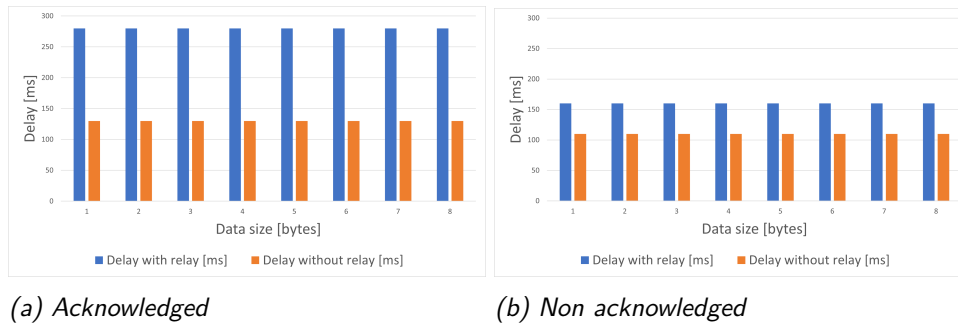


Figure 5.2 Minimum period between messages vs data size

The data size has no impact perceivable by this setup. The main limitations are the acknowledgment and relay of messages.

From this data and the way this experiment was set up, it is possible to conclude that the limiting factor for periodic data reception is the network buffers. When using lower periods between messages, the reception worked for a short time, until the device was out of network buffers. Only when using the minimum period the reception could stabilize and stop outputting errors. This means that the minimum period between messages represent the time a message takes to be fully processed by the *Zephyr BT Mesh API*.

Data rate vs number of nodes

Difficulties making the setup work and inconsistency of the data measured made this experiment unusable. It was chosen to not include the data, as it is irrelevant and can lead to misinterpretations. The nodes must relay the messages, otherwise it is impossible to be certain that every data are received by the provisioner. But this relaying of many messages makes the network unstable and causes crashed, that can't be monitored on the Thingy:52.

5.3 Data rate between BT Mesh Network and ROS

Unfortunately, this could not be evaluated due to a bug in the *microROS Component for ESP-IDF*[12]. When publishing from other threads than the *main thread* and *microROS thread*, the ESP32 will sometimes crash. This is due to the multi-threading capabilities of the *microROS Component for ESP-IDF* being impossible to enable.[14] This problem is accentuated when publishing data more frequently, making this point impossible to evaluate to this day.

6 | Conclusions

6.1 Project summary

Most objectives have been reached. ROS can partially control the Mesh Network. New Mesh groups currently cannot be created, new ROS topics cannot be created, but nodes can be provisioned from ROS, data can flow between ROS and the Mesh Network. Although currently, the microROS library has a bug that crashes and reboots the device, the microROS node is stateless, making its use annoying but possible.

The primary data rate bottleneck is, as expected, the Mesh Network. Every data must pass through the [Provisioner](#), and given the limited data throughput of the underlying [BLE](#) physical layer, it comes as no surprise that the continuous data rate is really limited.

6.2 Comparison with the initial objectives

Objective	Achieved	Comment
Evaluate the platform	Yes	-
Create Mesh nodes	Yes	Using multiple hardware
Create microROS node	Yes	-
Create Provisioner	Yes	-
ROS <-> Mesh communication	Yes	-
Provisioning from ROS	Yes	-
Creating new BT Mesh groups from ROS	Not done	Lack of time
Creating new ROS topics linked to Mesh groups	Not done	Lack of time
Deleting Mesh nodes from persistent storage	No	With the current Zephyr Mesh API it seems to be impossible to remove properly mesh nodes from CDB
Examine BT Mesh boundaries	Partially	BT Mesh data rate is the main bottleneck
Examine BLE2ROS boundaries	Not done	microROS contains a bug, making this step impossible

Table 6.1 Comparisons with initial objectives

6.3 Encountered difficulties

6.3.1 UART

When starting the development, the initial priority was the UART. The goal was to transmit a BLE2ROS frame with the Ping command and a character string and receive a Ping response with the same string.

On the nRF53, the UART1 was used, with the default pins (P1.00 and P1.01). Everything worked great, but when adding the support for BT Mesh on the nRF53 config file, the UART1 was no longer available, and when reading the pins, they always printed the string "*** Booting Zephyr OS build v3.0.99-ncs1 ***".

The nRF5340-DK board comes with three Virtual COM ports, and those COM ports are linked to pins, not UARTs. The UART1 was switched to other pins for testing, and when using the Python script for UART testing and a USB UART Dongle, nothing was read from the new pins. However, the UART1 could still receive the frame. The problem was that the USB UART Dongle could not read the nRF53 UART levels (around 3V). And when used directly with the ESP32, everything worked as expected.

No documentation was found about using those pins (P1.00 and P1.01) by Bluetooth.

6.3.2 Hardware

The UART problem made the initial *Shield Design* unusable. Medard told me to redo the schematic and send it to one of his interns. They made the routing and manufactured the PCB. Nevertheless, the board has some layout problems and some short circuits.

6.3.3 microROS

microROS setup utility

When working with microROS, the recommended utility to build nodes is the *microROS setup utility*[15]. However, when working with *ROS humble* (the current version with Ubuntu22.04), the compilation of ROS Python packages generates warnings that are interpreted as errors by the *colcon build tool*. These warnings occur because the tool used to "compile" Python code is the deprecated *setup.py*. This makes the *microROS setup utility* unusable and made me switch to the *microROS Component for ESP-IDF*[12], which is easier to use and compiles correctly.

ROS Messages

When testing the basics of microROS using standard int32 publishers and subscribers, everything worked as expected. But when using custom ROS messages with Strings or arrays of data, the developer must create a buffer and allocate memory for those arrays. It seems logical when working with embedded systems, but I did not think about it directly, costing me time.

Limit of publishers, subscribers, and services

By default, the microROS component for ESP-IDF comes with a max limit of Two publishers, two subscribers and one service. This limit can be changed on the *colcon.meta* file of the component. Documentation on the matter isn't available, except on the *Github Issues* of the repository.

Publishing causes crash of the ESP32

When publishing many messages from the ESP32, it sometimes causes crashes and restarts the system.

I submitted a *Github Issue*[\[14\]](#) to the microROS ESP-IDF component repository, and the maintainer **pablogs9** said that I should enable the multi-threading profile on the *colcon.meta* file. However, when doing so, the microROS library no longer finds the *FreeRTOS' headers*.

The maintainer is still working on this problem and has not provided feedback as of today.

6.3.4 Zephyr Documentation

When developing the [BT Mesh Provisioner](#), I had trouble finding every details to make the provisioner work and to correctly use Mesh models. The documentation is missing many details because it is generated from *C Header files*, and most valuable comments can be found in the *C Source files*. This made the first developments really slow, but thanks to **Thierry Hischier**, who sent me documents to help me get started, I was able to get back on the work quickly.

6.4 Future perspectives

6.4.1 Implementing the full [BT Mesh](#) management from ROS

Actually, it is impossible to create new groups and make nodes subscribe to those groups from ROS. It is also impossible to remove nodes properly from the provisioner [CDB](#) persistent storage. Implementing those features will be straightforward, as the UART Interface is flexible and easy to use.

6.4.2 Evolution of the microROS middleware for ESP32

As of today, microROS implements the XRCE-DDS standard as its middleware.

"It was chosen to allow high performance on low resources devices, it is multi-platform, compiler dependencies free and easy to use. XRCE-DDS isn't built for a specific transport protocol as Serial or UDP and gives the possibility of implementing the needed transport easily." [\[16\]](#)

But the microROS team is implementing a new experimental middleware for microROS [\[17\]](#). This new middleware, *embeddedRTPS*, is really interesting because it allows microROS to speak natively with ROS2, with **no agents**.

A | UART MAC - Zephyr and ESP-IDF

A.1 UART MAC TX - Zephyr

A.1.1 Header

```
/**
 * @file          uart_mac_tx.h
 * @company       HES-SO//Valais-Wallis, CH-1950 Sion
 * @author        Sammy Francelet
 * @version       0.1
 * @date          30. June 2022
 * @brief         Higher level uart tx thread
 */

/* @attention
 *
 */

*/

#ifndef UART_MAC_TX_H
#define UART_MAC_TX_H

/*****
INCLUDES
*****/

#ifdef __cplusplus
extern "C" {
#endif

/*****
DEFINES
*****/

/*****
TYPEDEFS
*****/
typedef struct _st_ble2ros_uart_msg st_ble2ros_uart_msg;

/*****
EXTERNAL VARIABLES
*****/

/*****
DEFINITIONS
*****/
void uart_tx_thread(void);

void send_light_state(bool state);
void send_joystick_data(int16_t x, int16_t y);

void to_mac_tx(st_ble2ros_uart_msg msg);

/*****
END
*****/

#ifdef __cplusplus
}
#endif
#endif
```

```
#endif //UART_MAC_TX_H
```

Appendix A. UART MAC - Zephyr and ESP-IDF

A.1.2 Source

```
/**
 * @file          uart_mac_tx.c
 * @company       HES-SO/Valais-Wallis, CH-1950 Sion
 * @author        Sammy Francelet
 * @version       0.1
 * @date          30. June 2022
 * @brief         Higher level uart tx thread
 *
 * @attention
 *
 */

/*****
INCLUDES
*****/
#include <logging/log.h>

#include "uart_mac_tx.h"
#include "uart.h"

/*****
DEFINES
*****/
#define LOG_MODULE_NAME uart_mac_tx
LOG_MODULE_REGISTER(LOG_MODULE_NAME, LOG_LEVEL_DBG);

/*****
MACROS
*****/

/*****
TYPEDEFS
*****/

/*****
PROTOTYPES
*****/

/*****
LOCALS
*****/
K_MSGQ_DEFINE(txQueue, sizeof(st_ble2ros_uart_msg),
↳ BLE2ROS_UART_MAX_MSG_QUEUE_ELEMENTS, 4);

/*****
DEFINITIONS
*****/
void uart_tx_thread(void) {
    int err = 0;
    st_ble2ros_uart_msg txMsg;

    LOG_INF("UART TX Thread -> Start infinite loop...");
    while (1) {
        err = k_msgq_get(&txQueue, &txMsg, K_FOREVER);

        if (!err) {
```

```

        LOG_DBG("Must send message :\n\rMID: %d\n\rlength: %d\n\rCRC: %d",
                txMsg.mid, txMsg.length, txMsg.crc);

        err = uart_send(txMsg);
        if (err == -EBUSY) {
            LOG_WRN("uart busy, trying later...");
            to_mac_tx(txMsg);
        } else if (err) {
            LOG_ERR("Failed to transmit message");
        }
    }
}

void send_light_state(bool state) {
    LOG_DBG("Sending light state %d", state);
    st_ble2ros_uart_msg msg;

    msg.mid = BLE2ROS_UART_ONOFF_MID;
    msg.length = 1;
    msg.payload[0] = (uint8_t) state;
    msg.crc = calc_crc(msg);

    to_mac_tx(msg);
}

void send_joystick_data(int16_t x, int16_t y) {
    st_ble2ros_uart_msg msg;

    msg.mid = BLE2ROS_UART_JOYSTICK_MID;
    msg.length = 4;
    memcpy(&msg.payload[0], &x, 2);
    memcpy(&msg.payload[2], &y, 2);
    msg.crc = calc_crc(msg);

    to_mac_tx(msg);
}

void to_mac_tx(st_ble2ros_uart_msg msg) {
    LOG_DBG("Sending message to tx thread :\n\rMID: %d\n\rlength: %d\n\rpayload:
    ↪ %s\n\rCRC: %d",
            msg.mid, msg.length, msg.payload, msg.crc);

    while (k_msgq_put(&txQueue, &msg, K_NO_WAIT) != 0) {
        /* message queue is full: purge old data & try again */
        k_msgq_purge(&txQueue);
    }
}

```

A.2 UART MAC RX - Zephyr

A.2.1 Header

```

/**
 ↪ *****
 * @file          uart_mac_rx.h
 * @company       HES-SO/Valais-Wallis, CH-1950 Sion
 * @author        Samy Francelet
 * @version       0.1
 * @date          30. June 2022
 * @brief         Higher level uart rx thread

```

Appendix A. UART MAC - Zephyr and ESP-IDF

```
↳ *****
* @attention
*
↳ *****
*/

#ifndef UART_MAC_RX_H
#define UART_MAC_RX_H

/*****
INCLUDES
*****/

#ifdef __cplusplus
extern "C" {
#endif

/*****
DEFINES
*****/

/*****
TYPEDEFS
*****/
typedef struct _st_ble2ros_uart_msg st_ble2ros_uart_msg;

/*****
EXTERNAL VARIABLES
*****/

/*****
DEFINITIONS
*****/
void uart_rx_thread(void);

void to_mac_rx(st_ble2ros_uart_msg msg);

/*****
END
*****/

#ifdef __cplusplus
}
#endif

#endif //UART_MAC_RX_H
```

A.2.2 Source

```

/**
↪ *****
 * @file                uart_mac_rx.c
 * @company              HES-SO//Valais-Wallis, CH-1950 Sion
 * @author               Samy Francelet
 * @version              0.1
 * @date                 30. June 2022
 * @brief                Higher level uart rx thread
↪ *****
 * @attention
 *
↪ *****
 */

/*****
INCLUDES
*****/
#include <logging/log.h>

#include "uart_mac_rx.h"
#include "uart.h"
#include "app/ble2ros.h"

/*****
DEFINES
*****/
#define LOG_MODULE_NAME uart_mac_rx
LOG_MODULE_REGISTER(LOG_MODULE_NAME, LOG_LEVEL_DBG);

/*****
MACROS
*****/

/*****
TYPEDEFS
*****/

/*****
PROTOTYPES
*****/
static void send_bad_crc();
static void cmd_ping(st_ble2ros_uart_msg pingMsg);

// External function
void to_mac_tx(st_ble2ros_uart_msg msg);

/*****
LOCALS
*****/
K_MSGQ_DEFINE(rxQueue, sizeof(st_ble2ros_uart_msg),
↪ BLE2ROS_UART_MAX_MSG_QUEUE_ELEMENTS, 4);

/*****
DEFINITIONS
*****/
void uart_rx_thread(void) {
    int err = 0;
    st_ble2ros_uart_msg rxMsg;

```

Appendix A. UART MAC - Zephyr and ESP-IDF

```
uart_init(DT_LABEL(DT_NODELABEL(uart1)));

LOG_INF("UART MAC RX Thread -> Start infinite loop...");
while (1) {
    err = k_msgq_get(&rxQueue, &rxMsg, K_FOREVER);

    if (!err) {
        if (rxMsg.crc == calc_crc(rxMsg)) {
            // Correct CRC
            switch (rxMsg.mid)
            {
                case BLE2ROS_UART_GENERAL_MID_CMD_PING:
                    cmd_ping(rxMsg);
                    break;

                case BLE2ROS_UART_PROV_MID_PROV_BEACON:
                    prov_beacon(rxMsg.payload);
                    break;

                case BLE2ROS_UART_ONOFF_MID:
                    if (rxMsg.length >= 1) {
                        ble2ros_pub_light(rxMsg.payload[0]);
                    }
                    break;

                case BLE2ROS_UART_GENERAL_MID_CMD_RESTART:
                    reset_mesh();

                default:
                    break;
            }
        } else {
            // Bad CRC
            LOG_WRN("Bad CRC! Expected: %d, Received: %d", calc_crc(rxMsg),
                ↪ rxMsg.crc);
            send_bad_crc();
        }
    }
}

void to_mac_rx(st_ble2ros_uart_msg msg) {
    LOG_DBG("Sending message to rx thread : \nMID: %d\nlength: %d\npayload:
    ↪ %s\nCRC: %d",
        msg.mid, msg.length, msg.payload, msg.crc);

    while (k_msgq_put(&rxQueue, &msg, K_NO_WAIT) != 0) {
        /* message queue is full: purge old data & try again */
        k_msgq_purge(&rxQueue);
    }
}

static void send_bad_crc() {
    st_ble2ros_uart_msg msg;

    msg.mid = BLE2ROS_UART_GENERAL_MID_RSP_INVALID_CRC;
    msg.length = 0;
    msg.payload[0] = '\0';
    msg.crc = calc_crc(msg);

    to_mac_tx(msg);
}

static void cmd_ping(st_ble2ros_uart_msg pingMsg) {
    st_ble2ros_uart_msg pingRsp;
```



```

    LOG_INF("Ping command with payload : %s", pingMsg.payload);

    pingRsp.mid = BLE2ROS_UART_GENERAL_MID_RSP_PING;
    pingRsp.length = pingMsg.length;
    for (int i = 0; i < pingMsg.length; i++) {
        // Copy payload
        pingRsp.payload[i] = pingMsg.payload[i];
    }
    pingRsp.payload[pingMsg.length] = '\0'; // NULL char for correct logs

    pingRsp.crc = calc_crc(pingRsp);
    to_mac_tx(pingRsp);
}

/*****
END
*****/

```

A.3 UART MAC TX - ESP-IDF

A.3.1 Header

```

/**
↪ *****
 * @file          uart_mac_tx.h
 * @company       HES-SO/Valais-Wallis, CH-1950 Sion
 * @author        Sammy Francelet
 * @version       0.1
 * @date          1. July 2022
 * @brief         Higher level uart tx thread
↪ *****
 * @attention
 *
↪ *****
*/

#ifndef UROS_NODE_UART_MAC_TX_H
#define UROS_NODE_UART_MAC_TX_H

/*****
INCLUDES
*****/
#include <stdint.h>
#include <stdbool.h>

#ifdef __cplusplus
extern "C" {
#endif

/*****
DEFINES
*****/

/*****
TYPEDEFS
*****/
typedef struct _st_ble2ros_uart_msg st_ble2ros_uart_msg;

```

Appendix A. UART MAC - Zephyr and ESP-IDF

```

/*****
EXTERNAL VARIABLES
*****/

/*****
DEFINITIONS
*****/
void uart_tx_thread(void* arg);

void prov_beacon(uint8_t uuid[16]);

void set_light(bool state);

void reset_mesh();

void to_mac_tx(st_ble2ros_uart_msg msg);

/*****
END
*****/

#ifdef __cplusplus
}
#endif

#endif //UROS_NODE_UART_MAC_TX_H

```

A.3.2 Source

```

/**
 * @file          uart_mac_tx.c
 * @company       HES-SO/Valais-Wallis, CH-1950 Sion
 * @author        Sammy Francelet
 * @version       0.1
 * @date          1. July 2022
 * @brief         Higher level uart tx thread
 *
 * @attention
 *
 */

/*****
INCLUDES
*****/
#include "freertos/FreeRTOS.h"
#include "freertos/queue.h"

#include <string.h>

#include "uart_mac_tx.h"
#include "uart.h"

#include "esp_log.h"

/*****
DEFINES
*****/

/*****
MACROS
*****/

/*****
TYPEDEFS
*****/

/*****
PROTOTYPES
*****/

/*****
LOCALS
*****/
static const char* TAG = "uart_tx_thread";

static QueueHandle_t txQueue;

/*****
DEFINITIONS
*****/
void uart_tx_thread(void* arg) {
    st_ble2ros_uart_msg txMsg;

    txQueue = xQueueCreate(BLE2ROS_UART_MAX_MSG_QUEUE_ELEMENTS,
        sizeof(st_ble2ros_uart_msg));

```

Appendix A. UART MAC - Zephyr and ESP-IDF

```
ESP_LOGI(TAG, "UART MAC TX Thread -> Start infinite loop...");
while (1) {
    if (xQueueReceive(txQueue, &txMsg, (TickType_t)portMAX_DELAY)) {
        ESP_LOGD(TAG, "Must send message :\nMID: %d\nlength: %d\npayload:
        ↪ %s\nCRC: %d",
            txMsg.mid, txMsg.length, txMsg.payload, txMsg.crc);

        uart_send(txMsg);
    }
}
vTaskDelete(NULL);
}

void prov_beacon(uint8_t uuid[16]) {
    st_ble2ros_uart_msg msg;

    msg.mid = BLE2ROS_UART_PROV_MID_PROV_BEACON;
    msg.length = 16;
    memcpy(msg.payload, uuid, 16);
    msg.crc = calc_crc(msg);

    to_mac_tx(msg);
}

void set_light(bool state) {
    st_ble2ros_uart_msg msg;

    msg.mid = BLE2ROS_UART_ONOFF_MID;
    msg.length = 1;
    msg.payload[0] = (uint8_t)state;
    msg.crc = calc_crc(msg);

    to_mac_tx(msg);
}

void reset_mesh() {
    st_ble2ros_uart_msg msg;

    msg.mid = BLE2ROS_UART_GENERAL_MID_CMD_RESTART;
    msg.length = 0;
    msg.crc = calc_crc(msg);

    to_mac_tx(msg);
}

void to_mac_tx(st_ble2ros_uart_msg msg) {
    ESP_LOGD(TAG, "Sending message to tx thread :\nMID: %d\nlength: %d\npayload:
    ↪ %s\nCRC: %d",
        msg.mid, msg.length, msg.payload, msg.crc);

    xQueueSend(txQueue, &msg, portMAX_DELAY);
}
```

A.4 UART MAC RX - ESP-IDF

A.4.1 Header

```
/**
↪ *****
* @file          uart_mac_rx.h
* @company       HES-SO//Valais-Wallis, CH-1950 Sion
```

```

* @author          Samy Francelet
* @version         0.1
* @date            1. July 2022
* @brief           Higher level uart rx thread

↳ *****
* @attention
*

↳ *****
*/

#ifndef UROS_NODE_UART_MAC_RX_H
#define UROS_NODE_UART_MAC_RX_H

/*****
INCLUDES
*****/

#ifdef __cplusplus
extern "C" {
#endif

/*****
DEFINES
*****/

/*****
TYPEDEFS
*****/
typedef struct _st_ble2ros_uart_msg st_ble2ros_uart_msg;

/*****
EXTERNAL VARIABLES
*****/

/*****
DEFINITIONS
*****/
void uart_rx_thread(void* arg);

void to_mac_rx(st_ble2ros_uart_msg msg);

/*****
END
*****/

#ifdef __cplusplus
}
#endif

#endif //UROS_NODE_UART_MAC_RX_H

```

A.4.2 Source

```
/**
 * @file          uart_mac_rx.c
 * @company       HES-SO/Valais-Wallis, CH-1950 Sion
 * @author        Sammy Francelet
 * @version       0.1
 * @date          1. July 2022
 * @brief         Higher level uart rx thread
 *
 * @attention
 *
 */

/*****
INCLUDES
*****/
#include "freertos/FreeRTOS.h"
#include "freertos/queue.h"

#include "uart_mac_rx.h"
#include "uart.h"

#include "esp_log.h"

// #include "uros/ping_srv.h"
#include "uros/provision.h"
#include "uros/lights.h"
#include "uros/joystick.h"

/*****
DEFINES
*****/

/*****
MACROS
*****/

/*****
TYPEDEFS
*****/

/*****
PROTOTYPES
*****/
static void cmd_ping(st_ble2ros_uart_msg pingMsg);
static void rcv_unprov_beacon(st_ble2ros_uart_msg msg);
void to_mac_tx(st_ble2ros_uart_msg msg);

/*****
LOCALS
*****/
static const char* TAG = "uart_rx_thread";

static QueueHandle_t rxQueue;

/*****
```

```

DEFINITIONS
*****/
void uart_rx_thread(void* arg) {
    st_ble2ros_uart_msg rxMsg;
    int16_t x;
    int16_t y;

    rxQueue = xQueueCreate(BLE2ROS_UART_MAX_MSG_QUEUE_ELEMENTS,
        ↪ sizeof(st_ble2ros_uart_msg));

    ESP_LOGI(TAG, "UART MAC RX Thread -> Start infinite loop...");
    while (1) {
        if (xQueueReceive(rxQueue, &rxMsg, (TickType_t)portMAX_DELAY)) {
            if (rxMsg.crc == calc_crc(rxMsg)) {
                switch (rxMsg.mid) {
                    case BLE2ROS_UART_GENERAL_MID_CMD_PING:
                        cmd_ping(rxMsg);
                        break;
                    case BLE2ROS_UART_GENERAL_MID_RSP_PING:
                        //to_ping_srv(rxMsg);
                        break;
                    case BLE2ROS_UART_PROV_MID_UNPROV_BEACON:
                        rcv_unprov_beacon(rxMsg);
                        break;
                    case BLE2ROS_UART_ONOFF_MID:
                        if (rxMsg.length >= 0) {
                            light_status(rxMsg.payload[0]);
                        }
                        break;
                    case BLE2ROS_UART_JOYSTICK_MID:
                        memcpy(&x, &rxMsg.payload[0], 2);
                        memcpy(&y, &rxMsg.payload[2], 2);
                        joystick_update(x, y);
                        break;
                    default:
                        break;
                }
            } else {
                ESP_LOGW(TAG, "Bad CRC! Expected: %d, Received: %d",
                    ↪ calc_crc(rxMsg), rxMsg.crc);
                //send_bad_crc();
            }
        }
    }
    vTaskDelete(NULL);
}

static void cmd_ping(st_ble2ros_uart_msg pingMsg) {
    st_ble2ros_uart_msg pingRsp;
    ESP_LOGI(TAG, "Ping command with payload : %s", pingMsg.payload);

    pingRsp.mid = BLE2ROS_UART_GENERAL_MID_RSP_PING;
    pingRsp.length = pingMsg.length;
    for (int i = 0; i < pingMsg.length; i++) {
        // Copy payload
        pingRsp.payload[i] = pingMsg.payload[i];
    }
    pingRsp.payload[pingMsg.length] = '\0';
    pingRsp.crc = calc_crc(pingRsp);
    to_mac_tx(pingRsp);
}

static void rcv_unprov_beacon(st_ble2ros_uart_msg msg) {
    st_mesh_beacon beacon;
    memcpy(beacon.uuid, msg.payload, 16);
}

```

Appendix A. UART MAC - Zephyr and ESP-IDF

```
    unprov_beacon(beacon);
}

void to_mac_rx(st_ble2ros_uart_msg msg) {
    ESP_LOGD(TAG, "Sending message to rx thread :\nMID: %d\nlength: %d\npayload:
    ↪ %s\nCRC: %d",
              msg.mid, msg.length, msg.payload, msg.crc);

    xQueueSend(rxQueue, &msg, portMAX_DELAY);
}
```


B | UART Physical - Zephyr

B.1 Header

```

/**
 * @file          uart.h
 * @company       HES-SO//Valais-Wallis, CH-1950 Sion
 * @author        Samy Francelet
 * @version       0.1
 * @date          30. June 2022
 * @brief         Contains interfaces to access the Zephyr UART Async API
 *
 * @attention
 *
 */

#ifndef UART_H
#define UART_H

/***** INCLUDES *****/
#include <drivers/uart.h>

#include "uart_mac_rx.h"
#include "uart_mac_tx.h"

#ifdef __cplusplus
extern "C" {
#endif

/***** DEFINES *****/
#define BLE2ROS_UART_PAYLOAD_BUFF_SIZE      100
#define BLE2ROS_UART_FRAME_SIZE             3
#define BLE2ROS_UART_BUFF_SIZE              BLE2ROS_UART_FRAME_SIZE +
↳ BLE2ROS_UART_PAYLOAD_BUFF_SIZE
#define BLE2ROS_UART_MAX_MSG_QUEUE_ELEMENTS 5
#define BLE2ROS_UART_RECEIVE_TIMEOUT        100

// BLE2ROS UART MID defines

// Basic ping test
#define BLE2ROS_UART_GENERAL_MID_CMD_PING    0x01
#define BLE2ROS_UART_GENERAL_MID_RSP_PING    0x02

// Restart command
#define BLE2ROS_UART_GENERAL_MID_CMD_RESTART 0x0A

// General errors
#define BLE2ROS_UART_GENERAL_MID_RSP_INVALID_CRC    0xFB
#define BLE2ROS_UART_GENERAL_MID_RSP_INVALID_MID    0xFD
#define BLE2ROS_UART_GENERAL_MID_RSP_INVALID_LENGTH 0xFF

// Provisioning
#define BLE2ROS_UART_PROV_MID_UNPROV_BEACON    0x10
#define BLE2ROS_UART_PROV_MID_PROV_BEACON      0x11
#define BLE2ROS_UART_PROV_MID_NODE             0x12

// Generic OnOff
#define BLE2ROS_UART_ONOFF_MID                 0x20

```

```

#define BLE2ROS_UART_JOYSTICK_MID                                0x30

/*****
TYPEDEFS
*****/
typedef struct _st_ble2ros_uart_msg {
    uint8_t mid;
    uint8_t length;
    uint8_t payload[BLE2ROS_UART_PAYLOAD_BUFF_SIZE];
    uint8_t crc;
} st_ble2ros_uart_msg;

/*****
EXTERNAL VARIABLES
*****/

/*****
DEFINITIONS
*****/
void uart_init(const char* uartName);

int uart_send(const st_ble2ros_uart_msg msg);

st_ble2ros_uart_msg msg_from_data(const char* data);
void data_from_msg(uint8_t* data, const st_ble2ros_uart_msg msg);
uint8_t calc_crc(st_ble2ros_uart_msg msg);

/*****
END
*****/

#ifdef __cplusplus
}
#endif

#endif //UART_H

```

B.2 Source

```

/**
↳ *****
* @file          uart.c
* @company       HES-SO//Valais-Wallis, CH-1950 Sion
* @author        Sammy Francelet
* @version       0.1
* @date          30. June 2022
* @brief         Contains interfaces to access the Zephyr UART Async API
↳ *****
* @attention
*
↳ *****
*/

/*****
INCLUDES
*****/
#include <logging/log.h>

#include "uart.h"

/*****
DEFINES
*****/
#define LOG_MODULE_NAME uart
LOG_MODULE_REGISTER(LOG_MODULE_NAME, LOG_LEVEL_DBG);

/* Size of stack area used by each thread */
#define TASK_STACKSIZE 1024

/* Scheduling priority used by each thread */
#define TASK_PRIORITY 7

/*****
MACROS
*****/

/*****
TYPEDEFS
*****/

/*****
PROTOTYPES
*****/
static void uart_cb(const struct device *dev, struct uart_event *evt, void
↳ *user_data);

/*****
LOCALS
*****/
static char rx_buf[BLE2ROS_UART_BUFF_SIZE];
static char* tx_buf;
static const struct device* uart;

/*****
DEFINITIONS
*****/
void uart_init(const char* uartName) {

```

```

    int err = 0;

    LOG_INF("Enabling %s and starting async_api", uartName);

    uart = device_get_binding(uartName);
    if (uart == NULL) {
        LOG_ERR("Could not find %s!", uartName);
        return;
    }

    err = uart_callback_set(uart, uart_cb, NULL);
    if (err) {
        LOG_ERR("Can't set uart callback (error %d)", err);
        return;
    }

    err = uart_rx_enable(uart, rx_buf, sizeof(rx_buf),
        ↪ BLE2ROS_UART_RECEIVE_TIMEOUT);
    if (err) {
        LOG_ERR("Can't enable uart rx (error %d)", err);
        return;
    }
}

int uart_send(const st_ble2ros_uart_msg msg) {
    int err = 0;

    uint8_t dataSize = msg.length + BLE2ROS_UART_FRAME_SIZE;

    tx_buf = (char*) k_malloc(dataSize * sizeof(uint8_t));
    data_from_msg(tx_buf, msg);

    for (int i = 0; i < dataSize; i++) {
        LOG_DBG("%d", tx_buf[i]);
    }

    err = uart_tx(uart, tx_buf, dataSize, SYS_FOREVER_US);
    if (err) {
        LOG_ERR("Can't send data to uart (error %d)", err);
        return err;
    }
    return 0;
}

static void uart_cb(const struct device *dev, struct uart_event *evt, void
    ↪ *user_data) {
    int err = 0;

    switch (evt->type) {
    case UART_TX_DONE:
        // The whole TX buffer was transmitted
        /* code */
        LOG_DBG("Transmission done on %s", dev->name);
        k_free(tx_buf);
        break;

    case UART_TX_ABORTED:
        // Transmitting aborted due to timeout or uart_tx_abort call
        /* code */
        LOG_ERR("Transmission aborted on %s", dev->name);
        break;

    case UART_RX_RDY:
        // Some data was received
        // and received timeout occurred (if enabled)

```

Appendix B. UART Physical - Zephyr

```
// or receive buffer is full
/* code */
LOG_DBG("Received something, length: %d", evt->data.rx.len);

struct uart_event_rx data = evt->data.rx;
st_ble2ros_uart_msg rxMsg = msg_from_data(data.buf + data.offset);

LOG_DBG("Received message :\nMID: %d\nlength: %d\npayload: %s\nCRC: %d",
        rxMsg.mid, rxMsg.length, rxMsg.payload, rxMsg.crc);

uart_rx_disable(dev); // Disable to avoid buffer overflow

to_mac_rx(rxMsg);

break;

case UART_RX_BUF_REQUEST:
    // Drivers requests next buffer for continuous reception
    /* code */
    LOG_DBG("%s RX buffer request", dev->name);
    break;

case UART_RX_BUF_RELEASED:
    // Buffer is no longer used by UART driver
    /* code */
    LOG_DBG("%s RX buffer released", dev->name);
    break;

case UART_RX_DISABLED:
    // This event is generated whenever receiver has been stopped,
    // disabled or finished its operation (receive buffer filled)
    // and can be enabled again
    /* code */
    LOG_DBG("%s RX disabled, re-enabling it", dev->name);
    err = uart_rx_enable(dev, rx_buf, sizeof(rx_buf),
        ↪ BLE2ROS_UART_RECEIVE_TIMEOUT);
    if (err) {
        LOG_ERR("Can't re-enable uart rx (error %d)", err);
    }
    break;

case UART_RX_STOPPED:
    // RX has stopped due to external event
    /* code */
    LOG_ERR("%s RX STOPPED!!!", dev->name);
    break;

default:
    break;
}

}

st_ble2ros_uart_msg msg_from_data(const char* data) {
    st_ble2ros_uart_msg msg;

    msg.mid = data[0];
    msg.length = data[1];
    for (int i = 0; i < msg.length; i++) {
        msg.payload[i] = data[i+2];
    }
    msg.payload[msg.length] = '\0'; // NULL char for correct logs

    msg.crc = data[msg.length+2];

    return msg;
}
```

```
}

void data_from_msg(uint8_t* data, const st_ble2ros_uart_msg msg) {
    data[0] = msg.mid;
    data[1] = msg.length;

    for (int i = 0; i < msg.length; i++) {
        data[i+2] = msg.payload[i];
    }
    data[msg.length + 2] = msg.crc;
}

uint8_t calc_crc(st_ble2ros_uart_msg msg) {
    uint8_t crc = 0;

    crc += msg.mid;
    crc += msg.length;

    for (int i = 0; i < msg.length; i++) {
        crc += msg.payload[i];
    }

    return crc;
}

/*****
END
*****/
```


C | UART Physical - ESP-IDF

C.1 Header

```

/**
 * @file          uart.h
 * @company       HES-SO//Valais-Wallis, CH-1950 Sion
 * @author        Samy Francelet
 * @version       0.1
 * @date          1. July 2022
 * @brief         Contains interfaces to access the ESP32 UART
 *
 * @attention
 *
 */

#ifndef UART_H
#define UART_H

/***** INCLUDES *****/
#include "driver/uart.h"

#ifdef __cplusplus
extern "C" {
#endif

/***** DEFINES *****/
#define BLE2ROS_UART_PAYLOAD_BUFF_SIZE      100
#define BLE2ROS_UART_FRAME_SIZE             3
#define BLE2ROS_UART_BUFF_SIZE              BLE2ROS_UART_FRAME_SIZE +
↳ BLE2ROS_UART_PAYLOAD_BUFF_SIZE
#define BLE2ROS_UART_MAX_MSG_QUEUE_ELEMENTS 5
#define BLE2ROS_UART_RECEIVE_TIMEOUT        1000

// BLE2ROS UART MID defines

// Basic ping test
#define BLE2ROS_UART_GENERAL_MID_CMD_PING    0x01
#define BLE2ROS_UART_GENERAL_MID_RSP_PING    0x02

// Restart command
#define BLE2ROS_UART_GENERAL_MID_CMD_RESTART 0x0A
#define BLE2ROS_UART_GENERAL_MID_RSP_RESTART 0x0B

// General errors
#define BLE2ROS_UART_GENERAL_MID_RSP_INVALID_CRC    0xFB
#define BLE2ROS_UART_GENERAL_MID_RSP_INVALID_MID    0xFD
#define BLE2ROS_UART_GENERAL_MID_RSP_INVALID_LENGTH 0xFF

// Provisioning
#define BLE2ROS_UART_PROV_MID_UNPROV_BEACON    0x10
#define BLE2ROS_UART_PROV_MID_PROV_BEACON      0x11
#define BLE2ROS_UART_PROV_MID_NODE             0x12

// Generic OnOff
#define BLE2ROS_UART_ONOFF_MID                 0x20

#define BLE2ROS_UART_JOYSTICK_MID              0x30

```

```

/*****
TYPEDEFS
*****/
typedef struct _st_ble2ros_uart_msg {
    uint8_t mid;
    uint8_t length;
    uint8_t payload[BLE2ROS_UART_PAYLOAD_BUFF_SIZE];
    uint8_t crc;
} st_ble2ros_uart_msg;

/*****
EXTERNAL VARIABLES
*****/

/*****
DEFINITIONS
*****/
void uart_init(uart_port_t uartNum);

void uart_send(const st_ble2ros_uart_msg msg);

st_ble2ros_uart_msg msg_from_data(const char* data);
void data_from_msg(char* data, const st_ble2ros_uart_msg msg);
uint8_t calc_crc(st_ble2ros_uart_msg msg);

/*****
END
*****/

#ifdef __cplusplus
}
#endif

#endif //UART_H

```

C.2 Source

```

#include <string.h>

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"

#include "esp_log.h"
static const char *TAG = "uart_phy";

#include "uart.h"
#include "uart_mac_rx.h"

#define BUFF_SIZE 1024

QueueHandle_t uartQueue;
static uart_port_t uartPort;

static void uart_event_task(void* arg) {
    uart_event_t event;
    st_ble2ros_uart_msg rxMsg;
    uint8_t* dataBuff = (uint8_t*) malloc(BLE2ROS_UART_BUFF_SIZE);

    ESP_LOGI(TAG, "uart[%d] event task -> starting infinite loop...", uartPort);
    while (1) {
        // Wait for UART event
        if (xQueueReceive(uartQueue, (void*)&event, (TickType_t)portMAX_DELAY)) {
            bzero(dataBuff, BLE2ROS_UART_BUFF_SIZE);
            ESP_LOGD(TAG, "uart[%d] event:", uartPort);

            switch (event.type) {
                case UART_DATA:
                    //Event of UART receiving data
                    /*We'd better handler data event fast, there would be much
                     ↳ more data events than
                     other types of events. If we take too much time on data event,
                     ↳ the queue might
                     be full.*/
                    ESP_LOGD(TAG, "Received something, length: %d", event.size);

                    uart_read_bytes(uartPort, dataBuff, event.size, portMAX_DELAY);
                    rxMsg = msg_from_data((const char*) dataBuff);

                    ESP_LOGD(TAG, "Received message :\nMID: %d\nlength:
                     ↳ %d\npayload: %s\nCRC: %d",
                        rxMsg.mid, rxMsg.length, rxMsg.payload, rxMsg.crc);

                    to_mac_rx(rxMsg);
                    break;
                case UART_FIFO_OVF:
                    //Event of HW FIFO overflow detected
                    ESP_LOGW(TAG, "hw fifo overflow");
                    // If fifo overflow happened, you should consider adding flow
                    ↳ control for your application.
                    // The ISR has already reset the rx FIFO,
                    // As an example, we directly flush the rx buffer here in
                    ↳ order to read more data.
                    uart_flush_input(uartPort);
                    xQueueReset(uartQueue);
                    break;
                case UART_BUFFER_FULL:
                    //Event of UART ring buffer full
                    ESP_LOGI(TAG, "ring buffer full");

```

```

        // If buffer full happened, you should consider encreasing
        ↪ your buffer size
        // As an example, we directly flush the rx buffer here in
        ↪ order to read more data.
        uart_flush_input(uartPort);
        xQueueReset(uartQueue);
        break;
    case UART_BREAK:
        //Event of UART RX break detected
        ESP_LOGW(TAG, "uart rx break");
        break;
    case UART_PARITY_ERR:
        //Event of UART parity check error
        ESP_LOGW(TAG, "uart parity error");
        break;
    case UART_FRAME_ERR:
        //Event of UART frame error
        ESP_LOGW(TAG, "uart frame error");
        break;
    case UART_PATTERN_DET:
        ESP_LOGI(TAG, "uart pattern detected, somehow...");
        break;
    default:
        ESP_LOGI(TAG, "uart event type: %d", event.type);
        break;
    }
}
}
free(dataBuff);
dataBuff = NULL;
vTaskDelete(NULL);
}

void uart_init(uart_port_t uartNum) {
    uartPort = uartNum;

    // Configuring UART port
    uart_config_t uartConfig = {
        .baud_rate = 115200,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        // .rx_flow_ctrl_thresh = 122,
        // .source_clk = UART_SCLK_DEFAULT,
    };
    uart_param_config(uartPort, &uartConfig);

    // Set UART pins
    uart_set_pin(uartPort, 19, 18, 16, 17);

    // Install UART driver, and sets the queue
    uart_driver_install(
        uartPort, // Port number
        BUFF_SIZE, // RX Buff size
        BUFF_SIZE, // TX Buff size
        20, // Queue size
        &uartQueue, // Pointer to queue
        0 // Flags to allocate interrupts
    );

    // Create task handler for UART event from ISR
    xTaskCreate(uart_event_task, "uart_event_task", 4096, NULL,
        ↪ CONFIG_UART_TASKS_PRIO, NULL);
}

```

Appendix C. UART Physical - ESP-IDF

```
void uart_send(const st_ble2ros_uart_msg msg) {
    uint8_t dataSize = msg.length + BLE2ROS_UART_FRAME_SIZE;

    char data[BLE2ROS_UART_BUFF_SIZE];
    data_from_msg(data, msg);

    /*ESP_LOGI(TAG, "Sending data...");
    for (int i = 0; i < dataSize; i++) {
        ESP_LOGI(TAG, "%d", data[i]);
    }*/

    uart_write_bytes(uartPort, data, dataSize);
}

st_ble2ros_uart_msg msg_from_data(const char* data) {
    st_ble2ros_uart_msg msg;

    msg.mid = data[0];
    msg.length = data[1];
    for (int i = 0; i < msg.length; i++) {
        msg.payload[i] = data[i+2];
    }
    msg.payload[msg.length] = '\0'; // NULL char for correct logs

    msg.crc = data[msg.length+2];

    return msg;
}

void data_from_msg(char* data, const st_ble2ros_uart_msg msg) {
    data[0] = msg.mid;
    data[1] = msg.length;

    for (int i = 0; i < msg.length; i++) {
        data[i+2] = msg.payload[i];
    }
    data[msg.length + 2] = msg.crc;
}

uint8_t calc_crc(st_ble2ros_uart_msg msg) {
    uint8_t crc = 0;

    crc += msg.mid;
    crc += msg.length;

    for (int i = 0; i < msg.length; i++) {
        crc += msg.payload[i];
    }

    return crc;
}
```

D | UART Message emulator

Appendix D. UART Message emulator

```
import serial
import time

BLE2ROS_UART_GENERAL_MID_CMD_PING = "0x01"
BLE2ROS_UART_GENERAL_MID_RSP_PING = "0x02"

BLE2ROS_UART_GENERAL_MID_CMD_RESTART = "0x0A"
BLE2ROS_UART_GENERAL_MID_RSP_RESTART = "0x0B"

BLE2ROS_UART_GENERAL_MID_RSP_INVALID_CRC = "0xFB"
BLE2ROS_UART_GENERAL_MID_RSP_INVALID_MID = "0xFD"
BLE2ROS_UART_GENERAL_MID_RSP_INVALID_LENGTH = "0xFF"

BLE2ROS_UART_RECEIVE_TIMEOUT = 1000/1000000 # in us

UART_PORT = "COM3"
BAUDRATE = 115200

class ble2ros_uart_msg:
    mid = BLE2ROS_UART_GENERAL_MID_CMD_PING
    length = 0
    payload = []
    crc = 0

    def __init__(self, mid = BLE2ROS_UART_GENERAL_MID_CMD_PING, payload = []):
        self.mid = mid
        self.set_payload(payload)

    def __str__(self):
        txt = "-----\n"
        txt += "BLE2ROS Frame\n"
        txt += "MID: " + self.mid + " "
        if self.mid == BLE2ROS_UART_GENERAL_MID_CMD_PING:
            txt += "CMD_PING"
        elif self.mid == BLE2ROS_UART_GENERAL_MID_RSP_PING:
            txt += "RSP_PING"
        elif self.mid == BLE2ROS_UART_GENERAL_MID_CMD_RESTART:
            txt += "CMD_RESTART"
        elif self.mid == BLE2ROS_UART_GENERAL_MID_RSP_RESTART:
            txt += "RSP_RESTART"
        elif self.mid == BLE2ROS_UART_GENERAL_MID_RSP_INVALID_CRC:
            txt += "INVALID_CRC"
        elif self.mid == BLE2ROS_UART_GENERAL_MID_RSP_INVALID_LENGTH:
            txt += "INVALID_LENGTH"
        elif self.mid == BLE2ROS_UART_GENERAL_MID_RSP_INVALID_MID:
            txt += "INVALID_MID"

        txt += "\n"
        txt += "Length: " + str(self.length) + "\n"
        if (isinstance(self.payload, str)):
            txt += "Payload: " + self.payload + "\n"
        else:
            txt += "Payload: " + "".join(chr(c) for c in self.payload) + "\n"

        txt += "CRC: " + str(self.crc) + "\n"
        txt += "-----"
        return txt

    @staticmethod
    def from_bytes(data):
        print("Msg from bytes :", data)
        msg = ble2ros_uart_msg()
        msg.mid = "0x%02X" % data[0]
        msg.length = data[1]
```

```

        tmpPayload = []
        for i in range(msg.length):
            tmpPayload.append(data[i+2])

        msg.payload = tmpPayload
        msg.crc = data[msg.length + 2]
        return msg

    def calc_crc(self):
        self.crc = int(self.mid, 16)
        self.crc += self.length

        for i in range(self.length):
            self.crc += int(ord(self.payload[i]))

        self.crc = self.crc%256

    def set_payload(self, payload):
        self.payload = payload
        self.length = len(self.payload)
        self.calc_crc()

    def to_bytes(self):
        frame = []
        frame.append(int(self.mid, 16))
        frame.append(self.length)
        for i in range(self.length):
            frame.append(int(ord(self.payload[i])))

        frame.append(self.crc)
        return bytearray(frame)

def sendMsg(msg, name = UART_PORT, baudrate = BAUDRATE, bad_crc = False):
    ser = serial.Serial(name, baudrate)
    print("Writing to port", ser.name)
    if bad_crc:
        print("Sending bad CRC")
        msg.crc = msg.crc+1
    ser.write(msg.to_bytes())
    ser.flush()
    ser.close()

def readMsg(name = UART_PORT, baudrate = BAUDRATE, timeout=0.1):
    ser = serial.Serial(name, baudrate, timeout=timeout)
    print("Reading from port", ser.name)
    buf = ser.read(100)
    ser.close()

    msg = ble2ros_uart_msg.from_bytes(buf)
    return msg

def flushInput(name = UART_PORT, baudrate = BAUDRATE):
    ser = serial.Serial(name, baudrate)
    ser.flushInput()
    ser.close()

ping_txt = "Hello friend !"
msg = ble2ros_uart_msg(BLE2ROS_UART_GENERAL_CMD_PING, ping_txt)
print(msg)

flushInput()
sendMsg(msg, bad_crc=False)
rxMsg = readMsg()
print(rxMsg)

```


E | BLE2ROS Test Data

Relay buffers	30	Simple buf tailroom max	20
RX_SEG_MAX	32		
TX_SEG_MAX	32		

Data Rate vs Data size - unack				
Data Size [bytes]	Delay with relay [ms]	Delay without relay [ms]	Max Data Rate with relay[bytes/s]	Max Data Rate without relay[bytes/s]
1	160	110	6,25	9,09
2	160	110	12,50	18,18
3	160	110	18,75	27,27
4	160	110	25,00	36,36
5	160	110	31,25	45,45
6	160	110	37,50	54,55
7	160	110	43,75	63,64
8	160	110	50,00	72,73

Data Rate vs Data size - ack				
Data Size [bytes]	Delay with relay [ms]	Delay without relay [ms]	Max Data Rate with relay[bytes/s]	Max Data Rate without relay[bytes/s]
1	280	130	3,57	7,69
2	280	130	7,14	15,38
3	280	130	10,71	23,08
4	280	130	14,29	30,77
5	280	130	17,86	38,46
6	280	130	21,43	46,15
7	280	130	25,00	53,85
8	280	130	28,57	61,54

Bibliography

- [1] Open Robotics. *ROS Website*. URL: <https://ros.org/>.
- [2] Open Robotics. *ROS Documentation*. URL: <https://docs.ros.org/en/humble>.
- [3] microROS Team. *microROS Documentation*. URL: <https://micro.ros.org/docs/overview/features/>.
- [4] microROS Team. *microROS Documentation, custom transports*. URL: https://micro.ros.org/docs/tutorials/advanced/create_custom_transports/.
- [5] Bluetooth SIG. *Mesh Networking*. 2017. URL: <https://www.bluetooth.com/learn-about-bluetooth/recent-enhancements/mesh/>.
- [6] SIG. *Bluetooth Mesh Glossary of Terms*. 2022. URL: <https://www.bluetooth.com/learn-about-bluetooth/recent-enhancements/mesh/mesh-glossary/>.
- [7] Silicon Labs. *AN1137: Bluetooth® Mesh Network Performance*. Tech. rep. Silicon Labs, 2022. URL: <https://www.silabs.com/documents/public/application-notes/an1137-bluetooth-mesh-network-performance.pdf>.
- [8] Silicon Labs. *Benchmarking Bluetooth Mesh, Thread, and Zigbee Network Performance*. 2022. URL: <https://www.silabs.com/wireless/multiprotocol/mesh-performance>.
- [9] Zephyr Project. *Zephyr UART peripheral*. 2022. URL: <https://docs.zephyrproject.org/latest/hardware/peripherals/uart.html>.
- [10] Zephyr Project. *Zephyr DMA peripheral*. 2022. URL: <https://docs.zephyrproject.org/latest/hardware/peripherals/dma.html#dma-api>.
- [11] Espressif Systems. *ESP32 UART peripheral*. 2022. URL: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/uart.html>.
- [12] microROS Team. *micro-ROS setup utility*. GitHub. 2022. URL: https://github.com/micro-ROS/micro_ros_setup.
- [13] Eduardo De Leon and Majid Nabi. *An Experimental Performance Evaluation of Bluetooth Mesh Technology for Monitoring Applications*. Tech. rep. Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, the Netherlands, 2020. URL: <https://www.es.ele.tue.nl/~mnabi/content/publications/WCNC2020.pdf>.
- [14] Sammy Francelet. *ESP32 crashes when listening `message_r` reliably*. GitHub issue. 2022. URL: https://github.com/micro-ROS/micro_ros_espidf_component/issues/155.
- [15] microROS Team. *micro-ROS ESP-IDF Component*. GitHub. 2022. URL: https://github.com/micro-ROS/micro_ros_espidf_component.

Bibliography

- [16] eProxima. *eProxima Micro XRCE-DDS*. 2018. URL: <https://micro-xrce-dds.docs.eprosima.com/en/latest/#main-features>.
- [17] Merlan Maria. *embeddedRTPS the new experimental middleware for micro-ROS*. 2022. URL: <https://micro.ros.org/blog/2021/10/10/embeddedRTPS/>.

Acronyms

BLE Bluetooth Low Energy. [11–13](#), [43](#)

BT Mesh Bluetooth Mesh Networking. [2](#), [3](#), [5](#), [11–15](#), [19–25](#), [28](#), [30–33](#), [38–41](#), [45](#)

CDB Configuration DataBase. [32](#), [45](#)

IoT internet of things. [1](#), [5](#), [15](#)

M2M machine-to-machine communication. [1](#), [5](#), [15](#)

microROS ROS for microcontrollers. [2](#), [3](#), [10](#), [15](#), [18–21](#), [23](#), [26](#), [28](#), [33](#), [34](#)

OOB Out Of Band. [14](#), [32](#)

ROS Robot Operating System. [2](#), [3](#), [5–10](#), [15](#), [20](#), [23–25](#), [33](#), [35](#), [38](#), [39](#), [41](#), [44](#)

RTOS Real Time Operating System. [10](#), [27](#)

Glossary

Provisioner A Provisioner is a device that can add another device to the network. As such, it is responsible for generating and distributing NetKeys. It is expected that Provisioners will typically be smartphone or tablet applications. Other implementations are also possible. Source: <https://www.bluetooth.com/learn-about-bluetooth/recent-enhancements/mesh/mesh-glossary/> . 2, 3, 14, 15, 18–21, 23, 24, 26, 28, 32, 33, 38–41, 43, 45

RCL Executor The RCL Executor (ROS Client Library Executor) is the system responsible for dispatching messages received by the Middleware to the corresponding subscribers of the ROS node.. 20