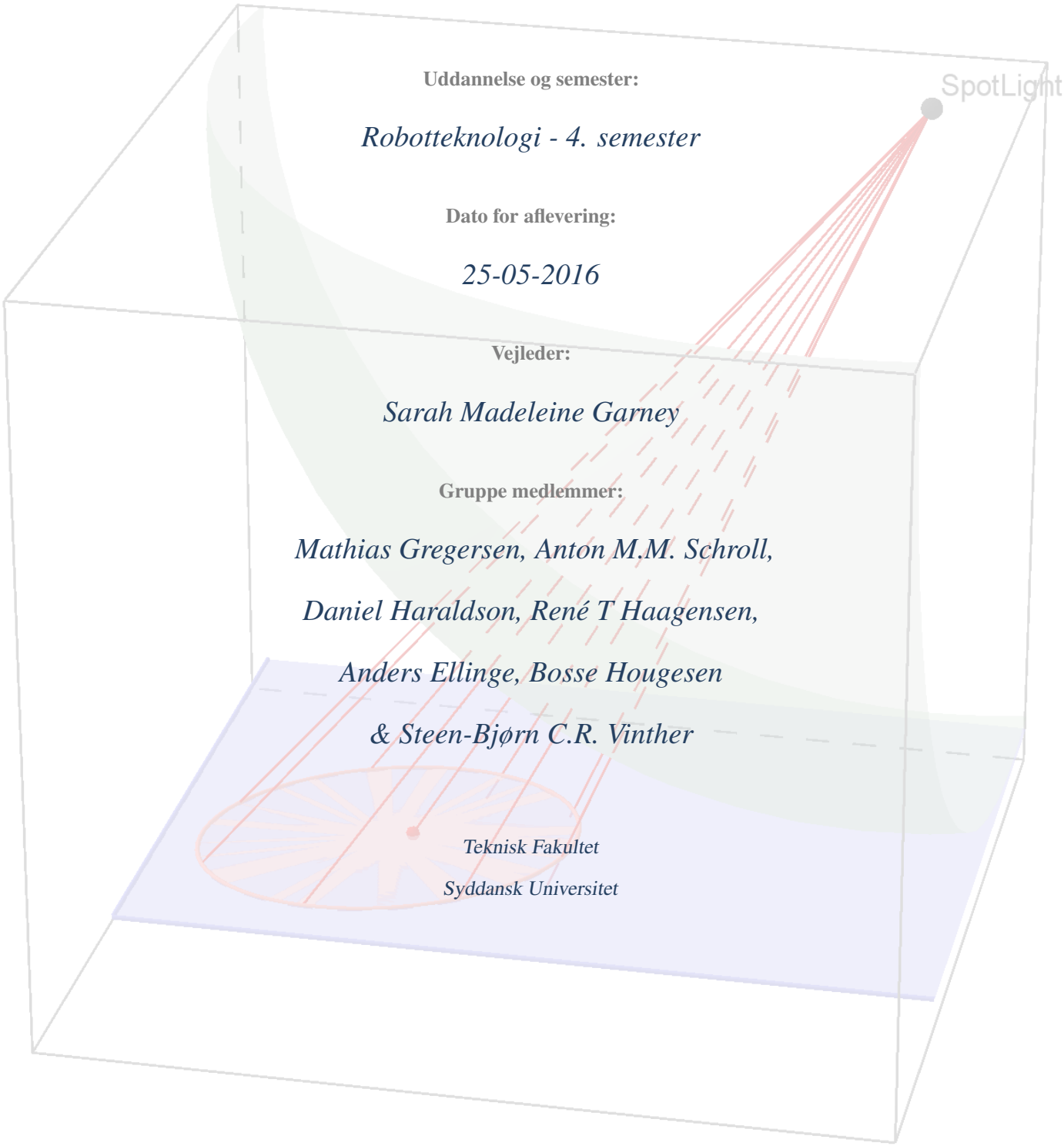
Pan & Tilt

Regulering og Kontrol af Robotsystemer - Gruppe 5



Uddannelse og semester:

Robotteknologi - 4. semester

Dato for aflevering:

25-05-2016

Vejleder:

Sarah Madeleine Garney

Gruppe medlemmer:

Mathias Gregersen, Anton M.M. Schroll,

Daniel Haraldson, René T Haagensen,

Anders Ellinge, Bosse Hougesen

& Steen-Bjørn C.R. Vinther

Teknisk Fakultet

Syddansk Universitet

Preface

This report is written by seven 4th semester robotic engineering students from the University of Southern Denmark in spring 2016. The report is based on the 4th semester's project. It is aimed at people with basic knowledge in the following disciplines: Digital programmable electronics, Embedded Programming, Control Engineering and Computer systems. Furthermore, in order to understand the content of this report, it is considered a requisite that the project introduction has been read [1]. This report describes the theory and analysis involved in designing the controller used to control the Pan & Tilt system, as well as the motor driver and the application used to give commands to the system.
The report is written as a conclusion to the project and contains documentation of the process and the experiences gained. It should therefore be possible to recreate the project using this report.

[1] Projektoplæg - RB-RKO4 forår 2016

1 Abstract

The purpose of this paper is to show how to make a controller and how to use it to control a spotlight consisting of two connected frames, two motors and one light. Precise motor control is an essential requirement in most applications involving any kind of actuators that needs to behave in a predictable and controlled way. A spotlight capable of relative precise movement is important when performing on a stage. In order to make a controller, several measurements of the system's response to different voltages have been analysed. Using Matlab and control theory the values of the controller have been estimated. The results show; with the right controller, the system is easily controllable and will behave in a predictable way.

Table of content

2 Problem formulation

When performing on a stage, proper handling of light is important. This can be achieved through controllable spotlights. They have to be able to rotate horizontally and vertically, thereby targeting specific points on the stage and do so autonomously.
The Pan & Tilt System (P & T System) can be made to meet these requirements.

2.1 Definition of goals

The main goals of the product is, to be able to point the cone of light from the spotlight, at a specific point on a predefined surface or follow a human walking around on a surface. The user should have the option, to specify the limits of the spotlight's working space. The positions or paths should be set by the user and the movement of the cone should be pleasant to watch.

2.2 Requirements specification

The requirements will be divide into primary and secondary requirements, so as to identify the most important objectives.

2.2.1 Primary requirements

Hardware requirements

- The controller has to be implemented on the FPGA(Field-programmable gate array).
- The communication between the microprocessor and the FPGA has to be done with SPI(Serial Peripheral Interface Bus).
- The FPGA has to control the PWM(Pulse-width modulation) signal for the motors.

Driver requirements

- The system cannot have an overshoot of more than 10%.
- An increment in the hall sensor counter must have a settling time of at least 50 mS.
- The spotlight can have a steady state close to zero

Application requirements

- The application must support for the spotlight to be placed at different heights.
- The application must support different stage dimensions, to a maximum of 14 m x 14 m.

2.2.2 Secondary requirements

These are the requirements that would be nice improvement to the project, but are not strictly necessary for the completion of the project.

- The movement between two points should be in a smooth motion.
- The P&T system is to recalibrate the position of the Pan and the tilt during startup.
- The spotlight can be controlled by an application.
- The spotlight can be controlled by a joystick.
- At a scene of dimension 5 m x 5 m, from a height of 5 meters, the system must achieve a velocity of at least $2.5m/s$[1] relative to the scene, which is the maximum velocity the average person would walk. [2]

3 Indledende idé fase

Applikation
Hvordan kan arbejdsområdet afgrænses efter brugerens behov?
Hvilke input / parametre skal kontrolsystemet manipulere?
Controller
Hvordan kan controlleren modelleres?
Hvordan kan de forskellige opgaver opdeles i tasks?
Hvordan udarbejdes en styring til motoren?
Hvilke parametre ønskes kontrolleret?
Hardware
Hvordan anvendes de indbyggede sensorer?
Hvordan kan platformen modelleres?
Hvordan kan systemet styres pålideligt?
Hvilke fysiske begrænsninger medfører hardwaren?
Dette bygger på antagelsen at pan and tilt projektet altid virker
Bevæge systemet i forhold til koordinater (Systemet skal kunne bevæge sig, i forhold til givne koordinater)

4 The System

Here goes the tekst.

5 The Microcontroller

Given the design requirements in section 2.2, and the block diagram in section 4. This includes all the functionality that is to be placed in the microcontroller. From here the next stage is to define the content of these blocks, and how they interact with each other.

[2] Wikipedia, Preferred walking speed, `https://en.wikipedia.org/wiki/Preferred_walking_speed`, 23/05/16

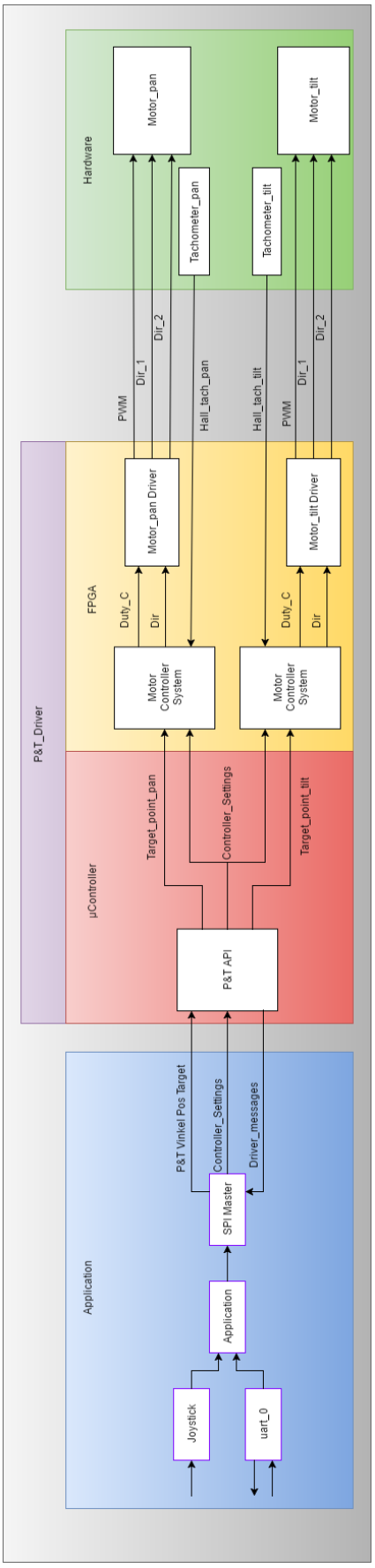


Figure 1: The overall system block diagram

The microcontroller is to establish an application programmer interface (API) to the P&T driver placed in the FPGA. The microcontroller will also run the application that uses this interface in order to help the user achieve his/her goal, which is to control the spotlight.

The program will be written in the C language, and it will be documented using task diagrams and state machines.

5.1 Choice of Operating System

It is safe to assume that this program will be using more than one task, some sort of operating system (OS) is required. The decision of witch operating system to use, will have great effect on the entire project, since it defines the domain in witch the entire application will have to operate within. This OS must therefore as a minimum support the ARM Cortex M-4 processor, which is used in this project.

By using a Real Time Operating System (RTOS) like freeRTOS, it would give the advantages of preemptive scheduling. Being able to preempt a low priority task to pave the way for a task of higher priority, is an advantage when low response time is important. However. the application will only benefit from using a RTOS if its size extends 1 MB. Application smaller then 64KB simply would not be large enough for a RTOS to have any effect relative to a standard OS.[3]

Assuming that the project will reach this size, combined with the fact that this OS is used in one of the courses doing this semester, freeRTOS was chosen.

5.2 Program Structure

Now when the operating system is defined, the next stage will be to define the actual application that needs to be implemented. Given the block diagram of figure 1, the blocks that make up the system is already defined.

5.2.1 Defining tasks

First step is to identify the tasks that make up the system. A good rule of thumb is to make a driver task for every peace of hardware that the system is to interact with. This idea is based on the Parallel criteria which implies that things can be separated in to different tasks if the functionality runs independent and simultaneously, which indeed will be the case for a hardware driver. However the UART0 driver that supports connection between the PC and micro controller, can send and receive data independently. Therefore this task is split up in to two tasks. A receive task, and a transmit task. Doing so also have the advantage of reusability, for other applications requiring the same hardware.

Now when all the hardware is taken care of, the only thing remaining is the application it self. The application should be able to handle commands given by the user. Commands could be ”set coordinate pan, tilt”, ”run light-show #2” an ”set minimum velocity tilt”. Since the entire application will consist of function calls, the main body of the application will take the shape of a kernel task that execytes the functions defined in the function list shown in appendix A. However by executing the command ”run light show” it require of the application to feed the P&T driver with new target points once the previus target point has been reached. considering that one still

[3]Texas Instruments, Why use RTOS in MCU Applications, `http://www.ti.com.cn/cn/lit/wp/spry238/spry238.pdf`, 23/05/16

would like to send commands while running a light-show this will make two tasks based on the parallel criteria. The kernel Task and the light-show task.

Last but not least the application operate on the current position in order to handle a light-show. But gaining the current position requires pulling from the application. This task therefore makes the final task in the system based on the Periodical Criteria.

The application must therefore consist of the following tasks:

Driver tasks:

- UART0_rx_task
- UART0_tx_task
- Joystick_task
- SPI_Master_task

Application tasks:

- app_kernel_task
- app_lightshow_task
- app_update_current_task

5.2.2 Connecting the tasks

Now the tasks are to be linked together. This will happen in a structure so that every task that requires input from other tasks has one and only one queue to pull from. This is decried since the OS can pause the task while waiting for one queue. The UART tasks and the joystick tasks will both send commands to kernel via the application queue, an shared state memory. The same goes for the connection between the kernel task and the light-show task.

Each queue needs to be semaphore protected in order to prevent a race condition when reading an writing to the queue, but since freeRTOS handles who gets to use the queues, it must be safe to amuse that such conditions is taken care of in the OS.

The shared state memory (SSM), needs semaphore protection as well. Since the number of memory locations is relatively small (less then 64), combined with the fact that the critical section only consists of one read ore write. the entire SSM will be handled by the same semaphore for simplicity.

The tow application tasks app_update_current_task and app_kernel_task will both be accessing the spi_tx_queue, but this will only be a problem if they wish to send multiple messages in context, which is not the case for this application. The same goes for the application_queue. Here the send messages and events content will be independent form each other.

Applying all these thoughts will result in the final application task diagram as shown in figure 2. Now when the task diagram has been made, it is time to considerer possible structural weakness such as deadlocks and handling og queues.

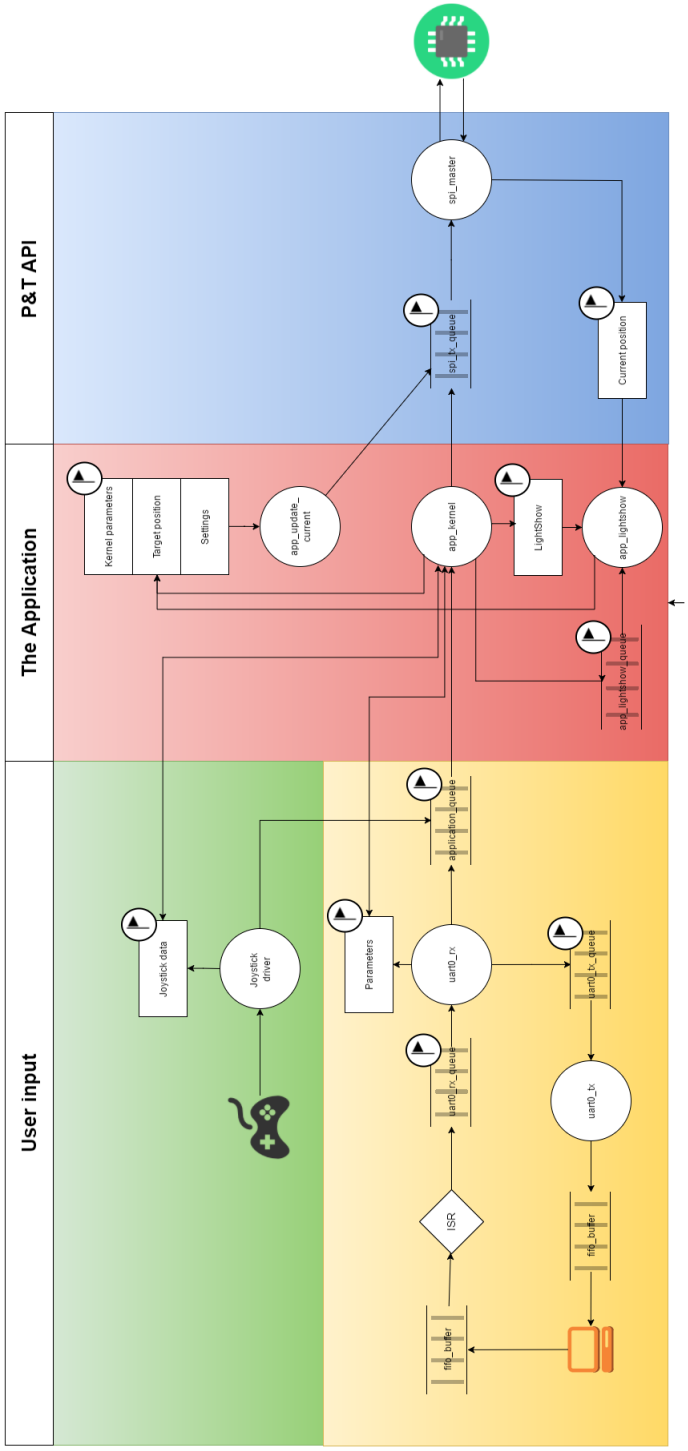


Figure 2: The application task diagram

5.3 The Application Programmer Interface

When handling a peace of hardware like the P&T system or the UART0 unit, it is favorable to make reusable drivers, but if the unit is to interact with the application following specific protocols, you quickly looses the reusability. Instead you may be writing a seperate module called an Application Programmer Interface (API), containing all the features necessary, for the application to handle the driver easily and correct.

An example of this can be found in the P&T API. This module basically handles the format of the messages send ot the spi_tx_queue and received by the spi_master_task. Here the message must have ”n” address bits and ”m” message bits, but say that we one day were to change this message format to something different, then it would only effect the P&T API module, while leaving the rest of the application as is. Using API’s will therefor make the application more agile.

5.4 Event Queues and Shared State Memory

When operating with tasks, it should come as no surprise that it will be implemented as a state machine. This structure will operate in one state until a specific condition is met, from here the tasks will operate en ways defined by the new state. For this application the tasks will mostly switch states as they receive the correct events. In short an event is a number that the programmer has assigned a specific meaning to. It is therefor impotent for the application to know if it is reading an envent ore not. In other words, if the sending task uses the queue for both user data and events, it will be impossible for the reviver to know, if the number 214 is to be interpreted as the value, or the event e.g ”UPDATE_DISPLAY”.

Suppose an event is an eight bit unsigned integer, and it is send to a task through a queue. It is impossible for the receiver task to know what data type it is, and who send it. It will therefore have to send extra information stating the data type in the shape of a header message. An alternative would splitting up the communication in a event_queue and a SSM buffer. Only events will go in the queue, and data will first be placed in the SSM and then an SSM_READ event is send to the receiver task. Form here the receiver will read the data and set it to zero. This structure is shown in figure 3.

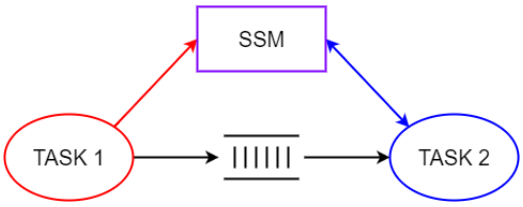


Figure 3: Communication using event queues

It is possible to first send a header message telling what to do with the next message, however, this solution would not be particularly robust, since all it takes for the system to get lost is for one message to fail at either get send or received incorrectly. It will then go out of synchronization and header will be read as data and the other way around, and since this error only happens at rarely this error will be hard to debug. It is therefore preferred to keep the messages atomic, by using a queue and a SSM address.

5.5 Deadlock Protection

A deadlock has happed when two or more process wait for each other to release a resource, but since they all are waiting, no one will ever finish their task. The tasks and resources will therefore be unusable or deadlocked for the rest of the applications lifetime.

1. **Mutual exclusion**: is the concept of making sure that two or more concurrent processes are not in their critical section at the same time.
2. **Hold and wait**: is when a process holds on to a resources and wait for the rest of the resources it needs.
3. **no preemption**: means that only the process that holds the resource can release it. If preemption is allowed, the program is able to take resources from a process and allocate it to another process if needed.
4. **Circular wait**: is when one process is waiting for at resource that another process have, and the new process is waiting for a resource held by the first one. As long at this makes up a closed chain of waiting processes there is a circular wait.

All four conditions needs to hold true in order for a deadlock to occur. You can therefore provide a deadlock-free environment by avoiding one of these conditions[4].

To avoid deadlocks in the system, the circular wait condition is removed via semaphores. However since the system only have one semaphore circular wait will never be an option.

5.6 SPI-Master

using the build in free-scale spi-module in the TM4C123GH6PM Microcontroller, or creating the protocol from scratch.

The making of the state machine that is the SPI-master

A state is only a state if you have to wait till you move on

5.7 The Application

5.7.1 The Kernel task

making the state mashine for the kernel-task

separating commands in groups depending on the number of parameters to use.

the list of commands that will be supported (se the list on google drive XXXX)

5.7.2 The light-show task

making the state machine for the light-show-task

[4]Operating system Concepts 9th edition, by Abraham Silbershatz, page 319

5.8 Overview

The functionality was meant as a fully dressed system containing all the features that would be needed while working with a real spotlight. However, not all features were primary requirements, making their implementation optional.

The joystick: Even thought a driver was made for the joystick, it the the corresponding API was not. Therefore this

Point at a specific location.

This is the functions that did not made it to the implementation

1. ps2 controller and its corresponding api
2. calculating the tack-position from the relative coordinates given by the application
3. follow a waking person

6 Driver in the FPGA

This section describes how the driver is implemented in the FPGA. The section starts with an overall analysis of the system and afterwards goes into details with some blocks defined in the analysis and ends with an discussion on how the PID controller could have been implemented better as a filter.

6.1 Overall analysis

The easiest way to describe how the overall system works is by looking at the general components and how they are connected. The general components used in the implementation are: a comparator, a controller, a sensor, a motor driver with a PWM module and a SPI slave with a data bank. A generic multiplexer display was also added to simplify the debugging process, however this should not be seen as a part of the final product. Since the goal is to control two motors a separate controller setup was made for each motor. This means that each motor is connected to its own setup of a comparator, a sensor, a controller and a motor driver. The SPI slave component is common for the motors. On figure **??** an illustration of how the components are internally connected. This figure also shows information on the signals between the individual components.

HUSK AT LAVE PLADS TIL OVERALL DRIVER FIGURE

The components can be divided into the following blocks, each with a specific purpose.

The first block is the SPI slave block which contains only the SPI slave component. The purpose of this block is to receive and store information used by the rest of the components. It then shares the information with the components that are in need of them. An example of this could be the maximum and minimum speed of a motor which is send to the controller (this connection can also be seen on figure # Overall driver #).

The second block is the controller block which contains the comparator and the controller components. The purpose of this block is to regulate the motors according to a target input and the feedback. The block generates a duty cycle and a direction that is passed onto the motor driver.

The third block is the sensor block which consists only of the sensor component. The purpose of this block is to keep track of the current position of the motor. This is done by looking at the hall sensors from the motors and relating their output to the current position.

The last block is the motor driver which contain the motor driver component and its internal PWM module. The purpose of this block is to control and power the motor. This means that the block generates a PWM signal and outputs it on the correct pin according to the desired direction.

7 Motor driver block

This section will describe in detail how the motor driver block receives a duty cycle and a direction, then uses these to generate and output a **PWM** signal according to the direction and speed set by the controller block.

7.1 Description

An easy way to visualize the inner workings is to divide the Motor Driver into two separate blocks. The first block's task is to create a **PWM** signal which is equivalent to the duty cycle given by the controller. The second block's task is to set the output of the **PWM** signal to the right pin on the H-Bridge in order to control the direction of the motor.

7.2 Theory

This section will describe the theories used for the implementation.

7.2.1 PWM generation

The **PWM** signal is created based on a looping counter and a threshold. Imagine a counter resetting whenever it reaches a certain value, creating a loop. If the value is set to the period of the wanted **PWM** signal and the **PWM** signal output is dependent on the counter value relative to a threshold, then you will be able create a looping output that is dependant on the threshold. Figure # PWM implementaion principle # show an illustration of the behavior of **PWM** signal based on the looping counter and threshold.

NOGET MED H BRO

7.3 Implementation

The implementation was done with the separation of the block described before. A **PWM** component was created which has the sole purpose of creating a **PWM** signal based on a duty cycle. The other part has the purpose of directing the **PWM** signal to the right output pins based on the direction wanted.

7.3.1 PWM part

The PWM block was implemented so that whenever the counter value is below the threshold the PWM signal output is high and when the counter value is above the threshold the PWM signal output is low. The section of the PWM code that makes the counter can be seen on picture # Counting example #.

7.3.2 Direction part

This block is simply implemented as a switch that toggles on the wanted direction. As an example if the wanted direction is set to 0 then the PWM is driven to the relevant motor control output and the other is kept at zero.

7.4 Tests

To verify that this motor controller is working as intended, a test bench was created and various inputs was tested. The enable was set and a duty cycle of 8 bits was input, which corresponds to a certain duty cycle. A clear spike in PWM output could be seen, of the same length every time PWM is outputting. To further verify the test, the edges of the PWM signal was measured and used to find the exact duty cycle. Table # Table from other motor driver # showcases the tests of PWM.

7.5 Discussion

The tests show a deviation from the desired duty cycle of a small percentage. However these deviations have been deemed insignificant. The errors is created by the method that is used to calculate the PWM from the duty cycle, when converting the bit string, it is multiplied by the PWM period and divided by 256. This yields an integer, that when combined with the counter, yields a PWM very close to the desired duty cycle. Beyond that margin of error, a single clock cycle is lost due to resetting the counter, which corresponds to 1 ns.

7.6 Conclusion

The motor driver has been sufficiently explained and tested, and has become a solid module to create a PWM signal for the P & T system. The PWM calculation can be optimized very slightly by improving the conversion from duty cycle to PWM, and optimizing the counter, as to never lose any clock cycles.

8 Sensor block

This section describes in detail how the sensor block handles the the output of the Hall effect sensors and sends it to the comparator block.

8.1 Description

The responsibility of the sensor block is to handle the output from the Hall effect sensors and determine the position of both the pan part and the tilt part of the system. The controller block rely on this information and thus

this block form the basis of the control method chosen.

8.2 Theory

Hall effect sensors are sensors that changes their voltage output based on proximity of a magnetic field. If combined with the right circuitry it can be used a switch, giving a digital output. When placed around the rotating axis of a motor with the right relative distance, the Hall effect sensors can be used determine the position, direction, rotational velocity and the position of the axis. There is a disc on the axis of the motor. It is divided into 6 sections, alternating between two levels of magnetization. Around the magnet, two Hall effect sensors are placed at an angle of slightly less than 90 degrees(See figure 4). As the motor axis rotate, the magnetized parts move past the sensors and the magnetic field, they are subjected to, changes. Because the Hall effect sensor is a little "off" the output will have an overlap see figure 5. This is what is used to determine the direction.

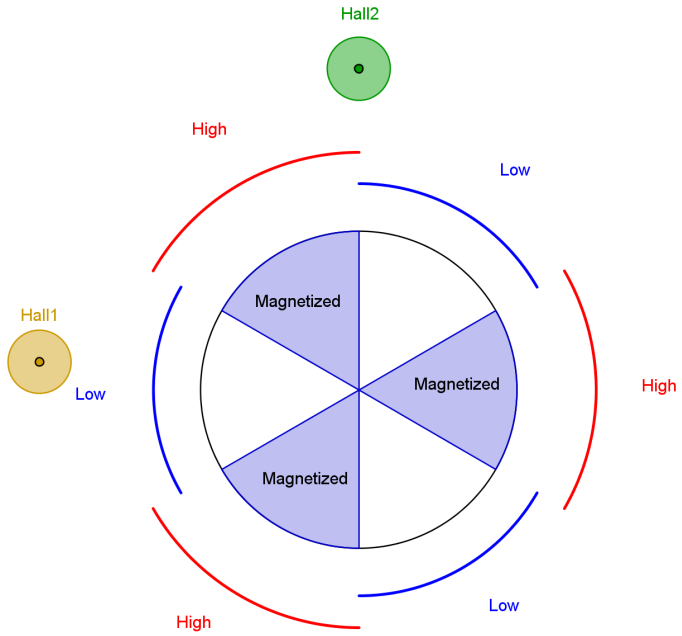


Figure 4: This is an example of how the Hall effect sensors is placed around the disk.

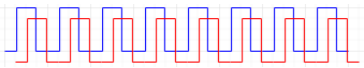


Figure 5: As the axis rotates, the output changes

8.3 Implementation

The Hall counter is the sensor block. Every time the Hall effect sensors gives an output the Hall counter is updated. There are 1080 "tachs" or changes in the output of the Hall effect sensor, for one revolution. To be able to move both clockwise and counterclockwise, the starting position i set to half of the total: 540. This i done to avoid working with negative values, as that can contribute to unnecessary difficulty in the implementation.

It starts out with sampling the value of the sensors. The value of the sensor is read and shifted into a vector of two bits as seen in figure 6

If no change have been registered, the position doesn't change and a new sample is taken. As seen in figure 7. If a

```
if rising_edge(CLK) then -- Sample and shift
   debounce_hall1 <= debounce_hall1(0) & HALL1;
   debounce_hall2 <= debounce_hall2(0) & HALL2;
end if;
```

Figure 6: An example of how the the value is sampled and shifted

change is detected then a series of “if statements” determines whether the change in position is positive or negative.

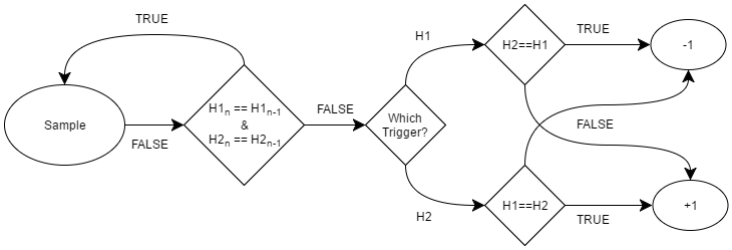


Figure 7: Flowchart of how the position changes in relation to different Hall effect sensor outputs

A small example of the of the code involved can be seen in figure 8

```
if falling_edge(CLK) then -- Calculate pos
   if debounce_hall1 = "01" then
      if HALL2 = '1' then
            pos := pos + 1;
      else
            pos := pos - 1;
      end if;
```

Figure 8: Small bit of code that shows how the position is calculated based on the Hall effect sensor output

8.4 Test

To make sure the code behaves in the intended way a test bench have been made. It simulate the codes response to changing output from the Hall effect sensors. The first two bars show the Hall effect sensors and their changing values. The next bar is the clock and under that one the position.

As can be seen in figure 9, the position is steadily decreasing in value. If the position reaches the max value (108010 or 00000100001110002) the position is set to 0. If the position reaches -1 it is set to the max value minus one.

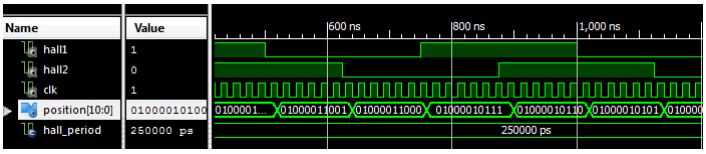


Figure 9: Testbench of how the position changes based on Hall effect sensor outputs

8.5 Discussion

As there is only two Hall effect sensors and the max rate of change in their output is a lot less than the possible sampling rate, it is fairly easy to calculate the change in position of the motor. However having only two sensors and only 6 different magnetization levels limits the accuracy of the measurements. If the axis of the motor moves in the area between a rising or falling edge but never reaching one on either sensors the change in position cannot be measured. More sensors would give a higher precision but also require more calculations in order to determine whether the change is positiv og negativ. One more error that can be experienced is if the axis end up in a position where one of the sensors is "stuck" on a rising or falling edge. If that happens the position will constantly flip between the two values. Lastly as magnets are used they sometimes "stick" to a certain position. All this combined with the slack in the belt transferring the rotation from the motor to either the pan or tilt and the small distance before the teeth of the belt and gear meshes, this limits the overall precision of the system

8.6 Conclusion

The component works as expected and sends the current position of the system to the comparator block.

9 SPI slave block

This section will describe in detail how the SPI slave in the FPGA receives and transmit data according to the description of the SPI communication protocol mentioned before.

9.1 Description

The SPI slave has two different tasks. The first task is to exchange data with the SPI master in the microcontroller. The second task is to store the information that is received and present it to the components that need the data. The SPI slave could be called also some sort of information bank for the driver.
The data exchange part follow the SPI communication protocol mentioned before. The information bank part is made by having the block hold the information in the actual output of the block. This is done so that the data is constantly supplied to the block in need of the data. However this requires a lot of data paths, but this is not much of a problem since this is going on internally in the FPGA. The reason the data is stored in the SPI slave component was also to make it more manageable since all of the data is stored at the same place.

9.2 Implementation

The SPI slave could have been implemented as a state machine. However because of the rather simplistic nature of the SPI communication protocol it was chosen not to implement it as a state machine. Instead it was implemented as a series of different events that trigger different responses in the SPI block. The first event is the S_CLK pin is having an rising edge. This event makes the the SPI slave read the input from the MOSI pin and prepare the next output for the MISO pin. The code that handles this event can be seen in figure 10.

```
if S_CLK_event = "01" then -- S_CLK rising edge

   if S_CLK_counter = 10 then -- Received all of address
      case input_shift_register(13 downto 11) is
         when "000" =>
            output_buffer <= "000" & CURRENT_POS_TILT; -- Swap output buffer with current position tilt

         when "001" =>
            output_buffer <= "000" & CURRENT_POS_PAN; -- Swap output buffer with current position tilt

         when others =>
            output_buffer <= (others => '0'); -- Set output buffer to default
      end case;
   end if;

   if S_CLK_counter /= -1 then -- If last entry, move to input buffer
      input_shift_register(S_CLK_counter) <= MOSI; -- Put data in shift register
      S_CLK_counter <= S_CLK_counter - 1; -- Decrement counter
   end if;

end if;
```

Figure 10: An S_CLK code example

The second event is when the SS pin has a rising edge. This signifies that the transmission is over and that all of the packet should be sent. This event triggers the SPI slave to update the data received on the correct output according to the received address for the data. An example of the code doing this can be seen in figure 11.

```
if SS_event = "01" then -- SS rising edge
   S_CLK_counter <= 13; -- Counter is reset

   case input_shift_register(13 downto 11) is -- Send data to address
      when "000" =>
         TARGET_POS_TILT_temp <= input_shift_register(10 downto 0);

      when "001" =>
         TARGET_POS_PAN_temp <= input_shift_register(10 downto 0);

      when "010" =>
         MAX_SPEED_TILT_temp <= input_shift_register(7 downto 0);

      when "011" =>
```

Figure 11: An SS code example

Under the data transaction two temporary registers are used, one for the input and one for the output. This is done to stabilize the data send so that it does not change mid transmission which would cause the data to become corrupt. There are 4 different kinds of settings that can be controlled via the SPI slave.The first setting is the target position used in the controller block, the second setting is the maximum speed for the motor, the third setting is the minimum speed of the motor and the last setting is the enabling of the motor. All of these 4 settings can be set for the pan and tilt part separately. This results in 8 different values needing an address resulting in an address length of 3 bits. Since the data length has to be constant, the data length was chosen according to the largest value needed to be sent. This was the target position, needing 11 data bits hence the data length was chosen to be 11 bits.

The output from the SPI slave to the SPI master was chosen to mimic the address and then send some data according to the address received under the transmission. This means that the output buffer has to be changed under the transmission. This would be a problem in most cases, however since the FPGA works much faster than the microcontroller this is not a problem. Since the only information the SPI master is interested in is the current position, the output buffer is only changed when this information is polled. It was made so that this information was always polled when a target position was sent, meaning that if only a poll on the current position is wanted the target position is resend with the same value. In all other scenarios then the master sending the target position, the value returned to the master is the address the master sent and a data containing zeroes. The code handling this can also be seen in 10.

9.3 Tests

The SPI slave component has been tested in a Xillinx ISE testbench. This was done by simulating a transaction to the SPI slave. In figure 9 the test bench can be seen. SS can be seen going low signifying the start of the data transfer.

The S_CLK then starts to toggle and the data is shifted into the input register and as SS goes high the data from the input register is set to the placement the address is pointing to. In this case, a value to the target of the pan is being send. This has the address of 001 which can also be seen on the three first bits on in the input register.

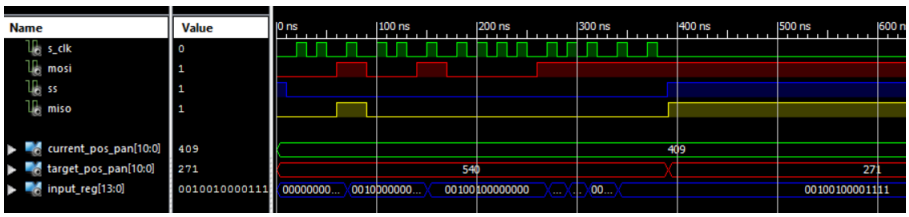


Figure 12: An Xilinx testbench simulation of the SPI

9.4 Discussion

One of the features you would get from creating the SPI slave component as a state machine is more robustness against errors. When an error happens in a state machine it would be unable to make the same amount of mess because it is locked to the state. However since no error should be happening and the fact that at no time while testing the connection was an error made, it is not really of any concern.

9.5 Conclusion

Both the testbench and physical test with the FPGA and microcontroller showed that this component functioned as it should.

10 Driver conclusion

Even though the driver runs quite smooth some changes could have been made in order to make it better. One of these changes would be to make a controller that actually follows the control theory for a PI controller a little better. The reason why this was not done from the start was because of a lack of understanding how a controller could be implemented. As an example it was discovered rather later in the development stage that the optimal way of implementing the PI controller would have been by making a filter. This was however discovered too late to change. A filter was created but an unknown error shut down the idea of implementing it as a filter since the time did not allow it to be completed. However if more time was given implementing the controller as a filter would be one of the upgrades that would make the product easier to tune because the filter follows control theory closer then the implemented controller. Another thing would be to implement the controller as a PID controller instead since this would make a smoother controller. The control theory section also derives that the optimal controller would be the PID. The easiest way to implement the D part would be to make a speed sensor, this was however discovered late in the development stage after a failed attempt at implementing the D part through standard derivative mathematics. An error was also found in the link between the controller block and the motor driver block since the controller is actually based on controlling a voltage rather than the duty cycle. This error can be fixed by making a conversion from voltage to duty cycle in the controller block.

11 SPI communication

This section will explain how the connection between the FPGA and the microcontroller is made via an SPI communication protocol. This section can be divided into two parts with the first part concerning the general SPI communication and the second part concerning addressing of the data sendt. Note that the second part has, in principle, nothing to do with the actual SPI communication, however since it is important for both the FPGA microcontroller it makes sense to talk about it bore going into detail with the FPGA and microcontroller.

11.1 SPI communication protocol

The SPI communication protocol is used to send data, one bit at a time. The SPI communication protocol runs a master slave relationship which means that one of the parts involved is the master part and fully controls the connection and the other parts simply does what the master part tells them to do. An example of this can be seen on figure 13.

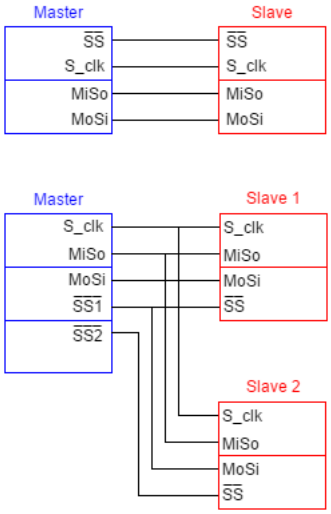


Figure 13: Diagram of how the master and slave is connected

In order to send the data four wires, called Slave Select (SS), Master out Slave in (MOSI), Master in Slave out (MISO) and Serial Clock (S_CLK), are used. The MISO and MOSI wires is where the data bits are through and SS and S_CLK are wires that controls the transfer, note that SS and S_CLK is controlled only by the master. When the SS is active a data transfer is initiated between the master and a slave, the master is basically saying to the slave that they are now communicating. When the SS is deactivated the transfer is over and all of the data is send. Note that the SS is active low which means that it is active when the SS signal is low and deactivated when the signal is high.

The S_CLK is used to signal a read of the MOSI and MISO line. This is very important since this is where the synchronization of the data transfer is made. Without this synchronization the data transfer would be impossible. An example of a SPI data transfer can be seen on figure 14.

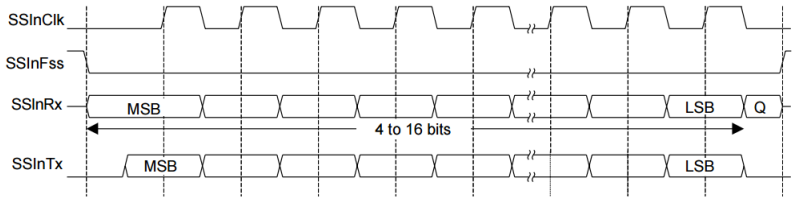


Figure 14: Illustration of how the package is constructed

11.2 Addressing

Instead of running several SPI slaves in the communication to send the data to the right part of the FPGA, giving the data an address instead was chosen. This makes the amount of data sent longer, however since the controller is a part of the FPGA the speed which information is send does not matter. So since the addressing was found to be the easier solution to make it was chosen. Addressing works by appending extra bits onto the data, much like a header. An example of this can be seen on figure 15. The address bits then define what kind of data is being send and/or where it has to go. In the case of the SPI slave in the FPGA, the address symbolizes where the data has to go, since what the data symbolizes is an information not needed in the FPGA. In the case of the SPI master in the microcontroller, the address symbolizes what kind of data was received.

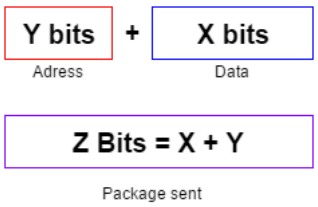


Figure 15: Illustration of how the package is constructed

When receiving a packet via the SPI communication, the receiver separates the address from the data and then uses the data according to what the address tells about the data. An example of this can be seen on figure

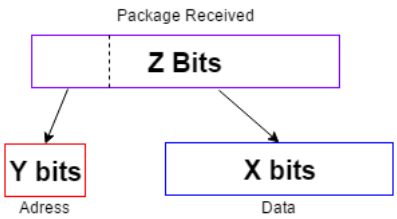


Figure 16: Illustration of how the package is destructed

12 The Plant

The purpose of these tests are to analyse and generate a blackbox model for the motor, based on its step response.

12.1 Setup

The motor was set up with a switch connected to a power supply set at the desired voltage to test. This would create a physical step response from the motor when the switch was toggled. The two encoders output on the EMG30 are open collectors, so the outputs were connected to pull-up resistance to produce a desired signal. The pull-up resistors for these tests was picked to be around 300 ohm.

An oscilloscope was set to measure the two encoders outputs on its each own individual channel. The sampling starts on an edge trigger with a sampling rate of 2 mHz and stops the sampling after 0.5 seconds. These 2 set of samples would together be the data defining an step response.

Following step responses were sampled(Figure 17), and the information extracted (Figure 18) from the data was calculated in matlab. See files for actual data.

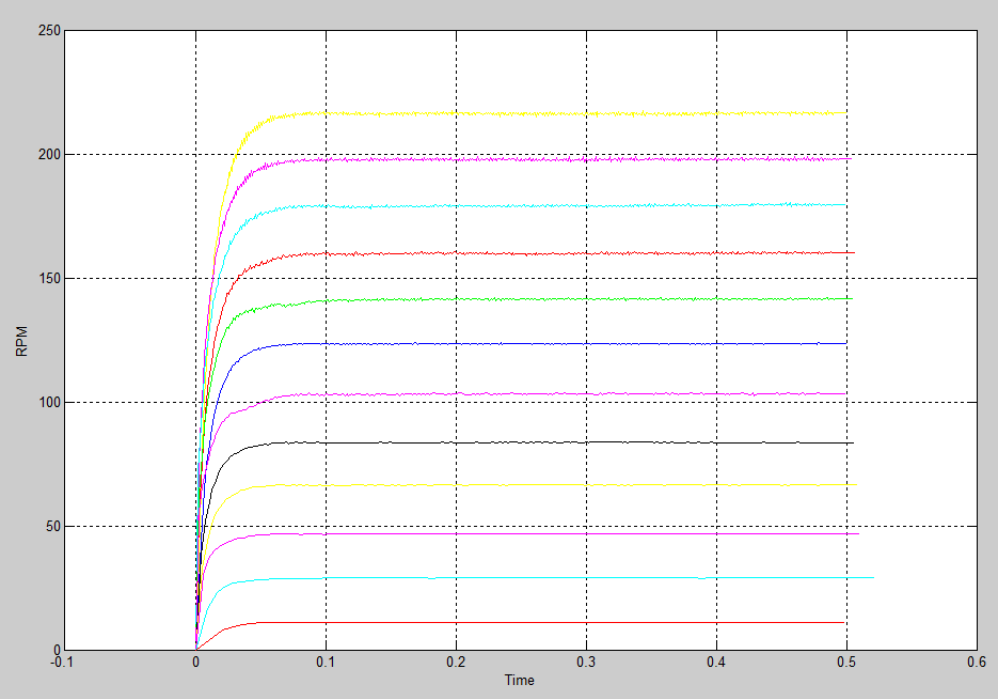


Figure 17: DC motor step responses from 1 to 12V. The bottom red line is 1V, and the upper yellow line is 12V

As noted, the step responses of the motor appear to be of a first-order linear-time-invariant (LTI) system. This means the system can be estimated as:

$$DCgain\frac{1}{\tau+1} \tag{1}$$

The DC gain is the steady state value divided by the steady state (constant) value of the input step.

No Load\Volt	Measured RPM	Rated RPM	Offset	DC gain	Time constant Tau	Tachs/s
1	10,94	18	3,269 %	10,9	0,0185	65,64
2	28,92	36	3,278 %	14,5	0,0111	173,52
3	46,9	54	3,287 %	15,6	0,005	281,4
4	66,5	72	2,546 %	16,6	0,0085	399
5	83,5	90	3,009 %	16,7	0,0077	501
6	103,3	108	2,176 %	17,2	0,0053	619,8
7	123,4	126	1,204 %	17,6	0,0096	740,4
8	141,4	144	1,204 %	17,7	0,0068	848,4
9	160,2	162	0,833 %	17,8	0,0087	961,2
10	179,4	180	0,278 %	17,9	0,0075	1076,4
11	198,1	198	-0,046 %	18,0	0,0081	1188,6
12	216,4	216	-0,185 %	18,0	0,0108	1298,4
Avg			1,738 %	16,6	0,0090	

Figure 18: Table summarizing the information from the DC motor tests.

$$\frac{216}{12V} = 18\frac{rpm}{V} \tag{2}$$

This indicates that RPM is directly proportional to a voltage. The expected proportionality (in blue) is plotted below together with the actual measured RPM (in green) in figure 19

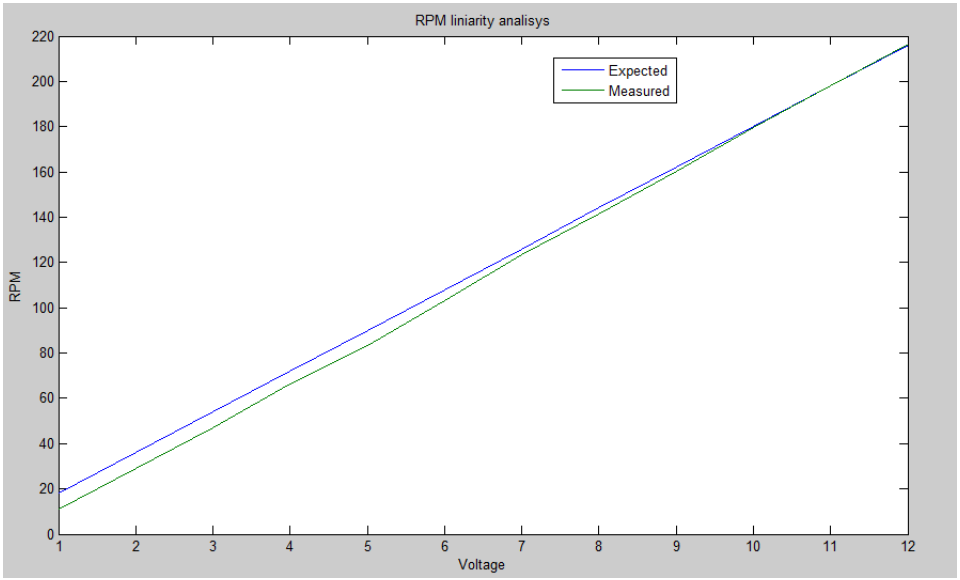


Figure 19: Comparing measured RPM values to expected ones

As seen, the actual RPM of the motor is not directly proportional to a voltage, but the difference is at max a 3.287% offset from the expected. Since this is relatively low, the system is estimated to be proportional. Though at low voltages the relative DC gain at that voltage starts to differ significantly from the estimated values. Therefore the DC gain is estimated as the mean value of the actual DC gains measured for the voltages 4V to 12V, since the DC gain in this range is relatively linear. The DC gain is: $\approx 17.6\frac{rpm}{V}$

The time constant tau is the time when the step response reaches $1 - 1/e \approx 63.2\%$ of its final value. This time constant is expected to be constant, but as seen in figure 20, at low voltages there seems to be some uncertainties regarding the time constant.

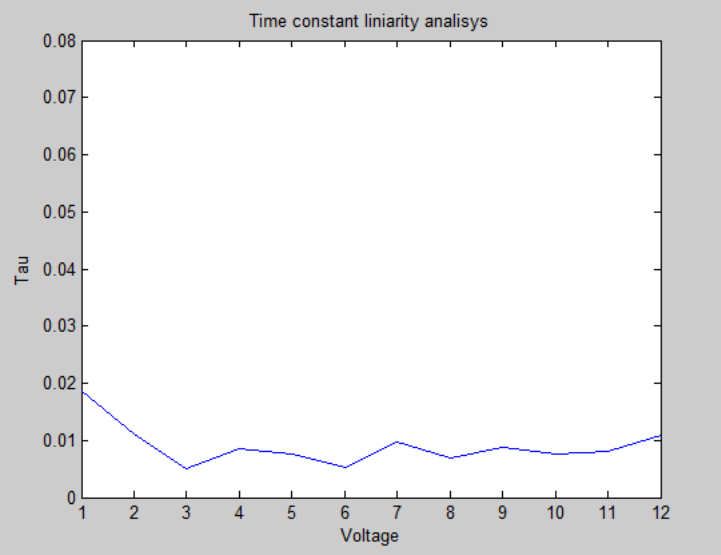


Figure 20: Time constant for the DC Motor at different voltages

This behavior can be explained by variations in the actual experiments. Since the encoders are not uniformly placed in the motor. The start position of the encoders can then have some impact regarding the experimental results. Other than that, it would be expected to see some differences from the lower voltages compared to the higher, since the DC gain of the system indicated some nonlinearity. The time constant, Tau, is then estimated to be the mean value from 1 to 12 volts $\approx 0.009 seconds$.

The transfer function for the motor is therefore now approximated as follows:

$$G(s) = \frac{17.6}{0.009s + 1} \tag{3}$$

As seen on Figure 21 the first-order approximation G(s) of the motor is relatively accurate.

12.2 Discussion

Having modelled our DC-motor as a first order LTI system, it is relevant to discuss the reliability and accuracy of this model. In reality, the transfer function for the motor includes both an external, mechanical part and an internal, electrical part (the armature).[5]

$$\frac{\theta(s)}{V_f(s)} = G(s) = \frac{K_m/(bR_f}{s(\tau_f s + 1)(\tau_L s + 1)} \tag{4}$$

This model is actually a second order system. However the time constant Tf, denoting the response time of the internal electronics of the motor, is negligible compared to the mechanical dynamics of the motor. This makes

[5]Model and transfer function from Control Systems: Twelfth Edition p. 72

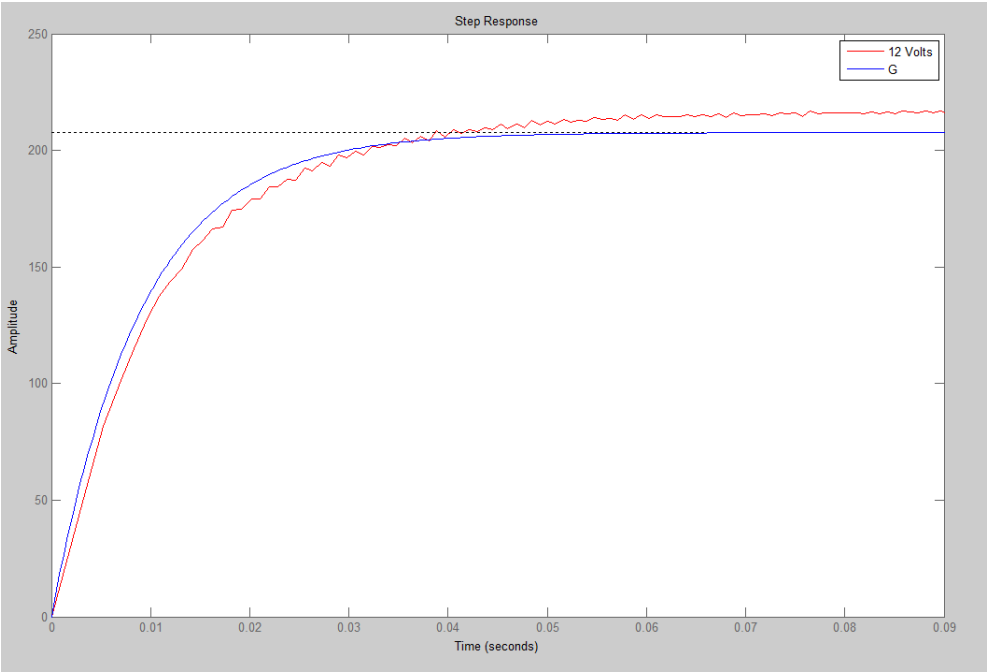


Figure 21: Comparation of the first order approximation step response and the actual step response from research.

sense both intuitively and mathematically, as the relatively faster armature with an associated pole further away from origin will have less of an impact on the overall system response, and can thus be omitted for the sake of simplicity. Typical values for these time constants show this aswell.

The first and second order transfer functions generated using these time constants are close to identical, as can be seen in figure 22. The motor constants used:

Field time constant: $\tau_f = 1ms$

Rotor time constant: $\tau = 100ms$ [6]

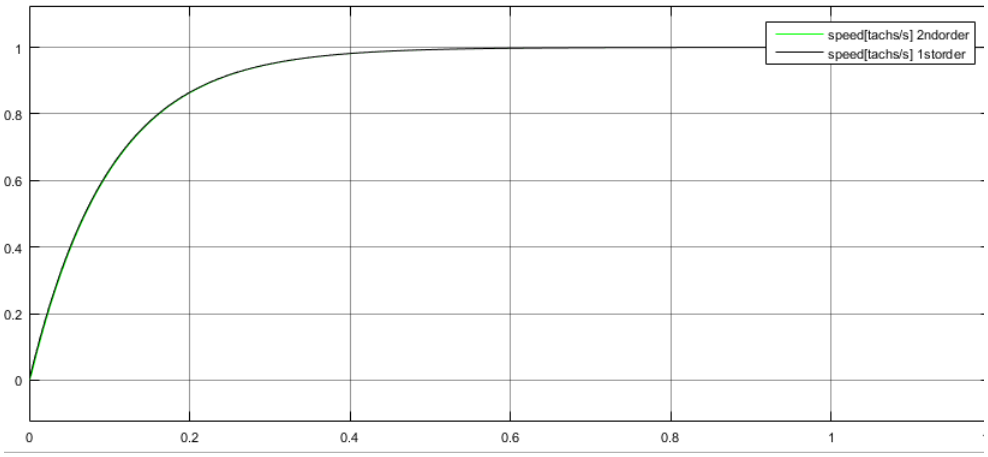


Figure 22: Comparing 1st and 2nd order transfer function step responses.

Omitting the armature from the derived transfer function gives us the following, which essentially equals our first

[6] from Modern Control Systems: Twelfth edition p77.

order estimation.

$$\frac{K_m/(R_a b + K_b K_m)}{s(\tau_1 s + 1)} \tag{5}$$

12.3 SI-units for DC Gain

Looking at the model in the larger context of our control system, it is evident that using RPM/Vas the SI-unit for DC-gain is less than optimal. The feedback from the Hall sensors is in the form of tachs, and would thus require a conversion in order to be interpreted as motor rotations. In addition to that, the controlled variable of our system is the position of the motor in terms of tachs. The transfer function can easily be modified to reflect this through an integration, effectively making our transfer function a second order system.

Speed:

$$G(s) = \frac{105}{0.009s + 1} \frac{[tachs/sec]}{[V]} \tag{6}$$

Position:

$$G(s) = \frac{105}{0.009s^2 + s} \frac{[tachs/sec]}{[V]} \tag{7}$$

The behaviour of these transfer functions given a step input can be seen in figure 23.

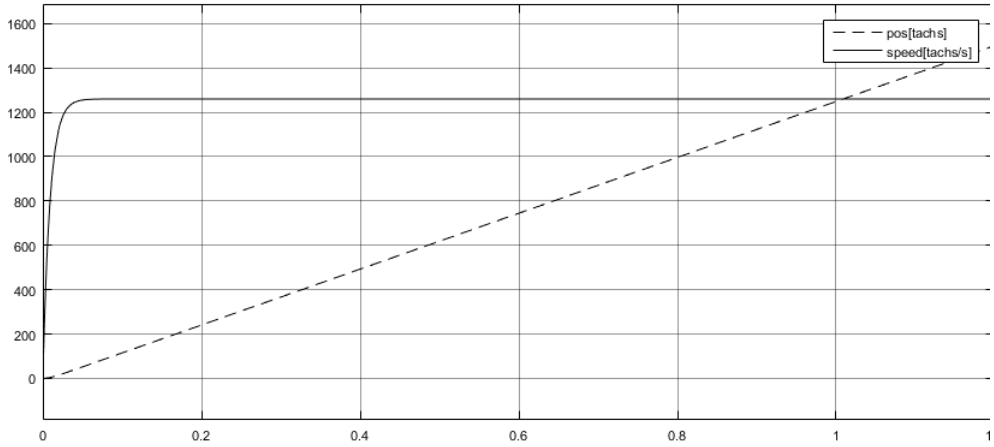


Figure 23: speed & position simulation given a 12V input.

12.4 Inertia in the P&T system

Because of inertia present in the pan and tilt system parts, it can be expected that the constants of the individual motors will look different than those describing the motor without load. Also, because transfer function of the DC motor without load is approximated to be linear, we assume the same from the specific transfer functions of the P& T motors.

12.5 Tests of the P&T motors

Step response tests were applied to the motors on the actual Pan and Tilt system. Because of restrictions on the pan part of the system, voltages tested ranged from 3V to 8V. The data has been analysed to find values of constants related to the method of approximating a plant of a DC motor.

Tilt Load/Volt	Measured RPM	DC gain(tach/s/v)	Time constant Tau	Tachs/s
3	34	68,0	0,0649	204
4	51	76,5	0,0713	306
5	69	82,8	0,0671	414
6	85	85,0	0,0688	510
7	105	90,0	0,0707	630
8	125	93,8	0,0761	750
Avg		82,7	0,0698	
Pan Load/Volt	Measured RPM	DC gain(tach/s/v)	Time constant Tau	Tachs/s
3	12	24,0	0,1353	72
4	31	46,5	0,2136	186
5	41	49,2	0,2172	246
6	59	59,0	0,2390	354
7	75	64,3	0,2323	450
8	92	69,0	0,2326	552
Avg		52,0	0,2117	

Figure 24: Table of P&T step response data

As seen below in figure 25, the step response spikes at first. It is assumed this spike occurs primarily because of the following:

- In the gear network connecting the actual motor and the pan or tilt part, there are small gaps between individual teeth. The motor is then switched on and while the collective gap closes, the motor runs as if it had no inertia.When the gap is closed the motor is slowed drastically because of the inertia of the pan or tilt part.

The two things together create the spike we see on GRAF below. Because we assume linearity of the actual systems, we disregard the spike and look at the graphs after the spike has ended to find Tau.

Applying the method for creating a plant for a DC motor, discussed earlier, the actual constants for the pan and tilt systems are used in this context to create dedicated transfer functions for each motor in the P&T system. The plants can be seen as equation 8 for the pan part and equation 9 for the tilt part:

$$G_{pan}(s) = \frac{52}{0.212s + 1} \tag{8}$$

$$G_{tilt}(s) = \frac{82.7}{0.07s + 1} \tag{9}$$

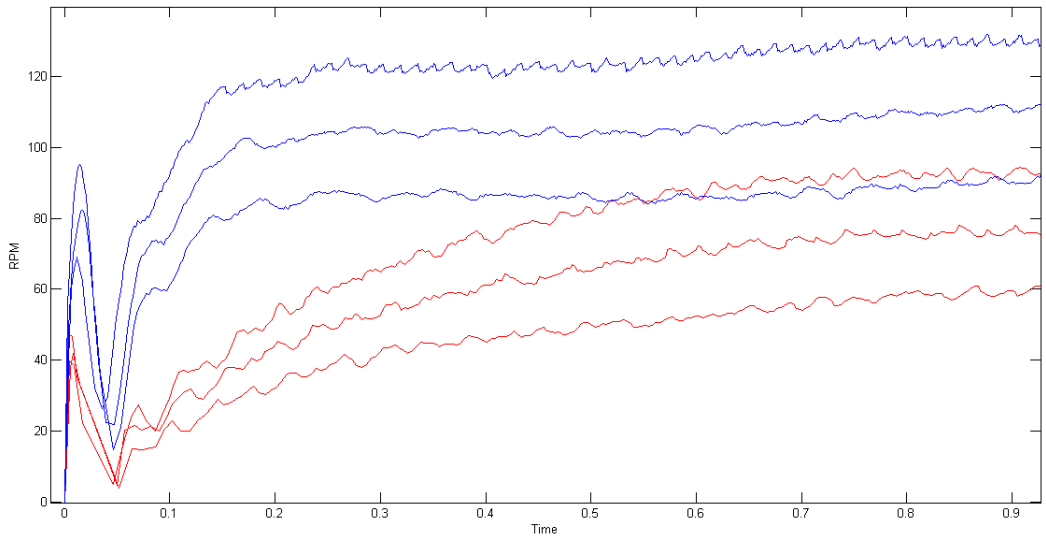


Figure 25: The P&T response to 6V-7V-8V. Blue: Tilt, Red: Pan

13 PID

13.1 Introduction

A controller is an essential part of most autonomous systems. This project is no exception - having control over where the pan and tilt system is pointing at is crucial to meeting the specified requirements. The following questions might then transpire.

- What controllers are available?
- Is a complex or simple controller desired?
  - Pros and cons?
- How do you determine which one to use?
- How do you implement them?

When discussing what controller to use, figure 26 is the point of reference, where the plant is the previously estimated transfer function, **SKRIV FORMLEN** EQUATION REF, of the motor of the system.

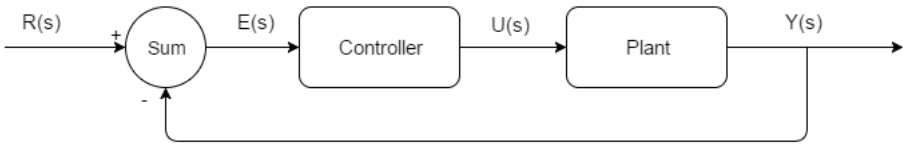


Figure 26: A generic way of representing a close-looped control system

13.2 The Controller

For simplicity's sake Proportional (P), Proportional-Integral (PI), Proportional-Derivative (PD) and Proportional-Integral-Derivative (PID) be the only controllers taken into considerations for this project. Though Lead-Lag compensators could be considered as well, since in essence they can do the same as before mentioned controllers.

A PID controller is made up by three parts: the proportional gain looks at the current error, the integral looks at the past errors and the derivative looks at current rate of change. Each term has a tuneable gain called the k constants. In general these constants are what defines a PID controller's behavior. See figure 27 for illustration.

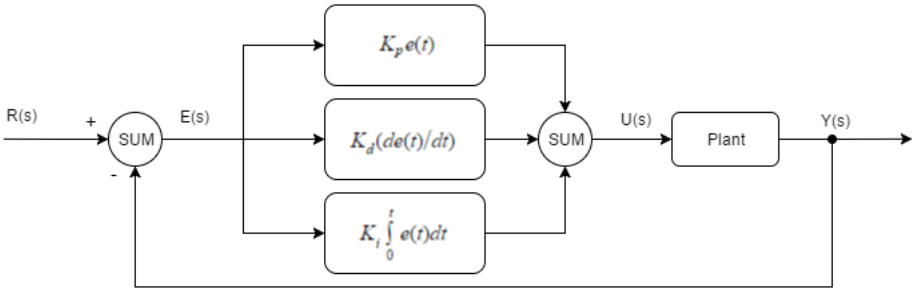


Figure 27: A representation of the PID controller principle

This is the equation of the standard PID controller in the time domain:
$K_p e(t) + K_i \int_0^t \mathrm{e}(t)\,\mathrm{d}t + K_d \frac{de(t)}{dt}$

If transformed to the s domain the equations looks like this:

$$G(s) = K_p + \frac{K_i}{s} + K_d s \tag{10}$$

13.3 Proportional Controller

The P controller is by far the most straightforward controller, being that it is simply adding a gain to the open-loop transfer function. A P controller is a desired controller, since it is very simple to implement and tune, though this project have to meet the stated requirements.

To help analyse and give some intuition about how the motor's transfer function behaves, a root locus of the open-loop transfer function is plotted see figure 32. As one would expect it seems like this kind of controller will not suffice for the project.

As seen in figure 28, the step response of the closed-loop transfer function, has an overshot of 14.1 percent, and a settling time of 0.055 seconds. Meeting the specified requirements for the project with this controller is not going to happen. The overshot can be decreased by reducing the gain below 1, but this induces a longer settling time and vice versa.

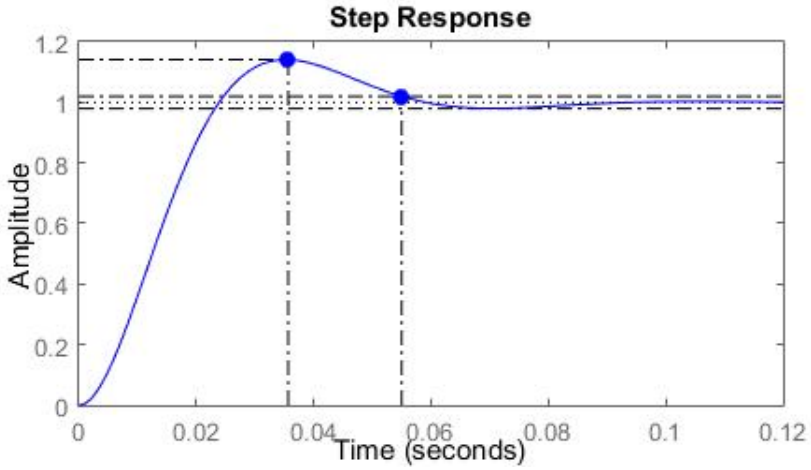


Figure 28: Peak amplitude: 1.14 - Overshot(%): 14.1 - At time(seconds): 0.0356 Settling time (seconds): 0.055

13.4 Proportional-Integral Controller

The PI controller accumulates past error terms over time to eliminate steady-state errors in the system. This can cause an increase in overshoot called the integrator wind up, and an increase in settling time. Tough if a requirement is complying a steady-state error of 0, then it is up to the designer to decide, if reaching state-state is more vital to the system, than having more overshot and a longer settling time.

Using a PI controller is the same as adding a pole and a zero to the open-loop transfer function. As seen in figure 32, this controller does not seem to live up to the requirements either. Placing the zero and pole differently could make a better controller, but it would never be able to adjust to both design criterias.

The step response of the PI controller is seen on figure 29, the only difference from the P controller is a slightly longer peak time and settling time and a larger overshoot, as expected. This controller would not meet the requirements.

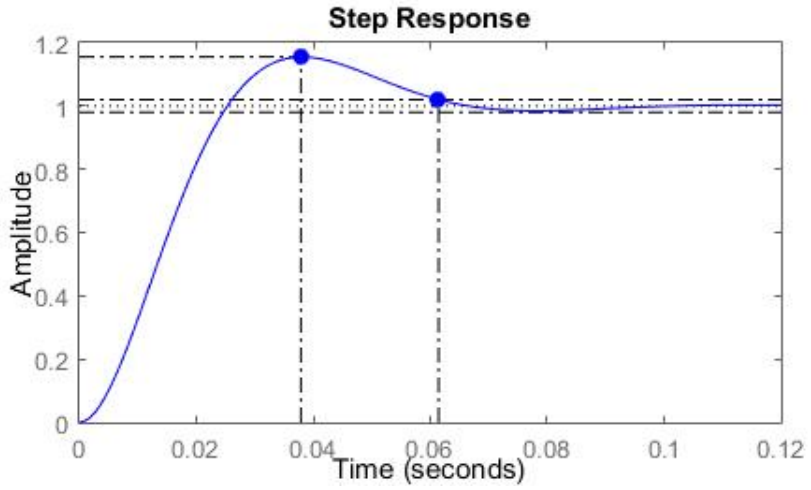


Figure 29: Peak amplitude: 1.15 - Overshot(%): 15.5 - At time(seconds): 0.0379 Settling time (seconds): 0.0614

13.5 Proportional-Derivative Controller

A PD controller is equivalent to the addition of a simple zero, **INSERT EQUATION HERE**, which improves the transient response. From a different point of view, the PD controller may also be used to improve the settling time or stability, because it anticipates large errors and attempts corrective action before they occur.

FIGURE5 shows a promising root locus of the PD controller, it seems like that with the right amount of gain the system would fulfill the requirements.

The step response on figure 30 shows, that in fact this controller would be ideal for this project. There is no overshot, and the settling time is well below 0.05 seconds.

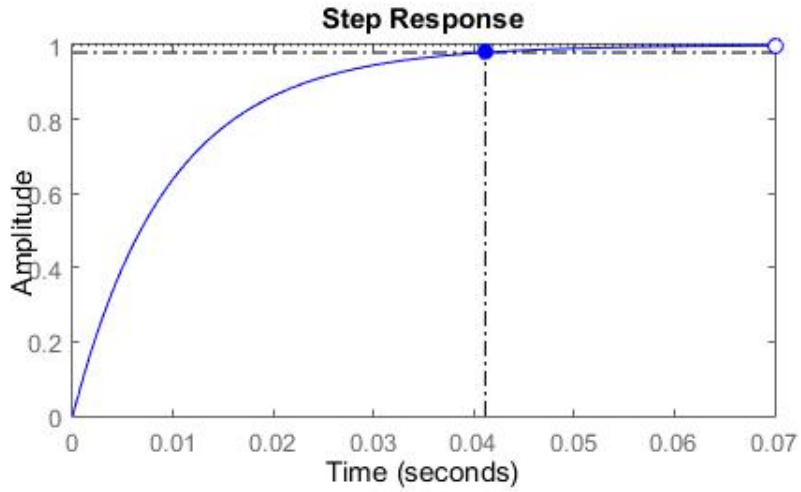


Figure 30: Peak amplitude: $>= 0.998$ - Overshot(%): 0 - At time(seconds): > 0.07 Settling time (seconds): 0.0411

13.6 Proportional-Integral-Derivative Controller

The PID controller is the most complex controller which is going to be discussed. As noted earlier an PD controller was sufficient for a controller. So why discuss the PID? The reason is that, in reality systems might have limitations which analysis have not taken into account yet. Reaching a steady-state error of 0 is actually a problem with a PD controller for this project, since the pan and tilt system does not function below a certain range of voltage, which means that an integral is needed.

Notice on figure 32, that an PID controller is the addition of 2 zeros and 1 pole to the open-loop transfer function. The root locus shows that this system with a relative gain still meets the requirements of the design.

The step response of the PID controller, figure 31, shows that indeed a PID controller would be a good choice of controller for this project. There is well below 10% overshot and the settling time is a great deal below 0.05 seconds.

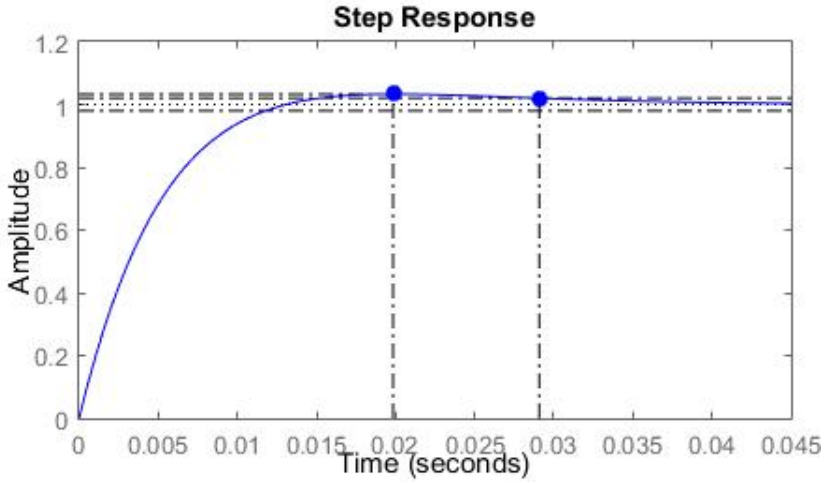


Figure 31: Peak amplitude: 1.03 - Overshot(%): 2.92 - At time(seconds): 0.0201 Settling time (seconds): 0.0277

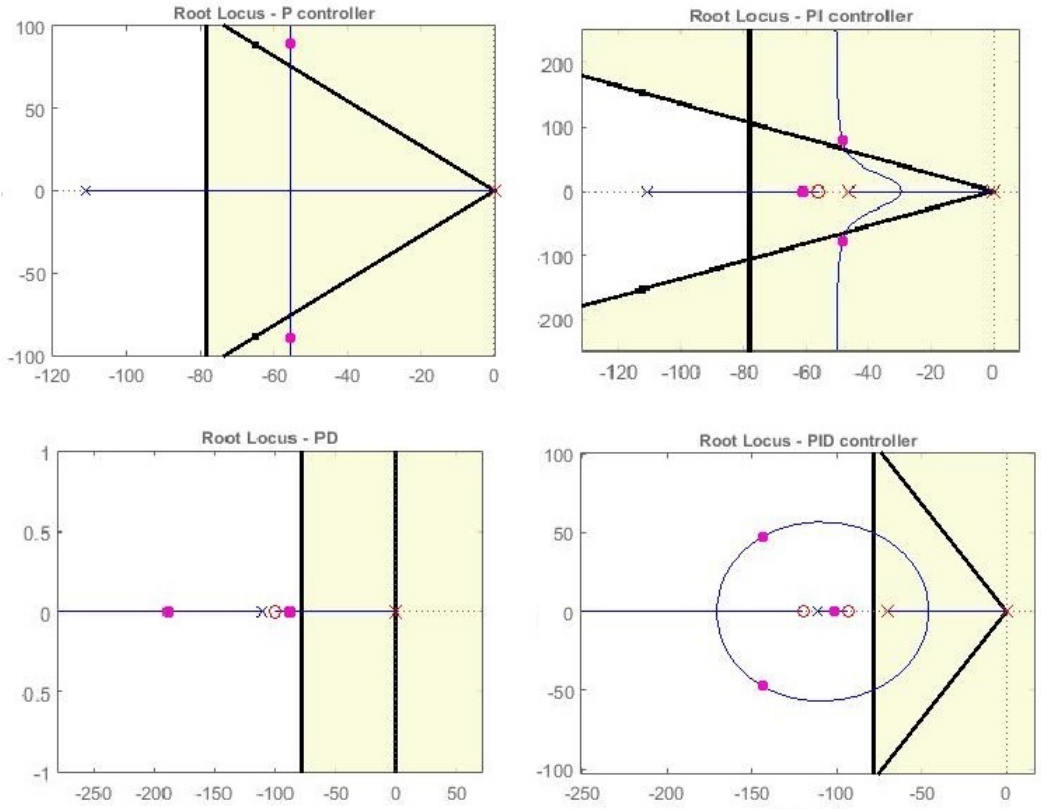


Figure 32: Open-loop transfer function Root Locus plots of the estimated motor transfer function, EQUATION REF, with different controllers. The shaded areas indicates designs not met if poles of the closed-loop transfer function is placed there. The designs are: Overshot $>$ 10%, settling time ¡ 0.05 seconds.

13.7 K constants

Once either a P, PI, PD or PID controller has been chosen for the project, the next step would be to estimate the k constants of the respective controller. Finding these constants to meet one's requirements can be done in various ways. In this section the Manual tuning and Root Locus method will be discussed.

13.8 Manual tuning

This method requires the system to be online and running to tune it, though if a simulation of the system has been made this tuning method is still viable on the simulation - the catch is that a simulation's artificiality cannot be denied, thus the actual online system proves more truthful.

There is no right way to tune manual, since experience can lead to a quick tuning. A known procedure is, however, to first set one's k_i and k_d to zero, and then increase the k_p until the system starts to oscillate, k_p should then be set to approximately half this value. The k_i is now increased to correct for any offset within the desired time. Lastly, if needed, the k_d is increased until the requirements are met.

13.9 Root Locus tuning method

This method is a mathematical approach with the help of software like MatLab to find k_p, k_i and k_p from the root locus. It is not required to have the online system up and running to use this method, which in many cases can prove useful (imagine a system where unpredicted behavior can have fatal consequences). Though a mathematical model of the system is needed.

The root loci on figure 32 can be used, since they are the open-loop transfer function of the system. It has already been established that using a PID controller would be a good choice to meet the project's requirements. Therefore this example is based on the PID root locus from figure 32. The root locus was made with the SISO tool from Matlab with the open-loop transfer function EQUATION REF. The two zeros at -90 and -55 on the real axis and a zero at the origin is what makes up the PID controller, and they were placed there to satisfy the design requirements. The SISO tool Compensator Editor of this root locus shows these zeros and pole as an compensator, which is another way to represent a PID controller. In the example of the PID controller from figure 32 the compensator looks like this:

$$200 * \frac{(1 + 0.011s)(1 + 0.019s)}{s} \tag{11}$$

Rewriting equation 11 to look like equation 10 makes it quite easy to read the K constants.

$$\frac{0.0396s2 + 5.8s + 200}{s} \tag{12}$$

As seen in 12 the K constants for the PID controller are K_P = 5.8, K_I = 200 and K_D = 0.0396.

13.10 Controller constants for Pan and Tilt systems

Applying the root locus method to the transfer functions for the Pan and Tilt systems, PD and PID controller constants were derived for both of these.

$$G_{pan}(s) = \frac{52}{0.212s + 1} \tag{13}$$

Skriv konstanterne ind i latex

13.11 Simulation

Making a simulation of the system is a good way to figure out whether the controller needs to be redesigned or tweaks needs to taken into consideration. Since the motors on the pan and tilt are expected to have similar characteristic as the load free motor (though pan and tilt are expected to have different time constants), the simulation is then first done on the load free motor transfer function with the PD controller from figure 32. The K constants were found with the same method as discussed in the Rooth Locus tuning Method section. The constants are K_P = 2 and K_D = 0.02. The simulation is done in MatLab's Simulink software, see figure 33. the Step block is a step input of 1, and since the DC gain of the plant is in the units of $\frac{tacs}{s*V}$, this simulation illustrates the system rotating the rotor 1 tach. The step input is fed to the feedback summation, where the feedback is subtracted from the reference input. The summation result is now the error signal, and this signal is then fed into the PD controller. The PD controller reacts depending on the rate of change of the error signal and passes on its calculated value. The motor plant produces physical rotation, which is integrated to tachs. The output is now sampled as feedback and is then fed back to the reference signal. The Time Scope will generate the plot of the simulation.

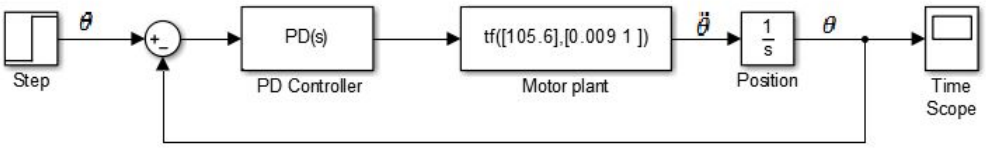


Figure 33: A simulink simulation of a PD controller of the load free motor's transfer function.

The step response of the simulation is expected to look like the step response from the PD analysis. As seen on figure 34 looks a lot like expected. Upon inspecting the time 0.0226 seconds, the amplitude is 0.982 which is within 2% of the settling time, so the simulation is very accurate to the expected.

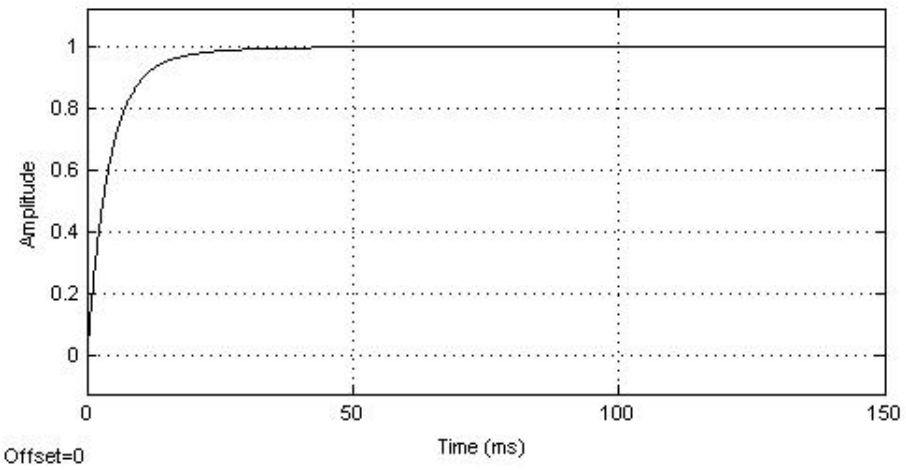


Figure 34: The step response of the simulation from figure 33

Since this is a simulation over the physical system, then the physical aspects that are obvious should be taken into

consideration. The motor of the system cannot take inputs higher than 12 volts, therefore a saturation limit will be added to the PD controller to simulate this. Other than that, the friction and inertia in the system makes it impossible for the system to move below 2 volts. This means that inputs ranging from -2 volts to +2 volts to the plant, must always be set to 0 to simulate the idling of the motor. On figure 35 these changes can be seen.

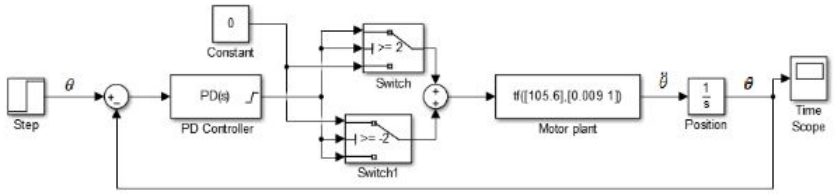


Figure 35: A Simulink simulation of a PD controller of the load free motor's transfer function with saturation of -12 and +12 voltage and constraints with output 0 at voltage range -2 to +2 volts .

The step response of figure 35 is seen on figure **??** and as it is seen that the motor idles before reaching the target. This means in reality the motor never reaches the target, and since this is a secondary requirement this is something that would be nice to compensate for. Luckily an PID controller has been discussed and the constants for it is already known, and this controller is ready for simulation. On is FIGURE14 the step response of the simulation of the PID controller, with the constants found in the Root Locus tuning method section, in the same setup as FIGURE12. The step response shows that with a PID controller the simulation now reaches the desired target while still having a settling time below 50 milliseconds. In conclusion a controller for the load free motor has been designed and simulated to work for the desired requirements.

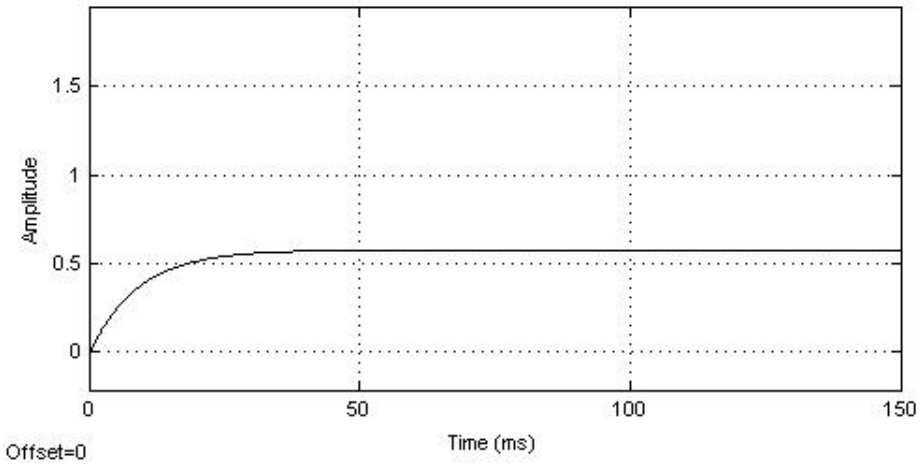


Figure 36: The step response of the simulation from figure 35 with a PD controller

Since the simulation of the load free motor needed a PID controller to meet the requirements, the pan and tilt motor on the actual system is expected to be of same behavior, and would also need a PID controller. Also the system constrains on the system might prove of a problem for the PID controllers of the pan and tilt, since the saturation and the deadband between -2 volts and +2 volts, has not been take into considerations while designing the control system. The transfer functions of the pan and tilt were simulated in the same manner as 35 with both PD and PID controller constants shown in the **Controller constants for Pan & Tilt systems section.**

On figure 39 the step responses of the pan motor with PID and PD control can be seen, it is clear that, as expected,

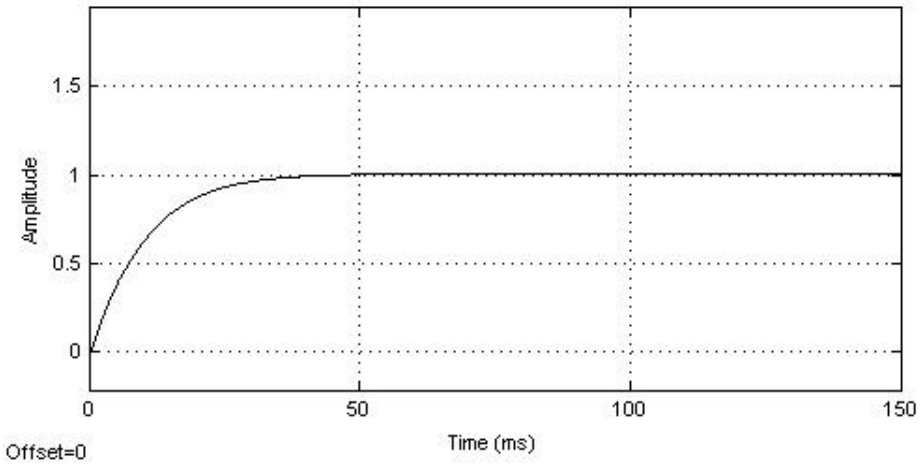


Figure 37: The step of the simulation from 35 with a PID controller

the PD control does not reach an acceptable steady state, while also being alot slower than the required settling time of 50 milliseconds. The PID control however reaches a steady state acceptable, nonetheless the settling time of 0.689 seconds is also too slow, also there is an overshoot of 11.2The slow settling time and overshoot can be explain by the saturation of the plant, and the deadband between -2 volts and +2 volts. Because the controllers designed does not compensate for these factors.

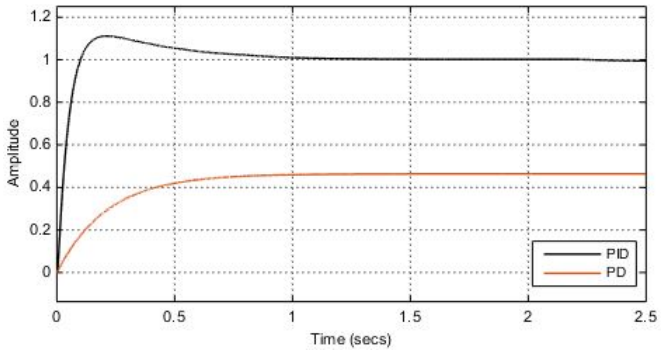


Figure 38: Step response of the simulation from figure 35 with pan motor and results of both PD and PID controllers.

Likewise on 39 the step responses of the tilt motor with PID and PD control can be seen. As expected the PD control never reaches an acceptable steady state, and the settling time is slower. The PID controller reaches an acceptable steady state, but also have a long settling time of 0.491, while also having an overshot of 18.7Same as for tilt motor, these behaviors can be explained by the saturation of the plant and the deadband between -2 volts and +2 volts.

Another simulation on the pan and tilt motors were run. This time an step input of 540 was set. This is equal to half a revolution on either motors. This simulation was run, because in practice the system should be able to handle the kind of inputs. On FIGURE17 the step response of 540 can be seen, as noted the pan and motors overshoot with about 100% and the settling time is greatly increased because of this. This happens because of integrator windup, and was not compensated for when designing the controllers. In the simulation an anti windup can be set, to simulate an implementation of windup protection. On FIGURE18 the result of a step input of 540

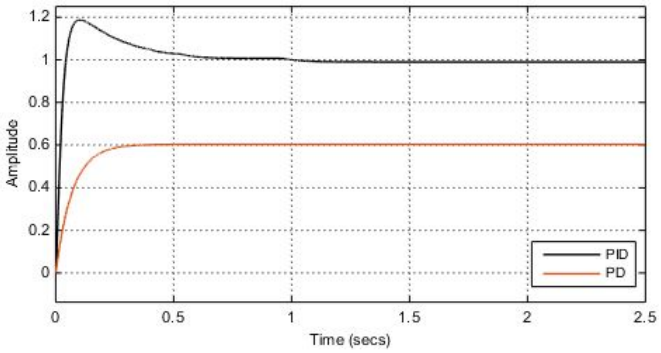


Figure 39: Step response of the simulation from 35 with tilt motor and results of both PD and PID controllers.

with windup protection on can be seen. The pan and tilt on FIGURE18 seems to behave acceptable, the tilt has no overshoot and the pan has an overshot of 3.4 % overshoot. Though the settling time is still slow, however this is to be expected as the system is saturated.

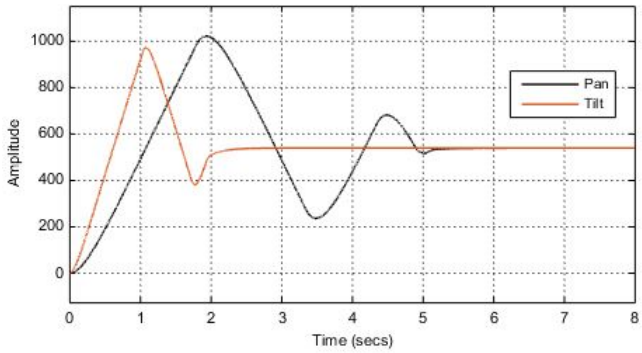


Figure 40: Step response of the simulation from figure 35 and 540 input instead of 1. The pan and tilt motor are without wind up protection.

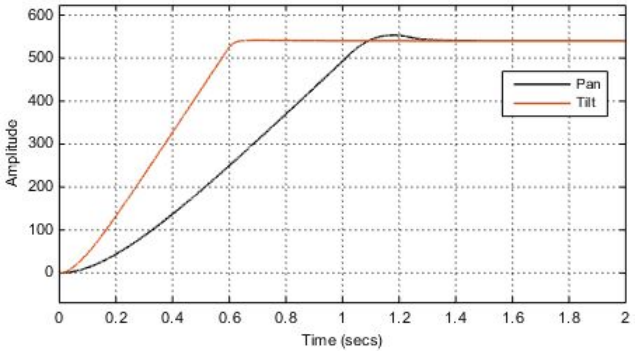


Figure 41: Step response of the simulation from figure 35 and 540 input instead of 1. The pan and tilt motor are with wind up protection on.

The transfer functions and the controllers for the pan and tilt have now been tested in simulation, and the designed controllers seems promising. The tilt motor would on a small step input not fulfil the requirements of less than 10% overshoot and the settling time requirement is not met either. It would seem like that reaching a settling time of 0.05 seconds or less would be impossible for the tilt, since the system cannot move fast enough. Tough for a step input of a larger scale, the overshoot is nigh zero and well within the requirement. As for the pan motor, a

small step input has an overshoot really close to 10%, but a long settling time. The requirement of having overshoot less than 10% is therefore not fulfilled, either is the having a settling time of 0.05 seconds for the pan motor. On a larger input the pan motor has an overshoot of only 3.4% and is well within the requirement.

In general, the overshoot requirement of 10% seems to met for step inputs of a relative size, and the settling time requirement of 50 milliseconds is impossible for the system to meet, because of saturation and deadband.

13.12 Implementation - digital

13.12.1 Direct implementation

One way of implementing the controller is a direct implementation of the PID-controller. In this case, the plant input will be a summation of the 3 PID contributions, that will all have to be calculated in real-time. This method makes a lot of sense intuitively, and provides easy access to modifying the K-constants during the tuning-process, as these are simply constants that are multiplied with the 3 contributions. In order to use this method, the mathematical operations of integration and differentiation will be have to implemented.

13.12.2 Filter approach

An alternative way to implement the PID controller is converting it into a discrete-time filter. This method has the advantage of not having to do any actual integration or differentiation on the FPGA. Given a continuous-time transfer function and a sampling rate, the matlab command “c2d” can convert the transfer function to the z-domain. Through algebraically isolating the current output of the controller, the filter coefficients for the implementation can be calculated.

The sampling rate should ideally be as high as possible, as this will improve the accuracy of the discrete filter. Through matlab tests, it was found that sampling rates of higher than 10kHz do not create further improvement in the accuracy of the filter, and thus there was no need to increase the sampling rate beyond this value, even though it was technically possible.

13.12.3 Discrete filter for Pan-system [10 kHz]

$$\frac{u(z)}{e(z)} = \frac{1.65z^2 - 3.3z + 1.65}{z^2 - 2z + 1} = \frac{1.65 - 3.3z^{-}1 + 1.65z^{-}2}{1 - 2z^{-}1 + z^{-}2} \tag{14}$$

$$u(z)(1 - 2z^{-}1 + z^{-}2) = e(z)(1.65 - 3.3z^{-}1 + 1.65z^{-}2 \tag{15}$$

$$u[n] - 2u[n-1] + u[n-2] = e[n] - 3.3e[n-1] + 1.65e[n- \tag{16}$$

$$u[n] = e[n] - 3.3e[n-1] + 1.65e[n-2] + 2u[n-1] - u[n-2] \tag{17}$$

13.12.4 Discrete filter for Tilt-system [10 kHz]

$$\frac{u(z)}{e(z)} = \frac{0.3z^2 - 0.5999z + 0.2999}{z^2 - 2z + 1} = \frac{0.3 - 0.05999z^-1 + 0.299z^-2}{1 - 2z^-1 + z^-2} =>$$ (18)

$$u(z)(1 - 2z^-1 + z^-2) = e(z)(0.3 - 0.05999z^-1 + 0.299z^-2$$ (19)

$$u[n] - 2u[n-1] + u[n-2] = 0.3e[n] - 0.05999e[n-1] + 0.299e[n-$$ (20)

$$u[n] = 0.3e[n] - 0.05999e[n-1] + 0.299e[n-2] + 2u[n-1] - u[n$$ (21)

14 H-bridge

An H-bridge is a collecting of 4 transistors, designed to control a DC motor. In figure 42 the concept is illustrated. When the red switches are on, the motor runs in one direction and vica versa for the blue switches. There are two H-bridges in the system, one for the pan motor and one for the tilt motor. For further information about H-bridges, consult the # DATASHEET H-BRO # in the notes.

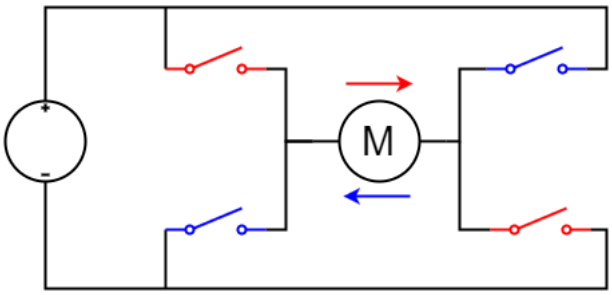


Figure 42: An example of a H-bridge

15 Perspectivation

Here goes the tekst.

16 Conclusion

17 References

- Ref1
- Ref2
- Ref3
- Ref4

Table 1: Instruction list

Function	Opcode (char)	Parameters	Notes
Connection_Check	o	0	Returns a message to the console
Enable_FPGA	f	1	Enable = 1, Disable = 0, by default FPGA is enabled
Goto_Coordinate	g	2	Updates current target in MCU, and sends a new input to the FPGA
Set_Height	h	1	Sends height constraint to MCU
Set_Width	w	1	Sends width constraint to MCU
Set_Depth	d	1	Sends depth constraint to MCU
Set_Scene	s	1	Select a predefined scene, and update MCU to those constraints
Set_Min_Vel_Pan	p	1	Sends minimum speed setting for pan to FPGA
Set_Max_Vel_Pan	P	1	Sends maximum speed setting for pan to FPGA
Set_Min_Vel_Tilt	t	1	Sends minimum speed setting for tilt to FPGA
Set_Max_Vel_Tilt	T	1	Sends maximum speed setting for tilt to FPGA
Run_Lightshow	l	1	Starts lightshow corresponding to parameter value
Stop_Lightshow	c	0	Stops lightshow
Enable_Joystick	j	1	Starts joystick user input

18 Appendix

A Kernel Instruction List