Ecole Polytechnique
Fédérale de Lausanne

Onboard Computer Programming

Semester Project

Author
Andrea Grillo

Professor/Lab
KLUTER Theo / LAP

Supervisor
AMACHER Robin, CROCE David

Swiss Solar Boat

Fall 2023

Contents

1 Abstract **2**
1.1 Solutions and importants outcomes . . . 2
1.2 Workflow . . . 2
1.3 Key schematic . . . 3

2 Introduction **4**
2.1 Global Context . . . 4
2.2 Project's Context . . . 5

3 System Description **6**
3.1 Hardware Architecture . . . 6
3.2 Software Technical Requirements . . . 7

4 Framework Architecture Proposal **7**
4.1 ROS . . . 7
4.2 Choice of ROS version . . . 8

5 ROS system design **9**
5.1 ROS components . . . 9
5.2 ROS interfaces . . . 9
5.3 ROS graph . . . 10
5.4 Development of CAN_to_ROS node . . . 11
5.4.1 First version - Proof of Concept . . . 11
5.4.2 First version limitations . . . 12
5.4.3 DBC CAN database files . . . 13

6 Development Environment **14**
6.1 Docker Image . . . 14
6.2 GIT repositories . . . 15
6.3 Node templates . . . 15

7 Issue with selected Onboard Computer **16**
7.1 MCP2515 Configuration . . . 16
7.2 OBC considerations . . . 17

8 Tests **18**

9 Next steps **19**
9.1 CAN_to_ROS . . . 19
9.2 Useful ROS tools . . . 19
9.3 Onboard Computer choice and Electronics . . . 19
9.4 Network Architecture . . . 20

10 Appendix **21**
10.1 Abbreviations . . . 21
10.2 Python Node Template . . . 22
10.3 C++ Node Template . . . 23
10.4 NodeJS Node Template . . . 24

11 References **25**

1 Abstract

1.1 Solutions and importants outcomes

List of solutions and importants outcomes		
Title	Description/assumption	Value/Solution
Framework	software to be run on the onboard computer	ROS
ROS Distro	choice of distribution and version	ROS2 Humble Hawksbill
Development Environment	definition of tools for development	Docker Image

Table 1: Important outcomes of the project

1.2 Workflow

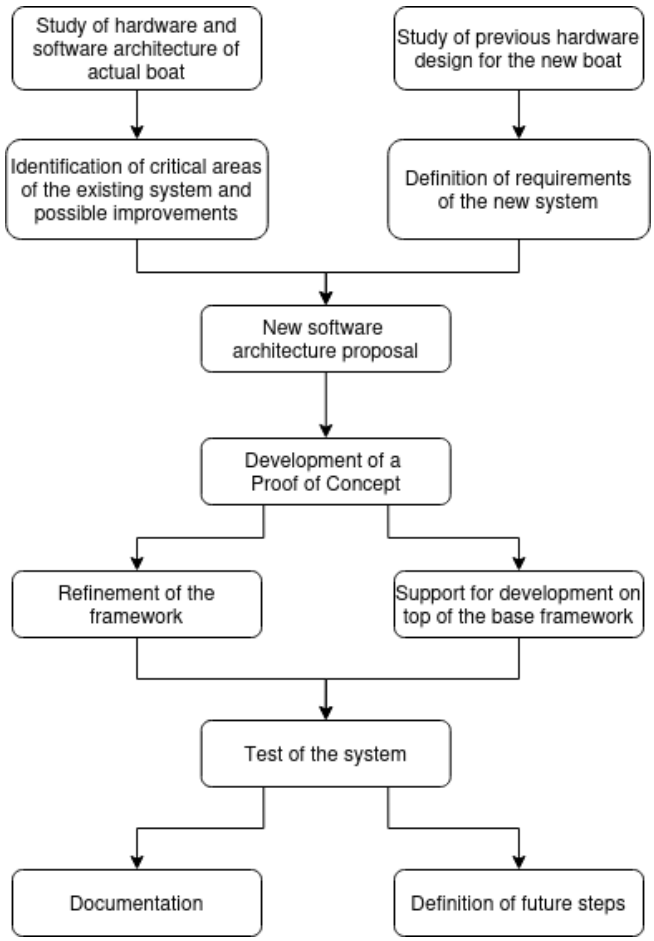

Figure 1: Flowchart of the work done during the semester

1.3 Key schematic

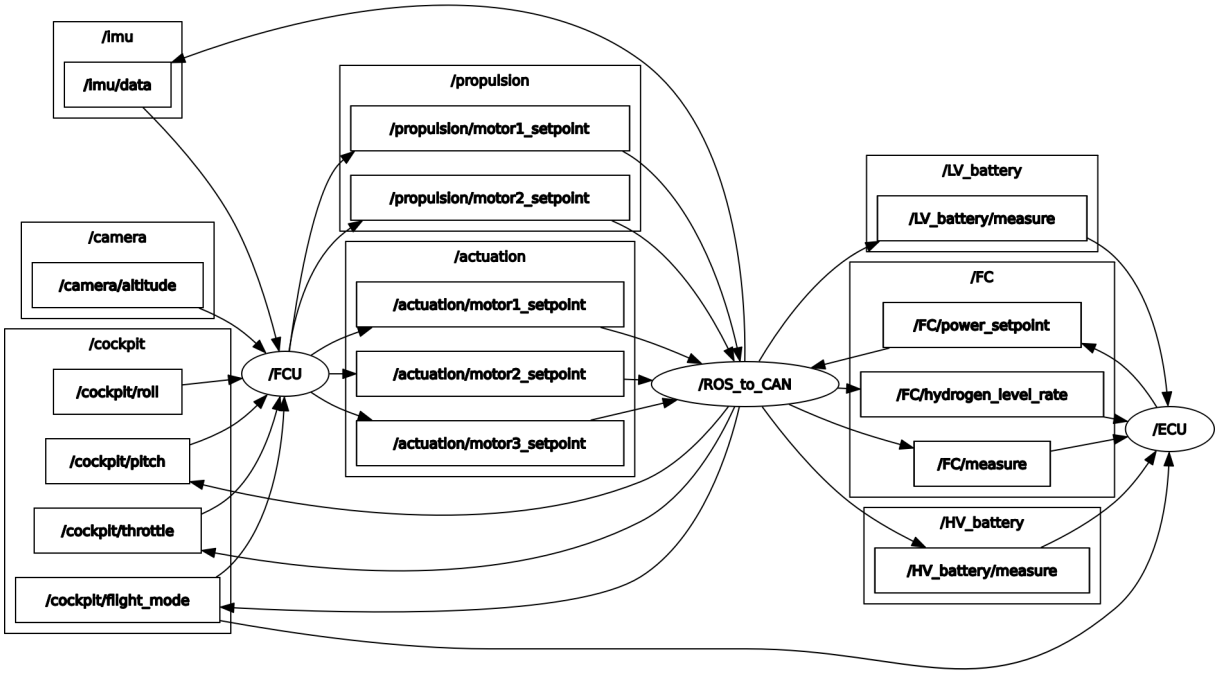

Figure 2: ROS graph

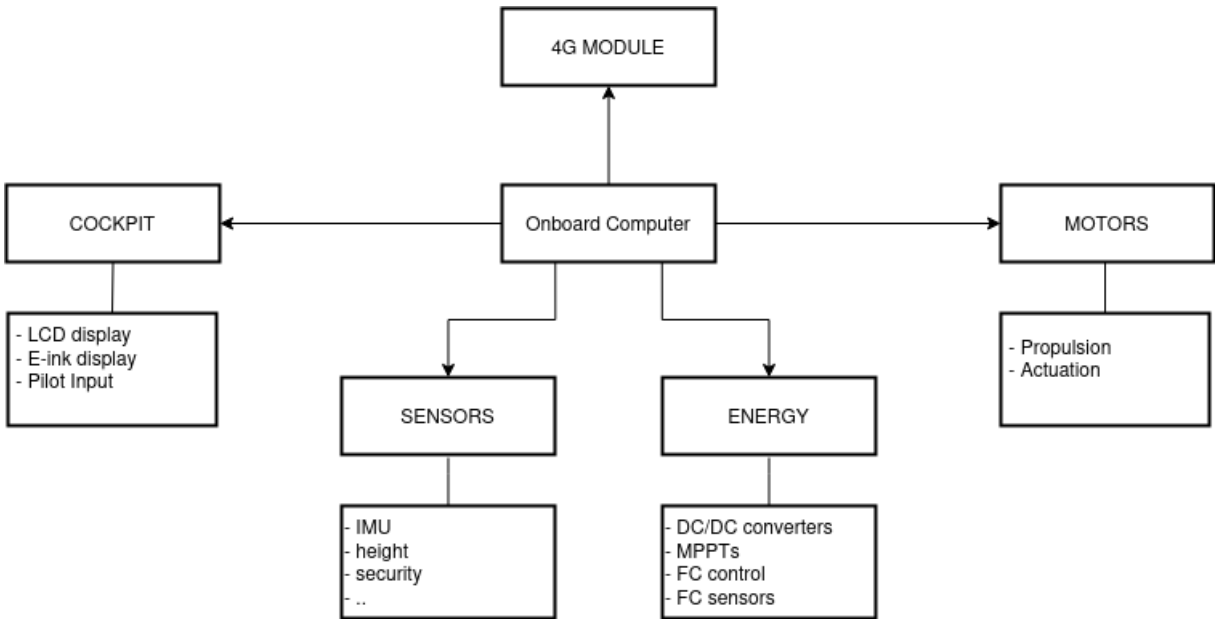

Figure 3: Structure of the system

2 Introduction

2.1 Global Context

After two participations at the MEBC (Monaco Energy Boat Challenge) in 2021 and 2022, the Swiss Solar Boat team is now turning to a new challenge aiming at placing itself halfway between pure optimization for the purpose of a competition and emerging industrial actors in the field of sustainable boating. To do so, the team has decided to densify the energy on board and reach higher speeds than the previous boat. This is achieved by using pressurized hydrogen gas as an energy carrier.
In this state of mind Swiss Solar Boat is developing a new boat by horizon 2026 with the following main characteristics:

- 3 passengers
- 150 km of autonomy
- 25 kts cruising speed
- 30 kts of top speed
- Electric propulsion powered by a fuel cell and PVs

Simuntaneously, the team is using the current boat, the Dahu, as a test platform for the new boat. As such, the team is dedicated to hybridizing the Dahu with pressurized hydrogen with the aim of extending its range significantly.
In this context, this report aims to outline and present the work and results achieved in a semester project focused on advancing the new boat's design in the area of the electronics and software architecture.
The project took place during the 3rd semester of the boat's development, representing progress toward finalizing the design phase and preparing for production. It followed a 1st semester of pre-design and a 2nd semester dedicated to the design phase of the new boat.

Figure 4: The Dahu, the actual boat.

2.2 Project's Context

The project is aimed at designing and implementing a framework for the electronics and software architecture for the new boat, allowing the development of the control system, the onboard safety, the communication subsystem with the ground station, and the collection and logging of data.
The project lies in the middle of the hardware and software departments, as the idea is to build a sound structure for the development of the various subsystems, providing an hardware abstraction layer for the communication between the components that will be built on top of it.
During previous semesters, the electronic architecture of the boat has beed defined, which includes the choice of the Onboard Computer (OBC), the choice of the communication bus, the high level connections between the main components, the development of electronic PCB (Printed Circuit Boards) such as a shield for the Onboard Computer [1] [2] [3]. In this project, one of the objectives has been to evaluate and validate the choices made for the electronics architecture, to be sure that software and hardware components can be integrated in the best possible way.
The most important point about this project is that it defines the framework for the work that will be done during the next semesters, for this reason special care has to be taken in the design phase. It is important that the whole system is flexible and future-proof. During the future development many requirements may need to be changed, the technology used can be improved, so the goal is to have a flexible base structure that can adapt to these changes without being a limitation. The system has to be modular, easily configurable and modifiable to account for fast development of the subsystems.

Inputs that were needed		Outputs provided through this project	
From	Briefly what	To	Briefly what
Elec-hardware	High-level hardware architecture	Elec-software / Energy	Framework and support for the development of the Flight Control Unit, Energy Control Unit, Safety Control Unit
Elec-software	Requirements of the control system	Elec-software	Communication architecture for remote control and monitoring through the Ground Station

Table 2: Summary of the Inputs and the Outputs of this project

3 System Description

The boat is a complex system, made by various modules, that have to communicate and cooperate together to work properly and make the boat functional.
The electronics architecture should provide an effective and reliable structure for these modules to be controlled in an efficient way.
The architecture has been defined during projects in previous semesters, the main components are listed in Table 3.

Main components of the system	
Component	Purpose
Onboard Computer	Runs ECU, FCU, SCU, Ground Station communication, datalogging
Cockpit	Handles pilot input and supplies boat data to display on cockpit screen
Motor Control Box Actuation	Interface with actuation motor controllers
Motor Control Box Propulsion	Interface with propulsion motor controllers
IMU	Inertial Measurement Unit
Fuel Cell Controller Box	Interface with FC controller and safety sensors
Front of the boat	Interface with sensors in the front
Low Voltage Battery	Interface with low voltage battery BMS
High Voltage Battery	Interface with low voltage battery BMS
Energy Management System	Interface with energy management components (DC-DC converters, ..)

Table 3: Main components of the system

All those systems will communicate together using a CAN (Controller Area Network) bus. In the work of Kai Wen Loo [1] in previous semester, the configuration of the bus has been defined, a speed of 500 kbit/s has been chosen, as well as the use of isolated CAN transceivers and the load of the bus has been estimated to be 33,2% with said configuration.
The purpose of my project is to define the software framework for the Onboard Computer to be able to communicate with all the subsystems and run the needed programs.
The tasks to be run on the Onboard Computer are listed in Table 4.

Tasks of the Onboard Computer	
Task	Function
FCU	Flight Control Unit
ECU	Energy Control Unit
SCU	Safety Control Unit
GS Communication	Send telemetry data and receive commands
Datalogging	Save data in local storage and remotely
LCD Monitor	Interface with pilot

Table 4: Tasks of the Onboard Computer

3.1 Hardware Architecture

The Onboard Computer needs the following hardware components connected to it [1]:

- Power Supply
- 4G Module
- CAN-SPI translation chips
- LCD display

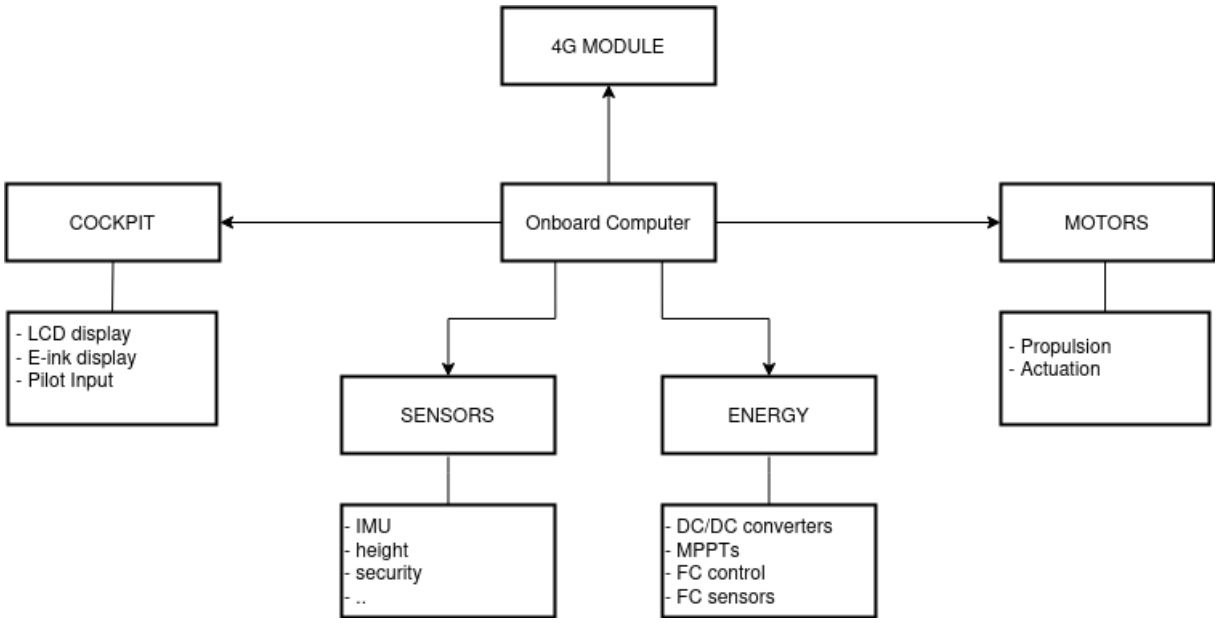

Figure 5: Hardware architecture [1] [2]

Two CAN lines are connected to the OBC, one which is used for all the sensors and actuators of the boat, and the other is used just to communicate with the IMU.

3.2 Software Technical Requirements

The requirements for the OBC framework are resumed in the following table:

Technical Requirements
Provide a hardware abstraction layer for the programs
Ensure reliable CAN communication with other components
Provide remote communication with Ground Station
Run multiple processes in parallel
Provide a way to communicate between the different processes

Table 5: Technical Requirements of the System

4 Framework Architecture Proposal

After having considered the defined hardware architecture and identified the requirements of the system to be implemented, an architecture for the software part has been proposed.

The main idea proposed is about the use of ROS (Robot Operating System).

4.1 ROS

ROS provides functionality for hardware abstraction, communication between processes, possibility of having a distributed system over multiple machines, tools for testing, visualization, datalogging and much more.

ROS provides a way to connect a network of processes (nodes). Nodes can be developed in different programming languages, can be run on multiple devices, and they can communicate easily with each

other using ROS interfaces.

ROS is made and maintained by a big developer and user community. This results in a great amount of reusable packages that are simple to integrate, thanks to the architecture of the system. This allows for a design of a complex system by connecting existing solutions for small problems.

A brief overview of what ROS provides:

- Design a modular system, in which it is easy to replace components with a defined interface, removing the need of adapting the system to changes.
- Clear definition of the system graph, which helps to define the flow of the data in the whole system.
- Integration of submodules developed in different programming languages / frameworks.
- ROS is a distributed system. It is possible to create nodes over a network of devices, without worrying about where code is run. It implements Interprocess Communication (IPC) and Remote Procedure Call (RPC) systems.
- A vast library of existing open-source tools and modules already developed for similar architecture that can be used as inspiration or be part of the system.

4.2 Choice of ROS version

Considerable attention was paid to the choice of ROS version to be used.
ROS2 has been chosen over ROS1, as the latter is now outdated and used just for legacy reasons.
The goal is to have a stable system with support that lasts as long as possible, so the *Humble Hawksbill* distribution has been chosen instead of the last *Iron Irwini*, because it is a LTS (Long Term Support) version and its support will end in 2027, which seems a reasonable timeframe for the development of the system.

Distro	Release date	Logo	EOL date
Iron Irwini	May 23rd, 2023		November 2024
Humble Hawksbill	May 23rd, 2022		May 2027
Galactic Geochelone	May 23rd, 2021		December 9th, 2022
Foxy Fitzroy	June 5th, 2020		June 20th, 2023

Figure 6: Overview of *ROS*2 distributions

5 ROS system design

ROS is a complex system, it offers a wide range of methods to implement the system, so a preliminary phase of the development has been about evaluating and selecting ROS features. [4] [5]

5.1 ROS components

- **Packages** are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), libraries, datasets, configuration files. Packages are the most atomic build item and release item in ROS.
- **Nodes** are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a control system usually comprises many nodes.
- **Topics**, a transport system with publish / subscribe semantics for data exchange between nodes. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.
- **Bags** are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

5.2 ROS interfaces

The ROS Interfaces are the way data can be exchanged between the components of the system. There are three primary styles of interfaces [6].
Below a brief description of those:

- **Topics**
  - Should be used for continuous data streams (sensor data, ...).
  - Are for continuous data flow. Data might be published and subscribed at any time independent of any senders/receivers. Many to many connection. Callbacks receive data once it is available. The publisher decides when data is sent.
  - The definition of a message is done in a *.msg* file.
- **Services**
  - Should be used for remote procedure calls that terminate quickly. They should never be used for longer running processes, in particular processes that might be required to preempt if exceptional situations occur and they should never change or depend on state to avoid unwanted side effects for other nodes.
  - Simple blocking call. Mostly used for comparably fast tasks as requesting specific data. Semantically for processing requests.
  - The definition is done in a *.srv* file, which is composed by: Request, Result.
- **Actions**
  - Should be used for any discrete behavior that runs for a longer time but provides feedback during execution.
  - The most important property of actions is that they can be preempted.
  - More complex non-blocking background processing, used for longer tasks. Semantically for real-world actions.
  - The definition is done in a *.action* file, which is composed by: Goal, Result, Feedback.

5.3 ROS graph

A first version of the ROS graph has been designed.
It is a draft of the whole system, that will be probably modified during the future developments of the subsystem.
The idea is that most of the data is coming from the CAN bus. It is then translated to ROS topics by the *CAN_to_ROS* node, which does the parsing of CAN messages. The data is then elaborated by the other ROS nodes, such as FCU and the ECU, and then it is read by the LCD node, the GS node, and also translated back to CAN messages by the same *CAN_to_ROS* node.
A simple system has been designed, without the use of actions and services. The requirements of the submodules are not still clear, so the graph has been designed with simplicity in mind. Once the submodules will be conceived, a more complex structure can be designed to adapt to complex requirements.

Below is a list of the nodes that have been included in this first draft:

Nodes of the ROS graph	
Task	Function
CAN_to_ROS	Translation from CAN messages to ROS topics
FCU	Flight Control Unit
ECU	Energy Control Unit
SCU	Safety Control Unit
GS	Host Ground Station web server - runs on server
Datalogging	Save data in local storage and remotely
LCD GUI	Interface with pilot

Table 6: Nodes of the ROS graph

A template for each of this nodes has been written, as well as sample topics for communication between the nodes. This allows the use of the *rqt_graph* tool to visualize the resulting ROS graph:

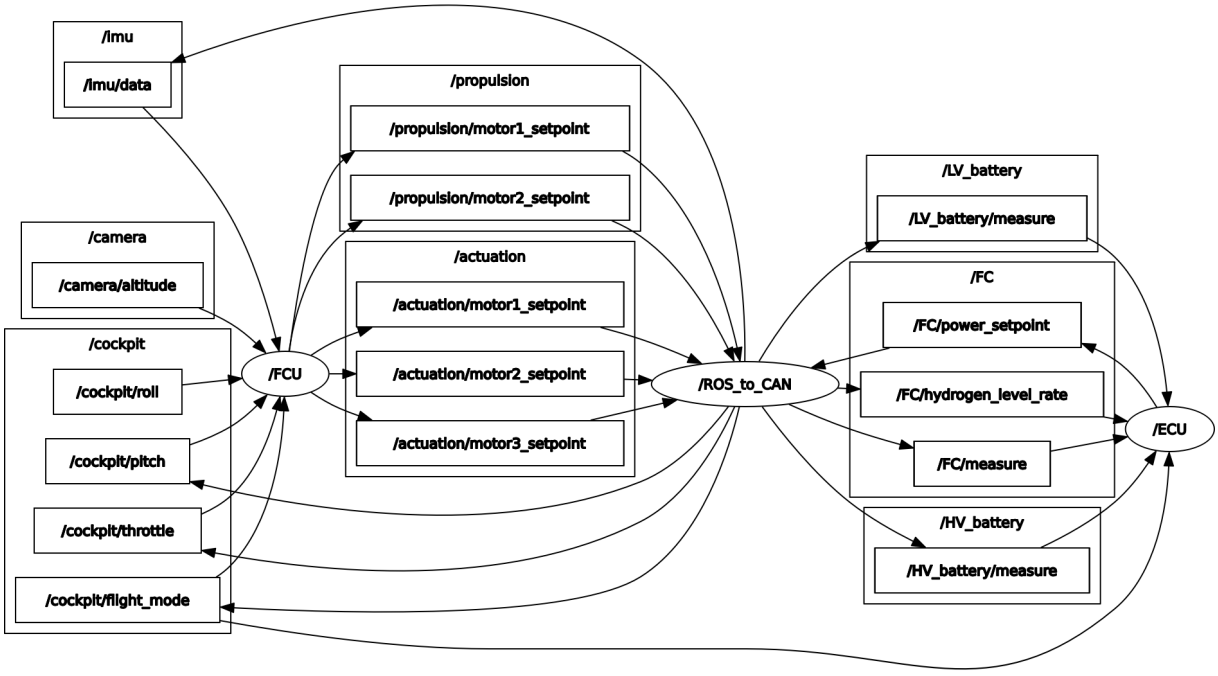

Figure 7: ROS graph

5.4 Development of CAN_to_ROS node

At the heart of the designed system, there is the *CAN_to_ROS* node. Its role is to provide the hardware abstraction layer (HAL) from the hardware CANBUS serial to the ROS interfaces and topics.
A CAN message has the following structure:

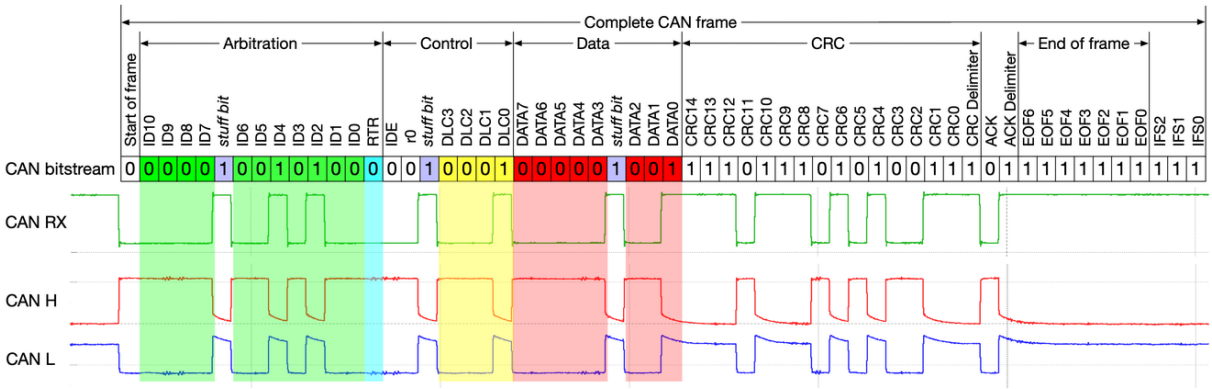

Figure 8: CAN frame structure and signals[10]

The two fields to consider are the AID (Arbitration ID) and the Data. In a normal frame, the AID has 11 bits, and the DATA can be 0 to 8 bytes long. The length of the data is encoded in the DLC (Data Length Code).
In a complex system, there are many different messages sent by different subsystems, identified by AID. As an example, an excerpt of the actual boat message configuration [11] is shown below:

Message count	Message name	Frequency [Hz]	Direction	Length [Bytes]	CAN ID (hex)	CAN ID (dec)	Field Name	Field Count	Send Group	Type	Scaling	Unit	Offset
							sec	15	0	uint8_t	-	s	5
							micro_sec	16	0	uint16_t	e-2	s	6
6	IMU_Delta_Velocity	200	out	6	0x2A2	674	d_vel_x	17	0	int16_t	e-2	m/s^2	0
							d_vel_y	18	0	int16_t	e-2	m/s^2	2
							d_vel_z	19	0	int16_t	e-2	m/s^2	4
7	EKF_Euler	100	out	6	0x1A3	419	roll	20	2	int16_t	e-4	rad	0
							pitch	21	2	int16_t	e-4	rad	2
							yaw	22	2	int16_t	e-4	rad	4
8	EKF_Euler_Accuracy	10	out	6	0x2A3	675	roll_acc	23	2	uint16_t	e-4	rad	0
							pitch_acc	24	2	uint16_t	e-4	rad	2
							yaw_acc	25	2	uint16_t	e-4	rad	4
9	EKF_Position	10	out	8	0x3A1	929	latitude	26	3	int32_t	e-7	deg	0
							longitude	27	3	int32_t	e-7	deg	4
10	EKF_Altitude	10	out	6	0x3A2	930	altitude	28	3	int32_t	e-3	m	0
							undulation	29	0	int16_t	5*e-3	m	4
11	EKF_Pos_Accuracy	10	out	6	0x3A0	928	latitude_acc	30	3	uint16_t	e-2	m	0
							longitude_acc	31	3	uint16_t	e-2	m	2
							altitude_acc	32	3	uint16_t	e-2	m	4
12	EKF_Vel_NED	100	out	6	0x1A4	420	velocity_n	33	2	int16_t	e-2	m/s	0
							velocity_e	34	2	int16_t	e-2	m/s	2
							velocity_d	35	2	int16_t	e-2	m/s	4
13	EKF_Vel_NED_Acc	10	out	6	0x3A3	931	velocity_n_acc	36	2	uint16_t	e-2	m/s	0
							velocity_e_acc	37	2	uint16_t	e-2	m/s	2
							velocity_d_acc	38	2	uint16_t	e-2	m/s	4

Figure 9: Actual boat CAN message configuration [11]

In this picture it can be seen that any AID correspond to a different type of message, with a different message structure and encoding. In the 8 bytes of a CAN message there can be different fields with different meanings for each message type.
Those messages have to be translated to ROS messages to be sent over ROS topics. This is not an easy task, as the encoding is not unique, so it has to be defined for each type of message. On the other hand, the system has to be modular, to allow easy addition of new components without having to modify the CAN_to_ROS module.

5.4.1 First version - Proof of Concept

Before going into the details of the translation of the messages, a first version of the CAN_to_ROS module has been developed, to be used as a Proof of Concept of the whole system and for testing

purposes.
The code has been written in Python, using the Python-Can [12] library for interfacing with the CAN bus.
The translation is done in a very simple way, using a Python associative array (dictionary) for storing the configuration.
The configuration of the module is written in the following way:

```
# From CAN to ROS = ROS publishers
# This contains any data coming from CAN bus
id_to_topic = {
    0x21: ("HV_battery/measure", Float32), # should be

    0x12: ("FC/hydrogen_level_rate", Float32),
    0x13: ("FC/measure", Float32),

    0x22: ("LV_battery/measure", Float32),

    0x50: ("cockpit/flight_mode", Float32),
    0x51: ("cockpit/roll", Float32),
    0x51: ("cockpit/pitch", Float32),
    0x52: ("cockpit/throttle", Float32),

    0x60: ("imu/data", Float32),
}

# From ROS to CAN = ROS subscribers
# This contains all information sent to the CAN bus
topic_to_id = {
    "actuation/motor1_setpoint": (0x21, Float32),
    "actuation/motor2_setpoint": (0x21, Float32),
    "actuation/motor3_setpoint": (0x21, Float32),

    "propulsion/motor1_setpoint": (0x21, Float32),
    "propulsion/motor2_setpoint": (0x21, Float32),

    "FC/power_setpoint": (0x21, Float32),

}
```

Figure 10: First Version CAN messages configuration

The *id_to_topic* dictictionary associates the AID of each CAN message to a topic name and a message type. This is used for parsing data coming from CAN and to create the publishers to publish the data.
The *topic_to_id* array associates the topic names to the CAN AIDs. This is used to create the subscriptions to the topics and to do the encoding for data to be sent over the CANBUS network.

5.4.2 First version limitations

The main limitation of the first version, is that there is a direct connection between a CAN message type and a topic. A CAN message is translated to a single topic message, and a message is translated to a defined CAN message.
The problem of this, is that the ROS graph and the topics are at a higher level than the CAN messages, and this is due to the size limitation of a CAN message.
For this reason, it is important to conceive and design a system that can adapt to these two levels of abstraction, while maintaining the modularity and the possibility to easily configure the system and adapt to changes during further development.
Therefore, it is important to have a flexible and modular configuration of the system, which allows the addition of new components to the CAN network and configurations of its message with ease.
In the existing Dahu system, a single monolithic configuration file contains the whole CAN message dictionary, which makes it difficult to maintain and adapt.

```yaml
- name: height_sensor
  canID: 0x3D3
  fields:
    - name: height_sensor_status
      unit:
      format: uint8_t
      offset: 0
      scaling: 1
    - name: height_measure
      unit: "m"
      format: uint16_t
      offset: 1
      scaling: 0.001
```

Figure 11: Excerpt from Dahu CAN dictionary

Moreover, in this first version the parsing of the CAN messages is hardcoded for each type of message, which is not optimal and can be improved to be based on the configuration file.

5.4.3 DBC CAN database files

The industry standard for maintaining CAN dictionaries is the DBC format [16].
The DBC format allows the definition of dictionaries of different messages in an easy and organised way [15].
There are many tools readily available which provide the parsing support from the DBC file, which makes the development of the final version of the *CAN_to_ROS* module easier and more easily maintainable and configurable.
The one that seems to be interesting to be easily integrated in the existing codebase and provides all the necessary features is the Python library Cantools [13].
It provides encoding and decoding of the messages based on a DBC dictionary, and it is very well documented and vastly used for similar applications.

6 Development Environment

The project is about building a framework for the Onboard Computer. During next semester, all the submodules will be conceived and developed on top of this framework. For this reason, part of my work has been to create a development environment to ease the next phases.
I wanted to provide the tools to be able to easily develop, test and deploy the systems.

6.1 Docker Image

The first tool developed is a Docker Image.

Figure 12: Docker Logo

Docker is a virtualization software, which is used for the deployment of applications in lightweight containers so that applications can work in different environments in isolation [22].
The creation of a Docker Image allows the testing of the ROS system in a defined environment, which is the same as the one that will be deployed on the Onboard Computer.
This eliminates the need of having the actual hardware to test the rolling development advancements.
The Docker Image has been designed as a development environment, so it contains all the tools needed for developing the subsystems, starting from ROS building tools, to NodeJS environment for the Ground Station, and tools for dealing and debugging the CAN connection.
A fair effort has been put in the writing of a good documentation to help others get started with the development in the Docker Image, small utilities scripts have been written to automate the building and launch of the container.
A small excerpt of the *README* documentation can be seen below:

Virtual CAN setup

☐ Run the following commands to setup the virtual CAN for testing purposes. Note that it is necessary to run them at each reboot.

```
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
```

Repository

☐ Clone this repository and move to its directory:

```
git clone https://gitlab.epfl.ch/swiss-solar-boat1/ref/onboard-computer/docker-development.git
cd docker-development
```

☐ Run the build script. It will download the ssb-ros2 module and generate the docker image (the first time it will take some time, as it will download the ros-humble image from repositories):

```
source ./build.sh
```

☐ Create a new container and run it. The ros2 workspace will be built (colcon build) automatically at startup.

```
source ./run.sh
```

☐ To attach another terminal to the existing container:

```
source ./attach.sh
```

Figure 13: Excerpt of the documentation

The documentation has been written during the development of the Docker Image and tested on my machine, but it has also been tested by other members of SSB which have started to work on the system, and it has been improved during the use by them.

6.2 GIT repositories

To provide a functional working environment for the next semester, also big attention has been paid to the creation of an efficient structure of GIT repositories.
The designed ROS architecture is a complex system, in which many submodules are developed separately and integrated afterwards, so particular attention has to be paid in maintaining the versioning of all the codebase.
The idea has been to create a main repository, called *SSB ROS2*, which serves only as a placeholder to keep the versions of all the working submodules, which are contained in different GIT repositories.
So the structure of the GIT repositories is the following:

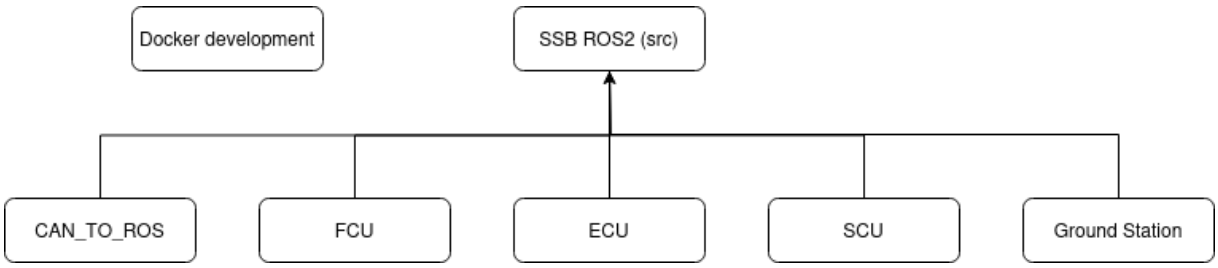

Figure 14: Structure of the repositories created

- **Docker development** - contains the Dockerfile for the generation of the Docker image and the scripts to automate the development
- **SSB ROS2** - Contains only references to submodules, it is used to maintain a coherent and working version of the whole system during development, it is used as the *src* folder of the ROS2 workspace in the deployment.
- **CAN_to_ROS** - Contains the node *CAN_to_ROS*, at the moment written in Python.
- **FCU** - Contains the template of the FCU node.
- **ECU** - Contains the template of the ECU node.
- **SCU** - Contains the template of the SCU node.
- **Ground Station** - Contains the template of the Ground Station node, in NodeJS.

6.3 Node templates

In the created repositories, ROS packages have been created to contain the nodes that will be developed. In the packages, a draft of the node has been included, with sample subscriber and publisher definition. This has been done for various reasons:

- To test that the system is behaving as planned.
- To provide a template for the future development of the code.
- To have the graphical representation of the ROS graph as provided by the *rqt_graph* ROS tool.

The templates for the nodes can be found in the GIT repositories of the team.
For completeness, a template of a sample generic node in Python, C++ and NodeJS has been provided in the Appendix of this report.

7 Issue with selected Onboard Computer

The main issue faced during the project, which has taken a lot of time and slowed down the development of the system, has been a problem with the hardware of the Onboard Computer.
During last semester, in the project of Mathias Arnold [2], the Khadas VIM4 [18] has been chosen as the OBC, a shield has been developed to be connected to it, and provide CAN connectivity.
As the Khadas VIM4 does not provide a CAN interface, 3 MCP2515 chips have been used to convert CAN to SPI, which is present on the VIM4.

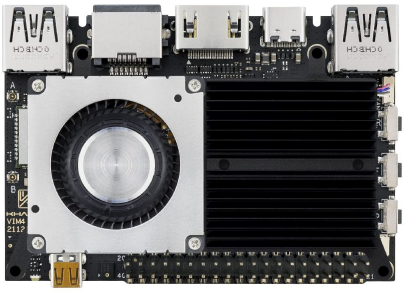

Figure 15: Khadas VIM4

7.1 MCP2515 Configuration

Linux usually exposes the CAN line as a normal network interface. For this to happen, there is the need of a kernel module which communicates via SPI to the MCP2515 chip.
In the official Linux kernel, the mcp251x module [17] is included, which can manage up two MCP2515 chips on the same SPI bus.
The module has to be configured, using the Device Tree Overlay [19] configuration.
The Device Tree allows to configure the hardware bindings of the SOC (System on Chip), to define to which pin of the processor is the MCP2515 chip connected, at which frequency is the chip running, how is the interrupt needed handled, etc.. This configuration is very specific to the SOC used, and is usually provided as part of the stock distribution that comes with the device.
In the stock Ubuntu Image provided by Khadas there is no such file, so it has to be coded and compiled.
The configuration is made in *.dts* files, which are then compiled to *.dtb* (Device Tree Blob) files by the use of the dtc command (Device Tree Compiler).
The documentation provided by the vendor for the VIM4 has not been sufficient to gather the information needed for an easy configuration of the DT.
The only resource found online was a forum page related to a user who manage to get the MCP2515 chip working with a different version of the SOC, the VIM3 [20].
Multiple tests have been done to try to get to the right configuration, but they were all unsuccessful.
A forum thread has been started to try to seek help from the small community and from Khadas employees [21].
The advice received from the forum has lead to advancements in the comprehension of the problem, but not to a final solution.

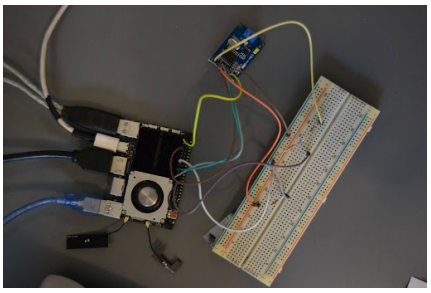

Figure 16: VIM4 during tests with MCP2515 chip

7.2 OBC considerations

Regardless of the effort needed to solve the problem and make the CAN communication working on the Khadas VIM4, many considerations have been made on the choice of this Onboard Computer. The lack of thorough documentation and support by community may create other problems during further development.
The VIM4 platform is interesting from a hardware point of view, as it has the computational power for the purpose of the OBC, but it may be difficult to deploy due to the aforementioned problems. For this reason, other possibilities have been evaluated.
A Raspberry Pi 4 has been bought, as it is known to be working with the MCP2515 chip, it is already used on the Dahu, and it has been used for testing the whole system.
During the semester, the Raspberry Pi new version 5 has become available, so it has been bought, but it has arrived only during the last week, so it has not been tested yet.
A good characteristic of the ROS system is that it is portable. As long as ROS can be installed, it is completely Operating System agnostic. The only component which is dependent on the OS is the CAN communication, but the CAN is usually exposed as a Linux network interface, which does not change depending on the hardware platform.

8 Tests

To prove that the idea developed makes sense, that the CAN_to_ROS module first version and the whole system was working, a test of the architecture has been conceived.
The idea was to test a simplified version of all the components of the system, from the CAN connection to the communication with the Ground Station.
An Arduino has been used to emulate a sensor of the boat. It has been used to read data from a potentiometer and send this data over the CAN network using a MCP2515 chip.
A Raspberry Pi 4 has been used as the Onboard Computer. ROS is running on it, with the CAN_to_ROS module active for the CAN communication using the MCP2515 chip, as will be in the final configuration.
The Raspberry Pi 4 has been connected to a Wireguard VPN to communicate with the Ground Station server. The server was also running ROS, receiving the topics sent from the Onboard Computer.
A schematic of the conducted tests together with a picture of the test setup can be seen below:

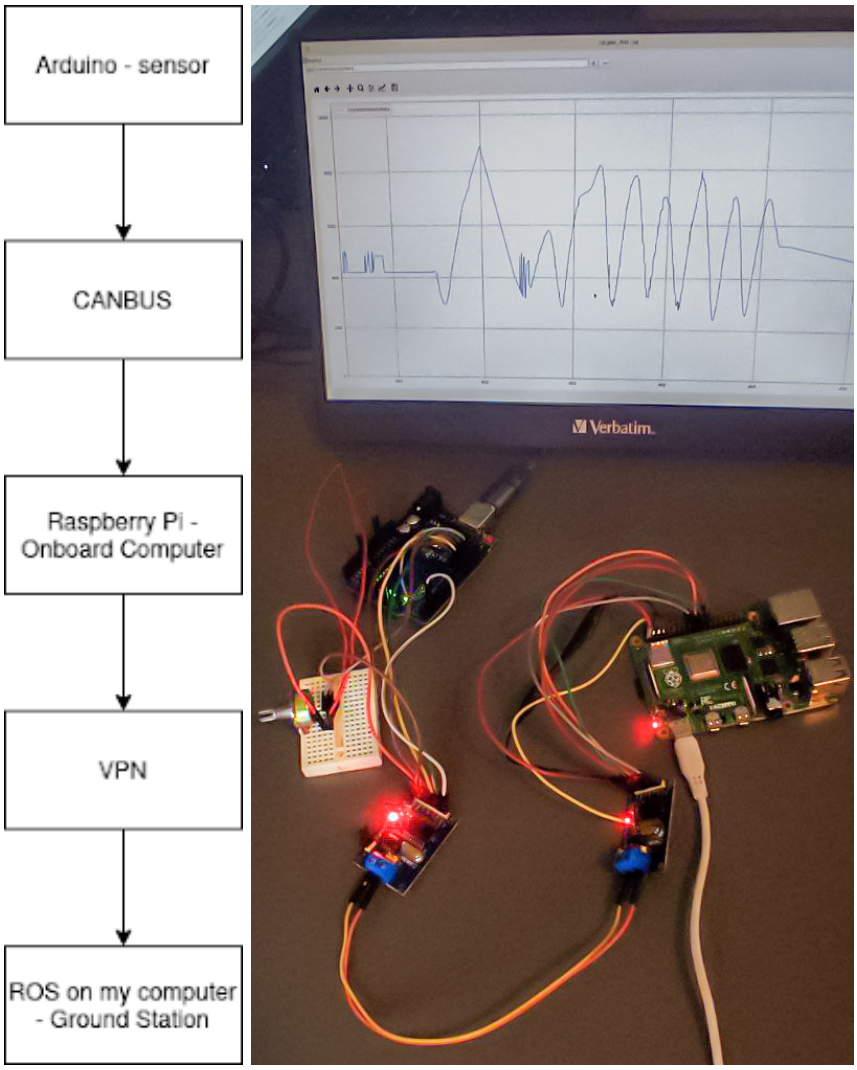

Figure 17: Schematic and Test setup

The tests have lead to promising results. The system is working as expected, and although the performances would need further investigation and evaluation, it worked well as a Proof of Concept, and it seems the development is going in the right direction.

9 Next steps

9.1 CAN_to_ROS

As described in Section 5.4, the actual version of the *CAN_to_ROS* module is just a Proof of Concept, and it has many limitations for the future real-life usage.
The specifications of the final module have been clearly defined, and an evaluation of the tools to be used for the development has been done.
During next semester, a new version of the module will be written which will overcome the limitations and provide a sound and strong structure for the whole architecture.

9.2 Useful ROS tools

As previously stated, ROS offers a vast range of different modules and tools, which can be useful for the development. One of the tasks for the next semester will be to evaluate those tools and understand which ones can be used and for what.
A non-exhaustive list of the tools to be evaluates is the following:

- rosbag and rqt_bag for data logging and playing back
- rqt_plot for live plotting
- rqt_graph for system visualization
- RViz for graphic data visualization
- Gazebo for simulation
- micro-ROS to be run on microcontrollers
- The TF Subsytem
- Foxglove Studio
- Webviz

9.3 Onboard Computer choice and Electronics

Given the problems faced with the configuration of the Onboard Computer, many considerations have been made on the choice of the model to be used.
At the moment there are three possibilities:

- **Khadas VIM4** - powerful hardware platform, lack of good documentation and community, MCP2515 still not working.
- **Raspberry Pi 4** - wery well estabilished platform, very good documentation and community, MCP2515 working and tested, platform already used on the Dahu.
- **Raspberry Pi 5** - new platform from Raspberry Pi, still without much resources on the net but with promising future. More powerful than the 4 version, exposes a PCIe bus which could be interesting for the 4G module.

During the next semester a final choice has to be made, which could eventually lead to the need of the redesign of the PCB shield previusly designed for the Khadas VIM4.

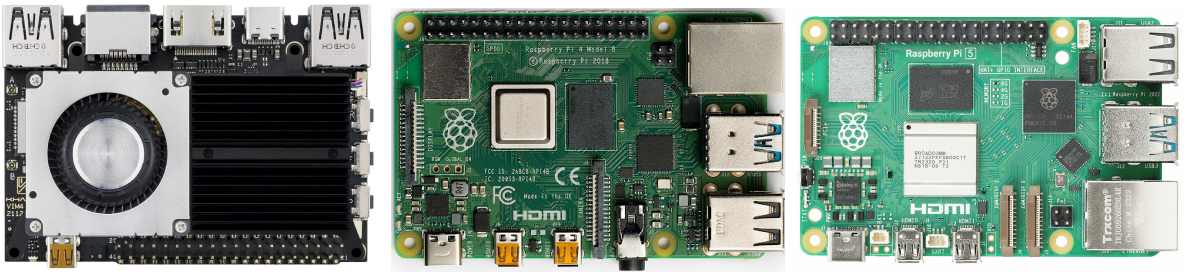

Figure 18: From left to right: Khadas VIM4, Raspberry Pi 4, Raspberry Pi 5

9.4 Network Architecture

The Onboard Computer will have the need to communicate with the Ground Station. To use ROS to communicate, the two devices have to be in the same network in order for the ROS middleware to establish the communication. During the tests, a Wireguard VPN with a server based in Belgium has been used to prove that the idea works.
However, this area has to be further explored to understand what will be the optimal configuration for the boat.

10 Appendix

10.1 Abbreviations

- AID - Arbitration ID
- CAN - Controller Area Network
- DLC - Data Length Code
- ECU - Energy Control Unit
- FC - Fuel Cell
- FCU - Flight Control Unit
- GS - Ground Station
- GUI - Graphics User Interface
- HAL - Hardware Abstraction Layer
- IDL - Interface Description Language
- IMU - Inertial Measurement Unit
- IPC - InterProcess Communication
- LCD - Liquid Crystal Display
- LTS - Long Term Support
- OBC - OnBoard Computer
- ROS - Robot Operating System
- RPC - Remote Procedure Call
- SCU - Safety Control Unit
- SOC - System on Chip

10.2 Python Node Template

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MyExampleNode(Node):
    def __init__(self):
        super().__init__('node_example')
        self.publisher_ = self.create_publisher(String, 'pub_topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

        self.subscription = self.create_subscription(
            String,
            'sub_topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1

def main(args=None):
    rclpy.init(args=args)

    my_example_node = MyExampleNode()

    rclpy.spin(my_example_node)

    my_example_node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

10.3 C++ Node Template

```
#include <chrono>
#include <functional>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

class ExampleNode : public rclcpp::Node
{
  public:
  ExampleNode()
    : Node("example_node"), count_(0)
    {
      publisher_ = this->create_publisher<std_msgs::msg::String>("pub_topic", 10);
      timer_ = this->create_wall_timer(
      500ms, std::bind(&ExampleNode::timer_callback, this));

      subscription_ = this->create_subscription<std_msgs::msg::String>(
        "sub_topic", 10, std::bind(&ExampleNode::topic_callback, this, _1));
      }
    }

  private:
    void timer_callback()
    {
      auto message = std_msgs::msg::String();
      message.data = "Hello, world! " + std::to_string(count_++);
      RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
      publisher_->publish(message);
    }
    void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
    {
      RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
    }

    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    size_t count_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<ExampleNode>());
  rclcpp::shutdown();
  return 0;
}
```

10.4 NodeJS Node Template

```
const rclnodejs = require('rclnodejs');

rclnodejs.init().then(() => {
    const node = rclnodejs.createNode('example_node');

    node.createSubscription('std_msgs/msg/String', 'sub_topic', (msg) => {
    console.log(`Received message: ${typeof msg}`, msg);
    });

    const publisher = node.createPublisher('std_msgs/msg/String', 'pub_topic');

    let counter = 0;
    setInterval(() => {
    console.log(`Publishing message: Hello ROS ${counter}`);
    publisher.publish(`Hello ROS ${counter++}`);
    }, 1000);

    rclnodejs.spin(node);
});
```

11 References

[1] Integration of Low Power Electronics: CAN bus and component selection - Kai Wen LOO

[2] Integration of Low Power Electronics: Design Microcontroller and Main Electronics Box - Mathias ARNOLD

[3] Preliminary Design of the REF Electronic systems - Baptiste RANGLARET

[4] ROS2 Documentation - Introduction http://wiki.ros.org/ROS/Introduction

[5] ROS2 Documentation - Concepts http://wiki.ros.org/ROS/Concepts

[6] ROS2 Documentation - Topics - Services - Actions https://docs.ros.org/en/humble/How-To-Guides/Topics-Services-Actions.html

[7] ROS2 Documentation - Package creation https://docs.ros.org/en/humble/How-To-Guides/Developing-a-ROS-2-Package.html

[8] ROS2 Introduction https://www.toptal.com/robotics/introduction-to-robot-operating-system

[9] Offenburg University - Black Forest Formula Team - similar application https://github.com/Black-Forest-Formula-Team/bfft_can_bus_msgs_to_ros_topic

[10] Wikipedia - CANBUS https://en.wikipedia.org/wiki/CAN_bus

[11] Dahu CAN messages database https://docs.google.com/spreadsheets/d/1QLoGUTFDRtmR8xjmXSLVbKhhBwtYzgOz1sdkR_hTFdo/edit#gid=665153634

[12] Python-CAN documentation https://python-can.readthedocs.io/en/stable/message.html

[13] Cantools documentation https://cantools.readthedocs.io/en/latest/

[14] Similar Application to read data from a Tesla CANBUS https://medium.com/starschema-blog/collecting-can-bus-data-with-ros2-and-qt-11cd4e4e0918

[15] DBC format specification https://github.com/stefanhoelzl/CANpy/blob/master/docs/DBC_Specification.md

[16] DBC format https://www.influxtechnology.com/post/understanding-can-dbc

[17] MCP251X kernel module https://github.com/torvalds/linux/blob/master/drivers/net/can/spi/mcp251x.c

[18] Khadas VIM4 https://docs.khadas.com/products/sbc/vim4/start

[19] Linux Kernel Device Tree Overlays https://docs.kernel.org/devicetree/overlay-notes.html

[20] CANBUS using MCP2515 on Khadas VIM3 - forum page https://forum.khadas.com/t/setting-up-can-bus-mcp2515-at-linux-6-2/19950

[21] Khadas forum page opened to seek for help https://forum.khadas.com/t/can-bus-mcp2515-on-vim4/20817/13

[22] Wikipedia - Docker https://en.wikipedia.org/wiki/Docker_(software)

Note: All the links included in the Reference have been accessed on January 3, 2024.