Universitetet i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Bachelor i ingeniørfag / Automatisering og elektronikkdesign	Høstsemesteret 2022 Åpen
Forfatter(e): Asbjørn Stokka	
Fagansvarlig: Kristian Thorsen Veileder(e): Kristian Thorsen	
Tittel: mikrokontrollerbasert plattform for arbeid med autonom kjøring Engelsk tittel: Microcontroller based platform for autonomous driving	
Studiepoeng: 20	
Emneord: kretskortdesign, mikrokontrollere, RTOS	Sidetall: 61 + vedlegg/annet: 16 Stavanger 15. desember 2022

Innhold

Content **i**

Summary **1**

Terms, Abbreviations, and Acronyms **1**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem definition 2

1.3 Usecases 2

1.4 Existing work 3

1.5 Planning 3

1.5.1 Product Requirement Specification 3

1.5.2 Functionality Specification 4

1.5.3 Product/System Design Specification 8

1.5.4 Notes and alternate Specification 11

2 Theory 12

2.1 Communication - Media layers 12

2.1.1 I2C - Inter-Integrated Circuit 12

2.1.2 UART - Universal asynchronous receiver-transmitter [2] 14

2.1.3 CAN bus 14

2.1.4 Data Link layer 16

2.1.5 CAN Filtering 19

2.1.6 Wireless communication 20

2.1.7 Communication needs 20

2.1.8 Discussion on communication choices 21

2.2 TDD - Test Driven Development and Testing 23

2.3 Firmware/Software 25

2.3.1 STM32Cube 25

2.3.2 VSCode 25

2.3.3 Zephyr RTOS 26

2.3.4 Python 26

2.4 3D printing and laser cutting 27

3 Design and construction of the hardware, firmware, software and physical components 29

3.1 Single Board Computer - Embedded Linux 29

3.1.1 Raspberry pi 29

3.1.2 BeagleBone 30

3.1.3 Conclusion 31

3.2 PocketBeagle, a development tool 31

3.2.1 Debugging 31

3.2.2 Automated testing with Zephyr 32

3.3 SBC system 32

3.4 CAN-Sensor-Node 33

3.4.1 PCB design 33

3.4.2 Firmware 35

3.4.3 Use case for the sensor node 45

3.4.4 Further improvements 45

3.4.5 Conclusion 45

3.5 RC Platform Board 46

3.5.1 PCB 46

3.5.2 Manufactured parts 47

3.5.3 Zephyr RTOS firmware 48

3.5.4 BLE Over The Air firmware update 53
3.5.5 Further improvements 54
3.6 Python software 55
3.6.1 Computer Control System 56
3.6.2 CAN bus Socket server 56
3.7 Building a small autonomous car 57

4 Results and discussion **58**

4.1 SBC - development tool 58
4.2 Testing 58
4.3 Small single-board car 59
4.4 Small autonomous car 59
4.5 Python software 60
4.6 Discussion 60

5 Conclusion **61**

Bibliography **63**

Vedlegg **63**

A Tests **64**

B Design files **75**

C Datablad **80**

Abstract

This report covers setting up a development system for embedded development for microcontrollers and Linux Embedded SBC systems. And, the development and testing of a microcontroller-based platform that can be used for prototyping and developing autonomous driving. The Platform can be used as a single board small car, or as a small multi-sensor-node vehicle with CAN bus connecting the different nodes. A SBC system can be used for computer vision and controlling and driving the car. It should be built using cheap, off-the-shelf components making it accessible for anyone interested.

Terms, Abbreviations, and Acronyms

3D-printing	A form of additive manufacturing where you add material to an object
ABS	Acrylonitrile butadiene styrene. A common thermoplastic polymer. Offers desired properties like impact resistance, structural strength, electrical insulating properties, as well as being easy to paint and glue. It is considered cheap, though somewhat advanced/difficult to use for 3D printing.
BeagleBone	High-performance, low power, and Open Source system. SBC available in several configurations.
BeagleBone Cape	Capes are daughter-board addons extending the functionality of BeagleBone and PocketBeagle products.
CAD	Computer Aided Design
CAN	Controller Area Network, a bus for communication between microcontrollers
CRC	Cyclic Redundancy Check
Git commit	A commit is a snapshot of a repository at a specific time. As you "commit"your changes, you also add a message to that commit. This message can also tag one or more issues triggering automation, and linking the commit to that issue.
GitHub	A development platform to build, scale, deliver, and manage software projects with Git.

Github actions	Github actions is a yaml based framework for adding automation to repositories. Each action will be triggered only on specified events, and can be used for error checking, linting, releases etc.
GitHub issue	An issue is a note added to a github repository containing. An issue can contain information, discussions, tags, be given a deadline, assigned to a user, be tracked as part of a project milestone. As an issue is closed, it may also be tagged in a commit to signify that the commit has addressed the issue.
GNSS	More than just GPS, typically also includes GPS, GLONASS, Baidu, Galileo and other satellite systems.
GPIO	General-Purpose Input/Output
HAL	Hardware Abstraction Layer - libraries and functions to let you write more universal code that can be compiled for other microcontrollers
KiCad	A Cross Platform, Open Source Electronic design software suite
Linux Embedded device	
Makerspace	A place were people with a shared interest in computing and technology gather around their shared interest. Often sharing ideas, knowledge, and/or equipment.
MicroPython	A small implementation of Python3 with a subset of the standard library meant to run on microcontrollers.
OSI layer model	Open Systems Interconnection model, an abstract conceptual model describing communication between computers by splitting it into seven different layers: Physical, Data Link, Network, Transport, Session, Presentation, and Application
PCB	Printed Circuit Board
Rapid prototyping	A production process to move quickly from a CAD model to a produced prototype

Raspberry pi	The name of the UK foundation behind the Raspberry Pi SBC. First started to develop a small, cheap computer solution that would have a low power usage. Now the Raspberry Pi boards (and Pi compute modules) can be found in many hobby projects and embedded systems.
Repo	Repositories contains project files, and version history for all the managed files.
RTOS	Real Time Operating System, operating system for systems with real time applications where events have critically defined time constraints. Switches between tasks based on their priorities.
SBC	Single Board Computer. Started as a tool for teaching computer science, was conceptually used in computers meant to give access to computers in development countries with little/unreliable power. And are now widely used in embedded systems.
Voron project	A community driven project to create a nn-o-compromise"3d printer design. They have various 3D printer designs for core-xy 3d printers capable of printing high quality fdm prints in various materials.

Kapittel 1

Introduction

This bachelor thesis is a

1.1 Motivation

Starting a UiS we early on had a great project using the LEGO EV3 robots for projects in applied physics and mathematics. Though we quickly met several limitations when developing scripts to control them in Matlab it was very interesting to work with. Later on, as a student assistant in ELE130, I tested a similar syllabus where Matlab could be changed for micro-python. Though it offered increased processing capabilities, there were large issues in debugging the code. And students often ran into issues with communication between the computer software and the robot.

And, most of all, at the end of the semester everyone has to hand in the Lego hardware and is left without the possibility to continue their projects and efforts. Buying their own Lego robots would be a large investment as it is somewhat expensive, and you can only use Lego pieces, sensors, motors, and parts to build them.

I have for several years worked on a variety of hobby projects, and been active in the local Makerspace community. Through working on increasingly advanced projects I have come to see that there is a lot of possibilities with various Open Source systems. And, that it is possible to get far using those solutions instead of expensive industry software/hardware. For hobbyists a lot of companies offer free or cheap licenses, but that is not always the case for startups, and small/medium sized companies.

1.2 Problem definition

Building a small car, where the main goal is not the finished car itself, but the total framework. The goal is to make resources for a set valuable tools (both software and hardware) available for working on embedded projects (both SOB and micro-controllers). The project will be a combination of SBC (Single Board Computer) and micro-controller based boards communicating together using an industry standard protocol. Creating a low-cost platform that can be used for control theory training, computer vision experimentation, and for developing autonomous vehicles. With a set of relevant development, and debugging tools easily available.

1.3 Usecases

The main usecase of this project is to make ready a platform to use in a bachelor degree in computer science in driving with computer vision. The project result can be used for computer science projects, studying or working on embedded development, for control-theory studies, as an alternative platform for applied science and math projects in robot programming, for developing autonomous vehicles and/or projects in computer vision, or simply as a guide/toolbox for using Embedded Linux on SBC in CAN bus projects, getting started with Zephyr RTOS and remote flashing/GDB debugging using an Embedded Linux platform, or for getting started with automated pyTest testing. The repositories also contain example GitHub actions to inspire the use of pipelines with a more automated workflow.

1.4 Existing work

From ING100 (now ELE130) we had a project on applied physics and math in robot programming, where we programmed in Matlab to control LEGO EV3 robots for various tasks. The robots came with a set of different sensors and motors, and you could choose a set of four sensors and four motors to use at the same time. With ELE130 the subject also had the possibility to program the robots using Micro-Python. Using Micro-Python allowed for faster processing of sensor data, and more responsive application. However, debugging the Micro-Python scripts proved more difficult for the students. The LEGO EV3 robots are also quite costly, which is something most students do not have access to outside of labs and classes.
Before starting this bachelor thesis I made a small prototype robot based on an Atmega328p micro-controller and L298N motor driver. It worked quite well but had limited communication capabilities besides a radio transmitter for direct control signals.

1.5 Planning

1.5.1 Product Requirement Specification

The end product here is divided in three parts. An Embedded Linux platform running on a SBC communicating with and controlling a vehicle. A system to be used in a small form factor vehicle with sensors, and motors that can be programmed to be used for a variety of purposes (control theory/distance-servo regulating). A modular platform for sensors that can be used and connected to the system to give extra information. These are meant to be used on a slightly larger, but still relatively small form factor vehicle.

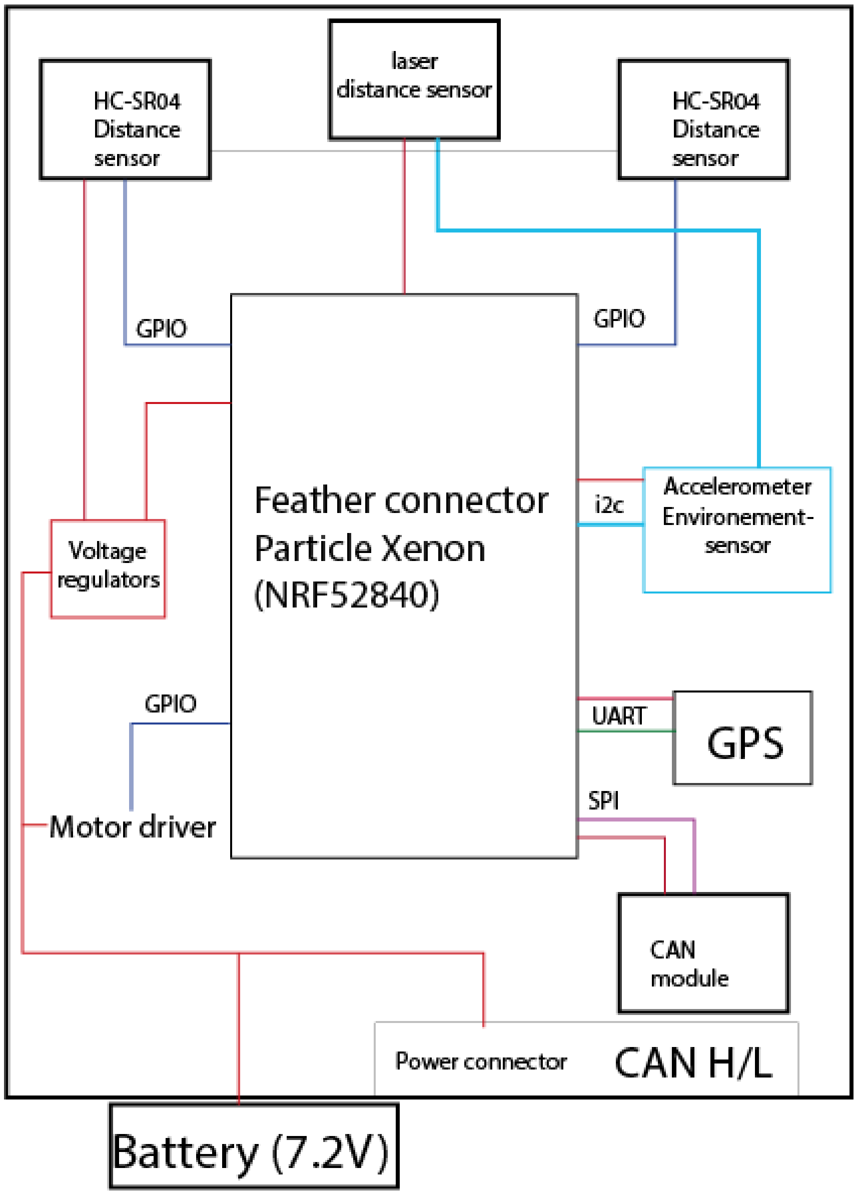
- The SBC can communicate with computers, and with the platform cards.
- The system logs sensor data, presents them, and shows system status.
- The system can control motors/servo-motor for mobility.

- The sensor-nodes require a distance sensor.
- The sensor node should be powered and have communication with a single cable and should be easy to connect to the system.
- The sensor nodes should be modular in such a way that you can easily change the number of nodes, and where they are connected.
- The platform card should be possible to use stand-alone and/or in combination with the sensor nodes and embedded Linux SBC system.

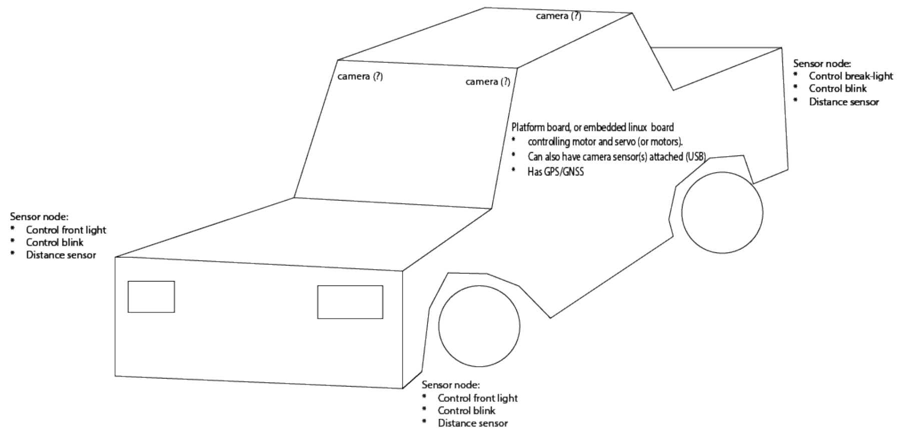
1.5.2 Functionality Specification

- Power supply for the nodes, and the platform card.
- Needs to have a useful guide to set up the development environment.
- Needs to be a user-friendly interface for controlling and testing.
- A distance sensor based on transmission and reflection of ultrasonic sound.
- Easy to expend/change sensors, and a varied basic set of sensors and sensor data.
- The system communicates by CAN bus.

Figure 1.1 is a sketch of a basic version of the system featuring only a single platform board with directly connected sensors. It can be used as a stand-alone training platform in control theory, or other autonomous tasks using the somewhat limited amount of sensors available. Given a BLE/Bluetooth connection it can communicate with an external control system. It can also be connected to an SBC Embedded Linux system and used with image sensors, for example in a small-scale track/testing environment. It is small, lightweight, cheap, and accessible. Built of cheap parts, and a frame/modules needed to build it can quickly be laser-cut and/or 3D printed.

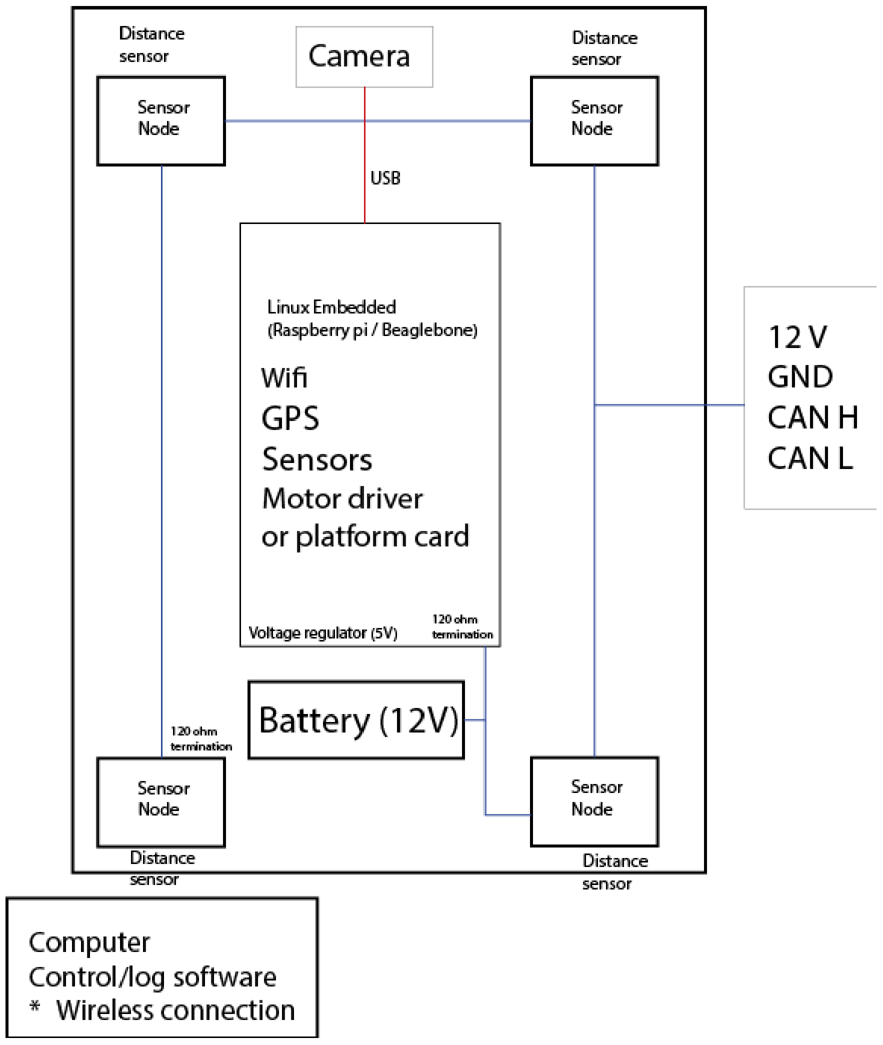


Figur 1.1: Small form-factor car with RC-platform board



Figur 1.2: Sketch of full system with Embedded Linux controller, computer vision and sensor nodes

Figure 1.2 is a concept sketch of a small battery-powered vehicle with a CAN bus for sensors, lights, and motor control. It can be used for computer vision using a number of USB image sensors etc. it can also feature a wireless connection to a computer control station where you can monitor and record sensor values.



Figur 1.3: Diagram of a full sensor system with CAN bus and image sensors

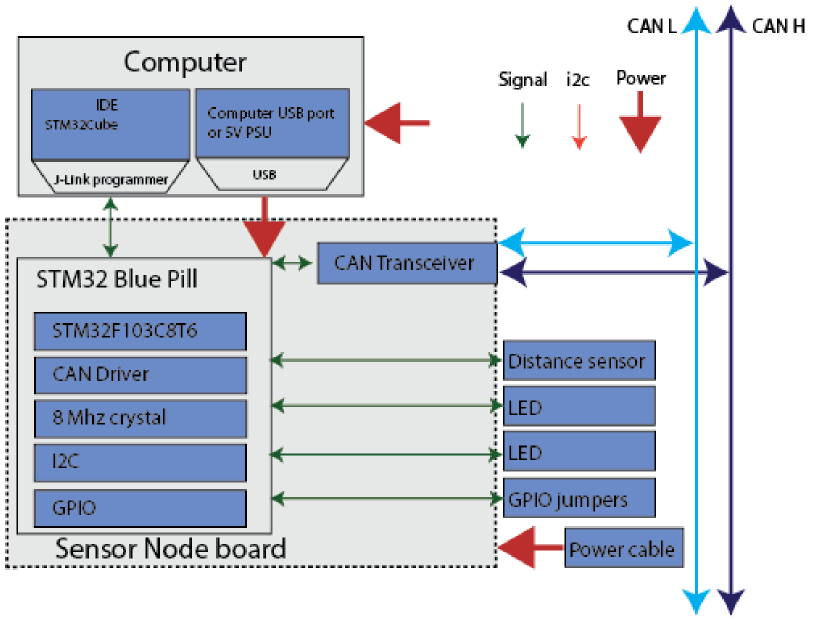
Figure 1.3 is a diagram of the full scale"modular system with a SBC bb-rain"with an Embedded Linux CAN bus connection, sensor nodes, power delivery, and image sensors. It could be made as shown in the conceptual drawing in figure 1.2. Communication between nodes in the system is based on CAN bus. Other sensors and functionality can easily be added by connecting them to the CAN bus and adding message-id/packet description to relevant parts of the system. The "car system"can communicate with an external control system using a Socket connection over WiFi. If in an area with access points the range of this would be good. In remote areas, it would be smart to consider a 4G or 5G modem.

1.5.3 Product/System Design Specification

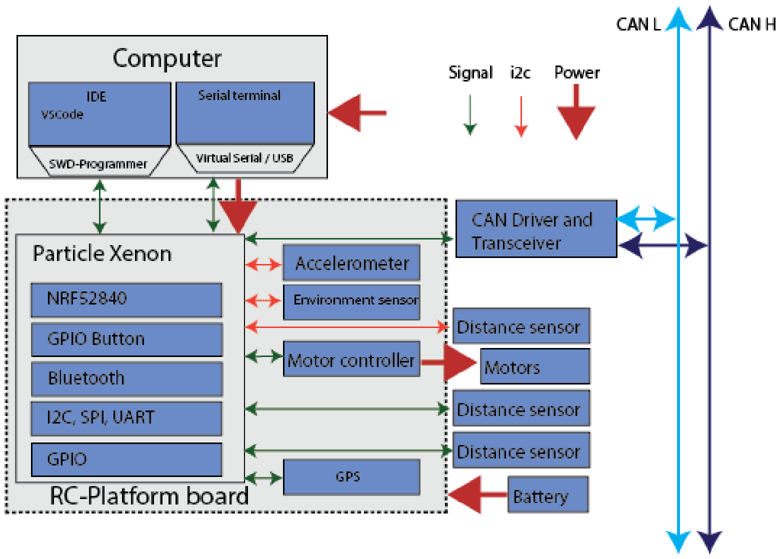
- The control system can run on a computer or an Embedded Linux system (Raspberry Pi or BeagleBone).
- The Graphical User Interface on the sensor system is built using Python with Qt.
- The Embedded Linux system can communicate on CAN bus, and over a computer network.
- The firmware for the sensor-node (STM32F1) is written in C,
- The firmware for the platform card is built on Zephyr RTOS.
- To build the sensor-board an STM32F103C8T6 Blue pill board will be used for development. And the dedicated small board will be made based on the STM32F103C4T6A chip.
- To build the platform board will be easy to solder and have a feather connector. It will use a Particle Xenon (NRF52840), and a set of i2c breakout board sensors in development.
- For a use case with manufacturing the feather connector can be redesigned and changed to a SMD E73-2G4M08S1C (nRF52840), and a set of sensors can be mounted directly on the board instead of on breakout boards keeping it compatible with the firmware.
- The platform board can control 2 motors using the onboard motor controller, and/or an external motor using PWM, as well as a servomotor.
- The platform board can communicate using CAN bus, and using UART over serial.
- Optionally the platform board can communicate over Bluetooth/BLE.

- To maintain a constant sampling interval, interrupts are used to get samples from the distance sensor and accelerometer (in STM32Cube fw, SysTick timer).
- The platform board has a PID regulator.
- The platform board uses FPU for floating point calculations in the microcontroller.
- Communication between the embedded Embedded Linux SBC and computer uses a network socket or SSH.
- The power supply for the different parts of the system is done through voltage regulators on each card. The platform is designed to have between DV 7.2 V and DC 12 V battery power supply. The Embedded Linux system generally uses a 5 V USB power input and will need a voltage regulator accordingly.
- On the platform card, a switched-mode power supply takes the voltage from 7.2-12 V to 3.3 V. And an LDO voltage regulator from 7.2-12 V to 5 V.
- On the sensor-node boards LDO voltage regulators will be used to take the voltage from 7.2-12 V to 3.3 V/5 V.

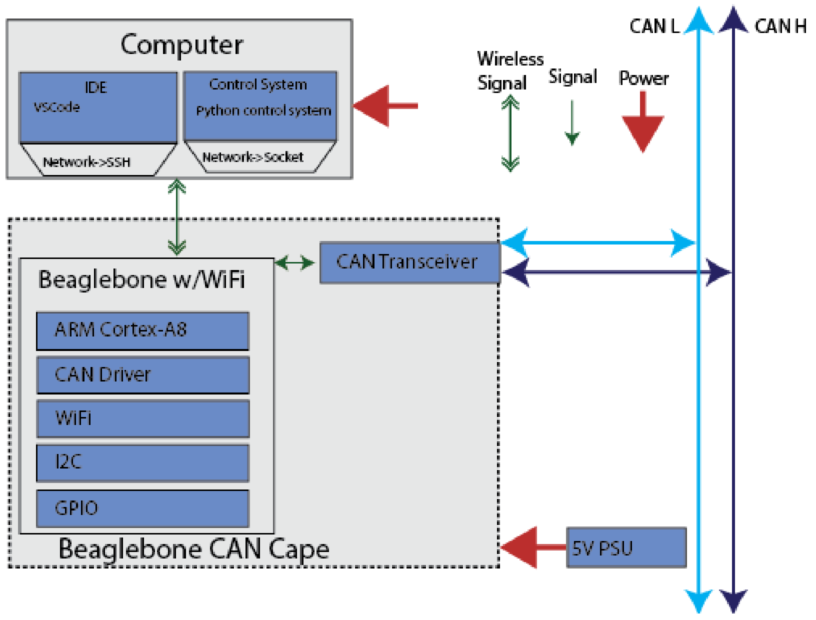
Following here are three figures 1.4, 1.5, and 1.6 showing a diagram of the different boards. Generic names have been used for several of the sensors where they can be easily exchanged for similar devices. For the exact modules used refer to the BOM for that board. Not also the CAN H and CAN L lines that connect the systems, and they should have a 120Ω resistor for termination at each end. Read more about this in the theory chapter about CAN bus. And, the wired computer connections shown are used for programming and direct interaction with the board/microcontrollers. They are not needed when the system is in use.



Figur 1.4: STM32 Sensor CAN Node



Figur 1.5: RC Platform board



Figur 1.6: SBC Embedded Linux device

1.5.4 Notes and alternate Specification

The Particle Xenon can be switched for an STM32-based board by making new board files for the Zephyr RTOS system, and defining the various hardware interfaces. That would require designing a new board, or finding a suitable feather-based STM32 microcontroller.
Figure 1.4 shows the sensor board as used in development with the STM32 Blue Pill dev board. As the first revision of the CAN Sensor Node board had a design flaw.
In figure 1.6 the Beaglebone WiFi board can easily be substituted with a Raspberry Pi by adding an SPI CAN driver. It is also possible to use Beaglebone boards without built-in WiFi by using a wired ethernet connection, USB wifi module, or without an internet connection.

Kapittel 2

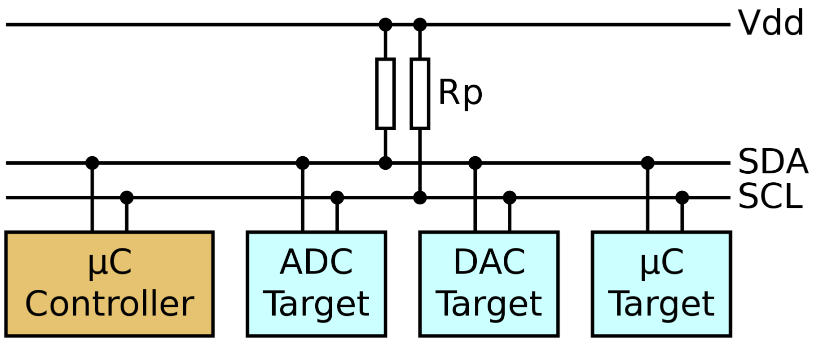
Theory

2.1 Communication - Media layers

In the following section, we will briefly go through some of the relevant communication alternatives focusing on the media layers: the physical layer and the data link layer of the OSI model. That will be relevant for hardware design choices, and be useful for testing, debugging, and verification of the hardware.

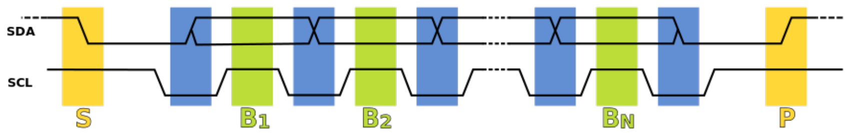
2.1.1 I2C - Inter-Integrated Circuit

I2C is a synchronous controller/target (formerly known as master/slave) bus. It is widely used for connecting low-speed peripherals over short distances. And, it is very popular in prototyping systems, with a great number of cheap breakout boards available. By design, the peripheral and the processor should be on the same PCB, and be wired together at a relatively short distance, of less than 30 cm. There can be a single controller (master) and multiple target (slaves) nodes, and the lines have a pull-up to Vdd as seen in figure 2.1.



Figur 2.1: i2c example schematic, illustration from Wikipedia [3]

The SDA (data) and SCL (clock) lines of the bus should have a passive pull-up, and be at Vdd when be bus is inactive. The lines are never driven high, but is pulled to ground to give a logic "0", and left floating by the controller (pulled high by the pull-up resistor) for logic "1". And, messages should be started by the controller claiming the line by creating a falling edge on the SDA line. Each bit is read on the SDA line after the rising edge on the SCL (clock) line.

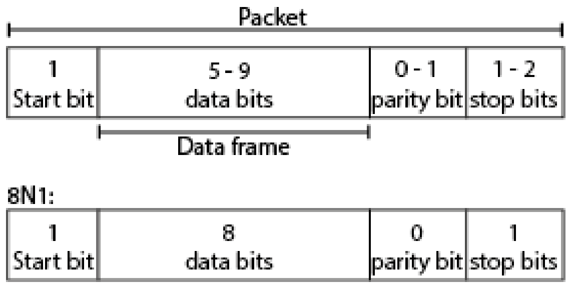


Figur 2.2: i2c signal example, illustration from Wikipedia [3]

A significant feature of i2c is clock stretching, where a target device can hold the SCL (clock) line low to indicate that it needs more time to process data before continuing the transmission.

2.1.2 UART - Universal asynchronous receiver-transmitter [2]

UART transmits data sequentially in frames starting with a start bit, then 5-9 bits of data, and then it can have a parity bit (stating the odd/evenness of the data), and stops with 1 - 2 stop bits. UART communication does not have any shared timing/clock. The operation is controlled by an internal clock signal, and you set this by the baud rate setting. It is important that both the sender and receiver have a matching clock setting. The transmitter and receiver can be connected in different modes, one of them being full-duplex where Tx (transmit) on the transmitter is connected to Rx (receive) on the receiver (and again the same for Rx/Tx for the changed roles).



Figur 2.3: UART data packet

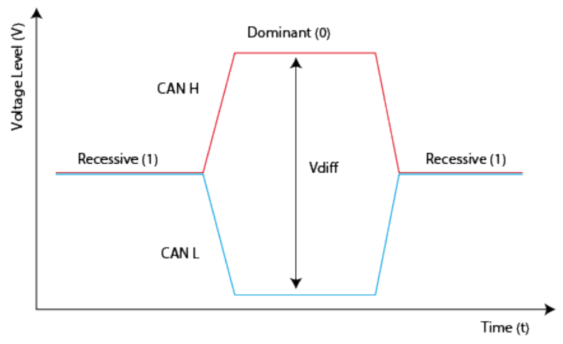
When idle, the line should be high (logic '1') showing that the line and transmitter are working. A common setting for UART data frames is shown in figure 2.3 called 8N1 where there is 1 start bit (logic '0'), 8 data bits, no parity, and 1 stop bit (logic '1').

2.1.3 CAN bus

CAN bus [1] (Controller Area Network) was first developed for the automotive industry. It is a robust standard designed for microcontrollers and other devices to communicate. The bus allows for multiple devices to communicate on the same bus and has a priority system to avoid collisions. There are several different specifications, and the latest one is CAN 2.0. We will focus on the CAN 2.0A specification with an 11-digit identifier. The CAN standards have also been extended with CAN FD (Flexible Data-Rate), which can optionally switch to other data rates (faster).

Physical layer

The CAN-bus is a differential bus with two digital signals CAN-H and CAN-L (CAN High and CAN Low). It is asynchronous as it does not use a clock signal. The two signal lines (CAN H and CAN L) are terminated in each of the two ends with a 120ω resistor. The termination helps suppress reflections, as well as return the bus to the recessive state when not kept dominant by a node. There are two logical states specified in CAN, recessive and dominant. When recessive (a logic 1) both CAN H and CAN L are approximately 2.5 V, and the differential voltage is below the minimum threshold (<0.5 V). For the dominant (a logic 0) state CAN H is high at approximately 3.5 V, and CAN L is low at approximately 1.5 V. The differential voltage is then 2 V. A dominant bit overdrive a recessive bit, and this is used in arbitration [13]. A recessive and dominant signal is illustrated in figure 2.4.



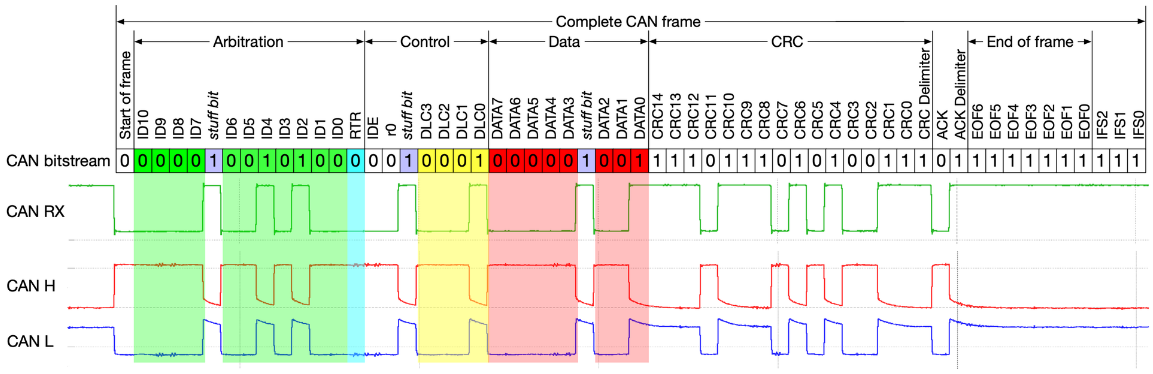
Figur 2.4: CAN signal (inspired by [13])

There is no specification made on connectors, colors, labels, or pin-out for CAN. But some of the electrical aspects like voltage and current can be found in ISO 11898-2:2003. Even though different vendors use different connectors, one common connector is a 9-pin D-sub male connector with:

- Pin 2: CAN-L
- Pin 3: GND
- Pin 7: CAN H
- Pin 9: Can V+ (power)

2.1.4 Data Link layer

When transmitting data, a message is called a frame. As mentioned in the CAN bus introduction there are two frame formats: CAN 2.0A and CAN 2.0B where CAN 2.0B has an extended frame supporting 29-bit identifiers. To make sure the signal is synchronized, if there are 5 consecutive bits with the same polarity, a bit of the opposite polarity (stuff bit) will be added. This bit is simply removed again by the receiver.



Figur 2.5: CAN frame (Source Wikipedia [1])

Base CAN frame format:

Field name	Length	Bit value	Description
Start-of-frame	1	0	Denotes the start of a frame
Id	11	1/0	Message Id, also used for arbitration/message priority
RTR	1	0: data frame 1: request frame	Remote transmission request frame, requests data
IDE	1	0	Identifier extension bit, 0 for base frame format with 11-bit identifier.
r0	1	0	Reserved bit, must be 0
DLC	4	1/0	Data Length Code: 4 digit number bytes of data (0-8 byte) will be in the data field
Data field	0-64	1/0	Transmitted data
CRC	15	1/0	Cyclic redundancy check. For error checking the frame.
CRC delim	1	1	CRC delimiter, must be recessive (1)
ACK slot	1	1	Transmitter sends recessive (1), receiver can ACK with dominant (1)
ACK delim	1	1	ACK delimiter, must be recessive (1)
EOF	7	1	End Of Frame. Must be recessive (1)
IFS	3	1	Inter-Frame-Spacing. Separation between frames.

Id and Message priority

The id are the bits marked in green in figure 2.5. For arbitration on a CAN bus if two nodes start transmitting at the same time there is a system for arbitration and message priority. Both Nodes would start off by sending a dominant (0) for SOF (start of frame) and then start transmitting their ID. The first node to see a dominant (0) on the line when it is trying to transmit a recessive (1) loses the arbitration and stops transmitting. Remember from the physical layer description that a recessive signal is the passive state for the line. While the node with the lowest message Id continues transmitting. A lower id will always win the arbitration and have higher priority on the bus (has the most leading dominant (0) bits in the id field). The Frame id field has to be unique for each node to avoid two nodes completing the arbitration without a winner, causing collisions. And the id field is also used for filtering which will be covered in more detail later.

Control

IDE (in white) is a 0 for 11-bit identifiers, and 1 for extended id (29-bit).
r0 (also white) is a reserved bit and is always 0.
The Data length code is the set of bits marked in yellow in figure 2.5. This is a binary number for how many bytes of data will be given in the data field (bits marked in red). 0001 is 1 byte of data, if you changed it to 1000 that would be 8 bytes of data.

Data

A number of bytes of data are given by the data bits. This is the actual data transmitted with the data frame. Request frames do not have these bytes.

CRC - Cyclic Redundancy Check

This is a check to see if there are any errors in the received data packet. The sender calculates a 15-bit checksum and adds it as the CRC bits of the frame. The CRC is generated by doing a polynomial division on the data, using a polynomial based on the bit pattern in the beginning of the frame. The remainder of the division is used as the checksum. When the receiving node checks the data, the checksum it calculates should match the checksum given in the CRC bits. The microcontroller uses XOR operations and bit-shifting on the data to perform this division.

ACK

After the CRC check there is a slot where a node receiving the frame can write a dominant bit (0) to the bus to signal to the sending node that the frame has been received.

End of Frame

The end of the frame consists of 7 recessive bits (1). And there is a spacing of 3 bit left recessive before the next frame is allowed to be sent on the bus.

2.1.5 CAN Filtering

CAN bus has a system for filtering incoming frames by id. This filtering is done to reduce interrupts as only the accepted frames will cause an interrupt, while the rest are discarded by the controller. The CAN controllers can filter based on a list or mask mode.

Id filter

The incoming frame id is checked with a list of accepted ids. If the incoming frame id is present in the filter bank identifier register, the frame is accepted.

Mask filter

A filter bank register is used as an identifier register, and another as a mask register. The mask register identifies the bits in the identifier register to compare, and what bits to ignore. If the bits to compare in the identifier register matches those in the incoming frame id, the frame is accepted. The remaining bits in the id field are simply ignored (not compared, but still follow the frame).

2.1.6 Wireless communication

For the wireless connections, we will not go into depth on the physical, data link, network, or transport layer of the OSI model [4]. But only make some remarks regarding the use of wireless networks and Bluetooth in embedded systems.

WiFi

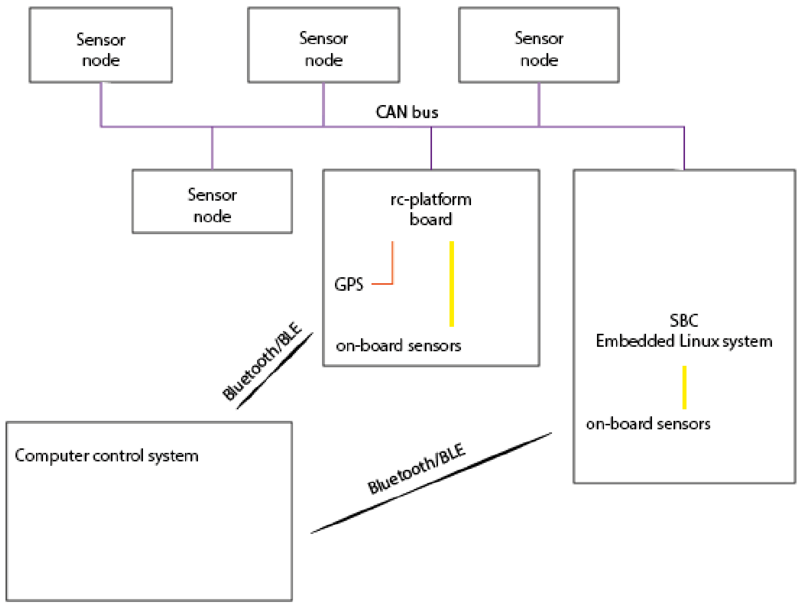
A WiFi connection (like 802.11n or 802.11ac) has a range of 10 - 100 meters and is easy to configure using standard tools in the operating system. Having a standard network connection on an Embedded Linux system will allow the use of a Socket connection for communication (primarily the transport layer), which will be discussed later on under software as it is more relevant there.

Bluetooth Low Energy

Bluetooth low energy has been developed to offer a low-energy alternative for radio communication. It has a similar range to WiFi with a nominal range of 100 meters, though there is also BLE Long Range/Coded PHY that has been tested with a range of up to 1300 meters [8].

2.1.7 Communication needs

In this project, there are several different communication needs. The microprocessor on the board needs to communicate with the different sensors on the board to read sensor data. The different boards/nodes need to communicate with each other, and to keep it modular it needs to be a bus where you can easily change what type and number of nodes are connected. For GPS a lot of the available modules communicate using UART and/or i2c. As cable between an autonomous vehicle and the control interface would be a limiting factor, it should have wireless communication.



Figur 2.6: Machine- and software platform

2.1.8 Discussion on communication choices

It would be wrong to say that one of these types of communication is simply better than the other, but they have different use cases. I2C is designed for, and very suitable for inter-circuit communication within a board, but would quickly perform badly because of noise if used between boards and on longer distances. It needs only two GPIO pins (SDA/SCL), and it is possible to have multiple targets on the same line. It is a popular interface used on a lot of sensor ICs. This makes it a good choice for communication with sensors on the RC platform board, and also for possibly connecting a couple of sensors directly to the SBC Linux Embedded device (can also be done with a cape/pi-hat board).

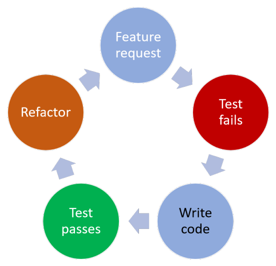
CAN bus is not a single controller bus, it uses differential lines, and support longer distances. It was developed for and is the industry standard for communication in cars. The bus has a system for message priority where critical frames win arbitration and go before less important frames. And nodes can use filters on incoming frames to avoid the CPU handling interrupts due to frames not being relevant for their functionality. These features make it a good choice for communication between nodes.

WiFi was chosen as the main focus mainly because it needs little setup/configuration, it is widely available on SBC devices directly or using a USB peripheral. Devices can connect to existing infrastructure/access points or WiFi hotspots. Most Android/iPhones can easily create a local hotspot, or in a larger test site, it is possible to connect to a larger network with one or more access points and larger coverage.
The negative side of microcontrollers using wifi directly is the high power consumption, and the space it takes to implement network and IP stack support on smaller embedded devices. A Zephyr RTOS device could however use an ESP32 as a modem for wireless connections, especially if using a driver that offloads part of the network stack. For a stand-alone RC platform board setup, BLE/Bluetooth uses far less power, making it a viable alternative for wireless communication between the RC platform board and a computer. And it can be used for both OTA flashing of firmware and transmitting commands/sensor data.

2.2 TDD - Test Driven Development and Testing

Test-driven-development focus on converting the requirement to test cases before starting on the implementation. Writing the test case before writing the code keeps the tests from being tied to the specific code implementation, and makes sure it is tied to testing the required functionality instead. The steps as described on Wikipedia [7] and Uncle Bob (Clean Code [12]) are:

- Add a test
- Run all tests. The new test should fail for expected reasons
- Write the simplest code that passes the new test
- All tests should now pass
- Refactor as needed, using tests after each refactor to ensure that functionality is preserved
- Repeat



Figur 2.7: TDD Sequence

Following these steps, you will end up with both a fully tested codebase, and know for sure that your refactoring of the code does not break anything. The goal is to avoid having to debug, as that is often very time-consuming. Code with a test suite also helps ensure that library upgrades/changes, and later changes, refactoring or feature additions to the project code do not unintentionally introduce bugs. A common type of test is a "Unit test". A Unit test is a test for a unit of code, testing a single unit of functionality. On a higher level, there are integration tests, testing functionality across different units. To create unit tests that do not rely on an external dependency it is possible to use "mocking"where you make a pretend interface to test against.

As an example, if you want to test code related to interacting with socket communication you could create a mocking socket object where it doesn't actually interface with a network connection but rather just pretend to do so. That way you could let the mock object store connections attempted, data being sent with the socket, or hand out data, and let the Unit test system check input given to the mock object.

Working with TDD and version control platforms you can create a pipeline to run Unit tests etc on pushing new commits to the tree, or on pull requests or releases. Often a test can also be good documentation of functionality and an easy place for programmers to read and understand how to use a library or function [12].

Writing good tests, learning to use different test frameworks, mocking, and fakes definitely require effort. Learning to write tests and testable code is not always as straightforward in practice as it seems when you read or hear about it. One of the interesting questions is what value it adds, and how it affects the resulting code.

TDD also seems quite a bit more straightforward for cod libraries and functions doing different calculations, than in embedded development interacting with hardware GPIO pins, and external systems. It is possible to mock these interfaces, but would that give the same value? As an alternative, it is also possible to have a microcontroller connected to an Embedded Linux system and run tests where the chip is programmed and the logical level of GPIO pins checked in different states of the firmware. As well as automated testing it is also possible to create documentation for manual testing describing how to perform the tests and the expected results. However, doing manual testing is quite costly in terms of time and will most likely not be run as often, and more changes to the code between the tests.

2.3 Firmware/Software

When working in development it is important to have good tools for the job. Here is an introduction to some of the key software tools for this project.

2.3.1 STM32Cube

STM32Cube [6] is a complete ecosystem with libraries and tools for working with STM32 microcontrollers. The STM32Cube IDE is based on the Open Source Eclipse and GNU C/C++ toolchain and offers capabilities to write code, compile code, debug and write/verify devices.
The development, debugging and programming of the STM32 (sensor-can-node) boards have been done using the STM32Cube IDE, and libraries.

2.3.2 VSCode

VSCode [9] is an extensible IDE with plugins to extend it to work with a huge variety of programming languages and use cases. It has built-in debugging capabilities and works seamlessly with GitHub and several unit testing systems.

VSCode has been used for the development and debugging of the Python code, and the Zephyr RTOS-based firmware. It also supports remote debugging with GDB. The built-in GitHub integration gives you direct access to information and management of issues, commits, pull requests, and lets you see who (and when) has made different changes in the code. For Unit testing, using pyTest [5] it will let you run the test by simply clicking test, and show you the status of the tests with green marks in the test file for tests that pass. When building code it parses the compiler output and tries to give hints and directions with red lines and mouseover text directly in the source code.

2.3.3 Zephyr RTOS

Zephyr RTOS Project website

[14] Real-time operating systems for embedded systems offer support for multiple boards, devices, and drivers. They can simplify a lot of the work when building connected devices sending and receiving data as they can handle a lot of the low-level work. The abstraction offered when developing firmware based on a RTOS means you can keep a lot of the code when switching components. You just have to redefine the board dts and config files to describe the new hardware and ensure the new hardware has any necessary device or GPIO capability. There is a broad number of device drivers available with Zephyr, and you can also build your own driver if a piece of hardware you are using is not already supported. You can define, compile and run the firmware using Posix for testing/debugging. And, it has a testing framework for writing Unit tests. At the time of writing the current LTS version of Zephyr is 2.7.0 (Long term support).

Figur 2.8: Zephyr RTOS logo

2.3.4 Python

Python website

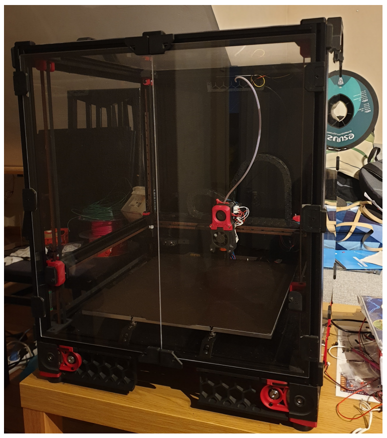
[14] Python is a high-level programming language available for all the major platforms, and it is suitable for rapid development. There are bindings for the Qt framework for GUI, and a lot of libraries available through pip. It is possible to use a Dependency management system like Poetry to keep track of libraries and dependencies to make distribution and deployment easier. It is possible to use the pyTest framework for Unit testing.

Figur 2.9: Python logo

2.4 3D printing and laser cutting

Vorondesign Project website

Working on rapid prototyping projects both 3D printers and laser cutters can be great assets. 3D printing is a type of additive manufacturing and refers to manufacturing primarily by building layer by layer. Two very common and available 3d printing technologies are SLA and FDM. SLA [11] or Stereolithography refers to printing by exposing a resin to light from a screen to build a model layer by layer. FDM [10] or Fused Deposition Modeling (also referred to as FFF, Fused Filament Fabrication) is a printing technology where the printer layer-by-layer extrudes filament to build the model. There is a wide range of available materials that can be used, and each of them with specific physical properties making them more or less suitable, as well as different environmental impacts. However, 3D printing wastes less material being an additive manufacturing technique compared to subtractive manufacturing methods both because you only print the structure you want to keep (and in some cases some support structures), and because you will often use only 10 - 40% infill in the parts. Print quality, printing capabilities, print speed, and material compatibility depend on the specific printer.


Figur 2.10: Voron 2.4r2 printer used for this project

Laser cutters work on flat surfaces/objects by using a laser to cut through or engrave on the object. You can plan for three-dimensional objects by making slots and puzzling together a set of boards. It is a very fast method for cutting out parts, and it works very well on acrylics and MDF. For safety, it is important to make sure not to use materials that will not melt, or make toxic fumes when cut by the laser cutter.

When planning for 3d printing or laser cutting a lot of models can be found using online resources like Thingiverse or Grabcad. Or designed using CAD software like Fusion 360. Most PCB design software can also export a 3D model of the finished board (including components), allowing you to import it to create a digital twin of your project. The CAD software can provide you with model files, technical drawings, and simulations that you can use when preparing for production, and use in production.

Kapittel 3

Design and construction of the hardware, firmware, software and physical components

3.1 Single Board Computer - Embedded Linux

Single Board Computers have a central role in this project as a tool, and platform in development, automated testing, and as a part of the system. Because of limited availability, and slight differences in specifications and use cases, both BeagleBone and Raspberry Pi boards have been in the scope of the project. Is one of them a clearly better choice than the other?

3.1.1 Raspberry pi

The Raspberry Pi 3+ and 4 boards used to be fairly cheap, there are a lot of different expansion pi-hat boards, and you can find a lot of system images you can use as they are. That made it a very popular board for a lot of hobby projects, and smart house systems. Over the last year or so there has been a shortage of Raspberry Pi boards on the market, making it difficult to find them in stores, and the second-hand prices for them increased a lot.

And, even if you have one, the SBC is not without shortcomings. Using a MicroSD card as the boot and main storage drive is known to eventually cause disk errors, it does not work well with other Linux distributions than the Raspberry Pi OS, the GPIO pins are somewhat limited, and more importantly for this project: it does not have a CAN bus controller.

It has a lot to offer as an affordable option, with low power consumption, and a fairly fast CPU, especially on the Raspberry Pi 4. It has built-in WiFi and Bluetooth, an onboard connector for a raspberry pi camera, and for a screen, and four USB ports. The Raspberry Pi OS has a fairly good library of available software packages. And it is easy to find information and examples on a lot of topics and uses for it online.

3.1.2 BeagleBone

The BeagleBone boards are made as Open Hardware and come in several different flavors. They run well on Debian for Arm, and there is a lot of information available in online communities for them. They are less focused on features like HDMI output or being used as a personal computer. But feature a lot more interesting Hardware when you look at the GPIO side of it. With the Programmable real-time unit and industrial communications subsystem (PRU-ICSS) you have a fast 200 Mhz processor with direct access to a number of pins, and access to the internal memory and peripherals of the board.

The BeagleBone family features the tiny PocketBeagle with a 1 Ghz CPU, 2x PRU 32-bit microcontrollers, and a Click board-compatible header with room for two click boards facing opposite directions. The larger BeagleBone Black (or BeagleBone Black WiFi), and SeedStudio BeagleBone Green (and BeagleBone Green Wifi). For AI purposes there is also the BeagleBone AI with DSP cores, and embedded vision engine cores available through an optimized TIDL machine learning API.

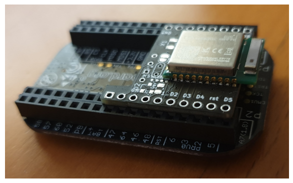
3.1.3 Conclusion

As it is for this project's purposes possible to go either way. Designing to make it possible to use any of them seems to be a good choice. That requires a slightly different approach for the CAN bus as the BeagleBone boards have a CAN bus driver, while the Raspberry Pi does not. But it is easy to overcome that obstacle by using an SPI CAN driver. For most purposes, both boards can be used, and the Python scripts will run fine as long as the CAN bus interface is configured.

3.2 PocketBeagle, a development tool

Embedded Linux repository

3.2.1 Debugging


Figur 3.1: PocketBeagle with e73 Click Board

By programming a PRU-SWD interface in assembly (as the PocketBeagle has the PRU-ICSS CPU), and by slightly patching and configuring OpenOCD for use on the PocketBeagle (Based on OpenOCD patched for Zephyr from https://github.com/zephyrproject-rtos/openocd). The PocketBeagle can be used as an SWD programmer or a remote GDB debugger for Zephyr. Through VSCode you can attach to the remote GDB session running on the PocketBeagle. A deb package with the configured OpenOCD (including the PRU-SWD binary file is located in the Embedded Linux repository) together with production files for a PocketBeagle Cape for connecting an e73 chip (NRF52840-based microcontroller) and a WiFi module (connected to the PocketBeagle USB pins), and another PocketBeagle Cape for connecting only an e73. The two of them can currently not be connected at the same time due to physical constraints. It is however possible to connect to the PocketBeagle through a network connection when it is connected by USB, and also to share an internet connection with the PocketBeagle through the computers (Also documented in the Embedded Linux repository).

3.2.2 Automated testing with Zephyr

As described in the Zephyr documentation their Unit testing framework also works with pyTest. And having a microcontroller connected to an Embedded Linux system makes it possible to run automated testing to evaluate GPIO signals, and interact with the microcontroller on I2C, CAN bus, etc. The same board used for debugging could also be used for this. Due to limited time, this was left as a theoretical possibility and not completed.

3.3 SBC system

Familiarity with flashing a system image to a MicroSD card, using basic Linux terminals, connecting to an SSH server, basic network knowledge, and being able to use the Raspberry pi pin-out to connect an SPI device is expected to follow along with the guides to set up the SBC system. But, there is a guide for settings and usage for both Raspberry Pi and BeagleBone, including a description of example hardware. And a link to a page where you can find design files for a BeagleBone CAN bus Cape.

SBC CAN bus guide on GitHub

This setup is the prerequisite to get the CAN network interface used by the Python scripts, and it enables the SBC to be used for testing CAN communication by using candump and cansend to listen to messages and send messages on the CAN bus. A couple of things worth noting:

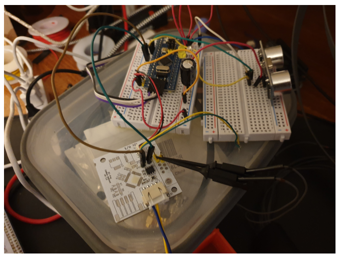
- If there are no other nodes on the CAN bus sending an ACK, the CAN network on the SBC gets unhappy. This can be fixed by restarting the network.

- When you have a functional CAN network on the SBC the Python scripts will not know any difference between running on a Raspberry Pi or BeagleBone as they use a library communicating with the CAN network interface.
- The guide currently does not have the required steps to configure the CAN bus CAPE for BeagleBone AI boards as that requires also compiling a uBoot overlay file for am572x. If you try to use the overlay for am335x that will cause it to freeze.

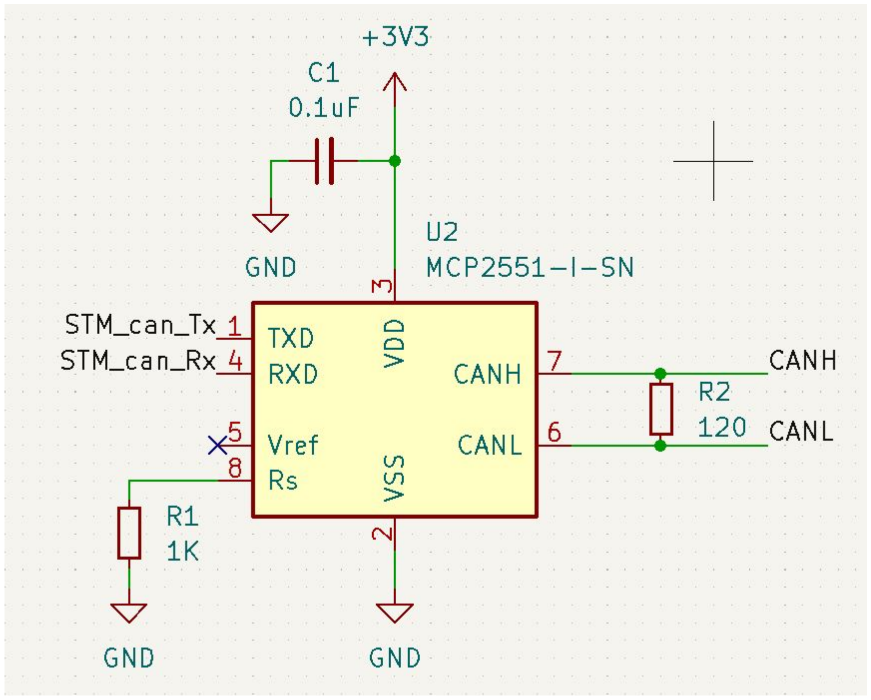
3.4 CAN-Sensor-Node

3.4.1 PCB design

STM32 CAN Node PCB repository


Figur 3.2: PocketBeagle with e73 Click Board

The first revision, rev 1.0, of the CAN sensor board was not functional due to a couple of design errors. After production several flaws became apparent, and steps were taken to get a working prototype with basically the same hardware. When ordering the CAN transceiver chips from China, a couple of STM32 Blue Pill boards were also purchased. This was done as a redundancy as the existing development experience with STM32 chips and CAN bus was limited, at best. The Blue Pill dev board has the same family STM32 chip that is also used on the CAN-Node-PCB. After realizing the flaws in the rev 1.0 design of the board, a Blue Pill dev board was connected with the CAN transceiver on the prototype CAN sensor node board using the transceiver test pins, and to two LEDs and a distance sensor on a breadboard. It is not a very elegant solution but it is a solution that allows for developing and testing both hardware and firmware before ordering a new revision of the board.



Figur 3.3: STM32 CAN transceiver schematic

Through the design process, the CAN transceiver chip was changed between different pin-compatible options while checking the availability on RS-Online. While changing back and forth the choice landed on the MCP2551, as that was the chip also used on the MCP2515 CAN driver module. Later on the datasheet revealed that V_DD on the MCP2551 should be between 4.5 V and 5 V, not as low as 3.3 V. The other design issue was that the STM_canTx label had been misplaced on the STM32 chip routing it to the incorrect pin. The correct connection on the STM32F103C8Tx would be:

- PA12 (CAN_TX) connected to U2 pin 1 (TxD)
- PA11 (CAN_RX) connected to U2 pin 4 (RxD)
- PD0 and PD1 should have an 8 Mhz crystal. (this is correct in rev 1.0)

Note also the possibly unintuitive, but correct, the connection from CAN-

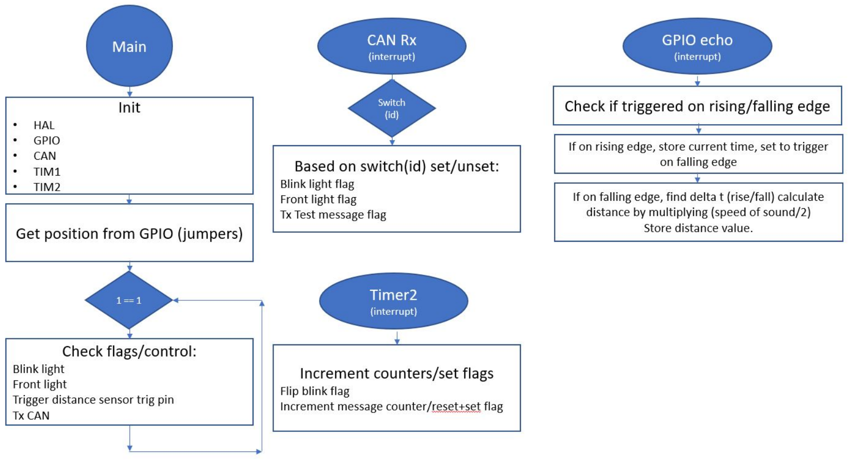
Tx to Tx and CAN-Rx to Rx. As there will be a new revision of the board anyway several modifications have been suggested as well while investigating possible ways to use the board on a mini-car system.

- Include a connector for an HC-SR04 sensor (and 5 V LDO for it, and voltage divider for the echo pin)
- Add two jumpers for assigning sensor position (front/back, left/right)
- Add two or three pins connected to N-channel Mosfets (with pulldown on the gate) to control external lights (at least for signal light and brake light).

Besides these comments, there is not much to note about the sensor node PCB as it was a fairly small and simple design, and it is meant to have a small and limited purpose. The initial idea was to create a generic sample PCB, where that design could be easily modified for specific use cases before running production. After working on the project for while, creating a more specific sensor-node-PCB for a single distance sensor and external light control seems to be a better approach.

3.4.2 Firmware

STM32 CAN Node firmware repository
The STM32 CAN Node firmware is a very basic test firmware with CAN bus communication, distance sensor reading, reading two GPIO pins to set a position, and controlling two separate LEDs (front light and signal light). It has an example of parsing incoming packages for commands/settings, and it can send a test message on the CAN bus (data that can be used when working on CAN software on the SBC).

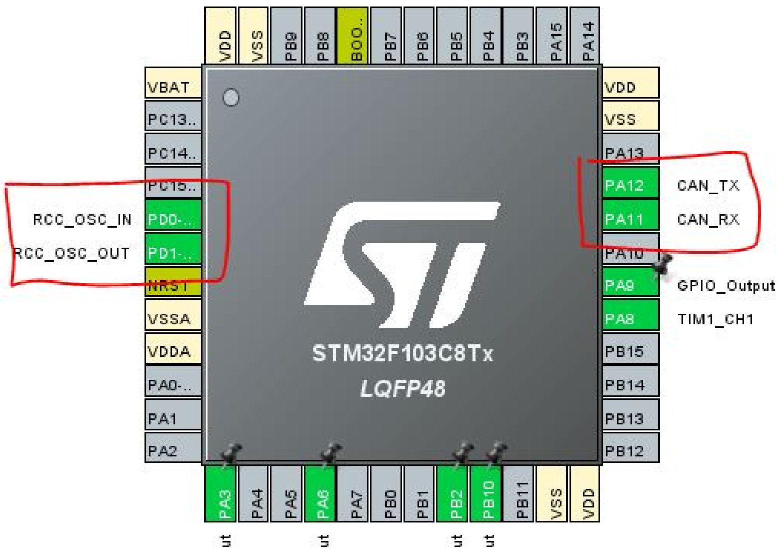

Main
Init
HAL
GPIO
CAN
TIM1
TIM2
Get position from GPIO (jumpers)
1 == 1
Check flags/control:
Blink light
Front light
Trigger distance sensor trig pin
Tx CAN
CAN Rx
(interrupt)
Switch
(id)
Based on switch(id) set/unset:
Blink light flag
Front light flag
Tx Test message flag
Timer2
(interrupt)
Increment counters/set flags
Flip blink flag
Increment message counter/reset+set flag
GPIO echo
(interrupt)
Check if triggered on rising/falling edge
If on rising edge, store current time, set to trigger on falling edge
If on falling edge, find delta t (rise/fall) calculate distance by multiplying (speed of sound/2)
Store distance value.

Figur 3.4: Program flow of the CAN Sensor node firmware

The current program flow in the firmware is very simple. A timer was initially used as an interrupt for the timed event of flipping the blinking light on/off and given a time interval suitable for that. And, the idea was to also add a SysTick interrupt to set flags for timing sensor readings, messaging, and blinking. And using preprocessor macros to store values for the different intervals making them easy to modify. #define macros have been added to set the various pin numbers for different functionalities, and a set of #define macros have been added to Inc/defines.h for values shared across platforms. This file is used to make it easier to synchronize the set of relevant id numbers and static values used in the project.

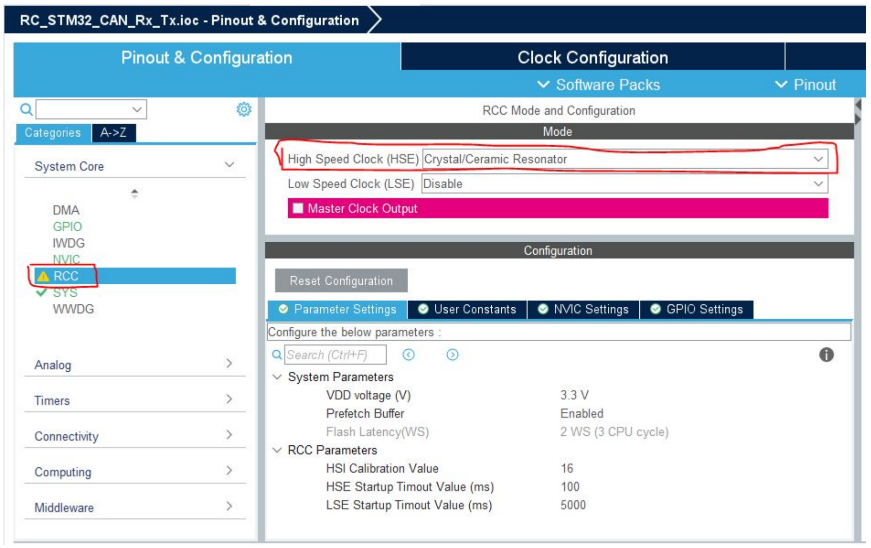
Guide to setting up CAN bus in STM32Cube

Start by opening the .ioc file with the project settings.



Figur 3.5: Pinout view

Change the PD0 and PD1 pins to RCC_OSC_IN/OUT (this is where the 8 Mhz crystal is connected). And Change PA11/PA12 to CAN_Rx and CAN_Tx.



RC_STM32_CAN_Rx_Tx.ioc - Pinout & Configuration

Pinout & Configuration | Clock Configuration | Software Packs | Pinout

Categories | A->Z

System Core: DMA, GPIO, IWDG, NVIC, RCC, SYS, WWDG

Analog, Timers, Connectivity, Computing, Middleware

RCC Mode and Configuration

Mode

High Speed Clock (HSE)	Crystal/Ceramic Resonator
Low Speed Clock (LSE)	Disable

Master Clock Output

Configuration

Reset Configuration

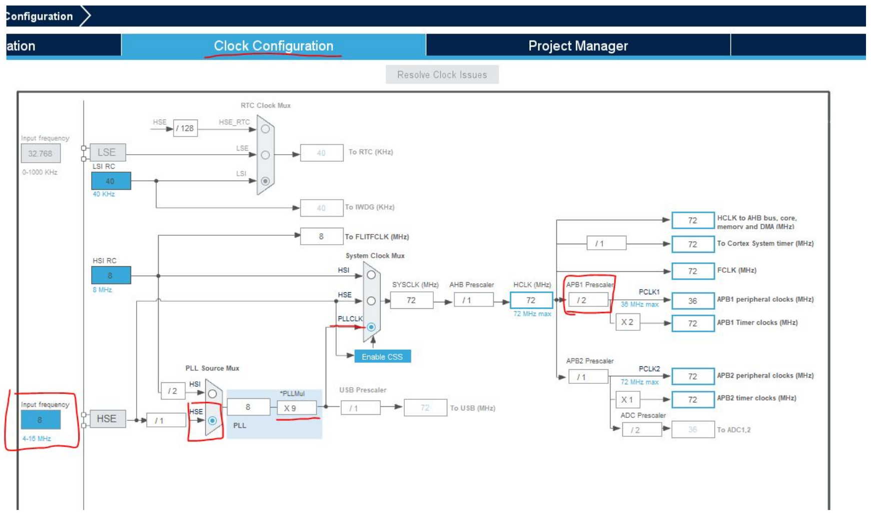
Parameter Settings | User Constants | NVIC Settings | GPIO Settings

Configure the below parameters :

System Parameters	
VDD voltage (V)	3.3 V
Prefetch Buffer	Enabled
Flash Latency(WS)	2 WS (3 CPU cycle)
RCC Parameters	
HSI Calibration Value	16
HSE Startup Timout Value (ms)	100
LSE Startup Timout Value (ms)	5000

Figur 3.6: Reset and Clock Control settings

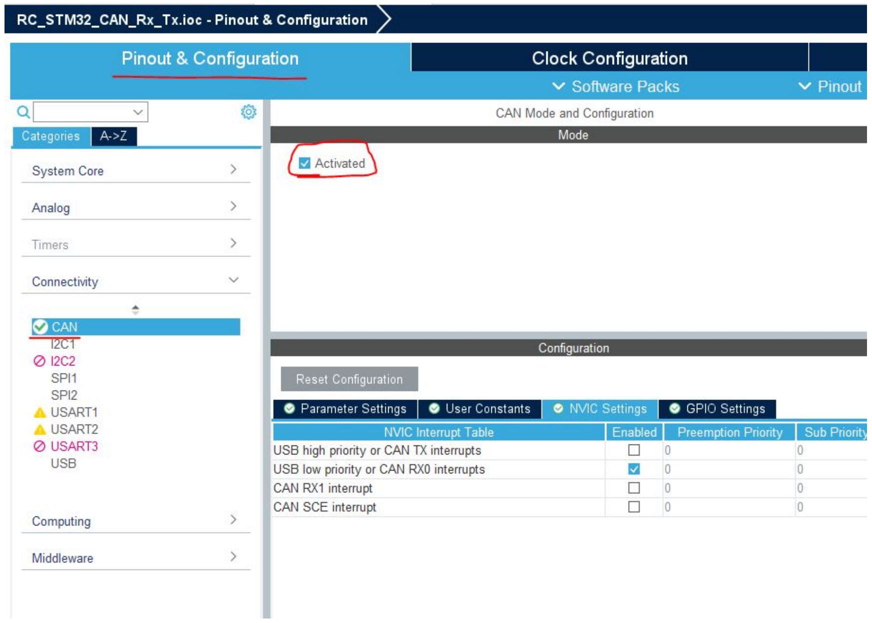
Go to the System Core->RCC (Reset and Clock Control) settings, and change the "High Speed Clock (HSE)to Crystal/Ceramic Resonator to configure the controller to use the external high-speed oscillator.



Figur 3.7: Clock config

Change tab to "Clock config"and change the following values (as also shown in figure 3.7:

- Input frequency is 8 Mhz (matching the external oscillator)
- HSE is selected in PLL Source Mux
- Change PLLMul to X9
- Select PLLCLK in System Clock Mux
- Change APB1 Prescaler to /2

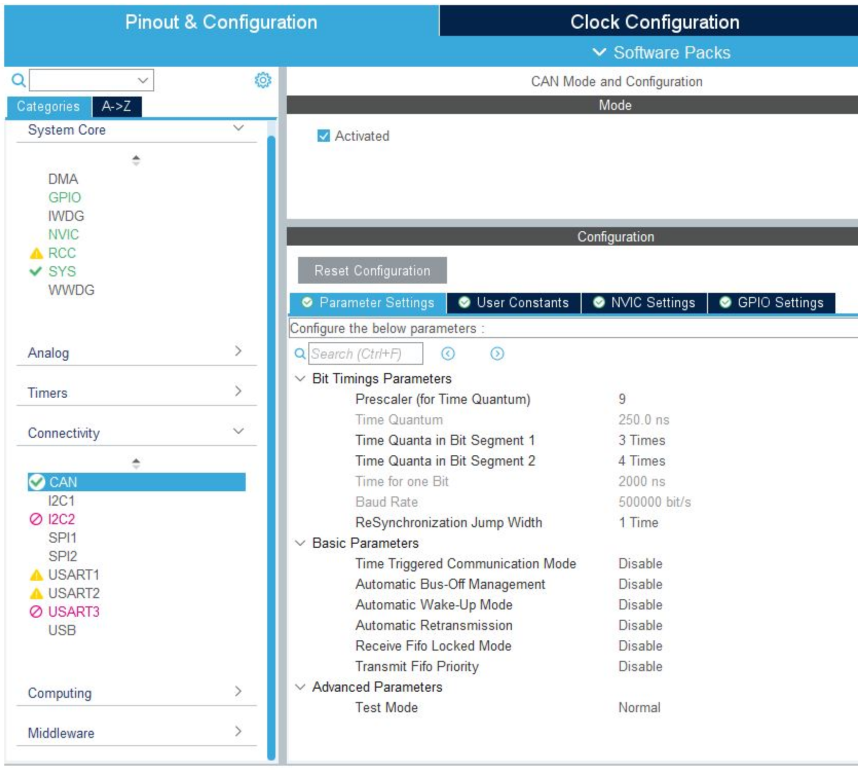


☑ Activated

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
USB high priority or CAN TX interrupts	☐	0	0
USB low priority or CAN RX0 interrupts	☑	0	0
CAN RX1 interrupt	☐	0	0
CAN SCE interrupt	☐	0	0

Figur 3.8: Activate CAN

Go to the pinout and configuration tab. In connectivity, choose CAN. Check the box underneath "modeto activate CAN.



Activated

Configure the below parameters :

Bit Timings Parameters

Prescaler (for Time Quantum)	9
Time Quantum	250.0 ns
Time Quanta in Bit Segment 1	3 Times
Time Quanta in Bit Segment 2	4 Times
Time for one Bit	2000 ns
Baud Rate	500000 bit/s
ReSynchronization Jump Width	1 Time

Basic Parameters

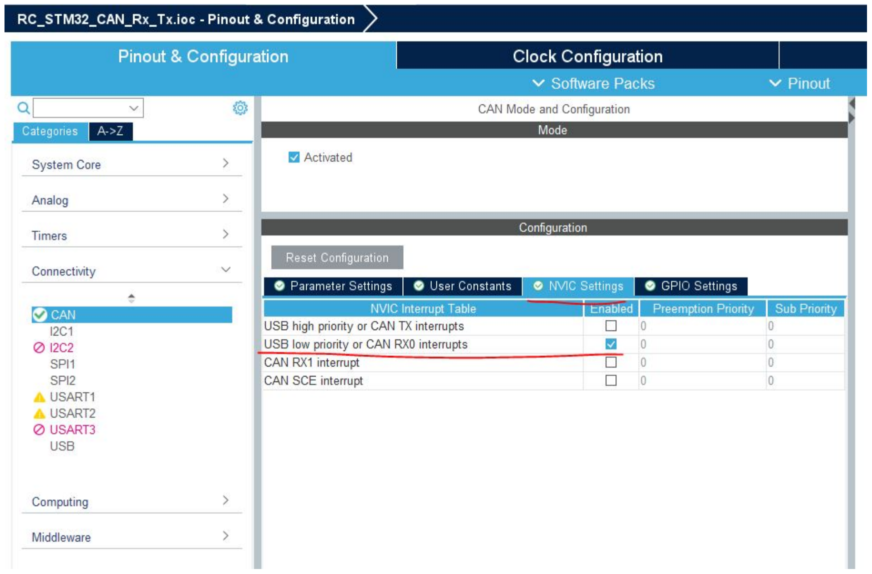
Time Triggered Communication Mode	Disable
Automatic Bus-Off Management	Disable
Automatic Wake-Up Mode	Disable
Automatic Retransmission	Disable
Receive Fifo Locked Mode	Disable
Transmit Fifo Priority	Disable

Advanced Parameters

Test Mode	Normal

Figur 3.9: Configure the CAN bus

- Using the set prescaler as 9, giving a Time Quantum of 250.0 ns.
- Time for 1 bit = Tq + (TQ* TQ in bit segment 1") + (TQ * TQ in bit segment 2")
- Time Quanta in Bit Segment 1 = 3 Times
- Time Quanta in Bit Segment 2 = 4 Times
- 250 + (250*3) + (250*4) = 2000 ns
- Gives a baud rate of 500 kbps



✓ Activated

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
USB high priority or CAN TX interrupts	☐	0	0
USB low priority or CAN RX0 interrupts	✓	0	0
CAN RX1 interrupt	☐	0	0
CAN SCE interrupt	☐	0	0

Figur 3.10: Activate CAN rx interrupt

Finally, we head over to NVIC Settings and check USB low priority or CAN RX0 interrupts to enable CAN Rx interrupts. Saving this configuration will trigger quite a bit of code to be generated in our project to reflect the configuration we did.

First, we add some variables for RX and Tx CAN headers, a can data buffer, filter, and mailbox.

```
    CAN_RxHeaderTypeDef rxHeader;          //CAN Bus ...
        Transmit Header
    CAN_TxHeaderTypeDef txHeader;          //CAN Bus ...
        Receive Header
    uint8_t canRX[8] = {0,0,0,0,0,0,0,0};  //CAN Bus ...
        Receive Buffer
    CAN_FilterTypeDef canfil;              //CAN Bus Filter
    uint32_t canMailbox;                   //CAN Bus Mail ...
        box variable
```

Next, we add code in main(), before the loop, to fill the filter and Tx header with data. We can let the Filter be 0, 0 now to allow all frames through. And the last lines will insert the filter, start the CAN bus and activate notifications.

```
    canfil.FilterBank = 0;
    canfil.FilterMode = CAN_FILTERMODE_IDMASK;
    canfil.FilterFIFOAssignment = CAN_RX_FIFO0;
    canfil.FilterIdHigh = 0;
    canfil.FilterIdLow = 0;
    canfil.FilterMaskIdHigh = 0;
    canfil.FilterMaskIdLow = 0;
    canfil.FilterScale = CAN_FILTERSCALE_32BIT;
    canfil.FilterActivation = ENABLE;
    canfil.SlaveStartFilterBank = 14;

    txHeader.DLC = 8;
    txHeader.IDE = CAN_ID_STD;
    txHeader.RTR = CAN_RTR_DATA;
    txHeader.StdId = CAN_TEST_MSG_ID;
    txHeader.ExtId = 0x02;
    txHeader.TransmitGlobalTime = DISABLE;

    HAL_CAN_ConfigFilter(&hcan,&canfil);
    HAL_CAN_Start(&hcan);
    HAL_CAN_ActivateNotification(&hcan,CAN_IT_RX_FIFO0_MSG_PENDING);
```

And then add the function called by the CAN receive interrupt. This can be placed in USER CODE 4. This function will be called when there is incoming data on the CAN bus.

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef ...
    *hcan1)
{
    HAL_CAN_GetRxMessage(hcan1, CAN_RX_FIFO0, &rxHeader, ...
        canRX);
    switch (rxHeader.StdId) {
    case CAN_LIGHT_CONTROL_ID:
        if (canRX[0] == 0) {
            flag_blink_light ^= 0x01;
        }
        break;
    case CAN_BLINK_CONTROL_ID:
        if (canRX[0] == 1) {
            light_status = 1;
        } else {
            light_status = 0;
        }
        break;
    case CAN_DEVICE_SETTINGS_ID:
        if (canRX[0] == Placement) {
            if (canRX[1] == 1) {
                flag_send_test_msg = 1;
            } else {
                flag_send_test_msg = 0;
            }
        }
        break;
    }
}
```

Code for interrupt with timer

The other interesting piece of code is the code for the HC-SR04 distance sensor as which uses interrupts with a timer on the echo pin. First, a trig signal is sent on the trig pin for 10 us, and htim1 is enabled.

```
void HCSR04_Read (void)
{
    HAL_GPIO_WritePin(TRIG_PORT, TRIG_PIN, GPIO_PIN_SET);
    delay(10);  // wait for 10 us
    HAL_GPIO_WritePin(TRIG_PORT, TRIG_PIN, GPIO_PIN_RESET);
    __HAL_TIM_ENABLE_IT(&htim1, TIM_IT_CC1);
}
```

When the echo pin is triggered with a rising flank the capture callback is triggered, and it will check a flag to see if it is the first or second time it has been triggered (rising or falling edge). This will be the first trigger for this distance reading and running the following code:

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) {
    if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)  {
        if (!flag_hcsr04_first_captured) {
            IC_Val1 = HAL_TIM_ReadCapturedValue(htim, ...
                TIM_CHANNEL_1);
            flag_hcsr04_first_captured = 1;  //
            __HAL_TIM_SET_CAPTUREPOLARITY(htim, ...
                TIM_CHANNEL_1, ...
                TIM_INPUTCHANNELPOLARITY_FALLING);
```

This stores the value captured by the timer HAL_TIM_ReadCapturedValue, sets the flag to show that the first value has been captured, and changes the polarity of the interrupt. When the echo signal falls again, the timer will trigger once more. And the remaining code in the function will run (because of the flag).

```
    } else if (flag_hcsr04_first_captured) {
            IC_Val2 = HAL_TIM_ReadCapturedValue(htim, ...
                TIM_CHANNEL_1);
            __HAL_TIM_SET_COUNTER(htim, 0);
            if (IC_Val2 > IC_Val1) {
                Difference = IC_Val2-IC_Val1;
            } else if (IC_Val1 > IC_Val2) {
                Difference = (0xffff - IC_Val1) + IC_Val2;
            }
            hcsr04_dist = Difference * .034/2;
            flag_tx_sensor_data = 1;
            flag_hcsr04_first_captured = 0;
            // set polarity to rising edge, in case it was ...
                changed
            __HAL_TIM_SET_CAPTUREPOLARITY(htim, ...
                TIM_CHANNEL_1, ...
                TIM_INPUTCHANNELPOLARITY_RISING);
            __HAL_TIM_DISABLE_IT(&htim1, TIM_IT_CC1);
        }
    }
}
```

What remains is to record the second value from the timer. Calculate the difference between them. Multiply delta t with the speed of sound divided by two (the ultrasonic sound travels out to an object, bounces and returns). Set a flag that there is a new distance measurement available, reset the first capture flag, reset the polarity of the interrupt and disable the timer.

3.4.3 Use case for the sensor node

The sensor node board can easily be developed to serve several functions on a small car system. It can handle regular distance measurements at multiple locations (by adding multiple boards) and can be connected to control signal lights, brake lights, and front lights. Controlling the lights might not be useful for working on autonomous driving with computer vision, but it serves a purpose by making the model more realistic, and it can give immediate visual feedback on actions planned by the vehicle (stopping or changing direction). Implementing other or more sensors can be done by further developing the firmware, and adding respective channels to the definitions files on the various systems.

3.4.4 Further improvements

Adding support for an i2c accelerometer to the firmware might be useful, as that could give acceleration data from all four corners of the vehicle. The CAN bus sending code should use a bitwise "or"operation on the id with the position to make sure all ids are unique. And the code used for parsing received data should similarly check if a message to activate a signal light has an id corresponding with the left or right side (matching its own position). The id set should also be expanded with useful id numbers for a variety of control functions and sensor logs. The id numbers should be configured in such a way that the most important/critical messages have the lowest id, and the least important is left with the highest id values, to make sure the important messages will win arbitrations. The sensor nodes could also filter irrelevant messages to avoid unnecessary interrupts.

3.4.5 Conclusion

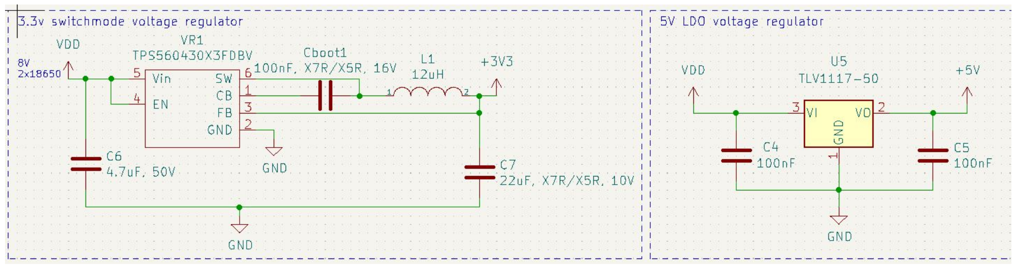
The firmware works well as a sample, and testing firmware. It works as a proof of concept, and there are only a few steps left to integrate it into a small autonomous car platform as conceptualized in the introduction.

3.5 RC Platform Board

The first revision of the concept was done using an atmega328p microcontroller, a L298N motor controller, and a HC-SR04 distance sensor. A wireless PS2 controller was added for controlling the RC-car (also giving affecting the name), even though the idea was to work on control theory and computer vision. Starting with the bachelor thesis the ambitions and requirements were put to a far higher level, especially when challenged to implement CAN bus for communication.

3.5.1 PCB

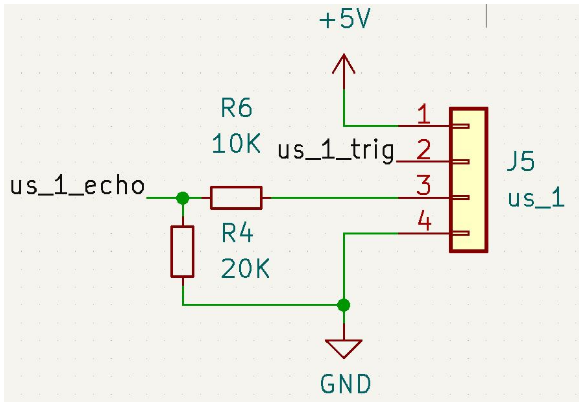
RC-feather-platform-board
The current PCB is very simple but also works very well for its purpose. As it builds on a microcontroller dev board with a feather connector, relies on i2c for most of the sensors, and has one UART device (GPS), and one SPI device (CAN bus) the schematic turned out to be fairly simple. The need for both 3.3 V and 5 V made one interesting decision to be made. The decision landed on making a switchmode 3.3 V voltage regulator, and a 5 V LDO voltage regulator. The reasoning behind this was that it is primarily the distance sensors (if used) and the GPS (if even connected) that use 5 V, and the total current draw will be fairly limited anyway. Most of the board relies on 3.3 V, and even though the current draw there is also quite limited, a switchmode power supply is more efficient.



Figur 3.11: Voltage regulators

Another part worth noting on the PCB is the voltage divider for the HC-

SR04 distance sensor. It runs on a nominal 5 V Vcc, but will trigger on the 3.3 V logical signals from the Particle Xenon board. When the echo signal returns, the Particle Xenon cannot handle a 5 V input signal on the GPIO pins and has to have a voltage divider to lower the signal voltage. The maximum voltage for the Particle Xenon GPIO pin is 0.7 x VCC, 0.7x2,31 V. I have also found sources stating the minimum to be 2.1 V. With a voltage divider with a 20K and a 10K resistor, the input voltage should be 3.33 V.



Figur 3.12: Voltage divider for the distance sensors

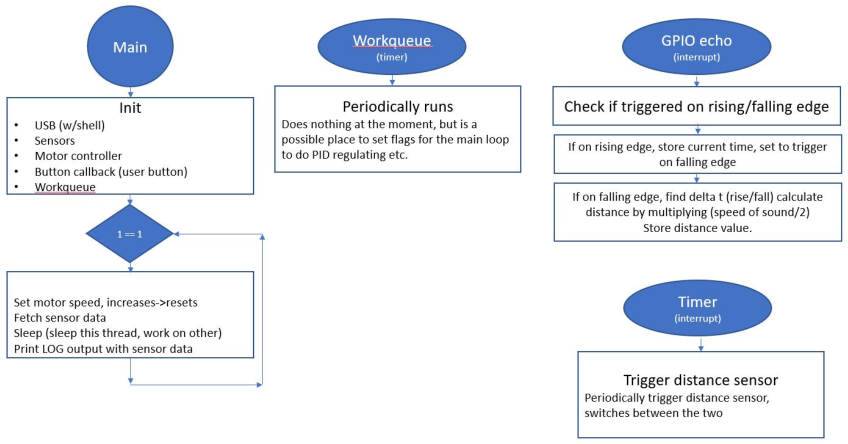
An almost identical design, with an e73 SMD chip, was created and manufactured at the same time as the feather connector one. After taking into consideration the difficulties of soldering that chip successfully the design was abandoned.

3.5.2 Manufactured parts

There is a set of model files for manufacturing the needed parts to assemble a small robot car with the RC-platform-board, a handful of sensors, a rolling ball wheel, and two geared motors. Assembling the robot requires mostly M3 screws, heat inserts, a bit of soldering, and making cables for the distance sensors. The parts can be produced in about an hour, the soldering can be done quite quickly even for people with little experience, and the parts are easy to assemble. Drawings and technical documentation of the parts can be found in the attachments at the end of the report. The model files are available on GitHub.

3.5.3 Zephyr RTOS firmware

RC-platform Zephyr firmware
The current firmware implements most of the functionality on the board. Here are some interesting pieces of code highlighting some of the differences when working with a RTOS compared to developing in C with STM32Cube and HAL. The advantages of using a RTOS would have been even greater when adding BLE and GATT, a network stack or other more advanced IoT functionality. As it is currently written the firmware will make sensor readings and output the data as logs to the console, and periodically run the motor driver at different speed levels in both directions. This is meant as a functionality test, and as a basis to create a small car with a distance regulator, or a larger setup combining the RC-platform board with CAN sensor nodes, and/or a SBC working on autonomous driving with computer vision.


Main
Init
USB (w/shell)
Sensors
Motor controller
Button callback (user button)
Workqueue
1 == 1
Set motor speed, increases->resets
Fetch sensor data
Sleep (sleep this thread, work on other)
Print LOG output with sensor data
Workqueue
(timer)
Periodically runs
Does nothing at the moment, but is a possible place to set flags for the main loop to do PID regulating etc.
GPIO echo
(interrupt)
Check if triggered on rising/falling edge
If on rising edge, store current time, set to trigger on falling edge
If on falling edge, find delta t (rise/fall) calculate distance by multiplying (speed of sound/2)
Store distance value.
Timer
(interrupt)
Trigger distance sensor
Periodically trigger distance sensor, switches between the two

Figur 3.13: Activate CAN

Particle Xenon Board config

Example of part of the board config (Kconfig based) for the Particle Xenon board in the Zephyr RTOS firmware. Here drivers for various bus and sensors are defined. These defined tags can also be used in #ifdef preprocessor flags making it possible to adjust the firmware depending on activated modules in the board configuration.

```
# CAN bus
CONFIG_SPI=y
CONFIG_CAN=n
CONFIG_CAN_MCP2515=n

# Sensors
CONFIG_SENSOR=y
CONFIG_I2C=y
CONFIG_BME280=y
## CONFIG_MPU6050_TRIGGER_NONE=n
CONFIG_VL53L0X=y
CONFIG_VL53L0X_PROXIMITY_THRESHOLD=100

CONFIG_CBPRINTF_FP_SUPPORT=y

# # GPIO
CONFIG_GPIO=y
CONFIG_PWM=y
CONFIG_PWM_NRFX=y

CONFIG_FPU=y
```

Particle Xenon Board overlay

The following code is from the Particle Xenon overlay file. That file adds to the board .dts file, and describes what is connected, and where it is connected. The pwm modules are listed, and you can see the channels referring to the GPIO pin linked to that channel. GPIO0 have pin numbers from 0 to 31 (0*32 + pin number), while GPIO1 has GPIO from 32-63 (1*32 + pin number). Underneath you can see the i2c definition, without any SDA/SCL pin definition. The pin definition for SDA/SCL on i2c0 is defined in the Particle Xenon dts files, but it is possible to override them in the overlay file, or leave it out to keep the predefined values. It is activated by setting status = "okay", and connected sensors are listed with their id. SPI1 was defined with pin numbers to configure it with suitable pins for the RC-platform-board layout. The mcp2515 SPI CAN driver has also been added to the board overlay, with suitable configuration for the BUS as also configured on the other nodes. The combination of adding the hardware to both the config, and the .dts/overlay files adds the device to the system device tree and makes it accessible in the firmware.

```
    &pwm0 {
     status = "okay";
     ch0-pin = <44>;
    };
    &pwm1 {
     status = "okay";
     ch0-pin = <42>;
     ch1-pin = <40>;
     ch2-pin = <43>;
    };
    &pwm3 {
     status = "okay";
     ch0-pin = <34>;
     ch1-pin = <33>;
    };

  &i2c0 {
     status = "okay";
     bme280: bme280@76 {
         compatible = "bosch,bme280";
         reg = <0x76>;
         label = "ENVIRONMENTAL_SENSOR";
     };
     vl53l0x: vl53l0x@29 {
         compatible = "st,vl53l0x";
         reg = <0x29>;
         // xshut-gpios = <&gpioc 6 GPIO_ACTIVE_HIGH>;
         label = "LASER_DISTANCE_SENSOR";
     };
  };
 &spi1 {
     status = "okay";
     compatible = "nordic,nrf-spi";
     sck-pin = <47>;   /* 32 + 15 */
     mosi-pin = <45>;  /* 32 + 13 */
     miso-pin = <46>;  /* 32 + 14 */
     cs-gpios = <&gpio0 4 GPIO_ACTIVE_LOW>;
     can1: mcp2515@0 {
         compatible = "microchip,mcp2515";
         spi-max-frequency = <1000000>;
         int-gpios = <&gpio0 3 0>; /* D2 */
         status = "okay";
         label = "CAN_1";
         reg = <0x0>;
         osc-freq = <16000000>;
         bus-speed = <125000>;
         sjw = <1>;
         prop-seg = <2>;
         phase-seg1 = <7>;
         phase-seg2 = <6>;
         #address-cells = <1>;
         #size-cells = <0>;
         sample-point = < 875 >; // ?
     };
 };
```

GPIO and PWM control

The high level of abstraction can also be seen in the code when interacting with hardware. Here is an example of interacting with GPIO pins, and adding callbacks. This code has the same functionality as the example from the CAN sensor node firmware when sending the TRIG pulse to the HC-SR04 distance sensor. In a situation where it was interesting to port this code to another hardware, the GPIO pins would be defined in the board/config file of the other board, while the rest of the source code would remain the same. And, after creating the new board files, it would be possible to compile the same project for both boards.

```
void send_trig() {
    trig_time = k_cycle_get_32();
    if (!cur_dist_sensor) {
        gpio_pin_set_dt(&trig_0, 1);
        k_sleep(T_TRIG_PULSE_US);
        gpio_pin_set_dt(&trig_0, 0);
        gpio_add_callback(echo_0.port, &echo_0_cb_data);
        LOG_DBG("Trig sent on %s pin %d", ...
            echo_0.port->name, echo_0.pin);
    } ...
```

Working with PWM signals is also quite straightforward. In the pwm_control.c file there is code to send PWM signals the motor controller using the int pwm_pin_set_cycles(structdevice *dev, u32_t pwm, u32_t period, u32_t pulse, pwm_flags_tflags) function. period and pulse are given in a number of clock cycles and are hardware specific, but there are functions to get a calculation in microseconds if the exact timing is important. Here it is set based on adjusting the pulse as a given part of the period:

```
    pwm_pin_set_cycles(pwm3_dev, MOTOR_1_0_PWM_PORT, 200, ...
        speed_value, 0);
    pwm_pin_set_cycles(pwm3_dev, MOTOR_1_1_PWM_PORT, 200, ...
        200, 0);
```

In sensor_reading.c there is functionality to initialize the sensors. Here is an example of code to initialize and test the BME280 environment sensor, if defined in the board config. The initialization function will output error logs on the interface configured to receive log output.

```
int init_sensors() {
#ifdef CONFIG_BME280
    dev_env = DEVICE_DT_GET_ANY(bosch_bme280);
    if (dev_env == NULL) {
        LOG_ERR("Error: BME280 not found.");
    }
    if (!device_is_ready(dev_env)) {
        LOG_ERR("Device BME280 is not ready, check the ...
            driver initialization logs for errors.");
    }
    LOG_INF("Found device BME280 getting sensor data\n");
#endif
```

SEGGER/RTT interface as a console

Using the J-Link programmer it is possible to connect to the device to receive a lot of useful information over RTT. RTT can be configured in the board config, or as an overlay file, and you can specify extra custom overlay files to use when running the build command. With that method, you can have special settings for debugging and log output that you compile with only when you are compiling for development or debugging, and you can simply use the RTT interface for accessing the logs, instead of UART/GPIO pins that might be in use for other purposes. Here are config options to enable the RTT backend:

```
CONFIG_RTT_CONSOLE=y
CONFIG_USE_SEGGER_RTT=y
CONFIG_LOG_BACKEND_RTT=y
```

Version control

To keep track of different versions of the firmware, and to make it easier to retrieve the source code they are built on the code in git_watcher.cmake"is designed to retrieve git information, and insert it into a header file src/headers/git.h". That header file has been imported in relevant modules of the firmware. This is triggered through code in "CMakeLists.txtresponsible for the build process.

3.5.4 BLE Over The Air firmware update

Zephyr MCUBoot bootloader repository
For the microcontroller device: compile and flash a bootloader using a suitable board file for that device. Particle Xenon works for the microcontroller used throughout this project, and it has built-in BLE.
Zephyr smp-svr sample project
Compile and flash a signed firmware of the smp_svr from the sample project. Make some changes to it, and prepare an additional firmware file (bin) that can be uploaded over Bluetooth.

As the SBC runs Linux, and has Bluetooth, I decided to use that to run mcumgr, but Windows/Mac/Linux computers can also be used for this.
To enable Bluetooth on the BeagleBone Green, and run bluetoothctl to scan for nearby devices run these commands:

```
sudo bb-wl18xx-bluetooth
sudo bluetoothctl
```

Then install Go, and download and compile mcumgr:
https://go.dev/doc/install
go install github.com/apache/mynewt-mcumgr-cli/mcumgr@latest

The following is an example of commands to use to upload a firmware bin file to a device using mcumgr over Bluetooth, and then list what is in the two firmware slots on the device (remember to get the firmware hash at this point), test if the new image is good, and then reset the device. By doing a listing after the reset you can confirm that the device now runs the upgraded firmware. Flashing firmware over Bluetooth typically takes several minutes.

```
sudo mcumgr --conntype ble --connstring ...
    ctlr_name=hci0,peer_name='Zephyr' echo hello
sudo ./mcumgr --conntype ble --connstring ...
    ctlr_name=hci0,peer_name='Zephyr' image upload ...
    /tmp/zephyr.signed.bin
sudo ./mcumgr --conntype ble --connstring ...
    ctlr_name=hci0,peer_name='Zephyr' image list
sudo ./mcumgr --conntype ble --connstring ...
    ctlr_name=hci0,peer_name='Zephyr' image test [hash of ...
    the firmware to test]
sudo ./mcumgr --conntype ble --connstring ...
    ctlr_name=hci0,peer_name='Zephyr' reset
sudo ./mcumgr --conntype ble --connstring ...
    ctlr_name=hci0,peer_name='Zephyr' image list
```

Flashing new firmware OTA to the device works, but is for development and testing purposes far too slow and takes too much effort to be a good solution. For devices and products where there are few updates, and infrequent firmware upgrades it would be a more interesting possibility. At this time it does not make sense to add this feature to the RC-platform-board, but it is useful to know of the possibility.

3.5.5 Further improvements

The RC-platform-board would be greatly improved as a stand-alone project base by setting up BLE and defining GATT for different sensor data readings. Creating drivers from the conceptual code for distance sensors and the motor driver would also improve on the code. The project is configured with "out-of-treedrivers, and it is possible to put custom drivers in a subfolder in the project. Currently, the implementation for the distance sensor is less than ideal as it works very differently from other connected sensors. I planned to spend more time getting familiar with pyTest and twister and creating tests for Zephyr, but in the end, I did not get to it.

3.6 Python software

The two Python programs written for this project have been made with Poetry as the dependency manager. Poetry could probably have been used far better than it is, as this is the first attempt on using it.

Besides the Python GUI class, each of the other written classes has an example of using it that you can test by running that python script, and the main applications start by running the __main__.py files. I also started implementing some tests using the pyTest framework and tried to find a good way of testing this project. My lack of experience with sockets and can bus limited my options on implementing them.
Even though the programs are said to be using Threads", the Python interpreter is in most cases not really multithreading as the Global Interpreter Lock (GIL) functions as a mutex protecting it.

The main goal of the Python programs is to be a sample of how Python can communicate over Socket, gather data from, and send data to the CAN bus. As well as being a nice test implementation showing as a proof of concept that data sent from a number of sensor nodes can be quickly streamed over CAN and socket and be shown in the Qt GUI of the control system. And, the concept of "Abstract classes"was used to allow for other server implementations"and data storage implementationsthan the ones already used.

3.6.1 Computer Control System

Computer control system
The computer control system is built similarly to the control system built in the computer design project in ELE340 [14]. It is currently not very feature-rich but has a basic Qt GUI design created in Qt designer showing a badly designed car shape from above. This can easily be exchanged by modifying the Qt designer file or creating a new one with controllers with the same name. The program itself starts up, checks the input arguments given, and defaults to standard arguments if not given as input arguments. And then, launches the Qt GUI. The GUI creates a socket client object, and a data_storage object. And spawns a thread that keeps checking the socket connection for data. When data is received it is sent to the data_storage. The GUI has a timer triggering a function that reads data from the data_storage and updates the GUI. The data_storage is an abstract class, and currently, the only implementation is an SQLite-based one that reads/writes to a database stored as a file.

3.6.2 CAN bus Socket server

The CAN Socket server spawns up a thread with a CAN Bus interface listening for new messages, and a thread running a socket server allowing incoming connections. Any incoming message on either will be stored in a FIFO list in the object. The main thread keeps looping, checking for new data on either FIFO list, and forwards it if anything is found.

This software can easily be extended by making a module that makes data available in a similar way as the two existing server classes. It can create a new thread to work on logic in the background, and data can be sent to it as well from the other servers to receive sensor data or commands over the socket connection. It is also possible to use the CAN server, more or less directly, or use this implementation as an inspiration to build a python program that communicates on the CAN bus from scratch.

3.7 Building a small autonomous car

Figur 3.14: Building a small autonomous car

As a summary of the evaluations for each of the preceding parts. Yes, the board concepts, firmware shells, and Python scripts resulting from this project can be used to make this small autonomous car a reality. A new revision of the can sensor node board is needed, but the proof of concept version of it shows real promise. Both the BeagleBone and the Raspberry Pi boards work well with CAN bus and run the Python programs for CAN and Socket communication well. Three motor choices are available, but testing them is still to be done. And there is a bit of work on manufacturing fastening brackets and parts to fix boards, cameras, etc to. When the physical car is ready, there is also a bit of planning left on commands. One of the possible motor choices is to control the main drive motor directly using a BeagleBone, instead of the RC-platform-Card, and leaving the RC-platform card as a base only used on the smaller single-board car. An alternative BeagleBone Cape called BBBMini aimed to make a BeagleBone into an ArduPilot-based flight controller and is also an alternative as the main control board for such a car. Ardupilot, and that Cape offers support for multiple sensors, GPS, and motor control, and it has a CAN transceiver for communicating with CAN nodes.

Kapittel 4

Results and discussion

4.1 SBC - development tool

Aiming to provide a range of ways how to use an SBC, focusing on Raspberry pi and BeagleBone, as development tools the project was successful. There is still a lot left to be tested, but it has already started to be a useful asset for embedded development. Flashing and debugging with it works well, even though the limitation of the current board design is that you can only flash/work on a chip soldered on the PocketBeagle Cape. Using it for automated testing is still to be tested.

4.2 Testing

Implementing a TDD workflow, and integrating Unit testing in the development process was not successful. As the project was wide and consisted of a lot of moving parts, elements needed to be designed, and both an unfamiliar microcontroller (STM) and an unfamiliar bus (CAN bus). A lot of attention went into piecing things together, and testing was continuously being postponed. Even though few tests were written throughout the process, a lot of thought has gone into how it might have been possible to implement good tests, especially for the Python code.

4.3 Small single-board car

The RC-platform-board, is primarily designed to be used as a stand-alone solution. It turned out quite well. And, besides the microcontroller, the remaining electrical components can be ordered cheaply from China, and most of the supporting parts can be manufactured in less than an hour with a 3D printer and a laser cutter. The Zephyr RTOS development environment can be easily set up, and anyone with a basic knowledge of C could be able to start playing with the car within a reasonably short amount of time. The board also features, an external motor PWM pin, servo control, and an SPI CAN driver adding the possibility to extend it to be controlled by a SBC, or be part of a larger system. With just small modifications a SBC can be directly mounted on the existing frame. Part of the project requirements specifications was to implement BLE and GATT for the stand-alone setup, but that has not yet been finished.

4.4 Small autonomous car

Developing a system that can be used to make a set of sensor nodes, a motor controller board, combined with a SBC controller. Did in many ways end up as the ultimate goal of the project. The CAN sensor node PCB design was flawed, but the errors have been found, and the design has been tested using a development board, and breadboards. This development setup has been less reliable than a finished design would be. And electrical connections on the test setup are most likely to be more affected by noise, and a new revision CAN sensor node board should give even better results than the development system. The test system reliably transmitted large amounts of CAN frames and acted on them. The sensor node by controlling the LEDs, and the Python system by logging and displaying the data in the Python software.

4.5 Python software

The Python software works as described, and offers a basis that can easily be extended when starting a project on autonomous driving using computer vision with OpenCV. The Python software can work on both the single-board scale and the small care scale. And as long as the motor response is tweaked, it is possible to start building algorithms for control on rc-platform-board, and then move the SBC to a physically larger small car, and use the same algorithms and logic with the larger vehicle.

4.6 Discussion

The project might have been a bit ambitious, large, and a bit divided. Working on a combination of platforms, with different microcontrollers, as well as single-board computers, and computer software. Aiming to both set up a useful set of development tools for embedded development, and to use it in the development of a platform for working on a future project in computer vision. The large amount of new information led to some design mistakes that should have been avoided. But, it was possible to make corrections and still complete a working prototype system. Even though there is a working prototype system today, there is still a lot of work remaining to extend and build on this autonomy platform. Is it better than existing alternatives? Compared to the LEGO EV3 robot you can program using Python or Matlab this would require programming in C. Compared to the Duckietowm autonomous cars this project does not offer a working setup driving on a track.

Kapittel 5

Conclusion

A small set of guides to set up development tools, and debugging tools have been created and tested. Boards that can be used to create a stand-alone driving vehicle, or a larger (but small) car/vehicle consisting of a set of nodes (sensor and controllers), have been created and tested. There is a working prototype firmware for both controller and sensor nodes. As well as prototype software that works on Linux Embedded devices. The software is written in Python and can easily be extended and built on to implement different strategies to control the car/vehicle. Both the smallest and the larger vehicles have CAN bus communication, making it possible to change between the differently sized platforms with the same controller hardware. The rc-platform-board is cheap and is easy to assemble by solder, most structural parts can be manufactured quickly using a 3d printer and laser cutter. And, can also be used on similar projects as the suggested projects in the applied physics and math subject. Programming to control it using Python is quite accessible, but there is still work to be done before it would be possible to do a project without having to do any embedded development to modify the platform slightly first.

Bibliografi

[1] Can bus. https://en.wikipedia.org/wiki/CAN_bus.

[2] I2c. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter.

[3] I2c. https://en.wikipedia.org/wiki/I%C2%B2C.

[4] Osi model. https://en.wikipedia.org/wiki/OSI_model.

[5] pytest: helps you write better programs. https://docs.pytest.org.

[6] Stm32cube ecosystem. https://www.st.com/content/st_com/en/ecosystems/stm32cube-ecosystem.html.

[7] Test driven development. https://en.wikipedia.org/wiki/Test-driven_development.

[8] Tested by nordic: Bluetooth long range. https://blog.nordicsemi.com/getconnected/tested-by-nordic-bluetooth-long-range.

[9] Vscode. https://code.visualstudio.com/.

[10] What is fdm 3d printing? – simply explained. https://all3dp.com/2/fused-deposition-modeling-fdm-3d-printing-simply-explained/.

[11] What is sla 3d printing? https://www.hubs.com/knowledge-base/what-is-sla-3d-printing/.

[12] Robert Cecil Martin. Clean code - uncle bob.

[13] Pat Richards. A can physical layer discussion. https://ww1.microchip.com/downloads/en/appnotes/00228a.pdf.

[14] A. Stokka. Ele340 computer design gr. f. Technical report, Universitet i Stavanger, 2021. Project report.

Vedlegg A

Tests

- Test report 01 - Functionality test the circuit board designs
- Test report 02 - Distance sensor readings

ELE340
Functionality testing of rc-feather-pcb and stm32-can-node-pcb
Test 1

Asbjørn Stokka (959810)

Tested: November 24, 2022
Revised: December 13, 2022

Summary

Designed PCBs using KiCad 6, manufactured the PCB at JLCPCB, and assembled them manually. Tested the main functionality in order to make sure that the main functionality is in place and works according to the project requirements. Write a list of issues if there are changes that have to be made for the next revision, and make necessary adjustments for the prototype boards usable at this stage.

1 Introduction/Purpose

This report documents functionality testing using an oscilloscope, lab power supply, multimeter, and signal generator to test and debug main parts of the circuits individually making sure voltages and signals work per the project requirements. Testing and assembly were done incrementally,

2 Equipment

- RC-feather-PCB rev 1.0
- RC-feather-PCB parts as described in rev 1.0 BOM for setup with UART GPS
- RC-feather-PCB rev 1.0 Schematic and PCB-layout for reference (iBom)
- STM32-CAN-Node-PCB rev 1.0
- STM32-CAN-Node-PCB basic parts as described in rev 1.0 BOM
- STM32-CAN-Node-PCB rev 1.0 Schematic and PCB-layout for reference (iBom)
- FLUKE 101 digital multimeter

- Lab power supply
- 4 channel oscilloscope with a signal generator
- 2 channel signal generator
- Banana plug cables and connectors to connect equipment to the PCB test-points

3 Method RC-feather-board

3.1 PCB preparation

Checked the machining to make sure vias, holes, and routes had been machined out correctly according to the board layout.

3.2 Voltage regulator assembly

Assembled the 3.3 V voltage regulator circuit (VR1) with associated resistors and capacitors, and tested them using 5 V input (power connector). After successful testing assembled the 5 V regulator (U5), and associated capacitors. Tested it using 8-10 V input (100 mA limit). Lowered the input voltage step by step to find the lower voltage limit where it would still provide a stable voltage output. 3.3 V and 5 V output can be tested on any of the headers rated for that supply output for external boards.

3.3 Remaining PCB assembly

Soldered remaining pin sockets and components on the board (excluding the extra i2c connectors and the i2c GPS module). And inspected the soldering with a digital microscope. Connected external modules and sensors according to the board layout.

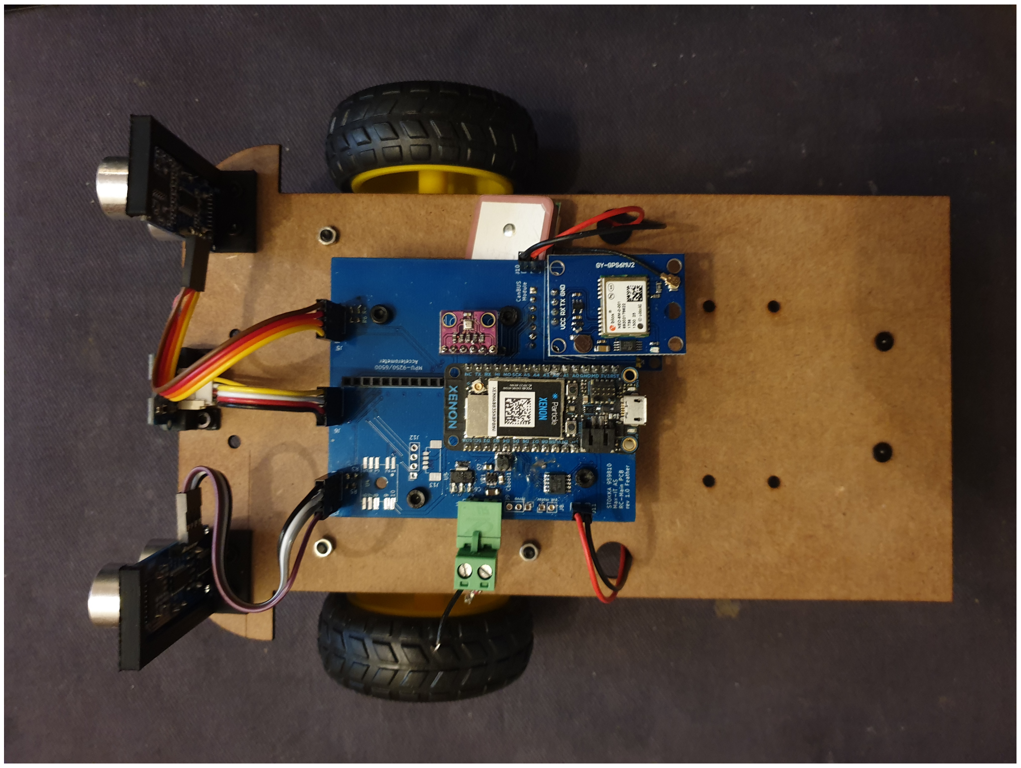


Figure 1: Assembled PCB

The board works nicely as a prototype/testing board. It is fairly straightforward and easy to solder. The hardest part was the 3.3 V voltage regulator circuit.

4 Method STM32-CAN-Node

4.1 PCB preparation

Checked the machining to make sure vias, holes, and routes had been machined out correctly according to the board layout. Through the inspection, several layout issues became apparent, among them the Rx/Tx for the CAN transceiver being flipped (on the STM32F103C6Tx side). And the fact that the planned CAN transceiver would need 5V VCC. An issue was filed on Github Issue #1 for this to be fixed in the next revision of the board. And, I deviated from the planned testing and use of the board. This issue should have been avoided by better control of the schematic and board layout before production. Better knowledge and more experience with the STM32 chip and CAN bus transceivers might also help avoid this sort of error.

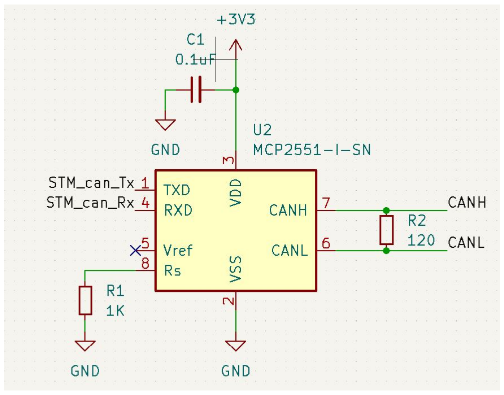


Figure 2: PCB before assembly

4.2 Planned voltage regulator assembly

Assemble the 3.3 V voltage regulator circuit (U3) with associated resistors and capacitors, and test them using 5 V input (power connector).

4.3 Planned PCB assembly

After successful testing, assemble the STM32 IC, oscillator, and connected resistors and capacitors. Test that it can be reached through the programming header. Solder and test the CAN bus transceiver and signal output connectors.

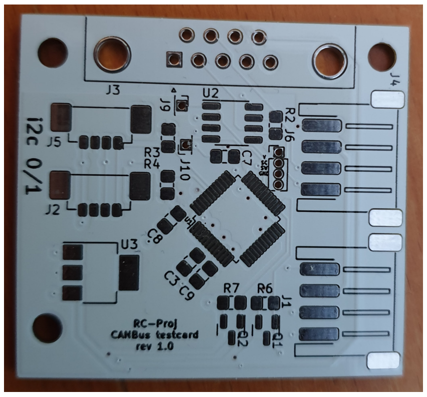


Figure 3: PCB before assembly

4.4 PCB assembly

I checked the size of the footprints to make sure there were no issues with footprint sizes and/or components being placed too close to each other. Assembled the CAN transceiver (U2) and test pins

J9/J10 (Tx/Rx), as well as J5 (for 5 V supply) and CAN H and CAN L for J4/J5. The board would in that way serve as a breakout board for the CAN transceiver for testing purposes.

I used an STM32 Blue Pill dev board with the CAN-Node PCB assembling it on a small breadboard. The planned output pins with MOSFETs for controlling lights were for simplicity replaced by two LEDs. This change also inspired to add a HC-SR04 distance sensor to the board.

4.5 Remaining RC-feather-PCB testing

4.5.1 Motor controller

Sent different signal combinations through the pins on the feather pin header and measured the motor output signal to see if it outputs the correct high/low voltages.

4.5.2 HC-SR04 Distance sensors

Checked for connection on the pins to the sensor (V, GND, trig, echo).

4.5.3 i2c devices

Used shell on the particle xenon and scanned for known addresses for the different sensors.

5 Tests and results

5.1 3.3 V voltage regulator

5.1.1 Schematic

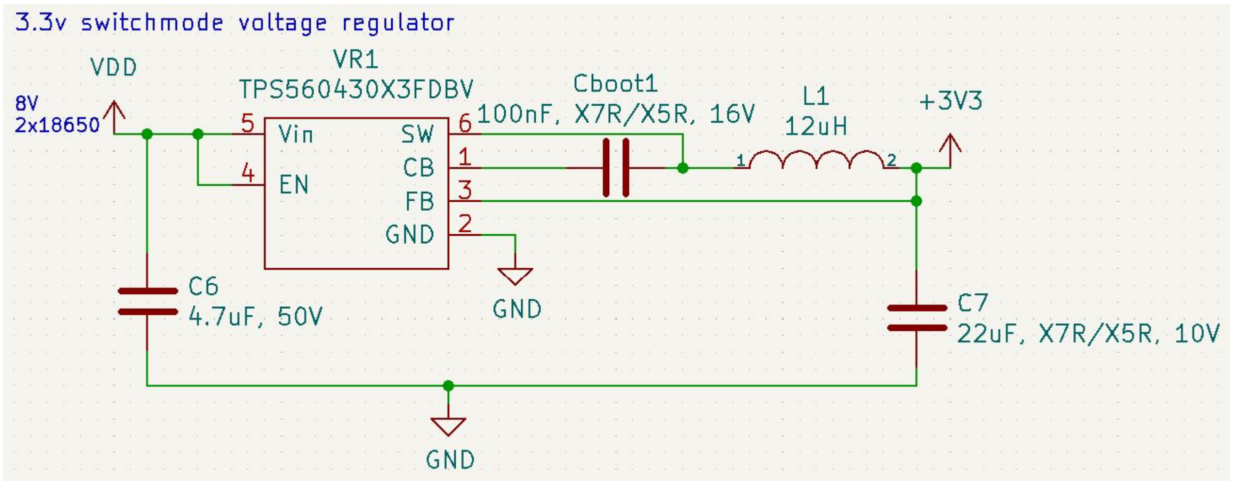


Figure 4: 3.3 V switchmode voltage regulator circuit

5.1.2 Description

Connected DC 5 V input (100 mA current limit) and ground to the board power connector. Measured voltage on the 3.3 V pin of the feather connector. The measured voltage should be 3.3 V. Tested ok on 13. Dec, using a handheld FLUKE 101 multimeter. The voltage source has a stable voltage output close to 3.3V averaged over 10 seconds, with insignificant deviation.

5.2 Linear 5 V LDO voltage regulator

5.2.1 Schematic

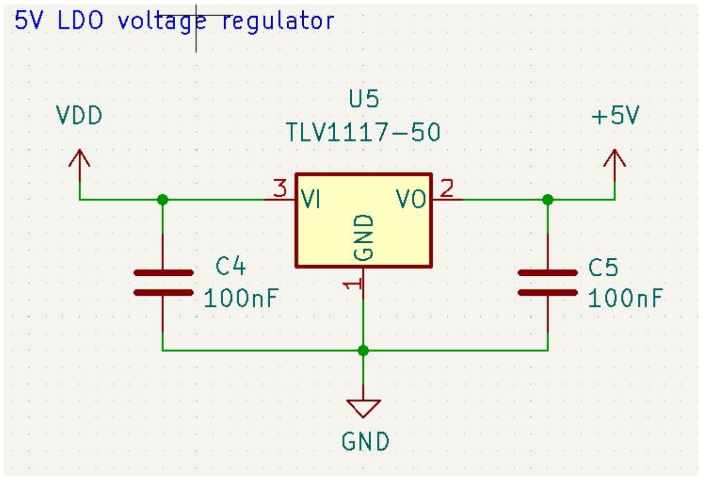


Figure 5: 5 V voltage regulator circuit

5.2.2 Description

Connected DC 8 V input (100 mA current limit) and ground to the board power connector. Measured voltage one the connector of the HC-SR04 distance sensor. The measured voltage should be 5 V. Tested ok on 13. Dec. The voltage source has a stable voltage output close to 5 V averaged over 30 seconds readings, with insignificant deviation.

5.2.3 Comments

Tested ok on 13. Dec. When testing for minimum input the input voltage was lowered until there was a significant drop in output voltage. The output voltage on the 3.3 V dropped significantly when the input voltage was at 3.4 V and below. The input voltage was increased again until 3.8 V where the output voltage was stable at 3.3 V again. The minimum input voltage found to achieve a stable output voltage of 3.3 V was 3.8 V.
The output voltage of the 5 V dropped significantly when the input voltage was at 5.9 V and below. The input voltage was increased again until 6.0 V where the output voltage was stable at 5 V again. The minimum input voltage found to achieve stable voltage output for the system is 5.9 V.

5.3 Remaining RC-feather-PCB testing

5.3.1 Motor controller

The motor output pins had the described high/low voltages as described in the motor controller datasheet for combinations of high/low signals on the input pins.

5.3.2 HC-SR04 Distance sensors

Used the FLUKE 101 multimeter to check that there was low resistance/good connection from feather connector pins to the sensors, and for GND/5 V.

5.3.3 i2c devices

Used shell on the particle xenon and scanned for known addresses for the different sensors. And, the sensors replied as expected.

6 Conclusion

Throughout the functionality testing some issues, and possible improvements have been identified with rev 1.0 of the CAN-node-pcb, and bug issues have been filed on Github accordingly (issue #1, #2, and #3). However, it has been possible to make adjustments in such a way that it possible to work with the current revisions of the PCB's through the development process. They are both fairly simple boards, relying on digital signals, and using fairly low frequencies. The manufactured prototype boards can be used in their current modified form.

ELEBAC
Calibrating and testing HC-SR04 ultrasonic distance sensor readings
Test 02-2022

Asbjørn Stokka (959810)

Tested: December 13, 2022

Summary

Systematic and averaged sensor readings to check that the sensors give reliable data.

1 Introduction/Purpose

This report documents the lab work done to get confirmation data for the distance sensors. The sensors have been individually tested while implementing them on the systems. And, this test can be seen as an acceptance test that the sensors are functioning as expected on the final system(s). The system was set up and connected, and CAN bus reading software was used to read the distance readings.

2 Equipment

- STM32 Blue pill dev-board
- STM32 CAN node board
- HC-SR04 distance sensor
- Folding ruler
- Cardboard box
- Computer with terminal software
- Embedded Linux device (Raspberry pi / Beaglebone)
- Stokka-elebac-22_embedded-linux CAN tools
- Stokka-elebac-22_stm32-can-node-fw STM32 firmware

3 Method

3.1 Data sheet

The HR-SR04 distance sensor is a contactless ultrasonic distance sensor. The datasheet states a range of 2cm - 400 cm for the sensor, with an accuracy of up to 3mm. The sensor receives an input trigger for at least 10 us. The module sends eight 40 kHz pulses and detects if there are any pulse signals bounced back. If the sensor receives a signal back, it will keep the echo signal high for a duration such that you can calculate the distance: distance = time high * velocity of sound / 2. (Velocity of sound: 340 M/s). The measuring angle of the sensor is 15 degrees, making it difficult to test the furthest ranges of the sensor within the physical constraints of the room. At a later stage, it might be possible to do new tests for larger distances.

3.2 Hardware setup

The Blue pill was connected to the CAN transceiver on the STM-node PCB (giving 5V as VCC for the CAN transceiver), and CanRx connected to CanRx and CanTx connected to CanTx. The HR-SR04 sensor was connected according to the given pin-out in the firmware for trig/echo pin (the echo pin was chosen depending on an available timer in the STM32 chip). The STM32 was powered through USB. And CAN-H and CAN-L lines were connected between the Linux-embedded (Beaglebone) device and the STM node. An oscilloscope was connected to the CAN line to monitor the signals as well. The sensor was aimed away from tables/obstacles, and a cardboard box was used as an obstacle to bounce the signals on.

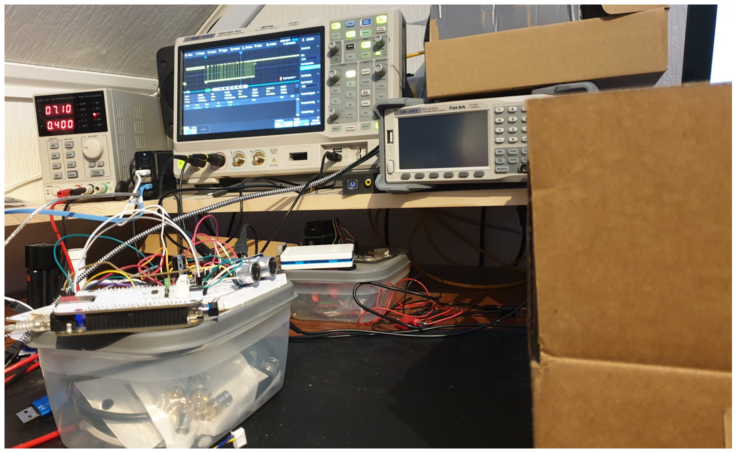


Figure 1: Fully assembled test system

3.3 Software setup

STM32 Blue pill card was flashed with the CAN node firmware using STM32Cube. And, the Linux-Embedded device was configured with a CAN network with a baud rate of 500000 (as described in the setup instructions in Linux embedded). candump was used to track CAN packages with id 0x30 containing distance measurements.

3.4 Testing instructions

Put the cardboard box at a set distance. Measure the distance. Record the distance shown in candump, and calculate the distance in cm from the hex value. If the measurements are unstable, average them.

4 Results

Distance	Calculated distance	Transmitted value
cm	cm	hex
1	5	0x05
2	2	0x02
3	3	0x03
7	7	0x07
14	14	0x0E
21	21	0x15
62	62	0x3E
105	101	0x65
117	118	0x76
142	142	0x8e

The measurement done with a smaller distance than the operating range of the sensor was wrong. Except for that the distances are usually exact. For a couple of measurements, the distance was a couple of cm off at the time of doing the reading, but that could be due to it being hard to measure exact distances with the available equipment.

5 Conclusion

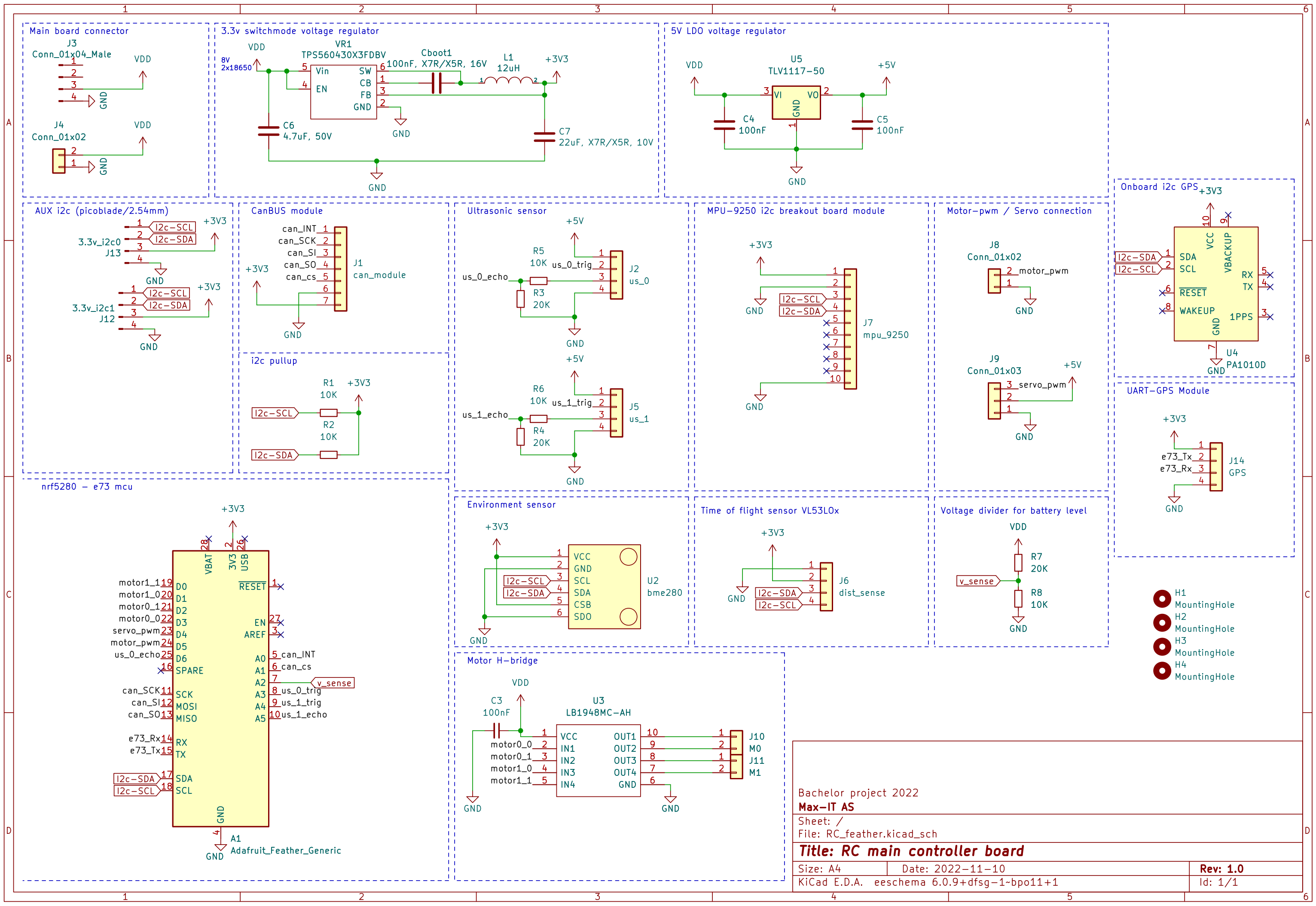
The sensor readings have been rounded to whole numbers to avoid floating-point operations on the STM32 CPU. Aimed towards being one of several sensors on a larger vehicle the sensor performs well and offers reliable measurements within the tested range. The datasheet states a 0.3mm accuracy for the measurements, but that has not been tested due to the rounding done by the firmware when calculating the distance.

The same calculations and method have also been used on the Particle Xenon firmware with the NRF52840 CPU. However, as the NRF52 has FPU and is aimed toward more accurate and smaller settings it also stores the decimal values in the sensor framework of the RTOS. However, due to the 3mm accuracy of the sensor, it will most likely need filtering for the added decimals to give any use at all.

Vedlegg B

Design files

- RC-Platform-board rev 1.0 schematic
- RC-Platform design
- Laser cutting template for the RC-Platoform body
- STM32 CAN Sensor Node Schematic



Bachelor project 2022

Max-IT AS

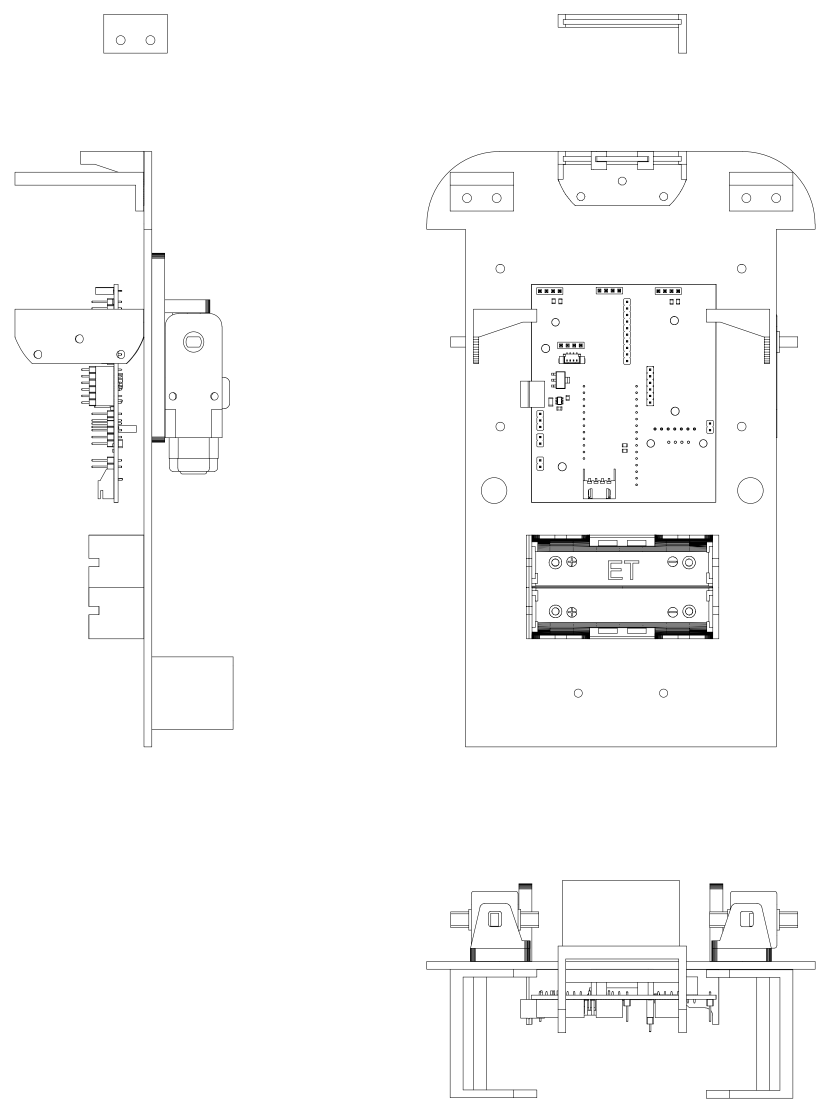
Sheet: /

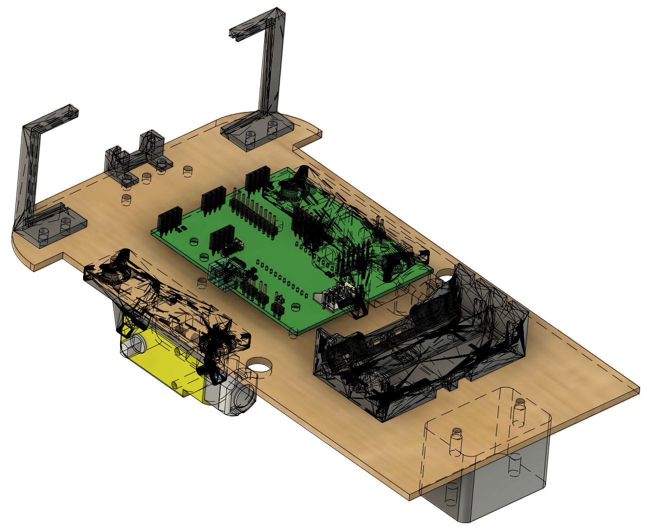
File: RC_feather.kicad_sch

Title: RC main controller board

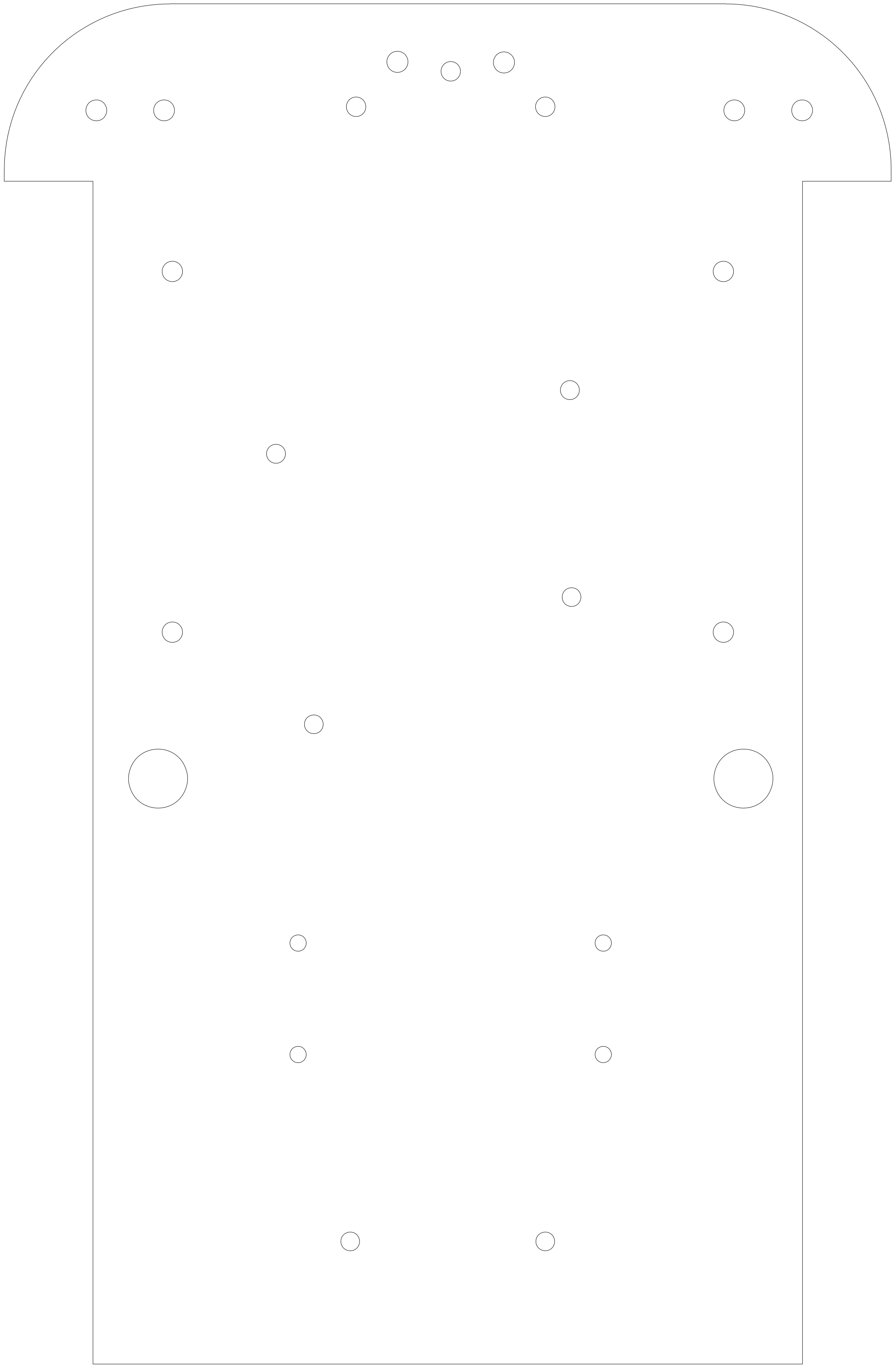
Size: A4 Date: 2022-11-10 **Rev: 1.0**

KiCad E.D.A. eeschema 6.0.9+dfsg-1-bpo11+1 Id: 1/1

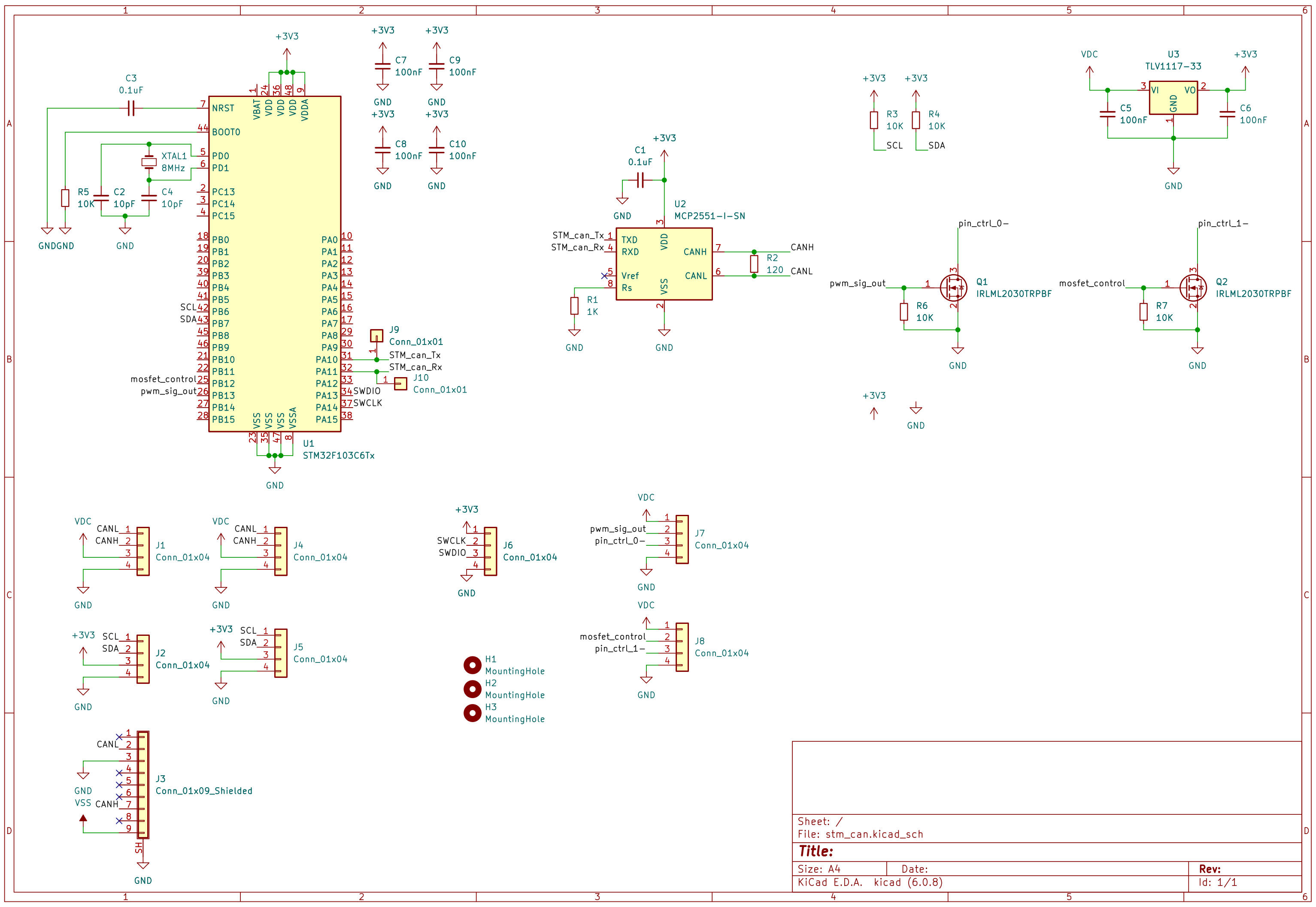




Dept.	Technical reference	Created by Asbjørn Stokka 15.12.2022	Approved by	
		Document type	Document status	
		Title Car_design	DWG No.	
			Rev.	Date of issue	Sheet 1/1



Dept. ELEBAC	Technical reference	Created by Asbjørn Stokka 27.11.2022	Approved by
		Document type Laser cut design	Document status
		Title RC-Car body	DWG No.
			Rev. 1.0 | Date of issue | Sheet 1/1


+3V3
C3
0.1uF
NRST
BOOT0
PD0
PD1
XTAL1
8MHz
PC13
PC14
PC15
R5
10K
C2
10pF
C4
10pF
GNDGND
GND
PB0
PB1
PB2
PB3
PB4
PB5
PB6
PB7
PB8
PB9
PB10
PB11
PB12
PB13
PB14
PB15
SCL
SDA
mosfet_control
pwm_sig_out
VBAT
VDD
VDD
VDD
VDDA
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
PA8
PA9
PA10
PA11
PA12
PA13
PA14
PA15
VSS
VSS
VSS
VSSA
U1
STM32F103C6Tx
GND
J9
Conn_01x01
STM_can_Tx
STM_can_Rx
J10
Conn_01x01
SWDIO
SWCLK
C7
100nF
C9
100nF
C8
100nF
C10
100nF
C1
0.1uF
U2
MCP2551-I-SN
TXD
RXD
Vref
Rs
VDD
VSS
CANH
CANL
R2
120
R1
1K
R3
10K
R4
10K
VDC
U3
TLV1117-33
VI
VO
GND
C5
100nF
C6
100nF
pin_ctrl_0-
pin_ctrl_1-
Q1
IRLML2030TRPBF
Q2
IRLML2030TRPBF
R6
10K
R7
10K
J1
Conn_01x04
J4
Conn_01x04
J6
Conn_01x04
J7
Conn_01x04
J2
Conn_01x04
J5
Conn_01x04
J8
Conn_01x04
H1
MountingHole
H2
MountingHole
H3
MountingHole
J3
Conn_01x09_Shielded
VSS
SH
Sheet: /
File: stm_can.kicad_sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (6.0.8)
Id: 1/1

Vedlegg C

Datablad