Zurich University
of Applied Sciences

School of
Engineering
InES Institute of
Embedded Systems

Bachelor thesis:
Formula Student: Vehicle software incl. Driverless connection

Zurich University of Applied Sciences

Institute of Embedded Systems

Authors	Marco Rau
	Tim Roos
Supervisors	Prof. Dr. Matthias Rosenthal
	Patrick Cizerl
	Jonas Koch

Date	June 6, 2024

Zürcher Hochschule
für Angewandte Wissenschaften

zh aw School of Engineering

DECLARATION OF ORIGINALITY

Bachelor thesis at the School of Engineering

Declaration of originality

I hereby declare that I have written this thesis independently or together with the listed group members.

I have only used the sources and aids (including websites and generative AI tools) specified in the text or appendix. I am responsible for the quality of the text and the selection of all content and have ensured that information and arguments are substantiated or supported by appropriate scientific sources. Generative AI tools have been summarized by name and purpose.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

Place, Date:	**Name, Signature:**
Winterthur, 06.06.2024	Tim Roos
Winterthur, 06.06.2024	Marco Rau

Abstract

The increasing awareness and interest in electric vehicles (EVs) and autonomous driving technologies present new challenges and opportunities in automotive engineering. This bachelor thesis addresses the development of a new Electronic Control Unit (ECU) system for the 2024 Formula Student racing car of Zurich University of Applied Sciences (ZHAW) Racing Team. Building upon previous projects, this work aims to create a modular, maintainable, and readable software architecture that facilitates seamless communication with both purchased and self-developed components. The new ECU system is designed to support the evolving requirements of the vehicle, ensuring compatibility and ease of integration for future developments. Key objectives include enhancing the modularity of the system, improving the maintainability of the hardware interfaces, and prioritizing code readability to aid future team members. The project also incorporates the development of a driverless functionality, expanding the capabilities of the racing car. This thesis outlines the state of the art of the 2023 and 2024 EV systems, details the conceptual and practical implementation of the ECU, and presents the results of extensive testing and performance evaluations.

Contents

1. Motivation

This bachelor thesis focuses on the development of a new Electronic Control Unit (ECU) system for the ZURICH University of Applied Sciences (UAS) Racing Team, which is the Formula Student team of the Zurich University of Applied Sciences (German: Zürcher Hochschule für Angewandte Wissenschaften) (ZHAW). This Bachelor thesis builds on a previous project entitled "Formula Student: Telemetry and Sensor" by M. Rau and T. Roos, which was carried out in the autumn semester of 2023. This project work forms the basis for the newly developed ECU and has fundamentally defined both the software architecture and the interfaces.

Formula Student is an international engineering competition in which students design, develop and race small formula cars. The focus is on the construction and design of the Electronic Vehicle (EV), encouraging innovation and the practical application of theoretical knowledge. However, the competition also includes races in which performance is tested. These races are made up of different disciplines, which evaluate different aspects of the EV. The disciplines include endurance, acceleration, skidpad and autocross, each of which presents unique challenges.
To be successful in these areas, several interdisciplinary teams must bring together their new approaches and ideas.

ZURICH UAS Racing is an organisation, in which both the team and the EV undergo annual changes. As only students are permitted to develop the EV, the time of individual team members is limited. Consequently, the team is dependent on new members. The extensive and annual reorganisation carries the risk that a significant portion of the acquired knowledge is lost when an experienced member leaves.
According to the Formula Student regulations [4], the technical development of the vehicle is a continuous task that has to be fulfilled every year. This means that a Formula Student team must develop and build a new EV every year. It is permitted to use components from the previous year. Therefore, it is particularly important to develop components on the EV in such a way that they can also be reused in future.
Since ZURICH UAS Racing was founded in 2019, the team has already taken part in three racing seasons and has therefore already built three EVs. The last EV was the one for the 2023 racing season, which had a total weight of 220 kg and was powered by four wheel hub motors. The EV's engine output totalled 140 kW, which enabled it to accelerate from 0 to 100 km/h in 2.5 seconds. [18]
Since the summer of 2023, the team of around 70 people from various study programmes has been busy building the fourth racing car for the racing season 2024 [17]. The ECU developed as part of this project will be used for the first time in this 2024 EV and should also enable driverless functionality.

The aim of this thesis is therefore to create a completely new solid software foundation for the new 2024 ECU, which should greatly simplify the future embedding of new features and optimisations. In order to achieve this goal, the software framework must both, withstand the constantly changing requirements of the vehicle. When developing the new ECU, it is crucial to ensure seamless communication with both purchased and self-developed components. The system should also allow components to be replaced easily without causing compatibility problems. However, the constant change of team members must not be neglected in the development of the new ECU software. A system that is too complex would be very difficult to adapt for new, inexperienced team members. For this reason, readability should have a higher priority than performance and ensure long-term progress.

The main focus of this project is therefore to improve modularity, maintainability and readability. Modularity facilitates system customisation without the need for a comprehensive understanding of the entire system structure. Maintainability relates to the hardware and aims to simplify the interface for customising the hardware. Readability extends throughout the whole project to ensure that future students can make system modifications with minimal background knowledge.

2. State of the art

This chapter provides an overview of the EV system of the 2023 and 2024 race car. In this context, the focus is on the components that are directly linked to the ECU. The aim is to provide insights into the planned changes and further developments that will affect the new 2024 ECU. In addition, the new 2024 ECU platform will be compared with the older 2023 ECU platform in order to assess risks and opportunities.

2.1. 2023 Electrical Vehicle

Figure 2.1 shows all Controller Area Network (CAN) bus participants of the 2023 season, which are divided into four CAN buses and a distribution box. The most important component of the EV besides the dSPACE ECU, which is discussed in the chapter 2.3, is the Distribution box, which is directly connected to the ECU. As a result, all ECU signals are directly coupled to this box.

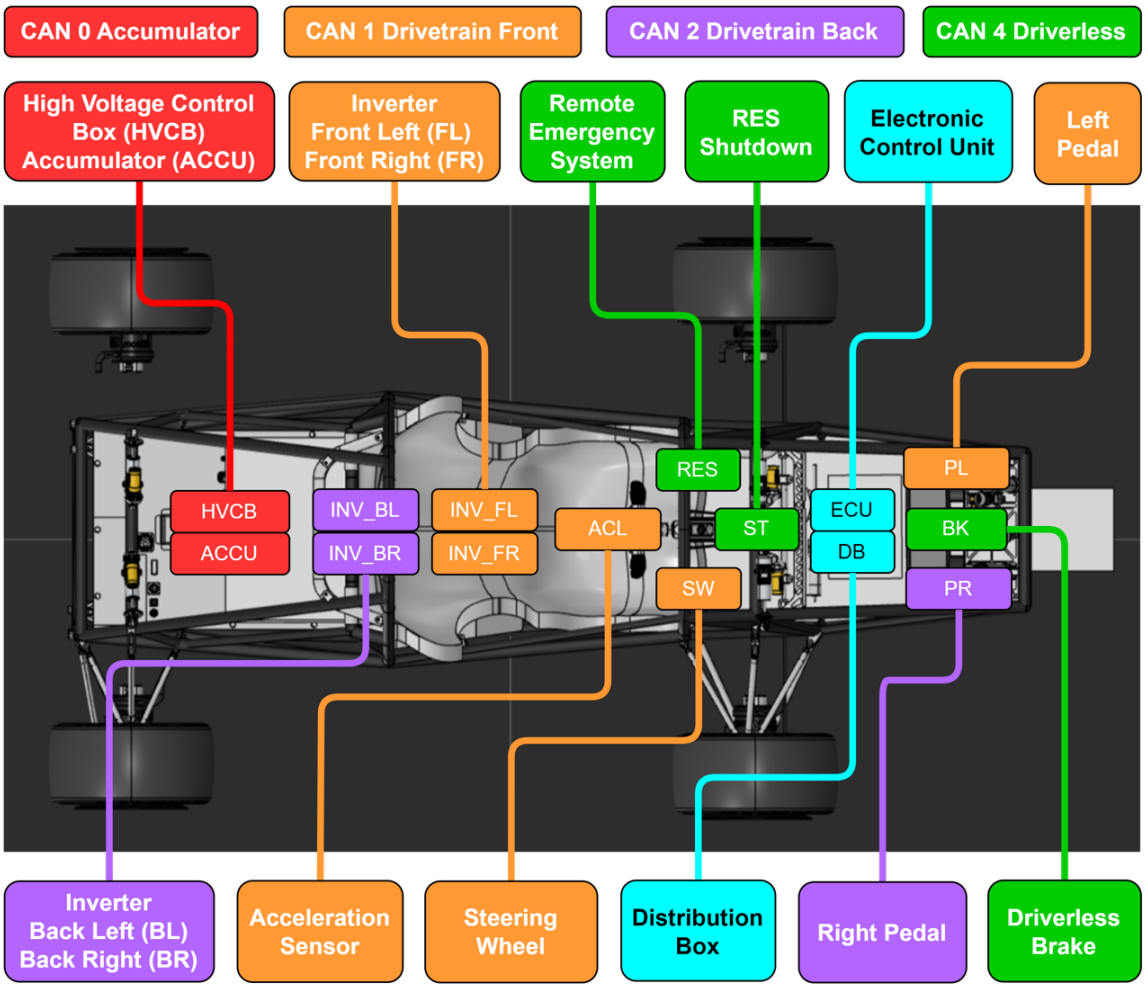


Figure 2.1.: Old 2023 EV CAN Bus overview

Figure 2.2 shows the connections between the distribution box and all hardware components. This box enables the ECU to control certain hardware components via a CAN message. However, the exact functions of the Distribution Box will not be discussed in detail here, as it will not further be used in the new season. The reason for this is the cumbersome handling to extend or modify hardware components that are in direct contact with the Distribution Box.

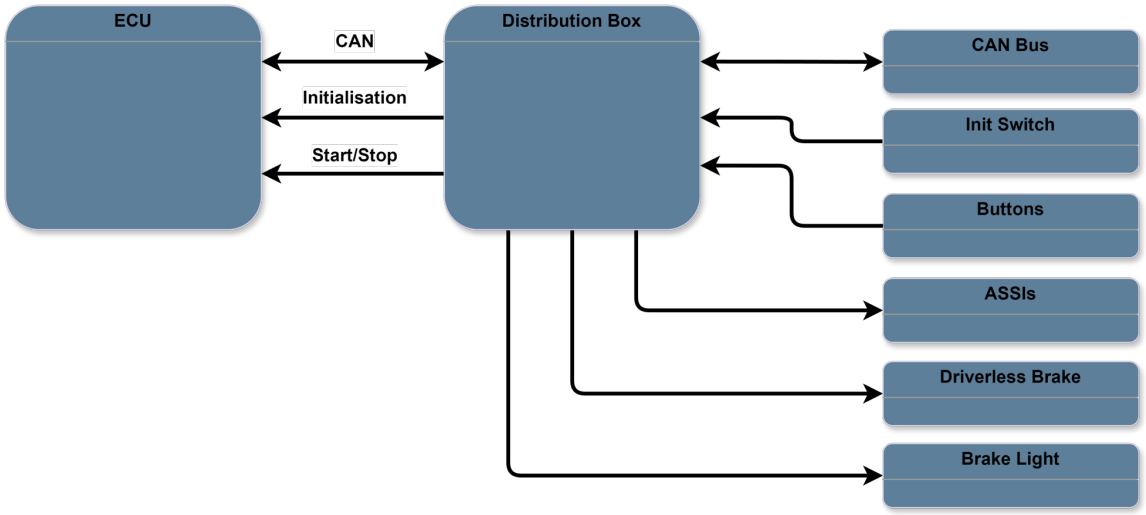


Figure 2.2.: Distribution Box connections

A special feature that must be taken into account is the Remote Emergency System (RES), which must be in a Formula Student EV in order to take part in the races. This system must be integrated into the EV's Shutdown Circuit (SDC) in order to be able to switch off the entire EV in the event of an emergency and is therefore required in every season. [8]

However, the most important components are located on the remaining three CAN buses, which ultimately make the EV drivable.

CAN 0 forms the foundation for operating the EV, as the High Voltage Control Box and the Accumulator are connected to this bus. Both systems go hand in hand and have their own internal safety precautions, which means that no communication is required to handle possible errors to prevent potential damage to the hardware. Communication is primarily necessary in order to put the EV in a driveable state and to be able to operate it in an optimal way.

CAN 1 and 2 are the most critical elements of the entire EV, as one pedal sensor is connected on each CAN bus, which determine the power to be transmitted to the inverters. For this reason, the pedals are designed redundantly in order to be able to recognize a malfunction. Furthermore, there are two inverters on each CAN bus. It is important to note that the inverters require a message at regular intervals so that they can continue to operate. If communication is interrupted, they would therefore no longer provide any power to the motors.

An feature, which is not directly visible in figure 2.1, is the driverless system, which runs on a Jetson and is to be connected to the ECU. The implementation has already been initiated in the 2023 ECU, which is why the first approaches in the area of driveless brakes are already visible. However, due to the increasing complexity, the idea came up to implement a new ECU on the Jetson platform.

2.2. 2024 Electrical Vehicle

Alongside the implementation of the new ECU, minor adjustments to the 2024 EV were introduced. Notably, a reduction from four CAN buses to three was enacted. While Formal Student regulations stipulate the necessity of two CAN buses, an additional one was incorporated to ensure ample redundancy in the event of a bus failure during a race [6]. Furthermore, the decision to utilize three buses instead of four was motivated by their reduced complexity and maintenance requirements.

In addition, the CAN wiring was also slightly reworked on the 2024 EV, as shown in Figure 2.3. As a result, signals that were routed via the signal distribution box will be divided into smaller groups and connected directly to the CAN bus. This will improve the modularity and therefore also the overview of the overall system.

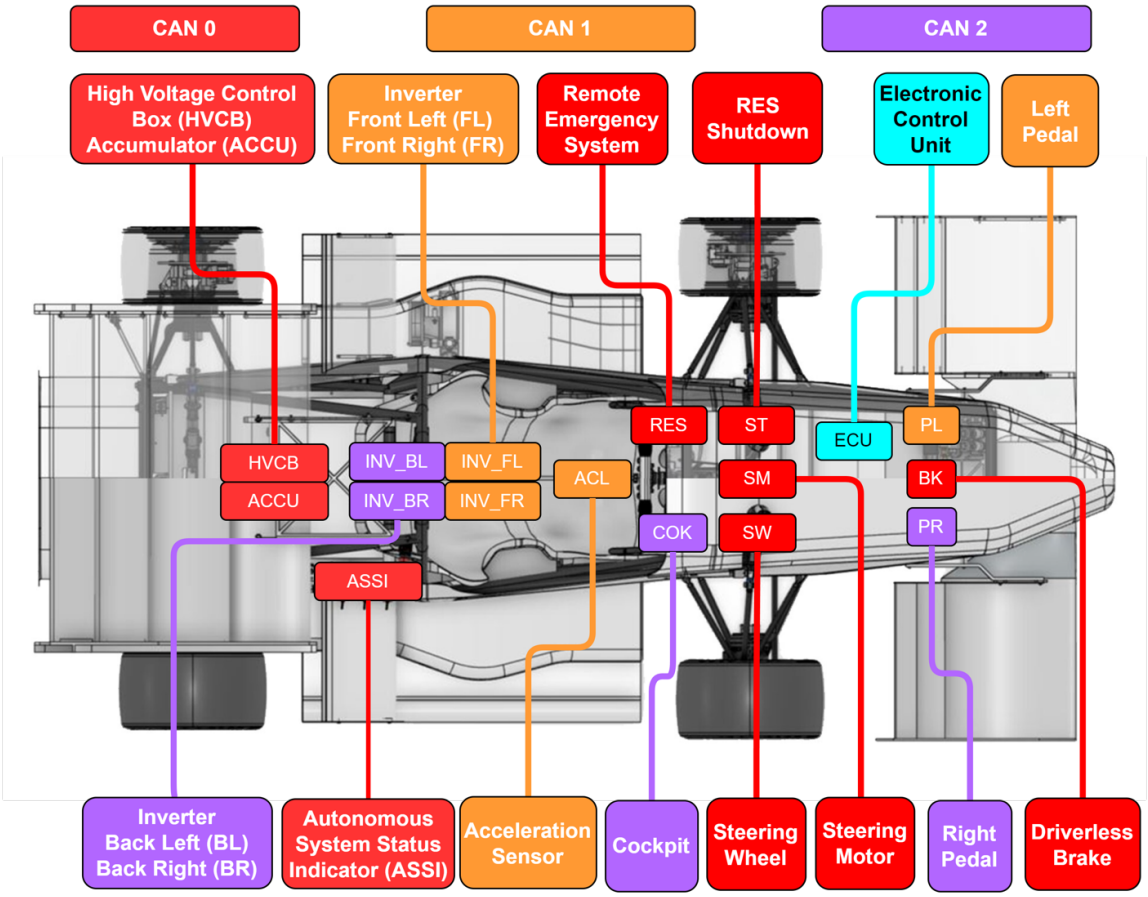


Figure 2.3.: New 2024 EV CAN Bus overview

Furthermore, communications have to be revised or new ones added. The accumulator is being completely redesigned in another Bachelor thesis, which is why the communication will change. A detailed list of all ECU-relevant messages can be found in Appendix A.2. Another modification is brought about by the driverless system. This will involve controlling a motor that enables to control the steering wheel. The functionality of this motor can also be expanded in the future so that servo support is feasible during normal driving. Moreover, it is possible to collect data on the EV to enable further improvements.

2.3. Electronic Control Unit

The old 2023 dSPACE MABX III ECU, shown in the Figure 2.4, is a highly developed and robust technology, both in terms of hardware and software, as the company's history shows.

dSPACE is a company that was founded in 1988 and has been dealing with real-time development systems ever since. In 1998, their first ECU for vehicle use came onto the market. One year later, MATLAB (Simulink) was used to program an ECU. Since then, the company has mainly focused on automated systems. [13]

Figure 2.4.: dSPACE MABX III ECU [14]

The new 2024 ECU is a new platform based on the Nvidia Jetson, which is being developed by the Institute of Embedded Systems (InES) at the ZHAW. This platform makes it possible to connect various hardware modules. The crucial point is therefore that it is possible to equip the Nvidia Jetson with one or more CAN interfaces. With the given computing power and a corresponding CAN module, it is therefore possible to replace the 2023 dSPACE ECU.

The previous project work, on which this work builds upon, was based on an older version of the baseboard. The basic hardware and software structure of the ECU was therefore already laid down in that work. In addition, a CAN module was developed, which should make it possible to connect three CAN buses, but the CAN module could not yet be fully tested during the project work and is thus implemented in this thesis.

This Bachelor thesis is based on the newest baseboard available at the time of writing, which can be seen in Figure 2.5. The baseboard has been reduced in size, which means that fewer connection options are available, but this does not cause a problem from the ECU's point of view. The new platform also provides the option of connecting a networking interface module, which opens up further possibilities for this platform, like LTE live sever connection.

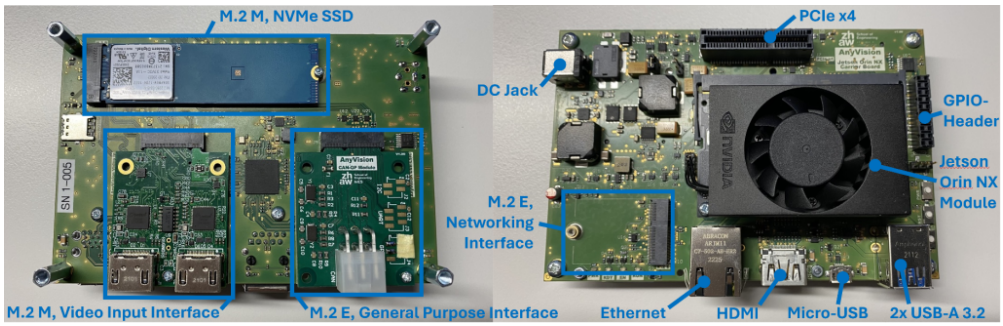


Figure 2.5.: Baseboard with Nvidia Jetson Orin NX [22]

2.4. Risks and Opportunities

The transition to a new platform is associated with a range of potential risks and opportunities. Especially when considering the overall picture, as new hardware is being utilized with little experience in the automotive sector. In addition, this is a project that is part of a larger project, which is why the actual implementation of the solution is dependent on factors that cannot be influenced. Nevertheless, there are also risks and opportunities within the platform itself.

Risks

If the software of the Jetson is considered, a Jetson Linux 35.1 [15] is installed, which is based on Ubuntu. This in turn can impact the real-time capability of the ECU, as other processes can also influence the behavior of the ECU. In addition, there is no framework like the one dSPACE uses for programming the ECU, which makes development more difficult and can lead to necessary optimizations within the Operating System. A predefined framework can therefore also quickly provide a good overview of the overall system without having to deal with system-critical elements, such as the functionalities of a CAN bus. Not having a framework also means that more work is required to provide an equally effective platform.
Furthermore, the hardware of the Jetson is not intended for automotive use. The use of this hardware in vehicles can cause various problems, which is why attention should be paid to various factors as power supply, interference signals, temperatures or vibrations.

Opportunites

Nonetheless, the Jetson also offers advantages, as the software used is open source and therefore many libraries can be utilized. Furthermore, there are no Matlab or dSPACE licenses, which in turn simplifies working in a larger team. Especially when the cooperation of ECU, driverless and other software based solutions is considered. Fundamental changes can be made to the entire system, as there are very few limitations without a framework.
In addition, the baseboard offers the possibility of providing every possible interface when a corresponding module is created. The weight can also be reduced as the 2023 ECU and some additional hardware components can be omitted.

In summary, the new system opens up many possibilities for adding various functionalities to the EV, which can support development in all areas. However, it will not achieve the same robustness as a complete system that has been specially developed for the specific application of an ECU. Nevertheless, the development of an ECU on a Linux based system offers the significant advantage that the software can be implemented on other Linux systems and is therefore independent of the used hardware, as long as the necessary computing power and interfaces are available.

3. Concepts

This chapter describes the basic concepts of the 2024 ECU. The aim of these concepts is to ensure the modular expandability of the ECU and to make the entire software easier to understand for future software developers.

C++ was chosen as the programming language for the 2024 ECU because of its object-oriented nature, hardware-related programming capabilities, and a large number of open-source libraries available, for example for CAN bus control. An object-oriented programming language like C++ allows the creation of small, self-contained blocks of code, so that less software expertise is required to implement changes and extensions.

Figure 3.1 shows the basic functionalities of the new 2024 ECU. It thus shows that the ECU should be designed as transparently as possible to create a simple interface to the EV in order to simplify future developments and adaptations. First and foremost, this is intended to enable driverless functionality. However, this functionality can be expanded even further, for instance to allow data logging.

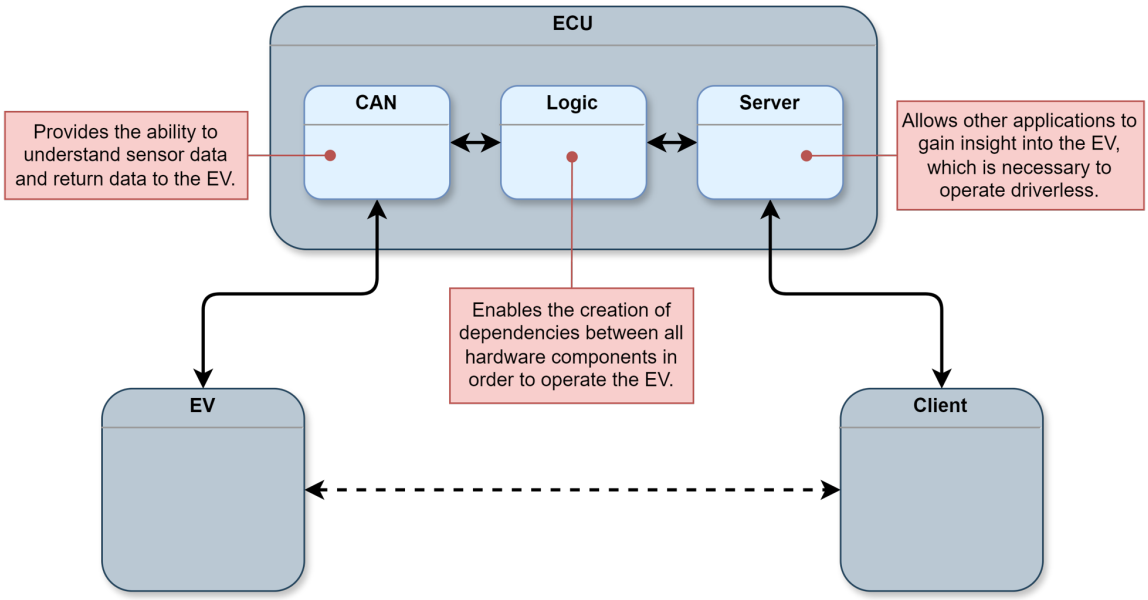


Figure 3.1.: Functionalities of the new 2024 ECU

3.1. ECU Architecture

The previous project work defined the basic architecture of the ECU software. This resulted in a framework, which can be seen in figure 3.2. It consists of two large segments containing several modules.
The *System* segment acts as a container for various smaller system-control modules that contain the EV logic, such as the engine and accumulator logic. Conversely, the *Data* segment houses various interface modules that are required for the ECU to communicate with the entire EV. For example, it houses modules that are responsible for CAN communication and those that manage User Datagram Protocol (UDP) communication with other external devices or Docker containers.

Both the *Data* and the *System* segment have their own memory block. For the *Data* segment, this memory is called "sensorData" and is used to store incoming messages from external sources. Within the *System* segment, it is called "systemData" and is used to store the calculation results of the individual system-control modules. A data bridge connects the *Data* and *System* segments so that the *System* can access the "sensorData" for necessary calculations in the EV logic. In addition, the *System* can share parts of its own memory with the *Data* segment to facilitate data transfer from the ECU to the EV's components.

Consequently, it is up to the *System* segment to manage the data bridge of both segments and ensure that each system-control or interface module only accesses the data it needs to maintain encapsulation and avoid errors due to memory overwrites.

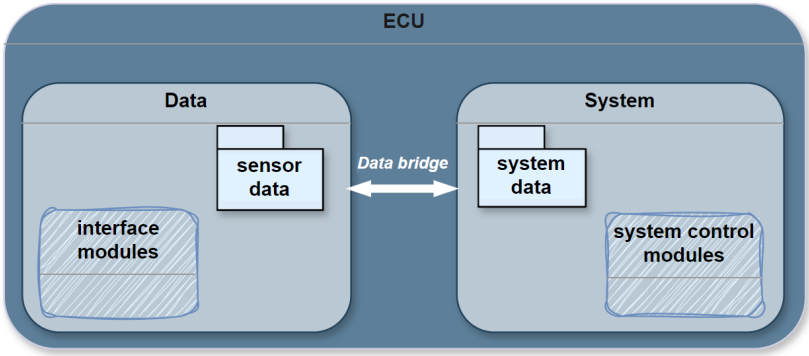


Figure 3.2.: ECU software framework

The modular design is also a major advantage for future EVs, because if a certain component in the EV is changed, only the module needs to be replaced and not the entire software. For instance, if the EV is fitted with a new engine in the future, only the engine control module needs to be replaced. This modularity is especially important for the Zurich UAS Racing Team, as the software developers change almost every year and not everyone has the time to familiarise themselves with the entire software. As a result, in future, software developers will only need to know the modules that they really want to change. The primary purpose of dividing the segments and memory blocks into *Data* and *System* segments is to protect the logic within the individual *System* control modules and the "systemData" memory from external influences by adding an additional barrier. This prevents unauthorised access from the *Data* segment to the *System* segment and reduces the risk of possible damage to the EV logic. However, the System segment retains access to the "sensorData" and can make the "systemData" available to the *Data* segment for transmission. In addition, the separation of the *System* and *Data* segment creates responsibilities for the individual interface and system-control modules as well as for the memory blocks, which minimises the risk of memory leaks.

3.2. CAN Communication

The UAS Racing's EV uses CAN as the standard protocol for internal communication. CAN is a widely used standard in the automotive industry, enabling the control of various vehicle systems and the acquisition of sensor data. By using the CAN bus system, the individual components within the EV can communicate and coordinate efficiently, ensuring seamless integration of different systems.

The communication concept of the CAN bus system in the 2024 UAS Racing EV remains unchanged from the 2023 model. It is structured in a way that the ECU acts as the main hub for all communication. This means the ECU is connected to all CAN buses and can receive all information exchanged via these buses.

Various components are connected to the CAN bus, ranging from simple sensors to more complex control units such as an accumulator or an inverter. These components regularly send their measurement data via the CAN bus, ensuring it is available to the ECU when needed. Regardless of conditions, these components constantly send update messages.

The ECU can also send messages to these individual components to control them as required. Unlike the components, the ECU does not need to send messages at regular intervals.

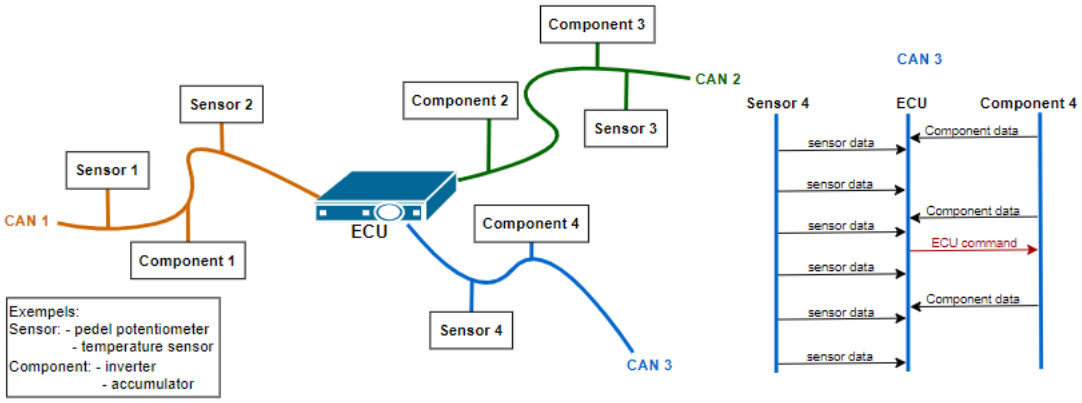


Figure 3.3.: CAN concept

The major advantage of this type of CAN communication is that the constant transmission of messages from each component ensures that the loss of a single message has no negative impact, since a new message from the same component will follow shortly. This eliminates the need for a complex acknowledgment mechanism, which can be maintenance-intensive and complicate the addition of new components to the EV. Consequently, this CAN communication method allows for the simple and modular integration of new components into the CAN bus. Additionally, many commercially available sensors support cyclic message transmission.

However, a disadvantage of this CAN communication is that the large number of messages can overwhelm the ECU, which may not process them as quickly as they are sent, leading to unnecessary traffic. While adjusting the transmission intervals of the components can significantly reduce this issue, it cannot be entirely eliminated. Nevertheless, this drawback is acceptable, as the EV has relatively few components per CAN bus, and it is more important to maintain a simple and comprehensible system for the frequently changing software developers in the UAS Racing Team.

3.3. Server Communication

The task of the server within the ECU is to provide a method of exchanging parameters with external applications. It is therefore the aim to be able to support other developments outside the ECU.

A concept for the server was already created during the project work. However, this concept will be overhauled in this work in order to make it more future-proof. The fundamental change to the concept is the splitting of a single large server into several small servers. This promotes the modularity and thus also the independence of the individual applications. As a result, an overview of all outside influences on the ECU can be maintained and appropriate precautions can be taken. For example, the datalogger only requires read rights, which is why it is only permitted to read data. The driverless, meanwhile, requires read and write permissions in order to be able to operate the EV. Furthermore, the driverless must be granted a higher priority than the datalogger, as the driverless is a highly important real-time system.

Due to the broad range of applications that the server needs to support, the protocol must be able to provide many functionalities in order to be able to integrate future projects. This results in the protocol having to be dynamically expandable in order to provide flexibility for future extensions.
At the same time, it should also be straightforward to read. There are several reasons for this. On the one hand, it can facilitate a quicker understanding of the protocol and on the other hand, it can avoid errors during development. This is particularly important with regard to the merging of several different projects.

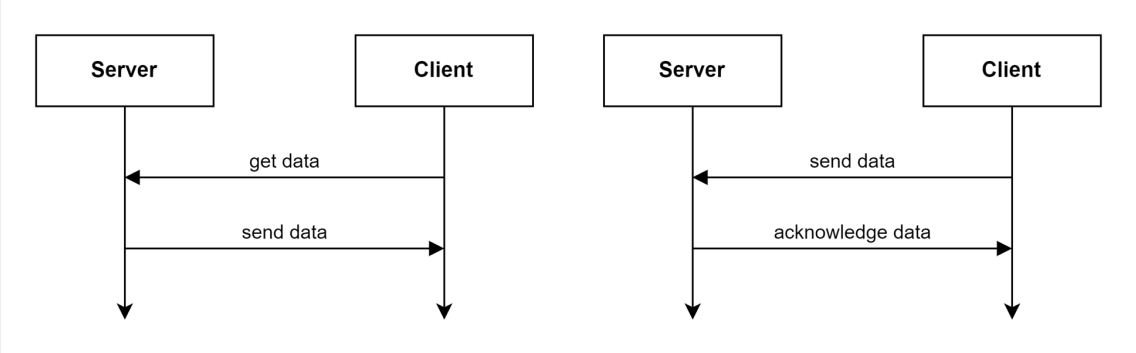


Figure 3.4.: Protocol for server

To meet these requirements, a protocol based on the request-response model was chosen, as shown in Figure 3.4. The request should be used to retrieve system parameters and to send control signals to the ECU. This is followed by a corresponding response that should inform the client about system parameters or the successful execution of the control signals. As a result, the client is solely responsible for the data exchange.

The reason why this model was chosen is because it is a very simple way of exchanging data. In addition, data is only sent if the client requires new data or if new data is available that can be sent to the ECU. A more efficient protocol would also have a higher probability of a bug and would therefore be less comprehensible.

4. Implementation

This chapter covers the implementation of all functions required for the ECU for the UAS Racing Team. Figure 4.1 shows the simplified hardware system in which the ECU will be implemented. It is evident that the ECU is located in a Docker container, where the ECU should not only be able to communicate with the hardware of the vehicle via the CAN bus, but also with other applications over UDP. These applications can be located both inside and outside the Jetson, in other computers or Docker containers.

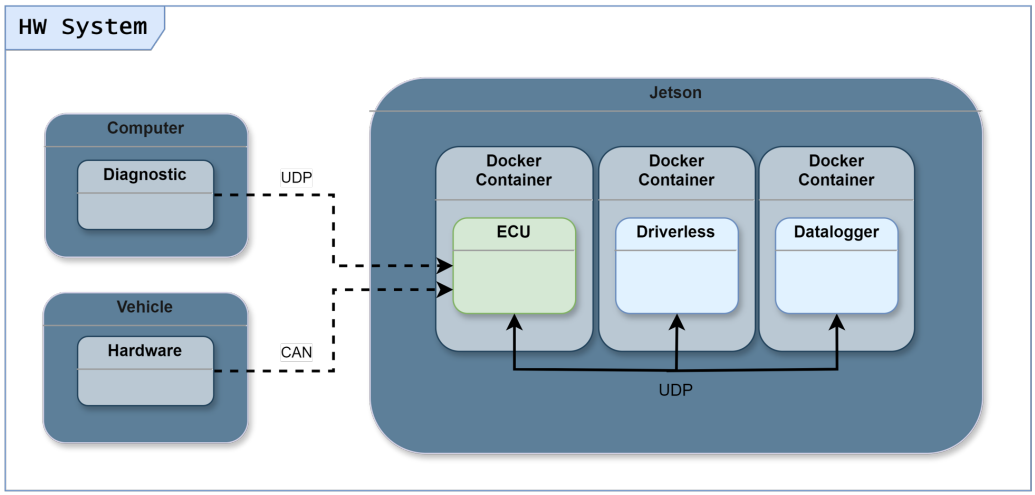


Figure 4.1.: HW system overview

In order to fulfill these functionalities, an ECU framework is required to be able to implement various modules. These modules contain both the EV logic and the logic of the individual interfaces of the ECU. Figure 4.2 shows the overall overview of the ECU software. As the ECU is programmed in object-oriented C++, each individual module block represents an object of a class.

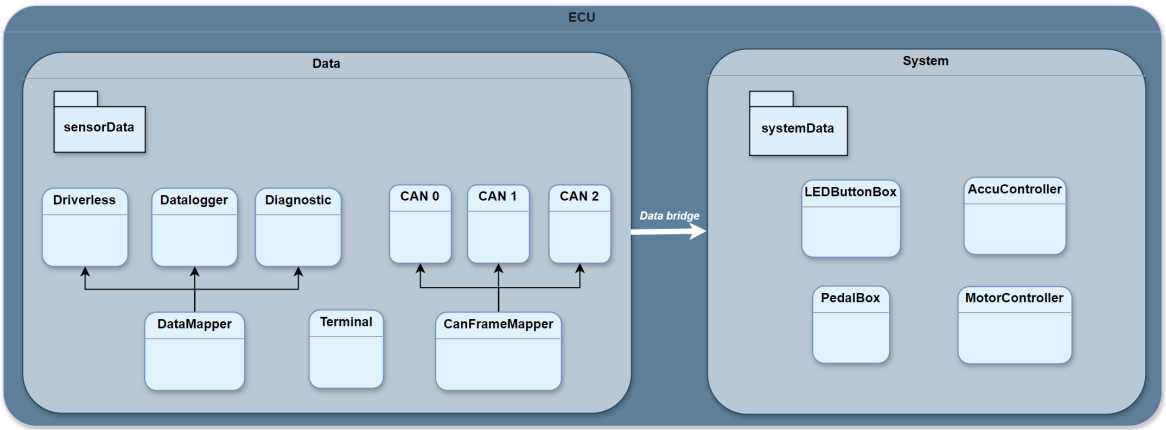


Figure 4.2.: ECU software overview

4.1. ECU System

As mentioned in the chapter 3.1, the ECU *System* segment is used to house the individual vehicle logic modules. These modules have been transferred in a very similar way from the old 2023 ECU to the new 2024 ECU. This means that all the basic logic responsible for EV control on the new Jetson ECU is exactly the same as on the dSPACE MABX III ECU from 2023 [3]. The difficulty, however, was that the dSPACE MABX III ECU is programmed in Matlab (Simulink) and the Jetson ECU in C++, so all the Matlab code had to be converted manually into object-oriented C++ code.

Figure 4.3 provides an overview of how the individual *System* modules are connected to each other, or in other words, which data inputs and outputs they have. The "sensorData" memory, in which all received CAN messages are stored, can be seen on the left of the image. From here, the individual data is forwarded to the system modules. The "CANoutData" memory, which is part of the "systemData" memory, is shown on the right. From the "CANoutData" memory, the individual data is sent back to the CAN bus to the vehicle components.

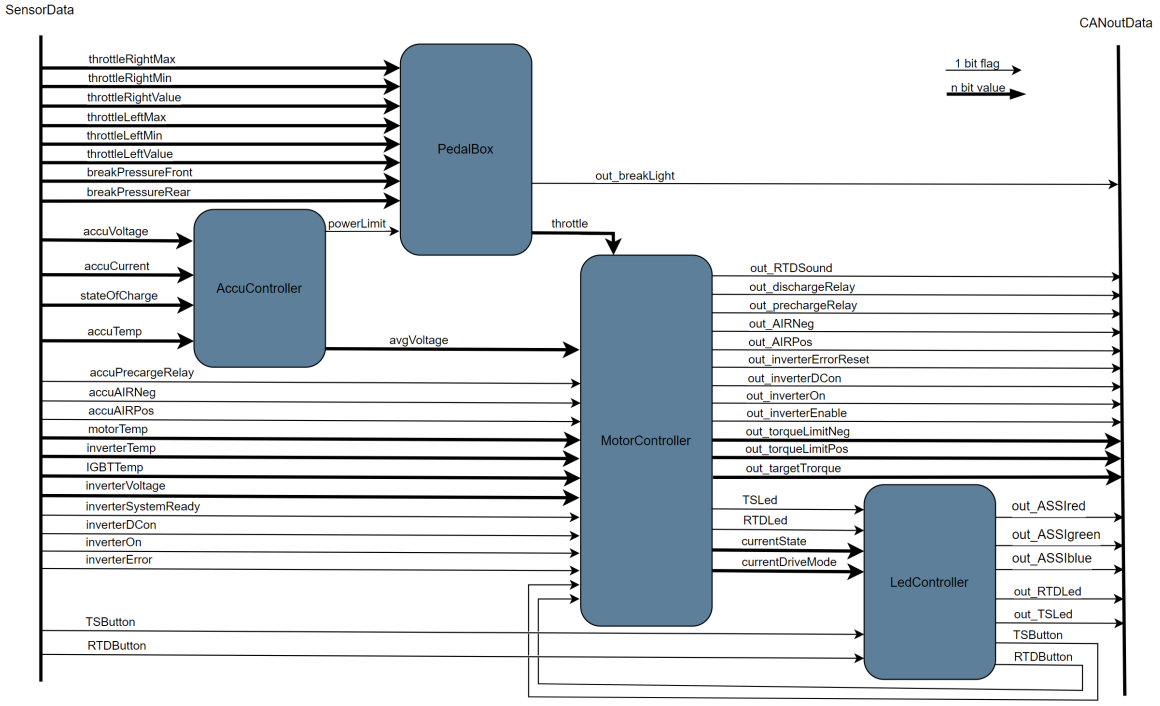


Figure 4.3.: *System* segment module connections

The individual logic modules of the *System* segment are explained in more detail in the following chapters. It is important to note that some of the illustrations and explanations are slightly abstracted to make the function and concept easier to understand. This is particularly necessary for the *PedalBox* and *MotorController* modules, as they are very complex and control numerous things at the same time.

4.1.1. LEDButtonBox

The main task of the *LEDButtonBox* is to read out two buttons and control their integrated Light Emitting Diode (LED) on the dashboard of the EV. These two buttons can be seen in Figure 4.4. The right button is the Tractive System (TS) button. An active TS means that the 600 V of the accumulator is applied to the inverters and that the inverters are ready to turn the motors. The TS button can be compared to the start button of a conventional car, that is used to start the engine. The left button is the Ready To Drive (RTD) button. As soon as it is pressed, the EV is ready to drive and the driver can press the accelerator pedal to accelerate the car. In this case the RTD button can be compared to a simplified version of a gearstick that can shift the vehicle from Neutral to Drive.

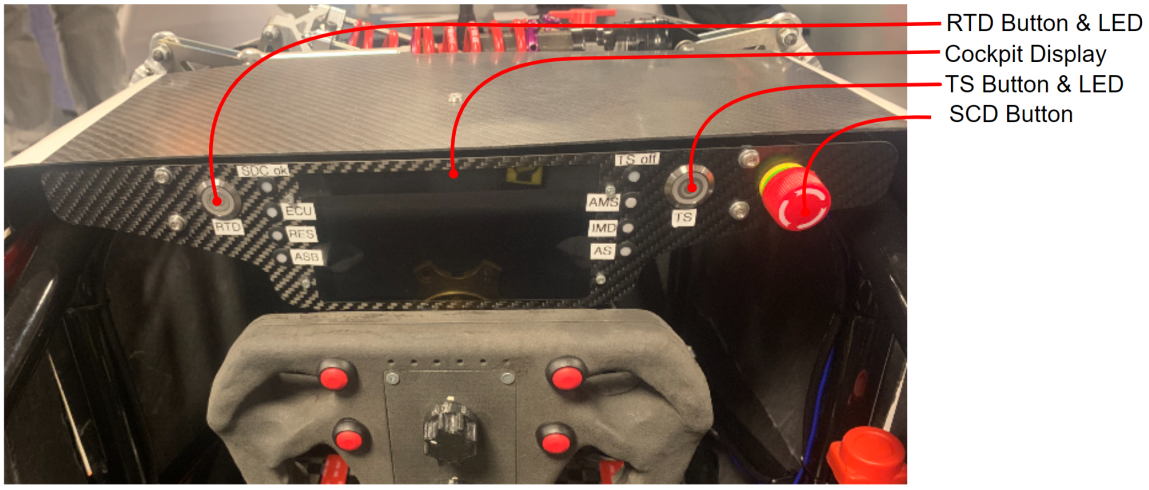


Figure 4.4.: EV dashboard with buttons

The two buttons TS and RTD can also be used to select the various driving modes such as normal, slow and endurance driving. To select a drive mode, it is necessary to press the SDC button, only then can the drive mode be selected using the various button combinations. Figure 4.5 shows which button combination is used for which drive mode.

Button combinations for driving mode selection		
	TS Button	RTD Button
Select Normal Mode	Pressed	Pressed
Select Slow Mode	Not Pressed	Pressed
Select Endurance Mode	Pressed	Not Pressed

Figure 4.5.: Button combinations for driving mode selection

The LEDs of the buttons are also controlled by the *LEDButtonBox*, but the *MotorController* has the ability to give commands to the *LEDButtonBox* about how the LEDs should light up. The connections between the *MotorController* and the *LEDButtonBox* are shown in Figure 4.6. In the EV, the *MotorController* only controls the LEDs during start up, in other words when the TS is ready to be switched on, the *MotorController* gives the command to switch on the TS LED, and when the EV is ready to drive, the RTD LED lights up to indicate to the driver that the button can be pressed. The LED button start up sequence is shown in Figure 4.7.

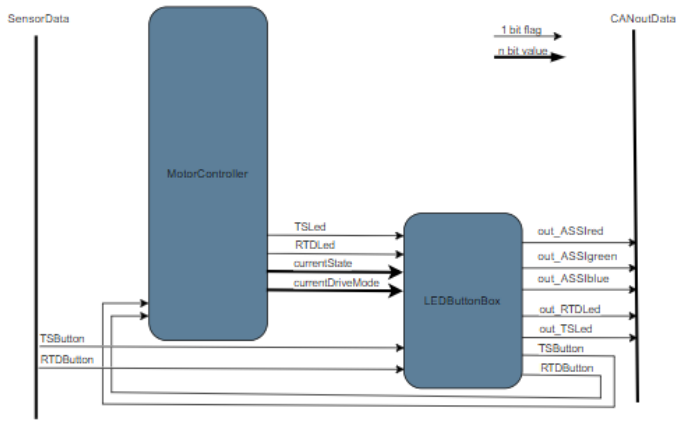


Figure 4.6.: *MotorControler* and *LEDButtonBox* connections

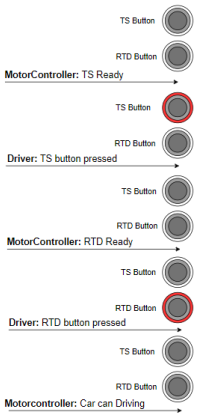


Figure 4.7.: Button and LED startup sequence

During the drive, the LEDs are used to provide the driver with information about the EV by displaying different light patterns. The various light patterns are shown in Figure 4.8. These lighting patterns are programmed according to the dark cockpit philosophy, which means that if nothing is illuminated, the EV is ready to drive and as soon as something is illuminated, the driver is alerted to something.

Dashboard LED patterns		
	TS LED	RTD LED
Overheat	Off	Blinking
Error	Blinking	Off
Select Normal Mode	Off	Off
Select Slow Mode	Blinking	Blinking
Select Endurance Mode	On	On

Figure 4.8.: Drive mode: dashboard LED light patterns

Since the 2024 EV has a cockpit display, this LED display is somewhat superfluous and unnecessarily complicated for the driver. For redundancy reasons, the LED display function has been retained in the 2024 model, as the driver still has the option of receiving feedback from the EV in the event of a display malfunction during a race.

An additional function of the *LEDButtonBox* is the control of the three Autonomous System Status Indicator (ASSI) lights. According to the Formula Student Rules form, these three ASSI lights are mandatory for a vehicle that wants to take part in a driverless race and must be mounted on the EV as shown in Figure 4.9 [10].

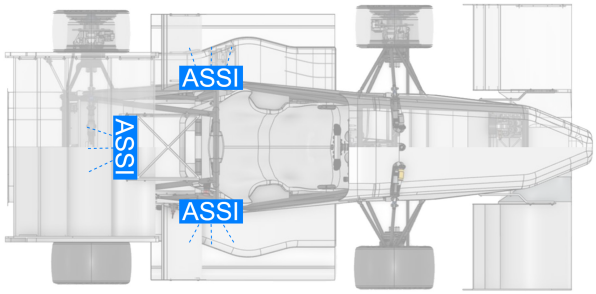


Figure 4.9.: ASSI light position

The lights must be able to light up blue and yellow. But the lights may only be active when the driverless system is activated. These ASSI lights are programmed in accordance with the regulations and function as shown in Figure 4.10.

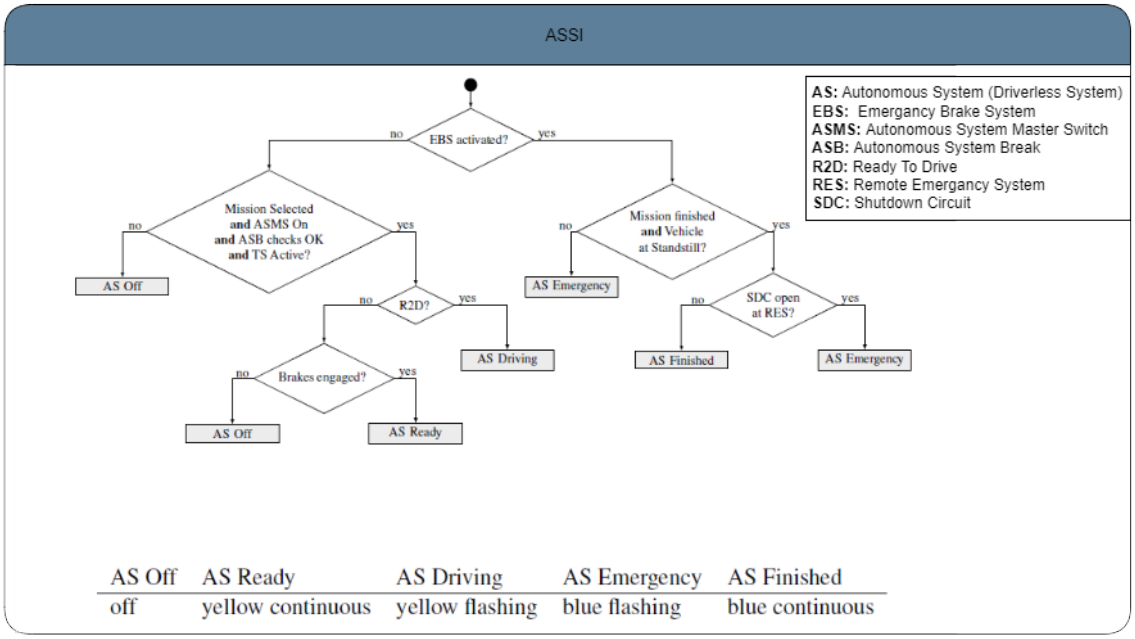


Figure 4.10.: ASSI rules [9]

The *LEDButtonBox* in the 2024 ECU has been programmed to be as modular as possible. This makes it feasible to connect new buttons and lights to the EV in the future, which can then simply be added using a new function in the *LEDButtonBox*. At the moment, as already mentioned and shown in Figure 4.11, the functions for the dashboard-LEDs, dashboard-buttons and ASSI-lights are located in the *LEDButtonBox* module.

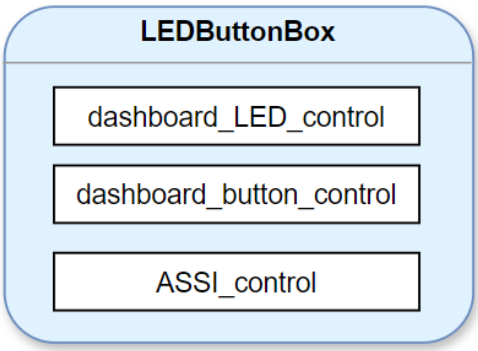


Figure 4.11.: *LEDButtonBox* module with functionality

4.1.2. AccuController

The *AccuController* of the ECU has two tasks. The first is to calculate the average current and voltage of the last 100 measurements on the accumulator. The second task is to calculate a power limit that signals to the *Pedalbox* that it is no longer allowed to accelerate. To enable this functionality, three different connections are required, which can be seen in Figure 4.12. The left-hand side of the *AccuController* shows the data that it receives from the battery via the CAN bus, and the right-hand side shows the data that it sends to the *MotorController* and the *PedalBox*.

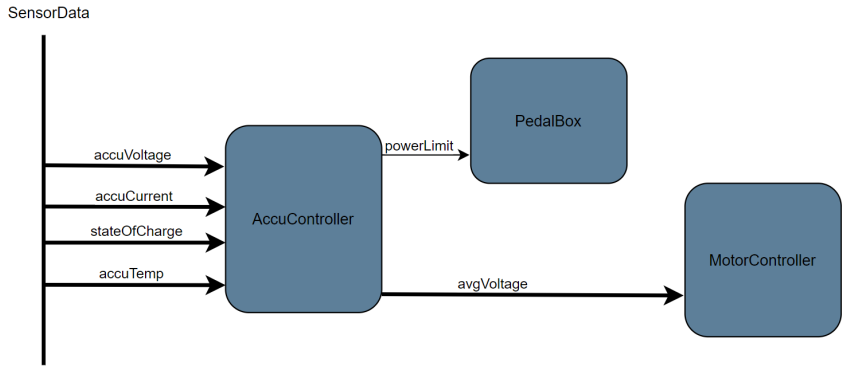


Figure 4.12.: *AccuController* connections

The average current and voltage values calculated in the *AccuController* are required for calculating the power limit. However, the average voltage value is also required in the *MotorController*. The *MotorController* in turn uses this voltage to check whether the inverters are receiving the correct voltage from the accumulator, or whether there has been a voltage drop somewhere in the electrical system.

The power limit is calculated as shown in Figure 4.13. Firstly, the maximum discharge current is calculated as shown on the left. To determine this, the *AccuController* requires both the temperature of the accumulator and the current state of charge. The maximum discharge current can then be compared with the average current, as shown in the image on the right. The power limitation can then be used to decide whether the vehicle may continue to accelerate or whether the accumulator no longer permits this action.

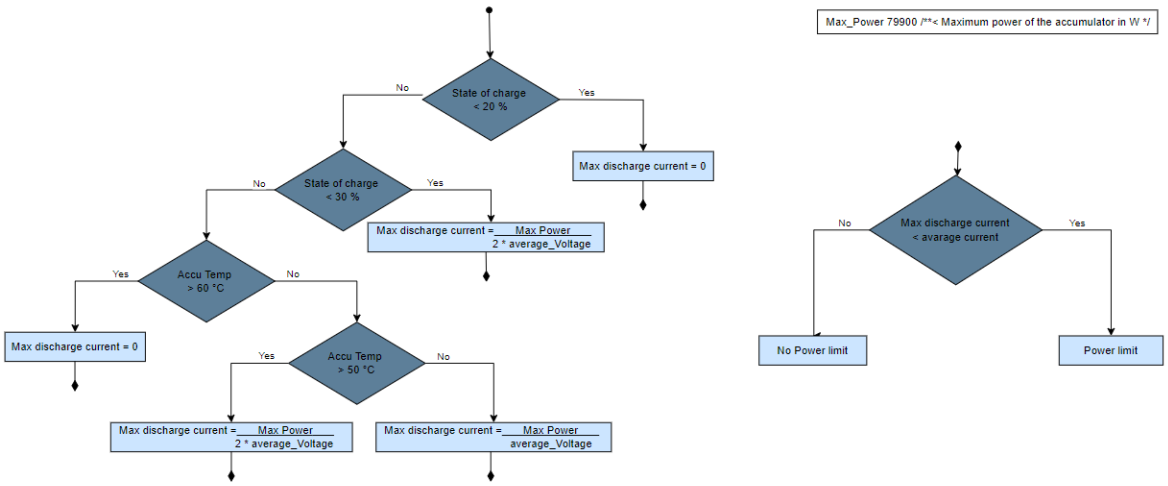


Figure 4.13.: Power limit calculations

The *AccuController* in the 2024 ECU does not have too many complicated tasks, as the accumulator in the EV already performs many of the battery management tasks itself. Even more complicated calculations, such as determining the state of charge, are carried out by the accumulator itself and then transmitted to the ECU via CAN.
However, if a new accumulator is installed in a future EV, which is not as self-managing as the current one, the *AccuController* can easily be equipped with additional functions. Figure 4.14 shows the modular structure of the *AccuController* with its current functions, which can easily be expanded with new functions.

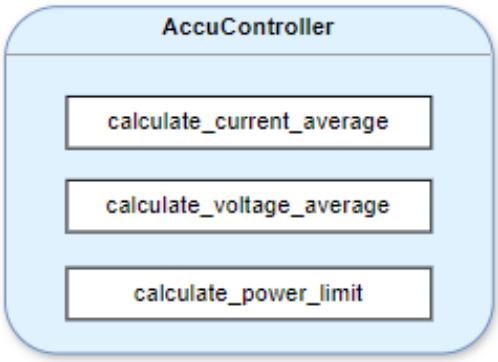


Figure 4.14.: *AccuController* module with functionality

4.1.3. PedalBox

The *PeadalBox* module is used to read the throttle and brake pedal inputs and process them so that they can be used by other modules, such as the *MotorController*, to turn the engine according to the throttle pedal input. It also checks the pedal values for errors, so that an incorrect input cannot cause the EV to malfunction. Figure 4.15 shows the connections of the *PedalBox* in the *System* segment.

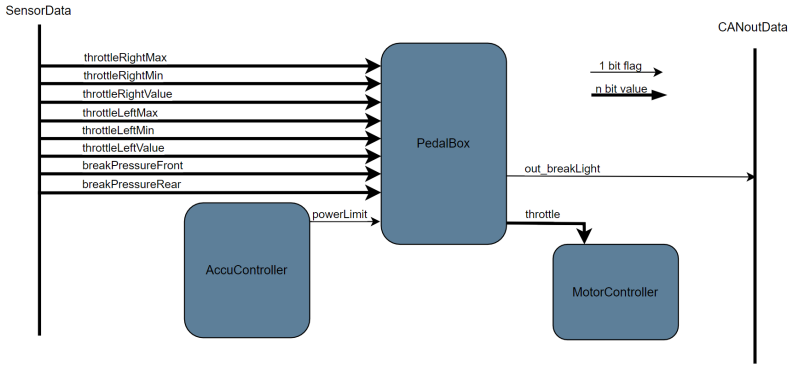


Figure 4.15.: *PedalBox* system connections

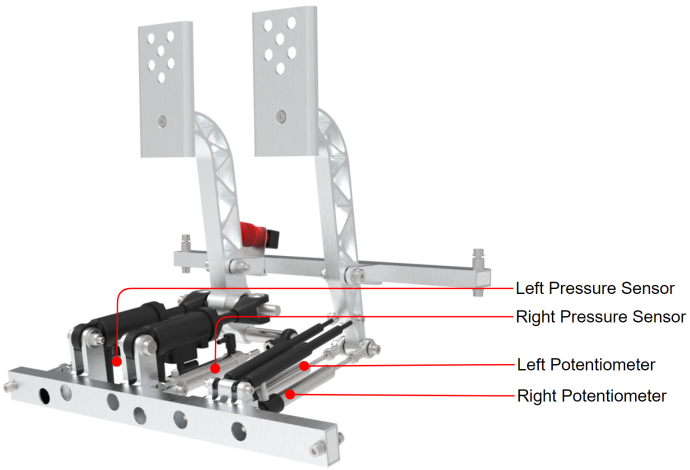


Figure 4.16.: Pedal arrangement in EV

To avoid the risk of sudden uncontrolled acceleration due to a malfunction, the throttle pedal in the EV has a redundant design. This redundancy is demonstrated by the installation of two pedal sensors, known as potentiometers. In addition to the safety aspect, the redundancy of the pedal sensors is also a rule of the Formula Student regulations, which stipulates that the values of the individual pedal potentiometers must be transmitted to the ECU via two separate data cables [6]. For this reason, the values of the two potentiometers are sent to the ECU via two different CAN buses. Figure 4.17 shows the arrangement of the pedal sensors on the ECU.

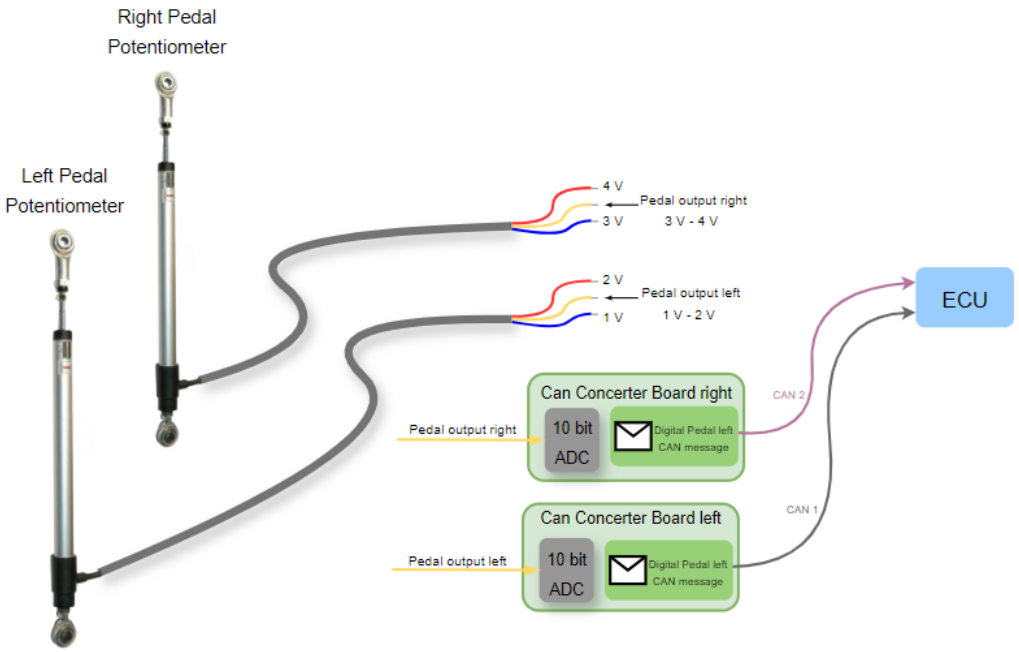


Figure 4.17.: Redundant pedal potentiometer

These sensors operate at different voltage levels, which makes it possible to detect short-circuits between them, as required by the rules [7]. Two types of short-circuit detection are provided for both sensors. The first is a short to ground or Voltage at the Common Collector (VCC) test. This test confirms whether the sensor signal from a 10-bit Analog to Digital Converter (ADC) indicates either zero or 1024. The second detection method focuses on detecting a short circuit between the two sensors. As they operate at different voltage levels, they should never give identical readings under normal conditions.
The deviations between the percentage actuation's for each pedal sensor are calculated and compared. Again, the rules state that if the deviation is within 10 %, the sensor inputs are considered correct for throttle calculation [6].
When calculating the throttle valve, the average of the two sensors is taken as a percentage of the full throttle pedal range, which is then provided to the other modules. The following figure 4.18 shows an overview of the throttle pedal functions in the *PedalBox* module of the ECU.

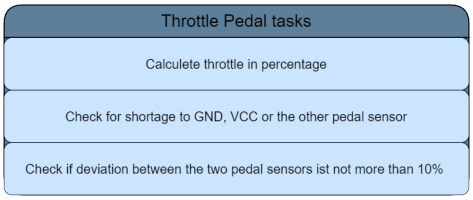


Figure 4.18.: Throttle pedal tasks in the *PedalBox*

The brake pedal also has a redundant design. It consists of two separate hydraulic brake systems, one for the two rear brakes and one for the front brakes.These two braking systems are also prescribed in the Formula Student regulations [5].
Both brake systems are additionally equipped with one pressure sensor each, which transmits the respective pressure to the ECU via CAN. The *PedalBox* in the ECU now checks whether the brake pressures deviate from each other and can thus recognise a defect in the brake system. This means that if the pressure in one brake system is lower than in the other, it can be assumed that this brake

system has a leak in the hydraulic line. This brake pressure deviation test is not prescribed in the regulations, so a value of 10 % is used for the deviation. This corresponds to the equal value of the 2023 ECU. In a functioning brake system, the *PedalBox* has the task of switching the brake lights on and off when a certain brake pressure is reached. The following figure 4.19 shows an overview of the functions of the brake pedal in the *PedalBox* module of the ECU.

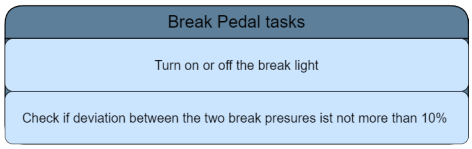
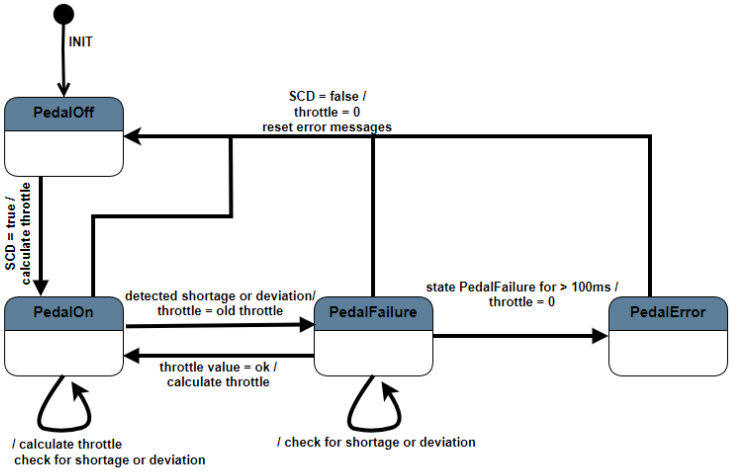


Figure 4.19.: Break pedal tasks in the *PedalBox*

The *PedalBox* uses a state machine to perform the tasks of the throttle and brake pedals, which helps to efficiently manage and control the different states and transitions between normal and fault operation. The *PedalBox* has four states: "PedalOff", "PedalOn", "PedalFailure" and "PedalError". Figure 4.20 shows the different states and their transition conditions.



Pedal-Box State Machine				
	state PedalOff	*state PedalOn*	*state PedalFailure*	*state PedalError*
stateAction();	throttle = 0 reset error messages	calculate throttle check for deviation check for shortage	throttle = old throttle check for deviation check for shortage	throttle = 0 set errror message
stateUpdate();	vehicleStatus on => state PedalOn	vehicleStatus off => state PedalOff shortage or deviation => state PedalFailure	vehicleStatus off => state PedalOff no shortage or deviation => state PedalOn State Failure for > 100ms => state PedalError	vehicleStatus off => state PedalOff

Figure 4.20.: *PedalBox* state machine

To improve the readability and modularity of the *PedalBox* in the 2024 ECU, the state machine was programmed with the state design pattern [19]. In the state design pattern, a separate class is created for each state. Each of these state classes contains a **stateAction()** and a **stateUpdate()** method, which hold the respective state actions and state change criteria. This fixed state class structure ensures that the code remains readable, as all information on a state is located in one place. It also enables simple modular expansion of the state machine to include new states, which can be of great benefit for future *PedalBox* extensions in particular.

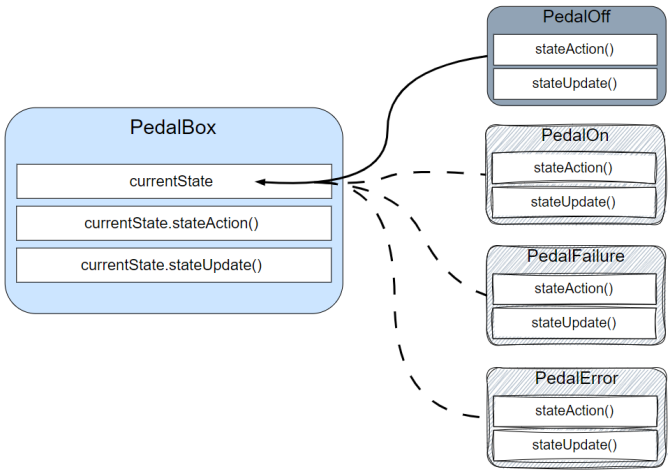


Figure 4.21.: *PedalBox* module with functionality

4.1.4. MotorController

The *MotorController* is the centrepiece of the ECU *System*. It interacts with all other modules in the *System* segment. The main task is to operate the four motors with the desired torque. To do this, it controls the inverter, but also the accumulator relays that supply the inverter with 600 V. Figure 4.22 shows how the *MotorController* is integrated into the *System* segment and how it is connected.

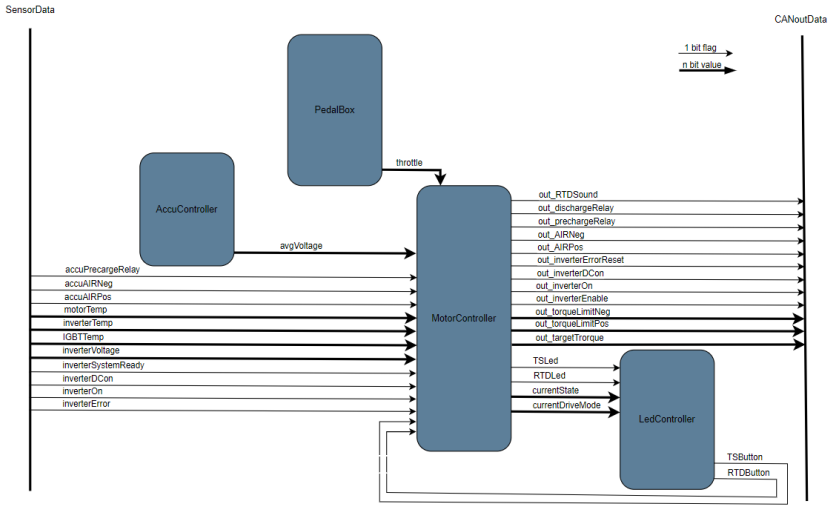


Figure 4.22.: *MotorController* and *System* connections

Figure 4.23 depicts a simplified representation of the motor in the EV. The accumulator, which supplies the inverter, is visible on the left-hand side. Additionally, three relays are integrated in the accumulator, which can open and close the lines between the accumulator and the inverter. The relays are controlled via the CAN bus, which establishes a connection between the ECU and the relays. Upon initiation of the inverter, all relays are initially open. In the initial phase, the Accumulator Isolation Relay (AIR)-negative and the precharge relay can be closed, resulting in the inverter being supplied with a lower voltage than that of the accumulator. The precharge relay assumes the function of gradually activating the inverter in order to protect sensitive components from voltage peaks. It is

only possible to close the AIR-positive and reopen the precharging relay when the precharging process is complete. As soon as the AIR-positive relay is closed, the full 600 V are available at the inverters, what makes the inverters ready for operation. The inverter status flags enable the *MotorController* to receive the inverter status via the CAN bus, thus enabling coordination between the accumulator and inverter. The *MotorController* controls both the accumulator relays and the inverter. In the case of the inverter, it receives the inverter status flags, but can also send commands that cause the motor to be operated with a certain torque, for example.

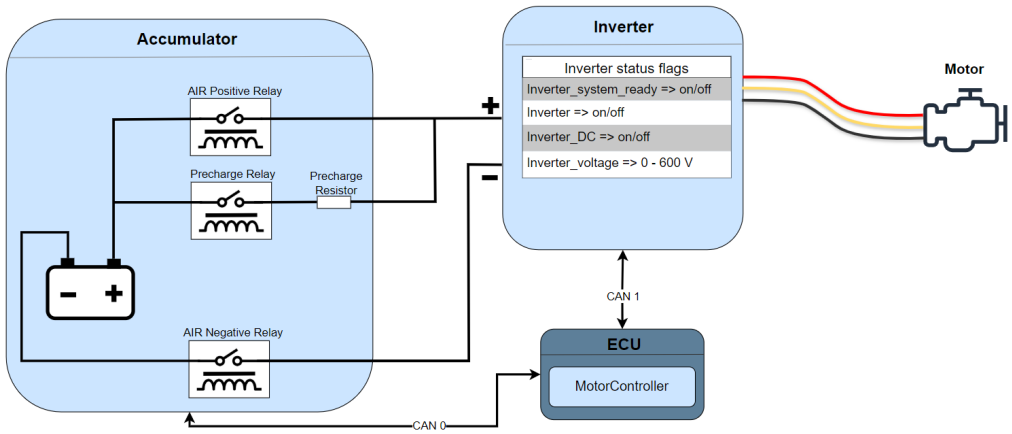


Figure 4.23.: Connections of accumulator, inverter, motor and ECU

The *MotorController* was implemented as a state machine in order to coordinate the various states, such as precharge or other faults, in the best possible way. Figure 4.24 shows the more detailed *MotorController* state diagram. This state machine has a total of eight different states. The different states can in turn be categorised in groups.

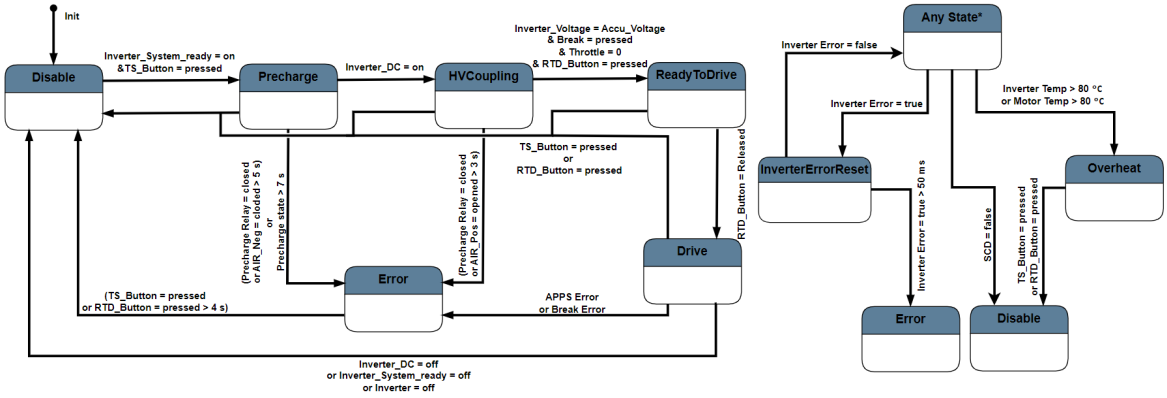


Figure 4.24.: *MotorController* state machine diagramm

Firstly, there are the start-up states, which are responsible for connecting the accumulator to the inverter. The start-up states include the "Precharge", "HVcoupling" and the "ReadyToDrive" state. In the "Precharge" state, the inverter is precharged using the precharge relay. In the "HVcoupling" state, the AIR-positive relay closes and the precharge relay opens so that the full 600 V is applied to the inverter. In the "ReadyToDrive" state, the *MotorController* simply waits for input from the driver so that the EV can be started in a controlled manner.

MotorController State Machine start-up states			
	state Precharge	**state HVCoupling**	**state ReadyToDrive**
stateUpdate();	Inverter_DC = on => HVCoupling	(Inverter_Voltage = Accu_Voltage & Break = pressed & Throttle = 0 & RTD_Button = pressed) =>ReadyToDrive	RTD_Button = Released => Drive
stateAction();	AIR Pos Relay = open Precharge Relay = closed AIR Neg Relay = closed	AIR Pos Relay = closed Precharge Relay =open AIR Neg Relay = closed	AIR Pos Relay = closed Precharge Relay =open AIR Neg Relay = closed

Figure 4.25.: *MotorController* State Machine start-up states

The second state group is responsible for controlling normal operation. This includes the "Disable" and "Drive" states. In the "Disable" State, the inverter is disconnected from the accumulator, which means that all three accumulator relays are open. In the "Drive" state, the EV is ready to drive, in other words, the full 600 V is applied to the inverter and the *MotorController* starts to send the desired torque to the inverter. The inverter will then turn the motor with the desired torque.

MotorController State Machine normal operation states		
	state Disable	**state Drive**
stateUpdate();	(Inverter_System_ready = on &TS_Button = pressed) => Precharge	(TS_Button = pressed or RTD_Button = pressed) => Disable
stateAction();	AIR Pos Relay = open Precharge Relay =open AIR Neg Relay = open	Send target_torque = throttle to Inverter AIR Pos Relay = closed Precharge Relay =open AIR Neg Relay = closed

Figure 4.26.: *MotorController* State Machine normal operation states

The third and final state group are the error states. These include "Error", "InverterErrorReset" and "Overheat" state. A change to the "Overheat" state is possible at any time as soon as the inverter or motor temperature has reached a value of over 80 °C. In the "Overheat" state, the two AIR relays are opened to ensure that the inverter and motor cool down as quickly as possible. A change to the "InverterErrorReset" state takes place as soon as the inverter sends an error message to the *MotorController*. An attempt is then made to reset the inverter in this state. The change to the "Error" state occurs as soon as a deviation from the planned functionality occurs in any state. This leads to the desired torque in the inverter being set to zero, meaning that the EV can no longer continue driving and comes to a stop.

MotorController State Machine error states			
	state Error	**state Overheat**	**state InverterErrorReset**
stateUpdate();	(TS_Button = pressed or RTD_Button = pressed > 4 s) => Disable	(TS_Button = pressed or RTD_Button = pressed) => Disable	Inverter Error = false => old_state Inverter Error = true > 50 ms => Error
stateAction();	Send target_torque = 0 to Inverter AIR Pos Relay = closed Precharge Relay =open AIR Neg Relay = closed	AIR Pos Relay = open Precharge Relay =open AIR Neg Relay = open	Try to reset Inverter Send target_torque = 0 to Inverter AIR Pos Relay = closed Precharge Relay =open AIR Neg Relay = closed

Figure 4.27.: *MotorController* State Machine error states

The *MotorController* is highly complex due to its large number of states and state transitions. In order to make this complexity manageable and to make the state machine understandable and customisable for future software developers, it was programmed using the state design pattern. This design pattern ensures that the individual states have a predefined structure. Each state is therefore a separate class and has a method **stateUpdate()** and a method **stateAction()**. This defines clear responsibilities as to which method is responsible for which task. Furthermore, readability is improved as all state-relevant actions are summarised in one class and every developer can easily find the individual code sections. However, the code of the *MotorController* state machine is particularly large due to the state design pattern. As this state machine has eight states, it also needs eight classes, which means that a lot of code is only needed for structural reasons and is not responsible for the actual *MotorController* logic. However, this disadvantage must be accepted in order to obtain a architecture that is as flexible, maintainable and easy to read as possible.

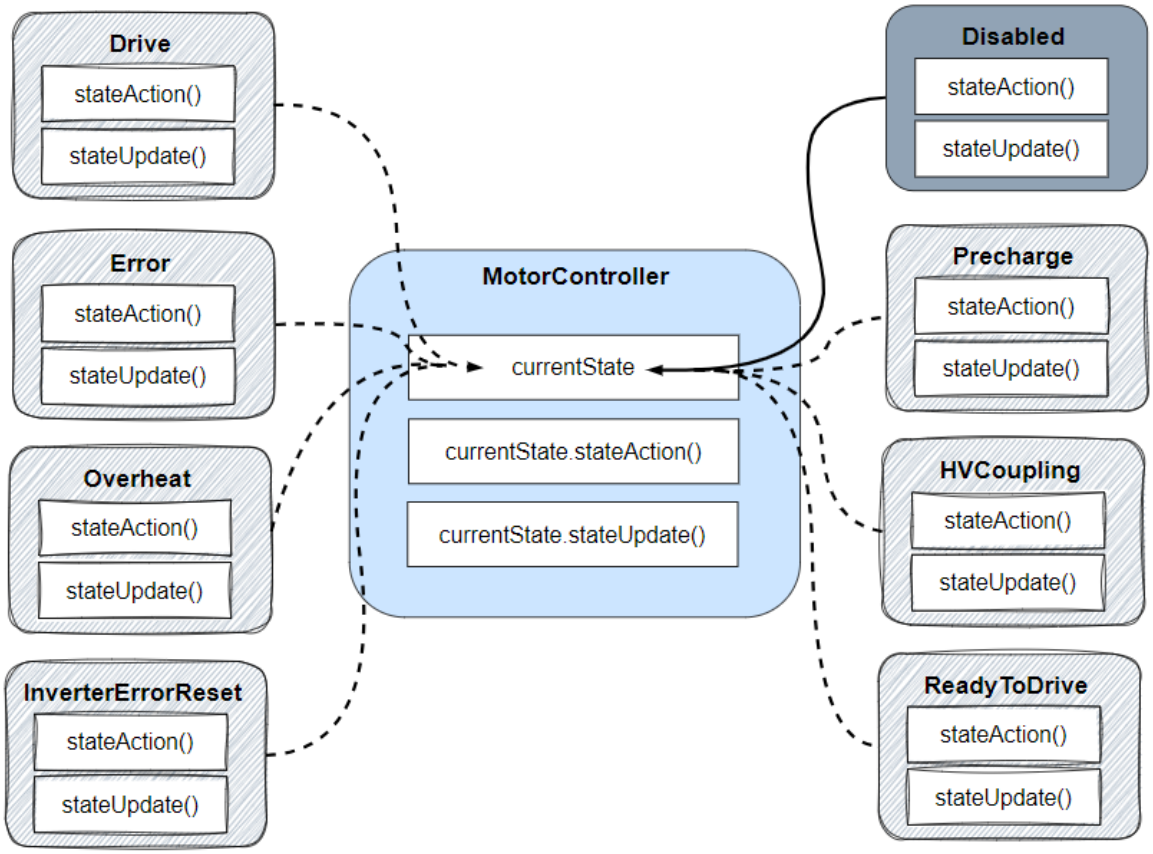


Figure 4.28.: *MotorController* module with functionality

4.2. ECU Data

The ECU *Data* segment is responsible for external communication of the ECU. For this reason, it consists of several *Server* and *CAN* modules and a single *Terminal* module, all of which have different responsibilities. The task of the *Data* segment is therefore to correctly process all these different data streams and provide the ECU *System* segment with the corresponding information, as shown in Figure 4.29.

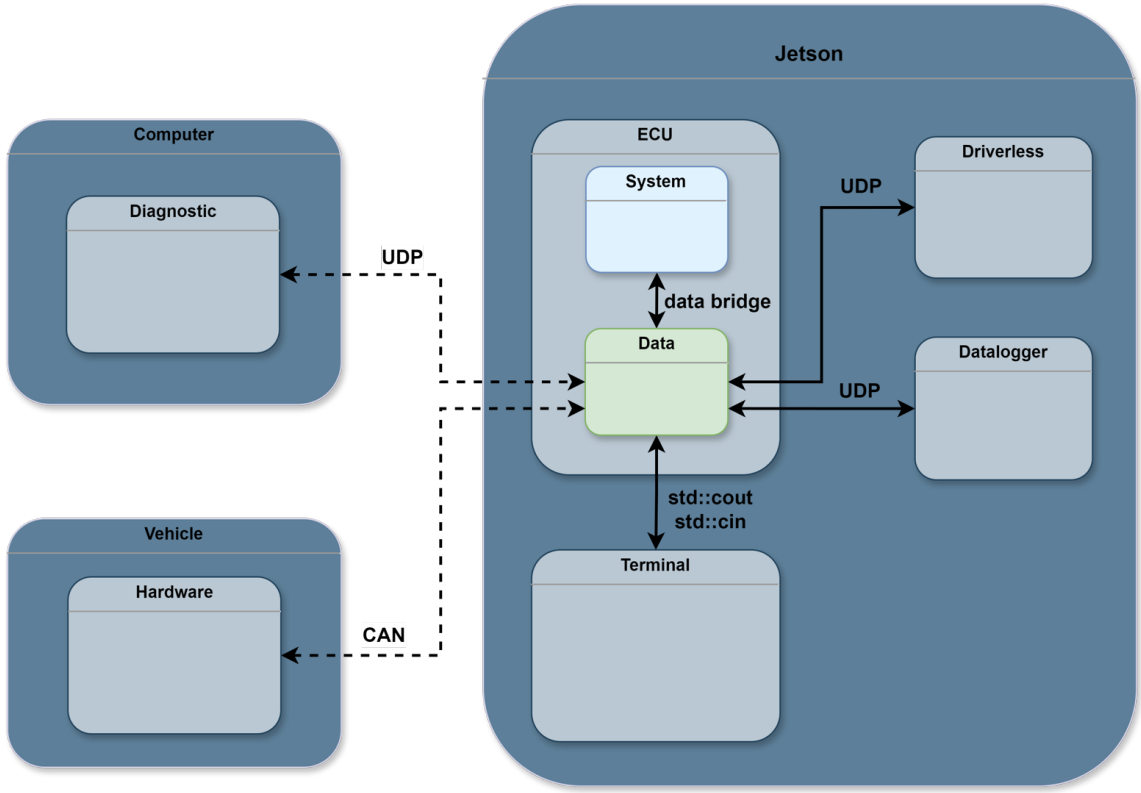


Figure 4.29.: *Data* segment overview

The challenge of *Data* segmentation is therefore to merge all data streams. For this reason, many functionalities within the various modules are executed as threads. This makes it possible to create independence from the various modules, which in turn ensures modularity. The difficulty with this approach is that all data within the *Data* segment must be synchronized correctly. This can be achieved by using mutex within the modules, which in turn can increase the clarity within the *Data* segment. The reason for this is that when the method of a module is called, it can be ensured that access to a variable is permitted if it really has valid data. On the other hand, it can be ensured within the *Data* segment itself which values can be changed by which modules. This means that data may only be overwritten by the *Driverless* module if the driverless system is actually active.

4.2.1. CAN

The *Data* Segment has a total of three *CAN* modules. These modules are identical to each other, or more precisely, the *CAN* modules are three objects of the same class. Each module is responsible for sending and receiving on a specific CAN bus. The basic functions of the *CAN* modules, such as sending and receiving messages, have already been implemented in the previous project work. In summary, a thread is automatically created within the *CAN* module, which reads the data from the CAN bus independently of the main program and stores it in an internal buffer. This buffer enables the *Data* segment to read several sensor data at regular intervals. The sending of data is therefore solved by using mutex, so that no attempt is made to write when data is being read. The implementation was carried out with the help of the can-utils library, which is integrated in Linux [1].

The three CAN buses have different hardware structures. It should be noted that a CAN node always consists of two hardware components: a CAN controller and a CAN transceiver. The Jetson, on the other hand, has an integrated CAN controller, but lacks two additional controllers. An adapter board for the Jetson base board was therefore developed in an earlier project. The adapter board contains three CAN transceivers and two additional CAN controllers. The two additional CAN controllers are controlled by the Jetson via an SPI bus. Figure 4.30 shows the hardware structure of the three CAN buses.

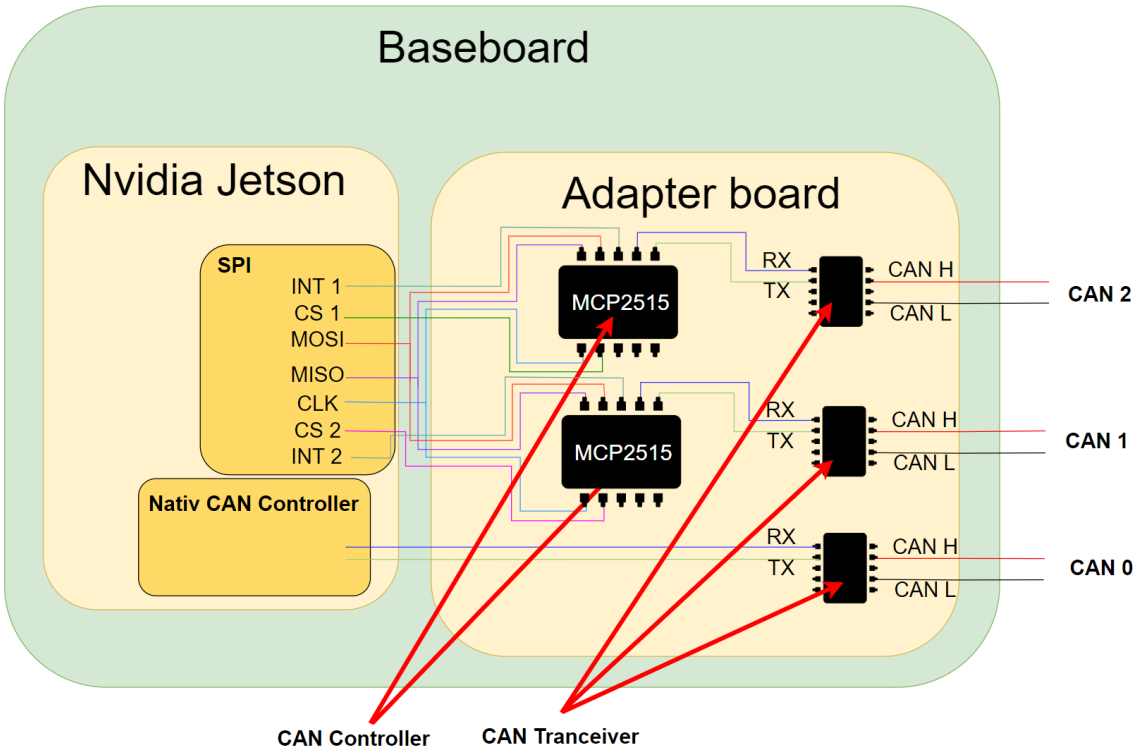


Figure 4.30.: Nvidia Jetson CAN hardware

A kernel driver is required to control the two MCP2515 CAN controllers in exactly the same way as the native CAN controller. Its implementation is described in detail in Chapter 4.4. Finally, the kernel driver enables the two MCP2515 CAN controllers to be controlled in the operating system as an IP link, just like the native CAN controller.

CAN Frame Mapping

As part of this work, a complete mapping of the CAN frames was developed and implemented. The CAN frame mapping has the basic task of storing the received CAN messages in the "sensorData" memory block. However, a mapping must also be available when a CAN message is sent, which is able to take data from the "systemData" memory block and pack it into a CAN message. Figure 4.31 illustrates the structural composition of the CAN frame utilized for the transmission and reception of messages, as well as the internal architecture of the "sensorData" memory block. The objective of the mapper is to transfer the data from the received "can_frame" into the appropriate variables of the "sensor_data_t", or vice versa when sending messages.

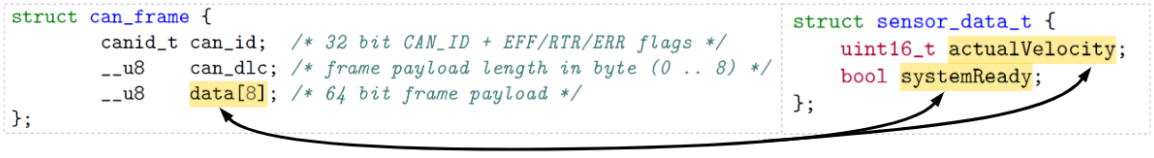


Figure 4.31.: Objective of the mapping

A simple and obvious approach would be to statically map every CAN message that the ECU receives or sends directly in the program. However, this approach would be error-prone and not particularly easy to maintain, as the static mapping of many CAN message requires large amounts of code that is difficult to structure. Static mapping is not particularly modular or easy to maintain, as it is difficult to recognise which parts of the large mapping code need to be changed when the CAN message structure is modified.
Dynamic mapping appears to be a more promising solution. Here, the individual CAN messages and the memory to which they are to be mapped are saved in a separate file. This allows the ECU to create the corresponding mapping links independently when the program is started. This type of mapping has very good scalability and modularity properties, as only the mapping file needs to be changed for future modifications. In addition, no program knowledge is required to make these changes, which makes it very maintenance-friendly.

For the reasons mentioned above, a dynamic CAN frame mapper was implemented in the ECU. The individual CAN messages to be received or sent by the ECU were saved in a JavaScript Object Notation (JSON) file. Figure 4.32 shows an example of such a can message stored in the mapping JSON file. This example message is sent from the inverter to the ECU. The message contains a 16-bit "actualVelocity" value, which is contained in frame payload bits 16-31. The name "actualVelocity" also specifies in which variable in "sensor_data_t" this value must be saved. The message also contains a "systemReady" flag, which is located on bit 8 of the frame payload. However, the "Values" and "Flags" arrays can also contain several elements so that the frame is fully utilised. As can be seen, this JSON implementation is very flexible and customisable in terms of how the CAN frame can be defined. This makes it easy to adapt future changes to the structure of the CAN frame. For example, the "systemReady" flag could also be defined on bit 0. Furthermore, additional flags or values can be inserted into the frame without great effort.

```
{
    "CAN0":[
        {
            "Name": "Inverter",
            "ID": "283",
            "IN/OUT Frame": "IN",
            "Values":[
                {
                    "Name": "actualVelocity",
                    "Start Bit": 16,
                    "Bit Size": 16
                }
            ],
            "Flags":[
                {
                    "Name": "systemReady",
                    "Start Bit": 8
                }
            ]
        }
    ],
    "CAN1":[],
    "CAN2":[]
}
```

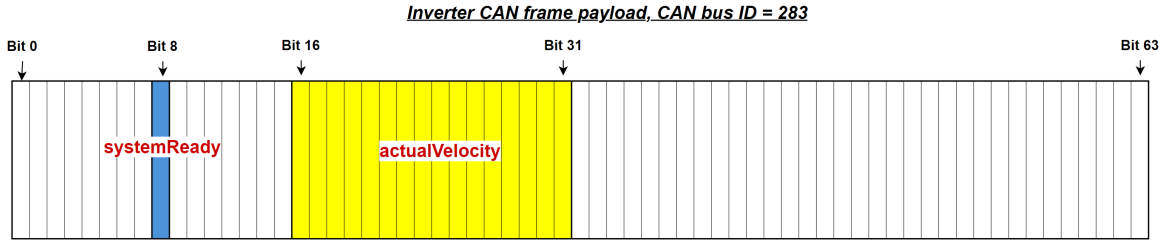


Figure 4.32.: JSON mapping file structure

The mapping program has the task of reading in the JSON mapping file when the ECU is started and saving it in such a way that the CAN messages can be mapped with as few resources as possible. The mapping is created using two different modules. One is the *CANFrameMapper* module, which is located in the *Data* segment with the *CAN* modules. On the other hand, there is the *CANFrame-Builder* module, which is located in the individual *CAN* modules. The structure of the mapping can be seen in Figure 4.33.

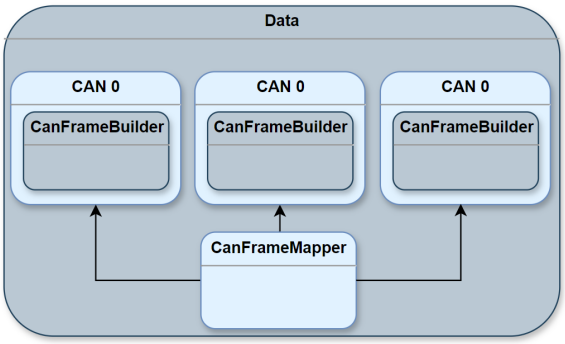


Figure 4.33.: Mapping modules

The *CANFrameBuilder* module is responsible for the conversion between "Raw CAN Frame" and "Structured CAN Frame". This conversion can be observed in Figure 4.34. The "Raw CAN Frame" is the frame that is received and transmitted on the CAN bus. It only contains a 64-bit large payload, which is unstructured. The "Raw CAN Frame" is then structured into a "Structured CAN Frame". This means that it provides direct access to the individual values and flags that the frame contains. Furthermore, the "Structured CAN Frame" also contains the names of the variables to which the values and flags are to be mapped.

Raw CAN frame

```
struct can_frame {
        can_id = 283,
        can_dlc = 3,
        data[0] = 0,
        data[1] = 1,
        data[2] = 100
};
```

Strucrured CAN frame

```
struct can_frame_t {
   name = "inverter",
   canBusID = 283,
   value_map = {{actuelVelocity, 100}},
   flag_map = {{systemReady, true}}
};
```

Figure 4.34.: Conversion between Raw Can Frame and Structured Can Frame

The *CanFrameBuilder* module was developed using a slightly modified builder design pattern [20]. It has both a receive frame map and a transmit frame map in which all possible "Structured CAN Frames" are stored. Figure 4.35 shows the structure of the individual maps. The content of the maps is generated at the start of the program using the JSON mapping file. The detailed functionality of the maps will be discussed later in this chapter.

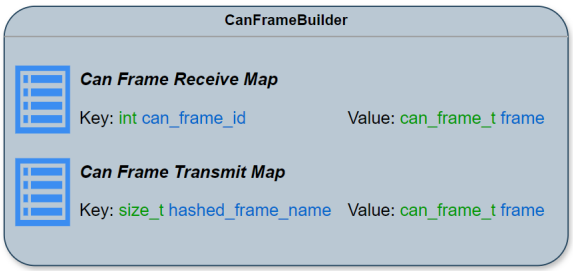


Figure 4.35.: *CanFrameBuilder* map structure

The *CanFrameMapper* module, has the task of connecting the "Structured CAN Frame" with the "sensorData" or "systemData" memory block. This means that when a CAN message is received, the values and flags are written from the "Structured CAN Frame" to the "sensorData" memory and when a CAN message is sent, the values are written from the "systemData" memory to the "Structured CAN Frame". Figure 4.36 shows the process when receiving the CAN message.

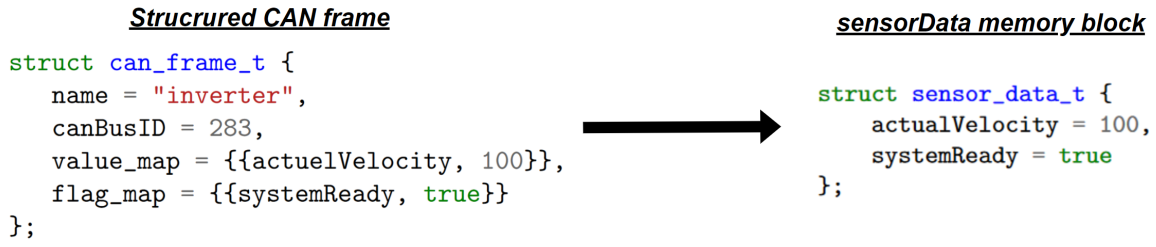


Figure 4.36.: Structured CAN frame to memory mapping

The *CanFrameMapper* was programmed in such a way that it requires a map for this function. This map contains a pointer to each individual element in the "sensorData" and "systemData" memory block. This allows the map to return the corresponding pointer using the memory element name. This map is also loaded with all pointers at the start of the program. Figure 4.37 shows the structure of the map in the *CanFrameMapper*.

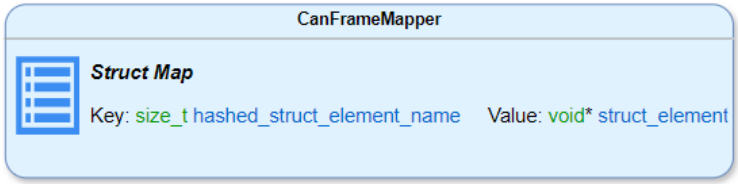


Figure 4.37.: *CanFrameMapper* map structure

Figures 4.38 and 4.39 illustrate the detailed functionality of the two modules *CanFrameBuilder* and *CanFrameMapper*. They demonstrate both the sending and receiving of a CAN message.

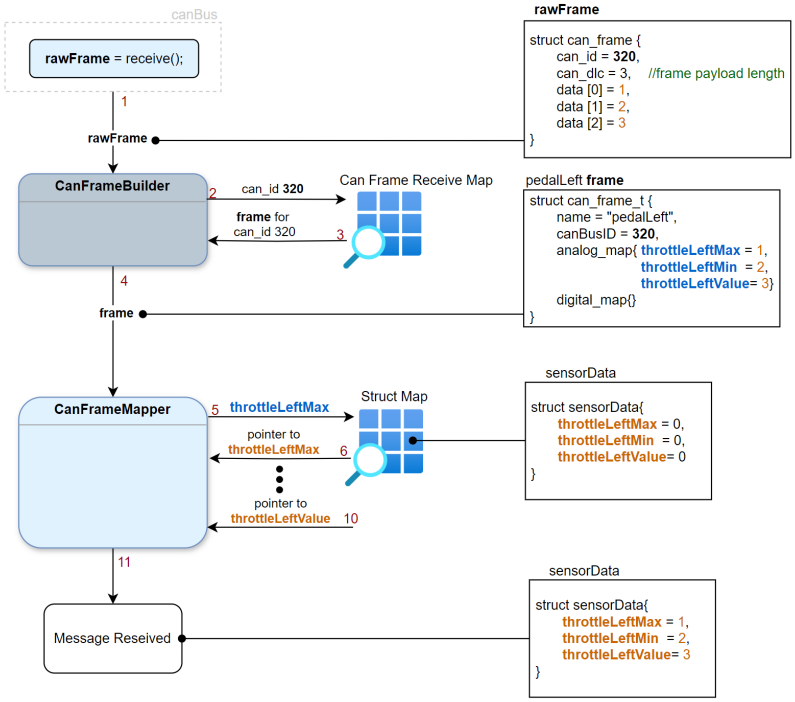


Figure 4.38.: Mapping of a receiver CAN message

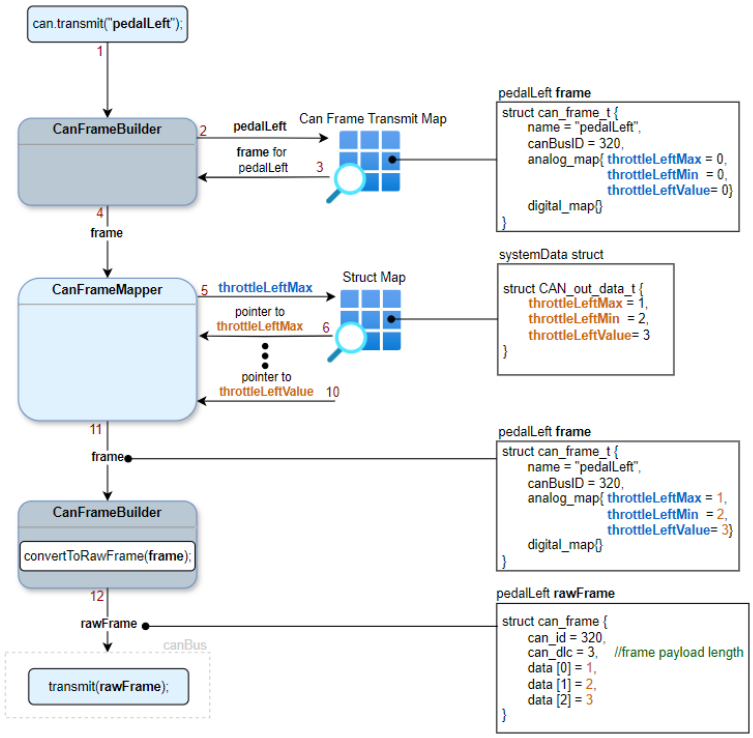


Figure 4.39.: Mapping of a transmitted CAN message

4.2.2. Server

Since the basic functionalities of the server were established in the project work, the structure of the server must be slightly modified in order to be able to implement the concept in chapter 3.3.

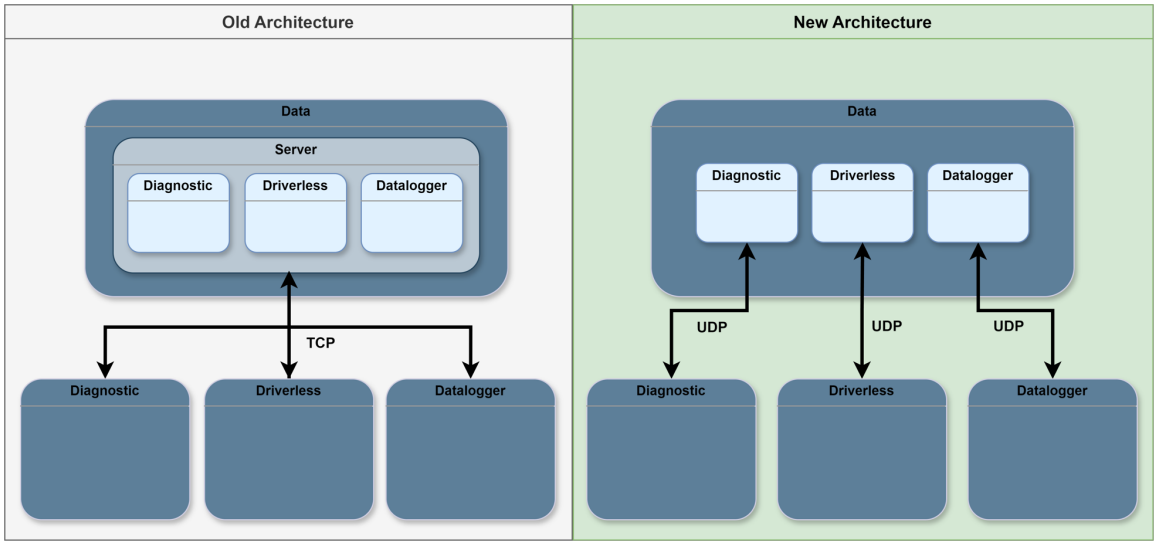
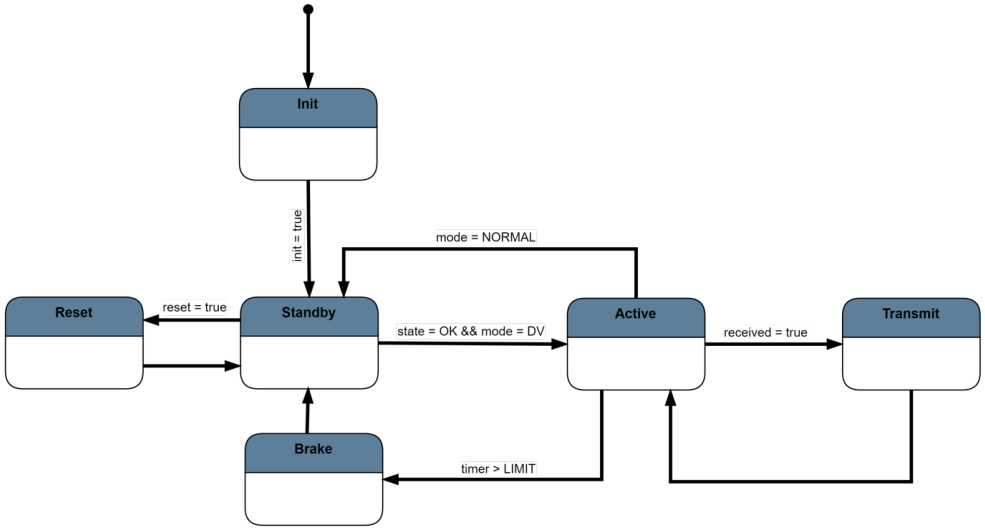


Figure 4.40.: Comparison between old(right) and new(left) architecture

The reason for this is that the architecture, which was created in the project work, does not allow several different servers to run in different threads. It is therefore only a single thread that opens a port to handle all server-relevant tasks, which would make the architecture of the server very complex over time as new functions are added. The structure of the server thread must therefore be divided into the individual smaller server threads *Diagnostic*, *Driverless*, *Datalogger*, as shown in Figure 4.40. This means that a socket must be created for each thread, which results in a port number per application. As a result, it is easy to distinguish between different applications and make the appropriate restrictions.
Also, the transport layer is changed from Transmission Control Protocol (TCP) to UDP. Since UDP offers faster communication and greater bandwidth for exchanging real-time data. In addition, it must be determined within the ECU anyway whether the client transmitted new data within a certain time limit in order to be able to process failures safely. Especially when the control parameters of the driverless are considered.

Driverless

As the driverless system is an independent system with its own hardware components, it currently only requires a small amount of information about the components of the vehicle. These consist of input to set the driverless mode as well as system parameters such as accumulator status and temperature. This is why the data is mainly sent to the ECU to control the vehicle's brakes, throttle and steering. For this reason, the driverless is also one of the system's critical points. A watchdog must therefore be implemented within the *Driverless* module in order to be able to detect a possible failure of the driverless system. This means that as soon as the driverless takes control of the EV, messages must be sent to the ECU at short intervals so that the *Driverless* module within the ECU remains active. In the event of a driverless system failure, emergency braking must therefore be initiated within the *Driverless* module in order to stop the EV as quickly as possible. Emergency braking is the best option here, as there will be no other vehicles on the racetrack during the use of the driverless and potential damage to the EV can therefore be minimized. Figure 4.41 shows a more detailed overview of the structure of the *Driverless* module, with minor details omitted in order to provide a more comprehensive overview.

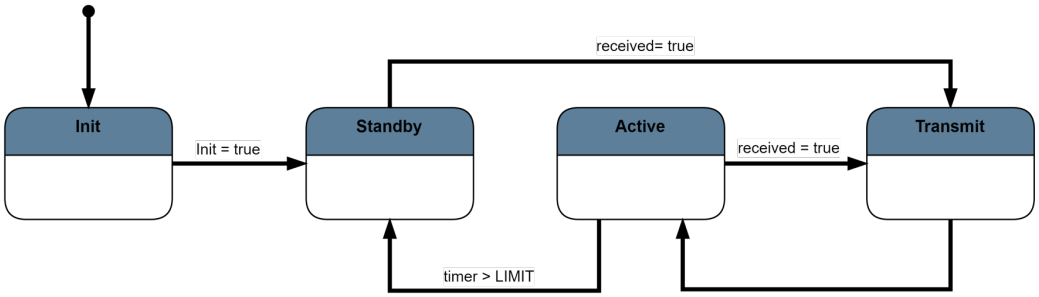


	state Init	*state Standby*	*state Active*	*state Transmit*	state Brake	state Reset
stateAction();	open port create thread init = true	check for reset check for mode sleep for STANDBY_TIME	check for receive sleep for ACTIVE_TIME	read received data transmit data back timer = 0	status = ERROR init brake of vehicle	status = OK
stateUpdate();	init true => state Standby	mode DV && status OK => state Active	mode NORMAL => state Standby received true => state Transmit timer > LIMIT => Brake	state Active	state Standby	state Standby

Figure 4.41.: *Driverless* state machine

Datalogger

The purpose of the external data logger is to record sensor and actuator values from the EV, which are connected to the CAN. This should make it possible to display the overall parameters graphically in order to be able to analyze the behavior of the EV. It is therefore also important to mention that the external data logger does not collect information, warnings or errors about the ECU. This makes the state machine of the *Datalogger* module very simple, as shown in Figure 4.42. In order to conserve resources, a "standby" and a "active" state are defined. The main difference is that longer breaks are taken in the "standby" state to conserve resources. In addition, the *Datalogger* module automatically switches to "standby" if the external data logger does not request any more data over a longer period of time.



	state Init	state Standby	state Active	state Transmit
stateAction();	open port init thread init = true	check for receive sleep for STANDBY_TIME	check for receive sleep for ACTIVE_TIME	read received data transmit data timer = 0
stateUpdate();	init true => state Standby	received true => state Transmit	received true => state Transmit timer > LIMIT => state Standby	state Active

Figure 4.42.: *Datalogger* state machine

Diagnostic

The *Diagnostic* module is intended to provide a simple interface to the ECU, which should enable basic maintenance to be carried out through this module. Examples of this include reading log messages from the ECU, controlling individual hardware elements or changing CAN messages. In addition, chapter 4.5 User Interface addresses the implementation on the client side, which is why this module is closely linked to this chapter. The basic structure of the *Diagnostic* module is very similar to figure 4.42 of the *Datalogger* module, except that more data must be processed in the transmit state in order to support all possible configurations.

As the *Diagnostic* module has a significant influence on the entire ECU and therefore also on the behavior of the EV, it must be viewed critically. A distinction is therefore made between system parameters and configurations. Parameters are understood here as values that can be changed during runtime and can therefore be adapted at any time. System configurations are values that are initialized directly after the ECU is started and do not change during the entire runtime in normal operation.
Consequently, system parameters and configuration have different influences on the system. However, the system parameters have significant importance, as incorrect entries can quickly cause damage to vehicle components. The configuration involves defined values that are responsible for the initialization of certain components. Incorrect configuration means that the system cannot be started up correctly, which is still bad, but does not carry any physical damage.

Protocol

In order to be able to implement the different server modules, a suitable protocol is required that can support all functionalities of the current and possible future modules. The first step is therefore to find a way to organize the data in a structured format within the protocol. As there are very few suitable standards for this purpose, Sensor Measurement Lists (SenML) was quickly chosen. SenML is a protocol that is used in the IoT sector to organise sensor values and their metadata [16].

A corresponding example of a SenML message is shown in Figure 4.43. The first line of the JSON file specifies the source of the subsequent entries. In this case, the base name ("bn") therefore stands for the ECU, with the following "n" entry indicating the more precise position of the values within the ECU. The "vs" is used in SenML for a string entry, in this case it is explained that the following entries are sensor data. It can therefore be concluded that the following variable ("n") entries are sensor values. The unit ("u") entry also makes it apparent that the sensors are acceleration sensors. This metadata allows other applications to implement the value ("v") with little additional descriptions.

```
[
    {"bn":"ECU","n":"cockpit","vs":"sensor"},
    {"n":"x","u":"m/s2","v":4.5},
    {"n":"y","u":"m/s2","v":3.1},
    {"n":"z","u":"m/s2","v":0.5}
]
```

Figure 4.43.: SenML message in JSON format

Of course, this structure also has disadvantages. The biggest disadvantage is that the payloud becomes larger due to the additional metadata. In addition, because it is dynamically structured, the messages must first be processed on both server and client side in order to receive or send the desired data.
Nevertheless, due to its structure, it is very easy to read without having to obtain additional information, which makes troubleshooting simpler. It can also be expanded dynamically, which supports its extensibility. Furthermore, it is a well-documented protocol, which means that many things are already predefined.

Apart from this, the protocol can be set up using the existing hardware. This means that there is not a single data structure, but several small ones, which can be combined to form a large data structure. This dependency also makes it possible to quickly track and adapt changes made to the hardware in the software.

In addition, this data format can also be used to send readable error messages, as it is also possible to send strings as parameters. To reduce data traffic and computing effort, a description of error codes can be retrieved from the ECU during the first data exchange. As a result, the corresponding error messages to the error code only have to be rewritten within the SenML. This promotes maintainability, as error messages can be easily modified or added within the ECU's program.

There are various options for the protocol itself, which is responsible for exchanging the data. However, the protocol must be as light as possible, as the amount of data to be exchanged can be very small. SenML can be used in combination with Constrained Application Protocol (CoAP), which simplifies the overview of a larger network. As this is a local network, with clients connected directly to the ECU over a individual port number, it is also a simple data exchange. In this case, just the SenML data structure can already be used as a protocol, as shown in Figure 4.44. Due to the structure of the base names, the ECU can draw conclusions as to what actions are required. Thus, "bn":"ECU","n":"cockpit","vs":"sensor" makes it clear that sensor data about the cockpit must be sent back to the client. For this reason, the ECU must find the corresponding entries for the according source and attach the values to the original message.

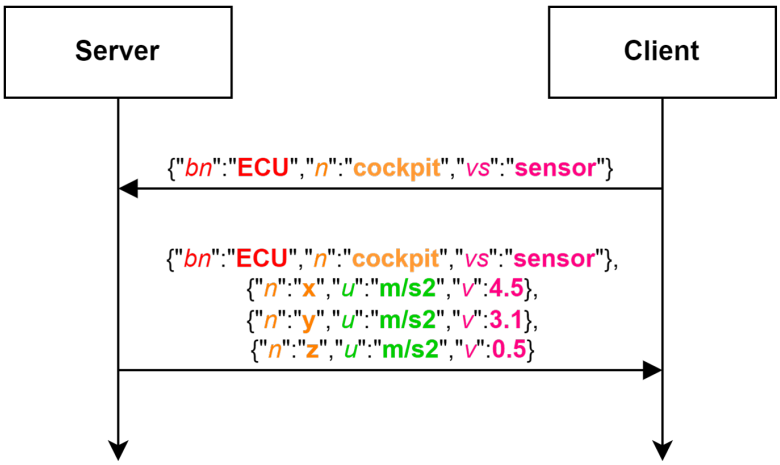


Figure 4.44.: ECU transmits a single SenML messages

In addition, the SenML also offers the possibility that commands can be sent to the ECU, as shown in Figure 4.45. The response of the ECU can be used by transmitting back the received base name to confirm a successful adoption of all values. This structure means that there is hardly any difference between sending and receiving data, which simplifies the handling of communication and makes it easy to understand.

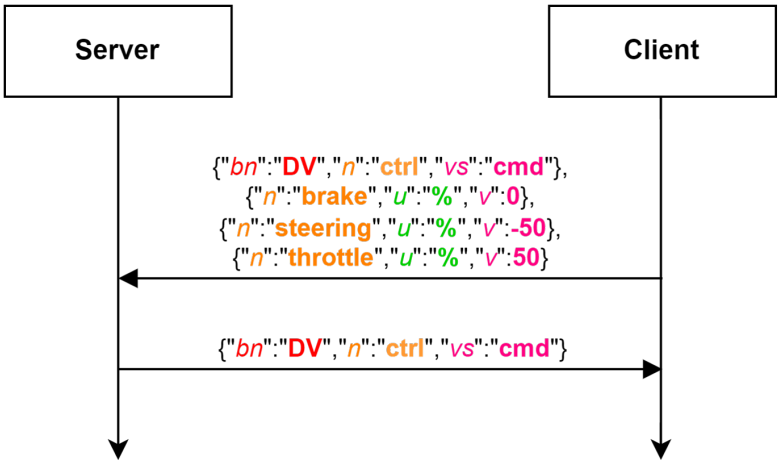


Figure 4.45.: ECU receives a single SenML messages

By differentiating the various base names, it is possible to send and receive data from the ECU over one cycle, which is reflected in Figure 4.46. This reduces the percentage of UDP overhead, allowing data to be exchanged more efficiently. Furthermore, with this approach, the protocol has a robust structure. As long as the variables ("n") follow the corresponding location ("bn"), it does not matter which data comes first. This makes it possible to exchange complex data structures without great effort, as the procedure is always the same.

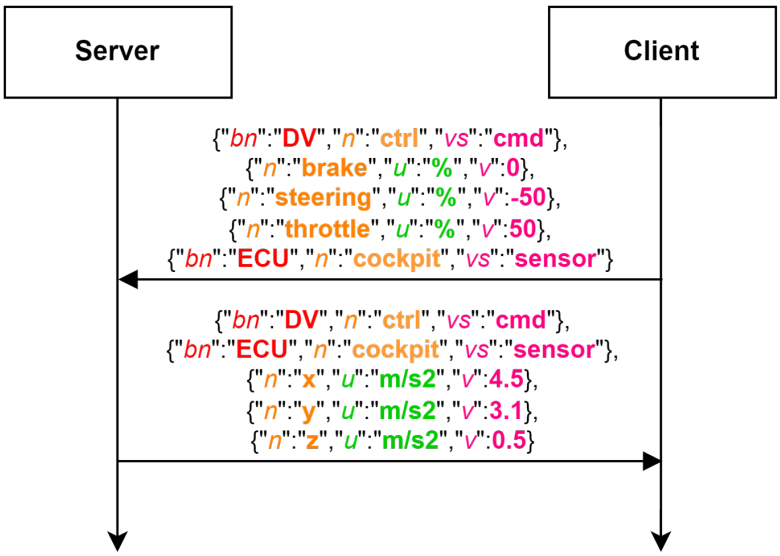


Figure 4.46.: ECU receives and transmit two SenML messages

Data Mapper

As part of this work, a *DataMapper* module was developed and implemented that can map both transmit and receive SenML messages to the "sensorData" or "systemData", as shown in Figure 4.47. The mapper must load the values of a received SenML message into the "sensorData" memory block at the correct location. Conversely, when sending a message, the mapper must take the values to be sent from the "sensorData" or "systemData" memory block and pack them into a SenML message. It is of particular importance that the mapper may only write the received values to the "sensorData" memory in order to continue to guarantee the encapsulation of the "systemData" memory, as previously mentioned in the concept chapter 3.1. In other words, the *DataMapper* module may only have read access authorisation to the "systemData" memory block.

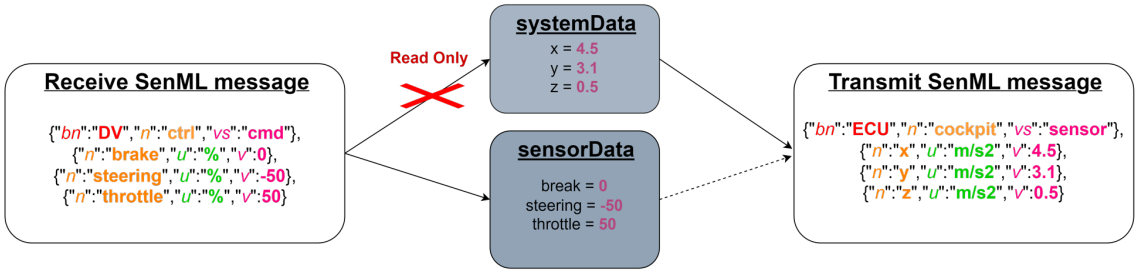


Figure 4.47.: *DataMapper* basic function

As with the *CanFrameMapper* module, the *DataMapper* module was also implemented as a dynamic mapper. This ensures the modularity and maintainability of the mapper as well as its easy adaptability in the future. The dynamic mapping was implemented using a JSON configuration file, which makes it easy to define all SenML messages. Figure 4.48 illustrates an example of the JSON file. All SenML messages are saved in this file as they are to be sent or received later. The only difference between the JSON configuration file and the message is that the value of key "v" or "vb" is not the value, but the name of the variable to be mapped.

```
[

    [
      {"bn":"ECU","n":"pedal","vs":"sensor"},
            {"n":"throttl","u":"%","v":"throttle"},
            {"n":"brake","u":"%","v":"breakPercentage"},
            {"n":"brake pressed","vb":"breakPressed"}
    ],[
      {"bn":"ECU","n":"inverter","vs":"sensor"},
            {"n":"motor temp","u":"c","v":"motorTemp"},
            {"n":"inverter temp","u":"c","v":"inverterTemp"},
            {"n":"IGBT temp","u":"V","v":"IGBTTemp"}
    ]
]
```

Figure 4.48.: *DataMapper* JSON config file

The *DataMapper* module implements a function that reads the JSON configuration file when the ECU is started and then saves it in a map. This map and its detailed structure can be seen in Figure 4.49. The map uses the base name of the individual messages as a key. The value of the map is a struct, which contains both the JSON message and an array with the individual pointers to the variables to be mapped. The mapper is therefore able to use the base name to read and describe the corresponding variables and generate the desired SenML messages. It is also able to save the values of received messages in the correct locations with this map.

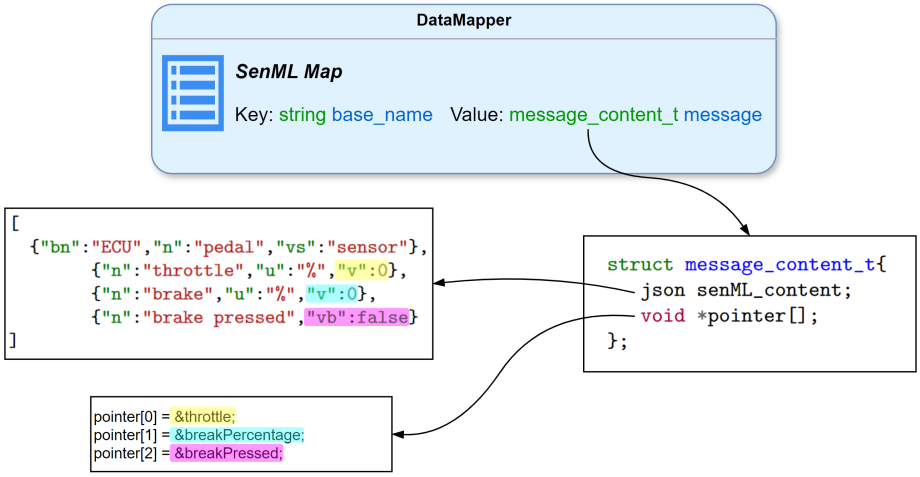


Figure 4.49.: *DataMapper* map

4.2.3. Terminal

The architecture of the *Terminal* module was already defined during the project work. Nevertheless, the structure of the terminal is described here in order to clarify the architecture of the ECU and to be able to cover its traceability.

First and foremost, the terminal is intended to simplify debugging, as it makes it possible to adjust ECU parameters during runtime in order to better analyze the behavior of the ECU. To enable this function, the *Terminal* module is structured as shown in Figure 4.50. Consequently, it is apparent that the terminal has access to the setting and sensor struct, which are located in the *Data* segment. This structure ensures that the *Terminal* module has access to all parameters. This makes it possible to extend the *Terminal* with any number of terminal commands that can change the sensor data and setting data as required.

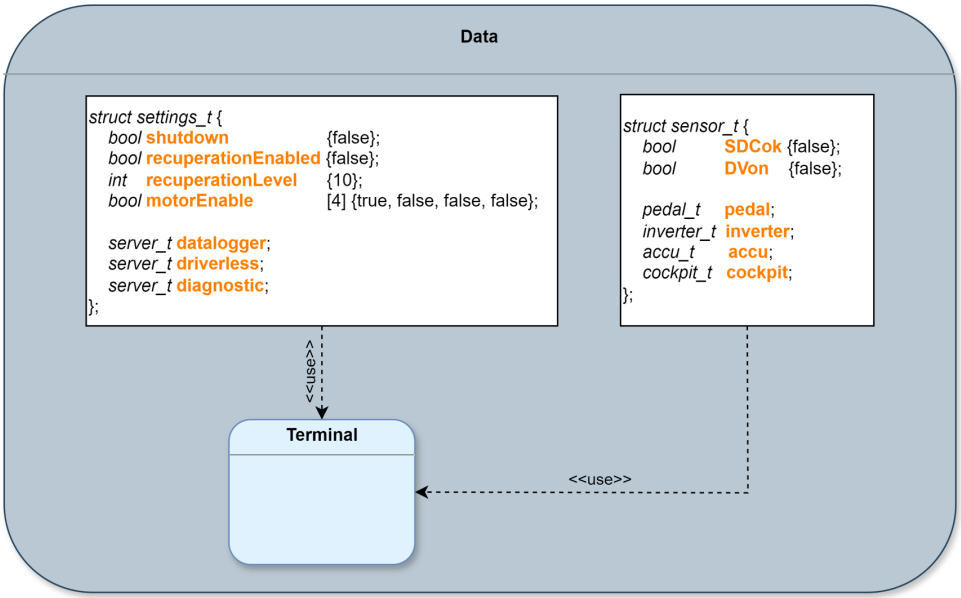


Figure 4.50.: *Terminal* data access

Within this work, the two structs and the *Terminal* module were therefore expanded in order to be able to control the various servers modules and to be able to make adjustments to the ECU. A few examples are listed below.

- system:
  - sys off: Turn EV off.
  - sys on: Turn EV on.
  - sys reset: Turn EV off and on again.
- Server:
  - diagn off: Turn diagnostic server off
  - diagn on: Turn diagnostic server on
  - diagn port 8080: Change port of diagnostic server

4.3. ECU Testing

This chapter describes the various phases of testing. It was decided to use the V-model, which can be seen in Figure 4.51. The reason for this decision is that the entire logic of the new 2024 ECU is based on the logic of the old 2023 ECU. This makes it easy to estimate how the individual functions of the ECU must behave in order for the entire system to function.
For this reason, it is possible to implement all necessary Unit Tests, which are covered in the first chapter 4.3.1.
This is followed by the chapter 4.3.2, in which the Integration Test will be discussed.
The third phase, System Testing is covered in chapter 4.3.3 and enables the entire system to be tested. With regard to the ECU, it is also important to mention that not only the logic is tested, but also the entire hardware and other software components outside of the ECU of the Jetson platform.
The last phase of testing would be the Acceptance Testing phase, which in this case consists of a EV test. Therefore, this phase should check whether all components of the EV are correctly interpreted by the ECU. However, this phase cannot be covered with in this work, as various components were not yet ready to be connected to the ECU at the time of this documentation.

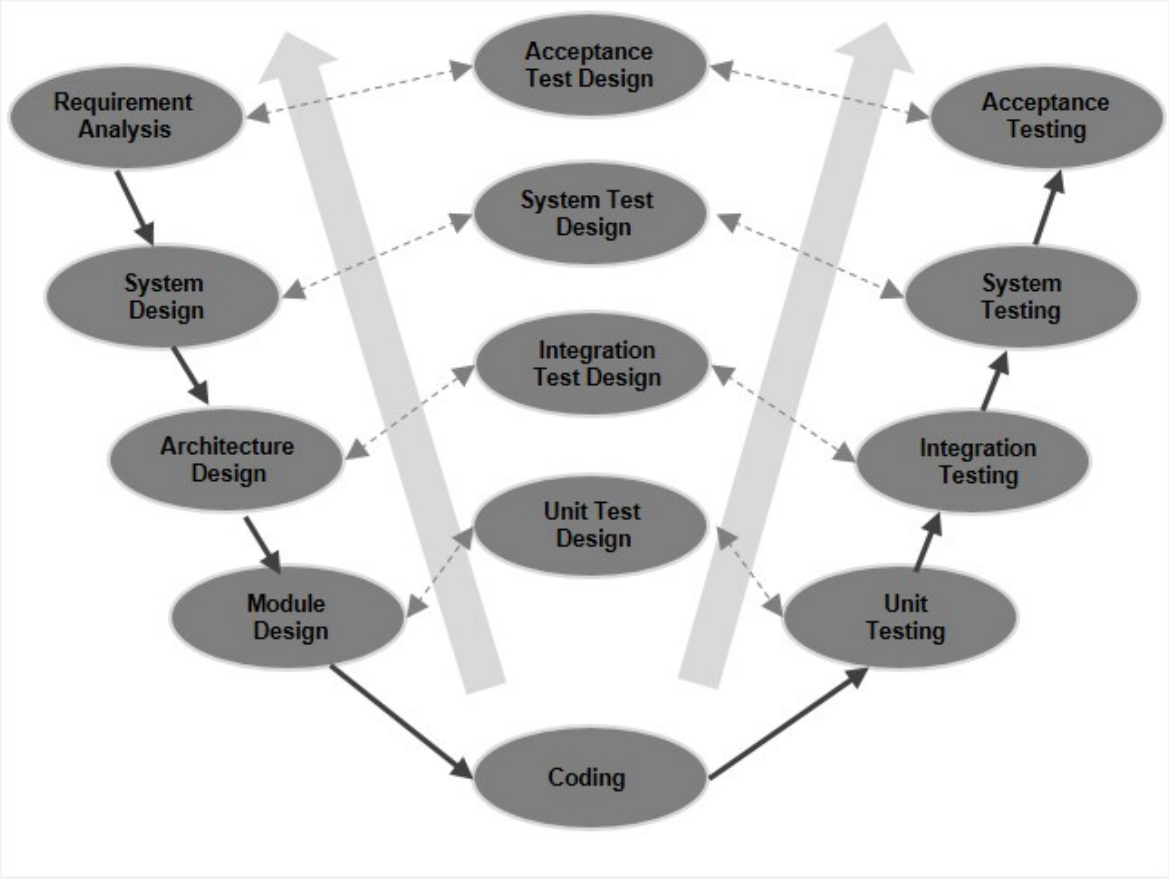


Figure 4.51.: V-Model[21]

It is important to mention that certain modules could not be fully tested. The affected modules consist of modules that are responsible for the external communication of the ECU. Testing these modules requires an effort that is not possible due to time constraints. In addition, these modules are dependent on various external factors, not all of which can be covered, such as the electromagnetic compatibility characteristics of the vehicle.

4.3.1. Unit Test

The GTest and Gmock libraries from Google were used for the unit tests, which are characterised by a high level of user-friendliness [2]. With the help of these libraries, the individual unit tests could be created without much effort. In these unit tests, the individual modules of the *Data* and *System* segments were tested. Each individual module unit test consists of many smaller tests that can test the individual functions of the module. Depending on its complexity, a module has more or fewer tests. The number of tests per module unit test therefore varies between 5 and 40 tests. There are a total of seven different module unit tests, resulting in a total of 100 different tests.
The individual module unit tests were designed in such a way that they can be used repeatedly in the future to verify the functionality of the modules. In the event of future changes to a module, the module unit test can be used to determine whether the module is still functional after the changes. In addition, the tests have been programmed in such a way that they display the functionality of the test on the terminal, making it easier for new software developers to understand the tests.

The interfaces, such as the *CAN* and *Server* module, were not tested with unit tests. The reason for this is that the testing of CAN and UDP interfaces cannot be carried out particularly well with unit test libraries. For this reason, these modules were only tested manually by hand. This means that it was tested whether CAN or UDP messages can be sent and received. The manual test procedure for the interface modules is much faster than programming a dedicated, very complex unit test, but it is not reproducible. This means that if something is changed on the interface modules in the future, they must always be tested manually and cannot simply be tested using a pre-programmed unit test.

4.3.2. System Integration Test

In the system integration test, the individual modules are tested together. This test was also carried out with the Gtest library, but the big difference to the unit test is that several modules are tested together in the integration test [2].

Given the limitations of the Gtest Library in terms of testing the interfaces, it was not possible to perform an integration test of the *Data* segment. This would require a significant amount of time, and was therefore not feasible. For this reason, it was decided that the *Data* segment would only be tested in the EV simulation. However, this has the disadvantage that it is more difficult to recognise or find errors during the EV simulation, as the entire ECU is tested during the simulation and the error radius is therefore larger.

For this reason, only the *System* segment was tested during the integration test. The system integration test primarily tested the starting and acceleration of the motor. However, various error cases such as pedal or accumulator errors were also tested. As with the unit tests, the system integration test was also designed to be as readable and reusable as possible so that this test can also be used to test the entire *System* segment in the event of future changes.

4.3.3. EV Simulation

This chapter discusses the "system Testing" phase of the V-model. The reason why this phase consists of a simulation is because certain hardware components were not available at that time of testing. In addition, a simulation makes it possible to ensure that the ECU behaves as planned, which is particularly important when it comes to controlling motors.

Since this is the first phase in which the logic is tested in combination with the hardware, it is necessary to use additional hardware to test the entire system. The decision for the platform on which the simulation should run was quickly made in opting for an Espressif Systems 32-bit (ESP32) microcontroller, as this platform has many easy-to-use libraries. This simplifies the setup considerably, as only one external CAN transceiver is required, which must be connected to the ESP32 with two pins, as can be seen in Figure 4.52. In addition, a display was connected, which enables all parameters of the hardware to be simulated to be displayed. Thanks to this display, no additional hardware is required to check the behavior of the individual nodes, which should simplify the setup of the simulation and thus also debugging.

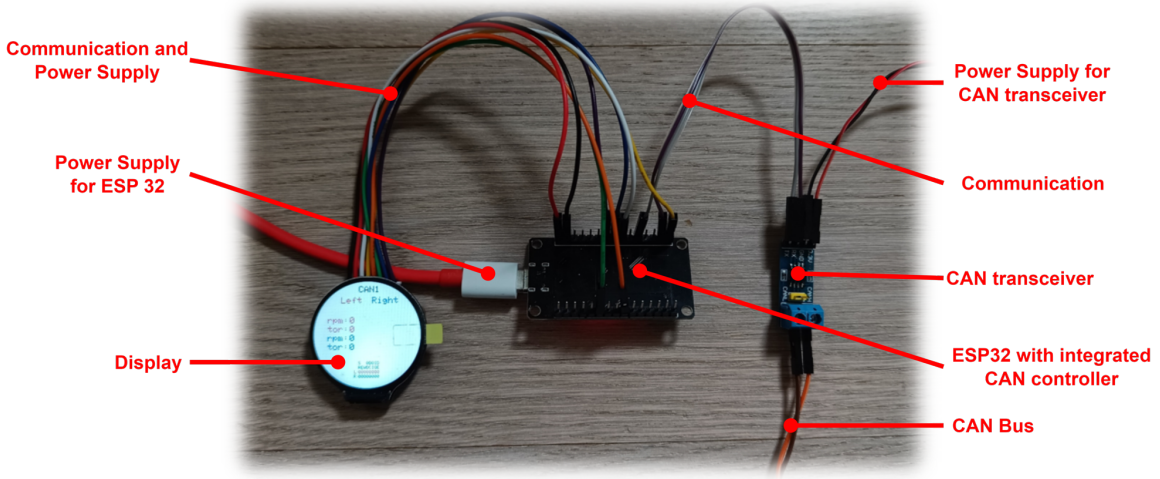


Figure 4.52.: ESP32 node

C++ was chosen for the software structure of the ESP32, as it does not require any rethinking. Furthermore, an architecture similar to that of the ECU can be used, as shown in Figure 4.53. This architecture also makes it possible to simulate several hardware components in the same ESP32, which simplifies the hardware structure.

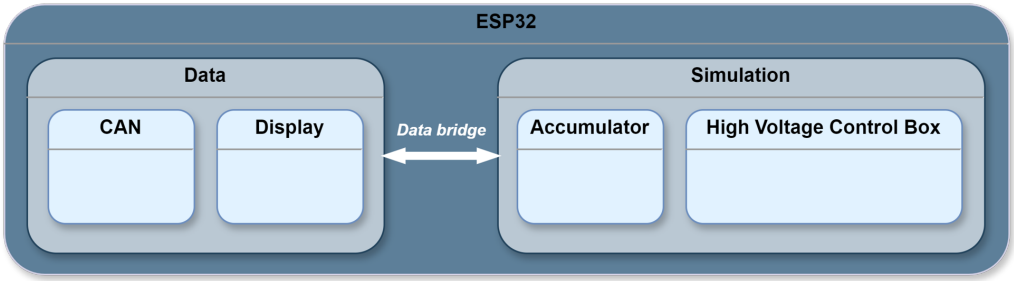


Figure 4.53.: ESP32 architecture

However, it is important for the simulation that the processing and sending of data can be completed within a few milliseconds. A display update influences this process, which is why it is only updated every 500 milliseconds, but a certain time affect can still not be avoided. Nevertheless, this can be neglected as it has no influence on the behaviour of the ECU, as the simulated hardware does not contain any redundant data that needs to be synchronized.

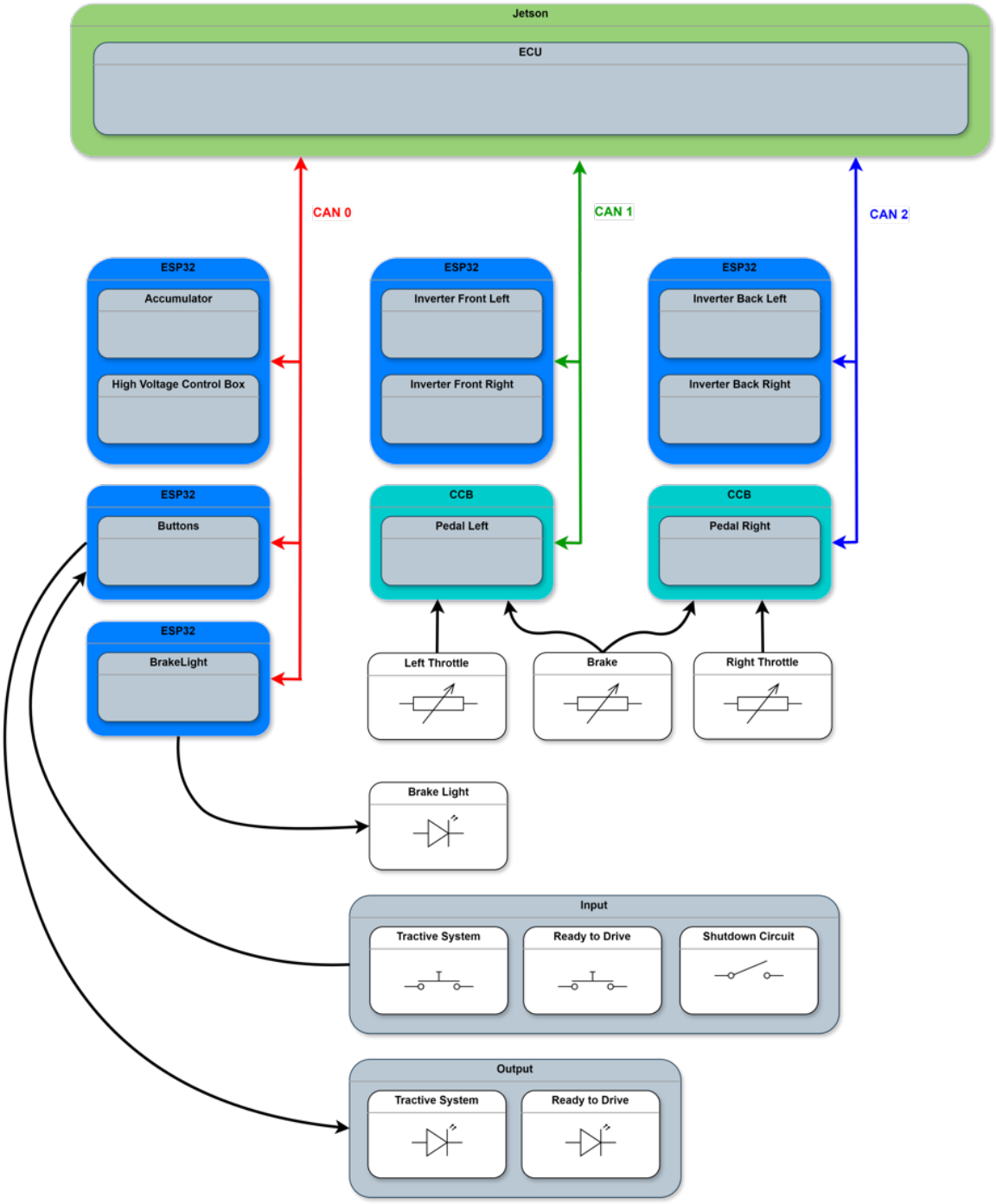


Figure 4.54.: Overview of simulation

The complete assembly of the simulation is shown in Figure 4.54. The simulated hardware is connected to the planned CAN interfaces, to which the real hardware in the vehicle will also be connected. The ESP32 with the “Buttons” and “BrakeLight” simulation are two exceptions, as they do not have a display. In these cases, the inputs and outputs of the ESP32 are used to control the ECU and display status changes.
Furthermore, a CAN Converter Board (CCB) has already been connected to CAN 1 and 2, which will be in the EV and therefore do not need to be simulated. The CCB is a Printed Circuit Board (PCB) developed by UAS Racing Team, which enables sensor data to be sent to the ECU via CAN. It is also possible to send actuator data from the ECU to the CCB, for example to light up an LED.

It is important to mention that CAN 0 has the native CAN controller within the Jetson. CAN 1 and CAN 2 are two Serial Peripheral Interface (SPI)-to-CAN interfaces, which have to share one SPI interface. As a result, it is not possible for CAN 1 and CAN 2 to receive or send messages at the same time, which can lead to a time delay for sending messages. This problem is further increased by the occasional fluctuations of the nodes, which cannot always send messages at the exact same time.

As a result, a problem with the CCBs was already apparent at the start of the first simulation. With the CCBs, it can be seen that the throttle is activated via two potentiometers that are actuated simultaneously, which is ensured by a mechanical construction. Nevertheless, there were error messages because the two messages arrived at different times, which sometimes caused the pedal Left to arrive twice in a row and no synchronization could be achieved. This therefore had to be compensated within the data segment with a greater tolerance in the synchronization of the messages.

In addition, the simulation enabled further logic errors within the system segment to be fixed. These errors are not addressed here, as they occurred within modules that had no direct relation to the architecture or intended logic of the ECU. Similar errors were also found in the simulated hardware, which consequently were solved both within the ECU and in the simulated hardware.
Furthermore, certain hardware parts had to be replaced. One example of this is a CCB that had a defective analog input, which is why the brake always had a differential, although the expected voltage was applied to both CCBs.

4.4. Kernel Driver

One implementation that is not directly visible within the ECU is the interface between hardware and software, which is regulated in the kernel. It is important to note that the Nvidia Jetson Orin NX only has one native CAN interface, which is why the other two interfaces need to be realized using SPI. The native CAN interface has already been successfully implemented in the project work, as only the corresponding drivers had to be loaded. This enables a configuration similar to that of a normal Internet Protocol (IP) connection under Linux and is therefore also structured similarly in C++. The implementation of the SPI-to-CAN interface is basically the same, except that there is a kernel module between SPI and IP interface, which controls the data conversion. After the driver was loaded and the relevant configuration was made, both SPI-to-CAN interfaces could be used to receive and send data.

However, a problem arose from the combination of driver and configuration. In order to understand this problem, the driver for the SPI-to-CAN interface must also be briefly analyzed. The reading of the new CAN data is interrupt controlled, which is very useful for clearing the input buffer as quickly as possible for new incoming data. However, when configuring the interrupt pin on the Jetson, the problem was that the interrupt could only be set to falling edge and not to low level, because the interrupt pin was connected to a General-Purpose Input/Output (GPIO) expander. This expander is in turn controlled by another driver via Inter-Integrated Circuit (I2C), which does not allow an interrupt to be set to low or high level, as otherwise it would always have to be polled via I2C, which would overload the bus. As a result, no falling edge may be missed, as the corresponding CAN to SPI interface would stop without a error message. Since a miss would mean that the interrupt can no longer be cleared and therefore no falling edge can be detected. However, by sending data within a few milliseconds, it could be determined that the interrupt could no longer be cleared and therefore no more data could be received.

To avoid this problem, it was decided to change the reading in the kernel from interrupt to polling. All interrupt relevant code lines were replaced with suitable thread functionalities. During the first test, however, the driver could no longer be unloaded correctly. A closer look at the log messages showed that the thread was unloaded twice, causing a deadlock within the driver. The reason why the driver wants to close the thread twice is because two CAN interfaces are running via the driver. Therefore, two threads had to be created within the kernel module so that the SPI interface can still be used by both CAN interfaces and the thread can be closed correctly.

However, there was a problem with the driver completely blocking the sending of messages at irregular intervals. A closer look at the kernel log files revealed that the message "irq 294: nobody cared (try booting with the "irqpoll" option)" appeared in the kernel at the same time. The "irq 294" is the interrupt pin of the GPIO expander, to which the interrupt pin of the SPI-to-CAN controller "mcp2515" is connected. It was therefore decided to re-implement the interrupt in the driver and deactivate the interrupt after initialization. This solved the problem of the error message and the resulting consequences.

Although the polling approach was effective, an alternative solution was evaluated by changing the interrupt pin. The challenge was that all GPIO pins on the baseboard that tolerate a 3.3 V level are connected via a GPIO expander, necessitating a different interface. Fortunately, the CAN hardware module includes Universal Asynchronous Receiver/Transmitter (UART) and I2C interfaces, both supporting 3.3 V levels. The I2C interface could not be modified as it controls other components on the baseboard. However, the UART interface is solely routed to the CAN hardware module, allowing the Clear To Send (CTS) and Receive (Rx) pins of the UART interface to be reconfigured as interrupt pins. Due to the depth of this operating system modification, the image had to be recreated and loaded onto the Nvidia Jetson. The default SPI-to-CAN driver, "mcp251x," could then be successfully activated, making this solution viable.

Ultimately, the interrupt solution was chosen because it required no modifications to the Linux driver. Furthermore, the driver is only active when the CAN interface needs to receive or to transmit data.

4.5. User Interface

The User Interface (UI) will run on an external computer and therefore must support the most commonly used operating systems found on a commercially available computer. Its purpose is to achieve the functionalities described in the chapter 4.2.2 Server under Diagnostic. Like the ECU, it should be based on C++ so that no major rethinking of the syntax is necessary. This allows implementation to be carried out more quickly on UI and ECU. As a result, the Qt framework was chosen, which makes it possible to build a UI with graphical blocks. This makes it simple to customize the design of the UI. Furthermore, it should be a simple interface, which is why the functionalities within this framework will be more than sufficient to achieve the desired goal.

As with the ECU, the aim of the UI is to ensure modularity in order to be able to implement future functions. First and foremost, it should be easy to use without having any background information about the ECU. For this reason, it is important to set all changeable parameters with the necessary limit values and to inform the user of any incorrect settings that may have been applied by the user. In addition, all modules of the UI must be divided into suitable overviews, as shown in Figure 4.55.

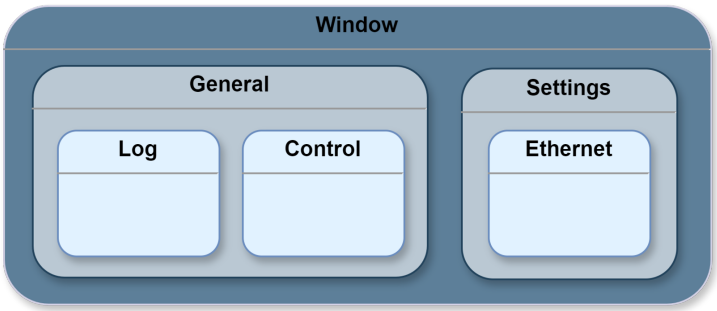


Figure 4.55.: UI *Window* module

In order to create a functional UI, the architecture had first to be determined, as shown in Figure 4.56. The architecture of the UI was divided into a *Frontend* and *Backend* segment and is very similar to the architecture of the ECU. The *UDP* module is responsible for exchanging data with the ECU, while the *Config* module makes the data available to the *Frontend* segment. In this way, the *Config* module serves as a database for all values that are sent from the ECU to the UI. This allows the UI elements within the *Window* module to be set to the corresponding values. In addition, the *Popup* module can inform the user about possible errors. The *Popup* module offers the possibility to inform the user about actions and their consequences, as for example possible errors or the transfer of settings to the ECU.

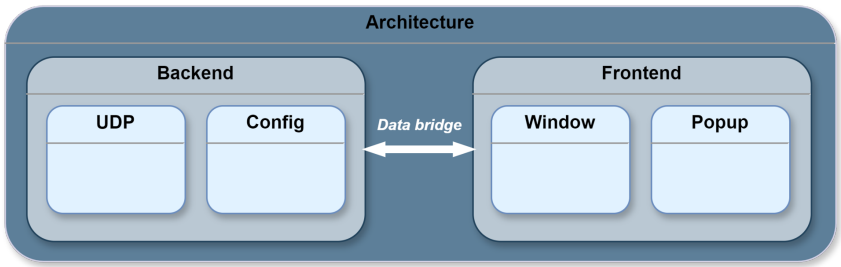


Figure 4.56.: Architecture of the UI

Nevertheless, the UI is not covered in this work to the same extent as the ECU, as it is a completely different platform. First and foremost, the UI is therefore a tool for the ECU to identify possible difficulties. Ultimately, the UI should still offer basic functions that should be helpful for further developments.

4.5.1. General

The general overview is intended to allow the user to interact with the ECU without having to change the configuration within the ECU itself. First and foremost, this is for debugging an error on the EV, which should be made possible by the "Log" panel. Furthermore, it should be possible to control individual actuators with exact parameter values, which in turn should be enabled by the "Control over UI" panel. Both panels need to be executed as threads for two different reasons. The first reason is that control signals have to be sent in regular intervals in order to be able to control the ECU continuously, as the ECU has to reset everything to the default parameters if the connection is terminated. The second reason is that new log messages are constantly being created by the ECU, which must be automatically loaded into the UI.

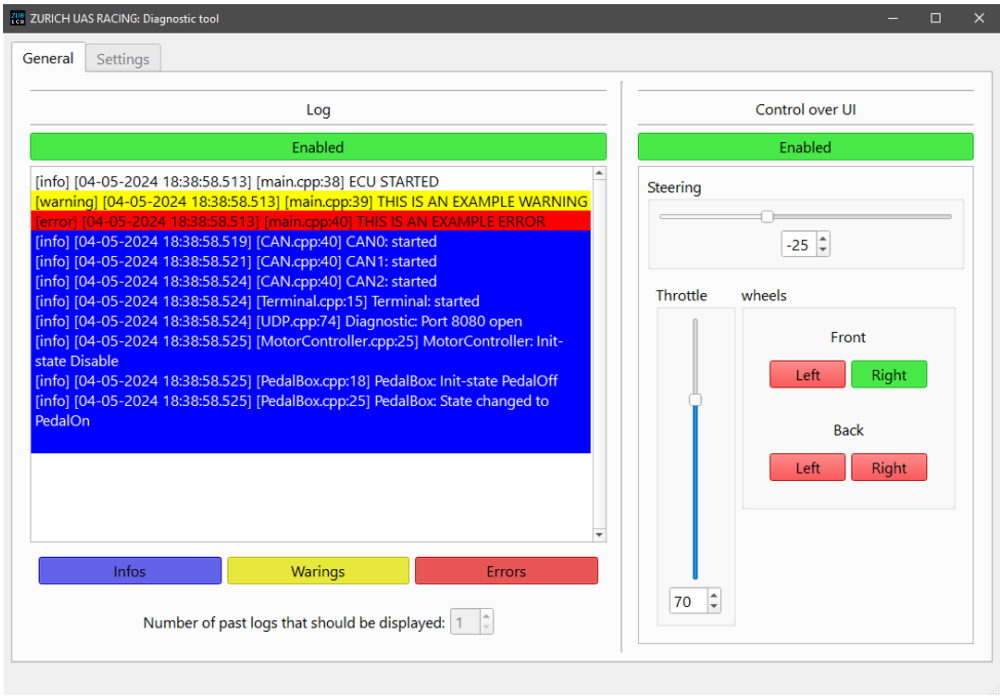


Figure 4.57.: General overview in the UI

Log

The requested logs from the ECU are read directly from the log file and forwarded to the UI. As this is pure plain text, the interval has been limited to 500 milliseconds, as the entire log is sent from the desired point in order to keep communication as simple as possible. All the UI has to do is therefore to differentiate between information, warnings and errors in order to be able to apply the filter. The number of logs can be also set via “Number of past logs that should be displayed” so that logs after the last start of the ECU can be displayed. In addition, the three buttons “Infos”, “Warnings” and “Errors” can be used to blank out the corresponding messages.

Control

The requirement of the "Control over UI" panel is for the ECU to check the basic function of the EV. For this purpose, slider or numerical values can be used to set the desired values. Furthermore, it

is possible to switch the individual inverters respectively the individual motors of the wheels on or off in order to debug possible problems. In this case, the sending of messages is repeated every 100 milliseconds to maintain the connection with the ECU.

4.5.2. Settings

The Settings overview is intended to give the user the opportunity to make changes to the ECU configuration without having direct contact with the code. The aim is therefore to be able to make slight adjustments as quickly as possible, which are not directly related to the logic of the ECU.

Figure 4.58 shows the available settings. The "Power" and "CAN" panels shown in the illustration were not implemented at the time of this work for time reasons. The "Fetch" button on the bottom allows the data of the UI to be synchronized with the ECU again and the "Apply" button saves corresponding changes on the ECU.
The illustration shows that the first steps were taken in "Power" and "CAN" settings. However, these panels could not be completed due to time constraints, as the "CAN" panel in particular is a complex panel that should enable various CAN messages to be processed. Consequently, only the "Ethernet" panel is implemented.

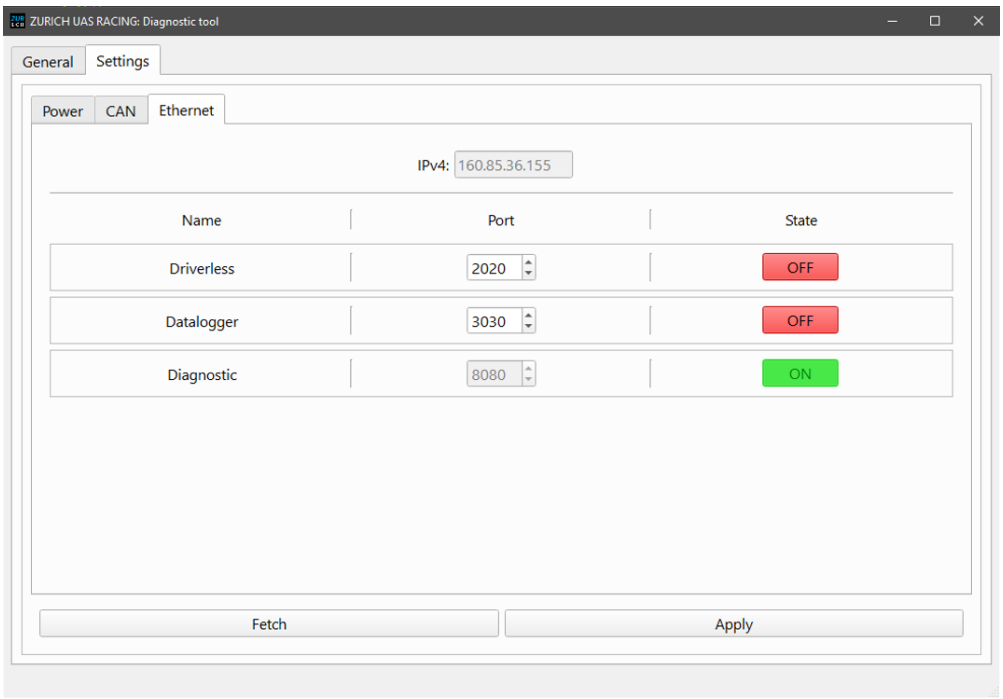


Figure 4.58.: Settings overview in the UI

Ethernet

The "Ethernet" panel is the first panel that was created because it offers the possibility of establishing a connection with the *diagnostic* module of the ECU. When starting the application, it should therefore first be possible to enter the Internet Protocol Version 4 (IPv4) address and the corresponding port of the *diagnostic* module in order to be able to load the configurations from the ECU into the application.
The second function provided by the panel is the ability to configure the *Driverless* and *Datalogger* modules, as shown in Figure 4.58. This allows individual port numbers to be changed or the corresponding module to be switched on or off.

5. Results

The ECU software for the Jetson platform was finalised as part of this bachelor thesis. In addition, the individual components of the ECU were tested intensively using various testing methods, as described in chapter 4.3. For future planning purposes, it is important to know the current utilisation and performance of the ECU in order to be able to estimate how much the ECU can be optimised. This chapter provides an overview of the utilisation of the individual CAN buses, as well as the performance and utilisation of the ECU on the Jetson. It also outlines the difficulties that could not be overcome in the context of this thesis.

5.1. ECU

The "htop" command line tool was used to evaluate the performance impact of the ECU on the Central Processing Unit (CPU). The following Figure 5.1 shows the "htop" view while the ECU is active. As can be seen, the ECU has no significant effect on CPU utilisation. This implies that the CPU cores of the Jetson have sufficient capacity to expand the ECU or to run other applications such as driverless or data logger in parallel.
The Figure 5.1 also shows the various threads that are integrated into the ECU program. It can be observed that the main thread takes up the most resources, which is plausible due to its central function. The terminal and diagnostic threads, on the other hand, require little to no resources, which indicates that they hardly use any processing time in normal operation when they are not required.

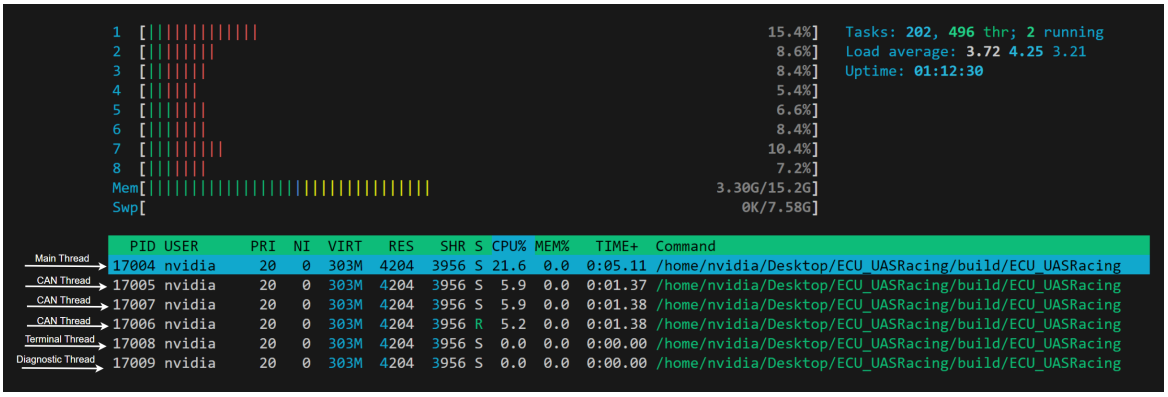


Figure 5.1.: htop view of the Jetson during ECU operation

A profiling of the ECU software was also created using the Chrome Trace Tool [23]. For this profiling, the times of the individual functions are measured during their execution and written to a JSON file. This JSON file can then be loaded into the Chrome Trace Tool, whereupon a profiling is automatically created. The creation of the profiling JSON leads to a significant slowdown of the ECU. This means that the timings in the profiling tool are not correct and are shorter in reality than in profiling. For this reason, profiling is only used to better visualise the ECU process and not to measure timings.

Figure 5.2 depicts an overview of the profiling and shows the individual threads of the ECU. In the main thread, it can be seen that the *Data* and *System* segment is called at a certain interval. Furthermore, it is possible to observe short interruptions of the ECU during which it is blocked by the operating system, which are marked with red arrows in the image. However, these interruptions do not pose any problems for the ECU.

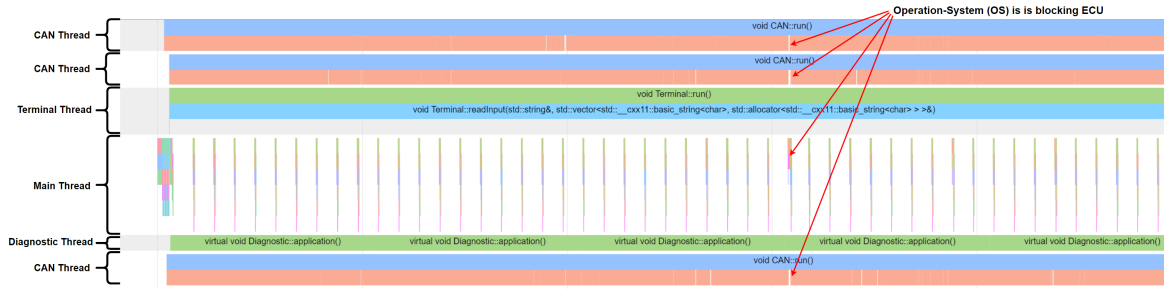


Figure 5.2.: Profiling of the entire ECU

Figure 5.3 illustrates the ECU start-up process. In particular, the start-up of the *CanStructMapper* and *CanFrameBuilder* is clearly recognisable. It can be seen that starting up the entire can-mapping cycle involves a lot of work. This is due to the fact that the ECU must first read in the mapper initialisation JSON files and then generate the maps for the mapping. The high initialisation effort of the mapping ensures that the ECU can map as quickly as possible during operation. Initialisation of the *DataMapper* cannot be seen in this image as for technical reasons it is only initialised when a UDP connection is started, which is not possible during profiling.

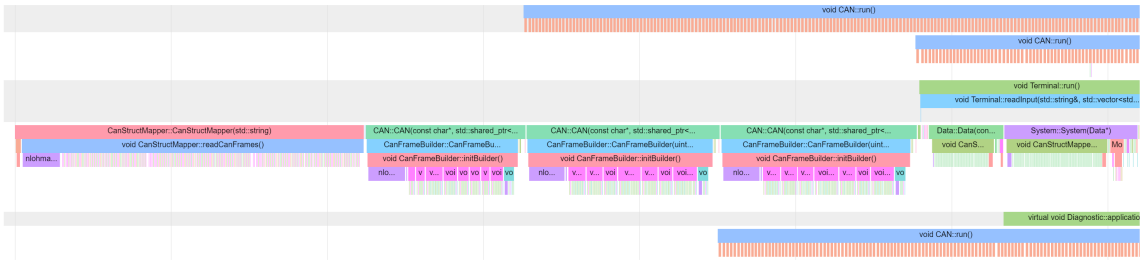


Figure 5.3.: Profiling start up of the ECU

Figure 5.4 illustrates a main thread update cycle in which both the *Data* and *System* segment are updated once. The selected update interval of the ECU is 10 ms. This means that the update function of both the *Data* and *System* segment is called in the main thread every 10 ms. The function of the *Data* segment is to save the CAN messages received in the sensor data memory. The individual system control modules, which fulfill their respective tasks, are called up in the *System* segment. An example of how the system works is the reading of the accelerator pedal value by the *PedalBox* or the calculation of the target torque by the *motorController*, both of which are carried out in the *System* segment. The results of these calculations are then sent to the EV's components via the *Data* segment using CAN.

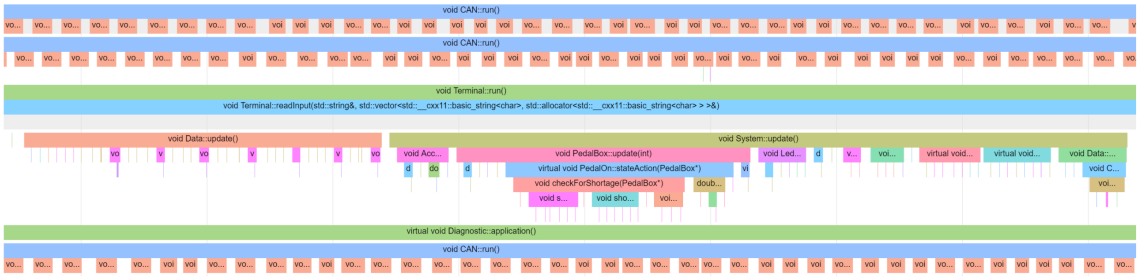


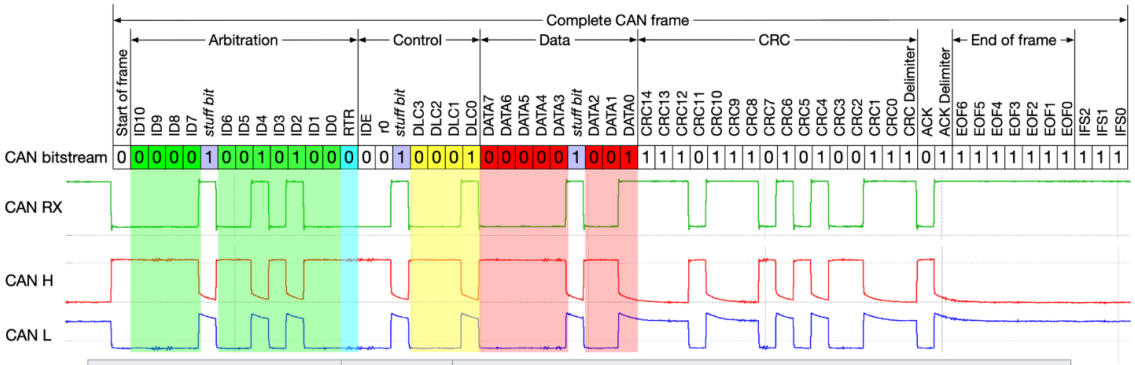
Figure 5.4.: Profiling ECU update cycle

5.2. CAN

The first section of this chapter deals with the overall utilization of three CAN bus interfaces. As the EV's electronics were not yet fully developed at the time of writing, the CAN bus utilization were calculated theoretically and should provide a good reference point. Section two describes the challenges that have arisen in the course of the project in relation to the CAN bus. The last section explains how the problem with the CAN interface has been temporarily solved so that the EV can still participate in racing events during the current 2024 season.

5.2.1. Utilisation

In order to calculate the utilisation of the individual CAN buses, it is necessary to determine the maximum frame rate. This is a challenging task due to the variable frame size of the individual frames caused by bit stuffing. In the present context, the term "bit stuffing" refers to a method in which five consecutive bits of equal value are followed by an inverse stuffing bit. It should be noted that the stuffing bits in question have no influence on the content of the individual frames. This implies that the size of the frame transmitted over the line may vary depending on the content of the frame. Figure 5.5 depicts a CAN frame that incorporates stuffing bits. [11]



Field name	Length (bits)	Purpose
Start-of-frame	1	Denotes the start of frame transmission
Identifier (green)	11	A (unique) identifier which also represents the message priority
Stuff bit	1	A bit of the opposite polarity to maintain synchronisation; see § Bit stuffing
Remote transmission request (RTR) (blue)	1	Must be dominant (0) for data frames and recessive (1) for remote request frames (see Remote Frame, below)
Identifier extension bit (IDE)	1	Must be dominant (0) for base frame format with 11-bit identifiers
Reserved bit (r0)	1	Reserved bit. Must be dominant (0), but accepted as either dominant or recessive.
Data length code (DLC) (yellow)	4	Number of bytes of data (0–8 bytes)[a]
Data field (red)	0–64 (0-8 bytes)	Data to be transmitted (length in bytes dictated by DLC field)
CRC	15	Cyclic redundancy check
CRC delimiter	1	Must be recessive (1)
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
ACK delimiter	1	Must be recessive (1)
End-of-frame (EOF)	7	Must be recessive (1)
Inter-frame spacing (IFS)	3	Must be recessive (1)

Figure 5.5.: A complete CAN bus frame, including stuff bits and inter-frame spacing [11]

Figure 5.6 shows the calculation of the CAN frame length, where "n" stands for the number of data field bytes and can range from 0 to 8. Assuming the worst case, a CAN message with an 8-byte data field is a maximum of 132 bits long. In the best case, a CAN message with an 8-byte data field can also be only 108 bits long, if the message does not require a stuffing bit at all.

frame type	before stuffing	after stuffing	stuffing bits	total frame length
base frame	$8n + 44$	$8n + 44 + \left\lfloor \frac{34 + 8n - 1}{4} \right\rfloor$	≤ 24	≤ 132

Figure 5.6.: Stuffing bit formula [11]

The maximum CAN bus frame rate can now be calculated on the basis of both the worst-case and best-case frame length. The frame rate can be determined taking into account the bit rate, the frame length and the inter-frame space (IFS). The resulting frame rates are therefore based on the bit rate of 500 KBit/s of the EV.

Frame rate formula:

$$\text{Frame rate} = \frac{\text{Bit rate}}{\text{Frame length} + \text{IFS}} \tag{5.1}$$

Best case max Frame rate:

$$\frac{500 \text{ Kbit/s}}{108 \text{ bit} + 3 \text{ bit}} = \underline{\underline{4504 \text{ frame/s}}} \tag{5.2}$$

Worst case max Frame rate:

$$\frac{500 \text{ Kbit/s}}{132 \text{ bit} + 3 \text{ bit}} = \underline{\underline{3703 \text{ frame/s}}} \tag{5.3}$$

In order to calculate the utilisation of the individual CAN buses as a percentage, the number of messages per CAN bus that are transmitted and received per second was counted. The number of frames per second of the individual CAN buses can be seen in Figure 5.7. An overview of the components connected to the CAN buses can be found in chapter 2.2.

	Nr. of frames per second	*Best case utilisation*	*Worst case utilisation*
CAN 0	*280 frames/s*	*6.2 %*	*7.6 %*
CAN 1	*922 frame/s*	*20.4 %*	*24.9 %*
CAN 2	*922 frame/s*	*20.4 %*	*24.9 %*

Figure 5.7.: CAN bus utilisation

The utilisation rates presented in Figure 5.7 are only an approximate guide value. It should be mentioned that the actual values may deviate from the assumptions presented here, as they can be influenced by a variety of factors. One of these is that errors can occur during transmission, which will lead to a retransmission and thus to an additional deviation in the frame rate. Furthermore, in practice it is generally not possible to operate a bus at full capacity. This makes the exact calculation of the utilisation considerably more difficult. However, the calculated best and worst-case utilisation of the CAN bus demonstrate that there is theoretically still sufficient capacity to handle a larger number of components and sensors. This even applies in the event that all CAN nodes of the EV were connected to a single bus. Under these circumstances, the worst-case utilization would be relatively low at only 57 %. This shows that the EV will theoretically not have a utilisation problem in the future with three CAN buses.

5.2.2. Problems

At this point, it should be noted that CAN 0 is the native CAN interface, which has an integrated CAN controller on the Jetson, while CAN 1 and 2 have their own external CAN controllers. Due to the fact that the two external CAN controllers are controlled via the same SPI bus, there is a certain mutual dependency between them. For this reason, individual and joint measurements were carried out. In the joint measurement, both CAN interfaces were operated together, although the frame rate in Figure 5.8 only refers to one CAN interface frame rate.
The maximum transmit frame rate was determined using a bash script and a CAN analyser. The script, which can be found in Appendix A.3, was used to send messages as quickly as possible within one minute. With the analyser CAN it was then checked how many messages were actually transmitted.

	Max transmit frame rate	Max receive frame rate
CAN 0	3700 frames/s	3900 frames/s
CAN 1	900 frame/s	1400 frames/s
CAN 2	900 frame/s	1400 frame/s
CAN1 &CAN2	450 frame/s*	700 frame/s*

* Frame rate of only one CAN interface, but two CAN interfaces are in operation at the same time.

Figure 5.8.: Measured max frame rates

With regard to the native CAN 0, it can be observed that the measured maximum frame rate for both sending and receiving lies between the calculated best and worst case maximum frame rate, as indicated in Formulas 5.2 and 5.3. The values determined for CAN 0 correspond to expectations and demonstrate that the CAN interface is capable of interacting with a load of almost 100 % on the bus. CAN 1 and 2, in contrast, have a significantly lower frame rate, which is almost a quarter of the maximum calculated frame rate. This result is surprising and leads to the possible conclusion that it could be related to the MCP2515 or the associated driver module.
As soon as CAN 1 and 2 are operated together, there is a reduction in the maximum frame rate to half per CAN interface. This is due to the fact that both interfaces have to share the SPI bus and can therefore only be controlled half as often.

The difference between the maximum transmit and receive frame rate can most likely be explained by the fact that the command used for transmitting a message requires more time than for receiving a message. This means that the "cangen" command, which is used for sending random messages, is significantly more labour-intensive. A reason for this is that a random message has to be generated and sent, whereas the receive command “candump” only has to accept the message. This would in turn explain the slightly lower maximum transmission frame rates.

Based on these findings, more precise measurements were made with regard to the receive and transmit behaviour of the CAN interfaces. For this purpose, an attempt was made to investigate the behaviour of the interfaces using different frame rates in order to identify a possible cause of the problem. A time period of one minute was therefore defined for the receive and transmit measurements in order to be able to average the measurements.

Transmit

For the transmit measurements, the same script was used as for determining the maximum frame rate. Only the interval parameter had to be adjusted. With this modification it is possible to send 60 frames within one minute at an interval of one second, as the pseudo code in Figure 5.9 shows.

```
START
  SET START_TIME to current_time()
  SET END_TIME to START_TIME + 60 seconds

  WHILE current_time() < END_TIME DO
    SEND CAN0, MESSAGE
    WAIT 1 second
  END WHILE
END
```

Figure 5.9.: Pseudo code for one second frame interval over one minute

In order to be able to count the CAN messages, an external CAN analyser was used. The counted frames were compared with the transmission counters in the operating system, whereby the values from the CAN analyser and operating system were always the same. During this measurement, all CAN participants were removed from the Jetson in order to be able to determine the maximum frame rates on a given interval without external influence.

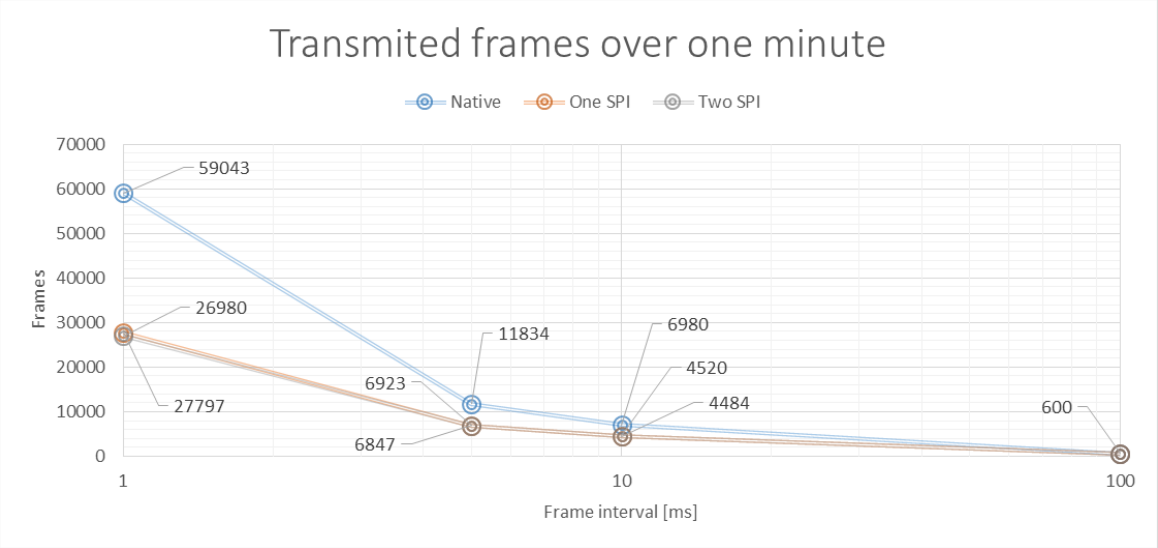


Figure 5.10.: Transmitted frames with different frame intervals

Nonetheless, a gap between the native and SPI-to-CAN is evident in the measurements in Figure 5.10, which increases more and more with smaller intervals. This gap can only be explained by a longer processing time from SPI-to-CAN, which is why the interval until the next message is extended. This behaviour is also noticeable with native CAN, but is hardly tangible due to the faster processing time.

Receive

For the received measurement, the CAN analyser was used as a transmitter, and after the measurement the transmitted frames were compared with the received frames. Thereby it can be determined how many frames could not be read by the Jetson over a period of one minute.

Figure 5.11 displays the collected measurements. The measurements on the left clearly show that the native CAN can read all frames, while the SPI-to-CAN interface has massive difficulties at higher frame rates. The measurements on the right show the detected missed frames of the operating system, i.e. the frames that could not be read and were therefore identified as missing.

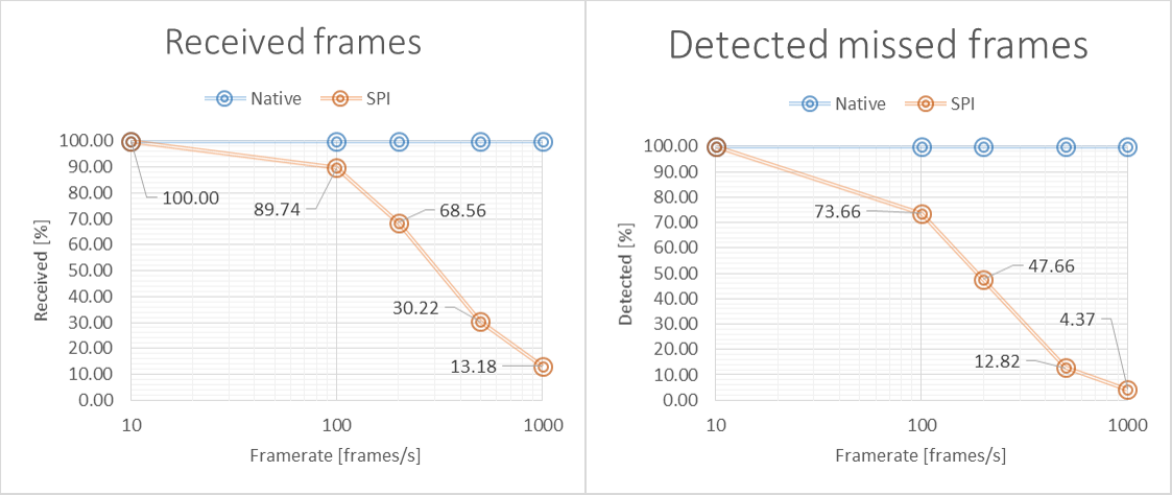


Figure 5.11.: Received frames with different frame rates

The drop in detected missed frames can be explained by the buffer of the MCP2515 CAN controller. When one of the two buffers is overwritten, an error flag is set. However, this error flag can only be set once. Therefore, if the buffer is overwritten multiple times, no new error flag will be set. Consequently, as the frame rate increases, fewer missed frames are detected because the buffer is overwritten several times before the chip can be read out, as shown in Figure 5.12.

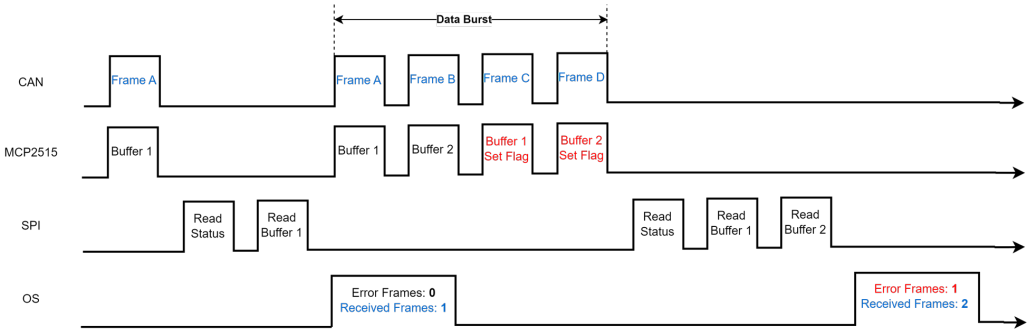


Figure 5.12.: MCP2515 buffer overwrite

Based on the measurements taken, it can be assumed that there is a problem with the SPI communication and the MCP2515. On the one hand, the MCP2515 has too small buffer to be able to intercept all incoming frames. On the other hand, the frames cannot be fetched fast enough from the MCP2515 by the SPI interface, which causes the small buffer to overflow too quickly.

5.2.3. Solution

In order to solve the problem with the SPI-to-CAN interface as quickly as possible due to a lack of time, the CAN bus nodes were split up differently. All CAN nodes that do not require redundancy have been assigned to CAN 0. This means that only one accelerator and brake pedal is connected for each SPI-to-CAN interface. Initially, this means that only one frame type is received per SPI-to-CAN interface in a given interval. As a result, there is a new CAN utilization, which is presented in Figure 5.13.

	Nr. of frames per second	*Best case utilisation*	*Worst case utilisation*
CAN 0	*1970 frames/s*	*43.7 %*	*53.2 %*
CAN 1	*77 frame/s*	*1.71 %*	*2.08 %*
CAN 2	*77 frame/s*	*1.71 %*	*2.08 %*

Figure 5.13.: New CAN utilization

Due to this new distribution, both SPI-to-CAN interfaces are in the range of 100 frames/s. If the previous receive measurements are taken into account, an average of 90 % of the frames can be read when the SPI-to-CAN interface is utilized. This results in no messages from the pedals being received for a duration of 100 ms over a period of one second. As these 100 ms are distributed over the entire second, these misses are therefore in the lower two-digit millisecond range, which is not optimal but tolerable. However, this solution has the consequence that the CAN 0 is under much more traffic. As a result, there is little room for future expansion.

Furthermore, the polling rate of the CAN threads within the ECU was shortened to an interval of a few microseconds. In addition, the involved threads had to be executed with real-time priority, as otherwise the microsecond interval could not be maintained. This change made it possible to increase the CAN interface speed within the ECU. However, the exact reason why this is the case could not be determined. Nevertheless, this improvement leads to a higher CPU utilization, which is shown in Figure 5.14.

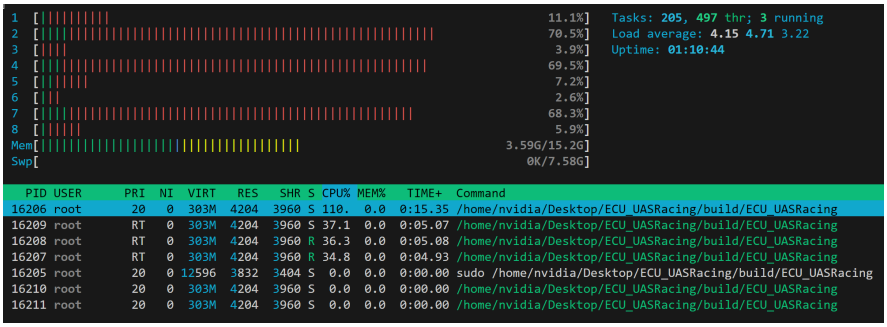


Figure 5.14.: Three CAN threads with real time First In First Out

6. Conclusion

The aim of this bachelor thesis was to develop and implement a new ECU based on a Nvidia Jetson Orion NX platform. This included the further development of the existing ECU software framework, which had already been created in an earlier project. It further involved defining and implementing the various interfaces such as the CAN and UDP interfaces in more detail. The focus for both the ECU software and the interfaces was on making them as modular, maintainable and easy to understand as possible.

The ECU is characterised by its *System* and *Data* segment structure. This makes it possible to replace or add individual modules, ensuring a high degree of flexibility. The modular structure also means that the individual program parts are quick to find and easy to understand. Another key function of ECU are the mappers, which guarante the expandability of the CAN and UDP interfaces. This means that even a non-expert can modify new CAN or UDP messages using a configuration file. This significantly improves the maintainability of the ECU.

Within this bachelor thesis a solid basis for an ECU has been created that can be easily expanded in the future. In contrast to the old ECU, which was installed in the 2023 EV, the new ECU does not have a limiting framework. This means that future expansions are almost unlimited if new modules are to be integrated into the ECU. The ECU software is also relatively platform-independent. This implies that the ECU software can run on an alternative platform that is based on Linux and hascorresponding CAN hardware.

As part of the implementation process, the ECU underwent a comprehensive series of tests, including Unit Tests, System Integration Tests, and an extensive EV simulation. However, it was not feasible to test the ECU on the EV during this phase of the project, as the EV had not yet reached its finalised state by the conclusion of this paper. Additionally, it was only possible to test the UDP communication with the driverless display and datalogger through simulation, as these systems had not yet been finalised by the end of the project.

As part of the work carried out on the ECU, a vulnerability was identified that relates to the CAN hardware. Specifically, this concerns the two MCP2515 CAN controller chips. It was found that these chips could only handle a low frame rate before they started to lose large amounts of frames. The small frame buffer size of two frames in the MCP2515 is most likely responsible for the vulnerability. However, this was resolved during this work by reducing the load on the two affected CAN buses. For this purpose, a large part of the entire CAN traffic was shifted to the one native CAN bus so that the two CAN buses with the MCP2515 chips only receive a few frames per second. Nevertheless, this measure is only a temporary solution and should be rectified for the EV to be released in 2025 at the latest, as otherwise the expansion of the CAN buses will be very limited. For this reason, future work on the ECU is recommended to address this problem.

Despite our best efforts and the investment of significant time and resources, we have not yet achieved the level of perfection we were aiming for in our ECU. However, we believe that the work we have done has laid a solid foundation for future enhancements. Additionally, we have established a framework that will streamline the development of the ECU, potentially leading to its perfection.

Bibliography

[1] linux can. "linux/can-utils". Available:
https://github.com/linux-can/can-utils.
Accessed on: May 5, 2023.

[2] google. "google/googletest". Available:
https://github.com/google/googletest.
Accessed on: May 5, 2023.

[3] UAS Racing Team. "ECU MATLAB/Simulink Code". Available:
https://github.zhaw.ch/FSZHAW/ECU-SW/tree/main/Simulink%20Model.
Accessed on: May 1, 2023.

[4] FSSwitzerland. "Formula Student Rule 2024". Available:
https://formulastudent.ch/docs.php.
Accessed on: May 7, 2023.

[5] FSSwitzerland. "Formula Student Rule 2024" Rule:T6.1. Available:
https://formulastudent.ch/docs.php.
Accessed on: May 7, 2023.

[6] FSSwitzerland. "Formula Student Rule 2024" Rule:T11.8. Available:
https://formulastudent.ch/docs.php.
Accessed on: Dezember 7, 2023.

[7] FSSwitzerland. "Formula Student Rule 2024" Rule:T11.9. Available:
https://formulastudent.ch/docs.php.
Accessed on: May 7, 2023.

[8] FSSwitzerland. "Formula Student Rule 2024" Rule:T14.3. Available:
https://formulastudent.ch/docs.php.
Accessed on: May 18, 2024.

[9] FSSwitzerland. "Formula Student Rule 2024" Rule:T14.9. Available:
https://formulastudent.ch/docs.php.
Accessed on: May 1, 2024.

[10] FSSwitzerland. "Formula Student Rule 2024" Rule:T14.10. Available:
https://formulastudent.ch/docs.php.
Accessed on: May 1, 2024.

[11] "CAN bus," Wikipedia, 2024. [Online]. Available:
https://en.wikipedia.org/wiki/CAN_bus.
Accessed on: May 26, 2024.

[12] "MicroAutoBox III," dspace, 2024. [Online]. Available:
https://www.dspace.com/de/gmb/home/products/hw/micautob/microautobox3.cfm.
Accessed on: April 27, 2024.

[13] "Meilensteine," dspace, 2024. [Online]. Available:
https://www.dspace.com/de/gmb/home/company/companyhistory.cfm.
Accessed on: May 3, 2024.

[14] "MicroAutoBox III," dSPACE, 2024. [Online]. Available:
https://www.dspace.com/de/gmb/home/products/hw/micautob/microautobox3.cfm#179_57263.
Accessed on: May 11, 2024.

[15] "Jetson Linux 35.1," NVIDIA, 2022. [Online]. Available:
`https://developer.nvidia.com/embedded/jetson-linux-r351`.
Accessed on: May 17, 2024.

[16] "Sensor Measurement Lists," rfc8428, 2018. [Online]. Available:
`https://datatracker.ietf.org/doc/html/rfc8428`.
Accessed on: May 1, 2024.

[17] "ZUR TEAM," ZURICH UAS RACING, 2024. [Online]. Available:
`https://zurichuasracing.ch/zur-team/`.
Accessed on: April 23, 2024.

[18] "UNSERE RENNBOLIDEN IM ÜBERBLICK," ZURICH UAS RACING, 2024. [Online]. Available:
`https://zurichuasracing.ch/progress/`.
Accessed on: April 23, 2024.

[19] "State designe Pattern" refectoring guru, 2023. [Online]. Available:
`https://refactoring.guru/design-patterns/state`.
Accessed on: May 7, 2024.

[20] "Builder designe Pattern" refectoring guru, 2023. [Online]. Available:
`https://refactoring.guru/design-patterns/builder`.
Accessed on: May 7, 2024.

[21] "SDLC - V-Model," tutorialspoint, 2024. [Online]. Available:
`https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm`.
Accessed on: May 9, 2024.

[22] "Nvidia Jetson Orin NX Modular Vision System," ZHAW Blog, 2024. [Online]. Available:
`https://blog.zhaw.ch/high-performance/2024/03/11/nvidia-jetson-orin-nx-modular-vision-system/`

Accessed on: May 11, 2024.

[23] "Chrome Trace," google, 2024. [Online]. Available: Chrome browser at
`chrome://tracing/`.
Accessed on: May 26, 2024.

List of Figures

Acronyms

ADC Analog to Digital Converter.

AIR Accumulator Isolation Relay.

ASSI Autonomous System Status Indicator.

CAN Controller Area Network.

CCB CAN Converter Board.

CoAP Constrained Application Protocol.

CPU Central Processing Unit.

CTS Clear To Send.

ECU Electronic Control Unit.

ESP32 Espressif Systems 32-bit.

EV Electronic Vehicle.

GPIO General-Purpose Input/Output.

I2C Inter-Integrated Circuit.

IFS inter-frame space.

InES Institute of Embedded Systems.

IP Internet Protocol.

IPv4 Internet Protocol Version 4.

JSON JavaScript Object Notation.

LED Light Emitting Diode.

PCB Printed Circuit Board.

RES Remote Emergency System.

RTD Ready To Drive.

Rx Receive.

SDC Shutdown Circuit.

SenML Sensor Measurement Lists.

SPI Serial Peripheral Interface.

TCP Transmission Control Protocol.

TS Tractive System.

UART Universal Asynchronous Receiver/Transmitter.

UAS University of Applied Sciences.

UDP User Datagram Protocol.

UI User Interface.

VCC Voltage at the Common Collector.

ZHAW Zurich University of Applied Sciences (German: Zürcher Hochschule für Angewandte Wissenschaften).

A. Appendix

A.1. Code (GitHub Links)

Old 2023 ECU

The following GitHub repository linked contains the folder titled 'Simulink Model', housing the Matlab/Simulink code for the old 2023 ECU:

https://github.zhaw.ch/FSZHAW/ECU-SW

New 2024 ECU

The following GitHub repository linked contains folders titled 'include' and 'source,' housing the C++ code for the new 2024 ECU:

https://github.zhaw.ch/roostim1/ECU_UASRacing

Simulation

The following GitHub repository contains the source code of the ESP32, which was used to test the ECU:

https://github.zhaw.ch/raumar02/ECU-simulation

User Interface

The following GitHub repository contains the source code of the Qt UI:

https://github.zhaw.ch/raumar02/ECU-diagnostic-tool

A.2. CAN Bus Message Overview

Bus	ID	Message	Direction	Cycle	Hardware
CAN0	0x42F	HVCB: Init	output	none	CCB
CAN0	0x420	HVCB: Command	output	17 ms	CCB
CAN0	0x320	HVCB: Info	input	17 ms	CCB
CAN0	0x46F	ASSIS: Init	output	none	CCB
CAN0	0x460	ASSIS: Command	output	100 ms	CCB
CAN0	0x44F	Cockpit: Init	output	none	CCB
CAN0	0x340	Cockpit: Info	input	23 ms	CCB
CAN0	0x410	Accumulator: Command	output	50 ms	Accumulator
CAN0	0x310	Accumulator: Info	input	17 ms	Accumulator
CAN0	0x001	Steering: Command	output	100 ms	CubeMars
CAN0	0x002	Steering: Info	input	100 ms	CubeMars
CAN1	0x41F	Pedal Left: Init	output	none	CCB
CAN1	0x310	Pedal Left: Info	input	13 ms	CCB
CAN1	0x184	Inverter Front Left: Target	output	5 ms	AMKmotion
CAN1	0x283	Inverter Front Left: Actual	input	11 ms	AMKmotion
CAN1	0x284	Inverter Front Left: Info	input	29 ms	AMKmotion
CAN1	0x185	Inverter Front Right: Target	output	3 ms	AMKmotion
CAN1	0x284	Inverter Front Right: Actual	input	7 ms	AMKmotion
CAN1	0x286	Inverter Front Right: Info	input	23 ms	AMKmotion
CAN2	0x42F	Pedal Right: Init	output	none	CCB
CAN2	0x320	Pedal Right: Info	input	13 ms	CCB
CAN2	0x184	Inverter Back Left: Target	output	5 ms	AMKmotion
CAN2	0x283	Inverter Back Left: Actual	input	11 ms	AMKmotion
CAN2	0x285	Inverter Back Left: Info	input	29 ms	AMKmotion
CAN2	0x185	Inverter Back Right: Target	output	3 ms	AMKmotion
CAN2	0x284	Inverter Back Right: Actual	input	7 ms	AMKmotion
CAN2	0x286	Inverter Back Right: Info	input	23 ms	AMKmotion

A.3. Frame Rate Evaluation Script

```
#!/bin/bash

# Get the current time in seconds since the epoch
start_time=$(date +%s)

# Calculate the end time (60 seconds from the start time)
end_time=$((start_time + 60))

# Start generating CAN messages on can0 with specific parameters
# -g 0: use no delay between messages
# -I 01: set the CAN ID to 01
# -L 8: set the data length to 8 bytes
# -p 10: wait for 10 ms if buffer is full
cangen can0 -g 0 -I 01 -L 8 -p 10 &

# Save the process ID of the cangen command
cangen_pid=$!

# Loop until the current time is less than the end time (runs for 60 seconds)
while [ $(date +%s) -lt $end_time ]; do
    # Sleep for 1 second to reduce CPU usage
    sleep 1
done

# Kill the cangen process after 60 seconds
kill $cangen_pid

# Print a message indicating the end of the script
echo "Finished sending messages for 60 s"

# Display network interface configurations
ifconfig
```

A.4. Assignment

Bachelorarbeit BA24_rosn_329

School of Engineering

Studiengang:	ET	Studierende:	Marco Rau (raumar02) Tim Roos (roostim1)
Jahr:	FS 2024		
Betreuer:	Prof. Dr. Matthias Rosenthal (rosn) Patrick Cizerl (cize) Jonas Koch (UAS Team)		
Ausgabetermin:	Montag, 12. Februar 2024	Abgabetermin:	Freitag, 7. Juni 2024
Anzahl Credits:	12 (ca. 360h Arbeitsaufwand)	Präsentation:	Form und Zeitpunkt folgen später

Thema: Formula Student: Fahrzeugsoftware inkl. Anbindung Driverless

1. Einleitung und Zielsetzung

Formula Student ist der grösste Ingenieurswettbewerb der Welt. Studierende planen, zeichnen, bauen und testen innerhalb eines Jahres ein Rennauto, um sich an internationalen Rennen von Juli bis August zu messen. Im Herbst 2019 wurde ein Team an der ZHAW gegründet: Den Verein Zurich UAS Racing. In der Rennsaison 2023 wurde ein erstes Telemetrie und Sensor System am Fahrzeug integriert. Auf dieser Basis soll ein nachhaltiges Konzept für zukünftige Fahrzeuge entwickelt werden. Die Daten Verarbeitung soll über die neue Boardrechnerplattform (Nvidia Jetson) gemacht und wireless zur Auswertung an einen Server übertragen werden. (Datenerfassung, Sensorik, Nvidia Jetson).

Am Institut für Embedded Systems (InES) wurde eine neue, modulare Nvidia Jetson Plattform entwickelt, welche über verschiedene Erweiterungs-Steckplätze optimal an diverse Anwendungen angepasst werden kann. In dieser Arbeit soll die neue Jetson Plattform (Jetson Orin NX) sowohl in Software als auch – falls nötig – in Hardware für einen Einsatz als Boardrechner im Rennfahrzeug des Zurich UAS Racing Teams angepasst werden.

In der vergangenen Projektarbeit wurde bereits die Basis für eine Programmierung der Electronic Control Unit (ECU) auf die Nvidia Jetson Plattform gelegt. In dieser Bachelorarbeit soll nun die komplette ECU auf die Nvidia Plattform integriert werden. Das System soll anschliessend im neuen Fahrzeug von UAS Racing eingebaut und getestet werden.

Ausserdem wird zurzeit an einem Driverless-System gearbeitet, welches bereits auf einem Nvidia Jetson arbeitet. Beide Systeme (ECU und Driverless-System) sollen in dieser Arbeit auf der gleichen Plattform zusammen funktionieren und im Fahrzeug getestet werden. Um möglichst viel Flexibilität zu erreichen, soll dabei mit Containern (Docker) gearbeitet werden.

2. Aufgabenstellung

a. Inbetriebnahme der neuen Nvidia Jetson Orin NX Plattform zusammen mit dem neuen CAN-Bus Interface, welches in der Projektarbeit entwickelt wurde.

b. Vollständige Portierung und Anpassung des Driver-Systems, welches zurzeit vom UAS Team in Matlab vorliegt.

c. Ausbau und Vervollständigung des bereits in der PA erarbeiteten Konzeptes zur Steuerung des Rennfahrzeugs vom Jetson mittels CAN-Bus. Dabei sollen alle Interfaces, Steuerleitungen, Echtzeitfähigkeit und Software-Aspekte einbezogen sein. Das Konzept soll spätestens in der Mitte der Arbeit den Betreuern vorgestellt und mit ihnen diskutiert werden, bevor die Implementation geschieht (Meilenstein-Besprechung).

d. Erste Tests und Validierung der Steuerung direkt am Fahrzeug.

e. Integration des «Driverless» Systems des UAS-Teams zusammen mit der eigenen ECU.

f. Erarbeiten einer Teststrategie und Durchführung ausgiebiger Testreihen im Fahrzeug.

3. Bericht/Präsentation/Randbedingungen

- Die Studierenden übernehmen die Leitung des Projektes. Sie organisieren, leiten und treffen alle nötigen Massnahmen, um das Projekt zu einem erfolgreichen Abschluss zu führen.
- In der Regel findet eine wöchentliche Besprechung statt (online oder vor Ort). Rechtzeitig vor der Besprechung ist wöchentlich ein kurzer Report per Mail an die Betreuer zu senden, mit präzisen Angaben zu den Berichtspunkten „**diese Woche gemacht**“, „**für nächste Woche geplant**“, „**Probleme**“, etc.
- Die Studierenden organisieren, in Absprache mit den Betreuern, eine Meilenstein-Besprechung mit allen Beteiligten. Zu dieser Meilensteinsitzung gehören eine formelle Einladung mit Traktandenliste und ein Protokoll, welches den Beteiligten rasch zugesendet wird und dann im Anhang des Berichtes erscheint.
- Über die Projektarbeit ist ein Bericht zu schreiben. Der Bericht ist termingerecht abzugeben. Es gelten die üblichen Dokumente zum Ablauf und zur Form (Leitfaden_PABA.pdf, Zitierleitfaden, etc.).

4. Bewertungskriterien, Gewichtung:

Das Bewertungsraster sieht wie folgt aus: Arbeitsweise/Ingenieurfähigkeiten/Vorgehen (Gewicht ca. 1/3), Inhalt/Resultate (Gewicht ca. 1/3) und Bericht/Präsentation (Gewicht ca. 1/3).

2

A.5. Project time schedule