BACHELOROPPGAVE:

BO19E-15
Sensorbil for praktisk anvendelse av tverrfaglig teori

Bjørnar Alvestad
Erlend Lone
Benjamin Odland Skare

30. mai. 2019

Dokumentkontroll

Rapportens tittel: BO19E-15 Sensorbil for praktisk anvendelse av tverrfaglig teori BO19E-15 Sensor vehicle for use by students in practical application of interdisciplinary skills	*Dato/Versjon* 30. mai. 2019/1.0 *Rapportnummer:* BO19E-15
Forfatter(e): Bjørnar Alvestad Erlend Lone Benjamin Odland Skare	*Studieretning:* HEAU16 *Antall sider m/vedlegg* 92
Høgskolens veileder: Adis Hodzic	*Gradering:* Åpen
Eventuelle merknader: Vi tillater at oppgaven kan publiseres.	

Oppdragsgiver: Høgskulen på Vestlandet avd. Bergen	*Oppdragsgivers referanse:*
Oppdragsgivers kontaktperson(er) (inklusiv kontaktinformasjon): Adis Hodzic – adis.hodzic@hvl.no	

Revisjon	**Dato**	**Status**	**Utført av**
1.0	30.05.2019	Endelig versjon av rapport	Alvestad, Lone og Odland Skare

Forord

Denne rapporten er skrevet i forbindelse med vår avsluttende bacheloroppgave ved Høgskulen på Vestlandet (HVL) avdeling Bergen våren 2019. Oppgaven har bestått av utvikling av en sensorbil som skal benyttes i undervisning- og laboratoriesammenheng på Institutt for elektrofag ved HVL.

Vi vil takke Knut Øvsthus, professor ved Institutt for elektrofag, som introduserte oss for bilen høsten 2018 og har gitt oss frie tøyler i videreutviklingen av denne. Institutt for datafag ved HVL har latt oss bruke 3D-printeren sin, og fortjener en stor takk for dette. Vi vil takke overingeniør Ørjan Sæbø ved IT-avdelingen på HVL som har sørget for at de ulike datamaskinene vi har brukt under utviklingen av den endelige løsningen har fast IP-adresse på HVLGuest, samt gjort det mulig å kommunisere med disse når man er tilkoblet Eduroam. Takk til alle som har hjulpet oss med deler, verktøy og tips.

Til slutt vil vi takke veilederen vår, Adis Hodzic, som gjennom hele arbeidsperioden har kommet med gode tips og innspill. Han har vist stor interesse for oppgaven helt fra starten av, noe som har vært en god motivasjon for oss.

Sammendrag

Dette bachelorprosjektet har gått ut på å utvikle et læringsverktøy for studenter som skal drive med programmering av automatiske og autonome systemer på HVL. Et slikt system kan bestå av en datamaskin som gjør beregninger, sensorer som samler inn data fra omgivelsene og motorer som skal gjøre en bestemt handling. Eksempler på dette er robotstøvsugere og selvkjørende biler og busser.

Hensikten med oppgaven er å knytte sammen flere emner som studentene tar i sitt utdanningsløp, som blant annet robotikk og reguleringsteknikk. Ved å ha et ferdigutviklet hovedprogram på sensorbilen som tar seg av kommunikasjon og innsamling av data fra sensorer, kan studentene fokusere på å skrive kode som styrer bilen, samt teorien fra de forskjellige emnene.

Det har blitt utviklet en sensorbil med forskjellige sensorer på, som kan programmeres med det samme språket studentene lærer på automatiseringslinjen. Studenter programmerer på sin egen PC og sender koden trådløst over WiFi til sensorbilen som er koblet til skolens nettverk. Om man ikke har den fysiske bilen tilgjengelig, kan koden kjøres lokalt på PC-en opp mot en simulator. Simulatoren har et kart over en korridor på HVL hvor man kan plassere en simulert bil og få testet kontrollogikken man har skrevet.

Prosjektet har bestått av utvikling og testing av maskinvare (hardware), hovedprogram og simulator. Maskinvareutviklingen har gått i flere steg, og startet med å få testet maskinvaren opp mot programvare. Videre ble designet på bilen utviklet, før den endelige versjonen ble laget ved hjelp av 3D-printer, laserkutter og lodding av komponenter på kretskort.

Hovedprogrammet som kjører på bilens datamaskin kan betjenes med en touchskjerm. Her kan man prøve de ulike funksjonene på bilen, starte og stoppe kontrollogikker og se hvordan de ulike sensorene fungerer.

En simulator ble laget for at det skal være mulig å få testet kontrollogikken man skriver uten å ha sensorbilen tilgjengelig. Det ble utført tester og målinger for å få den simulerte bilen til å oppføre seg likt som den fysiske bilen. Dette for at verdiene som kommer fra sensorene i simulatoren skal være så virkelighetsnære som mulig. Simulatoren var også til stor hjelp under utvikling av hovedprogrammet.

Prosjektet resulterte i et produkt som dekker kravene, og det er lagt til flere ekstrafunksjoner. Både programvaren og maskinvaren er laget for at det skal være mulig å legge til flere moduler og sensorer. Derfor har bilen et godt utgangspunkt for fremtidige laboratorie- og/eller bacheloroppgaver.

Forenklet innholdsfortegnelse

Dokumentkontroll 2
Forord 3
Sammendrag 4
Forenklet innholdsfortegnelse 5
Innholdsfortegnelse for hovedkapittel 6
Innholdsfortegnelse for appendiks 8
1 Innledning 9
1.1 Organisering av rapporten 9
1.2 Oppdragsgiver 9
1.3 Problemstilling 10
1.4 Ønsket situasjon for den nye bilen 10
2 Kravspesifikasjon og analyse av problemstilling 11
2.1 Overordnede designvalg basert på kravspesifikasjonene 11
2.2 Om vi får tid – ekstrafunksjonalitet utover kravspesifikasjonen 11
3 Organisering og utvikling 12
3.1 Organisering i gruppen 12
3.2 Prototype 1 (v.0.1) 13
3.3 Prototype 2 (v.0.2) 14
3.4 Sensorbil v.1.0 14
4 Systemets oppbygning og bruksområde 15
4.1 Hovedprogrammet 15
4.2 Simulatoren 23
4.3 Andre programmer 24
4.4 Systemoversikt 26
5 Testing 28
5.1 Testing av kode 28
5.2 Test av sensorer 28
5.3 Tester for simulatorfysikk 29
5.4 Strømmålinger 33
6 Oppsummering 34
6.1 Forslag til forbedringer 35
7 Referanser 36

Innholdsfortegnelse for hovedkapittel

Dokumentkontroll 2
Forord 3
Sammendrag 4
Forenklet innholdsfortegnelse 5
Innholdsfortegnelse for hovedkapittel 6
Innholdsfortegnelse for appendiks 8
1 Innledning 9
1.1 Organisering av rapporten 9
1.1.1 Hovedkapitlene i rapporten 9
1.1.2 Appendiksene 9
1.2 Oppdragsgiver 9
1.3 Problemstilling 10
1.4 Ønsket situasjon for den nye bilen 10
2 Kravspesifikasjon og analyse av problemstilling 11
2.1 Overordnede designvalg basert på kravspesifikasjonene 11
2.2 Om vi får tid – ekstrafunksjonalitet utover kravspesifikasjonen 11
3 Organisering og utvikling 12
3.1 Organisering i gruppen 12
3.1.1 Arbeidssted og verktøy for samarbeid 12
3.1.2 Fremdriftsplan 12
3.1.3 Risiko 13
3.2 Prototype 1 (v.0.1) 13
3.3 Prototype 2 (v.0.2) 14
3.4 Sensorbil v.1.0 14
4 Systemets oppbygning og bruksområde 15
4.1 Hovedprogrammet 15
4.1.1 Navigasjonsmeny 16
4.1.2 Kontrollogikk 16
4.1.3 Observasjon av sensordata 18
4.1.4 Strømstyring 20
4.1.5 Innstillinger 21
4.1.6 Unntakshåndtering 22
4.2 Simulatoren 23

4.2.1 Bruk av simulatoren 24
4.3 Andre programmer 24
4.3.1 Program med ekstrafunksjonalitet - videostrømming på nettside 24
4.3.2 Androidprogram for fjernstyring 25
4.3.3 Mikrokontrollerkode 25
4.4 Systemoversikt 26
5 Testing 28
5.1 Testing av kode 28
5.2 Test av sensorer 28
5.2.1 Lidar 28
5.2.2 Ultralyd 28
5.2.3 Encoder 29
5.3 Tester for simulatorfysikk 29
5.3.1 Rettlinje-test 30
5.3.2 Sirkelbane-test 31
5.3.3 Rotasjonstest 32
5.4 Strømmålinger 33
6 Oppsummering 34
6.1 Forslag til forbedringer 35
7 Referanser 36

Innholdsfortegnelse for appendiks

Appendiks A Lister 40
A.1 Forkortelser og ordforklaringer 40
A.2 Figurliste 40
Appendiks B Git – versjonskontroll og samarbeid på kode 42
B.1 Bruk av GitHub i Visual Studio 42
Appendiks C Brukerdokumentasjon 44
C.1 Installasjon av nødvendig programvare 44
C.2 Lage egen kontrollogikk 45
C.3 Kjøring av hovedprogrammet 46
C.4 Brukergrensesnitt for program 47
C.5 Ekstrainformasjon til brukere 48
Appendiks D Maskinvare – Bilens oppbygging 49
D.1 Prosesseringsenheter 49
D.2 Sensorer 53
D.3 Fremdrift 56
D.4 Strømforsyning 58
D.5 Karosseri 60
Appendiks E Software – Hovedprogram 63
E.1 Prosjektoversikt 64
E.2 Klasseoversikt 68
E.3 Hvordan alt er knyttet sammen i App.xaml.cs (IoC og DI) 75
E.4 «Student Logic» - Studentenes kontrollogikk 78
E.5 «SocketServer» - bruk og kommandoer 80
Appendiks F Software – Simulator 83
F.1 Prosjekt og klasseoversikt 83
F.2 Lage kart for simulator med «Tiled» 84
Appendiks G Software – Terminologi og benyttede prinsipper 87
G.1 UWP – Universal Windows Platform 88
G.2 Windows Template Studio 89
G.3 MVVM – Model-View-ViewModel (med Prism/Template10) 90
G.4 Løs kobling mellom modellene 91

1 Innledning

1.1 Organisering av rapporten

Ettersom arbeidet med denne bacheloroppgaven har bestått av en betydelig mengde maskinvaredesign/-utvikling og programmering, vil det bli overveldende for lesere uten teknisk bakgrunn dersom alt arbeidet skal bli presentert i hoveddelen av rapporten. Derfor står de utfyllende tekniske detaljene i appendiksene.

1.1.1 Hovedkapitlene i rapporten

Hovedkapitlene er skrevet på en slik at vi går fra problemstillingen, til ønsket situasjon og kravspesifikasjon, for så å vise utviklingsprosessen for å komme dit. Deretter beskrives det hvordan sluttproduktet ser ut.

Utviklingsprosessen og systemets oppbygning vil bare bli presentert på et forenklet nivå, ettersom vi ønsker at det fortsatt skal være forståelig for ikke-teknologer. I kapittelet om systemets oppbygning finner man et enkelt eksempel for bruksområde til bilen.

Til slutt er det et kapittel om testing. Dette kan oppfattes som en del mer teknisk enn de foregående kapitlene.

1.1.2 Appendiksene

I appendiksene finnes blant annet brukerdokumentasjon som beskriver hvordan man kan skrive egen kode som styrer bilen (heretter kalt kontrollogikk). Her finnes det også en mer teknisk forklaring på bilens fysiske oppbygning og hvordan programvaren er laget.

For de som ønsker å dykke enda dypere inn i det som er gjort, så er alt av 3D-tegninger, kretsdiagram og kode er tilgjengelig på GitHub repositoriet «BO19E-15_SensorVehicle». Her finner man også versjonshistorikk hvor man kan se hvordan programmene har utviklet seg. En kopi av alt innholdet fra repositoriet blir levert sammen med rapporten.

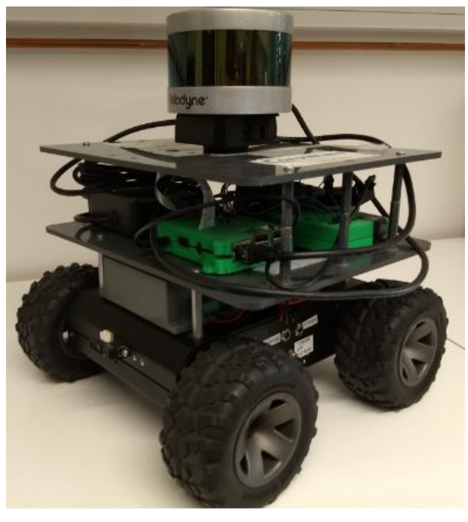
1.2 Oppdragsgiver

Under følger en kort introduksjon av oppdragsgiveren vår, Høgskulen på Vestlandet (avd. Bergen).

Høgskolen i Bergen, Høgskulen i Sogn og Fjordane og Høgskulen Stord/Haugesund ble 1. januar 2017 fusjonert til Høgskulen på Vestlandet (HVL). Totalt har HVL rundt 16 000 studenter, fordelt på studiestedene i Bergen, Sogndal, Førde, Stord og Haugesund [1]. HVL avdeling Bergen har to campuser, en som er lokalisert på Kronstad og en i Møllendalsveien. HVL har rundt 8000 studenter og rundt 900 ansatte i Bergen. Skolen består av fire fakulteter med tilhørende institutter. Vår oppdragsgiver er Fakultet for ingeniør- og naturvitenskap (FIN), institutt for elektrofag.

1.3 Problemstilling

HVL hadde en «Lidarbil» som ble brukt i emnet om robotikk høsten 2018. Den brukte en avansert sensor som kan måle avstand 360° rundt seg og 30° i høyden. Sensoren fungerer på samme måten som en radar, men mens en radar sender ut radiobølger, sender sensoren på bilen ut lysstråler (light), og blir derfor kalt lidar[1].

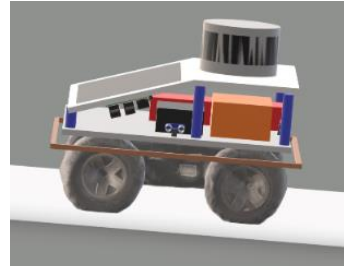


Figur 1: Opprinnelig bil. Bilde tatt høst 2018

Bilen hadde et stort potensial i emnet ved at studenter kunne hente avstander fra lidaren og lage sin egen kontrollogikk som styrer bilen. Dessverre var det noen begrensninger. Lidaren har et måleområde på 1-100 meter, noe som betyr at man ikke har gyldige målinger for avstander under 1 meter. Dette er ikke ideelt når bilen i hovedsak skal kjøres innendørs. Den brukte lidaren som eneste avstandsmåler, og dataene kom ofte for sent til programmet som styrer bilen. Studentene hadde heller ingen mulighet å teste kontrollogikken sin uten at de hadde bilen til stede. Kontrollogikken kjørte lokalt på studentens egen PC, mens lidardata og hjulkommandoer ble sendt over nettverket mellom PC og bil. Dersom bilen kjørte mellom aksesspunkt ble det brudd i kommunikasjonen, noe som skapte problemer for studentenes kontrollogikk. Det ble brukt et programmeringsspråk som er annerledes enn det studentene lærer på HVL, noe som var uheldig da det tok bort fokuset fra det man ønsket å oppnå med bilen.

1.4 Ønsket situasjon for den nye bilen

Det er ønskelig at bilen skal være et læringsverktøy som studentene kan bruke for å se hvordan teori de har lært kan brukes i praksis. I hovedsak er dette teori innenfor C#-programmering, robotikk og robotikk-relatert programmering, reguleringsteknikk og instrumentering. Terskelen for å bruke bilen bør være lav, noe som forhåpentligvis vil resultere i at studenter vil eksperimentere med den på eget initiativ.



Figur 2: Første skisse av ønsket design

Siden en vanlig klasse gjerne består av 25-30 studenter er det også ønskelig at det blir laget et simuleringsverktøy. Dette er ikke et krav, men vil være til stor hjelp da det vil være vanskelig at flere grupper skal teste koden sin på bilen uten at dette medfører mye venting.

[1] Light Detection and Ranging

2 Kravspesifikasjon og analyse av problemstilling

Alle i gruppen hadde robotikkemnet som benyttet bilen høsten 2018, og det er her idéen for oppgaven kommer fra. Sent på høstsemesteret ble oppgaven definert av et medlem i gruppen, og kravspesifikasjonene ble så satt av gruppen i felleskap.

Det ble satt krav om at det skal bli enklere å kjøre kontrollogikken direkte på bilen. Bilen skal være utstyrt med forskjellige sensorer som studentene kan hente data fra og bruke med grunnleggende prinsipper fra forskjellige emner. Studentene skal kunne bruke programmeringsspråket C# til å styre bilen, og det skal bli laget programvare som gjør selve styringen av bilen mer intuitiv. Dette fordi studentene allerede lærer C# i flere emner og at de skal kunne fokusere på å anvende det de har lært.

2.1 Overordnede designvalg basert på kravspesifikasjonene

For at studentene skal kunne skrive kontrollogikk i C# og enkelt kjøre denne på bilen, har vi valgt å bruke operativsystemet Windows 10 IoT[2] Core. Dette begrenser hvilke datamaskiner som kan brukes. Vi valgte Raspberry Pi (RPi) ettersom gruppen hadde kjennskap til denne og den var relativt enkelt å få tak i. RPi er en kompakt datamaskin som er bygget på ett enkelt kretskort. Den opprinnelige bilen benyttet en RPi 3B+. Ettersom Windows 10 IoT Core ikke støtter denne versjonen av RPi, valgte vi å bruke RPi 3B, en tidligere versjon. Valget av operativsystem fører også til at studentene ved få tastetrykk kan koble seg til bilen.

For å sørge for at man har gyldige avstandsmålinger på korte avstander benyttes ultralydsensorer, som sender ut en rekke lydbølger og lytter på refleksjonen. Ved å måle tiden det tar, kan man regne seg frem til avstanden. Sensorene som er brukt er HC-SR04 og har et målområde på 2-400 cm [2].

Bruken av gjennomsiktige plater og det at komponentene står uten «beskyttelse» er et bevisst valg. Dette gjør det betydelig lettere for studentene å se hvilke komponenter bilen består av og hvordan disse henger sammen, sammenlignet med om komponentene hadde blitt plassert inne i en beskyttende boks man ikke kunne se gjennom.

2.2 Om vi får tid – ekstrafunksjonalitet utover kravspesifikasjonen

Dersom kravene er oppfylt i god tid før fristen, vil vi se på mulighetene til å legge inn gyroskop for å registrere bilens orientering, encoder for å vite hvor langt bilen har kjørt, kamera, og eventuelt andre sensorer. Det er også av interesse å se på mulighetene for lagring av sensordataene og få vist disse i en app eller på en nettside. Dersom vi får tid til å vise dataene på en nettside, vil det også bli aktuelt å se på kryptering av dataene som blir sendt.

[2] Internet of Things

3 Organisering og utvikling

Utviklingen av selve bilen har fulgt en inkrementell utviklingsprosess, mens programvareutviklingen har vært inspirert av agil utviklingsmetodikk. Det som kjennetegner en inkrementell utviklingsprosess er at man utvikler produktet i flere «omganger» for å minske risikoen for at man tar for store steg i utviklingen, og man enklere kan identifisere og håndtere risikomomenter. Agil utviklingsmetodikk derimot, fokuserer blant annet på å alltid ha en fungerende programvare istedenfor å bruke omfattende dokumentasjon, og respondere på endringer istedenfor å slavisk følge en plan [3].

Det ble tidlig bestemt at bilen måtte bygges opp igjen fra bunnen av. De eneste komponentene som ble tatt med videre var lidaren, batteriet og metallchassiset som inneholdt motorene og en motordriver. I tillegg ble det funnet et ekstra chassis, som har vært uvurderlig i utviklingen av de forskjellige prototypene. Det ekstra chassiset sørget for at utviklingen av maskinvaren til den neste versjonen har kunnet pågått uten at utviklingen og testingen av programvaren ble påvirket.

3.1 Organisering i gruppen

Ingen av medlemmene har jobbet i isolasjon, men det har vært forskjellige fokus- og ansvarsområder for at arbeidet skal kunne gå så effektivt som mulig. Benjamin har hatt fokus på maskinvaredesign og -utvikling, inkludert koden på mikrokontrollerne, samt hatt ansvar for innkjøp og kontroll over det økonomiske. Bjørnar har hatt hovedansvaret for hovedprogram og simulator, mens Erlend har hatt hovedansvaret for program som tar for seg visning av direktevideo på nettside. Erlend har også tatt oppmålinger og tegnet kart for simulator, samt bidratt innenfor de andres fokusområder.

3.1.1 Arbeidssted og verktøy for samarbeid

Tirsdager og torsdager har vært faste arbeidsdager på skolen. Gruppen har da brukt PLS-laben på HVL som arbeidssted for enkel tilgang til bilen, verktøy og diverse deler som er brukt i prosjektet. De andre dagene har hvert gruppemedlem stått fritt til å jobbe hjemme eller på skolen, alt etter hva man trenger for å gjennomføre arbeidsoppgavene best mulig. For å sikre god kommunikasjon og effektivt samarbeid har vi benyttet:

- *Git*, et versjonskontrollsystem som primært brukes for kode, se Appendiks B .
- *Slack*¸ en samarbeidskanal for team, til kommunikasjon og deling av kortlevd informasjon.
- En delt OneNote-notatbok for notater og info som skal lagres over lengre tid.
- En delt OneDrive-mappe for deling av filer, som ikke er kode.
- Microsofts innebygde samarbeidsfunksjon for Word- og PowerPoint-dokumenter.

3.1.2 Fremdriftsplan

Vedlagt ligger en fremdriftsplan som har vært til stor hjelp innledningsvis i utviklingen av bilen, da prosjektet består av programvare som er avhengig av maskinvare og motsatt. Planen er laget i *TeamGantt*, som er et nettbasert program som gjør det enkelt å opprette en fremdriftsplan og legge inn aktiviteter som avhenger av hverandre. Etter hvert som arbeidsrutiner og ansvarsområder kom på plass, ble fremdriftsplanen mindre brukt.

3.1.3 Risiko

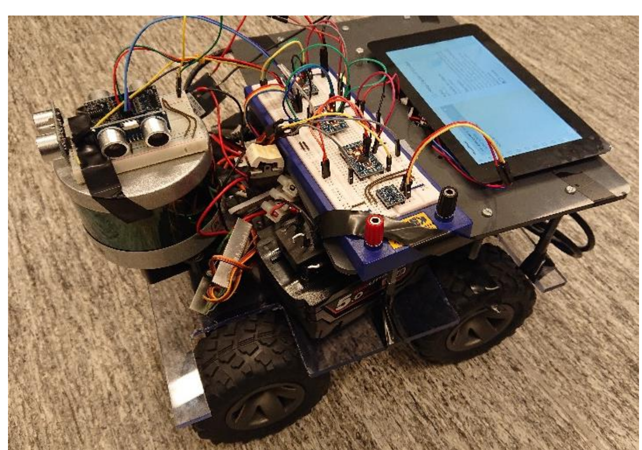
Til sensorbilen har det blitt utviklet mye kode til de forskjellige programmene. En stor risiko med kodeskriving er først og fremst at koden skal gå tapt. Neste risiko som også kan ta mye tid fra utviklingen er hvis det blir kodet feil og dette må rettes opp i. Som nevnt i 3.1.1 har vi benyttet *Git* som versjonskontrollsystem. *Git* har kontroll på alle endringer, koden lagres på en server i tillegg til at den vil ligge lokalt på PC-ene som har tilgang til koden.

En risiko for utviklingen av maskinvare er om det ble mangel på deler eller deler blir bestilt for sent. For å begrense denne risikoen ble det i starten av utviklingen tatt i bruk private komponenter og verktøy. Det ble tidlig i prosjektet bestilt inn komponenter og deler man så et behov for. Vi har valgt å bruke komponenter som er tilgjengelige fra flere butikker og leverandører. Dette har bidratt til å redusere ventetiden, men gjør det også lettere å skaffe erstatningskomponenter i fremtiden dersom noe skulle gå i stykker.

Gruppekonflikt ble sett som en liten risiko da alle i gruppen kjenner hverandre, har tidligere jobbet sammen i prosjekter, og samarbeider godt.

3.2 Prototype 1 (v.0.1)

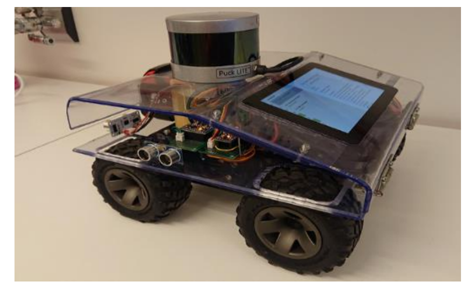
Den første prototypen ble i hovedsak laget for å teste hvordan de ulike maskinvarekomponentene fungerte, teste enkeltmoduler av kode og få til kommunikasjon mellom prosesseringsenhetene og sensorene. Som man kan se på Figur 3, var v.0.1 satt sammen for å «få noe som fungerer», og det ble ikke lagt særlig vekt på design. I dette stadiet ble det ble valgt å ha to ultralydsensorer som ser fremover for å dekke hjørnene fremme på bilen bedre, noe som ble implementert på den neste prototypen av bilen.

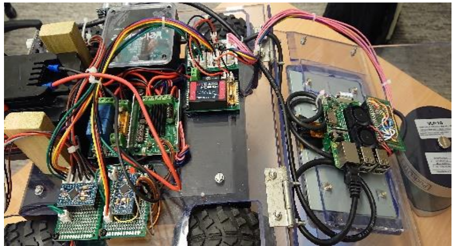


Figur 3: Sensorbil v.0.1

3.3 Prototype 2 (v.0.2)

Prototype nummer to ble satt sammen med betydelig større fokus på design, og den var fullt funksjonell. For å gjøre koblingene mellom de forskjellige maskinvarekomponentene mer solid, ble det laget kretskort, og det ble benyttet kontakter på ledningene for å gjøre det enklere å skifte ut enkeltkomponenter dersom de ble ødelagte eller måtte endres på i fremtiden. Det ble laget to flater av plastglass. Ett som ble plassert på chassiset, og ett som fungerte som et lokk over sensorene og prosesseringsenhetene. Hovedformålet med platen som ble plassert på chassiset var å øke området for plassering av maskinvarekomponentene. Platen som ble brukt som et lokk fungerte delvis som beskyttelse for sensorene og prosesseringsenhetene, men sørget også for at lidaren ser fritt rundt seg, og at skjermen ble enklere å betjene.



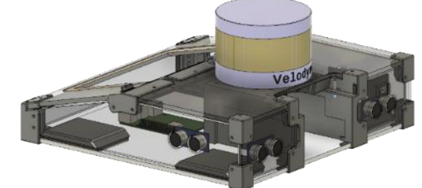


Figur 4: Øverst: Sensorbil v.0.2. Nederst: Forbedret koblinger mellom komponenter

Den komplette programvaren ble utviklet ved å bruke v.0.2, og utviklingsfasen til v.0.2 varte betydelig lenger enn både v.0.1 og v.1.0.

3.4 Sensorbil v.1.0

Den endelige versjonen ble bygget parallelt med at v.0.2 ble brukt for å støtte utviklingen av den komplette programvaren. For å gjøre platene mer estetiske ble det benyttet laserkutter for å skjære ut selve platene og gravere inn diverse mønstre. Istedenfor å bruke plastglass ble det benyttet plater av plexiglass. Alt av plater, hengsler, braketter og andre fester er tegnet med Autodesk Fusion 360 deretter 3D-printet. En av fordelene med å 3D-printe de ulike delene er at det gjorde det lettere å tilpasse dem etter ønsket design.



Figur 5: Øverst: Sensorbil v.1.0 Nederst: Sensorbil v.1.0 topp.

Det er mulig å vippe opp platen som skjermen ligger i. Dette gjør at man kan lett få tilgang til og se sammenhengen mellom kretskortene, datamaskinen, skjermen og sensorene. De kortene man trenger tilgang til er plassert under skjermen og er de som er mest interessante for studentene.

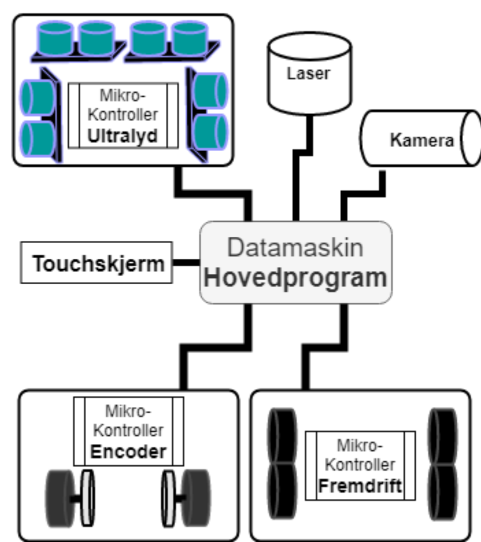
Dersom en foreleser skal introdusere bilen for en stor klasse, kan det være vanskelig at hele klassen ser bilen under presentasjonen. Derfor er det montert et webkamera på v.1.0 som kan brukes til videostrømming til nettside. En annen mulighet kameraet gir, er datasyn. Dersom dette implementeres i hovedprogrammet på bilen, og data fra kameraet kan brukes i studentenes kontrollogikk, åpner dette for nye muligheter. Et eksempel på dette er at kameraet kan detektere et spesielt mønster og så få bilen til å ta passende beslutninger basert på dette.

4 Systemets oppbygning og bruksområde

Bilens system er bygget opp med RPi som hoveddatamaskin, det er her hovedprogrammet med studentenes kontrollogikk blir lastet opp til og kjørt.

RPi er en ettkortsdatamaskin, forkortelsen SBC[3] vil bli brukt videre i rapporten.

Flere mikrokontrollere, som er «enkle kompakte datamaskiner», blir brukt til å behandle data fra sensorer og til å styre fremdriften. Mikrokontrollerne kommuniserer med SBC-en på en felles kommunikasjonslinje, en databuss.



Figur 6: Enkel systemskisse

Bilen har fire ultralydsensorer som brukes til avstandsmåling, der én ser til venstre, én ser til høyre, og to ser fremover. En mikrokontroller kjører logikken som trengs for å måle avstand på hver av de fire ultralydsensorene.

Bilen har to encodere, én på venstre bakhjul og én på høyre bakhjul. Encoderne måler hvor mange runder hjulet har gått, og man kan regne seg frem til avstanden bilen har kjørt og hvilken hastighet bilen holder. Mikrokontrolleren som styrer fremdriften på bilen får kommandoer fra SBC-en som inneholder hvilken kraft som skal settes på motorene. Kombinert med et signal direkte fra mikrokontrolleren til ultralyden, beregner den hvilket signal som skal sendes videre til motordriveren.

Lidaren og webkameraet er det eneste som er tilkoblet direkte til SBC-en. Lidaren er koblet til nettverksporten, og har en ferdig protokoll på hvordan data overføres. Webkameraet er et vanlig USB-kamera og er koblet til ene USB-porten.

Flere detaljer om dette finnes i Appendiks D .

4.1 Hovedprogrammet

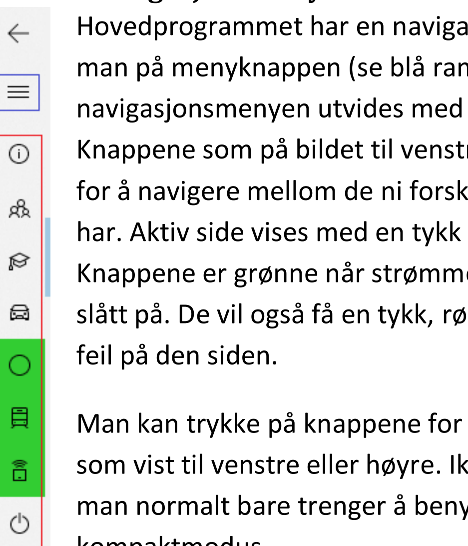
Dette kapittelet viser hvordan hovedprogrammet ser ut og hva det kan brukes til. For lesere som er interessert i å vite mer om hva som ligger bak, anbefales det å lese Appendiks E i tillegg til dette kapittelet. Dersom man ønsker å lage egen kontrollogikk, eller bare teste ut programmene på sin egen PC, vil brukerguiden i Appendiks C være nyttig for å komme i gang.

Brukergrensesnittet til hovedprogrammet lar brukeren blant annet observere sensordata og starte/stoppe kontrollogikkene som er lagt inn. Hovedprogrammet kan enten lastes over på selve bilen hvor bruker kan interagere med programmet via bilens trykkskjerm, eller det kan kjøres lokalt på brukers egen PC (mot simulator). Det kreves bare noen få, enkle tastetrykk for å bytte mellom

[3] Single-board computer

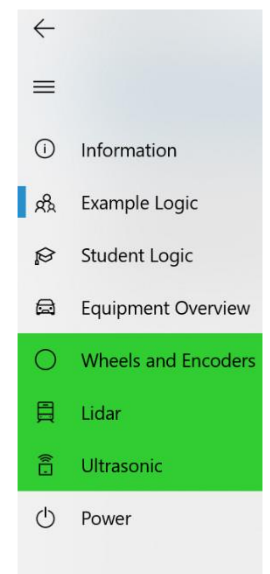
hvor koden skal lastes opp. Programmet vil automatisk tilpasse seg til hvor den kjører, slik at studentene ikke trenger å forholde seg til dette.

4.1.1 Navigasjonsmeny



Hovedprogrammet har en navigasjonsmeny på venstre side. Trykker man på menyknappen (se blå ramme på bildet til venstre), vil navigasjonsmenyen utvides med beskrivelser som vist på bildet til høyre. Knappene som på bildet til venstre er markert med rød ramme brukes for å navigere mellom de ni forskjellige sidene som hovedprogrammet har. Aktiv side vises med en tykk blå strek (som her er «Example Logic»). Knappene er grønne når strømmen til utstyret som tilhører den siden er slått på. De vil også få en tykk, rød ramme dersom det har oppstått en feil på den siden.

Man kan trykke på knappene for å navigere enten man har den i modus som vist til venstre eller høyre. Ikonene er relativt selvforklarende, slik at man normalt bare trenger å benytte navigasjonsmenyen i kompaktmodus.

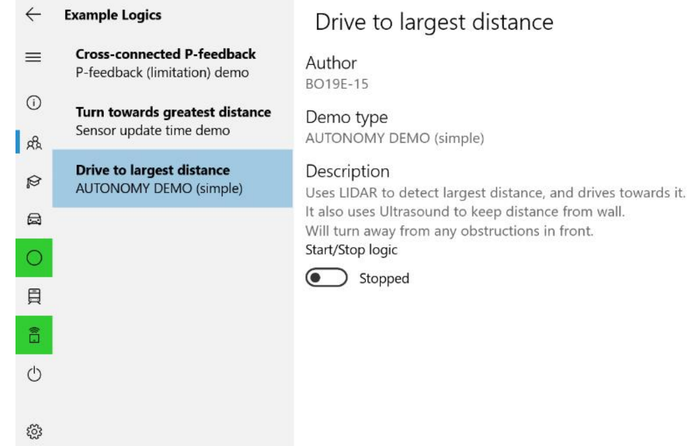


4.1.2 Kontrollogikk

Kontrollogikkene er koden som utfører valg som får bilen til å kjøre. Vi har utviklet et bibliotek av funksjoner for sensorer og hjul som gjør det enkelt å hente ut måledata fra sensorene og gi kommandoer til hjulene. I hovedprogrammet finnes det to sider for kontrollogikker: «Example Logic» og «Student Logic». Til venstre på sidene vises en liste over kontrollogikkene, samt en kort beskrivelse på hver av disse. Til høyre vises en mer detaljert beskrivelse av den valgte kontrollogikken. Det er her man kan starte/stoppe koden man har skrevet.

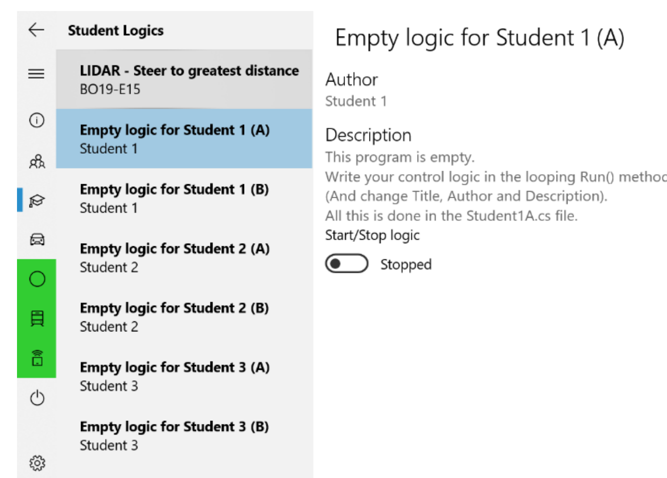
4.1.2.1 Eksempellogikk

På siden for eksempellogikk finner man komplette kontrollogikker. «Drive to largest distance» er ment for å demonstrere hva bilen kan brukes til, og er en enkel kontrollogikk som gjør at bilen kan kjøre rundt i korridorene av seg selv.
Andre kontrollogikker er ment for å illustrere et bestemt konsept.
For eksempel viser «Turn towards greatest distance» hvordan oppdateringshastigheten til måledataene påvirker evnen til å kontrollere bilen.



4.1.2.2 Studentlogikk

På siden for studentlogikker finner man mange blanke maler, slik at studentene enkelt kan komme i gang med å skrive sin egen kontrollogikk. I C.2 er det beskrevet hvordan dette gjøres.



Det er også enkelt å opprette flere slike filer for kontrollogikk dersom det skulle bli nødvendig. Hovedprogrammet er designet slik at de vil automatisk dukke opp i listen, uten å måtte gjøre noe med GUI-koden[4].

Beskrivelsen som vises på skjermbildet kommer fra en tekststreng man skriver inne i kodefilen for kontrollogikken. Denne kan derfor lett lages/endres når man jobber med kontrollogikken.

Det finnes også kontrollogikker her som er ment som inspirasjon/startpunkt for studentenes egne kontrollogikker.

4.1.2.2.1 Kontrollogikken «LIDAR – Steer to greatest distance»

«LIDAR – Steer to greatest distance» er en halvfungerende kontrollogikk. Denne er ment som inspirasjon, men er også et hjelpemiddel for å forstå hvordan man kan dra nytte av en sensor, samt få innsikt i dens begrensninger.

```
public override void Run(CancellationToken cancellationToken)
{
    float angleToLargestDistance = _lidar.LargestDistanceInRange(260, 100).Angle;

    SteerTowardsAngle(angleToLargestDistance, baseSpeed: 100);

    Thread.Sleep(millisecondsTimeout: 50);
}

1 reference | BjAlvestad, 9 days ago | 1 author, 1 change
private void SteerTowardsAngle(float angleDeviation, int baseSpeed)
{
    int leftSpeedReduction = angleDeviation > 180 ? 360 - (int)angleDeviation : 0;
    int rightSpeedReduction = angleDeviation < 180 ? (int)angleDeviation : 0;

    _wheels.SetSpeed(leftValue: baseSpeed - leftSpeedReduction, rightValue: baseSpeed - rightSpeedReduction);
}
```

Figur 7: Simpel kontroll logikk som bruker for å kjøre mot største avstand

De få kodelinjene man ser på Figur 7 er tilstrekkelig til at bilen kan kjøre rundt i korridorene. Bilen vil dog ikke kompensere for avstand til veggen på sidene, og kan derfor få problemer dersom den kommer for nær. Lidaren ser også gjennom glass, noe som kan være problematisk dersom den største avstanden den ser er gjennom et vindu.

Denne kontrollogikken er derfor et fint utgangspunkt for en oppgave hvor man legger til funksjonalitet for å håndtere disse problemstillingene.

[4] Graphical User Interface

4.1.3 Observasjon av sensordata

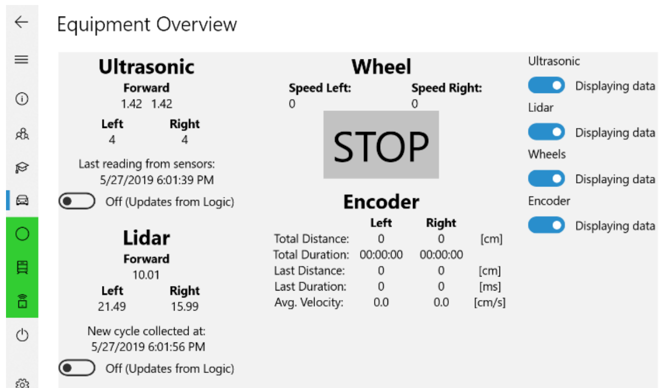
I hovedprogrammet finnes det flere sider for de forskjellige sensorene. Disse sidene kan være nyttige for å bli kjent med hvordan sensorene fungerer. Ved å observere dataene som kommer inn på sensorsidene mens man manipulerer bilen manuelt, kan man bli kjent med sensorenes styrker og svakheter. Dette kan nok for mange være en mer effektiv måte å bli kjent med sensorenes virkemåte på, enn å bare lese teorien.

Spesielt for lidaren er det også en del innstilinger man kan endre mens hovedprogrammet kjører. Slik kan man enkelt se hvordan de forskjellige innstilingene påvirker sensoren. Man kan også endre på disse mens man kjører en kontrollogikk som benytter seg av lidaren for å se hvordan dette påvirker kontrollogikken. Alle innstillingene man kan endre kan også settes fra koden, og man trenger ikke benytte skjermen. Muligheten for å også kunne endre disse fra skjermen kan være veldig nyttig i tidlig testfase av kontrollogikken.

Oversiktssiden «Equipment Overview» kan være nyttig når man prøver å feilsøke kontrollogikken sin, for å forstå hvorfor den tar de valgene den tar.

I kapitelene under følger en kort innføring for de forskjellige sidene med sensordata/-innstillinger.

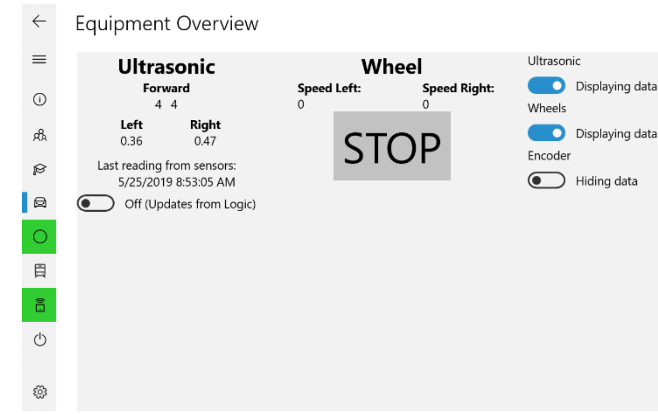
4.1.3.1 Oversiktsside over alt utstyr



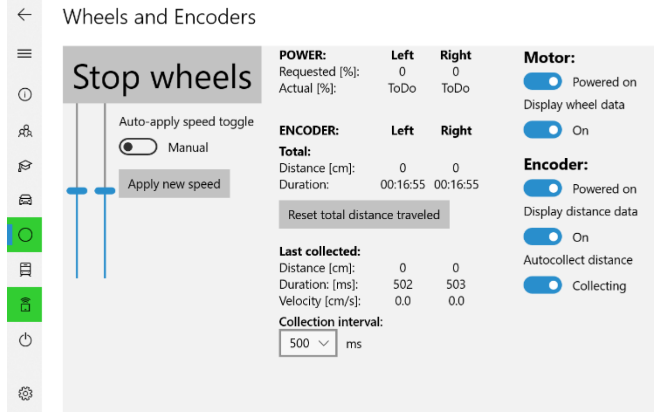
For å kunne se data fra flere sensorer samtidig er det laget en oversiktsside som viser data fra alle sensorene. Grunnet plassbegrensninger er ikke all dataen for lidaren tatt med på denne siden.

Bryterne på høyre side brukes til å bestemme hvilke sensorer som skal vises. Selve bryterne er bare synlig dersom strømmen for det tilhørende utstyret er slått på.

På bildet til høyre ser vi at data fra encoderne ikke vises ettersom encoder-bryteren ikke er aktiv. Strømmen til lidaren er ikke slått på, så bryteren for å vise lidardata er ikke synlig.

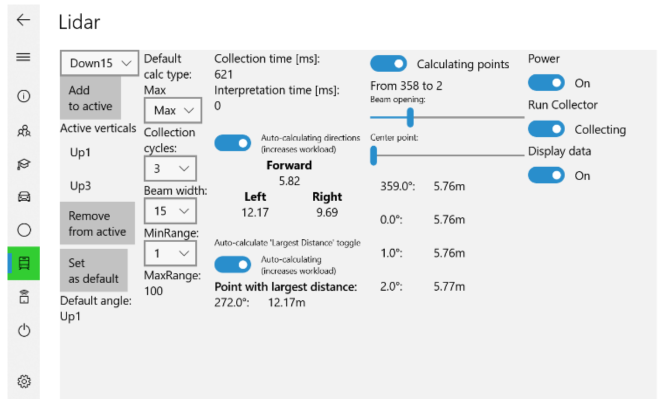


4.1.3.2 Hjulkommando og encoder



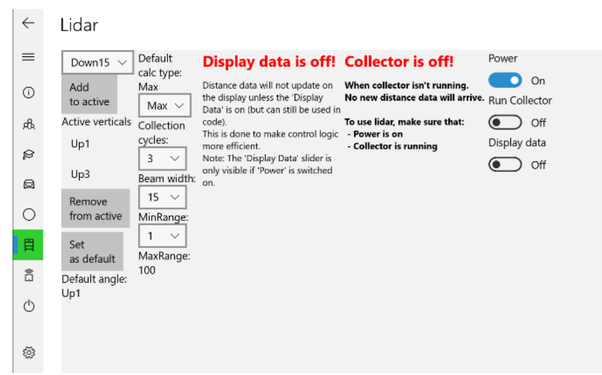
På skjermsiden for hjul og encoder kan man blant annet styre hjulhastigheten manuelt med glidebryterne til venstre. I midten av siden kan man observere data fra hjul og encoder, samt manuelt resette distansen som er målt. Dersom den nederste blå knappen er aktiv, vil distansen bilen har kjørt bli beregnet med et fast intervall. Dersom den derimot ikke er aktiv, dukker det opp en knapp man må trykke på for å beregne distansen.

4.1.3.3 Lidar



Skjermsiden for lidaren er den med flest innstillinger. Dette er en konsekvens av at lidaren er den mest komplekse sensoren som benyttes. Den krever derfor en del mer databehandling, og kan samle inn mer data som er av interesse å vise.

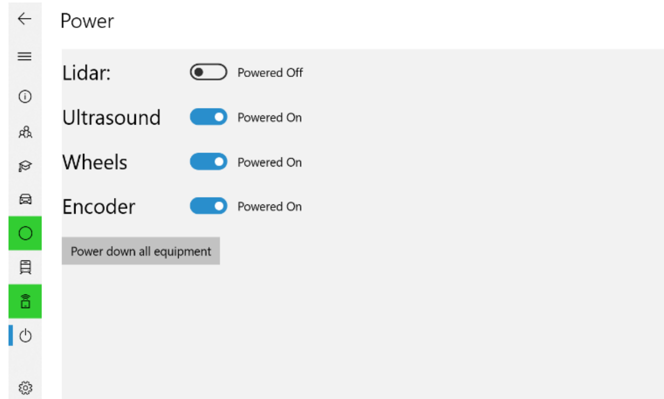
For å bruke lidaren må både «Power» og «Run Collector» slås på. Dersom «collectoren» ikke kjører vil det vises tydelig informasjon om dette på lidarsiden.



4.1.3.4 Ultralyd

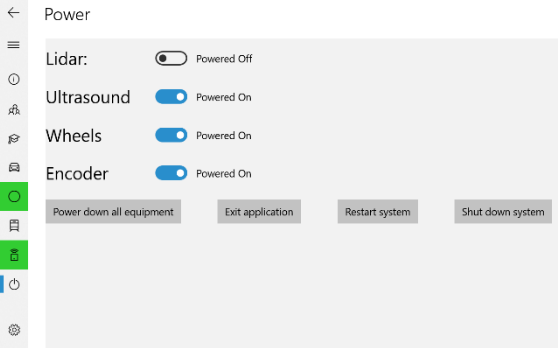
På skjermsiden for ultralyd kan man observere avstandsmålingene fra ultralydsensorene. Mikrokontrolleren for ultralydsensorene tar seg av logikken for å måle avstand og beregningene som trengs. Hovedprogrammet bare spør etter disse avstandsmålingene fra mikrokontrolleren, og det er derfor relativt lite innhold på skjermsiden for ultralydsensorene.

4.1.4 Strømstyring

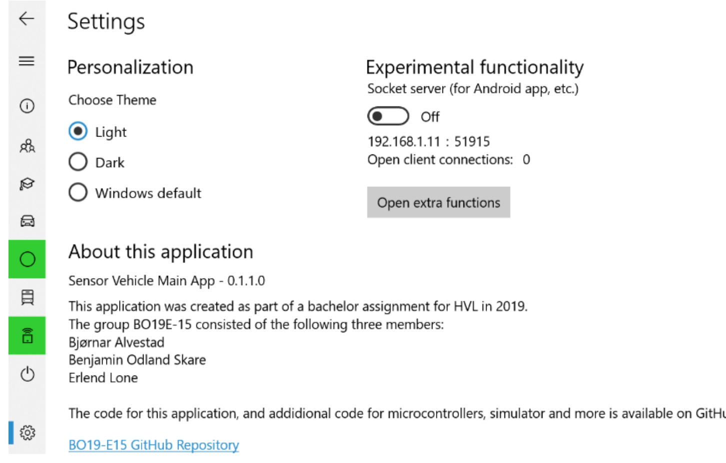


På skjermsiden for strømstyring har man en sentralisert plass for å slå av/på strøm til de forskjellige komponentene. Dette er de samme power-knappene som man finner under de individuelle komponentene. Man har også en knapp «Power down all equipment» som slår av strømmen til alt utstyret.

Skjermbildet ovenfor viser hvordan skjermsiden ser ut når programmet kjører på en PC, mens bildet til høyre viser hvordan det ser ut når programmet kjører på bilen.

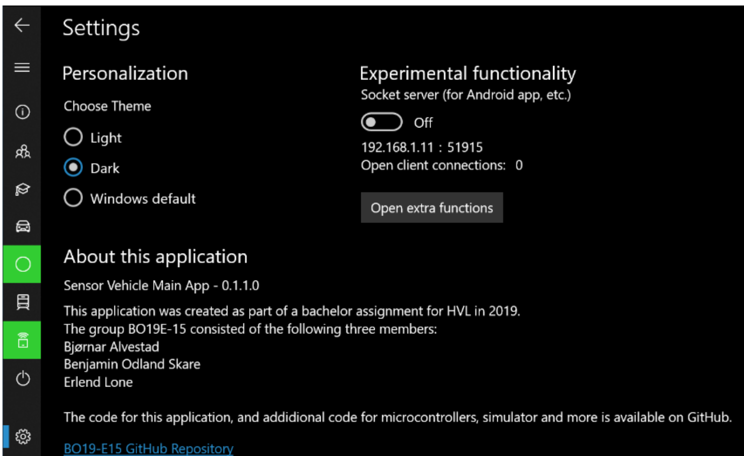


4.1.5 Innstillinger



På skjermsiden for innstillinger finner man informasjon om applikasjonen og link til all kode for bilen. Man kan også slå på mulighet for fjernstyring av bil, samt åpne «SensorVehicle Extras» for videostrømming til nettside.

Ved førstegangs kjøring av hovedprogrammet på PC vil fargemodusen være det samme som man har satt som standard på PC-en. Som vist under «Personalization» på bildet til høyre er det mulig å velge enten lys eller mørk appmodus. Programmet husker hvilken innstilling som er valgt, slik at man slipper å endre det hver gang programmet starter.



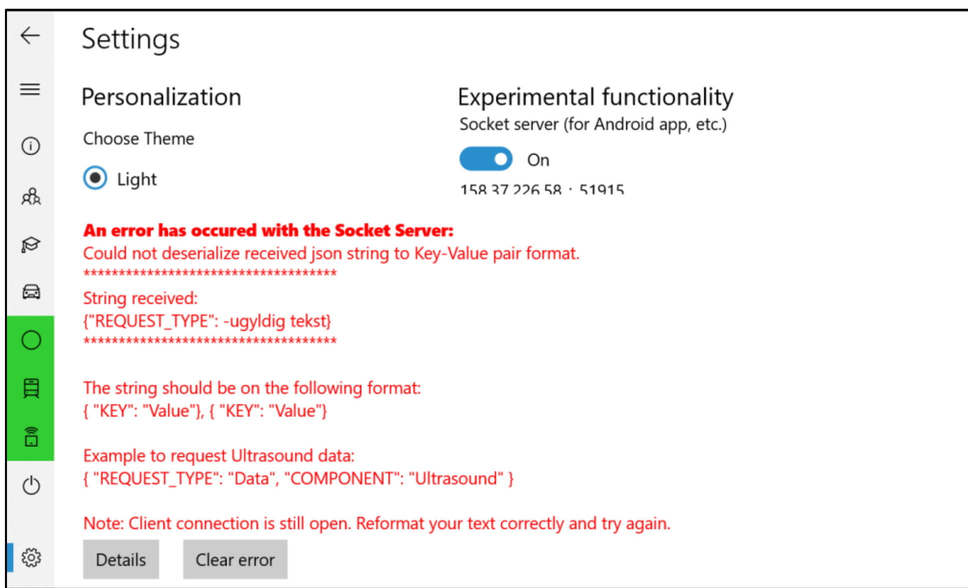
På bilen er standardmodus satt til mørk. Skjermbildene som er tatt for denne rapporten har vært med lyst tema ettersom det er lettere å lese i et tekstdokument.

4.1.6 Unntakshåndtering

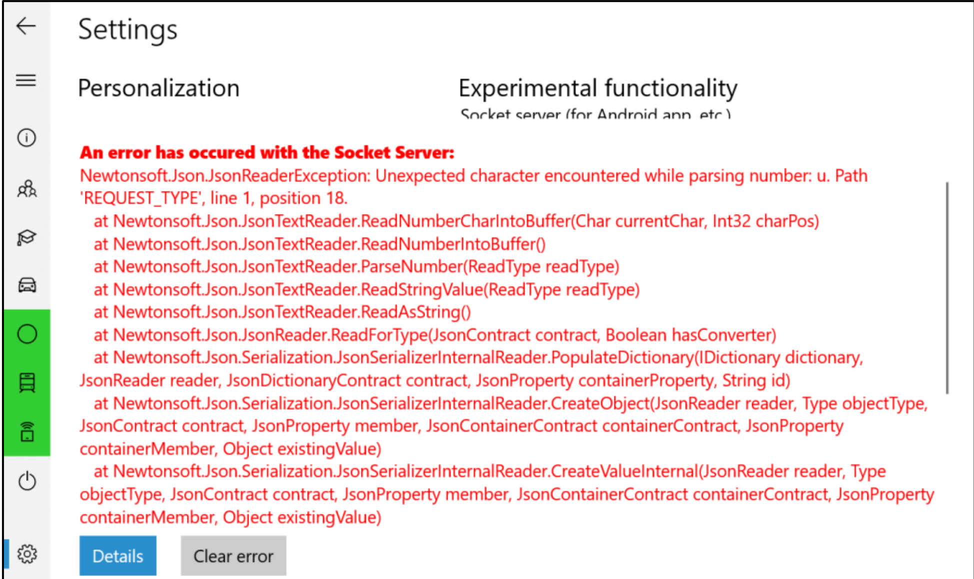
Skjermsidene for kontrollogikk, sensorer og innstillinger vil vise feilmeldinger knyttet til seg. Brukeren kan signere ut denne feilmeldingen, eller trykke på Details-knappen for å flere detaljer rundt denne.

4.1.6.1 Eksempel på feilmelding for «Socket Server»

På Figur 8 er det fremprovosert en feilmelding på socketserveren ved å sende den en melding på ugyldig format. Ved å trykke på Details-knappen får man opp det som vises i Figur 9. Dersom man trykker på Details-knappen en gang til vil opprinnelig feilmelding vises. Dersom feilmeldingen er for lang, kan man scrolle opp eller ned i teksten ved hjelp av datamus eller touchskjerm.



Figur 8: Feilmelding ved sending av melding på ugyldig format



Figur 9: Detaljert feilmelding

4.1.6.2 *Kontrollogikkens oppførsel ved sensorfeil*

Dersom en feil oppstår i selve kontrollogikken, vil hovedprogrammet automatisk stoppe bilen og vise en feilmelding.

Ved en feil på en sensor, vil det vises en feilmelding på sensorens side og det aktuelle sensorikonet i venstre meny i hovedprogrammet blir markert med en tykk, rød ramme. Bilen vil ikke automatisk stoppe ved sensorfeil. Den som lager kontrollogikken må selv sjekke om sensoren har en feil, og så håndtere dette som en feil i kontrollogikken, dersom det er ønskelig. At bilen ikke stopper av seg selv er et bevisst valg som ble tatt under design hovedprogrammet. Den som skal lage kontrollogikk ønsker kanskje å håndtere dette på en annen måte, som for eksempel å bytte hvilken sensor man benytter dersom en feil skulle oppstå på én av dem.

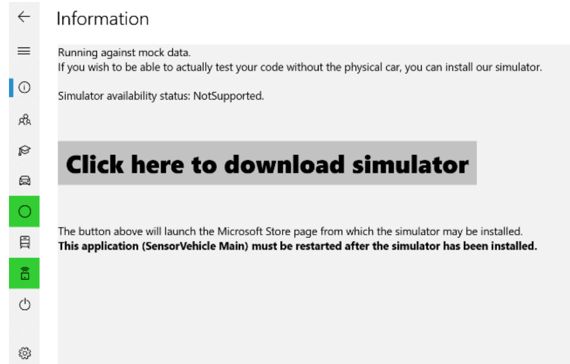
Merknad: Bilen har et kollisjonssikringssystem som kjører internt mellom mikrokontrollerne for ultralyd og hjulstyring som fortsatt vil være funksjonelt uavhengig av hva som skjer med hovedprogrammet.

4.2 Simulatoren

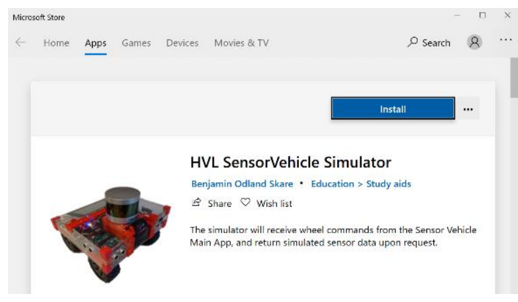
Dette kapittelet tar for seg hvordan simulatoren ser ut og hvordan den kan brukes. Detaljer om hvordan koden for simulatoren er bygget opp, hvilke verktøy som er benyttet og hvordan man lager egne kart er beskrevet i Appendiks F

Simulatoren er et verktøy som muliggjør testing av kontrollogikk selv om man ikke har bilen tilstede. Vi har utført målinger på den fysiske bilen og ved hjelp av regneverktøy funnet matematiske formler som beskriver bilens bevegelse. Disse er videre brukt for å sørge for at den simulerte bilen oppfører seg svært likt den fysiske bilen.

HVL SensorVehicle Simulator er lagt ut på *Microsoft Store* slik at den skal være enkel å installere. Når man velger å kjøre hovedprogrammet på sin egen PC, vil simulatoren automatisk starte opp dersom den er installert. Hvis hovedprogrammet derimot oppdager at simulatoren ikke er installert, vil dette vises tydelig på hovedprogrammets startside (se Figur 10).



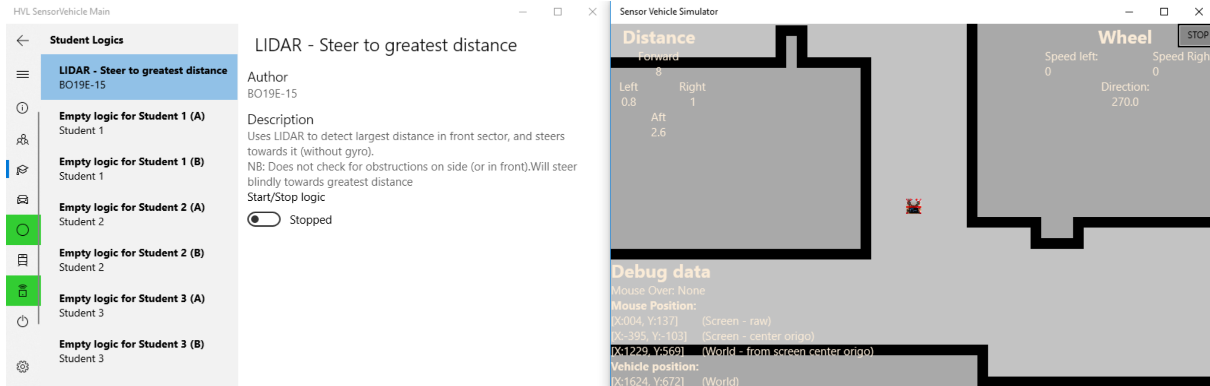
Figur 10: Hovedprogrammets startside på PC uten simulator



Figur 11: Simulatoren er lagt ut på Microsoft Store

4.2.1 Bruk av simulatoren

Hovedprogrammet vil automatisk starte opp simulatoren når det kjører på en PC, og koden vil tilpasse seg til at den skal hente data fra og sende kommandoer til simulatoren, og ikke den fysiske bilen.



Studentene kan videre bruke hovedprogrammet som om det kjørte på den fysiske bilen. Bilen i simulatoren vil følge hjulkommandoene den blir gitt av kontrollogikken, og gir simulerte måleverdier for lidar og ultralyd når dette etterspørres. Encoderne returnerer også verdier, men dette er bare hastigheten som er satt på hjulene og ikke avstand, noe den fysiske bilen gjør. Utvikling av dette ble prioritert vekk grunnet tidsnød.

Det er mulig å flytte eller rotere bilen manuelt i simulatoren med henholdsvis venstre og høyre musetast. Dette kan være nyttig for å sette bilen i den posisjonen man ønsker å teste kontrollogikken sin for. Man kan også flytte/rotere bilen slik mens kontrollogikken kjører for å sjekke hvordan den håndterer eksterne forstyrrelser.

Man kan også zoome inn og ut på kartet med «PageUp» og «PageDown» på tastaturet. Trykker man «Home»-knappen resettes zoomen til opprinnelig nivå.

Skjermbildet på simulatoren er alltid sentrert på bilen, med unntak av når man flytter bilen manuelt med venstre musetast. Da låses kamera for å unngå at bildet hopper mye rundt, men kartet flytter på seg dersom man drar bilen mot kanten av skjermen.

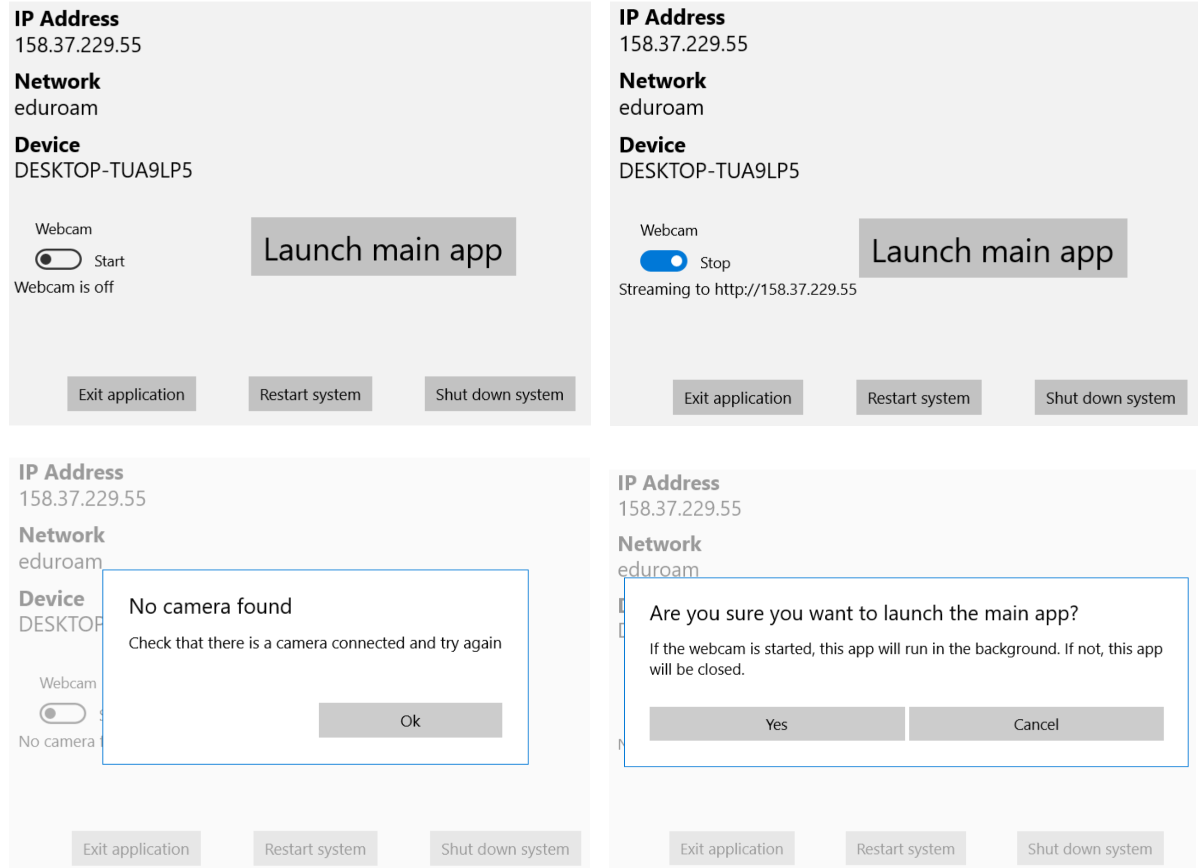
4.3 Andre programmer

4.3.1 Program med ekstrafunksjonalitet - videostrømming på nettside

Webkameraet som er montert på bilen brukes til å strømme direktevideo til en nettside. Appen som muliggjør strømmingen er kalt «Sensor Vehicle Extras», og baserer seg på HttpWebcamLiveStream [4]. Dette er en app som er laget av SaschaIoT og lisensiert under MIT Licence. MIT Licence sier at enhver person som får tilgang til programvaren og dens dokumentasjonsfiler fritt kan bruke, kopiere og endre programvaren, så lenge man legger ved opphavsrettserklæringen [5].

Når brukeren forsøker å starte strømmingen ser appen om et webkamera er tilkoblet. Dersom den finner et webkamera, starter strømmingen, og man får tilgang til denne ved å koble seg til bilens IP-

adresse via en nettleser[5]. Etter hvert som bilen kjører gjennom korridorene vil den endre aksesspunkt flere ganger. For å forhindre at strømmingen stopper når dette skjer, oppdateres nettsiden hvert 10. sekund.



Appen gir brukeren mulighet å åpne hovedprogrammet. For å opprettholde strømmingen etter at hovedprogrammet er startet, vil appen fortsette å kjøre i bakgrunnen. Dersom strømmingen derimot ikke er startet, vil appen lukkes, og det vil bare være hovedprogrammet som kjøres.

4.3.2 Androidprogram for fjernstyring

Vi laget en Android-app for fjernstyring av bilen som semesteroppgave i emnet «ELE122 – Nettverksprogrammering» samtidig som prosjektet. Denne appen kommuniserer med socketserveren som ble laget på bilen i forbindelse med bacheloroppgaven. Koden for denne appen ligger også i repositoriet «BO19E-15_SensorVehicle» på GitHub.com.

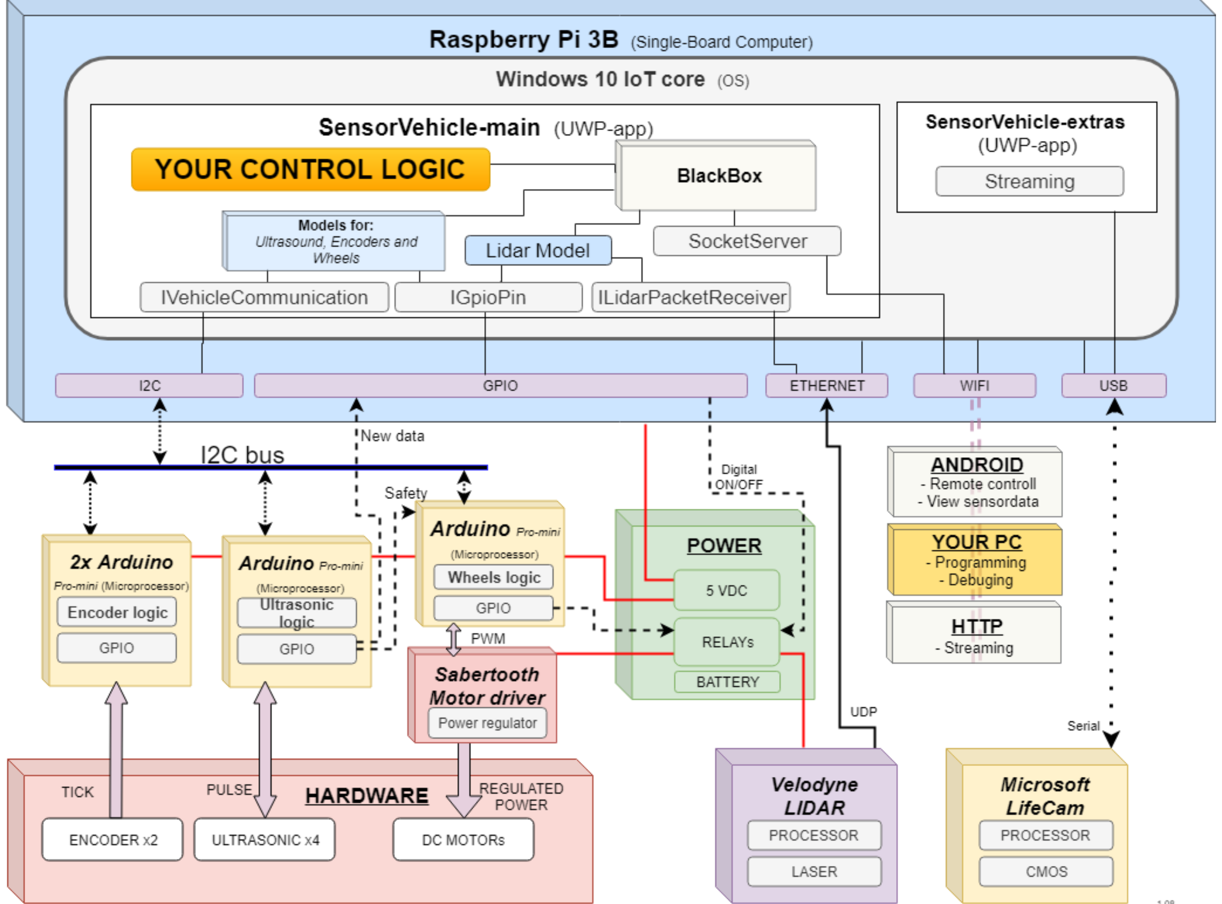
4.3.3 Mikrokontrollerkode

Programmene som kjører på mikrokontrollerne er å regne som fastvare, og det er ikke tiltenkt at studentene skal endre på disse. Det ligger litt informasjon om mikrokontrollerkoden i D.2 , og fullstendig kode er tilgjengelig i repositoriet «BO19E-15_SensorVehicle» på GitHub.com.

[5] Har tidvis vært problemer med tilkobling via Mozilla Firefox

4.4 Systemoversikt

Her kommer en rask oversikt av komponentene og sammenhengen mellom disse. For flere detaljer, se Appendiks D



Figur 12: Systemskisse

Raspberry Pi 3B (RPi) er en fullverdig datamaskin laget på ett kort (Single-Board Computer).

Windows 10 IoT Core er et operativsystem som optimalisert for mindre enheter med ARM- eller x64/x86-prosessor [6].

SensorVehicle-main er hovedprogrammet til bilen og er en Universal Windows Platform (UWP) applikasjon som kjører i operativsystemet. Appen er skrevet i programmeringsspråket C#.

SensorVehicle-extras gir brukeren mulighet for strømming av video, og er en UWP-app som bruker C#, JavaScript og HTML.

BlackBox inneholder alle klassene som er brukt i hovedprogrammet. Se E.2 for mer detaljert tegning.

I2C buss er en kommunikasjonsprotokoll som går over to ledere og holder kommunikasjonen mellom RPi og Arduinoene.

GPIO «General-Purpose Input/Output» er digitale inn- og utganger brukeren kan kontrollere i koden.

WiFi Bilen er tilkoblet HVLGuest og PC-en som skal laste opp program til bilen må være på Eduroam.

USB er benyttet for å koble til webkamera og touchinput på skjerm

Arduino Pro Mini er en mikrokontroller som programmeres med C++ og har en rekke digitale og analoge inn- og utganger.

Power består av sikringer, 5 volts spenningsregulator, batteri, rele til å styre strøm til lidar og motor.

Android er en app som ble laget i ELE122 som en semesteroppgave og benytter seg av socket-porten på bilen og kan styre og lese data.

Your PC er din pc hvor du kan programmere eller styre bilen.

http Man kan koble seg til IP-adressen til bilen via en nettleser og få opp en videooverføring fra kamera på bilen**.**

Sabertooth er en motordriver som regulerer strømmen fra batteriet til bilens motorer.

PWM står for pulsbreddemodulasjon, og er et signal med fast frekvens av firkantpulser hvor bredden på firkantsignalene er det som justeres.

Velodyne Lidar sender ut laser og måler de reflekterte pulsene. Lidarer blir ofte brukt til å kartlegge og lage 3D-representasjoner av det som er rundt.

Microsoft LifeCam er kameraet som er koblet til bilen og brukes til videostrømming.

5 Testing

5.1 Testing av kode

Testverktøyet xUnit.net til å skrive unit-tester for noen av metodene i hovedprogrammet. xUnit.net er gratis og har åpen kildekode [7]. Koden for unit-testene er tilgjengelige i GitHub repositoriet «BO19E-15_SensorVehicle».

5.2 Test av sensorer

Alle avstandsmålingene i Tabell 1 ble sjekket opp mot lasermåleren Bosch PLR15 som har et måleområde på 0.15-15.00 m, og nøyaktighet på +/- 3,0 mm [8]. Målingene ble ikke blir utført under laboratorietilstander, og usikkerheten knyttet til målingene vil derfor være større en det oppløsningen på lasermåleren skulle tilsi. Formålet med disse målingene var å sjekke at både sensorene og koden som tolker dataen fra dem fungerer korrekt. Det er ikke vårt mål å strengt følge statistiske fremgangsmetoder for målingene.

5.2.1 Lidar

Målingene fra lidaren ble sjekket ved at bilen ble flyttet nærmere og nærmere en flat, hvit vegg, mens måleverdiene som ble presentert i hovedprogrammet kontinuerlig ble sjekket mot målingene fra PLR15. Lidaren klarte å måle med maksimalt 1 cm avvik helt ned til 0,5 meter, men fikk ikke inn målinger som var nærmere enn dette. Manualen lover bare presise målinger ned til 1.00 m.

Målingene er gjort fra sentrum av lidaren.

5.2.2 Ultralyd

For ultralydsensorene er det viktigste at det er lite avvik mellom de enkelte sensorene på samme avstand. For at det ikke skal skape problemer for styringen av bilen, bør avviket mellom sensorene være mindre enn 10 cm, men det er foretrukket å ha mindre enn 5 cm avvik.

Lasermåler	***0,250 m***	***0,500 m***	***1,000 m***	***2,000 m***	***3,000 m***	***4,000 m***
Left	0,24 m	0,48 m	0,97 m	1,95 m	2,96 m	3,94 m
Fwd Left	0,24 m	0,49 m	0,97 m	1,95 m	2,94 m	3,91 m
Fwd Right	0,24 m	0,49 m	0,97 m	1,95 m	2,94 m	3,91 m
Right	0,24 m	0,49 m	0.97 m	1,95 m	2,94 m	3,92 m

Tabell 1: Kontrollsjekk av avstandsmålingene fra ultralyd, tatt i en romtemperatur på 23.3°C

Fra målingene ovenfor ser vi at det største avviket er på 3 cm mellom sensorene, som er innenfor ønsket avvik.

Absoluttavstanden har et avvik som er omtrent +2 cm per meter, noe som resulterer i et absoluttavvik på litt i underkant av 10 cm når man måler på maksavstanden som er 4 meter. Dette er ikke noe stort problem ettersom disse sensorene vil primært benyttes for å sjekke hvilke som har størst avstand, ikke eksakt hva avstanden er. Det vil uansett være fordelaktig å modifisere formelen i mikrokontrollerkoden ved neste planlagte endring. Dette vil kreve en ny omgang med testing av sensorene og interaksjon inn mot hovedprogram.

5.2.3 Encoder

For å teste om encoderne måler korrekt ble bilen kjørt 6 meter (målt med PLR15) rett frem mot en vegg. Avstanden encoderne målte ble så sammenlignet med avstanden PLR15 gav.

Målingene ble utført på forskjellige hastigheter for å kontrollere at koden som tolker målingene fra encoderne var skrevet effektivt nok til at den ikke gikk glipp av noen av pulsene, selv ved høyere hastigheter (i.e. den måler kortere avstand ved høyere hastigheter).

Kraft	***30%***	***50%***	***100%***
Venstre encoder	595 cm	593 cm	594 cm
Høyre encoder	599 cm	599 cm	601 cm

Tabell 2: Kontrollsjekk av encodermålingene ved forskjellige hastigheter

Som man kan se fra Tabell 2 er det et avvik på 2 cm mellom målingene på samme encoder ved de forskjellige hastighetene. Dette viser at koden er skrevet effektivt nok til at den ikke mister noen pulser ved høyere hastighet.

Det var opptil 7 cm avvik mellom encoder på venstre og høyre side. Dette kommer av at bilen ikke kjørte langs en rett linje under testen. Måleusikkerheten knyttet til kontrollmålingene blir derfor relativt stor, og vi estimerer den til å være rundt 2 cm per meter.

5.3 Tester for simulatorfysikk

I første utgave av simulatoren benyttet vi formler for glidestyring. Disse fikk den simulerte bilen til å tilsynelatende oppføre seg fint, men sammenlignet med bevegelsene til den fysiske bilen var bevegelsene for raske. Dermed ville samme kontrollogikk på simulator og på fysisk bil ha forskjellig evne til å kontrollere bilen. For å få bilen i simulatoren til å oppføre seg likt som den virkelige bilen ble det utført flere målinger på den fysiske bilen (med fullt batteri). Tiden ble målt med stoppeklokke, og avstanden med samme lasermåler som ble brukt for sensortestene i 5.2.

Videre ble Excel og det nettbaserte verktøyet mycurvefit.com brukt til å finne formler ved hjelp av regresjon. Det ble benyttet to forskjellige regneverktøy ettersom Excel bare støttet noen regresjonstyper. Den som passet best til vår data som Excel kunne generere var «potensregresjon». Mycurvefit.com har derimot et mye større utvalg av regresjonstyper, og brukte «symmetrisk sigmoid regresjon» for å best mulig representere dataene våre. Denne har dog et mer komplisert formeluttrykk, og bruk av denne vil kreve flere regneoperasjoner av datamaskinen simulatoren kjører på. Begge formeltypene er tilgjengelige i koden for simulatoren, og man kan bytte mellom disse ved å endre én variabel (enum). Som standard benyttes sigmoidregresjonsformelen.

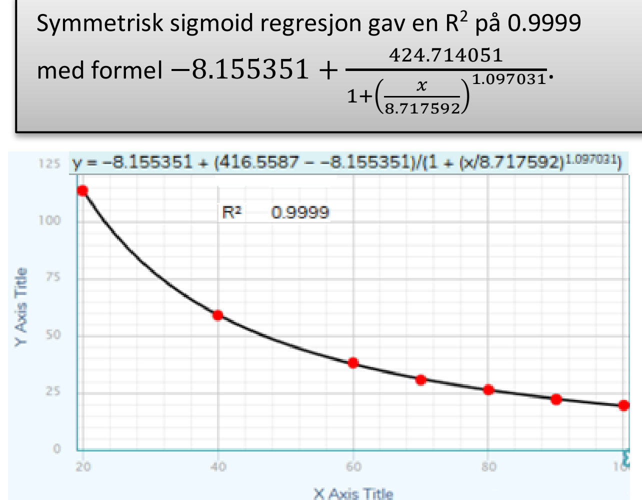
5.3.1 Rettlinje-test

Settpunkt for hjul venstre og høyre	$\frac{sekund}{10\ meter}$
20%	113,66 sek
40%	58,95 sek
60%	38,03 sek
70%	30,61 sek
80%	26,21 sek
90%	22,01 sek
100%	19,45 sek

Tabell 3: Målinger for rettlinjet bevegelse

Disse målingene ble gjort ved at det ble satt lik hjulkraft på venstre og høyre side, og målt tiden bilen brukte på å kjøre 10 meter (målt med PLR15). Etter at måledataene var samlet inn benyttet vi regresjon for å finne en formel som beskrev forholdet mellom kraftkommando til hjul, og tiden den brukte på 10 meter.

Potensregresjon gav oss en R^2 på 0.9942 samt en enkel formel $3235.5x^{-1.1}$ som beskriver forholdet.



Figur 13: Symmetrisk sigmoid regresjon for rettlinjet bevegelse

Endring i posisjon (i rett linje)

Tid per 10m

y = 3235,5x^-1,1
R² = 0,9942

Gjennomsnitlig kraftkommando på hjul

Figur 14: Potens regresjon for rettlinjet bevegelse

5.3.1.1 Test av formlene for rettlinjet bevegelse i simulator

Begge formlene ble testet i simulatoren.

Settpunkt for hjul venstre og høyre	Fysisk bil	Simulator m/ Potensregresjon	Simulator m/ Sigmoidregresjon
20%	113,66 sek	124,20 sek	114,56 sek
40%	58,95 sek	58,09 sek	59,73 sek
60%	38,03 sek	37,36 sek	37,87 sek
70%	30,61 sek	31,43 sek	31,45 sek
80%	26,21 sek	27,15 sek	26,33 sek
90%	22,01 sek	24,00 sek	22,74 sek
100%	19,45 sek	21,32 sek	19,47 sek
% kraft	*Tid i sekund per 10 meter.*		

Tabell 4: Sammenligning mellom rettlinjet bevegelse for fysisk og simulert bil

Det ble ikke observert nevneverdig differanse i ressursbruk mellom potensformelen og sigmoidformelen for testmaskinen[6] simulatoren ble kjørt på.

[6] Surface Pro 4 med Intel i7-6650U CPU og 8GB RAM

5.3.2 Sirkelbane-test

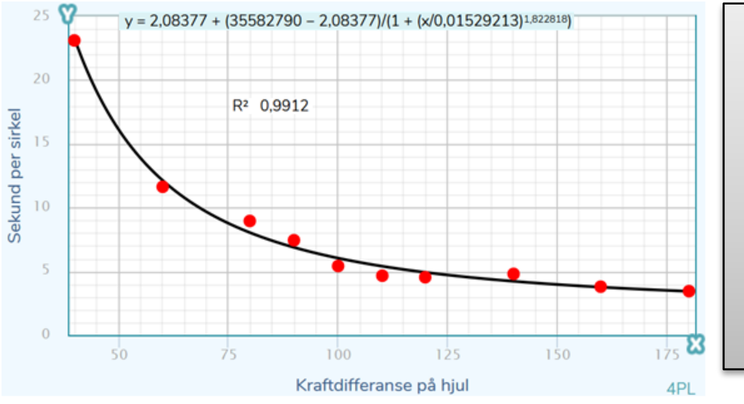
Disse testene er gjennomført ved at venstre hjul ble holdt konstant på 100% mens kraften på høyre hjul ble variert.

Differanse i settpunkt for venstre og høyre hjul	$\frac{sekund}{360°}$ (sekund per sirkel)	Diameter på sirkel
100-80=20%		2x 4.7m
100-60=40%	23,1 sek	2.75m
100-40=60%	11,63 sek	1.43m
100-20=80%	8,95 sek	1.0m / 1,127m
100-10=90%	7,44 sek	0,991
100-0=100%	5,42 sek	0.75m / 0,801
100-(-10)=110%	4,66 sek	0,729
100-(-20)=120%	4,54 sek	0,712
100-(-40)=140%	4,8 sek	
100-(-60)=160%	3,8 sek	
100-(-80)=180%	3,45 sek	

Tabell 5: Målinger for sirkelbane

Den uavhengige variabelen er differansen i settpunkt på hjulene. Dvs. prosentkraft på venstre hjul (som alltid er 100%) minus prosentraft på høyre hjul. Den avhengige variabelen er tiden bilen bruker på å fullføre en full sirkel.

Diameter på sirkelbanen er en ekstra måling som ikke brukes for formelen, men er tatt for å kunne sammenligne om svingradiusen blir omtrent den samme for simulert og fysisk bil.



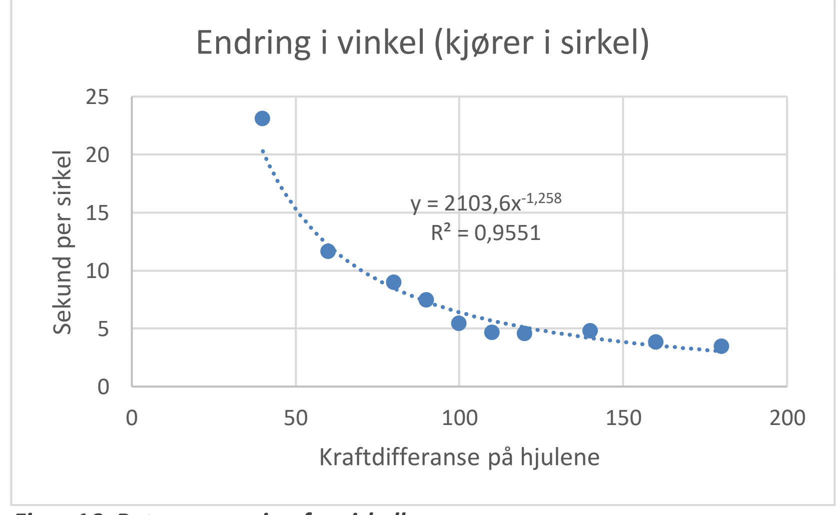
Symmetrisk sigmoid regresjon gav

R^2 på 0.9912

Med formel:

$$2{,}08377 + \frac{35582787{,}9162}{1 + \left(\frac{x}{0{,}01529213}\right)^{1.822818}}$$

Figur 15: Sigmoidregresjon for sirkelbane



Potensregresjon gav

R^2 på 0.9551

Med formel:

$$2103.6x^{-1.258}$$

Figur 16: Potensregresjon for sirkelbane

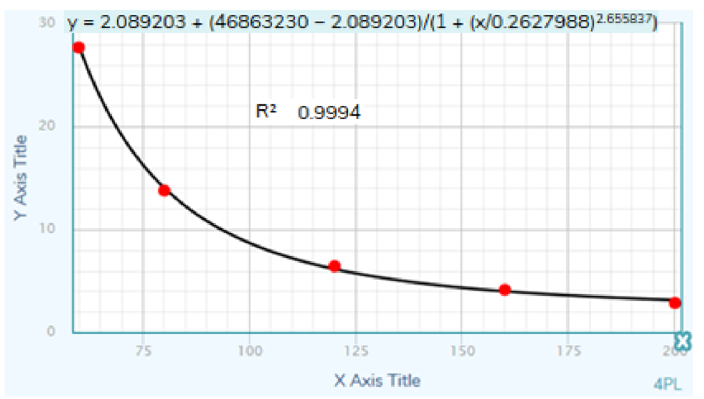
5.3.3 Rotasjonstest

Differanse i settpunkt for venstre og høyre hjul	$\frac{sekund}{360°}$ (sekund per rotasjon)
30-(-30)=60%	27,67 sek
40-(-40)=80%	13,76 sek
60-(-60)=120%	6,43 sek
80-(-80)=160%	4,13 sek
100-(-100)=200%	2,85 sek

Tabell 6: Målinger for rotasjon rundt egen

Her settes lik, men motsatt rettet prosentraft til hjulene på venstre og høyre side, slik at bilen roterer rundt sin egen senterakse. I.e. i en sirkel uten noen diameter.

Den uavhengige variabelen er differansen i settpunkt på hjulene, og den avhengige variabelen er tiden bilen bruker på å fullføre en full rotasjon. Det var nødvendig å utføre disse målingene hvor bilen roterer rundt sin egen akse, i tillegg til målingene i 5.3.2 hvor bilen kjører i en sirkelbane, ettersom førstnevnte krever mer kraft ettersom dekkene vil slepes mer over flaten.



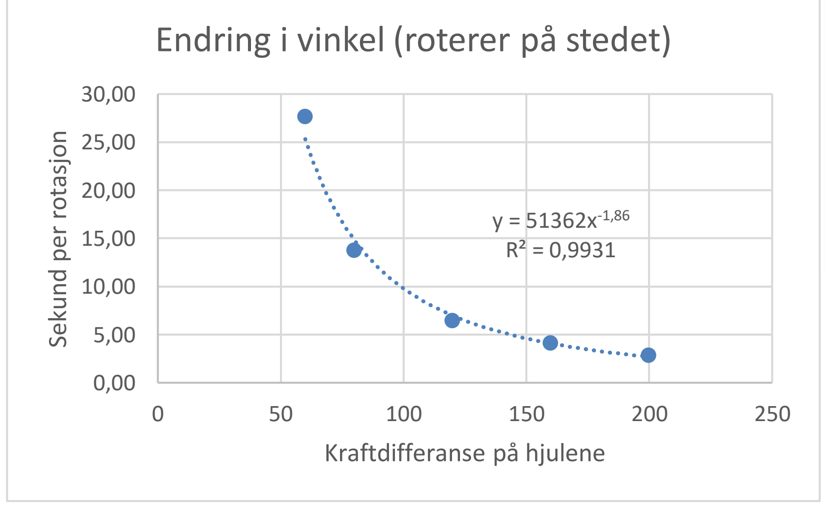
Figur 17: Sigmoidregresjon for rotasjon

Symmetrisk sigmoid regresjon gav:

R^2 på 0,9994

Med formel:

$$2{,}089203 + \frac{46863227{,}9108}{1 + \left(\frac{x}{0{,}2627988}\right)^{2{,}655837}}$$



Figur 18: Potensregresjon for rotasjon

Potensregresjon gav:

R^2 på 0,9931

Med formel:

$$51362x^{-1{,}86}$$

5.4 Strømmålinger

Det har blitt tatt noen enkle tester for å se hvor mye strømtrekk det er til de ulike komponentene på bilen. Til dette ble det brukt et multimeter av merke INSTRUTEK modell MAMY-60. Multimeteret ble brukt til å finne ut hvilke sikringer som skulle settes i strømforsyningen.

Fra et fulladet batteri (18 volt):
(SBC er inkludert vifte og skjerm)

- Når kun SBC med hovedprogrammet kjører: 0,34 ampere.
- SBC, motorer med full kraft og motstand (kjører rett fram): 1,5 ampere.
- SBC, motorer med full kraft og med maks motstand (hjul står nesten i ro): > 4 ampere.
- SBC og lidar: 0,85 ampere.
- SBC, lidar og motorer med motstand: ca. 2 ampere.

Fra 5 volts regulatoren:

- SBC: 0,9 – 1,2 ampere. Der 1,2 ampere var under mye databehandling fra lidaren.
- Mikrokontrollerkort: 0,3 ampere.

6 Oppsummering

Sensorbilen som har blitt utviklet gjennom dette bachelorprosjektet kan brukes som et læringsverktøy der studenter har mulighet til å selv anvende teori fra forskjellige emner i praksis. Studenter skiver sin egen kontrollogikk i programmeringsspråket C#, som de lærer i studiet. Koden kan enten kjøres lokalt på egen PC og bli testet med en simulator, eller den kan lastes direkte over til den fysiske bilen via skolens trådløse nettverk.

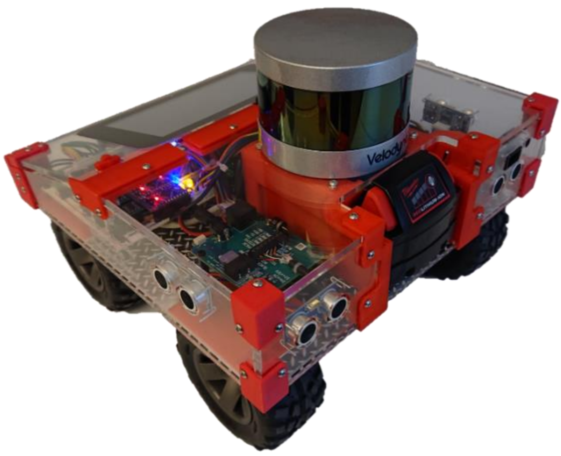
Sensorbilen møter kravspesifikasjonene gruppen har satt, og oppgaven har blitt løst med et langt bedre resultat enn først forventet. Dette kommer nok av at gruppen selv så potensiale i en slik sensorbil og definerte kravene selv. Flere ekstrafunksjoner utover kravspesifikasjonen har blitt lagt til bilen, og det er lagt til rette for videreutvikling av både programvare og maskinvare.

En av de største oppgavene var å få data fra sensorer og styring av fremdriften lett tilgjengelig for studenter som skal skrive en kontrollogikk. Dette ble løst med å utvikle et fleksibelt, objektorientert program som kjører på bilens hoveddatamaskin (SBC-en), noe som resulterte i at det ble enklere å legge til flere sensorer enn først antatt. Samtidig med utviklingen av programvare, ble maskinvaren utviklet i tre forskjellige iterasjoner. Funksjoner ble testet først, så ble designet utviklet, før den endelige bilen ble laget.

Siden gruppen har utviklet et læringsverktøy som kan brukes i flere av emnene de selv har hatt, er det blitt tilrettelagt for at alt av koden og annen informasjon skal være tilgjengelig for de som interesserte. Dette fordi kommende studenter skal lettere kunne forstå hvordan et større system kan være satt sammen, og hvordan store, objektorienterte program er bygget opp. Repositoriet «BO19E-15_SensorVehicle» har blitt opprettet på GitHub.com hvor alt fra kretstegninger til kode ligger tilgjengelig.

Ved å bygge bilen med hjelp av 3D-printer, laserkutter og ha alt av elektronikken synlig, er målet å fange interessen til studenter og vise hvordan alt er koblet sammen.

Gruppen har fått utnyttet kunnskap fra flere av emnene de har hatt gjennom studiet, men har også tilegnet seg mye ny kunnskap i forbindelse med prosjektet. Av dette vil vi trekke frem utvikling av moderne Windows-applikasjoner og oppbygningen av et totalt system helt fra maskinvarenivå til det brukeren ser på skjermen.



Figur 19: Sensorbil v.1.0

6.1 Forslag til forbedringer

For å sørge for at de forskjellige appene som er laget skulle kjøre så raskt som mulig, og at kvaliteten på videostrømmingen skulle bli bedre, ble det kjøpt inn en kraftig ettkortsdatamaskin, kalt UP Squared [9]. Denne har fire ganger så mye RAM[7] som RPi 3B og egen grafikkprosessor. UP skriver på nettsiden sin at det er mulig å konfigurere pinnene etter hva de skal brukes til [10]. Per 22.05.2019 stemmer ikke dette, og vi har lagt ut en post på UP Community som UP har sagt de vil kommentere på når de finner en løsning. Med dette kortet har man mulighet til å få utnyttet lidaren fullt ut da dette vil kreve mye datakraft. En annen ting som kan tas i bruk er datasyn.

Det ble også kjøpt inn gyroskop som ble planlagt brukt for å finne bilens retning. Dette ble ikke implementert grunnet tidsnød, men er noe som virkelig kan anbefales å få på plass. Retningen til bilen er noe som er forholdsvis enkelt å lage reguleringsøvinger om.

[7] Random Access Memory

7 Referanser

[1] Høgskulen på Vestlandet, «Om Høgskulen på Vestlandet,» 20 Desember 2018. [Internett]. Available: https://hvl.no/om/. [Funnet 9 Januar 2019].

[2] Cytron Technologies, «User's Manual V1.0,» Mai 2013. [Internett]. [Funnet 5 April 2019].

[3] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jefferies, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland og D. Thomas, «Manifesto for Agile Software Development,» Agile Alliance, 2001.

[4] SaschaIoT, «github.com,» 31 Desember 2017. [Internett]. Available: https://github.com/SaschaIoT/HttpWebcamLiveStream. [Funnet 28 Februar 2019].

[5] Open Source Initiative, «opensource.org,» [Internett]. Available: https://opensource.org/licenses/MIT. [Funnet 10 Mai 2019].

[6] Microsoft, «An overview of Windows 10 IoT Core,» Microsoft, 18 Januar 2018. [Internett]. Available: https://docs.microsoft.com/nb-no/windows/iot-core/windows-iot-core. [Funnet 24 Mai 2019].

[7] .NET foundation, «xUnit.net,» [Internett]. Available: https://xunit.net/. [Funnet 6 April 2019].

[8] Bosch, «Bosch PLR 15 Digital Laser Measure,» Bosch, [Internett]. Available: https://www.bosch-do-it.com/ae/en/diy/tools/plr-15-3165140765541-199929.jsp#tab_technical. [Funnet 8 April 2019].

[9] UP, «UP Squared Specifications,» [Internett]. Available: https://up-board.org/upsquared/specifications/. [Funnet 12 Mars 2019].

[10] Alingwu, «UP Squared Windows IoT Core image (beta),» UP, 22 Januar 2018. [Internett]. Available: https://downloads.up-community.org/download/up-squared-windows-iot-core-image-beta/. [Funnet 14 Mars 2019].

[11] Komplett.no, «Raspberry Pi 3 Model B,» [Internett]. Available: https://www.komplett.no/product/875746#productinfo. [Funnet 13 Mai 2019].

[12] Arduino, «Arduino Pro Mini - Tech Specs,» 2019. [Internett]. Available: https://store.arduino.cc/arduino-pro-mini. [Funnet 29 Januar 2019].

[13] NXP Semiconductors, «NXP Semiconductors,» 4 April 2014. [Internett]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf. [Funnet 13 Mai 2019].

[14] Texas Instruments, «TCA4311A Hot Swappable 2-Wire Bus Buffers,» 2018.

[15] Komplett, «Raspberry Pi Touch Screen 7",» [Internett]. Available: https://www.komplett.no/product/888104/datautstyr/pc-komponenter/hovedkort/tilbehoer/raspberry-pi-touch-screen-7#. [Funnet 15 Februar 2019].

[16] Calkboard Electronics, [Internett]. Available: https://www.chalk-elec.com/?page_id=1280#!/7-black-frame-universal-HDMI-LCD-with-capacitive-multi-touch/p/21750201. [Funnet 21 Mars 2019].

[17] Arduino Playground, «Reading Rotary Encoders,» [Internett]. Available: https://playground.arduino.cc/Main/RotaryEncoders/. [Funnet Mai 2019].

[18] Microsoft, «Hardware compability list,» 28 August 2017. [Internett]. Available: https://docs.microsoft.com/en-us/windows/iot-core/learn-about-hardware/hardwarecompatlist. [Funnet 14 Februar 2019].

[19] Velodyne LiDAR, «VLP-16 User's manual and programming guide,» Velodyne LiDAR, 2016.

[20] DimensionEngineering, «Sabertooth 2x12 User’s Guide,» 2012.

[21] Miwaukee Tool, «Features,» [Internett]. Available: https://www.milwaukeetool.eu/m18-50-ah-battery/m18-b5-eu/. [Funnet Mai 2019].

[22] Pololu, «Pololu 5V, 9A Step-Down Voltage Regulator D24V90F5,» [Internett]. Available: https://www.pololu.com/product/2866. [Funnet April 2019].

[23] MonoGame, «MonoGame,» [Internett]. Available: http://www.monogame.net/. [Funnet 24 05 2019].

[24] Tiled Documentation Writers, «Introduction - Tiled 1.2.4 Documentation,» 2019. [Internett]. Available: https://doc.mapeditor.org/de/latest/manual/introduction/. [Funnet 25 Mai 2019].

[25] Microsoft, «What is a UWP app,» 7 Mai 2018. [Internett]. Available: https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide. [Funnet 28 April 2019].

[26] Microsoft, «Choose a UWP version,» 19 April 2019. [Internett]. Available: https://docs.microsoft.com/en-us/windows/uwp/updates-and-versions/choose-a-uwp-version. [Funnet 28 April 2019].

[27] Microsoft, «App capability declarations,» 19 April 2019. [Internett]. Available: https://docs.microsoft.com/en-us/windows/uwp/packaging/app-capability-declarations. [Funnet 28 April 2019].

[28] Microsoft, «Sockets,» 3 Juni 2018. [Internett]. Available: https://docs.microsoft.com/en-us/windows/uwp/networking/sockets. [Funnet 28 April 2019].

[29] Microsoft, «Create and consume an app service,» 16 Januar 2019. [Internett]. Available: https://docs.microsoft.com/en-us/windows/uwp/launch-resume/how-to-create-and-consume-an-app-service. [Funnet 29 April 2019].

[30] D. Jacobson, «.NET Native - What it means for Universal Winodws Platform (UWP) developers,» 20 August 2015. [Internett]. Available: https://blogs.windows.com/buildingapps/2015/08/20/net-native-what-it-means-for-universal-windows-platform-uwp-developers/#xzWrurTq20D347Ec.97. [Funnet 28 April 2019].

[31] D. Jacobson, «Microsoft .NET - .NET and Universal Windows Platform Development,» Oktober 2018. [Internett]. Available: https://msdn.microsoft.com/en-us/magazine/mt590967.aspx. [Funnet 12 Mars 2019].

[32] Microsoft, «github - Windows Template Studio,» Microsoft, [Internett]. Available: https://github.com/Microsoft/WindowsTemplateStudio. [Funnet 28 April 2019].

[33] Microsoft, «Databinding and MVVM,» 10 Februar 2018. [Internett]. Available: https://docs.microsoft.com/en-us/windows/uwp/data-binding/data-binding-and-mvvm. [Funnet 7 April 2019].

[34] Prism, «PrismLibrary GitHub,» [Internett]. Available: https://prismlibrary.github.io/docs/. [Funnet 29 April 2019].

[35] Microsoft / OpenSource, «Template 10 (github),» [Internett]. Available: https://github.com/Windows-XAML/Template10. [Funnet 29 April 2019].

[36] Microsoft, «Interfaces (C# Programming Guide),» 21 August 2018. [Internett]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/. [Funnet 29 April 2019].

[37] Microsoft, «Polymorphism (C# Programming Guide),» 20 Juli 2015. [Internett]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism. [Funnet 29 April 2019].

[38] L. Bugnion, «MVVM - IOC Containers and MVVM,» Februar 2013. [Internett]. Available: https://msdn.microsoft.com/en-us/magazine/jj991965.aspx. [Funnet 24 Mai 2019].

[39] HiB, UiB, NHH, UiO og Nasjonalbiblioteket, «Søk og Skriv,» 12 12 2014. [Internett]. Available: http://sokogskriv.no/. [Funnet 12 12 2014].

[40] HackerBoards.com, «PocketBeagle - Product Spesification,» 29 Januar 2019. [Internett]. Available: https://www.hackerboards.com/product/204/.

[41] Høgskulen på Vestlandet, «Campus Bergen,» 7 Desember 2018. [Internett]. Available: https://hvl.no/studentliv/studentliv-bergen/campus-bergen/. [Funnet 9 Januar 2019].

[42] IvenSence Inc. , «MPU-6000 and MPU-6050 Register Map and Description Revision 4.2,» 8 August 2013. [Internett]. Available: https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf. [Funnet 13 Mars 2019].

[43] Arduino Playground, «MPU-6050 Accelerometer + Gyro,» 2019. [Internett]. Available: https://playground.arduino.cc/Main/MPU-6050/. [Funnet 13 Mars 2019].

[44] Microsoft IoT Core, «Raspberry Pi 2 & 3 Pin Mappings,» 28 August 2017. [Internett]. Available: https://docs.microsoft.com/en-us/windows/iot-core/learn-about-hardware/pinmappings/pinmappingsrpi. [Funnet 30 Januar 2019].

Appendiks A Lister

A.1 Forkortelser og ordforklaringer

GUI	Graphical User Interface
http	Hypertext Transfer Protocol
I^2C	Inter-Integrated Circuit
IoT	Internet of Things
Lidar	Light Detection and Ranging
MIT	Massachusetts Institute of Technology
RAM	Random Access Memory
RPi	Raspberry Pi
SBC	Single-Board Computer
SPI	Serial Peripheral Interface

A.2 Figurliste

Figur 1: Opprinnelig bil. Bilde tatt høst 2018 10
Figur 2: Første skisse av ønsket design 10
Figur 3: Sensorbil v.0.1 13
Figur 4: Øverst: Sensorbil v.0.2. Nederst: Forbedret koblinger mellom komponenter 14
Figur 5: Øverst: Sensorbil v.1.0 Nederst: Sensorbil v.1.0 topp. 14
Figur 6: Enkel systemskisse 15
Figur 7: Simpel kontroll logikk som bruker for å kjøre mot største avstand 17
Figur 8: Feilmelding ved sending av melding på ugyldig format 22
Figur 9: Detaljert feilmelding 22
Figur 10: Hovedprogrammets startside på PC uten simulator 23
Figur 11: Simulatoren er lagt ut på Microsoft Store 23
Figur 12: Systemskisse 26
Figur 13: Symmetrisk sigmoid regresjon for rettlinjet bevegelse 30
Figur 14: Potens regresjon for rettlinjet bevegelse 30
Figur 15: Sigmoidregresjon for sirkelbane 31
Figur 16: Potensregresjon for sirkelbane 31
Figur 17: Sigmoidregresjon for rotasjon 32
Figur 18: Potensregresjon for rotasjon 32
Figur 19: Sensorbil v.1.0 34
Figur 20: Boks for utfylling av IP-adresse 46
Figur 21: Chassiset 49
Figur 22: SBC Raspberry PI 3B 49
Figur 23: RPi pinout, hentet fra [44] 49
Figur 24: Kretstegning av GPIO på RPi 49
Figur 25: Mikrokontroller Arduino Pro Mini 50
Figur 26: Øverst: Bilde av buffer ic Nederst: koblinger, fra datablad [14] 51
Figur 27: Kobling av I^2C til Arduino 51
Figur 28: Kretstegning av mikrokontroller til ultralydsensorer 53
Figur 29: Kode fra mikrokontrolleren til ultralydsensorene 53

Figur 30: Pulser for kanal A og B, hentet fra [17] 54
Figur 31: Figur over lidar, hentet fra manual [19] 55
Figur 32: Kretstegning av strømforsyning....58
Figur 33: Klassediagram for "Sensorer/Utstyr" 69
Figur 34: Klassediagram for "Kontrollogikk"....71
Figur 35: Klassediagram for "Socket Server" 72
Figur 36: Klassediagram for "ViewModels og "Views" (GUI) 73
Figur 37: Oversiktsbilde av Tiled....84
Figur 38: Vindu som spretter opp ved åpning av installasjonsguide for Tiled 84
Figur 39: Innstillinger for tegning av kart 84
Figur 40: Tiled ved oppretting av nytt kart 85
Figur 41: Oversikt over Tile Layers....85
Figur 42: Sørg for at Infinite ikke er huket av 85
Figur 43: Innstillinger for .png og .tmx 86

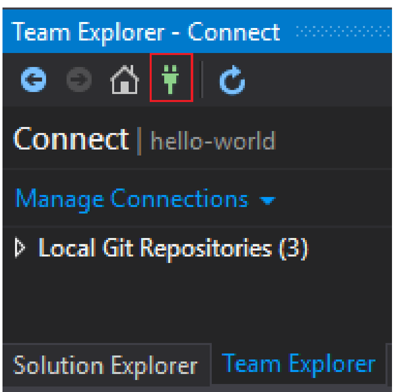
Tabell 1: Kontrollsjekk av avstandsmålingene fra ultralyd, tatt i en romtemperatur på 23.3°C 28
Tabell 2: Kontrollsjekk av encodermålingene ved forskjellige hastigheter 29
Tabell 3: Målinger for rettlinjet bevegelse 30
Tabell 4: Sammenligning mellom rettlinjet bevegelse for fysisk og simulert bil 30
Tabell 5: Målinger for sirkelbane....31
Tabell 6: Målinger for rotasjon rundt egen 32

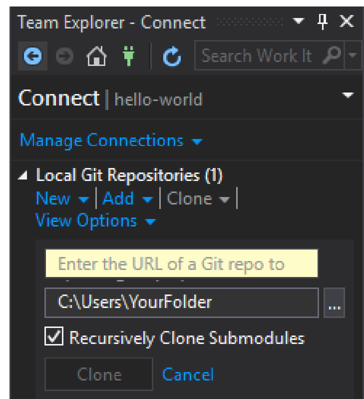
Appendiks B Git – versjonskontroll og samarbeid på kode

Git er et verktøy for versjonskontroll av kode. Jobber man alene kan det være nyttig å ha et versjonskontrollsystem for å ha mulighet til å gå tilbake til versjoner man vet fungerer, eller å se hvilke deler av koden som legger til ønsket funksjonalitet. Er man et team som jobber på en felles kodebase er det tilnærmet essensielt.

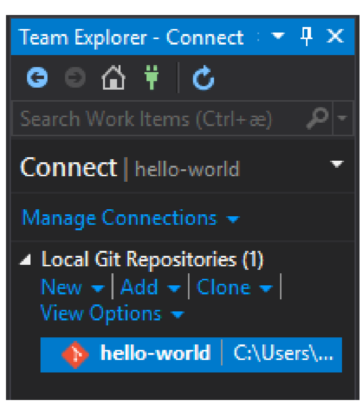
B.1 Bruk av GitHub i Visual Studio

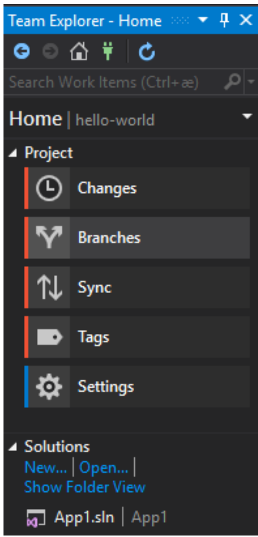
For å få tilgang til GitHub-funksjonaliteten i Visual Studio må man åpne Team Explorer. Dette gjør man ved å velge «View» → «Team Explorer» fra menyen øverst i Visual Studio. For å klone repositorier fra GitHub åpner man *Local Git Repositories* og velger «Clone». Man limer så inn URL-en til det aktuelle repositoriet og velger hvor man skal lagre det. URL-en finner man ved å finne repositoriet på github.com og trykker «Clone or download».





Etter at man har klonet og valgt det repositoriet man ønsker å jobbe i, får man opp menyen som er vist på bildet under til høyre.

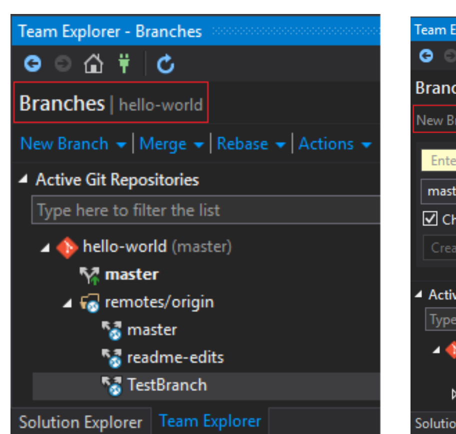


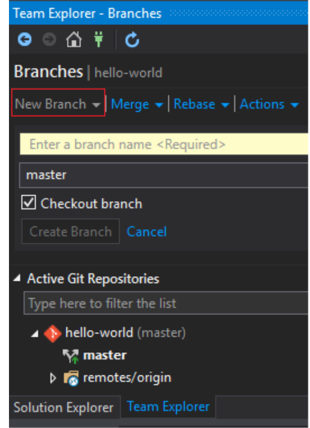


Changes viser endringer man har gjort i arbeidsmappen og som ikke er synkronisert med filene som ligger på GitHub. Dersom man høyreklikker på en fil man får opp en meny med noen valg. «Compare with Unmodified…» lar brukeren sammenligne filen han har jobbet med, med den tilsarende filen som ligger på GitHub. Dersom det har blitt gjort endringer i flere filer, og man vil separere disse endringene i flere comitter, kan man gjøre dette enkelt ved å bruke «Stage». Man kan se på comitter

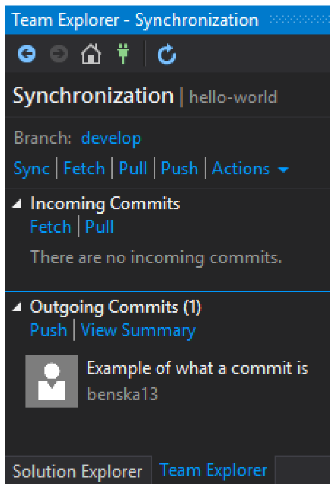
som endringer man ønsker å ta med seg videre. «Undo Changes» sletter endringene som er gjort i den valgte filen.

Branches viser en oversikt over de ulike greienene i repositoriet. Under repositorienavnet finner man greienene man har lokalt på PC-en, og greinene som ligger under *remotes/origin* viser greinene som ligger på GitHub. Grovt forenklet er en grein en unik versjon av koden hvor man kan gjøre endringer uten å påvirke de andre versjonene. Å lage en ny grein er nyttig dersom man skal legge til ny funksjonalitet eller endre store deler av koden. Alle repositorier skal ha en mastergrein (master) og en utviklingsgrein (develop). Mastergreinen er den første greinen man lager, og den greinen alle endringene til slutt skal *merges* inn i. Utviklingsgreinen skal lages ut fra mastergreinen, og alle de senere greiner skal så lages ut fra utviklingsgreinen. *Merge* betyr å flette sammen, og brukes når man har gjort endringer i en grein som man vil flette inn i en annen grein.





Sync tar seg av synkroniseringen av greinen man jobber i. *Fetch* vil hente endringer som ligger på GitHub, men som brukeren ikke har på sin PC. Disse vil vises under «Incoming Commits». *Pull* vil hente endringene og fletter disse inn i greinen man har lokalt. *Push* sender «Outgoing Commits» til GitHub. *Sync* vil gjennomføre en *pull* og så en *push*.

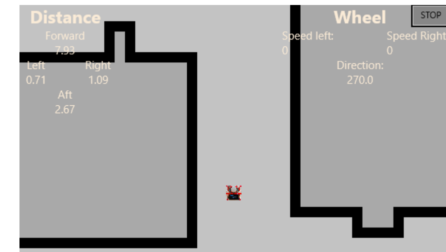


Appendiks C Brukerdokumentasjon

C.1 Installasjon av nødvendig programvare

C.1.1 *Installere simulator*

Det er sterkt anbefalt å installere simulatoren.

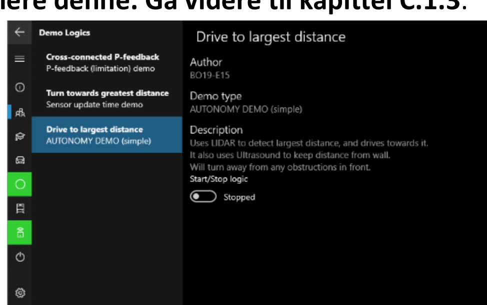


Denne lar deg enkelt teste kontrollogikken din uten å ha tilgang til den fysiske bilen. Så lenge den er installert på PC-en, så vil den startes automatisk når du kjører hovedprogrammet med koden din på din egen PC.

«HVL SensorVehicle Simulator» kan installeres fra Microsoft Store.

C.1.2 *Valgfritt - Installere hovedprogram*

Hvis du skal lage egen kontrollogikk trenger du ikke å installere denne. Gå videre til kapittel C.1.3.



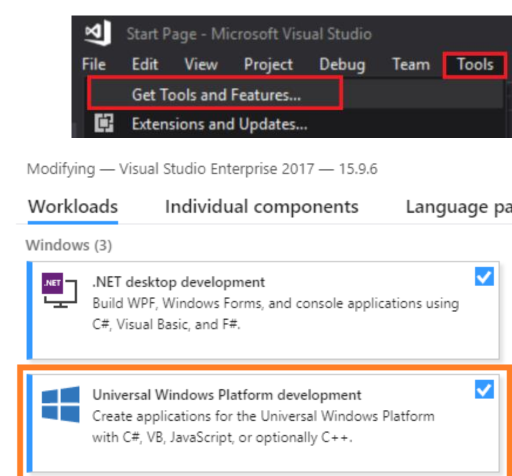
Dersom du **ikke** ønsker å lage egen kontrollogikk så er det også mulig å laste ned hovedprogrammet «HVL SensorVehicle Main» fra Microsoft Store.

Dette vil ha begrenset nytteverdi, men kan være greit for å se hvordan det grafiske grensesnittet ser ut. Man kan også kjøre noen av demologikkene på simulatoren.

C.1.3 *UWP utviklingsverktøy for Visual Studio*

Vi tar her utgangspunkt i at du allerede har Visual Studio installert på din PC.

Hovedprogrammet som kontrollogikken skal skrives i er en UWP-applikasjon. Derfor må «Universal Windows Platform development» er installert.



I Visual Studio velger man på menylinjen:
«Tools» → «Get Tools and Features»

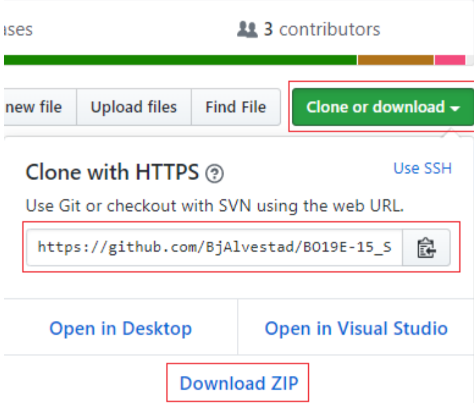
I vinduet som åpner seg må «Universal Windows Platform development» være valgt.

Dersom denne allerede er haket av, så trenger du ikke å gjøre noe. Hvis ikke så må du hake av og installere denne «Workloaden»

Hvis du ikke har Visual Studio fra før:
Last ned og installer Visual Studio IDE. Når du skal installere vil du få samme valgmulighet som vi har vist ovenfor. Huk av for «Universal Windows Platform development» under «Workloads».

C.2 Lage egen kontrollogikk

Det første man må gjøre når man skal lage egen kontrollogikk er å sørge for at man har lastet ned repositoriet [BO19E-15_SensorVehicle-Simplified](). Dette kan man gjøre ved å følge linken og klone repositoriet (som beskrevet i B.1) eller trykke «Clone or Download» → «Download ZIP». Når .zip-filen er lastet ned må man pakke den ut, og finner .sln-filen i mappen «Sensorvehicle-main».



Man navigerer seg så frem til en av StudentX.cs-filene og fyller inn tittel, forfatter og beskrivelse på kontrollogikken.

```
2 references | BjAlvestad, 40 days ago | 1 author, 1 change
public class Student1A : StudentLogicBase
{
    21 references | BjAlvestad, 40 days ago | 1 author, 1 change
    public override StudentLogicDescription Details { get; }

    1 reference | BjAlvestad, 40 days ago | 1 author, 1 change
    public Student1A(IWheel wheel, IEncoders encoders, ILidarDistance lidar, IUltrasonic ultrasonic) : base(wheel)
    {
        Details = new StudentLogicDescription
        {
            Title = "Empty logic for Student 1 (A)",
            Author = "Student 1",
            Description = "This program is empty.\n" +
                          "Write your control logic in the looping Run() method.\n" +
                          "(And change Title, Author and Description).\n" +
                          "All this is done in the Student1A.cs file."
        };

        _wheels = wheel;
        _lidar = lidar;
        _ultrasonic = ultrasonic;
        _encoders = encoders;
    }
```

Det er i metodene Initialize() og Run() man skriver kode som er del av kontrollogikken. Se kommentarer på bildet under.

```
public override void Initialize()
{
    // If you have any code you want to run ONCE when starting the control logic, you may put that here (optional).
    // E.g. setting desired number of collection cycles for LIDAR, or default vertical angle etc.

}

private IWheel _wheels;
private ILidarDistance _lidar;
private IUltrasonic _ultrasonic;
private IEncoders _encoders;
8 references | BjAlvestad, 50 minutes ago | 1 author, 2 changes
public override void Run(CancellationToken cancellationToken)
{
    // ** WRITE YOUR CONTROL LOGIC HERE **
    // The Run() method will loop until control logic is stopped (or exception occurs)
    // If you write your own loops inside here,
    // then remember to add '&& !cancellationToken.IsCancellationRequested' to the loop check
}
```

Metodene for «_wheels», «_lidar» osv. har kodebeskrivelse som er kompatibel med IntelliSense, slik at når man skriver «.» etter en av disse objektene, så får man opp en liste med mulige kommandoer og en beskrivelse av hva disse gjør.

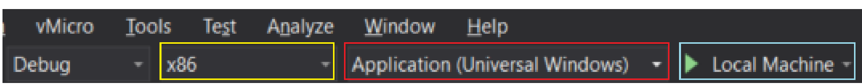
C.3 Kjøring av hovedprogrammet

Husk å bruke «Run Without Debugging» (CTRL + F5), med mindre man faktisk har behov for å debugge.

C.3.1 *Førstegangskjøring på egen PC (med simulator)*

Dersom man skal kjøre programmet på sin egen PC velger man *x86* i nedtrekksmenyen markert i gult, *Application (Univeral Windows)* i nedtrekksmenyen markert i rødt, og *Local Machine* i nedtrekksmenyen markert i blått. Man må så deployere *Application (Universal Windows)*. Dette gjøres ved å høyreklikke på *Application (Universal Windows)* i *Solution Explorer*, for så å velge *Deploy*. Når deployeringen er ferdig trykker man på *Local Machine*, og hovedprogrammet og simulatoren vil starte.

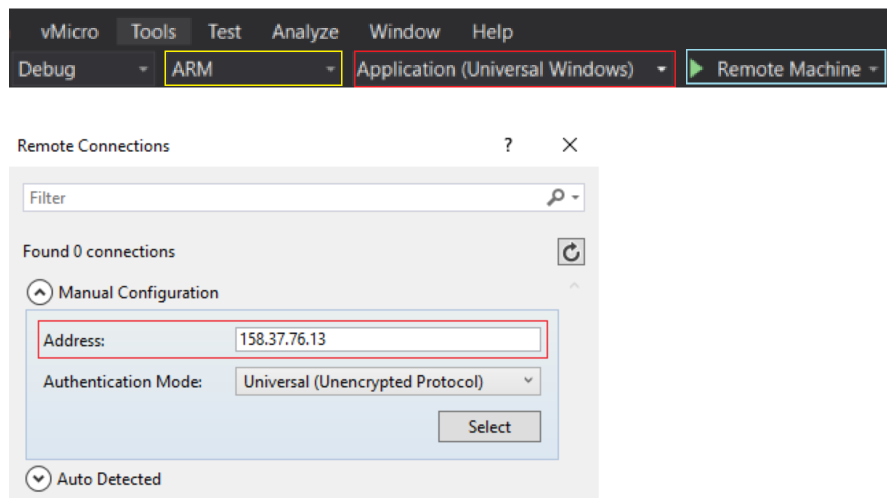
MERK: Første gang programmet kjøres kan det ta litt tid.



C.3.2 *Førstegangsopplasting til fysisk bil*

For å laste opp programmet til den fysiske bilen velger man *ARM* i nedtrekksmenyen vist i den gule rammen på bildet under, *Application (Universal Windows*) i nedtrekksmenyen vist i den røde rammen, og *Remote Machine* i nedtrekksmenyen som er vist i den blå rammen. Når man velger *Remote Machine* spretter boksen som er vist i Figur 20 opp. Sørg for å skrive inn korrekt IP-adresse her. Dersom man senere vil kjøre koden på en annen IP-adresse, kan man endre dette ved å velge *Debug → Application properties...*

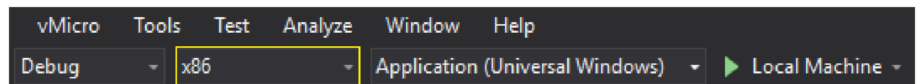
MERK: Første gang man laster opp til bilen kan det ta litt tid.



Figur 20: Boks for utfylling av IP-adresse

C.3.3 *Bytte mellom kjøring på PC og fysisk bil*

Etter man har kjørt programmet både på PC og fysisk bil endrer man enkelt hvor koden skal kjøres ved å endre *x86* til *ARM* eller motsatt i den gule rammen på bildet under. For å kjøre koden velger man «Run without debugging» (CTRL + F5).

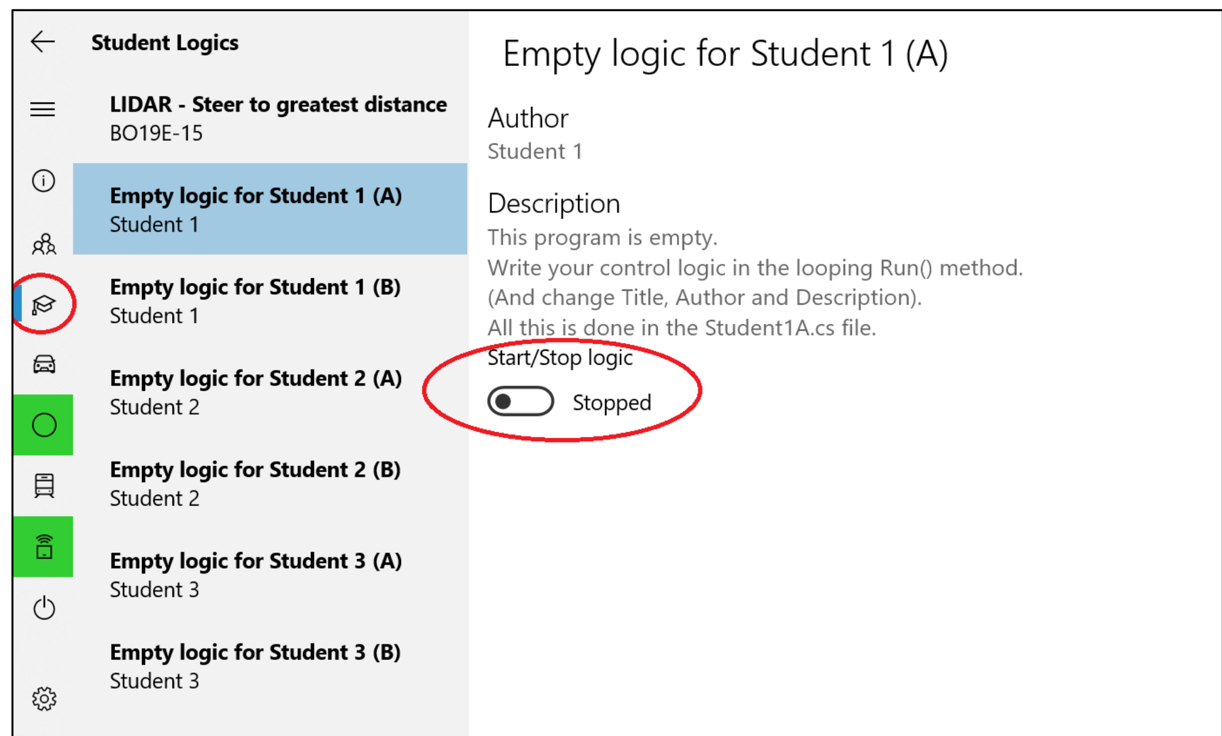


C.4 Brukergrensesnitt for program

Brukergrensesnittet er det samme enten hovedprogrammet kjører på din egen PC eller på bilen.

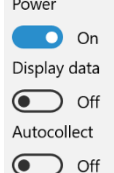
C.4.1 *Starte din egen kontrollogikk*

Enten man starter hovedprogrammet på sin egen PC eller den fysiske bilen vil man kunne starte ønsket kontrollogikk fra skjermsiden «Student Logics.



Sensorer det er ønskelig å brukt i kontrollogikken må være slått på via sensorens side før bruk. Det er også mulig å slå den på ved å sette *Power*-propertien til *true* i *initialize()*-metoden, slik at slik at det skjer automatisk i det man slår på kontrollogikken.

C.4.2 *Bryterne på sensorsiden*



På sensorsidene varierer det litt hvilke brytere man finner. Felles for dem alle er at de har en «Power»-bryter som slår på/av strøm til komponenten. Det er også en «Display data»-bryter som bestemmer om sensordataene skal trigge «Property Changed»-notifikasjoner. «Display data» må være på for å kunne se sensordataen på skjermen. Men ettersom dette vil føre til at kontrollogikken vil kjøre senere, kan brukeren selv styre om denne skal være på eller ikke.

«Autocollect»-knappen gjør at programmet automatisk spør etter data. Hvis denne er slått av så vil ikke måledataen oppdateres med mindre dette gjøres fra en kontrollogikk som kjører.

C.4.3 *Simulator*

Simulatoren startes automatisk når hovedprogrammet starter. En kort forklaring på simulatoren er tilgjengelig på hovedprogrammets startside.

C.5 Ekstrainformasjon til brukere

C.5.1 *Videostrømming*

Hvis man i hovedprogrammet går inn på «Settings»-siden, og trykker «Open extra functions», så åpnes en ny app hvor man kan slå på videostrømming til nettside.

Man kan da observere denne videostrømmingen ved å koble seg til bilens IP-adresse i en nettleser[8].

C.5.2 *Dioder og brytere*

Fra mikrokontrolleren til ultralydsensorene går det et signal til SBC om at det er foretatt nye målinger på alle sensorene og SBC-en ikke har spurt etter data. Dette kan man se på den blå lysdioden. Fra samme mikrokontroller går det to signal til mikrokontrolleren for framdrift, her blir det telt binært (0-3) som viser avstanden. Dette kan man se på de to gule lysdiodene.

Bryter 1, reduserer sikkerhetsavstanden før bilen stopper. Den er koblet til mikrokontrolleren for ultralyd.

Bryter 2, kobler vekk sikkerhetsfunksjonen og setter hastigheten i en redusert modus.

C.5.3 *Tilgang til all kode*

For at det skal være enkelt å komme i gang har vi tidligere i brukerdokumentasjonen henvist til koden som ligger i repositoriet BO19E-15_SensorVehicle-Simplified I det repositoriet ligger bare hovedprogrammet, og i hovedprogrammet er mange av prosjektene «skjult» som *.dll*-filer for å unngå at det skal bli for overveldende for studentene.

Dersom man ønsker å se all kode som er laget for bilen, så er denne tilgjengelig i repositoriet BO19E-15_SensorVehicle på GitHub.com.

C.5.4 *Liste over MAC- og IP-adresser*

SBC / WiFi-dongle for SBC:

RaspPi - Sensorbil v.1.0	b8-27-eb-b3-4e-36	158.37.76.13 (fast)
RaspPi – Prototype 2:	b8-27-eb-c4-57-99	158.37.76.20 (fast)
Wireless dongle (Up2):	50-3E-AA-B8-59-AB	158.37.76.21 (fast)

Lidar – VLP-16:

Serienummer:	MAC adresse:	IP adresse (fast):
AG28920118	60:76:88:10:4E:96	192.168.1.201

[8] Det har tidvis vært problemer med tilkobling via Mozilla Firefox

Appendiks D Maskinvare – Bilens oppbygging

Den gamle bilen ble helt demontert for å kunne starte på nytt med å bygge den nye sensorbilen. I chassiset som vist i Figur 21 er motorer med gir og encodere. Girene ble smurt, encodere og motorene fikk nye ledninger og overhalt loddinger. Lydisoleringsmatter ble limt på veggene inni chassiset for å dempe lyd fra motor og gir.

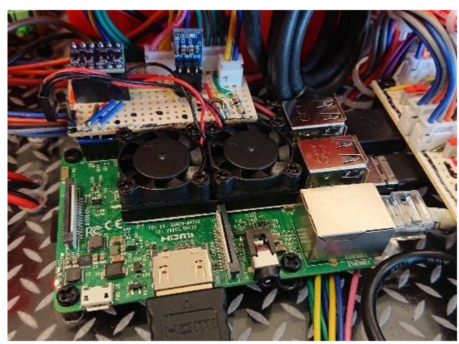
Figur 21: Chassiset

D.1 Prosesseringsenheter

D.1.1 *Raspberry Pi 3B*

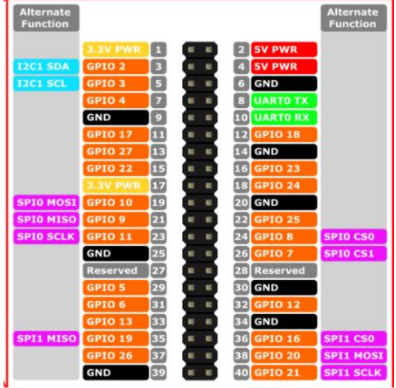
Raspberry Pi 3B er en tredjegenerasjons Rasberry Pi som er en ettkortsdatamaskin (SBC) med: [11]

- 1,2 GHz 64-bits Quad Core ARM Cortex-A53 CPU
- 802.11n trådløs LAN
- Bluetooth Low Energy (BLE)
- 4 USB-porter
- 40 GPIO pins
- Full HDMI-port
- Ethernet-port
- Kombinert 3,5 mm audio jack og komposittvideo
- Kamera-grensesnitt (CSI)
- Display-grensesnitt (DSI)
- Micro SD-kortspor
- VideoCore IV 3D grafikkjerne

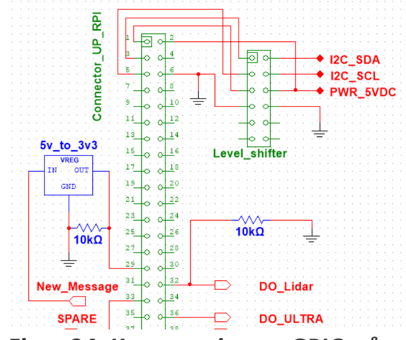


Figur 22: SBC Raspberry PI

Kortet har to rekker med digitale inn- og utganger. På bilen blir I^{2}C brukt til kommunikasjon til mikroprosessorene og GPIO[9] til å slå av og på strøm til mikroprosessorene.



Figur 23: RPi pinout, hentet fra [44]



Figur 24: Kretstegning av GPIO på RPi

GPIO på RPi bruker 3,3 volt, mens mikroprosessorene vi har valgt bruker 5 volt. Derfor må vi ha en «level converter» på I^{2}C-signalene.

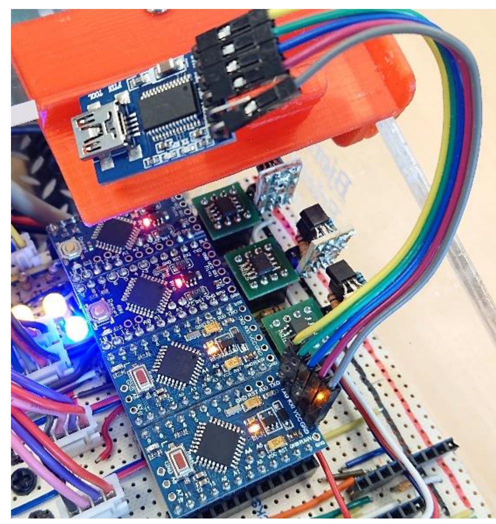
På «ny data» fra ultralyd brukes en 5 til 3,3 volts regulator. Her ble det først prøvd å bruke «level converteren», men det oppstod da et problem med at den holder signalet høyt til mikroprosessoren har slått seg på. Da ble signalet tolket feil i programmet på RPi, og det ble en «exception».

9 General-Purpose Input/Output

RPi kommer uten kjøling på prosessoren. Etter noen tester av programmet som er blitt laget, ble det kjøpt inn en kjøleribbe med vifte grunnet oppvarming av prosessorchippen. Viften koblet rett på 5 volt, men en mer elegant løsning hadde vært å lage en regulering som styrer viften etter temperaturen i prosessoren.

D.1.2 *Arduino Pro Mini*

Arduino Pro Mini er en mikrokontroller basert på prosessoren ATmega328P og er laget for bruk i prosjekter med permanent oppsett. Den har blant annet ikke USB-port på kortet. For å programmere må man ha en ekstern seriellport, f.eks. «TTL-232R USB - TTL Level Serial», og man kobler seg på RX, TX, reset, 5V og GND.



Figur 25: Mikrokontroller Arduino Pro Mini

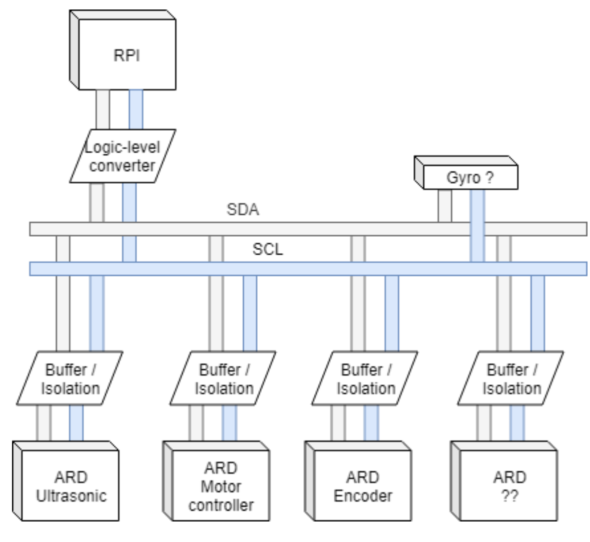
Til programmering brukes Visual Micro, som er en tilleggspakke til Visual Studio, og språket er C++.

Spesifikasjoner: [12]

- 3.35 -12 V (3.3V model) or 5 - 12 V (5V model)
- Digital I/O Pins 14
- PWM Pins 6
- UART 1
- SPI 1
- I2C 1
- Analog Input Pins 6
- External Interrupts 2
- DC Current per I/O Pin 40 mA
- Flash Memory 32KB of which 2 KB used by bootloader
- SRAM 2 KB
- EEPROM 1 KB
- Clock Speed 8 MHz (3.3V versions) or 16 MHz (5V versions)

D.1.3 *Kommunikasjon*

Under produksjonen av den første prototypen, v.0.1, ble det testet og sett på flere kommunikasjonsprotokoller mellom mikroprosessorene og SBC-en. Disse presenteres under.



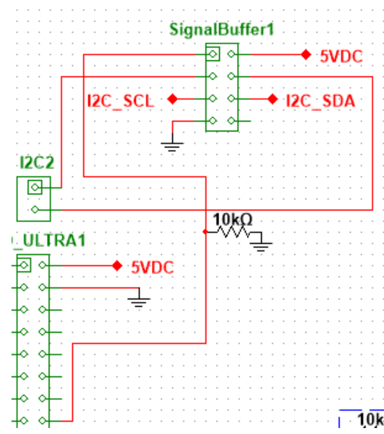
SPI (Serial Periperal Interface): Synkron, seriell databuss som arbeider i full dupleks med egen klokkelinje. Denne ble lagt bort da det ville kreve en del mer kabling når det skal kommuniseres med flere mikrokontrollere.

Seriell (RX, TX): Ble sett på for å sende hjulkommandoer, men ble lagt bort da vi måtte

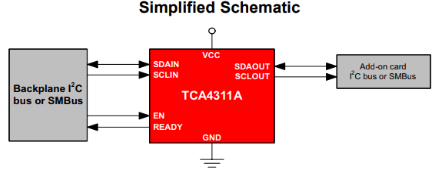
lage en egen meldingsprotokoll for å få det til å virke effektivt og det da var lettere å bygge denne videre fra en ferdig protokoll.

Kommunikasjonsprotokollen som ble valgt er I²C, og står for Inter-Integrated Circuit. Dette er en master/slave seriell databuss utviklet av Philips Semiconductor i 1982, opprinnelig til bruk internt i TV-er [13]. Nå er I²C populær til bruk mellom sensorer, mikrokontrollere, displayer og mer. Det er laget ferdige kodebibliotek til Arduino og til Windows 10 IoT Core, noe som gjør det enkelt å komme i gang.

Det trengs to buss-ledere, en «Serial Data Line» (SDA) og en «Serial Clock Line» (SCL). I²C har en standard overføringshastighet på 100 kbit/s, men kan og settes til 400 kbit/s som er «Fast-mode». På bilen fungerer ettkortsmaskinen som master, og er satt til «Fast-mode». Nye moduler kan kobles rett på bussen, og i hovedprogrammet er det laget en enum med alle modulene og adressene.



Figur 27: Kobling av I²C til Arduino



Figur 26: Øverst: Bilde av buffer ic
Nederst: koblinger, fra datablad [14]

Et problem med I²C og strømstyring til mikrokontrollerene var at når strømmen ble brutt så trakk pull-up motstandene til I²C -bussen på Arduinoene, spenningen ned, og ødelagte for andre enheter på bussen. For å løse dette brukes en «Hot Swappable 2-Wire Bus Buffer»-IC [14] som kan isolere mikrokontrolleren fra bussen. Når mikrokontrolleren har slått seg på, aktiverer den IC-en på Enable-inngangen.

Det ble laget en egendefinert meldingsprotokoll som spesifiserer hvordan meldingene som blir sendt med I²C skal være utformet.

- Første byten er adressen hvor datapakken kommer fra eller blir sendt til.
- Andre byten er en meldingskode som brukes til å overføre feilmeldinger eller andre innstillinger. Dette er ikke tatt i bruk under denne utviklingen av bilen.
- Tredje byten er antall heltall som sendes med datapakken
- Videre blir det antall heltall som skal sendes fordelt ved hjelp av bit-shift.

```
public static byte[] ToByteArray(int byteArraySize, Device deviceAddress, MessageCode message, params int[] integers)
{
    byte[] byteArray = DisassembleIntsToByteArray(byteArraySize, emptyElements:3, integers);
    byteArray[0] = (byte) deviceAddress;
    byteArray[1] = (byte) message;
    byteArray[2] = (byte) integers.Length;

    return byteArray;
}
```

Se i Arduino-koden og «VehicleCommunication.cs» under Communication.Vehicle på GitHub for full forklaring.

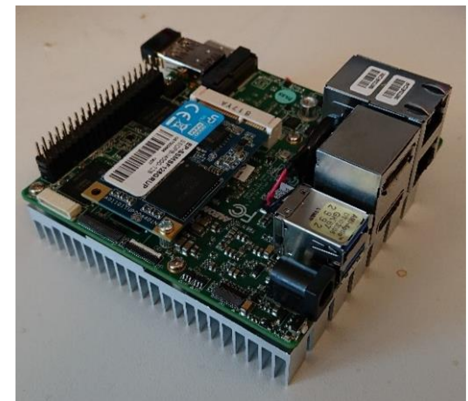
D.1.4 *Skjerm*

Det har blitt kjøpt inn to forskjellige touchskjermer, en «Raspberry Pi Touch Screen 7"» [15] og en «7" black frame universal HDMI LCD with capacitive multi-touch» [16].

RPi sin skjerm kobles rett på displayutgangen på RPi-kortet, her er touch og bilde i én kabel. Skjermen har en oppløsning på 800x480 pixler.

Universalskjermen har HDMI port og USB for touch og strøm. Oppløsningen er på 1280x800 pixler. Denne skjermen ble kjøpt for å gi en bedre brukeropplevelse med både oppløsning og innsynsvinkel som er på 89° i alle retninger. Skjermen kan også kobles på andre datamaskiner om RPI skal byttes ut.

D.1.5 *UP-Board*



Det ble godkjent å kjøpe inn en kraftigere datamaskin enn RPi for å ha større utviklingsmuligheter og bedre brukeropplevelse. Kravene vi så etter:

- Mulig å installere Windows 10 IoT Core
- Ha egen dedikert GPU-prosessor
- Lik GPIO pinner som RPI
- USB, HDMI, nettverkskort
- Mer prosessorkraft enn RPi

Egen dedikert GPU-prosessor var ønskelig for å få en mer effektiv prosessering av det som skal vises på skjermen. Samt at man kan i framtiden ta i bruk webkamera og/eller lidar for å prosessere bilder og vise disse direkte på skjermen.

Det ble sett på flere forskjellige SBC-er, men mange faller bort da de ikke kan kjøre Windows IoT eller har lik GPIO som RPi. De som ble vurdert var kort fra DragonBoard, Minnowboard og UP. UP Squared ble valgt da det så ut som det var godt tilpasset Windows 10 IoT Core og det var enkelt å koble til flere komponenter. Det ble kjøpt med en SSD på 128 GB som festes rett på hovedkortet. Tanken med SSD var for å kunne ta i bruk database lokalt på datamaskinen for å lagre store mengder data til f.eks. maskinlæring. SSD-en er mer robust enn HDD og SD-kort, og med å ha operativsystemet kjørende på denne ville man fått en raskere oppstart.

Dessverre viste det seg at markedsføringsavdelingen var litt for raske med å gå ut med denne informasjonen. Per 22.05.2019 støtter ikke kortet I^2C på Windows 10 IoT Core. Vi har lagt ut en post på UP Community som UP har sagt de vil kommentere på når de finner en løsning. Dette medfører at bilen blir levert med RPi som SBC.

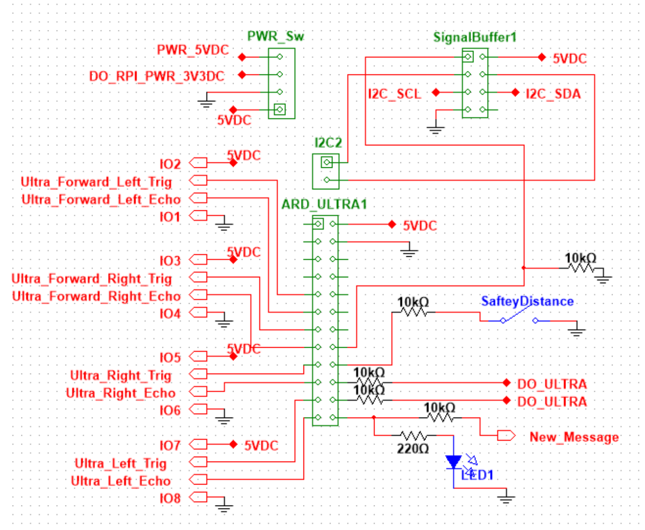
D.2 Sensorer

D.2.1 *Ultralyd [2]*

På den opprinnelige bilen var det kun lidar som eneste avstandsmåler, den fungerer dårlig på avstander under én meter. Derfor ble det testet og montert fire ultralydsensorer av typen HC-SR04 på bilen. Disse er laget for enkel bruk med mikrokontrollere og selve sensoren tar for seg fysikken med å sende og lytte på pulser.

Spesifikasjoner:

- 4 pinner, (VCC, Trig, Echo, GND)
- Bruker 5 volt DC, 15 mA arbeidstrøm
- Målevinkel på 30°
- Fra 2 til 400 cm
- Oppløsning 0,3 cm



Figur 28: Kretstegning av mikrokontroller til ultralydsensorer

Sensoren må aktiveres med en puls på minimum 10 µs på Trig-pinnen. Da sender sensoren ut 8 pulser på 40 kHz og venter på refleksjonen. Når den detekterer pulser som kommer tilbake blir Echo-pinnen satt høy.

```
digitalWrite(trig_pin, LOW);
delayMicroseconds( us: 2);
digitalWrite(trig_pin, HIGH);
delayMicroseconds( us: 10);
digitalWrite(trig_pin, LOW);
const long duration = pulseIn(echo_pin, HIGH,  timeout: 24000);
const int distance = duration * 0.034 / 2;
```

```
distance_forward_left = ultrasonic(pin_trig_forward_left, pin_echo_forward_left);
check_distance(distance_forward_left);
delay(pause);
```

Figur 29: Kode fra mikrokontrolleren til ultralydsensorene

Det har blitt lagt inn en sikkerhetsfunksjon hvor mikrokontrolleren som styrer ultralydsensorene sender et signal som er 0-3 på binærform til motorkontrolleren. Hvilken binær verdi den sender avhenger av de målte avstandene. Etter at det er foretatt en avstandsmåling blir avstanden sjekket opp mot fire hardkodede verdier som knytter seg til de fire binære verdiene som er nevnt over. Dersom avstanden er mindre enn 15 cm blir {00} sendt ut, {01} blir sendt ut dersom avstanden er 15-20 cm, {10} blir sendt ut dersom avstanden er 20-25 cm, og {11} blir sendt ut dersom avstanden er større enn 25 cm.

En bryter er lagt inn for å endre avstanden på modusene. Blir bryteren slått på, reduseres avstanden med 5 cm på alle modusene. Dette gjør at {00} sendes ut dersom avstanden er mindre enn 10 cm.

Det er anbefalt å vente rundt 60 ms mellom hver måling for hver sensor. Dette for å sørge for at pulsene som blir sendt ut rekker å komme tilbake før nye blir sendt ut. Som følge av dette er det mulig at studentenes kontrollogikk spør etter avstand før nye målinger er tilgjengelige. Det er lagt inn en interrupt-pinne på RPi-en som gjør at den bare vil spørre om nye avstandsdata fra mikrokontrolleren når det faktisk er nye avstandsmålinger tilgjengelige. Dette reduserer mengden datatrafikk på I^{2}C-bussen betydelig.

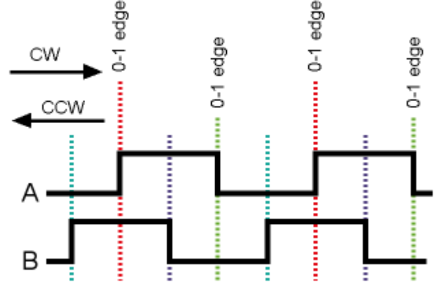
D.2.2 *Encoder [17]*

Encoderne som er brukt er innebygde i motorene, og er inkrementelle encodere. Disse fungerer ved at en skive med flere «aktive» områder roterer, og to utgangspinner som enten har verdien logisk 1 eller 0. Kanalene A og B sender ut firkantpulser, og er logisk 1 dersom pinnen som hører til den aktuelle kanalen er i kontakt med en «aktiv» del av encoderen. Siden kanal A og B er montert med en faseforskjell på 90° kan man detektere hvilken retning skiven i encoderen roterer.

Hver gang encoderen detekterer bevegelse sender den ut et «tick». Ved å måle hvor mange «ticks» encoderen sender ut per centimeter hjulet som er festet til motoren roterer, kan man enkelt regne seg frem til hvor langt hjulet har rotert.

```
void do_encoder_a()
{
    past_b ? encoder0Pos-- : encoder0Pos++;
}

void do_encoder_b()
{
    past_b =! past_b;
}
```



Figur 30: Pulser for kanal A og B, hentet fra [17]

Encoderne er koblet til hver sin mikrokontroller der A- og B-signalet er koblet på hver sin interrupt-pinne, som aktiver hver sin metode. Metoden do_encoder_a() blir aktivert hver gang det kommer en positiv flanke, mens do_encoder_b() blir aktivert hver gang det skjer en endring. past_b er en boolsk verdi. Så, er past_b lav og _a() blir aktivert, går hjulet ene veien. Motsatt om past_b er høy og _a() inntreffer.

encoder0Pos teller opp eller ned etter hvilken vei hjulet går. Hver gang SBC spør etter data, blir antall tiks regnet om til cm og sendt i retur, samt tiden fra sist det ble spurt etter data. Deretter blir teller for tiks og tid satt til null.

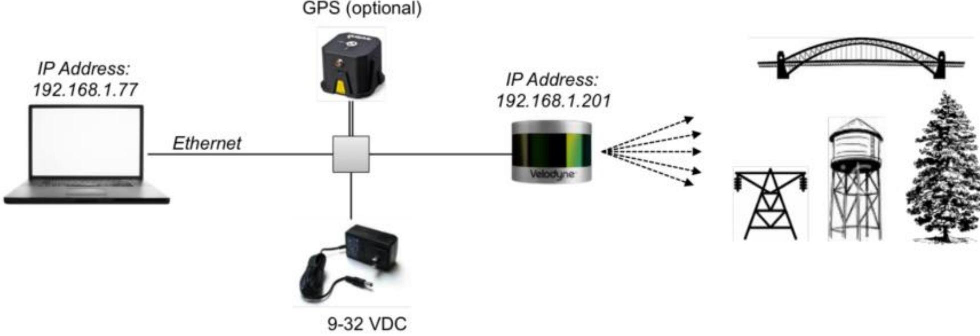
D.2.3 *Kamera*

Kameraet som er koblet til bilen er et Microsoft LifeCam HD-3000. Dette ble valgt fordi det var det kameraet som var lettest å skaffe av de som står listet opp på Microsoft sin oversikt over maskinvare som er kompatibelt med Windows 10 IoT Core [18].

D.2.4 *Lidar [19]*

Lidar er en sensorteknologi med roterende lasere som brukes for å måle avstand og vinkel til punkter i rommet rundt den. Horisontalvinklene den måler på er litt forskjøvet for hver gang, dermed vil man få høyere oppløsning jo flere rotasjoner man måler over.

På toppen av bilen er det montert en Velodyne VLP-16 lidar (også kjent som Velodyne LiDAR Puck). Dette er en sensor med 16 vertikale lasere som spinner rundt i en hastighet på 300RPM (konfigurerbar 300-1200RPM via web-interface på 192.168.1.201) og utfører tusenvis av målinger per sekund [19, p. 2]. Det utføres 24 målinger, hvor det måles avstand og vinkel til punktene laserne treffer, før en pakke med data fra målingene genereres og sendes ut som en UDP-broadcast via nettverket, prosessen repeteres så på nytt. Tiden det tar mellom hver måling er den samme uansett hvilken hastighet lidaren kjører på. Man får derfor ikke inn flere målinger ved å velge en høyere rotasjonshastighet på lidaren, men bare større vinkel mellom hver måling. Med andre ord vil man med 600RPM fullføre en full rotasjon dobbelt så raskt som ved 300RPM, men samtidig halveres antall målinger på denne rotasjonen. Dvs. det blir dobbelt så stort vinkelgap mellom hver måling.



Figur 31: Figur over lidar, hentet fra manual [19]

Manualen for lidaren er lettlest og går mye mer inn i detalj på hvordan lidaren fungerer og kan brukes.

D.3 Fremdrift

Bilen har fire hjul med «ballongdekk» som er festet på hver sin DC-motor med gir. DC-motorene får strøm fra en motordriver som blir styrt med PWM signal fra en motorkontroller.

D.3.1 *DC motor*

Ved 12 volt er den oppgitt med å ha 115 RPM etter giret.

Motorene er parallellkoblet på hver side av bilen. Et problem med dette er at strømmen alltid vil gå den letteste veien som skjer når ett av de to hjulene kommer opp fra bakken og andre hjulet får motstand. Dette kan skje om bilen kjører skrått over en dørkarm og et hjul på hver side kommer opp fra bakken.

Motorene fulgte med den gamle bilen og er av ukjent alder. Giret på noen av motorene har låst seg et par ganger, men har løsnet ved å vri frem og tilbake på hjulet. Bør muligens byttes til noen nyere dersom det blir et problem.

D.3.2 *Motorkontroller*

Vi har programmert en mikrokontroller, til å ta imot kommandoer fra RPi over I^2C og gi ut to PWM signal til motordriveren. Fra RPi kommer det to verdier mellom -100 og 100, som er i prosent hvilken effekt som skal settes på motorene på hver side av bilen. Motordriveren er stilt inn til å motta PWM signal mellom 1000 og 2000 ms der 1000 er full effekt ene veien og 2000 er full effekt andre veien. Ved 1500 er det null effekt ut.

DC-motoren trenger litt strøm før den får nok kraft til å klare å drive bilen fremover. Derfor har vi med en helt enkel test funnet ut når bilen begynner å bevege seg. Siden motorene er laget for 12 volt har vi valgt å begrense maks og min, etter som motordriveren får strøm rett fra 18 volts batteriet. Maks og min fant vi ved å måle spenning over motor mens PWM pulsene økte og stoppet da vi nådde ca. 12 volt. Dette gjør at 0-100 fra RPI blir til 1600-1840 ut fra mikrokontrolleren.

```
const int pwm_start_reverse = 1330;
const int pwm_full_reverse = 1130; // 12v, min (1000)
const int pwm_start_forward = 1600;
const int pwm_full_forward = 1840;  // 12v, max (2000)
const int pwm_standstill = 1500;
const int ramp_up = 20;
```

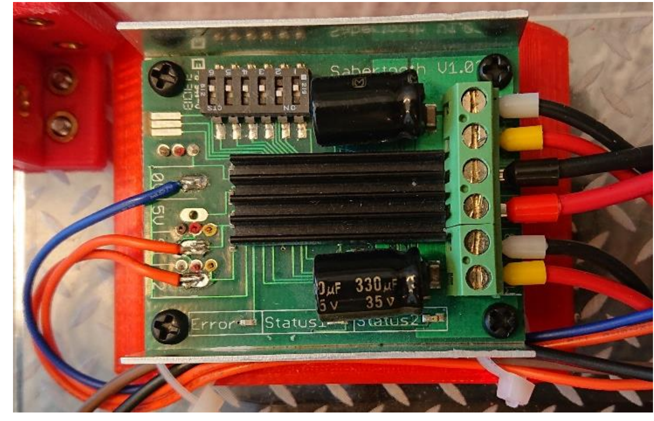
Mikrokontrolleren styrer også reléet som gir strøm til motordriveren. Dette er tenkt som en ekstra sikkerhet, slik at den skal være mulig å slå av om det skjer noe uventet. En annen grunn til å styre strømmen til motordriveren er at den må få inn PWM signal på 1500 for at motorene skal stå i ro. Hvis motordriveren er på før mikrokontrolleren, kan den få ugyldige signal som resulterer i at bilen begynner å kjøre.

```
int mode_check(int data)
{
    if (!digitalRead(pin_mode_1) && !digitalRead(pin_mode_2) )
    {
        mode = 0;
        return 0;
    }
    else if (!digitalRead(pin_mode_1) && digitalRead(pin_mode_2))
    {
        mode = 1;
        data = data / 3;
        return data;
    }
    else if (digitalRead(pin_mode_1) && !digitalRead(pin_mode_2))
    {
        mode = 2;
        data = data / 2;
        return data;
    }
    else if (digitalRead(pin_mode_1) && digitalRead(pin_mode_2))
    {
        mode = 3;
        return data;
    }
    return 0;
}
```

Mikrokontrolleren får og to digitale signal fra ultralydsensoren som velger hvilket «modus» den skal kjøre i, 0 til 3, der 3 er «full effekt» og 0 er «ingen effekt». Dette kan overstyres med en bryter som setter den i en fast modus som reduserer effekten som skal settes på motorene.

D.3.3 *Motordriver*

Til den gamle bilen ble det kjøpt inn en «Sabertooth 2X12 RC» [20] til å drive motorene. Den har to kanaler som kan gi opptil 12 ampere hver. Den er laget veldig sikker bruk og skal hindre overoppheting og ukontrollert strømtrekk.



Motordriveren har flere forskjellige innstillinger som kan settes med seks mikrobrytere som står montert på kortet. Til oppsettet som er på bilen, er innstillingene satt til at motordriveren:

- tar imot to PWM signal, ett til hver kanal/side på bilen
- «Exponetial Mode» gjør responsen mindre sensitiv i senter av spekteret
- «Lithium Mode» er på, den slår av driveren hvis spenningen på hver celle kommer under 3 volt.
- «Acceleration Ramping» gjør at motorkontrolleren hindrer raske endringer som lager store strømtrekk.

Se i databladet [20] for flere innstillinger.

D.4 Strømforsyning

På det gamle bilen var det ikke noen form for strømstyring til de ulike komponentene. Der ble det satt strøm på alle komponentene med én gang. Dette gjorde at man ikke hadde noen form for kontroll hvis det skulle skje noe uventet. Et annet problem var at lidaren trekker en del strøm, noe som gjorde at batteriet ble tappet unødvendig fort. Det var også en unødvendig 12 volts DC regulator som ble brukt til lidaren. Lidaren har egen regulator som takler spenning opp til 36 volt. [19]

D.4.1 *Batteri [21]*

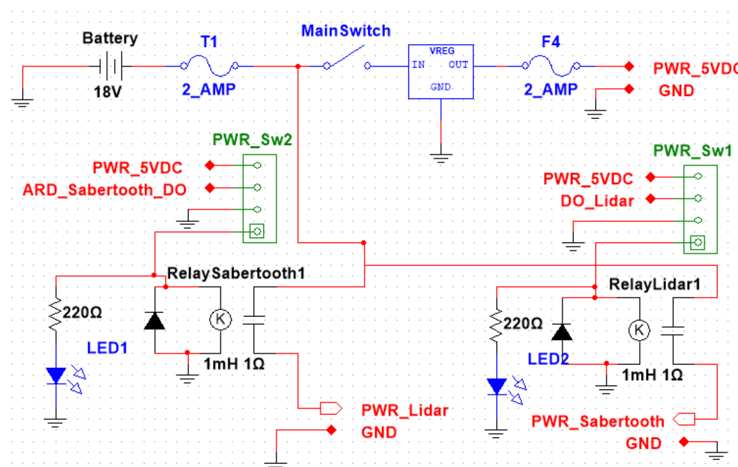
To Milwaukee M18™ 5.0 AH batterier med lader ble kjøpt inn til den gamle bilen. Disse batteriene er av type Li-ion, 18 volt DC fulladet, og har en kapasitet på 5.0 amperetimer.

Disse ble tatt med videre da de er «hyllevare», solid bygget og har bra kapasitet. Batteriet har innebygget display som viser hvor mye kapasitet som er igjen på batteriet. Det er også mulig å hente ut denne verdien på to kontaktpunkt på batteriet, som eventuelt andre enheter kan overvåke. Dette ville vi ha sett mer på om vi hadde hatt mer tid.

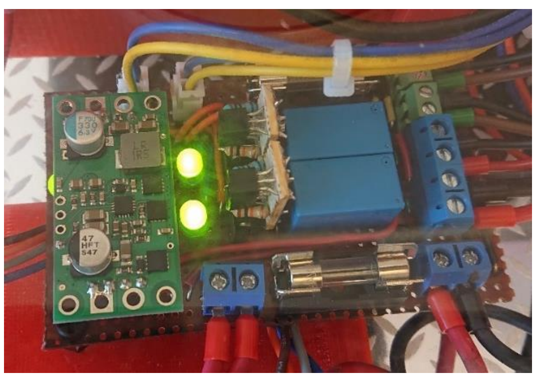
Sensorbilen bruker rundt 0,34 amper til å drive all elektronikk som RPi, vifte, Arduino og skjerm. Med maks kraft på motorene, trekker elektronikken og motordriveren 1,5 ampere. Dersom man i tillegg slår på lidaren blir totalt strømtrekk rundt 2 ampere. Strømtrekket vil variere en del etter hvor mye motstand motorene får og hvor mye spenning som er igjen på batteriet.

D.4.2 *Strømstyringskrets*

Sensorbilen er bygget opp med felles jord (GND) over alt, her er batteriets minuspol koblet til. Batteriets plusspol går til en bryter som står ved batteriet. Blir bryteren slått på, får 5 VDC regulatoren strøm og RPi slår seg på. SBC styrer det ene reléet som gir strøm til lidaren. Motorkontrolleren styrer reléet som gir strøm til motordriveren.



Figur 32: Kretstegning av strømforsyning



Det er plassert en treig 3,15 ampere sikring mellom batteri og resten av elektronikken. Ut fra regulatoren er det montert en rask 2 ampere sikring for å skåne den mest følsomme elektronikken. Lidaren har en egen 3 ampere sikring på sitt kort.

Det er montert på lysdioder for at man kan visuelt se om det er strøm på systemet og om reléene er slått på.

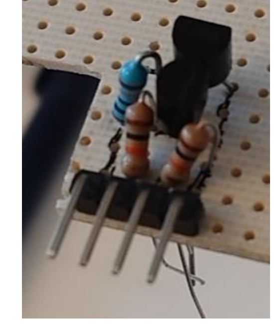
D.4.3 *5 Volts DC regulator*

«Pololu 5V, 9A Step-Down Voltage Regulator D24V90F5» [22] ble kjøpt inn for å ha nok kraft til å drive Up-Board som krever opp mot 6 ampere strøm. Egenskaper:

- 5-38 volt inn
- Opptil 9 ampere ut (avhengig av spenning inn, ca. 6 ampere ved 5V inn)
- Sikkerhet for temperatur, tilbake-spenning, overstrømtrekk.
- Mellom 80 og 95% effektiv.
- Egen «Power good» signal fra regulatoren

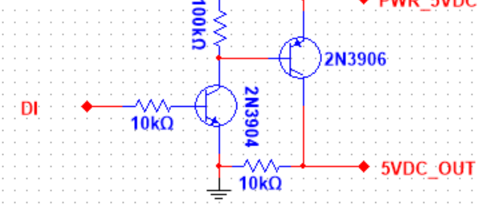
Den har og en «enable»-inngang som kan settes lav for å slå regulatoren av. Når sluttproduktet leveres går all strøm fra batteriet gjennom en bryter som blir brukt til å slå av og på hele systemet. En bedre løsning hadde vært og tatt nytte av «enable»-inngangen på regulatoren og fått til en bedre og sikrere strømstyring fra hovedprogrammet.

D.4.4 *Strømstyring til Arduino og Raspberry Pi*



Både Arduino Pro Mini og Raspberry Pi sine digitale utganger er laget for signalstyring og ikke strømforsyning, og kan ofte ikke gi ut mer enn 20 til 40 mA. Derfor ble det laget en strømstyringskrets som får inn et signal, enten 3,3 volt fra RPi eller 5 volt fra Arduino, som blir brukt til å gi strøm til mikrokontrollerne og reléene.

Kretsen virker på den måten at med en digital høy (3,3v/5v) inn på basen på NPN transistoren vil gjøre at den begynner å lede mellom kollektor og emitter. Dette gjør at basen på PNP transistoren blir trukket ned mot jord og den begynner å lede.



Med dette får vi at digital høy på inngangen gir 5 volt ut. En mer «normal» løsning er å kun bruke en NPN transistor og bryte strømmen på negativ side av komponentene som trekker strøm, men for å ha en enkel forståelse av når komponenter blir slått av og på er det muligens lette å tenke at HØY er 5 volt og LAV er 0 volt/GND.

Kretsen er og laget på et eget lite kretskort som gjør at den kan lett byttes eller forbedres etter behov.

D.5 Karosseri

Krav til oppgaven var å lage bilen mer solid og brukervennlig. Til dette ville vi bygge bilen med gjennomsiktige plater for at det skal være mulig å se på oppbyggingen av elektronikken inni uten å måtte demontere noe på bilen. Lidaren trenger fri sikt rundt seg og den laseren som peker mest nedover har en vinkel på -15 grader. Dette var utgangspunktet for vinkelen til platen som skjermen er festet på.

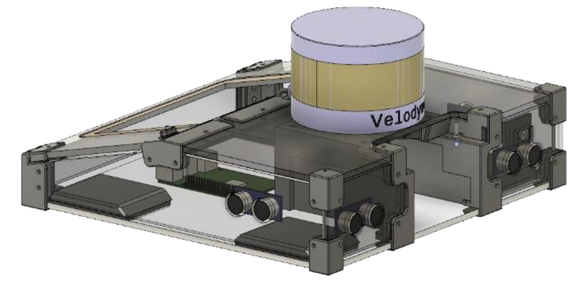
D.5.1 *Plater v.0.2*

På **versjon 0.2** ble det brukt Biltemas plastglass til å bygge flatene på bilen. Dette er plater som er enkle å jobbe med og man kan raskt få bygget opp en prototype.

Til å utarbeide platene ble det brukt fres på kanter og til å senke skjermen ned i platen, stikksag til å skjære med og bor. Varmluft ble brukt til å varme platene for så å bøye dem over et kosteskaft.

Denne måten å bygge opp bilen på funket fint for å lage en prototype raskt. Produktet ble laget etter «papirtegninger» og kreativitet. Et problem med håndverktøy er at utskjæringer og detaljer ikke blir så nøyaktige som man skulle ønsket i et sluttprodukt.

D.5.2 *Autodesk Fusion 360*



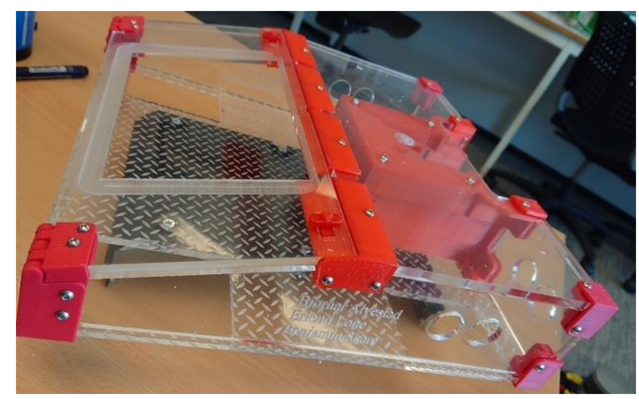
Til neste versjon ble Fusion 360, et program fra Autodesk hvor man kan tegne, modellere og simulere det meste, brukt. Vi har tegnet hele karosseriet til bilen ved bruk av 3D-modellering.

Fusion 360 har laget gode videoer som forklarer hvordan alt skal tegnes, og for noen av komponentene finnes det ferdige tegninger på produsentenes nettsider.

Hver del er tegnet som en komponent og satt sammen for å se hva som ikke passer. Brakettene som er fremme i hjørnene er samme komponent, men komponenten har blitt kopiert og speilet for å passe de ulike hjørnene.

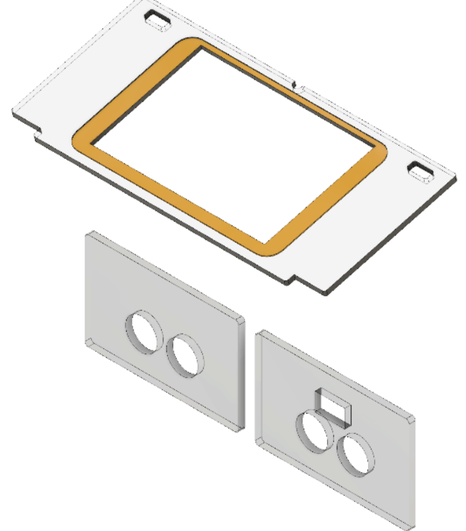
Tegninger og bilder ligger i ZIP-fil til prosjektet og i repositoriet til prosjektet på GitHub.

D.5.3 *Plater v.1.0*



På **versjon 1.0** ble det brukt Plexiglass kjøpt av en lokal glassmester. Plexiglass er hardere og mer ripebestandig enn plastglasset fra Biltema. Plexiglass kan også brukes i en laserkutter uten at platen smelter.

Marineholmen Makerspace har en laserkutter på 60 watt som vi valgte og brukte til å få produsert de nye platene. Etter man er med på kurs for å få opplæring i laserkutteren kan man bruke den fritt om man blir medlem.



Fra tegningene i Fusion kan man hente ut en skisse av delene som kan lastes rett inn i programmet til laserkutteren. Laserkutteren sender en laserstråle gjennom en rekke speil og linser ut til en arm som kan bevege seg langs to akser. Programmet til laserkutteren lager forskjellige farger på det som laseren skal jobbe med. I hver farge kan man sette innstillinger som antall pulser, styrke og hastigheten på hodet. Dette er innstillinger man må «prøve seg fram» for å få ønsket resultat. Til plexiglasset tilpasset vi tre forskjellige farger, én til gravering av tekst, én til nedsenkingen av skjermen og én til å kutte helt igjennom platen.

Målene til skjermen ble hentet fra dens datablad, hvor det var spesifisert lengde og bredde, samt radiusen til hjørnene.

Et problem med laserkutteren er at støvet som kommer fra der laseren brenner, legger seg på platen ved siden av og blir delvis brent fast i platen. Her må man nok sette seg mer inn i ulike metoder man kan bruke laserkutteren på for å unngå dette. Et annet problem er at når laseren beveger seg sakte for å skjære helt igjennom, varmer den opp platen rundt strålen og ødelegger overflaten på platen.

D.5.4 *Braketter*

Til v.0.2 ble det brukt skruer og muttere og vanlig hengsler av stål. Til v.1.0 derimot ble det tatt i bruk 3D-printer, noe som gjorde det mulig å designe alt slik vi ønsket.

3D-printeren som ble brukt er en Makerbot Replicator Z18 som står på HVL Kronstad. Den bruker en plastikktråd som blir presset gjennom en varm dyse som beveger seg i x- og y-retning. En plate under dysen beveger seg i z-retning nedover etter hvert som hvert lag med plastikk blir lagt oppå.

På nett finnes det mange bibliotek med tegninger andre folk har lastet opp. Her kan man for eksempel finne ferdige holdere for batteriet vi bruker. Dette var vanskelig å utnytte seg av da det ble komplisert å tilpasse tegningene til sånn som vi ville ha de. Etter litt prøving og feiling ble alle mål tatt og alle brakettene ble tegnet fra bunn av.

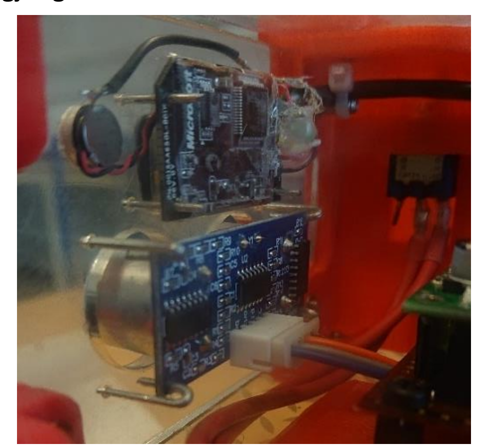
Hengslene som skjermen henger i er printet ut «i en del» der begge deler henger sammen, men kan beveges. Dette er et godt eksempel på hva 3D-printer kan brukes til.

D.5.5 *Festemidler*

Bunnplaten er festet i chassis med seks nedsenkede M4 skruer med låsemutter.

Kortene er plassert på avstandstykker av plast med skruer av plast. Det ble brukt plast istedenfor metall for å være på den sikre siden at det ikke blir en kortslutning.

Til brakettene ble det kjøpt inn ferdige «gjenger» som kan smeltes inn i 3D-printede komponenter. Diameteren i hullet som printes er litt mindre enn diameteren til gjengen som smeltes inn med en redusert temperatur på loddebolten. Platene blir da klemt fast i braketten med M3 skruer.



Til oppheng av kamera og ultralyder har det blitt brukt tynne lange spikre som har blitt bøyd og smeltet inn i et forboret hull i plexiglassplaten.

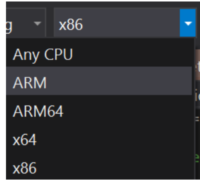
Superlim er brukt til å lime fast avstandsstykker til kretskort og andre komponenter på bilen. Strips er brukt til å samle kabler og føre de fint inne i bilen samt for strekkavlastning på kabler og ledninger.

Appendiks E Software – Hovedprogram

Det er inne i hovedprogrammet studentene skal skrive sin kontrollogikk. Studentene kan enkelt skrive kontrollogikken sin inn i «StudentLogic»-prosjektet, og trenger ikke å forstå/forholde seg til resten av programmet. Dette vedlegget er gir en forklaring på hvordan programmet er bygget opp/fungerer, og er laget for de som er interessert i «hva som ligger bak».

Hovedprogrammet er skrevet i programmeringsspråket C# og XAML for UI delen, ved å bruke Visual Studio 2017. Programkoden er organisert i en «solution» som består av flere «prosjekt». Programmet er laget slik at det er enkelt å utvide med ny funksjonalitet.

Man kan enkelt velge om man skal kjøre programmet mot simulatoren på sin egen PC, eller på den fysiske bilen, ved å veksle mellom x86 og ARM som prosessortype (etter å ha skrevet inn bilens IP adresse den første gangen man kjører på fysisk bil). Programmet er laget slik at det vil automatisk tilpasse seg hvor det kjører, uten at studentene trenger å forholde seg til dette.



I kapitelene under tar vi utgangspunkt i at leseren kjenner til UWP, MVVM, interface, DI[10], IoC[11] og lignende. Hvis noe av dette er ukjent så kan man lese Appendiks G for en kort innføring.

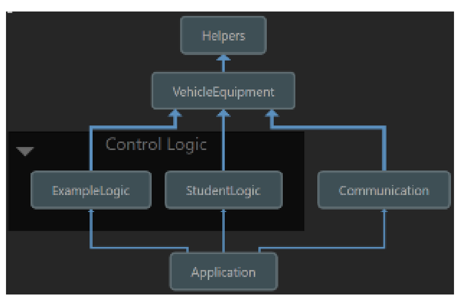
[10] Dependency Injection – avhengighetene mates inn via konstruktør (se G.4.2 for mer info).
[11] Inversion of Control – designprinsipp for å oppnå løs kobling mellom klasser (se G.4.2 for mer info).

E.1 Prosjektoversikt

I dette kapittelet blir det gitt en oversikt over prosjektstrukturen, mens i kapittel E.2 står det mer detaljert om hvordan selve klassene knyttes sammen.

Hele hovedprogrammet består av flere «prosjekt» som er samlet i en «solution» med navnet «SensorVehicle-main».

Solutionen består av åtte prosjekt hvor, «Application» er det eneste prosjektet som kan kjøres direkte. Det er fem klassebibliotek som inneholder kode som «Application» benytter seg av, men som ikke kan kjøres på egenhånd. Det er også to test-prosjekt som er brukt for å automatisk verifisere at koden fungerer som forventet.



Diagrammet til venstre viser en oversikt over hvordan de forskjellige prosjektene refererer til hverandre.

Her ser vi at «Helpers» biblioteket er helt uavhengig, mens alle andre prosjekt referer til «Helpers»-biblioteket. «VehicleEquipment»-biblioteket benyttes også av alle (med unntak av «Helpers»).

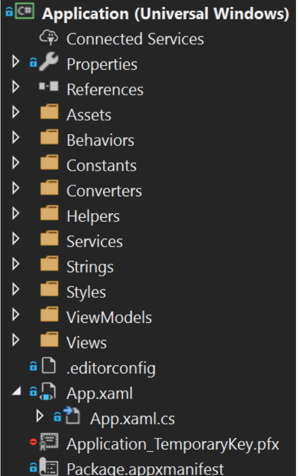
Bibliotekene for «ExampleLogic», «StudentLogic» og «Communication» er det bare «Application»-prosjektet som refererer til.

Alle klassebibliotek vi har laget, med unntak av «Communication», er av typen .NET Standard 2.0. .NET Standard 2.0 er valgt der det er mulig ettersom denne typen bibliotek kan gjenbrukes i alle typer prosjekt (enten det er en UWP-app eller .NET Framework-app). «Communication»-klassebiblioteket benytter seg av kode som normalt bare er tilgjengelig for UWP prosjekt (e.g. I^2C og GPIO), og måtte derfor være et UWP bibliotek.

E.1.1 *UWP prosjektet – «Application»*

Det kjørbare prosjektet «**Application**» er av typen UWP[12]. Den er opprettet ved hjelp av «Windows Template Studio», hvor vi benytter MVVM[13] (Prism/Template10) som «design pattern».

For å holde prosjektet oversiktlig er nesten all kode delt inn i mapper.



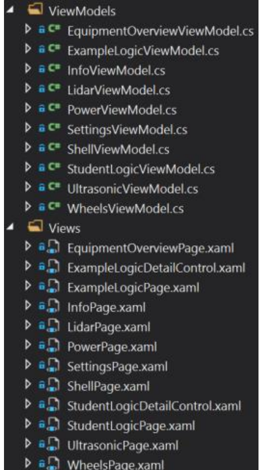
App.xaml.cs - Knutepunktet for hele programmet:

Når et UWP prosjekt starter er koden i App.xaml.cs filen den første som kjøres (man kan tenke på denne som «Main»-metoden i et konsollprogram). Det er i denne filen vi har skrevet koden som knytter hele programmet sammen (mer om dette i E.3).

[12] Universal Windows Platform – en prosjekttype for en moderne grafisk applikasjon (se G.1 for mer info).
[13] Model-View-ViewModel (se G.3 for mer info).

Vi har benyttet IoC/ DI for å oppnå løs kobling mellom klassene. Det er dette, sammen med utstrakt bruk av interfacer og polymorfisme (se G.4.1), som gjør programmet så fleksibelt når det kommer til fremtidige utvidelser, og som lar programmet enkelt og elegant tilpasse seg etter hvor det kjører.

«Views» og «ViewModels»:

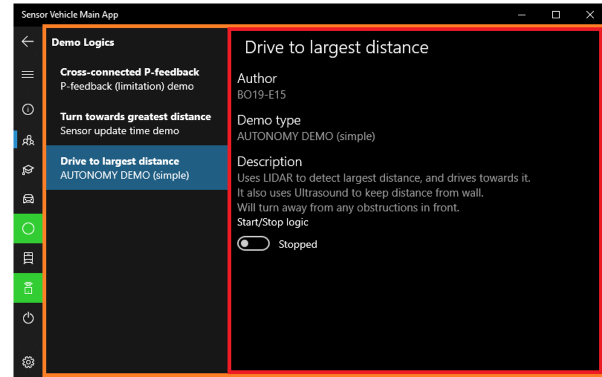


Dette prosjektet inneholder blant annet «Views» og «ViewModels» for MVVM-appen vår. Selve modellene befinner seg i klassebibliotekene (hovedsakelig i «VehicleEquipment»-biblioteket).

Appen har ni skjermsider. For hvert skjermbilde eksisterer det minst én «View», som håndterer hvordan dataen skal vises, og en tilhørende «ViewModel» som fungerer som bindeledd mellom «View» og «Model».

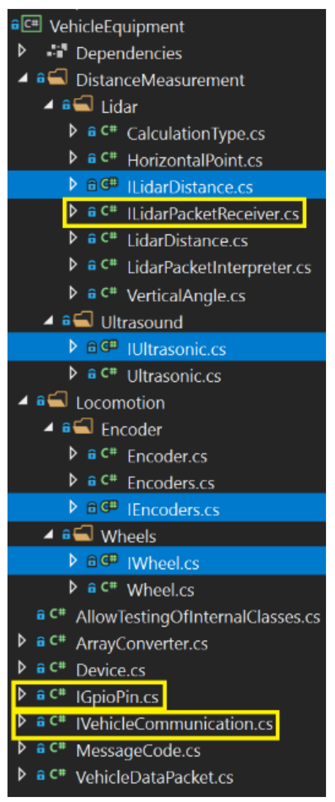
Både «StudentLogic» og «ExampleLogic» har to «Views» hver, ettersom disse har to skjermområder. En liste til venstre hvor man kan velge en spesifikk kontrollogikk, og en rute til høyre hvor data for den valgte kontrollogikken vises.

På bildet til høyre ser vi «ExampleLogicPage.xaml» inne i den oransje ruten. Mens «ExampleLogicDetailControl.xaml» bestemmer hvordan informasjonen i den røde ruten skal se ut, som endres basert på hvilke element er valgt i listen.



Et spesielt «View/ViewModel» par er «Shell». Dette er den overordnede strukturen på det man ser på skjermen, som de andre «Viewene» lastes inn i. Navigasjonsikonene man ser i kolonnen helt til venstre på skjermbildene er definert i «ShellPage» og er synlige uansett hvilke side man ser på. De forskjellige «Viewene» lastes i ruten som er markert med en oransje ramme på bildet over.

E.1.2 *Klassebiblioteket - «VehicleEquipment»*



«VehicleEquipment» prosjektet inneholder koden for bilens utstyr, som sensorer og motorer. Studentene som skal lage kontroll logikk for bilen vil hovedsakelig forholde seg til de fire interfacene som er merket med blått på bildet til venstre. Det samme gjelder for dataen som vises på skjermbildene i programmet. Resten av klassene i dette prosjektet er å regne som implementasjonsdetaljer.

Det er en del flere klasser for Lidaren enn det andre utstyret. Dette er fordi lidaren er koblet direkte til SBC-ens ethernetport, og alt databehandlingen foregår i hovedprogrammet.

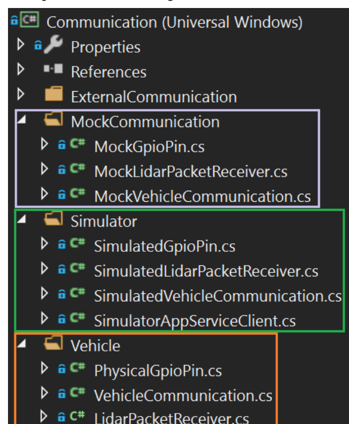
«Wheel», «Ultrasonic» og «Encoders» har en enklere implementasjon i hovedprogrammet, ettersom det er en mikrokontroller for hver komponent som tolker og omformer rådataen fra sensorene slik at hovedprogrammet bare kan spørre etter den tolkede dataen via f.eks. I^2C-bussen.

På bildet til venstre er det også tre interfacer som det er satt gul ramme rundt. Det er disse som benyttes av klassene i VehicleEquipment, og de mates inn via klassenes konstruktør. Interfacene i seg selv kan ikke instansieres. Klassene som implementerer disse interfacene, og som er de faktiske instansene som mates inn i konstruktørene når programmet kjører befinner seg i «Communication»-prosjektet.

E.1.3 *Klassebiblioteket - «Communication»*

«Communication»-prosjektets kode kan deles inn i to hovedområder.

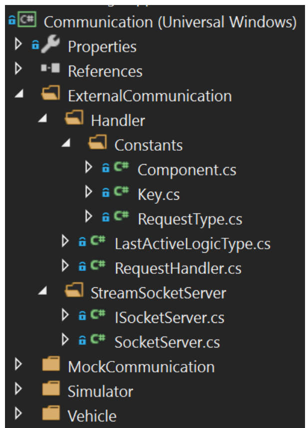
Implementasjon for interfacene (med gul ramme) i «VehicleEquipment»



Vi har skrevet kode i App.xaml.cs filen som sjekker hvor programmet kjører når det først starter opp. Dvs. vi ser om programmet kjører på den fysiske bilen (i.e. Windows 10 IoT Core), eller om det kjører på en vanlig PC. Hvis sistnevnte er tilfellet, så sjekkes det om simulatoren er installert. Vi har da de tre mulige tilstandene «kjører på bil», «kjører på PC med simulator» og «kjører på PC uten simulator».

Inne i App.xaml.cs instansieres én av de de tre mulige implementasjonene for hver av av interfacene fra «VehicleEquipment», basert på hvor koden kjører (i.e. IGpioPin blir enten PhysicalGpioPin, SimulatedGpioPin eller MockGpioPin ved runtime). Disse mates så inn i konstruktørene til de klassene som benytter disse interfacene. For flere detaljer rundt dette og fordeler dette gir, se E.3 .

Socket Server for ekstern kommunikasjon

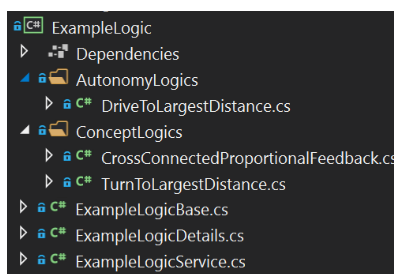


Her finnes kode som gjør det mulig for andre program å koble til bilen via en socket-tilkobling, for så å kommunisere med bilen (eller simulatoren - hvis programmet kjører på en PC) ved bruk av JSON-strenger.

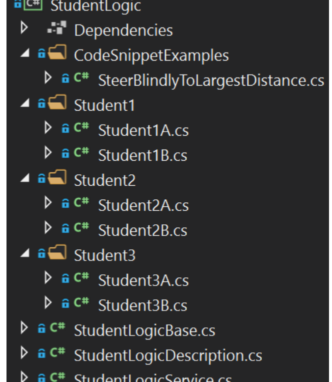
Vi har laget en Android-app som fungerer som en fjernkontroll. Men det er også fullt mulig å gi kommandoene som JSON-formaterte tekststrenger direkte ved å koble til socketserveren med PuTTY, eller man kan lage sitt eget program for dette.

I E.5 finnes mer info om socketserveren, som blant annet hvordan kommandostrengene må være formatert.

E.1.4 *Klassebibliotekene – «ExampleLogic» og «StudentLogic»*



Disse to prosjektene inneholder kontrollogikken som skal styre bilen. De benytter seg av modellene fra «VehicleEquipment» til å gi kommandoer til hjulene basert på data den mottar fra sensorene.



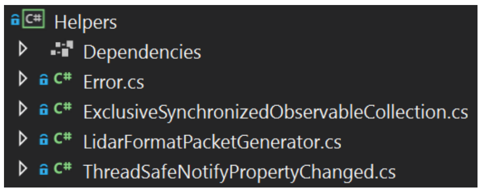
«ExampleLogic» prosjektet til venstre og «StudentLogic» prosjektet til høyre er svært like. Det som skiller dem er i hovedsak hvilke typer kontrollogikk de er ment å holde.

«ExampleLogic»-prosjektet inneholder kontrollogikker som enten er ment å demonstrere hva man kan gjøre med bilen (e.g. «DriveToLargestDistance» demonstrerer autonom kontrollogikk), eller for å illustrere et konsept (e.g. effekten av for lang oppdateringstid på måledata i «TurnToLargestDistance»).

«StudentLogic»-prosjektet inneholder flere filer som er tilrettelagt for å komme enkelt i gang med å lage sin egen kontrollogikk. Den inneholder også noen kontrollogikker som kan benyttes som utgangspunkt eller inspirasjon for sin egen kontrollogikk.

For mer informasjon om kontrollogikk se E.4 .

E.1.5 *.NET Standard biblioteket – «Helpers»*



«**Helpers**» inneholder noen fåtalls klasser som ikke er avhengig av noen andre prosjekt, og som alle prosjekt kan benytte seg av.

E.2 Klasseoversikt

Diagrammene/delkapitlene under gir en oversikt over hvordan klassene i hovedprogrammet henger sammen. For å holde fokuset på den overordnede strukturen på programmet har ikke alle klasser blitt tatt med i diagrammene.

E.2.1 *Symbolforklaring*

Symbolene vi har benyttet i klassediagrammene baserer seg på «UML klassediagram». Vi har ikke fulgt de formelle UML-reglene slavisk, ettersom vårt mål er å gi et overblikk over hvordan programmet er bygget opp, ikke å dokumentere programmets oppførsel til den minste detalj.

Pil-symboler

Symbol	Forklaring
——◆	Vi har benyttet komposisjonssymbolet for å vise at klassen som diamanten peker på **inneholder en (unik) instans av klassen linjen kommer fra**. Klassen linjen kommer fra blir enten instansiert inne i klassen diamanten peker på, eller utenfor og så matet inn via konstruktøren til kun den klassen som diamanten peker på.
——◇	Vi har benyttet aggregeringssymbolet for å vise at klassen som diamanten peker på **inneholder en (delt) referanse til én instans av klassen linjen kommer fra**. Klassen linjen kommer fra blir instansiert utenfor klassen diamanten peker på, og så blir den samme referansen matet inn via konstruktøren til alle klassene som har diamanten pekende på seg.
——▷	Klassen som linjen kommer fra arver fra klassen pilen peker på.
– – –▷	Klassen som linjen kommer fra implementerer interfacet som pilen peker på.
———	Vi har benyttet assosiasjonssymbolet for å vise en generell kobling mellom klasser. F.eks. hvor en klasse benytter seg av metoder i en statisk klasse

I diagrammene nedenfor er det flere overlappende linjer. Disse representerer fortsatt flere individuelle linjer, men er bare tegnet på toppen av hverandre for å holde diagrammene oversiktlige.

Teksttyper:

Teksttype	Betydning
Kursiv	Brukes for å vise at en klasse eller metode er abstrakt.
Understreking	Brukes for å vise at en klasse er statisk («static»).

Farger:
Gul for å fremheve interfacene tydeligere. Andre farger som har blitt benyttet har ikke universell betydning for alle diagrammene, men er bare benyttet for å enklere kunne referere til spesifikke deler av diagrammet.

Glødende kanter:
Typer som er registrert som IoC-containere med Unity i App.xaml.cs er merket slik at de gløder rundt kantene. De som er registrert med interface gløder oransje/gult, mens de som er registrert direkte uten interface gløder grønt. Disse er alle registrert i containeren som «singeltons», dvs. at det benyttes bare en instans av disse i hele programmet.

De vises med en sterk glød der hvor diagrammet viser hvordan disse er bygget opp. Mens der hvor de har blitt benyttet som et startpunkt for resten av diagrammet vises de med en svak glød.

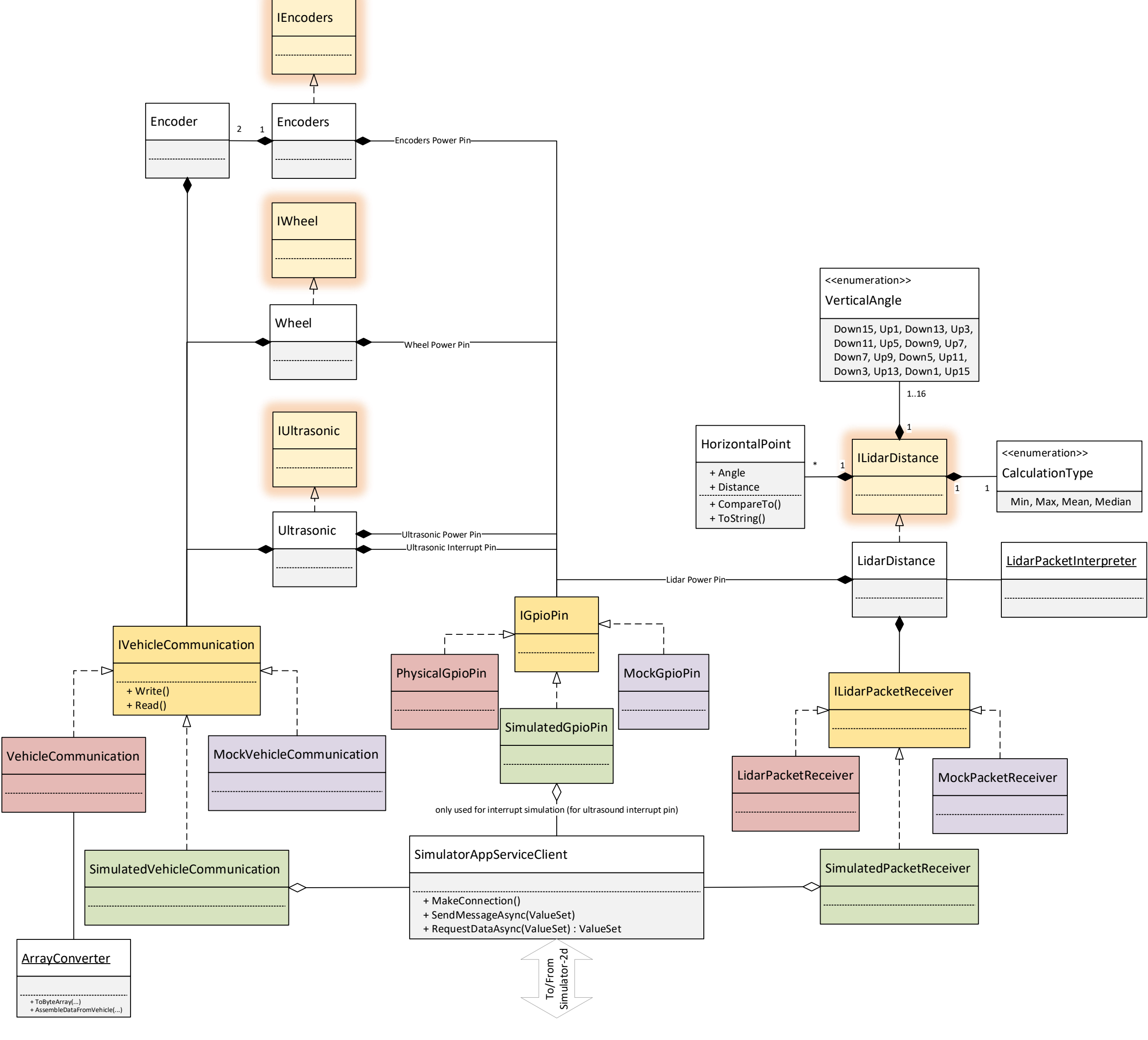
E.2.2 *Klassediagram for Sensorer/Utstyr*

Figur 33 vises en forenklet tegning av hvordan modellene for bilens utstyr er bygget opp. Disse gløder oransje rundt kantene for at de skal skille seg ut i tegningen.

Komponentene har hver sin egen instans av IGpioPin for å kunne slå på/av strømmen. «Ultrasonic» har i tillegg en ekstra IGpioPin-instans som brukes for interrupt-signal når ny måledata er tilgjengelig på mikrokontrolleren.

Alt utstyret som drives av mikrokontrollere (de på vestre side av tegningen) har alle hver sin IVehicleCommunication-instanse. Data fra lidaren behandles direkte i hovedprogrammet, og mottas via TCP. Derfor benytter «LidarDistance» i stedet en «ILidarPacketReceiver.

Interfacene kan egentlig ikke instansieres. Men man kan la interfacene holde en referanse til en hvilken som helst klasse som implementerer disse interfacene. På denne måten oppnår vi stor fleksibilitet i programmet ved å velge hvilken implementasjon som skal brukes når programmet kjører. De røde klassene benyttes når koden kjører på fysisk bil, de grønne hvis koden kjører på PC (med simulator), og lilla hvis simulator ikke er tilgjengelig.



Figur 33: Klassediagram for "Sensorer/Utstyr"

Figur 33 viser også «SimulatorAppServiceClient» som alle de grønne implementasjonene har en delt referanse til. Denne benyttes for kommunikasjon med simulatoren, som kjører som en helt separat app.

Det er flere klasser som ikke har blitt tatt med i diagrammet for å holde det mer oversiktlig. Noen av disse er:

«Error»-klassen:
Denne benyttes for å lagre feilmeldinger/feiltilstand i forbindelse med exception-handling. Modellene «Encoders», «Wheels», «Ultrasonic» og «LidarDistance» har alle hver sin instans av «Error»-klassen.

«VehicleDataPacket»-structen:
Denne benyttes for å oppbevare data som skal sendes til / er mottatt fra mikrokontrollerne på et format som er lettere for mennesker å jobbe med, men som fortsatt er standardisert nok til at det enkelt kan konverteres til/fra en tabell med bytes.
«IVehicleCommunication», og dermed alle de tre klassene som implementerer dette interfacet, benytter seg av denne structen som returtype i «Read()» metoden. Den benyttes også av «ArrayConverter» i «AssembleDataFromVehicle()» metoden.

«MessageCode»-enum:
Denne er en del av meldingen som sendes til/fra microkontrollerene. Den er på nåværende tidspunkt ikke brukt til noe, og innehar dermed alltid verdien «NoMessage» som har tallverdi 0. Da formatet på dataen som skulle sendes mellom mikrokontrollerne og hovedprogrammet ble bestemt, ble det bestemt at det skulle være en byte tilgjengelig for en meldingskode (for fleksibilitet med tanke på fremtidige endringer/utvidelser av programmet).

«Device»-enum:
Denne inneholder adressene til I^{2}C-enhetene som brukes for kommunikasjon.
Den benyttes i «ArrayConverter», «VehicleDataPacket» og i «IVehicleCommunication» (og dermed alle de tre klassene som implementerer dette interfacet).

Den abstrakte klassen «ThreadSafeNotifyPropertyChanged»:
Denne inneholder metoder som gjør bruk av INotifyPropertyChanged mer ryddig (inspirert av Prism sin «BindableBase»). Det som har blitt lagt til i denne klassen, som ikke er med i BindableBase er:

«RaiseNotificationForSelective» boolean, samt tilhørende metoder som bare fyrer av «PropertyChanged» eventet dersom den er satt til *true*. Dette gjør at sluttbruker kan velge om man ønsker å se dataen oppdatere seg på skjermen, eller om man ønsker å slå av den funksjonaliteten for å få koden til å kjøre raskere / mer effektivt.

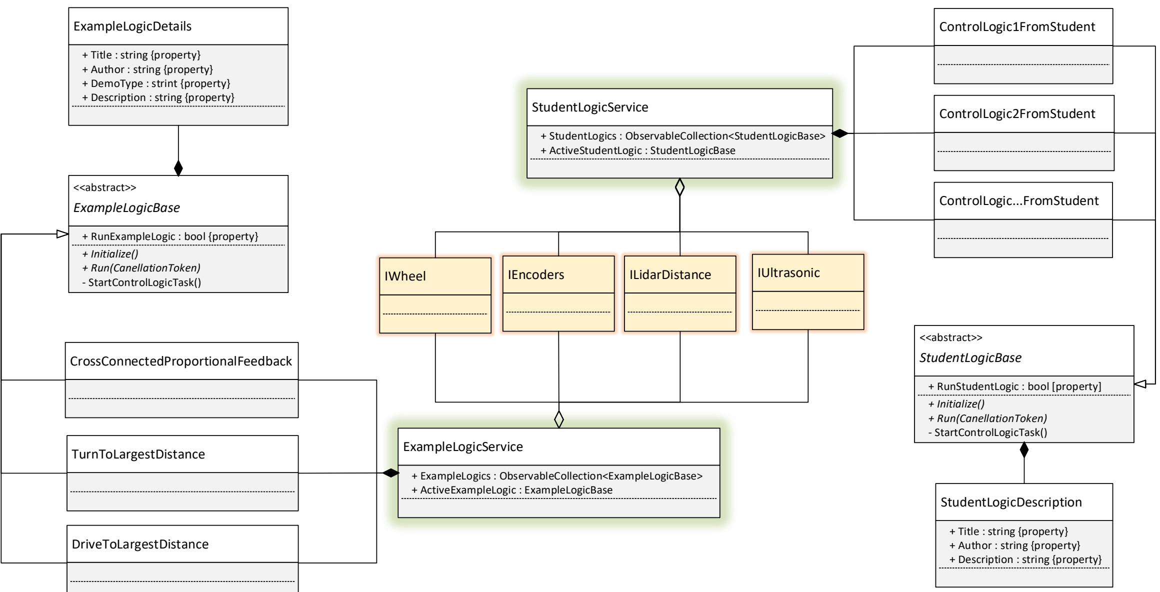
«SynchronizationContext» benyttes for å kunne sette propertier slik at PropertyChanged-eventet trigges selv om koden ikke kjører på UI-tråden (noe som er høyst nødvendig da brukerens kontrollogikk kjører på en egen tråd).

Andre klasser som vi ikke har forklart her (se i kildekoden for info om hvordan de brukes):
«ExclusiveSynchronizedObservableCollection<T>», «ArrayConverter» og «LidarFormatPacketGenerator».

E.2.3 *Klassediagram for «Kontrollogikk»*

Figur 34 viser hvordan modellene for kontrollogikker er bygget opp. Disse gløder grønt rundt kantene for at de skal skille seg ut i tegningen.

Klassene for eksempellogikk og studentlogikk er bygget opp på samme måte, men befinner seg i forskjellige prosjekt for å holde koden så ryddig som mulig, og for å unngå forvirring for studentene. Beskrivelsene her vil ta utgangspunkt i studentlogikk (høyre side av klassediagrammet), men samme prinsipp gjelder også for eksempellogikken.



Figur 34: Klassediagram for "Kontrollogikk"

«StudentLogicService» henter inn en referanse til alt utstyret på bilen (de fire gule interfacene). Disse er bare satt som parametre i kontruktøren til «StudentLogicService», og ettersom de fire interfacene er registrert som IoC-containere med Unity, så blir de automatisk injisert inn i konstruktøren uten at vi har eksplisitt gjort dette i kode.

«StudentLogicService» bruker ikke noen av interfacene direkte, men gir de videre inn i konstruktøren til kontrollogikkene. Kontrollogikkene som studentene lages (vist på klassediagramet med navnene «ControlLogic1FormStudent» osv.) initialsieres her i en liste av type «StudentLogicBase».

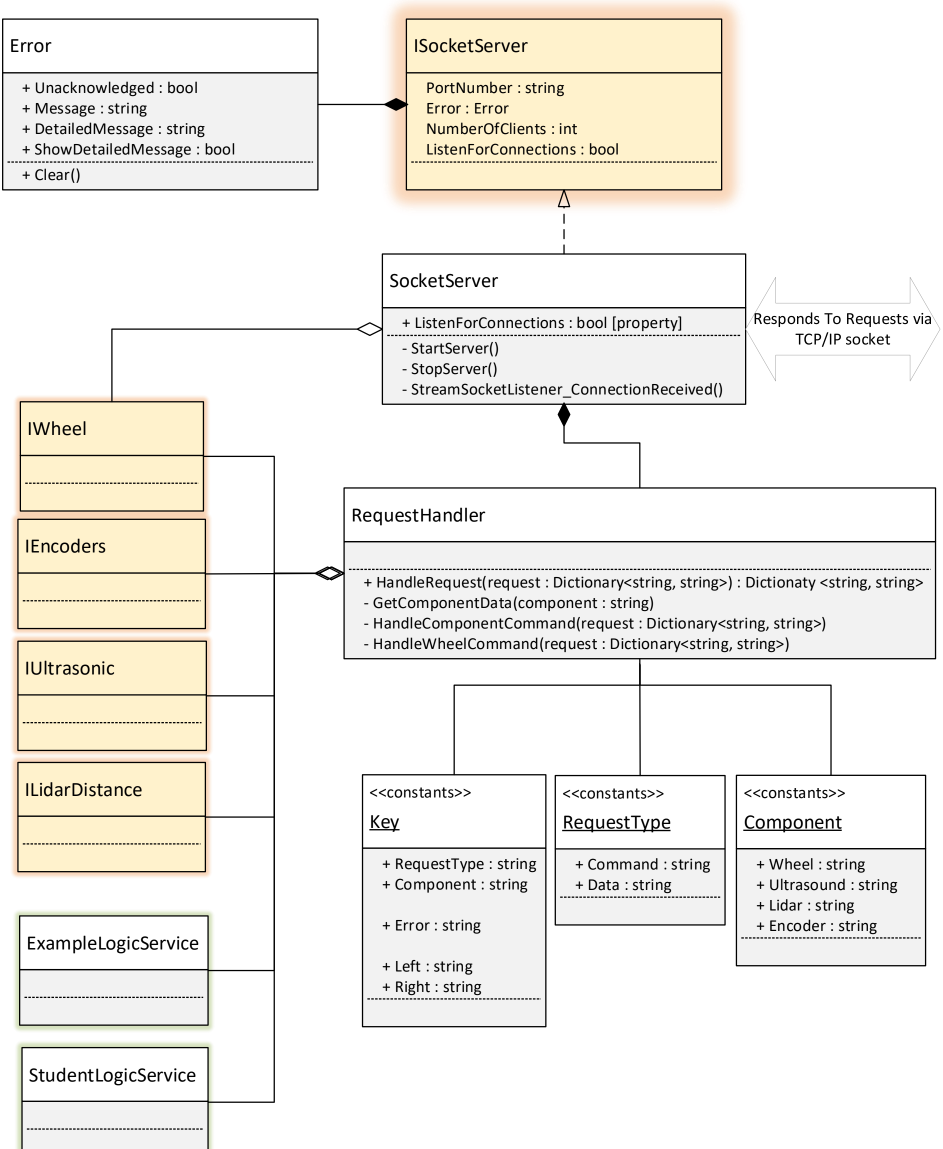
Alle kontrollogikker arver fra den abstrakte klassen «*StudentLogicBase*». Dette lar oss dra nytte av polymorfisme, samt dra ut litt felles kode som unntakshåndtering (*exception handling*) og start/stopp av kontrollogikk.

Studenten trenger bare å forholde seg til en enkel «Run()»-metode i en klasse som arver fra «StudentLogicBase». De bør også legge inn en beskrivelse i «StudentLogicDescription» som de blir tvunget til å override fra «StudentLogicBase», men å faktisk skrive noe inn i den er strengt tatt valgfritt.

Se E.4 for mer informasjon om hvordan man manuelt kan opprette nye studentlogikker.

E.2.4 *Klassediagram for «Socket Server»*

Figur 35 viser hvordan socketserveren er bygget opp. Dette er en eksperimentell funksjonalitet som lar brukerne sende forespørsler til bilen som å sette hjulkommando, stoppe/restarte kontrollogikk eller etterspørre sensordata.



Figur 35: Klassediagram for "Socket Server"

«SocketServer»-klassen tar inn referanser til alle sensormodellene og «Example/StudentLogicService» i konstruktøren. Sensormodellene er registrert som IoC-containere, og blir dermed automatisk injisert av «Unity», slik at man ikke trenger å mate dem inn eksplisitt noen plass i koden.

«SocketServer» beholder bare en referanse til IWheel for å kunne sette hjulhastighet til 0 når en klient avslutter tilkoblingen (enten det er et resultat av et exception eller en kontrollert avslutning). Resten av referansene mates bare videre inn i konstruktøren til «RequestHandler», som benytter disse til å håndtere de forskjellige etterspørslene som SocketServer mottar via TCP/IP-socketen.

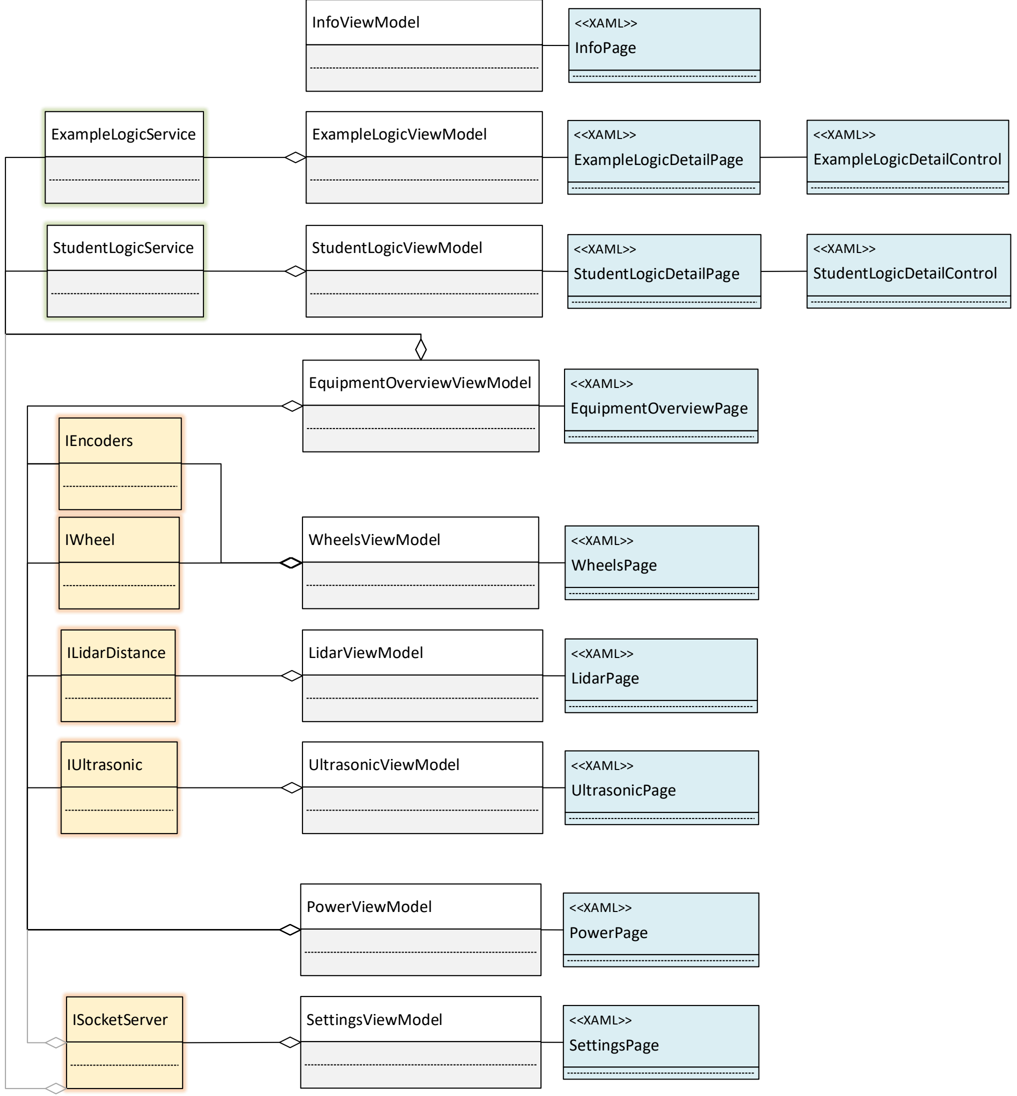
De som ønsker å lage et program som kommuniserer med socketserveren kan se E.5 for mer info angående gyldige kommandoer og korrekt formatering.

E.2.5 *Klassediagram for «ViewModels» og «Views» (GUI)*

Figur 36 viser hvordan Views (XAML-filene merket med blått) og ViewModelene henger sammen, samt hvilke Modelles som benyttes i de forskjellige ViewModelene.

Merk at filer som slutter på «Page» eller «View» er to forskjellige navn-konvensjoner på samme ting. I diagrammet under (og i programmet) slutter «Viewene» på «Page» i stedet for «View» da dette er konvensjonen brukt i Windows Template Studio / Prism.

Alle modellene (første kolonne fra venstre) er registrerte som IoC-containere med unity. Dermed kan man bare skrive dem inn i parameterlisten for konstruktørene til ViewModelene, uten å eksplisitt mate dem inn noe sted.



Figur 36: Klassediagram for "ViewModels og "Views" (GUI)

Viewene binder til propertier som befinner seg i ViewModelene. Vi har for det meste valgt å «expose» modellene i ViewModellen (dvs. vi lager en property i ViewModelen med referanse til Modellen).

«PowerViewModel» tar inn en referanse fra hver av sensorene/utstyret, ettersom den trenger tilgang til «Power»-propertien som ligger inne i hver enkelt komponent. Den benytter ikke noen av de andre propertiene inne i modellene.

«EquipmentOverviewViewModel» tar også inn alle fire, ettersom den tilhørende «Viewen» er designet for å vise data fra flere sensorer samtidig. Den tar også inn «Student/ExampleLogicService» slik at man også skal kunne stoppe en kjørende kontrollogikk fra denne siden.

Sidene for Wheels, Lidar og Ultrasonic tar bare inn sine tilhørende modeller.

«SettingsViewModel» tar inn ISocketServer for at man skal kunne slå socketserveren på/av fra denne siden, samt at bruker skal kunne se meldinger om eventuelle feil som skulle oppstå.

For å holde tegningen så lettlest som mulig, så er det ikke tatt med at Student/ExampleLogicService tar inn referanser fra alle sensor/utstyrsmodellene (se Figur 34: Klassediagram for "Kontrollogikk" for detaljer på dette). For ISocketServer er dette tegnet inn, men med grå linjer i stedet for svarte, for å unngå at det stjeler fokus fra det dette klassediagrammet fokuserer på.

«ShellViewModel»- og «ShellPage»-paret, som er den som definerer den overordnete strukturen på programvinduet (og menylinjen til venstre), er heller ikke tatt med. De andre viewene lastes inn i området definert av ShellPage-viewen.

E.3 Hvordan alt er knyttet sammen i App.xaml.cs (IoC og DI)

Når programmet først starter er koden i «App.xaml.cs» det første som kjøres. Det er her vi har skrevet koden som knytter hele programmet sammen.

Dette kapittelet inneholder en del mer detaljert kode enn tidligere deler av dokumentet, men er viktig for å forstå hele programstrukturen og hvilke designvalg vi har tatt som gjør programmet så fleksibelt. Koden er forenklet litt for å gjøre den mer lettlest. Se kildekoden for å se nøyaktig hvordan den er satt opp.

E.3.1 *Steg 1 – finne ut hvor koden kjører*

Når programmet starter opp så sjekkes det først hvor koden kjøres, og eventuelt om simulator er tilgjengelig. Informasjonen om hvor koden kjører lagres i enumen «RunningState».

```
private void CheckRunningState()
{
    if (Windows.System.Profile.AnalyticsInfo.VersionInfo.DeviceFamily == "Windows.IoT")
    {
        ProgramRunningState = RunningState.OnPhysicalCar;
    }
    else
    {
        SimulatorAppAvailabilityStatus = ... Sjekker simulatorens tilgjengelighetsstatus
        ProgramRunningState = (SimulatorAppAvailabilityStatus == LaunchQuerySupportStatus.Available) ?
                RunningState.AgainstSimulator : RunningState.AgainstMockData;
    }
}
```

E.3.2 *Steg 2 – Registrere IoC-containere*

Ettersom det nå er kjent hvor programmet kjører, og eventuelt om simulatoren er tilgjengelig, så kan korrekt implementassjon for ILidarPacketReceiver, IVehicleCommunication og IGpioPin instansieres, før modellene som skal bruke dem instansieres.

Koden på bildet nedenfor er forenklet litt ved å bare vise initialiseringen for tre av interfacene som er listet opp over switch-blokken.

```
protected override void ConfigureContainer()
{
        ...
    ILidarPacketReceiver lidarPacketReceiver;
    IVehicleCommunication ultrasonicCommunication;
    IVehicleCommunication encoderLeftCommunication;
    IVehicleCommunication encoderRightCommunication;
    IVehicleCommunication wheelCommunication;
    IGpioPin lidarPowerPin;
    IGpioPin ultrasoundPowerPin;
    IGpioPin wheelPowerPin;
    IGpioPin encoderPowerPin;
    IGpioPin ultrasoundInterruptPin;
    switch (ProgramRunningState)
    {
        case RunningState.AgainstMockData:
            lidarPacketReceiver = new MockLidarPacketReceiver();
            ultrasonicCommunication = new MockVehicleCommunication(Device.Ultrasonic);
            encoderPowerPin = new MockGpioPin(GpioNumber.EncoderPower);
            ...
            break;
        case RunningState.AgainstSimulator:
            lidarPacketReceiver = new SimulatedLidarPacketReceiver(_simulatorAppServiceClient);
            ultrasonicCommunication = new SimulatedVehicleCommunication(Device.Ultrasonic, _simulatorAppServiceClient);
            encoderPowerPin = new SimulatedGpioPin(GpioNumber.EncoderPower, GpioPinDriveMode.Output);
            ...
            break;
        case RunningState.OnPhysicalCar:
            lidarPacketReceiver = new LidarPacketReceiver();
            ultrasonicCommunication = new VehicleCommunication(Device.Ultrasonic);
            encoderPowerPin = new PhysicalGpioPin(GpioNumber.EncoderPower, GpioPinDriveMode.Output);
            ...
            break;
    }
    Container.RegisterType<ILidarDistance, LidarDistance>(new ContainerControlledLifetimeManager(),
        new InjectionConstructor(lidarPacketReceiver, lidarPowerPin, new VerticalAngle[] { VerticalAngle.Up1, VerticalAngle.Up3 }));
    Container.RegisterType<IUltrasonic, Ultrasonic>(new ContainerControlledLifetimeManager(),
        new InjectionConstructor(ultrasonicCommunication, ultrasoundPowerPin, ultrasoundInterruptPin));
    Container.RegisterType<IEncoders, Encoders>(new ContainerControlledLifetimeManager(),
        new InjectionConstructor(new Encoder(encoderLeftCommunication), new Encoder(encoderRightCommunication), encoderPowerPin));
    Container.RegisterType<IWheel, Wheel>(new ContainerControlledLifetimeManager(),
        new InjectionConstructor(wheelCommunication, wheelPowerPin));

    Container.RegisterType<ISocketServer, SocketServer>(new ContainerControlledLifetimeManager());
    Container.RegisterType<ExampleLogicService>(new ContainerControlledLifetimeManager());
    Container.RegisterType<StudentLogicService>(new ContainerControlledLifetimeManager());
}
```

Nederst i koden registreres klassene med Unity sin IoC-container.

«ContainerControlledLifetimeManager» gjør at disse containerene blir registrert som «singeltons», dvs. at når man spør Unity etter en instanse av klassen, så vil den bare instansiere klassen én gang. Når noen spør etter en klasse som allerede har blitt instansiert, vil de i stedet få referansen til den eksisterende klassen. Dette gjør at man kan benytte samme instans i hele programmet, uten at man må designe selve klassene som singeltons.

«InjectionContructor» brukes til å mate inn argumenter hvor man registrerer klasser med konstruktører som har parametere som ikke i seg selv er registrert med IoC-containeren.

Det at objektene som andre objekt er avhengig av opprettes her, for å så injisere dem, i stedet for å bare opprette dem lokalt inne i objektene gir programmet stor fleksibilitet med tanke på modifikasjon og utvidelse.

E.3.3 *Steg 3 – Starte resten av applikasjonen (og potensielt simulatoren)*

Etter at typene er registrete i IoC-containeren så lastes hele applikasjonen inn.

Simulatoren vil også startes dersom det i steg 1 ble detektert at programmet ikke kjører på bilen, men på en PC med simulator installert.

```
protected override async Task OnLaunchApplicationAsync(LaunchActivatedEventArgs args)
{
    ApplicationView.PreferredLaunchViewSize = new Size(800, 480);
    ApplicationView.PreferredLaunchWindowingMode = ApplicationViewWindowingMode.PreferredLaunchViewSize;

    await LaunchApplicationAsync(PageTokens.InfoPage, null);
    if (ProgramRunningState == RunningState.AgainstSimulator)
    {
        await LaunchSimulator();
    }
}
```

Når programmet åpnes er det «Info Page» som vises, men denne kan enkelt endres ved å sette inn en av de andre «PageTokens» i «LaunchApplicationAsync()».

E.4 «Student Logic» - Studentenes kontrollogikk

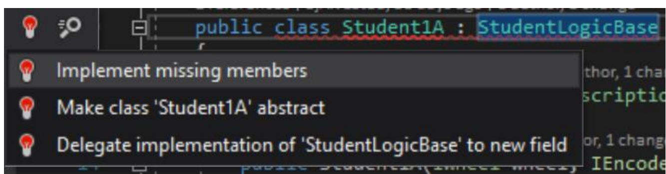
Dette kapittelet viser hvordan kontrollogikken er bygget opp, og hvordan man kan lage egne klasser for kontrollogikk. Vi har allerede laget flere tomme maler for studentlogikk, slik at studentene ikke trenger å gjennomføre det som er beskrevet i dette kapittelet.

Hvis man bare ønsker å se hvordan man kommer i gang med å bruke en av de eksisterende malene for kontrollogikk som vi har laget kan man se C.2

E.4.1 *Opprette ny klasse for kontrollogikken*

For å lage en ny klasse som man kan skrive kontrollogikk i så oppretter man en klasse i «StudentLogic»-prosjektet. Denne klassen må arve fra «StudentLogicBase».

Når man har gjort dette får man en rød strek under klassenavnet ettersom klassen ikke har implementert de abstrakte metodene/propertien «Initialize», «Run» og «Details». Trykk på lyspæren som dukker opp når du markerer konstruktøren for å implementere de manglende medlemmene.



Denne klassen må ha en konstruktør som tar inn de sensorene/utstyret man ønsker å benytte seg av i kontrollogikken. «Wheels» må også gis videre til baseklassen. De resterende lagrer man i lokale variabler.

```
public Student1A(IWheel wheel, IEncoders encoders, ILidarDistance lidar, IUltrasonic ultrasonic) : base(wheel)
{
    Details = new StudentLogicDescription...;

    _wheels = wheel;
    _lidar = lidar;
    _ultrasonic = ultrasonic;
    _encoders = encoders;
}
```

«Details» bør også instansieres her, og man legger inn navn og beskrivelse for kontrollogikken.

```
Details = new StudentLogicDescription
{
    Title = "Empty logic for Student 1 (A)",
    Author = "Student 1",
    Description = "This program is empty.\n" +
                  "Write your control logic in the looping Run() method.\n" +
                  "(And change Title, Author and Description).\n" +
                  "All this is done in the Student1A.cs file."
};
```

For å skrive selve kontrollogikken benytter man «Initialize»- og «Run»-metodene som automatisk ble opprettet tidligere da vi implementerte de manglende medlemmene.

```
public override void Initialize()
{
    // If you have any code you want to run ONCE when starting the control logic, you may put that here (optional).
    // E.g. setting desired number of collection cycles for LIDAR, or default vertical angle etc.

}
```

```
public override void Run(CancellationToken cancellationToken)
{
    // ** WRITE YOUR CONTROL LOGIC HERE **
    // The Run() method will loop until control logic is stopped (or exception occurs)
    // If you write your own loops inside here, then remember to add '&& cancellationToken.IsCancellationRequested' to the loop check

    throw new NotImplementedException();
}
```

E.4.2 *Instansiere den nye klassen*

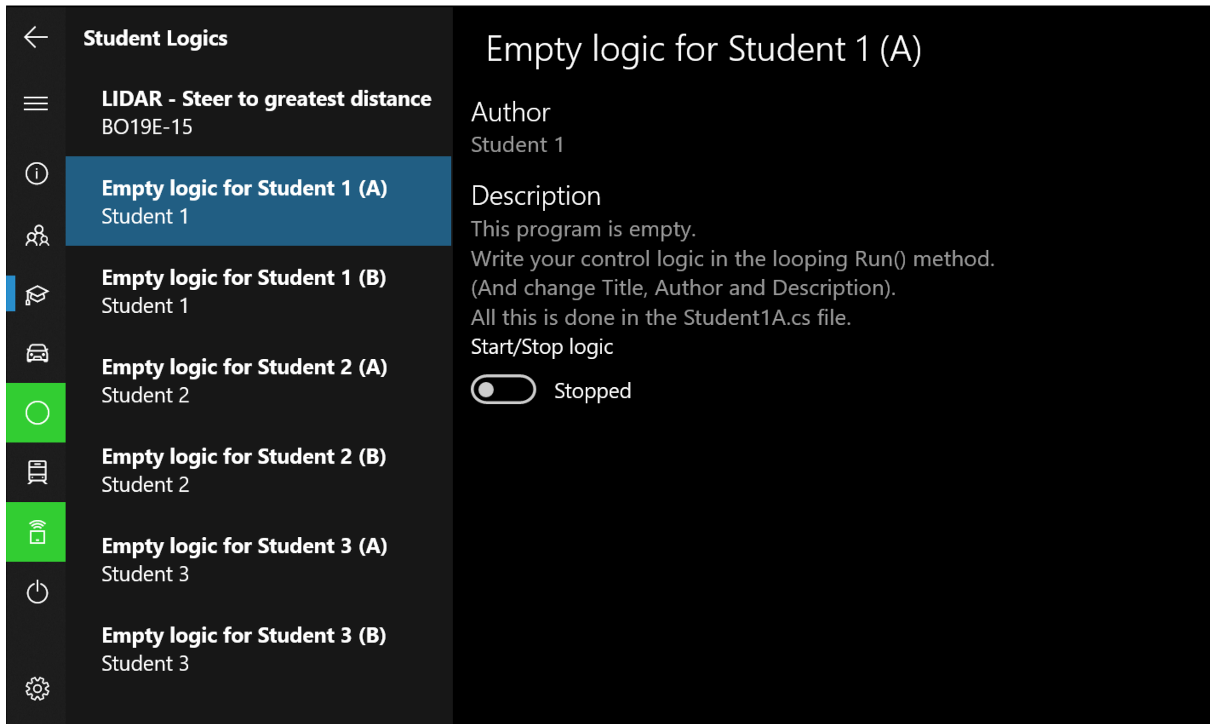
Den nye klassen som nå er opprettet vil ikke være tilgjengelig i programmet før vi har instansiert den i «StudentLogicService.cs». Dette gjøres ved å legge til en linje i «StudentLogics» listen.

```
// Any interface/class registered as a container may be added to the constructor without any further actions
0 references | BjAlvestad, 16 days ago | 1 author, 4 changes
public StudentLogicService(IWheel wheels, IEncoders encoders, ILidarDistance lidar, IUltrasonic ultrasonic)
{
    StudentLogics = new ObservableCollection<StudentLogicBase>
    {
        // Child classes instatiated in the StudentLogics collection will automatically appear in the GUI
        // Pass the sensors to be used as arguments (the ones specified in the constructor of the child class).
        new SteerBlindlyToLargestDistance(wheels, lidar),
        new Student1A(wheels, encoders, lidar, ultrasonic),
        new Student1B(wheels, encoders, lidar, ultrasonic),
        new Student2A(wheels, encoders, lidar, ultrasonic),
        new Student2B(wheels, encoders, lidar, ultrasonic),
        new Student3A(wheels, encoders, lidar, ultrasonic),
        new Student3B(wheels, encoders, lidar, ultrasonic)
    };

    ActiveStudentLogic = StudentLogics.FirstOrDefault();
}
```

E.4.3 *Starte kontrollogikken*

Den nye kontrollogikken vil nå automatisk dukke opp i listen over «Student Logics» hvor man kan starte eller stoppe kontrollogikken.



E.5 «SocketServer» - bruk og kommandoer

Socketserveren kan slås på eller av inne på skjermsiden for «Settings». Den kan motta JSON-formaterte kommandoer, og responderer med informasjon på samme format.

Android-appen vi laget som semesteroppgave i ELE122 bruker disse kommandoene.

Man kan også lage sitt eget program som kommuniserer med bilen over det trådløse nettet med disse kommandoene.

Legg merke til at socketserveren er en eksperimentell funksjon. Om man tenker å lage et komplekst program som benytter seg av denne, så bør man vurdere om man ønsker å refaktorere/modifisere socketserverens «RequestHandler»-kode først.

E.5.1 *Koble seg til «SocketServeren»*

For å koble seg til socketserveren må man gå til «Settings»-siden, og slå på socketserver funksjonaliteten.Der vises IP-addressen og porten til serveren. Porten er fastsatt til 51915. Men IP-adressen avhenger av hvor hovedprogrammet kjører. Den fysiske bilen har fast IP-addresse. På tidspunktet sluttproduktet ble overlevert HVL har den fysiske bilen en SBC har som med 158.37.76.13 som reservert IP-adresse på gjestenettet.

For å koble seg til «SocketServer» på bilen så må man være koblet til Eduroam (mens bilen er koblet til HVLGuest).

For å koble seg til «SocketServer» når programmet kjører lokalt på PC-en, kan det hende at man manuelt må åpne for tilgang på port 51915 i brannmuren.
PS: På grunn av «nettverksisolasjon» kan man ikke koble seg til «SocketServer» fra et program som kjører på samme PC som hovedprogrammet. Dette er en beskyttelse som er lagt inn for UWP apper.

E.5.2 *Gyldige format*

Kommandoene må være på formatet { "NØKKEL": "Verdi"}, { "NØKKEL": "Verdi"}.

Gyldige nøkler som kan sendes til «SocketServer» er:
`REQUEST_TYPE`, `COMPONENT`, `LEFT` og `RIGHT`.

Gyldige verdier for REQUEST_TYPE nøkkelen er:
`Command` og `Data`.

Gyldige verdier for COMPONENT nøkkelen er:
`Wheel`, `Ultrasound`, `Lidar`, `Encoder`, `StopControlLogic`, `RestartControlLogic.`

Gyldige verdier for LEFT og RIGHT nøklene er:
Heltall (som streng) mellom -100 og 100.
LEFT og RIGHT nøklene brukes bare med REQUEST_TYPE Command, og COMPONENT Wheel.

E.5.3 *Sette hjulhastighet*

Det er mulig å styre hjulhastigheten på bilen via kommando til socketserveren. Kommandoen vil bare aksepteres dersom det ikke kjører noen kontrollogikk. Socketserveren returnerer enten en bekreftelse på satt hjulhastighet eller en feilmelding.

Gyldige verdier for LEFT og RIGHT kommando er verdier mellom -100 og 100.

Kommando til server:
{ "REQUEST_TYPE": "Command", "COMPONENT": "Wheel", "LEFT": "30", "RIGHT": "-30" }

Svar fra server – ved suksess:
{"Wheel":"Left: 30, Right: -30."}

Svar fra server – hvis kontrollogikk kjører:
{"Wheel":"Can't give command while control logic is running. Stop control logic first."}

E.5.4 *Stopp og re-start av kontrollogikk*

Man kan stoppe en kjørende kontrollogikk, eller starte kontrollogikken man stoppet på nytt.

Kommando til server:

Stopp kjørende kontrollogikk:
{ "REQUEST_TYPE": "Command", "COMPONENT": "StopControlLogic"}

Start forrige kontrollogikk:
{ "REQUEST_TYPE": "Command", "COMPONENT": "RestartControlLogic"}

Svar fra server:

Etter stoppkommando:
{"StopControlLogic":"Stopped demo logic: 'Cross-connected P-feedback'"}

Etter restartkommando:
{"RestartControlLogic":"Started demo logic: \n'Cross-connected P-feedback'"}

Hvis kontrollogikk ikke kan stoppes eller startes:
{"StopControlLogic":"No running logic that can be stopped.\nStopped wheels instead."}
{"RestartControlLogic":"No previous control logic available to start"}

E.5.5 *Lese sensordata*

Generell kommando til server for å lese sensordata:
{ "REQUEST_TYPE": "Data", "COMPONENT": "Comp1 Comp2 etc" }

Hvor gyldige COMPONENT-verdier er `Wheel`, `Ultrasound`, `Lidar` (og `Encoder`). Flere kan listes opp i en melding, separert med mellomrom.

Eksempel på å lese data bare fra ultralyd:

Komando til server:
{ "REQUEST_TYPE": "Data", "COMPONENT": "Ultrasound" }

Svar fra server:
{"Ultrasound":"Left: 0.52, Fwd: 1.25, Right: 0.95."}

Eksempel på å lese fra ultralyd og hjul:

Komando til server:
{ "REQUEST_TYPE": "Data", "COMPONENT": "Ultrasound Wheel" }

Svar fra server:
{"Ultrasound":"Left: 0.52, Fwd: 1.19, Right: 0.95.","Wheel":"Left: 0, Right: 0."}

Merk at selv om nøkkelordet for Encoder har blitt lagt inn, så har ikke socketserverens «RequestHandler» blitt konfigurert for å sende ut verdier for denne enda (sender bare melding om at dette ikke er implementert).

Dataene som returneres i verdifeltet for hver komponent kan enten være målte verdier, som vist i eksemplene ovenfor, eller det kan være en feilmelding/beskjed.

Ved bruk av ugyldige komponentnavn vil serveren svare med at den komponenten ikke er støttet:

Eksempel på kommando til server med feil komponentnavn:
{ "REQUEST_TYPE": "Data", "COMPONENT": "Ultraso" }

Svar fra server:
{"Ultraso":"THIS COMPONENT DOES NOT SUPPORT DATA REQUESTS --> Ultraso."}

E.5.6 *Svar fra server ved feil i nøkkel eller formatering*

Hvis det er feil i kommandoen som sendes til serveren, som f.eks. ugyldig nøkkel eller feilformatert JSON-streng, så vil serveren respondere med en melding på følgende format:

{"ERROR":"Informasjon om feilen kommer her."}

Eksempel hvor man ikke har med nøkkelen COMPONENT:
{ "REQUEST_TYPE": "Data" }

Svar fra server:
{"ERROR":"Unable to handle request. \nException message: \nData request did not contain the required key --> COMPONENT"}

Appendiks F Software – Simulator

Simulatoren er, i likhet med hovedprogrammet, en UWP-app. For å lage simulatoren har vi benyttet «MonoGame» som er et rammeverk med åpen kildekode for spillutvikling i C# [23]. Simulatoren er derfor egentlig et spill som hovedprogrammet kommuniserer med.

Simulatoren vil starte av seg selv når hovedprogrammet starter. Selve simulatoren kan enkelt installeres fra «Microsoft Store». Informasjonen i dette kapittelet er derfor hovedsakelig relevant for de som skulle ha spesiell interesse av å modifisere simulatoren. Kildekoden finner man i GitHub repositoriet «BO19E-15_SensorVehicle».

For å kunne jobbe med simulatorkoden må man først installere «MonoGame».

F.1 Prosjekt og klasseoversikt

Simulatoren er et betydelig enklere program enn hovedprogrammet. Bildet viser en oversikt over hele prosjektstrukturen til simulatoren.

Klassene som ligger i mappene «Sprite and sensors» og «Screen and scaling» er klasser som benyttes i selve spillogikken.

Klassene som ligger i «Map and boundaries» mappen (se gul ramme) er for innlasting av kart og generering av «boundaryboxes» rundt veggene/hindringer (som så brukes for avstandsmålingen).

Klassene som ligger i mappen «SensorVehicleApp interface» (se grønn ramme) inneholder en klasse for kommunikasjon med hovedprogrammet, og klasser for å generere simulert data for hovedprogrammet basert på simulatorens spillogikk.

Teksten som vises på skjermen i simulatoren er definert i «GamePage.xaml».

«Content.mgcb» og «SimulatorGame.cs», merket med røde rammer, er spesielle for «MonoGame». «**Content.mgcb**» brukes for å laste inn ressurser som skal brukes i programmet (f.eks. kart og bilde for den simulerte bilen). «**SimulatorGame.cs**» er hovedfilen for simulatoren, og er hvor koden for oppdatering av spillogikk og tegning av skjermbilde ligger. Det er fra denne filen vi benytter alle de andre klassene.

I tillegg til vår egen kode benytter vi to små bibliotek som er lagt til via NuGet. Disse er TiledSharp, et bibliotek for importering av kart laget i *Tiled*, og Comora - et enkelt 2D kamera for *MonoGame*. Vi har valgt disse da de er små og oversiktlige, og uten mange overflødige funksjoner som vi ikke har bruk for. Begge bibliotekene er tilgjengelige på GitHub.

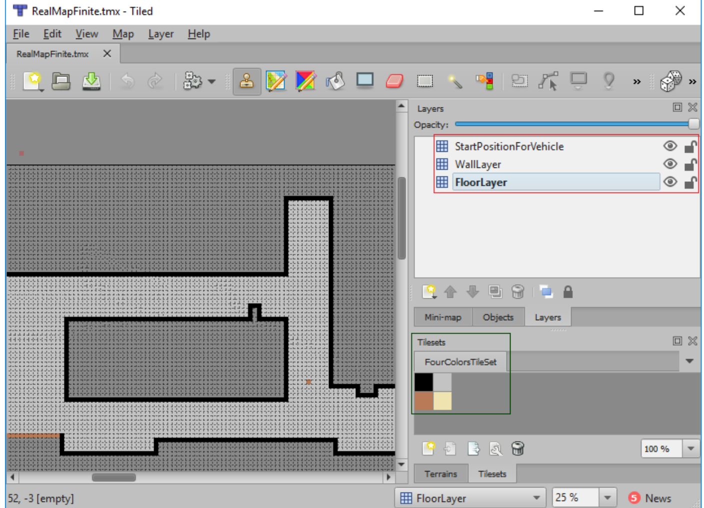
For tegning/design av kart har vi benyttet et program som heter «Tiled».

F.2 Lage kart for simulator med «Tiled»

Det er laget et kart som vil bli levert sammen med sluttproduktet. Dette er tegnet etter korridorer på HVL. Det vil i dette kapittelet bli presentert hvordan man skal gå frem dersom man vil lage andre kart for simulatoren.

Tiled er et program som brukes til å tegne 2D-kart for spillprogrammering. Kartene tegnes ved at man legger inn forskjellige lag (*layers*) og velger hvilket nivå de enkelte lagene skal ligge i. Man har mange tilgjengelige typer lag, men vi har bare benyttet oss av det som kalles *Tile Layer*.

Tile layers gjør det mulig å enkelt tegne store ensfargede områder som i seg selv ikke inneholder mye informasjon. Kartene som brukes i simulatoren inneholder tre lag – ett som skal illustrere gulvet, ett som blir tolket som vegger, og ett som bestemmer startposisjonen til bilen. Den røde rammen på Figur 37 viser de ulike lagene, og den grønne viser de forskjellige fargene som kan brukes på dette kartet.

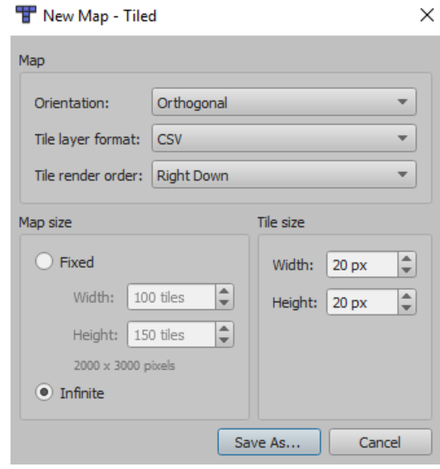


Figur 37: Oversiktsbilde av Tiled

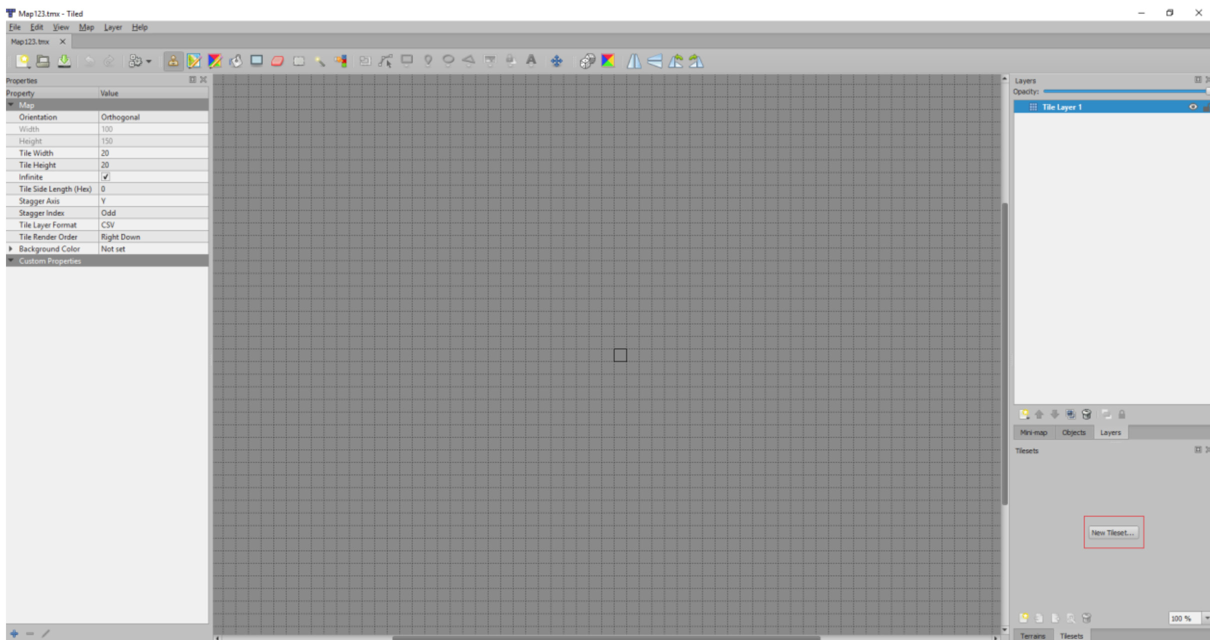
F.2.1 *Hvordan lage kart*

1. Gå til https://thorbjorn.itch.io/tiled og last ned «Tiled».
2. Dersom vinduet som vises på Figur 38 spretter opp, velg «Mer info» og så «Kjør likevell»
3. Følg så installasjonsguiden
4. Åpne «Tiled» og velg «File» → «New» → «New Map…»
5. Innstillingene for tegning av kart skal settes slik som vist på Figur 39.
    a. MERK: Det å velge uendelig kartstørrelse (infinite) gjør selve tegningen enklere, men for at kartet skal brukes i simulatoren **må** kartetstørrelsen endres til endelig (finite) etter man er ferdig å tegne
6. Legg så til et «Tileset». Det anbefales å bruke «Tilesetet» som følger med sluttproduktet, men det er mulig å lage sitt eget.
    a. Dersom man ønsker å tegne sitt eget «Tileset» gjør man dette i Paint, og man må da sørge for at hver fargerute har samme dimensjoner som «Tile size» man velger på Figur 39.

Figur 38: Vindu som spretter opp ved åpning av installasjonsguide for Tiled

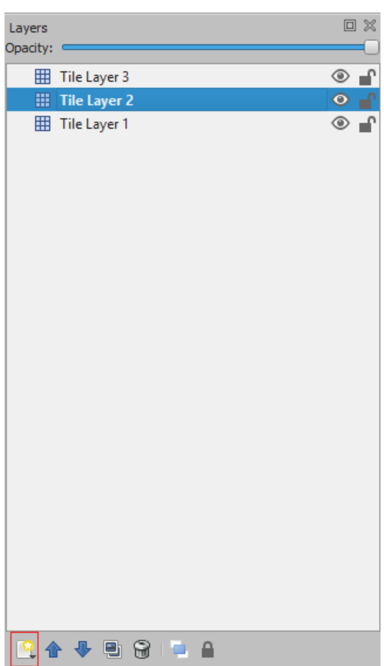


Figur 39: Innstillinger for tegning av kart

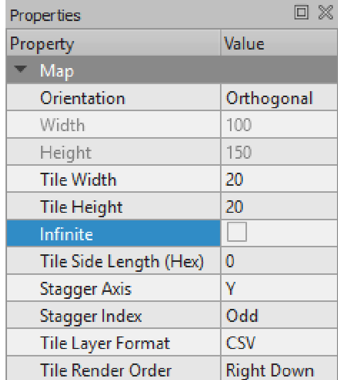


Figur 40: Tiled ved oppretting av nytt kart

7. Velg «New Tileset…» som er vist i den røde rammen på Figur 40.
8. Velg «Browse» i vinduet som spretter opp og finn ønsket «Tilset».
    a. For at man skal slippe å sette inn parameterene hver gang man skal tegne et kart med det samme «tilesetet» er det anbefalt at «checkboxen» for «Embed in map» ikke er huket av [24].
9. Legg så til flere lag i kartet ved å trykke på knappen i den røde rammen på Figur 41 og velg «Tile Layer».
    a. Man kan flytte lagene opp og ned ved å bruke de blå pilene
    b. «Tile Layer 1» blir tolket som gulv.
       «Tile Layer 2» blir tolket som vegger.
       «Tile Layer 3» bestemmer startposisjonen til bilen.
10. Trykk så på ønsket del på «Tilesetet» og begynn å tegne.
11. Sørg for at kartet ikke er uendelig (se Figur 42).
12. Last så både kartet (.tmx) og det tilhørende «tilesetet» (.png) inn i «Monogame Pipeline Tool»
    a. Dette gjøres ved å åpne «Content.mgcb». Dersom filen åpnes som tekstfil, høyreklikker man på filen og velger «Open with…» → «Monogame Pipeline Tool».
    b. Høyreklikk så på mappen «Maps» og velg «Add» → «Existing Item…» og finn kartet og «tilesetet». Dersom det spretter opp en boks som sier at filen ikke er i målmappen, velg «Copy to directory» → «Add»



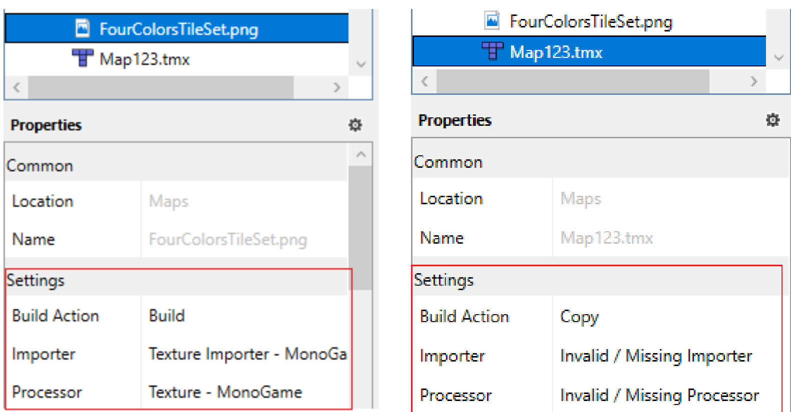
Figur 41: Oversikt over Tile Layers



Property	Value
Map	
Orientation	Orthogonal
Width	100
Height	150
Tile Width	20
Tile Height	20
Infinite	☐
Tile Side Length (Hex)	0
Stagger Axis	Y
Stagger Index	Odd
Tile Layer Format	CSV
Tile Render Order	Right Down

Figur 42: Sørg for at Infinite ikke er huket av

c. Figur 43 viser innstillingene tilesetet og kartet skal bygges med.



Figur 43: Innstillinger for .png og .tmx

Appendiks G Software – Terminologi og benyttede prinsipper

Hovedappen, simulator-appen og «extras»-appen er alle prosjekt av typen UWP (Universal Windows Platform).

Hovedprogrammet er opprettet ved hjelp av Windows Template Studio, hvor vi benytter MVVM (Prism/Template10) som design pattern.

«Extras» programmet er opprettet med default UWP template i Visual Studio, og benytter «CodeBehind» som design pattern.

Simulatoren er også en UWP app, men er opprettet med «MonoGame» template.

I kapitlene nedenfor forklarer vi generelt om prosjekttype, verktøy og prinsipper vi har brukt. Kapittel G.2 - G.4 gjelder primært for hovedprogrammet.

G.1 UWP – Universal Windows Platform

De «moderne» appene man finner på Windows 10 PC-er er UWP apper. Disse kan programmeres i språkene C#, C++, VB eller JavaScript. For UI-delen kan man benytte XAML, HTML eller DirectX [25]. I både hovedprogrammet og simulatoren benyttes C# og XAML. Dette gjør også extras-appen, men i tillegg benyttes JavaScript, HTML og CSS for å opprette en nettside og strømme video.

UWP-appene har en del ulikheter fra Desktop-apper som Windows Forms og WPF (Windows Presentation Foundation) som er verdt å nevne.

G.1.1 *Målversjon og manifest*

Desktop apper, som Windows Forms og WPF, har en spesifikk .NET Framework versjon som mål. For UWP velger man derimot et område av Windows-versjoner, avgrenset av minimumsversjon og målversjon. Ved å velge høyere versjon får man flere funksjoner som kan benyttes, men man begrenser samtidig hvilke maskiner det kan kjøres på [26].

Man må også deklarere hvilke kapabiliteter programmet skal ha i manifestet «Package.appxmanifest». Når appen kjøres får brukeren info om / må bekrefte tilgang til de etterspurte resursene [27].

G.1.2 *Nettverksisolasjon i UWP apper (og App Service)*

På grunn av «nettverksisolasjon» er det ikke mulig å opprette en sockettilkobling mellom to UWP-apper som kjører på samme PC [28]. Det ser ut til at det samme er tilfellet selv for socket-kommunikasjon mellom en UWP app og en Desktop app (PuTTY), men vi har ikke funnet et offisielt utsagn for at dette skal være tilfellet.

Nettverksisolasjon kan slås av lokalt når man debugger ved å velge «Allow local network loopback» under prosjektets «Debug» meny.

For å kommunisere mellom UWP-apper kan man benytte «AppService», som lar oss sende «ValueSets» (nøkkel-verdi-par) mellom appene [29]. Det er dette vi har benyttet for kommunikasjon mellom hovedapp og simulator.

G.1.3 *Native kode vs. Intermediate Language*

Windows Forms og WPF kompileres som standard til Intermediate Language (IL). Desktop-programmet kjører i en virtuell maskin (VM), hvor IL tolkes til maskinkode når programmet kjører (kalt just-in-time kompilasjon).

UWP-apper er litt spesielle ved at de kompileres til IL hvis man velger Debug som «solution configuration» i Visual Studio (for å spare tid), mens hvis man velger Release, så komplieres koden til Native Language (som er mer likt den koden som sluttbruker kjører på sin PC). Kode som kjører native (e.g. C++, og UWP-C#) har vanligvis bedre ytelse enn tolket kode [30].

UWP-appene i Microsoft Store ligger der som IL, men kompileres til nativ kode for brukerens maskin når du installerer appen [31].

G.2 Windows Template Studio

Windows Template Studio er en utvidelse fra Microsoft for Visual Studio 2017 og 2019 som hjelper til med å generere et utgangspunkt for en app som er slik man ønsker. Man kan blant annet velge «design pattern» man ønsker som f.eks. den tradisjonelle «Code Behind», eller den kan hjelpe til med å sette opp MVVM med f.eks. Caliburn Micro, Prism eller andre MVVM framework.

Benytter man Windows Template Studio for å generere et nytt UWP-prosjekt i stedet for å benytte templatet som følger med Visual Studio, kan man spare flere timer med kjedelig arbeid for å sette prosjektet opp slik man ønsker [32].

G.3 MVVM – Model-View-ViewModel (med Prism/Template10)

MVVM – Model-View-ViewModel er et arkitekturmønster som ble utviklet av «Microsoft Patterns and Practices» for å forenkle utviklingen av brukergrensesnittet i et hendelsesdrevet program (e.g. hvor programflyten styres av hendelser som klikk med mus, eller input på touchskjerm).

WPF (Windows Presentation Foundation) og UWP (Universal Windows Platform) er begge tilrettelagt for bruk av MVVM.

View er XAML-koden (for WPF og UPW prosjekt). I denne skrives koden som omhandler det brukeren ser på skjermen. Selve dataen kommer gjerne fra modellene, men view kjenner ikke til modellene direkte. De «eksponeres» eller «wrappes» i viewmodelen, og viewen «binder» til metodene/propertiene i viewmodellen.

ViewModel er bindeleddet mellom «Model» (klassene våre) og «View» (XAML koden). Viewmodelen har ikke noen referanser til view, det er view som binder til viewmodelen.

Model er «business-logikken», dvs. de klassene man ville brukt enten man skulle lage programmet som et konsollprogram, Windows Forms, UWP etc. For modellene som skal brukes i MVVM kan det være en fordel å ha dem til å implementere «INotifyPropertyChanged» for å forenkle binding til propertier i modellen som «exposes» via ViewModelen.
Modellen har ingen kjennskap til View eller ViewModel.

Binding har blitt nevnt flere ganger ovenfor som det som brukes mellom View og ViewModel. I WPF så benyttes kommandoen «Binding» for å opprette en kjøretidsbinding til en metode, eller en property (som vanligvis implementerer INotifyPropertyChanged). I UWP ble det i tillegg introdusert en ny type binding kalt x:Bind, som er en «compile-time»-binding. Denne gir større sikkerhet og bedre ytelse enn «Binding» (hvor feil i bindingen ikke oppdages før programmet kjører).

Ved å bruke MVVM så får man separert / oppnår løs kobling mellom kode for brukergrensesnitt og andre deler av koden. Dette forenkler «unit testing» og gjør koden lettere å vedlikeholde [33].

Det finnes en rekke rammeverk som gjør det enklere å benytte MVVM. Den vi har benyttet heter «Prism», som ble opprinnelig laget av «Microsoft patterns & practices», men nå vedlikeholdes/videreutvikles som åpen kildekode [34]. Den versjonen vi benytter av Prism for UWP er egentlig «Template 10.2», som har blitt flyttet tilbake til Prism [35].

G.4 Løs kobling mellom modellene

At koden er løst koblet er viktig både for å unngå bugs og for å gjøre programmet mer fleksibelt med tanke på endringer/utvidelser i ettertid.

I G.3 var det beskrevet hvordan MVVM kunne benyttes for å oppnå løs kobling mellom kode i forbindelse med brukergrensesnittet. I dette kapittelet presenteres metoder man kan dra nytt av for å oppnå løs kobling mellom modellene.

G.4.1 *Interface og polymorfisme*

Et interface beskriver signaturen til (public-)metoder/properties som klassene som implementerer den må inneholde. Den fungerer dermed som en garanti/kontrakt på at alle klasser som implementerer et gitt interface kan kjøre metodene/propertiene som interfacet beskriver. Selve implementasjonen av metodene er det opp til hver enkelt klasse å definere [36].

Et **interface** kan ikke instantieres, men vi kan benytte et interface som referanse til et objekt av en hvilken som helst klasse som implementerer dette interfacet. Man har da kun tilgang til de metodene som interfacet spesifiserer, og må caste tilbake til objektets run-time-klasse dersom det er ønskelig å benytte metoder som interfacet ikke spesifiserer.

Det at man kan ha objekter av forskjellige klasser knyttet til et interface-referanse når programmet kjører, mens man i programkoden bare forholder oss til metodekallene som fines i interfacet gjør at man kan dra nytte av *polymorfisme*.

Polymorfisme betyr «mange former» og er regnet som en av de fire søylene av objektorientert programmering (de andre er abstrahering, innkapsling og arv) [37]. Hvis man kaller metoder via interfacer, så kan man lett endre oppførselen til et program ved å endre hvilken type klasse interfacet peker på.

I E.2.2 står det beskrevet hvordan vi har dratt nytte av interfacer og polymorfisme, slik at programmet vårt kan endre oppførsel avhengig av om det kjører på PC eller den fysiske bilen.

G.4.2 *Inversion-of-Control container og Dependency Injection*

Ved å benytte «Inversion-of-Control» (IoC) flyttes ansvaret med å opprette og holde instansen fra den konsumerende klassen til en ekstern container (f.eks. «Unity» av Microsoft) [38].

«Dependency injection» (DI) er når man instansierer objektene en klasse skal benytte seg av utenfor klassen, for så å mate instansene inn via konstruktøren. Dette skaper en løsere kobling mellom klassene enn om man bruker «new» nøkkelordet inne i klassen til å instansiere objektene man trenger. Dersom man i tillegg benytter interfacer oppnår man enda større fleksibilitet i programmet, ved at man kan mate inn en hvilken som helst klasse som implementerer interfacet. Da får man endret programmets oppførsel uten at man trenger å gjøre noe i selve klassen.

For å endre klassen uten DI til å benytte seg av en annen klasse «SimulertKomunikasjon» som implementerer «IKommunikasjon», så må man gjøre inngrep i selve klassen. Det må også legges til logikk inne i «KlasseUtenDI» som bestemmer hvilke som skal brukes når etc (se bildene til høyre).

```
class KlasseUtenDI
{
    private IKommunikasjon kommunikasjon;
    0 references
    public KlasseUtenDI()
    {
        kommunikasjon = new I2cKommunikasjon();
    }

    // ...
}
```

Hvis vi i stedet hadde benyttet oss av dependency injection, kunne man matet inn referansen til «SimulertKommunikasjon» i stedet for «I2cKommunikasjon». Med andre ord så slipper «KlasseMedDI» å forholde seg til hvilken implementasjon av interfacet den tar inn. Noe som gjør programmet mye mer fleksibelt. Vi skal ikke gå mer detaljert inn på alle fordelene dette gir her, men for lesere som driver med objektorientert programmering, og ikke har kjennskap til dependency injection, anbefales det på det sterkeste lese seg litt opp på dette og benytte det i sine fremtidige program.

```
class KlasseMedDI
{
    private IKommunikasjon kommunikasjon;
    0 references
    public KlasseMedDI(IKommunikasjon injisertKommunikasjon)
    {
        kommunikasjon = injisertKommunikasjon;
    }

    // ...
}
```

Dependency injection brukes ofte sammen med IoC til å stappe inn instansene av objekter som en klasse skal bruke via dens konstruktør, uten at vi eksplisitt må mate dem inn.