INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Bidirectional CAN bus telemetry on a Formula Student/SAE car

Florian Eich

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Bidirectional CAN bus telemetry on a Formula Student/SAE car

Florian Eich

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Nils gentschen Felde Tobias Guggemos
Abgabetermin:	25. April 2016

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 25. April 2016

.....
(Unterschrift des Kandidaten)

Abstract

In professional motorsports, live telemetry systems have long been the state of technology. In Formula 1, cars have been sending data to the pits since the late 80s, with continuous data transmission starting in the early 90s. Bidirectional telemetry was permitted for a few years but banned in 2003 [f1r]. Thus, commercial systems do not usually implement it, at the expense of customers who might want to make use of it, like Formula SAE/Student teams. Other drawbacks of commercial solutions include their high prices and their lack of openness which results in a severe lack in adaptability.

In this Bachelor's thesis, the feasibility of a bidirectional telemetry system is demonstrated for a Formula SAE/Student car. The possibility of communicating with the commercial control units through CAN bus is created, as well as adaptable interfaces for data acquisition and telemetry. The resulting system is then evaluated through testing.

The methods chosen are largely rooted in a *Systems Engineering* context, with guidelines by the IEEE and several different RFCs providing the framework for the proceedings. As a result, a clear specification of the system requirements is provided. The system design and the following implementation is then executed accordingly, resulting in a system prototype which is then measured against the specified requirements.

Contents

1. Introduction

Cars are data generators. Sensors on board of cars repeatedly make measurements of the physical conditions of the car, such as fluid temperatures and pressures or acceleration forces, and send them via buses to electronic control units (ECUs) where they are processed. In some cases, these measurements are used to generate signals to mechanical components stabilizing the car or given to the driver as a warning. In motorsports, engineers have to evaluate the data generated by the car in real-time so they can, together with the driver, make sure the car is operating to its full capacity. This requires the data to be transmitted to the engineers wirelessly. The entire process from measurement to remote evaluation is called *telemetry*, which is thus the namesake for systems providing it.

Figure 1.1.: Pit wall setup of BMW Sauber F1

Figure 1.1 shows the pit wall setup of the BMW Sauber F1 team, around 2007. This is where the track engineers sit during tests, qualifying and races, analyzing data and communicating with the drivers. This setup is reflective of the setup used in other racing series as well, such as Formula Student/SAE. However, in contrast to Formula 1 and most other racing series, Formula Student/SAE allows for telemetry systems that work bidirectionally to be installed in cars, meaning that engineers are able to send data to the car and its control units (except for throttle and brake signals, which are only allowed to be operated by pedals inside the car), thereby optimizing the car's capability. This aspect is key to teams, since it enables them to change settings on the car during a run and even during a lap, which is beneficial for two reasons:

- Engineers can test settings quicker and interactively. This is a major advantage in terms of driver assistance functionality development and deployment.
- The number of functions controlled by the driver is reduced. The driver is therefore more focused on his primary task of driving the car.

Ever since software has become a major part in controlling any car, this has been of central interest. Having different settings for different parts of the circuit can give significant advantages as demonstrated by the former *active suspensions* used in Formula 1, which adjusted the height of the cars depending on which section of the circuit they were on in order to optimize aerodynamic efficiency for each section. Furthermore, it is still common in Formula 1 for the pit stand to tell the driver the settings he has to make in order to, for example, reduce fuel usage to complete the race - this could be done by the engineers directly without distracting the driver.

These *remote control* features are now banned in Formula 1 and all other professional racing series. In Formula Student/SAE it is still legal to use bidirectional telemetry and teams are even encouraged to do so within the regulations of the sport. It has therefore been of major interest to all competing teams to gain remote access to their cars.

1.1. Background

Formula SAE (FSAE) [SAE] is an international constructor's competition for university students. The idea is that students from a university can form a team to engineer a racecar according to a universal set of rules which are published annually by the SAE International [Int] (formerly Society of Automotive Engineers). An event is held every year where the teams come together to compete against each other. The competitions for Formula SAE are held in North America, but there are adaptations of the competition all over the world which are all based on the original rules, enabling teams to compete internationally. In Germany, the competition is called Formula Student Germany (FSG) [Ger]. The main event for FSG is held at the Hockenheim racetrack [Gmba] in August of every year. Cars running there have to comply with the FSAE rules [Int15] and the FSG rules [e.V15], the latter being an addendum to the former.

Being familiar with Formula Student (or motorsports in general) is not required for understanding this thesis. The important aspects will be explained where necessary. The interested reader will find a detailed explanation of Formula Student elsewhere, the German Wikipedia entry on *Formula Student Germany* [fsg] being a plentiful first resource. However, it is helpful to be aware of the level of engineering displayed by these cars.

The team from the University of Applied Sciences (UAS) Munich, municHMotorsport [mun], designs and builds two cars every season to compete in both the internal combustion engine and the electric motor category of Formula Student/SAE. The electric car from the 2015 season, the PWe6.15[1], is displayed in figure 1.2. It weighs close to 190kg without the driver and features four electric motors at the wheels.

[1]The "PW" in these names is short for the motto of municHMotorsport, "Passion Works". The number before the dot in the name is given to the cars consecutively whereas the number after the dot indicates the season the car was built for. A further distinction is the "e", which the cars with electric motors carry additionally as they came later (which is also indicated by the lower first number). Should other, similarly looking names occur in this thesis, please read them accordingly.

Figure 1.2.: The PWe6.15

Its resemblance to Formula 1 cars is not merely coincidental, as the rules, like in Formula 1, allow for engineering freedom unseen in any other racing series, only placing restrictions on the overall car architecture, size, power output and safety and leaving it entirely up to the teams how to engineer within these boundaries. This freedom, paired with the passion and dedication of teams worldwide, has enabled Formula Student Electric cars to set the world record for 0 to 100km/h sprints with four wheeled electric vehicles for the past 2 years.

In modern automotive technology, electronic components play an ever increasing role. Originating from the need to control engines, they are now used for virtually everything, from active and passive safety functionality to driver comfort and infotainment systems. In the context of automotive electronics, the term electronic control unit (ECU[2]) is commonly used. This is also true for race cars and more specifically Formula Student/SAE race cars, which today carry a plenitude of ECUs for various purposes, such as gathering and evaluating sensory input or distributing signals to active components. Especially the four wheel drive cars with electric motors rely heavily on ECUs, commonly using a central unit using driver input and sensory data to compute the power output for each individual wheel. These central ECUs are commonly referred to as vehicle control units (VCU).

With several ECUs and potentially a VCU present, communication between these components must be established. A few networking technologies have been developed for on-board communication between ECUs in vehicles. Most notably, in the 1980s the Robert Bosch GmbH developed the controller area network (CAN) bus [Rob91], which is still widely in use today and also primarily used in Formula Student/SAE. A detailed description of the CAN protocol is provided in chapter 2.

[2]Originally, ECU stood for *engine control unit*, but with the growing number of electronic systems in vehicles, this acronym came to mean *electronic control unit* over time. There are still acronyms for specific modules commonly found in cars, such as ECM or TCM for *engine control module* or *transmission control module* respectively.

Furthermore, to establish a possibility for engineers who are not sitting in the car to evaluate sensory data and communicate with the car's ECUs, external communication must be implemented as well. While the technology to wirelessly connect two endpoints with each other within the limits of a Formula Student/SAE track exists and has been implemented, as of now there does not exist an open and portable solution in terms of hardware and software to establish a stable toolchain. In this thesis, a bidirectional telemetry system for a Formula Student/SAE car will be developed with the goal of serving as a cheaper, more flexible and more robust replacement of the current solutions. The team requesting this development is municHMotorsport, who also set the general project constraints and defined the coarse functional requirements in the form of user stories and will be responsible for the mission critical hardware implementation, as this is not within the scope of this thesis. In this thesis, municHMotorsport is referred to as *the team*.

1.2. Related Work

There have been efforts to implement bidirectional telemetry by other teams in Formula Student/SAE. Braune shows in his semester thesis [Bra14] that Wi-Fi is suitable for Formula Student/SAE telemetry including video live stream with his design offering 5MB/s throughput at distances up to 600m, with no impact given by having one of the clients moving at speeds up to 75km/h. Although his work is focused mostly on wireless data link reliability, telemetry is implemented using the software proprietary to the VCU used by Akademischer Motorsportverein Zürich (AMZ), the team from Eidgenössische Technische Hochschule (ETH) Zürich he was developing for. In the section on opportunities for future work, he states that increased platform independence, among other things, would be desirable.

In his Bachelor's thesis [Haa07], Haase develops a bidirectional telemetry system for a Formula Student/SAE car and shows that communication with the car's ECUs over a Wi-Fi link is possible using the car's CAN bus. He develops the entire telemetry system in the *remotely accessible local measurements* sense, putting much effort into developing the sensory and communication network on the car and introducing a time triggered CAN (TTCAN) to enable message prioritization beyond the ID based arbitration of regular CAN. His work strongly focuses on the overall design of all the hardware components.

Hadaller, Li and Sung analyze unidirectional telemetry over Wi-Fi in their project report [DH04], which also focuses on the quality of the wireless connection. Their achieved throughput is far below that of the system developed by Braune [Bra14], but concludes that it is important to avoid visual obstructions between the client antennas, as these have a grave impact on the Wi-Fi signal strength. This is an important aspect to be considered for antenna placement.

In his Master's thesis [Cop09], Copeto creates a data logging and telemetry system for a Formula Student/SAE car using ZigBee and achieves a throughput of 4.28kB/s when sending CAN messages wirelessly over a distance of up to 300m. His approach for the data analysis toolchain is to create a GUI based data analysis tool.

All of the above either focus heavily on the wireless data link quality or the development of a full data acquisition solution. They either rely heavily on proprietary software solutions or take the approach of designing a full data analysis solution. While they provide important data for this work such as the data link quality and achievable throughput rate of wireless transmission, none of them provides a modular approach to incorporate the pre-

existing toolchain, thereby creating an application programming interface unaffected by the connecting tools. They solve the same problem, but do not provide an open platform and thus are mostly car specific, something this work specifically strives to avoid in order to enable the team to use the resulting system in different setups.

Of course, there are commercial telemetry systems offered by several vendors. 2D Debus & Diebold Meßsysteme GmbH, Bosch, Magneti Marelli, MoTeC, Cosworth, Pico Technology and McLaren Applied Technologies all offer solutions commonly used in professional motorsports. These solutions cover a huge range of functionality and are most commonly only available to use together with licensed data analysis software tools. Costs vary greatly and are only available upon request from all vendors. Considering the price of the Formula Student/SAE license fee of the 2D data analysis tool *WinARace* (further explanation see chapter 2) of €500.- per year in addition to the hardware package with an initial cost of €1500.-, substantial costs for upgrades and renewal licenses are to be expected.

Furthermore, none of the commercial solutions offer bidirectional telemetry out of the box as this is not permitted in any professional motorsports application. Some vendors offer optional software packages or offer service packages including field application engineers, both of which adds to the cost of the packages. Most vendors do not openly offer the functionality at all.

1.3. Methodology and Outline

As this is a systems engineering effort, the methodology is loosely modeled on the process described in *Systems Engineering Fundamentals* [Pre01], a text by the Defense Acquisition University [Uni], which is a university of the United States Department of Defense. A graphical outline of this process can be seen in figure 1.3, taken from the same text.

Following the best practices of systems engineering, this work begins with the analysis of the current state and its shortcomings in chapter 2. A description of the current systems can be found here, defining the technology base and the prior development effort. Chapter 3 contains the identification and analysis of the requirement inputs, resulting in a set of requirements for the system. These requirements comply with IEEE P1233 [syr98], a guideline for developing *System Requirements Specifications* (SyRS) developed to provide unambiguous, verifiable requirements to the team. Chapter 4 covers the functional analysis and design synthesis of the system. Fine-grained definitions of system functionality as well as software architecture, data structure and protocol definition can be found here. In chapter 5, the new system is implemented using the outputs of the previous chapters as a basis. The focus lies on the software, although a hardware implementation is provided as well. This is followed by tests in the same chapter to conclude the success in fulfilling the given requirements. Lastly, chapter 6 contains the conclusion of this work including an outlook for future work.

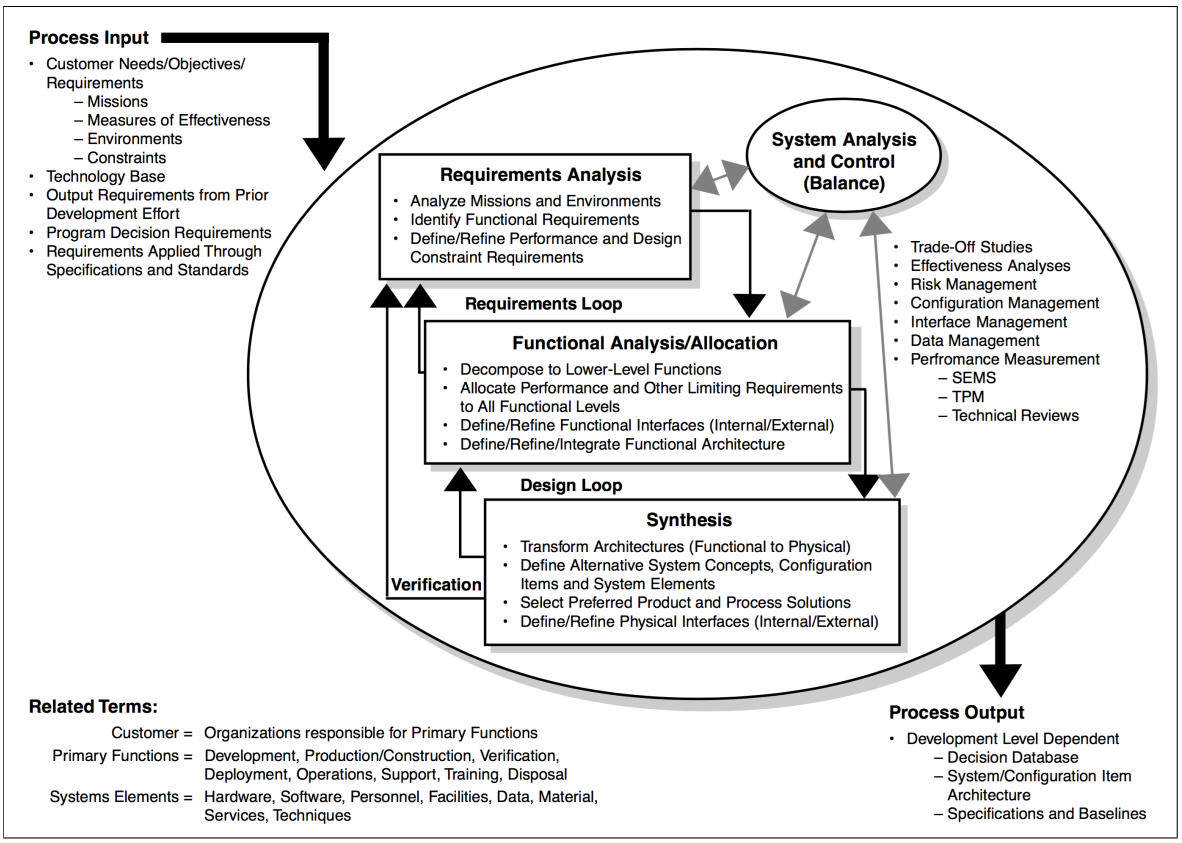


Figure 1.3.: Systems Engineering Process according to “Systems Engineering Fundamentals”

2. Problem Analysis

To pinpoint the existing problems, this chapter analyzes the current setup. An overview of the generic setup of on-board electronics used in all recent cars by the team is given, exemplifying what is to be assumed to be a viable setup for any Formula Student/SAE car. As explained in chapter 1.1, Formula Student/SAE cars in general and the cars designed by the team in particular communicate via controlled area network (CAN) bus, which is thus explained in detail in section 2.1. In order to complete the description of the technology base, the Wi-Fi standard in use by the current setup is identified and briefly explained in section 2.2. Then, the current telemetry setups and their shortcomings are highlighted in order to outline the development effort prior to this work in section 2.4. Thus, this chapter identifies the first set of inputs for the requirements analysis 3.

2.1. Controller Area Network Bus

The controller area network (CAN) bus was developed by Robert Bosch GmbH in the 1980s and first released in 1986, but has since been revised multiple times, most notably in 1993 with the CAN Specification Version 2.0 [Rob91] and as recently as 2012 with the CAN FD 1.0 specification [Rob12] which introduced the option to use a flexible data (FD) rate while retaining backwards compatibility with CAN 2.0. Buses operating according to the CAN 2.0 protocol are widely used in passenger vehicles today with CAN being part of on-board diagnostics (OBD) standards from the United States of America (OBD-II), the European Union (EOBD) and Australia (ADR). Evidently, the CAN bus is very popular in the automotive industry and other industries such as aerospace, military technology and more recently also industrial application. This popularity is due to the robustness and simplicity of CAN:

- twisted pair cable with very good electromagnetic immunity without additional shielding is used to transport data
- daisy chaining is possible if done correctly
- an exceedingly low residual failure probability of $R_{\text{message}} \cdot 4.7 \cdot 10^{-11}$ where $R_{message}$ is the message error rate [Rob91] due to high bus stability combined with multiple levels of error handling strategy

The CAN standard implements the physical and the data link layer in the Open Systems Interconnection (OSI) model.

CAN is a serial bus connecting two or more ECUs using twisted pair cable as mentioned above. Different bit rates between 125kBit/s and 1MBit/s are possible, whereas an increased bit rate decreases the possible length of the bus. Since bus length is kept short and is thus not a concern throughput is a major concern in Formula Student/SAE cars, which is why

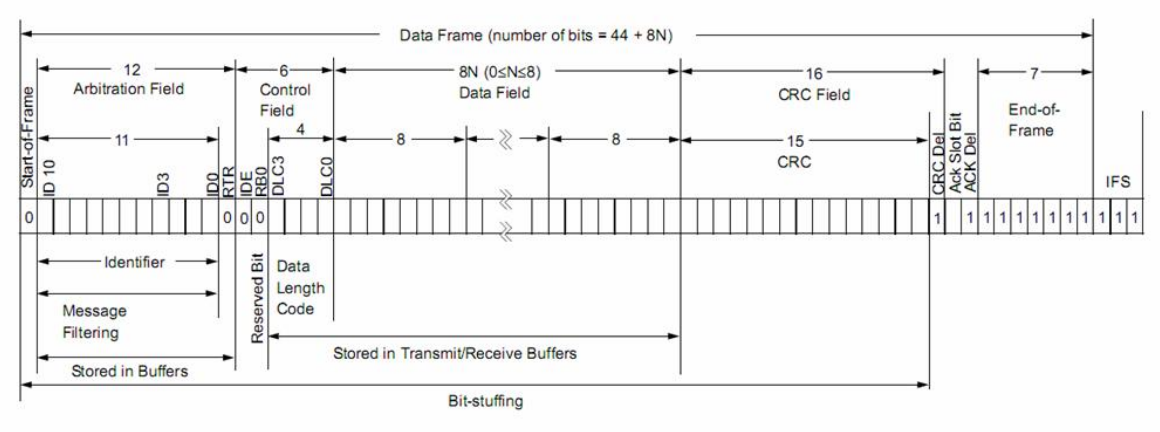


Figure 2.1.: Structure of a CAN data frame without extended ID

high speed CAN (500kBit/s or 1MBit/s of throughput) is commonly used. The team has used both throughput rates in the recent cars.

Messages transported on the CAN bus are received by all the devices connected to the bus and have a fixed format, including an identifier which also serves as an arbitration field. When the bus is free (i.e. not busy), any ECU can start broadcasting. Medium access control is realized by bit arbitration so the identifier also determines the priority with which the message is sent. It is thus possible to prioritize messages by their ID.

The structure of a CAN data frame without extended ID is shown in figure 2.1. There are four types of CAN frames in total:

1. data frame
2. remote request frame
3. error frame
4. overload frame

A signal on the CAN bus is either dominant (0) or recessive (1). Table 2.1 shows the content of a CAN frame in more detail, covering data frames without extended identifier and remote request frames. CAN frames with extended ID are structured in much the same way, as shown in table 2.2.

The maximum overall length of a CAN data frame is thus either 108 Bits or 129 Bits, depending on whether extended ID is used or not. For a payload of 64 Bits, this means an overhead of 40-50%. Error frames and overload frames are similar in structure and differ mainly in usage, as different scenarios trigger them.

Error frames Error frames consist of two fields: The flags field, which contains a superposition of error flags from different stations (6 - 12 Bits) and the error frame delimiter field, containing 8 recessive (1) Bits. Error frames are transmitted by ECUs which detect an error on the bus.

Overload frames Overload frames consist of two fields: The flags field, which contains six dominant (0) Bits, and the overload frame delimiter containing 8 recessive (1) Bits. Overload frames are sent by ECUs intending to indicate that they need a delay before the next data transmission, or wanting to indicate that they detected a dominant bit

Field name	**Length (Bit)**	**Content**
Start-of-frame	1	Denotes the beginning of a transmission. Always dominant (0).
Identifier	11	Identifier.
Remote Transmission Request (RTR)	1	Dominant (0) for data frames, recessive (1) for remote request frames
Identifier Extension (IDE)	1	Dominant (0) for standard, recessive (1) for extended ID frames
Reserved Bit (r0)	1	Reserved bit. Must be dominant (0).
Data Length Code (DLC)	4	Indicates number of bytes of data between 0 (if remote request) and 8. If first bit is set (equivalent to 8), the others are ignored.
Data Field	0 - 64	Contains the data. Length indicated by DLC.
Cyclic Redundancy Check (CRC) Field	15	Contains CRC code.
CRC Delimiter	1	Delimits the CRC code. Must be recessive (1).
Acknowledge (ACK) Slot	1	Recessive (1) in transmitter message, receivers send dominant (0). Recessive Bit on the bus is thus an indication for the transmitter that no receiver picked up the frame.
ACK delimiter	1	Must be recessive (1).
End-of-frame (EOF)	7	Must be recessive (1).

Table 2.1.: Frame format for CAN data and remote request frames

Field name	**Length (Bit)**	**Content**
Start-of-frame	1	Denotes the beginning of a transmission. Always dominant (0).
Identifier (A)	11	Identifier part A.
Substitute Remote Request (SRR)	1	Must be recessive (1).
Identifier Extension (IDE)	1	Dominant (0) for standard, recessive (1) for extended ID frames
Identifier (B)	18	Identifier part B.
Remote Transmission Request (RTR)	1	Dominant (0) for data frames, recessive (1) for remote request frames
Reserved Bits (r0, r1)	2	Reserved bits. Must be dominant (0).
Data Length Code (DLC)	4	Indicates number of bytes of data between 0 (if remote request) and 8. If first bit is set (equivalent to 8), the others are ignored.
Data Field	0 - 64	Contains the data. Length indicated by DLC.
Cyclic Redundancy Check (CRC) Field	15	Contains CRC code.
CRC Delimiter	1	Delimits the CRC code. Must be recessive (1).
Acknowledge (ACK) Slot	1	Recessive (1) in transmitter message, receivers send dominant (0). Recessive Bit on the bus is thus an indication for the transmitter that no receiver picked up the frame.
ACK delimiter	1	Must be recessive (1).
End-of-frame (EOF)	7	Must be recessive (1).

Table 2.2.: Frame format for CAN data and remote request frames

in the timeslot between data frames (intermission, indicated by at least 3 Interframe Spacing (IFS) Bits between data and remote request frames.)

2.2. Wi-Fi

Wi-Fi is a wireless transmission technology trademarked by the Wi-Fi Alliance, a conglomerate of countries all over the world that tests equipment for compatibility with the IEEE 802.11 set of standards. This family of standards specifies media access control (MAC) and physical layer (PHY) for WLAN networks using different frequency bands, most notably 2.4GHz and 5GHz. The standards are under active development with amendments being released every few years since 1997, during which time achievable ranges as well as data throughput have continuously been increasing. The Ubiquiti Bullet^{TM} in use by the team operates according to the IEEE802.11b/g/n amendments to the original standard, in the 2.4GHz frequency band. It is *Conformité Européenne* (CE) certified and is therefore compliant with EU legislation. It has an output power of 28dBm and is, according to the vendor, capable of covering "distances over 50km" when choosing an appropriate antenna [Ubia]. The cable based interface it provides is Ethernet 10/100 BASE-TX RJ-45, compatible with regular Ethernet patch cables and delivering throughput of up to 100MBit/s.

Products like the Ubiquiti Bullet^{TM} are common in outdoor use. Due to their high output power, these systems can cover comparatively large distances with ease using the existing IEEE 802.11 standards. As mentioned in section 1.2, it has been found that using these outdoor antennas is viable for Formula Student/SAE cars, i.e. transmission is stable over the required distances given that line of sight can be guaranteed. An additional aspect of the IEEE 802.11 series of standards is the incorporation of features providing different levels of intrusion security.

When the team encountered problems with the telemetry link of a specific on-board ECU they were using, they suspected the Wi-Fi connection to be lacking and thus conducted tests with the Bullet in order to determine its performance. Although the hardware used by Braune [Bra14] is not the same, his results in terms of connection distance were confirmed, eliminating the Wi-Fi connection as the reason for connection losses.

2.3. On-Board Electronics Setup

In 2015, the team developed and built two Formula Student cars, one for the internal combustion engine class and one for the electric motor class, named PW9.15 and PWe6.15 respectively. Several CAN buses run through both of these cars, whereas the buses are organized hierarchically by importance of signal or where this is not possible by logical group.

The PW9.15, the combustion car, communicates all sensory data to the engine control unit and the energy management system (EMS[1]) through 2 CAN bus lines. One of them, the primary CAN, is only used for operation critical data and the other for everything else, e.g. additional sensors or a selection of sensors required for a certain testing routine.

The PWe6.15, the electric car which has four driven wheels, has 4 CAN buses in use, named CAN1 through CAN4 here: CAN1 carries messages from the vehicle control unit to the inverters, communicating the desired torque to the powertrain. CAN2 carries vital

[1]The energy management system (EMS) of an internal combustion car is used as the central hub for the wiring loom. It controls, among other things, the cooling systems for water and oil.

signals like throttle position or steering angle which are required for driver interaction. The other two CAN buses are configured similar to the PW9.15, carrying primary and secondary sensory input respectively.

A generalized overview of the setups is provided in figure 2.2. There are several ECUs installed throughout the cars. Two ECUs collect sensory data in both the front and the rear of the cars and transmit it via the CAN bus, on which central ECUs and loggers are able to receive them. The dashboard ECU also collects sensory data but mainly serves as the human machine interface (HMI), displaying information to the driver and providing the driver with input devices. The central control units, which are represented by the engine control unit in the combustion cars and by the vehicle control unit in the electric cars, collect data relevant to their operation and control their respective propulsion system. Logging is done by several different systems on each car, with telemetry being available using proprietary systems with workarounds. In addition to the systems displayed, there are several more ECUs on each car controlling components like cooling systems.

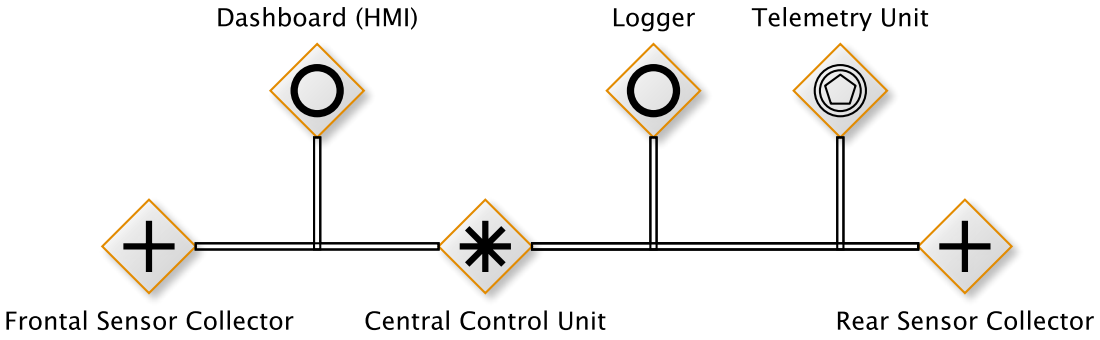


Figure 2.2.: System diagram (abstraction of both cars)

PW9.15 The engine control module of the PW9.15 is the M800 [MoT], an ECU manufactured by the Australian company MoTeC [Ltd] and has the ability to log data vital to its operation. A 2D Debus & Diebold Meßsysteme GmbH system is also installed to log sensory data. Additionally for testing, the PW9.15 has the GL2000 [Veca], a raw CAN data logger by Vector Informatik GmbH, on board. Telemetry is realized for the ECU by installing a MoTeC CAN-to-USB device connected to a USB-to-Ethernet module connected to a Ubiquiti Bullet^{TM} Wi-Fi router [Ubia] in the car.

PWe6.15 The vehicle control unit (VCU) of the PWe6.15 is the MicroAutoBox II [dSP] by dSPACE GmbH [ddspceG]. It provides live telemetry functionality via Ethernet port, to which a Ubiquiti Bullet^{TM} Wi-Fi router is connected. However, while the software provided by dSPACE allows for an Ethernet connection to the device, it was never intended to be wirelessly bridged, which thus can lead to undefined behaviour. The VCU also logs data relevant to its operation with further data logging functionality provided by a 2D Debus & Diebold Meßsysteme and the same logger as on the PW9.15, the Vector Informatik GmbH GL2000.

Logging of data can be done in two different ways, both of which have upsides and downsides. They can be described in the following way.

- Storing sensory values as time series, e.g. tables with the timestamp as primary key and wheel speed, steering angle etc. as values.
- Storing CAN messages as time series, e.g. tables with timestamp as primary key and raw CAN message contents as values.

Since on-board data distribution is time critical and throughput is limited, CAN messages mostly carry the data for more than just one sensory value. Therein lies the difference: the central ECUs and the 2D systems only store sensory values as time series, which is why in order to gain access to a full recording of the CAN bus traffic, an extra device is required in the form of the Vector Informatik GmbH loggers. These devices also capture error frames and overload frames, which sensory value loggers do not, thereby enabling the engineers to recognize any problems with the physical buses more easily. However, since the Vector Informatik GmbH systems are very large and heavy compared to the scale of Formula Student/SAE cars, they are not usually in use during competition, which is seen as a moderate drawback by the responsible engineers.

The other non-ECU specific piece of hardware beside the Vector Informatik GmbH loggers, the Ubiquiti Bullet™, is a configurable Wi-Fi device with an Ethernet interface. The current setup uses two of these devices, one in the car and one where the engineers are sitting. The former is configured as a client and connected to the latter, which is configured as an access point with DHCP. Other setups using the Bullet™ are possible, but all setups share the common characteristic of providing an Ethernet interface to the wireless bridge. The team has conducted tests with the Bullets and is content with their performance.

2.4. Current Setup

There are several problems with the status quo. While data logging works within the limits set by the vendors of the solutions in place and the fact that a full CAN recording is not available during competition, the existing telemetry systems are not so much telemetry solutions but rather workarounds to gain remote access to the central ECU of the car. As such, they sometimes cause system failures that have negative impact on the team's results and even in the best of times only allow for data to be evaluated through the respective ECU, which limits their scope of use. Even more, they are specific to the type of central ECU, which means not only that if telemetry is to be used the team is bound to the corresponding vendor but also means that only engineers who are familiar with the respective toolchain and have access to the respective software license are able to supervise the car. The following analysis describes the situation in terms of these problems and outlines them in relation to on-track experience of the engineers.

Live telemetry On the PW9.15, the goal of the team was to gain remote access to the M800 engine control module. The M800 comes with a CAN-to-USB adapter, an external device plugged into the ECU for analysis purposes. This adapter can be used to extract data and interact with the device using the MoTeC Software, a tool called *ECU Manager*. The team connected a USB-to-Ethernet adapter to the CAN-to-USB-adapter, which was then connected to the Ubiquiti Bullet™ mounted in the car. Using the bridge setup of airOS®, another Bullet™ with a Laptop connected via Ethernet cable was used to realize a connection to the M800 remotely. This is the

setup currently in use. While the M800 with ECU Manager connected is able to provide real-time monitoring of a limited amount of data, it bears all the problems outlined above and adds a chain of single points of failure. On the PWe6.15, live telemetry has been implemented similarly to combustion car. The team's goal also was to gain remote control of the central control unit, dSPACE's MicroAutoBox II. The setup is also essentially the same as on the PW9.15, although the USB-to-Ethernet adapter is rendered unnecessary by the Ethernet port the MicroAutoBox II has. That being said, the usage of the Ethernet port in this fashion was not anticipated by dSPACE and causes undefined behaviour, which is suspected of crashing the VCU when the connection fails or is otherwise impaired, resulting in the car failing to finish races on multiple occasions. This issue could not be resolved by the team nor dSPACE, even though the Wi-Fi connection has been eliminated as the source of error. In testing, the VCU has always restarted after a connection failure, but the system can still not be considered robust under these circumstances.

There is another problem: the software provided by MoTeC and dSPACE to interact with their respective control units is licensed. While the tools are for the most part well engineered and give users the ability to quickly create GUIs for monitoring data as shown by figure 2.3, this means that should the team ever decide to change systems and thus not immediately require these licenses any longer, older cars will not have access to live telemetry. Especially for the electric cars, this is problematic: The PWe4.13 and PWe5.14, electric cars from seasons 2013 and 2014, are still operational, but the temperature of their batteries has to be monitored at all times during operation. Without a basic telemetry system, there is no way to operate these cars safely.

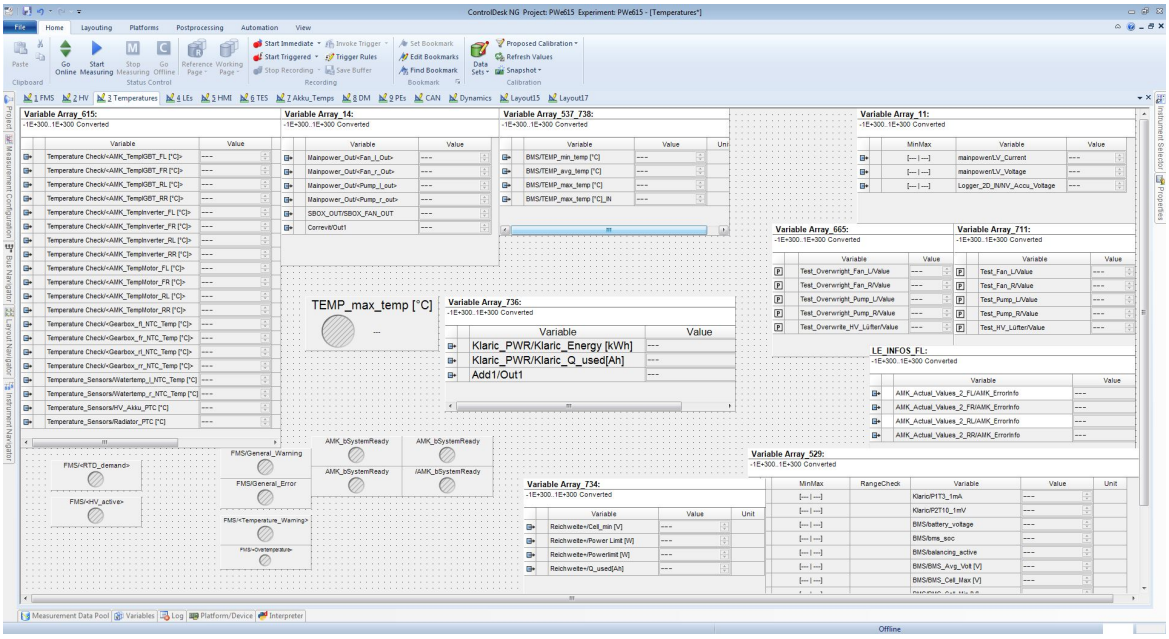


Figure 2.3.: Screen shot of Control Desk, the configuration and telemetry tool by dSPACE

Logging Logging and data analysis is done mainly through the 2D Debus & Diebold Meß systeme GmbH system as pointed out earlier. The "Formula Student Kit" consists of a hardware logger that has accelerometers, a GPS module and 2 CAN inputs. Included

in the bundle, which costs €1500.-, is the 2D Debus & Diebold Meßsysteme GmbH software package which includes a data analysis tool called WinARace, shown in figure 2.4. While the hardware stays in the hands of the team, the software license has to be renewed every year for €500.-. All of this is not a problem as such, especially because WinARace is a capable piece of software also in use in professional application but the team would prefer to have 1. a system that can also be used if WinARace is not available and 2. a system that also offers live telemetry capabilities at the same time. If possible, providing a data interface for WinARace would be a clear advantage, although especially the engineering team for the electric cars is looking to move to MATLAB®/Simulink with the data analysis in order to simplify the toolchain.

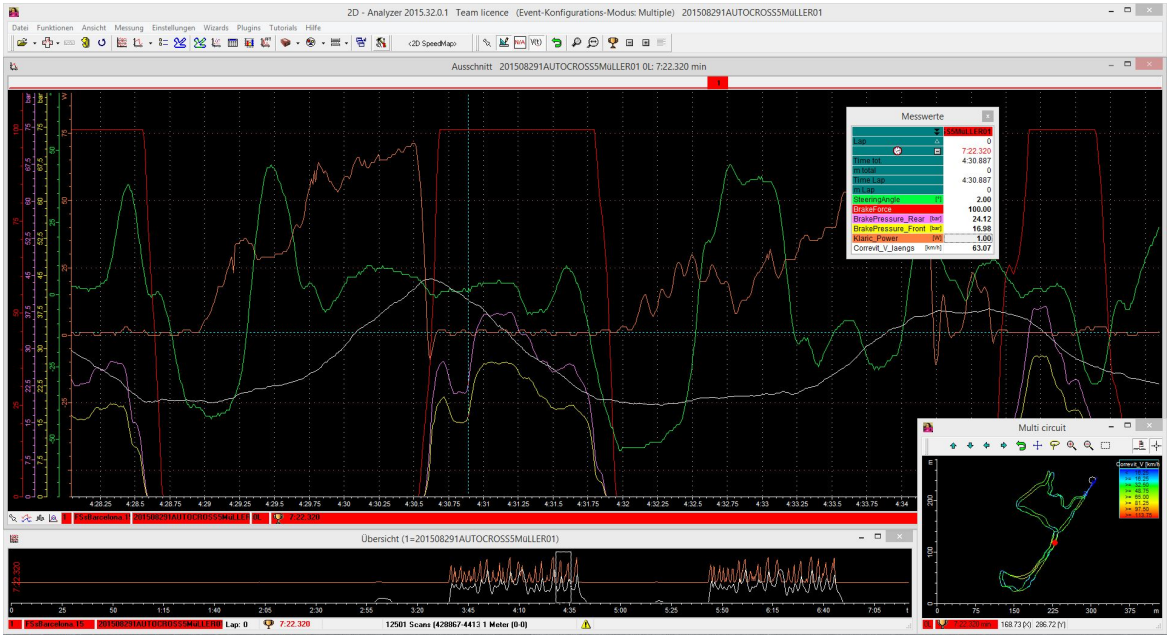


Figure 2.4.: Screenshot of 2D Debus & Diebold Meßsysteme GmbH WinARace

Usability Usability is limited by licensing. The computer running the respective telemetry and/or data analysis software needs to be licensed by the vendor of either system. The licensing argument is crucial: It is highly likely that the team will at some point want to switch away from a particular ECU/VCU vendor, which would mean losing licensing for the respective tool kits. For the 2016 season, the team's electric car division is switching to a VCU from a different vendor, thereby losing the dSPACE licensing required to make use of the MicroAutoBox II's telemetry capabilities. This is, however, currently the only possibility of implementing telemetry on all the recent cars. Without the tool kits, the older cars cannot be operated any longer despite being fully functional because no supervision of critical parameters is possible.

An additional limit is complexity. Currently, the level of expertise a car operation supervisor needs is very high. There is usually only one group capable of operating a combustion car and one group capable of operating an electric car available. Therefore these groups are required to operate all cars which are running even if they are running at the same time, which for special occasions such as marketing events can be up to six cars. This greatly limits the possibility of operating cars for promotional use.

2.5. Conclusion

At present, the team has setups for live telemetry and logging in place, although they suffer from a few problems. The technology base is established: the data generated by the cars is communicated between the ECUs via CAN buses and telemetry is achieved by implementing remote control of the respective central ECU via Wi-Fi. To conclude, there are currently three main problems, listed hereunder.

1. Real-time bidirectional telemetry is limited to the central ECUs and their functionalities.
2. Several different proprietary toolchains are required for logging and analyzing data.
3. The level of know-how required to operate cars is too high.

Track operations engineers are able to work with the current setups, but they are not satisfied with their stability, even though they are not able to precisely and quantitatively define how often and in what manner the systems fail to provide their functionality. For both cars, the telemetry setups work as remote control systems for the respective central ECU. For the PWe6.15, the telemetry system "works well as long as the connection doesn't break", indicating a software problem in the VCU when wirelessly bridging the connection to the control software. However, it is not known when or why these connection losses occur. The wireless bridge, provided as a Wi-Fi connection by the Ubiquiti Bullet™, is not at fault. The team stated having verified this by monitoring signal strength during car testing. The data provided by Braune in his work [Bra14] allows for the same conclusion.

Furthermore, the current data handling configuration only allows to process data using proprietary tools. They are not necessarily negatively perceived by engineers, but the limitation given by licensing is an unresolved issue. Should the team decide to change parts of their tool chain, the lack of basic telemetry could render otherwise operational cars inoperable. A simplification of the tool chain is also seen as beneficial especially by the electric car design group, who want to work with MATLAB®/Simulink as much as possible to simplify their toolchain. MATLAB®/Simulink licensing is not perceived as a problem since the tool is licensed through Formula Student Germany and thus guaranteed to be available independent of ECU vendor choice.

Lastly, the limiting factor for car operation beyond testing and competition is currently the lack of personnel capable of operating cars. This is a situation the team management is very unhappy with as it would be much in favor of more promotional use of operational cars, since organizing show runs and sponsor drives is very popular with sponsors and manufacturers.

3. Requirements Analysis

The term *Requirements Analysis* describes a process which produces the framework for developing a system. It has a set of inputs, among which are the technology base and the outputs from a prior development effort as outlined in chapter 2. The remaining inputs will be identified in this chapter, followed by a structured analysis resulting in a set of well-formed system requirements.

3.1. Methodology

The methodology used in this chapter is described in IEEE P1233 [syr98], "IEEE Guide for Developing System Requirements Specifications". This text defines guidelines on how to identify and organize requirements in a System Requirements Specification (SyRS) which is appropriated by this thesis. These guidelines make it possible to develop a set of requirements which have the following properties as listed in the guide (list is a direct quote):

1. *Unique set.* Each requirement should be stated only once.
2. *Normalized.* Requirements should not overlap (i.e., they shall not refer to other requirements or the capabilities of other requirements).
3. *Linked set.* Explicit relationships should be defined among individual requirements to show how the requirements are related to form a complete system.
4. *Complete.* An SyRS should include all the requirements identified by the team, as well as those needed for the definition of the system.
5. *Consistent.* SyRS content should be consistent and noncontradictory in the level of detail, style of requirement statements, and in the presentation of the material.
6. *Bounded.* The boundaries, scope, and context for the set of requirements should be identified.
7. *Modifiable.* The SyRS should be modifiable. Clarity and nonoverlapping requirements contribute to this.
8. *Configurable.* Versions should be maintained across time and across instances of the SyRS.
9. *Granular.* This should be the level of abstraction for the system being defined.

These requirements are developed by evaluating the following process inputs:

- Technology base and prior development as described in chapter 2.

- Customer raw requirements of functionality
- Environmental influences from the categories
  - Operational scenarios
  - Interfaces with other systems
  - Legal constraints
  - Constraints imposed by technical policies
  - Constraints imposed by the physical environment
  - Competition in the marketplace
- Customer feedback when refining raw requirements
- Technical feedback

These inputs make it clear that identifying requirements is a process which includes a feedback loop between the analyst and the team, which in this case is the customer. The resulting requirements collection is then used to build well-formed requirements, which are subsequently organized into the following categories:

- Functions: What the system has to do.
- Performance: How well the functions have to be performed.
- Interfaces: Environment in which the system will perform.
- Other requirements and constraints.

The result of these efforts is a list of well-formed requirements which serve as the basis of system development.

3.2. Customer Raw Requirements

When it comes to system development and requirements specifications, the team is the central agent of control. As such, it gives a first set of raw requirements, which includes the first system concept and user stories developed from interviewing key stakeholders on the team side. In this analysis, *team* is therefore synonymous with *customer*.

3.2.1. System Concept

The customer has formulated a system concept as follows:

> *"We would like to replace our current live telemetry setups with a flexible solution we can maintain and sustain for the foreseeable future. The system should be able to store and wirelessly communicate data generated by the cars while enabling us to become independent from licensing. The current setup unfavorably limits us to a set of ECU vendors and incorporates tools we feel may not be sustainable in our toolchain."*

This statement reflects the problems identified in chapter 2.

3.2.2. User Stories

The collection of the team's raw requirements cannot be compiled without interviewing the users of the potential system. Since it was not clear from the start who would be the users, the major stakeholders were asked and from their statements and in accordance with them, the following user stories were developed. They identify the user's expectations towards the system.

Maximilian Preis, CTO PWe6.15

"As a technical director, I want as much consistency and simplicity for the toolchains we use as possible in order to achieve the maximum possible efficiency. As we rely on MATLAB®/Simulink for the software design of the car and therefore need it in our toolchain, it would be preferable to be able to use it for other tasks as well in order to save training time for engineers and enable direct data transfer."

During the engineering design process, it is often important to make the best possible use of the time available. The technical director therefore tries to optimize time usage by simplifying the toolchain. If it incorporates tools the engineers already know, time is saved.

Christian Rosenmüller, High Voltage System PWe6.15

"As a track engineer, I want a simple workflow to get data from the car to the analysis tool and a robust and flexible telemetry system which provides me with the data I need to operate the car. While having tools like 2D WinARace available is nice, it is also often important to have the data as raw as possible when debugging at the track."

Testing sessions are constrained in terms of time because track time is limited. Track engineers therefore rely on a fast and robust toolchain to best make use of the time available. However, availability of data is a major concern when debugging, which is why having raw data as well as preformatted data is equally important.

Dominik Schurg, CFO PWe7.16

"As an operations responsible, I want to operate several cars at the same time with small number of skilled personnel. Ideally, a marketing manager is able to supervise car operation after a five minute introduction. This, however, must be possible within a sensible budget."

From a managing perspective, the most valuable resource is workforce: having workforce available alleviates many other concerns. Therefore freeing up manpower is of paramount interest for any Formula Student/SAE team, as long as it is financially viable.

The concerns and expectations of the stakeholders were perceived as being very diverse during the interviews. From the user stories, the following short list of raw customer requirements was compiled:

- Simple, consistent toolchain
- Simple workflow for trackside data analysis
- Robust data transmission while the car is being operated and robust functionality of log transmission
- Simple user interface for supervision of critical data
- Competitive cost efficiency

These expectations, as well as the entire list of requirements, will evolve during the implementation and testing period. The team is an active part of the development process in order to ensure its feedback finds its way into the final product.

3.2.3. Raw Requirements

In conclusion, the team expects the system to:

- get sensory data from the car without relying on vendor licensing.
- write the data to persistent storage.
- transmit the data wirelessly to the track engineers.
- enable the track engineers to wirelessly send data to the car.
- simplify the team's toolchain by integrating the tools it already uses.
- be robust.
- provide a simplified user interface (UI) which can be used by minimally trained personnel safely.
- be cost effective.

As would be expected, these raw requirements describe the system only vaguely and will have to be refined later on. In order to do this, the environment the system operates in has to be defined.

3.3. Environmental Influences

Environmental influences play a central role in every context the system operates in and at every stage of the system's lifecycle. In this case, this means that technical policy constraints are especially important because compliance with the Formula Student/SAE rules is of critical importance to the team. Marketability or cost at scale on the other hand are not relevant and will thus not be analyzed. This chapter collects all relevant environmental influences on the system, analyzes them and derives requirements. A feedback loop with the team was created so as to include its needs directly in the analysis.

3.3.1. Operational Scenarios

The operational scenarios describe the ways in which the system is going to be used and by whom. This is necessary to identify all use cases and users, which is necessary to determine the environment the system will be operated in and thus in determining the environmental influences.

3.3.1.1. Assembly in Car

Although not strictly an operational scenario in which the system is in action, this is still a highly important part of the system's lifecycle. During its life the system will be subjected to this scenario multiple times, possibly even on several different cars, and will be handled by operators of a wide range of skill. It will have to endure this with no failures. The team has also requested the system to work plug-and-play after an initial setup, thus eliminating the need for a recurring setup routine.

3.3.1.2. Car Testing

In car testing the system will gather data generated by the car and provide it to data engineers live and additionally as persistent data upon request. The team has expressed that the system does not need to handle sensory values but rather CAN frames in this operational scenario. The requirement thus is to gather raw data from the CAN bus and transmit it wirelessly to the engineers as well as storing it persistently and provide the stored data upon request. At least part of the system will be on-board the car during operation, meaning that it has to withstand the physical influences in the race car as described by section 3.3.5.

3.3.1.3. Competition

The operational scenario for competition is the same as for testing with the addition that the connection to the car should fulfill an adequate level of information security to keep other teams from "spying", as well as a high level of data integrity to keep outsiders from manipulating data on the car - if only accidentally. As other teams are exposed to the system for about a week during competition, the requirement for the data link security is to be able to withstand attacks for one week before being breached. As for the distance the wireless data link has to bridge, the team requires full coverage of the Formula Student Germany track as this is the design focus for all cars.

3.3.1.4. Promotional Car Use

The operational scenario for promotional car use is the same as for testing. Yet the team states that during promotional car use personnel with minimal training must be able to operate the car safely by being able to monitor a small set of sensory values, e.g. temperature and voltage of the battery, temperature of the motors, temperature of the cooling system. The team wants to be able to configure this set of parameters freely.

3.3.2. Interfaces with other Systems

There are two main interfaces for the system outlined in the following section.

The car. On the car side data is provided on the CAN bus and power is provided through the wiring loom.

- **CAN bus** The CAN buses in use by the team operate at 500kBit/s or 1MBit/s data throughput. To ensure data integrity, the team only puts a 60% load on the bus, corresponding to 300kBit/s and 600kBit/s respectively.
- **Power** The wiring of the cars runs a 5V and a 12V power supply through the car to power ECUs and active sensors. The current provided is 5A.

The user. On the user side the team requires the system to provide an application programming interface (API) for the current tool chain to plug into (e.g. MATLAB®) and a CAN interface acting as a direct interface to the car's CAN bus. The latter was required by the team in order to be able to use the software of ECUs that only allow for communication through CAN bus. Additionally, the team requires a simple user interface for personnel with minimal training and no toolchain knowledge. The system has to provide user configurability for this interface. There is no constraint for power supply on the user side.

3.3.3. Legal Constraints

Compliance with the law must be ensured. Although competitions are held worldwide, the team states that operation outside of Europe does not need to be considered in developing the system as it is a marginal scenario. Nevertheless, users must be aware of potential legal compliance issues outside of Europe. However, this means that European law must be respected when using the system.

As the system will not operate on a road vehicle, the laws from road vehicle context do not apply. There are, however, laws to be respected when transmitting data wirelessly. When doing so one must comply with the European radio transmission laws which limit transmission power for different frequency bands, e.g. 100mW for the 2.4GHz range and 500mW for the 5GHz range, the ranges in which Wi-Fi mainly operates.

Other legal compliance issues arise from the fact that the system's software must respect code licensing of foreign code which is used, e.g. if code is used which is licensed using the GNU public license (GPL) [Foua] the team has to open the source of the system's software. The team has expressed they do not see any issue with doing so.

IC1.13.1	The APPS[1] must be actuated by a foot pedal. Pedal travel is defined as percent of travel from a fully released position to a fully applied position where 0% is fully released and 100% is fully applied.
IC1.13.11	Any algorithm or electronic control unit that can manipulate the APPS signal, for example for vehicle dynamic functions such as traction control, may only lower the total driver requested torque and must never increase torque unless it is exceeded during a gearshift. Thus the drive torque which is requested by the driver may never be exceeded.
EV2.3.2	The torque encoder[2] must be actuated by a foot pedal.
EV2.3.12	Any algorithm or electronic control unit that can manipulate the torque encoder signal, for example for vehicle dynamic functions such as traction control, may only lower the total driver requested torque and must never increase it. Thus the drive torque which is requested by the driver may never be exceeded.

Table 3.1.: Excerpt from FSAE Rules 2016 [Int15]

3.3.4. Technical Policy Constraints

As stated in chapter 1, Formula Student/SAE works according to a set of rules not unlike professional motorsports. There are two documents in this context which are of central importance: the FSAE rules [Int15] and the FSG rules [e.V15]. All competitions the team attends work according to one of the two. Additionally, the FSG rules are an addendum to the FSAE rules.

The FSG rules of 2016 make no mention of telemetry, bidirectional or not, nor do they contain any rules or specification that pose constraints to the system. The FSAE rules of 2016, while not mentioning telemetry of any kind either, contain information concerning the throttle systems. They state that the throttle system must be under full control of the driver at all times. Table 3.1 displays relevant excerpts from sections IC1.13 and EV2.3 of the FSAE rules.

Additionally, the FSAE rules for both categories (internal combustion and electric vehicle) contain the requirement that the plausibility of throttle position sensor (TPS), accelerator pedal position sensor (APPS), torque encoder (also called Throttle Pedal Position Sensor, TPPS) and brake system encoder (BSE) must be verified by a redundant setup. External interference with the system by any means, including a bidirectional CAN telemetry system, would be in violation of the rules, specifically sections IC1.11, IC1.12, IC1.13, IC1.14, IC1.15, IC1.16, EV2.3 and EV2.4.

However, the FSAE rules do not specify the means of data transportation from the respective sensors (TPS, APPS, TPPS, BSE) to the corresponding controllers (i.e. ECUs), allowing for analog or digital transmission of any kind (sections IC1.12.7, IC1.13.8, IC1.14.3,

[1] Accelerator Pedal Position Sensor. Applies to internal combustion engine vehicles. This sensor is mounted to the foot pedal which is used by the driver to indicate a torque request, i.e. a request for longitudinal acceleration force.

[2] Also called throttle pedal position sensor (TPPS). Applies to electric vehicles. Equivalent function to the APPS on the internal combustion vehicles (see above).

EV2.3.9, EV2.4.4). More specifically, no constraints are imposed on the protocol selection either, thus allowing the team to select CAN identifiers for the signals freely.

3.3.5. Physical Environment

Especially the part of the system which is mounted in the car is placed under restrictive constraints imposed by the physical environment. As elsewhere in motorsports, Formula Student/SAE car designers strive for small and light weight, yet reliable solutions. The current systems differ in their design space and weight with the LG-μCAN11_Pro-000 by 2D Debus & Diebold Meßsysteme GmbH being the smallest unit, measuring roughly 48x32x105mm and weighing about 150g according to the data sheet [2D]. These dimensions and weight serve as the requirement for the new system. The Vector GL 2000 has dimensions of 194x137x35mm (weight not specified by Vector Informatik GmbH) but is not used as reference because it is not mounted in the car at all times, i.e. it is not present during competition because of its size and weight.

The system must endure substantial temperatures inside the car. In the electric vehicles the cooling system for the battery reaches the highest temperatures with up to 65°C. In the cars with internal combustion engines even higher temperatures are reached but according to the team ECUs are positioned in the car in such a way that they are cooled by the surrounding air flow, hence they do not need to satisfy a higher temperature requirement than for the electric vehicles. As cars are only run in safe conditions, it can safely be assumed that temperatures experienced by the system will never be lower than 3°C.

Additionally, the system may be exposed to various kinds of fluids. As such, the enclosure of the system and its wiring will have to satisfy the same requirements as all other electronic systems on the cars. In the electric car the additional danger of being subject to electromagnetic compatibility issues exists. Although the high voltage system and the wiring harness of the low voltage system are designed to minimize electromagnetic influence from the beginning (i.e. analog signals are never carried parallel to high voltage wiring, among other things), appropriate shielding is still required. As is typical for race car use, the components on the boards will have to withstand substantial vibrations as well.

However, the part of the system which is not mounted in the car also has to endure a motorsports environment, albeit not as restrictive. The limit in size derives from the logistics of testing a Formula Student/SAE car and the usage at the side of the track. Here, the part of the system not mounted on the car is subject to the same environment the engineers' standard issue laptops are.

It also has to be taken into consideration that in competition, with approximately 3500 students at the Hockenheim circuit, the over-the-air link might suffer connectivity issues due to the dominant presence of interfering devices, e.g. smartphones.

3.3.6. Organizational Constraint

Aside from functional and legal aspects, the team has stated organizational constraints. These are, as indicated by the raw requirements, that the system must be maintainable and sustainable for the team's organization. This means that the system should be developed further after handing it over to the team so the system can be updated in the future to incorporate new functionality or to comply with new and changed rules. In order to facilitate

this, software documentation must be provided together with a short system setup and handling manual in addition to this work.

At the same time, the team wants the system to be cost effective. Based on its familiarity with the cost of microcontroller development platforms and electronic hardware, a target price range of €250.- to €350.- has been mentioned. The team specifically asks to be involved in the process of hardware choice. Flexibility in terms of hardware is seen as a clear advantage, as it allows for independent development of hardware and software after the system has been turned over to the team.

3.4. Well-Formed Requirements

In this chapter, the requirements collected up to this point are refined, resulting in a set of organized, well-formed requirements. This is done by forming simple sentences stating what the system is supposed to do or not which are clear and unambiguous and limited to one function in order not to overlap with other requirements. The words MUST, MUST NOT, SHOULD and SHOULD NOT are used as defined by RFC 2119 [Bra97]. Together with the team, the analyst develops performance measures for the requirements, thus ensuring testability. The requirements list is organized in three categories: functional performance, legal and policy constraints, physical characteristics.

3.4.1. Functional Performance Requirements

The functional performance requirements derive from the raw user requirements outlined in section 3.2 as well as the operational scenarios 3.3.1 and the interfaces with other systems 3.3.2, both described in section 3.3. The team has been involved in identifying suitable performance measures for each functional requirement.

Data extraction from the car. The CAN buses used on the cars operate at a maximum data throughput of 1Mbit/s with a load of 60%, corresponding to 600kBit/s of data. The system should be able to read this amount of data from the CAN bus.

Persistent data storage. The data read should be persistently stored in a machine readable format for further use in the team's preexisting toolchain. The team has requested that the system store data of an entire day of testing, which at a maximum of 6 hours amounts to 1.62GB of data in raw form, overheads and data format not considered.

Extraction of stored data. The team has requested that data extraction should be possible in under one minute per 10 minute run. With a 10 minute run amounting to 45MB of raw data, this means a transfer throughput of at least 6MBit/s, overheads and data format not considered.

Wireless transmission of data. At the core of bidirectional telemetry is the possibility of getting data from and to the car during a run. The system should communicate the data read from the CAN bus wirelessly to the trackside engineers and from them to the car at every point on the Formula Student Germany track in Hockenheim, thus covering a distance of 250m (see figure 3.1, big yellow circle). The team has requested that a flexible solution is provided to handle a potential data throughput drop on the wireless link which enables the team to determine which data to drop.

Presentation of data. The data which has been transmitted to the trackside personnel must be provided in such a way that 1. the team can use his preexisting toolchain to evaluate the data and 2. that the team has a toolchain independent option of displaying a small set of mission critical data, thus enabling minimally trained personnel to supervise the cars during promotional use.

Bidirectional communication. The team has required that the system is able to communicate with the car bidirectionally through the toolchain and a CAN interface available to the trackside personnel.

Plug and play. The system should be capable of plug and play operation after an initial setup.

Security. During competition the system will be exposed to potential attackers for a maximum of one week. Thus the security of the wireless data link must be strong enough to withstand attacks for one week to ensure data integrity.

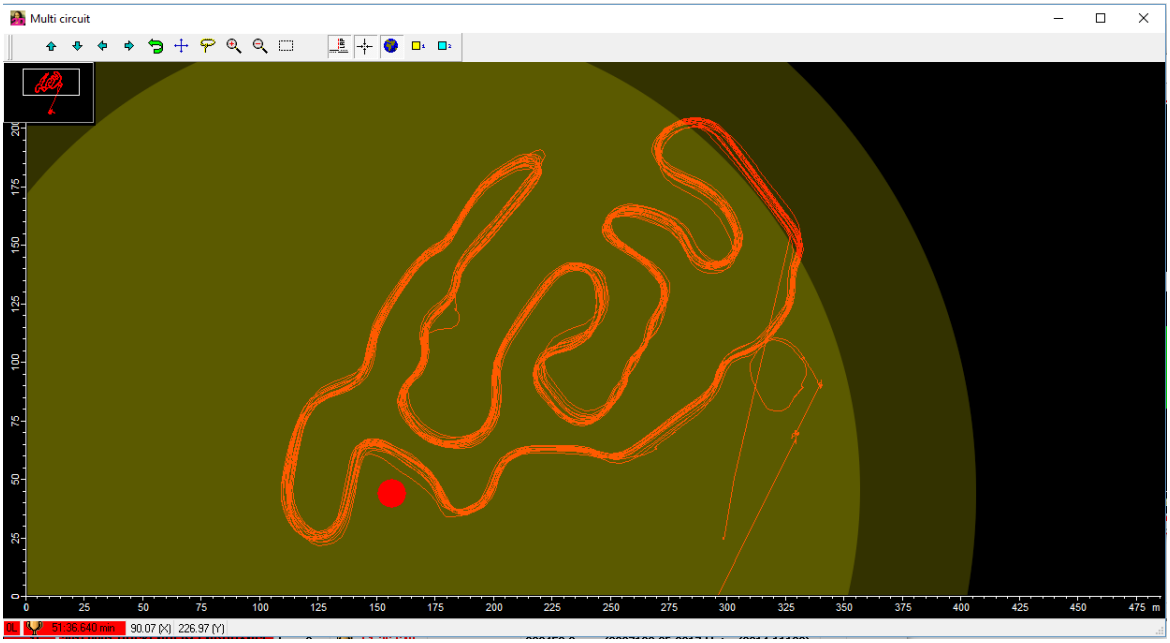


Figure 3.1.: Formula Student Germany track in Hockenheim as provided by the team. Red dot is base station location, yellow circles are 200m (small) and 250m (big) radius.

The well-formed requirements for functional performance are listed in table 3.2 in section 3.4.4.

3.4.2. Legal, Policy and Organizational Requirements

The legal and policy constraints are outlined in sections 3.3.3 and 3.3.4. Organizational constraints are described in section 3.3.6. The well-formed legal, policy and organizational requirements derived are listed in table 3.3 in section 3.4.4.

3.4.3. Physical Requirements

The physical requirements of the system are derived from the physical environment it will perform in, described in section 3.3.5. Since a difference needs to be made between the part of the system mounted in the car and the part not mounted in the car, a distinction in terms is necessary. The part of the system mounted in the car is thus called vehicle module (VM in short) in this section, while the part not mounted in the car is referred to as trackside module (TM in short). The requirements as derived from section 3.3.5 are:

- Upper size and weight limit VM: 48x32x105mm - 150g
- Temperature range in which VM has to operate: 3°C to 65°C
- Resistance to fluids and vibration in car (VM)
- Electromagnetic compatibility, especially with the electric cars (VM)
- VM must endure repeatedly being mounted in and dismounted from cars
- Small and light enough to avoid causing additional logistical effort (TM)
- Basic resistance to weather (TM)
- Data link quality must not suffer in competition with a great number of interfering devices (e.g. smartphones, other cars) present

The well-formed physical requirements derived are listed in table 3.4 in section 3.4.4.

3.4.4. Tables of Well-Formed Requirements

This section summarizes the well-formed requirements as described in sections 3.4.1, 3.4.2 and 3.4.3. The requirements are grouped in three tables, one containing the well-formed functional performance requirements, another the well-formed organizational, legal and technical policy requirements and the last one the well-formed physical requirements. These tables are reproduced later in this work for the purpose of visual reference when analyzing requirement fulfillment.

The system MUST read data from a CAN bus at a throughput rate of 600kBit/s.
The system MUST be able to persistently store 1.62GB of raw data in real time.
The system SHOULD allow extraction of stored data at a throughput rate of 6MBit/s.
The system MUST transmit data wirelessly over a distance of 250m.
The system MUST provide the team with a flexible solution to prioritize data.
The system SHOULD provide the team with a flexible API for its preexisting toolchain.
The system SHOULD provide an external CAN interface to the car's CAN bus.
The system MUST be able to communicate with the car bidirectionally.
The system SHOULD enable unskilled personnel to monitor mission critical data.
The system SHOULD work plug and play after an initial setup.
The system MUST withstand attacks on the wireless data link for one week.

Table 3.2.: Well-formed functional performance requirements

The system MUST NOT exceed the transmission power limit set by European law.
The system's software MUST respect code licensing if foreign code is used.
The system MUST NOT interfere with throttle/accelerator pedal or brake pedal signals to comply with Formula Student/SAE rules.
The system's documentation MUST enable further development after handover.
The system SHOULD cost between €250.- and €350.-.

Table 3.3.: Well-formed organizational, legal and technical policy requirements

The VM SHOULD NOT be bigger than 48x32x105mm.
The VM SHOULD NOT weigh more than 150g.
The VM SHOULD operate in a temperature range between 3°C and 65°C.
The VM SHOULD be resistant to fluids.
The VM SHOULD be electromagnetically compatible with all cars the team uses it in.
The VM MUST endure vibrations common in the team's cars.
The VM MUST endure repeatedly being mounted in and dismounted from cars.
The TM MUST be small and light enough to not cause additional logistical effort.
The TM SHOULD be weather resistant.
Data link quality SHOULD NOT suffer from a great number of interfering devices.

Table 3.4.: Well-formed physical requirements

4. System Design

The system design process is divided in two fundamental steps: the functional analysis, where required functionality and the resulting functional structure is analyzed in detail, and the design synthesis, where the options in hardware selection are outlined and a software architecture is developed. Implementation options are suggested, but no specific solutions determined. The result is a system blueprint capable of providing the basis for the implementation.

4.1. Methodology

The functional analysis (section 4.2) describes the functional flow and characteristics as well as the data flow of the system in detail. The requirements developed in chapter 3 serve as inputs, while a description supplemented by visual presentation of the system's function are developed as outputs. These serve in turn as inputs for the design synthesis (section 4.3), where they are used to develop a suitable architecture for the system and its components. Closing the chapter will be section 4.4 with an analysis of the aspects of the system design and their relation to the given requirements.

4.2. Functional Analysis

In order to arrive at a complete representation of the functions offered by the system, the functions have to be logically grouped. Thus, section 4.2.1 describes only the operational functionality of the system, i.e. everything the system does when it is in active operation. Section 4.2.2 then outlines the maintenance functionality, such as assembly and system setup, as well as the physical functions.

4.2.1. Operational Functionality

The functionality requirements have been specified in chapter 3. Now the functional flow containing these requirements must be analyzed. Figure 4.1 shows the basic configuration of actors the system deals with (cp. section 3.3.2). In order to analyze the functions which need to be performed by the system to bridge the gap ("Wireless Data Link"), functionality is broken down top-to-bottom and visualized in functional flow block diagrams (FFBD). This enables the analyst to review the system at the granular level and to logically group functions, thus producing the functional structure which is necessary to develop hard- and software architecture in section 4.3.

Since the communication between car and trackside personnel is bidirectional, two top level functional flow block diagrams are necessary. To avoid misunderstanding, the functional flow transporting data from the car to the trackside personnel is called *"telemetry"* and the functional flow transporting data from the trackside personnel to the car is called *"remote control"* in this section.

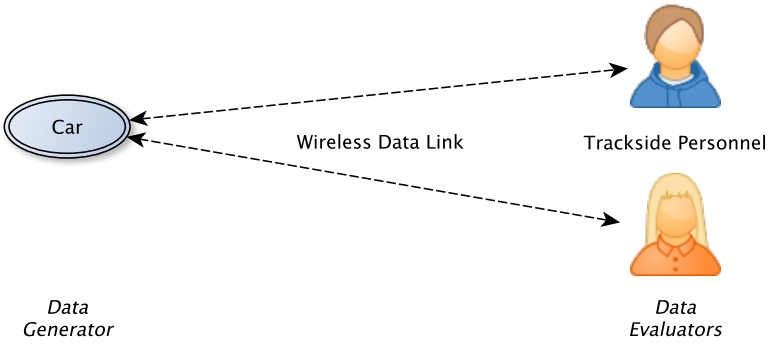


Figure 4.1.: Basic configuration of actors

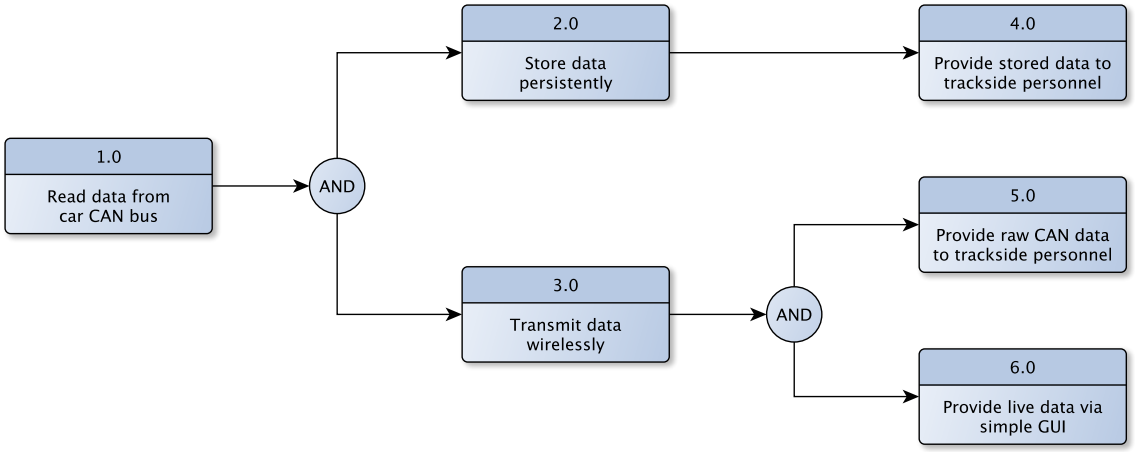


Figure 4.2.: FFBD[1] Block Diagram of Telemetry function

4.2.1.1. Telemetry

Figure 4.2 shows the top level representation of the functional flow for the telemetry in a functional flow block diagram (FFBD). The main functions are displayed as reading from the CAN bus and providing the data to the trackside personnel as per requirements definition 3.2.3.

A more detailed analysis of the steps necessary between reading from CAN bus (1.0) and storing persistently (2.0) as well as wirelessly transmitting (3.0) is laid out in figure 4.3. The diagram shows parallelism once the type of message read from the CAN bus is determined.

As for the steps between storing data persistently (2.0) and providing the data from storage to the team (4.0), laid out in figure 4.4, the FFBD is substantially simpler than figure 4.3. Note that there is no wireless transmission involved as it has not been requested by the team. Interviewing the team has revealed that this is currently done via cable and is not seen as obstructive but it would be welcome to have the possibility to perform this step wirelessly. However, it was mentioned that this is seen as insubstantial to the success of the system.

[1]Functional Flow Block Diagram, used in systems engineering to visualize the flow of functionality in blocks without modeling design or architecture (as described in [Pre01], Supplement 5-A).

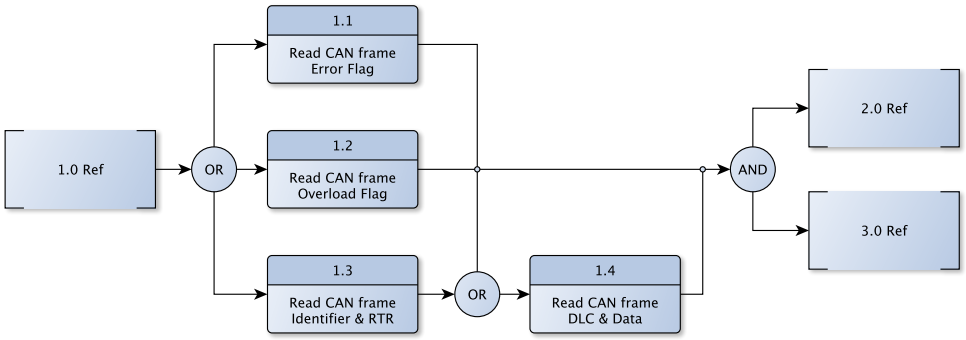


Figure 4.3.: FFBD[1] of functionality from 1.0 to 2.0 and 3.0 from figure 4.2

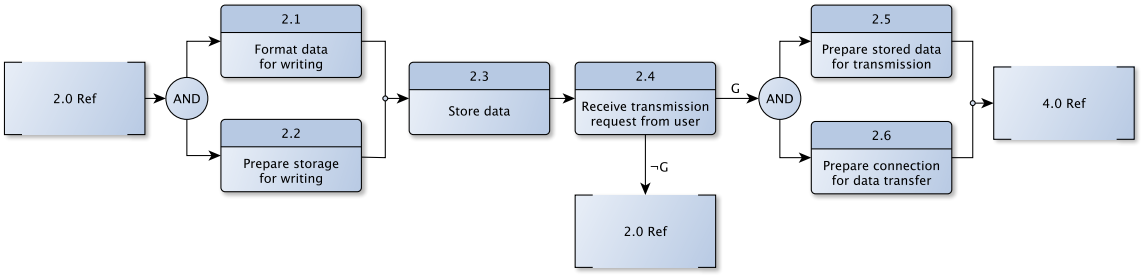


Figure 4.4.: FFBD[1] of functionality from 2.0 to 4. from figure 4.2

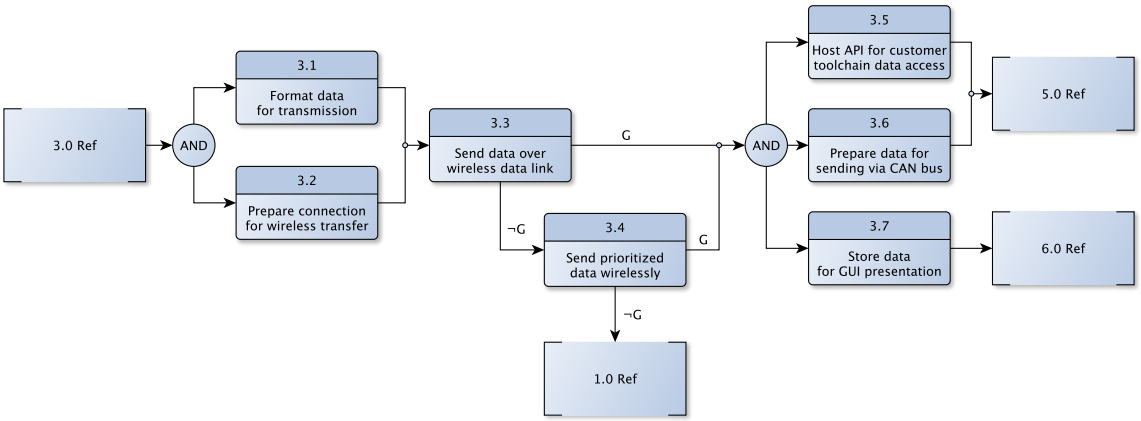


Figure 4.5.: FFBD[1] of functions from transmitting data wirelessly to presenting it to the user

The steps necessary for wireless transmission of data (3.0) to providing access to the car's CAN bus for trackside personnel (5.0) and providing live data via a simple GUI presentation for marginally trained trackside personnel (6.0) are visualized in figure 4.5. The function to prioritize data in case the wireless link is not able to transmit data or cannot be trusted to do so is visualized through a GO/NO-GO fork. If transmitting the prioritized data wirelessly fails, the strategy is to go back to reading from the CAN bus (1.0) and try again next time. Parallelism occurs close to data presentation to the user since different interfaces are provided.

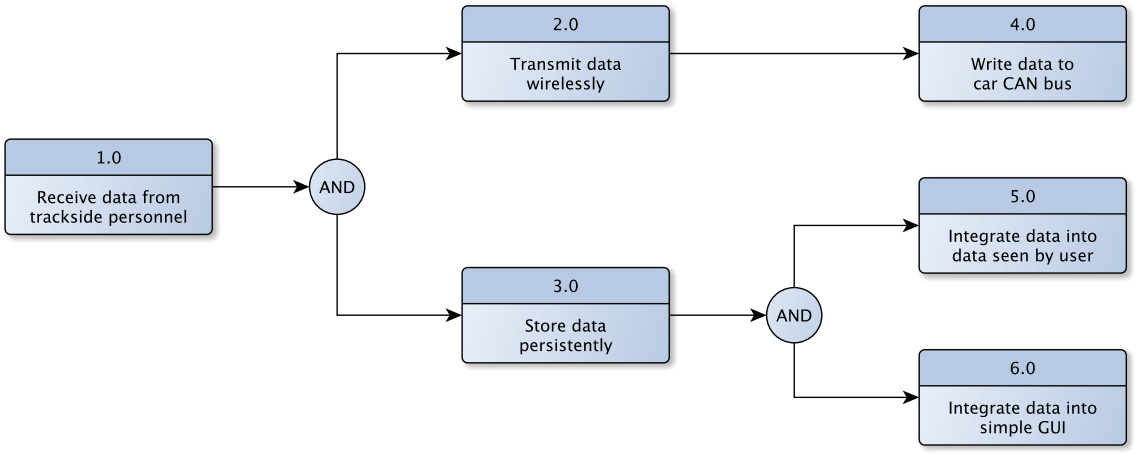


Figure 4.6.: FFBD[1] of remote control function

4.2.1.2. Remote Control

Figure 4.6 shows the top level functional flow block diagram for the remote control functionality, the set of functions allowing trackside personnel to send data to the car to gain remote control of ECUs. Storing the data persistently is also required on the user side in order to integrate sent data in the user interfaces. This way, the team can reconstruct at a later point in time what data was sent to the car without having to rely on the wireless connection alone.

The steps necessary to receive data from trackside personnel (1.0) to transmitting it over the wireless data link (2.0) and storing it persistently (3.0) are visualized in figure 4.7. Note that also here, reading from CAN bus is necessary since the requirements specify remote access via CAN interface. In order to comply with the legal constraints stated in section 3.4.2, a filter for messages which inhibits messages that would break compliance ("blacklisted content") from being written to the CAN bus must be provided.

In figure 4.8, the steps to sent user generated data to the car's CAN bus are pictured. The GO/NO-GO fork at the point of wireless transmission (step 2.3) shows that in the case of unsuccessful transmission the user is notified that his data could not be sent and then the system expects another set of data from the team.

The functions necessary to persistently store data sent by trackside personnel in order to be able to offer it through the user interfaces are depicted in figure 4.9. Storage, at least intermittently, is necessary to provide time-series based data for a run.

4.2.1.3. Conclusion of Functional Flow Analysis

The functional flow block diagrams show all functions as stated in the functional performance requirements (see section 3.4.1). Missing from the functional flow diagrams is the possibility for the user to set the filter for data that must not be sent to the car for legal and policy compliance reasons, as well as the possibility for the user to set the filter for prioritizing data in the case of the wireless link only offering limited data throughput. These functions do not have an impact on the overall functional flow, nor do they influence the data flow in the system. However, they are necessary for fulfilling the requirements and are thus a part

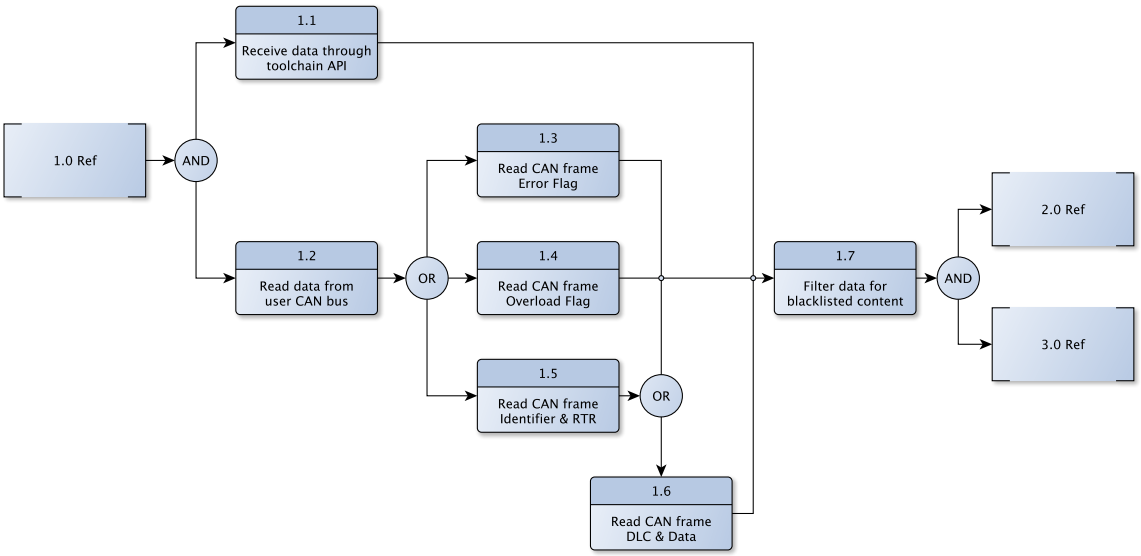


Figure 4.7.: FFBD[1] of functions necessary to get data from the user and process it further

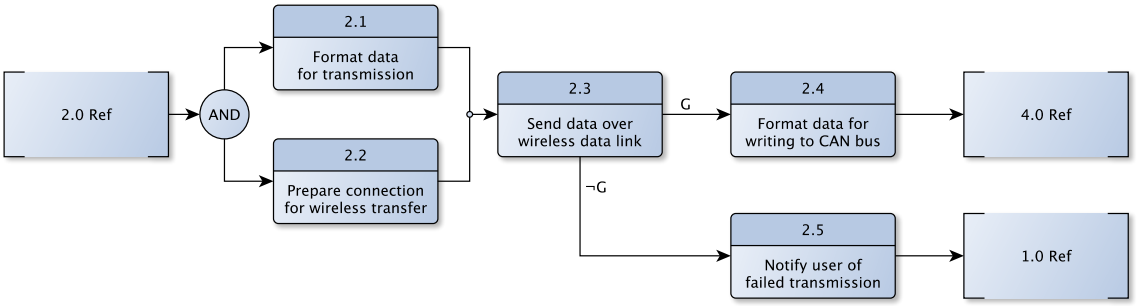


Figure 4.8.: FFBD[1] of functions necessary to transmit data wirelessly and write it to the car CAN bus

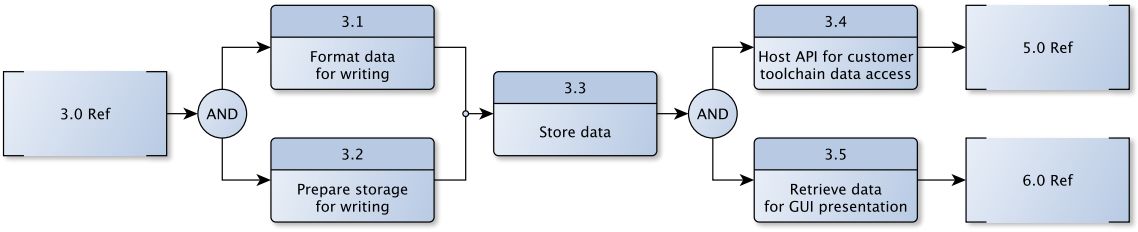


Figure 4.9.: FFBD[1] of functions necessary to integrate data into user interfaces

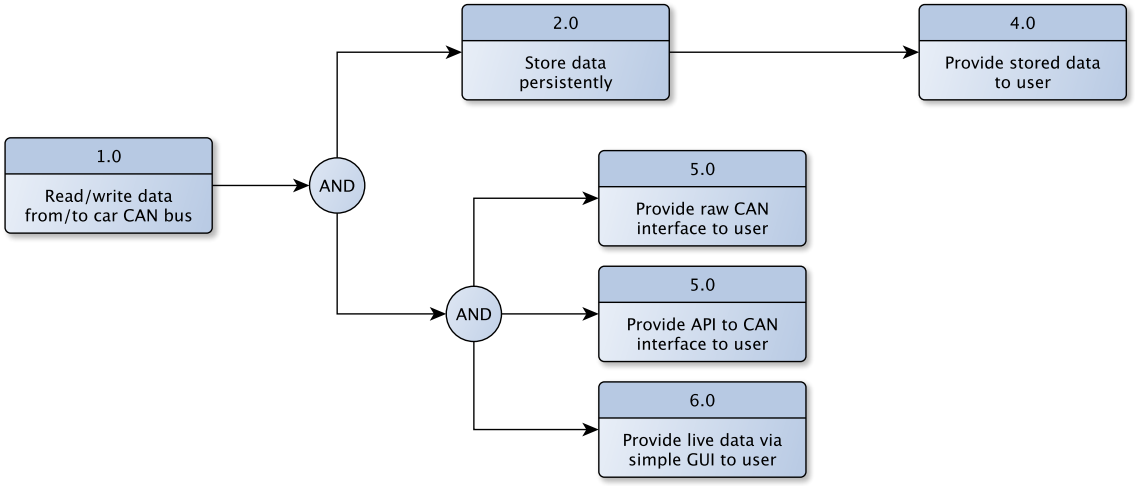


Figure 4.10.: FFBD[1] of system abstraction

of the functional diagram in section 4.2.3.

There is furthermore a striking symmetry between the *telemetry* and the *remote control* functionality: Both read data from CAN bus, store it persistently and send it over a wireless link simultaneously. The remote control scenario goes beyond this by also reading from other sources, but since this is done to talk to the car's CAN bus on the other end, the data is translated to CAN protocol format eventually. Both receive data through a wireless data link, store it persistently and write it to a CAN bus interface. In both cases, stored data is provided to the team upon request, although the telemetry scenario expands on this by also providing the team with interfaces to live data.

As an abstraction, consider the functionality visualized in figure 4.10. Depicted is a set of functions which enable logging and monitoring of a CAN bus without sending data over a wireless link. This is, at its core, a classical producer-consumer pattern: the CAN bus produces a time series of data which the user wants to consume, ideally in exactly the same order but at least in approximately the same order in which it was produced.

4.2.2. Maintenance and Physical Functionality

Maintenance and physical functionality mainly consists of mounting, dismounting and remounting the system multiple times and plugging and unplugging the physical interface or interfaces on every occurrence. The requirements state that this must be possible for the system without taking physical damage, but this cannot be displayed or more thoroughly analyzed with the help of a functional flow block diagram.

However, there is a functional requirement to be considered: the plug-and-play functionality. The steps for plugging in the system on the car side are shown in figure 4.11, while the steps for starting the user side part of the system are shown in figure 4.12.

4.2.3. Functional Analysis

With the functional flow established, functional groups and their relationship can be identified and visualized in a full system overview. Figure 4.13 shows the full operational func-

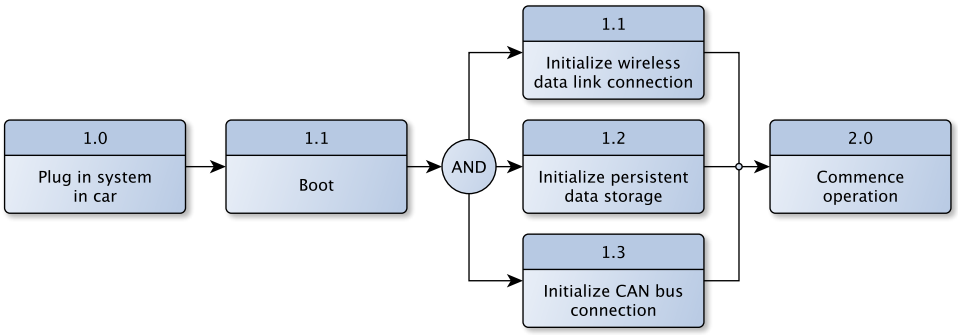


Figure 4.11.: FFBD[1] of plugging in the system on the car side

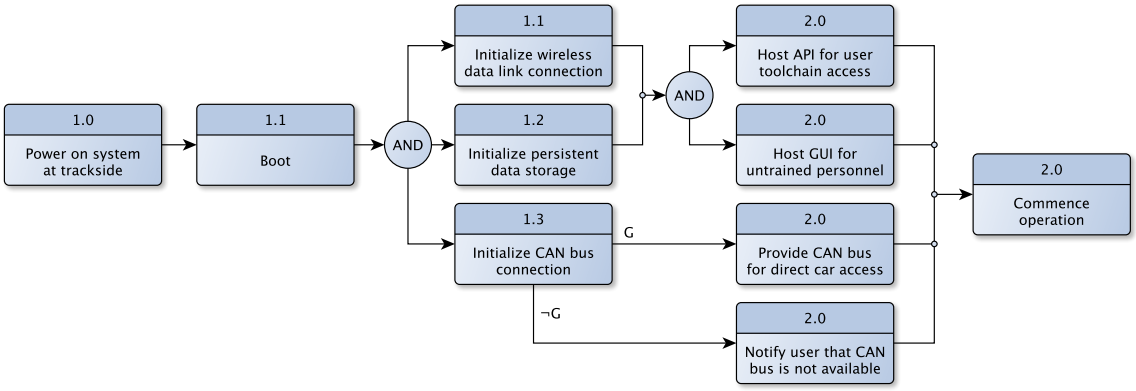


Figure 4.12.: FFBD[1] of plugging in the system at the trackside

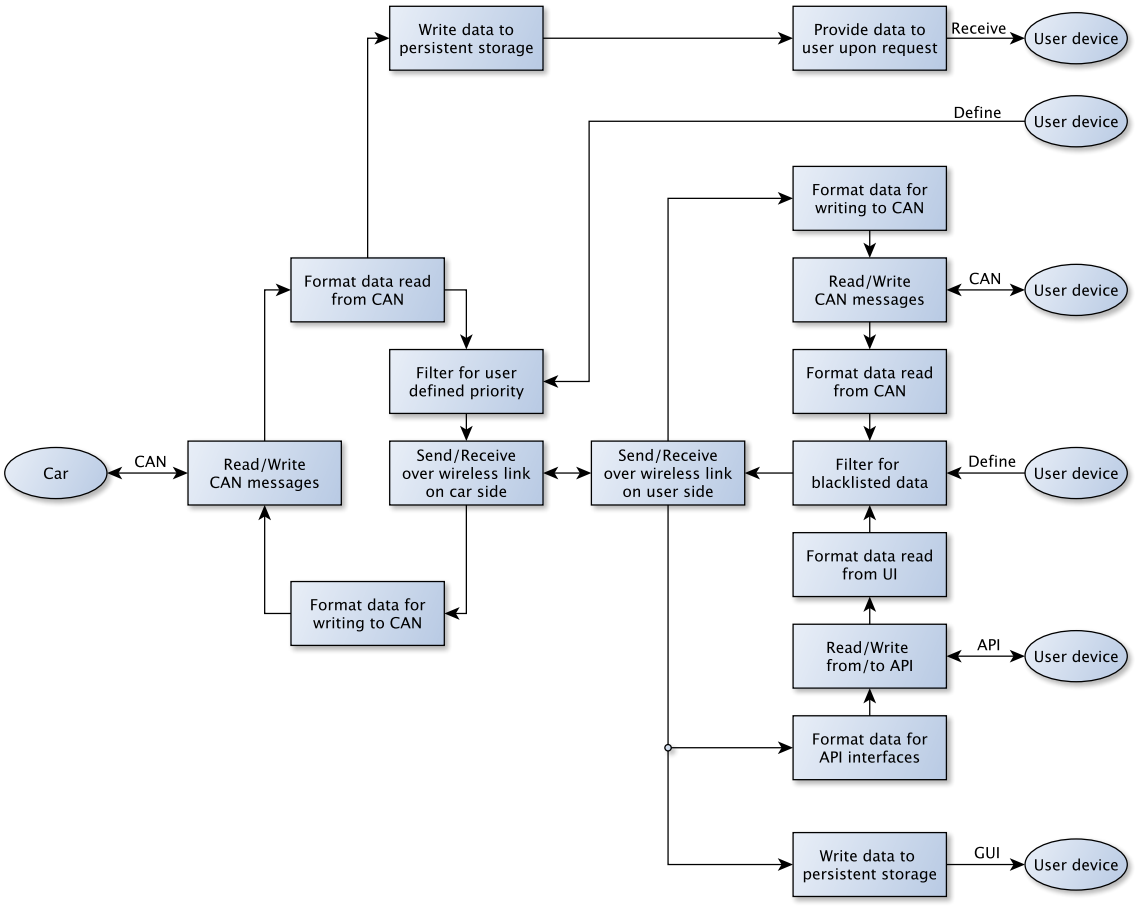


Figure 4.13.: Full system functional block diagram

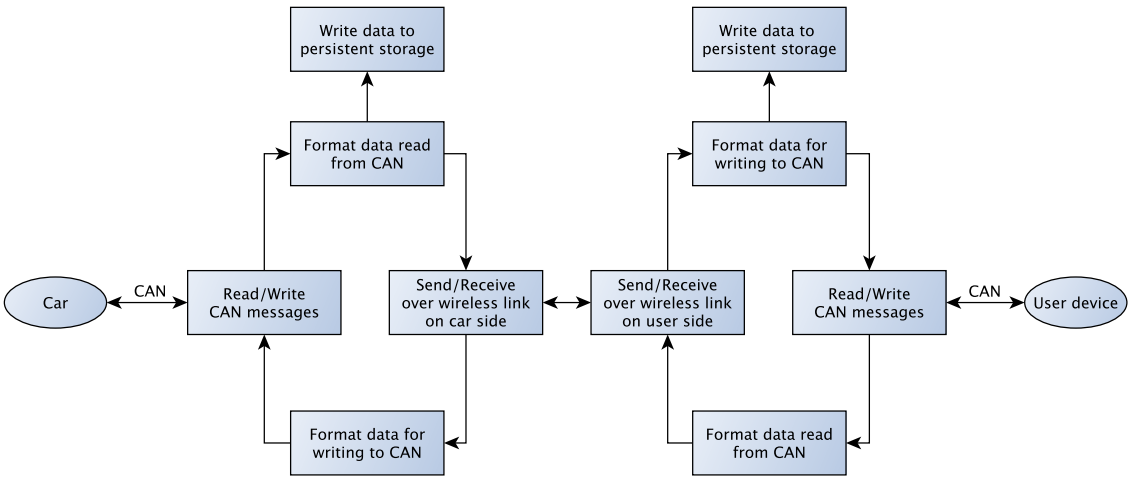


Figure 4.14.: CAN bridge functional block diagram

tionality of the system. Note the user definable filter to blacklist data which would break compliance with legal and technical policy constraints as specified in section 3.4.2 and the prioritization of data sent over the wireless link as specified in section 3.4.1.

The block diagram in figure 4.13 clearly shows that there is one interface to the car - the CAN bus - and many to the user, of different functional scope. However, consider the symmetry of the diagram in figure 4.14. This eliminates all user interfaces except the CAN bus one and is thus completely symmetrical. The resulting diagram outlines the functionality of a pure CAN bridge with data storage, which is a major part of the required functionality.

4.3. Design Synthesis

In this section, the process of synthesizing the analyses from the previous chapters is described. In order to arrive at the software architecture and the hardware selection, the functionality blocks from figure 4.13 in section 4.2.3 have to be logically grouped in the schematic block diagram of the system in section 4.3.1. The software architecture is then defined in section 4.3.2, which includes defining the data structure 4.3.2.1 and describing the viable options for creating an API platform 4.3.2.2 to be used by the team's toolchain. Finally, in section 4.4, a requirements feedback loop will determine how the requirements relate to the proposed solution.

4.3.1. Schematic Block Model

Using the functional block diagram of the full system shown in figure 4.13, the schematic block diagram groups functionality in logical blocks. For the schematic block diagram, the separation between car side and user side is practical for better overview. Note that the wireless link is on the right side in both diagrams.

Figure 4.15 shows the functionality required to cover the operational functional requirements on the car side. The team does not require a specific format for raw data, so it is up to the implementation to choose a suitable format. The same is true for the user side schematic, shown in figure 4.16.

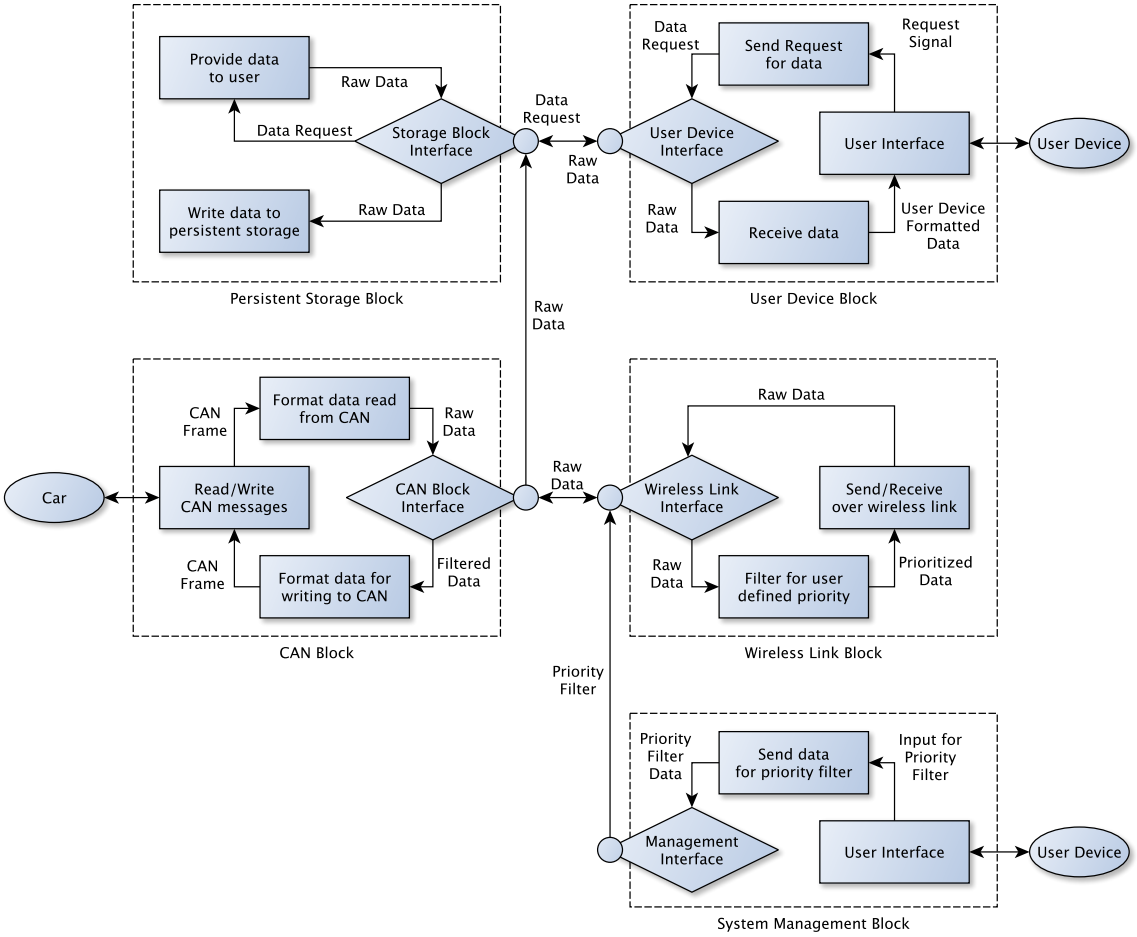


Figure 4.15.: Schematic block diagram of car side functionality

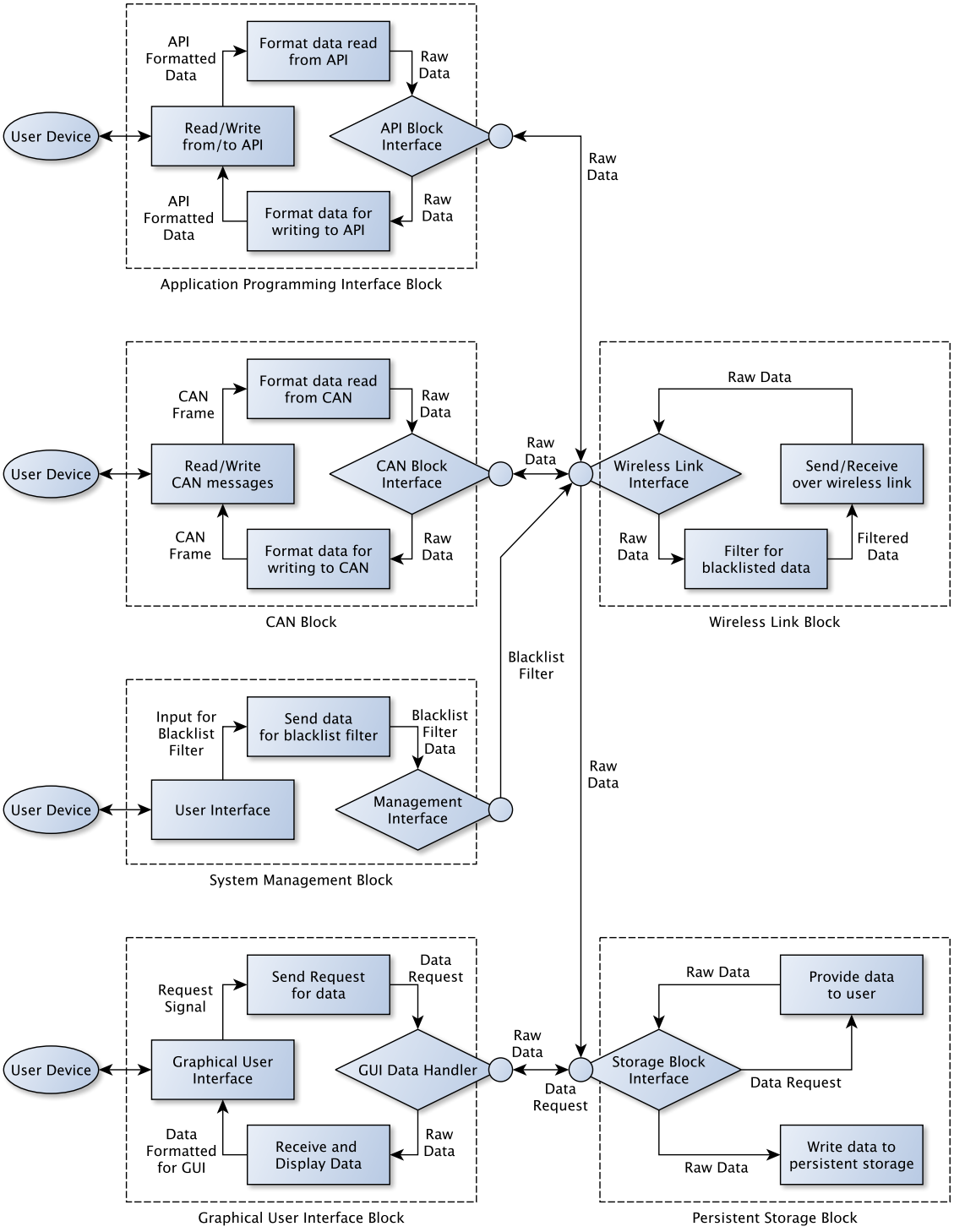


Figure 4.16.: Schematic block diagram of user side functionality

4.3.2. Software Architecture

In this section, the software architecture of the system is discussed. The schematic block model serves as the basic layout. Coming back to the symmetry observed in section 4.2.3, a unified software architecture is developed. This architecture, when implemented, will run on the car mounted part of the system as well as on the trackside part. The blocks defined in the schematic block model of the user side system part as shown in figure 4.16 are used in figure 4.17 but extended to include the functionality required by the car side (figure 4.15). Thus, a unified architecture emerges.

Note that some functionality has moved in the functional flow. The distribution block now takes care of internally passing data around. Filtering for blacklisted data is now done when receiving through the wireless link, not before sending. Storing persistently is now done whenever data is received through any interface, thus creating a full log of all data flow through the system. These changes are made possible and at the same time necessary through the architecture unification:

- Both parts of the system now have separate blacklist and priority filters, so they can and need to do both on both ends.
- Storing data is no longer practical for incoming and outgoing data separately. Instead everything passing the distribution block is now stored.

4.3.2.1. Data Structure

The data is structured as CAN frame when it enters the system. The information included is thus:

- Boolean: Error Frame (ERR)
- Boolean: Remote Transmission Request (RTR)
- Boolean: Extended Arbitration Identifier (IDE)
- Bit sequence (11 or 29 Bits): Message Identifier (MID)
- Integer (between 0 and 8): Data Length Code (DLC)
- Byte array (length DLC): Data Field (DF)

Note that in the case of error and overload frame, the transmitted flag does not contain any information. Thus error and overload frame can be grouped together as they have the same structure and differ only in when they are transmitted.

For the purposes of this system, more information than is contained in the raw CAN frame is required, namely:

- Timestamp: Timestamp (TS)
- String: Bus Identifier (BID)
- String: Device Identifier (DID)

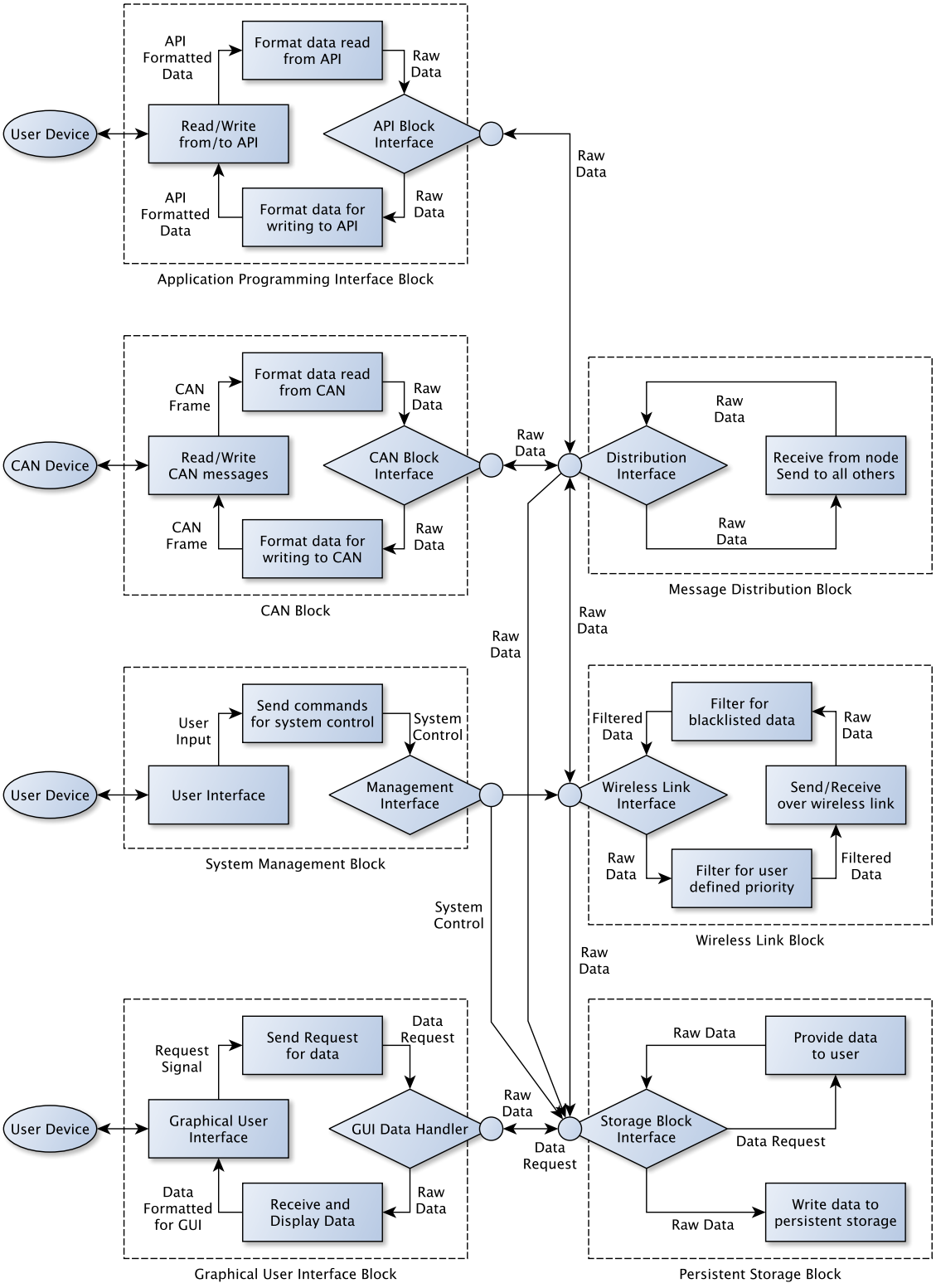


Figure 4.17.: Block diagram of unified software architecture

A timestamp is required to track data as time series. The bus identifier is necessary to clearly identify the CAN bus interface the system was reading from by name. The team has expressed the wish to name the bus (CAN1, CAN2 and so on; see section 2.3) in order to remain consistent with the current setups and wants stored data to contain this information. The device identifier is necessary to identify the source of messages. The message (MSG) structure is thus as follows, using the abbreviations defined above:

$$
\begin{aligned}
\langle MSG \rangle &::= \langle DID \rangle \ \langle BID \rangle \ \langle TS \rangle \ \langle CONTENT \rangle \\
\langle CONTENT \rangle &::= \langle ERR \rangle \mid \langle RTR \rangle \ \langle IDE \rangle \ \langle MID \rangle \mid \langle IDE \rangle \ \langle MID \rangle \ \langle DLC \rangle \ \langle DF \rangle \\
\langle DLC \rangle &::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \\
\langle DF \rangle &::= \langle BYTE \rangle \mid \langle BYTE \rangle \ \langle DF \rangle \\
\langle BYTE \rangle &::= 0 \mid 1 \mid ... \mid 255
\end{aligned}
$$

Note that the number of BYTE in DF must correspond to DLC.

A machine readable data format which is able to represent this structure must be chosen. The format must further enable transmission over a wireless network connection and writing to file. Finally, the team's toolchain has to be considered in choosing the data format.

4.3.2.2. Application Programming Interface

The requirements state a real-time application programming interface (API) for the team's toolchain, i.e. an API capable of working with live data as opposed to stored data. As per system architecture definition, data is received in raw format by the API and then offered to the team locally, not directly over the wireless data link.

There are several possibilities to implement this and the choice is left to the implementation. However, it is worth considering that the unified architecture uses a wireless communication path which, when implemented correctly, can serve as an API for a custom application to plug into. Enabling this would require three main aspects to be realized:

- Connecting to the system must be possible via a widely used connection protocol such as TCP or UDP to ensure that any platform or programming language the team wants to use can implement the connection.
- A clear protocol for connecting and communicating data must be established.
- The data format must be well defined.

If these aspects are realized for the wireless connection within the system itself, a custom application can make use of the same interface as an API.

4.3.2.3. Graphical User Interface

The requirements further state that a simple graphical user interface (GUI) for less well trained personnel to monitor measurement data critical to car operation The possibilities for providing an GUI fall into two major categories, namely:

- using a framework such as Qt [Com] which is stand-alone on a user's machine, but potentially requires different versions for different operating systems and specifically requires the user to execute an application binary locally

- using Hyper Text Transfer Protocol (HTTP) to serve content to the user's browser, which at least with a very limited user group using up-to-date browsers is platform independent and does not require local binaries

While both are viable options, a browser based solution has the clear advantage of not requiring any distribution neither either setup nor update, thus keeping user maintenance to a minimum. Additionally, a solution using HTTP would not require further protocols but could serve content directly if done correctly. It is left to the implementation to make a choice.

4.3.3. Technology and Hardware Availability

For the technology and hardware selection, requirements beyond the functionality are relevant. Price and form factor as well as available interfaces have to be considered when choosing the base system, as well as the software platform provided. Extensibility is a factor, so is reliability. In this section the available technology and hardware is analysed starting with CAN bus technology in section 4.3.3.1, followed by wireless technology in section 4.3.3.2 and base systems in section 4.3.3.4 and lastly, programming languages are analyzed for suitability in section 4.3.3.5.

4.3.3.1. CAN technology

CAN bus interfaces are commonplace in microcontrollers commonly used in the automotive industry, but otherwise rather rare. A CAN bus interface consists of two main components, a CAN controller and a CAN transceiver. The controller is responsible for medium access while the transceiver physically encodes and decodes CAN frames on the bus. In automotive microcontrollers like the Infineon TriCore™ [Inf], these components are integrated in the chip directly. In more common, ARM based controllers like the AM335x Sitara™ Processor [Tex] used in the Beaglebone Black or the Allwinner A20 [Alw] used in the BananaPi and BananaPro platforms, only the CAN controller is integrated on chip and a CAN transceiver has to be added as a breakout board. Other CAN modules for base systems like Arduino or Raspberry Pi use the SPI interfaces of these systems which are exposed in their headers. Finally, there is a plenitude of USB devices for CAN communication available, e.g. USBtin [Tho], an open source USB to CAN interface.

On the software platform side, microcontroller hardware usually comes with libraries instructing the developer on how to use the CAN interface. ARM based platforms running Linux like the Beaglebone Black and the Raspberry Pi profit from the fact that Linux has started to include kernel modules for several CAN devices using different interfaces, e.g. the MCP2515 CAN controller, an SPI device, and a module for serial devices such as the USBtin called SerialCAN. When connected and activated, CAN devices show up as network devices in the device tree and can be used via `socketcan`, a set of open source CAN drivers and networking stack developed by Volkswagen Research. The alternative to using the network socket based `socketcan` [soc], which is included in the mainline kernel, is using a character device based driver like `can4linux` [can].

4.3.3.2. Wireless Technology

There are several data link layer options for transmitting data wirelessly between car and base station. ZigBee, an IEEE 802.15.4[GCB03]-based specification, has been evaluated by Copeto [Cop09], who achieved a data throughput rate of 4.28kB/s. As the requirements state that at least 600kBit/s, equivalent to 75kB/s, has to be transmitted wirelessly, ZigBee can be ruled out. Braune [Bra14] determined in his analysis of Wi-Fi as defined by 802.11g [IEE07] that "throughput is close to constant up to a distance of 400m and then begins to drop off" ([Bra14], p. 21) using hardware similar to the Ubiquiti Bullet™ the team is using, the Ubiquiti PicoStation™ M2 HP [Ubib][2]. The minimum throughput achieved by Braune is around 5MBit/s for 600m range, which is still in excess of the system requirement of this work. In his extensive analysis, Braune furthermore finds that movement up to 75km/h has no substantial impact on throughput nor latency, thereby identifying Wi-Fi as suitable technology for his goal of transmitting a video stream along with live VCU data.

Wi-Fi has the additional advantage of implementing security. The IEEE 802.11i standard defines network access protection, commonly referred to as "Wi-Fi Protected Access" or WPA in short. The standard was introduced in 1999 and updated in 2004 to IEEE 802.11i-2004, also called WPA2. The update introduced the Advanced Encryption Standard (AES) block cipher as the base for encrypted transmission (as opposed to the Rivest Cipher 4 (RC4) stream cipher, which had been used before) thus making WPA2 virtually impenetrable by current standards. The remaining vulnerability lies in using weak passwords for the authentication with the wireless network. WPA and WPA2 rely on passphrases of 8 to 63 printable ASCII characters, which are then hashed to 64 hexadecimal digits, resulting in a password entropy of 256 Bits. To crack an 8 character password using only alphanumeric characters using a brute-force attack, 26 (lowercase letters) + 26 (uppercase letters) + 10 (digits) = 62 possible values exist, thus the number of attempts (if the last one is successful) calculates to

$$A(8) = v^8 = 62^8 = 218340105584896 \approx 2.2 * 10^{14} \tag{4.1}$$

where v is the number of possible values. In order for an attacker to crack such a password within a week, approximately $1.8 * 10^8$ attempts per second would have to be performed (assuming the attacker hits the password after half of the attempts he makes on the correct password length). This is not trivially possible with current hardware. Thus, a strong password of even minimal length required by WPA/WPA2 should suffice to satisfy the team's requirement. If an even longer password of otherwise similar characteristics is used, this results in even stronger security. The team has been advised accordingly.

Another option exists in the form of cellular networks which provide sufficient throughput in their 3G (UMTS) and 4G (LTE) standards. However, the team has expressed concern mentioning severe lack of cellular network coverage at several of the testing sites used as well as potentially problematic cost when operating outside of Germany. Thus, the option has not been examined further.

[2] Ubiquiti define their products according to, among other things, their range. In their portfolio, the PicoStation™ covers the "medium range", defined as "up to 500m", while the Bullet™ covers the "long range", defined as "over 50km", although this last number can presumably be only achieved using a directed antenna on top of the Bullet™, which the team does not to. The antenna used by the team is a vertically polarised omnidirectional antenna similar to that of the PicoStation™.

4.3.3.3. Network Communication Protocols

For network communication above layers 1 and 2 (physical and data link) in the OSI model, a protocol implementing the network layer (layer 3) isn't strictly necessary but provides flexibility both in terms of hard- and software. Internet Protocol version 4 (IPv4) is still widely in use and easily accessible and allows for managed routing and traffic control.

On the transport layer (layer 4), the two most commonly used protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is reliable in that it is connection oriented via three-way-handshake and guarantees ordered and error-checked delivery of payload at the cost of bigger overhead and thus greater latency. UDP is datagram based and, compared with TCP, only offers data integrity checking via checksum. It is, however, far superior to TCP in terms of latency due to reduced overhead and complete absence of handshaking. It is left to the implementation to choose which protocol combination to use for which functionality.

4.3.3.4. Base Systems

As mentioned in section 4.3.3.1, automotive microcontrollers like the Infineon TriCore™ have CAN interfaces built-in. However, these controllers are not usually available as deployment ready boards as they are meant to be used by original equipment manufacturers (OEMs) in proprietary board layouts. They are available as development boards, which in the case of the Infineon TriCore are outside the required price range (see the hitex online shop for TriCore™ starter kits [Hit]) and in general tend to be fitted with unnecessary sensors, potentiometers and on-board displays because they are intended for engineers to test the controller's functions. Additionally, most of these controllers have development environments which are under proprietary license (most notably the TriCore™).

The Arduino platform is worth mentioning in this context. All the required interfaces and essentials are available as so called *shields*. However, using shields, the required form factor is difficult to comply with and extensibility is virtually non-existent, e.g. only one CAN will ever be possible with the platform.

When looking at base systems with ARM® processors running Linux, there is a wide range to choose from:

- Raspberry Pi series
- Banana Pi
- Banana Pro
- Beaglebone series
- Cubieboard series

Many more can be found. Essentially, the Linux-powered Internet of Things (IoT) is on the rise, producing small and cheap one-chip computers. Looking at the form factors, the Raspberry Pi series boards and the Banana boards are close to the requirement. The Beaglebone Black is even smaller than these, but still doesn't fulfill the requirement and lacks the plenitude of ports the Raspberry Pis and Banana series boards offer. The form factor of the Cubieboards is far from fulfilling the requirements and they are thus excluded from selection.

4.3.3.5. Programming Language

In the microcontroller realm, C is the most commonly used language with all above mentioned platforms offering C libraries and bindings. The Linux platform also uses C as the preferred language with not only the system itself, but also all drivers and modules written in C. As such, the `socketcan` library is available by default in the `/usr/include` directory of Linux systems.

As for higher level languages, Python offers native `socketcan` support since version 3.3 and has a library for using `socketcan` which serves as a wrapper of all `socketcan` functionality. No other languages have CAN support built-in in any way, thus requiring the use of native C through the respective foreign function interface (where available).

4.4. Requirements Feedback Loop

At this point, with no implementation provided yet, it is not possible to state which requirements can already be covered. However, the specifications provided do affect feasibility of fulfilling certain requirements.

The architecture outlined in section 4.3.2 includes a description of all functionality required by table 4.1, yet without performance measures. Section 4.3.2.1 on the data structure and 4.3.2.2 on the application programming interface platform describe in more detail what basis the data format and interface and thus, the API, must be built on. Section 4.3.3 on technology and hardware availability describes available hardware, making clear that fulfilling the performance measures as well as legal and policy requirements stated by table 4.2 and physical requirements stated by table 4.3 can be fulfilled with the right technology choices. All requirements affected by chapter 4 have been put in italic in the respective tables.

The system MUST read data from a CAN bus at a throughput rate of 600kBit/s.
The system MUST be able to persistently store 1.62GB of raw data in real time.
The system SHOULD allow extraction of stored data at a throughput rate of 6MBit/s.
The system MUST transmit data wirelessly over a distance of 250m.
The system MUST provide the team with a flexible solution to prioritize data.
The system SHOULD provide the team with a flexible API for its preexisting toolchain.
The system SHOULD provide an external CAN interface to the car's CAN bus.
The system MUST be able to communicate with the car bidirectionally.
The system SHOULD enable unskilled personnel to monitor mission critical data.
The system SHOULD work plug and play after an initial setup.
The system MUST withstand attacks on the wireless data link for one week.

Table 4.1.: Well-formed functional performance requirements

The system MUST NOT exceed the transmission power limit set by European law.
The system's software MUST respect code licensing if foreign code is used.
The system MUST NOT interfere with throttle/accelerator pedal or brake pedal signals to comply with Formula Student/SAE rules.
The system's documentation MUST enable further development after handover.
The system SHOULD cost between €'250.- and €'350.-.

Table 4.2.: Well-formed organizational, legal and technical policy requirements

The VM SHOULD NOT be bigger than 48x32x105mm.
The VM SHOULD NOT weigh more than 150g.
The VM SHOULD operate in a temperature range between 3°C and 65°C.
The VM SHOULD be resistant to fluids.
The VM SHOULD be electromagnetically compatible with all cars the team uses it in.
The VM MUST endure vibrations common in the team's cars.
The VM MUST endure repeatedly being mounted in and dismounted from cars.
The TM MUST be small and light enough to not cause additional logistical effort.
The TM SHOULD be weather resistant.
Data link quality SHOULD NOT suffer from a great number of interfering devices.

Table 4.3.: Well-formed physical requirements

5. Implementation and Testing

In this chapter, the implementation of the system is documented, carried out and then tested. This includes the selection and setup of the hardware and its packaging in section 5.1 and the selection of platforms and data formats as well as the software implementation in section 5.2. Tests are carried out continuously during implementation with a virtual test bench and later validated by real-world testing with the PWe6.15[1] and documented in section 5.3. Finally, requirement satisfaction will be verified in section 5.4 to ensure the team's requirements have been fulfilled.

5.1. Hardware Selection and Setup

This section describes the hardware selection and setup and details the relation the choices bear to the requirements for the system. Section 5.1.1 discusses the selection of the hardware while section 5.1.2 describes the setup. Section 5.1.3 briefly describes the packaging used for the system prototype.

5.1.1. Hardware Selection

In a feedback loop with the team, it has been determined that a platform running Linux is the favorable choice for a list of reasons, most notably:

- Hardware limitation is mitigated by the broad range of available devices.
- Cost is low for the same reason.
- Availability of online material (software, documentation, general support) is very high for the same reason. This not only speeds up development, but also eases development after handing the system over to the team.
- Modularity is kept high.
- No licensing cost is involved nor is it to be expected in the future.
- No additional development environment is required.
- The team expects a rising number of Linux devices to be used on the cars in the future with Formula Student Driverless[2] on the horizon.

To keep development effort low for the prototype, the Raspberry Pi 2 together with the USBtin have been chosen. The Raspberry Pi 2 is a single-board computer with a 900MHz

[1] Tests have been performed in the laboratory as the car was not available for driving.
[2] Formula Student Germany is introducing a competition for driverless vehicles for 2016, thus supporting the trend towards autonomous vehicles in the automotive industry.

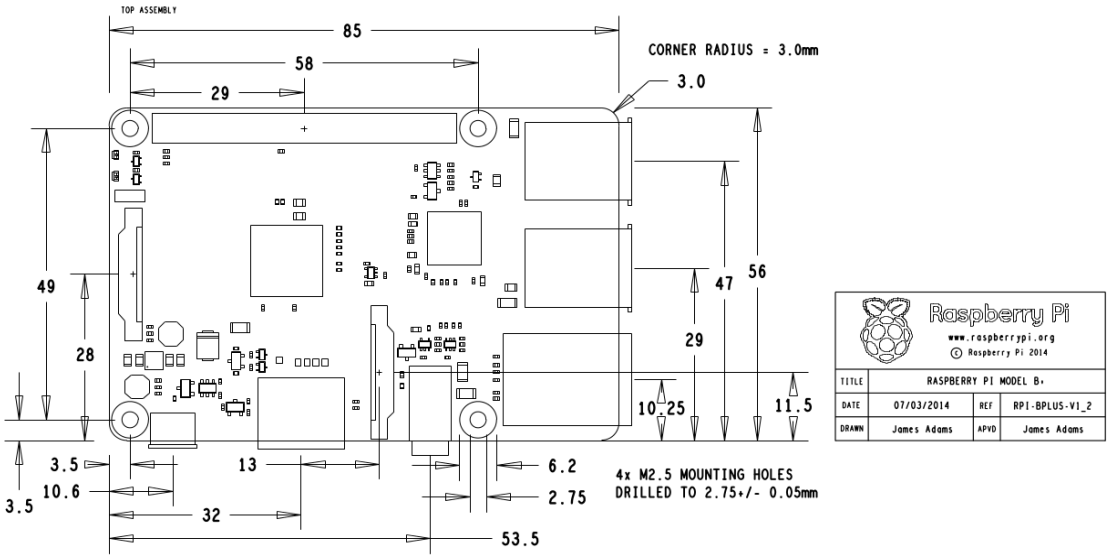


Figure 5.1.: Drawing of the Raspberry Pi 2 with dimensions included

quad-core ARM[3] Cortex-A7 processor which is powered over USB. It has 4 USB ports and an Ethernet port as well as an High-Definition Multimedia Interface (HDMI) output, thus making it a suitable development platform. Figure 5.1 shows the planar dimensions of the Raspberry Pi Model B+. Since this version of the Raspberry Pis, these dimensions have remained the same, as has the arrangement of the interfaces. At 85x56mm its form factor is slightly different from the 2D reference system. Together with the case, which has been modified to also contain the USBtin, the overall system size in the car is at 70x25x96mm plus an outer cable (system shown in figure 5.2). The Raspberry Pi 2 runs its operating system from a MicroSD card which was chosen to be 16GB in size for this project. With Raspbian and all required packages and software loaded, the main development system still has 9.2GB of space on the card, thus satisfying the requirement of 1.62GB raw data (assuming a data format overhead of less than 7.58GB per 1.62GB raw data).

The fact that there is no further hardware development necessary (such as creating a breakout board with a CAN transceiver) and the fact that online resources indicate that this combination is hassle free in setup make this choice ideal for the initial system development. As for the wireless technology, Wi-Fi through the Ubiquiti Bullet^{TM} has been chosen for the prototype as it is currently available with no cost since the team is already in possession of the devices. The Bullet is excluded from the size requirements since it had already been installed on the cars before, but not from the bill of materials (BoM) for the system as it is a required part for the complete system.

The combination of Raspberry Pi 2 and USBtin as components is thus - while being slightly larger than the stated size requirements - capable of satisfying the weight parameters stated by the requirements table 3.4 in chapter 3 at a weight of $\approx$ 110g and exceeds the price

[3]device segment.

Component	Cost per Unit	Units Required	Sum for System
Raspberry Pi 2 Model B+	€ 35.04	2	€ 70.08
Rydges Case	€ 6.49	2	€ 12.98
SanDisk 16GB Micro SD	€ 7.99	2	€ 15.98
Edimax USB Wi-Fi Dongle	€ 9.25	1	€ 9.25
USBtin USB-CAN Adapter	€ 37.90	2	€ 75.80
USB A to Micro-B Cable 0.3m	€ 2.99	2	€ 5.98
Copper Wire	€ 0.10/m	0.5m	€ 0.05
Sub D Plugs	€ 0.20	2	€ 0.40
Total w/o Wi-Fi router			€ 190.52
Ubiquiti Bullet™	€ 82.38	2	€ 164.76
Alfa Network Antenna	€ 13.62	1	€ 13.62
F-type N-type adapter	€ 9.16	1	€ 9.16
TP-Link Wi-Fi Antenna	€ 4.99	1	€ 4.99
Ethernet Cable	€ 1.20	2	€ 2.40
Total			€ 385.45

Table 5.1.: Overall system bill of materials (BoM)

requirements stated in table 3.3 only slightly (also chapter 3) as displayed in table 5.1[4]. The Ubiquiti Bullet™ is pre-chosen by the team because it already owns this equipment but could be replaced with a cheaper equivalent (e.g. the MikroTik Groove 52HPn, €56.84, reducing overall cost by €53.08) Beside the Raspberry Pis, the USBtins and the Bullets, there are other components in the list necessary for a fully functioning system. The Alfa Wi-Fi antenna directly connects to the Ubiquiti Bullet™ located at the trackside while the TP-Link Wi-Fi antenna is connected to the Bullet in the car through the F-type N-type adapter. The Raspberry needs to be capable of Wi-Fi to connect to the trackside Bullet, for which the Edimax Wi-Fi Dongle is used. Ethernet cables are necessary to connect the Raspberry Pi 2 and the Bullet in the car and to power the Bullet at the trackside. USB A to Micro-B cables are required to connect the USBtins to the Raspberry Pis. Sub D plugs are required to connect the USBtins to the CAN bus and copper wire is used to connect the Sub D plugs to the USBtins.

5.1.2. Hardware Setup

The parts the team did not have in-house (Raspberry Pi 2, USBtin) were ordered online and upon arrival checked for functionality using CAN testing tools the team provided. The operating system tailored to the Raspberry Pi 2 is a Debian Linux derivative called Raspbian and does not contain the kernel modules for driving CAN interface hardware, although these modules are available in the mainline kernel and distributed by many Linux distributions, most notably Debian itself. For reasons to do with the way Raspbian is set up which could not be clearly figured out in detail, native kernel module compilation cannot trivially be done using a kernel from kernel.org [LKO]. There is, however, a github repository called `rpi-source` [un] which provides a script for downloading kernel sources preconfigured for Raspbian, thus making it easy to compile and use the kernel modules in question (`can`,

[4]Prices are all taken from amazon.de, found on April 10, 2016.

Figure 5.2.: Vehicle module

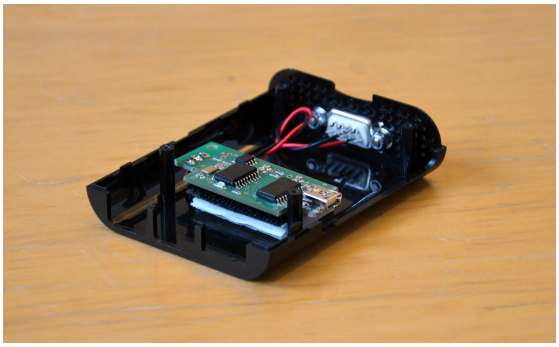


Figure 5.3.: USBtin mounted in vehicle module

`can-raw`, `vcan` and `slcan`). Together with the CAN utilities bundled with socketcan and available through Raspbian's package management system (`apt-get install can-utils`), the functionality test went through with no further complications.

The system within its case and integrated USBtin is shown in figure 5.2. In order to fit the USBtin in the Raspberry Pi's case, the audio connector was desoldered from the Raspberry Pi 2 and the case was modified to allow for the USBtin's plug to be accessible and for the Sub-D connector to be attached. The USBtin was then attached to the case using zip lock tape as shown in figure 5.3.

The setup of the two Ubiquiti Bullet^{TM} has been used for some time by the team. The Bullet in the car acts as a bridge between the device connected to its Ethernet port and the wireless network it is connected to. The Bullet at the trackside acts as a Wi-Fi router to which the other devices, including the Raspberry Pi 2 through its Wi-Fi dongle, connect to. Both Bullets are powered over Ethernet, for which the team has developed a self-built solution on both ends.

Figure 5.4 shows the setup used on board the car, assembled on the workshop table. Not shown is the power supply for the devices. The team uses a modified Ethernet cable to power the Ubiquiti Bullet^{TM} and will use a connector for the Raspberry Pi's GPIO power in. In testing an external battery powering the Raspberry Pi 2 over USB was used. The setup for the part of the system used at trackside is almost identical, the only difference being that the Raspberry Pi 2 connects to the Wi-Fi network created by the Ubiquiti Bullet^{TM} via Wi-Fi instead of Ethernet cable. The Raspberry Pi's Ethernet connector can thus be used to connect to a laptop, although this is also equally possible via Wi-Fi. The team has been advised that using an Ethernet cable for this connection might be favorable in order to reduce the load on the wireless network.

The Raspberry Pi 2 is powered over USB in the test setup. When possible a power plug is used, otherwise an external battery powers the board. For in-car usage the Raspberry Pi 2 is powered through its general-purpose input output (GPIO) pins, for which the team has created an adapter to the car's power supply.

5.1.3. Packaging

The team is planning to manufacture a carbon fiber reinforced plastic (CFRP) case for the system. This case will satisfy the requirements concerning fluids, temperature, vibration and electromagnetic compatibility. For the prototype in this work, the Rydges case for Raspberry

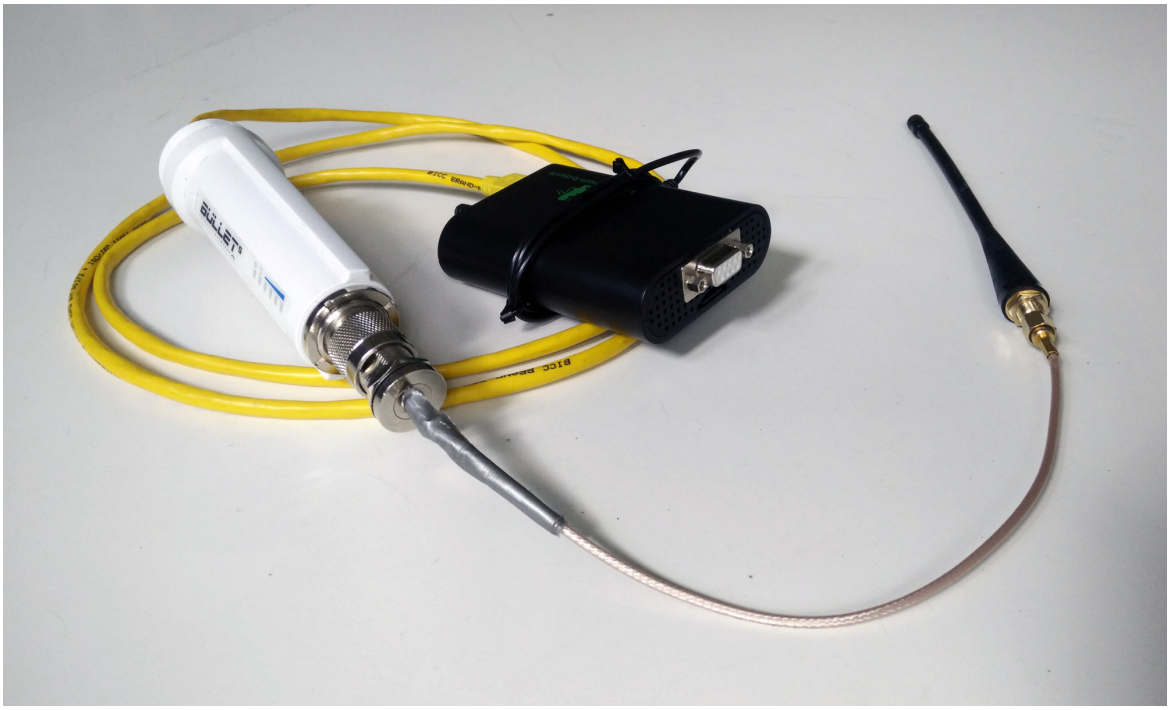


Figure 5.4.: Setup used in car (not shown: power supply)

Pi 2 has been modified to house also the USBtin (see figure 5.2). Further development is left to the team.

5.2. Software Implementation

This section discusses the software implementation in detail. This includes the data format, outlined in section 5.2.1, the selection of programming language, libraries and platforms and of course the implementation, all described in section 5.2.2.

5.2.1. Data Format

The structure of the data has been provided in section 4.3.2.1. In order to work with the data, a machine readable data format able of implementing the provided structure is necessary. There are several ubiquitous file-based data formats which are machine readable and writable:

- Comma Separated Value (CSV)
- Tabulator Separated Value (TSV)
- Extensible Markup Language (XML)
- JavaScript Object Notation (JSON)
- Hierarchical Data Format (HDF) (more specifically HDF4 and HDF5)

Additionally, relational database management systems (RDBSM) are a viable alternative when dealing with persistently stored data. These systems mostly work with the structured query language (SQL) and are thus named after it, e.g. MySQL and PostgreSQL. However, the team has mentioned a preference for file-based storage options.

Among the file-based formats, there are differences in terms of serialization throughput and blob size. CSV (and similarly TSV) is generally considered the fastest and smallest, yet least flexible of formats. Although there are a number of "dialects" in use, it is defined in IETF RFC4180 [Sha05]. XML and JSON are widespread in the internet community and used for their versatility and interoperability. HDF5 is a data format widely used in the scientific community; in particular, the team's toolchain product MATLAB® uses it as its native format.

For this work Comma Separated Value (CSV) was chosen. The team suggested that working with CSV files was seen as a benefit as some of the ECUs also provide CSV formatted data. Furthermore, CSV is character based, which makes it possible to send messages over the wireless link directly, reducing serialization steps to reading from and writing to CAN.

However, a further data format is to be considered. DBC, a file format for storing information that describes a CAN network, is a proprietary format developed by Vector Informatik GmbH. This means that it lists all the ECUs, IDs and signals contained within these IDs together with information such as the location of signals within frames, or how often a frame is sent (ID cycle time). Being a proprietary format, DBC is not well defined by a standard, which is why a detailed description cannot be given. However, Vector Informatik GmbH offer a software library which allows extraction of data from DBC files [Vecb], thus making it possible to work with them without knowing the precise structure. The data provided by the libraries includes (but is not limited to):

- List of board units (ECUs) by name
- List of frames by ID and name
- For every frame ID: list of signals
- For every signal: name, unit, value range and cycle time

As DBC files are an integral part of the team's toolchain, they have been chosen to be used for this work as well for several functions. The requirements specify the need for the user to flexibly prioritize data in the event that the signal strength of the wireless link cannot be guaranteed. This is realized by a DBC file containing the priority frame IDs which the user uploads to the system. Another requirement states that the system must not interfere with the brake and accelerator signals. To implement this, the user uploads a DBC file containing the respective frame IDs, which are then filtered from being sent to the CAN bus. Lastly, DBC files are used for configuring the GUI@Ḃy using DBC, the user can set the configuration of everything concerning CAN frames and signals using the DBC editor provided by Vector Informatik GmbH as a free tool.

5.2.2. Implementation

Python is chosen as programming language as it offers all the necessary utility including libraries for CAN, HTTP and DBC, it allows for fast development and it is sufficiently

ubiquitous to be used by a team member in the future, thus facilitating further development after system handover.

The basic structure of the software follows the structure outlined in section 4.3.2, displayed in figure 4.17, with only few adaptations. The format chosen for raw data is CSV.

5.2.2.1. Network Communication

In the communication between the part of the system situated in the car, referred to as *vehicle module* in this section, and the part of the system situated at the trackside, referred to as *trackside module* in this section, overhead needs to be kept low. Therefore, UDP has been chosen as the transport protocol. Higher level protocols will be avoided on this communication path for the same reason. The same is true for the communication path between the trackside module and the team's toolchain, since the team wants the access to the bus as direct as possible. As the team has signaled that CSV formatted data is satisfactory for the API, no reformatting is required. From a functional point of view the toolchain API is thus the same as the wireless link interface used by the vehicle module and the trackside module. It follows that an additional implementation of the user API is not necessary. The port used for communication is fully configurable through a configuration file which by default sets it to `5252`.

The protocol for these communication paths is kept as simple as possible, knowing only the following verbs:

- `REG`: Modules (vehicle, trackside or user) coming alive broadcast this verb into the networks they are connected to in order to register themselves with any other modules alive on the network or send it to the module they want to register with directly.
- `ACK`: Modules receiving the `REG` verb answer with the `ACK` verb and register the IP address of the sender in their list of known modules. Modules receiving the `ACK` verb register the IP address of the sender in their list of known modules.

Thus, all modules participating store a list of known modules where the IP, the last message received from the module and the time of receiving the last message from the module are stored. Modules that become inactive are purged from the lists of the other modules after an inactivity timeout period of 5 seconds, thus requiring modules which do not routinely send CAN messages to send the `ACK` verb at least every 5 seconds as a keep-alive signal. Modules send data received on their CAN to all modules in their known modules list.

Messages received from either CAN or other modules are stored and broadcast to all listeners if they pass the filter. Thus, a duplication of stored data happens intentionally as it allows for immediate data analysis of data received but guarantees that no data that enters the system is lost. The stored files are synchronized upon user request as described in section 5.2.2.3.

5.2.2.2. Data Format

The format of the transmitted CSV strings as per grammar in section 4.3.2.1 and RFC4180 [Sha05]:

$$\langle MSG \rangle ::= \langle DID \rangle,\langle BID \rangle,\langle TS \rangle,\langle CONTENT \rangle\langle CRLF \rangle$$
$$\langle CONTENT \rangle ::= \langle ERR \rangle \mid \langle RTR \rangle,\langle IDE \rangle,\langle MID \rangle \mid \langle IDE \rangle,\langle MID \rangle,\langle DLC \rangle,\langle DF \rangle$$

$\langle DLC \rangle$ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
$\langle DF \rangle$ ::= $\langle BYTE \rangle$ | $\langle BYTE \rangle$ $\langle DF \rangle$
$\langle BYTE \rangle$ ::= 0 | 1 | ... | FE | FF

with

- String: Device Identifier (DID)
- String: Bus Identifier (BID)
- Timestamp: Timestamp (TS)
- Boolean: Error Frame (ERR)
- Boolean: Remote Transmission Request (RTR)
- Boolean: Extended Arbitration Identifier (IDE)
- Bit sequence (11 or 29 Bits): Message Identifier (MID)
- Integer (between 0 and 8): Data Length Code (DLC)
- String: Carriage Return Line Feed (CRLF)

As communication is string based, all fields have to be represented by strings. Timestamps are given in seconds with six decimals. Booleans are represented by 0 (False) and 1 (True). ID is given as an integer value between 0 and 2047 or 536870911 for extended ID. The payload is given as a hexadecimal number. The device identifier is set through a configuration file, the bus identifier is the name of the CAN interface on the device. An example message using standard CAN ID (as opposed to Extended ID) thus looks like this:

```
PW6.15,slcan0,132.334924,0,0,0,6296,8,2a303f1e13bc2a81\n
```

This is also the storage format, further simplifying proceedings. The message strings are written as to a file in the same format. File name and subdirectories are fully configurable through the configuration file relative to the root `/var/www/loggings`, allowing for dynamic generation of storage file name using time.

5.2.2.3. Data Extraction and User Interface

For the data extraction stable and reliable transfer has a higher priority than throughput. Since the car is stationary during the extraction, the requirement of 6MBit/s of data throughput can be met by any modern data link, most notably also by the wireless interface in use for the other functionality of the system. This remains true if TCP overhead is introduced, with TCP guaranteeing safe transmission as opposed to UDP, which does not. Therefore, TCP based communication via an HTTP based interface has been chosen. HTTP offers a number of request methods, defined in RFC 2616 [FGM+99], among which are:

- `GET`: request a resource
- `POST`: offer data attached to the request to the server

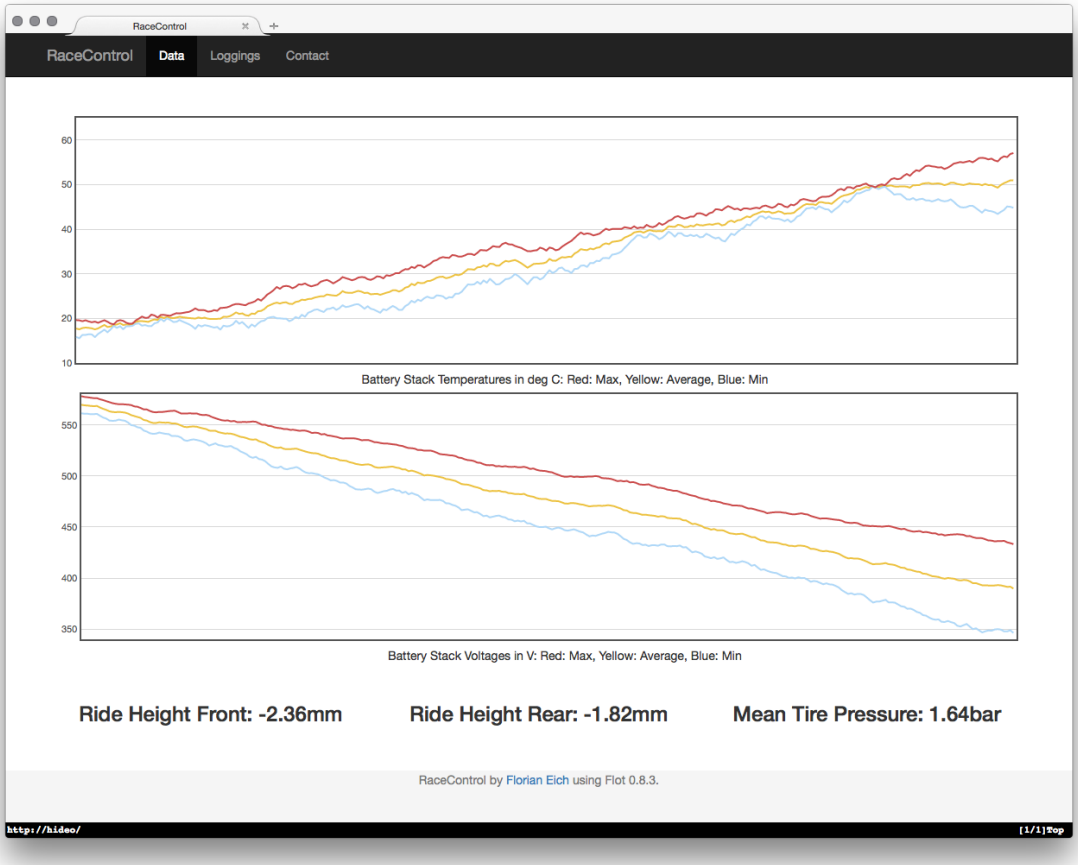


Figure 5.5.: Screenshot of the Graphical User Interface

- `PUT`: request that data attached to request be stored under resource
- `DELETE`: delete the specified resource

The resource is specified as a file with the server's public root as reference, or simply as a string to be interpreted by the server, also commonly referred to as `route`. HTTP was created to enable the world wide web and is thus used for the transferral of Hyper Text Markup Language (HTML) and web content in general, thus naturally lending itself to implement browser-based graphical user interfaces (GUIs). With version 5, HTML saw the introduction of server-sent events (SSE), a method to stream data to a web browser from the server, something that was not possible previously and instead implemented via user-side applications polling a HTTP resource.

Thus, the graphical user interface is implemented as a web service using HTTP. Upon browser request of the module's IP address, which is a `GET` request on port 80, the module serves a combination of HTML and JavaScript with the embedded server-sent event container receiving the continuous data input from the server. This is illustrated by figure 5.5, which is a screenshot of the GUI. The link in the navigation which reads "Loggings" directs the user to a directory index page where all the loggings are stored. This is statically served by

```
slcan_attach -f -s8 -o /dev/ttyACM0
slcand ttyACM0 slcan0
ifconfig slcan0 up
```

Listing 5.1: Attaching USBtin serial CAN device

```
if [ $(tty) = "/dev/tty1" ]; then
  while true; do
    /usr/local/bin/racecontrol;
    echo "Again [$?]...";
  done
fi
```

Listing 5.2: Starting and restarting RaceControl on boot on tty1

the web server, which has to be configured accordingly. From here, the user can download the loggings as CSV files directly.

The `route`s for the web service only use the `GET` verb. There are two `route`s serving HTTP:

- `/`
  The index or root `route` serves the data presentation directly.
- `/loggings`
  This `route` takes the user to a web index of the loggings directory. From here, loggings can be downloaded directly.

The web application is built using the web framework `Flask` [Ron] and served through the web server `nginx` [Inc] on port 80. The web index of the storage directory is server directly through the web server, also on port 80.

5.2.3. System Setup

As mentioned in section 5.1.2 the base system of both modules is Raspbian, which is "a free operating system based on Debian optimized for the Raspberry Pi hardware" [Foub] and thus, a distribution of the GNU/Linux operating system. The entire system runs from a MicroSD card. In the `/etc/modules-load.d/`, the init system of Raspbian looks for files ending in `.conf` containing modules to be loaded. This is where the file `can.conf`, which contains the module names `can`, `can-raw`, `slcan`, `vcan`, resides. During boot, these modules are loaded in the kernel.

In order for the USBtin to work, it must already be connected to the system during boot and ideally also to a CAN bus. Since these components are fixed together and installed on the car, this is the case. Then, the commands to attach the serial CAN interface are run after system boot when the boot finished up by running `/etc/rc.local`, which has been appended with the code in listing 5.1.

The Raspberry Pi 2 is configured so that it automatically logs into the user account `pi`. The user configuration file contains another bit of code shown in listing 5.2, which starts the

system's software and upon crash restarts it in an infinite loop. Thus, the system works plug-and-play after an initial setup, fulfilling one of the requirements in table 3.2. Furthermore, a web server is running on the Raspberry Pi 2. It proxies port 5000, on which the system's software is serving HTTP, to port 80.

5.2.4. Summary

In summary, the software implementation follows the model as closely as possible while respecting the requirements as well as making use of the given tools. A class diagram is shown in figure 5.6. Only the classes implemented specifically for this system are depicted, classes from libraries are omitted for clarity.

Furthermore, appendix A has the documentation of the entire system. Unit tests are provided with the software and are written with the help of the `py.test` framework.

5.3. Testing

Basic function tests as well as unit tests of the software have been performed throughout development. This section details the overall system functionality tests and provides measurement data. The tests target the requirements directly as per section 3.4.4.

5.3.1. In-car Test

As no car was available for testing in real-live conditions, workshop tests were performed qualitatively. For this, the system was hooked up to one of the CAN buses of the PWe6.15 and then started. The results obtained were satisfactory for the team but could not be quantitatively measured since no reference data for the used setup was available. The analyst made the observation that for signals with a cycle time below 10ms, every few seconds error frames were produced for the corresponding identifiers in no particular pattern. According to the team, this is expected behavior for signals in this cycle time range. Further tests will be conducted by the team as soon as cars become ready for real-live testing. Performance tests with the system's CAN bus interface are detailed in the following section.

5.3.2. Data Throughput

There are several requirements concerning data throughput rates, which are discussed in this section. First of all, there is a requirement concerning reading from the CAN bus, which is discussed in paragraph 5.3.2. Writing the data read from the CAN bus to storage is also placed under a requirement as outlined and tested in paragraph 5.3.2. Finally, the requirement for data throughput during extraction of stored data is tested in paragraph 5.3.2.

Reading from CAN The requirement for the data throughput when reading from the CAN bus is 600kBit/s. This was tested in two ways:

- with dummy data using the virtual CAN interface `vcan` which is bundled as a module with the Linux Kernel and can be used with the tools provided by the `can-utils` [ftlcp] package.

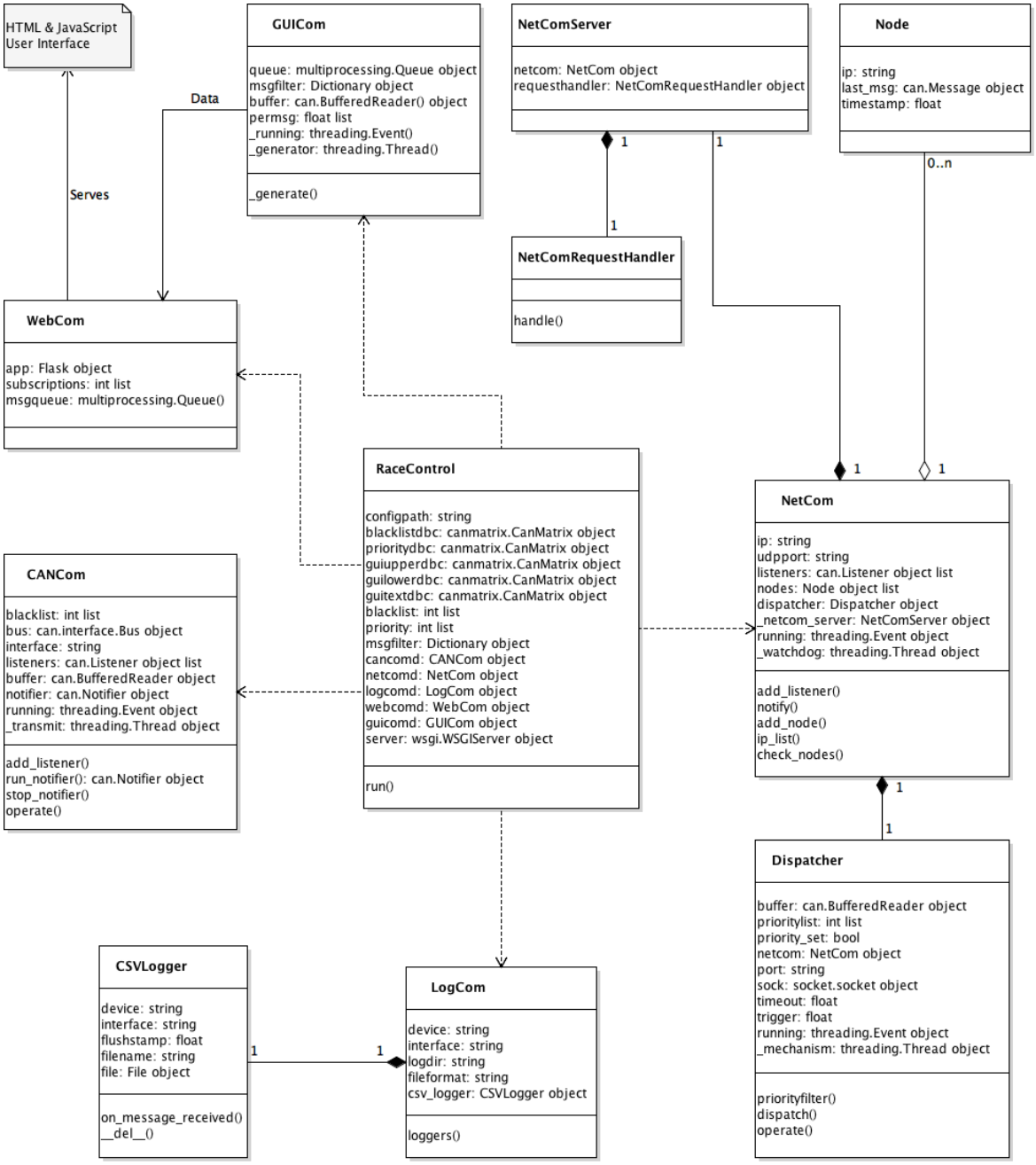


Figure 5.6.: Class diagram of the software system

- with real data provided on the serial CAN interface USBtin using a USB to CAN adapter by PEAK-System Technik GmbH.

The setup for the tests is the same in both cases with only the CAN interface differing. However, an unknown factor is that in the case of the virtual CAN bus, the test system also has to write to the CAN at the same time, thus possibly affecting throughput rate.

To allow for a realistic test, the CAN communication module from the system's implementation is used. To simulate a load on the CAN bus of 600kBit/s, the correct number of CAN frames has to be sent on the other side. For frames using standard ID, this means sending

$$\frac{600000}{111 \cdot 100} \quad \approx \quad 54.05 \quad \approx \quad 55 \tag{5.1}$$

every 10ms, thus equivalent to 5500 and 4600 frames sent per second. The extra three Bits in either CAN IS (111 instead of 108 and 132 instead of 129 Bit) are the inter frame spacing Bits specified to be a minimum of three Bits long. Both numbers have been rounded up to the next integer value. The frames used were a fixed set with randomly selected IDs between 100 and 999 and 8 Bytes of random data.

The test script the instantiates a `CANCom` object which spawns a total of four processes reading from CAN and serializes the incoming messages into CSV strings as per section 5.2.2.2, which it then puts into a queue provided by the test script. The test script gets the messages from the queue but does not process them any further.

To get the most likely value for a big number of iterations, the expectation value is commonly used. It is calculated using the formula

$$\langle x \rangle \quad = \quad \sum_{i=1}^{n} x_i \cdot p_i \quad = \quad \frac{1}{1000} \sum_{i=1}^{n} x_i \tag{5.2}$$

where x_i are the measured values and p_i are their respective probabilities. Since the probability is assumed to be the same for every iteration and the sum takes care of measurements with the same outcome, this becomes $\frac{1}{1000}$ for every sum component.

Figure 5.7 shows the result calculated from 100 iterations of reading from the CAN bus with every iteration lasting for a duration of 10 seconds. Evidently, the rate of reading from the virtual CAN is not very stable jumping within a range equivalent to the order of magnitude of the values, the expectation value of the throughput rate is also much lower than 600kBit/s at $\approx$ 215kBit/s. However, when increasing the bus load, reading from the virtual CAN interface increased in a linear fashion, which led to the conclusion that Linux's `vcan` interface is not suited for throughput performance testing.

Unfortunately, the serial CAN bus interface, the USBtin, did not achieve the desirable throughput either with an only slightly higher expectation value of $\approx$ 223kBit/s. To evaluate the root problem, the test routing was rewritten in C in order to eliminate the Python interpreter as the source of this reading speed limitation. This yielded very similar results, with an expectation value for the read speed of $\approx$ 200kBit/s. The tests were then taken to a different testing environment, a system providing an Intel® Core™ i5-3320M, which had no significant impact on the test results either. Tests performed with a different CAN to USB device, the PCAN-USB from PEAK-System Technik GbmH [Gmbb], concluded that the bottleneck must be the USBtin CAN to USB adapter.

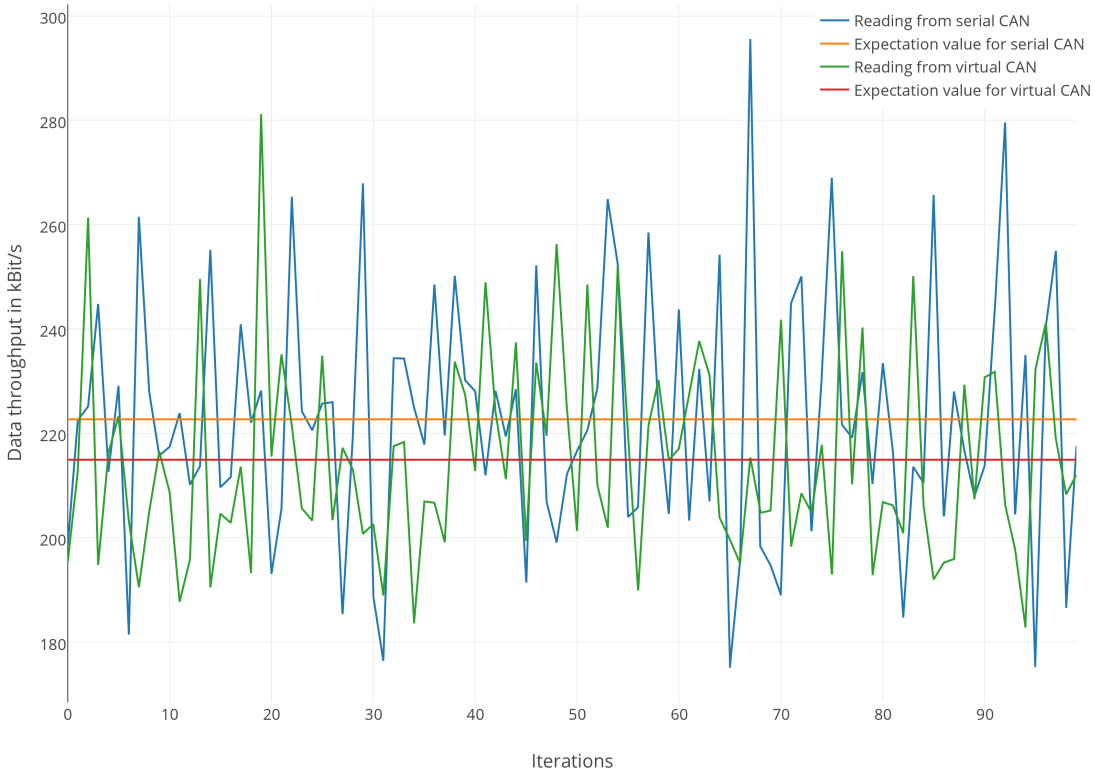


Figure 5.7.: Read Throughput Plot

Writing to Storage The requirement for the data throughput when storing data is real-time, which means that the throughput rate for reading from the CAN bus has to be matched. This means that writing CAN frames to a file in string representation as per definition in section 5.2.1 has to be possible with at least 600kBit/s.

A CAN frame in the string representation from section 5.2.1 is at most 55 characters long for standard CAN ID (11 Bit) and at most 60 characters long for extended CAN ID (29 Bits). To test the write performance for strings of these lengths, 1000 iterations were made writing 100000 strings to a file each. This was done through Python 3.4, which is also the language the system is written in. The throughput rate can then be calculated using the formula

$$R_{\text{throughput}} = \frac{8 \cdot l_{\text{string}} \cdot 10^5}{t_{\text{elapsed}}} \frac{\text{Bit}}{\text{s}} = \frac{8 \cdot l_{\text{string}} \cdot 10^2}{t_{\text{elapsed}}} \frac{\text{kBit}}{\text{s}} \tag{5.3}$$

where l_{string} is the respective string length and t_{elapsed} is the elapsed time for one iteration. To get the most likely value, equation 5.2 is again used to determine the expectation value of the procedure. Furthermore, the time for a given throughput rate $R_{\text{given}} = 600\frac{\text{kBit}}{s}$ can be found by inserting the value in equation 5.3 and solving for x as such:

$$t_{\text{permitted}}\ \text{s} = \frac{8 \cdot l_{\text{string}} \cdot 10^2}{600}\ \text{kBit} \tag{5.4}$$

These formulae work because strings are made of ASCII characters taking up one byte of storage space each and thus, the string length is equal to the bytes written. It should furthermore not be surprising that the time elapsed is inversely proportional to the throughput rate.

The calculated permitted times for every iteration of writing a standard ID CAN frame representation of 55 characters and for an extended ID CAN frame representation of 60 characters are thus

$$t_{\text{perm,standard}} = 73.\bar{3}\ \text{s}, \qquad t_{\text{perm,extended}} = 80.0\ \text{s} \tag{5.5}$$

Figure 5.8 shows the measured values and the expectation value as a red line. The maximum permitted value is not shown in the plot as they are way above the values, thus negatively impacting the scale of the plot. The requirement for writing to file is thus fulfilled, with write speeds an order of magnitude faster than required for even the slowest iterations (in which the throughput rate drops to approximately 8.3MBit/s and 8.1MBit/s for 55-character strings and 60-character string respectively), thus leaving room for error.

Data Extraction To test data extraction a script using `curl` was used to repeatedly download a logging file from the system. The logging file was 17.65MB in size, thus requiring extraction times of

$$t_{\text{extraction}} = \frac{17.65\text{MB} \cdot 8\text{Bit}}{6\frac{\text{MBit}}{\text{s}}} \approx 23.53s \tag{5.6}$$

at the minimum to fulfill the requirement. The measurement was made with the Raspberry Pi connected to a router via Ethernet and the device performing the downloads connected to the router via Wi-Fi, a setup which closely resembles the setup of the on-board device.

Figure 5.9 shows the result of 100 iterations of downloading and measuring. The expectation value was again calculated using formula 5.2 and is also shown in the graph. At

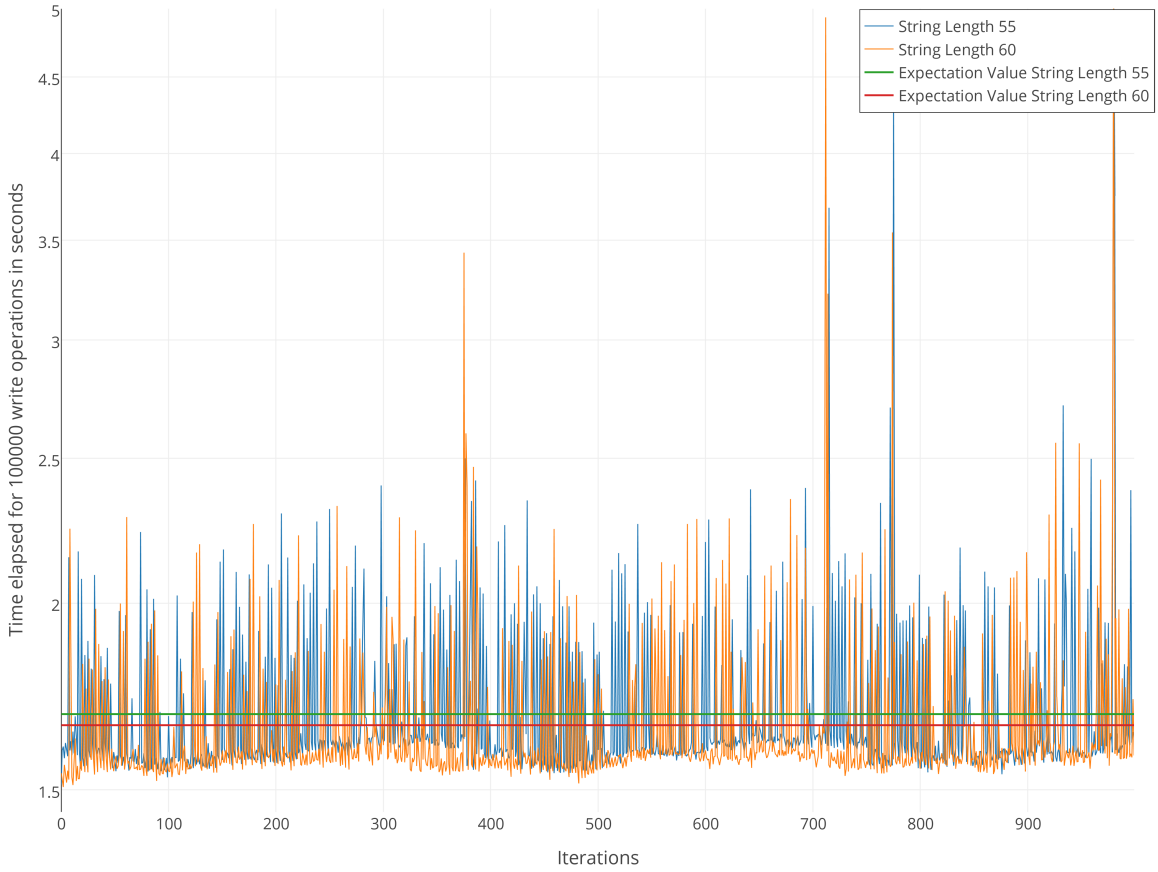


Figure 5.8.: Write Throughput Rate Plot

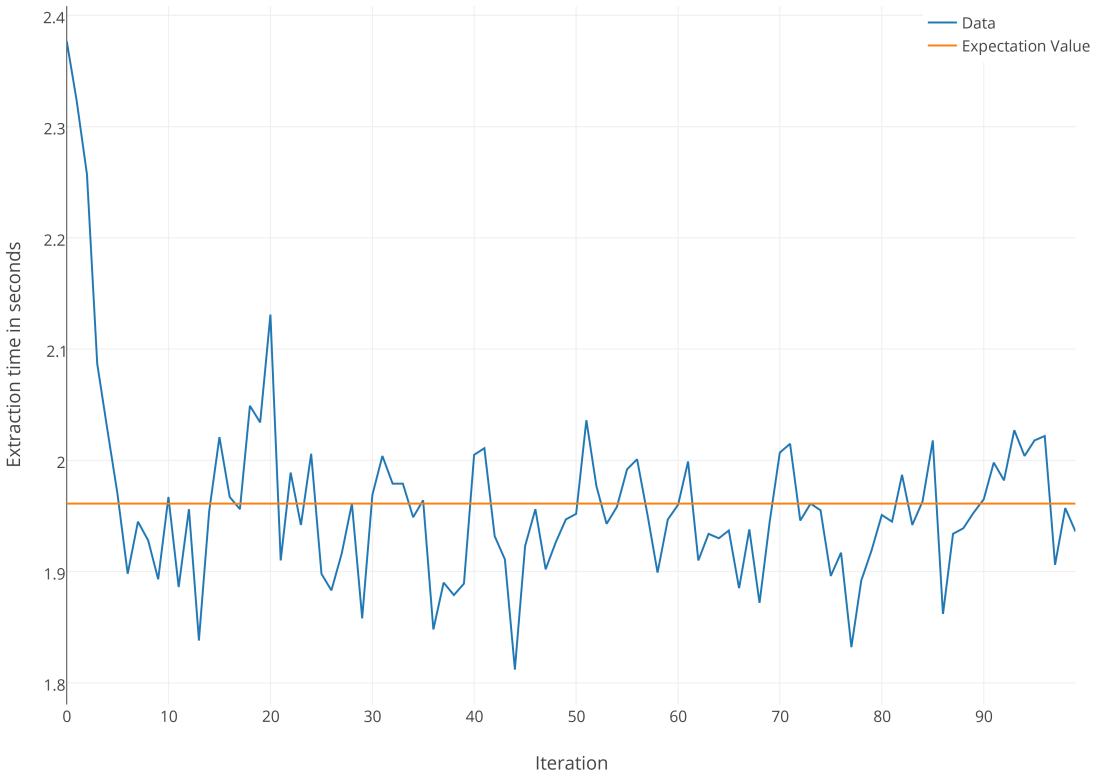


Figure 5.9.: Extraction Times Plot

approximately 1.96s, an expected extraction throughput rate of approximately 72MBit/s was reached, which is comfortably in excess of the required rate of 6MBit/s. The maximum value as per requirement is not shown as it would have negatively impacted the scale of the graph.

5.3.3. Physical Requirements

The requirements concerning the physical environment of the system are mostly related to the vehicle module. Not everything could be tested in live conditions as no cars were available for driving during the creation of this work. Some basic tests have been performed to ensure the system's general capabilities.

Weight Requirement The requirement for the vehicle module states that it should not weigh more than 150g. The vehicle module as shown in figure 5.2 was weighed on a fine scale and came in at 108.6g, thus fulfilling the requirement.

Electromagnetic Compatibility To test the electromagnetic compatibility of the system, the vehicle module was placed inside the electric car from the 2015 season, the PWe6.15, and the car's tractive system (high voltage powertrain) was turned on. For this, the module was placed in the spot the team has designated for it which is close to the power electronics, which according to the team is where the highest strengths of electromagnetic field are generated. The module operated with no further concerns.

This is, of course, no proof of requirement fulfillment. As no other testing options were available, it is left to the team to perform further tests. However, as the system could only be tested in one car, the requirement of being electromagnetically compatible with all cars has to be considered unmet.

5.4. Requirements Feedback Loop

To check which requirements could be met, a list is compiled which contains all features in respect to the requirements they fulfill. This is done in the same order as the tables, starting with the functional performance requirements in section 5.4.1, further examining the organizational, legal and policy requirements in section 5.4.2 and finally analyzing the physical requirements in section 5.4.3.

5.4.1. Functional Requirements Feedback Loop

Functionality and testing to fulfill functional requirements is found throughout this work. It can be separated in the two categories data throughput and storage, discussed in paragraph 5.4.1 and functionality, discussed in paragraph 5.4.1.

Data Throughput and storage The data throughput capabilities have been examined in section 5.3, subsection 5.3.2. From the data provided it is clear that the throughput rate in terms of reading from the CAN bus (600kBit/s) could not be met with current hardware while storing data in real time (600kBit/s) and extracting stored data from the car (6MBit/s) could be met. In an evaluation together with the team, it has been concluded that the achieved data throughput rate is acceptable for telemetry purposes, yet for storing CAN

data an alternative solution will still have to be used for now. Alternatives to the USBtin USB to CAN device, such as using a Beaglebone Black - which uses a processor that has an integrated CAN controller and would as such not need to rely on a USB or SPI adapter - have been briefly analyzed, but not evaluated reliably.

Since a moving car was not available for testing, the distance requirement of transmitting data over 250m could not be verified, but the data provided by the work of Braune [Bra14] makes it clear that this requirement can be met with appropriate hardware and that, since Braune uses similar but less powerful hardware, a suitable performance of the Ubiquiti Bullet^{TM} can confidently be expected.

As for the storage requirement of 1.62GB, the Raspbian operating system used on the Raspberry Pis in a full install (including LXDE desktop environment) and all the components required to run the system take up 4.8GB of storage space, thus leaving 9.1GB on the 16GB MicroSD card. Therefore the storage requirement has also been met.

Functionality The functionality requirements are met through the implementation and the setup of the software.

- The solution provided to the team for flexibly prioritizing data has been implemented using part of their tool chain, CAN database files (DBC). This is as of now a static solution: Only the messages listed in the priority list are transmitted wirelessly. The team is content with this as the status quo but has indicated doing further development on how to dynamically activate the priority setting.
- A flexible API has been provided as per request of the team in form of a UDP service.
- A CAN interface on the trackside module is also naturally provided, as the unified architecture enables both sides to run the same software and the hardware has also been provided.
- Bidirectional communication between the modules has been implemented using the same UDP service.
- The web interface showing data as requested by the team fulfills the requirement of enabling unskilled personnel to monitor mission critical data.
- The setup of the system is capable of plug and play operation with the current setup (the boot time for the Raspberry Pi was initially considered a problem but this concern was dismissed by the team as the cars themselves have an even longer ready-to-operate time).
- Finally, the use of WPA2 secured Wi-Fi ensures that the system withstands attacks on the wireless data link for one week.

5.4.2. Organizational, Legal and Policy Requirements Feedback Loop

The organizational, legal and policy requirements have been addressed throughout this work. The resulting system complies with most of them.

- The wireless transmission does not exceed the transmission power limit set by European Law as specified by the conformity regulations placed on hardware such as the Ubiquiti Bullet™. airOS®, the operating system running on the Bullets, offers a drop down menu in its settings where the user can choose the region he wants to operate the system in, according to which the transmission power is then configured by the Bullet. It is the responsibility of the team not to violate the law by setting the Bullet to a different region in order to allow for higher transmission power.
- Code licensing has been respected: some code used is subject to the GNU Public License version 3 (GPLv3), which the code of the system is also placed under.
- Interference with throttle/accelerator pedals or brake pedals cannot be ensured directly as it is not prematurely known on which CAN frame ID these devices are using. Teams are free to choose their own CAN bus structure so these IDs change with every car. Thus a flexible solution is provided to the team in the form of a filter, which like the priority filter is configured via a CAN database (DBC) file.
- The system's documentation has been kept thoroughly within the code itself in addition to this work. The documentation is found in appendix A.
- While €385.45 is more than stated in the requirements, it is close to the goal and most of the components were fixed since they were provided by the team. If purchasing a new system, alternatives to the Ubiquiti Bullet™ can be chosen, saving a substantial amount. The team is also planning to design and manufacture their own case, eliminating this point almost entirely. Furthermore, money can be saved by buying the USBtins as kit and soldering them in-house. The evaluation of these measures is left to the team.

5.4.3. Physical Requirements Feedback Loop

The physical requirements are discussed in section 5.1. However, it is difficult to determine the fulfillment of some of the requirements which could only have been tested on running cars as no cars were available for driving.

- At 70x25x96mm, the system is slightly larger than stated by the requirement. However, as the team is planning the design and manufacturing of a new case, the volume of the reference system could be undercut with the given hardware. The weight requirement has been met at ≈110g.
- Operation in the given temperature range could not be tested reliably. The data sheets for the components of Raspberry Pi and USBtin, which are the only components of the vehicle module affected by the requirement (the Ubiquiti Bullet™ has already been extensively used on several cars and is known to function), specify temperature ranges, the lowest of which is 0 deg C to 70 deg C (LAN9512-JZX USB hub and Ethernet controller [SMS]). Considering that the vehicle module is shielded from temperatures by an additional housing, the assumption is viable that the temperature range requirement is fulfilled.
- The fluid resistance of the system could not be evaluated. The case used for the prototype is not fully closed, hence fluids could get in. Whether this is fatal for the

system and when is unknown as of yet. The team has indicated that tests for fluid resistance will be carried out upon finalizing the system's new case.

- Electromagnetic compatibility has been qualitatively analyzed by testing the vehicle module inside the electric car from the 2015 season, the PWe6.15. Although this is not a final assurance, it has been assessed that the module works in this car. The requirement concerning the electromagnetic compatibility cannot be considered fulfilled as it states electromagnetic compatibility with *all* cars of the team, which has not been tested.
- As for the fluid resistance, no tests have been carried out for the vibration resistance and no data is available for the components. The team has stated that while they do not expect any immediate problems, the connectors of the system are to be seen as the weak point and it is unclear how long the connections between the board and the connector sockets are going to withstand the vibrations in the car. Therefore the requirement has to be considered unfulfilled.
- Mounting and dismounting the system repeatedly is unproblematic due to the fact that no electronic components are openly exposed and industrial connectors are used. As stated before, the connectors are in danger from vibration in the car, which might weaken their socket connection, but as they are built for plugging and unplugging by human actors this is not seen as a risk factor. A reasonable hardware lifetime (>1 season) is to be expected and the requirement is therefore considered fulfilled.
- Since the trackside module is also a Raspberry Pi, it is considered light and small enough to not cause additional logistical effort.
- Weather resistance, like fluid and vibration resistance, was not tested. However, the same durability as for the team's laptops can be expected from the current setup. Therefore, this requirement is considered to be fulfilled.
- Data link quality could not be tested directly but since the team has not encountered any issues with data link quality using the same hardware at competition, this requirement is considered to be fulfilled.

5.4.4. Tables of Well-Formed Requirements Fulfillment Evaluation

In order to check which requirements have been fulfilled by the proposed system, the requirements from the tables of well-formed requirements shown in section 3.4.4 have to be revisited. This is done below, where the tables are reproduced with check marks noting the requirements that were met.

The system MUST read data from a CAN bus at a throughput rate of 600kBit/s.	✗
The system MUST be able to persistently store 1.62GB of raw data in real time.	✓
The system SHOULD allow extraction of stored data at a throughput rate of 6MBit/s.	✓
The system MUST transmit data wirelessly over a distance of 250m.	✓
The system MUST provide the team with a flexible solution to prioritize data.	✓
The system SHOULD provide the team with a flexible API for its preexisting toolchain.	✓
The system SHOULD provide an external CAN interface to the car's CAN bus.	✓
The system MUST be able to communicate with the car bidirectionally.	✓
The system SHOULD enable unskilled personnel to monitor mission critical data.	✓
The system SHOULD work plug and play after an initial setup.	✓
The system MUST withstand attacks on the wireless data link for one week.	✓

Table 5.2.: Well-formed functional performance requirements

The system MUST NOT exceed the transmission power limit set by European law.	✓
The system's software MUST respect code licensing if foreign code is used.	✓
The system MUST NOT interfere with throttle/accelerator pedal or brake pedal signals to comply with Formula Student/SAE rules.	✓
The system's documentation MUST enable further development after handover.	✓
The system SHOULD cost between €250.- and €350.-.	✗

Table 5.3.: Well-formed organizational, legal and technical policy requirements

The VM SHOULD NOT be bigger than 48x32x105mm.	✗
The VM SHOULD NOT weigh no more than 150g.	✓
The VM SHOULD operate in a temperature range between 3°C and 65°C.	✓
The VM SHOULD be resistant to fluids.	✗
The VM SHOULD be electromagnetically compatible with all cars the team uses it in.	✗
The VM MUST endure vibrations common in the team's cars.	✗
The VM MUST endure repeatedly being mounted in and dismounted from cars.	✓
The TM MUST be small and light enough to not cause additional logistical effort.	✓
The TM SHOULD be weather resistant.	✓
Data link quality SHOULD NOT suffer from a great number of interfering devices.	✓

Table 5.4.: Well-formed physical requirements

6. Conclusion

In this work, a bidirectional CAN bus telemetry system for a Formula Student/SAE has been specified and implemented. In Formula SAE/Student, as in common motorsports, engineers analyzing data have a high interest in real-time data availability, as this enables them to optimize car performance live. The problem to be solved was thus that the Formula Student/SAE team from UAS Munich, municHMotorsport, was not satisfied with the current telemetry solutions, which had shown themselves to be inflexible and in some cases unreliable to the point where one of them was suspected of causing a car to not finish a race. A detailed analysis of the status quo further revealed that the team was also unsatisfied with the current space requirements of the logging solutions and particularly unhappy with the situation in terms of software, which was seen as a limiting factor due to licensing fees as well as unsatisfactory in terms of adaptability and toolchain interaction.

Together with the team the requirements for an alternative solution were analyzed and compiled in a list of well-formed requirements found in chapter 3, section 3.4.4. These requirements provided the basis for the functional analysis and design synthesis, resulting in a system comprised of two modules, one mounted on board of the car and one stationed at the trackside. For these two parts of the system, named *vehicle module* and *trackside module* respectively, a unified architecture for the software viable for both modules has been devised and is shown in section 4.3.2, figure 4.17 along with an analysis of viable hardware to be used in this application.

The proposed system was then implemented according to the specifications as outlined in chapter 5, sections 5.1 and 5.2, and tests were performed, which are detailed in section 5.3 of the same chapter. It became clear that the requirement for data throughput when reading from CAN could not be met with the selected hardware, which is clearly a flaw of the current implementation. This was hard to anticipate, as the manufacturer of the hardware in question did not give any indication of a throughput limitation and the initial tests as well as the tests on the car did not indicate bus overload on the hardware and this flaw was only detecting during dedicated performance testing. In hindsight, more testing should have been performed before the selection of the hardware. However, the team stated that the system will still be viable for remotely monitoring data but an alternative solution would have to be used for performing lossless logging of the bus communication until a system update would fix the issue. As for the other requirements, the tables in section 5.4.4 list all requirements and their fulfillment status.

Most importantly, the goal of creating an open platform free of licensing problems which incorporates the team's toolchain as per requirements has been achieved. Even for the configuration of the system, the team can rely on their own toolchain by using the same file formats used in configuring the CAN bus itself. Moreover, the system can be further developed by the team since its source code is open and the system has been extensively documented (see appendix A). Thus, the team is now able to adapt the system to any new toolchain parts they want to use if this becomes necessary. Lastly, the graphical user interface provides the team with an easy to use solution enabling less trained personnel to operate a

car, thus freeing human resources to be employed elsewhere. All of these requirements were of major concern to different stakeholders throughout the team. The system has hence been handed over to the team, which will continue development.

Future Work There are many opportunities for future work in this project. The team has been advised to start with the following issues.

First and foremost, a different hardware setup should be evaluated to alleviate the data throughput issue raised during testing. Hardware in similar form factors is available using processors more common in the automotive industry which have CAN controllers built into the processor die, albeit at a higher price point. An evaluation whether or not this trade-off is viable should be made and a different hardware platform should be built accordingly.

Furthermore, testing in the production environment should be performed as soon as possible. As no car was available for driving during the making of this work, only workshop tests have been conducted so far. The team plans to perform these tests as soon as the new car is available for testing.

Another issue raised by the team addressed the flexibility of the data prioritization as it is currently implemented. The team wishes for the system to dynamically select when to enter priority mode and only send data specified by the priority list, which currently is not implemented. An approach using the Simple Network Management Protocol (SNMP) interface of the Ubiquiti Bullet^{TM} has been suggested to the team.

There is also work to be done in terms of the system's packaging. The team has already started developing a case capable of fulfilling all the stated physical requirements. This should be completed and evaluated.

Acknowledgments

I would like to thank Dr. Nils gentschen Felde for allowing me to write this thesis in a Formula Student setting and for his patience and support in all matters. Furthermore, I thank Tobias Guggemos who has provided valuable feedback these past weeks. Lastly, I want to thank everybody at municHMotorsport for the team spirit I have had the pleasure to experience as a member of the team.

A. Appendix: Software Documentation

Please find attached the documentation document provided with the software. It has been generated using Doxygen [Dox] from comments in the source code.

RaceControl

1.0

Generated by Doxygen 1.8.11

Contents

Chapter 1

About

Controller Are Network (CAN) bus is a data transportation system widely used in the automotive and aerospace industry because it is robust and simple. It features

- data transmission over twisted pair cable which gives it very good electromagnetic immunity without additional shielding
- daisy chaining is possible if done correctly
- an exceedingly low residual failure probability of *message_error_rate* $*$ *4.7* $*$ *10^-11* and it implements the physical and the data link layer in the OSI model. For more information, the `official Bosch document` has it. Another good source is the `CAN bus Wikipedia page`.

If you've ever had to operate any vehicle carrying data on the CAN bus, there's a fair chance that you have been required to use licensed, closed source, non adaptable software. There are open source programming libraries available (`socketcan` and `can4linux` are the best known examples, but there is more), but starting this deep down may be cumbersome to you or you just want to (or have to) get something running quickly. In any case, the licensed tools are still widely used and in fairness well supported, but generally bundled with hardware and, by trend, rather expensive.

`RaceControl` provides a free and open source alternative that runs on Linux, thus leaving the choice of hardware to you (it's tested on Raspberry Pi, Raspberry Pi 2 and several Intel CPU based laptops). It uses `socketcan`, which means the choice of CAN adapter is also left to you as long as it's supported by `socketcan`.

What does it do?

`RaceControl` logs CAN data to files, the name of which you can specify through the configuration file, and it provides data as a web service to be consumed by your browser. That means you can use more or less any device to look at your data (small screens don't work as well for plots and currently, they don't resize either, so you may want to use something upward of iPad size). The data it provides via web service is also configurable via DBC files, uploaded to the `.config/racecontrol/dbc` directory, which sits in the home directory of the user you are running `RaceControl` as. Your loggings can be accessed through a browser as well, given that your `nginx` is also configured correctly. For more on this see *Install and Setup*.

What doesn't it do?

It doesn't guarantee full data integrity. More precisely, it doesn't guarantee to preserve message order or to even fully cover data. Realistically, you can expect it to log about 20% of bus load on a 1MBit/s CAN bus. Furthermore, it specifically doesn't provide a sophisticated data analysis user interface. The data analysis interface it provides is very basic. You are provided with a CSV file to read into a data analysis tool of your own choosing.

How is it licensed?

It uses the GNU Public License version 3.

Install

When in the project directory, execute `sudo pip3 install .` (or alternatively `python3 setup.py install`) to install it and its dependencies for your local `python3`.

CAN bus and web server

For it to work, you need to fire up your CAN interfaces and configure your `nginx` web server. The CAN interfaces, aside from the hardware need the Linux kernel modules called `can` and `can-raw` and if you're using a serial CAN interface also `slcan`. `vcan` provides a virtual CAN interface to be used on machines where no CAN hardware is installed and of course for testing. Other modules for different drivers are also available; please consult the respective documentation for your system. On a Raspbian, you need the tool `rpi-source` to download, build and install the missing modules. Elsewhere, just get the code for your kernel (`uname -a`) from `kernel.org` and build and install using that (the `rpi-source` page has tutorials for this which are helpful even if you're not using `rpi-source` but are compiling from kernel code downloaded directly). As soon as you have these modules compiled and installed and loading on boot (put them in a file called `/etc/modules-load.d/can.conf` for this, with each module name on a new line), you can use the scripts provided in this package to start. `vcan_start` starts the virtual CAN interface, `slcan_start` starts the serial CAN interface. `slcan_start` has to be provided with two arguments specifying bus speed and and interface name of your choosing (for more information on the bus speed, execute `slcan_start` without arguments on the command line).

As for the web server: I recommend you use `nginx`, although any other web server capable of proxying WSGI applications should work. `nginx`, if you are using it, must be configured thusly:

```
events {
    worker_connections  1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;

    server {
        listen       80;
        server_name  your.racecontrol.server;

        location ^~ /loggings {
          alias /var/www/loggings/;
          autoindex on;
        }

        location / {
            proxy_pass      http://127.0.0.1:5000/;
            proxy_redirect  off;

            proxy_set_header Host            $host;
            proxy_set_header X-Real-IP       $remote_addr;
            proxy_set_header X-Forwared-For $proxy_add_x_forwarded_for;

            proxy_set_header Connection '';
            proxy_http_version 1.1;
            chunked_transfer_encoding off;

            proxy_buffering off;
            proxy_cache off;
        }
    }
}
```

This is not a complete `nginx` configuration. Please see the default nginx.conf for details.

In order to make your system plug and play, it is suggested to have the init system (`systemd`, `runit`, ...) start `nginx` and also `RaceControl`. Chances are `nginx` already exists as a `systemd`-service in your system (if it uses `systemd`; otherwise, consult the corresponding documentation) and you only need to enable it (`sudo systemctl enable nginx`) after installing. For `RaceControl`, you have to create a service. Please consult the documentation of your init system for details.

Further Information

This documentation contains only information concerning the Python backend of RaceControl. Please refer to `the Twitter Bootstrap documentation` for information concerning the HTML in use, and to the code itself located in `racecontrol/static/js/raceflot.js`, which has been heavily commented, for information concerning the JavaScript.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

Class Documentation

4.1 racecontrol.cancom.CANCom Class Reference

Public Member Functions

- def **__init__** (self, blacklist, interface=CAN_IFACE, listeners=[ ])
- def **add_listener** (self, listener)
- def **run_notifier** (self, timeout=None)
- def **stop_notifier** (self)
- def **operate** (self)

Public Attributes

- **blacklist**
- **bus**
- **interface**
- **listeners**
- **buffer**
- **notifier**
- **running**

4.1.1 Detailed Description

```
CANCom class for establishing the CAN bus connection and
sending/receiving on it.

The CANCom class, when instantiated, establishes the CAN bus connection and
starts a thread to transmit CAN data received from other application
sources. It also instantiates a can.Notifier object which is itself a
threaded listener on the CAN bus and connects it to the listeners it knows.
```

4.1.2 Member Function Documentation

4.1.2.1 def racecontrol.cancom.CANCom.add_listener (*self, listener*)

```
Method to add a listener to a CANCom object. All listeners in a CANCom
object are notified when a message is read from the bus.
```

4.1.2.2 def racecontrol.cancom.CANCom.operate (*self*)

```
Method which is the target of the thread. It listens to the object's
message buffer for messages from the other application members and, if
there is a message, sends it to the CAN bus after filtering it with the
blacklist.
```

4.1.2.3 def racecontrol.cancom.CANCom.run_notifier (*self, timeout* = None)

```
Method to create a new Notifier on the CANCom object's bus. Returns an
instantiated can.Notifier object. Used to create a new notifier object
whenever a listener is added.
```

4.1.2.4 def racecontrol.cancom.CANCom.stop_notifier (*self*)

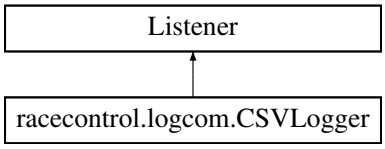
```
Method to stop the current notifier. Used whenever a listener is added
to stop the old notifier so the thread won't become zombied in the
interpreter.
```

The documentation for this class was generated from the following file:

- racecontrol/cancom.py

4.2 racecontrol.logcom.CSVLogger Class Reference

Inheritance diagram for racecontrol.logcom.CSVLogger:



Public Member Functions

- def **__init__** (self, device, interface, filename)
- def **on_message_received** (self, msg)
- def **__del__** (self)

Public Attributes

- **device**
- **interface**
- **flushstamp**
- **filename**
- **file**

4.2.1 Detailed Description

```
Implements the can.Listener interface and writes messages in RaceControl
CSV files.

This class implements the can.Listener interface, which makes it callable
from objects of the can.Notifier class with can.Message objects. On
instantiation, it sets a timeout counter for flushing to file in case the
file is downloaded intermittently and opens a file with user defined
filename.
```

4.2.2 Constructor & Destructor Documentation

4.2.2.1 def racecontrol.logcom.CSVLogger.__del__ (*self*)

```
Standard method called when the interpreter's garbage collector picks
the object up. Since data is flushed via timeout and the garbage
collector should close the open file object eventually as well, this is
not technically necessary, but added for safety.
```

4.2.3 Member Function Documentation

4.2.3.1 def racecontrol.logcom.CSVLogger.on_message_received (*self, msg*)

```
Implements can.Listener's on_message_received method. Then proceeds to
join all elements in the can.Message object it gets passed into a
RaceControl CSV string and writes the string to file.
```

The documentation for this class was generated from the following file:

- racecontrol/logcom.py

4.3 racecontrol.netcom.Dispatcher Class Reference

Public Member Functions

- def **__init__** (self, netcom, prioritylist, port=D_PORT, timeout=100)
- def **priorityfilter** (self, msg)
- def **dispatch** (self, payload)
- def **operate** (self)

Public Attributes

- **buffer**
- **prioritylist**
- **priority_set**
- **netcom**
- **port**
- **sock**
- **timeout**
- **trigger**
- **running**

4.3.1 Detailed Description

```
Handles dispatching can.Message objects over the UDP connection.

The Dispatcher object is in charge of sending CAN messages over the UDP
connection. It offers a priority_set variable which can be used to set
priority mode. It also holds a can.BufferedReader receiving connections
from other RaceControl objects. It starts a thread to operate the
connection using the operate() method.
```

4.3.2 Member Function Documentation

4.3.2.1 def racecontrol.netcom.Dispatcher.dispatch (*self, payload*)

```
Dispatches bytearrays to all nodes known to the NetCom object this
Dispatcher object is connected to.
```

4.3.2.2 def racecontrol.netcom.Dispatcher.operate (*self*)

```
Target for the thread. Reads messages from the message buffer and puts
them into the priority filter. If they come back and are not empty, the
operate() method puts them into the dispatch() method. Furthermore, it
dispatches a register protocol messages to all nodes as a keepalive
message every few seconds in case there are no CAN messages to be
transmitted.
```

4.3.2.3 def racecontrol.netcom.Dispatcher.priorityfilter (*self, msg*)

```
If priority_set is set to True, this method filters messages through
the priority list and returns None if the message is not in the list.
If priority_set is set to False, it returns the msg instantly.
```

The documentation for this class was generated from the following file:

- racecontrol/netcom.py

4.4 racecontrol.webcom.GUICom Class Reference

Public Member Functions

- def **__init__** (self, queue, msgfilter)

Public Attributes

- **queue**
- **msgfilter**
- **buffer**
- **permsg**

4.4.1 Detailed Description

```
Gets can.Message objects and parses them through the msgfilter necessary
to create web interface suited CSV data.

The GUICom class has a threadsafe and processsafe queue it pushes the
generated message to and a can.BufferedReader object which is notified by
other RaceControl objects with can.Message objects. It starts a thread
which runs the web interface data generator.
```

The documentation for this class was generated from the following file:

- racecontrol/webcom.py

4.5 racecontrol.logcom.LogCom Class Reference

Public Member Functions

- def **__init__** (self, logdir=LOGDIR, fileformat=FILEFORMAT)
- def **loggers** (self)

Public Attributes

- **device**
- **interface**
- **logdir**
- **fileformat**
- **csv_logger**

4.5.1 Detailed Description

```
Instatiates CSVLogger objects with user defined timestamps as filename
patterns.

The LogCom class, when instantiated, parses the input strings containing
the timestamping patterns for the logging filename through the arrow
library, which provides beautifully formatted strings with timestamps. It
then creates a CSVLogger object it uses to log messages to a file.
```

4.5.2 Member Function Documentation

4.5.2.1 def racecontrol.logcom.LogCom.loggers (*self*)

```
Function which returns the current loggers owned by the LogCom object.
The main function of the RaceControl package loops through this and
distributes the objects to the other application members. Loggers have
to implement the can.Listener interface for this to work.
```

The documentation for this class was generated from the following file:

- racecontrol/logcom.py

4.6 racecontrol.netcom.NetCom Class Reference

Public Member Functions

- def **__init__** (self, prioritylist, ip='192.168.11.26', udpport=D_PORT, listeners=[ ], node_ips=NODES)
- def **add_listener** (self, listener)
- def **notify** (self, msg)
- def **add_node** (self, node_ip, last_msg=None)
- def **ip_list** (self)
- def **check_nodes** (self)

Public Attributes

- **ip**
- **udpport**
- **listeners**
- **nodes**
- **dispatcher**
- **running**

4.6.1 Detailed Description

```
Handles UDP network communication.

The NetCom class instantiates Node objects for all node IPs give in the
config file and registers them. It then broadcasts the protocol message
used to register with other nodes and starts threads for both the UDP
server and the watchdog. The watchdog takes care of checking node activity
and remove inactive nodes from the NetCom object's register.
```

4.6.2 Member Function Documentation

4.6.2.1 def racecontrol.netcom.NetCom.add_listener (*self, listener*)

```
Adds can.Listeners to the NetCom object. These are notified by other
objects with can.Message objects. If the listener added is not a
can.Listener, a TypeError is raised.
```

4.6.2.2 def racecontrol.netcom.NetCom.add_node (*self, node_ip, last_msg* = None)

```
Adds Node objects to the NetCom object.
```

4.6.2.3 def racecontrol.netcom.NetCom.check_nodes (*self*)

```
Watchdog target method. Checks when a message was last received from
all the nodes and removes Node objects from the NetCom object's
register if they have been inactive for 5 seconds.
```

4.6.2.4 def racecontrol.netcom.NetCom.ip_list (*self*)

```
Returns the IPs of all Node objects registered with the NetCom object.
Used to return IP list externally.
```

4.6.2.5 def racecontrol.netcom.NetCom.notify (*self, msg*)

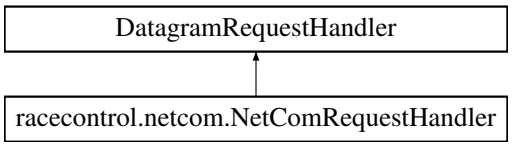
```
Notifies objects from classes that implement the can.Listener interface
with can.Message objects. Used when messages are received over the UDP
connection.
```

The documentation for this class was generated from the following file:

- racecontrol/netcom.py

4.7 racecontrol.netcom.NetComRequestHandler Class Reference

Inheritance diagram for racecontrol.netcom.NetComRequestHandler:



Public Member Functions

- def **handle** (self)

4.7.1 Detailed Description

```
Inherits from socketserver.DatagramRequestHandler to handle UDP
requests.

For further information, read the handle() methods documentation or the
Python library documentation for socketserver.
```

4.7.2 Member Function Documentation

4.7.2.1 def racecontrol.netcom.NetComRequestHandler.handle (*self*)

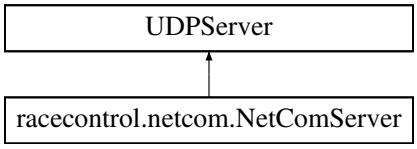
```
This method reimplements the handle() method from
socketserver.DatagramRequestHandler. It reads the incoming message.
Through its own sender variable, it accesses the NetCom object
associated with its UDP server object and checks if the source of the
received message is in the node registry. If so, the timestamp for the
node is reset and the message is filtered for protocol words, then
passed to the NetCom object's notify() method. If not, the message is
checked for protocol words. In case this check is successful, the
source IP is passed to the NetCom object's add_node() method to
register it and an appropriate response is generated and sent back to
the source.
```

The documentation for this class was generated from the following file:

- racecontrol/netcom.py

4.8 racecontrol.netcom.NetComServer Class Reference

Inheritance diagram for racecontrol.netcom.NetComServer:



Public Member Functions

- def **__init__** (self, server_address, NetComRequestHandlerClass, netcom)

Public Attributes

- **netcom**

4.8.1 Detailed Description

```
Inherits the socketserver.UDPServer class.

Addition made to the parent's __init__() method: Stores a reference to an
associated NetCom object. (Should be a weak reference to not confuse the
garbage collector but currently isn't.)
```

The documentation for this class was generated from the following file:

- racecontrol/netcom.py

4.9 racecontrol.netcom.Node Class Reference

Public Member Functions

- def **__init__** (self, ip, last_msg=None)

Public Attributes

- **ip**
- **last_msg**
- **timestamp**

4.9.1 Detailed Description

```
Holds node data for all network communication partners.

The Node class holds every communication partner's ip, the last message
received from the partner and the time of reception.
```

The documentation for this class was generated from the following file:

- racecontrol/netcom.py

4.10 racecontrol.racecontrol.RaceControl Class Reference

Public Member Functions

- def **__init__** (self)
- def **run** (self)

Public Attributes

- **cancomd**
- **netcomd**
- **logcomd**
- **webcomd**
- **guicomd**

4.10.1 Detailed Description

```
Central RaceControl application.

Here, the run() method is located, which starts RaceControl.
```

4.10.2 Member Function Documentation

4.10.2.1 def racecontrol.racecontrol.RaceControl.run (*self*)

```
Main method of the RaceControl application. From here, the
configuration files are read and all objects are created and connected
and the server for the web application is started. This script is
installed on the target machine's /usr/local/bin and can then be
executed from the command line. Be sure to execute racecontrol as
superuser so the /var/www/loggings directory is writable to store
loggings under.
```

The documentation for this class was generated from the following file:

- racecontrol/racecontrol.py

4.11 racecontrol.webcom.WebCom Class Reference

Public Member Functions

- def **__init__** (self)

Public Attributes

- **app**
- **subscriptions**
- **msgqueue**

4.11.1 Detailed Description

```
Handles serving web content to the user.

The WebCom class holds the Flask object which serves web content to the
user. Therein defined are methods for route responses, which encode data
from the WebCom object's message queue into a server-side event data string
and pass it on to clients.
```

The documentation for this class was generated from the following file:

- racecontrol/webcom.py

Index

List of Figures

Bibliography

[2D] 2D Debus & Diebold Meßsysteme GmbH. Datasheet PDF of LG-muCAN11_Pro-000, PDF. http://2d-datarecording.com/Downloads/Datasheets/Logger/Pdf/LG-%C2%B5CAN11_Pro-000-DINA4.pdf.

[Alw] Alwinner Technologies Co., Ltd. A20 User Manual. http://dl.linux-sunxi.org/A20/A20%20User%20Manual%202013-03-22.pdf.

[Bra97] S. Bradner. IETF RFC 2119: Key words for use in RFCs to Indicate Requirement Levels. 1999. *Internet Engineering Task Force: http://www. ietf. org/rfc/rfc2119. txt*, March 1997.

[Bra14] Nils Braune. Telemetry Unit for a Formula Student Race Car. Semester thesis, Eidgenössische Technische Hochschule Zürich, 2014.

[can] can4linux project page on Sourceforge. https://sourceforge.net/projects/can4linux/.

[Com] The Qt Company. Website of Qt, a popular framework for creating graphical user interfaces. https://www.qt.io/.

[Cop09] David Rua Copeto. Automotive data acquisition system - FST. M.eng. thesis, Universidade Técnica de Lisboa, 2009.

[ddspceG] dSPACE digital signal processing and control engineering GmbH. Website of dSPACE, a manufacturer for ECUs. https://www.dspace.com/.

[DH04] Alex Sung David Hadaller, Herman Li. Formula SAE Telemetry Collection Unit. Project report, University of Waterloo, CA, 2004.

[Dox] Doxygen. Website of the Doxygen documentation generator. http://www.doxygen.org/.

[dSP] dSPACE GmbH. Datasheet of the MicroAutoBox II by dSPACE. https://www.dspace.com/shared/data/pdf/2013/ProductBrochure_MicroAutoBox-HW_E_ebook.pdf.

[e.V15] Formula Student Germany e.V. Formula Student Germany Rules 2016. https://www.formulastudent.de/uploads/media/FSG_Rules_2016_v1.0.0_v20151210.pdf, 2015.

[f1r] Wikipedia entry about rules changes in Formula One over the years. https://en.wikipedia.org/wiki/History_of_Formula_One_regulations.

[FGM+99] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol — HTTP/1.1. RFC 2616, RFC Editor, Fremont, CA, USA, June 1999.

[Foua] Free Software Foundation. GNU Public License Website. `http://www.gnu.org/licenses/licenses.en.html`.

[Foub] Raspberry Pi Foundation. Website of the Raspbian Operating System. `https://www.raspbian.org/FrontPage`.

[fsg] German Wikipedia entry about Formula Student Germany. `https://de.wikipedia.org/wiki/Formula_Student_Germany`.

[ftlcp] Package Maintainer for the linux-can project. Linux-CAN/SocketCAN user space applications (available as Debian package). `https://github.com/linux-can/can-utils`.

[GCB03] Jose A. Gutierrez, Edgar H. Callaway, and Raymond Barrett. *IEEE 802.15.4 Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensor Networks*. IEEE Standards Office, New York, NY, USA, 2003.

[Ger] Formula Student Germany. Website of Formula Student Germany. `http://www.formulastudent.de/`.

[Gmba] Hockenheim-Ring GmbH. Website of the Hockenheimring, a racetrack in Germany. `http://www.hockenheimring.de/`.

[Gmbb] PEAK-System Technik GmbH. User Manual for the PCAN-USB CAN Interface for USB. `http://www.peak-system.com/produktcd/Pdf/English/PCAN-USB_UserMan_eng.pdf`.

[Haa07] Sebastian Haase. Telemetrie im Formula Student Rennwagen auf Basis von CAN Bus, Datenspeicherung und Wireless LAN Technologien. B.sc. thesis, University of Applied Sciences Hamburg, 2007.

[Hit] Hitex GmbH. Online shop for TriCore™ starter kits. `http://www.ehitex.de/en/starter-kits/for-tricore/`.

[IEE07] IEEE Std 802.11-2007. IEEE standard for information technology — telecommunications and information exchange between systems — local and metropolitan area networks — specific requirements — part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications, June 2007.

[Inc] NGINX Inc. Website of the nginx web server. `http://nginx.org/en/`.

[Inf] Infineon Technologies AG. TriCore™ TC1796 Microcontroller Datasheet. `http://www.infineon.com/dgdl/Infineon-SAK-TC1796-256F150E+BE-DS-v01_00-EN.pdf?fileId=5546d46249a28d750149a34e1f28045d`.

[Int] SAE International. Website of the SAE International, an international association of engineers focused on automotive, aerospace and commercial vehicle engineering. `http://www.sae.org/`.

[Int15] SAE International. 2016 Formula SAE Rules. `http://www.fsaeonline.com/content/2016_FSAE_Rules.pdf`, 2015.

[LKO] Inc. Linux Kernel Organization. The Linux Kernel Archives. `https://www.kernel.org/`.

[Ltd] MoTeC Pty Ltd. Website of MoTeC, an Australian manufacturer for racing ECUs. `http://www.motec.com/home`.

[MoT] MoTeC Pty Ltd. Brochure for the MoTeC M800, also containing the datasheet. `http://www.motec.com/filedownload.php/M800%20M880%20Brochure.pdf?docid=2392`.

[mun] municHMotorsport. Website of municHMotorsport, Formula Student team from University of Applied Sciences Munich. `https://www.munichmotorsport.de`.

[Pre01] Defense Acquisition University Press. Systems engineering fundamentals. 2001.

[Rob91] Robert Bosch GmbH. CAN Specification Version 2.0. `http://www.bosch-semiconductors.de/media/ubk_semiconductors/pdf_1/canliteratur/can2spec.pdf`, 1991.

[Rob12] Robert Bosch GmbH. CAN FD Specification Version 1.0, 2012.

[Ron] Armin Ronacher. Website of the Flask web framework for Python. `http://flask.pocoo.org/`.

[SAE] Formula SAE. Website of the Formula SAE. `http://fsaeonline.com/`.

[Sha05] Y. Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180 (Informational), October 2005.

[SMS] SMSC. LAN9512 USB 2.0 Hub and 10/100 Ethernet Controller Datasheet. `http://ww1.microchip.com/downloads/en/DeviceDoc/9512.pdf`.

[soc] Kernel documentation for socketcan. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/plain/Documentation/networking/can.txt`.

[syr98] Ieee guide for developing system requirements specifications. *IEEE Std 1233, 1998 Edition*, pages 1–36, Dec 1998.

[Tex] Texas Instruments Inc. AM335x Sitara^{TM} Processors Technical Reference Manual. `http://www.ti.com/lit/ug/spruh73m/spruh73m.pdf`.

[Tho] Thomas Fischl. USBtin EB - USB to CAN interface board Datasheet. `http://www.fischl.de/usbtin/USBtin_EB_v2.pdf`.

[Ubia] Ubiquiti Networks, Inc. Datasheet of the Ubiquiti Bullet^{TM}, a commercial outdoor wireless bridge, PDF. `https://dl.ubnt.com/datasheets/bulletm/bm_ds_web.pdf`.

[Ubib] Ubiquiti Networks, Inc. Datasheet of the Ubiquiti PicoStation^{TM}, a commercial outdoor wireless bridge, PDF. `https://dl.ubnt.com/datasheets/picostationm/picom2hp_DS.pdf`.

[un] Github user "notro". Github Repository for the Raspberry Pi Kernel Source Installer. `https://github.com/notro/rpi-source/wiki/`.

[Uni] Defense Acquisition University. Website of the Defense Acquisition University. `http://www.dau.mil/`.

[Veca] Vector Informatik GmbH. Datasheet of GL Logger family, online print version. `http://vector.com/vi_logger_comparison_portal_iframe_en.druck`.

[Vecb] Vector Informatik GmbH. Fact Sheet for CANdbLib, the software library for using CAN databases. `http://vector.com/portal/medien/cmc/factsheets/CANdbLib_FactSheet_EN.pdf`.