Prosjektoppgave Vår20

NTNU

Kandidater (Etternavn, Fornavn):

Vu, Johan

Berg, Michael

Jacob, Stefan

Premkumar, Bharat

Almenningen, Elias Olsen

Dato:	Fagkode:	Gruppe (Navn/Nr):	Ant sider/bilag:	Bibl. nr:
10/04	IELET1002	Gruppe 29	66/ 11	NA

Faglærer(e):

Arne Midjo

Tittel:

Prosjektoppgave i Mikrokontrollerprogrammering og Sensorer

Sammendrag:

Skriv et kort sammendrag av hva dere har utviklet

I dette prosjektet har vi valgt å fordype oss i modulene Zumo Linje og ESP32 Sensornode.

Zumo Linje:
Vi har utviklet et brukergrensesnitt på roboten hvor det skal være mulig å velge hvilken kjøremåte man har lyst å utføre. Dette gjøres ved å bla med robotens innebygde knapper i henholdsvis til en beskrivende test på LCD-skjermen. Det skal være fem valgmuligheter i menyen; kjøre i firkant, kjøre sirkel, kjøre slalåm og linjefølging. I tillegg er det anvendt flere ulike sensorer til å lage et program som får roboten til å unngå kollisjon (Zumo Obstacale avoidance).

ESP32 Sensornode:
Vår sensornode inneholder tre sensorer, VL6180x ToF, NTC termistor og en MQ2 gass sensor. Sensornoden måler lysintensitet, temperatur og gasser i lufta. Vi har koblet sensornoden opp mot Blynk og en webserver. Sensornoden har en buzzer og en rød lysdiode som fungerer som en alarm. Det er i tillegg lagt til en LCD-skjerm og en knapp.

INNHOLDSFORTEGNELSE

1 MODUL - ZUMO LINJE

1.1 SAMMENDRAG

Denne delen av rapporten inneholder prosessen gruppen gikk gjennom for å fullføre kravene til modulen «Zumo linje». Først blir terminologien og fagbegreper definert og forklart, før problemstillingen blir gjennomgått og de ulike oppgavene blir fordelt innad i gruppen. Videre går vi igjennom de tre ulike fasene av prosjektet: design, bearbeiding og resultater, og drøfting. Til slutt konkluderer vi i prosjektet som en helhet, der vi ser på hva vi har lært, hva vi skulle ønske vi kunne arbeidet med videre og på hvordan vi håndterte oppgaven som gruppe.

1.2 TERMINOLOGI

Gyroskop -	En sensor som brukes til å måle eller opprettholde orientering og vinkelhastighet.
Zumo32U4 -	En Arduino kompatibel robotbil med en rekke innebygde sensorer som: akselerator, gyroskop og nærhetssensor.
Encoder -	Komponent som brukes til å lese og regne hastighet og posisjon av et objekt den er koblet til.
LCD -	En elektrisk skjerm som man kan skrive på gjennom en mikrokontroller.
Akselerometer -	En komponent som måler akselerasjon til et objekt. Måleenheten er meter per sekund i kvadratet: m/s^2 eller i G-krefter (g). En enkelt G-styrke for oss her på planeten Jorden tilsvarer 9.81 m/s^2. Akselerometre er nyttige for å registrere vibrasjoner i systemer eller for orienteringsapplikasjoner.
Proximity sensor -	En sensor som kan sanse objekter i nærheten uten noe form for fysisk kontakt
Linjefølger sensor –	Et sett med sensorer som kan lese posisjonen til Zumoen på bakken
PWM -	Puls Width Modulation

1.3 INNLEDNING – PROBLEMSTILLING

Programmeringsdelen i oppgaven er tredelt, påfulgt av en generell beskrivelse av hvordan Zumoen kjører og fungerer. Første problemstilling er å få roboten til å kjøre autonomt i forskjellige mønstre. Mønstrene er å kjøre i firkant, kjøre i sirkel, kjøre rett fram-snu-kjøre tilbake og kjøre slalåm mellom kjegler. I tillegg skal mønsteret velges ved bestemte knappetrykk, som betyr at roboten må ha et brukergrensesnitt for å fritt kunne velge hvilket mønster som skal kjøres til enhver tid. Andre del av oppgaven er å få roboten til å følge en tapebane med ulike utfordringer som slake og krappe svinger, blindvei m.m. Bilen skal med

andre ord kunne kjøre rundt banen på en behagelig måte, med og uten PID-regulator. Siste problemstilling er å anvende de ulike sensorene i Zumoen og til et prosjekt. Her er det valgt å lage en Obstacle Avoidance system for Zumoen. Hvor flere sensorer av Zumoen kom til bruk som: Proximity sensor, Encoders, Akselerometer og Gyroskop.

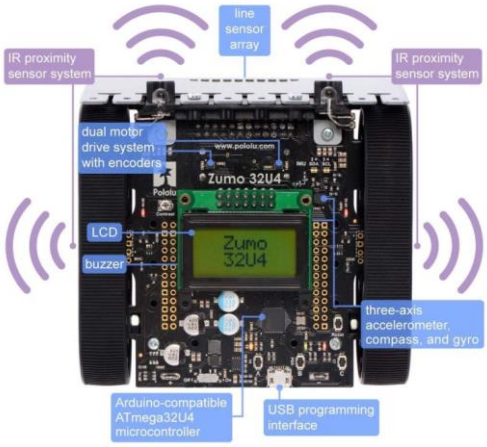
1.4 BAKGRUNN - TEORETISK GRUNNLAG

Gyroskop

Et gyroskop er en sensor som brukes til å måle eller opprettholde en gjenstands orientering og vinkehastighet rundt omkring i området. Det fungerer som en slags svinghjul, med rotasjonsakser satt opp med flere ringer symmetrisk om hverandre. Symmetrien om hjulene symboliserer retningen roboten skal beveges seg i. Dette er godt forklart i metoden.

Zumo32U4

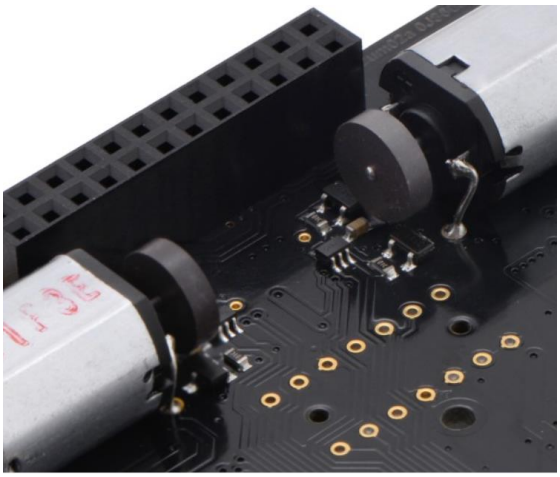
Zumo32U4 er en robotbil som styres ved hjelp av en Arduino-kompatibel mikrokontroller kalt ATmega32U4. Den sørger for at roboten kan overføre og innhente data mellom mikrokontrolleren og selve roboten. Roboten får instrukser fra egenskrevne koder til å utføre forskjellige oppgaver i form av autonom styring. Den har innebygde sensorer som: akselerator, gyroskop, og nærhetssensor. Den bruker også en H-bro som dreier på motoren og andre innebygde sensorer. Selve roboten er utstyrt med knapper med egendefinert grensesnitt i form av en input, LED-indikatorer og en LCD-skjerm for å informere eller vise informasjon vi etterspør.



Figur 1: Bilde av Zumo32U4 75:1 HP [17]

Encoder

Encoderen har en puls på 12 tellinger per omdreining av motorakselen. For å beregne antall puls per omdreining, multipliserer man girkassenes gir-forhold med 12. Zumoen har 75: 1 motorer, det mer nøyaktige girforholdet er 75,81: 1 [20]. Det betyr at en omdreining tilsvarer $75{,}81 \times 12 \approx 909{,}7$ puls. Lengden per omdreining regnes ut ved å finne omkretsen til en hjulaksel som gir en lengde på ≈ 12 cm.



Figur 2: Bilde av Encoders brukt i Zumo[17]

Akselerometer

Et akselerometer brukes til å navigere og styre luftfartøyer og f.eks. ubåter. Dette er nødvendig for å beregne akselerasjonen mot et punkt om det skal f.eks. skal unngå en gitt posisjon. Det er vanlig å bruke et akselerometer sammen med et gyroskop. Prinsippet bak målingene til et akselerometer er at den ene delen skal beveges seg fritt imens den andre delen beveger seg i forhold til underlaget, og ut ifra dette vil avstanden eventuelt kraften bli målt som en vektorstørrelse. Akselerometeret er i chipen ST LSM303D sammen med et magnetometer, der magnetometeret brukes til å kalibrere akselerometeret. Den fungerer med spenningsnivå 3.3V og er koblet til ATmega32U4 som opererer med en spenning på 5V. Dette lar seg gjøre ved at vi har såkalte «level shifters» innebygget i hovedbordet som gjør at det er mulig å oversette signaler med en type spenningsnivå til en annen.

Nærhetssensor (Proximity sensor)

På Zumo32U4 består proximity sensoren av fire deler «left, center-left, center-right and right». Sensorene sender ikke noe form for lys, men kan fange IR signaler gitt ved 38KHz på hovedbrettet. Objekter blir oppdaget ved at det er endringer i felt eller i retursignalet. Det finnes forskjellige nærhetssensorer til forskjellige bruk. Eksempler på dette kan være en kapasitiv nærhetssensor som er egnet for et plastmål kontra en induktiv nærhetssensor som krever et metallobjekt som mål. I dette tilfellet blir nærhetssensoren brukt som en slags berøringsbryter. Eks. roboten avverger en kollisjon ved at nærhetssensoren gir et signal på at et objekt er innenfor bevegelsessonen.

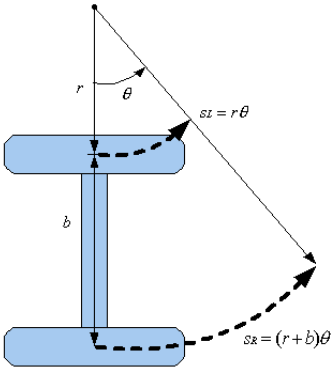
Linjesensor

Foran ved “front sensor arrayen” til Zumo32U4 ligger det 5 linjesensorer fordelt utover “sensorarrayen”: en helt til høyre, en helt venstre, og tre i midten. Linjesensorene søker etter flater som enten er lyst eller mørkt, men kan også bli brukt til å detektere IR-lys fra f.eks solen. Dette gjøres slik at roboten skal kunne kjøre i en gitt teipbane. Sensorene består av en infrarød (IR) transmitter og en photoresistor. IR-transmitteren sender ut IR-lys. Lyset blir reflektert av bakken, og videre avlest av phototransistoren. Ved å bruke data fra alle sensorene kan vi dermed

finne ut hvordan zumo32U4 kjører i forhold til teipbanen, og skriver følgende linjekode ut ifra dette.

Motorfartforhold i en sirkelbevegelse

Når Zumoen skal ta en sving, må den ene motorfarten være større enn den andre. I eksemplet på figur 3, utfører roboten en venstresving, som vil bety at høyremotoren er større enn venstremotoren. For å beholde en konstant avstand fra sentrum, må forholdet mellom motorene være en bestemt verdi, slik at roboten ikke over- eller underkjører.



Figur 3: Distansen til robotens motorer i en sving [21]

Strekningen til venstremotor S_L er definert som r*q, mens høyremotorens banelengde S_L blir (r+b)*q. Variabelkonstanten "r" er definert som radiusen fra sentrum til roboten, mens "b" er bredden på roboten. Ettersom motorene skal bevege seg like mange grader i en sving, i tillegg til at det skal skje like kjapt, blir variablene theta og tiden felles for å beregne farten til hver av motorene. Forholdet mellom dem blir derfor:

Strekning til venstre- og høyremotor:

$$S_L = r\theta$$
$$S_R = (r + b)\theta$$

Farten til venstre- og høyremotor:

$$fart = \frac{strekning}{tid}$$

$$V_L = \frac{r\theta}{t}$$

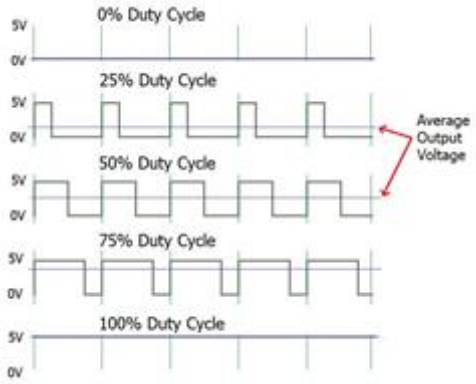
$$V_R = \frac{(r+b)\theta}{t}$$

Forholdet mellom venstre- og høyremotor:

$$\frac{V_L}{V_R} = \frac{\frac{r\theta}{t}}{\frac{(r+b)\theta}{t}}$$
$$\frac{V_L}{V_R} = \frac{r}{r+b}$$

PWM

PWM står for Pulse-Width Modulation (PBM eller pulsbreddemodulering på norsk). Det er en måte å få analoge verdier ut ifra digitale hjelpemidler. Dette får vi til ved å manipulere hvor lenge en pulsbredde er på et firkantpulstog. Ved å variere/modulere lengden på denne «på tiden», får man sendt ut ulike analoge signaler, som kan styre styrken en LED lyser med, hvor lenge en servo skal svinge, etc.



Figur 4: PWM illustrasjon ***Error! Reference source not found.***

Globale vs. lokale variabler og generell informasjon

En variabel er en plass hvor man lagrer data i form av navn, en verdi eller type. Variabler er kategorisert i to grupper: globale variabler og lokale variabler. Globale variabler er ikke deklarert innenfor en funksjon, og er derfor tilgjengelig over hele koden. De er som regel deklarert øverst i kodeskissen. Lokale variabler derimot er variabler som er deklarert i en funksjon, og vil kun operere under den gitte funksjonen. Det kan
konkluderes at globale variabler og lokale variabler har samme oppbyggingsmønster, men forskjellige områder de opererer på.

Variabler er forskjellige, og vi har såkalte «typer» som bestemmer hva slags type tegn variabelen skal ta i bruk. Under «Type» så deklarer vi først hvor stor plass vi trenger ved å velge imellom "short, unsigned short, unsigned, long, long long, signed, float, double og long double." De beskriver hvor mye data/bytes de kan lagre i forhold til datatypen, i form av et intervall. Det er derfor vanlig å lagre data i bestemte lengder, for å hindre og bruke mer minne enn nødvendig. Vi kan finne verdiene til typene på bildet under, i kolonnen: «Range». Videre beskriver vi hva slags datatype vi ønsker å deklarere, altså hvilke verdier/tegn vi ønsker at variabelen skal inneholde. Eksempler på det er «int», «char» og «double». «Char» er en datatype som lagrer bokstaver, i form av en bokstav eller en streng, men kan også inneholde tall. Det går derfor an å utføre aritmetiske operasjoner, men den vil følge ASCII tabellen.
Størrelsen er på 8 bit, men lengdetypen gjør at rangen vil være forskjellig. Fordi «**unsigned char**» kun holder på positive variabler, kontra «**signed char**» som kan holde på både negative og positive variabler og negative (se på bildet under). Vi har også datatypen «int», som lagrer heltall, som er både positive og negative. Den lagrer negative tall ved å benytte 2.komplementsmetoden. Float er en annen datatype som lagrer tall med desimaler. «Floating-point numbers» blir som oftest brukt til analoge og kontinuerlige verdier. Til slutt tar vi for oss datatypen double, som fungerer på samme måte som datatypen «float». Forskjellen er at «double» er dobbelt så *nøyaktig* som «float», og kan dermed lagre dobbelt så mange desimaler bak kommaet.

Type	Bytes	Bits	Range	
short int	2	16	-32,768 -> +32,767	(32kb)
unsigned short int	2	16	0 -> +65,535	(64Kb)
unsigned int	4	32	0 -> +4,294,967,295	(4Gb)
int	4	32	-2,147,483,648 -> +2,147,483,647	(2Gb)
long int	4	32	-2,147,483,648 -> +2,147,483,647	(2Gb)
long long int	8	64	veldig langt	
signed char	1	8	-128 -> +127	
unsigned char	1	8	0 -> +255	
float	4	32		
double	8	64		
long double	12	96		

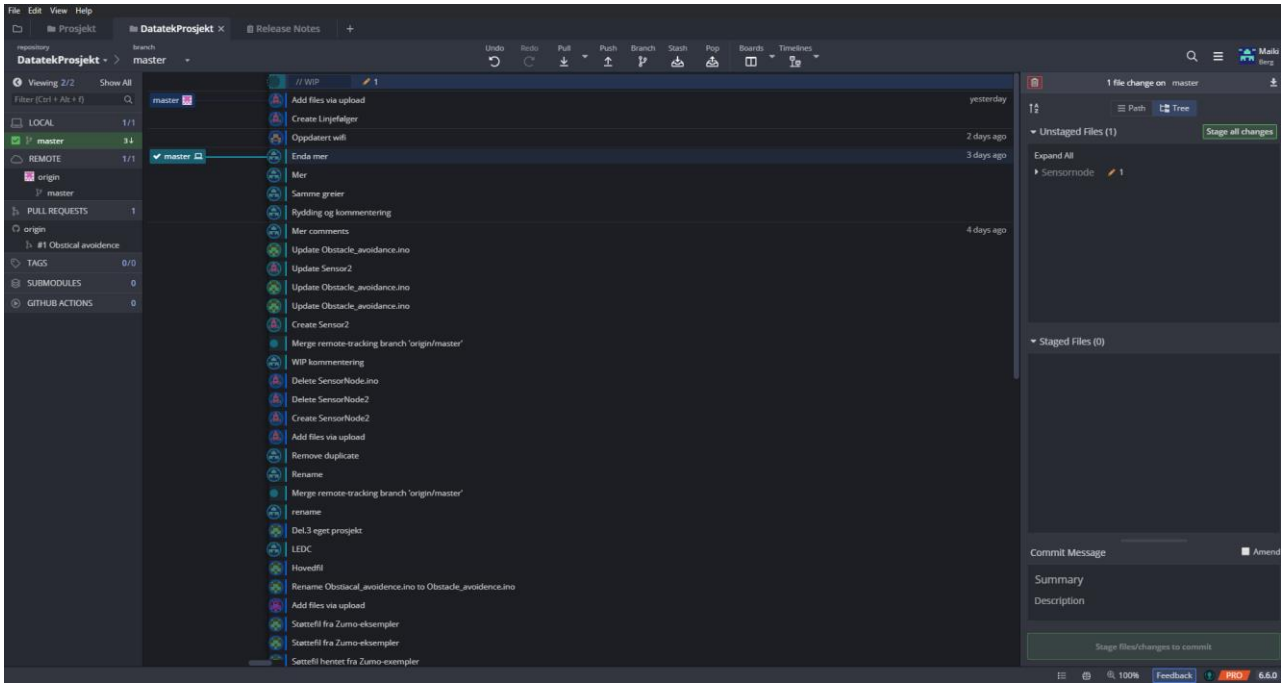
Figur 5: Datatyper

1.5 METODE – DESIGN

For å kunne fordele arbeidsoppgaver på en effektiv måte, var en strukturert plan nødvendig. Hele gruppen delte seg i to for å kunne gå mer i dybden i hver sin modul, og mot slutten var planen å presentere arbeidet for hverandre, slik at alle fikk et innblikk i hva resten gjorde. Gruppen på tre som fikk ansvaret for Zumo Linje, fordelte seg igjen slik at bare én person jobbet med én deloppgave. Om noen ble ferdig med sin del, skulle personen hjelpe andre i delgruppen sin for å oppnå forventet resultat.

Planen var å primært bruke Github til deling av kode, hvor denne nettsiden er en åpen kilde tilrettelagt for at folk blant annet skal kunne publisere og dele prosjekter. I tillegg kunne Microsoft Teams bli brukt til gruppekommunikasjon og møter. Skjermdeling er en nyttig funksjon i Teams, siden da er det mulighet at andre samhandler med koden til personen som deler hvis han satt fast med noe. Roboten skulle også bli byttet rundt jevnlig, slik at alle fikk prøvekjørt koden de hadde laget. Om det ble nødvendig å simulere koden uten Zumoen tilgjengelig, ble det en enighet om å få den sendt på Github slik at personen som hadde roboten på det tidspunktet, fikk prøvekjørt koden innen kort tid. Av den grunn ble prosjektet ikke saknet ned, til tross for at folk ikke alltid har mulighet til å samle seg.

For å gjøre det enda enklere å dele kode brukte vi GitKraken for å redige og dele kode sømløst. Med GitKraken blir det som å dele en mappe med gruppen din, der alle har tilgang til å redigere på innholdet i mappen. Det redigerte innholdet i mappen kan så deles til resten av gruppen der de kan hente inn det redigerte innholdet ved ett knappetrykk. Det er sammenlignbart med å dele et Word-dokument med gruppen sin.



Figur 6: GitKraken brukergrensesnitt

Vi bestemte oss for at koden skulle kommenteres godt og deles opp i avsnitt. Før vi begynte å skrive kode skulle vi lage flytdiagram for å få en oversikt over oppgaven og hvordan koden skulle fungere.

```
//============================================================
//          Dette er et eksempel på hvordan kodeavsnitt skal deles opp
//============================================================
```

Sensoren som gruppen benyttet seg av i Zumo Linje var:

- Encoder
- Gyro
- Nærhetssensor (Proximity)
- Akselerometer
- Linjefølgersensor

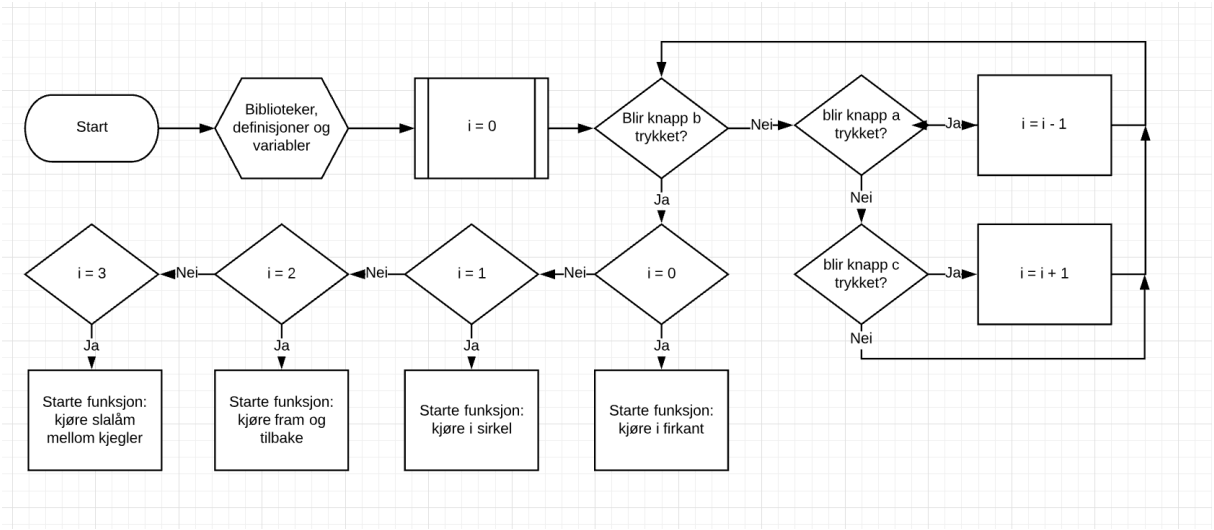
Bibliotek for disse sensorene var dermed en nødvendighet for å kunne bruke de.

For å lage en strukturert og brukervennlig kode, var planen å bygge oppgaven på mange funksjoner. Tankegangen bak dette var at hver enkelt funksjon har én spesifikk oppgave, slik

at når de blir brukt, vil funksjonsnavnet gi en overordnet beskrivelse av hva som foregår. Forskjellige deler av koden blir lettere å finne, ettersom de er kategorisert i funksjoner. Av den grunn blir koden oversiktlig.

Meny

Metoden til oppgaven om autonom kjøring var veldig rett frem. Koden skulle i hovedsak bestå av fire funksjoner for de ulike kjøremåtene, samt et brukergrensesnitt for å kunne velge mellom funksjonene. Det var ikke nok knapper på Zumoen til hver funksjonalitet, og av den grunn måtte en annen form for framvisning utvikles. Det kom fram til en bla-funksjonalitet med switch case hvor de innebygde Zumo knappene A og C var til å bla med, mens knapp B skulle utføre den kjøremåten som var utvalgt. LCD-skjermen måtte også oppdatere seg samtidig som man bladde med knappene. Denne fremgangsmåten gjør det i tillegg lett å legge inn flere kjøremåter i framtiden, ettersom det bare er å utvide switch-casen.

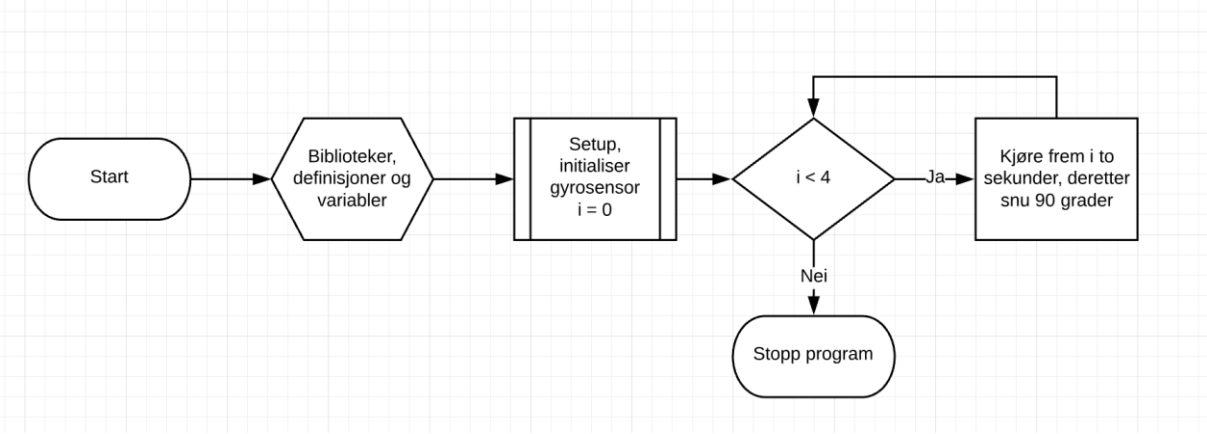


Figur 7: Flytdiagram til menyen på Zumoen

Kjøre i firkant

Første funksjon som skulle få Zumoen til å kjøre i firkant, og ville bli bygget av en løkke som repeterte seg selv fire ganger. Den skulle på et vis kjøre rett frem, ta en 90-graders sving, også gjøre det igjen. Da vil den i teorien komme tilbake til startpunktet. Å få motorene til å kjøre rett fram var kun til å sette motorfarten til venstre og høyre det samme, og kjøre de like lenge. For å snu presist 90 grader, var planen om å bruke en gyrosensor en grei løsning. Ettersom den

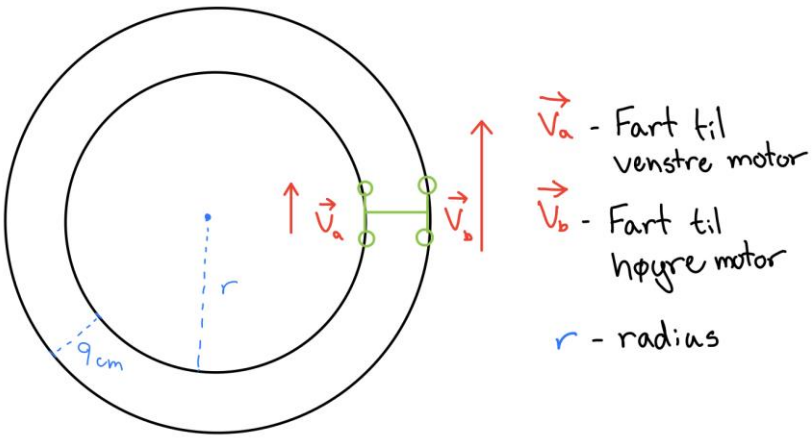
måler hvor mange grader roboten snur seg etter startpunktet, skulle motorene bli satt i en while-løkke hvor motorene tok en høyre sving helt til antall ønskede grader var nådd. Deretter skulle den som nevnt kjøres om igjen fire ganger, og i utgangspunktet slutte der den begynte. Dette er flytdiagrammet til koden.



Figur 8: Flydiagram til roboten som skal kjøre i en firkant

Kjøre i sirkel

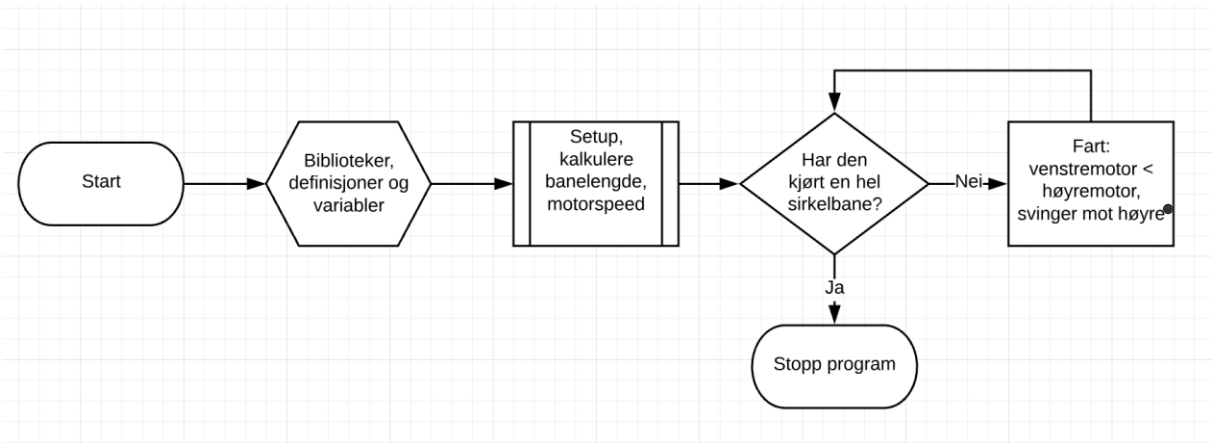
For å kunne kjøre i sirkel, var planen for utførelsen å definere kraften til venstre- og høyremotor etter hvor langt de skulle kjøre, henholdsvis til en vilkårlig radius. Et faktum var at den ene motorkraften var større enn den andre, grunnet av at det var forskjellig avstand fra sentrum og den ene hadde lengre kjørebane enn den andre. For at begge motorene skulle være på samme punkt, måtte det derfor bli en hastighetsforskjell, som vist på bildet under.



Figur 9: Tegning av banen til roboten i en sirkelbevegelse

Bredden til Zumoen ble målt til å være 9 cm. Dette medførte til at sirkelbanen til venstremotor V_a ville være 2*π*r, mens sirkelbanen til høyremotor V_b ville være 2π(r+9). Med den informasjonen, ville forholdet mellom motorene være r/(r+9). Beviset for at dette forholdet stemmer, finnes i teoridelen. Det ville med andre ord bety at hvis standardfarten til den ytterste motoren var 100, ville den innerste motoren ha en styrke på $100 * \frac{r}{r+9}$.

Ettersom hastigheten da var definert til å kjøre i en sirkelbane med en vilkårlig radius, måtte roboten vite når den skulle stoppe. Metoden som ble brukt var å ta i bruk Encoders for å finne ut hvor langt hjulene hadde kjørt til en gitt tid. Som forklart i teoridelen, var en rotasjon rundt 910 puls. Denne informasjonen måtte bli omgjort til centimeter, hvor 1 rotasjon tilsvarte rundt 12 cm. Ved å ta sirkelbane delt på 12cm, fikk man antall rotasjoner til hjulet, og deretter multiplisere med 910 puls for å finne antall puls for å fullføre en hel sirkel. Det siste som gjenstod var det å sette motorene i en while-løkke, med vilkårene at motorene skulle kjøres helt til Encoderene til motorene nådde ønsket mål. Flytdiagrammet til sirkelprogrammet så slik:

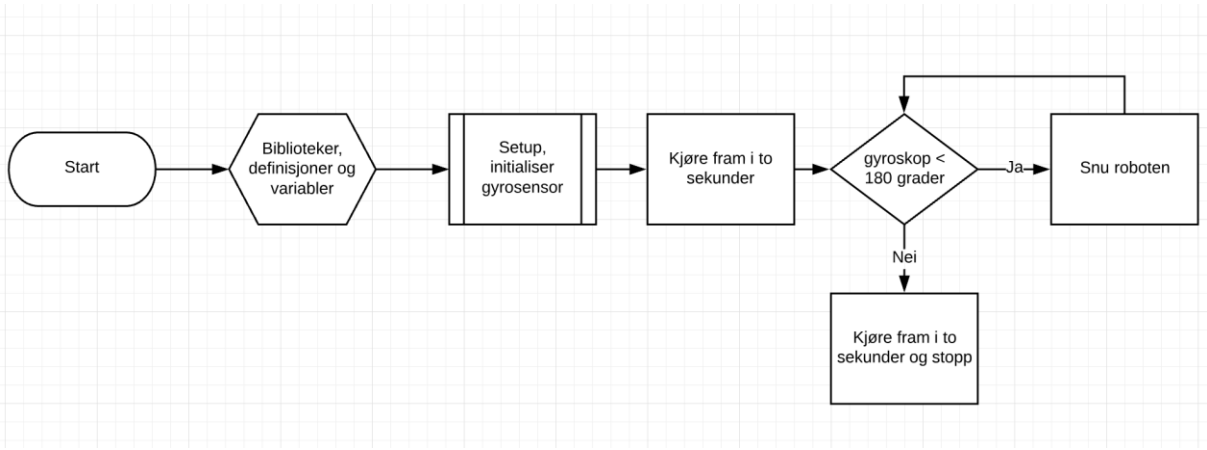


*Figur 10: Flytdiagram til roboten som skal kjøre i en sirkel, venstre**

Kjøre frem og tilbake

Tredje funksjonalitet til roboten var å få den til å kunne kjøre frem, ta en 180-graders vending, og kjøre tilbake til startpunktet. Løsningen var også veldig rett frem, ettersom Zumoen allerede hadde brukt en gyrosensor i en tidligere deloppgave. For å få den til å kjøre rett frem, måtte begge motorene ha samme hastighet like lenge. Deretter ble motorene innstilt slik at den tok en høyresving helt til gyrosensoren hadde målt en 180-graders endring. Dette ble gjort med en

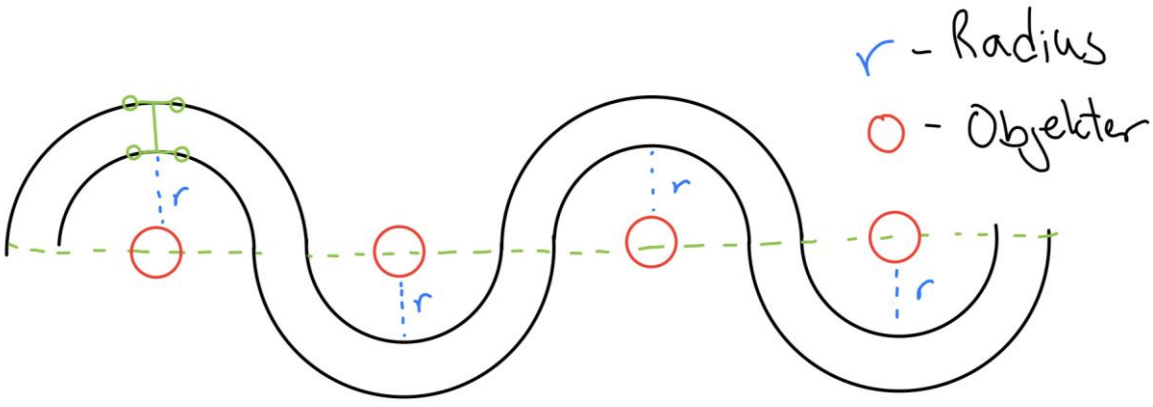
while-løkke. Til slutt skulle den kjøre like langt tilbake, med samme motorhastighet på begge, like lenge. Dette er flytdiagrammet til koden for å kjøre frem og tilbake:



Figur 11: Flytdiagram til roboten som skal frem og tilbake

Kjøre slalåm

Siste mønsteret var å få Zumoen til å kjøre slalåm mellom kjegler. Ettersom avstanden mellom hindrene allerede var kjent, ble det i teorien lett å adaptere kjøremåten fra sirkelkjøring til denne oppgaven. Tankegangen bak fremgangsmåten var å få roboten til å kjøre i halvsirkler med en radius hvor roboten ikke krasjet i kjeglene. Halvsirkelen skulle speilvendes etter hver gang den ble utført, som til slutt ville føre til at Zumoen kjørte sikk-sakk mellom gjenstandene, slik som illustrasjonen vist under:

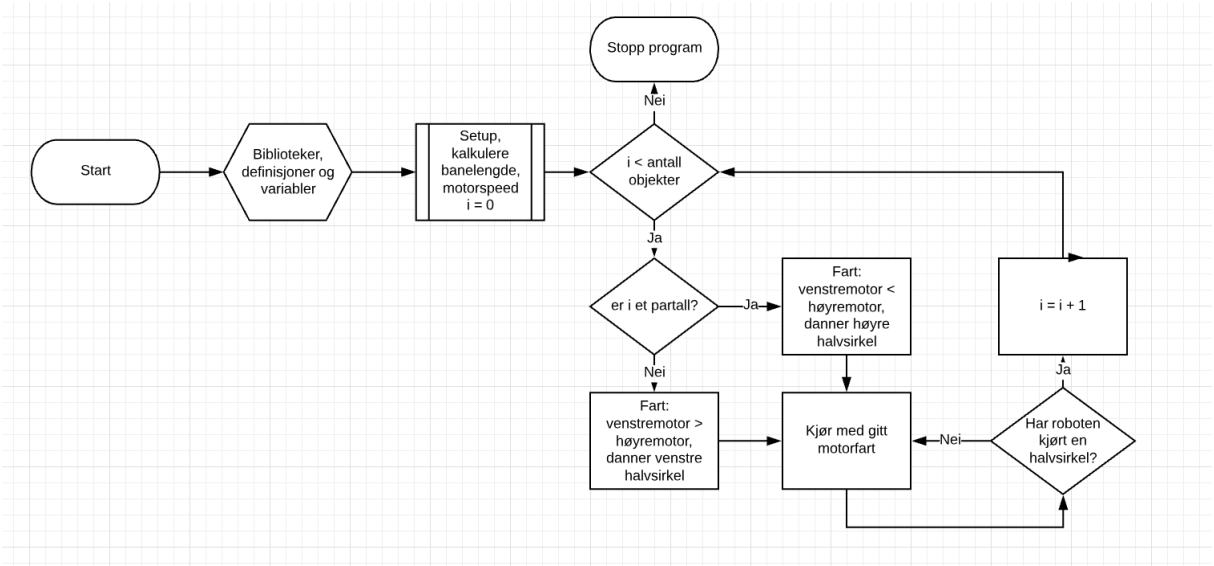


Figur 12: Tegning av banen til roboten i en sikk-sakk-bevegelse

Sammenlignet med sirkelkoden tidligere, skulle roboten kun kjøre i en halv sirkel. Sirkelbanen til den innerste motoren ble derfor π*r, mens sirkelbanen til den ytterste motoren ble π(r+9).

Likeledes med sirkelkoden, ble det brukt Encoders til å finne ut når roboten hadde kjørt en halv sirkel.

For å vite hvor langt roboten skulle kjøre og hvor mange ganger den skulle repetere seg selv, måtte koden ha med den kjente avstanden fra hinderets sentrum til roboten, og hvor mange objekter som var til å komme. Planen var å kjøre en for-løkke så mange ganger som det var av hindre, slik at den kom seg forbi alle. Motorene måtte også på et vis speilvende verdiene sine annen hver gang etter at den hadde fullført en halv-sirkel. Dette kunne man gjøre ved å sjekke om omløpstallet i for-løkka var et partall eller odde, noe som veksles annenhver gang ettersom man teller oppover. Ved partall kunne man dermed si at roboten skulle lage en halvsirkel på sin høyre side, og ved oddetall skulle den lage en halvsirkel på sin venstre side. Dette var fremgangsmåten til å kjøre slalåm mellom kjeglene. Flytdiagram til funksjonen som kjører slalåm ser slik:

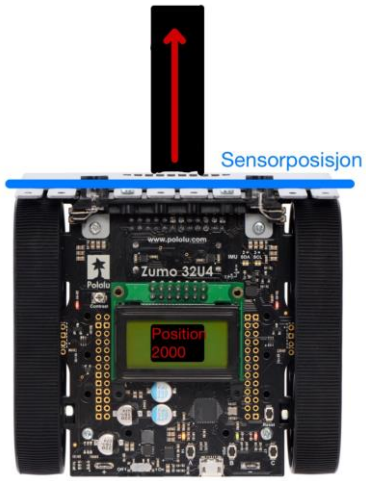


Figur 13: Flytdiagram til roboten som skal kjøre i en slalåmbevegelse

Linjekjøring

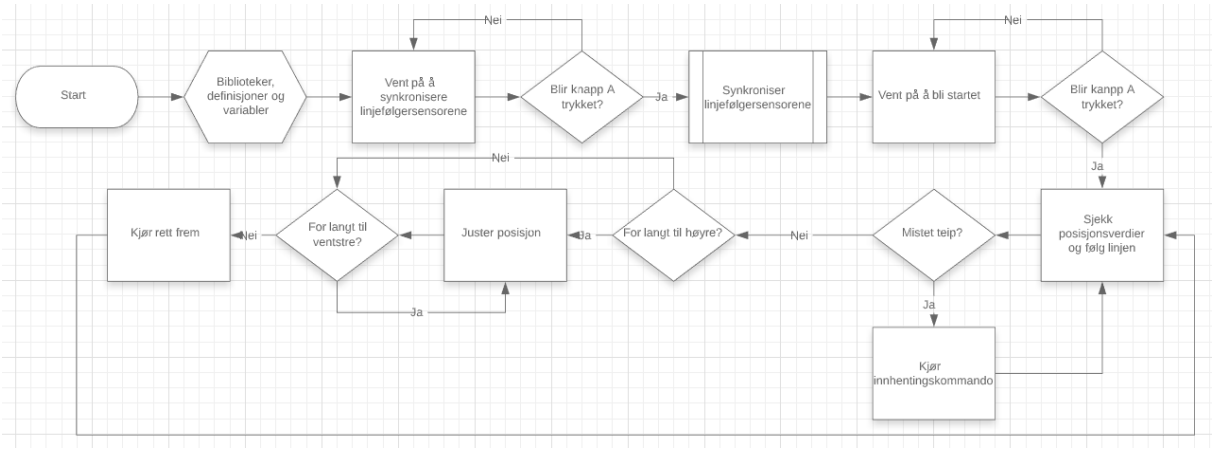
Den neste funksjonaliteten til Zumoen var linjekjøring. Her skulle den følge en bane markert med svart elektrikertape. Det skulle produseres to ulike løsninger til denne oppgaven: en kode som benyttet seg av "hardkodet" linjefølging og en kode som benyttet seg av en PID-regulator. Begge versjonene bygget på samme grunnprinsipp, nemlig å bruke lysforskjellen mellom bakke med teip, og bakke uten teip for å kunne navigere bilen. For å få denne informasjonen, ble «line-follower» sensorene benyttet. Zumoen skulle også kunne "overkomme" ulike utfordringer langs veien.

De funksjonene som følger med biblioteket til Zumoen inneholder en rekke nyttige funksjoner, der en av dem gir ut et tall mellom 0 og 4000, der 2000 forteller at Zumoen befinner seg midt over teipveien. Dersom verdien er 0, befinner den seg for langt til høyre, men er den 4000, for langt til venstre.



Figur 14: Posisjonsverdien når Zumoen befinner seg midt over teipbanen

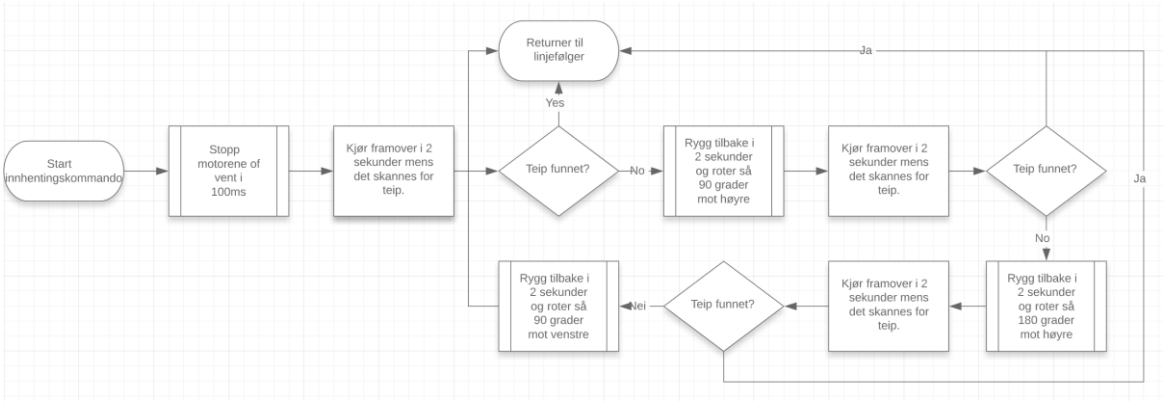
Planen var at Zumoen skulle kontinuerlig sjekke hvilken posisjon den hadde i forhold til teipen, og dersom den avvek innenfor en gitt terskel, skulle rette seg opp.



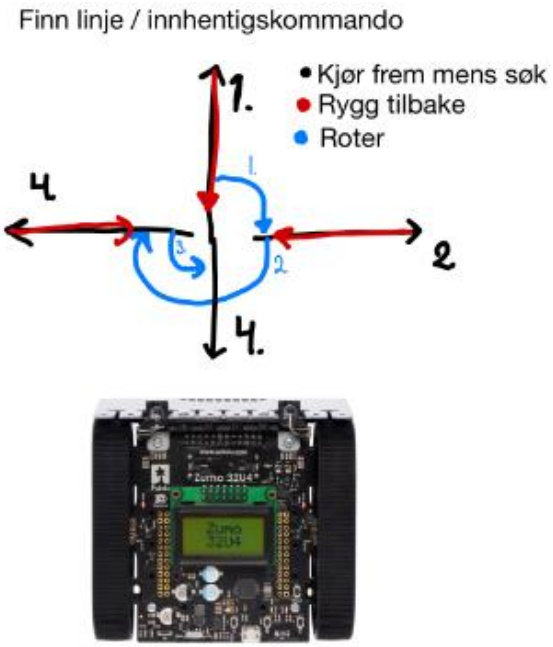
Figur 15: Flytdiagram til Zumoen når den skal bruke "hardkodet" linjefølger

Hvis Zumoen mistet teipen helt, skulle den stoppe opp, kjøre litt framover og samtidig sjekke for teip. Hvis den ikke fant noen, skulle den rygge tilbake til der den stoppet, rotere 90 grader, og repetere samme prosedyre på nytt. Hvis den heller ikke her fant noe, roter 180 grader, sjekk

på ny, og hvis den til slutt ikke fant noe her, roter de siste 90 gradene slik at den kan følge samme vei tilbake.



Figur 16: Flytdiagram til innhentingskommandoen



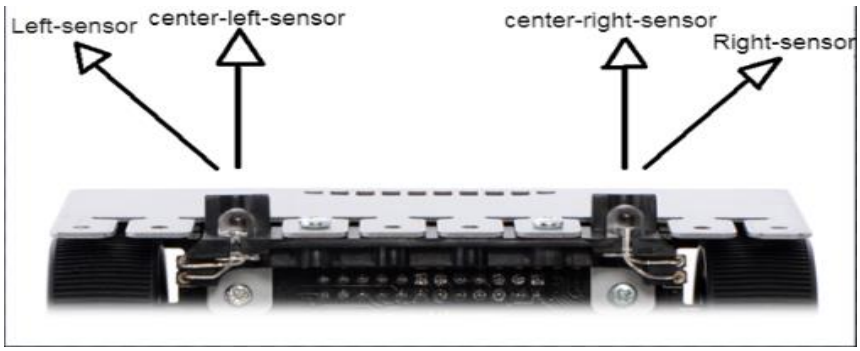
Figur 17: Visualisering av innhentingskommandoen

Når det kom til de to løsningsmetodene var planen å gjøre slik at mesteparten av koden kunne brukes i begge versjonene. Når Zumoen kun skal følge linjen, kan enten en funksjon som kun er «hardkodede» instrukser, eller en funksjon som bruker PID-regulering, kalles. Ved å gjøre dette kunne man ha to like koder der grunnfunksjonaliteten kunne enkelt byttes ut, mens de mer avanserte funksjonene som tar seg av «utfordringene» i løypen kun trenger å kodes/konstrueres en gang.

Obstacle avoidance

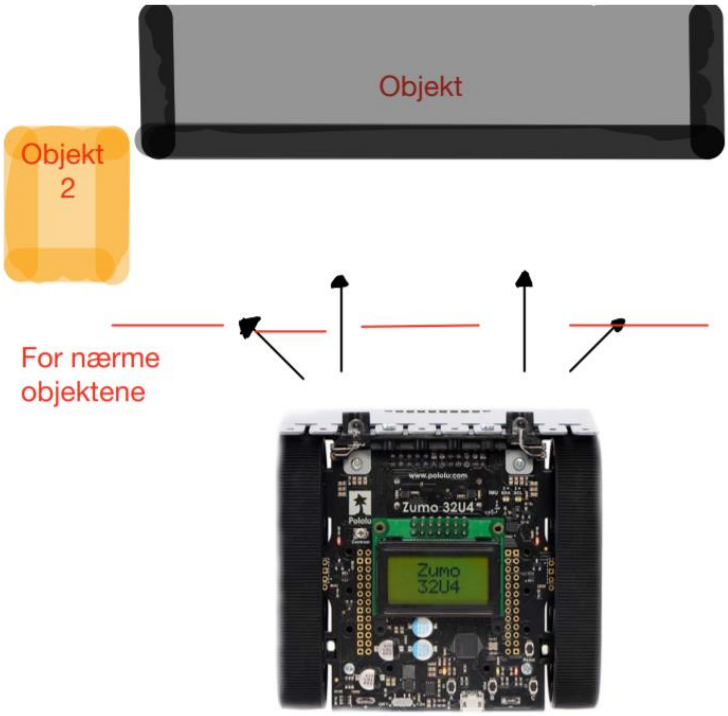
Siste funksjonaliteten til Zumoen er Obstiacal Avoidance. For dette prosjektet ble de ulike sensorene i Zumoen anvendt å få til denne funksjonaliteten. Hensikten med dette prosjektet var å gi Zumoen egenskapen til å kunne kjøre fritt og dukke unna gjenstander som er i veien. Denne funksjonaliteten styres hovedsakelig av proximity sensoren og gyroskopet. Prosjektet blir styrt av en rekke funksjoner som avgjør hvordan Zumoen skal kjøre rund i omgivelsen den er plassert i. Prosjektet vil bestå av fire funksjoner: kjør fram, til venstre, til høyre og reverser. Disse funksjonene ville bli kjørt ut ifra hva proximity sensoren leser og vil bli styrt av gyroskopet.

Proximity sensoren leser av avstanden mellom Zumoen og tilfeldige kollisjonsobjekter. Dersom objektet er nærme nok vil gyroskopet og proximity sensoren samarbeide og finne en vei for å dukke unna objektet. Proximity sensoren leser av verdier fra fire punkter:

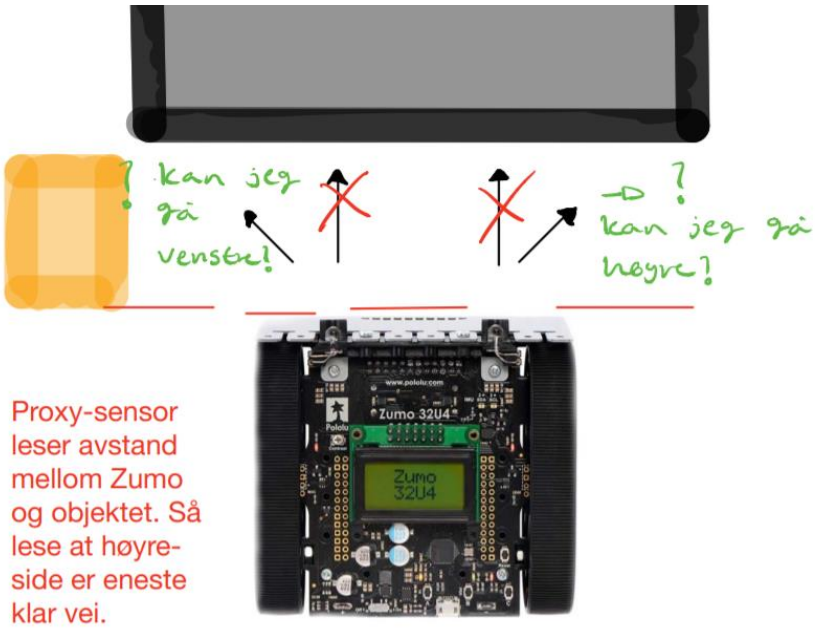


Figur 18: Illustrasjon på hvor proximity-sensorene måler avstand

Maksverdien for hver av de fire punktene er 6 som vil indikere at et objekt er veldig nærme Zumoen, og minimum 0 som indikerer på at objektet langt fra Zumoen. Dersom sentralsensorene (center left/right) får en verdi på fem eller seks, vil Zumoen bruke «left» - og «right» sensorene til å finne en klar vei for Zumoen å kjøre gjennom. Bildene nedenfor illustrer prosjektets funksjonalitet.

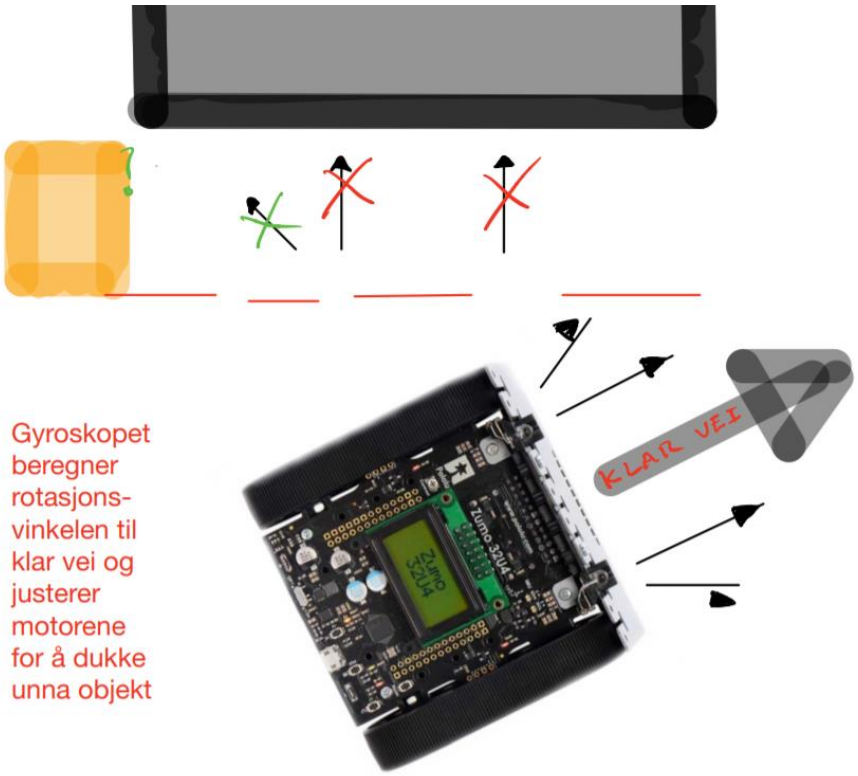


Figur 19: Zumoen kjører mot et kollisjonsobjekt



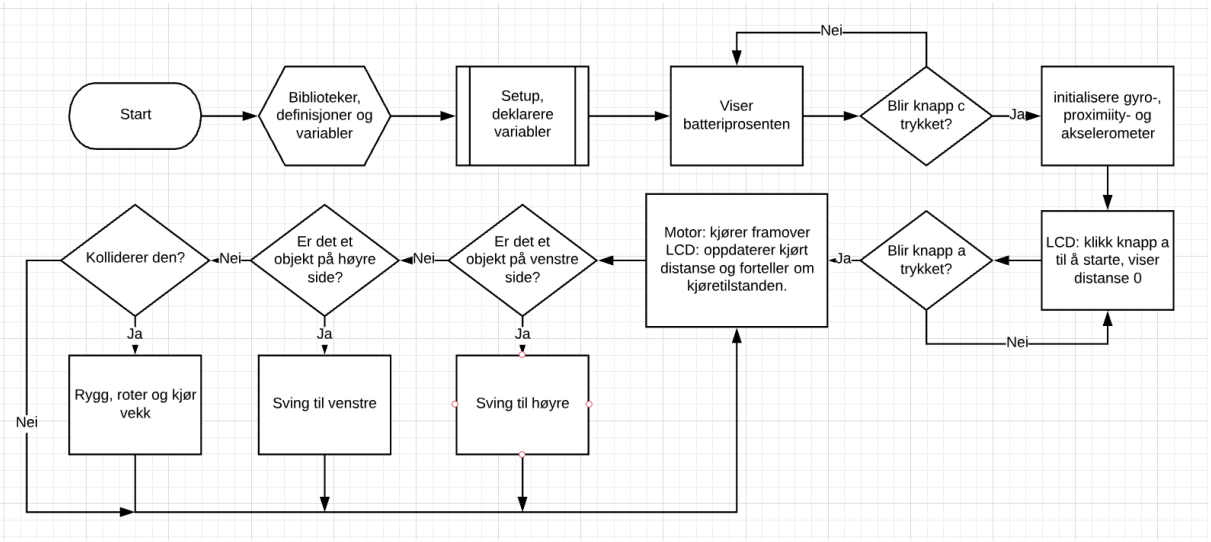
Figur 20: Proximity sensoren leser av at et objekt er for nærme og gyroskopet beregner antall grader den skal rotere for å kjøre inn i en klar vei

Gyroskopet skal brukes for å beregne hvor mange grader Zumoen skal rotere for at den kjør fremover inn mot den klare veien.



Figur 21: Zumoen roteres og kjører videre i en klar vei

Som pynt til dette prosjektet ble Encodere benyttet for å beregne total distanse Zumoen kjører som blir kontinuerlig fremstilt på en LCD-display. Samtidig ble funksjonen 'readBatteryMillivolts()' anvendt til å gi forbrukeren en oversikt av Zumoens batteriprosent. Dette blir også fremstilt på LCD-displayet. Nedenfor er det et bilde av flytdiagrammet til programmet:



Figur 22: Flytdiagram til Obstacle avoidance prosjektet

1.6 BEARBEIDING OG RESULTATER

Meny

Under utviklingen av menyen gikk alt som planlagt, hvor knappene A og C ble brukt til å bla gjennom alle funksjonalitetene, og knapp B ble brukt til å initialere valgt modus. Gruppen valgte ikke å legge til en meny for linjekjøring og Obstacle Avoidance, ettersom koden ble for lang, men som nevnt i metodedelen var det lett å legge til flere kjøremønstre.

Kjøre i firkant

Under programmeringen av første funksjonalitet, oppstod det en liten utfordring. Koden var satt opp slik at motorfarten til venstre og høyre var like, påfulgt med et delay på tre sekunder. Med andre ord, skulle roboten kjøre rett frem i tre skunder. Problemet som oppstod var roboten kjørte litt skeivt, selv om den i teorien burde ha kjørt strakt fremover. Dette avviket kan skyldes av slitasje i motoren, skeivhet i hjulene og liknende. Problemet som oppstod var pregende for alle oppgaver framover, og resultatet medførte til at roboten havnet noen få cm unna startpunktet. Grunnet av noe som antageligvis bare noe feil med roboten vår, har gruppen valgt å se bort i fra dette problemet i stor grad. Ellers fungerte 90-graders svingene bra med gyrosensoren.

Kjøre i sirkel

Koden til sirkelbanen inneholdt en del beregninger og var mer komplekst enn den forrige funksjonaliteten. Ettersom koden ble prøvekjørt med beregningene fra metodedelen, var det

tydelig at Zumoen var langt ifra å kjøre i en hel sirkel. Observasjonene var at roboten stoppet altfor tidlig, og første tanke som oppstod var at det muligens var noen beregningsfeil under konverteringen av Encoders til reell distanse. Det så ikke ut til å være tilfellet, og det ble gjennomført flere testkjøringer for å finne ut hva problemårsaken var.

En observasjon som etterhvert ble gjort var at radiusen til sirkelbanen som ble utført i praksis, var altfor stor, sammenlignet med radiusen som var fastsatt i koden. Da var det tydelig at forholdet mellom farten til motorene ikke samsvarte, ettersom beregningene med Encoders så ut til å være korrekte. Roboten stoppet med andre ord til rett tid, men grunnet av feil forhold mellom motorene, kjørte den i en for stor sirkelbane. Det kalkulerte forholdet mellom venstre- og høyremotor burde i teorien være $\frac{r}{r+b}$, slik som henvist til teoridelen. I dette tilfellet var bredden på bilen "b" på 9 cm, i tillegg til at den vilkårlige radiusen også ble satt som 9 cm. Ettersom standardfarten i koden var satt til å være 100, burde den reduserte farten til den andre motoren være kalkulert til å bli $100 * \frac{9}{9+9} = 50$. Med de fartene ville roboten i teorien danne en sirkelbane med 9 cm radius, noe som ikke stemte.

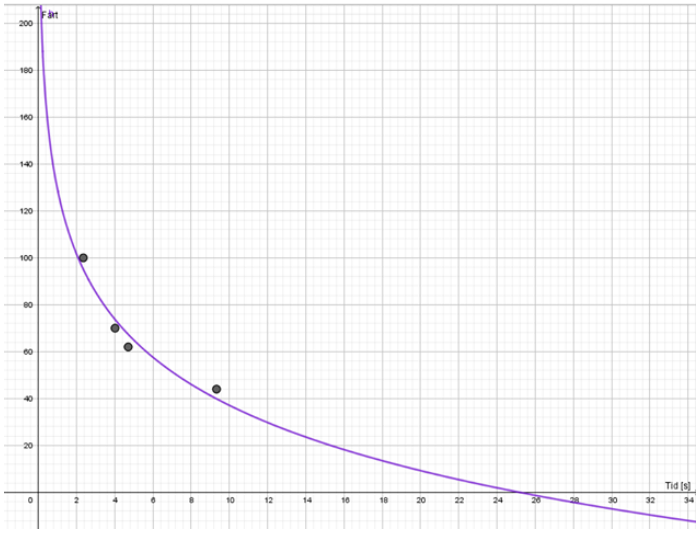
Dette problemet oppstod grunnet en antagelse av at motorfartens verdier var proporsjonale, ergo at forholdet mellom motorene fulgte en rasjonal graf. Det ble derfor gjort eksterne målinger på motorene for å fastslå eller avvikle om motorfartens verdier fulgte en proporsjonalitet. Testen gikk ut på hvor lang tid roboten tok for å kjøre 30 cm for ulike fartsverdier. Resultatet ble slik:

Fart	**Måling 1**	**Måling 2**	**Måling 3**	**Gjennomsnitt**
100	2,36s	2,38s	2,34s	2,36s
62	4,79s	4,68s	4,66s	4,71s
71	4,05s	3,98s	4,03s	4,02s
44	9,34s	9,41s	9,24s	9,33s

Figur 23: Tabell på tiden det tar for å kjøre 30cm med ulik hastighet

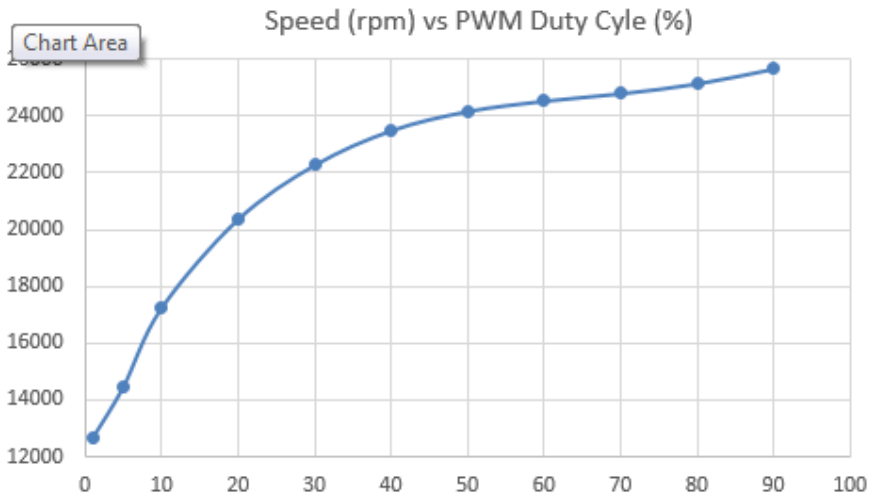
Tanken bak akkurat de utvalgte verdiene som ble testet var for å finne ut hvilken fart som ga dobbelt som lang tid som med 100 i motorfart. Denne farten så ut til å være på 62, ettersom forholdet mellom gjennomsnittstiden til fartene 62 og 100 ble $\frac{2,36s}{4,71s} \approx 0,5$. Med den informasjonen var det dermed mulig å spekulere at 0.62 var forholdet mellom venstre- og

høyremotor. For å bekrefte denne påstanden ble en annen tilfeldig motorfart testet for å se om denne sammenhengen samsvarte. 71 i motorfart ble valgt, og da burde 71*0.62 ≈ 44 i motorfart gi dobbelt så lang tid på 30 cm kjørebane. Resultatene viste derimot noe annet. Forholdet mellom fartene til 71 og 44 ble så lite som $\frac{4{,}02s}{9{,}33s} \approx 0{,}43$. Det ble deretter laget en regresjonsanalyse av punktene:



Figur 24: Regresjonsanalyse av sammenhengen mellom fart og tid på en konstant strekning

Grafen visualiserte sammenhengen mellom hvor lang tid det tok å kjøre 30 cm med ulik fart. Informasjonen analysen av punktene ga var at tiden (x-aksen) tydelig ikke var lineær med motorfarten (y-aksen), men i stedet hadde tidsgrafen noen fellestrekk med en omvendt proporsjonal graf. Om grafen hadde fulgt en linearitet, ville det med andre ord bety at forholdet mellom motorfartene var en bestemt verdi, noe som tidligere ble bekreftet at ikke stemte. Det var dermed også mulig å karakterisere sammenhengen mellom farten og PWN-syklusen til motorene, i tillegg til inngangsspenningen til motorene, som følge av at det var en korrelasjon mellom dem. Grafen så noe slik ut:



Figur 25: Sammenhengen mellom farten og PWM syklus [19]

PWM ble brukt til å generere pulser med ulike sykluser, i den hensikt om å kunne kontrollere et pådrag for å blant annet oppnå en ønsket motorfart. Dette ble gjort med å kontrollere terminalspenningen til motorene. Grafen i figur 15 hadde logaritmiske egenskaper, og ikke en lineær egenskap. Det var dermed avklart at motorens verdier ikke fulgte en proporsjonalitet.

Av alle de tidligere grunnene var det avklart at motorens verdier ikke fulgte en proporsjonalitet. Til tross for at det ikke var en tydelig sammenheng mellom motorfartens verdi, medførte det til at forholdet mellom motorene måtte inneholde en ekstern konstant for at motorfartene skulle samstemme. Ellers kjørte sirkelkoden slik som ønsket.

Kjøre frem og tilbake

Metoden for tredje funksjonalitet var noenlunde lik første i kompleksitet, ettersom det kun var å definere hvor mye gyrosensoren skulle bevege seg før og etter at Zumoen hadde kjørt. Dette gikk feilfritt.

Kjøre slalåm mellom kjegler

Siste modul i autonom kjøring var å kjøre mellom kjegler. I koden var det viktig å holde styr på hvilke variabelnavn inneholdt hvilken data, av den grunn at alle verdier måtte bli byttet om annen hver gang for å kjøre slik som vist i planleggingsfasen. Ettersom problemet med motorforholdet allerede var løst, burde koden i teorien ha fungert som den skulle.

Koden ble testet hvor det ble satt inn at Zumoen skulle kjøre forbi tre objekter, og radiusen mellom dem var 18 cm. En observasjon var at første og tredje halvsirkel var ulik halvsirkel nummer to. Zumoen kjørte med andre ord i lik halvsirkel når den vendte mot høyre første og tredje gang, men når den gikk mot venstre andre gangen, overkjørte den slik at Zumoen havnet på feil plass til slutt. Det som også kunne bli lagt merke til var at radiusen til halvsirkel nummer to var mindre en første og tredje halvsirkel. Det ble da gjort en serial.print* av encoder-verdiene til begge motorene gjennom hele utkjøringen, slik at det blant annet var mulig å vite om Zumoen stoppet til rett tid. Følgende data fra vedlegget var nyttig for videre løsning av problemet:

```
    [...]
[1]  12:20:46.217 ->
[2]  12:20:46.217 -> 6429
[3]  12:20:46.217 -> 6429
[4]  12:20:46.217 -> 2785
[5]  12:20:46.217 -> 4286
[6]  12:20:46.217 ->
[7]  12:20:46.217 -> 1
[8]  12:20:46.217 -> 4286
[9]  12:20:46.217 -> 0
[10] 12:20:46.217 -> 6429
[11] 12:20:46.217 ->
    [...]
```

Figur 28: Encoder-verdiene etter første omvending

```
    [...]
[1]  12:20:51.415 ->
[2]  12:20:51.415 -> 2351
[3]  12:20:51.415 -> 4286
[4]  12:20:51.415 -> 6429
[5]  12:20:51.415 -> 6429
[6]  12:20:51.415 ->
[7]  12:20:51.415 -> 0
[8]  12:20:51.415 -> 6429
[9]  12:20:51.415 -> 0
[10] 12:20:51.415 -> 4286
[11] 12:20:51.415 ->
    [...]
```

Figur 26: Encoder-verdiene etter andre omvending

```
    [...]
[1]  12:20:56.959 ->
[2]  12:20:56.959 -> 6428
[3]  12:20:56.959 -> 6429
[4]  12:20:56.959 -> 2742
[5]  12:20:56.959 -> 4286
[6]  12:20:56.959 ->
[7]  12:20:56.959 -> 6429
[8]  12:20:56.959 -> 6429
[9]  12:20:56.959 -> 2742
[10] 12:20:56.959 -> 4286
[11] 12:20:56.959 ->
```

Figur 27: Encoder-verdiene da roboten stoppet

Ved å studere figurene ovenfor var dataen kategorisert i firer-grupper, adskilt med mellomrom. Disse verdiene var strukturert slik:

- Første linje: Nåværende Encoder-verdi til venstre motor
- Andre linje: Ønsket Encoder-verdi til venstre motor for å kjøre i en halvsirkel
- Tredje linje: Nåværende Encoder-verdi til høyre motor
- Fjerde linje: Ønsket Encoder-verdi til høyre motor for å kjøre i en halvsirkel

Noe som kunne bli lagt merke til var at ikke begge Encoder-verdiene oppnådde ønsket resultat, før den koden byttet om på verdiene for å speilvende halvsirkelen. Årsaken til dette var at den ene motoren måtte bli multiplisert med en ekstern konstant formulert i sirkeloppgaven, men dette hadde ingen påvirkning på kjøresystemet ettersom det kun var nødvendig at en av motorene nådde ønsket resultat.

Dataen som blir framstilt, fortalte at den Encoder-verdien til svakeste motoren i alle tre tilfeller av omvendingen ikke var lik, noe den burde ha vært som følge av at de skulle kjøre i like store halvsirkler. I første og tredje halvsirkel nådde verdien rundt 2760 (figur 16 - linje 4 og figur 18 - linje 9). Encoder-verdien i halvsirkel nummer 2 rakk å nå 2351 før omstillingen (figur 17 - linje 2). Til tross for at disse verdiene i teorien burde være like, var det mulig å fastslå at forholdet mellom venstre- og høyre motor ikke var det samme som forholdet mellom høyre- og venstre motor, påtrykt med samme spenning. Med andre ord betydde det at en konstant påtrykt spenningsverdi ville gi to forskjellige verdier for farten til venstre motor og høyre motor. Løsningen var at konstantene for å regne ut riktig forhold i en høyresving, var ulik multiplikasjonskonstanten til å regne ut forholdet i en venstresving. Utenom det kjørte Zumoen som planlagt.

Linjekjøring

Hensikten med denne delen var som tidligere nevnt å få Zumoen til å følge en bane av sort elektrikertape. Som definert av oppgaveteksten skal denne banen bestå av: «skarpe og slakke høyre- og venstresvinger, rettvinklede høyre og venstresvinger, et veistykke med manglende teip, og et t-kryss der veien rett fram er en blindvei, mens veien til venstre er riktig vei». Dette skulle altså forsøkes å løses ved to tidligere nevnt metoder. I utgangspunktet greide Zumoen å følge en løype med både 90-graders- og vanlige svinger, og når den traff en tom veibit, ville den bare snu og følge veien tilbake.

Det første problemet vi støtte på var å kunne komme seg over veistykket med manglende teip. Ved hardkodeversjonen og PD-regulerings-versjonen, vil IR-sensorene lese at Zumoen er kommet av linjen. Ved å da stoppe bilen, og kjøre funksjonen som scanner etter ny teip, vil bilen finne ny teip, og fortsette banen (innhentingskommando i flytdiagram). Problemet med denne løsningen, er at idet Zumoen kommer til en veldig krapp (evt 90-graders) sving, vil sensorene lese av at den er av teipen, og kjøre innhentingskommandoen. Mesteparten av tiden resulterte dette i at Zumoen kjørte seg av banen. Vi forsøkte å løse dette problemet ved å forandre på parameterne som ba den kjøre funksjonen, men det fungerte dårlig. Et annet forsøk gikk ut på å «hardkode» denne innhentingskommando til å alltid først sjekke om Zumoen befinner seg i et hjørne, ved å rotere 90 grader hver vei og se på sensorverdiene. Hvis den ikke

befant seg i et hjørne, skulle den fortsette med å kjøre innhentingskommandoen. Dette viste seg å bli uoversiktlig og lite vennlig å kode videre på for andre. Det fungerte for så vidt heller ikke.

Dette førte til av vi skrapet hele ideen med å korrigere først når sensorene sier vi er av banen. Istedenfor begynte vi å bruke avstanden vi var ifra ønsket posisjon som mål for om Zumoen befant seg på linjen. Siden funksjonen som ble brukt i koden returnerer et tall mellom 0 og 4000, der 2000 indikerer at Zumoen befinner seg midt på linjen, kan man heller få en indikasjon på hvor mye man må korrigere. Avstandsfeilen fant vi ved å ta 2000 og subtrahere den faktiske posisjonen. Dette tallet forteller hvor mye feil, og hvor mye som må rettes opp. Er Zumoen litt skjev, vil dette tallet kanskje være 400, som indikerer at det må foretas en liten korreksjon. Kommer man til en skarp sving, vil dette tallet være mye større (for eksempel 700), som indikerer at en mye større korreksjon må foretas. Slik så hardkoden vår for å justere på motorfarten:

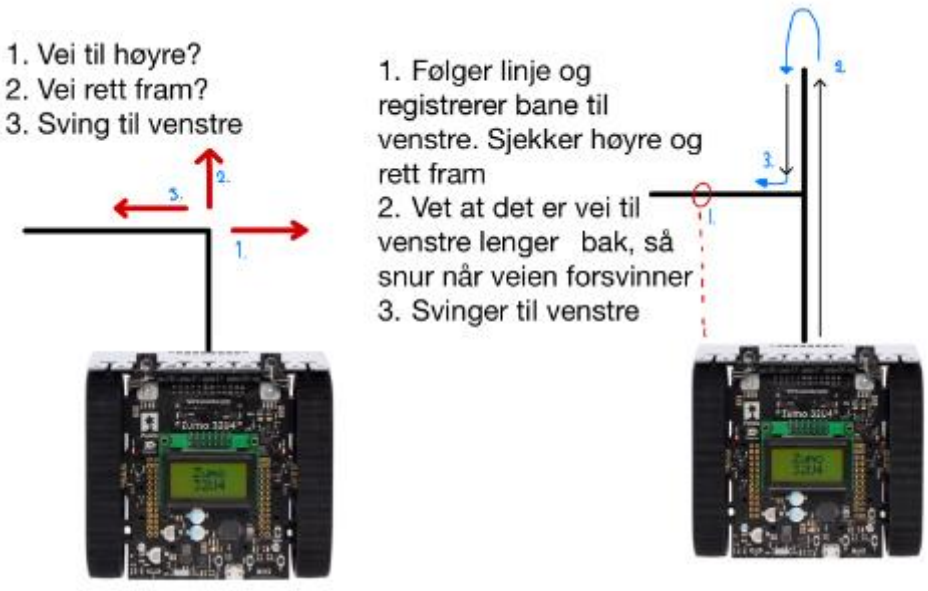
```
// Small ajustments
  if (error < smallAdjustment && error > -smallAdjustment) {
    if (error < -100 && error > -smallAdjustment) {
      motors.setSpeeds(10, defaultSpeed);
    } else if (error > 100 && error < smallAdjustment) {
      motors.setSpeeds(defaultSpeed, 10);
    } else if (error <= 100 && error >= -100) {
      motors.setSpeeds(defaultSpeed, defaultSpeed
    }

[…]
```

Denne fremgangsmetoden fungerer i praksis på nesten samme måte som den første ideen, men når de ulike hindringene skal takles, gir den mye større frihet til å foreta korreksjoner.
Når Zumoen kommer til en 90-graders venstresving, vil de tre IR-sensorene i midten, og IR-sensoren helt til venstre lese av verdier. Dette gjør slik at avvikstallet øker raskt.

Siden det skal være et kryss ett eller annet sted i banen, og vi vet at venstresvingen skal først tas etter at vi har kjørt blindveien, vil Zumoen først sjekke om det er vei til høyre. Er det ikke det, vil den sjekke rett fram, og er det ikke noe der heller, vil den svinge til venstre. Hadde den funnet en vei til høyre, ville den tatt den, og hadde den funnet en vei rett fram, hadde den tatt den. Kjører den rett fram, og det er registrert en venstresving, vet den at den vil komme til en blindvei, og når veien slutter, roterer den bare 180 grader, kjører tilbake til krysset, og svinger

til høyre. På denne måten vet den også at hvis veien plutselig stopper opp, kan den bare kjøre videre, fordi det er ikke noen «utfordringer» som krever at den skal gjøre noen store handlinger utenfor teipbanen.



Figur 29: Zumoens oppførsel i» 90 graders sving»- og "veikryss"-utfordringen

I motsetning til den originale ideen, fungerte denne fremgangsmetoden betraktelig bedre, og koden ble mye mer oversiktlig, lett å forstå, og «vedlikeholds»- og forbedringsvennlig. Det var fremdeles et problem, og det var at vi benyttet oss av delay() for å kontrollerer rotasjonene. Siden motorene leverer forskjellig effekt ut ifra bla, batteristyrke, vil ikke Zumoen rotere nøyaktig 90- eller 180 grader hver gang. Dette bydde også på en del hodebry.
Vi løste det med å implementere gyrosensoren. Ved å benytte samme prinsipp som i «firkantkjøring» koden, fikk vi endelig presise rotasjoner og Zumoen greide å følge teipbanen.

Når det kommer til «PID-regulator»-versjonen av linjefølgerkoden, er det bygd opp på akkurat samme måte som «hardkode»-versjonen. Er det et lite avvik, må en liten korreksjon utføres, og kommer Zumoen til en sving, vil den kjøre den samme sjekken som beskrevet i Figur 29 og rotere ved hjelp av gyrosensoren.

Selve «PID» koden er laget som en egen funksjon, som blir kalt hver gang programmet looper, så lenge korrigeringsavviket er lite. Denne koden hentet vi inspirasjon fra eksempelet som følger med biblioteket til Zumoen. Vi forandret litt på konstantene, slik at reguleringspåtrykket blir tilpasset farten og svingene vi benytter oss av. Dette resulterte i at Zumoen følger linje mye bedre enn når den er «hardkodet».

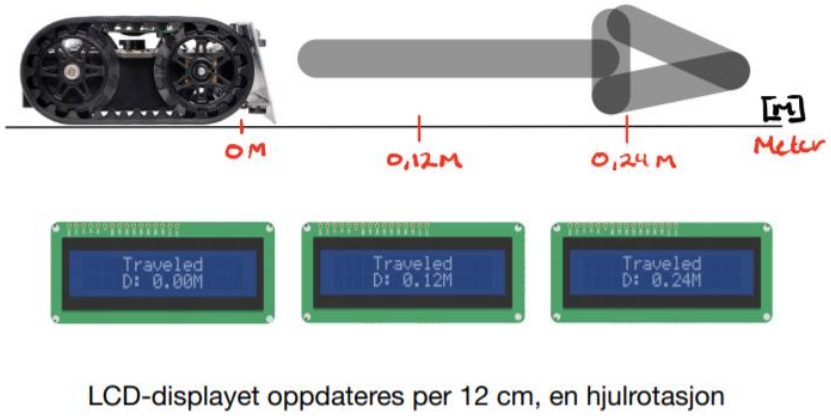
Obstacle avoidance

Hensikten med dette prosjektet var å programmere et «obstacle avoidance» - system til Zumoen. Dette prosjektet var vellykket og prosjektets funksjonalitet fungerte som det skulle, men før programmet ble 100 prosent funksjonell var det tre hovedproblemer som måtte bli løst. Det første problemet var 'hjernen' til prosjektet, som besto av en rekke av if-setninger. Det ble mye komplikasjoner med beregning av hvor sensitiv prosjektet skulle være til sensorverdiene. Problemet med prosjektet var at Zumoen begynte å spinne rundt i sirkler og aldri kjørte rett frem. Denne feilen skyldes av at prosjektet var stilt til at Zumoen skulle tro at det var objekter i veien da proximity-sensorens verdier var på det laveste som: en eller to. Problemet ble løst ved å sette en grense for at Zumoen skulle dukke unna objekter kun da proximity – sensoren fikk verdi større 4. Denne endringen fikk Zumoen til å kjøre frem, sving og dukke unna objekter da den faktisk skulle.

Det andre problemet var å finne en måte å regne ut den totale distansen Zumoen kjørte per test ved hjelp av Encodere. Det tok tid å skjønne hvordan en Encoder funket, hva den målte og hvordan det målte verdier. Zumo manualen [20] ga en bedre forståelse av Encoderens funksjonalitet. Fra teoretisk grunnlag nevnt tidligere ble det regnet ut at en omdreining tilsvarer en puls på 910 tellinger og en lengde på 0.12 meter. Mer om dette finnes i kapitel 1.4. Lengde per omdreining ble regnet slik:

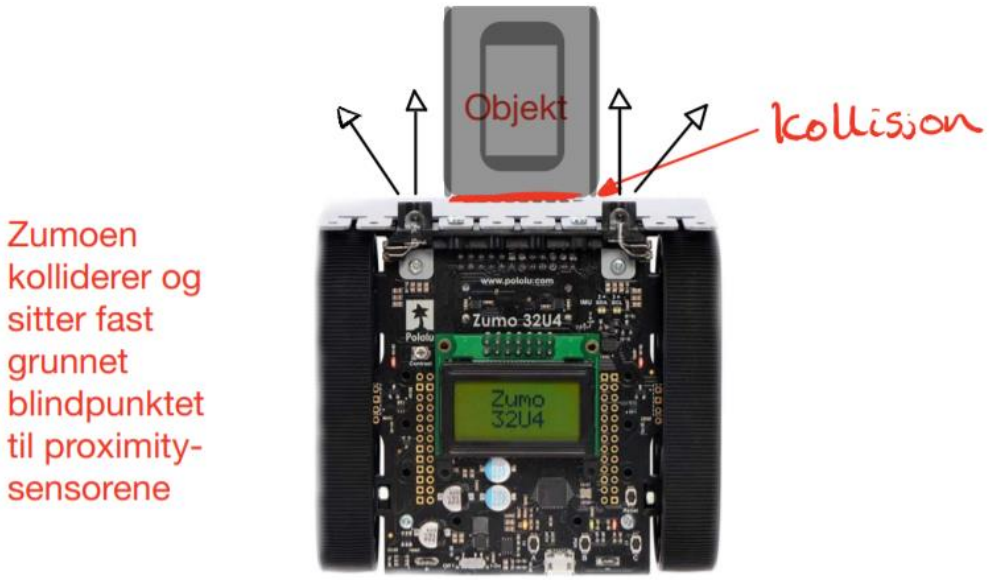
Figur 30: Banen til hjulene i en sving

Med disse verdiene tilgjengelig var det ikke vanskelig å deklarere noen variabler og sette opp en if-setning, som utgjør hele funksjonaliteten til distansemåleren som kontinuerlig oppdateres på LCD-displayet. For å se funksjonaliteten kodet se kilde [29]. Illustrasjon for funksjonaliteten nedenfor:



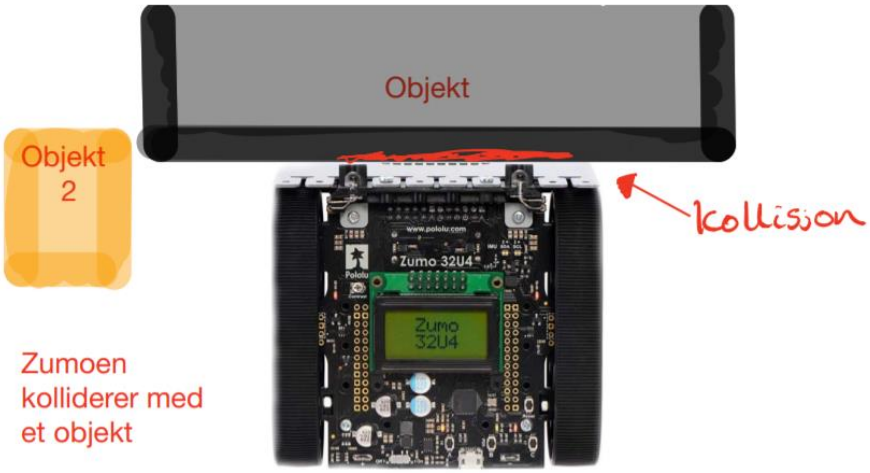
Figur 31 Illustrasjon av distansemåler av Encoders

Det siste problemet var å finne en løsning for blindpunktet som oppstår ved bruk av proximity sensoren. Som nevnt tidligere består proximity sensoren av fire punkter: «left, left-center, right-center og right». Det ble oppdaget under testing av prosjektet at det fantes et blindpunkt mellom senterpunktene. Mellom senterpunktene er det ett mellomrom på 4 cm hvor ingenting blir detektert av proximity sensoren, som da refereres som blindpunktet. Dette problemet førte til at dersom det var et objekt foran Zumoen innenfor blindpunktet ville Zumoen kollidere i objektet, og motorene ville fortsette å presse fremover. Dette førte da til press på Zumoens motorer som ble presset til å kjøre i høy hastighet med mye motstand som kan ha slitt beltene (dekkene) til Zumoen eller ødelagt motorene.

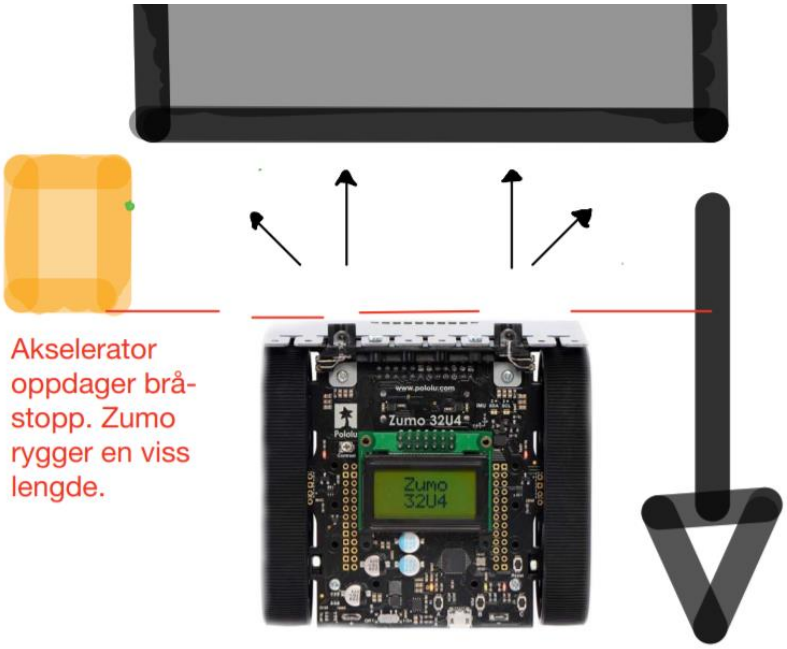


Figur 32: Zumo kollisjon

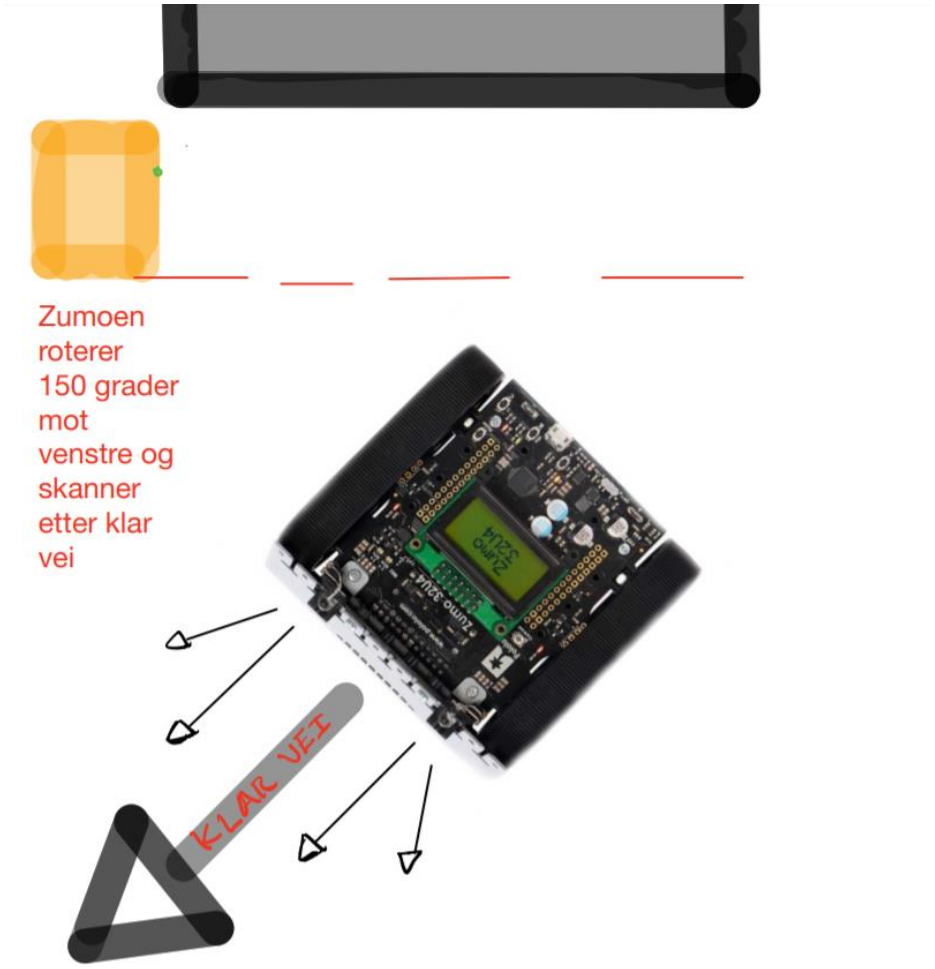
For å løse dette problemet ble akseleratoren anvendt i prosjektet. Som nevnt i terminologien er akseleratoren en tre akses akselerator. For dette prosjektet ble det kun x- og y-akse som ble tatt i bruk. Det ble deklarert tomme x og y arrays som fikk verdier da Zumoen begynte å kjøre. Oppstod det en kollisjon ble arraysene multiplisert og multiplumet ble sammenlignet med en kollisjons-variabel med en verdi på 250 000 000, som gruppen bestemte indikerte et brått stopp. Kollisjons-verdien ble funnet ved å lese av multiplumet til vektorene med Serial.print da Zumoen kolliderte med en hastighet på 250-200 (motors.speed). Dette var standard hastigheten til prosjektet og multiplumet til arraysene ga en verdi som var større kollisjons-variabelen. Dersom multiplumet av arraysene var større en kollisjons-variabelen skjedde følgende: Akseleratoren merker et brått stopp som indikerer en kollisjon. Umiddelbart etter ville Zumoen stoppe, rygge, og rotere 150 grader mot venstre, skanne etter klar vei og fortsette å kjøre videre.



Figur 33: Zumo kolliderer med et objekt, og akseleratoren oppdager et brått stopp.



Figur 34: Zumoen reverserer unner objektet



Figur 35: Zumoen snur, skanner og kjører hvor det er klar vei

2 MODUL – ESP32 SENSORNODE

2.1 SAMMENDRAG

Hensikten med denne delen av rapporten er å beskrive hvordan vi løste oppgaven, og hva vi har gjort på modulen «Sensornode». Vi koblet opp sensorer og diverse komponenter over to brødbrett og videre koblet vi dette opp mot Blynk. Tilkoblingen til Blynk gjorde det mulig å kommunisere mellom Sensornoden (ESP32) og Blynk sin app på mobiltelefonen. Gjennom telefonen kunne man da se sensordataen, styre en servo motor, styre gjennomsnittsfilteret og bestemme hvilken sensor det skulle leses fra. Til slutt ble en enkel webserver opprettet der vi sendte sensordata over WiFi. Vi gjorde noen få ekstra ting utenom oppgavens spesifikasjoner som å hente tid fra en server og vise den på vår egen webserver. Vi la også til noen ekstra komponenter som en knapp og en LCD-skjerm til sensornoden.

2.2 TERMINOLOGI

Mikrokontroller -	En mikrokontroller er en programmerbar prosessor som samtidig inneholder innganger og utganger m.m.
Arduino -	Arduino er en plattform for prototyping av elektronikk basert på program- og maskinvare med åpen kildekode.
ESP32 -	ESP32 er et utviklingsbrett bygget rundt Espressif sin ESP32-WROOM-32D mikrokontroller. Vi bruker den som en Arduino, men den er på mange måter nyere og bedre, blant annet så kommer den med WiFi.
PWM -	Pulse Width Modulation
Blynk -	En IOT plattform som lar deg koble opp en mikrokontroller til mobiltelefonen.
IOT -	«Internet Of Things», et system av ting som er oppkoblet mot internettet.
Sensor -	Et instrument som kan måle fysiske endringer.
I2C -	En kommunikasjonsprotokoll som brukes for å kommunisere mellom chipper, består av en klokkelinje (SCL) og en datalinje (SCL).
Buzzer -	En elektrisk komponent som kan skape lyder.
MQ2 -	En sensor som gir ut et analogt spenningssignal som økes med mengden ugasser i lufta.
Ugass -	Uønsket type gass f.eks. karbongasser, røyk, alkohol, hydrogen og mer.
VL6180x -	En sensor som kan måle avstand og belysningsstyrke
LED -	En diode som kan emittere lys.
LCD -	En elektrisk skjerm som man kan skrive på gjennom en mikrokontroller.
PCA9685 -	En I2C PWM kontroller
Servo -	En elektrisk komponent som kan styre en «arm» med høy presisjon.
Webserver -	En plass man kan sende, motta, vise og lagre informasjon.
http -	Hypertext Transfer Protocol, definerer hvordan meldinger er formattert og sendt.
Polling -	Å hente data fra en server.
Gjennomsnittsfilter -	Regner ut det løpende gjennomsnittet for å filtrere.
Verdensveven -	WWW «World Wide Web», globalt informasjonssystem.

2.3 INNLEDNING – PROBLEMSTILLING

Bakgrunnen for sensornoden er å koble opp tre sensorer til en ESP32 som skal kobles opp mot Blynk. Det skal være kommunikasjon mellom ESP32 og Blynk. Gjennom Blynk skal man kunne lese de tre sensorene sine verdier, velge hvilken sensor det skal leses fra, teste servo og stille på gjennomsnittsfilteret. Det skal også være et alarmelement som går av dersom to eller flere av tre sensorer gir for høye verdier.

Til slutt, som ekstraoppgave, skal dataen visualiseres på en webserver.

2.4 BAKGRUNN - TEORETISK GRUNNLAG

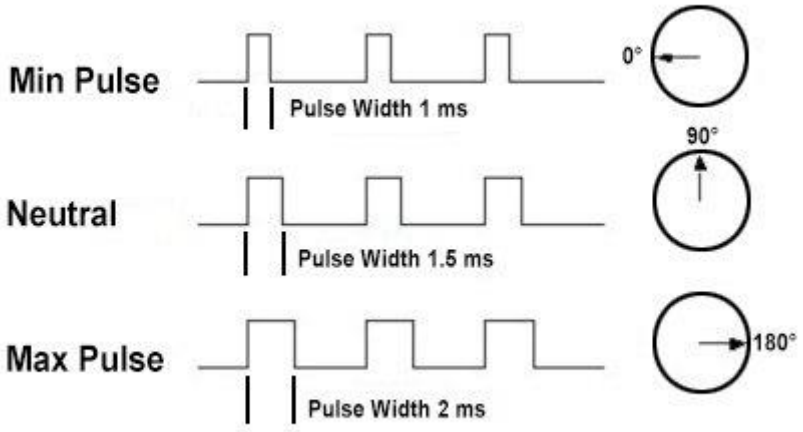
Blynk

Blynk er en IOT plattform som gjør det enkelt å koble en mikrokontroller med tilgang til WiFi til «The World Wide Web», eller verdensveven. Gjennom virtuelle pins kan man sende data fram og tilbake mellom mikrokontrolleren og Blynk på telefonen din.

Servo

En servo er en presisjonsmotor som kan bevege seg 0-180 grader med repeterbar nøyaktighet. For å gå til 180 grader må man sende en spenningspuls med bredde på to millisekund, og for å gå til 0 grader må man sende en spenningspuls med bredde på ett millisekund.

Figur 36: Servomotor



Figur 37: Servopuls

PCA9685

Dette er en 16 kanals I2C PWM driver, den kan altså styre 16 PWM utganger samtidig, mens den tar inn instruks over I2C. Du sender kommandoer som sier hvilken kanal du skal sende PWM på, så sier du frekvensen, når signalet skal gå fra lavt til høyt, også når signalet skal gå tilbake fra høyt til lavt. **Error! Reference source not found.**

Figur 38: Adafruit PCA9685 Servo Driver

Millis vs delay

I programmeringsspråket Arduino er det en innebygd funksjon som gjør at hele programmet stopper opp for et gitt antall millisekunder. Denne funksjonen heter delay(). Det som er kjekt med denne funksjonen er at den er enkel å forstå, og er derfor fin for nybegynnere. Problemet med dette er derimot at hele programmet stopper opp, og ingenting kan skje under denne «delay» tiden. Derfor er det en annen måte man kan få programmet til å vente, men samtidig kjøre kode. Man kan si at man «multitasker» mikroprosessoren. En måte å gjøre dette på et å bruke millis() funksjonen og «if» eller «while» løkker. Selve millis() funksjonen returnerer hvor lenge mikrokontrolleren har vært på, og vil ikke nullstille seg selv før rundt 50 dager. Ved å ta et tidsstempel og sammenligne summen av denne og en tidligere definert ventetid, med millis()-funksjonen, kan man «låse» koden i denne ventetiden, men annen kode kan fremdeles kjøre.

Millis()

```
tid_nu = millis();

if(tid_nu > forrige_tid + periodetid){  //Dersom det har gått periodetid..
  ledstate = !ledstate                  //Endrer ledstatus 0/1
  digitalWrite(ledPin, ledstate);       //Skrur av/på led
  forrigetid = tid_nu;                  //Definerer ny tid
  }
```

I kodesnutten med millis() kan vi gjøre flere ting, som for eksempel å kjøre en buzzer og servo, samtidig som vi henter inn sensordata og sender det til Blynk.

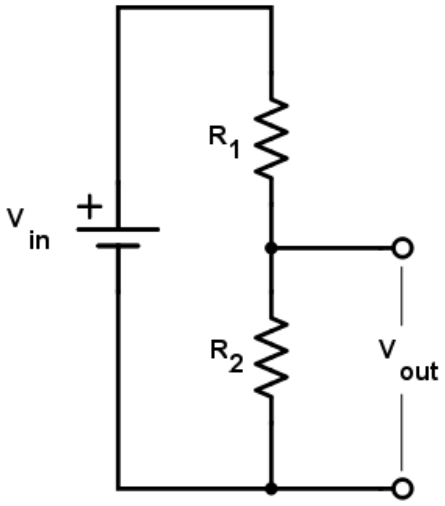
Delay

```
digitalWrite(ledPin, LOW);    //Her skrur vi av LED
delay(1000);                  //Vent i ett sek
digitalWrite(ledPin, HIGH);   //Skru på LED
delay(1000);                  //Vent i ett sek
```

I kodesnutten med delay kan vi derimot ikke gjøre noe mens delay funksjonen kjører. Delay er noe vi ikke kunne ha brukt i et så omfattende program som sensornoden.

Spenningsdeling

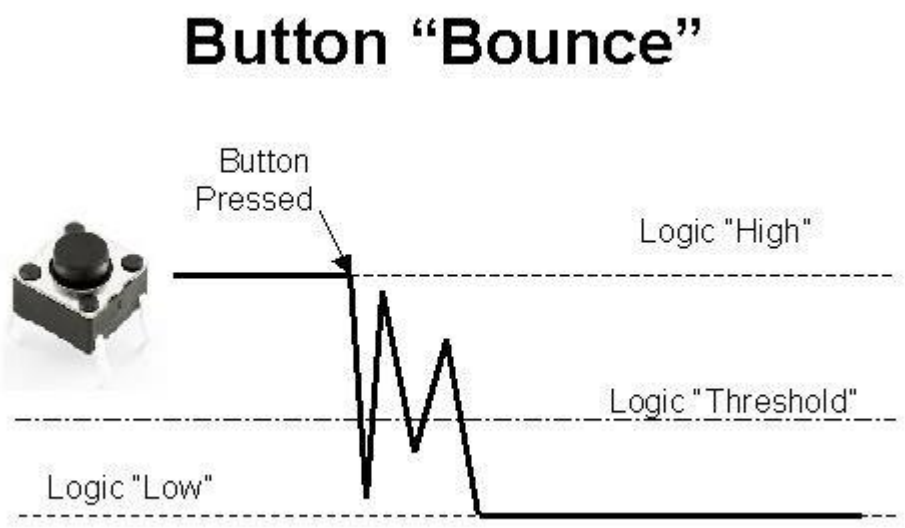
Spenningsdeling er koblingsmetode som gjør at man kan forandre på spenningen over komponentene i en elektrisk krets. Dette er hensiktsmessig å gjøre da man gjerne vil «steppe» ned en utgangsspenning ifra en inngangsspenning. Den enkleste formen for kobling er to motstander etter hverandre i serie. Lar man den ene motstanden være variabel, kan man regulere spenningen man får ut over de to mostandene. Dette er hovedprinsippet brukt i mange sensorer.



Figur 39: Spenningsdeler [16]

Debounce

Knapper, som mange andre fysiske elementer, er ikke helt perfekte. Når man trykker ned knappen hender det at mikrokontrolleren kan registrere det som flere knappetrykk på grunn av mekaniske og fysiske feil i knappen som fører til forskjellige spenningsnivå under prosessen av å trykke ned knappen, dette er kjent som «bouncing». For å unnlate å registrere ett knappetrykk som flere må vi debounce. Vi har brukt millis() funksjonen til å sikre oss at det kan kun være ett knappetrykk i et bestemt tidsrom.

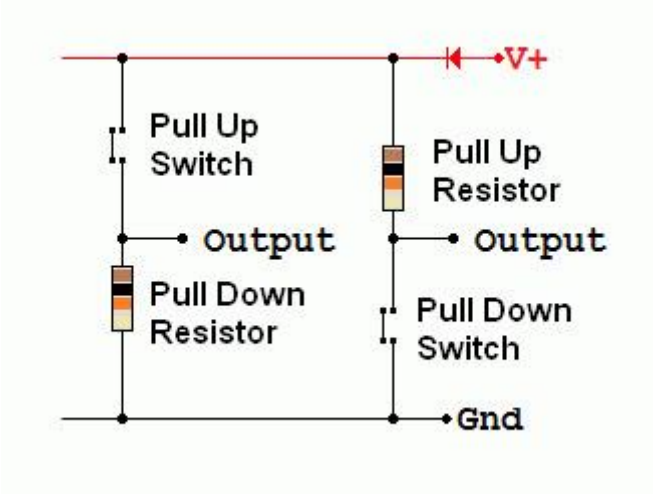


Figur 40: Knappedebounce[10]

Pull-up vs Pull-down motstander

Man kan koble opp knappekretser med pull up eller pull down motstander, de fungerer helt likt, men gir motsatt utgangssignal. En pull up motstand vil være en kobling mellom en spenning og en GPIO pin på mikrokontrolleren, da vil GPIO pinnen registrere logisk høy når knappen ikke er nedtrykt. I en pull down motstand kobler vi GPIO pinnen på ESP32en til jord, og da registrerer vi logisk lav når knappen ikke er nedtrykt.

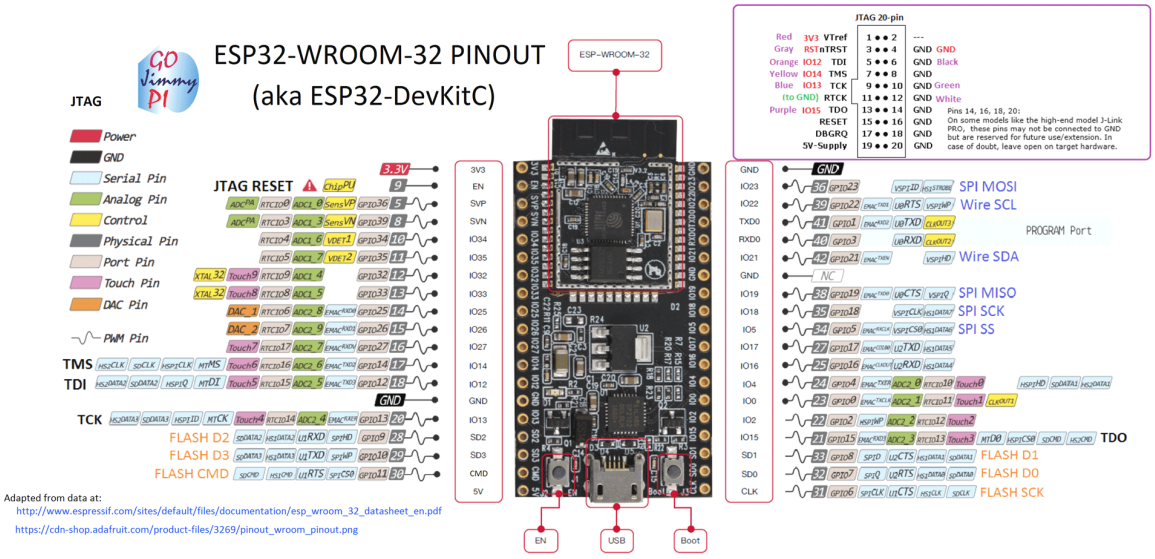
Knapp nedtrykt?	Pull up motstand utgangssignal	Pull down motstand utgangssignal
NEI	HØY	LAV
JA	LAV	HØY



Figur 41: Pull up vs pull down motstander [23]

ESP 32

ESP 32 er en mikrokontroller fra Espressif, det er etter løperen til ESP8266. Den kommer med innebygd WiFi og Bluetooth, noe som gjøre den ypperlig til prosjekter som kan kobles opp mot internett. ESP32en har flere GPIO pinner enn en Arduino Uno og operer på en mer moderne driftsspenning på 3.3V, kontra Arduino Uno sin eldre driftspenning på 5V. ESP32en har en maks operasjonsfrekvens på 240MHz, i motsetning til Arduino Uno som er på 16MHz. ESP32en er på mange måter bedre enn en Arduino Uno. [7]



Figur 42: ESP32[15]

GPIO-pins

GPIO star for "General Purpose Input/Output". I vårt tilfelle fungerer kan man bruke dem til å lese spenninger fra 0-3.3V med en oppløsning på 4096 bits, man kan også bruke dem til å levere en spenning på 3.3V, med muligheten for PWM. GPIO pinene på ESP32en ble brukt til å kontrollere buzzer, LED, servo, LCD og anskaffe sensorverdier fra termistoren, knappen, VL6180x og MQ2 på sensornoden.

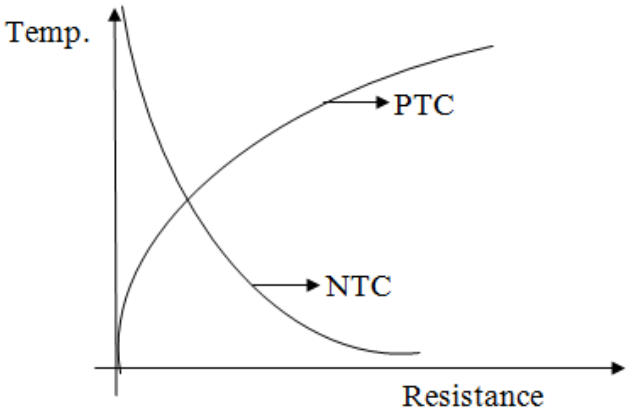
VL6180x er en «Time of Flight» sensor, som også har en innebygd lysstyrkesensor (lux). Prinsippet bak «Time of Flight» er å måle tiden et signal bruker fra det blir sendt ut, til det treffer en flate, blir reflektert tilbake, og treffer sensoren igjen. Det er forskjellige signaler som kan sendes ut, som for eksempel lyd eller lys. I VL6180x's tilfelle, er det laserlys som blir brukt. Siden det blir brukt laserlys, og ikke ultralyd, blir nøyaktigheten veldig stor, innenfor en rekkevidde fra 5mm til rundt 100mm. Sensoren får Lux verdier gjennom en ALS, «Ambient Light Sensor», ALS brukes i telefoner og datamaskiner for å justere lysstyrken på skjermen. [12]

Figur 43: Adafruit VL6180x

MQ2 er en gass sensor som måler konsentrasjonen av ulike gasser i luften. Sensoren er en MOS (Metal Oxide Semiconductor) som fungerer utfra resistiviteten til «følermaterialet» når det kommer i kontakt med gasser. Den bruker en spenningsdelingskrets. Jo høyere konsentrasjon gass i lufta, jo høyre er utgangspenningen fra sensoren. Ved helt ren luft vil det ikke være noe utgangsspenning.[6]

Figur 44: MQ2 Gass Sensor

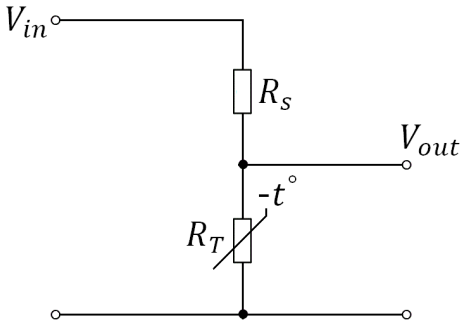
NTC termistoren vi brukte er en MF52. Den er en variabel motstand som forandrer resistivitet basert på temperatur. Ved å koble denne opp i en spenningsdelingskrets, kan vi lese av spenningen som ligger over termistoren og bruke den til å regne ut temperaturen i rommet.



Figur 45: NTC og PTC temperatur/motstand kurve [11]

For å finne temperaturen til en termistor i spenningsdeler må man først regne ut motstanden over termistoren ved å bruke formelen:

$$R_T = \frac{V_{out}}{V_{in} - V_{out}} * R_S$$



Figur 46: Spenningsdeler med termistor[14]

Deretter finner man temperaturen I celsius:

$$T = -273.15 + \left(\frac{1}{T_0} + \frac{1}{b} * \log\frac{R_T}{R_S}\right)^{-1}$$

Symbolforklaring:

R_T – Motstand over termistor

R_S – Motstand i serie med termistor

V_{out} – Spenningsdeling mellom R_S og R_T

V_{in} – Totalspenningen

b - «Beta» verdien til termistoren, denne finner du i databladet

T_0 – Starttemperatur i rommet

T – Temperatur i celsius.

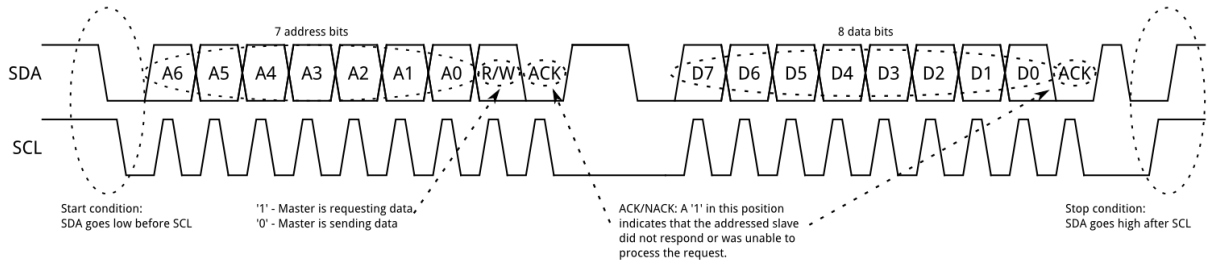
IOT

IoT står for «Internett of Things» og er et raskt voksende område innenfor elektronikk. Det er et samlebegrep for alle «ting» som er koplet opp mot et nettverk og som kan sende data til og fra hverandre uten å ha menneskelig interaksjon. Det ofte de apparatene som utgjør et «smart hjem» som forbrukerne tenker på når IoT brukes. Noen eksempel på slike apparater kan være: smartkjøleskap, *Hjemmeassistent* (Google Home, Amazon Alexa), smartlys, smarttermostat, robotstøvsuger, værstasjon etc. Dette omtales ofte som *den fjerde industrielle revolusjon*.

I2C

I2C er en master/slave kommunikasjonsprotokoll som er bygget rundt en datalinje (SDA) og en klokkelinje (SCL). I2C brukes ofte for å kommunisere mellom mikrokontrollere og tilleggsenheter til mikrokontrollere. Hver slave enhet i et I2C nettverk har sin egen adresse som brukes av masteren til å kommunisere med en spesifikk slave om gangen.

En I2C melding består av en adresse som er etterfulgt av data. Slaven leser alt som masteren sender ut, men utfører bare det som dataen, som er etterfulgt av adressen, sier at den skal. Dermed kan man ha flere slaver som er koblet opp mot en master, som i vårt tilfelle vil være ESP32-en.



Figur 47: I2C kommunikasjon ***Error! Reference source not found.***

2.5 METODE – DESIGN

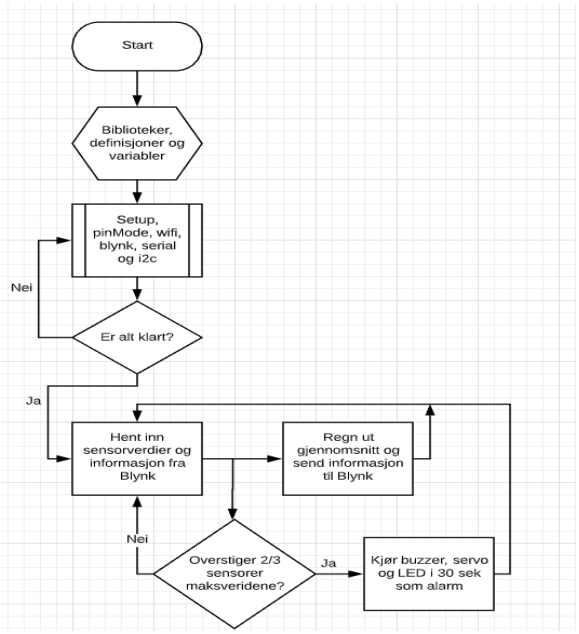
Verktøy, strategier og prinsipper er det samme som nevnt i kapittel 1.5. Gruppeorganisasjonen var noe forskjellig ettersom Michael tok på seg denne oppgaven under påskeferien. Sensorene vi benyttet oss av var; Adafruit VL6180x, NTC termistor og en MQ2 Gass sensor. Vi benyttet oss også av; Buzzer, LED, PCA9685 Servo Driver, Servo motor og I2C LCD skjerm. For å få

en god oppkoblingsplass måtte vi benytte oss av to brødbrett, siden ESP32en er akkurat for brei til å passe på ett.

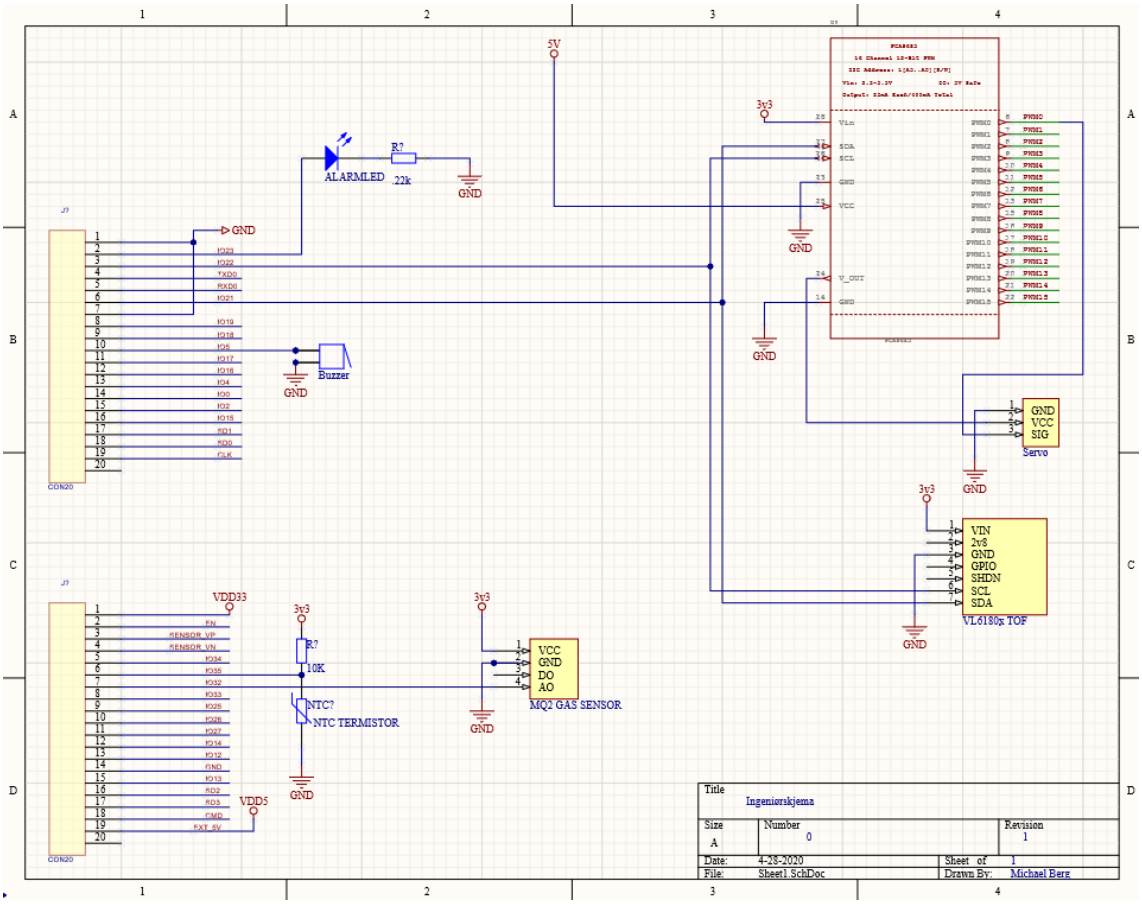
Sensornoden

Sensornodens oppgave var å innhente sensorverdier fra tre ulike sensorer, sende verdier til Blynk. Samtidig som Blynk kan sende verdier tilbake til sensornoden for å teste servomotoren. Fra Blynk appen skal man kunne velge hvilken sensor det skulle leses fra og det skal være mulig å justere på «gjennomsnittsfilteret». Vi startet med å lage en helhetlig oversikt for så å fordype oss på de mindre delene. Figuren under viser flytdiagrammet som vi designet for å være en referanse til hvordan vi skulle designe resten av sensornoden.

Koden til sensornoden skulle startes slik som alle Arduino koder, med inkludering av biblioteker, definisjoner, variabler og kobles opp mot diverse sensorer, datamaskinen ogWiFi. Så når alt er klart skal den begynne datainnhentingen fra sensorene og sende dette til Blynk. Samtidig som den henter informasjon ut av Blynk. Den skulle så regne ut det løpende gjennomsnittet og kontinuerlig sjekke om minst to av tre sensorer overgikk terskelverdiene. Dersom det skulle hende skulle en alarm utløses.



Figur 48: Flytdiagram sensornode oversikt



Figur 49: Sensornode design ingeniørskjema

Pin på ESP32	**Tilkoblet…**
IO5	Buzzer
IO21	SDA linje
IO22	SCL linje
IO23	Alarm LED
IO35	NTC spenningsdeling
IO32	MQ2 Gass sensor

Figur 50: ESP32 pins som skal brukes og hva de er koblet til

Sensoravlesninger

Første deloppgave omhandlet sensorene. Det skulle velges tre forskjellige sensorer som vi skulle koble opp mot ESP32en og få fornuftige avlesninger fra. Når vi designet sensornoden, hadde vi den praktiske oppgaven til sensornoden i tanke. Der den kunne muligens befunnet seg i et parkeringshus for å varsle om farlig høyt nivå av uønskede gasser fra forbrenningsmotorer, eller varsle om brann. For å oppfylle den overnevnte funksjonaliteten bestemte vi oss for at vi

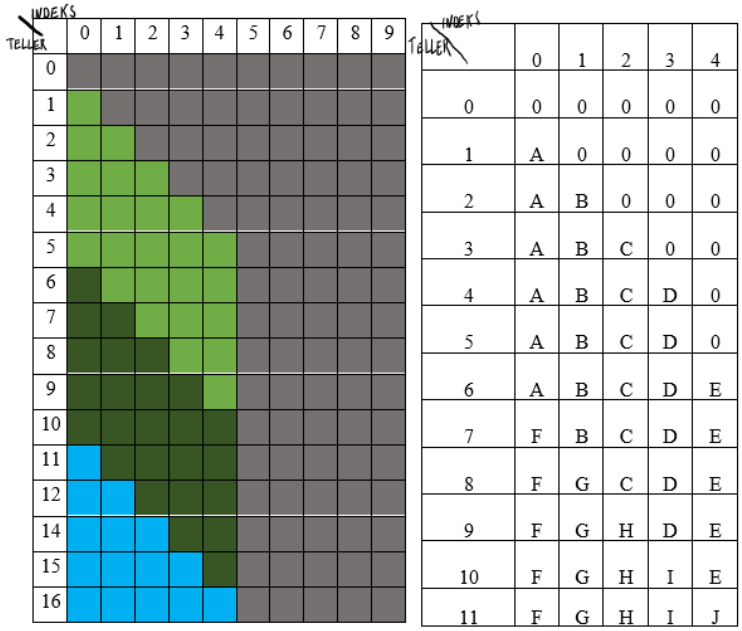
trengte sensorer som kan si noe om gassene i lufta, temperaturen og lysintensiteten. Vi bestemte oss da for å bruke sensorene; Adafruit VL6180x, NTC termistor og MQ2 Gass sensor.
NTC termistoren og MQ2 Gass sensoren er analoge sensorer, og krever derfor ingen bibliotek. Adafruit sensoren, derimot, bruker I2C og krever følgende bibliotek;

- Wire.h (For I2C kommunikasjon)
- Adafruit_VL6180X.h (For selve sensoren)

Gjennomsnittsverdien

Andre funksjonalitet var å finne den løpende gjennomsnittsverdien til de 2-50 siste sensoravlesningene. For å lage en dynamisk løsning som kunne brukes til alt, bestemte vi oss for å bruke en array med lengden 50, der vi kun bruker det vi vil av arrayet og overskriver gamle verdier.

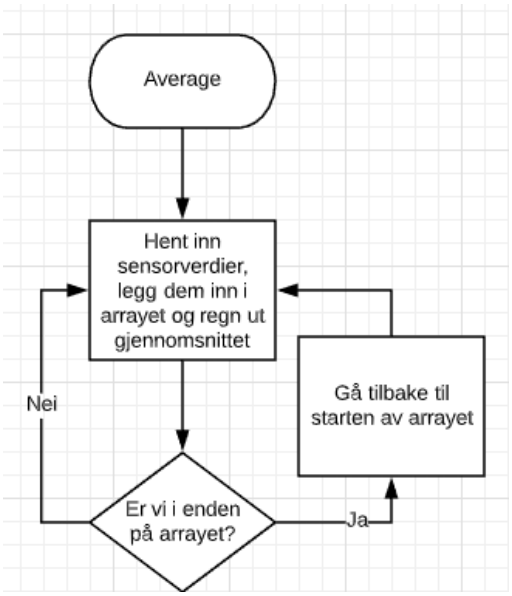
Arrayet skal fylles opp som illustrert under, der man starter på 0 indeksen og jobber seg bortover med å fylle inn sensorverdier, når man er på enden av det relevante spektrumet, så skal man starte på nytt på 0 indeksen og fylle opp fra der igjen.



Figur 51: Eksempel på hvordan array fylles

Her representerer de grå blokkene tomme arrayplasser, når vi starter å fylle arrayet med sensorverdier (lysegrønne blokker) så kan vi fortsette å indeksere helt normalt til og med indeks nr.4. Da kan vi få gjennomsnittet av de siste 5 avlesningene. Så må vi gå tilbake til indeks 0 og

begynne å fylle arrayet fra start igjen samtidig som vi finner gjennomsnittet for hver gang vi oppdaterer en verdi i arrayet.



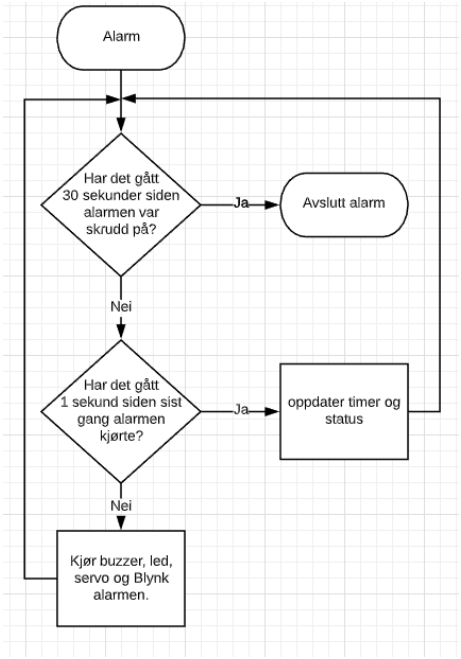
Figur 52: Flytdiagram til gjennomsnittsfilter

Max/Min alarm

Vi skulle også ha et alarmelement, der vi fikk i oppgave å kjøre en buzzer, servo, LED og et alarmelement i Blynk samtidig. Vi var klar over at delay-funksjonen og Blynk ikke var kompatible, derfor måtte vi bruke millis-funksjonen til å styre timingen på alarmen. Millis hadde uansett vært den beste løsningen ettersom delay vil sette hele programmet på pause.
Planen var å bruke «Notify» funksjonen i Blynk som alarm element, bruke PWM på buzzer, skru av og på LED og kjøre servoen mellom ytterposisjonene.

For å drive servoen brukte vi PCA9685 Servo Driveren, den trengte

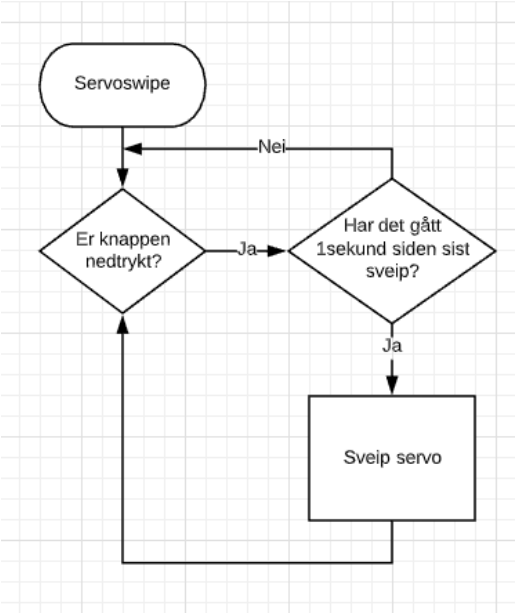
- Wire.h (For I2C kommunikasjon)
- Adafruit_PWMServoDriver.h (For selve Servo Driveren)



Figur 53: Flytdiagram for alarm

Servotest

Servoen viste seg å være et kritisk element i alarmen. Derfor måtte vi ha en måte å teste servoen på, gjennom Blynk. For å få servoen til å sveipe mellom ytterposisjonene måtte vi vite hvilke instrukser vi måtte sende til servoen. Vi bestemte oss for å bruke PCA9685, der man kan styre frekvensen og når PWM signalet skal gå fra lavt til høyt, og høyt til lavt. Videre skulle vi bruke en frekvens på 50Hz, som gir en periode på 20 millisekund. Fra der skulle vi sende pulsbredder på en og to millisekund for å sveipe servoen mellom ytterposisjonene.

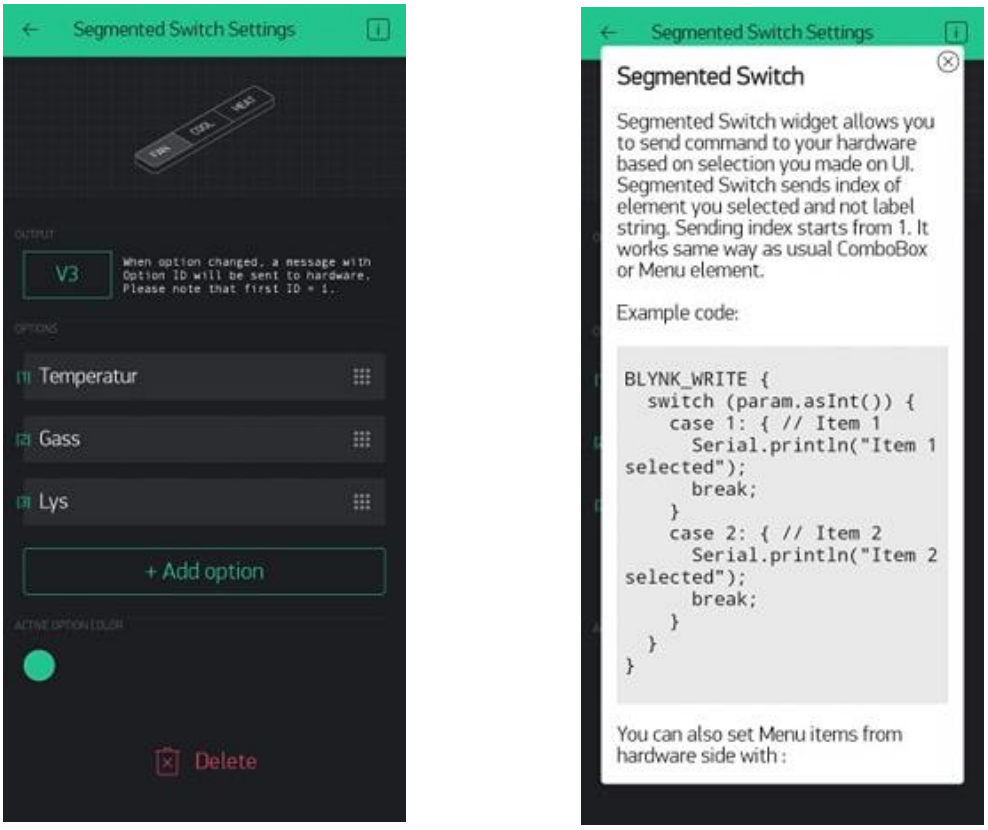


Figur 54: Flytdiagram for servotest

Blynk funksjonalitet

Planen for å koble opp sensornoden til Blynk var å bruke virtuelle pins, fordi det tillater enkel overføring av data mellom ESP32 og Blynk. Blynk har masse god dokumentasjon på sine egne sider, og på appen. Under ser du et bilde som illustrerer hvordan Blynk appen kan hjelpe med å designe et fungerende brukergrensesnitt som man kan bruke for å kommunisere mellom ESP32 og Blynk. Blynk har også andre funksjoner som vi planla å bruke, for eksempel;

- Superchart (For å plotte verdiene)
- Ulike valuedisplay (For å vise tallverdier)
- LED og Notify (Som alarm element)
- Slider (For å stille på gjennomsnittsfilteret)
- Terminal (For å skrive ut tekst om sensorene)
- Knapp (For å teste servosveiping)



Figur 55: Blynk utforming

For å kunne bruke Blynk brukte vi disse bibliotekene:

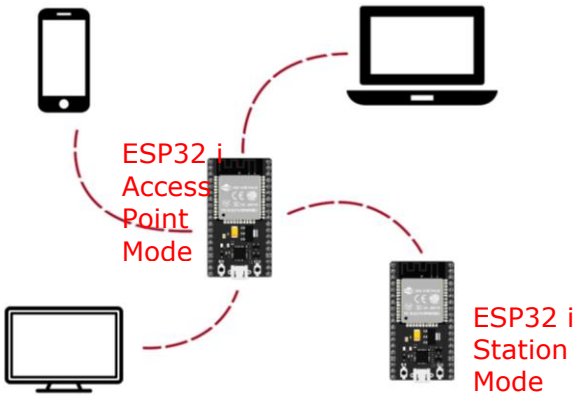
- BlynkSimpleEsp32.h (For Blynk funksjonaliteter)

- WiFi.h (Muliggjør tilkobling til WiFi)
- WiFiClient.h (For å skape en «client» som kan tilkobles en IP)

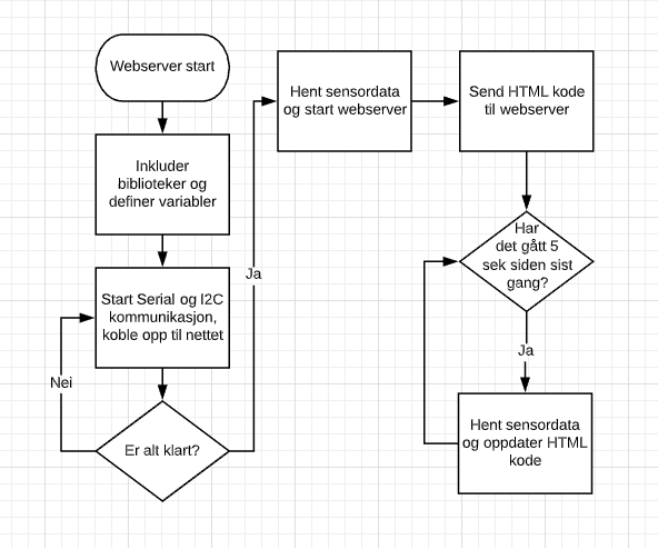
Webserver

Å lage en webserver var noe helt nytt. Derfor var vi klare over at vi var nødt til å bruke nettressurser. Vi brukte Electropeak **Error! Reference source not found.** sin «Create a w ebserver with ESP32» som ressurs. Vi bestemte oss tidlig for å bruke Station mode, da kunne vi koble oss på webserveren uten å måtte koble oss på ESP32en som om den var en ruter, som hadde vært tilfellet i Access Point Mode. Dersom vi derimot skulle befinne oss en plass uten WiFi, kunne vi ha bestemt oss for å bruke Access Point Mode, det er fordi da fungerer ESP32en som sin egen ruter som du må koble deg til for å få tilgang til webserveren. For å lage en webserver med ESP32 brukte vi følgende bibliotek:

- WebServer.h (For webserver funksjonalitet)
- WiFi.h (For å koble til WiFi)



Figur 56: En oversikt over Access Point og Station mode ***Error! Reference source not found.***



Figur 57: Flytdiagram til webserver

2.6 BEARBEIDING OG RESULTATER

Sensornoden ble hovedsakelig laget i Stavanger under påskeferien. Av den grunn har det hovedsakelig vært i Michael sine hender. Etter påsken leste gruppemedlemmene gjennom koden og kom med forslag til forbedringer, og med det bidro til ferdigstillingen av sensornoden.

Sensornoden

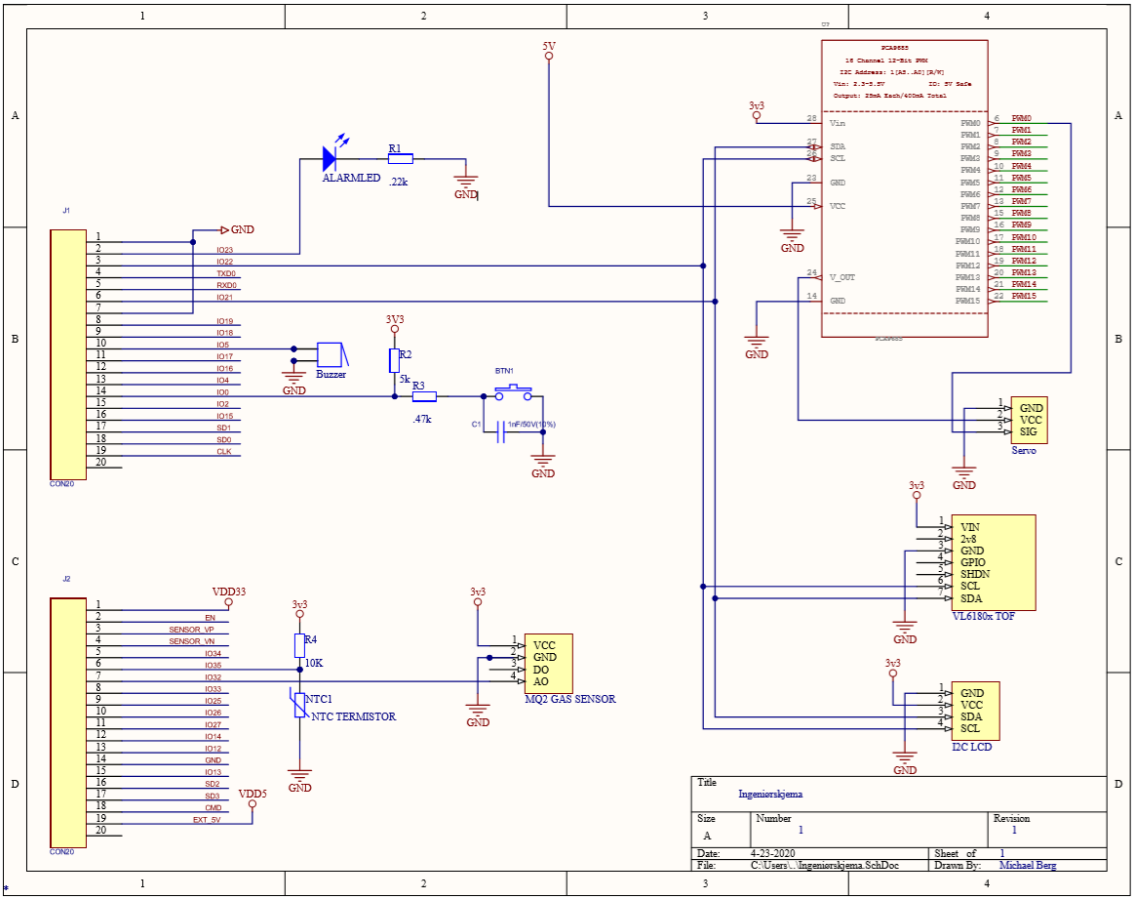
Den ferdigstilte sensornoden kom med noen ekstra funksjonaliteter, som å bruke boot knappen på ESP32en som knapp, og vi la til en I2C LCD skjerm i tillegg. Boot knappen på ESP32en er koblet opp til IO0, i en pull-up konfigurasjon. I2C LCD skjermen gjør det mulig å se sensorverdiene uten å måtte gå veien om Blynk. Dermed kan man få innsyn i sensordataen uten å måtte ha en mobiltelefon på seg.

Vi debouncet knappetrykket via millis funksjonen, og hadde en fungerende I2C kommunikasjon mellom ESP32 (som master) og VL6180x, LCD skjerm og PCA9685 (som slaver). Når vi heklet opp flere ting på I2C linjene måtte vi være nøye med adressene til hver slave, dersom to eller flere slaver hadde fått samme addresse hadde det sansynligvis ikke fungert slik vi hadde ønsket.

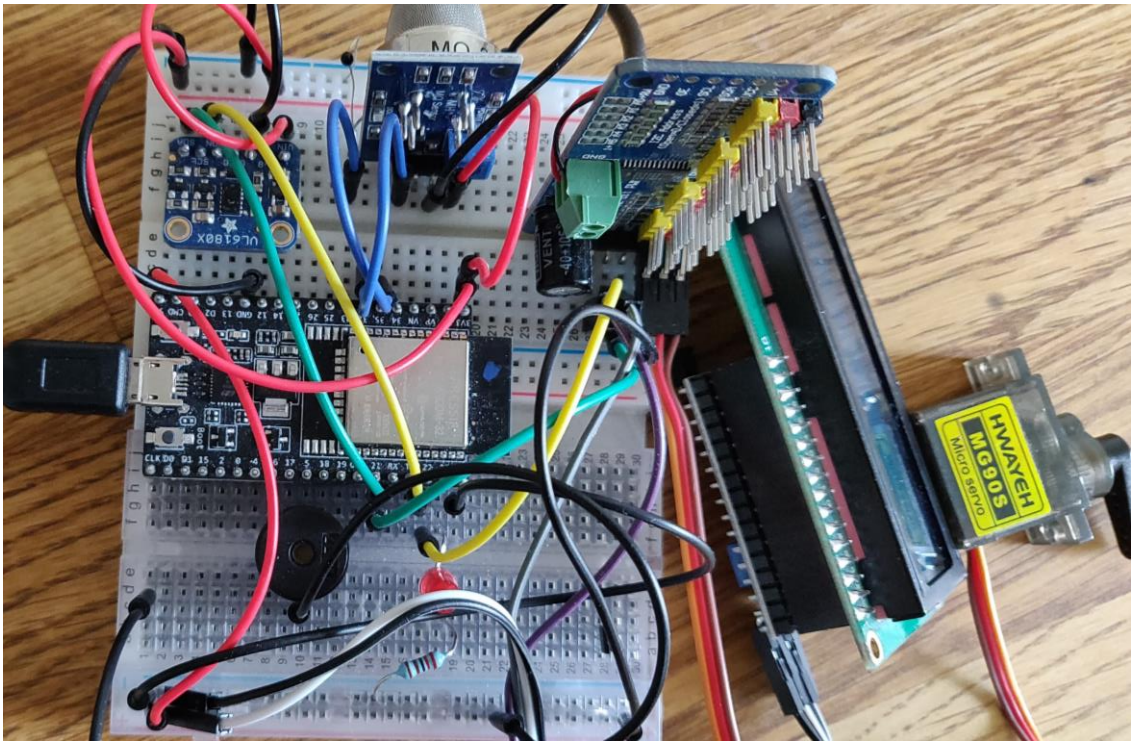
BOOT knapp «debounce»

```
if (digitalRead(0) == LOW && knappstatus == 0) {    //Nedtrykt

  if ((tid_nu - knapptid) > debouncetid) {
    knappstatus = 1;                    //Sett knappstatus til 1
    knappteller++;                      //Legg til en på telleren
    knapptid = tid_nu;                  //Ny knappetid
  }
}
if (knappstatus == 1 && digitalRead(0) == HIGH) {  //Ikke nedtrykt
  knappstatus = 0;                      //Reset
}
```



Figur 58: Endelig ingeniørskjema til sensornode



Figur 59: Sensornoden

Sensoravlesninger

Vi klarte å utvikle en programvare til esp32en som lot oss lese verdier fra tre sensorer. Å lese fra de analoge gjorde vi med analogRead() funksjonen, og vi fikk informasjon fra VL6180x sensoren gjennom I2C.

Inkludering av bibliotek og objekt til VL6180x

```
#include <Adafruit_VL6180X.h>
Adafruit_VL6180X vl = Adafruit_VL6180X(); //BRUKER 0x29 I2C adresse
```

Sensoravlesninger

```
aRead = analogRead(tempPin);                                  //Leser av analog spenningsverdi
R = aRead / (4095 - aRead) * R_0;                             //Regner ut
temp = - 273.15 + 1 / ((1 / T_0) + (1 / b) * log(R / R_0));   //Regner ut
gass = analogRead(gassPin);                                   //Leser
lux = vl.readLux(VL6180X_ALS_GAIN_5);                         //I2C les
```

Gjennomsnittsverdien

Planen som var lagt i design fungerte bra, og ga oss en fungerende kode for å finne det løpende gjennomsnittet av de siste 2-50 avlesningene.

Kode for gjennomsnittsfilter

```
//Globale variabler
const int numReadings = 50;                 //Maks antall avlesninger
float avlesningerTemp[numReadings];         //Lager en array
float total = 0;                            //Total for å finne gjennomsnitt



//Kode som kjøres når sensorverdier er innhentet
avlesningerTemp[readIndex] = float(temp);
 //Iterer til neste posisjon i arrayet:
 readIndex = readIndex + 1;

 if (readIndex >= relevantnumReadings) {
   // Start om igjen..
   readIndex = 0;
 }

//Funksjon for å finne gjennomsnittet
float gjennomsnittArray(float * array, int len) {
 total = 0;
 for (int i = 0; i < len; i++)                //Itererer
 {
   total += float(array[i]);                  //Legger til

 return (float(total) / float(len));          //gjennomsnitt
```

Max/Min alarm

En utfordring var at ESP32en ikke har en tone() funksjon slik som Arduino.

ESP32en har et analogWrite bibliotek, men dersom vi skulle brukt den, hadde vi ikke hatt kontroll på hvilken PWM kanal som ble brukt. Dette har skapt problemer for andre grupper som ikke fikk samkjørt buzzer og servo av den grunn. For å få en større grad av kontroll brukte vi derfor ledcWrite() funksjonen som ga oss mye mer kontroll over koden.

Notify funksjonen fungerte heller ikke i Blynk, dette kan trolig ha noe med at vi brukte NTNU.io skyen, der det heller ikke er mulig å få «Authentication Tokens» på e-post. Derfor brukte vi en LED i Blynk som skulle blinke i takt med alarm LED-en.

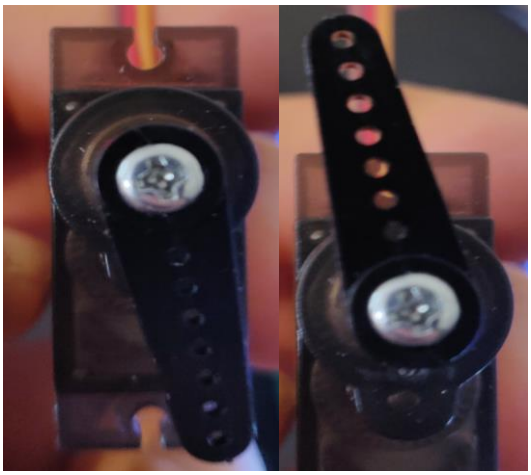
Buzzer ledcWrite

```
void setup() {
  ledcAttachPin(buzzerPin, 0);      //Sett buzzerPin på kanal 0
  ledcSetup(0, 4000, 8);            //Kanal 0, PWM frekvens, 8-bit oppløsning
  }
ledcWrite(0, 100);                  //Send PWM signal på buzzer
```

Servotest

Det sies at en servo skal ha en pulsbredde på ett millisekund for å være i nullposisjon, og to millisekund for å være i ytterposisjon. Dette var ikke tilfellet i praksis med denne servo motoren. Etter testing kom vi fram til at vi trengte pulsbredder på 0.5 og 2.5 millisekund for å kjøre disse servoene slik vi ønsket.

Figur 61: 1 og 2 ms pulsbredde



Figur 60: 0.5 og 2.5 ms pulsbredde

For å sende pulsbredder til servo motoren brukte vi, som nevnt i design delen, en PCA9685 I2C Servo Driver. Vi sendte instrukser til PCA9685 driveren over I2C.

For å finne ut av hvor stor pulsbredden var måtte vi forstå hvordan PCA9685 biblioteket fungerte. Den kom med en funksjon som het setPWM(), der man legger inn hvilken PWM kanal (0-15) man skal bruke, når den skal skrus av og når den skal skrus på.
Med en frekvens på 50Hz som gir en periode på 20 millisekund. Da teller chipen 0-4095 i løpet av 20 millisekund, over og over igjen. For å få en pulsbredde på ett millisekund må man da ha $\frac{1}{20} * 4095 \approx 205.$

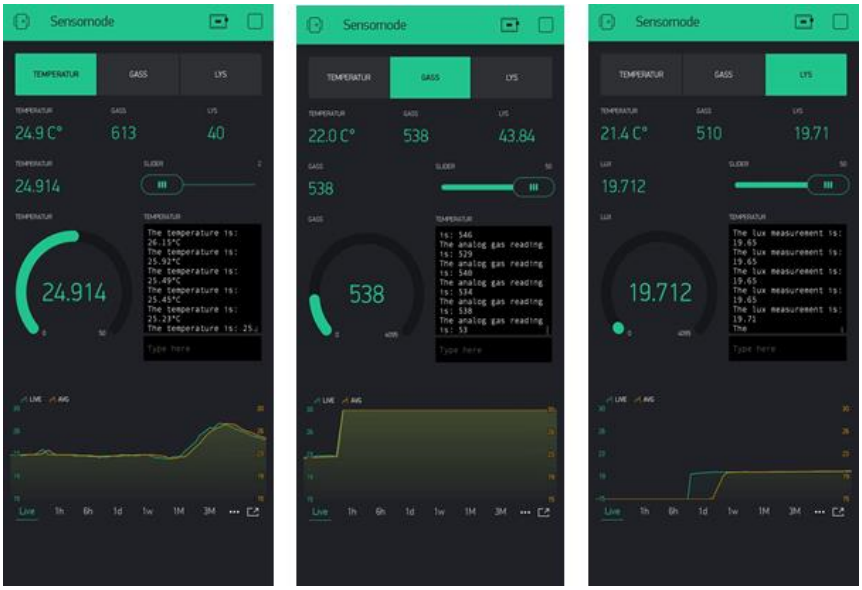
```
pca9685.setPWM(0, 0,205);       //Sett Servo pulsbredde til 1ms
```

Her sender vi instruksen der den skal kjøre PWM på kanal 0, der servoen skrus på ved start, og skrus av etter ett millisekund.
Vi prøvde oss fram med å stille på verdiene i setPWM funksjonen, vi kom fram til at
Verdiene 102 og 512 funket bra. Disse verdiene gir pulsbredder på $\frac{102}{4096} * 20 = 0.5ms$ og $\frac{512}{4096} * 20 = 2.5ms$

```
pca9685.setPWM(0, 0,102);       //Sett Servo pulsbredde til 0.5ms
pca9685.setPWM(0, 0,512);       //Sett Servo pulsbredde til 2.5ms
```

Blynk funksjonalitet

Et problem var å få plotte maksimums- og minimumsverdier til tre sensorer i et «Superchart». Ettersom ett «Superchart» tar maks 4 inputs, og maksimums- og minimumsverdier til tre sensorer vil være seks verdier måtte vi gjøre noen kompromiss. Et av kompromissene var å plotte kun maks dataene, ettersom det er kun dem som er relevante. Det andre kompromisset var å plotte de tre sensorenes maksimums og minimums verdier i egne «Supercharts».



Figur 62: Sensornode display

Figur 63: Resten av Blynk brukergrensesnittet

Webserver

Med Webserveren oppdaget vi at vi måtte manuelt oppdatere siden for å få inn nye sensorverdier. For å fikse dette problemet gjorde det slik at HTML siden sendte en «refresh request» hvert femte sekund, altså like hyppig som vi innhentet nye sensorverdier. Denne formen for å oppdatere nettsider kalles for «polling».

«Refresh Request» i HTML kode

```
<meta http-equiv=\"refresh\" content=\"5\">
```

Vi bygget videre på webserveren og hentet inn klokkeslett og dato fra «pool.ntp.org» for å vise den informasjonen på webserveren. Dette var i hovedsak for å demonstrere at det er mulig å hente data fra andre servere samtidig som man er vert for sin egen server.

Innhenting av klokkeslett og dato

```
#include <time.h>
const char* ntpServer = "pool.ntp.org";
const int  gmt = 3600;
const int   daylight = 3600;

void setup()
configTime(gmt, daylight, ntpServer);
{

void getTime()
{
 struct tm timeinfo;
 if(!getLocalTime(&timeinfo)){
   Serial.println("Noe gikk galt, fikk ikke hentet tidsinfo");
   return;
 }
 hh24 = timeinfo.tm_hour;
 mi  = timeinfo.tm_min;
 ss  = timeinfo.tm_sec;

 dd = timeinfo.tm_mday;
 mm = timeinfo.tm_mon + 1;
 yy = timeinfo.tm_year + 1900;

datostring = String(dd) + ":" + String(mm)+ ":" +String(yy);
tidstring = String(hh24)+  ":" +String(mi)+  ":" +String(ss);
//Bruker formatet dd:mm:yy og hh24:mi:ss
}
```

Sluttresultatet ble en HTML-side som vist under. Det er en rent informativ side, uten noen form for CSS.

Måling	Verdi
Temperatur	21.01 *C
Lysstyrke	41.92 Lux
Gass	766
Klokke	18:49:3
Dato	27:4:2020
Uptime	18s

Figur 64: HTML siden til webserveren

3 DRØFTING

Autonom mønsterkjøring

Løsningene på de ulike kjøremønstrene ble utført på en god måte, ettersom det viste mangt med kunnskap rundt bruken av gyrosensoren og Encoderene, i tillegg til å kunne anvende dem i en praktisk sammenheng. Fremgangsmåten kunne også ha blitt gjort på en mye lettere måte enn det gruppen valgte å gjøre, ved å "hardkode" seg frem til løsningene og bruke delay. Da hadde man ikke nødvendigvis trengt gyrosensoren til å ta skarpe svinger. For å løse sirkel- og slalåmoppgaven kunne man også bare ha satt to tilfeldige motorverdier, hvor den ene var større enn den andre for å få en konstant bue på bevegelsesbanen til Zumoen. Da kunne man deretter ta tiden på hvor lang tid det tok for Zumoen til å kjøre i en hel sirkelbane i sirkeloppgaven, likeledes med halvsirklene i slalåmoppgaven. Vi ville derimot vise mer kunnskap enn å bare prøve og teste ulike verdier, slik at det viste god forståelse ved bearbeiding av dataene fra de ulike sensorene.

I denne sammenhengen utnyttet vi gyrosensoren til å vite posisjonen til Zumoen, for å deretter bestemme hvor mange grader den skulle snu. Hovedpoenget med å få Zumoen til å kunne kjøre i en sirkelbane med en vilkårlig radius, var for å bruke matematikken til å regne ut sirkelbanene,

i tillegg til at det skulle være en universal kode for alle mulige typer sirkler, og ikke en fast bestemt. Planen var også at det skulle være lettere å utføre slalåmoppgaven, ettersom det bare var å "taste" inn hvor langt det var mellom de ulike hindrene.

Noen ulemper med løsningen vår var at det ble vanskelig å finne konstantene til forholdet mellom motorfartene, for at de skulle kunne kjøre i en kalkulert bane. Dette medførte også til at sirkelbanekjøringen kun var egnet for én spesifikk hastighetsverdi, ettersom konstanten måtte endres hvis farten var annerledes. Årsaken til at det ikke fungerte som planlagt var dermed på grunn av at det ble vanskelig å definere RPM-en til motorene og med andre ord farten til Zumoen. Alle oppgavene ble veldig lik fremgangsmetoden og inneholdt mange funksjoner lik som beskrevet i metode-delen, som førte til at det ble enkelt å vite hvor en var.

Linjefølger

Vi fikk til slutt løst linjefølgerkoden etter en del redesign av koden. Ettersom den var den siste programmeringsdelen som ble påbegynt, var det flere som forsøkte seg på å finne fornuftige og gode løsninger på den, siden flere var ferdig med hovedoppgaven sin. Det resulterte i en del forskjellige ideer. Den første angrepsmetoden ble designet relativt hurtig, og det viste å ikke lønne seg. Den bygget i hovedsak på en tidligere øving, som kun krevde å følge en «komplett» teipbane og ikke på å håndtere en rekke «utfordringer». Så når den skulle forsøke å håndtere de diverse utfordringene, ble koden knotete, innviklet og uoversiktlig. I stedet for å prøve å få koden oversiktlig og effektiv, ble hele grunnideen forkastet og versjon 2 ble designet opp fra bunn av. Den viste seg å være mye mer oversiktlig og enklere å arbeide med. Vi greide å opprettholde ideen om å lage en modulær kode, der «linjefølger-kjernen» enkelt kan byttes ut, slik at det kan variere mellom PID-regulering og «hardkoding».
Kort oppsummert er gruppen fornøyd med det endelige resultatet, og linjefølgeren har vært en lærerik det av prosjektet.

Obstacle avoidance

Prosjektet ble løst på en god og strukturert måte, og dens funksjonalitet fungerte bedre enn forventet. Til tross for at det var tre hovedproblemer [kapitel 1.6] som gjorde prosessen med å gjøre prosjektet 100 prosent funksjonell relativt vanskeligere, ordnet det seg til slutt. På starten var koden knutete, som nevnt tidligere var det første av tre hovedproblemer «hjernen» til

Zumoen. Det var mange feiltester og frustrasjon oppstod, men med hjelp fra gruppemedlemmene og godt samarbeid og gode ideer som ble iverksatt for å løse problemet ble koden til prosjektet mer strukturert og oversiktlig.

En annen svakhet for dette prosjektet var at det viste seg å være et blindpunkt på proximity sensorene som igjen bygde opp frustrasjonen hos gruppen med at Zumoen kolliderte med smale gjenstander. Hensikten med prosjektet var at Zumoen kunne kjøre fritt uten å kollidere med andre gjenstander, men det at proximity sensoren hadde ett blindpunkt var ikke forventet. Årsaken til blindpunktet er fortsatt ukjent, men gruppen håper å finne et svar på dette i senere fremtid. Som nevnt tidligere var løsningen for dette problemet og redde dette prosjektet å anvende akselerometeret. Nettopp til å oppdage kollisjon og bruke reverserings funksjonen til Zumoen til å kjøre vekk fra kollisjonsobjektet.

Alt i alt følte gruppen seg fornøyd med hvordan vi utførte dette prosjektet. Dette prosjektet var modul. 3 som gikk ut på få Zumoen til å gjøre noe spektakulært. Et obstacle avoidance system inkludert med distansemåler laget av førsteklassinger, vil gruppen si seg meget fornøyd med.

Sensornode

Vi fikk løst oppgaven på en fin måte der ting fungerte som de skulle. Vi fikk også lagt til noen ekstra ting på sensornoden og webserveren. Vi er fornøyde med vår løsning på denne oppgaven. En ting som plager oss litt er svakheten med Blynk er at man ikke kan endre på maksimums- og minimumsverdiene til «Superchart» gjennom ESP32en, av den grunn så kan det være vanskelig dersom man skal bruke samme «Superchart» til å plotte en sensor sine avlesninger og gjennomsnittsverdi om gangen, der man skal kunne bytte på hvilken sensor det leses fra. Webserveren også kunne vært bedre med tanke på design, kunne også vært litt mindre kjedelig og hatt mere farger og illustrasjoner.

Samarbeid i gruppen

Koden vår ble fint strukturert og oppdelt i forhold til kriteriene vi hadde laget for kodestrukturen. Det var magert med kommunikasjon i gruppen i starten av prosjektet, men etter hvert som de fleste av oss samlet oss ble kommunikasjonen bedre. Når kommunikasjonen bedret seg, ble det også betydelig mye mer arbeid gjort med prosjektet. En delt prosjektgruppe

fungerte ikke så bra fordi det medførte at de som var fysisk til stede i Trondheim jobbet mer med prosjektet, og bar mer av lasset. Når gruppen ikke var fysisk samlet, var kommunikasjonen problematisk. Det hendte ofte at samtlige gruppemedlemmer bare leste meldingene, spesielt når man prøvde å delegere oppgaver. Til tross for vanskeligheten i gruppen mener vi at vi fikk løst de tildelte oppgavene på en god måte, der vi ofte gjorde mer enn det oppgaven spurte om.

4 KONKLUSJON - ERFARING

Prosjektet har gitt gruppen en bedre forståelse av hvordan Arduino-plattformen fungerer med å linke til biblioteker og andre filer. I tillegg til dette har gruppen fått en solid og dyp forståelse av hvordan sensorer som: akselerometer, termistor, lyssensor, gass sensor, «proximity» (distanse) og gyroskop fungerer og deres bruksområde. Samtidig som at arbeid med Zumoen har gitt gruppen muligheten til å utføre prosjekter hvor de nevnte sensorene kan anvendes til store prosjekter som obstacle avoidance og linjefølger. Gruppen har også fått ett innsikt av hva et godt arbeidsmiljø og en god prosjektstyring har å si for et vellykket prosjekt.

Fra start til slutt av prosjektet har det skjedd mye, og det har vært tider hvor prosjektet så ut til å være avlyst grunnet kjente årsaker. I mars hadde alle gruppemedlemmene reist tilbake til sine hjembyer. Gruppen var skilt over store avstander: Stavanger og Oslo, og det var vanskelig å for oss å samarbeide og teste kodene våre uten Zumoen eller ESP'en tilgengelig for de fleste i gruppen. I april ble det planlagt at alle i gruppen skulle reise tilbake til Trondheim og gjøre vårt beste for å fullføre prosjektet. Da flesteparten av gruppen var samlet i ett sted ble prosjektstyringen lettere og arbeidskravene ble fort krysset av fra listen.

Gjennom prosjektet har vi opparbeidet en bedre forståelse for hvordan IoT fungerer. Vi har fått en introduksjon til webservere og http protokollen, og forstår bedre hvordan de fungerer og begrensningene de kommer med. Vi har også lært om I2C, der vi bruker tre slaver med forskjellige adresser på samme nettverk. Videre har vi lært å lese og tegne ingeniørskjema og blitt kjent med programvaren Altium, dette gjør oss bedre rustet til å designe kretskort i framtida.

Arbeidet med prosjektet har også gjort oss flinkere til å lese datablad effektivt, å finne fram til relevant informasjon innen rimelig tid.

Mulige utvidelser

Våre framtidige planer med Zumoen hadde vært å kjøre den gjennom en labyrint og lage en logaritme til å finne den korteste veien ut. I tillegg til dette kunne vi ha tenkt å koble opp Zumoen til en joystick og styre den manuelt.

Til sensornoden kunne vi tenkt oss å bruke en annen protokoll som gjør at vi slipper å laste inn hele siden på nytt, slik at ESP32en kan pushe data til serveren og vi får «live» oppdaterte verdier. Å bruke Socket.io og Node.js hadde trolig gjort dette mulig. Videre hadde det vært ønskelig å tilkoble serveren til verdensveven slik at vi kan koble oss til serveren fra hvor som helst i verden.

5 REFERANSER

[1] https://electropeak.com/learn/create-a-web-server-w-esp32/
[2] https://learn.sparkfun.com/tutorials/i2c/all
[3] https://randomnerdtutorials.com/esp32-ntp-client-date-time-arduino-ide/
[4] https://github.com/espressif/arduino-esp32
[5] https://cdn-shop.adafruit.com/datasheets/PCA9685.pdf
[6] https://components101.com/mq2-gas-sensor
[7] https://www.espressif.com/en/products/devkits/esp32-devkitc/overview
[8] https://learn.sparkfun.com/tutorials/i2c/all
[9] https://www.elprocus.com/pull-up-and-pull-down-resistors-with-applications/
[10] https://embedds.com/software-debouncing-of-buttons/
[11] https://www.researchgate.net/figure/The-thermistor-temperature-characteristic-curve-In-Figure-1-the-thermistor-temperature_fig1_309267235
[12] https://learn.adafruit.com/adafruit-vl6180x-time-of-flight-micro-lidar-distance-sensor-breakout/downloads
[13] https://github.com/fdebrabander/Arduino-LiquidCrystal-I2C-library
[14] https://www.petervis.com/electronics%20guides/calculators/thermistor/thermistor.html
[15] https://components101.com/microcontrollers/esp32-devkitc
[16] https://www.allaboutcircuits.com/tools/voltage-divider-calculator/
[17] https://www.pololu.com/category/170/zumo-32u4-robot
[18] https://github.com/pololu/zumo-32u4-arduino-library
[19] https://electronics.stackexchange.com/questions/283947/dc-motor-speed-vs-pwm-duty-cycle
[20] https://www.pololu.com/docs/0J63/3.4
[21] http://rossum.sourceforge.net/papers/DiffSteer/
[22] https://embedds.com/software-debouncing-of-buttons/
[23] https://www.elprocus.com/pull-up-and-pull-down-resistors-with-applications/
[24] https://www.arduino.cc/en/tutorial/PWM
[25] https://github.com/IELET1002GRUPPE29/DatatekProsjekt
[26] https://www.arduino.cc/en/tutorial/variables

Figurer

6 VEDLEGG

Nummer 1 – Encoderverdier

https://raw.githubusercontent.com/IELET1002GRUPPE29/DatatekProsjekt/master/AutonomousDriving/encoderverdi_slalom.rtf

Nummer 2 – Sensorverdier (Sensornode)

https://raw.githubusercontent.com/IELET1002GRUPPE29/DatatekProsjekt/master/Sensordata.txt

Nummer 3 – PWM

https://www.arduino.cc/en/tutorial/PWM

Nummer 4 – Zumo 32U4

https://github.com/pololu/zumo-32u4-arduino-library

https://www.pololu.com/category/170/zumo-32u4-robot

Nummer 5 – ESP32

https://github.com/espressif/arduino-esp3259

https://www.espressif.com/en/products/devkits/esp32-devkitc/overview

Nummer 6 – Sensornode kode

https://github.com/IELET1002GRUPPE29/DatatekProsjekt/blob/master/Sensornode/Sensornode.ino

Nummer 7 – Webserver kode

https://github.com/IELET1002GRUPPE29/DatatekProsjekt/blob/master/esp32web/esp32web.ino

Nummer 8 – Zumo obstacle avoidance

https://github.com/IELET1002GRUPPE29/DatatekProsjekt/blob/master/Obstacle-Avoidance/Obstacle_avoidance.ino

Nummer 9 – Zumo autonomous driving

https://github.com/IELET1002GRUPPE29/DatatekProsjekt/blob/master/AutonomousDriving/Prosjekt.ino

Nummer 10 – Zumo PD Linjefølger

https://github.com/IELET1002GRUPPE29/DatatekProsjekt/blob/master/pid-regulator/pid-regulator.ino

Nummer 11– Ingeniørskjema til sensornode

https://github.com/IELET1002GRUPPE29/DatatekProsjekt/blob/master/Schematics/Ingeni%C3%B8rskjema.pdf