Closed loops POC

July Trendafilov



Version table

|  |  |
| --- | --- |
| Version 1 – 20.12.2022 | First POC for closed loops |
| Version 2 – 14.01.2023 | Added Closed Loop Assignment Part |
| Version 3 – 19.01.2023 | Working PID + Mapping explanation |

Contents

[MatLab 3](#_Toc125045342)

[Servo Code 8](#_Toc125045343)

[Tests 13](#_Toc125045344)

[Closed Loops Assignment 17](#_Toc125045345)

[Design 17](#_Toc125045346)

[Processing Data 19](#_Toc125045347)

[Rotating the servo 20](#_Toc125045348)

[The new PID controller code 20](#_Toc125045349)

[PID angle mapping 22](#_Toc125045350)

[Testing 24](#_Toc125045351)

[Result 24](#_Toc125045352)

[Bibliography 24](#_Toc125045353)

# MatLab

To calculate the PID values we need to make a simulation in MatLab. In the simulation we can get the necessary PID values for our code.

To make a PID simulation we need several components: a variable, control signal, controlled variable, a plant, a feedback signal.

Diagram

Description automatically generated

The transfer function is to simulate a servo because it has a response delay when given signal. The transfer function adds this delay. To calculate it we need to look at the spec of the servo.

Chart, line chart

Description automatically generated

Here we have the maximum amplitude of the unloaded speed as 2.25 approx.

Image with text, inside

Automatically generated description

T is equal to 63% of the total speed. To get the T in this diagram we must make the following calculation:

2.25 x 0.63 = 1.4175

Graphical user interface, application, table

Description automatically generated

Our value is closest to 1.42 so the time in ms is 109.

From the same datasheet we can see when the rotation starts:

Image with table

Automatically generated description

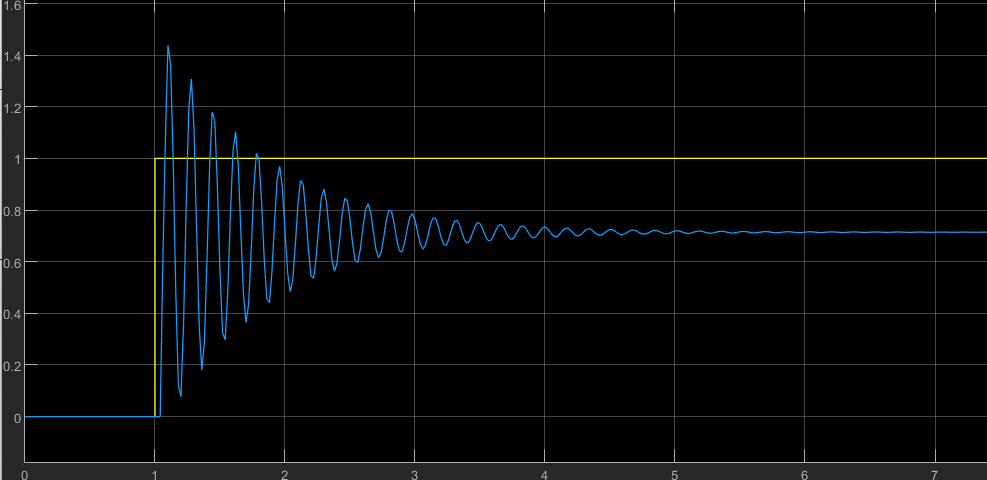
The first rotation is at 42ms. To get the Tau we must do:

109 – 41 = 68ms which is 0.068s

When we have the correct transfer function value, we can start tuning the PID.

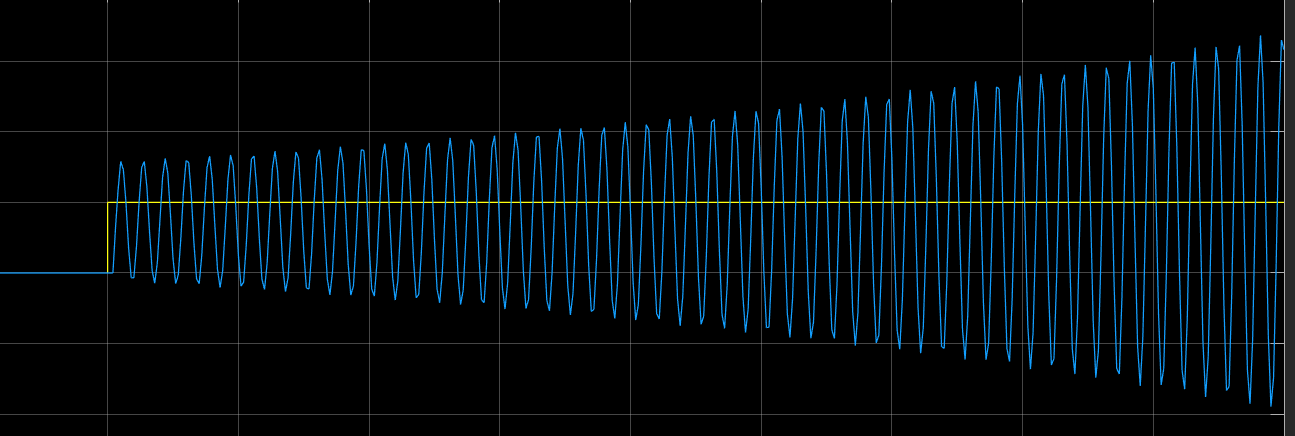
To get the correct values we can use the Ziegler-Nichols method:

To do it we set all PID values to 0 and start increasing P until we get a constant oscillation. We must do a couple tunes to find the right value. If the value is too low the oscillation will decrease:



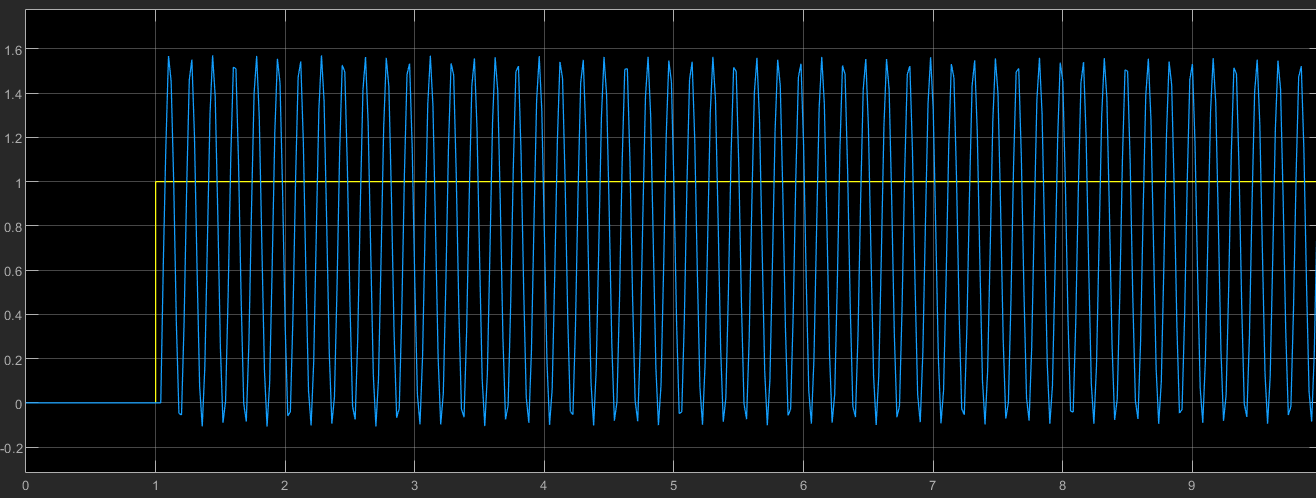
kP is 2.5

If the value is too big the oscillation will increase over time:



kP is 2.75

After a couple tries, we can get the oscillation to be constant:

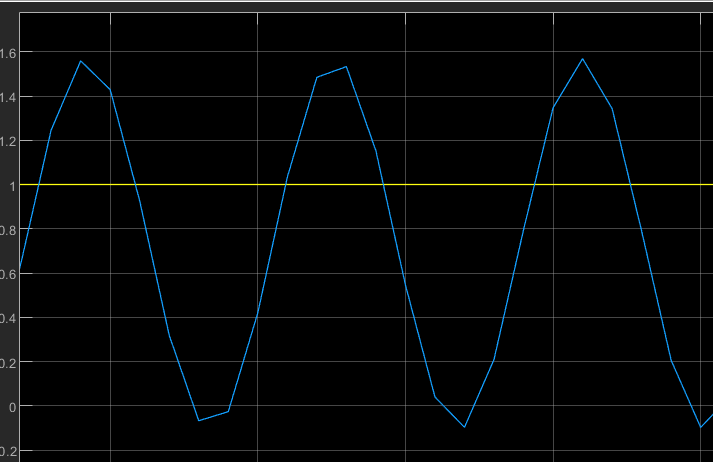


kP is 2.725

Now to get the correct P we must make a calculation depending on the method table. To get the correct P we must do:

2.725 \* 0.45 = 1.226 kP

Now to calculate the kI we must calculate the time between each oscillation:



3.12 - 2.91 = 0.21

2.12 - 1.94 = 0.18

9.98 - 6.81 = 0.17

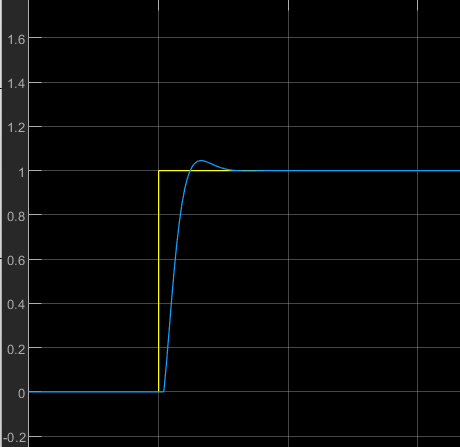
4.96 - 4.8 = 0.16

After gathering data from different seconds I can conclude that the average time is

((21 + 18 + 17 + 16) / 100) / 4 = 0.18 s

Now I = (0.54 \* 2.725) / 0.18 = 8.175

After a little tuning on the P we get a good signal:



P = 0.526

I = 8.175

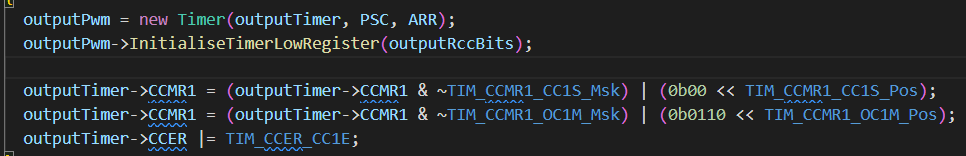
D = 0

The result is that we have the proper position is about 1.55 ms. Most of the times the values from this method are not correct so one must tune them additionally.

# Servo Code

The cod is for the Parallax 360 servo. This servo has only two functions:  
Read the current angle and rotate. To do this the servo needs one timer to read the PWM input and one timer for the PWM output.

To configure the output, I am using a timer from my library:



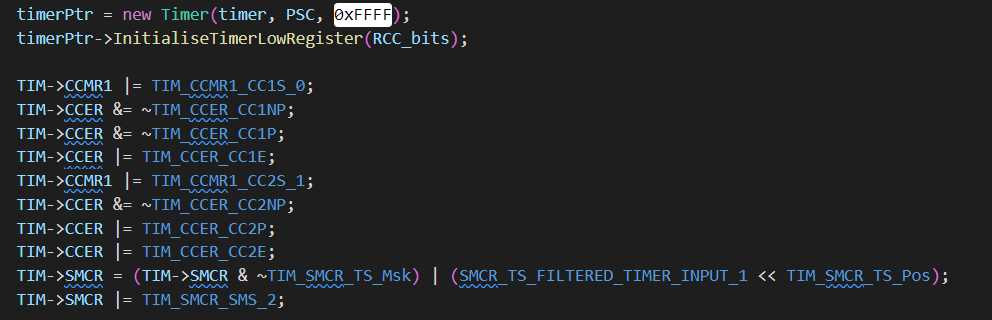
I set the prescaler to 72 so I get 1MHz and set the ARR to 20000 so I can measure 0.02s.

72 000 000 / 72 = 1 000 000 = 1MHz

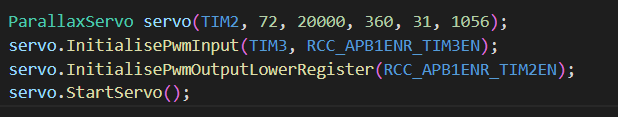
0.02s \* 1 000 000 = 20000 ticks

Output timer is the pointer to the address (eg. TIM2, TIM3). When set the timer, we must set the correct alternate function to the correct Pin.

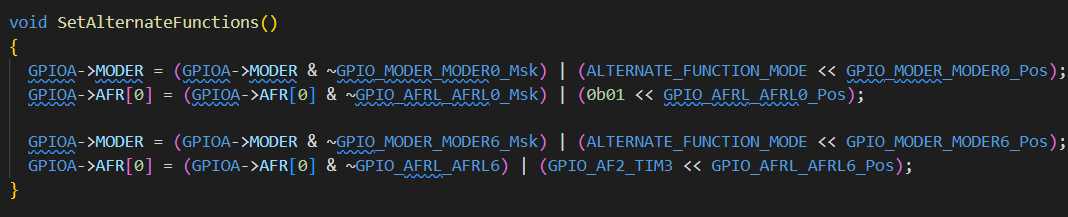
To configure the input we use a timer again with the correct registers set (see “Timers“ presentation)



To setup a servo we use the current class for the parallax. To initialize one servo object we use this:

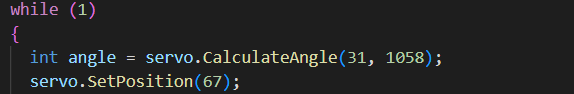


As shown previously we set the correct parameters for the servo, then initialize its input and output.

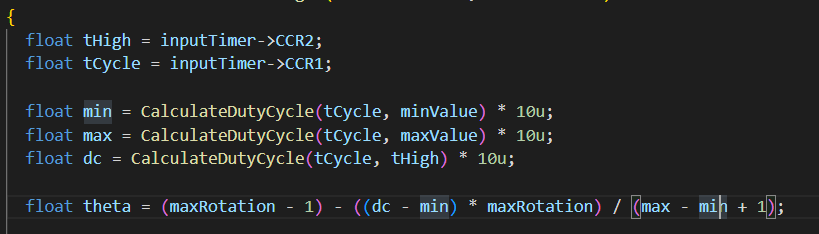


This function sets the alternate functions to PA0 – TIM2 and PA6 – TIM3.

To use the servo there are two functions which are called in the main loop so that the angle is continuously reached and calculated for the PID controller:

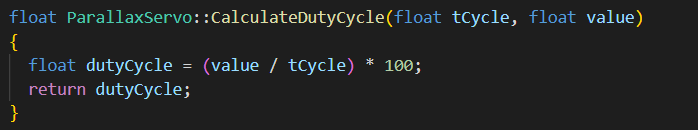


The function CalculateAngle is called inside setPosition. Here we call the function additionally to read the angle so we can print it to the serial monitor.



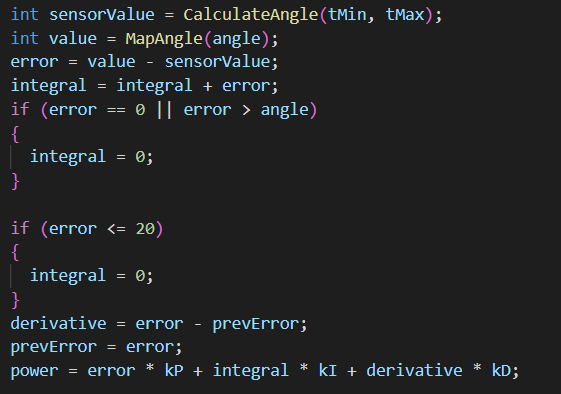


To calculate the angle we must calculate the duty cycles

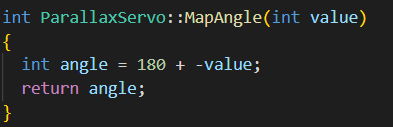


For the actual angle we must calculate the Min dutyCycle, the Max dutyCycle and the current dutyCycle.

The SetPostion function does all the work.

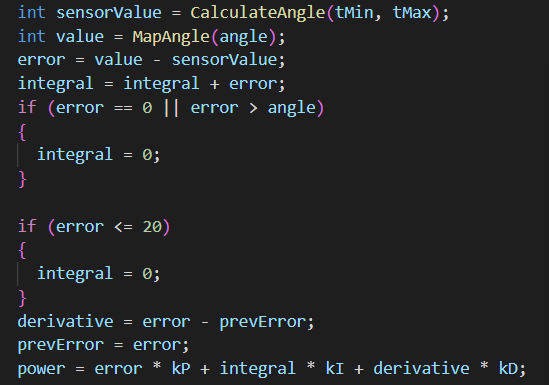


It calculates the angle and then maps it. In other words we read the current degrees and we want to convert it to angle 0 – 359 degrees.



If we get a value 0 that means we are on one side. Otherwise the servo will get to the middle because the Parallax has a range of -180 till +180 so 0 degrees is in the middle and if we give 360 that means -180 which is the other side.

PID controller look as:



Each cycle we calculate the error and the integral and then we use the following code from the datasheet to calculate the correct signal.

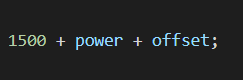
In our case the PID has the values:

P = 0.526

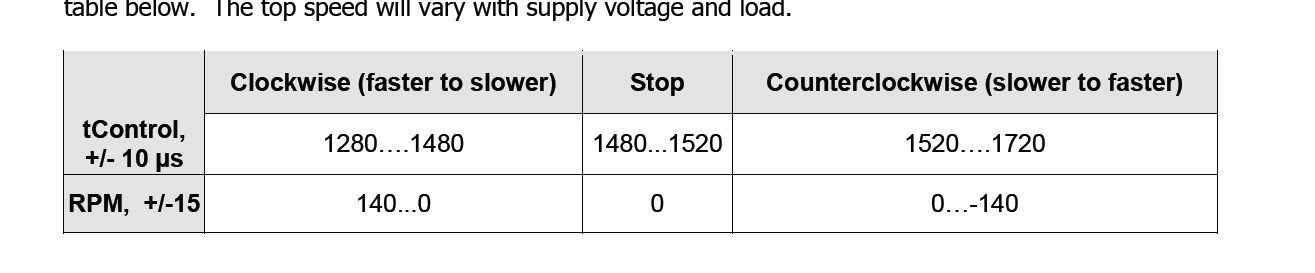
I = 8.175

D = 0 (no overshoot)

The actual value for the duty cycle is actually calculated as



This is because in the datasheet it is stated:



This table shows that values from 1280 to 1480 turn the servo left at a different speed, a value from 1480 to 1520 is not moving the servo and a value from 1520 to 1720 is moving the servo to the right.

To not exceed the value there are some checks in the code to keep the right values.

Image with text, monitor, screenshot, screen

Automatically generated description

If the PID wants more power the controller can take, it wont allow it to request the illegal value. The PWM signal to be sent to the servo needs a small offset to get exactly to an angle. In this case, an offset of 45/-45 ensures that a servo is placed exactly on an angle (based on the stop values).

# Tests

Map

Description automatically generated

A picture containing background pattern

Description automatically generated

The current angle is set to -150 which is 180 + 150 = 330 degrees

Map

Description automatically generated

Background pattern

Description automatically generated with low confidence

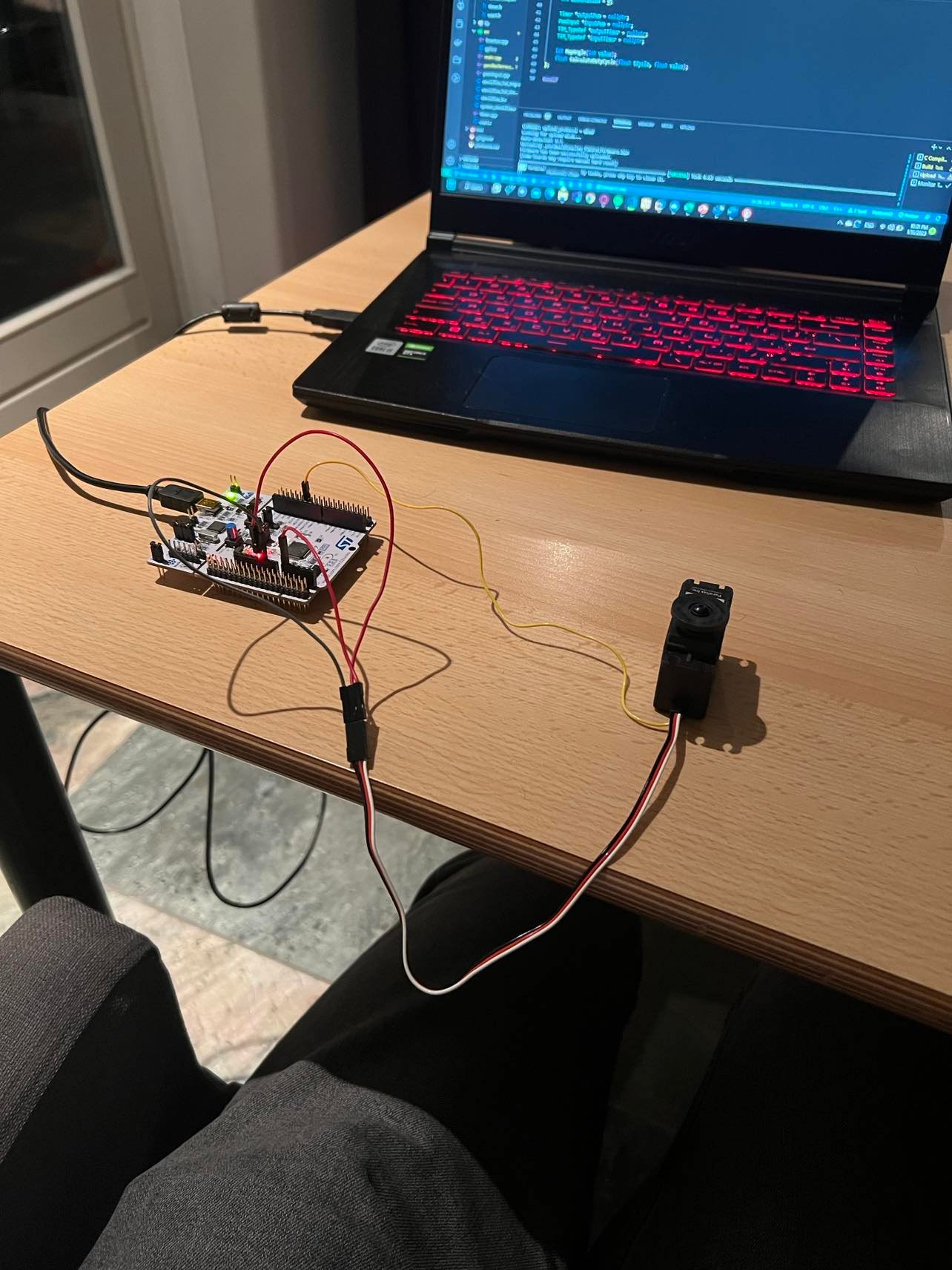
If we set the angle to 60 that gives us 180 – 60 = 120

# Closed Loops Assignment

The assignment for the closed loops is to make an application where the user can tune the PID values in real time. To do so the implementation has a thread which is responsible for collecting data from the serial monitor and a thread which controls the servo. They look as:

## Design

The design for this assignment is one STM32 board and a loaded servo. The servo is connected to the base of the robot arm of our Project.



The software diagram is as follows. The user enters data which the Serial thread formats and pushes to the MessageQueue. Then the Servo thread pops value from the message queue and updates the PID and returns to Serial Receiver.

Diagram

Description automatically generated

## Processing Data

Text

Description automatically generated

When we receive data from the serial port we must clear the interrupt flag. If a message is read from the Serial Port, then we must format the input into different variables. In our case we have a structure which has the field P, I and angle because we send those values from the Serial Monitor using space separator. When the structure is initialized we then put it into a stack.

## Rotating the servo

Text

Description automatically generated

Our second thread is for rotating the servo. If there is available data in the Message Queue, we pop it into a structure of type Message. Then we tune the P and I values using the SetKi and SetKp. The SetPosition is always in the running code because the PID controller must be working at all times. The osDelay is 20ms to give the servo some time to rotate and change the PID error.

## The new PID controller code

Text

Description automatically generated

The new PID controller code was inspired by:

Graphical user interface, text

Description automatically generated

If the error is 0 or passes the useful range, we must set the integral to 0.

Text

Description automatically generated

## PID angle mapping

The PID implementation works in a 360 degrees range [0,359]. To calculate the error we must read the current position of the servo and set the desired position. To calculate the current angle we use:

Text

Description automatically generated

To calculate the angle, we must calculate the duty cycles of the Min and Max value and the current duty cycle. From the value stored in the falling counter, we can calculate those duty cycles. Then we calculate the current angle using the formula from the datasheet. The angle is calculated between 0 and 359.

Text

Description automatically generated

Then we must convert the value passed by the user from [-180:180] into [0:359] so that the current angle and the desired angle have the same boundaries and the error from the PID is correct. Else we will be comparing 180° to 359° and the error will be incorrect.

Text

Description automatically generated

Then in the SetPosition we use the calculated angle and the desired angle for the correct error.

## Testing

For the test we will attach load to the servo and test different PI values.

## Result

The result is that we can tune a PID controller in Real Time and move a servo by giving it different angles. In this assignment I have learned how to make proper PID controllers and tune the PID using the serial monitor on STM boards.

# Bibliography

*Bakx, R. (2021), Closed loops systems presentation.* (n.d.).

Bakx, R. (2022). Retrieved from GitLab: https://git.fhict.nl/technology/t-sem3-db/-/tree/master/embedded-systems/Presentations

*STMicroelectronics N.V (2017), STMReference Manual- STM32 advanced ARM®-based MCUs.* (n.d.). Retrieved from toolbox