

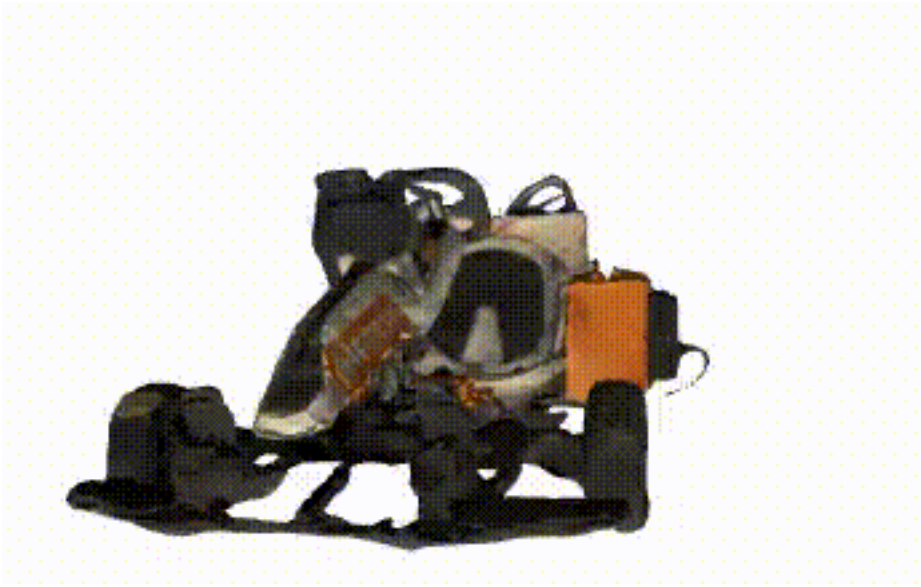
Contents

1 Autonomous Platform (Gen4)	4
1.1 Introduction	5
1.2 Features and Capabilities	6
1.2.1 Drive by wireless controller	6
1.2.2 Remote access to platform & software	6
1.2.3 Accelerating, Brake & Steering	7
1.2.4 Scalability	7
1.2.5 Power Module	7
1.2.6 Digital Twin	7
1.3 System Architecture	7
1.3.1 High-level control	8
1.3.2 Low-level control	8
1.3.3 Embedded control	9
1.4 Start AP4	9
1.5 Connect (wireless) and start AP software stack	10
1.6 Repository overview	11
1.6.1 Generation 4 - changelog	11
1.6.2 Version 4, June 2023	12
1.6.3 Version 4.1, July 2023	13
1.6.4 Version 4.1.1 August 2023	13
1.7 Hardware Design Principles	14
1.7.1 Modular Mounting of Components	14
1.7.2 General idea	14
1.7.3 Mounting the aluminum plates to the frame	15
1.7.4 Component list	17
1.8 Software Components	19
1.8.1 Information and Background	19
1.8.2 IDE	20
1.8.3 Linux Ubuntu 22.04	20
1.8.4 Git	20
1.8.5 Containerization	20
1.8.6 Containers	21
1.8.7 Robot Operating System 2 (ROS2)	21
1.8.8 PlatformIO & VSCode	26
1.9 Project standards and conventions	26
1.9.1 Git commit messages	26
1.9.2 Merge commits	26
1.9.3 Documentation	26
1.10 Automated Testing (CI/CD)	26
1.10.1 Resources for automated testing of ROS2 software - Jenkins	27
1.10.2 Resources for automated testing of embedded software platformio with Jenkins	27
1.10.3 Offline local PlatformIO software tests	27
1.11 Component Communication	27
1.12 Onboarding Procedure	28
1.12.1 Schedule	28
1.13 Useful skills/software knowledge	29
1.14 Extend AP	29
1.14.1 Design Principles	29
1.14.2 Software	29
1.14.3 Hardware	30
1.14.4 Documentation:	30
1.14.5 Prerequisite Software & Skills	30
1.14.6 Adding Wheel Speed Sensor	30
2 Embedded Software and Hardware (CAN nodes)	35
2.1 Introduction	35
2.1.1 Prerequisites	35
2.1.2 Introduction	35


2.1.3	Centralized E/E Architecture Principles	36
2.1.4	Embedded ECUs	37
2.1.5	HW_Node_SPCU : Steering and Propulsion Control Unit	37
2.1.6	HW_Node_SSCU : Speed Sensor Control Unit	39
2.1.7	Generic ECU Hardware and Software	39
2.2	Generic ECU Template	40
2.2.1	Key Components of Generic ECU	41
2.2.2	Software Template and Setup	42
2.2.3	Bill of Material for Generic ECU	43
2.2.4	Interfacing with Hardware Interface and Low Level Software	44
2.2.5	Unified CAN Database	44
2.3	Steering Propulsion Control Unit (SPCU)	46
2.4	Speed Sensor ECU	49
2.4.1	Functional block diagram	49
2.4.2	Wiring overlay	49
2.4.3	Circuit diagram	49
2.5	How to extend Embedded Software	50
2.5.1	Prerequisites	50
2.5.2	Add New Functionalities	51
2.5.3	Standard base ECU	51
2.5.4	build a new ECU	51
2.5.5	Extend with new function specific ECUs	67
2.5.6	Unified CAN database and how to modify	67
2.5.7	Adding New CAN Frames And Signals	71
2.5.8	Controller Area Network Database	71
2.5.9	dbc editor	72
2.5.10	How to use DBC to C library	72
3	Low-level Control & Hardware	73
3.1	Introduction	73
3.2	CAN Signals To ROS2 Topic Converter	74
3.2.1	Vehicle Control	74
3.2.2	Joystick Manual Control	74
3.2.3	Convert Data Format To ROS2 Standard	74
3.2.4	Low-Level Control & Hardware Interface - Software / Hardware Requirements	75
3.2.5	USB Devices Connected	75
3.2.6	Interfacing With Embedded ECUs	75
3.2.7	Interfacing with High Level Control Software	77
3.2.8	Adding New Functionalities	78
3.2.9	Automatic Startup of Software	78
3.2.10	Installing Base Software on Fresh Raspberry Pi 4b	78
3.3	Extend Hardware Interface and Low Level Software	78
3.3.1	Prerequisites	78
3.3.2	How to Add New Functionality	79
3.3.3	Software functionality	79
3.3.4	Sensors	79
3.3.5	USB Based Sensors	79
3.3.6	Embedded Microcontroller Sensors	79
3.4	Setup Hardware Interface Low Level Code (Raspberry Pi)	83
3.4.1	Required Hardware	85
3.4.2	Installing Base Software on Fresh Raspberry Pi 4b	86
3.4.3	Install GPIO library on raspberry pi running ubuntu	86
3.4.4	Install can-utils library	86
3.4.5	Raspberry pi 4 configuration	87
3.4.6	Setting the CAN speed	87
3.4.7	Fixing problems with GPIO pins on raspberry Pi	87
3.4.8	Setting up automatic start of software upon boot-up	87
3.4.9	Testing and debugging (system level)	88
3.5	Testing and verifying basic functionality	88

4	High-Level Control Software	88
4.1	High-Level Control Hardware Requirements	88
4.1.1	How To Start	89
4.1.2	Autonomous Driving Stack	91
4.1.3	Digital Twin	91
4.1.4	Design Of Digital Twin	92
4.1.5	Sensor plugins	93
4.1.6	Gazebo worlds	93
4.1.7	Software Control Switch	94
4.1.8	Digital Twin Software Package Structure	94
4.1.9	How to Connect to Physical Platform	95
4.1.10	How to verify	95
4.1.11	High Level Control Underlying Software Components	95
4.1.12	Containerization	95
4.1.13	Robot Operating System 2 (ROS2)	95
4.2	Extend High Level Software	96
4.2.1	Prerequisites	96
4.2.2	Add a New Functionality	96
5	Power Module	96
5.0.1	Battery and Charger	98
5.0.2	User interface	99
6	Testing, Debugging, Known Issues and Future Work	101
6.1	Test and debugging (System level)	101
6.1.1	Possible Errors When Starting AP4	101
6.1.2	Docker Container starting/Building	102
6.1.3	Testing and Debugging Components	105
6.2	Test and Debugging of Hardware Interface Low Level Computer	108
6.2.1	Raspberry Pi 4b Bootup failed	108
6.2.2	Software container not started?	108
6.2.3	No ROS2 topic shows in terminal	109
6.2.4	ROS2 Node did not start when starting container	109
6.3	Testing and Debugging High Level Control Software	109
6.3.1	Building the high level control software docker container	109
6.3.2	Gazebo and Rviz does not show up when starting the container	109
6.4	Known issues	110
6.4.1	Priority : Fix known Bugs / Issues	110
6.5	Future work	111
6.5.1	Improving Generic ECU nodes with PCB	111
6.5.2	Power module	111
6.5.3	High Level Control Software	112
6.5.4	Digital Twin	112
6.5.5	Mounting High Performance Computer on AP4	112
6.5.6	Sensor Measurements	112
6.5.7	Camera	112
6.5.8	IMU	112
6.5.9	Sensor Fusion	113
6.5.10	Testing out existing Autonomous Drive algorithms	113
6.5.11	Integrating sensors used in the automotive industry	113
6.5.12	Naming of sensor messages convention (sensor interfaces)	113
6.5.13	Naming of navigational messages convention (interface)	114
6.5.14	Automatic and dynamic generation of ROS2 package 'can_msgs_to_ros2_topic_pkg'	114
6.5.15	Configure docker containers to run on Windows	114
6.5.16	Remote flashing of ECU software over CAN bus	114
7	CAD Appendix	115

1 Autonomous Platform (Gen4)




This is the repository for Autonomous Platform at Infotiv Technology Development. The new generation of Autonomous Platform was created by **Fredrik Juthe** and **Erik Magnusson** as part of a master thesis at Chalmers University of technology spring 2023 under supervision of **Hamid Ebadi**. The master thesis designed the system E/E architecture and implemented a base on which future functionality could be added to. A published version of the thesis with the title of “**Design of a modular centralized E/E and software architecture for a small-scale automotive platform**” is

accessible in [in this link](#)  .





- The **latest public** version of documentation and source code for Autonomous Platform (AP) project is available in the following address: <https://infotiv-research.github.io> and <https://github.com/infotiv-research>.
- The printer friendly version of this document is **available in this link**  .

1.1 Introduction



Autonomous Platform (generation 4) project is a platform on which internal and external research projects can be tested on. (i.e Autonomous Drive algorithms) and to expand the knowledgeable in different aspects of software and hardware system development. The purpose of this repository is to collect all the software components in one mono-repository, meaning all software (& hardware design) for the different components is placed in a single repository. Any future work on autonomous

platform should be integrated into one of the repositories sub directories. This means that any future development or research can always refer back to previous work, making it a viable long term project at Infotiv AB.

1.2 Features and Capabilities

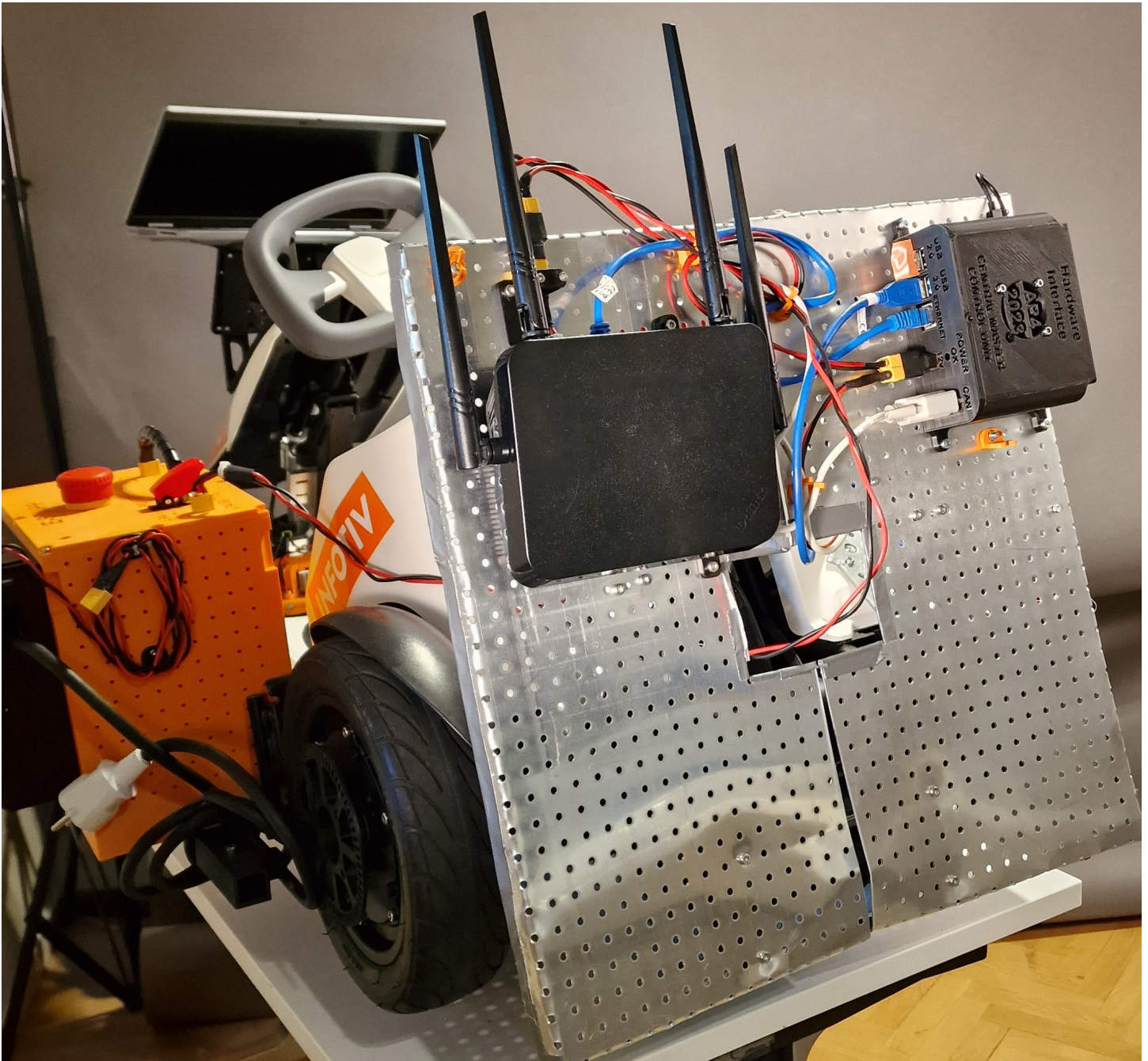
Below, the capabilities and functions of autonomous platform generation 4 are summarized.

1.2.1 Drive by wireless controller

AP4 is capable to be remotely controlled using an xbox 360 controller. The xbox wireless adapter should be connected to the Raspberry Pi 4b usb port.

1.2.2 Remote access to platform & software

A wifi router is mounted on the back plate, this allows developers to connect to the software whilst it is running to monitor internal states. High-level software can stream commands to the low-level software remotely.



1.2.3 Accelerating, Brake & Steering

The platform can be driven forwards, backwards and can be steered. This is done programmatically, meaning “Drive by Wire”. Joysticks commands from the xbox controller can therefore be configured to control the platform. In the same way it is trivial to take commands from Autonomous Drive algorithms to control the platform.

1.2.4 Scalability

The idea with autonomous platform generation 4 is that it should be a scalable base platform on with future functionality should be added onto. Scalability and modularity is therefore a key concept for the platform.

Hardware wise, every component can be mounted where it is deemed appropriate using a pre defined hole pattern. It is possible to move existing hardware as time goes on. New hardware can be added as long as it uses the pre defined hole pattern present on the metal sheets.

Embedded sensors can be added using ECUs following a set template. These can then be added to the existing system without breaking backwards compatibility using a defined CAN bus network. High data throughput sensors can be added to the platform using USB connection.

The software is built to be scalable and modular. The software structure is described in detail below. The software is deployed inside docker containers, making it easily deployable and scalable. The controlling software is written using the Robot Operating System 2 (ROS2) Framework. This allows developers to easily add onto existing software.

1.2.5 Power Module

Consists of a led acid battery and a power supply unit, both can be connected to power the autonomous platform. Meaning the platform can be powered whilst driving around from battery. Whilst in a lab environment power can be supplied to the platform through a wall socket.

1.2.6 Digital Twin

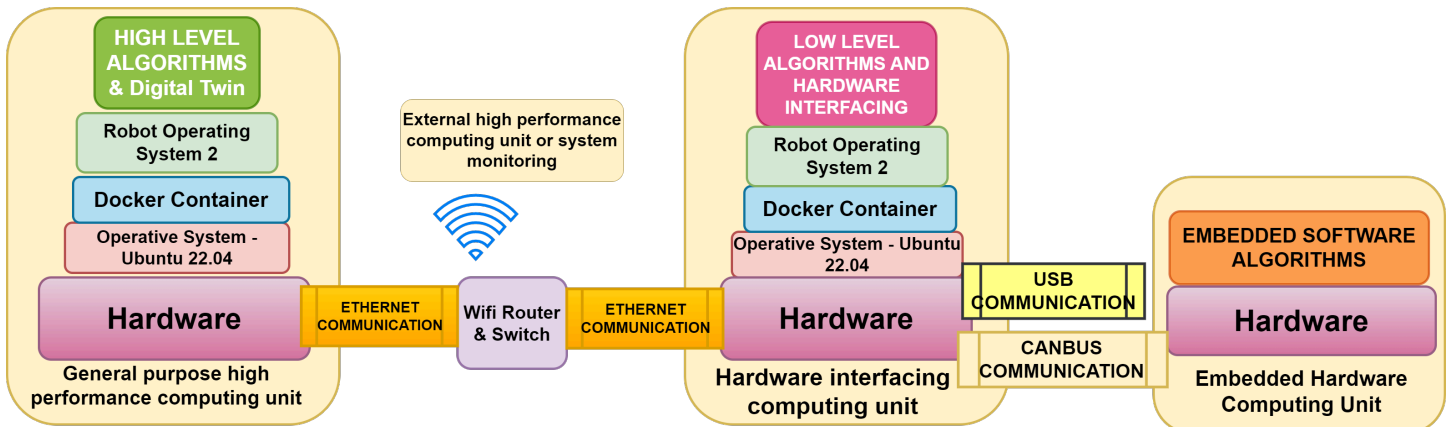
The digital twin is implemented in the high-level software. It is described in detail in `High_Level_Control_Computer/README.md`. It is automatically started when the high-level docker container is started.

1.3 System Architecture

The autonomous platform has three different software components:

- High-level control
- Low-level control
- Embedded control

High-level is supposed to be run on a high performance computer and can be run without being connected to the rest of the system. Low-level control is run on the Raspberry Pi 4b mounted on the hardware platform. The embedded control is run on the ECU nodes mounted to the platform. Software in high-level and low-level are communicating using the Robot Operating System (ROS2) framework. Software in the embedded control is written using the Arduino framework.

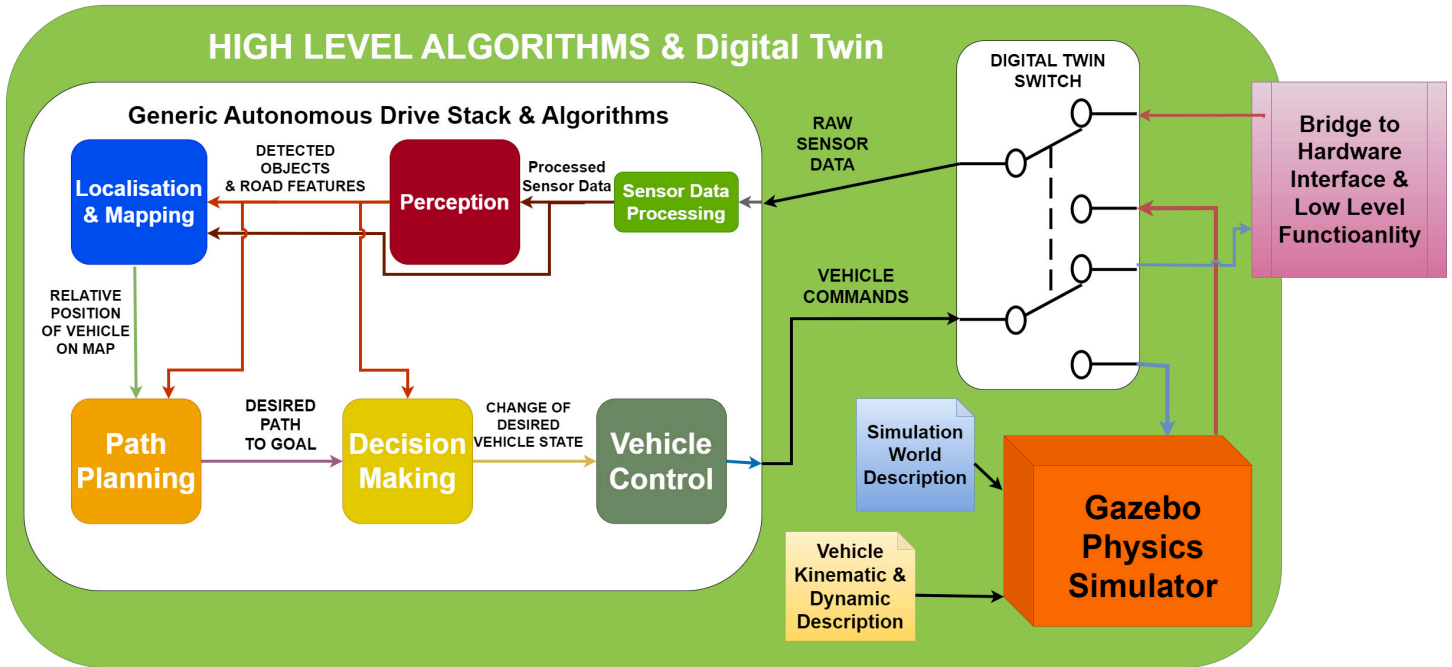


The three software components are connected through hardware and software links. High-level and low-level software is connected using an ethernet interface, meaning the two softwares should be run whilst connected to the same network. The

low-level control is connected to the embedded control through a CAN bus network. A custom CAN library and interface has been created to link the embedded control with the low-level control.

1.3.1 High-level control

The high-level software consists of three components, an autonomous driving algorithm and a digital twin interface with the Gazebo physics simulator and a switch which sends commands to either the digital twin or the physical platform. As of June 2023 only a simple digital twin is implemented in this software layer. Starting up the high-level control container will start up Gazebo and Rviz. See `FUTURE_WORK.md` for what could be implemented.



In the illustration above, three components can be seen. Autonomous driving stack, digital switch and gazebo physics simulator. These components should be implemented as a set of ROS2 computational nodes. Note: This is the intended structure but has not been implemented yet.

- **Autonomous Drving Stack** is the proposed set of algorithms which takes in sensor information and responds with suitable actuator control commands.
- **Digital Twin Switch** is the proposed software switch that is supposed to re-route AD commands to either the physical platform or the digital twin. In the same way, sensory data should be routed from the physical platform or digital twin. The same actuator commands and sensor data type should be used on the digital twin as the physical platform. This ensures that proposed AD algorithms can first be tested on the digital twin and then seamlessly be tested on the physical platform.
- **Gazebo Physics Simulator** Is the proposed digital twin simulation environment. It integrates seamlessly with the ROS2 framework. The physical dimensions and driving dynamics can be configured in xml format. Different simulation environments can be setup and saved as world files. Meaning different AD scenarios can be tested easily.

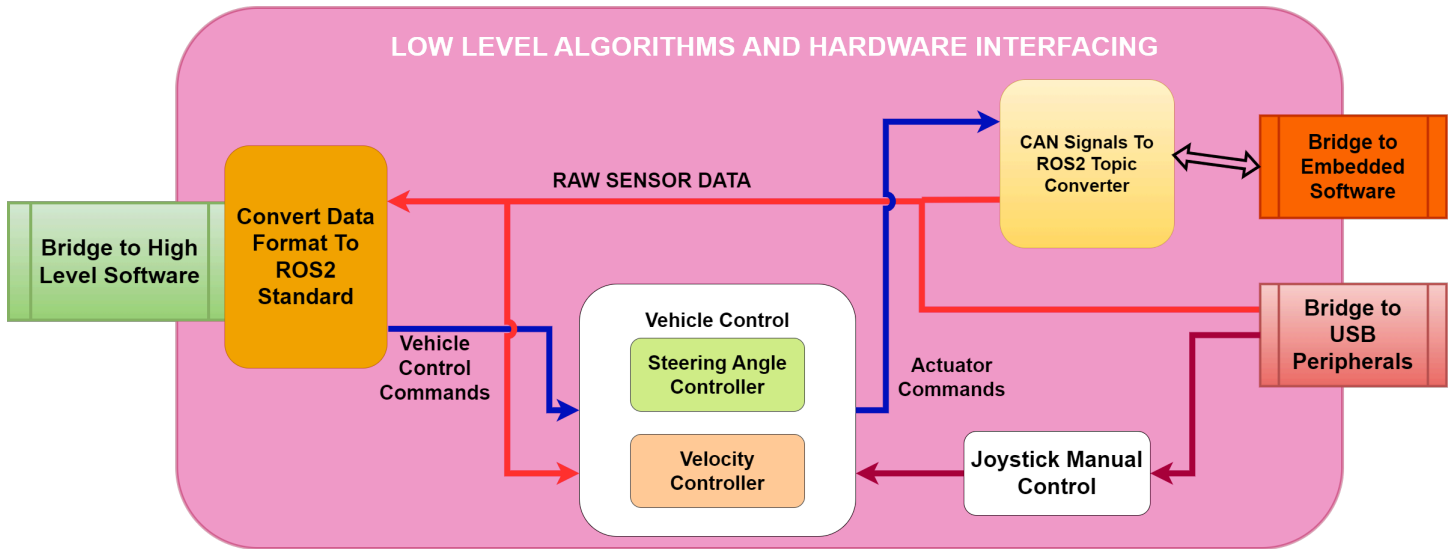
The high level control folder, `High_Level_Control_Computer` consist of a docker container and a folder with software. The software can start a simple digital twin of Autonomous platform Generation 4. The folder contains a `docker-compose` file which starts a docker container and mounts the `ap4_hlc_code` folder in the container. A ROS2 launch file is then run which starts up the digital twin. The container can be started on any device supporting graphics, i.e a laptop.

1.3.2 Low-level control

The low-level software is responsible for taking platform commands sent from higher-level software and convert it into commands which are sent to the actuators. It is implemented on a Raspberry Pi 4b. A simple vehicle control converts commands sent on `\cmd_vel` ROS2 topic into ROS2 topics which gets relayed to the embedded control. In the same way the low-level software is responsible for taking any sensor input and passing it forwards to higher-level software. A custom package to convert ROS2 topics into CAN bus has been created, making it simple to interface high-level software with the embedded control software.

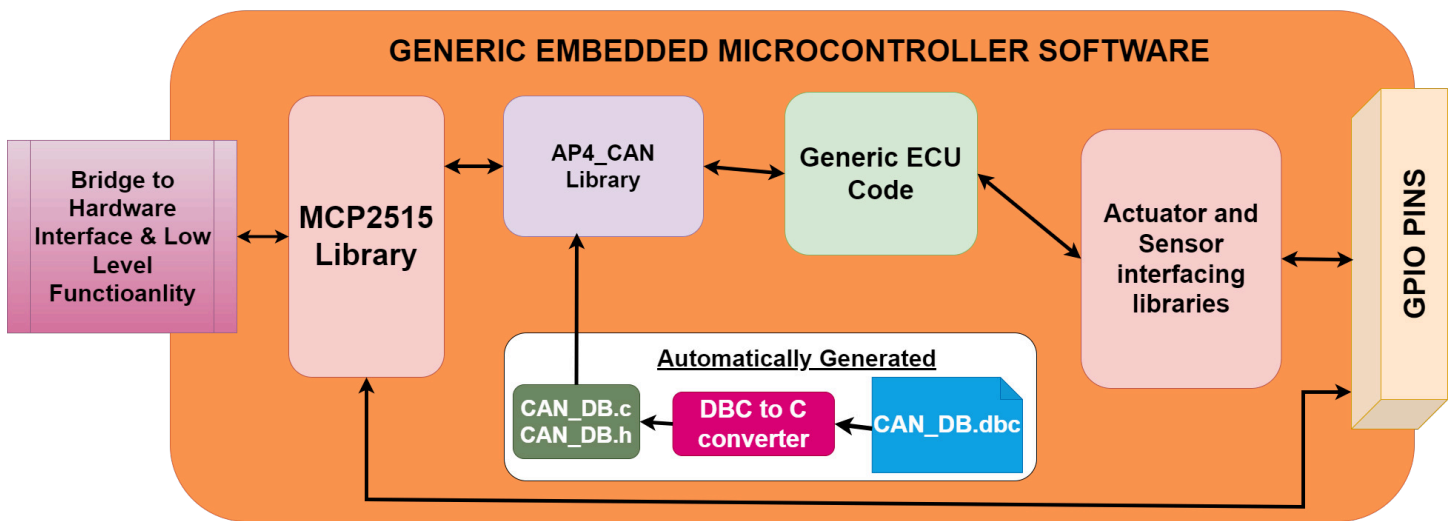
The hardware interface folder, `Hardware_Interface_Low_level_pc` , contains a `docker-compose` file which starts a docker container and mounts the `ap4_hwi_code` folder in the container. A ROS2 launch file is then run which starts up software that

interfaces with hardware. This software is specific for the hardware mounted on autonomous platform, therefore it can only be started properly on the Raspberry Pi 4b mounted onto the platform.



1.3.3 Embedded control

The embedded software is implemented on the ECUs placed on the platform. This firmware acts as an interface between hardware (actuators or sensors) and pass data to and from the low-level software. Currently a single embedded software is implemented, Steering and Propulsion Control Unit (SPCU).



CAN_Nodes_Microcontroller_Code folder contain PlatformIO projects for each ECU, that is the embedded software running on the generic ECUs. An example of ECU is the SPCU (Steering, Propulsion Control Unit).Furthermore here is the CAN database and the corresponding CAN-library located.

1.4 Start AP4

Below follows a quick guide on how to start autonomous platform generation 4 and control it using an xbox 360 controller.

- Connect xbox wireless adapter to Raspberry Pi 4b
- Connect Power source: (Battery or PSU) to POWER IN plug.
- Release Emergency stop
- Flip start switch
- Turn on Nine-bot segway (on backside of platform) Segway will beep once.

System will now boot up. The wheels will move back and forth.

- Wait 60 seconds (hardcoded in the SPCU ECU unit)

The platform should beep again after waiting 60 seconds and turn right to left, meaning it is ready to use.

The platform should be controllable using the xbox controller. By holding down the X button and moving the left joystick the front wheels should move and back wheels turn forwards.

1.5 Connect (wireless) and start AP software stack

To check that the software is up and running as expected one can connect to the platform wirelessly and monitor the ROS2 network. On an external computer (dev-laptop or other linux computer) start the high-level software container. Make sure the computer is connected to the wifi network of autonomous platform.

NOTE: User credentials are available only in the infotiv internal gitlab repository.

NOTE: high-level software automatically starts the digital twin simulation environment, the RVIZ and gazebo windows can be ignored.

In a new terminal, start the docker container:

```
cd High_Level_Control_Computer
docker-compose build
docker-compose up
```

In a new terminal, enter the docker container

```
docker exec -it ap4hlc bash
```


Source environment variables and export configuration environment variables

```
cd ap4hlc_ws
source install/setup.bash
export ROS_DOMAIN_ID=1
```

The low-level software components should be visible to the high level software components. This can be checked with

```
ros2 node list
```

The expected output will be something like this:

A screenshot of a terminal window on a Raspberry Pi. The terminal title is 'root@ap4-hw-interface-rpi4: ~/ap4_hwi_docker_dir/ap4hwi_ws'. The user 'ap4' has executed 'docker exec -it ap4hwi bash'. The prompt is 'root@ap4-hw-interface-rpi4:~/ap4_hwi_docker_dir#'. The user has navigated to the workspace 'cd ap4hwi_ws/' and sourced the setup script 'source install/setup.bash'. They have also set the domain ID 'export ROS_DOMAIN_ID=1'. The command 'ros2 node list' is executed twice, resulting in the following output: '/can_ros2_interface_node', '/feed_forward_ctrl_node', '/joy_node', '/joy_node', '/launch_ros_201', '/socket_can_receiver', '/socket_can_sender', and '/teleop_twist_joy_node'. A warning message is displayed: 'WARNING: Be aware that are nodes in the graph that share an exact name, this can have unintended side effects.' The prompt is now 'root@ap4-hw-interface-rpi4:~/ap4_hwi_docker_dir/ap4hwi_ws#'.

Information sent between nodes (such as sensor data or CAN bus traffic) can be monitored using

```
ros2 topic echo <topic_name>
```

For a list of active ROS2 topics, the following command can be used.

```
ros2 topic list
```

If the topics and nodes show up, the system has started correctly and can be controlled by either an xbox control or any future high-level software control.

1.6 Repository overview

The repository consists of three directories containing software (with a few subdirectories each):

- [CAN_Nodes_Microcontroller_Code](#)
- [Hardware_Interface_Low_Level_Computer](#)
- [High_Level_Control_Computer](#)

Within these directories, there exists useful documentation regarding each software component and sub components.

- [CAN_Nodes_Microcontroller_Code/README.md](#)
- [CAN_Nodes_Microcontroller_Code/CAN_LIBRARY_DATABASE/README.md](#)
- [Hardware_Interface_Low_Level_Computer/README.md](#)
- [High_Level_Control_Computer/README.md](#)

The documentation is split up into smaller parts to keep information manageable and separated according to what can be relevant at a given time. The major documentation files located in the root directory of the repository are:

- [README.md](#) : The documentation you first see when you open the gitlab, this document.
- [SOFTWARE_DESIGN.md](#) : A quick introduction to important software and frameworks used on AP4 and how to install the required software. In illustration of how the network of hardware is connected is also can be found there.
- [HARDWARE_DESIGN.md](#) : An introduction to hardware that is used on AP4 and how to add a new one.
- [HOW_TO_EXTEND.md](#) : Are you a new team member? Look at this file for a quick introduction to the project and how to get going. It also has a general design principles for the project to keep in mind when adding new functionality.
- [TEST_DEBUGGING.md](#) : Contains a list of known errors when working with AP4 and how to solve them
- [ISSUES_AND_FUTURE_WORK.md](#) : A Comprehensive list of known bugs & issues and possible future work to be done on the platform.
- [REQ_SPEC_BOOM.xlsx](#) contains a list of decided upon requirements and specifications which the autonomous platform should follow. A complete bill of materials (hardware components) can be found in this file. It contains items, what purpose they serve, where they can be bought and a total estimated cost. As of *June 2023* the total cost is 16500 SEK to build the base platform and have spare parts over.

1.6.1 Generation 4 - changelog

Autonomous Platform Generation 4 was started by Fredrik Juthe and Erik Magnusson for their master's thesis during spring 2023. Since then Alexander and Seamus and continued working on it.



AP4 was meant as a fresh start for the autonomous platform project, removing the development issues that occurred with AP3 over time. AP4 was designed from the beginning to be a long time project with clear requirements and better development processes.

Another goal with AP4 is to make it more modular than previous versions. Previous versions had many proprietary and hardcoded solutions for a specific problems. These could later on affect the development process by making the previous implementations hard to follow.

A new concept which is trialed on AP4 is the concept of “Centralized” Electrical / Electronic (E/E) and software architecture. In short this means that the heavy computations are performed on a centralized computing unit and the ECUs are only responsible for relaying information to and from the computing unit.

It is built on an electric segway go-kart platform. Starting from scratch on a new go-kart.

1.6.2 Version 4, June 2023

AP4 was created by Fredrik Juthe and Erik Magnusson as part of a master thesis at Chalmers University of technology spring 2023. Supervisor: Hamid Ebadi



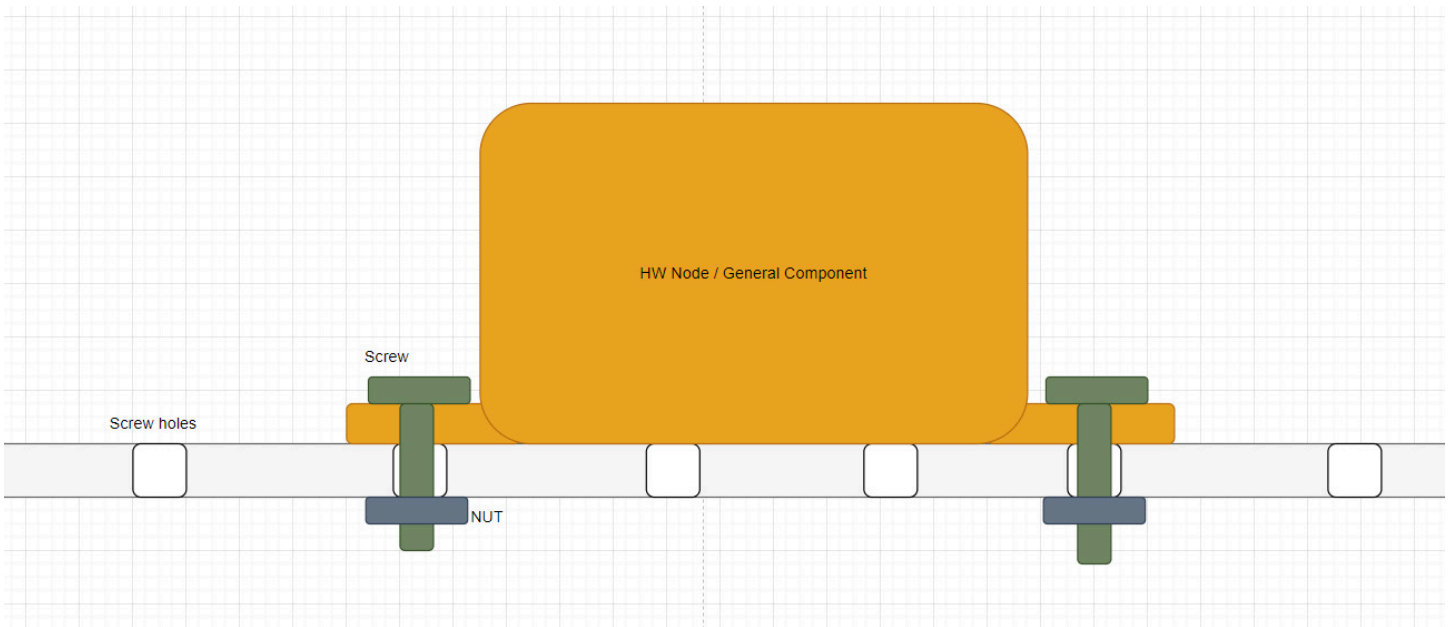
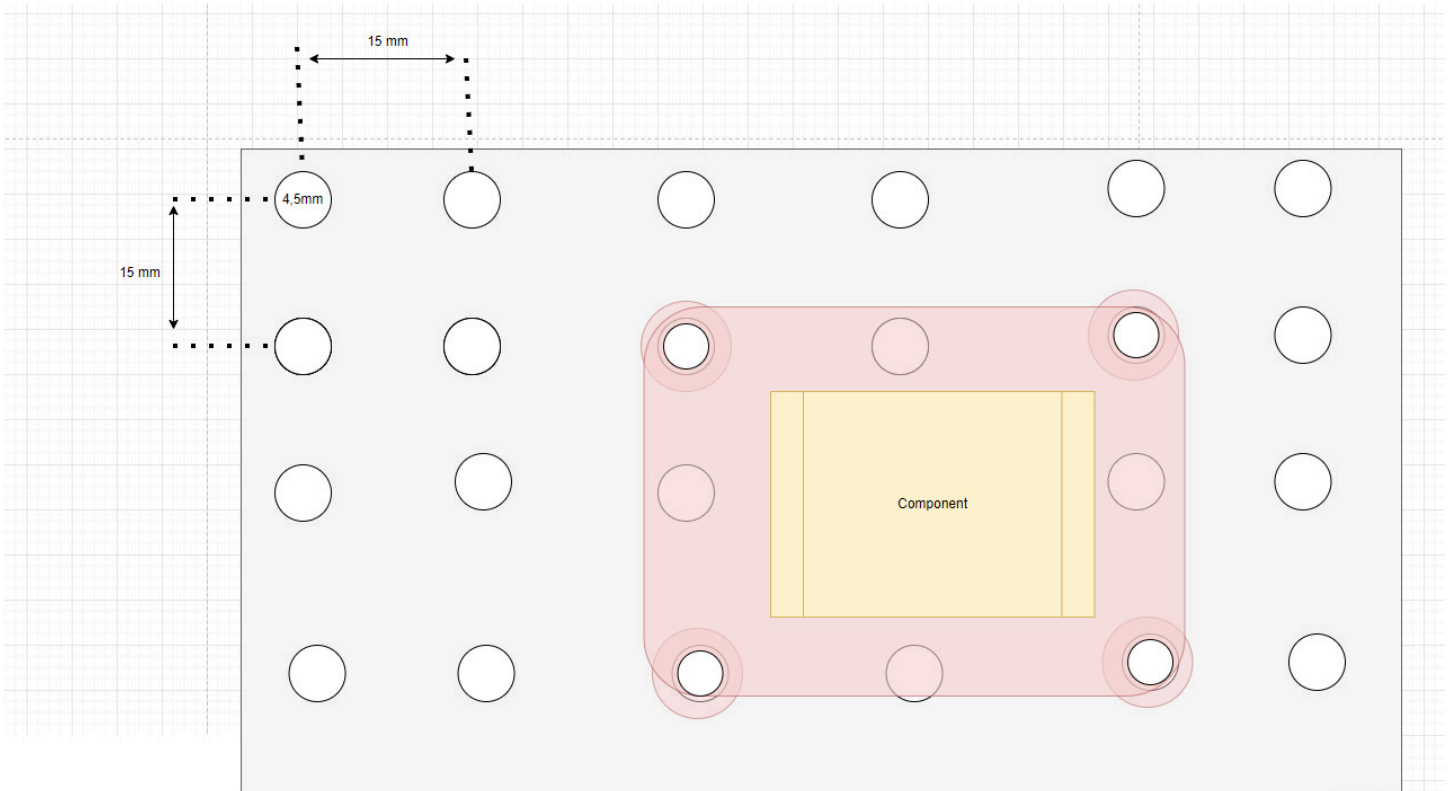
1.6.3 Version 4.1, July 2023

Improving documentation and adding speed sensor by Seamus Taylor, Alexander Rydevald. Supervisor: Hamid Ebadi



1.6.4 Version 4.1.1 August 2023

Improving documentation, Solving comments left in README files. Refactored repository, removed old/unused files, and made it more clean. by Erik Magnusson and Seamus Taylor. Supervisor: Hamid Ebadi



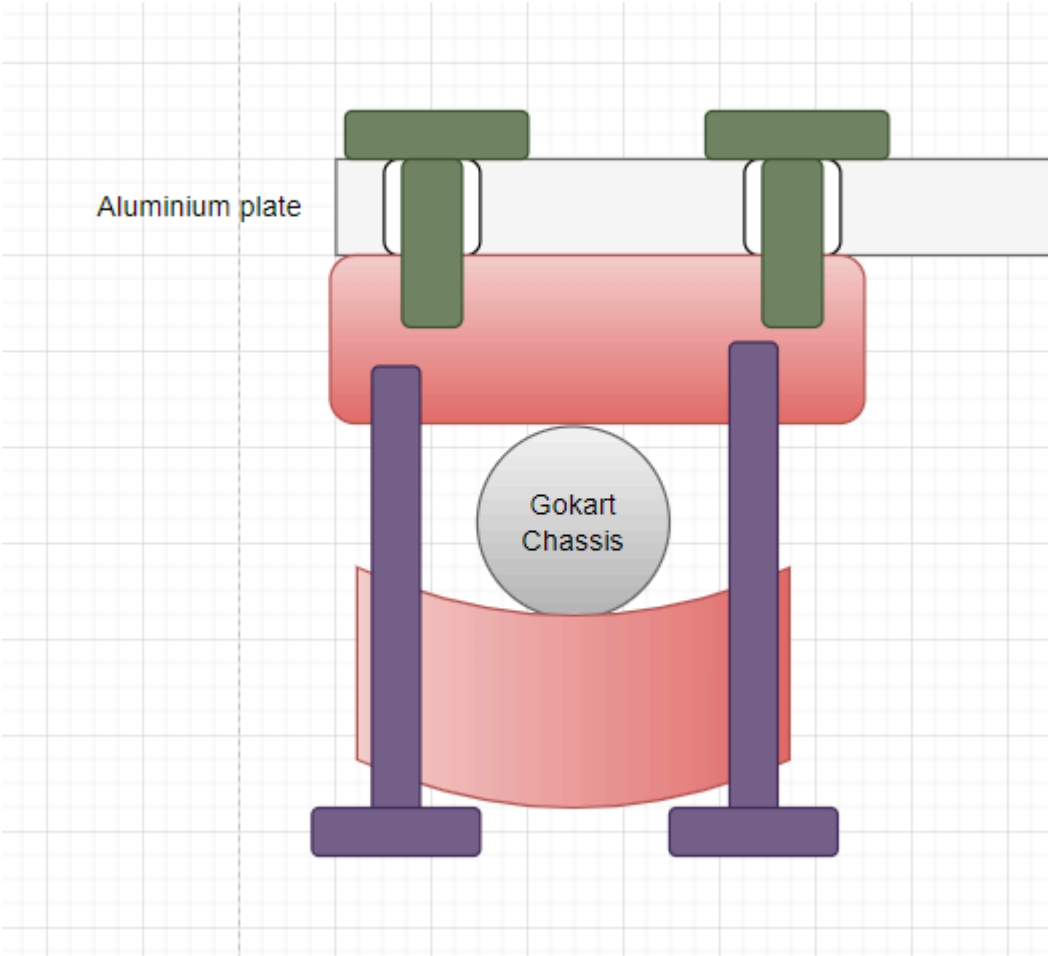
Specific solutions for this needs to be CADed later on.

1.7.3 Mounting the aluminum plates to the frame

One can either use the pre existing mounting holes on the gokart chassis and screw directly into the chassis.



Or one can create 3d printed fixtures to mount the plates, I.e:



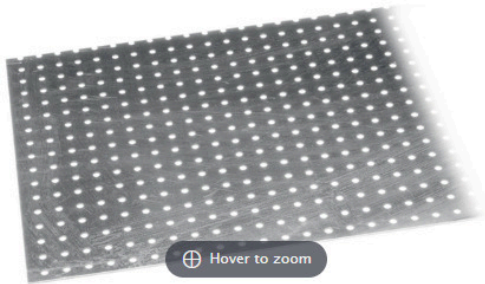
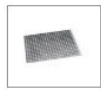
1.7.4 Component list

- Aluminum metal sheet with be fabricated holes: Aluminum Lochblech blank 250x500x1.5mm article nr: 148-21-919 OR 4001116378041 <https://www.elfa.se/en/aluminium-perforated-plate-round-holes-500x250x1-5mm-alfer-4001116378041/p/14821919> Holes are spaced 15 mm apart, hole diameter 4.5 mm

4001116378041 - Aluminium Perforated Plate (Round Holes), 500x250x1.5mm, Alfer

alfer®

Distrelec Article Number: [148-21-919](#) Manufacturer Part Number: [4001116378041](#) Brand: **Alfer**



13 In stock for immediate* dispatch

*Available for next business day delivery.

Additional stock will be available in 15 days

PRICE PER PIECE

SEK 512,50 (inc. VAT)

SEK 410,00 (exc. VAT)

1 + SEK 410,00

5 + SEK 372,00

10 + SEK 342,00

- 1 +

Add to Cart

1 min order

Image is for illustrative purposes only. Please refer to product description.

Shopping list

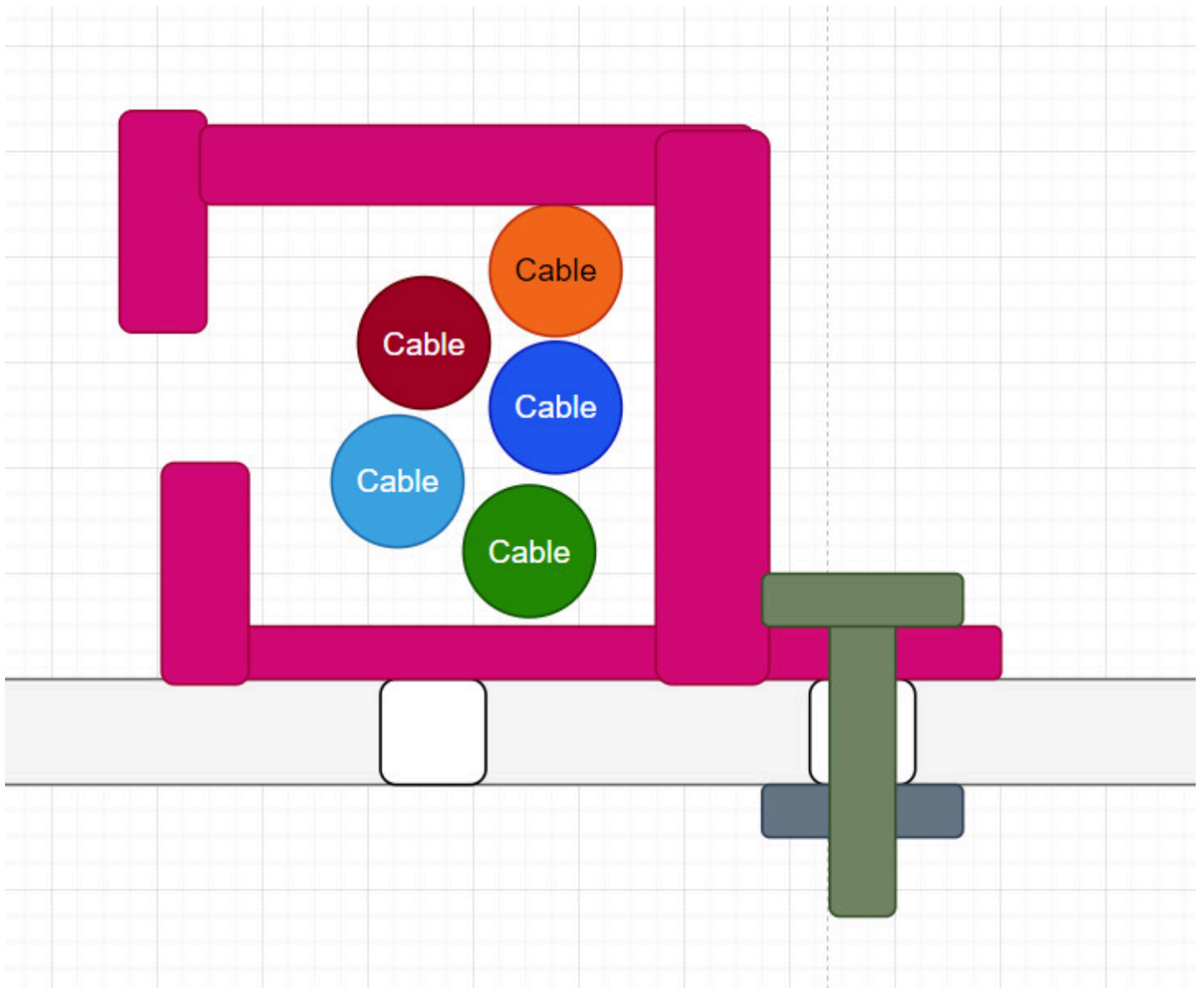
Compare

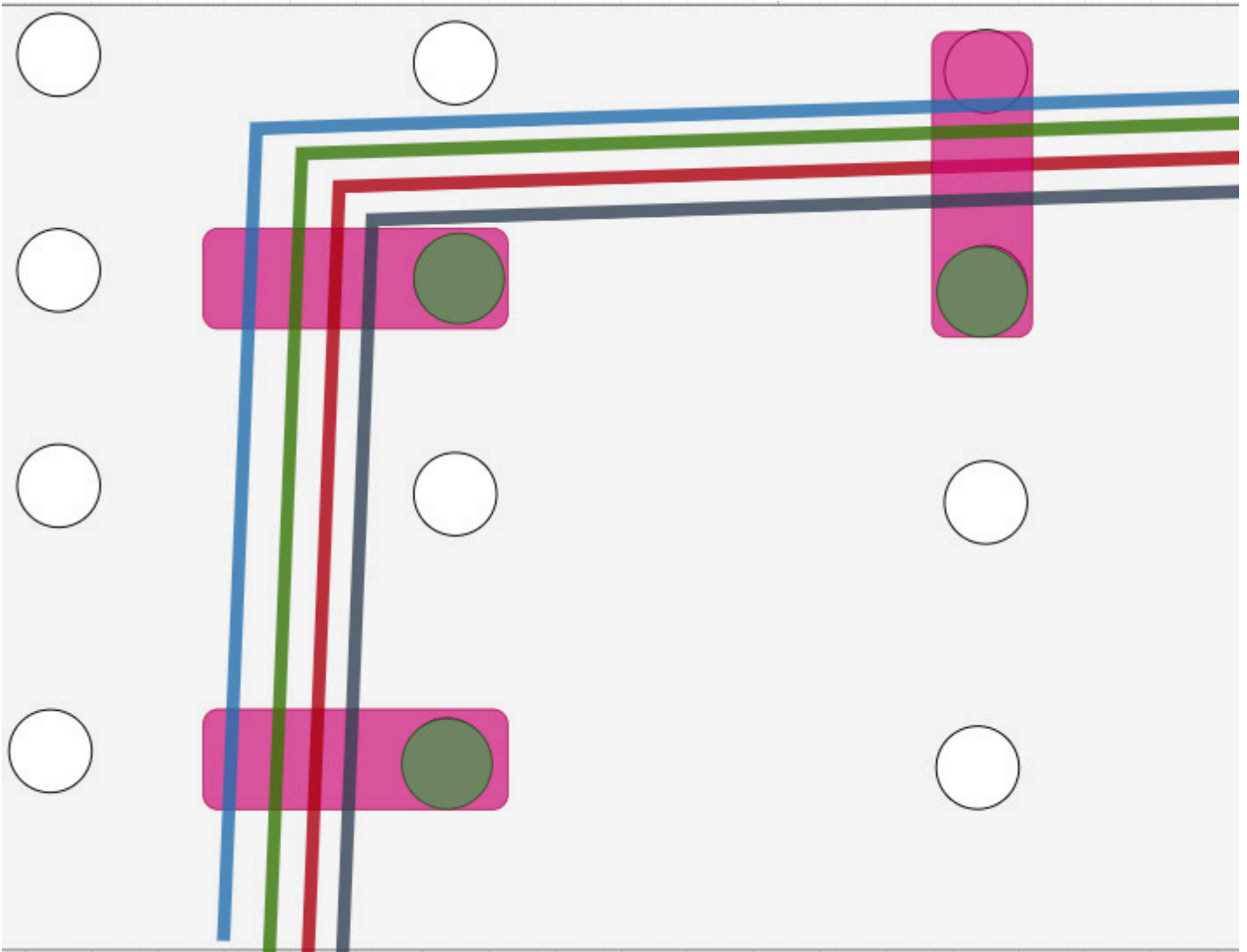
- M4 machine screws
- M4 nuts

One can either use pre existing cable routing hardware as in the image below



or can 3d print simple cable holders such as this image





1.8 Software Components

The procedure of installing some software will be mentioned in this section. You need:

- Preferably have a linux based computer if you want to run code.
- If you only want to read through the documentation, staying in the browser is sufficient.
- OR if you only want to download and look through the repository a Windows host computer is sufficient.

1.8.1 Information and Background

This README file is meant to collect information about all the software components used to construct Autonomous Platform Generation 4 or any tools used in this project. A lot of Software is used both in `Hardware_Interface_Low_Level_Computer` and `High_Level_Control_Computer`, and instead of explaining the same thing twice it can be collected here. It will also give new members of the project a structured document to read through.

This is meant to give new members a quick overview and explain the most important concepts related to the autonomous platform project.

The software and components that will be described in detail in this document are

- [VSCode IDE](#)
- [Linux Ubuntu 22.04](#)
- [Git](#)
- [Containerization](#)
- [Robot Operating System 2 \(ROS2\)](#)
- [PlatformIO & VSCode](#)
- [Jenkins](#)

1.8.2 IDE

The recommended Integrated Development Environment (IDE) is Visual Studio Code (VSCode). It is a free versatile IDE with many useful plugins. The built in file explorer makes navigation of the AP4 repository very easy.

See installation guide [here](#) for Windows / Linux.

Visual Studio Code supports the use of plugins, referred to as extensions. These can be installed simply inside VSCode by navigation to the extensions tab. (Tab with four small squares to the left)

The recommended list of extensions are:

- C/C++
- Python
- PlatformIO

1.8.3 Linux Ubuntu 22.04

The project software is meant to be run on Linux Ubuntu 22.04, therefore it is good to have basic understanding on how to use linux. (Good ability to google linux commands and copy paste commands is recommended!)

A linux for beginners guide related to development using Robot Operating System Framework can be found [here](#). It is an online interactive tutorial suitable for a project member which has NEVER worked in linux before. If you know how to navigate using the linux terminal and copy paste commands, skip this!

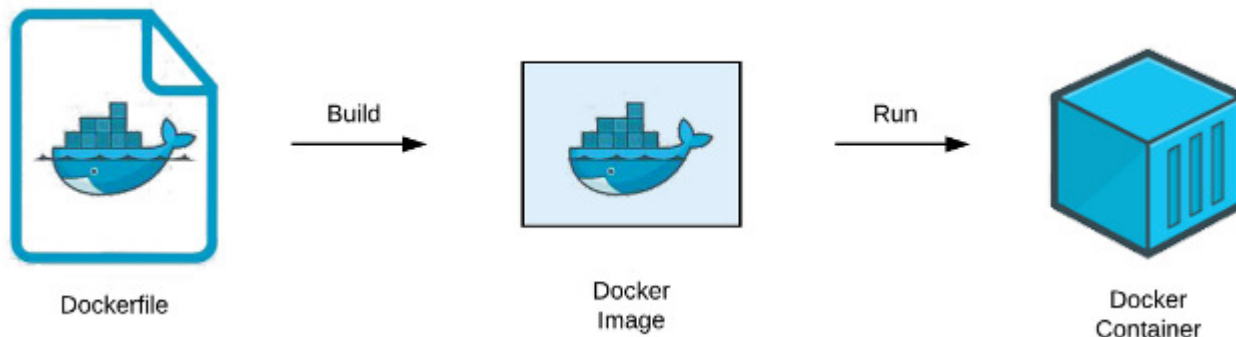
1.8.4 Git

Autonomous Platform Generation 4 uses the “MonoRepo” mantra. Everything related to AP4 is collected inside ONE repository called `autonomous_Platform_Generation_4`. If you manage to clone this onto a computer you have everything you need to get started.

Installing Git through command line is preferred. See the official guide [here](#) and choose appropriate operating system. Running the code has not been tested on Windows machines but it is still possible to read through the documentation and inspect files on a Windows device.

1.8.5 Containerization

The High level and low level software runs inside a docker containers (on the Development laptop and Raspberry Pi 4b). [Official Docker Page](#). It is similar to a Virtual Machine, in that it can run software in a separate compartment from the host computer software. It does not take up as much resources as a VM and is much faster to spin up (start). Docker containers uses the following components to create containers.



Dockerfile: [Dockerfile Official Documentation](#). A dockerfile is a text file describing the software environment which will be run in the container. For example, running Ubuntu as a base with with X,Y standard linux packages installed onto it. So every library or package one would install on a linux desktop version, one could define in this file. This dockerfile can then be built into a docker image. (everytime a container is rebuilt, anything saved in it is LOST). See `Hardware_Interface_Low_Level_Computer/dockerfile` for an example of a dockerfile. Therefore software should be written outside the container and passed into the container through volumes. See `Hardware_Interface_Low_Level_Computer/docker-compose` how this is performed by mounting volumes.

A docker image can then be spun up which creates a container.

A container can be spun up with several configurations commands. [List of commands](#). For example, it is possible to mount volumes (directories) into the container.

1.8.5.1 Installation of Docker See official guide [here](#) Choose appropriate operating system.

As of August 2023 AP4 containers have only been configured to run on a Linux Host computer.

1.8.6 Containers

The containers for autonomous platform contains two docker related files, **dockerfile** and **docker-compose**. The dockerfile describes the software environment (ubuntu 22.04 Jammy) with additional software and onto this the ROS2 framework and additional libraries are installed. The docker-compose file uses the dockerfile to launch a container with a set of parameters. For example, passing in and out graphical elements, and passing created software (as a directory) (performed by mounting a volume) into the container. In the docker-file a command which runs when starting the container can be defined. This can be useful to call a startup script that would run inside the container when starting it. This is done using the command: keyword.

Note: Docker can be run on Windows, but certain commands/parameters used on AP4 are linux specific. I.e passing graphics to and from the container. As of August 2023 it is therefore only possible to run the high level software on linux. See 'FUTURE_WORK.md' for instructions on how to possibly solve this issue.

The container, with the configurations, can be started using: (Terminal path has to be located in the directory where the dockerfile and docker-compose is located at)

```
docker-compose up
```

Note: The host PC needs to be configured to pass graphical elements to the high level computer container. (before starting the container)

```
xhost +local:*
```

It is possible to enter a running container environment using

```
docker exec -it <container-name> bash
```

Running containers can be listed using

```
docker container ps
```

A running container can be stopped by either 'Ctrl+C' in the terminal in which 'docker-compose up' was run. OR in a new terminal:

```
docker stop <container-name>
```

1.8.7 Robot Operating System 2 (ROS2)

Robot Operating System 2 (ROS2) is a framework / middleware developed to create very complex robotic software. The first version was released around 2007 and its successor, ROS2 was released around 2019. The version used for AP4 is ROS2-humble. [ROS2 Humble Docs](#). ROS2 Humble has an End of Life (EOL) of 2027. Future work should still use Humble as it has the most available packages. Different distributions of ROS are NOT compatible. In theory it is possible to bridge a ROS1 distribution with a ROS2 distribution using ROS bridge if a package does not exist for ROS2, this is not implemented on AP4. [ROS bridge link](#)

ROS2 has tier 1 support for Linux Ubuntu 22.04. And tier 3 support for Linux 20.04, on autonomous platform generation 4 Ubuntu 22.04 is used.

ROS2 applications can be written in either C++ or Python. There are tutorials for both versions. Different nodes (Computational applications) written in separate languages can still communicate with another using the ROS2 API & Framework. Therefore, use what programming language you are most comfortable with. In applications where execution speed is of utmost importance, using C++ can be preferred. But for testing new concepts or developing high level control algorithms Python is more than sufficient. Even if something is written in Python it still has to be compiled into a ROS2 package before it can be run.

1.8.7.1 ROS2 Useful commands quick reference A common methodology to debug software written for the ROS2 framework is to observe the behavior through a terminal window. Here is a list of useful ROS2 commands for quick reference in the future.

1.8.7.2 Commonly used commands to debug Source underlying ros2 environment variables

```
source /opt/ros/humble/setup.bash
```

Sourcing environment variables specific to a workspace (open terminal in workspace directory)

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

List what topics are currently being broadcasted to on the ROS2 network.

```
ros2 topic list
```

Read in terminal what is broadcasted onto a specific topic

```
ros2 topic echo ${topic_name}
```

Publish data onto a specific ros2 topic. If environment variables are sourced tab autocomplete works.

```
ros2 topic pub ${topic name} ${message type} ${message data}
```

List active nodes on ROS2 network

```
ros2 node list
```

Display useful information about a specific node.

```
ros2 node info ${node name}
```

1.8.7.3 Visualization Visualize the ROS2 network of nodes and topics in a GUI.

```
ros2 run rqt_graph rqt_graph
```

Plot data sent over different topics. Useful for I.e plotting requested velocity vs measured velocity to debug system and observe behavior. Any numerical data sent over a topic can be plotted.

```
ros2 run rqt_plot rqt_plot
```

1.8.7.4 Saving data sent on topics Save topic information transmitted over the ROS2 network. [Documentation here.](#) Record what is transmitted over a specific topic (Ctrl + C to stop recording):

```
ros2 bag record -o ${your bag file name} ${topic name 1} ${...} ${topic name n}
```

Display information regarding the bagfile that was previously recorded

```
ros2 bag info ${bag file name}
```

To record all information sent over all topics in the ROS2 network

```
ros2 bag record -o ${your bag file name} -a
```

The information recorded can be played back and will be outputted to the ROS2 network on the same topics that they were recorded on, using:

```
ros2 bag play ${bag file name}
```

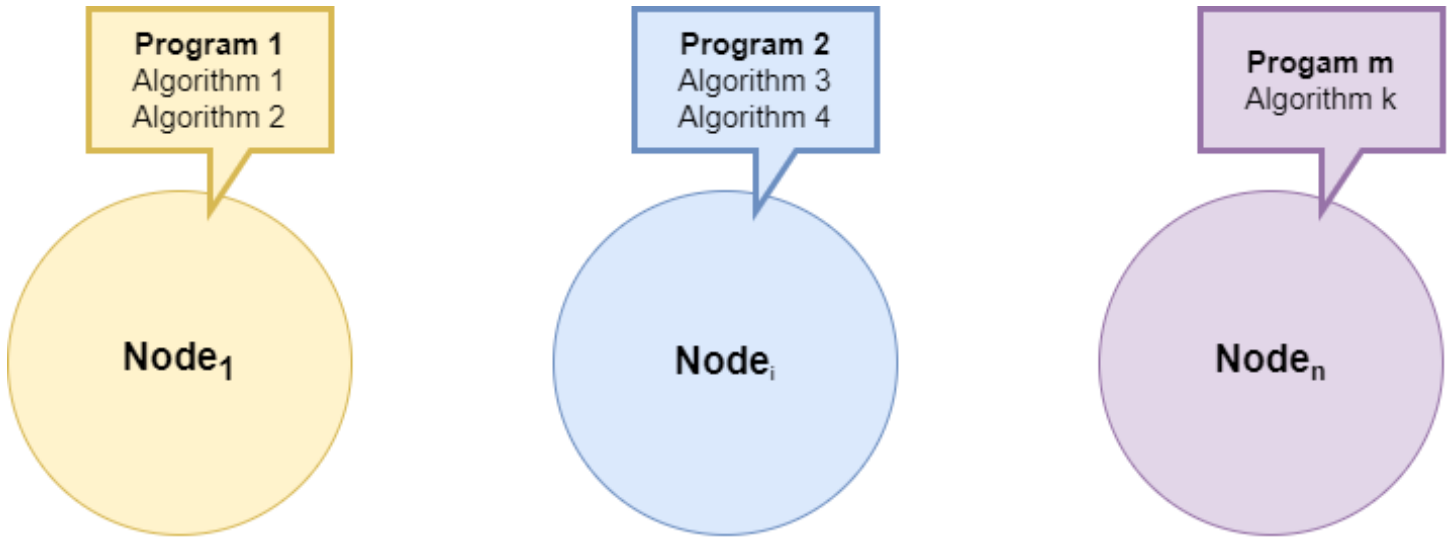
1.8.7.5 Available ROS2 Resources Here are resources on ROS2 which can be useful for future work. * Beginner level [Interactive ROS2 tutorial](#) with online excercies. VERY USEFUL

- Beginner Tutorial: [Writing a simple publisher and subscriber \(Python\)](#)
- Beginner Tutorial: [Writing a simple publisher and subscriber \(C++\)](#)
- [ROS_DOMAIN_ID](#)
- [ROS2 Interfaces](#) Fields and datatypes in ROS2 topics.
- [Official How-to Guides](#) For common problems when developing software in ROS2 framework.
- [ROS2 On Raspberry Pi](#) @TODO MOVE THIS POINT TO LOW LEVEL SOFTWARE
- [API Documentation](#)

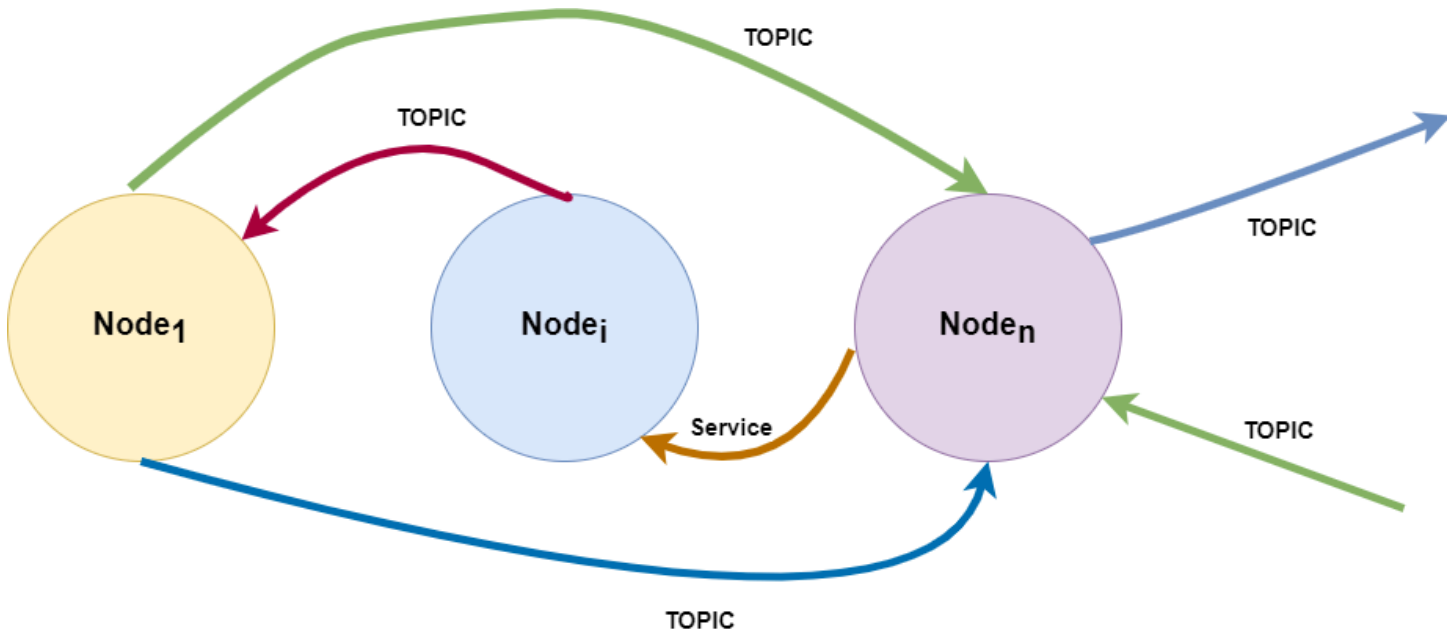
Here is a hands on video series on how ROS2 was used to create a mobile robot and a digital twin. It goes through the basics and then moves on to cover the advanced aspects of ROS2. [Link - Articulated Robotics](#) Many of these concepts are used on Autonomous Platform Generation 4.

1.8.7.6 ROS2 Concepts There are a few concepts which are very important to understand in order to understand how the high-level software is constructed on AP4. The official ROS2 documentation is a useful tool [ROS2 Beginner: CLI tools](#).

1.8.7.7 Node [ROS2 Nodes official Documentation](#)



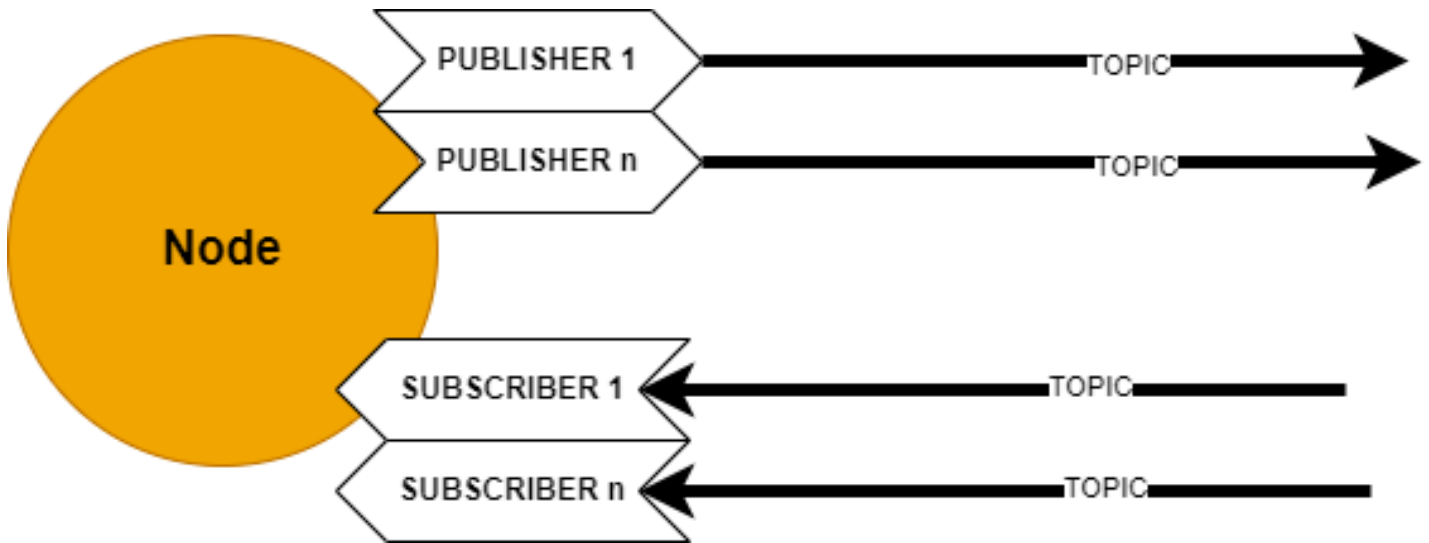
Nodes are where the software applications are written. Each node can be seen as a program (written in C++ or Python) performing some task. Whether it be processing information or interfacing with some hardware/software component. A node can perform multiple tasks and supports parallelization if configured correctly. [See Execution Management](#)



Nodes can communicate with other nodes using Topics, Services or Actions. This means sending information between them using a standardized communication interface. It is up for the developer to structure how information should be received and sent from nodes. This is done in the nodes themselves.

The communication between different ROS2 components on a network uses [DDS Middleware](#).

1.8.7.8 Topic [ROS2 Topic Official Documentation](#)



Topics are the means in which different Nodes can send and receive specific information. Information can be broadcasted using publishers and listen to using subscribers. These communication channels are called topics, each topic contains a topic name and a data field. The topic name determines what channel it will be sent over and the data field contains the information itself.

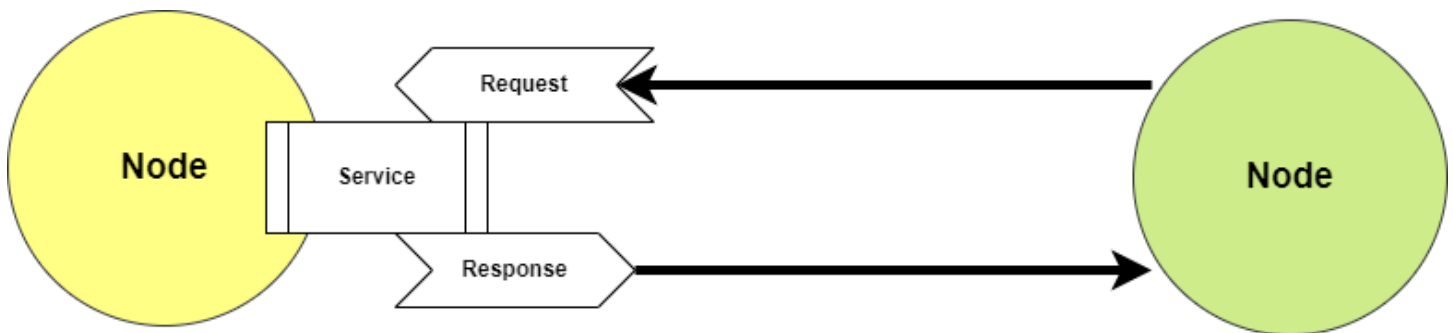
[List of generic built in topic data types](#). Custom data field types, containing multiple values sent in a single message can be created by a developer. [Link how-to](#)

Topic name
data

Topics are broadcasted over the whole ROS2 network by publishers, meaning any other Node on the network can listen to any publisher. In order to direct data to a specific node, a developer must create a subscriber in the desired node, which listens to the correct topic name.

Note: There exists multiple [Quality of Service](#) parameters which can be configured. These can tell the DDS middleware what topics should be prioritized when sending over the ROS2 framework. As of June 2023 these are set as standard priority.

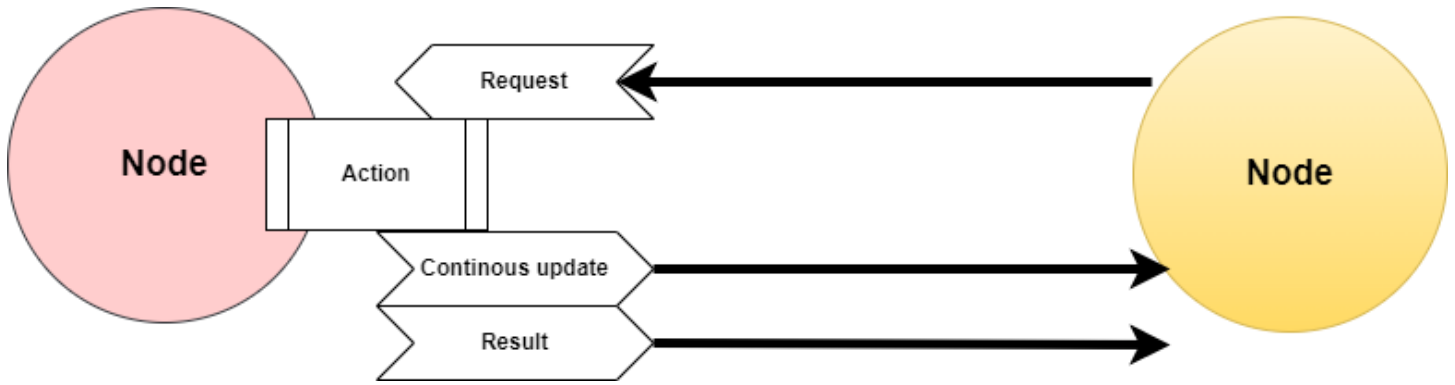
1.8.7.9 Services [ROS2 Services Official Documentation](#)



Services work much like the topics, but there is a key difference. a node can request information (structured as a topic) from one node, and is guaranteed to receive information back.

Request functionality has to be created by the developer and is not something that is automatically built into a node. See tutorial [C++](#), [Python](#)

1.8.7.10 Actions [ROS2 Actions Official Documentation](#)



1.8.7.11 Launch files A launch file can be run in order to start several nodes at the same time. Launch files from existing packages can be called from within a launch file.

[ROS2 Documentation: Creating a launch file.](#)

[Call launch files from inside launch files](#)

1.8.7.12 Creating a ROS2 software package In ROS2 a package is a form of library with functionality. Software inside a package is related to one function, therefore ROS code is very decentralized. This is very useful since existing packages can started separately and mixed together depending on application needs. Existing software packages can run parallel with custom user developed plugins to create new functionality. New packages can also be built on top of existing packages by defining dependencies.

A package can contain code for several nodes. So it does NOT have to be one node per package.

See a list of available ROS2 (version = Humble) packages [here](#). Make sure filter is set to 'Humble'.

1.8.7.13 Structure of a ROS2 package The file structure of a ROS2 workspace is standardized and can look something like like this: (don't worry most of the files in a package are automatically generated). All created packages are stored in the 'src' directory in the ROS2 workspace folder. Packages written in Python and C++ differ somewhat in content. Below is illustrated how to packages, one written in C++ and one written in Python are located in the filesystem.

```

└─ ros_ws
  └─ src
    └─ example_pkg_1 (C++)
      └─ launch (Optional)
      └─ src
        └─ <CODE>
      └─ CMakeLists.txt
      └─ package.xml

    └─ example_pkg_2 (Python)
      └─ launch (Optional)
      └─ resource
      └─ test
      └─ example_pkg_2
        └─ __init__.py
          └─ <CODE>
      └─ package.xml
      └─ setup.cfg
      └─ setup.py
  
```

1.8.7.14 Creating a ROS2 package The official ROS2 guide can be found [here](#). The steps are summarized below. Packages can EITHER use CMake or Python depending on what programming language you want to use. You cannot mix programming languages within a package.

In order to create a package you need to be located inside a ROS2 workspace. For autonomous platform 4 this has already been setup. For curious members the documentation on how to create a ROS2 workspace can be found [here](#).

1.8.8 PlatformIO & VSCode

PlatformIO is a Visual Studio Code (VSCode) extension.

In order to program the STM32F103C8T6 (Bluepill) microcontroller used for the embedded software a software driver is needed.

Windows:

STSW-LINK009. ST-LINK, ST-LINK/V2, ST-LINK/V2-1, STLINK-V3 USB driver signed for Windows7, Windows8, Windows10

Driver can be downloaded [here](#). It enables a windows computer to use the ST-Link v3 programmer which is used to program the bluepill microcontroller boards.

1.9 Project standards and conventions

1.9.1 Git commit messages

The project is using [conventional commits](#) as a commit standard meaning the commit messages are structured using the following standard:

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Types can be found under the summary section on <https://www.conventionalcommits.org>.

Each section should be separated with a blank line.

The footer **SHALL** contain the Azure task ID using the following format **Azure-ID: xxxx**.

With conventional commits, a version can automatically be determined and assigned based on the “type” of commit made. The version follows the [Semantic Version \(SemVer\)](#) specification.

1.9.2 Merge commits

When merging locally the commit message will be declined by the commit-msg git hook. This can be overwritten by using the `--no-verify` option. In PyCharm this can be found by: 1. Navigating to the **Git** menu 2. Clicking on **Merge** 3. Clicking on the **Modify options** dropdown and selecting the `--no-verify` option

1.9.3 Documentation

Documentation is done in a markdown file. Try to keep our documentation close to your code. Keep the documentation short and clear.

- `#` : Only used once for the title of the project
- `##` : usually once in each markdown file and usually share semantic with the markdown filename.
- `###` : Different topics, try to split your documentation into at least 4 topics
- `####` : sub topics. Try to avoid when you can use simple paragraphs

1.10 Automated Testing (CI/CD)

This document aims to describe how various software components on AP4 could be tested in Jenkins. As of August 2023 no tests are implemented, instead this document collects useful links and resources that might lead to future automated software tests.

There are three layers of software that needs to be tested:

- Embedded Software in ECUs : Uses PlatformIO environment / framework
- hardware interface and low level software : Uses ROS2 framework
- high level control software and digital twin : Uses ROS2 framework

1.10.1 Resources for automated testing of ROS2 software - Jenkins

- [Jenkins Basic for Robotics](#) Online course on how to test ROS software using Jenkins
- [How to setup the Jenkins master](#) for ROS (note outdated version of ROS but should exist similar for newer version)

1.10.2 Resources for automated testing of embedded software platformio with Jenkins

A simple hardware test platform would need to be constructed and connected to the jenkins server as i understand it.

- [Callet91/DEMO_Jenkins_PlatformIO](#) online github repository with step by step guide on how to setup platformIO and jenkins

1.10.3 Offline local PlatformIO software tests

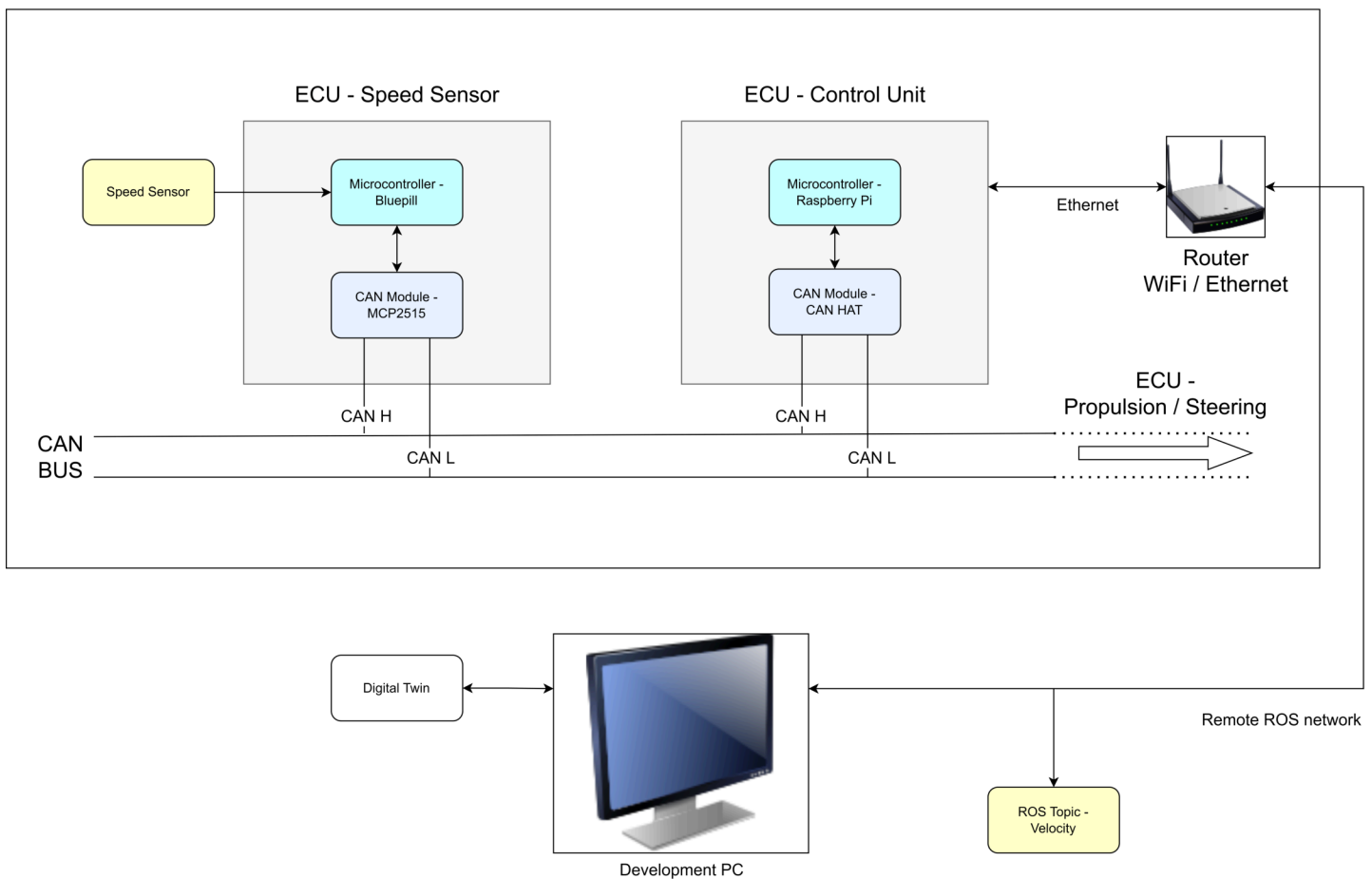
PlatformIO developed code could be tested offline on a local host computer connected to the micro-controller.

- [Unit Testing of a "Blink" Project](#) - Needs to be connected to hardware card?
- [Unit Testing](#) PlatformIO official guide

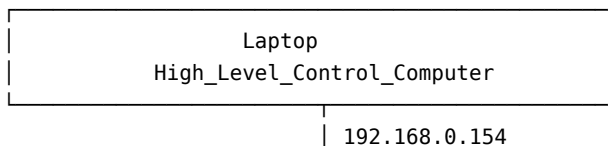
1.11 Component Communication

The image below illustrate how a speed sensor sends data to High_Level_Control_Computer and further to development PC.

Autonomous Platform (Gokart)



The diagram below shows different types of communication channels and IP address for different components.



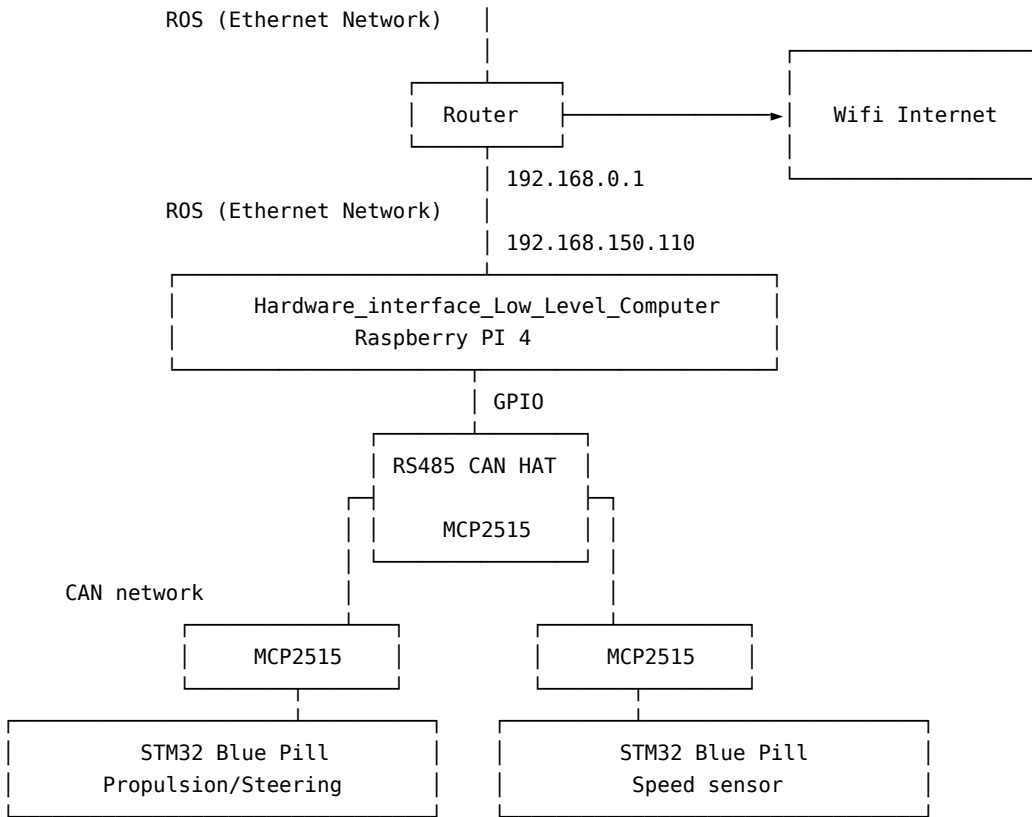


Illustration done using asciiflow online editor. (If any future changes has to be done)

1.12 Onboarding Procedure

This document is meant to structure the introduction of Autonomous Platform Generation 4 (AP4) to new members. If you feel something was unclear after you read through this, please add to this document to make it easier for future team members!

The goal of this document is for new team members to have a basic understanding of what Autonomous Platform Generation 4 is and how get working within a reasonable amount of time.

It will describe what prerequisites are required to follow along, how to read through the provided documentation and how to get going continuing the development of the autonomous platform project.

This document does not specify any development tasks - It is up to the supervisor and you to define the goal for the given time period.

1.12.1 Schedule

Here follows a quick step by step procedure on how to get familiar with the project. You DO NOT need to understand everything, the most important thing is that you get a good overview of the project and get an understanding of where YOU can apply your experience and knowledge to contribute.

Remember, you have a supervisor, be sure to ask for clarifications whenever something is unclear.

These should be the goals for a new team member. These are simply a guideline and are not set in stone. Talk with your supervisor what parts may be relevant for you or what time plan is suitable.

1.12.1.1 Week 1

- Understand the purpose of the Autonomous Platform project
- Have a feeling of where certain parts of the project are located at within the repository
- Cloned the repository to your work computer
- Started up the physical AP4 platform by following the startup procedure documentation.
- Read through the major documentation files - Presented in “Where can I find What” Don’t expect do understand everything that is mentioned.
- Install needed software (Git, Docker, VSCode, PlatformIO)

1.12.1.2 Week 2

- Start playing around with existing software (preferably on a new git branch to not brake existing functionality). Try to start different components and see if you can get expected output.
- Try and read sensor information from platform and display in terminal. I.e steering angle. Does it change when the wheels are moved?
- Try to control the actuators through command line. Can you set the steering angle?
- Find something that is lacking within the documentation and can be improved upon
- **Decide upon a project which you can perform within a reasonable time frame - Talk with your supervisor what may be suitable for you.**
- Lay out a plan on how to make your project possible. Do you know where in the code to make changes?
- Start internal educations if applicable. (CAN)
- Start external educations if applicable. (Git, Docker, ROS, Linux, etc)

1.12.1.3 Week 3

- Work on your proposed project - Keep your supervisor in the loop
- Continue internal educations if applicable. (CAN, Embedded Development, etc)
- Continue external educations if applicable. (Git, Docker, ROS, Linux, etc)

1.12.1.4 Week 4

- Finished internal educations if applicable. (CAN, Embedded Development, etc)
- Finished external educations if applicable. (Git, Docker, ROS, Linux, etc)
- Work on your proposed project - Keep your supervisor in the loop
- End of week 4: Present the results of your small scale project to some Infotiv members.

1.13 Useful skills/software knowledge

You do not need to be familiar with every software or concept mentioned below.

- Linux Basics and be able to flash OS images to SD cards & USB drives
- Docker containerization basics
- Infotiv AB CAN internal bus education
- Robot Operating System 2 (ROS2) introduction
- Embedded Software Development Introduction

An unpublished copy of the Master's Thesis report is available to read in the repository.

1.14 Extend AP

The process of extending the three specific software components are described in detail in the following documents:

- `CAN_Nodes_Microcontroller_Code/HOW_TO_EXTEND.md`
- `Hardware_Interface_Low_Level_Computer/HOW_TO_EXTEND.md`
- `High_Level_Control_Computer/HOW_TO_EXTEND.md`

1.14.1 Design Principles

Follow KISS principles:

- “keep it super simple”
- “keep it short and simple”
- “keep it simple and straightforward”
- “keep it small and simple”
- “keep it stupidly simple”

Make subsystems modular, plug-and-play, and independent of each other. According to Murphy's Law, "Anything That Can Go Wrong, Will Go Wrong", so in other word, things will go wrong in any given situation, if you give them a chance. Make modules that can be replaced and keep each part small and modular.

1.14.2 Software

- Emphasize code modularity by utilizing Docker containers.
- Instead of writing installation documents, create a build script (e.g., Makefile) for streamlined setup.

- Include at least one unit test for each module to ensure functionality and reliability.
- Create test scripts to address specific issues and automate the debugging process. For example:
 - For wireless communication issues, implement a bash script that performs ping tests with other nodes.
 - Automate the debugging of CAN communication by utilizing candump in a simple bash script.
- Integrate these tests into your system, running them before executing components for comprehensive validation.
- Ensure well-written and clean code, prioritizing meaningful variable and function names over excessive documentation.
- Be mindful of the side effects of premature optimization and consider them while making performance improvements.

1.14.3 Hardware

- Utilize standard power sockets and ensure compatibility with standard CAN and power cables/connectors.
- Adhere to standard color coding for wires to ensure consistency and ease of identification.
- Conceal all wires and protect them from accidental disconnection. (assume a kids going to use the gokart)
- Avoid hard wiring between different physical modules/nodes. - Use detachable cables for connections between modules.
- Design all components to be easily opened, inspected, and replaced when necessary.
- Ensure quick and efficient component replacement, similar to the swift tire changes performed during a Formula 1 racing car pit stop. Don't do a Ferrari and make your pit stop more difficult than necessary.

1.14.4 Documentation:

- Begin by documenting the “what” and “why” aspects of the project or concept. The focus should be on understanding the purpose and reasoning behind it and “How” is usually not important.
- Begin by documenting the high-level idea or overview before delving into low-level details
- Use markdown format. It is faster, easier and convertible to other more complex format such as LaTeX, HTML and PDF.

1.14.5 Prerequisite Software & Skills

In order to understand the implemented architecture, software and hardware here are some general useful resources:

- Linux Basics and be able to flash OS images to SD cards & USB drives
- Docker containerization basics
- Infotiv AB CAN internal bus education
- Robot Operating System 2 (ROS2) introduction
- Embedded Software Development Introduction

1.14.6 Adding Wheel Speed Sensor

This is an example procedure on speed sensors should be added onto the platform.

This is an overview of how Fredrik and Erik would have done it but you are free to make your own decision if you follow the principles.

There are supposed to be four wheel speed sensors on the platform. The front wheel speed sensors are mounted but not wired. The back wheel speed sensors need to be 3d printed again. The wheel speed sensor models can be found in CAD\Speed_Sensor_Back\, import stl files into Ideamaker 3d printing slicing software, slice model to G-code and print!

The wheel speed sensor hardware, is located in the see-through plastic bin on the platform.

[link to wheel speed sensor hardware](#)

The project would be split up in smaller parts (we would start with low level)

0. Set up repository and SW applications
1. Create Embedded-SW close to hardware 1.1 Create test rig
2. Get hardware sensor working
3. Wire up new ECU base
4. Adjust CAN database & Generate new CAN_DB.c/h files
5. Adjust Low-level Hardware interfacing ROS2 code 5.1 Adjust can_msgs_to_ros2_topic_pkg package 5.2 Adjust vehicle controller

So in detail:

- Install docker
- Clone repository
- Create new personal gitlab branch to work in, Only working code should be added to the main branch.

- Learning how to use PlatformIO and st-link v2 to flash Bluepill microcontroller in VSCode editor
- (look at Infotiv sharepoint internal educations Edu: Get your first LED blinking on STM32)
- (look in `CAN_Nodes_Microcontroller_Code\HW_NODE_CODE_TEMPLATE\src`)
- Make sure things compile properly and can be uploaded to bluepill microcontroller
- Researching how the speed sensor works
- Create a small local solution test rig, (breadboard and jumper wires, Bluepill and speed sensors)
- Wire Components, Speedsensor
- Some work has been done on this already see the following code (Note!! not up to date GPIO-pin wise with the standard ECU wiring) `CAN_Nodes_Microcontroller_Code\HW_NODE_DEV_Speed_Sensor_Lib\src` `CAN_Nodes_Microcontroller_Code\Shared_HW`
- Read the velocity using bluepill board and print result over serial such that one can observe behavior
- Investigate how ECU base is wired (see existing code / documentation)
- CAD files for ECU base are located at `\CAD\Node Box`
- Wire a ECU base, Bluepill, MCP2515, Logic-level-converter, DC-DC converters, ...
- Connect ECU to CAN network, send some dummy data
- understand how to setup linux socket can interface - look into how it is automatically started on raspberry pi
- verify that dummy data shows up on linux cansocket candump
- Adjust CAN database `CAN_DB.dbc` (dbc file) using kvasar located at `\CAN_Nodes_Microcontroller_Code\CAN_LIBRARY_DATABASE`
- Add new frame(s) for speed measurements
- Generate new C code, `CAN_DB.c` and `CAN_DB.h`, see documentation for procedure.
- Compare new vs old `CAN_DB.c/.h` to see that the changes have been made
- Use the Template code for a ECU and add the speed-sensor-library functionalities Make a copy of `\CAN_Nodes_Microcontroller_Co`
- Create a library in `CAN_Nodes_Microcontroller_Code` for the speed measuring ECU
- Transfer code from test-rig into new created library
- Instead of printing to serial, send velocity over CAN bus
 - See how throttle voltage / steering angle was sent over CAN bus in SPCU library
- Verify that velocity measurements show up on linux socketcan candump
- Append newly created measurement CAN frame (&signals) into the ROS2 `can_msgs_to_ros2_topic_pkg` source code. Located at `Hardware_Interface_Low_Level_Computer\ap4_hwi_code\ap4hwi_ws\src\can_msgs_to_ros2_topic_pkg\src` create publishers, subscribers, callback functions, add new switch cases
- rebuild `can_msgs_to_ros2_topic_pkg` package (colcon build)
- Start the can translator node from `can_msgs_to_ros2_topic_pkg`
- (It is automatically started if restarting the raspberry pi docker container) or run ROS2 launch file manually
- Use speed measurements for feedback control of velocity in vehicle controller node.
- (Make a copy of `Hardware_Interface_Low_Level_Computer\ap4_hwi_code\ap4hwi_ws\src\ xbox_controller_feed_forward_ctrl_pkg`)
- Rename package and add feedback PID/PIDf control theory logic
- Adjust launch file to start the new vehicle controller when docker container is spun up See how is currently implemented `Hardware_Interface_Low_Level_Computer\ap4_hwi_code\ap4hwi_ws\src\launch_hwi_software_pkg\launch`

1.14.6.1 Get familiar with the sensor (blink and print) A good first step is to get familiar with the sensor and understand how to read measurements from it. (Some of these steps could be skipped if experienced working with VScode and programming microcontroller)

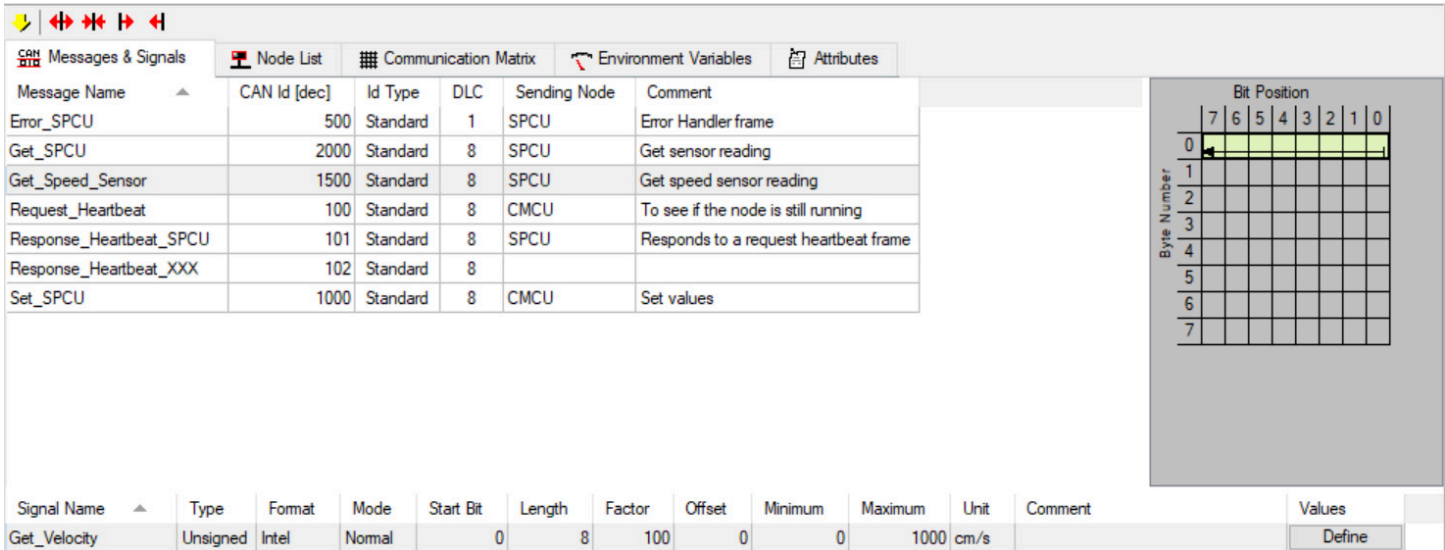
1. Clone the AP4 repository and create a new personal gitlab branch to work in.

NOTE: Do not work in main branch until you know things work.

2. Download VSCode editor, install and learn how to use the Platform IO extension.
3. Make a simple blink script using microcontroller (bluepill, Arduino etc.) connected to a ST-link v2 (used for programming microcontroller). Necessary downloadable software for the ST-link v2 found [here](#). If unsure how to create blink script, refer to the Infotiv sharepoint education: [Get your first LED blinking on STM32](#). Make sure things compile properly and can be uploaded to the microcontroller.
4. Research how the sensor works (pin ports, operating voltage etc.).
5. Create a small local solution test rig and wire components (breadboard, jumper wires, microcontroller, ST-Link v2, USB-mini and sensor).
6. Read the sensor measurement using the test-rig board and print the result over the serial monitor in VSCode.

1.14.6.2 Adding a new CAN frame It is recommended to have completed the [internal Infotiv CAN education course](#) beforehand.

1. Adjust the existing CAN_DB.dbc (dbc file) located under /CAN_Nodes_Microcontroller_Code/CAN_LIBRARY_DATABASE. It is recommended to use KVASER database editor for this, as it provides a simple interface for adding frames and signals. Download link to Kvaser database editor found [here](#). A video explaining how to use it found [here](#). More specific information related to AP4 found in README.md under [CAN_Nodes_Microcontroller_Code](#).
2. Add new frame(s) and signal(s) for sensor measurements similar to figure below.



NOTE: Use Linux PC for next step (step 3). Push updated CAN_DB.dbc file to git repo. and git pull main branch on Linux PC.

1. Generate new C code: CAN_DB.c and CAN_DB.h, documentation how this is done found under [CAN_LIBRARY_DATABASE](#)
2. Compare new vs old CAN_DB.c/.h to see that the changes have been made.

NOTE: Push changes to git repo and git pull main branch on a Windows PC.

1.14.6.3 Code Sensor to CAN message

1. Use the ECU template code found under [CAN_Nodes_Microcontroller_Code/HW_NODE_CODE_TEMPLATE](#). Check the comments inside the source code and get an overall understanding of the different steps.
2. Extend the code with CAN message decode/encode and sensor functionality.

TIP: Check previous ECU codes (SPCU and Speed_sensor ECU) located under [CAN_Nodes_Microcontroller_Code](#) to see which additions need to be made.

In short, what needs to be added to the ECU template are: - Inclusion of any additional library. - Specify global variables. - Specify configurations such as input/output pin ports and baud rate. - `can_interface.AddPeriodicCanSend(CAN_ID,time)` inside `setup()` function. - Sensor algorithm, decode and encode CAN frame inside `loop()` function. The format of the decode and encode messages can be found inside the generated CAN_DB.h file, example from another sensor can be seen below.

```
int decode_can_0x5dc_Get_Velocity(const can_obj_can_db_h_t *o, double *out);
int encode_can_0x5dc_Get_Velocity(can_obj_can_db_h_t *o, double in);
```

1. Make sure the code is buildable and upload the code to the microcontroller.

1.14.6.4 Build an ECU

1. Extended the test-rig built in *Get familiar with your sensor* and add a CAN module.

NOTE: Look through the specifications of the components to understand each pin port, operating voltage and how to connect everything.

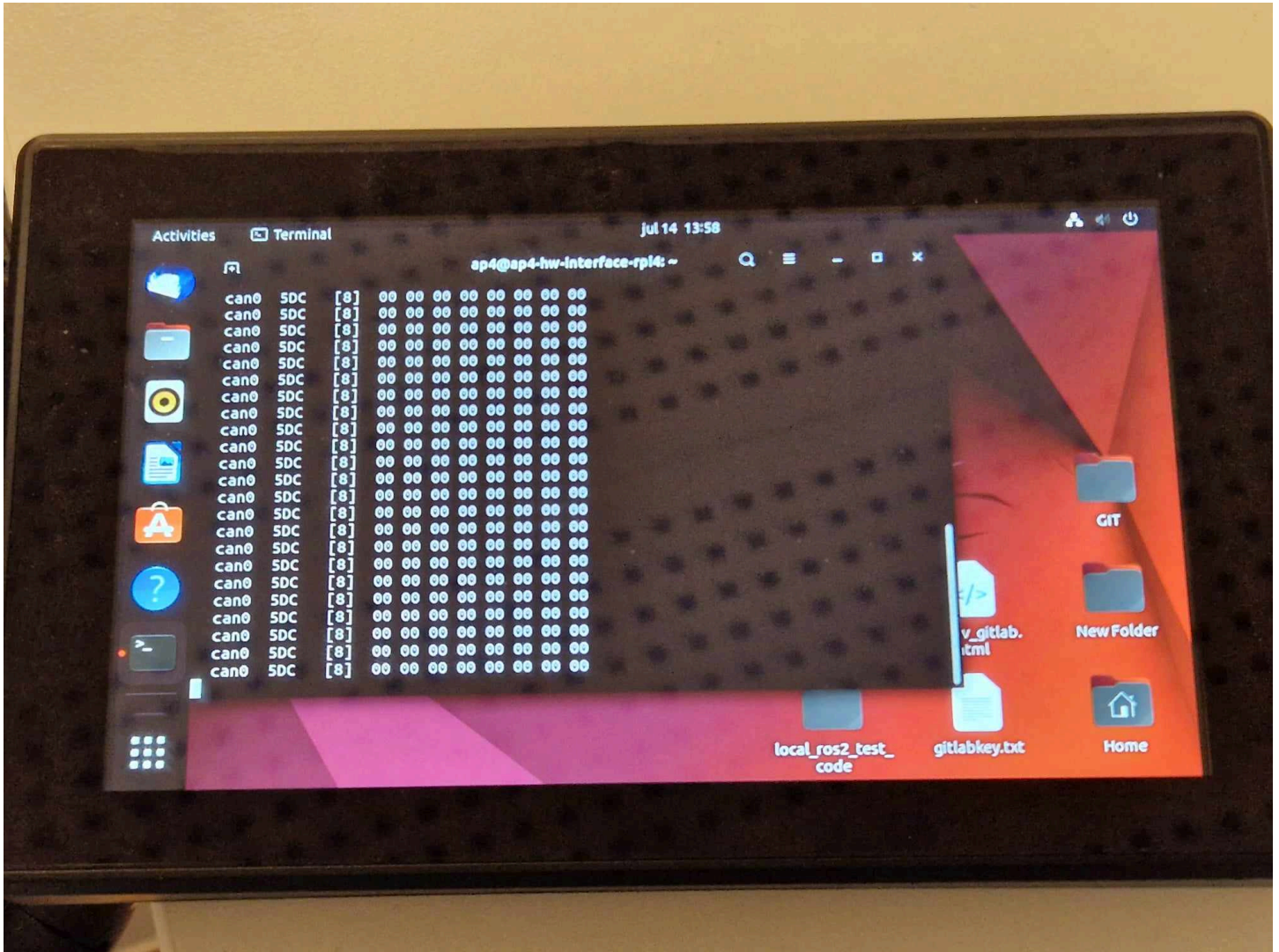
2. Connect a DB9 connector between the CAN module test-rig and the Raspberry Pi mounted on AP4. Update the files on the Raspberry Pi by powering on AP4 using the flip switch, navigate on the display using the wireless mouse and keyboard (labeled Infotiv 3) and git pull the repo.

1.14.6.5 Raspberry Pi 4b 4gb

3. Verify that sent CAN messages from the microcontroller show up on the Linux cansocket candump. On the Raspberry Pi this is done by opening a new terminal and running the command:

```
candump can0
```

Below you can see an output example from candump. It shows which channel the message is being transmitted on (can0), the ID (5DC), the frame size [8 bits] and the data bits (00 00 ...).



1.14.6.6 CAN to ROS communication

1. Append newly created measurement CAN frame (&signals) into the ROS2 [can_msgs_to_ros_2_topic_pkg](#) source code. This procedure is pretty straight forward and a detailed description can be found under [Hardware_Interface_Low_Level_Computer](#).
2. Git push the changes and git pull repo on the development laptop (Linux).

1.14.6.7 Development laptop

1. Rebuild `can_msgs_to_ros2_topic_pkg` package (colcon build). The procedure is described under [Hardware_Interface_Low_Level_Computer](#).

NOTE: For the next step it is important that both the Raspberry Pi and the Development laptop are connected to the same wifi/ethernet. This can be double checked using a simple “Ping test”, which tests their communication.

Restart the Raspberry Pi and Development laptop and use the following commands in a new terminal on respective device:

1.14.6.8 On the Raspberry Pi

- Enter the existing docker container

```
docker exec -it ap4hwi bash
```

- Source variables

```
source /opt/ros/humble/setup.bash
```

- Start the CAN translator node

```
ros2 launch ros2_socketcan socket_can_bridge.launch.xml
```

1.14.6.9 On the Development laptop

- In the terminal, navigate to /High_level_Control_Computer
- Build the docker container

```
docker-compose build
```

- Disable the access control (necessary in order to start the container)

```
xhost +
```

- Start the docker container

```
docker-compose up
```

The Gazebo environment should start up, ignore this for now.

- Double check that the docker container is up and running

```
docker ps
```

- Open a new terminal window and enter the docker container

```
docker exec -it ap4hlc bash
```

- Source variables

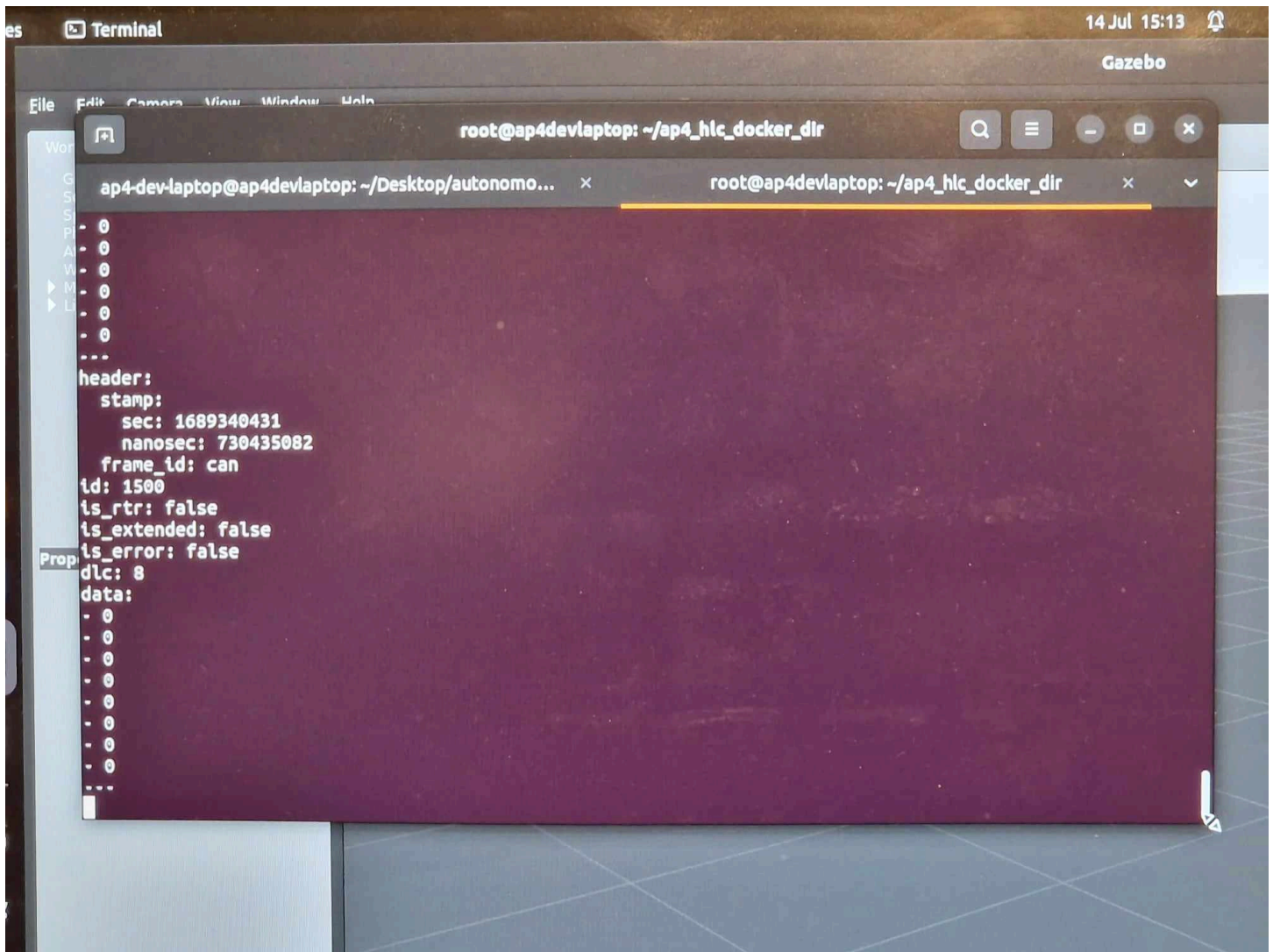
```
source /opt/ros/humble/setup.bash
```

- View topics on the ROS network

```
ros2 topic list
```

- If everything works, the sensor message should appear under the topic named /from_can_bus. This can be viewed by running the following command:

```
ros2 topic echo /from_can_bus
```



2 Embedded Software and Hardware (CAN nodes)

2.1 Introduction

This document aims to describe how the embedded software layer is built in software and hardware, what purpose embedded software serves, highlight the design principles and how to extend functionality.

Keep in mind, this directory contains the software for several different ECUs mounted to autonomous platform generation 4.

2.1.1 Prerequisites

These are useful topics, skills and softwares to have some understanding of in order to work with the embedded software, hardware and CAN database

Recommended softwares to have installed:

- PlatformIO VSCode extension + ST-Link v3 Windows drivers
- KVASER CAN database editor
- Fusion360 (If you have to MODIFY existing CAD files)

Versions used when developing AP4 spring 2023: - Visual Studio Code, version: 1.79.0 (user setup) - Platform IO: version core: 6.1.7 - Kvaser Database Editor: version 2.4, [link to download](#)

2.1.2 Introduction

This directory contains several software components that is run in an embedded environment on ECUs mounted to the autonomous platform. There can be many different ECUs running at the same performing different tasks. All the code is

contained in this directory in separate sub directories. This directory also contain the unified CAN database files which describes how data and information is sent between the different ECUs and the Hardware Interface Low Level software running on the Raspberry Pi 4b.

ECU functionality can be split up in several ways, for example by functionality. Software related to controlling the autonomous platform can be implemented on one ECU. Then on another ECU software for measuring speed can be run. It can also be split up by when certain functionality was added, instead of adding functionality to an existing ECU it can be easier, software wise, to create a new ECU with new software.

The new generation of Autonomous Platform is designed with the centralized Electrical / Electronic (E/E) architecture in mind.

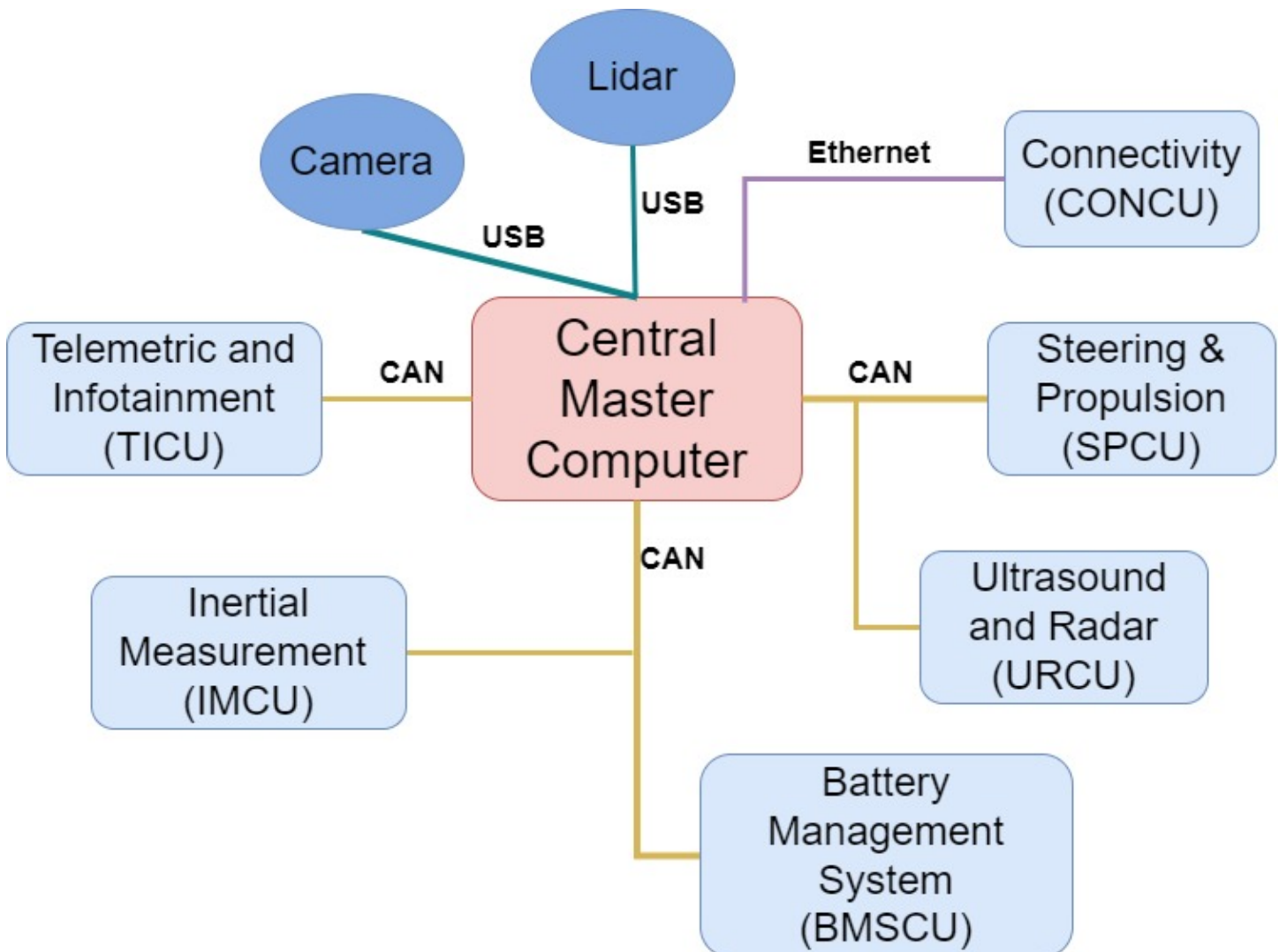
HOW_TO_EXTEND.md: The process of constructing a new generic ECU base from scratch

HOW_TO_PROGRAM_A_BLUEPILL.md : The flashing procedure (Upload New Software Into ECU)

2.1.3 Centralized E/E Architecture Principles

The E/E Architecture of autonomous platform generation 4, developed during a thesis spring 2023 by Erik Magnusson and Fredrik Juthe. Currently (june 2023) only central master computer, CONCU and SPCU are implemented.

A proposed system architecture can be seen below, where the “Central Master Computer” is consists of the Raspberry Pi 4b and development laptop. Every light blue rounded box would then be a separate ECU split by functionality. The specific ECU nodes are not set in stone and was more of a suggestion long term goal for future work during the spring 2023 thesis. What different ECUs are actually implemented are described further down below.



[Paper on approaches to centralized hardware can be found here..](#) The main points to keep in mind when developing software and hardware for a centralized E/E architecture are:

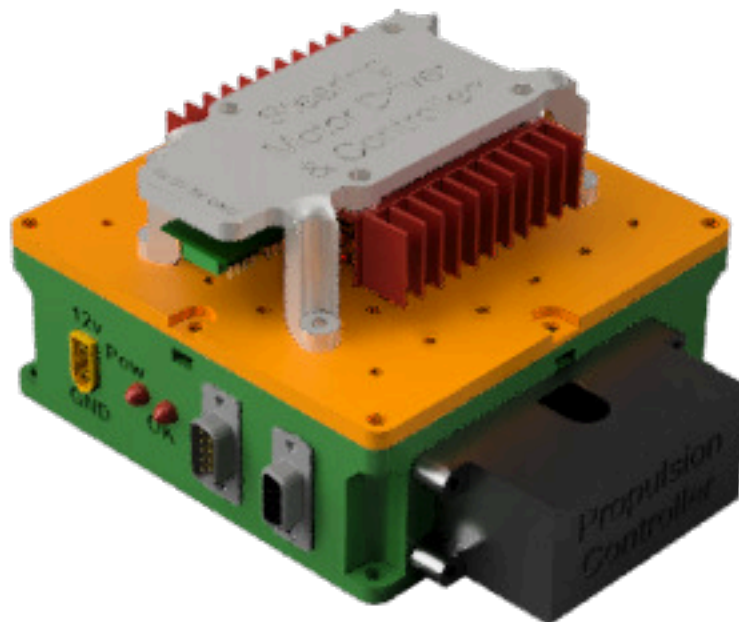
- No Heavy computations inside ECUs
 - Only serve as an INTERFACE with hardware
 - Minimal number of computations and instructions in each ECU to keep the run-time-cycle short and enable real-time capabilities
 - Main loop should not exceed 20ms in runtime
 - Read actuator commands from central master computer and actuate actuators
 - Read sensor data from hardware and send back to central master computer
 - Heavy computations are done in Central master computer and its High level algorithms
- Communication between ECUs and Hardware Interface and Low Level Software shall be done through CAN bus protocol
 - Due its robustness
 - Scalable
 - Standards, possible to use 3rd party softwares and tools to analyze behavior

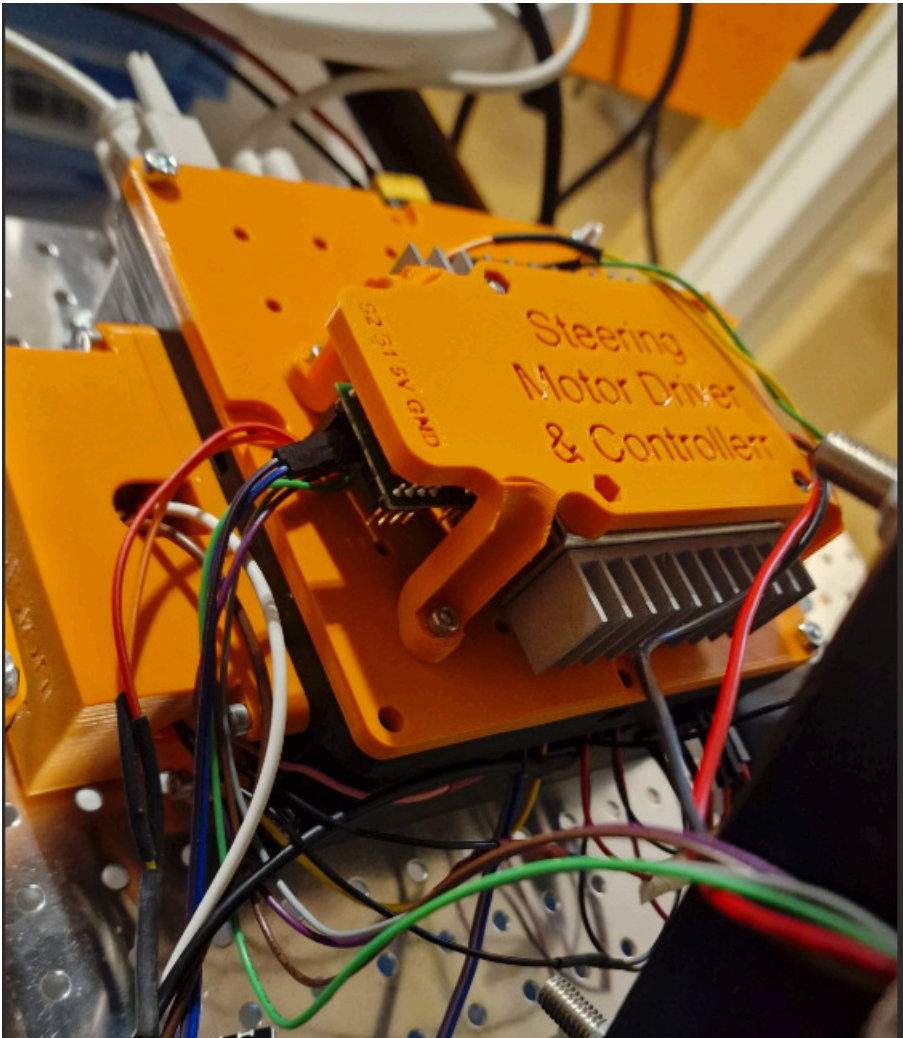
2.1.4 Embedded ECUs

This section describes the embedded ECUs present on autonomous platform generation and what function they serve. As of august 2023 there are two ECUs implemented.

2.1.5 HW_Node_SPCU : Steering and Propulsion Control Unit

Developed by Fredrik Juthe and Erik Magnusson spring 2023.





This ECU is responsible for actuating the propulsion and steering of autonomous platform generation 4. It also sends measurement of the current steering angle back to the hardware interface low level software.

The code is located in `HW_Node_SPCU` folder.

Specific CAD and stl files for front and back speed sensors are located in `autonomous_platform\CAD\Speed_sensor_holder`.

More about this ECU, development processes and documentation can be found in `Propulsion_Steering.md`, `Shared_HW_Node_Libraries/` and `Shared_HW_Node_Libraries/PropulsionInterface/documentation_and_research/README.md` markdown file located in this directory.

2.1.5.1 Additional Components Compared to a generic ECU on autonomous platform, the SPCU ECU has some extra components:

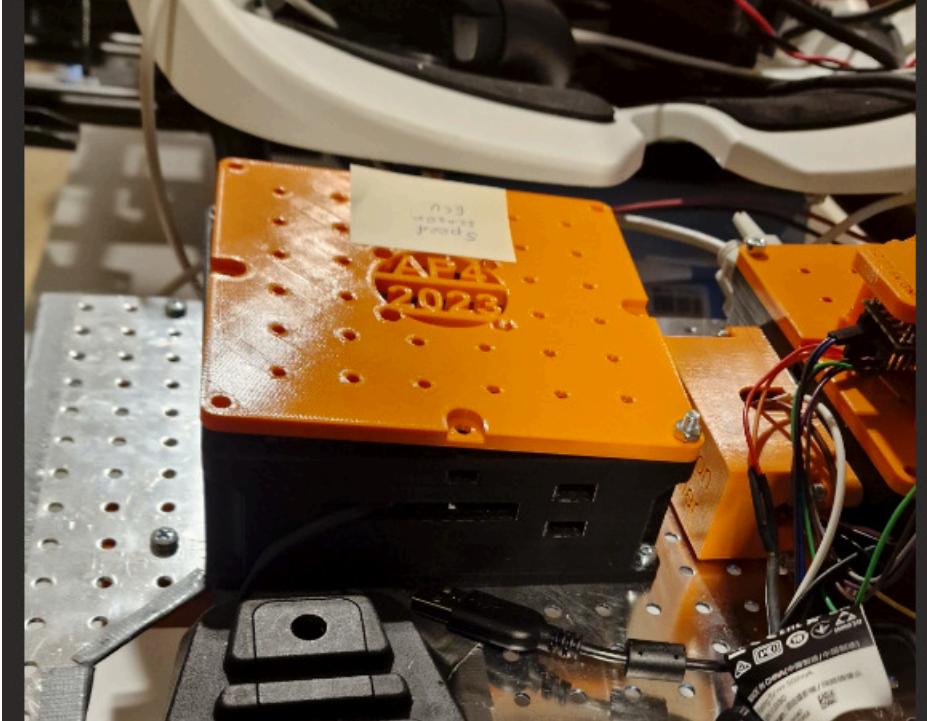
- Kangaroo x2 Controller + Sabertooth 2x50 DC motor driver
- MCP4725 DAC converter

The kangaroo x2 controller board is connected to the sabertooth 2x50A DC motor driver. It is a closed loop controller for the steering wheel motor. The kangaroo card can be controlled through a serial connection from the ECU. The current steering angle can be read from the kangaroo card. A custom library was created in order to control the steering angle.

To control the propulsion of autonomous platform generation 4, it was decided to send out analog voltages to the gokart pedals to spoof the pedals being pressed. (Throttle + Break). This is done by wiring a Digital to Analog Converter (DAC) to each pedal in parallel. The gokart has an internal controller black box mounted in the front nose, this will interpret the pedal presses and actuate the electric motor driving the platform forward. A brake pedal press will always override a accelerator pedal press due to how the black box works. This is a useful feature, when an operator is sitting on the gokart he or she can always stop the platform by applying the brakes.

2.1.6 HW_Node_SSCU : Speed Sensor Control Unit

Developed by Seamus Taylor and Alexander Rydeval summer 2023.



This ECU is responsible for relaying the current velocity of the autonomous platform to the hardware interface and low level software. One speed sensor was mounted to the front right wheel on the hardware platform and it was showed that the velocity could be read from one wheel by measuring how fast the wheel turned.

There exists CAD files for the remaining three speed sensors (one for every wheel).

The code is located in HW_Node_SSCU folder.

Specific CAD and stl files for front and back speed sensors are located in `autonomous_platform\CAD\Speed_sensor_holder`.

More about this ECU can be found in `Speed_Sensor.md` markdown file located in this directory.

2.1.6.1 Additional Components Compared to a generic ECU on autonomous platform, this ECU has some extra components:

- MCP4725 - Speed Sensor

2.1.7 Generic ECU Hardware and Software

The first version of the generic ECU was developed by Fredrik Juthe and Erik Magnusson spring 2023.

Every ECU present on autonomous platform generation 4 shall be based on the same base hardware and software. Onto this base features specific to the function of a ECU can be added. By having a standardized base ECU it will be easier to understand how previous functionality was added, and it will be easier to add future functionality as much information can be transferred from previous ECUs.

A problem on autonomous platform generation **3** was that it was unclear how previous functionality was implemented. There were more than three types of microcontroller used on the platform and every solution was custom made. As the people who implemented this left the project it was very unintuitive for new project members to understand how functionality was implemented. Many functions were hardcoded on unknown hardware.

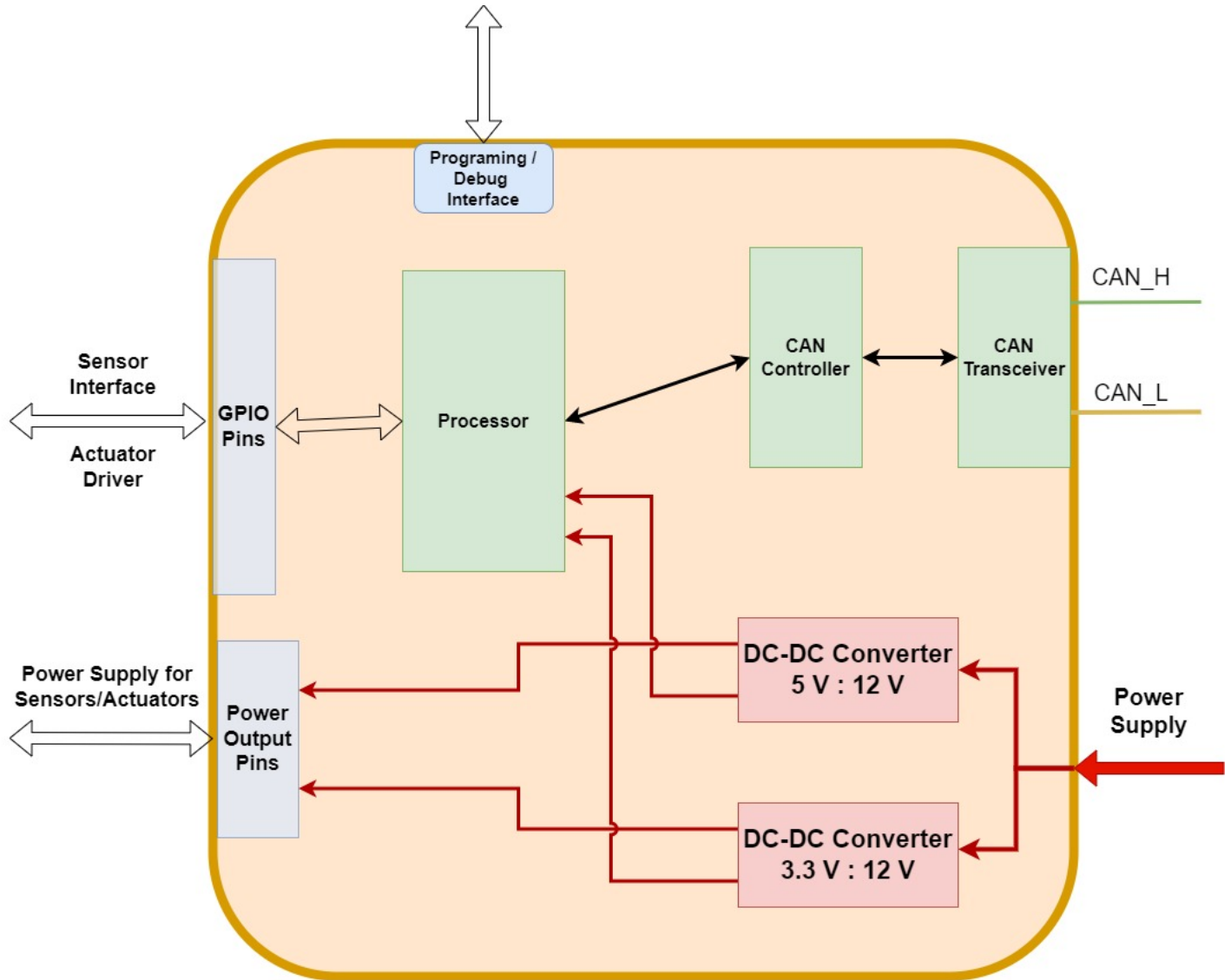
As an improvement to this, on autonomous platform generation 4 the idea was therefore to standardize how embedded hardware and software implemented. If one chooses a suitable microcontroller with a lot of performance overhead, in theory any embedded sensor or actuator could be connected to it. The hardware for each embedded system could therefore use the same microcontroller. And to simplify things further, the most used components could be constructed as a finished hardware package. The software and connection for a specific function could be configured.

That was how autonomous platform generation 4 ended up with a **generic ECU base**. A hardware template on which to connect any sensor or actuator to a powerful microcontroller. Any future ECU should follow this standard of implementing functionality on top of a ECU base.

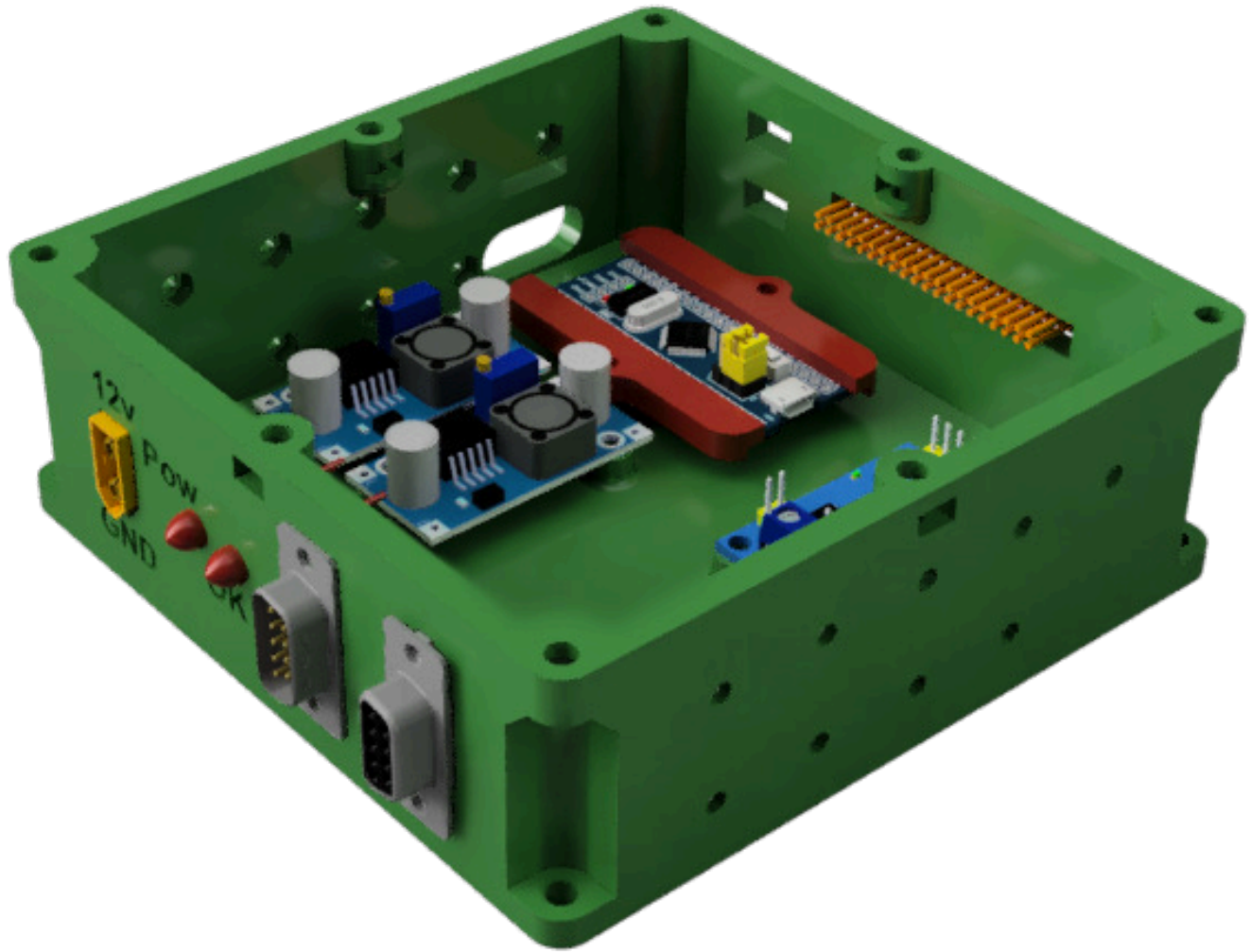
2.2 Generic ECU Template

The generic ECU has predefined set of hardware. It was designed with future proofing in mind so that it could be used for future work as well.

A high level schematic of the components present in the generic ECU base can be seen below. For any new embedded functionality (Input or output), these components would be needed. Therefore it might as well be assembled into a module.



A rendered illustration of the ECU can be seen below. The top cover is not illustrated.



2.2.1 Key Components of Generic ECU

- **Microcontroller** : The microcontroller was Chosen to be STM32F103C8T6, often referred to as an STM32 bluepill. It has plenty of I/O, a fast processor and there exists very much documentation and resources for this microcontroller. The official documentation can be found for the STM32F103C8T6, can be found [here](#).
- **CAN Reciever and Controller**: The ECU needs to be connected to the Hardware Interface and Low Level Software through a CAN bus, therefore the generic ECU needs a CAN bus interface. The MCP2515 / TJA1050 board was chosen as it combines a CAN controller and a CAN tranciever on a single PCB. There exists many arduino libraries for this. The datasheet for MCP2515 / TJA1050 can be found [here](#) and [here](#).
- **DC-DC Converters** : The components inside the generic ECU needs to be powered using 3.3v and 5v. And any additional sensor or actuator may need to be powered as well. Therefore, the generic ECU base has two DC-DC voltage step down converters. The LM2596 DC-DC converter was deemed suitable. It can output a maximum of 3 A of current which is well above what any embedded sensor would use. Datasheet for the LM2596 dc-dc converter can be found [here](#).
- **Inputs and Outputs**: The generic ECU node has a predefined set of Input and Outputs. This allows the connection

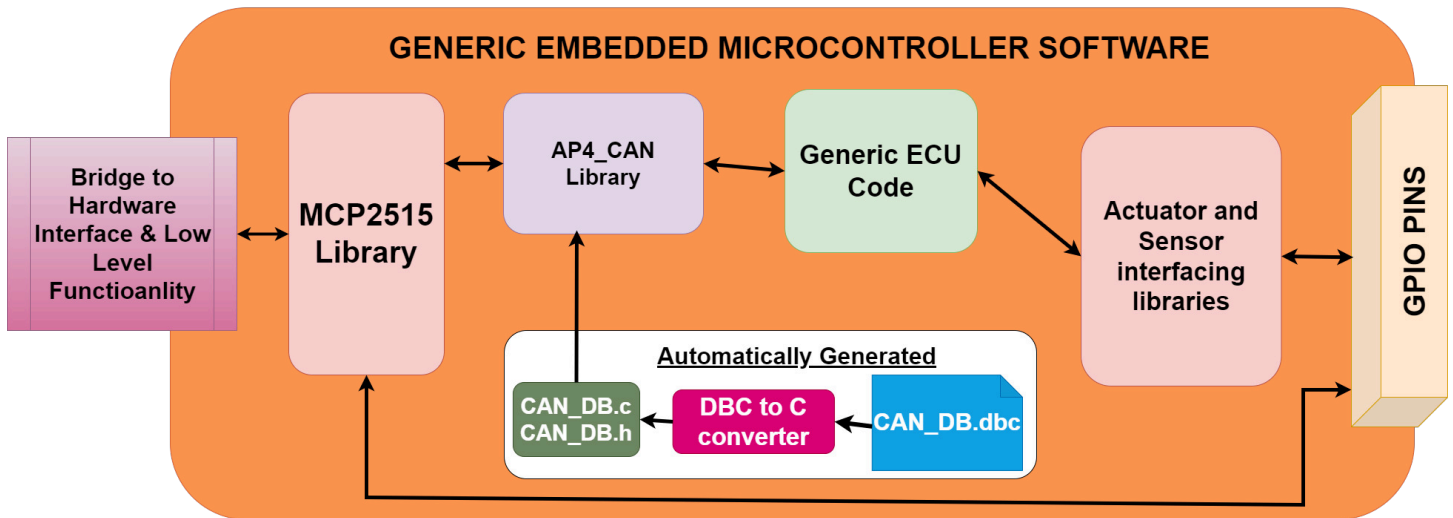
between various ECUs and the Raspberry Pi 4b hardware interface and low level software to be standardized. The CAN communication on AP4 uses a D-Sub 9 pin connector. The generic ECUs can be chained together with D-Sub 9 pin cables to create a CAN bus network. In the same way the power distribution of autonomous platform generation 4 can be standardized, power is supplied to the generic ECU through an XT60 connector. This one can also be chained between nodes, providing 12v input to each generic ECU.

Component	Purpose	Documentation
STM32-F103C8T6 “Bluepill”	Programable processor and GPIO pins	Datasheet
Logic converter	To step down voltage to SMT32-F103C8T6 logic levels (3.3 V). Important at the CAN module interfacing with bluepill	Bought at
DC-DC converter	LM2596, possible to manually change voltage levels	Datasheet
CAN controller and Transeceiver	MCP2515, to interface with CAN network, decode and encodes messages	Datasheet
PC Fan	Fan to cool each component in the ECU	Bought at
DB9 Male	Outlet for CAN bus communication	
XT60 Male	Outlet for power supply	

2.2.2 Software Template and Setup

In the same way as the hardware as been standardized, work has been done to standardize the software development and deployment on autonomous generation 4 embedded hardware.

The idea is that basic software needed for every generic ECU base should already be implemented and setup. A developer should not have to start from scratch and rewrite the basic interfaces has already been used on other generic ECU nodes.



An illustration of what software components are needed for every generic ECU can be seen above.

When developing a new ECU with new functionality, the developer only has to think about the green and purple code blocks, the main loop code and what libraries to add.

Therefore, a template ECU code directory has been created, `HW_NODE_CODE_TEMPLATE`, this is a PlatformIO project that have been configured with the correct path variables to find the custom CAN library or any other custom made library.

Microcontroller and other configuration variables have been set in the `platformio.ini` file. To create software for any new future ECU one simply has to make a copy of this directory to get a basic software running on the ECU node.

Just as the Arduino IDE, PlatformIO has a library for almost any sensor or actuator. You can search the project library here. To include an existing library into a PlatformIO project one simply has to include it under **lib deps** tag in the `platformio.ini` file.

2.2.3 Bill of Material for Generic ECU

The complete bill of materials for a generic ECU base can be found below.

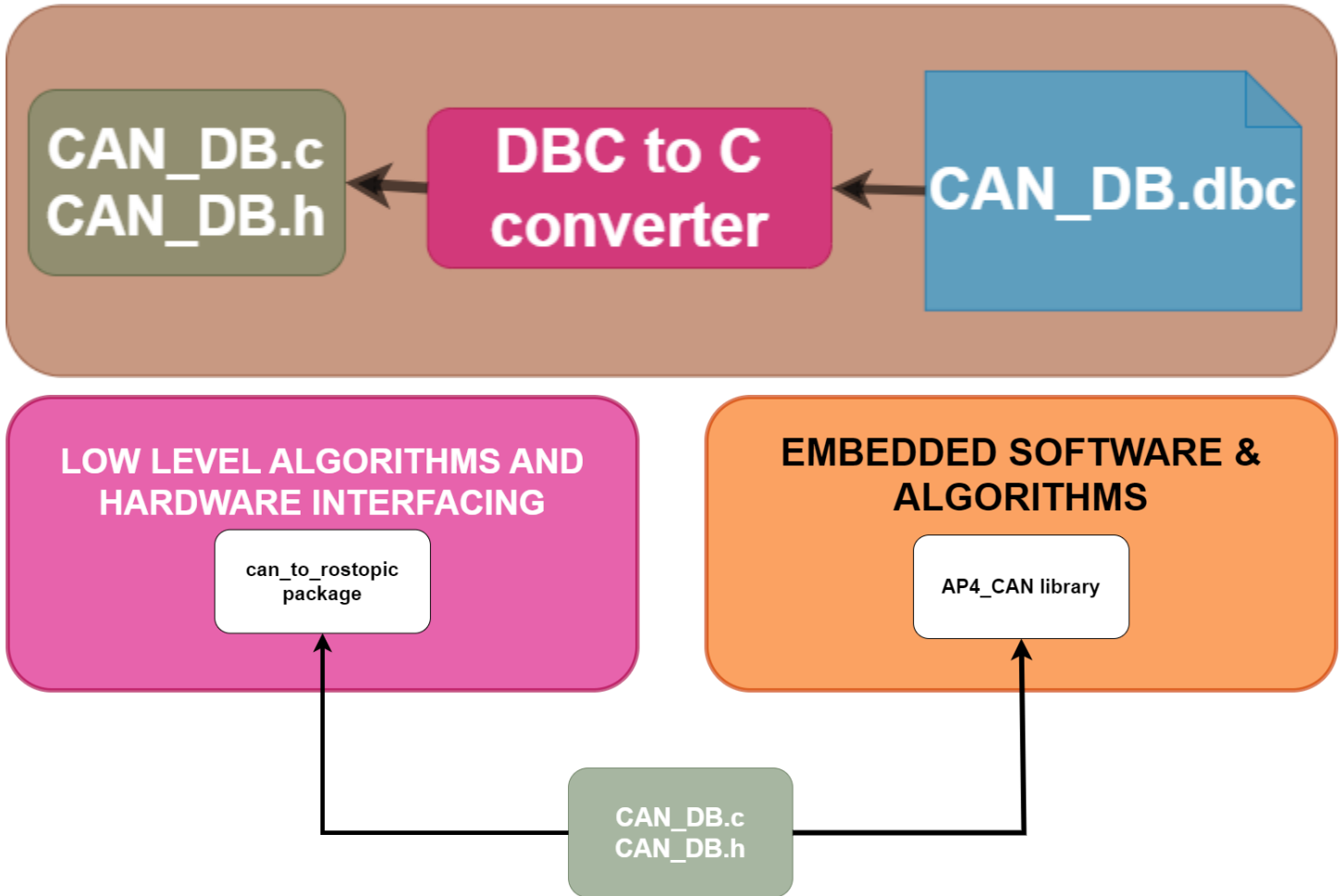
Component Name	Function	Quantity	Datasheet/ Manual	Product Buy Link	Estimated price (SEK)
STM32F103C8T6 “Bluepill”	Microcontroller - Runs embedded SW	1	Link	Link	100
MCP2515	CAN controller / transceiver	1	Link	Link	50
Logic Level Converter	Converts MCP2515 5V signals to 3V which makes it compatible with STM32 Bluepill	1	Link	Link	40
LM2596S	Converts 12V to 5 & 3.3V	2	Link	Link	50
DB9 Breakout Board	Exposes individual pins on DB9 Connector to Connect wires to	2	—	Link	130
40x40x10 mm 12V Fan	Cool Down Internal Components in ECU base	1	—	Link	52
Breadboard Jumper Wires Pack of 10	Wire Internal Components of ECU	2	—	Link	20
Biltema Mini Fuse Holder	Internal ECU Fuse holder	1	—	Link	33
Mini Fuse 3A	—	1	—	Link	25
XT60 Connector Male	Connect to AP4 12V Power System	1	—	Link	21
(Optional) XT60 Y Splitter	Splits one XT60 Connecting in Y configuration	1	—	Link	70
Green LED	@Future work Indicate System Status of ECU	1	—	Link	2
Red LED	@Future work Indicate Power Status of ECU	1	—	Link	2
M3x10mm Screw	Screw Down Internal Components	16	—	Link	—
M3x10mm Nut	Screw Down Internal Components	14	—	Link	—

2.2.4 Interfacing with Hardware Interface and Low Level Software

The embedded software can interface with the hardware interface and low level software. Physically, this is done through a CAN bus network (D-Sub 9 connector between nodes). Software wise, a custom encoding and decoding CAN library has been implemented in C language. Onto this custom library, two software libraries have been created for autonomous platform generation 4.

The data sent over the CAN bus needs to be available on ROS2 topics in order for ROS2 nodes to be able to use the data in computations in the hardware interface and low level software. In the same way, actuator commands sent from ROS2 nodes on ROS2 topics need to be sent to the ECUs over CAN.

The CAN frames sent over the CAN network must be encoded and decoded correctly and in the same way on both the hardware interface and embedded ECUs. Therefore a unified CAN database file has been constructed for autonomous platform generation 4. A helper library is used to convert the CAN database file into a set of C-style data structs.



Two custom made libraries have then been implemented, one in the embedded software and one in ROS2 in hardware interface and low level software.

How to extend this functionality with new CAN frames and signals is explained in detail in `Hardware_Interface_Low_Level_Computer\HOW_TO_EXTEND.md`. (The parts that are relevant to change in hardware interface and low level software is documented here). How to extended the software for the embedded ECUs is described in `CAN_Nodes_Microcontroller_Code\HOW_TO_EXTEND.md`.

2.2.5 Unified CAN Database

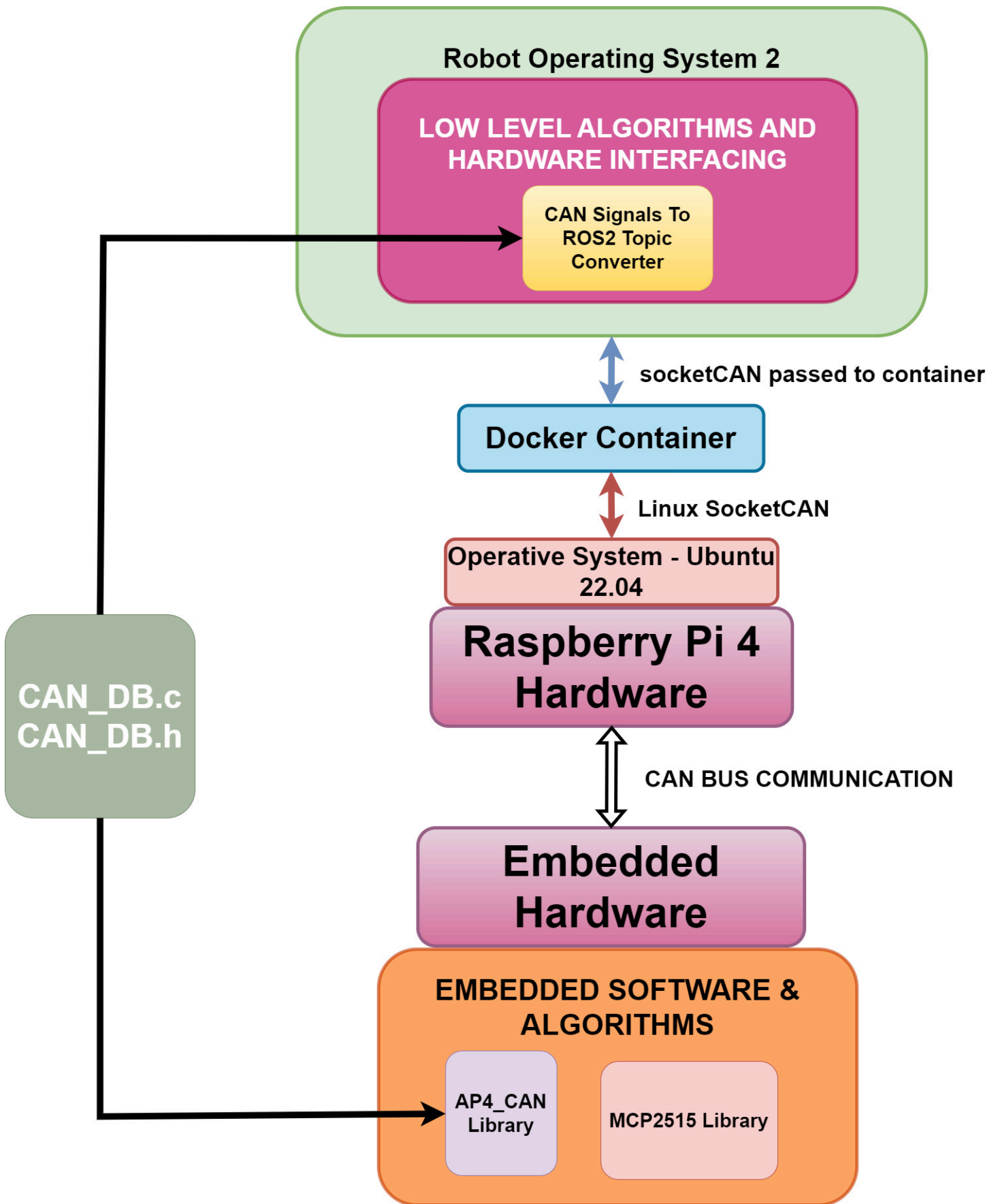
A unified CAN database has been created for autonomous platform generation 4. It uses the standardized .dbc file format and can be read/written to using commonly available softwares. I.e [Kvaser DBC editor](#).

The CAN library database file is located in `\CAN_Nodes_Microcontroller_Code\CAN_LIBRARY_DATABASE\CAN_DB.dbc`.

Every CAN bus frame and signal present on autonomous platform is defined in this file. This means when adding a new signal to the platform, it only has to be defined in one single file.

With an external library (added as a git submodule to this repository) standardized C code can be automatically generated. A simplified interface to encode and decode CAN frames and signals is created. The library used can be found [here](#). How to use it is described in `\CAN_Nodes_Microcontroller_Code\CAN_LIBRARY_DATABASE\README.md`

This CAN_DB.dbc (and autogenerated code from this file) can then be linked to various software components on autonomous platform generation 4. This is illustrated below.

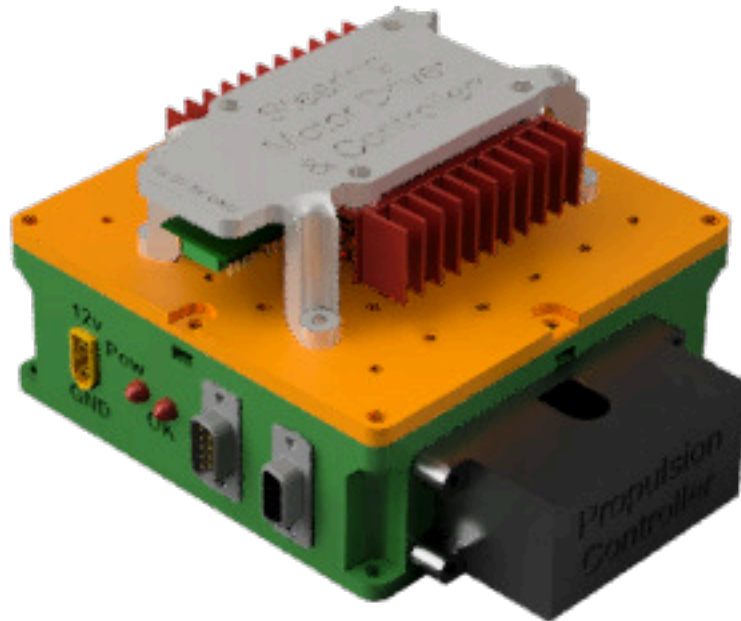


2.3 Steering Propulsion Control Unit (SPCU)

The implemented SPCU follows the structure and base mentioned in previous chapters. In the figure below is the circuit and schematic of the components used. Each module has a designated sw library located at: */Shared_HW_Node_Libraries*.

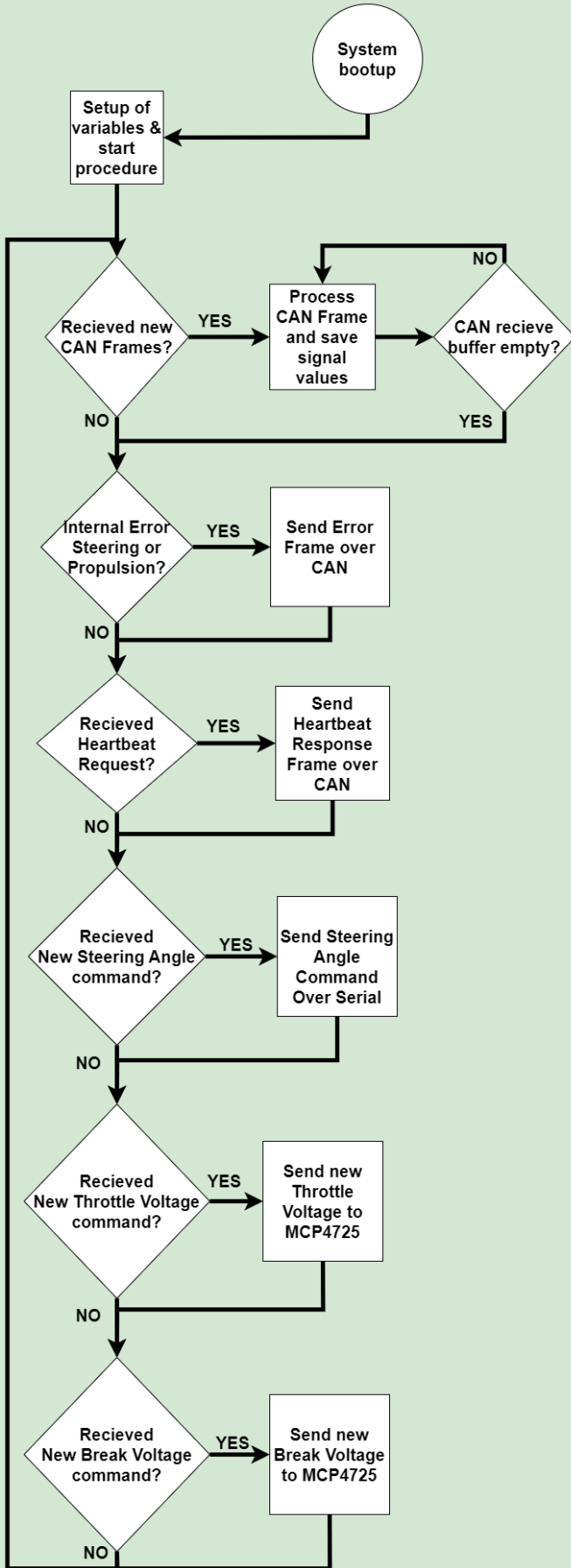
Below is a table of specific added modules for SPCU.

Component	Purpose	Documentation Repo Paths
MCP4725	Translates PWN voltage signals to Analog. In order to simulate the pedal voltages (0.8-4.3 V)	*\Documentation\Segway_Propulsion\Manuals
Sabertooth Kangaroo x2	Serial communication to send motor commands. SI interface and utilizes the limit switches.	*\Documentation\Steering_Motor\manuals
Sabertooth dual motor driver	Motor driver to control the DC motor used for steering, feedback with the built in encoder.	*\Documentation\Steering_Motor\manuals



Above is a rendered illustration of the SPCU, with its corresponding modules used for steering and propulsion. Down is a flowchart of the software used for the SPCU. The median run-time of the main-loop is 3 ms. Thus passing the requirement of 20ms for real-time systems.

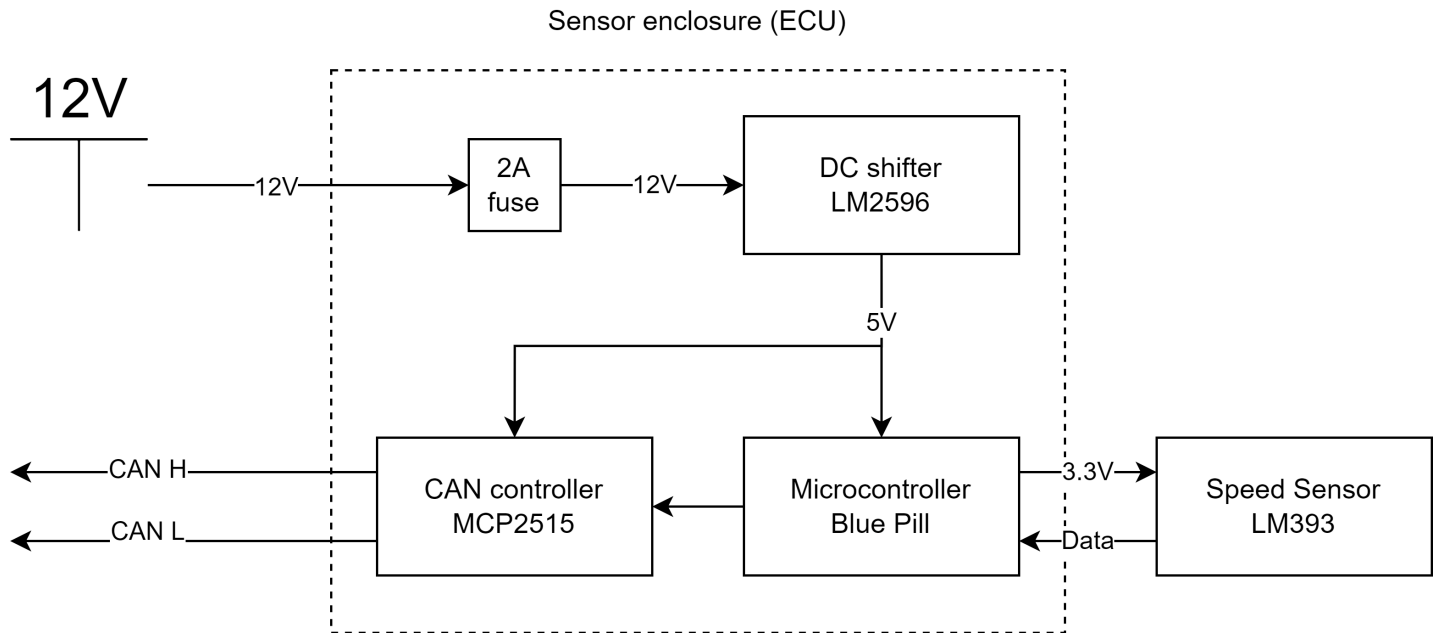
SPCU Embedded Software Logic Flowchart



2.4 Speed Sensor ECU

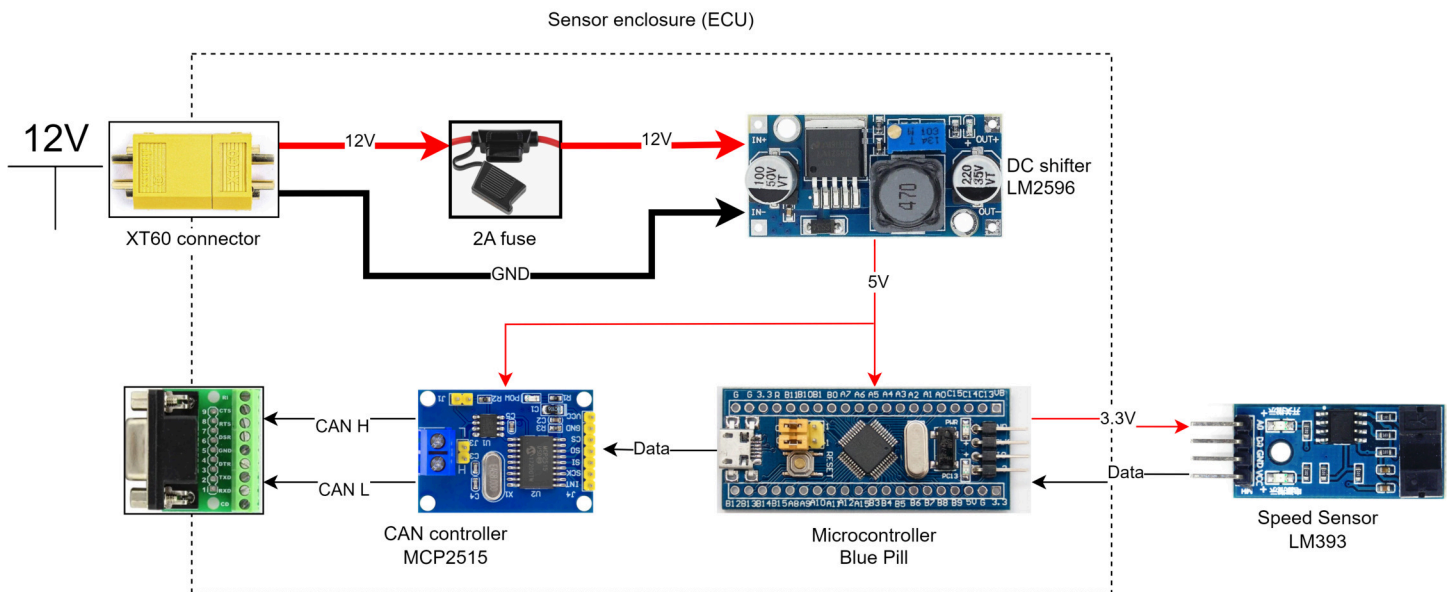
These are diagrams that describe the physical wiring of the speed sensor module. The design files are in the [diagrams](#) directory and can be edited. The diagram tool is diagrams.net, while the circuit diagram is made with EasyEDA online.

2.4.1 Functional block diagram



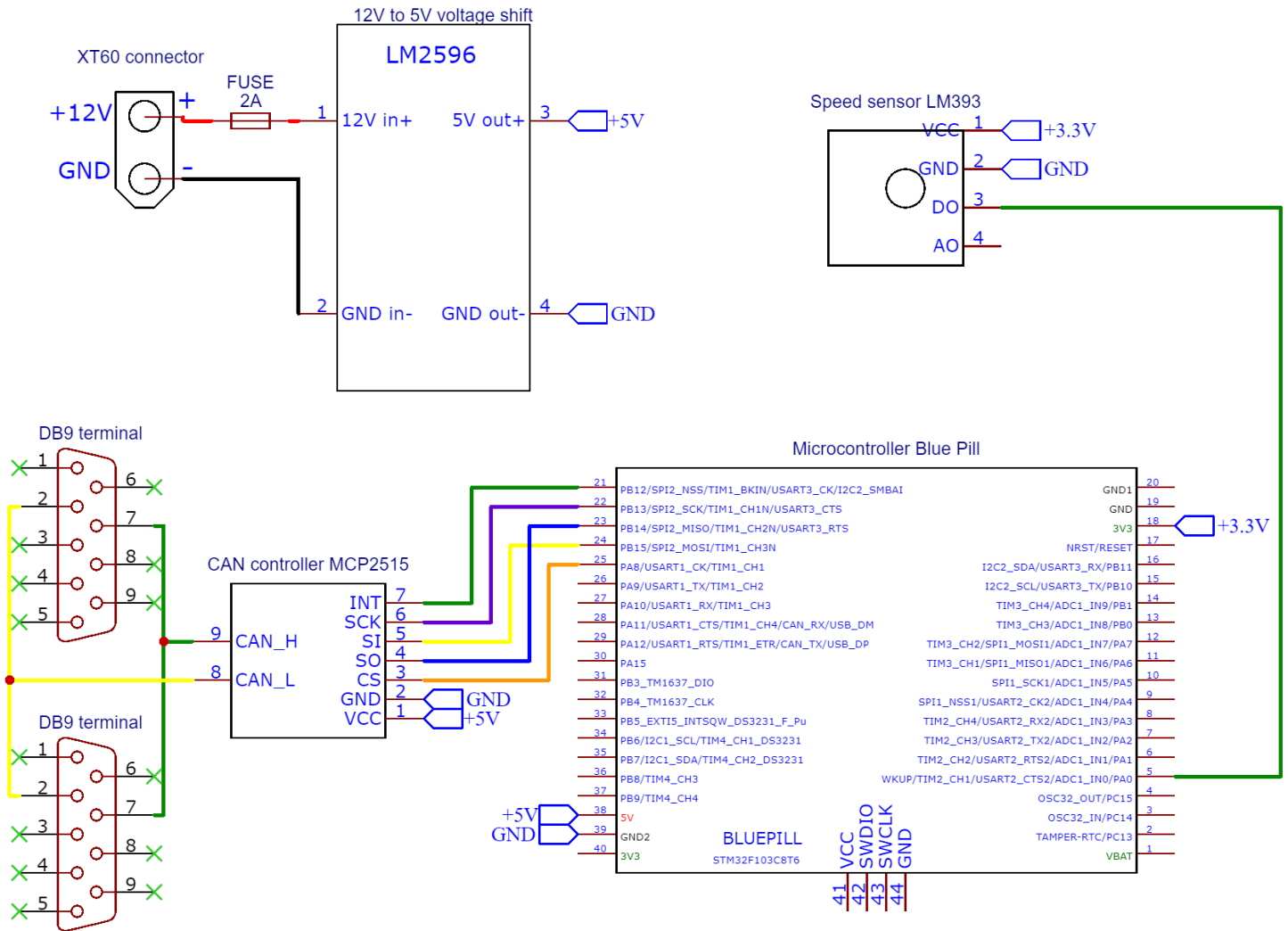
2.4.2 Wiring overlay

This is an overlay of the components in the circuit.



2.4.3 Circuit diagram

This is a complete representation of the how the ECU and its components are wired internally.



2.5 How to extend Embedded Software

This document aims to describe the process of how to extend the hardware interface and low level software.

Before starting to develop and adding code to the embedded software you need to first make sure you need to add something here.

If you;

- Want to add an embedded sensor / actuator
- Need to add a new CAN signal or frame to existing sensor / actuator

Then you are in the right spot!! If not, take a look at `Hardware_Interface_Low_Level_Computer` or `High_Level_Control_Computer`, maybe you intended to add functionality there!

2.5.1 Prerequisites

In order to start adding functionality it is recommended to have a basic understanding of:

- C++ OR Python development
- Embedded Development
- CAN bus
- docker containers (How to start, stop, restart and configure)
- Basic Linux - The container software environment is mainly navigated in through a terminal

Software wise, you need to have the following installed:

- docker
- git
- VSCode (recommended but any IDE may be suitable)

- PlatformIO extension

Hardware wise, it is recommended you have:

- Linux based x86 host computer

2.5.2 Add New Functionalities

First of all make sure you have read the general design principles document for autonomous platform generation 4 located at `autonomous_platform/HOW_TO_EXTEND.md`. This document takes precedence over anything written in this document in order to unify the development process across all software layers.

2.5.3 Standard base ECU

Autonomous platform generation 4 uses a standard base ECU (Electronic Control Unit) meaning every embedded software should use the same hardware. The ECU or node box has a standardized Input and Output. ECUs are connected together in a CAN bus network, this network contains the Hardware Interface hardware, the Raspberry Pi 4b, which links the network with higher level software. Onto the base ECU specific hardware for a specific function can be added. For example, for the SPCU (Steering Propulsion Control Unit) there are DACs and Motor Controller hardware connected to the ECU.

Inputs:

- CAN bus passthrough (Through DB 9 connector)
- 12v Power (Through XT60 connector)

Outputs:

- CAN bus passthrough (Through DB 9 connector)
- 5v Power
- 3.3v Power
- GPIO pins

It is preferred to build a new ECU node when adding functionality to autonomous platform to prevent breaking existing functionality. ECUs should preferably

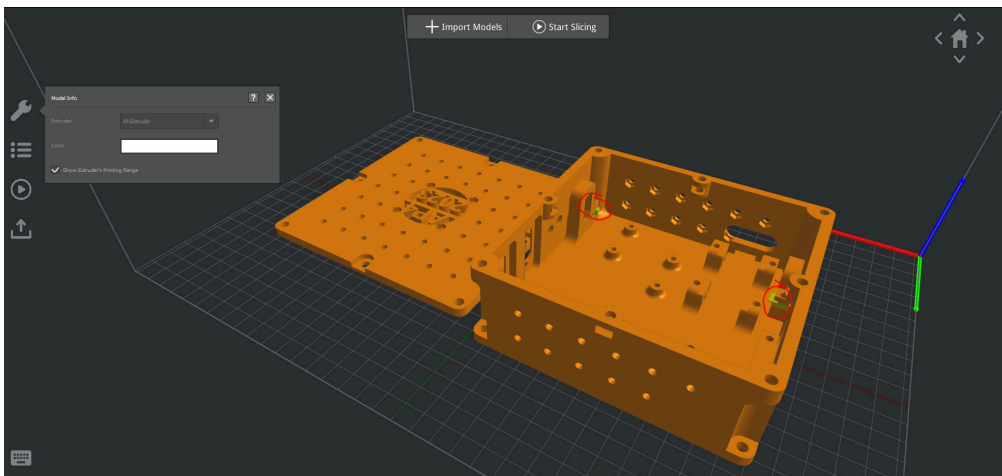
If new functionality cannot be implemented on an existing ECU node, a new hardware node must be built. See [How to Build new ECU](#).

2.5.4 build a new ECU

1. Start 3D printing generic ECU components (top, bottom, internal component holders) Please check `CAD` folder for the list of components that need to be printed.

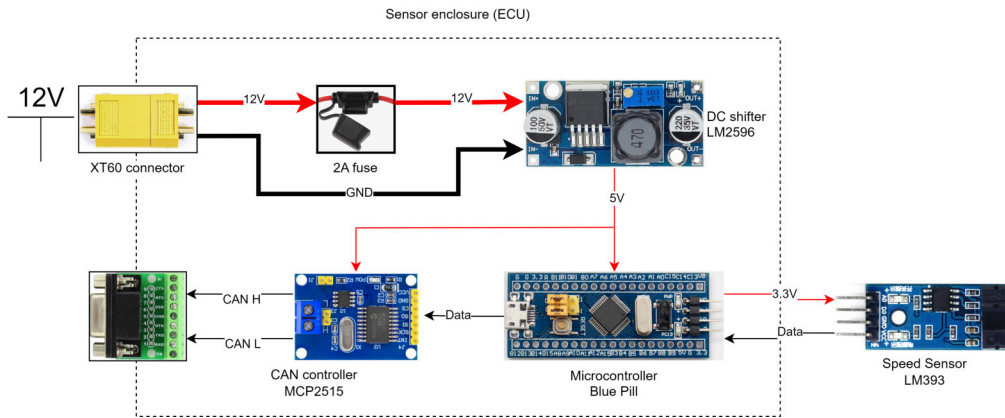
The estimated material consumption (with reasonable slicer settings) is around 200 grams. And total print time is around 16-18 hours. Grab some coffee and leave it printing overnight! Make sure the first layers stick properly to the printing bed before leaving.

Note: Make sure you add supports manually for the print-in-place XT60 connector clampdowns. Any other support is not strictly necessary. These are circled in red below.

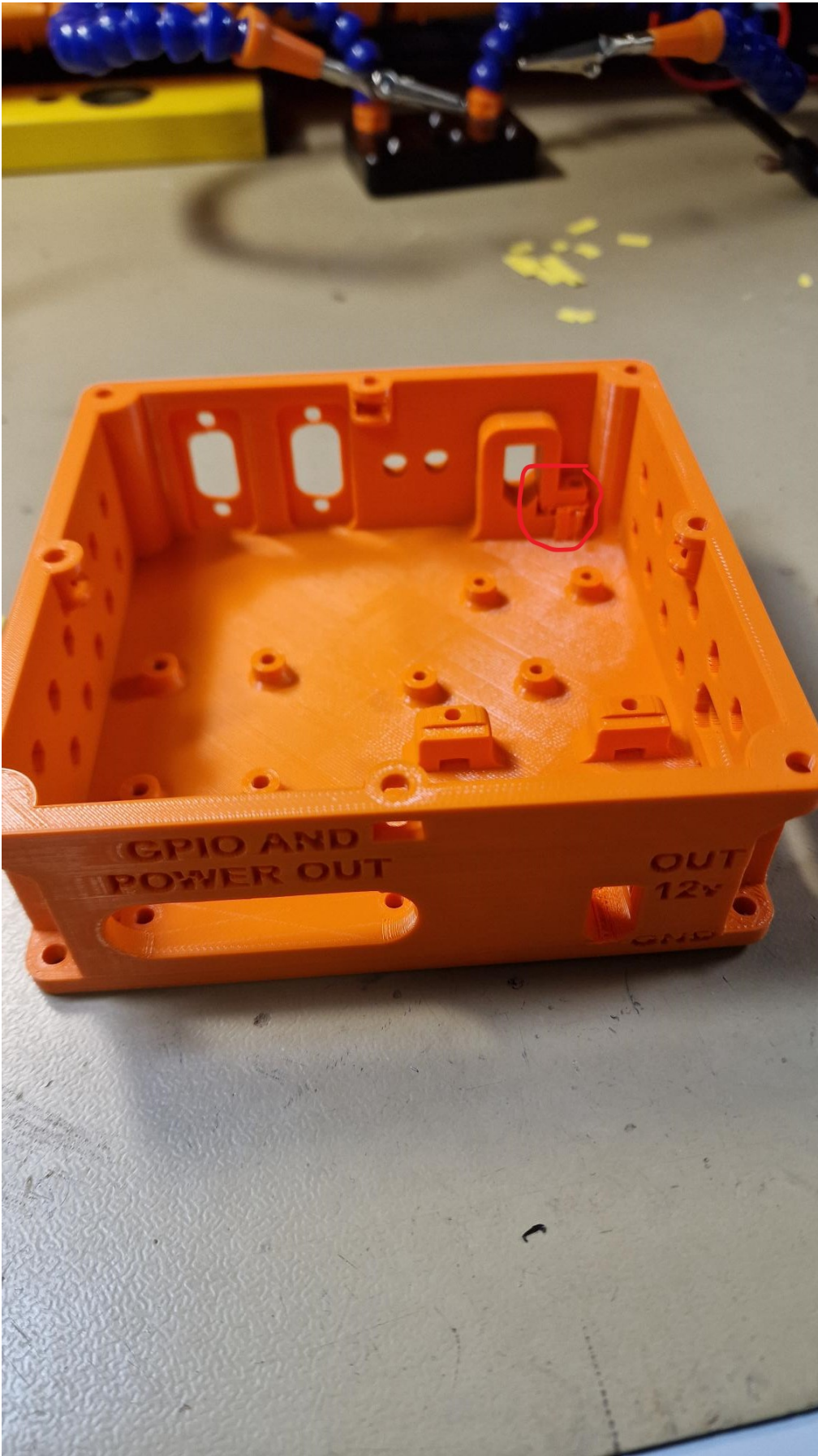


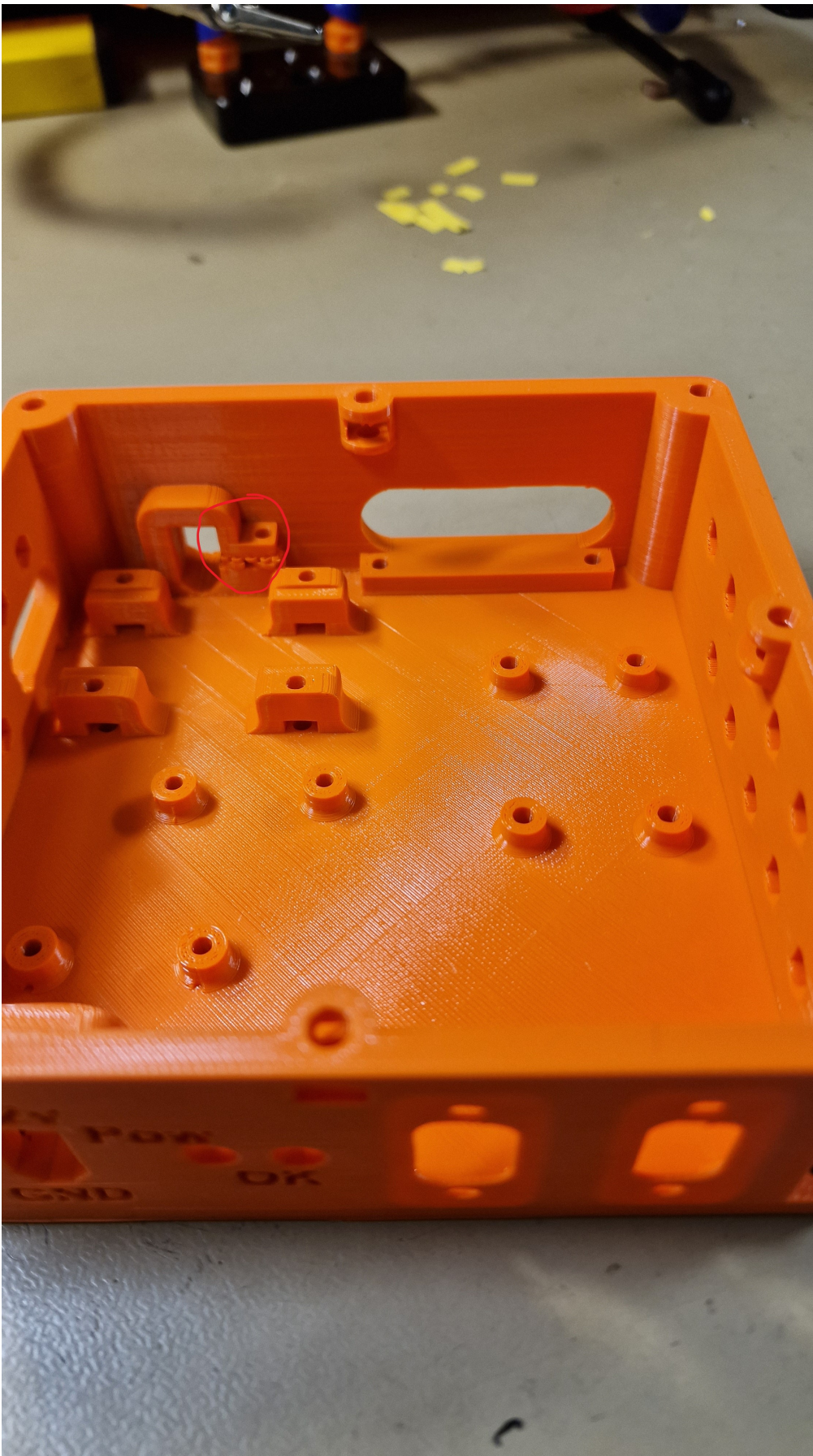
2. Make sure all components are working individually before mounting them inside the node. I.e flashing the bluepill, measuring output from dc-dc converters, making sure the MCP2515 can bus board is working

A schematic illustration of the components inside the generic ECU can be seen below.



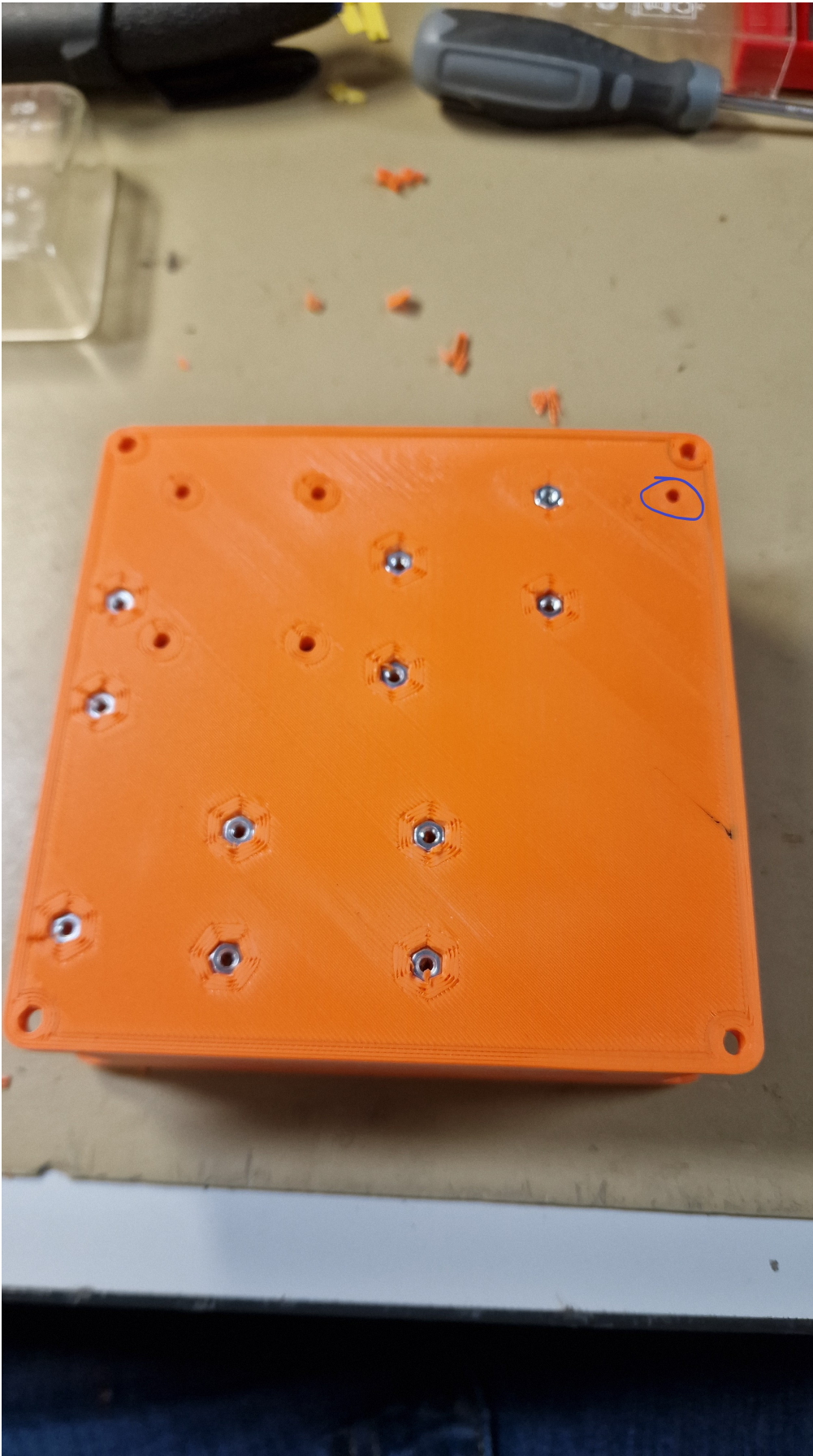
3. Once 3D print is complete, remove added support material for the XT60 clampdowns





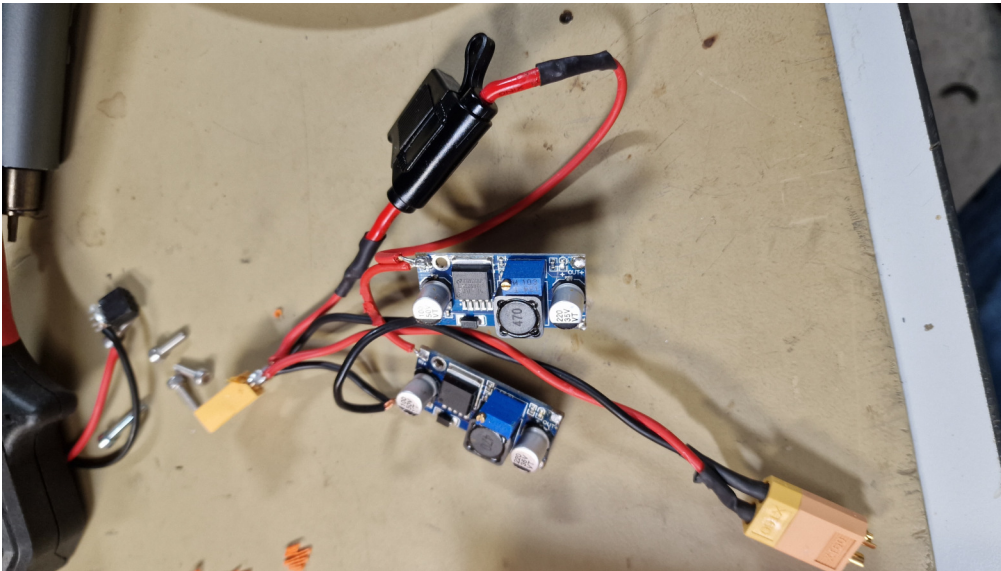
4. Insert 12 M3 nuts in the hex shaped holes on the underside

Note: The hole circled in blue shall also have a m3 nut.



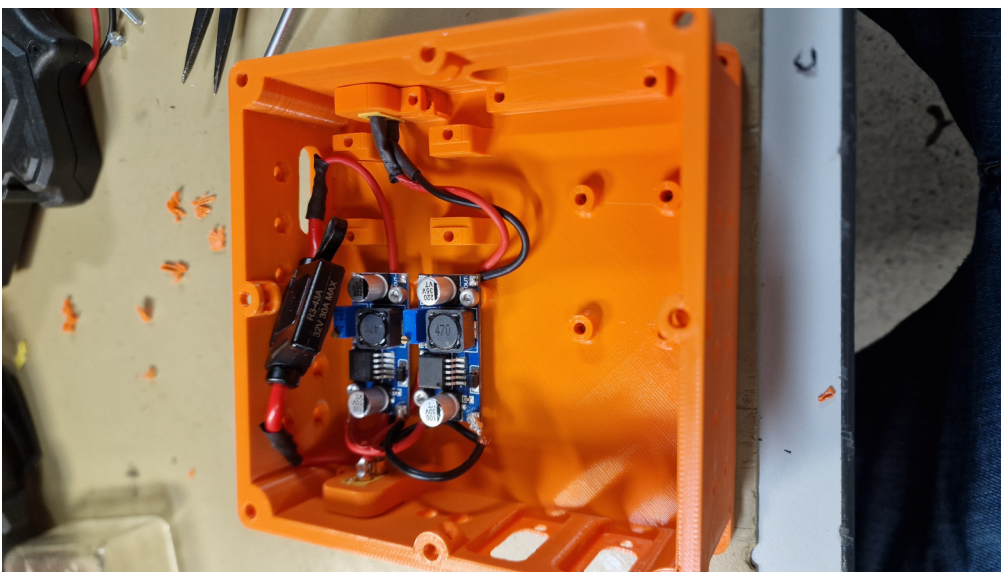
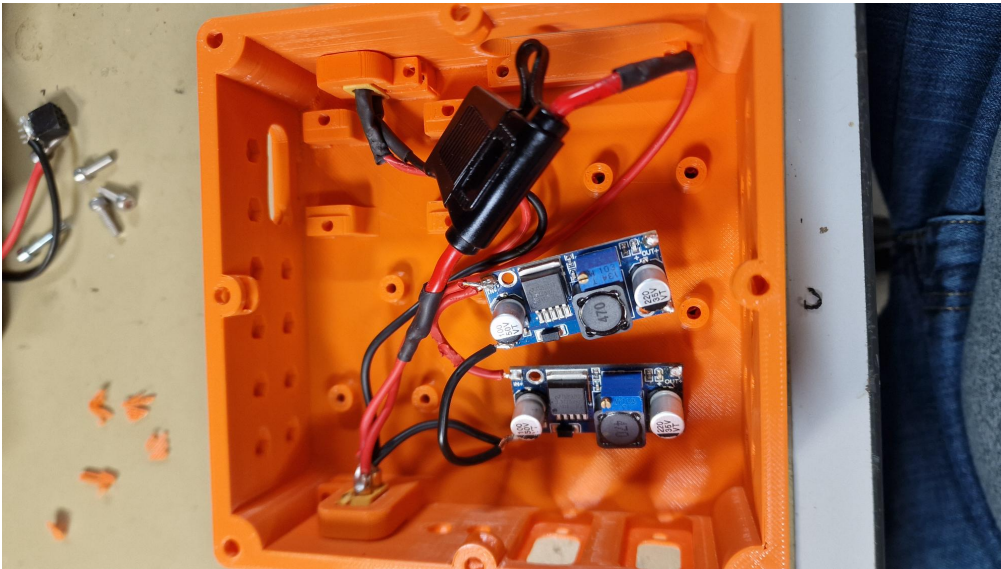
5. Assemble power distribution and mount inside HW node.

A schematic for the power distribution inside the HW node can be found below.

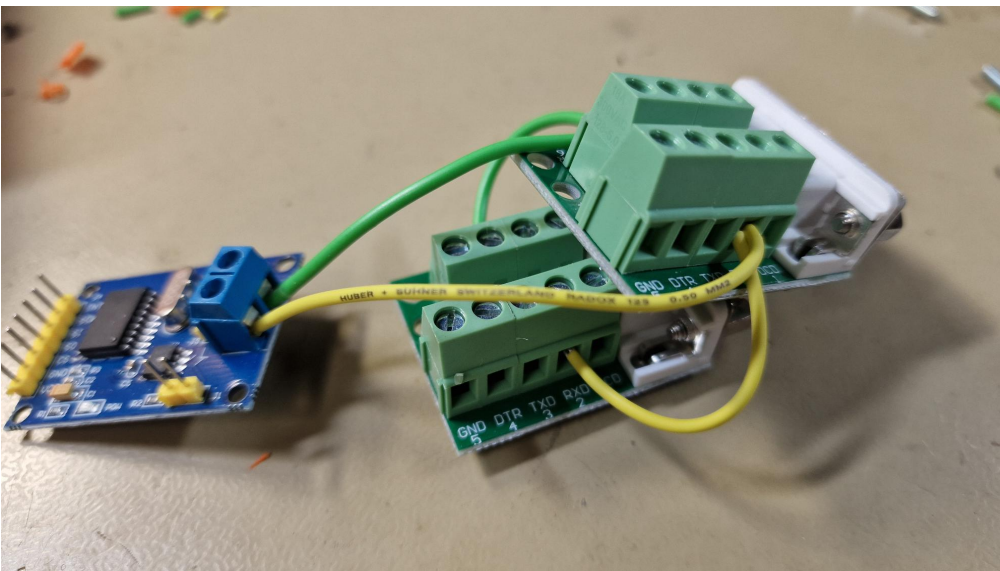
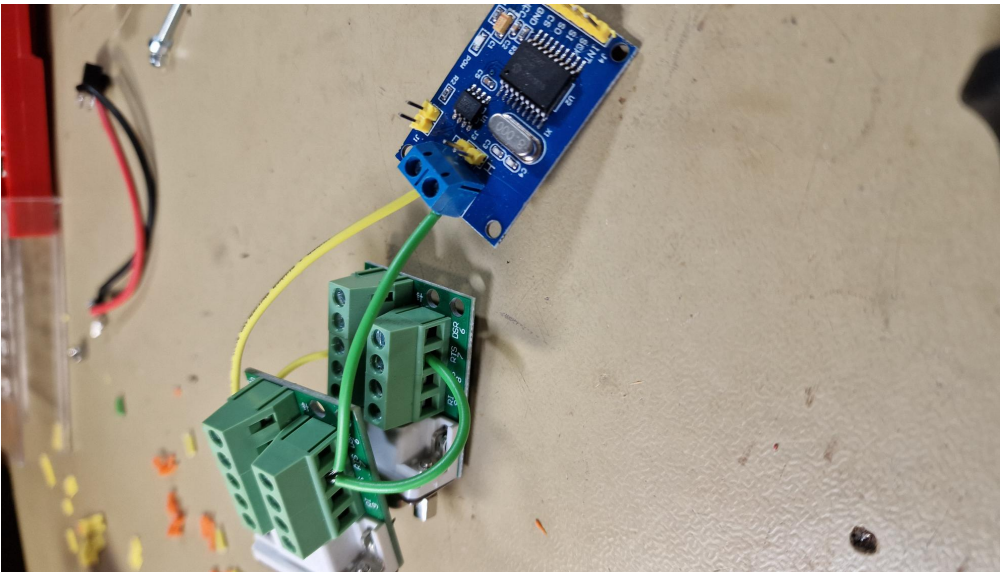


6. Mount power distribution inside node.

The XT60 contacts are inserted into the 3D printed clamp. The DC-DC converters are screwed in with 2 M3x10mm screws respectively.

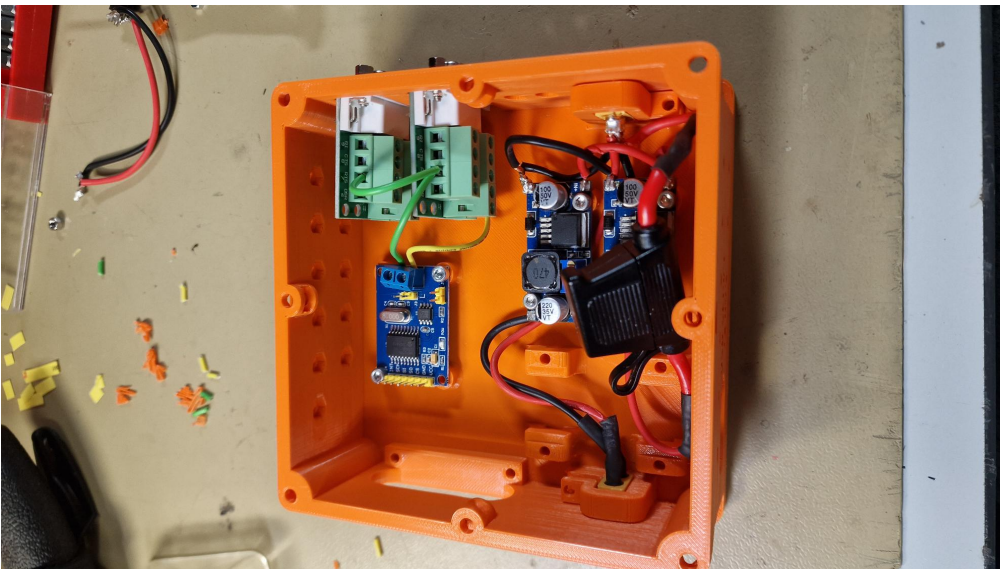


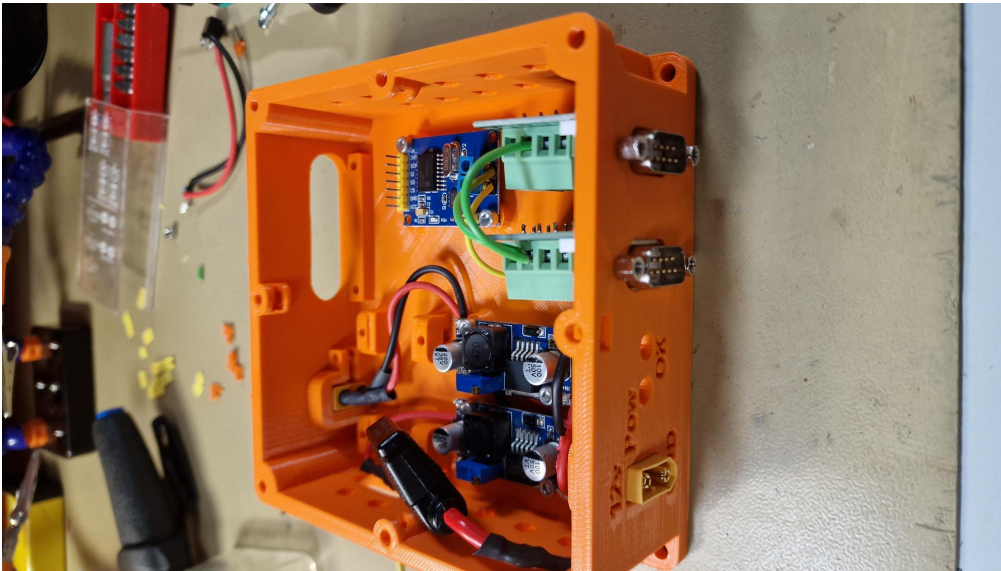
7. Assemble CAN module and D-Sub 9 connectors. A wiring diagram can be found below.



8. Mount CAN module and D-Sub 9 connectors inside HW node.

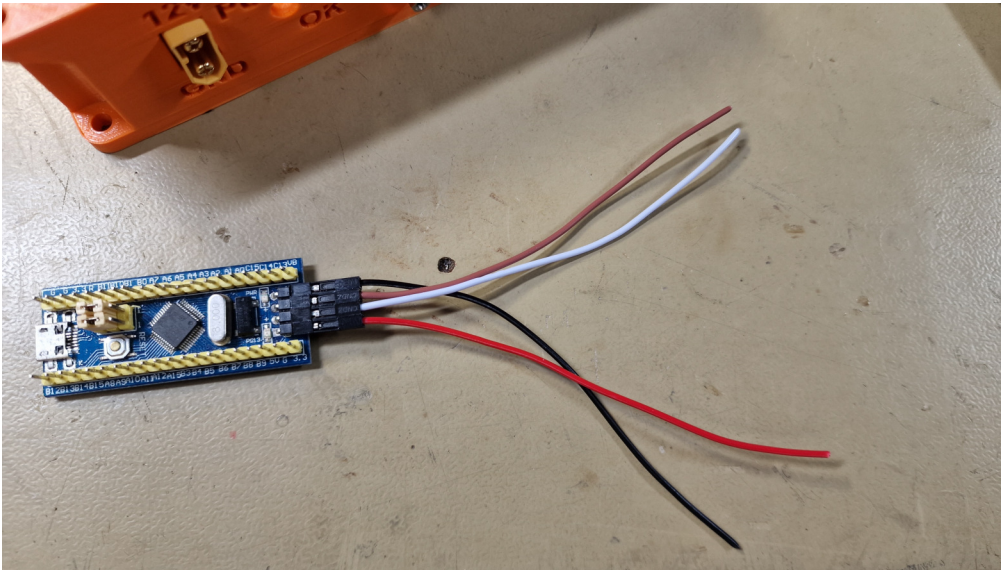
D-Sub 9 connectors are screwed in from outside. The MCP2515 card is screwed in with two M3x10mm screws.



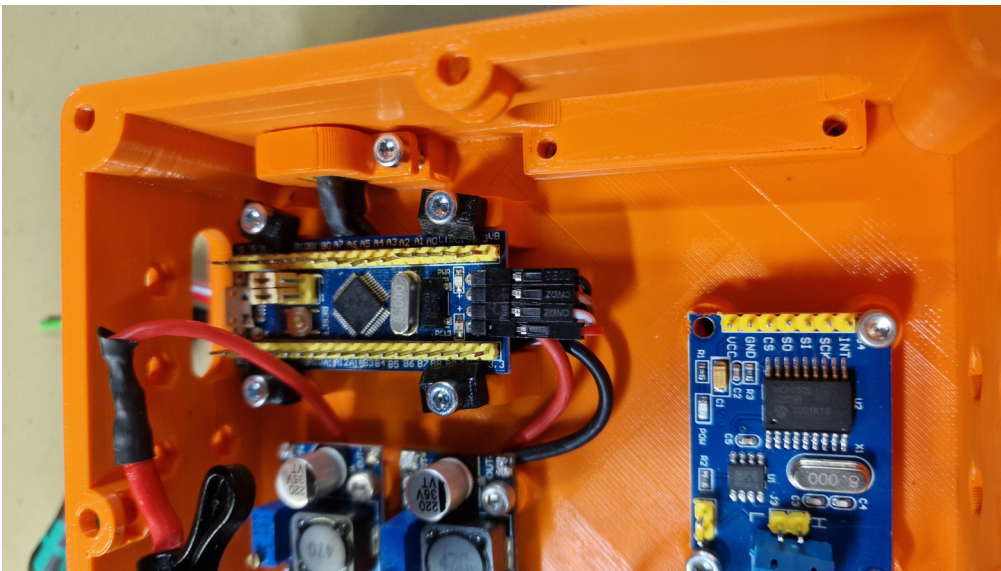


9. Insert 4 M3 nuts inside STM32 hold down pillars. Make sure the hole in the pillars align with the inserted nuts.

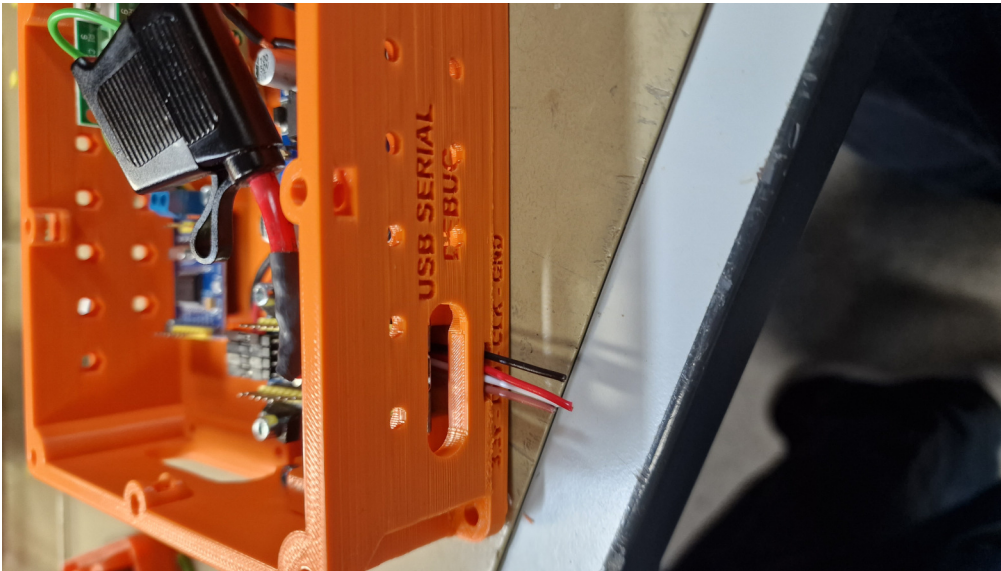
STM32F103C8T6 Bluepill Pin	Wire Colour
3.3v	Red
IO	Brown
SCLK	White
GND	Black



11. Using 4 STM32 bluepill clampdown (3D printed) tabs, clamp down the STM32 bluepill microcontroller.



The four wiring cables shall be passed through to the side of the HW node.



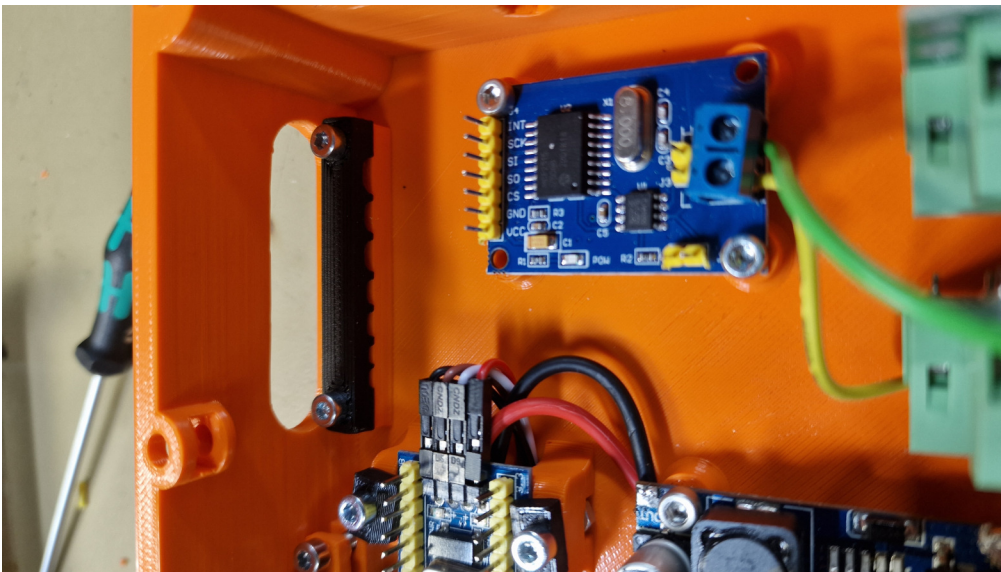
Cramp JST SM 2.54 female connectors on the outgoing programming wires. Insert a heat shrink cable over the cables.



Insert the female JST SM 2.54 crimped connectors into a male JS connector as seen below, it is important that the colour and placement are the same between ECU nodes.

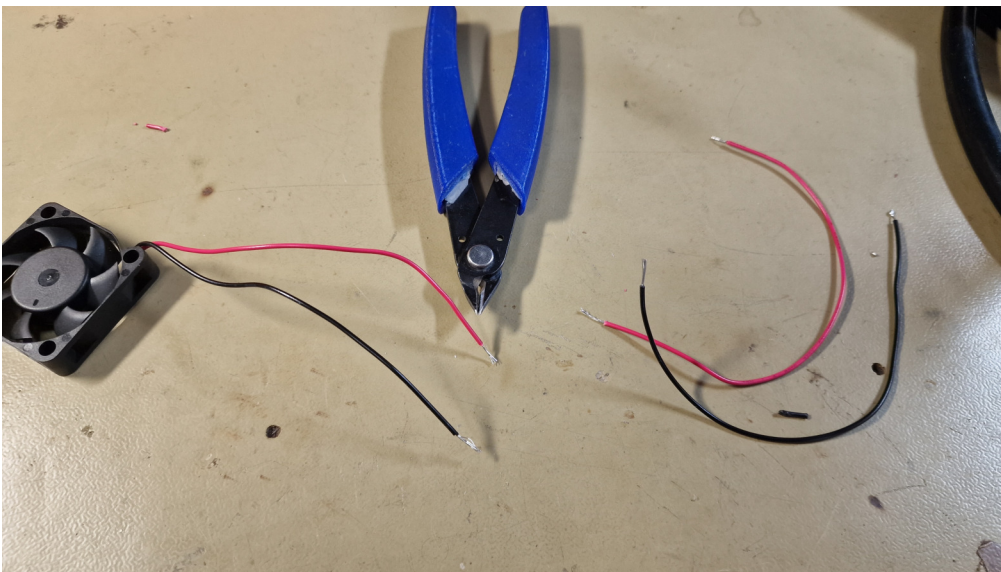


12. Screw down the outgoing cable hold-down (3D printed component). Use two M3x16mm screws.

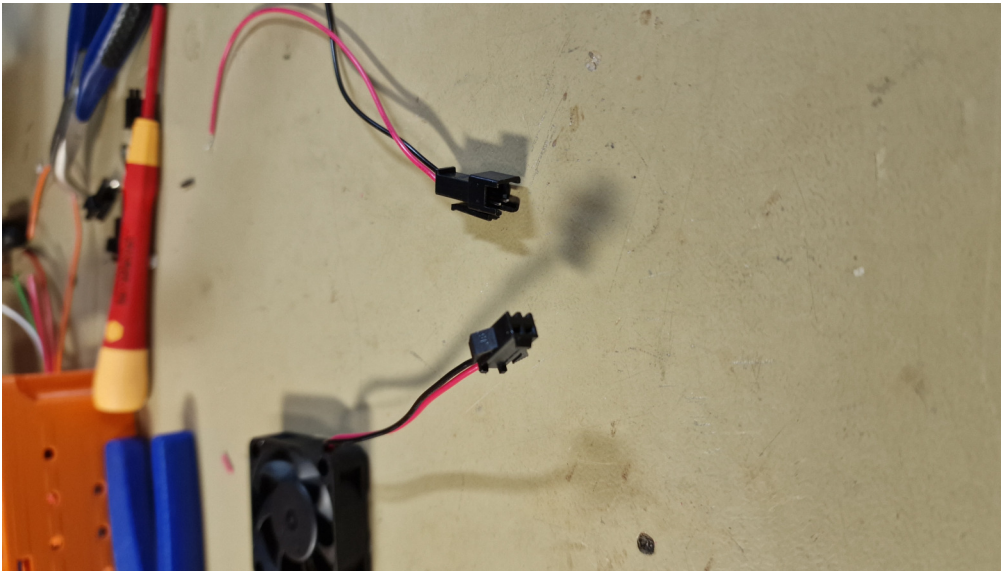


13. Installing 12v dc fan for cooling. A cooling fan should be screwed into the top cover of the ECU using 4 M3 screws and nuts.

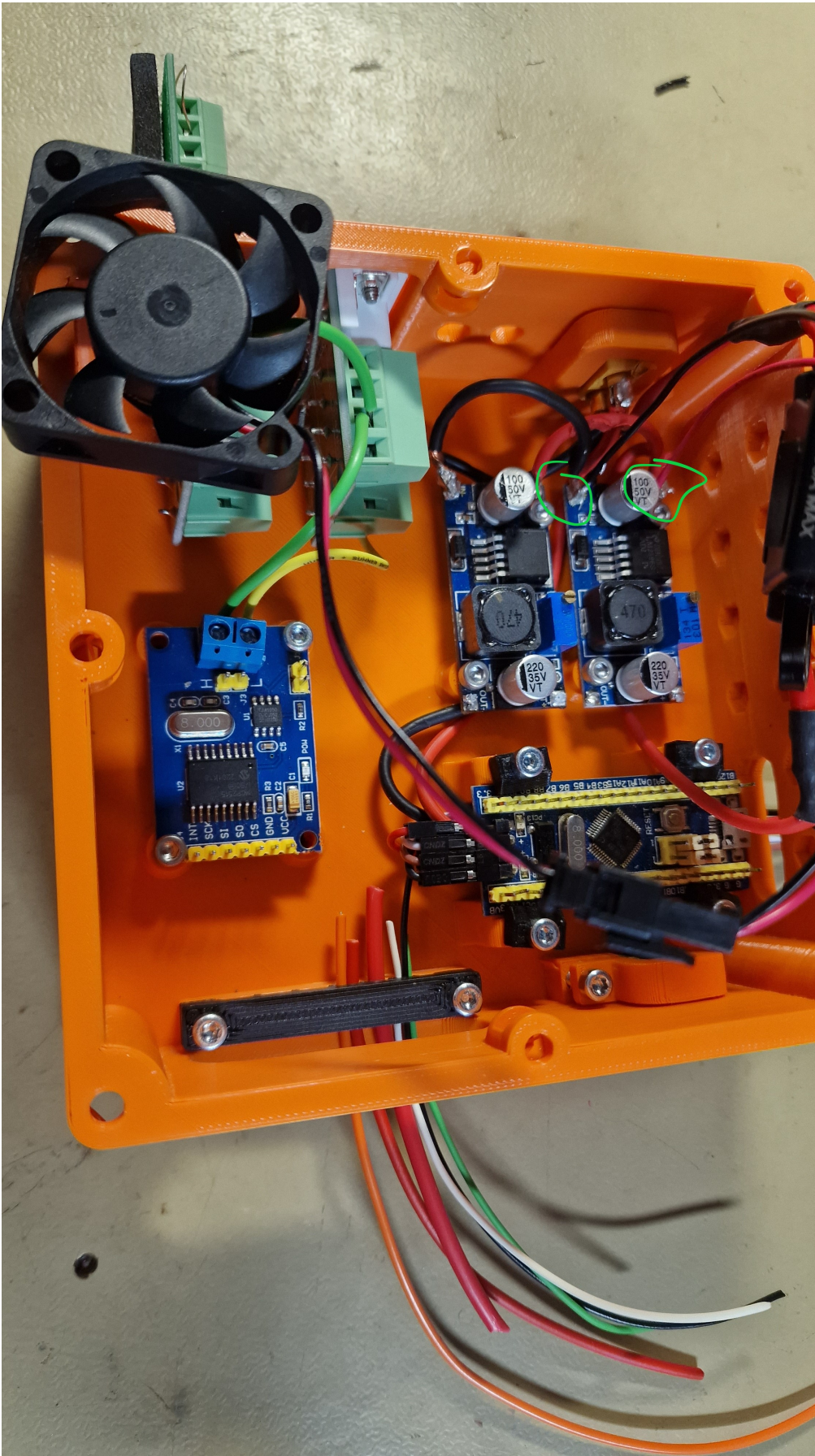
First, cut the fan cable in two pieces.



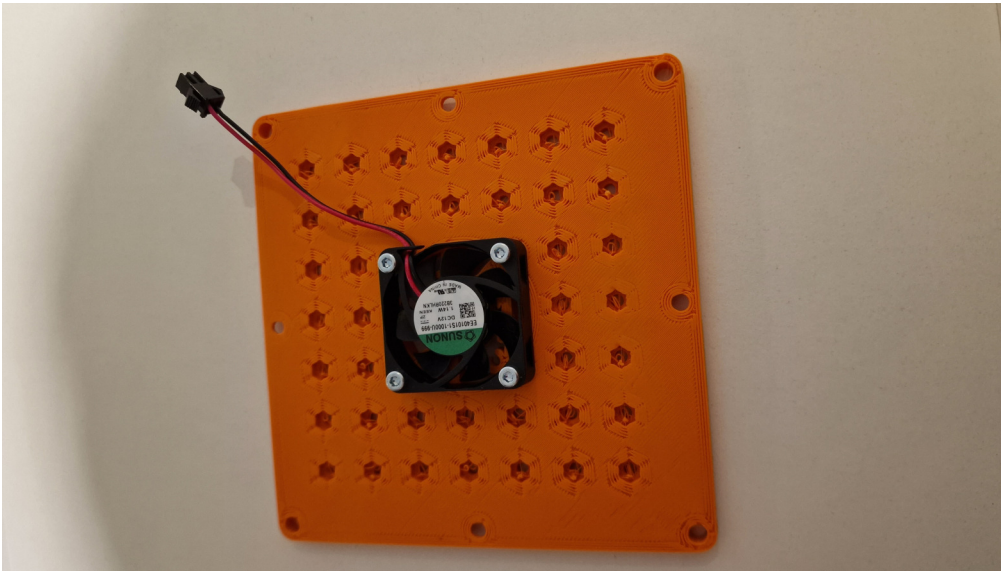
Connect the two cables together using JST SM 2.54 2 pin connector. This is to make the top cover completely removable if desired.



Solder the fan +12v and GND to the power in. (After the fuse). This can be done by connecting the wires to the dc-dc converters in terminals.



The fan shall then be mounted to the top cover as illustrated below. Using 4 M3x16mm screws and 4 M3 nuts.



14. Power up module

To verify that the previous steps have been done correctly, connect 12v to the XT60 12v in plug on the front.

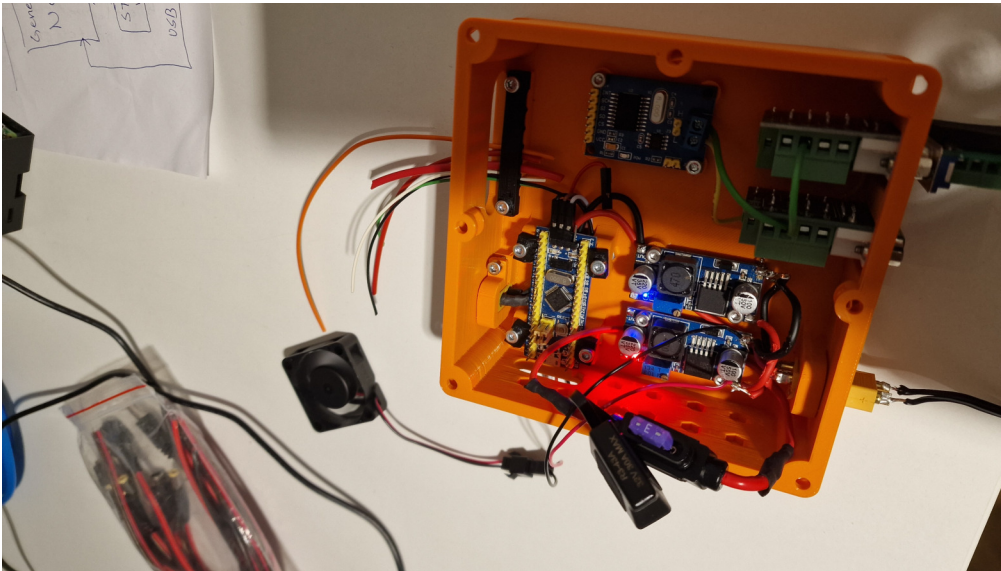
The expected result would be that the fan turns on and the dc-dc converters light up with a led shining.

The center dc-dc converter should output 3.3v and the dc-dc converter closest to the wall should output 5v.

If it does not power up correctly;

- Make sure you have inserted a 2A, 3A or 5A fuse into the fuse holder
- Check soldering connections using a multimeter

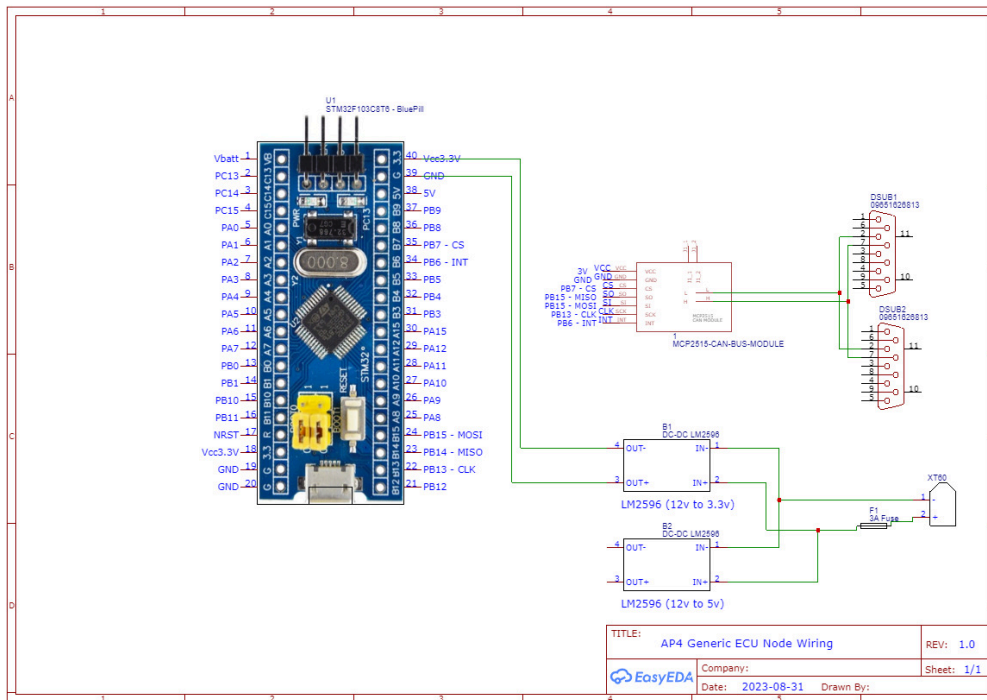
If dc-dc converter out voltage is not correct, adjust the blue potentiometers with a screwdriver until desired voltage is reached.g



15. Internal wiring of standard components. This can be done with either jumper wires or soldering new wires between internal components.

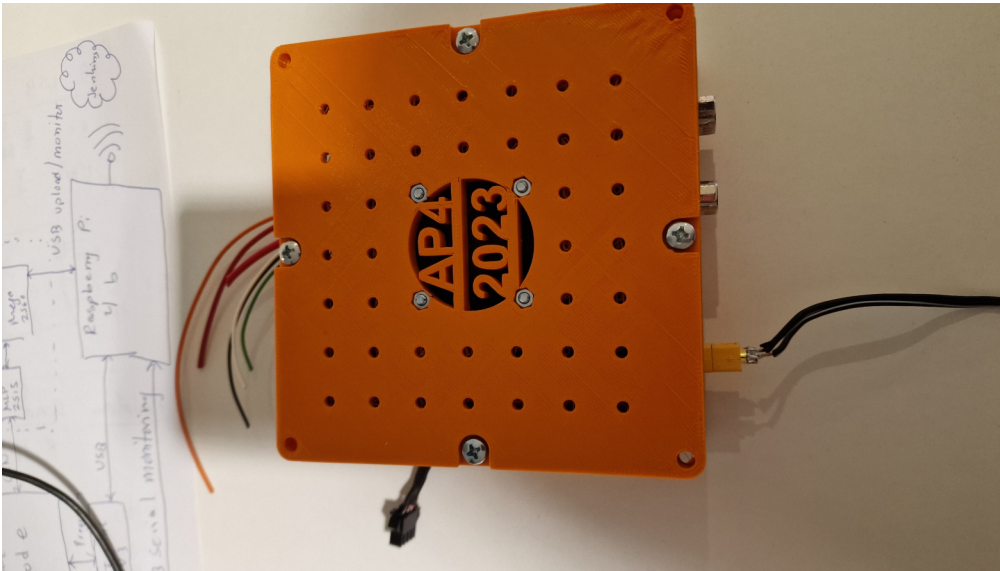
Here is the wiring diagram for a generic ECU HW node.

The internal wiring of a ECU can be seen bellow. This has to be setup in order to run software on the bluepill microcontroller and communicate over CAN bus.



16. Mount top cover onto ECU node.

The top cover can be mounted as pictured below using 4 M4x10mm screws and 4 M4 nuts.



17. Connect any needed embedded sensor(s)

Note down the extra wiring required and how you connected it and add to the node specific documentation.

18. Flash new software

How to flash new software to the STM32 bluepill can be found in the `HOW_TO_PROGRAM_A_BLUEPILL.md` document.

2.5.5 Extend with new function specific ECUs

These are the general guidelines and tips when adding a new function (or sensor) to a generic ECU base.

Questions to ask yourself;

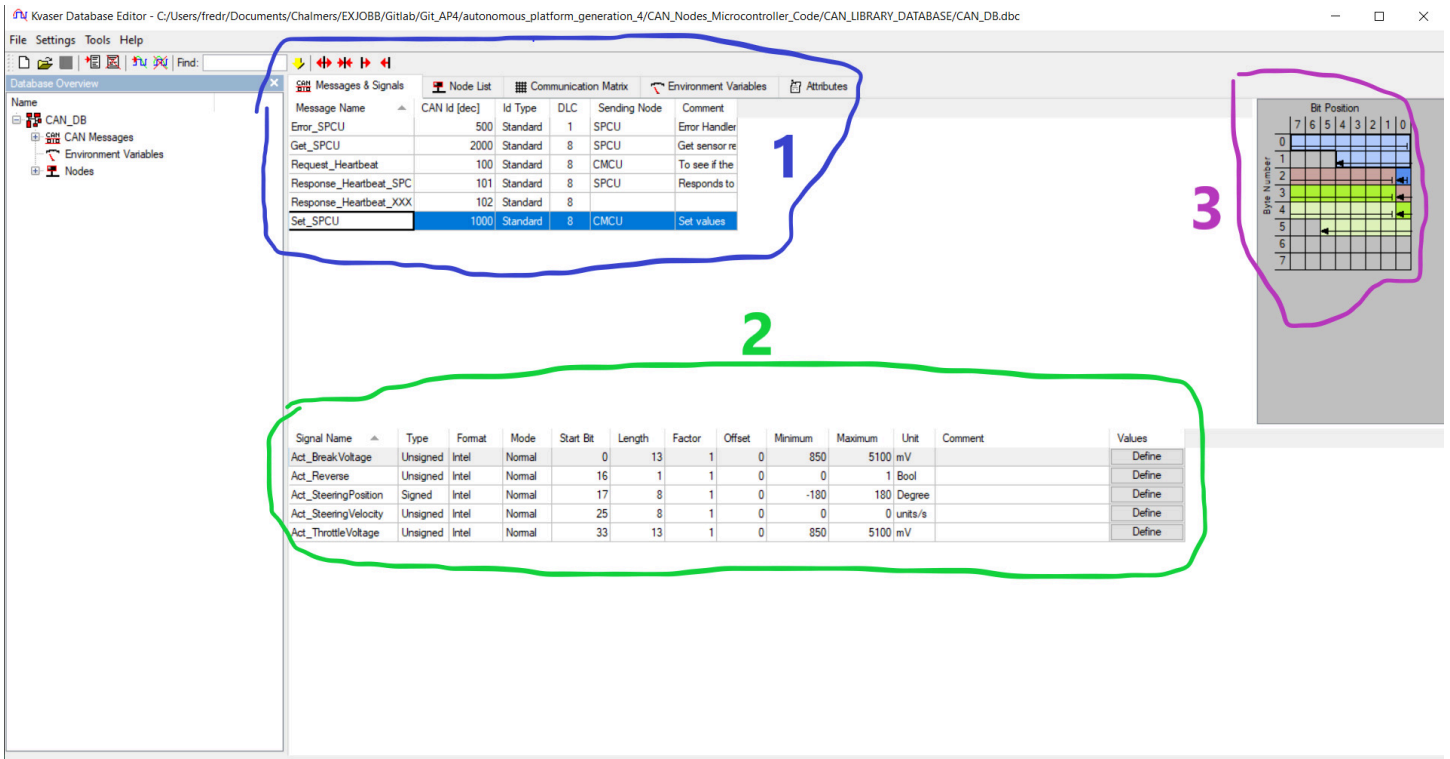
- How can the sensor be connected to a microcontroller? Does it require any communication protocol or can you use a digital or analog pin?
- What power requirements does it have? High power or low power? If the sensor/actuator requires more power (and or different voltage) from what the generic ECU can supply you need to take this into consideration
- Am I adding new functionality to an existing ECU or am I adding completely new functionality? Maybe it is not necessary to build a new ECU node from scratch
- What parts do I have in inventory?
- What parts do I have to buy?

Very briefly explained, here are the steps you should take to implement new functionality with embedded sensors or actuators.

- Add specific hardware modules
- Build a small test-rig with a microcontroller & breadboard to verify functionalities and software libraries needed.
- Build, solder and connect a generic ECU base (see SPCU). Interface the specific hardware module.
- Create a CAD model of the new sensor and how it should be mounted to the platform. (Make sure it fits the standardized hole pattern if it should be mounted to an ECU or aluminum sheet)
- If needed, update the CAN protocol by editing the dbc file and autogenerate new database in c-format.
- Copy the Template code for ECUs.
- Flash both the ECU and central master computer (update the CAN to ROS2 topic converter) with the latest CAN protocol. See `HOW_TO_EXTEND.md` in `Hardware_Interface_Low_Level_Computer` directory
- Add generic ECU code in the embedded software, follow the structure of the SPCU.
- When the embedded software of the ECU is verified, connect the to the rest of AP4 using a db9 Female-Female cable and a xt60 outlet to supply.

2.5.6 Unified CAN database and how to modify

The CAN database follows dbc format, [CAN and dbc guide](#). The dbc file can be edited using a dbc-editor an example of this is the [KVASER-DATABASE-EDITOR](#). When opening the dbc file for AP4 in kvaser, it will look like the figure below:



1. **Message Name** in the database. This is the frames defined for a CAN message. What id, Data length (bytes) etc.
2. **Signals** in each message. User defined regarding name, placement in the data-field, offsets, factor and so on.
3. **Graphical Illustration of datafield** for the selected message/frame in the database.

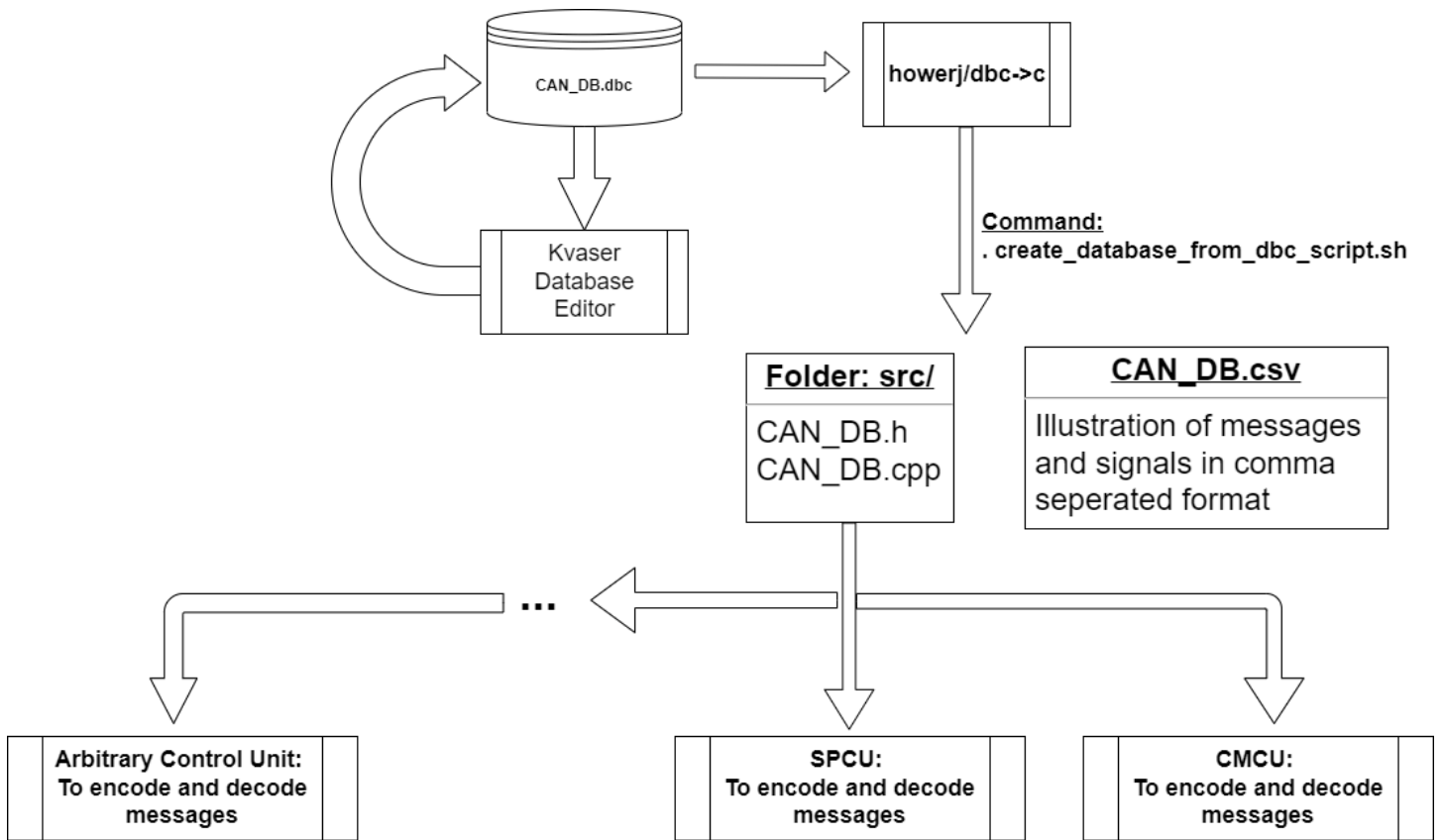
In order to make sense of the dbc files, the embedded software needs the information of the dbc file in a c format. Thus AP4 converts dbc file into c-functions and data types, in order to simply encode and decode. Using [howerj dbcc repo](#) to convert.



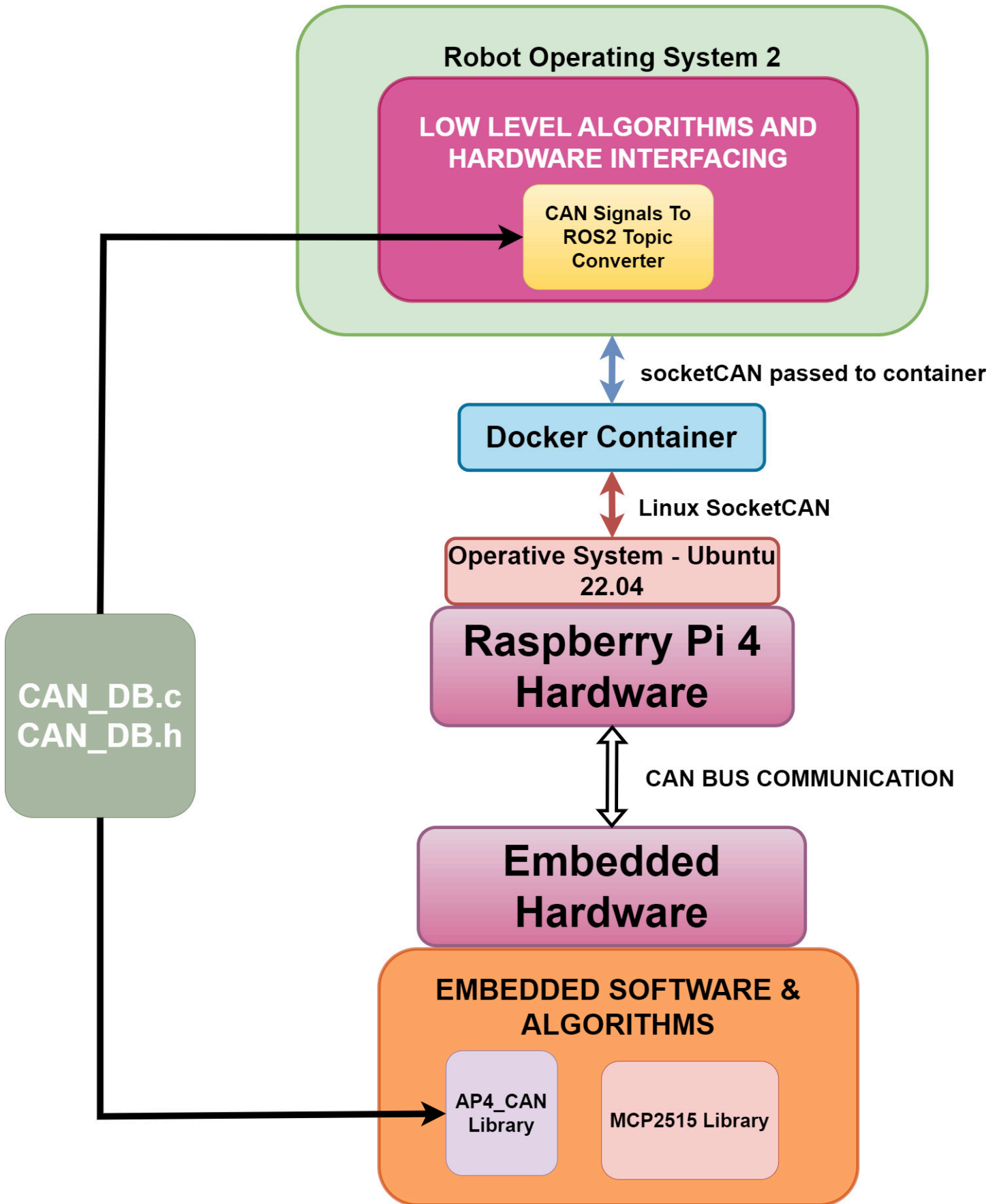
When the dbc is edited and saved, run the following command in a **Linux Machine** whilst located in path: CAN_Nodes_Microcontroller_Code\CAN_LIBRARY_DATABASE

```
. create_database_from_dbc_script.sh
```

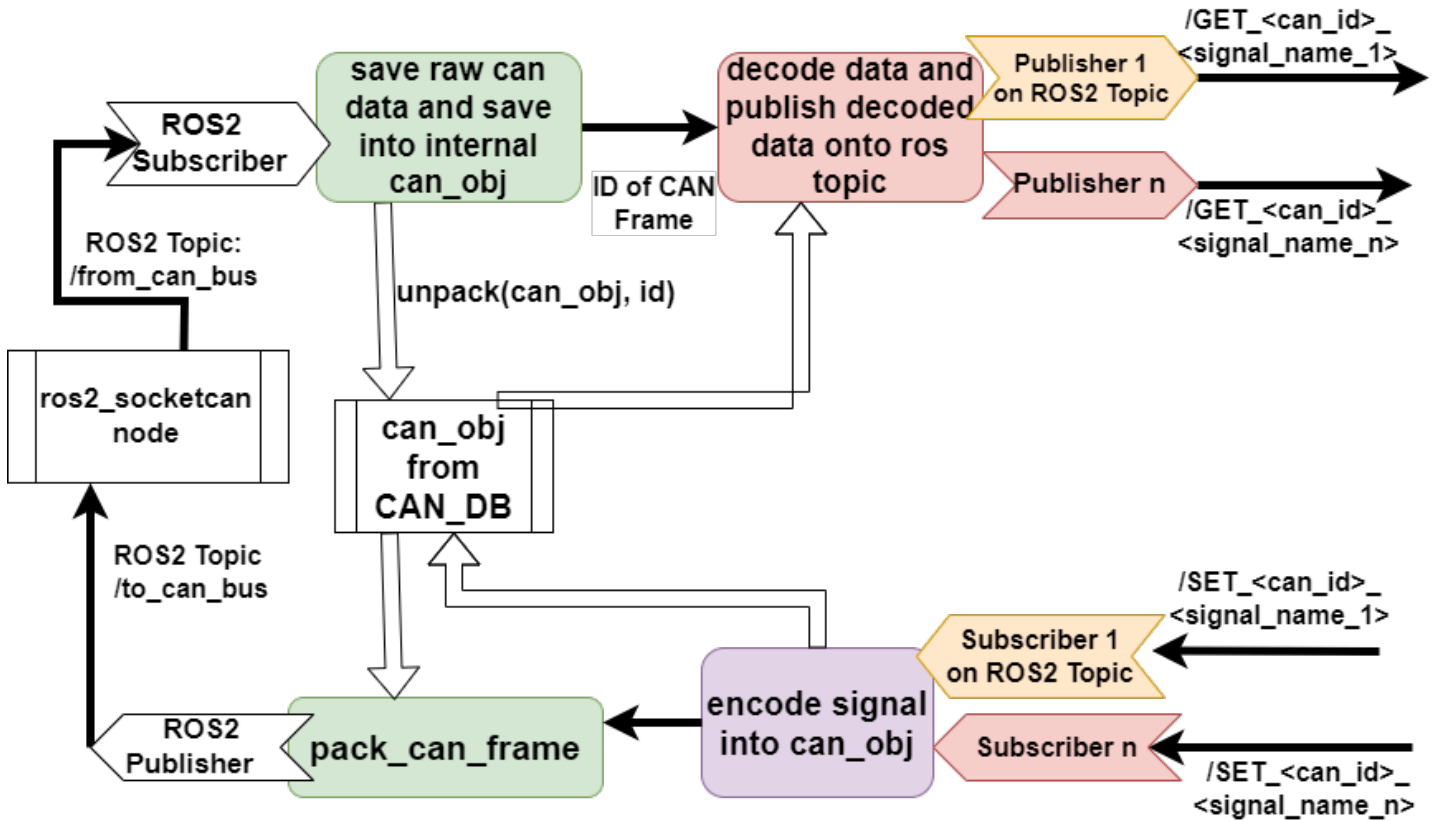
Down below is a simple flowchart of the way of working with the CAN protocols auto generation and dbc editing.



Down below can be seen how all the Software components refers to the same database meaning that a change in the CAN protocol at a single place will affect the whole system.



Also can be seen above how the central master computer, converts CAN messages to ROS2 topics to be later used in higher level algorithms. The *CAN signals To ROS2 Topic Converter* is illustrated in the figure below, implemented on the Raspberry Pi4b that acts as an interface in the central unit. The converter also converts topics to CAN frames that are then transmitted onto the CAN network.



2.5.7 Adding New CAN Frames And Signals

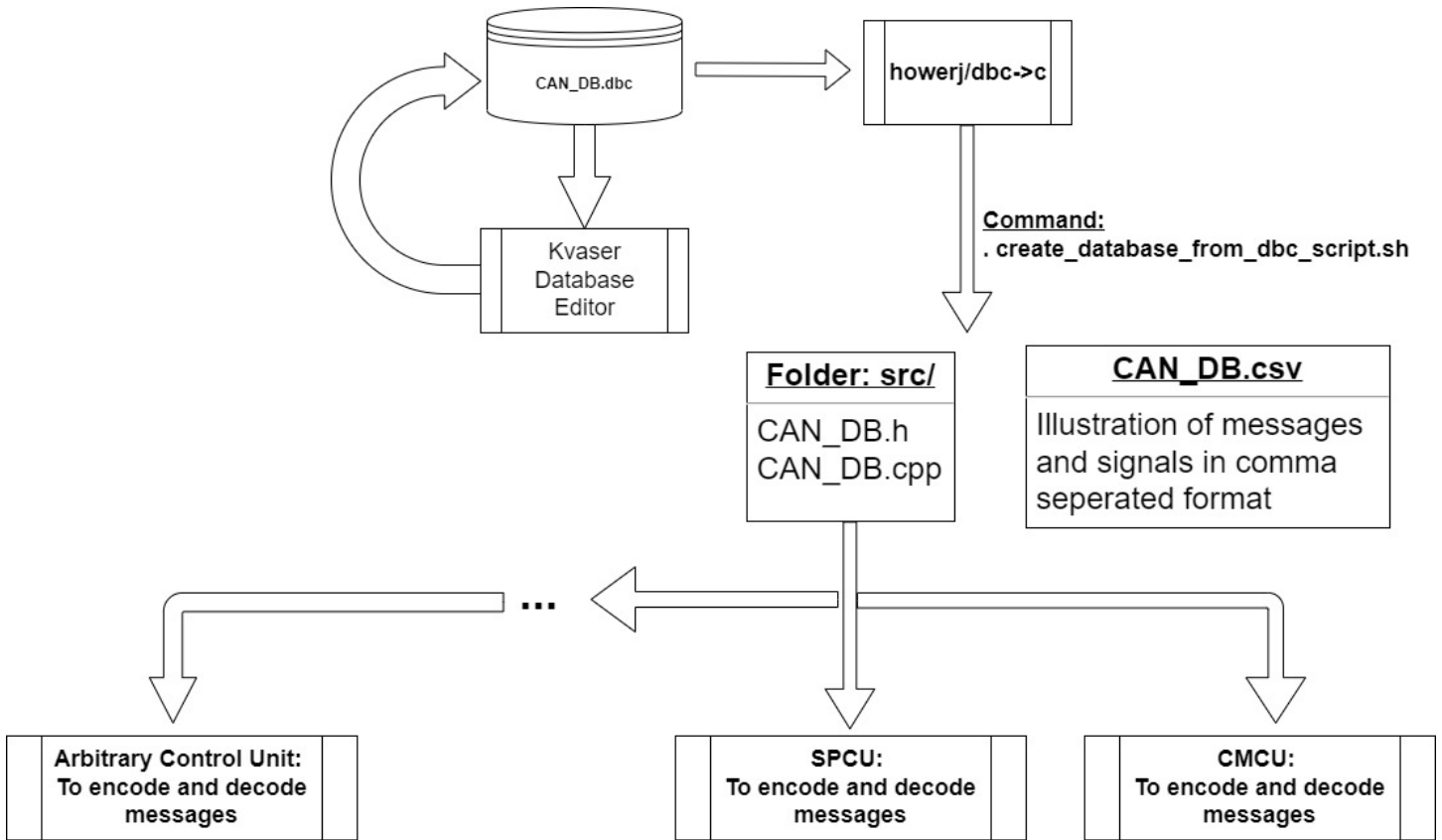
When adding new CAN frames and signals, changes have to be made in the “CAN_DB.dbc” file, new C code has to be generated as described in CAN_Nodes_Microcontroller_Code\CAN_LIBRARY_DATABASE\README.md. Lastly, the software in hardware interface and low level software has to be adjusted for the newly added frames. This is described in Hardware_Interface_Low_Level_Computer\HOW_TO_EXTEND.md.

When adding new CAN frames it is recommended to add new frames and signals instead of changing existing frames/signals. IF EXISTING CAN FRAMES/SIGNALS ARE CHANGED, EXISTING ECUS MUST BE REFLASHED WITH NEW SOFTWARE. If one appends new signals and frames to the database it is not strictly necessary to re-flash existing ECUs.

Adding new CAN frames and or signals is described in detail in CAN_Nodes_Microcontroller_Code\HOW_TO_EXTEND.md

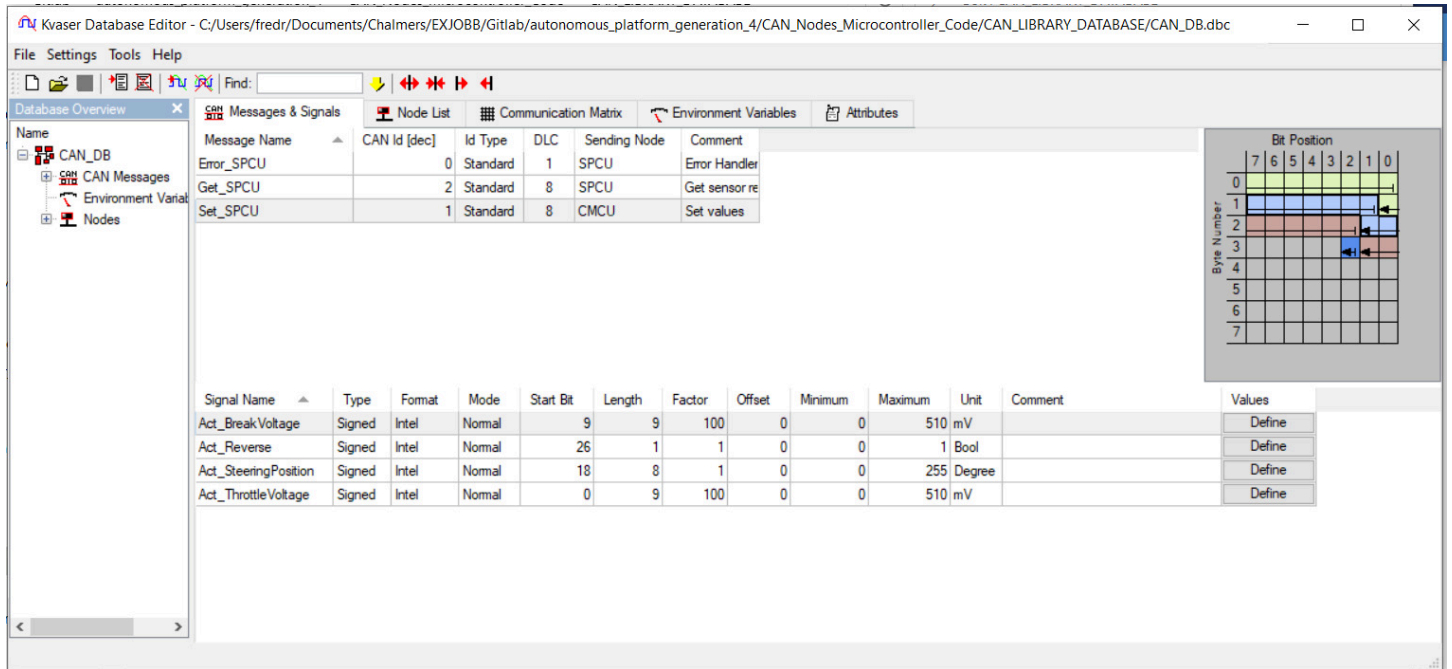
2.5.8 Controller Area Network Database

The way of working with CAN database for AP Gen 4 is illustrated below. Using a database in dbc format being parsed through the script from howerj repo (see below), that is producing a csv file and cpp file with functions to use to encode and decode messages. Thereafter it is included to all control units that is communicating over the CAN buss system.



2.5.9 dbc editor

There are several different softwares that can be used. One example is <https://kvaser-database-editor.software.informer.com/2.4/>, which is free to download. It is very intuitive to use and build up dbc files, an example can be seen below. Where the frames and signals is defined for the Steering, Propulsion control unit (SPCU). To the right the bit's position in the messages is illustrated, however one important note is that the editor don't check if some signals overwrite each other!



Below is an detailed description of the commands used.

2.5.10 How to use DBC to C library

The dbc to c (dbcc) repository is added as a submodule.

If the `dbc_to_c` folder does not show up as a directory, init and update repository submodules with

```
git submodule init
git submodule update
```

Link to repository for documentation: <https://github.com/howerj/dbcc>

Make sure the library is built using 'make' move into the `c_to_dbc` directory and write

```
make
```

If 'make' does not work, try running the following two commands first to make sure that it is installed.

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y make
```

The command 'make' also requires the GNU C compiler (`gcc`) to be installed. Run the following command to install the compiler/to confirm that it is installed.

```
sudo apt install gcc
```

If make has run properly, a new file called `dbcc`, should then appear inside the `dbc_to_c` folder.

Lastly, simply exit the `dbc_to_c` folder and run the `.sh` file as following:

```
./create_database_from_dbc_script.sh
```

For additional debugging:

- Parse `ex1.dbc` to a `.c` and `.h` file to a directory called `output_dir`:

```
./dbcc -v -o output_dir ex1.dbc
```

In `output_dir` two new files should appear, `ex1.c` and `ex1.h`

- Parse `ex1.dbc` to csv file to a directory called `output_dir`:

```
./dbcc -C -v -o output_dir ex1.dbc
```

A `ex1.csv` file should appear in the `output_dir`

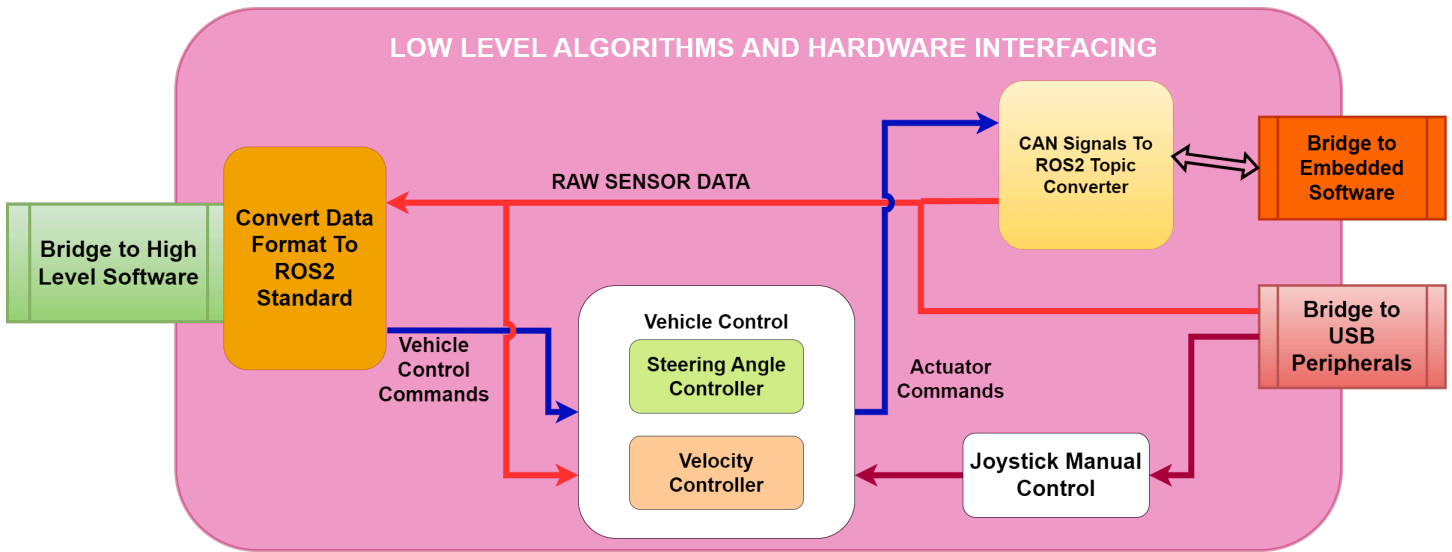
3 Low-level Control & Hardware

3.1 Introduction

This directory contains the software stack running on the Raspberry Pi 4b mounted to the hardware platform. It also aims to describe how each software component is designed / implemented.

In summary, the Low-level Control & Hardware Interface SW & HW is Hardware specific. It shall be run on the platform, connected to the embedded SW, CAN busses etc. It works as a bridge between high-level control software and the embedded ECU code which controls actuators / reads sensor data.

Below an illustration of the low level and hardware interfacing software can be seen.



The software itself is implemented using the ROS2 framework, this software is then run inside a docker container. It is only meant to run on the Raspberry Pi 4b mounted onto autonomous platform and is very hardware specific.

In the diagram above, four ROS2 packages are illustrated as rounded rectangles. These are;

3.2 CAN Signals To ROS2 Topic Converter

This is a custom made ROS2 package made spring 2023. The purpose of it is to be a software translation of the CAN bus data from the embedded software layer to useful information available on ROS2 topics.

This abstracts the transfer of information between low level software and embedded software running on ECUs in the CAN network.

It uses the unified CAN bus database file (.dbc) for autonomous platform in order to decode the information sent on the CAN network. This database file is located at `autonomous_platform\CAN_Nodes_Microcontroller_Code\CAN_LIBRARY_DATABASE\CAN_DB.dbc`. A custom application is then written that maps a specific CAN signal onto a specific ROS2 topic. This means that every data signal sent over the CAN network can be used in algorithms located in the hardware interface and low level software. In the same way, ROS2 topics can be used to set the value of certain signals, meaning commands can be sent from the hardware interface and low level software to the various ECUs mounted onto autonomous platform.

3.2.1 Vehicle Control

This is a custom made ROS2 package to control the steering angle and velocity of the hardware on a low level. It receives a desired velocity and steering angle on the `/cmd_vel` ROS2 topic and uses a controller gain to scale the values. The values have to be processed and rescaled in order for the embedded ECUs to actuate correctly.

I.e to control the velocity, the embedded software in the SPCU expects a desired voltage to be sent on the CAN bus. Therefore the desired velocity needs to be converted into a desired voltage that should be applied on the accelerator pedal.

More advanced control behavior can be implemented later on, I.e some sort of feedback from the velocity sensors. As of now it is open loop controlled

3.2.2 Joystick Manual Control

This is a set of ROS2 packages which enables a joystick to be connected to autonomous platform. The output will be in the form of a velocity commands on the `/cmd_vel` ROS2 topic. This output can then be read in vehicle control to control the gokart.

3.2.3 Convert Data Format To ROS2 Standard

This is not implemented yet for every data input

The software layer connected to the hardware interfacing software layer expects input and output to be on specific ROS2 topics and uses standard ROS2 interfaces. I.e velocity commands to the platform will come to the hardware interfacing and low level software on the topic `/cmd_vel`, it is then up to this ROS2 package to convert it to a format which the lower level algorithms can use.

This can be as an abstraction layer.

The rectangular squares in the diagram above can be seen as input and outputs to the hardware interface and low level software layer. Higher level commands to the platform will come from the high level software. In the same way, this software layer will interface with the CAN network and USB peripherals for input and output.

3.2.4 Low-Level Control & Hardware Interface - Software / Hardware Requirements

In terms of hardware, the low-level control software and hardware interfacing software is run on a Raspberry Pi 4b 4Gb. A RS485 CAN hat has been installed onto the GPIO pins. Power (5V,3A) is supplied from the platform using an LM2596 DC-DC converter to step down the platforms 12V power supply to 5V. A 32GB micro SD card is all the storage available. The hardware should be connected to the CAN network on Autonomous Platform Generation 4 (DB9 Connector) and to the wifi router/switch through an ethernet cable.

- [Raspberry Pi 4b Specifications](#)
- [RS485 CAN HAT Specifications](#)
- [LM2596 Specifications](#)

Software wise, it uses a base installation of Ubuntu 22.04. A detailed guide on how to setup the software on a new fresh Raspberry Pi 4b can be found [here](#).

When developing code for the low-level software it is possible to open the low-level software docker container on any linux based computer. It can be useful to have access to a proper IDE and not have to write code in notepad. Note: the CAN bus interfacing libraries will throw errors due to not being connected hardware wise. But once software compiles changes can be pushed and then pulled to the raspberry pi 4b.

3.2.5 USB Devices Connected

There exists a USB hub mounted to the autonomous platform. This is connected to the Raspberry Pi 4b.

The following devices are connected * Wireless keyboard and mouse * Xbox Wireless Adapter

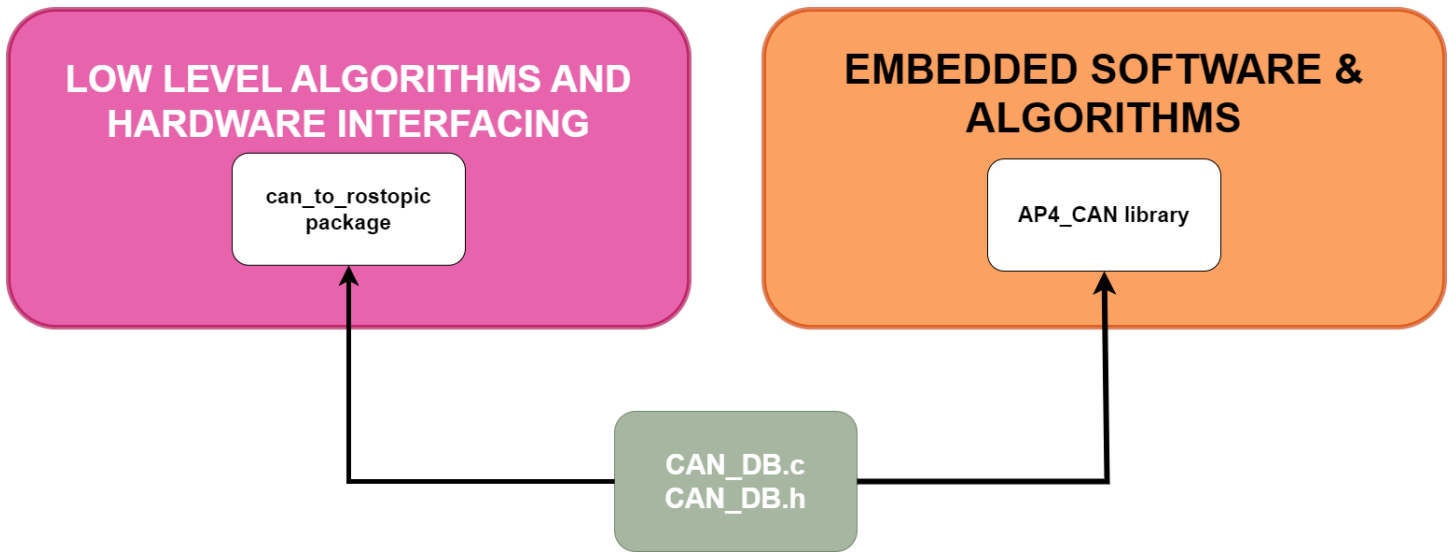
Currently the USB hub is not mounted

3.2.6 Interfacing With Embedded ECUs

The hardware interface and low level software can communicate with the embedded ECUs. Physically this is done through a Controller Area Network (CAN) bus with a D-Sub 9 connector. Software wise, a custom solution has been implemented for autonomous platform. The data sent over the CAN bus needs to be available on ROS2 topics in order for ROS2 nodes to be able to use the data in computations. In the same way, actuator commands sent from ROS2 nodes on ROS2 topics need to be sent to the ECUs over CAN.

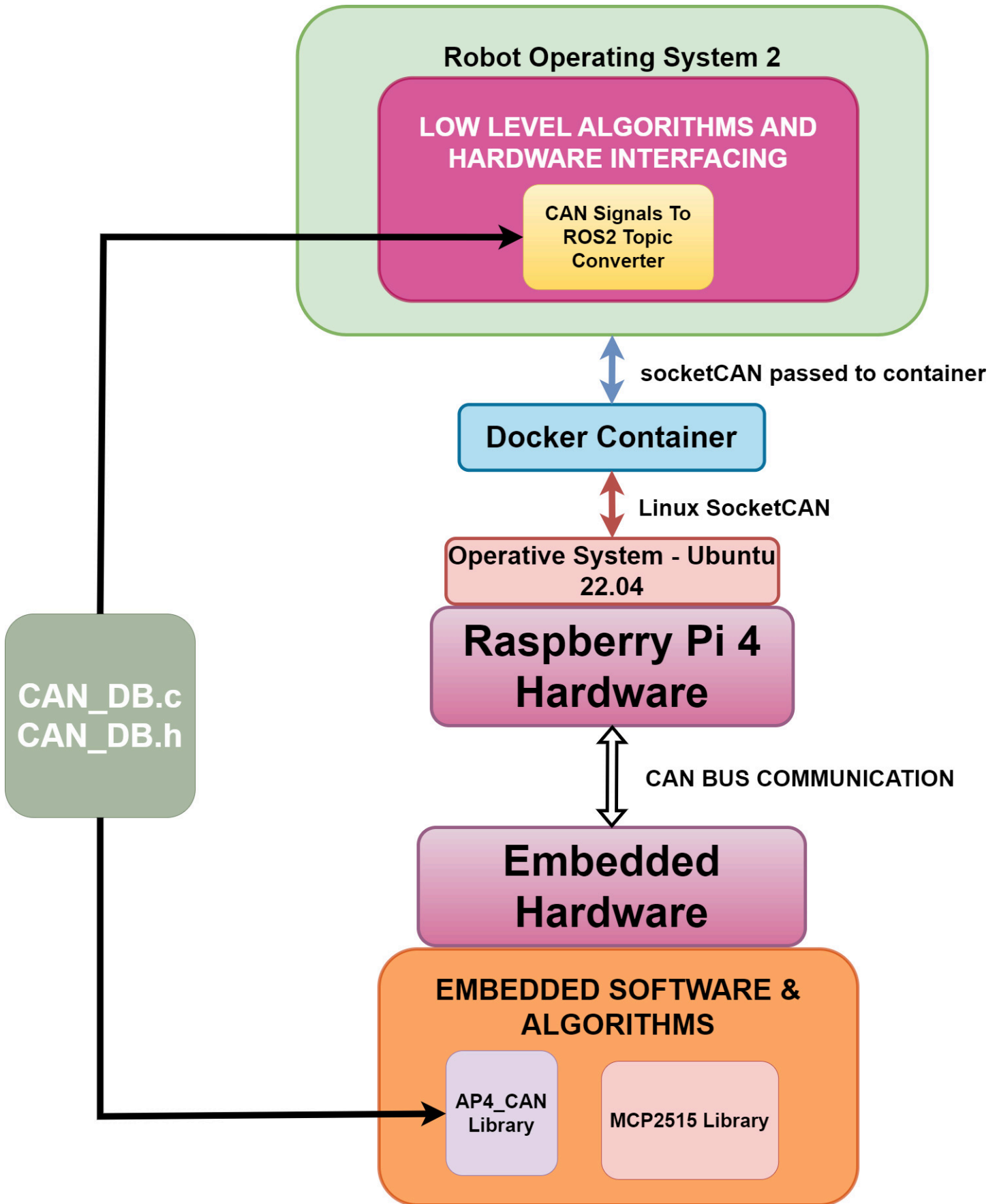
The CAN frames sent over the CAN network must be encoded and decoded correctly and in the same way on both the hardware interface and embedded ECUs. Therefore a unified CAN database file has been constructed for autonomous platform . A helper library is used to convert the CAN database file into a set of C-style data structures.





Two custom made libraries have then been implemented, one in the embedded software and one in ROS2 in hardware interface and low level software.

How to extend this functionality with new CAN frames and signals is explained in detail in `Hardware_Interface_Low_Level_Computer\HOW_TO_EXTEND.md` (The parts that are relevant to change in hardware interface and low level software is documented here). How to extended the software for the embedded ECUs is described in `CAN_Nodes_Microcontroller_Code\HOW_TO_EXTEND.md`.



3.2.7 Interfacing with High Level Control Software

The hardware interface and low level software can communicate with the high level control software running on an external computer. (I.e linux laptop- or intel NUC) This is done through an ethernet communication. A Wifi router is mounted to

autonomous platform which means this can be done wirelessly.

Once an ethernet connection has been established, the two distinct ROS2 software networks (one on raspberry pi and one on laptop) will be able to detect each other. This means that the two smaller ROS2 networks of nodes will become one large network with distributed computing power.

Network SSID and password is noted down in main `README.md` file of repository.

An important note, the `ROS_DOMAIN_ID` has to be configured to the value of 1 on both ROS2 networks.

3.2.8 Adding New Functionalities

How to add new functionality to the hardware interfacing software and low level algorithms is explained in `Hardware_Interface_Low_Level_Computer\HOW_TO_EXTEND.md`.

3.2.9 Automatic Startup of Software

Once the Raspberry Pi 4b boots up it will automatically start the hardware interfacing and low level algorithm software. This is done using linux services that run on boot-up which in turn call the script “testing.bash”.

If you need to change what is done during boot-up, edit the `testing.bash` script.

If the software does not automatically start, see **Software container not started?** in `TEST_DEBUGGING.md` located in this directory.

3.2.10 Installing Base Software on Fresh Raspberry Pi 4b

See section in `SETUP_OF_RASPBERRY_PI.md`

3.3 Extend Hardware Interface and Low Level Software

This document aims to describe the process of how to extend the hardware interface and low level software.

Before starting to develop and adding code to the low level control software you need to first make sure you need to add something here.

If you;

- Want to add a new sensor to autonomous platform
- Want to process sensor information
- Want to design hardware specific interfaces (I.e specific to the electric gokart)

Then you are in the right spot!!

If not, take a look at `High_Level_Control_Computer` or `CAN_Nodes_Microcontroller_Code`, maybe you intended to add functionality there!

3.3.1 Prerequisites

In order to start adding functionality it is recommended to have a basic understanding of:

- C++ OR Python development
- docker containers (How to start, stop, restart and configure)
- Linux - The container software environment is mainly navigated in through a terminal
- Robot Operating System 2
- CAN bus

Software wise, you need to have the following installed:

- docker
- git
- VSCode (recommended but any IDE may be suitable)

Hardware wise, it is recommended you have:

- Linux based x86 host computer

3.3.2 How to Add New Functionality

First of all make sure you have read the general design principles document for autonomous platform located at `autonomous_platform/HOW_TO_EXTEND.md`. This document takes precedence over anything written in this document in order to unify the development process across all software layers.

This section is split up into two parts; * Software Functionality * Sensors * USB Based Sensors * Embedded Microcontroller Sensors

3.3.3 Software functionality

If the sensor information already exists and you need to add new software functionality the general implementation steps are:

3.3.4 Sensors

Sensors can be split up into two groups, sensors with a USB interface and sensors that require an embedded (microcontroller) interface. Integration of sensors that require them to be connected to the embedded ECU. How to add these on the ECU level is described in `CAN_Nodes_Microcontroller_Code/HOW_TO_EXTEND.md`.

3.3.5 USB Based Sensors

If a sensors with a USB interface is to be added to autonomous platform then this can be done solely in `Hardware_Interface_Low_Level_Computer` without touching the embedded software. The general procedure would be.

1. Mount the sensor hardware on platform
2. Connect USB to AP4 USB hub.
3. Make sure new device is detected
4. Find suitable existing ROS2 package that fits the given sensor. (Hint: Google)
5. Add ROS2 package installation command into dockerfile
6. Manually start the ROS2 package and see if sensor data is outputted as expected (Hint: `ros2 run ${package name} ${node name}` OR `ros2 launch ${package name} ${launch file name}`). Look into package document to see what is available.
7. Configure parameters. Some packages require software to start with certain parameters, if so create a new ROS2 package and save the configuration parameters in this.
8. Configure package dependencies in order to access the downloaded package API
9. Create a launch file, in the newly created ROS2 package, add a launch directory and create a launch file to start the required ROS2 nodes for the USB sensor.
10. Add a `documentation_and_research` folder in the ROS2 package. Write down the steps taken and purpose with package. Document what you have done, if it works as expected etc...
11. Call the newly created launch file for the USB sensor in the main launch file for hardware platform `autonomous_platform\Hardware_Interface_Low_Level_Computer\ap4_hwi_code\ap4hwi_ws\src\launch_hwi_software_pkg\launch\lau`
12. Restart platform, everything should start as configured and new sensor information should be available on a topic.

3.3.6 Embedded Microcontroller Sensors

Once these sensors have been implemented in the ECUs as embedded code and transmitted over the CAN bus network then some development needs to be done in the `Hardware_Interface_Low_Level_Computer` code in order to decode the CAN frame and make the sensor information available on a ROS2 topic.

Once raw CAN bus messages are sent over the CAN bus network and received in the hardware interface low level computer the messages needs to be processed. The general procedure for adding new CAN signals can be seen below:

3.3.6.1 Before you begin Make sure that that the CAN bus message you want to decode is received over the CAN bus from an embedded ECU. On the Raspberry Pi 4b, perform a candump and analyze it. In a new terminal:

```
candump can0
```

The expected output would be similar to:

```
can0 7D0 [8] 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00
```

How to analyze; The candump is structured as follows. Each row is a separate CAN frame message consisting.

```
<can socket id in HEX format> <Frame ID> <length of CAN frame in Bytes> <Value of bytes in HEX>
```

Meaning;

- can0 : On which CAN socket the CAN bus frame was read from
- 7D0 : The Frame ID (specified in the CAN dbc file)
- [8] : Length of CAN frame.
- 00 00 00 00 00 00 00 00 : The data sent over CAN bus in HEX

If you see your frame ID that you just added in the embedded software you can continue. If not, go back and look into your newly developed embedded software. Make sure the data is sent as expected, either periodically or when a certain event is triggered. It is not trivial to read and understand the raw hex data, hence we need to programmatically decode this in order to use the data later on.

3.3.6.2 General procedure A unified standard for the CAN Frames sent across the platform makes the flow of information more intuitive, signal names will have the same name in embedded SW as in ROS2 SW.

When adding a new CAN bus Frame/Message onto AP4 hardware interface there are a few steps to take:

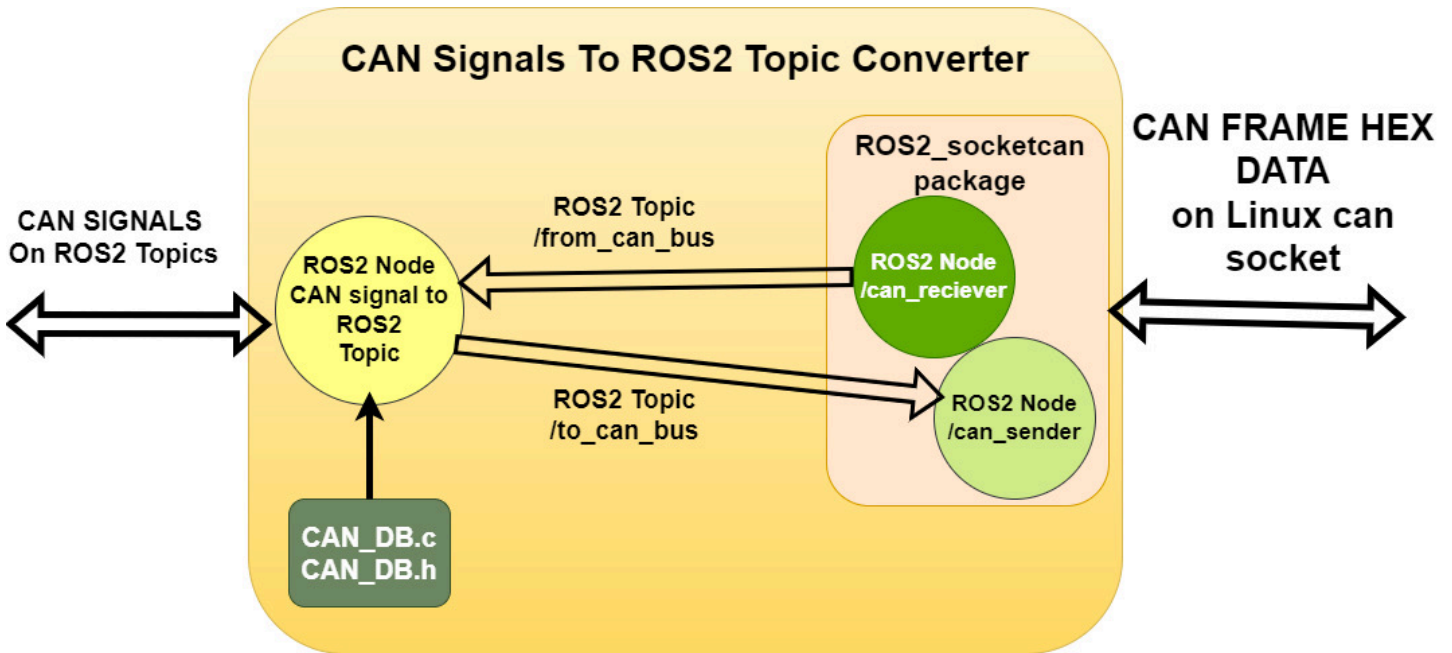
HE: ??? 4. **Update CAN HW Signal to ROS2 TOPIC converter** with new signals.

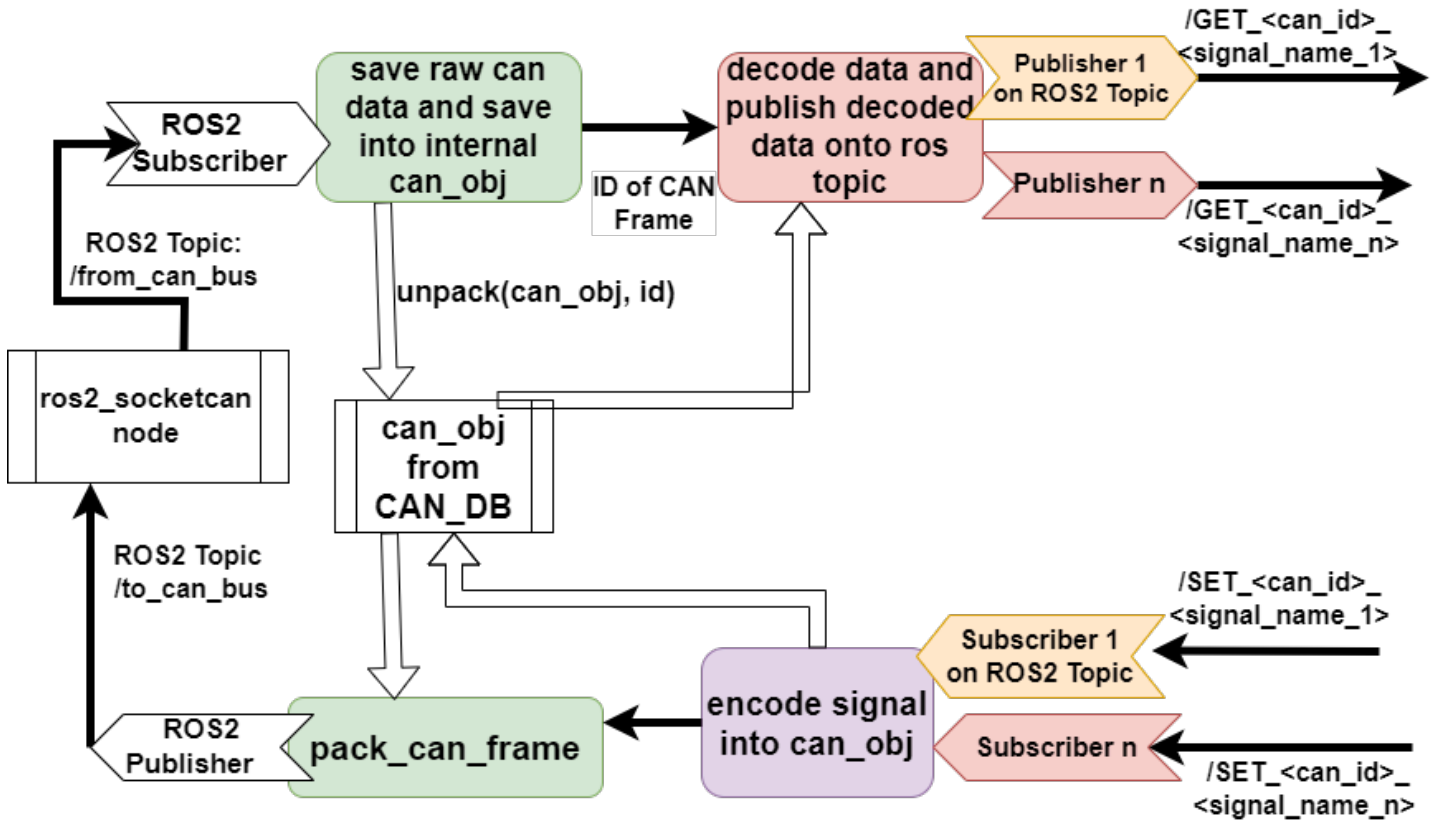
Meaning, update the code in `can_msgs_to_ros2_topic_pkg`. Located at:

Hardware_Interface_Low_Level_Computer\ap4_hwi_code\ap4hwi_ws\src\can_msgs_to_ros2_topic_pkg\src\can_msgs_translator_interf

To describe what does software does in short, this package spawns one ROS2 node which listens to / publishes to, two topics `/from_can_bus` and `/to_can_bus`. These two topics come from the `ros2_socketcan` ROS2 library.

An illustration for the behavior can be seen in the two figures below:





So therefore, for every CAN signal, there has to be a ROS2 publisher and subscriber.

For every **NEW** CAN signal added in the CAN database file there are five actions which must be taken inside this source file. (can_msgs_translator_interface.cpp)

NOTE: Signal type is defined in DBC file, choose a suitable type in ROS2. (i.e if Signal in CAN dbc is defined as 8 bit integer, choose Int8 in ROS2). A reference for data type conversion standard between C++ and ROS can be found [here \(2.1.1 Fieldtypes\)](#).

These ROS2 data types have to be included as message type headers in the top of the file before being able to be used. Example:

```
#include "std_msgs/msg/string.hpp"
#include "std_msgs/msg/float64.hpp"
#include "std_msgs/msg/int8.hpp"
#include "std_msgs/msg/int16.hpp"
#include "std_msgs/msg/u_int8.hpp"
#include "std_msgs/msg/u_int16.hpp"
...
```

Below:

- < FRAME ID > should be replaced with the frame ID specified in Kvaser in base 10.
- < SIGNAL NAME > should be replaced with the signal name outputted into the DBC file from KVASER

The five steps are:

1. Add publisher and subscriber object definitions. (private variables)

Define the class object variables, publisher and subscribers, for EVERY new CAN signal added. Frame ID should be the CAN frame ID set in KVASER, and signal name should be the generated name in CAN_DB file.

```
// Frame ID XXX publisher
rclcpp::Publisher<std_msgs::msg::UInt16>::SharedPtr publisher_frame_<FRAME ID>_<SIGNAL NAME>;
```

```
// Frame ID XXX subscriber
rclcpp::Subscription<std_msgs::msg::UInt16>::SharedPtr subscriber_frame_<FRAME ID>_<SIGNAL NAME>;
```

2. Init publisher and subscriber objects inside class constructor TranslatorCANtoROS2Node()

```
// Frame ID XXX publisher
publisher_frame_<FRAME ID>_<SIGNAL NAME>_ = this->create_publisher<std_msgs::msg::UInt16>("/GET_<FRAME ID>_<SIGNAL NAME>_");
```

Each subscriber will have a callback function, when a new ROS2 message is received on a topic and read by the Node, it will call this function to process the message sent over the topic. This definition will come later, as for now, follow the standard and add a callback function according to the naming standard.

```
// Frame ID XXX subscriber
subscriber_frame_<FRAME ID>_<SIGNAL NAME> = this->create_subscription<std_msgs::msg::UInt16>("/SET_<FRAME ID>_<SIGNAL NAME>_");
```

3. Add to PublishIncomingDataOnRosTopics()

The switch case should be appended with a new case for every NEW FRAME added. The general idea is to publish every SIGNAL inside a frame onto a unique ROS2 topic, using the publisher objects created above. See the code for inspiration. The structure is the same for every frame. Remember to do this for every SIGNAL in a FRAME.

```
switch(frame_id)
{
    case CAN_ID_REQUEST_HEARTBEAT:
    {
        /*
        publish every can signal contained in frame with id = 100
        */

        /*
        General procedure for each signal in can frame
        1: decode data into variable
        2: publish data
        */

        uint64_t temp_data;
        decode_can_0x064_Sig_Req_Heartbeat(&can_storage_container, &temp_data);
        std_msgs::msg::Float64 send_data;
        send_data.data = temp_data;
        publisher_frame_100_Req_Heartbeat_->publish(send_data);

        break;
    }

    case NEW_FRAME_ID:
    {
        ...
        break;
    }
}
```

4. Create CALLBACK functions for every ROS2 subscriber.

Example:

```
// Callbacks for signals in Frame with ID = 1000
void Callback_frame_1000_Act_BreakVoltage(const std_msgs::msg::UInt16 msg)
{
    encode_can_0x3e8_Act_BreakVoltage(&can_storage_container, msg.data);
    PublishCanFrameToCanNetwork(CAN_ID_SET_SPCU);
}
```

```
// Callbacks for signals in Frame with ID = 1000
void Callback_<FRAME ID>_<SIGNAL NAME>(const std_msgs::msg::UInt16 msg)
{
    encode_can_<FRAME ID>_<SIGNAL NAME>(&can_storage_container, msg.data);
    PublishCanFrameToCanNetwork(<FRAME ID>);
}
```

For every new subscriber created in 1. Create a callback function with the general logic described above.

5. Build the ROS2 Software package.

To check that everything has been setup, try to compile the software. (whilst inside the docker container environment)

- Navigate to ap4hwi_ws directory
- source environment variables

```
source /opt/ros/humble/setup.bash
```

Build the ROS2 package

```
colcon build
```

Expected output; package built successfully, no errors.

If nothing else has been changed, push changes to git and pull on the Raspberry Pi 4 on the platform. Restart platform and system should be up and running.

3.3.6.3 How to verify that CAN messages are read and translates properly to a ROS2 topic Make sure that the platform has the latest code version and that the packages can be built inside the docker container running on the Raspberry Pi 4b.

1. Start up the physical platform
2. Wait for system to boot up
3. Connect display to Raspberry Pi 4b
4. Enter the docker container

```
docker exec -it ap4hwi bash
```

Container NOT started? Look in TEST_DEBUGGING.md for solutions.

5. Enter correct directory and source environment variables

```
cd ap4hwi_ws
source install/setup.bash
ROS_DOMAIN_ID = 1
```

6. Look at what ROS2 topics are available

```
ros2 topic list
```

Are there no topics available? Look in TEST_DEBUGGING.md for solutions.

In the terminal your newly added ROS2 topic should show up (along with many other topics)

```
...
/GET_<FRAME_ID>_<SIGNAL_NAME>
/SET_<FRAME_ID>_<SIGNAL_NAME>
...
```

The information sent over the newly topic can be listen to in the terminal

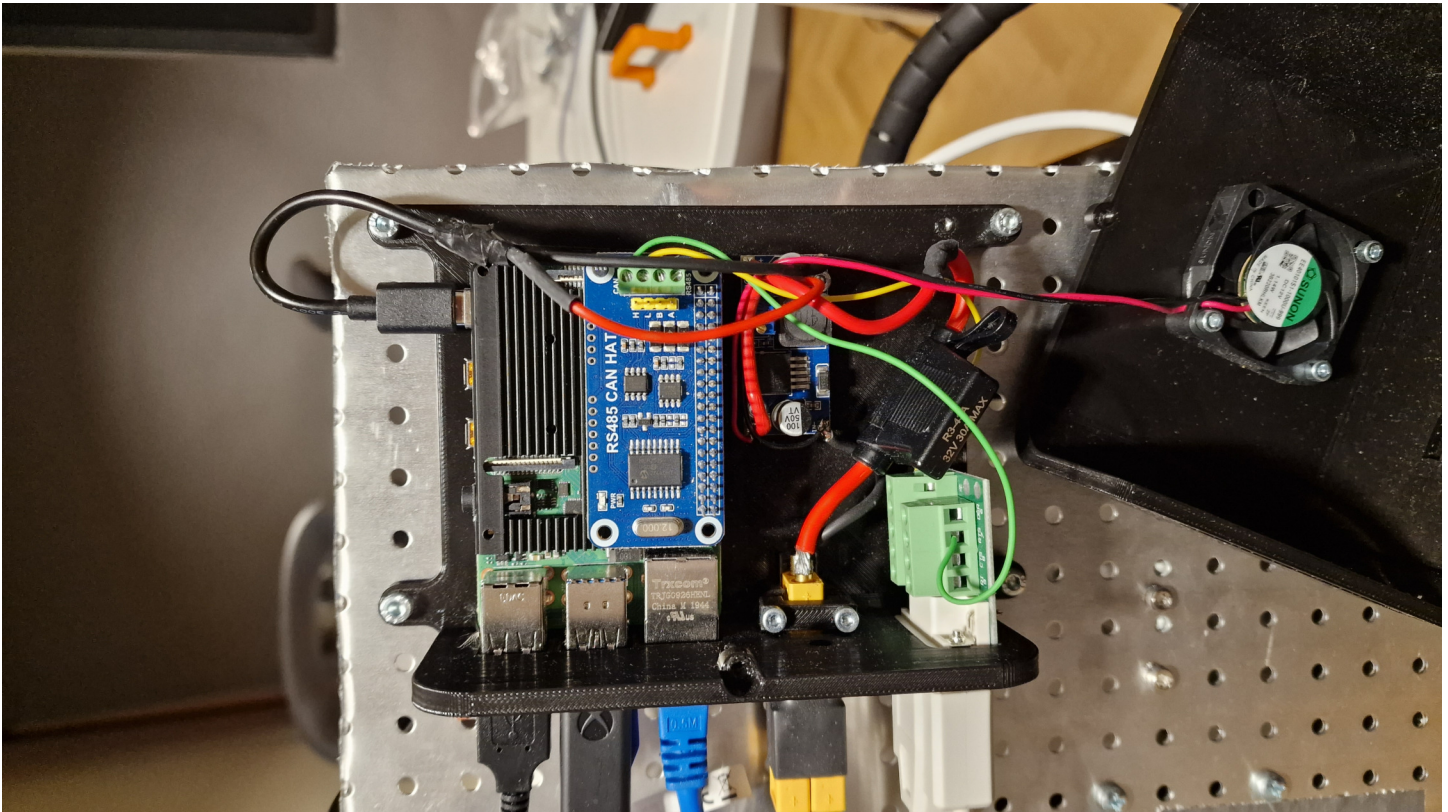
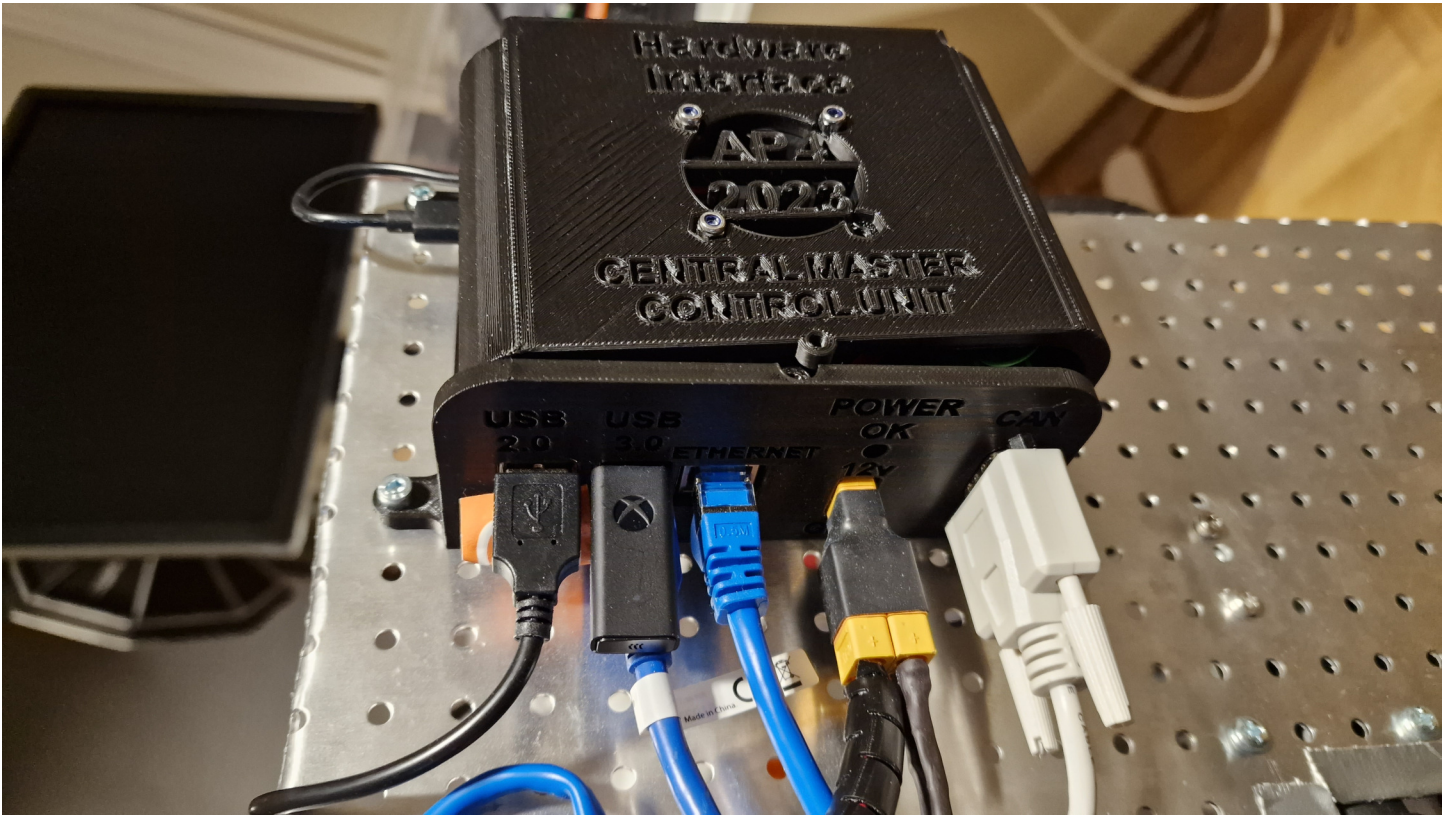
```
ros2 topic echo /GET_<FRAME_ID>_<SIGNAL_NAME>
```

3.4 Setup Hardware Interface Low Level Code (Raspberry Pi)

This document aims to describe how to configure a fresh Raspberry Pi to run the low level software on the autonomous platform.

In theory this should not have to be done again as the raspberry pi mounted on the platform is already configured. But this document can be useful for future reference if the raspberry pi decides to die. The end result of following this document should look something like:





3.4.1 Required Hardware

This is the required hardware to get the hardware interface low level computer up and running from scratch.

Components to buy:

- Raspberry Pi 4b [Link to purchase](#) Recommended 4 or 8 GB RAM version.
- Micro SD card from reputable brand. Minimum recommended size 32GB. 64 is recommended.
- RS485 CAN hat [Link](#)

- Mini-HDMI to HDMI M-M cable
- LM2596 DC-DC converter
- 40x40x10 12v fan
- XT60 female connector
- DB9 Female connector
- Biltema fuse holder
- 3 A mini fuse
- USB C cable to slice
- M3 nut and bolts
- M4 nut and bolts

Components to 3D print:

- Top
- Bottom
- XT60 holder

A computer that can flash a micro sd card is required.

3.4.2 Installing Base Software on Fresh Raspberry Pi 4b

- Raspberry Pi 4b, 32 GB micro SD card
- Download Raspberry pi imager v1.7.3 and select Ubuntu 22.04
- Insert micro SD card (recommended minimum size 32gb)
- Connect display cable, mouse, keyboard
- Plug in power - Raspberry Pi should boot up
- Follow the standard linux installation process presented on screen
- Raspberry pi system configuration
- Name: ap4-hardware-interface
- Computer Name ap4-hw-interface-rpi4
- Username: find it in the main README.md
- Password: find it in the main README.md

Now you can install any new software updates

```
sudo apt-get update && sudo apt-get upgrade
```

Restart Raspberry Pi. Can bus hardware setup can be found in `autonomous_platform\Documentation\Raspberry_Pi_canbuss_research` directory.

The canbuss communication socket speed can be set using (has to done once every boot-up) (is currently done automatically in startup script)

```
sudo apt install can-utils
```

```
sudo ip link set can0 up type can bitrate 1000000
```

Docker [installationsguide](#)

Install can-utils

```
sudo apt-get update && sudo apt-get upgrade
```

3.4.3 Install GPIO library on raspberry pi running ubuntu

If running a ubuntu distribution on raspberry pi, make sure libraries for controlling the gpio are installed.

```
sudo apt install python3-lgpio
```

3.4.4 Install can-utils library

In order for linux to be able to handle can messages, install can utils

```
sudo apt-get install can-utils
```

3.4.5 Raspberry pi 4 configuration

There are some software configurations to do in order to enable a can interface using the mcp2515 board. Once configured, the can0 should show up as an available interface when using the command ‘ifconfig’

Here is a guide we follow in order to do so: <https://www.youtube.com/watch?v=fXiOIUZtV10> and here is a different guide in written format <https://harrisonsand.com/posts/can-on-the-raspberry-pi/>

The MCP2515 module we have is has a clock of 8 Mhz. This is important because it may vary depending on supplier. (The clock frequency should be printed on the clock module on the mcp2515.)

Here is the procedure as described more in detail in the above resources:

1. Update the system

```
sudo apt-get update
sudo apt-get upgrade
```

2. Enable SPI bus and load kernel modules:

Using nano editor

With a raspberry pi 4 running on ubuntu 22.04 this file was located at /boot/firmware/config.txt

hence the command becomes

```
sudo nano /boot/firmware/config.txt
```

In this file, append, where 12000000 is the clock frequency of the RS485 hat

```
dtparam=spi=on
dtoverlay=mcp2515-can0,oscillator=12000000,interrupt=25
dtoverlay=spi-bcm2835-overlay
```

3. Reboot raspberry pi

If everything is setup as it should, running the following command should not throw an error

```
ifconfig can0
```

3.4.6 Setting the CAN speed

The canbuss communication speed can be set using

```
sudo ip link set can0 up type can bitrate 125000
```

Where 125000 is the speed of the connected canbuss

3.4.7 Fixing problems with GPIO pins on raspberry Pi

The MCP2515 CAN controller and the TJCxxx transceiver can operate at different voltages. (But are mounted on the same board referred to as MCP2515)

Raspberry Pi GPIO pins are ONLY 3.3V tolerant, meaning the MCP2515 board had to be modified in order for the logic signals from the card to be understood by the raspberry pi. The CAN transceiver MUST be powered from 5V since the CAN is a 5V network.

3.4.8 Setting up automatic start of software upon boot-up

It is very useful for the software running on the raspberry pi 4b to start when booting up.

This can be done using linux services. The service will call a bash script “testing.bash” located in this directory.

The full procedure can be read [here](#).

The steps are summarized below.

in /etc/systemd/system add a new service file. Name it:

```
startup_lowlevel.service
```

With the following contents:

```
[Unit]
Description=description about this service
After=network.target
StartLimitIntervalSec=0

[Service]
Type=oneshot
User=root
WorkingDirectory=/home/ap4/Desktop/GIT/autonomous_platform/Hardware_Interface_Low_Level_Computer
ExecStart=/home/ap4/Desktop/GIT/autonomous_platform/Hardware_Interface_Low_Level_Computer/testing.bash

[Install]
WantedBy=multi-user.target
```

Reload linux service file to include the newly created service.

```
sudo systemctl daemon-reload
```

Start the newly created service

```
sudo systemctl start startup_lowlevel.service
```

To enable the service to run on boot, enter

```
sudo systemctl enable startup_lowlevel.service
```

The status of the service can be observed using

```
sudo systemctl status startup_lowlevel.service
```

After this the bash script will execute upon boot-up. And the docker container will always run on startup.

3.4.9 Testing and debugging (system level)

Here is a list of things to test and verify

- Does it receive CAN messages?
- Can the docker container be started
- Does the docker container start automatically on boot-up?
- Can raspberry pi 4 be pinged from a computer on the same network?

verify that CAN works expected

3.5 Testing and verifying basic functionality

The Raspberry Pi was set up as a CAN sniffer onto the Infotiv CAN lab kit (CAN Education course - Excercise 2). First the kit was setup to run exercise 2 and send temperature over the can bus. The two nodes could send and receive. Thereafter the raspberry pi was connected onto the CAN bus.

Some errors occurred such as:

- Make sure the CAN speed is set up right
- The oscillator frequency is set to 8Mhz (NOT 20!!)
- Common ground over the complete CAN system, (GND was connected from lab kit to raspberry pi)

4 High-Level Control Software

This directory contains the high level control software which is responsible for sending high level control commands to the low level control software.

4.1 High-Level Control Hardware Requirements

To run the high-level control software it is preferable to run it on Linux. (Tested on Ubuntu 22.04). The processor should of x86 type. It is preferable to have a dedicated graphics card if one has to run a lot of digital twin simulations.

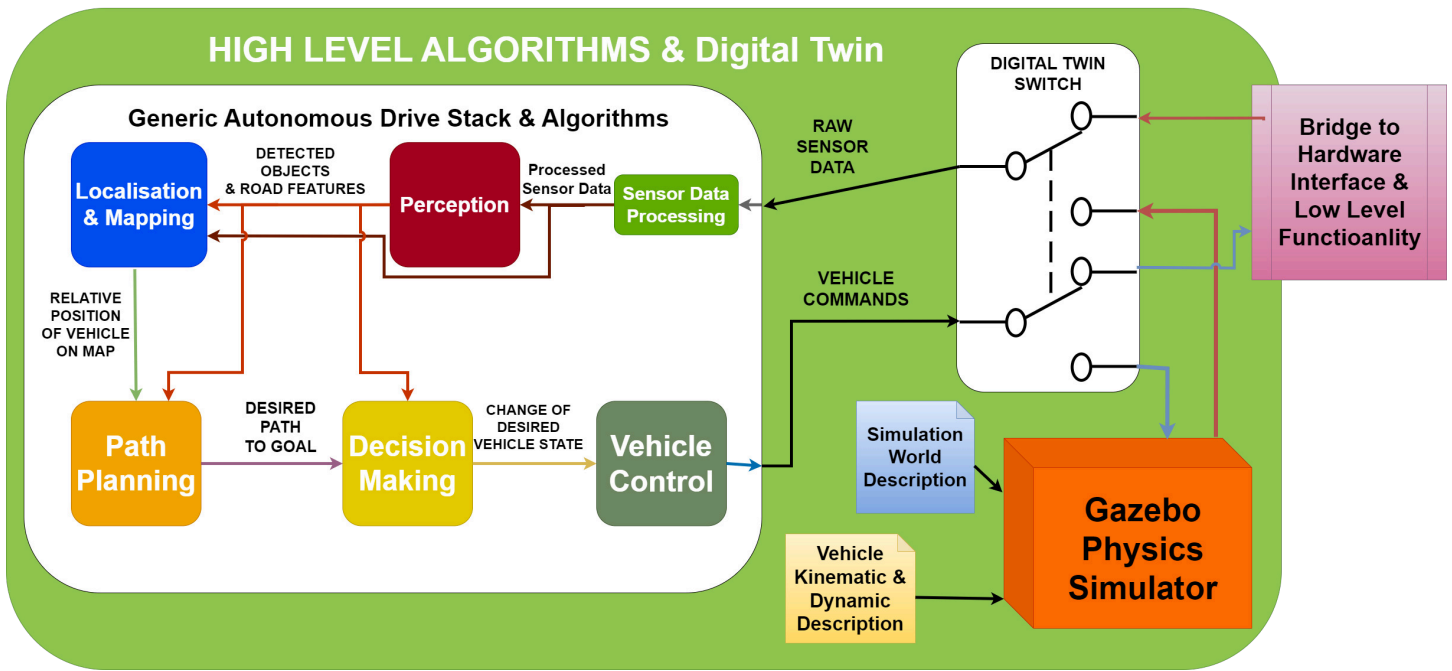
See physics simulator Gazebo Simulator hardware requirements [here](#).

To run high-level control on windows the docker graphics passthrough methodology needs to be modified. Currently it has only been setup for Linux. There will also be a greater performance loss when running a linux based docker container on windows compared to linux.

The high level software is the highest level software layer, it is supposed to be hardware agnostic. It should not care what specific hardware is implemented on the physical autonomous platform.

The high level control software is supposed to tell the autonomous platform WHAT to do, whilst the low level control software is supposed to tell the platform HOW it should do it. This means that the algorithms developed / used in high level control software can be transferred to any physical platform as long as there exists an interface for it.

As an example: A high level software component wants the platform to move forward. It relays this on a generic ROS2 topic (i.e /cmd_vel) to the low level software, the low level software then processes it and sends commands specific to the physical platform to the embedded software layer. The low level software would then output hardware specific commands over the CAN bus network.



Above is a schematic diagram of how the software in high level control software should be designed. As of August 2023 only a simple digital twin is implemented so far.

4.1.1 How To Start

The high level software container should be started on the development laptop. NOT on the Raspberry Pi since it can not render the 3D gazebo simulation.

If any error occurs, TEST_DEBUGGING.md, for troubleshooting.

Note: As of August 2023 it starts the digital twin software on ROS_DOMAIN_ID = 0 meaning it will not be able to interact with the physical platform even if the computers are located on the same wifi network. Hardware platform ROS2 network has ROS_DOMAIN_ID = 1.

Note: The host computer needs to be configured to pass graphical elements to the container. (before starting the container)
(Make sure your terminal path is located in this directory)

```
xhost +local:*
```

First, rebuild the container using

```
docker-compose build
```

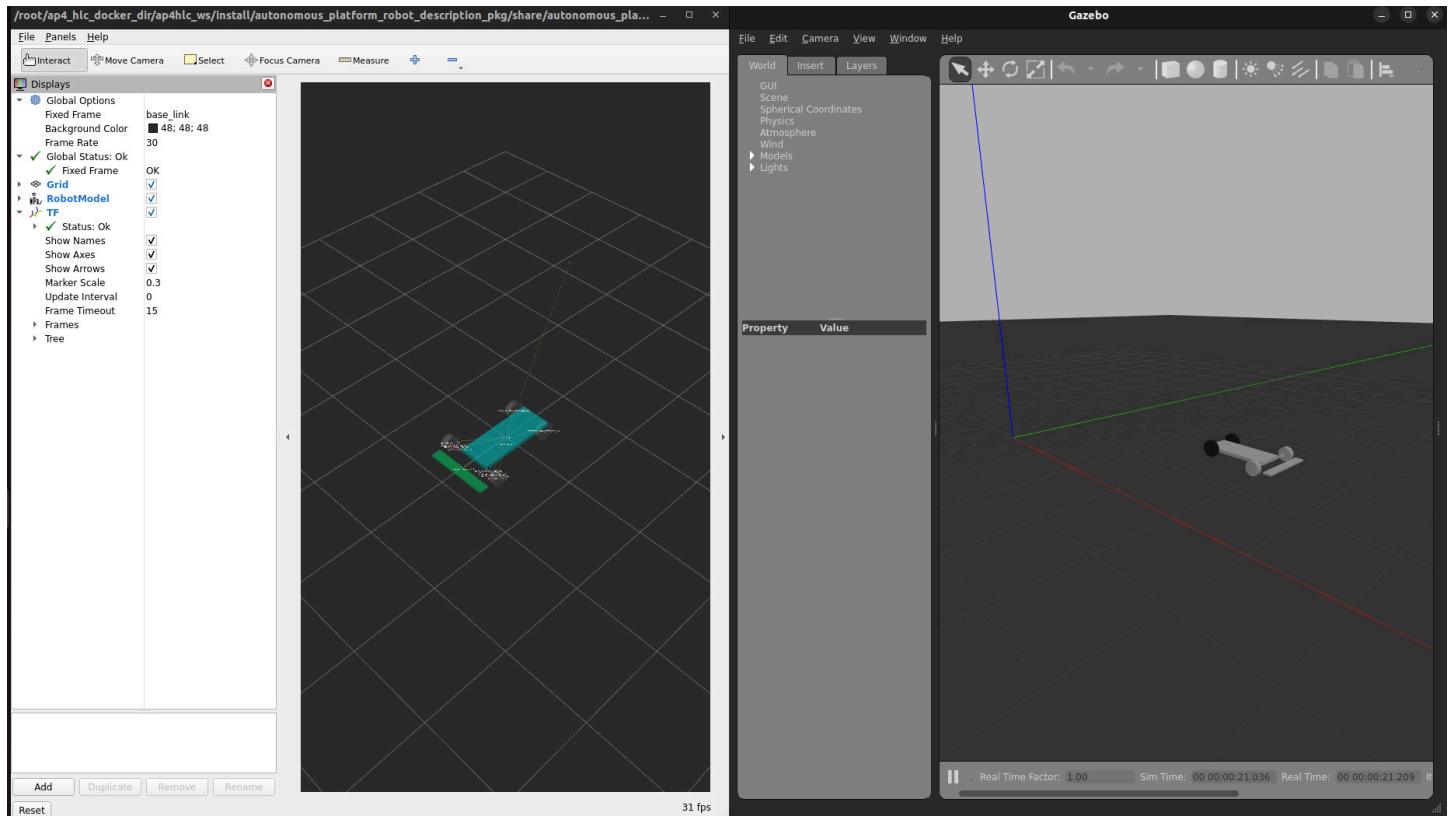
The high level software container, with the configurations, can be started using

docker-compose up

The expected terminal output is:

```
ap4-dev-laptop@ap4devlaptop: ~/Desktop/autonomous_platform_generation_4/High_Level_Control_Computer
Starting ap4hlc ... done
Attaching to ap4hlc
ap4hlc | Starting >>> autonomous_platform_robot_description_pkg
ap4hlc | Finished <<< autonomous_platform_robot_description_pkg [0.09s]
ap4hlc |
ap4hlc | Summary: 1 package finished [0.18s]
ap4hlc | [INFO] [launch]: All log files can be found below /root/.ros/log/2023-08-28-09-14-12-594590-ap4devlaptop-1
ap4hlc | [INFO] [launch]: Default logging verbosity is set to INFO
ap4hlc | [INFO] [robot_state_publisher-1]: process started with pid [114]
ap4hlc | [INFO] [rviz2-2]: process started with pid [116]
ap4hlc | [INFO] [gzserver-3]: process started with pid [118]
ap4hlc | [INFO] [gzclient-4]: process started with pid [120]
ap4hlc | [INFO] [spawn_entity.py-5]: process started with pid [122]
ap4hlc | [rviz2-2] Authorization required, but no authorization protocol specified
ap4hlc | [rviz2-2] qt.qpa.xcb: could not connect to display :0
ap4hlc | [rviz2-2] qt.qpa.plugin: could not load the Qt platform plugin "xcb" in "" even though it was found.
ap4hlc | [rviz2-2] This application failed to start because no Qt platform plugin could be initialized. Reinstalling the application may fix this problem.
ap4hlc | [rviz2-2] Available platform plugins are: eglfs, linuxfb, minimal, minimalegl, offscreen, vnc, xcb.
ap4hlc | [robot_state_publisher-1] [WARN] [1693214053.002320711] [kdl_parser]: The root link base_link has an inertia specified in the URDF, but KDL does not support a root link with an inertia. As a wo
rkaround, you can add an extra dummy link to your URDF.
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002406557] [robot_state_publisher]: got segment backwheel_drive_wheel_left_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002459651] [robot_state_publisher]: got segment backwheel_drive_wheel_right_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002464216] [robot_state_publisher]: got segment base_footprint
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002467749] [robot_state_publisher]: got segment base_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002470840] [robot_state_publisher]: got segment body_extension_left_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002474253] [robot_state_publisher]: got segment body_extension_right_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002477413] [robot_state_publisher]: got segment frontwheel_left_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002480725] [robot_state_publisher]: got segment frontwheel_right_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002483793] [robot_state_publisher]: got segment gokart_frontwing_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002486687] [robot_state_publisher]: got segment steering_swivel_left_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002489845] [robot_state_publisher]: got segment steering_swivel_right_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002492902] [robot_state_publisher]: got segment swivel_extension_left_link
ap4hlc | [robot_state_publisher-1] [INFO] [1693214053.002495837] [robot_state_publisher]: got segment swivel_extension_right_link
ap4hlc | [ERROR] [rviz2-2]: process has died [pid 116, exit code -6, cmd '/opt/ros/humble/lib/rviz2/rviz2 -d /root/ap4_hlc_docker_dir/ap4hlc_ws/install/autonomous_platform_robot_description_pkg/share/aut
onomous_platform_robot_description_pkg/rviz2/urdf_config.rviz --ros-args -r __node:=rviz2'].
ap4hlc | [spawn_entity.py-5] [INFO] [1693214053.201051162] [spawn_entity]: Spawn Entity started
ap4hlc | [spawn_entity.py-5] [INFO] [1693214053.201446217] [spawn_entity]: Loading entity established on topic robot_description
ap4hlc | [spawn_entity.py-5] /opt/ros/humble/local/lib/python3.10/dist-packages/rcipy/qos.py:307: UserWarning: DurabilityPolicy.RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL is deprecated. Use DurabilityPoll
cy.TRANSIENT_LOCAL instead.
ap4hlc | [spawn_entity.py-5] warnings.warn(
ap4hlc | [spawn_entity.py-5] [INFO] [1693214053.203132918] [spawn_entity]: Waiting for entity xml on robot_description
ap4hlc | [spawn_entity.py-5] [INFO] [1693214053.325048669] [spawn_entity]: Waiting for service /spawn_entity, timeout = 30
ap4hlc | [spawn_entity.py-5] [INFO] [1693214053.325963774] [spawn_entity]: Waiting for service /spawn_entity
ap4hlc | [gzserver-3] Authorization required, but no authorization protocol specified
ap4hlc | [spawn_entity.py-5] [INFO] [1693214053.332256856] [spawn_entity]: Calling service /spawn_entity
ap4hlc | [gzserver-3] [INFO] [1693214053.964284645] [spawn_entity]: Spawn status: SpawnEntity: Successfully spawned entity [AP4 digital twin]
ap4hlc | [gzserver-3] [INFO] [1693214053.972554302] [gazebo_ros_joint_state_publisher]: Going to publish joint [frontwheel_left_joint]
ap4hlc | [gzserver-3] [INFO] [1693214053.972883745] [gazebo_ros_joint_state_publisher]: Going to publish joint [frontwheel_right_joint]
ap4hlc | [gzserver-3] [INFO] [1693214053.972896729] [gazebo_ros_joint_state_publisher]: Going to publish joint [drive_wheel_left_joint]
ap4hlc | [gzserver-3] [INFO] [1693214053.972904442] [gazebo_ros_joint_state_publisher]: Going to publish joint [drive_wheel_right_joint]
ap4hlc | [gzserver-3] [INFO] [1693214053.972910829] [gazebo_ros_joint_state_publisher]: Going to publish joint [steering_swivel_left_joint]
ap4hlc | [gzserver-3] [INFO] [1693214053.972917470] [gazebo_ros_joint_state_publisher]: Going to publish joint [steering_swivel_right_joint]
ap4hlc | [gzserver-3] [WARN] [1693214053.987874688] [gazebo_ros_arkermann_drive]: Steering wheel joint [steering_wheel_joint] not found
```

Two new windows should open up, Gazebo and Rviz, and will look something like this:



The digital twin can be controlled (drive around manually) using the keyboard in a new terminal window.

Enter the running container

```
docker exec -it ap4hlc bash
```

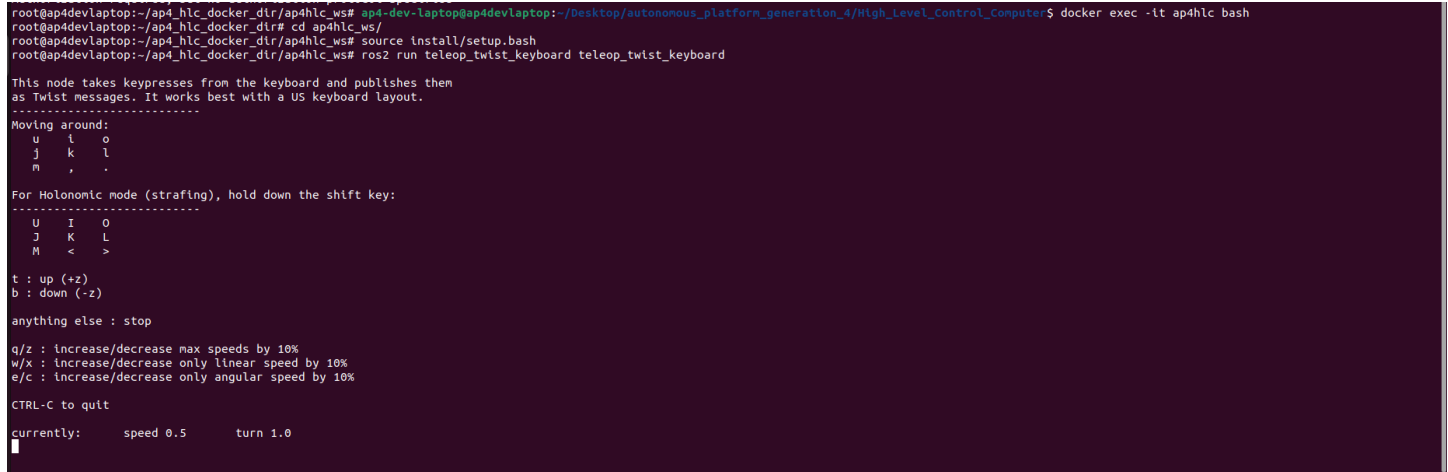
Navigate to workspace, source environment variables

```
cd ap4hlc_ws
source install/setup.bash
```

Start tele-operation twist keyboard:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

By using ‘i’, ‘j’, ‘l’, ‘k’ and ‘,’ one can now control the digital twin. See expected terminal output below.



```
root@ap4devlaptop:~/ap4_hlc_docker_dir/ap4hlc_ws# ap4-dev-Laptop@ap4devlaptop:~/Desktop/autonomous_platform_generation_4/High_Level_Control_Computer$ docker exec -it ap4hlc bash
root@ap4devlaptop:~/ap4_hlc_docker_dir# cd ap4hlc_ws/
root@ap4devlaptop:~/ap4_hlc_docker_dir/ap4hlc_ws# source install/setup.bash
root@ap4devlaptop:~/ap4_hlc_docker_dir/ap4hlc_ws# ros2 run teleop_twist_keyboard teleop_twist_keyboard

This node takes keypresses from the keyboard and publishes them
as Twist messages. It works best with a US keyboard layout.
-----
Moving around:
  u  t  o
  j  k  l
  m  ,  .

For Holonomic mode (strafing), hold down the shift key:
-----
  U  I  O
  J  K  L
  M  <  >

t : up (+z)
b : down (-z)

anything else : stop

q/z : Increase/decrease max speeds by 10%
w/x : Increase/decrease only linear speed by 10%
e/c : Increase/decrease only angular speed by 10%

CTRL-C to quit

currently:   speed 0.5      turn 1.0
█
```

A running container can be stopped by either ‘Ctrl+C’ in the terminal in which ‘docker-compose up’ was run. OR in a new terminal:

```
docker stop ap4hlc
```

4.1.2 Autonomous Driving Stack

This has not been implemented as of August 2023.

In the future, the high level control software shall contain an autonomous drive stack. It would be the highest form of control software to the platform, taking in sensor data, processing it and deciding on an actuator output. There exists available AD driving stacks which one can use, i.e OpenPilot ([Link here](#)). Hamid has previous experience with using OpenPilot. One could also develop some autonomous driving stack in-house or as a part of thesis project.

The idea would still be the same, sensor information and physical platform states would be received on standardized ros2 topics. The information would be processed in some ROS2 nodes and eventually the output would be a change of vehicle state. The output should follow ROS2 convention of controlling robots. I.e a new desired linear velocity and rotational velocity should be output on the /cmd_vel ROS2 topic.

For topic conventions see ROS2 topic message Sensor Messages ([link](#)) and Navigational Messages ([link](#)).

For quick reference

- (INPUT) Platform Odometry (Current position + Orientation) see “/odom”
- (OUTPUT) Velocity in 3D (linear + angular) /cmd_vel
- (INPUT) Sensor readings. See common sensor topics [here](#)

4.1.3 Digital Twin

A simplified digital twin has been implemented as of August 2023.

An important part of the high level software control is the digital twin. It is a completely different component then the autonomous driving stack and should be agnostic to what AD stack is used. This means that any AD stack should be able to be tested on the digital twin. This is accomplished by using standard ROS2 topics as interfaces as briefly described in the previous section. Inputs to the digital twin would be sent over the /cmd_vel ROS2 topic, in the same way outputs (vehicle state and sensor readings) would be sent from the digital twin simulation over standard ROS2 topics. I.e “/odom” for vehicle state.

As of August 2023 the digital twin is run using the Gazebo Physics Simulator. [Link official gazebo documentation](#). It is a software that integrates seamlessly with ROS2 and has been used for a long time to simulate robots in complex environments.

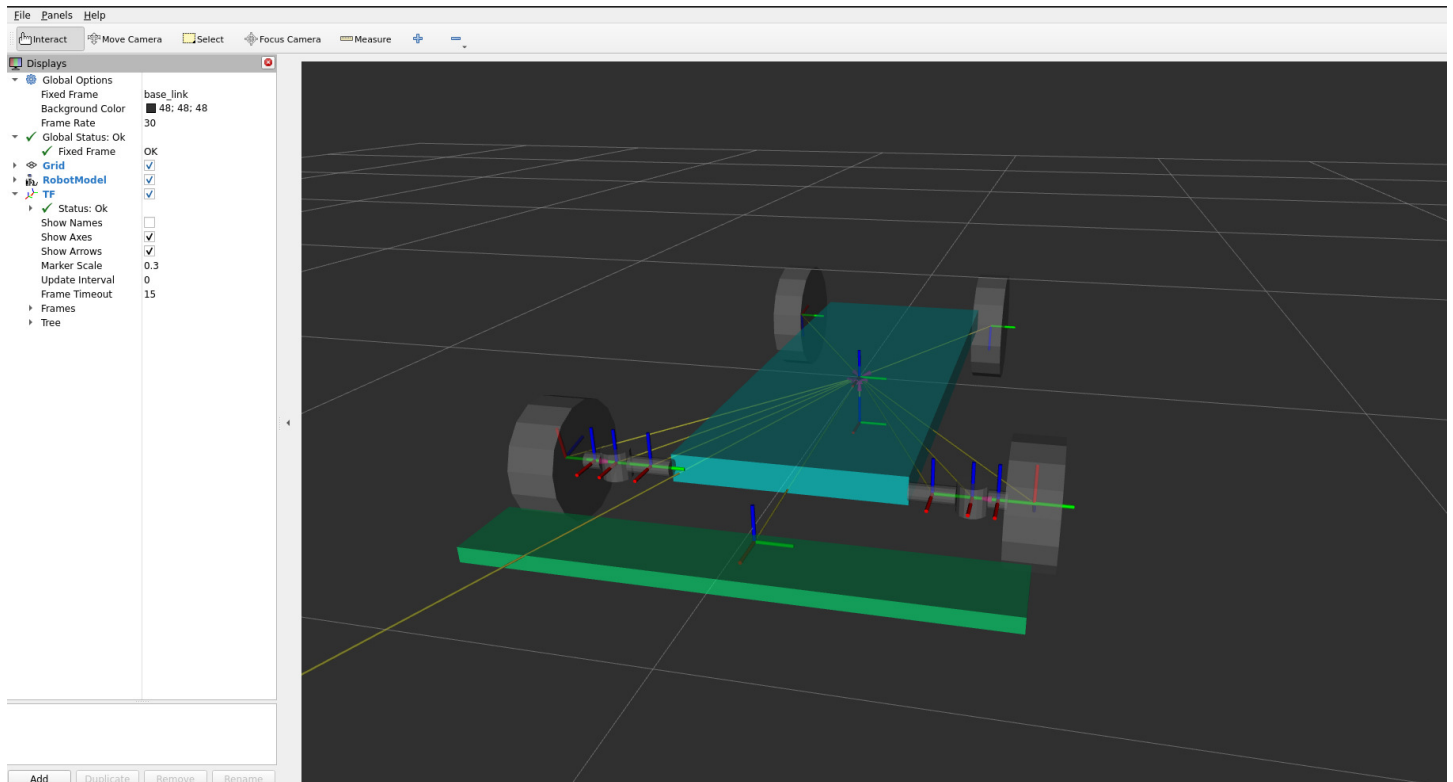
The code structure for the digital twin is as follows: A ROS2 package has been created, `autonomous_platform_robot_description_pkg` which contains the digital twin.

```
└─ High_Level_Control_Computer
  └─ ap4_hlc_code
    └─ ap4hlc_ws
      └─ src
        └─ autonomous_platform_robot_description_pkg
          └─ launch
            └─ rviz
              └─ src
                └─ description
                  └─ ap4_robot_description.urdf
                  └─ gazebo_control.xacro
                  └─ gokart_chassi.urdf.xacro
                  └─ inertia_macros.xacro
                └─ worlds
          └─ Dockerfile
          └─ docker-compose.yaml
          └─ README.md <-- You Are Here !!!
```

4.1.4 Design Of Digital Twin

The digital twin of autonomous platform is created using Universal Robot Description Files (URDF) and xacro files. The digital twin is described in an xml type format and are located in `autonomous_platform\High_Level_Control_Computer\ap4_hlc_code\ap4hlc`

These files describe everything from the physical platform properties to what sensors are simulated. How the platform is controlled can also be configured. The end result can be seen below, a simplified gokart platform with wheels that can be controlled.



4.1.5 Sensor plugins

An important aspect of using Gazebo is the concept of plugins. Plugins allows for new functionality to be added to the digital twin. For example different vehicle movement controllers or virtual sensors.

As of August 2023 no sensors have yet to be added. Future sensors to add could be:

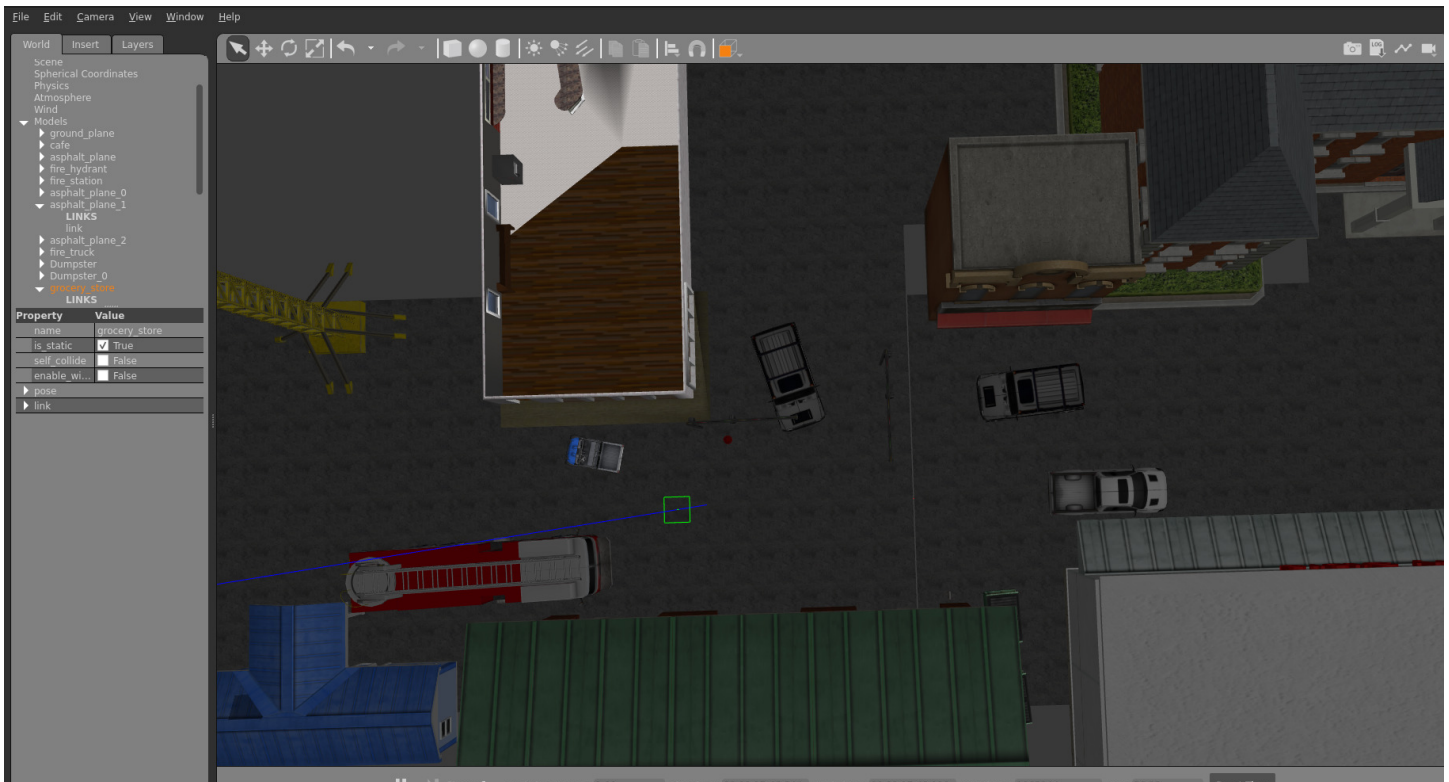
- [Camera](#)
- [Lidar](#)
- [Radar](#)
- [IMU](#)
- [Ultrasonic short range sensor](#)

4.1.6 Gazebo worlds

In the same way as the digital twin could be configured, the simulated environment itself can be configured. This allows a developer to create interesting scenarios in which to place the autonomous platform.

Worlds can be created using the Gazebo graphical application and saved into the worlds directory. The digital twin can be configured to start in a specific world by configuring the launch file located in launch directory.

Gazebo has access to an exentsive library of 3D models which can be used to crate interesting environments.





4.1.7 Software Control Switch

This has not been implemented as of August 2023.

The idea of having a software control switch is for the autonomous driving stack algorithms to EITHER control the physical platform or the digital twin. And be able to switch seamlessly between them. The autonomous driving stacks could be quickly tested on a digital twin to make development faster and when the algorithms are in a mature state they could be tested on the physical platform.

It is therefore very important that the digital twin and physical platform have the same output (sensor readings / vehicle state). I.e on the same ROS2 topics. In the same way, the digital twin and the physical platform should be controlled in the same way in the high control software on /cmd_vel topic.

For future reference this switching of control could be actualized using [namespaces](#) or configuring different [ROS2 domain IDs](#). By setting the ROS2 DOMAIN ID for the AD nodes to the same ID as the physical platform commands would be forwarded to the low level software control. And by setting a different domain one could ensure that the commands are only sent between the digital twin and AD algorithms. ROS_DOMAIN_ID is specifically developed to prevent cross talk between different domains.

4.1.8 Digital Twin Software Package Structure

The software for the digital twin is located in `autonomous_platform_robot_description_pkg`. Below follows a quick guide on what each subdirectory contains:

- `launch` - Contains launch files
- `rviz` - contains a parameter file which can be inserted into Rviz to show useful information upon startup
- `worlds` - save gazebo simulation environments (worlds) to be run later.
- `/src/description`: Contains the xacro and urdf files which describes how the digital twin is built up.

The launch file `launch_digital_twin_simulation.launch.py` launches the following nodes

- `robot_state_publisher`
- `Rviz`
- `gazebo_node`
- `spawn_entity_node`

The robot state publisher node, takes the robot description files and broadcasts them over the ROS2 network. The Gazebo physics simulator is then launched with a corresponding ROS2 node. Lastly, a spawn entity node is created, which spawns

the digital twin inside the gazebo simulation using the information on the robot state publisher information topic. As the digital twin is inserted into the simulation, it spawns further nodes which makes it possible to control the digital twin using the `/cmd_vel` topic.

The digital twin is defined inside the description folder, it is built in modules with the `ap4_robot_description.urdf` as a base. Onto this base. The `gokart_chassi.urdf.xacro` describes the kinematics and dynamics of AP4. `gazebo_controls.xacro` describes how an ackermann drive plugin is used to control the joints of AP4.

Future sensors, such as cameras, should be added as xacro modules and included in the `ap4_robot_description.urdf` file.

4.1.9 How to Connect to Physical Platform

The high-level control software is meant to be run in two modes; connected to the autonomous platform and completely detached.

The long term goal is that the high level control software should be able to interface with the low level control software running on the Raspberry Pi 4b. This is done through an ethernet connection, meaning it could be done both wirelessly and by wire. The two docker containers for high level software and low level software should therefore be started from devices located on the same network.

As of August 2023 this is NOT needed yet, since no Autonomous Driving algorithms have been implemented yet. But for future reference:

- Connect the development laptop to the AP4-ROUTER_2.4Ghz network. ssid and pw credentials can be found in root directory README file.
- Make sure the two docker containers are started
- Make sure high level software is running on `ROS_DOMAIN_ID=1`

This should be sufficient for the underlying ROS2 Data Distribution Service (DDS) to find ROS2 nodes available on the same network and same `ROS_DOMAIN_ID`.

4.1.10 How to verify

If any error occurs, `TEST_DEBUGGING.md`, for troubleshooting.

4.1.11 High Level Control Underlying Software Components

This section will describe what software components are used to construct the high level software and how they are used.

4.1.12 Containerization

An overview of containerization and how it works is explained in `SOFTWARE_DESIGN.md` located in root directory.

Note: Docker can be run on Windows, but certain commands/parameters used on AP4 are linux specific. I.e passing graphics to and from the container. It is has therefore only been tested and guaranteed to work on linux host computers.

Docker enables software to be collected and run from a virtual environment, similar to a virtual machine but with much less performance overhead compared to a virtual machine. Docker allows the virtual environment to be configured, i.e what operating system should be run and what software should be installed.

The environment in which the high level software is run in is described inside the `Dockerfile` located in this directory. In this file the virtual environment is configured as a base version of Ubuntu 22.04 with relevant linux packages installed. Robot Operating System 2 - Humble is installed.

The container is started using a set of configuration parameters, these are located in `docker-compose.yaml` in this directory. Configuration parameters can be for example to pass through graphical elements from the container to the host computer desktop.

4.1.13 Robot Operating System 2 (ROS2)

Robot Operating System 2 (ROS2) is a framework / middleware developed to create very complex robotic software. This framework is used to split up computations into separate executions using “nodes”.

For an in depth explanation of Robot Operating System 2 (ROS2) and how it works see `SOFTWARE_DESIGN.md` in root directory.

The following subsection will describe how the high level software control is designed using the Robot Operating System Framework.

The idea with using ROS2 framework is that computational applications can be split up into smaller components that perform very specific tasks. I.e software related to taking a joystick input and outputting a desired velocity should have the minimum amount of dependencies as possible and not break any other dependencies for other software applications.

ROS2 package has to be created for this functionality. A package collects all the resources and dependency configurations for a specific function. I.e the package for the digital twin `autonomous_platform_robot_description_pkg` should be as standalone as possible. When adding future functionality, the digital twin package should be left as is, and instead a new package for the new specific functionality should be added.

4.2 Extend High Level Software

This document aims to describe the process of how to extend the high level control software.

Before starting to develop and adding code to the high level control software you need to first make sure you need to add something here.

If you;

- Want to do something with the digital twin
- Work on high level hardware agnostic autonomous drive algorithms
- Evaluate autonomous drive algorithms
- Are NOT adding new hardware
- Are NOT interfacing with the physical platform

Then you are in the right spot!!

If not, take a look at `Hardware_Interface_Low_Level_Computer` or `CAN_Nodes_Microcontroller_Code`, maybe you intended to add functionality there!

4.2.1 Prerequisites

In order to start adding functionality it is recommended to have a basic understanding of:

- C++ OR Python development
- docker containers (How to start, stop, restart and configure)
- Linux - The container software environment is mainly navigated in through a terminal
- Robot Operating System 2 (how to create packages and start new nodes)

Software wise, you need to have the following installed:

- docker
- git
- VSCode (recommended but any IDE may be suitable)

Hardware wise, it is recommended you have:

- Linux based x86 host computer, preferably with dedicated graphics (not a must)

4.2.2 Add a New Functionality

First of all make sure you have read the general design principles document for autonomous platform located at `autonomous_platform/HOW_TO_EXTEND.md`. This document takes precedence over anything written in this document in order to unify the development process across all software layers.

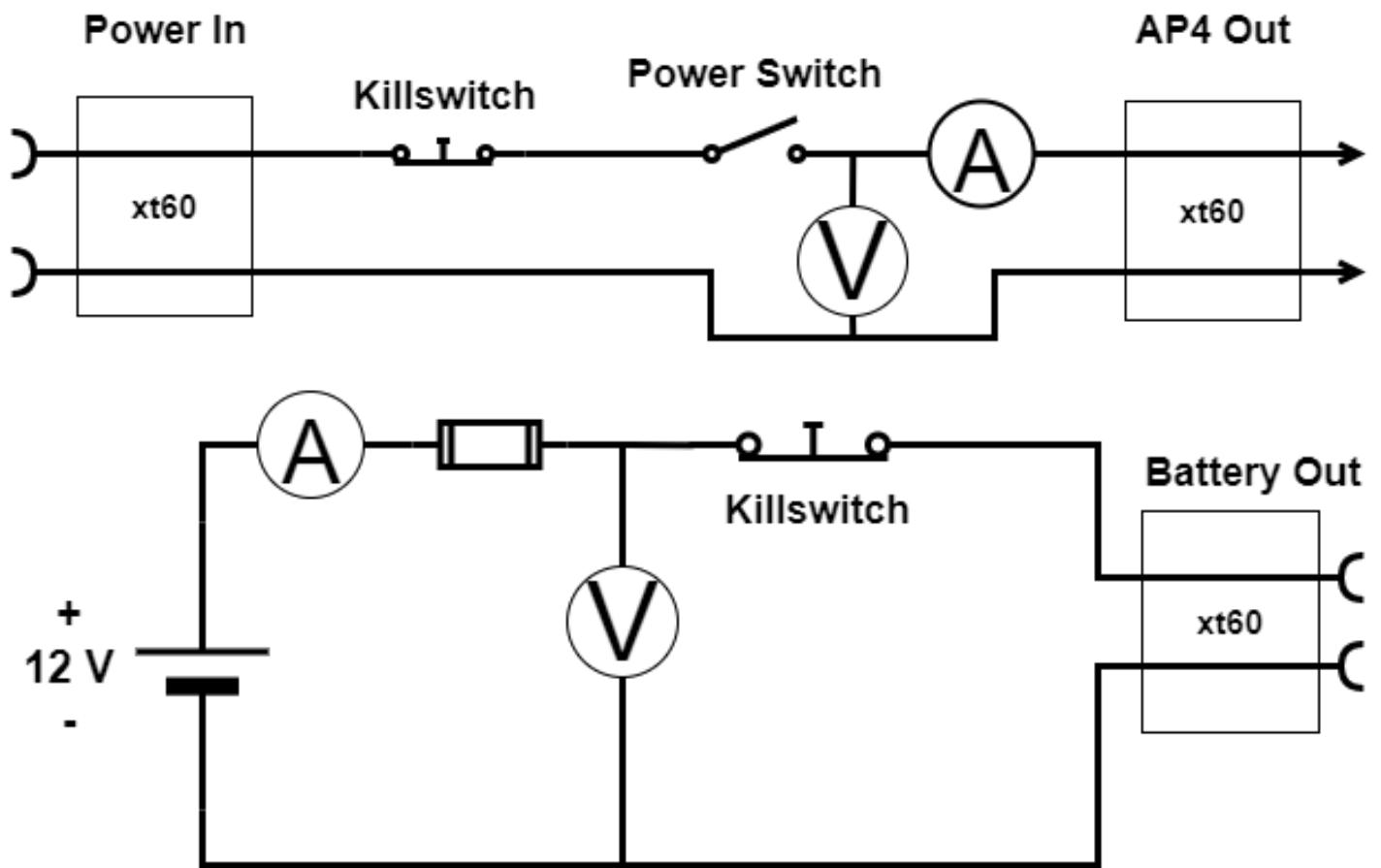
Software functionality is created inside ROS2 packages. These can be seen as code libraries that are configured to run and perform a specific task within a ROS2 network.

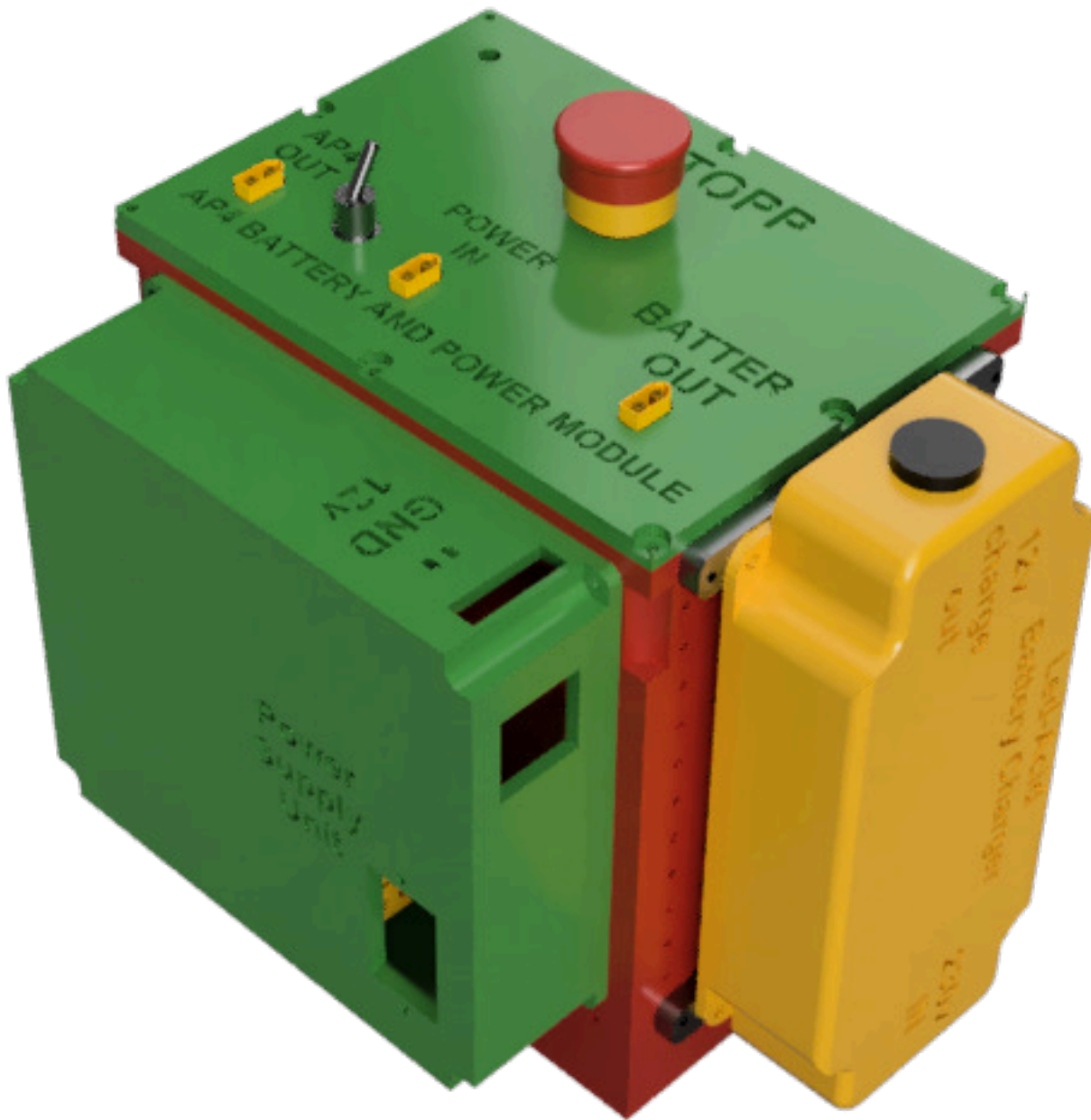
5 Power Module

The power consumption of the implemented AP4's hardware has been calculated in the table below. The values have been derived from each component's product manual, which can easily be found on the internet. The values are based on the nominal values in order to define overall power consumption. Some components have been neglected such as the losses in cables, wires and switches. The power calculation follows the well-known electrical direct current power formula: $P = U \cdot I$ [W] where U is the voltage and I is the current.

Component	Voltage	Current A	Quantity	Power W
Raspberry Pi 4b	5	1.01	1	5.1
DC-Motor EMG49	24	2.1	1	50.4
Cooling Fan	12	0,095	2	2.28
Bluepill	5	0.035	1	0.175
MCP2515, CAN module	5	0,05	1	0,25
MCP4725, DAC	5	0,05	2	0,05
Router	12	1	1	12
Camera, Logitech c920	5	0.5	1	2.5
Total Power Consumption:		73.2		

The power module consists of three components, a battery, a battery charger and a power supply unit (PSU). Furthermore, there is a user interface including a power switch, emergency kill switch and outlets that select which power source that should be used for the autonomous platform. Also, there are voltage- and current measurement units mounted, but they are not implemented in software or physically connected to an ECU. The complete electrical circuit of the power module and a rendered illustration of the CAD files for the power module can be seen in here:





5.0.1 Battery and Charger

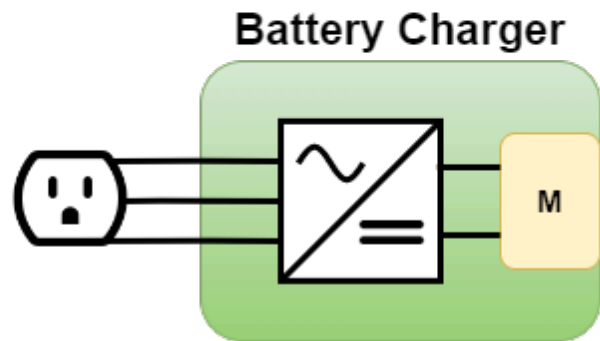
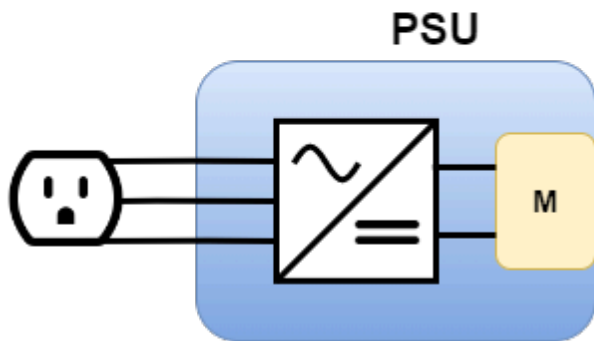
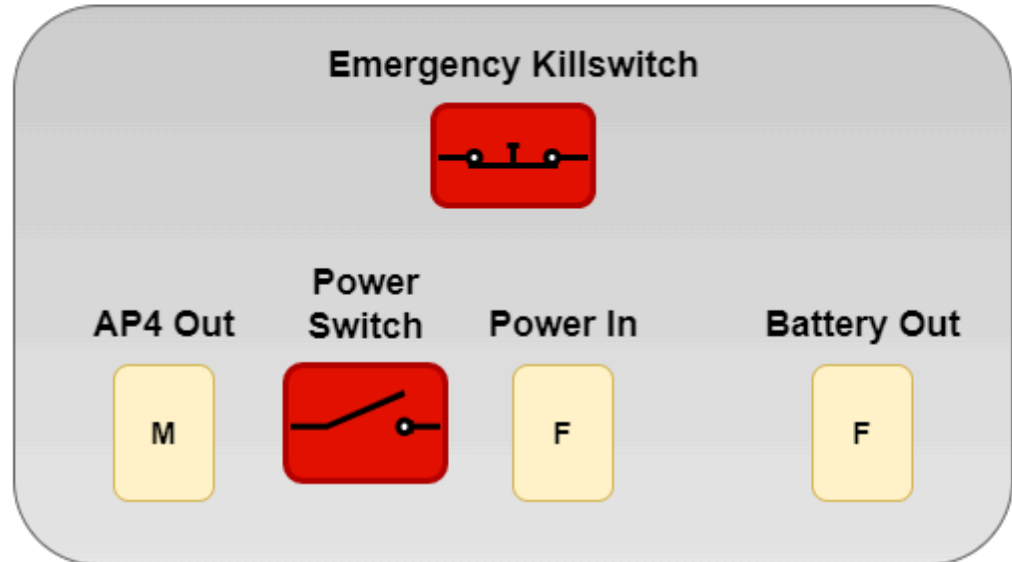
The battery chosen is a 12 V lead acid battery from Biltema. The battery is typically used for lawnmowers and similar machines. The corresponding battery charger selection is based on compatibility with the battery, also from Biltema. The battery capacity is 30 Ah and weighs about 8kg. Thus from the derived system current consumption the theoretically run-time of the system will be: $t = 30/8.33 = 3.6$ hours

Having a power supply unit (PSU) connected to the electrical grid will enable the use of AP4 in a controlled environment. This can be done inside of a lab with the AP4 raised, so no wheel contact on the surface. Thus enabling the testing of higher levels of algorithms without the movement and unnecessary use of the battery that will shorten its lifespan.

5.0.2 User interface

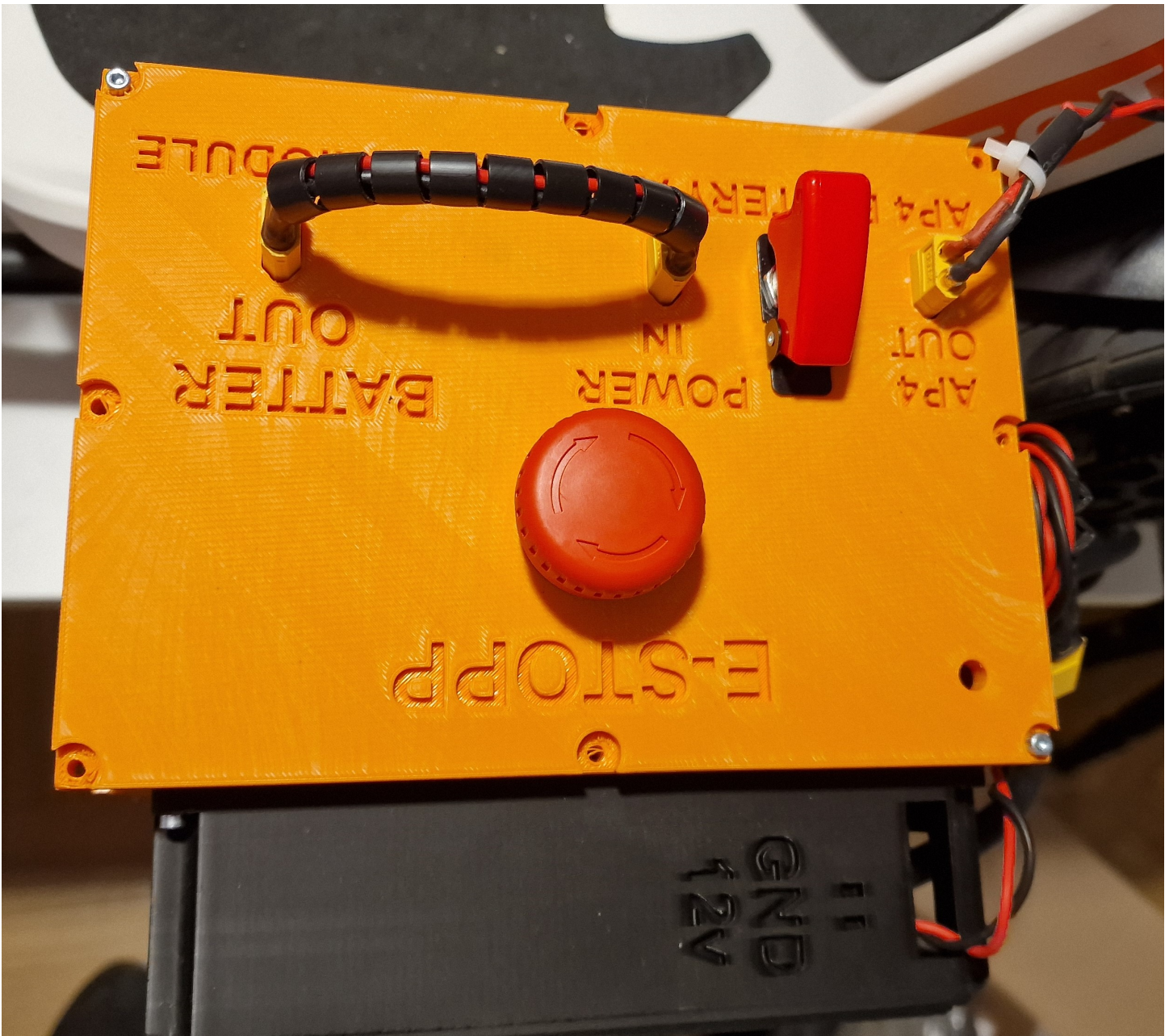
The electrical circuit for the user interface and connected sensors and the schematic of the power module can be seen in the images below. The user interface includes three different outlets in order to select which power source that should be used. These outlets are called, battery out, power in, AP4 out. Battery out is directly connected to the battery terminals, with the kill switch in series.

Interface Power Module



Power in outlet will power the whole platform. The power in is connected to the kill switch, power switch and fuses, then passed out to AP4 out that is connected to the rest of AP4. The selection of power source is based on which source either PSU or battery is connected to the power in outlet. For example, the PSU could power the platform whilst the battery is charged at the same time.





6 Testing, Debugging, Known Issues and Future Work

6.1 Test and debugging (System level)

This document collects known problems and issues with the autonomous platform and how to resolve them.

If the existing documentation could not provide a sufficient solution for a problem. Please add your problem and new solution to this document. It will help future members of this project.

6.1.1 Possible Errors When Starting AP4

Problem: Steering works but not propulsion?

Solution(s): Check that segway is turned on (and charged), check that the the propulsion can be controlled manually by pressing the pedals. The back wheels should spin.

Problem: Segway is beeping? and wheels are turning backwards

Solution: the segway is in reverse mode, quickly press the brake pedals twice. The beeping should stop and segway should move forward when pressing the accelerator pad.

Problem: Segway is beeping and a yellow/orange light flashes on the front console / nose.

Solution: The batteries for the front black box circuits need to be replaced. Open the black compartment on the nose and replace the 6 AA batteries.

Problem: Neither steering or propulsion works.

Solution: Wait for two minutes, The hardware interface software on the raspberry pi could take a while to boot up

Problem: Neither steering or propulsion works after waiting 2 minutes after power ON.

Solution: Connect a screen (portable small screen) and keyboard + mouse to the raspberry pi to verify that it has booted up properly.

Sub-problem: Raspberry Pi could not boot up? Or stuck on boot-up sequence?

Solution: Fix errors using command line, try rebooting system. Follow steps down below to make sure the system is working correctly if the raspberry pi has successfully booted up.

1. Check that the Hardware_Interface docker container is running on the Raspberry Pi In a terminal:

```
docker container ps
```

Expected output should look something like this, the important thing is that the container is up and running.

```
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS        NAMES
xxxx          hardware_interface_low_level_computer_ros2  xxxxx                  10 seconds ago  Up 10 seconds        ap4hwi
```

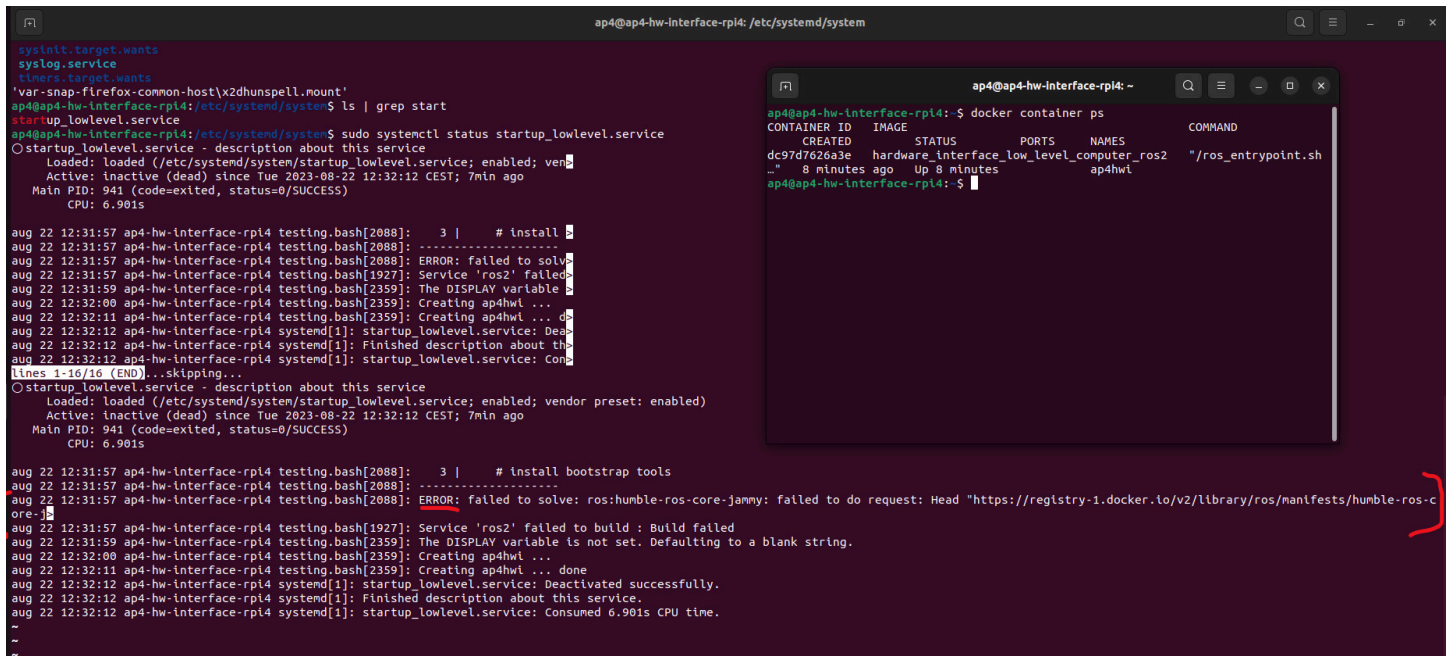
If the container has not started there are two possibilities: 1. The container is still being built (could take a lot of time if new changes have been pulled). 2. The startup service has not been started properly (could have failed due to docker building errors) [systemd service guide](#)

The startup service will run the 'testing.bash' script located in Hardware_Interface_Low_Level_Computer once called.

In a terminal check that the service has been started

```
sudo systemctl status startup_lowlevel.service
```

output:



In the terminal it would explicitly say if something has failed and the service did not start properly. Note that it says ERROR in the screenshot above. Something is therefore wrong!

6.1.2 Docker Container starting/Building

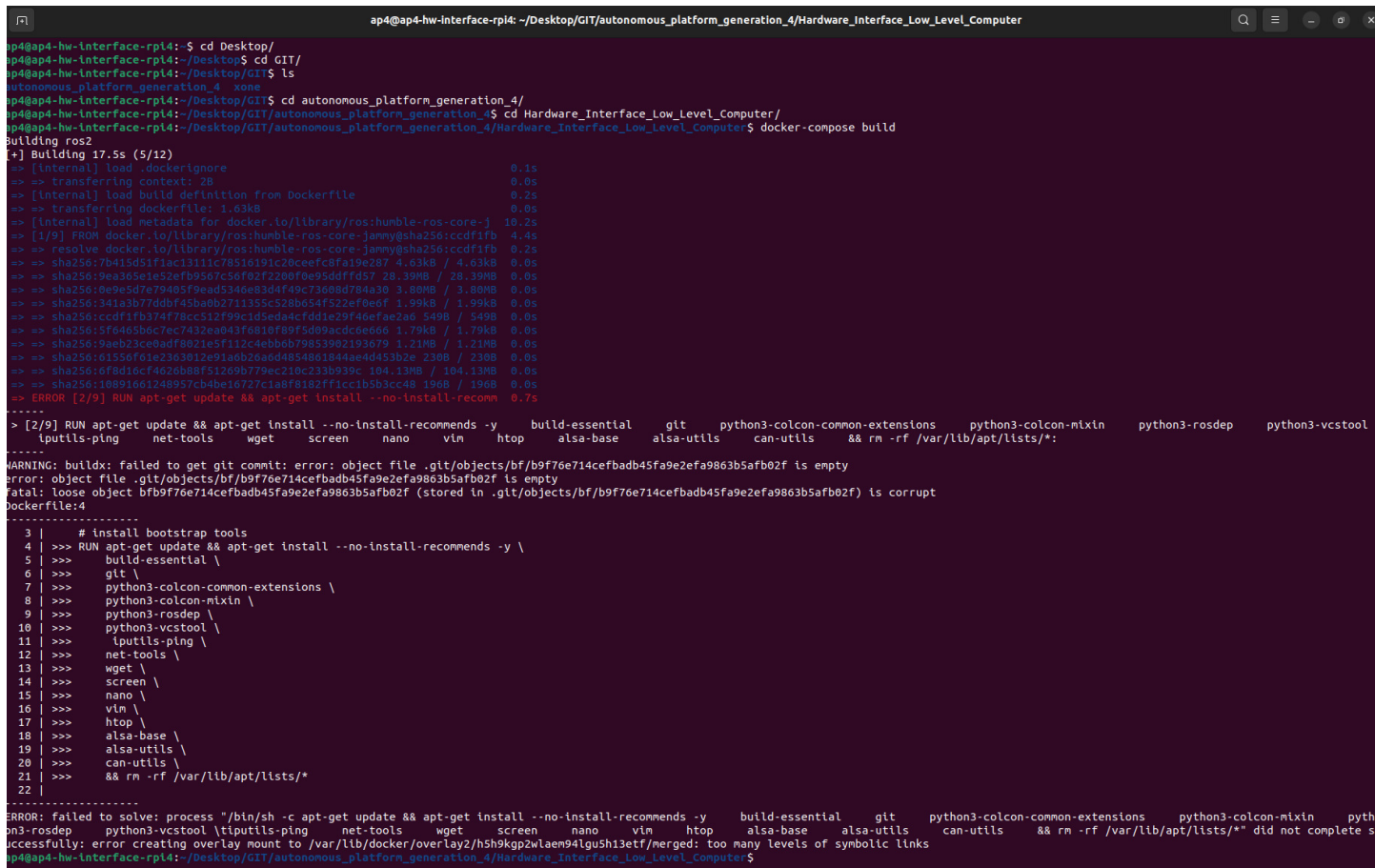
If the docker container is not built properly none of the hardware interface software will start properly.

Try rebuilding the docker container:

```
cd Desktop
cd GIT
cd autonomous_platform
cd Hardware_Interface_Low_Level_Computer
docker-compose build
```

If this fails, there could be something wrong with the dockerfile or docker-compose file.

One possible error is “error creating overlay mount...”. See image below.



```
ap4@ap4-hw-interface-rpi4: ~/Desktop/GIT/autonomous_platform_generation_4/Hardware_Interface_Low_Level_Computer
ap4@ap4-hw-interface-rpi4:~$ cd Desktop/
ap4@ap4-hw-interface-rpi4:~/Desktop$ cd GIT/
ap4@ap4-hw-interface-rpi4:~/Desktop/GIT$ ls
autonomous_platform_generation_4  xone
ap4@ap4-hw-interface-rpi4:~/Desktop/GIT$ cd autonomous_platform_generation_4/
ap4@ap4-hw-interface-rpi4:~/Desktop/GIT/autonomous_platform_generation_4$ cd Hardware_Interface_Low_Level_Computer/
ap4@ap4-hw-interface-rpi4:~/Desktop/GIT/autonomous_platform_generation_4/Hardware_Interface_Low_Level_Computer$ docker-compose build
Building ros2
[+] Building 17.5s (5/12)
=> [internal] load: dockerignore                                0.1s
=> => transferring context: 28                                  0.0s
=> [internal] load build definition from Dockerfile             0.2s
=> => transferring dockerfile: 1.63kB                          0.0s
=> [internal] load metadata for docker.io/library/ros:humble-ros-core-j 10.2s
=> [1/9] FROM docker.io/library/ros:humble-ros-core-jammy@sha256:ccd1fb 4.4s
=> => resolve docker.io/library/ros:humble-ros-core-jammy@sha256:ccd1fb 0.2s
=> => sha256:7b415d51f1ac13111c78516191c20ceefc8fa19e287 4.63kB / 4.63kB 0.0s
=> => sha256:9e3c85a1e52ef89307c50f027200f0e99d4ff65f 28.39MB / 28.39MB 0.0s
=> => sha256:0e9e5d7e79405f9e0d5346e8304f49c730e0d784a39 3.80MB / 3.80MB 0.0s
=> => sha256:341a3b77ddb45ba0b2711355c528b654f522ef0e0f 1.99kB / 1.99kB 0.0s
=> => sha256:ccd1fb374f78cc512f99c1d5eda4cfd1e29f46eae2a6 549B / 549B 0.0s
=> => sha256:5f0465bec7ec7432ea043f6810f89f5d09acd6e66a 1.79kB / 1.79kB 0.0s
=> => sha256:9aeb23cedad8021e5f112c4ebbb079853902193679 1.21MB / 1.21MB 0.0s
=> => sha256:61556f0e2363012e91a6b26a6d4854861844ae4d453b2e 230B / 230B 0.0s
=> => sha256:6f8d16cf4026b88f51269b779ec210c233b939c 104.13MB / 104.13MB 0.0s
=> => sha256:108916e1248957cb4be1672c1abf8182f1ccc1b5b3cc48 196B / 196B 0.0s
=> ERROR [2/9] RUN apt-get update && apt-get install --no-install-recomm 0.7s
-----
> [2/9] RUN apt-get update && apt-get install --no-install-recommends -y build-essential git python3-colcon-common-extensions python3-colcon-mixin python3-rosdep python3-vcstool
      | iputils-ping net-tools wget screen nano vim htop alsa-base alsa-utils can-utils && rm -rf /var/lib/apt/lists/*:
-----
WARNING: buildx failed to get git commit: error: object file .git/objects/bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f is empty
error: object file .git/objects/bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f is empty
fatal: loose object bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f (stored in .git/objects/bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f) is corrupt
dockerfile:4
-----
3 | # install bootstrap tools
4 | >>> RUN apt-get update && apt-get install --no-install-recommends -y \
5 | >>> build-essential \
6 | >>> git \
7 | >>> python3-colcon-common-extensions \
8 | >>> python3-colcon-mixin \
9 | >>> python3-rosdep \
10 | >>> python3-vcstool \
11 | >>> iputils-ping \
12 | >>> net-tools \
13 | >>> wget \
14 | >>> screen \
15 | >>> nano \
16 | >>> vim \
17 | >>> htop \
18 | >>> alsa-base \
19 | >>> alsa-utils \
20 | >>> can-utils \
21 | >>> && rm -rf /var/lib/apt/lists/*
22 |
-----
ERROR: failed to solve: process "/bin/sh -c apt-get update && apt-get install --no-install-recommends -y build-essential git python3-colcon-common-extensions python3-colcon-mixin python3-rosdep python3-vcstool |iputils-ping net-tools wget screen nano vim htop alsa-base alsa-utils can-utils && rm -rf /var/lib/apt/lists/*" did not complete successfully: error creating overlay mount to /var/lib/docker/overlay2/5h9kqp2wlaem94lgush13etf/merged: too many levels of symbolic links
ap4@ap4-hw-interface-rpi4:~/Desktop/GIT/autonomous_platform_generation_4/Hardware_Interface_Low_Level_Computer$
```

This could occur because a specific docker layer could not be built. Try pruning the existing docker caches and rebuilding. (This will take some time on Raspberry Pi hardware, enter the commands and grab some coffee!!) Rebuilding a docker container without any saved caches takes about 10 minutes.

```
docker system prune
```

```
docker-compose build
```

Another error that has previously occurred can be seen in the image below.

6.1.3 Testing and Debugging Components

2. Make sure CAN bus messages are sent and received on linux SocketCAN interface On the raspberry Pi desktop. Open a new terminal and dump CAN messages

```
candump can0
```

Expected output (or similar). The import thing is that can messages are received.

```
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 5DC [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
can0 7D0 [8] 00 00 00 00 00 00 00 00 00
```

If nothing shows up, it means that there is either something wrong with the CAN bus hardware, or in the low-level software. If the command could not be run, it means that the startup service did not complete and initialize the right commands. (such as `sudo ip link set can0 up type can bitrate 1000000`) Try rerunning the startup service script by restarting the service.

```
sudo systemctl daemon-reload
```

```
sudo systemctl enable startup_lowlevel.service
```

The next step is to enter the low level software and rerunning the same command. The output should be the same as above. If this it works it means that the linuxsocketcan has been passed through properly.

```
docker exec -it ap4hwi bash
```

```
candump can0
```

3. Check that the ROS2 nodes have been launched correctly

First enter the docker hwi container environment

```
docker exec -it ap4hwi bash
```

Secondly, enter the correct directory, source environment variables and set the correct environment variables

```
cd ap4hwi_ws
source install/setup.bash
export ROS_DOMAIN_ID=1
```

To check which ROS2 nodes are up and running, enter

```
ros2 node list
```

The expected output (as of 22/8-2023):

```
root@ap4-hw-interface-rpi4: ~/ap4_hwi_docker_dir/ap4hwi_ws
ap4@ap4-hw-interface-rpi4:~$ docker exec -it ap4hwi bash
root@ap4-hw-interface-rpi4:~/ap4_hwi_docker_dir# cd ap4hwi_ws/
root@ap4-hw-interface-rpi4:~/ap4_hwi_docker_dir/ap4hwi_ws# source install/setup.bash
root@ap4-hw-interface-rpi4:~/ap4_hwi_docker_dir/ap4hwi_ws# export ROS_DOMAIN_ID=1
root@ap4-hw-interface-rpi4:~/ap4_hwi_docker_dir/ap4hwi_ws# ros2 node list
root@ap4-hw-interface-rpi4:~/ap4_hwi_docker_dir/ap4hwi_ws# ros2 node list
WARNING: Be aware that are nodes in the graph that share an exact name, this can have unintended side effects.
/can_ros2_interface_node
/feed_forward_ctrl_node
/joy_node
/joy_node
/launch_ros_201
/socket_can_receiver
/socket_can_sender
/teleop_twist_joy_node
root@ap4-hw-interface-rpi4:~/ap4_hwi_docker_dir/ap4hwi_ws#
```

If the correct ROS2 nodes have not been started, check that the container built successfully. Try rebuilding the container manually. Make sure the Raspberry Pi has internet access.

IF the container has started properly yet no ROS2 node is up, it can mean that the ROS2 software packages could not be built. To check this, try building them manually and running the ROS2 launch file.

Enter the docker container

```
docker exec -it ap4hwi bash
```

Navigate into the ROS2 workspace

```
cd ap4hwi_ws
```

Source ROS2 environment

```
source /opt/ros/humble/setup.bash
```

Build the low level software ROS2 components

```
colcon build
```

It will explicitly output in the terminal how many packages succeeded / failed. A C / Python output will be noted in the terminal. If any of packages failed, resolve the issue which is presented in the terminal.

Source new ROS2 environment

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

set environment variable

```
EXPORT ROS_DOMAIN_ID=1
```

Run the ROS2 launch file which starts up the software nodes

```
ros2 launch launch_hwi_software_pkg launch_hwi_ros_software.launch.py
```

The output should look something similar to this once the startup script has been run:

```

root@ap4-hw-interface-rpi4:~/ap4_hwi_docker_dir/ap4hwi_ws# ros2 launch launch_hwi_software_pkg launch_hwi_ros_software.launch.py
[INFO] [launch]: All log files can be found below /root/.ros/log/2023-08-22-13-06-58-627599-ap4-hw-interface-rpi4-305
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [socket_can_receiver_node_exe-1]: process started with pid [317]
[INFO] [socket_can_sender_node_exe-2]: process started with pid [319]
[INFO] [can_ros2_interface-3]: process started with pid [321]
[INFO] [joy_node-4]: process started with pid [323]
[INFO] [joy_node-5]: process started with pid [325]
[INFO] [teleop_node-6]: process started with pid [327]
[INFO] [feed_forward_node-7]: process started with pid [331]
[socket_can_receiver_node_exe-1] [INFO] [1692709620.657192411] [socket_can_receiver]: interface: can0
[socket_can_receiver_node_exe-1] [INFO] [1692709620.657567023] [socket_can_receiver]: interval(s): 0.010000
[socket_can_receiver_node_exe-1] [INFO] [1692709620.657666597] [socket_can_receiver]: use bus time: 0
[teleop_node-6] [INFO] [1692709621.055040631] [TeleopTwistJoy]: Teleop enable button 2.
[teleop_node-6] [INFO] [1692709621.059308988] [TeleopTwistJoy]: Turbo on button 5.
[teleop_node-6] [INFO] [1692709621.063947309] [TeleopTwistJoy]: Linear axis x on 1 at scale 0.700000.
[teleop_node-6] [INFO] [1692709621.067833722] [TeleopTwistJoy]: Turbo for linear axis x is scale 1.500000.
[teleop_node-6] [INFO] [1692709621.067896555] [TeleopTwistJoy]: Angular axis yaw on 0 at scale 0.400000.
[teleop_node-6] [INFO] [1692709621.067923740] [TeleopTwistJoy]: Turbo for angular axis yaw is scale 1.000000.
[socket_can_sender_node_exe-2] [INFO] [1692709621.099126670] [socket_can_sender]: interface: can0
[socket_can_sender_node_exe-2] [INFO] [1692709621.099690059] [socket_can_sender]: timeout(s): 0.010000

```

Try restarting the container and verify that the corrected nodes has been started successfully as described above.

VERY IMPORTANT: MAKE SURE THAT THE hwi_startup.bash script is actually run when the container is spun up. Look into the docker-compose file in `Hardware_Interface_Low_Level_Computer` directory.

There is a command argument. command: xxxxxx

```

!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MAKE SURE THAT #command: /bin/bash -c "./hwi_startup.bash" IS UNCOMMENTED
!!!!!!!!!!!!!!!!!!!!!!!!!!!!

```

Whilst debugging code it can be useful to comment this line out but the software will not start up when the container is started.

This is the expected output when the container is manually started:

```

ap4hwi@ap4-hw-interface-rpi4:~/Desktop/GIT/autonomous_platform_generation_4/Hardware_Interface_Low_Level_Computer$ docker-compose up
Recreating ap4hwi ... done
Attaching to ap4hwi
ap4hwi | Running Hardware Interface docker startup bash script
ap4hwi | Entering ap4hwi_ws...
ap4hwi | Sourcing ros humble...
ap4hwi | Building ros packages...
ap4hwi | WARNING: Package name "AP4_init_test_package" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits, underscores, and dash
es.
ap4hwi | Starting >>> can_msgs_to_ros2_topic_pkg
ap4hwi | Starting >>> xbox_controller_feed_forward_ctrl_pkg
ap4hwi | Starting >>> AP4_init_test_package
ap4hwi | Finished <<< can_msgs_to_ros2_topic_pkg [2.94s]
ap4hwi | --- stderr: xbox_controller_feed_forward_ctrl_pkg
ap4hwi | /usr/lib/python3/dist-packages/setuptools/command/install.py:34: SetuptoolsDeprecationWarning: setup.py install is deprecated. Use build and pip and other standards-based tools.
ap4hwi |   warnings.warn(
ap4hwi | ---
ap4hwi | Finished <<< xbox_controller_feed_forward_ctrl_pkg [5.45s]
ap4hwi | Starting >>> launch_hwi_software_pkg
ap4hwi | --- stderr: AP4_init_test_package
ap4hwi | /usr/lib/python3/dist-packages/setuptools/command/install.py:34: SetuptoolsDeprecationWarning: setup.py install is deprecated. Use build and pip and other standards-based tools.
ap4hwi |   warnings.warn(
ap4hwi | ---
ap4hwi | Finished <<< AP4_init_test_package [5.50s]
ap4hwi | Finished <<< launch_hwi_software_pkg [0.04s]
ap4hwi |
ap4hwi | Summary: 4 packages finished [6.86s]
ap4hwi | 2 packages had stderr output: AP4_init_test_package xbox_controller_feed_forward_ctrl_pkg
ap4hwi | Sourcing built packages...
ap4hwi | Setting ROS_DOMAIN_ID=1...
ap4hwi | Running launch_hwi_software_pkg launch file...
ap4hwi | [INFO] [launch]: All log files can be found below /root/.ros/log/2023-08-22-13-14-02-855108-ap4-hw-interface-rpi4-200
ap4hwi | [INFO] [launch]: Default logging verbosity is set to INFO
ap4hwi | [INFO] [socket_can_receiver_node_exe-1]: process started with pid [212]
ap4hwi | [INFO] [socket_can_sender_node_exe-2]: process started with pid [214]
ap4hwi | [INFO] [can_ros2_interface-3]: process started with pid [216]
ap4hwi | [INFO] [joy_node-4]: process started with pid [218]
ap4hwi | [INFO] [joy_node-5]: process started with pid [220]
ap4hwi | [INFO] [teleop_node-6]: process started with pid [222]
ap4hwi | [INFO] [feed_forward_node-7]: process started with pid [239]
ap4hwi | [socket_can_sender_node_exe-2] [INFO] [1692710044.839094813] [socket_can_sender]: interface: can0
ap4hwi | [socket_can_sender_node_exe-2] [INFO] [1692710044.839422017] [socket_can_sender]: timeout(s): 0.010000
ap4hwi | [socket_can_receiver_node_exe-1] [INFO] [1692710044.942744781] [socket_can_receiver]: interface: can0
ap4hwi | [socket_can_receiver_node_exe-1] [INFO] [1692710044.957695319] [socket_can_receiver]: interval(s): 0.010000
ap4hwi | [socket_can_receiver_node_exe-1] [INFO] [1692710044.963415567] [socket_can_receiver]: use bus time: 0
ap4hwi | [Teleop_node-6] [INFO] [1692710045.270019780] [TeleopTwistJoy]: Teleop enable button 2.
ap4hwi | [Teleop_node-6] [INFO] [1692710045.270329058] [TeleopTwistJoy]: Turbo on button 5.
ap4hwi | [Teleop_node-6] [INFO] [1692710045.270366299] [TeleopTwistJoy]: Linear axis x on 1 at scale 0.700000.
ap4hwi | [Teleop_node-6] [INFO] [1692710045.270400836] [TeleopTwistJoy]: Turbo for linear axis x is scale 1.500000.
ap4hwi | [Teleop_node-6] [INFO] [1692710045.270426521] [TeleopTwistJoy]: Angular axis yaw on 0 at scale 0.400000.
ap4hwi | [Teleop_node-6] [INFO] [1692710045.270449058] [TeleopTwistJoy]: Turbo for angular axis yaw is scale 1.000000.

```

4. Check communication between low-level-software and SPCU

If the expected ROS2 nodes have been started correctly and the platform still cannot be controlled. Verify that the velocity command from `\cmd_vel` topic is being published on.

```
ros2 topic echo /cmd_vel
```

Next, verify that the vehicle controller is outputting desired set voltages for pedals and steering

Next Verify that the SPCU software and low-level software can communicate. The low-level software can request a ping signal from the ECUs on one topic and receive ping confirmations on another. The ping will be sent over the CAN

6.2 Test and Debugging of Hardware Interface Low Level Computer

This document aims to describe how to test and debug the low level software if any error occurs. If you stumble upon a new error which has been described below, please add it to the documentation and how you resolved it. It will help future project members.

6.2.1 Raspberry Pi 4b Bootup failed

The Raspberry Pi 4b can fail to boot-up if there are corrupted files on the micro SD card or if the micro SD card has run out of empty storage. The docker temporary files can grow very large in size.

This is a known issue and is noted down in `ISSUES_AND_FUTURE_WORK.md`.

Connect a display, mouse and keyboard and resolve issues presented in terminal. You probably have to clean something in the filesystem.

If this fails, there is a backup micro SD card containing a copy of the system available [here](#) for download. Make sure you have permission to access this project on Infotiv sharepoint. If this does not work, the procedure of setting up a fresh Raspberry Pi 4b is described in `SETUP_OF_RASPBERRY_PI.md` located in this directory.

6.2.2 Software container not started?

Has the Raspberry Pi 4b booted up without automatically starting the low level software docker container?

There could be several causes for this, some of them are;

- Linux startup service did not run during boot-up
- docker container could not be built due to;
 - Lack of Internet connection
 - Errors in dockerfile configuration
- Corrupted files on micro SD card.

First, check if the container is running or not using the following command.

```
docker container ps
```

If the docker container is running, the expected output would be:

```
CONTAINER ID   IMAGE                                COMMAND                  CREATED          STATUS          PORTS          NAMES
xxxx          hardware_interface_low_level_computer_ros2  xxxxx                  10 seconds ago  Up 10 seconds  ap4hwi
```

If the docker container is not running there could be two causes:

- The container is still being built (Can take up to 10 minutes when building from scratch)
- The container failed to build

Either of these alternatives can be verified by looking at the status of the startup service. In a terminal check the status of the service.

```
sudo systemctl status startup_lowlevel.service
```

output:

Solution: Graphics permission have not been set, the docker container is not allowed to display GUI elements. Run this command from a terminal outside of the container and restart the docker container

```
xhost +local:*
```

6.4 Known issues

6.4.1 Priority : Fix known Bugs / Issues

There are some known bugs and issues and should have priority

- Battery voltage drops during load and steering control is lost
- Lost wifi connection during test day @ gokartcentralen
- Raspberry Pi 4b SD card gets filled with junk files and breaks hardware interface software.

It is a known problem that the raspberry pi 4b fills up with junk files and slows down the raspberry pi and eventually break things. Docker caches also grow in size over time. Possible solution get a bigger SD card? Find how to periodically remove junk files?

- Raspberry Pi 4b corrupted files

Over time files on the SD card gets corrupted and affects behavior. Common problems are that the docker container cannot be built. A cause for this may be the shutdown procedure of the raspberry pi. Unplugging it to power down may not be optimal. Need to find a solution for properly safely powering down the raspberry pi. Terminal output below.

```
ap4@ap4-hw-interface-rpi4: ~/Desktop/GIT/autonomous_platform_generation_4
ap4@ap4-hw-interface-rpi4:~$ cd Desktop/
ap4@ap4-hw-interface-rpi4:~/Desktop$ cd GIT/
ap4@ap4-hw-interface-rpi4:~/Desktop/GIT$ cd autonomous_platform_generation_4/
ap4@ap4-hw-interface-rpi4:~/Desktop/GIT/autonomous_platform_generation_4$ git pull
error: object file .git/objects/bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f is empty
error: object file .git/objects/bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f is empty
error: object file .git/objects/bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f is empty
error: object file .git/objects/bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f is empty
error: object file .git/objects/bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f is empty
error: object file .git/objects/bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f is empty
remote: Enumerating objects: 173, done.
remote: Counting objects: 100% (101/101), done.
remote: Compressing objects: 100% (91/91), done.
error: object file .git/objects/bf/b9f76e714cefbadb45fa9e2efa9863b5afb02f is empty
fatal: cannot read existing object info bfb9f76e714cefbadb45fa9e2efa9863b5afb02f
fatal: fetch-pack: invalid index-pack output
ap4@ap4-hw-interface-rpi4:~/Desktop/GIT/autonomous_platform_generation_4$
```

See TEST_DEBUGGING.md to see how it is currently resolved.

- Low voltage input to the raspberry pi makes it slow and unresponsive

Needs to be resolved properly. Could become a problem once more data is sent to and from the hardware interface code. * PlatformIO build fails when repository is cloned to a path containing a whitespace

Opening the embedded ECU software in platformIO and trying to build it fails when repository filepath has whitespace in it. See error output below. This was when trying to build firmware for SPCU.

Filepath to SPCU was C:\Users\erimag\OneDrive - Infotiv AB\Skribbordet\git\autonomous_platform\CAN_Nodes_Microcontroller_Co

```
Linking .pio\build\bluepill_f103c8\firmware.elf
arm-none-eabi-g++: error: Infotiv: No such file or directory
arm-none-eabi-g++: error: AB/Skrivbordet/git/autonomous_platform/CAN_Nodes_Microcontroller_Code/SPCU/.pio/build/bluepill_f
[.pio\build\bluepill_f103c8\firmware.elf] Error 1
```

I tried the solution presented in this [forum post](#) but could still not get it to work. Temporary solution, clone the repository to a filepath without whitespace?

If there is any error related to long path, open shell as administrator, configure git to use long path names, or else some files will be missed when cloning the repository

```
git config --system core.longpaths true
```

6.5 Future work

As the thesis of 2023 concluded there several functionalities that could not be implemented due to time restrictions. Therefore for future work we suggest:

6.5.1 Improving Generic ECU nodes with PCB

The generic ECU base hardware should be looked over and iterated over. The I/O proposed should be kept the same to keep compatibility with the rest of the system.

Many loose / Unlabeled Wires coming out from the back of ECU. Hard to follow existing implementation. Cables can be shock loosed by accident very easily. - Possible solution: Add a 'Phoenix' style connector to the back of the ECU. Standardized connector type and easily available to buy. Would allow for ECU to be easily removed. A specific phoenix connector which fits into the existing size constraints of the ECU needs to be found. Existing CAD model of ECU HW Node needs to be adjusted accordingly. Existing nodes should be upgrades first. Tommy at Infotiv has a product manual with different types of Phoenix Connectors which can be ordered by Infotiv.



6.5.2 Power module

By Power Module it is referring to the battery and Power Supply Unit (PSU).

The power module created during the thesis is sufficient for testing AP4 without anyone sitting in the gokart. During the test day at gokartcentralen (Spring 2023) it was found that the friction of the road surface made the power module insufficient when sitting in the gokart and running from battery power. The power spiked and blew a fuse. On a more slippery surface, such as in the office one could sit in the gokart and drive around without problem.

The voltage from the battery is not regulated, meaning that when the battery depletes the voltage to all the components decreases. Ideally the voltage to the system should remain at a constant 12v.

Current issues with Power Module:

Voltage to the AP4 components drops as battery SoC drops. Could be a cause to unexplained behavior during test day. I.e wifi router stopped working after 3 hours. - Possible solution: Install a voltage regulator coming out from the battery such that the output voltage from the power module is at a constant 12v regardless of the SoC of battery.

Monitoring power drain during runtime of platform. - Possible solution: Connect the Voltage sensors and Current Sensor to a ECU and send information to the Central Computer. The hardware for this is installed to battery inside the power module box but has not been connected. By reading voltage and current consumption continuously it could provide useful information to the tests performed and send an alarm when the voltage of the battery gets too low.

Analysis of power consumption of platform is insufficient. Platform blew a fuse unexpectedly whilst sitting in the gokart on a rough surface and turning the steering. The existing analysis (22/8-23) can be found in the thesis report in the 'Images' folder. - Possible solution: Do a new power requirement analysis of the platform. Taking current spikes into account. Implement changes to the platform to avoid future problems.

6.5.3 High Level Control Software

The high level control software is very much unfinished. So far a docker container has been setup with the correct environment and software installed.

For future work regarding the digital twin, see **Digital Twin** section below.

6.5.4 Digital Twin

The digital twin implemented in spring 2023 is a simple twin created in Gazebo and not very well tuned. The movement dynamics do not behave as the physical AP4 platform does. This needs to be tuned. The dynamics of the physical platform should be investigated, noted down and transferred to the digital twin.

Investigating integration of AP4 into CARLA or other simulations could also be done. As long as the simulation software can integrate with Robot Operating System 2 it should work.

Infotiv already has experience using CARLA for vehicle scenario simulation. Therefore to ease future development of digital twin it could be interesting to transfer the digital twin to the CARLA simulator instead of Gazebo. - Possible solution: [ROS bridge installation for ROS2 - Carla Documentation](#) Note: This is for a previous version of ROS2, AP4 builds on ROS2 Humble so a fix for this needs to be found.

6.5.5 Mounting High Performance Computer on AP4

During spring 2023, the high performance computer was specified as the development laptop. It was sufficient for testing and connecting wirelessly to the hardware interface (Raspberry Pi 4b). When future AD/ADAS algorithms gets implemented it would be beneficial to have this computing unit mounted permanently onto the platform. This would remove any latency / range issues caused by the platform being controlled over wifi.

Find a suitable Intel NUC mini PC and mount onto the back of Autonomous Platform Generation 4. Install the software currently running on the development laptop onto it.

6.5.6 Sensor Measurements

Any Autonomous Drive (AD) or Advanced Driver Assistance System (ADAS) needs sensory input in order to understand its environment and make decision on. As of (22/8-23) the platform does not have such functionality

6.5.6.1 Velocity A crucial component of any autonomous vehicle is knowing its own state, its velocity is one of such states. During July 2023 Seamus and Alexander started to work on measuring the velocity.

6.5.7 Camera

A camera sensor can be used to identify obstacles around AP4.

A USB camera (logitech c920) has been bought and mounted to the front of the platform but has not been connected. Some investigation on how to connect is has been done during spring 2023.

The USB hub needs to be connected to the Raspberry Pi 4b (Hardware interface) and onto it the camera needs to be connected. Software wise, a suitable ROS2 camera package should be installed into the low level container. It needs to be configured (through ROS2 launch scripts) to start when the ROS2 software starts up.

Possible suitable ROS2 Camera packages: https://index.ros.org/r/v4l2_camera/

The output from the camera package software would then be a video stream on a known ROS2 topic. This would then need to be processed through some Machine Learning stack to extract useful information (Detected objects, depths, distance, position on road). A ROS2 package would have to be created which listens to the video stream topic and does the computing. Providing the output from the machine learning algorithm on a new ROS2 topic.

6.5.8 IMU

An Inertial Measurement Unit (IMU) provides information such as linear acceleration and rotation. This information can be useful to determine the orientation and acceleration of the autonomous platform.

This requires an IMU sensor to be integrated onto the platform. See `HOW_TO_EXTEND.md` for general procedure of connecting a new functionality.

6.5.9 Sensor Fusion

Sensor fusion is the process of combining multiple sensor values in order to get a better approximation of the vehicle state. The information from the sensors above, Camera, Velocity, Lidar and Radar can be combined using sensor fusion algorithms in order to get a better overview of the environment around the autonomous platform.

This can be suitable as a thesis project as it is an advanced topic.

6.5.10 Testing out existing Autonomous Drive algorithms

When sensors have been integrated onto the platform and sensor data has been processed it can then be used in AD/ADAS algorithms to control the vehicle given its current state.

There exists ready solutions such as OpenPilot.

Software wise, sensor information on topics needs to be routed to the decision making software. Hamid has great knowledge on how to use OpenPilot.

This was investigated somewhat during the 2023 spring thesis, it was found that OpenPilot ran on Ubuntu 20.04 whilst the software for AP4 runs on ubuntu 22.04. Therefore ROS2 Humble could not be installed in the same container as Openpilot and communication between the decision making software and AP software could not be established.

- Possible solution: Due to time constraints this could not be investigated further during the masters thesis spring 2023. Possible solutions to the problem presented could be to install ROS2 from source in the OpenPilot docker container, or installing OpenPilot from source in a new docker container running ubuntu 22.04. This has to be investigated further before ruling OpenPilot out as a unsuitable AD software for AP4.

Other alternatives to OpenPilot should be investigated, else it could be interesting to develop an simplified autonomous drive stack in-house at Infotiv.

6.5.11 Integrating sensors used in the automotive industry

See `HOW_TO_EXTEND.md` on how to integrate sensors which needs to be connected using microcontroller.

For a list of sensors to possibly integrate see [above](#) in ‘Sensor Measurements’.

Important note added: Once embedded sensors have been implemented according to `HOW_TO_EXTEND.md` information will be available on topic `/GET___`. This information will then need to be transferred to known standard name topics to make the information flow compatible with the information sent from the digital twin.

I.e velocity commands to the platform should be sent on `/cmd_vel {x, y, z, Rx, Ry, Rz}` and needs to be converted into commands which can be sent to the platform. I.e `"/SET_0x3e8_Act_SteeringPosition"`, `"/SET_0x3e8_Act_ThrottleVoltage"` and `"/SET_0x3e8_Act_BreakVoltage"`. This is performed inside `"xbox_controller_feed_forward_ctrl_pkg"` (NOTE this package should probably be renamed). Given information on `/cmd_vel` topic, it is processed and used to control a P-Control, which in turn outputs information on the three topics that are sent to the actuators on the CAN bus.

In a similar way, future sensor information on `"/GET_<FRAME ID>_<SIG NAME>"` should be processed to output the sensor information on according to naming conventions. This would be done by creating a new package which creates a node. The node would listen to `/GET_<FRAME ID>_<SIG NAME>`, process it (scale, offset, etc) and output the new information on topic `/imu` for example. The output topic should be structured according to how common sensor packages in ROS2 does it, to keep compatibility.

And for velocity specifically, a new ROS2 package should be created, which starts a ROS2 node. It would listen to information on `/SET_0x5dc_Get_Velocity` and output onto `\odom` topic. The output should be on the `\odom` topic using `nav_msgs/msg/Odometry` type message.

6.5.12 Naming of sensor messages convention (sensor interfaces)

See [article link here](#) under “Sensor Introduction”

There exists a few common packages which specify how sensor messages should be mapped:

[Full list of ROS2 interfaces can be found here](#)

- [sensor_msgs](#) Examples: Imu, Camera, Joystick, LaserScan, etc
- [radar_msgs](#)
- [vision_msgs](#)

6.5.13 Naming of navigational messages convention (interface)

There exists corresponding interfaces (a given standard) for information sent over the ROS2 network regarding navigation.

Some navigational interfaces are:

- [nav_msgs](#) Example: Odometer interface which represents a current Pose (Position+Orientation) and Velocity The goal is to have sensor information from the physical platform have the same data structure and format as the digital twin. And by following a provided standard it allows for easier cross compatibility between created software packages and existing software packages.

See IMU sensor ROS2 topic format [here](#).

Sensors which require more data throughput, such as cameras/lidars are often bought as existing solutions and provide USB interface. These can be integrated with AP4 by connecting the sensors to the hardware interface using USB. Software wise there exists ROS2 packages for the commonly used automotive sensors which needs to be installed and configured.

6.5.14 Automatic and dynamic generation of ROS2 package ‘can_msgs_to_ros2_topic_pkg’

This is the ROS2 software package responsible for translating incoming CAN bus data on the Raspberry Pi 4b Hardware_Interface_Low_Level_Computer Linux SocketCAN into useful data on ROS2 topics. Currently it is written by hand and has to be updated every time a new frame or signal is added into the CAN_DB.dbc file for information to flow through the software layers from embedded to low level software. This is a tedious process and can cause a lot of headache if one is not familiar with C++ or the ROS framework.

Future work could therefore be to write a parser (in python?) to automatically generate this package (mainly C++ main.cpp code) using the information stored in the CAN_DB.dbc file.

The current work process is described in Hardware_Interface_Low_Level_Computer README file.

One possible issue/problem is how to select what ROS2 common datatype to use. Maybe look into <https://docs.ros.org/en/foxy/Concepts/ROS-Interfaces.html> - Look at Field types. I.e a double in C++ should be defined as a float64 in a ROS2 message.

Autogenerating this code could save a lot of time and headache (Once it has been tested and verified properly). It would also be easier for future AP4 members to add functionality. Messages would only need to be defined in the CAN_DB.dbc file and everything from the embedded structures to the ROS2 interface would be generated automatically.

6.5.15 Configure docker containers to run on Windows

As of August 2023 the docker containers for autonomous platform are configured to be started on a linux host machine. This is how they are currently configured.

- `High_Level_Control_Computer` container : Starts a graphical simulation of the digital twin. Is meant to run on any linux computer with 3D graphics capability. Is meant to be portable and be started by any member wanting to develop higher level algorithms with the option to run it without being connected to the physical platform.
- `Hardware_Interface_Low_Level_Computer` : Starts the software responsible for interfacing with and controlling the hardware platform. Is meant to be run on a Raspberry Pi 4b connected to the platform. To make future development of AD/ADAS algorithms more streamline and more accessible it would be useful to start the high level software container on Windows host machines as well.

A new configuration of the docker-compose file located at `High_Level_Control_Computer` would need to be added. Passing graphical elements out from the container works a bit different on Windows than on a linux host machine. One suggestion would be to use X11 server to display the container desktop GUI inside a web browser. See <https://medium.com/geekculture/run-a-gui-software-inside-a-docker-container-dce61771f9> for instructions regarding this option.

6.5.16 Remote flashing of ECU software over CAN bus

Suggested by Hamid during a meeting, noted down by Erik.

It is quite a bit of a headache to flash the ECUs with new software. If multiple ECUs would have to be re-flashed for some reason it can take quite some time and lead to user errors when moving cables around. When flashing using the ST-Link v3 internal power cables have to be plugged out temporary. This is cumbersome and may lead to user error.

An interesting future work would therefore be to look into flashing the ECUs over the existing CAN bus network remotely.

By flashing over CAN bus, all ECUs can be upgraded automatically when a change have to be pushed. This could in the long term save a lot of time and make the development process pipeline easier to use.

A project with good potential for this functionality can be found [here](#). It is a software that installs a custom boot-loader onto the embedded microcontroller. This needs to be investigated further to see if it is possible to flash an STM32 bluepill STM32F103C8T6 microcontroller or if can be only be done on other specific hardware.

ECUs on the CAN network could then potentially be flashed by using the sister-repository linked [here](#). This would ideally be run on the Raspberry Pi 4b hardware interface computer

The AP4 CAN bus library used for communication may have to be changed out for the one proposed on the github page mentioned above.

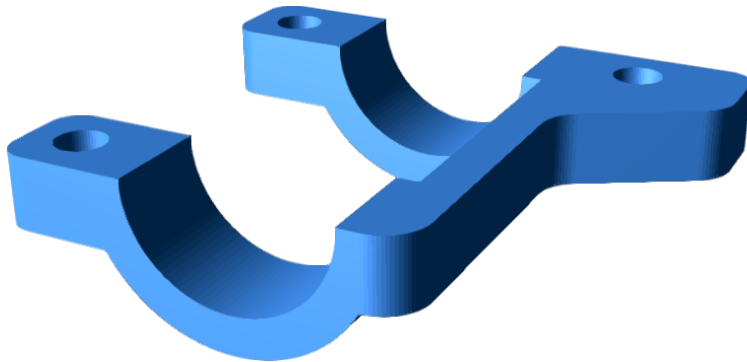
Here is a list of other similar projects, none specifically for the STM32F103xx though.

- [marcinbor85/can-prog](#) - should be looked into
- [Hoernchen20/st-can-flash](#).
- [effenco/stm32-can-bootloader](#)
- [matejx/stm32f1-CAN-bootloader](#)
- [Someone successfully implemented STM32103xx bootloader - blogpost](#)

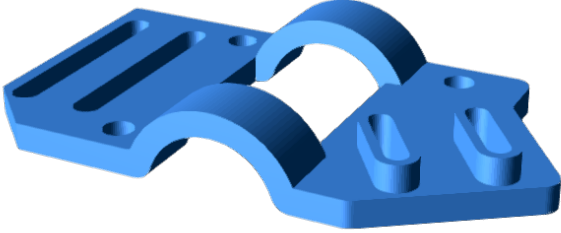
7 CAD Appendix

STL files can be opened using any 3D printer slicer software, at Infotiv we use 'Ideamaker'. [Download here](#). Use a slicer configuration for an 'Raise3d E2' 3d printer. PLA material choice is fine if you don't leave autonomous platform in a hot car during the day. A base slicer profile for the Raise3D E2 can be found in the CAD profile. It is a nice starting point.

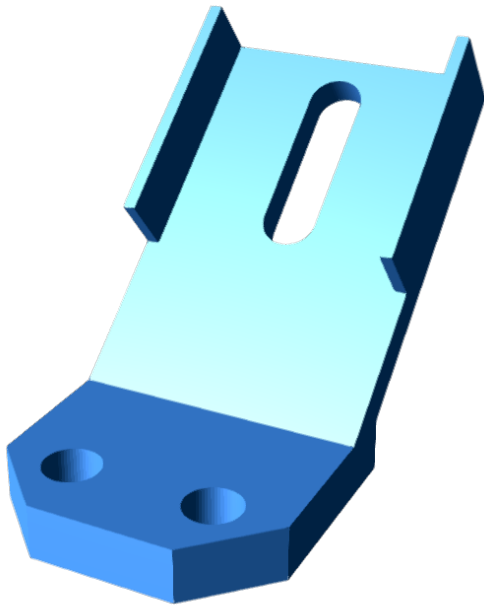
./Limitswitch_steering_holder/limittswitch_bottom_holder.stl:



./Limitswitch_steering_holder/limitswitch_holder_top.stl:



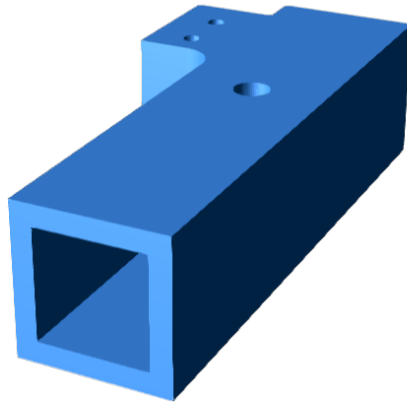
./Limitswitch_steering_holder/sensor_holder.stl:



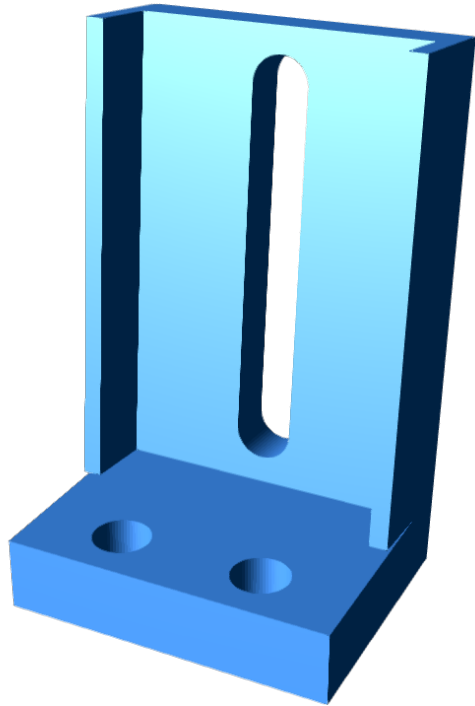
./camera_sensor_mount/camera_holder.stl:



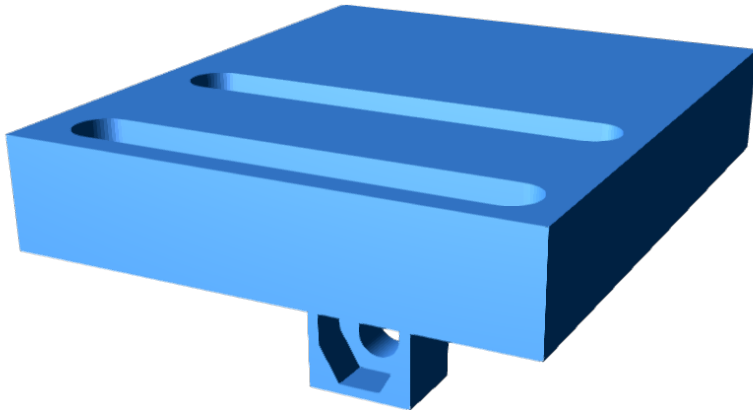
./Speed_Sensor_Back/Holder_base.stl:



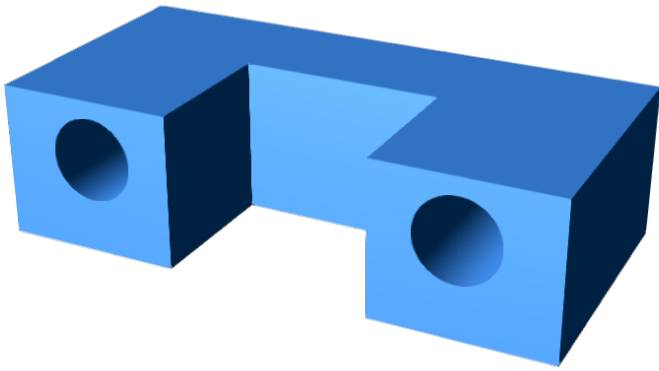
./Speed_Sensor_Back/Sensor_Holder.stl:



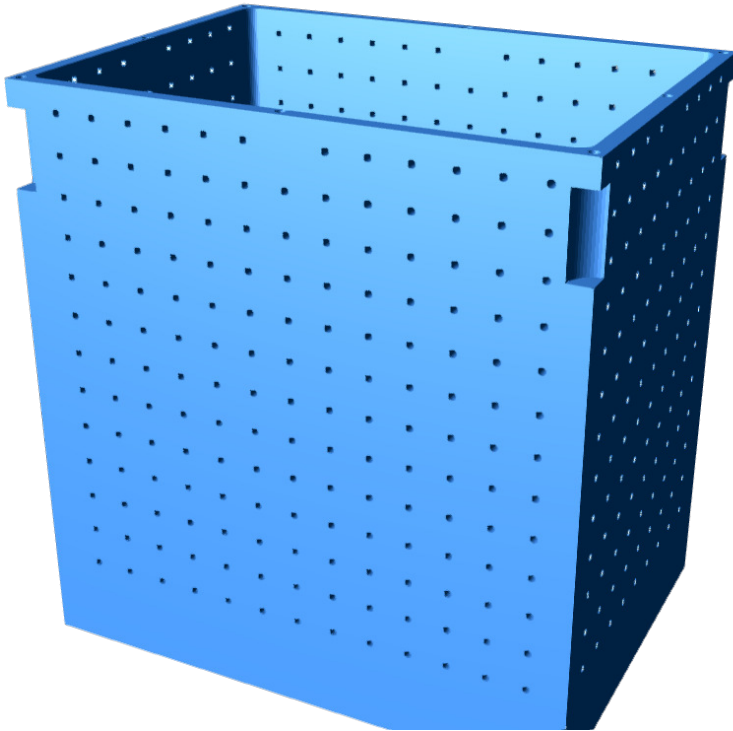
./Speed_Sensor_Back/Holder_ajustable.stl:



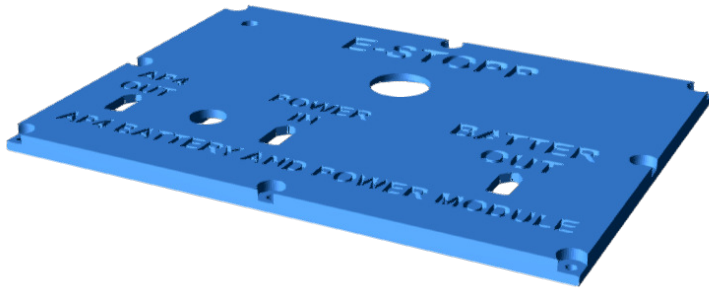
./Battery_BOX/x60_clamp.stl:



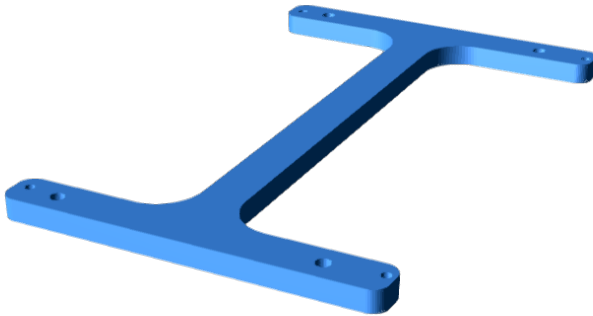
./Battery_BOX/Holder.stl:



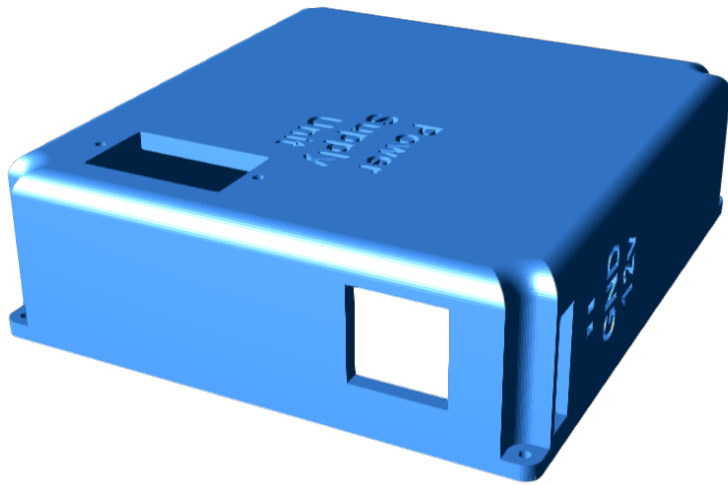
./Battery_BOX/Top.stl:



./Battery_BOX/Battery_charger_holder.stl:



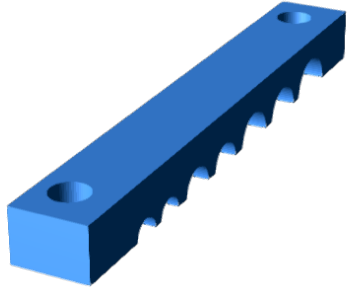
./Battery_BOX/din_rail_psu_holder.stl:



./Node_Box/Termination_Resistor_Bottom.stl:



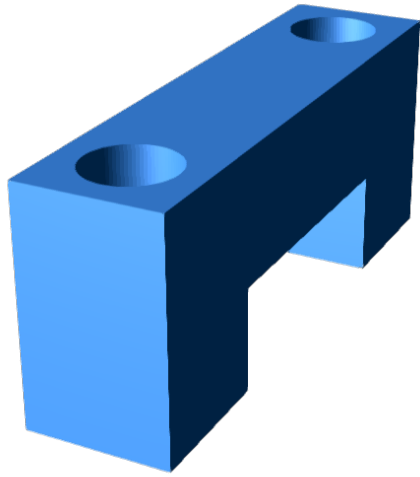
./Node_Box/cable_out_clamp.stl:



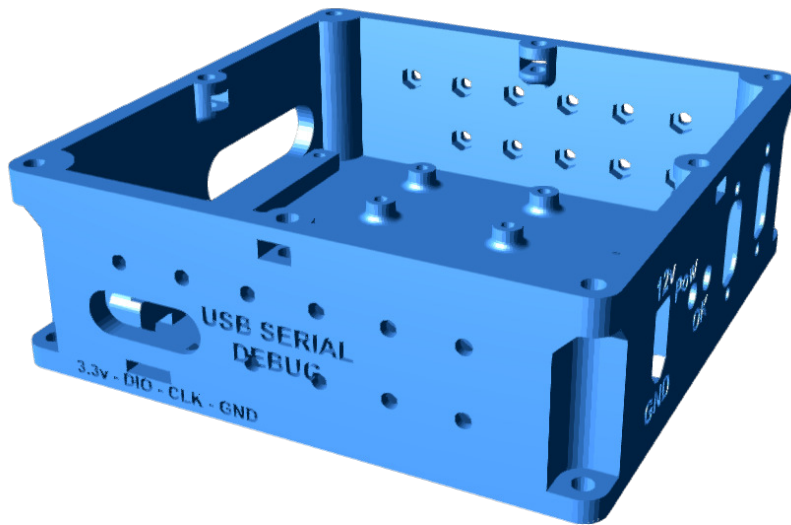
./Node_Box/ecu_nametag.stl:



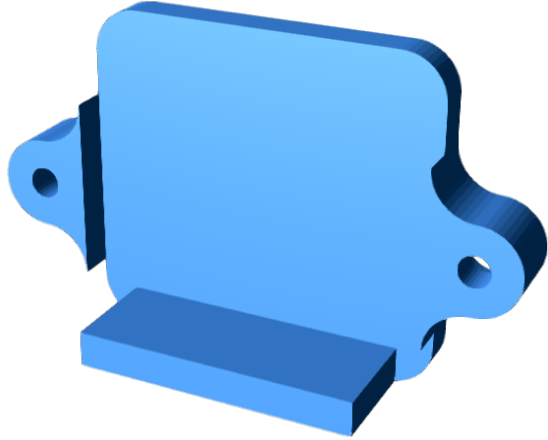
./Node_Box/xt60_clampdown.stl:



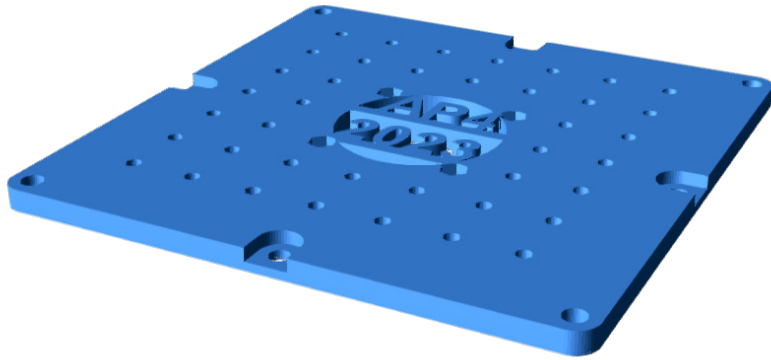
./Node_Box/HW_Node_Bottom_Box.stl:



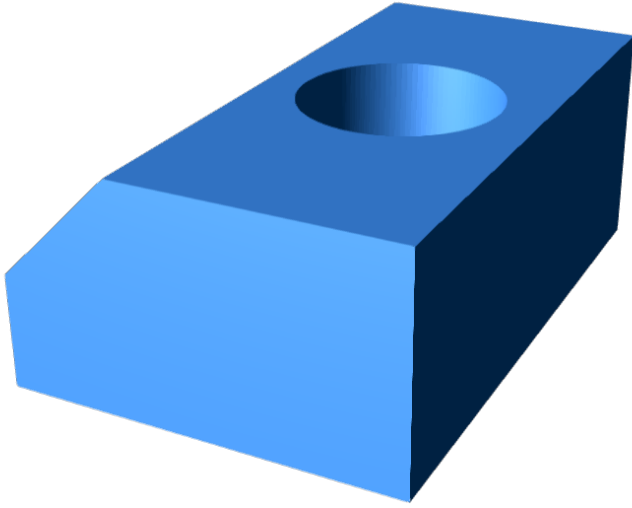
./Node_Box/Termination_Resistor_Bottom_Top.stl:



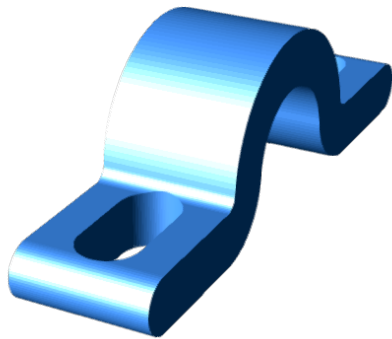
./Node_Box/HW_Node_Top_Cover.stl:



./Node_Box/STM32_bluepill_holddown.stl:



./Cable_Management/small_spiral_sleve_holder.stl:



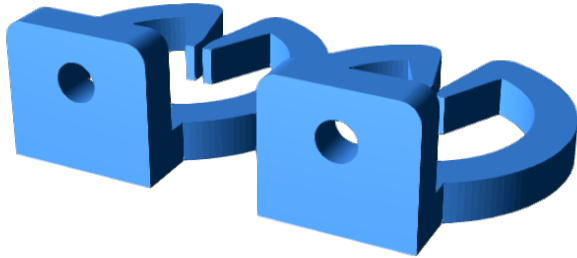
./Cable_Management/Cable_management_v2.stl:



./Cable_Management/big_spiral_holder_1.stl:



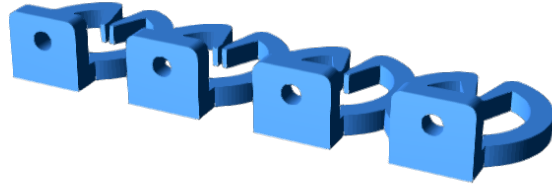
./Cable_Management/files/Cable_management2.0_2.stl:



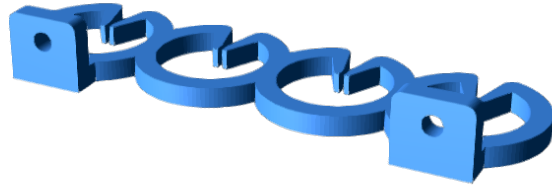
./Cable_Management/files/Cable_management2.0.stl:



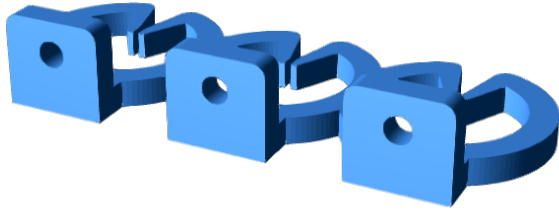
./Cable_Management/files/Cable_management2.0-4.stl:



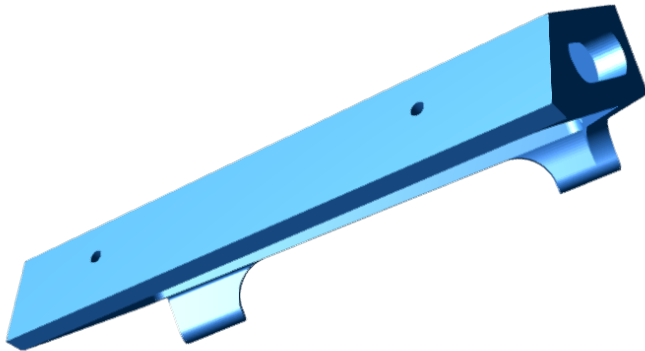
./Cable_Management/files/Cable_management2.0-4.2.stl:



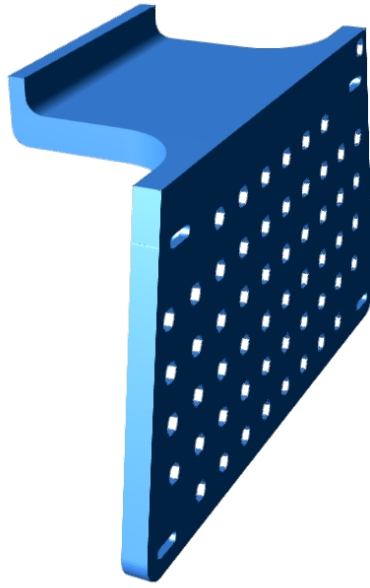
./Cable_Management/files/Cable_management2.0-3.stl:



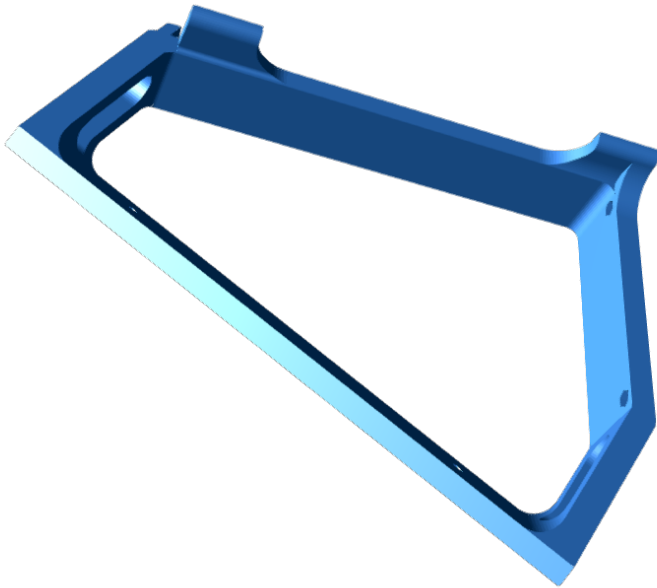
./laptop_holder/laptop_left_holder.stl:



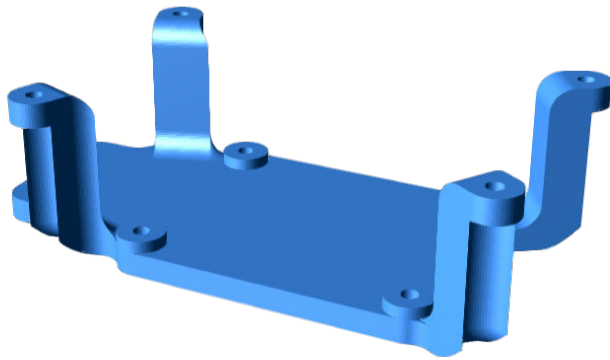
./laptop_holder/laptop_holder_hole_board.stl:



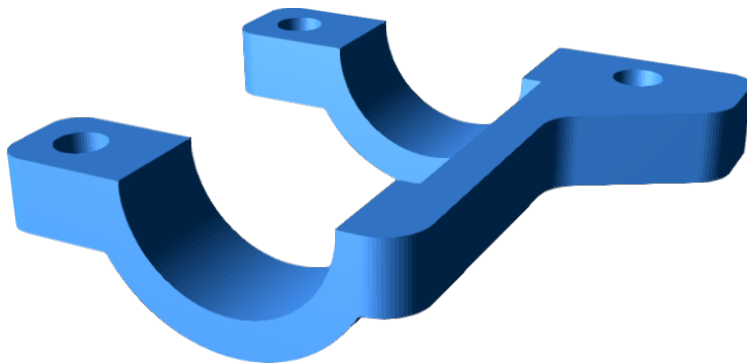
./laptop_holder/laptop_right_holder.stl:



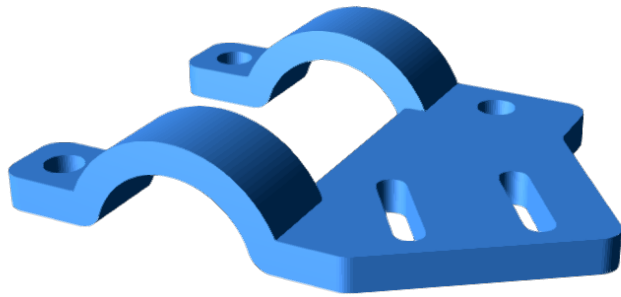
./Steering_Node_Box/sabertooth_holder.stl:



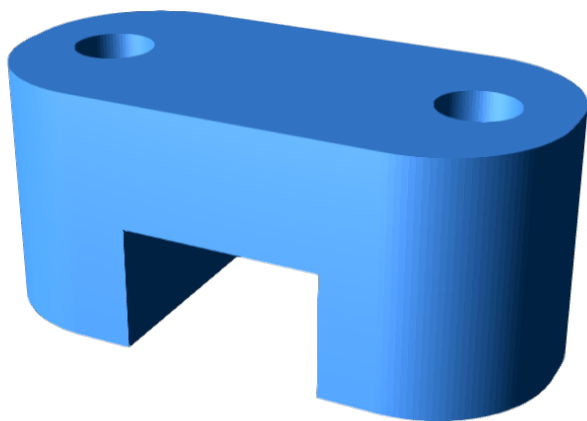
./Steering_Node_Box/limitswitch_bottom_holder.stl:



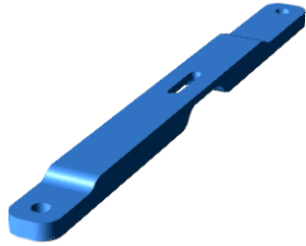
./Steering_Node_Box/limitswitch_holder_top.stl:



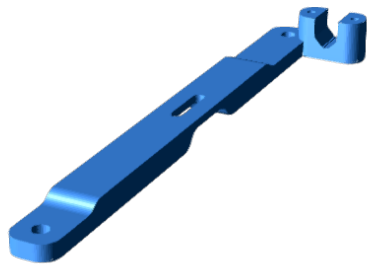
./wifi_router_holder/xt60_holder.stl:



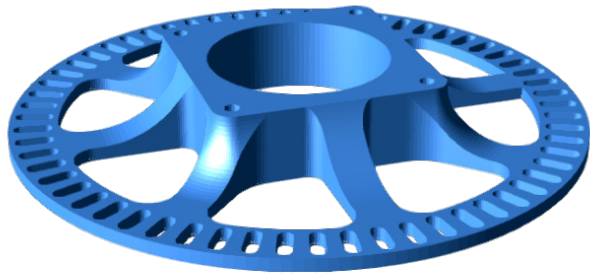
./wifi_router_holder/left_holder.stl:



./wifi_router_holder/right_holder.stl:



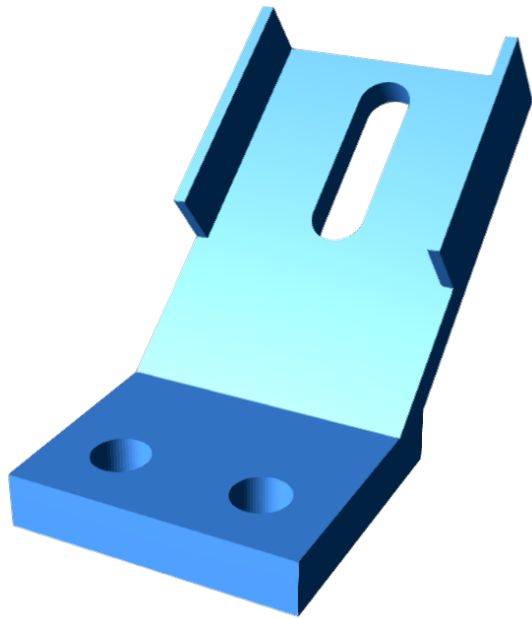
./Speed_sensor_holder/hole_wheel_speed_sensor.stl:



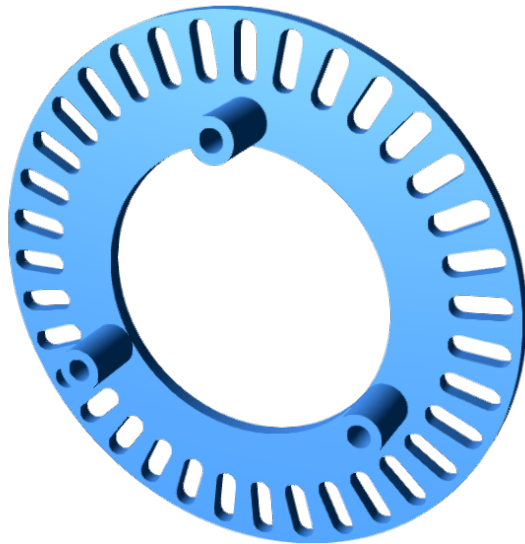
./Speed_sensor_holder/front_holder.stl:



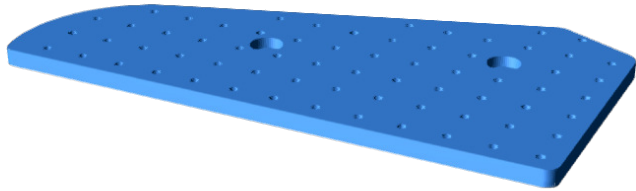
./Speed_sensor_holder/front_sensor_holder_lmitswitch.stl:



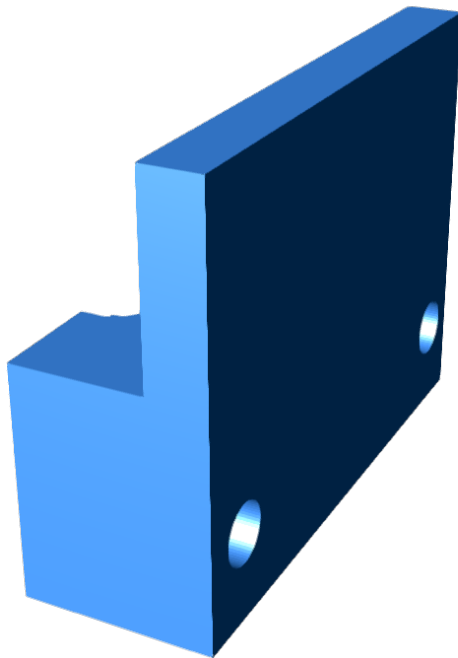
./Speed_sensor_holder/front_speed_sensor_disc.stl:



./front_win_grid/front_wing_grid.stl:



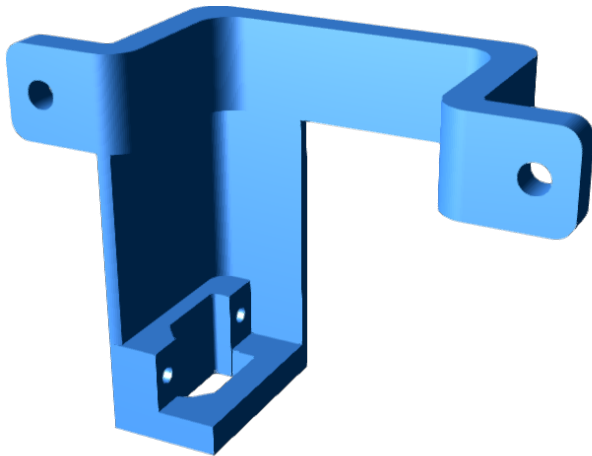
./USB_Hub_clampdown/power_connection_holddown.stl:



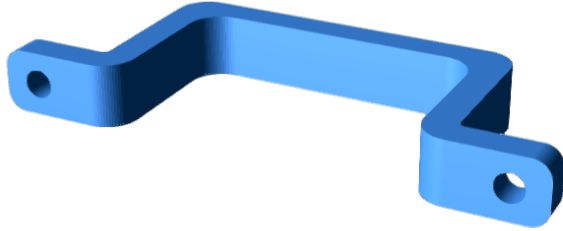
./USB_Hub_clampdown/hub_clampdown_bottom.stl:



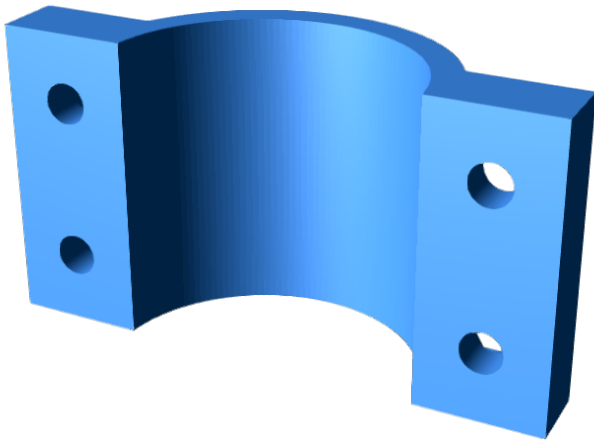
./USB_Hub_clampdown/hub_clampdown_power.stl:



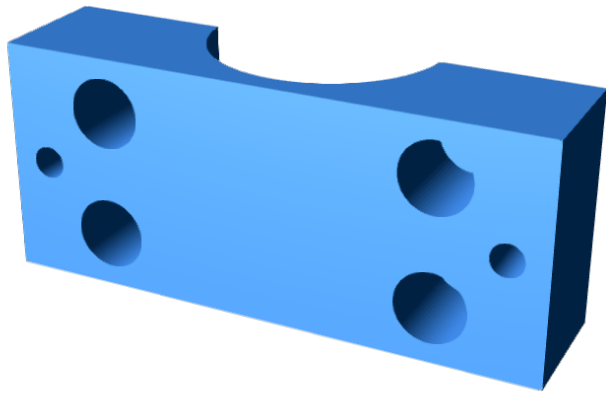
./USB_Hub_clampdown/hub_clampdown_top.stl:



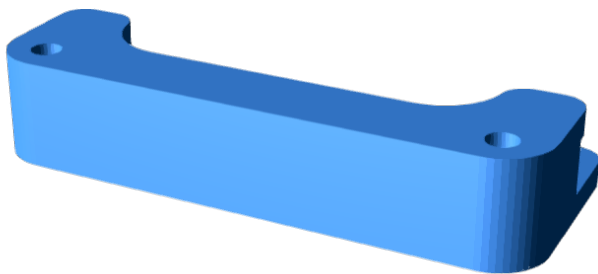
./Back_plate_holder/holder_under.stl:



./Back_plate_holder/holder_top.stl:



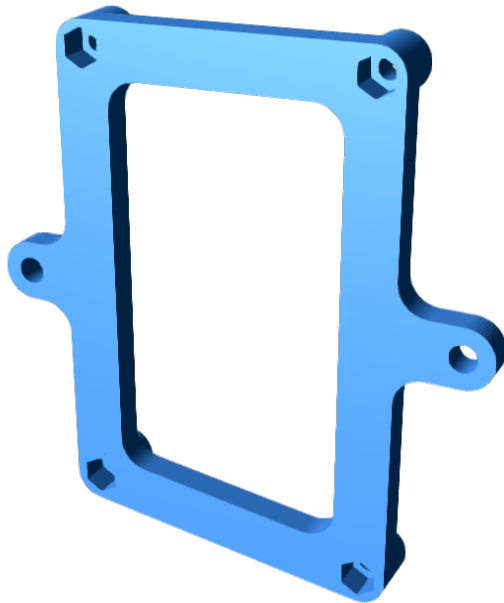
./portable_screen_holder/Screen_holder.stl:



./Arduino_Mega_holder/bottom_nut_holder.stl:



./Arduino_Mega_holder/adapter.stl:



./Mounting_Plate/plate_holder.stl:

