

BEERTRESS

System design og implementering

Semesterprojekt 3, Gruppe 8

Peter Wann

201907121

August Hjerrild Andersen

201907251

Simon Phi Dang

201705957

Henry Pham

201606071

Alexander Flarup Wodstrup

201810602

Lucas Friis-Hansen

201811527

Jim Sørensen

201602614

Shynthavi Prithviraj

201807198

17. december 2020

Indhold

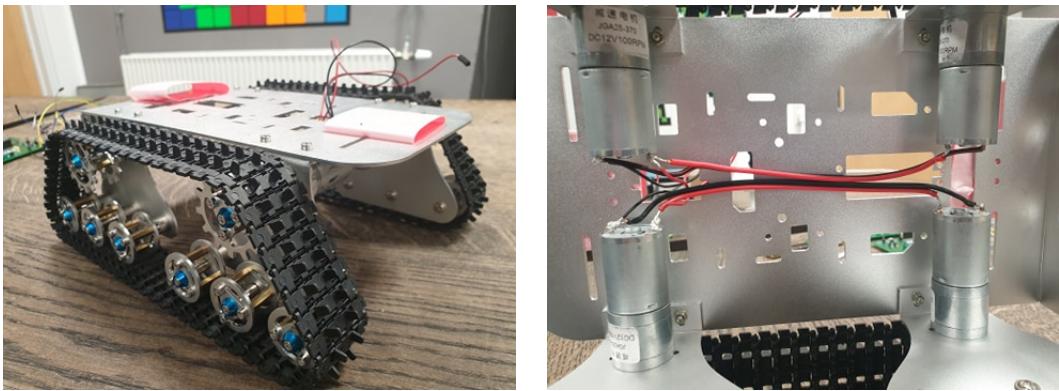
1 Hardware Design	2
1.1 Køretøj	2
1.2 Batterier og Spændingsregulator (PSU)	2
1.3 Farvesensor	3
1.4 Afstandssensor	4
1.5 Vægt sensor	4
2 Hardware Implementering	6
2.1 Køretøj	6
2.2 Batteri og Spændingsregulator (PSU)	7
2.3 Farvesensor	9
2.4 Vægt sensor	10
3 Software Design	11
3.1 InterfaceController_SW	11
3.1.1 Sekvensdiagram	12
3.1.2 Kundegrænseflade	13
3.2 WaiterApp_SW	16
3.3 MotorController_SW	16
4 Software Implementering	20
4.1 WaiterApp_SW	20
4.2 InterfaceController_SW	21
4.2.1 Websocket	24
4.2.2 Kundegrænseflade	24
4.3 MotorController_SW	26
4.3.1 i2cKommunikation	27
4.3.2 Scale	28
4.3.3 motorStyring	30
4.3.4 Farvesensor	32
4.3.5 Afstandssensor	37

1 Hardware Design

I det følgende afsnit vil hardware designet til projektet Beertress blive beskrevet. Her vil blive gennemgået valg af komponenter og hvordan de implementeres.

1.1 Køretøj

Eftersom vi lånte vores køretøj ved Embedded Stock[7] vil denne ikke blive beskrevet dybdegående. Den består af et aluminiumskarosseri og fire 12V DC-motor, der er sat sammen parvis og styrer henholdsvis venstre og højre side af larvefødderne. Køretøjet kan ses på nedenstående billede:



Figur 1: Køretøj lånt af Embedded Stock

1.2 Batterier og Spændingsregulator (PSU)

Der blev lånt en batterikuffert ved Embedded stock[7], som bestod af et 7.2V NiHm batteri og et 9.6V NiMh samt oplader. Disse kan ses på nedenstående billede.

Systemet skulle forsynes med 9.6V til motor, 3.3V til Raspberry Pi og 5V til resten af systemet. Der skulle derfor designes en spændingsregulator, der kunne regulere spændingen fra batteriet på 7.2V om til hhv. 5V og 3.3V. Designet til kredsløbet kan ses på nedenstående billede, hvori komponenterne LM7805[5] og LM1085[4] bruges. LM7805 regulerer forsyningsspændingen om til 5V og LM1085 regulerer de 5V om til 3.3V.

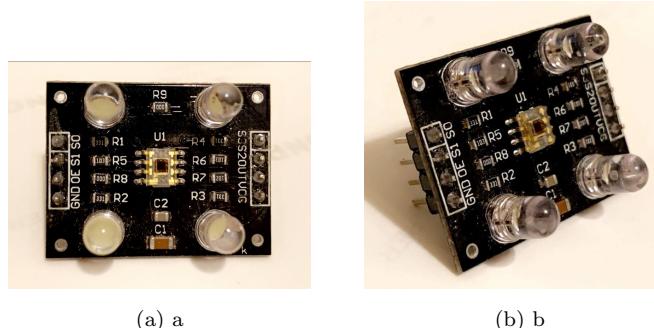
Som det kan ses på proberne, får vi det ønskede output, så kredsløbet opfører sig som forventet.



Figur 2: Batterier lånt af Embedded Stock

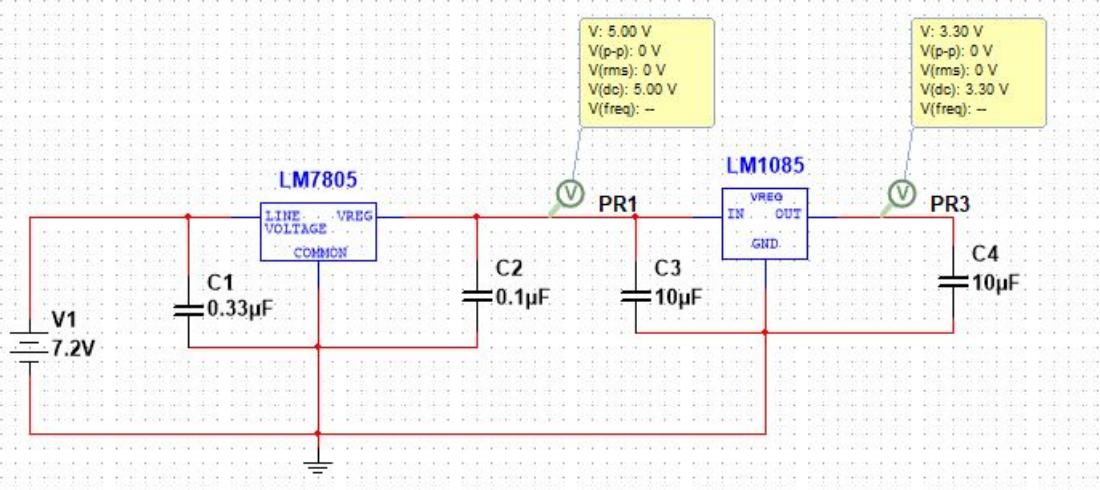
1.3 Farvesensor

Farvesensoren har til formål at navigere BeerTress hen til bordene og tilbage til startposition. Der er derfor blevet lånt en farvesensor TCS230 i Embedded Stock. ColorSensor modulet består af selve farvesensoren og fire infrarøde LED'er.



Figur 4: Color Sensor fra Embedded Stock

Selve sensoren, som ligger i midten, består af et 8x8 array fotodioder med 4 farvefiltre i toppen. Disse fire typer af filtre er rød, blå, grøn og clear. Da der er flere fotodioder til hver farve, hjælper det med at give en præcis aflæsning af en specifik farve. Farvesensoren har otte pins til sammen. Disse består af en ground, VCC, output frekvens, OE og signalerne S0, S1, S2 og S3. S0 og S1 er output frekvens skalering, som er en intern clock chip der tæller input signalet. Jo højere værdien er, desto højere er følsomheden. S2 og S3 er fotodiodetyperne, der afgør hvilken farven sensoren skal aflæse.



Figur 3: Kredsløb for Spændingsregulator (5V og 3.3V)

1.4 Afstandssensor

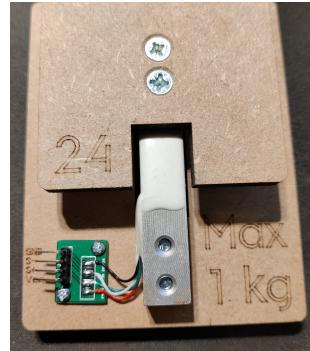
Formålet med afstandssensoren er at undgå sammenstød mellem Beertress og objekter. Da der hardwaremæssigt ikke er foretaget ændringer på afstandssensoren, henvises der til databladet [1], og afstandssensoren beskrives derfor kun kort og overfladisk.

Sensoren består af 4 pins, VCC, Ground, Trigger og Echo. For at aktivere sensoren sendes et højt signal til trigger i 10us. Det får den til at udsende en lydbølge, som består af 8 impulser på 40 kHz. Straks efter impulserne er sendt, går signalet på Echo højt, og når lydbølgen kommer tilbage, går signalet lavt. Tidsperioden hvor signalet på Echo har været højt, kan derefter konverteres til en afstand ved at kende lydens hastighed.

1.5 Vægt sensor

Vægtsensoren som har formålet til at holde øje med vægten af ”produktet”, hvilket i dette tilfælde er øl som der skal serveres. Når vægt sensoren detektere at vægten ændrer sig, skal dette sendes videre til PSoC, der skal behandle signalet. Vægtsensor modulet består af en load cell og en amplifier med formålet at øge signalet, der kommer fra load cellen. Komponenter der anvendes kan ses på figur 5. Da der er begrænset mulighed for hvor meget vægt, der kan komme på Beertress, var der valgt at anvende en 1 kg load cell. Det var ikke nødvendigt at anvende mere end dette, da der kun

bliver målt 1 øl dåse.



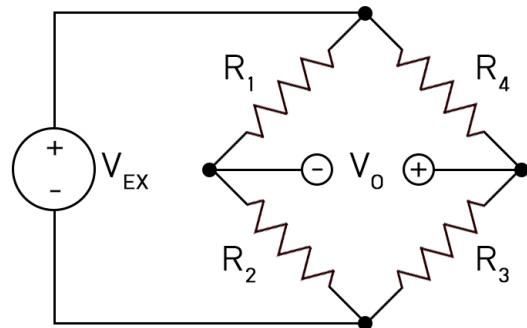
(a) Load Cell



(b) Load Cell Amplifier

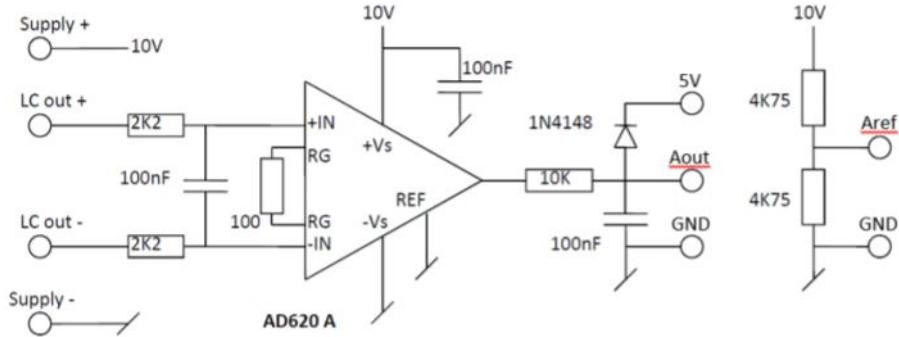
Figur 5: Moduler til vægt sensor

Disse komponenter er blevet lånt af elektronik værkstedet i anledning af det blev anvendt til en Undervisnings øvelse fra et af semesters fag. Load cellen består af en wheatstone bridge, som kan ses nedenfor på figur 6



Figur 6: Wheatstone bridge [12]

Ved at måle på forskellen mellem + og - på wheatstone bridge kan forskellen i spændingen måles. Denne forskel er ikke særlig stor, ud fra hvor meget kraft man putter på load cellen. Derfor ved hjælp af en amplifier kan signalet/forskellen i spændingen forstørres og sendes ud som signal. Amplificeren, som blev anvendt, er en AD620, hvor kredsløbet kan ses nedenfor på figur 7.

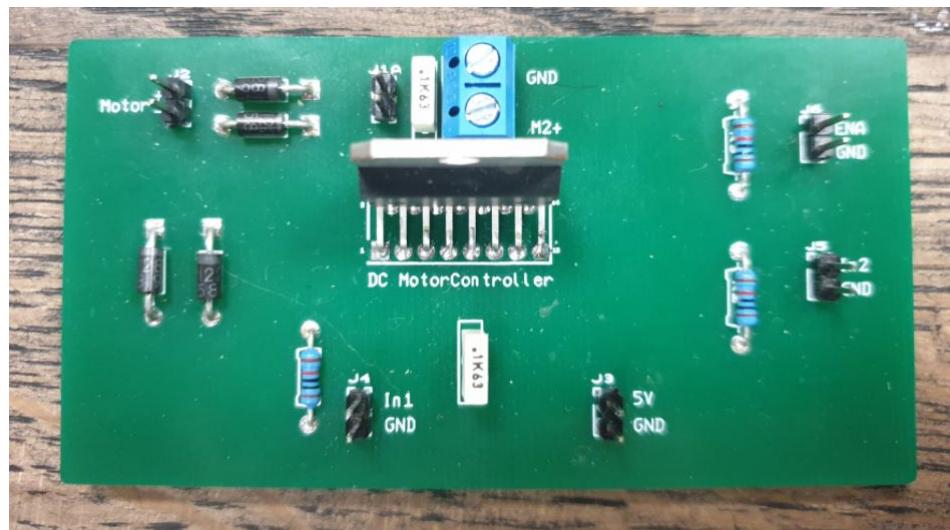


Figur 7: Kredsløb for load cell amplifier [9]

2 Hardware Implementering

2.1 Køretøj

Efter at have lavet en simpel test på det udleverede køretøj fandt vi ud af, at larvefødderne kørte invers af hinanden. Derfor blev der lavet en implementation med to H-broer[11] så man kunne vende polariteten af strømmen på DC-motorerne. Derved ville det være muligt at få DC-motor til at køre i samme retning. De udleverede H-bro prints fra GFV, som kan ses på figur 10, blev brugt, da det ikke ville være relevant eller tidsmæssigt forsvarligt at lave det selv i netop dette projekt. Man kan se i kredsløbsdiagrammet[8], at kredsløbet skal forsynes med 5V. Kredsløbet er bygget op omkring L298, som er en Dual Full Bridge Driver. Hvis man kigger i databladet[3] på figur 9, kan man se at L298 kan forsynes med op til 50V.



Figur 8: H-Bro print

L298

ABSOLUTE MAXIMUM RATINGS

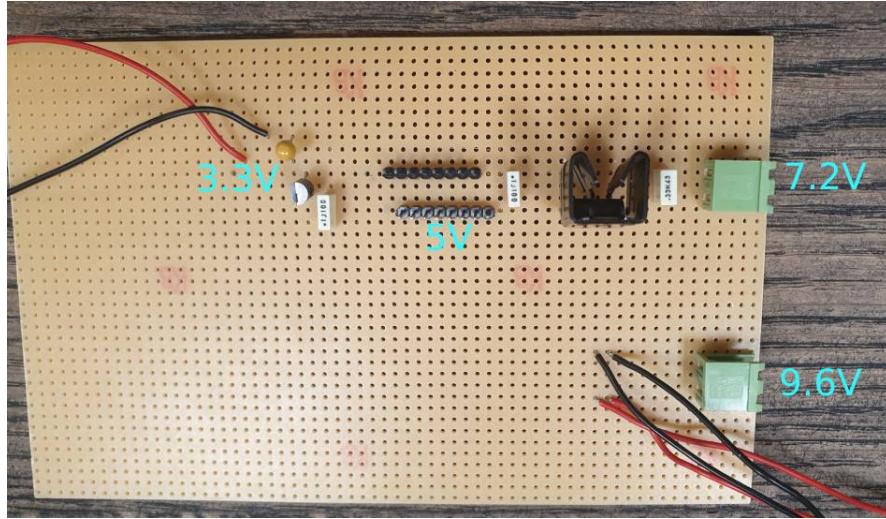
Symbol	Parameter	Value	Unit
V_S	Power Supply	50	V
V_{SS}	Logic Supply Voltage	7	V
V_I, V_{EN}	Input and Enable Voltage	-0.3 to 7	V
I_O	Peak Output Current (each Channel) - Non Repetitive ($t = 100\mu s$) - Repetitive (80% on -20% off; $t_{on} = 10ms$) - DC Operation	3 2.5 2	A
V_{SENS}	Sensing Voltage	-1 to 2.3	V
P_{TOT}	Total Power Dissipation ($T_{case} = 75^\circ C$)	25	W
T_{OP}	Junction Operating Temperature	-25 to 130	$^\circ C$
T_{STG}, T_J	Storage and Junction Temperature	-40 to 150	$^\circ C$

Figur 9: L298 Dual Full Bridge[3]

2.2 Batteri og Spændingsregulator (PSU)

Kredsløbet til batteriet blev bygget på et breadboard. Her blev der sat to batteriterminaler på, så batterierne nemt kunne tilsluttes kredsløbet. På udgangen af 9.6V batteriterminalen blev der sat to røde ledninger (9.6V) og to sorte ledninger (GND). Disse vil blive brugt som forsyning til H-broerne og altså derved DC-motorerne. På den anden udgang blev der først sat LM7805 på for at regulere forsyningsspændingen på 7.2V om til 5V. Hernæst blev der sat 8 Harwin pins til 5V og 8 til ground,

så alle de dele af systemet, der skulle have 5V, kunne forsynes. Dernæst blev LM1085 sat på for at regulere spændingen på 5V om til 3.3V, og satte en rød ledning (3.3V) og en sort ledning (GND) på udgangen. Denne vil blive brugt til at forsyne Raspberry Pi i systemet. Billede af breadboard med alle komponenter kan ses på nedenstående billede og komponentliste kan ses på tabel 1



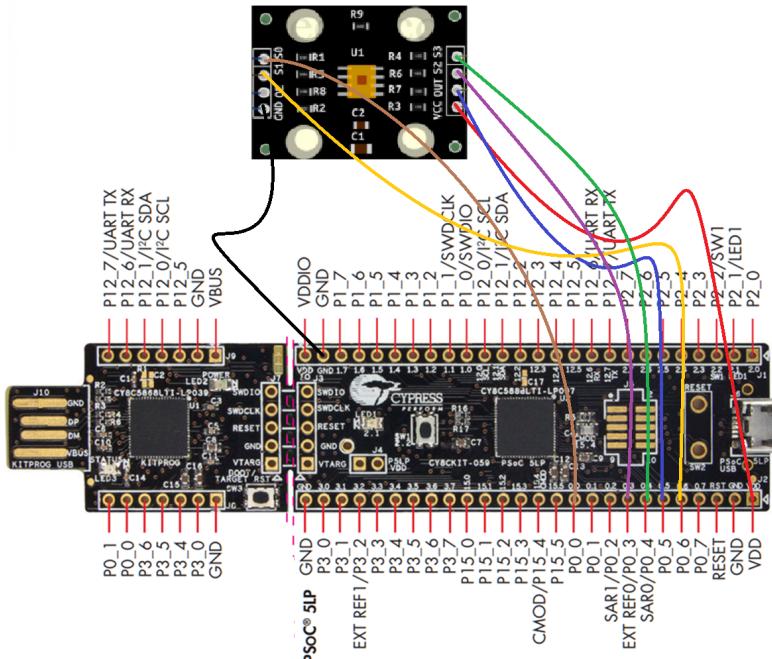
Figur 10: Breadboard med batteriterminaler og Spændingsregulator

Komponent	Beskrivelse
1 stk. LM7805	Spændingsregulator 5V
1 stk. LM1085	Spændingsregulator 3.3V
1 stk. Kølelegeme	Køler til LM7805
2 stk. Batteriterminal	Terminal til 9.6V og 7.2V batteri
2 stk. $10 \mu F$ kondensator	Reducerer støj
1 stk. $0.33 \mu F$ kondensator	Reducerer støj
1 stk. $0.1 \mu F$ kondensator	Reducerer støj
16 stk. Harwin pins	Til at tilslutte ledninger fra system
Diverse ledninger	Til at tilslutte forsyning til system

Tabel 1: Komponentliste for Batteritilslutning og Spændingsregulator

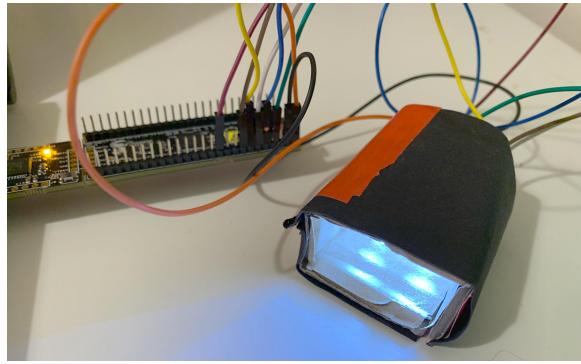
2.3 Farvesensor

Som tideligere nævnt har farvesensoren otte pins. Disse pins samt deres forbindelser kan ses på figur 11. Farvesensoren forsynes med 5V via PSoC'en. VCC forbindes derfor med en rød ledning til forsyningen på breadboardet. GND forbindes til PSoC'ens GND via en sort ledning. På nedenstående figur 11 vises hvorledes farvesensoren forbindes til PSoC, når der skal laves modultest



Figur 11: Farvesensor TCS230 forbindelse til PSoC

De resterende signaler bestående af S0, S1, S2, S3 og OUT forbindes også til PSoC'en med forskellig farvede ledninger, alt efter hvad pin numrene er sat til i PSoC projektet. Farvesensoren pakkes ind i en papkasse, så omkringliggende lysforstyrrelse ikke vil påvirke aflæsningerne.



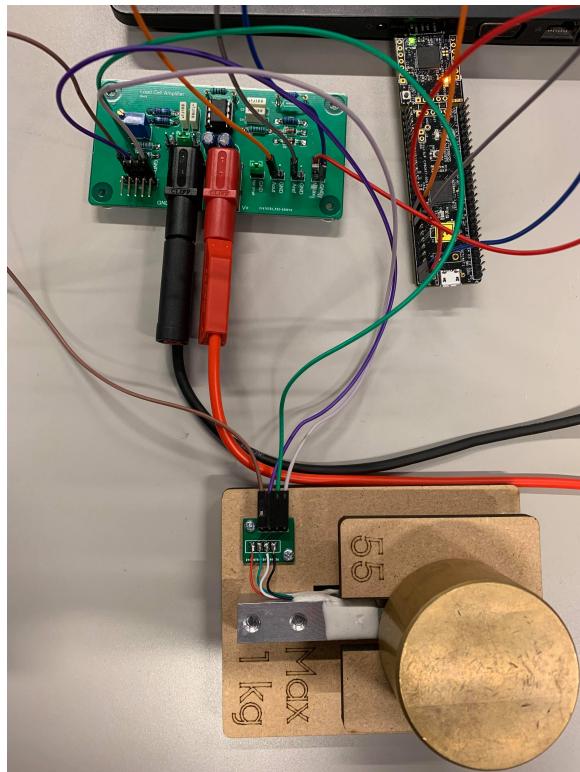
Figur 12: Color sensor inden i papkasse

På denne måde vil farvesensoren lyse direkte ned og detektere den farve, der ligger under den. Konsekvensen af ikke at inkludere papkassen er, at farvesensoren let kan blive forstyrret hvis der er lidt skygge på den ene side, hvilket kan resultere i fejl under aflæsning. Dog er farvesensoren kalibreret til at den kan aflæse farver i forskellige lysomgivelser .

Farvesensoren kan opfange fire farver: Rød, blå, grøn og clear. Vi anvender derfor farvet gaffatape til at markere banen. Den røde farve repræsenterer banen og den grønne farve repræsenterer bordene og slutposition. Den blå farve kommer til at ligge til højre for den røde tape. Selve kørebanen kommer derfor til at bestå af tre farver: rød, blå og unknown. (Den røde farve skal sørge for at bilen kører lige. Når bilen begynder at køre til højre, detekteres den blå farve, som skal sørge for at bilen drejer til venstre indtil den rammer rød igen. Ligeledes med unknown, blot at den skal sørge for at bilen drejer til højre, indtil den rammer rød.

2.4 Vægt sensor

For vægtsensor blev der forsynet med 5V via spændingsregulator med en tilhørende ground forbindelse til load cell amplifieren. Load cell amplifieren giver så spændingen videre til load cell. Load cell har så S+ og S- hvilket er spændingsforskellen som amplificeres. Efter signalet blev forstærket sendes signalet videre til vores PSoC ved hjælp af Aout pin. Aref pin bliver anvendt som ADC reference, der er forbundet imellem Load cell amplifier og PSoC. Til at beskytte input pin på PSoC bliver der anvendt en 5V pin, som er forbundet imellem amplifier og PSoC, hvor der også er tilsluttet en tilhørende fælles ground imellem load cell amplifier og PSoC. Vægt sensoren blev kalibreret i en tidligere GFV øvelse, hvor load cell amplifieren blev indstillet, og der blev fundet den generelle funktion som konverterer ADC signalet om til vægten i kg. Kalibreringen kan ses i nedenstående figur



Figur 13: Vægtsensor Kalibrering

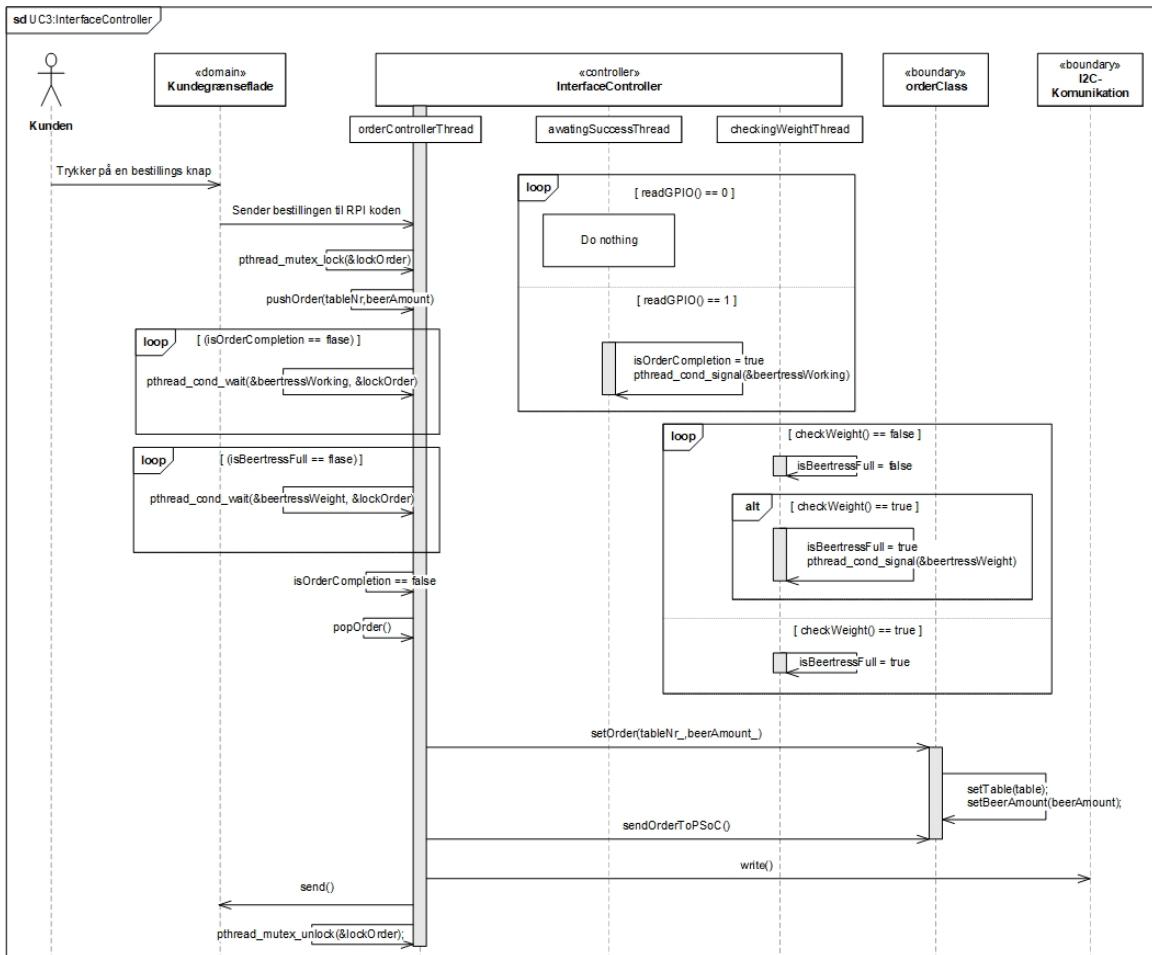
3 Software Design

3.1 InterfaceController_SW

Nedestående kan der ses et overblik over designet af software til InterfaceController_SW (RPI'en).

Diagrammet vil blive gennemgået længere nede.

3.1.1 Sekvensdiagram



Figur 14: Overbiks sekvensdiagram for main rpi kode

Eftersom at programmet skal kunne håndtere flere ting af gangen er det vigtigt, at det bliver lavet som et flere trådet program. Før vi uddyber hvad hver tråd håndterer, skal isOrderCompletion og isBeertressFull forklaries. isOrderCompletion er en boolean, som viser om beertræs er færdig med en ordre eller igang med en. Vores to tråde, ”orderControllerThread” og ”awatingSuccessThread” har begge adgang til denne boolean og bruger den til at vide, hvornår de skal arbejde og hvornår de skal vente. isBeertressFull er en boolean, som viser, om der er placeret en øl på vægten. Denne boolean bliver benyttet i ”orderControllerThread” og ”checkingWeightThread”, men det er kun ”checkingWeightThread” der ændre på dens værdi og derved bestemmer hvornår ”orderControllerT-

hread” må arbejde.

isOrderCompletion = false	Beertress er i gang med at udføre en ordre og kan derfor ikke modtage en ny ordre før den er færdig.
isOrderCompletion = true	Beertress er færdig med at udføre en ordre og venter derfor på en ny ordre.
isBeertressFull = false	Der mangler øl på Beertress. Beertress venter indtil der blevet placeret en/flere øl på dens vægt.
isBeertressFull = true	Der er påfyldt øl på Beertress og er klar til at gøre til næste ordre.

Vores awatingSuccessThread sørger for at konstant køre readGPIO funktionen. Dette er en function, som åbner en local driver, som tillader os at aflæse en fysik pin på vores RPi, der er direkte forbundet til PSoC'en. Dette er lavet til at når Beertress er færdig med en ordre, sender den et signal ud, som er forbundet med vores fysiske pin. Når dette signal bliver 1 sætter vores tråd isOrderCompletion = true og signalerer til vores orderControllerThread, at den gerne må sende en ordre videre til PSoC'en. Vores orderControllerThread sørger for at modtage den data, der bliver modtaget fra kundegrænsefladen og putter det ind i en kø, hvorefter den tjekker om isOrderCompletion er true eller false. Hvis den som sagt er false sætter den sig til at vente på at awatingSuccessThread signalerer den til at arbejde videre. Når isOrderCompletion bliver true, bliver den sat til false igen kort efter, for at sikre at næste gang vores tråd køre igennem loopet, så vil den vente og sikre at beertrees ikke er igang med en ordre før den sender en ny. Tråden tager nu den første ordre i vores kø og sender den til vores orderClass, som sætter de forskellige variabler, i forhold til hvad der er bestilt. Derefter bliver sendOrderToPSoC() kaldt, som får vores class til at sende orden til vores PSoC. Dette bliver gjort igennem I2C, som vil blive forklaret yderligere i et senere afsnit. Når orden er blevet sendt over I2C, vil der blive sendt en besked tilbage til kundegrænsefladen, at orden er sendt afsted til Beertress.

3.1.2 Kundegrænseflade

Når der skal designes en brugergrænseflade, er det vigtigt at starte med en skitse, så man kan danne sig et billede af, hvordan det færdige resultat vil komme til at se ud. På figur 15 ses en skitse, som blev konstrueret, inden selve implementationen gik i gang.

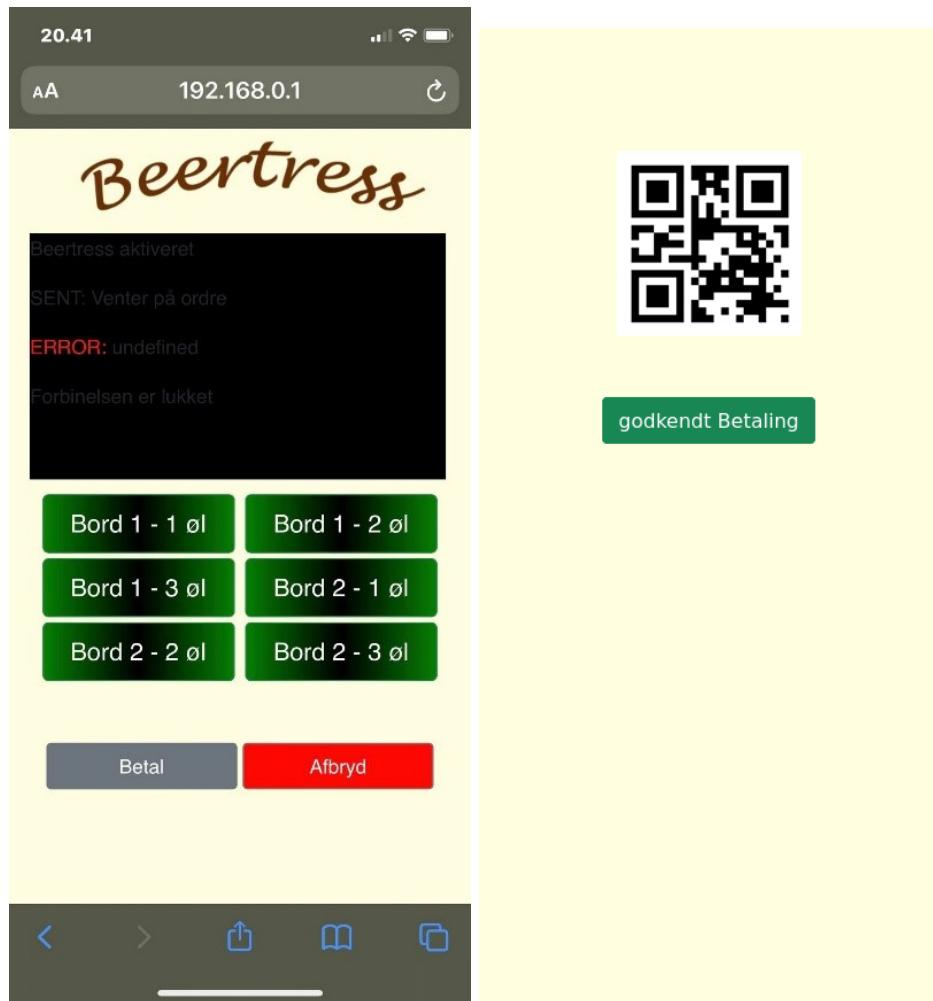
Som kunde skulle det være muligt at bestille service igennem en app eller en brugergrænseflade, monteret ved det bord man sad ved. Her skulle det være muligt, også at fortælle Beertress, hvor

mange genstande, man ønskede. Til at starte med valgte vi at lave en dropdown menu, hvor man kunne vælge antal øl.



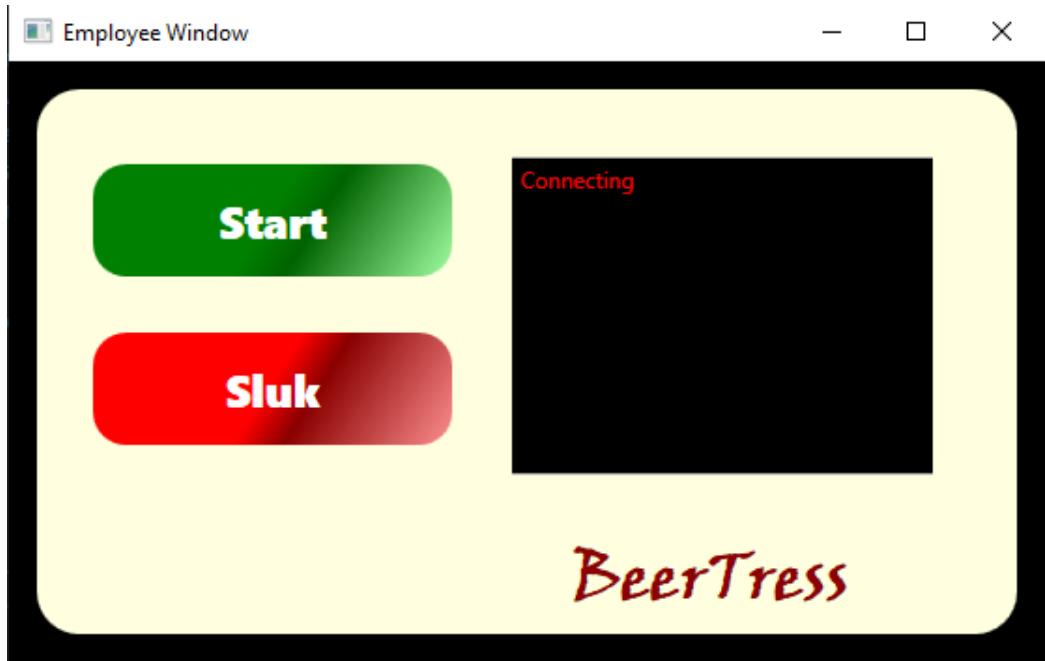
Figur 15: Skiste af brugergrænse flade

Dette udgangspunkt blev dog ændret til følgende design, som ses på figur 16.



Figur 16: Hjemmeside med QR kode

3.2 WaiterApp_SW



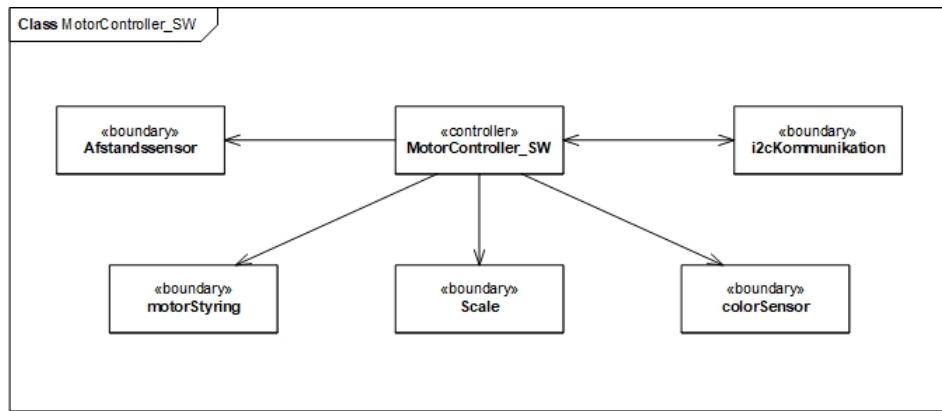
Figur 17: Personalegrænseflade design

Personale-brugerfladen er lavet som en wpf .net framework Desctop App. Designet er lavet så det er simpelt og enkelt for de ansatte at bruge. Den består af en grøn knap, som starter beertress og en rød, som slukker. På højre side ses en sort skærm, hvor der bliver udskrevet, hvor i processen beertress befinder sig. Det hele er styled med Xaml, som er WPF's alternativ for css.

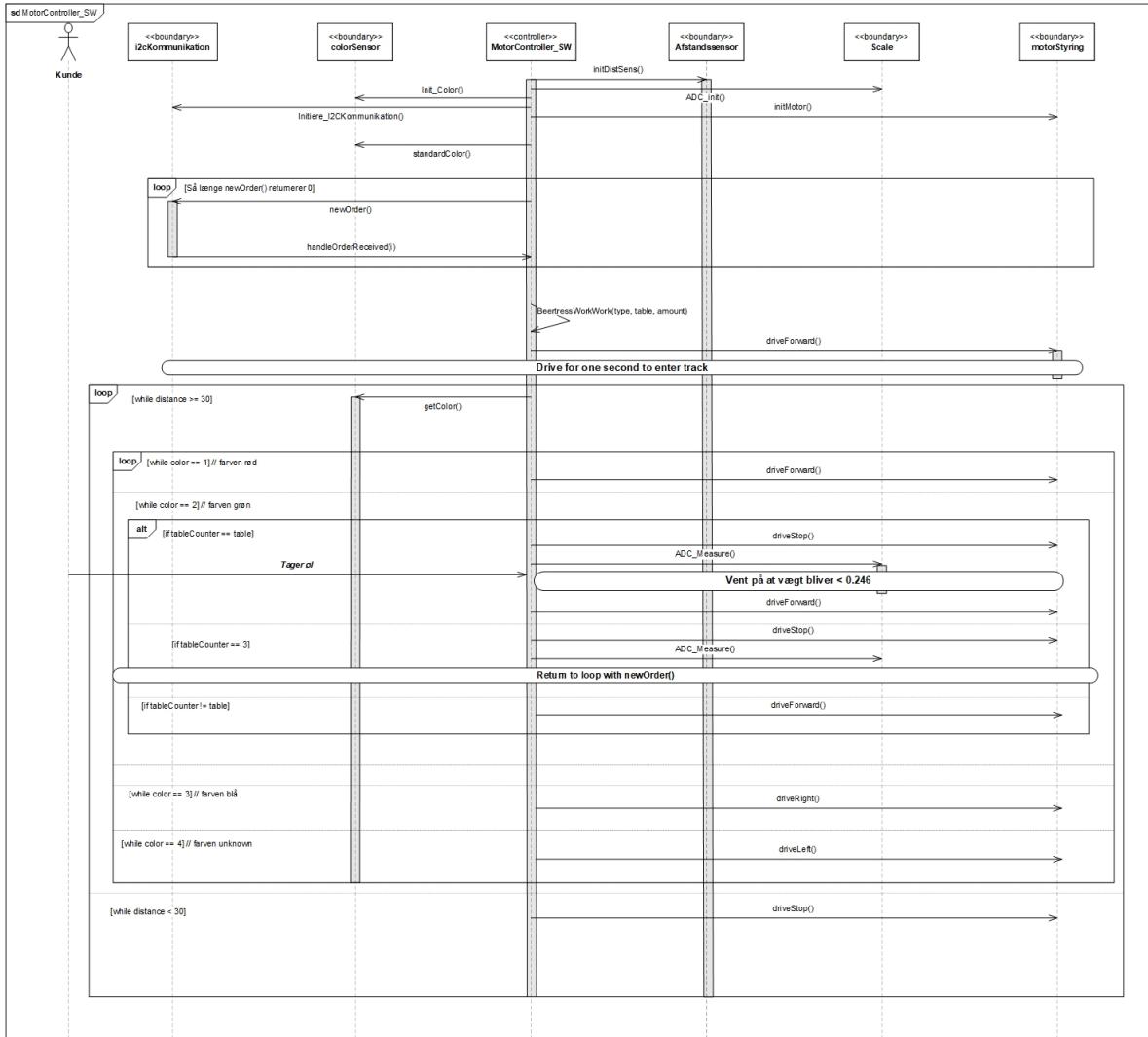
3.3 MotorController_SW

Figur 18, 19 og 20 viser designet af MotorController_SW, og mere bestemt hvordan alle sensorerne snakker sammen, når Beertress får en bestilling.

Med udgangspunkt i arkitekturen, er der blevet lavet et design af softwaren, hvor kommunikationen mellem komponenterne er forsøgt tydeliggjort. Hver sensor har fået en boundary, da de er interfacet ud til omverdenen. De nedenstående diagrammer er altså mest af alt en beskrivelse af, hvordan MotorController_SW indtager og leverer en bestilling. Alt koden er håndteret i PSoC Creator.



Figur 18: Klassediagram af MotorController_SW

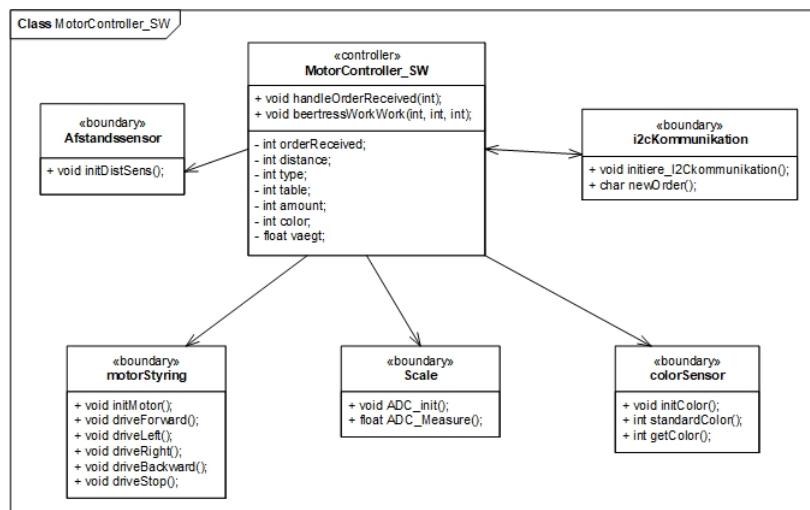


Figur 19: Sekvensdiagram af MotorController_SW

Som det kan ses på figur 19, starter det hele fra vores controller, MotorController_SW, som initierer alle sensorerne og motoren. Nogle af disse sensorer, f.eks. afstandssensoren, består af funktioner som ikke fremstår af sekvensdiagrammet. Disse bliver beskrevet i implementeringsafsnittet. MotorController_SW venter på en ordre fra RPi inde i en for-løkke, og bliver i denne løkke så længe ingen ordre er modtaget. Når en ordre modtages, bliver denne ordre først håndteret inde i BeertressWorkWork(), der tager tre input parametre - en for typen af drikkevare, en for bordnummer og en for antal øl bestilt. Ved hjælp af farvesensoren navigerer Beertress derefter afsted mod det ønskede bord, og hhv.

drejer eller køre ligeud alt efter hvilken farve underlaget har.

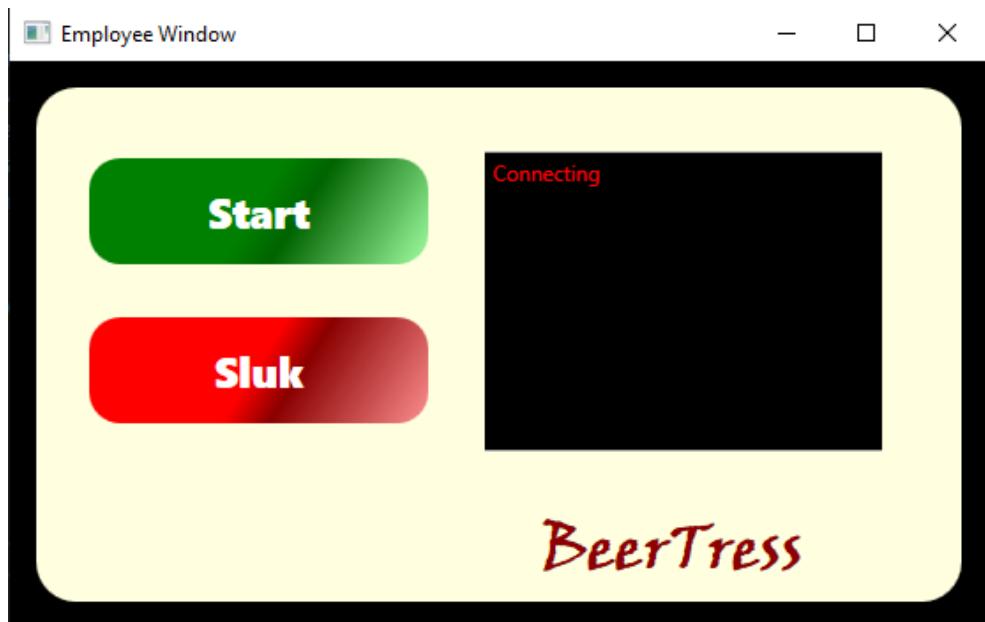
Når den grønne farve rammes, kan der enten køres videre ligeud og en variabel, tableCounter, tælles op, eller også rammes det ønskede bord. Når det ønskede bord rammes stopper Beertress, og kunden tager det ønskede antal øl. Efter en nedskalering af Beertress, indeholder Beertress en dåseøl på 33 cl, og når denne øl fjernes, kommer vægten under 0.246kg, og Beertress kører videre igen. Til slut kan man ramme start, hvilket er når tableCounter er = 3. I dette tilfælde stopper Beertress og noterer, om der skal påfyldes mere øl. Under hele denne proces står afstandssensoren og noterer om et objekt er kommet ind foran sensoren. I så fald vil Beertress stoppe. I implementeringsafsnittet vil funktionerne beskrives yderligere.



Figur 20: Klassediagram med funktioner

4 Software Implementering

4.1 WaiterApp_SW



Figur 21: Personalegrænseflade design

Personalegrænseflade er lavet i en asp.net 3.1 Windows-Applikation. En af fordelene ved at benytte Microsoft Asp.net framework til personalets brugergrænseflade (figur15), er at man får stillet nogle værktøjer til rådighed, som gør implementeringen nemmere. Når man fx arbejder med websockets, kan der hentes ekstra pakker ind i projektet fra Microsoft NuGet, med indbyggede funktioner, der blot skal implementeres. I dette tilfælde kunne Nuget-pakker som System.Net.WebSockets(4.3.0) og Websocket.Client(4.3.21). hentes ind. Med de to pakker gjorde det arbejdet med at lave forbindelsen nemmere. Softwaren i bruger-grænsefladerne består af c#,C++, css, bootstrap, og html. Valget af at benytte .net desktop App er udelukkende fortaget, fordi vi på det tidspunkt ikke havde kendskab til, de muligheder, som findes. At bruge en Windows App, som kører på en pc er en fin løsning, men det ville være smartere at have brugerflade, som ligger online på en server i stedet.

4.2 InterfaceController_SW

I dette afsnit vil vi komme ind på hvordan InterfaceController_SW koden er blevet implementeret, samt hvordan vi har opbygget en driver der kan åbne og aflæse en fysisk pin på RPI'en. Der vil også blive forklaret hvordan RPI'en er sat op til at køre programmet ved opstart, så beertress programmet bliver kørt i det der bliver sat strøm til.

Som gennemgået i design sektionen skal vores RPi kode kunne håndtere sending og modtagning af char beskeder fra kundegrænsefladen. Denne besked bliver kopiret til en string for at benytte nogle af de funktioner, der følger med string libary'et. Her bliver der søgt igennem beskeden efter hvilket bord nr og antal øl, der er bestilt til. Disse information bliver herefter gemt, så de kan blive sendt til en kø. Her er blevet brugt et queue libary. Bestillings informationerne kan nu blive hentet igen, når programmet er klar til at sende en ordre afsted til vores PSoC.

Når en ordre skal sendes via I2C til PSoC'en, som er vores slave, skal nogle bestemte funktionskald først laves. Eftersom linux på forhånd har en I2C Driver, vil denne blive benyttet i projektet. Først skal denne driver dog åbnes, hvilket sker via en open() funktion, og dernæst skal masteren, altså RPI'en, vide, hvilken slave adresse den skal sende til. Dette gøres via et ioctl() kald. I vores tilfælde er slave adressen 0x08. Når disse to er sat rigtigt op, skal der blot kaldes en write() funktion, som skal sende 3 bytes af en buffer, eftersom protokollen er lavet således. Informationerne om indholdet af disse tre bytes er givet i den besked, som RPI'en modtager fra kundegrænsefladen, og bliver sat på hhv. plads 1, 2 og 3 i bufferen.

På samme måde kan read() også kaldes. Her kræves det blot en read buffer, som kan gemme de informationer. som skal læses. Dette bliver brugt til at tjekke på, om der skal genopfyldes.

For at vide hvornår PSoC'en er klar til at modtage en ny ordre, bliver RPI'en og PSoC'en forbundet med en ledning, hvor PSoC'en vil sende et signal til RPI'en, når den er klar til at modtage en ny ordre. For at opnå dette er der blevet bygget en char driver på RPI'en. Denne driver vil, når den bliver kaldt, oprette en forbindelse til GPIO 25. GPIO står for "general-purpose input/output", det vil sige, at vi kan sætte denne pin til at være enten et input eller et output. Vores driver sørger for, at den bliver sat til at være input. Når der skal aflæse på den fysik pin sørger vores driver for at finde dens værdi og sende den videre til vores InterfaceController_SW kode, i form af et 1 eller 0. Siden at alt på en RPI arbejder i filer så for at benytte denne driver skal vores InterfaceController_SW selv

åbne driveren. Til dette formål er det blevet oprettet en funktion, `readGPIO()`, i `InterfaceController_SW`. Denne funktion har til formål at kalde en open funktion til den filplacering vores driver har, i dette tilfælge er det `/dev/orderCompleteGPIO`. Når driver filen nu er åben, har vi mulighed for at benytte dens read funktion, som aflæser værdien på den fysiske pin. Denne værdi bliver gemt, hvorefter at driveren nu bliver lukket.

RPi'en skal også kunne modtage og håndtere data sent fra PSoC'en angående den implementeret vægt. Igennem vores implementerings fase er vi kommet frem til, at den prototype vi bygger ikke vil kunne håndtere en fusage, så derfor er fustagen blevet udskiftet med dåse øl. Derfor er vi kommet frem til at PSoC'en nu vil sende et 1 over I2C til RPi'en, når der ikke længere er øl på vægten. Dette indebærer, at `InterfaceController_SW` koden indeholder en function kaldt `"checkWeight()"` i klassen `orderClass`, der aflæser på I2C og returnere et true, hvis der mangler øl på beertress eller et false, hvis der stadig er øl på beertress.

For at RPi'en skal kunne håndtere ordre samt at aflæse dens GPIO pin og læse på I2C, har det været nødvendigt at lavet det som et flere trådet system. Når programmet bliver startet op bliver 3 tråde oprettet:

<code>orderControllerThread</code>	Denne tråd sørger for at håndtere største delen af vores program: Aftyde beskeder fra hjemmeside, opbevaring af bestillinger, sende bestillinger til PSoC og sende en godkendelse besked til hjemmesiden.
<code>awatingSuccessThread</code>	Denne tråd sørger for at aflæse på vores GPIO pin om vi får sendt et 1 eller 0. Denne værdi bliver så brugt til at bestemme der må blive sendt en ny ordre eller om der skal ventes.
<code>checkingWeightThread</code>	Denne tråd sørger for at aflæse på I2C om vi modtager et 1 eller 0. Denne værdi bliver så brugt til at bestemme der må blive sendt en ny ordre eller om der skal ventes.

I programmet der er oprette to globale boolean som begge har indflydelse på hvornår `orderControllerThread` skal arbejde og hvornår den skal vente.

isOrderCompletion = false	Beertress er i gang med at udføre en ordre og orderControllerThread skal derfor ikke sende en ny ordre før den er færdig. orderControllerThread bliver sat til at vente.
isOrderCompletion = true	Beertress er færdig med at udføre en ordre og orderControllerThread får nu tillades til at arbejde videre.
isBeertressFull = false	Beertress mangler øl på vægten. orderControllerThread bliver sat til at vente indtil der bliver placeret en øl på vægten.
isBeertressFull = true	Beertress mangler ikke øl på vægten. orderControllerThread får nu tillades til at arbejde videre.

"isOrderCompletion" bliver styret af både "awatingSuccessThread" og "orderControllerThread". Som start er "isOrderCompletion" sat til at være true så "orderControllerThread" kan få lov til at sende den første ordre. Efter der er tjekket om "isOrderCompletion = true", sætter "orderControllerThread" den til false. Dette sikrer at næste gang den vil til at sende en ordre skal den vente indtil "awatingSuccessThread" har sat "isOrderCompletion" til true igen. "awatingSuccessThread" er sat at køre "awatingSuccessFunction" som konstant tjekker om readGPIO() funktionen returnerer et 1 eller 0. Hvis der bliver returneret et 1 vil "isOrderCompletion" sat til true og der vil blive signaleret til "orderControllerThread" at den må arbejde videre igen.

"isBeertressFull" bliver kun styret af "checkingWeightThread". "checkingWeightThread" er sat til at køre funktionen "checkingWeightFunction", som konstant tjekker "checkWeight()" funktionen, og ser om der bliver returneret et true eller false. Hvis der bliver returneret false vil "isBeertressFull" sat til false, som gør "orderControllerThread" vil blive sat til at vente, fordi der ikke længere er øl på vægten. Hvis der bliver returneret true vil "isBeertressFull" blive sat til true som resultere i at "orderControllerThread" får lov til at arbejde videre fordi der er øl på vægten.

Når "orderControllerThread" får lov til at arbejde videre efter de ovenstående krav, vil de tage den forreste bestillingen i køen og sende den over I2C til PSoC'en. Her efter vil den sende en godkendelse besked tilbage til hjemmesiden der skriver hvilken bestilling der er sendt til PSoC'en.

For at få vores program til at køre når RPi'en tilsluttet til strøm, bliver der oprettet et script, som står for at opsætte og indsætte vores GPIO driver og derefter gøre vores program. Dette script er

blevet sat til at køre når RPi'en tænder.

4.2.1 Websocket

Kommunikationen imellem Kunderne og Beertress forgår igennem en Websocket forbindelse. Der findes også det der kaldes Signal-R, som er en abstraktion over det, som er specifikt for C-Sharp og Asp.net, hvor Websocket er den generelle standart, som fungerer på alle servere.



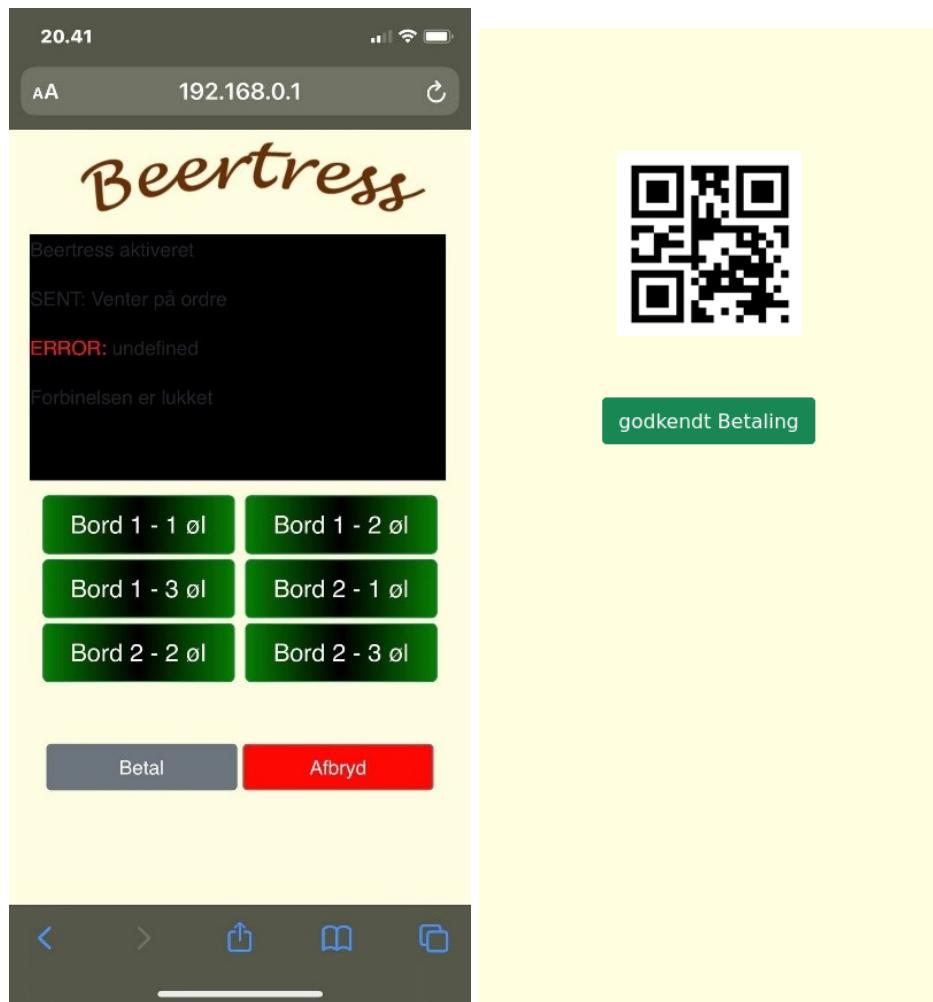
Figur 22: Websocket

Websocket giver adgang til fuld-dublex kommunikation kanaler over en single TCP-forbindelse. Protokollen blev standardiseret af IETF som RFC 6455 i 2011. W3C. Websocket er designet til servere og webbrowsere, men kan bruges alle steder, hvor der findes en klient eller server applikation. Default porten for kommunikationen er TCP port nummer 80. Hvis man bruger wss. (Websocket secured) er portnummeret 443 som standard, men ligesom http kan man køre på hvilken som helst port. At vælge et andet portnummer end standart portene, er ikke en god idé, fordi det kan skabe problemer i forhold til at kunne komme igennem firewalls.

4.2.2 Kundegrænseflade

For at gøre kunde-brugergrænsefladen lettest og hurtigst tilgængelig for kunderne, er der lavet en simpel html hjemmeside, der fungerer som en App. Som kunde skal det nemlig være enkelt og

hurtigt at kunne få adgang til systemet og få den ønskede service, uden unødig at interagere med personalet. Når først man har adgang til det netværk, hvor Appen befinner sig, er det nemt at gå ind og bestille. Alternativet ville være, at man lavede en rigtig app som kunderne kunne hente ned, men denne løsning er smartere og hurtigere, fordi de blot skal gå ind på en hjemmeside/IP-adresse, og få sekunder efter kan der fortages en bestilling. Selve stylingen af Appen er gjort ved hjælp af css. For at gøre brugerfladen mobil-venlig, er der brugt Bootstrap 4.0, som automatisk skalerer elementerne, når størrelsen ændres.



Figur 23: Hjemmeside med QR kode

For at nedskallere vores program lidt valgte vi at implementere 6 knapper der styrer, om der bliver

bestilt imellem 1-3 øl til enten bord 1 eller bord 2, afhængig af hvad kunden vælger. Den ideelle løsning havde været at lave en drop down menu eller et tekst felt, hvor kunden selv kan vælge det bord, de sidder ved, eller hvor mange øl de ønsker. For at opnå de ønskede krav til vores kundegrænseflade er der benyttet websocet HTML og CSS. Her er det sat op så, når der bliver trykket på en knap vil der blive kørt websocet funktionen doSend("Bord: " + tableNr + "Antal øl: " + beerAmount). Denne function sørger for at køre funktionen, der skriver den sendte bestilling på skærmen i vores log samt sender char beskeden videre til vores InterfaceController_SW kode.

4.3 MotorController_SW

MotorController_SW initierer motoren og sensorerne, og kalder standardColor funktionen, som er beskrevet i colorSensor afsnittet. Dernæst er der implementeret et uendelig for-loop, som håndterer den char, som newOrder fra i2cKommunikation sender. Denne char bliver så brugt i handleOrderReceived, som håndterer den modtagne char i et switch case statement. Ved hjælp af den udvidede protokol i tabel 3 vides det så, hvad der skal sendes videre til funktionen beertressWorkWork, som indeholder 3 variable, nemlig type, bord og antal.

Den første parameter, type, bliver håndteret i et switch-case statement. Eftersom vi i vores prototype kun har én type drikkevare, nemlig øl, er det kun denne case, der bliver implementeret. I denne case sætter vi en tableCounter til 0, og denne bliver så brugt til at tælle borde op med, hver gang vi passerer en grøn farve med farvesensoren. Når tableCounter bliver lig table parameteren stopper Beertress og venter på at kunde tager sin drikkevare indtil vægten når under 246 gram (for at indikere at der ikke er flere øl på vægten). Når det er sket, bliver table parameteren nulstillet og der køres videre indtil tableCounter bliver 3, da den så vil være ved start-position igen. Her vil der så blive sendt et højt signal til Raspberry Pi via Done pin'en, for at indikere at ordren nu var afsluttet og en read-buffer, som kan læses af Masteren(RPi) bliver sat til 1, hvis vægten indikerer, at der er behov for genopfyldning.

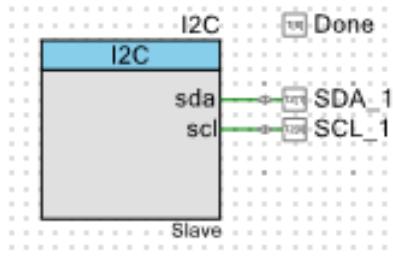
For at håndtere banen har vi implementeret while-loops, så når farvesensoren indikerer rød, skal Beertress køre ligeud, blå er til venstre, "unknown" er til højre, og grøn er til stop. Her er afstandssensoren også implementeret, så hvis denne registrerer noget under 30 centimer foran den, vil Beertress stoppe med at køre, indtil der er fri bane igen. Alt dette er implementeret i beertressWorkWork funktionen.

4.3.1 i2cKommunikation

Ved i2cKommunikation er der taget udgangspunkt i protokollen, og ved hjælp af switch states, er protokollen blevet implementeret.

Koden er skrevet i C, da PSoC Creator ikke understøtter C++, men selve opdelingen er inspireret af C++ med .h og .c filer for at skabe et bedre overblik.

Som figur 24 viser, indeholder topdesignet en I2C slave, som er en komponent inde på PSoC Creator. Slave adressen bliver manuelt tildelt 0x08, som altså er den slave-adresse, masteren skal sende til.



Figur 24: Topdesign af I2C fra PSoC Creator

i2cKommunikation består af følgende funktioner:

Funktion	Beskrivelse
void initiere_I2Ckommunikation()	Initierer I2C slaven, samt klargøre en buffer til at modtage ordre via I2C
char newOrder()	Håndterer den ordre der er modtaget via I2C

Tabel 2: Funktioner i i2cKommunikation

I2C-slave komponenten har nogle indbyggede funktioner, som kan benyttes til at klargøre slaven til at modtage informationer fra Masteren. Disse funktioner er fundet i slavens datablad[2]. Fx bruges funktionen I2C_SlaveInitWriteBuf til at fortælle masteren, hvilken buffer den skal sende til, samt størrelsen på det antal bytes som skal sendes. Denne funktion er placeret inde i ”initiere_I2Ckommunikation”, så der med det samme er mulighed for masteren til at tilgå slaven.

Funktionen newOrder har så til formål, at håndtere det som bliver sendt fra Masteren til slaven i forhold til protokollen. Ved inspiration fra databladet for slaven[2], er det blevet implementeret sådan, at når en succesfuld write dataoverførsel er overstået, altså når Masteren har skrevet til slaven med data uden fejl(dette tjekkes via funktionen I2C_SlaveStatus()), så starter et switch case statement, som håndterer de modtagne bytes. Alt efter hvilke bytes der modtages i forhold til protokollen, returneres en int på 1 til 6, eftersom der er 6 mulige udfald i vores tilfælde. Dette vil kunne skaleres til 255 forskellige cases for hver byte, så der er rigelig plads til skalering. Den int der returneres bliver så brugt i MotorController_SW.

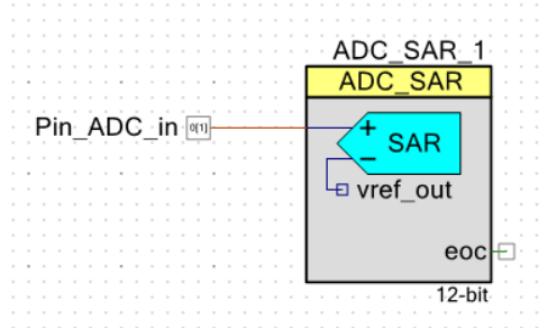
For at undgå fejl er der lavet en kort udvidelse til protokollen mht. retur værdien alt efter hvilken ordre der modtages. Eftersom dette kun er aktuelt i PSoC delen er det ikke blevet indført i den overordnede protokol, da det ikke har noget med RPi'en at gøre:

	Type	Bord	Antal	Retur værdi:
Bord 1, 1 øl:	0b00000001	0b00000001	0b00000001	0x1
Bord 1, 2 øl:	0b00000001	0b00000001	0b00000010	0x2
Bord 1, 3 øl:	0b00000001	0b00000001	0b00000011	0x3
Bord 2, 1 øl:	0b00000001	0b00000010	0b00000001	0x4
Bord 2, 2 øl:	0b00000001	0b00000010	0b00000010	0x5
Bord 2, 3 øl:	0b00000001	0b00000010	0b00000011	0x6

Tabel 3: Protokol med retur-værdi

4.3.2 Scale

Ud fra sekvensdiagrammet på figur ?? og figur 20 skal softwaren have en init funktion og en måling funktion. Koden blev skrevet i PSoC Creator i C. Topdesign som kan ses på figur 25 har en ADC_SAR som har en ADC input. Dette ADC input kommer fra Aout. Vref kommer fra Aref på load cell amplificeren.



Figur 25: Topdesign af ADC fra PSoC Creator

Funktion	Beskrivelse
void ADC_init()	Inititerer ADC_SAR og starter konverteringen så der bliver målt spændingen der kommer ind fra Aout på PIN_ADC_in pin
float ADC_Measure()	Håndtere udregningen af ADC signalet der kommer ind og konverttere det til kg og returnere dette

Tabel 4: Funktioner i Scale

Funktionen ADC_Measure skal håndtere signalet, der kommer ind fra ADC_SAR, når den er færdig med at konverttere en måling. Når den har fået resultatet ind i en uint16_t bliver dette omregnet om til en float ved hjælp af ligningen, der blev fundet ved kalibreringen af vægten. Dette blev fundet i en tidligere undervisnings øvelse og bliver ikke beskrevet her. Implementeringen af ADC_Measure kan ses nedenfor i figur

```

float ADC_Measure()
{
    if (ADC_SAR_1_IsEndConversion(ADC_SAR_1_WAIT_FOR_RESULT))
    {
        uint16_t result = ADC_SAR_1_GetResult16();
        float kg = 0.0005 * result - 0.0553;
        return kg;
    }
    return 0;
}

```

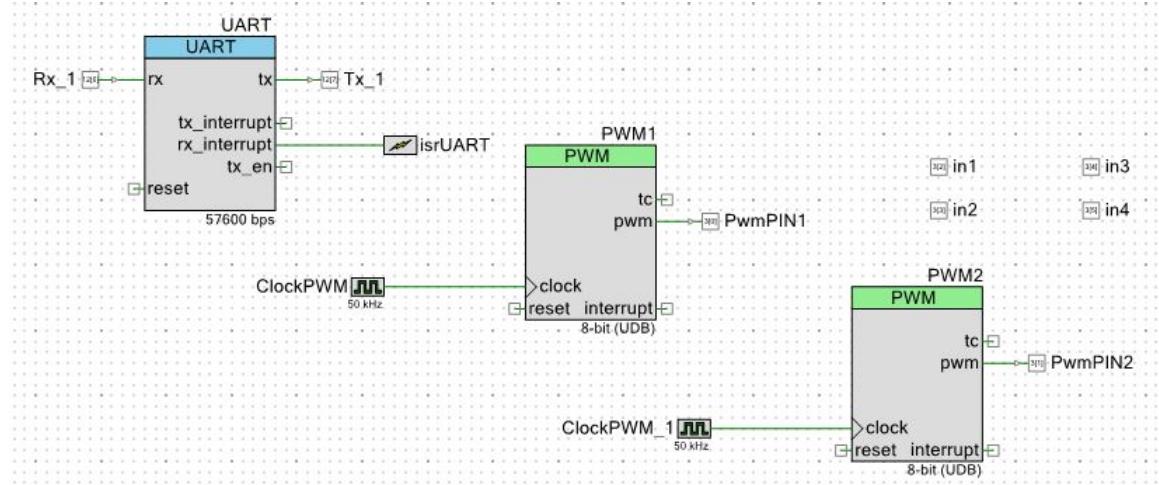
Figur 26: ADC_Measure

4.3.3 motorStyring

Som man kan se i sekvensdiagrammet for MotorControlleren på figur ?? skal motoren kunne køre fremad, til venstre og til højre, men eftersom vi implementerer køretøjet med H-broer, har vi også muligheden for at køre baglæns. Derfor bliver dette også implementeret i motorstyringen, så man fremadrettet kan skalere systemet til at navigere i alle retninger.

Koden til Motorstyring blev skrevet i PSoC Creator i C, da det ikke kan understøtte C++.

Topdesignet indeholder to PWMs, fire direction pins og en UART (som kun vil blive brugt i testøjemed). Hver H-bro vil blive styret af hver sin PWM og to direction pins til at styre retningen af strømmen. Topdesignet kan ses på nedestående billede:



Figur 27: Topdesign af Motorstyring fra PSoC Creator

Motorstyringen indeholder i alt seks funktioner, der blev defineret i en .h fil og implementeret i en .c fil og disse kan ses i nedestående tabel samt en beskrivelse af hver:

Funktion	Beskrivelse
void initMotor()	Initiere begge PWMs
void driveForward()	Sætter direction på pins til fremad og starter PWM med 100 procent på begge PWMs.
void driveStop()	Stopper begge PWMs
void driveLeft()	Starter PWM i venstre side med 100 procent og stopper PWM i højre side
void driveRight()	Starter PWM i højre side med 100 procent og stopper PWM i venstre side
void driveBackward()	Sætter direction på pins til bagud og starter PWM med 100 procent på begge PWMs.

Tabel 5: Funktioner i Motorstyring Software

Nedestående kodeudsnit viser koden til driveForward-funktionen. Samme fremgangsmåde er blevet brugt til de andre funktioner. Her ses det, at PWMs initieres igen. Dette gøres i tilfælde af, at driveStop-funktionen er blevet kaldt og dermed har stoppet PWM. Begge PWMs bliver sat til 100 procent ved hjælp af WriteCompare. H-Bro, der styrer venstre side, får in1 og in2, der bliver sat hhv. høj og lav. Dette gør, at retningen bliver fremadrettet. H-Broen, der styrer højre side, får in3 og in4, og her sker det inverse ift. den venstre H-Bro, så også dens retning bliver fremadrettet.

```

1 void driveForward()
2 {
3     // Saetter direction paa DC motor ved hjælp af H-Broer
4     in1_Write(1);
5     in2_Write(0);
6     in3_Write(0);
7     in4_Write(1);
8     // Starter PWMs
9     PWM1_Start();
10    PWM2_Start();
11    // Saetter PWMs til 100%
12    PWM2_WriteCompare(100);
13    PWM1_WriteCompare(100);

```

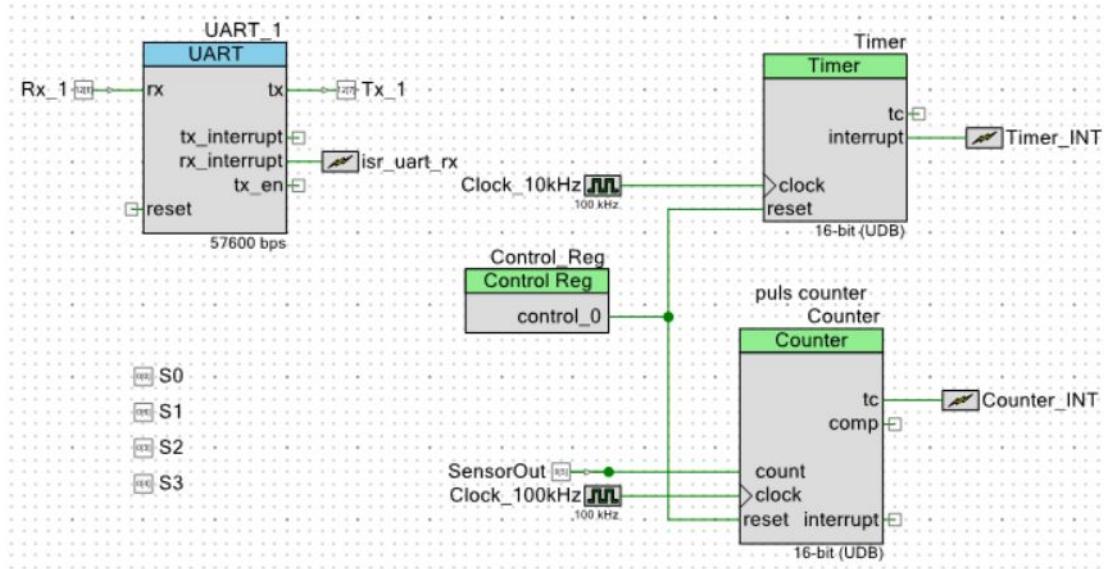
Listing 1: Kodeudsnit for driveForward

Udover disse funktioner blev der initieret en UART. Denne vil blive brugt til at teste alle funktioner enkeltvis i modultesten ved hjælp af switch cases alt efter inputtet fra brugeren. Eftersom denne ikke bliver brugt, andet end til testen, vil den ikke blive beskrevet mere dybdegående her, men den kan ses under bilaget for koden til Motorstyring.

4.3.4 Farvesensor

I dette under afsnit vil farvesensoren blive yderligere beskrevet og implementeringen for de særligt vigtige metoder vil ligeledes blive beskrevet. Det er tidligere nævnt, hvordan denne sensor skal være en del af selve systemet, og hvordan den spiller en væsentlig rolle i funktionaliteten af robotten. Forstået på den måde, at robotten er afhængig af farvesensoren for at kunne navigere rundt på banen.

Softwaredelen for farvesensoren består af en række funktioner, som tilsammen afgør, om der er detekteret nogle af farverne Rød, Grøn eller Blå. Der foretages målinger flere gange i sekundet, som er med til at sikre at robotten kan reagere hurtigt på eventuelle ændringer i farverne. Nedenstående vises topdesignet for colorSensor, bestående af signalerne S0-S3, UART, Timer, Counter og Control Register:



Figur 28: Topdesign af Color Sensor fra PSoC Creator

I nedenstående tabel vil de mest væsentlige funktioner fra colorSensor koden blive præsenteret. Efterfølgende vil disse uddybes og der vises kodeafsnit fra de enkelte funktioner.

Funktion	Beskrivelse
void initColor()	Initierer Counter og Timer
void read_color(void)	Omdanner input til frekvens gennem Counter og Timer
int get_freq(char color)	Her defineres signalerne S0-S3 og farvefiltrene tilføjes
int standardColor()	Aflæser startfrekvensen af hver farve via getFreq()
int getColor()	Aflæser den farve, som sensoren detekterer

Tabel 6: Funktioner i Color Sensor Software

initColor():

Til at initiere colorSensor-programmet dannes en 'initColor'-funktion. Denne har til formål at gøre timer og counter klar til at blive brugt. Interrupt service routinen for henholdsvis timer og counter tilføjes til begge dele. Denne funktion skal kun kaldes en enkelt gang i begyndelsen af main, så man sikrer at de efterfølgende funktioner fra colorSensor-programmet virker.

```
1 void initColor(void)//Starter de indbyggede funktioner fra topdesign: UART, Timer,
    Counter
```

```

2 {
3     CyGlobalIntEnable;
4     UART_1_Start();
5     Timer_INT_StartEx(timer_isr);
6     Counter_INT_StartEx(counter_isr);
7 }
```

Listing 2: Kodeudsnit for initColor()

read_color(void):

Funktionen er med til at omdanne inputs til frekvenser, hvilket gøres via counter og timer. Denne funktion indeholder outputværdier samt opstart af counter og timer.

```

1
2 void read_color(void) //Omdanner inputs til frekvens vha. counter og timeren
3 {
4     Control_Reg_Write(1);
5     CyDelay(1);
6     Control_Reg_Write(0);
7     dataReady = 0;
8     Counter_cycle = 0;
9     Counter_Start();
10    Timer_Start();
11 }
```

Listing 3: Kodeudsnit for read_color(*void*)

Denne funktion skal kaldes inden man går videre til at finde selve frekvenserne. Frekvenserne findes ved hjælp af funktionen get_freq(char color).

get_freq(char color):

I denne funktion defineres output frekvens skaling, som kontrolleres af S0 og S1, samt fotodiode typerne kontrolleret af logic inputs, S2 og S3. Alt afhængig af om S0 og S1 er HIGH eller LOW kan man bestemme output frekvensskaleringen, som kan være på 100%, 20%, 2% eller power down. I datasheetet for TCS230 gives en tabel over, hvad signalerne skal være. [6]

S0	S1	OUTPUT FREQUENCY SCALING (f_o)	S2	S3	PHOTODIODE TYPE
L	L	Power down	L	L	Red
L	H	2%		H	Blue
H	L	20%		L	Clear (no filter)
H	H	100%		H	Green

Figur 29: Tabel af signal options fra datasheet

I dette tilfælde er S0=0 og S1=1, hvilket vil sige, at output frekvensens skaleringen er på 2%. Dette vil sige, at sensoren mäter frekvensen af en bestemt farve, og udskriver 2% af frekvens-værdien. Således har vi at gøre med lavere værdier, som er nemmere at arbejde med. Det ville ikke have gjort den store forskel, hvis skaleringen var sat op på 20% eller 100%. Der arbejdes blot med større tal. Dette er blot for overblikket. Derudover tilføjes farvefiltrene ved at definere 'color' for r,b,g og c. Dette gøres via en if-sætning, hvor S2 og S3 defineres på fire forskellige måder. Eksempelvis når det røde filter skal tilføjes, sættes S2=0 og S3=0. Ligeledes gøres det for de andre filtre.

```

1 int get_freq(char color)//Modtager input p baggrund af indstillinger for
2 //farvefilter samt scaling. Der er valgt at nedskalere frekvens til 2%
3 {
4     //scaling 2\%
5     S0_Write(0);
6     S1_Write(1);
7
8     if(color=='r')//R d farve-filter tilf jes og frekvensen registeres
9     {
10         S2_Write(0);
11         S3_Write(0);
12         read_color();
13     }
14
15     else if(color=='g')//Gr n farve-filter tilf jes og frekvensen registeres
16     {
17         S2_Write(1);
18         S3_Write(1);
19         read_color();
20     }
21
22     else if(color=='b')//Bl farve-filter tilf jes og frekvensen registeres
23     {
24         S2_Write(0);
25     }

```

```

24     S3_Write(1);
25     read_color();
26 }
27 else if(color=='c')//Clear' farve-filter tilfjes og frekvensen registeres
28 {
29     S2_Write(1);
30     S3_Write(0);
31     read_color();
32 }
33
34 while(!dataReady)
35 {
36 ;
37 }
38 return freq;
39 }
```

Listing 4: Kodeudsnit for $get_freq(char color)$

standardColor():

En af de vigtigste funktioner er standardColor(), hvis eneste funktion er at registrere frekvenserne for de tre farver, når robotten skal køre sin opstart, og er placeret på en neutral overflade. Dette er betydeligt for funktionaliteten af resten af programmet, da disse værdier bliver brugt til at registrere ændringer i frekvenserne. Et konsekvens af ikke at implementere dette er, at hvis sensoren registrerer rød som startfrekvens, vil frekvensen for rød ikke stige, og dermed vil bilen ikke køre.

```

1 int standardColor()
2 {
3     read_color();
4
5     freqR_0 = get_freq('r');
6     freqG_0 = get_freq('g');
7     freqB_0 = get_freq('b');
8
9     return 0;
10 }
```

Listing 5: Kodeudsnit for standardColor()

I ovenstående kodeudsnit kan det ses, at funktionen starter med et kald til read_color(), hvorefter frekvensen for de tre farver gemmes i 3 globale variabler af typen unsigned int (uint). Disse variabler

skal bruges i programmets getColor()-funktion.

getColor():

Denne funktion er afgørende i projektet, da den indeholder information om hvilken farve der senest er registreret. Koden for getColor()-funktionen kan ses i den vedhæftede kode. Denne funktion kaldes i main og lægges i en for-løkke, så man hele tiden kender den seneste registrerede farve.

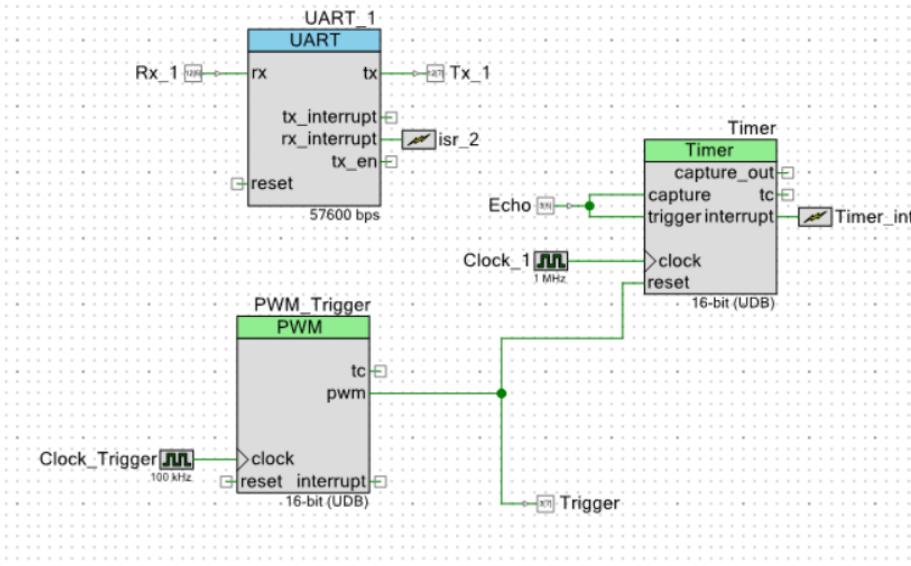
Funktionen består af if-statements og counters, som udgør logikken for denne funktion. If-sætningerne skal blandt andet afgøre hvilke farvefrekvenser, som har ændret sig mest i forhold til den oprindelige frekvens, hentet fra standardColor().

Disse if-sætninger indeholder blandt andet også logik, der tester om ændringen af frekvensen er over 10% i forhold til den tidligere. Kun hvis den specifikke farve stiger med 10%, vil farven blive registreret. Desuden er der for funktionalitetens skyld tilføjet en counter ved hver farve, som skal sørge for at den samme farve er registreret, før der returneres en melding om at denne farve er registreret. Dette er også med til at sikre, at der ikke registreres fejlmålinger, som kan have stor konsekvens for resten af systemet.

Funktionen returnerer tallene 1-4 alt afhængig af hvilken farve, der er registreret. Når funktionen returnerer 1 er farven rød blevet registreret. Det samme gør sig gældende for 2 og grøn, 3 og blå, samt 4 og 'unknown'. Alt afhængig af hvilket tal der returneres, så vil robotten skulle agere ud fra disse informationer.

4.3.5 Afstandssensor

PSoC Creator er benyttet til at skrive kode, da det understøtter PSoC. Her er det dog kun muligt at anvende C.



Figur 30: Topdesign af afstandssensor fra PSoC Creator

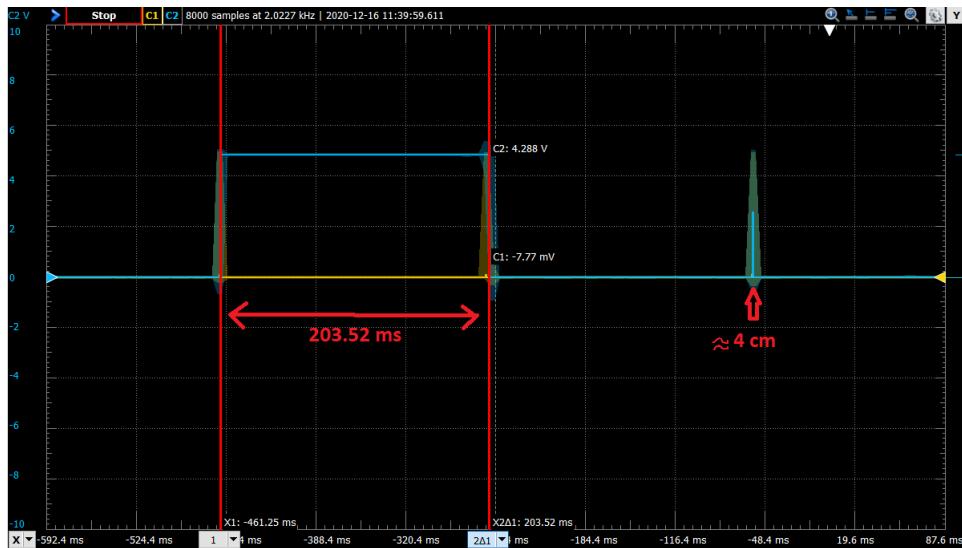
Topdesignet indeholder en PWM, som er tilsluttet 'Trigger'-pin. Den sørger for, at sende en impuls på 10us hver periode. Inde i den digitale PWM-komponent er én periode sat til 200ms. Samtidig nulstilles timeren og når 'Echo' går lav igen, gemmes værdien af timeren i 'capture'-registeret. Deretter bliver et interrupt aktiveret, som gemmer værdien i 'capture'-registeret i en variabel. 'Clock' er sat til 1 MHz, således at 1 count svarer til 1 us.

Timer-komponenten er sat til at udløse et interrupt, når echo går lav.

UART-komponenten benyttes til at teste systemet ved at udskrive den konverterede afstand ud på en terminal.

I datasheetet til HC-SR04 anbefales det, at vælge en målecyklus på mere end 60 ms, for at undgå at et trigger-signal sendes, mens sensoren stadig er ved at afvente en tidligere sendt lydbølge.

Ved målingen på figur 31 ses det dog, at echo-signalet kan vente op til 203ms, for at lydbølgen kommer tilbage. PWM-komponenten er derfor valgt til 300ms, for at sikre at et triggersignal ikke bliver sendt, mens sensoren venter.



Figur 31: Måling som viser, når der ikke detekteres et objekt

Funktion	Beskrivelse
void initDistSens()	Initierer PWM og Timer
void calcDist()	Aflæser capture-register og omregner til distance
int getDistance()	Returnerer distancen
void outputDistanceToUART(int distance)	Outputter distancen ud til terminal

Tabel 7: Funktioner i Afstandssensor

```

1 void calcDist()
2 {
3     int capture = Timer_ReadCapture(); //Capturing timer count
4     distance = (65535 - capture) * 0.017; //Calculating distance in cm
5     return distance;
6 }
```

Listing 6: Kodeudsnit for calcDist()

Funktionen calcDist() aflæser counteren på timeren og udregner en afstand i cm. I funktionen ses det, at afstanden udregnes ved at tage counteren og trække det fra 65535. Det gøres, da counteren tæller ned fra 65535. Derefter ganges der med 0.017, da lydens hastighed er ca. 343 m/s ved en

temperatur på 20 grader. Der benyttes formlen[10]:

$$afstand = captureVærdi \times 340(m/s)/2$$

Da afstanden udregnes i cm og counteren tæller i us, omskrives det til 0.0343 cm/us. Da lydbølgen både skal frem til objektet og tilbage, halveres afstanden.

outputDistanceToUART() benyttes kun til modultest, og benyttes ikke til integrationstesten. Den er oprettet for at se afstanden direkte på terminalen for at sikre, at afstanden er omregnet rigtigt, og for ikke at måle på oscilloskopet hver gang afstanden skal bestemmes.

Et problem som afstandssensoren har, er, at når den ikke registrerer et objekt inden for 400cm, vil den returnere en afstand på 3-4cm på trods af at oscilloskopet viser en længde på 203.52ms, som svarer til en afstand på ca. 3400cm.

Det ses på figur 31, at det kan skyldes, at når sensoren ikke modtager et signal tilbage, vil echo-signalet sende en meget kort impuls, som restarter timeren, og gemmer en ny værdi svarende til 4cm. Der er forsøgt at tage højde for denne fejl i listing 7 mellem linje 5 og 8.

```
1 void calcDist ()  
2 {  
3     int capture = Timer_ReadCapture(); //Capturing timer count  
4     distance = (65535 - capture) * 0.017; //Calculating distance in cm  
5     if(capture > 65250 && capture < 65340)  
6     {  
7         distance = 500;  
8     }  
9     return distance;  
10 }
```

Listing 7: Kodeudsnit for calcDist()

Værdierne på linje 5 er valgt, da count-værdien ligger mellem 65250 og 65340, når afstanden er længere end 400cm. Det vil dog medfører at Beertress ikke vil reagere, hvis et objekt faktisk er mellem 3.8 til 4.2cm væk.

Litteratur

[1] Datablad for HC-SR04. <https://cdn.sparkfun.com/datasheets/Sensors/Proximity/>

HCSR04.pdf.

- [2] *Datablad for I2C-slave komponent i PSoC.* <https://www.cypress.com/file/130946/download>.
- [3] *Datablad for L298 Dual Full Bridge.* <https://www.st.com/resource/en/datasheet/l298.pdf>.
- [4] *Datablad for Spændingsregulator LM1085.* <https://bit.ly/2Wa7eoB>.
- [5] *Datablad for Spændingsregulator LM7805.* <http://ee-classes.usc.edu/ee459/library/datasheets/LM7805.pdf>.
- [6] *Datablad for TCS230.* <http://www.w-r-e.de/robotik/data/opt/tcs230.pdf>.
- [7] *Embedded Stock til lån af elektroniske dele.* <https://stockmanager.ase.au.dk/>.
- [8] *Kredsløbdiagram over H-Bro udleveret af GFV.* <https://bit.ly/3nosHq3>.
- [9] *Load Cell Amplifier, side 5, figur 3.* https://blackboard.au.dk/bbcswebdav/pid-2873321-dt-content-rid-9904902_1/courses/BB-Cou-UUVA-91816/Lab%20-%20Build%20a%20scale%281%29.pdf.
- [10] *Speed of sound.* https://www.engineeringtoolbox.com/speed-sound-d_82.html.
- [11] *Teori om H-Broer.* <https://blog.digilentinc.com/what-is-an-h-bridge/>.
- [12] *Wheatstone bridge eksempel.* https://en.wikipedia.org/wiki/Load_cell.