Timer challenge ES

Voor ons eindproject gaan wij een autonome lijn volger robot maken. Dit project doen wij met ons tweeën, Jarno en Bart. Dit project moet voldoen aan een aantal functionele eisen. Die hebben wij op de volgende pagina op een rijtje gezet. We hebben het project zo aangepakt dat het ook onze leerdoelen aan toont.

Inhoud

[Functional requirements: 2](#_Toc61598699)

[Non functional requirements 2](#_Toc61598700)

[Flowcharts 4](#_Toc61598701)

[PID control 4](#_Toc61598702)

[Threading 5](#_Toc61598703)

[V2 7](#_Toc61598704)

[Bronnen threading 7](#_Toc61598705)

[Zelfgemaakte queue systeem 8](#_Toc61598706)

[Hoe werkt het 8](#_Toc61598707)

[Add to queue 9](#_Toc61598708)

[RetrieveFromQueue 13](#_Toc61598709)

[OOP 14](#_Toc61598710)

## Functional requirements:

* Moet een MCP zijn
  + Start/stop knop
  + Rotary encoder
  + Aan/uit led
* Serial monitor
  + Via serial ook aan en uit kunnen zetten
* Moet sensoren bevatten met een input signaal met een specifieke timing
  + Feedback servo
* Actuator of output signaal met een specifieke timing
  + 2 high speed 360 servo’s
* Timer in een handige manier gebruiken
  + Servos
* Belangrijke waardes kunnen aanpassen tijden het rijden
  + Rotary encoder
  + Start/stop knop
* Afgeleid van closed loop opdracht:
* Sensor met PID feedback signaal
  + Infrarood sensoren

## Non functional requirements

* 1micro second timer ticks
* Applicatie reageert altijd meteen op user input
* Geen memory leaks
* Coding standaard
* (s)UI staat los van de applicatie logistiek

**Losse requirements leerdoelen**

Gebruik van eigen queue algoritme ofzo

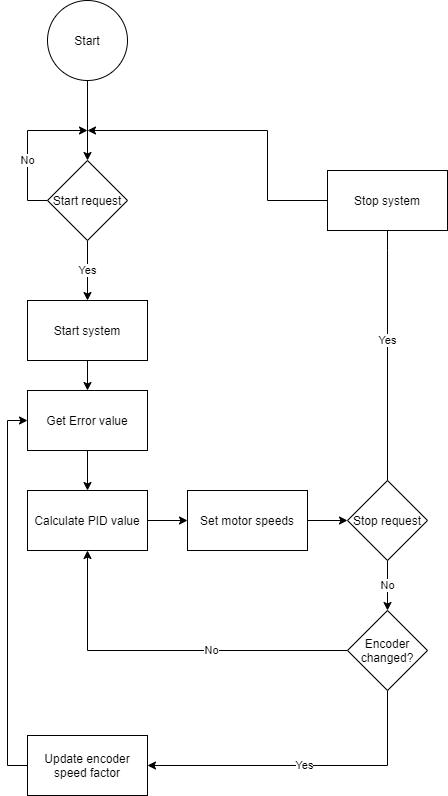
Synchronisatie ontwerp uitleg

Es unit testen

Code overeenkomen met design

## Flowcharts

Nucleo main

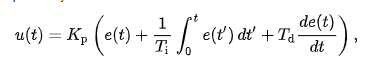


# PID control

Om zo snel mogelijk de lijn te blijven volgen is het belangrijk om dit te doen zonder te veel schokkerige bewegingen. Dit wordt opgelost door het te gebruiken van een PID regeling. Dit houdt in dat we de motoren afremmen om de lijn in het midden te houden.

De PID regeling maakt gebruik van de input van de IR sensoren op de voorkant van de auto. Als die opgehaald zijn kan de PID waarde berekend worden.

De waarde van hoever de motor moet remmen kunnen we berekenen met de PID formule.



Dit is een heel ingewikkelde formule, dus gebruiken we een simpelere versie:

**PIDvalue = (Kp\*P) + (Ki\*I) + (Kd\*D)**

Waar:

Kp = de constant die gebruikt wordt om de grootte van de verandering aan te passen om bij het set point te komen.

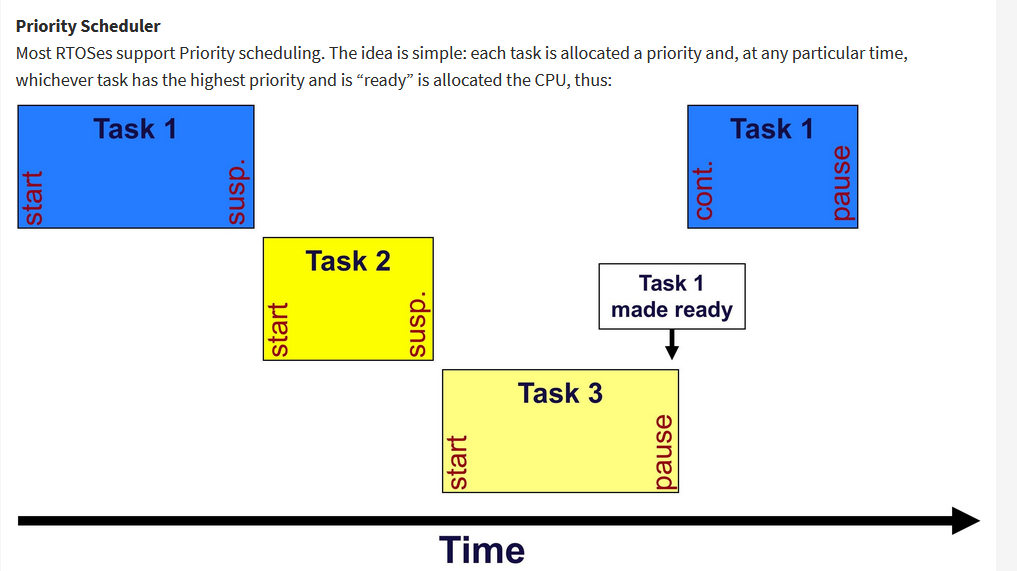
Ki = de constant die gebruikt wordt om de snelheid van verandering aan te passen.

Kd = de constant die gebruikt wordt om de stabiliteit van het systeem te veranderen.

Hier komt een waarde uit die we van de totale waarde van de motoren af kunnen halen waardoor de motoren afremmen en het autootje stuurt.

# Threading

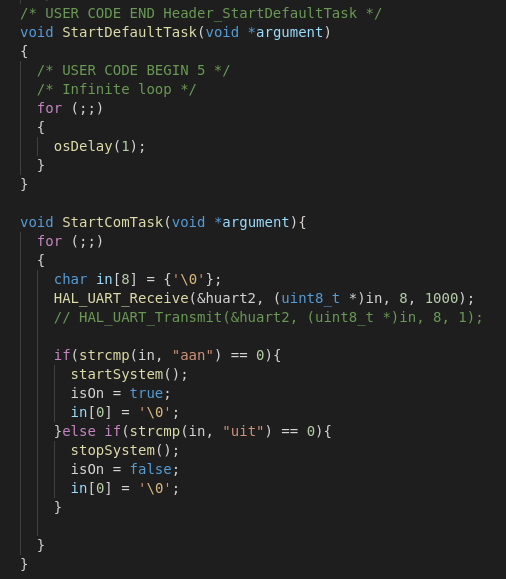
Voor de robot zijn er verschillende requirements. Onder andere moet het kunnen communiceren via seriële communicatie en fysieke knoppen. Hiernaast moet het programma nog andere taken verrichten. Om dit overzichtelijk te maken en blokkerende code tegen te gaan wordt dit in losse taken verdeeld. Ook wel bekend als threads. Niet dit verwarren met multithreading. De microcontroller die de robot aanstuurt, de Nucelo F303RE, heeft als besturingssysteem freeRTOS. Deze RTOS(real time operating system) bied ondersteuning voor deze taken. Elke thread zijn eigen taak. Zo maken wij bijvoorbeeld een taak aan voor het communiceren via seriële port, de sensoren, actuatoren en dergelijke.



Figuur 1: uitleg bij tasks

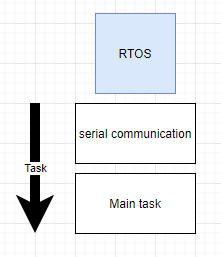
Zoals er in de afbeelding gezien kan worden als er prioriteit gegeven wordt aan een taak kunnen andere taken gepauzeerd worden. Dit kan misschien problemen leveren bij bijvoorbeeld serieel data lezen van de terminal op de computer. Hierom wordt er misschien geen prioriteit aan de andere taken. Dit moet getest worden.

Na het testen bleek het geen problemen te leveren. De hoofdtaak had een normale prioriteit gekregen terwijl de communicatietaak een lagere kreeg.



Onderstaande afbeelding laat zien wat ons thread design is.

### V2



Communicatie tussen taken kunnen echter soms tot grote problemen leiden. Als de main task iets intern wilt communiceren met de com task, hoe zou hij dat dan moeten doen? Je zou globale variabelen kunnen instellen omdat beiden taken daar toegang naar hebben. Dit is alleen niet thread safe. Thread safe houdt in dat beiden tasks weten wanneer er naar een stuk geheugen geschreven kan worden of het er veilig van opgehaald kan worden. Om het programma thread safe te maken heb je meerdere manieren om dit op te lossen. Zo hebben wij gekeken naar mutexes en queues.

Als er naar mutexes wordt gekeken kan je ook aan denken aan een meeting in een groep. Hier mag je alleen praten als je een rubber eendje hebt. Je mag aangeven dat je het eendje wilt en daarna moet je gaan wachten met praten totdat je het eendje hebt. Als je nu de personen veranderd met threads en het eendje met een mutex dan heb je de uitleg over hoe mutexes werken. Hoe dit eruit ziet bij ons project is ook heel makkelijk uit te leggen, mutexes zorgen er voor dat er maar 1 taak tegelijkertijd bezig kan zijn. Dit wordt ondersteund in de CMSIS library die wij gebruiken in freeRTOS.

Echter gaan wij geen gebruik maken van mutexes maar van queues. Dit doen wij om 2 redenen. Ten eerste willen wij onze leerdoelen aan kunnen tonen. 1 van die leerdoelen is dat wij zelf een een data structuur en algoritme implementeren. Dit kan gedaan worden door zelf een queue te maken.

De tweede reden is omdat wij het overbodig vinden. Door de mutexes kan maar 1 van de 2 taken “tegelijkertijd”(verdeeld, niet multi-threading) werken. Als je terug kijkt naar task sceduling zie je dat als je een taak een hogere prioriteit geeft onderbreekt die taak de andere taak met een lagere prioriteit. Dit zou niet kunnen met mutexes. In het volgende hoofdstuk wordt meer uit gelegd over de queue.

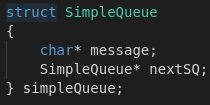
### Bronnen threading

<https://www.embedded.com/tasks-and-scheduling/>

<https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__ThreadMgmt.html>

<https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__MutexMgmt.html>

# Zelfgemaakte queue systeem



Het project heeft minstens 2 taken, de main en communicatie task. Als het systeem een melding krijgt in bijvoorbeeld de main task en het moet geprint worden in de seriële monitor is het lastig intern te communiceren met de communicatie task. De communicatie task weet niet wanneer een bericht klaar is om te versturen. Hiervoor moet een queue systeem komen zodat de communicatietaak weet dat een bericht klaar staat. Aangezien dit een leerdoel is maken wij de queue structuur zelf.

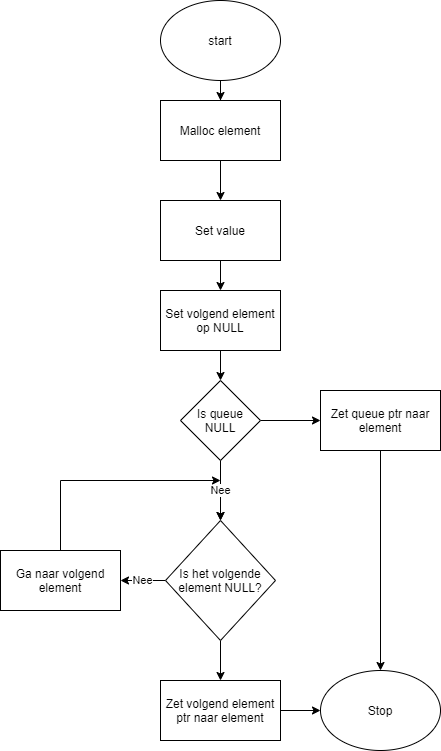
### Hoe werkt het

De queue is maakt van een struct met twee waarden, het bericht en een pointer naar de eerstvolgende struct. Als het systeem een bericht heeft die verstuurd moet worden via de communicatietaak wordt het bericht in een queue struct gestopt. Als een extra bericht word toegevoegd wijst de eerste struct pointer naar de volgende queue struct. Zo krijg je een hele stack aan berichten die verstuurd kunnen worden.

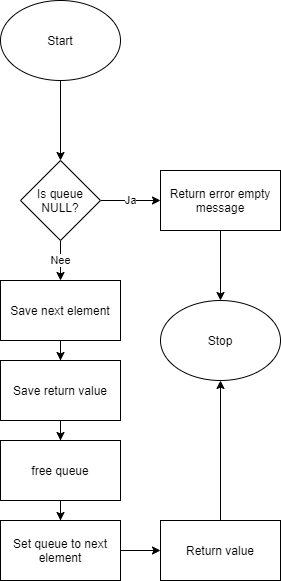
Als de communicatietaak begint word er gekeken of er een bericht in de queue zit. Als er een bericht in zit wordt het bericht opgehaald, verwijderd uit de queue en verzonden over UART.

### Add to queue

Dit is versie 4 van onze add to queue flowchart.

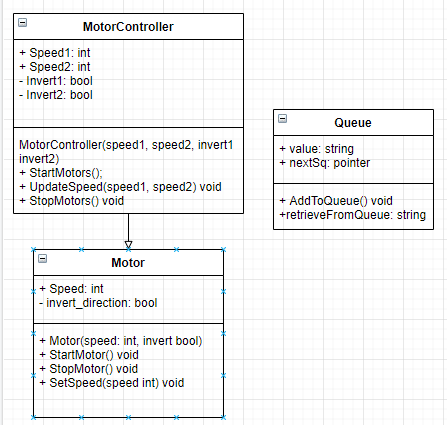


### RetrieveFromQueue



# OOP

Klasse diagram

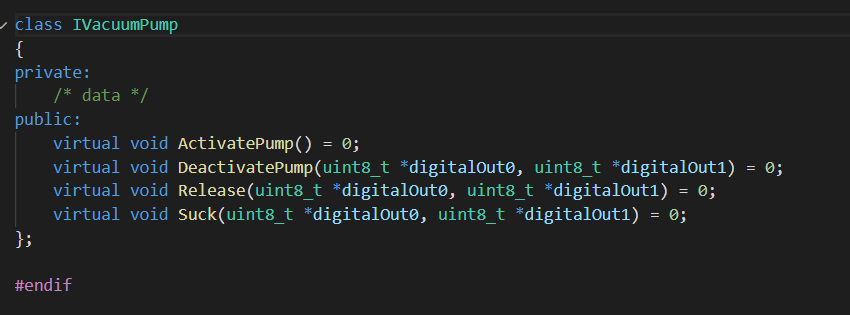


Door de onderdelen in klassen te onderverdelen komt er een abstractie laag over de hardware. Hierboven is een voorbeeld klassediagram. Echter hebben wij er voor gekozen dit niet te implementeren om wij te weinig tijd over hadden, Interrupts niet werken buiten main.c en wij moeilijk de registers konden aanpassen buiten het hoofdbestand. Verder vonden wij ook dat voor veel onderdelen een klasse niet nodig waren omdat ze specifiek ingesteld moeten worden met specifieke registers die lastig te implementeren in een klasse zonder te hardcoden.

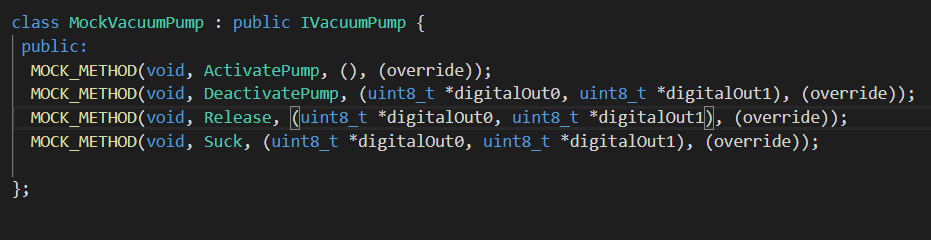
# Unit testing

Een manier om programma’s te testen is unit testing. Unit tests zijn kleine tests gericht om een klein deel van je code te testen. Door deze kleine tests te schrijven kun je gemakkelijk controleren of je code nog het gewenste resulte resultaat levert. Voor het unit testen van c plus plus maken wij gebruik van google test. Dit test framework is erg fijn want naast het gewone unit testen je kunt hier klassen mee emuleren oftwel “mocken”. Erg fijn als je hardware moet testen maar niet wilt dat de hardware werkelijk wat doet. Hier onder is een voorbeeld over hoe wij een mock klasse maken en deze testen.

Eerst maken we boven op de klasse een interface. Je hebt een interface nodig om hiervan een mock klasse te kunnen maken.



Daarna maken we de mock aan en hierin zetten we alle overrides in en maken we de methodes van de mock aan.



Daarna maken we de test die we gaan runnen. Hier onder zie je dat we verwachten dat de ActivatePump methode een keer aangeroepen wordt. Ook verwachten we dat de StartSucking methode true returned.

