Projecten van het werkveld

STM32 MCU met FreeRTOS & LwIP

Graduaat in Internet of Things

Gysen Bart

Academiejaar 2022-2023

Campus : Kempen

VOORWOORD

Met trots presenteer ik u dit verslag dat de resultaten en ervaringen weergeeft van mijn project. Ik vond het belangrijk om met een voor mij nieuw soort microcontroller te werken namelijk een STM32. Dit project bood me de gelegenheid om mijn kennis verder uit te breiden en mij voor te bereiden op het professionele leven.

Als student ben ik altijd gedreven geweest om nieuwe mogelijkheden te verkennen en uitdagingen buiten mijn comfortzone aan te gaan. Daarom koos ik ervoor om te werken met een microcontroller die ik nog niet eerder had gebruikt. Ik geloofde dat dit mijn begrip van embedded systemen en elektronica in het algemeen zou vergroten.

Gedurende dit project ben ik geconfronteerd met complexe problemen en onverwachte obstakels. Het kostte tijd en moeite om vertrouwd te raken met de microcontroller, zijn configuratie tool en drivers. Maar dankzij mijn vastberadenheid en doorzettingsvermogen kon ik deze obstakels overwinnen en mijn kennis enorm vergroten.

Dit project bood me niet alleen technische kennis, maar ook persoonlijke groei. Ik ontwikkelde mijn probleemoplossende vaardigheden, leerde beter organiseren, en verbeterde mijn vermogen om zelfstandig te werken. Deze vaardigheden zijn van onschatbare waarde in mijn voorbereiding op het professionele leven.

Ik wil graag van deze gelegenheid gebruik maken om mijn oprechte dank uit te spreken aan Peter Van Grieken, die mij enorm veel heeft bijgeleerd en ontzettend veel tijd en moeite in mijn project heeft gestoken. Zijn begeleiding en ondersteuning waren onmisbaar tijdens mijn ontdekkingsreis. Ik ben hem zeer dankbaar voor zijn geduld, expertise en toewijding.

INHOUDSOPGAVE

OVERZICHT FIGUREN

INLEIDING

Dit verslag beschrijft het project waarin we FreeRTOS en LwIP hebben geïntegreerd op een STM32-microcontrollerWe hebben FreeRTOS, een real-time besturingssysteem, en LwIP, een lichtgewicht TCP./IP-stack, geïntegreerd op een STM32F4 microcontroller. Dit stelt ons in staat om de verschillende componenten als taken te plannen en prioriteren, en biedt netwerkfunctionaliteit via TCP/IP-protocollen.

In dit verslag behandelen we de configuratie, implementatie van taken, intercommunicatie, netwerkinterface-configuratie en functionaliteitstests. We bespreken ook obstakels en oplossingen die we tijdens het ontwikkelproces zijn tegengekomen.

De integratie van FreeRTOS en LwIP biedt mogelijkheden voor geavanceerde embedded systemen met netwerkfunctionaliteit. Dit verslag geeft een overzicht van het project, documenteert de stappen en overwegingen die tot een succesvolle integratie hebben geleid.

1 HET PROJECT

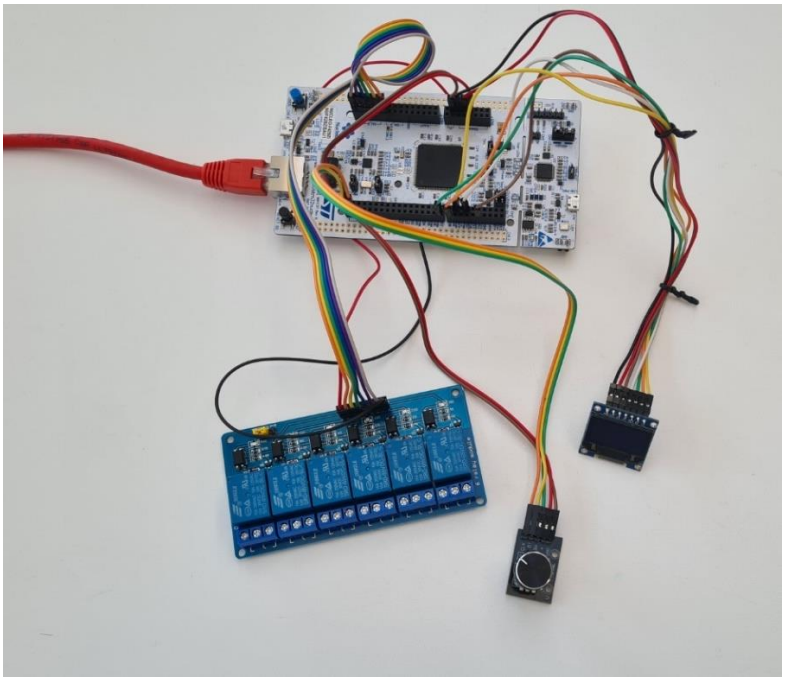
In het project is gebruik gemaakt van de devkit NUCLEO-F429ZI, FreeRTOS, LwIP, een MQTT-broker in Home Assistant, de Home Assistant app, een klein OLED-display, een rotary encoder en GPIO’s met externe relais.

In dit project lag de focus meer op de achterliggende software dan op het “visuele en tastbare”. Uiteindelijk kan het de status van de rotary encoder weergeven op het OLED-display, deze informatie versturen naar de MQTT-broker en deze weergeven in Home Assistant. Verder kan het nog met behulp van Home Assistant ook externe relais schakelen. Deze relais kunnen bijvoorbeeld worden gebruikt om verlichting, apparaten of andere elektrische componenten in en uit te schakelen. Het aansturen van de relais via de Home Assistant-app biedt de mogelijkheid om op afstand apparaten te bedienen en automatiseringsscenario's te creëren.

De project doelstellingen zijn:

- Gebruik maken van een 32-bit microcontroller met ethernet
- Uitlezen van een rotary encoder
- Aansturen van een OLED-display
- Gebruik maken van FreeRTOS
- Gebruik maken van Lwip & MQTT
- Aansturen van externe relais via GPIO
- Aansturen van de relais via Home Assistant

De meeste doelstellingen van dit project heb ik eerst afzonderlijk onderzocht, gemaakt en uitgebreid getest. Ik ga in dit verslag ook alles per ‘deelproject’ toelichten. Aan het einde van het project is alles samengebracht en gefinetuned waar nodig. Het totaal plaatje kan je in onderstaande afbeelding terug vinden.



Figuur 1 Project

2 HARDWARE COMPONENTEN

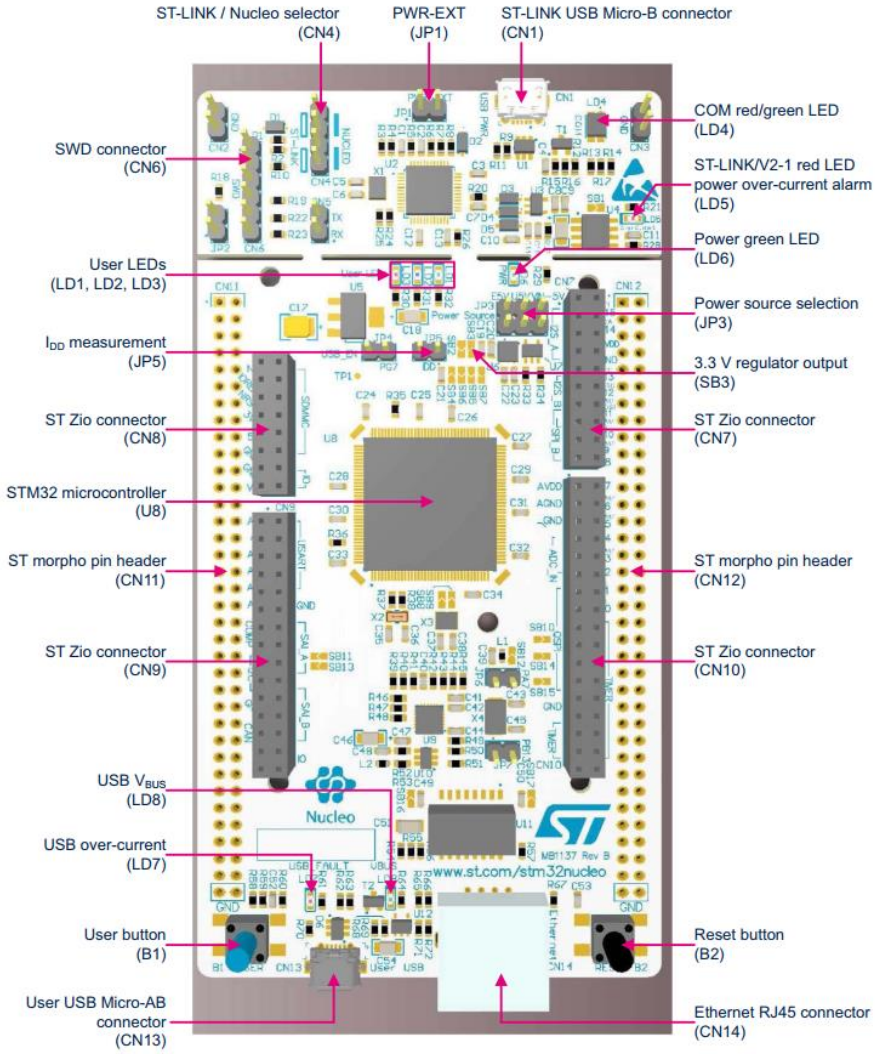
In dit hoofdstuk zal ik de verschillende hardware componenten van het project toelichten. Deze componenten vormen de bouwstenen van het project. Door een dieper inzicht te krijgen in hun functionaliteit, zult u een beter begrip krijgen van het project als geheel.

2.1 NUCLEO-F429ZI

Figuur 2 Nucleo-F429ZI

In het project maken we gebruik van de Nucleo-F429ZI. Dat is een STM32 Nucleo-144 Development Board dat gebruik maakt van de STM32F429ZI microcontroller dat is ontworpen en geproduceerd door STMicroelectronics. Het maakt deel uit van hun Nucleo-familie van development boards, die bedoeld zijn om ontwikkelaars, studenten en hobbyisten een eenvoudige en betaalbare manier te bieden om projecten te prototypen en te evalueren. Het beschikt over ST Zio headers wat het mogelijk maakt om arduino shields te gebruiken, ook beschikt het over de ST Morpho headers, die maken het mogelijk om alle pinnen te gebruiken. Het bord beschikt over de volgende functies:

- Interne ST-LINK/V2-1 debugger
- Flexibele bordvoeding 5V van ST-LINK/V2-1 USB VBUS
- Externe voedingsbronnen 5V en 7 tot 12V op ST Zio of ST morpho connectoren
- USB OTG of full-speed apparaat met micro-AB connector
- IEEE-802.3-2002-compatibele Ethernet-connector
- Drie user-LED's
- Twee drukknoppen - USER en RESET
- 32.768kHz Kristaloscillator - LSE kristal



Figuur 3 Nucleo-F429ZI overview

2.1.1 STM32F429ZI

De STM32F429ZI is een microcontroller unit (MCU) uit de STM32F4-serie van STMicroelectronics. Het is gebaseerd op de ARM Cortex-M4 core en biedt verschillende functies die nuttig zijn voor embedded toepassingen. Hier zijn enkele belangrijke kenmerken:

- **Hoge prestaties**: De STM32F429ZI werkt op een hoge 180Mhz frequentie, wat zorgt voor krachtige verwerking.
- **Flashgeheugen**: Er is tot 2MB flashgeheugen aanwezig. Hierdoor is er voldoende opslagruimte aanwezig voor code en gegevens.

- **RAM**: De controller is uitgerust met maximaal 256 KB aan RAM, wat ruimte biedt voor variabelen, stack en opslag van runtimegegevens.

- **Peripherals**: De controller biedt een uitgebreide set aan peripherals, waaronder meerdere USART, SPI, I2C en CAN interfaces, USB OTG, Ethernet MAC en SDIO interfaces.

- **Analoge functies**: De STM32F429ZI bevat een 12-bits analog-to-digital coverter (ADC) met maximaal 24 kanalen. Het bevat ook een digital-to-analog

converter (DAC) met maximaal 2 kanalen voor het genereren van bijvoorbeeld audio signalen.

- **Timers**: De controller bevat tot twaalf 16-bit waaronder twee moter-control PWM, verder zijn er nog twee 32-bit en twee watchdogtimers.

- **LCD-controller**: Er is een geïntegreerde LCD-controller aanwezig waardoor het geschikt is voor toepassingen die grafische gebruikersinterfaces vereisen.

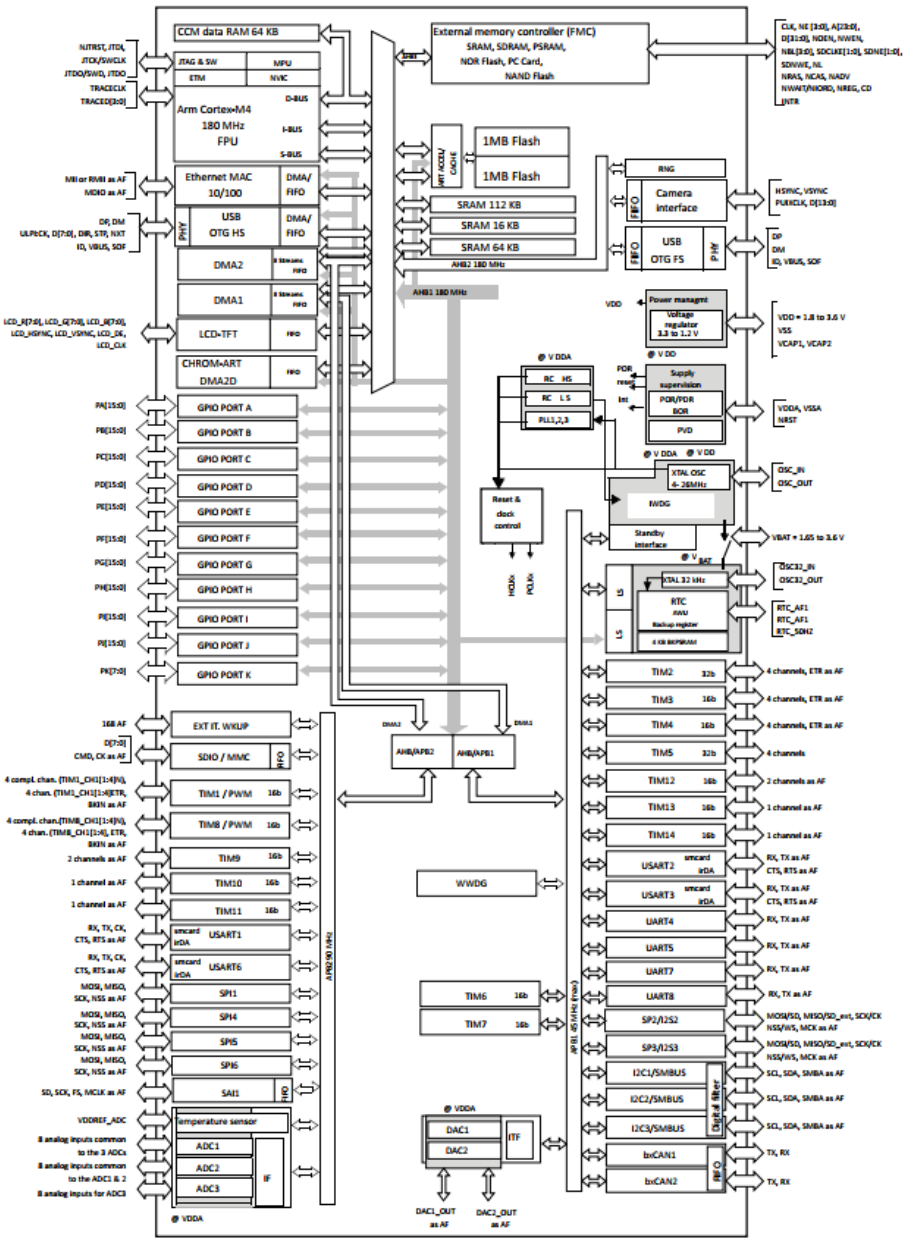
- **Snelle connectiviteit**: De MCU ondersteunt snelle communicatie via USB 2.0 Full-Speed en High-Speed (met On-The-Go ondersteuning), Ethernet MAC en verschillende seriële communicatie-interfaces.

- **Digital Signal Processing (DSP) functies**: De Cortex-M4 kern bevat een hardware floating-point unit (FPU) en DSP-instructies, waardoor efficiënte signaalverwerking en wiskundige bewerkingen mogelijk zijn.

- **Ondersteuning voor Real-Time Operating System (RTOS)**: De MCU is compatibel met verschillende RTOS-oplossingen, zoals FreeRTOS, waardoor ontwikkelaars complexe en multitasking-toepassingen kunnen bouwen.

- **Ontwikkelingsondersteuning**: STMicroelectronics biedt een uitgebreid ondersteuning, inclusief ontwikkelingsboards, software-librarys en uitgebreide documentatie.

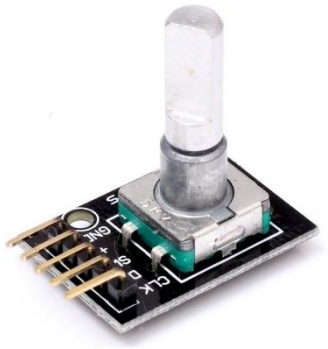
Deze kenmerken maken de STM32F429ZI een krachtige en veelzijdige microcontroller die geschikt is voor veel toepassingen, waaronder industriële regelsystemen, consumentenelektronica, medische apparaten en Internet of Things (IoT) toepassingen.



Figuur 4 Blokschema STM32F429ZI

2.2 Rotary encoder

Een rotary encoder is een elektronisch toestel met een roterende as, dat pulsen genereert als de as verdraait. Er bestaan ook versies die pulsen genereren langs een translatie-as.



Figuur 5 incrementele encoder

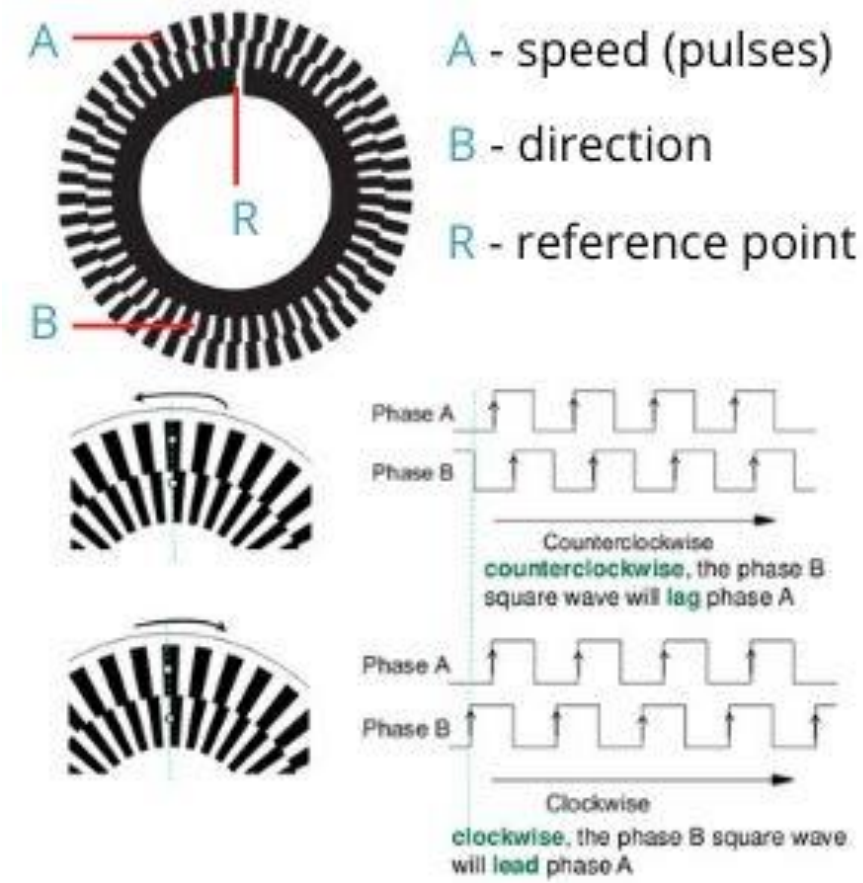
Er zijn in principe twee soorten encoders:

- incrementele encoders
- absoluut encoders

In dit project gaan we enkel gebruik maken van een incrementele encoder.

Een incrementele encoder geeft een simpele puls trein. De pulsen worden geteld en omgezet naar een rotatiehoek (of een translatie-afstand). Door twee puls treinen te gebruiken (A, B) die 90 graden in fase zijn verschoven, kan de elektronica van de encoder ook de richting van de beweging afleiden uit de puls treinen. Daarnaast is er vaak nog een derde signaal dat een enkele puls geeft op 0 graden (Z of C). In veel encoders is van deze signalen ook de inverse aanwezig (A, /A, B, /B, Z, /Z).

In onderstaande afbeelding is te zien dat wanneer we tegen de klok draaien Phase A hoog wordt terwijl Phase B laag blijft. Als we met de klok meedraaien is het andersom, hier gaat eerst Phase B hoog worden en daarna pas Phase A. Door deze signalen uit te lezen in de software kunnen we de richting bepalen.



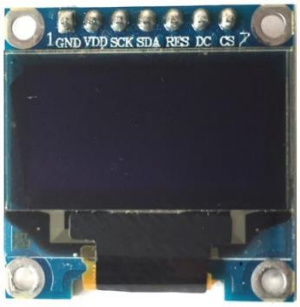
Figuur 6 Rotary encoder mechanisch voorbeeld

2.3 OLED Display

OLED is Organic Light Emitting Diode die licht uitzendt als reactie op een elektrische stroom. Een OLED-scherm werkt zonder achtergrondverlichting, zodat het diepe zwartniveaus kan weergeven. Het is klein van formaat en licht in gewicht dan Liquid Crystal Displays.

Een 128x64 OLED-display is een eenvoudig dot matrix grafisch display. Het heeft 128 kolommen en 64 rijen waardoor het in totaal 128x64 = 8192 pixels weergeeft. Door deze pixel's led aan / uit te zetten, kunnen we een grafisch beeld van elke vorm erop weergeven. Het OLED-display wordt aangedreven door een SSD1306 driver IC. SSD1306 is een CMOS OLED-driver met controller voor een dot-matrix OLED-display. Door het gebruik van de SSD1306-driver is het aantal benodigde externe componenten en het stroomverbruik verminderd.

Het OLED-display dat in dit project gebruikt wordt werkt zowel via I2C als SPI.



Figuur 7 OLED display

2.4 Relais module

Omdat veel elektrische systemen meer vermogen nodig hebben dan een microcontroller kan leveren, kan de relaismodule in veel verschillende toepassingen worden gebruikt. Relaismodules zijn eenvoudigweg printplaten die een of meer relais bevatten. Ze zijn er in verschillende vormen en maten, maar zijn meestal rechthoekig met 2, 4, 6 of 8 relais erop gemonteerd, soms zelfs tot wel 16 relais. Relaismodules bevatten ook andere componenten dan de relais zelf. Waaronder indicatie-LED's, beschermingsdiodes, transistors, weerstanden en andere onderdelen.

Figuur 8 Relais module

3 SOFTWARE COMPONENTEN

In dit hoofdstuk zal ik de verschillende software componenten die gebruikt zijn in het project toelichten. Deze componenten vormen de bouwstenen van de software in het project.

3.1 STM HAL drivers

De HAL-drivers zijn ontworpen om een uitgebreide set API's te bieden en gemakkelijk te kunnen communiceren met de bovenste applicatielagen.

Elke driver bestaat uit een reeks functies die de meest voorkomende peripherals dekken. De ontwikkeling van elke driver wordt gestuurd door een gemeenschappelijke API die de driverstructuur, de functies en de parameterbenamingen standaardiseert.

De HAL-drivers omvatten een set drivermodules, waarbij elke module is gekoppeld aan een zelfstandige peripheral. In sommige gevallen kan de module echter gekoppeld zijn aan een functionele mode van een peripheral. Als voorbeeld zijn er meerdere modules beschikbaar voor de USART peripheral: de UART-drivermodule, de USART-drivermodule, de SMARTCARD-drivermodule en de IRDA-drivermodule.

De belangrijkste kenmerken van HAL zijn:

- Een draagbare set API's die geschikt is voor verschillende microcontrollers en die zowel gemeenschappelijke peripherals als uitbreidings-API's voor specifieke peripherals dekken.
- Drie programmeermodellen voor API's: polling, interrupt en DMA.
- De API's zijn compatibel met RTOS:
    - Volledig reintegratie van API's.
    - Systematisch gebruik van time-outs in polling-modus.
- Ondersteuning van meerdere instanties van peripherals, waardoor gelijktijdige API-aanroepen mogelijk zijn voor meerdere instanties van een bepaalde peripheral (bijvoorbeeld USART1, USART2...).
- Alle HAL-API's implementeren het mechanisme van user-callbackfuncties:
    - De HAL Init/DeInit-functies kunnen user-callbackfuncties aanroepen om peripheral initialisatie/de-initialisatie (klok, GPIO's, interrupt, DMA) uit te voeren.
    - Interrupts.
    - Error events.
- Objectvergrendelingsmechanisme: veilige toegang tot hardware om meerdere onbedoelde toegangen tot gedeelde bronnen te voorkomen.
- Time-out wordt gebruikt voor alle blokkerende processen: de time-out kan een eenvoudige teller zijn of een tijdbasis.

3.2 CubeMX

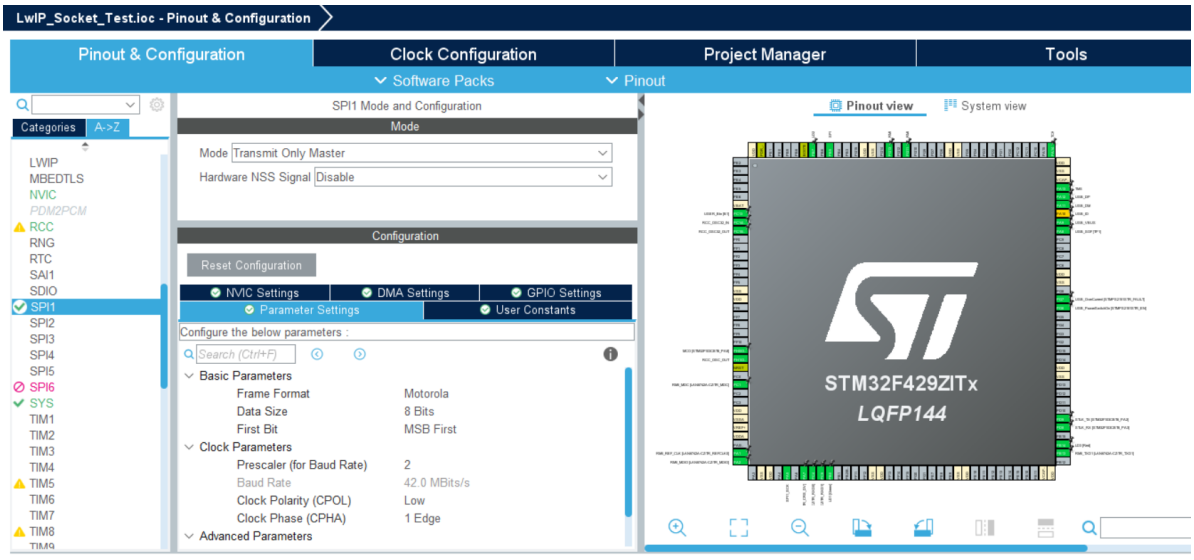
STM32CubeMX is een grafische tool die een zeer eenvoudige configuratie van STM32-microcontrollers en microprocessors mogelijk maakt, evenals het genereren van de bijbehorende initialisatie C-code voor de Arm Cortex-M-kern of een gedeeltelijke Linux-apparaatboom voor Arm Cortex-A-kern, via een stapsgewijs proces.

De eerste stap bestaat uit het selecteren van een STMicrolectronics STM32-microcontroller, microprocessor of een ontwikkelingsplatform dat overeenkomt met de vereiste set randapparatuur, of een voorbeeld dat op een specifiek ontwikkelingsplatform wordt uitgevoerd.

Voor microprocessors maakt de tweede stap het mogelijk om de GPIO's en de klokopstelling voor het hele systeem te configureren en om randapparatuur interactief toe te wijzen aan de Arm Cortex-M of aan de Cortex-A-wereld. Specifieke hulpprogramma's, zoals DDR-configuratie en -afstemming, maken het gemakkelijk om aan de slag te gaan met STM32-microprocessors. Voor Cortex-M core bevat de configuratie extra stappen die precies vergelijkbaar zijn met die beschreven voor microcontrollers.

Voor microcontrollers en microprocessor Arm Cortex-M bestaat de tweede stap uit het configureren van elke vereiste embedded software dankzij een pinout-conflictoplosser, een helper voor het instellen van een klokstructuur, een energieverbruikscalculator en een hulpprogramma dat de randapparatuur (zoals GPIO of USART) en de middleware-stacks (zoals USB of TCP / IP) configureert.

De standaard software en middleware stacks kunnen worden uitgebreid dankzij verbeterde STM32Cube uitbreidingspakketten. De partnerpakketten van STMicrolectronics of STMicrolectronics kunnen rechtstreeks worden gedownload van een speciale pakketbeheerder die beschikbaar is binnen STM32CubeMX, terwijl de andere pakketten vanaf een lokale schijf kunnen worden geïnstalleerd.



Figuur 9 Voorbeeld CubeMX

3.3 FreeRTOS

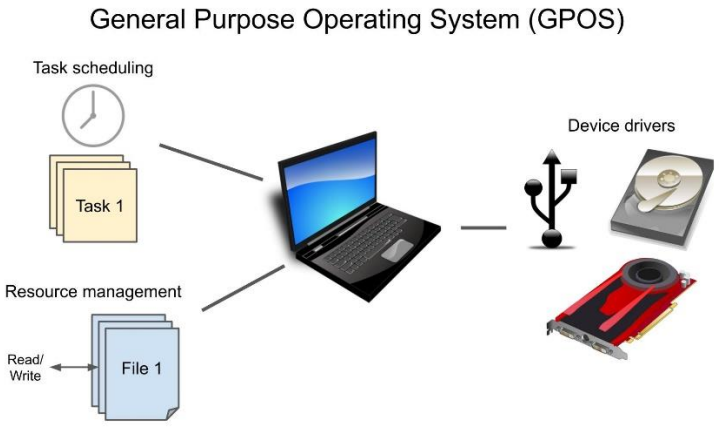
Een real-time besturingssysteem (RTOS) is meestal een lichtgewicht besturingssysteem dat multi-threaded applicaties draait en real-time deadlines kan halen. Met deadlines duiden we een niveau van determinisme aan wat betekent dat we kunnen uitzoeken wanneer bepaalde taken worden uitgevoerd voorafgaand aan de runtime. De meeste RTOS'en bevatten een scheduler, resourcemanagement en device drivers. In dit project wordt er gebruik gemaakt van FreeRTOS. FreeRTOS is lang niet het enige RTOS systeem, zo bestaan er nog:

- VxWorks
- QNX
- RTLinux
- embOS
- zephyrOS

3.3.1 GPOS vs RTOS

General Purpose Operating System (GPOS)

Vaak, als we denken aan "besturingssystemen", denken we aan dingen als Windows, macOS en Linux. Dit zijn voorbeelden van besturingssystemen voor algemeen gebruik (GPOS). Meestal zijn deze besturingssystemen ontworpen voor gebruikersinteractie en bieden ze een soort interface, of dat nu een opdrachtregelinterface (CLI) of een grafische gebruikersinterface (GUI) is. Ze zijn ook ontworpen om meerdere toepassingen uit te voeren, vaak met multithreading, en bieden andere voordelen, zoals resource- / bestandsbeheer en drivers. Omdat gebruikersinteractie meestal de belangrijkste focus is voor GPOS-ontwerp, is enige latentie acceptabel (zolang gebruikers geen lange vertragingstijd opmerken). Als gevolg hiervan zijn exacte taakdeadlines moeilijk (zo niet onmogelijk) te voorspellen in een GPOS.



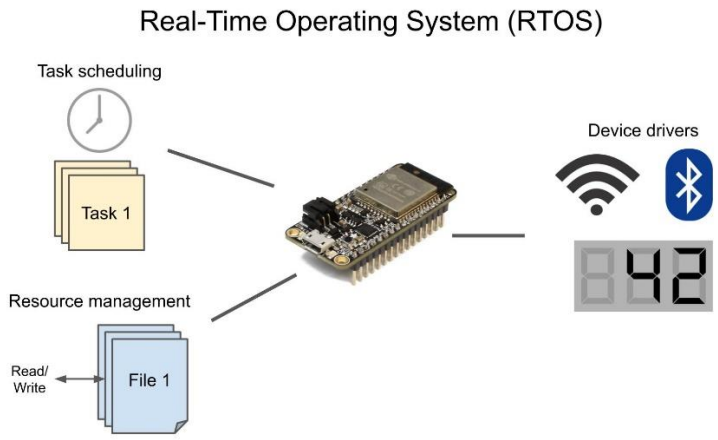
Figuur 10 General Purpose Operating System

Real-Time Operating System (RTOS)

De meeste RTOS'en zijn ontworpen om op microcontrollers te werken. Als gevolg hiervan kunnen ze vaak afzien van complexe gebruikersinterfaces om een paar taken

tegelijk uit te voeren zonder gebruikersinvoer te hoeven verwerken. Bovendien kunnen toepassingen die zijn geschreven voor microcontrollers strikte timingdeadlines hebben die moeten worden gehaald, zoals het afvuren van een bougie om de zoveel milliseconden, het besturen van een medisch apparaat om iemand in leven te houden en het afvuren van stuwraketten op een satelliet op een precies tijdstip om in een baan om de aarde te blijven.

Veel RTOS'en worden ook geleverd met resource management libraries, zoals de mogelijkheid om bestanden te lezen en ernaar te schrijven, evenals low-level drivers, zoals Wifi- en Bluetooth-stacks en LCD-drivers.

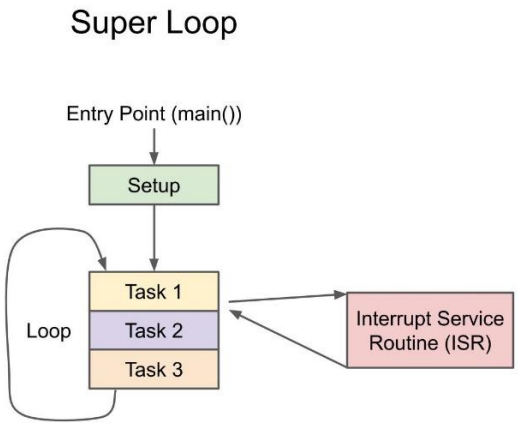


Figuur 11 Real-Time Operating System

3.3.2 Bare-Metal vs RTOS

Bare-Metal

In een microcontroller maken we gebruik van aparte taken zoals bijvoorbeeld het aansturen van een led, een statemachine, uitlezen van een buffer, etc. Vooral in kleinere en meer beperkte microcontrollers wordt dit geprogrammeerd in Bare-Metal. Hierbij wordt er geen gebruik gemaakt van een operating system maar wordt alles achter elkaar, elke keer opnieuw uitgevoerd (Super-Loop).

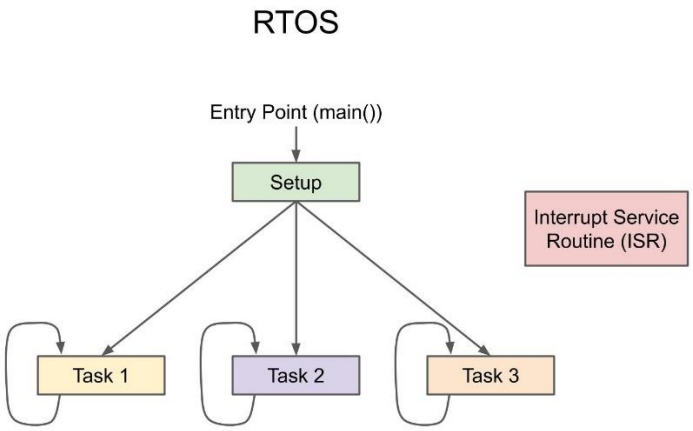


Figuur 12 Super Loop

In feite is Bare-Metal nog steeds een van de meest populaire manieren om een microcontroller te programmeren, omdat het gemakkelijk te implementeren en gemakkelijk te debuggen is. Er kan zelfs gebruik worden gemaakt van interrupts, die zorgen ervoor dat het programma stopt en willekeurige code uitvoert wanneer een externe gebeurtenis optreedt (zoals een timer die afloopt of een knop die wordt ingedrukt).

RTOS

Het probleem gaat zich voordoen wanneer er zoveel taken wordt toegevoegd dat sommige deadlines beginnen te missen of voorkomen dat andere functies werken. Dit is waar een RTOS kan helpen. In plaats van alles achter elkaar uit te voeren, kun je in alles "tegelijkertijd" uitvoeren. RTOS zorgt ervoor dat er verschillende taken naast elkaar kunnen draaien zodat de taken elkaar niet kunnen ophouden. Er kan nog steeds gebruik gemaakt worden van interrupts. De werking is vergelijkbaar met die van de Bare-Metal manier, de taken worden gestopt, interrupt wordt uitgevoerd en de taken gaan weer verder.



Figuur 13 RTOS Task loops

3.4 LwIP

Het verbinden van embedded microcontrollers met het internet is een cruciale taak in moderne toepassingen. De ingebouwde controllers, met name in consumentenapparaten en wearables, vereisen tegenwoordig een internet verbinding. Dit is belangrijk omdat de apparaten slimmer worden. Verbinding maken met internet is niet alleen cruciaal, maar ook een flinke taak. De typische TCP/IP-stack is te resource-intensief voor bare-metals om te implementeren. Microcontrollers kunnen nooit TCP / IP-stack implementeren vanwege weinig geheugen, RAM en rekenkracht, dus ze hebben iets lichtgewichts nodig om dezelfde functionaliteit te emuleren.

Light-Weight Internet Protocol (LwIP) is een kleine onafhankelijke implementatie van de TCP/IP-stack voor embedded systemen. Oorspronkelijk ontwikkeld door Adam Dunkels aan het Swedish Institute of Computer Science (SICS), is LwIP een gratis,

open-source TCP/IP-stack die nu bij Savannah wordt onderhouden. Verschillende gerenommeerde fabrikanten van embedded systemen hebben LwIP geadopteerd, zoals Texas Instruments, Altera Corporation, Xilinx, STMicroelectronics, Freescale en Analog Devices, voor netwerkdrivers van hun embedded apparaten. Het protocol wordt zelfs gebruikt door Arduino Ethernet Shield en WiFi-ontwikkelborden zoals ESP32 en ESP8266 voor het implementeren van netwerkfuncties. Elke hardware fabrikant kan het protocol gebruiken met het besturingssysteem of met de driver op zijn platform als een open protocol onder de gewijzigde BSD-licentie. Het protocol kan ook worden gebruikt in toepassingen op voorwaarde dat het onderliggende systeem de LwIP-poort ondersteunt.

Met LwIP kan een volwaardige TCP/IP-stack op embedded systemen worden geïmplementeerd terwijl er minimale systeembronnen en geheugen worden verbruikt. De stack is zelfs aangepast aan Arduino-libraries. De stack vereist slechts 40kb flash-geheugen en verbruikt slechts tientallen Kbytes RAM tijdens runtime. Zelfs een op maat gemaakte API is beschikbaar voor de stack die geen gegevens hoeft te kopiëren.

LwIP biedt de basisfunctionaliteit voor het opzetten van netwerkverbindingen, het verzenden en ontvangen van gegevens via IP-netwerken en het implementeren van verschillende netwerkprotocollen zoals:

- Internet Protocol (IP)
- Internet Control Message Protocol (ICMP)
- Neighbor Discovery (ND)
- Multicast listener discovery for IPv6 (MLD)
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
- Internet Group Management Protocol (IGMP)
- Address Resolution Protocol (ARP)
- Point-to-Point Protocol (PPP)

Verder beschikt het ook nog over drie API’s:

- low-level "core" / "callback" of "raw" API
- netconn API
- socket API

3.5 MQTT

MQTT protocol (Message Queuing Telemetry Transport) is een gemeenschappelijke taal waardoor sensoren, actuatoren en machines met elkaar kunnen communiceren. Het is een lichtgewicht publish en subscribe systeem. Het protocol is ontwikkeld als eenvoudig systeem om met een lage bandbreedte data over te dragen. Dankzij deze eigenschappen is MQTT erg geschikt om te gebruiken binnen Internet of Things applicaties. MQTT wordt gebruikt in verschillende industrieën, zoals de auto-industrie, productie, telecommunicatie, olie en gas. MQTT maakt gebruik van de volgende basis begrippen: Publish/Subscribe, Messages, Topics en Broker.

De broker is de centrale spil voor alle verzonden berichten. De betrokken apparaten communiceren alleen met de broker en kennen elkaar onderling verder niet. Ze hoeven de ip-adressen en de technische details van de andere deelnemers dus niet te weten. Het is de taak van de broker om berichten te accepteren en aan de juiste ontvanger door te geven.

3.6 Home Assistant

Home Assistant is een open source platform voor centrale aansturing slimme apparaten en domotica, met focus op vrijheid en privacy. De software kan zowel als losse software, binnen een container of met een bijbehorend besturingssysteem gebruikt worden. Home Assistant is vervolgens toegankelijk via een web interface en mobiele applicaties voor Android en iOS. Het platform kan onafhankelijk van een internetverbinding of externe diensten werken en wordt daarom vaak aangewezen als een privacy vriendelijk alternatief voor andere domoticaplatformen.

Home Assistant kan via een breed scala aan integrations, gemaakt door vrijwilligers, communiceren met gangbare domoticaplatformen van onder andere Google, IKEA, Amazon, Apple (HomeKit), Signify (Philips Hue), Tuya en meer. Home Assistant fungeert dan ook als een "hub", van waaruit de gebruiker slimme logica, dashboards en automatisering kan programmeren zonder afhankelijk te zijn van verschillende applicaties van verschillende leveranciers van domotica hardware. Home Assistant biedt verder ondersteuning voor verschillende draadloze communicatieprotocollen zoals Bluetooth, Z-Wave en ZigBee. Verder kunnen slimme meters en controlesystemen van bijvoorbeeld zonnepanelen geïntegreerd worden om zo de gebruiker een live overzicht van energieverbruik te bieden, en op basis van verandering in deze data geautomatiseerd actie te ondernemen.

Home Assistant wordt openbaar ontwikkeld en bijdragen komen van vrijwilligers. Tussen september 2013 en juli 2022 hebben in totaal meer dan 2880 personen bijgedragen aan het open source project dat geleid wordt door Paulus Schoutsen. Hiervan werkt een subgroep van 20 tot 30 mensen intensief mee aan de doorontwikkeling van Home Assistant.

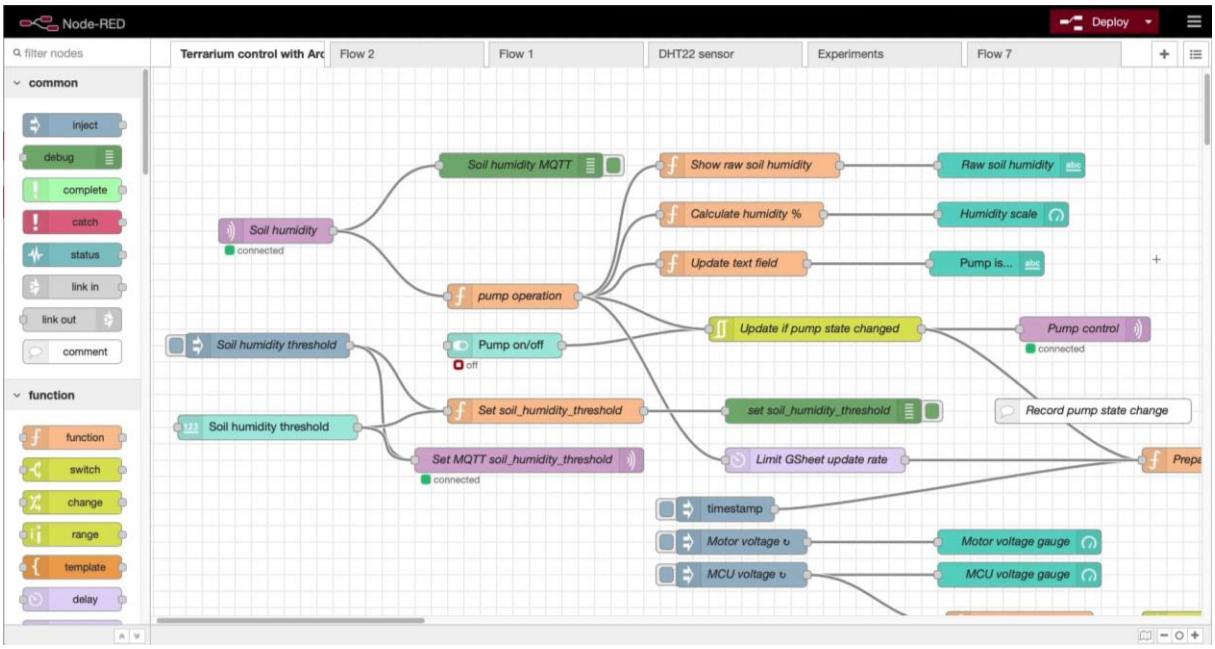
Home Assistant wordt gebruikt om de externe relais aan te sturen, ook wordt in het dashboard de stand van de rotary encoder weergegeven.

3.7 Node-Red

Node-RED is een open-source visuele programmeertool waarmee hardware-apparaten, APIs en online diensten aan elkaar gekoppeld kunnen worden. Het stelt gebruikers in staat om stromen te maken, of reeksen van acties, met behulp van een drag-n-drop-interface. Deze stromen kunnen worden gebruikt om apparaten aan te sturen, gegevens op te halen en te verwerken, en berichten en meldingen te versturen. Node-RED is gebouwd op Node.js en wordt vaak gebruikt in Internet of Things (IoT)- en domotica-projecten.

Node-RED gebruiken ik om de data uit de mqtt broker te halen en te sturen

De afbeelding hieronder illustreert een voorbeeld van een node-RED programmatie.



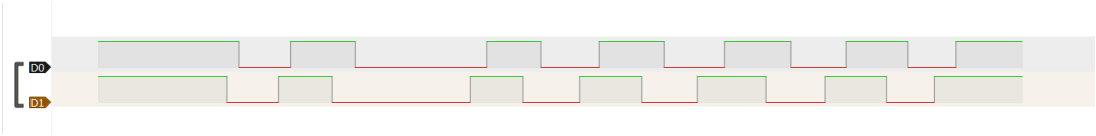
Figuur 14 Voorbeeld Node-Red

4 DEELPROJECTEN

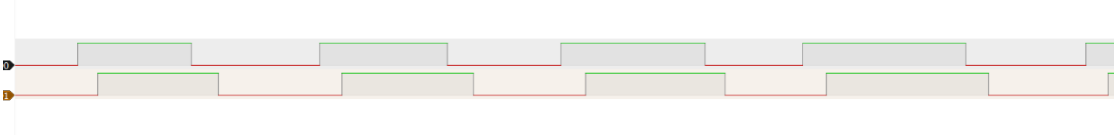
Omdat dit project een beetje complex ging worden heb ik gebruik gemaakt van deelprojecten. Dit was handig omdat ik dan het overzicht kon houden over de verschillende onderdelen van het project. Ook als er iets zou mislopen zijn de andere onderdelen nog een beetje beschermd. Ook geeft dit als voordeel dat de onderdelen gemakkelijk worden aangepast aan de noden van de andere deelprojecten.

4.1 Rotary encoder

Zoals in 2.2 Rotary encoder al is uitgelegd, geeft de encoder twee signalen die we moeten uitlezen. Voor de zekerheid heb ik dit nog gecontroleerd met een logic analyzer.



Figuur 15 Signaal rotary links



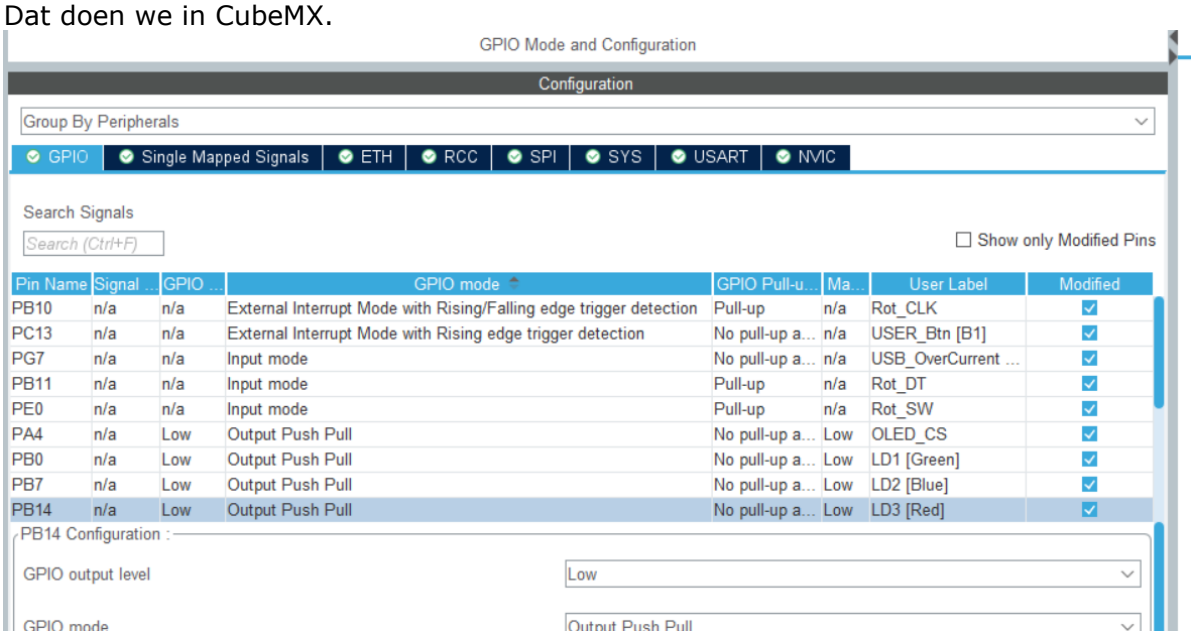
Figuur 16 Signaal rotary rechts

Zoals we in bovenstaande afbeeldingen kunnen zien klopt het inderdaad dat de signalen 90° verschillen. De rotary encoder dat ik gebruik hebben de pinnen DT (data) en CLK (clock). Het eerste signaal is CLK, het tweede DT. Als ik naar links draai ijlt DT voor op CLK, naar rechts ijlt DT na op CLK.

Dit gaat allemaal zeer snel, om op die snelheid te reageren ga ik gebruik maken van interrupts. Voor het wat duidelijk te houden ga ik een interrupt laten triggeren wanneer er een verandering van signaal is op CLK, daarna ga ik de stand van DT controleren om de richting te bepalen. Als we de richting hebben bepaald gaan we aan de hand van de richting een variabele optellen of aftellen.

4.1.1 Configuratie

Voor dat er code voor kan geschreven worden moeten we eerst de inputs configureren. Dat doen we in CubeMX.



Figuur 17 Rotary GPIOs CubeMX

De volgende pinnen heb ik geconfigureerd:

- **PB10 -> Rot_CLK**: Dit heb ik ingesteld als ext interrupt op rising en falling edge, dit zorgt ervoor dat bij elke verandering de interrupt wordt getriggerd.
- **PB11 -> Rot_DT**: Dit is ingesteld als een gewone input.
- **PE0 -> Rot_SW**: Ook dit is ingesteld als gewone input.

4.1.2 **Test**

Voor dit te testen heb ik code geschreven aan de hand van de uit te lezen signalen dat ik eerder al heb uitgelegd. We gaan de richting bepalen door eerst het CLK signaal te lezen omdat daarop wordt getriggerd, dit doen we door Rot_CLKstate. Eens dat gebeurd is gaan we aan de hand van die waarde het DT signaal uitlezen (Rot_DTstate), hier gaan we de richting in bepalen.

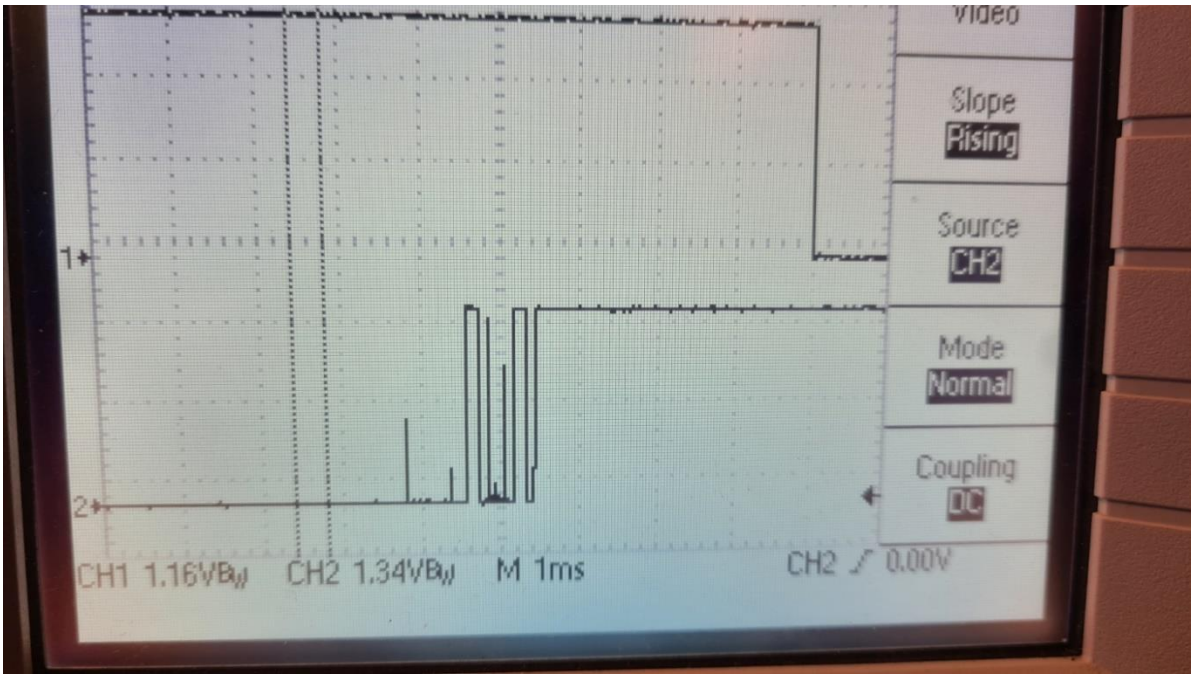
```
/**
  * @brief This function handles TIM3 global interrupt.
  */
void TIM3_IRQHandler(void)
{
  /* USER CODE BEGIN TIM3_IRQn 0 */
    // start routine only when there is an timer overflow

    if(IRQ_Triggerd)
    {
        HAL_GPIO_TogglePin(LD1_GPIO_Port, LD1_Pin);

        Rot_CLKstate = HAL_GPIO_ReadPin(Rot_CLK_GPIO_Port, Rot_CLK_Pin);
        Rot_DTstate = HAL_GPIO_ReadPin(Rot_DT_GPIO_Port, Rot_DT_Pin);

        if (Rot_CLKstate)
        {
            if(Rot_DTstate)
            {
                Rot_cnt--;
            }
            else
            {
                Rot_cnt++;
            }
        }
        else
        {
            if(Rot_DTstate)
            {
                Rot_cnt++;
            }
            else
            {
                Rot_cnt--;
            }
        }

        ROT_IRQ_Flag = 1;

        IRQ_Triggerd = 0;
    }
  /* USER CODE END TIM3_IRQn 0 */
  HAL_TIM_IRQHandler(&htim3);
  /* USER CODE BEGIN TIM3_IRQn 1 */
  HAL_TIM_Base_Stop_IT(&htim3);
  /* USER CODE END TIM3_IRQn 1 */
}
```

Figuur 18 Uitlezen rotary interrupt

4.1.3 **Denderen**

Bij de test heb ik ondervonden dat de variabele Rot_cnt inorm schommelde in waarde’s. Door dit te debuggen viel het op dat de interrupt meermaals werd getriggerd door één positie te veranderen.

De rotary encoder geeft of schakelt een mechanisch contact. Wordt er contact gemaakt, dan wordt in werkelijkheid het contact binnen een paar microseconden een aantal malen gemaakt en verbroken. Dit gaf de problemen in de interrupt, elke keer als het signaal veranderde werd de interrupt getriggerd. Dit wordt schakel dender of bouncing genoemd. Met de oscilloscope werd dit ook duidelijk.



Figuur 19 Schakeldender scope

Om dit probleem op te lossen moeten we ervoor zorgen dat als de interrupt wordt getriggerd er een overbrugging is totdat de dender is verdwenen. Dit heb ik gedaan door gebruik te maken van timer in output compare mode. De timer zal geactiveerd worden wanneer er een interrupt plaatsvindt en zijn teller zal blijven tellen tot deze gelijk is aan de ingestelde counter period. Wat er dan voor zorgt dat de timer een interrupt triggerd en daar worden dan de signalen uitgelezen, erna wordt de timer gestopt zodat deze niet opnieuw de interrupt doet triggeren. Als er in de "overbrugging" periode een nieuwe interrupt wordt getriggerd zal de counter value van de timer worden gerest. Dit zorgt ervoor dat ik vanaf nu altijd correcte waardes heb.

```
231 /**
232   * @brief This function handles EXTI line[15:10] interrupts.
233   */
234 void EXTI15_10_IRQHandler(void)
235 {
236   /* USER CODE BEGIN EXTI15_10_IRQn 0 */
237     if(!IRQ_Triggerd)
238     {
239         IRQ_Triggerd = 1;
240         HAL_TIM_Base_Start_IT(&htim3);
241     }
242
243     else
244     {
245         // set timer3 count value to zero
246         TIM3->CNT = 0;
247
248     }
249   /* USER CODE END EXTI15_10_IRQn 0 */
250   HAL_GPIO_EXTI_IRQHandler(Rot_CLK_Pin);
251   HAL_GPIO_EXTI_IRQHandler(USER_Btn_Pin);
252   /* USER CODE BEGIN EXTI15_10_IRQn 1 */
253
254   /* USER CODE END EXTI15_10_IRQn 1 */
255 }
```

Figuur 20 External interrupt Rotary

4.2 OLED display

Zoals ik in 2.3 OLED-display al had aangehaald kunnen we dit scherm op twee manieren gebruiken, I2C en SPI. Ik heb voor SPI gekozen. Ik ben ook opzoek gegaan naar een geschikte library voor de ssd1306 driver om deze te kunnen implementeren om het scherm te sturen.

4.2.1 SSD1306 driver library

De library heb ik gevonden op github en kan via de volgende link gedownload worden. https://github.com/afiskon/stm32-ssd1306

4.2.1.1 Ssd1306_conf_template.h

De driver moet eerst geconfigureerd worden in de file ssd1306_conf_template.h na de configuratie dient _template te worden verwijderd. De driver kan gebruikt worden met zowel SPI en I2C. Welke we gaan gebruiken dient geselecteerd te worden. De driver is ook getest op de volgende microcontrollers:

- STM32F0
- STM32F1
- STM32F3
- STM32F4
- STM32F7
- STM32L0
- STM32L1
- STM32L4
- STM32H7
- STM32G0

Hier geldt hetzelfde, wat we gaan gebruiken moeten we selecteren. Het volgende dat dient te gebeuren is het instellen van de SPI port en de bijhorende GPIO's. In de template staan een aantal define's i.v.m. SPI configuratie, wat hier dient te gebeuren zijn de benamingen te kopieren naar de desbetreffende pins die worden geconfigureerd via CubeMX dit zijn we in 4.2.3 Configuratie.

4.2.1.2 Ssd1306.c

Voor het gebruik van de driver zijn er drie functies die met de hardware gaan praten. Het is ook enkel in deze functies dat SPI transmitting wordt gebruikt.

- **void ssd1306_Reset(void)**
  Deze functie gaat de driver resetten, dit gebeurt door middel van gpio's.
- **void ssd1306_WriteCommand(uint8_t byte)**
  Deze functie wordt gebruikt om commando's naar de driver te sturen om bijvoorbeeld het display te initialiseren of om te laten weten dat we data gaan sturen.
- **void ssd1306_WriteData(uint8_t* buffer, size_t buff_size)**
  Deze functie gaat het framebuffer verzenden naar de driver.

main.c | Final.ioc | ssd1306.c | ssd1306_conf.h

```
    HAL_I2C_Mem_Write(&SSD1306_I2C_PORT, SSD1306_I2C_ADDR, 0x40, 1, buffer, buff_size, HAL_MAX_DELAY);
}

#elif defined(SSD1306_USE_SPI)

void ssd1306_Reset(void) {
    // CS = High (not selected)
    HAL_GPIO_WritePin(SSD1306_CS_Port, SSD1306_CS_Pin, GPIO_PIN_SET);

    // Reset the OLED
    HAL_GPIO_WritePin(SSD1306_Reset_Port, SSD1306_Reset_Pin, GPIO_PIN_RESET);
    HAL_Delay(10);
    HAL_GPIO_WritePin(SSD1306_Reset_Port, SSD1306_Reset_Pin, GPIO_PIN_SET);
    HAL_Delay(10);
}

// Send a byte to the command register
void ssd1306_WriteCommand(uint8_t byte) {
    HAL_GPIO_WritePin(SSD1306_CS_Port, SSD1306_CS_Pin, GPIO_PIN_RESET); // select OLED
    HAL_GPIO_WritePin(SSD1306_DC_Port, SSD1306_DC_Pin, GPIO_PIN_RESET); // command
    HAL_SPI_Transmit(&SSD1306_SPI_PORT, (uint8_t *) &byte, 1, 250);
    HAL_GPIO_WritePin(SSD1306_CS_Port, SSD1306_CS_Pin, GPIO_PIN_SET); // un-select OLED
}

// Send data
void ssd1306_WriteData(uint8_t* buffer, size_t buff_size) {
    HAL_GPIO_WritePin(SSD1306_CS_Port, SSD1306_CS_Pin, GPIO_PIN_RESET); // select OLED
    HAL_GPIO_WritePin(SSD1306_DC_Port, SSD1306_DC_Pin, GPIO_PIN_SET); // data
    HAL_SPI_Transmit(&SSD1306_SPI_PORT, buffer, buff_size, 250);
    HAL_GPIO_WritePin(SSD1306_CS_Port, SSD1306_CS_Pin, GPIO_PIN_SET); // un-select OLED
}

#else
#error "You should define SSD1306_USE_SPI or SSD1306_USE_I2C macro"
#endif
```

Figuur 21 ssd1306.c hardware functions

Voor data uit de applicatie naar het scherm te sturen zijn er ook nog een aantal functies die we gaan gebruiken.

- **void ssd1306_SetCursor(uint8_t x, uint8_t y)**
  Deze functie wordt gebruikt om aan te geven op welke positie op het scherm we de data willen tonen, hier gaan we de x en y as meegeven.

```
/* Position the cursor */
void ssd1306_SetCursor(uint8_t x, uint8_t y) {
    SSD1306.CurrentX = x;
    SSD1306.CurrentY = y;
}
```

Figuur 22 ssd1306_SetCursor

- **char ssd1306_WriteString(char* str, FontDef Font, SSD1306_COLOR color)**
  Wordt gebruikt voor het sturen van een string, deze functie maakt gebruik van de *WriteChar* functie.

```
277 /* Write full string to screenbuffer */
278 char ssd1306_WriteString(char* str, FontDef Font, SSD1306_COLOR color) {
279     while (*str) {
280         if (ssd1306_WriteChar(*str, Font, color) != *str) {
281             // Char could not be written
282             return *str;
283         }
284         str++;
285     }
286 
287     // Everything ok
288     return *str;
289 }
```

Figuur 23 ssd1306_WriteString

- **char ssd1306_WriteChar(char ch, FontDef Font, SSD1306_COLOR color)**
  Deze functie wordt gebruikt om een karakter te sturen daarvoor wordt gebruik gemaakt van de *DrawPixel* functie. De parameters die we hebben ingegeven bij de functie *SetCursor* worden hier gebruikt om met *DrawPixel* mee te geven.

```
243 char ssd1306_WriteChar(char ch, FontDef Font, SSD1306_COLOR color) {
244     uint32_t i, b, j;
245 
246     // Check if character is valid
247     if (ch < 32 || ch > 126)
248         return 0;
249 
250     // Check remaining space on current line
251     if (SSD1306_WIDTH < (SSD1306.CurrentX + Font.FontWidth) ||
252         SSD1306_HEIGHT < (SSD1306.CurrentY + Font.FontHeight))
253     {
254         // Not enough space on current line
255         return 0;
256     }
257 
258     // Use the font to write
259     for(i = 0; i < Font.FontHeight; i++) {
260         b = Font.data[(ch - 32) * Font.FontHeight + i];
261         for(j = 0; j < Font.FontWidth; j++) {
262             if((b << j) & 0x8000)  {
263                 ssd1306_DrawPixel(SSD1306.CurrentX + j, (SSD1306.CurrentY + i), (SSD1306_COLOR) color);
264             } else {
265                 ssd1306_DrawPixel(SSD1306.CurrentX + j, (SSD1306.CurrentY + i), (SSD1306_COLOR)!color);
266             }
267         }
268     }
269 
270     // The current space is now taken
271     SSD1306.CurrentX += Font.FontWidth;
272 
273     // Return written char for validation
274     return ch;
275 }
```

Figuur 24 ssd1306_WriteChar

- **void ssd1306_DrawPixel(uint8_t x, uint8_t y, SSD1306_COLOR color)**
  Deze functie gaat alle data in pixels in de framebuffer steken.

```
206 void ssd1306_DrawPixel(uint8_t x, uint8_t y, SSD1306_COLOR color) {
207     if(x >= SSD1306_WIDTH || y >= SSD1306_HEIGHT) {
208         // Don't write outside the buffer
209         return;
210     }
211 
212     // Draw in the right color
213     if(color == White) {
214         SSD1306_Buffer[x + (y / 8) * SSD1306_WIDTH] |= 1 << (y % 8);
215     } else {
216         SSD1306_Buffer[x + (y / 8) * SSD1306_WIDTH] &= ~(1 << (y % 8));
217     }
218 }
```

Figuur 25 ssd1306_DrawPixel

- **void ssd1306_Line(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, SSD1306_COLOR color)**
  Met deze functie kunnen we een lijn tekenen. Hier wordt gebruik gemaakt van Bresenhem's algorithme. Meer informatie hierover kan je vinden in de volgende link [Bresenham's line algorithm - Wikipedia]

```
297 /* Draw line by Bresenhem's algorithm */
298 void ssd1306_Line(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, SSD1306_COLOR color) {
299     int32_t deltaX = abs(x2 - x1);
300     int32_t deltaY = abs(y2 - y1);
301     int32_t signX = ((x1 < x2) ? 1 : -1);
302     int32_t signY = ((y1 < y2) ? 1 : -1);
303     int32_t error = deltaX - deltaY;
304     int32_t error2;
305 
306     ssd1306_DrawPixel(x2, y2, color);
307 
308     while((x1 != x2) || (y1 != y2)) {
309         ssd1306_DrawPixel(x1, y1, color);
310         error2 = error * 2;
311         if(error2 > -deltaY) {
312             error -= deltaY;
313             x1 += signX;
314         }
315 
316         if(error2 < deltaX) {
317             error += deltaX;
318             y1 += signY;
319         }
320     }
321     return;
322 }
323 
```

Figuur 26 ssd1306_Line

- **void ssd1306_UpdateScreen(void)**
  Deze functie zal de framebuffer naar de driver sturen door gebruik te maken van de functie's *WriteCommand* en *WriteData.*

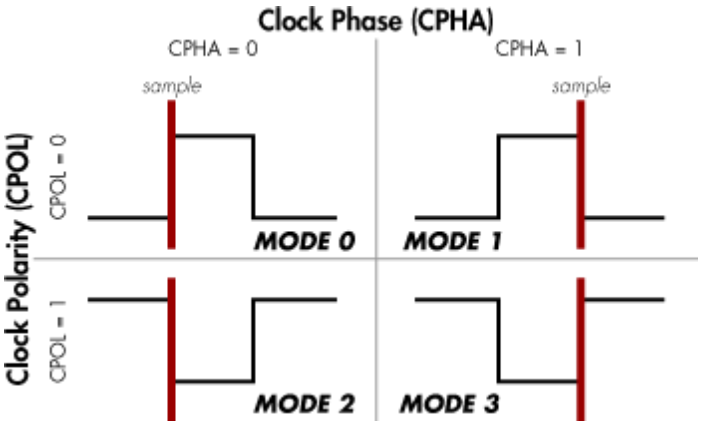
```
184 /* Write the screenbuffer with changed to the screen */
185 void ssd1306_UpdateScreen(void) {
186     // Write data to each page of RAM. Number of pages
187     // depends on the screen height:
188     //
189     //  * 32px   ==  4 pages
190     //  * 64px   ==  8 pages
191     //  * 128px  ==  16 pages
192     for(uint8_t i = 0; i < SSD1306_HEIGHT/8; i++) {
193         ssd1306_WriteCommand(0xB0 + i); // Set the current RAM page address.
194         ssd1306_WriteCommand(0x00 + SSD1306_X_OFFSET_LOWER);
195         ssd1306_WriteCommand(0x10 + SSD1306_X_OFFSET_UPPER);
196         ssd1306_WriteData(&SSD1306_Buffer[SSD1306_WIDTH*i],SSD1306_WIDTH);
197     }
198 }
```

Figuur 27 ssd1306_UpdateScreen

4.2.2 Onderzoek

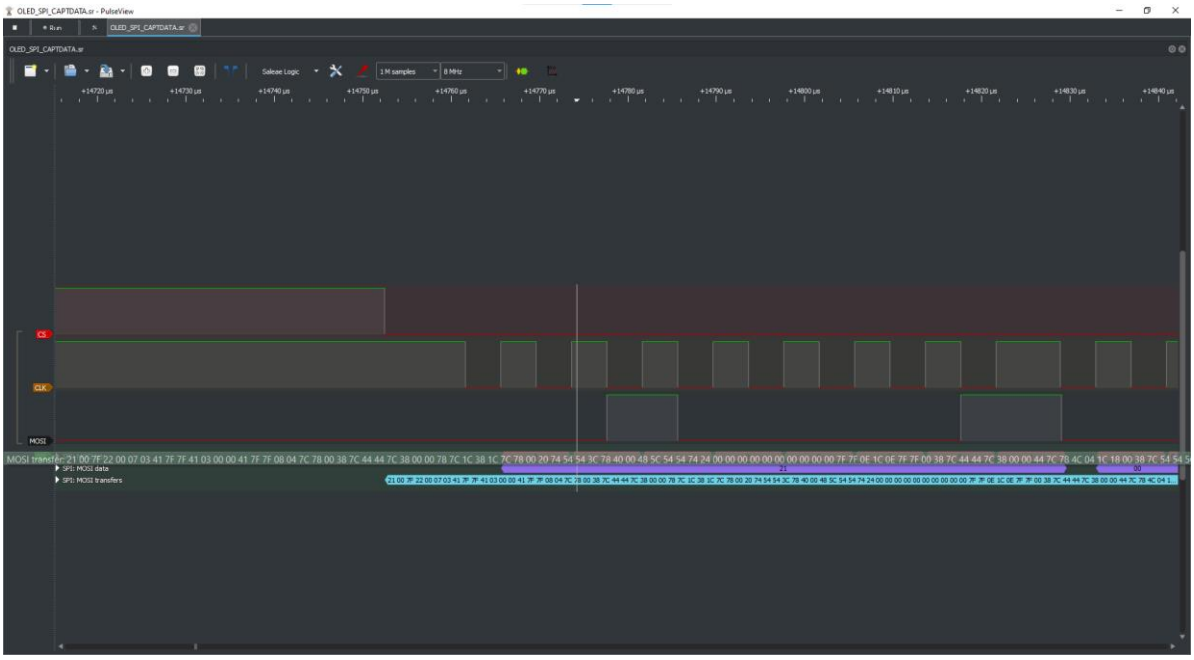
Voor dat we de configuratie in CubeMX kunnen doen hebben we nog een aantal gegevens nodig. SPI kan in verschillende modi worden gebruikt.

- Mode 0 -> Clock idle laag, sample 1ste flank
- Mode 1 -> Clock idle laag, sample 2de flank
- Mode 2 -> Clock idle hoog, sample 1ste flank
- Mode 3 -> Clock idle hoog, sample 2de flank



Figuur 28 SPI modi

Welke we gaan gebruiken hangt af van de SSD1306 driver. Het scherm heb ik in een ander project al gebruikt, ik heb dan met een logic analyzer nagemeten hoe deze in het ander project was geconfigureerd. Het resultaat is te zien in de volgende afbeelding.



Figuur 29 SPI modi analyzer

Zoals we kunnen zien is de clock idle hoog en wordt de data gesampeld op de 2de flank. Door dit zo te onderzoeken weet ik hoe SPI in CubeMx moet geconfigureerd worden.

4.2.3 Configuratie

GPIO

Zoals ik eerder had aangehaald moeten we een paar benamingen van define's toewijzen aan GPIO's. De GPIO heb ik hier geconfigureerd, het betreft:

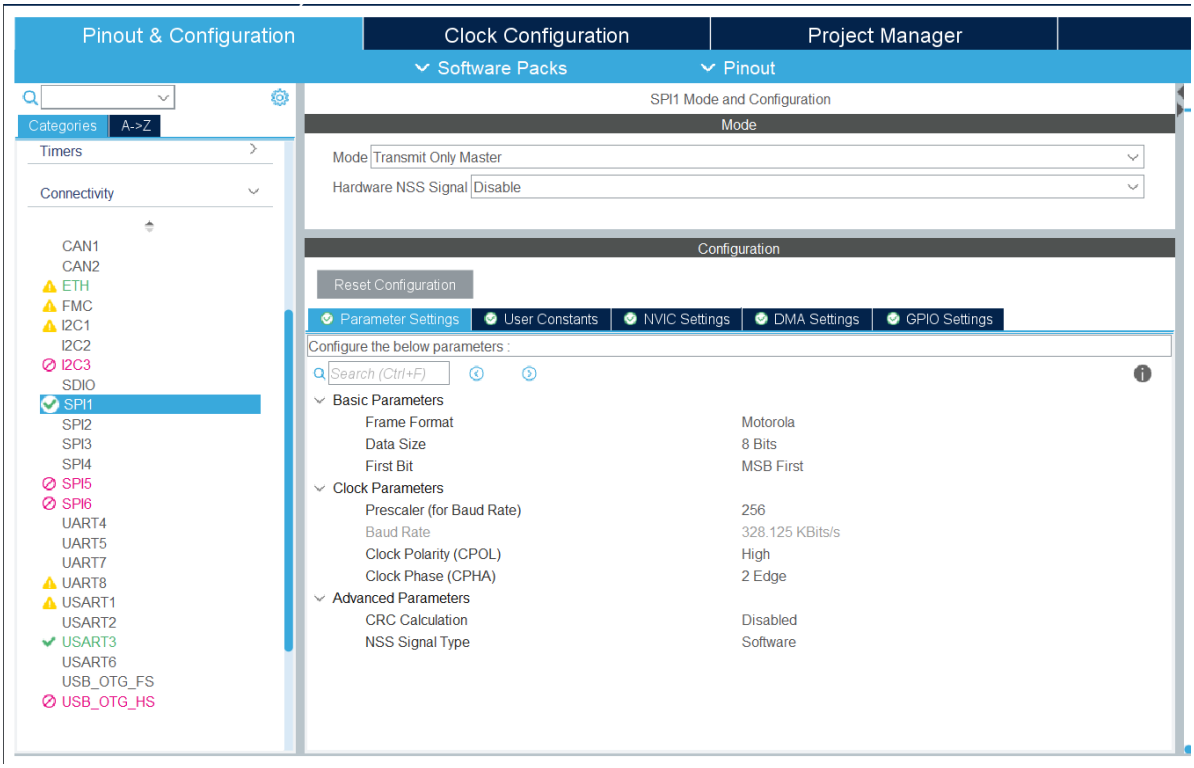
- PA4 -> OLED_CS
- PF12 -> OLED_DC
- PF13 -> OLED_Res

Pin Name	Signal ...	GPIO ou...	GPIO mode	GPIO Pull-up/Pull-down	Maximum output spee...	User Label	Modified
PB0	n/a	Low	Output Push Pull	No pull-up and no pull-down	Low	LD1 [Green]	☑
PB7	n/a	Low	Output Push Pull	No pull-up and no pull-down	Low	LD2 [Blue]	☑
PB14	n/a	Low	Output Push Pull	No pull-up and no pull-down	Low	LD3 [Red]	☑
PA4	n/a	Low	Output Push Pull	No pull-up and no pull-down	Low	OLED_CS	☑
PF12	n/a	Low	Output Push Pull	No pull-up and no pull-down	Low	OLED_DC	☑
PF13	n/a	Low	Output Push Pull	No pull-up and no pull-down	Low	OLED_Res	☑
PG1	n/a	High	Output Push Pull	No pull-up and no pull-down	Low	Relais_1	☑
PF9	n/a	High	Output Push Pull	No pull-up and no pull-down	Low	Relais_2	☑
PF7	n/a	High	Output Push Pull	No pull-up and no pull-down	Low	Relais_3	☑

Figuur 30 Config GPIO SPI

SPI

De SPI peripheral zelf wordt ook in CubeMX geconfigureerd. Ik heb voor de Transmit Only Master gekozen omdat we geen data terug verwachten en dus enkel MOSI gaan gebruiken. Verder hebben we uit het onderzoek geleerd dat we de mode van de SPI peripheral moeten configureren op Clock polariteit HIGH en Clock Phase 2de edge. Verder heb ik alles standaard gelaten. De configuratie kan terug gevonden worden in de onderstaande afbeelding.



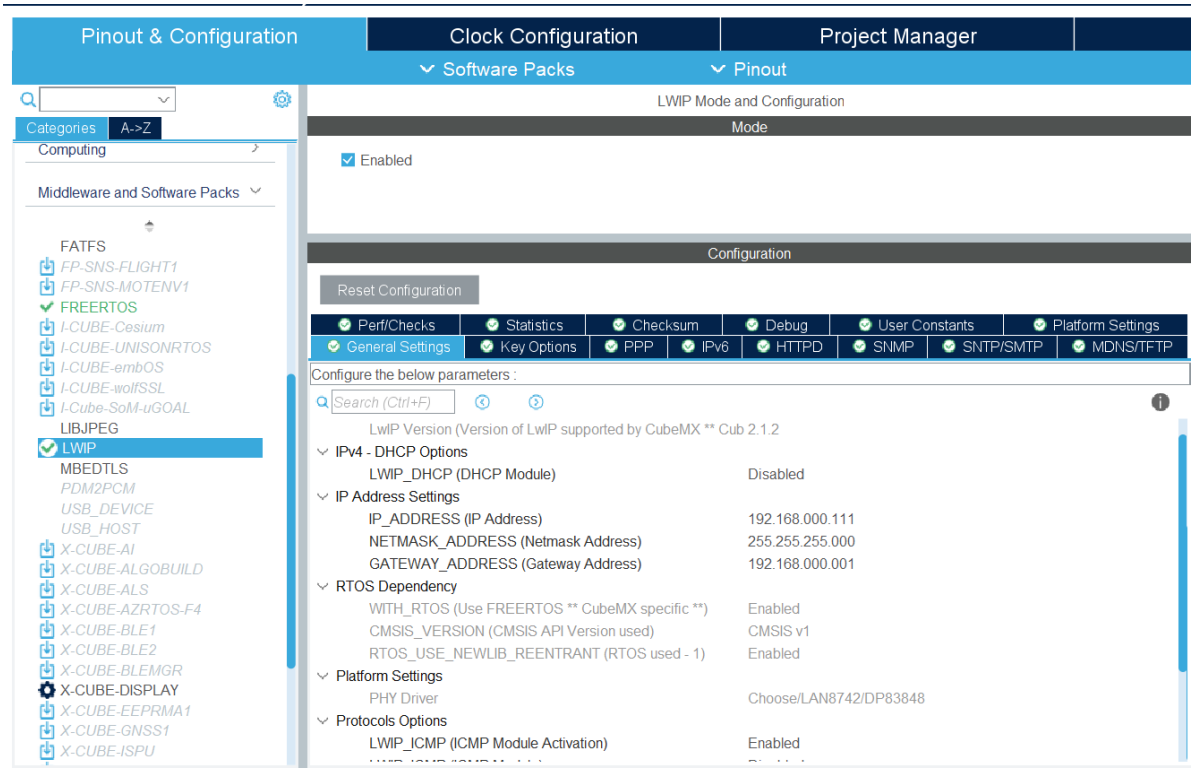
Figuur 31 CubeMX SPI

4.3 LwIP & MQTT

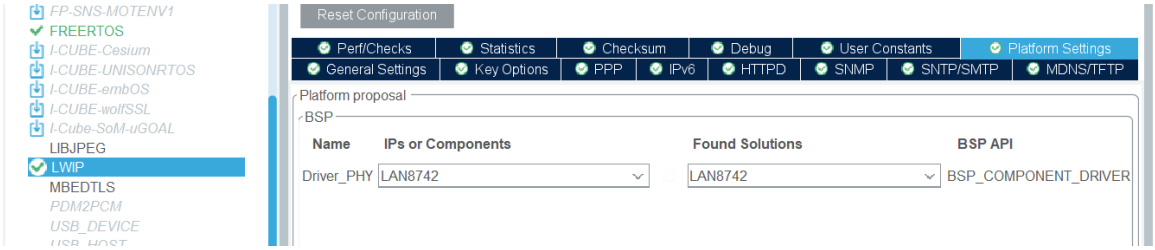
Voor het gebruik van LwIP en MQTT heb ik enkel LwIP moeten configureren in CubeMX. MQTT komt in een applicatie laag in de LwIP library zelf. LwIP zelf wordt enkel gebruikt door MQTT, in dit project heb ik geen applicatie geschreven dat rechtstreeks gebruik maakt (zonder MQTT) van LwIP.

4.3.1 LwIP

Voor de configuratie heb ik enkel een static ip ingesteld en de PHY geselecteerd.



Figuur 32 CubeMX LwIP Static IP



Figuur 33 CubeMX LwIP PHY

4.3.2 MQTT

Voor het gebruik van MQTT zijn er een aantal structs en functies nodig. Dit is nodig voor het aanmaken van een client, de initialisatie, het ontvangen en het versturen van berichten. Alle functies kunnen terug gevonden worden in de mqtt_global.c file.

- **mqtt_client_t* mqtt_client**
  Deze struct is nodig voor het aanmaken van een client, hier zitten alle benodigdheden in alsook het id van de client. De client wordt aangemaakt in de main file, vlak voor de initialisatie.

```
602    mqtt_client = mqtt_client_new();
603    mqtt_init(mqtt_client);
```

Figuur 34 mqtt_client_new

- **static const struct mqtt_connect_client_info_t mqtt_client_info**
  Deze struct wordt gebruikt voor bijkomende client informatie zoals een username en password, etc.

```
23 static const struct mqtt_connect_client_info_t mqtt_client_info =
24 {
25   "test", /* ClientId */
26   "mqtt", /* user */
27   "mqtt", /* pass */
28   0,  /* keep alive */
29   NULL, /* will_topic */
30   NULL, /* will_msg */
31   0,    /* will_qos */
32   0     /* will_retain */
33 #if LWIP_ALTCP && LWIP_ALTCP_TLS
34   , NULL
35 #endif
36 };
```

Figuur 35 mqtt_client_info

- **void mqtt_init(mqtt_client_t* mqtt_client)**
  In deze functie wordt het IP toegewezen en wordt er verbinding gemaakt met de broker.

```
38 void mqtt_init(mqtt_client_t* mqtt_client)
39 {
40       char uartmsg[50];
41       err_t err;
42       uint8_t IP_ADDRESS[4];
43       /* This can be modified to the arguments */
44       /* IP addresses initialization */
45       IP_ADDRESS[0] = 192;
46       IP_ADDRESS[1] = 168;
47       IP_ADDRESS[2] = 0;
48       IP_ADDRESS[3] = 164;
49
50       IP4_ADDR(&mqtt_ip, IP_ADDRESS[0], IP_ADDRESS[1], IP_ADDRESS[2], IP_ADDRESS[3]);
51
52       err = mqtt_client_connect(mqtt_client, &mqtt_ip, MQTT_PORT, mqtt_connection_cb, 0, &mqtt_client_info);
53
54       /* For now just print the result code if something goes wrong */
55       if(err != ERR_OK)
56       {
57         const char uartmsg[] = "mqtt_connect failed\n";
58         printf("mqtt_connect failed return: %d\n", err);
59         HAL_UART_Transmit(&huart3, uartmsg, sizeof(uartmsg), 100);
60         HAL_UART_Transmit(&huart3, (uint8_t)err, sizeof(err),100);
61       }
62       else
63       {
64         char uartmsg[] = "mqtt_connect succeed\n";
65         printf("mqtt_connect succeed");
66         HAL_UART_Transmit(&huart3, uartmsg, sizeof(uartmsg), 100);
67
68           const char *pub_payload= "INIT SUCCES";
69           const char *topic = "DEBUG";
70           u8_t qos = 0; /* 0 1 or 2, see MQTT specification */
71           u8_t retain = 0; /* No don't retain such crappy payload... */
72           err = mqtt_publish(mqtt_client, topic, pub_payload, strlen(pub_payload), qos, retain, mqtt_pub_request_cb, NULL);
73           if(err != ERR_OK)
74           {
75           printf("Publish err: %d\n", err);
76           HAL_UART_Transmit(&huart3, "Publish error", sizeof("publish error"), 250);
77           }
78       }
79 }
```

Figuur 36 mqtt_init

- **static void mqtt_connection_cb(mqtt_client_t *client, void *arg, mqtt_connection_status_t status)**
  Deze functie is de callback functie die wordt aangeroepen als de connectie gelukt of mislukt is. Hier wordt er dan als de connectie gelukt is, geabonneerd op topics.

```c
static void mqtt_connection_cb(mqtt_client_t *client, void *arg, mqtt_connection_status_t status)
{

  err_t err;
  if(status == MQTT_CONNECT_ACCEPTED)
  {
      char uartmsg[] = "mqtt_connection_cb: Successfully connected\n";
    printf("mqtt_connection_cb: Successfully connected\n");
    HAL_UART_Transmit(&huart3, uartmsg, sizeof(uartmsg), 100);

    /* Setup callback for incoming publish requests */
    mqtt_set_inpub_callback(client, mqtt_incoming_publish_cb, mqtt_incoming_data_cb, arg);

    /* Subscribe to a topic named "subtopic" with QoS level 0, call mqtt_sub_request_cb with result */
    err = mqtt_subscribe(client, "A", 1, mqtt_sub_request_cb, arg);
    err = mqtt_subscribe(client, "B", 1, mqtt_sub_request_cb, arg);
    err = mqtt_subscribe(client, "C", 1, mqtt_sub_request_cb, arg);
    err = mqtt_subscribe(client, "D", 1, mqtt_sub_request_cb, arg);

    if(err != ERR_OK) {
        char uartmsg[] = "mqtt_subscribe failed\n";
      printf("mqtt_subscribe return: %d\n", err);
      HAL_UART_Transmit(&huart3, uartmsg, sizeof(uartmsg), 100);

    }
  }
  else
  {
      char uartmsg[] = "mqtt_connection_cb: Disconnected\n";
    printf("mqtt_connection_cb: Disconnected, reason: %d\n", status);
    HAL_UART_Transmit(&huart3, uartmsg, sizeof(uartmsg), 100);

    /* Its more nice to be connected, so try to reconnect */
    mqtt_connect_broker();

  }
}
```

Figuur 37 mqtt_connection_cb

- **static void mqtt_incoming_publish_cb(void *arg, const char *topic, u32_t tot_len)**
  Deze callback wordt aangroepen wanneer er een bericht op een topic binnenkomt, hier wordt er gekeken naar het desbetreffende topic en er wordt een id gegeven.

```c
static int inpub_id;
static void mqtt_incoming_publish_cb(void *arg, const char *topic, u32_t tot_len)
{
  printf("Incoming publish at topic %s with total length %u\n", topic, (unsigned int)tot_len);

  /* Decode topic string into a user defined reference */
  if(strcmp(topic, "print_payload") == 0) {
    inpub_id = 0;
  } else if(topic[0] == 'A') {
    /* Relais 1*/
    inpub_id = 1;
  } else if(topic[0] == 'B'){
    /* Relais 2 */
    inpub_id = 2;
  } else if(topic[0] == 'C'){
    /* Relais 3 */
    inpub_id = 3;
  } else if(topic[0] == 'D'){
    /* Relais 4 */
    inpub_id = 4;
  }
}
```

Figuur 38 mqtt_incoming_publisch_cb

- **static void mqtt_incoming_data_cb(void *arg, const u8_t *data, u16_t len, u8_t flags)**
  Deze callback wordt ook aangeroepen als er een bericht is binnengekomen, hier gaan we vooral kijken op welk id er iets is binnengekomen en wat de data is. Hier gaan we er ook voor zorgen dat we relais gaan schakelen. Als we via Home Assistant een commando sturen voor het relais te schakelen gaat dat hier toekomen.

```
189 static void mqtt_incoming_data_cb(void *arg, const u8_t *data, u16_t len, u8_t flags)
190 {
191     uint8_t i = 0;
192   printf("Incoming publish payload with length %d, flags %u\n", len, (unsigned int)flags);
193
194   if(flags & MQTT_DATA_FLAG_LAST)
195   {
196     /* Last fragment of payload received (or whole part if payload fits receive buffer
197        See MQTT_VAR_HEADER_BUFFER_LEN)  */
198
199     /* Call function or do action depending on reference, in this case inpub_id */
200     if(inpub_id == 0)
201     {
202       /* Don't trust the publisher, check zero termination */
203       if(data[len-1] == 0)
204       {
205         printf("mqtt_incoming_data_cb: %s\n", (const char *)data);
206       }
207     }
208     else if(inpub_id == 1)
209     {
210       /* Call an 'A' function... */
211         if (data[0] == '1')
212         {
213             HAL_GPIO_WritePin(Relais_1_GPIO_Port, Relais_1_Pin,GPIO_PIN_RESET);
214         }
215         else
216         {
217             HAL_GPIO_WritePin(Relais_1_GPIO_Port, Relais_1_Pin,GPIO_PIN_SET);
218         }
219
220     }
221
222     else if(inpub_id == 2)
```

Figuur 39 mqtt_incoming_data_cb

- **uint8_t publish_message(mqtt_client_t *client ,const void * pub_payload , const char *topic, void *arg)**
  Met deze functie gaan we berichten versturen.

```
278 uint8_t publish_message(mqtt_client_t *client ,const void * pub_payload , const char *topic, void *arg)
279 {
280   err_t err;
281   u8_t qos = 0; /* 0 1 or 2, see MQTT specification */
282   u8_t retain = 0; /* No don't retain such crappy payload... */
283   err = mqtt_publish(client, topic, pub_payload, strlen(pub_payload), qos, retain, mqtt_pub_request_cb, pub_payload);
284   if(err != ERR_OK) {
285     printf("Publish err: %d\n", err);
286     HAL_UART_Transmit(&huart3, "Publish error", sizeof("publish error"), 250);
287
288   }
289   return err;
290 }
291
```

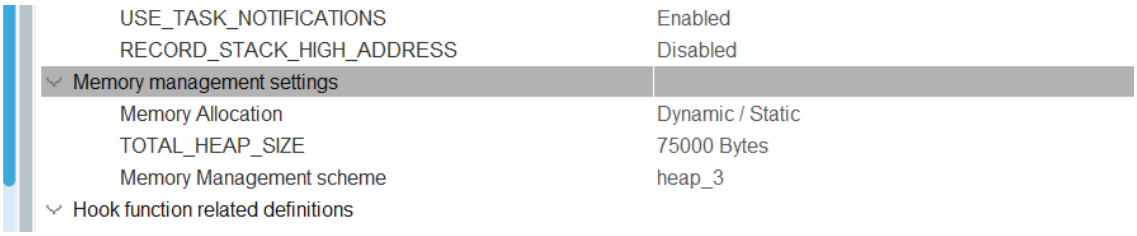
Figuur 40 mqtt publisch_message

4.4 FreeRTOS

FreeRTOS CMSIS_V1 is het operating system dat ik heb gebruikt in dit project.

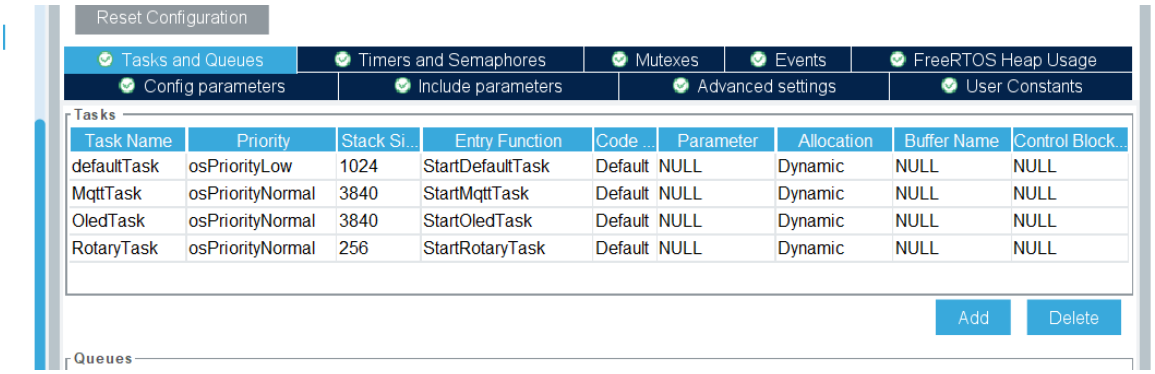
4.4.1 Configuratie

In de configuratie heb ik gebruik gemaakt van HEAP size 75000 bytes en maak ik gebruik van heap_3.



Figuur 41CubeMX RTOS memory

Het volgende dat is ingesteld zijn den tasks met hun priority's en stacksize en de queue.



Figuur 42 CubeMX RTOS tasks en queue

4.4.2 Global setters & getters

Setters en Getters worden in dit geval gebruikt om data in of uit een shared resource te halen. We maken hier gebruik van een mutex, van het moment dat task A één van deze functies aanroept zal niemand anders toegang krijgen tot deze shared resource. Pas wanneer task A klaar is kan task B aan de shared resource. Ik heb dit gebruikt voor:

- **Rotary encode value**
  Omdat de value van de rotary encoder een bijna alle tasks wordt gebruikt is het een must deze te beveiligen.

```
50 void SetROTvalue(uint16_t value)
51 {
52     if (xSemaphoreTake(Mutex_ROTvalue, 250))
53     {
54         ROTvalue = value;
55         xSemaphoreGive(Mutex_ROTvalue);
56     }
57 }
58
59 uint16_t GetROTvalue()
60 {
61     uint16_t value;
62     if (xSemaphoreTake(Mutex_ROTvalue, 250))
63     {
64         value = ROTvalue;
65         xSemaphoreGive(Mutex_ROTvalue);
66     }
67     return value;
68 }
69
```

Figuur 43 getter-setter rot value

- **ETH state**
  Omdat ik ervoor moet zorgen dat de mqtt initialisatie pas wordt uitgevoerd wanneer de ethernet connectie up is ga ik er op deze manier voor zorgen. Deze setter wordt pas gebruik wanneer de ethernet initialisatie voltooid is. Pas dan worden alle functies van mqtt vrijgegeven.

```
30 void SetETHstate(uint8_t status)
31 {
32     if (xSemaphoreTake(Mutex_ETHstate, 250))
33     {
34         ETHstatus = status;
35         xSemaphoreGive(Mutex_ETHstate);
36     }
37 }
38
39 uint8_t GetETHstate()
40 {
41     uint8_t status;
42     if (xSemaphoreTake(Mutex_ETHstate, 250))
43     {
44         status = ETHstatus;
45         xSemaphoreGive(Mutex_ETHstate);
46     }
47     return status;
48 }
49
```

Figuur 44 getter-setter ETH state

De mutexen worden geinitialiseerd in de main file.

```
70 SemaphoreHandle_t Mutex_ETHstate;
71 SemaphoreHandle_t Mutex_ROTvalue;
72
```

Figuur 45 mutex handles

```
142   /* USER CODE BEGIN RTOS_MUTEX */
143   /* add mutexes, ... */
144
145   Mutex_ROTvalue = xSemaphoreCreateMutex();
146   Mutex_ETHstate = xSemaphoreCreateMutex();
```

Figuur 46 mutex create

4.4.3 Queue

Omdat de rotary value uit een interrupt routine komt, wat trouwens geen deel uitmaakt van RTOS maar onafhankelijk ervan werkt kan dit niet zomaar als een globale variabele gedeeld worden. Mutexen kunnen ook niet gebruikt worden in een interrupt dus heb ik gekozen voor een queue. Het aanmaken gebeurd in de main file.

```
158   /* Create the queue(s) */
159   /* definition and creation of ROT_Queue */
160   osMessageQDef(ROT_Queue, 2, uint16_t);
161   ROT_QueueHandle = osMessageCreate(osMessageQ(ROT_Queue), NULL);
162
```

Figuur 47 queue create

4.4.3.1 Queue interrupt routines

De queue ga ik als eerst gebruiken in de interrupt routine van de rotary encoder die ik al had toegelicht in 4.1.2 Test. Hier da ik de value in de queue steken.

```
227             {
228                 Rot_cnt++;
229             }
230             else
231             {
232                 Rot_cnt--;
233             }
234         }
235
236         osMessagePut(ROT_QueueHandle, Rot_cnt, 0);
237
238         ROT_IRQ_Flag = 1;
239
240         IRQ_Triggerd = 0;
241     }
```

Figuur 48 queue put value rotary encoder

Verder ga ik in de routines ook gebruik maken van een reset knop waarmee de rotary value wordt gereset naar 0. Hier ga ik ook gebruik maken van de queue.

```
173 void EXTI0_IRQHandler(void)
174 {
175   /* USER CODE BEGIN EXTI0_IRQn 0 */
176     Rot_cnt = 0;
177     osMessagePut(ROT_QueueHandle, Rot_cnt, 0);
178   /* USER CODE END EXTI0_IRQn 0 */
179   HAL_GPIO_EXTI_IRQHandler(Rot_SW_Pin);
180   /* USER CODE BEGIN EXTI0_IRQn 1 */
181
182   /* USER CODE END EXTI0_IRQn 1 */
183 }
```

Figuur 49 queue put value reset rotary encoder

4.4.4 Tasks

Zoals in de configuratie al is getoond heb ik gebruik gemaakt van 4 thread/tasks.

- DefaultTask
- MqttTask
- OledTask
- RotaryTask

Het aanmaken van deze tasks gebeurt in de main file

```
155   /* Create the thread(s) */
156   /* definition and creation of defaultTask */
157   osThreadDef(defaultTask, StartDefaultTask, osPriorityLow, 0, 1024);
158   defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);
159
160   /* definition and creation of MqttTask */
161   osThreadDef(MqttTask, StartMqttTask, osPriorityNormal, 0, 3840);
162   MqttTaskHandle = osThreadCreate(osThread(MqttTask), NULL);
163
164   /* definition and creation of OledTask */
165   osThreadDef(OledTask, StartOledTask, osPriorityNormal, 0, 3840);
166   OledTaskHandle = osThreadCreate(osThread(OledTask), NULL);
167
168   /* definition and creation of RotaryTask */
169   osThreadDef(RotaryTask, StartRotaryTask, osPriorityNormal, 0, 1024);
170   RotaryTaskHandle = osThreadCreate(osThread(RotaryTask), NULL);
171
```

Figuur 50 create tasks

4.4.4.1 DefaultTask

In de default task wordt er niets gedaan, enkel de initialisatie van LwIP en de watchdogtimer. Ik heb daarom ook gekozen om de priority op low te zetten aangezien we buiten de LwIP initialisatie en de Watchdogtimer niets hiermee doen.

```
/* USER CODE END Header_StartDefaultTask */
void StartDefaultTask(void const * argument)
{
  /* init code for LWIP */
  MX_LWIP_Init();
  /* USER CODE BEGIN 5 */
#ifdef WDT
  MX_IWDG_Init();
#endif
  uint8_t txt[] = "Task1\n";

/* Infinite loop */
for(;;)
{

    HAL_UART_Transmit(&huart3, txt, sizeof(txt), 100);
      osDelay(150);
}
  /* USER CODE END 5 */
}
```

Figuur 51 DefaultTask

4.4.4.2 MqttTask

Deze task zorgt voor alles wat met MQTT te maken heeft, er zit een kleine state-machine in die er voor zorgt dat de initialisatie maar één keer gebeurd. Eens dat gedaan is wordt er enkel nog maar de value van de rotary encoder gestuurd via publish message. Voor dit gedaan wordt gaan we eerst de connectie met de broker controleren. Als dit niks oplevert gaan we naar een while(1) loop gaan en het systeem resetten, als er geen connectie is kunnen we niet veel meer doen, vandaar dat dit belangrijk is. Als we geconnecteerd zijn gaan we de rotary value halen met de rotary getter en kijken of deze veranderd is ten opzichte van de vorige value. Is dit het geval gaan we de value converteren naar een string en een linefeed eraan toevoegen. vervolgens wordt dit verzonden.

```
538   /* Infinite loop */
539   for(;;)
540   {
541       if(GetETHstate())
542       {
543 #ifdef WDT
544       HAL_IWDG_Refresh(hiwdg);
545 #endif
546           switch(state)
547           {
548           case MQTTINIT:
549               /* MQTT initialization must done here */
550               printf("State: %d \n", state);
551               mqtt_client = mqtt_client_new();
552               mqtt_init(mqtt_client);
553               state = MQTTSEND;
554               break;
555 
556           case MQTTSEND:
557               /* State sending data to the broker */
558               printf("State: %d \n", state);
559               if (Check_client_connected())
560               {
561                   CurrentValue = GetROTvalue();
562                   if (PreviousValue != CurrentValue)
563                   {
564                       sprintf(RotaryValueStr, "%d", CurrentValue);
565                       strcat(RotaryValueStr, "\n");
566                       publish_message(mqtt_client, RotaryValueStr, topic, NULL);
567                       PreviousValue = CurrentValue;
568                   }
569               }
570               else
571               {
572                   /* if client is not connected, reset */
573                   while(1)
574                   {
575                       HAL_NVIC_SystemReset();
576                   }
577               }
578               break;
579 
580           default:
581               break;
582           }
583       }
584     osDelay(100);
585   }
```

Figuur 52 MqttTask

4.4.4.3 OledTask

Deze task verzorgd alles wat met het display te maken heeft. Eerst gaat het de layout verzender die onveranderd blijft gedurende de applicatie. Daarna gaat het net zoals de MqttTask de rotary value ophalen met de rotary getter. Daarna gaan we kijken of deze veranderd is. Als deze veranderd is en de value is 0 dan gaan we de array helemaal leeg maken zodat de overige getallen worden verwijderd uit de array. Anders blijft dit op het scherm. Daarna gaan we de waarde als karakters in een aray steken. Dit wordt dan geprint op het scherm.

```
596 void StartOledTask(void const * argument)
597 {
598   /* USER CODE BEGIN StartOledTask */
599   char txt1[] = "STM32";
600   char txt2[] = "Rotary Index:";
601   char index[5];
602   uint16_t PreviousValue;
603   uint16_t CurrentValue;
604   uint8_t i;
605   ssd1306_SetCursor(0, 0);
606   ssd1306_WriteString(txt1, Font_11x18, White);
607   ssd1306_Line(0, 18, 90, 18, White);
608   ssd1306_Line(0, 19, 90, 19, White);
609   ssd1306_SetCursor(0, 25);
610   ssd1306_WriteString(txt2, Font_7x10, White);
611   ssd1306_UpdateScreen();
612   /* Infinite loop */
613   for(;;)
614   {
615       CurrentValue = GetROTvalue();
616     if (PreviousValue != CurrentValue)
617     {
618         if (CurrentValue == 0)
619         {
620             for(i = 0; i < 4; i++)
621             {
622                 index[i] = '\0';
623             }
624         }
625         sprintf(index, "%d", CurrentValue);
626         ssd1306_SetCursor(50, 45);
627         ssd1306_WriteString("              ", Font_7x10, White);
628         ssd1306_SetCursor(50, 45);
629         ssd1306_WriteString(index, Font_7x10, White);
630         ssd1306_UpdateScreen();
631         HAL_UART_Transmit(&huart3, index, sizeof(index), 250);
632         PreviousValue = CurrentValue;
633     }
634     osDelay(100);
635   }
636   /* USER CODE END StartOledTask */
637 }
```

Figuur 53 OledTask

4.4.4.4 RotaryTask

In de rotary task gaan we eerst controlleren of er een nieuwe waarde in de queue zit. Deze gaan we dan opslaan in de Rotary setter. Die we dan in al de andere tasks gaan gebruiken. Dit doe ik omdat wanneer ik dit uit de queue haal deze waarde ook effectief uit de queue wordt verwijderd. Daarom dat we deze gaan opslaan in de rotary setter. Die kunnen we dan blijven gebruiken totdat ze hier weer veranderd wordt. Verder ga ik controlleren als de opgehaalde waarde boven de 100 en boven de 500 zit zodat we hier ook weer relais kunnen schakelen.

```
698 /* USER CODE END Header_StartRotaryTask */
699 void StartRotaryTask(void const * argument)
700 {
701   /* USER CODE BEGIN StartRotaryTask */
702       osEvent  evt;
703 
704   /* Infinite loop */
705   for(;;)
706   {
707       evt = osMessageGet(ROT_QueueHandle, 0);
708       if (evt.status == osEventMessage)
709       {
710           SetROTvalue(evt.value.p);
711       }
712 
713     if (evt.value.p >= 100)
714     {
715 
716         HAL_GPIO_WritePin(Relais_5_GPIO_Port, Relais_5_Pin, GPIO_PIN_RESET);
717 
718     }
719     else
720     {
721 
722             HAL_GPIO_WritePin(Relais_5_GPIO_Port, Relais_5_Pin, GPIO_PIN_SET);
723 
724     }
725     if (evt.value.p >= 500)
726     {
727         HAL_GPIO_WritePin(Relais_6_GPIO_Port, Relais_6_Pin, GPIO_PIN_RESET);
728     }
729     else
730     {
731         HAL_GPIO_WritePin(Relais_6_GPIO_Port, Relais_6_Pin, GPIO_PIN_SET);
732     }
733     HAL_UART_Transmit(&huart3, "Task Rotary\n", sizeof("Task Rotary\n"), 250);
734     osDelay(100);
735   }
736   /* USER CODE END StartRotaryTask */
737 }
```

Figuur 54 RotaryTask