CHALMERS
UNIVERSITY OF TECHNOLOGY

Design of a modular centralized E/E and software architecture for a small-scale automotive platform

Master's thesis in Systems, Control and Mechatronics

Erik Magnusson
Fredrik Juthe

DEPARTMENT OF ELECTRICAL ENGNEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

Design of a modular centralized E/E and software architecture for a small-scale automotive platform

Erik Magnusson

Fredrik Juthe

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electric Engineering
Division of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2023

Design of a modular centralized E/E and software architecture for a small-scale automotive platform

Erik Magnusson
Fredrik Juthe

Supervisor: Hamid Ebaldi, PhD in Computer Science, Infotiv
Examiner: Jonas Fredriksson, Professor, Department of Electrical Engineering

Master's Thesis 2023
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Image of autonomous platform generation 4.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

Design of a modular centralized E/E and software architecture for a small-scale automotive platform

Erik Magnusson
Fredrik Juthe

Department of Electrical Engineering
Chalmers University of Technology

Abstract

Autonomous vehicles of tomorrow will require computing units with a high computational capacity to be able to process larger amounts of data. Centralized E/E architecture can fulfill this demand, but it can also lead to cheaper development costs, shorter lead times, and improved modularity of software. Furthermore, it can simplify the development of advanced autonomous driving (AD)- and advanced driver assistance system (ADAS) algorithms that require high computational capacity in real time. This thesis investigates the possibility of using a centralized electrical/electronic (E/E) and software architecture on a small-scale automotive platform.

A centralized E/E architecture with a central master computer has successfully been implemented on an autonomous platform. The autonomous platform is built on an electric go-kart and is augmented with hardware and software to enable self-driving algorithms to control it. The designed system is modular, scalable, and suitable to be used in future research and development. Furthermore, a primitive digital twin to the autonomous platform was developed using Gazebo and ROS2. The software and the digital twin enable rapid model-based development that can be easily transferred to the physical platform twin. The end result is a small-scale automotive platform currently controlled by a drive-by-wire system. Future research would be to implement ADAS and AD algorithms.

Keywords: Centralized E/E Architecture, Autonomous Vehicle, Platform, Go-kart, System Design, Implementation, Modularity, Robot Operating System, Docker

Acknowledgements

We would like to extend our gratitude towards Infotiv in Gothenburg who granted us the opportunity to do our master's thesis there during spring of 2023. Without the help, great feedback and deep discussions with the people working there, this project would not have turned out as it did. We would especially like to thank our supervisor, Hamid Ebadi, PhD in Computer Science, for often believing in us, giving feedback along the way and pushing us to do great work. A big thank goes out to GokartCentralen in Kungälv for letting us test the system there. Lastly, we would also like to thank our examiner, Professor Jonas Fredriksson for being engaged in our thesis work throughout the duration and giving us great feedback along the way.

Fredrik Juthe and Erik Magnusson, Gothenburg, June 2023

List of Acronyms

Below is the list of acronyms that have been used the most throughout this thesis listed in alphabetical order:

AD	Autonomous Drive
ADAS	Advanced Driver Assistance System
ADS	Autonmous Drive Systems
API	Application Programming Interface
AP	Autonomous Platform
CAN	Controller Area Network
CEM	Central Electronic Module
CS	Chip Select
DDS	Data Distrubution Service
DT	Digital Twin
EBCM	Electronic Brake Control Module
ECU	Electronic Control Unit
GPIO	General Purpose Input-Output
HPC	High-Performance Computing
I/O	Input/Output
I^2C	Inter-Intergrated Circuit
IMA	Integrated Modular Avionics
LSB	Least Significant Byte
MCU	Microcontroller Unit
MISO	Master Input/Slave Output
MOSI	Master Output/Slave Input
MSB	Most Significant Byte
OEM	Original Equipment Manufacturer
PCM	Powertrain Control Module
ROS	Robot Operating System
SAE	Society of Automotive Engineers
SDA	Serial Data
SLC	Serial Clock
SPI	Serial Peripheral Interface
SPCU	Steering and Propulsion Control Unit
UART	Universal Asynchronous Receiver/Transmitter
URDF	Unified Robot Description Format

Contents

List of Acronyms **ix**

List of Figures **xv**

List of Tables **xxi**

1 Introduction **1**
1.1 Background . . . 1
1.2 Purpose . . . 2
1.3 Objective . . . 2
1.4 Research questions . . . 2
1.5 Delimitations . . . 3
1.6 Outline of the thesis . . . 3

2 Theory **5**
2.1 E/E architecture . . . 5
2.1.1 Centralized E/E architecture . . . 6
2.1.2 Decentralized E/E architecture . . . 6
2.1.3 Trends in the automotive industry . . . 7
2.2 Autonomous drive . . . 8
2.2.1 Background of autonomous drive . . . 8
2.2.2 Sensors used in autonomous drive . . . 9
2.3 Relevant software and frameworks . . . 10
2.3.1 Containerized software . . . 10
2.3.1.1 Docker . . . 10
2.3.2 Scalable and modular software middleware . . . 11
2.3.2.1 Robot operating system 2 . . . 11
2.3.2.2 ROS2 components and API interfaces . . . 12
2.3.3 Embedded software and systems . . . 14
2.3.3.1 Integrated development environment . . . 14
2.3.3.2 Micro controllers . . . 14
2.4 Data communication . . . 14
2.4.1 Serial communication . . . 15
2.4.2 Control area network . . . 17
2.5 Digital twins . . . 19
2.5.1 Benifits of digital twins . . . 20
2.5.2 Unified robot description format . . . 20

2.5.3 Digital twin simulation environment - Gazebo 21
2.5.4 Digital sensor twin 21

3 Analysis of Previous Generations 23
3.1 Previous generations 24
3.2 Ninebot S and go-kart kit 24
3.2.1 System description 25
3.2.2 Analysis of functionalities 26
3.2.3 System overview and hardware 28
3.2.4 Driveline and steering 30
3.2.5 Bugs and issues 30

4 System Requirements, Specification and Verification 31
4.1 Requirement and specification list 31
4.1.1 Requirements of AP4 33
4.1.2 Specification of AP4 34

5 Hardware Platform Design 35
5.1 System overview 35
5.2 Generic ECU base 37
5.3 Implemented ECUs 39
5.3.1 Central master computer & connectivity 39
5.3.2 SPCU 40
5.4 Modular hardware design 42
5.5 Power module 44
5.5.1 Power consumption 44
5.5.2 Components 44
5.5.3 Battery and charger 45
5.5.4 Power supply unit 45
5.5.5 User interface 46

6 Software Design of Next Generation Autonomous Platform 47
6.1 Software design overview 47
6.2 Linking of software layers 48
6.2.1 High-level to low-level software link 48
6.2.2 Low-level to embedded software link 49
6.3 High level software design 51
6.3.1 High-level control algorithms 52
6.3.2 Digital twin 52
6.4 Low-level software design 53
6.4.1 CAN translation 53
6.4.2 Real-time capabilities in software 59
6.5 Embedded software design overview 59
6.5.1 Implemented embedded software 60

7 Results 63
7.1 Verification results 63

7.2 Physical results on modular design 70
7.3 Images from test day session 73
7.4 Digital twin of autonomous platform 75
7.5 Autonomous platform 4th generation capabilities 76

8 Discussion **79**
8.1 Centralized E/E architecture: key insights from design and development 79
8.2 Validation of requirements 80
8.3 Results from verification 81
8.4 Choices made during development 82
8.4.1 Hardware 82
8.4.2 Software 83
8.5 Future work and research 84
8.5.1 Digital twin 84
8.5.2 Propulsion and steering of autonomous platform 85
8.5.3 Sensors on autonomous platform 86
8.5.4 Battery 86
8.5.5 Autonomous driving functionality 86

9 Conclusion **87**

Bibliography **89**

A Electrical Circuits of Power Module **I**

B Rendered Figures of Other Components **III**

C Bill of Materials **VII**

D Code Listings **XIII**

List of Figures

2.1 Centralized E/E architecture schematic overview including a central master computer, control units, and interfaces. As can be seen, the central master computer is the center for all communication. 6
2.2 Schematic overview of decentralized E/E architecture. Where each domain is controlled through a domain controller, the domains communicate over the backbone. Clearly illustrating the detachment of functions to decentralized domains. 7
2.3 Publisher and subscriber in a node, illustrating how communication and dataflow for topics. 12
2.4 Service request and response between two different ROS nodes. 13
2.5 ROS2 Node algorithm and program structure. The image illustrates how many algorithms inside a node build a program. And how different nodes can split up software capabilities. 13
2.6 Schematic overview of how a ROS network, can communicate and distribute functions among a set of nodes. 13
2.7 Serial Communication using 4 data bits and little Endian is illustrated. There is a transmitter (blue) and receiver (red) which sends data bits and a clock signal from the transmitter to the receiver. . . 15
2.8 Parallell Communication using 4 data bits. In the image, a transmitter (blue) and receiver (red) are illustrated. From the transmitter, there are 4 data bits and a clock signal sent to the receiver in parallel. 15
2.9 Illustration of the relationship between Master and Slave units in I^2C communication protocol. SDA and SCL are sent from the master unit to the slave units. 16
2.10 Illustration of the relationship between Master, Slaves in SPI communication protocol. MOSI, MISO and SCL are connected to every slave device. CS 1 and 2 determine which slave device is currently being communicated with. 17
2.11 Schematic illustration of UART communication. Each device has an RX and TX pin. RX on device 1 is connected to TX on device 2. TX on device 1 is connected to RX on device 2 17
2.12 Illustration of the voltage differentiation in a CAN bus. Where a voltage differential between CAN high and CAN low corresponds to a dominant bit and no voltage differential corresponds to a recessive bit. 18

2.13 Example illustration of ECUs CAN communication between nodes. The nodes are connected together with a CAN high and CAN low cable. Each node has a CAN- controller and transceiver. The two lines are terminated by two 120 Ω resistors 19

3.1 Image of Autonomous Platform third generation (AP3). It shows the current hardware, regarding sensors and computing units. Reprinted, with permission, from Infotiv AB 23
3.2 Ninebot S and Go-kart kit stock configuration Image of go-kart kit before AP4 implementation. 24
3.3 Overview of the Go-kart key components in a schematic view and the schematic of voltage measuring setup. An Arduino UNO is connected to the brake and throttle voltage out wires. 27
3.4 System overview Autonomous Platform generation 3. The system overview illustrates how different components are connected and how functionality is split up. Safety, diagnostics & autonomous drive are separate in HW & SW from power management. Each block is an ECU with specific functionality. Reproduced, with permission, from Infotiv AB 28
3.5 Hardware descriptions of AP3. Illustrating sensors and functionality. Mounted sensors and actuators are illustrated and described in the images. Reproduced, with permission, from Infotiv AB 29

5.1 E/E Architecture of Autonomous Platform 4th generation. This schematic view illustrates what components should be integrated into the system and how they should be connected. As of this thesis, only the SPCU, Camera, Connectivity and battery management system are implemented. 36
5.2 Illustration of the features and components inside each generic ECU base. Communication is handled with a CAN controller and receiver, Power is supplied by two DC-DC converters and a processor which handles computations and can interface with hardware using GPIO pins. 38
5.3 Illustration of how the internal interface of the central master computer and the connectivity capability. The physical communication between high-level software and low-level software is Ethernet. The low-level software is connected to the rest of the system using a CAN controller and transceiver. USB hardware can be accessed by physical ports on the low-level hardware. 40
5.4 Circuit diagram for the Speed and Propulsion ECU and how components are wired. The generic ECU interfaces with the steering using Serial Communication. To control the propulsion, the ECU base is connected to two MCP4725 boards which in turn send analog voltages to the throttle and brake pedals. 41
5.5 Visualisation of modularity on AP4. Illustrating the aluminum plate and how modular designed casings could be mounted in different configurations enabling a flexible physical layout of the system. 42

5.6 Rendered illustration of the generic ECU. Highlighting how components are mounted within and how it interfaces with the rest of the system. 43

5.7 Rendered illustration of the Power module. The main component is the battery holder (red) to which the remaining components are mounted to. The battery holder holds the lead acid battery. The battery charger (yellow) is mounted onto the side. A PSU module (green) is mounted to the side. The power module interface (green) is mounted on top of the battery holder, providing an interface to the power module. An emergency stop, on switch and power connectors. . 45

5.8 Illustration of the power module top interface. Showing the three different power connectors, *AP4 OUT*, *POWER IN* and *BATTERY OUT*. The Emergency killswitch and power switch is also illustrated. The PSU and battery charger and their respective power connector is shown in the bottom om the figure. 46

6.1 Software architecture components, illustrating the relationship between the high-level software, low-level software and lastly the embedded software. Each software is implemented on a separate hardware unit, ranging from HPC down to a simple microcontroller. 48

6.2 Overview of software layers communication inside the HPC unit and respectively HWI unit which runs a docker container on its operative system on Ubuntu. These two software components communicate using Ethernet and pass the ethernet interface to each container. The ROS2 software uses DDS communication which interfaces with the Ethernet to allow software in the HWI unit to communicate with software in the HPC unit. 49

6.3 Common CAN database for each component within the AP4 system. Used for both the low-level and embedded software in the ECUs. In order to encode and decode each CAN message according to a set standard. 50

6.4 Conversion from CAN database to C code schematic flow is illustrated. The dbc file can be changed manually to add CAN frames and signals to the software. The DBC to C external library can be run by a developer when the dbc file has been changed to generate new C files with a specified structure containing every CAN frame and signal. 51

6.5 Overview of High-Level Software where the Gazebo simulator and the vehicle control are actually implemented. However, the overall structure should follow this to enable a structured way of working in AD systems, described in subsection 2.2.1. 52

6.6 Overview of Low-level Software implemented on the HWI unit. It consists of several ROS packages including vehicle control, joystick manual control, CAN translation, and bridge to the high-level software unit. 53

6.7 ROS package to convert topics to CAN frames and vice versa, that consist of three different nodes and the auto-generated c files described in subsection 6.2.2 to encode and decode CAN frames. ROS2_socketcan package interfaces with the CANhat mounted in raspberry pi to enable CAN communication. 54
6.8 Flowchart of the ROS2 Node CAN signal to ROS2 topic which can be seen in the Figure 6.7. That translates CAN objects into ROS topics as well as building CAN frames from ROS topics. 55
6.9 Kinematics of AP4, illustrating the relationship between the base frame's origo, velocity direction and steering angle. The complete description of the kinematics with actual transposes and transformations is described in an URDF-file. 56
6.10 Vehicle Open Loop Controller, consisting of three proportional gains, mapping control inputs to valid values for the steering angle, and voltage for brake pedal and throttle respectively. 57
6.11 ROS package for joystick manual control consisting of two nodes, Joy_node which interacts with device inputs and publishes as topic /joy. Then Telop twist joy converts it into ROS convention of the datatype Twist in /cm_val. 59
6.12 Embedded software template functionalities for the generic ECU base, including libraries to encode, and decode CAN frames. 60
6.13 Embedded software in SPCU including the general code from the generic ECU base template code and special libraries to interface with steering and propulsion modules. 61
6.14 SPCU logic flowchart describing initialization of variables, set up of communication protocols, CAN frame handling, error handling, and how the SPCU interface with the actuators. 62

7.1 Front view of Ap4 including laptop holder, front wing and an overview of the complete system. 71
7.2 Back view of Ap4 including its components mounted such as HWI and router. 71
7.3 Power Module and its corresponding user interface and modules to power the AP4 with a PSU instead. 72
7.4 The implemented SPCU and its corresponding module holders are based on the generic ECU base. 72
7.5 Steering system of AP4 which consist of a steering DC motor and the tooth pulley system that is connected to the steering rods of the go-kart conversion kit. 73
7.6 AP4 around the track, AP4 is controlled through a handheld Xbox controller, enabling drive-by-wire system. Complete a lap around the track on Gokart Centralen, Kungälv. 74
7.7 Driving AP4 on a parking lot, when measuring the brake distance of AP4, utilizing the known distance of each parking spot. 74
7.8 AP4 taking the apex, from drive session of driving the complete track of go-kart Centralen, Kungälv 75

7.9 Digital twin, visualized in Rviz. Which consist of the back wheels which are moving the platform forwards, a body plate, and the front wheels with Ackermann steering geometry. 76

A.1 *Power in* to *AP4 out* flow described in a electrical circuit. Including voltage sensor, current sensor, switches for power on and emergency stop killing all the power in AP4. I
A.2 Battery supply to the outlet *Battery out* flow described in a electrical circuit. Including voltage sensor, current sensor, fuse and emergency stop killing all the power in AP4. I

B.1 Flexible wire holder to drop and click wires and cables, to manage wiring of AP4. III
B.2 Casing for the hardware interfacing computing unit, for the chosen component of Raspberry Pi 4b. III
B.3 Casing for SPCU, steering module (grey) and casing for propulsion, based of the generic ECU casing, figure 5.6. IV
B.4 Mountings for holding the aluminium plate. IV
B.5 Front wings with the standardised grid layout. V
B.6 Laptop holder, placed on the central panel of the go-kart. V
B.7 Speed sensor (yellow)- and limit switch holder, placed on the gokarts steering geometry with an encoder wheel (blue). VI

List of Tables

3.1 Specification of Ninebot S self-balancing segway [46]. 25
3.2 Specification of Go kart conversion kit. Relevant specifications for the gokart conversion kit are placed in a table [47]. 26
3.3 Voltage magnitude for throttle and brake pedals. Also, the time delay between pressing the brake pedals two times in order to enter reverse mode. 27

4.1 Requirement and specification table structure. The table aims to describe how the requirement and specification list are constructed. . 32

5.1 Power Consumption of AP4 summarized. The voltage and current consumption of the major components on AP4 are noted down and converted into a power consumption value. 44

7.1 Platform battery voltage measured at the start and end of the testing session. The session lasted five hours and tested the capabilities of the platform. Utilizing all mounted components, including SPCU hardware node, Raspberry Pi 4, wifi router, and steering motor. . . . 64
7.2 Voltage readings of power supply for the platform during different loads. No load is without moving the steering and with load is whilst moving the steering motor between maximum and minimum limits. . 64
7.3 Platform current consumption measured at the start of the testing session. Utilizing all mounted components, SPCU hardware node, Raspberry Pi 4, wifi router, and steering motor. Idle means no movement of the platform, load means moving the steering motor between its maximum and minimum limits, and load is with a person sitting on the platform. 64
7.4 Measured steering values of steering hardware with respect to requested steering angles in software. 65
7.5 Minimum specification of computing unit compared with chosen hardware to see if it meets the stated specifications. In the table the chosen hardware is a Raspberry Pi 4b. 66
7.6 SPCU timer, measured periods and frequency for running the SPCU code. 67
7.7 Technical specification on the capabilities of AP4 regarding its driving characteristics, software and hardware compatibilities. 77

1

Introduction

1.1 Background

Many large automotive companies such as Tesla, Volvo, Google and General Motors has during recent years focused a lot of their research on developing autonomous driving system (ADS) and autonomous driving (AD)[1]. According to the U.S National Highway Traffic Safety Administration, there are six different levels of autonomy (0-5), and as of 2022 car manufacturers have reached level 4 [1]. This level of autonomy denotes that a human driver does not need to pay much attention to AD's tasks of monitoring the environment and taking action. However, this level 4 is not available for public use. There are still many issues to be solved before reaching level 5, where the human is entirely a passenger. Another perspective on the lower level of autonomy within vehicles involves a lot of different safety systems. Advanced driver assistance system (ADAS) that covers levels 1 to 2, where the system aids the driver with lane keeping, braking, and cruise control.

AD cannot be described as just one technology or system, it consists of several subsystems that make up the complete system [2]. An AD system can be divided into three major components systems: algorithms, client systems, and cloud platforms. Algorithms include techniques to make sense of sensor data, also called sensing, perception, and decision, client systems consist of the physical hardware platform and the operating system to integrate the aforementioned algorithms, and cloud platforms provide heavy computing and storage capabilities for autonomous vehicles.

A major component of the client system is the hardware platform that will either enable the possibility for AD or not, through physical limitations in hardware capability. The hardware platform can be designed through numerous system architectures, the conventional approach is the so-called *decentralized architecture* [3]. A decentralized E/E architecture is where numerous electronic control units (ECU) are placed out with domain-specific functionalities. Each ECU conducts data processing and computations whilst communicating with other domains and ECUs. An E/E architecture that is fairly new to the automotive industry is the *centralized architecture*, where instead the heavy computations are conducted by a single central master computer or a small number of computers [4]. The central master computer has far more computational power than ordinary ECUs that are used in the decen-

tralized case. Also, a centralized architecture has local ECUs, but they act more like a gateway for input/output (I/O) for sensors and actuators[4], thus requiring less computation capability.

At Infotiv, a project called "*Autonomous Platform*" [5] has been active for several years and has gone through 3 generations (AP3). The current generation has undergone several iterations with many different definite solutions, making it hard to integrate or modify existing components and algorithms. Thus it is the purpose of this thesis to construct and design autonomous platform generation 4 (AP4).

1.2 Purpose

The purpose of an autonomous platform is to have the option to develop and test AD and ADAS software functions on a simplified hardware platform. As modern vehicles of today include many interacting subsystems, it is difficult to verify and validate new functions, therefore creating a minimum viable autonomous platform to develop and test features related to autonomous driving and active safety is needed. Furthermore, the platform can act as an education platform to introduce automotive technologies or as a research platform.

1.3 Objective

The objective of the thesis is to develop a new generation autonomous platform with a centralized hardware and software architecture. The proposed system shall be modular so that future research projects can easily modify functionalities and hardware components, to enable a better working process and standard for software and hardware when working with the platform.

1.4 Research questions

These are the research questions that will be addressed throughout this master's thesis report:

- What are the strengths and weaknesses of having a centralized E/E architecture for an autonomous drive on a small-scale vehicle compared to a decentralized one?

- How can a centralized E/E architecture be designed and implemented on a small-scale automotive vehicle such that the system has high modularity and flexibility?

- How can a digital twin of the small-scale automotive vehicle be created to simulate and verify, the perception of its environment and decision-making in its AD algorithms?

1.5 Delimitations

This project will not address:

- *Development of high-level algorithms such as Simultaneous Location and Mapping or other autonomous drive features.* Instead, the focus will be on the low-level algorithms to enable the possibility of high-level functionality. By building the fundamental parts of the Autonomous Platform for communication, driving, steering and sensor readings.

- *The platform on which to implement the functionality.* There already exists a go-kart platform, NineBot Gokart S which consists of an electric drive unit, steering and go-kart chassis. This project will start from scratch on a fresh go-kart.

- *Development of product-ready Printed Circuit Board (PCB) circuits.* In an early stage of development it will not be necessary to develop dedicated electrical boards to satisfy functionality. Since the platform is supposed to be an ongoing modular research project, locking into a specific hardware solution early on may hinder later modifications to the platform. Creating dedicated PCBs will also create unnecessary hardware dependencies.

1.6 Outline of the thesis

The outline and structure of the chapters in the thesis and the content of each are described below. The theory in chapter 2 introduces background theory about the technologies and applications used in this thesis. Specifically, relevant background theory about different E/E architectures, AD, data communication, software, and digital twins. In chapter 3 an analysis of the previous generations of platforms is presented. The chapter summarizes the previous designs and their limitations that laid the foundation for the work done in this thesis. Furthermore, there is also an analysis of the Ninebot Segway S and Ninebot Gokart conversion kit, which is the base system on which the new platform is implemented. Chapter 4.1 contains the requirements for the next-generation AP. Furthermore, measurable specifications and methods to verify that the final result meets the requirements. The hardware of the next generation AP is described in detail in chapter 5. The selected circuit boards, the power supply, and how modular design is achieved are explained in detail. Chapter 6, gives information about the software structure, digital twin, pipelines, and software functionalities, including the set-up of communication, connectivity, and control algorithms of the AP4. The results in chapter 7 contain the verification result, where the method is derived from the verification chapter. Furthermore, there is a summary of the AP4 capabilities regarding control algorithms, differences in E/E architectures, technical specifications, and inclusion of the digital twin. Chapter 8 contains discussions on the methods used, source of faults, validation of the requirements, future research, and chapter 9 presents the conclusions

made.

2

Theory

The theory chapter aims to present the essential theory behind the concepts used in the autonomous platform generation 4. Firstly E/E architecture will be described as a concept in section 2.1, focusing on centralized and decentralized architectures. An introduction to autonomous drive and how it is currently used is described in section 2.2, presenting theory regarding functionality and what sensors can be used. Theory and information about relevant software and frameworks are described in section 2.3, more specifically containerization and a software framework on which to develop functionality. Communication protocols and methods are presented in section 2.4. Lastly, the theory behind the concept of having a digital twin is presented in section 2.5.

2.1 E/E architecture

E/E architectures are how the system of ECUs interact with each other. The ECUs are small programmable embedded systems often having a microcontroller to execute desired functions [6]. The microcontroller interfaces with both sensor readings and actuators through general-purpose input-output (GPIO) pins. Furthermore, an ECU also includes one or several peripherals to communicate with other ECUs. Often a specific function is directly implemented on a single ECU. Some examples of ECU modules are the powertrain control module (PCM) and electronic brake control module (EBCM). Modern vehicles are constructed with a large number of ECUs, often exceeding 100 [6]. Thus it is vital to design an E/E architecture that optimizes the ECU functionalities and the communication among them.

There are two major architectures, decentralized E/E architecture, and centralized E/E architecture. There are several different subcategories as well and there is no strict definition for each architecture, but this thesis will categorize domain-oriented E/E architecture as decentralized and zone-oriented E/E architecture as more centralized. Differences, strengths, and weaknesses will be touched upon during the next sections as well as the trends in the industry.

2.1.1 Centralized E/E architecture

In a centralized E/E architecture several functions and features are located on a single powerful central master computer [6]. The central master computer is a high-performance computing (HPC) unit, capable of very complex and heavy calculations [6]. Instead of traditional ECUs, in a centralized E/E architecture, the ECUs are so-called zone controllers or edge nodes, that act as coordination units [7], see Figure 2.1. The coordination units have limited computing power, are programmed with only relaying functions, and mainly interface with sensors and actuators. Whereas the actual control algorithms are performed in the central master computer [7].

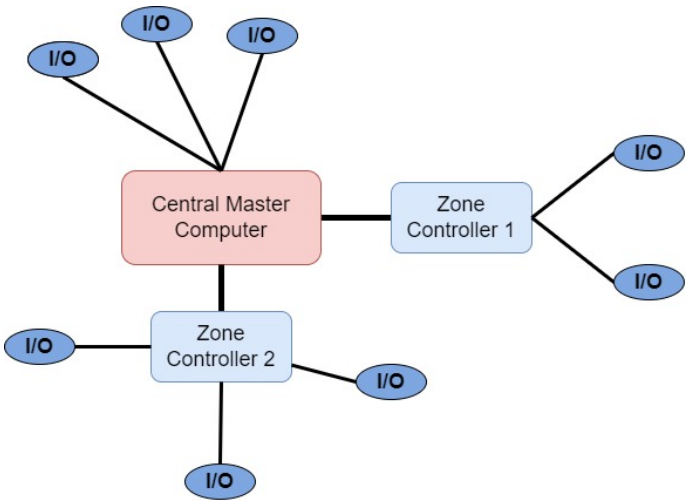


Figure 2.1: Centralized E/E architecture schematic overview including a central master computer, control units, and interfaces. As can be seen, the central master computer is the center for all communication.

2.1.2 Decentralized E/E architecture

The decentralized E/E architecture is the traditional way of designing and constructing a vehicle and its components, [3]. In a decentralized E/E architecture, there is a one-to-one mapping of features/functions and ECU, resulting in many ECUs. The ECUs in a decentralized E/E can process data from sensor readings and make calculations for controlling actuators. Each ECU also communicates with others to achieve higher advanced-level functionalities.

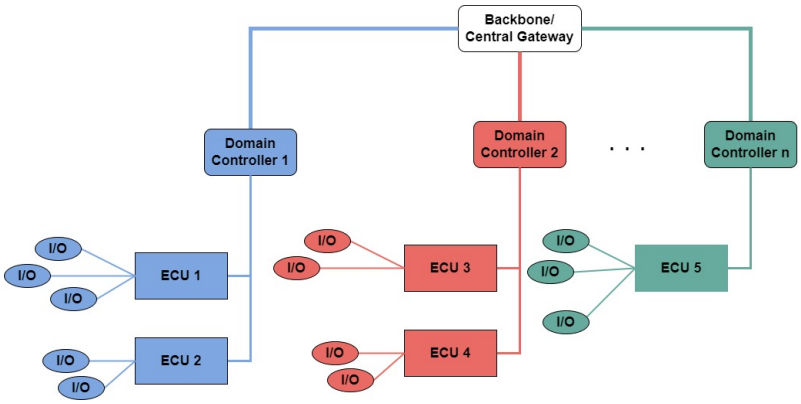


Figure 2.2: Schematic overview of decentralized E/E architecture. Where each domain is controlled through a domain controller, the domains communicate over the backbone. Clearly illustrating the detachment of functions to decentralized domains.

In Figure 2.2 a domain-oriented E/E architecture is illustrated, but in the context of this thesis, it will be considered and categorized as a decentralized E/E architecture. It is also called federated architecture [7],[4], as ECUs with tightly connected features are grouped into domains, like infotainment, active safety, and powertrain. Each domain communicates through intra-domain communication channels and is controlled by the domain controller. Communication between each domain takes place in the central gateway, central electronic module (CEM), also called backbone [4].

2.1.3 Trends in the automotive industry

Due to increasing software complexity and the development of the next generation of vehicles, driven by new technologies such as AD, functional safety, connectivity, and infotainment, there is a need for new architectures, see e.g., [3] and [4]. A key enabler for being able to handle these new technologies seems to be the centralized E/E architecture, [3], and there are numerous already commercialized products with centralized E/E architecture, BMW, Volkswagen, and Jaguar [3]. Furthermore, suppliers of E/E architectures have also shifted towards providing centralized solutions, such as Delphi and Bosch.

The aerospace industry has already faced the same issues regarding decentralized and centralized E/E architectures [3]. The result was a successful transition to centralized E/E architectures. The centralized E/E architecture named Integrated Modular Avionics (IMA), is known for enabling cost-effectiveness in avionics. Advantages are a reduction of computing units, communication channels, and I/O modules. Moreover, IMA helped increase modularity and efficiency in power electronics and hardware use. Since the aerospace industry faced a more complex and strict environment to develop a centralized E/E architecture, it is a testimony that the transition is doable in the automotive industry as well.

2.2 Autonomous drive

This section aims to describe the background of AD, what purpose it serves, and what capabilities exist today. Secondly, the set of sensors and how they can be utilized within an ADS will be presented.

2.2.1 Background of autonomous drive

As mentioned earlier, there are six levels of autonomous drive functionality defined by the Society of Automotive Engineers (SAE) [8]. Level zero has the capability of a traditional fully manual car. As the levels increase the functionality and capability increase readily and a vehicle with level 5 is defined as fully autonomous. Up to level 2 a human driver is still required and responsible for being ready to take control of the vehicle. When reaching level 3, the driver can sit back during nominal operations and take control only if the autonomous system fails. As of 2022, there are no level 3 or higher commercial autonomous vehicles for sale to the public. Even though Audi presented the technology for level 3 autonomy, regulations did not make it possible to sell the vehicle. Most traditional carmakers are still working towards bringing level 2 autonomy to the vehicles sold today. A level 4 autonomous drive can drive itself fully within a predefined area with known conditions. The final level, level 5, is when a vehicle can drive itself under any conditions without the need for a driver.

Vehicle automated driving functionality can be split into five separate components; perception, localization and mapping, path planning, decision-making, and vehicle control, [9]:

- **Perception** is the component that takes in sensor information, processes and transforms it into useful information that subsequent areas can use, [10]. The set of sensors used on the vehicle decides what aspects the vehicle can observe from its environment. Commonly cameras and LiDARs are used.

- **Localization and mapping** plays the role of identifying where the autonomous vehicle is located in relation to its environment, [10]. Information from the perception layer or raw sensory data can be used. Localization is the part that determines the vehicle's position in the environment and mapping is the part of constructing a view of its surroundings. If there exists no map of the environment, these two processes can be done simultaneously using algorithms such as SLAM.

- **Path planning** has the role of creating a path from where the vehicle currently is, to where it should be in the future, [11]. It takes in information about its surroundings from the localization and mapping to plan a suitable path.

- **Decision-making** relies on the information gathered from the perception component to decide on how the autonomous vehicle should act, [12]. There are many things on which to act, some examples are when to change lanes,

when to accelerate, or when to brake.

- **Vehicle control** is the component that is responsible for tracking the trajectory provided from the path planning and decision-making components, [13]. A desired change of vehicle state to reach the desired goal has to be converted into commands that can be executed by the platform. Vehicle dynamics and kinematics have to be taken into account to provide good driving behavior. Often longitudinal and lateral movement of the vehicle is controlled separately. These control commands can then be sent to the accelerator, brake, or steering actuators to change the state of the platform.

2.2.2 Sensors used in autonomous drive

Sensors are vital components of an autonomous system [14]. Sensors allow the algorithms used in self-driving to perceive the environment around the vehicle to feed the decision-making algorithms with information, as mentioned earlier. Using sensory data the environment can be understood, where the road is, where obstacles are, and how the environment is changing. Sensor fusion is a tool where data from multiple sensor sources can be combined to get a better perception of the surrounding environment or to filter out noise and disturbances. Furthermore, using sensor data from several different types of sensors, the set of sensors will cover up each sensor's shortcomings and complement each other with individual advantages to increase the overall perception.

The most common sensors used in AD are cameras, LiDARs and radars [14]. Each sensor has its advantages and disadvantages, the most common ones are described below:

- **Camera** video streams are often used. Images can be fed to algorithms that are very good at recognizing and classifying perceived objects [14]. One potential problem with camera sensor information is that the information in the image may not be useful in every scenario, as objects in the image have to stand out from the background.

- **LiDAR** emits laser beams at extremely high speeds that bounce back from objects and are then detected by a camera detector [14]. The registered beam detection will then compose a three-dimensional point cloud of the surroundings, with information regarding the depth of objects. Thus making the LiDAR sensor a perfect tool to give vehicles a three-dimensional perception of their surroundings. However, the accuracy will drop in bad weather such as rain or snow.

- **Radar** transmits radio waves in pulses, when the waves hit objects they bounce back [14]. Thus giving information regarding the speed and position of the object. Radars work best for highly reflective materials such as metal objects.

Radars work great in bad weather but lack the performance of LiDARs to 3D map the surroundings.

- **Ultrasonic Sensor** transmit and receive an ultrasound, sound waves that are beyond human hearing of 20kHz [14]. The sensor can measure a distance of up to a few meters, making the application of the sensors perfect for low-speed and tight quarters.

2.3 Relevant software and frameworks

There are several relevant software and frameworks related to AD and how they can be developed. The following section describes the software that was utilized to create a coherent and clear software structure for AP4.

2.3.1 Containerized software

Software often requires specific prerequisites, in terms of libraries and configurations specific to the host computer on which the code runs. Containerized software enables simplified packing of several software components, from code to libraries and the underlying operating system into one bundle, called a container [15]. Containerization allows one to define all of this information into one single file and then execute programs using that configuration. Using containerization software this process can be very lightweight in terms of computing power overhead, meaning it will not take up much more resources compared to running the code on a standard computer as usually done. This also makes the created software very portable, meaning it can be run on new host computers easily without having to download additional libraries to make the code run. Containerization of software also makes the applications created very scalable, several instances of the software can be run simultaneously without much performance overhead.

Containerization within the automotive industry has lately started to gain attraction. As an example, Volkswagen, [16], has started to use containerization to test its in-house developed software. When software has been changed, it automatically gets run inside containerization software that lets the software interact with other simulated components in the software stack. This has led to decreased lead times. Volkswagen is currently working on implementing digital twins for its software components using containerization.

2.3.1.1 Docker

A commonly used, free, and open-source containerization software is Docker. It provides a simple software interface that can be used to create highly specialized container software environments [17]. It is a framework to manage, create, and deploy containers with software. In the docker framework, there are three different terms, docker file, docker image, and docker container. The docker file is a blueprint

of how the software environment should be set up, i.e. what underlying software and libraries it requires. A docker file can then be built and compiled into a docker image that consists of all the software libraries specified in the docker file. The docker image can then be started as a docker container and the software will run inside of it. There is an open repository of existing docker files, called Dockerhub, which developers can use as is or modify to meet a project's requirements. It is possible to deploy docker containers on small computing units with limited computational capacity that operate on the ARM architecture, as an example Raspberry Pi 4. Docker containers have the possibility to be real-time capable as presented in [18] as long as the hardware is real-time capable.

Docker Networking is a useful built-in functionality, that makes it is possible to communicate and send information to, from, and between containers [19]. The container's networking can be configured in different modes, such as hosts or bridges to enable different functionalities.

2.3.2 Scalable and modular software middleware

Software for automotive applications needs to meet requirements and have a specific set of functionalities before it can be safely used [20]. These design requirements are modularity and extensibility, performance, simulation and debugging, and usability and support. There exists several frameworks that meet these criteras, such as Automotive Data and Time-Triggered Framework and ROS

ROS was developed to create modular and complex software for robotics that could support distributed computing [20]. The ROS framework implements the underlying communication between distributed software components on a standardized transfer of information system and a way for developers to split up software components into smaller parts but still be able to perform high-level tasks.

2.3.2.1 Robot operating system 2

Since ROS was first developed in 2007, it has gained a lot of attraction within the robot research area and many of its limitations and faults have been made clear [21]. In 2017 its successor, ROS2, was released. This version aimed to fix the shortcomings of the first version and added new functionality such as real-time capabilities and enhanced software security. The core structure of how to use the framework and its application programming interface (API) is very similar to ROS1. It is possible to run the ROS2 software framework inside docker containers, as has been shown in [22]. Utilizing the inherent portability from using containers and the modularity, scalability and flexibility of the ROS2 framework.

Automotive software of tomorrow has a strict set of requirements to fulfill in order to be robust [23]. The software in self-driving vehicles of tomorrow needs to be flexible, real-time capable but most importantly, safe. ROS2 has the capability to match many of the requirements set for the future automotive software by the *AUTomotive Open System ARchitecture* (AUTOSAR) if configured properly. Natively, ROS2 does

not fulfill all these requirements for commercialized vehicles.

2.3.2.2 ROS2 components and API interfaces

Utilizing the ROS framework one can use several distinct components to construct complex software with advanced capabilities. The building blocks are part of the ROS framework API, which means that a developer or system designer does not need to build the infrastructure behind everything, but simply use the API calls. This means a system can quickly be redesigned or augmented in a modular way, the building blocks can simply be rearranged to produce new complex behavior. [20]

Communication in ROS2

Topic is a simple building block [20]. It gives nodes the functionality to communicate important information between different parts of the software network of computational nodes. One part is the **publisher**, which continuously outputs information onto the ROS2 network. The other part is the **subscriber** which listens for information available on the network. To separate different types of information into separate communication channels, topics are used. A Topic contains a topic name and a data field. A computational node only needs to listen to topics that are relevant to the computations. Any other information on other topics will not be received. There is a set of standard data type fields available to send over a topic, such as Integers or strings. A topic can be configured to relay multiple data types creating modular messages sent over the network. This allows for efficient communication between different computational nodes in a software network. Information on ROS2 topics will be pushed to the network even if there is no one listening to that information currently, there is, therefore, no guarantee that the information will be used.

Figure 2.3: Publisher and subscriber in a node, illustrating how communication and dataflow for topics.

Service is another useful building block, it allows one node to request information from a different node in the network and it must respond. A difference from using topics to relay important information is that a service request is always guaranteed to be received and a response must be sent back. In ROS2 this can be done both synchronously and asynchronously by configuration, meaning the program can be configured in such a way that it must receive a response before continuing to do

other tasks. This is a useful feature to guarantee that data or commands have been received at a target node.

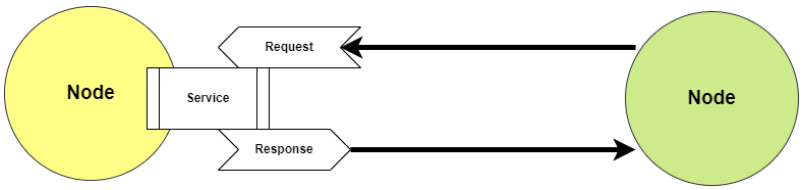


Figure 2.4: Service request and response between two different ROS nodes.

ROS node

A ROS node can be seen as the smallest computational component in a ROS network, commonly called application [20]. It can be a program of any size executing logic or doing computations, as seen in Figure 2.5. Using topics, the node can take in external information or send information out from the node. There are often many nodes in a ROS network doing different tasks, which is illustrated in Figure 2.6. Some can be pure computational, or simply converting data from one topic to another whilst other nodes can interface with hardware, i.e. reading sensory data or sending actuator commands. Nodes can be written in either C++ or Python.

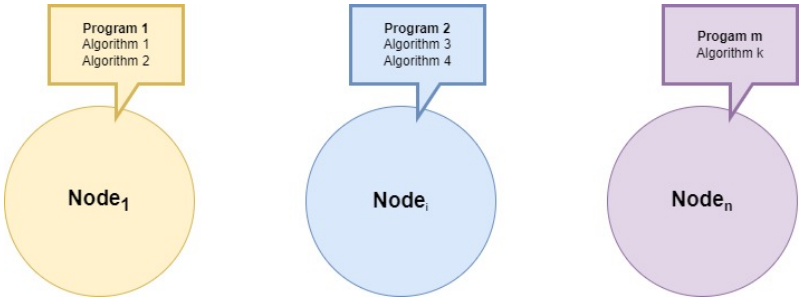


Figure 2.5: ROS2 Node algorithm and program structure. The image illustrates how many algorithms inside a node build a program. And how different nodes can split up software capabilities.

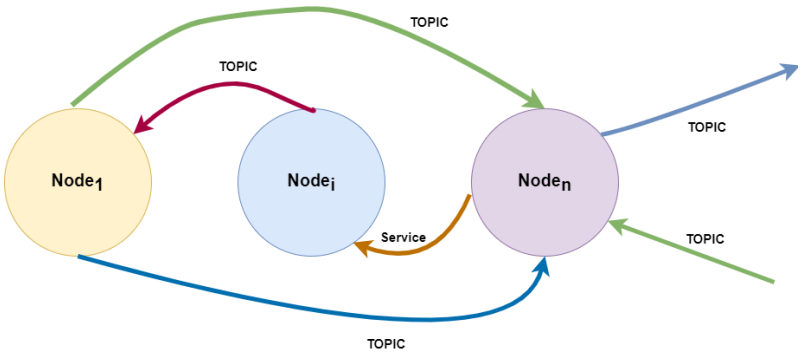


Figure 2.6: Schematic overview of how a ROS network, can communicate and distribute functions among a set of nodes.

2.3.3 Embedded software and systems

Embedded software applications are programs that control specific functions of the embedded system [24]. Unlike PC applications, embedded software applications are developed with fixed requirements in order to execute on a specific device. In the automotive industry embedded systems are referred to as the previously mentioned ECUs.

2.3.3.1 Integrated development environment

Integrated Development Environment (IDE) is an application for the development of source code [25]. The IDE contains many features to make development easier. In the same application, it is possible to compile the source code into executable machine code, as well as debug the code before compilation to detect errors.

One example of IDE is the so-called **PlatformIO**, an extension tool for visual studio code [26]. PlatformIO is a professional tool for embedded systems and software developers in order to develop cross-platforms, cross-architecture, and multiple frameworks. Thus a developer avoids the problematic setup procedures of including correct toolchains, and libraries and having proper IDE for a specific board. PlatformIO supports many frameworks, such as the popular Arduino IDE, furthermore, it supports many thousands of different development boards. PlatformIO automatically downloads and installs all required dependencies that are stated during the configuration of the project.

2.3.3.2 Micro controllers

A microcontroller is an integrated circuit containing a processor, GPIO, and memory to control a specific task in an embedded system [27]. Microcontrollers can also be referred to as embedded controllers or microcontroller units (MCU). The processor in a microcontroller is where arithmetic and logic operations take place and other data processing. The processor follows a set of instructions, that is programmed in the program memory. The program memory is non-volatile meaning that the program instructions are stored over time without a power source. Correspondingly the data memory is a temporary data storage for when the processor is executing the instructions. However, the data memory is volatile meaning that the information will be lost once the microcontroller reboots. The processor interface with other components through its GPIO pins, enabling the microcontroller to receive data and send instruction in binary. The GPIO peripherals can be connected to sensors, actuators, and other circuit modules to enable for example a communication protocol.

2.4 Data communication

A parable to explain data communication between different units is how we humans communicate. Both parties must speak the same language to be understood and carry on the conversation.

Similarly, computers communicate with each other through protocols instead of languages. Protocols are predetermined procedures and sets of rules to transmit data between electronic units or devices [28]. The sets of rules are needed for the devices to structure the data information, for either sending or receiving it. There is a considerable amount of well-established protocol standards for different areas of use, the relevant ones are presented below in the forthcoming chapters.

2.4.1 Serial communication

This type of communication protocol builds on the principle of sending one bit at a time[29]. The serial protocol can be classified according to 3 different transmission modes; Simplex, Half Duplex, and Full Duplex. **Simplex** communication is one-way meaning that if the transmitter transmits the receiver can only read where only one client is being active. However in **half Duplex**, both clients are active at not at the same time. For example, one client can send a request to another, which processes the request and then responds. **Full duplex** is simply when both clients of sender and receiver can transmit and receive at the same time.

As previously mentioned, serial communication is defined as sending one bit at a time while parallel communication sends batches of data, 8, 16 or 32 bits at a time[29]. One key difference is of course the speed to send data, where parallel is far quicker than serial. However, the overall efficiency in serial communication often outbalances the speed aspect. It requires far fewer wires and GPIO pins, making it cheaper and easier to implement. The Figure 2.7 and Figure 2.8, illustrate the difference between serial communication and parallel when sending a dataset of 4 bits.

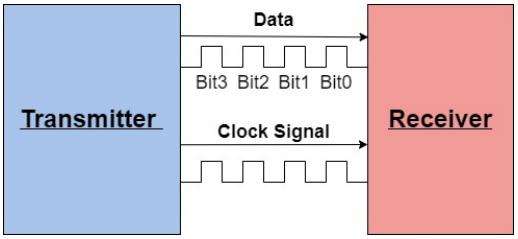


Figure 2.7: Serial Communication using 4 data bits and little Endian is illustrated. There is a transmitter (blue) and receiver (red) which sends data bits and a clock signal from the transmitter to the receiver.

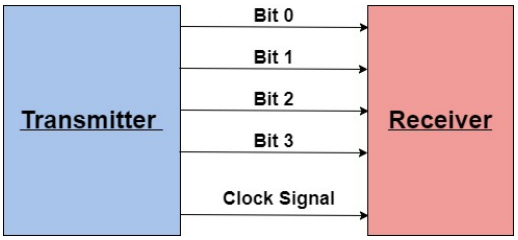


Figure 2.8: Parallell Communication using 4 data bits. In the image, a transmitter (blue) and receiver (red) are illustrated. From the transmitter, there are 4 data bits and a clock signal sent to the receiver in parallel.

I^2C

Inter-Intergrated Circuit (I^2C) is a very common communication protocol within embedded systems and is categorized as a half-duplex. More specifically I^2C can contain several or a single master node, communicating with several or single Slave

nodes [30]. The protocol builds on using two wires, Serial Data (SDA) and Serial Clock (SLC). SDA is the wire that sends the data bit by bit and SCL is the wire that sends the clock signal, synchronizing the communication between all units. The data is sent through messages, where each message contains an address frame for a specific slave that the master wants to communicate with. The I^2C message protocol also includes acknowledge/no-acknowledge bits, e.g. slave acknowledges that the address frame or date frame was successfully received and thereby returns an acknowledge bit to the master.

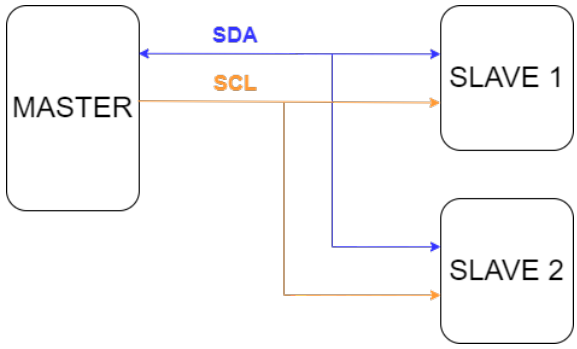


Figure 2.9: Illustration of the relationship between Master and Slave units in I^2C communication protocol. SDA and SCL are sent from the master unit to the slave units.

SPI

Serial Peripheral Interface (SPI) is another common protocol for communication between microcontrollers and sensors [31]. One major difference between SPI and the previously mentioned I^2C is that SPI master and slave can transfer data in full duplex. SPI also has the master-slave relationship, where the master node controls one or several slaves, as shown in Figure 2.10. Master Output/Slave Input (MOSI) is the wire that sends data to the slave and the corresponding Master Input/Slave Output (MISO) instead sends data to the master from the slave. Like most protocols, the SCL is used to synchronize the bit sending and reading. Chip Select (CS) wire selects which slave should receive the data from the master. However, if the number of GPIO pins is limited one CS wire can be used for all slaves, called daisy-chained.

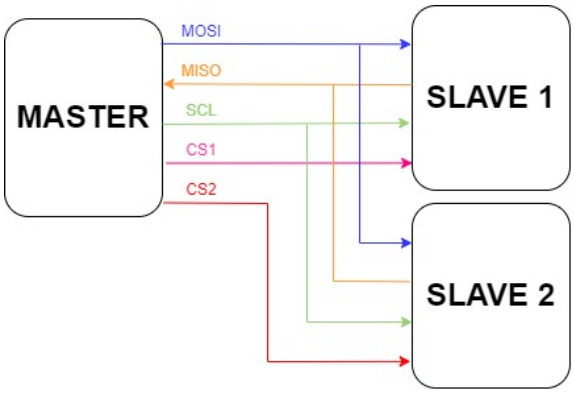


Figure 2.10: Illustration of the relationship between Master, Slaves in SPI communication protocol. MOSI, MISO and SCL are connected to every slave device. CS 1 and 2 determine which slave device is currently being communicated with.

UART

Universal Asynchronous Receiver/Transmitter (UART) is actually a physical circuit. The circuit's purpose is to transmit - and receive serial data [32]. UART can only communicate between two fixed units, with a set baud rate since the communication is asynchronous and without a clock signal. The dataflow is that the transmitter Tx of unit 1 is sending data to the receiver Rx of unit 2 to decode, and vice versa in the other direction. UART transmits its data into packets, where each packet can contain up to 9 data bits. Advantages of UART include features such as it only needs two wires and is a well-documented and established method. However, it is not possible to add multiple slave-master systems like other protocols. See Figure 2.11 for an illustration of how UART works.

Figure 2.11: Schematic illustration of UART communication. Each device has an RX and TX pin. RX on device 1 is connected to TX on device 2. TX on device 1 is connected to RX on device 2

2.4.2 Control area network

Control Area Network (CAN) is a serial communication bus protocol and is described by the standard: ISO-11898 [6]. CAN was developed by BOSCH in the 80s, to replace the complex wiring systems used in the automotive industry. Since the reason behind the development of CAN was the automotive industry, the technical characteristics include robustness and high reliability. CAN enable effective and high-performance communication between different ECUs without the need for a host computer.

CAN is a multi-master protocol consisting of several nodes, where the data is sent through only two wires CAN_H and CAN_L. The voltage differential between these two corresponds to the logic levels of bits being transmitted. To avoid collisions when several nodes want to send data, CAN uses two logical levels: dominant (logic 0) and recessive (logic 1) which is illustrated in Figure 2.12. The dominant will have priority over the recessive, as the dominant corresponds to a voltage difference whilst the recessive bit has the same voltage. Thereby the dominant will overwrite the recessive bit.

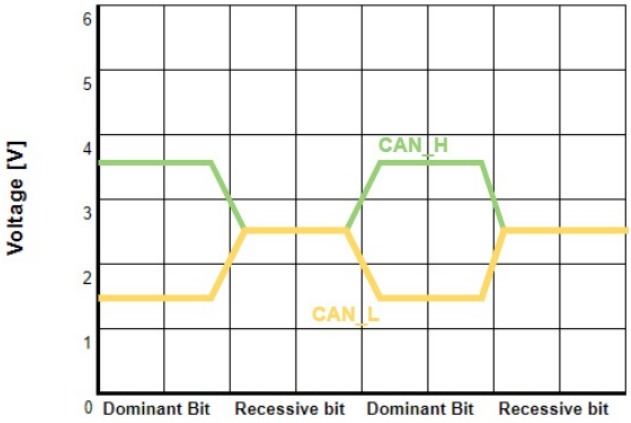


Figure 2.12: Illustration of the voltage differentiation in a CAN bus. Where a voltage differential between CAN high and CAN low corresponds to a dominant bit and no voltage differential corresponds to a recessive bit.

The data that is transmitted is packed into frames, that are built up by several fields. These frames describe for example the identifier of the frame, other mutual bit settings for the protocol, and the data frame. The data frame is where the information is stored, the frame supports up to 8 bytes of data. The baud rate is supported up to 1Mbit/s with a maximum length of 40 meters in the CAN_H and CAN_L wires.

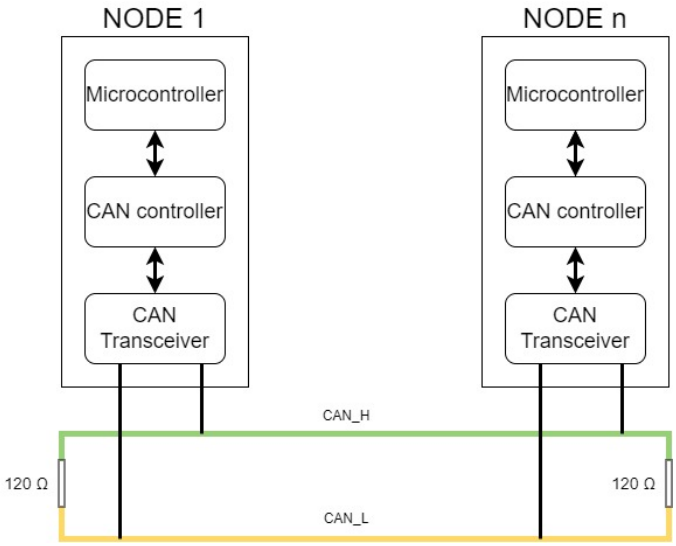


Figure 2.13: Example illustration of ECUs CAN communication between nodes. The nodes are connected together with a CAN high and CAN low cable. Each node has a CAN- controller and transceiver. The two lines are terminated by two 120 Ω resistors

In Figure 2.13 a typical automotive system consisting of several ECUs is illustrated. Each ECU's CAN node consists of two components, CAN controller and CAN transceiver [6]. The CAN transceiver transmits data from the CAN controller onto the CAN bus. Similarly, the transceiver also acts as a receiver and reads data from the CAN bus and converts it for the CAN controller. The CAN controller processes data both ways between the transceiver and microcontroller. The microcontroller is where the computing of the data being received or data that should be sent.

2.5 Digital twins

The concept of digital twins has been around for a long time. The term has come to mean different things within separate industries [33]. A proposed definition of digital is presented as follows; It exchanges data in both directions between the digital model and physical system. Closely related to digital twins, there are two important alternatives. Digital shadow and digital model. A digital shadow can only transfer information from the physical object to the digital object and a digital model can only mimic the behavior of a physical model.

The fidelity of the simulations with respect to real-life performance plays an important role in how well the digital twin can test and verify functionality [34]. Regardless if the digital twin is a good representation of the physical system, if the simulation environment cannot represent the physical environment where the product is to be used it will not produce any useful results, [35]. Therefore, a well-constructed digital twin, tested in a simulation environment that represents the environment where the

physical product is used is necessary if the simulation results can be used to test and verify functionality in a trustworthy and repeatable way.

There are several software available in which a developer can insert the digital twin into a simulated environment [36]. The most common ones are Gazebo, CoppeliaSim, MORSE, and Webots.

2.5.1 Benifits of digital twins

There are many applications within the automotive industry and the development of software where digital twins are used. An example where a digital twin is utilized within the automotive industry is to have a digital twin of vehicle-to-vehicle communications [37]. Vehicle-to-vehicle communication could have big potential in the vehicles of the future. The digital twin is a software replica of the physical 5G network that would be used to simulate sending information between vehicles. Digital twins can be used for many systems or subsystems in an autonomous vehicle, an example of a sub-system is a digital twin for the electric drive and propulsion. [38].

Digital twins are a useful tool to use when developing software and functionality for modern automotive applications [33]. They allow functionality to be tested in a very early stage of development in a development or design phase. In later stages of development, they can be used to verify functionality. When the functionality is mature and moved onto a physical device or platform a digital twin can be used to monitor the system behavior and verify new functionality in an easy way. The digital twin model needs to advance in capability over the lifecycle of a product to keep being representative of the physical product and its capabilities.

2.5.2 Unified robot description format

A digital twin, its appearance, kinematic behavior, and more can be defined using a standardized format in ROS [39]. A Unified Robot Description Format file (URDF) is used to describe the kinematics and dynamics of robotic systems in a standardized manner. The file is written using XML syntax and due to its standardization can be read and utilized in many existing software packages in ROS.

The URDF file describes the physical aspects of a robot, such as kinematic and dynamic behavior, and how it should be visualized using geometrical shapes or textures. A combination of basic shapes such as boxes, cylinders, and spheres can be used to visualize any type of right-body robot. The description of a robot is built using a chain of *links* and *joints*. Every robot starts with a *base link* and onto this link once can configure how the robot should behave. Common attributes to configure for a link are; position relative to another link, visualization, inertial and collision behavior [40]. Two links can then be connected to one another by a joint [41]. The most commonly used joint types are revolute, continuous, prismatic, and fixed. The joints can then be configured with dynamical properties.

2.5.3 Digital twin simulation environment - Gazebo

Gazebo is an open-source physics simulator first created in the mid-2000s [42], yet it is still relevant and very useful today [36]. Gazebo was released in 2004 as a 3D dynamic open-source physics simulation tool with the purpose of simulating multi-body and multi-robot environments [42]. The physics simulators of the time directed towards robotics were either closed source or lacked the functionality gazebo presented. This new physics simulator was made with the intent to accurately reproduce any environment the robot would likely be subjugated to in real-life scenarios. Every object within the simulation is simulated with mass, velocity, and friction to capture the dynamics of objects. Gazebo was built with the player device server compatibility in mind, meaning a robot client connected to the simulator would not be able to distinguish between the simulated sensory data feedback from real sensory data. A robot control system can integrate actuator commands and receive sensory data using external interfaces. At the time of release, Gazebo was not meant to be used for large multiple robot systems, but at most around ten robots at the same time [42].

Gazebo is still being developed on today due to its open-source nature, it has a big community around it developing plugins, creating robot models and adding functionality [36]. Today Gazebo has native support for ROS making it very ideal to use when developing robots and digital twins using the ROS framework. The core functionality of Gazebo in modern times, are plugins. A close concept to interfaces which was presented in the early 2000s. A plugin is in its most simple form, a piece of code compiled and inserted into the gazebo simulator to read or write data from the simulation.This allows developers to interface with Gazebo and control many aspects of the simulation. From changing the simulated world to controlling robots within the simulation. Some important groups of plugins are model, sensor, system and world plugins

2.5.4 Digital sensor twin

As described in subsection 2.5.3, with the use of plugins one can simulate and or interface with almost anything inside Gazebo, this enables sensor plugins to retrieve simulation information and turn it into sensory information. There are different plugins for different types of sensors. For each type of sensor, there is a set of parameters that can be tuned in order to reach high fidelity with the real-life sensor counterpart. The plugins can also return the sensor data with a data type identical to what would be seen when interfacing with a hardware sensor. When Gazebo was first released, it had interfaces to retrieve camera output, odometry and ray proximity sensor information [42]. Due to the open-source nature of the gazebo project, more sensor interface plugins have since been added [43]. With regard to the trends within the automotive industry and research into autonomous driving, the most important sensors are cameras, radars and LIDARs [14]. Gazebo can simulate these and many more sensors related to the dynamics of a vehicle. There are plugins for cameras, laser, Inertial Measuring Units and Ackermann vehicle control [44].

3

Analysis of Previous Generations

Infotiv initiated 2019 a project to develop a platform for autonomous systems in order to follow the trends within the automotive industry [5], see Figure 3.1, and thereby increase their knowledge and understanding of the problems that the industry is facing. A platform similar to an actual vehicle enables more R&D possibilities and proves scalability. The end goal of the project is to have a fully self-driving go-kart, the platform also allows the development and testing of other functionalities such as lane assistance, driver interface, and adaptive cruise control.

There have been several iterations and numerous theses working on the project resulting in three previous generations. This chapter will process the specifications of previous generations and problems with them. Furthermore, an analysis of the given system on which the AP will be implemented is presented.

Figure 3.1: Image of Autonomous Platform third generation (AP3). It shows the current hardware, regarding sensors and computing units. Reprinted, with permission, from Infotiv AB

3.1 Previous generations

There exist three previous generations of the Autonomous Platform [45]. The first generation is based on a smaller platform inspired by the sizes of RC car toys. The system architecture is decentralized, with several different domains or as they are called within the project, modules. The modules are connected through three different CAN networks and a central electronics module gateway (CEM). The second generation platform is an iteration of the first one with a new code base and documentation approach but without any new implementations. The third and latest generation is far bigger in size and is based on a go-kart system, this platform is the one the new generation will be based upon and will be analyzed in more detail in the next section. The third generation has also a decentralized system architecture and is mainly a scaled-up version of the second platform.

3.2 Ninebot S and go-kart kit

Ninebot S is a Self-balancing scooter and with the integration of the Ninebot Electric go-kart conversion kit can be turned into a light-weight electric go-kart. This complete vehicle system is the foundation for AP3 and will be the same platform for AP4.

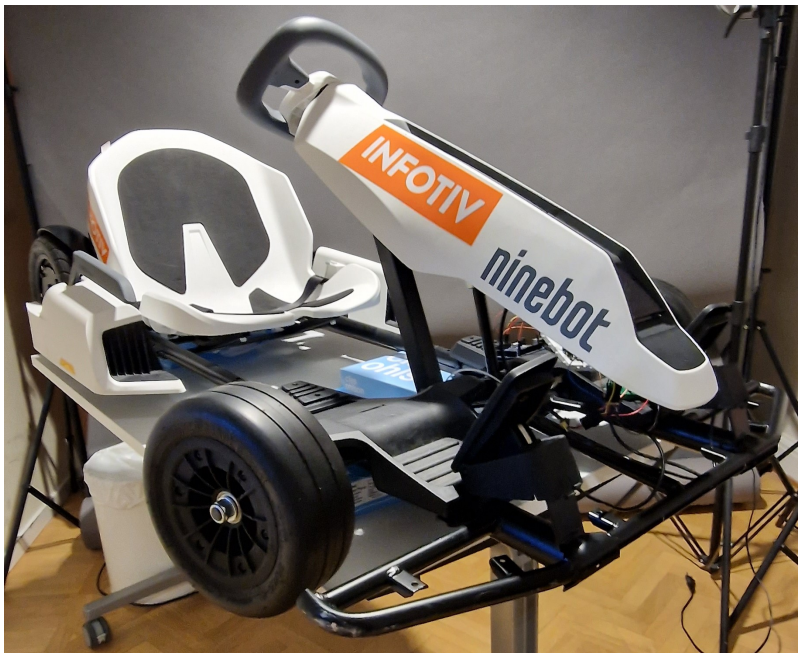


Figure 3.2: Ninebot S and Go-kart kit stock configuration Image of go-kart kit before AP4 implementation.

3.2.1 System description

Ninebot S

The Ninebot S has two electric motors inside the hoverboard wheels to drive it forward. It powers on through a power button on the back of the board, where informative LED lights give the user feedback on settings and states. Furthermore, Ninebot S can be connected through Bluetooth to the Phone and control the segway with the so-called Segway-Ninebot S app. This app can control speed limits and give users informative and illustrative feedback on speed, state of charge in the battery, etc. In the table below, key technical specifications are stated.

Table 3.1: Specification of Ninebot S self-balancing segway [46].

Category	Item	Parameters	Units
Dimensions	Length	260	mm
	Width	548	mm
	Height	595	mm
Veichle	Max Speed	16	km/h
	Max Slope	15	degree
	Operating Temperature	-10 <=>40	degree
Battery Pack	Rated Voltage, DC	54.8	V
	Maximum Charge Voltage	59.5	V
	Rated Capacity	236	Wh
	Max discharge power	1000	W
Motor	Rated power	400x2	W
	Maximum power	800x2	W

Go-kart conversion kit

To implement the go-kart kit the Segway Ninebot S is needed. The go-kart kit has a separate energy storage to power its electronic components such as a circuit board, pedal sensors, and headlights. When connecting the go-kart kit with the Ninebot S through the power outlet of the Segway, Ninebot S will enter its go-kart mode. Thus it will not act as a self-balancing Segway, instead, it will get propulsion commands from the circuit board of the go-kart kit through a serial communication wire. Key technical specifications of the Gokart kit implemented and the segway are stated in Table 3.2.

Table 3.2: Specification of Go kart conversion kit. Relevant specifications for the gokart conversion kit are placed in a table [47].

Category	Item	Parameters	Units
Dimensions	Length	1383	mm
	Width	600	mm
	Height	822	mm
Veichle	Max Speed	24	km/h
	Reversing speed limit	3	km/h
	Range	15	km
	Steering Ratio	2.1:1	-
	Max Slope	15	degree
	Operating Temperature	-10 <=>40	degree
Brake	Brake method	eletronic and mechanical brake	
	Breaking distance	6	m

3.2.2 Analysis of functionalities

Communication between segway and go-kart

The communication between the segway and go-kart is composed of a UART serial communication. When developing the third generation, Infotiv conducted some analysis of the serial communication [48]. The analysis was based on an open-source application called 9BMetrics. However, the protocol in the application is used for another product, an electric scooter from Ninebot. Thus the communication protocol could not be entirely deciphered.

In the scope of this thesis, an analysis of the UART communication between the segway and go-kart was made. In order to replicate the results from previous analyses and investigate if other conclusions could be made. Without any documentation of the method from Infotiv's analysis, an entirely new approach was needed in order to sniff the UART data. Using a microcontroller such as the Arduino Mega and connecting it to either Tx or Rx, it was possible to log the data. However, the results from the previous analysis could not be recreated or verified. Without the correct protocol and information about baud rates, it would be nearly impossible to decode all data messages.

Propulsion

The propulsion of the go-kart is controlled through two pedals, throttle and brake. The pedals were disassembled in order to investigate how the sensor is constructed to read the pedal position. Under each pedal, there is a magnet and a hall sensor. The hall sensor is connected with three wires, ground, supply voltage, and position signal. The hall sensor can detect the magnet's field and its relative position. When the pedals are pressed down, the magnet's position will change and the position signal will increase. The go-kart board will read the signals and convert these into instructions for the segway that are transmitted through the UART wires. The complete schematic of the go-kart communication and its key components is illustrated in figure 3.3.

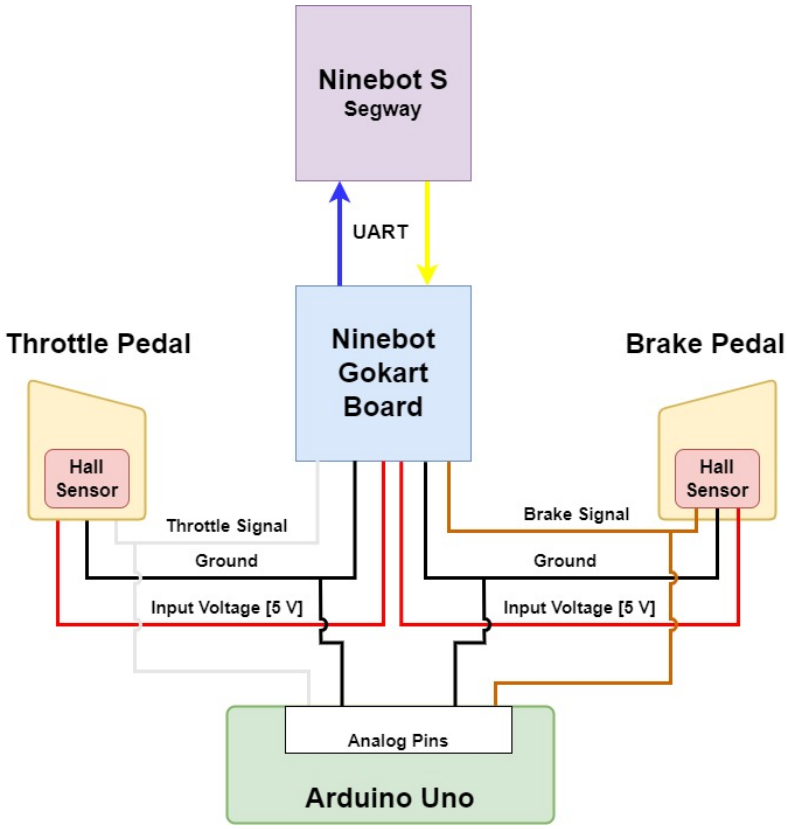


Figure 3.3: Overview of the Go-kart key components in a schematic view and the schematic of voltage measuring setup. An Arduino UNO is connected to the brake and throttle voltage out wires.

In Figure 3.3 it is illustrated how measuring the voltage magnitude of the throttle- respectively the brake signal. By connecting the throttle and brake signal in parallel to a microcontroller's analog input it is possible to conduct a measuring session, the circuit diagram is illustrated in figure 3.3. An Arduino Uno was used since it met the requirements of measuring analog signals with sufficient accuracy and sampling rate. Since the purpose of the measuring was to determine the limits of voltage magnitude in the pedals, requirements for accuracy and the sampling rate were very low and Arduino Uno had enough hardware. Despite there exist complex heavy-duty equipment such as oscilloscopes and multi-meters. Another advantage of using a microcontroller is the possibility to log the values over time and plot the values in real-time. The results are summarized in the table below.

Table 3.3: Voltage magnitude for throttle and brake pedals. Also, the time delay between pressing the brake pedals two times in order to enter reverse mode.

Signal	**Magnitude**		**Unit**
	Max	**Min**	
Throttle	4.35	0.82	V
Brake	4.33	0.84	V
Enable Reverse	200	50	ms

3.2.3 System overview and hardware

As previously mentioned the third generation is based on a decentralized architecture with CEM routing three different CAN busses. Where each of the three CAN busses corresponds to a domain specific CAN network, and the CEM acts a central gateway of the backbone.

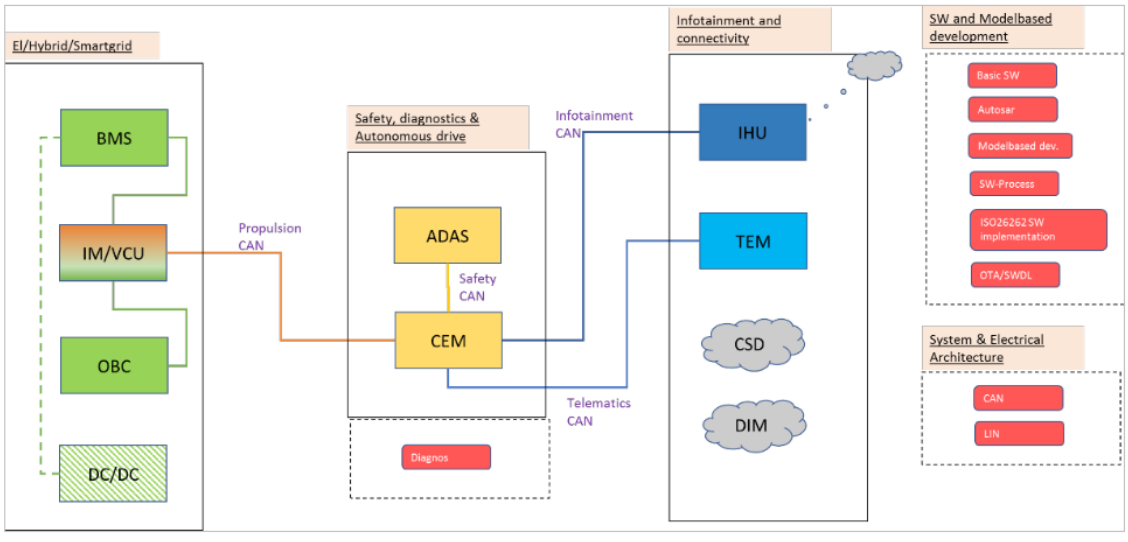
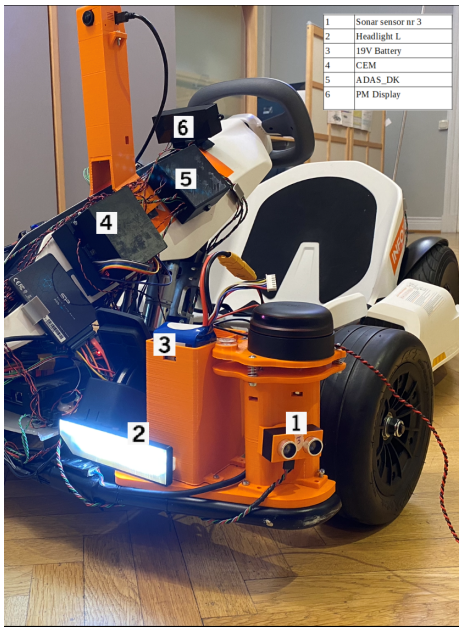


Figure 3.4: System overview Autonomous Platform generation 3. The system overview illustrates how different components are connected and how functionality is split up. Safety, diagnostics & autonomous drive are separate in HW & SW from power management. Each block is an ECU with specific functionality. Reproduced, with permission, from Infotiv AB



(a) Description of hardware components in the front of AP3

(b) Description of hardware components in the front of AP3

Figure 3.5: Hardware descriptions of AP3. Illustrating sensors and functionality. Mounted sensors and actuators are illustrated and described in the images. Reproduced, with permission, from Infotiv AB

Figure 3.5, describes some of the sensors and other hardware and where it is implemented. Furthermore, each ECU consists of a wide range of different microcontrollers and boards. Often it is based on STM32F103C8T6, also known as Bluepill, but also Arduino nano and the heavy-duty NVIDIA Jetson TX2 Developer Kit.

3.2.4 Driveline and steering

The driveline is based on the segway Ninebot S and is controlled using the accelerator- and brake pedals. The voltage signals read from the pedals are low-pass filtered before being sent to the drive motor. The steering consists of a DC motor EMG49 with a built-in encoder [45]. The motor is connected to a tooth pulley system to gear up the torque and actuates upon the steering shaft of the go-kart kit. Furthermore, the motor is controlled through a motor drive board called SABERTOOTH DUAL 2X25A/6-24V MOTOR DRIVER. The motor drive board is then interfaced with a motor controller called Sabertooth Kangaroo x2 which receives commands from a microcontroller through a serial port. The turn ratio is determined by a calibration of Sabertooth Kangaroo x2 and a set of limit switches.

3.2.5 Bugs and issues

There are several issues with previous generations, actually, so much it is the reason behind this thesis [49]. The platform is out-of-date as it is based on a decentralized architecture, and the trend in the automotive industry is toward a centralized architecture. The documentation is lacking and component descriptions are missing. Wires and microcontrollers are not labeled, and most implementations are quick fixes. Overall the complexity of the system makes it difficult to work on.

4

System Requirements, Specification and Verification

In order to design a system there needs to be a clear set of requirements to define the system's performance and capabilities. This will make the designing and construction easier, knowing the system's limits and what stakeholders want. Thus the development makes it easier to prioritize the resources and to acknowledge when a specific subsystem fulfills the wanted functionalities. The end result can also be analyzed through a set of verification and validation criteria to fulfill the stakeholder's wishes.

Requirement defines the performance or capabilities of the system [50]. When stating requirements it should answer three elements, what, when, and how. Having boundaries for writing requirements is beneficial in order to have measurable conditions. Thus making it possible to verify each requirement through *specifications*. A *specification* is a measurable quantity that states when a requirement is fulfilled. Another key point of stating requirements is the use of appropriate level, thus for the scope of this thesis in developing a new platform.

Verification is the method chosen to confirm that the system meets the requirements, in a preferably measurable way [50]. The verification results are compared to the *specifications* of the requirements.

The following subsections describe the methods of developing the requirements, specifications, verification, and validation for the platform. Furthermore, the system's performance and capabilities from stakeholders are described in the requirement list.

4.1 Requirement and specification list

The requirements and specifications of the system for the platform are influenced by stakeholders within Infotiv's autonomous platform project, section 3. The list of requirements and specifications follows a determined structure, the template is illustrated in table 4.1. The structure addresses each desired function of the system and then states the requirements related to the same domain. The minimum

requirement is strict and shall be fulfilled, however, the desirable requirements are non-mandatory. Furthermore, the list also weighs each function in order to prioritize resources and address the most important requirements first. The complete list of requirements can be found in subsection 4.1.1.

Table 4.1: Requirement and specification table structure. The table aims to describe how the requirement and specification list are constructed.

Requirement nr	**Function**	**Minimum requirement**	**Desirable requirement**	**Priority**	**Minimum specificaions**	**Desirable specifications**
	X	- Sub requirement 1 - Sub requirement 2	- Sub requirement 1	[10 HIGH - 1 LOW]		

4.1.1 Requirements of AP4

Requirement nr	Function	Minimum requirements	Desireble requirements
1	BMS and Power Management System	**Shall have:** - Switch between battery and PSU - Supply 12 V - Enough energy storage capacity to operate for longer periods	**Inclusive the minimum requirement, it shall also have**: - Internal fuses to protect battery and circuits on the platform - Measure voltage and amperage - Simple user inerface and be easily operated - Killswitch - function to check SoC
2	Steering control	**Shall have:** - Set steering angles for max left and right	**Inclusive the minimum requirement, it shall also have:** - easy calibration method for steering angles - measuring the angle - set steering accurate according to a given angle
3	Propulsion control	**Shall have:** - From a command drive forward, accelerate, brake - Possibility to activate reverse mode - Simple Proportional Controller	**Inclusive the minimum requirement, it shall also have:** - Fast Feedback controller with a speed sensor
4	Computing unit	**Shall have:** - Capability to run a operating system - Capability to run highlevel programs such as python - GPIO pins - USB port - Enough memory - Ethernet - Possibility to connect monitor, keyboards and mouse	**Inclusive the minimum requirement, it should also have:** - Possibility to replace and uppgrade computer HW
5	E/E Architecture and modularity	**Shall have:** - Centralized E/E architecture - Possibility to integrate new ECUs components - Follow automotive and other standards - Possibility to change the physical layout of HW	**Including minimal requirements, it shall also have:** - Easy integration of new ECUs - Highly scalable and modular functionalities - Intuitive Software design for easier uppgrading/replacing code
6	Transportation Rig / Lift aid		**It shall:** - Simple to use and ease/prevent of back-pain - Fit into car with back seats folded down - Lift the gokart of the ground - Be able to pull the gokart with the rig
7	Usability / Unit Tests	**Shall have:** - Simple startup/init procedure - Internal tests - Give user feedback when enecountered errors - Easy to modify SW	
8	(SW) Digital Twin	**Shall have:** - Central computing units SW enables the use of a digital twin	**Including the minimum requirements, it shall also have:** - A accurate digital twin of the AP4 implemented -A digital twin to test high level functionalities -Simplified 3D model and visualization of AP4 and/or its sensor readings - Should be modular and scalable
9	(SW) Modular Software for Control	**Shall have:** - Capability to use external software tools/libraries to modify and control the AP4	
10	Safety	**Shall have:** - All circuits should be dimensioned accordingly to the specifications of each circuit board. Furthermore, a sufficient wire diameter shall be used according to the current flowing through. - Battery should be placed and used in a safe way by users - Battery should have a kill switch connected to emergency stop / manual switch - Go Kart shall have a designated fire extinguisher for electronics purpose - Clear schematics of current carrying cables - No open terminals	
11	Sensors in AP4 (HW and SW)	**Shall have:** - Possibility to easily add new sensors - Should fit on AP4	
12	Budget	**Shall have:** - Meet the budget requirement from Infotiv	

4.1.2 Specification of AP4

Requirement nr	Function	Minimum specification	Desireble specification	Verification Method
1	BMS and Power Management System	**Shall meet:** -Supplies 12 V +-0.5 V to system continously - voltage does not drop more than 0.5 V during load - Autonomous platform should be able to run for one hour continously without being connected to a wall outlet	**Including minimum specification shall also meet:** - 10 A of current continously for 1 hour - 20 A of current during peaks < 5 seconds	**Minimum** - Measure the voltages in each output using multimeter - Timing of test run **Desireble**: - For switches/Charging measure the current and voltage so that it behaves expected. i.e when current should flow or not in each circuit
2	Steering control	**Shall meet:** - Possibility to manually calibrate steering angles - within a 20% margin meet a given steering angle	**Inclusive the minimum specification, it shall also meet:** - Calibration at start up - measure the angle within a 10% margin	- Manually measure with a protractor - Optically see how the steering limits changes after manual calibration - Optically see how the calibration algroritms returns the limits
3	Propulsion control	**Shall meet:** - In reversemode, internal gokart system detects and starts beeping - When throttle is increased the relative velocity is positive - When braking is increased the relative velocity goes to 0 - At full brake, the braking distance should be lower than 6m - The delay between the HWI communication and the go-kart segway shall be less than 100 ms.	**Inclusive the minimum specification, it should also meet:** - time constant of less than 1 sec - error of maximum 10% after 3 sec	**Minimum:** - Optically see how the gokart moves according to commands given - manually measure the braking distance - time the proportional controller **Desireble:** - perform a test run with unit step input - Measure and log the speed with time stamps - Log reference values with time stamps - Compare, visualize and calculate the key values
4	Computing unit	**Shall meet:** - requirement list for running ubuntu - 10 GPIO pins - 1 usb port - 1 hdmi port - 120 GB hardrive - standard ethernet port	**Including minimum specification, shall also meet:** - multiple usb 3.0 ports - ethernet 100 Mbit/s - 30 GPIO pins - Can run multithread	- Checking technical specification of possible computing units
5	E/E Architecture and modularity	**Shall meet:** - one central comuting unit for highlevel control - one hardware interfacing ECU - Follows mostly the standards - Response time ECU <-> Central, less than 500 ms - ECU update frequency should not be lower than 20 Hz - while integrating new ECUs, older ones should still be compatible without needing updates (SW and HW)	**Including the minimum specification, it should also meet:** - ECU update frequency should not be lower than 100 Hz - Multiple ECUs integrated - Response time ECU <-> 10 ms - Guides and instructions on how to implement future functionalities	- Use timers in programs - Standards
6	Transportation Rig / Lift aid		**Shall meet:** - The physical limits of the gokart	- Test equipment on gokart and see possible tensions/ fracture points
7	Usability / Unit Tests	**Shall meet:** - Automatic startup when powers on - When error occured, give user information of where and why it occured. In each ECU and central computer - Documentation on how each ECUs HW/SW is constructed and how to modify		- Automatic scripts and prints error
8	(SW) Digital Twin	**Shall meet:** - Use of SW development tools that can include simulation of physical dynamics and controls	**Including the minimum specification, it shall also meet:** - should include all sensors and actuators that is used - sensors measuring variation should not differ more than 10% from the actual HW - Digital Twin having same control highlevel algorithms	- Log data and sensor input from HW - Use sensor inputs in digital twin - Compare the differences
9	(SW) Modular Software for Control	**Shall meet:** - Is the SW control designed in such a way that velocity and steering can be set without any updates on the low level SW and communication tools. - Possible to access the state of AP4 and its readings from sensors		- Create a very simple high level control, only affecting the mentioned high level SW
10	Safety	-	-	
11	Sensors in AP4 (HW and SW)	**Shall meet:** - A list of the most common used sensors in AD and ADAS - Meet the reasonable physical/ scale limitation of AP4		- Litterature study investigation of what sensors are most commonly used in the automotive industry
12	Budget	**Shall meet:** - total cost up to 30 000 sek		- Budgethandling in excel

5

Hardware Platform Design

This chapter aims to describe how the autonomous platform system is designed and constructed hardware-wise in order to satisfy the requirement specification list. The chapter is structured to be top-down, meaning that in section 5.1 consists of an overall system description. section 5.2describes the functions and features of the general ECU base, which enables a modular base and choices regarding standard outlets. Further, a detailed illustration and description of each component used in the implemented ECUs and in the Central Master Computer is in section 5.3. In section 5.4 it is described how the modular design was accomplished and what standards have been brought into AP4. Lastly, section 5.5 touches upon the power supply for AP4.

5.1 System overview

The developed centralized E/E architecture of the autonomous platform is illustrated in figure 5.1, influenced by the requirements in section 4.1.1 and the generic description of a centralized E/E architecture, figure 2.2. Down below follows a description of each component of the centralized E/E architecture. Each ECU follows a generic structure described in 5.2. The Software of each component is described in chapter 6.

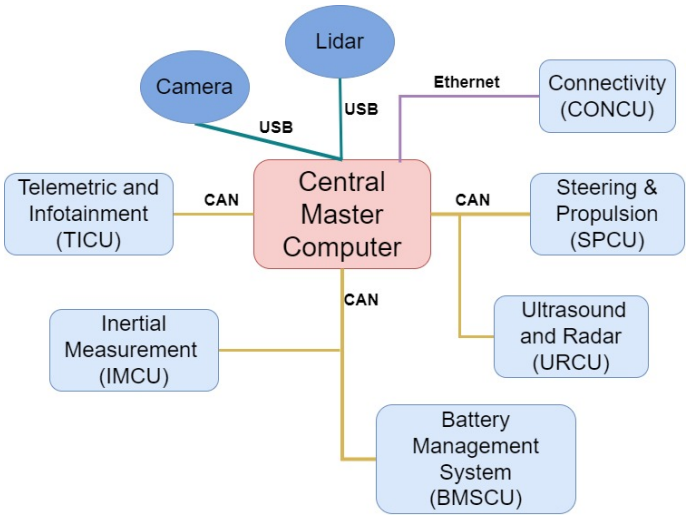


Figure 5.1: E/E Architecture of Autonomous Platform 4th generation. This schematic view illustrates what components should be integrated into the system and how they should be connected. As of this thesis, only the SPCU, Camera, Connectivity and battery management system are implemented.

The central master computer contains the high-level computation algorithms. The computer supports a direct interface of GPIO pins in order to communicate with the other ECUs through CAN. Furthermore, it also consists of USB and ethernet ports for dataflows requiring higher baud rates than CAN can support. More details of the chosen hardware components can be found in section 5.3.1.

The Steering and Propulsion Unit (SPCU) controls the steering and propulsion of the go-kart. This includes a DC motor with encoding and a motor driver module for the steering and two digital-to-analog converter (DAC) circuit boards for controlling the propulsion. More details of the chosen hardware components can be found in section 5.3.2.

The Ultrasonic and Radar Unit (URCU) contains sensors such as ultrasound sensors and radar, to enable localization and higher levels of AD algorithms. The selection of sensors is based on commonly used sensors in the automotive industry, which is further described in section 2.2.2. This ECU is **not implemented** due to time restrictions.

The Battery Management System Unit (BMS) controls the power management, this includes current and voltage readings of the battery module and actuators for switches. This enables the possibility to implement power optimization and state-of-charge estimations. This ECU is **not implemented** due to the time restrictions. However, the electric circuits and descriptions of the power module can be found in section 5.5.

The Inertial Measurement Unit (IMCU) contains sensors such as an accelerometer and a gyroscope, thus enabling the possibility of developing filters for estimations of

the platform's orientation and movement. Therefore, it is a necessary functionality that enables higher functionalities in an autonomous vehicle, such as AD and ADAS. State estimation of the vehicle is a crucial part of autonomous vehicles, it enables the vehicle to gain information about its location and orientation. This ECU is **not implemented** due to time restrictions.

The Telemetric and Infotainment Unit (TICU) includes user interfaces such as touchpads, monitors, and buttons. This ECU is **not implemented** due to time restrictions.

Camera and LiDAR sensors are essential when developing AD algorithms. These sensors are directly interfacing with the central master computer unit since the data feed from these sensors will overload the CAN bus (1 Mbit/s). USB 3.0 can easily support up to 5 Gbit/s, thereby ensuring video feeds and point clouds. A front-facing view camera **Logitech c920** is directly connected to the central master computer. However, a LiDAR is **not** yet implemented.

The Connectivity Unit (CONCU) will ensure wireless interfacing with the AP4, enabling over-the-air updates and remote control. It also allows for technology using cloud computing. The chosen methodology to ensure connectivity is the use of wifi.

5.2 Generic ECU base

Having a generic ECU base, see figure 5.2, for all nodes of the centralized E/E architecture will make the development of new ECUs simpler. Because the only new implementation needed will be the actual sensor module, whilst the base fulfills the requirements for power supply and communication. The base will provide power supply in the most common voltage ranges, and interface with sensors and actuators through GPIO pins. Furthermore, each base will be provided with a CAN transceiver and controller. Serial communication is used to flash the processor with new instructions. The serial port can also be used for debugging.

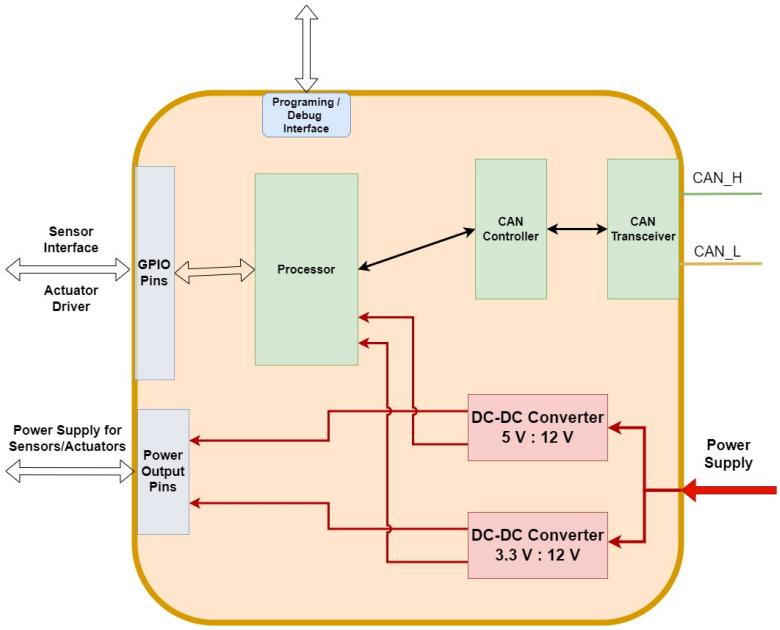


Figure 5.2: Illustration of the features and components inside each generic ECU base. Communication is handled with a CAN controller and receiver, Power is supplied by two DC-DC converters and a processor which handles computations and can interface with hardware using GPIO pins.

The chosen hardware is described below for each functionality in the generic ECU base in figure 5.2.

- **Processor, debug interface and GPIO pins:** Having a microcontroller will enable all wanted features. The chosen microcontroller is the STM32-F103C8T6 also called *Bluepill*. The microcontroller is cheap, small in size, and high-performing, having a clock speed of 72 MHz. It supports the most common communication protocols through its 37 GPIO pins. However, it has a relatively limited memory space with 128 KB of flash memory and 20 KB SRAM. The Bluepill is supported in Platformio and the use of Arduino framework, making the development of software easier. There are multiple other possible microcontrollers that could be used, such as the popular Arduino boards, but the selection of *Bluepills* is based on the better specification and that the previous generation used the same board.

- **DC-DC Converter:** The chosen converter is LM2596, which supports several voltage ranges through a simple turn of a screw, which makes it very user-friendly. The input voltage can vary between 4.5 V to 35 V and the corresponding output voltage can vary between 1.2 V to 40 V. Furthermore, it can supply up to 2 Amperes, thus it is far enough to supply the ECU and corresponding modules. Moreover, there is also a fuse in the power supply port to ensure that the ECU is not overloaded.

- **CAN controller and transceiver:** The MCP2515 CAN-bus module and

transceiver TJA1050 correspond to the controller and transceiver that can interact with the microcontroller through SPI communication. The selection criteria include usability since there exist many different libraries for the module. However, the Bluepill's logic level is 3.3 V whilst the can-buss module's SPI communication is at 5 V, a logic level converter is needed (BOB-12009 circuit).

- **Cooling fan:** In each generic ECU base, there is also a 12 V PC fan installed, 40x40 mm. It enables cooling of the components in order to lower the possibility of thermal damage and overheating.

5.3 Implemented ECUs

The physically implemented ECUs are described in the following sections. This includes the selection of sensors and actuators, and how they interface with the generic ECU base, described in section 5.2.

5.3.1 Central master computer & connectivity

The central master computer consists of two units, the first being the hardware interfacing (HWI) computing unit and the other called the general purpose high-performance computing unit (HPC), the central master computer is visualized in Figure 5.3. The reason behind having two units is that most common off-the-shelf PCs and laptops do not support a GPIO interface. The HWI is a Raspberry Pi 4b that is used to interface with the ECUs through CAN and additional sensors directly through USB ports. The CAN communication is enabled through the RS485 CAN HAT that is developed for Raspberry Pis. The HPC unit can vary quite a lot and has not been decided on strictly, the only limitations being that the unit supports Linux Ubuntu and docker. The HPC can be a simple laptop or any other high performing computer, for example an Intel NUC PC.

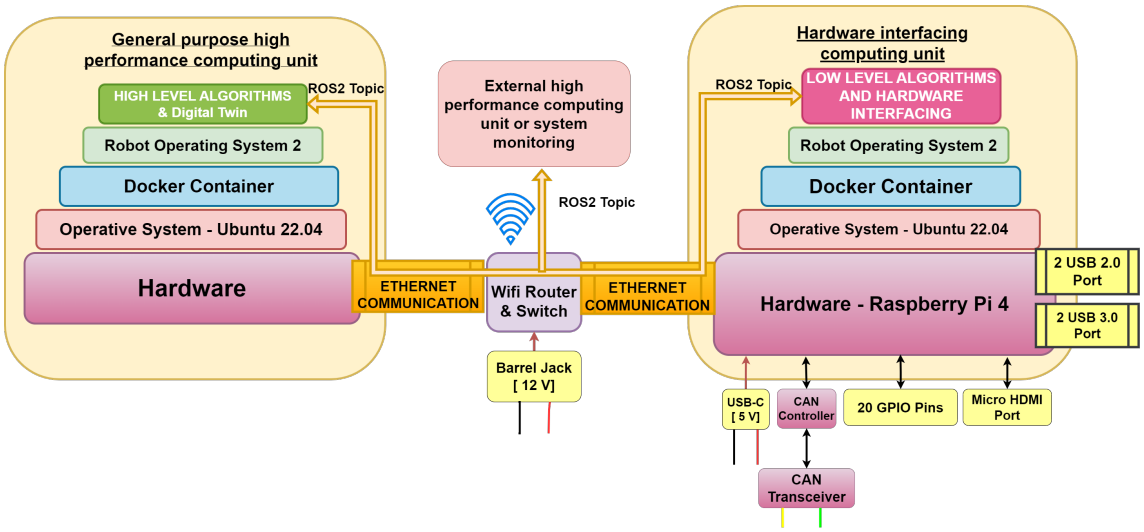


Figure 5.3: Illustration of how the internal interface of the central master computer and the connectivity capability. The physical communication between high-level software and low-level software is Ethernet. The low-level software is connected to the rest of the system using a CAN controller and transceiver. USB hardware can be accessed by physical ports on the low-level hardware.

The interface between the two units is over a router. The router supports wireless connection over wifi, which enables remote connection to AP4. The remote connection could also be used for monitoring or even cloud computing.

5.3.2 SPCU

The SPCU is based on the generic ECU base described in section 5.2. The SPCU consists of two modules, the steering module, and the propulsion module. The SPCU is shown in Figure 5.4.

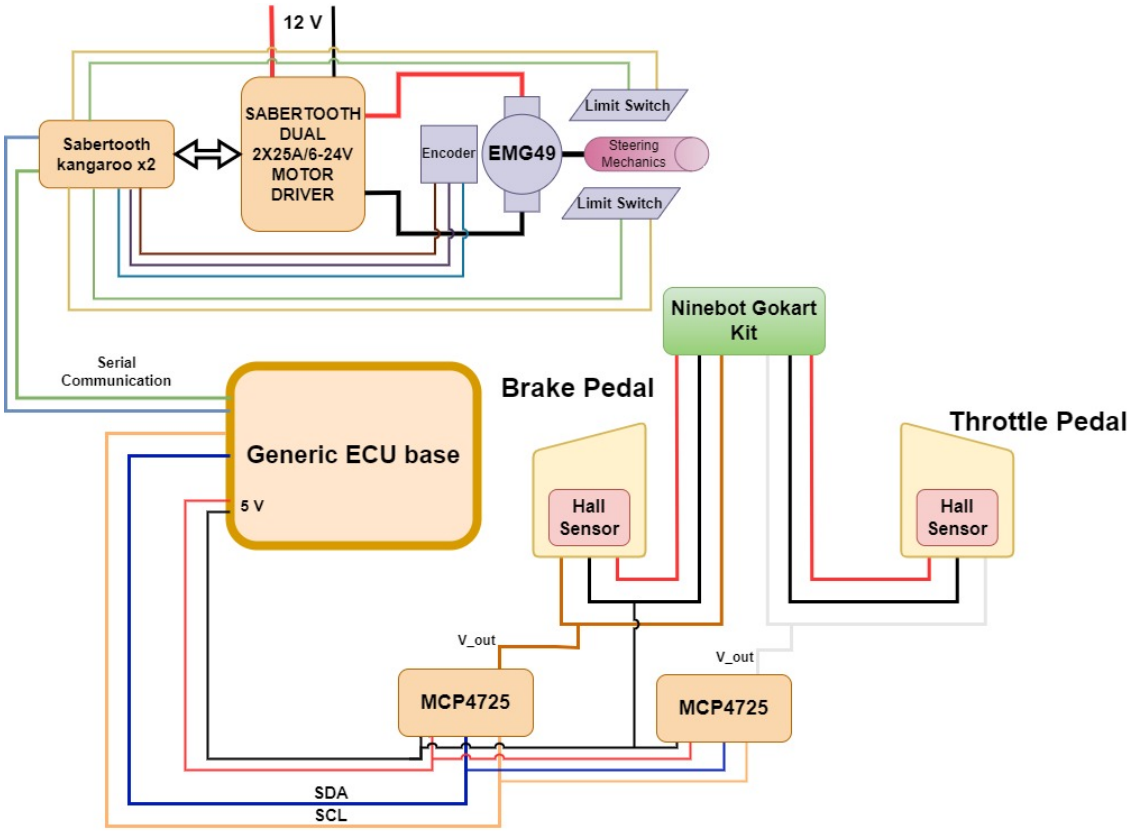


Figure 5.4: Circuit diagram for the Speed and Propulsion ECU and how components are wired. The generic ECU interfaces with the steering using Serial Communication. To control the propulsion, the ECU base is connected to two MCP4725 boards which in turn send analog voltages to the throttle and brake pedals.

The steering mechanics is the same as in the Generation 3 version of the platform. The steering mechanics is controlled by a motor drive module *Sabeerthooth dual 2x25A/6-24V motor driver* and motor control module *Sabertooth Kangaroo x2*. The steering can be controlled using serial communication.

Based on the analysis of the go-kart pedal functionality, described in subsection 3.2.2. The pedals will output an analog voltage that is linearly proportional to the position. A voltage source could thereby act as a pedal by sending out an analog voltage signal with the same characteristics as the real pedal. An analog signal can be generated by a MCP4725 module that is connected to a microcontroller. By wiring the pedal and the MCP4725 in parallell it enables them to act independently of eachother. Meaning that go-kart pedals will still be usable when the microcontroller is connected. This for example, allows an operator riding in the go-kart to brake manually if necessary.

5.4 Modular hardware design

To enable modularity and flexibility of hardware placement, a modular design based on an aluminum sheet with a rectangular pattern with holes, as illustrated in Figure 5.5, is used. The dimension of the aluminum sheet is 500mm long, width of 250 mm, and a thickness of 1.5 mm. The hole pattern has holes of 4 mm in diameter and is evenly distributed in a rectangular pattern of 15 mm between each hole. This hole pattern is set as a standard for the complete AP4.

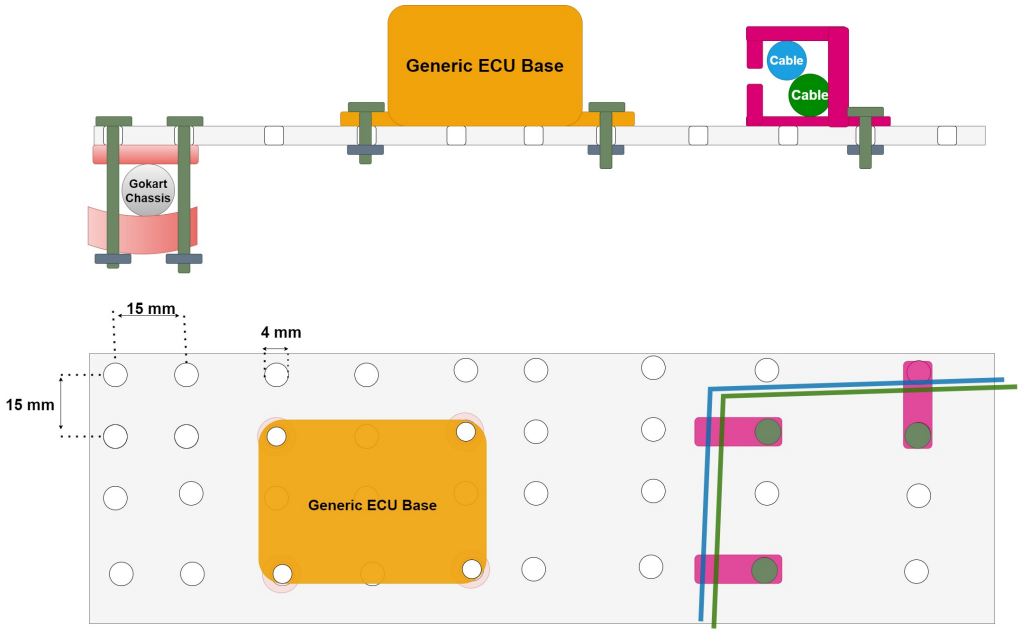
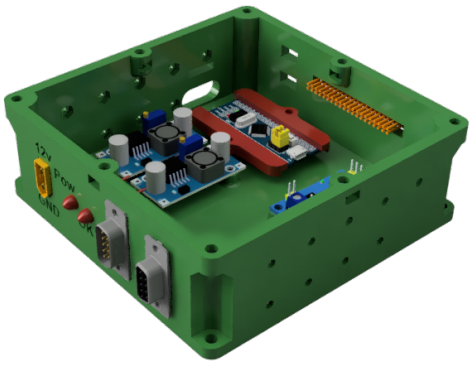
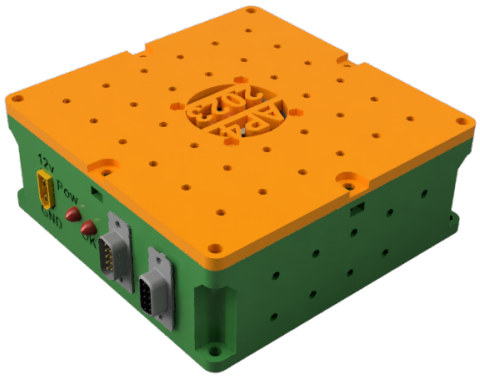


Figure 5.5: Visualisation of modularity on AP4. Illustrating the aluminum plate and how modular designed casings could be mounted in different configurations enabling a flexible physical layout of the system.

The ECU casing is designed to align with the aluminum plate's hole pattern making it possible to mount the casings arbitrarily on the plate, see Figure 5.5. Furthermore, each casing for all hardware also has the same pattern on the sides and on top, in order to increase the flexibility placement of other modules as well. Moreover, the use of established standardized outlets such as xt60 for power supply, makes the complete system more standardized and it also increases flexibility. For the CAN protocol, a DB9 cable female-female is used and all CAN transceivers are connected through male outlets. The generic ECU casing can be seen in Figure 5.6. The yellow outlet is the xt60 and the grey outlets are DB9 outlets. Furthermore, there is an opening in the back panel to enable the use of GPIO pins on the Bluepill.



(a) Generic ECU casing without top, showing the physical layout of circuit boards and components. The circuit boards functionality and actual hardware are described in section 5.2



(b) Generic ECU casing with the top plate mounted. There is a hole in the top where a fan is mounted.

Figure 5.6: Rendered illustration of the generic ECU. Highlighting how components are mounted within and how it interfaces with the rest of the system.

The other casings for example the central master computer, front wings, and mounting plate holders have been designed in the same way. That is, to utilize the standardized hole pattern and the use of standardized outlets and cables. Rendered figures of other casings and mounting fixtures can be found in Appendix B.

5.5 Power module

5.5.1 Power consumption

The power consumption of the implemented AP4's hardware has been calculated in Table 5.1. The values have been derived from each component's product manual. The values are based on the nominal values in order to define overall power consumption. Some components have been neglected such as the losses in cables, wires, and switches. The power calculation follows the well-known electrical direct current power formula:

$$P = U \cdot I \qquad [W] \tag{5.1}$$

where U is the voltage and I is the current.

Table 5.1: Power Consumption of AP4 summarized. The voltage and current consumption of the major components on AP4 are noted down and converted into a power consumption value.

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

5.5.2 Components

The power module consists of three components, a battery, a battery charger, and a power supply unit (PSU). There is a user interface including a power switch, emergency killswitch, and outlets that select which power source should be used for the autonomous platform. Also, there are voltage- and current measurement units mounted, but they are not implemented in software or physically connected to an ECU. The complete electrical circuit of the power module can be found in Appendix A and a rendered illustration of the CAD files for the power module can be seen in Figure 5.7.

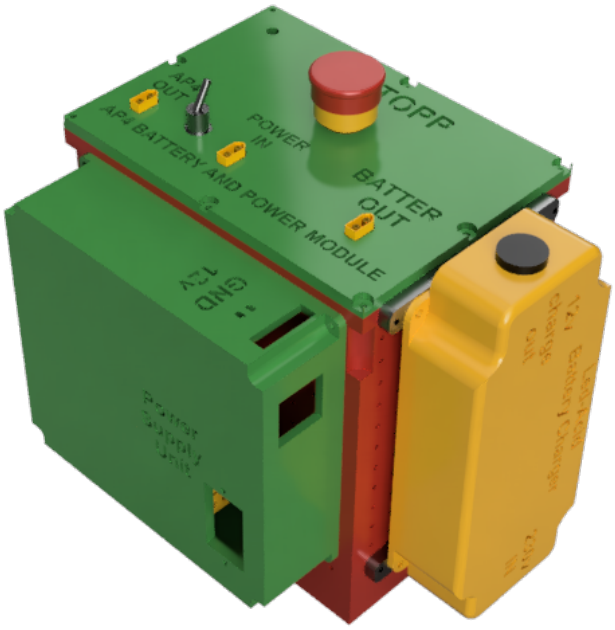


Figure 5.7: Rendered illustration of the Power module. The main component is the battery holder (red) to which the remaining components are mounted to. The battery holder holds the lead acid battery. The battery charger (yellow) is mounted onto the side. A PSU module (green) is mounted to the side. The power module interface (green) is mounted on top of the battery holder, providing an interface to the power module. An emergency stop, on switch and power connectors.

5.5.3 Battery and charger

The battery for the go-kart is a 12 V lead acid battery. The battery is typically used for lawnmowers and similar machines. The corresponding battery charger selection is based on compatibility with the battery. The battery capacity is 30 Ah and weighs about 8kg. From the derived system power consumption found inTable 5.1, the theoretically run-time of the system can be determined:

$$t_{run} = \frac{12 \cdot 30}{73.2} \approx 4.9 \qquad [h] \qquad (5.2)$$

5.5.4 Power supply unit

To enable testing of higher levels of algorithms without the movement and unnecessary use of the battery that will shorten its lifespan, a power supply unit (PSU) connected to the electrical grid is used.

5.5.5 User interface

The user interface includes three different outlets in order to select which power source should be used. These outlets are called, *Battery out*, *Power in*, *AP4 out*. *Battery out* is directly connected to the battery terminals, with the killswitch in series. The schematic of the power module can be seen in Figure 5.8. The electrical circuit for the user interface and connected sensors is located in Appendix A. For example, the PSU could power the platform whilst the battery is charged at the same time.

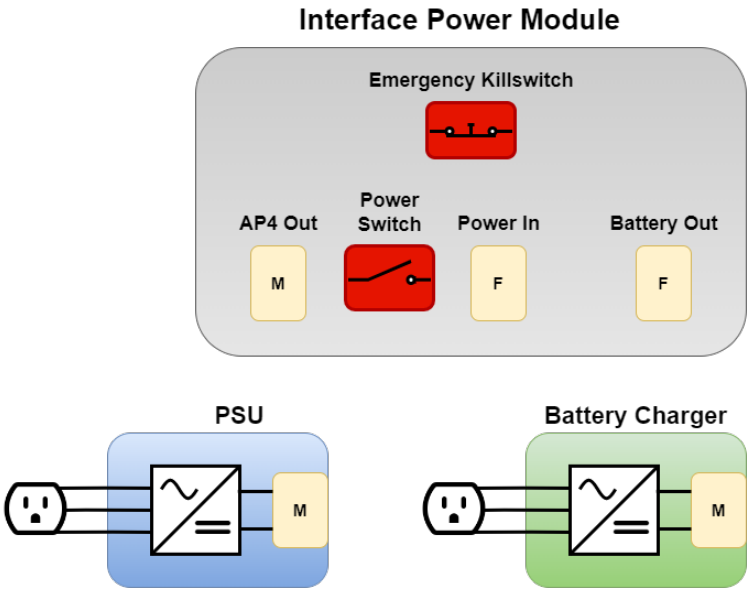


Figure 5.8: Illustration of the power module top interface. Showing the three different power connectors, *AP4 OUT*, *POWER IN* and *BATTERY OUT*. The Emergency killswitch and power switch is also illustrated. The PSU and battery charger and their respective power connector is shown in the bottom om the figure.

6

Software Design of Next Generation Autonomous Platform

This chapter aims to describe the software architecture and functionality of autonomous platform generation 4. The software will be described top down, starting with the purpose of the software architecture. Next, the three different software layers that have been implemented in this thesis are described. Then moving through the different software layers, describing functionality and structure.

6.1 Software design overview

In order to enable autonomous driving there are many different software components that need to interact. Some processes and hardware interfacing code are time-critical. Therefore, to separate software functions that are time-critical in different aspects, the software architecture of the autonomous platform is split into three separate components, a high-level control software component, a low-level hardware interfacing component, and an embedded system software component. This can be seen in Figure 6.1.

By splitting the software into three separate parts, the probability of being held up by another software component unit will be less likely. This still places a high demand on time criticality on software closer to the hardware, but the risk of being held up decreases. The embedded software works in time chunks measured in micro- or milliseconds whilst higher-level control software can be allowed to process data and take up computing resources for several seconds. It is therefore reasonable to implement algorithms that may take a longer time to execute on high-level software. Furthermore, the choice of splitting up the software into these three parts also allows one to choose a suitable programming language for each task and also onto separate hardware. Higher-level control algorithms can be implemented using Python and take advantage of numerous libraries available. Python is not the best choice when requiring fast execution time, but is very suitable to design advanced algorithms in an easy way. At the same time, lower-level software that is very time-critical can be implemented using C or C++. It also allows for the different software compartments to be configured individually. For example, enabling a real-time kernel

on the operating system responsible for one software block. For the autonomous platform, a more detailed view of the software blocks and their underlying software dependencies can be seen in Figure 6.1. The following sections will describe how each software block is designed in detail.

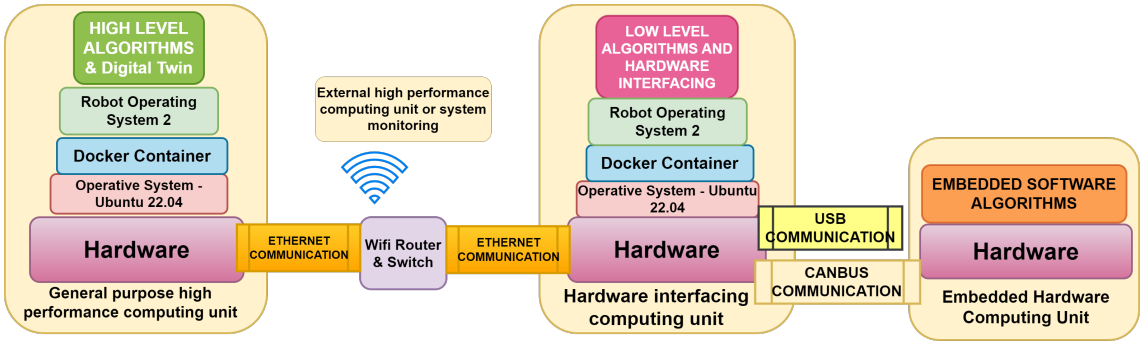


Figure 6.1: Software architecture components, illustrating the relationship between the high-level software, low-level software and lastly the embedded software. Each software is implemented on a separate hardware unit, ranging from HPC down to a simple microcontroller.

6.2 Linking of software layers

This section aims to describe how the three different software layers are communicating.

6.2.1 High-level to low-level software link

The high-level and low-level software components are linked together using an Ethernet interface, as illustrated in Figure 6.2. Software-wise, this makes the connection between high and low-level software very robust and easy to use. The software components can be connected together as long as they are located on the same network. By configuring a ROS environment variable, *ROS_DOMAIN_ID*, the underlying communication middleware, DDS, will ensure that all software running in any location on the network can communicate with one another using the ROS2 API. This also means that the connection between high-level and low-level software can be done wirelessly over wifi, and is not limited to Ethernet.

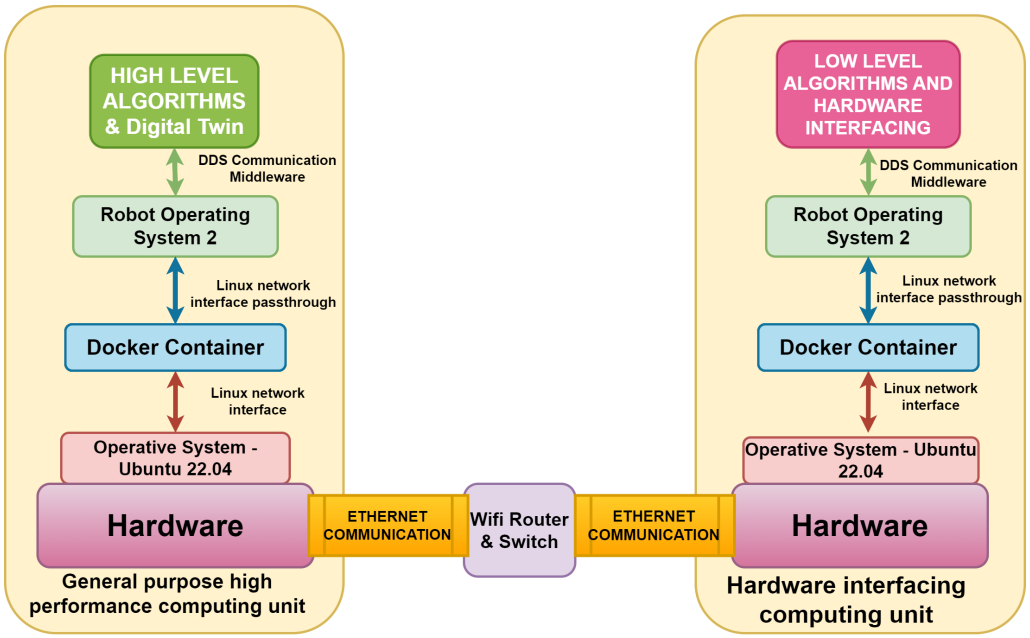


Figure 6.2: Overview of software layers communication inside the HPC unit and respectively HWI unit which runs a docker container on its operative system on Ubuntu. These two software components communicate using Ethernet and pass the ethernet interface to each container. The ROS2 software uses DDS communication which interfaces with the Ethernet to allow software in the HWI unit to communicate with software in the HPC unit.

6.2.2 Low-level to embedded software link

A major component in the low-level software is the interface between ROS2 and the embedded software. This component can be seen in Figure 6.6 Hardware-wise it uses a CAN bus network, as explained in section 5.2. The low-level software and embedded software need to communicate seamlessly and without any overhead or disturbances to make the system as robust as possible.

A detailed illustration of how the low-level SW and embedded SW communicate can be seen in Figure 6.3. The hardware components are described in subsection 5.3.1, software-wise, a Linux socketCAN is created which interfaces with the hardware. Any CAN messages received from the embedded SW will be put on the Linux socket CAN interface. This interface is then exposed to the container containing the low-level SW. Therefore, CAN messages received or sent on the socket CAN interface will be relayed to the hardware components on the central computer. CAN frames are encoded and decoded using a unified library, *CAN_DB*, to ensure that the two different software components are fully compatible and send data in exactly the same format.

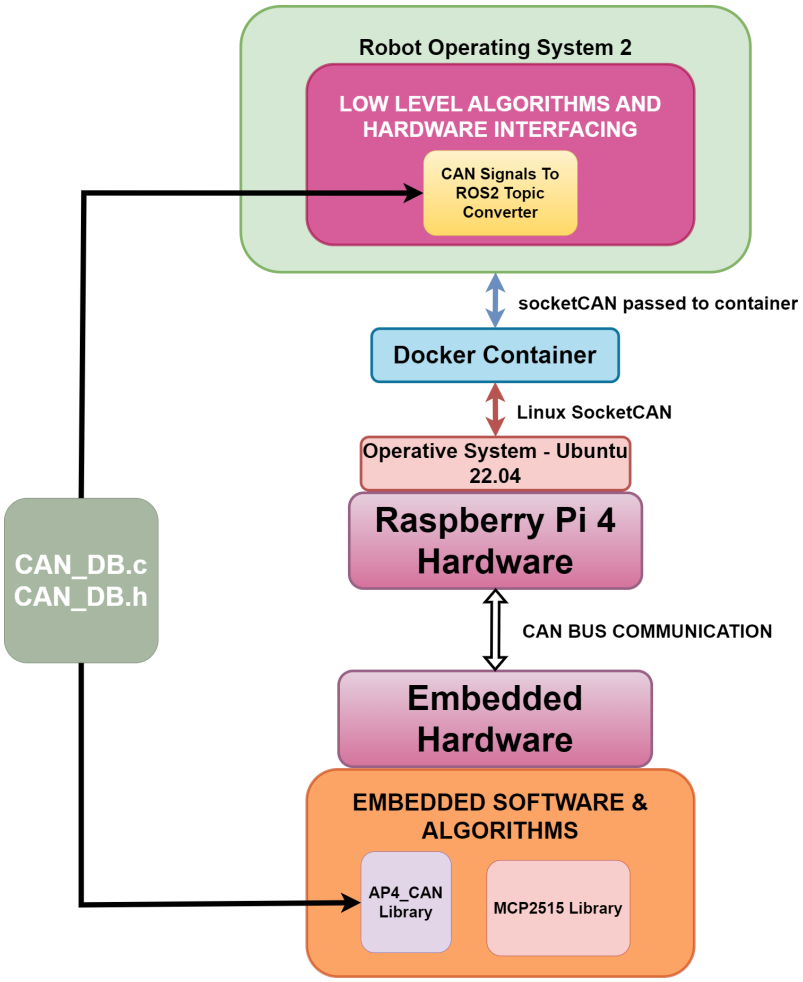


Figure 6.3: Common CAN database for each component within the AP4 system. Used for both the low-level and embedded software in the ECUs. In order to encode and decode each CAN message according to a set standard.

An external library is used to generate the unified CAN database, *CAN_DB*, into C code. It turns the CAN frames defined inside a CAN database file (dbc) into objects and structures that can be accessed programmatically. The CAN database can be found in Appendix D. The C code also provides an easy interface API for encoding and decoding CAN frame data. Every time the dbc-file is changed, the accessible structures and objects in the code will be changed. It also makes the CAN frame database very scalable, since any new code representing frames and or signals will be generated according to the standard specified by the dbc to C converter library. The workflow is illustrated in Figure 6.4.

Figure 6.4: Conversion from CAN database to C code schematic flow is illustrated. The dbc file can be changed manually to add CAN frames and signals to the software. The DBC to C external library can be run by a developer when the dbc file has been changed to generate new C files with a specified structure containing every CAN frame and signal.

The *CAN_DB.c* and *CAN_DB.h* is generic C code and will therefore be the base on which software specific to AP4 can be built. From this base, SW specific to the local level and embedded level can be written to utilize the generated *CAN_DB* programming structures. These two components are *CAN_AP4* and *can_to_ros2_topic_converter*, the dependency can be seen in Figure 6.3. By being based on the same base code, CAN frames can be encoded and decoded using the same method.

6.3 High level software design

The high-level software block is designed to be hardware agnostic and function regardless of what algorithms and hardware exist on the lower-level software blocks. This abstraction makes the high-level system very modular and compartmentalized. This is done by using the ROS2 convention of naming and manipulating data. By using the ROS2 frameworks API, the software can be created in a modular way. An overview of the high-level software functionality can be seen in Figure 6.5.

The developed higher-level software consists of two software functionalities, autonomous driving and a digital twin.

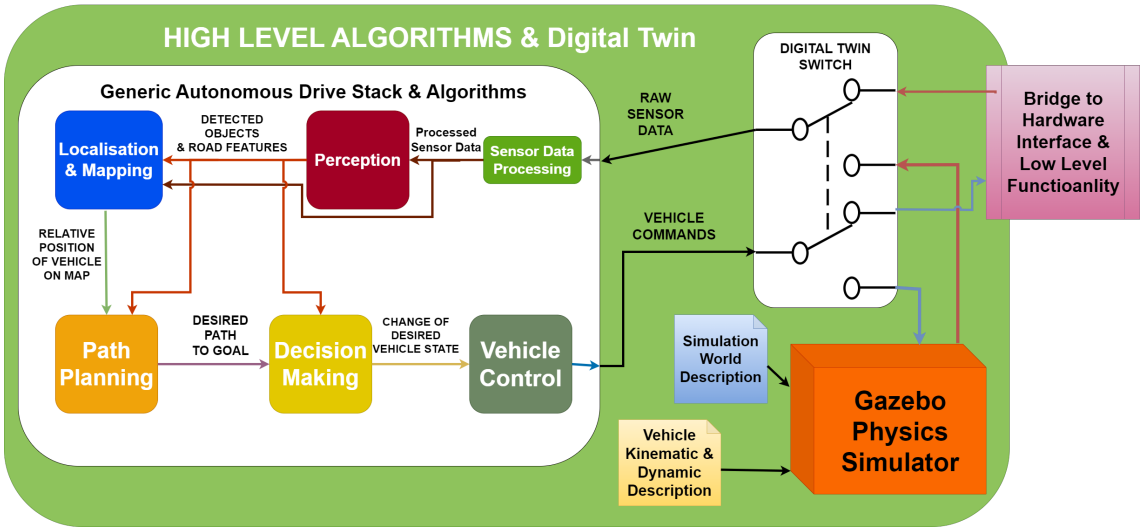


Figure 6.5: Overview of High-Level Software where the Gazebo simulator and the vehicle control are actually implemented. However, the overall structure should follow this to enable a structured way of working in AD systems, described in subsection 2.2.1.

6.3.1 High-level control algorithms

The high-level software is expected to match the capability of autonomous driving functions used in modern vehicles of today and tomorrow. To accomplish hardware agnostics, the input to high-level software algorithms will come in the form of sensor data on ROS2 topics which follow the ROS2 convention of describing the vehicle state, sensor states, and such. This transformation of sensor data into ROS2 standardized topics is further described in the low-level software section 6.4.

As described in subsection 2.2.1, AD and ADAS software often consist of several functions connected together, this is visualized in Figure 6.5. Therefore it is important that high-level software can accommodate these functionalities. By using ROS2 there is the possibility to write software in Python or C++ which can be decided by the developer, the output from these algorithms can use the standardization of the ROS2 API to make different software components written in different programming languages compatible. As for testing and developing new AD functions, utilizing Python and its readily available libraries is advantageous. By creating a software bridge to the ROS2 network it is possible to connect existing AD/ADAS functionalities developed outside the ROS2 framework.

6.3.2 Digital twin

The digital twin functionality is implemented in the high-level software system. The digital twin is described in detail in section 7.4. Placing the digital twin in the high-level software block has several advantages. Firstly, the control and sensor interface between the simulation and control software can use generic standardized ROS2

formats. Meaning the high-level control functions can be switched from sending its commands to the low-level software components and instead send the commands to the digital twin. Secondly, since this software layer is placed inside a docker container, the high-level control functions can be tested on the digital twin on a separate host computer not connected to the physical platform, seamlessly.

6.4 Low-level software design

The low-level software block is supposed to be the interface that connects the high-level control algorithms and the embedded software blocks. It is also responsible for taking in sensor data from the autonomous platform and converting it into a suitable format that the high-level control algorithms then can use. The software developed is designed to be hardware-specific and is therefore not supposed to be portable, i.e., the software should always run on a device that is connected is mounted on the platform. The low-level software block has four major components as shown in Figure 6.6. The components are *CAN translation*, *vehicle control*, and *joystick manual control*.

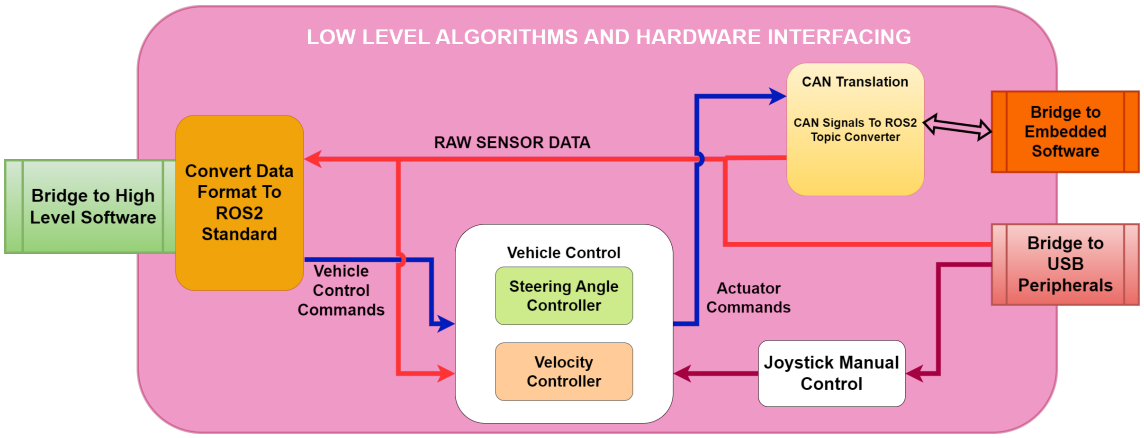


Figure 6.6: Overview of Low-level Software implemented on the HWI unit. It consists of several ROS packages including vehicle control, joystick manual control, CAN translation, and bridge to the high-level software unit.

6.4.1 CAN translation

An interface to convert CAN bus message data into ROS2 topics has been developed for AP4, as mentioned earlier. The low-level SW contains a software module called *CAN Signals To ROS2 Topic Converter*, see Figure 6.6. This software module is built on two ROS2 packages, a native ROS2 package called *ROS2_socketcan* and a custom-made package called *CAN_signal_to_ros2_topic*, as illustrated in Figure 6.7. The *ROS2_socketcan* package starts two ROS2 nodes, *can_sender* and *can_reciver*, which interfaces with the Linux socket CAN interface, which enables them to receive or send CAN frames from the CAN bus.

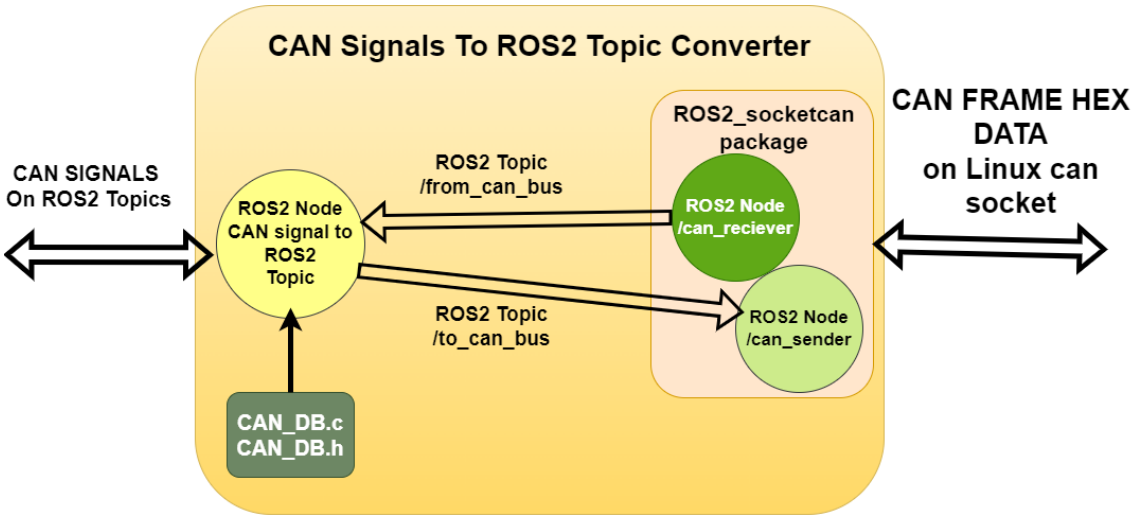


Figure 6.7: ROS package to convert topics to CAN frames and vice versa, that consist of three different nodes and the auto-generated c files described in subsection 6.2.2 to encode and decode CAN frames. ROS2_socketcan package interfaces with the CANhat mounted in raspberry pi to enable CAN communication.

The *CAN_signal_to_ros2_topic* package exposes individual CAN frame signals onto distinct ROS2 topics. It allows CAN frame signals to either be read or written from the ROS2 topic API. A detailed view of this ROS2 node can be seen in Figure 6.8. On a high level, it receives CAN frame data coming from the *ROS2_socketcan* node *can_reciever*. It then decodes the CAN frame and splits it into its signals which are then published on separate topics. One topic for each CAN signal. In the same way, it listens to signal topics in order to set signal values for a specific frame. Once a new signal value has been received, it encodes the signal values into a CAN frame and publishes them onto a ROS2 topic which will be received by the *ROS2_socketcan* node *can_sender*. This will in turn publish the frame containing the specific signal onto the CAN bus network. The *CAN_DB* code is used to decode and encode the CAN frame signals in the same manner as the embedded SW, making them compatible. The translator node is a highly scalable solution and builds upon the common interfaces in *CAN_DB*, meaning it can be modified to process more CAN frames and signals as the CAN database file grows. It also provides a layer of abstraction for any higher-level algorithms that need to communicate with the embedded SW.

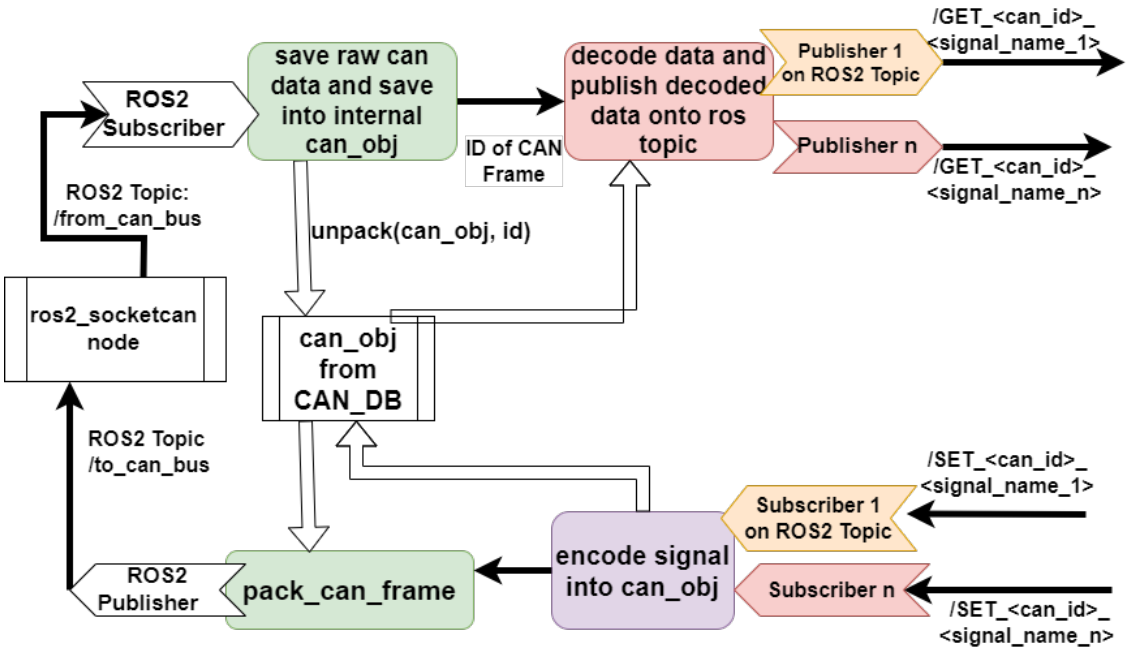


Figure 6.8: Flowchart of the ROS2 Node CAN signal to ROS2 topic which can be seen in the Figure 6.7. That translates CAN objects into ROS topics as well as building CAN frames from ROS topics.

Vehicle control

The vehicle control software algorithms ensure that requested vehicle velocity commands are processed and converted into signals that are sent to the vehicle control actuators in the embedded software block. There are two attributes to control, velocity and steering rate. The higher-level software components can request a motion velocity and turning rate by sending a Twist-type message on the ROS2 topic /cmd_vel. This message contains six relevant fields, velocity in $\frac{m}{s}$ in x, y and z and turning rate in $\frac{rad}{s}$ around each axis, R_x, R_y and R_z.

Relative to the autonomous platform frame of view, it can only move longitudinally forward or backward and change its turning rate. Therefore only the linear x component and angular z components from the /cmd_vel topic can be used to command the platform, this is visualized in Figure 6.9. The longitudinal velocity can be changed by controlling the propulsion actuators and the turning rate can be changed by controlling the steering angle.

An open loop controller converts the velocity inputs from high-level control to a voltage that is applied to the pedal actuators. An additional open loop controller is used to convert steering angles into corresponding encoder values for the DC motor. Essentially, the controllers are open-loop control systems, simply scaling to the correct units and limiting the output values according to the capabilities of the actuators. These controllers are illustrated in Figure 6.10.

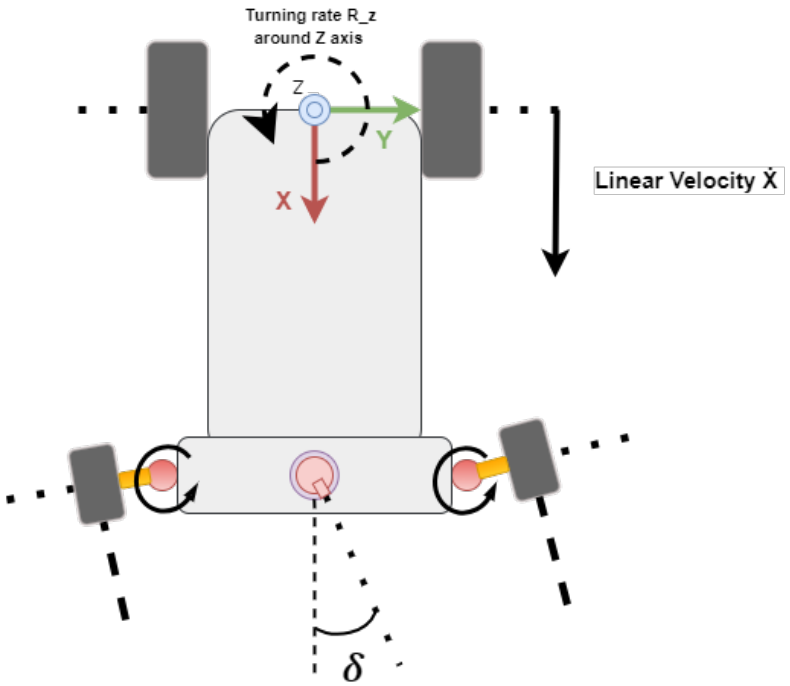


Figure 6.9: Kinematics of AP4, illustrating the relationship between the base frame's origo, velocity direction and steering angle. The complete description of the kinematics with actual transposes and transformations is described in an URDF-file.

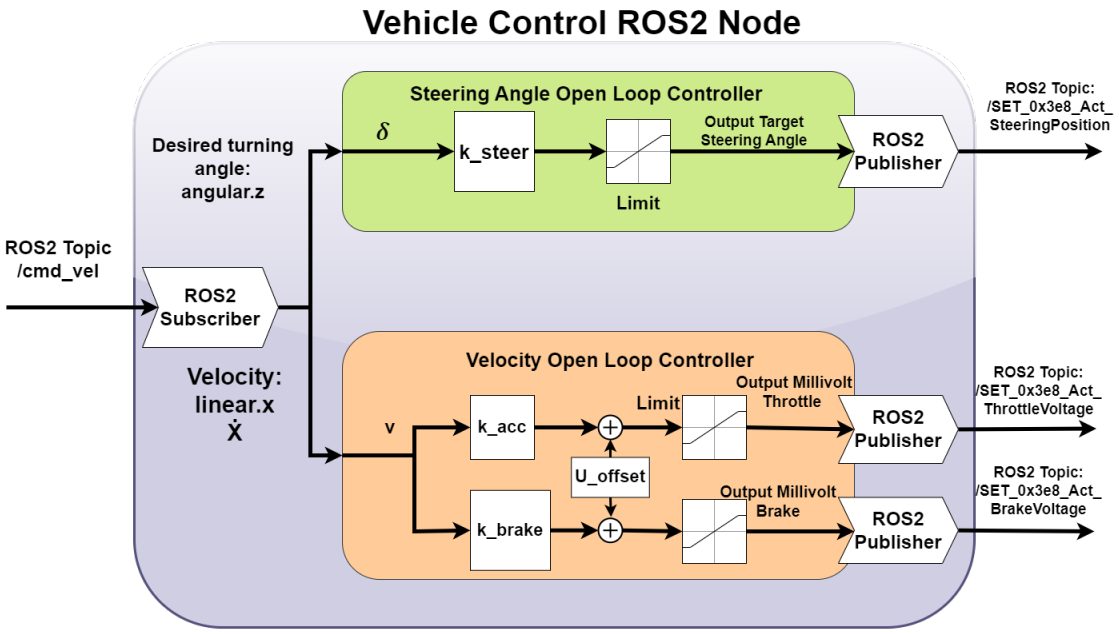


Figure 6.10: Vehicle Open Loop Controller, consisting of three proportional gains, mapping control inputs to valid values for the steering angle, and voltage for brake pedal and throttle respectively.

Steering angle controller gain

The steering hardware has an internal controller, therefore it is sufficient to control the requested steering angle actuator command using an open loop with a proportional gain. The steering angle domain of the physical steering system of AP4 is between -40 degrees and 40 degrees. However, the input to the vehicle controller is of type /cmd_vel where the turning rate R_Z around the z-axis will be used and converted to degrees. Assuming a linear relationship between the turning rate and the string angle:

$$\delta_{steer} = k_{steer} \cdot R_z \qquad [\text{degree}] \tag{6.1}$$

As the turn rate R_z is limited to [-0.4,0.4], whilst the steering angle is limited to [-40,40], k_{steer} can computed as

$$k_{steer} = \frac{MAX(\delta_{steer})}{MAX(R_z)} = 100 \qquad \left[\frac{\text{degree}}{rad/s}\right] \tag{6.2}$$

A safety margin is included to not meet the limit points and potentially damage the equipment, so k_{steer} is chosen to 90.

Velocity controller gain

The embedded SPCU software actuates the propulsion on the platform, as described in subsection 3.2.2, and the measured voltage applied to the brake and throttle pedals is between 820 and 4350 mV which can be seen in Table 3.3. Therefore the gain of the velocity open loop control system needs to convert a desired velocity v in m/s as input from the /cmd_vel to a voltage magnitude U_{out}.The control system can be seen in Figure 6.10 and the schematic of actuators can be seen in Figure 5.4, the general formula for controlling the pedal voltage is formulated as:

$$U_{out} = U_{offset} + K \cdot v \ [m\text{V}] \qquad 820 \leq U_{out} \leq 4350 \ [m\text{V}] \tag{6.3}$$

where v is the velocity reference as an output from the cmd_vel ROS topic, and K is the velocity controller gain. Furthermore. the offset U_{offset} is set to be a constant with the value of 820 mV. Because that is the measured idle voltage when the go-kart's pedals are not pressed. Assuming a linear relationship between applied voltage to the throttle pedal and actuated velocity, a simple proportional gain can be implemented to convert the velocity in m s to voltage V. Just like in the calculation of the steering control system gain, the calculation is based on the limits of the actuator. The velocity range for the go-kart is between 0 and 2.2 m s, which results in:

$$K_{acc} = \frac{4.350 - 0.820}{2.2 - 0} = \frac{3.530}{2.2} = 1.6 \left[\frac{V}{\text{m/s}}\right] \tag{6.4}$$

$$K_{brake} = -\frac{3.530}{2.2} = -1.6 \left[\frac{V}{\text{m/s}}\right] \tag{6.5}$$

To engage reverse on the platform there has to be a discrete transition, by engaging the brake pedal two times in a short time period. When the go-kart is in reverse mode, reverse velocity is controlled by the throttle voltage. This makes the reversing act just like in a real car, by engaging the reverse gear and controlling the velocity through the car's pedals.

Joystick manual control

The joystick control functionality utilizes two standard ROS2 packages, *joy* and *teleop-twist-joystick*. The first package takes a joystick input on the host computer and turns it into data available on a ROS2 topic. Secondly, *teleop-twist-joy* takes this data and converts it into the desired velocity published on a ROS2 topic according to a set standard. The *teleop-twist-joy* node can be configured to output the desired speed and turning rate within certain limits.

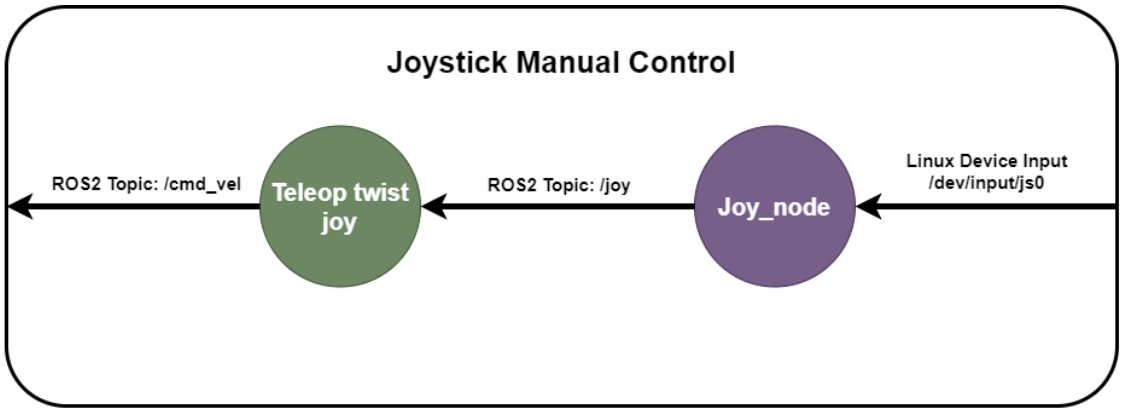


Figure 6.11: ROS package for joystick manual control consisting of two nodes, Joy_node which interacts with device inputs and publishes as topic /joy. Then Telop twist joy converts it into ROS convention of the datatype Twist in /cm_val.

6.4.2 Real-time capabilities in software

Having dedicated software responsible for interfacing with hardware and running fast control loops ensures that the hardware interfaces will not be kept waiting by higher-level control. The underlying framework, ROS2, has been proven to satisfy soft real-time capabilities in [51]. The docker container in which the framework runs and the underlying Linux kernel can be configured to be able to run real-time applications. Therefore low-level software can be considered to be real-time capable if the underlying software is configured correctly.

6.5 Embedded software design overview

Every embedded software block is designed according to a predefined structure, as seen in Figure 6.12. This structure is designed to be modular and expandable for future functionality. It consists of a software bridge to the low-level software, built on CAN bus protocols. A CAN bus device library interfaces with the CAN bus device. A custom CAN bus interface, AP4_CAN, has been developed to interface between the low-level and embedded software as seamlessly as possible. The packaging of individual can frames is abstracted into calling functions to retrieve and send data. The custom library can be configured to periodically send important data over the CAN bus interface. The embedded software contains a generic code section that can be adjusted for all scenarios where one would need to convert data on the CAN interface to control actuators or pass sensor data through. This code can be adjusted to run any code that could be run on a microcontroller software platform. Further libraries, specific for the sensors and actuators used, can then be integrated into the embedded software stack which interacts with the embedded hardware.

Making the embedded software blocks as generic as possible allows for adding several embedded software units according to a designed software standard. It also ensures that future embedded software can be compatible with existing embedded

software. This is made possible by abstracting the CAN interface into a dedicated can interface library. Each embedded software can still be configured around this pre-defined structure, in order to connect any new external sensor or actuator, by simply appending a relevant library onto the embedded software stack.

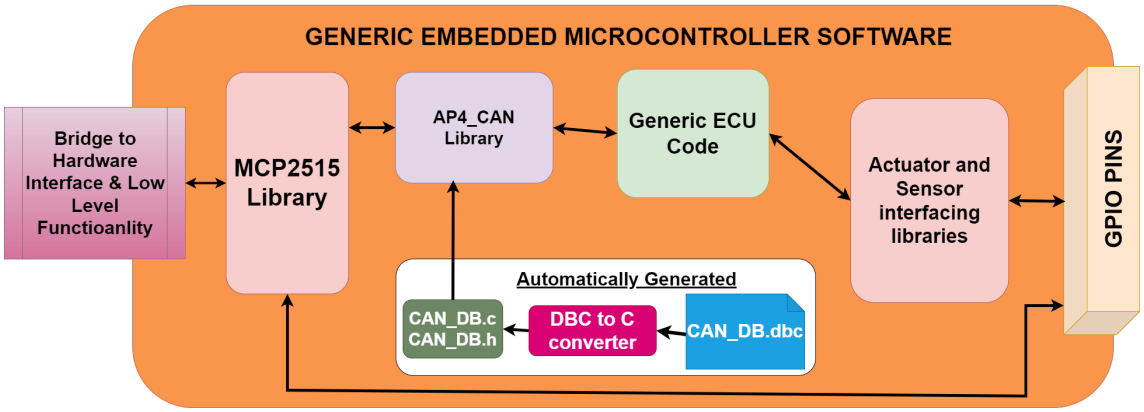


Figure 6.12: Embedded software template functionalities for the generic ECU base, including libraries to encode, and decode CAN frames.

6.5.1 Implemented embedded software

The SPCU has been designed and implemented as an embedded software component. The software in this ECU is responsible for receiving commands from the low-level software block and controlling the desired actuators, an illustration of the software functionalities and building blocks can be seen in Figure 6.13. It also sends back the measured steering angle at a rate of 10 Hz. Another functionality is that the software can respond to a heartbeat request, meaning if a heartbeat request has been sent over the can bus the SPCU will respond and notify that it is still working as expected. If any error occurs during runtime, this will be sent as a frame onto the can bus. An in-depth description of the software and how it works is illustrated as a flowchart in Figure 6.14

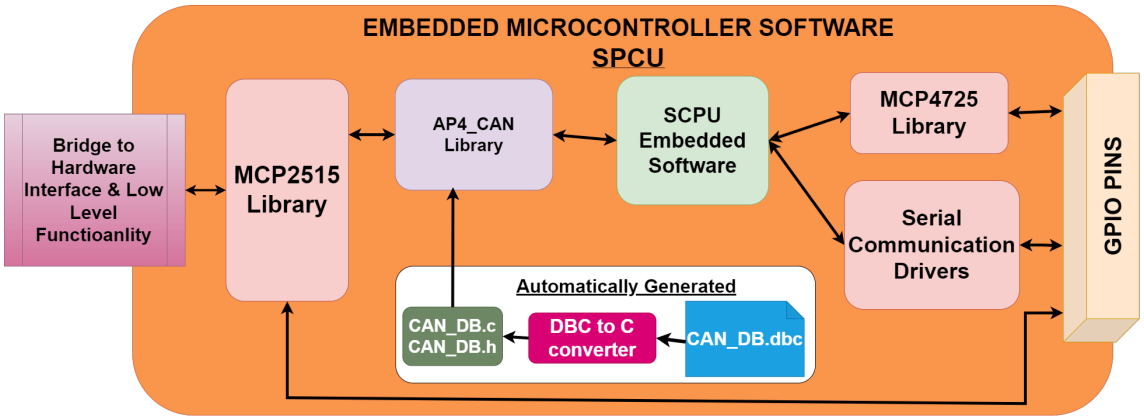


Figure 6.13: Embedded software in SPCU including the general code from the generic ECU base template code and special libraries to interface with steering and propulsion modules.

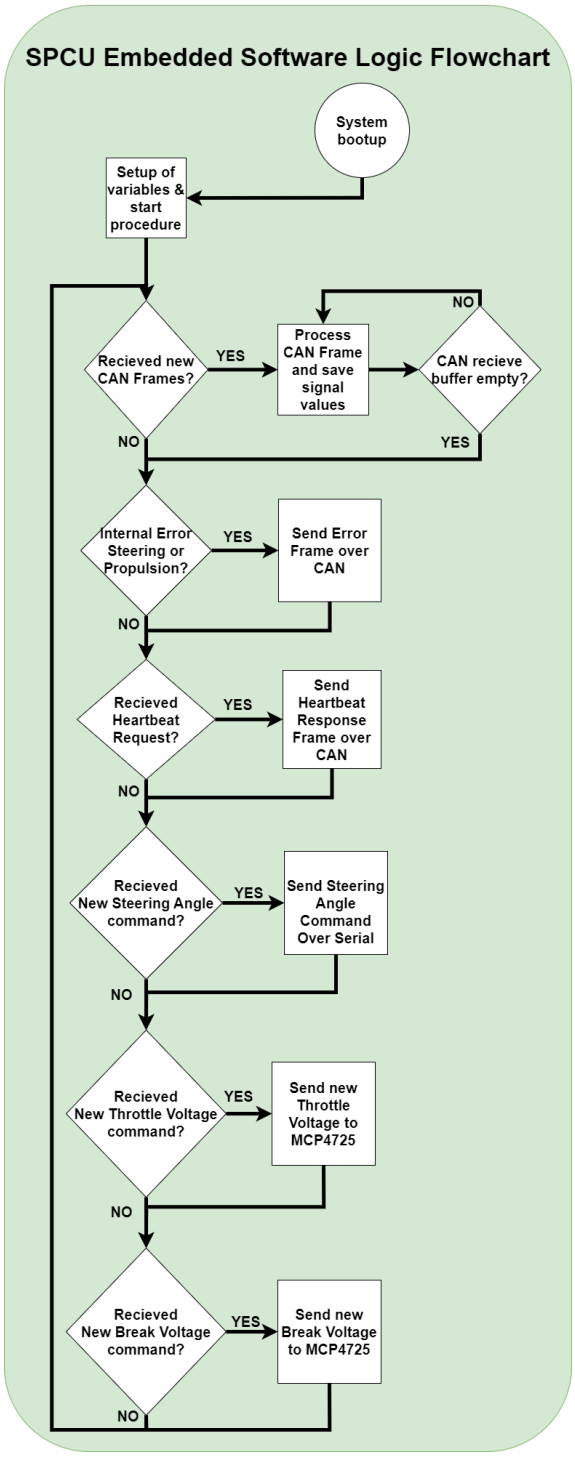


Figure 6.14: SPCU logic flowchart describing initialization of variables, set up of communication protocols, CAN frame handling, error handling, and how the SPCU interface with the actuators.

7

Results

The results chapter will serve the purpose of showing the results from the verification, highlighting the autonomous platform key features, and describing the digital twin functionality.

7.1 Verification results

The methodology used to verify each specification on the autonomous platform is summarized in chapter 4. This chapter will present the specifications, verification methodology and results of the verification. This is done for every function in the specification list.

BMS and power management system

Specifications are as follows:

- Supplies 12 V $\pm$ 0.5 V to system continuously
- Voltage does not drop more than 0.5V during load
- Autonomous platform should be able to run for one hour continuously without being connected to a wall outlet

Verification is done by:

- Measuring voltages in the system using a multimeter
- Timing the runtime during tests

Result: The BMS and power management system was run for half a day with mixed usage, i.e. being driven with a controller and varying different aspects of the platform. Below *idle* is defined as the platform being stationary with all ECUs being powered up, *no load* as driving the go-kart around without a person sitting in it and *load* as driving around with a human operator in the seat.

Table 7.1: Platform battery voltage measured at the start and end of the testing session. The session lasted five hours and tested the capabilities of the platform. Utilizing all mounted components, including SPCU hardware node, Raspberry Pi 4, wifi router, and steering motor.

	Start	**End**
Boot down (Voltage)	12.92	**12.62**
Boot up (Voltage)	12.62	**12.19**

Table 7.2: Voltage readings of power supply for the platform during different loads. No load is without moving the steering and with load is whilst moving the steering motor between maximum and minimum limits.

	Idle	No Load	Load
Voltage of system	12.62	12.01	11.2

As can be seen in Table 7.2, the voltage drop during usage drops below the specified voltage during load. Therefore it does not meet the specification of being less than 0.5 V. The voltage measurements were conducted using a multimeter *Fluke 115*. Furthermore, the run-time specification is also met with mixed driving during 5 hours, approximately using 60% of the time resulting in a continuous run-time of 3 hours corresponding well to the theoretically calculated value of 4.9 hours.

On top of the minimum requirements, a crucial aspect of the desirable requirements for the power management system was measured. The continuous current consumption was monitored during different load scenarios, to validate that the system could provide enough amperage according to the desirable specification. That states that the system could feed 10 A continuously. The current was measured using a current clamp *Fluke 365*, clamped around the battery positive out cable. The platform was placed on a go-kart track surface whilst performing the tests. Note, that even though the current was measured to be 6.5 amps, the battery fuse of 15 amps blew during the load test. The Ninebot segway propulsion unit had 78% state of charge at the end of the test session. Due to the voltage dropping below the specification during load, the platform does not meet the requirements.

Table 7.3: Platform current consumption measured at the start of the testing session. Utilizing all mounted components, SPCU hardware node, Raspberry Pi 4, wifi router, and steering motor. Idle means no movement of the platform, load means moving the steering motor between its maximum and minimum limits, and load is with a person sitting on the platform.

	Idle	No load	With load
Current Consumption [Ampere]	0.65	2.4	6.5

Steering control

Specifications are as follows:

- Possibility to manually calibrate steering angles
- Measured steering angle should be within 20% of the set steering angle

Verification is done by:

- Measuring steering angle of the platform
- Optically see how the steering limits change after manual calibration

Result: The steering limits could be set by tuning the motor driver that is mounted to the steering shaft. After tuning, the maximum and minimum limits were observed to change.

The turn radius was measured by moving setting the steering angles to their maximum steering limits and slowly driving forward until the platform had completed half a circle. The diameter of this circle was then measured. This was repeated three times. This was done on a go-kart track surface. The steering radius while turning to the left was 3.2 meters and 4.1 meters when turning to the right. The requested steering angle versus the actual steering angle was measured by requesting a steering angle in software and measuring the set angle on the hardware. This was done using a protractor. The results from this can be seen in Table 7.4.

Table 7.4: Measured steering values of steering hardware with respect to requested steering angles in software.

Reference (degree)	-30	-15	0	15	30
Measurement (degree)	-25	-10	3	10	25

Propulsion control

Specifications are as follows:

- In reverse mode, the internal go-kart system detects the state and starts beeping
- When the throttle is increased, the relative velocity is positive
- When braking is increased the relative velocity goes toward zero
- At full bake, the braking distance should be lower than 6 meters.
- The delay between the HWI communication and the go-kart segway shall be

less than 100 ms.

Verification is done by:

- Optically see how the platform moves according to the commands given.
- Manually measuring braking distance from maximum speed

Result: The propulsion control behavior was measured by connecting an external joystick to the autonomous platform and performing the verification as set up in section 4.1. The Segway which is responsible for propulsion was configured to be in safe mode, limiting the velocity to 8 km/h. During the tests, the following behavior was observed:

- While requesting a positive velocity, the platform moved forward
- The platform could enter reversing mode by applying the brake pedal twice in quick succession. This is further explained in subsection 3.2.2. The segway unit then started to beep.
- While in reverse mode and requesting a negative velocity, the platform moved backward.
- The braking distance from maximum speed, without a driver, is on average 0.65 meters
- The braking distance, from maximum speed, with a driver, is on average 2 meters

The propulsion control, therefore, meets the specifications.

Computing unit

The computing mounted on the platform meets the specification in section 4.1. The Raspberry Pi and the HPC are seen as one unit.

Table 7.5: Minimum specification of computing unit compared with chosen hardware to see if it meets the stated specifications. In the table the chosen hardware is a Raspberry Pi 4b.

	Requirement	**Chosen HW**	**Compliance**
Can run Ubuntu?	Yes	Yes (Ubuntu 22.04)	Yes
Nr GPIO pins	10	40	Yes
Nr USB ports	1	4 (2 USB 2.0, 2 USB 3.0)	Yes
Nr HDMI ports	1	2 (Micro HDMI)	Yes
Storage Capacity	120 GB	32 GB	No
Ethernet port	Yes	Yes (1000 Mb/s)	Yes

The Raspberry Pi 4b uses micro-SD card storage, therefore the current storage capacity does not meet the stated requirement. This storage can be upgraded in size though. As mentioned earlier, the HPC will have the possibility to expand the storage capacity of the central master computer and the combined computing units will, therefore, meet the stated specifications.

E/E architecture and modularity

Specifications are as follows:

- One central computing unit for high-level control
- One hardware interfacing ECU
- Follows mostly the industry standards
- Response time ECU $\leftrightarrow$ Central, less than 500 ms
- ECU update frequency not lower than 20 Hz.
- While integrating new ECUs, older ones should still be compatible without needing updates (SW and HW)

Verification is done by:

- Measure time of the program in the SPCU
- Script to ping the response time of the pipeline, ECU $\leftrightarrow$ Central Master Computer

Result: As can be seen in the Table 7.6 the frequency and response time specifications are met. One exception is when the SPCU reaches its maximum update periodicity but the median of 3 ms. Furthermore, as described in chapter 5, the characteristics of a centralized E/E architecture are achieved and as well as the modularity of adding new physical components. The modularity of adding new software is also met, which is described more in chapter 6.

Table 7.6: SPCU timer, measured periods and frequency for running the SPCU code.

Function	Max	Median
SPCU update period [ms]	60	3
SCPU update frequency [Hz]	16.7	333.3

Transportation rig / lift aid

In order to test the autonomous platform, it was moved to and from a test site, this tested the portability of it. No dedicated rig to transport the autonomous platform could be constructed in time. It could fit in a Volvo V90 trunk with two of the three back seats folded down. There was also room left for test equipment. Therefore it does not meet the requirement of having a dedicated transportation rig, even though it is possible to transport the rig as it is now.

Usability / Unit Tests

Specifications are as follows:

- Automatic startup procedure when powers on
- When an error occurs, give the user information about where and why it occurred. In each ECU and Central Master Computer
- Documentation on how each ECUs constructed hardware and software, how to modify

Verification is done by

- Automatic scripts to detect errors and print

Result: As can be seen in Figure 6.14 error handling is implemented in embedded software. Furthermore, a debug interface is enabled through serial communication in the general ECU base which can be seen in section 5.2. The startup procedure occurs both in the embedded software as well as in the central computer unit. The documentation consists of internal documentation and descriptions in this thesis.

(SW) Modular software for control

Specifications are as follows:

- Is the SW control designed in such a way that velocity and steering can be set without any updates on the low-level SW and communication tools?
- Is it possible to access the state of AP4 and its readings from sensors?

Verification is done by:

- Creating a very simple high-level control, only affecting the mentioned high-level SW

Result: The software control of the autonomous vehicle can be considered to be modular. It is designed in such a way that velocity and turning rate can be set on

a high level without changing anything in the low-level code or embedded software. The velocity and steering angle can be set by publishing commands on the *cmd_vel* topic, which is a ROS convention on how to define these values. This can command can come from any high-level software, it will be interpreted by the low-level software in the same way. A detailed description of how this is done can be found in section 6.4 As a proof of concept for this modular control, a joystick is connected to the system and sets requested velocity and steering angles. In the same way, sensory data can be configured to be accessed from lower-level software algorithms using standardized ROS conventions. The autonomous platform is therefore considered to meet the specifications for *Modular Software for Control*.

Safety

The requirements are as follows

- All circuits should be dimensioned accordingly to the specifications of each circuit board. Furthermore, a sufficient wire diameter shall be used according to the current flowing through.
- Battery should be placed and used in a safe way by users
- Battery should have a kill switch connected to the emergency stop/manual switch
- Go-kart shall have a designated fire extinguisher for electronic purpose
- Clear schematics of current carrying cables
- No open terminals

These are specified as hard requirements and will either be met or not.

Result: The circuits designed and implemented for the autonomous platform are described in Figure A.2. These have been dimensioned accordingly to their power consumption. As described in section 5.5, the battery is mounted in a sealed housing and has a standardized power outlet. Therefore there are no open terminals carrying current. It has an emergency stop button that will turn off every component on the platform except for the segway used for propulsion. When the SPCU is turned off, the segway will switch to listening to pedal inputs. Meaning if no throttle is applied it will engage the motor to brake. While developing and testing the platform, a fire extinguisher is close by. In summary, the requirements are met.

Sensors on autonomous platform (HW and SW)

Specifications are as follows:

- A list of most commonly used sensors in AD and ADAS

- Meet the reasonable physical scale limitations of AP4

Verification is done by:

- Literature study

Result:
A list of the most common sensors used for autonomous drive and theory regarding them can be found in subsection 2.2.2. Taking the size limitations into account it would be possible to integrate some version of the onto the autonomous platform. The only sensor which is integrated so far is the camera, which does fit onto the autonomous platform. Therefore this function meets the stated specification.

Budget

Specifications is as follows:

- Total cost of upgrading the go-kart into an autonomous platform shall cost up to 30 000 SEK

Verification is done by:

- Listing budget in an Excel sheet

Result:
The Bill of Materials (BOM) for the autonomous platform can be found in Appendix C. The BOM cost is well below 30000 SEK. Therefore the specification is met.

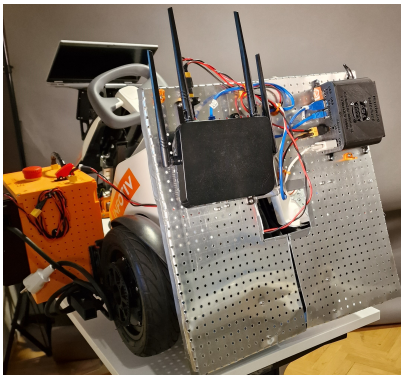
7.2 Physical results on modular design

This section illustrates the physical autonomous platform built by showing photographs of the hardware, highlighting components developed in this thesis.

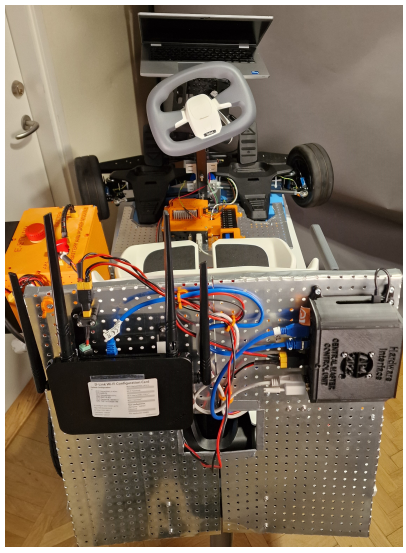
(a) Front view on the left side of AP4.

(b) Front right on the right side of AP4.

Figure 7.1: Front view of Ap4 including laptop holder, front wing and an overview of the complete system.

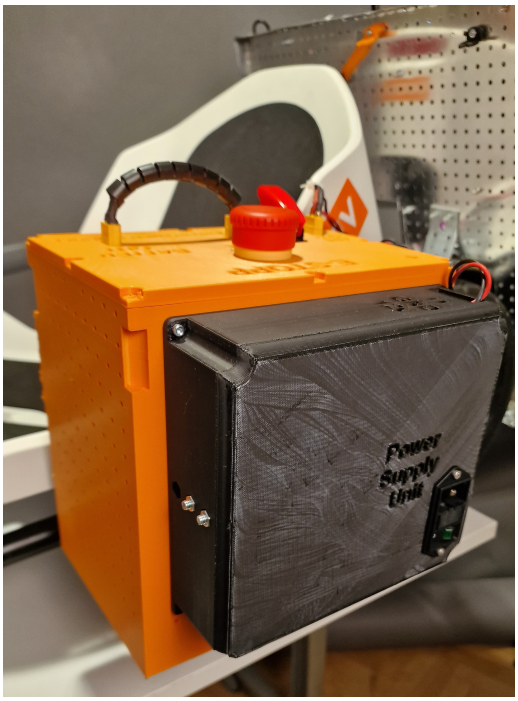


(a) Back view of AP4, focusing on the HWI outlets.

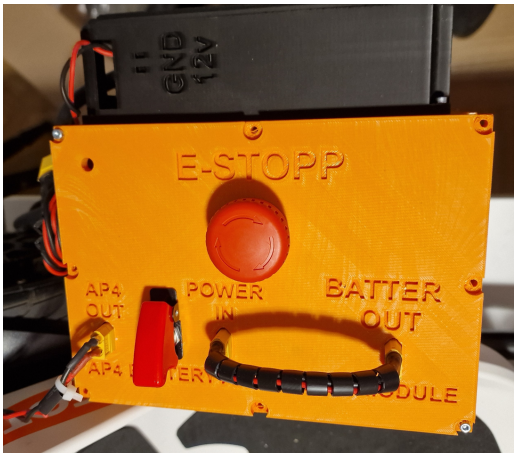


(b) Back view of AP4, with focus on the router and HWI unit.

Figure 7.2: Back view of Ap4 including its components mounted such as HWI and router.

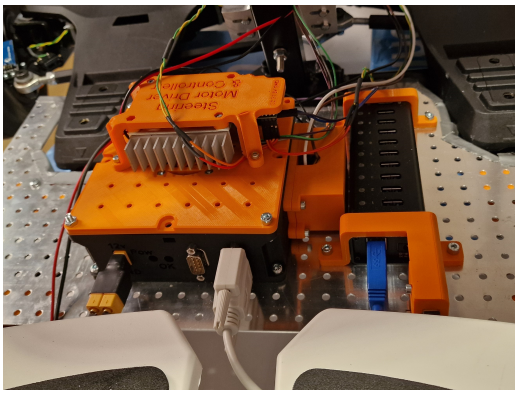


(a) Side view of power module with PSU mounted on the side.

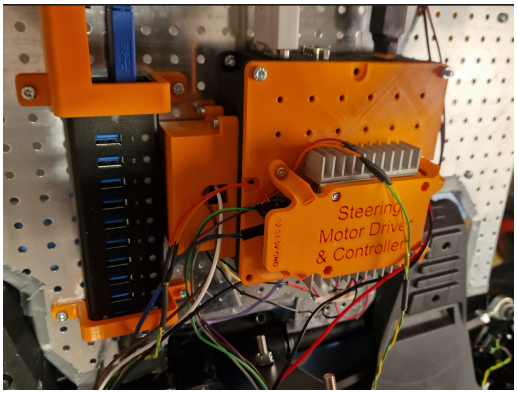


(b) View over power module's interface, killswitch- and power on buttons.

Figure 7.3: Power Module and its corresponding user interface and modules to power the AP4 with a PSU instead.



(a) View of the outports of the SPCU.



(b) Top view of the SPCU, with the steering module on top and the propulsion module on its side.

Figure 7.4: The implemented SPCU and its corresponding module holders are based on the generic ECU base.

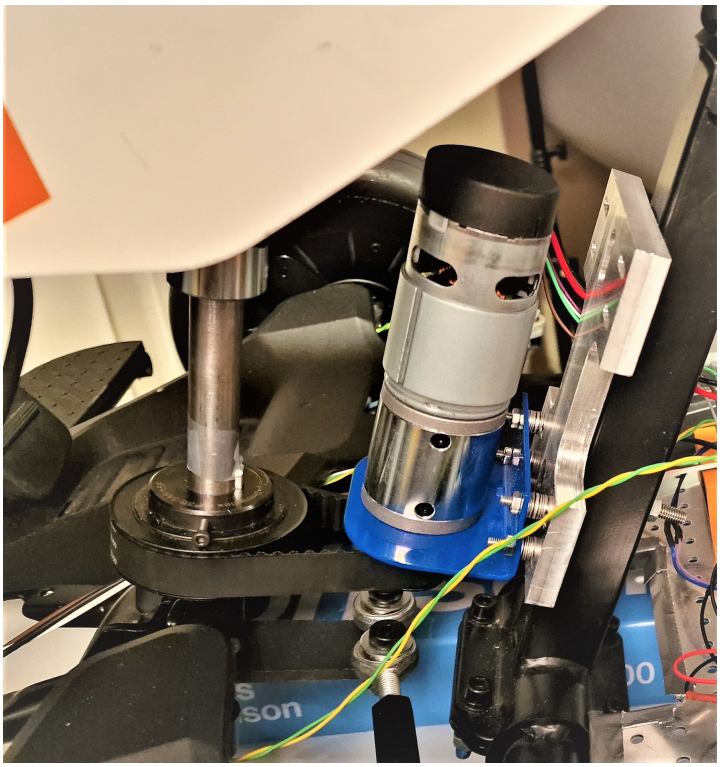


Figure 7.5: Steering system of AP4 which consist of a steering DC motor and the tooth pulley system that is connected to the steering rods of the go-kart conversion kit.

7.3 Images from test day session

This section illustrates pictures from the test day conducted to verify the requirements, comparing specifications with actual measurements and tests of AP4.

Figure 7.6: AP4 around the track, AP4 is controlled through a handheld Xbox controller, enabling drive-by-wire system. Complete a lap around the track on Gokart Centralen, Kungälv.

Figure 7.7: Driving AP4 on a parking lot, when measuring the brake distance of AP4, utilizing the known distance of each parking spot.

Figure 7.8: AP4 taking the apex, from drive session of driving the complete track of go-kart Centralen, Kungälv

7.4 Digital twin of autonomous platform

By using the ROS2 framework one can take advantage of its seamless integration with the Gazebo physics simulator. By choosing and configuring Gazebo plugins, the common data streams can be mapped to Gazebo plugins in order to interface with the simulation software. Together with the software switch, this enables the high-level software control algorithms to control the digital twin. There exist many readily available gazebo plugins that can expand the digital twin capability in the future. As of now, two gazebo plugins are used. An Ackermann steering plugin, which controls the autonomous platform movement inside the simulation, provides means to control the platform and get the state of the platform back onto ROS topics. There also exists a camera plugin. Which simulates a mounted camera on the platform.

In order to represent the autonomous platform hardware, its kinematics, and dynamics a simple URDF file has been created. This file describes how the autonomous platform is driven and how the steering geometry is set up. The URDF file can also be configured to define sensors that are mounted on the platform and how gazebo plugins should simulate them. In summary, the digital twin can be controlled using the same commands as the physical twin.

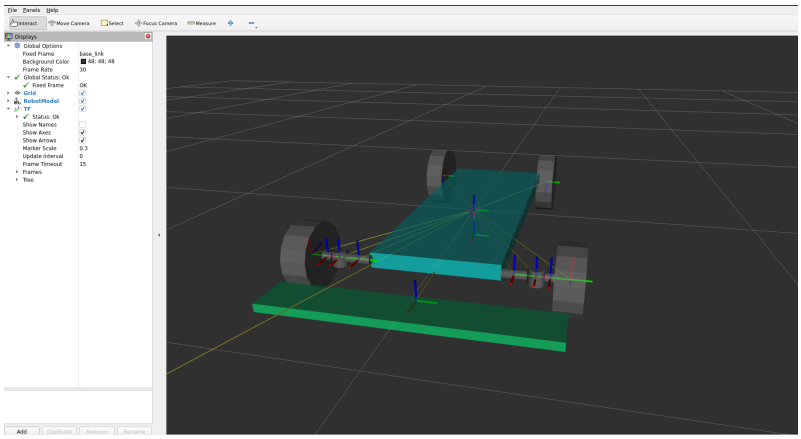


Figure 7.9: Digital twin, visualized in Rviz. Which consist of the back wheels which are moving the platform forwards, a body plate, and the front wheels with Ackermann steering geometry.

7.5 Autonomous platform 4th generation capabilities

In table, Table 7.7, the driving characteristics of AP4 and software versions are described. Other characteristics of the AP4 include a modular and flexible design that enables the developers to quickly change and implement new hardware. The low-level functionalities in the central master computer control the fundamental dynamics of the AP4 such as steering and propulsion. Thus the base for further development is constructed, by having minimal changes in higher-level functionalities. Furthermore, each ECU has internal run-time tests to verify that the given commands are within limits and that the communication to modules has not been dropped. If an error is encountered an error frame will be sent to the central master computer.

Table 7.7: Technical specification on the capabilities of AP4 regarding its driving characteristics, software and hardware compatibilities.

Capabilities			
Domain	**Function**	**Descprition**	**Unit**
Driving Characteristics	Maximum Velocity	When ninebot is selected to be in sport mode, in the app	24 [km/h]
	Reverse Velocity	When ninebot gokart is in reverse mode by pressing brake two times	3 [km/h]
	Motor Power	Rated power of the Segways motors	2x400 [W]
	Turning Radius	Radius with AP4 steering system	3 [m]
	Range	The range of the Segway	15 [km]
Communication	CAN	Main communication between ECUs and Central Master Computer	1 Mbit/s
	Ethernet	Communication between the two units of the central computer	1000 Mbit/s
	wifi	Possibility to connect remotely	can vary
Softwares of AP4	ROS2-Humble	Is the middleware/framework where highlevel control is located	-
	Docker-23.03.3	Containerized software where the SW of AP4 is running on	-
	Ubuntu-22.04	Common operating system for SW developing	-
	C++11	Lowlevel SW of each ECU is written in Arduino Framework, which is based o C++.	-
Hardware Capabilities	Battery runtime	The time for operating the AP4s components	3 [h]

8

Discussion

This chapter will discuss the requirements for the autonomous platform itself, the verification results, how well the performance is, choices made during development, and possible future research and development of the autonomous platform project.

8.1 Centralized E/E architecture: key insights from design and development

The selection of a centralized E/E architecture has shown to be suitable and appropriate. Centralized E/E architecture allows for shorter development times and higher flexibility for future functionalities. The shorter development times are based on the construction of the architecture itself by using a centralized HPC unit that conducts all the higher-level computations and decisions and the ECUs for low-level software interfaces to enable sensor readings and actuation outputs. Thus during new software development and releases, a minimum number of units will be affected namely mostly the central master computer thereby allowing the possibility to develop more computationally costly systems. In contrast to a decentralized architecture where several ECUs together constitute higher-level algorithms and functions. Moreover during software updates and releases several ECUs are affected and compatibility must be verified, making it far more complex to work on.

The centralized E/E architecture has enabled software to be built in layers, from high-level to actuator controls. This has made the software structure intuitive and easy to follow. Meaning that the project can continue to be developed for a long time. The modularity in software layers also leaves a lot of space for future improvements, layers can be upgraded later on to increase functionality, complex dependencies of functionalities can be avoided, and to implement a single function, only a specific part of the software needs to be changed. Doing the same in a decentralized architecture means that several ECU software need to be upgraded, separately, which greatly increases the chance of breaking dependencies and existing functionality. Furthermore, it also takes a longer time. Moreover, by sending sensor data to the high-level software, it can be processed using high-performance hardware and very complex algorithms, doing this in a decentralized system would affect the performance of ECUs. As long as the communication between hardware-interfacing ECUs

and high-level software is sufficient, this has proved to be a good pipeline for the autonomous platform project. As the trend within the automotive industry is to integrate more and more functionality into vehicles, there has to be computational power to process all the data. A centralized architecture shows great promise to solve the issues present in a decentralized architecture.

When it comes to practical use within the automotive industry, the systems used in commercial vehicles have several orders of magnitude greater requirements on software robustness and safety criticality compared to the autonomous platform project. This means the specific solutions proposed in this thesis may not be applicable to an autonomous vehicle.

8.2 Validation of requirements

As described in subsection 3.2.2 previous generations suffer from several problems regarding usability and flexibility. Thus the requirements regarding modularity, flexibility, and scalability of the new platform meet the wishes of Infotiv, thereby the new platform will facilitate further development in a more standardized and easier way. Furthermore, the requirements of following industry standards will both make the platform more accurate to a real vehicle and simplify the process for laymen to understand the new platform. However, the driver safety requirements are quite vague compared to a real vehicle that must follow many safety standards and safety-critical protocols to be allowed on the roads. These types of hard and distinct safety requirements are not applicable to a miniature automotive platform, since the platform is a tool for developing proof-of-concept technologies and will only be used in confined and controlled environments. The requirements regarding vehicle driving characteristics and power supply are rational for its area of use, thereby the propulsion and steering requirements will compose the new generation with fundamental capabilities, to control its direction and velocity. Having the option to select the power supply offers flexibility. Using the battery will make the AP4 mobile and thus possible to test and verify AD systems running the vehicle. Having a stationary power supply unit will allow testing the AP4 stationary in the lab environment thus lengthening the life cycle of the battery.

The requirement for choosing centralized E/E architecture is based on the latest trends in the automotive industry. Thus the AP4 will be up to date and allow to development of the latest technology. Furthermore having a centralized E/E architecture as a requirement has influenced other requirements such as hardware requirements. In such a way that for example, the computing unit has the possibility to implement a centralized E/E architecture with a central high-performance computer.

8.3 Results from verification

The verification results regarding the battery voltage limits do not meet the required specifications. Since the voltage level varies between 12.62 V at idle and 11.2 V at full load. The exceeding overvoltage that differs from the nominal voltage of 12.0 V, is not severe. Since the chosen hardware circuit boards contain built-in over-voltage protection, the relatively low overvoltage is not critical. Furthermore, the system is designed to be able to handle small overvoltages using a voltage regulator, to protect the embedded systems. However the voltage drop of 11.2 V that occurs at full load, a person sitting when steering moves along the complete range of motion. The reason behind this is the steering motor that requires far more current to overcome the increased friction from the extra weight and change the steering angle. This can be seen in Table 7.3, where the DC motor consumes roughly 4 A more than the case of no load. This will result in higher current output from the battery and lower voltage for a short time period. The increased current might cause problems, such as the fuses being blown. Something was experienced during the test day at Go-kart Centralen when conducting the steering test with a full load, where a 15A fuse of the battery was blown. The same test was conducted at Infotiv's office with no fuses being blown, so the only factor is the flooring characteristics. The go-kart track is made to create more friction in order to allow the go-karts to turn at higher speeds whilst the office floor is made out of laminated wood. The friction constant of the go-kart Centralens floor is assumed to be so much larger, requiring far more torque in the motor to overcome the increased friction, since torque is proportional to current an increased current spike will be observed. Furthermore, the battery's energy capacity is far enough to be able to power AP4 for over 5 hours during mixed driving. Thus meeting the requirements and desires with a good result.

The steering specification of having a margin of 20% or less is not met. As can be seen in Table 7.4 there seems to be an offset of 5 degrees, this might be some errors from a bad calibration. Another source of error is the not-so-scientific measurement method using a protractor. Despite the results, the steering accuracy can be seen as accurate enough for current use and future use. The implemented drive-by-wire systems use a reference from a handhold controller and a person who will act as a feedback controller that will compensate for the control error. Some AD and ADAS systems have approximately the same approach, where the steering request is a normalized value between maximum and minimum steering angles, the feedback comes from some sort of localization function that requires more or less steering angle.

The minimum propulsion specification is completely met as described in Table 7.1. It is possible to both move forward and enable the reverse mode in order to go backward. The longest braking distance of 2 meters is within the limits. One important requirement being that the physical pedals work in parallel with AP4 was also verified. However, due to time restrictions and some connection problems when conducting tests at Go-kart Centralen it was not possible to measure different velocities. Measuring the set reference voltage and corresponding velocity of the

vehicle could make it possible to model the complete propulsion system that now acts as a black box. Having a complete and accurate model, it could be integrated into the digital twin and also optimize a future velocity feedback controller, preferably a PID to minimize any velocity error.

The Computing unit requirements are completely met with the chosen hardware of a Raspberry Pi 4b as seen in Table 7.5. One important note is that the central master computer is built up of two different units where the high-level controlled unit can be easily exchanged due to the containerized software. For the moment, a laptop is far enough to meet the capabilities of a drive-by-wire system.

The specification regarding the modularity in both software and hardware is fulfilled. Hardware-wise it can be seen in section 5.4 how the system has been designed in order to fulfill modularity, scalability, and usability. The physical result can be seen in section 7.2, which is built upon the standards of the project. Having a standardized design increases usability and eases the understanding of the complete system. Due to the rectangular pattern of mounting holes, it is possible to change the physical layout relatively easily, by just screwing and unscrewing bolts. Furthermore, the system also has the possibility in the same way add completely new hardware components on top of the mounting plates or the already mounted casings that have the same hole pattern. Scalability is achieved by having the general ECU bases to build specific functionalities on. With the same reasoning modularity in software has also been achieved in the structured coding bases for ECUs. In high-level software, modularity has been achieved by using ROS2 packages and nodes to have the high flexibility and modularity to add or remove said nodes or packages without breaking dependencies. The implemented *CAN signals to ROS2 Topic Converter* standardizes a pipeline to communicate from the central master computer and to the rest of the components of the platform. Thus the communication is very flexible for adding new CAN frames without software dependencies in other packages.

8.4 Choices made during development

This section discusses the choices that were made during development regarding hardware and software.

8.4.1 Hardware

The selection of hardware components such as the Raspberry Pi 4b, Bluepill, and CAN transceivers MCP4725 were choosen as this is a prototype and the component choices may change in future. Furthermore, the selection of hardware components is off-the-shelf boards, thereby having the advantage of being well documented as well as belonging to the open-source software libraries that could be utilized. This shortens development time, as only higher embedded software is needed to be developed. Lastly, the selection of lead-acid batteries is based on the far more stable characteristics of usage. Even the small risk of having an accident or failure of a lithium battery can result in catastrophic consequences. Thus there exist far more

safety regulations and handling regarding lithium batteries that can thus be avoided.

By reusing the steering system and components from AP3 a lot of time and effort could be saved. The steering motor and system of pulley system in AP3 were analyzed in subsection 3.2.2, the conclusion being made that it worked almost seamlessly but it had some loose pulleys. Since the previous thesis on AP3 made spare parts for steering these were selected to be used on AP4 as well. However, the loose pulleys were taken care of using tighter screws and a thread locker.

The propulsion of the AP4 could be solved in two distinct ways either by imitating the hall sensors of the pedals or by decoding and integrating them into the internal communication system of the go-kart kit. The latter is way too time-consuming and deemed almost impossible without having the correct protocol, which was concluded in subsection 3.2.2. However, one drawback of using the communication approach would be that the pedal readings would be overwritten and ignored. Since the segway motors would get their commands from the ECU mimicking the communication of the go-kart system. So the choice was made to connect the voltage source parallel to the pedals. Thus also having the benefit of the pedals still being operative and working in parallel to the ECU. Thereby the braking pedal will always have priority and brake the AP4 regardless of the throttle voltage. Another design choice of having the seat still mounted raised several questions and discussions. Removing the seat would free up a lot of space that could be used for hardware placement instead. Thereby also achieving a more geometric central point of mass, with the battery and computers placed where the seat is mounted. However, the advantage that weighed up against the removal, was the possibility of still having a person sitting during movement. Thus the driver could hit the brake if necessary or the emergency killswitch if some electrical failure occurs.

8.4.2 Software

Software that is used on autonomous drive vehicles needs to be very robust, time-critical, and may never cause undesired behavior. Autonomous platform generation 4 can be seen as a testbed and developing platform in that regard and does not have the same robustness requirement. The platform will not be harmful to nearby humans if something goes wrong with the software due to its physical size and low velocity. This allows the software development and its design to be more flexible compared to what will be used on a commercial vehicles. Meaning that early stages of AD and ADAS algorithms can be tested on a small platform to verify its principles and strategies.

The software on an autonomous platform should be flexible, in terms of functionality and how algorithms are implemented. It should be easy to try out different control algorithms, autonomous drive functionality, and sensor processing. Without having to restructure the whole codebase. By splitting the software into three distinct blocks, it adds three layers of abstraction to the software. Functions implemented in one layer should not affect the behavior of functions from another layer. This

creates a robust system where changes can be made to the system without breaking it. An underlying component of any software is the communication and transfer of data between algorithms and control algorithms. On an autonomous platform, ROS2 is used to accomplish this.

The framework and middleware on which the high and low-level software functionality is built on the Robot Operating System 2. This framework makes it possible to design software in such a way that it is very modular and robust. The exchange of data and information between different software components is therefore guaranteed to be reliable and on time. It is very easy to use and pass data created in one part of the system to a different software component using topics, services, or actions. The description of ROS2 and its capabilities can be found in subsection 2.3.2. By configuring the framework and the host operating system it is guaranteed to be real-time capable, making it very easy for a developer to create software that needs to meet hard-time requirements. By using ROS2 and its APIs, many of the complex software functions such as data transfers are already handled by the framework. This gets exposed to the developers by using topics.

Both the high-level and low-level code blocks are placed inside Docker containers. As presented in subsection 2.3.1, using containers does not add affectable performance overhead or timing overhead. The small loss of performance is outweighed by the portability of the software created. In practice, this also allows for external remote access to the connected software system by connecting yet another container running the robot operating system. Since the containers are set up, a secondary high-performance computer can open and run the digital twin whilst a secondary container controls the autonomous platform hardware. Containerization also makes the software very portable, as containers in theory can be deployed on any hardware supporting it. For the lower-level software, this would not make sense though, as the software inside it has been configured to run on a Raspberry Pi. But it is a feature for the higher-level control software container. Several developers can work on different parts of the software at the same time and test it using the digital twin located in the higher-level software container. To test new algorithms, the container on a development laptop would simply need to be connected with the lower-level software container over wifi or ethernet.

8.5 Future work and research

This section will discuss possible future work and continuation on autonomous platform generation 4. Firstly the shortcomings of the platform should be addressed and solved and then functionality could be expanded as a robust modular HW and SW base has been created so far.

8.5.1 Digital twin

Although the digital twin can be controlled using the same commands as the physical platform, it needs some development regarding its driving dynamics and kinematics

to be more realistic. The Gazebo plugin used to control it, called Ackermann drive, needs to be tuned properly. It would also be possible to adjust dynamic behavior by tuning parameters for the go-kart chassis inside the URDF file. Inserting sensors into the digital twin is trivial, but making the digital twin act in the same way as the physical twin is not.Because the dynamics and kinematics are not properly tuned according the to the physical platform. As presented in theory, many digital twins fail due to not representing the proper driving dynamics of the physical twin.

Being able to use the same high-level control of both digital twin and physical twin shows great potential in being useful for future work. The control software can be tested and verified in a simulated environment before being tested on actual hardware, in turn saving a lot of development time.

8.5.2 Propulsion and steering of autonomous platform

Propulsion for the autonomous platform is a vital component and sets the base functionality to which future functionality can be added. Without precise propulsion control, it would not be relevant to implement further sensors or AD functions. As it is now, the autonomous platform has rudimentary propulsion control. It can drive forward with a velocity between 0 and 8 km per hour. However since it is currently controlled through an open-loop controller, there is no guarantee that the requested velocity in the software will be reached by the propulsion motor. By guaranteeing that a requested velocity will be met future research on this platform can trust its behavior more. Therefore, an important future work would be to implement a feedback propulsion controller. The hardware for this functionality is mounted on the platform, but due to time restraints, it could not be implemented in software for this thesis.

The steering control works very well, even if only a simplified open-loop controller is implemented inside the low-level software. Requested steering angles can be set reliably since the DC motor mounted to the steering shaft has internal encoders and it has closed-loop capabilities on the motor driver board already.

As it is implemented right now, the steering and propulsion control systems are decoupled. Meaning they are controlled independently. As a proof of concept, this worked sufficiently, but it does not reflect a bicycle or Ackermann kinematic model which is often used to model vehicle behavior. For example, as it is implemented now, a requested turning rate for the platform is converted into a steering angle directly, without taking into account the physical dimensions. An interesting area in regard to this and future work would be to model the Ackermann steering and propulsion behavior and implement a suitable controller for it. Investigating other useful sensors with regard to vehicle dynamics would also be useful, i.e. using an Inertial Measuring Unit together with the wheel speed sensors to detect in which direction the platform is moving. The more accurately the movement of the platform can be described, the more complex trajectories can be followed.

8.5.3 Sensors on autonomous platform

Due to time constraints, the only sensor fully integrated into the platform was a web camera. Future work and research in this area would be integrating further sensors used in autonomous driving. As stated in subsection 2.2.2 the most common sensors are radar, lidar, and ultrasonic. Cost-effective versions suitable to use on an autonomous platform would need to be found. In order to implement these in the digital twin simulation environment the specification of the sensors would need to be verified by doing some tests on the hardware chosen to mount on the platform. Another interesting sensor technology to integrate into AP4 would be a 3D lidar.

8.5.4 Battery

The voltage dropped very low whilst turning the steering shaft when an operator was sitting on the platform and the 15 Amp fuse broke. It is therefore clear that the power management system of the autonomous platform needs to be further developed in order to satisfy all requirements and specifications.

A more thorough power consumption analysis needs to be performed, according to the specifications of the steering shaft motor it could draw up to 13 Amps of current, yet it broke a 15A fuse during the test day. The surface on which the platform is driven on also plays a role in power consumption, the same test performed on a smooth office floor did not break the fuse. To get further insight into this one could implement continuous voltage and current draw monitoring in software on the autonomous platform. The hardware is there, but due to time constraints, it could not be implemented in software. A possible solution to the voltage dropping well below the specification could be to implement a voltage regulator in series after the battery module.

8.5.5 Autonomous driving functionality

There are many possibilities for future work in the area of integration of different autonomous driving functions. By integrating an existing autonomous driving functionality with the platform one would verify the platform's ability to be modular in software, as well as test the ease of integration. There are many open source alternatives that can be tried out. One limitation is that the choosen AD algorithm has to be compatible with the software of AP4. Furthermore the hardware requirements to enable AD has to be implemented and verified on the platform first.

9

Conclusion

The purpose of this thesis is to develop an autonomous platform with a centralized E/E architecture to implement and test AD and ADAS systems. Using a centralized E/E architecture enables modularity, scalability, and flexibility both hardware- and software-wise. A centralized E/E architecture allows for more flexibility and more computational capacity by having a high-performance computational unit that performs all high-level and complex calculations and low-level ECUs to handle sensor information and actuator control. Another key advantage of centralized is the lowered complexity of software since it can be vertically integrated in-house. Furthermore, a centralized architecture allows for the possibility of designing generic ECU bases for the low-level units, which is a well-suited foundation for plug-and-play software and hardware development.

The developed platform includes the design of a centralized architecture using a high-computational unit and low-level ECUs. The communication between the units is accomplished by using the standard ROS2. Furthermore, a framework using containerized software (docker) is to offer modularity and flexibility in software design. The platform also includes the design of a modular and flexible mounting system for hardware components, through the use of premade aluminum plates with holes. The holes can used to mount different casings, containing circuit boards, ECUs, etc. This allows an easy change of the physical layout, mounting new casings, or even removing parts without any strict dependencies. Furthermore, the usage of standardized outlets further improves the scalability of the system, since it is possible to connect in parallel with premade cables. The complete system architecture has successfully been implemented on a miniature vehicle platform.

Apart from the physical design, a digital twin of the vehicle platform has been developed. The digital twin uses the built-in functionalities in ROS2, which offers seamless integration with the physics simulator Gazebo. Even though a mere simple digital twin is implemented, there is great potential to make it represent the physical platform driving dynamics in future work. A properly configured digital twin has many benefits, such as faster testing of implemented functionality. In this thesis, it was proved possible to use the same high-level control commands on the digital twin as for the physical platform.

The autonomous platform project can be seen as a work in progress since the end goal is subjective by having a self-driving system. The performance can vary and the methods used can also be varied or optimized along the way. This includes adding more sensors, actuators, and ECUs on the platform constructed within the thesis. Nonetheless, the produced platform is a solid base to conduct future research and implementation.

Bibliography

[1] Ben Lutkevich. *self-driving car (autonomous car or driverless car)*. URL: https://www.techtarget.com/searchenterpriseai/definition/driverless-car. (accessed: 06.04.2023).

[2] Jie Tang Shaoshan Liu Liyun Li. *Creating Autonomous Vehicle Systems, Second Edition*. Synthesis Lectures on Computer Science. Springer Cham, 2022. ISBN: 978-3-031-01805-3.

[3] V. Bandur et al. "Making the Case for Centralized Automotive E/E Architectures." In: *IEEE Transactions on Vehicular Technology, Vehicular Technology, IEEE Transactions on, IEEE Trans. Veh. Technol* 70.2 (2021), pp. 1230–1245. ISSN: 0018-9545.

[4] Lucas Mauser, Stefan Wagner, and Peter Ziegler. "Methodical Approach for Centralization Evaluation of Modern Automotive E/E Architectures." In: (2022).

[5] Hamid Ebadi. *Infotiv Autonomous Platform*. Accessed on 25.04.2023. 2022. URL: https://infotiv-research.github.io/gokart-documentation/.

[6] Samuel Davis. *Managing Electric Vehicle Power*. SAE International, 2020. ISBN: 978-1-4686-0144-2.

[7] Miroslaw Staron. *Automotive Software Architectures. [electronic resource] : An Introduction.* Springer International Publishing, 2017. ISBN: 9783319586106. (accessed: 16.05.2023).

[8] Matt Markel. "1.1.3 Autonomous Driving Levels." In: *Radar for Fully Autonomous Driving*. Artech House, 2022. Chap. 1.1.3 Autonomous Driving Levels, p. 4. ISBN: 978-1-5231-4576-8.

[9] Jessica Van Brummelen et al. "Autonomous vehicle perception: The technology of today and tomorrow." In: *Transportation Research Part C* 89 (2018), pp. 384–406. ISSN: 0968-090X.

[10] Gustavo Velasco-Hernandez et al. "Autonomous Driving Architectures, Perception and Data Fusion: A Review." In: *2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP), Intelligent Computer Communication and Processing (ICCP), 2020 IEEE 16th International Conference on* (2020), pp. 315–321. ISSN: 978-1-7281-9080-8.

[11] Fatima M. Abd-Alrasool, Ahmed A. Al-Moadhen, and Haider G. Kamil. "Intelligent Control for Obstacle Avoidance in the Self-Driving Car." In: *2022 International Conference on Data Science and Intelligent Computing (ICDSIC), Data Science and Intelligent Computing (ICDSIC), 2022 International Conference on* (2022), pp. 141–146. ISSN: 979-8-3503-3488-3.

[12] X. He et al. "Robust Lane Change Decision Making for Autonomous Vehicles: An Observation Adversarial Reinforcement Learning Approach." In: *IEEE*

Transactions on Intelligent Vehicles, Intelligent Vehicles, IEEE Transactions on, IEEE Trans. Intell. Veh 8.1 (2023), pp. 184–193. ISSN: 2379-8858.

[13] Chinmay Vilas Samak, Tanmay Vilas Samak, and Sivanathan Kandhasamy. "Control Strategies for Autonomous Vehicles." In: (2020).

[14] "Comparative study of Automotive Sensor technologies used for Unmanned Driving." In: (2021), pp. 346–350.

[15] *IBM What is containerization?* Accessed on May 17th, 2023. URL: https://www.ibm.com/topics/containerization.

[16] Cliff Saran. "How containerisation helps Volkswagen develop advanced in-vehicle software." In: *Computer Weekly* (2021), pp. 8–10. ISSN: 00104787.

[17] Md Sadun Haq, Ali Saman Tosun, and Turgay Korkmaz. "Security Analysis of Docker Containers for ARM Architecture." In: *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC), Edge Computing (SEC), 2022 IEEE/ACM 7th Symposium on, SEC* (2022), pp. 224–236. ISSN: 978-1-6654-8611-8.

[18] M. Sollfrank et al. "Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation." In: *IEEE Transactions on Industrial Informatics, Industrial Informatics, IEEE Transactions on, IEEE Trans. Ind. Inf* 17.5 (2021), pp. 3566–3576. ISSN: 1551-3203.

[19] Sushant Chamoli and Varsha Mittal. "Docker Networking: A Security Review." In: *2023 7th International Conference on Trends in Electronics and Informatics (ICOEI), Trends in Electronics and Informatics (ICOEI), 2023 7th International Conference on* (2023), pp. 624–629. ISSN: 979-8-3503-9728-4.

[20] Andre-Marcel Hellmund et al. "Robot operating system: A modular software framework for automated driving." In: *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), Intelligent Transportation Systems (ITSC), 2016 IEEE 19th International Conference on* (2016), pp. 1564–1570. ISSN: 978-1-5090-1889-5.

[21] Max Aartsen et al. "Analyzing Interoperability and Security Overhead of ROS2 DDS Middleware." In: *2022 30th Mediterranean Conference on Control and Automation (MED), Control and Automation (MED), 2022 30th Mediterranean Conference on* (2022), pp. 976–981. ISSN: 978-1-6654-0673-4.

[22] Shunki Shibuya et al. "Seamless Rapid Prototyping with Docker Container for Mobile Robot Development." In: *2022 61st Annual Conference of the Society of Instrument and Control Engineers (SICE), Instrument and Control Engineers (SICE), 2022 61st Annual Conference of the Society of* (2022), pp. 1063–1068. ISSN: 978-4-9077-6478-4.

[23] Jacqueline Henle et al. "Architecture platforms for future vehicles: a comparison of ROS2 and Adaptive AUTOSAR". In: *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*. 2022, pp. 3095–3102. DOI: 10.1109/ITSC55140.2022.9921894.

[24] Siemens. *Embedded Software*. Accessed on 16.05.2023. URL: https://www.plm.automation.siemens.com/global/en/our-story/glossary/embedded-software/64121.

[25] Codeacademy Team. *What Is an IDE?* Accessed on 16.05.2023. URL: https://www.codecademy.com/article/what-is-an-ide.

[26] PlatformIO. *What is PlatformIO? PlatformIO latest documentation.* Accessed on 16.05.2023. URL: https://docs.platformio.org/en/latest/what-is-platformio.html#awards.

[27] Ben Lutkevich. *What is a Microcontroller and how does it work?* Accessed on 16.05.2023. URL: https://www.techtarget.com/iotagenda/definition/microcontroller.

[28] The Editors of Encyclopaedia Britannica. *protocol computer science.* Accessed on 11.04.2023. Feb. 2022. URL: https://www.britannica.com/technology/protocol-computer-science.

[29] Swaroop CODREY. *What is Serial Communication and How it works?* Accessed on 11.04.2023. Nov. 2018. URL: https://www.codrey.com/embedded-systems/serial-communication-basics/.

[30] Scott Campbell. *BASICS OF THE I2C COMMUNICATION PROTOCOL.* Accessed on 19.04.2023. 2016. URL: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/#:~:text=I2C%5C%20is%5C%20a%5C%20serial%5C%20communication,always%5C%20controlled%5C%20by%5C%20the%5C%20master..

[31] Scott Campbell. *BASICS OF THE SPI COMMUNICATION PROTOCOL.* Accessed on 20.04.2023. 2016. URL: https://www.circuitbasics.com/basics-of-the-spi-communication-protocol.

[32] Campbell Scott. *Basics of UART Communication.* Accessed on 19.06.2023. 2016. URL: https://www.circuitbasics.com/basics-uart-communication/.

[33] C. Schwarz and Z. Wang. "The Role of Digital Twins in Connected and Automated Vehicles." In: *IEEE Intelligent Transportation Systems Magazine, Intelligent Transportation Systems Magazine, IEEE, IEEE Intell. Transport. Syst. Mag* 14.6 (2022), pp. 41–51. ISSN: 1939-1390.

[34] Markus Borg et al. "Digital Twins Are Not Monozygotic – Cross-Replicating ADAS Testing in Two Industry-Grade Automotive Simulators." In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), Software Testing, Verification and Validation (ICST), 2021 14th IEEE Conference on, ICST* (2021), pp. 383–393. ISSN: 978-1-7281-6836-4.

[35] Gill Lumer-Klabbers et al. "Towards a Digital Twin Framework for Autonomous Robots." In: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), Annual Computers, Software, and Applications Conference (COMPSAC), 2021 IEEE 45th, COMPSAC* (2021), pp. 1254–1259. ISSN: 978-1-6654-2463-9.

[36] Andrew Farley, Jie Wang, and Joshua A. Marshall. "How to pick a mobile robot simulator: A quantitative comparison of CoppeliaSim, Gazebo, MORSE and Webots with a focus on accuracy of motion." In: *Simulation Modelling Practice and Theory* 120 (2022). ISSN: 1569-190X.

[37] Martina Barbi et al. "Simulation-based Digital Twin for 5G Connected Automated and Autonomous Vehicles." In: *2022 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit), Networks and Communications & 6G Summit (EuCNC/6G Summit), 2022 Joint European Conference on* (2022), pp. 273–278. ISSN: 978-1-6654-9871-5.

[38] Anton Rassolkin et al. “Digital twin for propulsion drive of autonomous electric vehicle.” In: *2019 IEEE 60th International Scientific Conference on Power and Electrical Engineering of Riga Technical University (RTUCON), Power and Electrical Engineering of Riga Technical University (RTUCON), 2019 IEEE 60th International Scientific Conference on* (2019), pp. 1–4. ISSN: 978-1-7281-3942-5.

[39] Daniella Tola and Peter Corke. “Understanding URDF: A Survey Based on User Experience.” In: (2023).

[40] Open Robotics. *Official Documentation ROS urdf/XML/joint*. [Online]. Accessed 23/5 2023. URL: `http://wiki.ros.org/urdf/XML/link`.

[41] Open Robotics. *Official Documentation ROS urdf/XML/joint*. [Online]. Accessed 23/5 2023. URL: `http://wiki.ros.org/urdf/XML/joint`.

[42] N. Koenig and A. Howard. “Design and use paradigms for Gazebo, an open-source multi-robot simulator.” In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566), Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on, Intelligent robots and systems* 3 (2004), p. 2149. ISSN: 0-7803-8463-6.

[43] Jonathan Cacace, Nicola Mimmo, and Lorenzo Marconi. “A ROS Gazebo plugin to simulate ARVA sensors.” In: *2020 IEEE International Conference on Robotics and Automation (ICRA), Robotics and Automation (ICRA), 2020 IEEE International Conference on* (2020), pp. 7233–7239. ISSN: 978-1-7281-7395-5.

[44] Open Source Robotics Funcdation. *Tutorial: Using Gazebo plugins with ROS*. [Online]. Version: 1.9, Accessed 6/5 2023. URL: `https://classic.gazebosim.org/tutorials?tut=ros_gzplugins`.

[45] *Protected Document for AP project*. Unpublished Internal Infotiv AB Document. Accessed 1/4 2023.

[46] *Ninebot S USER MANUAL*. Version 1.0. Segway Inc., NH, USA.

[47] *Ninebot Gokart kit USER MANUAL*. Segway Inc., NH, USA. 2018.

[48] *Kommunikation för gokart*. Unpublished Internal Infotiv AB Document. Accessed 1/4 2023.

[49] PhD in Computer Science Hamid Ebadi. *Private Conversation*. Unpublished Internal Infotiv AB Document.

[50] *Writing Requirements*. Unpublished Internal Infotiv AB Document. Accessed 1/4 2023.

[51] C. Plasberg et al. “Towards Distributed Real-Time capable Robotic Control using ROS2.” In: *2022 IEEE 18th International Conference on Automation Science and Engineering (CASE), Automation Science and Engineering (CASE), 2022 IEEE 18th International Conference on* (2022), pp. 2205–2210. ISSN: 978-1-6654-9042-9.

A

Electrical Circuits of Power Module

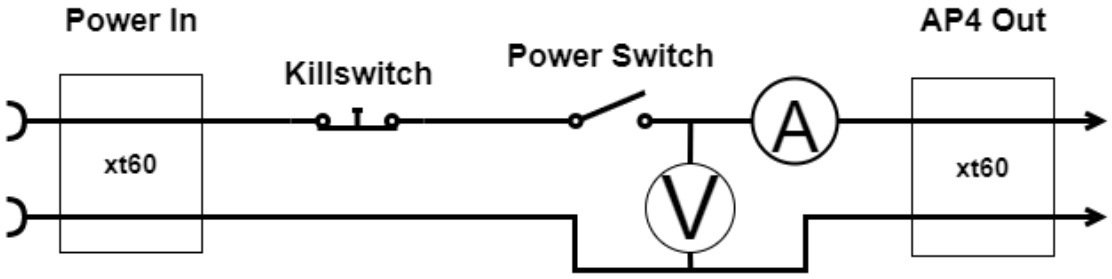


Figure A.1: *Power in* to *AP4 out* flow described in a electrical circuit. Including voltage sensor, current sensor, switches for power on and emergency stop killing all the power in AP4.

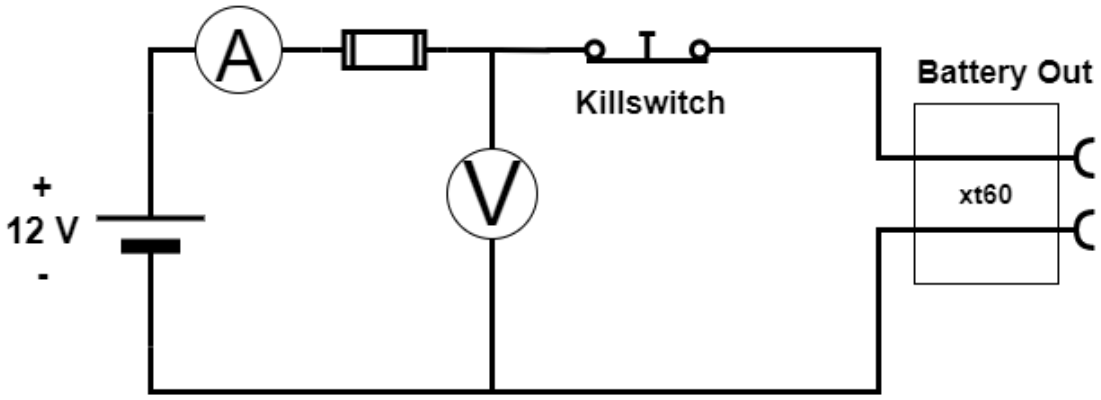


Figure A.2: Battery supply to the outlet *Battery out* flow described in a electrical circuit. Including voltage sensor, current sensor, fuse and emergency stop killing all the power in AP4.

B

Rendered Figures of Other Components

Figure B.1: Flexible wire holder to drop and click wires and cables, to manage wiring of AP4.

Figure B.2: Casing for the hardware interfacing computing unit, for the chosen component of Raspberry Pi 4b.

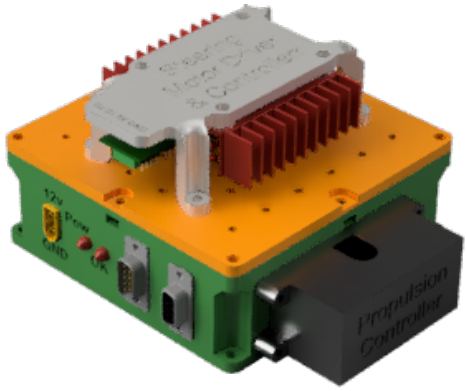
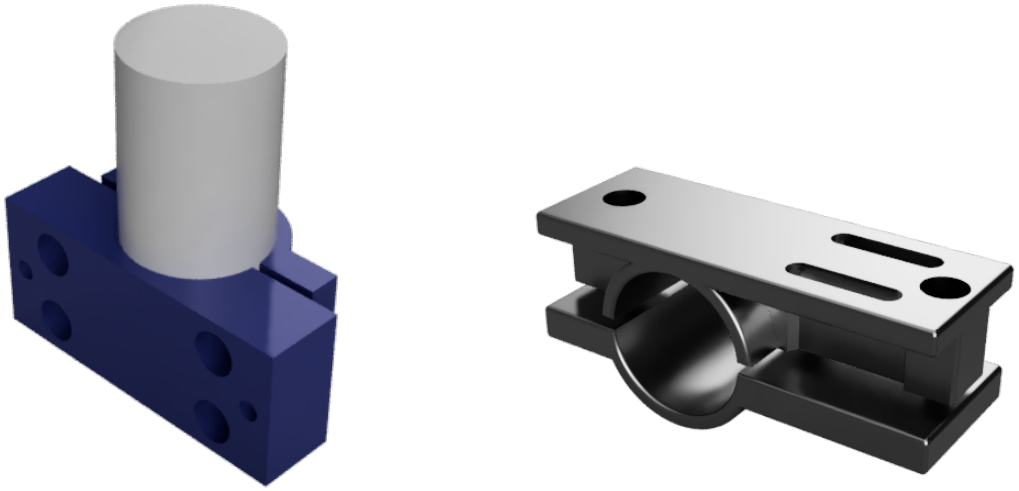


Figure B.3: Casing for SPCU, steering module (grey) and casing for propulsion, based of the generic ECU casing, figure 5.6.



(a) Back mounting plate holder **(b)** Mountingplate holder.

Figure B.4: Mountings for holding the aluminium plate.

(a) Front left wing. **(b)** Front right wing.

Figure B.5: Front wings with the standardised grid layout.

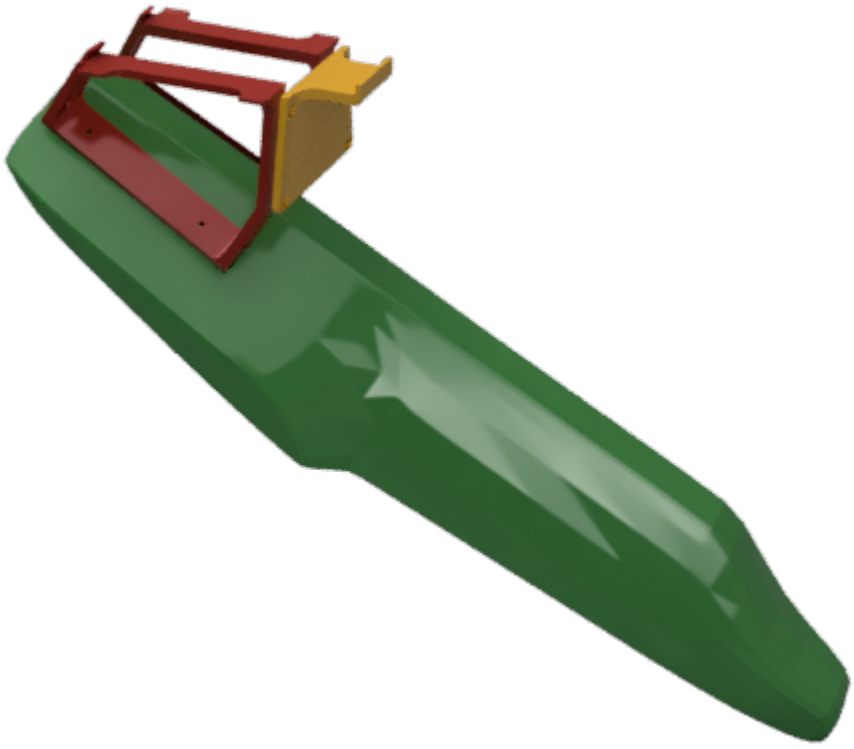


Figure B.6: Laptop holder, placed on the central panel of the go-kart.

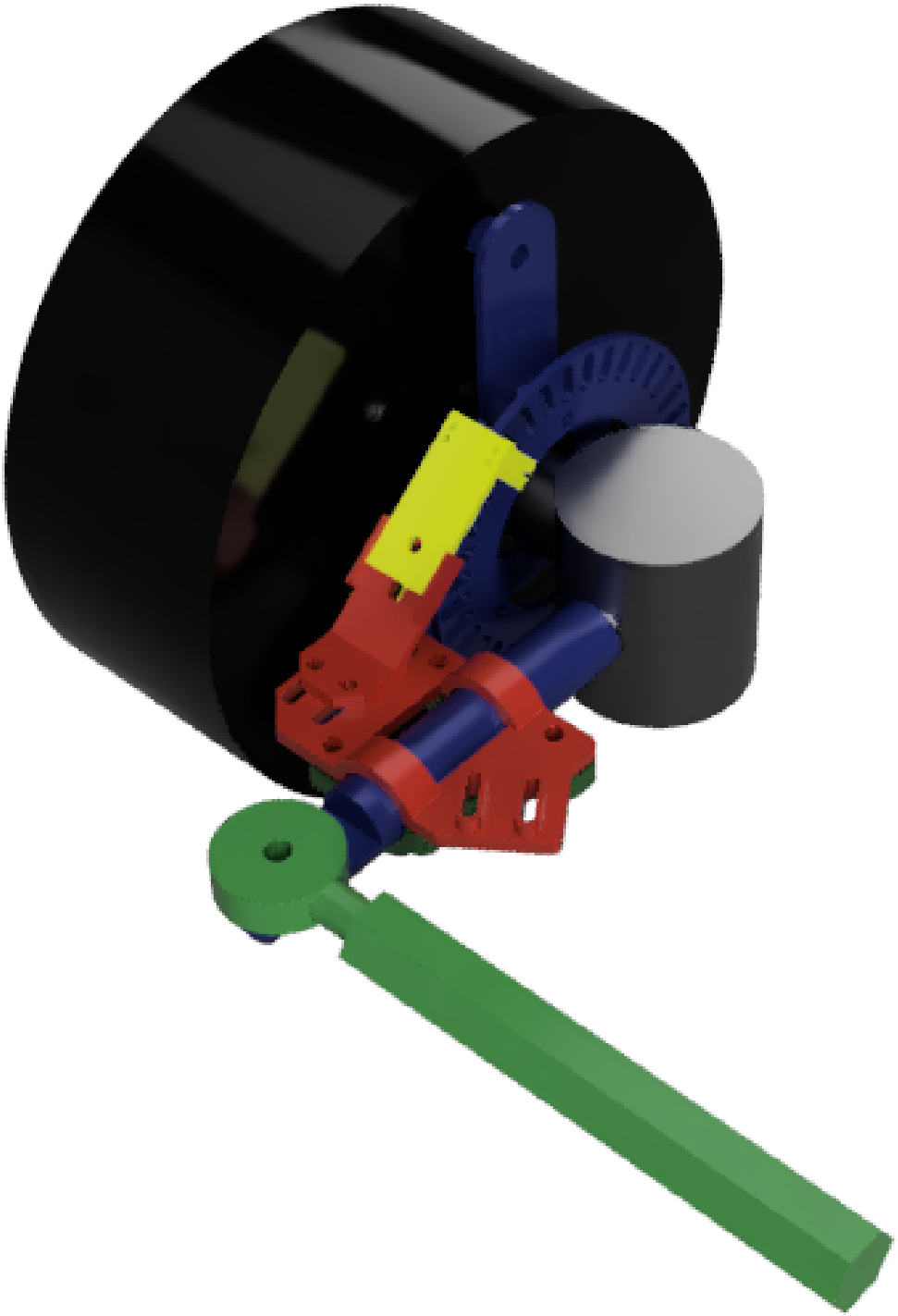


Figure B.7: Speed sensor (yellow)- and limit switch holder, placed on the gokarts steering geometry with an encoder wheel (blue).

C

Bill of Materials

This appendix describes the list of components that is mounted on the autonomous platform and their price in SEK.

BOM Autonomous Platform Generation 4

		TOTAL PRICE SEK:	16139,3
Part Name	**Quantity**	**Area of Use**	**Price (SEK)**
CENTERAL MASTER COMPUTING UNIT			
Raspberry Pi 4 Starter Kit 4 GB	1	Rasberry Pi 4b for Low Level Hardware interface	1890
LM2596 DC-DC converter	1	Power components on AP4. In voltage 3.2-46 Output voltage 1.25 - 30 3A	25
DB9 BREAKOUT BOARD Male	1	HW Node standardised DB 9 can bus connector	66
DIR-842-V2 Wifi Router	1	Wifi router / switch with 1000 MB/s ports	499
RS485 CAN HAT Raspberry Pi	1	Prebuilt CAN shield for raspberry Pi	203
Raspberry Pi 4 7 mm heatsink	1	Low profile raspberry pi cooling heatsink and fan.	120
Axial Fan DC 40x40x10mm 12V	1	Cool Down hardware interface	57
Xbox controller	1	Wireless controller and USB reciever for remote control of ap4	899
7inch IPS Capacitive Touch Screen 1024x600 Resolution LCD Display	1	Display for hardware interface raspberry Pi 4, System monitoring whilst running platform	900
Power Module			
XT60 Male	1	Connecting power to various components on autonomous platform. Max 60 Amp.	21
XT60 Female	2	Connecting power to various components on autonomous platform.	40
Current ant voltage reading sensors	1	Possibility to measure voltage of AP4 and current during runtime programatically	120
Emergency stop	2	Emergency stop if something unexpected happens, breaks power to all components except ninebot segway	274,52
Led Acid Battery	1	12 V, 30 Ah	799
Battery Charger	1	Led Acid Battery Charger IP68, 12 V, 8 A	649
Automotive grad fuse	1	Blow fuse if ftoo much power is drawn from battery	25
Fuse Holder	1	Connects fuse to battery circuit	33
C14 Contact	1	Intag, C14, 250V, Schurter, till batterimodulen	126
Din rail mounted PSU	1	90% Efficiency, 12V, 10A, 120W	875
Fuse connector for C14 outlet	1		42

BOM Autonomous Platform Generation 4

		TOTAL PRICE SEK:	16139,3
Part Name	**Quantity**	**Area of Use**	**Price (SEK)**
Generic ECU base Components			
		Quantity inside brackets are required for ONE ECU	
LM2596 DC-DC converter	3(2)	Power components on AP4. In voltage 3.2-46 Output voltage 1.25 - 30 3A	110
XT60 Male	1	Connecting power to various components on autonomous platform. Max 60 Amp. Package of 20	209
XT60 Female	1	Connecting power to various components on autonomous platform. Max 60 Amp- Package of 20.	199
XT60 splitter	(1)5	Parallell wired M-M to M Y connector for XT60 contact. Enables one to parallell wire new components on platform	550
Jumper wire F-F	6	Jumper wires bradboard, Female to Female, pack ofo 10	72
Jumper wire M-M	6	Jumper wire breadboard, Male to Male, pack of 10	78
Jumper wire M-F	6	jumper wire breaboard, Male to Female, pack of 10	78
DB9 BREAKOUT BOARD Male	9	HW Node standardised DB 9 can bus connector	596
STM32 bluepill		Ersätta dem stm32 bluepillsen vi tagit från infotivs inventarier OBS köp 5 packet	490
MCP2515 can-bus modul	5	Ersätta dem mcp2515 korten vi tagit från infotiv	151
Flätad kabel 4 x 0.14 mm²	1	Ihophäftade kablar, gör de mer neat när man drar kablar inuti HW noden. 10 Meter	89
GPIO-staplingslist 2x20	6	Connect sensors to HW Node	222
Logic Level Converter	5	Converts 5v logic levels to 3.3v logic level Used to interface stm32 bluepill (3v) to 5V CAN module	160
Axial Fan DC 40x40x10mm 12V	4	Cool down each ECU	228
LED röd	25	Show ECU power status	42,25
LED grön	25	Show node software status	42,25
Automotive grad fuse	5	Blow fuse if ftoo much power is drawn from battery	165
Fuse Holder	5	Connects fuse to battery circuit	125
ST-link v2		Program stm32f103c8t6 Bluepill microcontroller	
Jumping wire crimping kit with	1	Kit med crimp housings och hankontakter (inte hittat hankontakter på infotiv?)	112
SPCU Components			
Digital to Analog Converter MCP4725	2	Circuit to send analog voltages to gokart pedals	120
Sabertooth 25x2 Motor driver	1	Drives the dc-dc motor	1300

BOM Autonomous Platform Generation 4

		TOTAL PRICE SEK:	16139,3
Part Name	**Quantity**	**Area of Use**	**Price (SEK)**
Sabertooth Kangaroo x2 Motion Controller	1	Circuitboard motion controller board daughterboard to sabertooth -Send positional command through serial to it -connect EMG49 Motor Hal sensors to it, and two endstop switches and it will configure itself	270
Microswitch 440 mN	2	Microswitch to detect steerng maximum and minimum limits	30
IR Speed sensors	4	Speed sensors front and back wheels	164
emg49 dc-dc motor with 50-1 gearbox	1	DC-DC motor to controll the steering wheel	
EMG49 mounting bracket	1	mounting L bracket for motor	
EMB49 mounting bracket holder		See AP3 documentation for dimensions, leftover component was used on AP4	

Mounting Hardware & Components

Screw m6x50mm	1	**Insexskruv M6 x 50, rostfri A2, 10 st.**	60
Angled Steel bracket	6		59,4
threadscrew	1		159
Screw M5x35 mm	1		40
Screw m3x10	2	set of 100 screws	130
Screw m3x16	2	set of 100 screws	130
Screw m4x10	2	set of 100 screws	220
Screw m4x16	2	set of 100 screws	
nut M4	4	sets of 100 nuts	260
nut M3	4	sets of 100 nuts	104,88

Wire Management

Sprial Wire Tube, 10mm	5	Wrap around loose cables	150
Sprial Wire Tube, 23mm	5	Wrap around loose cables	300

List of wires

Ethernet			
DB9			
1.5 mm coppar wire red			
1.5 mm coppar wire black			

BOM Autonomous Platform Generation 4			
		TOTAL PRICE SEK:	16139,3
Part Name	**Quantity**	**Area of Use**	**Price (SEK)**
Sensors			
USB kamera, Logitech c920	1	Front facing camera AP4.Documented to work well with existing ros2 camera packages!	1260

D

Code Listings

Listing D.1: dbc format for CAN protocol

```
VERSION "HIPBNYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY/4/%%%/4/'%**4
   YYY///"


NS_ :
        NS_DESC_
        CM_
        BA_DEF_
        BA_
        VAL_
        CAT_DEF_
        CAT_
        FILTER
        BA_DEF_DEF_
        EV_DATA_
        ENVVAR_DATA_
        SGTYPE_
        SGTYPE_VAL_
        BA_DEF_SGTYPE_
        BA_SGTYPE_
        SIG_TYPE_REF_
        VAL_TABLE_
        SIG_GROUP_
        SIG_VALTYPE_
        SIGTYPE_VALTYPE_

BS_:

BU_: SPCU CMCU


BO_ 500 Error_SPCU: 1 SPCU
 SG_ Heartbeat : 0|1@1+ (1,0) [0|1] "Bool" Vector__XXX
 SG_ Propulsion_Error : 1|1@1+ (1,0) [0|1] "Bool" Vector__XXX
 SG_ Steering_Error : 2|1@1+ (1,0) [0|1] "Bool" Vector__XXX

BO_ 1000 Set_SPCU: 8 CMCU
 SG_ Act_ThrottleVoltage : 33|13@1+ (1,0) [850|5100] "mV"
    Vector__XXX
 SG_ Act_BreakVoltage : 0|13@1+ (1,0) [850|5100] "mV" Vector__XXX
```

```
 SG_ Act_SteeringPosition : 17|8@1- (1,0) [-180|180] "Degree"
    Vector__XXX
 SG_ Act_Reverse : 16|1@1+ (1,0) [0|1] "Bool" Vector__XXX
 SG_ Act_SteeringVelocity : 25|8@1+ (1,0) [0|0] "units/s"
    Vector__XXX

BO_ 2000 Get_SPCU: 8 SPCU
 SG_ Get_SteeringAngle : 0|9@1- (1,0) [-180|180] "Degree"
    Vector__XXX
 SG_ Get_ReverseMode : 9|1@1+ (1,0) [0|1] "Bool" Vector__XXX

BO_ 100 Request_Heartbeat: 8 CMCU
 SG_ Sig_Req_Heartbeat : 0|64@1+ (1,0) [0|255] "" Vector__XXX

BO_ 101 Response_Heartbeat_SPCU: 8 SPCU
 SG_ Response_Heartbeat_sig : 0|8@1+ (1,0) [0|1] "" Vector__XXX

BO_ 102 Response_Heartbeat_XXX: 8 Vector__XXX


CM_ BO_ 500 "Error Handler frame";
CM_ SG_ 500 Heartbeat "heartbeat to detect that the ecu is still
   active";
CM_ SG_ 500 Propulsion_Error "Boolean to detect if some error has
   occrued in propulsion control";
CM_ SG_ 500 Steering_Error "Boolean to detect if some error has
   occured in Steering Control";
CM_ BO_ 1000 "Set values";
CM_ SG_ 1000 Act_ThrottleVoltage "";
CM_ BO_ 2000 "Get sensor reading";
CM_ BO_ 100 "To see if the node is still running";
CM_ SG_ 100 Sig_Req_Heartbeat "Request heartbeat signal from HW
   nodes";
CM_ BO_ 101 "Responds to a request heartbeat frame";
CM_ SG_ 101 Response_Heartbeat_sig "Sends a value of 1 to respond
   to a heartbeat request";
BA_DEF_ SG_  "SPN" INT 0 524287;
BA_DEF_ BO_  "VFrameFormat" ENUM  "StandardCAN","ExtendedCAN","
   reserved","J1939PG";
BA_DEF_  "DatabaseVersion" STRING ;
BA_DEF_  "BusType" STRING ;
BA_DEF_  "ProtocolType" STRING ;
BA_DEF_  "DatabaseCompiler" STRING ;
BA_DEF_ BO_  "GenMsgSendType" ENUM  "cyclic","spontaneous";
BA_DEF_ BO_  "GenMsgCycleTime" INT 2 50000;
BA_DEF_ BO_  "GenMsgAutoGenSnd" ENUM  "No","Yes";
BA_DEF_ BO_  "GenMsgAutoGenDsp" ENUM  "No","Yes";
BA_DEF_ SG_  "GenSigAutoGenSnd" ENUM  "No","Yes";
BA_DEF_ SG_  "GenSigAutoGenDsp" ENUM  "No","Yes";
BA_DEF_ SG_  "GenSigEnvVarType" ENUM  "int","float","undef";
BA_DEF_ SG_  "GenSigEVName" STRING ;
BA_DEF_ BU_  "GenNodAutoGenSnd" ENUM  "No","Yes";
BA_DEF_ BU_  "GenNodAutoGenDsp" ENUM  "No","Yes";
BA_DEF_  "GenEnvVarEndingDsp" STRING ;
BA_DEF_  "GenEnvVarEndingSnd" STRING ;
BA_DEF_  "GenEnvVarPrefix" STRING ;
```

```
BA_DEF_DEF_  "SPN" 0;
BA_DEF_DEF_  "VFrameFormat" "J1939PG";
BA_DEF_DEF_  "DatabaseVersion" "DEMO PLUS";
BA_DEF_DEF_  "BusType" "";
BA_DEF_DEF_  "ProtocolType" "";
BA_DEF_DEF_  "DatabaseCompiler" "";
BA_DEF_DEF_  "GenMsgSendType" "spontaneous";
BA_DEF_DEF_  "GenMsgCycleTime" 100;
BA_DEF_DEF_  "GenMsgAutoGenSnd" "Yes";
BA_DEF_DEF_  "GenMsgAutoGenDsp" "Yes";
BA_DEF_DEF_  "GenSigAutoGenSnd" "";
BA_DEF_DEF_  "GenSigAutoGenDsp" "";
BA_DEF_DEF_  "GenSigEnvVarType" "undef";
BA_DEF_DEF_  "GenSigEVName" "";
BA_DEF_DEF_  "GenNodAutoGenSnd" "Yes";
BA_DEF_DEF_  "GenNodAutoGenDsp" "Yes";
BA_DEF_DEF_  "GenEnvVarEndingDsp" "Dsp";
BA_DEF_DEF_  "GenEnvVarEndingSnd" "Snd";
BA_DEF_DEF_  "GenEnvVarPrefix" "Env";
BA_ "ProtocolType" "J1939";
BA_ "BusType" "CAN";
BA_ "DatabaseCompiler" "CSS ELECTRONICS (WWW.CSSELECTRONICS.COM)";
BA_ "DatabaseVersion" "1.0.0";
```

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY