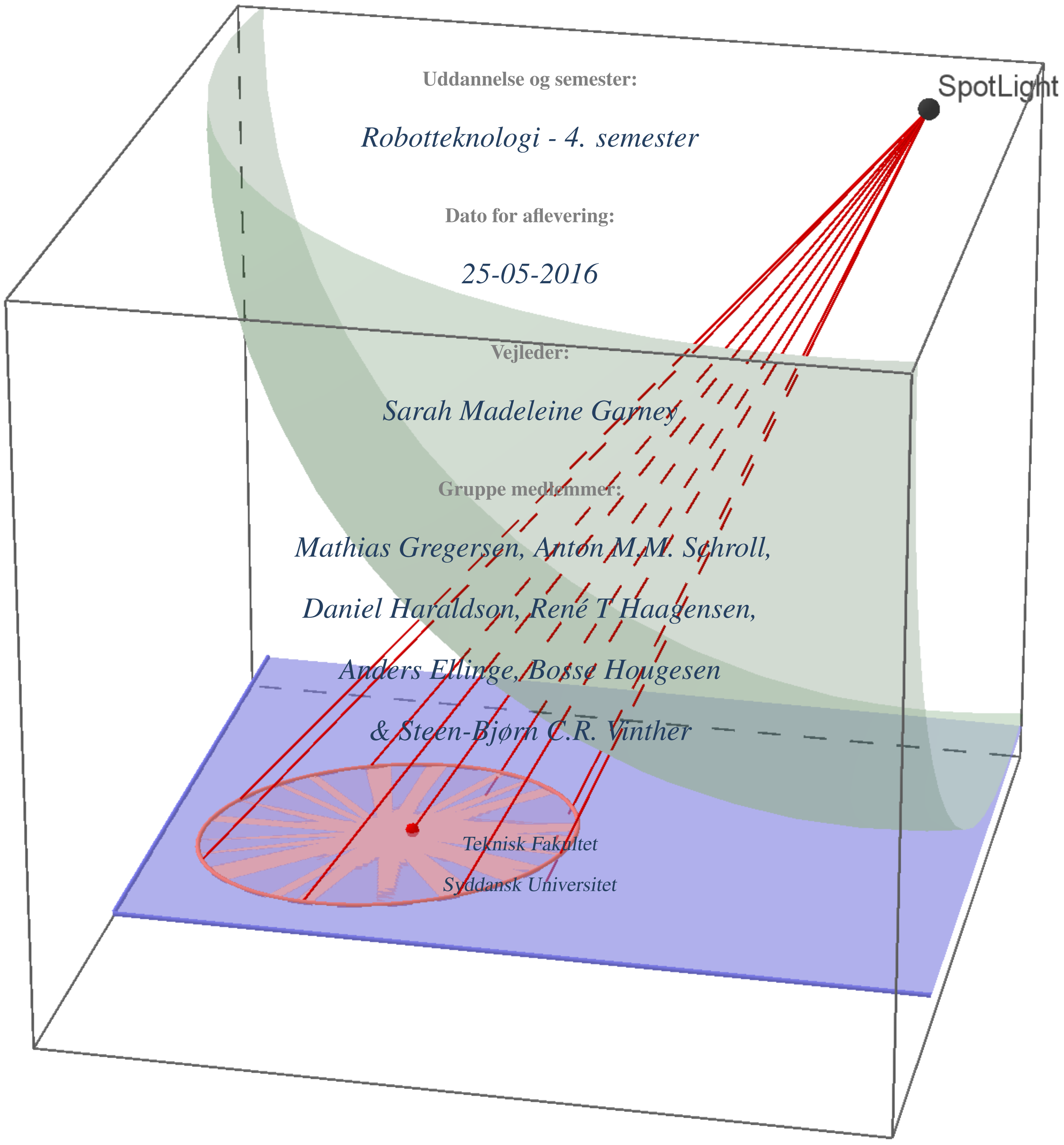
Pan & Tilt

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 Robotteknologi students from University of Southern Denmark in spring 2016. The report is based on the 4th semester project. It is aimed at people with basic knowledge in the disciplines: Digital programmable electronics, Embedded programming, Control Engineering and Computer systems. It describes the theory and analysis involved in designing the controller used to control the Pan & Tilt 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 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 Indledning

Indledning

3 Problem formulation

When performing on a stage, light in the right places is important. An easy way to get that, is via 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 those requirements.

3.1 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.

3.2 Requirements specification

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

3.2.1 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 steeling 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 hight
- The application must support different stage dimensions up to 14 m x 14 m.

3.2.2 Secondary requirements

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

- The movement between two points shall be in a smooth motion.
- The P&T system is to recalibrate the position of the Pan and the tilt under startup.
- The spotlight has to be able to 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 hight 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. [1]

4 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)

5 The System

Here goes the tekst.

6 The Microcontroller

Given the design requirements in section 3.2, and the block diagram in section 5. 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.

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

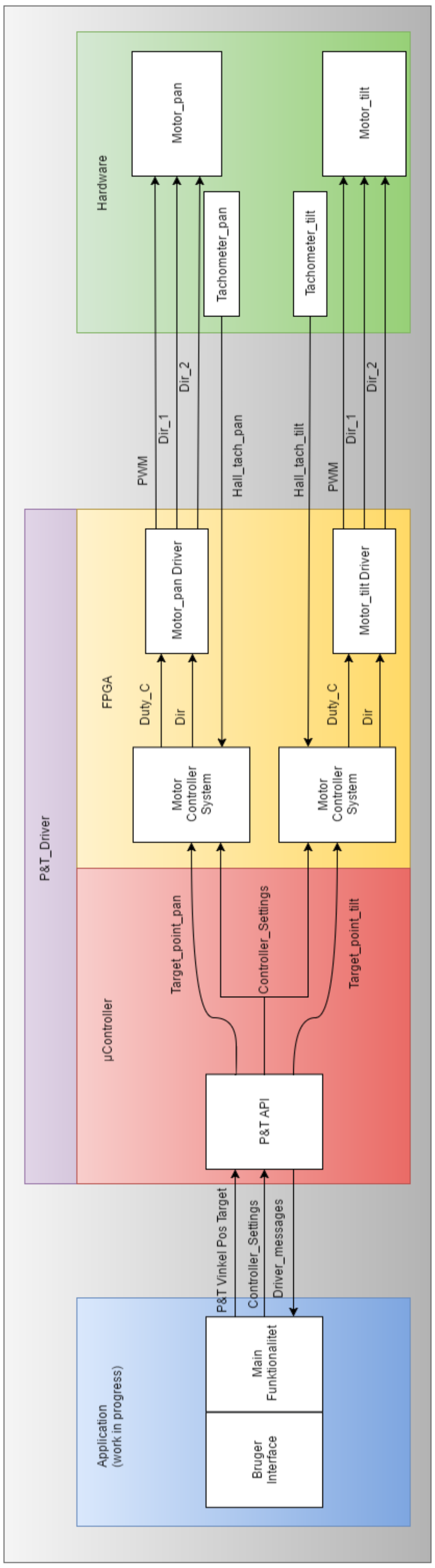


Figure 1: The overall system block diagram

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 states machines.

6.1 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.[2]

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.

6.2 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.

6.2.1 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

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

the P&T driver with new target points once the previus target point has been reached. considering that one still 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

6.2.2 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 make 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.

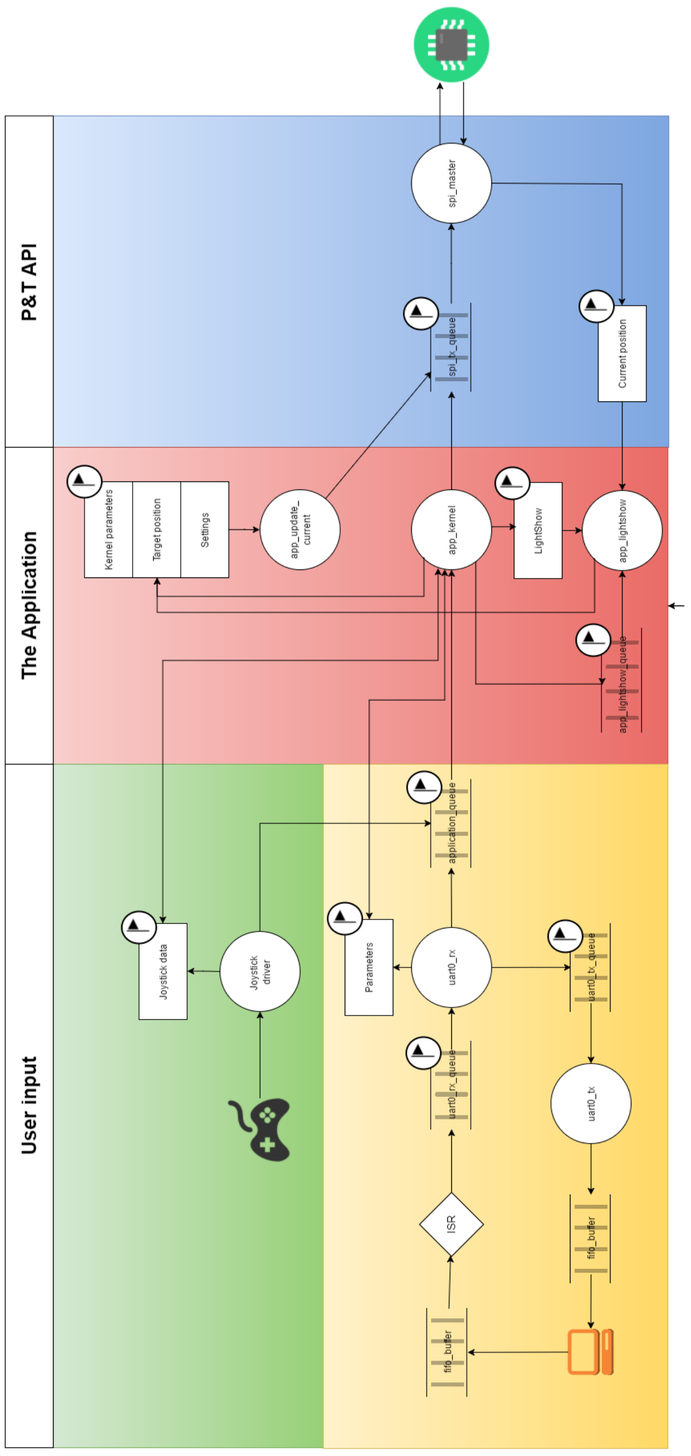


Figure 2: The application task diagram

6.3 The Application Programmer Interface

what is this sorcery - and what will i achieve by using it it is there to represent all the functionality available for each driver in the program.

where in the program will this be used

how will it be implemented

6.4 Event Queues and Shared State Memory

it will ensure that only events can be found in the queue, data will be saved and an event stating that data can be read will be send enstead

this is so that the data never will look like an event.

How is this implemented

6.5 Deadlock Protection

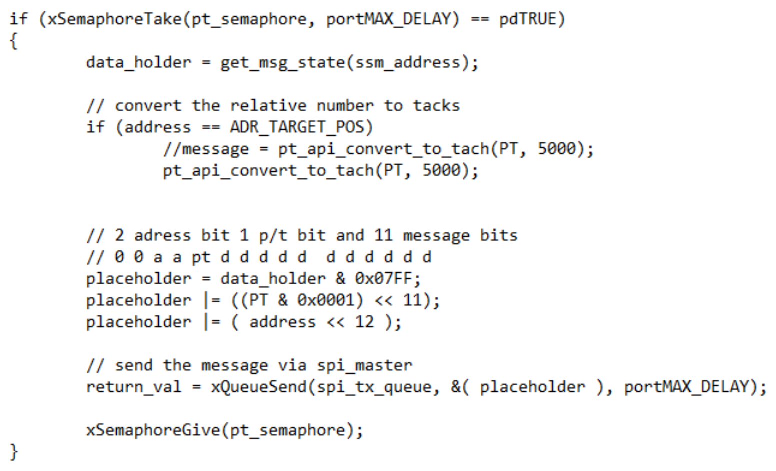
A deadlock is when one or more process cannot proceed because some of the resources it need is held by another process. The two processes will ”compete” with each other and neither of them will finish.

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. The critical section is a part of the program that cant be used by more than one process at the time.
   If more than one process is in the same critical section at the same time errors are bound to happen.
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.
4. **Circular wait** is when one process is waiting for at resource that another process is has and the other process is waiting for a resource held by the first one. It could also be a chain of processes where every process needs a resource held by the next and the last one needs a resource held by the first.
   (add source XXXX)

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 the conditions.

To avoid deadlocks in the system, the circular wait condition is removed via semaphores. This insures that deadlock cannot happen. In figure 3 a code example of how it is used.

First the process takes one semaphore. Then it takes another, do **SKRIV HVAD DEN GØR**. After that it releases the first semaphore and then the last. This insures that it has all the resources it needs to do its work and no deadlock is possible.



```
if (xSemaphoreTake(pt_semaphore, portMAX_DELAY) == pdTRUE)
{
        data_holder = get_msg_state(ssm_address);

        // convert the relative number to tacks
        if (address == ADR_TARGET_POS)
                //message = pt_api_convert_to_tach(PT, 5000);
                pt_api_convert_to_tach(PT, 5000);


        // 2 adress bit 1 p/t bit and 11 message bits
        // 0 0 a a pt d d d d d  d d d d d d
        placeholder = data_holder & 0x07FF;
        placeholder |= ((PT & 0x0001) << 11);
        placeholder |= ( address << 12 );

        // send the message via spi_master
        return_val = xQueueSend(spi_tx_queue, &( placeholder ), portMAX_DELAY);

        xSemaphoreGive(pt_semaphore);
}
```

Figure 3: Semaphore code example

6.6 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

6.7 The Application

6.7.1 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)

6.7.2 The light-show task

making the state machine for the light-show-task

6.8 Overview

The above mentioned functionality was ment as a fully dressed system containing all the features that would be needed while working with a real spotlight. However, not all features was primary requirements, which makes it okay that it never got implemented.

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

7 The FPGA

Here goes the tekst.

8 Driver

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.

8.1 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.

9 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.

9.1 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.

9.2 Theory

This section will describe the theories used for the implementation.

9.2.1 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

9.3 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.

9.3.1 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 #.

9.3.2 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.

9.4 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.

9.5 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.

9.6 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.

10 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.

10.1 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.

10.2 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.

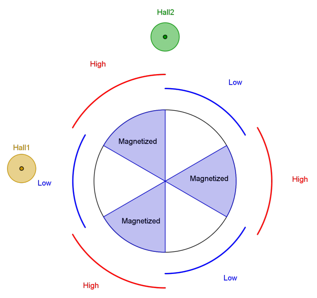


Figure 4: This is an example of how the Hall effect sensors is placed around the disk.

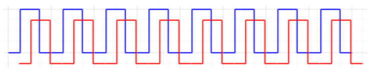


Figure 5: As the axis rotates, the output changes

10.3 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 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

If no change have been registered, the position doesn't change and a new sample is taken. As seen in figure 7. If a change is detected then a series of "if statements" determines whether the change in position is positive or negative.

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

10.4 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

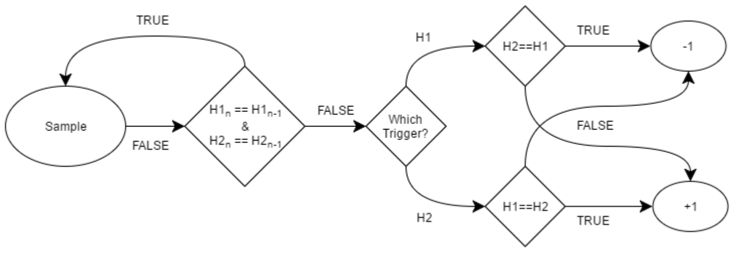
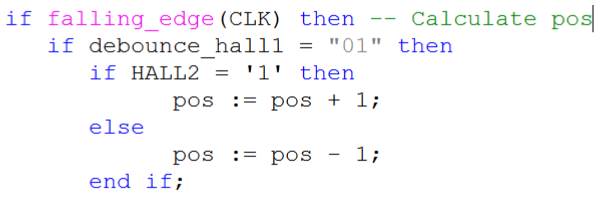


Figure 7: Flowchart of how the position changes in relation to different Hall effect sensor outputs

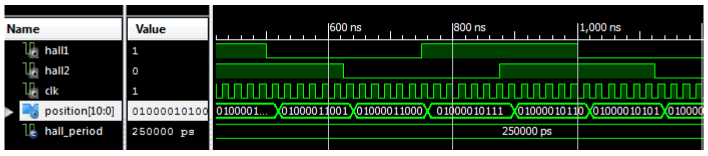


```
if falling_edge(CLK) then -- Calculate pos
   if debounce_hall1 = "01" then
      if HALL2 = '1' then
            pos := pos + 1;
      else
            pos := pos - 1;
      end if;
```

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

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.



```
Name              Value
hall1             1
hall2             0
clk               1
position[10:0]    01000010100
hall_period       250000 ps
```

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

10.5 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

10.6 Conclusion

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

11 SPI slave block

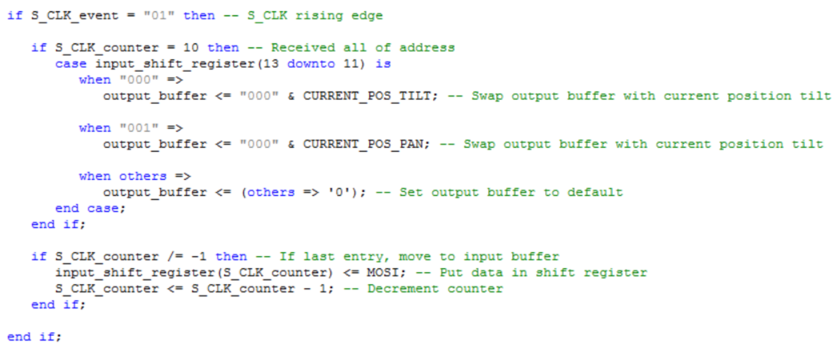
This section will describe in detail how the SPI slave in the FPGA receives and transmit data according to the description of the SPI communication protocol mentioned before.

11.1 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.

11.2 Implementation

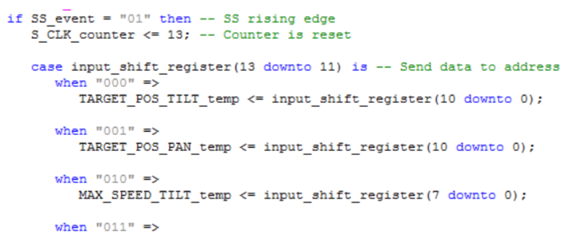
The SPI slave could have been implemented as a state machine. However because of the rather simplistic nature of the SPI communication protocol it was chosen not to implement it as a state machine. Instead it was implemented as a series of different events that trigger different responses in the SPI 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.

11.3 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.

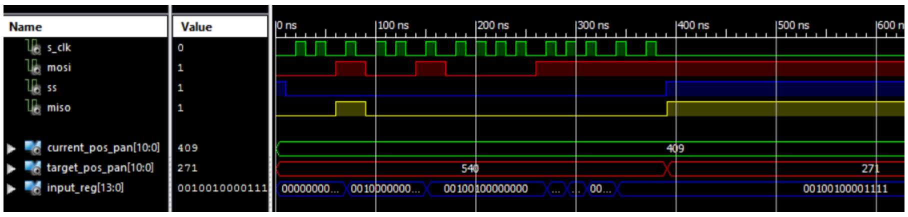


Figure 12: An Xilinx testbench simulation of the SPI

11.4 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.

11.5 Conclusion

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

12 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.

13 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.

13.1 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.

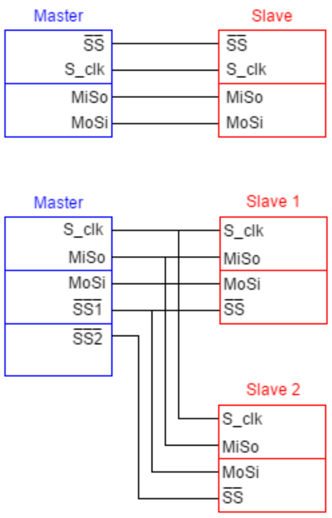


Figure 13: Diagram of how the master and slave is connected

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.

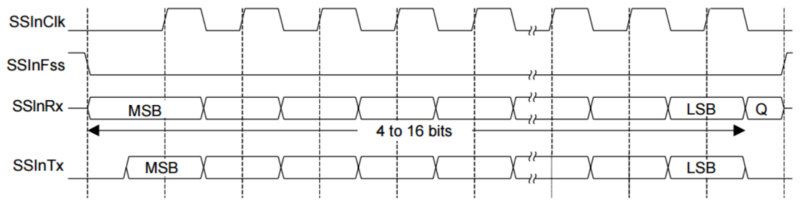


Figure 14: Illustration of how the package is constructed

13.2 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.

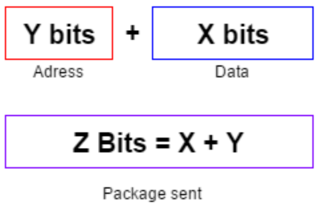


Figure 15: Illustration of how the package is constructed

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

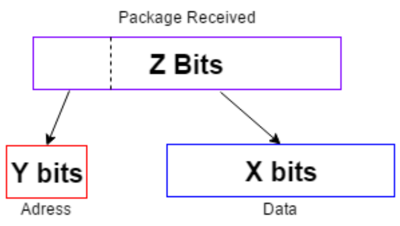


Figure 16: Illustration of how the package is destructed

14 The Controller

Here goes the tekst.

15 PID

15.1 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 17 is the point of reference, where the plant is the previously estimated transfer function, **SKRIV FORMLEN** EQUATION REF, of the motor of the system.

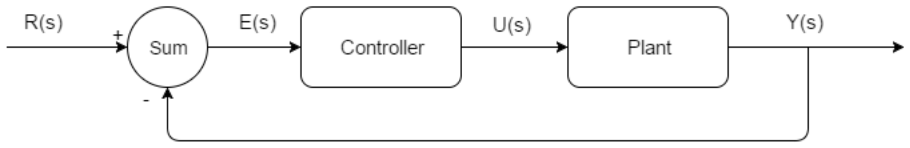


Figure 17: A generic way of representing a close-looped control system

15.2 The Controller

For simplicity's sakel 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 18 for illustration.

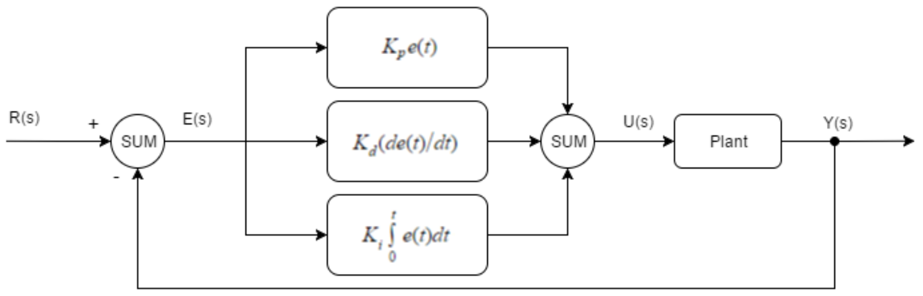


Figure 18: A representation of the PID controller principle

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:

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

15.3 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 23. As one would expect it seems like this kind of controller will not suffice for the project.

As seen in figure 19, 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.

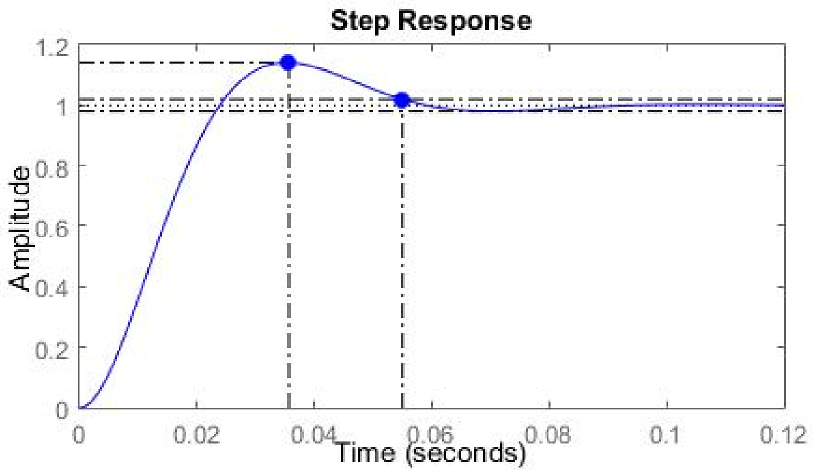


Figure 19: Peak amplitude: 1.14 - Overshot(%): 14.1 - At time(seconds): 0.0356 Settling time (seconds): 0.055

15.4 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 23, 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 20, 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.

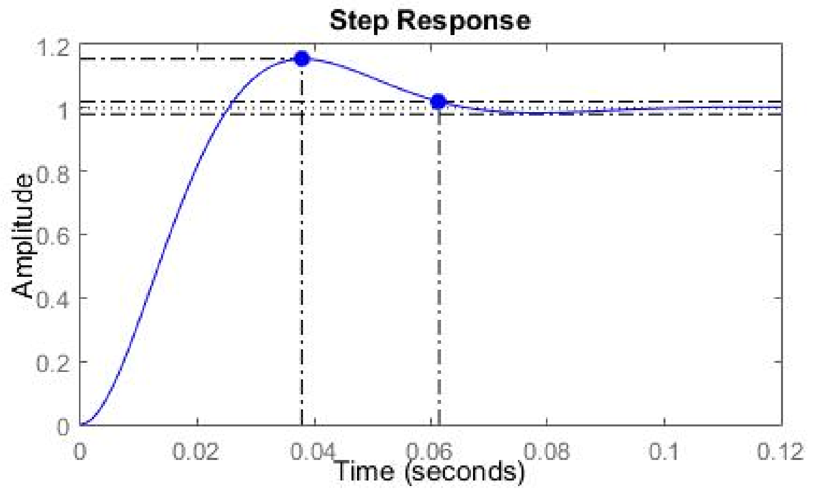


Figure 20: Peak amplitude: 1.15 - Overshot(%): 15.5 - At time(seconds): 0.0379 Settling time (seconds): 0.0614

15.5 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 21 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.

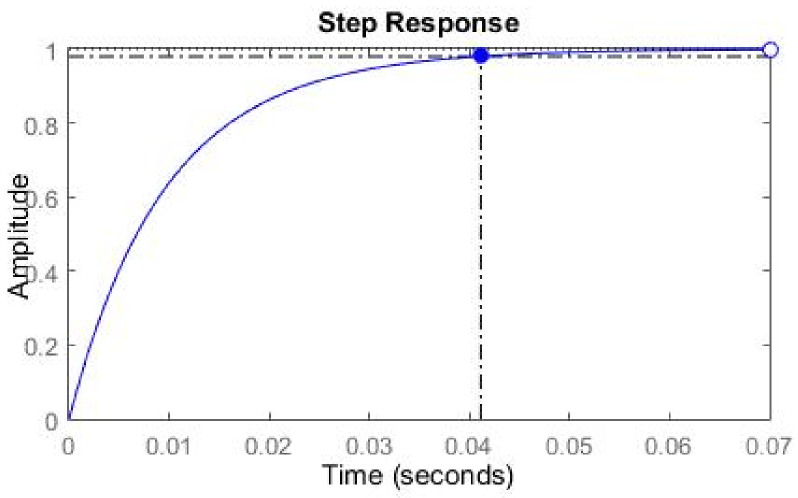


Figure 21: Peak amplitude: >= 0.998 - Overshot(%): 0 - At time(seconds): > 0.07 Settling time (seconds): 0.0411

15.6 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 23, 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 22, 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.

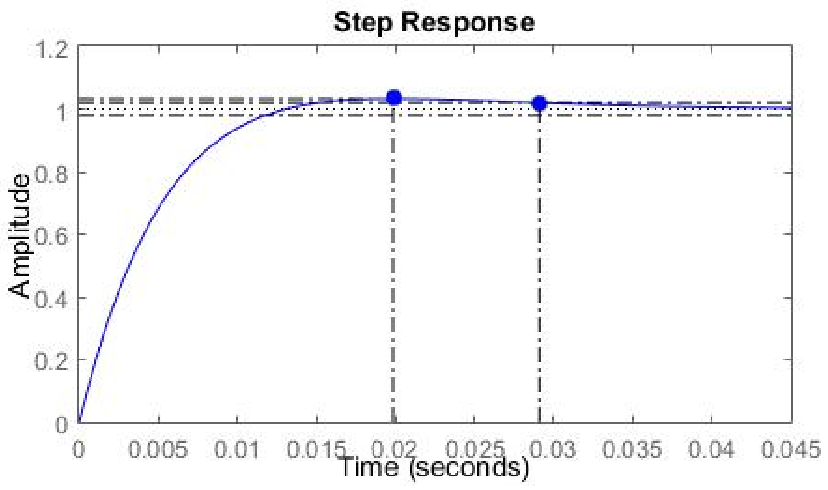


Figure 22: Peak amplitude: 1.03 - Overshot(%): 2.92 - At time(seconds): 0.0201 Settling time (seconds): 0.0277

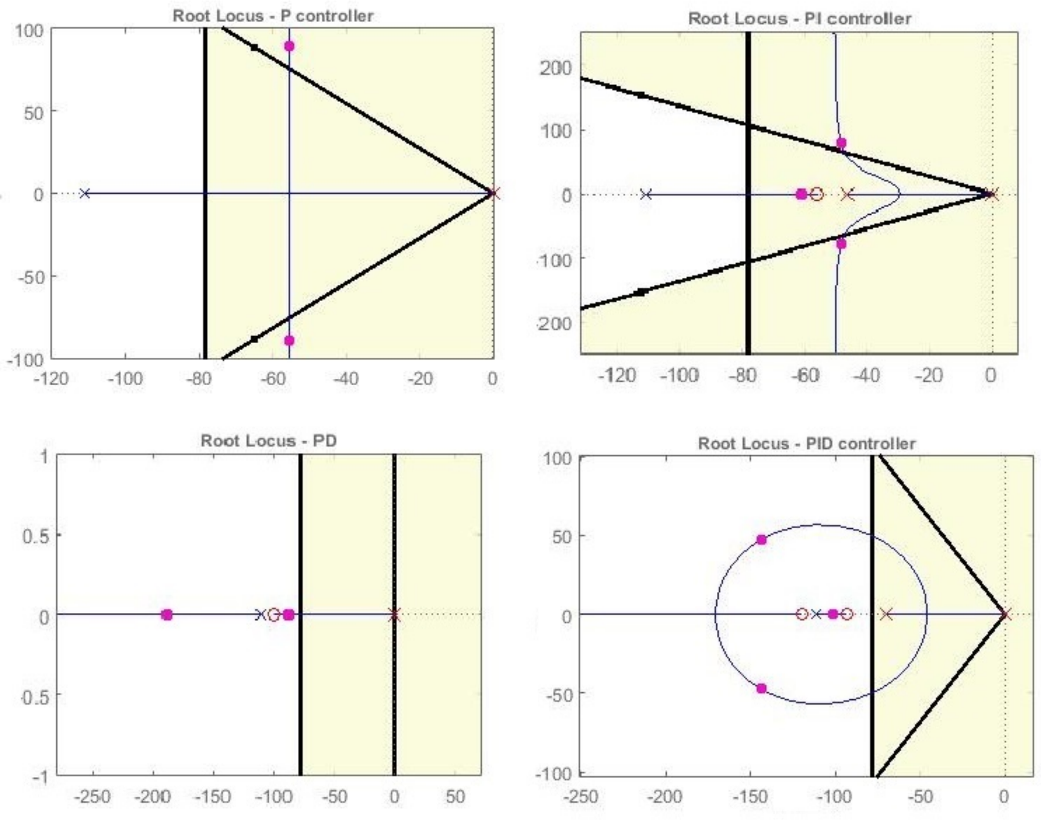


Figure 23: Open-loop transfer function Root Locus plots of the estimated motor transfer function, EQUATION REF, with different controllers. The shaded areas indicates designs not met if poles of the closed-loop transfer function is placed there. The designs are: Overshot > 10%, settling time ¡ 0.05 seconds. 7

16 Perspectivation

Here goes the tekst.

17 Conclusion

18 References

- Ref1
- Ref2
- Ref3
- Ref4

Table 1: Instruction list

test	test	test	test

19 Appendix

A Kernel Instruction List