Universitetet
i Stavanger

DET TEKNISK-NATURVITENSKAPLIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/Spesialisering: Skavhellen: Datateknologi Almås: Elektroingeniør, y-vei	Vårsemester, 2022 Åpen / Lukket Åpen
Forfattere: Sindre Skavhellen Lars Almås	Signatur: Sindre Skavhellen Lars Almås
Fagansvarlig(e): Morten Mossige, Karl Skretting, Ståle Freyer	
Oppgavetittel: Utvikling av REST interface og robotkontroller-logikk på en Raspberry PI Engelsk tittel: Development of REST interface and robot controller logic on Raspberry PI	
Studiepoeng: 40 (2*20)	
Emneord: Raspberry PI, Robotkontroller, REST, Hexapod, EtherCAT, SOEM	Sidetall: 37, uten vedlegg Vedlegg: A og B, 5 sider og en .zip-fil Stavanger 14. Mai 2022

Abstrakt

Ettersom datamaskinene blir mindre og mer kompakt, følger også kontrollsystemer for maskiner og roboter denne trenden.
Denne rapporten utvikler og realiserer en robotkontroller og et API som kjører på en Raspberry PI.

Robotene i dette prosjektet får alle sine kommandoer over EtherCAT-protokollen. En kan dermed styre robotene gjennom enheter som muligheten til å bruke EtherCAT-protokollen for kommunikasjon over Ethernet. Dette realiseres ved hjelp av SOEM, som er et rammeverk for å kommunisere over EtherCAT.

Robotkontrolleren kjører to ulike tilstandsmaskiner som er opprettet ved hjelp av Pytransitions-biblioteket. Dette overvåker tilstander i robotene og behandler feilmeldinger. Kommandoer som kommer inn blir behandlet og utrekninger blir utført før de blir sendt til robotene.

En Flask-server startes opp slik at programmets API kan opprette REST-ressurser og WebSocket. REST-ressursene gir mulighet til innsending av kommandoer for styring av robotene og robotkontrolleren. Hendelseslogg og signaler fra robotene blir sendt ut til klienter fortløpende. Dette gjennonmføres via en WebSocket som blir etablert med SocketIO.

Forord

Først og fremst ønsker vi å takke Morten Mossige for arbeidet med å drive frem dette prosjektet og alt arbeid og engasjement han har lagt inn. Samtidig ønsker vi å takke Karl Skretting og Ståle Freyer som også har bidratt til å sørge for at prosjektet kom i mål.

Vi ønsker også å rette en takk til de som har deltatt i prosjektet med andre oppgaver, Aleksander, Kestutis, Ådne, Vebjørn og Andreas.
Det har aldri vært vanskelig å få hjelp om vi har trengt det.

Dette har vært et spennende og lærerikt prosjekt der vi har fått brukt kunnskap og evner som ikke har blitt så mye brukt på en stund. Vi har også lært mye nytt, som helt garantert vil komme til nytte senere.

Takk også til Erlend Tøssebro, en bedre spillmester skal en lete lenge etter. Uten han hadde ikke vi som har jobbet med denne oppgaven møttes.

Sindre
Jeg vil gi en stor takk til familie og venner for deres støtte gjennom hele utdannelsen, uten de hadde jeg ikke vært her jeg er i dag. De har løftet humøret mitt når det har vært på sitt tyngste.

Lars
Jeg ønsker å rette en stor takk til familie og venner som har bidratt med moralsk støtte og hjulpet gjennom hele utdannelsen. Vil også takke arbeidsplassen min, Ullrigg Test Center ved NORCE, som har hjulpet til og lagt til rette for at det skulle være mulig å kunne jobbe og ta denne utdanningen samtidig.

Innholdsfortegnelse

Kapittel 1

Introduksjon

1.1 Oppgavebeskrivelse

Utvikle en robotkontroller og et grensesnitt ved bruk av Python, som skal kjøre på en Raspberry Pi. Robotkontrolleren vil kommunisere gjennom en EtherCAT protokoll med driverene som kjører motorene til roboten. Målet er en generell kontroller som kan kobles mot flere forskjellige typer roboter. For denne oppgaven er hovedfokuset å koble den opp mot en robot som står på Universitetet i Stavanger (UiS), med mulig utvidelse mot andre roboter om tiden tillater det.

Robotkontrolleren trenger å kunne logge hendelser og motta kommandoer fra et grafisk brukergrensesnitt. Det må også implementeres sikkerhetsbarrierer for roboten slik at ingen ulykker skjer under kjøring. Når kontrolleren starter opp skal man kunne velge hvilke type robot som skal kjøres og om det er en fysisk eller simulert versjon av den type robot.

1.2 Mål for oppgaven

Styring av robot
Det skal sendes kommandoer fra robotkontrolleren til sanntidsprogrammet, som kommuniserer med roboten.

Brukergrensesnitt
Det skal kommunisere med et brukergrensesnitt over nettverk.

Simulert robot
Det skal være mulig å kjøre robotkontrolleren mot en simulert modell.

Datasikkerhet
Det skal hindres at hvem som helst kan sende kommandoer.

Maskinsikkerhet
Roboten skal kreve fysisk tilstedeværelse.

1.3 Oversikt

Oppgaven er lagt opp som følger:
Kapittel 1: Introduserer oppgaven.

Kapittel 2: Forklarer tilstanden på prosjektet ved oppstart og arbeid som har gått parallelt med denne oppgaven. Går over utstyr og forklarer om robotene.

Kapittel 3: Går over kode og program samt forklaringer av programmet.

Kapittel 4: Inneholder litt om resultater, kompatibilitet og feilkilder.

Kapittel 5: Sammenligner resultatet mot det som var oppgaven, og bakgrunn for hvorfor enkelte valg er tatt.

Kapittel 6: Omhandler hvordan prosjektet har gått, og kommer med noen forslag til videre arbeid.

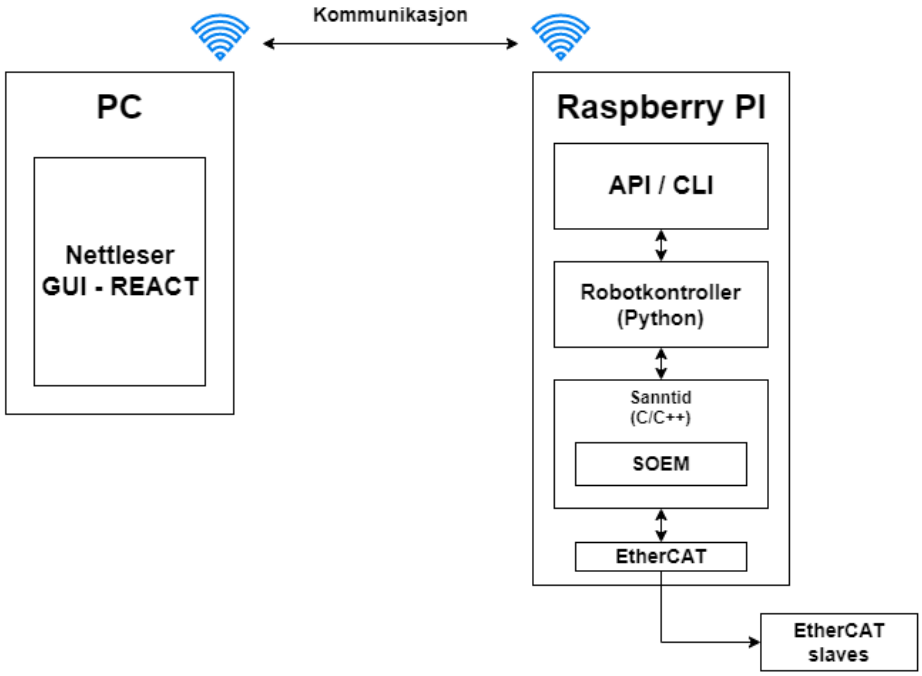
1.4 Forkortelser

API - Application Programming Interface / Programmeringsgrensesnitt
CLI – Command Line Interface / Kommandolinjegrensesnitt
GPIO – General Purpose Input Ouput / Generelle inn- og utganger
GUI – Graphical User Interface / Grafisk brukergrensesnitt
REST - Representational State Transfer / Representativ Tilstandsoverføring
SOEM – Simple Open Ethercat Master
UiS – University of Stavanger / Universitetet i Stavanger

Kapittel 2

Bakgrunn

Systemet består av en Raspberry PI, som skal styre en robot. Programmet kjører primært i python, med en del som kjører i C/C++.
For å styre roboten kan en enten kjøre den manuelt gjennom et kommandolinjegrensesnitt (CLI), eller koble til et grafisk brukergrensesnitt (GUI) gjennom et programmeringsgrensesnitt (API) som ligger i robotkontrolleren.
For å kommunisere med roboten sendes det kommandoer til et rammeverk, Simple Open EtherCAT Master (SOEM), som kjører i sanntid i et eget C/C++ program for å kommuniserer med driverene til roboten.



Figur 2.1: Oversikt over oppsettet

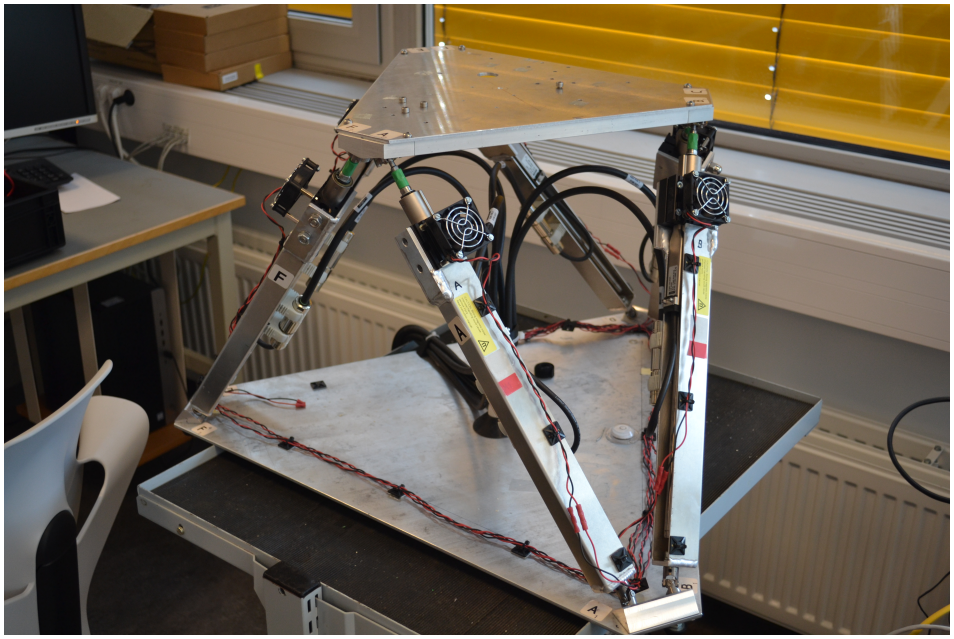
Prosjektet er delt opp i følgende oppgaver:
Robotkontroller/API - Bacheloroppgave (denne oppgaven)
Grafisk brukergrensesnitt - Masteroppgave
Kinematikk/modellering - Masteroppgave
Kommunisere med sanntids modulen - ELE630 prosjekt

Prosjektet bruker kontinuerlig integrasjon gjennom *GitHub*. Dette betyr at alle endringer som gjøres ligger tilgjengelig for alle utviklere. Når endringer på prosjektet lastes opp kjøres *GitHub Actions*. Dette er tester som kjøres automatisk på kodebiblioteket.
De tre hovedkategoriene for dette er *unittester*, *Linting* og *Doxygen*. *Unittester* tester enkelt funskjoner, *Linting* gjør kodeanalyse og *Doxygen* står for dokumentering. Det kjører også en *GitHub Action* på den delen av programmet som har med GUI.

2.1 Robotene

Robotkontrolleren er programmert slik at det er mulig å utvide med forskjellige roboter i fremtiden. De to robotene som er programmert inn nå er en hexapod og en vippe.

Hexapod
Roboten programmet i hovedsak utvikles for er en hexapod, en maskin bestående av to flater som er festet med seks lineære aktuatorer, fig. 2.2.



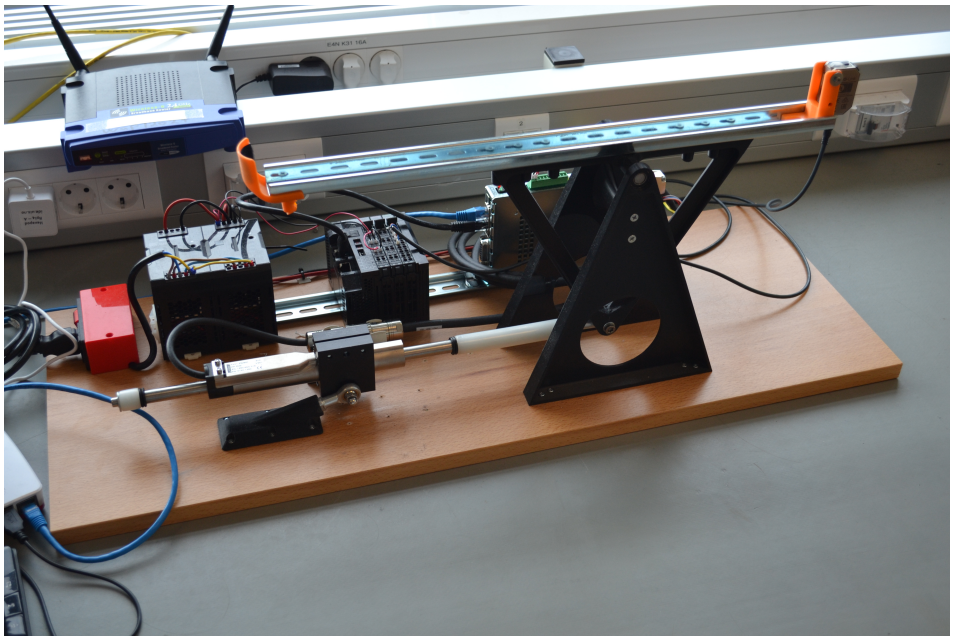
Figur 2.2: Hexapod

Denne konstruksjonen er kjent som en Stewarts plattform. Det gir oss muligheten til å styre forflyttelse i alle 3 dimensjoner, og rotasjon i alle 3 dimensjoner for den øverste platen[1].

Hexapoden brukt her er bygd opp med elektriske lineære motorer, som drives av egne drivere. Både motorer og drivere kommer fra produsent LinMot.

Vippe

Den andre roboten brukt i dette prosjektet er en vippe, eller en rocker, som den blir referert til i programmet. Prinsippet ved denne roboten er å styre vinkelen på en skinne som det ligger en kule på.



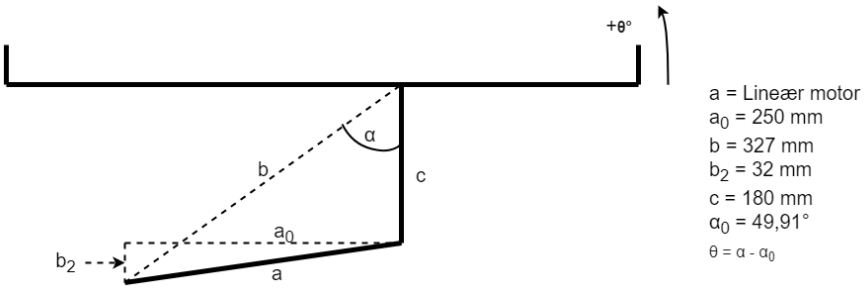
Figur 2.3: Vippe

2.2 Kommunikasjon med robotene

Robotkontrolleren skal kjøres på en Raspberry Pi, en ettkortsdatamaskin som kjører operativsystemet Linux. På dette kortet vil det kjøre et program i C/C++ som kommuniserer med driverne til roboten. Dette oppnås ved at i C/C++ delen av programmet kjøres det et bibilotek kjent som Simple Open EtherCat Master (SOEM). Dette kan brukes i robotkontrolleren ved å sende kommandoer fra python til C/C++. Driverene er av typen LinMot C1150-EC-XC-0S-000 og er det som driver motorene som er lineære elektromotorer, LinMot PS01-23x80F-HP-R20. Driverene og Raspberry Pi kommuniserer sammen gjennom EtherCAT protokoll, gjennom en nettverkskabel.

2.3 Modellering og invers-kinematikk

Det er tatt i bruk modeller, målinger og utrekninger gjort av Karl Skretting og Morten Mossige for å gjøre utrekninger fra lengder på motorer til robotenes posisjoner. Modellering av Hexapod er ikke en del av denne oppgaven.



Figur 2.4: Skisse av vippens geometri, basert på skisse av Karl Skretting

$$a^2 = b^2 + c^2 - 2bc * cos\alpha$$
$$\alpha = cos^{-1}\left(\frac{a^2 - b^2 - c^2}{-2bc}\right)$$

2.4 Brukergrensesnitt

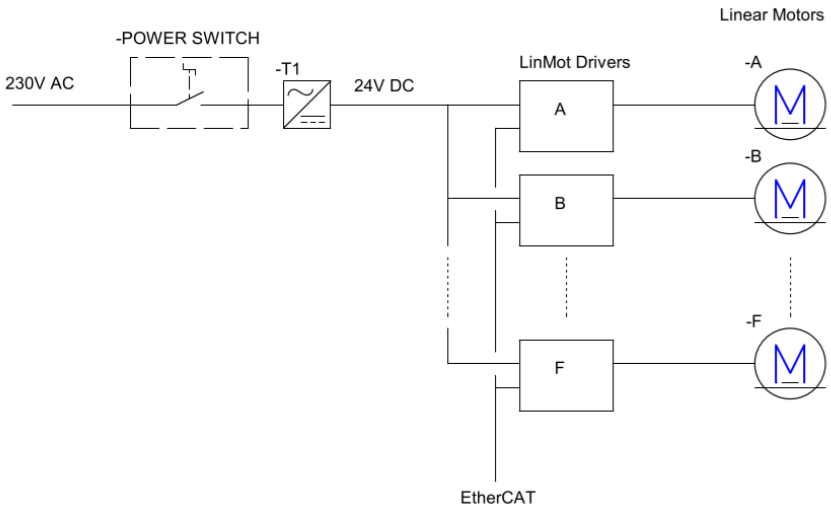
De to brukergrensesnittene som har blitt tatt i bruk i dette prosjektet er et grafisk web-grensesnitt og et kommandolinjebasert grensesnitt. Det grafiske grensesnittet er under utvikling, som del av en masteroppgave. Kommandolinjegrensesnittet var startet av Morten Mossige. Dette grensesnittet er basert på CMD2 biblioteket. CMD2 er et bibliotek som lar deg definere kommandoer i et DOS-lignende grensesnitt.

2.5 Programmets startpunkt

Ved starten på prosjektet ble det avgjort at det var best å fortsette arbeidet på tidligere kode. Denne koden kommer fra Morten Mossige. Det var i hovedsak ett rammeverk som inneholdt kommandolinjebasert grensesnitt for å kommunisere med driverene til motorene. Struktur og en del klasser var opprettet for videre utvikling av programmet.

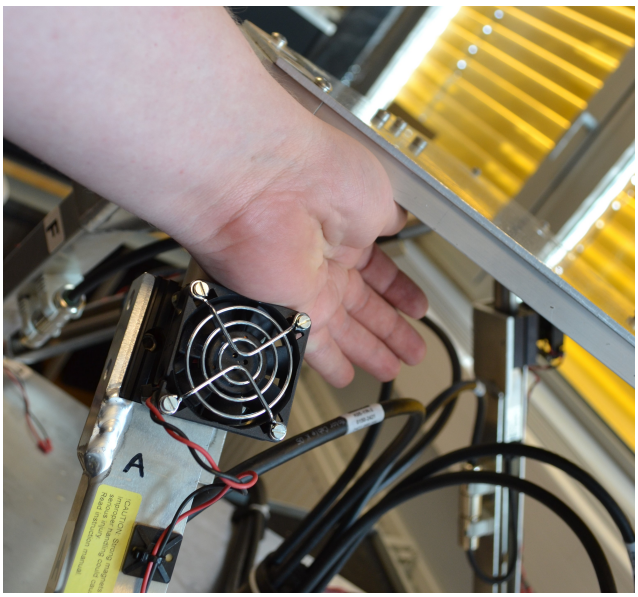
2.6 Risikovurdering av Hexapod

Elektrisk støt:
Vurderes som liten risiko da roboten driftes på 24V.



Figur 2.5: Elektrisk enlinjeskjema for Hexapod

Mekaniske bevegelser: Vurderes middels risiko.
Lav sjanse for at det skjer, men motorene har potensial til store krefter og raske bevegelser.
Motorstag er skjulte, men det er en potensiell klemfare. Motorene har potensial til å akselerere raskt nok til å slå ganske hardt.
Motor maks hastighet blir satt til en lav hastighet for å redusere fare samt at spenning til motor blir redusert.

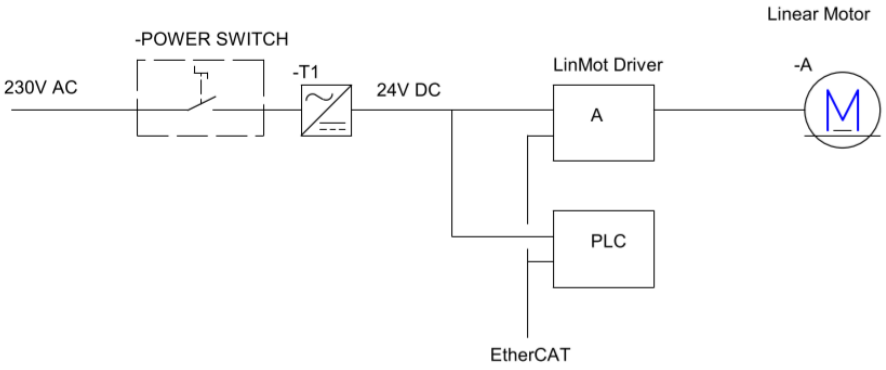


Figur 2.6: Klemfare ved Hexapod

Tap av spenning på motorer:
Vurderes som middels risiko.
Lav sjanse for at det skjer, men kan føre til personskade og skade på utstyr.
Ved tap av spenning på motorer vil det øverste planet på hexapoden falle og knekke de mekaniske sikringene, eller lande på noe.
Dette forebygges ved at maskinen kjøres til sikker posisjon ved nedstenging, og at området under det øverste planet defineres som farlig område.

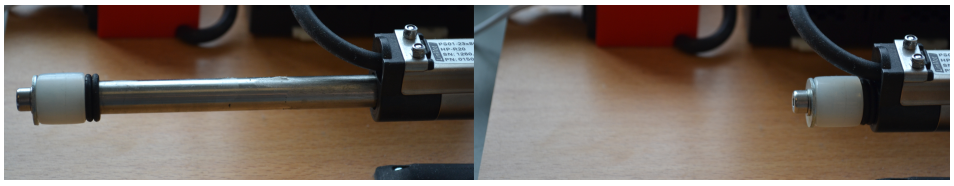
2.7 Risikovurdering av Vippe

Elektrisk støt:
Vurderes som liten risiko da roboten driftes på 24V.



Figur 2.7: Elektrisk enlinjeskjema for Vippe

Mekaniske bevegelser:
Vurderes middels risiko.
Ved tiltrekking av motorlengde vil resterende lengde stikke ut bak, og kan i noen tilfeller treffe eller klemme personer eller gjenstander. Tiltak blir å gi opplæring før bruk, dokumentering og sørge for at det er klar bane for hele robotens bane, sammen med å begrense farten til motoren for å unngå slag.



Figur 2.8: Forskjell på posisjon på vippe.

2.8 Utstyrsliste

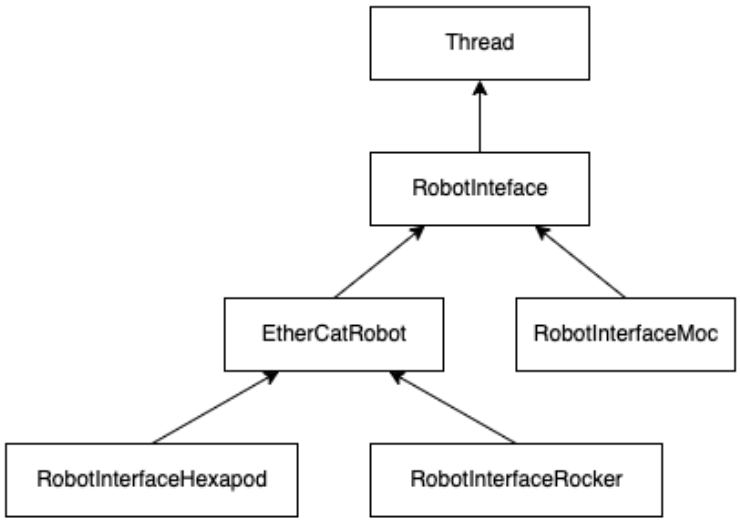
Utstyr:
Raspberry PI 4 Model B
Hexapod robot
Vippe/Rocker robot
LinMot drivere: C1150-EC-XC-0S-000
LinMot motorer: PS01-23x80F-HP-R20
PLS: OMRON NX-ECC202
Hexapod power supply: MSP-300-24
Vippe power supply: OMRON S8VK-S12024
Python

Python-biblioteker:
SOEM
Transitions
CMD2
Flask
Flask-Restfull
Flask-SocketIO
Flask-CORS
Requests
Time
datetime
Logging
JSON
Threading
Gevent
Gevent-socket

Kapittel 3

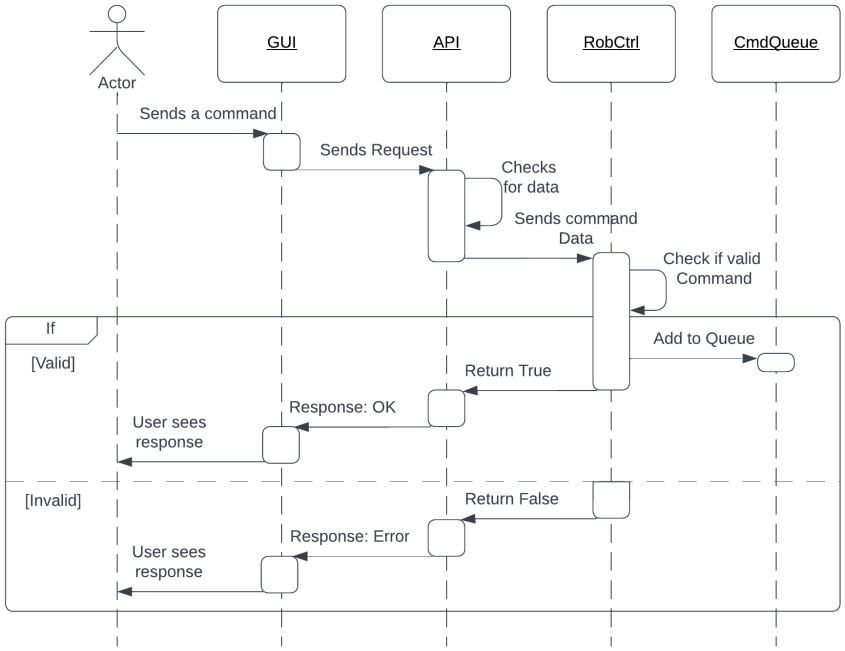
Konstruksjon

Dette programmet er et generelt kontrollprogram for å styre roboter. Ulike roboter defineres som en spesifikk klasse som arver fra en base-klasse som inneholder det meste av styring og logikk. Hvilken type som skal startes opp, avgjøres under oppstart og dette definerer hvilken klasse av *RobotInterface* som starter.



Figur 3.1: Arvingsdiagram av RobotInterface

Det er lagt opp til ulike klasser, som arver av den samme baseklassen, for å kunne ha omdefineringer av metoder og klasser for å kommunisere med ulike antall drivere og drivere med ulike navn. Det gir også muligheten til å kunne definere egne oppstarts- og stoppsekvenser. Programmet inneholder en server som tar seg av kommunikasjon mot GUI. Kommandoer sendes inn til server, deretter sendes de videre til tilstandsmaskinen i programmet. Denne tolker kommandoene, konverterer, henter ut og gjør nødvendige utrekninger og sjekker før kommandoene blir sendt til sanntidsdelen. Roboten mater tilbake hvilken posisjon motorene er i til *SignalStorage* som blir sendt videre til GUI via *DataUpdateThread*.



Figur 3.2: Sekvensdiagram for hvordan en kommando blir lagt til i køen.

3.1 Oppstart av programmet

Programmet er satt opp for å kunne kjøres med to ulike roboter. En vippe og en hexapod. Hvilken type som kjøres blir definert ved oppstart av programmet ved at en legger til parameteret --robot [type]. En kan enten velge Hexapod eller Rocker (vippe). Dette argumentet velger hvilken klasse av robotkontrolleren som skal starte opp.
Programmet kan også startes med en simulert modell av roboten, da legger en ved parameteret --sim. Se vedlegg B for flere oppstartsparameter. Når programmet startes må dette kjøres med administratorrettigheter gjennom *sudo*, slik at SOEM får tilgang til de ressursene som trengs[2].

På oppsettet for vipperoboten er det også koblet til en Omron PLS, NXECC202. Denne kan brukes til å lese av verdier fra en sensor på vippen.

Samtidig som robotkontrolleren starter, starter også serveren, hendelsesloggeren og *SignalStorage*. *SignalStorage* er en klasse som er laget for at robotkontrolleren kan oppdatere signalene de får fra hver motor, og serveren kan hente disse ut og sende dette videre til GUI. Etter dette startes kommandolinjegrensesnittet.
Hendelsesloggeren settes opp slik at alt som loggføres kan bli videresendt til GUI gjennom *DataUpdateThread* klassen. *SignalStorage* blir satt opp slik at både server og robotkontroller har tilgang til den. Oppdatering av signaler opp mot GUI skjer i *DataUpdateThread* til GUI.

3.2 Programmeringsgrensesnitt

Programmeringsgrensesnittet (API) er delen av programmet som kommuniserer med verden utenfor gjennom *HTTP*. Dette gjennom POST, PUT, PATCH, DELETE, GET og HEAD forespørsler i URL-en. http://localhost:5000/api/robot/move er en av disse URL-ene som kan motta forespørsler. I forespørselen som sendes må det legges ved en JSON formatert data streng for de lenkene som skal motta data.

Det settes også opp en *WebSocket* for å kunne strømme data fra APIet til klienten. Gjennom denne sendes robot signaler, loggføring av hendelser og en *Watchdog*. *Watchdog*-en settes opp av sikkerhetsgrunner slik at roboten skrur seg av dersom klienten kobler ut over lengre tid.

Lenkene som blir satt opp av serveren følger en fast struktur. Alle ressurser på API delen starter med */api/*, og *WebSocket* starter alltid med */ws/*.

3.2.1 Tilleggs biblioteker

Alle pakker og tilleggs biblioteker som brukes i programmet er blitt listet opp i *Requirements.txt* filen slik at dette lett kan installeres før bruk.

I programmet er det brukt rammeverket *Flask* for å starte opp en web applikasjon. Mange andre biblioteker er utviklet for å jobbe i samarbeid med dette. Integrasjon av ulik funksjonalitet som ikke følger med i standard *Flask*-rammeverket kan derfor lettere settes opp enn det ville om en utvikler dette fra bunnen av.

Det er da brukt *Flask-Restfull* for å sette opp REST funksjonalitet slik at serveren kan motta kommandoer eller sende spesifikk informasjon som blir forespurt.

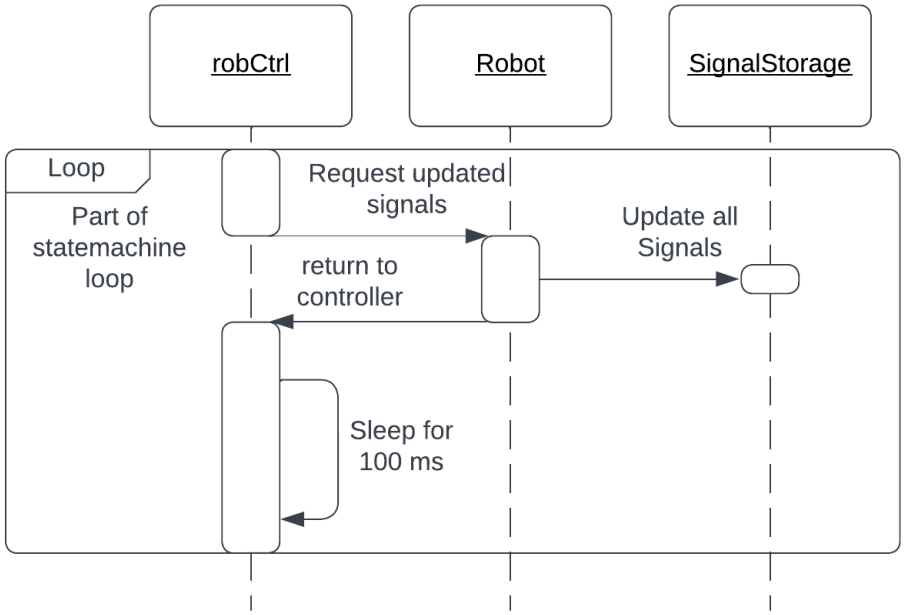
Flask-SocketIO brukes for å etablere en *WebSocket* hvor *Eventlog* og informasjon om signalene kan sendes til sesjoner uten at en forespørsel må gjennomføres for å motta dette. Denne blir også brukt mot den sesjonen som har styringsrettigheter som en *WatchDog*. Dette tillater serveren å kontakte klienten for å se at den er tilstede. Om den ikke er koblet til vil programmet skru av roboten og slå programmet av.

Flask-CORS er satt opp for å integrere *Cross Origin Resource Sharing* (CORS). Dette gjør det mulig å åpne opp for hvilke HEADERS som kan brukes av forespørsler og hvilke metoder de kan bruke, uten å måtte sette dette for hver enkel ressurs.

3.2.2 Server

SignalStorage.py

SignalStorage blir startet og er tilgjengelig for både robotkontrolleren og serveren. Denne brukes til å oppdatere de nyeste verdiene fra motorene slik at dette kan hentes og strømmes til de web-sesjonene som er koblet til via API. Denne blir oppdatert av robotkontrolleren når det er mulig og blir sendt videre til *web-socketen* gjennom klassen *DataUpdateThread*.



Figur 3.3: Hvordan oppdateringen av SignalStorage gjennomføres på slutten av hovedløkken til robotkontrolleren

Config.py

Alle innstillinger og variabler en vil kunne bestemme før oppstart av programmet er tilgjengelig gjennom en egen konfigurasjonsfil. Her kan en sette IP-adresse og port, og er en felles måte for Rest og Web-Socket til å få tilgang til selve serverinstansen eller til robotkontrolleren.

```
1    SERVER_IP = '127.0.0.1'
2    WS_PORT = 5000
```

Figur 3.4: Utdrag fra Config.py med hvordan sette API-server sin IP-adresse

3.2.3 Rest

RestInterface.py og **RestResources.py**

Når REST starter opp legges det til ressurser for hver mulige kommando som kan gjøres på *APIet*, her setter man også lenken til ressursen. Hver ressurs har en tilhørende klasse hvor det blir bestemt hva forespørsler skal gjøre, basert på metoden brukt i forespørselen. Grunnet at klassene arver fra klassen *Resource* i *Flask-Restfull* vil de som ikke blir definert i underklassen fremdeles kunne brukes, men gir tilbakemelding med statuskode 501 for at de ikke er implementert.

RestInterface.py

```
1         self.api.add_resource(RR.Move,"/api/robot/move")
```

Resources.py

```
1     class Move(Resource):
2         def get(self):
3             abort_if_serverif_doesnt_exist()
4             abort_if_robot_doesnt_exist()
5             return cfg.getRobIf().getCmdQueue(), 200
```

Figur 3.5: Eksempel på at /api/robot/move ressurs blir satt opp og at GET methoden blir implementert for denne.

Når en forespørsel kommer inn fra GUI, sendes det alltid en statuskode i respons. Disse statuskodene er standardiserte og første siffer i den 3 sifrede koden forteller hvilken statustype koden symboliserer. Koder som 1xx betyr informasjon, 2xx betyr suksess, 3xx betyr videreføring, 4xx er klientfeil og 5xx er server feil.[3] Dersom en forespørsel sendes inn og en statuskode er kommet i respons betyr dette at forespørselen er ferdig håndtert fra web-serveren, men en feil kan fremdeles oppstå i selve robotkontrolleren. Om det skulle oppstå en feil i robotkontrolleren vil klienten få vite dette gjennom hendelsesloggen, da markert enten som en feil eller advarsel.

2xx	Suksess
200	OK
201	Laget
204	Ingen innhold

4xx	Klient feil
401	Uautorisert
404	Ikke funnet
409	Konflikt
422	Kan ikke prosesseres

5xx	Server feil
501	Ikke implementert

Figur 3.6: Brukte statuskoder og deres betydning.

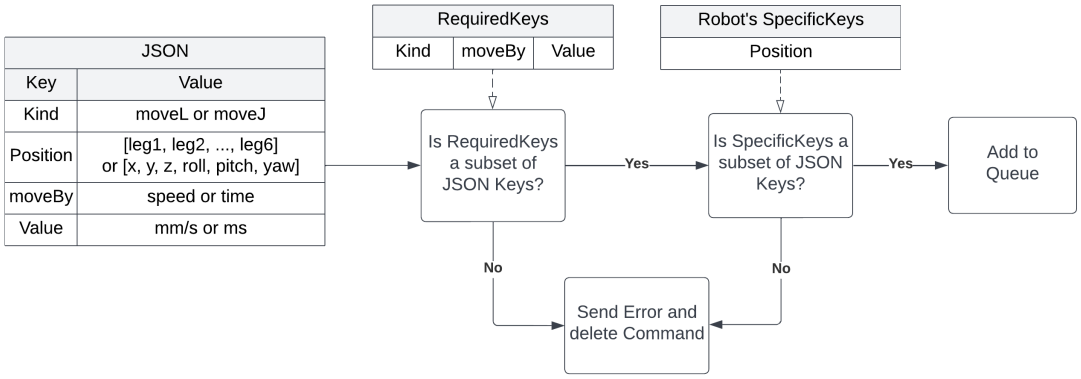
Noen forespørsler vil kreve data for å kunne gjennomføres. Disse variablene må da sendes inn til APIet i JSON format. Dette blir så konvertert om til en *python dictionary* som robotkontrolleren kan jobbe videre med. I programmet vil aldri en JSON formatert variabel sendes til robotkontrolleren ettersom det er et format kun brukt i web baserte applikasjoner. Et eksempel på en forespørsel hvor JSON formatet må brukes er på /api/robot/move med POST metoden.

```
Rocker_Example_cmd = {
    "0": {
    "angleorlength": 0,
    "moveBy": "speed",
    "value": "1200",
    "kind": "moveL"
    }
}
```

Figur 3.7: Eksempel på JSON struktur, her av Vippens ønsket oppsett.

For at en slik kommando skal kjøres må den ha de rette nøkkelverdiene for kommandoen til roboten den skal kjøre. Derfor blir disse kommandoene sendt inn til ett filter før de går videre til kommandokøen. Før kommandoen går inn til filteret sjekkes det at det faktisk er kommet med data, og respons blir sendt tilbake om det ikke fulgte med. Om det faktisk ble sendt med data blir dette sendt inn til filteret, hvor den henter ut en liste over alle nøkkelene som er i *dictionariet* og sjekker den opp mot to andre lister.

Disse listene blir definert i *RobotInterface* klassen hvor den ene settes med de nøklene det er forventet at alle kommandoene skal ha, og den andre blir satt tom slik at det kan fylles inn i klassene til robot typene for å kunne ha mer spesifike krav. Det sjekkes da at disse listene er en delmengde av listen med nøkler som kom fra forespørselen. Om de er det vil kommandoen legges til i kommandokøen. Etter dette vil statuskoden bli sendt til GUI, for om kommandoen ble lagt til i køen og om den ikke ble det kan det gi en pekepinne på hvor problemet ligger.

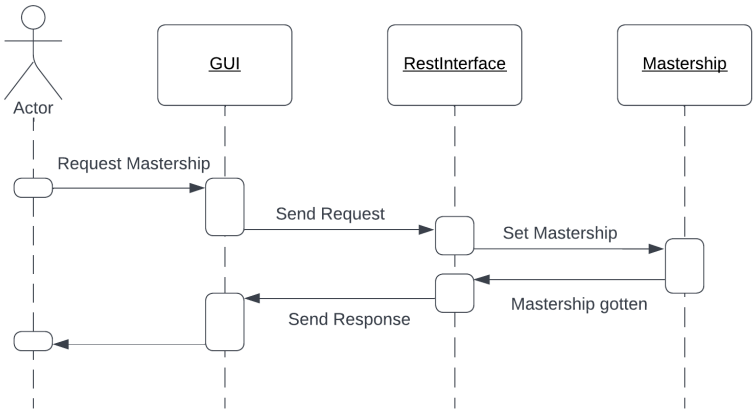


Figur 3.8: Konsept for kommandofilter

En full liste av ressurser og nøkkelene JSON-filene må inneholde ligger i vedlegg B, delkapittel 4.

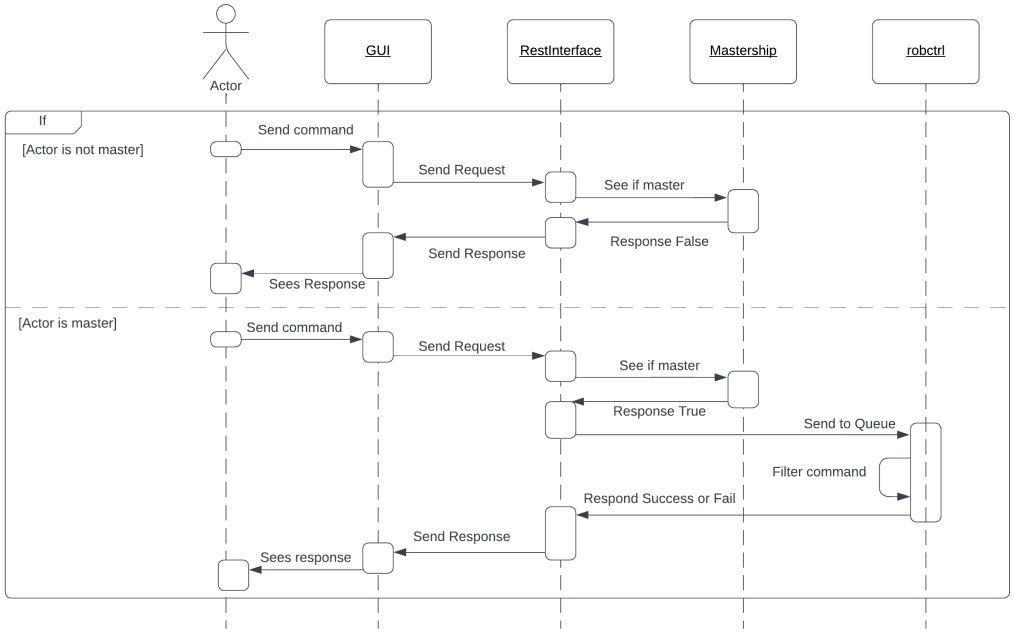
Mastership.py

Ved oppstart i kommandolinjen kan en velge at *CMD2* er eneste som har styringsrettigheter og om ikke det blir satt kan en kommando sendes for å gjøre det. Om et grafisk brukergrensesnitt kobler seg på og starter opp roboten, blir ip-adresse sendt i en forespørsel om styringsrettighet. Dersom ingen andre har styringsrettigheter lagres dette og denne klienten kan sende kommandoer til roboten. Hvis en annen klient allerede har styringsrettigheter, får klienten beskjed om at dette allerede er gitt og eneste de har tilgang til er å se informasjon og statuser.



Figur 3.9: Etablering av styringsrettighet til klienten

Når kommandoer sendes fra GUI vil API sjekke opp om klienten har styringsrettigheter, dersom klienten ikke har det vil GUI få en respons med statuskode 401, for at klienten ikke er autorisert til handlingen. Om klienten har styringsrettighet vil kommandoen kjøre videre som vanlig.



Figur 3.10: Hvordan API svarer om klient har eller ikke har styringsrettighet.

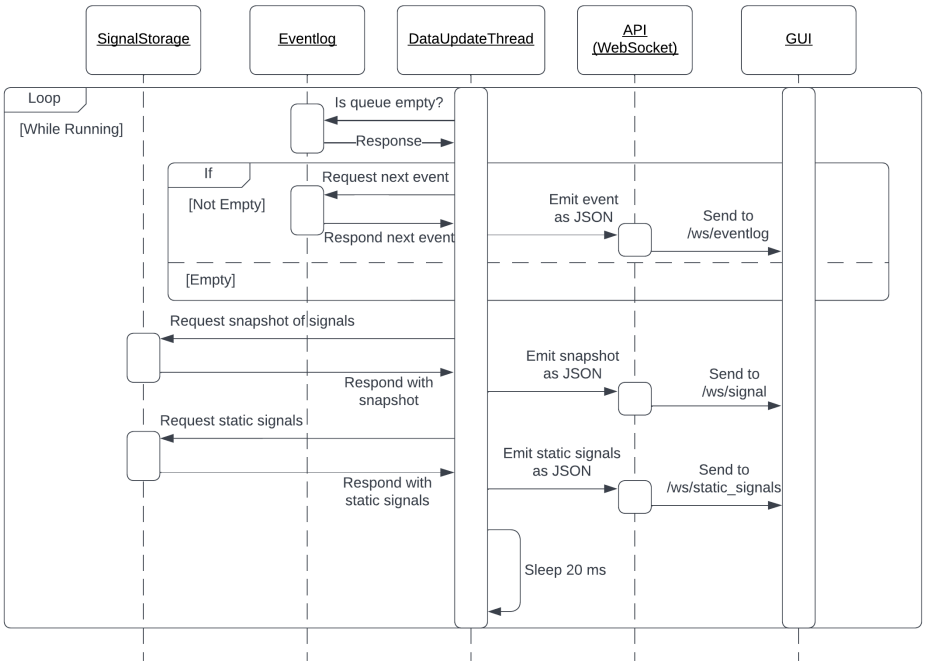
3.2.4 WebSocket

WSInterface.py

WebSocket grensesnittet er den siste delen som startes opp i serveren, dette er gjort ved hjelp av *Flask-socketio* for å ha ett lettere rammeverk å bygge opp *websocketen* med. For å starte socketen må serveren ha installert *gevent* og *gevent-websocket*. Dette er grunnet at *Flask* sin utviklingsserver ikke har støtte for websocket protokoll og det må derfor settes opp en produksjonsserver med støtte for det. Gjennom socketen strømmes hendelseslogger og signaloppdatering ved hjelp av *DataUpdateThread*, denne definerer også hvilke namespace som er tilgjengelig gjennom Websocket.
Her startes også opp en watchdog som skal monitorere at klienten med styringsrettigheter ikke kobler seg fra. Dette da ved å sende ett ping hvert femte sekund, og om ikke et pong mottas inne 25 sekunder markeres klienten som frakoblet. Dette kaller en feil i systemet som kan fikses gjennom kommandolinjen dersom klienten kobler seg på igjen innen 25 sekunder har gått etter dette.
Om ikke klienten har koblet seg opp igjen innen fristen, endres feilen til en *Fatal error* og serveren må startes på nytt for å kunne koble seg på igjen. Grunnet måten websocket regner frakoblet, vil denne feilen kun oppstå dersom det er ett avbrudd i tilkoblingen, eller GUI blir skrudd helt av. Dette er slik at roboten samsvarer best mulig med maskinforskriften.(Se kapittel: 3.7.2)

DataUpdateThread.py

DataUpdateThread klassen starter en loop for å sende data gjennom websocketen, dette er for at en kan samle trådene**?** fra *SignalStorage* og fra køen med hendelsesloggføringer. Her settes det også et pauseintervall på 20 millisekunder, dette er for å ikke overstige det avbruddsintervallet som Raspberry Pi 4 sin CPU har.



Figur 3.11: Sekvensdiagram for hendelsesforløpet til *DataUpdateThread*

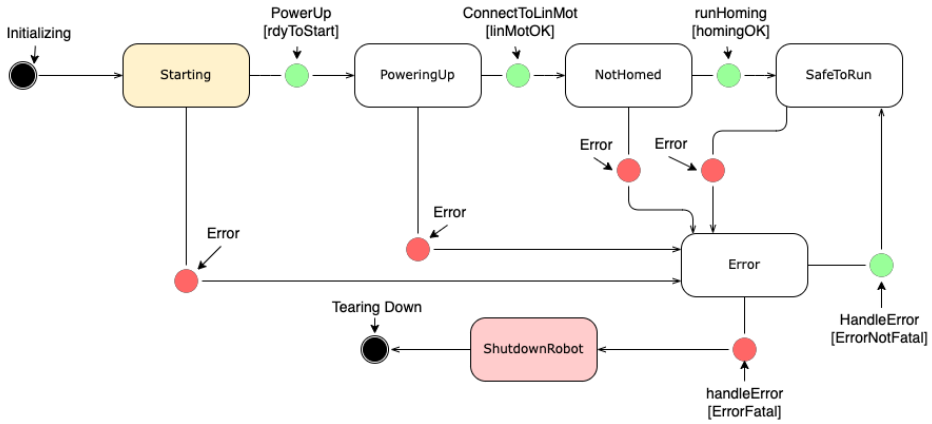
3.3 Robotkontrolleren

Robotkontrolleren inneholder en klasse som kalles *RobotInterface*. Det er denne som inneholder det nødvendige av informasjon som trengs for hver spesifikke robot og inneholder selve logikken til kontrolleren. Det er denne de spesifikke robotene arver fra når programmet starter.

Programmet inneholder to tilstandsmaskiner som styrer maskinen. Den ene er dedikert til sikkerhet og den andre tar seg av kommandoer. Disse er implementert ved hjelp av *pytransitions* bibiloteket. For å gå fra en tilstand til en annen må det defineres hvilke tilstander en skal gå fra og til, en trigger for å starte overgangen og en betingelse som må oppfylles for at overgangen skal utføres. [4] Triggerene blir kalt i hovedløkka av programmet som kjører kontinuerlig opp mot 10 ganger i sekundet.

3.3.1 Sikkerhetsmaskin

Tilstandsmaskinen som tar seg av maskinsikkerheten overvåker tilstanden til driverene og sjekker om det er sendt inn noen feilmeldinger fra serveren. Det er denne som tar seg av oppstart og nedstenging av maskinen. Når en har nådd tilstanden *SafeToRun*, vil en kunne kjøre kommandomaskinen. Så lenge det ikke er noen problemer, vil da sikkerhetsmaskinen ligge i denne tilstanden.



Figur 3.12: Tilstandskart for sikkerhetsmaskinen

Overganger er definert i kartet som *trigger[betingelse]*.
Tilstandene i denne tilstandsmaskinen er:
Starting,PoweringUp,NotHomed,SafeToRun,ShutdownRobot,Error.

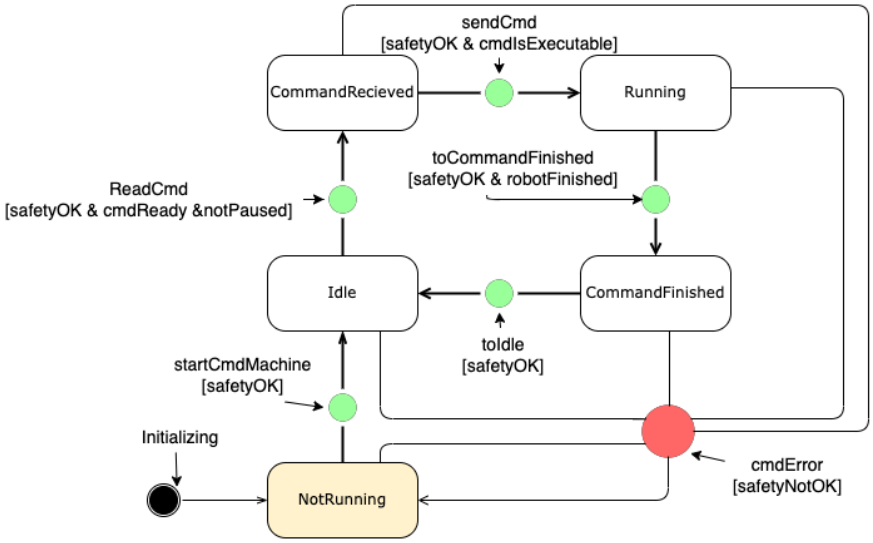
Ved oppstart starter tilstandsmaskinen i *Starting*. Deretter vil det i hovedløkka kalles en trigger for å gå videre i tilstandsmaskinen. Den neste tilstanden kan nås når betingelsen rdyToStart returnerer *True*. Denne returnerer *True* når en har trykket på *power up* knappen på GUI og bekreftet ved å kjøre kommandoen *RobotConfirmPowerUp*. Eller ved å kjøre kommandoen *RobotPowerUp* i CLI. Dette setter tilstandsmaskinen i tilstand *PoweringUp*.

Den neste tilstanden trigges automatisk og sjekker at alle LinMot driverene returnerer svar og at de har spenning. Deretter går tilstandsmaskinen videre til *NotHomed*. Betingelsen for å gå videre her er at alle driverene bekrefter at de har kjørt homing-sekvens. Ved oppstart uten strømbrudd, er det ofte mulig at driverene ikke trenger å kjøre ny homing-sekvens.

Homing-sekvensen kan utføres ved å trykke på *Run Homing* knappen i GUI, eller i CLI ved å sende kommando *RobotRunHoming*. Denne sjekker om hver enkelt motor har kjørt homing-sekvens. Dersom dette ikke er gjort, kjøres homing-sekvens. Når det er utført vil betingelsen for neste tilstand være oppfylt og tilstandsmaskinen vil gå til tilstand *SafeToRun*. Behandling av feilmeldinger er beskrevet i 3.7.2.

3.3.2 Kommandomaskin

Tilstandsmaskinen tar seg av kommandoer, tolking av kommandoer og sørger for at de blir sendt til driverne.



Figur 3.13: Tilstandskart for kommandomaskinen

Tilstandene i denne tilstandsmaskinen er:
NotRunning, Idle, CmdRecieved, Running, CommandFinished.

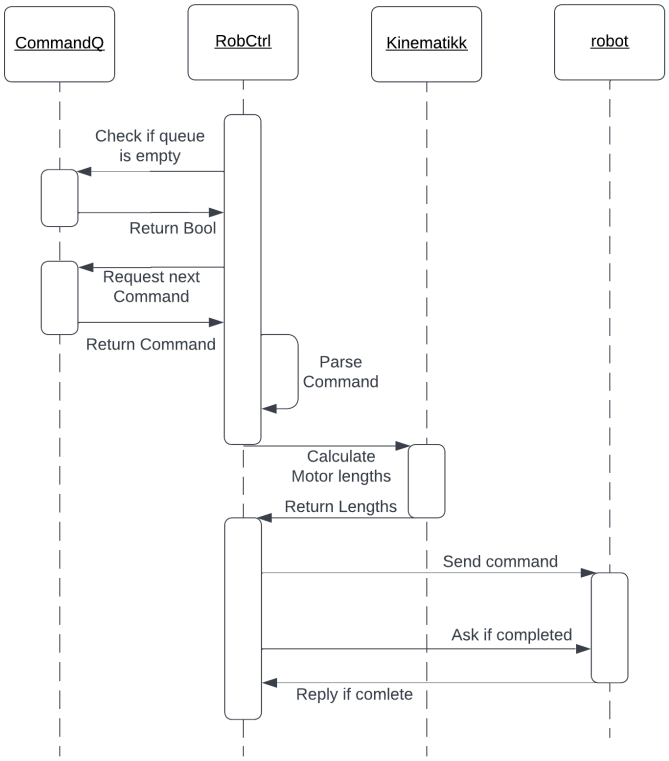
Tilstandsmaskinen starter i tilstanden *NotRunning*, og er der helt til tilstandsmaskinen for maskinsikkerheten når tilstand *SafeToRun*, da kjøres triggeren for å sette denne tilstandsmaskinen til *Idle*. Betingelsen for denne overgangen er kun at maskinsikkerhetsmaskinen er ok. Dersom denne på noe som helst tidspunkt ikke er noko, så vil tilstandsmaskinen for kommandoer gå til tilstand *NotRunning*.

Når tilstandsmaskinen er nådd *Idle*, venter den på at det skal komme kommandoer i køen. Dette er betingelsen som sender videre til *CmdRecieved*. Derpå vil kommandoen leses og tolkes før den sendes til driverene så lenge det ikke finnes noe feil med kommandoen. Dette setter tilstandsmaskinen i tilstanden *Running*.

Når alle driverene returnerer at de har nådd sin ønskede posisjon, går tilstandsmaskinen til *CommandFinished*. Deretter går tilstandsmaskinen tilbake til *Idle*, forutsatt at sikkerhetsmaskinen ikke rapporterer noen feil.

3.4 Kommandoer

Inkommende kommandoer kommer som et dictionary innpakket i json format. Det vil si at hver kommando inneholder en del nøkler, som er navnet til en verdi. En av nøkkelene som skal ligge i kommandoen er *kind*. Dette er en verdi som bestemmer hvilken type kommando som blir sendt, *moveJ* eller *moveL*. Dette brukes til å bestemme hvordan kommandoen skal tolkes før den sendes til driverene.

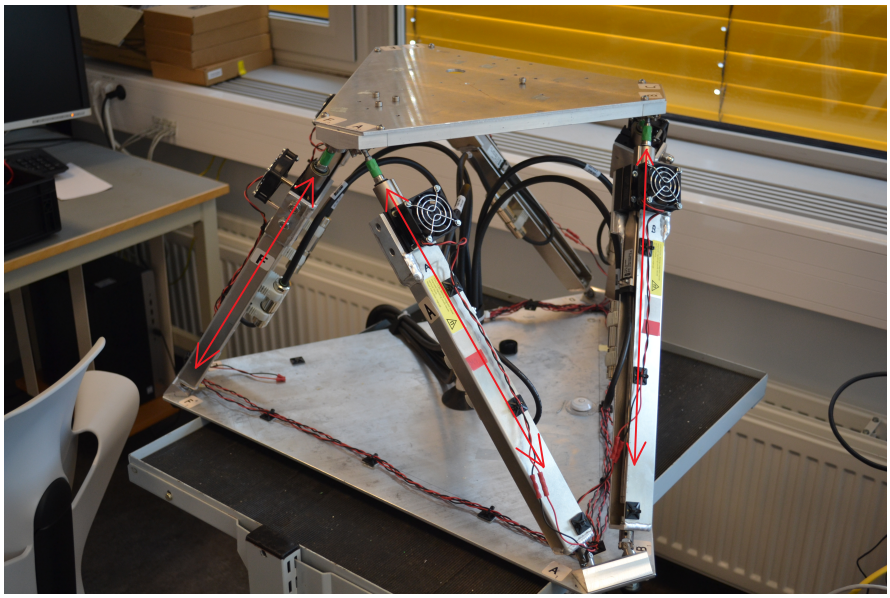


Figur 3.14: Behandling av kommando som ligger i kø

De to typene kommandoer som er tatt i bruk er *moveJ* og *moveL* som står for *move joint*, og *move linear*. De brukes til å kjøre roboten på to ulike måter.
Ved *moveJ* beveger spesifikke deler av roboten seg til en viss posisjon. Det kan være en enkelt lineær motor som skal til en viss posisjon. I denne type kommando tas ikke banen til roboten med i betraktning. For denne kommandotypen sendes det direkte ønsket posisjon på motor.
Ved *moveL* beveges roboten til en spesifikk posisjon gjennom en lineær bane. I denne type kommando sendes det en ønsket posisjon roboten skal oppnå. Dette er ulikt fra robot til robot. Vippen styres ved å sende en ønsket vinkel. På hexapoden er dette et punkt som inneholder posisjon og rotasjon, [X, Y, Z, Roll, Pitch, Yaw].

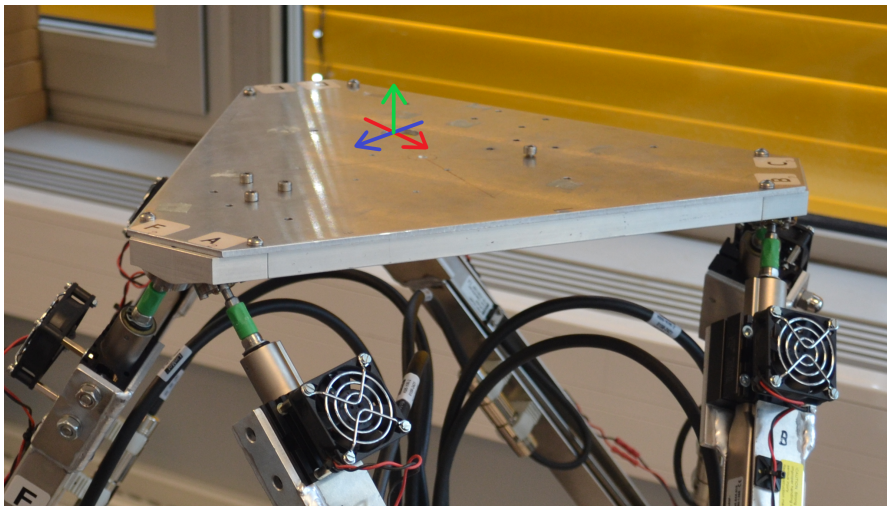
3.4.1 Hexapod

moveJ
En moveJ kommando på denne roboten er framdeles mulig uten å kunne oversette koordinater til lengder, da en setter individuell lengde på hver motor og ber roboten gå til denne posisjonen uten å ta høyde for robotens bane. Hver lengde settes i GUI og ved sendt kommando, sjekkes hver lengde at de er innenfor tillatte parametre. Dersom en eller flere er utenfor grensene, kjøres ikke kommandoen og en får en feilmelding.



Figur 3.15: moveJ beveger hver motor individuelt.

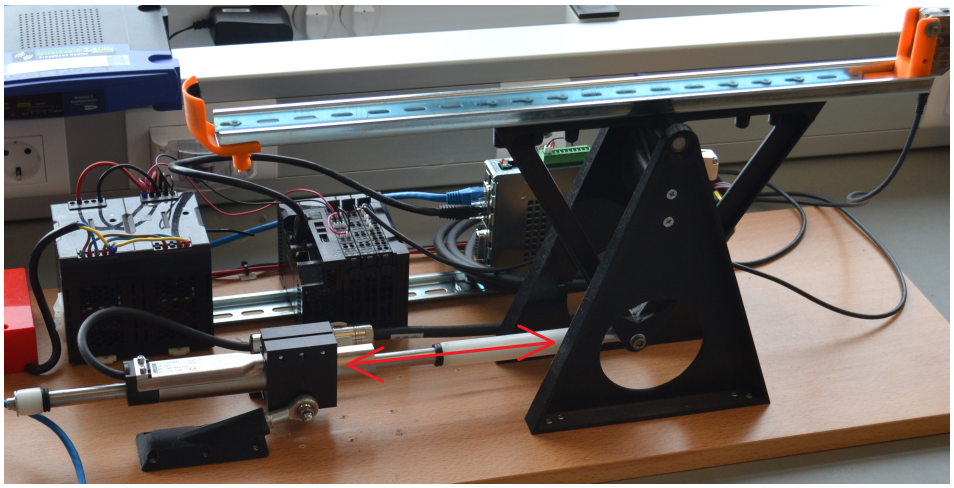
moveL
Geometri for hexapod roboten inngår ikke i denne oppgaven, men strukturen for implementering av dette eksisterer.



Figur 3.16: moveL beveger roboten til en posisjon/rotasjon

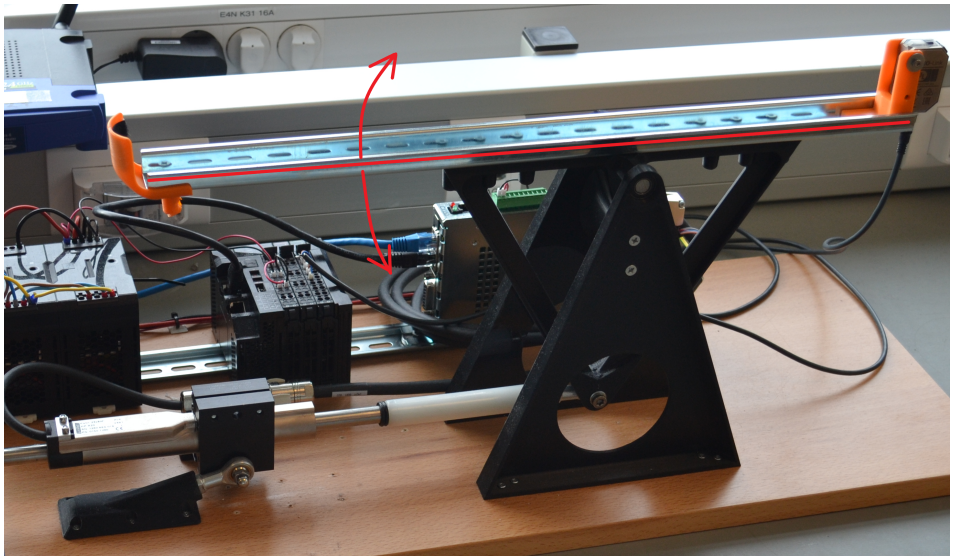
3.4.2 Vippe

moveJ
Denne type kommando sender kun en lengde til motoren uten å tenke på bevegelsen eller vinkelen på planet. Den utføres ved at en skriver inn en lengde i GUI og trykker «send». Dersom den satte lengden er innenfor de tillatte grensene, sendes det en kommando til driverene om å kjøre motoren til ønsket posisjon.



Figur 3.17: moveJ kjører motoren til en posisjon

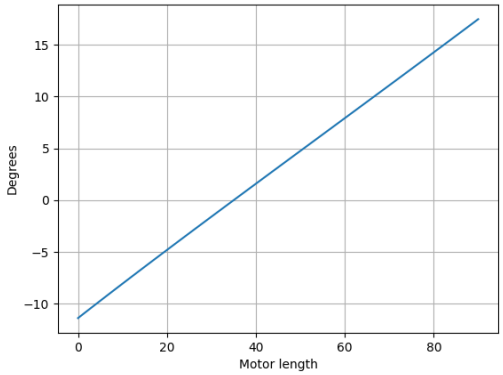
moveL
Ved en moveL kommando, settes den ønskede vinkelen, og en må finne den passende lengden basert på forholdet mellom motorlengde og vinkel.



Figur 3.18: moveL kjører vippen til ønsket vinkel

3.4 Kommandoer

Vippen er en maskin som overfører lineær bevegelse til vinkling på en skinne som er laget for å holde en ball balansert. Med utgangspunkt i tegning og målinger gjort av Karl Skretting (se fig. 2.4), eksisterer følgende forhold mellom motorlengde og vinkel på planet.



Figur 3.19: Sammenheng mellom motorlengde og vinkel på vippe.

```
1     def moveL(self,cmd):
2         newAngle = cmd['angleorlength']
3         newPosition = self._robKin.AngleToLength(newAngle)
4         self._ecatIf._etherCatSlaves['Axis1'].ExecVAIGoToPos(newPosition)
```

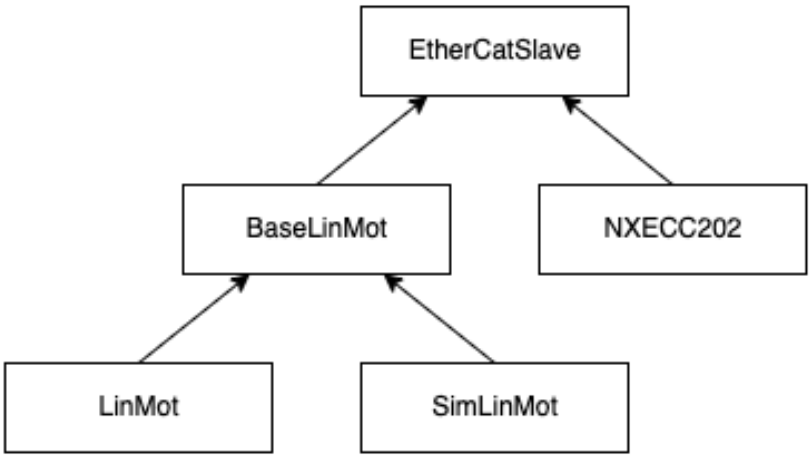
Figur 3.20: Vippe moveL kommando.

3.5 EtherCAT

Raspberry PI og robotene kommuniserer gjennom EtherCAT, som er et feltbussystem utviklet for bruk med industrielle sanntidssystemer. Det baseres på å sende telegrammer med informasjon fram og tilbake[5]. EtherCAT realiseres på Rapsberry PI ved hjelp av biblioteket *Simple Open EtherCAT Master* (SOEM)[2].

3.6 Simulert modell

Morten Mossige har bygget en simulert modell som inneholder de samme metodene som grensesnittet til LinMot for robotkontrolleren. Denne returnerer også en simulert posisjon og tilstand. Ideen bak dette er at robotkontrolleren ikke skal ha noen kjennskap til om den kjører en simulert eller reell robot, den skal behandle de to tilfellene likt.



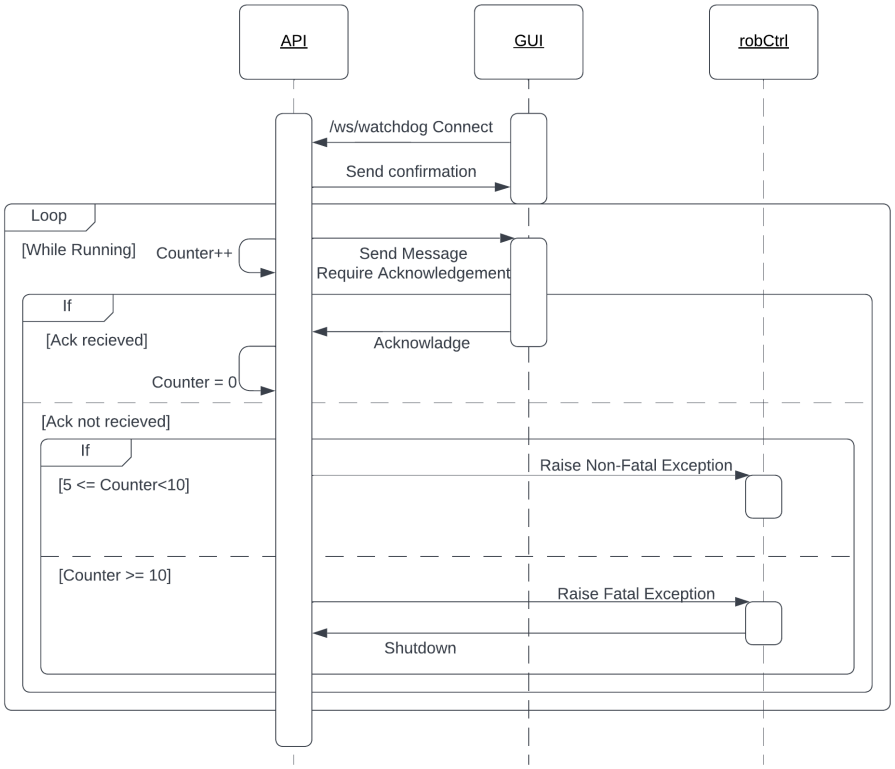
Figur 3.21: Arvingsdiagram av EtherCAT

3.7 Sikkerhet

3.7.1 Datasikkerhet

Ved enhver POST, PUT eller DELETE request som blir sendt inn til API-et vil denne sjekkes opp i mot om brukergrensesnittet har styringsrettighet. Styringsrettighet tildeles enten ved oppstart til CLI, gjennom en kommando i CLI eller gjennom at API lager en request ved tilkobling. Kun en sesjon kan ha styringsrettighet om gangen, med unntak av CLI som kan velge å være eneste eller dele med API. Dersom en ny sesjon startes vil ikke de få styringsrettigheter og kan kun få GET requests, Eventlogging og retur signaler.

En *Watchdog* blir etablert med API-sesjonen, som har styringsrettighet og etablerer et *WebSocket* rom. I dette rommet sendes det en melding med et interval definert i *Config.py*, til eier og øker en teller. Om svar kommer i retur settes telleren tilbake til 0. Om telleren passerer 5 vil eier bli koblet fra maskinen og det blir sendt en *Non-fatal Error*. Kobler sesjonen seg opp igjen innen telleren er kommet til 10 vil maskinen kunne brukes videre etter noen kall på kommandolinjegrensesnittet. Om telleren passerer 10 blir feilstatusen oppgradert til en *Fatal Error* og programvaren vil måtte startes på nytt for å kunne brukes videre.



Figur 3.22: Sekvensdiagram av *Watchdogen*.

3.7.2 Maskinsikkerhet

Basert på forskrifter og krav fra Maskinforskriftene [6]. De to robotene i dette prosjektet går under Kapittel 1, §1.2.h som en maskin bygd spesielt til forskningsformål og for midlertidig bruk i laboratorie. Det vil si at robotene faller utenfor disse forskriftene, men det legges opp til at det skal kunne følges for framtidige endringer.

Ytre påkjennelser
Strømbrudd kan føre til at maskinen faller fra posisjon. Dette er for øyeblikket uunngåelig og området under maskinen defineres dermed som et farlig område, både av den grunn og grunnet klemfare.

Feil i programmets styring og logikk
Ved feil i styresystemet som feilmeldinger, brudd i kommunikasjon og feil i kommandoer skal maskinen tolke feilen og så stoppe. Brudd i kommunikasjon vil over en bestemt tid føre til nedstenging.
Dersom en feil oppstår, plukkes denne opp og sjekkes om det er en kjent feil. Sikkerhetsmaskinen settes til *Error* og feilen sendes til behandling. Dersom feilen er kjent og av en art som har blitt bestemt ufarlig vil programmet sette en variabel som fører til at programmet vil gå tilbake til *SafeToRun* og *Idle* for de to tilstandsmaskinene, før programmet venter på ny kommando.

Dette gjøres ved å kjøre hovedløkka i en *try except*-blokk for å fange opp feilmeldinger. Ved noen tilfeller blir det gjort manuelle sjekker underveis i programmet. For å definere en feil som ufarlig, legges det ved en kode på feilmeldingen.
Dersom feilkoden er kjent og ufarlig, så vil programmet fortsette og vente på ny kommando. Dersom det ikke er lagt ved en feilkode eller feilkoden ikke er i listen over godtatte feilkoder, vil det kjøres en stopp på hele maskinen. Dette setter sikkerhetsmaskinen til *ShutdownRobot* og programmet kjører stoppsekvens.

```
1         if e.args[0]['code'] in nonFatalErrorCodes:
2             self.setErrorFatality(False)
3             logging.warning('Error is non-fatal, going back to idle.')
4         else:
5             self.setErrorFatality(True)
6             logging.error('Error is fatal, shutting down!')
```

Figur 3.23: Behandling av feilmeldinger

e = Exception
nonFatalErrorCodes er en liste med alle feilkodene som er blitt definert til å være trygge.
setErrorFataility setter en variabel som leses når det avgjøres om maskinen skal stanses eller gå tilbake til klartilstand i feilbehandling. Denne er normalt satt til å være *True*.

Kommandoer i feil format vil bli stoppet og beskjed blir sendt til GUI. Det kan være feil som at en ønsket motorlengde er utenfor definert område.

1000	Error in command, command not executed.
1001	Follow up commands functionality is not implemented.
1002	Non-fatal disconnect

Menneskelig feilhandling
Det kan med rimelighet forestilles at dersom bruker sender en kommando og det ikke skjer noe, for eksempel at det er brudd i kommunikasjonen i en kort periode, så vil bruker fort prøve å sende flere kommandoer. Det vil i vårt tilfelle ikke føre til noen problemer da kommandoene som blir sendt inneholder absolutte posisjoner som ikke er avhengige av tidligere informasjon. Det vil si at dersom det bes om å sette en motorlengde til 600 mm, så har det ikke noe å si om det blir sendt en gang, eller flere ganger.

Om bruker skulle sende en kommando med feil format eller feil innhold, som havner i køen, kan bruker pause kjøring av kommandokøen og slette kommandoen som er sendt. Det er også mulig å plassere en kommando inn i køen på en valgfri plass eller flytte en kommando til en annen posisjon. Kjøringen av Roboten må være satt på pause for å kunne flytte, slette eller å plassere en kommando i midten av køen.
Gjennom Watchdog socket tilkoblingen vil brukeren ha totalt 50 sekunder fra GUI kobles fra til å koble seg på igjen dersom de skulle miste tilkoblingen. Dette er slik at bruker ikke kan koble seg fra roboten mens den kjører.

Uventet start
For å sette spenning på motorene må en uansett kjøre en kommando på CLI, enten ved oppstart via CLI eller ved oppstart via GUI, som må bekreftes i CLI. Dette er for å bekrefte at en er fysisk tilstede.

Ved oppstart fra CLI kjøres kommandoen *RobotPowerUp*.
Ved oppstart fra GUI trykkes det på *Power ON* knappen, deretter må en kjøre kommandoen *RobotConfirmPowerUp*.
Ved stans av maskinen vil strøm til motorene fjernes. Dermed må en gå gjennom oppstartssekvensen på nytt for å kunne kjøre roboten.

Igangsetting
Den valgte betjeningsinnretningen for vår maskin blir definert i det en starter programmet. En kan enten bruke kommandolinjegrensesnittet eller grafiske grensesnitt. Kommandogrensesnittet brukes som et debug-verktøy og innehar dermed muligheten til å kunne styre, men ved normaldrift så er det kun den valgte betjeningsinnretningen som har muligheten til å sende kommandoer til maskinen.
Det vil si at dersom et nytt grafisk grensesnitt kobler seg til mens det allerede eksisterer noen som har styrerettigheter vil det nye kun ha muligheten til å observere tilstander og alle kommandoer det prøver å sende vil bli sett bort ifra.
Programmet kan også startes med kommandolinjegrensesnittet som master. Dette vil si at alle grafiske grensesnitt som kobles til etter det, ikke kan styre maskinen.

Full liste over flag som kan brukes ved oppstart finnes i vedlegg B

```
1        sudo python3 RsPiRobot.py --robot Hexapod --sim --cmd2Master
```

Figur 3.24: Eksempel på oppstart av en simluert Hexapod med CLI som eier av styringsrettigheter

Normal stopp

Normal stopp kan gjennomføres ved å lukke programmet med exit kommandoen i cmd2.
Det kan også gjøres i det grafiske grensesnittet ved å bruke *stop robot* knappen.
Ved sendt stopp melding til maskinen kjøres det en funksjon som sender det øverste planet ned til trygg posisjon. Det er en aktiv tilbakemelding til kommandovinduet som viser at maskinen holder på å stenge ned. Når maskinen er i trygg posisjon kuttes strøm slik at maskinen ikke faller og treffer noe, eller ødelegges grunnet fall.

Driftsstopp

Det er ikke definert noen driftsstopp funksjon for denne maskinen. Det skal ikke være noen tilfeller under normal drift der den ikke skal kunne skrues helt av.

Pause

Roboten kan settes i pause ved hjelp av GUI eller CLI. Dette setter en variabel som ligger som betingelse for å kunne lese kommandoer. Når da pågående kommando er ferdig å kjøre vil maskinen ikke lese inn neste kommando før den har fått beskjed om å gjennoppta operasjon.
Dette gjør at en kan legge inn flere kommandoer i køen og kjøre alle fortløpende når en sender kommando for å fortsette roboten.
RobotPause for å pause robot.
RobotResume for å gjenopta operasjon.

Kapittel 4

Resultat

4.1 Forsinkelser

Målinger er utført ved å logge når kommandoer blir mottat fra GUI og når kommandoen blir sendt ut til roboten i *LinMot.py*.

Fra 10 målinger som ble gjort, er det registrert følgende tidsforsinkelser fra en kommando er mottat til kommando blir sendt:

280ms	222ms	238ms	271ms	194ms	226ms	304ms	227ms	311ms	288ms

Figur 4.1: Forsinkelse fra kommando mottat, til kommando sendt

Dette gir en snittverdi på 256.1ms.

4.2 Kompatibilitet

Ettersom det er opprettet en LinMot klasse vil det være mulig å opprette spesifikke robotklasser for andre roboter som bruker samme drivere/motorer. For å koble til annet utstyr som kommuniserer over EtherCAT må det opprettes spesifikke klasser for kommunikasjonen mot disse.

4.2.1 Operativsystem

Dersom programmet kjører på en datamaskin med Windows operativsystem, må det installeres en virtuell maskin som Linux. Dette kan gjøres på Windows med *Windows Subsystem for Linux version 2 (WSL2)*. Den første versjonen *WSL*, har vist seg å ikke være kompatibelt med *SOEM* så det ble brukt *WSL2* i dette prosjektet.

MacOS er basert på UNIX, dette er det samme som Linux er basert på og fungerer derfor uten å måtte installere en virtuell maskin for å være kompatibel.

4.3 Video

Demonstrasjon av Vippe:
`https://youtu.be/AxgRVIJXuXc`

Demonstrasjon av Hexapod:
`https://youtu.be/XUzQpsbdWJU`

Kapittel 5

Diskusjon

5.1 Sammenligning av resultat mot original oppgave

Ganske tidlig i prosjektet viste det seg at utvikling mot hexapod-roboten bydde på utfordringer. Det ble avgjort at fokuset skulle ligge på vippen til å begynne med for å kun måtte forholde seg til en motor og en akse. Det ble utviklet en simulert modell som hadde de samme funksjonskallene som selve roboten, og mye av utviklingen av robotkontrolleren foregikk på denne modellen. Etter hvert som kommunikasjon mot driverene og GUI ble etablert ble disse implementert inn i robotkontrolleren. Det er implementert noen sikkerhetsfunksjoner, som trygg stans av roboten ved feilmeldinger.

Av de målene som var satt i starten av prosjektet, var det noen som ikke ble fullstendig fullført grunnet begrensninger i tid.

Det er mulig å styre hexapod roboten ved *moveJ* kommandoer, men *moveL* kommandoer er ikke implementert.
Strukturen er lagt til rette for at dette skal kunne implementeres etter at det er utviklet kinematikk for denne. Det er flere variabler det var planalgt å styre roboten etter, som intern eller ekstern interpolering, fart eller tid, som det ikke ble tid til å implementere.

API-et setter opp en *watchdog* slik at avbrudd på tilkoblingen vil starte prosessen den er laget for. Det å lukke vinduet til GUI telles ikke som frakobling. Problemer med nettverkstilkoblingen og lukking av GUI-serveren vil derfor være de eneste tilfellene som fremkaller de ønskede feilmeldingene.

5.2 Alternative metoder

5.2.1 Tilstandsmaskinen

Et alternativ til å bruke *pytransitions* sitt bibliotek for tilstandsmaskinene er å lage en tilstandsmaskin på egen hånd.
Fordelene med dette er at det vil være enklere å implementere spesialfunksjoner og oppnå funksjonalitet som ønsket. Det var for eksempel en del utfordringer med å få *pytransitions* til å fungere med to tilstandsmaskiner samtidig. En mulig forbedring ved å programmere dette uten bibliotek er at tilstandsmaskinen kan tilpasses formålet mer spesifikt.

For eksempel ville det vært mulig å ha definert *trigger* metodene selv og gitt de funksjonalitet utover det å styre tilstandsmaskinen. Dette kunne nok ryddet opp i programmet en del. Ved å bruke *pytransitions*, kunne ikke *trigger* metodene omdefineres uten å lage trøbbel for biblioteket.

Fordelene ved å bruke *pytransitions* er at det er et enkelt bibliotek å bruke og komme i gang med. Det inneholder mange funksjoner og sparte oss for mye tid i utviklingsfasen.

5.2.2 Køer eller lister

I koden er det brukt *SimpleQueue* for å lage en kommandokø og en kø for å videresende hendelseslogger. En alternativ måte å løse dette på er å sette opp en liste med funksjonalitet til en kø. Dette ville gjort det enklere å håndtere endringer i rekkefølgen av kommandoer og å slette innsendte kommandoer.

I prosjektet ble det vurdert som en bedre løsning å bruke *SimpleQueue* til å håndtere dette. *SimpleQueue* håndterer kappløppsituasjoner (race conditions), som kan oppstå når en bruker *Threading* selv slik at ikke rekkefølgen for kommandoer blir feil. Med *SimpleQueue* kan en bare hente inn objekter eller legge til objekter.

Et eksempel på hvorden det da må håndteres er ved å lage en ny kø som tar alle objekter utenom det ønskede objektet og legger det til i en ny kø for å slette ett objekt.

5.2.3 Timing av robotkontrolleren

Som en ser i figur 4.1 er det en del variasjon og forsinkelse i robotkontrolleren. En av hovedgrunnene til forsinkelsen er en ventefunksjon i hovedløkka på 100 ms.

Grunnen til at denne ikke er blitt gjort kortere er for å spare prosessoren for unødig arbeid. Dersom det skal sendes en strøm av kommandoer vil det være en fordel å ha raskere respons i løkka. Dette kan endres i ventefunksjonen som ligger i hovedløkka om det blir aktuelt. I dette tilfellet bør det programmeres en overvåking på prosessoren for å være sikker på at den ikke overbelastes.

5.2.4 Maskinsikkerhet

Det kan nok med fordel kunne monteres en nødstopp til dette systemet, om ikke annet enn at den er lett gjenkjennelig og alle vet hva den gjør. Om den skal kutte strøm eller kjøre roboten til sikker posisjon er et vanskeligere spørsmål.

På en side, så vil en tenke seg at det er best at maskinen går til en posisjon der den ikke kan falle på noe, og gjøre mer skade. Dette spesielt om det står noe på platen med vekt. På den andre side er det mest sannsynlig at dersom det skjer et uhell så er det fordi noe er kommet under, eller i klem i roboten. Det vil da kunne medføre større skader dersom roboten kjøres ned. I dette tilfellet vil det å kutte strøm være det rette svaret.

Ett annet alternativ er å be hexapod stoppe all bevegelse i den posisjonen den er, men for å kunne gjøre det må en ha muligheten til å avbryte kommandoer som er sendt til driverene.
For å hjelpe å unngå slike situasjoner kan det monteres utstyr som detekterer personer inne i det farlige området og som låser maskiner i posisjon om det skjer. Dette kan være laser, trykkfølsomme matter på gulvet, fysiske hindringer, osv. Det kan løses ved å bruke analoge eller digitale inn- og utganger fra Raspberry PI modulen for å koble inn fysiske brytere og sensorer.

5.3 Forslag til videre arbeid

Det er en del ting vi ser på som naturlige områder å jobbe videre med. Noen av de er mindre enn andre, men det er ting det ikke ble tid til eller ikke kunne gjennomføres av andre grunner.

- Valg av interpoleringsmetode og kunne bruke egen interpoleringskode.
Dette åpner opp for å sende en liste av koordinater til roboten som utføres fortløpende. En kan da ha baner som inkluderer flere punkter.

- Nødstoppsbryter og mulighet for å kunne avbryte bevegelser ved nødstopp.
Dette er mest for maskinsikkerhetens skyld, men det kan være lurt å ta med dersom dette skal jobbes videre med. Det bør og sees på muligheten for å avbryte en kommando som er sendt, for å oppnå raskere stans.

- Kjøre kommandoer basert på fart/tid.
Det er mulighet i LinMot å kjøre kommander basert på fart, så det vil være mulig å sette opp for dette både ved intern og ekstern interpolering. Nå endres ikke disse variablene under kjøring. API-et tar i mot disse variablene slik at dette kan settes opp når både intern og ekstern interpolering er klart.

- Hexapod moveL.
Dette bør implementeres dersom det skal kjøres baner på hexapoden.

- Async.
Integrasjon av *async* vil forbedre hendelsesforløpet til serveren. Dette vil kunne forbedre håndtering av tidsbaserte hendelser, og passe på at dette skjer med mer nøyaktighet på tidsintervaller.

- Flask-login.
Flask-login kan settes opp for å skape autoriserte brukere, som vil gi bedre tilgangskontroll. Mastership klassen kan da endres til å kobles mot en bruker istede for en IP-adresse.

- Flask-Session.
Programmet nå styrer sesjoner gjennom flask-socketio. Sesjonsobjekter er derfor vanskeligere å få tilgang til enn om sesjonskontroll blir etablert separat i programmet.

- Datasikkerhets funksjonalitet for om programmet skal legges åpent på nett.
Om programmet skal settes opp for å ligge tilgjengelig over åpent nett og ikke bare gjennom ett lokalt nettverk, kan det være ønskelig å integrere videre beskyttelse mot programmeringsgrensesnittet.

5.4 Konklusjon

Styring av robot
Styring av robotene er mulig.
Vippe kan styres med moveJ og moveL kommandoer.
Hexapod kan styres med moveJ kommandoer.
Begge robotene kan kjøres fra kommandolinjegrensesnitt.

Brukergrensesnitt
Det kan kobles til et grafisk brukergrensesnitt over nettverk for å styre og få tilbakemeldinger om roboten.
Roboten kan også styres fra kommandolinjegrensesnitt.

Simulert robot
Programmet kan kjøres mot en simulert robot for kjøring offline.

Datasikkerhet
Tilgangskontroll for å kjøre kommandoer er etablert slik at kun en klient kan sende inn kommandoer til roboten. Andre klienter vil kunne se all informasjon, men ikke sende inn kommandoer som gjør endringer i systemet eller på roboten. Dersom klienten med styringsrettighet kobler ut vil det prøves å etablere koblingen på nytt og programmet, samt roboten, vil skru seg av om dette ikke er mulig.

Maskinsikkerhet
Programmet må startes på Raspberry PI og det skrives inn kommando i CLI for å aktivere motorer.
Det er en egen tilstandsmaskin som sjekker at det ikke er noen feilmeldinger fra driverene og som stopper roboten ved feil.
Det er lagt inn stoppsekvens som kjører roboten til sikker posisjon ved stopp.

Bibliografi

[1] T. E. S. Andreas Kverneland, Ole André Hansen, "6dof parallell manipulator med hiv- og rullkompenserende krane," tech. rep., 2016. BsC, UiS, 2016.

[2] "Simple open ethercat master or soem, 2022. [online]. hentet fra:." https://openethercatsociety.github.io/doc/soem/index.html.

[3] "Http methods, 2021. [online]. hentet fra:." https://restfulapi.net/http-methods/.

[4] "pytransitions, 2022. [online]. hentet fra:." https://github.com/pytransitions/transitions.

[5] "Ethercat - the ethernet fieldbus, 2022. [online]. hentet fra:." https://www.ethercat.org/en/technology.html.

[6] "Forskrift om maskiner, 2022. [online]. hentet fra:." https://lovdata.no/dokument/SF/forskrift/2009-05-20-544.

Vedlegg A

Program

Programkode er lagt ved besvarelsen i .7z format. (Lastet ned fra GitHub-repository 14.05.2022)

Vedlegg A - .7z av kildekode

Vedlegg B

Bruksanvisning

B.1 Installering av software

1. Installer WSL2 for å simulere samme miljø som Raspberry PI

2. Installer Visual Studio Code og python.

3. Clone repository

4. Installer alle pakker fra requirements.txt

5. Bygg SOEM som shared, se rspi/README.md

B.2 Oppstart av programmet

1. Naviger til /rspi

2. start programmet som sudo med ønskede instillinger

```
1          sudo python3 RsPiRobot.py --robot Hexapod --sim
```

Figur B.1: Oppstart av simulert hexapod

Tilgjengelige instillinger for oppstart:

Flag	Data	Funksjon
--robot	[Rocker","Hexapod"]	Valg av type robot.
--sim		Starter simulert robot
--cmd2Master		Gir CLI mastership
--servotime	integer	Ikke i bruk

3. Skru strøm på motorene
Enten kjør kommando *RobotPowerUp*, eller trykk på *Power ON* i GUI og kjør kommando *RobotConfimPowerUp*.

4. Kjør homing-sekvens
Enten kjør kommando *RobotRunHoming*, eller trykk på *Run Homing* i GUI.

B.3 Oppstart av GUI

Sjå /spa/README.md for installasjon av GUI.

IP adresser kan settes i config fil som kan finnes i /spa/public/config.json og i rspi/Server/Resources/config.py

B.4 Bruk av API

Før oppstart må IP-addresse og port settes opp i *config.py* .

B.4.1 Liste av Ressurser og tilgjengelige HEADER

Alle REST ressurser startet med */api/* og alle *websocket* ressurser starter med */ws/* .

Dersom noe er markert med AUTH, betyr det at klienten må ha styringsrettigheter for å kunne kjøre kommandoen.

Om det er markert med JSON, krever denne kommandoen inndata i form av en JSON fil. Nødvendige nøkler til hver kommando er listet i B.4.2 - JSON .

REST ressurser

[ip-adresse]:[port]/api/master

POST - gir styringsrettigheter om ikke dette er gitt til en annen klient, eller låst til CLI.

[ip-adresse]:[port]/api/shutdown

GET - AUTH - Skrur av programmet.

[ip-adresse]:[port]/api/signals/subscription

GET - Henter ut trestrukturen til signalene.
POST - JSON - Legger til motor signal til å få oppdateringer på.
DELETE - JSON - Fjerner motor signal oppdateringer på ønsker motor.

[ip-adresse]:[port]/api/signals/snapshot

GET - Henter ut nyeste oppdaterte posisjon for hver motor.

[ip-adresse]:[port]/api/robot/info

GET - Henter ut informasjon om hvilken robot type programmet er startet for og servo-tiden til denne.

[ip-adresse]:[port]/api/robot/status

GET - Sender informasjon om robotkontrolleren. Dette er da SafetyState, CommandState, Antall kommandoer i køen og Status på motorer.

[ip-adresse]:[port]/api/robot/PowerON

POST - AUTH - Dersom en klient er koblet til kan denne brukes til å starte opp roboten. Det kreves fremdeles godkjenning via CLI for at roboten starter opp.

[ip-adresse]:[port]/api/robot/home

POST - AUTH - Starter homing prosedyren på roboten som må gjøres før bruk.

[ip-adresse]:[port]/api/robot/pause-queue

POST - AUTH - Stopper kjøring av kommandoer. Den vil fullføre en kommando om den er påbegynt.
DELETE - AUTH - Starter opp igjen kjøring av kommandoer om den er pauset.

[ip-adresse]:[port]/api/robot/move

GET - Henter ut en liste av kommando køen.
POST - JSON, AUTH - Legger til en kommando bakerst i køen.
PUT - JSON, AUTH - Plasserer en kommando til en valgfri indeks.
PATCH - JSON, AUTH - Flytter en kommando til en valgfri indeks.
DELETE - JSON, AUTH - Fjerner kommando-en på ønsket indeks.

WebSocket ressurser

[ip-adresse]:[port]/ws/eventlog

[ip-adresse]:[port]/ws/watchdog

[ip-adresse]:[port]/ws/signal

[ip-adresse]:[port]/ws/static_signals

B.4.2 JSON

JSON struktur for kommandoer til [ip-adresse]:[port]/api/signals/subscription

POST metoden vil ha indeks for signalet som skal abonneres på.

```
"signal": indeks
```

DELETE metoden vil ha indeks for signalet abonnementet skal avsluttes på.

```
"signal": indeks
```

JSON struktur for kommandoer til [ip-adresse]:[port]/api/robot/move

Kommando struktur til Hexapod - moveL og moveJ

```
"0": {
    "kind": "moveL"
    "position": [0,0,0,10,1,0],
    "moveBy": "speed",
    "value": "1200",
    },
"1": {
    "rpyxyz": [0,0,0,10,5,0],
    "moveBy": "speed",
    "value": "1200",
    "kind": "moveL"
}
```

Kommando struktur til Rocker (vippe) - moveL og moveJ

```
"0": {
    "angleorlength": 0,
    "moveBy": "speed",
    "value": "1200",
    "kind": "moveL"
}
```

POST metoden vil ha inn JSON data som en av kommandostrukturene

DELETE metoden vil ha en indeks for ønsket objekt å slette.

```
    "value": indeks
```

PUT metoden vil ha en indeks for ønsket plassering og en kommando som skal puttes inn der.

```
    "value": indeks,
    "cmd": kommando formatert som kommando strukturen ønsket for roboten
```

PATCH metoden vil ha to verdier for fra indeks og til indeks.

```
    "to": indeks,
    "from": indeks
```