ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

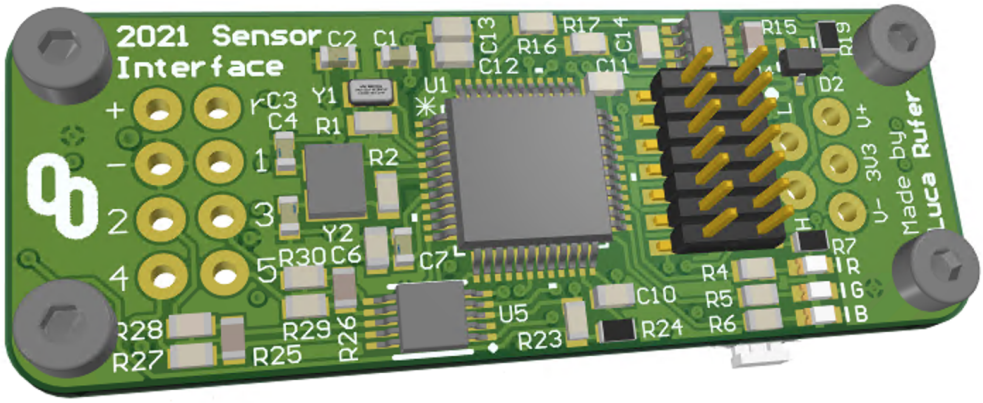
Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and
Electrical Engineering

Autumn Semester 2020

Design and Development of a Sensor Network Architecture for Hyperloop Vehicles

Semester Project



Luca Rufer
lrufer@student.ethz.ch

January 2021

Supervisor: Dr. Michele Magno, michele.magno@iis.ee.ethz.ch
Professor: Prof. Dr. L. Benini, lbenini@iis.ee.ethz.ch

Acknowledgements

I would like to thank the Integrated Systems Laboratory (IIS) and especially my supervisor Dr. Michele Magno for making this Semester Project possible and for providing the necessary materials to realize a functioning prototype.
I also want to thank the whole Swissloop team for supporting me with ideas, materials and tools during this project, especially the electrical lead Philip Wiese and the operations lead Yvan Bosshard. The many conversations we had allowed for great advancements in the project. Without them, this project would not have been possible.

Abstract

In this project, a sensor network architecture based on a linear bus topology using the Controller Area Network (CAN) protocol was developed to replace the current sensor topology that Swissloop used in past years. Swissloop builds Hyperloop vehicles based on Linear Induction Motors (LIMs), which require a wide range of sensors to control the vehicle end ensure safety. The sensor network is based on a layered approach and provides features like time synchronization of nodes, address management and sending commands. The system is highly configurable to allow for many different use-cases.

A special adapter Printed Circuit Board (PCB) was developed to connect sensors to the network. The adapter supports a wide range of digital interfaces and can also collect various analog signals.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Design and Development of a Sensor Network Architecture for Hyperloop Vehicles

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Rufer	Luca

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Unterentfelden, 14.01.2021	L. Rufer

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Contents

List of Figures

List of Tables

List of Acronyms

ADC . . . . . .Analog to Digital Converter

CAN . . . . . .Controller Area Network

CAN FD . . . .Controller Area Network with Flexible Data rate

CRC . . . . . .Cyclic Redundancy Check

DHCP . . . . . .Dynamic Host Configuration Protocol

EMI . . . . . .Electromagnetic Interference

GPIO . . . . . .General-Purpose Input/Output

HAL . . . . . .Hardware Abstraction Layer

I^2C . . . . . .Inter-Integrated Circuit

IC . . . . . .Integrated Circuit

IIS . . . . . .Integrated Systems Laboratory

JTAG . . . . . .Joint Test Action Group

LED . . . . . .Light-Emitting Diode

LIM . . . . . .Linear Induction Motor

MAC . . . . . .Medium Access Control

MCU . . . . . .Microcontroller Unit

List of Acronyms

OPAMP . . . . . Operational amplifier

PCB . . . . . Printed Circuit Board

RMS . . . . . Root Mean Square

SPI . . . . . Serial Peripheral Interface

STM . . . . . STMicroelectronics

SWD . . . . . Serial Wire Debug

UART . . . . . Universal asynchronous receiver-transmitter

USART . . . . . Universal synchronous and asynchronous receiver-transmitter

VCU . . . . . Vehicle Control Unit

Chapter 1

Introduction

1.1. Hyperloop

In 2013, SpaceX and Tesla CEO Elon Musk published a white-paper called *Hyperloop Alpha* [1], where they describe the Hyperloop concept as a "fifth mode [of transportation] after planes, trains, cars and boats" to transport passengers and goods at velocities of up to 1220 km/h. The main idea behind the Hyperloop concept is the use of pressurized vehicles, called *pods*, in a near vacuum tube. The pods use contact-less propulsion and levitation to keep friction as low as possible.
To further promote his idea, Musk organized an annual Hyperloop Competition that first took place in 2017 and last took place in 2019. There, student teams from around the world competed against each other to build a self-propelled prototype that reaches the highest possible velocity. For the competition, SpaceX built a 1.2 km long near-vacuum test track in Hawthorne, California.

1.2. Swissloop

Swissloop [2] is a student organization with students from ETH Zurich and other Swiss universities that participates in the Hyperloop Competition since the first competition. In the 2019 competition, the team reached the second place with a maximum speed of 252 km/h using their custom build linear induction motor and inverter. In 2020, the team built another Hyperloop Prototype called *Simona de Silvestro*, where the team used the knowledge gained in the previous year to improve the linear induction motor and inverter. However, due to the COVID-19 pandemic it was not possible for the team to participate in any competition.
Swissloop planes to participate in the European Hyperloop Week [3] in July 2021, where

simultaneous levitation and propulsion using a linear induction motor will be demonstrated with a new Hyperloop prototype.

Chapter 2

Background

This chapter aims to provide some more information on why this project was realized by covering previous Swissloop pods and their sensor network architecture in more detail. It is especially focused on the latest functional pod built by Swissloop in 2019 to 2020, called *Simona de Silvestro*. For this Hyperloop prototype, at first a 2.2 m long pod was planned, but due to the cancellation of the SpaceX Hyperloop Competition, a smaller 1.3 m variant was realized. The realized variant contains all the Systems from the initial design, but just in a smaller size or number. As the following section contains a mix of information from both designs, the initially designed pod and the realized smaller one will further be referenced as *Competition Pod* and *Prototype Pod*, respectively.

2.1. Sensor Network Architecture of previous Swissloop Pods

In the four Hyperloop prototypes built by Swissloop from 2016 to 2020 a *Star Network* topology was used, where all sensors of the pod are connected directly to a central star point called the Vehicle Control Unit (VCU). The 2020 *Competition Pod* was planned to have 26 sensors for vehicle control, excluding the sensors for the battery management system and the inverter. These sensors are distributed all over the pod as shown in Figure 2.1. The Prototype Pod with the reduced number of systems only has 17 vehicle sensors. The sensors for the Competition Pod have different signal types: 2 NPN[1] sensors, 10 thermistors, 10 sensors with current output, 3 sensors with RS485 communication and an encoder. The thermisor values are converted on an external Analog to Digital Converter (ADC) and sent to the VCU using the Inter-Integrated Circuit (I^2C) protocol,

[1] An NPN or PNP sensor basically acts as a switch

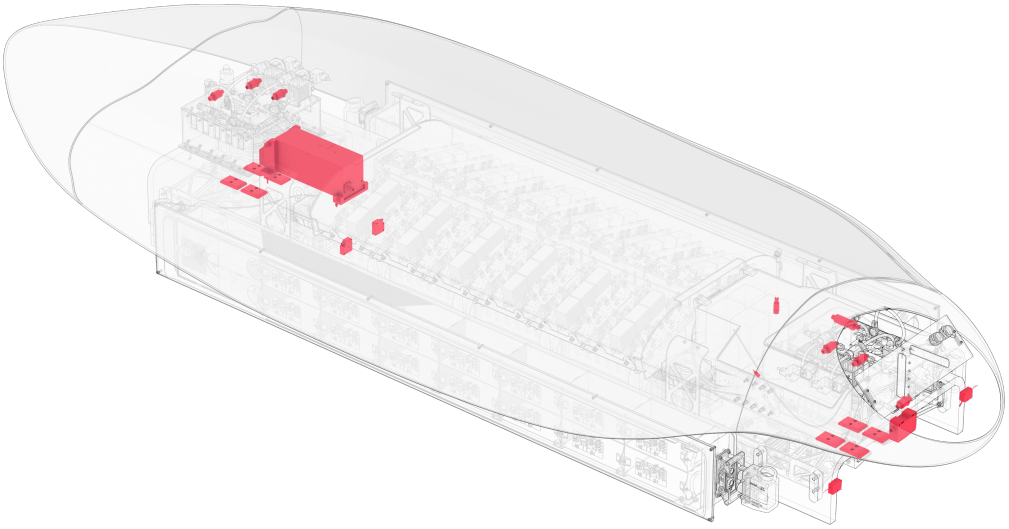


Figure 2.1.: Sensor locations of the Swissloop 2020 Competition Pod

while the signals from the other sensors are combined into a total of 11 cables connected directly to the VCU. This results in 19 m of sensor cable in the 2.2 m long *Competition Pod* and almost 10 m of sensor cable used in the 1.3 m long *Prototype Pod*. The cables and connectors weigh about 3 kg in the *Competition Pod* and 1.5 kg in the *Prototype Pod*. The cables and connectors are therefore a major contributor to the total weight and cost of the prototype. Additionally, the PCB for the VCU shown in Figure 2.2 and 2.3 is quiet large, as the connectors require a lot of space.

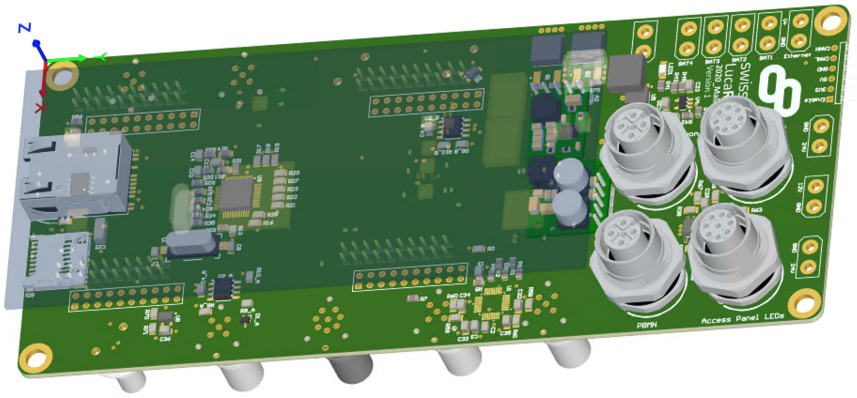


Figure 2.2.: VCU PCB of the 2020 Swissloop Pod, Top View.

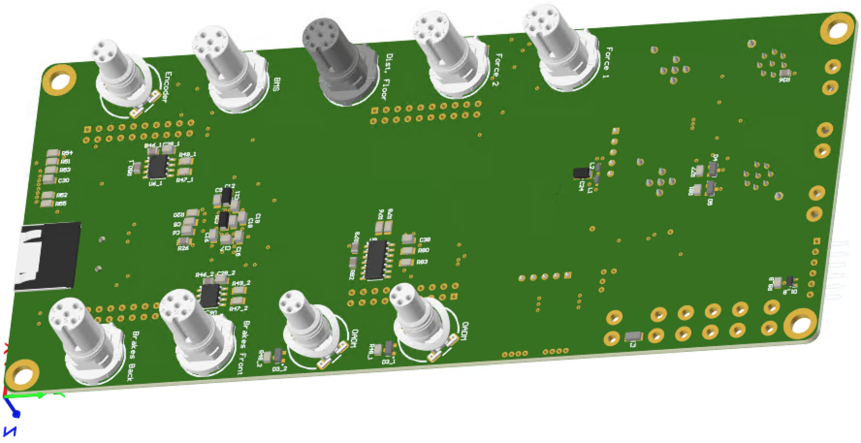


Figure 2.3.: VCU PCB of the 2020 Swissloop Pod, Bottom View.

Chapter 3

Network Architecture

The goal of this project is do design and test both software and hardware for a sensor network that improves on the flaws of the sensor architecture that previous Swissloop pods used. In this project, an architecture is developed that is configurable and compatible with a large number of sensors and cover all the needs of a Hyperloop pod sensor network. The resulting software and hardware will be integrated into the new Pod that Swissloop is currently developing.

3.1. Network Topology

The foundation of a network architecture is its topology. As mentioned in Chapter 2.1, previous Hyperloop pods built by Swissloop based their sensor networks on a star topology (Figure 3.1). While this topology is well suited and easy to implement for a small amount of sensors, it is not scalable for larger amounts of sensors and inefficient regarding size and weight, especially if multiple sensors are located close to each other, but every sensor has a separate connection to the VCU.
As Hyperloop pods are relatively long and narrow, a linear topology can reduce the amount of cable used to a single one from the back to the front with stubs to the sensor locations and therefore reduce weight and cost. Therefore, a linear bus topology was chosen for this project, as shown in Figure 3.2. A linear bus topology also has the advantage that sensors can be added to or removed from the system at any point, making the design process much easier. In order to be able to connect a variable amount of sensors with different outputs to a common bus, a interface to the bus is needed. This interface is responsible for reading data from the sensor and sending it to the communication bus in an appropriate form.

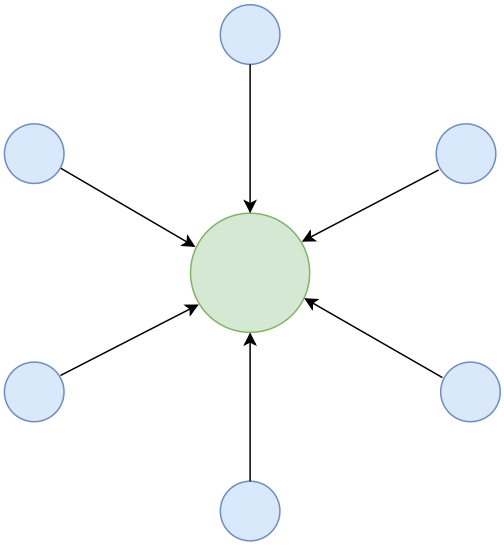


Figure 3.1.: Star Network Topology

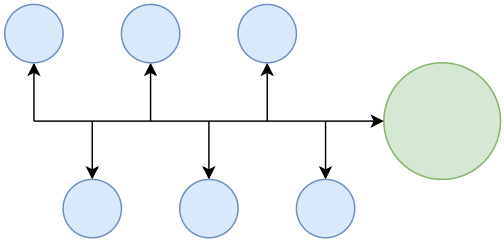


Figure 3.2.: Bus Network Topology

3.2. Communication Protocol

The CAN Protocol was chosen as the communication protocol, as it is a multi-master priority-based bus protocol with non-destructive content-based arbitration [4]. It was primarily developed for automotive applications. The CAN protocol has many useful properties, as listed below:

- Multi-Master: The network does not require a master to decide which node is allowed to send a message frame. This allows a high degree of freedom to be given to the sensor nodes and allows them to send their data autonomously without the need for the network master to poll the sensor nodes for new data. Furthermore, this also reduces the latency of the data sent by the sensor nodes.
- Medium Access Control (MAC) Realized in Hardware: Bus access control does not have to be implemented in software, which makes the implementation much easier and more resistant to errors.
- Content-based Arbitration: CAN frames have an identifier at the beginning of each frame, which is used to determine the priority of the frame. If multiple nodes on the bus try to send a frame simultaneously, the node sending the frame with lower priority stops sending and the message with the higher priority can continue to send its frame without restarting the transmission.
- Integrated Cyclic Redundancy Check (CRC): The integrated CRC makes the protocol more resistant to single bit flip errors or error bursts cased by e.g. Electromagnetic Interference (EMI).

- Integrated Error Handling: a large variety of errors within a transmission are detected by hardware and frames can be automatically re-transmitted in case of an error.

The new version of CAN, Controller Area Network with Flexible Data rate (CAN FD), allows to change the bitrate in a certain part of the message called data phase, which contains the data and the CRC. Additionally, a message frame can contain up to 64 Byte instead of up to 8 Bytes per frame, and the CRC was improved. Many modern Microcontroller Units (MCUs) already have an integrated CAN FD peripheral.

3.3. Software Architecture

To distinguish the devices connected to the sensor bus, roles are used. Every node on the bus can have any combination of following roles:

- Network Master: The network master is responsible for the address assignment and configuration of the other nodes on the bus. There is exactly one network master per sensor bus.
- Sensor Node: A sensor node is a node that collects data from a sensor connected to it and sends this data to the bus. A network normally has multiple sensors. The number of sensor is limited to 62 for reasons explained further below.
- Listener: A listener collects the data sent by the sensor nodes. It does not send any data itself. A network can have multiple listeners.

The role of the listener allows other systems (like an inverter) of a pod to be connected directly to the sensor bus. These systems can access the data directly and do not have to wait for the data to be relayed by the network master, which massively reduces the latency of the data. For most use-cases, the VCU acts as the network master and a listener in order to control the network and collect the data from the sensor nodes. Figure 3.3 shows an example of a sensor network with a VCU on the right acting as the network master (green) and listener (gray), multiple listeners (gray) and sensors (blue) with a bus adapter (yellow). Some sensor nodes also have the listener role to collect data from other sensor nodes. In such cases, the adapter is indicated as yellow and gray. This can be useful if a sensor produces data depending on other sensors.

A CAN frame can have either an 11-bit normal or a 29-bit extended identifier. The sensor network only uses the shorter 11-bit identifier as it suffices to implement the necessary functionalities. The division of the 11-bit identifier is shown in Figure 3.4. The lower 6 bits are used to determine the address of the device and the upper 5 bits are used to identify the type of the message.
Only sensor nodes have an address whereas the network master and the listeners have no specific addresses. A listener does not have an address because it cannot send any

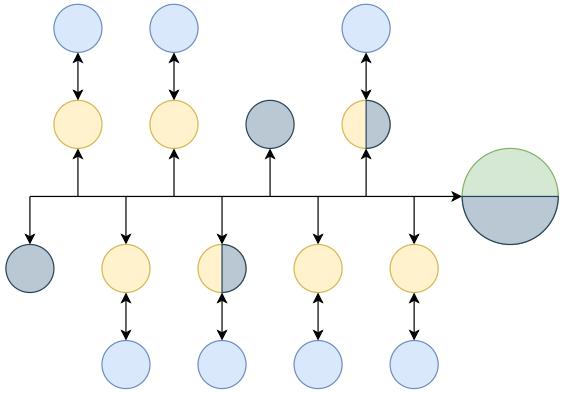


Figure 3.3.: Example topology of the sensor network with roles

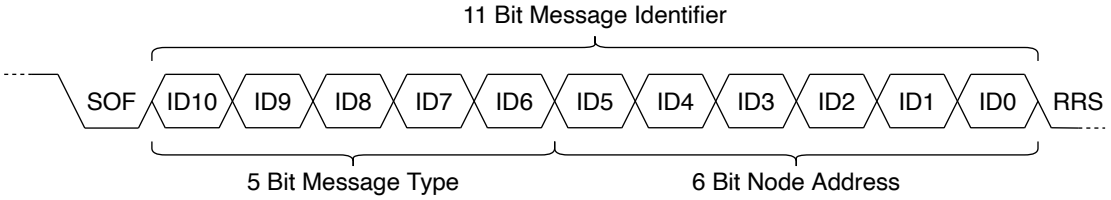


Figure 3.4.: Division of the CAN Message Identifier Field

data on the bus. If a master wants to communicate with a specific sensor node, it sends a message with the address of the sensor node. If the master wants to send a message to all nodes, it sends a message using the *broadcast address* 0x00. Another special address is the *unassigned address* 0x3F, which the sensors nodes use that have not been assigned an address yet. The process of address assignment is elaborated in Section 4.1.1 under Address Management.
The 5 bits used for the type of the message allow for 32 different message types. An overview of the existing message types is shown in Figure 3.5. The order of the message type ensures that more important message types have a lower type number and therefore a higher priority. This guarantees that important messages cannot be blocked or delayed by less important messages. The message type also dominates the address in regard to priority, as the type occupies the higher 5 bits of the CAN identifier.

3.4. Sensor Support

The goal of the sensor network is to support a large variety of sensor signals and at least the signal types of the sensors used in previous Swissloop pods described in Section 2.1. The signal types shown in Figure 3.6 cover common sensor outputs and these will be supported by the network adapter.

ID	Name	Byte								Sendable by	
		0	1	2	3	4	5	6	7	Master	Node
0x00	*Reserved*										
0x01	Init/reset	Flags								X	
0x02	*Reserved*										
0x03	Time Sync	Timestamp								X	
0x04	*Reserved*										
0x05	Error	Flags								X	X
0x06	Address Assignment	Flags	Node Addr	Node ID						X	
0x07	Address Request	Flags		Node ID							X
0x08	Command callback	Flags		Command ID							X
0x09	Write callback	Flags		Address			Data				X
0x0A	Read request callback	Flags		Address							X
0x0B	*Reserved*										
0x0C	Command Ack	Flags		Command ID							X
0x0D	Write request Ack	Flags		Address							X
0x0E	Read request Ack	Flags		Address							X
0x0F	*Reserved*										
0x10	Command request	Flags		Command ID			*Optional Byte-wise Data*			X	
0x11	Write request	Flags		Address			Data			X	
0x12	Read request	Flags		Address						X	
0x13	*Reserved*										
0x14	*Reserved*										
0x15	*Reserved*										
0x16	Sensor data	*Optional Data*									X
0x17	*Reserved*										
0x18	*Reserved*										
0x19	*Reserved*										
0x1A	*Reserved*										
0x1B	*Reserved*										
0x1C	*Reserved*										
0x1D	*Reserved*										
0x1E	*Reserved*										
0x1F	*Reserved*										

Figure 3.5.: Message Types

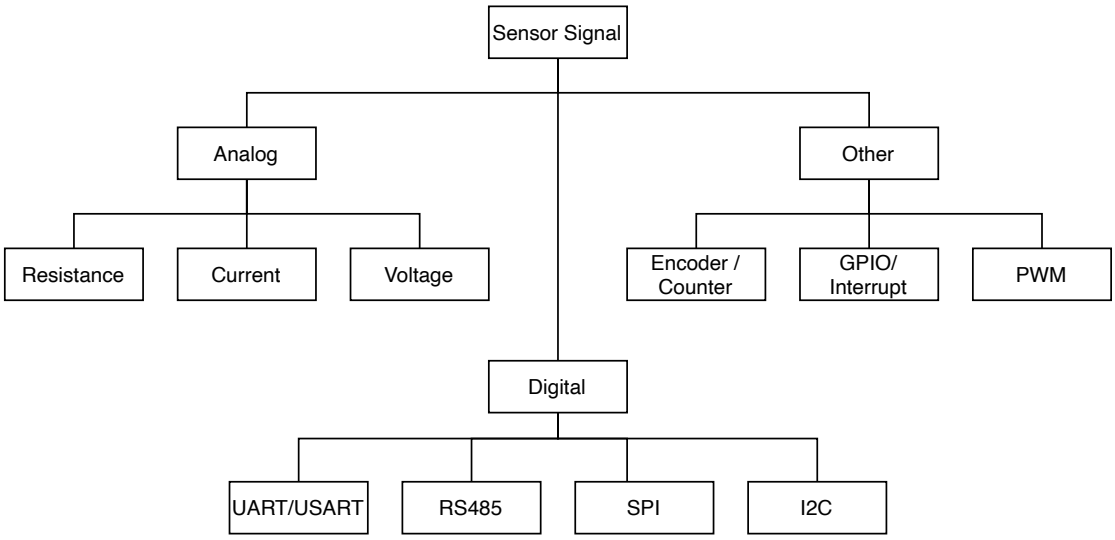


Figure 3.6.: Sensor signal types

Chapter 4

Design Implementation and Results

This chapter covers the actual implementation of the sensor network. It covers both the implementation in software and the hardware implementation of the adapter PCB. At the end of this chapter, a few performance metrics resulting from the implementation are shown.

4.1. Software implementation

The software stack is divided into multiple layers with distinct functionality. The software stack differs depending on the role of the network node. The software stack for both the network master and the listeners is shown in Figure 4.1 and the software stack for the sensor node is shown in Figure 4.2. The lowest layer, the physical layer, is largely defined by the CAN standard [4]. The layer above the physical layer, the data link layer, is completely defined by the CAN standard and implemented in hardware in a CAN controller. The lowest layer implemented in software a Hardware Abstraction Layer (HAL). As described in Subsection 4.2.1, a STMicroelectronics (STM) MCU was used and therefore the HAL provided by the chip manufacturer was used for this layer.

4.1.1. Network Layer

The network layer is built on top of the HAL. Its main task is to pack and unpack the CAN messages and configuring the CAN transmissions. Additionally, it provides a time base for all layers above and it is responsible for time synchronization and address management, which will be described in the following sections.

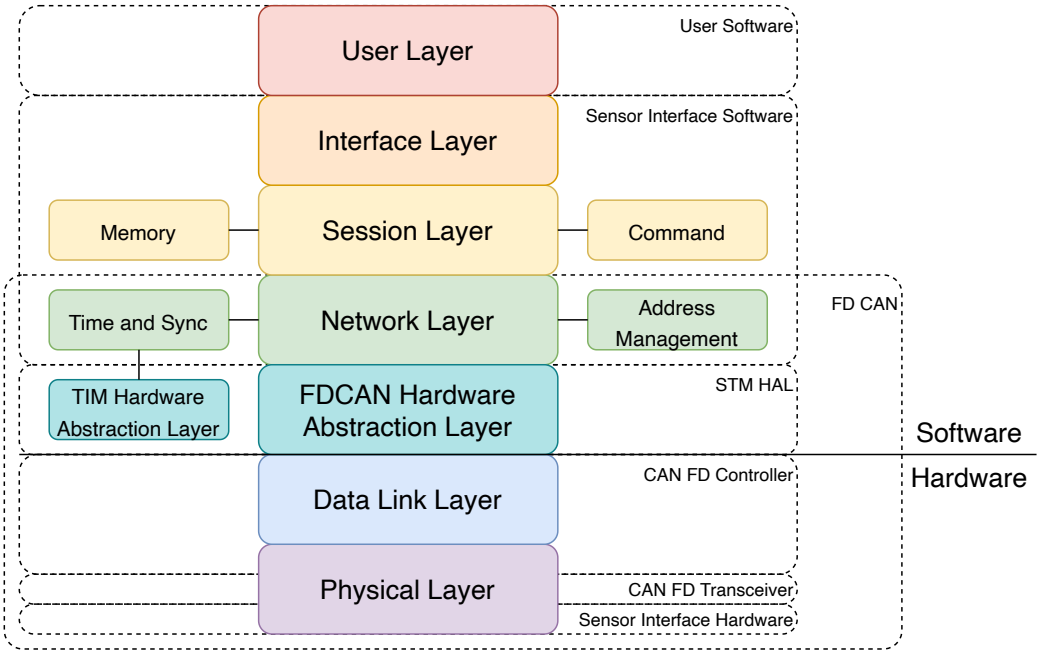


Figure 4.1.: Software stack for network master and listeners

Time and synchronization

For the network layer and all layers above a local system time is established. System time is a 64 Bit value, which holds the time in microseconds since system startup. The lower 16 Bits of the timestamp taken from a hardware timer, and the upper 48 bits are taken from an overflow counter of said counter.

The sensor nodes have a high degree of autonomy and can perform actions like sampling the sensor regularly by themselves. This, however, makes the sensor nodes susceptible to time drift, caused by imperfections of oscillators. Multiple sensor nodes can only perform action or sample the sensors simultaneously if they have the same system time, which requires time synchronization. Additionally, some sensors like event triggered sensors require the global system time as part of the data they produce, which also requires a network-wide time synchronization.

While many synchronization algorithms like [5] and [6] suggest algorithms that correct the time drift by jumping in time, this is not really a viable option for this system, as the system uses the capture-compare channels of the system time timer. Instead, an algorithm that adjusts the prescaler of the system timer was implemented to speed up or slow down time for nodes that are behind or ahead of the master's system time. The adjustment of the prescaler changes the slope s of the difference of local times $\Delta t = t_{slave} - t_{master}$,

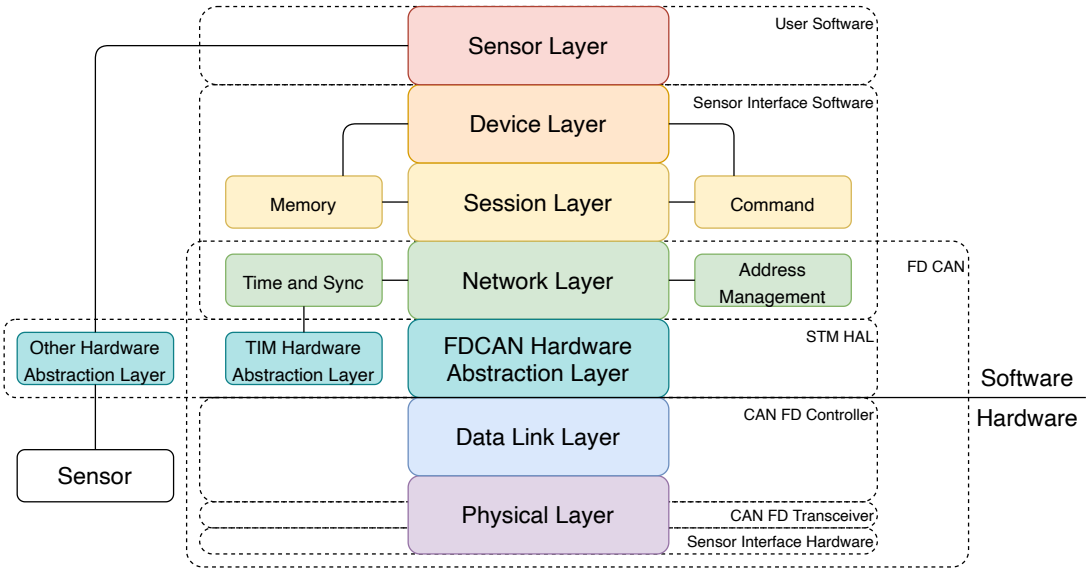


Figure 4.2.: Software stack for sensor nodes

as seen in Figure 4.3. The slope s_j is given by Equation 4.1, where d is the time drift, P_0 is the default prescaler value and P_j is the prescaler value during time sync period j. The time difference $\Delta t_{j+1}(t)$ can be calculated using Equation 4.2, which can be derived from Figure 4.3.

$$s_j = d + \frac{P_0 - P_j}{P_0} \tag{4.1}$$

$$\Delta t_{j+1}(t) = \Delta t_{sync,j} + s_{j+1} \cdot (t - (t_j + t_{sync})) \quad \text{for} \quad t_j + t_{sync} \leq t \leq t_{j+1} + t_{sync} \tag{4.2}$$

The network master also acts as the time master, which periodically sends a reference message. Using the message timestamp functionality of the CAN peripheral of the MCU [7, p. 1900], the send timestamp of the reference message is stored by the time master as t_j. The time slaves store the receive timestamp of reference message using the same mechanism as $t_{j,\alpha}$. As soon as the time master finished sending the reference message, it sends a sync message containing the timestamp t_j of the last sent reference message. At local slave time $t_{j,\beta}$ right after the slave received the sync message, the slave executes Algorithm 1 to update the timer prescaler. The algorithm aims to keep the Root Mean Square (RMS) time difference (see Equation 4.3) as low as possible. The RMS time difference is minimal for the P_{j+1} derived in Equation 4.4. As the exact value of the

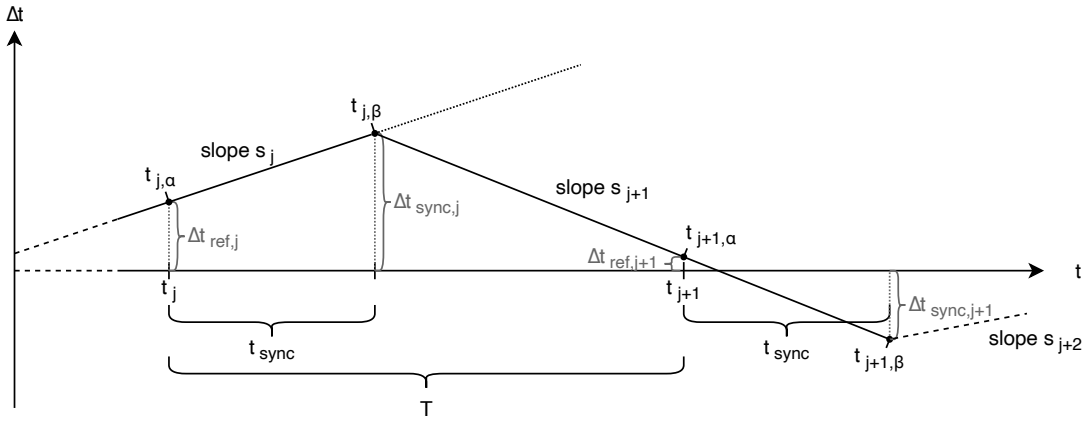


Figure 4.3.: Time drift and drift compensation

oscillator drift d is not known, it has to be estimated using Equation 4.5, which can be derived from Figure 4.3.

$$E^2_{rms,j+1} = \frac{1}{T}\int_{t_j+t_{sync}}^{t_{j+1}+t_{sync}} (\Delta t_{j+1}(t))^2 dt = \Delta t^2_{sync,j} + \Delta t_{proc,j} \cdot s_{j+1} \cdot T + \frac{1}{3} \cdot s^2_{j+1} \cdot T^2 \quad (4.3)$$

$$\frac{dE^2_{rms,j+1}}{dP_{j+1}} \stackrel{!}{=} 0 \Leftrightarrow P_{j+1} = \frac{3}{2} \cdot P_0 \cdot \frac{\Delta t_{sync,j}}{T} + P_0 \cdot (1+d) \quad (4.4)$$

$$P_0 \cdot (1+d) = \frac{(\Delta t_{sync,j} - \Delta t_{sync,j-1})}{T} \cdot P_0 + P_j \quad (4.5)$$

Address Management

As the address of a node reflects its priority to send messages on the bus, the addresses have to be manageable and configurable by the network master. Therefore, an address assignment and management process was implemented. Initially, a process similar to the Dynamic Host Configuration Protocol (DHCP) was planned. However, the process of nodes actively requesting addresses was deemed as unnecessarily complicated, as this would require the implementation of an additional MAC: All sensor nodes would send their request simultaneously upon startup, resulting on collisions on the bus because all nodes send with the same CAN identifier. Because in a normal application of this sensor network all connected sensors are known in advance anyway, it was easier to implement a process where nodes are assigned addresses by the network master without sending a request first.

Algorithm 1: Time synchronization of time slaves

input: Timestamp t_j when the reference message was sent, in master time
input: Timestamp $t_{j,\alpha}$ when the reference message was received, in slave time
input: Timestamp $t_{j,\beta}$ when the sync message was received, in slave time

$\Delta t_{ref,j} \leftarrow t_{j,\alpha} - t_j$;
`/* if the time difference is too large, jump in time */`
if $|\Delta t_{ref,j}| > \Delta t_{ref,max}$ **then**
  $current\_time \leftarrow t_j + t_{j,\beta} - t_{j,\alpha}$;
  $P_{j+1} \leftarrow P_0$; `// Reset prescaler to default value`
  $\Delta t_{sync,j} \leftarrow 0$; `// No time drift after time jump`
  $P_{d,est} \leftarrow P_0$; `// Reset to default value (d=0)`
else
  $\Delta t_{sync,j} \leftarrow \frac{t_{j,\beta} - t_{j-1,\beta}}{t_{j,\alpha} - t_{j-1,\beta}} \cdot (\Delta t_{ref,j} - \Delta t_{sync,j-1}) + \Delta t_{sync,j-1}$;
  $P_{d,est,new} \leftarrow \frac{(\Delta t_{sync,j} - \Delta t_{sync,j-1})}{T} \cdot P_0 + P_j$; `// Eq. 4.5`
  $P_{d,est} \leftarrow \alpha \cdot P_{d,est} + (1 - \alpha) \cdot P_{d,est,new}$; `// Filter` $P_{d,est}$
  $P_{j+1} \leftarrow \frac{3}{2} \cdot P_0 \cdot \frac{\Delta t_{sync,j}}{T} + P_{d,est}$; `// Eq. 4.4`
end
`UpdatePrescaler(`P_{j+1}`)`

At the beginning of this process, all sensor nodes use the *unassigned* bus address 0x3F. Every sensor node has a unique 32-bit identifier, which is used to differentiate between the nodes. Using a predefined list of address - node ID pairs, the network master assigns the addresses one by one using a dedicated address assignment message. The message contains the node ID and the CAN address assigned to it. If a sensor nodes gets an address assignment message containing its node it, it buffers the CAN address. However, the sensor node still continues to communicate with its old address. When the network master finished sending all addresses, it sends an init message with a special flag set indicating the nodes to update their network layer with the newly assigned address. The state diagrams for the network master and sensor node can be seen in Figure 4.4 and Figure 4.5, respectively.

4.1.2. Session Layer

The main task of the session layer is handling communication sequences that consist of multiple messages. These primarily are the write, read and command request. They will be further explained in the subsections below. All three requests consist of a request by the master, followed by an acknowledgement from the sensor node that indicates if the request is granted. If the request was granted, the sensor node also sends a callback message after the request was performed, e.g. the data was written/read for a write/read request or the command has finished execution in case of a command request.

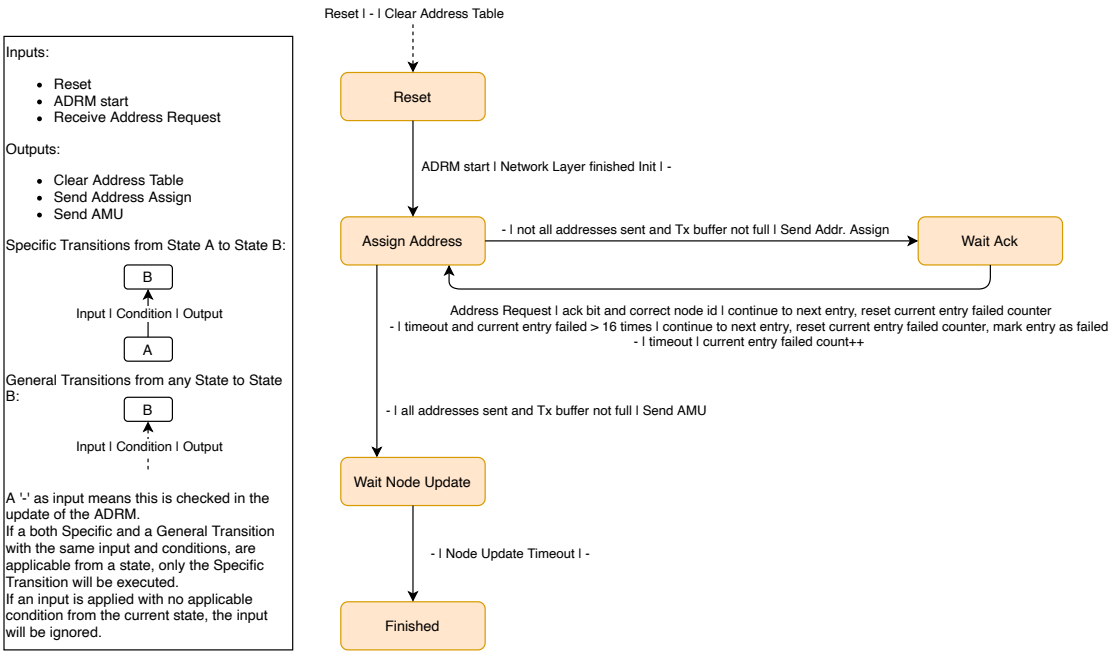


Figure 4.4.: Address Management: State machine for network master

Memory

The sensor interface memory supports up to 256 memory sections with up to 256 32-bit words each. The sensor interface has four memory sections implemented:

- Error Code Section: This section holds which error flags are set for the whole sensor interface software. It also holds masks to filter which error flags will be reported to the other nodes on the network if they change. Merging all these error codes into a single structure allows for easy read and clear access by the master.
- Hardware Section: This section holds the hardware definitions for the sensor node interface. It depends on the connected device and is used to indicate the presence or absence of certain components or the values of components like resistors or capacitors that can be varied.
- General SI Section: This section holds the general sensor interface configuration, like which sensor is connected and at which period it shall be sampled.
- Sensor Specific Section: This section can be used to pass configurations or information directly to the selected sensor.

While all of the above sections exist for the sensor node role, only the error code section exists for the network master and listener. The network master can read or write a 32-bit

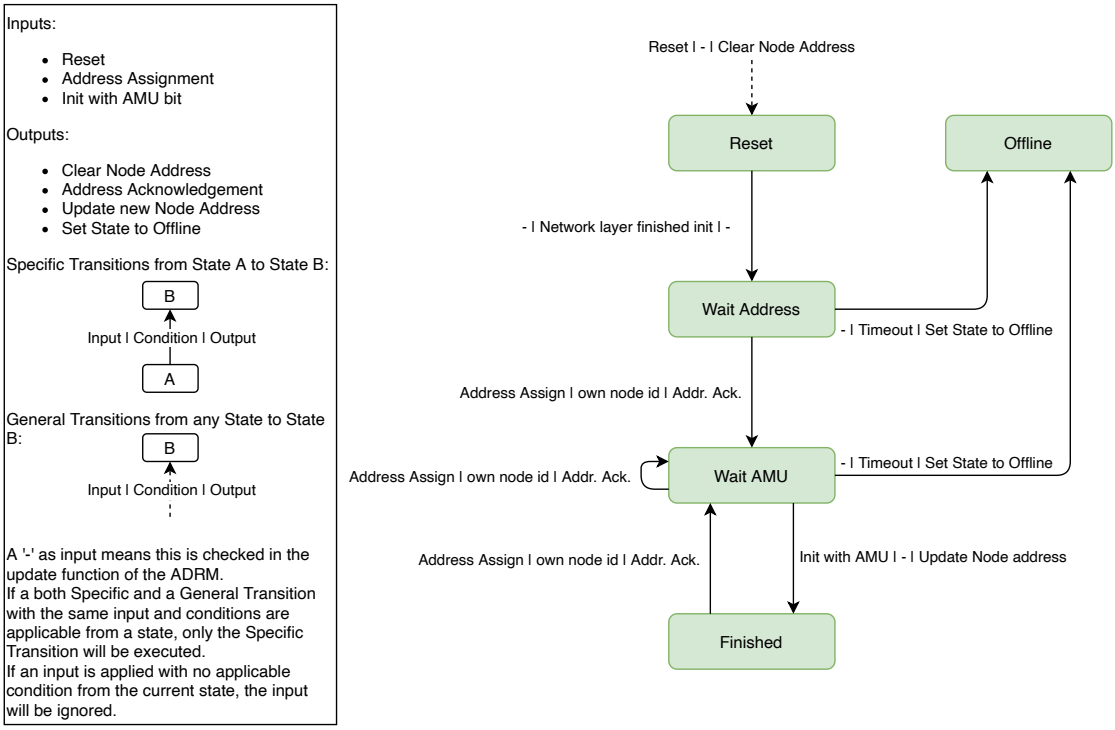


Figure 4.5.: Address Management: State machine for sensor node

word from or to the sensor node memory by using a read or write request. The sensor node can write-protect each 32-bit to prevent the master from overwriting data.

Command

The network master can let the sensor nodes execute commands using a command request. Commands can be directed either to the sensor interface software in general, to the interface device or directly to the sensor layer. Commands for the latter two depend on the device and the sensor, respectively. A variable amount of data between 0 and 60 Bytes can be sent with the command to give additional information for the command which has to be executed. Examples for commands are *Start Data Collection*, which is addressed to the interface software in general, or *Enable Sensor Power*, on which a device supporting this command would enable power to the sensor.

4.1.3. User Interface

The user interface is supposed to make the interaction with the sensor interface easier. It is especially focused on network startup, configuring sensors as well as starting and stopping sensor data collection. However, it does not offer all the functionality that the

session layer offers, which means that the user has to interface directly with the session layer for more in-depth usage of the sensor interface software.

4.1.4. Device Layer

The device layer abstracts the sensor interface hardware. It provides the hardware initialization functions for the peripherals, configures and redirects interrupts where needed and abstracts the hardware to make the programming of the sensors easier.

4.1.5. Sensor Layer

Programming the actual interface with the sensor happens in the sensor layer. For every sensor that can be connected, a separate header-source pair is written that has to implement at least the functions listed below:

- *init*: In the init function, the communication with the sensor is initialized. This varies from sensor to sensor and can be actions like initializing a communication peripheral or the ADC.
- *update*: This function is called regularly upon which regular task like state or health checks can be executed.
- *get data*: When this function is called, a data item will be collected from the sensor. This function is called either periodically by the device if a sampling period is set and/or upon a command from the master. When the data item is collected from the sensor, it is passed to a predefined *send data* function implemented by the device, which sends the collected data to the network bus.
- *deinit*: A function used to de-initialize the sensor, e.g. shut down the sensor, release all occupied peripherals and memory.
- *execute command*: Execute a command requested by the master. What the command does largely depends on the sensor. The command is further explained in Section 4.1.2.

4.2. Hardware Implementation

As mentioned at the beginning of this chapter, the sensors have to be connected to the network using some kind of adapter. This adapter was realized as a custom made PCB. For any other devices on the bus that are not sensor nodes, no specific hardware was realized. The only hardware requirement for these is the availability of a CAN FD interface. The main design parameters of the adapter PCB were to keep it as small and light as possible, while still being compatible with a large range of sensors. The PCB is shown

in Figure 4.6 and 4.7 with its key functional blocks and components encircled. The size of the PCB is 45 mm by 17 mm, the schematics of the PCB can be found in the appendix.

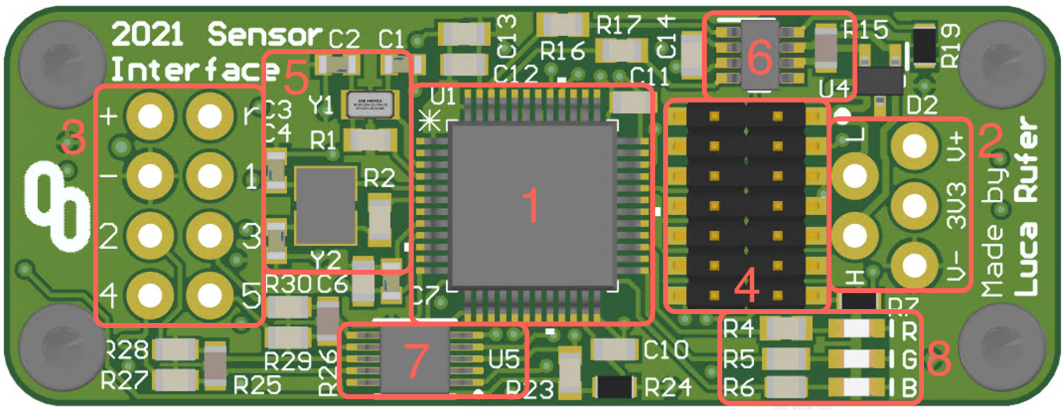


Figure 4.6.: Top view of the Sensor Interface PCB with key components

4.2.1. PCB Components

In the center of the top side, the MCU (Label 1) is located. The decision to use a MCU from STM was made early on in the project, as they produce a large variety of MCUs and many of those feature a CAN FD peripheral. As the MCU on the sensor interface PCB does not have to be a high performance MCU, the selection was focused on the STM32L5 product line, a low-power MCUs line with an integrated CAN FD periperal. Finally the STM32L552 MCU was chosen due to the availability of a development board [8].
The PCB has three connectors: The first one is the bus connector (Label 2), which has the CAN-High and CAN-Low signal lines (inscription H and L, respectively), a ground connection (V-), a 3.3 V supply connection to power the Integrated Circuits (ICs) on the PCB and a sensor voltage connection (V+). As the sensors on the bus might require different supply voltage levels and therefore multiple different supply voltage levels are needed on the bus, no fixed connector was used for the bus connector. For the connections, the wires of the bus cables can be soldered directly into the pad holes of the respective connector. To provide more stability, the cable can be fixed using a 3D printed part that is screwed to the PCB with the M2 screw holes in the corners. The pads of the connector have a 100 mil pitch, allowing standard pin header to be directly soldered onto the PCB for testing and debugging purposes.
The second connector is the sensor connector (Label 3). To allow many different configurations of sensors to be connected, there is no fixed connector used and the same

considerations as for the bus connector apply. The connector has a pad for ground (indicated with -), one for sensor supply (indicated with +) and one for a reference voltage (indicated with r). For communication with the sensor, five signal pads (indicated with 1 to 5) are used. Each of these five signals is filtered and passed to the MCU. The filters are discussed further below.
The third and last connector is a 14 pin 50 mil pitch connector used to program and debug the MCU. The pinout is compatible to the STLink-V3 [9] and supports Joint Test Action Group (JTAG) and Serial Wire Debug (SWD) interface for programming and debugging. It also features a virtual COM port that can be used for debugging using Universal asynchronous receiver-transmitter (UART).
Additional components on the top side worth mentioning are the external oscillators (Label 5), the CAN transceiver (Label 6) including optional termination resistor, a optional full-duplex RS485 transceiver (Label 7) and three Light-Emitting Diodes (LEDs) (Label 8) that can be programmed in the sensor layer.

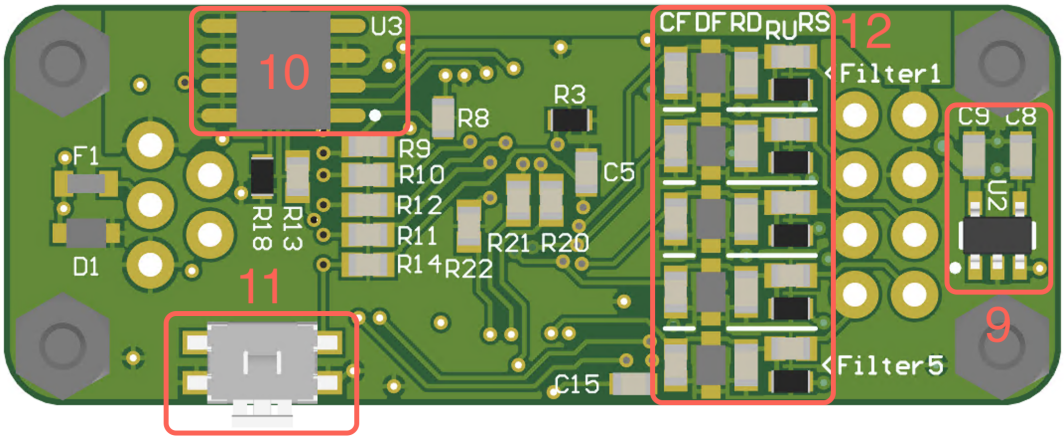


Figure 4.7.: Bottom view of the Sensor Interface PCB with key components

The bottom side of the adapter PCB has a precision 2.5 V voltage reference IC (Label 9) for the internal ADC of the MCU and can be used by sensors like thermistors as an accurate voltage reference.
The largest IC on the bottom side is a high side power switch (Label 10) that can be used to enable and disable the sensor supply power, which makes it possible to shut down the sensor by software when it is not needed. This also allows to reduce the power consumption of the system.
The PCB also features a reset button (Label 11) to reset the MCU. It is only intended to be used during debugging and is removed when the PCB is integrated into the vehicle.

The signal filters (Label 12) are used to filter the signals from the sensors and provide pull-up or pull-down on signal lines if necessary. Which components are present and what value these resistors and capacitors have depends on the signal type and sensor.

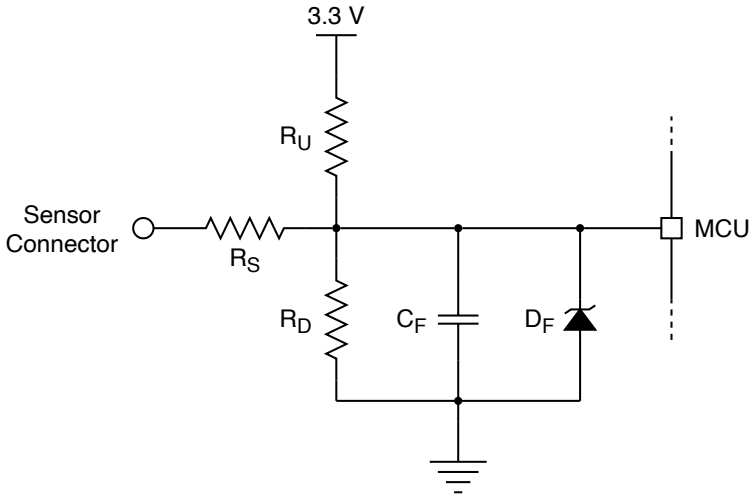


Figure 4.8.: Sensor signal filter circuit

Table 4.1.: Component example values for hardware filter

Sensor signal	R_S	R_U	R_D	C_F	Resulting filter output
Analog Current, 4...20 mA	22 Ω	NC	120 Ω	4.3 µF	0.48...2.4 V, f_g = 2 kHz
Analog Voltage, 0...5 V	500 Ω	NC	500 Ω	636 nF	0...2.5 V, f_g = 1 kHz
Thermistor, e.g. PT100 with R_0 = 1 kΩ	10 Ω	NC	1 kΩ	16 µF	Temp. dependant Voltage, f_g = 1 kHz
SPI MISO	22 Ω	10 kΩ	NC	NC	Pull-up and reduction of signal ringing
RS485, any of Tx/Rx +/-	NC	NC	NC	NC	Filter not used, connect R27 to R30 instead
No filter	0 Ω	NC	NC	NC	Filter bypassed

The Zener diodes are used to protect the MCU from voltage spikes. Each of the five signal pads has its own filter. A circuit diagram of the filter can be seen in Figure 4.8. While this system allows for a wide range of possibilities, it has the downside that - once the filter components are soldered - it can only be used with a specific type of sensor. However, as these boards are designed to be installed in a vehicle permanently, this is not a big concern. Some examples for filter values are given in Table 4.1. If R_U is not used, the corner frequency f_g of a filter is given by Equation 4.6.

$$f_g = \frac{1}{2\pi \cdot R_S \cdot C_F} \cdot (1 + \frac{R_S}{R_D}) \tag{4.6}$$

4.2.2. Sensor Compatibility

The sensor interface PCB is compatible with a large range of sensors. The hardware was implemented in a way that at least the sensor types listed in Figure 3.6 are supported. The implementation made even more sensors types possible than listed in Figure 3.6. In some cases, multiple sensor types can be combined with each other. The possible sensor interfaces are listed below:

- Digital communication interfaces for Serial Peripheral Interface (SPI) in master and slave mode, Universal synchronous and asynchronous receiver-transmitter (USART), I^2C and RS485.
- One Comparator and two Operational amplifiers (OPAMPs) with internal ADC connection.
- Four input capture or output compare channels to system timer with microsecond resolution or two input capture or output compare channels to timers for counter or encoder mode.
- ADC input, external interrupt or General-Purpose Input/Output (GPIO) functionality for each of the five signal pads.

4.3. Performance

The performance of the sensor network was evaluated using manufactured versions of the adapter PCBs discussed above and shown in Figure 4.9. Several example sensors were written to acquire the data shown in this chapter and test the functionality of the sensor network and explore its limits. The performance of the network depends on various factors like the number of sensors on the bus, the update frequency of each of the sensors and sensor data size for each sensor. The performance parameters calculated in this section are valid when the sensors on the network are collecting data, the nodes are not being configured (e.g. no write, read and command requests) and no errors occur. Also, the bus load caused by the time sync is neglected.

4.3.1. Message duration and bus load

The worst-case duration[1] that a message occupies the sensor bus can be calculated using Equation 4.7 [4], assuming a CAN FD format with standard identifier without an error occurring during transmission. In the Equation, t_{arb} is the arbitration bit time (1 Mbps by default), t_{data} is the data bit time (5 Mbps by default) and n is the number of data bytes in the message. A number of examples with different data length and samples

[1] Depending on the CAN identifier and the message data, a different number of stuff bits are inserted into the message, causing messages with the same amount of data to have different lengths

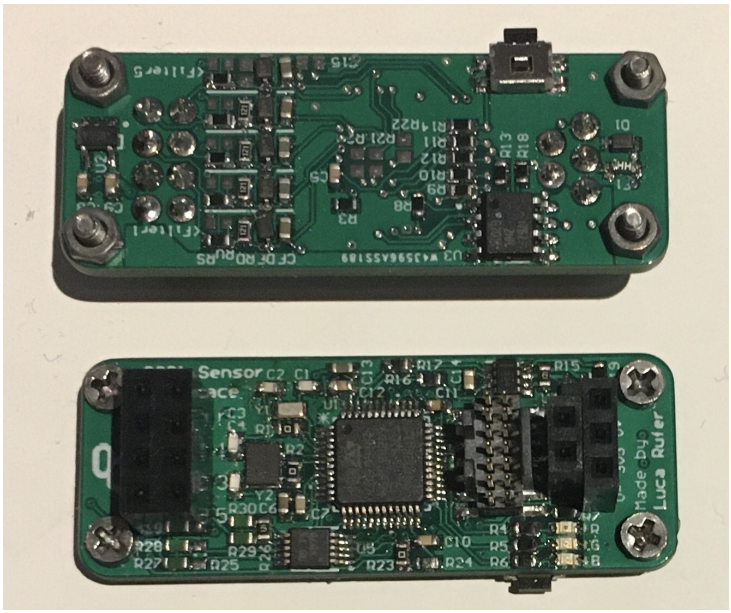


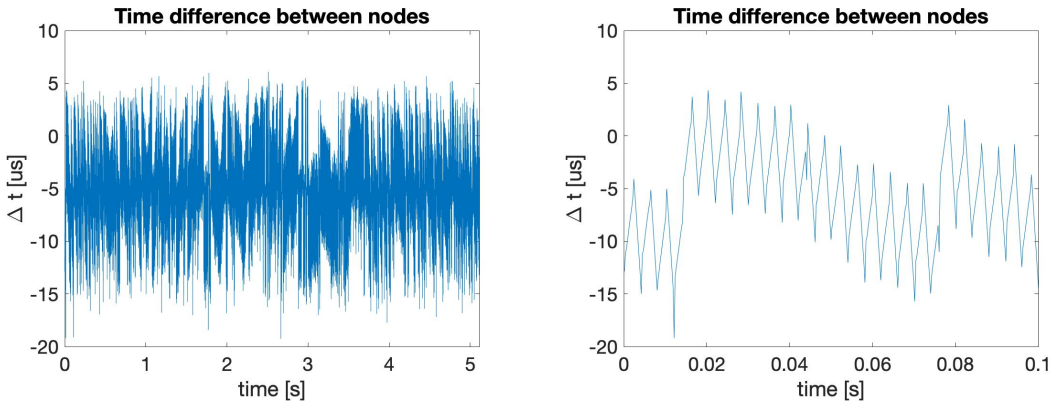
Figure 4.9.: Fully Assembled Prototypes of the Adapter PCB

per second is listed in Table 4.2. The table shows the resulting bus load and package utilization. Sensors that produce a low number of Bytes per data item have a very low packet utilization and are therefore less efficient. If the bus load it too high, starvation can occur for sensors with a low priority. During testing, the network was able to handle a theoretical bus load above 97 % for an extended period of time without loosing any data.

$$t_{msg} = 32 \cdot t_{arb} + \begin{cases} (33 + 10 \cdot n) \cdot t_{data}, & \text{if } n < 20 \\ (38 + 10 \cdot n) \cdot t_{data}, & \text{otherwise} \end{cases} \tag{4.7}$$

Table 4.2.: Bus load examples

Number of Data Bytes	Data Samples per Second	Bus Load	Packet Utilization
2	20’000	85.2%	7.5%
4	20’000	93.2%	13.7%
8	17’500	95.6%	23.4%
16	12’000	84.7%	36.3%
20	12’000	95.5%	40.2%
32	8’000	92.9%	49.4%
64	5’000	83.8%	61.1%



(a) five seconds test

(b) first 100 ms of the test

Figure 4.10.: Time difference between two nodes with time synchronization.

4.3.2. Latency

The latency of a data item, e.g. the time difference between the time instant where a sensor node starts collecting a data item and the time instant where a listener has the data available, depends on the message size (and thus the data size), the time it takes the sensor to collect the data item and on the bus load. Under optimal conditions where the data is immediately available to the sensor and the CAN message can be sent instantly, the average latency can be as low as 76.2 µs for a data size of 4 Byte. The average message duration in this case is 33.2 µs and it therefore takes on average 43 µs for the data to traverse the software layers on the sensor node and the listener.

4.3.3. Time Synchronization

The time synchronization algorithm that was described in Section 4.1.1 was also tested and its performance evaluated. When sending the time synchronization messages with a period of 2 ms, a RMS time drift of 6.46 µs was achieved. The maximum time difference between the nodes in this case was measured to be 19.26 µs. The relative oscillator drift of the nodes was measured to be 0.4 %. Figure 4.10a shows the measured difference between the nodes and Figure 4.10b shows the results from the same test for a shorter time span to show the influence of the time drift and the time instances where the prescaler is updated more clearly. Without the time synchronization enabled, the nodes would have drifted 400 µs apart within the first 100 ms.

Chapter 5

Conclusion and Future Work

5.1. Conclusion

The sensor network architecture developed in this project provides a robust network suitable for low and medium speed applications. The overhead caused by the CAN protocol is too big for high speed applications like fast control loops, but it satisfies the requirements posed by a Swissloop pod. The developed system is not only lighter than the system previously used by Swissloop, it is also more configurable and scalable.

The Address management and time synchronization of the network layer allow for synchronization of sensor data collection and allow for better timed data collection. The memory and commands of the session layer allow for a highly customizeable sensor interface. The user layer for the network master and listener provide an easy-to-use interface for the functionalities of the network.

The adapter PCB developed especially for the sensor network provides a platform to which many different kinds of sensors can be connected. The high configurability of the PCB also allow for a wide range of use cases. During this semester project, the bus was proofed to work with up to five Nodes at a time. Multiple example senors were written and tested to confirm that the network functions properly.

5.2. Future Work

The next step in the development of the sensor network is to test it in a larger scale and in a harsher environment. When the system is built into the Swissloop 2021 Pod, it will be tested with 11 Nodes on the bus, in a noisy environment near an inverter and a LIM.

Table 5.1.: Sensors Boards on the Sensor Bus in the 2021 Swissloop Pod

#	Sensor conn. to board	Signal	Location	Update Freq.	Data Size
1	3x Pressure Sensor	Analog Current	Brake	1000 Hz	6 Byte
2	5x NTC Thermistor	Variable Resistance	LIM	100 Hz	10 Byte
2	5x PTC Thermistor	Variable Resistance	LIM	100 Hz	10 Byte
1	Velocity Sensor	RS485	Front	100 Hz	6 Byte
2	4x Distance Sensor	Analog Current	Chassis	5000 Hz	8 Byte
1	Radar Sensor	Analog Current	Front	100 Hz	2 Byte

The sensor network currently has the flaw that the software of the sensor nodes can only be updated via the programming/debug connector on the PCB itself. Therefore, all sensor nodes in a vehicle must always be accessible in order to update the software, and all nodes have to be updated individually. This problem can be solved by implementing a firmware actualization method via the CAN bus. This could not be realized during the semester project, but it will be implemented in the future.

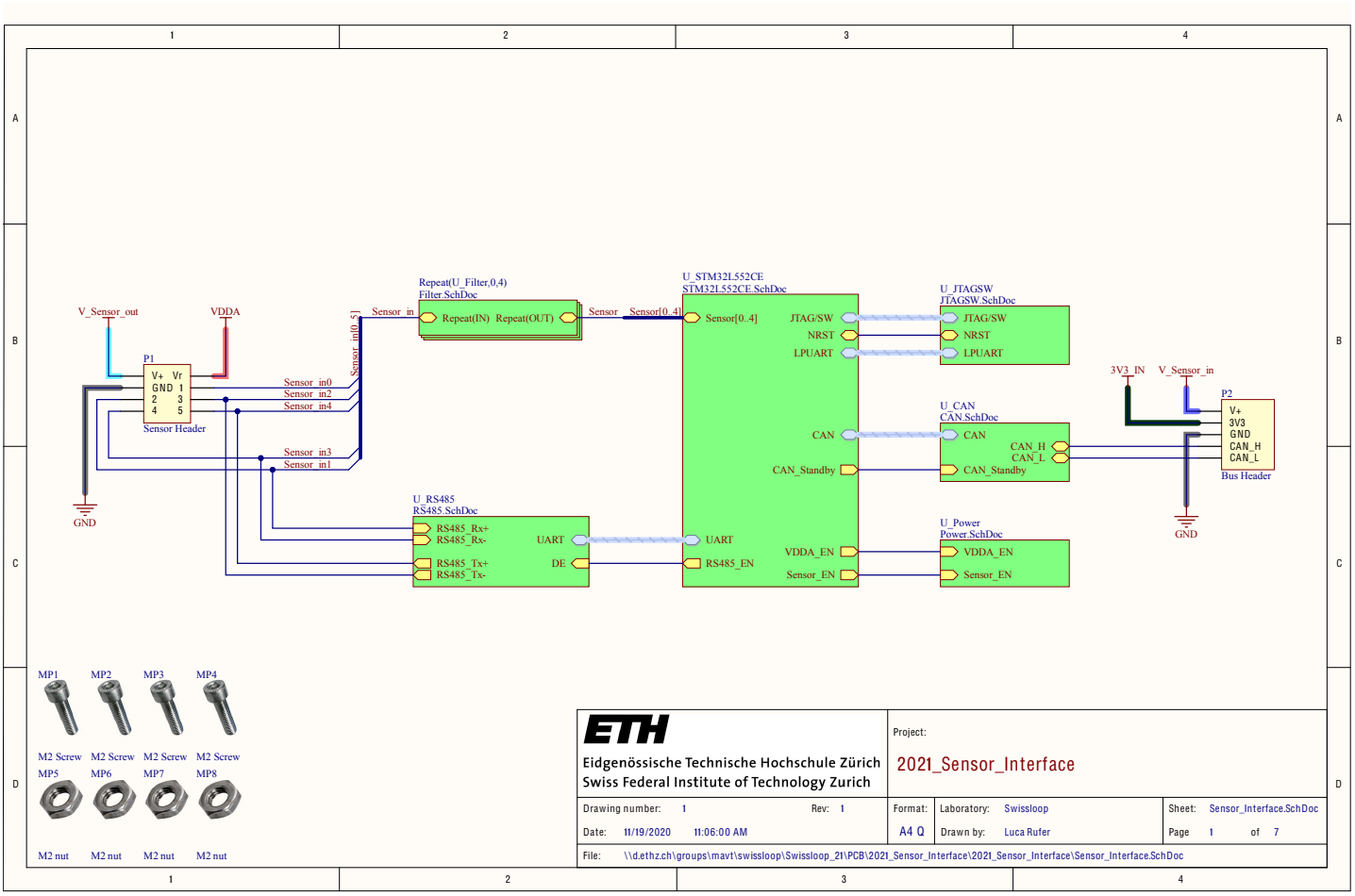
The software currently only contains a handful of sensors. During the project, a few example sensors were written to test key functionalities of the software and the network. However, these example sensors do still not cover all the possibilities of the sensor interface PCB and more sensors need to be written for the deployment of the system.

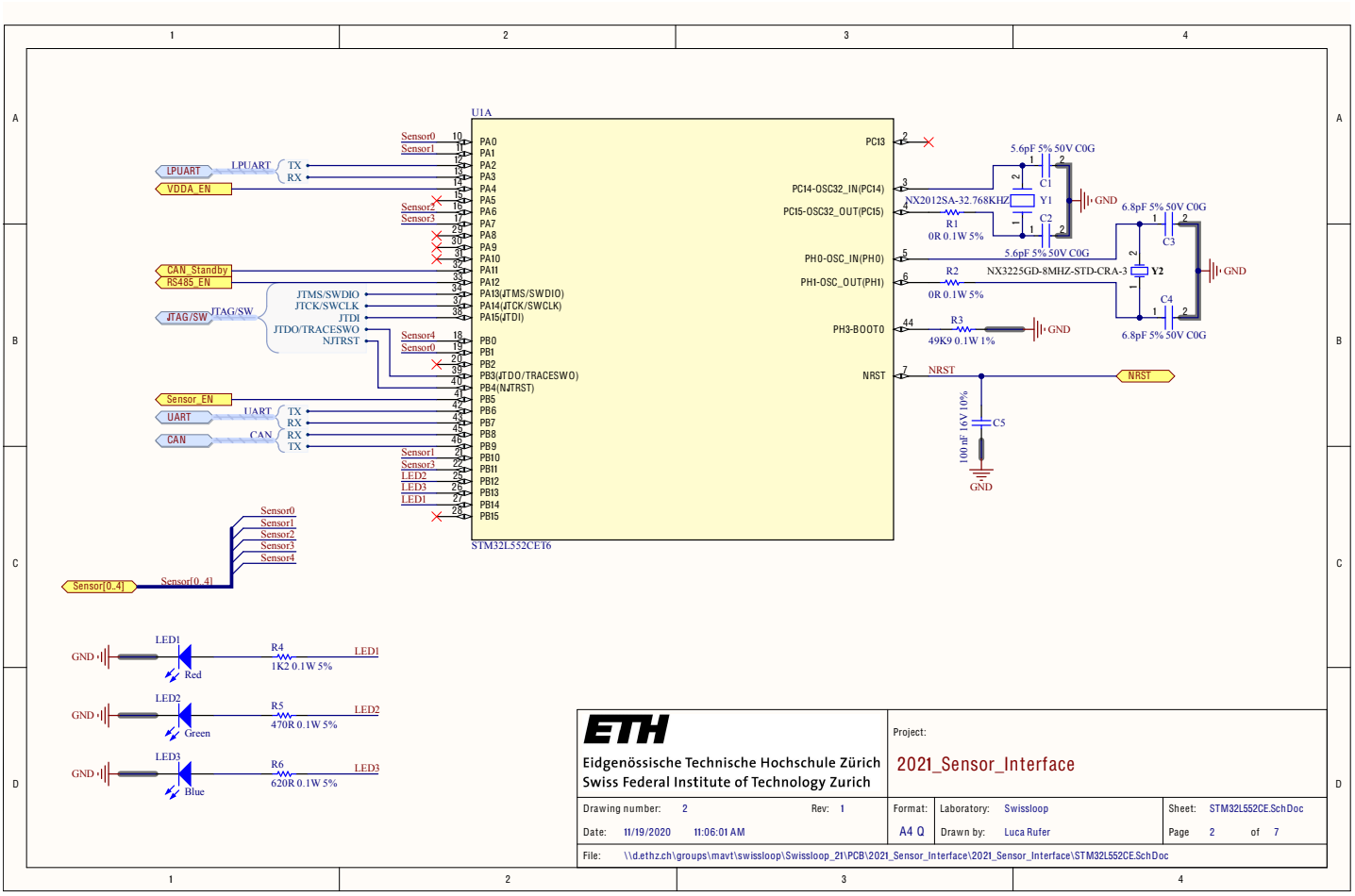
5.3. Implementation on Swissloop 2021 Pod

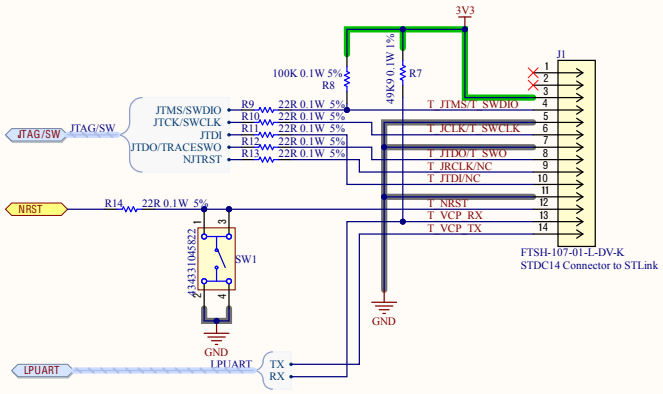
The sensor network developed in this semester project is going to be implemented in the Swissloop 2021 Pod. Development and planning for this pod started in September 2020 and is still ongoing as of the end of this semester project. A list of sensors that will be connected to the sensor bus is shown in Table 5.1. The leftmost column in the table indicates the number of sensor interface PCBs used, and the second column lists the number of sensors per PCBs and the type of sensor. The sensors producing the largest amount of data are the distance sensors which measure the position of the levitating pod relative to the track. When all sensors listed in Table 5.1 are active, 93'600 Bytes of payload data are produced per second. Using a time sync period of 2 ms, an arbitration bitrate of 1 Mbps and a data bitrate of 5 Mbps, this leads to a bus utilization of 67.96 %. The pod has a VCU that acts as a network master and listener to control the sensors and collect, log and transmit the data from the sensor bus. Additionally, the inverter is connected to the sensor bus as a listener and uses the data from the distance sensor for levitation control.

Appendix A

PCB Schematics


V_Sensor_out
VDDA
P1
V+ Vr
GND 1
2 3
4 5
Sensor Header
GND
Sensor_in0
Sensor_in2
Sensor_in4
Sensor_in3
Sensor_in1
Sensor_in[0..5]
Repeat(U_Filter,0,4)
Filter.SchDoc
Sensor_in
Repeat(IN) Repeat(OUT)
Sensor_Sensor[0..4]
U_STM32L552CE
STM32L552CE.SchDoc
Sensor[0..4]
JTAG/SW
NRST
LPUART
CAN
CAN_Standby
VDDA_EN
Sensor_EN
UART
RS485_EN
U_JTAGSW
JTAGSW.SchDoc
JTAG/SW
NRST
LPUART
U_CAN
CAN.SchDoc
CAN
CAN_H
CAN_L
CAN_Standby
U_Power
Power.SchDoc
VDDA_EN
Sensor_EN
U_RS485
RS485.SchDoc
RS485_Rx+
RS485_Rx-
RS485_Tx+
RS485_Tx-
UART
DE
3V3_IN
V_Sensor_in
P2
V+
3V3
GND
CAN_H
CAN_L
Bus Header
GND
MP1
MP2
MP3
MP4
M2 Screw
M2 Screw
M2 Screw
M2 Screw
MP5
MP6
MP7
MP8
M2 nut
M2 nut
M2 nut
M2 nut
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Project:
2021_Sensor_Interface
Drawing number: 1
Rev: 1
Date: 11/19/2020 11:06:00 AM
Format: A4 Q
Laboratory: Swissloop
Drawn by: Luca Rufer
Sheet: Sensor_Interface.SchDoc
Page 1 of 7
File: \\d.ethz.ch\groups\mavt\swissloop\Swissloop_21\PCB\2021_Sensor_Interface\2021_Sensor_Interface\Sensor_Interface.SchDoc


U1A
STM32L552CET6
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
PA8
PA9
PA10
PA11
PA12
PA13(JTMS/SWDIO)
PA14(JTCK/SWCLK)
PA15(JTDI)
PB0
PB1
PB2
PB3(JTDO/TRACESWO)
PB4(NJTRST)
PB5
PB6
PB7
PB8
PB9
PB10
PB11
PB12
PB13
PB14
PB15
PC13
PC14-OSC32_IN(PC14)
PC15-OSC32_OUT(PC15)
PH0-OSC_IN(PH0)
PH1-OSC_OUT(PH1)
PH3-BOOT0
NRST
Sensor0
Sensor1
Sensor2
Sensor3
Sensor4
LPUART
TX
RX
VDDA_EN
CAN_Standby
RS485_EN
JTAG/SW
JTMS/SWDIO
JTCK/SWCLK
JTDI
JTDO/TRACESWO
NJTRST
Sensor_EN
UART
CAN
LED1
LED2
LED3
Sensor[0..4]
GND
5.6pF 5% 50V C0G
C1
NX2012SA-32.768KHZ
Y1
R1
0R 0.1W 5%
C2
6.8pF 5% 50V C0G
C3
R2
NX3225GD-8MHZ-STD-CRA-3
Y2
0R 0.1W 5%
C4
R3
49K9 0.1W 1%
100 nF 16V 10%
C5
R4
1K2 0.1W 5%
Red
R5
470R 0.1W 5%
Green
R6
620R 0.1W 5%
Blue
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Project:
2021_Sensor_Interface
Drawing number: 2
Rev: 1
Date: 11/19/2020 11:06:01 AM
Format: A4 Q
Laboratory: Swissloop
Drawn by: Luca Rufer
Sheet: STM32L552CE.SchDoc
Page 2 of 7
File: \\d.ethz.ch\groups\mavt\swissloop\Swissloop_21\PCB\2021_Sensor_Interface\2021_Sensor_Interface\STM32L552CE.SchDoc



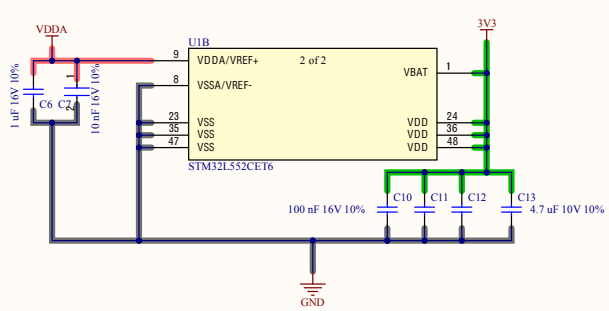
ETH

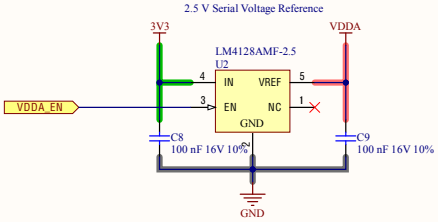
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

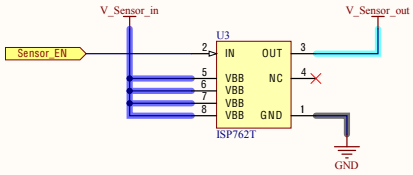
Project: 2021_Sensor_Interface

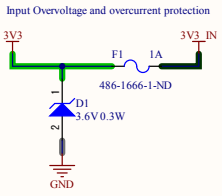
Drawing number: 3	Rev: 1	Format: A4 Q	Laboratory: Swissloop
Date: 11/19/2020 11:06:01 AM			Drawn by: Luca Rufer
Sheet: JTAGSW.SchDoc			
Page 3 of 7			

File: \\d.ethz.ch\groups\mavt\swissloop\Swissloop_21\PCB\2021_Sensor_Interface\2021_Sensor_Interface\JTAGSW.SchDoc

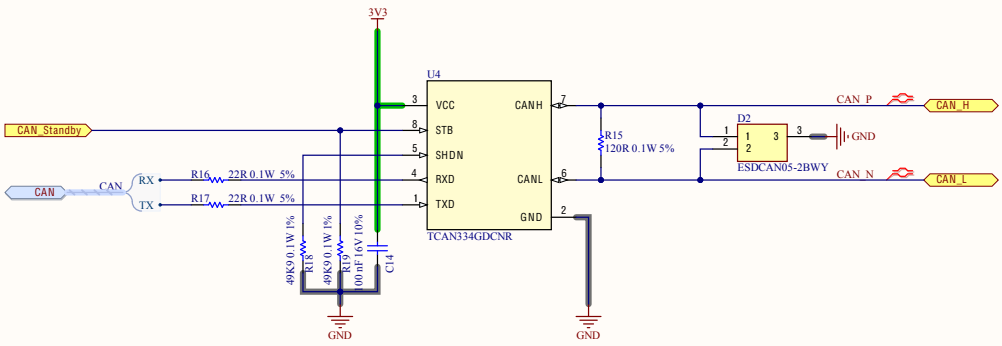




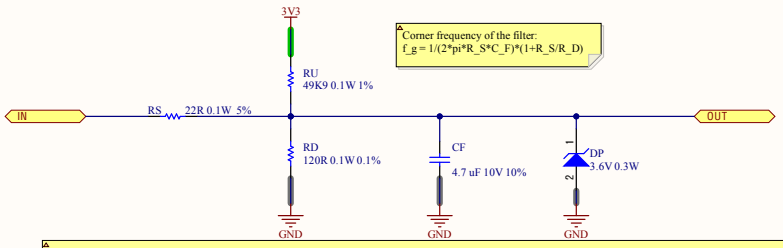




ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich			Project: 2021_Sensor_Interface		
Drawing number: 4	Rev: 1	Format: A4 Q	Laboratory: Swissloop	Sheet: Power.SchDoc	
Date: 11/19/2020 11:06:01 AM			Drawn by: Luca Rufer	Page 4 of 7	
File: \\d.ethz.ch\groups\mavt\swissloop\Swissloop_21\PCB\2021_Sensor_Interface\2021_Sensor_Interface\Power.SchDoc					



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich		Project: 2021_Sensor_Interface	
Drawing number: 5	Rev: 1	Format: A4 Q	Laboratory: Swissloop
Date: 11/19/2020 11:06:01 AM		Drawn by: Luca Rufer	Sheet: CAN.SchDoc
File: \\d.ethz.ch\groups\mavt\swissloop\Swissloop_21\PCB\2021_Sensor_Interface\2021_Sensor_Interface\CAN.SchDoc			Page 5 of 7



Example values for filter configurations:

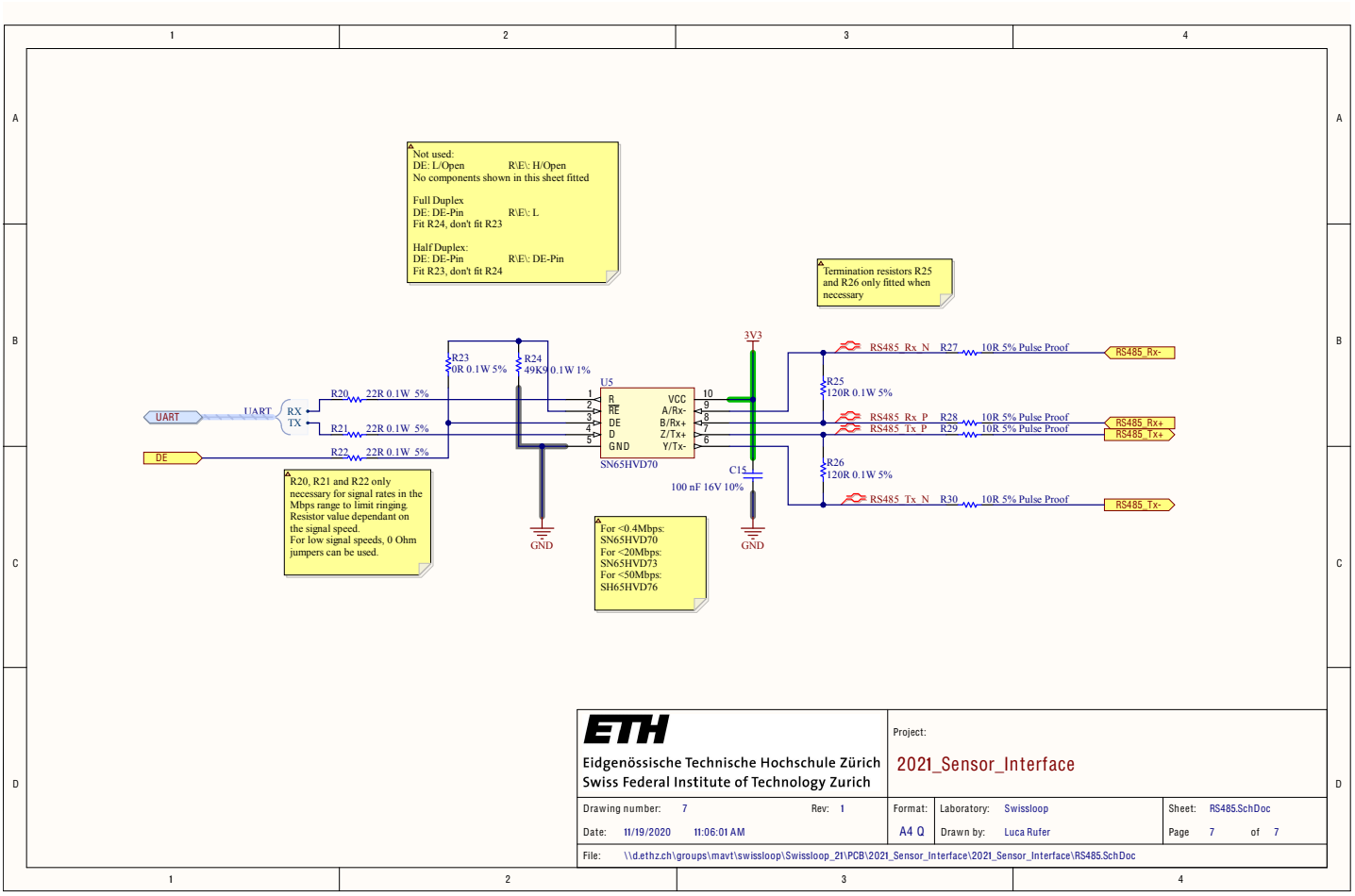
	RS	RU	RD	CF	Result
Current input (4 - 20 mA)	22 Ohm	NC	120 Ohm	4.7 uF	Out Voltage = 0.48 - 2.4 V f_g = 1.8 kHz
Voltage input (0 - 5 V)	500 Ohm	NC	500 Ohm	636 nF	Sensor input current = 0 - 5 mA f_g = 1 kHz
Thermistor R0 = 10kOhm	10 Ohm	NC	1 kOhm	16 uF	max VADD current = 2.5 mA on short circuit f_g = 1 kHz
SPI MISO	22 Ohm	10 kOhm	NC	NC	Pullup and reduction of signal ringing with RS f_g < 16 MHz due to board and Diode capacitance
RS485 Any of Tx/Rx +/-	NC	NC	NC	NC	Filter not used, connect R27-R30 instead
No Filter	0 Ohm (Jumper)	NC	NC	NC	Filter bypassed

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Project: 2021_Sensor_Interface

Drawing number: 6	Rev: 1	Format: A4 Q	Laboratory: Swissloop	Sheet: Filter.SchDoc	
Date: 11/19/2020	11:06:01 AM		Drawn by: Luca Rufer	Page 6	of 7

File: \\d.ethz.ch\groups\mavt\swissloop\Swissloop_21\PCB\2021_Sensor_Interface\2021_Sensor_Interface\Filter.SchDoc



ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Project: 2021_Sensor_Interface

Drawing number: 7	Rev: 1	Format: A4 Q	Laboratory: Swissloop	Sheet: RS485.SchDoc	
Date: 11/19/2020 11:06:01 AM			Drawn by: Luca Rufer	Page 7 of 7	

File: \\d.ethz.ch\groups\mavt\swissloop\Swissloop_21\PCB\2021_Sensor_Interface\2021_Sensor_Interface\RS485.SchDoc

Bibliography

[1] E. Musk, "Hyperloop alpha," 2013, [Online; accessed 21-December-2020]. [Online]. Available: https://www.tesla.com/sites/default/files/blog_images/hyperloop-alpha.pdf

[2] "Swissloop," 2020, [Online; accessed 21-December-2020]. [Online]. Available: https://www.swissloop.ch

[3] "European hyperloop week," 2020, [Online; accessed 21-December-2020]. [Online]. Available: https://hyperloopweek.com

[4] "Road vehicles — Controller Area Network (CAN) — Part 1: Data link layer and physical signalling," International Organization for Standardization, Geneva, CH, Tech. Rep. ISO 11898-1:2015(E), 2015.

[5] M. Akpinar, K. W. Schmidt, and E. G. Schmidt, "Improved clock synchronization algorithms for the Controller Area Network (CAN)," 2019.

[6] D. Lee and J. Allan, "Fault-Tolerant Clock Synchronisation with Microsecond- Precision for CAN Networked Systems," 2003.

[7] STM, *RM0438: STM32L552xx and STM32L562xx advanced Arm®-based 32-bit MCUs*, STMicroelectronics, May 2020.

[8] ——, *STM32L5 Nucleo-144 board (MB1361)*, STMicroelectronics, June 2020.

[9] ——, *STLINK-V3SET debugger/programmer for STM8 and STM32 (UM2448)*, STMicroelectronics, June 2020.