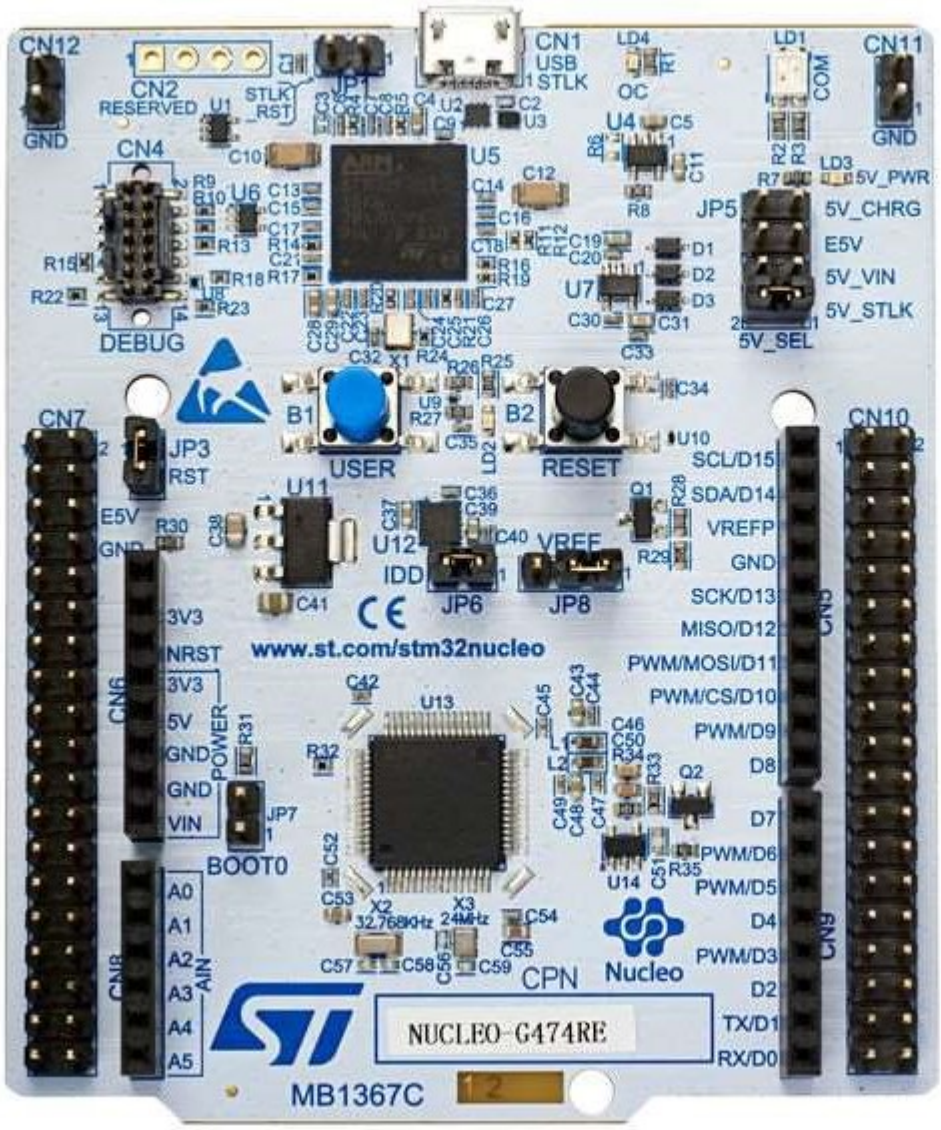
ES portfolio



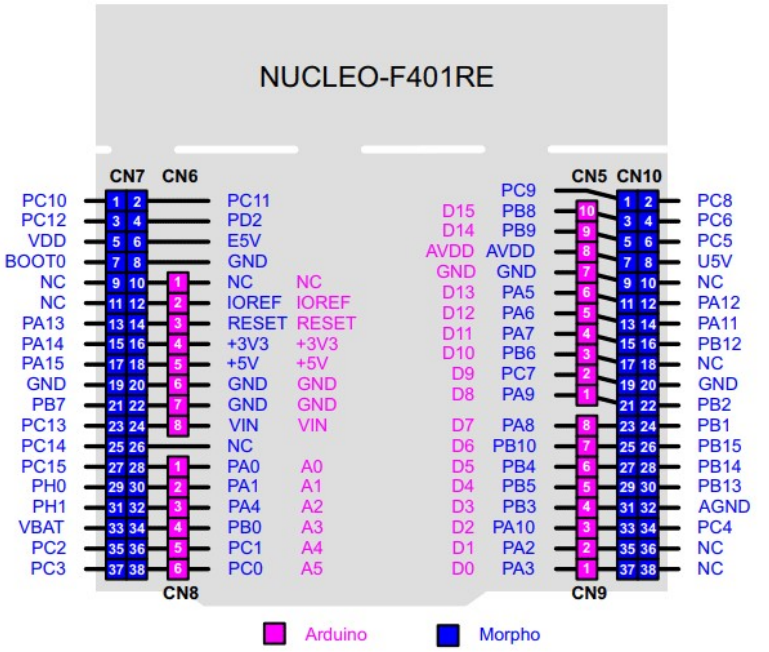
Gemaakt door: Daphne Gijsbers

Versie	Datum	Toevoeging
1.0	5-12-2022	Interrups, Encoder
1.1	19-12-2022	Low power Mode, Aanpassing encoder
1.2	6-1-2023	Timers
1.3	9-1-2023	RTOS
1.4	16-1-2023	PID, Eindopdracht
1.5	17-1-2023	Eindproject feedback

Inhoud

1.0 Onderzoek STM32

Ik ben begonnen met het onderzoeken hoe het STM32 board werkt omdat het heel anders dan arduino door de 32 bits in plaats van 16 bits.



In de afbeelding hierboven zijn alle pinouts te zien, ik ben begonnen met de GPIO A pinnen behalve PA2 en PA3 die worden al gebruikt door het bordje zelf.

2.0 Aansturing LED

Bij de aansturing van een led moet je 3 registers gebruiken. Eerst zet je de MODER op de pin waar je led is aangesloten is dus bij ons project is dat PA0, alleen bij het MODER register gaat het register per 2 bits dus moet je als je naar een hogere PA gaat dan moet je het register met bit shiften per 2 bits op de goede plek zetten. Daarna moet je de OTYPER register gebruiken om de output te bepalen die kan je zetten op …. Of …... Vervolgens als je dan het ledje aan of uit wilt zetten moet je gebruik maken van het output data register (ODR) om het ledje aan te doen moet je in het ODR bitshiften met een OR naar de goede PA. Om het ledje uit te zetten doe je eigenlijk hetzelfde alleen in plaats van dat je OR gebruikt gebruik je dan een AND.

```
#include "led.h"
#include "gpio.h"

Led::Led(int pin)
{
  GPIOA->MODER = (GPIOA->MODER & ~(0b11 << (pin*2))) | (0b01 << (pin*2));
  GPIOA->OTYPER &= ~(0b1 << pin);
  this->pin = pin;
}
void Led::On()
{
    GPIOA->ODR |= (GPIO_ODR_0 << pin);
}
void Led::Off()
{
    GPIOA->ODR &= ~(GPIO_ODR_0 << pin);
}
```

Om er voor te zorgen dat je niet bij elk ledje alle registeren opnieuw moet zetten, heb ik er een class van gemaakt die om het pin nummer vraagt waar je het ledje op hebt aangesloten en dat nummer gebruikt hij vervolgens om alles wat hierboven beschreven is goed te zetten.

3.0 Aansturing Button

Een button was weer iets anders omdat dat een input is en geen output dat zorgt ervoor dat je weer andere registers moet gaan gebruiken. Je gebruikt net als de led wel de MODER alleen die moet je nu op input zetten in plaats van output. Bij een button moet je kiezen of je pull-up of pull-down button wilt hebben dat kan je zetten in het PUPDR register, ik heb voor pull-up gekozen. Om de button uit te lezen moet je het input data register gebruiken en als je die dan gelijk is met 1 dan betekend dat hij ingedrukt is.

```
#include "button.h"
#include "gpio.h"

Button::Button(int pin)
{
    GPIOA->MODER = (GPIOA->MODER & ~(0b11 << (pin*2))) | (0b00 << (pin*2));
    GPIOA->PUPDR = (GPIOA->PUPDR & ~(0b11 << (pin*2))) | (0b01 << (pin*2));
    this->pin = pin;
}
```

Ook bij de button heb ik een class gemaakt zodat als ik met meerdere buttons makkelijk kan definiëren en dat ik niet bij elke button de registeren moet zetten.

4.0 Interrupts

Interrups worden gebruikt om er voor te zorgen dat als je iets doet in mijn applicatie op de button drukt dat hij dan in plaats van dat hij eerst de loop afmaakt en dan pas naar andere code gaat, gaat hij meteen naar de interrupt code. Die handelt hij af en dan gaat hij precies terug in de code waar hij gebleven was. Dit is handig om er voor te zorgen dat als er iets meteen afgehandeld moet worden bij een bepaalde actie dat je niet hoeft te wachten.

Nadeel is van de interrups op het STM32 bordje dat alle P0 pinnen op dezelfde EXTI0 zitten waardoor je over al die pinnen maar 1 interrupt kan zetten en door dat geeft dat een maximum aantal interrups die je kan doen voor alle pinnen.

Om een interrups te maken moet je een extra methode maken die extern “C” void heet, daarin moet je de interne flag die hoog staat door het aanroepen van de interrupt weer resetten omdat anders zodra hij uit de interrupt handler is gaat hij er meteen weer in door de hoge flag. Om die te kunnen resetten moet je in het register PR weer laag zetten.

```
extern "C" void EXTI1_IRQHandler(void)
{
  flag = true;
  EXTI->PR = EXTI_PR_PR1;
  snprintf(msgBuf, MSGBUFSIZE, "%s", "Interrupt\r\n");
  HAL_UART_Transmit(&huart2, (uint8_t *)msgBuf, strlen(msgBuf), HAL_MAX_DELAY);
}
```

Ik heb in de interrupt handler een globale flag aangeroepen en die op true gezet zodat ik daar in mijn main code nog kan gebruiken om een led aan of uit te zetten.

5.0 Low power mode

Low power mode oftewel sleepmode zorgt ervoor dat de power van de MCU heel laag word. Je kunt dit doen door na elke iteratie _WFI() te doen, als je dit doen dan word de MCU pas weer wakker na een interrupt en dan na de iteratie dan gaat de MCU weer slapen.

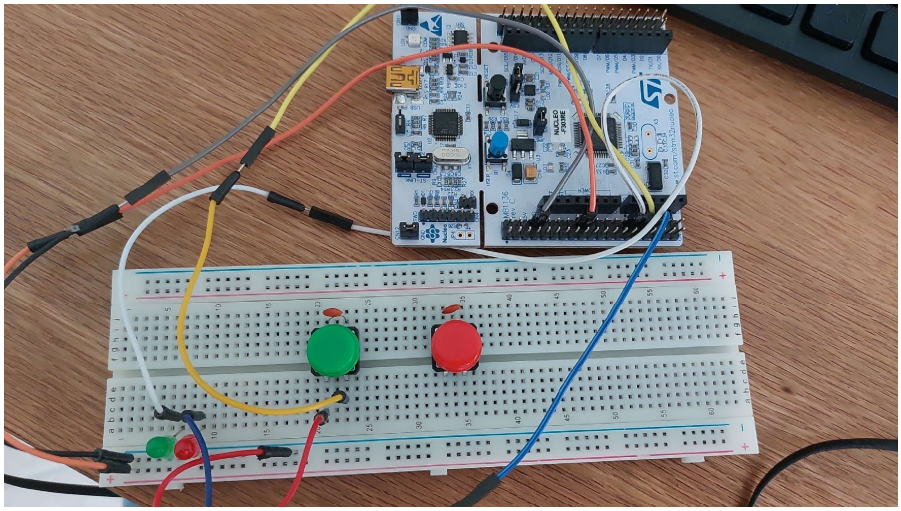
Ook kun je _WFE() doen dat zorgt ervoor dat je zelf een wake up event kan maken die er voor zorgt dat de MCU weer wakker word

```
// }
led_green.On();
HAL_Delay(1000);
led_green.Off();
HAL_Delay(1000);

__WFI();
}
```

6.0 Mijn applicatie

Mijn applicatie zorgt ervoor dat een ledje blijft knipperen in een loop. Maar als je dan op een knop drukt dan gaat hij een interrupt in die een globale flag hoog zet zodat hij die kan gebruiken in zijn code. Als hij dan verder in zijn code gaat dan gebruikt hij die flag om tegelijk met de andere led nog een led te laten knipperen.

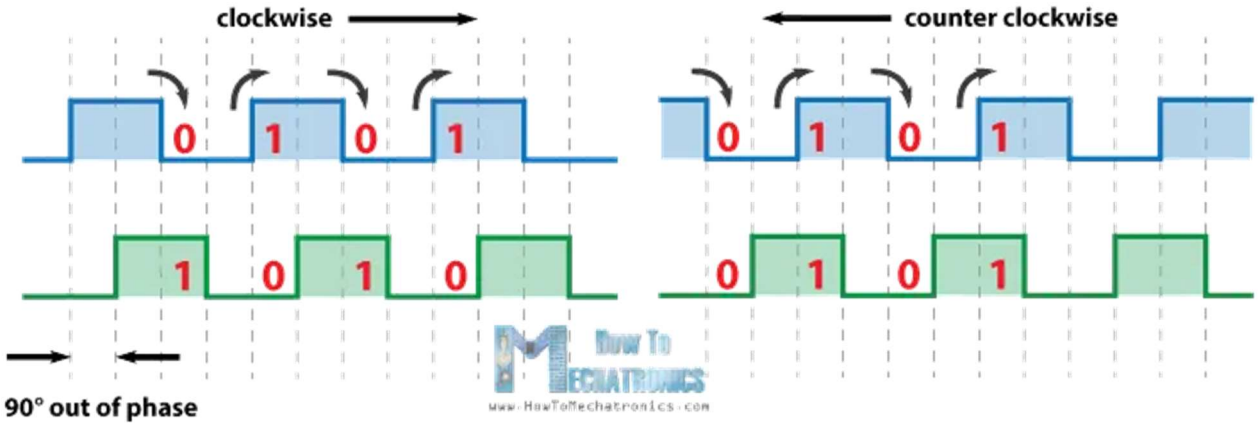
```
while (1)
{
  led_red.Off();
  if(flag == true)
  {
    flag = false;
    led_red.On();
  }
```



7.0 Encoder

7.1 Directie van de encoder

De encoder roept een interrupt aan bij elke slag die hij draait linksom of rechtsom. Alleen om de draairichting te bepalen van de encoder moet je de data pin en de clock pin uitlezen aan de hand van die lijnen kan je bepalen welke kan hij opdraait



In het bovenstaande plaatje kan je de data pin en de clockpin goed zien, daarbij zie je dat als allebei de lijnen de zelfde waarde hebben 0/1 dan draait de encoder linksom en als de waardes anders zijn van elkaar dan draait de encoder rechtsom

```
extern "C" void EXTI1_IRQHandler(void)
{
  EXTI->PR = EXTI_PR_PR1;
  if ((GPIOA->IDR & 0b1 << ENCODER_DATA) > 0)
  {
    encoderLastDir = -1;
  }
  else
  {
    encoderLastDir = 1;
  }
  encoderTicked = 1;
}
```

In bovenstaande plaatje zie je de interrupt handler van de rotery encoder direction, daarbij word gekeken of de clock en data pin dezelfde waarde hebben, als dat zo is dan word de encoderLastDir op -1 gezet die later in de while loop nog gebruikt gaat worden. Mochten de waardes niet gelijk aan elkaar zijn dan word de waarde 1. Ook word encoderTicked op 1 gezet zodat hij in de while weet dat er aan de encoder gedraaid is.

```
if (encoderTicked)
{
  encoderTicked = false;
  selectIndex += encoderLastDir;

  if (encoderLastDir == 1)
  {
    GPIOA->ODR ^= 0b1 << CAL_LED;
  }
  else
  {
    GPIOA->ODR ^= 0b1 << INDICATOR_LED;
  }
```

Dit plaatje laat zien wat er in de while loop in de main gebeurd met de eerder genoemde waardes. Daarbij word encoderTicked (soort flag) weer op false gezet zodat hij niet weer de if statement ingaat. Daarna word de index van de encoder bijgehouden door de waarde 1/-1 bij de oude index waarde op te tellen, dat zorgt ervoor dat je weet op welke positie de encoder staat. Ook laat ik een ledje branden als de encoder linksom draait of een ander ledje als hij rechtsom draait.

7.2 button encoder

Ook zit er nog een button op de encoder als je op het draaistaafje drukt, daar heb ik ook een interrupt aangehangen.

```
extern "C" void EXTI4_IRQHandler(void)
{
  EXTI->PR = EXTI_PR_PR4;
  encoderClicked = true;
}
```

Ook daar word de interrupt flag weer gereset en maar ik een soort globale flag aan encoderClicked die ik later kan gebruiken in de while loop in de main.

```
if (encoderClicked)
{
  encoderClicked = false;
  GPIOA->ODR ^= 0b1 << CAL_LED;
```

In het bovenstaande plaatje zie je het if statement die gebruik maakt van de globale flag die gezet is in de interrupt, als je de encoder inklikt gaat hij het if statement in zet hij de globale flag weer op false en zet hij de een ledje aan.

8.0 Timers

Om timers te gebruiken met PWM output voor het aansturen van een servo, gebruiken we een rotary encoder die eerder ook al gebruikt is om de positie van de servo te bepalen. Maar eerst moet de timer geinitializeerd worden, als eerste timer word er gekozen voor een 16 bits timer. Daarbij zetten we de prescaler op 72 om een 1MHz clock en het ARR register op 1/50 = 0.02 ms = 20000 us om een signaal te krijgen van 50 Hz.

```
RCC->APB1ENR &= ~0b1 <<RCC_APB1ENR_TIM3EN_Pos;
RCC->APB1ENR |= 0b1 <<RCC_APB1ENR_TIM3EN_Pos;
TIM3->PSC = 72;
TIM3->ARR = 20000;
TIM3->CCMR1 &= ~0b11;
TIM3->CCMR1 &= ~0b1111 <<TIM_CCMR1_OC1M_Pos;
TIM3->CCMR1 |= 0b0110 <<TIM_CCMR1_OC1M_Pos;
TIM3->CCR1 = 1000;
TIM3->CCER |= 0b1;
TIM3->CR1 |= 0b1;
```

In de datasheet voor de SG90 staat dat hij tussen de 1000 en de 2000 us pulsen in lengte heeft en dat staat gelijk aan 0 tot 180 graden.

Om de servo aan te sturen moest er bij de encoderTiked() functie het volgende bij komen te staan.

```
servoModifier = (servoModifier+(encoderLastDir*10))%1000;
TIM3->CCR1 = servoModifier + 1000;
```

Door direct naar het CCR1 register te schrijven met een waarde van minimaal 1000 en in stapjes van 1000 omhoog gaat. Het aantal graden waar de servo op moet staan word bepaald door de servoModifier die bij elke puls verhoogt word met 1000 en de modolo zorgt ervoor dat de waarde nooit boven de 2000 komt.

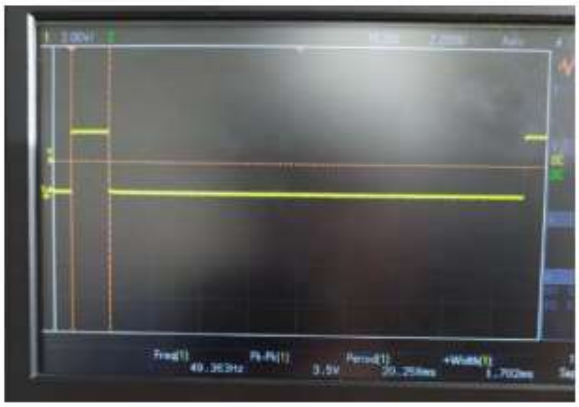
```
void setTimerPulsewidth(uint timerNumber, int widthInUS){
  TIM_TypeDef *timer = resolveTimer(timerNumber);
  timer->CCR1 = widthInUS;
}

void setTimerPeriod(uint timerNumber, int periodInUS, bool overridePulseWidth=true){
  TIM_TypeDef *timer = resolveTimer(timerNumber);
  timer->ARR = periodInUS;
  if(overridePulseWidth)timer->CCR1 = periodInUS/2;
}
```

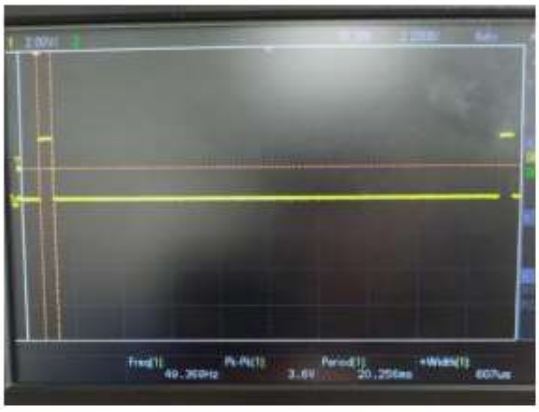
Deze functie in het bovenstaande plaatje zorgen er beide voor de de pulse width en de period of de timer. Deze gebruiken gelijkwaardige dingen zoals typedef en resolver stuctuur zodat de code compact en leesbaarder word.

Met het toevoegen van extra defines word de code veel leesbaarder. Deze defines zorgen ervoor dat je makkelijk kan kiezen of een pin een input/output of een andere gedineerde optie in de function.

```
#define GPIO_INPUT 0
#define GPIO_OUTPUT 1
#define GPIO_ALT 2
#define LOW 0
#define HIGH 1
#define TOGGLE 2
```

Na het meten van de PWM pin van de servo met een oscillioscope laat het volgende zien na het draaien van de rotary encoder: pulse width slinkt van 1.702 mS naar 607 uS





Om timers in 2 varianten te kunnen aantonen werd er gevraagd om een vorm of direct controle over een periode van de timer, met een interrupt werd er voor een korte periode controle genomen over de microcontroller. Eigenlijk was het na het maken van de timers in combinatie met de servo een heel klein beetje code aanpassen. Ook zou het de functie van de servo niet in de weg zitten.

```
timer->CR1 |= 001;
timer->EGR |= TIM_EGR_UG;
timer->SR = ~TIM_SR_UIF;
}
```

Eerder in dit hoofdstuk werd de initialize van de timers al besproken, maar voor deze extra timer aantoning moeten bovenstaande 2 regels toegevoegd worden. Deze code veranderd niks aan de PWM mode omdat we de EGR register voor de timer aanroepen. Hij genereert dan events en reset de interrupt flag. Allebei deze registers veranderen niks aan de PWM omdat de interrupts nog steeds masked zijn en niks triggeren intern.

```
NVIC_EnableIRQ(TIM3_IRQn);
TIM3->DIER |= (1<<TIM_DIER_UIE_Pos);
```

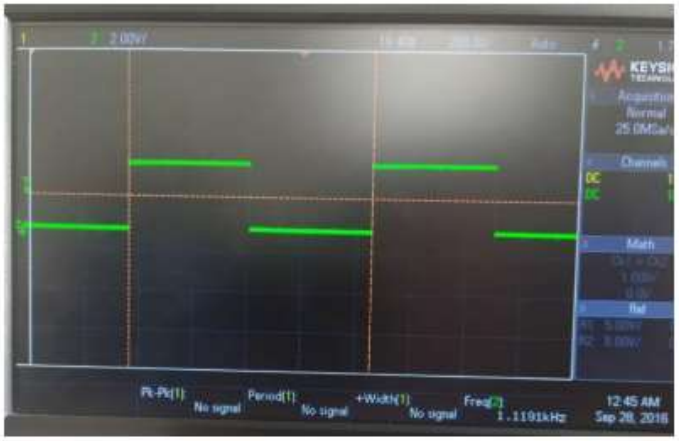
Bovenstaande lijnen zetten de IRQ aan en unmasken de interrupt. Vanaf nu hebben de interrupts effect op het systeem.

```
extern "C" void TIM3_IRQHandler(void)
{
  TIM3->SR = ~TIM_SR_UIF;
  setPin(SERVO_OUT+1,TOGGLE);
}
```

Om de periode van de timer te veranderen. Veranderen we de timerSetPulseWidth of de eerder genoemde encoder tick event van de timerSetPeriod commando. Zorgen dat de PWM niet overschreven word is nu de functie zijn default.

```
servoModifier = (servoModifier+(encoderLastDir*40))%1000;
//setTimerPulsewidth(3,servoModifier+1000);
setTimerPeriod(3,servoModifier+1000,false);
```

Om het te kunnen aantonen hebben we het signaal gemeten met de oscilloscoop daarbij zie je dat de eerste meting ongeveer 879 Hz is en na een tijdje het signaal 1.1191 kHz is.



9.0 RTOS

Ik ben begonnen met het uitzoeken wat free RTOS is en wat het basis concept is van met meerdere threads werken. Daarbij kwam ik erachter hoe handig het is om verschillende threads tegelijk te kunnen gebruiken zonder dat je systeem moet wachten met iets afhandelen tot hij bij die functie aan komt, dit zorgt er ook voor dat je eventuele meetwaardes accurater worden.

Daarna ben ik met de voorbeelden op de RTOS site, de powerpoint en op advies van de docent makkelijk begonnen en daarop steeds verder gaan bouwen.

```
osMutexId_t led_delay;

uint16_t leddelay =250;

const osMutexAttr_t Thread_Mutex_attr = {
  "ledDelay",
  osMutexPrioInherit,
  NULL,
  0U
};

const osThreadAttr_t defaultTask_attributes = {
    .name = "defaultTask",
    .attr_bits = osThreadDetached,
    .cb_mem = NULL,
    .cb_size = 0,
    .stack_mem = NULL,
    .stack_size = 128 * 4,
    .priority = (osPriority_t)osPriorityNormal,
    .tz_module = 0,
    .reserved = 0};
```

Om alles klaar te zetten voor de threads en de mutexen gebruik ik de default mutex en thread attributen, de thread attributen worden gegeven in de freertos.cpp file. De mutex is speciaal gemaakt voor de LED om mutexen aan te kunnen tonen.

```
 SystemClock_Config();

 MX_GPIO_Init();
 MX_USART2_UART_Init();
 osKernelInitialize(); /*
```

Aan het begin van de main functie moet je osKernelInitialize() zetten om er voor te zorgen dat freeRTOS klaar staat om te kunnen gebruiken.

```
osThreadNew(mainThread,  NULL, &defaultTask_attributes);
osThreadNew(outputThread,NULL, &defaultTask_attributes);
```

Eerst ben ik begonnen met een main thread en een output thread die er voor gaan zorgen dat de 2 ledjes gaan knipperen tegelijk.

```
void secondThread(void *argument){
  int internalDelay=250;
  while(1){
    setPin(INDICATOR_LED,TOGGLE);
    osMutexAcquire(led_delay,osWaitForever);
      internalDelay=leddelay;
    osMutexRelease(led_delay);
    osDelay(internalDelay);
  }
}

void mainThread(void *argument){
  led_delay = osMutexNew(&Thread_Mutex_attr);
  while(1){
    setPin(CAL_LED,TOGGLE);
    osDelay(500u);
  }
}
```

Hierboven zie je de functies van de eerder genoemde threads. In de Mainthread word de mutex gemaakt en in de secondthread gebruikt die mutex als guard voor de leddelay variabele. Die variabele word later veranderd door de rotery encoder dat kan je zien door de snelheid waarmee de led knippert. Dit om de code meer complex te maken door het gebruik van interrupts.

```
osKernelStart(); /* Start scheduler */
```

Om dan de hele setup van RTOS klaar te hebben moet je osKernalStart() om er voor te zorgen dat de threads beginnen met lopen.

Als je de LEDs meet op de oscilloscoop zien we dat de indicator_led (groene led) twee keer zo snel knippert als de cal_led (gele led). Dit om aan te tonen dat de thread allebei werken en tegelijk lopen.

```
void inputThread(void *argument){
  while(1){
    if(encoderTicked)
    {
      GPIOA->ODR ^= (1 << 5);
      encoderTicked=false;
      osMutexAcquire(led_delay,osWaitForever);
        leddelay += encoderLastDir*10;
        osDelay(1000);
      osMutexRelease(led_delay);
      encoderLastDir=0;
    }
    osDelay(1);
    IWDG->KR=0xAAAA;
  }
}
```

De volgende stap was om een input thread te maken die ook de eerder genoemde mutex gebruikt voor de leddelay variabele. Hij veranderd de variabele en laat het weer gaan nadat hij hem heeft gebruikt. Deze thread zorgt er ook voor dat de mutex niet gebruikt vord in de encoder ISR.

In de guarded deel van de mutex er is een regel osDelay(1000). Dit laat zien dat als dit deel aangeroepen word het knipperen van de LEDs stopt voor een seconde en daarna weer verder gaat. Het stopt omdat de secondThread aan het wachten is op de input thread tot hij de mutex weer los laat. Wanneer hij losgelaten is dan gaan de LEDs weer knipperen op hun nieuwe frequentie.

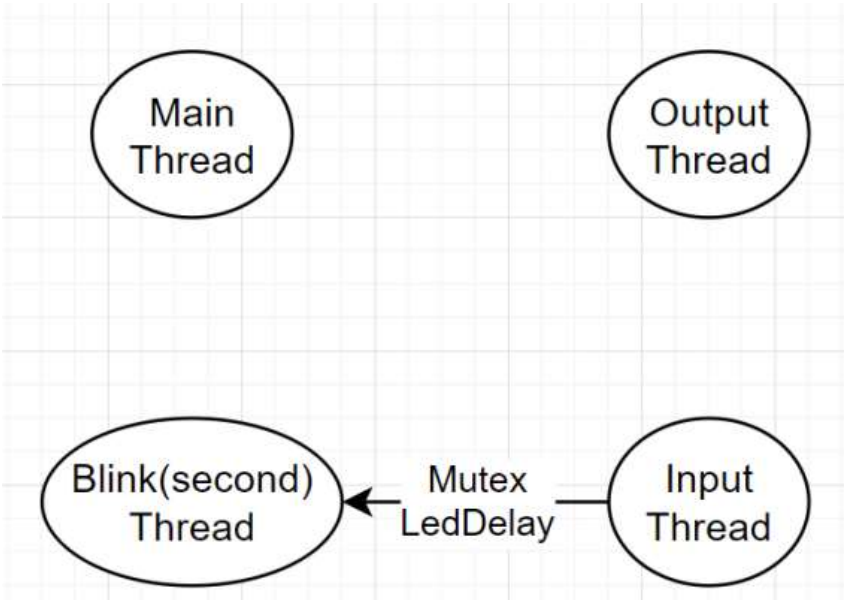
Als je meet op de indicator_led lijn dan kan je zien dat cal_led blijft knipperen en dat de indicator_led 1,5 seconden vast hangt en daarna weer verder gaat op een andere frequentie.

Om te kunnen debuggen heb ik een vierde thread aangemaakt als serial output lijn.

```
void outputThread(void *argument){
  char txBuf[80];
  snprintf(txBuf,80,"HJALLO!");
  HAL_UART_Transmit(&huart2,(uint8_t *)txBuf, strlen(txBuf), HAL_MAX_DELAY);
  while(1){
    if(encoderTicked){
      snprintf(txBuf,80,"Ticked!");
      HAL_UART_Transmit(&huart2,(uint8_t *)txBuf, strlen(txBuf), HAL_MAX_DELAY);
    }
  }

}
```

Deze thread zorgt ervoor dat hij een welcome bericht stuurt als het systeem draait en daarmee kan je zien dat de threads running zijn. Op het moment print hij ook ticked als de encoder ticked deze functie reset daarintegen niet de variabele dus de input thread moet nog steeds de functie afhandelen wanneer de variabele hoog is.

Deze functie word vooral gebruikt om watchdog aan te kunnen tonen.



In bovenstaand plaatje kun je zien wat voor een threads er op het systeem lopen op het moment en wat eventueel gebruik maakt van elkaar.

De InputThread krijgt de watchdog reset sleutel omdat deze thread elke milliseconde runt

```
osDelay(1);
IWDG->KR=0xAAAA;
```

We hebben de prescaler berekend op 128 en de RLR waarde op 3 voor de watchdog tijd zoals in de presentatie aanbevolen werd. De eind frequentie kwam daarmee op 104 ms.

Voordat deze waardes ingevuld worden moeten we eerst toegang vragen en opgeven van de watchdog settings dat doen we door de KR Sleutels te zetten. Voordat we toegang opgeven moeten we het KR resetten en daarmee de hele watchdog.

```
IWDG->KR= 0x5555;
IWDG->PR = 0b101;
IWDG->RLR = 3;
IWDG->KR = 0x0000;
IWDG->KR = 0xCCCC;
```

Om er voor te zorgen dat de watchdog niks aanpast in het normale systeem gebruiken we osSleep(1000) om er zeker van te zijn dat een thread niet een hold raised van langer dan de watchdog tijd.

Na het runnen van de code en het draaien van de encoder zie je het volgende op de serial:

```
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
HJALLO!Ticked!Ticked!Ticked!Ticked!Ticked!Ticked!Ticked!Ticked!
```

Het bordje start op te zien aan de HJALLO! zoals beschreven in de Serial Thread en daarna met het elke tick die de encoder maakt print de serial thread Ticked!.

Als we nu de eerder ingezette osDelay(1000) eruit halen dan is het volgende te zien op de serial:

```
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
HJALLO!Ticked!HJALLO!Ticked!HJALLO!Ticked!HJALLO!Ticked!HJALLO!
```

Nu na elke tick van de encoder de thread word telang vastgehouden en dat zorgt ervoor dat de watchdog een reset doet. Dat zie je doordat na elke Ticked! bericht meteen het welkom bericht HJALLO! weer komt. Dit werkt omdat de watchdog word gereset in de thread die vastgehouden word. Als je de reset naar een andere thread verhuist die ook een osSleep(0 van 1 ms heeft dan zou de watchdog nog steeds resetten en de hold in de input thread zou niks aanpassen aan de watchdog.

10.0 PID closed loop

Om closed loops aan te kunnen tonen gebruik ik een parallax 360 graden feeback servo in combi met PWM input timer.

Eerst moet er een nieuwe timer aangemaakt worden die op PWM input gaat werken, Daarvoor word TIM4 gebruikt een 32 bit timer. Omdat er bij PWM input meer verschillende instellingen ingesteld moeten worden word de orginele TimerInit() functie uitbereid. Tijdens het volgen van de stappen in de powerpoint hadden we de ARR register op 0 gezet dat zorgt ervoor dat de clock count op 0 gezet word en dan gereset. Om het werkend te krijgen mag je de ARR niet instellen zodat hij op zijn maximum waarde blijft en kan hij werken zoals bedoeld.

```
void timerInit(uint timerNumber, uint prescaler, u
  TIM_TypeDef *timer = resolveTimer(timerNumber);
  uint RCCOffset;
  switch(timerNumber){
    case 3: RCCOffset=RCC_APB1ENR_TIM3EN_Pos; brea
    case 4: RCCOffset=RCC_APB1ENR_TIM4EN_Pos; brea
  }
  RCC->APB1ENR &= ~(0b1 <<RCCOffset);
  RCC->APB1ENR |= 0b1 <<RCCOffset;
  timer->PSC = prescaler;
  if(timerNumber==3){
    timer->ARR = period;
    timer->CCMR1 &= ~0b11;
    timer->CCMR1 &= ~0b1111 <<TIM_CCMR1_OC1M_Pos;
    timer->CCMR1 |= 0b0110 <<TIM_CCMR1_OC1M_Pos;
    timer->CCR1 = initialPulseWidth;
    timer->CCER |= 0b1;
  }
  if(timerNumber==4){
    timer->CCMR1 &= ~(0b11 << TIM_CCMR1_CC1S_Pos);
    timer->CCMR1 |=  0b01 << TIM_CCMR1_CC1S_Pos;
    timer->CCER  &= ~(0b1 << TIM_CCER_CC1P_Pos);
    timer->CCER  &= ~(0b1 << TIM_CCER_CC1NP_Pos);
```

Na de timerInit() functie kan de timer ingesteld worden. Om timer 4 op channel 1 te zetten moest de timer op pin 11 aangesloten worden.

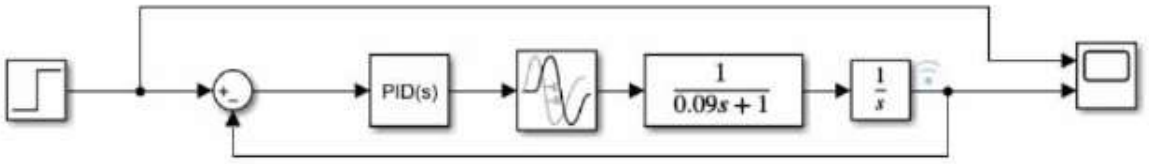
```
timerInit(4,72,0,0); //feedback channel
GPIOA->AFR[1] &= ~(0b1111 << GPIO_AFRH_AFRH3_Pos);
GPIOA->AFR[1] |= 0b1010 << GPIO_AFRH_AFRH3_Pos;
```

Om de count en de CCR registers te kunnen bekijken kon ik een paar debug lijnen toevoegen aan de al bestaande output UART thread. Elke seconde worden de count en de CCR registers 4 keer geprint.

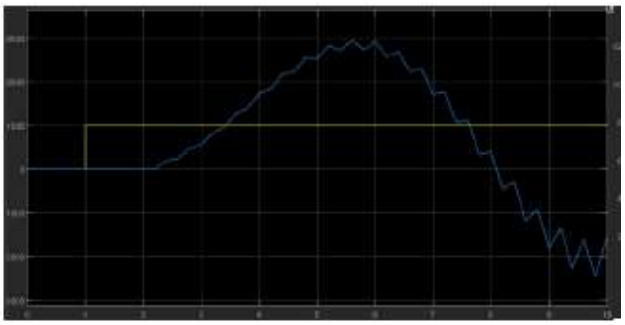
```
uint32_t debugCCR1 = TIM4->CCR1;
uint32_t debugCCR2 = TIM4->CCR2;
uint32_t debugCNT = TIM4->CNT;
if(cycles>=200){
  snprintf(txBuf,80,"pos: %lu, TGT: %lu, SPD: %li\n",TIM4->CCR2, servoTarget
  HAL_UART_Transmit(&huart2,(uint8_t *)txBuf, strlen(txBuf), HAL_MAX_DELAY);
  cycles=0;
} else {
  cycles = cycles+1;
```

Nu we alle waardes kunnen lezen kan de PID code gemaakt worden, om achter de PID waardes te komen voor de servo maken we een simulatie in simulink van matlab. In de simulaie maken we gebruik van de theta en de tau waardes van het model van de gegeven excel bestand deze waren ongeveer 0,04 en 0,09.

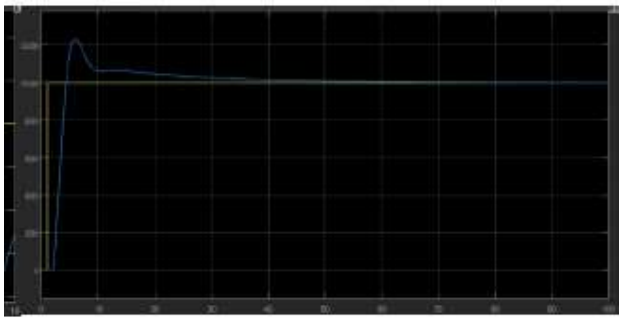
We hebben het model nagemaakt vanuit de powerpoint en alle waardes ingevoerd.



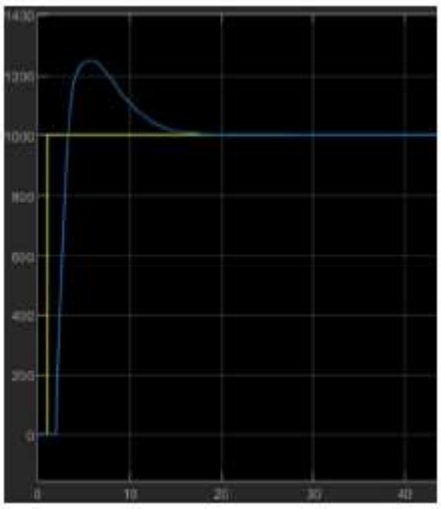
In stappen gingen we tunen zoals te zien in de afbeeldingen onderstaande:



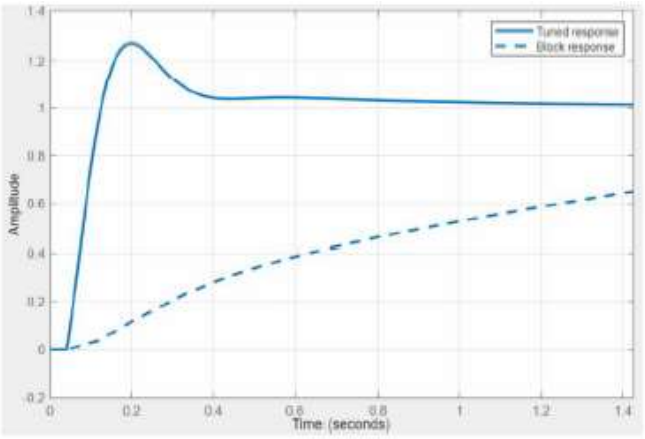
Figuur 1 Eerste scope zonder tuning



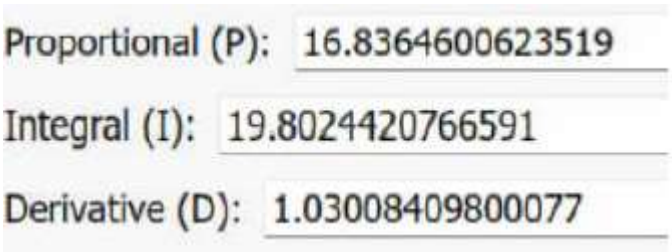
Figuur 2 Voorzichtige eerste tuning



Figuur 3 responsing tijd verminderd



Figuur 4 Laatste tuning



Figuur 5 Eind PID waardes

Na het draaien van de simulatie en het tunen van de PID, moeten de PID loop gaan maken. Om deze loop te kunnen draaien moeten we nog een nieuwe thread maken waarin we de nieuwe PID waardes zetten.

```
void PIDThread(void *argument){
  PIDA_value_mutex = osMutexNew(&MsgQueueMutexAttr);

  float Kp =15.83646006235l9;
  float Ki =19.8024420766591;
  float Kd =1.03008409800077;
  int32_t integral=0;
  int32_t derivative=0;
  int32_t error=0;
  int32_t previous_error=0;
```

Extra variabelen worden toegevoegd voor de berekeningen later in de functie. Je kunt zien dat de kp lager is dan origineel in de simulatie dit komt omdat er nog een beetje handmatig getuned is omdat de werking dan beter is.

De PID code komt van het PID document geleverd op de git waar de volledige pseudocode in staat.

```
while(1){
  error = (TIM4->CCR2) - (servoTarget);
  integral = integral + error;
  if (error == 0)
  {
    integral = 0;
  }
  if ( abs(error) > 40)
  {
    integral = 0;
  }
  derivative = error - previous_error;
  previous_error = error;
  speed=(Kp*error + Ki*integral + Kd*derivative)/20;
  if(speed>220)speed=120; //was 220
  if(speed<-220)speed=-120;
  setTimerPulsewidth(3,1500+speed);
```

We kwamen erachter dat de snelheid gecapped moest worden in verschillende manieren, eerst delen door 20 en dan een limiet zetten op 120.

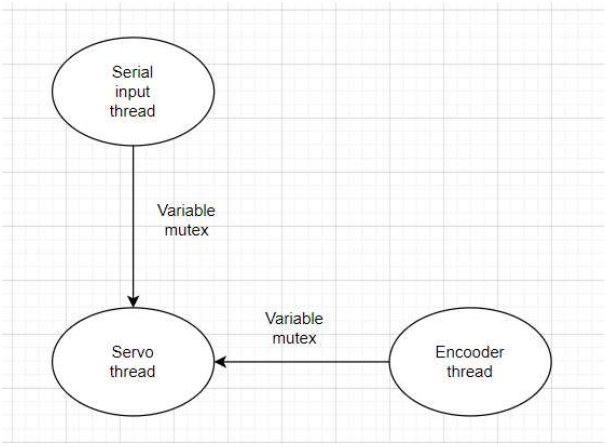
Als dit niet gebeurde dan zal de servo altijd zijn waarde overschieten. Het systeem loopt op een lagere snelheid maar kan nu naar de positie draaien die gevraagd word en daar ook blijven staan.

Dit is een voorbeeld van de serial printerwaar de target 2 keer is verhoogd. De snelheid verhoogt verder dan de reactie punt van de servo en gaat bewegen richting het target. Omdat hij zo langzaam beweegt is de overshoot bijna niet zichtbaar, bijvoorbeeld als de servo naar 110 graden moet dan eindigd hij op 112 als overschoot. Met de snelheid of -2 ka de servo niet meer bewegen en blijft hij staan op de target positie.

```
pos: 99, TGT: 100, SPD: 1
pos: 99, TGT: 100, SPD: 1
pos: 100, TGT: 100, SPD: 0
pos: 99, TGT: 100, SPD: 1
pos: 99, TGT: 100, SPD: 1
pos: 104, TGT: 110, SPD: 4
pos: 112, TGT: 110, SPD: -2
pos: 112, TGT: 110, SPD: -2
pos: 111, TGT: 110, SPD: -1
pos: 114, TGT: 120, SPD: -6
pos: 120, TGT: 120, SPD: 0
pos: 120, TGT: 120, SPD: 0
```

11.0 Eindproject

Als eindproject hadden we afgesproken om een project te maken waarbij we PID en RTOS en Timers konden aantonen, dat werd uiteindelijk een systeem dat de positie en de PID waardes van de servo kon aanpassen via de rotery encoder of via de serial monitor. Met gebruik van een variable mutex.



Er is begonnen met een paar variable die worden beschermd met een mutex.

```
osMutexId_t PIDA_value_mutex;

uint16_t leddelay =250;

float Protected_Kp =15.8364600623519;
float Protected_Ki =19.8024420766591;
float Protected_Kd =1.03008409800077;
volatile bool PIDValueUpdateFlag = false;
```

Onder het PID regulatie deel is er een deel toegevoegd die reageert op de PIDAupdateFlag variable die de PID waardes update elke keer als hij aangeroepen word.

```
//Daphne's solution)
if(PIDValueUpdateFlag){

  osMutexAcquire(PIDA_value_mutex,osWaitForever);
    Kp = Protected_Kp;
    Ki = Protected_Ki;
    Kd = Protected_Kd;
  osMutexRelease(PIDA_value_mutex);

  PIDValueUpdateFlag=false;
}
```

De PID thread kan nu de waardes updaten met behulp van de mutex variables, omdat alle waardes tegelijk updaten is er maar 1 mutex nodig voor alle verschillende variables.

Nu moet er nog een manier gemaakt worden dat er input vanuit de UART nieuwe waardes ingetypt moeten kunnen worden die dan door het systeem gebruikt gaan worden, hiervoor moest er een Serial input thread aangemaakt worden.

```
void UARTInputThread(void *argument){
  char receivedMsg[80];
  int msgPointer=0;
  uint8_t rxChar;
  HAL_StatusTypeDef status;

  //program reception of msges here.

  while(1){

    USART2->ICR |= USART_ICR_ORECF;
    status = HAL_UART_Receive(&huart2, &rxChar, 1, 0);
    if (status == HAL_OK) // No HAL_TIMEOUT, so a character is read.
    {
```

Het eerste deel volgt de standaard niet blokerende HAL_UART_Receive implementatie die in de PowerPoint beschreven werd.

```
if(rxChar=='\n'||rxChar=='\r'){
  receivedMsg[msgPointer++]='\0';
```

Als het character dat ontvangen is een endline character is dan verwerkt hij de regel die hij heeft door een null character in de receivedMsg te zetten om de string te eindigen.

```
} else {
  receivedMsg[msgPointer++]=rxChar;
}
```

Als het een normaal character is dan word hij toegevoegd aan de receivedMsg en de pointer verhoogd met 1.

In het verwerken van het character, elk character is gechecked of het een point character is of niet. Als het zo is dan zal de waarde behandeld worden als een float later in die functie.

```
int value=0;
float fValue=0;
bool floatFlag=false;
char valueText[80];
int count=0;

for(int i=1; receivedMsg[i]!=NULL; i++){
  if(receivedMsg[i]=='.'||receivedMsg[i]==','){
    floatFlag=true;
  }
  valueText[i-1]=receivedMsg[i];
  count++;
}
```

De characters worden ook in de valueText gezet, behalve het eerste character.

```
valueText[count]='\0';
if(floatFlag){
  fValue=atof(valueText);
} else {
  value=atoi(valueText);
}
```

De valueText string is gecapped en de waardes in tekst worden veranderd in numerieke waardes liggende aan welk type.

```
osMutexAcquire(PIDA_value_mutex,osWaitForever);
  switch(receivedMsg[0]){
    case 'P': Protected_Kp = fValue; break;
    case 'I': Protected_Ki = fValue; break;
    case 'D': Protected_Kd = fValue; break;
    case 'A': servoTarget = value; break;
  default: break;
  }
osMutexRelease(PIDA_value_mutex);
PIDValueUpdateFlag=true;
msgPointer=0;
```

De mutex van de PID waardes word verkregen en het eerste character van receivedMsg word in een switch statement gezet. De goede waarde word geüpdate aan de hand van de “command character”. Na dit word de mutex vrijgelaten, de PIDValueUpdateFlag word hoog en de msgPointer word terug gezet naar 0.

Nu kan de PID cycle reageren op de flag en update de waardes. Onderstaande afbeelding laat zien wat er gebeurd als we in de serial A500 intypen:

```
pos: 102, TGT: 100, SPD: 2
pos: 102, TGT: 100, SPD: 2
pos: 102, TGT: 100, SPD: 2
pos: 101, TGT: 100, SPD: 1
pos: 102, TGT: 100, SPD: 2
pos: 105, TGT: 500, SPD: 120
pos: 322, TGT: 500, SPD: 120
pos: 439, TGT: 500, SPD: 84
pos: 512, TGT: 500, SPD: -14
pos: 501, TGT: 500, SPD: 1
pos: 501, TGT: 500, SPD: 1
pos: 501, TGT: 500, SPD: 1
```

Er is geen echo dus de gebruiker kan niet meer zien wat hij heeft ingetypt. Je kan zien dat na een tijdje de TGT (Target positie) verhoogt word van 100 naar 500, dit kan je niet doen met de encoder omdat deze stapjes van 10 maakt. De SPD (speed) verhoogt en de servo begint te bewegen. Aan het eind zie je dat de target overshoot met 12, krijgt de servo een speed van -14 die er voor zorgt dat de servo terug draait naar zijn target. De overshoot word gecorrigeerd door de PID system en de servo komt op zijn target daarmee is de regulatie een succes.

11.0 Eindproject feedback

Door de zenuwen was de code minder goed uitgelegd dan verwacht en waren er 3 kleine feedback momenten die ik moet verwerken voordat het afgerond was.

1. Het aanmaken van de mutex voor de update van de PID waardes werd gemaakt in de PID thread, dat zorgt voor gevaar omdat je niet weet welke threads eerst aan de beurt zijn en kan het dus zijn dat dat je de mutax al wil claimen terwijl hij nog niet aangemaakt is. Om dit op te lossen moet de mutex in de main loop aangemaakt worden voordat de threads aangemaakt worden

```
PIDA_value_mutex = osMutexNew(&MsgQueueMutexAttr);
osThreadNew(mainThread,  NULL, &defaultTask_attributes);
osThreadNew(outputThread,NULL, &defaultTask_attributes);
osThreadNew(inputThread, NULL, &defaultTask_attributes);
//osThreadNew(secondThread,NULL, &defaultTask_attributes);
osThreadNew(PIDThread, NULL, &defaultTask_attributes);
osThreadNew(UARTInputThread, NULL, &defaultTask_attributes);
```

2. Als je de mutex wilde claimen in de PID thread dan maakte ik gebruik van een osWaitforever en dat zorgt ervoor dat als hij de mutex wil claimen dat hij eeuwig wacht tot de mutex vrij is, dat kan er voor zorgen dat het systeem blijft hangen als de mutex niet beschikbaar is. Dit heb ik opgelost door gebruik te maken van een if die kijkt of de osMutexAcquire(PIDA_value_mutex, 0) als die gelijk is aan de return waarde van de functie osOk dan geeft hij aan dat de mutex vrij is en dat de thread de mutex mag claimen. Als de mutex niet vrij is dan gaat hij de waardes niet updaten en blijft hij niet hangen in het wachten tot hij de mutex mag claimen

```
if(PIDValueUpdateFlag){

 if(osMutexAcquire(PIDA_value_mutex,0)== osOk){

   Kp = Protected_Kp;
   Ki = Protected_Ki;
   Kd = Protected_Kd;
 osMutexRelease(PIDA_value_mutex);}

 PIDValueUpdateFlag=false;
}
```

3. Laatste feedback was dat de osDelay aan het einde van de thread op 1 ms staat, dat klopt niet met de data die in de datasheet staat van de servo, dit doordat de servo maar elke 20 ms update en niet elke 1 ms. Omdat we in onze simulatie wel gebruik hebben gemaakt van 20 ms moet de osDelay ook 20 ms worden.

```
osDelay(20);
```