EE3L11

Motion Reference Unit Testing Platform
Software

Bachelor Thesis

in partial fulfillment of the requirements for the degree of
Bachelor of Science in Electrical Engineering

at Delft University of Technology
Faculty of Electrical Engineering, Mathematics, and Computer Science

by
Group B:
Dennis Benders, 4451546
Bastiaan Burgers, 4455347
July 13, 2018

Supervisors:
dr. ir. Rob Remis - TU Delft
dr. ir. Frank Nieuwenhuizen - Ampelmann Operations B.V.

Abstract

The Motion Reference Unit (MRU) is an important component in the Ampelmann Operation B.V. systems. In order to assess the performance of different MRUs a test system is developed. By using a one Degree of Freedom rail, wave motions can be simulated in the sway, surge and heave direction of a ship. The test system is divided in three parts: hardware, software and MRU assessment. This thesis focuses on the software design and implementation of the system. It turned out that the software performed well enough for the test system. However, due to limited project time, not all designed functionality could be implemented.

ACKNOWLEDGEMENTS

We would first like to thank our thesis supervisor Rob Remis of the Faculty of Electrical Engineering, Mathematics, and Computer Science at Delft University of Technology. He guided us through the process of the bachelor graduation project. We could always ask him for help in case we needed advice regarding the system as well as the project process.

Furthermore, we would like thank our thesis supervisor Frank Nieuwenhuizen of Ampelmann Operations B.V. for helping us with the design choices and practical issues during the system building up. Besides Frank, also Suzanne Weller, Alexander Verweij and Niels van der Geld guided us as a mentor through the process of building up the test system. All provided us with useful insights regarding the subsystems and real-time behaviour of components.

Moreover, we would like to thank Ioan Lager as project coordinator for his helpfulness during the project. He provided us all information needed to finish the project.

We would also like to acknowledge Rick Gijsberts from Go Controll for providing a useful code example regarding UDP communication in the beginning of the project.

Finally, we would like to thank our family for supporting us in busy times during the project.

Preface

This thesis is written as part of the graduation project of the bachelor Electrical Engineering at Delft University of Technology. For this project, a cooperation with Ampelmann Operations B.V. (”Ampelmann”) was set up to provide a project opportunity to be worked on, as well as the necessary assistance and supervision.

Ampelmann is a company developing offshore access systems for people and cargo. Due to sea motion, transfer from ships to offshore structures and back is usually a difficult and sometimes dangerous task. Ampelmann has developed a motion-compensating platform to make this process easier and safer.

The project is divided into three parts: hardware, software and MRU assessment. A separate thesis is written for each of these parts. This thesis will focus on all software required to develop the desired system.

Contents

1 Introduction

To compensate for the motion of a ship, first the motion needs to be measured. This is done by Motion Reference Units (MRUs). MRUs measure six Degrees of Freedom (6DoF): 3 translational (surge, heave and sway) and 3 rotational (yaw, pitch and roll). These motions are shown in Figure 1.

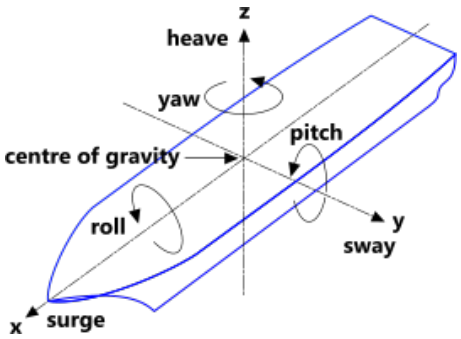


Figure 1: The six degrees of freedom of a ship on sea. [1]

MRU output values are used by the Ampelmann system, which compensates the ship's motion to keep a transfer platform stable with regard to the offshore structure. An example of such a system is shown in Figure 2.

Figure 2: Ampelmann system

It is useful for Ampelmann to test the MRU in conditions similar to the operating conditions of the Ampelmann system. Based on Ampelmann's requirements and the test results, the best suited MRU can be selected. The aim of this project is to quantify the performance of different MRUs. This will be done using a test setup on which an MRU can be moved over a 1DoF rail as shown in Figure 3.

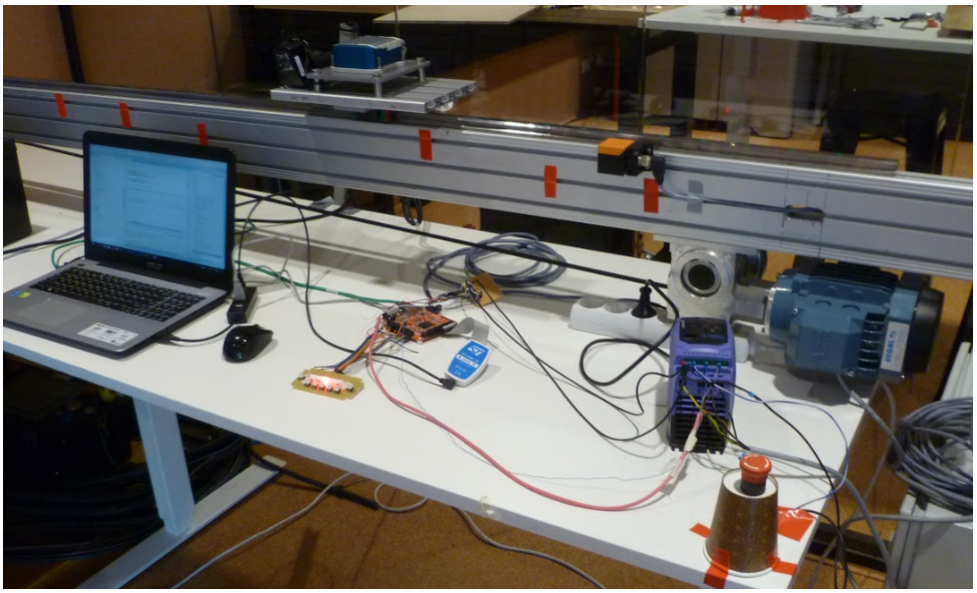


Figure 3: The 1DoF rail and the surrounding hardware.

Figure 4 shows the test system overview. The MRU is mounted on a cart which is moved on the rail using an electric motor. By controlling the movement of the motor, test scenarios, including realistic sea motions, can be simulated. The MRU output data and the position data of the MRU on the rail are collected to be post-processed and analyzed at a later stage.

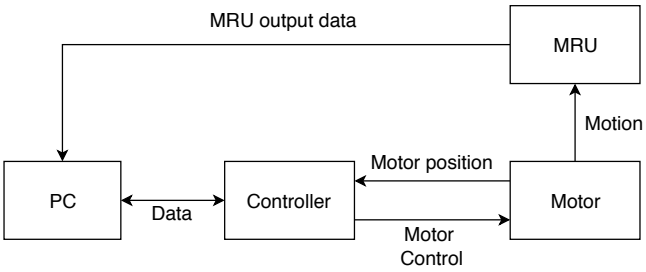


Figure 4: Test system overview

In order to simulate different types of wave motions with the test rail, software has to be written to control the test system. This section gives an introduction to the project setup for the software development during the project. Firstly, a state-of-the-art analysis is given regarding existing software which could potentially be used for this kind of real-time systems. Finally, the exact problem definition and thesis outline are described.

1.1 State-of-the-art analysis

This section does not only consist of scientific literature, but, in addition, also of market research and validation via multiple websites, because the scientific literature does not have the purpose of giving a good overview of which programming languages are mostly used and well-known.

The block diagram shown in Figure 4 consists of two components that require running software tools, which need to be determined in the thesis, in order to fulfil its desired functionality: a PC and a controller. These components will be described below.

1.1.1 PC

First of all, a PC in the test setup is used to analyse the results coming from the MRU and to provide a GUI. The PC shows the computed results to the user, so the software running on the PC should:

1. give the opportunity to easily program a GUI
2. be able to easily extract the frequencies used in the wave model
3. be able to communicate with the controller

In the PC a microprocessor executes all program functionality. However, this microprocessor can only understand machine code, consisting of bit sequences. Machine code is created from assembly language by the assembler. Assembly code, in turn, is generated by the compiler, which gets the high-level language algorithm as input [2]. Furthermore, several high-level languages exist, which can be based on other high-level languages, such as C [3]. From this can be concluded that the higher the programming language, the more functionality is included per line of code. Although high-level languages also have drawbacks when creating a user-interface [4], this will not apply in our case with a relatively simple GUI. Therefore, the GUI can best be programmed by a relatively 'high' high-level language, which are mostly class-based or object-oriented. Important and well-known examples of these types of languages can be found on the website of TIOBE.[1] This website provides a famous index on programming languages, which is also referred to in the scientific literature [5] [6]. From this list the most important high-level languages can be derived. These are listed in Table 1.1.

Besides the GUI, it is also good to have a feature on the PC to have easy switching between frequency- and time-domain representations of signals, which merely involves the wave model spectra and their time domain signals in our case. This results in the desire for a programming language in which this switching can be done in a few lines of code. Fortunately, this is the case in all above-mentioned high-level languages.

Finally, the software on the PC should also be able to communicate with the controller. This communication should take place either via Universal Serial Bus (USB), Ethernet or another serial connection (eg. RS-232 or RS-422). Fortunately, this third requirement is also satisfied by all programming languages, as Table 1.1 shows. Note that data receiving data and data post-processing are separable tasks, which can be executed by different programming languages.

From this table can be concluded that all programming languages are suitable for communicating via a certain connection protocol, whether this is USB, Ethernet or another serial connection does not really matter.

[1] https://www.tiobe.com/tiobe-index/

Table 1.1: Important and well-known high-level programming languages [1]

Programming language / Communication type	**Java**	**C**	**Python**	**MATLAB**
USB	+	+	+	+
Ethernet	+	+	+	+
Serial	+	+	+	+
GUI	+	+/-	+	+
FT/IFT	+	+	+	+

1.1.2 Controller

Furthermore, a controller is needed to control the input signals of the motor driver, which will finally result in a motion of the MRU. This system is a typical example of an embedded system and it is thus likely to be implemented by using a microcontroller [17]. The software running on the controller should:

1. fit the real-time application of controlling the linear motion system
2. be able to communicate with the PC
3. be able to send data to the motor controller

As mentioned above, the higher the programming language, the more functionality could potentially be implemented by one line of code. This means that one function calls other 'lower-level language functions', so it is a less efficient way of working and will negatively impact the performance of the real-time system. Therefore, the language level should be relatively 'low' [18]. Because of the fact that C allows to program on a relatively low level, it is mostly used to program microcontrollers, especially since the introduction of the Arduino platform [19] [20].

Of course, the controller also has to communicate with the PC and the motor controller. The communication with the PC is already described in Section 1.1.1. From this can be concluded that the controller should be able to communicate with the PC via USB, Ethernet or another serial connection. Most microcontrollers have an Ethernet and serial connection and an USB-to-serial converter can be used to convert the signal [21]. In order to communicate with the motor driver, firstly, the type of motor driver has to be known. For this project, the Optidrive E2 motor controller [22] is available to control the ABB 3GVA072001-CSC motor [23]. From the motor driver datasheet can be derived that the communication is via the Modbus RTU protocol [24] [25] with RS-485 signal format [26].

Now that the controller specifications are known, the controller can be implemented. As described above, the controller is likely to be implemented with a microcontroller. This microcontroller could be of the Arduino platform, for example. However, another possibility is to use a PLC [27], which also contains a microcontroller. For PLC, different languages exist of which Instruction List (IL) is a very low-level language and comparable to assembly language. IL is potentially better to use in this embedded system [28], but will take more programming time than the other PLC languages. Given the time limitations for the project, programming in this language will probably not give the desired end result. Therefore, another language, such as Ladder Diagram Chart (LDC) fits better in this case. For both types of controllers, application examples can be found to drive an induction motor [29] [30] [31] [32].

[1] The following sources were used to construct this table: [7], [8], [9], [10], [11], [12], [13], [14], [15] and [16]

1.2 Problem definition

The project's main purpose is to develop a properly working linear motion test system in order to evaluate different types of MRUs in different wave motion conditions. Nowadays, Ampelmann is using a specific, very expensive MRU, which certainly meets their system requirements. However, at this moment, Ampelmann does not have a system to test the accuracy of other MRU types, which are less expensive and have great potential of being used in future systems. This is certainly desired in order to make proper design choices in the future without needlessly spending resources. The test system is finished when Ampelmann is able to mount a sensor on the rail, press a start button and the system will perform all work necessary to give the final test results. These test result should at least be given in a data file and preferably in a visual interface.

The main challenge in this project is the time pressure: a useful and safe system should be designed and implemented within a time frame of circa 10 weeks.

Specifically speaking about the software, the problem could essentially be split up into two separate tasks: PC programming and controller programming. On the PC, the right wave model is created by the MRU assessment group, which should be communicated to the controller. The controller should store this data and subsequently start the real-time control of the test system. During runtime of the system, relevant data should be collected and send back to the PC, which stores it in a relevant format. The resulting data file will be given back to the software of the MRU assessment group.

1.3 Thesis outline

In order to ensure a structured explanation of all design steps taken, the next section starts with the Program of Requirements (PoR). These requirements describe the desired system behaviour and play an important role in making the design choices. The design choices will be described in Section 3. After designing the system, it should be implemented and tested. This is elaborated on in Section 4. Thereafter, a conclusion can be drawn on this project, especially focusing on the software part described above, which is given in Section 5. Section 6 provides a discussion about what could be improved and what future work is left over, followed by the list of references. Finally, Appendix A gives the test plan followed in the testing phase of the project, Appendix B the pinout of the development board used and Appendix C elaborates on the FSM by providing all fail states possible.

2 Programme of requirements

The software has to meet requirements which are necessary to be able to assess the MRUs sufficiently, according to Ampelmann's ideas. After setting up the requirements with Ampelmann and the whole project group the overall system is designed (see Figure 4). In order to implement this system, each subgroup creates new requirements for the other subgroups.

This section describes all requirements needed to develop the software of the real-time test system. The requirements are divided into categories. The first category discusses the requirements which are given by Ampelmann and which are important for the system as a whole. Thereafter, the requirements coming from the other subgroups are described. These requirements result in the definition of interfaces between the software part and hardware and MRU assessment part of the system, which is necessary for proper system functioning, but also allows the subgroups to develop parts of the system in parallel.

2.1 Ampelmann requirements

Ampelmann's requirements regarding the system as a whole can be divided into several categories: first of all, the overall system requirements. Secondly, specific system requirements and thirdly, specific software requirements. These requirements are described below:

The following overall system requirements are applicable:

1. System should be able to test performance of different MRU types for sway, surge and heave motions. This can be done in 3 different simulations
2. System should be able to run simulations for a relatively long time. 24 hours is considered to be sufficient
3. System should be safe to use
4. System should be easy to control
5. System should be designed in such a way that it can easily be adjusted for future testing purposes

Some of the overall system requirements directly result in more specific system requirements, which are listed below:

6. System should be able to simulate fluent wave motions, according to the North-Sea wave characteristics: Ts = 3-7 [s] and Hs = 0 - 2.5 [m] [33]. Other kind of wave forms are nice to have. It is desired to be able to select the previously simulated wave form in order to allow for quickly setting up a new test. Furthermore, the ship Response Amplitude Operator (RAO) used to create the wave form should represent the ships Ampelmann uses in its offshore operations (resulting from requirement 1)
7. In order to assess the heave motion reactions of an MRU, the test rail also has to function when placed vertically (resulting from requirement 1)
8. System should give the possibility to stop the system on the PC and by using a hardware button (resulting from requirement 3)
9. System should always be able to safely exit the program flow and go back to a safe state (state in which no mechanical movement is present in the system) in a controlled way (resulting from requirement 3)
10. System user should be able to control the system via hardware buttons. In this case the PC is only connected for communicating the wave form to be simulated (resulting from requirement 4)
11. It is preferable to provide the possibility to read out MRU data using already developed readout scripts in Matlab (resulting from requirement 4)
12. System should be designed in a modular way, this means well-defined interfaces between the different modules (resulting from requirement 5)

Ampelmann also has some requirements regarding the software. These are given below:

13. Software and needed packages should be free to use
14. Software is preferably already in use at Ampelmann
15. Software should be well documented and built up according to Ampelmann's standards
16. Software should be able to read out different MRU types (with different data formats)
17. Software should be able to send data real-time and asynchronous to the controller. The idea is to be able to send wave form data at a certain frequency (with possibly introduced delays) and to handle and process this data in a constantly running low-level control loop
18. Control loop should be able to send information to the motor driver at a frequency of at least 50 [Hz]
19. User Datagram Protocol (UDP) should be the standard communication protocol (between controller, PC and MRU)
20. Controller should function as a black box. In other words: the behaviour when controlling the system with the PC or with hardware buttons should be known (controller software will most likely not be changed)
21. The system should be able to stop the car when crossing a safety margin to prevent the cart from sliding of the rail under all circumstances

2.2 Overall system design

Using the above requirements, the overall test system can be designed. Figure 5 shows the overall system block diagram. This design step is necessary to define all interfaces between the different blocks. Furthermore, by designing the overall system, the subgroup dependencies become clear. The requirements given by the hardware and MRU assessment group, which are important for the software part, are described in Subsections 2.3 and 2.4, respectively.

2.3 Hardware requirements

The hardware subgroup is merely concerned about the hardware implementation of the system and all associated safety requirements and implementations. Most of the requirements set by this group are related to the system hardware limitations or components already available. The requirements are listed below:

22. Software should be able to run on the controller hardware
23. Test rail has a limited motion range (1.86 [m]), so the wave to be simulated has to fall within the safety range of the rail. The system should be able to stop the cart before it reached the end of the rail
24. Communication between motor driver and controller is achieved using an analog connection. The maximum update frequency of the motor driver is 1.44 [kHz]
25. An inductive proximity switch is used to determine the cart position on the rail [34]. The controller has to be able to read out this sensor and directly take action when this sensor indicates that the cart is at the same position
26. A quadrature incremental encoder [35] is used to follow the cart movement along the rail. The controller has to be able to read out the quadrature incremental pulse signal coming out of this encoder with a maximum refresh rate of 300 [kHz]. The encoder is not infinitely accurate: an accuracy of 44 [µm] can be reached, which should not give any problems in this project

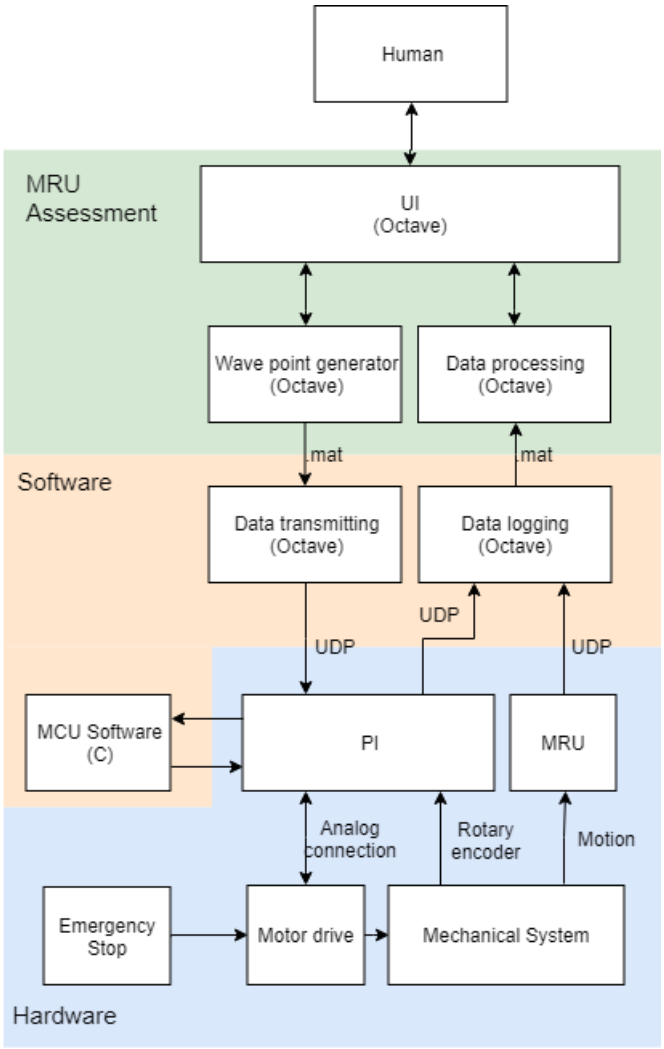


Figure 5: Block diagram test system

2.4 MRU assessment requirements

27. Wave forms with a maximum frequency of 0.5 [Hz] have to be simulated
28. The limited range of the test rail and safety margins of 15 [cm] on both rail sides results in a maximum wave motion amplitude of 78 [cm]. This is where the MRU assessment group checks on. To be practically able to simulate wave forms with exactly 78 [cm] amplitude, the system should not stop when the outermost wave positions are reached. Therefore, the software has to set a safety margin just beyond the outermost wave points
29. A maximum time error of 1 [ms] in time stamping is allowed with a 99% confidence interval

The exact content of the defined interfaces in the given block diagram do not only depend on the other subgroups, but also on the software part. Therefore, these interfaces will be discussed in more detail in the next section describing software design.

3 Design

This section describes all design choices and justifications made during the project. For example, some practical choices have to be made regarding devices and software languages to be used. Furthermore, the communication protocol between PC and controller has to be designed in order to have a structured way of communicating, which is understandable for people who are working with the system for the first time. All design choices either concern the PC or controller. Therefore, this section is split up into a part describing design choices concerning software running on the PC and a part for the controller.

3.1 Devices to be used

In order to design a system capable of simulating wave motions on a 1DoF rail, at least two requirements should be satisfied:

- The wave to be simulated should be constructed/calculated
- The wave form should be followed by the cart

For the first requirement a PC will be used, because it is implicitly stated as a requirement from Ampelmann (in order to easily read out MRU data using old readout scripts) and it allows the user to quickly control data (also in a visual way). This can be achieved by using the programming language Matlab, which gives the possibility of implementing a GUI, as can be derived from Section 1.1.1. Ampelmann personnel has experience with Matlab programming, so this would not give a problem. However, in order to have all communication according to the UDP protocol (as stated in requirement 19), the Instrument Control Toolbox has to be installed. This requires a special license. Therefore, Octave is chosen to be used in the project. Octave can be seen as a free version of Matlab with some differences. A disadvantage of Octave could be the relatively high difficulty level of programming a GUI. Given the fact that Ampelmann personnel has programming experience, a GUI is not necessarily needed, but just a User Interface (UI) does also satisfy the requirements.

A disadvantage of using a PC could be the inaccuracy of communication at a certain frequency. As requirement 17 states, the PC has to send wave data to the controller at a fixed frequency. Because of the fact that a high-level language is used on the PC and it is running on an operating system, the PC may not be able to ensure this exactly fixed frequency. The same holds for receiving and timestamping of incoming packets. This should be tested for, which will be described in Section 4. The last requirement can only be achieved in a real-time system containing a control loop in order to simulate waves accurately. From Section 1.1.2 can be concluded that a microcontroller is likely to be used for the implementation of the real-time functionality. Using a PLC could be another option. Given the background of the students in this project group and the relatively short development time of 10 weeks available, the microcontroller is chosen to be used. Furthermore, choosing a microcontroller is also in line with the student's educational goals and interests. After all, the main goal of this project is to learn and gain experience in systematically designing a technical system in cooperation with a customer.

A microcontroller is likely to be programmed in C language, which also satisfies the requirement of a low-level control loop. Programming in C can be done with a lot of Integrated Development Environments (IDEs). However, a microcontroller also needs some peripherals to communicate with the outside world, so a development board with integrated microcontroller is needed. The hardware subgroup set up all requirements for speed and external communication and came to the conclusion that the STM32F767ZI microcontroller [36] is relatively cheap and has a very high performance. However, in order to program a microcontroller, online examples are very useful, so the software group decided that the STM32F407ZGT6 Cortex M4 microcontroller [37] would be a better choice. This microcontroller also satisfies the requirements, so this controller is chosen to be used. In order to have good online documentation for the development board, the open source development board Olimex STM32-E407 [38] is chosen to be used.

For microcontroller programming in general holds the following: finding the right code implementing the actual microcontroller functionality could be relatively easy. However, finding the right code for all

desired register settings is much more difficult, because many comparable microcontrollers exist which do not exactly need the same register settings. Given the limited project time, the Eclipse programming environment is chosen, together with STM32CubeMX. STM32CubeMX is a program that allows the user to graphically set up all peripheral connections with corresponding settings. From these settings, STM32CubeMX can automatically generate the source code with all required register settings. In this way the register settings still have to be determined, but the exact source code is automatically generated.

To summarise, a PC is used together with the programming language Octave to construct the wave forms and the Olimex STM32-E407 development board with STM32F407ZGT6 Cortex M4 microcontroller (programmed in C in the Eclipse environment, together with STM32CubeMX) is used for the real-time part of the system.

3.2 PC

Figure 5 shows the blocks of the software part in which the PC is used: data transmitting and data logging. All design choices made in developing these blocks will be described in Subsections 3.2.1 and 3.2.2. Furthermore, the interface specifications will be given in these sections. It is important to keep in mind that the data transmitting and data logging are described in separate sections, but they actually run in 'parallel' in Octave: wave positions have to be sent at circa 50 [Hz] to the microcontroller, but real-time data from the MRU and microcontroller also has to be received and timestamped. These processes should ideally run at the same time. However, Octave only allows for sequential programming, so the functionality will be implemented in an 'infinite loop' (until the test is finished), where the processes are integrated into each other.

3.2.1 Data transmitting

The data transmitting block is located between the wave point generator and the microcontroller. Therefore, the interfaces between these blocks have to be defined. Furthermore, this block executes data processing on the incoming signal to create a logic data format to be sent to the microcontroller.

Figure 6 shows the overview of this block, including its interfaces. The blocks in the overview are separately described below.

3.2.1.1 Interface with wave point generator

The .mat file represents the interface with the wave point generator and contains the following elements:

- Sample frequency (Fs): frequency at which the continuous wave is sampled. The microcontroller needs this value to know when it can expect a new wave position to be received
- # wave repetitions (n): amount of repetitions of the total wave to be simulated. This variable is needed for the data transmitting block to indicate how many times the same sequence of wave positions has to be transmitted to the microcontroller
- Wave position array (s): array containing all wave positions to be sent to the microcontroller. The microcontroller uses this information in the motor control loop
- Wave velocity array (v): array containing all wave velocities (related to the wave positions)
- Wave acceleration array (a): array containing all wave accelerations (related to the wave positions)

Only the first three elements are used by the data transmitting block. "v" and "a" are saved in the .mat file, because they will be used by the MRU assessment group for other purposes, but they have no meaning in the rest of the system.

3.2.1.2 Data processing

In the data processing part, data- and control signals are used. All the data given in the wave point generator interface are single precision floating point numbers representing data signals in SI units (meters, meters

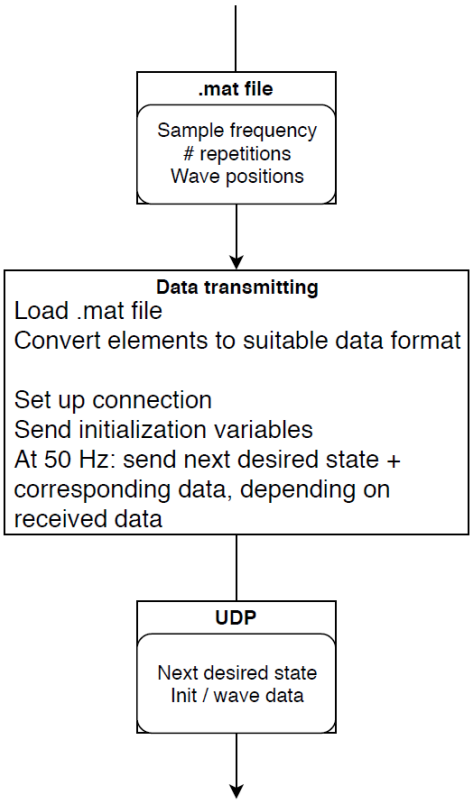


Figure 6: Overview data transmitting

per second, etc.). In order to have easy computations and readable code on the microcontroller, all data will be transmitted in SI units. Also, single precision floating points will be used for the sample frequency and amount of points, which are sent in the 'Idle2Initialized' state. The use of single precision floating points will give a fixed data packets size. This is preferable, because different data formats are more difficult to process. A fixed format leads to faster implementation. This will not give timing problems, because the micro controller has sufficient calculation power to handle two floats at 50 [Hz] (this is the mostly used MRU updating frequency within Ampelmann). Moreover, data in meters and meters per second will be used for the wave data until it is converted to a signal to be sent to the motor driver, which is a DAC voltage. This conversion will be elaborated on in Section 3.3.2. Furthermore, the data signals are provided by the wave point generator.

Besides data signals, the PC also has to send control signals to the microcontroller. These signals should indicate what the next system state should be (see Section 3.3.1 for more information regarding the possible states).

After determining the next control and data signals, this information has to be sent to the microcontroller. The next section describes how the exact UDP packet content.

3.2.1.3 Communication with microcontroller

When all data conversion is finished, the information has to be sent to the microcontroller. Communication is achieved via the UDP protocol in the transport layer. Each time the PC sends new information to the microcontroller, a new UDP packet is created and transmitted. Some variables regarding the rail set-up are sent once in the initialization phase, while the wave data is sent at 50 [Hz] (requirement 18). Wave data is sent 'continuously' in order to meet requirement 17. Furthermore, a frequency of 50 [Hz] is chosen to have a smoothly operating motor control system. Note that 50 [Hz] satisfies the Nyquist theorem: the sample frequency should at least be twice as high as the highest frequency present in the wave form (which is 0.5

Figure 7: Different communication formats PC-to-microcontroller

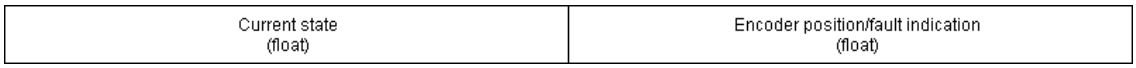


Figure 8: Communication format microcontroller-to-PC

[Hz], according to requirement 27).

In order to have fluent communication with the microcontroller and to meet requirement 5, a standard data communication protocol has to be designed. This protocol should allow the PC to send all data and control signals on time to the microcontroller, while the microcontroller should be able to give feedback regarding the current system state and encoder data.

As described above, the PC always has to send its control signals indicating the next desired state and data signals either containing initialized variables, such as the sample frequency, or wave data. Wave data could also contain an indication of a S-curve start (see Section 3.3.2 for more details about the S-curve). By choosing a fixed length for the UDP packet, there is no need for complicated data read out. The state received from the micro controller is enough for the PC to know, whether a wave points has to be send or an initialization variable, the format is shown in Figure 7.

Communication from the microcontroller side should contain the current state (to determine which next state is desired on the PC side) and the encoder position (for the MRU assessment). Figure 8 shows the standardised format for messages from the microcontroller. As can be seen in the figure, the microcontroller should also have the possibility to indicate a failure has occurred with corresponding fault indication.

3.2.2 Data logging

The purpose of the data logging block on the PC is to receive, timestamp, and save the incoming data from the MRU and encoder. This is done as shown in Figure 9. Actually, in order to get the best system performance, the microcontroller should be used for timestamping, because it can be programmed on a lower level than the PC. However, according to requirements 5, 16 and 20, different MRU types with different data formats will be tested in the future and the microcontroller should be a black box. This means that the microcontroller should be able to read out different data formats, but these formats are not known at the moment of programming. Obviously, this is not possible, so the MRUs will be read out in the data logging part on the PC. Therefore, all timestamping should also take place in this part of the system.

The first objective in this block is the connection set-up, setting the IP address, local port and remote port. Secondly, the connection is opened, in which Octave starts receiving the incoming data and puts it in a buffer. By using the `udp_read` function, the buffer is read out for the last packet. In the code the MRU is read out first followed by the encoder. Due to problems with the multi-threading functionality in Octave it is not possible to implement a function which will receive the packets in parallel. However, it is possible to use the Atlass package, which can use all processor cores and speed up the sequential process.

Timestamping on the PC leads to an extra delay. This delay consists of a constant and variable part. The constant delay can be ignored in the design, because it will be the same for both the MRU and encoder. The variable part should stay within limits, because this influences the final MRU assessment in the data processing block.

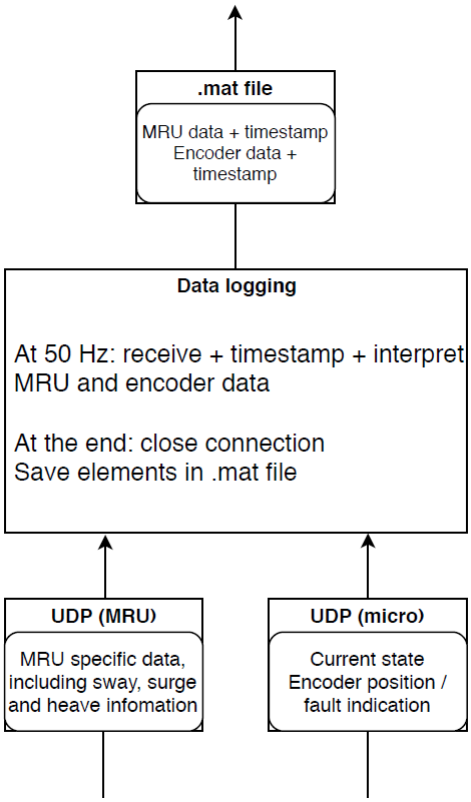


Figure 9: Data logging on PC

3.3 Microcontroller

3.3.1 Finite State Machine

By using an FSM (Finite State Machine), a well structured software code can be written on the microcontroller. During the FSM design process all requirements are taken into consideration. The final FSM results from an iterative process with the Ampelmann supervisors. This subsection describes the design philosophy, followed by a description of the main states. Thirdly, the requirements from Section 2 are described and finally the failed state is explained.

3.3.1.1 Design Principles

In order to have a proper functioning FSM, every possible defined path can be followed without a system failure. Also, it is not always easy to know what will happen when a system goes from state A $\rightarrow$ B $\rightarrow$ C and then goes from C $\rightarrow$ A. To deal with these problems, first a symmetric FSM is designed, which can later be expended with more paths. The first designed FSM is symmetric with only the possibility to go from state A to C, by passing B. But also to go from state C to A, by going through B. Figure 10 shows this implementation. The FSM also contains transition states (e.g. 'Homed2Neutral') in which checks are performed to see whether all conditions are met to be able to go to one of the main states (drawn in spheres). These checks are important to know if the system is still functioning properly. Furthermore, the transition state can also be used to let the system go into, for example, the required position in which a main state begins. Another reason to design a symmetric FSM is the likeability of the system working when programmed. First A $\rightarrow$ B and B $\rightarrow$ A can be programmed and tested. If this works B $\rightarrow$ C and C $\rightarrow$ B can be tested. Afterwards, all the combinations between state A, B and C can be tested, after which state D can be added. By doing this, the program can be build up piece by piece, while testing can be done

simultaneously. However, a well functioning FSM should not get stuck in one state because one condition can not be met. For this reason a 'Failed' state will be added, which can be triggered after a certain time period in which the conditions are not met. Also a failing input is given or a malfunctioning part can trigger the 'Failed' state. When the user detects a failure in the system an emergency button can be pressed, which will let the system go into the 'Failed' state. The design of the 'Failed' state will be explained at the end of this section, because an understanding of the FSM is important to see where failures can occur.

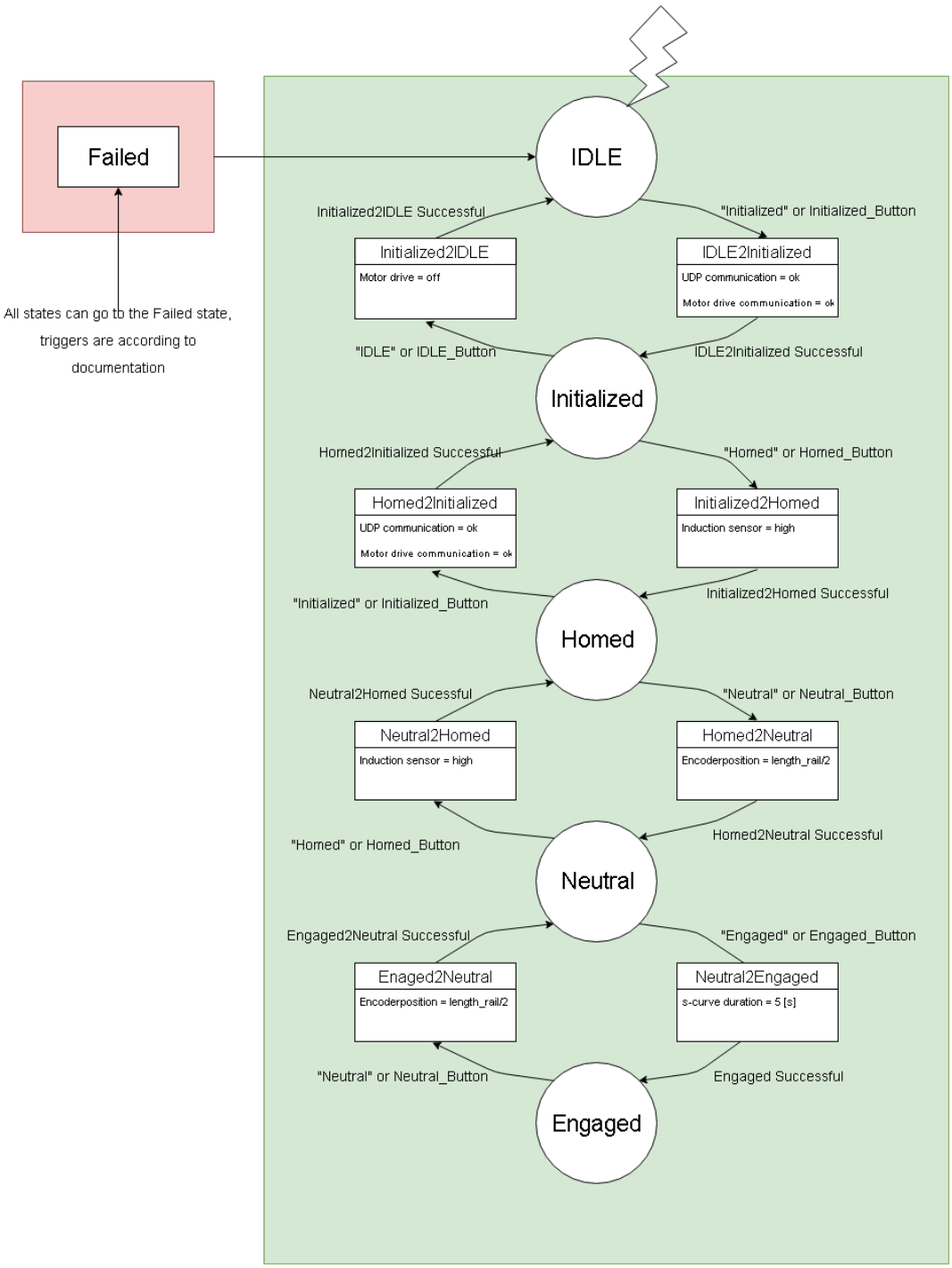


Figure 10: FSM final design

3.3.1.2 FSM description

To be able to understand the different states in the FSM, Figure 11, 12 and 12 will be used. These figures show the rails, motor and cart with attached MRU. The HMI (Human Machine Interface) is shown below the rails. The hardware interface consist of 8 buttons and 5 LEDs. Dyuring system runtime, the LED corresponding to the active state will be turned on. When the system is in a transition state, the LED for the next state will start blinking. In Figures 11, 12 and 13 a description and image from every state is also given.

3.3.1.3 Requirements

In Table 3.1 the requirements are given with the designed solution. This gives a structured view on how the designed FSM, block diagram and small choices, which are not yet discussed, contribute to meeting the requirements.

Requirement	**Design solution**
System should be able to test performance of different MRU types for sway, surge and heave motions. This can be done in 3 different simulations	The cart is able to perform wave motions in horizontal and vertical direction
System should be able to run simulations for a relatively long time. 24 hours is considered to be sufficient	By streaming data to the microcontroller, a PC is used to send data. A PC has memory in the order of GB and a 24 hours file is in the range of MB
System should be safe to use	The 'Failed' state checks the system to prevent safety issues (see Appendix C). When this is not enough an emergency button can be pressed
System should be easy to control	The 5 main states are in line with the states used in Ampelmann's systems. Furthermore, the system can be controlled by a standard script on the PC, in which parameters can manually be changed when needed. Moreover, the file with wave points can be changed by loading a new file. When no parameters have to be changed default waves and settings are already present in the set-up. Thus, the system can be used with its default settings, but can also be changed by adapting the code on the PC
System should be designed in such a way that it can easily be adjusted for future testing purposes	By using UDP as standard communication protocol, different MRUs can be read out in the future. Furthermore, the system is universally designed to test every type of wave, according to the requirements
System should be able to simulate fluent wave motions, according to the North-Sea wave characteristics: Ts = 3-7 [s] and Hs = 0 - 2.5 [m] [33]. Other kind of wave forms are nice to have. It is desirable to be able to select the previously simulated wave form in order to allow for quickly setting up a new test. Furthermore, the ship Response Amplitude Operator (RAO), used to create the wave form, should represent the ships Ampelmann uses in its offshore operations (resulting from requirement 1)	The rail is not long enough, so comparable waves with less height can be tested. Furthermore, the motor can provide enough power to reach a wave time of 3 [s] (see hardware thesis)

In order to assess the heave motion reactions of an MRU, the test rail also has to function when placed vertically (resulting from requirement 1)	The control loop will handle the vertical test setup control, because the motor will provide more power to go upwards and slow down when going downwards
System should give the possibility to stop the system on the PC and by using a hardware button (resulting from requirement 3)	An emergency button is present, which will stop the system. By pressing 'ctrl+c', the software will stop the system as well
System should always be able to safely exit the program flow and go back to a safe state (state in which their is no mechanical movement in the system) in a controlled way (resulting from requirement 3)	When the system functions properly, the user can go to the 'IDLE' state via the hardware buttons or using the PC. If the system fails, the 'Failed' state will be entered in which the cart is stopped and the system can return to the 'IDLE' state
System user should be able to control the system via hardware buttons. In this case the PC is only connected for communicating the wave form to be simulated (resulting from requirement 4)	By using hardware buttons every state can be accessed. When the user is in the 'Neutral' state and wants to go to 'Engaged', the Engaged button can be pressed and the user can start sending the wave points to the system
It is preferable to provide the possibility to read out MRU data using already developed readout scripts in Matlab (resulting from requirement 4)	The system will support these readout scripts from Ampelmann
System should be designed in a modular way, this means well-defined interfaces between the different modules (resulting from requirement 5)	This is done in the block diagram (see Figure 5).
Software and needed packages should be free to use	Octave is free to use. The same is true for Eclipse and all software needed to program the microcontroller
Software is preferably already in use at Ampelmann	Octave is already used by Ampelmann
Software should be well documented and built up according to Ampelmann's standards	This will be implemented in the code
Software should be able to read out different MRU types (with different data formats)	MRUs normally communicate via UDP. UDP packets can be received and the correct data bits can be selected by the software. Using the MRU manual, the positions of the desired data bits can be determined
A maximum time error of 1 [ms] in time stamping is allowed with a 99% confidence interval.	This depends on the delay introduced by Octave in the communication syntax and other factors, this has to be tested before certainty can be given for this requirement
Software should be able to send data real-time and asynchronous to the controller. The idea is to be able to send wave form data at a certain frequency (with possibly introduced delays) and to handle and process this data in a constantly running low-level control loop	By using a buffer, the control loop can synchronise the incoming wave points
UDP should be the standard communication protocol (between controller, PC and MRU)	This is implemented as can be seen in the block diagram (Figure 5)
Controller should function as a black box. In other words: the behaviour when controlling the system with the PC or with hardware buttons should be known (controller software will most likely not be changed)	This is done by providing the FSM of the microcontroller. Furthermore, the states are based on states used by Ampelmann, making it more intuitive

The system should be able to stop the car when crossing a safety margin to prevent the cart from sliding of the rail under all circumstances	This will be implemented in the software code. The system can also be stopped by pressing a hardware button (see hardware thesis)

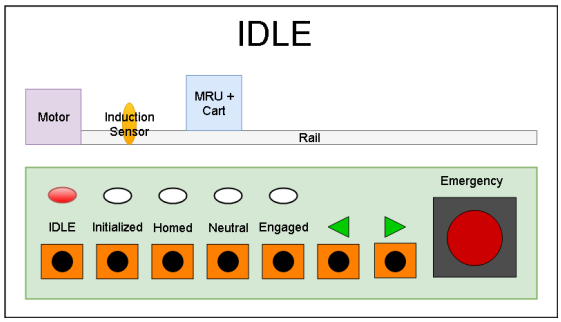
Table 3.1: Table of requirements and design solutions

3.3.1.4 Failed state

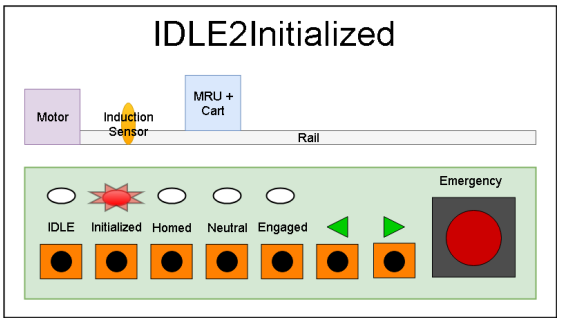
Safety is an important value within Ampelmann, because safety is hard to measure. The failed state is designed to handle as much potential risks as possible. Afterwards, the failed state is explained to Ampelmann, to check if it meets their safety standards. To be able to systematically handle failures, a list of potential failures is made. Secondly, every state is checked for handling all possible failures. Finally, a trigger is defined for when this failure will occur. Table 3.2 provides all failures that may occur in the 'IDLE2Initialized' state and the corresponding solutions. The full list of every state with the 'Failure and Solution' table is provided in Appendix C.

Failure	**Relevance and Solution**
Encoder fails (lose wire or no power)	X, not detectable no moving cart in this state
Band falls of the rail	X, not detectable no moving cart in this state
Motor driver detects a failure.	The motor driver failed state pin is low.
Lost communication with PC (maybe later other devices).	Go to Failed, when no Echo signal is received from the PC
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	X, not detectable no movement

Table 3.2: 'Failure and Solution' table for 'IDLE2initialized' state

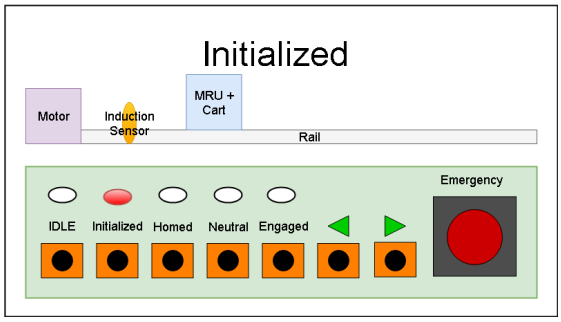


(a) When the microcontroller is powered on for the first time (or reset after execution) it goes to the 'IDLE' state. The MRU location is not known in this state and the motor is not controlled. Under normal conditions the state 'IDLE2initialized' can be entered, by sending an UDP packet with the intialization state in the header or by pressing the Initialized button.

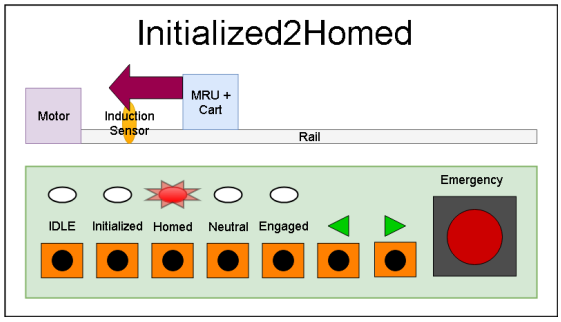


(b) In the 'IDLE2initialized' state, the micro checks for incoming intialization variables (rail length and distance to induction sensor). When these variables are received, it will enter the next state. The variables need to be send in a fixed sequence. This format is given in Section 3.2.1.

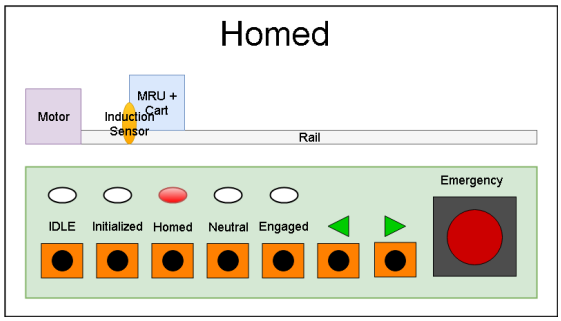
Figure 11



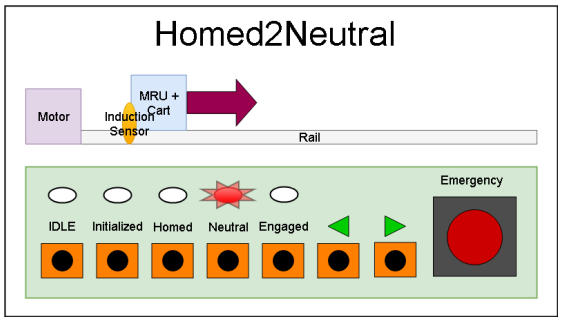
(a) In this state all transmitted parameters are saved. The motor is not active or controlled. From this state the 'Initialized2Homed' state can be entered by receiving a Homed state in the UDP packet to be send by the PC or by pressing the Homed button.



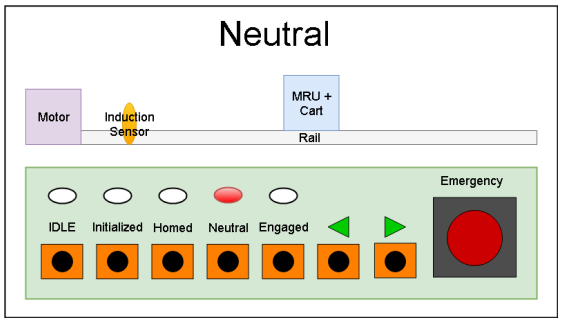
(b) This state is driving the cart towards the induction sensor, using the control loop. When the cart reaches the induction sensor the encoder position is saved as reference position. Once the induction sensor is reached the condition to go to the 'Homed' state is met.



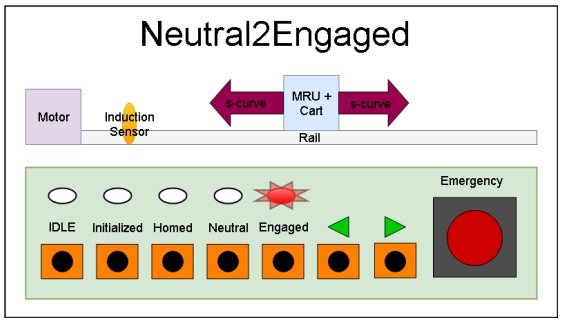
(c) The cart with the MRU is standing still at the induction sensor. The motor is not controlled and the 'Homed' state can be left by receiving the Neutral state as next desired state in the UDP packet from the PC or by pushing the Neutral button.



(d) The cart is moved to the middle of the rail using the control loop. Once the middle of the rail is reached the 'Homed2Neutral' state is successfully completed and the state is updated to 'Neutral'.

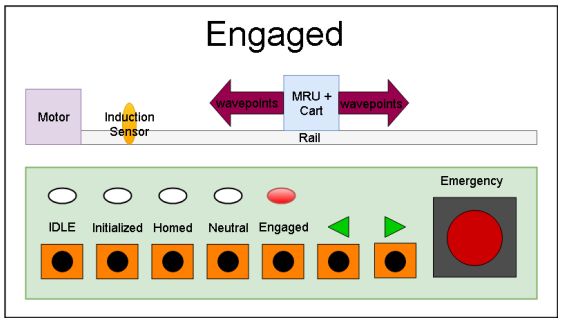


(e) In 'Neutral' the neutral position is actively maintained using the control loop (mainly for vertical). By pushing the Engaged button or sending an UDP packet with the engaged state in the header from the PC the 'Neutral2Engaged' state is entered. One can also choose to return to the 'Homed' state by sending a UDP packet with the Homed state as next desired state or by pushing the Homed button. The system will then go to the 'Neutral2Homed' state.

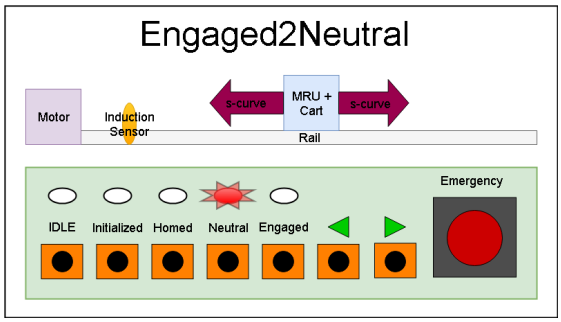


(f) The microcontroller is waiting for the wave points to come in and will follow these using an S-curve. After a fixed interval of 5 [s] the system continues to the 'Engaged' state.

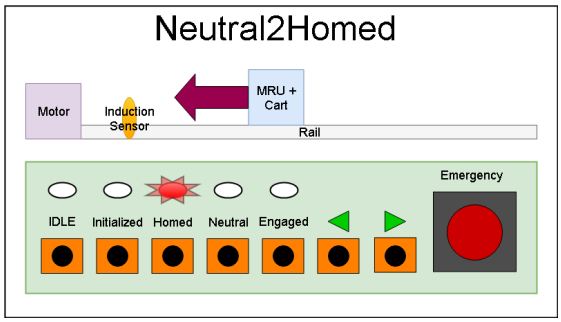
Figure 12



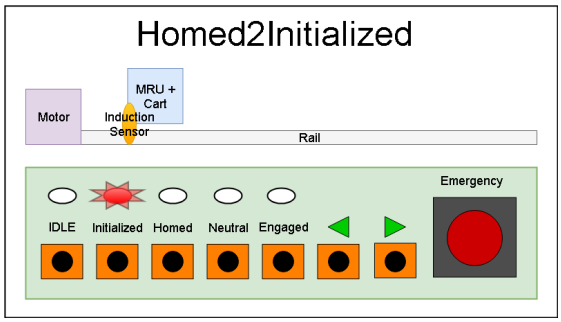
(a) In 'Engaged' the microcontroller will follow the incoming wave points using the control loop. The 'Engaged' state can be left to the 'Engaged2Neutral' state by sending neutral as next desired state from the PC or by pushing the Neutral button.



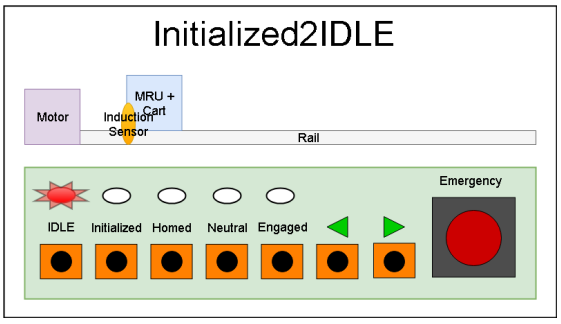
(b) The cart drives to the neutral position using the S-curve in the control loop. When the neutral position is reached, the 'Neutral' state will be entered.



(c) From the neutral position the cart is moved using the control loop towards the induction sensor. Once the sensor is reached, the induction sensor is high and the 'Homed' state will be entered.



(d) Once the sensor is Homed, the system should be able to continue to the 'Initialized' state, once the communication is checked with the PC.



(e) The motor control loop is turned off to go to the 'IDLE' state, the 'Initialized2IDLE' state is mainly to have a symmetric FSM

Figure 13

3.3.2 Control loop

In order to accurately simulate the wave models on the test rail, the motor has to be controlled in a proper way. To accomplish this goal, a low-level control loop has to be designed. The most important requirement of this control loop is to ensure a smooth wave motion. This section elaborates on the design procedure for the control loop.

Figure 14 shows the motor control loop. Note that data frequencies of 50 [Hz] as well as 500 [Hz] are indicated in the figure. The reason for this design choice will become clear in this section.

The figure shows that wave points will be sent with 50 [Hz] from PC to microcontroller. This frequency follows from Ampelmann's experience with motor controlling. The next block is the so-called 'S-curve creator'. During some states, the cart must go from a standstill to actually following the wave form. When the cart starts at the rail centre, this will most likely result in a big necessary acceleration to immediately follow the transmitted wave points. This is not desired, since it causes a big shock which could lead to damage of the system and it certainly leads to undesired MRU output data. Therefore, the S-curve creator's task is to smoothly go from the current position to the wave form. In this way, the cart slowly starts accelerating and within a certain time limit the cart will be exactly following the wave points.

As mentioned before, the PC's transmitting frequency cannot be assumed to be exactly 50 [Hz], while the control loop should get the wave points at exactly that frequency. Fortunately, the microcontroller is much better capable of handling tasks at the right frequency. Therefore, the S-curved wave points are stored in a buffer and used when the microcontroller says its time for the control loop to execute.

The actual controlling part of the loop starts after the wave point buffer. Figure 15 illustrates the controller functionality in a visual way. The wave points are passed on by the buffer at exactly 50 [Hz]. The controller works with the current to-be-reached wave point and requests the next wave point from the buffer. The space interval between these points will be linearised and the linearised curve will be divided into 10 positions (each position called 'Position with small interval' in Figure 14). The linearised part between 2 wave points corresponds to a certain speed. The motor driver has to be controlled by transmitting a speed level, so this linearised speed would ideally result in a linear motion from the current wave point to the next wave point without any deviation. Of course, the motor driver, motor and test rail are not ideal, so the cart will end up in a bit different position than desired. Therefore, the actual control loop is implemented. This loop runs at 500 [Hz] and adjusts the speed according to the difference between the real position (given by the encoder) and the desired linearised position (calculated by the controller). When the non-idealities lead to a slower speed and the cart will not reach the next linearised point within 2 [ms] (corresponding to 500 [Hz]), the feed-forward speed will be increased. In case the cart has already passed the linearised position, the feed-forward speed will be decreased. In this way, the cart is controlled 10 times in order to accurately follow the actual wave points at 50 [Hz].

Note that the control loop output speed is given in meters per second. However, the motor driver only accepts an analogue signal which is created by the DAC with a stepsize of 4096 and a voltage between 0-3.3 [V]. This voltage will be translated to a rotational motor speed. So in order to control the motor, a look-up table has to be created, which translates the desired speed (in $[\mathrm{m\,s^{-1}}]$) to a voltage (in [V]), which is essentially a number between 0-4095 for the DAC input. The look-up table will be set up by an experimental test where a certain analogue signal will be set for the motor driver and the cart speed will be measured.

Finally, note that the wave positions are still given as double precision floating point numbers, while the encoder data is given as incremental pulses resulting in an 'int16' number and has an accuracy of 44 [µm] (requirement 26). Given that the wave forms will be symmetrically divided over the rail, the 'int16' format is easier to use for calculations (the rail centre will always have the value 0). Therefore, the encoder data as well as the wave point data has to be converted to the 'int16' format. The incremental encoder data is translated to an 'int16' number in the 'Incremental-to-position' block. The controller takes care of the wave points conversion. Now, the two positions can be compared with each other to obtain the position difference for the main control purpose: speed control.

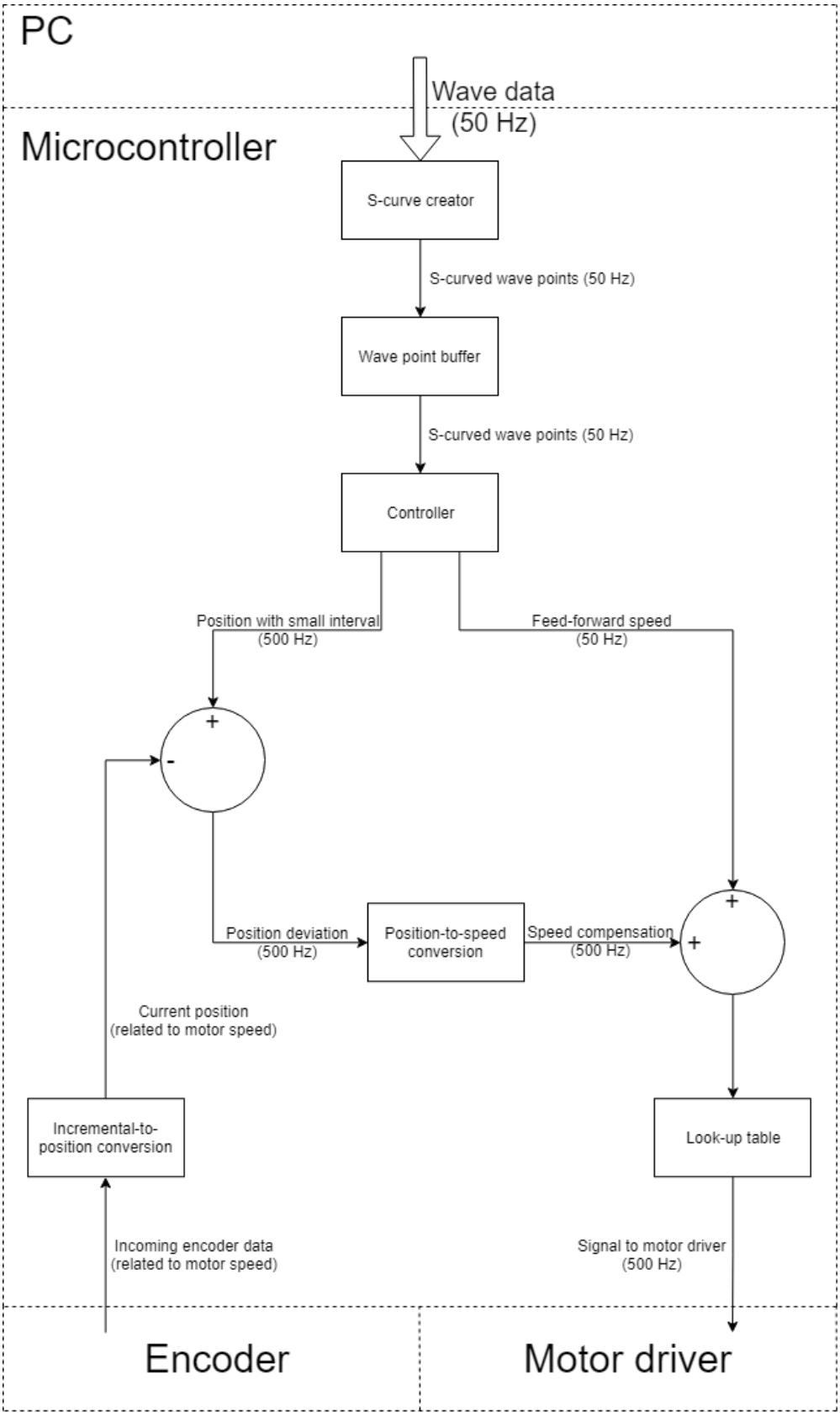


Figure 14: Block diagram motor control loop

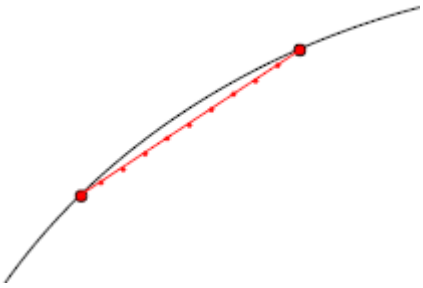


Figure 15: Visualised controller functionality

4 Implementation and validation

This section describes the real implementation and validation of the system software. During this project, an iterative design approach is followed, because many advancing insights occurred already in early project stages. Therefore, some design choices were adapted during the implementation phase. These choices will become clear in this section.

Furthermore, this section has a different layout than the design chapter, because some blocks from Figure 5 have to be implemented before one could test each individual block. For example, the communication between PC and microcontroller can only be tested when the PC and the microcontroller have a data transmitting as well as a data logging part. Therefore, this section follows the chronological order of the system implementation in practice. During the implementation phase it is important to keep the right priorities to work on: first of all, the most critical problems of the system implementation have to be tackled. These problems are fundamental for proper system functioning and they merely have to deal with peripheral communication. These problems are listed below:

- Setting up the communication between PC and microcontroller
- Encoder data readout
- Let the motor rotate by sending speed commands to the motor driver (this part's low-level implementation is implemented by the hardware subgroup, because of the limited project time available and in order to tackle the critical problems as fast as possible)

The following order of implementation is followed during the project: firstly setting up the communication between PC and microcontroller. This part also includes the final design and implementation of the communication protocol and data transmitting and logging blocks. Secondly, ensuring that the encoder can be read out in a good way. This step is very important since the MRU assessment uses the encoder data as the ideal reference signal, which is always considered to be the 'truth'. Thirdly, implementing the FSM in which the encoder can be read out, motor can be controlled and data can be sent to the PC. Before validating all subsystems, a test plan has to be set up. Appendix A describes the test plan and this section describes the exact effectuation of the plan.

Implementation and validation are directly coupled to each other: validation cannot be done without implementation and implementation is adjusted according to the results from validation. This is also a characteristic of following an iterative design process. Therefore, in all sections the whole iterative implementation process will be described, where the adjustments come from previous test results.

4.1 PC-microcontroller communication

This section describes the communication between PC and microcontroller. As mentioned above, this is one of the most critical problems in the software design. Without this functionality, the system will not be able to function. However, this part is also important, because the microcontroller only contains a LED for real-time debugging possibilities. As soon as the communication with the PC is constructed, the microcontroller could send real-time data to the PC where it can be displayed. This makes debugging a lot easier.

This section is split up into different communication aspects. Firstly, the UDP implementation is described. Thereafter, the high-level communication protocol implementation process is described (concerning the question: who sends which UDP packet and at what time?). Last but not least, the data transmitting and logging are described in which the performance of real-time processes on the PC is evaluated.

4.1.1 UDP

Maybe the most critical part of the system is the communication with the microcontroller. According to requirement 19, UDP should be the standard data transmitting protocol. Implementing the functionality to create and edit UDP packets in Octave is not a big problem. However, as was already mentioned in Section 3.1, getting code working on a microcontroller is much more difficult. This also holds for the UDP implementation. A limited amount of examples exist for the implementation of UDP on the STM32F407ZGT6 M4 Cortex microcontroller. Fortunately, this gave enough information to develop a properly working program in the end. The steps to be taken are described below.

First of all, some network theory is necessary in order to be able to communicate over UDP: the right IP addresses for the PC as well as the microcontroller, together with the right subnet mask has to be set. For this purpose, the LwIP stack is selected in STM32CubeMX. In order to have a fixed IP address for both devices, Dynamic Host Configuration Protocol (DHCP) is disabled. Figure 16 shows a screenshot of some other settings in STM32CubeMX regarding the Ethernet connection. Among these settings are the internal Media Access Control (MAC) address and PHY (standing for the physical layer of the OSI model) address. Furthermore, the RMII (Reduced media-independent interface) mode is selected in the pinout tab to be able to connect the MAC hardware within the microcontroller with as least pins as possible to the PHY chip, which is integrated on the Olimex E407 development board. Important to note is that FreeRTOS (Free Real-Time Operating System) is used in some of the useful example codes. This could potentially provide a lot of easiness regarding timing of all different tasks to be executed during the real-time system execution.

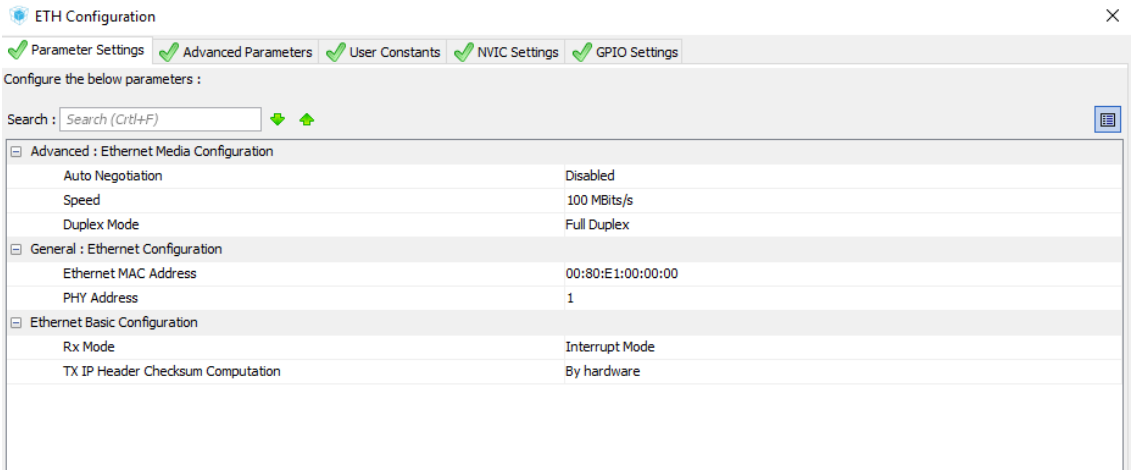


Figure 16: Screenshot of ethernet settings in STM32CubeMX

In the following sections, all different UDP implementations and corresponding validations are described. Different formats have been tested, because of advancing insights during the project, as will become clear in the following sections.

4.1.1.1 Echo server

The easiest way of implementing UDP communication is the echo server. Logically, this is the example to start with, because it quickly shows whether the connection can be created and the data can be received on both sides. As already mentioned, many examples are given in FreeRTOS, so FreeRTOS was also implemented in the code. In this case the microcontroller will exactly transmit back what it has received from the PC. On the microcontroller an interrupt is generated when a packet is received. In this interrupt a callback function will be executed where the data in the packet can be read out. This data is placed in the buffer and subsequently send back to the PC. Echo server functionality can easily be tested with the program SocketTest in which a character or value can be given as input and the received message is immediately displayed when received.

As in many cases, the code did not work at first. This was quite a problem, because nothing was received creating a relatively big black box of functionality where the problem could occur. Therefore, a LED was used in the callback function: every time the callback function was executed (thus a packet was received), the LED was toggled. Furthermore, the board was programmed via a ST-Link, which also functions as a debugging device. Using the ST-Link the real-time process could be debugged by placing a breakpoint in the callback function, for example. Debugging a real-time process could be problematic, because all timing is interrupted. However, when sending the same packets at a certain frequency to see whether the microcontroller is able to receive UDP packets, timing is not an issue. Therefore, debugging is a valid code testing method in this case.

Using the described debugging tools, it could be checked whether the function was called when a packet was received. Furthermore, Wireshark was used to see which communication took place on the Ethernet cable to which the microcontroller was connected.

After debugging to be able to communicate at all, the echo server test showed positive results. However, after sending 16 packets, no data could be transmitted by the microcontroller. This turned out to be caused by a full buffer, so it could easily be solved by deleting the buffer after sending and recreating a buffer before sending a new packet.

4.1.1.2 Multiple data formats

At this project stage the final PC and microcontroller data formats were not known. Therefore, multiple data formats were tested to ensure that the current way of working with UDP could be continued during the rest of the project. Important data formats to be tested were ‘char’, ‘uint8’, ‘uint16’, ‘uint32’, ‘float’ and ‘double’. ‘floats’ turned out to be important, because they are used in the final communication format designs (see Section 3.2.1). ‘char’ is especially useful in order to create readable code. In this way humans can easily understand what is transmitted to/received from the microcontroller. This type is only used during project tests, as will become clear below. ‘uint16’, ‘uint32’ and ‘double’ were expected to be used, but turned out to be of less importance. Finally, ‘floats’ were chosen to be used, because they are both accurate and standardized for both the PC and microcontroller.

Actually, the only difficulty in communicating via UDP with specific data formats is the interpretation of the packets on the microcontroller. The LwIP stack defines a struct with a pointer to the buffer in which received packets are placed. However, this pointer is by default a void pointer and could thus point to any data type. Therefore, this pointer has to be typecasted to a pointer pointing to the right data format. This means that the microcontroller should always know what data type it has received. Fortunately, this forms no problem, since the communication format onnly specifies ‘floats’ to be used.

In the testing phase this pointer is typecasted to a pre-defined pointer type, so that it can immediately interpret the data in a good way. Together with ensuring that data is received and interpreted well, also some calculations have been done on the microcontroller, of which the result was sent back to the PC. To test this, one has to pay attention to the fact that this callback function looses its data after execution.

Therefore, the data had to be stored in a global variable. In general, the program is designed to have as least global variables as possible, because these variables can be edited in every function which uses them, possibly causing another function to have an unexpected value to work with.

From the communication tests has been concluded that all mentioned data formats can be sent, received and interpreted in the right way on the PC side as well as on the microcontroller side.

4.1.2 Communication protocol

Correctly receiving and interpreting data on the microcontroller does not automatically work. Therefore, a communication protocol had to be set up for all described tests above. By using fixed data packages the PC can easily send data at a certain frequency and the microcontroller handles it each time a packet comes in. Sending at a certain frequency is at this point implemented by just inserting a delay between two sending commands. This does not result in a very accurate frequency, but it does not really matter in this case, because the microcontroller only picks one or a few values to perform calculations with. Timing is not an issue during these tests.

A more advanced communication protocol had to be developed in order to test the UDP communication in a similar way as it will be implemented in the final system. At the moment of UDP communication testing no final system FSM implementation existed. Therefore, a simple FSM was created to simulate the data transfer from PC to microcontroller using a pre-defined communication protocol. In order to be sure that every packet is received and interpreted correctly, an acknowledgement was sent from microcontroller to PC. Table 4.1 shows the communication protocol set up to test the communication.

PC-to-microcontroller message	**Microcontroller-to-PC response**
”Start”	”Start”
Fs	Fs
n	n
amountOfPoints	amountOfPoints
Wave point 1	Wave point 1
Wave point 2	Wave point 2
...	...
Last wave point	Last wave point
”Stop”	”Stop”

Table 4.1: Communication FSM

The messages mean the following:

- ”Start” (type ‘char’): string that is literally sent to the microcontroller
- Fs (type ‘uint16’): sample frequency at which the wave points are obtained from the continuous wave models coming from the wave point generator
- n (type ‘uint16’): amount of repetitions that the same wave model is repeated during simulation
- amountOfPoints (type ‘uint32’): amount of points that will be sent
- Wave point 1-... (type ‘uint16’): wave points to be sent
- ”Stop” (type ‘char’): string that is literally sent to the microcontroller

Note that this FSM assumes all wave points will be stored on the microcontroller, thereby requiring the PC to indicate the amount of wave points that will be sent in order to allocate enough memory. However, after this project phase Ampelmann came with an advancing insight resulting in requirement 17: wave points should be send ‘continuously’ at a fixed frequency to the microcontroller. This caused storage on the microcontroller to be useless. Fortunately, still a lot of written code for this test could be reused in the final UDP implementation.

Also note that this communication protocol is a real-time protocol in essence, because it is executed without any other processes running. The PC sends the first packet to the microcontroller, which will immediately generate the callback function interrupt. Once all queued interrupts are executed (which are not present in case of this test), the callback function will be executed, the packet is interpreted and the right message is set up and sent back. The process takes very little time, limited to the microcontroller clock speed. All data packets can be accurately received within a time frame of 10 [ms] from the moment the packet is sent. 10 [ms] is the timeout value set in Octave before the `udp_read` function will end, so the process could take much less time in the order of 1 [ms]. This speed is satisfying for the test, but in the final system other processes will be running at the same time. Therefore, the timing of UDP communication should also be tested when it is implemented in the final system.

In the final system implementation the PC will always send the next requested state together with an initialization variable (in the 'IDLE2initialized' state) or a wave point (in the 'Neutral', 'Neutral2engaged', 'Engaged' and 'Engaged2neutral' states). Important to note is that the microcontroller always determines whether the next requested state can be reached. This request will be ignored as long as the required transition conditions are not met. In that case the PC will keep sending this requested state.

4.1.3 Data transmitting

As described in Section 3.2.1, the data transmitting part consists of pre-determining the content to be sent to the microcontroller and actually transmitting this information. Reading in the .mat file constructed by the wave point generator is not a problem. This data is adjusted in the data processing part in order to be compatible with the communication formats. These processing lines of code were checked by plotting and printing the data to be sent, but did not give many problems. However, the communication contains some important real-time aspects to test on. As already described, different data formats and protocols has been tested in combination with the microcontroller. In these tests, no other processes were running at the same time. In the final system implementation, however, UDP packets containing wave points have to be send at 50 [Hz]. Therefore, the real-time performance of the PC had to be determined. Figure 17 shows the results of a test performed to assess the ability of the PC to send packet at exactly 50 [Hz]. The figure shows screenshot of the network analyser Wireshark indicating that packets from the PC (with IP address 192.168.0.24) are send at 50 [Hz] (time is given in seconds after starting the packet capturing), thereby satisfying requirement 17. The results were obtained by sticking to the frequency at which the microcontroller (with IP address 192.168.0.10) sends an update, which is also 50 [Hz]. Once a packet was received, a new packet was immediately transmitted. This is an easy way of ensuring a stable and robust communication protocol. From this can be concluded that the PC is able to deliver its packets real-time to the microcontroller.

Time	Source	Destination
0.604534	192.168.0.10	192.168.0.24
0.605327	192.168.0.24	192.168.0.10
0.620198	192.168.0.23	192.168.0.24
0.624442	192.168.0.10	192.168.0.24
0.625228	192.168.0.24	192.168.0.10
0.640033	192.168.0.23	192.168.0.24
0.644255	192.168.0.10	192.168.0.24
0.645010	192.168.0.24	192.168.0.10
0.660073	192.168.0.23	192.168.0.24

Figure 17: Wireshark data showing MRU, microcontroller and PC transmissions via UDP

4.1.4 Data logging

Data logging consists of reading out the MRU and microcontroller data during the real-time process, as explained in Section 3.2.2. All data coming from the microcontroller is sent in a packet containing the current system state. This information is used in data logging to determine which next state is desired. When the microcontroller indicates the neutral state, the PC can send the engaged state to be desired, for example. This can be simply implemented by a single `switch` statement evaluating the received state.

The most important part in this block to validate is the accuracy of receiving (thus timestamping) packets in Octave. In order for the MRU assessment group to perform data processing in a reliable way, the MRU and encoder data should be timestamped when received with an accuracy of 1 [ms] (according to requirement 29). Using the time and value information of both signals, the MRU and encoder can be compared with each other for the final MRU assessment. Accurate timestamping is thus of utmost importance.

In order to test the timestamping accuracy, different tests have been performed. First, the microcontroller timestamps its packets using its internal clock. Figure 18 shows the time intervals between two subsequently transmitted packets on the microcontroller side. It can be concluded that the microcontroller is able to transmit packets at exactly 50 [Hz]. The MRU is also assumed to send packets at exactly 50 [Hz].

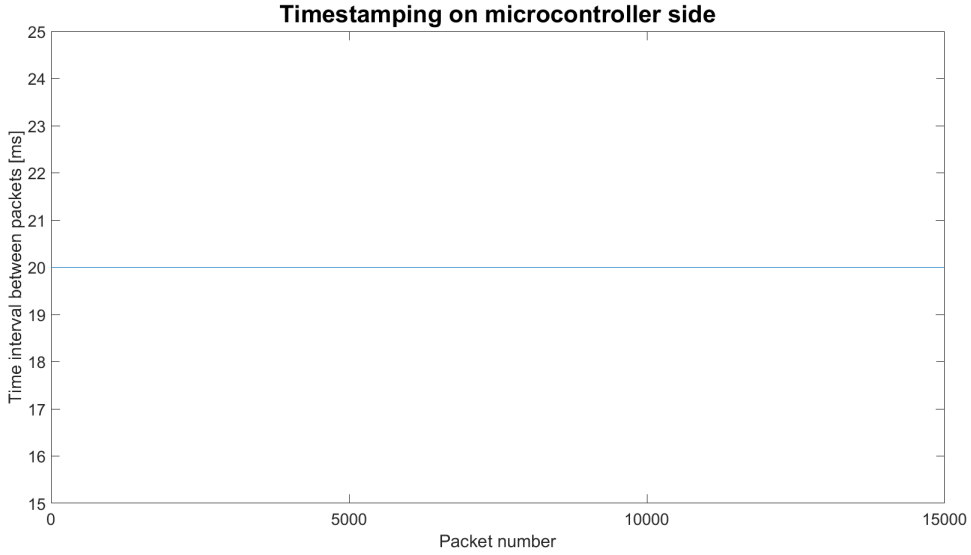


Figure 18: Time intervals between packets transmitted by the microcontroller

Thereafter, Wireshark is used to determine the time intervals at which the packets arrive on the PC side. Figure 19 shows a screenshot of Wireshark from which can be concluded that the microcontroller, as well as the MRU packets are received with an interval time of 20 [ms] (corresponding to 50 [Hz]. This also verifies the ability of the MRU to send packets at 50 [Hz]. The packages coming from 192.168.0.10 are from the MRU and 192.168.0.23 from the microcontroller. The time of arrival is shown in seconds, showing the difference between two incoming packages (with corresponding IP-adress) to be 20 [ms].

Package number	Time (s)	IP-source	IP-destination
237	4.566666	192.168.0.10	192.168.0.24
238	4.583651	192.168.0.23	192.168.0.24
239	4.586467	192.168.0.10	192.168.0.24
240	4.603713	192.168.0.23	192.168.0.24
241	4.606266	192.168.0.10	192.168.0.24
242	4.623648	192.168.0.23	192.168.0.24

Figure 19: Wireshark data showing MRU and microcontroller transmissions via UDP

As a last step, the timestamps given by Octave are plotted, which is shown in Figure 20. From this figure can be concluded that Octave causes the biggest delay in timestamping. From Figure 20 can be concluded that 7 point are lying more then 1 [ms] away from the 20 [ms]. This is 0.023% of the total points measured (30.000). Giving a confidence interval of 99.977%, which satisfies requirement 29.

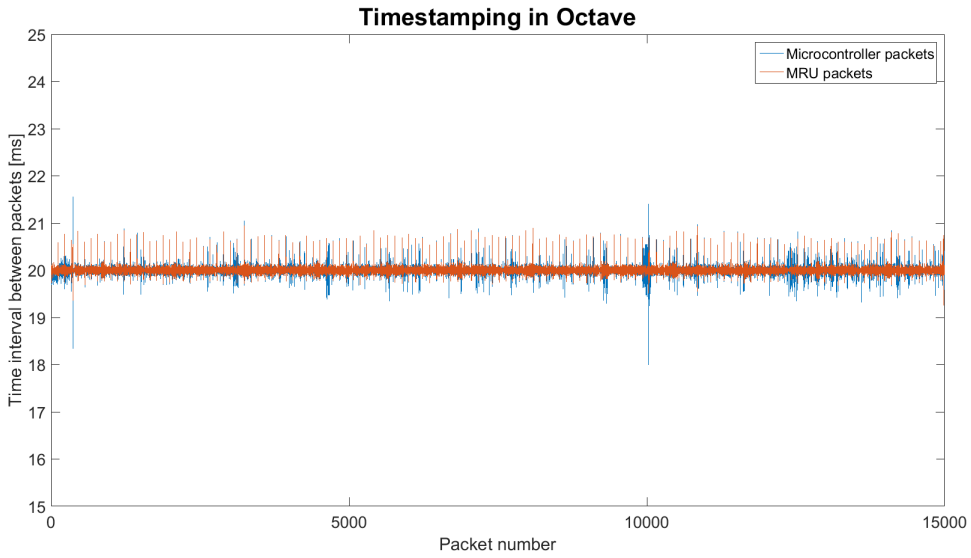


Figure 20: Time intervals between packets timestamped in Octave

Note that above tests do not determine the absolute error caused in each state, but they only show a approximately constant interval between the transmitted packets by the microcontroller and MRU. This is the only thing that should be ensured, because the MRU assessment group compensates for latency errors by performing the cross-correlation on the signals. As long as the encoder and MRU timestamping can be executed with the same performance, the data is valid to use.

From this section can be concluded that the timestamping of encoder as well as MRU data is accurate enough for the MRU assessment group to work with.

4.2 Encoder readout

An important input of the system is the encoder data. In the data processing block MRU data is compared with the encoder data to assess the MRU performance. The encoder data is thus considered as being the 'truth', so it should really be the 'truth' in order not to decrease the MRU assessing capabilities of the system.

The encoder is selected by the hardware subgroup and requires a 5 [V] power supply. This output is provided by the microcontroller board on channel 1 and 2 of timer 1: pin PE9 and PE11 (see Appendix B). These pins were set in encoder mode in STM32CubeMX in order to receive a quadrature incremental pulse signal. To construct this signal two wires are needed which can either have a low voltage or a high voltage. According to the encoder datasheet [35], the output voltage is lower than 0.5 [V] or higher than 2.5 [V]. The signals on the wires are shifted by 90 degrees, hence the quadrature term in the signal name [39]. Which signal is leading with respect to the other indicates the direction in which the encoder is turning (clockwise or counterclockwise). The signal period indicates the speed at which the encoder is rotating, hence the term incremental. Figure 21 shows a quadrature incremental pulse signal.

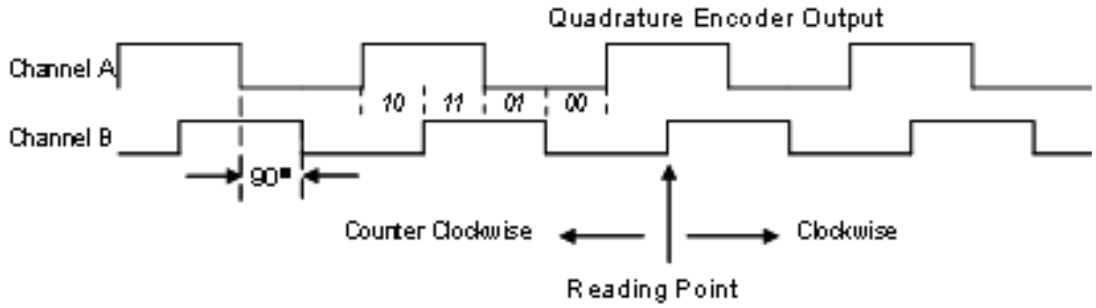


Figure 21: Quadrature incremental encoder signal [1]

Before connecting the encoder to the microcontroller, the electrical specifications had to be checked. The microcontroller can handle 3.3 [V] anyway, but higher voltages may give problems. Therefore, the microcontroller datasheet [37] was consulted to check on 5V tolerance for the corresponding encoder pins. Fortunately, this appeared to be possible in case of the pins used for encoder readout.

Important to note is that the encoder has differential output signals, while in the first encoder tests only one of the two wires per signal was connected to the board. In order to read out the differential signals, a differential amplifier had to be connected by the hardware subgroup. However, this amplifier was not available at the first moment of testing. The first tests were merely meant to ensure that encoder readout was possible in order to remove this uncertainty in the project. Therefore, it was decided not to wait until the differential amplifier was added to the circuitry.

After electrically connecting the encoder to the development board, the encoder readout in software could be tested. This was done by manually turning the wheel to which the encoder is attached. The measurements can be found in Table 4.2. First, the emphasis was on gathering data of a few measurements in the forward direction (defined as going away from the motor side of the rail). Thereafter the other direction had also to be tested. From this measurement was concluded that forward turning results in circa 9,000 pulses and backward turning in around 10,000 pulses, while the encoder datasheet specifies both directions as resulting in 10,000 pulses.

In order to have more accurate measurements, the wheel was turned twice. This measurement was done twice for both the forward and backward direction. These measurements are given in the second half of Table 4.2. These tests also indicated a difference of 1,000 pulses between the two directions.

[1] Source: https://www.eetimes.com/document.asp?doc_id=1274595

# wheel rotations	Direction	# encoder pulses
1	Forward	8,707
1	Forward	8,997
1	Forward	8,921
1	Backward	10,049
2	Forward	17,962
2	Forward	18,060
2	Backward	20,035
2	Backward	20,033

Table 4.2: First manually conducted encoder test results

Given that the encoder gives differential signals as output, the solution could lie in adding pull-down resistors to both encoder input channels. The measurement results with these resistors of a comparable test as the one before are given in Table 4.3.

# wheel rotations	Direction	# encoder pulses
2	Forward	20,029
2	Forward	20,198
2	Forward	20,081
2	Backward	20,049

Table 4.3: Second manually conducted encoder test

From this test can be concluded that the pull-down resistors provided the solution for the difference in encoder pulses depending on the turning direction. Of course, it is a logical solution, because the individual wires could be floating a bit. Therefore, the official output signal is a differential signal. However, the pull-down resistors ensured stable working through setting a constant reference voltage. In order to ensure robust encoder readout, also in case of electrical interference with power cables, the differential amplifier was implemented by the hardware subgroup. This was also tested, giving comparable results to the tests done with inserted pull-down resistors described in Table 4.3.

Except for reading out the encoder data, this data should also be translated to a position on the rail in meters, as described in Section 3.3.2. Therefore, the next step in the encoder readout implementation is to run an encoder calibration test. According to the datasheet, the encoder should give 10,000 pulses per rotation. However, the wheel circumference to which the encoder is connected needs to be known to make the translation from amount of rotations to a 1DoF displacement on the rail. This can be obtained by just measuring the wheel diameter, but this results in a relatively big error. Namely, translation from amount of encoder pulses to position in meters is given by Equation (4.1). As can be seen, the wheel radius is the only variable which causes uncertainty in this formula, assuming that the encoder is perfectly working. An error in the radius translates to a position error which is 2π times bigger. Therefore, the error is reduced by measuring the position distance on the rail and calculating the corresponding radius to work with. Although this might result in a wrong radius, the non-idealities of the system, in the transmission as well as the processing of the encoder signal, are also compensated for. Furthermore, the radius is the only changeable variable in the program that can be adjusted in order to perform the right encoder data conversion. Table 4.4 shows the results of this test, which consists of measuring the distance from the cart's movement on the rail, compared with the encoder pulses.

As can be seen, the wheel radius is circa 6.7 [cm], which also corresponds to the direct wheel circumference measurements. This means that the system from encoder to microcontroller does not have much impact on the electrical signals coming from the encoder. It can be concluded that the encoder readout is successful and can be used in the final system, as described in Section 4.4.

$$1DoFposition = \frac{2 * pi * r_{wheel} * \#pulses}{\#pulses\_per\_rotation} \quad [\text{m}] \quad (4.1)$$

1DoF displacement (in [m])	**# encoder pulses**	**Calculated radius (in [cm])**
1.557	36,981	6.70
1.562	36,976	6.72
1.563	37,269	6.67
1.557	37,037	6.69
1.560	37,085	6.69
1.564	37,179	6.70
1.561	37,199	6.68
1.570	37,423	6.68
1.562	37,200	6.68
1.561	37,138	6.69

Table 4.4: Encoder calibration test results

4.3 Finite State Machine

In order to have a well-defined functioning system the use of an FSM is a powerful tool. Still, FSMs can become tricky when the system gets more complex. For this reason a simple FSM is built, which only contains two states. One state in which a LED is blinking and one in which the LED is off. To ensure that every task will work, which is needed for the final FSM implementation. A new project is created in which tasks are added step by step and validated when added. The first added task is to read out the encoder and send these data to the PC. Next the task for the motor controller is added and finally a task is created in which the FSM will be implemented. In the FSM task a few states are added, in which the motor encoder sends data to the PC and one in which it does not send data to the PC. The same is done for the motor controller. During every step, it is validated whether there is a blinking LED, transmitted UDP packet or a change in the motor driver input. After all the tests are completed the FSM is partially implemented, with only a few states which are all incomplete. However, the FSM is capable of asynchronously receiving wave points in the 'Engaged' state (the goal is to receive 50 packets per second, in this test a random amount of packets per second is sent) from the PC and to use these in the feed forward control loop to drive the cart. After this worked, the FSM without the full 'Failed' state is implemented. With different conditions in which the transition states can switch for easy debugging. As soon as this worked, the conditions in which the transition state is allowed to go to the main state is added. After some debugging the full FSM worked without the fully implemented 'Failed' state. However, the 'Failed' state can be entered and an error message is send to the PC. Only the failure triggers should be implemented, but this will be done when the rest of the system operates normally. Eventually the fully implemented FSM software should look like the Nassi-Schneidermann diagram shown in Figure 22.

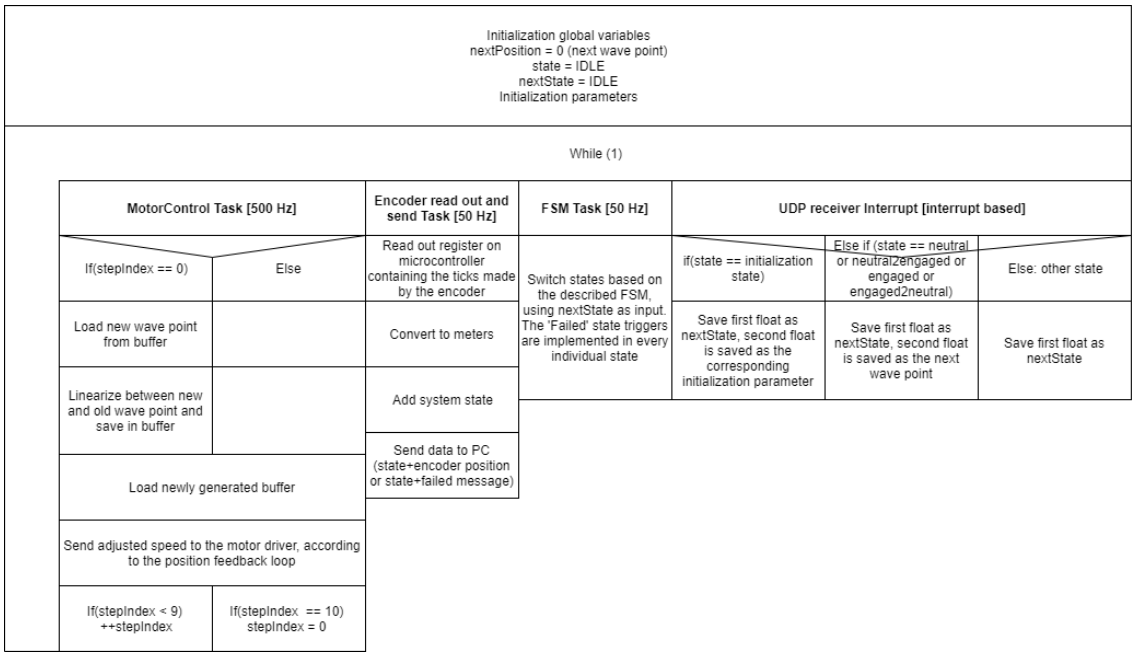


Figure 22: FSM Nassi-Schneiderman diagram

4.4 Final system

Before the system could be implemented as a whole, different subsystems have to be tested, according to the test plan given in Appendix A. These were described above. Unfortunately, at the time of writing, the final system is not finished. However, in order to provide already some relevant data to the MRU assessment subgroup representing the North Sea wave forms, a quick test setup was created.

On the PC a script was written executing the following tasks:

- Reading out, timestamping and interpreting MRU data
- Reading out and timestamping encoder data

These tasks implement the main functionality creating the desired data for the MRU assessment group: timestamped MRU and encoder data.

On the microcontroller side the following tasks had to be executed:

- Controlling the motor driver (feed-forward speed control)
- Reading out encoder data
- Sending encoder data to PC

Instead of implementing the whole FSM with multi-tasking, which should be tested step-by-step according to the test plan, a multi-rate system was implemented. Multi-rate means a single task is running at the highest frequency needed, while other functionality is executed at another frequency within the same task. To ensure that the motor was smoothly controlled (resulting in valid results for the MRU assessment group), the update frequency of the outgoing analog signal to the motor driver was safely set to 1 [kHz]. However, the transmitting frequency of the encoder data to the PC should be in the same order as the MRU update frequency, which is 50 [Hz]. The encoder should be read out just before transmission to the PC, so it should also run at 50 [Hz]. This could be accomplished by updating a counter in the only executed motor control task. The counter was updated every 1 [ms], so the encoder data should be sent every 20 counts within that task.

The system was turned on with above described functionality. Figure 23 shows the plot created after the first test. It can be seen that both MRU and encoder do not resemble a smooth wave form. This turned out to be caused by the relatively slow motor speed in which the motor is not able to deliver enough torque resulting in smooth motions. Implementing the motor control loop should solve this problem, because it compensates for position differences.

Furthermore, the figure shows that the encoder drifts away with respect to the MRU. In this case the MRU shows realistic values. This was checked by the persons running the test: the cart did not drift away by almost 0.5 [m]. The solution to this problem turned out to be the electrical interference of the motor power cables with the small encoder wires. The solution was validated by turning off the motor and manually moving the cart over the rail, thereby taking care of resembling a sine wave as good as possible. In that case the encoder did not drift away anymore, as can be seen in Figure 24. Another solution for this problem was to insert the differential amplifier into the circuitry, as described in Section 4.2. However, this was not the case at the moment when the test results were needed by the MRU assessment group.

Figure 25 shows a plot of the next test's result in which the wave form was again generated by the motor and all system cables were separated and shielded, if necessary. Another problem was still unsolved: the MRU was leading the encoder. This problem could be caused by the time delay occurring when timestamping the MRU and encoder data in Octave. Namely, the MRU data is read and timestamped before the encoder data in this test. Actually, it could also be caused by the bandpass filters in the MRUs, but this is part of MRU assessment group's task.

After simulating wave forms with a frequency of 0.3 [Hz], which represents the final wave forms to be simulated quite well, the frequency was increased until 0.5 [Hz]. This frequency is the highest frequency to be simulated as stated by requirement 27. Figure 26 shows the results of a test ran at 0.5 [Hz]. It clearly shows the cart drifted away on the rail in the positive direction. This problem will be solved by the motor control loop, which is has negative position feedback in order to compensate for drift effects.

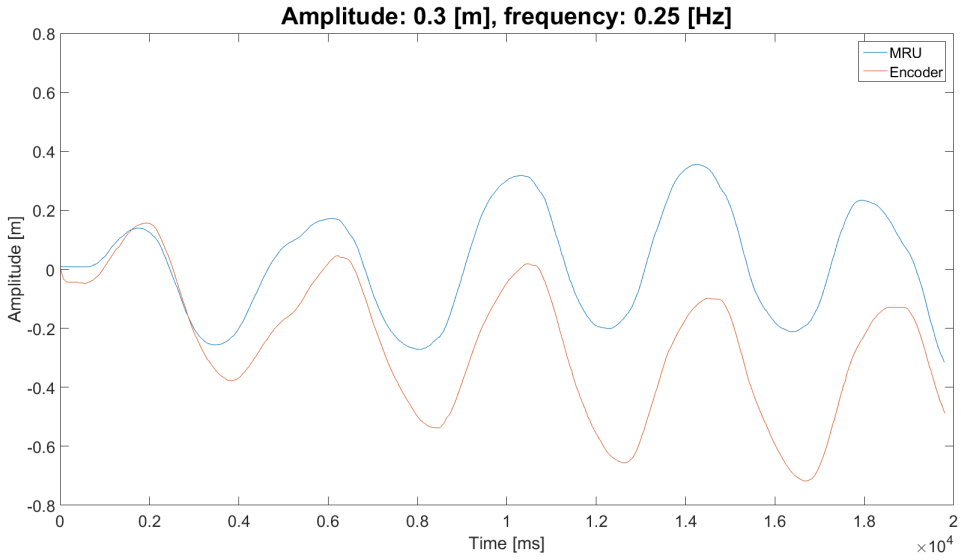


Figure 23: Encoder drift and no fluent wave forms

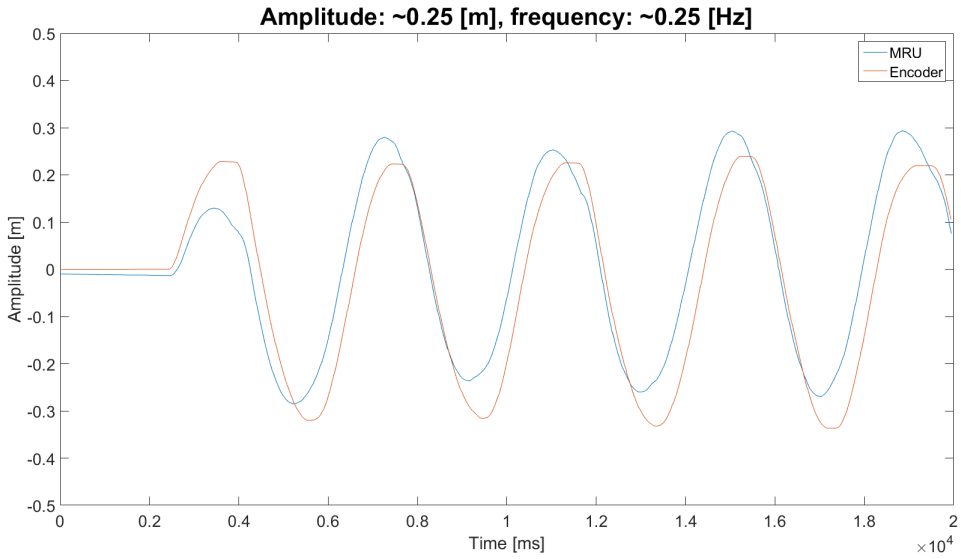


Figure 24: Manual test to check for electrical interference

From these tests can be concluded that the system is able of driving the motor and reading out MRU as well as encoder data. However, the motor control loop certainly needs to be implemented, because it will probably solve the problem of creating smooth motions at low motor speed and the problem of the cart drifting away on the rail. Also the 'Failed' state should be implemented in the final system.

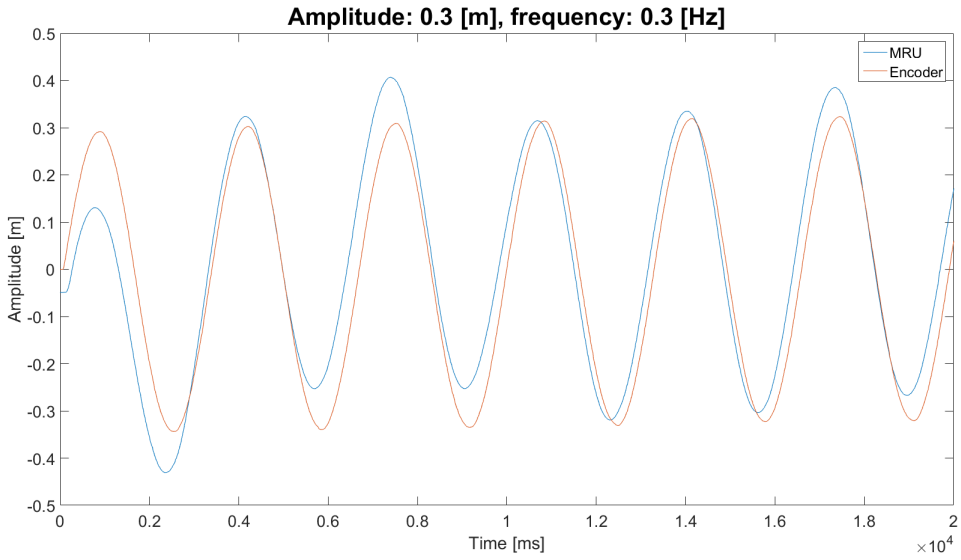


Figure 25: Leading MRU

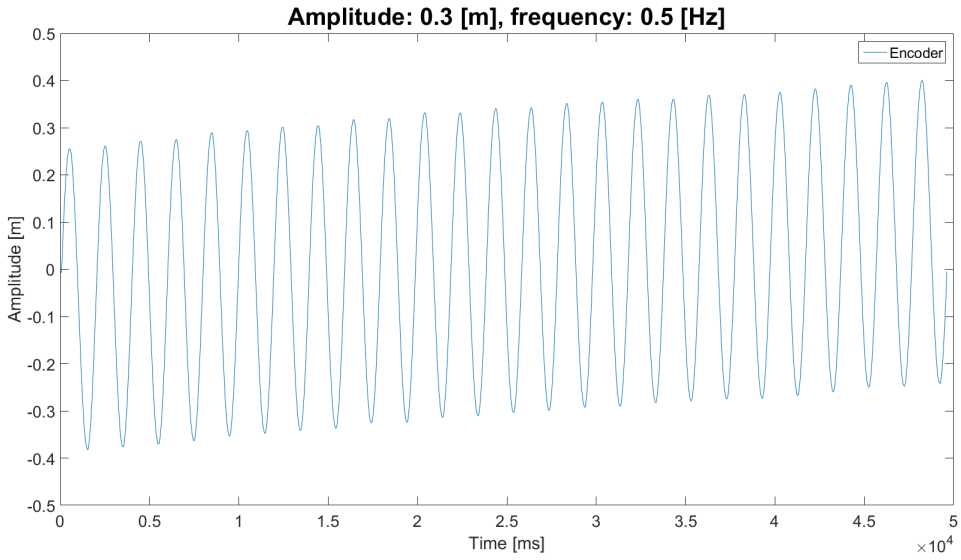


Figure 26: Cart drifting on the rail

5 Conclusion

A lot of work is put into designing and implementing a well functioning system to be able to meet the requirements from Ampelmann. However, not everything is implemented yet. Table 5.1 shows the requirements with a short summary of the implementation in the final system. This will hopefully be finished before the thesis defence. Luckily, the different interfaces connected to the microcontroller are all functioning, including the encoder, motor controller and the UDP communication. Furthermore, the whole FSM (except for some failure triggers) is implemented. Moreover, the script on the PC for the data logging and data transmitting is implemented and working properly. Unfortunately, validated conclusions about the performance of the motor control loop can not be given at this stage. Most safety features in the FSM are working, but not all of them are implemented yet. However, it is possible within the remaining project time to deliver a system in which the 'Failed' state is properly functioning. Not only because safety is an important aspect for Ampelmann, but it is also an important value in society. During testing, the emergency button was sufficient to guarantee safety. However, safety plays a more important role when taking the bigger picture into account. When the delivered test set-up is not working properly, the MRUs are not assessed in a good way. This will finally result in a bad MRU choice for the big offshore operations where human safety is directly involved. Therefore, ethics play an implicit role in this project. In case of the test system itself, ethics does not really play a role, but taking the bigger picture into account, it will certainly play a role.

Finally, taking everything into consideration, the system is not completely finished yet, but based on the implemented parts, it can be concluded that the system will most likely work safely when finished following the design procedure and test plan. The final block diagram of the system can be found in Figure 27.

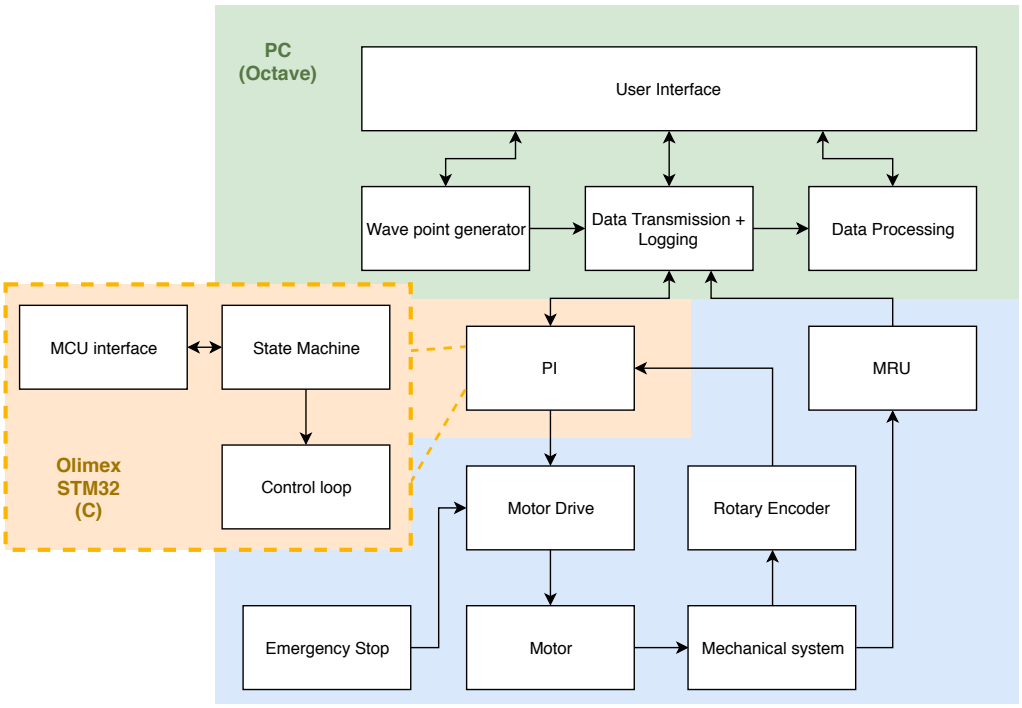


Figure 27: Final block diagram of the MRU test system

Requirement	Implementation
System should be able to test performance of different MRU types for sway, surge and heave motions. This can be done in 3 different simulations	The system can test the sway and surge motion. The heave motion is not implemented
System should be able to run simulations for a relatively long time. 24 hours is considered to be sufficient	This is possible under normal conditions without human interruption
System should be safe to use	The most important 'Failed' states are implemented and tested. Furthermore, an emergency button is present on the system
System should be easy to control	The user only needs to select the desired wave simulation in the UI, after which the full simulation is performed. Also the hardware interface can be used in combination with the PC, this requires some more steps
System should be designed in such a way that it can easily be adjusted for future testing purposes	UDP is used as standard communication protocol. All software is documented in order to add extra features to the system, when desired
System should be able to simulate fluent wave motions, according to the North-Sea wave characteristics: Ts = 3-7 [s] and Hs = 0 - 2.5 [m] [33]. Other kind of wave forms are nice to have. It is desirable to be able to select the previously simulated wave form in order to allow for quickly setting up a new test. Furthermore, the ship Response Amplitude Operator (RAO), used to create the wave form, should represent the ships Ampelmann uses in its offshore operations (resulting from requirement 1)	The desired wave forms of the North-Sea can be simulated (also with an RAO), when Hs is not larger then 1.56 [m], due to the rails' length. Moreover, the UI gives the option to select the previously simulated wave form
In order to assess the heave motion reactions of an MRU, the test rail also has to function when placed vertically (resulting from requirement 1)	This is not implemented yet, but the system is designed with the idea of placing it vertically, so Ampelmann can easily add this feature
System should give the possibility to stop the system on the PC and by using a hardware button (resulting from requirement 3)	An emergency button is present, and the PC can let the system go into the fail stated
System should always be able to safely exit the program flow and go back to a safe state (state in which no mechanical movement is present in the system) in a controlled way (resulting from requirement 3)	The 'Failed' state, which can be entered via the PC or Emergency button, will directly stop the cart from moving
System user should be able to control the system via hardware buttons. In this case the PC is only connected for communicating the wave form to be simulated (resulting from requirement 4)	By using hardware buttons every state can be accessed. When the user is in the 'Neutral' state and wants to go to 'Engaged', the Engaged button can be pressed and the user can start sending the wave points to the system. However, the system expects in the 'IDLE2Initialized' state some initialization variables send from the PC
It is preferable to provide the possibility to read out MRU data using already developed readout scripts in Matlab (resulting from requirement 4)	The system saves the data in the correct format to be able to use it in Matlab

System should be designed in a modular way, this means well-defined interfaces between the different modules (resulting from requirement 5)	This is accomplished by constructing the block diagram shown in Figure 5
Software and needed packages should be free to use	Octave is free to use. The same is true for Eclipse and all software needed to program the microcontroller
Software is preferably already in use at Ampelmann	Octave is already used by Ampelmann
Software should be well documented and built up according to Ampelmann's standards	Documentation is provided to Ampelmann and adjusted based on their feedback. Furthermore, the software is according to their standards based on received feedback
Software should be able to read out different MRU types (with different data formats)	Different MRU types can be read out, as long as it uses UDP
A maximum time error of 1 [ms] in time stamping is allowed with a 99% confidence interval.	The system has a maximum time error of 1 [ms] in a 99.977 % confidence interval
Software should be able to send data real-time and asynchronous to the controller. The idea is to be able to send wave form data at a certain frequency (with possibly introduced delays) and to handle and process this data in a constantly running low-level control loop	By using a buffer, the control loop can synchronise the incoming wave points, which are sent 50 times per second, with a varying time interval
UDP should be the standard communication protocol (between controller, PC and MRU)	This is implemented
Controller should function as a black box. In other words: the behaviour when controlling the system with the PC or with hardware buttons should be known (controller software will most likely not be changed)	This is accomplished by providing the FSM of the microcontroller. Furthermore, the states are based on states used by Ampelmann, making it more intuitive
The system should be able to stop the car when crossing a safety margin to prevent the cart from sliding of the rail under all circumstances	This is implemented in the 'Failed' state

Table 5.1: Table of requirements and design solutions

6 Discussion

This section gives the project evaluation based on three aspect: project process, future work and recommendations. Project process is about the group's way of working and all aspects learned during the project. Future work and recommendations are about the system itself: what should be done to make the system perform better and how could the test set-up be adjusted beyond its current limitations to accomplish a higher goal. These aspects are described in Section 6.1, 6.2 and 6.3, respectively.

6.1 Project process

One of the main difficulties within a project is mostly the teamwork and planning. However, the group for this project was formed a long time before the official project start. All team members knew each other well and a confidential atmosphere existed within the group. Furthermore, the whole group stuck to a structured way of working. This included a well-prepared group meeting every week of which the agenda was shared the day before. Moreover, a meeting with the supervisor from the TU was organised when needed. In that case the supervisor was informed in advance and the meeting was prepared by the group itself, so that the meeting itself was very efficient. Finally, weekly meetings with all supervisors from Ampelmann were held, in which the current project state was discussed. In the beginning of the project everyone had to get used to the project set-up and the new working environment, so the communication within Ampelmann could be improved. This was quickly recognised by the group, resulting in quicker getting feedback regarding the design choices and a faster iterative process.

Planning was a bit more difficult in this project, mainly because of the iterative way of working. Every study project until now was well-organised and everyone could read the manual in which all tasks were explained. Therefore, making a planning was not very difficult. However, during this project only a global project planning could be created in the beginning. As soon as everyone started with designing part of the system it became clear that requirements and interfaces played a very important role. One of the main aspects learned during the project was to get all requirements written down in a very early stage. Design choices can only be made when all requirements are known. This should be done by meeting with the customer very often to discuss the new insights. In this project, Ampelmann played a very good role as being a customer wanting a product to be made, where the engineer should be able to recognise all requirements and interfaces needed in order to develop a good product. In the future, the group will focus more on first writing down all requirements before going on to the design phase. However, in this case it was very difficult for the group to know all details about the system beforehand, because of a lack of knowledge. Therefore, the group gained much experience in handling new scenarios in which design choices have to be made. Furthermore, this project also provided the group with much knowledge about real-time system control.

6.2 Future work

The currently designed software part of the system actually consist of two parallel developed solutions. The first solution being the multi-rate system. This system was built to provide the MRU assessment group with combined MRU and encoder data to perform some data analysis to describe in their thesis. The reason for developing this system was the lack of time in this project. The system as it was designed could not be implemented before the thesis deadline. The second solution, being the final system as it should be, is the multi-task system in which all desired functionality is included. On the PC side, the following functionality should be included:

- Reading out and timestamping MRU data
- Reading out, timestamping and interpreting encoder data
- Depending on the incoming encoder data, sending the right data to the microcontroller

On the microcontroller side, the program should include the following tasks:

- Running a state machine representing the process through which the user should be guided when using the system
- Controlling the motor by sending the right velocity signal to the motor driver
- Reading out the current encoder position
- Sending the current state and encoder position to the PC at a specified frequency
- Send an error signal to the PC in case of a system failure

As mentioned, the final system does not contain all required functionality yet. The following functionality has still to be implemented and tested in the residual project weeks:

- Failure triggers in the FSM
- Motor control loop validation and optimization

The implementation of these software tasks is expected to take less time than it took in earlier project stages, because of gained programming experience.

Besides the remaining software functionality, the software documentation also has to be written. Furthermore, a system user manual will be given to Ampelmann, which describes the normal system flow and what to do in case of emergencies.

6.3 Recommendations

The main goal of this project is to assess different MRU types per wave model representing a sea area where Ampelmann is operating. The wave motions on a ship can be represented by a 6DoF model, including sway, surge, heave, pitch, roll and yaw motions. Only sway, surge and heave motions can be simulated with the current test set-up. Therefore, a huge improvement could be made when the test system is able of simulating motions in all 6DoF.

References

[1] CalQlata, "Vessel 6-degrees of movement." [Online]. Available: http://www.calqlata.com/productpages/00059-help.html

[2] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, ser. The Morgan Kaufmann series in computer architecture and design. Elsevier Inc. UK, 2014.

[3] B. J. Cox, "The objective-c environment: past, present, and future," in *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference*, Feb 1988, pp. 166–169.

[4] T. T. Cheng, E. D. Lock, and N. S. Prywes, "Use of very high level languages and program generation by management professionals," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 552–563, Sept 1984.

[5] H. Krapfenbauer, D. Ertl, H. Kaindl, and J. Falb, "Support for programming embedded software with dynamically typed languages," in *2009 Fourth International Conference on Systems*, March 2009, pp. 163–169.

[6] M. O. Pendergast, "Evaluation of java 1.5 network api for use in peer-to-peer and client-server applications," in *Proceedings 2007 IEEE SoutheastCon*, March 2007, pp. 7–7.

[7] usb4java Team, "usb4java," http://usb4java.org/index.html, 2014, Accessed: 2018-04-28.

[8] Oracle, "Oracle Java Documentation," https://docs.oracle.com/javase, 2009, Accessed: 2018-04-28.

[9] libusb, "libusb," http://libusb.info/, 2014, Accessed: 2018-04-28.

[10] M. J. Donahoo and K. L. Calvert, *TCP/IP SOCKETS IN C: Practical Guide for Programmers*, ser. Morgan Kaufmann. Elsevier USA, 2001.

[11] R. Bayer, *Windows Serial Port Programming*, March 2008, Accessed: 2018-04-28.

[12] C. Gray, "How can I do GUI programming in C?" https://stackoverflow.com/questions/5450047/how-can-i-do-gui-programming-in-c, 2016, Accessed: 2018-04-28.

[13] C. Petzold, *Programming Windows*. Microsoft Press, 1998.

[14] M. Frigo and S. G. Johnson, *FFTW*, October 2017, Accessed: 2018-04-28.

[15] Python Software Foundation, "Python 3.6.5 documentation," https://docs.python.org/3/, 2014, Accessed: 2018-04-28.

[16] The MathWorks, Inc., "MathWorks," https://nl.mathworks.com/help/, 2018, Accessed: 2018-04-28.

[17] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011.

[18] S. S. Maurer, "A survey of embedded systems programming languages," *IEEE Potentials*, vol. 21, no. 2, pp. 30–34, Apr 2002.

[19] M. M. M. Asad, I. A. Marouf, and H. M. Enshasy, "An effective way to program microcontrollers for high speed control operations," in *2017 IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS)*, March 2017, pp. 1–4.

[20] S. Vestel, "University of minnesota, lecture notes: principles of real-time and embedded systems," 2005.

[21] *USB to RS232 Serial Converter Range of Cables*, FTDI Chip, December 2009.

[22] *Optidrive E2 Handleiding*, Invertek Drives.

[23] *M3000 Aluminium Motors*, ABB, January 2002.

[24] *MODBUS over Serial Line: Specification & Implementation Guide*, MODBUS, Inc., December 2006.

[25] *Modbus Protocol: Reference Guide*, MODBUS, Inc., June 1996.

[26] *RS485 Communications Interface: Technical Manual*, Parker SSD Drives, 2007.

[27] K. T. Erickson, "Programmable logic controllers," *IEEE Potentials*, vol. 15, no. 1, pp. 14–17, Feb 1996.

[28] M. Maslar, "Plc standard programming languages: Iec 1131-3," in *Conference Record of 1996 Annual Pulp and Paper Industry Technical Conference*, June 1996, pp. 26–31.

[29] T. J. Ho and C. H. Chang, "Effective speed tracking of induction motors using microcontroller-based intelligent control," in *2017 International Conference on Applied System Innovation (ICASI)*, May 2017, pp. 535–538.

[30] F. A. Samman, T. Waris, T. D. Anugerah, and M. N. Z. Mide, "Three-phase inverter using microcontroller for speed control application on induction motor," in *2014 Makassar International Conference on Electrical Engineering and Informatics (MICEEI)*, Nov 2014, pp. 28–32.

[31] M. G. Ioannides, "Design and implementation of plc-based monitoring control system for induction motor," *IEEE Transactions on Energy Conversion*, vol. 19, no. 3, pp. 469–476, Sept 2004.

[32] H. Wu, Y. Yan, D. Sun, and S. Rene, "A customized real-time compilation for motion control in embedded plcs," *IEEE Transactions on Industrial Informatics*, pp. 1–1, 2018.

[33] D. J. Cerda Salzmann, "Ampelmann development of the access system for offshore wind turbines," Ph.D. dissertation, Delft University of Technology, October 2010.

[34] *IM5097*, Seltec, March 2003.

[35] *Rotary encoder*, LS Mecapion, February 2010.

[36] *STM32F765xx STM32F767xx STM32F768Ax STM32F769xx datasheet*, ST, September 2017.

[37] *STM32F405xx STM32F407xx datasheet*, ST, September 2016.

[38] *Olimex STM32-E407 development board USER'S MANUAL*, Olimex, March 2018.

[39] N. Instruments, "White paper: Quadrature encoder fundamentals," Tech. Rep., March 2016. [Online]. Available: http://www.ni.com/white-paper/4763/en/

Appendices

A Test plan

In every project a test plan is important to keep a structured work flow to prevent unnecessary mistakes. This appendix describes the test plan followed during the implementation phase and the logic behind the design of the test plan. The equipment used for the test plan are:

- A computer
- The software programs: STM32CubeMX and Eclipse IDE
- The Olimex STM32-E407 development board
- A 1DoF-rail
- A motor driver (Optidrive E2) and motor (ABB motors 3GVA072001)
- An incremental encoder (Daido corp. H48 series)
- An induction sensor: IFM electronic IM5097
- An MRU
- 8-hardware buttons and 5 LEDs
- An oscilloscope

Early on in the project, the communication with the test-board did not work. After a couple of days, it was decided to change the board with another type of board. Based on the knowledge gained so far the test plan was completed. The main priority was to be able to have a working UDP connection (Section A.1, A.2 and A.3). Secondly, the encoder should be tested (Section A.4), because this requires an external quadrature read-out extension on the microcontroller, which can lead to complications. Afterwards, the motor controller can be tested (Section A.5), because this is implemented as a Digital-to-Analog Converter (DAC) on the microcontroller by the hardware subgroup. Finally, when the encoder can be read out, the UDP communication and the motor control loop works, the FSM can be implemented (Section A.6). The FSM will be build up by combining the different blocks mentioned above and will be put together piece by piece, after which the states will be implemented. The test plan overview consist of a 'checklist' of items which should be validated. When parts in the system do not function properly certain steps can not be finished. In this case common sense should be used to solve it. However for some steps remarks are given as support. The last added part in the test plan is the back-up plan (Section A.7), which was designed after the green-light assessment. It became clear in this stage that time pressure could start playing a role and for this scenario the test plan should be able to anticipate on the situation.

A.1 UDP connection

1. Set-up an echo server
2. Check for outgoing messages in Wireshark
3. Blink a LED on the microcontroller for every incoming message
4. Check for incoming messages in Wireshark
5. Change data format ('char', 'uint8', 'uint16', 'uint32' and 'double')
6. Perform calculations on the incoming data and send it back
7. Save incoming data in a buffer
8. Build a simple FSM in which every state expects another data format to be received

Step 1 and 3 not working:

- Check IP settings on PC

- Check port number on PC
- Check Ethernet settings on microcontroller (STM32CubeMX)
- Check communication time-out
- Use debugging functionality via ST-link
- Check for interference with tasks
- Check data format

Step 7 not working:

- When a packet is not received after a certain state, try to blink a LED in that state
- Use only two states in the beginning when nothing functions
- Check for interfering tasks (e.g. a task which has no delay, causing it to run constantly leaving no space for interrupts or other tasks)

A.2 UDP transmitting

1. Send data from microcontroller to PC with 50 [Hz]
2. Let the PC send data back after every received packet
3. Use Wireshark to see if after every received packet on the PC a new packet is send to the microcontroller
4. Repeat step 1 till 3, when the system is almost in its final format

Step 2 not working:

- Check for time-out in Octave

Step 4 not working:

- Change the wait time of the `fread` function in octave
- Check incoming and outgoing data in Wireshark
- Check if the packets are sent in the encoder readout task and not the callback function `packetHandler`

A.3 UDP receiving

1. Receive MRU packets
2. Plot MRU data, while also moving the MRU
3. Receive encoder packets
4. Plot encoder messages, while moving the cart
5. Run the (almost) final script in octave in which first the MRU is read out followed by the encoder readout
6. Timestamp the encoder data on the microcontroller before it is send to the PC
7. Read out the timestamps in Wireshark
8. Read out the timestamps in octave
9. Validate the delay for the timestamped signal in octave compared to the timestamped signal given by the microcontroller
10. Repeat step 6 till 8 until requirements are met

Step 1 not working:

- Check MRU settings in the web interface
- Check IP settings on receiving PC

- Check port number
- Check communication time-out

A.4 Encoder

1. Connect the encoder to the correct pins on the board
2. Rotate the encoder and send the encoder data to the PC
3. Check if the encoder data increases when moving forward and decreases when moving backward
4. Calibrate the encoder: move the cart over a distance between 0.5 and 2 meter and calculate the amount of pulses per centimetre (in order to make the translation from amount of pulses to 1DoF position on the rail)

Step 2 not working:

- Blink a LED in the encoder readout function
- Check pin connections using an oscilloscope

Step 3 not working:

- Flip the two encoder wires with the quadrature signal

A.5 Control loop

1. Send a slow speed to the motor drive
2. Send a slow sinusoidal speed profile to the motor drive
3. Measure the amplitude per given speed from the sinusoidal
4. Create a look-up table in which the desired speed can be translated to the speed the motor drive needs to have
5. Implement the control loop with the speed compensation set to 0, making a feed forward control loop
6. Increase the proportional factor in the 'position-to-speed conversion' block (based on estimated value from Simulink model), until the system gives smoother data (check the FFT of the signal for different frequencies) and does not contain drift
7. Add the S-curve generator to the system
8. Test the control loop by going from 'neutral2engaged' to 'engaged' by sending wave points representing the North-Sea wave models

Step 6 not working:

- Check for used units
- Check for conversions made in the software (encoder position to real position)

A.6 FSM

1. Build an FSM which contains 2 states: state 1 with a LED off and state 2 with a LED on. The states can be switched using a button
2. Create a new project and start with only implementing one task in which the encoder is read out and send via UDP to the PC
3. Add the motor control to the project and send an analog signal to the motor drive to move the cart
4. Add two states to the project in which different tasks can be turned on and off, use the buttons to go to the different states

5. Add multiple states in which the motor control and encoder readout can be turned on and off
6. Receive UDP packets from the PC, while blinking a LED on every received packet
7. Send packets containing velocity points to the motor drive at approximately 50 [Hz]
8. Read the outgoing analog signal to see if the speed changes
9. Move the cart using the transmitted packets by the PC
10. Add all main system states step by step and test every transition ($A \rightarrow B$, $B \rightarrow A$) after adding a stage
11. Implement the 'Fail' state and test all possible fail triggers **(as described in the FSM, see Figure.…)**, which should be active in every state

Step 1-5 not working:

- Check for timer problems in STM32CubeMX
- Check if every task has a delay, to prevent one task from continuously running
- Comment everything in the new added task, except the time interval in which it should run and let a LED-blink in the task

A.7 Back-up plan

In case of time pressure, certain parts can be done after the thesis is handed in, while still being able to validate the most important parts of the system:

1. The s-curve, feedback (based on position) and the look-up table in the control loop can be done after the thesis is handed in, leading to a system with a feed-forward control loop. Because testing the control loop requires an almost functioning system this part is most likely not finished in a short time period
2. Communication protocol optimisation can be done later, as long as the communication works, the probability of getting a fast enough communication is estimated high enough
3. The FSM can be implemented partially with the fail states and some of the main states being implemented afterwards. As long as the signals needed in the fail and main states can be received, the probability of getting the entire FSM working is estimated high enough

B Pinout development board

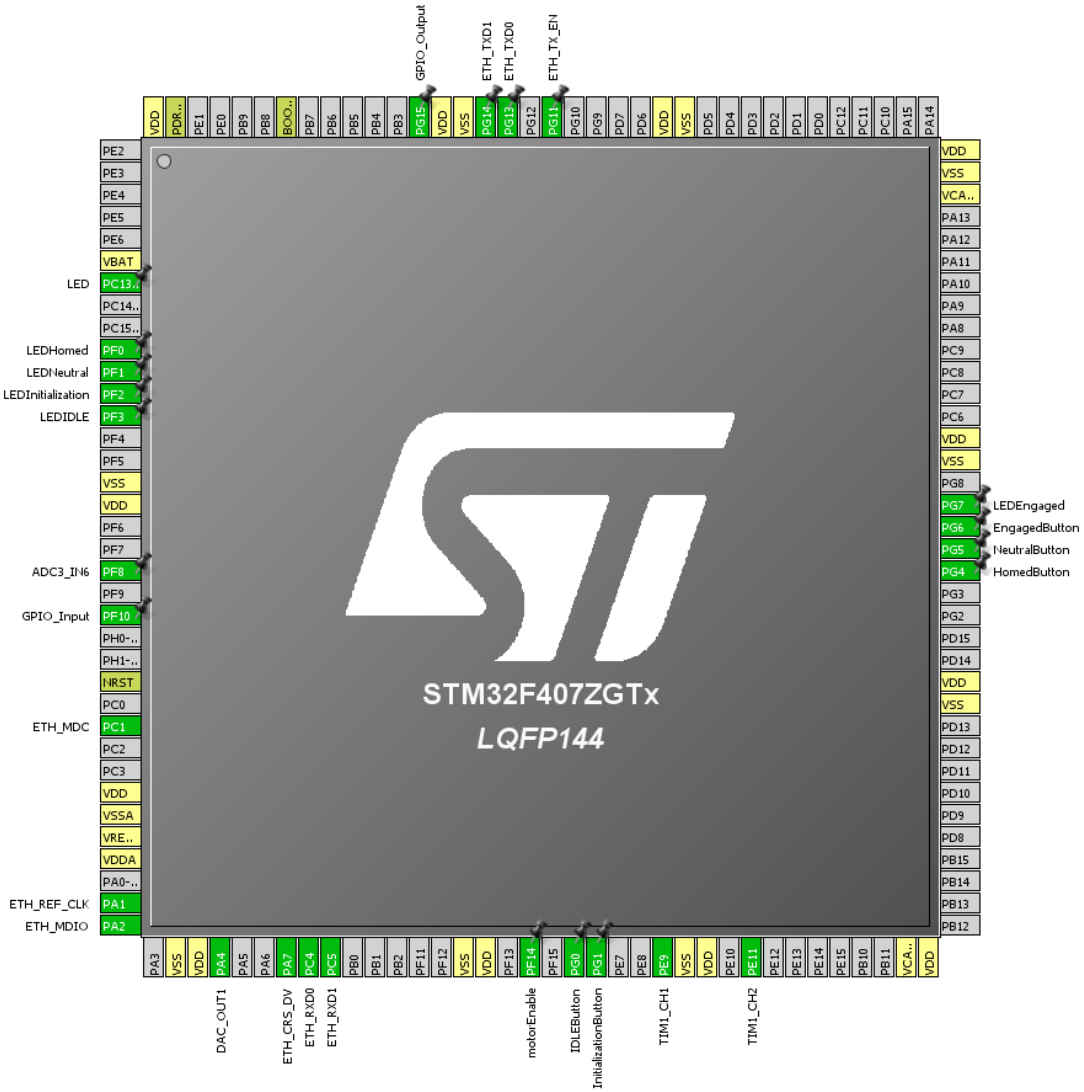


Figure 28: Pinout in STM32CubeMX

C Failed state triggers

IDLE2Initialized

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	X, not detectable no moving cart in this state
Band falls of the rail	X, not detectable no moving cart in this state
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices).	Go to Failed, when no Echo signal is received from the PC
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	X, not detectable no movement

Initialized

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	X, not detectable no moving cart in this state
Band falls of the rail	X, not detectable no moving cart in this state
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no Echo signal is received from the PC
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	X, not detectable no movement

Initialization2Homed

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	Check for changes in the encoder register. This should change at least in 0.5 [s]
Band falls of the rail	Triggered when gap between encoder position and setpoint is above 10 [cm]
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no Echo signal is received from the PC
Broken induction sensor	Induction pin low or no signal change detected after 20 [s]
Wires crossed connected to encoder	Encoder change is opposite to given speedt

Homed

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	X, not detectable no moving cart in this state
Band falls of the rail	X, not detectable no moving cart in this state
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no Echo signal is received from the PC
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	X, not detectable no movement

Homed2Neutral

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	Check for changes in the encoder register. This should change at least in 0.5 [s]
Band falls of the rail	Triggered when gap between encoder position and setpoint is above 10 [cm]
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no Echo signal is received from the PC.
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	Encoder change is opposite to given speed

Neutral

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	Encoder gives 0 on its register
Band falls of the rail	Triggered when gap between encoder position and setpoint is above 10 [cm]
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no wave points are received within three times sample frequency
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	X, not detectable no moving cart in this state

Neutral2Engaged

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	Encoder gives 0 on its register
Band falls of the rail	Triggered when gap between encoder position and setpoint is above 10 [cm]
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no Echo signal is received from the PC
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	Encoder change is opposite to given speed

Engaged

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	Encoder gives 0
Band falls of the rail	Triggered when gap between encoder position and setpoint is above 10 [cm]
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no wave points are received within three times sample frequency
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	Encoder change is opposite to given speed

Engaged2Neutral

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	Encoder gives 0
Band falls of the rail	Triggered when gap between encoder position and setpoint is above 10 [cm]
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no Echo signal is received from the PC.
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	Encoder change is opposite to given speed

Neutral2Homed

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	X, not detectable no moving cart in this state
Band falls of the rail	X, not detectable no moving cart in this statet
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no Echo signal is received from the PC.
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	X, not detectable no movement

Homed2Initialization

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	X, not detectable no moving cart in this state
Band falls of the rail	X, not detectable no moving cart in this state
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no Echo signal is received from the PC.
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	X, not detectable no movement

Initialization2IDLE

Failure	Relevance and Solution
Encoder fails (lose wire or no power)	X, not detectable no moving cart in this state
Band falls of the rail	X, not detectable no moving cart in this state
Motor driver detects a failure.	The motor driver fail state pin is low.
Lost communication with PC (maybe later other devices)	Go to Failed, when no Echo signal is received from the PC.
Broken induction sensor	Induction pin low
Wires crossed connected to encoder	X, not detectable no movement