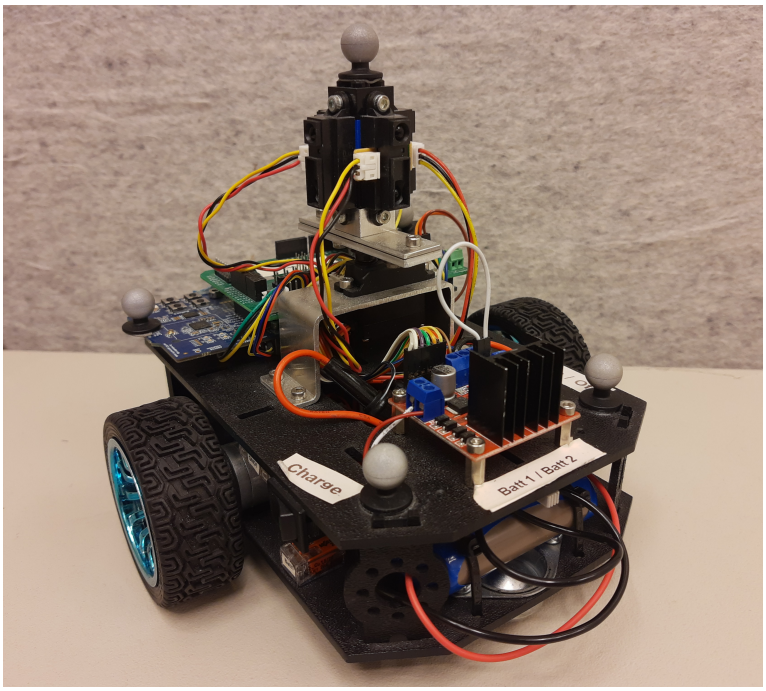
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Maria Gilje

Implementing Bluetooth LE on a MATLAB controlled nRF52840 robot

Master's thesis in Cybernetics and Robotics
Supervisor: Tor Onshus
June 2022



NTNU
Norwegian University of
Science and Technology

Maria Gilje

Implementing Bluetooth LE on a MATLAB controlled nRF52840 robot

Master's thesis in Cybernetics and Robotics
Supervisor: Tor Onshus
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU

Kunnskap for en bedre verden

Problem description

The objective of this thesis is to implement Bluetooth LE on a MATLAB controlled nRF52840 robot and make it operative and able to receive and follow instructions from a server. The robot was constructed as a part of a larger SLAM project and will work together with other robots to provide the server with environment data which it uses to create a cohesive map of an area. Without Bluetooth, the MATLAB robot is disconnected from the SLAM project. This thesis will examine how to implement Bluetooth on the MATLAB robot, in an effort to include it into the SLAM project. Following the initial examination, Bluetooth will be implemented on the MATLAB robot. The implementation will need testing, and the performance of the robot will be evaluated. The final product of this thesis should be a robot which contributes to the SLAM project while also being controlled from MATLAB.

Summary and conclusion

The robot project consists of several robots working in concert with a server application written in Java to map out an area. Within this project, the MATLAB robot exists. This specific robot is based on an nRF52840-DK and is partially programmed from MATLAB. That is, the controller and estimator is implemented in MATLAB and MATLAB Coder is used to generate C code from the implemented MATLAB functions. The generated C code is used within an application for the robot which is written in C code.

The first iteration of the MATLAB robot did not have Bluetooth functionality, which caused Bluetooth implementation to be a central part of the objective for this thesis. The implementation was performed in alternating stages characterized by code merging and testing the edited application on the robot.

The final product from this thesis is a robot with a controller and estimator created in MATLAB, which is able to connect to the Javaserver and receive commands from it. The robot is able to partially follow only the first command from the server. When the robot receives further commands it gets stuck in a program loop where it periodically turns 90 degrees on its own axis and does not move laterally. The reason for this program loop was investigated, but no clear cause was found.

To conclude, the MATLAB robot is able to include a control system implemented in MATLAB as well as a stable Bluetooth functionality. The observable behaviour of the robot changed during the work, with reasons for the change being a combination of the robot application and the hardware.

Oppsummering og konklusjon

Robotprosjektet består av flere roboter som jobber sammen med en serverapplikasjon skrevet i Java for å kartlegge et område. Som del av dette prosjektet eksisterer MATLAB-roboten. Denne roboten er basert på et nRF52840-DK og er delvis programmert med MATLAB. Mer spesifikt, så er regulatoren og estimatoren implementert i MATLAB og MATLAB Coder blir brukt til å generere C-kode fra de implementerte MATLAB-funksjonene. Den genererte C-koden blir brukt inni et program laget til roboten som er skrevet i C-kode.

Den første utgaven av MATLAB-roboten hadde ikke Bluetooth-funksjonalitet, som førte til at implementasjon av Bluetooth på roboten er hovedarbeidet i denne rapporten. Implementasjonen ble utført i vekslende stadier som består av sammenfletting av eksisterende robot-program og jevnlig testing av det redigerte programmet mens det var under endring.

Sluttproduktet fra denne oppgaven er en robot med en regulator og estimator fra MATLAB som kan koble seg til Javaserveren og ta imot kommandoer fra den. Roboten klarer kun å delvis følge den første kommandoen fra serveren. Når roboten mottar flere kommandoer, setter den seg fast i en gjentakende bevegelse som innæbærer at den roterer 90 grader om sin egen akse, og beveger seg ikke framover. Grunnen til denne gjentakende bevegelsen ble undersøkt, men ingen definitiv årsak ble funnet.

Avslutningsvis fungerer MATLAB-roboten når den består av et reguleringssystem implementert i MATLAB i tillegg til å ha en stabil Bluetooth-funksjonalitet. Den synlige oppførselen til roboten endret seg under arbeidet, og årsakene til endringen var en kombinasjon av robotapplikasjonen og maskinvare.

Preface

This report is a master's thesis which contributes 30 credits to my master's degree. It is a continuation of my specialization project from the spring of 2021. Before and during the work of this thesis, the available hardware was: a robot[1] based on an nRF52840-DK and named *NRF_3*, and an nRF51 dongle. The software used was MATLAB R2021b, nRF5 SDK 15.0.0, J-Link V7.58b, Apache NetBeans 12.1, Java 8.0, and OptiTrack: Motive Tracker. All previous theses within the Robot project, as well as their associated delivery files, were available during the work leading up to this report.

The journey from formulating a problem description to finishing this report has been challenging on several levels. I have needed a lot of support, both academically and personally. Without all the support during this time, this thesis would not have been finished. For this reason I wish to thank some people who has been pivotal during my production of this thesis.

Primarily, I want to thank my supervisor, Tor Onshus, for his endless patience and invaluable guidance during my work on this thesis. Secondly, I want to thank my guidance counsellor, Ellen Beate Hove, for helping me with pushing my deadline when the previous ones was not obtainable. Next I want to thank the other students at GF313b for helpful discussions and a good work environment.

On a more personal level, I am eternally grateful for the love, encouragement, and faith my partner, family and friends have shown me. I want to particularly thank my wonderful partner, Tom Andre, for proofreading this report, and my best friend, Marte, for helping me to better understand the code and for being a great support throughout. Last, but not least, I want to thank the Sit master group for the inspiration, tools, and methods the course and participants has given me, both for the writing process as well as for life outside of the master's thesis. *I wish you all the best of luck going forward!*

[1]Five (5) reflective spheres, four (4) infrared sensors, two (2) wheels, motors, batteries, and power switches, and one (1) nRF52840-DK, servo, custom chassis, IMU, and custom shield.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This thesis is an extension of the specialization project at the department of Engineering Cybernetics at NTNU, spring 2021 by the author of this thesis, Maria Gilje[1]. The specialization project was a study on how to implement Bluetooth Low Energy (Bluetooth LE)[1] on a robot which was controlled with a controller and estimator implemented in MATLAB. For simplicity, all future references to this specific robot will be *the MATLAB robot*. The MATLAB robot is a part of a larger project, dubbed *the robot project*, and the following sections will describe the robot project, the MATLAB robot, and the motivation for this iteration of a robot within the robot project.

1.1 The robot project

The robot project was started in 2004 by Tor Onshus, and is supervised by him to this day. Starting out, the project consisted of one robot built out of LEGO with LEGO Mindstorms as the computing device. This initial iteration of the project was dubbed *the LEGO project*. It has since been expanded in several stages to where it is today, i.e. several robots which is based on a variety of microcontrollers. The inclusion of different computing devices inspired a name change for the project, resulting in the current title, the robot project. The objective of the project is to have the robots cooperate on mapping out an unknown area by driving around and sending their sensor data to a server. The server stitches the data from the different robots together to create a map.

[1]Bluetooth and Bluetooth LE is used interchangeably during the course of this thesis

1.1.1 The nRF52 robot

The first robot application was made for a previous iteration of an nRF52 robot, and has been modified to fit the newer nRF52 robot built by Jølsgård. nRF52 – and nRF51 – microcontrollers are programmed in C code, and nRF5 Software Development Kit (SDK) is used to create applications for them.

The nRF52 robot consists of components which can be categorized into three areas: hardware, software, and firmware. The hardware components are the robot with all its parts – including the nRF52840-DK – and the nRF51 dongle, the software is nRF5 SDK and the server, and the firmware is the SoftDevice, and the compiled robot and dongle applications.

1.1.2 The MATLAB robot

The first iteration of the MATLAB robot was developed by three students during their specialization project in the fall of 2020 [2, 3, 4]. The components for the MATLAB robot is as described in Section 1.1.1, except MATLAB was an additional software component. This iteration consisted of a controller, estimator and simulator written in MATLAB as well as the pre-existing code base for the robot. The simulator was used in the development phase and is not part of the robot application. To make the integration of the MATLAB code into the robot application simple, only one function was used to generate C code from. This function made a function call to a manager task, which called the estimator and controller functions. The generated code was combined with the robot application in Visual Studio Code (VS Code). VS Code was the chosen IDE for this project.

In the specialization project leading up to this master's thesis, the objective was to implement Bluetooth LE on the MATLAB robot. This objective was not fulfilled, and therefore transferred to this master's thesis. The system implemented during the specialization project was unfinished and there were a significant amount of issues with timing and Bluetooth stability. Even though the application was not functioning as desired, the familiarization of the robot project – and the MATLAB robot in particular – obtained during the specialization project was sufficient for further work to be done during this master's thesis.

1.2 Motivation

The motivations for the MATLAB robot fall into three categories: health and safety, using high level languages, and using MATLAB. The nRF52 robot is of a relatively small

size, which makes it able to access areas which are too small for humans to enter. The on-board IR-sensors also enables the robot to map out an area which is obscured by e.g. an opaque gas. These two capabilities could make such a robot important in ensuring health and safety for people while collecting needed information about an area. The resulting mapping can be used to more efficiently move a robot without SLAM capabilities through the same area.

The nRF52840 microcontroller, which the robot is based on, needs to be programmed with C – a low-level programming language. However, most specialization branches at the Cybernetics and Robotics (C&R) study program focus on using *high* level programming languages. Developing a system where C&R students can develop applications for a microcontroller using a high level language, could bridge the gap between the two areas and introduce the students to the low level world from a high level perspective. This can be obtained without needing the same focus on the lower level *programming languages* as the students who chooses a specialization within the embedded world. The breadth this bridge provides, contributes to a wider knowledge base within the already narrow field of C&R.

A common language which is used in C&R at NTNU is MATLAB. This language has libraries with functions which can be used to create complex systems with fewer lines of code compared to C. Within the MATLAB IDE there are also an abundance of extensions which have a broad range of functionality. One of those extensions is the MATLAB Coder, which can use an implemented MATLAB function and generate code in another programming language. In the case for the nRF52 robot, the MATLAB Coder was used to generate C code.

Chapter 2

Previous work

The various components of this system has different and several creators and authors. The current robot hardware was put together by Jølsgård [5]. The robot application was originally made for the first version of an nRF5x robot, and the latest revision before the MATLAB robot was for the nRF52840 robot and was created with the three robots NRF1, NRF2 and NRF3 in mind. The Javaserver and the application for the nRF51 dongle was created by previous students as well.

2.1 Hardware

The robot was built by Eivind Jølsgård[5], and is described in detail in his specialization project report. To summarize; it is based on an nRF52840-DK and a shield which interfaces the DK with actuators and sensors. Communication between the Javaserver and the robot is obtained by connecting the nRF52840-DK to an nRF51-dongle. The dongle is inserted into a computer and is connected via a COM port to the Javaserver. The source code for the dongle is unchanged and pre-compiled before the start of this thesis.

2.2 Software

The software component of the MATLAB robot consists of three parts: the server, the robot application, and the MATLAB functions. The server and the MATLAB functions are unchanged from the beginning, throughout the duration, and until the end of the thesis work. The robot application has gone through significant changes during the thesis work. All parts will be described further in their own sections.

2.2.1 Server application

The server application is written in Java, and developed and run with Apache Netbeans IDE. The set up and project files used during the work on this thesis are sourced from Jølsgård's delivery folder [5], made available for the author by Onshus.

2.2.2 Robot application

The robot application is made as a project within version 15.0.0 of the nRF5 SDK, and includes drivers, functions, and definitions needed to interface the physical components of the robot. The application includes drivers and configurations for interfacing the DK with the rest of the hardware as well as a collection of tasks which run concurrently with the help of FreeRTOS. Some of these tasks were included in the source code the author started with, and others were designed during the thesis work. In the resulting system, the tasks which are most central to the robot's visible performance are vMainCommunicationTask, vMainSensorTowerTask, and vApiTask.

MATLAB group

Three students collaborated on controlling, estimating and simulating the nRF52 robot via MATLAB. They created separate MATLAB functions which they combined with a manager task and generated C code of this combined function. The generated function was called from a function implemented directly in C, API(), which in turn was started as a FreeRTOS task. Before calling the MATLAB generated function, api(), a route for the robot was set up, i.e. a one metre square. Readings from the hardware was also fetched, and the input parameters needed for api() was created and initialized before a function call for api() was performed within API(). After api() was called, the drivers were used to update the hardware to reflect the change the controller had put out.

2.3 Specialization project

The set up and familiarization with the SLAM project and the robot itself was done in the author's specialization project[1]. A short summary is included here for context.

The work done in the specialization project was mainly a study on how to introduce Bluetooth to the MATLAB robot. The Bluetooth set up was from the application delivered by Berglund in his specialization project [6]. The merge was not successful, and will therefore be continued in this thesis.

Chapter 3

Theory and definitions

There are four areas of this work which need some theoretical backing. These areas are: Simultaneous Localization and Mapping (SLAM), coordinate frames, the estimator, and the controller. Each area has their own dedicated section in this chapter.

3.1 Simultaneous Localization and Mapping

SLAM is a method where a mobile object is estimating its position and collecting data from its surroundings at the same time. The data collected is used to create a map of the surrounding area.

In the case of this thesis the SLAM system consists of separated units working together. The mobile part, i.e. the nRF52 robot, has an on board estimator which uses encoder data to estimate how far the robot has traveled from the starting point and its orientation in terms of coordinates and an angle. The calculations as well as the data from the on-board IR sensor tower is sent to a server over Bluetooth LE and compounded into a map.

3.2 Coordinate frames

The coordinate frames used for the robots and the environment are a standard cartesian XY coordinate system. This coordinate system is used in two areas: for the robot and for the server. Further details on each of them is described in the following sections.

3.2.1 Body coordinate frame

The robot's coordinate frame is placed with its origin at the geometric centre of the robot with X pointing in the direction of the robot's movement. The positive rotation is counterclockwise. The starting position of the sensor tower servo is defined to be at 0 degrees.

3.2.2 Central coordinate frame

The Javaserver has a map layout which represents the ground plane. When establishing a connection between a robot and the server, the orientation and position of the robot needs to be specified by the operator to ensure that the mapping is close to reality.

3.3 Estimator

The estimator was designed and implemented by Eivind Sjøvold, and is described in detail in his specialization project report[4]. The estimator type is a direct encoder estimator, which is a simple estimator sensitive for model inaccuracies. The equation for the estimate is shown in Equation (3.1) and with the explaining expressions in Equations (3.2) to (3.4).

$$\begin{bmatrix} \dot{\hat{x}} \\ \dot{\hat{y}} \\ \hat{\omega} \end{bmatrix} = \begin{bmatrix} v\Delta t\cos(\theta) \\ v\Delta t\sin(\theta) \\ \omega \end{bmatrix} \tag{3.1}$$

$$v_{\text{direction}} = [\text{mm per tick}]\frac{\text{Ticks}_{\text{direction}}}{\Delta t} \tag{3.2}$$

$$d_{\text{direction}} = v_{\text{direction}}\Delta t \tag{3.3}$$

$$\omega = \frac{d_{\text{right}} - d_{\text{left}}}{\text{length of wheelbase}} \tag{3.4}$$

3.4 Controller scheme

This controller was designed by Eystein Gulbrandsen in his specialization project from the fall of 2020[3]. The full derivation and implementation of the controller is described in detail in his report. A summary of the derivation is included in this thesis to contribute to a cohesive report.

The robot has two forms of movement: lateral and rotational. Both forms are in need of a controller, and this is the reasoning for deriving a controller which can be divided into two parts. The respective parts are named the turning controller and the distance controller. The turning controller will output a bidirectional output vector meant to cause the robot to turn on its current point. The distance controller will output a vector which moves the robot in a straight direction, or in a slight curve if the current trajectory is off-kilter. For ease of computation, the distance controller consists of two parts: one for the traversing and one for angle correction. The latter works similarly to the turning controller, with the difference of the output not being bidirectional, but instead having a difference in magnitude. The output from each of the main controller schemes is on the form shown in Equation (3.5), where u_l and u_r are the outputs for the left and right motor, respectively.

$$\boldsymbol{U} = [u_l, u_r] \tag{3.5}$$

The error correction term for the whole robot is calculated from the estimated position and the reference position. Equation (3.6) expresses this calculation, in which x represents the state, $\boldsymbol{X}$ is a vector containing all state variables, r denotes the reference, k is a discrete timestamp, and $\hat{\boldsymbol{X}}$ is the estimated state.

$$\boldsymbol{e}_x[k] = \boldsymbol{X}_r - \hat{\boldsymbol{X}}[k] \tag{3.6}$$

3.4.1 Turning controller

For the turning, the general error correction term in Equation (3.6) can be simplified to the scalar case for the angle. Simplifying the term makes computation faster. The new expression can be found in Equation (3.7), where k is a discrete timestamp, r denotes reference, and $\hat{\theta}$ is the estimated angle.

$$e_\theta[k] = \theta_r - \hat{\theta}[k] \tag{3.7}$$

3.4.2 Necessary parameters

The parameters needed for computing controller outputs are the position estimate, starting position, reference point, and the previous angle estimate. The starting position is specified by the operator on starting the robot and connecting it to the server. The body coordinate frame described in Section 3.2.1 is initialized upon connecting the robot to the server. The reference point is specified by the operator or by the server's exploration function. The position and previous angle estimates is calculated by the estimator.

3.4.3 Constraints

On the grounds that the controller will be implemented on physical motors, constraints need to be taken into account. The motors are controlled with a PWM input which need to be in the range $15-50\%$ to be sufficient and within voltage limits. A too low percentage will not be sufficient for the motors to move the weight of the robot, while a too high percentage will cause overvoltage in the motors.

3.4.4 Distance controller

The equation for the distance controller is shown in Equation (3.8), where u_{x_1} and u_{x_2} are calculated below in their own dedicated sections for the traversing and angle correction, respectively.

$$\boldsymbol{U}_x[k] = \begin{bmatrix} u_{x_1}[k] - u_{x_2}[k] \\ u_{x_1}[k] + u_{x_2}[k] \end{bmatrix}^{\mathrm{T}} \tag{3.8}$$

Traversing controller

The error between the reference point and the estimated position is calculated with the l^2 norm, i.e. it is a calculation of the shortest distance between the two points. This results in the error for the traversing distance controller being as shown in Equation (3.9).

$$e_{x_1}[k] = \|\boldsymbol{X}_r[k] - \hat{\boldsymbol{X}}[k]\|_2 \tag{3.9}$$

The output function for the traversing distance controller is shown in Equation (3.10), where K_{P_1} is scalar and a proportional gain.

$$u_{x_1} = K_{P_1} e_{x_1}[k] \tag{3.10}$$

Angle correction controller

The error for the trajectory angle has the same base form as the turning controller error and is shown in Equation (3.11).

$$e_{x_2} = \theta_r - \hat{\theta}[k] \tag{3.11}$$

Though this form is susceptible to cumulative growth, resulting in deviating values over time. For that reason, an integral term will be considered as well. The resulting equation for the angle correction controller is shown in Equation (3.12), where K_{P_2} is a proportional gain and K_I is the integral gain. Both K_α's are scalar.

$$u_{x_2}[k] = K_{P_2} e_{x_2}[k] + K_I e_{\theta_2}[k] \tag{3.12}$$

Chapter 4

Method and implementation

Now that the theoretical and practical background is out of the way, the next part to be presented is the methodology and overall implementation. This chapter will start with an overview of the whole system, followed by a description of the work done. Next up is an overview of the robot application and MATLAB functions, which is followed by quick guide for running the robot application on the robot.

4.1 System overview

The system consists of components which are categorized into software, hardware, and firmware. The software components are the Javaserver, MATLAB code, and robot application in C. The hardware components are the nRF52840 robot and nRF51 dongle. The firmware components are a precompiled hex-file for the dongle, a precompiled SoftDevice hex-file for the robot, and a compiled hex-file from the robot application. A graphical representation of the components and the information flow between them are shown in Figure 4.1. It includes flow within the robot application as well as the functions and interfaces which are connecting two different components.

From the beginning of this thesis to the end, some components has remained unchanged and others has gone through significant changes. The components which have remained unchanged are the Javaserver, the MATLAB code, and the precompiled hex-files for the dongle and SoftDevice. For comprehension purposes, the structure of the MATLAB code relating to the overall application is included, as well as some functionality related to the dongle and Javaserver. The former is included in Figures 4.1 and 4.3 and Section 4.3.1 The component which has gone through modifications is the robot application source code, and by extension the resulting hex-file from the building of the application. The robot application in general as well as the changes made within it will be described in detail.

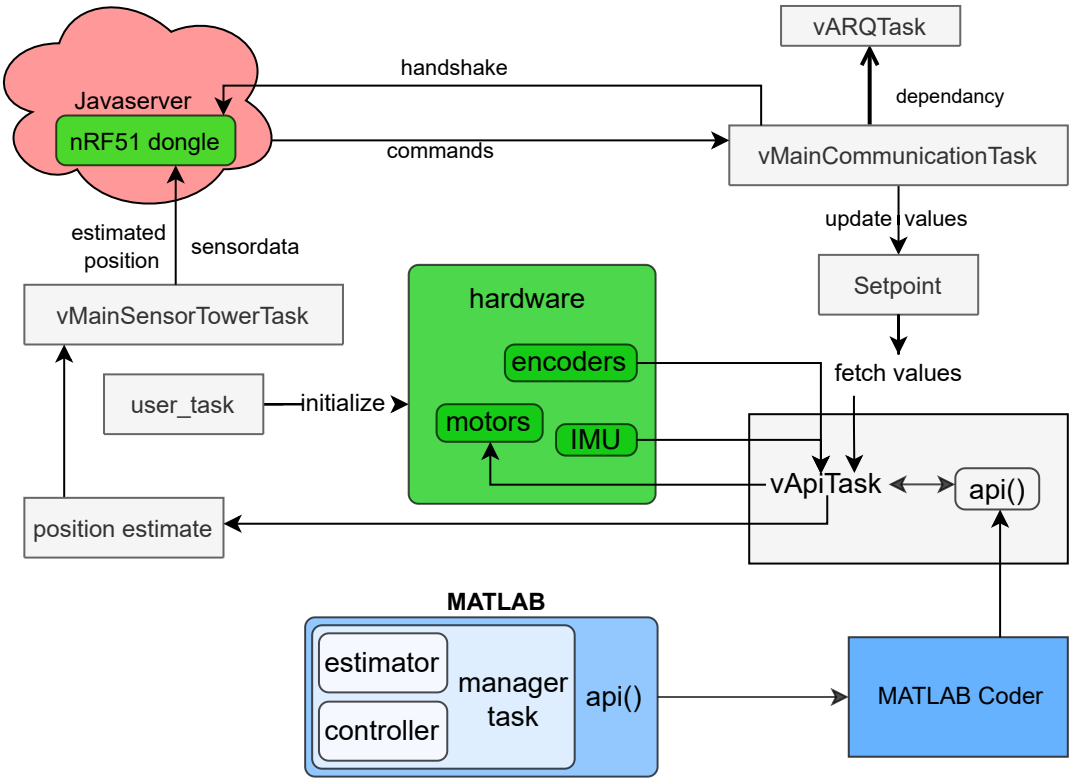


Figure 4.1: Graphical representation of robot application. MATLAB is shown in blue, hardware in green, and the server in red. All other blocks are from the robot application.

All the components which are interacting in some direct way are included in Figure 4.1.

4.1.1 Central modules

There are three central modules in this project: MATLAB generated C code, explicitly written C code, and hardware. It is necessary that the explicitly written C code is between the MATLAB generated C code and the hardware. This is because the drivers needed to interface the physical hardware is implemented within the robot application. The information flow between them can be described in three ways: Moving from MATLAB generated C code to the hardware, moving from the hardware to the MATLAB generated C code, moving back and forth between adjacent modules. These flow categories are illustrated in Figure 4.2.

All arrows in Figure 4.2 indicate direction of information flow, either between different components or functions, or across the interface of two different parts of the system. The arrows labelled A-D classifies information flow which cross the same interface(s) in the same direction. A and B represents information between the robot application and the hardware or – in the case for Bluetooth – another physical component. C and D

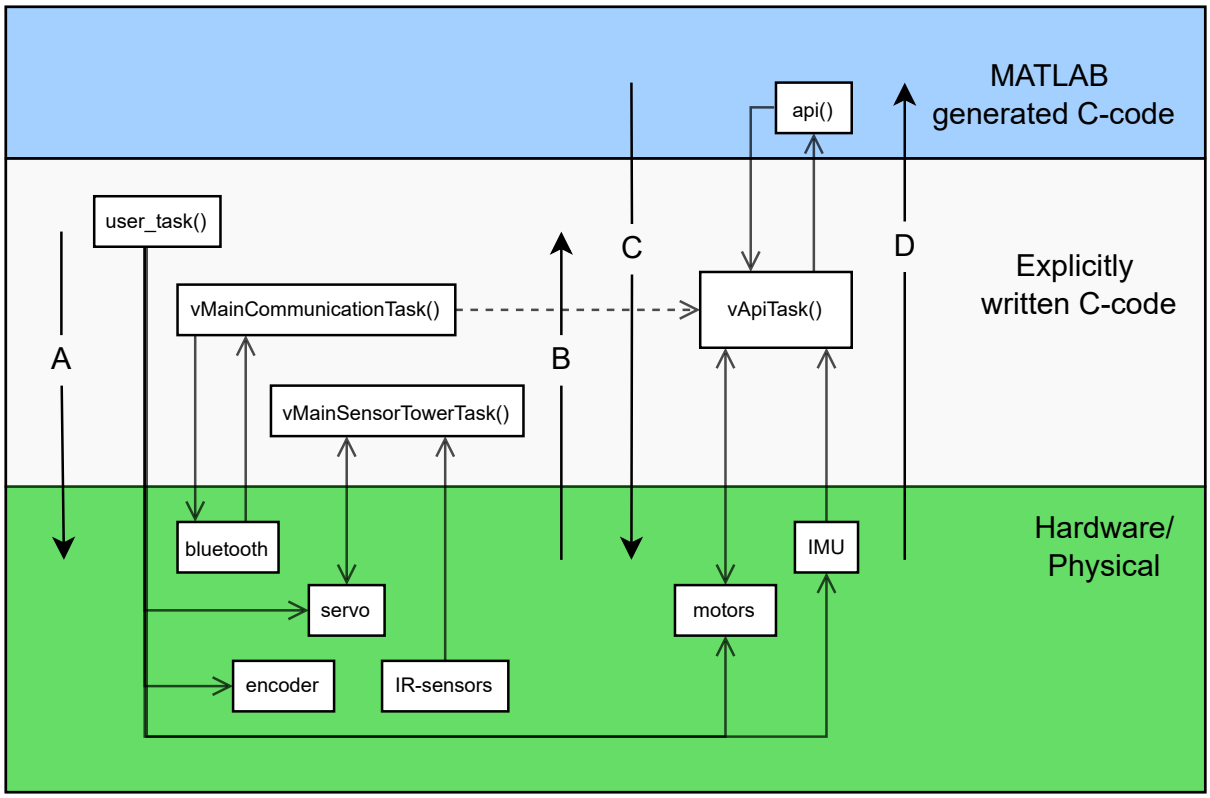


Figure 4.2: Visualization of the data flow between the three central parts of the system.

represents flow which goes from MATLAB, via the robot application and to the hardware. In Table 4.1, the specific information represented by the arrows is categorized into one of the four groups. All function names corresponds to the names used in the code for the final application. The dashed line symbolizes access to shared variables, which the origin updates and the terminus fetches.

4.1.2 Sensor tower

The sensor tower consists of a servo motor on which four IR sensors are mounted. The orientation is such that the axis of rotation is perpendicular to the ground plane and has a positive direction upwards. The servo is located at the geometric centre of the robot (seen from top-down view).

The tower is rotated 90°anticlockwise and then back to the starting position. The IR sensors send their readings to the DK which formats the data and forwards to the Javaserver. When the robot is moving to a new location, the sensor tower is idle and at the starting position.

C code ⟶ hardware flow		
A	**origin**	**termini**
initialization of encoder, servo, motors and IMU	user_task	peripherals which are initialized
messages which will be sent to Javaserver	vMainCommunicationTask	bluetooth

(a) Categorization of the information moving from C code to the hardware (physical).

hardware ⟶ C code flow		
B	**origin**	**termini**
IR and servo readings	IR and servo	vMainSensorTowerTask()

(b) Categorization of the information moving from hardware to C code.

MATLAB ⟶ hardware flow		
C	**origin**	**termini**
new motor outputs	api()	motors

(c) Categorization of the information moving from MATLAB to hardware (physical) via C code.

hardware ⟶ MATLAB flow		
D	**origin**	**termini**
IMU and encoder readings	IMU and encoder	api()

(d) Categorization of the information moving from hardware to MATLAB via C code.

Table 4.1: Categorization of information flow between the three central modules. Supplement to Figure 4.2 where the labels A-D corresponds to the arrows labeled A-D in the figure.

4.2 Implementation and testing

The implementation and testing of a new FreeRTOS task within the robot application was performed in two alternating modes. These two modes were implementation or merging of code, and testing on the hardware. Within the testing stage there were two phases as well: testing the Bluetooth connection between the nRF52840 and the Javaserver via the nRF51 dongle, and testing the response of the robot on receiving commands from the server. A detailed overview and description of the implementation and testing of the robot application will follow in this section and the subsequent subsections.

4.2.1 Choice of IDE

For the code implementation, VS Code was chosen as an IDE. The main reason for this is familiarity with the interfaces, particularly the explorer interface within the IDE. A number of included features such as the integrated terminal also contributed to the selection of this IDE. Having these resources within the same program simplified moving between files and versions of the robot application. It was also possible to maintain a git repository from the terminal, making it simpler to work on test setups and backtrack with ease if the changes made the application worse.

4.2.2 Initial available code resources

From the time working on the specialization project[1], the delivery folder from the MATLAB group was available for full use. Inside the folder, the MATLAB code as well as the robot application was found. Additionally, the reports produced during the same semester as the MATLAB group created their version of the robot application was accessed. The reason for considering other reports and their versions of the robot application was to examine how Bluetooth was implemented in those applications. The application version which was chosen to continue working on was created by Gabriel Berglund[6]. The reason for choosing Berglund's version was its readability, overall structure, and the simplicity included for changing the active robot: replacing the definition for NRF_ROBOT_2 to NRF_ROBOT_3. In addition to replacing the robot definition, the default values for NRF_ROBOT_3 was replaced by the tuned values from the MATLAB robot application.

4.2.3 Choice of application to expand

When comparing the MATLAB robot application to Berglund's robot application, some aspects with the latter stood out as preferable. The overall structure was clear and tidy, and the naming convention was consistent, intuitive, and resembled the FreeRTOS naming convention. All functions which were made into FreeRTOS tasks were declared and implemented in separate files with identifying titles. In the MATLAB robot application, the function calling api() was implemented inside main.c and made into a FreeRTOS task in main(). This implementation appeared less clear than Berglund's. This is the reason for using Berglund's method for organizing and naming functions for the new function created in this thesis work. The new function, vApiTask(), called the MATLAB generated function, api(), after the parameters api() needed was initialized. Additionally, a delay was included before and after api() was called to ensure that a Bluetooth timeout for the Javaserver connection was not reached. Said timeout was defined in the configuration for the project.

4.2.4 Choice for direction of code merging

In addition to Berglund's application having the structural perks mentioned previously, it also had the Bluetooth functionality implemented and functioning with his controller and estimator. Previous experience with nRF52 programming includes the rule of thumb of merging non-Bluetooth application into applications with Bluetooth instead of the other way around. This rule of thumb adds to the list of reasons to choosing Berglund's application as the basis.

4.2.5 FreeRTOS task requirements

When constructing a FreeRTOS task, the function needs some amount of heap to function. The original MATLAB code used a predefined route which was a one metre square. The task then only needed to iterate four times, one for each corner of the square. When modifying the setup to allow for the active setpoint to be updated during the runtime of the application, the amount of heap required by the task changed. The final amount used was found by experimenting. I.e. the original value for the MATLAB group's task was doubled, halved, maxed out (within the heap amount constraints of the microchip) and made as small as possible.

4.2.6 Setpoint storing and datatype casting

Regarding the communication with the server, the commands entered for the robot to follow was saved to a global struct variable in the robot application, and queues were used from the controller and estimator tasks to fetch the updated data. Because the server command was saved to a global variable, forwarding the coordinates to the MATLAB generated estimator and controller was fairly simple. I.e. the same queue functionality was used to get the updated values, and the X- and Y-coordinates was passed to the MATLAB generated api() as input parameters. It was made sure that the variable type matched by casting the members of the struct from the saved type: float, to the required type: double. The datatype for the data received from the server was a signed integer.

4.2.7 Testing method

The testing was performed in two phases. The first test phase consisted of running the robot application on the DK without any voltage provided to the peripherals (sensor tower servo, IR sensors, IMU, motors). This method was utilized to exclude distracting components while troubleshooting and developing the server connection to the desired

functionality. Additionally, this modularization allowed for a more gradual way of learning how to implement a function meant for a FreeRTOS task from scratch. In particular, it made setting up the hardware related variables and functions needed by the controller and estimator simpler as well as more clear. The goal for this stage was to receive a command from the server and output said command as a message in a serial terminal. By piecewise including additional lines of code in vApiTask(), this goal was achieved.

The next phase was to examine whether the setpoints from the server could be passed to api() to make the robot attempt to reach the setpoint. For this phase, voltage needed to be provided to the peripherals. During this phase, it was discovered that one of the batteries did not charge, and the rest of the work was performed with one battery. The physical change of now providing the peripherals with voltage caused new timing issues, though fortunately it was sufficient to include delays directly before and after calling api(), and modifications within api() itself was not required. The timing seemed more vulnerable when the peripherals were included into the mix than when they were out of the picture. This added vulnerability makes logical sense, because including more components into a system with several tasks running concurrently will put more pressure on the system to ensure that all tasks still are able to perform their part as well as including the new components into the mix.

4.2.8 Function creation and initial issues

Constructing the function which called the MATLAB generated api() consisted of first setting up the variables needed by api(). Each stage of the construction was built and flashed to the robot to ensure that the other active tasks all worked the same as before any changes were made. In Section 4.4.1 it is explained how building and flashing is performed. During the start phases, the functionality was not as it started out. Some understanding of FreeRTOS's scheduler was missing during this time, and a lot of trial and error to get to a functioning system was gone through. One specific tactic during the initial introduction of a new function was to have the only line of code in the new function be to output a log to a serial terminal, i.e. NRF_LOG_INFO with a text string as an input parameter. This tactic still yielded a stop of logging.

The observation from the serial terminal, J-Link RTT Viewer, was that the logging stopped at an early point during the application. Some discussion with a co-student working on the robot project, led to finding that the perpetrator was vApiTask() hoarding all the resources from the microcontroller, causing no other FreeRTOS tasks to be able to activate. This, of course, included the logger task, and the reason for why the logging stopped was suddenly apparent.

4.2.9 Issues while testing

During the experimental function construction, these were the problems which ensued:

- The logging stopped after the application had run for a bit. Usually some initialization functions were not registered as initialized, but others were.
- The robot did not show up on the Javaserver after restarting both the server and the DK. It still was connected to the dongle.
- The robot disappeared from the server after being connected for a time. The connection to the dongle was maintained.
- The robot turned around on its axis instead of idling or traversing.

The problems caused a lot of testing by varying the code composition. Some test setups were only the communication skeleton and vApiTask() under construction, while other setups omitted one or two functions which seemed to function as they should when not including vApiTask(). Though the code was tested profusely, the majority of the problems seemed to be caused by the timing issue.

One outlier was the first problem which – in addition to the timing – was caused by an endless loop in vApiTask(). The loop was endless because of missing peripheral input during the first testing phase.

The other outlier was the erroneous turning, which required meticulous testing and research to arrive at a point where some hardware appeared to be the source of the error.

4.2.10 Timing issue solution

To prevent the vApiTask() from hoarding the resources, a delay was included in the function. The result was the the other tasks were able to be activated during the pause in the new function. From this point on, it was made sure that delays were included at points in the code which had the potential of being computationally heavy.

4.3 Application

It was stated in Sections 4.2.2 and 4.2.3 that several applications suitable for the robot was available. At the beginning of the thesis, two of them were selected to be used as bases

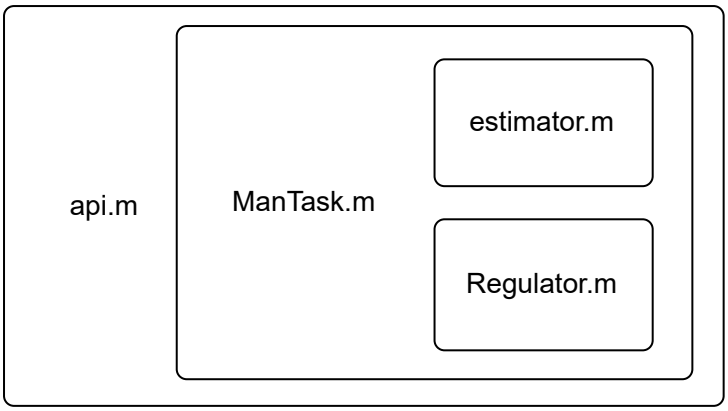


Figure 4.3: Visualization of function relation in MATLAB.

for a new application. The first of these was the MATLAB robot application, which was evident because of the nature of the thesis objective. The second application chosen was developed by Gabriel Berglund in his specialization project [6]. The general procedure for merging the applications was to use the general setup created by Berglund and replace any parts related to estimating and controlling the robot. This method was chosen because it was not in the scope of this thesis to implement Bluetooth connection from scratch, but rather to fuse a Bluetooth application together with the MATLAB control system.

4.3.1 MATLAB functions

The controller and estimator are implemented as MATLAB functions, `estimator.m` and `Regulator.m`, and combined into one task with the manager task, `ManTask.m` – also implemented as a MATLAB function. `ManTask` is called from the function `api`, which has the parameters needed to both estimate the state and compute a control output. The function relation is visualized in Figure 4.3, where the functions are depicted with boxes, and a box which is inside another symbolizes that the bigger box calls the function in the smaller box. Following the MATLAB Coder instructions for generating C code, `api` is converted to C code, and ready to be used in the robot application.

4.3.2 Robot application

The application which is flashed to the robot – the nRF52840 DK – includes driver initialization, connection establishment with the Javaserver, a controller and estimator generated from MATLAB and ran as a unit, and a rotation pattern for the IR sensor tower. The flow of the application is that most operations are placed within tasks which are managed by the FreeRTOS scheduler. The scheduler makes it possible for tasks to run pseudo concurrently when the application is running. Each task is given a heap amount

enabling it to perform its objectives. The microcontroller has a maximum amount of heap which the sum of all heap amounts need to stay below for the application to be allowed to run.

The api() is called from vApiTask() after the hardware and estimate related variables are initialized and available as input parameters for api(). New estimates for the position and the angle are calculated and passed to the controller module. The controller then calculates new outputs for the motors by using the estimates as well as the active reference point.

4.4 Running the application

When testing the system as a whole, the methods described in these sections were followed. They describe how to build and flash the robot application to the robot, and how to power up the system while making sure that each component functions as expected.

4.4.1 Building and flashing

The application is built with the C compiler cmake using a Makefile. The command for compiling and subsequently programming the microcontroller is `make flash`, and needs to be called in a prompt terminal from the project folder. When exclusively building, the command is `make`. These commands can also be found in the Makefile and redefined if needed. The commands are called in a prompt terminal from the folder the Makefile is located.

4.4.2 Method for expected behaviour

When running the robot application together with the Javaserver, the order of the steps needed is important for a successful run.

1. Begin with plugging the nRF51-dongle into the computer and launching the Javaserver through Apache Netbeans. Choose the correct COM-port for the dongle.
2. Power up the robot with its power switch while the DK is off.
3. Plug in the DK to a power source and switch it on before pressing the reset button.
4. After a short amount of time, the robot appears in the server GUI, and can be selected to connect to.

This order of execution will ensure that every component performs as expected, but other orders may also result in a correctly performing system.

Chapter 5

Results

The two main objectives was to implement Bluetooth on the MATLAB robot, and to make it able to follow instructions from the Javaserver. The Bluetooth implementation was successful, and is presented in Section 5.1. The current robot application is not able to follow all instructions from the Javaserver, though the results obtained are presented in Section 5.2.

5.1 Connection between robot and server

The robot successfully connected to the Javaserver, and the sensor tower started its rotation pattern when the connection had been established. The data from the IR sensors were received by the server, and the classification of obstacles and clear paths were correct. All coordinates entered into the UI were received by the robot.

5.2 Trajectory and reference path

The application implementation yielded a robot which was able to follow one instruction on the form $(X, Y) = (a, 0)$, where a was a positive constant, sent to it from the Javaserver. After reaching this first setpoint, the robot started to rotate 90 degrees counterclockwise in intervals. This rotation loop occurred both when another setpoint was sent from the server to the robot and when the first setpoint was the only instruction sent to the robot. The robot behaviour is visualized in Figure 5.1 where it is compared to the reference path for the robot.

A somewhat different result occurred when the first instruction required the robot to rotate before traversing. The format for such an instruction would be $(X, Y) = (a, b)$,

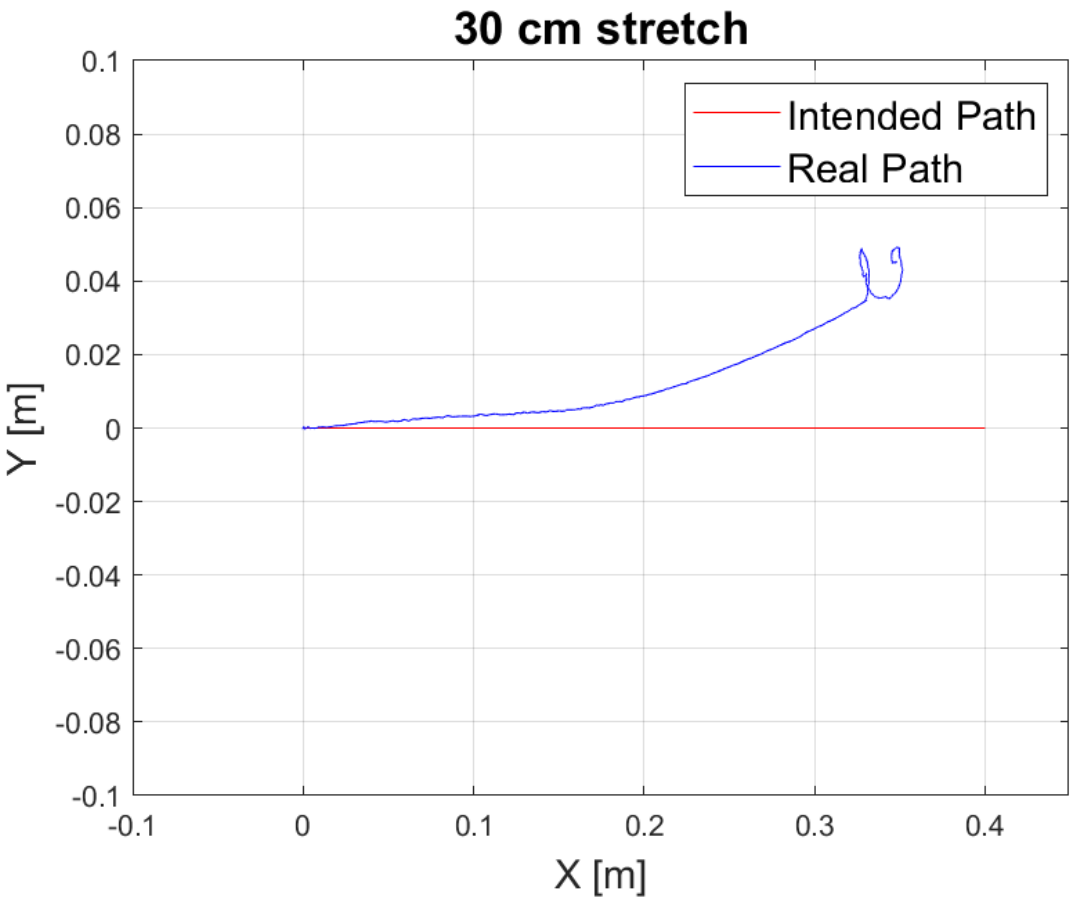


Figure 5.1: The robot's path when instructed to move to $(X, Y) = (30, 0)$ [cm].

where $a, b \in \mathbb{R}$ and also $b \neq 0$. In this case, the robot would rotate to face the correct direction, but then would continue to rotate with the same pattern as described above. The robot remained in this rotation pattern loop instead of traversing towards the instructed coordinate.

The rotation loop motivated a meticulous troubleshooting which uncovered several possible sources of error. The main two observations was that the magnitude of $\hat{\theta}$ increased each rotation and that the value of $\hat{\theta}$ would not wrap to zero when the robot had rotated a full 360 degrees. This discrepancy was tracked to the error correction function within api(). The troubleshooting also uncovered that the left encoder would sporadically output high values whilst the motor was not active. This discrepancy was not tracked towards its source, but the possibilities of its origin will be discussed in Section 6.4.

Chapter 6

Discussion

The discussion points are categorized into three groups: Choice, obstacle management, and results. Falling into *choice* is the MATLAB generated functions and the controller scheme, which are discussed in Sections 6.1 and 6.2 respectively. *Obstacle management* is centered around the hardware and discussed in Section 6.3, and *results* includes an analysis and a discussion of the robot performance found in Section 6.4.

6.1 MATLAB generated functions

The amount of functions generated from MATLAB decides which limitations, considerations and possibilities exist for the overall implementation of the robot application. Limitations include timing constraints and resource availability. Considerations consist of cooperating tasks and minimal adjustments. The possibilities are independent development and robustness. The two scenarios which will be taken into account are to generate **one** or **more than one** function from MATLAB.

When generating a single function, two limitations and one consideration present themselves. The first limitation is a timing constraint related to the Bluetooth connection with the Javaserver. The server demands periodic messages from the robot within a timeout value declared in the application source code. When a code block within a FreeRTOS task requires more time than the timeout value defines, the connection is terminated. To avoid termination of the connection, the time consuming task needs to yield sufficiently to let the other tasks run. If the MATLAB generated function – api() – executes many calculations, it may exceed the defined timeout value. At its current stage, the computation time required by api() is smaller than the timeout. However, the implemented estimator requires a conservative amount of computation time. If the estimator is replaced by a more accurate estimator – like the extended Kalman filter, the new api() could require a

computation time exceeding the timeout value.

The second limitation when generating a single function is related to resource availability and execution time. The FreeRTOS scheduler is responsible for allocating time for each task to be executed. However, each task also needs to assist the scheduler to ensure that the internal timing in every task is appropriate. Similar to the first limitation, api() requires a significant amount of time. When the FreeRTOS task which includes this time consuming code is executed without any points of delay or yielding, all other tasks are put on a permanent or extended hold. The paused tasks may contain important or useful functionality like Bluetooth, logging, or hardware related tasks. To ensure that all tasks receives their opportunities to run, the yielding in the task which encapsulates api() is again central.

The consideration when generating a single function relates to incorporating the function into the robot application. When there is only one function to incorporate, every input parameter needed is seen from the one function implementation. This visibility simplifies the incorporation compared to needing to take several functions into account. Other necessary considerations to take during the incorporation is to maintain a consistent datatype and unit for the variables which are created in the robot application and input to api().

When generating several functions, one consideration and two possibilities arise. The consideration is the requirement for the functions to cooperate. A decoupled control system may involve the two central functions to have several internal modes of operation depending on a few factors. These are; which variables are available and which point of the process the control system as a whole has arrived at. The function cooperation increases the complexity of the incorporation into the robot application, but simultaneously decreases the time requirement for each task – provided that the two components are included in separate tasks inside the robot application.

The first possibility when generating several functions is independent development. Each function can be revised and changed independently as long as the input and output parameters remains unchanged. This independence makes it possible to develop new iterations of function or module, while the other stays unchanged. The development could also be performed in parallel for two independent modules, making collaboration between developers organizationally simpler.

The second possibility is robustness. Generating several functions will result in each function being less computationally heavy compared to a single encompassing function. When taking this into consideration with the timing requirement for the Bluetooth connection, the application will gain robustness from incorporating several smaller functions. The robustness will also contribute to the overall stability of the robot application.

6.2 Choice of controller scheme

The choice made by the MATLAB group, specifically Gulbrandsen[3], was to decouple the controller into two main parts, with the lateral controller again being decoupled into two parts. This method is very tidy, and the controlling problem is simplified. Dividing the movement into two separate parts makes for a cleaner movement, and is easier to manage.

The current setup between the MATLAB robot and the server is that the robot should fulfill one command at a time. The decoupled controller will result in the most efficient route from a stopped position to the setpoint because the robot will attempt to move the Euclidean distance between the two points.

A combined controller complicates the control problem which in turn may increase computation time. For the current server–robot setup, this will doubtfully result in higher performance, but may be beneficial in a more advanced system. This hypothetical system could be based on an instruction queue from which the robot should calculate the most efficient path. The complexity increase in the server–robot setup could also require an even more complex controller than simply joining the turning and traversing controller into one.

6.3 Hardware

The loss of a battery during the implementation phase, made further testing and work flow more cumbersome. With only one working battery, the robot needed to be charged daily if testing was also performed daily. Additionally, it was not possible to read the battery percentage, and the voltage reported to the Javaserver was static, and never changed. Evaluating when the battery was running was reduced to a game of evaluating the performance of the robot on whether its faulty behaviour was caused by a functionality breaking change in the application or low battery percentage. This method of checking the battery levels, significantly slowed down both the testing and the implementation.

Though the method for evaluating the battery percentage was time consuming, and caused some tests to be carried out twice, it was still preferable to measuring the voltage and current with a multi-meter every day.

It was observed that the left encoder output erroneous values during one of the last testing sessions. This output was a considerable source of error for the calculation for $\hat{\theta}$. The most significant way the erroneous readings occurred was that they caused a great discrepancy between the estimated $\hat{\theta}$ and the observed angle of the robot. The extra

readings were sporadic and not consistent in neither value nor interval. These irregularities made the troubleshooting complicated. With the irregularities in combination with the time constraint towards the conclusion of the thesis, the reason for the erroneous encoder readings was not discovered.

6.4 Robot performance

The results from Section 5.2 the updated $\hat{\theta}$ consistently increased in magnitude when running the robot application with a test path of one point 30cm straight forward: $(X, Y) = (30, 0)$. Inside api() the if-statement in Listing 6.1 evaluated such that the turning controller was executed the majority of the iterations as a consequence of this increase.

Listing 6.1: api() deciding on distance or turning controller

```
error_theta = rt_atan2d_snf(error_y, error_x) - *gTheta_hat;
if (fabs(scale) < 0.087266462599716474) {
    // Distance controller code
} else {
    // Turning controller code
}
```

The reference angle does not change for each magnitude increase of $\hat{\theta}$, which in turn causes a magnitude increase for e_θ. When e_θ is large, the robot rotates with the minimum output for the motors in repeating rotations instead of rotating once with a larger output. When e_θ is very large, this takes a lot of time. This inefficiency reads as an issue originating in software. Though very large errors seems to only occur whilst the encoders are outputting unrealistically high tick amounts, or outputting non-zero tick amounts whilst the robot is idling – at least regarding rotational and lateral movement.

The left encoder put out high values when the motor was idle. The cause for this erroneous output is likely to be caused by a hardware issue. Or it could be a delayed reading from when the motor moved last. A delayed reading could be a software issue, or a combination of hardware and software.

Some issues with the estimator was that the estimated theta kept increasing in magnitude and not resetting at 2π radians. Code for this reset was not found in the code, though it should be sufficient for the calculation of the reference angle to be an atan2 calculation. The constants ϵ and `radEps` was decided by the MATLAB group, though a 5 degree deviation in angle and a 15 mm deviation in distance seems fair enough.

The comments in the MATLAB code includes that the previous and initial values need to be changed for the robot to be able to move from one point to another. This directly contradicts the behaviour of the robot when it was first handed over for the specialization project of spring 2021. As an initial test, the robot was started, and it was able to drive in a metre square – that is, it drove to four points. If the comment in the code was to be taken as fact, the robot should not be able to follow that route.

The active setpoint may also be interpreted by the robot as the next setpoint as well, even though the robot has already performed the instruction. With the initial position and heading always being reset to zero, this may cause issues with calculating the outputs in the controller.

It seems like the robot does not store its current orientation, and everything is calculated with the starting orientation as the heading of the robot. What the robot actually needs to do, is to calculate the error from its current heading to the next heading. The next heading is calculated from the robots current coordinates and the setpoint coordinates. Because of this method, there is no reason for the robot to store the initial position. Though, the initial position and angle which the MATLAB code uses, is constant, and does not change after the robot has followed an instruction and a new setpoint has been given.

Each instruction could be viewed as an instance of a controller sequence. That is, when the robot is within the closeness-parameters set by the controller, the new initialization is the setpoint and a calculation of the angle of the robot. This calculation will be based on where in the coordinate system the previous instruction was in relation to the starting posistion. This change would need to be an input parameter for the manager task `ManTask.m` and then the values passed to `Regulator()` would be the active initialization depending on which instruction the robot is handling. Seeing as `ManTask()` needs the augmented initialization, it either needs to be created in `api()`, or be an input parameter there as well.

Chapter 7

Further work

Following are a brief description of some suggestions for further development of the MATLAB robot after this thesis.

7.1 Improving vApiTask()

Improvements which focus on improving vApiTask() to a state where the robot is able to fulfill several instructions received from the Javaserver.

7.2 Sensor tower MATLAB module

To expand which parts of the Robot application which originates from MATLAB, a function for the behaviour of the sensor tower can be implemented. The functionality it would entail could be a rotation pattern and robot status conditions where the behaviour of the tower should change.

7.3 Extended Kalman filter

The extended Kalman filter is more suited than the direct encoder estimator because of the non-linearity of the robot's dynamics. Implementing it in MATLAB will make the estimator more computationally heavy, but also contribute to a more precise system.

Bibliography

[1] Maria Gilje. Controlling an nRF52 Robot from MATLAB. Specialization project, Norwegian University of Science and Technology, August 2021.

[2] Viljar G. Bliksvær. Simulation of the Matlab-controlled nRF52-robot. Specialization project, Norwegian University of Science and Technology, December 2020.

[3] Eystein Gulbrandsen. Controlling the nRF52-robot using Matlab generated code. Specialization project, Norwegian University of Science and Technology, December 2020.

[4] Eivind Sjøvold. Position Estimation on an nRF52-Robot Using Matlab. Specialization project, Norwegian University of Science and Technology, December 2020.

[5] Eivind H. Jølsgård. Embedded nRF52 robot. Specialization project, Norwegian University of Science and Technology, December 2020.

[6] Gabriel Berglund. Embedded nRF52840 DK Robot. Specialization project, Norwegian University of Science and Technology, December 2020.

Appendices

vApiTask()

Listing 1: Implementation of vApiTask().

```
void vApiTask(void *arg){
    vServo_setAngle(0);

    struct sCartesian Setpoint = {0, 0};

    double gX_hat = 0.0;
    double gY_hat = 0.0;
    double gTheta_hat = 0.0;
    double gLeft = 0.0;
    double gRight = 0.0;
    double leftU = 0.0;
    double rightU = 0.0;
    double ticks_Left = 0;
    double ticks_Right = 0;
    uint8_t robotMovement = moveStop;

    while (true) {
        vTaskDelay(200);
        double X_hat = gX_hat;
        double Y_hat = gY_hat;
        double Theta_hat = gTheta_hat;

        encoderTicks Ticks = encoder_get_ticks_since_last_time();
        ticks_Left = (double)Ticks.left;
        ticks_Right = (double)Ticks.right;

        IMU_reading_t gyro;
```

```
IMU_reading_t accel;
IMU_read();
gyro = IMU_getGyro();
accel = IMU_getAccel();
double gyro_x = gyro.x;
double gyro_y = gyro.y;
double gyro_z = gyro.z;
double accel_x = accel.x;
double accel_y = accel.y;
double accel_z = accel.z;

if (gHandshook) {
    xQueueReceive(poseControllerQ, &Setpoint, 0);

    vTaskDelay(100);
    taskYIELD();

    // Matlab generated C code
    api((double)Setpoint.x, (double)Setpoint.y, ticks_Left,
            ticks_Right, gyro_x, gyro_y, gyro_z,
            accel_x, accel_y, accel_z, X_hat, Y_hat,
            Theta_hat, &gX_hat, &gY_hat, &gTheta_hat,
            &gLeft, &gRight, &leftU, &rightU);
    vTaskDelay(100);

    // Cast to int before sending to motor
    int uR = (int)rightU;
    int uL = (int)leftU;

                xSemaphoreTake(xPoseMutex, 15);
    set_position_estimate_heading(gTheta_hat);
    set_position_estimate_x(gX_hat/1000); // mm to m
    set_position_estimate_y(gY_hat/1000); // mm to m
    xSemaphoreGive(xPoseMutex);

    if (checkForCollision() == true){
        motor_brake();
        robotMovement = moveStop;
        xQueueSend(scanStatusQ,&robotMovement,0);
    }
```

```
                else{
                    robotMovement = moveForward;
                    xQueueSend(scanStatusQ,&robotMovement,0);
                    vMotorMovementSwitch(uL,uR);
                    taskYIELD();
                    if(uL == 0 && uR == 0){
                        robotMovement = moveStop;
                        xQueueSend(scanStatusQ,&robotMovement,0);
                        // break;
                    }
                }
            } //endif (gHandshook)
        }
}
```

Acronyms

Bluetooth LE Bluetooth Low Energy. 1

C&R Cybernetics and Robotics. 3

DK Development Kit. 4

GUI Graphical User Interface. 20

IDE Integrated Development Editor. 5

IR Infrared. 13

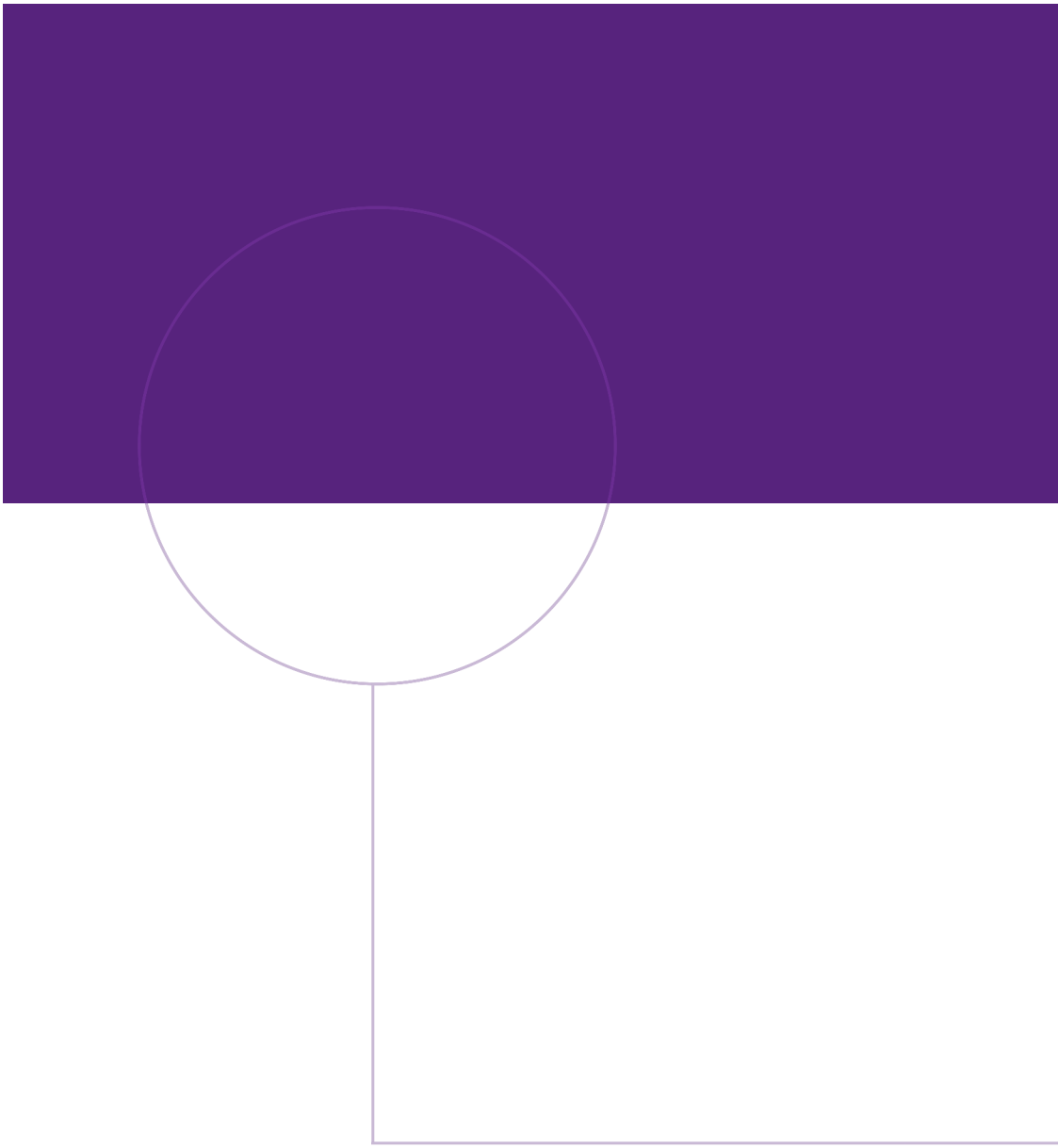
PWM Pulse-Width Modulation. 9

RTOS Real Time Operating System. 5

SDK Software Development Kit. 2

SLAM Simultaneous Localization and Mapping. 5, 6

VS Code Visual Studio Code. 2, 15



NTNU

Norwegian University of
Science and Technology