CAMERA CRANE

- Final Report -

MF2059 Mechatronics Advanced Course

Little Bergmans

Christoffer Bator, Karl Enoksson, Eduardo Fuentevilla, Katherine Gonzalez,
Anthony Loccisano, Pradhyumn Khaitan, Kristoffer Sliwinski, Kaige Tan

December 16, 2018

Flying Bergman AB

Kungliga Tekniska Högskolan
Engineering Design - Mechatronics

Acknowledgements

This project was heavily supported by the stakeholders, Gunnar and Björn. Their initiative, involvement and enthusiasm were quite refreshing and made the project more fun. Bengt was also a huge help and provided guidance during some of the murkier times. As were some professors like Lei, Gustav and Micke. We also appreciated Björn Möller and others in the department for helping us with space and taking attention to all of our requirements. The feedback from all those present in the design review helped us focus on the important aspects of the project and presentation. And finally we'd like to thank our classmates and friends that shared our joys and struggles throughout the semester. Much love.

Abstract

A camera crane is a mechanical arm mounted on top of a vehicle that supports and moves a suspended load. This device allows film producers to capture scenes while in motion. The cranes available on the market today are heavy and expensive, requiring specialized vehicles and multiple operators. There are efforts to lower the weight and thereby make the system available to a wider range of film makers. However, the price is still a hurdle. This report covers the development of a lighter, more accessible camera crane prototype for the small budget producer. Traditional gyro-stabilization technology, a major source of weight and expense, is replaced with a combination of passive damping and active controls. The two degrees of freedom, luffing for tilt and slewing for swing, are actuated with a linear actuator and right angle gear motor, respectively. Both bought and in-house manufactured electrical hardware was combined with a Raspberry Pi micro-processor, Arduino micro-controller and Cypress micro-controller to manage the system controls. The operator interacts with a custom built control panel to enable specific functionalities. The components were successfully assembled and integrated with the electrical hardware and controllers. While all the functionalities desired were not reached robustly, they are still present. The final prototype serves as a good platform for demonstrations and testing.

Contents

List of Figures

List of Tables

Nomenclature & Definitions

Nomenclature

Table 0.1: Abbreviations and Meaning

Abbreviation	Meaning
ABS	Acrylonitrile Butadiene Styrene
AC	Alternating Current
ADC	Analog to Digital Converter
BLDC	Brushless Direct Current
CAD	Computer Aided Design
CAN	Controller Area Network
COG	Center Of Gravity
DC	Direct Current
DOF	Degree(s) Of Freedom
EMI	Electromagnetic Interference
EEPROM	Electronically Erasable Programmable Read-Only Memory
FETS	Field Effect Transistors
GPIO	General Purpose Input Output
HIL	Hardware-in-the-loop
HW	Hardware
IC	Integrated Circuit
ISR	Interrupt Service Routine
KBS	Kilobytes per second
KTH	Kungliga Tekniska Högskolan
LED	Light Emitting Diode
LP	Low-Pass
MAF	Moving Average Filter
MCU	Microcontroller Unit
NVM	Non-Volatile Memory
PCB	Printed Circuit Board
PLA	Polyactic Acid
PVC	Polyvinyl chloride
PWM	Pulse Width Modulation
RPM	Revolutions Per Minute
SOTA	State Of The Art
SPI	Serial Peripheral Interface
SW	Software
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

Definitions

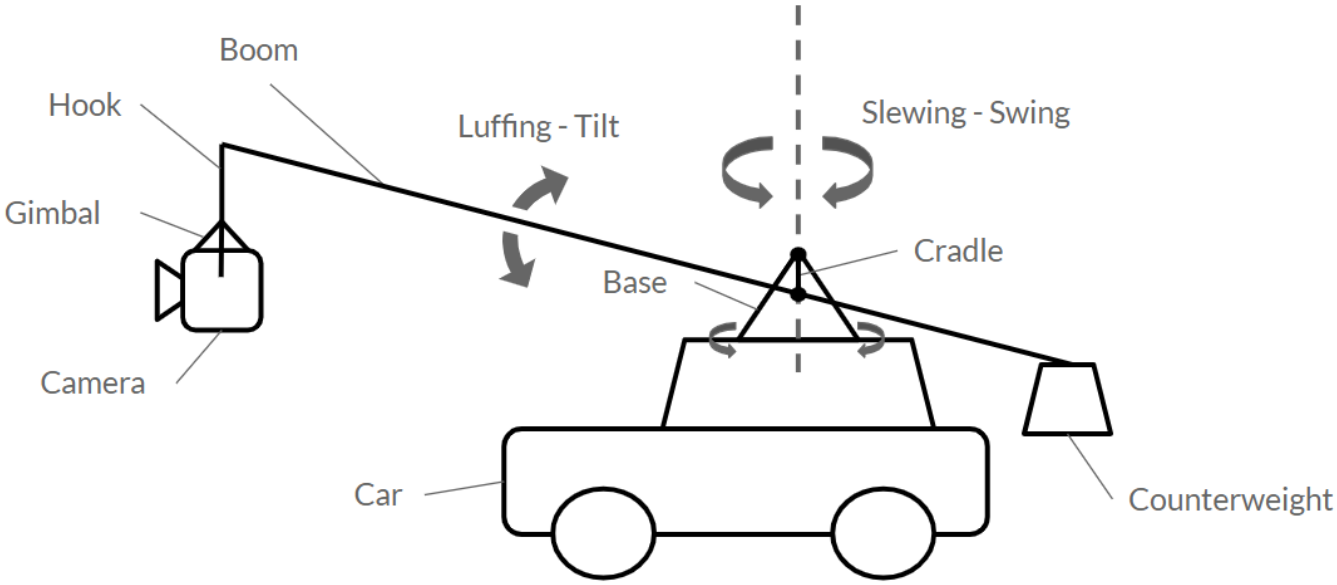


Figure 0.1: Overall System Diagram - Major components and motions of the camera crane system.

Chapter 1

Introduction

1.1 Background & Purpose

In film-making and video production, various mechanical devices are used to capture unique and interesting scenes. One such device is a jib, or camera crane, which consists of a long arm, or boom, balanced on a pivot that can move a suspended load through sweeping arcs. In this case the load is usually a camera-gimbal system, where the gimbal provides extra stabilization. Operation of these devices can range from one to several people, depending on the size of the system and level of control required.

To extend the functionality of camera cranes they have been mounted onto vehicles. This allows them to capture film while moving at higher speeds. While this technology has been around for decades, only recently has demand increased enough to spawn several companies, including TYR, MotoCrane, Scorpio Arm, Russian Arm, and Ultimate Arm.

Among the products available today, all utilize gyro-stabilization technology in the crane's base structure and as a result are relatively heavy and expensive, requiring specialized vehicles, large capital investment and man-power. The stakeholder, Flying Bergmans, has been developing their own camera crane without gyro-stabilization, in order to make it more accessible to the industry. After many iterations, the mechanical design is ready to incorporate more sophisticated electronics and controls. These are intended to allow a single driver or driver-operator pair, to fully control the crane.

The KTH Mechatronics team was tasked with developing this control structure and creating a prototype to demonstrate the desired movements. The purpose of this report is to document the team's findings and development methodology so that the information can be replicated or modified for future products.

1.2 Scope

The team's work will focus on the actuators, electronics, controls and software for a prototype crane. While the prototype may use some components that will be found in the final product, a complete system

ready for market is beyond the scope of this project.

1.3 User Requirements

The goal of the project is therefore to create a functional crane prototype with both manually and automatically controlled motions. Specifically, the stakeholder has asked for the following:

1.3.1 Physical Requirements

- The crane shall be able to slew 360 degrees in 15 seconds (4 rpm).
- The boom shall be able to luff up to a tilt angle of 30 degrees.
- The boom shall be able to reach angles lower than $-30°$ when Underground Mode is enabled.
- The boom shall be able to luff and slew at varying speeds.
- The system shall be battery powered.

1.3.2 Functionality Requirements

- Smooth slewing and luffing motions.
- Ability to maintain a stable position while in motion; to be robust against environmental and road disturbances such as wind and bumps.
- Satisfy the following User Stories:
  - As an operator, to be able to remap the direction of the joystick controller. For example, be able to switch a certain joystick motion from tilting the boom down to tilting the boom up.
  - As an operator, to control the velocity of the boom.
  - As an operator, to be able to vary the maximum velocity of the crane via a knob.
  - As an operator, to have a on/off switch for enabling and disabling drivers and controllers.
  - As an operator, to go back to an original position when pressing home button.
  - As an operator, to have an emergency button to cut off power when pressed.
  - As an operator, to have a screen which can visualize current control mode.
  - As an owner, a password to be required for full control mode and enabling an underground mode.
  - As a driver, position buttons that only tilt the crane from slew angles of 0 or 180 degrees.
  - As a renter, to be able to lease the camera crane car in a restricted motion mode.
  - As an owner, to be able to program positions and execute them via buttons.

- As an owner, to be able to record and log the motions the crane has taken.
- As a developer, to have extra buttons and space for future functionality on the controller.

These requirements serve as the original targets but some have been adjusted according to feedback from the stakeholder.

1.4 Purpose of Report

This report covers the development of a camera crane prototype. The prototype was built in modules and then integrated into a system. Each module is therefore covered, as is the system testing and performance.

Chapter 2

Approach

2.1 Organization

The team decided to split the work into different modules, the largest being: Luffing, Slewing, and the Control Panel. Structure, hardware, software, communication, and others also came up. Once each major module was complete, it was integrated into the system and tested. The smaller modules were developed continuously based on the level of the system and what was required at the time. The report covers the development process for each module and then their integration and system results.

2.2 Literature Review and State of the Art

A State Of The Art (SOTA) report was conducted before the project started to explore the design possibilities in manufacturing and controlling a camera crane. The existing camera cranes, as well as other related technologies were analyzed for function and practicality. This helped the team identify the types of components necessary, such as which actuators to use and what functionality could be expected. These decisions are presented in the report as stemming from the SOTA.

Some information had to be researched during the development process and the theory is explained where appropriate. In general it is expected that the reader have a good understanding of basic components and mechatronic technologies.

2.3 General Methodology

Originally, it was planned that the crane would be manufactured over the summer by the stakeholder. While they would take into account recommendations from the SOTA it was understood that the team would work with whatever was ultimately provided. In the end, the crane ended up being built during the first month of the project out of 3D printed plastic and towards the end, certain parts were replaced with aluminum. This led to many design decisions being made based on what was available.

2.4 Sustainability

Throughout the project, care was given to choose technologies and processes that were efficient and waste averse. While the longevity of the components was considered a full life cycle analysis was not performed, as the final deliverable was a prototype. During the manufacture of the final system it is recommended to look deeper into the sustainability of the materials and processes.

Chapter 3

Structure

3.1 Overview & Requirements

This chapter will cover certain aspects of the structure necessary for understanding the subsequent sections. The structure is the base for making the control possible. The requirements associated with the structure are as follows:

- Ability to maintain a stable position while in motion; to be robust against environmental and road disturbances such as wind and bumps.

3.2 Boom

The boom is the main arm of the crane and was originally planned to be an aluminum tube. However, it was prototyped as a plastic tube for convenience and because it was available. Not only is it easily modified but it is also lighter and shorter, making it easier to test indoors. All of the collars were created with the aluminum boom in mind and therefore have a larger diameter. This puts more stress on the pivot points but because the boom is lighter the effects were negligible. Of course, when the aluminum boom is installed, the models and motor controllers will need to be updated to take into account the additional inertia.

3.3 Counterweight

The counterweight used is a dumbbell set of roughly 13.5kg. This nearly offset another dumbbell handle used at the other end of the boom to simulated the camera. The weights were chosen based on availability and how closely the matched the desired ratio of the target system. For the prototype they were attached via wire.

3.4 Cradle & Damping

One of the major design modifications that the stakeholder made in comparison with cranes on the market is the cradle. It acts as a passive mechanism for handling sudden perturbations due to road

conditions. It also serves as an cushion for the edges of the luffing motion, providing a natural soft start and stop.

The cradle translates the pivot point down towards the center of the base structure. This pivot point, now suspended, is allowed to swing like a pendulum. The swinging motion is controlled passively with an adjustable damper. This is in contrast with almost all the cranes on the market today, which utilize gyro-stabilization technology, often based on a fly wheel or accelerometer. These components add a lot of weight and cost to the system.

More information about the specific operation and intricacies of the cradle can be found in the SOTA. Figures 3.1 and 3.2 give a general idea of operation. The damper was too strong, even in its lightest mode because it was specified for the aluminum boom, which is much heavier. Therefore, the cradle mechanism did not play a significant role in the dynamics of the system. When the aluminum boom is installed, the system will have to be remodeled to take into account this extra degree of freedom.

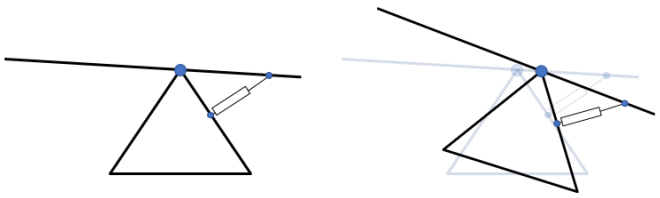


Figure 3.1: Traditional Crane Systems - Normal operation (left) and result of disturbance (right).

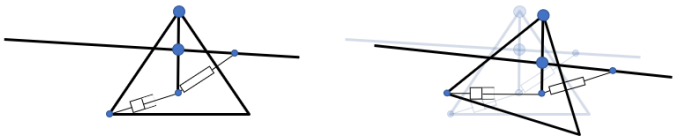


Figure 3.2: The Cradle Design, invented by stakeholder.

3.5 Gimbal

A gimbal is a 3-axis stabilizer and would be used for the camera at the end of the boom. Three motors on the gimbal counter unwanted shaking or vibrations, moving the opposite direction of the disturbances in real time. The gimbal is able to measure the unwanted movements by using an electrical gyro-stabilizer, which is explained in more detail in the SOTA report.

For the prototype, no gimbal was used. It would be too heavy for the system, requiring a very large counterweight. Plus, it is relatively expensive, so at this stage it was deemed an unnecessary risk to include. Updates to the model will be needed once the gimbal is employed.

3.6 Results

The crane was never tested in the real environment: on top of a car. Therefore, its robustness to disturbances cannot be guaranteed. However, the test in the workshop clearly showed a considerable damping of the vibrations produced by the crane motions accelerations. Being a prototype the stakeholder was fine with this. However, the first thing they said they'll do is mount the system on a car to see what happens.

The actual cradle design allow the crane to tilt until 20.4 degrees, failing to achieve the 30 degrees stated on the requirements. However, the successful structure design is helping to smooth both motions of the crane, making way easier to control the whole system.

Chapter 4

Luffing

4.1 Overview & Requirements

The movement of the boom in the vertical plane is referred to as luffing; it is the tilt of the boom. From the SOTA report, it was determined that an electric linear actuator was best suited for this task. The requirements for the actuator were flushed out via dynamic equations of torque and inertia. Ultimately, a different actuator was chosen by the stakeholder than what was recommended. This caused some design alterations that will be further explained in the proceeding sections. The orientation of the actuator had to be set so that the boom would cover its targeted range of motion, 30 degrees, through the actuators stroke length and provide enough force for lifting the expected load. Furthermore, the luffing design must guarantee a smooth motion, so needed for filming.

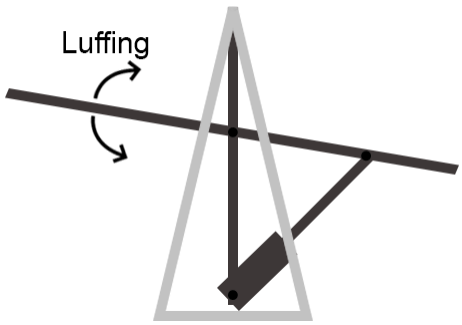


Figure 4.1: Explanation of the luffing motion

For the luffing motion to be considered satisfactory, it must meet the following set of requirements:

1. The boom shall normally operate within a continuous range of $\pm 30°$ from the equilibrium point of $0°$ (horizontal).

2. The boom shall be able to reach angles lower than $-30°$ when Underground Mode is enabled.

3. The boom shall be able to luff at varying speeds.

4.2 Design

4.2.1 Components

An ideal crane design has the crane balanced around the pivot point. This means that in the equilibrium state, the luffing angle is 0 degrees. The balance can be achieved in two ways: using a counterweight or by holding the crane in place. These designs were explored in the SOTA, and were evaluated for the criteria stated in Section 1.3 and for the requirement to reduce the overall weight of the system.

Both approaches would be able to reach the given targets, but the implementation of an actuator without the counterweight proved to be better for reducing the weight. However, due to the already finished crane design and upon stakeholder request, the design using both an actuator and a counterweight was chosen. The actuator type was previously decided in the SOTA. The main design parameters to be selected for the actuator were then the force, stroke length, and speed. Oversizing the actuator and then testing by trial and error could have also worked but was not chosen due to time restrictions and accuracy risk.

Since the boom is balanced on its pivot point with a camera on one and a counterweight on the other, it requires very little force to move. However, because the loads are displaced from the center of rotation, they create a significant amount of inertia and this must be accounted for.

The equations of motion for the crane are then modeled as a rotating rod with displaced masses. The worst case scenario can be identified as the boom starting with the camera in the highest position and luffing down at top speed. Then once the boom is horizontal, a new signal to luff back up is sent out. Changing direction when the boom is horizontal is representative of the largest amount of inertia that must be changed. Since a change in inertia is a torque, this equates to the highest torque as well.

In the end, the linear actuator supplied from the stakeholder was the CAHB-10-B1A-200311-ABBBHD-000, which datasheet can be found in Appendix A.

Table 4.1: Summary of important parameters for the CAHB-10

Variable	**Description**	**Value**	**Unit**
U_l	Rated voltage	24	V
$R_{m,l}$	Rotor resistance	2.6	Ω
$k_{t,l}$	Torque constant	0.32	$kgcm/A$
$V_{max,l}$	Rated speed with load	$5200 \pm 10\%$	RPM
l_l	Actuator length	320	mm
$l_{stroke,l}$	Stroke length	200	mm

4.2.2 Motor Model

To evaluate the behavior of the motor, a Simulink model was constructed using the parameters from Table 4.1. Initially, the model was designed to only look at the parameters of the motor.

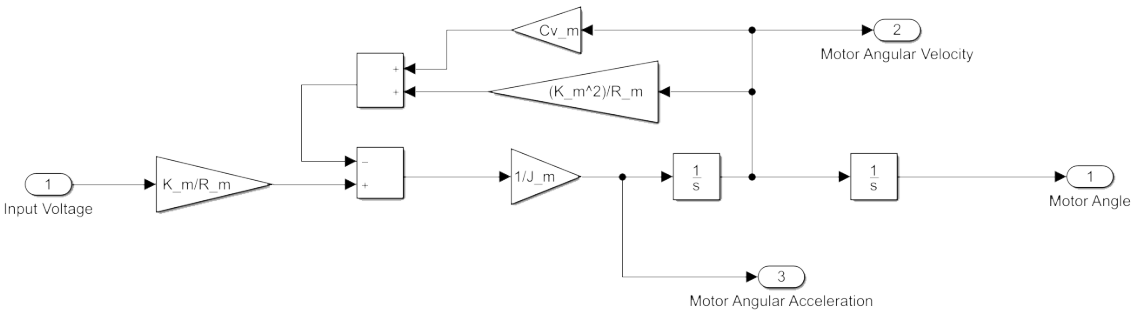


Figure 4.2: Model of the luffing motor in Simulink

Figure 4.2 depicts the standard motor model represented by a block model in Simulink that was obtained from the transfer function of the DC-model according to

$$\begin{aligned} U &= R_m i + k_{emf}\dot{\varphi}_m \\ J_m\ddot{\varphi}_m &= k_m i - c_v\dot{\varphi}_m \end{aligned} \tag{4.0}$$

which gives the transfer function,

$$\ddot{\varphi}_m = \tfrac{1}{J_m}[\tfrac{k_m}{R_m}U - (\tfrac{k_m^2}{R_m} + c_v)\dot{\varphi}_m] \tag{4.0}$$

and the step response,

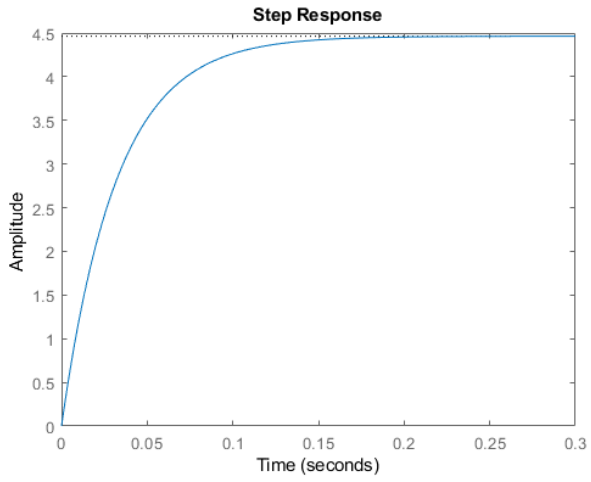


Figure 4.3: Step response for the motor model

With a working model it was possible to tune the motor parameters so that the model would follow how the real actuator was working. Parameter tuning was done using dSPACE (Microautobox II) until a satisfactory response was obtained. This resulted in a change in the parameters listed in Table 4.1, as well as the addition of some parameters. Most notable was the possibility to obtain the gear ratio n,

which was done from comparing the maximum output speed of the motor, as seen in Figure 4.4, to the maximum stroke speed, which was obtained from Appendix A.

Table 4.2: Luffing motor parameters from dSPACE processing

Variable	Description	Value	Unit
$J_{m,l}$	Moment of inertia	$9.0909 * 10^{-6}$	kgm^2
$R_{m,l}$	Rotor resistance	1.4286	Ω
$k_{m,l}$	Torque constant	0.0018	$kgcm/A$
$c_{vm,l}$	Dynamic friction constant	$2.8 * 10^{-4}$	Nm/RPM
n_l	Gear ratio	4.9242	-

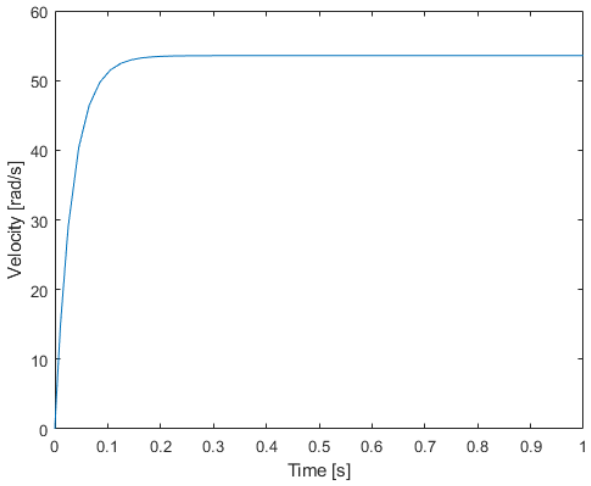


Figure 4.4: The output velocity of the luffing motor

4.2.3 Complete Model

When the model of the motor was working properly, new parameters were taken into consideration to expand it. The new motor model featured the addition of the gear ratio and the inertia from the boom. Initially the inertia of the boom had been estimated through the calculations in Appendix B.1.2, but the parameter was tuned using dSPACE to ensure that the model followed the real actuator.

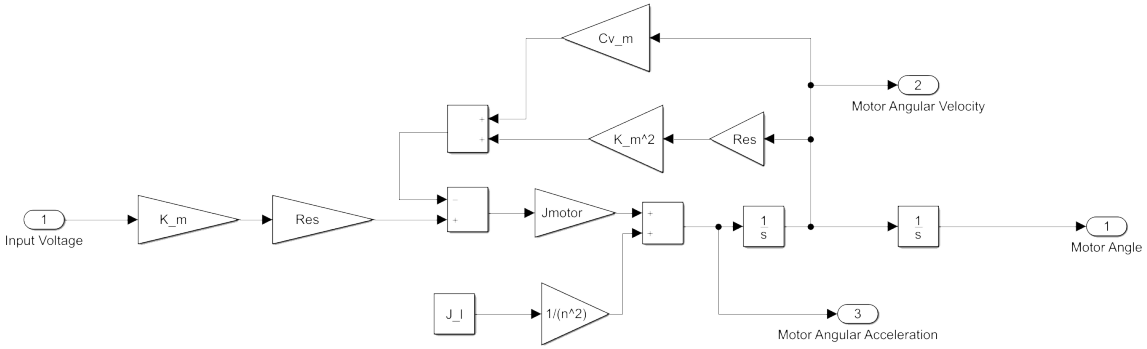


Figure 4.5: Model of the complete luffing motor model in Simulink

Figure 4.6 shows the system response for a step input, showing that the addition of the boom inertia and gear ratio made the system about 17 times slower than when observing just the motor.

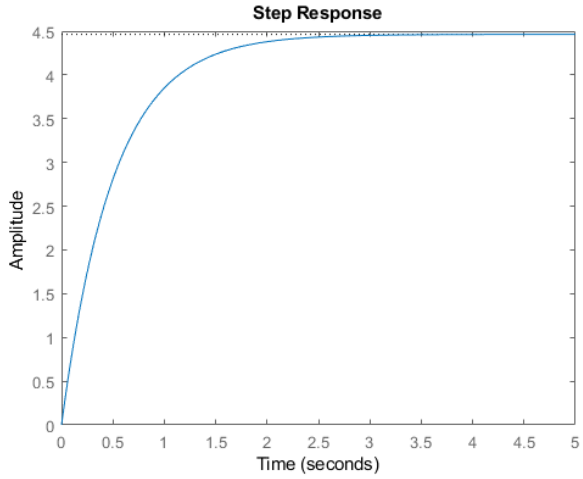


Figure 4.6: Step response for the complete motor model

Having a model that better represented the system as a whole, the two controllers were implemented. For velocity control, a PI-controller with a low-pass filter was used because the change in reference would not be very big, and because it was of importance for the user experience that the output velocity quickly adjusted to the reference velocity for higher comfort. The position controller on the other hand, used a PD-controller with a low-pass filter. Since the difference between actual position and reference position could be quite substantial, using a controller without an I-part removed the problem of having to implement anti-windup. Both controllers were constructed using a feed forward and a feedback part. Figure 4.7 shows the control logic for the position controller. The design is identical to that of the velocity controller, with the only difference being the feedback signal.

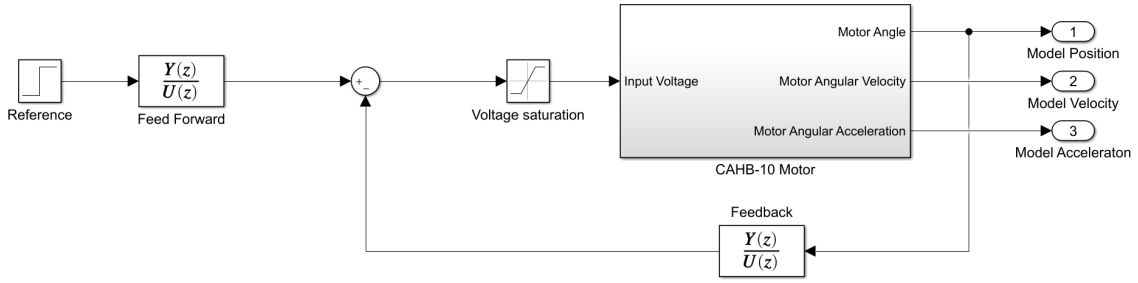
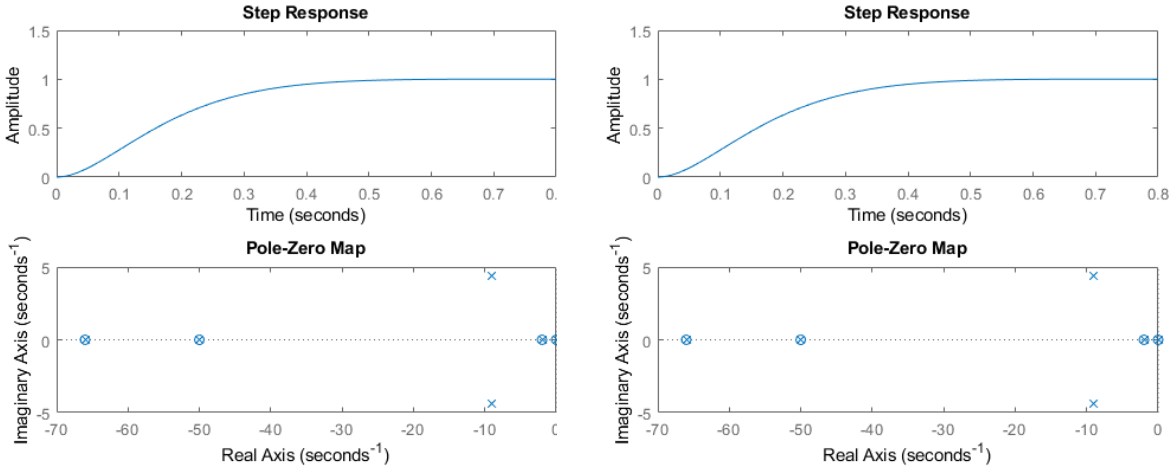


Figure 4.7: Simulink model for the luffing position control

With the control logic in place the poles were placed to generate a soft response for both controllers, as aligned with the user requirement for smooth motions. The step response and the pole-zero map can be seen in Figure 4.8 for the two controllers.



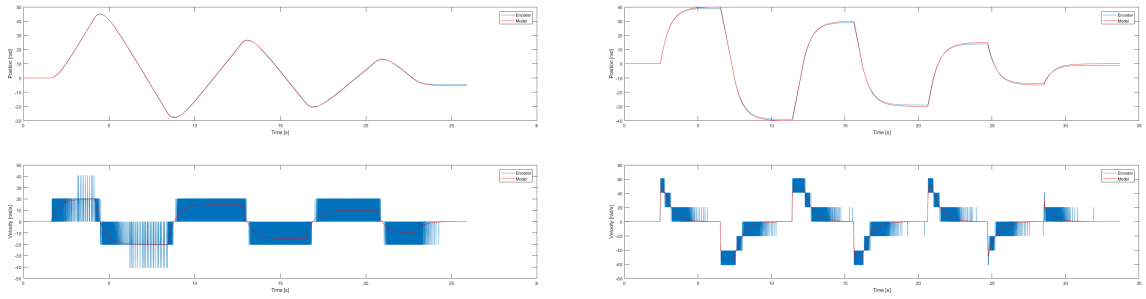
(a) PI w. LP-filter velocity controller

(b) PD w. LP-filter position controller

Figure 4.8: Step response and pole-zero map for the controllers

To finalize the tuning of the controllers, they were imported into dSPACE for performance evaluation. Both controllers were tested on their ability to follow a changing reference value, starting with large differences and ending with smaller and smaller differences.

Figures 4.9a and 4.9b shows the performance of the velocity controller and the position controller respectively. The low resolution of the encoder is what gives the velocity plots its noisy appearance. As the luffing and slewing readings were to be done at the same time, the sampling time was adjusted to fit the slewing controllers. A slower sampling time was used to get rid of the noisy appearance. While successful in that regard, the controller then became too slow and the original sampling time was used instead.



(a) Luffing velocity controller performance

(b) Luffing position controller performance

Figure 4.9: Controller performance for the two luffing controllers

Another solution that was tested to get a better signal for the velocity controller was to introduce a Moving Average Filter (MAF). A MAF takes a set amount of samples over an input time and then generates an output from the mean of the collected samples [20]. While giving a better result, as can be seen in Figure 4.10, there were problems with pairing the filter with the controller. The filter was therefore not used in the final product.

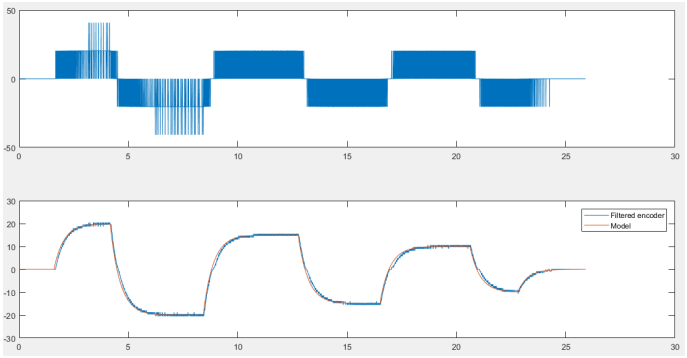
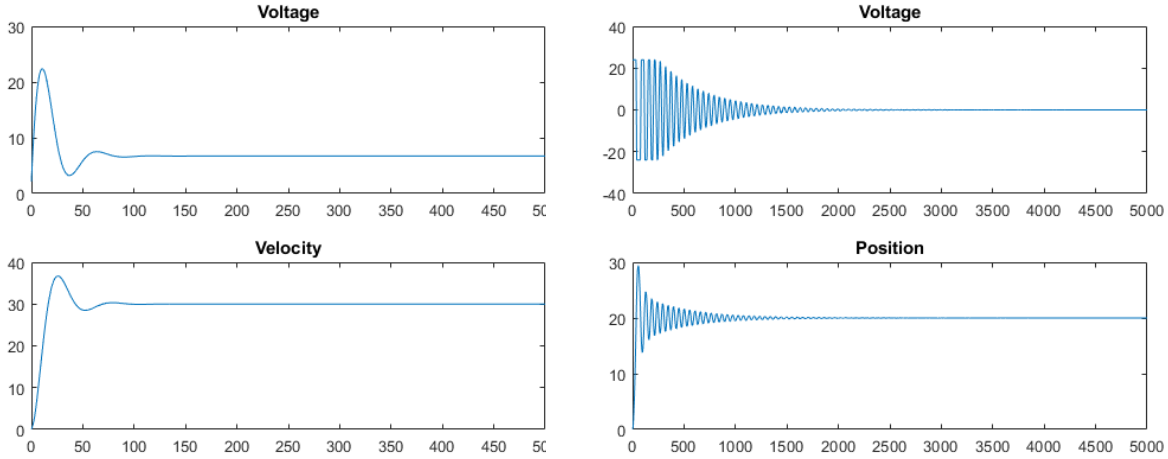


Figure 4.10: Luffing encoder values through a Moving Average Filter

The last step prior to implementing the controller on the microcontroller unit (MCU) was to estimate the amount of iterations required for the controllers to reach the reference values. This test was done on the discrete models of the controllers using a simple MATLAB script. Figures 4.11a and 4.11b show the results, where the values on the x-axis represents the amount of iterations made.



(a) Discrete velocity controller

(b) Discrete position controller

Figure 4.11: Evaluation for the two luffing controllers

4.2.4 Placement

For the actuator to be implemented properly, meaning that it would be able to fulfill all requirements, a mathematical model was constructed to investigate the geometrical constraints for the actuator. The model considered a specific case where the cradle of the crane was considered as fixed and can be seen in Figure 4.12. This more closely resembled the prototype as the damper was too stiff to allow natural cradle swing.

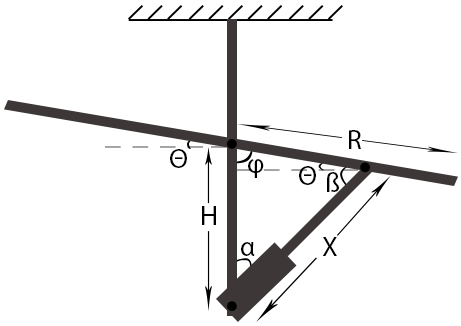


Figure 4.12: Model showing the investigated angles for the luffing case

To be sure that the actuator would meet the set requirements two conditions were introduced: that the actuator length, X, for the angle of +30 degrees would be l_l, and that the angle of -30 degrees would at least be reached for the stroke length of $l_l + l_{stroke}$.

φ was therefore set to a static 60 degrees, and X was set to l_l, to meet the first condition. Figure 4.13 shows how the connection points changes along the length of the boom R and the length of the cradle H as the angles α and β changes in the span $[0°, 120°]$ and $[120°, 0°]$ respectively.

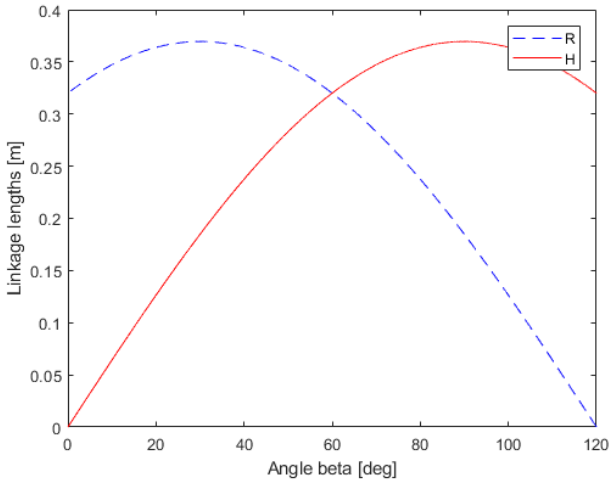


Figure 4.13: Model for how R and H change as a function of β

With the given span of R and H, the variance in θ could be obtained from Equation B.1.1 in Appendix B and is presented in Figure 4.14. By comparing how quickly the angle changed with respect to where on R the actuator was attached, as well as making sure that the actuator could reach the desired angle of -30 degrees, a decision could be made where to put the actuator at the shorter length of R, fulfilling the requirement of the second condition.

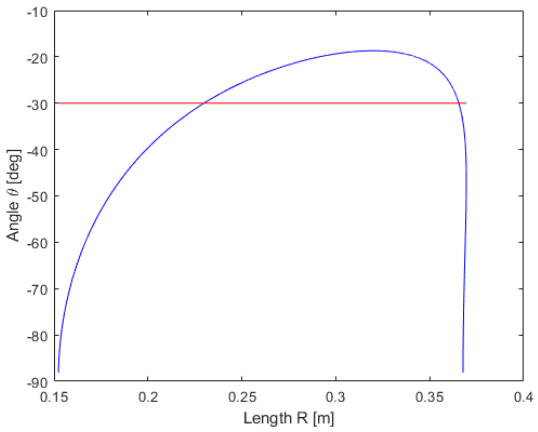


Figure 4.14: Luffing angles θ as a function of R

With R selected, H could be selected from Figure 4.13, and the response for how the luffing angle changed as a function of the actuator length can be found in Figure 4.15.

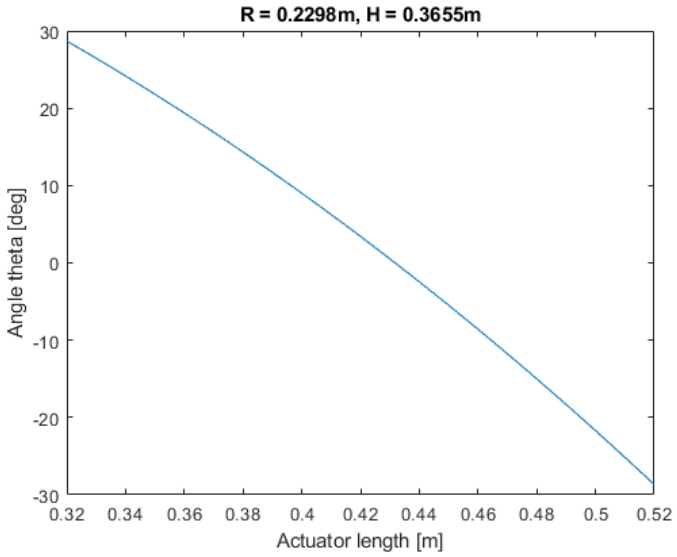


Figure 4.15: The luffing angle span for changed actuator length X

4.3 Implementation

With the actuator in hand and the orientation known, it was relatively straightforward to install the actuator in the proper position. Extra 3D printed parts were made to raise the pivot point and secure attachment points for the damper. This essentially fixed the cradle in place, as the damper was too strong for the plastic boom. In order to prevent the cables connected to the linear actuator from getting tangled during crane rotation, a 12 Wire Compact Capsule Slip Ring For Small Rotating Systems and Electric Motors was connected to the center of the slewing bearing through a rubber retainer that was 3D printed in the Prototype Center.

Figure 4.16: Slip ring connector for the actuator wires

Initially there were some problems on reading the values from the encoder on the actuator. However, after some troubleshooting and help from SKF it was discovered that adding 2 pull-up resistors between the signal pins of the sensor and the voltage input to the encoder, as shown in Figure 4.17, solved the problem.

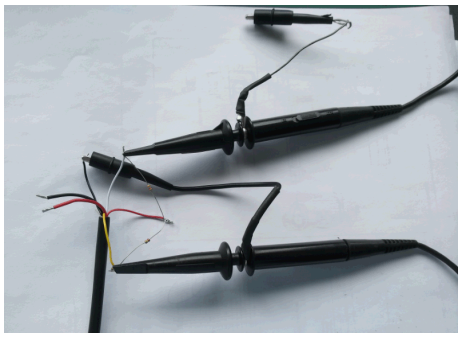


Figure 4.17: Addition of pull up resistors to read the encoder

The reference for the controller was sent by an operator from the developed control panel described in Chapter 6 to an MCU through a Controller Area Network (CAN). The control was done on an MCU developed by Cypress Semiconductor described in Chapter 7.

4.4 Verification and Validation

Verification and validation was performed iteratively as one task was completed so that the next task following the previous one could be started.

4.4.1 Encoder reading

The luffing encoder was tested immediately as the actuator arrived. The initial test was done using an oscilloscope with the motor being fed 24V from an external power source. Figure 4.18 shows the readings from the encoders without the pull up resistors and Figure 4.19 shows the readings from the encoder with the pull up resistors.

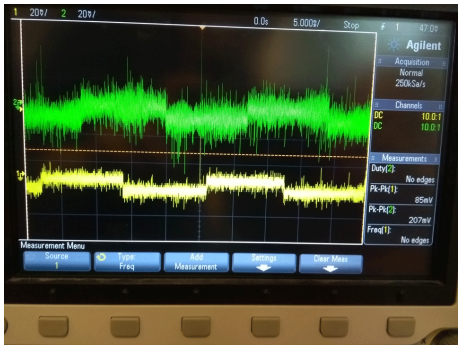


Figure 4.18: The readings from the encoder without the pull up resistors

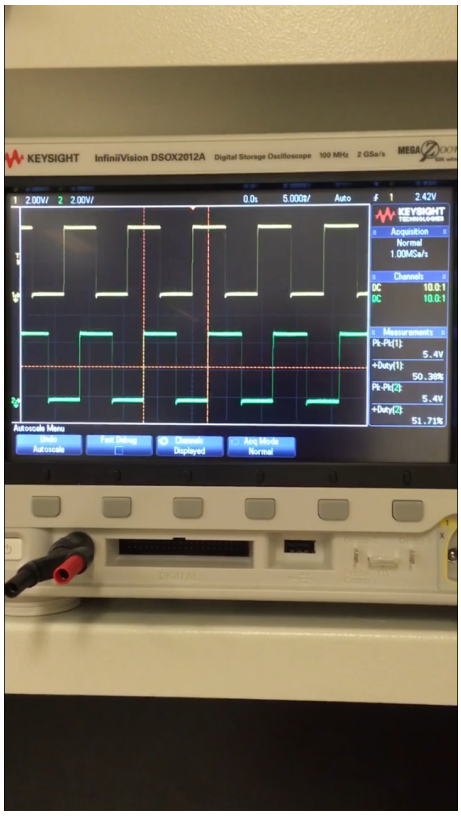


Figure 4.19: The readings from the encoder with the pull up resistors

4.4.2 Actuator Placement

After confirming the encoder was working, the actuator was put in place. Once the actuator was put in place it was tested that it could move from endpoint to endpoint without obstruction from the other mechanical parts.

4.4.3 Model Design

Establishing that the actuator could move unobstructed and that the encoder sent a distinguishable signal, the simulation model of the luffing case was developed. The model was evaluated in dSPACE so that it would perform the same way the real actuator would perform. When the performance was

deemed acceptable, the velocity and position controllers were added to dSPACE to be evaluated. The result of which is presented in Figure 4.9.

4.4.4 Controllability

During testing it became clear that the low resolution of the encoder made it impossible to use the velocity controller developed for the actuator. No sampling time or controller speed could generate a good enough response to still have a robust system. Instead, a ramp was added in the motor driver to achieve a soft start and soft stop that managed to recreate the same feel that was achieved by the controller.

The performance of the position controller did not encounter any problems with regards to controllability. There was some overshoot, however that was due to the flexibility in the PVC pipe.

4.4.5 Fulfilling the requirements

The final check was to test what different angles the crane could reach. When working within the safety limit of having underground mode disabled, the crane reached a top angle of 20.4 degrees and a bottom angle of -20.7 degrees as can be seen in Figures 4.20 and 4.21. With underground mode enabled, the crane managed to reach a bottom angle of -30.9 degrees as can be seen in Figure 4.22

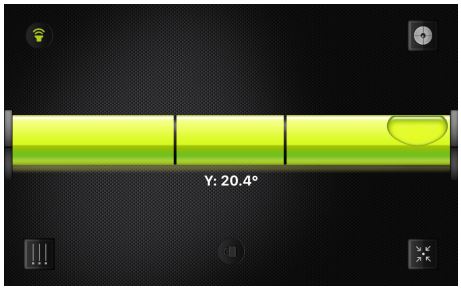


Figure 4.20: Highest angle reached with the crane

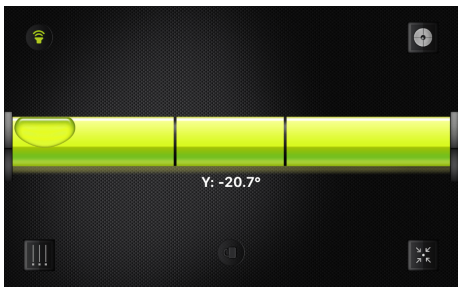


Figure 4.21: Bottom angle reached with the crane without underground mode

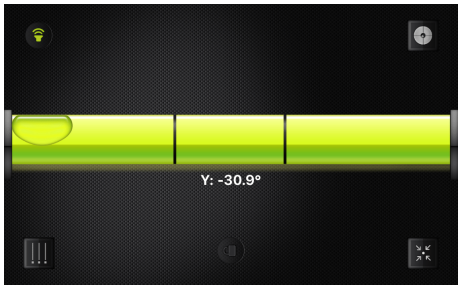


Figure 4.22: Lowest angle reached with the crane with underground mode enabled

4.5 Results

The upper limit of the luffing motion was not achieved. The selection of the actuator together with the mechanical design made impossible to reach the 30 degrees stated in the requirements. However, it was possible to reach the lower angles.

Despite not being able to integrate the velocity controller properly, the control could still be achieved by conditioning the voltage within the software on the MCU. This made it so that the luffing motor presented a fluid motion when it was actuated, fulfilling the necessary control needs.

The weakness of the actuator and the polyvinyl chloride (PVC) boom limited the capabilities of testing the complete crane structure. For a lightweight structure such as the PVC boom the actuator could perform its task very well, but the luffing system was very vulnerable to small changes in the balancing of the system.

Chapter 5

Slewing

5.1 Overview & Requirements

In crane terminology, slewing is the angular movement of a crane boom or crane jib in a horizontal plane. The rotational movement cane be seen in Figure 5.1.

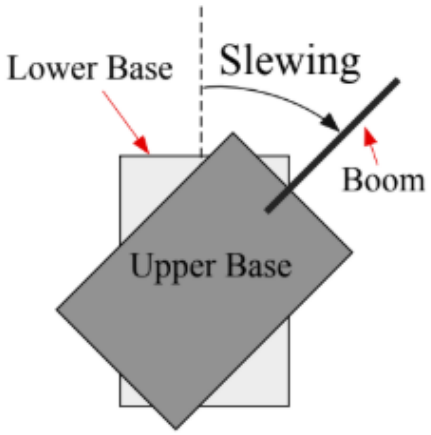


Figure 5.1: Slewing motion seen from Top View [10]

According to the requirements, the crane should be able to rotate clockwise and counterclockwise with a velocity that could be adjustable for each user. Specifically, a speed of at least 4 rpm was requested.

Unlike the linear actuator for luffing, the main challenges associated with the slewing movement lied in the integration of parts. The luffing action only involved one motor and the challenge was how strong, fast, long and where to put it. Slewing is 2 gears, 1 motor, and encoder and their function and integration.

5.2 Design

5.2.1 Components

The slewing mechanism was decided upon in the SOTA. A combination of a slewing ring and a spur gear was used because it is reliable for the range of speeds required, efficient, cheap and easy to manufacture.

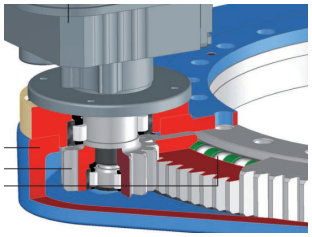


Figure 5.2: Slewing mechanism selected: Slewing Ring and Spur Gear

Source: [1]

For this project, the slewing ring PRT-01-100-TO-ST from the manufacturer Igus was selected. The sliding are made from a plastic called Iglide which is a material that assures to be lubrication- and maintenance-free. All the housing components are made from aluminum and all the surfaces parallel to the sliding elements are hard anodized with stainless-steel fasteners. The dimensions and characteristics are detailed in THE APPENDIX- MOVE THESE THERE Figure A.1 and Table A.1.

The spur gear used was a Misumi GEAHB 2-12-25-8. DETAILS CAN BE FOUND IN THE APPENDIX. The gear ratio between the slewing ring and spur gear 6:1 which lies within the most common used, 1:1 to 6:1 [7]. Their efficiency varies from 94% to 98% with lower gear ratios [8]. The modes were also matched to prevent binding.

In combination with the slewing mechanism, a motor is connected to the spur gear to provide motion to the system. The most common motors used in this particular application are geared motors. The motor provided by the stakeholder to use in the slewing motion was a brushless DC (BLDC) gear motor, model PCC - 24MP3N 100 B3 from Minimotor chosen off torque and speed requirements from the SOTA.

Figure 5.3: Worm Screw Gear Motor: Minimotor PCC - 24MP3N 100 B3 [13]

Table 5.1: Electrical characteristics of the PCC - 24MP3N 100 B3 [13]

Type	**Ratio**	**Voltage [V]**	**Current [A]**	**Absorbed Power [W]**	**Delivered Power [W]**	**Input [rpm]**	**Output [rpm]**	**Rated Torque [Nm]**
PCC 24MP3N	100	24	4.8	115	74.5	2800	28	13

In order to be able to acquire the position and calculate the speed of the motor, the team requested that an encoder be included inside the motor package. An incremental encoder came as a package when the motor was acquired, and this had implications for control implementation in the software, principally regarding position.

The incremental encoder can only register displacement from a home position that is established at startup. The device processing the encoder signal will track the number of full turns of the encoder disc from this home position. If the device is powered down, the number of full rotations counted will be lost and the device must be re-homed upon startup [16]. The encoder provided was the RC type from the manufacturer Minimotor, the details for which are found below.

Table 5.2: Electrical characteristics of the RC Optic Encoder [13]

Type Encoder	**Channels number**	**Resolution [pulses/turn]**	**Power Supply Voltage [V]**	**Output type**	**Max. Frequency Response [kHz]**	**Idling power drawtype [mA]**	**Nominal output current [mA]**
Optical	2	512	8-28	Push-Pull	100	5	50

In order to interface the low-current control signal coming from the MCU and enable it to drive the high-current signal requested by the motor, a motor driver is needed. However, a motor driver simply handles the power to drive the motors and the logic and digital control has to be done by an external device. A motor controller on the other hand, has all of the logic circuitry built in and can be controlled by a higher-level interface [17].

For this project, the ESCON 50/5, from the manufacturer Maxon Motor, was selected. It is a small-sized, powerful 4-quadrant PWM servo controller for the control of permanent magnet-activated brushed DC motors or brushless EC motors with an efficiency of 95%. The servo controller is controlled by an analog set value specified via analog voltage, external or internal potentiometer, fixed value, or by means of PWM signal with variable duty cycle. The ESCON 50/5 includes protective circuits against overcurrent, excess temperature, undervoltage and overvoltage, against voltage transients and short-circuits in the motor cable [18].

Figure 5.4: ESCON 50/5 Motor Controller

Source: [18]

5.2.2 Motor Model

At first, a state-space model depicting the motor was created. The general state space form is depicted in Equation 5.2.2:

$$\begin{aligned} \dot{\boldsymbol{x}} &= \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u} \\ \boldsymbol{y} &= \boldsymbol{C}\boldsymbol{x} + \boldsymbol{D}\boldsymbol{u}. \end{aligned} \tag{5.0}$$

Following this form and deriving the differential equations for a motor with a stiff shaft, the subsequent matrices A, B and C become:

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 \\ 0 & -\dfrac{k_{emf}k_m}{J_m R} \end{bmatrix} \qquad \boldsymbol{B} = \begin{bmatrix} 0 \\ \dfrac{k_m}{J_m R} \end{bmatrix} \qquad \boldsymbol{C} = \begin{bmatrix} 0 & 1 \end{bmatrix}, \tag{5.0}$$

where k_{emf} and k_m are motor constants, J_m is the motor shaft inertia and R is the resistance of the motor. Regarding the values of the parameters, some were given by the manufacturer of the motor and others were identified using dSPACE and the System Identification Toolbox from MATLAB. The values can be found in Table 5.3.

Table 5.3: Numerical values for the motor parameters

Parameter	**Symbol**	**Value**	**Unit**
Motor constant	k_{emf}	0.083	Vs/rad
Torque constant	k_m	0.083	Nm/A
Equivalent torque	J_{eq}	0.000 779 76	kgm^2
Motor resistance	R	4.32	Ω

Applying a 24 V step input to this system yields a response showing that the motor model reaches 289 rad/s, which is approximately 2800 rpm (full speed). This can be seen in Figure 5.5.

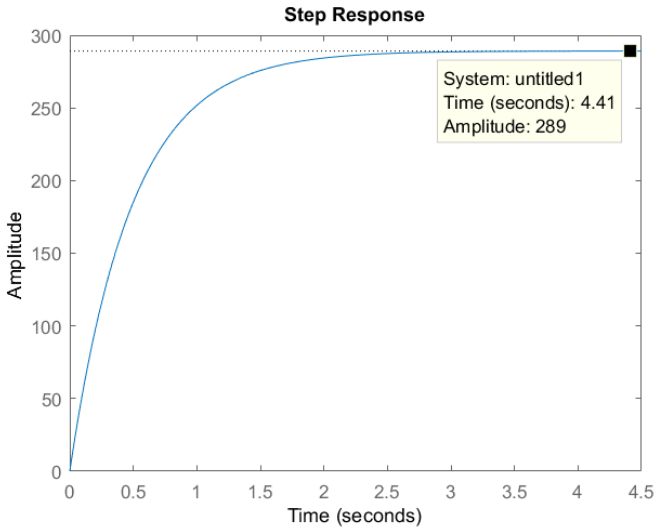


Figure 5.5: 24 V step response motor model.

5.2.3 Stiff Model

In this case, the boom of the crane was modeled as a stiff member and its inertia was identified. First, the inertia was calculated theoretically and verified through a simulation done in SolidWorks. Afterwards, this value was used as a starting point in the experimental parameter identification, again performed with the support of dSPACE.

The weight w_p of the PVC pipe can be calculated by multiplying the density by the volume, $w_p = \rho_m A_m l$. The cross-sectional area was a ring in this case, so the equation above can be rewritten as $w_p = \frac{\rho_m \pi (d_o^2 - d_i^2) l}{4}$, where d_o is the outer diameter of the ring, d_i the inner diameter and l the length of the boom. Substituting in the values for theses variables, one gets the result that the weight of the PVC pipe is about $w_p = 4.849kg$.

To find the inertia exerted by the PVC pipe, the parallel axis theorem was used. If the inertia I would be calculated around an axis passing through the center of gravity (COG) of the cylinder, the following formula could be used directly: $I = \frac{ml^2}{12}$. However, the axis of rotation does not go through the pipe's COG and thus, the parallel axis theorem must be applied. The inertia around the new axis $I_{newaxis}$ can be calculated via the theorem according to

$$I_{newaxis} = \frac{ml^2}{12} + md^2, \tag{5.0}$$

where m is the mass of the pipe, l the length of the pipe and d the perpendicular distance between the old and new axis of rotation. Substituting values for variables, the inertia is $I_{newaxis} = 4.7095kgm^2$.

Similarly, the inertias stemming from load and counterweight can be added in order to find the total inertia:

$$I_{total} = I_{newaxis} + m_{weight} * d^2 + m_{counterweight} * d^2_{counterweight}. \tag{5.0}$$

Substituting in known values for the variables, the resulting total inertia becomes $I_{total} = 10\,8913 kgm^2$.

As stated earlier, a simplified model of the boom was designed in Solidworks and the theoretical value obtained was verified with the aid of a simulation. Results are shown in Figure 5.6.

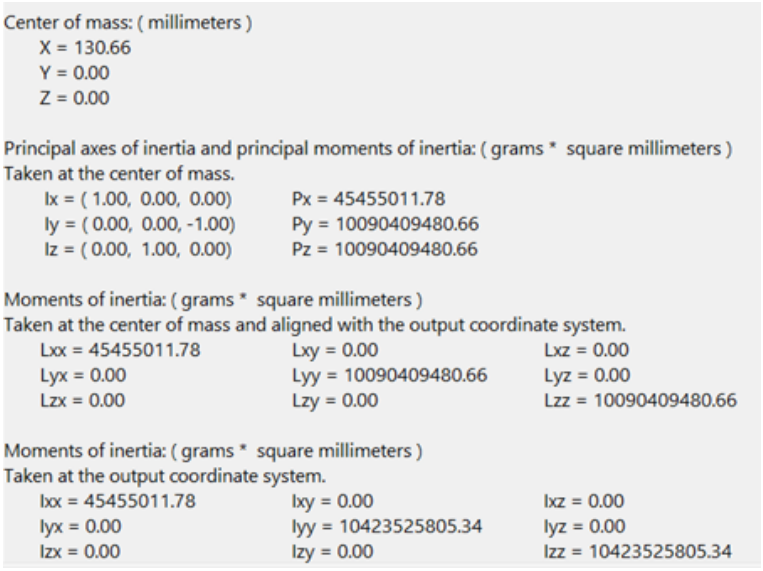


Figure 5.6: Results obtained from the properties of the simplified boom model

Once the theoretical value of the inertia was defined, it was used to build the stiff model of the crane. As before, following the state-space form and deriving the differential equations for a motor with a stiff boom connected to its shaft, the subsequent matrices A, B and C became:

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 \\ 0 & -\dfrac{k_{emf} k_m n}{J_{eq} R} \end{bmatrix} \qquad \boldsymbol{B} = \begin{bmatrix} 0 \\ \dfrac{k_m}{J_{eq} R} \end{bmatrix} \qquad \boldsymbol{C} = \begin{bmatrix} 0 & 1 \end{bmatrix}, \tag{5.0}$$

where k_{emf} and k_m are motor constants, n is the gear ratio, J_{eq} is the equivalent load inertia when moved to the motor side and R is the resistance of the motor. Several values were already estimated in the parameter identification performed for the motor model. J_{eq} was initialized with the values obtained through the calculation and then was experimentally obtained using dSPACE.

Table 5.4: Numerical values for the Stiff Model parameters

Parameter	**Symbol**	**Value**	**Unit**
Gear ratio	n	600	–
Equivalent torque	J_{eq}	0.0004	kgm^2

Having obtained the stiff model that better represented the system as a whole, the two controllers velocity and position, were implemented. For velocity, a PI-controller with a low-pass filter was used. For position, a PD-controller with a low-pass filter. These were chosen based on the orders of the transfer

functions and plants, 1st and 2nd order for velocity and position, respectively. A PID was considered for position but it was found to saturate and there was not enough time to develop an anti-windup. However, the steady state error with the PD controller was less than 5% and deemed acceptable for use. Both controllers were constructed using a feed forward and a feedback part. Figure 5.7 shows the control logic for the velocity controller. The design of the position controller is identical to the velocity one, with the only difference being the feedback signal.

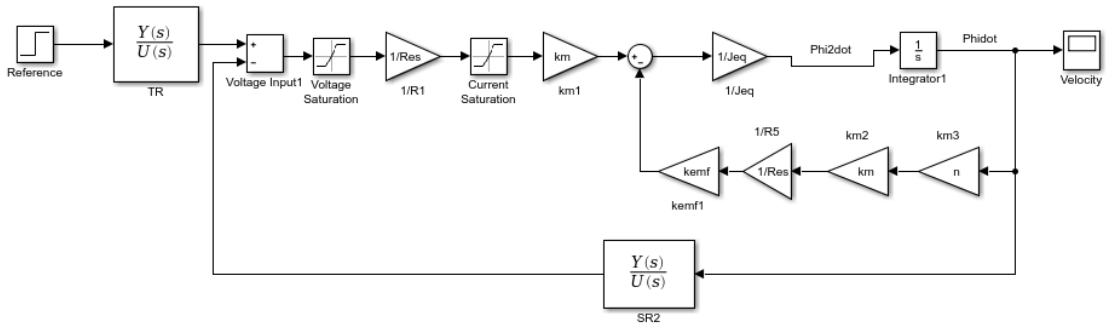
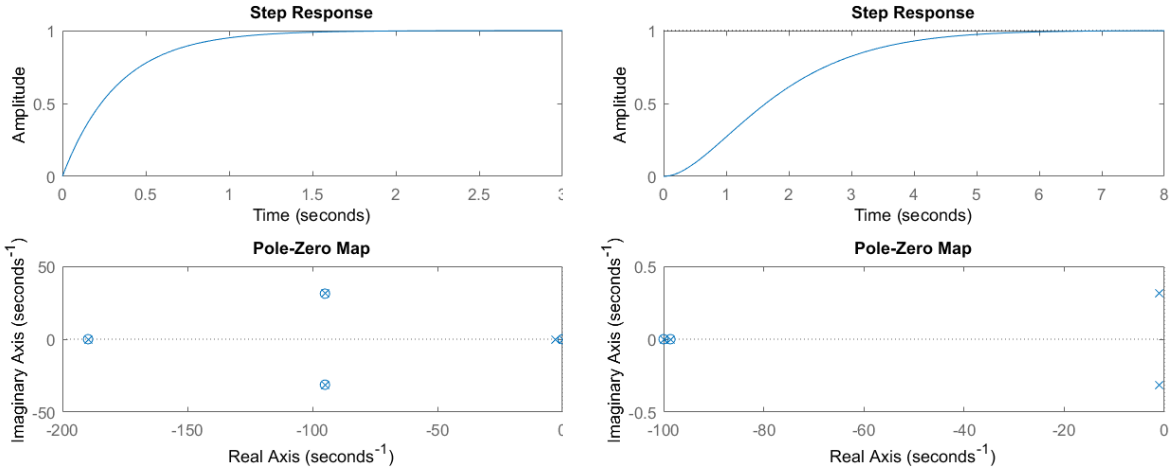


Figure 5.7: Simulink model for the slewing velocity control

With these control logics, the poles were placed to generate a soft responses for both controllers but taking into consideration the stakeholder requirements regarding time for a complete turn being around 14 seconds. Then, the step response and the pole-zero map of the two controllers can be seen in Figure 5.8.



(a) PI w. LP-filter velocity controller

(b) PD w. LP-filter position controller

Figure 5.8: Step response and pole-zero map for the controllers

5.2.4 Flexible Model

To make the model more realistic, the boom was modeled as a spring-damper system. The flexible boom model was first built in *Simulink*, and after that a *Simscape* model was created in order to verify the correctness of the initial model. The team was initially not sure whether the stiff model was sufficient for a good performance. A picture depicting the *Simscape* model can be found in Figure 5.9

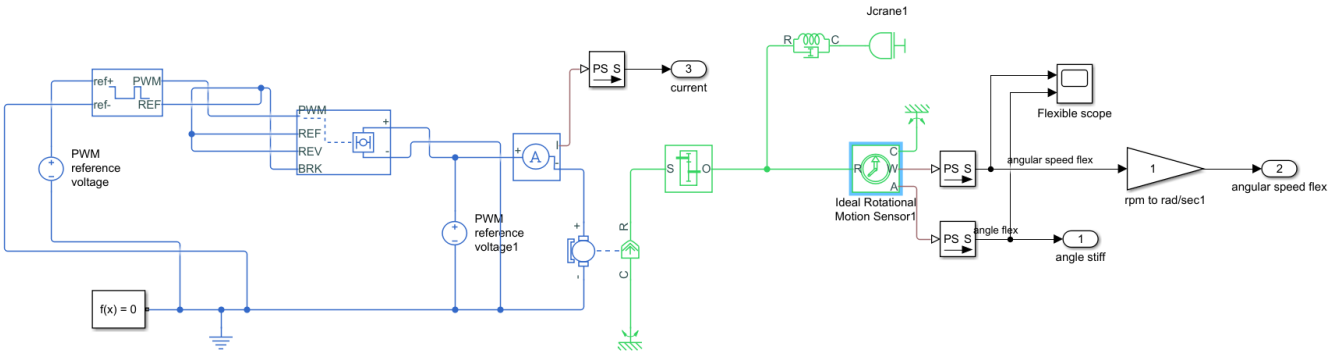


Figure 5.9: The *Simscape* model of the flexible boom.

For the controller of the flexible model, the block diagram seen in Figure 5.10 was utilized. The code used to generate the transfer functions can be found in Section **??** in the appendix.

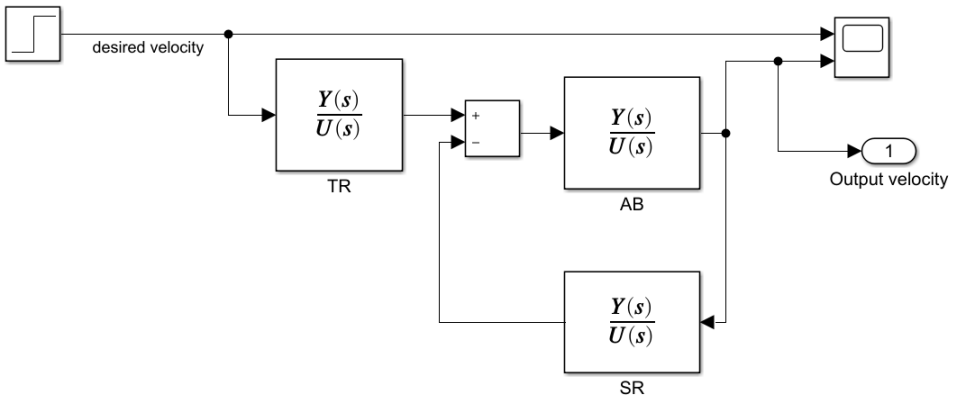


Figure 5.10: Block diagram depicting PI-controller for slewing motion.

Applying a reference velocity of 100rad/sec, the resulting behavior can be seen in Figure 5.11.

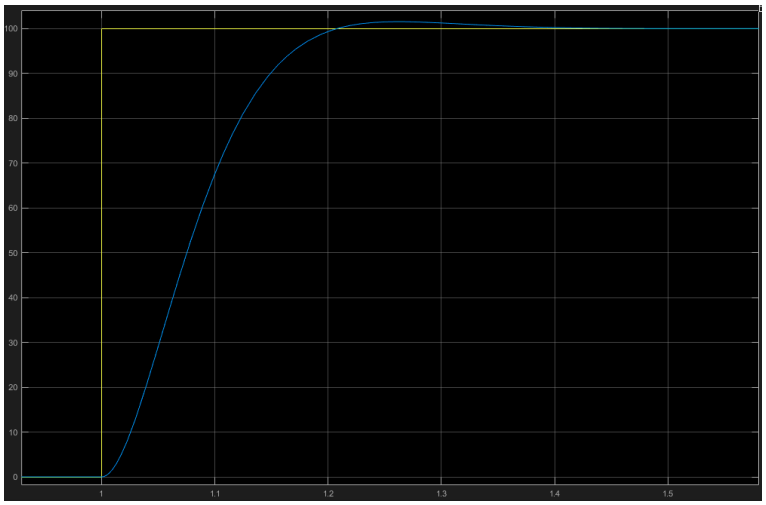


Figure 5.11: Step response of the flexible *Simulink* model with PI-controller.

A comparison between the two flexible models was made where a step was applied to the input of the respective plants and the resulting output velocity was observed. The resulting plot can be seen in Figure

5.12. The similarities in models lead the team to believe that the modeling was done correctly, especially with the help of *Simscape* intuitive layout.

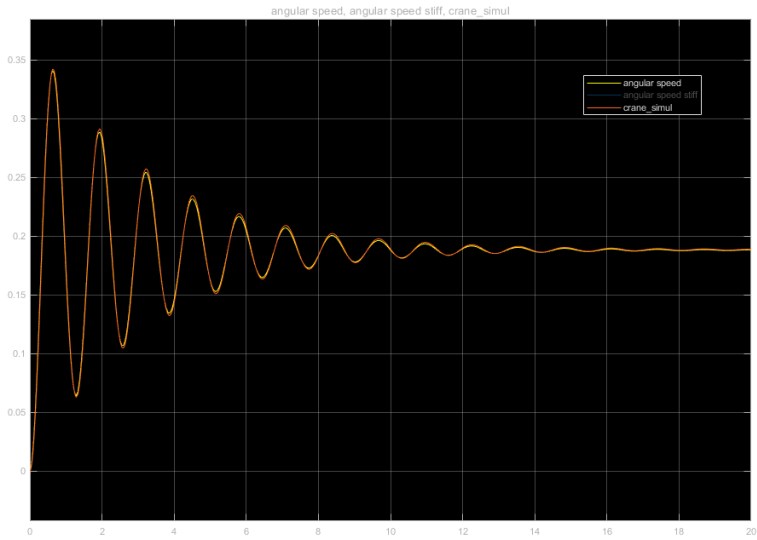


Figure 5.12: Comparison between the flexible *Simulink* and *Simscape* plant models. Orange curve corresponds to the *Simulink* model and yellow corresponds to *Simscape*.

However, after the physical system had been built and the controllers had been implemented, the boom's behaviour was deemed highly non-flexible. From that point onwards, the flexible models were abandoned and it was decided that further development would be on the stiff model.

5.3 Implementation

In this project, the Escon 50/5 was used in Current Control Mode and the velocity and position controllers were computed in an external microcontroller. The control signal coming from the microcontroller was connected to the Pulse Width Modulation (PWM) input of the driver and also the enabling signal that came from the control panel switch. The main electrical characteristics can be seen in the Figure 5.13 below. For more details about the connections review the next section: Assembly.

ESCON 50/5 (409510)		
Electrical Rating	Nominal operating voltage $+V_{CC}$	10…50 VDC
	Absolute operating voltage $+V_{CC\,min}$ / $+V_{CC\,max}$	8 VDC / 56 VDC
	Output voltage (max.)	0.98 x $+V_{CC}$
	Output current I_{cont} / I_{max} (<20 s)	5 A / 15 A
	Pulse Width Modulation frequency	53.6 kHz
	Sampling rate PI current controller	53.6 kHz
	Sampling rate PI speed controller	5.36 kHz
	Max. efficiency	95%
	Max. speed DC motor	limited by max. permissible speed (motor) and max. output voltage (controller)
	Max. speed EC motor	150'000 rpm (1 pole pair)
	Built-in motor choke	3 x 30 µH; 5 A
Inputs & Outputs	Analog Input 1 Analog Input 2	resolution 12-bit; –10…+10 V; differential
	Analog Output 1 Analog Output 2	resolution 12-bit; –4…+4 V; referenced to GND
	Digital Input 1 Digital Input 2	+2.4…+36 VDC (R_i = 38.5 kΩ)
	Digital Input/Output 3 Digital Input/Output 4	+2.4…+36 VDC (R_i = 38.5 kΩ) / max. 36 VDC (I_L <500 mA)
	Hall sensor signals	H1, H2, H3
	Encoder signals	A, A\, B, B\, (max. 1 MHz)
Voltage Outputs	Auxiliary output voltage	+5 VDC (I_L ≤10 mA)
	Hall sensor supply voltage	+5 VDC (I_L ≤30 mA)
	Encoder supply voltage	+5 VDC (I_L ≤70 mA)

Figure 5.13: ESCON 50/5 Motor Controller Main Characteristics

Source: [18]

Sensor Signal Interface

A signal conditioner (interface) is a device or conjunction of devices that transform one type of electronic signal into another type of signal. This transformation could cover Amplification, Electrical Isolation, Linearization, Excitation, and others. In this project, the encoder from the slewing motor was powered with 24V so the signals from A and B channels were also at 24V. For the microcontroller to be able to use these signals a reduction in the voltage level was made by two voltage divider circuits (one per channel). The output from this circuit is two DC signals at 5V.

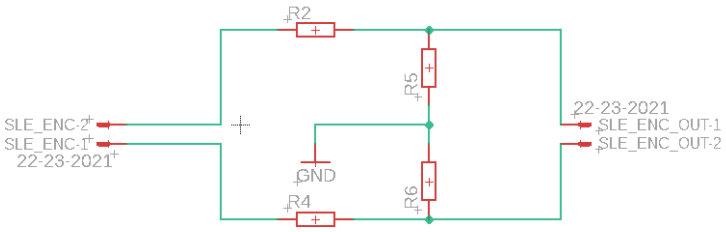


Figure 5.14: Signal conditioning circuit for Slewing Encoder

This circuit was combined with the pull up resistors required by the luffing encoder and a single PCB for interfacing the encoder signals was produced. For more details about the connections review the next section: Assembly.

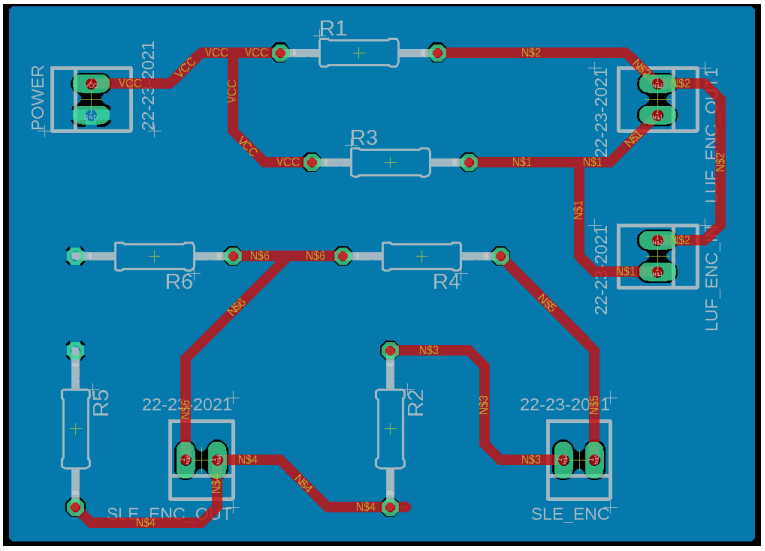


Figure 5.15: PCB for encoders interfacing

5.4 Assembly

5.4.1 Mechanical

First, 3D printed prototype parts for the structure were received from the stakeholder and the team started assembling the crane in combination with the components detailed in the previous section. For the slewing motion, the motor case was assembled. Then, the motor was allocated and fixed. Then, the slewing bearing was fixed to the case and the spur gear was coupled to the motor output in one side and to the slewing bearing in the other side. Once coupled, the slewing bearing was attached to the crane principal structure to enable the rotational motion of the complete system. To facilitate the wiring a 12 Wires Compact Capsule Slip Ring For Small Rotating Systems and Electric Motors was connected to the center of the slewing bearing through a rubber retainer that was 3D printed in the Prototype Center.

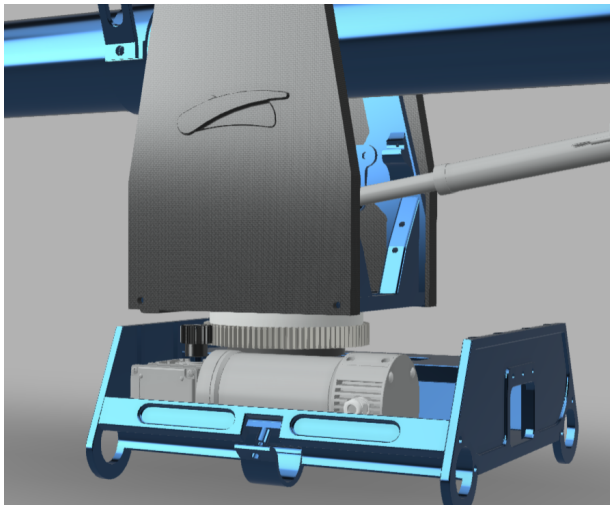


Figure 5.16: CAD showing the slew mechanical components assembled

5.4.2 Electrical

Regarding the electrical connections, both high level signals and low level signals were considered and connected according the following block diagram and wiring scheme:

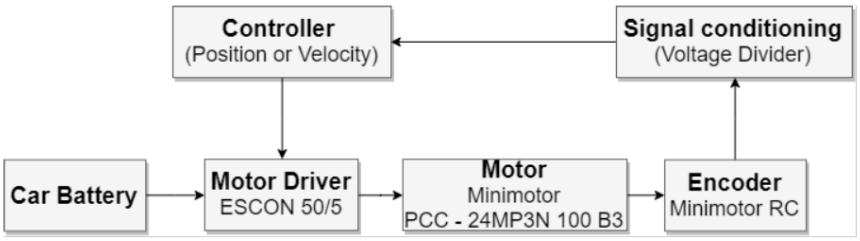


Figure 5.17: Block Diagram for the Slewing Motion

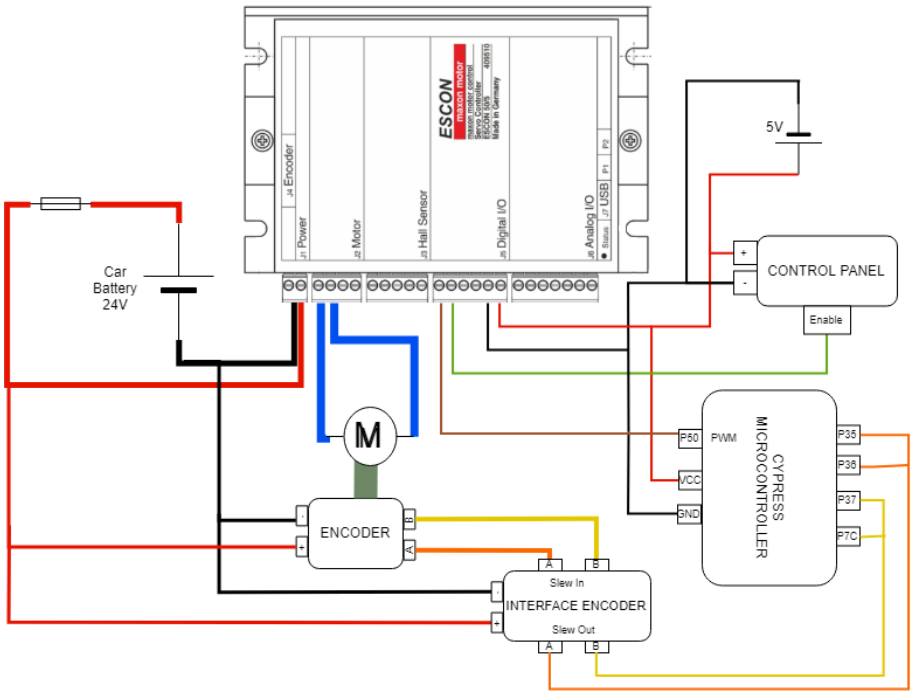


Figure 5.18: Wiring Scheme for the Slewing Motion

5.5 Testing & Verification

5.5.1 Motor

The following contains the Hardware-in-the-loop (HIL) simulations that were conducted with the help of dSPACE platform. These simulations correspond to the real motor and motor model response to different types of inputs. First, the results of the parameter identification are presented. Then, these parameters were used to design a velocity controller for the motor which was useful to analyze its response and performance without any load.

Parameter Identification-Step Response

The figure below corresponds to the parameter identification conducted for the motor. As can be seen, the transfer function obtained with manually tunning the parameters using dSpace was not completely satisfactory. Therefore, it was decided to use dSpace to capture the data and then the System Identification Toolbox from Matlab was used to analyze it. With this toolbox, a more fitted transfer function was obtained and based on the coefficients of the transfer function, the parameters were redefined.

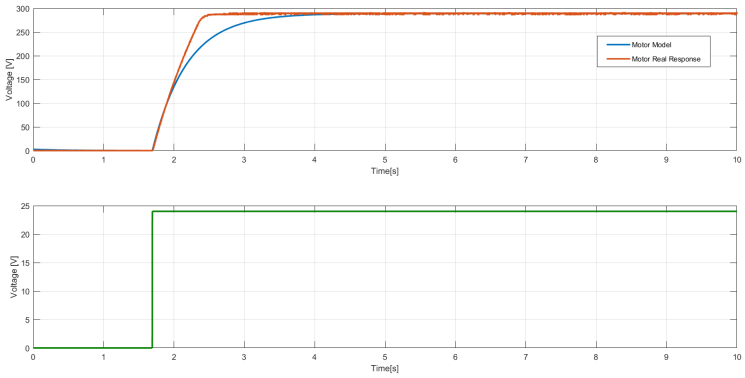


Figure 5.19: Parameter identification for the Slewing Motor

Velocity Control- Step Response

As mentioned earlier, a velocity controller (PI+LP filter) was implemented in the motor without any load connected to analyze its response and performance independently, which was satisfactory.

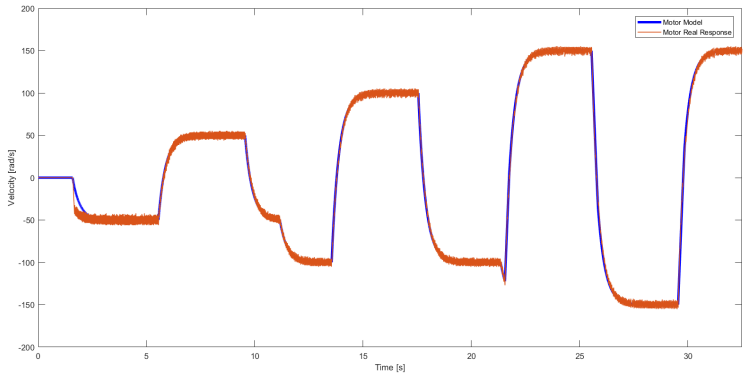


Figure 5.20: Velocity controller response to step input of different amplitudes

Velocity Control-Sine Response

With the same velocity control (PI+LP filter) a sine signal was input as a reference and yielded a satisfactory performance.

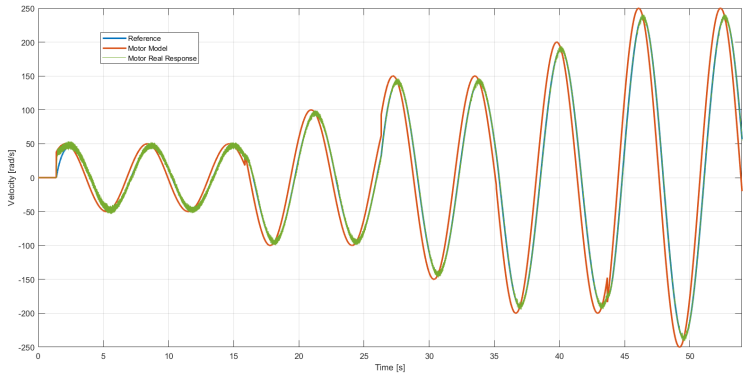


Figure 5.21: Velocity controller response to sine input of different amplitudes

5.5.2 Stiff Model

In this section the HIL simulations for the controllers (velocity and position) are presented. Both controllers were tested on their ability to follow different types of references and changing reference values.

Velocity Control- Step Response

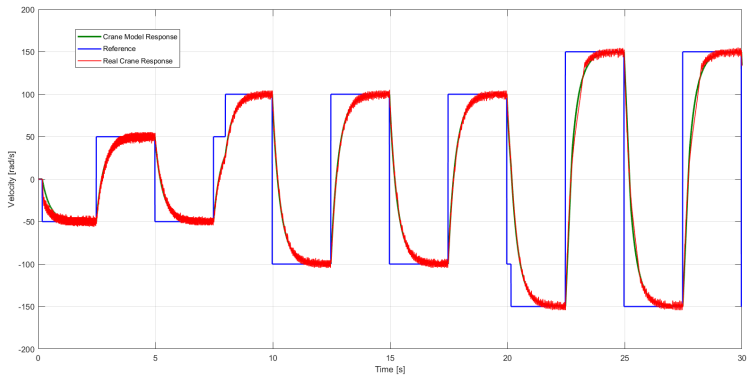


Figure 5.22: Velocity controller response of the crane to step input of different amplitudes

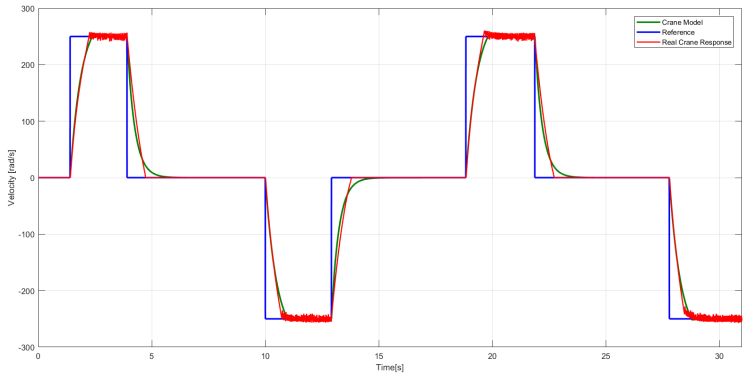


Figure 5.23: Velocity controller response of the crane to maximum step input of 251 rad/s (2400 rpm)

For clarification, 2400 rpm in the motor corresponds to 4 rpm in the crane. This was stated as a requirement from the stakeholder and was fully satisfied.

The plots show a response time of approximately 1 second and an average steady state error of less than 5%.

Velocity Control- Sine Response

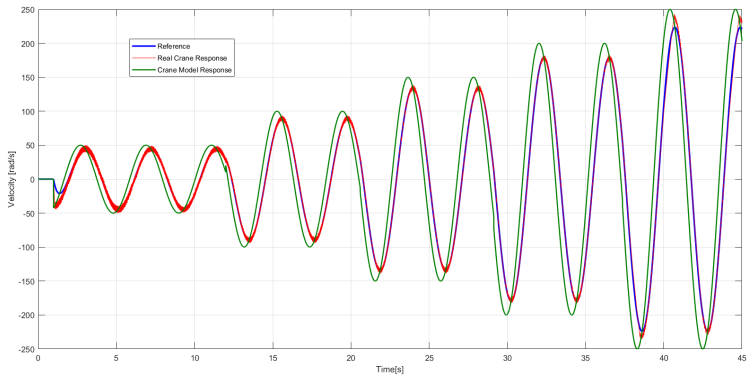


Figure 5.24: Velocity controller response of the crane to sine input of different amplitudes

Position Control

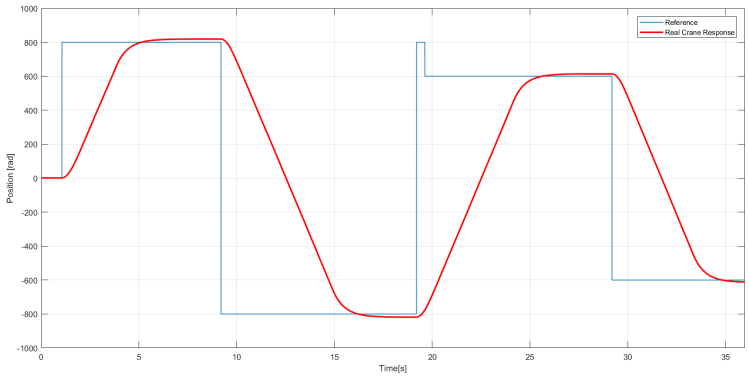


Figure 5.25: Position controller response of the crane to step input of different amplitudes

As it can be seen in the figure above, due to the type of controller (PD) there is still some steady state error. However, it was in average less than 2 degrees which was considered acceptable. For example, in the upper case, the error in steady state corresponds to 18.6 rads in the motor side, which will be 0.031 rads (1.77 degrees) in the crane side.

5.5.3 Results

The ability of slewing in both directions at the required speed of 4 rpm was proven through the testing and verification. Furthermore, the crane can rotate 360 degrees or even more.

A fluid movement is also achieved for this motion- no unexpected accelerations were witnessed.

Chapter 6

Control Panel

6.1 Overview & Requirements

The control panel is the interface between the operator and the crane. It allows the operator to control all motions and functionalities. In general, the stakeholder desired a large, robust joystick, buttons, knobs and switches, so that the crane could be controlled easily while in a moving car. Also, extra space for additional features was requested.

The final requirements for control are:

- Control panel shall control the velocity of the boom, both for luffing and slewing motions.
- Control panel shall have the ability to vary the maximum velocity of the crane.
- Control panel shall have the ability to remap the direction of the joystick controller.
- Control panel shall put the camera crane in a restricted motion mode, to be specified, enabling underground plane through a switch.
- Crane shall go back to original position when pressing home button.
- Control panel shall have the ability to record motion from user and can execute it upon pressing a specified button.
- Control panel shall have a on/off switch for enabling and disabling driver and controller.
- Control panel shall have an emergency button to cut off power when pressed.
- Control panel shall contain a screen which can visualize current control mode.
- Control panel shall have security measures implemented.
- Control panel shall have space for future functionality.

6.2 Design

The control panel layout and box was designed by the stakeholder after some recommendations from the team. It was printed in polyactic acide (PLA) plastic and is sloped to sit more comfortably on the lap of the operator.

6.2.1 Components

There are a variety of functionalities available on the control panel, as can be seen in Figure 6.1.

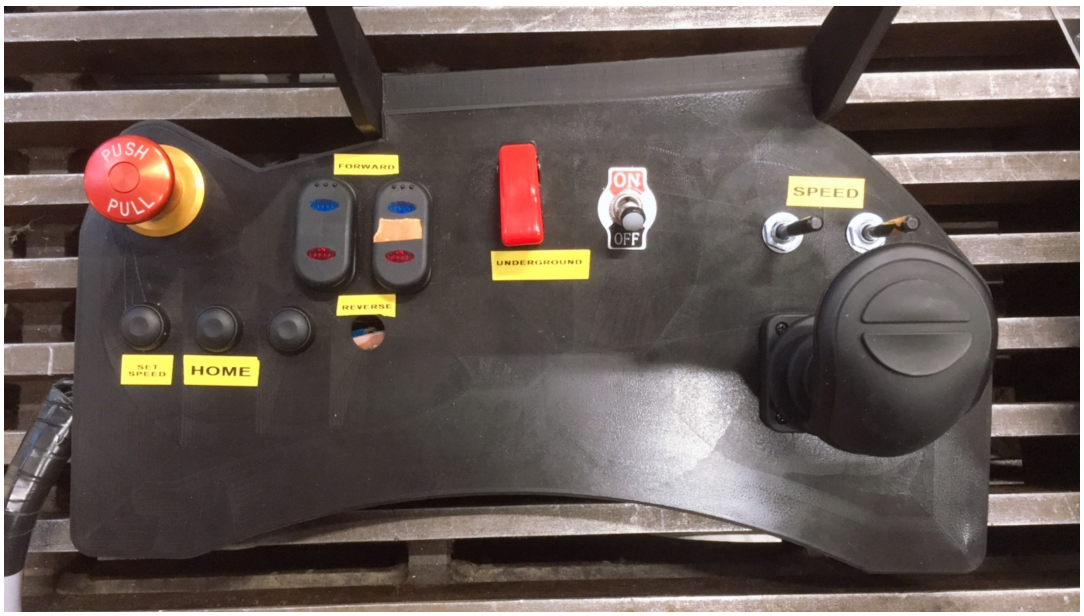


Figure 6.1: A picture of the control panel

These features include:

- Joystick

  Moves the crane system and boom. Luffing is controlled via front and back tilts while slewing is controlled with side-to-side tilts. There are two buttons on the joystick as well as z-axis rotation that are unused as of now, but they can be used for future features if desired.

- Buttons

  - Set Speed - This activates and sets whatever speed settings are currently on the speed knobs.
  - Home - This moves the crane into the ”Home” position.
  - Set Home - This captures the current position of the crane and sets it as the new ”Home” position.

- Switches

  - Left - Inverts the input of the joystick for luffing. For example: a forward tilt moves the boom down or when inverted, the same joystick motion moves the boom up.

  - Right - Inverts the input of the joystick for slewing. For example: a left tilt rotates the crane clockwise or When inverted, the same joystick motion rotates the crane counter-clockwise.
  - Underground - This switch enables or disables the ”Underground” mode, allowing the camera to go below the plane of the road.
  - On/Off - This enables the inputs on the control panel to reach the crane system or not.
  - Emergency - This button will shut down the crane system. It can only be reset by resetting the emergency button as well as the On/Off switch. More is explained in Section 8.2.3.

- Speed knobs
  Each knob is a potentiometer that controls the max speed for either luffing or slewing.

A Raspberry Pi was chosen as the main controller. There were many reasons for this choice. Among the biggest are:

- Ease of Use. Python3 is pre-installed on the Raspbian operating system and there are a lot of resources available about how to apply each function.
- I/O port. The MCU in the control panel needs to receive signals from many components, like the buttons and switches. A powerful feature of the Raspberry Pi is that it has a row of general purpose input output (GPIO) pins along the top edge of the board. These will be needed for the wide range of components required.
- CAN capabilities. The Reference signal will be sent to the controller through CAN communication so the MCU in the control panel must have CAN capabilities. A PiCAN2 board provides CAN Bus capabilities for the Raspberry Pi.
- Accessible: Compared to other choices, the Raspberry Pi is more popular, which means it is cheaper and easy to get from suppliers.

There is no analog to digital converter (ADC) in the Raspberry Pi, however. So a separate one is required to make the analog joystick signal readable. The MCP3008 ADC was selected as it is compatible with the Raspberry Pi. For the joystick, the HF Series Hall Effect joystick from APEM is selected. As shown in Figure 6.2, it has three axes of rotation and two buttons to configure and control. These joysticks are supplied with a Hirose DF11-12DP-2DS9(24) connector [3], which has 12 channels. Besides a power and ground channel, each axis has 2 x 2 output channels, common output and dual output, which will give different ranges of output from joystick. Only x-axis and y-axis output, channel 3 and channel 5 respectively, were used and sent to the MCP3008 for analog to digital conversion.

Figure 6.2: HF series Hall Effect Joystick [3]

With a 5V power supply, the MCP3008 uses channels 0 to 7 to receive analog signals from the joystick. Serial Peripheral Interface (SPI) was used to read signals from the chip. It is a synchronous serial bus commonly used to send data between micro controllers and small peripherals such as shift registers, sensors, and SD cards [6]. The MCP3008 is a 10-bit ADC, so it will read a value from 0 to 1023, where 0 corresponds to one end of joystick and 1023 corresponds the other end. This value will be regarded as the velocity reference value which will later be sent to the Cypress MCU.

6.2.2 Communication

The reference signal from the control panel is sent to the Arduino MCU through CAN communication. A CAN bus was chosen because information from multiple components can be communicated efficiently, quickly (1 Mbsp) and robustly within 3 meters.

The signal sent from the Raspberry Pi contains: luffing speed reference, slewing speed reference, home button status and underground switch status. The range of the speed reference is 0 to 1023, so for each speed reference, two bytes are used to transmit the signal. When the home button is pressed or when the underground switch is enabled, their messages should be 1. Messages will be 0 when the home button is released and when switch is disabled. In this way, another two bytes are occupied to send home and underground statuses. The layout for CAN message is as shown in Figure 6.3:

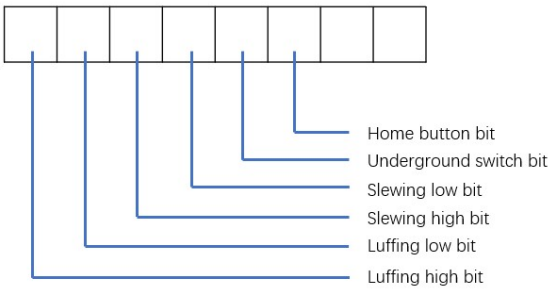


Figure 6.3: Layout of a CAN message

6.3 Implementation

The signals from the joystick and knobs were sent from the MCP3008 and directly read by the Raspberry Pi. To finish the button and switch functionality, the script needs to read the state of the buttons and switches and execute code based on that state. The GPIO library is imported to read the status of the pin and GPIO.setup is called to set the initial state of the pin.

The function of the buttons should only be executed when a high state is detected. Initial states for all GPIO input pins for buttons were set as low state. When a button is pressed, it will switch to high state and generate an interrupt which will call the specific function that is coupled to that button.

Binary switches, such as underground mode switch and On/Off switch, were used in the same way as buttons but could also detect low states. Both sides need to be detected to enable and disable functionality. Non-Underground mode and switch-off mode were enabled by default.

The final part is a three-state-switch for the inversion switches. Both ends are connected to two different GPIO ports. When one side is pressed, the corresponding mode will be enabled, and the other mode will be disabled at the same time. The inverting switches work together with the joystick to change the reference value for the camera crane. The reference value ref is dependent on the joystick value J, speed ratios from knobs m, and the state of the forward-reverse switch (swh). It is calculated according to:

$$ref = 512 + sgn(swh) \times m \times (J - 512) \tag{6.0}$$

For example, when the speed ratio goes to the maximum ($m = 1$) and joystick moves to the end position ($J = 1023$), when it's in forward mode, reference value will be 1023, while in reverse mode it will be 0.

A PICAN2 board is used to provide CAN-bus capability for the Raspberry Pi, and CAN is connected via standard 9-way sub-D connector. The CAN board is powered by the Raspberry Pi with a 5V input. The config.txt file in Raspberry Pi was changed to enable CAN functionality and the bitrate was set to 500 kHz.

6.4 Assembly

6.4.1 Mechanical

The Raspberry Pi, PICAN shield, PCB board and all the associated cabling were put inside the control panel box. A multi-stranded cable is used for power and CAN communication out of the panel.

6.4.2 Electrical

A PCB was designed to connect all necessary components in the control panel, including joystick, buttons, switches, knobs, Raspberry Pi and PICAN2 CAN shield.

Buttons and switches were connected to the GPIO pins on Raspberry Pi as digital inputs. The joystick and knobs were connected to the GPIO pins through MCP3008, the ADC chip, as analog inputs. Because the power source for the control panel comes from a 24V battery, a voltage divider is applied to transform the input voltage from 24V to 5V.

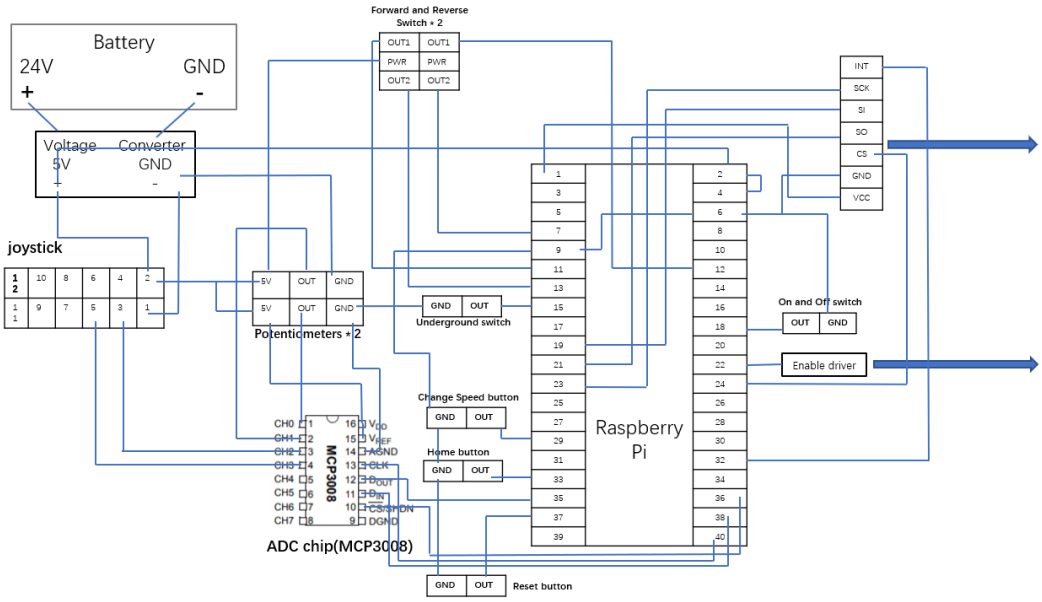


Figure 6.4: Wiring in control panel, output on the top is CAN message and output on the bottom is power source to driver

The PCB was designed in EAGLE and the board was printed and cut in the prototype center. A Schematic for the PCB is shown in Figure 6.5.

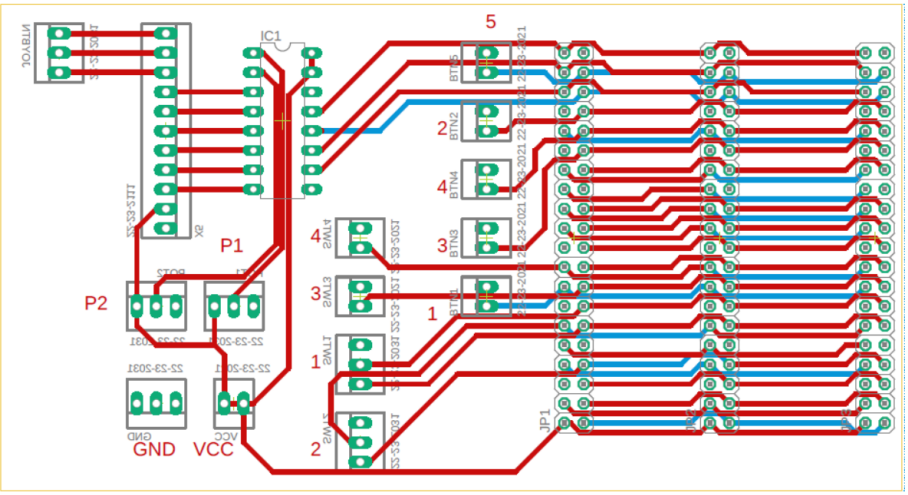


Figure 6.5: Schematic for the control panel PCB

The CAN board cannot connect directly to the Raspberry Pi since the GPIO ports on the Raspberry Pi were already in use. So a PCB was needed to make the connection between them. Several pins were connected to do the configuration and the wiring is shown in Figure 6.6. A CAN adapter is connected to CE0 and bus SPI0.0 is used.

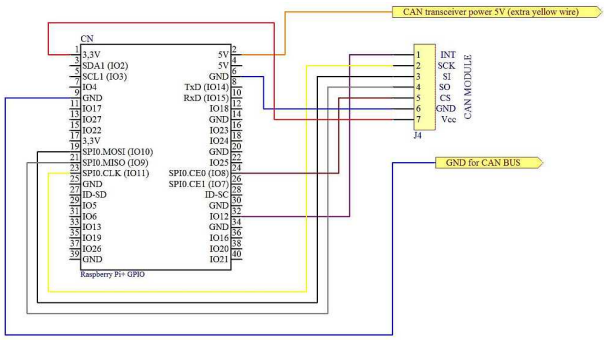


Figure 6.6: Wiring Pi to CAN adapter

Source: [5]

6.5 Testing & Verification

6.5.1 Specific Requirements

The requirement for testing control panel is to simulate as possible as can be thought of user's behaviors in real life with combination of different functions. Under those test case scenarios, control panel should be robust enough, work stably and send correct message through CAN. To test the message through CAN, a CAN King by Kvaser is used to print message information on another PC.

6.5.2 Test Cases

- Joystick
  - Till joystick 100% in one direction for luffing.
  - Till joystick 100% in one direction for slewing.
  - Tilt Joystick 100% towards one corner and repeat this step for each corner.
  - Repeat above tests for tilling joystick but at 25%, 50% 75%.
- Forward-and-Reverse Switches
  - Make combination of Slewing and Luffing direction switches in forward and reverse mode, in those four modes, slew crane to left and luff crane up.
  - With both Slewing and Luffing direction in forward, slew crane to left, stop, switch luffing direction to reverse, move joystick in the same direction as before.
  - With both Slewing and Luffing direction in forward, slew crane to left, stop, switch luffing direction to reverse, move joystick in the same direction as before.
  - With both Slewing and Luffing direction in forward, luff crane up, stop, switch luffing direction to reverse, move joystick in the same direction as before.
  - With both Slewing and Luffing direction in forward, luff crane up, stop, switch slewing direction to reverse, move joystick in the same direction as before.

- Knobs
  - Set speed to highest setting for Slewing and Luffing. Rotate crane 360 degrees at max velocity and record time.
  - Set speed to highest setting for Slewing and Luffing. Luff crane from bottom to highest position at max velocity and record time.
  - Set speed to 33% and 66% to highest setting for Slewing and Luffing. Luff crane from bottom to highest position and rotate crane 360 degrees at that velocity, record time and compare differences.
- Home Button and Underground Switch
  - Press home button and enable underground switch, check if CAN message changes to 1.
  - Press home button again and disable underground switch, check if CAN message changes back to 0.
  - Press home button and rotate till joystick, check if joystick function is disabled in Home mode.
  - Enable underground switch and luff crane down, check if crane can tilt to lower degree.
- E-stop Button and On/Off Switch
  - Move crane and ENABLE Emergency Stop. With Emergency Stop enabled, try to move crane.
  - Move crane and ENABLE OFF to On/Off switch. With switch OFF enabled, try to move crane.
  - With E-stop button pressed and switch OFF enabled, reset only one of them and try to move crane.
  - Move crane to random position, press Home Button, then ENABLE Emergency Stop or enable OFF to On/Off switch.
  - With crane on but not moving, ENABLE Emergency Stop and try to move crane.
  - With crane off but not moving, ENABLE Emergency Stop, turn on and try to move crane.

6.5.3 Results

Through the joystick, the both motions' velocity can be controlled. It is also possible to vary the maximum velocity thanks to the knobs allocated on the panel.

The functionality of reversing the direction of the both motions movement has been implemented. Furthermore, it is also possible to go beyond the degree limits of the crane when the underground mode is activated.

The home button functionality, which returns the crane to a preset position, is currently available.

The emergency button does work for shutting down the cranes motions. Except in the case of the Home button, where this motion does not use an enable function from the control panel as it is hard coded in the Cypress. This will need to be addressed as soon as possible after project hand-off.

The design of the control panel permits the addition of extra functionalities as was desired by the stakeholder.

Unfortunately, there are three requirements that could not be covered due to a lack of time. As of now, the panel does not allow for the recording and subsequent saving of crane positions. Only one position can be set and it must be specified in the Cypress. Restriction of use via password control was also not implemented. Furthermore, the panel does not contain a screen to visualize the current control mode.

Chapter 7

Hardware and Software

7.1 Overview & Requirements

Software was implemented on 3 different hardware platforms:

1. Raspberry Pi 3 B+: Interprets the control panel signals and translates them into CAN messages.

2. Arduino Uno: Acts as a communication bridge between the Raspberry Pi and the Cypress board. The Arduino receives the CAN message from the Raspberry Pi and passes it on to the Cypress via UART communication.

3. Cypress FM4-176L-S6E2CC-ETH: This board controls both motors. It is capable of reading encoder values, applying both position and velocity control laws as well as implementing the actions coming from the control panel.

The software is required to be able to translate the mechanical information the user sends through the control panel to the crane system. In other words, to be able to translate the user commands into a CAN message, which the Cypress can interpret correctly and subsequently execute the appropriate actions to control the crane. Furthermore, the software must be able to control the crane movements, slewing and luffing, in such a way that the crane moves as smooth and steady as possible. The capability of heading to a desired position is also a requirement depending mostly on the software implementation.

7.2 Hardware

7.2.1 Raspberry Pi

In the control panel, the basic idea for MCU is to receive messages from the user, translate them in a correct pattern and send the messages through CAN communication. The software in the Raspberry Pi is divided in different sections:

- GPIO, CAN and ADC Configuration: The first step is to choose correct GPIO port as input port to receive signals from buttons and switches. Both SPI0 and SPI1 are enabled in this step. For

ADC and CAN communication, CAN0 is set as the CAN channel and the CAN speed is set to 500 Kbps.

- Defining Function: Interrupts are used with Python on the Raspberry Pi for detecting state changes on buttons and switches. With interrupts, code can respond immediately to a specific event. Python can run more than one thread at once so a threaded callback function is used to go through more than one piece of code simultaneously [19]. With multi-threading, more functions and logic can be enabled in each triggered event. The basic idea for event detection with the GPIO ports is that the signal will be HIGH all the time until the state is changed, i.e. when a button is pressed that connects the port to ground, making it LOW. Raspberry Pi will detect the change and execute interrupt.
- Sending and Receiving: In the main function while loop, the Raspberry Pi keeps reading messages from the ADC MOSI port. Detailed logic for converting ADC message to CAN message is in Chapter 6.3, page 47. After conversion, Raspberry Pi sends CAN message through the CAN MISO port and makes a delay every millisecond to send the message in an appropriate period.

7.2.2 Arduino

It is worth mentioning that the Arduino was chosen as a back up plan. There were several issues trying to implement CAN communication on Cypress, leading the team to make this decision. Hence, the functionality sought after in the Arduino are essentially:

- CAN capabilities: The functionality of the Arduino is mainly receiving a CAN message and sending it to the Cypress through UART communication.
- Deliverability: The Arduino is a well known platform that could guarantee the results required in this specific application. Although not ideal, it was necessary to move the project forward.
- Affordable Price: As above mentioned, the Arduino is not part of the final product. Therefore, any microcontroller that can do the work is eligible for this functionality.

The CAN communication between the Raspberry Pi and Arduino is set 500 kbps. The code is constantly checking if there is a CAN message on the CAN bus. If there is, the message is taken and saved into a buffer. Following that, the data gathered in the buffer is sent serially as it is received via the UART communication. It is important that the data stored and sent in the buffer keeps the same CAN message order. Otherwise, the functionality commands will be changed, thus prohibiting the Cypress to interpret the message correctly.

7.2.3 Cypress

The MCU features that were desired regarding the control of motors are listed below:

- Powerful Microprocessor: The goal is to not miss encoder values. This guarantees the proper calculation of velocity and position for both motors, thereby yielding adequate controller actuation.

- Real Time Capability: Several tasks that cannot be interrupted are running on the MCU. Hence, the use of a scheduler seemed like a reasonable way of avoiding information loss.

- Motor Controlling Capability: There are some microprocessors that have a specific library for motor control purposes.

- I/O Port: An important specification is the required number of input and output ports. The idea was to have 8 external interrupts, 8 inputs ports (4 per motor), as well as 2 PWM output pins.

- CAN Capabilities: Since the functionalities are sent via a CAN message the receptor board had to be capable of acting as a CAN node.

- Affordable Price: Our final product goal is to have an affordable camera crane for independent film makers. Therefore, price was an important variable to consider.

- Deliverability: Time restraints warranted the preference of a product that could be quickly obtained.

A Raspberry Pi or Arduino seemed to be a good choice since there is a lot of information about them on the Internet, which means they would be easier and quicker to code. However, they do not have real time capability. The idea was to use a single MCU avoiding the necessity of using two, one for each motor. This design choice would imply a cheaper and more compact product. After researching various microprocessors the search was narrowed down into one, Arm Cortex M4. It was available on many development boards and also matched with the desired microprocessor features. Finally, pinout, deliverability and price were used as factors to choose between the development boards available. Among those, FM4-176L-S6E2CC-ETH by Cypress was the most convenient one. Its short delivery time allowed the team to start working as soon as possible.

7.3 Software

For a better understanding and a clearer picture, the software is divided in different sections or tasks.

7.3.1 Encoder Counting

The first step is to read the sensor value, in this case, an encoder. For this purpose the signal A and B from each encoder are connected to external interrupts. Unfortunately, Cypress does not have the capability of detecting changes in the signal. Therefore, one pin will detect HIGH and another LOW. Since there are two encoders, 8 pins are required. For counting the pulses properly, a lookup table was designed, making it possible to increase or decrease the encoder value according to the rotational direction.

7.3.2 Velocity and Position Calculation

: Knowing the encoder value and the pulses per revolution of the encoders, it is possible to calculate both position and velocity. There are 4 encoder values to 1 pulse. For velocity calculation, the calculation time interval is also required, so the calculation is implemented in a base timer.

7.3.3 Control Law

The control must occur periodically, always faster than or equal to the sampling time of the corresponding control law. Otherwise, it will lead to a bad control performance. The discretization and therefore the control law have been designed for a particular sampling time. Thus, the control is implemented in the base timer, one for each motor. What control paradigm is implemented (position or velocity) depends on the message received from the control panel. Parameters like voltage, reference and actual measured value are considered. For velocity, two prior time steps are considered in the algorithm. For position, one prior time step is considered.

7.3.4 Actuation

Depending of the output value given by the control law, the duty cycle of a PWM will be modified accordingly. This PWM will be the input of the driver for the motor.

7.3.5 CAN Message Receiver

Failing to implement CAN communication on Cypress due to lack of time and available information, an Arduino was used to receive the message. UART communication was used to pass the control panel CAN message received on the Arduino to the the Cypress.

7.4 Verification & Validation

It is really important to avoid any mistakes that could cause an error that would damage the whole system. This is why the team performed came up with as many test cases as possible to verify and validate the implementation, aiming for a reduction of our work in the long term.

7.4.1 Raspberry Pi

In order to make a good testing environment, the Raspberry Pi was connected to a screen through an HDMI cable to check if the code could print correct values of signals from the joystick, buttons and switches. Also, a CAN King by Kvaser was used to print CAN messages on the computer to check if the Raspberry Pi could send correct messages to controller.

7.4.2 Arduino

The Arduino Uno was tested against another Arduino Uno before integration into the final architecture. For CAN communication, CAN king by Kvaser was used.

7.4.3 Cypress

The software was checked on a test rig before being implemented on the real crane. The test rig consisted of a motor, driver, microcontroller and condition signal boards.

7.5 Results

The velocity control with the software results in a smooth and precise movement of the crane. Nevertheless, the movement presented a short delay from when the joystick was moved to when the crane actually moved. In general, the operator is able to control velocity as his/her will.

Furthermore, when position control is executed, the crane goes perfectly to the reference position. Upon further testing it was observed that the positioning algorithm fails to reach the desired position if the crane has been run for a long time. This reflects a lack of ability to measure the encoder values precisely. The software only presents the ability to go to the ”zero position” or initial position, failing the requirement of being able to program a desired position and execute the motion via buttons.

The remapping direction, underground mode and the maximum velocity functionality for the joystick controller and knobs are working as expected, fulfilling the requirement established by the stakeholder. However, the necessity of adding a password for using the control panel is not implemented via software as of now.

Regarding buttons and switches signal interpretation, there is a slight bouncing of the signals caused due to these being unstable once the switches or buttons are actuated. However, this time is insignificant, without affecting any aspect of the system.

After changing dSPACE for our architecture the control laws were changed by choosing slower poles. Meaning that Cypress is not as fast as dSPACE and could not handle such an aggressive performance demand. With such a fast poles, the system was calling for too much current, causing the drivers to shut down due to overpassing the established current limit.

There is also a shift in the CAN message after a long run with the crane. This is resulting in an uncontrolled behavior, since all functionalities are totally dislocated.

Chapter 8

System

8.1 Overview & Requirements

This chapter reviews how the crane was powered and its electrical considerations. Only one requirement is demanded in this area, that the system shall be battery powered. Nevertheless, the system requires more in depth attention and the addition of some other functionalities considered necessary.

8.2 Electrical

8.2.1 Power Supply & Management

The system is powered by two 12V car batteries. Power is run from the battery box to the fuse box and then divided to the whole crane. Inside the fuse box the power is supplied through two fuses. One 8A fuse going to the drivers and powering the motors and one 5A fuse powering all the micro controllers.

Battery life is preserved via the complete shut down. It was decided to not use a deep sleep process because of increased complexity.

8.2.2 Control Panel

The control panel is connected to the crane through a multiple strand cable holding 4 cables for the CAN communication, one for power, one for ground and a last one for the enable signal to the drivers. Inside the control panel a PCB was designed to connect all the buttons, the joystick signal and the CAN-shield to the Raspberry. There is also a voltage divider for turning the input voltage of 24V to 5V, supplying the Raspberry Pi with power. More can be read about the control panel in Chapter 6.

8.2.3 Emergency Stop

An emergency stop was implemented on the control panel to ensure a quick way to kill the motors if an unwanted motion occurs. The emergency stop is connected between the Raspberry Pi and the enable pin on the motor drivers and is in a normally closed position. When pressed, the switch opens the circuit and the enable signal is no longer sent to the drivers.

The emergency button also has a normally open signal going back to the Raspberry Pi. When the button is pressed the loop is closed and the signal is sent out and picked up, letting the Raspberry Pi know the state of the emergency switch.

8.2.4 Power loss detection and non-volatile memory storage

The system powers off via a switch on the control panel. There is also a manual switch by the batteries. Since the crane needs to record its position upon shut down, it is important that the system has enough time to save this data to non-volatile memory (NVM) before losing power. This functionality is necessary in order for the crane to know where the ”Home” position is. Without this functionality, data will be lost upon powering down the system and there will be no way of knowing where said home position is. It would then have to be redefined every time the system is turned on, which is a problem this solution aims to avoid.

The way that this functionality is implemented in this system is by having a Cypress MCU reading the encoder signals which holds information about the boom's position. This information is further sent periodically via UART communication to an Arduino MCU which has a NVM, known as EEPROM. NVM:s will continue to store data, even when power is no longer supplied. In order to give the Arduino some time to save critical data on power loss, an RC-circuit was placed on its input. Upon supplying power, the capacitor will thus be charged and upon disconnecting the main power supply, the capacitor will discharge and supply power to the circuit for a period of time that is long enough to save the critical data to the EEPROM. Then, upon switching the system back on, the encoder data is read from the EEPROM and further calculations can continue from there on. For detecting a loss of power, the voltage that the Arduino outputs from the 5V pin is monitored by coupling it to an analog input pin. The Arduino is programmed with an Interrupt Service Routine (ISR) that periodically checks the input voltage with a period of 10 ms and when it detects that the voltage is below 4.5 V, it saves the encoder data to EEPROM.

For a visual representation of the system, refer to Figure 8.1. Note that the Arduino is powered by USB, feeding it 5 V. The resistor value is 60 Ω and the capacitor has a capacitance of 6.8 mF, yielding a time constant of $\tau = RC = 400$ ms.

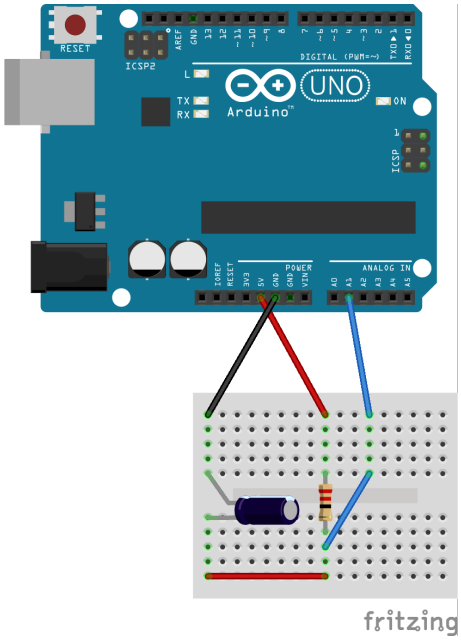


Figure 8.1: Fritzing sketch depicting the Arduino RC circuit

8.2.5 Results

As mentioned, the crane is successfully powered up with batteries. The requirement imposed is in this way achieved. Nevertheless, other functionalities were implemented, such as the emergency stop, which is vital for security reasons.

Although, the circuit for keeping on the Arduino in power lost circumstances is already on the architecture, the capability of saving the encoder value in the memory is not implemented.

Chapter 9

Discussion and Conclusions

9.1 Control Performance

The main challenge of the crane, the control performance, was a success; the boom was pretty steady all along its set of movements. Nevertheless, it can always be improved; some vibrations were noticeable on the boom in more aggressive movements.

9.2 Actuation Delay

Although the crane presented a good performance, a delay is noticeable from when the control panel is actuated until the crane responds. The main cause for this behavior is the unnecessary Arduino that translates the CAN message from the Raspberry Pi to the Cypress via UART communication. As a second solution, the reduction of the interrupts on the code would also help by avoiding the wait for other interrupts to be finished until the actuation interrupts happens.

9.3 Bouncing Signal

As mentioned earlier, the signal is unstable once a switch or button is actuated. Although, it does not really affect on the crane performance it is worth noting. This is definitely a electrical issue that cannot be sorted out via software. The most fair solution is to include some capacitors to damp the signals' rise and fall, avoiding this fluctuation of the signal. However, this solution might lead to a delay of the pin status. Therefore, the right capacitance must be chosen for this purpose, not too high so the delay is noticeable and not too low such that the fluctuations still occur.

9.4 CAN Message Shift

After actuating the crane for a long time, the CAN message sent from the control panel shifts one position. This leads the functionalities commands to be illogical, generating an unexpected output. The reason for this is probably dealing with the function reading the message being in the main; any interrupt will interrupt this function, hence, generating a disturbances on the message received. One

obvious solution would be the implementation of real time on the microcontroller or even adding this function on an interrupt. For both solutions, further changes might needed in the code, since currently there is one line that waits for a value to be received (the reason this function is in the main).

9.5 Encoder Values Precision

In order to detect the encoder values a lookup table was implemented. The rising and falling signal A and B of both encoder are detected using external interrupts. However, after running the crane for a while, data loss in the encoder values has been noticed. This could be explained due to the large number of interrupts in the code. There are 8 external interrupts just for detecting the encoder signal. Adding the rest of the interrupts makes the total amount 10. Interrupts are interrupting each other leading to loss of information in the encoder. The faster the crane is ran, the more interrupts will be thrown which might lead to more ISRs being interrupted by other ISRs.

9.6 Position Memory

Regarding the home button functionality, the main idea was to have a set home position. This home position would correspond to a certain encoder count value and the original idea was to save the encoder counts in a NVM so that the home position wouldn't be lost upon powering down the system. The Arduino was to receive the encoder count from Cypress and save it when a power-down was detected.

However, this functionality was never fully successfully implemented even though it was confirmed to work in a testing environment. There were reliability problems in the timing of the UART communication between Cypress and Arduino and therefore, the functionality was not implemented in the final build of the crane.

The consequence of this is that the encoder count will be reset every time the system powers down and whenever the home button is pressed, the crane will simply go to the position it was at when the system was powered on.

Chapter 10

Future Work

10.1 Path Programming

The controller in the camera crane can as of now only do velocity control based on a reference signal from the control panel and home mode position control based on preset home position. In future work, path programming can be added as a new function. The ground work for this functionality has already been laid down since path programming is based on position control. Cypress cannot record the correct encoder position now, once in the future the memory function is finished, developer can add more buttons on control panel to save several interesting positions in an array and trigger camera crane to travel around each position based on their encoder values and orders to accomplish path programming function.

10.2 Authorized Access

As our stakeholder mentioned, it is important to avoid any unauthorized operator to be able to operate the crane. Therefore, a password to be inserted on the HMI panel must be implemented as an access restricting tool.

10.3 Programmable Position

The necessity of being able to record a specific position and return to it once a button is pressed is a useful capacity that should be taken into considerations in the future. As initial though, it could comprise of a button for recording the actual position and a pad so it is possible to select which button the user wants to save that position to.

10.4 Real Time Capability

Unfortunately, the real time was not implemented on the Cypress. This would help to avoid meddling between interrupts, resulting in a higher encoder value precision and a proper actuation of the control law.

10.5 Encoder Values Precision

As described in discussion and conclusions chapter, some counts were lost. One possible solution is to apply a lookup table that only measures high or low, reducing the number of external interrupts in 4. Another possibility is to use base timer interrupted every specific known period. This way the encoder value is calculated only every period and not as frequently as if 8 external interrupts were implemented. Nevertheless, the real time implementation would help for this cause, avoiding interrupts to interrupts each other.

10.6 Bypass UART Communication

The future solution should get rid off the Arduino as a communication bridge between the raspberry and Cypress. Hence, Cypress will receive the CAN message directly from the HMI.

10.7 Further Dynamics Considerations

An extended and more accurate dynamic model of the crane will improve the control performance. As of now, only the motor and the boom have been considered in the model. However, for a more accurate model other camera crane parts must be modelled. For example, the cradle, the damped connection between cradle and the boom or the flexible connection between the camera and the boom.

10.8 Testing n Top of a Car

As the real environment of the crane, it will be required to test on top of a car to see the actual performance of the crane. Hence, some disturbances before not considered must be considered for the real scenario. Some examples of these disturbances are the wind or the vertical displacement of the car.

10.9 Implementation to Full Scale

As of now, the model is a representation of a 2m plastic boom with counterweight and tip weight simulating the real counterweight and the camera. Therefore, a new model must be implemented considering the 6m aluminum boom, the real camera as well as the corresponding counterweight.

10.10 Correctly Sized Components and Correct Encoders

Especially the luffing actuator was too weak for being implemented on the real actuator. Going forward it would be required to use a stronger actuator to use on the actual crane. To be able to control the velocity of that actuator would require an encoder that is designed for velocity control, and not only position control.

10.11 Safety

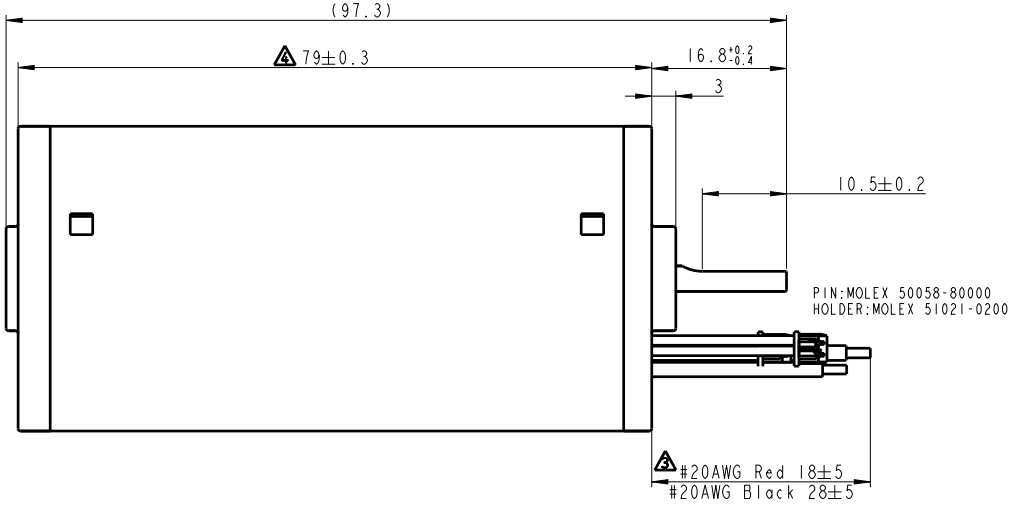
It is well known, that the crane will eventually be actuated on top of a car. Although mobility limits have been already implemented on the code, each car is different, therefore, different dimensions. One solution, although not recommended, is to modify the code accordingly to the car dimensions. One way of avoiding to modify the code every time the car is changed is to provide the crane with proximity sensor, automating this functionality.

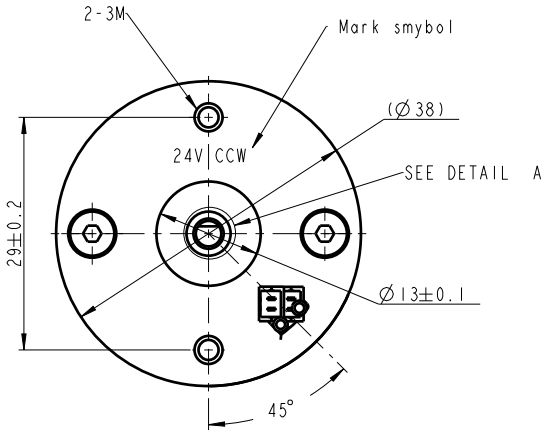
Appendix A

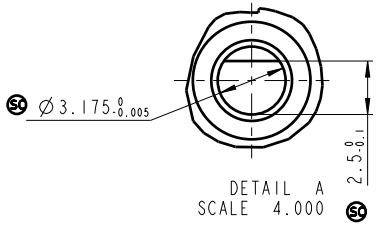
All Component Data Sheets

A.1 CAHB-10 Motor

REV	CHANGE DESCRIPTION	BY	DATE	APPROVED
2	更新BOM	Adong Wu	2011.07.04	Ford Cai
3	增加出线长度、CC、SC	Adong Wu	2012.08.27	Ford Cai
4	79±0.3 was 77.5±0.3	Adong Wu	2013.03.20	Ford Cai



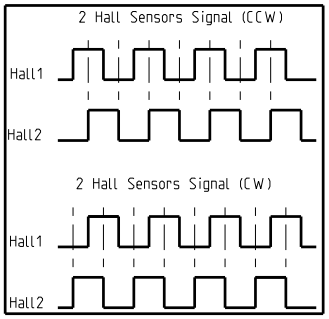




Notes:

1. Motor complied with EMC IEC61000-6-18-6-3.
2. Motor operating temperature range:-40~+85°C.
3. IP66s (EN60529) except of output shaft end without O-ring protection.
4. Relative humidity:90~100% at 40°C(IEC 60068-2-30).
5. Apperance anti-corrosion 700 hours passed for salt spray test (ISO9227 or equal), the tube surface treatment is zinc plating plus painting (light grey)
6. Motor shaft axial gap 0.05mm~0.5mm.
7. Motor shaft can not be given axial force over 30kgf
8. Hi-pot test:DC500V/1sec/3mA Max.
9. Duty cycle is 25%(2min ON/6 min off) Lifetime 10,000 cycles.
10. Thermal motor protection95°C;
11. Motor Low-voltage starting 1.5VDC max.
12. Motor can rotate CW and CCW,motor rotation direction: red with(+) black(-) CCW tunningview from motor shaft output end;
13. Two channel singal with 4 pole magnet ,Position sensor should have a rated voltage of 5 Vsensor shall have 2 Hall Effect sensors 45 degree (+/- 15 deg) electrically shifted.(typeI)
14. Motor comply to with RoHS.

Type I

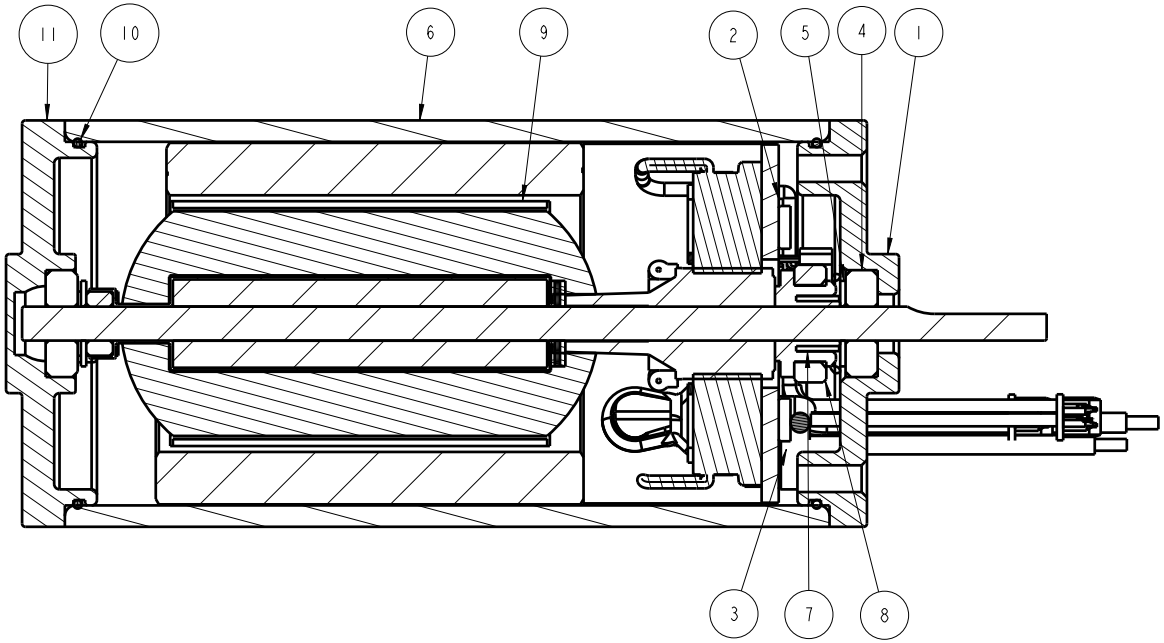


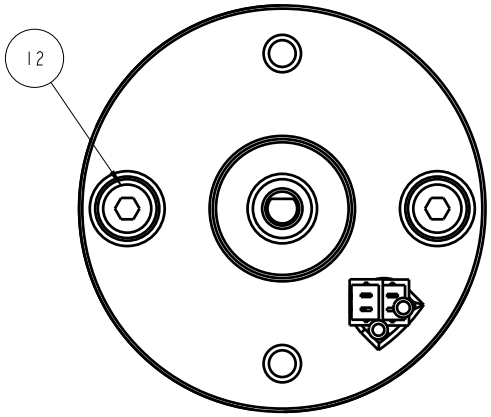
Motor Parameters	
Motor Type	3423
Voltage	DC 24V
Wire OD	∅0.3x45T
Rotor resistance	2.3~2.9 ohm(20°C)
No Load Speed	6000±10%RPM
No load Current	SC 0.3A Max
Rated Load	0.5kg.cm
Rated Speed	SC 5200±10%RPM
Rated current	SC 1.6A Max
Torque constant	0.32Kg.cm/A
Noise(At 0.5m Dimension)	<50dB

Pin Black 20AWG	Motor -
Pin Red 20AWG	Motor +
Pin Red 28AWG	+5V
Pin Black 28AWG	GND
Pin Yellow 28AWG	Hall sensor 1
Pin White 28AWG	Hall sensor 2

PRO-E/CAD FILE NO MANUAL CHANGES	DIM.IN	MM		MATERIAL		PART NAME: CAHB10 MOTOR / SHEET 1/2
NOTICE:UNLESS OTHERWISE STATED GENERAL TOLERANCE ACCORDING TO ISO2768-mK. SYMBOLS AND INDICATION ABOUT GEOMETRICAL TOLERANCE ACCORDING TO ISO 1101.	SCALE			FINISH		
SC MEANS SIGNIFICANT CHARACTERISTIC.	TOL.	>0-6	±0.1	DRW.	Adong Wu 2011.05.09	PART NUMBER: G61700-0160133 / REV 4
MEANS CRITICAL CHARACTERISTIC.		>6-30	±0.2	CKC.	Feng Gao 2011.05.09	
		>30-120	±0.3	CKD.	Bobby Guo 2011.05.09	SKF Actuation System(PingHu) Co.,Ltd.
		>120-400	±0.5	APPD.	Ford Cai 2011.05.09	

REV	CHANGE DESCRIPTION	BY	DATE	APPROVED
2	更新BOM	Adong Wu	2011.07.04	Ford Cai
3	增加出线长度、CC、SC	Adong Wu	2012.08.27	Ford Cai
4	79±0.3 was 77.5±0.3	Adong Wu	2013.03.20	Ford Cai





⚠2 ITEM	Part N.	Name	Spec	Material	UL	TYPE	QTY
1	G51700-065801	FRONT_COVER	Ø5.5*Ø38*12.25	ADC12	---	PART	1
2	G61700-017601	PCBA	SEN/RD18 BK28	---	---	ASSEMBLY	1
3	G51700-033302	SCREW	M2.5*6	SUS304	---	PART	2
4	G51700-042001	OIL_BEARING	Ø3.175*Ø10*3	CuFeZn	--	PART	2
5	G51700-018001	GASKET	Ø3.2*Ø8*0.5	PET	---	PART	2
6	G61700-017201	STATOR	Ø34*Ø38*71.5	---	---	ASSEMBLY	1
7	G51700-015301	SEAT FOR MAGNETIC	Ø3.175*Ø10*6.1	PET	---	PART	1
8	G51700-015401	MAGNETIC RING	Ø5.575*Ø11*3 4P	1A	---	PART	1
9	G61700-017001	ROTOR	COPPERWIREØ0.3*45T	---	---	ASSEMBLY	1
10	G51700-009001	O_RING	Ø0.8*Ø32.5	Si70	---	PART	2
11	G51700-065701	BACK_COVER	Ø13*Ø38*12.25	ADC12	---	PART	1
12	G51700-033303	CSREW	M3*71	SUS304	---	PART	2

PRO-E/CAD FILE NO MANUAL CHANGES

NOTICE:UNLESS OTHERWISE STATED GENERAL TOLERANCE ACCORDING TO ISO2768-mK. SYMBOLS AND INDICATION ABOUT GEOMETRICAL TOLERANCE ACCORDING TO ISO 1101.

KC MEANS SIGNIFICANT CHARACTERISTIC.
CC MEANS CRITICAL CHARACTERISTIC.

DIM.IN	MM		MATERIAL			PART NAME:	SHEET
SCALE			FINISH			CAHB10 MOTOR	2/2
TOL.	>0-6	±0.1	DRW.	Adong Wu	2011.05.09	PART NUMBER: G61700-0160133	REV 4
	>6-30	±0.2	CKC.	Feng Gao	2011.05.09		
	>30-120	±0.3	CKD.	Bobby Guo	2011.05.09	SKF Actuation System(PingHu) Co.,Ltd.	
	>120-400	±0.5	APPD.	Ford Cai	2011.05.09		

A.2 Slewing Ring

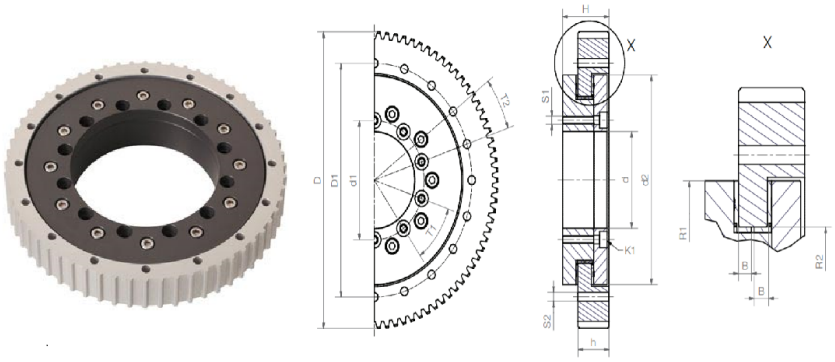


Figure A.1: Slewing Ring selected: IGUS PRT-01-100-TO-ST [2]

Table A.1: Dimensions of the slew bearing PRT-01-100-TO-ST [2]

Dimensions [mm]

Part Number	D1	d1	d	d2	h	T1	T2	S1	S2	K1	R1	R2	B	H
PRT-01-100-TO-ST	170	112	100	160	25	12x30°	16x22.5°	M5	5.5	DIN 912 M5	110	96.5	5.5	(37.5)

A.3 Spur Gear

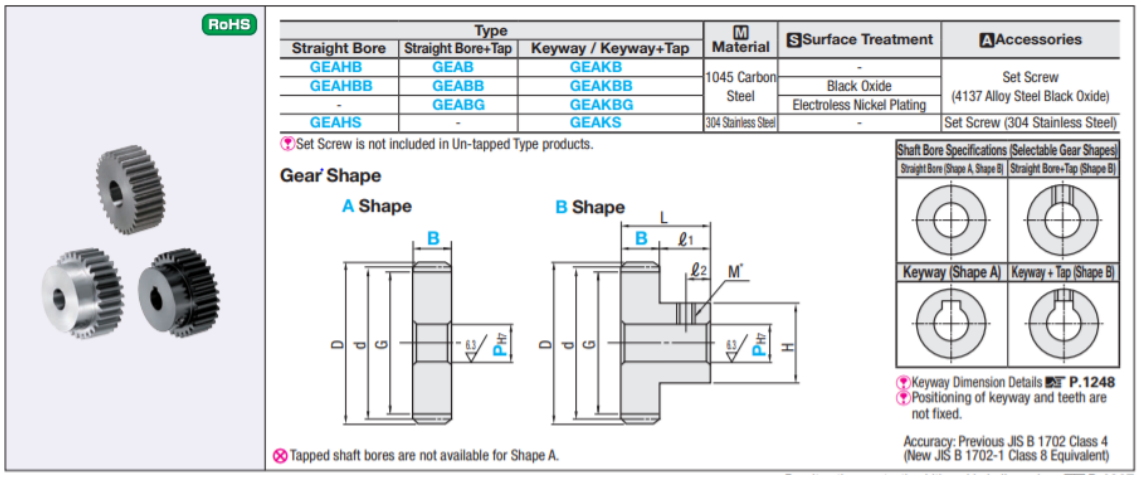


Figure A.2: Spur Gear selected: Misumi GEAHB 2-12-25-8 [9]

Table A.2: Dimensions of the spur gear GEAHB 2-12-25-8 [9]

Dim [mm]

Num of teeth	B	Gear Shape	Shaft Bore Diameter		d Ref Dia.	D Tip Dia.	G Root Dia.	H	L	l1	l2	M (coarse)	Allowable Transmision Force (N.m)
			Straight Bore/ Bore+Tap	Keyway/ Keyw.+Tap									
12	20	A	8	8N	24	28	19	18	30	10	5	M5	19.75

Appendix B

Equation Derivations

B.1 Luffing

B.1.1 Actuator Placement Geometry

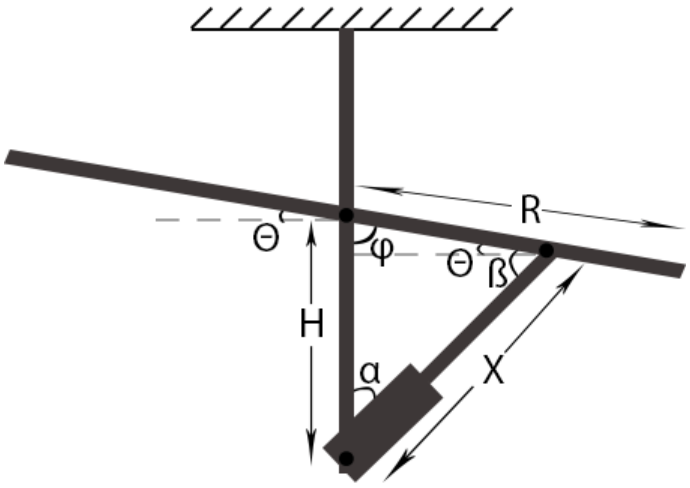


Figure B.1: Actuator geometry

Law of Sines:

$$\frac{sin(\beta)}{H} = \frac{sin(\varphi)}{X} \tag{B.0}$$

Law of Cosines:

$$H^2 = R^2 + X^2 - 2RXcos(\beta) \tag{B.0}$$

Equation B.1.1 and B.1.1:

$$\begin{aligned} sin(\theta) &= \frac{R^2+H^2-X^2}{2RH} \\ \theta &= sin^{-1}(\frac{R^2+H^2-X^2}{2RH}) \end{aligned} \tag{B.0}$$

B.1.2 Boom Inertia Calculations

To be able to estimate the inertia of the boom the boom was split into 2 parts, the front part and the back part. Both parts would have different overall volumes, but their cross-sectional area would be the same, equaling:

$$A_{boom} = A_{solid} - A_t$$

$$A_{solid} = \frac{\pi}{2} r_{boom}^2$$

$$A_t = \frac{\pi}{2}(r_{boom} - t)^2$$

where r_{boom} is the radius of the circular cross-section and t is the thickness of the boom. The volume for each part could then be obtained from:

$$V_{front} = A_{boom} L_{front}$$

$$V_{back} = A_{boom} L_{back}$$

With the total volume, the complete mass for each part could be obtained from:

$$m_{front} = \rho V_{front}$$

$$m_{back} = \rho V_{back}$$

where ρ is the density of the material. The inertia generated by each part was then obtained through:

$$I_{tot} = I_{front} + I_{back}$$

$$I_{front} = \frac{1}{3} m_{front} L_{front}^2 + m_{front}(\frac{L_{front}}{2})^2$$

$$I_{back} = \frac{1}{3} m_{back} L_{back}^2 + m_{back}(\frac{L_{back}}{2})^2$$

The values of the constants can be found in the Table X below

B.2 Control Equations

B.2.1 Velocity PI-controller w. LP-filter

The plant for the motor is represented by:

$$G_p(s) = \frac{B(s)}{A(s)} = \frac{b}{s+a}$$

where b and a are constants containing the motor parameters. The controllers feedback and feed forward parts are represented by:

$$G_{fb}(s) = \frac{S(s)}{R(s)} = \frac{Ps+I}{s(s+r_0)}$$

$$G_{ff}(s) = \frac{T(s)}{R(s)}$$

Desired poles are placed by ω_m, ζ_m and ω_o, which can be found in Table B.1. Solving the Diophantine equation gives the values for the control parameters from:

$$AR + BS = s^3 + (a + r_0)s^2 + (Pb + r_0 a)s + Ib$$

$$A_m A_o = s^3 + (2\zeta_m \omega_m + \omega_o)s^2 + (2\zeta_m \omega_m \omega_o + \omega_m^2)s + \omega_m^2 \omega_o$$

Static DC-gain of 1 for the feed forward part requires:

$$t_0 = \frac{A_m(0)}{B(0)} = \frac{\omega_m^2}{b}$$

$$T(s) = t_0 A_o(s) = \frac{\omega_m^2}{b}(s + \omega_o)$$

All parts then give the closed loop transfer function for the system according to:

$$G_{cl}(s) = \frac{G_{ff}(s)G_p(s)}{1+G_{fb}(s)G_p(s)}$$

Transformation from the continuous transfer function to the discrete transfer function was done using Tustin's method:

$$s = \frac{2}{T_s}\frac{z-1}{z+1}$$

The controller could then be implemented on the MCU for the control signal:

$$u(z) = \frac{T(z)}{R(z)}r - \frac{S(z)}{R(z)}y$$

where the controller was written on the appropriate form to be implemented in C as:

$$U[n] = t_1 R[n] + t_2 R[n-1] + t_3 R[n-2] - s_1 Y[n] - s_2 Y[n-1] - s_[3]Y[n-2] - r_2 U[n-1] - r_3 U[n-2]$$

where $t_1, t_2, t_3, s_1, s_2, s_3, r_2$ and r_3 are the elements of the *S, R* and *T* vectors.

B.2.2 Position PD-controller w. LP-filter

The plant for the motor is represented by:

$$G_p(s) = \frac{B(s)}{A(s)} = \frac{b}{s(s+a)}$$

where b and a are constants containing the motor parameters. The controllers feedback and feed forward parts are represented by:

$$G_{fb}(s) = \frac{S(s)}{R(s)} = \frac{P+Ds}{s+r_0}$$

$$G_{ff}(s) = \frac{T(s)}{R(s)}$$

Desired poles are placed by ω_m, ζ_m and ω_o, which can be found in Table B.1. Solving the Diophantine equation gives the values for the control parameters from:

$$AR + BS = s^3 + (r_0 + a)s^2 + (ar_0 + Db)s + Pb$$

$$A_m A_o = s^3 + (2\zeta_m\omega_m + \omega_o)s^2 + (2\zeta_m\omega_m\omega_o + \omega_m^2)s + \omega_m^2\omega_o$$

Static DC-gain of 1 for the feed forward part requires:

$$t_0 = \frac{A_m(0)}{B(0)} = \frac{\omega_m^2}{b}$$

$$T(s) = t_0 A_o(s) = \frac{\omega_m^2}{b}(s + \omega_o)$$

All parts then give the closed loop transfer function for the system according to:

$$G_{cl}(s) = \frac{G_{ff}(s)G_p(s)}{1+G_{fb}(s)G_p(s)}$$

Transformation from the continuous transfer function to the discrete transfer function was done using Tustin's method:

$$s = \frac{2}{T_s}\frac{z-1}{z+1}$$

The controller could then be implemented on the MCU for the control signal:

$$u(z) = \frac{T(z)}{R(z)}r - \frac{S(z)}{R(z)}y$$

where the controller was written on the appropriate form to be implemented in C as:

$$U[n] = t_1 R[n] + t_2 R[n-1] - s_1 Y[n] - s_2 Y[n-1] - r_2 U[n-1]$$

where t_1, t_2, s_1, s_2 and r_2 are the elements of the S, R and T vectors.

Table B.1: Values of the parameters for the pole placement

Motion	**Controller**	ω_m	ζ_m	ω_o
Luffing	Velocity	10	0.9	50
	Position	10	0.9	50
Slewing	Velocity	3	0.95	100
	Position	0.95	1	100

Appendix C

Test Cases

C.1 Luffing

C.2 Slewing

C.3 Control Panel

C.4 Software

Appendix D

Controller & Processor Code

D.1 Raspberry Pi Code

```
import RPi.GPIO as GPIO
import spidev
import os
import can
# import ADC
import Adafruit_Python_MCP3008
#import can
GPIO.setmode(GPIO.BOARD)

## ADC Configuration for joystick
spi = spidev.SpiDev()
spi.open(1, 0)

xjoych = 3
yjoych = 2
knob_x = 1
knob_y = 0

ratio_x = 1
ratio_y = 1

## GPIO configuration
GPIO.setup(15, GPIO.IN, pull_up_down = GPIO.PUD_UP)        # underground switch configu
GPIO.setup(33, GPIO.IN, pull_up_down = GPIO.PUD_UP)        # home button configuration
GPIO.setup(18, GPIO.IN, pull_up_down = GPIO.PUD_UP)        # control panel function on a
global OnSwitch
```

```
OnSwitch = GPIO.input(18)
GPIO.setup(37, GPIO.IN, pull_up_down = GPIO.PUD_UP)         # reset home position button
GPIO.setup(29, GPIO.IN, pull_up_down = GPIO.PUD_UP)         # changing ratio button confi

## Switch configuration
GPIO.setup(11, GPIO.IN, pull_up_down = GPIO.PUD_UP)
GPIO.setup(13, GPIO.IN, pull_up_down = GPIO.PUD_UP)
GPIO.setup(7, GPIO.IN, pull_up_down = GPIO.PUD_UP)
GPIO.setup(12, GPIO.IN, pull_up_down = GPIO.PUD_UP)

## set GPIO pin 32 as the output
GPIO.setup(32, GPIO.OUT)         # enable driver

## CAN configuration
os.system("sudo /sbin/ip link set can0 up type can bitrate 500000")         # set can spe
bus = can.interface.Bus(channel = 'can0', bustype = 'socketcan_native')

global IsHome
IsHome = int(not(GPIO.input(33)))
global under
under = int(not(GPIO.input(15)))
global ResetHome
ResetHome = int(not(GPIO.input(37)))
global switchx
switchx = GPIO.input(11)
global switchy
switchy = GPIO.input(13)

## reverse and forward mode for xjoystick(pin 11 and 13)
def switchx_callback1(channel):
        global switchx
        print("luffing joystick reverse")
        switchx = 0

def switchx_callback2(channel):
        global switchx
        print("luffing joystick forward")
        switchx = 1
```

```
## reverse and forward mode for yjoystick(pin 7 and 12)
def switchy_callback1(channel):
        global switchy
        print("slewing joystick reverse")
        switchy = 1

def switchy_callback2(channel):
        global switchy
        print("slewing joystick forward")
        switchy = 0

## read ratio when corresponding button pressed (pin 29)
def ratiox_callback(channel):
        global ratio_x
        ratio_x = float(getReading(knob_x))
        ratio_x = ratio_x/1024
        print("knob_x pressed")
        print(ratio_x)

        global ratio_y
        ratio_y = float(getReading(knob_y))
        ratio_y = ratio_y/1024
        print("knob_y pressed")
        print(ratio_y)

## define underground mode (using GPIO 22 --- pin 15)
def callback_IsUnder(channel):
        global under
        under = int(not(GPIO.input(15)))
        print("under = ", under)

## define home button (using GPIO 13 --- pin 33)
def callback_home(channel):
        global IsHome
        IsHome = int(not(GPIO.input(33)))
        print("Home = ", IsHome)

## define reset home button (using pin 37)
def callback_ResetHome(channel):
```

```
        global ResetHome
        IsHome = int(not(GPIO.input(37)))
        print("Reset Home = ", ResetHome)

## define open for the whole control panel
def callback_OnPanel(channel):
        global OnSwitch
        sleep(0.01)                # for bouncetime
        OnSwitch = int(not(GPIO.input(18)))
        GPIO.output(32, OnSwitch)
        print("Function Switch = ", OnSwitch)

## read value from joystick ---- Joystck function
def getReading(channel):
        spi.max_speed_hz = 3150000
        rawData = spi.xfer2([1, (8 + channel) << 4, 0])
        processData = ((rawData[1] & 3) << 8) + rawData[2]
        return processData


## x joystick forward and reverse switch
GPIO.add_event_detect(11, GPIO.FALLING, callback = switchx_callback1, bouncetime = 100)
GPIO.add_event_detect(13, GPIO.FALLING, callback = switchx_callback2, bouncetime = 100)

## y joystick forward and reverse switch
GPIO.add_event_detect(7, GPIO.FALLING, callback = switchy_callback1, bouncetime = 100)
GPIO.add_event_detect(12, GPIO.FALLING, callback = switchy_callback2, bouncetime = 100)

## home button and reset home button
GPIO.add_event_detect(33, GPIO.BOTH, callback = callback_home, bouncetime = 100)
GPIO.add_event_detect(37, GPIO.BOTH, callback = callback_ResetHome, bouncetime = 100)

GPIO.add_event_detect(18, GPIO.BOTH, callback = callback_OnPanel, bouncetime = 400)
GPIO.add_event_detect(15, GPIO.BOTH, callback = callback_IsUnder, bouncetime = 100)

## triggered by two buttons ---- get new ratio for joystick x and y
GPIO.add_event_detect(29, GPIO.FALLING, callback = ratiox_callback, bouncetime = 100)
```

```
while True:
        data_x = float(getReading(xjoych))
        data_y = float(getReading(yjoych))

        ## set threshold to avoid disturbance
        if data_x <= 522 and data_x >= 466:
                data_x = 512
        if data_y <= 522 and data_y >= 466:
                data_y = 512

        ## calculate difference to change reference
        diff_x = (data_x - 512) * ratio_x
        diff_y = (data_y - 512) * ratio_y

        ## chagne value when being in reverse mode
        if switchx == 1:
                int_x = 512 - int(diff_x)
        else:
                int_x = 512 + int(diff_x)

        if switchy == 1:
                int_y = 512 - int(diff_y)
        else:
                int_y = 512 + int(diff_y)

        print('ratio x is:', ratio_x)
        print('data x is:', int_x)
        print('ratio y is:', ratio_y)
        print('data y is:', int_y)

        ## build CAN message for luffing and slewing
        msg_x = [0, 0]
        msg_y = [0, 0]
        msg_x[1] = int_x % 256
        msg_x[0] = int(int_x/ 256)
        msg_y[1] = int_y % 256
        msg_y[0] = int(int_y/ 256)
```

```
msg = can.Message(arbitration_id = 0x7de, data = [msg_x[0], msg_x[1], msg_y[0],
bus.send(msg)         # CAN send message
sleep(1)         # make delay to send CAN message
```

D.2 Arduino Code

```
#include <EEPROM.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdlib.h>
#include <SPI.h>
#include "mcp_can.h"

int eeAddress = 9;
int Data;

// define luff and slew
volatile int luff_total, slew_total;
byte luff[2], slew[2];

const int SPI_CS_PIN = 9;
MCP_CAN CAN(SPI_CS_PIN);

// Use timer1, with an internal clock source, with suitable prescaling, to generate inte
void setup() {

  DDRB |= 0b00110000;       // define PB5 as output
  TCCR1A &= 0b00001100;     // normal port operation and CTC mode enabled
  TCCR1B &= 0b11100000;     // CTC-mode
  TCCR1B |= 0b00001100;     // prescaler = 256
  OCR1A = 625;              // compare value, 16'000'000/256 originally for 1 Hz.   OCR
  TIMSK1 |= 0b00000010;     // enable output compare A match interrupt
  SREG |= 0b10000000;       // global enable
  pinMode(A1, INPUT_PULLUP);

  //initialize serial
  Serial.begin(9600);

  while (!Serial) {}  // wait for serial communication to be initialized
  //send saved data to Cypress
  for (int i = 0; i < 4; i++) {
    Serial.write(EEPROM[eeAddress + i]);
  }
```

```
  // CAN setup
  while (CAN_OK != CAN.begin(CAN_500KBPS)) {} // wait until CAN is setup

}

void loop() {

  while (Serial.available() > 0) {

    Serial.readBytes(luff, 2);
    Serial.readBytes(slew, 2);
    luff_total = luff[0] * 256 + luff[1];
    slew_total = slew[0] * 256 + slew[1];
  }

  unsigned char len = 0;
  unsigned char buf[8];
  int buf2[8] = {0};


  // Can communication
  if (CAN_MSGAVAIL == CAN.checkReceive())           // check if data is coming
  {
    CAN.readMsgBuf(&len, buf);    // read data,  len: data length, buf: data buf
    unsigned long canId = CAN.getCanId();

    for (int i = 0; i < len; i++)
    {
      buf2[i] = (int)buf[i];  // cast buffer to int
      Serial.write(buf2[i]);  // write the CAN message
    }

  }

}


ISR(TIMER1_COMPA_vect) { // check every 10 ms whether power is lost
```

```
  cli();
  if ((analogRead(A1)) < (920)) { // power loss detected (voltage is less than 4.5 V)

    EEPROM.update(eeAddress, luff[0]);
    EEPROM.update(eeAddress + 1, luff[1]);
    EEPROM.update(eeAddress + 2, slew[0]);
    EEPROM.update(eeAddress + 3, slew[1]);
  }
  sei();

}
```

D.3 MATLAB code

```
% File: draft.m
% This program is for controlling the flexible plant
clear, clc

syms Kcrane dcrane n Jm Km Kemf R Jcrane
syms s

dcrane = 200;
n = 71.42*8;% Gearbox transmission
Jm = 0.002189; % Rotor+Gearbox(Motor) inertia
Km = 0.3533; % Motor constant
Kemf = 0.3533; % Motor constant
R = 0.6; % Motor electric resistance

Kcrane = 1*10^4;
Jcrane = 419;

% State space matrices
A = [0 1 0 0;-Kcrane/(n^2*Jm) -((Km*Kemf)/(R*Jm)-dcrane/(n^2*Jm)) Kcrane/(n*Jm) dcrane/(
    0 0 0 1; Kcrane/(n*Jcrane) dcrane/(n*Jcrane) -Kcrane/Jcrane -dcrane/Jcrane];
B = [Km/(R*Jm); 0; 0; 0];
C =[0 0 0 1];
D =0;
% states: angle motor, motor velocity, output angle, velocity of crane
```

```
SYS_SYM = C*(s*eye(4) - A)^-1*B + D;

% Convert symbolic system (SYS_SYM) to a transfer function (SYS1)
[symNum,symDen] = numden(SYS_SYM); %Get num and den of Symbolic TF
TFnum = sym2poly(symNum);    %Convert Symbolic num to polynomial
TFden = sym2poly(symDen);    %Convert Symbolic den to polynomial
SYS1 = tf(TFnum,TFden);

SYS2 = minreal(zpk(SYS1), 1e-2)    % tolerance + minreal lets us cancel the pole/zero pa
[num_tf, den_tf] = tfdata(SYS2);

% convert from cells to vectors, containing coeffeicients
num_coeffs = num_tf{1};
den_coeffs = den_tf{1};

% 1. Controller design
syms s2 s1 s0 r0
s_vector = [s^2; s; 1];
% PID with LP
S = [s2 s1 s0]*s_vector;
R = [1 r0 0]*s_vector;

% 2. C.L. Polynomial
B = num_coeffs*s_vector;
A = den_coeffs*s_vector;

Acl = A*R + B*S;
Aclfac = vpa(collect(Acl,s));
[Acl_coeff] = coeffs(Aclfac,s);

% 3. Select desired C.l. polynomial (4th order here)
syms z1 z2 w1 w2
z1= 0.8;
z2= 0.8;
w1= 20;
w2= 200;

Am= s^2 + 2*z1*w1*s + w1^2;
Ao= s^2 + 2*z2*w2*s + w2^2;
```

```
A_d = (Am)*(Ao);
[Ad_coeff] = coeffs(vpa(collect(A_d,s)),s);

%Diophantine equations
eqn1 = Acl_coeff(1) == Ad_coeff(1);
eqn2 = Acl_coeff(2) == Ad_coeff(2);
eqn3 = Acl_coeff(3) == Ad_coeff(3);
eqn4 = Acl_coeff(4) == Ad_coeff(4);

sol = solve([eqn1,eqn2,eqn3,eqn4],[r0,s0,s1,s2]);
r0eval = eval(sol.r0);
s0eval = eval(sol.s0);
s1eval = eval(sol.s1);
s2eval = eval(sol.s2);

%Auxiliary parameters
DCgain = eval(subs(Am/B, s, 0)); %OK

numTR = [DCgain DCgain*2*z2*w2 DCgain*w2^2];%OK
denTR = [1 r0eval 0]; %OK

numSR = [s2eval s1eval s0eval]; %OK
denSR = [1 r0eval 0]; %OK

TR = tf(numTR,denTR);
SR = tf(numSR,denSR);
BA = SYS2;
[numBA,denBA]=tfdata(BA);



% File: Motor_code.m
clear all, close all, clc;
% Luffing Sims (v.2)

% Variables from datasheet CAHB-10 MOTOR_REV4.pdf
U_in = 24;                  % [V]
% Wire_od = 0.3;            % [NOT SURE]
R_m = 1/0.7;          % [Ohm]
V_m_no_load = 6000;         % [RPM]
```

```
I_no_load = 0.3;            % [A]
% T_rated = 0.5;              % [kgcm]
V_m_rated = 5200;           % [RPM]
I_rated = 1.6;              % [A]
% k_mi = 0.32;               % [kgcm/A]

% Conversion of variables
T_rated = 4.9033e-2;        % [Nm] kgcm -> Nm
K_m = 0.0018;            % [Nm/A] kgcm/A -> Nm/A
Kemf_m = K_m;

% Motor variables to obtain through simulations
J_m = 1/110000;                 % [kgm^2]
Cv_m = 0.00028;             % [Nm/RPM]

Ts = 0.014;

%% Simulink variables

Res = 1/R_m;
Km2Rm = (K_m^2)/R_m;
Jmotor = 1/J_m;
n = 4.9242;
%J_l = 3.2285; % Actual value
J_l = 0.0032285;
J_tot = J_m + (J_l/(n^2));

%% Velocity controller

b = K_m/(J_tot*R_m);
a = (Cv_m*R_m + K_m^2)/(J_tot*R_m);

omega_mv = 10;
omega_ov = 50;
zeta_mv = 0.9;

[Bv,Av,Sv,Rv,Tv] = motor_velocity_controller(omega_mv,zeta_mv,omega_ov,b,a);      % Velo

Gpv = Bv/Av;
```

```
Gffv = Tv/Rv;
Gfbv = Sv/Rv;

Gclv = Gffv*Gpv/(1 + Gpv*Gfbv);

Gpdv = c2d(Gpv,Ts,'Tustin');
Gffdv = c2d(Gffv,Ts,'Tustin');
Gfbdv = c2d(Gfbv,Ts,'Tustin');

Gcldv = Gffdv*Gpdv/(1 + Gpdv*Gfbdv);

%% Position controller

omega_mp = 10;
omega_op = 50;
zeta_mp = 0.9;

[Bp,Ap,Sp,Rp,Tp] = motor_position_controller(omega_mp,zeta_mp,omega_op,b,a);

Gpp = Bp/Ap;
Gffp = Tp/Rp;
Gfbp = Sp/Rp;

Gclp = Gffp*Gpp/(1 + Gpp*Gfbp);

Gpdp = c2d(Gpp,Ts,'Tustin');
Gffdp = c2d(Gffp,Ts,'Tustin');
Gfbdp = c2d(Gfbp,Ts,'Tustin');

Gcldp = Gffdp*Gpdp/(1 + Gpdp*Gfbdp);

figure(1)
subplot(2,1,1)
step(Gclp)
subplot(2,1,2)
pzmap(Gclp)

figure(2)
subplot(2,1,1)
```

```
step(Gclv)
subplot(2,1,2)
pzmap(Gclv)

%% Discrete test
refp = 20;
refv = 10;

[plotvolp,plotpos,plotvolv,plotvel,t] = controller_test(Gffdp,Gfbdp,Gpdp,Gffdv,Gfbdv,Gpd

figure(3)
subplot(2,1,1)
plot(t,plotvolp)
title('Voltage')
subplot(2,1,2)
plot(t,plotpos)
title('Position')

figure(4)
subplot(2,1,1)
plot(t,plotvolv)
title('Voltage')
subplot(2,1,2)
plot(t,plotvel)
title('Velocity')


% File: motor_velocity_controller.m

function [B,A,S,R,T] = motor_velocity_controller(omega_m,zeta_m,omega_o,b,a)

s = tf('s');

r0 = 2*zeta_m*omega_m + omega_o - a;
P = (2*zeta_m*omega_m*omega_o + omega_m^2 - r0*a)/b;
I = (omega_o*omega_m^2)/b;

t0 = (omega_m^2)/b;
```

```
B = b;
A = s+a;
S = P*s + I;
R = s*(s+r0);
T = t0*(s+omega_o);

end


% File: motor_position_controller.m

function [Bp,Ap,Sp,Rp,Tp] = motor_position_controller(omega_mp,zeta_mp,omega_op,b,a)

s = tf('s');
r0 = 2*zeta_mp*omega_mp + omega_op - a;
D = (2*zeta_mp*omega_mp*omega_op + omega_mp^2 - a*r0)/b;
P = (omega_op*omega_mp^2)/b;

t0 = (omega_mp^2)/b;

Bp = b;
Ap = s*(s+a);
Sp = P + D*s;
Rp = s + r0;
Tp = t0*(s+omega_op);

end


% File: Actuator_placement.m
clear all, close all, clc;

Xmin = 320e-3; % Preferable at 30 degrees
Xmax = 520e-3;
Fmax = 120;
Xinterval = [Xmin:1e-4:Xmax];

%% Obtaining values of R and H for fixed actuator length Xmin
```

```
a = [0:1e-2:120];
for i = 1:length(a)
    R(i) = Xmin*sind(a(i))/sind(60);
end
for j = 1:length(a)
    b(j) = max(a)-a(j);
    H(j) = sind(b(j))*R(j)/sind(a(j));
end

figure(1)
subplot(1,3,1)
plot(b,R,'b--',b,H,'r')
legend('R','H')
xlabel('Angle beta [deg]')
ylabel('Linkage lengths [m]')


%% Checking intervals of theta for possible angles of a and b.

for k = 1:length(R)
    thetamax(k) = asind(((R(k)^2) + (H(k)^2) - (Xmax^2))/(2*R(k)*H(k)));
    if thetamax(k) > -90
        thetaminvec(k) = asind(((R(k)^2) + (H(k)^2) - (Xmin^2))/(2*R(k)*H(k)));
        thetamaxvec(k) = thetamax(k);
        Mvec(k) = sind(b(k))*R(k)*Fmax;
        avec(k) = a(k);
        bvec(k) = b(k);
        Rvec(k) = R(k);
        Hvec(k) = H(k);
        Minanglevec(k) = -30;
    end
end

thetaminvec = thetaminvec(thetaminvec ~= 0);
thetamaxvec = thetamaxvec(thetamaxvec ~= 0);
avec = avec(avec ~= 0);
bvec = bvec(bvec ~= 0);
Rvec = Rvec(Rvec ~= 0);
Hvec = Hvec(Hvec ~= 0);
```

```
Mvec = Mvec(Mvec ~= 0);
Minanglevec = Minanglevec(Minanglevec ~= 0);

subplot(1,3,2)
plot(Rvec,thetamaxvec, 'k',Rvec,thetaminvec,'r',Rvec,Minanglevec,'m');
ylabel('Angle Theta [deg]')
hold on
yyaxis right
plot(Rvec,Mvec)
xlabel('Length R [m]')
ylabel('Torque [Nm]')

%% Observing span of theta for fixed R and H and varying actuator length X

Rfixed = 0.2298;
Hfixed = 0.3655;

theta = asind(Rfixed^2 + Hfixed^2 - Xinterval.^2)/(2*Rfixed*Hfixed);

subplot(1,3,3)
plot(Xinterval,theta)
title('R = 0.2298m, H = 0.3655m')
ylabel('Angle theta [deg]')
xlabel('Actuator length [m]')

%% Inertia check

% Inertia of the boom

Lfront = 2.1-0.273;
Lback = 0.273;
rboom = (0.11/2);
t = 0.005;
Asolid = (pi/2)*rboom^2;
At = (pi/2)*((rboom-t)^2);
Aboom = Asolid - At;
Vfront = Aboom*Lfront;
Vback = Aboom*Lback;
Vboom = Vfront + Vback;
```

```
% For PVC
rhoPVC = 1.4e3;
mPVCfront = rhoPVC*Vfront;
mPVCback = rhoPVC*Vback;
mPVC = rhoPVC*Vboom;
IPVC1 = (1/3)*(mPVCfront*(Lfront^2)) + mPVCfront*((Lfront/2)^2);
IPVC2 = (1/3)*(mPVCback*(Lback^2)) + mPVCback*((Lback/2)^2);
IPVC = IPVC1+IPVC2;
```

D.4 Cypress Code

```
/*******************************************************************************
* \file             main.c
*
* \version          1.0
*
* \brief            Motors Controller: Luffing and Slewing.
*
* \Author           Little Bergmans
********************************************************************************
* \copyright
* Copyright 2016, Cypress Semiconductor Corporation. All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
* CYPRESS PROVIDES THIS SOFTWARE "AS IS" AND MAKES NO WARRANTY
* OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS SOFTWARE,
* INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
* MERCHANTABILITY, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR
* PURPOSE.
*******************************************************************************/
#include "mcu.h"
#include "pdl_header.h"

/* OTHERS: Variable */
uint32_t Counter_Terminal = 0; // Test functionality
int Compensation_Vel_Slew = 0;

/*UART CODE*/
volatile stc_mfsn_uart_t* UartCh12 = &UART12; // Needed to be globally, use in both Uart
int au8UartRxBuf[100];
int au4UartRxBuf[100];
int encoder_value[100];
uint8_t u8Cnt = 0;
uint8_t u4Cnt = 0;
uint8_t u4Cnt_main = 0;
signed int home_mode = 0;                       //Initial Initialization for home
signed int underg_mode = 0;                     //Initial Initialization for underground
```

```
boolean_t arduino_sent = FALSE;
boolean_t arduino_sent_CAN = FALSE;

/*CAN message interpretation*/
uint16_t Ref_Slew_2_Byte = 0;
uint16_t Ref_Luff_2_Byte = 0;
uint16_t Ref_Slew_2_Byte_Pos = 0;
uint16_t Ref_Luff_2_Byte_Pos = 0;
float CAN_RATIO_m_Slew = 290/512.0; // Supposing that 2800 rpm corresponds to 1024 and -
float CAN_RATIO_n_Slew = -290;
//float CAN_RATIO_m_Luff = 544/512.0; // Supposing that 2800 rpm corresponds to 1024 and
//float CAN_RATIO_n_Luff = -544;
float CAN_RATIO_m_Luff = 12/512.0; // Supposing that 2800 rpm corresponds to 1024 and -2
float CAN_RATIO_n_Luff = -12;
float CAN_RATIO_m_Slew_Pos = 3768/512.0; // Supposing that 2800 rpm corresponds to 1024
float CAN_RATIO_n_Slew_Pos = -3768;

/* BT8 PWM out parameter definition Slewing */
#define PWM_CYCLE_VALUE_100_DUTY               1200u
#define PWM_CYCLE_VALUE_90_DUTY                1080u
#define PWM_CYCLE_VALUE_50_DUTY                 600u
#define PWM_CYCLE_VALUE_10_DUTY                 120u
#define PWM_CYCLE_VALUE_0_DUTY                    0u
#define PWM_CYCLE_VALUE_12V_DUTY                840u
#define PWM_CYCLE_VALUE_MINUS_12V_DUTY          360u

/* BT2 parameter definition Slewing */
#define PROCESSOR_FREQ                         200E6
#define BT2_CYCLE_VALUE                         600u // 100u is a good value for Slewing

/* BT1 parameter definition Luffing */
#define BT1_CYCLE_VALUE                         600u //615u(0.0035) too fast for the lu

/* BT0 parameter definition Slewing Position */
#define BT0_CYCLE_VALUE                         20000u

/* BT3 parameter definition Luffing Position */
#define BT3_CYCLE_VALUE                         615u
```

```
/* Voltages */
#define MAX_VOLTAGE                              24u
#define MIN_VOLTAGE                             -24u

/* Slewing motor values */
#define MIN_TO_SEC                               60
#define REV_TO_DEGREES                           360
float REV_TO_RAD_S = 0.105;
#define PULSES_PER_REV_SLEW                      512
float PULSES_PER_REV_SLEW_FLOAT = 512;
#define GEAR_RATIO_SLEW                          100
#define MAX_SPEED_RPM_SLEW_CLOCKWISE            2800u  // BEFORE GEAR BOX
#define MAX_SPEED_RPM_SLEW_ANTICLOCKWISE       -2800u

/* Luffing motor values */
#define PULSES_PER_REV_LUFF                       88 // 48 rad the whole length (e.i. 24
float PULSES_PER_REV_LUFF_FLOAT = 88;
#define GEAR_RATIO_LUFF                          100 // To be modified
#define MAX_SPEED_RPM_LUFF_CLOCKWISE            5600u
#define MAX_SPEED_RPM_LUFF_ANTICLOCKWISE       -5600u

/* Lookup Table Logic */
uint32_t u32ExtInt1Count = 0, u32ExtInt2Count = 0, u32ExtInt3Count = 0, u32ExtInt4Count
static int8_t lookup_table[] = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};
boolean_t High_enc_A_Slew = FALSE;
boolean_t Low_enc_A_Slew = TRUE;
boolean_t High_enc_B_Slew = FALSE;
boolean_t Low_enc_B_Slew = TRUE;
boolean_t High_enc_A_Luff = FALSE;
boolean_t Low_enc_A_Luff = TRUE;
boolean_t High_enc_B_Luff = FALSE;
boolean_t Low_enc_B_Luff = TRUE;

/* Encoder Counting*/
static uint8_t Selection_Slew = 0;
static uint8_t Selection_Old_Slew = 0;
static uint8_t lookup_table_index_Slew = 0;

static uint8_t Selection_Luff = 0;
```

```
static uint8_t Selection_Old_Luff = 0;
static uint8_t lookup_table_index_Luff = 0;

/*Velocity Calculation*/

signed int Enc_value_Slew = 0;
signed int Enc_value_Slew_Old = 0;
//volatile float Vel_Slew = 0.0;
float Time_BT2 = 0.0;
static uint32_t PRESCALE_BT2 = 1;

signed int Enc_value_Luff = 0;
signed int Enc_value_Luff_Old = 0;
//float Vel_Luff = 0.0;
float Vel_Luff_Rod = 0.0;
float Time_BT1 = 0.0;
static uint32_t PRESCALE_BT1 = 1;

/*Position Calculation*/
float Pos_Slew = 0.0;
float Time_BT0 = 0.0;
//static uint32_t PRESCALE_BT0 = 1;
boolean_t Position_Control_Slew = FALSE;

float Pos_Luff = 0.0;
float Time_BT3 = 0.0;
//static uint32_t PRESCALE_BT3 = 1;
boolean_t Position_Control_Luff = FALSE;

/*Setting PWM Slewing*/
uint16_t Duty_Slew = 0;
uint16_t Duty_Slew_Pos = 0;
int Duty_Scan_Slew = 0;
//float RELATIONSHIP = (1080 - 120)/100.0;
//static float CYCLE_VALUE_RANGE_SLEW = PWM_CYCLE_VALUE_90_DUTY - PWM_CYCLE_VALUE_50_DU
float DUTY_VOL_RATIO_m = 20; // 24/(1080 - 600) -> 24 V corresponding to 1080 (90%) and
float DUTY_VOL_RATIO_n = 600;
//float DUTY_FINAL = 0;
```

```
/*Setting PWM Luffing*/
uint16_t Duty_Luff = 0;
uint16_t Duty_Luff_Pos = 0;
int Duty_Scan_Luff = 0;

/*Control Law Slewing Velocity*/
static float Vol_Slew = 0.0;
static float Vol_Slew_Past = 0.0;
static float Vol_Slew_Past_Past = 0.0;
static float Ref_Slew = 0.0;
static float Ref_Slew_Past = 0.0;
static float Ref_Slew_Past_Past = 0.0;
static float Out_Vel_Slew = 0.0;
static float Out_Vel_Slew_Past = 0.0;
static float Out_Vel_Slew_Past_Past = 0.0;
/* T(z)/R(z) = (a_1_Slew*z^2 + a_2_Slew*z + a_3_Slew)/(b_1_Slew*z^2 + b_2_Slew*z + b_3_S

static float a_1_Slew = 0.04678;
static float a_2_Slew = -0.0735;
static float a_3_Slew = 0.02905;
static float b_1_Slew = 1.0;
static float b_2_Slew = -1.62;
static float b_3_Slew = 0.6203;
static float c_1_Slew = 0.1119;
static float c_2_Slew = 0.001167;
static float c_3_Slew = -0.1107;

/*Control Law Luffing Velocity*/
static float Vol_Luff = 0.0;
static float Vol_Luff_Past = 0.0;
static float Vol_Luff_Past_Past = 0.0;
static float Ref_Luff = 0.0;
static float Ref_Luff_Past = 0.0;
static float Ref_Luff_Past_Past = 0.0;
static float Out_Vel_Luff = 0.0;
static float Out_Vel_Luff_Past = 0.0;
static float Out_Vel_Luff_Past_Past = 0.0;
/* T(z)/R(z) = (a_1_Slew*z^2 + a_2_Slew*z + a_3_Slew)/(b_1_Slew*z^2 + b_2_Slew*z + b_3_S
static float a_1_Luff = 0.07296;
```

```
static float a_2_Luff = 0.03783;
static float a_3_Luff = -0.03513;
static float b_1_Luff = 1.0;
static float b_2_Luff = -1.368;
static float b_3_Luff = 0.3679;
static float c_1_Luff = 0.4886;
static float c_2_Luff = 0.03783;
static float c_3_Luff = -0.4507;

/*Control Law Slewing Position*/
static float Vol_Slew_Pos = 0.0;
static float Vol_Slew_Pos_Past = 0.0;
static float Ref_Slew_Pos = 0.0;
static float Ref_Slew_Pos_Past = 0.0;
static float Out_Pos_Slew = 0.0;
static float Out_Pos_Slew_Past = 0.0;
/* T(z)/R(z) = (a_1_Slew*z^2 + a_2_Slew*z + a_3_Slew)/(b_1_Slew*z^2 + b_2_Slew*z + b_3_S
static float a_1_Slew_Pos = 0.02309;
static float a_2_Slew_Pos = -0.01621;
static float b_1_Slew_Pos = 1.0;
static float b_2_Slew_Pos = -0.7056;
static float c_1_Slew_Pos = -2.553;
static float c_2_Slew_Pos = 2.56;

/*Control Law Luffing Position*/
static float Vol_Luff_Pos = 0.0;
static float Vol_Luff_Pos_Past = 0.0;
static float Ref_Luff_Pos = 0.0;
static float Ref_Luff_Pos_Past = 0.0;
static float Out_Pos_Luff = 0.0;
static float Out_Pos_Luff_Past = 0.0;
/* T(z)/R(z) = (a_1_Slew*z^2 + a_2_Slew*z + a_3_Slew)/(b_1_Slew*z^2 + b_2_Slew*z + b_3_S
static float a_1_Luff_Pos = 10.42;
static float a_2_Luff_Pos = -5.019;
static float b_1_Luff_Pos = 1.0;
static float b_2_Luff_Pos = -0.3679;
static float c_1_Luff_Pos = 69.8;
static float c_2_Luff_Pos = -64.39;
/**
```

```
/*****************************************************************************
** \Counter function Slewing
*****************************************************************************/

void Counter_Slew(void) // Being AB, e.i. if High A and Low B = 0b10
{
  if (High_enc_A_Slew){
    if(High_enc_B_Slew){ // High A and High B
      Selection_Slew = 3;
    }
    else{                // High A and Low B
      Selection_Slew = 2;
    }
  }
  else{
    if(High_enc_B_Slew){ // Low A and High B
      Selection_Slew = 1;
    }
    else{                // Low A and Low B
      Selection_Slew = 0;
    }
  }


  switch (Selection_Slew) { // Being A_oldB_oldAB,e.i. if High_old A and Low_old B High
    case 0u:
              switch (Selection_Old_Slew) { // Being A_oldB_oldAB,e.i. if High_old A and
                case 0u:  //0b0000
                  lookup_table_index_Slew = 0;
                  break;

                case 1u:  //0b0100
                  lookup_table_index_Slew = 4;
                  break;

                case 2u:  //0b1000
                  lookup_table_index_Slew = 8;
                  break;
```

```
            case 3u:  //0b1100
              lookup_table_index_Slew = 12;
              break;
          }

  break;

case 1u:
          switch (Selection_Old_Slew) { // Being A_oldB_oldAB,e.i. if High_old A and
            case 0u:  //0b0001
              lookup_table_index_Slew = 1;
              break;

            case 1u:  //0b0101
              lookup_table_index_Slew = 5;
              break;

            case 2u:  //0b1001
              lookup_table_index_Slew = 9;
              break;

            case 3u:  //0b1101
              lookup_table_index_Slew = 13;
              break;
          }
  break;

case 2u:
          switch (Selection_Old_Slew) { // Being A_oldB_oldAB,e.i. if High_old A and
            case 0u:  //0b0010
              lookup_table_index_Slew = 2;
              break;

            case 1u:  //0b0110
              lookup_table_index_Slew = 6;
              break;

            case 2u:  //0b1010
              lookup_table_index_Slew = 10;
```

```
                break;

              case 3u:  //0b1110
                lookup_table_index_Slew = 14;
                break;
            }
    break;

  case 3u:
            switch (Selection_Old_Slew) { // Being A_oldB_oldAB,e.i. if High_old A and
              case 0u:  //0b0011
                lookup_table_index_Slew = 3;
                break;

              case 1u:  //0b0111
                lookup_table_index_Slew = 7;
                break;

              case 2u:  //0b1011
                lookup_table_index_Slew = 11;
                break;

              case 3u:  //0b1111
                lookup_table_index_Slew = 15;
                break;
            }
    break;
}

Enc_value_Slew = Enc_value_Slew + lookup_table[lookup_table_index_Slew];

if (Enc_value_Slew == PULSES_PER_REV_SLEW*4*600){ // 4 encoder is a pulse and the outp
  Enc_value_Slew = 0;                             // This will reset Encoder once a re
}
else if (Enc_value_Slew == -PULSES_PER_REV_SLEW*4*600){
  Enc_value_Slew = 0;                             // This will reset Encoder once a re
}
Selection_Old_Slew = Selection_Slew;
```

```
}

/**
 ******************************************************************************
 ** \Counter function Luffing
 ******************************************************************************/

void Counter_Luff(void) // Being AB, e.i. if High A and Low B = 0b10
{
  if (High_enc_A_Luff){
    if(High_enc_B_Luff){ // High A and High B
      Selection_Luff = 3;
    }
    else{                // High A and Low B
      Selection_Luff = 2;
    }
  }
  else{
    if(High_enc_B_Luff){ // Low A and High B
      Selection_Luff = 1;
    }
    else{                // Low A and Low B
      Selection_Luff = 0;
    }
  }


  switch (Selection_Luff) { // Being A_oldB_oldAB,e.i. if High_old A and Low_old B High
    case 0u:
              switch (Selection_Old_Luff) { // Being A_oldB_oldAB,e.i. if High_old A and
                case 0u:  //0b0000
                  lookup_table_index_Luff = 0;
                  break;

                case 1u:  //0b0100
                  lookup_table_index_Luff = 4;
                  break;

                case 2u:  //0b1000
```

```
            lookup_table_index_Luff = 8;
            break;

          case 3u:  //0b1100
            lookup_table_index_Luff = 12;
            break;
        }

  break;

case 1u:
        switch (Selection_Old_Luff) { // Being A_oldB_oldAB,e.i. if High_old A and
          case 0u:  //0b0001
            lookup_table_index_Luff = 1;
            break;

          case 1u:  //0b0101
            lookup_table_index_Luff = 5;
            break;

          case 2u:  //0b1001
            lookup_table_index_Luff = 9;
            break;

          case 3u:  //0b1101
            lookup_table_index_Luff = 13;
            break;
        }
  break;

case 2u:
        switch (Selection_Old_Luff) { // Being A_oldB_oldAB,e.i. if High_old A and
          case 0u:  //0b0010
            lookup_table_index_Luff = 2;
            break;

          case 1u:  //0b0110
            lookup_table_index_Luff = 6;
            break;
```

```
                case 2u:  //0b1010
                  lookup_table_index_Luff = 10;
                  break;

                case 3u:  //0b1110
                  lookup_table_index_Luff = 14;
                  break;
              }
      break;

    case 3u:
              switch (Selection_Old_Luff) { // Being A_oldB_oldAB,e.i. if High_old A and
                case 0u:  //0b0011
                  lookup_table_index_Luff = 3;
                  break;

                case 1u:  //0b0111
                  lookup_table_index_Luff = 7;
                  break;

                case 2u:  //0b1011
                  lookup_table_index_Luff = 11;
                  break;

                case 3u:  //0b1111
                  lookup_table_index_Luff = 15;
                  break;
              }
      break;
  }

  Enc_value_Luff = Enc_value_Luff + lookup_table[lookup_table_index_Luff];
  Selection_Old_Luff = Selection_Luff;

}


/**
```

```
 ******************************************************************************
 ** \Counter functions for extint
 ******************************************************************************/
void Main_ExtInt1Callback(void) /* Pin Function: Encoder A Slewing Rising */
{
    u32ExtInt1Count++;

    High_enc_A_Slew = TRUE;
    Low_enc_A_Slew = FALSE;

    Counter_Slew();

}

void Main_ExtInt2Callback(void) /* Pin Function: Encoder A Slewing Falling */
{
    u32ExtInt2Count++;

    High_enc_A_Slew = FALSE;
    Low_enc_A_Slew = TRUE;

    Counter_Slew();
}

void Main_ExtInt3Callback(void) /* Pin Function: Encoder B Slewing Rising */
{
    u32ExtInt3Count++;

    High_enc_B_Slew = TRUE;
    Low_enc_B_Slew = FALSE;

    Counter_Slew();
}

void Main_ExtInt4Callback(void) /* Pin Function: Encoder B Slewing Falling */
{
    u32ExtInt4Count++;

    High_enc_B_Slew = FALSE;
```

```
    Low_enc_B_Slew = TRUE;

    Counter_Slew();
}

void Main_ExtInt6Callback(void) /* Pin Function: Encoder A Luffing Rising */
{
    u32ExtInt6Count++;

    High_enc_A_Luff = TRUE;
    Low_enc_A_Luff = FALSE;

    Counter_Luff();
}


void Main_ExtInt20Callback(void) /* Pin Function: Encoder A Luffing Falling */
{
    u32ExtInt20Count++;

    High_enc_A_Luff = FALSE;
    Low_enc_A_Luff = TRUE;

    Counter_Luff();
}

void Main_ExtInt22Callback(void) /* Pin Function: Encoder B Luffing Rising */
{
    u32ExtInt22Count++;

    High_enc_B_Luff = TRUE;
    Low_enc_B_Luff = FALSE;

    Counter_Luff();
}

void Main_ExtInt28Callback(void) /* Pin Function: Encoder B Luffing Falling */
{
    u32ExtInt28Count++;
```

```
    High_enc_B_Luff = FALSE;
    Low_enc_B_Luff = TRUE;

    Counter_Luff();
}

/**
 ****************************************************************************
 ** \Saturation -24 to 24
 ****************************************************************************/
float Saturation_24 (float Voltage){ // Avoid going above 24 V or -24 V
  if (Voltage > 24){
      Voltage = 24;
    }
    else if (Voltage < -24){
      Voltage = -24;
    }
  return Voltage;
}

/**
 ****************************************************************************
 ** \Saturation -12 to 12
 ****************************************************************************/
float Saturation_12 (float Voltage){ // Avoid going above 12 V or -12 V
  if (Voltage > 12){
      Voltage = 12;
    }
    else if (Voltage < -12){
      Voltage = -12;
    }
  return Voltage;
}

/**
 ****************************************************************************
 ** \Saturation 10% to 90% Slewing
 ****************************************************************************/
```

```
uint16_t Saturation_Duty_Slew (uint16_t DUTY){ // Saturate the PWM duty cicle between 10
    if (DUTY > PWM_CYCLE_VALUE_90_DUTY - 200){ // Slewing is drawing/pulling more curren
      DUTY = PWM_CYCLE_VALUE_90_DUTY - 200;
    }

    else if (DUTY < PWM_CYCLE_VALUE_10_DUTY + 50){
      DUTY = PWM_CYCLE_VALUE_10_DUTY + 50;
    }
  return DUTY;
}

/**
 ****************************************************************************
 ** \Saturation -12V to 12V Luffing
 ****************************************************************************/
uint16_t Saturation_Duty_Luff (uint16_t DUTY){ // Saturate the PWM duty cicle between 10
    if (DUTY > PWM_CYCLE_VALUE_12V_DUTY - 100){
      DUTY = PWM_CYCLE_VALUE_12V_DUTY - 100;
    }

    else if (DUTY < PWM_CYCLE_VALUE_MINUS_12V_DUTY + 100){
      DUTY = PWM_CYCLE_VALUE_MINUS_12V_DUTY + 100;
    }
  return DUTY;
}
/**
 ****************************************************************************
 ** \Slewing PWM duty function
 ****************************************************************************/

void setPWM_Slew ( uint16_t DUTY) // Define duty cycle output for Slewing
  {
    // Relationship between Duty cicle and Duty cycle understand by Cypress Duty_Cycle =
    // Although, for the driver that we are using 100% duty is 90% and 0% is 10%. There
    DUTY = Saturation_Duty_Slew(DUTY);
    Bt_Pwm_WriteDutyVal(&BT8, DUTY); // Output duty value
  }

/**
```

```
 ******************************************************************************
 ** \Luffing PWM duty function
 ******************************************************************************/

void setPWM_Luff ( uint16_t DUTY) // Define duty cycle output for Luffing
  {
    // Relationship between Duty cicle and Duty cicle understand by Cypress Duty_Cycle =
    // Although, for the driver that we are using 100% duty is 90% and 0% is 10%. Therei
    DUTY = Saturation_Duty_Luff(DUTY);
    if (underg_mode == 0){ // Underground mode for disabling max and min angle
        // Luffing position calculation
        Out_Pos_Luff = ((Enc_value_Luff)/4)*1/PULSES_PER_REV_LUFF_FLOAT; //1 pulse per 4
        Out_Pos_Luff = Out_Pos_Luff*6.28; //Rev to rad

        if (Out_Pos_Luff <= -10 && DUTY < PWM_CYCLE_VALUE_50_DUTY){ // If in position -1
          DUTY = PWM_CYCLE_VALUE_50_DUTY;
        }
    }
    Bt_Pwm_WriteDutyVal(&BT9, DUTY);
  }

/**
 ******************************************************************************
 ** \Enabling driver ESCON PWM duty function
 ******************************************************************************/

void setPWM_driver_enabling ( uint16_t DUTY) // Define duty cycle output for enabling dr
  {
    DUTY = Saturation_Duty_Slew(DUTY);
    Bt_Pwm_WriteDutyVal(&BT6, DUTY);
  }

/**
 ******************************************************************************
 ** \Read CAN message from UART Arduino
 ******************************************************************************/

void Readmsg(void)
  {
```

```
while(u8Cnt < 8){ // Do this four times (For slewing and luffing velocity purpose)
      while(TRUE != Mfs_Uart_GetStatus(UartCh12, UartRxFull)); /* wait until RX bufi
      au8UartRxBuf[u8Cnt] = Mfs_Uart_ReceiveData(UartCh12); // Give the first in the
      //printf ("u8Cnt: %d \n", u8Cnt);
      u8Cnt++;
    }

if(u8Cnt >= 8){ // Just 8 values each time.
        u8Cnt = 0;
      }

Ref_Slew_2_Byte = 256*au8UartRxBuf[2] + au8UartRxBuf[3];
Ref_Luff_2_Byte = 256*au8UartRxBuf[0] + au8UartRxBuf[1];

Ref_Slew = CAN_RATIO_m_Slew*Ref_Slew_2_Byte + CAN_RATIO_n_Slew; // y = mx + n where
Ref_Luff = CAN_RATIO_m_Luff*Ref_Luff_2_Byte + CAN_RATIO_n_Luff; // y = mx + n where

home_mode = au8UartRxBuf[4]; //Variable for home mode
if (home_mode == 1){ // If home mode implement control position instead
  Position_Control_Slew = TRUE;
  Position_Control_Luff = TRUE;
  // Define home position as position 0
  Ref_Slew_Pos = 0;
  Ref_Luff_Pos = 0;
}
else{ // Else keep doing velocity control
  Position_Control_Slew = FALSE;
  Position_Control_Luff = FALSE;
}

underg_mode = au8UartRxBuf[5]; // Variable for underground mode


printf("Buf0: %d//Buf1: %d//Buf2: %d//Buf3: %d// Buf4: %d//Buf5: %d//Buf6: %d//Buf7:
//printf("//Out Slew Vel: %0.2f//Slew Duty: %0.2f//Voltage: %0.2f//Ref Vel: %0.2f//F
//printf("//Out Luff Vel: %0.2f//Luff Duty: %0.2f//Voltage: %0.2f//Ref Vel: %0.2f//F
//printf("//Out Luff Pos: %0.2f//Luff Duty: %0.2f//Voltage: %0.2f//Ref Pos: %0.2f//F
//printf("//Out Slew Pos: %0.2f//Slew Duty: %0.2f//Voltage: %0.2f//Ref Pos: %0.2f//F
//printf("//VALUE: %d\n", Enc_value_Slew);
```

```
  }

/**
 ****************************************************************************
 ** \Slewing Control Law Velocity
 ****************************************************************************/

void Control_Slew(void)
  {

    if (Ref_Slew == 0){ // The motor has a deviation; it moves faster in one way. Applyi
      if (Compensation_Vel_Slew == 1){
        Ref_Slew = 1;
        Compensation_Vel_Slew = 0;
      }
      else {
        Ref_Slew = -1.8;
        Compensation_Vel_Slew = 1;
      }
    }

    Vol_Slew = (a_1_Slew*Ref_Slew + a_2_Slew*Ref_Slew_Past + a_3_Slew*Ref_Slew_Past_Past
    Vol_Slew = Saturation_24(Vol_Slew);
    Duty_Slew = DUTY_VOL_RATIO_m*Vol_Slew + DUTY_VOL_RATIO_n;

    setPWM_Slew(Duty_Slew);

    /*Update values*/
    Vol_Slew_Past_Past = Vol_Slew_Past;
    Vol_Slew_Past = Vol_Slew;
    Ref_Slew_Past_Past = Ref_Slew_Past;
    Ref_Slew_Past = Ref_Slew;
    Out_Vel_Slew_Past_Past = Out_Vel_Slew_Past;
    Out_Vel_Slew_Past = Out_Vel_Slew;
  }

/**
 ****************************************************************************
 ** \Luffing Control Law Velocity
```

```
 ******************************************************************************/

void Control_Luff(void)
  {
    Vol_Luff = (a_1_Luff*Ref_Luff + a_2_Luff*Ref_Luff_Past + a_3_Luff*Ref_Luff_Past_Past
    Vol_Luff = Saturation_12(Vol_Luff); // Velocity is constant from 12 V to above; only
    Vol_Luff = - Vol_Luff; // Inverted connection of encoder signals
    Vol_Luff = Ref_Luff; // Bypass control law
    //Duty_Luff = DUTY_SPEED_RATIO_LUFF_m*Out_Vel_Luff + DUTY_SPEED_RATIO_LUFF_n; // Dut
    Duty_Luff = DUTY_VOL_RATIO_m*Vol_Luff + DUTY_VOL_RATIO_n; // Duty for the bypass con
    setPWM_Luff(Duty_Luff);

    /*Update values*/
    Vol_Luff_Past_Past = Vol_Luff_Past;
    Vol_Luff_Past = Vol_Luff;
    Ref_Luff_Past_Past = Ref_Luff_Past;
    Ref_Luff_Past = Ref_Luff;
    Out_Vel_Luff_Past_Past = Out_Vel_Luff_Past;
    Out_Vel_Luff_Past = Out_Vel_Luff;
  }


/**
 ******************************************************************************
 ** \Slewing Control Law Position
 ******************************************************************************/

void Control_Slew_Pos(void)
  {
    Vol_Slew_Pos = (a_1_Slew_Pos*Ref_Slew_Pos + a_2_Slew_Pos*Ref_Slew_Pos_Past - c_1_Sle
    Vol_Slew_Pos = Saturation_24(Vol_Slew_Pos);
    Duty_Slew_Pos = DUTY_VOL_RATIO_m*Vol_Slew_Pos + DUTY_VOL_RATIO_n;

    setPWM_Slew(Duty_Slew_Pos);

    /*Update values*/
    Vol_Slew_Pos_Past = Vol_Slew_Pos;
    Ref_Slew_Pos_Past = Ref_Slew_Pos;
    Out_Pos_Slew_Past = Out_Pos_Slew;
```

```
  }

/**
 ******************************************************************************
 ** \Luffing Control Law Position
 ******************************************************************************/

void Control_Luff_Pos(void)
  {
    Vol_Luff_Pos = (a_1_Luff_Pos*Ref_Luff_Pos + a_2_Luff_Pos*Ref_Luff_Pos_Past - c_1_Luf
    Vol_Luff_Pos = Saturation_24(Vol_Luff_Pos);

    Duty_Luff_Pos = DUTY_VOL_RATIO_m*Vol_Luff_Pos + DUTY_VOL_RATIO_n;

    setPWM_Luff(Duty_Luff_Pos);

    /*Update values*/
    Vol_Luff_Pos_Past = Vol_Luff_Pos;
    Ref_Luff_Pos_Past = Ref_Luff_Pos;
    Out_Pos_Luff_Past = Out_Pos_Luff;
  }


/**
 ******************************************************************************
 ** \BT2 Reload Timer function Slewing
 ******************************************************************************/

static void RtUnderflowIrqCallback_Slew(void) // every time we reach underflow this func
{
  if (Position_Control_Slew){
    /*Position calculation*/
    Out_Pos_Slew = ((Enc_value_Slew)/4)*1/PULSES_PER_REV_SLEW_FLOAT;
    Out_Pos_Slew = Out_Pos_Slew*6.28; // Rev to rad
    /*Control*/
    Control_Slew_Pos();

  }
```

```
  else {
    /*Velocity calculation*/
    Out_Vel_Slew = MIN_TO_SEC*((Enc_value_Slew - Enc_value_Slew_Old)/4)*1/Time_BT2*1/PUI
    Out_Vel_Slew = REV_TO_RAD_S*Out_Vel_Slew;
    /*Control*/
    Control_Slew();
   /*Update values*/
    Enc_value_Slew_Old = Enc_value_Slew;
  }


}


/**
 ******************************************************************************
 ** \BT1 Reload Timer function Luffing
 ******************************************************************************/

static void RtUnderflowIrqCallback_Luff(void) // every time we reach underflow this func
{
  if (Position_Control_Luff){
    /*Position calculation*/
    Out_Pos_Luff = ((Enc_value_Luff)/4)*1/PULSES_PER_REV_LUFF_FLOAT;
    Out_Pos_Luff = Out_Pos_Luff*6.28; //Rev to rad
    /*Control*/
    Control_Luff_Pos();
  }
  else {
    /*Velocity calculation*/
    Out_Vel_Luff = MIN_TO_SEC*((Enc_value_Luff - Enc_value_Luff_Old)/4)*1/Time_BT1*1/PUI
    Out_Vel_Luff = REV_TO_RAD_S*Out_Vel_Luff;
    // 5600 rpm Vel_Luff (Small motor inside actuator max speed) at >12 V. 45mm/s rod ve
    //Vel_Luff_Rod = (45/5600.0)*Vel_Luff; // mm/s
    /*Control*/
    Control_Luff();
    /*Update values*/
    Enc_value_Luff_Old = Enc_value_Luff;
  }
```

```
  // send value to arduino (IN PROGRESS)
  /*
  if (arduino_sent == TRUE && arduino_sent_CAN == TRUE){
  encoder_value[0] = Enc_value_Luff / 256;
  encoder_value[1] = Enc_value_Luff % 256;
  encoder_value[2] = Enc_value_Slew / 256;
  encoder_value[3] = Enc_value_Slew % 256;
  while(u4Cnt < 4)
    {
        //while (TRUE != Mfs_Uart_GetStatus(UartCh12, UartTxEmpty)); //wait until TX bu
        Mfs_Uart_SendData(UartCh12, encoder_value[u4Cnt]);
        printf ("%d\n", encoder_value[u4Cnt]);
        u4Cnt++;
    }
  u4Cnt = 0;
  }*/
}

/**
 ******************************************************************************
 ** \BT0 Reload Timer function Slewing position
 ******************************************************************************/
/*
static void RtUnderflowIrqCallback_Slew_Pos(void)
{

}*/


/**
 ******************************************************************************
 ** \BT3 Reload Timer function Luffing position
 ******************************************************************************/
/*
static void RtUnderflowIrqCallback_Luff_Pos(void) // every time we reach underflow(line
{

}*/
```

```
/**
 *****************************************************************************
 ** \PWM Initialization  BT8 (Slew) and BT9 (Luff) and BT6 (Enabling Driver)
 *****************************************************************************/
static void PWM_Init(void)
{
    stc_bt_pwm_config_t stcPwmConfig;

    /* Clear structures */
    PDL_ZERO_STRUCT(stcPwmConfig);

    /* Uses TIOA8_2 at the output pin of BT ch.8 TIOA / P50*/
    SetPinFunc_TIOA8_2_OUT();
    /* Uses TIOA9_2 at the output pin of BT ch.9 TIOA / P52*/
    SetPinFunc_TIOA9_2_OUT();
    /* Uses TIOA6_1 at the output pin of BT ch.6 TIOA */
    SetPinFunc_TIOA6_1_OUT();

    /* Initialize PWM timer BT8 and BT9 and BT6*/ // PWM that is output
    stcPwmConfig.enPres = PwmPres1Div16;  /* PWM clock = 175kHz @ PCLK = 90MHz */ // Ou
    stcPwmConfig.enMode = PwmContinuous;
    stcPwmConfig.enExtTrig = PwmExtTrigDisable;
    stcPwmConfig.enOutputMask = PwmOutputNormal;
    stcPwmConfig.enOutputPolarity = PwmPolarityLow;
    stcPwmConfig.enRestartEn = PwmRestartEnable;

    /**********BT8**********/
    Bt_Pwm_Init(&BT8 , &stcPwmConfig);
    /* Set cycle and duty value */
    Bt_Pwm_WriteCycleVal(&BT8, PWM_CYCLE_VALUE_90_DUTY); // PWM_freq = Freq_scaled_by_pr
    Bt_Pwm_WriteDutyVal(&BT8, PWM_CYCLE_VALUE_10_DUTY); //Do not set a value that makes
    /* Enable count operation */
    Bt_Pwm_EnableCount(&BT8);
    /* Start triggered by software */
    Bt_Pwm_EnableSwTrig(&BT8);
```

```
    /**********BT9**********/
    Bt_Pwm_Init(&BT9 , &stcPwmConfig);
    /* Set cycle and duty value */
    Bt_Pwm_WriteCycleVal(&BT9, PWM_CYCLE_VALUE_100_DUTY); // PWM_freq = Freq_scaled_by_p
    Bt_Pwm_WriteDutyVal(&BT9, PWM_CYCLE_VALUE_10_DUTY); //Do not set a value that makes
    /* Enable count operation */
    Bt_Pwm_EnableCount(&BT9);
    /* Start triggered by software */
    Bt_Pwm_EnableSwTrig(&BT9);

    /**********BT6**********/
    Bt_Pwm_Init(&BT6 , &stcPwmConfig);
    /* Set cycle and duty value */
    Bt_Pwm_WriteCycleVal(&BT6, PWM_CYCLE_VALUE_100_DUTY); // PWM_freq = Freq_scaled_by_p
    Bt_Pwm_WriteDutyVal(&BT6, PWM_CYCLE_VALUE_10_DUTY); //Do not set a value that makes
    /* Enable count operation */
    Bt_Pwm_EnableCount(&BT6);
    /* Start triggered by software */
    Bt_Pwm_EnableSwTrig(&BT6);
}

/**
 ******************************************************************************
 ** \Reload Timer Initialization  BT2 (Slew) and BT1 (Luff) BT0 (Slew position) and BT3
 ******************************************************************************/
static void Reload_Timer_Init(void)
{
    stc_bt_rt_config_t stcRtConfig;
    stc_rt_irq_en_t stcIrqEn_Slew;
    stc_rt_irq_cb_t stcIrqCb_Slew;
    stc_rt_irq_en_t stcIrqEn_Luff;
    stc_rt_irq_cb_t stcIrqCb_Luff;
    /*
    stc_rt_irq_en_t stcIrqEn_Slew_Pos;
    stc_rt_irq_cb_t stcIrqCb_Slew_Pos;
    stc_rt_irq_en_t stcIrqEn_Luff_Pos;
    stc_rt_irq_cb_t stcIrqCb_Luff_Pos;*/

    /* Clear structures */
```

```
PDL_ZERO_STRUCT(stcRtConfig);
PDL_ZERO_STRUCT(stcIrqEn_Slew);
PDL_ZERO_STRUCT(stcIrqCb_Slew);
PDL_ZERO_STRUCT(stcIrqEn_Luff);
PDL_ZERO_STRUCT(stcIrqCb_Luff);
/*PDL_ZERO_STRUCT(stcIrqEn_Slew_Pos);
PDL_ZERO_STRUCT(stcIrqCb_Slew_Pos);
PDL_ZERO_STRUCT(stcIrqEn_Luff_Pos);
PDL_ZERO_STRUCT(stcIrqCb_Luff_Pos);*/

/* BT2 & BT1 (Reload Timer) is used to change the duty value every 1ms */ // Timer t
stcIrqEn_Slew.bRtUnderflowIrq = 1;
stcIrqCb_Slew.pfnRtUnderflowIrqCb = RtUnderflowIrqCallback_Slew;
stcIrqEn_Luff.bRtUnderflowIrq = 1;
stcIrqCb_Luff.pfnRtUnderflowIrqCb = RtUnderflowIrqCallback_Luff;
/*stcIrqEn_Slew_Pos.bRtUnderflowIrq = 1;
stcIrqCb_Slew_Pos.pfnRtUnderflowIrqCb = RtUnderflowIrqCallback_Slew_Pos;
stcIrqEn_Luff_Pos.bRtUnderflowIrq = 1;
stcIrqCb_Luff_Pos.pfnRtUnderflowIrqCb = RtUnderflowIrqCallback_Luff_Pos;*/

/********BT2********/
/* Initialize reload Timer BT2 */
stcRtConfig.enPres = RtPres1Div1024; /* PWM clock = 11.25MHz @ PCLK = 180MHz */ // F
stcRtConfig.enSize = RtSize16Bit;
stcRtConfig.enMode = RtReload;
stcRtConfig.pstcRtIrqCb = &stcIrqCb_Slew;
stcRtConfig.pstcRtIrqEn = &stcIrqEn_Slew;
stcRtConfig.bTouchNvic = TRUE;
Bt_Rt_Init(&BT2 , &stcRtConfig);
/* Set cycle and duty value: 11250*1/11.25MHz = 1ms(11250 = 11249 + 1) */ // Our cas
Bt_Rt_WriteCycleVal(&BT2, BT2_CYCLE_VALUE);
/* Enable count operation */
Bt_Pwm_EnableCount(&BT2);
/* Start triggered by software */
Bt_Pwm_EnableSwTrig(&BT2);
/* Time Calculation */
switch (stcRtConfig.enPres) {
            case RtPresNone:
              PRESCALE_BT2 = 1;
```

```
            break;

        case RtPres1Div4:
            PRESCALE_BT2 = 4;
            break;

        case RtPres1Div16:
            PRESCALE_BT2 = 16;
            break;

        case RtPres1Div128:
            PRESCALE_BT2 = 128;
            break;

        case RtPres1Div256:
            PRESCALE_BT2 = 256;
            break;

        case RtPres1Div512:
            PRESCALE_BT2 = 512;
            break;

        case RtPres1Div1024:
            PRESCALE_BT2 = 1024;
            break;

        case RtPres1Div2048:
            PRESCALE_BT2 = 2048;
            break;
    }
Time_BT2 = (BT2_CYCLE_VALUE*PRESCALE_BT2)*1/PROCESSOR_FREQ;

/********BT1********/
/* Initialize reload Timer BT1 */
stcRtConfig.enPres = RtPres1Div1024; /* PWM clock = 11.25MHz @ PCLK = 180MHz */ // E
stcRtConfig.enSize = RtSize16Bit;
stcRtConfig.enMode = RtReload;
stcRtConfig.pstcRtIrqCb = &stcIrqCb_Luff;
stcRtConfig.pstcRtIrqEn = &stcIrqEn_Luff;
```

```
stcRtConfig.bTouchNvic = TRUE;
Bt_Rt_Init(&BT1 , &stcRtConfig);
/* Set cycle and duty value: 11250*1/11.25MHz = 1ms(11250 = 11249 + 1) */ // Our cas
Bt_Rt_WriteCycleVal(&BT1, BT1_CYCLE_VALUE);
/* Enable count operation */
Bt_Pwm_EnableCount(&BT1);
/* Start triggered by software */
Bt_Pwm_EnableSwTrig(&BT1);
/* Time Calculation */
switch (stcRtConfig.enPres) {
            case RtPresNone:
              PRESCALE_BT1 = 1;
              break;

            case RtPres1Div4:
              PRESCALE_BT1 = 4;
              break;

            case RtPres1Div16:
              PRESCALE_BT1 = 16;
              break;

            case RtPres1Div128:
              PRESCALE_BT1 = 128;
              break;

            case RtPres1Div256:
              PRESCALE_BT1 = 256;
              break;

            case RtPres1Div512:
              PRESCALE_BT1 = 512;
              break;

            case RtPres1Div1024:
              PRESCALE_BT1 = 1024;
              break;

            case RtPres1Div2048:
```

```
            PRESCALE_BT1 = 2048;
            break;
        }
Time_BT1 = (BT1_CYCLE_VALUE*PRESCALE_BT1)*1/PROCESSOR_FREQ;

/********BT0********/
/* Initialize reload Timer BT0 */
//stcRtConfig.enPres = RtPres1Div1024; /* PWM clock = 11.25MHz @ PCLK = 180MHz */ /
//stcRtConfig.enSize = RtSize16Bit;
//stcRtConfig.enMode = RtReload;
//stcRtConfig.pstcRtIrqCb = &stcIrqCb_Slew_Pos;
//stcRtConfig.pstcRtIrqEn = &stcIrqEn_Slew_Pos;
//stcRtConfig.bTouchNvic = TRUE;
//Bt_Rt_Init(&BT0 , &stcRtConfig);
/* Set cycle and duty value: 11250*1/11.25MHz = 1ms(11250 = 11249 + 1) */ // Our cas
//Bt_Rt_WriteCycleVal(&BT0, BT0_CYCLE_VALUE);
/* Enable count operation */
//Bt_Pwm_EnableCount(&BT0);
/* Start triggered by software */
//Bt_Pwm_EnableSwTrig(&BT0);
/* Time Calculation */
/*switch (stcRtConfig.enPres) {
            case RtPresNone:
              PRESCALE_BT0 = 1;
              break;

            case RtPres1Div4:
              PRESCALE_BT0 = 4;
              break;

            case RtPres1Div16:
              PRESCALE_BT0 = 16;
              break;

            case RtPres1Div128:
              PRESCALE_BT0 = 128;
              break;

            case RtPres1Div256:
```

```
              PRESCALE_BT0 = 256;
              break;

            case RtPres1Div512:
              PRESCALE_BT0 = 512;
              break;

            case RtPres1Div1024:
              PRESCALE_BT0 = 1024;
              break;

            case RtPres1Div2048:
              PRESCALE_BT0 = 2048;
              break;
          }
Time_BT0 = (BT0_CYCLE_VALUE*PRESCALE_BT0)*1/PROCESSOR_FREQ;*/

/********BT3********/
/* Initialize reload Timer BT3 */
//stcRtConfig.enPres = RtPres1Div1024; /* PWM clock = 11.25MHz @ PCLK = 180MHz */ /
//stcRtConfig.enSize = RtSize16Bit;
//stcRtConfig.enMode = RtReload;
//stcRtConfig.pstcRtIrqCb = &stcIrqCb_Luff_Pos;
//stcRtConfig.pstcRtIrqEn = &stcIrqEn_Luff_Pos;
//stcRtConfig.bTouchNvic = TRUE;
//Bt_Rt_Init(&BT3 , &stcRtConfig);
/* Set cycle and duty value: 11250*1/11.25MHz = 1ms(11250 = 11249 + 1) */ // Our cas
//Bt_Rt_WriteCycleVal(&BT3, BT3_CYCLE_VALUE);
/* Enable count operation */
//Bt_Pwm_EnableCount(&BT3);
/* Start triggered by software */
//Bt_Pwm_EnableSwTrig(&BT3);
/* Time Calculation */
/*switch (stcRtConfig.enPres) {
            case RtPresNone:
              PRESCALE_BT3 = 1;
              break;

            case RtPres1Div4:
```

```
                PRESCALE_BT3 = 4;
                break;

            case RtPres1Div16:
                PRESCALE_BT3 = 16;
                break;

            case RtPres1Div128:
                PRESCALE_BT3 = 128;
                break;

            case RtPres1Div256:
                PRESCALE_BT3 = 256;
                break;

            case RtPres1Div512:
                PRESCALE_BT3 = 512;
                break;

            case RtPres1Div1024:
                PRESCALE_BT3 = 1024;
                break;

            case RtPres1Div2048:
                PRESCALE_BT3 = 2048;
                break;
        }
    Time_BT3 = (BT3_CYCLE_VALUE*PRESCALE_BT3)*1/PROCESSOR_FREQ;*/
}


/**
 ******************************************************************************
 ** \Extint Initialization
 ******************************************************************************/

static void exint_Init(void)
{
    stc_exint_config_t stcExintConfig;
```

```
/* The internal pull-up resistance can be connected for the external
interrupt pin with falling edge detection if necessary */
/* E.g.: Gpio1pin_InitIn ( GPIO1PIN_Pxy, Gpio1pin_InitPullup(1u)); */


SetPinFunc_INT01_1(0u);                         /* Pin Function: Encoder A Slewing Ris
SetPinFunc_INT02_1(0u);                         /* Pin Function: Encoder A Slewing Fal
SetPinFunc_INT03_1(0u);                         /* Pin Function: Encoder B Slewing Ris
SetPinFunc_INT04_1(0u);                         /* Pin Function: Encoder B Slewing Fal
SetPinFunc_INT06_1(0u);                         /* Pin Function: Encoder A Luffing Ris
SetPinFunc_INT20_1(0u);                         /* Pin Function: Encoder A Luffing Fal
SetPinFunc_INT22_0(0u);                         /* Pin Function: Encoder B Luffing Ris
SetPinFunc_INT28_0(0u);                         /* Pin Function: Encoder B Luffing Fal

/* Configure interrupt ch.6 and ch.8 */
PDL_ZERO_STRUCT(stcExintConfig);

stcExintConfig.abEnable[ExintInstanceIndexExint1] = 1u;
stcExintConfig.aenLevel[ExintInstanceIndexExint1] = ExIntRisingEdge;
stcExintConfig.apfnExintCallback[ExintInstanceIndexExint1] = Main_ExtInt1Callback;

stcExintConfig.abEnable[ExintInstanceIndexExint2] = 1u;
stcExintConfig.aenLevel[ExintInstanceIndexExint2] = ExIntFallingEdge;
stcExintConfig.apfnExintCallback[ExintInstanceIndexExint2] = Main_ExtInt2Callback;

stcExintConfig.abEnable[ExintInstanceIndexExint3] = 1u;
stcExintConfig.aenLevel[ExintInstanceIndexExint3] = ExIntRisingEdge;
stcExintConfig.apfnExintCallback[ExintInstanceIndexExint3] = Main_ExtInt3Callback;

stcExintConfig.abEnable[ExintInstanceIndexExint4] = 1u;
stcExintConfig.aenLevel[ExintInstanceIndexExint4] = ExIntFallingEdge;
stcExintConfig.apfnExintCallback[ExintInstanceIndexExint4] = Main_ExtInt4Callback;

/* Before initialize external interrupt ch.6, make sure
* PDL_PERIPHERAL_ENABLE_EXINT0 is set to PDL_ON in pdl_user.h
* If external interrupt ch.6 is not used, set PDL_PERIPHERAL_ENABLE_EXINT6
* to PDL_OFF !! Otherwise, the external interrupt ch.6 may be mis-enabled. */
stcExintConfig.abEnable[ExintInstanceIndexExint6] = 1u;
```

```
    stcExintConfig.aenLevel[ExintInstanceIndexExint6] = ExIntRisingEdge;
    stcExintConfig.apfnExintCallback[ExintInstanceIndexExint6] = Main_ExtInt6Callback;

    stcExintConfig.abEnable[ExintInstanceIndexExint20] = 1u;
    stcExintConfig.aenLevel[ExintInstanceIndexExint20] = ExIntFallingEdge;
    stcExintConfig.apfnExintCallback[ExintInstanceIndexExint20] = Main_ExtInt20Callback;

    stcExintConfig.abEnable[ExintInstanceIndexExint22] = 1u;
    stcExintConfig.aenLevel[ExintInstanceIndexExint22] = ExIntRisingEdge;
    stcExintConfig.apfnExintCallback[ExintInstanceIndexExint22] = Main_ExtInt22Callback;

    stcExintConfig.abEnable[ExintInstanceIndexExint28] = 1u;
    stcExintConfig.aenLevel[ExintInstanceIndexExint28] = ExIntFallingEdge;
    stcExintConfig.apfnExintCallback[ExintInstanceIndexExint28] = Main_ExtInt28Callback;

    stcExintConfig.bTouchNvic = TRUE;

    Exint_Init(&stcExintConfig);

}

/**
 ******************************************************************************
 ** \UART to Arduino Initialization
 ******************************************************************************/


void InitUart_Arduino(void){

    stc_mfs_uart_config_t stcUartConfig;

    SetPinFunc_SIN12_0();
    SetPinFunc_SOT12_0();


    PDL_ZERO_STRUCT(stcUartConfig);
    /* Initialize UART TX and RX channel */
    stcUartConfig.enMode = UartNormal;
    stcUartConfig.u32BaudRate = 9600;
```

```
    stcUartConfig.enDataLength = UartEightBits;
    stcUartConfig.enParity = UartParityNone;
    stcUartConfig.enStopBit = UartOneStopBit;
    stcUartConfig.enBitDirection = UartDataLsbFirst;
    stcUartConfig.bInvertData = FALSE;
    stcUartConfig.bHwFlow = FALSE;
    stcUartConfig.bUseExtClk = FALSE;
    stcUartConfig.pstcFifoConfig = NULL;

    Mfs_Uart_Init(UartCh12, &stcUartConfig);

    /* Enable RX function of UART12 */
    Mfs_Uart_EnableFunc(UartCh12, UartRx);
    /* Enable TX function of UART12 */
    Mfs_Uart_EnableFunc(UartCh12, UartTx);
}

/**
 ******************************************************************************
 ** \Main Function
 ******************************************************************************/

int main(void)
{
    /* UART communication with the terminal 115200 Baud, MFS0, UART0*/
    Uart_Io_Init();
    /* Initialize PWM PB8 and PB9*/
    PWM_Init();
    /* Initialize Reload Timer PB2 and PB1*/
    Reload_Timer_Init();
    /* Initialize exint */
    exint_Init();
    /* UART initialization with Arduino 9600 Baud */
    InitUart_Arduino();
    //Enabling driver
    setPWM_driver_enabling (PWM_CYCLE_VALUE_90_DUTY); // More than 3V to enable driver

    /* //ARDUINO RETURN VALUE (IN PROGRESS)
    while(u4Cnt_main < 4)
```

```
    {
      while(TRUE!=Mfs_Uart_GetStatus(UartCh12,UartRxFull));
      au4UartRxBuf[u4Cnt_main]=Mfs_Uart_ReceiveData(UartCh12);
      printf("%d", au4UartRxBuf[u4Cnt_main]);
      u4Cnt_main++;
    }

    Enc_value_Luff = au4UartRxBuf[0] * 256 + au4UartRxBuf[1];
    Enc_value_Slew = au4UartRxBuf[2] * 256 + au4UartRxBuf[3];
    printf("Luff: %d Slew: %d \n", Enc_value_Luff, Enc_value_Slew);
    arduino_sent = TRUE;
    */
    while(1)
    {
      arduino_sent_CAN = FALSE; //Avoid the transmitting code from the interrupt to inte
      Readmsg();
      arduino_sent_CAN = TRUE;
    }
}
/* [] END OF FILE */
```

Bibliography

[1] Imo: Worm and Spur Gear Slewing Drives, Engineering and more,
https://www.imousacorp.com/spdescription/

[2] IGUS: Plastics for longer life,
https://www.igus.co.uk/info/slewing-ring-slewing-ring-with-gear-teeth

[3] APEM: HF Series joystick,
https://www.apem.com/us/hf-series-42.html

[4] SKF: Slewing Bearings,
http://www.skf.com/group/products/bearings-units-housings/slewing-bearings/index.html

[5] Raspberry Pi: CAN bus on raspberry pi with MCP2515,
https://www.raspberrypi.org/forums/viewtopic.php?t=141052

[6] How to control two external hardware peripherals using Raspberry Pi's SPI,
https://radiostud.io/understanding-spi-in-raspberry-pi/

[7] Carlos Gonzalez - "What's the Difference Between Spur, Helical, Bevel, and Worm Gears?",
http://www.machinedesign.com/whats-difference-between/what-s-difference-between-spurh

[8] MEAD Info - "Comparison of Gear Efficiencies - Spur, Helical, Bevel, Worm, Hypoid, Cycloid",
http://www.meadinfo.org/2008/11/gear-efficiency-spur-helical-bevel-worm.html

[9] Misumi USA - Spur Gears - Pressure Angle 20Deg., Module 2.0,
https://us.misumi-ec.com/vona2/detail/110300428610/

[10] Ehsan A. Maleki - "Dynamics and Control of a Small-Scale Mobile Boom Crane", Georgia Institute of Technology, December 2010

[11] Lisa Eitel, "What is a gearmotor? Technical Summary",
https://www.motioncontroltips.com/whatis- a-gearmotor-technical-summary/

[12] Oriental Motor, "AC or DC? Brushed DC or Brushless DC Gear Motor?",
http://www.orientalmotor.com/brushless-dc-motors-gear-motors/technology/AC-brushless-
html

[13] Worm Screw Gear Motors: PCC,
http://www.minimotor.com/eng/products/worm-screw-gear-motors/worm-screw-gear-motor-co

[14] BEI Sensors - "Choosing the Right Sensor Technology",
http://www.beisensors.com/customerresources/ bei-choosing-the-right-sensor-technology

[15] EPC, "What IS an encoder?",
http://encoder.com/blog/company-news/what-is-an-encoder

[16] DYNAPAR - "Angle Encoders: How to Measure Angle Using Encoders",
https://www.dynapar.com/knowledge/applications/angle-encoders/

[17] Core Electronics - "Motor Drivers vs Motor Controllers",
https://core-electronics.com.au/tutorials/motor-drivers-vs-motor-controllers.html

[18] Maxon Motor - "Escon 50/5",
https://www.maxonmotor.com/maxon/view/news/MEDIENMITTEILUNG-ESCON-50-5

[19] How to use interrupts with Python on the Raspberry Pi and RPi.GPIO,
https://raspi.tv/2013/how-to-use-interrupts-with-python-on-the-raspberry-pi-and-rpi-g

[20] How to Use Moving Average Filters,
https://medium.com/blueeast/how-to-use-moving-average-filter-to-counter-noisy-data-si