Fleet Monitoring System

Student Research Project

Florian Baumgartner

florian.baumgartner@ost.ch

Luca Jost

luca.jost@ost.ch

Advisor

Martin Willi

Examiner

Prof. Beat Stettler

Institute for Networked Solutions

Eastern Switzerland University of Applied Sciences

December 2021

Abstract

Onway AG offers WLAN and network access control solutions and software development. Their main fields of business are solutions for Network Access Control (NAC) as well as communication access for public transport. They are known for developing specialized industrial IoT applications. Onway AG is interested in providing an elegant solution for public transport fleets (e.g. buses) to gather low-level vehicle data and transmit them to a cloud-based system. This information can then be used to monitor the state of the vehicle and inform about possible issues in real-time.

The Fleet Management Systems Interface (FMS) is a standard interface developed by European commercial vehicle manufacturers in 2002. It defines a common interface for telematics applications and includes driving as well as diagnostics information. The data is coded onto a CAN bus. This student research project introduces an FMS monitoring system that transmits real-time data from the vehicle to a server.
The Fleet-Monitor Hardware was designed from scratch with the goal to make it simple, reliable, and deployable in a vehicle. The system is based around an ESP32-S2 system on chip and all the Software is written in C++. The device connects directly to the CAN-Bus, collects and filters incoming data, and forwards it to a host device over Ethernet or WiFi. Additionally, an accelerometer was added to monitor information about the driving performance. A file system was implemented to easily access the configuration. The configuration can be uploaded via the USB port or over the air.

The hardware was then tested to assess its functionality. Recorded data from a bus driving in Germany, as well as a J1939 simulator, were used to evaluate the system performance. These tests showed that the system is promising and satisfies all of the given requirements.

Contents

Acronyms

AI Artificial Intelligence. 37

AP Access Point. 30, 37, 44, 45

ASCII American Standard Code for Information Interchange. 30, 44

CAD Computer Aided Design. 21

CAN Controller Area Network. 6, 17–19, 25, 26, 34, 43, 45, 48

CDC Communications Device Class. 29

CM Common-Mode. 26

CPU Central Processing Unit. 23

CRC Cyclic Redundancy Check. 18

CSV Comma-separated values. 37

DC Direct Current. 22, 26

DFU Device Firmware Update. 23, 30, 42

DHCP Dynamic Host Configuration Protocol. 32, 44

DIN Deutsches Institut für Normung. 26

FAT File Allocation Table. 28

FMS Fleet Management System. 6, 20, 26, 30, 35, 37–41, 45, 47, 48

GUI Graphical User Interface. 38, 39

HTTP Hypertext Transfer Protocol. 5, 6, 33, 35, 37, 45, 47

I²C Inter-Integrated Circuit. 26

IDE Integrated Development Environment. 28, 42

IEEE Institute of Electrical and Electronic Engineers. 24

IoT Internet of Things. 3, 48

IP Internet Protocol. 24, 30, 32, 41, 44, 45

ISO International Organization for Standardization. 17, 25

JSON JavaScript Object Notation. 29–31, 33, 35, 37, 38

JTAG Joint Test Action Group. 23

LED Light-emitting Diode. 43–45

MIT Massachusetts Institute of Technology. 28

MSC Mass Storage Controller. 29

OEM Original Equipment Manufacturer. 20

OTA Over-the-Air. 47

PC Personal Computer. 41

PCB Printed Circuit Board. 5, 21, 23

PDU Protocol Data Unit. 19

PGN Parameter Group Number. 19, 20, 30, 38

PHY Physical Layer. 24

ppm parts per million. 33

PTC Positive Temperature Coefficient. 22

RF Radio Frequency. 23, 44

RGB Red Green Blue. 43

RTR Remote Transmission Request. 18

SAE Society of Automotive Engineers. 19, 20

SoC System on a Chip. 3, 6, 23, 29

SPI Serial Peripheral Interface. 23, 24, 26, 28, 29

SSID Service Set Identifier. 30, 41

TCP Transmission Control Protocol. 24

TWAI Two Wire Automotive Interface. 25

URL Uniform Resource Locator. 37

USB Universal Serial Bus. 22, 23, 28, 29, 41, 42, 46, 48

VCP Virtual COM Port. 29

WEP Wired Equivalent Privacy. 44

WPA Wi-Fi Protected Access. 44

Glossary

Arduino Is an open-source company providing software libraries and microcontroller kits. 28, 48

CANOpen Is a communication protocol and device profile specification for embedded systems used in automation. 19

DeviceNet Is a network protocol used in the automation industry to interconnect control devices for data exchange. 19

ESP-IDF Is Espressif's official IoT development framework for the ESP32 lineup. 28, 48

ESP32 Is a series of low-cost, low-power System on a Chip microcontrollers with integrated Wi-Fi. 3, 6, 23–26, 29, 33, 41, 42, 44, 47

OpenAI Codex Is an artificial intelligence model developed by OpenAI. It parses natural language and generates code in response. 37

PlatformIO Is a professional collaborative platform for embedded development. 28

Unix Is a system for describing a point in time. It is the number of seconds that have elapsed since the Unix epoch. 33

1 Introduction

1.1 Background

Managing a large fleet of vehicles requires enormous effort. In order to minimize downtime and offer seamless operation, it is important to monitor the whole fleet on the road. Costs associated with operation, fuel, and maintenance can quickly mount. To ensure that fleet operations are as efficient and cost-friendly as possible, solutions are required to identify and eliminate any unnecessary expenditures. But it is not all about the vehicle itself, there are other factors to be considered. During vehicle operation, the driver can be assisted and warned to ensure safe driving behavior. In general, fleet monitors can be used to improve efficiency, safety, and quality of fleet operations through the use of internet-connected sensors and software.

1.2 Scope

A dedicated device has been developed according to the requirements of our industry partner. A custom PCB was developed and manufactured. To house the electronics, a water-resistant case was selected and mechanical drawings were created to machine it in the workshop.

Firmware was written to handle all aspects of operation. An HTTP server was set up on a Raspberry Pi to receive and log incoming data. Furthermore, software utilities were developed to help the user visualize the collected data and configure the final product.

The Fleet-Monitor was tested over an extended period with a J1939 simulator to prove its functionality. Documentation, as well as a comprehensive manual, was written for future use.

1.3 Approach

Developed by European commercial vehicle manufacturers, the Fleet Management System (FMS) is a common standard for telematics applications. Both driving and diagnostic information can be gathered through this interface. This data is coded to a CAN bus, which makes it accessible to third-party devices. For the collection of vehicle data via the FMS interface, a dedicated device was developed.

The Fleet-Monitor is based on an ESP32-S2 System on a Chip and all of the software is written in C++. Directly connected to the FMS interface, the device collects and filters incoming data, then transmits it over Ethernet or WiFi to the host device. Furthermore, a motion sensor was incorporated to monitor driving performance.

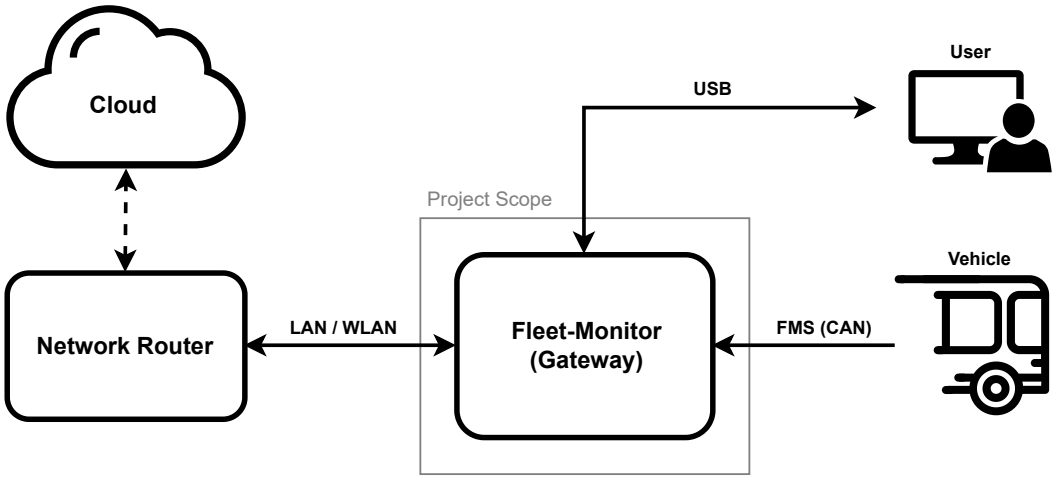


Figure 1.1: System Overview

Utility tools for the generating configuration files, visualizing data, and a HTTP server were developed. These are needed to support the hardware in its operation.

1.4 Open Source

From the beginning, it was decided that everything about the project would be released under an open source license. Both of us are huge supporters of open source and believe it will be the future of engineering. Building upon existing libraries and code under open source licenses, allows us to accelerate the design process. Sometimes open source is considered an act of charity, but in our case, the benefits of using it outweigh any closed source processes. All documents and files for this project can be found on our GitHub page. A short description of all the repositories can be found in the Appendix A.8.

2

Requirements Specification and Project Schedule

2.1 Requirements Specification

OST Ostschweizer Fachhochschule

Onway

Fleet Monitoring System

Requirements Specification

Florian Baumgartner, Luca Jost

Advisor: Martin Willi
Supervisor: Prof. Beat Stettler

October 12, 2021

Acronyms

CAN Controller Area Network. 3–5

FMS Fleet Management System. 3–7

GNSS Global Navigation Satellite System. 7

IMU Inertial Measurement Unit. 3–5, 7

IoT Internet of Things. 3

IP Internet Protocol. 3

LAN Local Area Network. 3–5

LED Light-emitting Diode. 4

NAC Network Access Control. 3

PCB Printed Circuit Board. 4

SAE Society of Automotive Engineers. 5

SoC System on a Chip. 2, 4

USB Universal Serial Bus. 3–5

WLAN Wireless LAN. 3–5

Glossary

ESP32 Is a series of low-cost, low-power System on a Chip microcontrollers with integrated Wi-Fi. 4, 5

1 Introduction

Onway AG offers WLAN and network access control solutions and software development. Their main fields of business are solutions for Network Access Control (NAC) as well as communication access for public transport. They are known for developing specialized industrial IoT applications. Onway AG is interested in providing an elegant solution for public transport fleets (e.g. buses) to gather low-level vehicle data and transmit them to a cloud-based system. This information can then be used to monitor the state of the vehicle and inform about possible issues in real time.

2 Task Definition

To collect vehicle data of the Fleet Management System (FMS) and provide IP-based access, a dedicated device has to be developed. It acts as a gateway (communication bridge) and connects the internal CAN-Bus to the on-board network router via LAN or WLAN.
A general block diagram of the system is shown in Figure 2.1.

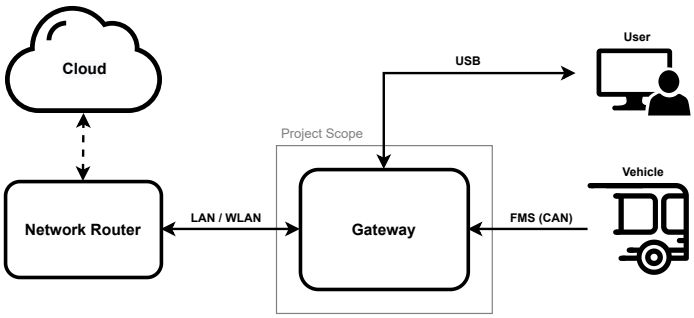


Figure 2.1: *System Block Diagram*

FMS-Packets are received and processed by the gateway. A configurable filter decides which packets get forwarded and limits the maximal transmission update rate. The network router acts as a receiver, which can upload the information over the internet to a cloud-based system. Device settings as well as the filter configuration can be updated by the user without the need of re-uploading a new firmware image. The user can access the device by an USB interface.
To gather additional motion data of the vehicle, an Inertial Measurement Unit (IMU) will be integrated. The collected motion data gets processed and will be available for further usage.

Various use cases were defined in Section 4.

3 Product Requirements

3.1 Hardware

The custom built hardware contains all necessary components integrated on a single Printed Circuit Board (PCB). The hardware needs to comply with the following requirements:

- The hardware shall be based around an ESP32 System on a Chip (SoC).
- The hardware shall use LAN/WLAN to communicate with the network router.

  The Ethernet interface shall be galvanically isolated from the rest of the hardware.
- The hardware shall have a CAN interface to receive FMS-packages from the vehicle.

  The CAN interface shall be galvanically isolated from the rest of the hardware.
- The hardware shall have an Inertial Measurement Unit based on a 3 axis accelerometer.
- The hardware shall be able to show the current device status (e.g. LEDs).
- The hardware shall be capable of operating with 9 to 28 Volts DC.
- The hardware shall be enclosed in a case in order to minimize dust and moisture getting into the system.
- The hardware should have an USB interface for configuration.

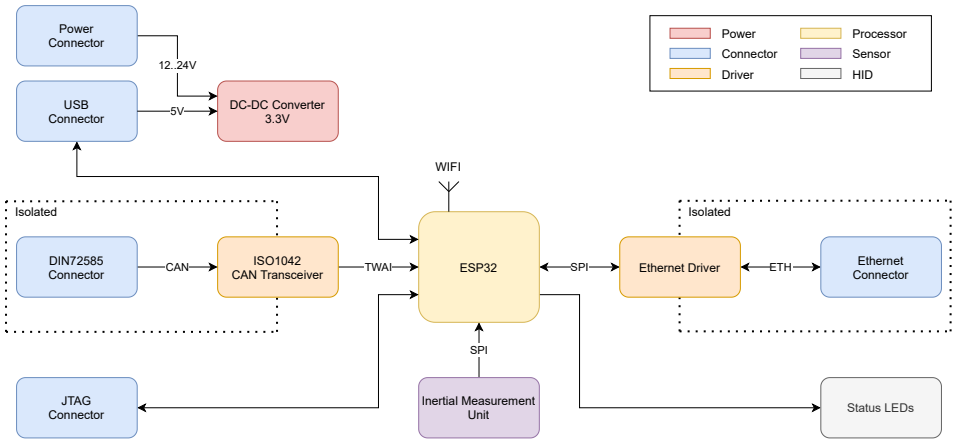


Figure 3.1: *Hardware Block Diagram*

3.2 Protocol and Interface Standards

The following protocol standards will be supported in this project:

- WLAN-Interface: Wi-Fi 802,11b/g/n
- LAN-Interface: 10Base-T / 100Base-TX
- USB-Interface: USB 2.0 (Device)
- CAN-Interface: SAE J1939
- FMS-Protocol: Version 04 (17.09.2021)

3.3 Firmware

The embedded firmware will run on the ESP32 and will be written in C. The requirements for the firmware are as follows:

- The firmware shall read FMS-packages and prepare them for transmission.
- The firmware shall transmit the packages to the router.

  A filter shall be implemented to select the packages to be sent out.

  A filter shall be implemented to select the maximum update rate of a package.

  A filter shall be implemented to send data only on change.
- The firmware shall display the current device status to the user.
- The firmware shall read the IMU-data.
- The firmware shall transmit the IMU-data.
- The firmware should read a configuration file from the router (e.g. the filter of what packages shall be sent).

3.4 Network Router Communication Tool

The network router communication tool is the sole device communicating with the hardware. The system has the following requirements:

- The system shall handle communication between the embedded system and the host.
- The system shall store the streamed data and make it available for further processing.
- The system should host files for the device (e.g. configuration file).

3.5 Configuration File Creator

The configuration file creator helps the user to easily change and adapt the behavior of the FMS-packet transmission. For each FMS command-type, specific filters can be applied (referenced in Section 3.2). The configuration file creator fulfills the following requirements:

- The file creator shall have the function to load and store configuration files.
- The file creator shall provide axes to change the filter settings.

3.6 Graphical Visualizer (optional)

This Python-based tool allows visualization of data that has been gathered and transmitted by the device. It should be implemented as a stand-alone application which primarily can be used offline. The graphical interface helps the user to easily understand the presented data. The requirements for this tool are as follows:

- The graphic visualizer shall open files containing FMS data packets.
- The graphic visualizer shall visualize user selected command-types.

4 Use Cases

4.1 UC1 Driving Quality Analysis

Having a clear understanding of a drivers behavior, is very important for fleet operators. Having data about the vehicle operators can lead to less emissions, improve customer satisfaction and improve the drivers behavior. Our system can be deployed to gather driving data in a very detailed manner. The Inertial Measurement Unit provides accurate motion data of the vehicle, which leads to a better understanding of the driving performance.

4.2 UC2 Vehicle Maintenance Report

Managing a large fleet of vehicles requires enormous effort. In order to minimize down time it is important to preemptively warn about upcoming and present issues. Our system makes it possible to gather continuous information about the status of the fleet without physical access.

4.3 UC3 Real Time Telemetry Data

Most of modern vehicles are already equipped with GNSS receivers for providing location updates to the Fleet Management System. However it is difficult to differentiate if a vehicle is currently parked or just slowly moving (e.g. due to traffic jams). With the help of knowing the exact driving velocity, a better prediction of potential arrival time can be achieved. Additional data, like the state of passenger doors can directly inform the system about a potential delay of departure.

2.2 Project Schedule

Task	Target FB (h)	Target LJ (h)	Actual FB (h)	Actual LJ (h)
Project start				
Problem analysis and concept discussion	8	8	6	6
Administration and project planning	5	5	5	5
Rapid prototyping for concept validation	10		8	
Hardware				
Detail block diagram and core component evaluation		5		3
Schematic design, bill of material (BOM) and **Component ordering**	12	15	12	10
Mechanical design (e.g. Case evaluation)	5		7	
PCB Layout	12	12	12	12
Review and production preparation	5	5	3	2
Assembly and testing	5	5	15	15
Firmware				
Familiarization with IDE, programmer, RTOS, chip periphery	10		4	
Concept design	5	5	5	5
Implementation of file system	15		25	
Implementation of USB-Interface	20		20	
Implementation of CAN-Interface		5		10
Implementation of LAN/WLAN-Interface		30		45
Implementation of IMU		5		5
Configuration file loading and parsing	15		12	
FMS-Package parser and data packing		30		20
Integration of modules and main application	25	20	25	25
Backend Software				
Implementation of network interface		10		10
Data fetching, processing and storing		15		8
Implementation of file serving for device	5		2	
Graphical Visualizer	10		10	
Configuration file creator	8		17	
System testing				
Use Case 1	8	8	2	2
Use Case 2	5	5	2	2
Use Case 3	5	5	2	2
Documentation				
Requirements Specification / Report	40	40	45	45
Abstract / Poster / Elevator speech (video)	7	7	10	10
Total (h):	240	240	249	242

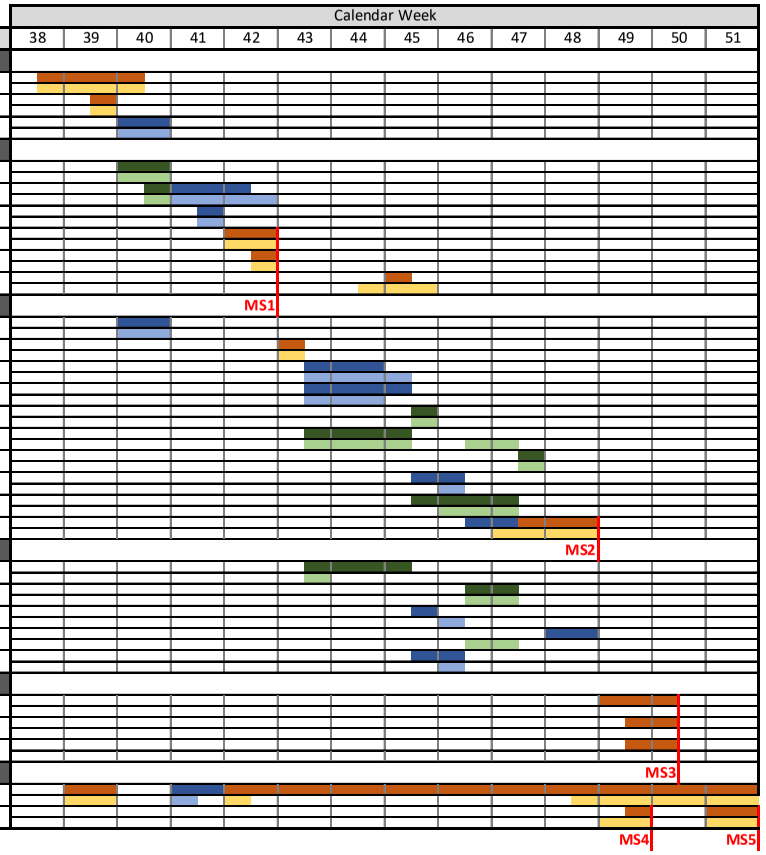


Figure 2.1: Project Schedule

3

Preliminaries

3.1 Controller Area Network (CAN)

3.1.1 Basics

ISO11898 describes the physical data link layer implementation of CAN. This specification describes a twisted-wire pair bus with 120 Ω line impedance, and differential signaling at a rate up to 1 Mbit/s. A network is constructed with two or more transceivers on the same bus lines. The network must be terminated with a 120 Ω resistor at each end of the bus, as shown in Figure 3.1.

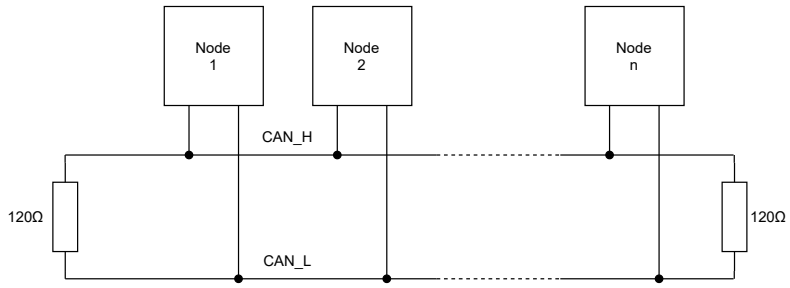


Figure 3.1: Topology of a CAN-Bus

As long as the bus is free, any node is allowed to transmit CAN messages. Each message is received by all nodes on the network, including the node that sent the message. This type of data broadcasting allows multiple nodes to use the transmitted data. It also allows the sending node to monitor the bus for errors. If two or more nodes try to transmit at the same time, the lower priority message will be overwritten, and this lower priority node will halt transmission upon sensing overwritten bits in its message identifier. The message is then re-transmitted when the bus is free again. This non-destructive process is called bit-wise arbitration.

Every node on the network reads the identifier of a message, and each node independently determines if the message is to be ignored or processed. Since the identifier is specific to the contents of the message rather than the identity of the originating node, new nodes may be added to the network, without modifying the firmware of any existing node on the network [2].

3.1.2 Signaling

The signals on the CAN bus are distinguished between two bus logic states, dominant and recessive. A recessive bit is defined as CANH being less than CANL + 0.5 V. A dominant bit is defined as CANH being more than CANL + 0.9 V. Figure 3.2 illustrates the nominal case. Since dominant bits overwrite recessive bits, CAN manages message collision through the process of bit-wise arbitration as described earlier.

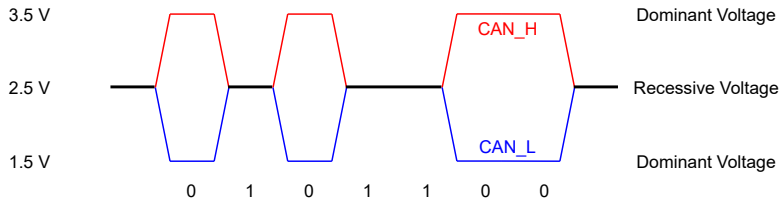


Figure 3.2: CAN-Bus Signaling

3.1.3 Frames

Messages over a CAN bus are referred to as frames. The most common frame types are defined as CAN 2.0A and 2.0B also known as base and extended frame. 2.0A is simply a CAN frame with an 11-bit identifier and 2.0B always uses 29-bit. A CAN frame is always split up into the following regions [4].

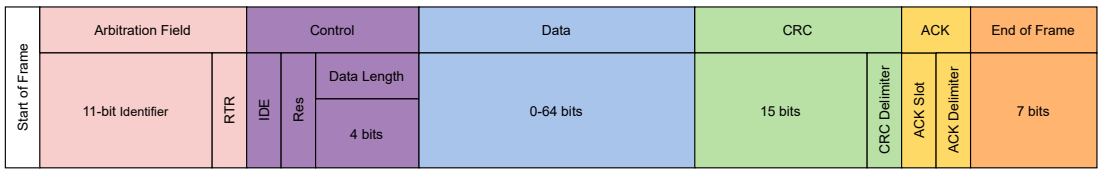


Figure 3.3: Base 11-bit identifier CAN frame

- The Arbitration Field, includes the identifier of the message and defines the priority. The identifier must be 11 or 29 bits long. Also in this field is the Remote Transmission Request (RTR) bit, which is used to request data.
- The Control Field is used to differentiate between the two frame types and defines the incoming data length.
- The data field can contain 0-8 bytes of information.
- The CRC field contains a 15-bit checksum. The checksum is used to detect errors in the transmission.
- The Acknowledgement Slot, is used to detect if a message was received by any of the devices on the bus. The transmitter checks for the presence of the Acknowledge bit and re-transmits the message if no acknowledge was detected.

3.2 SAE J1939

J1939 is the open standard developed by Society of Automotive Engineers (SAE). It is used for networking and communication in the commercial vehicle sector. J1939 is a higher-layer protocol utilizing CAN as its physical layer. SAE J1939 is primarily a data-driven protocol, providing far better data bandwidth than other automation protocols such as CANOpen and DeviceNet [1].

The standard specifies CAN bus speeds of 250 kbit/s or 500 kbit/s and uses the extended 29-bit identifier frame format. Most messages defined by the J1939 standard are intended to be broadcast only. This means that the data is transmitted on the network without a specific destination. This permits any device to use the data without requiring additional request messages. The 29-bit identifier used in J1939 is structured in the following way [8].

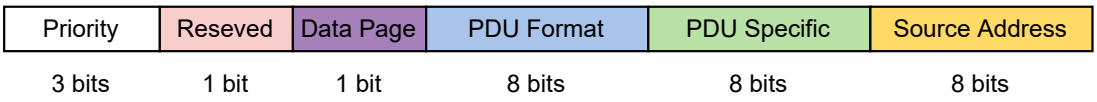


Figure 3.4: 29-bit J1939 Identifier

- The first three bits of the identifier are used for controlling a message priority during the arbitration process. A value of 0 has the highest priority.
- The Reserved bit, Data Page bit, Protocol Data Unit (PDU) Format field and PDU specific field are often grouped together and referred as Parameter Group Number (PGN).
- The last 8 bits of the identifier contain the address of the device transmitting the message. Two devices can not share the same address.

3.3 Fleet Management System (FMS)

The Fleet Management System (FMS) Interface is a standard interface developed by European commercial vehicle manufacturers in 2002. It defines a common interface for telematics applications and includes driving as well as diagnostics information. The data is coded according to the SAE J1939 standard. FMS is a broadcast-only system, a gateway between the internal bus and the FMS bus is implemented by the OEM. third party systems, like the Fleet-Monitor, are forbidden on the internal bus as shown in figure 3.5.

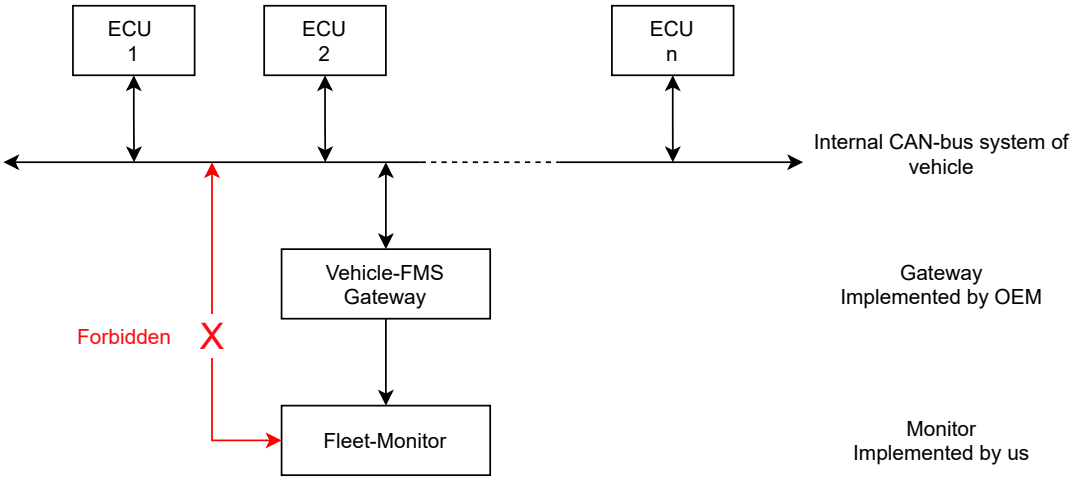


Figure 3.5: FMS Topology

At the time of writing, the standard includes 35 different packages, ranging from engine coolant temperature to door position. Depending on the frame, updates occur between 20 ms and 10 s. The convention set the Parameter Group Number (PGN) for J1939 identifiers, while the OEM specifies the source address and priority. Each frame has a fixed data length of 8 byte but not all of them contain information as shown in the example frame below.[3]

Fuel Consumption: LFC								
PGN Hex	0x00FEE9							
Repetition Rate	1000ms							
Byte Number	Data Byte 1	Data Byte 2	Data Byte 3	Data Byte 4	Data Byte 5	Data Byte 6	Data Byte 7	Data Byte 8
Name	Not Used	Not Used	Not Used	Not Used	Engine Fuel Used	Engine Fuel Used	Engine Fuel Used	Engine Fuel Used

Figure 3.6: Fuel Consumption FMS Frame

4

Development

4.1 Hardware Design

The hardware of the Fleet-Monitor was designed using Altium Designer 21. The integrated 3D CAD functionality simplified the overall development and lowered the possibility of errors in the design. The 4-Layer Printed Circuit Board (PCB) with the size of 140.5 mm x 79.5 mm has been manufactured and assembled by JLCPCB.
As a proof of concept, five prototypes have been made, which are all fully tested and in working condition.

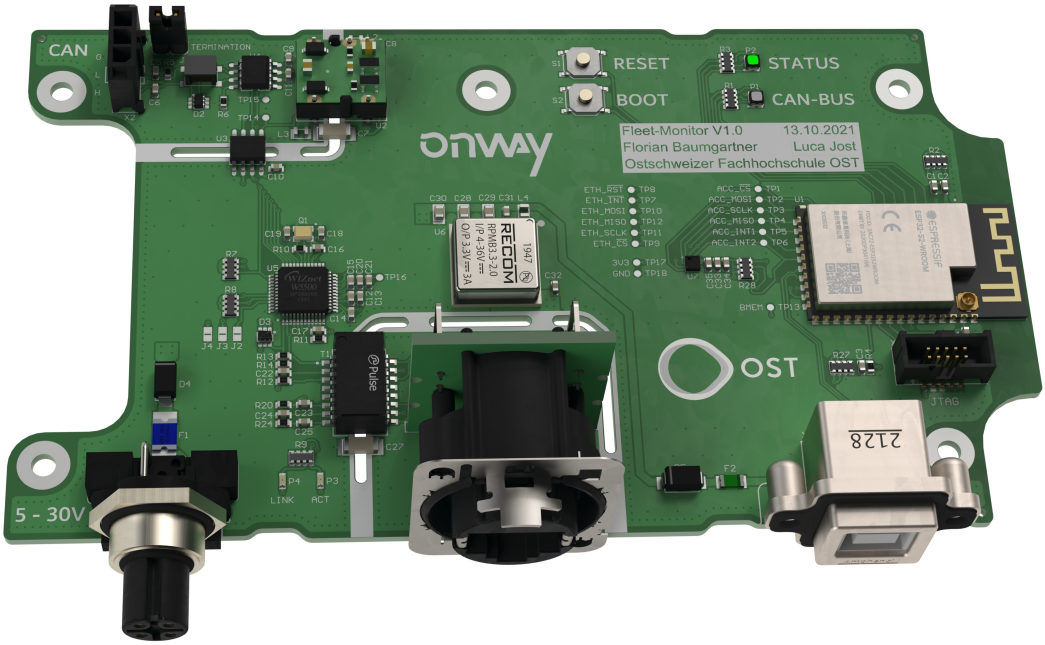


Figure 4.1: Assembled PCB 3D-Render

4.1.1 Block Diagram

The hardware block diagram in Figure 4.2 offers an overview of the system architecture.

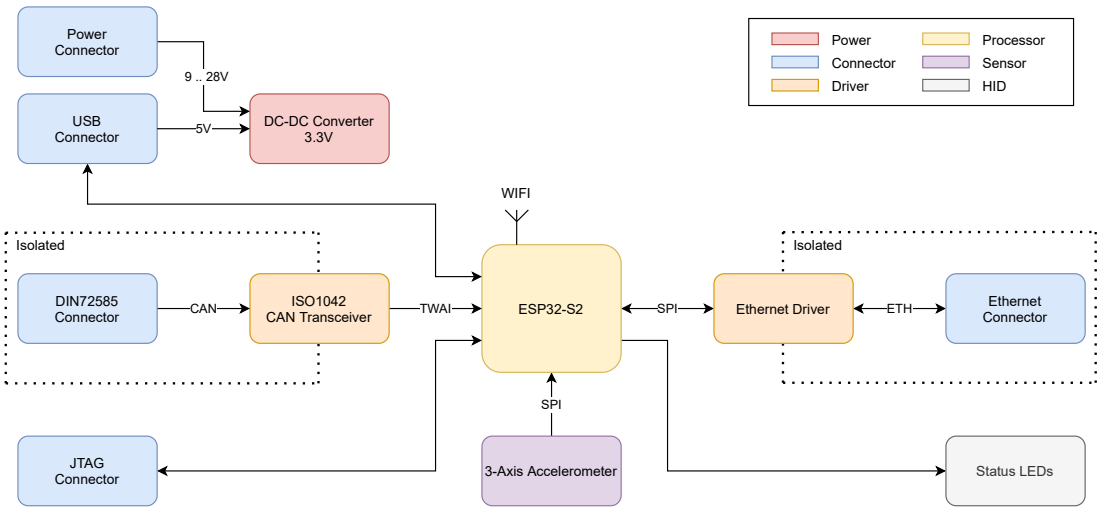


Figure 4.2: Hardware Block Diagram

4.1.2 Power Management

The device can be powered either by the USB-interface or an external DC power source. Both supply options are internally connected through Schottky diodes and provide a seamless switch-over. If both supply sources provide power at the same time, the external DC source is used.
The Universal Serial Bus (USB) Interface fulfills all guidelines of the USB specifications in terms of power consumption. Therefore it is guaranteed, that the maximum current of 500 mA is not exceeded. In case of failure, a resettable PTC Poly-fuse protects the power source from over current.
The external power source accepts a voltage range from 9 V to 28 V and is protected against reverse polarization and short circuit. As a suitable connector, the industry-standard M12 (5 Pin) type has been chosen. It fulfills the IP67 rating and is highly robust against accidental disconnection due to its threaded coupling. In addition, it is used in a wide variety of applications, resulting in great availability. The pin-out consists of pin 1 and 2 used for the positive input and pins 3 to 5 for ground. This has the advantage, that matching connectors with a different amount of pins (e.g. 2-pin connector) can be used as well.

The core of the power management unit is based on a Recom DC/DC converter of the RPMB-2.0 series. The very compact design, great performance and fairly low price offers an optimal solution. The internal system supply voltage is 3.3 V and the total power consumption of the device is on average around 1.5 W.

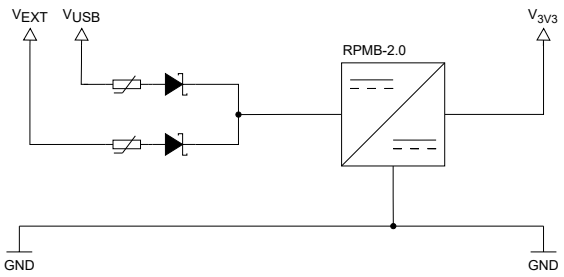


Figure 4.3: Simplified Power Management

4.1.3 ESP32-S2

The choice of a suitable microcontroller is crucial in the design of an embedded system. Several factors were carefully considered and key requirements have been set, such as:

- Integrated WiFi subsystem and RF front-end
- System performance: CPU speed, memory and peripherals
- Native USB 2.0 Interface
- Physical package and pin count
- Availability (especially in an ongoing worldwide chip-shortage)

The Espressif's ESP32 System on a Chip (SoC) family satisfies all listed requirements and is in addition advertised as a low cost solution.
To reduce design complexity and production cost, the ESP32-S2 has been embedded as a solder-on module of the type ESP32-S2-WROOM-I. This module has the advantage of containing the RF front-end inclusive an integrated PCB antenna as well as a 4 MB SPI flash chip.

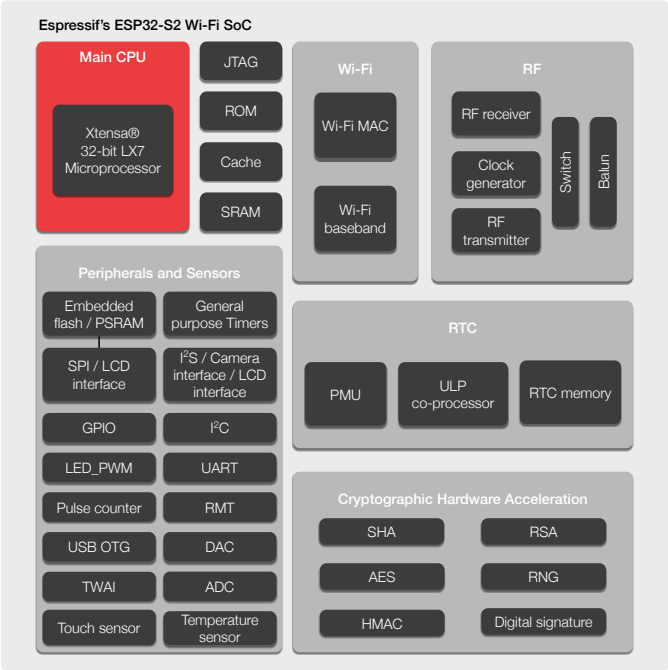


Figure 4.4: ESP32-S2 Block Diagram
Source: ESP32-S2 Datasheet [5]

The SoC can be programmed ether by JTAG and a suitable programmer (e.g. Espressif's ESP32-Prog) or conveniently over the integrated USB-Interface.
For uploading code via the USB-Interface, the ESP32-S2 has to be set into the device firmware update mode. This can be achieved by pressing the boot button while the device is booting (e.g. after powering on or a reset). If the boot button is not accessible, the user can force the device to enter the DFU mode by setting a flag in the system configuration. This procedure gets further explained in the user manual 5.1.2.

4.1.4 Ethernet Interface

In order to connect the device to a host server, an Ethernet interface was added.

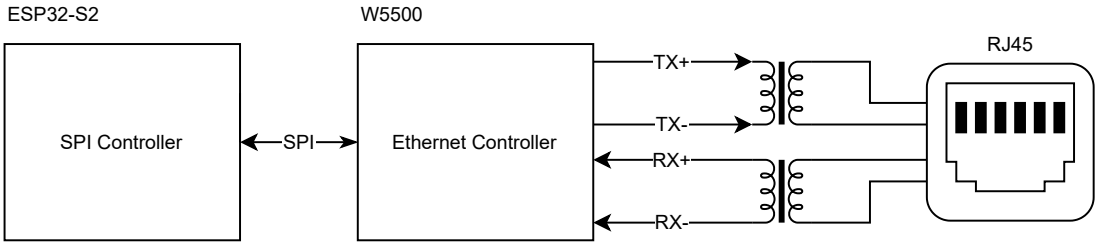


Figure 4.5: Simplified Ethernet Interface

Controller

The Ethernet interface utilizes a WIZNet W5500 Ethernet controller chip with an integrated PHY and TCP/IP stack. It is capable of transmission speeds of 10 or 100 MBit/s. The chip is connected through SPI with the main processing unit (ESP32).

Isolation

The IEEE standard 802.3 specifies a 1500 V_{RMS} isolation barrier between the Ethernet PHY Chip and the cable. To comply with these requirements a pulse transformer was used to magnetically couple the data lines between the connector and the chip. Additionally a ground clearance of 3 mm was chosen to avoid arc-over or tracking between electrical conductors as seen in Figure 4.6.

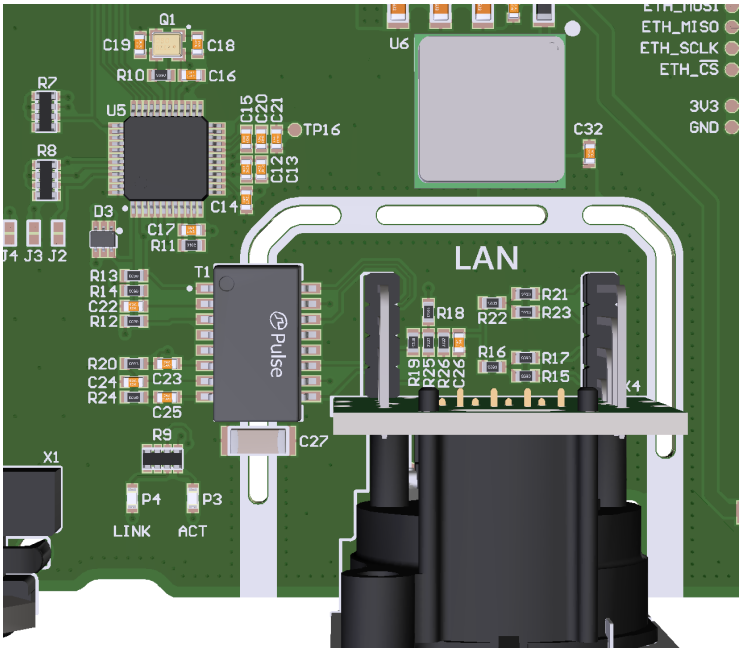


Figure 4.6: PCB view of Ethernet Interface

Connector

The etherCON RJ45 is a lockable connector system, it was chosen for its rugged design and water resistance. This connector system is often used in industrial applications, e.g. in the event industry. The receptacle accepts regular RJ45 plugs as well, but the use of proper etherCON connectors is recommended.

Figure 4.7: etherCON Connector
Source: Neutrik etherCON Connector NE8MX6 [7]

4.1.5 CAN-Bus Interface

The Two Wire Automotive Interface (TWAI) is a real-time serial communication protocol suited for automotive and industrial applications. It is compatible with CAN bus frames following the ISO-11898-1 standard. The ESP32-S2 contains a TWAI controller that can be configured to communicate on a CAN bus via an external transceiver.

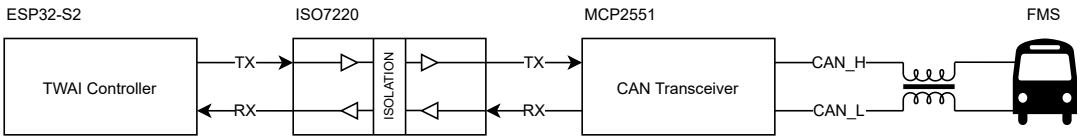


Figure 4.8: Simplified CAN Interface

Transceiver

The data lines are translated using a CAN transceiver from Microchip (MCP2551). The role of the transceiver is to drive and detect data to and from the bus. It converts the single-ended logic used by the controller to the differential signal transmitted over the bus. The MCP2551 device provides transmit and receive capabilities and is fully compatible with the ISO-11898 standard.

Isolation

A dual-channel digital isolator (ISO7221BDR) was used in conjunction with an isolated DC/DC converter from Recom (R1SX-3.305/H). These devices block high voltage and isolate grounds, as well as prevent noise currents on a data bus or other circuits from entering the local ground and interfering with or damaging sensitive circuitry.

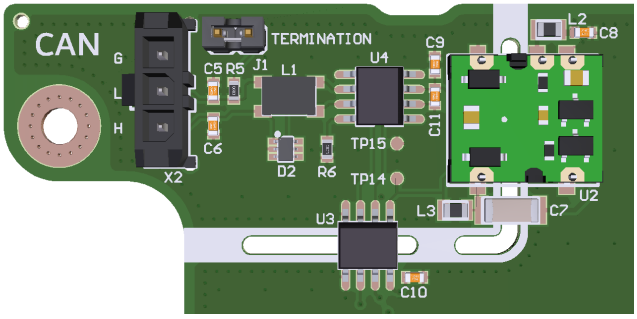


Figure 4.9: PCB view of CAN interface

Filtering

A common mode choke, as well as bypass capacitors, were added to the differential signal lines. A common mode choke is an electrical filter that blocks high-frequency noise common to two or more data or power lines while allowing the desired DC or low-frequency signal to pass. Common-Mode (CM) noise is typically radiated from sources such as radio signals, unshielded electronics, inverters, and motors. Additionally matching capacitors on the CANH and CANL lines were added to enhance the immunity against electromagnetic interference.

Connector

The FMS Standard specifies a DIN 72585 connector as the default physical interface. Since these connectors are only available for panel mount, an additional wire-to-board connector was selected. A Molex Micro-Fit 3.0 was chosen because it is widely available and being used in lots of applications.

Termination

In addition the FMS Standard specifies a 120 Ω CAN termination resistor to be added on the monitor side. To fulfill this requirement and to make it compatible with systems that do not need that resistor, a jumper was added to enable/disable the termination.

4.1.6 Accelerometer

By request of the industry partner, an accelerometer was added to gather additional data. The LIS2DH12 is an ultra-low-power high-performance three-axis linear accelerometer with digital I^2C/SPI serial interface output. The sensor has user-selectable scales of ±2 g up to ±16 g and is capable of measuring accelerations with data rates from 1 Hz to 5.3 kHz. The sensor is connected through SPI to the ESP32.

4.2 Mechanical Design

The automotive environment is known for its harsh conditions, such as vibrations, large temperature fluctuation and high humidity. To ensure long-term reliability, the device must be resistant to these factors. To meet the requirements and achieve an IP67 rating, optimal component selection was critical. Especially the selection of connectors was crucial. An enclosure made of polycarbonate with a transparent lid has been selected as a suitable case for the device. The very robust construction creates an ideal protection for all electronic components.
The case has been machined in the internal workshop of the university. The corresponding mechanical drawings have been made with SolidWorks 2020 and are attached in the Appendix: A.5

Figure 4.10: Final Product 3D-Render

4.3 Firmware

The firmware is written in C++ and is based on a combination of the Arduino and the ESP-IDF framework. As an IDE, Visual Studio Code with PlatformIO as an add-on has been used. This modern environment ensures rapid and effective development.
FreeRTOS has been used as a real-time operating system, which guarantees a reliable operation and handles multi-task operations on single-core systems.
The Arduino framework offers extensive library support, especially for the USB peripheral, file system and Ethernet interface. Thousands of users in the Arduino community make the framework more robust and reliable than other alternatives.

4.3.1 File system

The file system is an essential part of the device firmware. It enables accessing files stored on the internal SPI-Flash chip as well as creating an interface for additional libraries. The flash chip has a memory capacity of 4 MB and is partitioned into four different sections. The size of each partition can be configured in the *partitions_custom.csv* file. The two largest partitions are used for program memory and file system storage. Each of them has ruffly the size of ≈ 2 KB.

Name	Type	Offset	Size
nvs	data	`0x009000`	20K
otadata	data	`0x00E000`	8K
app0	app	`0x010000`	2048K
ffat	data	`0x210000`	1984K

Table 4.1: SPI-Flash Partitions

To provide a sufficient set of features, multiple libraries had to be used. Most of them are created by Adafruit Industries under MIT licensing.
As a low-level SPI-Flash chip driver, the `Adafruit SPIFlash` library has been used. It handles the communication over the physical interface and provides access to the file system library named `SdFAT`. The type of file system is FAT, which has the advantage of being compatible with most modern operating systems (Windows/Linux/Mac OS). Figure 4.11 shows an overview of all utilized core libraries and how they are dependent.

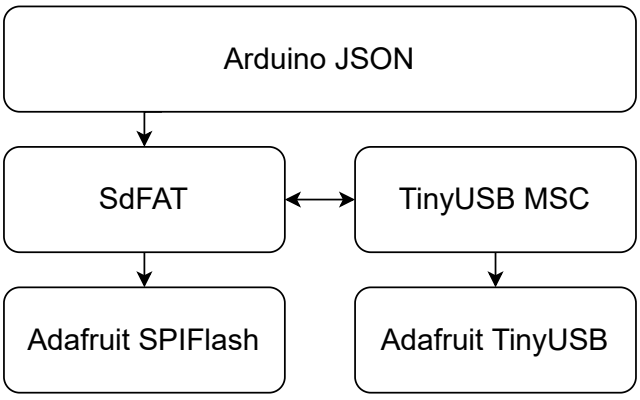


Figure 4.11: USB and File System Library Stack

4.3.2 USB Mass Storage Device

USB is known to be a very complex interface, however it provides lots of comfort to an end-user. To enable USB support on an embedded system, several layers of software are needed. Fortunately, the open source project called TinyUSB supports the ESP32-S2 SoC family. This very comprehensive software stack supports the USB Mass Storage Controller (MSC) protocol. As a result, the device acts just like a regular USB Flash drive and provides seamless file access to any host computer.
To pack all setup and configuration functions of the mentioned libraries together, utility functions were made. The `utils_init()` function checks if the SPI-Flash chip has already been correctly formatted, if this is not the case, an automatic formatting procedure gets executed. In this process, a disk label can be set with a maximal length of 8 characters, in this case: `MONITOR`. The code section below shows how those utility functions are called at the beginning of the program execution.

```
utils_init("MONITOR");  // Initialize peripherals and file system
utils_systemConfig("system.json");   // Load system configuration
utils_startMsc();       // Start USB mass storage controller
```

In addition to the USB MSC protocol, a Communications Device Class (CDC) has been set up. This creates a Virtual COM Port (VCP) over which the host computer can gain debug information from the device.
After initializing the USB-Interface and file system, the device configuration is loaded from the locally stored JSON-File, more details in the section 4.3.4.

4.3.3 JSON Parser Library

Due to the frequent usage of the JSON-File format in this project, using an advanced software library was key to accelerating the development process.
In this case, the open source ArduinoJSON library has been used. It makes use of modern C++14 syntax and is optimized for small architectures like microcontrollers. In addition, support for static and dynamic data allocation simplifies the serializing and deserializing of files without a predefined size. This is especially useful for parsing incoming data over the networking interface.

The following code snippet shows how to iterate over a list of elements. If a matching entry has been found, the corresponding name gets returned as a char string. This example shows how the iterating feature can be applied intuitively and how fields can be manipulated dynamically.

```
for (JsonVariant value : doc["frames"].as<JsonArray>())
{
  if (strncmp(value["pgn"].as<const char*>(), pgnStr, 4) == 0)
  {
    return value["name"].as<const char*>();
  }
}
```

4.3.4 System Configuration

The system configuration is defined in a JSON-File called `system.json`, stored in the root directory of the file system. This file is loaded on every startup. The following parameter can be configured:

Parameter	Type	Description	Example
`ssid`	string	SSID of Access Point (AP)	`"network"`
`password`	string	Password of AP *	`"secret"`
`connection`	string	Preferred connection type: `[auto, lan, wlan]`	`"auto"`
`config`	string	Location of config file: `[local, remote]`	`"remote"`
`host_ip`	string	IP Address of host server	`"10.3.141.1"`
`host_port`	integer	Port of host server	`8080`
`overwrite_file`	boolean	Overwrite locally stored config file with remotely downloaded version	`True`
`bootloader`	boolean	Restart device in DFU mode **	`False`

Table 4.2: System Configuration Parameter Description

* Password field gets cleared after config file has been loaded due to security reasons.
** If set `True`, device reboots immediately in DFU mode and field gets reset to `False`.

4.3.5 Frame Filter Configuration

The configuration of the FMS-Frame filter is based on a specific JSON-File called `config.json`, which is stored in the root directory of the file system. In addition the filter configuration can be downloaded from the host server if the `config` parameter in the system configuration is set to `remote`. This allows the user to adjust parameters while the device is in operation, even without physical access. Basically the JSON-File consists of two types of configuration parameters. First of all the general settings which describe how the data should be uploaded to the server:

`framename` enables or disables the transmission of the FMS-Packet name. Turning off this parameter, reduces the overall data upload size and thus minimizes network traffic. For debugging purpose, enabling this setting can help identifying FMS-Frames.

`unknownframes` enables or disables the transmission of unknown FMS-Packets, meaning frames that are not listed in the configuration settings.

The second part of the configuration file contains a look-up table with FMS-Frame information and filter settings in form of a list. The different filter types are further described in section 4.3.7. Each entry consists of multiple fields specified as followed:

Parameter	Type	Description
`pgn`	ASCII-HEX	Parameter Group Number (PGN) as 4 digit number
`name`	string	Human friendly frame name
`active`	boolean	Transmission state, `False` means ignore frame type
`filter`	string	Filter type: `[nofilter, change, interval]`
`time`	integer	Max. interval time in [ms], field exists only if filter type is set to `interval`

Table 4.3: Frame Filter Parameter Description

The following example shows how the JSON-File is structured. For better visibility some of the packet settings have been hidden.

```
▽ JSON {3}
  □ framename:     True
  □ unknownframes: False
  ▽ frames [34]
    ▽ 0 {4}
      □ pgn:    FEE9
      □ name:   Fuel Consumption: LFC
      □ active: True
      □ filter: nofilter
    ▽ 1 {4}
      □ pgn:    FE6C
      □ name:   Tachograph: TCO1
      □ active: True
      □ filter: interval
      □ time:   100
    ▽ 2 {4}
      □ pgn:    FE4E
      □ name:   Door Control 1: DC1
      □ active: True
      □ filter: change
    ▽ 3 {4}
      □ pgn:    FDA5
      □ name:   Door Control 2: DC2
      □ active: False
      □ filter: nofilter
    ▽ 4 {4}
      □ pgn:    FEEA
      □ name:   Vehicle Weight: VW
      □ active: True
      □ filter: interval
      □ time:   5000
    ▷ 5 {4}
    ▷ 6 {4}
    ...
    ▷ 34 {4}
```

Table 4.4: Frame Filter Configuration Example

4.3.6 Networking

The Fleet-Monitor needs to communicate with a server to stream data, exchange the current time and read configuration data.

Connection

The device can connect through WiFi or Ethernet with the host. Both interfaces utilize DHCP to negotiate an IP address and the server IP can be configured through the system configuration. The device can be connected to both interfaces at the same time but only one is used to communicate with the server.

The connection is checked every 5 seconds and works as follows. First the hardware status is checked, then the connection to a host and lastly if we received an IP address. The Ethernet interface is the prioritized communication method. It provides better data throughput and is less prone to errors than the over the air alternative. For example, if both interfaces received an IP address and are able to communicate with the server, Ethernet is used for the transmission. This is illustrated in Figure 4.12 bellow.

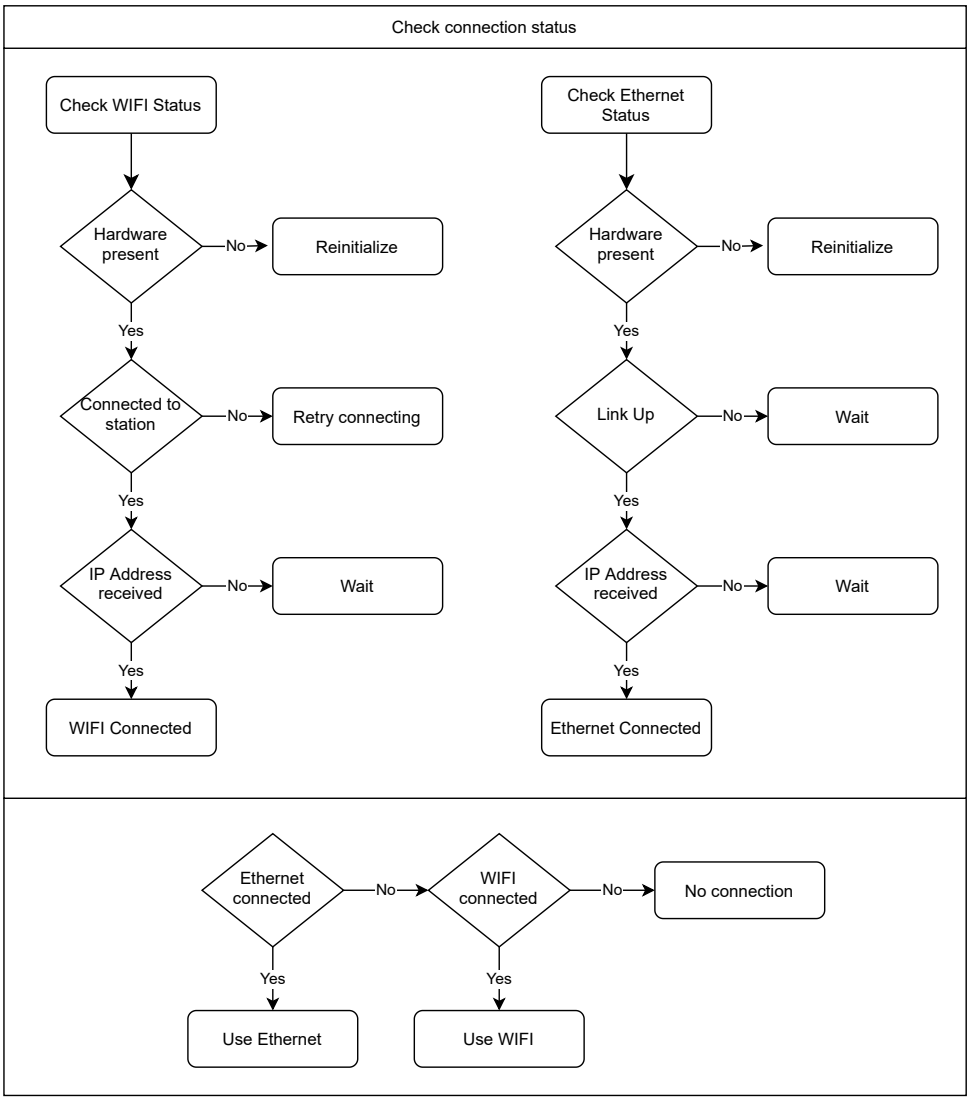


Figure 4.12: Flow chart of interface selection

HTTP

Hypertext Transfer Protocol (HTTP) is used to communicate with the server and allows the Fleet-Monitor to read files and stream data. We decided to go with this technology because it is well supported by all libraries used. Data is transferred using HTTP POST requests containing data inside the body. The server response is used to check if the data was received correctly and validate the server connection. HTTP GET methods are used to read files from the server.

Time synchronization

In order for data to be useful it needs to have a timestamp. Ideally this timestamp would translate to a real time and date. This is why we implemented on both the server and client side a mechanism to synchronize the two. Every POST response from the server includes a JSON string with a field called Date. This field represents the current date and time and is used by the Fleet-Monitor to set its real time clock. Since the ESP32-S2 WROOM module does not have a dedicated oscillator for time keeping, the default 40 MHz crystal is used. This oscillator has a tolerance of ± 10 ppm. The maximum 24 hour drift can be calculated using the Equation 4.1.

$$\delta_{rel} = \frac{1}{f} * \frac{f * p}{10^6} * t = \frac{1}{40\,MHz} * \frac{40\,MHz * \pm 10}{10^6} * 86400\,s = \pm 0.864\,s \tag{4.1}$$

where:

f = frequency in Hz
p = deviation in ppm
t = time in s

Since this drift is quite large, we decided to update the real time clock of the ESP32-S2 every 24 h to make sure both the server and the Fleet-Monitor always share the same time.

The timestamp is read out as `YYYY,MM,DD,hh,mm,ss` format and is stored locally in Unix time.

File Reload

Additionally a mechanism was developed to automatically detect if a configuration file needs to be reloaded. A reload is needed when the file on the server was modified. Every HTTP POST response contains a JSON field called `ConfigReload`. If this flag is set we will request the configuration file again from the server in order to clear the flag and get the newest configuration. The local configuration can be overwritten with the one from the server, if the option is selected in the system configuration.

4.3.7 FMS Frames

Reading out frames from the CAN bus, filtering them and finally transmitting them to the server is done using two FreeRTOS tasks and a FreeRTOS queue. Figure 4.13 illustrates this task view. The frame handler task has the highest priority as task switching while transmitting can cause issues.

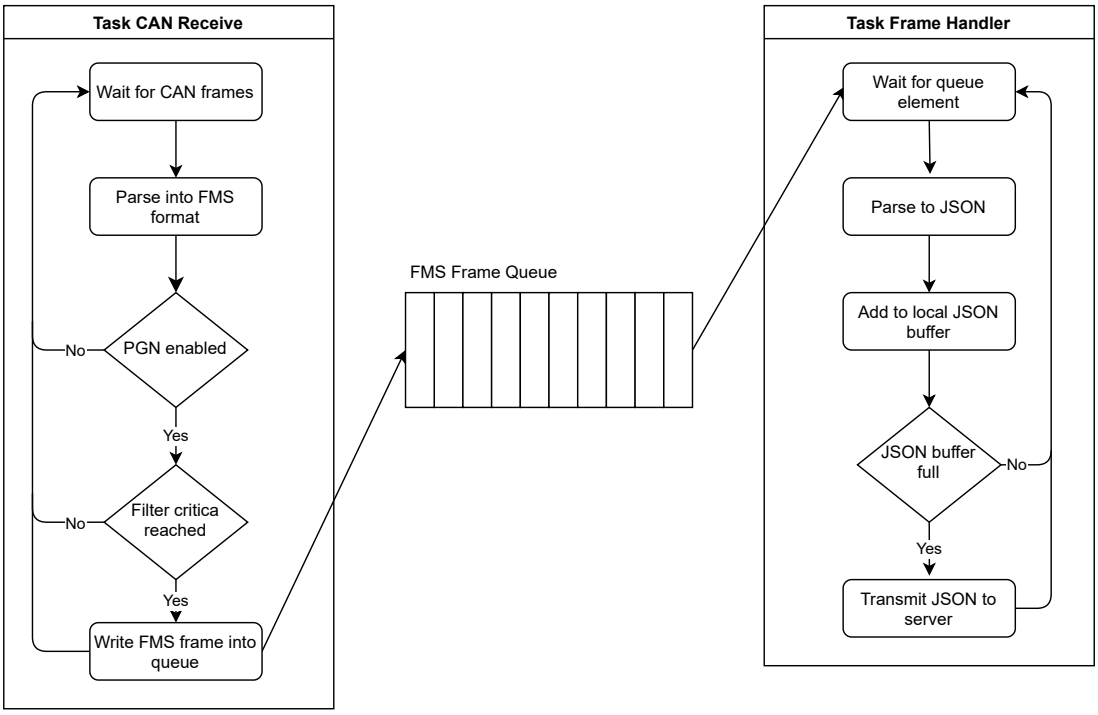


Figure 4.13: FreeRTOS Tasks and Queue for FMS Frames

FMS Filter

Both tasks will only begin its execution once a configuration was read from either the server or from the local file system. The configuration is explained in the Section 4.3.5.

Generally a frame can be enabled and disabled for the transmission, this helps remove unwanted packages from taking up valuable bandwidth. Additionally three filter methods were implemented to reduce the data rate and repetition of useless information:

- *On Change*, data is only transmitted when something inside the data field of the frame has changed. This is very useful for state supervision of error codes or door states.
- *Interval*, data is only transmitted with a set interval. Data in between the set interval is discarded. This filter method is used for things that are not very important but still good to know from time to time.
- *No Filter*, all the data received is being transmitted. This method is used for things that need real-time supervision.

Data inside the queue is always sent out to the server. Therefore the filtering is done inside the CAN-Read task and only adds data to the queue when it passes all the filters.

JSON Format

In order to hold the FMS data from the queue, a JSON buffer is allocated. The timestamp is added to each frame, and then all the other fields are added to the JSON array. The user can enable and disable the name in the configuration file. Adding the name simplifies parsing, but increases the data rate since it is added to every frame.

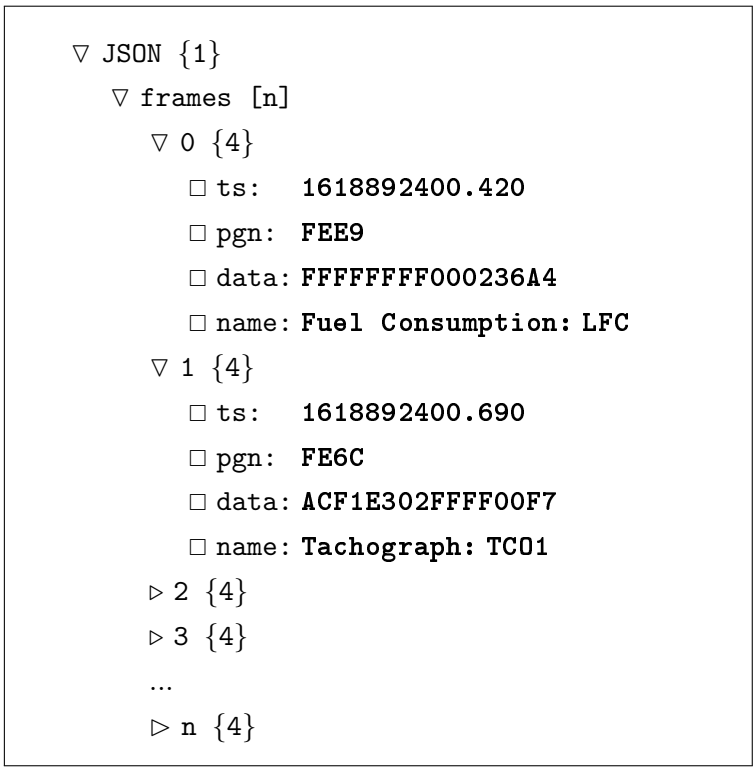


Table 4.5: Example FMS Data for Transmission

The JSON buffer can hold a maximum of 8 KB. Single frames are added to the buffer until it is filled up. At that point it is ready for the transmission.

Transmission

Data is communicated over the currently active networking interface, as previously explained in Section 4.3.6. The JSON buffer is serialized into a single string before being sent to the server as part of an HTTP POST request. In normal operation, the server confirms the request with a status code of `200`. The JSON buffer is discarded if a request to the server times out or if no confirmation is returned. This is due to the large amount of data collected by the bus. The length of the transmission body varies, however it is usually around 8000 bytes. The period between POST requests is significantly depending on the filter technique, but without any active filters, a transmission occurs around every second.

4.3.8 Accelerometer

The acceleration is read out from its own task with an update rate of 50 Hz. Since we are not using the data at this moment, we just read it out and leave its specific implementation open for the future.

On boot up over a period of 5 seconds, the accelerometer is calibrated to eliminate the offset on each axis and get rid of the gravity vector. The purpose of this is to make it easy to record high acceleration events. When the linear acceleration reaches a predefined threshold, the data can be transmitted to the server.

The axes of the accelerometer are oriented as shown in Figure 4.14.

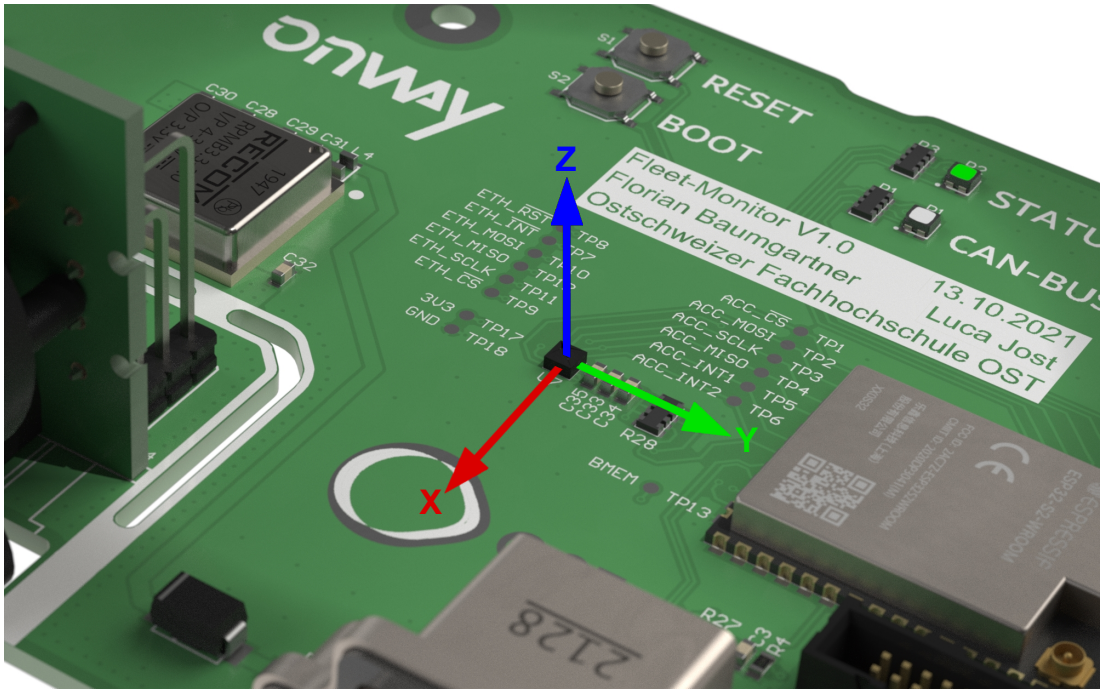


Figure 4.14: Accelerometer Axes Orientation

4.4 Utility Software Tools

Several software tools have been developed to support the hardware in its operation and to simplify the configuration of the device. These utilities were all written in the Python programming language. As a suitable platform to run the applications, a Raspberry Pi 400 was selected for its price and performance.

4.4.1 HTTP Server

A HTTP server was developed to communicate with the Fleet-Monitor. It was written with the help of an AI system that translates natural language to code called OpenAI Codex. It helped us to reduce the development time and produced comprehensive results for our application. The HTTP Server is a command line application that does not have a graphical user interface. Both HTTP GET and HTTP POST requests can be handled by the server. Each request to the server creates a new thread that will handle the response. Making it possible to handle multiple requests at the same time.

HTTP GET requests are used to access files stored locally. Filenames are included in the URL of the HTTP GET request. Using this method, multiple files may be served. If the requested file is located in the servers directory, the server responds with status code `200` and serves it. Otherwise, an empty page is returned with a status code of `404`.

Streaming data from the Fleet-Monitor to the server is extremely important for our application. To accomplish this we are using HTTP POST requests. The server expects a JSON string of FMS data inside the body of the request, if the data is found and no errors are detected it will respond with a status code of `200`. Moreover, the server will include the current time and date in the response, along with the information if a file has been changed in its directory. The JSON string from the request is split and then stored locally in a CSV file. By using the FMS Data Visualizer, the data can then be interpreted. Example CSV data:

```
0, 1639753080.803,F003,FFD9FFFFFFFFFFF
1, 1639753080.814,FE6C,485352203C330000
2, 1639753080.846,F004,FFFFFF8039FFFFFF
3, 1639753080.857,F003,FFD9FFFFFFFFFFF
4, 1639753080.868,FE6C,5768792061726520
5, 1639753080.911,FE6C,796F75206465636F
6, 1639753080.921,FE6C,64696E6720746869
7, 1639753080.954,FE6C,733F210000000000
8, 1639753080.964,F003,FFD9FFFFFFFFFFF
9, 1639753080.975,FE6C,4F53545355434B53
10, 1639753081.18,F003,FFD9FFFFFFFFFFF
11, 1639753081.29,F004,FFFFFF8039FFFFFF
...
```

Both the HTTP server and a utility tool called *RaspAP* are running on a Raspberry Pi. *RaspAP* is used to create an Access Point and configures the networking ports to act as a router, making it possible to connect to it over WiFi and Ethernet.

4.4.2 FMS Configuration Tool

Changing the configuration of the FMS-Frame filter in a text editor is not very pleasant. Therefore a Graphical User Interface (GUI) has been developed to make this procedure easier and ensure a better overview of all settings. This tool is able to generate the JSON-File structure as mentioned in Section 4.3.5.

Feature Description

The configuration tool is divided into *Global Settings* and *Frame Settings*. The general options can be enabled or disabled by clicking on the corresponding checkbox. Further in the Frame Settings section, a table allows the user to change the filter behaviour of each FMS-Packet type. The filter features are described as followed:

Active: Enables or disables the transmission of this specific packet type

PGN: Shows the Parameter Group Number in hexadecimal format *(Read-Only)*

Name: Shows the user friendly packet name *(Read-Only)*

Filter: Combo-box to select the filter type: *No Filter*, *On Change*, *Max. Interval*

Interval [ms]: Max. update rate, only available if filter type is set to: *Max. Interval*

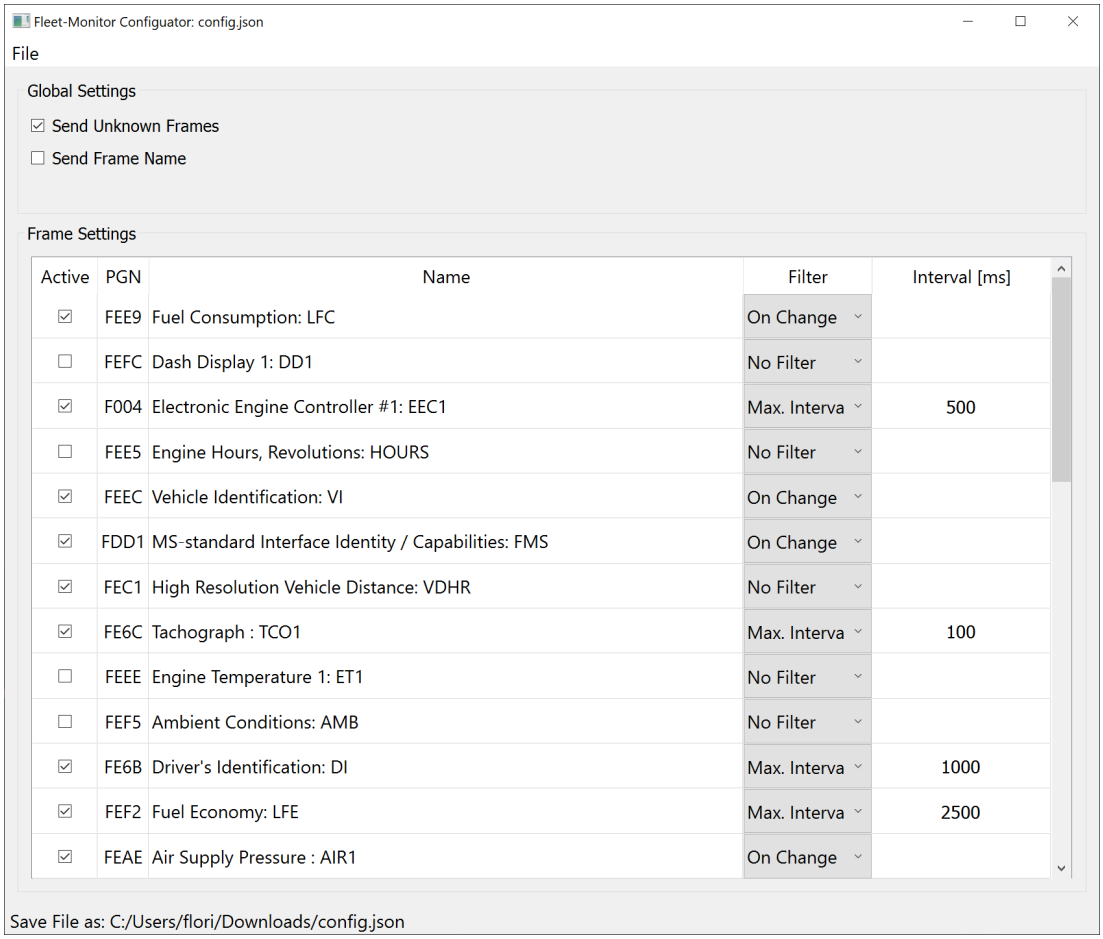


Figure 4.15: FMS Frame Configuration Tool

PyQt5

As a framework Qt Version 5 for Python, in short PyQt5, was used. This very comprehensive software package enables rapid development of GUIs. One of several major advantages is the platform compatibility. Therefore the tools run on all major operating systems without any changes to the code. The python interpreter executes the code in a closed environment and does only depend on the PyQt5 framework. No additional packages or frameworks are needed.
The dynamic scaling of the window including all of its features is essential. Modern applications must be able to run on a variety of screen resolutions.

File Loading & Saving

To open an existing configuration file, three methods are available. Firstly via the menu-bar section, by clicking on the *Open* button. Secondly by using the keyboard shortcut *Ctrl+O* and lastly by dragging a file into the application. Saving files is very similar and intuitive due to the use of common shortcuts.

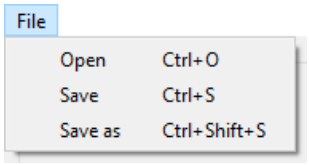


Figure 4.16: File Menu-Bar

4.4.3 FMS Data Visualizer

Since the host server receives the data in raw HEX format, post processing is needed to gain access to the contained information. To demonstrate this procedure, a Python script has been developed to parse and visualize FMS-Data.

FMS Frame Parsing

The FMS-Standard describes in detail how each frame type can be decoded. In the scope of this project, the focus has been set on parsing the most common frame types. As mentioned in Section 3.3, a frame contains 8 bytes of data. Figure 4.17 shows an example of how the ambient air temperature is encoded. In this case, only 2 bytes (Data Byte 4 & 5) are used.

Ambient Conditions: AMB								
PGN Hex	0x00FEF5							
Repetition Rate	1000 ms							
Byte Number	Data Byte 1	Data Byte 2	Data Byte 3	Data Byte 4	Data Byte 5	Data Byte 6	Data Byte 7	Data Byte 8
Name	Not Used	Not Used	Not Used	Ambient Air Temperature **0.03125 °C/Bit - 273 °C Offset**	Ambient Air Temperature **0.03125 °C/Bit - 273 °C Offset**	Not Used	Not Used	Not Used

Figure 4.17: FMS Frame Example: Ambient Conditions: AMB [3]

The Python code snippet below shows how to extract the ambient air temperature of the raw input data. The bytes are ordered as *little-endian*. The parser concatenates the individual bytes and applies global scaling and offset. In this case a scaling of 0.03125 °C / Bit and an offset of -273 °C gets applied.

```
amb = df[df["pgn"] == 0xFEF5]  # Filter data-set by frame type
amb["ambientAir"] = (amb["3"] + amb["4"] * 256) * 0.03125 - 273.0
```

List of Supported Frames

The current version of the FMS Data Visualizer supports parsing the following frame types listed in Table 4.6. Note that this tool is made for demonstration purposes only.

PGN	Frame Name
FE6C	Tachograph: TCO1
FD09	High Resolution Fuel Consumption (Liquid): HRLFC
FEF2	Fuel Economy: LFE
FDA5	Door Control 2: DC2
FE58	Air Suspension Control 4: ASC4
FEEE	Engine Temperature 1: ET1
FE56	Aftertreatment 1 Diesel Exhaust Fluid Tank 1 Information: AT1T1I
FEF5	Ambient Conditions: AMB
FED5	Alternator Speed: AS
FEC1	High Resolution Vehicle Distance: VDHR
FEAE	Air Supply Pressure: AIR1
FEF1	Cruise Control / Vehicle Speed 1: CCVS1
F004	Electronic Engine Controller #1: EEC1
F003	Electronic Engine Controller #2: EEC2

Table 4.6: Supported FMS-Frames by the Python Data-Parser

Plotly Library

As a visualizing framework *Plotly* was used. The framework enables creating modern looking interactive diagrams. A great advantage is the web-based interface which allows updating the plot in real time. The Figure shows an example of visualizing a FMS-Data dump from a public transport bus in Germany.

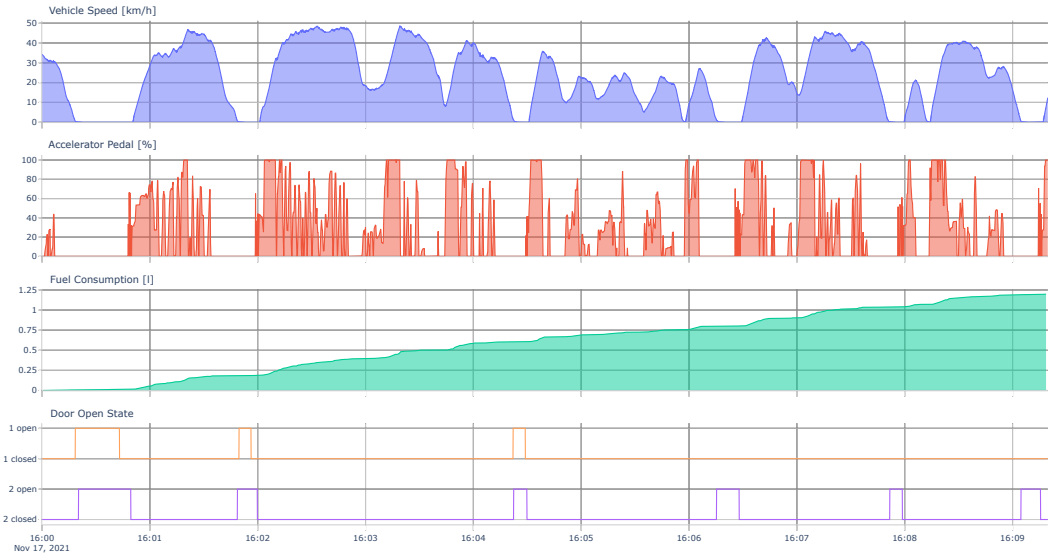


Figure 4.18: Visualization of FMS Data

5

User Manual

This chapter describes the function and operation of the device, the system specifications and possible error scenarios with a potential cause of the problem. In addition, instructions are available for setting up the device and change the configuration.

5.1 Configuration

5.1.1 System Setup

After production, the blank ESP32-S2 must be programmed with the latest firmware. This can be achieved by following the steps in Section 5.1.2. Next, the system configuration as well as the FMS-frame filter configuration must be loaded onto the root directory of the mass storage device. Figure 5.1 shows how the file structure should look like when plugged in to a PC via the USB-interface.

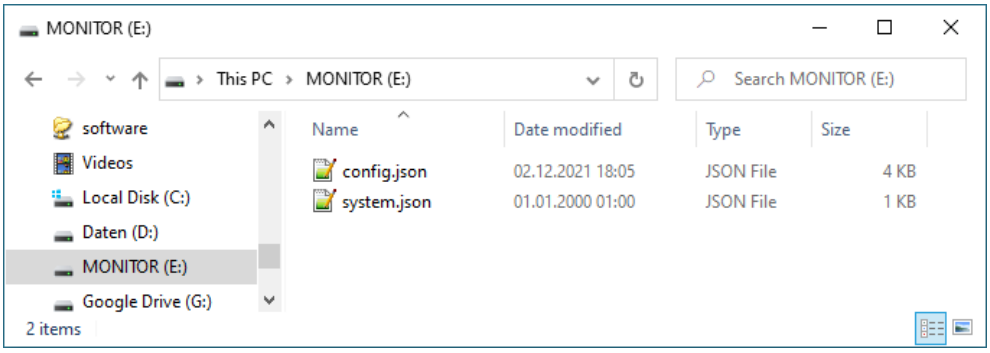


Figure 5.1: Fleet-Monitor Root Directory

Furthermore, parameters in the system configuration file must be adjusted to the needs of the overall setup. This contains setting the host IP address and port number, as well as the SSID and password if the device should connect over WiFi. Additional options can be defined, please refer to Section 4.3.4 for more information.

5.1.2 Firmware Update

Espressif offers a firmware download tool for the ESP32-Family [6]. When starting the application, the options in the dialog box should be set as shown in Figure 5.2. After pressing OK, the main window appears. To establish a connection to the ESP32-S2 over the USB-interface, it has to be set into DFU-Mode. This can be achieved by pressing the *boot* button while resetting or power-cycling the device. If the device is already programmed with a firmware version that supports entering the DFU mode by software, the `bootloader` flag can be set to `true` in the system configuration file as mentioned in Section 4.3.4.

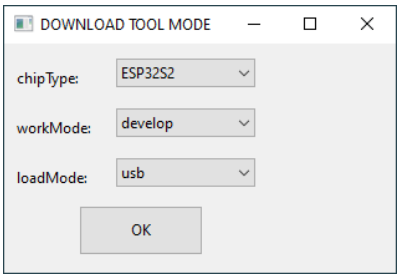


Figure 5.2: ESP32 Download Tool

The next step is to load the binary files at a specific location (offset as HEX index) in the flash memory. The first three files are called `bootloader_qio_40m.bin`, `partitions.bin` and `boot_app0.bin`. These files are independent of the firmware and must not be changed. The fourth file called `firmware.bin` can be loaded directly from the build folder of the IDE (e.g. Visual Studio Code) and placed at offset `0x10000`. In this case the following path is used: `FleetMonitor\.pio\build\esp32s2dev`

Change the programming options of the download tool so that they exactly match Figure 5.3. Now select the correct COM-Port and press the `START` button. It takes around 10 seconds to upload the firmware. After updating, the device has to be reset and the new firmware gets executed. Note that this procedure does not touch the file system and therefore no data is being lost.

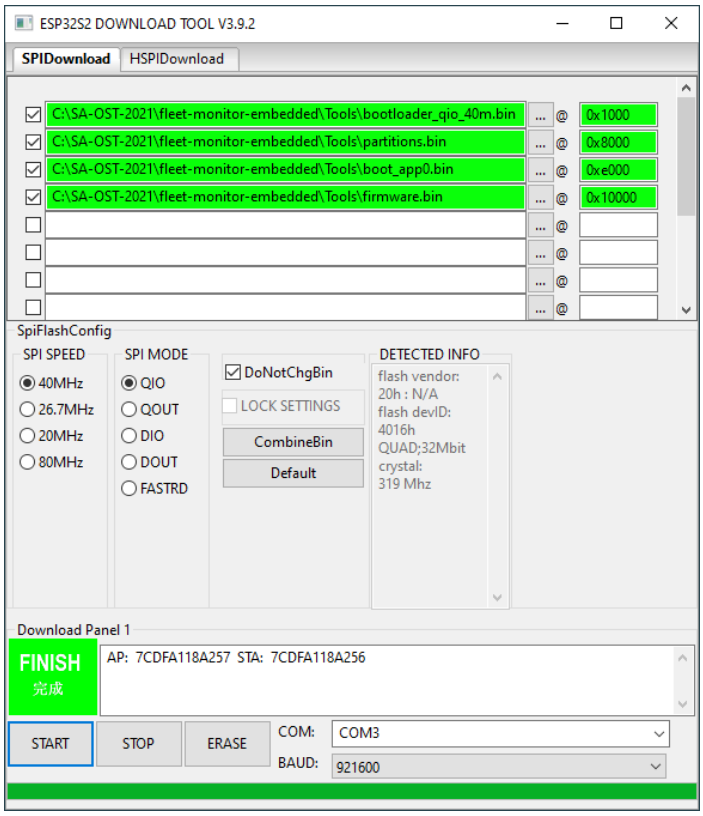


Figure 5.3: ESP32S2 Download Tool Settings

5.2 Specifications

External Input Voltage	9 V - 28 V
Power Consumption	2.5 W
Dimensions (L x W x D)	160 x 89 x 61 mm
Weight	350 g
Water Resistance Rating	IP67
Power Connector	M12 R/A F
Microprocessor	ESP32-S2
LAN-Interface	10 Base-T / 100 Base-TX
WLAN-Interface	802,11b/g/n
USB-Interface	USB 2.0 (Device)
CAN-Interface	250 kbit/s (SAE J1939)

5.3 Device Status and Troubleshooting

5.3.1 LED Status

The Fleet-Monitor uses two RGB LEDs to indicate the current status. Additionally there are two LEDs located to the left of the RJ45 connector showing the link status and link activity.

Status LED	Description
White breathing	System is booting up
Blue breathing	System tries to connect to network
Green	Connected to server over Ethernet
Yellow	Connected to server over WiFi
Magenta	Unable to establish server connection
Red blinking	Unable to load configuration file

CAN LED	Description
Off	Device is not receiving CAN data
Green blinking	Device is receiving CAN data
Red blinking	Error on physical interface

5.3.2 Resolving Issues

Status LED breathing white

Normally this status clears within one or two breathing cycles. If the status does not clear the device is most likely in a boot loop. If this happens something is seriously wrong with the hard- or software. Refer to Section 5.3.4 for a resolution.

Unable to connect to network

The blue LED will breathe while the device is trying to establish a network connection. Connecting over WiFi or Ethernet can take up to one minute. The status is cleared as soon as a connection is received and an IP address was assigned. If no connection can be established over an extended period of time the following things should be checked in order.

Connecting over WiFi:

1. Make sure the device is not obstructed and close to the Access Point in order to establish a connection. Metals, carbon fiber, and other RF shielding materials should be kept away from the antenna of the ESP32.
2. Make sure the Access Point is password protected with WPA or WPA2. Both WEP and WPA3 are not supported.
3. Make sure the Access Point name and password are set correctly in the system configuration. Please limit your passwords to ASCII characters, special Unicode characters are not allowed. Remember, the system automatically clears the password from the `system.json` file after a reboot but stores it locally.
4. Make sure other devices like phones or computers can reach the Access Point. If they are unable to, you may have a problem on the AP side.

Connecting over Ethernet:

1. Make sure the LEDs on the left side of the RJ45 connector are showing activity. If that is not the case check the following things.
   - Make sure the Access Point is powered on.
   - Make sure the RJ45 cable is plugged in completely on both sides.
   - Make sure the RJ45 cable is not faulty.
   - Try connecting another device to the RJ45 cable and see if it receives a connection.

   If all of these checks pass you may have a soft- or hardware issue. Refer to Section 5.3.4 for a resolution.
2. Make sure the Access Point is capable of DHCP. The Fleet-Monitor gets the IP address through the Dynamic Host Configuration Protocol. There is no support for static IP addresses.

Unable to establish server connection

If the status LED is not cleared after a short while, the Fleet-Monitor is unable to establish a connection to the server. The device is connected to an AP, received an IP address but the HTTP POST requests are unanswered or a bad status is returned. The following things should be checked to resolve this issue.

1. Make sure the HTTP server is running and can respond to POST requests. The Fleet-Monitor is expecting to receive a response from the server with the status code of `200`. Otherwise, it will not clear the error.
2. Make sure the IP address and port are configured correctly in the system configuration file.
3. Make sure to check if a request times out or an error code is returned from the server. The Fleet-Monitor will print the server response to the serial port. Section 5.3.3 explains how to access the serial port.

Unable to load configuration file

If the status LED is blinking red the device was unable to load the configuration file.

A missing or corrupt file system is possible if the configuration can not be loaded locally.

To force manual reformatting of the file system, the *boot* button must be pressed for 5 seconds while the device is in operation (not during a reset of the device).
ATTENTION: THIS PROCEDURE WILL DELETE ALL FILES!

In case the system is configured to load the file from the server you may need to refer to the error: Unable to establish server connection 5.3.2. The only difference is that these requests are done with HTTP GET instead of POST. Make sure your server can handle the requests to the Fleet-Monitor's specification.

Device is not receiving CAN data

In case the CAN LED is turned off, the device is unable to receive data from the CAN bus. You may want to try the following things.

1. Make sure the CAN termination is enabled on both ends of the bus.
2. Make sure the CAN data polarity is correct (CANH / CANL).
3. Make sure the cable has the correct pin-out and no wires are damaged.
4. Make sure the FMS Gateway from the vehicle is transmitting data.

CAN Hardware Error

In case the CAN LED is blinking red, something went wrong in the initialization process of the CAN interface. You may need to do a hard reset explained in Section 5.3.4.

5.3.3 Serial Monitor

Diagnostics information is printed to the serial monitor to help the user debug an issue. Accessing the terminal can be done in just a few steps.

1. Get a serial terminal tool for your computer. If you are running Windows, *TeraTerm* is highly recommended.
2. Connect the Fleet-Monitor to your computer with a USB cable.
3. Locate the assigned serial port (e.g. using the *Device Manager* on Windows).
4. Run your serial terminal tool and connect to the open port. In *TeraTerm*, under *File* → *New Connection* the serial port can be selected as shown in Figure 5.4. Your port may vary.

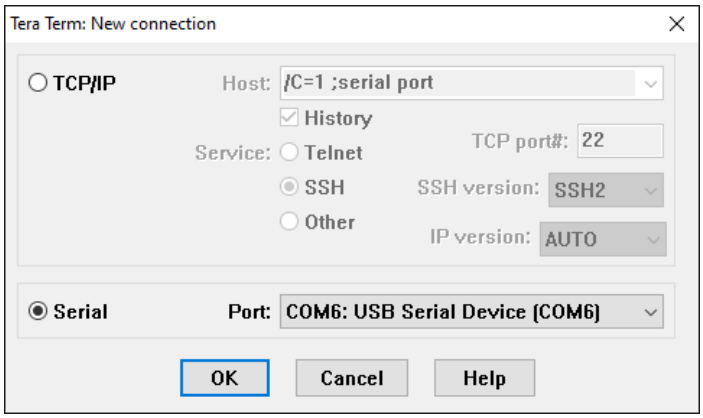


Figure 5.4: Open serial port through *TeraTerm*

5. Information is now being printed to the terminal and should look something like the example below.

```
Connected over WiFi
Loading config from server
Status code: 200
Config loading successful!
```

5.3.4 Hard Reset

If there is a serious issue with the device, the first action should be to reformat the file system and then upload the configuration files again. If this did not resolve the problem you may want to flash a known to be working firmware to the device. This is explained in Section 5.1.2. If the issue persists, some parts of the hardware could be damaged and need to be replaced.

6

Summary & Conclusion

To summarize, a fully comprehensive device has been developed. With it, additional software tools were created to support the hardware in its operation and to simplify the configuration for the user. The FMS-Monitor enables onway customers to get access to low-level vehicle data in a very convenient manner. In order to prove the functionality, five reliable and production-ready prototypes have been produced. The devices were tested over an extended period of time with a J1939 simulator. A test report was written and can be found in the Appendix A.6.

All requirements defined in the task definition have been satisfied. The only pending aspect is the transmission of accelerometer data. The implementation of this feature has been neglected, due to the insufficiently detailed description of the operation. However, this functionality can easily be added in the future if necessary.

6.1 Continuing Work

Although the designed prototypes provide a rich set of features, there are some aspects to improve or features to add. Of particular note is the fact that the system is designed for future enhancements. As an example, the flash storage of the ESP32-S2 provides lots of additional storage for larger firmware editions.
The following continuations are possible:

- Testing the device in the field (e.g. installed in a bus). The test results can show the impact of harsh weather conditions, vibrations and other environmental influences.
- Adding Over-the-Air (OTA) firmware update support to the device. This feature could provide quick bug-fixes and simplifying the addition of custom features. It would accelerate the updating process especially for larger fleets.
- Porting the Python based HTTP server application to dedicated onway hardware (network router).
- Cost reduction of the hardware. This can be achieved by different assembly options (e.g. with/without physical Ethernet interface).
- Implementation of accelerometer data transmission.

6.2 Reflection & Project Schedule

Our time management was excellent, all milestones were reached as scheduled and all functions were implemented on time. There were some parts of the firmware that turned out to be more complicated than expected, resulting in long workdays in order to stay on schedule. It proved to be a very substantial decision, to switch from the ESP-IDF framework to Arduino based core libraries. This has helped drastically to get the USB-Interface and file system working. The ESP-IDF lacks software support in these areas. Further, assembly and testing of the hardware took almost three times longer than we originally planned. This was caused by a small oversight in the schematic of the device, which led to a bad connection. After having rectified the problem in the Ethernet controller circuitry, everything else went smoothly.

6.3 Personal Reflections

Florian Baumgartner

This student research project allowed me to get involved to the CAN interface and the FMS protocol. Further I could make use of my previously gained knowledge in the field of embedded systems and software engineering. It was a very positive experience to develop a fully working product in such a small time frame. The detailed planing of the project proved to be very important. Thus, I'm especially proud of meeting each milestone on time and preventing major delays.
It was a pleasure to work with Luca Jost and we had overall a great time working on this project. My personal highlight was learning the PyQt5 framework as well as using the Plotly library. I'm fairly interested in the field of IoT devices, therefore this project was a great opportunity to deepen my knowledge and getting more experienced.

I'm glad that the hardware development went that smoothly although the ongoing worldwide chip shortage made it difficult to get access to the components needed. It payed out to premature focus on this particularly challenging situation and chose parts that were easily available. All in all, good communication was key to lead to a successful result.

Luca Jost

In general, this student research project overall has been very enjoyable. Within just a few weeks, we transformed an idea into a deployment-ready product. The Fleet-Monitor is working as designed and I am looking forward to seeing the device being used in the field. During the project, I was able to leverage my previous experience and build on it. I found it particularly fascinating to learn more about CAN systems, as it is a very popular technology in the industry currently. Working with embedded systems is something I have always liked and this project was no exception. Developing real-time operating systems with complex logic is something I enjoy. This project was not particularly complex, and unfortunately, there was nothing that challenged me significantly.

In past projects, I had trouble with time management, so we made sure to develop a realistic timetable this time. Spending the extra time on the schedule has proven to be very advantageous as I was able to deliver all tasks on time.

The experience of working with Florian Baumgarnter was extremely rewarding; his dedication and attention to detail are admirable.

A

Appendix

A.1 Declaration of Authorship

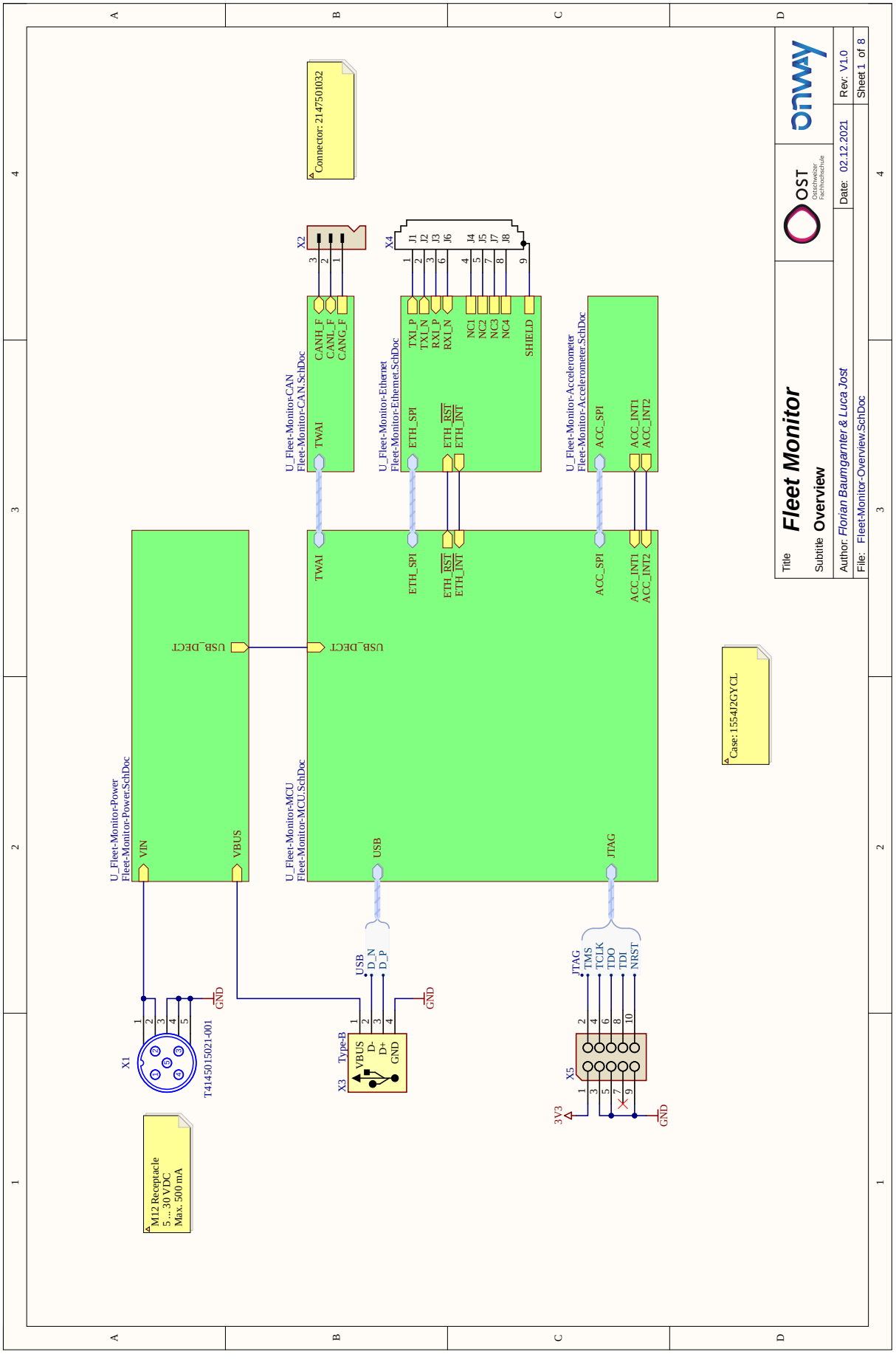
We hereby certify that the thesis we are submitting is entirely our own original work except where otherwise indicated. We are aware of the University's regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

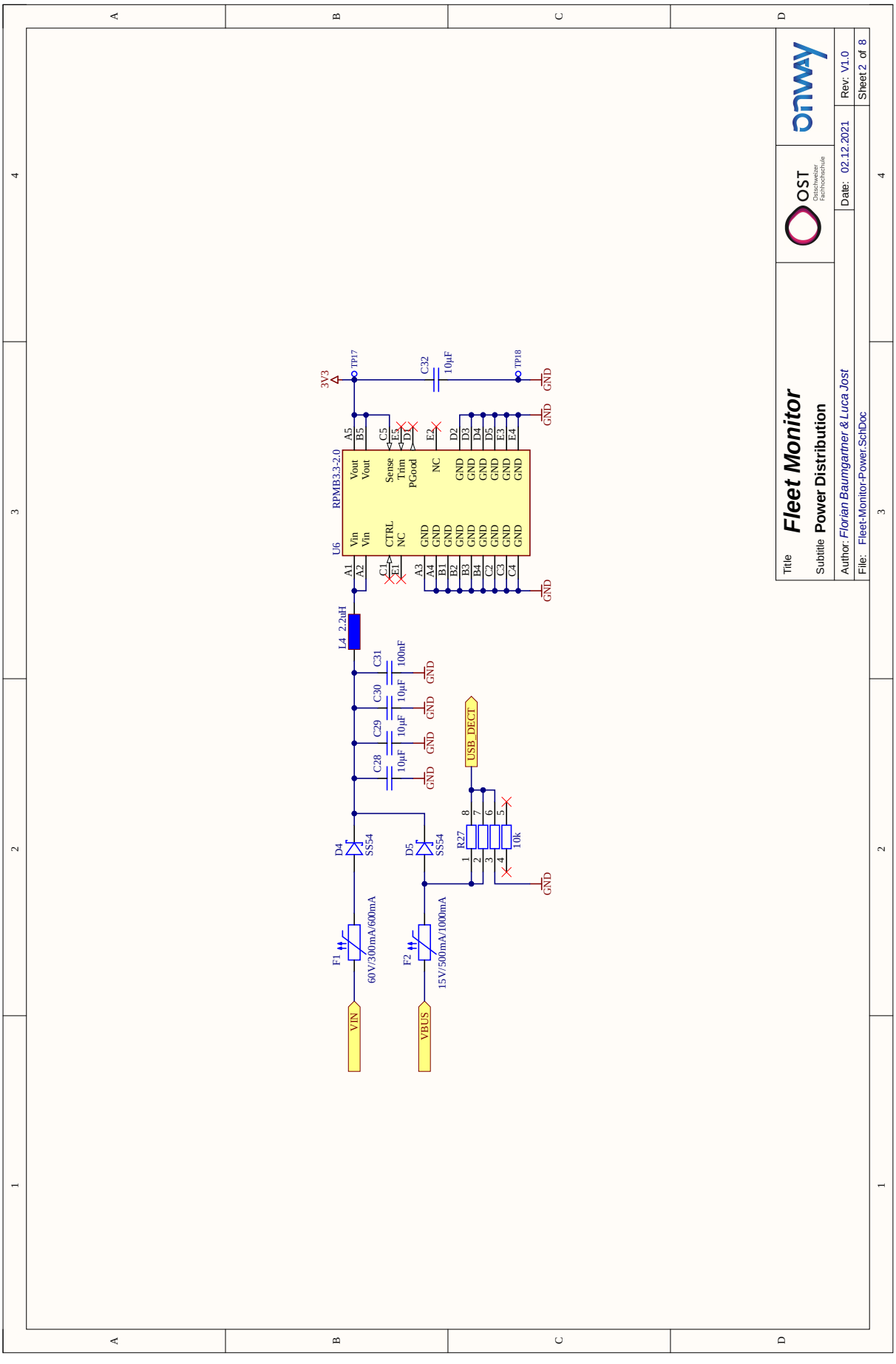
Location, Date
Rapperswil, 23. December 2021

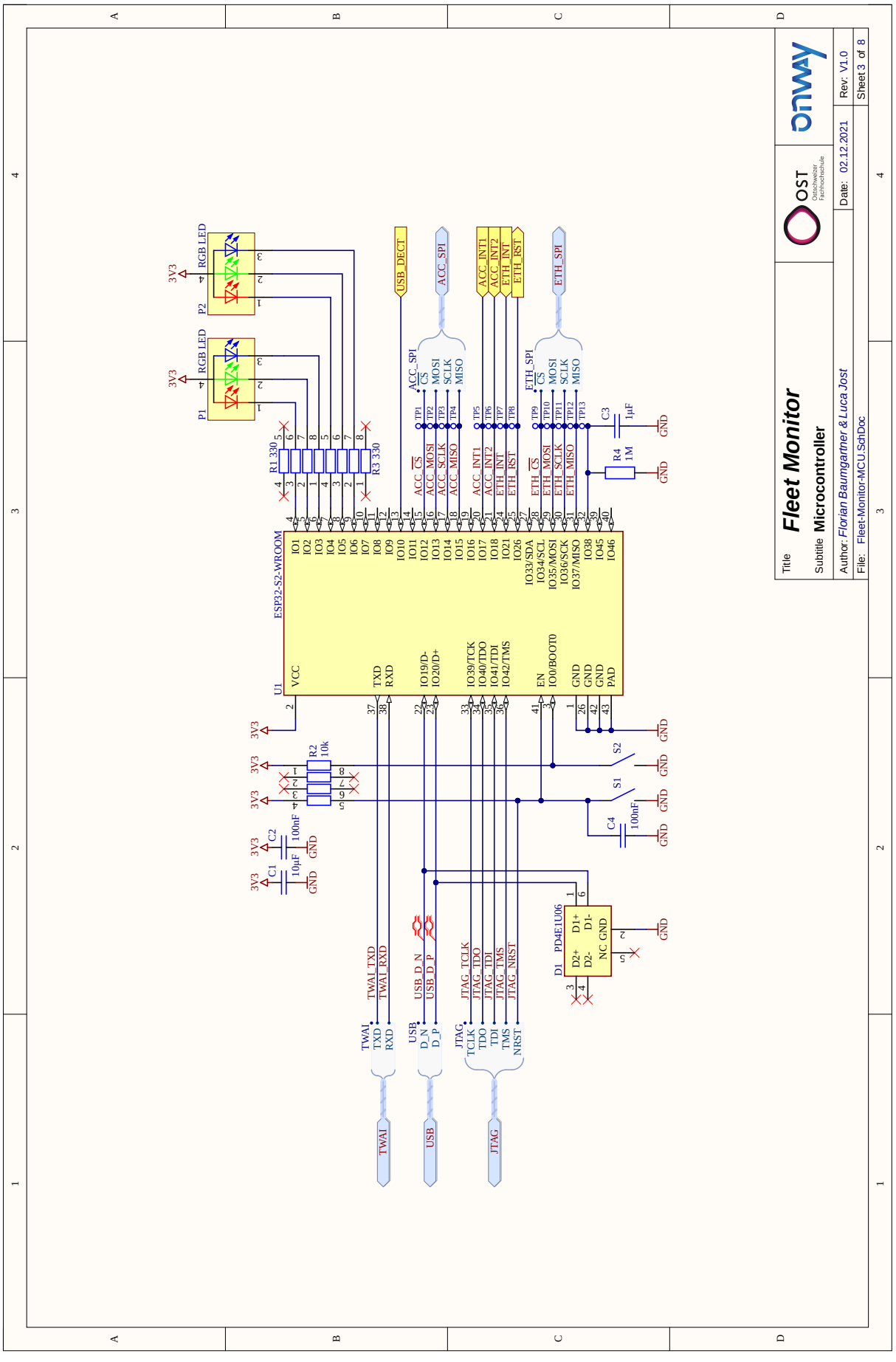
Florian Baumgartner

Luca Jost

A.2 Fleet-Monitor V1.0 Schematics




Title Fleet Monitor
Subtitle Power Distribution
Author: Florian Baumgartner & Luca Jost
File: Fleet-Monitor-Power.SchDoc
Date: 02.12.2021
Rev: V1.0
Sheet 2 of 8
OST
Ostschweizer Fachhochschule
3V3
TP17
TP18
C32
10µF
GND
U6
RPMB3.3-2.0
Vin
CTRL
NC
Vout
Sense
Trim
PGood
NC
L4 2.2uH
C31 100nF
C30 10µF
C29 10µF
C28 10µF
D4 SS54
D5 SS54
F1 60V/300mA/600mA
F2 15V/500mA/1000mA
VIN
VBUS
USB_DECT
R27 10k


Title Fleet Monitor
Subtitle Microcontroller
Author: Florian Baumgartner & Luca Jost
File: Fleet-Monitor-MCU.SchDoc
Date: 02.12.2021
Rev: V1.0
Sheet 3 of 8
OST
Ostschweizer Fachhochschule
ONWAY
ESP32-S2-WROOM
U1
RGB LED
P1
P2
R1 330
R3 330
R2 10k
R4 1M
C1 10µF
C2 100nF
C3 1µF
C4 100nF
S1
S2
D1 PD4E1U06
3V3
GND
USB_DECT
ACC_SPI
ACC_INT1
ACC_INT2
ETH_INT
ETH_RST
ETH_SPI
TWAI
USB
JTAG

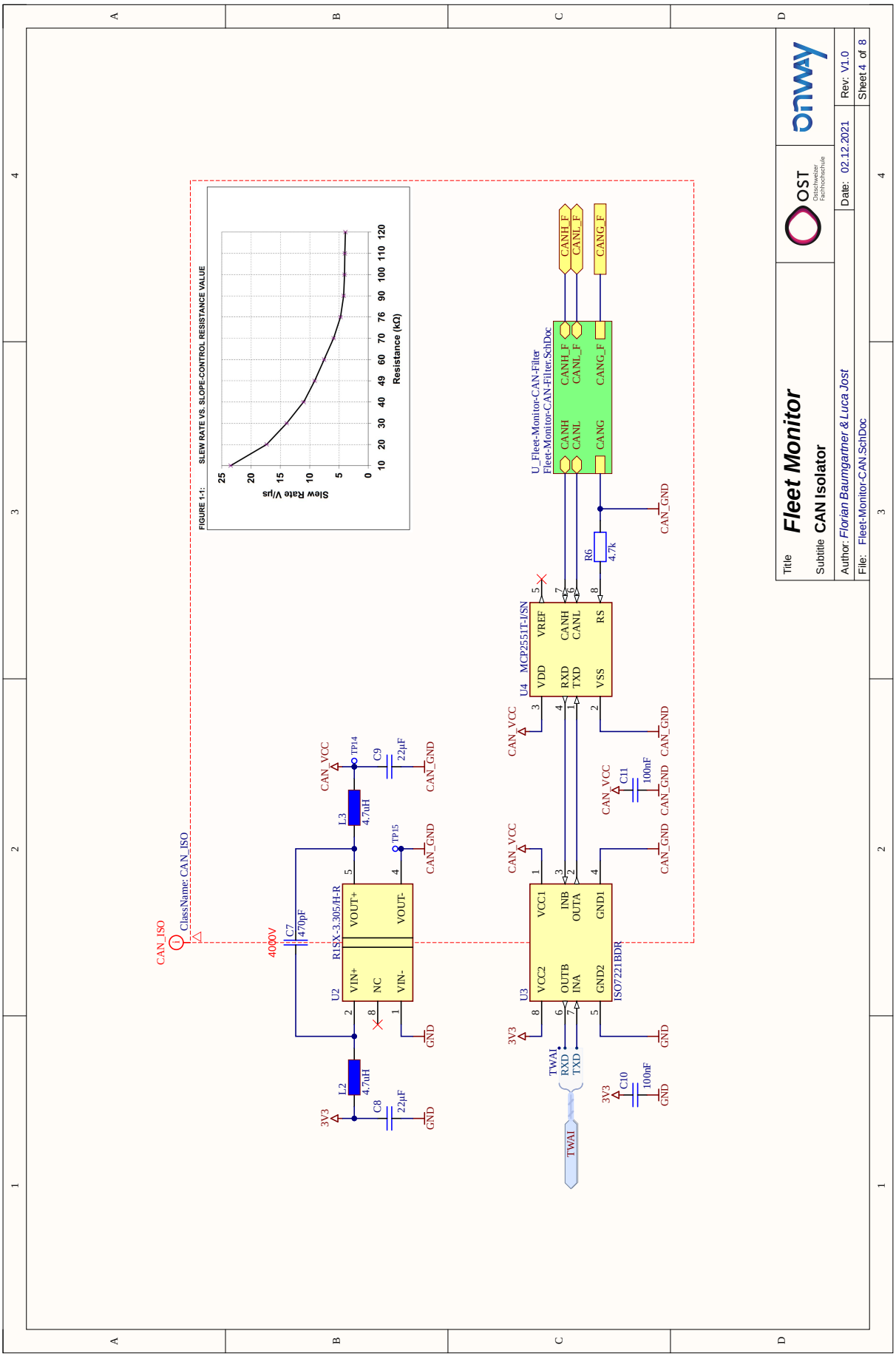
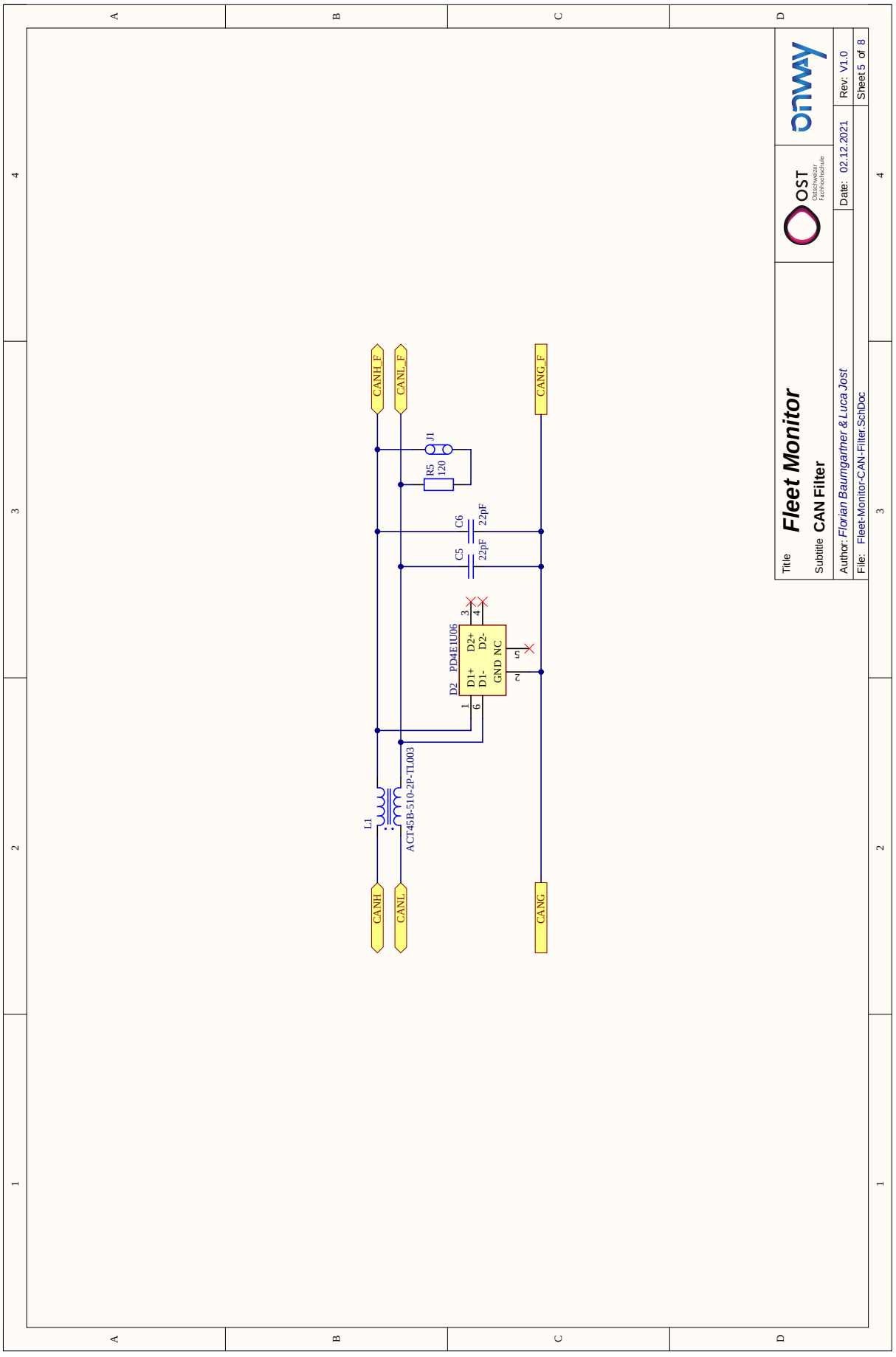
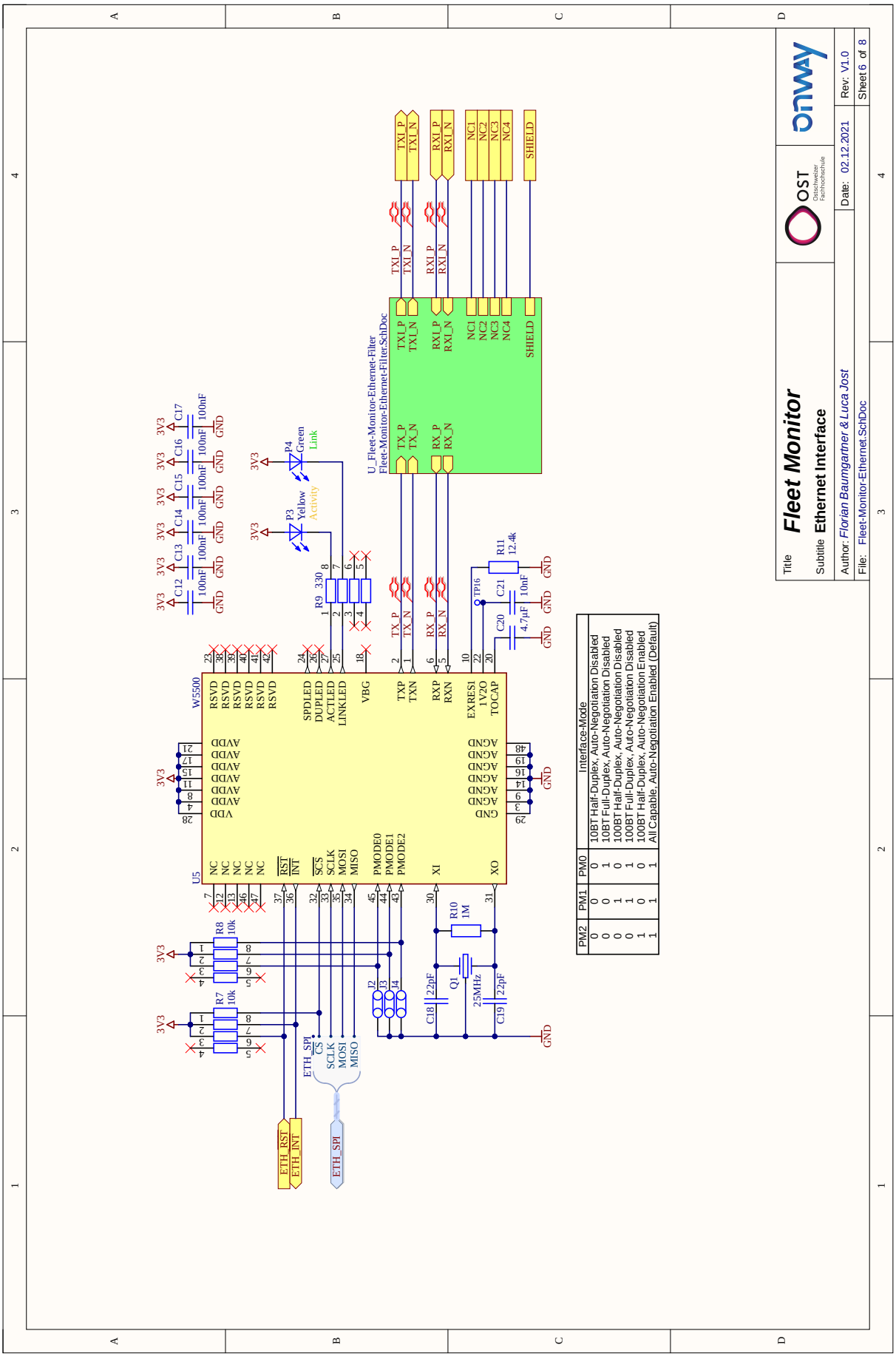
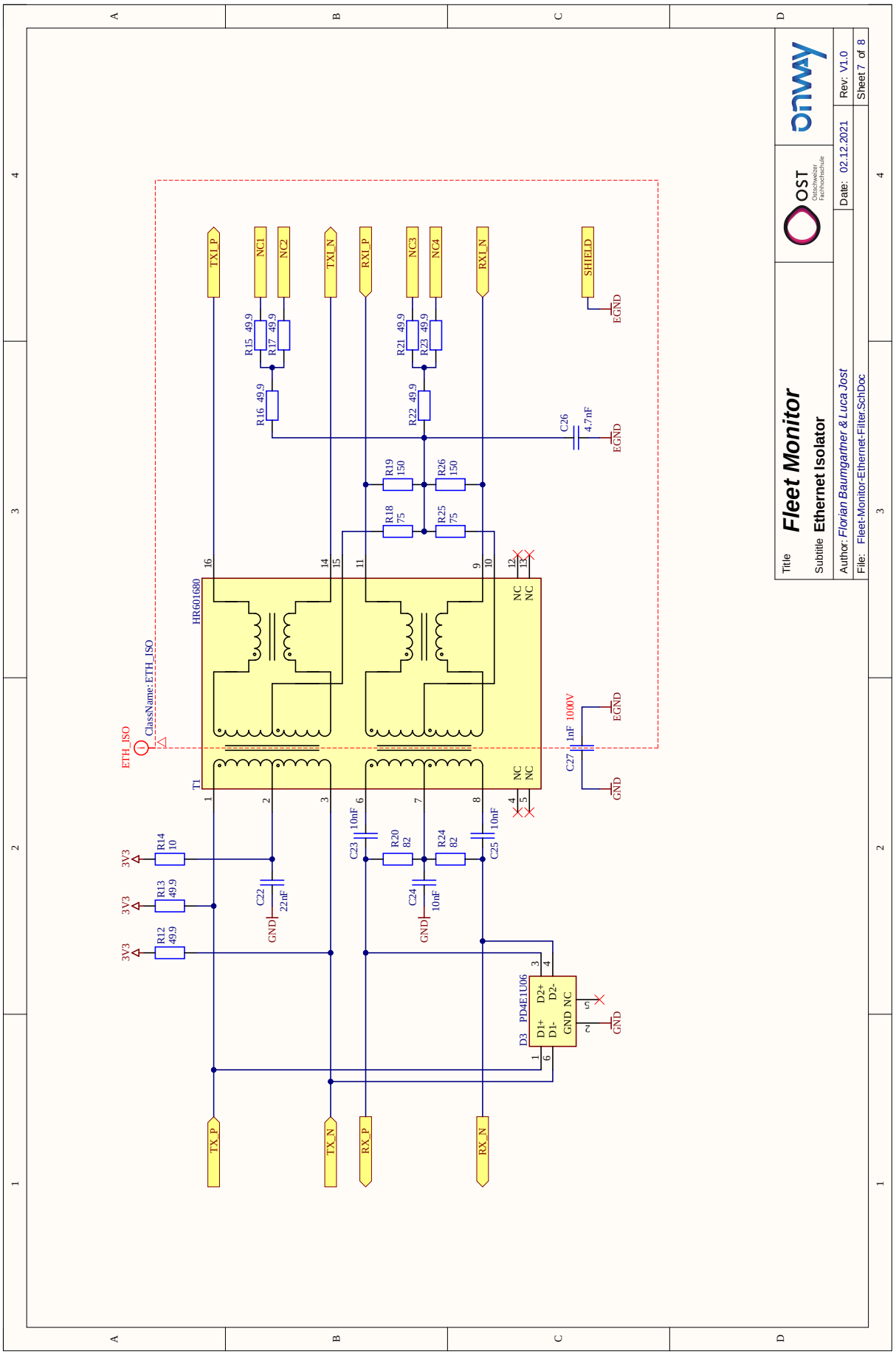
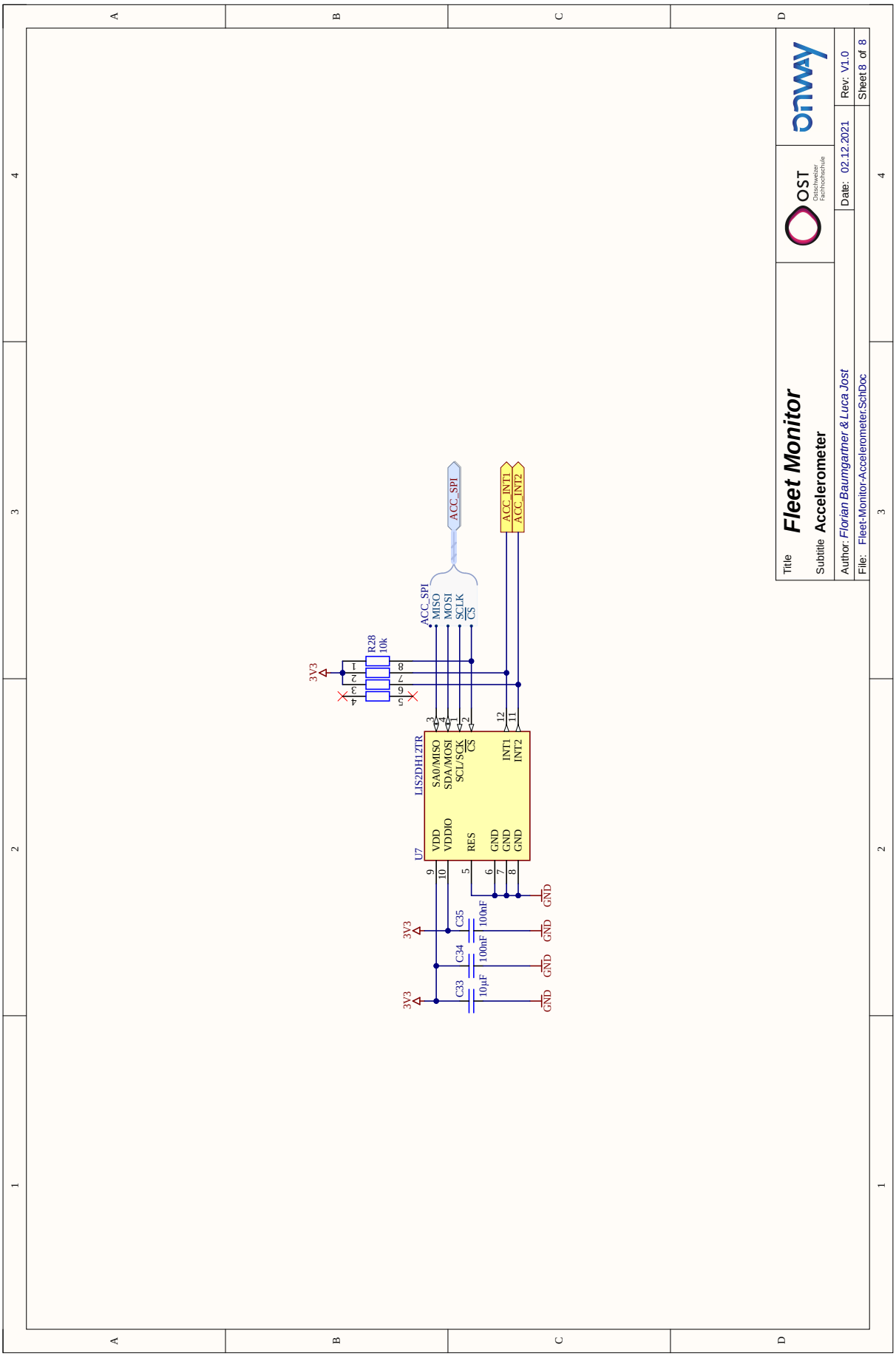


FIGURE 1-1: SLEW RATE VS. SLOPE-CONTROL RESISTANCE VALUE
Slew Rate V/µs
Resistance (kΩ)
Title Fleet Monitor
Subtitle CAN Isolator
Author: Florian Baumgartner & Luca Jost
Date: 02.12.2021
Rev: V1.0
File: Fleet-Monitor-CAN.SchDoc
Sheet 4 of 8
CAN_ISO
ClassName: CAN_ISO
U2 R1SX-3.305/H-R
U3 ISO7221BDR
U4 MCP2551T-I/SN
U_Fleet-Monitor-CAN-Filter
Fleet-Monitor-CAN-Filter.SchDoc
CANH_F
CANL_F
CANG_F
CAN_VCC
CAN_GND
3V3
GND
TWAI
RXD
TXD
4000V
C7 470pF
C8 22µF
C9 22µF
C10 100nF
C11 100nF
L2 4.7uH
L3 4.7uH
R6 4.7k
TP14
TP15


CANH_F
CANL_F
CANG_F
J1
R5
120
C6
22pF
C5
22pF
D2
PD4E1U06
D1+
D1-
GND
D2+
D2-
NC
1
6
3
4
5
2
L1
ACT45B-510-2P-TL003
CANH
CANL
CANG
Title
Fleet Monitor
Subtitle
CAN Filter
Author: Florian Baumgartner & Luca Jost
File: Fleet-Monitor-CAN-Filter.SchDoc
Date: 02.12.2021
Rev: V1.0
Sheet 5 of 8


Title Fleet Monitor
Subtitle Ethernet Interface
Author: Florian Baumgartner & Luca Jost
Date: 02.12.2021
Rev: V1.0
File: Fleet-Monitor-Ethernet.SchDoc
Sheet 6 of 8
OST
Ostschweizer Fachhochschule
U5
W5500
U_Fleet-Monitor-Ethernet-Filter
Fleet-Monitor-Ethernet-Filter.SchDoc
P3 Yellow Activity
P4 Green Link
ETH_RST
ETH_INT
ETH_SPI
PM2 PM1 PM0 Interface-Mode
0 0 0 10BT Half-Duplex, Auto-Negotiation Disabled
0 0 1 10BT Full-Duplex, Auto-Negotiation Disabled
0 1 0 100BT Half-Duplex, Auto-Negotiation Disabled
0 1 1 100BT Full-Duplex, Auto-Negotiation Disabled
1 0 0 100BT Half-Duplex, Auto-Negotiation Enabled
1 1 1 All Capable, Auto-Negotiation Enabled (Default)


Title Fleet Monitor
Subtitle Ethernet Isolator
Author: Florian Baumgartner & Luca Jost
File: Fleet-Monitor-Ethernet-Filter.SchDoc
Date: 02.12.2021
Rev: V1.0
Sheet 7 of 8
OST
Ostschweizer Fachhochschule
TXI_P
TXI_N
RXI_P
RXI_N
NC1
NC2
NC3
NC4
SHIELD
R15 49.9
R17 49.9
R16 49.9
R21 49.9
R23 49.9
R22 49.9
C26 4.7nF
EGND
R19 150
R26 150
R18 75
R25 75
HR601680
T1
ETH_ISO
ClassName:ETH_ISO
NC
C27 1nF 100V
GND
R14 10
R13 49.9
R12 49.9
3V3
C22 22nF
C23 10nF
C24 10nF
C25 10nF
R20 82
R24 82
D3 PD4E1U06
D1+ D1- GND D2+ D2- NC
TX_P
TX_N
RX_P
RX_N


Title Fleet Monitor
Subtitle Accelerometer
Author: Florian Baumgartner & Luca Jost
File: Fleet-Monitor-Accelerometer.SchDoc
Date: 02.12.2021
Rev: V1.0
Sheet 8 of 8
OST
Ostschweizer Fachhochschule
ACC_SPI
ACC_INT1
ACC_INT2
R28
10k
3V3
U7
LIS2DH12TR
VDD
VDDIO
RES
GND
SA0/MISO
SDA/MOSI
SCL/SCK
CS
INT1
INT2
C33
10µF
C34
100nF
C35
100nF
MISO
MOSI
SCLK
CS

A.3 Fleet-Monitor V1.0 BOM

Bill of Materials

File name: Fleet-Monitor
Project: Fleet-Monitor.PrjPcb
Variant: V1.0

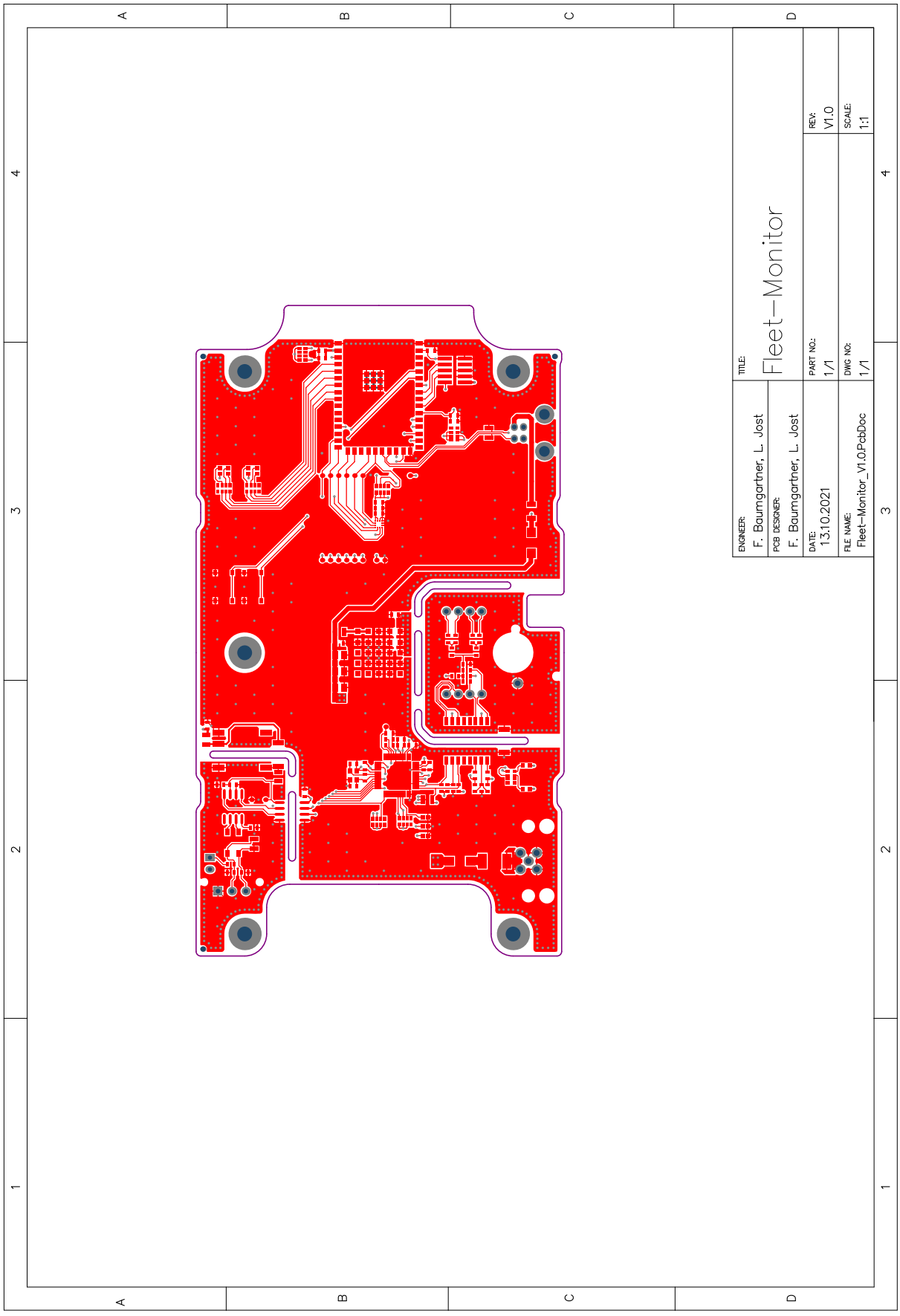
Creation Date: 24.10.2021
Print Date: 24.11.2021

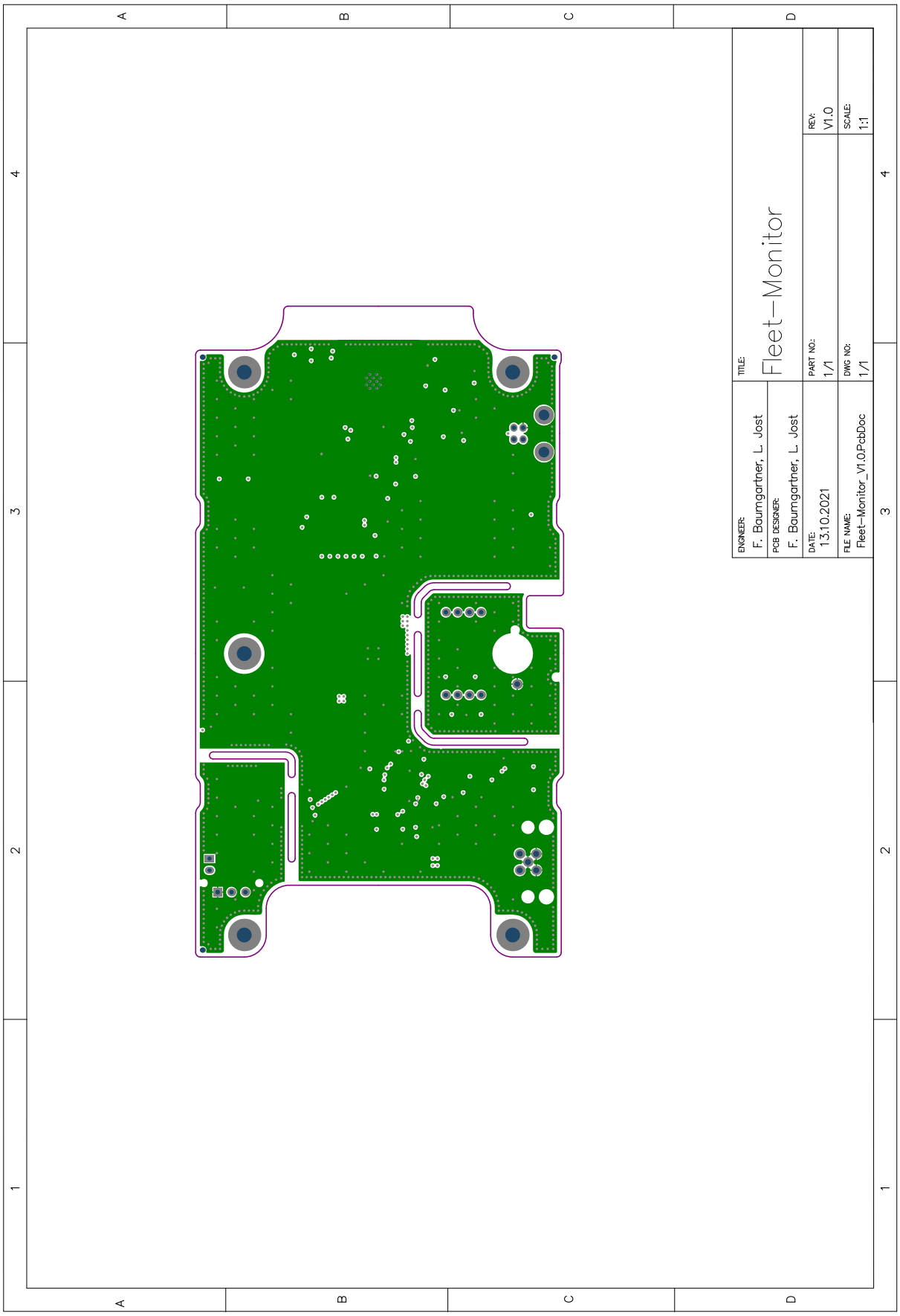
Designator	Component Description	Component Value	Footprint	JLC-Parts	Farnell	Mouser	Digikey	Quantity	Unit Price	Total Price
C1, C32, C33	Multilayer Ceramic Capacitors MLCC	10µF 10V	SMD Keramikkondensator 0603	C19702	2113070	963-LMK107BBJ106MALT	587-3258-1-ND	3	SFr. 0.01	SFr. 0.03
C2, C4, C10, C11, C12, C13, C14, C15, C16, C17, C31, C34, C35	Multilayer Ceramic Capacitors MLCC	100nF 50V	SMD Keramikkondensator 0603	C14663	2210822	810-CGA2B3X7R1H104K	445-6899-1-ND	13	SFr. 0.01	SFr. 0.13
C3	Multilayer Ceramic Capacitors MLCC	1µF 50V	SMD Keramikkondensator 0603	C15849	3013383	187-CL10A105KB8NNNC	1276-1860-1-ND	1	SFr. 0.01	SFr. 0.01
C5, C6, C18, C19	Multilayer Ceramic Capacitors MLCC	22pF 50V	SMD Keramikkondensator 0603	C1653	2332643	187-CL10C220JB8NNNC	1276-1023-1-ND	4	SFr. 0.01	SFr. 0.04
C7	Multilayer Ceramic Capacitors MLCC	470pF 3kV	SMD Keramikkondensator 1808	C106130	2812607	710-885342210003	732-12100-1-ND	1	SFr. 0.11	SFr. 0.11
C8, C9	Multilayer Ceramic Capacitors MLCC	22µF 25V	SMD Keramikkondensator 0603	C45783	3013459	187-CL21A226MAQNNNE	1276-2908-1-ND	2	SFr. 0.04	SFr. 0.08
C20	Multilayer Ceramic Capacitors MLCC	4.7µF 16V	SMD Keramikkondensator 0603	C19666	3013397	187-CL10A475KO8NNNC	1276-1784-1-ND	1	SFr. 0.01	SFr. 0.01
C21, C23, C24, C25	Multilayer Ceramic Capacitors MLCC	10nF 50V	SMD Keramikkondensator 0603	C57112	1833876	581-06031C103J	478-3700-1-ND	4	SFr. 0.01	SFr. 0.04
C22	Multilayer Ceramic Capacitors MLCC	22nF 50V	SMD Keramikkondensator 0603	C21122	3013425	187-CL10B223KB8NNNC	1276-1104-1-ND	1	SFr. 0.01	SFr. 0.01
C26	Multilayer Ceramic Capacitors MLCC	4.7nF 50V	SMD Keramikkondensator 0603	C53987	2332650	581-06035C472JAT2A	478-10355-1-ND	1	SFr. 0.01	SFr. 0.01
C27	Multilayer Ceramic Capacitors MLCC	1nF 3kV	SMD Keramikkondensator 1808	C173224	1886070	605-302R29W102KV4E	709-1047-1-ND	1	SFr. 0.06	SFr. 0.06
C28, C29, C30	Multilayer Ceramic Capacitors MLCC	10µF 50V	SMD Keramikkondensator 0805	C440198	3790440	81-GRM21BR61H106KE3L	490-18663-1-ND	3	SFr. 0.13	SFr. 0.39
D1, D2, D3	ESD Suppressor	PD4E1U06	SMD IC SOT-363 (SC-70-6)	C194628	3116516	595-TPD4E1U06DCKR	296-39347-1-ND	3	SFr. 0.12	SFr. 0.36
D4, D5	Schottky Barrier Diode	SS54	SMD Diode DO-214-AC	C22452	1861427	750-SS54-HF	641-SS54-HFCT-ND	2	SFr. 0.10	SFr. 0.20
F1	PTC Resettable Fuse	60V/300mA/600mA/100ms	SMD PPTC-Fuse 1812	C117855	1961501	576-1812L035/60MR	F8131CT-ND	1	SFr. 0.07	SFr. 0.07
F2	PTC Resettable Fuse	15V/500mA/1000mA/100ms	SMD PPTC-Fuse 1206	C315893	2752736	576-1206L050/15YR	F3733CT-ND	1	SFr. 0.23	SFr. 0.23
J1	Through Hole Jumper	2-Pol	THT Jumper	C358684	2668454	538-22-28-4023	WM50016-02-ND	1	SFr. 0.01	SFr. 0.01
J2, J3, J4	Solder Jumper	2-Pol	SMD Solder Jumper 1x2mm					3	SFr. 0.00	SFr. 0.00
L1	Common Mode Filter	ACT45B-101-2P-TL003	SMD Common Mode Choke 4.5x3.2	C88056	3407556	810-ACT45B1012PTL003	445-3959-1-ND	1	SFr. 0.56	SFr. 0.56
L2, L3	Power Inductor	4.7uH	SMD INDUCTOR/FERRITE 0805	C86094	2840129	81-LQM2MPN4R7MG0L	490-12073-1-ND	2	SFr. 0.09	SFr. 0.18
L4	Power Inductor	2.2uH	SMD INDUCTOR/FERRITE 0805	C136243	2470314	810-TFM201610GHM2R2A	445-173104-1-ND	1	SFr. 0.13	SFr. 0.13
P1, P2	RGB LED	67-23/R6GHBHC-B01/2T	SMD LED PLCC-4	C264607		593-VAOS-SP4RGB4	VAOS-SP4RGB4CT-ND	2	SFr. 0.19	SFr. 0.38
P3	General Purpose LED	Yellow (5mA, 2.0V)	SMD LED 0603	C72038	1465989	645-598-8040-107F	350-2032-1-ND	1	SFr. 0.34	SFr. 0.34
P4	General Purpose LED	Green (5mA, 2.4V)	SMD LED 0603	C72043	1465991	645-598-8070-107F	350-2035-1-ND	1	SFr. 0.34	SFr. 0.34
Q1	Crystal	25MHz	SMD Crystal 2520	C252264	3649710	520-250-10-36B-CTNT	XC3050CT-ND	1	SFr. 0.14	SFr. 0.14
R1, R3, R9	Isolated Resistor Array	330 (4x, 50V, 62.5mW, 5%)	SMD Widerstand Array 4-Fach 0603	C12018	2060160	667-EXB-V8V331JV	Y4331CT-ND	3	SFr. 0.01	SFr. 0.03
R2, R7, R8, R27, R28	Isolated Resistor Array	10k (4x, 50V, 62.5mW, 5%)	SMD Widerstand Array 4-Fach 0603	C29718	2060167	667-EXB-V8V103JV	Y4103CT-ND	5	SFr. 0.01	SFr. 0.05
R4, R10	Chip Resistor	1M (75V, 100mW, 1%)	SMD Widerstand 0603	C22935	2303325	667-ERJ-3EKF1004V	P1.00MHCT-ND	2	SFr. 0.01	SFr. 0.02
R5	Chip Resistor	120 (75V, 100mW, 1%)	SMD Widerstand 0603	C22787	2303067	667-ERJ-3EKF1200V	P120HCT-ND	1	SFr. 0.01	SFr. 0.01
R6	Chip Resistor	4.7k (75V, 100mW, 1%)	SMD Widerstand 0603	C23162	2303181	667-ERJ-3EKF4701V	P4.70KHCT-ND	1	SFr. 0.01	SFr. 0.01
R11	Chip Resistor	12.4k (75V, 100mW, 1%)	SMD Widerstand 0603	C194071	2059414	667-ERJ-3EKF1242V	P12.4KHCT-ND	1	SFr. 0.01	SFr. 0.01
R12, R13, R15, R16, R17, R21, R22, R23	Chip Resistor	49.9 (75V, 100mW, 1%)	SMD Widerstand 0603	C23184	2059274	667-ERJ-3EKF49R9V	P49.9HCT-ND	8	SFr. 0.01	SFr. 0.08
R14	Chip Resistor	10 (75V, 100mW, 1%)	SMD Widerstand 0603	C22859	2302958	667-ERJ-3EKF10R0V	P10.0HCT-ND	1	SFr. 0.01	SFr. 0.01
R18, R25	Chip Resistor	75 (75V, 100mW, 1%)	SMD Widerstand 0603	C4275	2059282	667-ERJ-3EKF75R0V	P75.0HCT-ND	2	SFr. 0.01	SFr. 0.02
R19, R26	Chip Resistor	150 (75V, 100mW, 1%)	SMD Widerstand 0603	C22808	2303076	667-ERJ-3EKF1500V	P150HCT-ND	2	SFr. 0.01	SFr. 0.02
R20, R24	Chip Resistor	82 (75V, 100mW, 1%)	SMD Widerstand 0603	C23255	2303052	667-ERJ-3EKF82R0V	P82.0HCT-ND	2	SFr. 0.01	SFr. 0.02
S1, S2	Tactile Switch	TS-1187A-B-A-B	SMD Button 5.2x5.2	C318884	3780596	611-PTS526SK15SMR2L	CKN12221-1-ND	2	SFr. 0.02	SFr. 0.04
T1	RJ45 Transformer	HR601680	SMD Transformer 500x375	C172799	2673883	652-SM13072APEL	SM13072APELCT-ND	1	SFr. 0.57	SFr. 0.57
U1	Xtensa single-core 32-bit LX7 MCU	ESP32-S2-WROOM	SMD ESP32-S2-WROOM	C967026		356-ESP32S2WRMI320UH	1965-ESP32-S2-WROOMTR-ND	1	SFr. 2.54	SFr. 2.54
U2	Isolated DC-DC-Converter	R1SX-3.305/H-R	SMD DCDC R1SX			919-R1SX-3.305/H-R	945-3197-1-ND	1	SFr. 3.94	SFr. 3.94
U3	Unidirectional Digital Isolator	ISO7221BDR	SMD IC SOIC-8	C138522		595-ISO7221BDR	296-38965-2-ND	1	SFr. 2.94	SFr. 2.94
U4	CAN Transceiver 1Mbps	MCP2551T-I/SN	SMD IC SOIC-8	C49558	2763381	579-MCP2551T-I/SN	MCP2551T-I/SNTR-ND	1	SFr. 1.04	SFr. 1.04
U5	Ethernet Controller	W5500	SMD IC LQFP-48	C32843		950-W5500	1278-W5500TR-ND	1	SFr. 3.76	SFr. 3.76
U6	DC-DC Stepdown Converter	RPMB3.3-3.0	SMD DCDC RPM 2.0		3551809	919-RPMB3.3-3.0	945-RPMB3.3-3.0CT-ND	1	SFr. 7.43	SFr. 7.43
U7	High-Performance 3-Xxis Accelerometer	LIS2DH12TR	SMD IC LGA-12	C110926		511-LIS2DH12TR	497-14851-2-ND	1	SFr. 0.87	SFr. 0.87
X1	M12 Receptacle Connector PCB Assembly	T4145015021-001	THT Connector M12				A126521-ND	1	SFr. 6.97	SFr. 6.97
X2	Pin Header Through Hole Wire To Board	436500315	THT Connector 0436500315	C293546	1654451	538-43650-0315	WM1918-ND	1	SFr. 0.38	SFr. 0.38
X3	USB-Buchse	MUSBD111M5	THT Connector MUSBD111M5		3728524	523-MUSBD111M5	MUSBD111M5-ND	1	SFr. 11.17	SFr. 11.17
X4	RJ45 etherCON IP65	NE8FDH-C5E-TOP	THT Connector NE8FDH-C5E-TOP		3014323	568-NE8FDH-C5E-TOP		1	SFr. 12.83	SFr. 12.83
X5	JTAG-Header (ARM-Cortex)	SHF-105-01-L-D-SM (2x5)	SMD Connector 1.27mm (2x5)		1885915	200-SHF10501LDSM	SAM10710-ND	1	SFr. 2.89	SFr. 2.89
Case	Polycarbonat with Transparent Lid	1554J2GYCL			2904345	546-1554J2GYCL	HM1799-ND	1	SFr. 21.23	SFr. 21.23
Plastic Vent	Vent Screw-Type M6	VENT-PS2NBK-O8001			2991687	523-VENT-PS2NBKO8001	1754-1549-ND	1	SFr. 3.80	SFr. 3.80
Cable Assembly	Micro-Fit 3.0 Female-to-Micro-Fit 3.0 Female	2147501032			3526486	538-214750-1032	900-2147501032-ND	1	SFr. 3.06	SFr. 3.06
FMS-Connector	KALI-2004 DIN 72585 4 Pole (Set), Kalitec Shop: 00013385	1-0967325-1 (1x), 1-0967402-1 (1x), 0-0965687-1 (1x), 0-0962981-1 (4x), 0-0962971-1 (4x), 0-0828920-1 (8x)						1	SFr. 7.15	SFr. 7.15
Connector Cable-Cap	Cap 180° for Corrugated Pipe NW 8.5, Kalitec Shop: 00011856	0-0965785-1						1	SFr. 0.62	SFr. 0.62
										SFr. 97.37

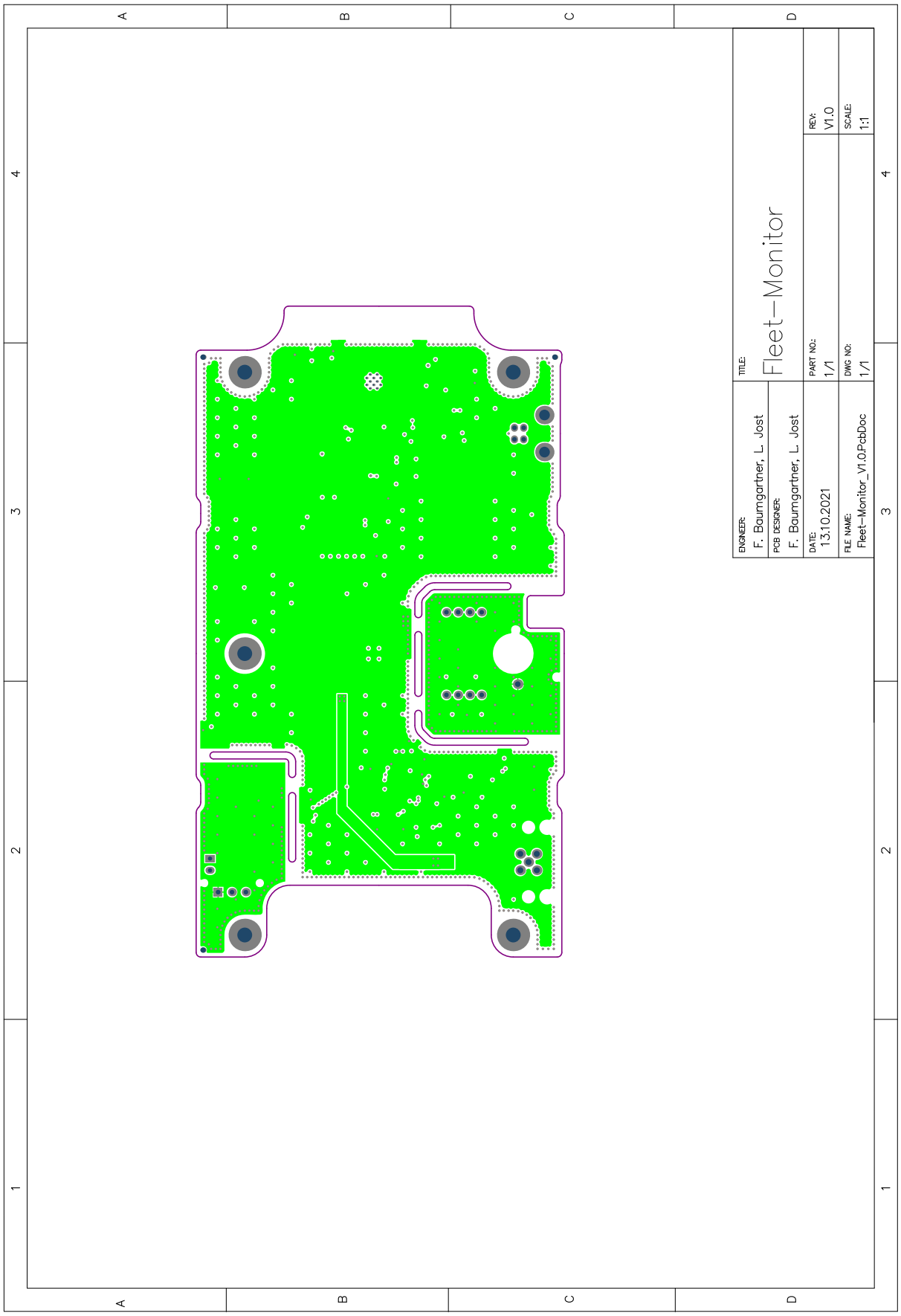
Autor

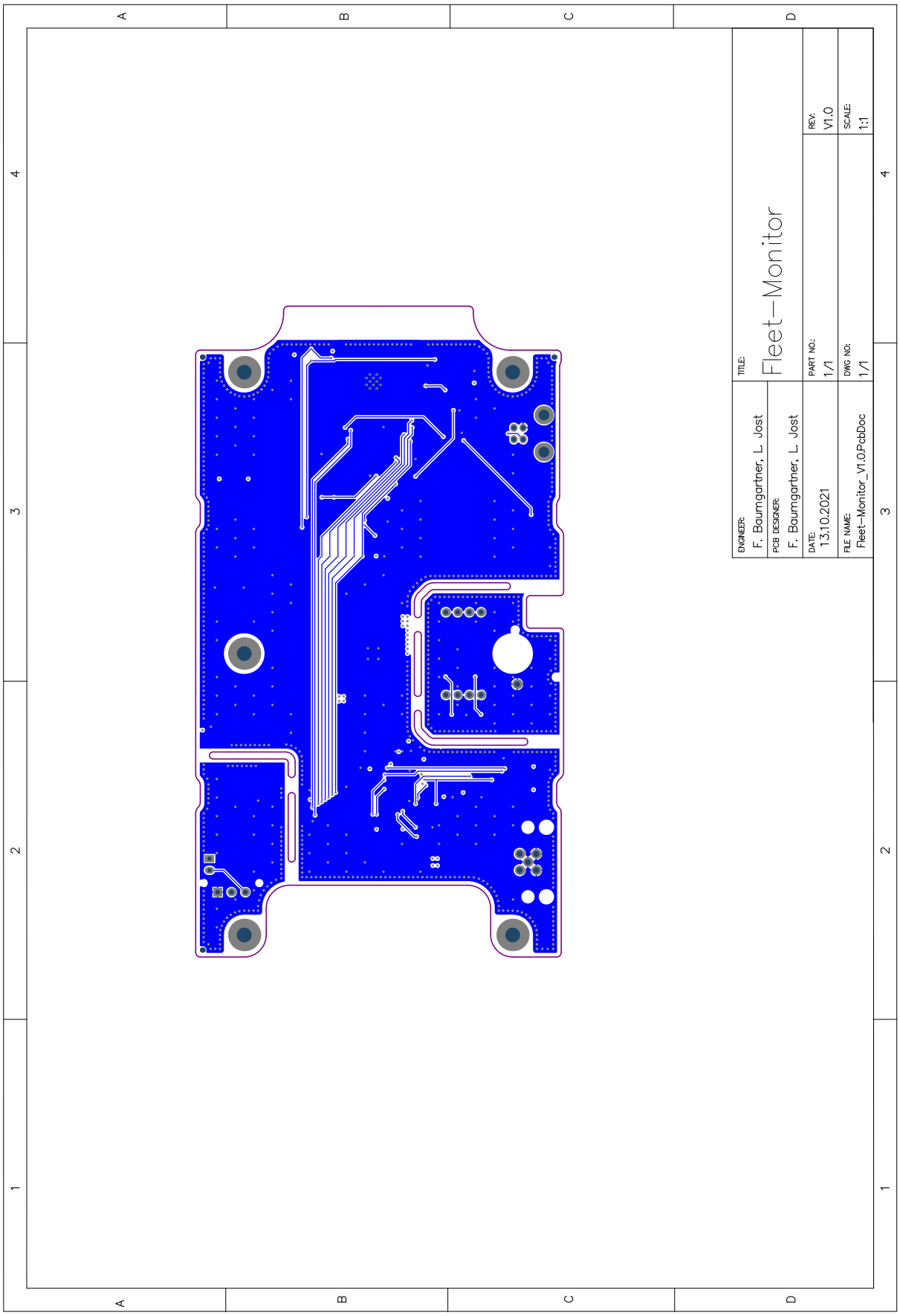
Florian Baumgarnter & Luca Jost

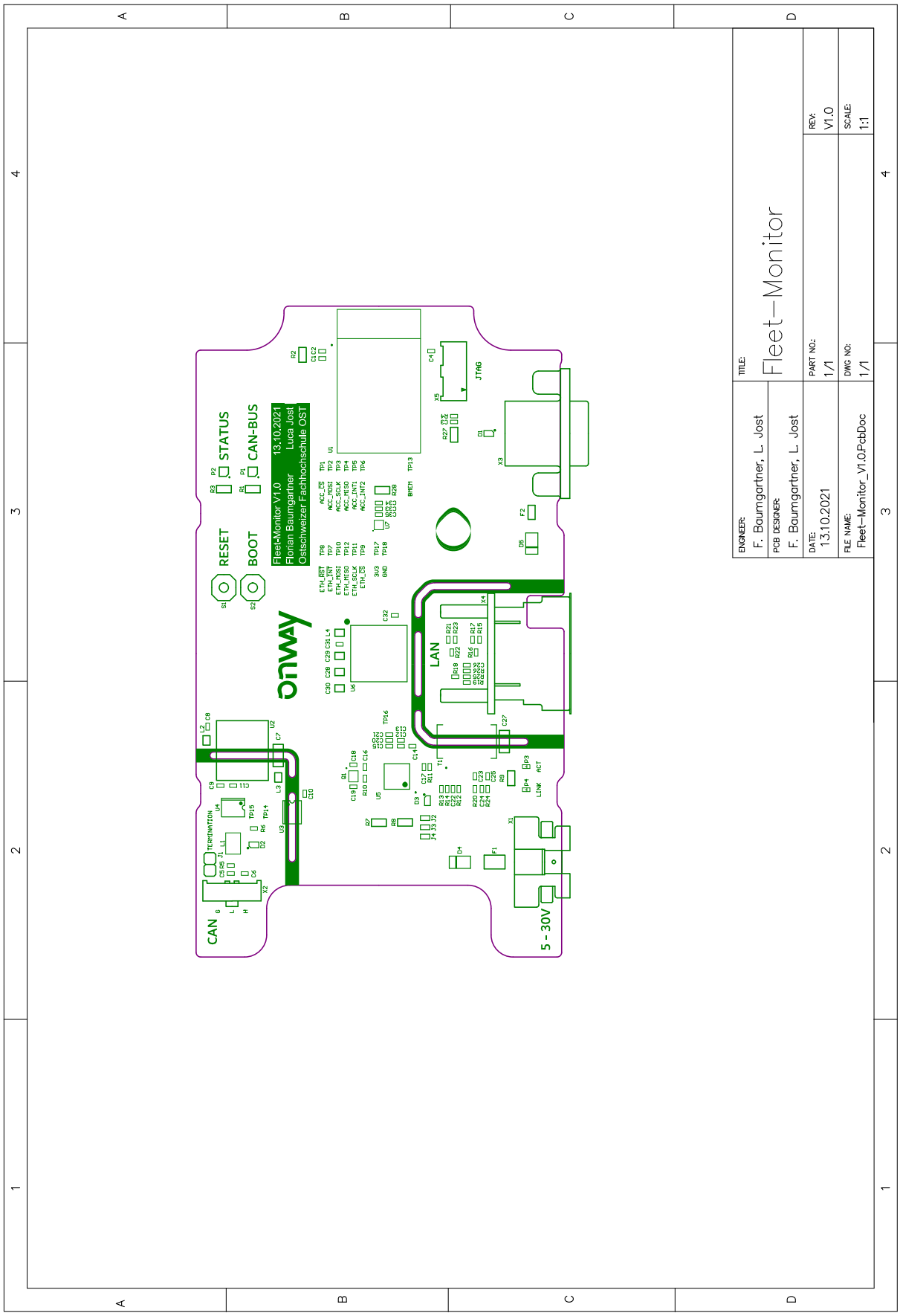
A.4 Fleet-Monitor V1.0 PCB Layout

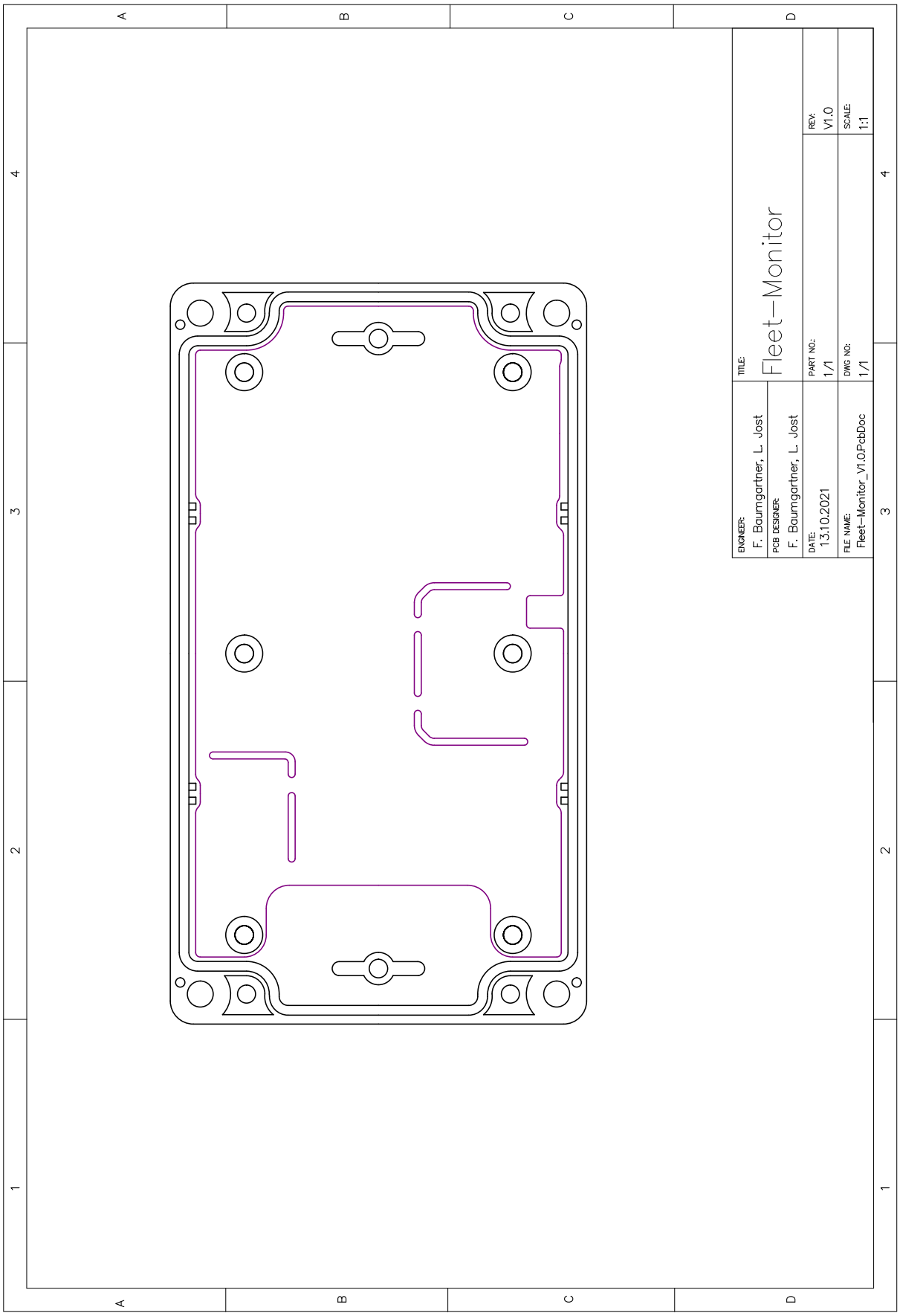



A
B
C
D
1
2
3
4
TITLE:
Fleet–Monitor
ENGINEER:
F. Baumgartner, L. Jost
PCB DESIGNER:
F. Baumgartner, L. Jost
DATE:
13.10.2021
FILE NAME:
Fleet–Monitor_V1.0.PcbDoc
PART NO.:
1/1
DWG NO:
1/1
REV:
V1.0
SCALE:
1:1

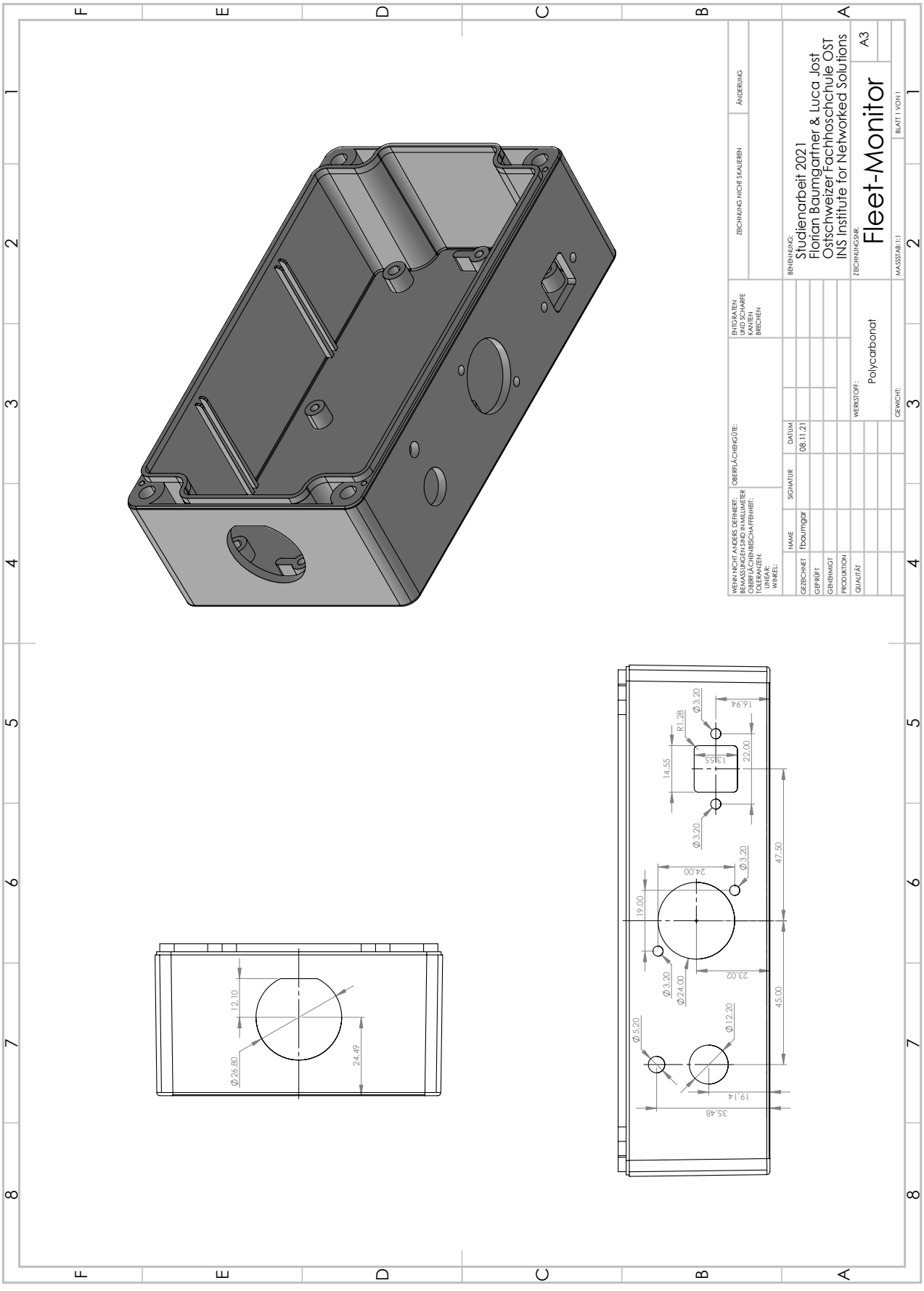

A
B
C
D
1
2
3
4
TITLE:
Fleet-Monitor
ENGINEER:
F. Baumgartner, L. Jost
PCB DESIGNER:
F. Baumgartner, L. Jost
DATE:
13.10.2021
PART NO.:
1/1
REV:
V1.0
FILE NAME:
Fleet-Monitor_V1.0.PcbDoc
DWG NO:
1/1
SCALE:
1:1


A
B
C
D
1
2
3
4
TITLE:
Fleet–Monitor
ENGINEER:
F. Baumgartner, L. Jost
PCB DESIGNER:
F. Baumgartner, L. Jost
DATE:
13.10.2021
FILE NAME:
Fleet–Monitor_V1.0.PcbDoc
PART NO.:
1/1
DWG NO:
1/1
REV:
V1.0
SCALE:
1:1


Fleet-Monitor V1.0
Florian Baumgartner
Ostschweizer Fachhochschule OST
13.10.2021
Luca Jost
RESET
BOOT
STATUS
CAN-BUS
LAN
JTAG
CAN
5 - 30V
TERMINATION
ENGINEER: F. Baumgartner, L. Jost
PCB DESIGNER: F. Baumgartner, L. Jost
DATE: 13.10.2021
FILE NAME: Fleet-Monitor_V1.0.PcbDoc
TITLE: Fleet-Monitor
PART NO.: 1/1
DWG NO: 1/1
REV: V1.0
SCALE: 1:1


A
B
C
D
1
2
3
4
TITLE:
Fleet–Monitor
ENGINEER:
F. Baumgartner, L. Jost
PCB DESIGNER:
F. Baumgartner, L. Jost
DATE:
13.10.2021
FILE NAME:
Fleet–Monitor_V1.0.PcbDoc
PART NO.:
1/1
DWG NO:
1/1
REV:
V1.0
SCALE:
1:1

A.5 Fleet-Monitor V1.0 Mechanical Drawing



A.6 Test Reports

A.6.1 Long Duration Test

Test Report Fleet-Monitor

Long Duration Test

Title	**Long Duration Test**
Project	Fleet-Monitor
Date	10.12.2021
Prepared by	Luca Jost
Checked by	Florian Baumgartner

Contents

Test Report Fleet-Monitor

1. Test Introduction

1.1. Scope of the Test

The scope of the long duration test is to verify the operation of the Fleet-Monitor over a long period of time.

In detail the following functions should be tested:

- Memory fragmentation over long period of time
- Connectivity issues
- WIFI connection
- Ethernet connection

The test should run for at least 48 hours to get a good result.

1.2. Acceptance criteria

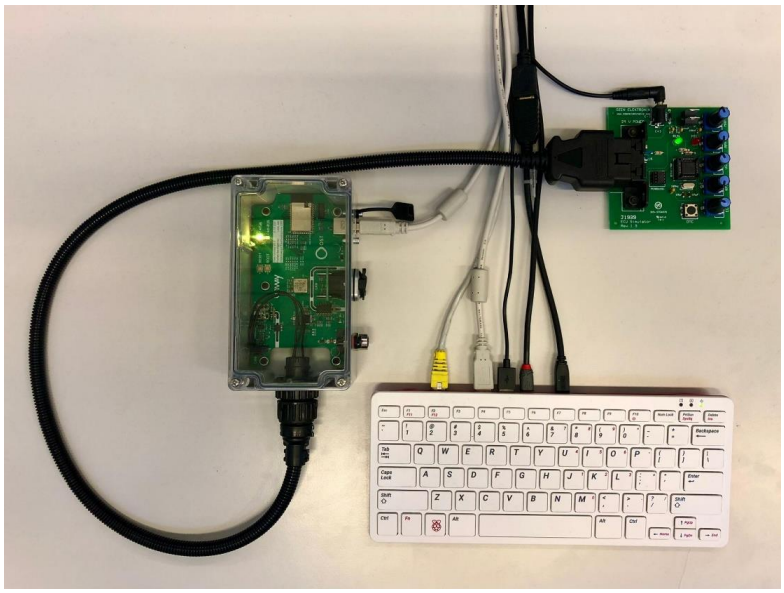
The test is successful if:

1. The device is still running after 48 hours
2. Data is still read from the CAN bus
3. Data is still transmitted to the server
4. Hot swapping networking interface is still working

2. Test Setup

2.1. Setup

The Fleet-Monitor is connected to the J1939 CAN simulator and connected through WIFI to the Raspberry Pi. The http server on the Raspberry Pi is continuously logging the incoming data.



2.2. Equipment

Tool	Further specification
Fleet-Monitor	
Raspberry Pi	With running WIFI hotspot and HTTP server software
J1939 Simulator	
CAN Cable	To connect simulator with Fleet-Monitor. Make sure the CAN termination is enabled on the Fleet-Monitor.
Ethernet Cable	To connect the Fleet-Monitor to the Raspberry Pi
USB A to B Cable	To power the Fleet-Monitor
USB C Cable	To power the Raspberry Pi
24V power supply	To power the J1939 simulator

3. Main Tests and Post-Tests

3.1. Test Variables

Duration at least 48h over the weekend

3.2. Main Test Procedure

Procedure	Comments	Status
Plug in Raspberry Pi	Make sure everything is running. Check with phone if WIFI hotspot is present.	Good
Start HTTP Server on Raspberry Pi	Check if no error code was thrown	Good
Plug in J1939 Simulator and connect it to Fleet-Monitor		Good
Plug in Fleet-Monitor	Check out LEDs and check if it receiving CAN messages and transmitting data to server	Good

3.3. Post-Test Procedure

Procedure	Comments	Status
Check if CAN messages are still received		Good
Check if Fleet-Monitor is still connected		Good
Check if HTTP server is still running		Not running
Read the data from the log file and see if first and last timestamp match	This is to see how long the device was logging data	Good
Plug in Ethernet cable	Check if Fleet-Monitor is still able to do hot-swap	Good

4. Results

4.1. Measured / Observed

What	Notes
First timestamp	1639069899 -> Thursday, December 9, 2021 17:11:39
Last timestamp	1639393915 -> Monday, December 13, 2021 11:18:55
Test Duration	90h

4.2. Test success

The device was powered on for an extended period of time and no anomalies were observed on the Fleet-Monitor side. It was signaling that the connection to the server was lost but was still connected through WIFI. After the restart of the HTTP server everything was still running normally. Hot swapping to Ethernet was also working fine.

4.3. Unexpected Observations

The HTTP server was not working anymore. We think this is because the log file got too big and caused an error. The server was logging for 90 hours and the log file exceeded 1GB.

4.4. Things to improve

This test shows that the Fleet-Monitor is running as expected and nothing needs to be improved on the monitor side. The HTTP server on the other hand should create new log files every hour or more in order to not crash after an extended period of time.

A.7 Financial Expenses

In table A.1 are the financial expenses listed of the project containing all development and production costs. Bills and other supporting documents can be found in the administrational repository (A.8) on GitHub. The total expenses of this project add up to a sum of `1'483.79` CHF.

Expense	Date	Value [CHF]
Adafruit	24.09.2021	271.50
DHL Fees	28.09.2021	39.40
Overleaf	30.09.2021	80.00
Kalitec Verbindungstechnik GmbH	12.10.2021	109.56
Aliexpress	13.10.2021	28.38
Post Zoll	20.10.2021	29.70
Thomann	21.10.2021	95.00
JLC-PCB	24.10.2021	213.80
JLC-PCB Extra Costs	25.10.2021	43.31
Digikey	25.10.2021	400.35
Digitec	25.10.2021	82.55
DHL Fees	04.11.2021	38.10
LCSC	19.11.2021	52.14

Table A.1: Total Financial Expenses

A.8 Data Archive

All created files and documents of this project are publicly available on GitHub. An institution called **SA-OST-2021** (`https://github.com/SA-OST-2021`) has been founded which contains repositories for each individual part of the project. A quick description of the repositories including the associated web link is listed below:

fleet-monitor-admin

Description: This repository contains all confidential information of the project.
URL: `https://github.com/SA-OST-2021/fleet-monitor-admin`
Type: Private

fleet-monitor-docs

Description: This repository contains all additional documentation of the project.
URL: `https://github.com/SA-OST-2021/fleet-monitor-docs`
Type: Public

fleet-monitor-requirements-specification

Description: This repository contains the Requirements Specification.
URL: `https://github.com/SA-OST-2021/fleet-monitor-requirements-specification`
Type: Public

fleet-monitor-report

Description: This repository contains this document.
URL: `https://github.com/SA-OST-2021/fleet-monitor-report`
Type: Public

fleet-monitor-hardware

Description: This repository contains hardware and mechanical related documents.
URL: `https://github.com/SA-OST-2021/fleet-monitor-hardware`
Type: Public

fleet-monitor-embedded

Description: This repository contains firmware source code written in C++.
URL: `https://github.com/SA-OST-2021/fleet-monitor-embedded`
Type: Public

fleet-monitor-configuration-tool

Description: This repository contains the filter configuration tool written in Python.
URL: `https://github.com/SA-OST-2021/fleet-monitor-configuration-tool`
Type: Public

fleet-monitor-network-tool

Description: This repository contains the network tool (server) written in Python.
URL: `https://github.com/SA-OST-2021/fleet-monitor-network-tool`
Type: Public

fleet-monitor-visualizer

Description: This repository contains the graphical data visualizer written in Python.
URL: `https://github.com/SA-OST-2021/fleet-monitor-visualizer`
Type: Public

fleet-monitor-rasp-image

Description: This repository contains an image of the Raspberry Pi SD-Card.
URL: `https://github.com/SA-OST-2021/fleet-monitor-rasp-image`
Type: Public

Bibliography

[1] A Brief Introduction to the SAE J1939 Protocol. https://copperhilltech.com/a-brief-introduction-to-the-sae-j1939-protocol. (accessed: 15.12.2021).

[2] A System Evaluation of CAN Transceivers. https://www.ti.com/lit/an/slla109a/slla109a.pdf. (accessed: 15.12.2021).

[3] Bus-FMS-Standard Description Version 04. http://www.fms-standard.com/Bus. (accessed: 16.12.2021).

[4] CAN Bus Protocol Tutorial. https://www.kvaser.com/can-protocol-tutorial. (accessed: 15.12.2021).

[5] ESP32-S2 Family Datasheet. https://www.espressif.com/sites/default/files/documentation/esp32-s2_datasheet_en.pdf. (accessed: 08.12.2021).

[6] Espressif Flash Download Tools. https://www.espressif.com/en/support/download/other-tools. (accessed: 17.12.2021).

[7] Neutrik etherCON NE8MX6 Product Page. https://www.neutrik.com/en/product/ne8mx6. (accessed: 14.12.2021).

[8] SAE J1939 Introduction and Overview. https://www.kvaser.com/about-can/higher-layer-protocols/j1939-introduction. (accessed: 15.12.2021).