TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Rim Puks 182558

LOW LEVEL CONTROLLER SOFTWARE FOR CLEVERON PACKAGE DELIVERY ROBOT

Master's thesis

Supervisor: Peeter Ellervee
PhD
Co-Supervisor: Martin Appo
MSc

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Rim Puks 182558

CLEVERONI PAKIVEOROBOTI MADALAMA TASEME KONTROLLERI TARKVARA

Magistritöö

Juhendaja: Peeter Ellervee
PhD
Kaasjuhendaja: Martin Appo
MSc

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Rim Puks

18.05.2020

Tallinn 2020

Abstract

Cleveron AS is developing a remote controlled vehicle robot. This thesis focuses on the lower level controller for the new protoype of this robot. The requirements for the new controller were defined and STM32F767ZI on Nucleo-144 board was selected. Program code was developed for it using FreeRTOS framework and HAL library. CAN communication with the higher level controller was implemented to receive commands and send feedback. The robot can be controlled over CAN bus and by Radio Control transmitter. The software controls the steering, driving and braking motors as well as the lights. The finished software was tested on the robot by driving around on a test course.

This thesis is written in English and is 57 pages long, including 6 chapters, 30 figures and 4 tables.

Annotatsioon

Cleveroni pakiveoroboti madalama taseme kontrolleri tarkvara

Cleveron AS on Viljandis asuv ettevõte, mis tegeleb pakirobotite tootmise ja arendusega. Viimased aastad on Cleveron arendanud sõitjata pakiveorobotit. Käesolev lõputöö tegeleb selle roboti uuele prototüübile madala kihi kontrolleri lahenduse arendamisega.

Magistritööd alustati uue kontrolleri nõuete kaardistamisega. Määrati vajaminevad perifeeriad ning nõuded jõudlusele ja skaleeritavusele. Nõuete analüüsi tulemusena osutus sobivaimaks STM32F767ZI Nucleo-144 arendusplaadil.

Kontrolleri tarkvararaamistik põhineb FreeRTOS-il. Tegemist on enamlevinud reaalajaoperatsioonisüsteemiga. FreeRTOS lihtsustab protsesside ajastamist ja soodustab tarkvara modulaarsust ja skaleeritavust.

Töö käigus arendati välja alumise kihi kontrolleri kahepoolne suhtlus ülemise kihiga üle CAN-i siini. Suhtluse organiseerimiseks loodi DBC fail mis defineerib erinevate CAN-i sõnumite sisu. Kasutades Python3-el põhinevat konverterit tekitati DBC failist C keele struktuurid ja definitsioonid.

Loodi ka alternatiivne roboti kontrollimise viis läbi raadiosidepuldi. Sellega saab operaator robotit manööverdada, kui seda ei saa või ei ole mugav üle CAN-i teha. Raadioside puldiga valitakse kanal, millega robotit juhitakse, kas CAN-i või raadioside kaudu.

Tarkvara realiseerib vastuvõetud käsklused juhtides rooli- ja veomootoreid, hüdropiduri ja avariipiduri ajameid ning roboti tulesid.

Testimiseks tarkvaraarenduse ajal kasutati projektimeeskonna poolt arendatud testplaati. Hiljem testitit programmikoodi ka valminud robotil nii pukkide peal kui testrajal.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 57 leheküljel, 6 peatükki, 30 joonist, 4 tabelit.

List of abbreviations and terms

TalTech	Tallinn University of Technology
MCU	Microcontroller
RC	Radio Control
ISR	Interrupt Service Routine
DMA	Direct Memory Access
FIFO	First In First Out
UART	Universal asynchronous receiver-transmitter
USART	Universal synchronous and asynchronous receiver-transmitter
CMSIS	Cortex Microcontroller Software Interface Standard
SPI	Serial Peripheral Interface
DAC	Digital-to-Analog Converter
I2C	Inter-Integrated Circuit
HAL	Hardware Abstraction Layer
CAN	Controller Area Network
DLC	Data Length Code
DBC	CAN Database
API	Application Programming Interface

Table of Contents

List of Figures

List of Tables

1 Introduction

The author of this thesis is employed in Cleveron AS. Cleveron is a package robot development and manufacturing company located in Viljandi, Estonia. In the recent years, Cleveron has been developing a package delivery vehicle robot. The goal of the robot is to deliver packages to clients without the need for a human driver. In the later stages of the product development, this robot would act autonomously. That would allow logistic companies to save on labor costs.

Cleveron AS first introduced the idea for a new package delivery robot in Robotex 2018 [27] . Since then, the development has gained media attention on multiple occasions [12] , [11] .

A new prototype for this robot, called Albert, is built on the spring of 2020. New hardware and software developments are tested on it. The author of this thesis is a member of the software team for this project and focuses his work on the low level controller of the robot.

Despite the fact, that the robot will be smaller in scale compared to regular cars, it will still use the same driveways. Due to this, the vehicle robot will have to abide by the same traffic laws as cars. In hardware aspects it means that the robot has to have turn signals, braking lights and low beams. Also, its braking capabilities will have to be similar to normal cars. Due to safety reasons the maximum speed of the prototype is planned to be slowly increased during testing. At first, it will be around 20 km/h. During the development of the robot, there will always be an safety operator driving behind it, ready to press the emergency stop button. The operator will also have a RC (Radio Control) remote allowing it to take over the control if needed.

Figure 1: Prototype "Albert"

Main controlling method of this prototype will be teleoperation using commercial 4G network. There are two controllers on the robot. One is a computer for higher level tasks, like camera image processing and communication with the teleoperation control station. The other, lower level controller, will receive commands from the higher or RC remote. It will control motors, lights and sensors and also provide feedback to the higher level controller.

Cleveron has developed multiple prototypes for this robot over the course of two years. Each one has been more complex than the previous. Prototypes this far have been using Arduino Mega 2560 development board as a low level controller [5] . The advantage of using an Arduino board is that it has allowed to quickly test out different components. Arduino has provided an extensive framework that makes programming the controller easier. It also has libraries for most of the hobbyist electronic components, for example Digital-Analog Converters and stepper motors. Due to the increasing complexity of the project, using Arduino Mega 2560 as a low level controller has become difficult. For

this reason, it has been decided to move to a new and more powerful controller on Albert.

Albert is part of the Cleveron AS project to develop a new product and due to this, some information about the project is confidential. To protect Cleveron business intrests, all confidential information is moved to the appendixes part of the thesis and will not be published with the main part. Due to the contracts with the higher level controller provider, information about it is omitted from both public and the confidential part.

Task

The goal of this thesis is selecting a new low level controller and developing its software. The controller must be able to:

- Run a Real Time Operating System
- Receive commands and send feedback over CAN.
- Receive commands from a RC transmitter.
- Control driving, steering and braking actuators.
- Control the vehicle lights.

Results

After defining the requirements, STM32F767ZI on Nucleo - 144 board was selected as the controller for the new prototype. Software was developed, that fulfills all of the functionality stated in the “Task” section. The program code was tested with the robot on stands and on the test track.

2 Background and planning

In this chapter the background research and the planning phase of the project is described. First part focuses on gathering the requirements for the controller, second part is about controller selection, third part is about Real Time Operating System and the fourth about different debugging tools used in the project.

2.1 Requirements

Before the new controller solution can be selected, the requirements to it have to be defined. This means the number of available GPIO pins, number of peripherals and other specifications.

Since the other components used in the robot are considered confidential, the list of them is moved to Error: Reference source not found and is not part of the publically available version of this thesis.

In the previous prototypes, the communication between the higher and lower level controller has been implemented over Ethernet. For this purpose, Arduino Mega 2560 was equipped with an Arduino Ethernet shield [3] . With the new prototype, it is planned to use CAN (Controller Area Network) for communication between controllers. CAN offers very reliable and relatively high speed connection. It lacks the high throughput of Ethernet, but that is not needed here. CAN also makes it very easy to add new devices to the network. To connect to the CAN bus, the new controller hardware must have a CAN peripheral.

The Arduino based prototypes have used the “millis” library for timing. This allows for rudimentary real time operations but becomes difficult and error-prone with more complex systems. It is planned, that the new controller will be running a Real Time Operating System (RTOS). This will make timing the tasks easier and will also provide

scalability to the project. Since RTOS requires more processing power, it is required that the new controller should have a 32 bit architecture.

Migrating from one software platform (Arduino) to another uses company resources, mainly developer work time. With this in mind, scalabilty is an important keyword. Another platform should be chosen as such that new migration would not be necessary in the future. In this sense, it is recommended that the chosen framework/controller would allow upgrading to more powerful (or if needed - less powerful) options without changing the program code.

Due to the rapid development of the project, it is decided that the controller should be available on a development board, removing a need to build a PCB around a microcontroller. This will allow to start developing instantly. The PCB can always be developed in the later stages of the development.

The lower level controller will be powered by the robot battery that also has to power the high level controller and the motors. Since in comparison, the power consumption of the low level controller is marginal, there are no power requirements for the controller. Same principle applies to the controller price, but it is still required that the cost would be under 100€.

It is required that the controller would also have at least one additional I2C, two SPI, 2 UART peripherals and 10 GPIO pins in case more sensors or actuators are to be installed to the robot. For example, devices like SPI based encoder and I2C based ultrasound distance sensors are planned to be added to the robot in the later development stage.

The number of different microcontrollers is vast. To narrow down the number of possible options, and thus make deciding easier, it was chosen that the controller should have a ARM Cortex-M family microprocessor. ARM is a world leading microprocessor developer and is implemented in a wide array of microcontrollers. Cortex-M family is meant specifically for low power and low cost embedded systems [6] . Other Cortex families are unnecessarily powerful and complex for our application.

2.1.1 Requirements summary

Summarizing this chapter and the component requirements stated in the appendixes, the requirements for the controller are:

- At least 21 GPIO pins.
- At least 2 I2C peripherals
- At least 2 SPI peripherals
- CAN peripheral
- 3 UART peripherals
- 2 hardware timers
- Cost is less than 100€
- Company provides other options with the same software framework.
- Available on developer board
- RTOS capable
- 32 bit architecture
- ARM Cortex-M family processor

2.2 Choosing the controller

There exists many different microcontroller families that implement ARM Cortex-M. Most of them offer developer boards that match the requirements stated above. Microchip has the SAM based Xplained series [28] , STMicroelectronics has the STM32 based Nucleo and Discovery series [35] , NXP has the LPC based LPCXpresso boards [31] and Texas Instruments has TIVA LaunchPAD series [39] . This means there is not a single best choice.

Eventually, STMicroelectronics STM32 Nucleo line was selected. Nucleo boards are reasonably priced and they come with an integrated debugger/programmer. The author and other engineers in Cleveron AS have had experience with them. STM32 has HAL (Hardware Abstraction Layer) software library that simplifies moving from one STM32 microcontroller to another, since the code does not need to be changed. Also, there is a configuration software CubeMX, that makes initial setup and configuration much easier since it generates all the code that is needed [35] .

Since the price difference between Nucleo boards is small and power consumption is not a priority, Nucleo-144 type with STM32-F767ZI microcontroller was chosen. It is one of the more powerful Nucleo boards with more available flash memory. The board can be seen below on Figure 2.

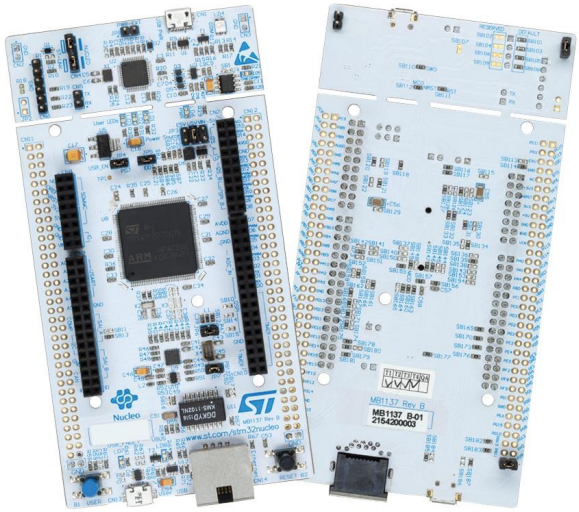


Figure 2: Nucleo-F767ZI[17]

Once the controller board was selected, the mechatronic engineers of the project team started building a PCB around Nucleo-F767ZI to make everything more compact. A short description of it can be found in the restricted access Error: Reference source not foundError: Reference source not found.

2.3 RTOS

The central part of this robots embedded software will be the Real Time Operating System. The main advantage of RTOS is that it allows better timing of tasks and more scalability for the software.

“An operating system is a computer program that supports a computer’s basic functions, and provides services to other programs (or applications) that run on the computer.”[2] What turns an OS into RTOS is a predictable (deterministic) execution pattern. RTOS must conform to real time requirements. “A real time requirement is one that specifies that the embedded system must respond to a certain event within a strictly defined time (the deadline).”[2]

2.3.1 FreeRTOS

Even though there are many different Real Time Operating Systems in the market, FreeRTOS is the most polular. It supports a wide range of devices, is open-source and free. It includes a kernel and different libraries for a wide array of use cases[19] . FreeRTOS is included in STM32CubeMx configuration tool, so installing it has been made simple. Enabling it in CubeMX auto-generates all the necessary files.

It is owned by Amazon and distributed under MIT open source licence. MIT open source licence allows re-licensing the software under new licence [29] . This means the robot application does not have to be open-source as would be with the GNU General Public Licence, that is very common with open-source software [20] .

2.3.2 CMSIS-RTOS

When CubeMX includes FreeRTOS middleware, it adds a CMSIS-RTOS API layer. This means the user will not use FreeRTOS default API but the one provided by CMSIS. CMSIS stands for Cortex Microcontroller Software Interface Standard. It is an abstraction layer created for ARM Cortex microcontrollers [13] . STM32 CubeMx auto-generated code expects the user to use CMSIS-RTOS v1 or v2 API. While testing both CMSIS versions, it was discovered, that semaphores fail to work correctly when using CMSIS v2. Due to that, CMSIS v1 is used in this project.

2.3.3 RTOS preemptive vs cooperative

The scheduler of FreeRTOS can be configured to be in preemptive or cooperative (non-preemptive) mode. The scheduler is the piece of software of the operating system that decides which task gets processor time and when [36] .

Preemptive mode means that each task gets a time slice. “Context gets switched when:

- Time slice has passed
- Task with higher priority has come
- Task goes to BLOCKED state (i.e. by call osDelay() function)
- Task goes to READY state (i.e. by call osThreadYield() function)” [36]

Cooperative mode means that there are no time slices and the tasks are not preempted by higher priorty tasks. “Context gets switched ONLY when RUNNING task

- goes to BLOCKED state (i.e. by call osDelay() function) or
- goes to READY state (i.e. by call osThreadYield() function) or
- is put into SUSPEND mode by the system (other task)” [36]

STM32 FreeRTOS MOOC (Massive Open Online Course) recommends using non-preemptive (cooperative) RTOS mode if possible [36] . That is due to more transparent time management, since all threads switch states at known positions. It also means that the program code will not have to deal with so many race conditions. But when using cooperative mode, the programmer has to assure that all of the processes are short enough that no threads would starve.

2.4 CubeMX configuration software

STM32 CubeMX is a graphical tool that makes initializing and configuring a STM32 project much easier. User can configure all the necessary MCU peripherals from drop down menus. It also shows the programmer which pins are already in use. Once the

configuration part is done, CubeMX auto-generates a project. The user can return to the configuration screen at any time to re-configure.

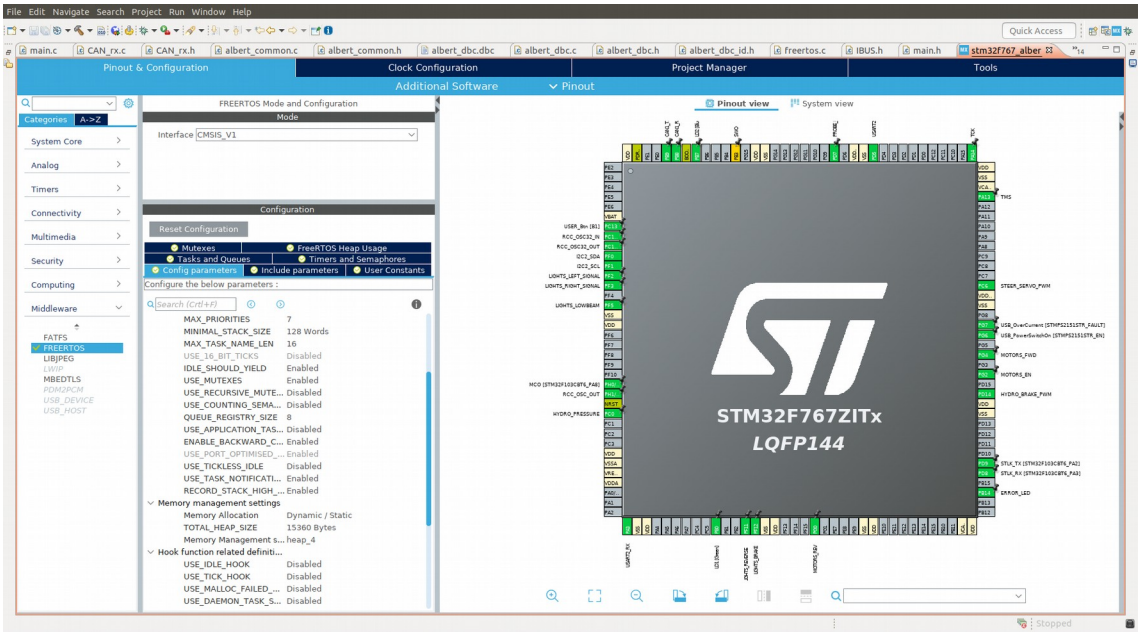


Figure 3: CubeMX page of STM32CubeIDE

But using CubeMx can be error-prone, since it still requires an understanding about the microcontroller for correct configuration. Most flags and configuration options are not explained so it is necessary to consult to the manual of the microcontroller while configuring. For example, the author had some problems with implementing UART for RC receiver. From debugging view, it seemed like UART peripheral sometimes failed to receive data. And after the first breakpoint, it always failed. Eventually, it was discovered, that the STM32CubeMx set the UART overflow error flag as default. This blocked UART if overflow was detected until the overflow flag was cleared. Since RC receiver constantly sends data over UART then sometimes the overflow happened before program code had a chance to set up UART receive. The overflow also happened every time the MCU was paused, as when in a breakpoint.

2.5 Debugging

An important part of developing is debugging, the act of finding and fixing errors in software. In this chapter, the software and hardware equipment used for debugging this project is described.

As stated before, the previous prototypes were based on Arduino boards. With Arduino, the main debugging methods used were sending data to serial port over UART and manipulating GPIO pins to light up LEDs or observe them with an oscilloscope/logic analyzer. Arduino Mega 2560 has JTAG interface and can be connected with a debugger to allow breakpoints and stepping through instructions [7] . But this option was not used since the debugger had not been acquired and the program was simple enough.

2.5.1 Nucleo-F757ZI debugging options

When working with Nucleo-F767ZI board, similar debugging options as with Arduino are available. It is possible to send data over serial line and observe it from PC with a serial monitor software. Also, unused GPIO outputs can be manipulated to signal events and read with oscilloscope or logic analyzer. While Arduino Mega 2560 requires an external debugger for connecting to JTAG interface, Nucleo-F767ZI has an embedded debugger/programmer ST-LINK/V2-1 [Figure 4]. If necessary, it can be broken off and used as an external debugger/programmer for other STM32 devices. With a debugger, it is possible to insert breakpoints, step through the instructions, see variable and register values and change them.

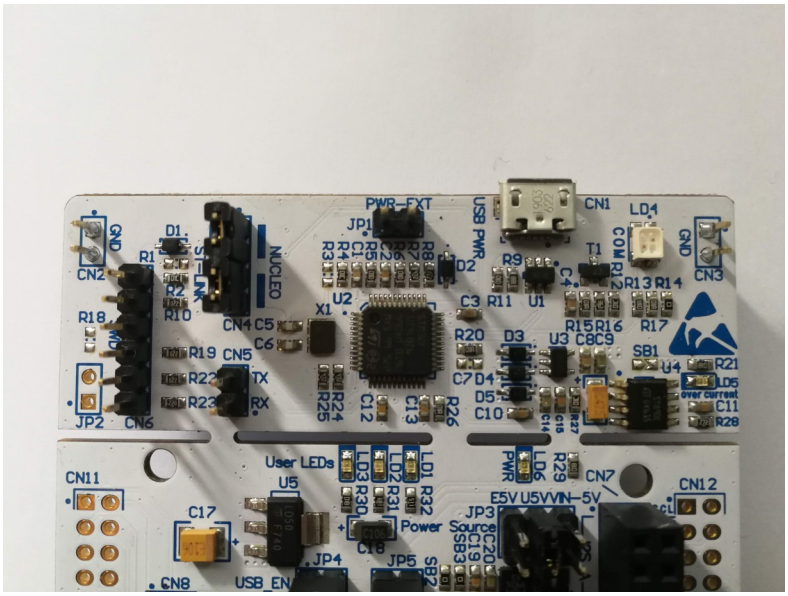


Figure 4: Embedded ST-LINK/V2-1 debugger/programmer

2.5.2 FreeRTOS debugger plugin

The debugging options described in the last section did not have any RTOS specific capabilities. For this, a FreeRTOS debugger plugin was installed to the STM32CubeIDE program. When the program was paused it allowed to observe the stack usage of each thread, making sure that stack overflow would not happen. Also, it was possible to observe in what states the threads were and how much of the total runtime did they take.

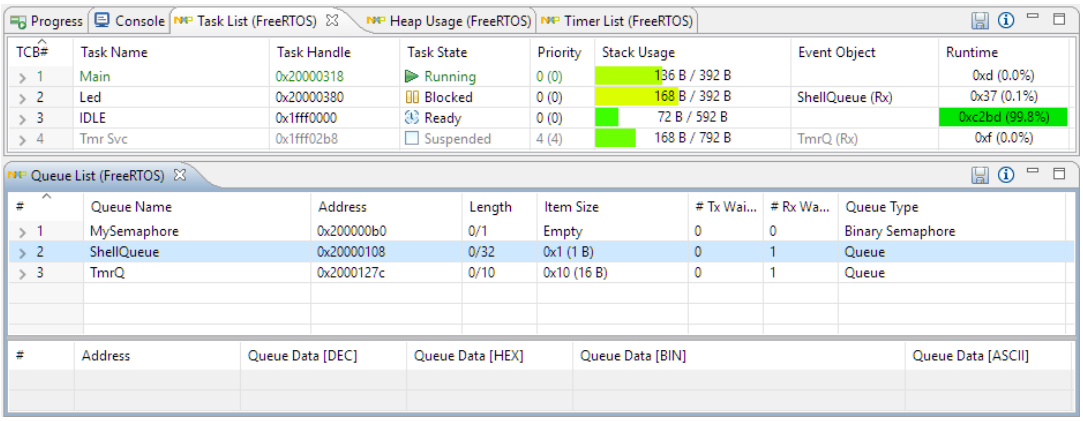


Figure 5: FreeRTOS debugger plugin[8]

The debugger plugin was developed by NXP and was installed here with the guide “Better FreeRTOS Debugging in Eclipse” provided by Erich Styger [8] . The plugin is for FreeRTOS and works on different ARM based MCUs.

2.5.3 Rigol digital oscilloscope

Rigol MSO2302A digital oscilloscope was used extensively during the development process to debug. The main features used were the logic analyzer and decoder. Logic analyzer allowed connecting up to 16 digital inputs. Decoder allowed decoding I2C, UART and CAN messages to confirm correct outputs as seen on Figure 6.

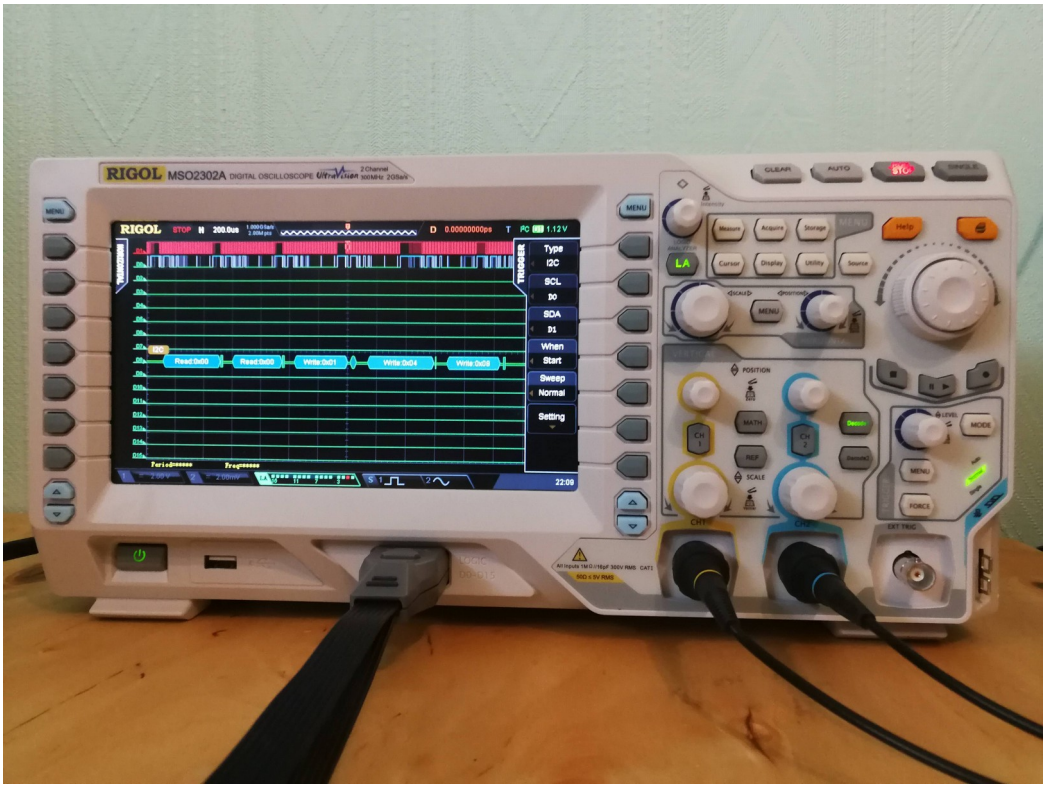


Figure 6: Debugging I2C with Rigol oscilloscope

2.5.4 Kvaser Leaf Light v2 and Kvaser CanKing

Kvaser Leaf Light v2 is a CAN interface for USB. It allows connecting a computer to CAN network. It supports both 11 bit and 29 bit identifiers and manages speeds up to 1 Mbit/s [26] .

Figure 7: Kvaser Leaf Light v2 [26]

Kvaser CanKing is a software for interacting with CAN bus with products like Kvaser Leaf Light v2. It allows sending and receiving CAN and Extended CAN messages, logging to a file, generating traffic and formating messages [25] .

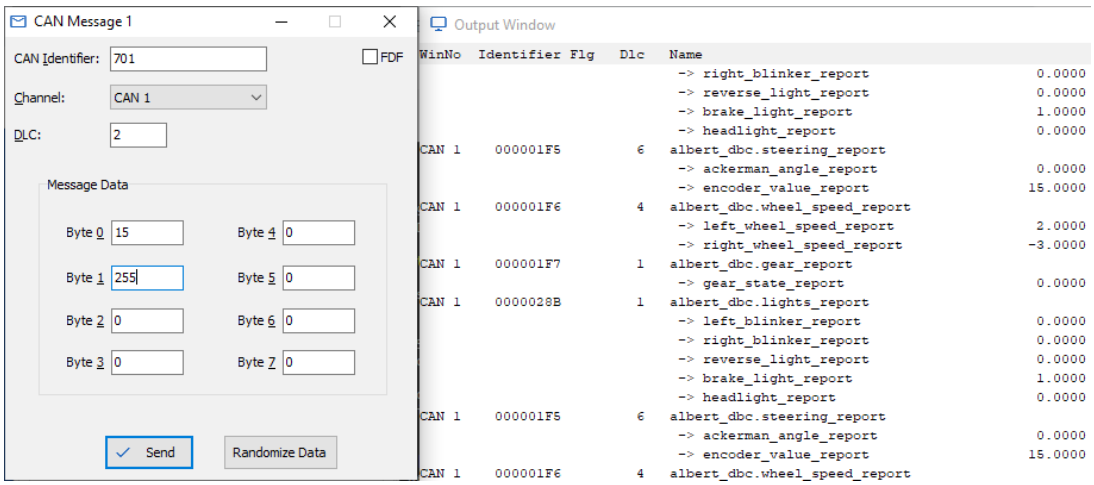


Figure 8: Sending and receiving messages with CanKing

Kvaser Leaf Light v2 was used with CanKing software to debug and test CAN bus and the DBC file [DBC file]. It allowed sending commands to robot and checking feedback. CanKing also allowed to confirm if the DBC file was correct and also if encoding and decoding was working as intended.

3 Control software

In this chapter the control software architecture and development is described. As stated before, the FreeRTOS framework is used for the software. Control of the different components are separated into threads. Threads are called tasks in the FreeRTOS environment, both names are used in this thesis.

Threads are created in the CubeMx FreeRTOS page. The code generation option is set to “extern”, meaning the code generator only creates the declaration and not definition. It is also possible to set it as “weak”. This would generate thread function definition with “weak” attribute into main.c file. If the linker finds any other definitions for this function it would use the non-weak definition.

For more modular approach, each thread has its own source file. The main.c is only used for peripheral configuration and has very little not auto-generated code. The different threads and their priorities and frequencies can be seen in Table 1.

Table 1: Threads and their priorities and frequencies

Thread	**Priority**	**Frequency**
IBUS thread (for RC control)	Real time	When a complete IBUS frame has arrived. ~142 Hz
CAN RX thread	Real time	When a new CAN bus message has arrived.
Steering thread	High	100 Hz
Driving thread	High	100 Hz
Hydraulic brake thread	High	100 Hz
CAN TX thread	Normal	50 Hz (Between full sets of reports)
Lights thread	Low	20 Hz

3.1 Steering motor control implementation

The steering motor installed to this robot is controlled the same way as a hobby servo. This made testing convenient since it could be done with a small hobby servo.

3.1.1 Hobby servo control

Hobby servos are mainly used in remote controlled cars, boats and airplanes. They are also popular with Arduino projects since it is easy to control them using MCU timers. The hobby servo is controlled using 50 Hz PWM. The position of the motor is determined by the length of the pulse as can be seen in Figure 9. Having a duty cycle 5% (1 ms) will turn the motor full clockwise. With 10% (2 ms) the motor will turn full counterclockwise and with 7,5% duty cycle (1,5 ms) it will be in the center position. All the positions between can also be achived by sending a duty cycle higher than 5% and lower than 10% [21] .

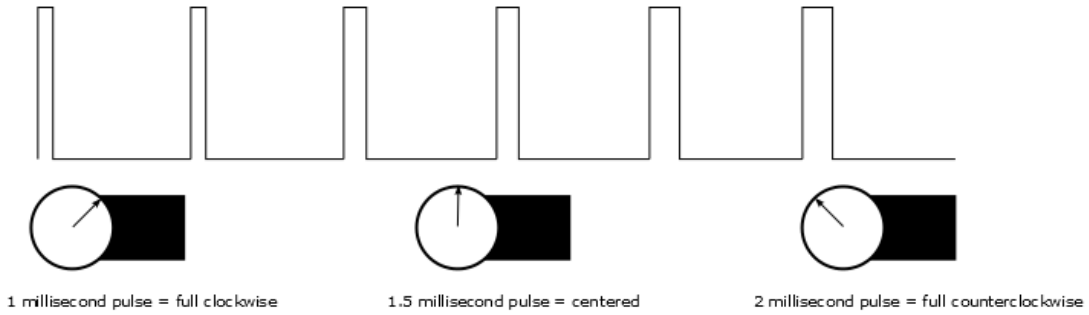


Figure 9: Hobby servo control [21]

In this application, the minimum and maximum positions of the servo are configurable. The algorithm for setting the PWM servo can be seen in Figure 10. The thread reads the steering value from a global steering command variable, converts it to servo PWM and then blocks itself for a specified time.

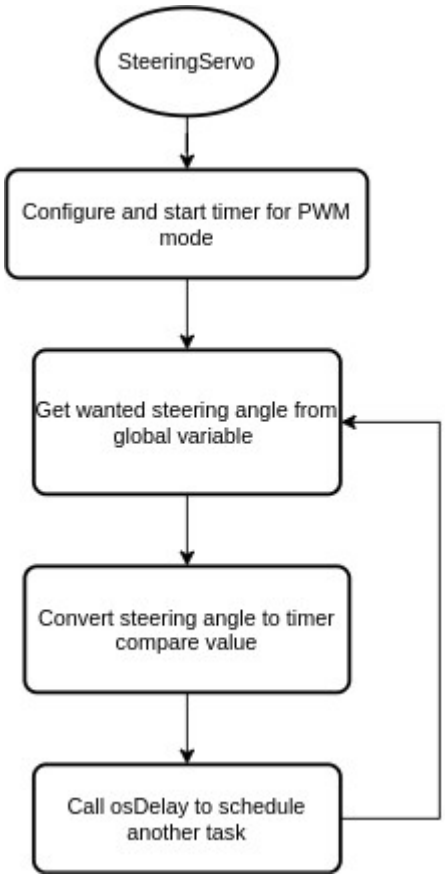


Figure 10: Steering servo control algorithm

3.2 Driving motors control implementation

The driving motor drivers of the robot are controlled with analog voltage and GPIO pins. In the current implementation, all 4 motor controllers receive the same commands, so the wheels will move with the same speed. Two analog and three digital signals get sent to the motor controller. The first analog signal controls the torque of the motor and the second controls the regenerative braking. There is a digital signal for enabling the controller in general and specifying the motor direction. The summary of the interface can be seen in Table 2.

Table 2: Driving motors controller interface

Signal	**Description**	**Physical characteristic**
Throttle	Controls the strenght of driving motors throttle	Analog voltage from 0 to 5 V
Brake	Controls the strenght of driving motors regenrative braking	Analog voltage from 0 to 5 V
Motors_Forward	Enables moving motors forward	GPIO input
Motors_Reverse	Enables moving motors backward	GPIO input
Motors_Enable	Enables the motors	GPIO input

The analog voltage is supplied by an external 4 channel Digital-to-Analog Converter (DAC) Texas Instruments DAC6574 [14] . The microcontroller uses I2C to send commands to the DAC.

3.2.1 I2C message format

The DAC module expects I2C messages to be in a format defined in the product datasheet. The format can be observed in Figure 11.

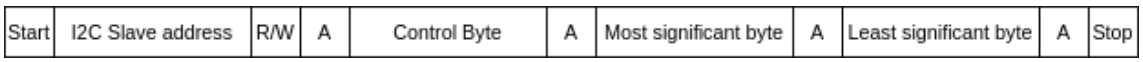


Figure 11: DAC I2C frame

The STM32 I2C peripheral handles the details such as start and stop conditions and acknowledge bits (A). HAL library handles configuring and communication with the I2C peripheral, so the user only has to call the HAL API function declared below.

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit_IT(I2C_HandleTypeDef * hi2c,
uint16_t DevAddress, uint8_t * pData, uint16_t Size);
```

As can be seen from the declaration, the function requires 4 parameters. The handle to the I2C peripheral, the I2C address of the DAC, a pointer to the unsigned integer buffer that holds the data to be transferred and the size of that buffer.

To conform to the I2C frame defined in the datasheet (Figure 11), the first member of the transfer buffer must be the Control byte. That byte specifies the operation mode (in this application only the normal mode is used) and the DAC channel to be updated.

The next 2 bytes are data bytes. The DAC has 10 bit resolution and the bytes have to be shifted as specified in the manual (Figure 12).

Figure 12: MS-Byte and LS-Byte of DMA I2C frame [14]

3.3 Hydraulic brake implementation

PWM is used to control the strength of the hydraulic brake. The hydraulic brake is activated if the global brake command variable CMD_brake_value passes a threshold. In the current implementation, this threshhold is 50% of the maximum CMD_brake_value. The hydraulic brake strength then rises in linear manner with the

brake value, with 100% brake value corresponding to 100% hydraulic brake value. Pseudocode in Figure 13 descibes how the hydrobrake duty cycle gets calculated.

```
if(CMD_brake_value > hydrobrake_threshold)
{
      hydrobrake_duty_cycle = (CMD_brake_value -hydrobrake_threshold)/
            (1.0 - hydrobrake_threshold)
}
else
{
      hydrobrake_duty_cycle = 0
}
```

Figure 13: Hydrobrake duty cycle pseudocode

3.4 Lights implementation

The lights thread implementation is relatively simple. Thread goes through if-else statements for brake, reverse and error lights. If brake command is over 10%, brake lights are activated. If the robot is in reverse gear, reverse lights get activated. And if any of the current error flags is up, error LED is activated. Other lights get activated by CAN commands directly in the callback functions.

4 Communications software

This chapter covers the communication part of the software. The lower level controller receives commands from 2 channels: RC transmitter and CAN bus. The latter is also used to send feedback about the robot state. In this thesis, RC is often referred to as IBUS, after the protocol the RC receiver uses to transmit data to MCU.

4.1 Switching between CAN and RC control

Since the robot movement can be controlled by two different methods, switching between them is required. It was decided that this should be done using the RC transmitter, since the robot prototype should always be supervised by a human operator. And this would allow the human to quickly switch to manual control in case something goes wrong.

The control scheme shown in Figure 14 is proposed. The control mode can be chosen from a switch on RC transmitter. This controls the multiplexer to select the correct input. Full stop state was added as an additional input for safety. If this command is multiplexed forward, the robot will come to a stop. This allows for safer parking, since the robot will not move when the operator accidentally touches the joysticks. The full stop mode also activates the emergency brake. Other control modes turn the emergency brake off.

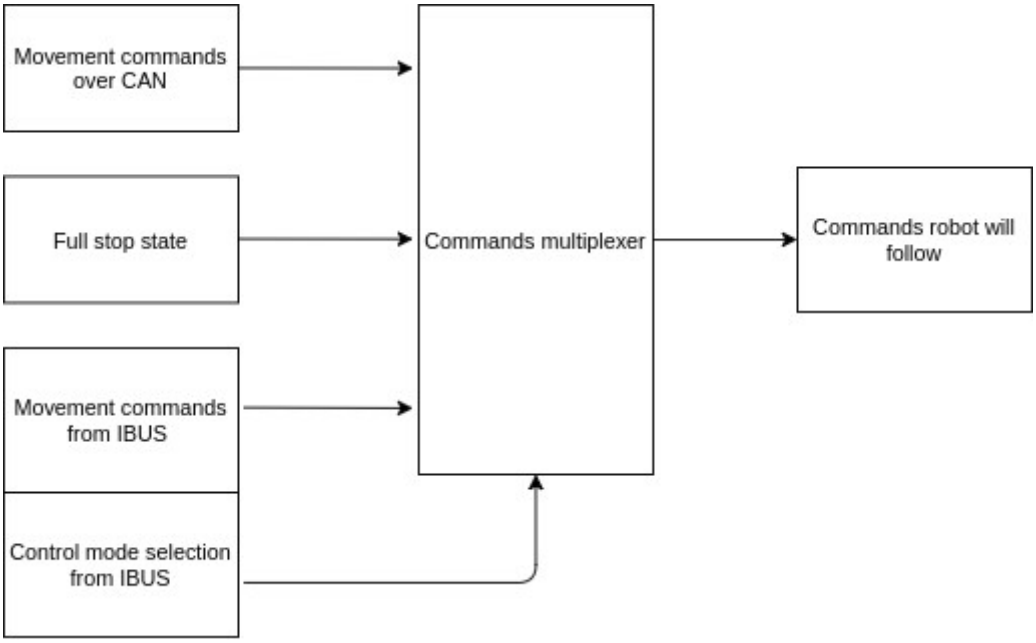


Figure 14: Movement commands multiplexer

The control scheme is implemented using C pointers. When the IBUS thread parses the data from the RC transmitter, it changes the value of the pointer if the control mode has been changed.

4.2 Radio control implementation

The description of the RC receiver and transmitter used for controlling the robot can be found in Error: Reference source not found. The receiver uses IBUS protocol to send data. Due to that, the thread is also named after IBUS.

4.2.1 IBUS protocol

IBUS is developed by a chinese hobby RC component manufacturer and developer Flysky[33] . There is no official IBUS reference manual freely available but the protocol is described in multiple technical enthusiast blog posts [22] ,[23] . There is an Arduino library for decoding IBUS protocol that was used with the previous Arduino based prototypes of the robot [4] . The author also used this library as reference for developing the decoder software for STM32.

The RC receiver sends one IBUS frame every 7,7 ms. The length of the IBUS frame is 32 bytes. The RC receiver sends with baudrate 115200. One UART frame consists of one start bit, 8 data bits, no parity bit and one stop bit (shorthand notation: 8N1). That means 10 bits per one UART frame. Since one IBUS frame consists of 32 bytes, it takes around 3 ms to receive one frame of information.

$$32 \cdot 10 \cdot (1/115200) \approx 3\,ms$$

This timing was verified using oscilloscope as can be seen from ΔX in Figure 15.

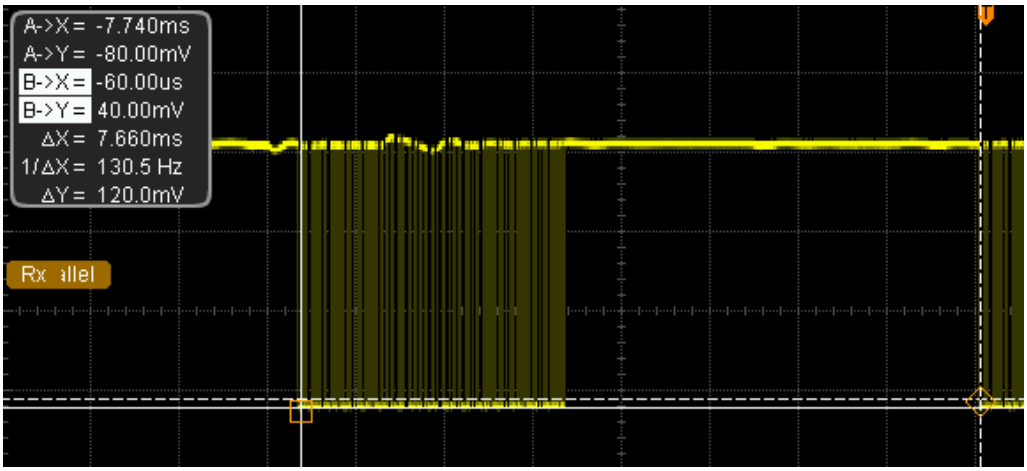


Figure 15. Oscilloscope capture of IBUS frame.

The contents of an IBUS frame are described in Table 3. First two bytes are always 32 and 64. That is used in the program code to detect the start of the frame, and align frame and buffer if needed (see 4.2.3). All the following data is two byte integers in little endian order. Joystick values can be all integers between 1000 and 2000. With 1000 in one side, 2000 in the other and 1500 in the middle. Switches have discreet values. First and fourth have 2 possible positions marked by 1000 and 2000. Second and third have 3 positions, marked as 1000, 1500 and 2000. Some channels of the frame are not configured from the RC transmitter, so their content will always be 1500. The frame ends with CRC value, that is calculated by subtracting all the byte values (not including CRC) of the frame from 0xFFFF.

Table 3. IBUS frame

Byte	Content(in decimal)	Description
0	Always 32	Length of frame
1	Always 64	Command code
2:3	1000-2000	Right joystick horizontal
4:5	1000-2000	Left joystick vertical
6:7	1000-2000	Right joystick vertical
8:9	1000-2000	Left joystick horizontal
10:11	Always 1500	Not used
12:13	1000 or 2000	First switch
14:15	1000, 1500 or 2000	Third switch
16:17	1000, 1500 or 2000	Second switch
18:19	1000 or 2000	Fourth switch
20:21	Always 1500	Not used
22:23	Always 1500	Not used
24:25	Always 1500	Not used
26:27	Always 1500	Not used
28:29	Always 1500	Not used
30:31	0xFFFF – all previous bytes	CRC value

4.2.2 Different UART reading methods

The HAL API provides methods for 3 different UART operation modes [15] :

- The polling mode that blocks the processor. The function returns when a specified number of bytes has been received or a timeout has been reached.
- Interrupt mode where UART peripheral generates an interrupt when a new byte has been received. Processor then moves the received data to specified buffer. A callback function can be customised that gets executed when user specified number of bytes have been received.
- DMA (Direct Memory Access) mode that is similar to the interrupt mode. When UART peripheral has received a new character, it signals the DMA controller to move the byte to a specified buffer. When a specified number of

bytes have been received, the DMA controller raises an interrupt flag, that calls a callback function the user can customise.

For this application, the polling and blocking option can not be used. The MCU would be blocked 3ms/7ms*100% = 43% of the time. This means the choice is between interrupt and DMA modes.

The advantage of the DMA would be that the processor would not need to transfer individual bytes from UART to memory, but it will still need to process the data received. To measure the advantage of using DMA, program code for reading 32 bytes from RC receiver using the interrupt mode was created. Then the UART ISR (Interrupt Service Routine) was modified so that a GPIO pin would be set at the beginning of the ISR and reset at the end. That pin was then observed with oscilloscope along with the UART data coming from RC receiver. It was measured that the ISR transferring one byte from peripheral to buffer took 3 microseconds. An IBUS frame is 32 bytes and one frame is transferred every 7,66 ms.

$$(32 \cdot 0{,}003\,[ms]) / 7{,}66\,[ms] \cdot 100\,\% = 1{,}25\,\%$$

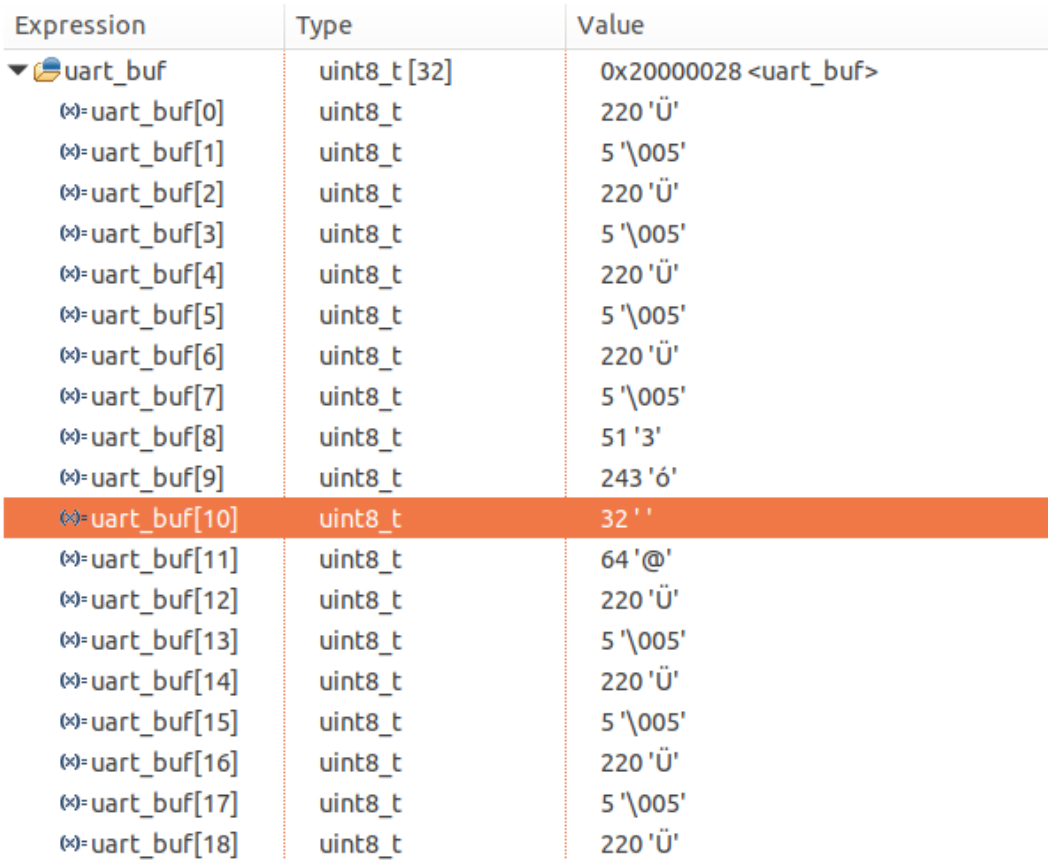
This means using DMA mode instead of interrupt mode only saves around 1,25% of processor time. This is measured with MCU clock speed 96 MHz and will be smaller with higher clock frequencies. The measured time is also slightly affected by the time it takes for the MCU to toggle the GPIO pin.

Even though the advantage gained from using DMA mode is only around 1% of processor time, this mode was still chosen since implementing it with HAL methods was simple.

4.2.3 IBUS misalignment problem

The RC receiver sends IBUS frames over UART periodically, without checking if they are received. That proves to be a problem when 32 bytes are received during one DMA transfer. Since UART line is active 43% of the time, there is a high probability that the DMA transfer is started during this window. This leads to DMA transferring the end of one IBUS frame and the beginning of the other. This can be observed in the UART

receive buffer where the two start bytes (32 and 64) are not in the beginning of the buffer as seen in Figure 16.



Expression	Type	Value
uart_buf	uint8_t [32]	0x20000028 <uart_buf>
uart_buf[0]	uint8_t	220 'Ü'
uart_buf[1]	uint8_t	5 '\005'
uart_buf[2]	uint8_t	220 'Ü'
uart_buf[3]	uint8_t	5 '\005'
uart_buf[4]	uint8_t	220 'Ü'
uart_buf[5]	uint8_t	5 '\005'
uart_buf[6]	uint8_t	220 'Ü'
uart_buf[7]	uint8_t	5 '\005'
uart_buf[8]	uint8_t	51 '3'
uart_buf[9]	uint8_t	243 'ó'
uart_buf[10]	uint8_t	32 ' '
uart_buf[11]	uint8_t	64 '@'
uart_buf[12]	uint8_t	220 'Ü'
uart_buf[13]	uint8_t	5 '\005'
uart_buf[14]	uint8_t	220 'Ü'
uart_buf[15]	uint8_t	5 '\005'
uart_buf[16]	uint8_t	220 'Ü'
uart_buf[17]	uint8_t	5 '\005'
uart_buf[18]	uint8_t	220 'Ü'

Figure 16. Screenshot of UART buffer contents during misalignment

Three different solutions for this problem were proposed:

- Ignoring the fact that some of the data is from the previous frame. 7 ms delay is not detectable when controlling the robot with RC. Using the two start bytes to find the beginning of the frame and process data, jumping up to the beginning of the buffer when the end is reached. This is the easiest solution to implement, but creates a multitude of problems. The CRC value can not be used, since some of the data is from a different frame. The buffer end can split a 16bit integer in half, meaning its high byte and low byte would be from different frames.

- Reading one byte at a time from UART peripheral. Implementing a FIFO buffer, that the ISR of UART would fill and the IBUS thread would empty. Activating IBUS thread periodically to process any data in the FIFO. Updating the global values only once CRC has been verified. This approach would be similar to the

Arduino IBUS library used in previous prototypes [4] . It is more complex to implement than the previous one, but would avoid faulty data.

- Realign buffer when misalignment is detected (using the two start bytes). This would mean discarding the currently active frame, but all the next ones would be valid. Since it is much more frequent for misalignment to happen during startup than the working state of the robot, the alignment process usually needs to be done once. This approach also allows the DMA to signal the IBUS thread to process the data immediately after the transfer is complete.

It was decided to use the realign method, since it allows for immediate response after CRC has been received. Also, it means that the thread does not have to be activated while frame is still being transferred. Realignment was implemented using the line IDLE interrupt of the UART peripheral. The interrupt would only be activated when realignment is needed. The ISR will then abort the ongoing DMA transfer and relaunch it. After relaunch, the two start bytes will be at the beginning of the buffer.

4.2.4 IBUS receive algorithm

As a conclusion to previous chapters an algorithm and program code was created for the task responsible for receiving and processing IBUS data, as seen in Figure 17. The IBUS task enables UART idle line interrupt and then uses the osSignalWait function. This suspends the thread until the osSignalSet is called from some other thread or interrupt. The UART idle line interrupt is responsible for launching DMA transfer between UART and IBUS buffer. It also disables itself, so the interrupt will not be called again until realignment is needed. Once DMA transfer is complete, the callback function signals the IBUS task to process the buffer. The IBUS task then checks the buffer for valid start bytes and correct CRC value. If this fails, UART line idle interrupt is enabled for realigning the buffer. If buffer validation is successful, the RC values (decimal numbers from 1000 to 2000) are converted to vehicle commands (i.e. vehicle brake command from 0.0 to 1.0). After this a new DMA transfer for 32 bytes is started and the IBUS task once again waits for signal to start processing.

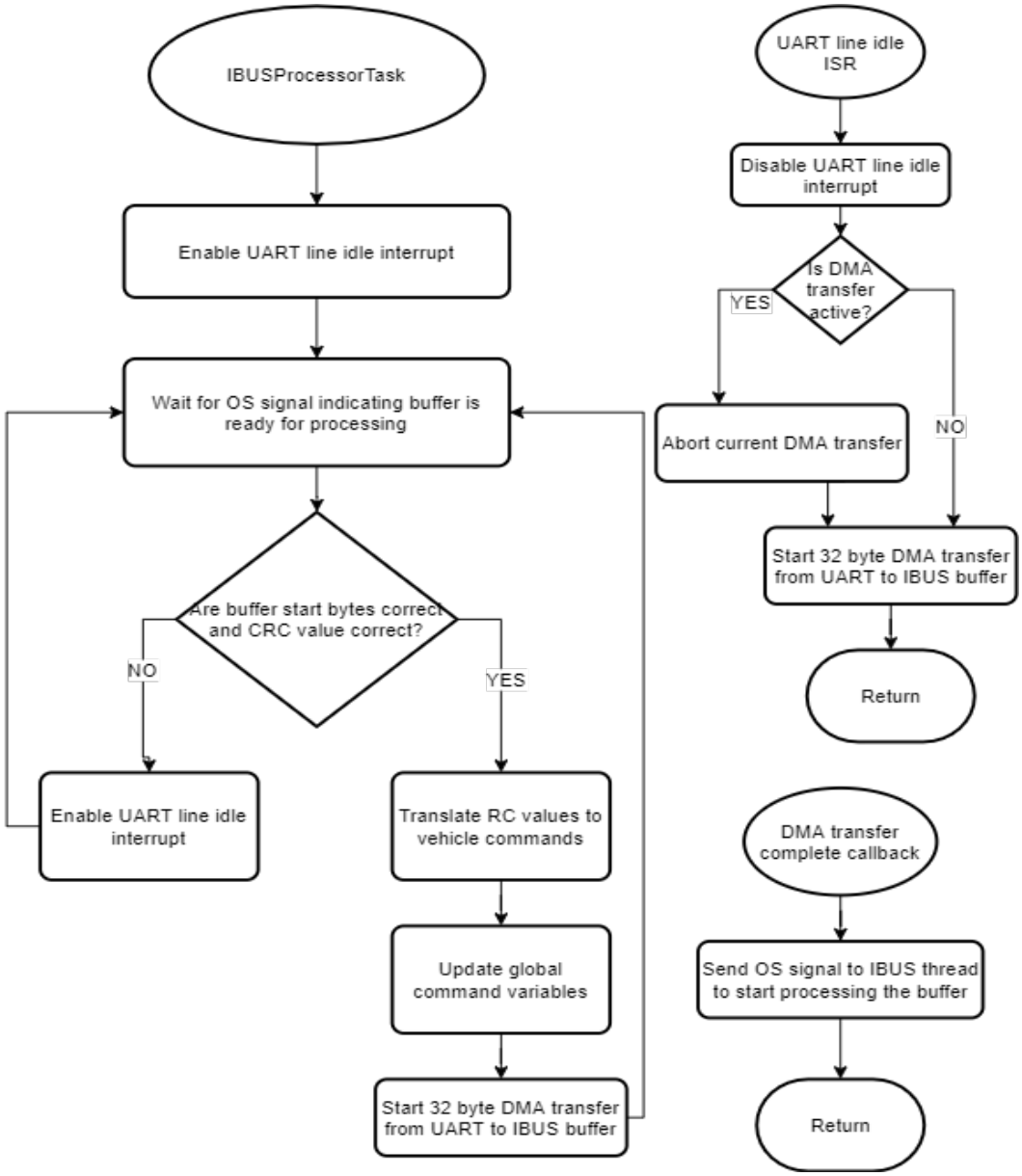


Figure 17: IBUS receive algorithm

4.3 Communication with higher level computer over CAN

The main method of robot control is planned to be over CAN interface. “The Controller Area Network (CAN) is a serial communication bus designed for robust and flexible performance in harsh environments, and particularly for industrial and automotive applications.” [24]

In CAN network, all messages are broadcasted to all nodes. Each message has a fixed ID number that defines its contents. Message ID is also used to set message priorities. A smaller number ID has priority over a larger number. Nodes filter out messages that are relevant to them. When 2 nodes try to broadcast a message in the same time, the controller whose message has lower priority backs down [24] .

To make the bus more reliable, CAN uses differential signaling. This means that the current logic state is determined by the voltage difference of the two bus lines called CAN HIGH (CANH) and CAN LOW (CANL). A logic “0” drives the bus into dominant state, meaning the differential voltage is above the threshold. And logic “1” drives it to recessive state, where the voltage difference between CANH and CANL is below the threshold. Managing the differential voltage of CANH and CANL is done by a CAN translator. The MCU uses CanTx and CanRx lines to communicate with the translator [24] .

The structure of the CAN frame can be seen on Figure 18. The frame starts with a Start-Of-Frame bit. After this comes identifier field that contains the message ID. Next is Remote Transmission Request bit, that states if the sending node requests data from another node. The Identifier Extension (IDE) bit shows if this is a regular identifier or an extended identifier CAN frame. Extended identifier is not used in this work and so is not shown. R0 bit is reserved and not used. The Data Length Code (DLC) shows how many bytes of data there is in the data field [24] . Next comes the Cyclic Redundancy Check (CRC) checksum for detecting errors. If everything worked correctly then the receiving node/nodes will write a dominant bit as the first Acknowledge (ACK) bit, the second one is a delimiter bit. The frame ends with 7 recessive bits called End-Of-Frame(EOF)[24] . The identifier, DLC code and data field are colored, since they are more important in the sense of this thesis.

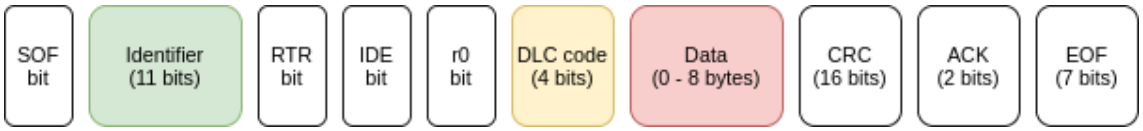


Figure 18: CAN frame

The disadvantages of CAN are its complexity and its relatively low data throughput. A single CAN frame only contains maximum 8 bytes of data.

4.3.1 HAL CAN driver initialization

Compared to the other communication peripherals, like UART, I2C and SPI, CAN is much harder to set up. There is a confusingly wide array of configuration options and corresponding HAL driver is also much more complex. To add to the confusion, a new HAL CAN driver with a new API was implemented a few years ago. There is very little documentation and examples about this new driver online. In the time of writing this thesis, the HAL user manual has not been updated with the new API functions [15] . Most posts in technical forums pointed to a single slideshow [30] and to the comments in CubeMx auto-generated C files for help on managing the API.

Before CAN peripheral can be used, it has to be initialized. In this application all configuration options except automatic bus-off management were left to their default disabeled state. Automatic bus-off management means that the CAN peripheral will try to automatically return to working state after some error has disabled it. Without it, this has to be done manually in software [38] .

4.3.2 Bit timing

An important part of initialization is setting up the baudrate of the CAN peripheral. The baudrate of this CAN bus is required to be 500 kB/s by the higher level controller. The time to send one bit is called nominal bit time. The relation between baudrate and nominal bit time can be seen below.

$$baudrate = 1/(nominal\ bit\ time)$$

Nominal bit time can be divided into 3 segments as can be seen in Figure 19. Each of those segments consist of one or multiple time quantas. A time quanta is a fixed length of time, derived from MCU clock using prescalers. The first segment is the synchronization segment where the bit change is expected to happen. This is always with the length of one time quanta. The second segment defines the location of the sample point. Even though the number of time quantas it contains is configured by the

programmer, it can be automatically changed by hardware to compensate for phase drifts. The sample point is when the CAN controller samples the bit and is usually shown in percentage of nominal bit time. The third segment is the last part of the nominal bit time and can also be automatically changed by the hardware.

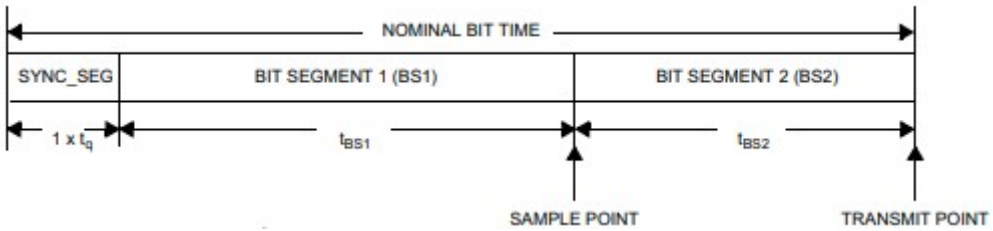


Figure 19: Nominal bit time[38]

The CAN configuration tool in CubeMx expects the programmer to configure the following values: number of time quanta in segment 1 and 2, prescaler and resynchronization jump width (SJW). The recommended SJW is one time quanta [9] . To calculate the remaining values, online CAN bit time calculator is used [9] . The calculator expects the clock rate and sample point position as inputs. The CAN controller receives the Advanced Peripheral Bus (APB) clock. The prescaler for APB clock is currently set to 2 and the main clock of the MCU is set to 96 MHz. That means the incoming clock rate for CAN peripheral is 96/2 = 48 MHz. The recommended sample point is 87,5% [9] . With those inputs, the calculator generates a table listing different baudrates and recommended prescalers and segment lenghts to achive them. The table can be seen on Figure 20 with recommended configurations highlighted as yellow. With baud rate 500 kbit/s, the recommended prescaler is 6, length of segment 1 is 13 time quantas and length of segment 2 is 2 time quantas.

Bit Rate	accuracy	Pre-scaler	Number of time quanta	Seg 1 (Prop_Seg+Phase_Seg1)	Seg 2	Sample Point at	Register CAN_BTR
1000	0.0000	3	16	13	2	87.5	0x001c0002
1000	0.0000	4	12	10	1	91.7	0x00090003
1000	0.0000	6	8	6	1	87.5	0x00050005
800	0.0000	4	15	12	2	86.7	0x001b0003
800	0.0000	5	12	10	1	91.7	0x00090004
800	0.0000	6	10	8	1	90.0	0x00070005
500	0.0000	6	16	13	2	87.5	0x001c0005
500	0.0000	8	12	10	1	91.7	0x00090007
500	0.0000	12	8	6	1	87.5	0x0005000b
250	0.0000	12	16	13	2	87.5	0x001c000b
250	0.0000	16	12	10	1	91.7	0x0009000f

Figure 20: CAN bit time table [9]

4.3.3 CAN message filter

After CAN peripheral initialization, the programmer has to configure the hardware filter. This will allow to choose which messages will be received and which ones will be blocked based on message ID. The filtering will happen on CAN peripheral level, meaning processor time will not be wasted. Since the current implementation only has 2 CAN nodes, no messages are filtered. The filter still has to be initialized to define which CAN receive FIFO is used.

Only the initialization part of setting up the communication is auto-generated. This means that programmer has to write his/her own program code for configuring the filter and for the processes described in the next section. The HAL CAN driver API used for this is described in the HAL CAN driver source file.

4.3.4 CAN Rx thread

CAN Rx thread is responsible for reading messages from CAN bus. CAN peripheral has two recive FIFOs to store incoming messages. Both FIFOs have room for 3 complete messages. The FIFO used is defined in the filter configuration[38] .

The algorithm for CAN Rx thread can be seen in Figure 21. Receiving CAN messages is implemented using interrupts. The Rx thread activates the interrupts and then waits for the operating system signal. While waiting, the thread will not be scheduled. If a message arrives into the CAN Rx FIFO, the message pending ISR is launched. This first

disables the interrupt so it would not be called again before needed. And then gives the OS signal to the CAN Rx thread to get the message from the FIFO. The message pending interrupt is re-activated. The FIFO is then emptied and message ID specific callback functions are launched. Callback functions use structures and functions from the DBC converter to unpack and decode the data. Once the FIFO is empty, the thread goes back to waiting for the OS signal.

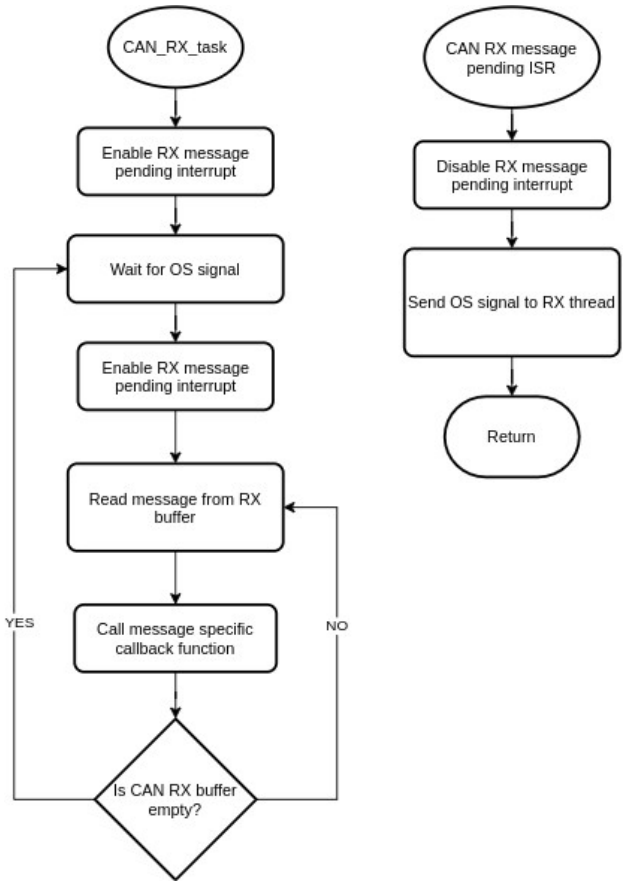


Figure 21: CAN Rx thread algorithm

4.3.5 CAN Tx thread

CAN Tx thread is responsible for sending reports (feedback) over CAN bus. In the current implementation the higher level controller expects reports with ~50 Hz frequency. This means that there should be a 20 ms delay after the last message is sent.

In the current implementation, there are 5 different reports to send: lights, speed, steering, gear and error flags. It is expected that this number will increase as development progresses. CAN peripheral provides 3 transmit mailboxes to set up messages for sending [38] . If the traffic on the CAN bus is high and the priorities of the messages in mailboxes are low, then it might take time before they get sent. This means that the thread should yield the CPU when it has no empty mailboxes.

The algorithm of CAN Tx thread can be seen in Figure 22. The report list is implemented as an array of function pointers, where each function packs and encodes the report according to the DBC file and then inserts it into a mailbox for sending.

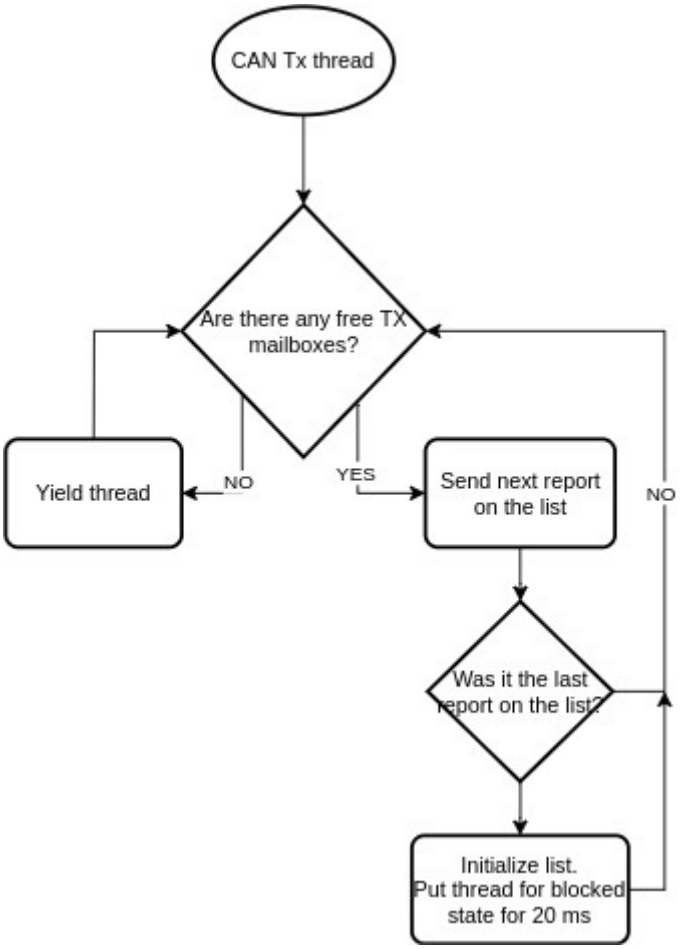


Figure 22: CAN Tx thread

4.3.6 Error report

Along with other reports, an error report is sent over CAN. This message contains information about the erroneous states STM32 controller. It allows to quickly debug the STM32 system if any problems arise.

In the program code, the error flags are grouped together as a struct. To save memory space, the struct was implemented with a C data structure called bit field. The problem with the bit field was that to check if any of the flags were up, it was necessary to for-loop through them all. That is why it was made part of an union. This allowed checking all of them together using the “raw” value in a single “If” statement. This was used with the Error LED to signal robot operator that something was wrong. The struct can be seen on Figure 23.

```
union{
	struct
	{
		uint8_t cubemx_error_handler_activated : 1;
		uint8_t can_filter_config_failed : 1;
		uint8_t can_start_failed : 1;
		uint8_t can_rx_activate_notification_failed : 1;
		uint8_t can_rx_deactivate_notification_failed : 1;
		uint8_t can_rx_get_msg_failed : 1;
		uint8_t can_tx_add_msg_failed : 1;
		uint8_t driving_motors_send_value_overflow : 1;
		uint8_t driving_motors_send_to_dac_failed : 1;
		uint8_t ibus_uart_abort_failed : 1;
		uint8_t ibus_dma_restart_failed : 1;
		uint8_t ibus_signal_timeout : 1;
		uint8_t ibus_buffer_misaligned : 1;
		uint8_t ibus_buffer_crc_mismatch : 1;
	} ;
	uint64_t raw_data;
}error_flags;
```

Figure 23: Error flags struct

The flags were set when the error happened. If it was possible for a system to recover from it, for example IBUS misalignement error, the error flag was reset on the occasion.

4.4 DBC file

CAN database (DBC) file is used to describe the data over CAN bus. This allows defining the messages and signals across the bus so that all devices would encode and decode the raw data same way. A custom DBC file was created for this project by the author. It can be observed in the restricted Error: Reference source not found.

4.4.1 DBC file format

The DBC file can be used to describe all aspects of the CAN bus. But in this application, it is only used to define the messages and signals they contain. In Figure 24 can be seen a message that contains two signals. The parts highlighted with red are syntax elements. A message definition starts with “BO_”. Then follows the message id, the name of the message and its size in bytes (DLC). The last part is message transmitter. If this is used, the nodes have to be defined in an another part of the DBC file. If it is not used, it has to be left to “Vector__XXX” [10] .

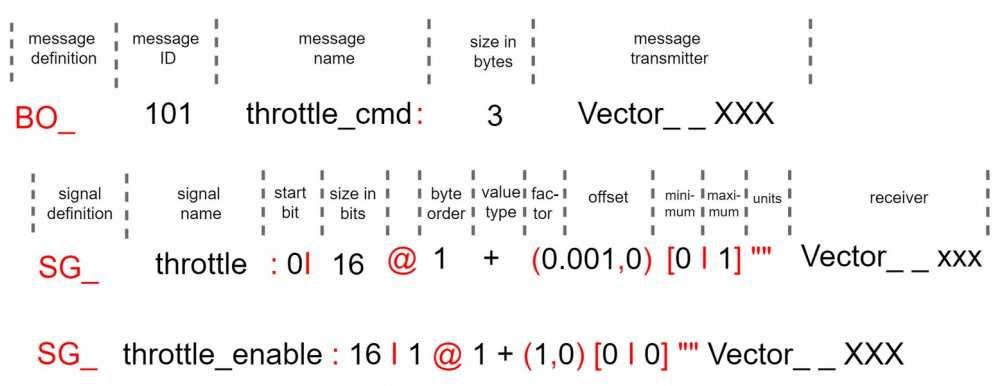


Figure 24: DBC message definition example

The second and third lines in Figure 24 are for defining signals. They have to start with “SG_”. Next comes the signal name. The start bit is used to define on which bit of the message does the signal start on. Then comes the size of the signal in bits. The “1” after the “@” sign signals the byte order: “1” stands for litte-endian and “0” for big-endian. The “+” symbol indicates that this signal is of type unsigned. Minus sign would mean type signed. Next comes the factor and offset parameters. These values allow converting between the raw value used in the transfer and physical value representing actual states.

This allows for more resolution and transferring floating point values. The formulas to transform them are as following:

$$physical = (raw * factor) + offset$$

$$raw = (physical - offset) \div factor$$

Enclosed in the square brackets are the minimum and maximum physical value of the signal. The physical value gets constrained between them. If they are both 0, no constraining is done. Next comes the unit of measure. For example, this could be "kmph" meaning kilometers per hour. If the signal has no unit, then just quotes are used. The last part is the receiver of the signal and is analogous to the transmitter part of the message definition.

4.4.2 DBC to C converter

The STM32 HAL library has no support for DBC files, so this functionality must be found elsewhere. Converter programs were researched that would generate C structures and functions from the DBC file. While testing different converters, the author found two that were easy to use and parsed the project specific DBC file. "dbcc" program by GitHub user Howerj [1] and Python3 module "cantools" [32] . The dbcc program had two shortcomings. Firstly, it included the message ID into function names. That meant that when message ID was changed in the DBC file (to adjust message priorities), program code would have to be corrected as well. Secondly, the program did not generate definitions for message ID-s and Data Length Codes, meaning that the user had to create them manually. The Pyhton3 based "cantools" did not have those problems.

The generated C code provides message specific structs and functions to pack/unpack and encode/decode the CAN messages and signals. The process is illustrated in Figure 25. To send a message, first a message struct is created that has to be filled with encoded signal values. An example of the struct can be seen in Figure 26. Encoding the signal means converting it from the physical value (variable type floating point) to the raw value (variable type integer) using factor and offset as described in the DBC file format chapter. If factor and offset is not used, then the signal does not have to be encoded. The filled struct is then inserted into the packing function that creates an

unsigned integer array with the length of the DLC code of the message. The HAL function for CAN transmit is then called and given the array as a parameter.

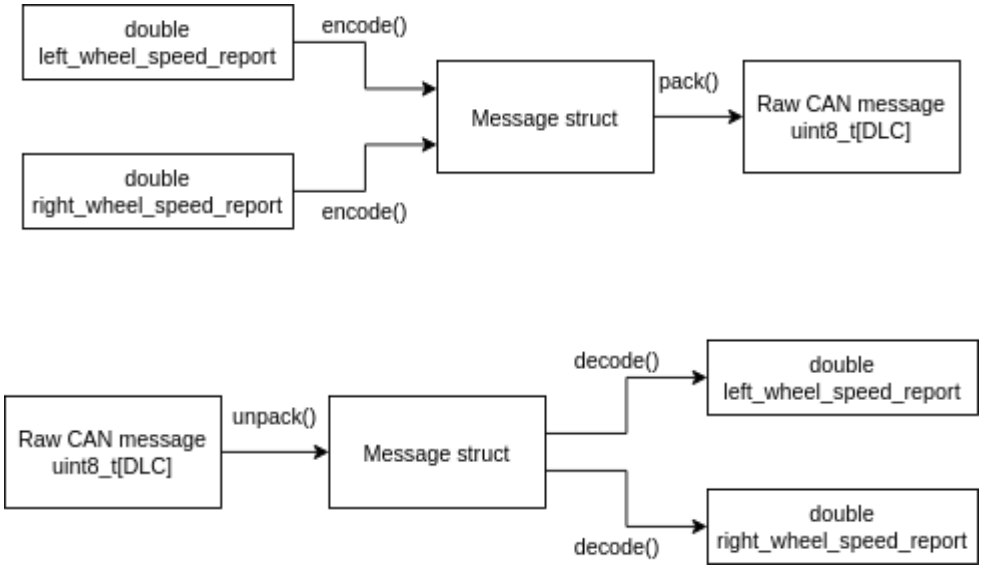


Figure 25: CAN message packing and unpacking

```
struct albert_dbc_wheel_speed_report_t {
    /**
     * Range: -
     * Scale: 0.01
     * Offset: 0
     */
    int16_t left_wheel_speed_report;

    /**
     * Range: -
     * Scale: 0.01
     * Offset: 0
     */
    int16_t right_wheel_speed_report;
};
```

Figure 26: DBC wheel speed report struct

5 Testing and analysis

This chapter focuses on the testing and analysis part of the development. It describes the real life testing and its results. Also analysis is done on the processor usage, risks and future plans.

5.1 Real life testing

The program code was tested on the test rig during development. Once the software was mature enough, testing was performed on the robot prototype.

5.1.1 FreeRTOS power bug

During the first time the program code was tested on the real robot, a weird phenomen was found. Each time the power was cycled on the entire robot, the MCU failed to initialize. If reset button on the Nucleo board was then pressed, the MCU booted up normally. The problem was reproduced with a simple FreeRTOS program code that had only one thread and blinked an onboard LED. The blink code without FreeRTOS middleware did not have this problem. That pointed towards a problem in the FreeRTOS layer. After some testing, it was managed to fix the problem, by inserting an one second delay after HAL_Init() and before SystemClock_Config() as can be seen in Figure 27. The author speculates, that the power was not stable enough during the system clock initialization part if done immediately.

```
int main(void)
{
  /* Reset of all peripherals, Initializes the Flash interface and the
Systick. */
  HAL_Init();
  /* USER CODE BEGIN Init */
  HAL_Delay(1000); //Delay to fix the power cycle bug,
  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  // . . .
```

Figure 27: Beginning of main function

5.1.2 First trial run

After removing minor bugs, the prototype was ready for the first trial run. Both control over RC and CAN bus were tested. All parts of the software were working as intended. With RC, manouvering on higher speeds was difficult due to the small size of the joysticks. With remote control over CAN, this was not an issue, since a steering wheel was used.

On Figure 28, the Albert prototype can be observed driving on the test track. On Figure 29 an operator can be seen controlling the robot remotely.

Figure 28: Robot on the test track

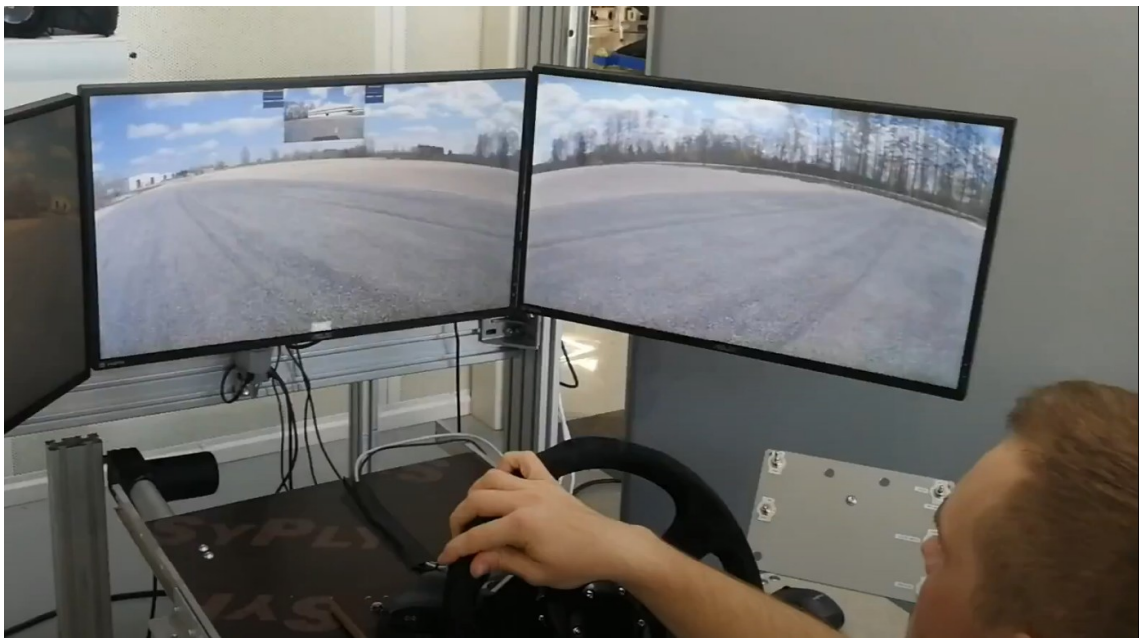


Figure 29: Remote control station

During the testing, it was observed that the moving speed of the steering servo motor is too sluggish for convenient operation. To fix this problem, it was proposed that the

motor would be given “boosted” setpoints. The error between the current value and the set value would be added to the set value, thus increasing the and making the steering servo move faster. For example, if the motor has to move the wheels The actual implementation of this feature was tasked to another team member, so it will not be covered in this thesis.

5.2 Processor usage

The FreeRTOS debugger was used to analyse the stack usage and runtime of the threads. As can be seen on Figure 30, the stack usage of CAN_tx_task is reaching the allocated 504 byte limit. This limit was increased manually in the CubeMx FreeRTOS configuration to ensure that stack overflow would not happen.

The runtime analysis shows that the processor spends 83% of the time in IDLE task, meaning the software only uses ~17% of the processor time. This shows that the software has room for scalability. It can also be seen that driving task takes a significant portion of the software runtime. This is since the current implementation does not use the osDelay function and rather sends the I2C command to the DAC as often as possible.

Task List (FreeRTOS)

TCB#	Task Name	Task State	Priority	Stack Usage	Runtime
9	IDLE	Running	0 (0)	88 B / 504 B	0xe88f4 (83.3%)
4	driving_task	Blocked	5 (5)	372 B / 504 B	0x2372b (12.7%)
7	CAN_tx_task	Blocked	3 (3)	460 B / 504 B	0x6204 (2.2%)
2	IBUS_task	Blocked	6 (6)	320 B / 504 B	0x2914 (0.9%)
3	steering_task	Blocked	5 (5)	340 B / 504 B	0x136c (0.4%)
5	hydrobrake_task	Blocked	5 (5)	356 B / 504 B	0x10d1 (0.4%)
6	lights_task	Blocked	1 (1)	316 B / 504 B	0x433 (0.1%)
1	default_task	Blocked	0 (0)	144 B / 504 B	0x3 (0.0%)
8	CAN_rx_task	Suspended	6 (6)	336 B / 504 B	0x2 (0.0%)

Figure 30: Thread runtime analysis

Even though the program code only uses around 1/5 of the processor power, it is still easy to starve low priority threads with sub-optimal program code. For example, the

CAN TX thread algorithm states, that if no mailboxes are free, the thread should yield processor time. But in case the CAN bus cable is disconnected, no mailboxes are ever freed. Once the thread yields, the scheduler checks for the highes priority thread that is in ready state. If no higher priority threads are ready, CAN TX thread gets the processor time again to check for mailboxes. Since the lights thread has lower priority than the CAN TX thread, it is starved and never run.

5.3 Risk analysis

The robot prototype is large enough to damage itself, property and people in case of an accident. For this reason, possible risks are defined and options to mitigate them are proposed. In addition to software mitigation options proposed here, there should always be a safety operator near the robot, ready to press the remote killswitch.

Table 4: Risks and mitigation options

Risk	**Mitigation**
RC receiver gets disconnected from MCU	When IBUS thread receives a timeout it switches the robot to STOP mode.
CAN wire gets disconnected	If CAN RX thread receives a timeout it switches the robot to STOP mode.
RC transmitter goes out of range	Receiver can be programmed with failsafe values, that get sent to MCU in this occasion.
Driving thread fails to send new values to DAC over I2C.	When thread reaches timeout, it switches the robot to STOP mode. This sends a disable signal directly to motor controllers.
Hydraulic brake fails to work.	The pressure of the hydrobrake should be measured and if it does not increase, the robot is switched to STOP mode, activating the emergency brake.
Emergency brake fails to work.	Hydraulic brake can be used to bring the robot to stop.
Steering motor fails to work	Encoder should be installed that allows to monitor the real steering angle of the robot.

5.4 Future plans

In this chapter, development options are proposed for the future. These are the ideas of the author for improvement, that did not fit into the scope of this thesis.

- **Code security** In case a third party gets physical access to the MCU, STM offers some ways to block using JTAG maliciously. It is possible to protect agains unauthorized readout and accidental or malicious write/erase operations.
- **Non-volatile changeable configuration** Saving configuration into EEPROM so that reprogramming would not reset it. Also, it will not be necessary to reflash the MCU to change configuration.
- **Watchdog** In case the software crashes, the watchdog would reset the controller.

6 Summary

During the work on this thesis, a new controller for the Cleveron package delivery robot prototype Albert was chosen. After the selection, software for it was developed.

STM32-F767ZI based Nucleo-144 board was chosen as the new controller for the robot. The board featured sufficient number of GPIO pins and I2C, CAN, SPI and UART peripherals. STMicroelectronics also provides CubeMX tool that simplifies the configuration and initialization by auto-generating code.

Software framework was based on FreeRTOS and HAL. Real Time Operating System allows for better timing and more efficient CPU use. STM32 HAL library made programming easier by providing an easy to use API for controlling the peripherals.

Program code was created to allow the robot to receive commands and send feedback over CAN bus. For better management of the bus, CAN database file (DBC) was created. An external DBC to C converter was chosen to generate C structures and functions. Those were then implemented into robot program code.

An IBUS driver was written to receive and translate Radio Control transmitter data. RC remote is used for maneuvering the robot when it is too dangerous or inconvenient to do it over CAN. RC remote is used to control which movement commands the robot acts on, CAN or RC.

Software was created to control driving, steering and braking motors. Also to control the lights of the robot.

During the development, test rig was used to test the program code. Finished code was tested on the real robot. First on the stands and then on the test course. Even though the software worked as intended, testing revealed multiple aspects where the software could be improved.

References

[1] “dbcc” DBC to C converter [*WWW*] https://github.com/howerj/dbcc (21.04.2020)

[2] About RTOS [*WWW*] https://www.freertos.org/about-RTOS.html (14.04.2020)

[3] Arduino Ethernet Shield V1 [*WWW*] https://www.arduino.cc/en/Main/ArduinoEthernetShieldV1 (23.02.2020)

[4] Arduino IBUS decoder [*WWW*] https://github.com/bmellink/IBusBM (13.02.2020)

[5] Arduino Mega 2560 [*WWW*] https://store.arduino.cc/arduino-mega-2560-rev3 (23.02.2020)

[6] ARM Processors [*WWW*] https://www.arm.com/products/silicon-ip-cpu (12.04.2020)

[7] Atmel Atmega640/V-1280/V-1281/V-2560/V-2561/V : Datasheet. Atmel Corporation, 2014.

[8] Better FreeRTOS Debugging in Eclipse [*WWW*] https://mcuoneclipse.com/2017/03/18/better-freertos-debugging-in-eclipse/ (05.04.2020)

[9] CAN bit time calculator [*WWW*] http://www.bittiming.can-wiki.info/ (19.04.2020)

[10] CAN DBC file format [*WWW*] http://read.pudn.com/downloads766/ebook/3041455/DBC_File_Format_Documentation.pdf (19.04.2020)

[11] Cleveron is developing a self driving delivery robot [*WWW*] https://www.ituudised.ee/uudised/2019/03/29/cleveron-arendab-isesoitvat-kullerrobotit (10.05.2020)

[12] Cleveron testing in Viljandi [*WWW*] https://sakala.postimees.ee/6827445/cleveron-katsetab-meie-teedel-kaugjuhitavat-autot (10.05.2020)

[13] CMSIS overview [*WWW*] https://developer.arm.com/tools-and-software/embedded/cmsis (23.02.2020)

[14] DAC6574 datasheet [*WWW*] http://www.ti.com/lit/ds/symlink/dac6574.pdf (22.03.2020)

[15] Description of STM32F7 HAL and Low-layer drivers : User Manual. UM1905, STMicroelectronics, 2017.

[16] Farnell Nucleo-F767ZI page [*WWW*] https://ee.farnell.com/stmicroelectronics/nucleo-f767zi/dev-board-nucleo-32-mcu/dp/2546569?ost=nucleo-f767zi&ddkey=https%3Aet-EE%2FElement14_Estonia%2Fsearch (14.04.2020)

[17] Farnell Nucleo-F767ZI store page [*WWW*] https://uk.farnell.com/stmicroelectronics/nucleo-f767zi/dev-board-nucleo-32-mcu/dp/2546569 (11.05.2020)

[18] Farnell webpage [*WWW*] https://ee.farnell.com/ (12.04.2020)

[19] FreeRTOS info page [*WWW*] https://www.freertos.org/RTOS.html (23.02.2020)

[20] GNU General Public Licence [*WWW*] https://www.gnu.org/licenses/gpl-3.0.en.html (14.04.2020)

[21] Hobby servo control [*WWW*] https://learn.sparkfun.com/tutorials/hobby-servo-tutorial/all (22.03.2020)

[22] IBUS protocol 1 [*WWW*] https://basejunction.wordpress.com/2015/08/23/en-flysky-i6-14-channels-part1/ (13.02.2020)

[23] IBUS protocol 2 [*WWW*] https://github.com/betaflight/betaflight/wiki/Single-wire-FlySky-(IBus)-telemetry (13.02.2020)

[24] Introduction to CAN [*WWW*] https://www.allaboutcircuits.com/technical-articles/introduction-to-can-controller-area-network/ (02.04.2020)

[25] Kvaser CanKing [*WWW*] https://www.kvaser.com/software/kvaser-canking/ (03.04.2020)

[26] Kvaser Leaf Light v2 product page [*WWW*] https://www.kvaser.com/product/kvaser-leaf-light-hs-v2/ (03.04.2020)

[27] Lotte courier robot [*WWW*] https://digi.geenius.ee/rubriik/uudis/esimest-korda-avalikkuse-ees-cleveron-toob-valja-kullerroboti-mis-toob-saadetised-sulle-koju-katte/ (10.05.2020)

[28] Microchip Xplained series [*WWW*] https://www.microchip.com/development-tools/xplained-boards (12.04.2020)

[29] MIT licence description [*WWW*] https://writing.kemitchell.com/2016/09/21/MIT-License-Line-by-Line.html (23.02.2020)

[30] New HAL CAN driver [*WWW*] https://st--c.eu10.content.force.com/sfc/dist/version/download/?oid=00Db0000000YtG6&ids=0680X000006HxTW&d=%2Fa%2F0X0000000ayX%2F88jLLXCT3K5cAKBDLIwfRvrqV8wrr5Rvq0_amyQl1dk&asPdf=false (01.04.2020)

[31] NXP LPCXpresso boards [*WWW*] https://www.nxp.com/design/development-boards/lpcxpresso-boards:LPCXPRESSO-BOARDS (12.04.2020)

[32] Python3 cantools [*WWW*] https://pypi.org/project/cantools/ (21.04.2020)

[33] RC control protocols [*WWW*] https://www.dronetrest.com/t/rc-radio-control-protocols-explained-pwm-ppm-pcm-sbus-ibus-dsmx-dsm2/1357 (13.02.2020)

[34] Runtime statistics with FreeRTOS [*WWW*] http://blog.atollic.com/visualizing-run-time-statistics-using-freertos (05.04.2020)

[35] ST Nucleo and Discovery boards [*WWW*] https://www.st.com/en/evaluation-tools/stm32-mcu-mpu-eval-tools.html (12.04.2020)

[36] STM32 FreeRTOS MOOC [*WWW*] https://drive.google.com/drive/folders/1vj2MYBeFF7nZz2WIb9_njcO7tiFsoqsg (10.04.2020)

[37] STM32 Nucleo-144 boards : User manual. UM1974, STMicroelectronics, 2017.

[38] STM32F76xxx and STM32F77xxx advanced Arm®-based 32-bit MCUs : Reference Manual. RM0410, STMicroelectronics, 2018.

[39] TI Launchpad [*WWW*] http://www.ti.com/design-resources/embedded-development/hardware-kits-boards.html (12.04.2020)