

### 3. Semesterprojekt - Goofy Candy Gun Dokumentation - Gruppe 3

Rieder, Kasper      Jensen, Daniel V.      Nielsen, Mikkel  
201310514            201500152            201402530

Kjeldgaard, Pernille L.      Konstmann, Mia      Kloock, Michael  
PK94398                201500157            201370537

Rasmussen, Tenna  
201406382

20. maj 2016

# Indhold

<b>Indhold</b>	<b>ii</b>
<b>Figurer</b>	<b>iv</b>
<b>1 Kravspecifikation</b>	<b>1</b>
1.1 Aktør kontekst diagram . . . . .	1
1.2 Use Case Diagram . . . . .	1
1.3 Aktør beskrivelse . . . . .	2
1.3.1 Aktør - Bruger . . . . .	2
1.4 Fully Dressed Use Cases . . . . .	2
1.4.1 Use Case 1 - Spil Goofy Candy Gun 3000 . . . . .	4
1.4.2 Use Case 2 - Test Kommunikationsprotokoller . . . . .	5
1.5 Ikke funktionelle krav . . . . .	6
<b>2 Accepttestspezifikation</b>	<b>8</b>
2.1 Use case 1 - Hovedscenarie . . . . .	8
2.1.1 Use case 1 - Extension 1 . . . . .	9
2.1.2 Use case 1 - Extension 2 . . . . .	9
2.2 Use case 2 - Hovedscenarie . . . . .	10
2.2.1 Use case 2 - Exception 1 . . . . .	10
2.2.2 Use case 2 - Exception 2 . . . . .	10
2.2.3 Use case 2 - Exception 3 . . . . .	11
2.3 Ikke-funktionelle krav . . . . .	11
<b>3 Systemarkitektur</b>	<b>12</b>
3.1 Domæne model . . . . .	12
3.2 BDD for Candygun 3000 . . . . .	13
3.2.1 Blokbeskrivelse . . . . .	13
3.3 IBD for Candygun 3000 . . . . .	14
3.3.1 Signalbeskrivelse . . . . .	14
3.4 Software Allokéringsdiagram . . . . .	17
3.5 Use Case 1 - Spil Goofy Candy Gun 3000 . . . . .	18
3.6 Use Case 2 - Test Kommunikationsprotokoller . . . . .	19
3.6.1 Applikationsmodel for User Interface Software . . . . .	19
3.6.2 Applikationsmodel for Nunchuck Polling Software . . . . .	21
3.6.3 Applikationsmodel for Motor Control Software . . . . .	24
3.7 Kommunikationsprotokoller . . . . .	25
3.7.1 SPI Protokol . . . . .	26
3.7.2 I2C Protokol . . . . .	27
<b>4 Design og implementering</b>	<b>30</b>
4.1 Software Design . . . . .	30
4.1.1 Devkit8000 . . . . .	30
Candydriver . . . . .	30
Interfacedriver . . . . .	30
4.2 PSoC Software . . . . .	32
4.2.1 SPI - PSoC . . . . .	33
Klassediagram . . . . .	33

	Klassebeskrivelser . . . . .	34
	SPI Indstillinger på PSoC . . . . .	35
4.2.2	I2CCommunication . . . . .	35
	Klassediagram . . . . .	35
	Klassebeskrivelser . . . . .	36
	I2C indstillinger på PSoC . . . . .	36
4.2.3	Nunchuck . . . . .	36
	Klassediagram . . . . .	36
	Klassebeskrivelser . . . . .	37
4.3	Afkodning af Wii-Nunchuck Data Bytes . . . . .	37
4.4	Kalibrering af Wii-Nunchuck Analog Stick . . . . .	37
4.5	Hardwaredesign . . . . .	38
	Motor . . . . .	38
4.5.1	Motorstyring . . . . .	38
	H-bro . . . . .	38
	MOSFET'er . . . . .	39
	Dioder . . . . .	42
	Modstande . . . . .	42
	Transistorer . . . . .	42
	Potentiometer . . . . .	42
	ADC . . . . .	43
<b>5</b>	<b>Modultest</b>	<b>45</b>
5.1	Software . . . . .	45
5.1.1	Modultest af Wii-Nunchuck . . . . .	45
5.1.2	SPI Protokol . . . . .	46
	SPI Bus Test . . . . .	46
5.1.3	I2C Protokol . . . . .	48
	Test af NunchuckData kommandotype . . . . .	48
	NunchuckData kommandotype test del 1 . . . . .	48
	NunchuckData kommandotype test del 2 . . . . .	49
5.2	Hardware . . . . .	50
5.2.1	H-bro . . . . .	50
5.2.2	detektor . . . . .	53
5.3	Integrationstest - Use case 2 . . . . .	57
5.4	Accepttest . . . . .	60
<b>6</b>	<b>Referencer</b>	<b>61</b>

## Figurer

1	Kontekst diagram for slikkanonen . . . . .	1
2	Use case diagram for slikkanonen . . . . .	1
3	Skitse af brugergrænsefladen . . . . .	7
4	Domæne model for Candygun 3000 . . . . .	12
5	BDD for Candygun 3000 . . . . .	13
6	IBD for Candygun 3000 . . . . .	14
7	Software allokatons diagram . . . . .	18
8	Overordnet sekvensdiagram for Wii-Nunchuck informations flow .	19
9	Sekvensdiagram for Devkit 8000 . . . . .	20
10	Klassediagram for Devkit 8000 . . . . .	21
11	Sekvensdiagram for PSoC0 SPI test . . . . .	22
12	Sekvensdiagram for PSoC0 I2C test . . . . .	22
13	Sekvensdiagram for PSoC0 Nunchuck test . . . . .	23
14	Klassediagram for PSoC0 . . . . .	23
15	Sekvensdiagram for PSoC1 . . . . .	24
16	Klassediagram for PSoC1 . . . . .	25
17	Forbindelser mellem systemets komponenter . . . . .	25
18	Sekvensdiagram for at læse fra SPI-slaven . . . . .	27
19	Timing Diagram af 1-byte I2C aflæsning . . . . .	28
20	Eksempel af I2C Protokol Forløb . . . . .	29
21	Klassediagram for Interface driveren på Devkit8000 . . . . .	31
22	Klassediagram oversigt for PSoC0 . . . . .	32
23	Klassediagram oversigt for PSoC1 . . . . .	33
24	Klassediagram over klassen SPICommunication . . . . .	34
25	Klassediagram for I2CCommunication klassen . . . . .	35
26	Klassediagram for klassen Nunchuck . . . . .	36
27	Kredsløb for H-bro . . . . .	39
28	Gate-to-Source Voltage REFERENCE TIL DATABLAD HER!!!! .	40
29	Gate-to-Source Voltage REFERENCE TIL DATABLAD HER!!!! .	41
30	Spændingsdeler formlen for potentiometeret . . . . .	43
31	Hvordan ADC virker . . . . .	44
32	. . . . .	45
33	Tidslinje af aflæste I2C beskeder af PSoC0 fra Wii-Nunchuck . .	46
34	Opsætning for SPI-test . . . . .	47
35	Måling af kommandotype for start SPI test . . . . .	47
36	Måling af returbesked for start SPI test . . . . .	48
37	Tidslinje af målt I2C kommandotype . . . . .	48
38	Afmåling af modtager-buffer på PSoC1 efter at have modtaget "NunchuckData"kommando typen. Intet input på Nunchuck'en . .	49
39	Afmåling af modtager-buffer på PSoC1 efter at have modtaget "NunchuckData"kommando typen. Den analoge stick er presset til venstre på Nunchuck'en . . . . .	49
40	Afmåling af modtager-buffer på PSoC1 efter at have modtaget "NunchuckData"kommando typen. Den analoge stick er presset frem på Nunchuck'en . . . . .	50
41	Opstilling af h bro på fumlebræt . . . . .	51
42	Modultest for H-bro, blå går høj . . . . .	52
43	Modultest for H-bro, orange går høj . . . . .	52

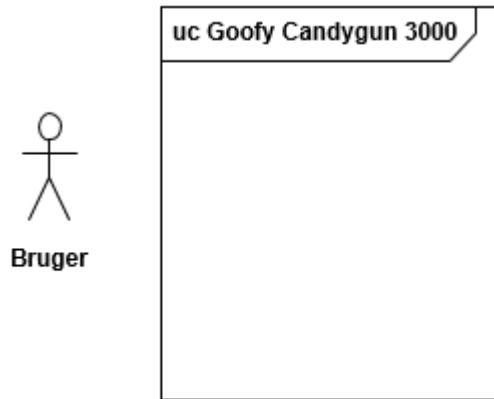
44	opstilling af test for detektor . . . . .	54
45	måleresultat fra analog Discovery, ved først del af test . . . . .	55
46	måleresultat fra ADC, ved først del af test . . . . .	55
47	måleresultat fra analog Discovery, ved anden del af test . . . . .	56
48	måleresultat fra ADC, ved anden del af test . . . . .	56
49	måleresultat fra analog Discovery, ved trejde del af test . . . . .	57
50	måleresultat fra ADC, ved trejde del af test . . . . .	57
51	Integrationstest opstilling . . . . .	58
52	Resultat af integrationstesten . . . . .	59
53	Resultat hvis man ikke klikker på nunchuck . . . . .	60

## 1 Kravspecifikation

Det følgende afsnit udpensler projektet ved specifikation af aktører, use cases, samt ikke-funktionelle krav.

### 1.1 Aktør kontekst diagram

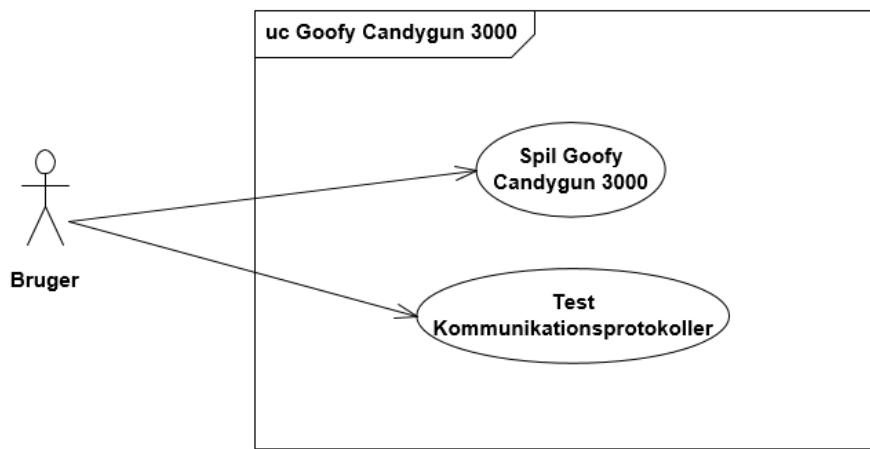
Figur 1 viser et kontekst diagram for Goofy Candygun 3000.



Figur 1: Kontekst diagram for slikkanonen

### 1.2 Use Case Diagram

Figur 2 viser et use case diagram for Goofy Candygun 3000.



Figur 2: Use case diagram for slikkanonen

### 1.3 Aktør beskrivelse

Det følgende afsnit beskriver de identificerede aktører for Goofy Candygun 3000.

#### 1.3.1 Aktør - Bruger

Aktørens Navn:	Bruger
Alternativ Navn:	Spiller
Type:	Primær
Beskrivelse:	Brugeren initierer Goofy Candy Gun, ved at vælge spiltype på brugergrænsefladen. Derudover har brugeren mulighed for at stoppe spillet igennem brugergrænsefladen. Brugeren vil under spillet interagere med Goofy Candy Gun gennem Wii-Nunchucken. Brugeren starter også Goofy Candy Gun system-testen for at verificere om det er operationelt.

### 1.4 Fully Dressed Use Cases

Det følgende afsnit indeholder de *fully dressed use cases* for Goofy Candy Gun, som kan findes under afsnittet **Use Case Diagram**.



### 1.4.1 Use Case 1 - Spil Goofy Candy Gun 3000

<b>Navn</b>	Spil Goofy Candygun 3000
<b>Mål</b>	At spille spillet
<b>Initiering</b>	Bruger
<b>Aktører</b>	Bruger
<b>Antal samtidige forekomster</b>	Ingen
<b>Prækondition</b>	Spillet og kanonen er operationel. UC2 Test kommunikationsprotokoller er udført
<b>Postkondition</b>	Brugeren har færdiggjort spillet
<b>Hovedscenarie</b>	<ol style="list-style-type: none"> <li>1. Bruger vælger spiltype på brugergrænseflade</li> <li>2. Bruger vælger antal skud til runde</li> <li>3. Bruger fylder magasin med slik tilsvarende antal skud</li> <li>4. Bruger indstiller kanon med analogstick på Wii-nunchuck</li> <li>5. Bruger udløser kanonen med Wii-nunchucks trigger</li> <li>6. System lader et nyt skud</li> <li>7. Brugergrænseflade opdateres med spillets statistikker</li> <li>8. Punkt 4 til 7 gentages indtil skud er opbrugt           <ul style="list-style-type: none"> <li>[Extension 1: Bruger vælger 2 player mode]</li> <li>[Extension 2: Bruger afslutter det igangværende spil]</li> </ul> </li> <li>9. Brugergrænseflade viser afslutningsinfo for runden</li> <li>10. Bruger afslutter runde</li> <li>11. Brugergrænseflade vender tilbage til starttilstand</li> </ol>
<b>Udvidelser/ undtagelser</b>	<p><b>[Extension 1: Brugeren vælger 2 player mode]</b></p> <ol style="list-style-type: none"> <li>1. Bruger overdrager Wii-nunchuck til den anden bruger</li> <li>2. Punkt 4 til 7 gentages indtil skud er opbrugt</li> <li>3. Use case genoptages fra punkt 8</li> </ol> <p><b>[Extension 2: Bruger afslutter det igangværende spil]</b></p> <ol style="list-style-type: none"> <li>1. Brugergrænseflade vender tilbage til starttilstand</li> <li>2. Use case afsluttes</li> </ol>

#### 1.4.2 Use Case 2 - Test Kommunikationsprotokoller

<b>Navn</b>	Test kommunikationsprotokoller
<b>Mål</b>	At teste kommunikations protokoller
<b>Initiering</b>	Bruger
<b>Aktører</b>	Bruger
<b>Antal samtidige forekomster</b>	Ingen
<b>Prækondition</b>	Systemet er tændt
<b>Postkondition</b>	Systemet er gennemgået testen og resultaterne er vist
<b>Hovedscenarie</b>	<ol style="list-style-type: none"> <li>1. Bruger vælger test system på brugergrænseflade</li> <li>2. Devkit sender start SPI test til PSoC0 via SPI</li> <li>3. PSoC0 sender acknowledge til Devkit via SPI [Exception 1: PSoC0 sender ikke acknowledge]</li> <li>4. Brugergrænseflade meddeler om gennemført SPI test</li> <li>5. Devkit sender start I2C test til PSoC0 via SPI</li> <li>6. PSoC0 sender start I2C test til PSoC slaver via I2C</li> <li>7. PSoC slaver sender acknowledge til PSoC0 via I2C [Exception 2: PSoC slaver sender ikke acknowledge]</li> <li>8. PSoC0 meddeler om gennemført I2C test til Devkit via SPI</li> <li>9. Brugergrænseflade meddeler om gennemført I2C test</li> <li>10. Brugergrænseflade anmoder bruger om at trykke på knap 'Z' på Wii-nunchuck</li> <li>11. Wii-nunchuck sender besked "Knap Z trykket" til PSoC0 via I2C [Exception 3: Wii-nunchuck sender ikke "Knap Z trykket"]</li> <li>12. PSoC0 videresender besked om "Knap Z trykket" til Devkit via SPI</li> <li>13. Brugergrænseflade meddeler om gennemført Wii-nunchuck test</li> <li>14. Brugergrænseflade meddeler at test af kommunikationsprotokoller er gennemført</li> </ol>

<b>Udvidelser/ undtagelser</b>	<p><b>[Exception 1: PSoC0 sender ikke acknowledge]</b></p> <ol style="list-style-type: none"> <li>1. Brugergrænseflade meddeler fejl i SPI kommunikation</li> <li>2. UC2 afsluttes</li> </ol> <p><b>[Exception 2: PSoC slaver sender ikke acknowledge]</b></p> <ol style="list-style-type: none"> <li>1. PSoC0 sender fejlmeldelse til Devkit</li> <li>2. Brugergrænseflade meddeler fejl i I2C kommunikation</li> <li>3. UC2 afsluttes</li> </ol> <p><b>[Exception 3: Wii-nunchuck sender ikke "Knap Z trykket"]</b></p> <ol style="list-style-type: none"> <li>1. PSoC0 sender fejlmeldelse til Devkit</li> <li>2. Brugergrænseflade meddeler fejl i I2C kommunikation med Wii-nunchuck</li> <li>3. UC2 afsluttes</li> </ol>
--------------------------------	--

## 1.5 Ikke funktionelle krav

1. Kanonen må ikke kunne rotere 360 °
2. Kanonen skal kunne affyre projektiler med en diameter på  $1,25 \text{ cm} \pm 2 \text{ mm}$
3. Kanonen skal kunne affyre sit projektil minimum 1 meter
4. Kanonens størrelse må maksimalt være 50cm høj, bred og dyb
5. Fra aftryk på trigger til affyring må der maksimalt gå ti sekunder
6. Affyring af kanonen skal kunne afvikles minimum tre gange pr. minut
7. Figur 3 viser en skitse af hvordan den grafiskbrugergrænseflade kommer til at se ud



Figur 3: Skitse af brugergrænsefladen

## 2 Accepttestspezifikation

### 2.1 Use case 1 - Hovedscenarie

Step	Handling	Forventet observation/resultat	Faktisk observasjon/resultat	Vurdering (OK/FAIL)
1	Vælg one-player mode.	Brugergrænsefladen viser spilside for one-player mode og anmoder om, at der fyldes slik i magasinet og at kanon indstilles.		
3	Fyld slik i kanon. Indstil kanon til affyring med Wii-nunchuck.	Kanon indstiller sig svarende til Wii-nunchucks placering.		
4	Udløs kanon med trigger på wii-nunchuck.	Kanon udløses.		
5	Gentag punkt 4 og 5 to gange.	Punkt 4 og 5 gentages.		
6	Tryk på knap for at vende tilbage til starttilstand.	Brugergrænseflade vender tilbage til startside.		

### 2.1.1 Use case 1 - Extension 1

Step	Handling	Forventet observasjon/resultat	Faktisk observasjon/resultat	Vurdering (OK/FAIL)
1	Vælg two-player mode.	Brugergrænsefladen viser spilside for two-player mode og anmoder om valg af antal skud.		
2	Fyld slik i kanon. Indstil kanon til affyring med Wii-nunchuck.	Kanon indstiller sig svarende til Wii-nunchucks placering.		
3	Udløs kanon med trigger på Wii-nunchuck.	Kanon udløses.		
4	Giv Wii-nunchuck til den anden spiller.	Den anden spiller modtager Wii-nunchuck.		
5	Gentag punkt 5 til 6 indtil skud er opbrugt.	Punkt 5 til 6 gentages.		
6	Tryk på knap for at vende tilbage til starttilstand.	Brugergrænseflade vender tilbage til startside.		

### 2.1.2 Use case 1 - Extension 2

Step	Handling	Forventet observasjon/resultat	Faktisk observasjon/resultat	Vurdering (OK/FAIL)
1	Vælg one-player mode.	Brugergrænsefladen viser spilside for one-player mode og anmoder om, at kanon indstilles.		
2	Tryk på knap for afslutning af spil.	Brugergrænseflade vender tilbage til startside.		

## 2.2 Use case 2 - Hovedscenarie

Step	Handling	Forventet observation/resultat	Faktisk observation/resultat	Vurdering (OK/FAIL)
1.	Tryk på "Systemtest" på GUI	Systemtest brugergrænsefladen vises på Devkittet.		
2.	Tryk på "Start Test" på GUI"	Brugergrænsefladen udskriver at SPI og I2C testen er godkendt. Brugergrænsefladen anmoder brugereren om tryk på Z på Wii-nunchuck		
3.	Tryk 'Z' knappen på Wii-nunchucken	Brugergrænsefladen udskriver at Wii-testen er godkendt		

### 2.2.1 Use case 2 - Exception 1

Step	Handling	Forventet observation/resultat	Faktisk observation/resultat	Vurdering (OK/FAIL)
1.	Fjern SPI-kablet fra Devkittet			
2.	Tryk på "Systemtest" på GUI	Systemtest brugergrænsefladen vises på Devkittet.		
3.	Tryk på "Start Test" på GUI	Brugergrænsefladen udskriver at SPI forbindelsen mislykkedes		

### 2.2.2 Use case 2 - Exception 2

Step	Handling	Forventet observation/resultat	Faktisk observation/resultat	Vurdering (OK/FAIL)
1.	Fjern I2C-kabel fra PSoC0			
2.	Tryk på "Systemtest" på GUI	Systemtest brugergrænsefladen vises på Devkittet.		
3.	Tryk på "Start Test" på GUI	Brugergrænsefladen udskriver at SPI forbindelsen lykkedes og I2C forbindelsen mislykkedes		

### 2.2.3 Use case 2 - Exception 3

Step	Handling	Forventet observation/resultat	Faktisk observation/resultat	Vurdering (OK/FAIL)
1.	Afkobl Nunchucken fra systemet			
2.	Tryk på "Systemtest" på GUI	Systemtest brugergrænsefladen vises på Devkittet.		
3.	Tryk på "Start Test" på GUI	Brugergrænsefladen udskriver at SPI og I2C forbindelsen lykkedes		
4.	Afvent timeout	Brugergrænsefladen udskriver at Nunchuck forbindelsen mislykkedes		

### 2.3 Ikke-funktionelle krav

Krav	Test	Forventet observation/resultat	Faktisk observation/resultat	Vurdering (OK/FAIL)
1	Bruger drejer kanonen så lang til venstre og højre som muligt.	Det observeres at kanonen ikke har roteret 360 °		
2	Et projektil på 1.25 cm i diameter $\pm$ 5mm affyres fra kanonen.	Projektilet bliver affyret		
3	Mål kanonens dimensioner med en lineal.	Dimensionerne overstiger ikke 50cm x 50cm x 50cm.		
4	Tryk på 'Z' knappen på Wii Nunchuck, og mål med et stopur hvor lang tid der går fra tryk, til kanonen bliver affyret.	Den målte tid er mindre end 10 sekunder.		
5	Kanonen affyres 3 gange, og et stopur startes ved første skud, og stoppes ved det tredje skud.	Den målte tid er mindre end 60 sekunder.		

### 3 Systemarkitektur

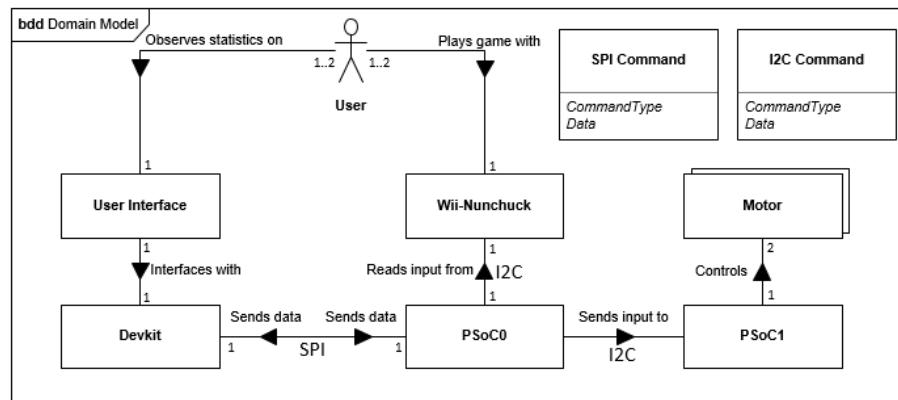
Systemarkitekturen har til formål at beskrive både hardware- og softwarekomponenter til sådan en grad at sammensætningen af hele system kan forstås. I dette afsnit beskrives arkitektur for både hardware og software.

I hardwarearkitekturen beskrives systemet ved at nedbryde det i blokke. I BDD'et er blokkenes porte og associationer til andre blokke angivet. Disse porte anvendes senere i afsnittet, hvor de benyttes i IBD'et.

I softwarearkitekturen udarbejdes der applikationsmodeller bestående af sekvensdiagrammer og klassediagrammer for hver use case, opdelt for hver CPU i systemet. Applikationsmodellerne har til formål at danne et overblik af de krav der stilles til softwaren for produktets uses cases. På denne måde kan de bruges som inspirerende grundlæg for software design og implementering. Applikationsmodellerne giver en overfladisk beskrivelse af CPU'ernes interaktioner.

#### 3.1 Domæne model

På figur 4 ses domæne modellen for systemet. Denne er lavet for at danne et overblik over systemet, og hvordan de forskellige dele overfladisk interagerer med hinanden.



Figur 4: Domæne model for Candygun 3000

I domæne modellen ses, at brugeren interagerer med både Wii-nunchucken og med brugergrænsefladen. Brugergrænsefladen er en grænseflade til softwaren devkittet. Devkittet kommunikerer med PSoC0, som læser den analoge data der kommer fra Wii-nunchucken. Denne data bliver derefter afkodet og videresendt til PSoC1, som styrer motorene ud fra dataene.

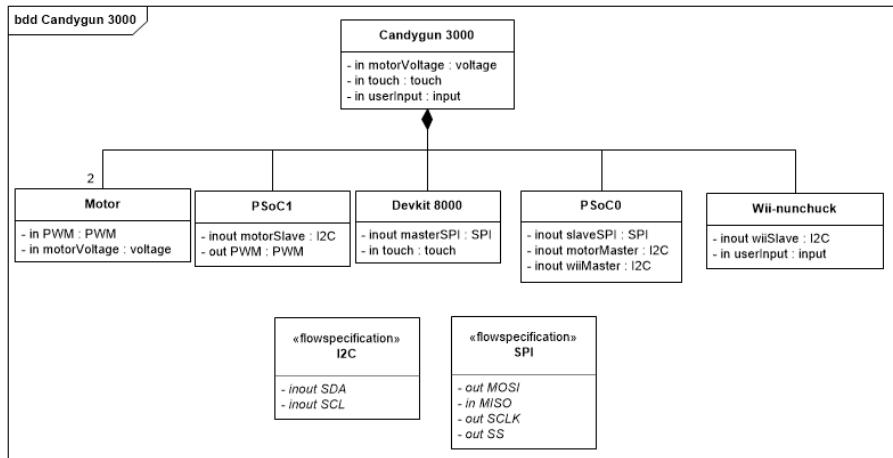
I "I2C Command"ses en grov estimering af interfacet mellem de forskellige I2C enheder i systemet. Commandtype bruges til at identificere hvilken data der bliver sendt. Data er en mængde tal-værdier der bliver tolket ud fra hvilken commandtype der er blevet sendt.

I "SPI Command"ses en estimering af interfacet mellem SPI enhederne i systemet. "Commandtype"bruges til at identificere hvilken opgave systemet skal udføre (f.eks. 'I2C-test'). Data kan indeholde tal-værdier, som repræsenterer

nunchuck værdier (buttonpress osv.). En mere detaljeret gennemgang af disse kommunikations protokoller kan findes i afsnit 3.7 - Kommunikationsprotokoller.

### 3.2 BDD for Candygun 3000

På figur 5 ses et *Block Definition Diagram (BDD)* hvor Candygun 3000 brudt ned i blokkene PSoC0, PSoC1 og Devkit 8000. Devkit 8000 er brugergrænsefladen, som brugeren kan interagere med via touchskærmen. Denne kommunikerer via SPI til PSoC0, som er SPI-slave. PSoC0 er forbundet til PSoC1, hvor kommunikeres via I2C og den fungerer som I2C-master. Udover bindelede mellem SPI og I2C kommunikationen har PSoC0 ansvaret for Wii-nunchucken. PSoC1 står for at aflæse og afkode input fra nunchucken, som også kommunikerer via I2C, og videresender dataene til PSoC1. PSoC1, som står for motorstyringen, tager dataene og konverterer dem til et PWM-signal. Dette signal sendes derefter til de to motorer. På figur 5 ses blokkenes associationer og deres porte.



Figur 5: BDD for Candygun 3000

#### 3.2.1 Blokbeskrivelse

Følgende afsnit indeholder en blokbeskrivelse samt en flowspecifikation for I2C og SPI. I flowspecifikationen beskrives I2C og SPI forbindelserne mere detaljeret fra en masters synsvinkel.

##### DevKit 8000

*DevKit 8000* er en embedded Linux platform med touch-skærm, der bruges til brugergrænsefladen for produktet. Brugeren interagerer med systemet og ser status for spillet via Devkit 8000.

##### Wii-Nunchuck

*Wii-Nunchuck* er controlleren som brugeren styrer kanonens retning med.

##### PSoC0

*PSoC0* er PSoC hardware der indeholder software til I2C og SPI kommunikationen og afkodning af Wii-Nunchuck data. PSoC0 fungerer som I2C master og

SPI slave. Denne PSoC er bindeleddet mellem brugergrænsefladen og resten af systemets hardware.

### Motor

*Motor* blokken er Candy Gun 3000's motorerer, der anvendes til at bevæge kanonen i forskellige retninger.

### PSoC1

*PSoC1* er PSoC hardware der indeholder software til I2C kommunikation og styring af Candy Gun 3000's motorer. PSoC1 fungerer som I2C slave.

### SPI (FlowSpecification)

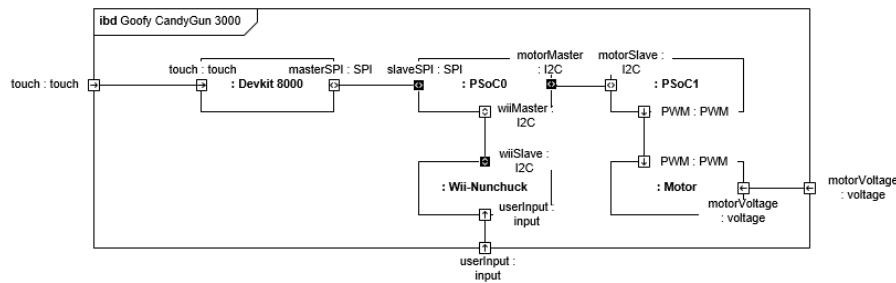
*SPI (FlowSpecification)* beskriver signalerne der indgår i *SPI* kommunikation.

### I2C (FlowSpecification)

*I2C (FlowSpecification)* beskriver signalerne der indgår i *I2C* kommunikation.

## 3.3 IBD for Candygun 3000

På baggrund af BDD'et er der lavet et *Internal Block Diagram (IBD)*. I IBD'et på figur 6 ses forbindelserne og portene mellem systemets blokke. Diagrammet viser grænsefladerne mellem blokkene og flowet i mellem disse.



Figur 6: IBD for Candygun 3000

### 3.3.1 Signalbeskrivelse

I signalbeskrivelsen gælder det, at når et signal beskrives som 'højt' opereres der i et spændingsområde på 3,5V til 5V. På samme måde er signaler beskrevet som 'lav' defineret som spændinger indenfor et område fra 0V til 1,5V. Disse spændingsniveauer er defineret ud fra standarden for CMOS kredse [? ].

Blok-navn	Funktionsbeskrivelse	Signaler	Signalbeskrivelse
Devkit 8000	Fungerer som grænseflade mellem bruger og systemet samt SPI master.	masterSPI	Type: SPI Spændingsniveau: 0-5V Hastighed: ?? Beskrivelse: SPI bussen hvori der sendes og modtages data.

		touch	Type: touch Beskrivelse: Brugertryk på Devkit 8000 touchdisplay.
PSoC0	Fungerer som I2C master for PSoC1 og Wii-Nunchuck samt SPI slave til Devkit 8000.	slaveSPI	Type: SPI Spændingsniveau: 0-5V Hastighed: ?? Beskrivelse: SPI bussen hvor der sendes og modtages data.
		wiiMaster	Type: I2C Spændingsniveau: ?? Hastighed: ?? Beskrivelse: I2C bussen hvor der modtages data fra Nunchuck.
		motorMaster	Type: I2C Spændingsniveau: 0-5V Hastighed: 100kbit/sekund Beskrivelse: I2C bussen hvor der sendes afkodet Nunchuck data til PSoC1.
PSoC1	Modtager nunchuck-input fra PSoC0 og omsætter dataene til PWM signaler.	motorSlave	Type: I2C Spændingsniveau: 0-5V Hastighed: 100kbit/sekund Beskrivelse: Indeholder formatteret Wii-Nunchuck data som omsættes til PWM-signal.
		PWM	Type: PWM Frekvens: 22kHz PWM %: 0-100% Spændingsniveau: 0-5V Beskrivelse: PWM signal til styring af motorens hastighed.

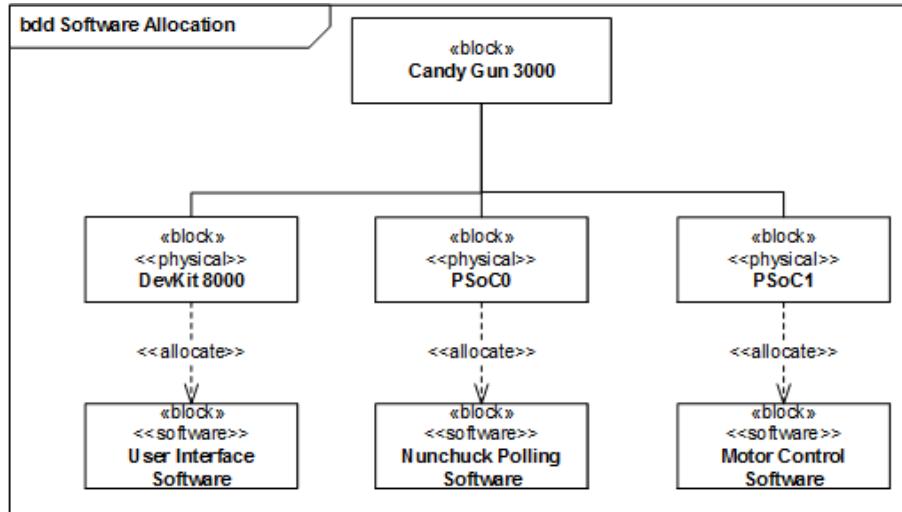
Motor	Den enhed der skal bevæge kanonen	PWM	Type: PWM Frekvens: 22kHz PWM%: 0-100% Spændingsniveau: 0-5V Beskrivelse: PWM signal til styring af motorens hastighed.
		motorVoltage	Type: voltage Spændingsniveau: 12V Beskrivelse: Strømforsyning til motoren
Wii-nunchuck	Den fysiske controller som brugeren styrer kanonen med.	wiiSlave	Type: I2C Spændingsniveau: 0-5V Hastighed: 100kbit/sekund Beskrivelse: Kommunikationslinje mellem PSoC1 og Wii-Nunchuck.
		userInput	Type: input Beskrivelse: Brugerinput fra Wii-Nunchuck.
SPI	Denne blok beskriver den ikke-atomiske SPI forbindelse.	MOSI	Type: CMOS Spændingsniveau: 0-5V Hastighed: ?? Beskrivelse: Binært data der sendes fra master til slave.
		MISO	Type: CMOS Spændingsniveau: 0-5V Hastighed: ?? Beskrivelse: Binært data der sendes fra slave til master.

		SCLK	Type: CMOS Spændingsniveau: 0-5V Hastighed: ?? Beskrivelse: Clock signalet fra master til slave, som bruges til at synkronisere den serielle kommunikation.
		SS	Type: CMOS Spændingsniveau: 0-5V Hastighed: ?? Beskrivelse: Slave-Select, som bruges til at bestemme hvilken slave der skal kommunikeres med.
I2C	Denne blok beskriver den ikke-atomiske I2C forbindelse.	SDA	Type: CMOS Spændingsniveau: 0-5V Hastighed: ?? Beskrivelse: Databussen mellem I2C masteren og I2C slaver.
		SCL	Type: CMOS Spændingsniveau: 0-5V Hastighed: ?? Beskrivelse: Clock signalet fra master til lyttende I2C slaver, som bruges til at synkronisere den serielle kommunikation.

Tabel 2: Tabel med signalbeskrivelse

### 3.4 Software Allokéringsdiagram

På figur 7 ses et software allokatons diagram. Dette diagram danner et overblik over hvilke CPU'er der findes i systemet, og hvilken software der skal allokeres på disse. Følgende applikationsmodeller tager udgangspunkt i softwaren der allokeres i figuren.



Figur 7: Software allokations diagram

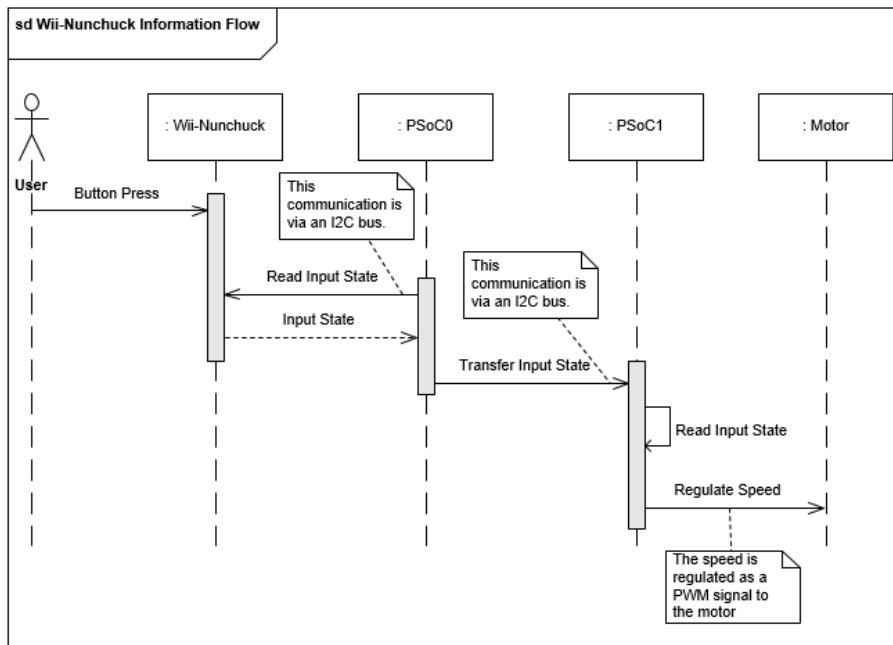
På tabel 3 er hvert allokeret software komponent beskrevet.

User Interface Software	Dette allokerede software er brugergrænsefladen som brugeren interagerer med på DevKit8000 touch-skærmen.
Nunchuck Polling Software	Dette allokerede software har til ansvar at polle Nunchuck tilstanden og videresende det til PSoC1.
Motor Control Software	Dette allokerede software har til ansvar at bruge den pollede Nunchuck data fra PSoC0 til motorstyring samt affyringsmekanismen.

Tabel 3: Beskrivelse af den allokerede software

### 3.5 Use Case 1 - Spil Goofy Candy Gun 3000

Følgende afsnit præsenterer applikationsmodeller relevante til Use Case 1 - Spil Goofy Candy Gun 3000, fordelt over systemets CPU'er.



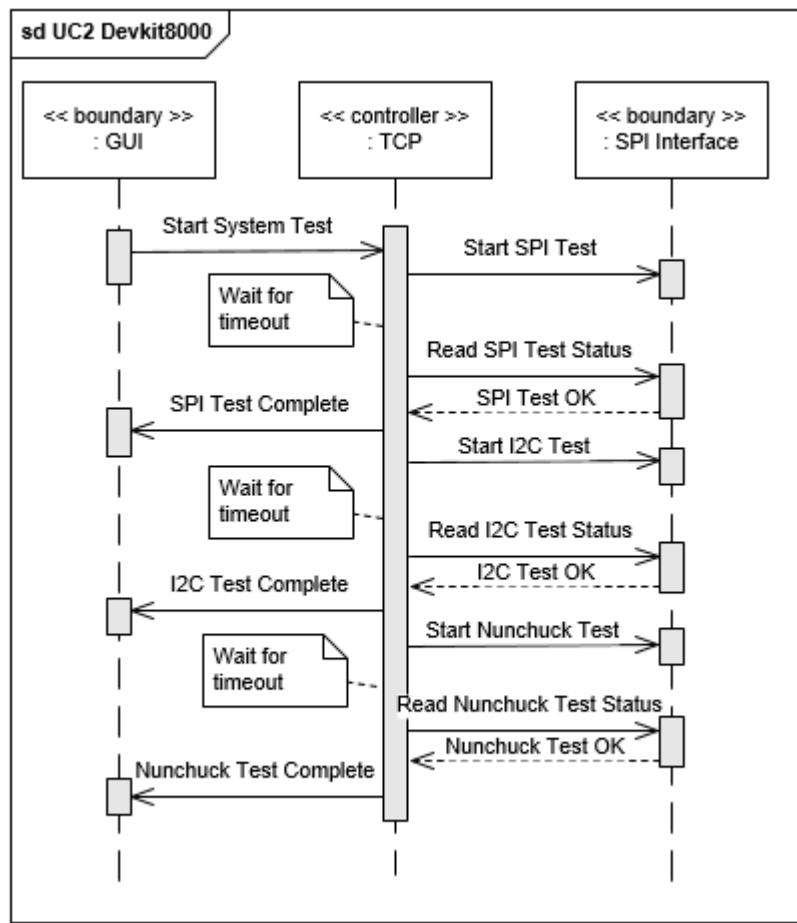
Figur 8: Overordnet sekvensdiagram for Wii-Nunchuck informations flow

### 3.6 Use Case 2 - Test Kommunikationsprotokoller

Følgende afsnit præsenterer applikationsmodeller relevante til Use Case 2 - Test Kommunikationsprotokoller, fordelt over systemets CPU'er.

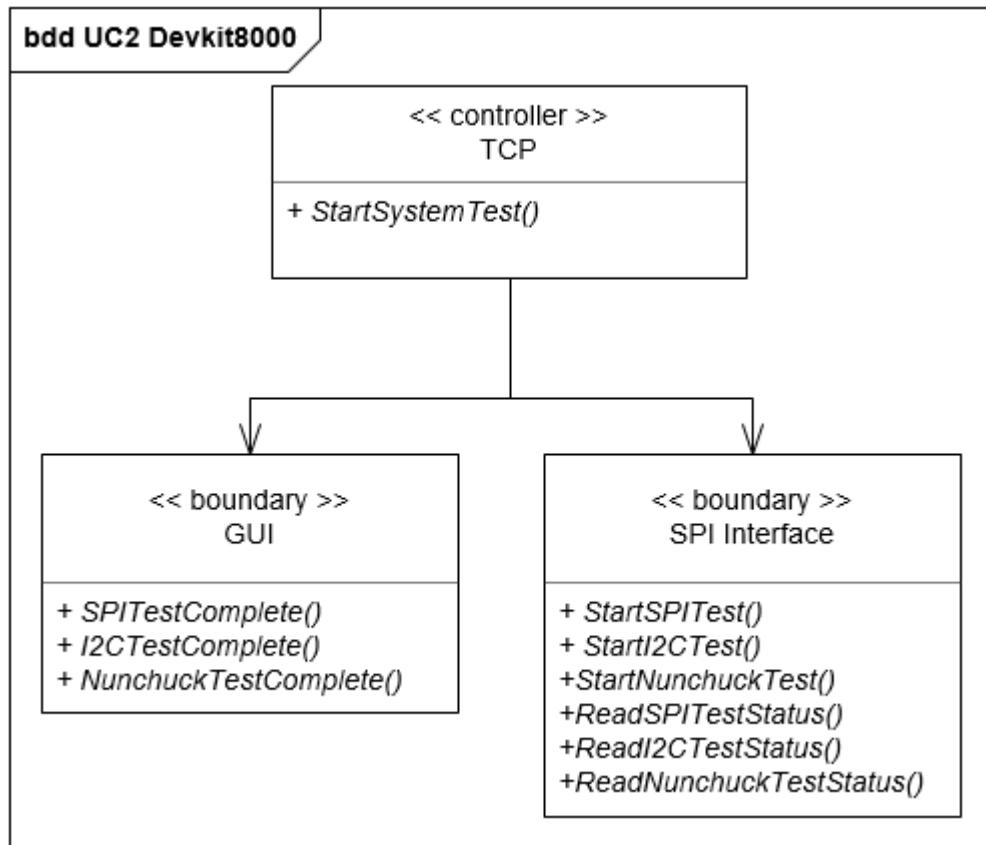
#### 3.6.1 Applikationsmodel for User Interface Software

Sekvensdiagrammet for Devkit 8000 ses på figur 9. Kontrolklassen er Test Communication Protocol, er på figurerne er forkortet til TCP. Der er to boundaryklasser, da DevKit 8000 skal håndtere kommunikationen mellem brugeren og PSoC0. Brugeren interagerer via en grafisk brugergrænseflade (GUI). Som det ses af diagrammet initieres testen af brugeren via brugergrænsefladen og derefter er det kontrolklassen, der sørger for, at de forskellige tests bliver sat i gang ved at kommunikere med PSoC0. Når en test er færdiggjort meldes resultatet ud til brugeren via GUI'en.



Figur 9: Sekvensdiagram for Devkit 8000

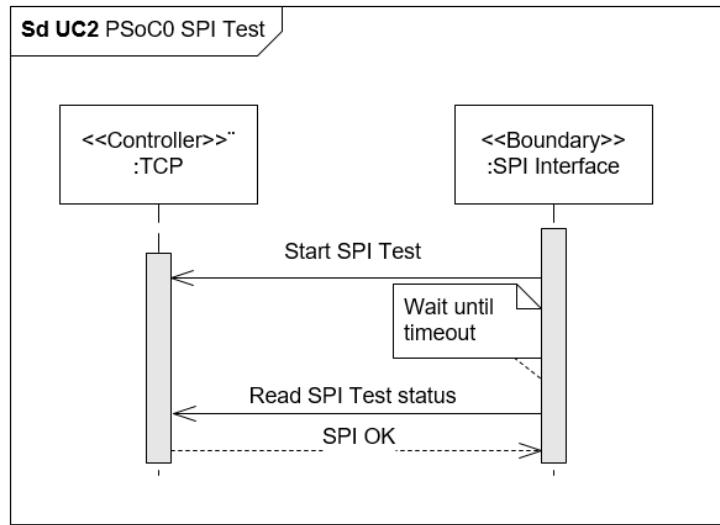
Ud fra sekvensdiagrammet for Devkit 8000 er der udledt metoder til klasserne. De udledte metoder ses i klassediagrammet på figur 10.



Figur 10: Klassediagram for Devkit 8000

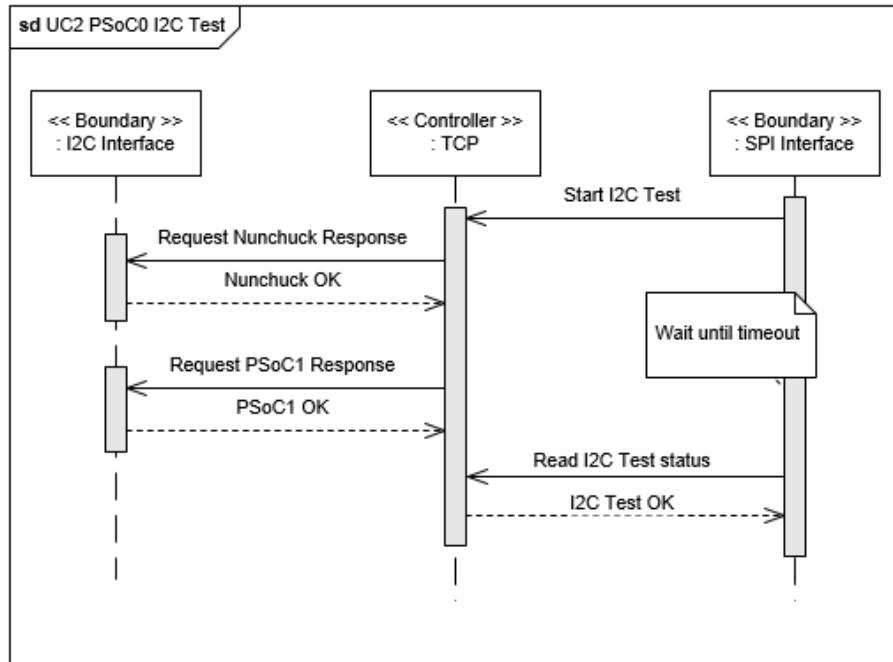
### 3.6.2 Applikationsmodel for Nunchuck Polling Software

På figur 11, 12 og 13 ses sekvensdiagrammer for PSoC0. Sekvensdiagrammerne er blevet opdelt i de 3 tests der gennemføres i use casen; SPI, I2C og Nunchuck kommunikations tests. Kontrolklassen er Test Communication Protocol, hvilket på figurerne er forkortet til TCP. Derudover er der tre boundaryklasser, da PSoC0 skal kommunikere både med Devkit 8000, Nunchucken og PSoC1.



Figur 11: Sekvensdiagram for PSoC0 SPI test

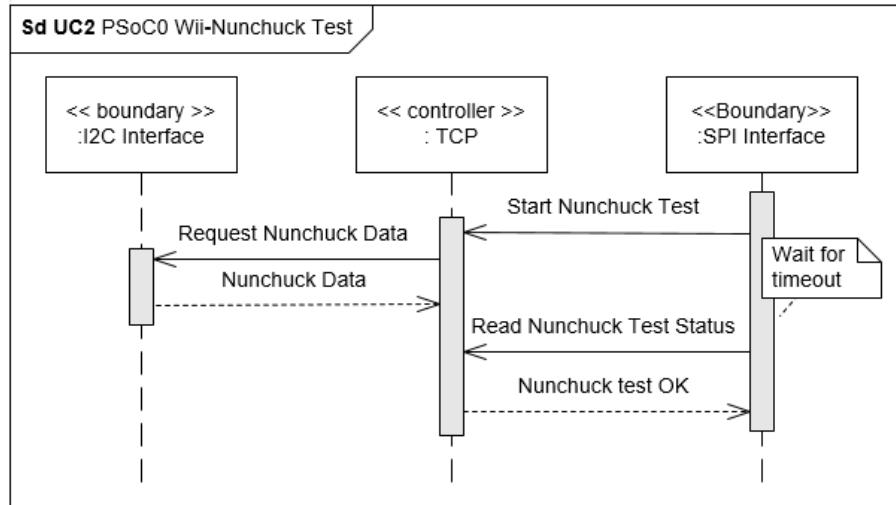
På figur 11 ses, at Devkittet sender en besked til kontrolklassen for at påbegynde SPI testen. Når testen er udført, svarer denne med en SPI OK og derved er SPI testen gennemført.



Figur 12: Sekvensdiagram for PSoC0 I2C test

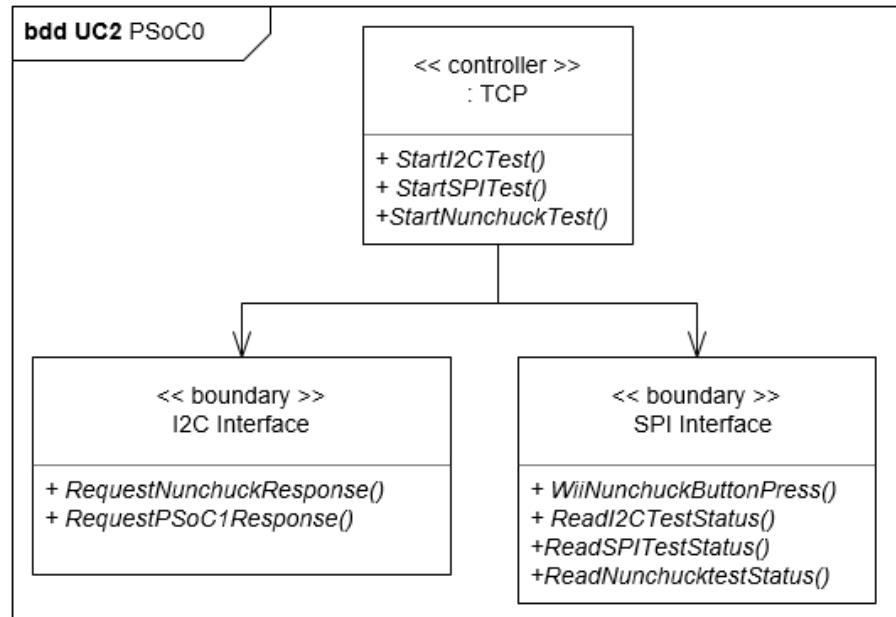
På figur 12 ses, at Devkittet starter I2C testen, ved at sende en besked til kontrolklassen. Kontrolklassen anmoder herefter om svar via I2C nettet fra

Nuchucken og PSoC1. Når disse enheder har svaret med et I2C OK er testen gennemført og der sendes besked til Devkittet med en I2C Test OK.



Figur 13: Sekvensdiagrammet for Wii-Nunchuck testen

På figur 13 ses sekvensdiagrammet for Wii-Nunchuck testen. Når der sker et tryk sender Nuchucken 'Z' knappens status til kontrolklassen. Herefter videresender kontrolklassen knappens status til Devkittet og derved er testen færdig.

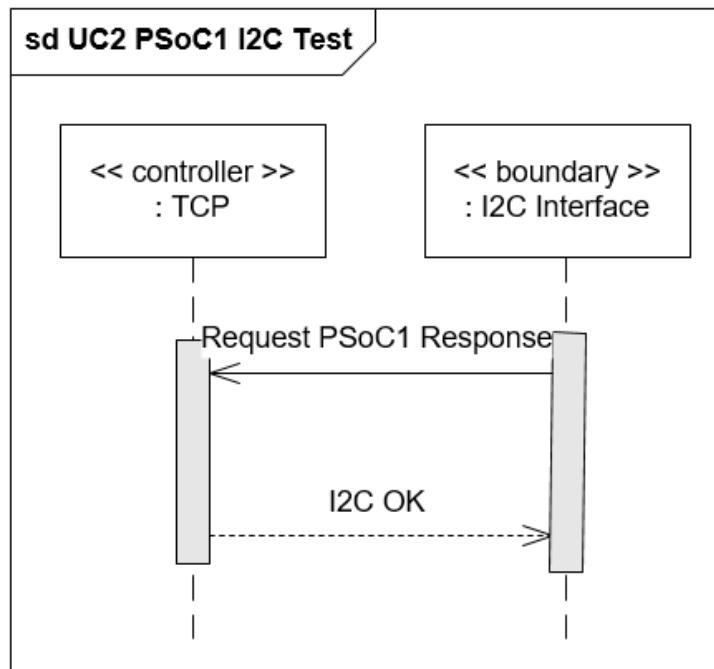


Figur 14: Klassediagram for PSoC0

I klassediagrammet på figur 14 ses kontrolklassen og de tre boundaryklasser, som hører til PSoC0. I klasserne er der tilføjet metoder, som er udledt ud fra sekvensdiagrammerne.

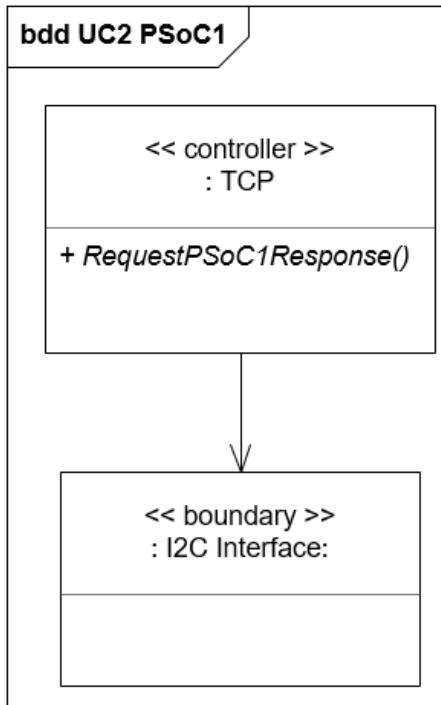
### 3.6.3 Applikationsmodel for Motor Control Software

Sekvensdiagrammet for PSoC1 ses på figur 15. Som forrig afsnit er kontrolklassen Test Communication Protocols, hvilket i diagrammet er forkortet til TCP. I dette tilfælde er der kun én boundaryklasse, da PSoC1, i denne use case, kun anvendes til I2C testen og derfor kun skal kommunikere med PSoC0.



Figur 15: Sekvensdiagram for PSoC1

På figur 15 ses, at boundaryklassen, PSoC0, anmoder om respons fra kontrolklassen til at bekræfte om at I2C slaven kan kommunikeres med. Hvis dette er tilfældet sendes der I2C OK tilbage til boundaryklassen.

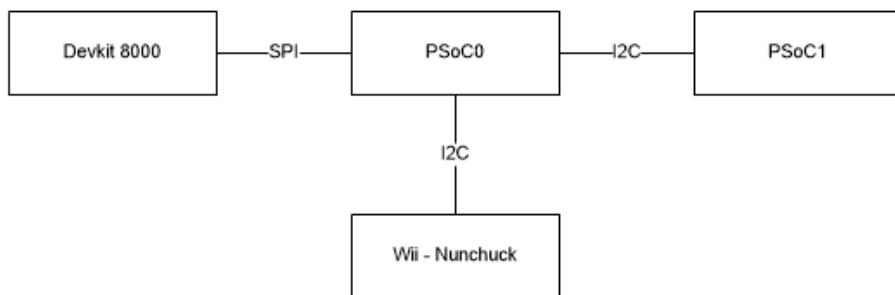


Figur 16: Klassediagram for PSoC1

Fra sekvensdiagrammet på figur 15 udledes et klassediagram som ses foroven i figur 16.

### 3.7 Kommunikationsprotokoller

I dette afsnit beskrives de kommunikationsprotokoller der anvendes til at sende data mellem systemets komponenter på de anvendte bustyper - I2C og SPI. På figur 17 ses hvilke bustyper der anvendes mellem systemets komponenter.



Figur 17: Forbindelser mellem systemets komponenter

### 3.7.1 SPI Protokol

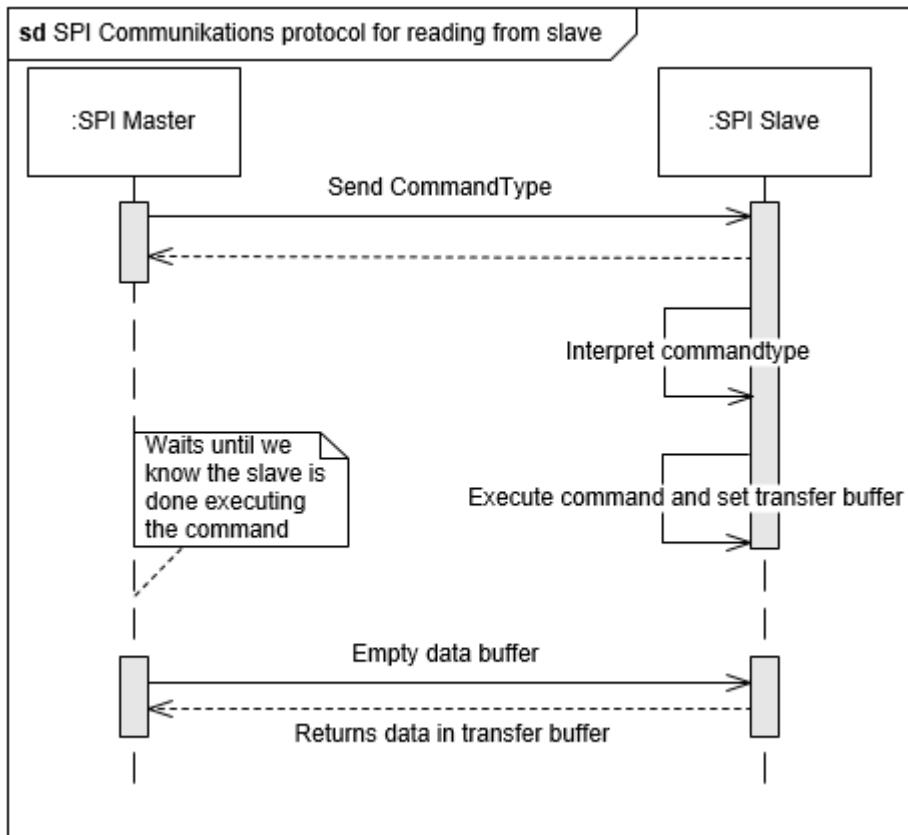
Til kommunikation mellem Devkittet og PSoC0 anvendes der *Serial Parallel Interface (SPI)*. SPI-forbindelsen består af fire signaler. Et af signalerne er *slave select* (SS). Det muliggør kommunikation med flere slaver. Selve dataen sendes via signalerne *Master Out Slave In* (MOSI) og *Master In Slave Out* (MISO). Det foregår ved *Full Duplex*, hvor der altid sendes i begge retninger på én gang. Hvis der kun ønskes at sende i én retning, kan der sendes 0'er i den anden retning. Derudover er der en *Clock* (SCLK), som sørger for at kommunikationen er synkron. I forhold til timing er det vigtigt, at både master og slave anvender samme *Timing Mode*, som styres af to bit, *SPI Clock Polarity Bit* (CPOL), som bestemmer om clock'en starter højt eller lavt, og *SPI Clock Phase Bit* (CPHA), som afgør om data samples på clock'ens rising- eller falling edge. I dette projekt anvendes *Timing Mode 3*, hvor clock'en starter høj og sampler på rising edge. De kommandoer, der i projektet sendes via SPI, består af 8 bit.

På tabel 4 ses SPI-protokollen. Til venstre på tabellen ses navnet på de kommandoer, der sendes. Alle kommandoer med "TEST" i navnet sendes fra Devkit8000 til PSoC0 via MOSI. De resterende kommandoer er svar på de forskellige tests, som sendes via MISO fra PSoC0 til Devkit8000. Det er værd at lægge mærke til, at det er bevidst, at der ikke er en kommando for SPI\_FAIL. Da SPI-forbindelsen ikke testes på PSoC0, men ved at PSoC0 sender *SPI\_OK* tilbage til Devkit8000 via SPI-forbindelsen. Dermed kan det først afgøres om SPI-forbindelsen fungerer korrekt, når Devkit8000, modtager (eller ikke modtager) succes-beskeden fra PSoC0.

Kommandotype	Beskrivelse	Binær Værdi	Hex Værdi
START_SPI_TEST	Sætter PSoC0 i 'SPI-TEST' mode	1111 0001	0xF1
START_I2C_TEST	Sætter PSoC0 i 'I2C-TEST' mode	1111 0010	0xF2
START_NUNCHUCK_TEST	Sætter PSoC0 i 'NUNCHUCK-TEST' mode	1111 0011	0xF3
SPI_OK	Signalerer at SPI-testen blev gennemført uden fejl	1101 0001	0xD1
I2C_OK	Signalerer at I2C-testen blev gennemført uden fejl	1101 0010	0xD2
I2C_FAIL	Signalerer at der skete fejl under I2C-testen	1100 0010	0xC2
NUNCHUCK_OK	Signalerer at NUNCHUCK-testen blev gennemført uden fejl	1101 0011	0xD3
NUNCHUCK_FAIL	Signalerer at der skete fejl under NUNCHUCK-testen	1100 0011	0xC3

Tabel 4: Kommandotyper der anvendes ved SPI kommunikation

For at aflæse en besked fra SPI-slaven, skal slaven først klargøre dens SPI-transfer buffer. Dette gøres ved, at SPI-masteren sender en commandotype til slaven, som derefter tolker kommandotypen, og eksekverer kommandoen, hvor SPI-transfer bufferen bliver sat. Imens dette sker, venter SPI-masteren i et bestemt stykke tid, for at sikre at transfer-bufferen når at blive sat. På figur 18 ses et sekvensdiagram der illustrerer dette.



Figur 18: Sekvensdiagram for at læse fra SPI-slaven

### 3.7.2 I2C Protokol

I2C[?] er en bus bestående af to ledninger. Den ene ledning bruges som databus og navngives *Serial Data Line* (SDA). Den anden ledning bruges til clock signalet, der synkroniserer kommunikationen, og navngives *Serial Clock Line* (SCL). Enheder på I2C bussen gør brug af et master-slave forhold til at sende og læse data. En fordel ved I2C bussen er at netværket kan bestå af multiple masters og slaver, hvilket udnyttes i dette system da tre I2C komponenter skal sende data mellem hinanden.

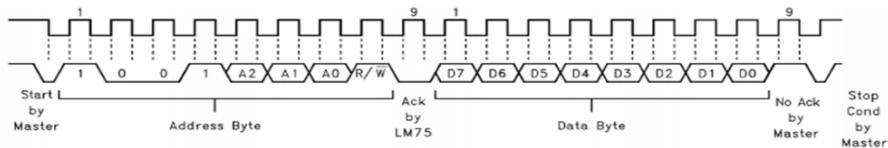
I2C gør brug af en integreret protokol der anvender adressering af hardwareenheder for at identificere hvilken enhed der kommunikeres med. På tabel 5 ses addresserne tildelt systemets PSoCs.

I2C Adresse bits	7	6	5	4	3	2	1	0 (R/W)
PSoC0	0	0	0	1	0	0	0	0/1
PSoC1	0	0	0	1	0	0	1	0/1

Tabel 5: Adresser der anvendes på I2C bussen

Den integrerede I2C protokol sender data serielt i pakker af 8-bit (1 byte).

På figur 19 ses et timing diagram for aflæsning af 1 byte. Figuren er et udsnit fra databladet for en LM75, der anvender fire faste adressebits, tre vilkårlige bit og en read/write bit. Her ses at transmissionen af data begynder med en adresse-byte til slaven efterfulgt en acknowledge til masteren og herefter sendes en data-byte.

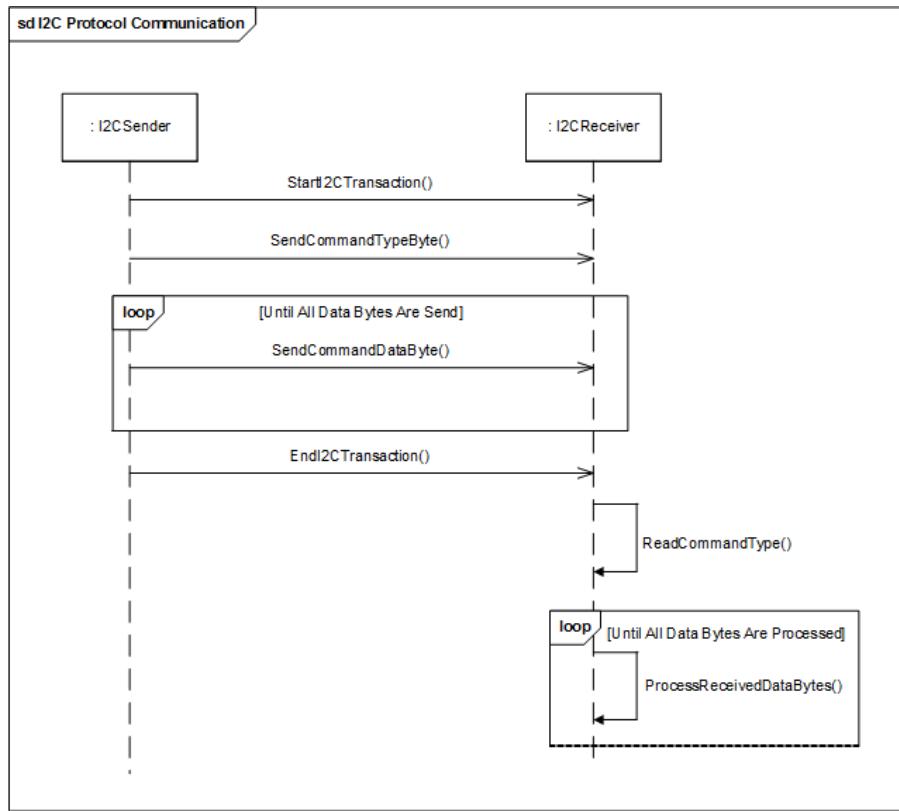


Figur 19: Timing Diagram af 1-byte I2C aflæsning

Goofy candygun gør brug af I2C protokollen via PSoC's I2C *Application Programming Interface* (API). Ved brug af denne API er der blevet udviklet en kommunikationsprotokol mellem systemets PSoCs, som gør det muligt at sende kommandoer og data.

Da I2C dataudveksling sker bytevist, er kommunikations protokollen opbygget ved, at kommandoens type indikeres af den første modtagne byte. Herefter følger  $N$ -antal bytes som er kommandoens tilhørende data.  $N$  er et vilkårligt heltal og bruges i dette afsnit når der refereres til en mængde data-bytes der sendes med en kommandotype.

Kommandoens type definerer antallet af databytes modtageren skal forvente og hvordan disse skal fortolkkes. På figur 20 ses et sekvensdiagram der, med pseudo-kommandoer, demonstrerer forløbet mellem en I2C afsender og modtager ved brug af kommunikations protokollen.



Figur 20: Eksempel af I2C Protokol Forløb

På figur 20 ses at afsenderen først starter en I2C transaktion, hvorefter typen af kommando sendes som den første byte. Efterfølgende sendes  $N$  antal bytes, afhængig af hvor meget data den givne kommandotype har brug for at sende. Efter afsluttet I2C transaktion læser I2C modtageren typen af kommando, hvor den herefter kan fortolke  $N$  antal modtagne bytes afhængig af den modtagne kommandotype.

På tabel 6 ses de definerede kommandotyper og det tilsvarende antal af bytes der sendes ved dataveksling.

Kommandotype	Beskrivelse	Binær Værdi	Hex Værdi	Data Bytes
NunchchuckData	Indholder aflest data fra Wii Nunchuck controlleren	0010 1010	0xA2	Byte #1 Analog X-værdi Byte #2 Analog Y-værdi Byte #3 Analog Buttonstate
I2CTestRequest	Anmoder PSoC0 om at starte I2C-kommunikations test	0010 1001	0x29	Ingen databyte
I2CTestAck	Anmodning om at få en I2C OK besked fra I2C enhed	0010 1000	0x28	Ingen databyte

Tabel 6: Kommandotyper der anvendes ved I2C kommunikation

Kolonneerne "Binær Værdi" og "Hex Værdi" i tabel 6 viser kommandotypens unikke tal-ID i både binær- og hexadecimalform. Denne værdi sendes som den første byte, for at identificere kommandotypen.

## 4 Design og implementering

### 4.1 Software Design

#### 4.1.1 Devkit8000

##### Candydriver

Candydriveren sørger for SPI-kommunikationen fra Devkit8000 til PSoC0. SPI-kommunikationen er implementeret med SPI bus nummer 1, SPI chip-select 0 og en hastighed på 1 MHz (et godt stykke under max på 20 MHz for en sikkerhedsskyld). Desuden starter clocken højt og data ændres på falling edge og aflæses på rising edge. Dermed bliver SPI Clock Mode 3. Derudover sendes der 8 bit pr transmission, hvilket passer med SPI-protokollen for projektet.

For at kunne anvende driveren, når SPI er tilsluttet, er der oprettet et hotplug-modul, som fortæller kernen, at der er et SPI device, som matcher driveren. Det kan SPI-forbindelsen ikke selv gøre, som usb fx kan. Selve driveren er i candygun.c opbygget som en char driver. For at holde forskellige funktionaliteter adskilt er alle funktioner, der har med SPI at gøre, implementeret i filen candygun-spi.c. Så når der fx skal requestes en SPI ressource i init-funktionen i candygun.c, så anvender driveren en funktion fra candygun-spi.c til det. I probe-funktionen sættes bits\_per\_word til 8, da vi sender otte bit som nævnt tidligere. I exit-funktionen anvender candygun.c igen en funktion fra candygun-spi.c - denne gang til at frigive SPI ressourcen. I write-metoden gives der data med fra brugeren. I dette tilfælde udgøres brugeren af Interface driveren og dataet er en 8 bit kommando fra SPI-protokollen. Dog er dataet fra brugeren i første omgang læst ind som en charstreng. I write-metoden bliver det så lavet om til en int. For at overføre dataet på en sikker måde anvendes funktionen copy\_from\_user() til at overføre data fra brugeren. Write-funktionen fra candygun.c anvender derefter en write-funktion fra candygun-spi.c, hvor den sender brugerinputtet med. I den spi-relatede write-funktion bliver bruger inputtet lagt i transfer bufferen og der NULL bliver lagt i receive bufferen, og med spi\_sync-funktionen bliver det sendt.

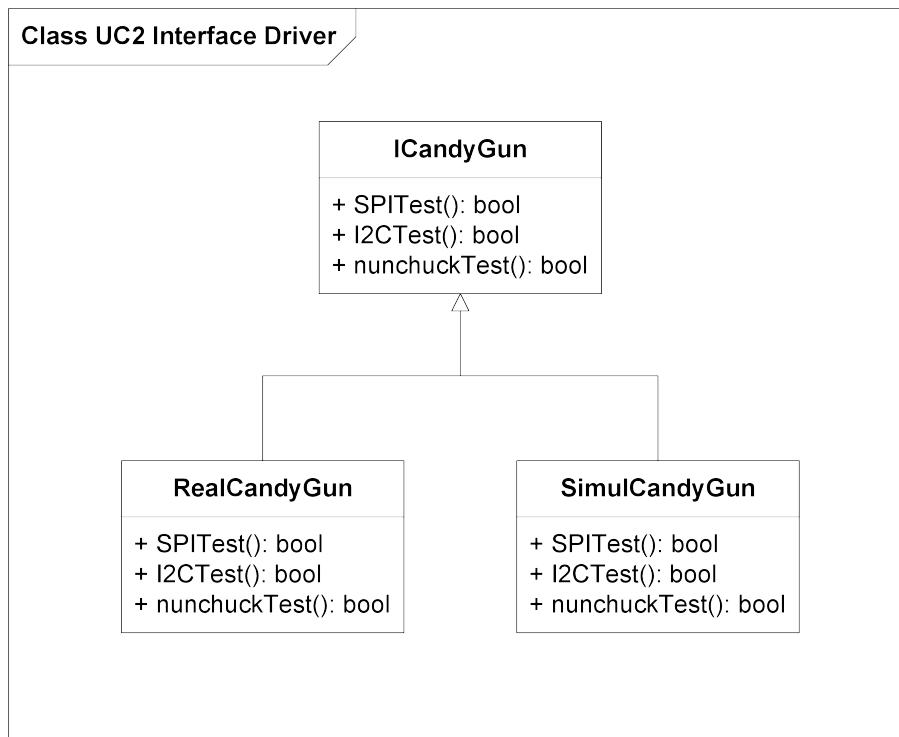
Ofte ville der en spi read-funktion først indeholde en write-del, som fortalte SPI-slaven, hvad der skulle læses over i bufferen. Det ville typisk efterfølges af et delay og så en read-del. Men i dette projekt skal der ofte afventes et brugerinput, som ikke kan styres af et fast delay, og der skal generelt sendes en aktiv kommando før der læses. Derfor er det besluttet at read-funktionen kun indeholder en read-del i transmissionen. Dermed skal write-funktionen altid aktivt anvendes inden der læses, da PSoC0 ellers ikke ved, hvad der skal gøres/lægges i bufferen.

Når funktionen har modtaget resultatet fra transmissionen returneres det til brugeren med funktionen copy\_to\_user(), som igen sørger for at overførslen af data foregår på en sikker måde.

##### Interfacedriver

Interface driveren er bindeled mellem brugergrænsefladen og candydriveren. Interface driveren er implementeret i c++ og opbygget med klasserelationen arv. Den indeholder tre funktioner til use case 2. De håndterer test af de forskellige kommunikationsforbindelser i systemet. Der er en basisklasse, ICandyGun,

som er abstrakt, da alle metoder er virtuelle. Derudover er der to afledte klasser, SimulCandyGun og RealCandyGun. SimulCandyGun simulerer svaret på funktionerne ved brug af rand()-funktionen. Den sikrer at brugergrænsefladen kan testes uden at være forbundet til det resterende system. RealCandyGun klassen implementerer de rigtig funktioner med forbindelse til det restende system, og anvender de nødvendige funktioner til at skrive til og læse fra et kernemodul. På figur 21 ses klassediagrammet for arvehierarkiet i Interface Driveren.



Figur 21: Klassediagram for Interface driveren på Devkit8000

I det følgende ses tabeller for funktionsbeskrivelser. Funktioner for ICandyGun-klassen er ikke beskrevet, da klassen er abstrakt og ingen af funktionerne dermed er implementeret i klassen. Til gengæld er både funktionerne for SimulCandygun og RealCandyGun beskrevet i hver sin tabel. Bemærk at funktionerne hedder det samme for begge klasser, da de begge er afledte funktioner i et arvehierarki, som det sås på figur 21. I tabel 4.1.1 ses funktionsbeskrivelser for SimulCandyGun.

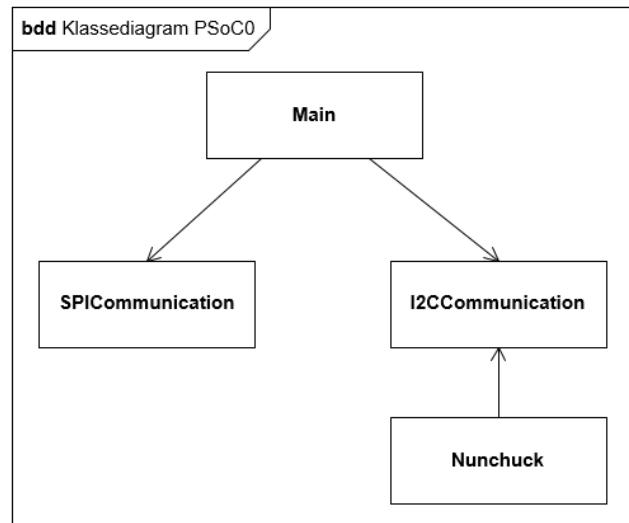
Metode	Beskrivelse
SPITest(): bool Der anvendes rand() til at returnerer et tilfældigt tal mellem 0 og 1.	Simulerer resultatet af SPI-testen i
I2CTest(): bool Der anvendes rand() til at returnerer et tilfældigt tal mellem 0 og 1.	Simulerer resultatet af I2C-testen i
nunchuckTest(): bool Der anvendes rand() til at returnerer et tilfældigt tal mellem 0 og 1.	Simulerer resultatet af nuchucktest

I tabel 4.1.1 ses funktionsbeskrivelser for RealCandyGun.

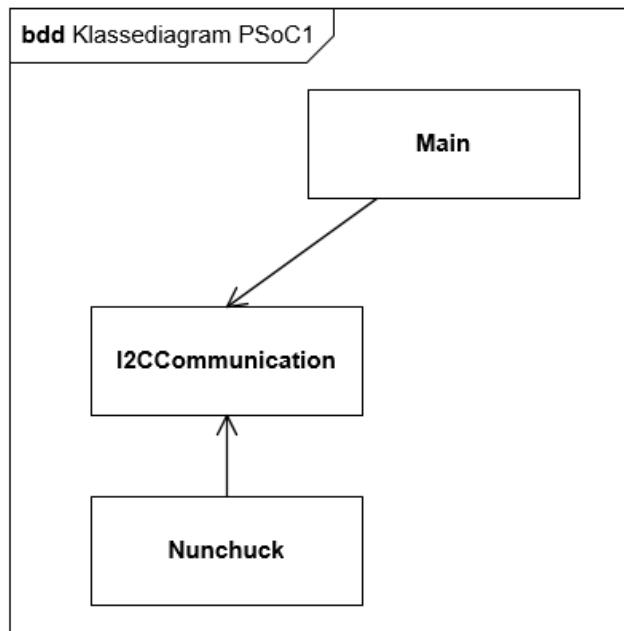
Metode	Beskrivelse
SPITest(): bool	Initierer SPI-test ved at skrive SPI-kommandoen "241" til "dev/candygun", som er
I2CTest(): bool	Initierer I2C-test ved at skrive I2C-kommandoen "242" til "dev/candygun", som er
nunchuckTest(): bool	Initierer nunchuck-test ved at skrive nunchuck-kommandoen "251" til "dev/candyg

## 4.2 PSoC Software

De følgende klassediagrammer på figur 22 og 23 giver et overblik over hvilke klasser der bliver gjort brug af på PSoC0 og PSoC1. De efterfølgende afsnit vil beskrive klasserne og deres funktioner.



Figur 22: Klassediagram oversigt for PSoC0



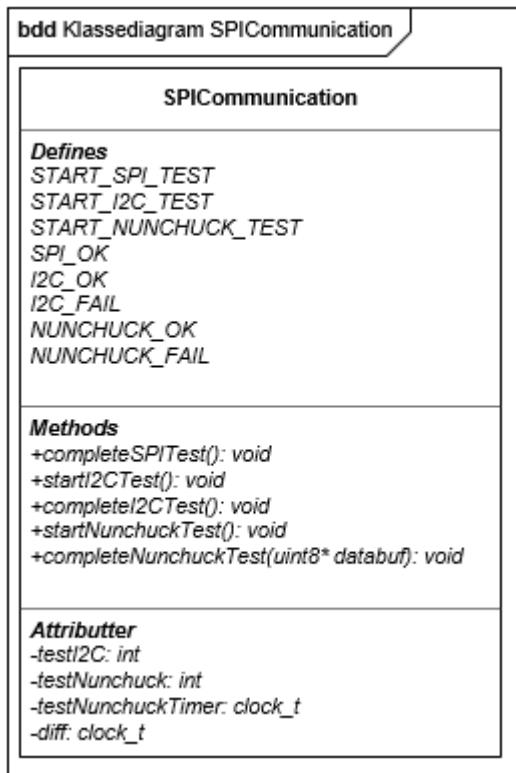
Figur 23: Klassediagram oversigt for PSoC1

#### 4.2.1 SPI - PSoC

I dette afsnit vil softwaren der specifikt omhandler SPI-kommunikationen mellem PSoC0 og DevKit8000 blive beskrevet. Dette gøres vha. et klassediagram og klassebeskrivelser

##### Klassediagram

På figur 24 ses klassediagrammet over SPICommunication klassen.



Figur 24: Klassediagram over klassen SPICommunication

### Klassebeskrivelser

I dette afsnit vil klassens metoder og defines blive beskrevet.

#### **void completeSPITest()**

Denne metode gemmer *SPI\_OK* i PSoC'ens SPI-transfer buffer.

#### **void completeI2CTest()**

Denne metode gennemfører I2C-Testen. Dette gøres ved at der sendes en besked til alle enheder på I2C-nettet, og hvis der ikke registreres nogen fejl på denne besked, bliver *I2C\_OK* gemt i PSoC'ens SPI-transfer buffer. Registreres der en fejl, bliver *I2C\_FAIL* gemt i SPI-transfer buffer.

#### **void completeNunchuckTest(uint8\* databuf)**

Denne metode gennemfører Nunchuck-testen. Dette gøres ved at der startes en timer på 6 sekunder. Hvis der sker et tryk på 'Z'-knappen på nunchucken indenfor disse 6 sekunder, vil *NUNCHUCK\_OK* blive gemt i SPI-transfer bufferen. Hvis der ikke registreres nogen tryk inden for de 6 sekunder, er det *NUNCHUCK\_FAIL* der gemmes.

I klassediagrammet er der beskrevet en række "Defines". Disse defines bruges i klassen som unikke ID'er der indikerer om en test er gennemført OK eller om

den er fejlet.

### SPI Indstillinger på PSoC

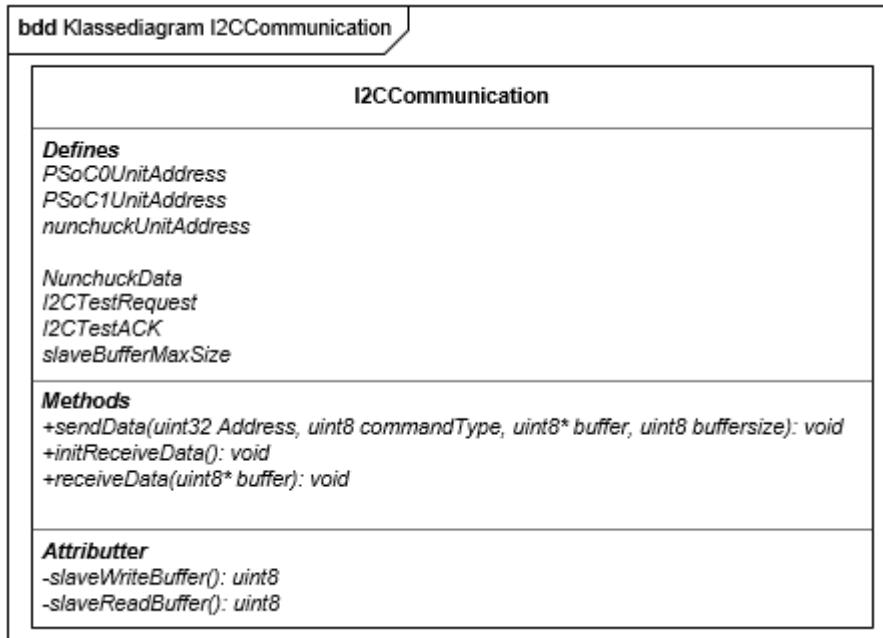
I forbindelse med SPI forbindelsen i systemet, skal der sættes nogle indstillinger på PSoC0, idéat at denne kommunikerer med Devkittet via SPI. PSoC'en sættes som en slave, idéat at det er Devkittet der aflæser og skriver til PSoC0. SCLK sættes til CPHA=1 og CPOL=1. Disse er valgt arbitrært, dog ud fra den forudsætning, at de skal stemme overens med indstillingerne på Devkittet, idéat disse indstillinger beskriver hvornår databit aflæses/sættes i forhold til clock'en. 'Data Rate' sættes til 1Mbps, da dette er en sikker/stabil overførselshastighed. Igen skal denne indstilling være ens for PSoC og Devkit. Den sidste indstilling der sættes, er 'transfer' og 'read' buffer size. Disse er valgt til at være 8-bit, da projektets SPI kommunikationsprotokol ikke har brug for større datamængder. Igen skal denne indstilling være ens for både SPI-master og slave.

#### 4.2.2 I2CCommunication

I dette afsnit vil softwaren der omhandler I2C-kommunikation blive beskrevet. Dette inkluderer et klassediagram, klassebeskrivelser og indstillingerne for I2C kommunikationen på PSoC'en.

##### Klassediagram

På figur 25 ses klassediagrammet for I2CCommunication.



Figur 25: Klassediagram for I2CCommunication klassen

### Klassebeskrivelser

Som det ses på klassediagrammet figur 25 indeholder klassen flere metoder. Disse metoder blive beskrevet her.

**void sendData(uint8 Address, uint8 commandType, uint8\* buffer, uint8 bufferSize)**

Denne metode sender, via PSoC Creators I2C-API, den data der ligger i "buffer" af kommandotypen "commandType" til slaven med adressen "Address".

**void initReceiveData()**

Denne metode initialiserer de to buffers (slaveWrite og slaveRead) der kræves på en I2C-slave, for at kunne gøre bruge af PSoC Creator's I2C-API.

**void receiveData(uint8\* buffer)**

Denne metode venter på at slaveRead bufferen er blevet fyldt. Når dette er sket, bliver slaveRead bufferen kopieret over i "buffer".

### I2C indstillinger på PSoC

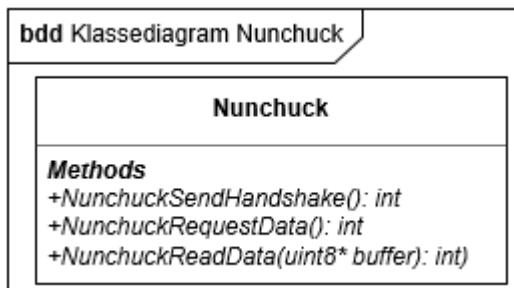
I forbindelse med at have en I2C forbindelse på PSoC'en, er der et par indstillinger der skal sættes. Hver PSoC der gør brug af I2C, sættes til at være en "Multi-master-slave". Dette gøres for at alle I2C enheder kan gøre brug af "I2CCommunication"klassen, idéat denne indeholder funktioner for både master og slave. En anden vigtig indstilling er I2C bussens 'Data Rate'. Denne er fra PSoC creators side sat til 100kbps som standard, hvilket er en passende hastighed for dette projekt. En hver enhed på I2C nettet skal også have en adresse. En tabel for adressefordelingen ses på tabel 5

#### 4.2.3 Nunchuck

I dette afsnit vil softwaren der specifikt omhandler kommunikationen mellem PSoC0 og Nunchucken blive beskrevet. Dette gøres vha. et klassediagram og klassebeskrivelser.

### Klassediagram

På figur 26 ses klassediagrammet for Nunchuck klassen.



Figur 26: Klassediagram for klassen Nunchuck

### Klassebeskrivelser

Metoderne fra klassediagrammet figur 26 vil blive beskrevet i dette afsnit.

#### **int NunchuckSendHandshake()**

Denne metode sender et "handshake" (Se dokumentationen afsnit **INSERT AF-SNIT HER #ref**) til Nunchuck enheden. Handshaket bruges til at "parre" PSoC'en med nunchucken. Metoden returnerer et '0' hvis der opstår en fejl.

#### **int NunchuckrequestData()**

Denne metode sender et 0x00 til nunchuck'en, og derved beder nunchuck'en om at klargøre data til overførsel. Metoden returnerer et '0' hvis der opstår en fejl.

#### **int NunchuckreadData(uint8\* buffer)**

Denne metode bruger PSoC Creator's I2C-API til at læse data fra nunchuck'en (data der blev klargjort fra NunchuckrequestData()). Disse data bliver derefter dekrypteret og gemt i "buffer", så de bliver tilgængelige uden for metodens scope.

I klassediagrammet er der en sektion kaldet "Defines". Disse Defines bruges i implementeringen til forskellige formål. *PSoC0UnitAddress*, *PSoC1UnitAddress* og *nunchuckUnitAddress* bruges til at definere adresserne for I2C-nettets slaver. *NunchuckData*, *I2CTestRequest* og *I2CTestACK* er kommando typer der bruges til at bestemme hvilken kommando type der er blevet sendt/modtaget, og hvor mange bytes der skal forventes at være gemt i databufferen (Se dokumentationens afsnit **INDSÆT REFERENCE TIL DOKUMENTATIONAFSNIT #ref**)

## 4.3 Afkodning af Wii-Nunchuck Data Bytes

Aflæste bytes fra Wii-Nunchuck - indeholdende tilstanden af knapperne og det analoge stick - er kodet når de oprindeligt modtages via I2C bussen. Disse bytes skal altså afkodes før deres værdier er brugbare. Afkodningen af hver byte sker ved brug af følgende formel:

$$\text{AfkodetByte} = (\text{AflæstByte } \text{XOR } 0x17) + 0x17$$

Fra formlen kan det ses at den aflæste byte skal XOR's (Exclusive Or) med værdien 0x17, hvorefter dette resultat skal adderes med værdien 0x17.

## 4.4 Kalibrering af Wii-Nunchuck Analog Stick

De afkodede bytes for Wii-Nunchuck's analoge stick har definerede standardværdier for dets forskellige fysiske positioner. Disse værdier findes i tabel 7

X-akse helt til venstre	0x1E
X-akse helt til højre	0xE1
X-akse centreret	0x7E
Y-akse centreret	0x7B
Y-akse helt frem	0x1D
Y-akse helt tilbage	0xDF

Tabel 7: Standardværdier for fysiske positioner af Wii-Nunchuck's analoge stick

I praksis skal de afkodede værdier for det analoge stick kalibreres, da slør pga. brug gør at de ideale værdier ikke rammes.

I projektet er de afkodede værdier for det analoge stick kalibreret med værdien -15 (0x0F i hexadecimal), altså ser den endelige formel for afkodning samt kalibrering således ud:

$$\text{AfkodetByte} = (\text{AflæstByte XOR } 0x17) + 0x17 - 0x0F$$

## 4.5 Hardwaredesign

På baggrund af BDD'et er der fundet følgende hardwareblokke, der skal udarbejdes:

- Motorstyring
- Tre motorer
- Affyringsmekanisme

Disse beskrives i de følgende afsnit.

### **Motor**

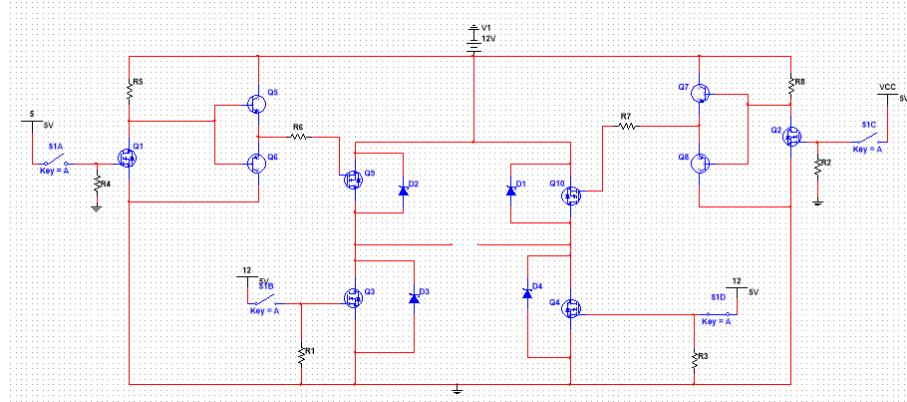
Der er truffet et valg om at bruge en DC-motor i alle tre tilfælde. De to motorer skal bruges til at styre kanonen i to retninger, og den sidste skal bruges i affyringsmekanismen.

#### 4.5.1 Motorstyring

Til at styre de tre motorer er der bygget en H-bro, der skal bruges i tre eksemplarer. To af disse motorer skal kunne styre kanonen, så den ene gør at den kan køre op og ned, og den anden gør, at den kan køre fra side til side. Den tredje skal bruges til at styre affyringsmekanismen.

### **H-bro**

Der blev først designet en H-bro, som bestod af to N-MOSFET's af typen IRLZ44 og to P-MOSFET's af typen ZVP3306. Denne kan ses i bilaget. Det viste sig dog, at den P-MOSFET, der var brugt, var for svag til at kunne trække den strøm, som motoren skulle bruge, hvilket betød, at den blev brændt af. Derfor blev denne H-bro modificeret, så de to P-MOSFET's blev udskiftet med to MOSFET's af typen IRF9Z34N, der kan trække en større strøm.



Figur 27: Kredsløb for H-bro

Tabel 8: Komponentbetegnelser på H-bro

Betegnelse	Komponent
VCC	5V
Q1	IRLZ44(mosfet N-Channel)
Q2	IRLZ44(mosfet N-Channel)
Q3	IRLZ44(mosfet N-Channel)
Q4	IRLZ44(mosfet N-Channel)
Q5	BC547
Q6	BC557
Q7	BC547
Q8	BC557
Q9	IRF9Z34N(mosfet P-Channel)
Q10	IRF9Z34N(mosfet P-Channel)
R1	10kΩ
R2	10kΩ
R3	10kΩ
R4	10kΩ
R5	10kΩ
R6	100Ω
R7	100Ω
R8	10kΩ
D1	IN5819
D2	IN5819
D3	IN5819
D4	IN5819

### MOSFET'er

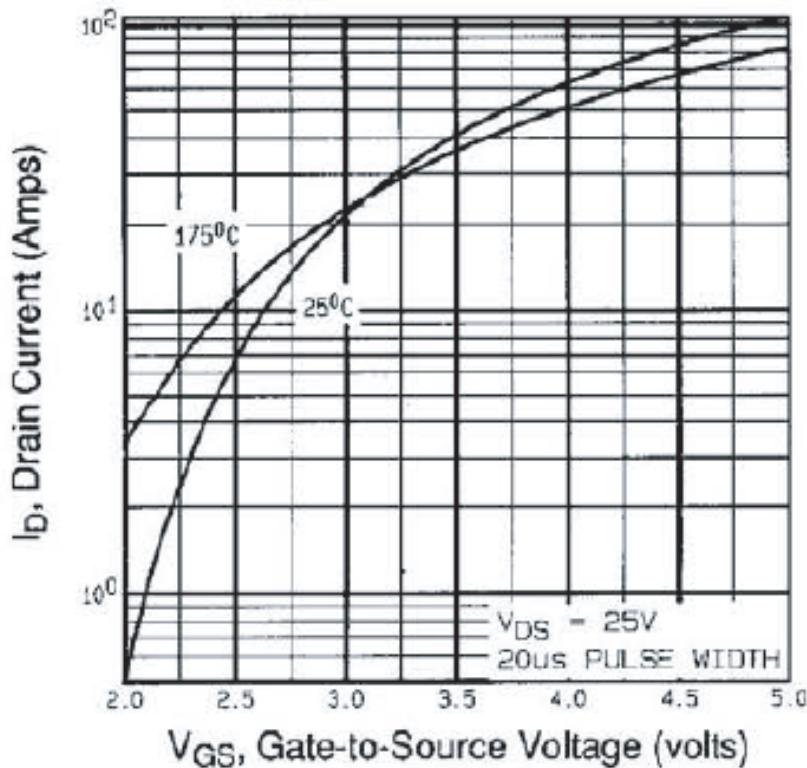
Til at styre motoren er der bygget en H-bro, som består af fire mosfet, hvor to af dem er af typen IRF9Z34N (mosfet P-channel, som er Q9 og Q10 på figur 27) og de to andre mosfet er af typen IRLZ44 (mosfet N-Channel, som er Q3 og Q4 på figur27). Det er valgt at bruge mosfet for at kunne styre H-broen, da

det ved denne er muligt at lukke og åbne for spændingen, og de bliver styret af spænding, i forhold til transistorer, som bliver styret af strøm.

- MOSFET N-kanal

Der er i denne H-bro brugt en N-kanals-MOSFET af typen IRLZ44. Denne MOSFET skal bruges til at trække strømmen fra den tilsvarende P-MOSFET til stel, så motoren kan begynde at køre. Det sker, når der kommer 5V ind på gate-benet.

MOSFET'en fungerer på den måde, at når der kommer positiv spænding ind på gate-benet åbner den, så der kommer forbindelse til stel og når der kommer 0V ind på dette lukker den igen.



Figur 28: Gate-to-Source Voltage REFERENCE TIL DATABLAD HER!!!!

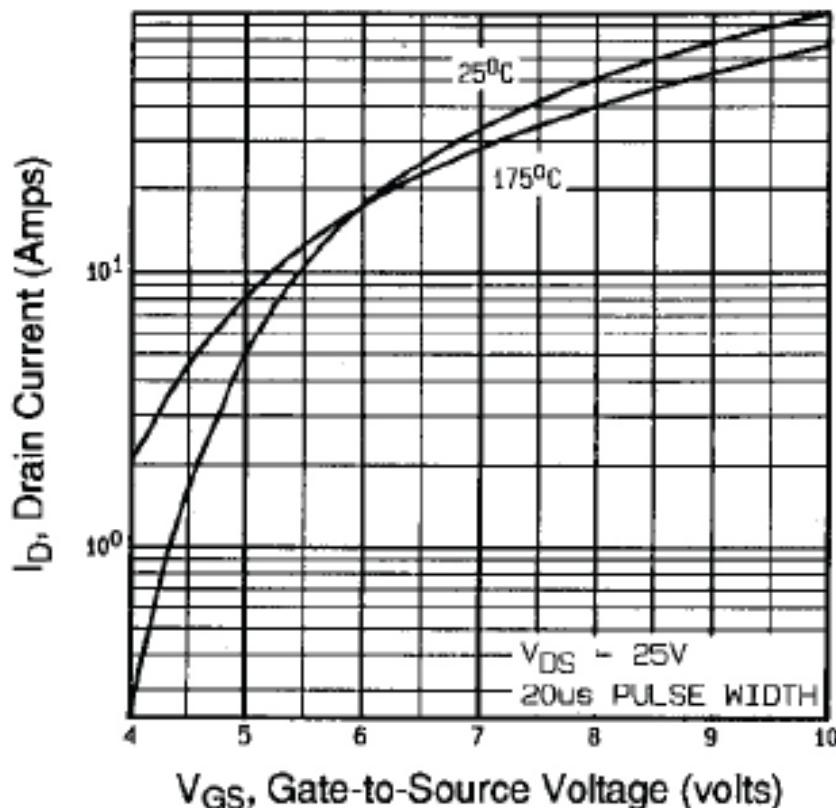
På figur 28 ses det, at når der er en gate-to-source-spænding på 5V, vil der MOSFET'en kunne klare, at der løber en strøm på op til 100A i følge datablad #ref Reference til datablad. Det vil altså ikke komme til at påvirke motoren, da denne kun kan trække en strøm på cirka 0,35A.

- MOSFET P-kanal

Der er valgt at bruge en P-MOSFET af typen IRF9Z34N. Denne MOSFET

skal bruges til at trække de 12V ned til motoren, så denne kan køre. Samtidig sørger den for, at de 12V ikke løber ned til motoren så længe, der ikke er negativ spænding på gate-benet. Denne type MOSFET kan trække en strøm på 6,7A ifølge databladet [#ref Reference til datablad](#). Det vil altså ikke komme til at påvirke motoren, da den kun kan trække en strøm på cirka 0,35A.

For at der kan løbe spænding igennem IRF9Z34N, skal den have en negativ spænding for at åbne og en spænding på over 0V for at lukke.



**Fig. 3 - Typical Transfer Characteristics**

Figur 29: Gate-to-Source Voltage [REFERENCE TIL DATABLAD HER!!!!](#)

På figuren 29 ses det, at når der er en gate-to-source-spænding på 5V, vil derr kunne løbe en strøm på omkring 5A igennem MOSFET'en, hvilket er mere end nok til at få motoren til at køre.

## Dioder

Over fire af mosfetene (Q9, Q10, Q3 og Q4) som set på figur 27 er der sat en diode af typen IN5819. Den skal fungere som beskyttelse af de fire mosfet (Q9, Q10, Q3 og Q4). Det, de gør, er, at de sikrer, at den spænding, som er tilbage i motoren, når man lukker for mosfetene, ikke løber tilbage ind i mosfetene og brænder dem af.

## Modstande

- Pull down modstande:

Der er blevet brugt fire pull-down-modstande (R1, R2, R3 og R4 som set på figur 27). Disse sørger for, at signalet vil blive holdt lavt, når der ikke er trykket, så dette ben ikke står og flyver, så det kan komme til at åbne en mosfet ved en fejl og derved kommer til at brænde en mosfet eller motoren af. Der er valgt en modstand på 10 kOhm, hvilket betyder, at den er lille nok til at trække de små spændinger ned, når der ikke er trykket, og den stor nok til at spændingen ikke løber derved, når der er trykket.

- Andre modstande

- R6 og R7

Grunden til, at R6 og R7(på figur 27), er der, er for at sikre, at transistorernes Absolute Maximum Ratings omkring strømmen, som ikke må overstige 100mA ifølge databladet. (tjek lige om det er rigtig)

- R5 og R8

Grunden til at R5 og R8(kan ses på figur 27) er sat ind i kredsløbet er, at der ifølge databladet kun kan løbe en strøm på omkring 30A igennem den N-MOSFET, der er brugt, Vgs er 10V. Da Vgs kun er sat til 5V, vil MOSFET'en altså ikke kunne klare en alt for stor strøm. Derfor er R5 og R8 sat ind for at forhindre, at MOSFET'en ikke brænder af.

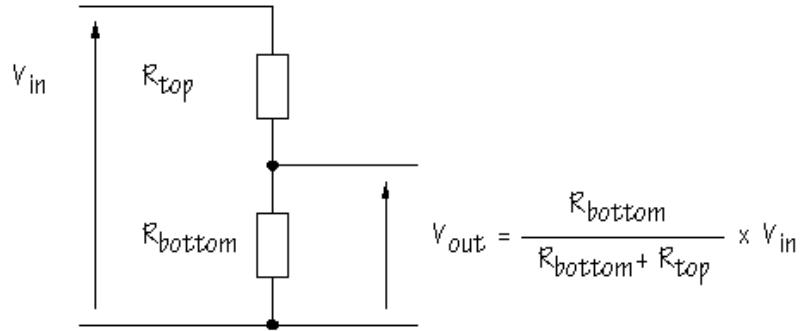
## Transistorer

- Q5 og Q7(kan ses på figur 27) Disse transistorer sidder i kredsløbet, fordi det tager tid for P-MOSFET'en at blive opladt helt, på grund af kondensator effekt imellem benene på mosfet'en, for ellers ville den tag for lang tid om at åbne helt. Det hjælper den disse transistorer til med.
- Q6 og Q8(kan ses på figur 27) Disse to transistorer sidder der for at hjælpe med at lukke P-MOSFET'en igen. Inden Q5 og Q7 blev sat ind tog det tid for at lukke P-MOSFET'en, men da de to blev sat ind, kunne de hjælpe til med at aflade mosfet'ene hurtigere. Derfor sidder de der, for at IRF9Z34N kan lukke hurtigt.

## Potentiometer

Motoren, som styrer platformen, skal kunne bevæge sig frit i intervallet, som er fastsat i kravspecifikationen **#ref Reference til kravspecifikation (ikke**

**funktionelle krav).** For at begrænse denne bevægelse anvendes der et potentiometer. Når motoren bevæger sig, ændres potentiometerets modstandsværdi og dermed ændres spændingsniveauet. Dermed kan positionen af motoren bestemmes ved at se på output spændingen fra potentiometret. For at regne ud hvor langt motoren er, kan man regne ud hvor meget spænding der kommer ud ved at bruge spændingsdeler formlen, som er vist på figur 30



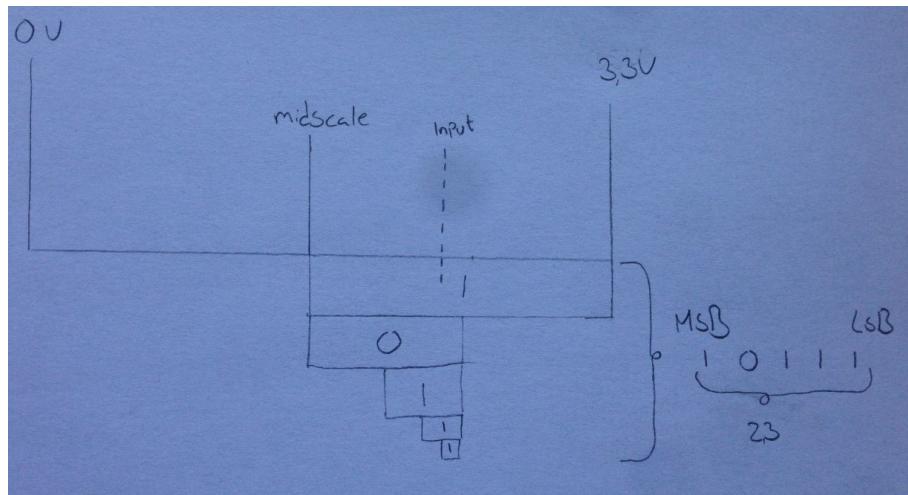
Figur 30: Spændingsdeler formlen for potentiometeret

Hvor man se Rtop og Rbottom, som en modstand, men den bliver delt af midter strengen, som fortæller hvor modstanden skal deles henne og som man kan se på formlen ved side af, der ved kan man regne ud hvad output spændingen fra potentiometret er.

Det potentiometer som bliver brugt i dette projekt er et 47Kohm, som er linæret, som vil sige at spændingen stiger proportionalt med modstanden. Derved kan man finde to værdier som motor ikke må komme over/under, som også holder sig inde for de krav som blev fast sat i kravspecifikationen **#ref Reference til kravspecifikation (ikke funktionelle krav)**. Så hvis de komme over/under en af værdierne, så må motoren ikke mere køre til den pågældende retning. Disse to værdier som blev fundet, er fundet ved hjælp af en vinkelmåler.

## ADC

For at kunne aflæse spændingen på potentiometeret, anvendes en AD converter af typen Sequencing Successive Approximation ADC. En sequencing SAR ADC indeholder et sample-hold kredsløb. Kredsløbet holder på et indgangssignal indtil det næste signal registreres på kredsløbets indgang. Dermed har converteren tid til at bestemme outputværdien. ADC'en er indstillet til midscale og bruger dette punkt til sammenligning med inputsignalet. Hvis signalet er højere end midscale bliver et bit sat til 1 og hvis den er lavere bliver bittet sat til 0. Disse bit bliver gemt i et register. Herefter sættes midpunktet man nu midten i mellem midscale og den øvre halvdel eller nedre halvdel af skalaen og sådan bliver man ved end til alle bit er blevet sat. Så tjekker man registeret for at læse hvilket tal som er fundet, ved at læse fra MSB(most significant bit) til LSB(least significant bit) og det er tal som man har sat på indgangen. På figuren 31 kan man få et bedre overblik over hvordan ADC funger.



Figur 31: Hvordan ADC virker

Det vil sige at jo flere bit man har jo mere præcise bliver det også, men man kan godt risiker at man er 1LSB over/under det resultat man skal have, men for dette projekt gør det ikke noget om man 1 LSB over/under.

Der skal også tages forbehold for at der en forskel på hvad potentiometeret sender ud og hvad ADC'en modtager, for ADC'en kan kun modtage op til ca. 3.3V og potentiometeret kan sende ca. 5V ud.

$$\frac{5V}{3.3V} = 1.515 \quad (1)$$

Så der er en forskel på ca. 1.515 mellem potentiometer og ADC. Så ADC er ca. 1,515 mindre end hvad potentiometer sender ud.

## 5 Modultest

### 5.1 Software

#### 5.1.1 Modultest af Wii-Nunchuck

På PSoC0 er der software til aflæsning af Wii-Nunchuck input data. Følgende afsnit beskriver test af dette software.

Aflæsning af Wii-Nunchuck sker i to skridt, som begge verificeres ved modul test. Først skal der sendes et *Handshake* fra PSoC0 til Wii-Nunchuck for at initialisere data udveksling, og herefter sker data udveksling hver gang PSoC0 sender en anmodning om det. Disse to skridt modultestes her.

##### Test af Wii-Nunchuck Handshake

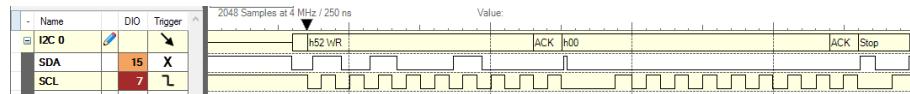
##### Test af data udveksling mellem PSoC0 og Wii-Nunchuck

PSoC0 blev programmeret til kontinuert aflæsning af Wii-Nunchuck. For at verificere data udveksling mellem PSoC0 og Wii-Nunchuck blev I2C bussen målt ved brug af Logic Analyzer fra Analog Discovery.

Data udveksling sker i to skridt. Først sender PSoC0 en byte med værdien 0 (0x00 i hexadecimal). Herefter sker den faktiske aflæsning, PSoC0 aflæser Wii-Nunchuck. Begge skridt testes her.

##### Afsendelse af 0x00 byte

Den første forventede I2C besked er en *0x00* byte fra PSoC0 for at starte en ny aflæsning. På figur 32 ses aflæsningen af I2C bussen på tidspunktet hvor anmodningen til Wii-Nunchuck bliver udført. Dette er en tidslinje læst fra venstre til højre.



Figur 32:

Det kan på figur 32 ses at den første besked der måles er af typen "WR" (Write) til addressen 0x52 (Wii-Nunchuck I2C Slave Addressen). Hertil kommer et tilhørende *ACK* (Acknowledge) fra Wii-Nunchuck. Til sidst sendes dataen Ox00 efterfulgt af et ACK fra Wii-Nunchuck. Til sidst afsluttes I2C transaktionen ved "Stop".

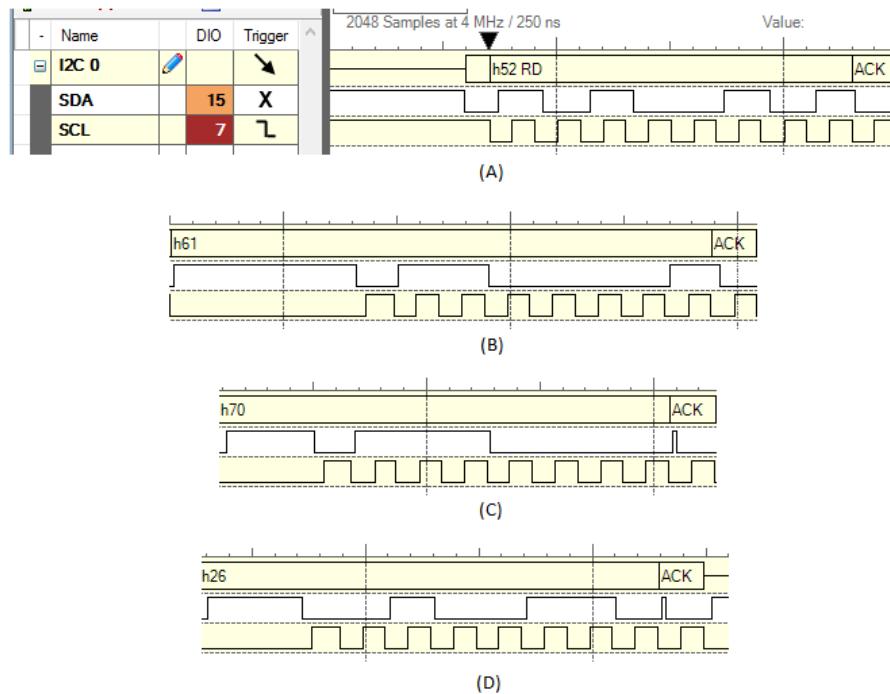
Det kan altså konkluderes at målingen er i overensstemmelse med forventningen om at en 0x00 byte skal sendes til Wii-Nunchuck for opstart af dataudveksling.

##### Aflæsning af Wii-Nunchuck

Efter vellykket afsendelse af 0x00 byten sker den egentlige aflæsning af Wii-Nunchuk input dataen.

Her forventes en række beskeder indeholdende

På figur 33 ses I2C beskederne der bliver udvekslet mellem PSoC0 og Wii-Nunchuck efter vellykket Wii-Nunchuck Handshake.



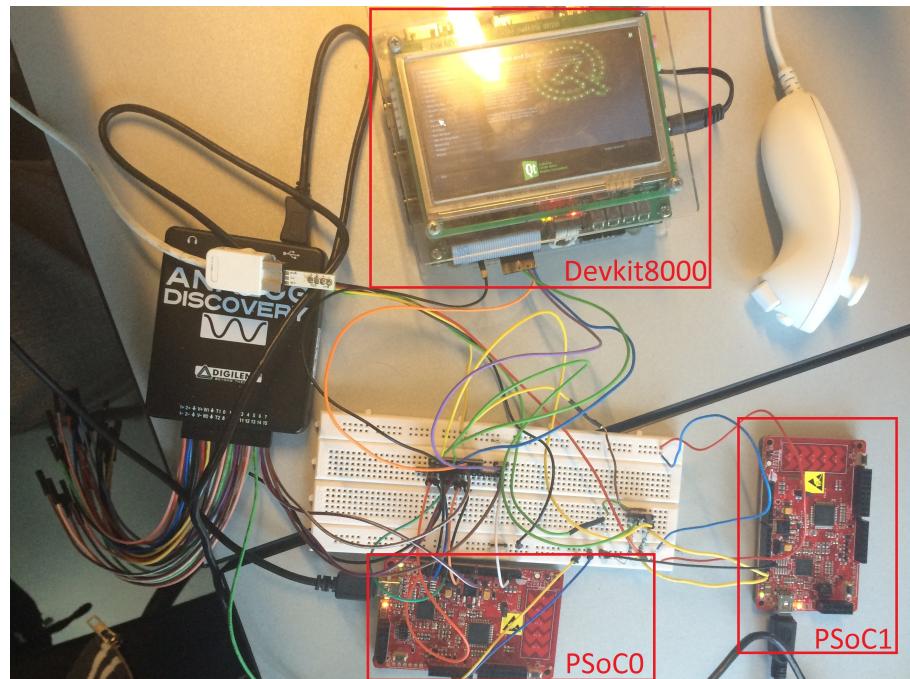
Figur 33: Tidslinje af aflæste I2C beskeder af PSoC0 fra Wii-Nunchuck

### 5.1.2 SPI Protokol

Devkittet kommunikerer over en SPI bus. Kommunikationen følger SPI kommunikations protokollen, som er beskrevet i afsnit 3.7.1. Dette afsnit beskriver test af denne protokol.

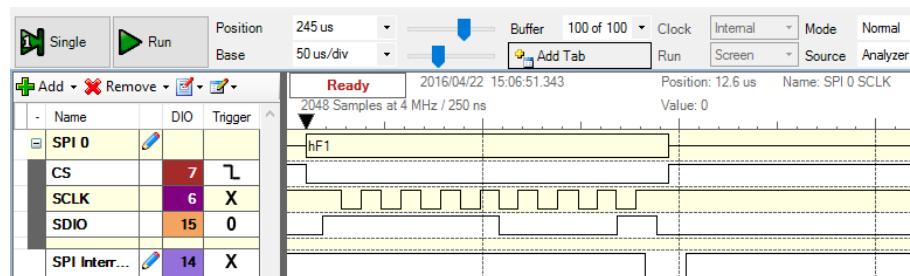
#### SPI Bus Test

Test af SPI bussen udføres i to dele. Første del af testen udføres ved at sende kommandotypen for start af SPI test i terminalen, og derefter verificere den sendte data vha. Analog Discovery. Den anden del af testen, er at aflæse returbeskeden på bussen med Analog Discovery. Målet med denne test, er at læse kommandotypen "SPI\_OK", der indikerer en successful aflæsning af SPI-slaven. På figur 34 ses test opsætningen. Devkit8000, PSoC0s SPI-bus og Analog Discovery er forbundet til hinanden igennem fumlebrættet. PSoC0 og PSoC1's I2C-forbindelser forbides til nunchucken igennem fumlebrættet, adskilt fra SPI-forbindelserne.



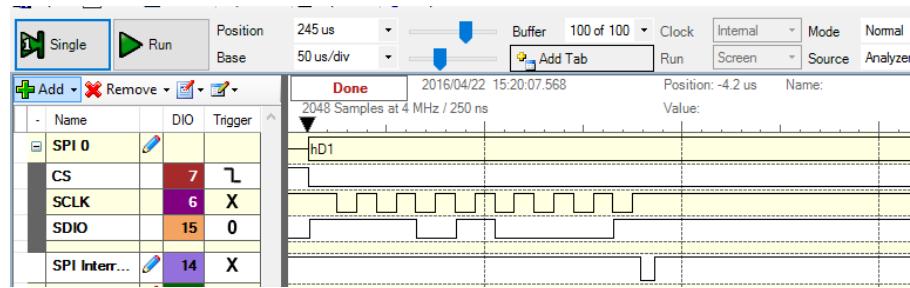
Figur 34: Opsætning for SPI-test

I testen afsendes SPI test kommandotypen, som har værdien 0xF1 jf. tabel 4. For at verificere, at kommandotypen er sendt ud på SPI-bussen korrekt, blev Analog Discovery's 'Logic Analyzer' brugt til at aflæse SPI-bussen. Det var forventet at 0xF1 blev aflæst på bussen. Det afmålte resultat stemte overens med det forventede. Målingen ses på figur 35.



Figur 35: Måling af kommandotype for start SPI test

Hvis SPI-kommunikationen fungerer korrekt, er det forventet at der aflæses 0xD1 på SPI-bussen, da dette betyder 'SPI\_OK' jf. tabel 4. Måling af returbeskeden ses på figur 36.



Figur 36: Måling af returbesked for start SPI test

På figur 36 ses at returbeskeden har den forventede værdi 0xD1, som indikerer at SPI bus testens første del er gennemført uden fejl.

Ud fra testenes resultater kan det konkluderes at implementeringen af SPI protokollen fungerer efter hensigten.

### 5.1.3 I2C Protokol

PSoC0 og PSoC1 kommunikerer over en I2C bus via I2C protokollen beskrevet i afsnit 3.7.2. Dette afsnit beskriver test af denne protokol. Følgende test tager udgangspunkt i kommandotypen *NunchuckData* beskrevet i tabel 6).

#### Test af NunchuckData kommandotype

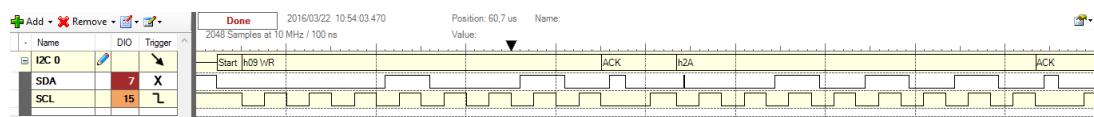
Testen blev udført i to dele. I første del måles I2C bussen ved brug af Analog Discovery's Logic Analyzer; for at verificere at den forventede kommandotype bliver overført via bussen. Anden del verificerer at den overførte data er modtaget korrekt via PSoC Creator's debugger.

#### NunchuckData kommandotype test del 1

I testen afsendes, som nævnt i afsnittets indledning, kommandotypen NunchuckData. Som vist i tabel 6 har denne kommandotype ID'et 0xA2, hvor de efterfølgende 3 data bytes indeholder input dataen fra Wii-Nunchuck.

Det forventede resultat af målingen er at første byte er kommandoentypens ID, som har værdien 0x2A. Kommandoens data - de efterfølgende bytes - verificeres først i anden del, disse indgår altså ikke i følgende måling.

Målingen ses på figur 37.



Figur 37: Tidslinje af målt I2C kommandotype

Det kan ses på figur 37 at I2C overførslen starter med en I2C *write*, som får et successfult acknowledge fra slaven PSoC1. Herefter kan det ses at den næste byte der sendes har værdien 0x2A. Denne byte er kommandoentypens ID, og er altså som forventet 0x2A.

Det kan altså verificeres at kommandoen overføres via I2C bussen. Dataens integritet er dog ikke inkluderet i denne del, og testes i del 2.

### NunchuckData kommandotype test del 2

For at verificere integriteten af den data der sendes mellem PSoC0 og PSoC1, bruges PSoC Creators indbyggede debugger. Igen er det kommandotypen NunchuckData der sendes mellem de to enheder, hvor de medfølgende data bytes fortolkes.

Testen gennemføres ved at fortolke den modtagne data tre gange, hvor nunchucken er i forskellige tilstande (hvilken retning det analoge stik er trykket) i hver test. Værdierne sammenlignes de forventede standardværdier som ses i tabellen på side 3 i [? , I2C Interface with Wii Nunchuck]. Da testene kun er fokuserede på, hvilken retning den analoge stick er presset, er det altså kun receivedDataBuffer[1] (den analoge stick x-akse) og receivedDataBuffer[2] (den analoge pinds y-akse) der er relevante for testen. Når den analoge stick er presset til venstre, forventes det ifølge tabel 7 at receivedDataBuffer[1] er lig 0xE och receivedDataBuffer[2] er 0x7B. Når den analoge stick er presset op, forventes det at receivedDataBuffer[1] er 0x7E och receivedDataBuffer[2] er 0xDF. Når der ikke er noget input på Nunchucken forventes det at receivedDataBuffer[1] er 0x7E och receivedDataBuffer[2] er 0x7B. Målingerne for testene kan ses på figur 38, 39 og 40.

Watch 1	
Name	Value
+/- receivedDataBuffer	0x2000012C (All)
receivedDataBuffer[2]	0x7D 'y'
receivedDataBuffer[3]	0xFB '373'
receivedDataBuffer[1]	0x7F '\177'
Click here to add	

Figur 38: Afmåling af modtager-buffer på PSoC1 efter at have modtaget "NunchuckData"kommando typen. Intet input på Nunchuck'en

Watch 1	
Name	Value
+/- receivedDataBuffer	0x2000012C (All)
receivedDataBuffer[2]	0x7C ' '
receivedDataBuffer[3]	0xC3 '303'
receivedDataBuffer[1]	0x1E '\036'
Click here to add	

Figur 39: Afmåling af modtager-buffer på PSoC1 efter at have modtaget "NunchuckData"kommando typen. Den analoge stick er presset til venstre på Nunchuck'en

Watch 1	
Name	Value
+ receivedDataBuffer	0x2000012C (All) 0
receivedDataBuffer[2]	0xDF 1337' 0
receivedDataBuffer[3]	0xB3 1263' 0
receivedDataBuffer[1]	0x82 1202' 0
Click here to add	

Figur 40: Afmåling af modtager-buffer på PSoC1 efter at have modtaget "NunchuckData" kommando typen. Den analoge stick er presset frem på Nunchuck'en

På figur 39 ses afmålingen af modtager-bufferen når Nunchuckens analoge stick er presset helt til venstre. ReceivedDataBuffer[1] blev aflæst til 0x1E og receivedDataBuffer[2] blev aflæst til 0x7C. ReceivedDataBuffer[1] stemmer overens med forventningerne. ReceivedDataBuffer[2] har en lille afvigelse (oversat til decimaltal blev der målt 124, hvor der forventes 123). Denne afvigelse kan skyldes, at det analoge stick ikke blev presset direkte til venstre, men at den også er blevet presset en smule frem under målingen.

På figur 40 ses afmålingen af modtager-bufferen når Nunchuckens analoge stick er presset frem. ReceivedDataBuffer[1] blev aflæst til 0x82, hvor det var forventet 0x7E. Dette er en afvigelse fra de forventede resultater med 4, og kan skyldes at det analoge stick ikke var helt centreret idet den blev presset frem under målingen. ReceivedDataBuffer[2] blev aflæst til 0xDF, hvilket stemmer overens med de forventede målinger.

På figur 38 ses afmålingen af modtager-bufferen når der ikke er noget brugerinput på nunchucks analoge stick. ReceivedDataBuffer[1] blev aflæst til 0x7F, hvor det forventede resultat var 0x7E. Denne afvigelse kan skyldes at det analoge stick ikke stod helt i midten under målingen (Det analoge stick er lidt "lös" og kan defor godt finde hvile i en position der ikke er fuldt centreret). ReceivedDataBuffer[2] blev aflæst til 0x7D, hvor det forventede resultat var 0x7B. Igen kan denne afvigelse skyldes at det analoge stick ikke var i centrum under målingen.

Ud fra testen kan det konkluderes at implementeringen af I2C-protokollen fungerer efter hensigten.

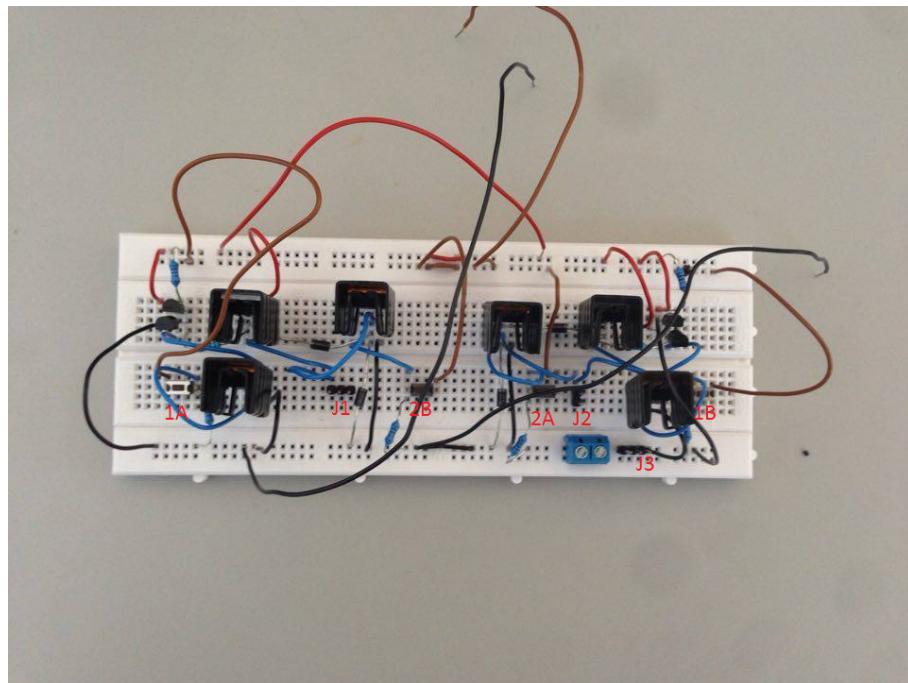
## 5.2 Hardware

### 5.2.1 H-bro

#### Formål

Formålet med denne test er at vise at motor kan styres i begge retninger, ved hjælp af h broen.

#### Overordnet opstilling



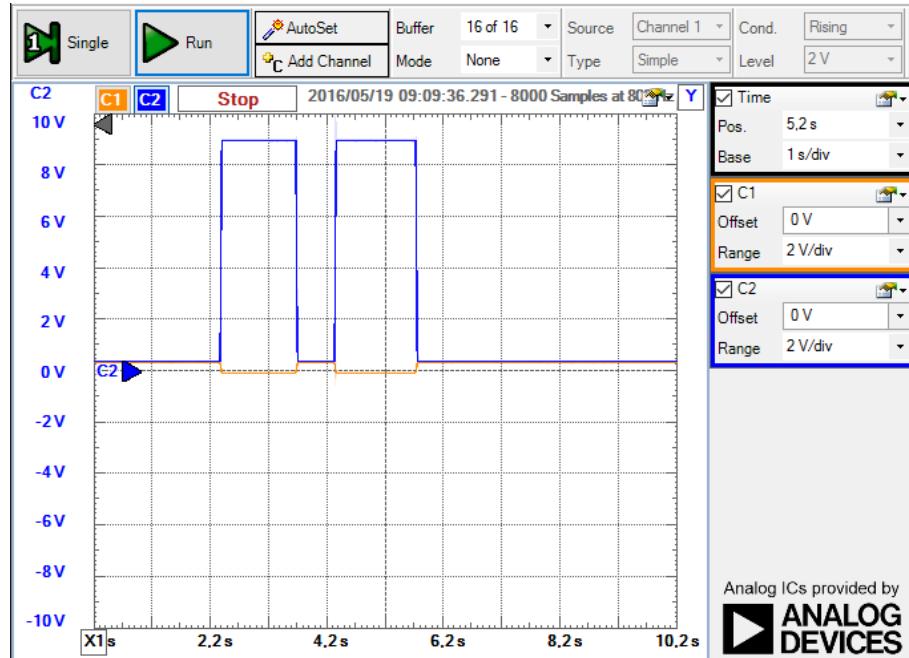
Figur 41: Opstilling af h bro på fumlebræt

- Røde ledninger er 9 V
- Brune ledninger 5 V
- Blå ledninger er spændinger inde imellem komponenter
- Sort er ground
- J1 og J2 er test pin
- 1a og 2a knapper til at styre motor til højre
- 1b og 2b knapper til at styre motor til venstre
- J3 er ground

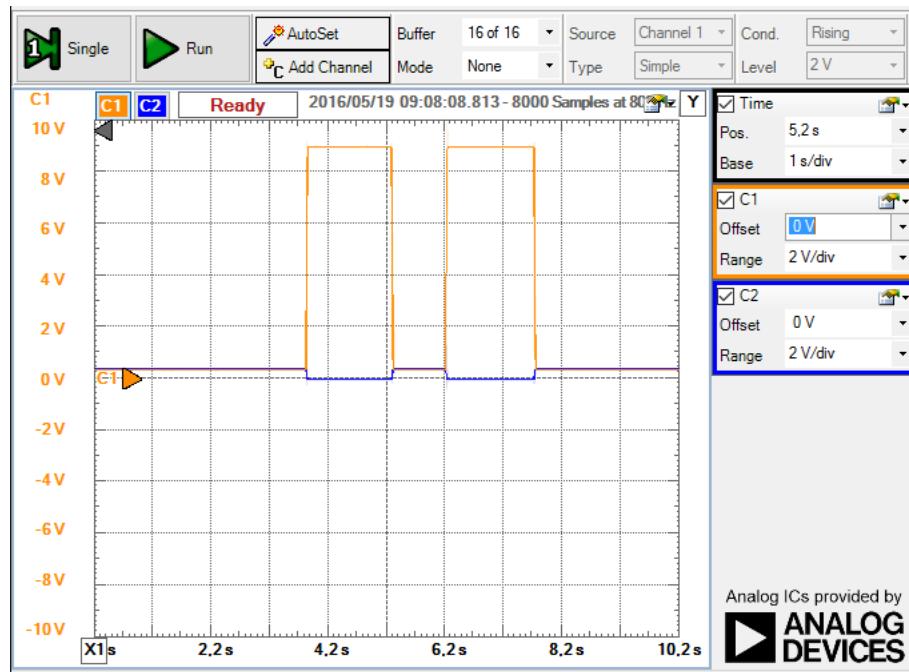
For at teste om h broen kan styre motoren i begge retninger, blev der sat Analog Discovery på de to test pin (J1, J2 og J3) og der blev sat 5V, 9V og ground til. Først del af testen var at tjekke om den kunne køre til højre, det blev gjort ved at trykke på knap 1a og 2a. Anden del af testen var at tjekke om den kunne køre til venstre det blev gjort ved at trykke på knap 1b og 2b.

**Forventet resultat** At når der blev trykket på knap 1a og 2a vil den ene kurv gå høj og den anden vil for blive lav. At når der blev trykket på knap 1b og 2b vil den anden kurv gå høj og den først vil for blive lav.

#### Opnået resultat



Figur 42: Modultest for H-bro, blå går høj



Figur 43: Modultest for H-bro, orange går høj

Som det fremgår på figur 42, kan man se at, i det der bliver åbnet for at

motor kan køre i den ene retning, så stiger den blå op til 9V fordi nu har den fået forbindelse til ground og fået forbindelse til de 9V. Hvor samtidigt at blå går højt, så går den orange lavere, for nu trækker den blå alt spændingen. Det samme sker når man åbner op for at den kan løbe spændingen kan løbe den anden vej, som ses på figur ??, i stedet for at det er den blå som går høj, er det den orange som går høj. Med det kan man se at motor kan køre i begge retninger uden problemer. Der er blevet valgt at der ikke tages billede af at motoren køre, for man ikke se på et billede hvilken vej den køre og overhovedet den køre. Der ses også på figur42 og figur 43 at der noget støj, når orange/blå går højt, men det er ikke noget som kommer til at påvirker vores styring af motoren

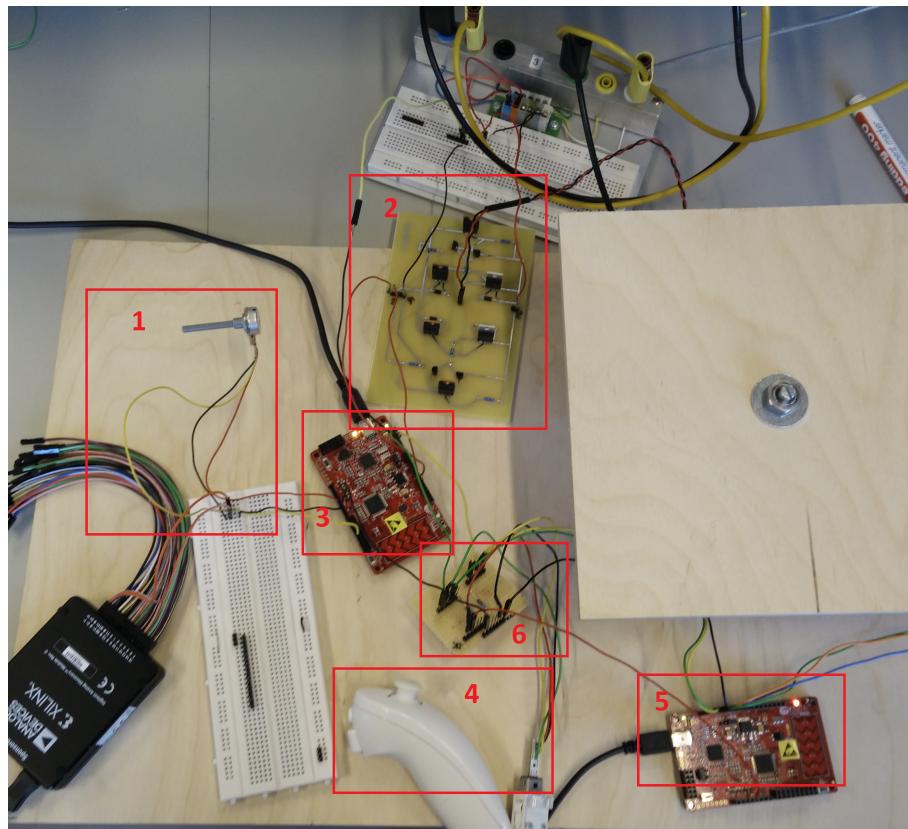
Man kan også på figur se at når den orange/blå går høj, så stiger den meget hurtig, grundens til den gør det var på grund af at, der er blevet sat de to transistor ind, får p mosfet, som gør at den bliver opladt hurtigere, end den ellers ville på grund af der sidder en form for kondensatorer inde i den mosfet, som skal have hjælp med at bliv opladt. Det hjælp kommer fra de transistor som er blevet sat ind før Mosfeten.

### 5.2.2 detektor

#### Formål

Formålet med denne test er at vise at motoren ikke kan køre ud over to fast satte værdier.

#### Overordnet opstilling



Figur 44: opstilling af test for detektor

- Rød firekant 1 er potentiometer
- Rød firekant 2 er h-broen på print
- Rød firekant 3 ”PSoC 1”
- Rød firekant 4 er wii nunchuck
- Rød firekant 5 er ”PSoC 0” hvor der 3 ledninger(lilla, blå og orange) som går til devkit
- Rød firekant 6 hvor der sidder pull up modstande til kommunikationen mellem ”PSoC 1” og ”PSoC 0”
- sort ledning går til ground
- gul ledning er data mellem de forskellige module
- grøn ledning er klokken
- rød ledningen er spænding til h -bro

For at teste om motor kunne køre i begge retninger når den er i mellem 900mV og 2000mV og når den kommer over 2000mV eller under 900mV skal den ikke

kunne køre i en af retningerne. For at tjekke hvilken værdi som blev brugt, blev der sat en analog discovery på potentiometeret og på ADC, blev der debugget og så for at tjekke om motoren kun kunne køre i en retning blev det en visueltest.

Første del af testen består i at tjekke om motoren kan kun køre mod venstre, når potentiometeret er drejet til en værdi under 900mV. Hvor der blev sat på hvilken værdi potentiometer gav og hvad ADC'en fik og en visueltest af om motoren kun kunne køre i en retning.

Anden del af testen består i at tjekke om motoren kunne køre begge vej, når potentiometeret er drejet til en værdi mellem 900mV og 2000mV. Hvor der blev sat på hvilken værdi potentiometer gav og hvad ADC'en fik og en visueltest af om motoren kun kunne køre i begge retninger.

Tredje del består i at tjekke om motoren kan kun køre mod højre, når potentiometeret er drejet til en værdi over 2000mV. Hvor der blev sat på hvilken værdi potentiometer gav og hvad ADC'en fik og en visueltest af om motoren kun kunne køre i den anden retning.

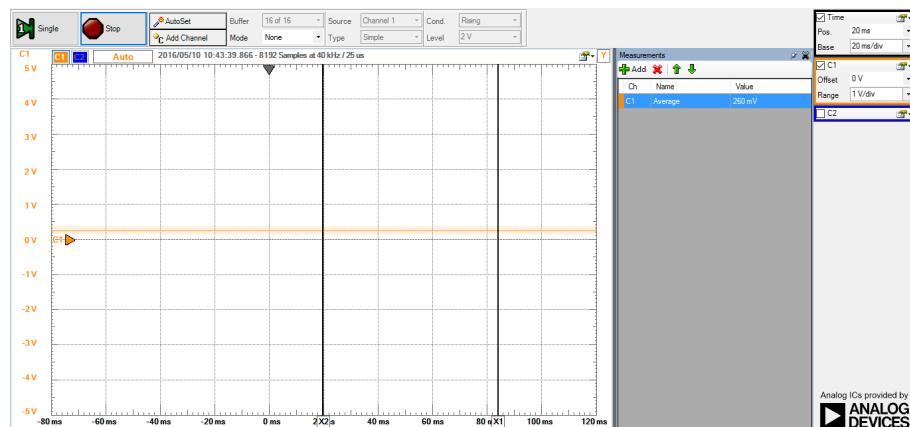
### Forventet resultat

Første del: at når der en værdi under 900mV bliver sendt ind, vil motoren kun kunne køre mod venstre.

Anden del: at når der en værdi mellem 900mV og 2000mV bliver sendt ind, vil motoren kunne køre i begge retninger.

Tredje del: at når der en værdi over 2000mV bliver sendt ind, vil motoren kun kunne køre mod højre.

### Opnået resultat

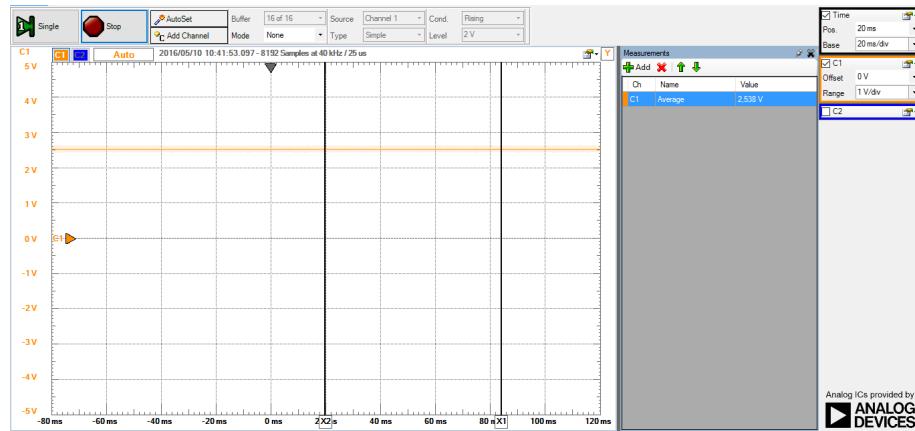


Figur 45: måleresultat fra analog Discovery, ved først del af test

Watch 1			
Name	Value	Address	Type
adc	187	0x20000FA8 (All)	float

Figur 46: måleresultat fra ADC, ved først del af test

Først del: Som det fremgår på figur ?? bliver der sendt fra potentiometeret en spænding på 260mV og på figur ?? kan man se at ADC'en modtog en værdi på 187mV, man skal huske på at der en forskel på 1.515 i mellem potentiometeret og ADC #ref Reference til DesignOgImplementering . Som forventet kunne motoren kun køre mod venstre

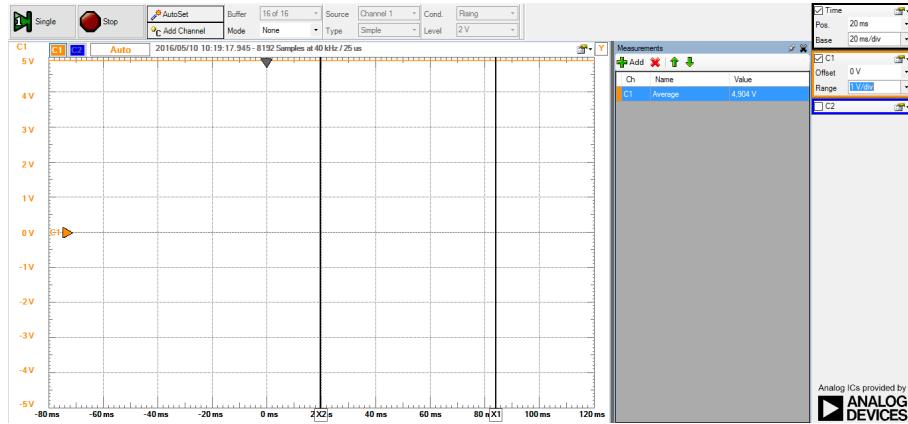


Figur 47: måleresultat fra analog Discovery, ved anden del af test

Watch 1		Address	Type	Radix
Name	Value			
adc	1595	0x20000FA8 (All)	float	Default

Figur 48: måleresultat fra ADC, ved anden del af test

Anden del: Som det fremgår på figur ?? bliver der sendt fra potentiometeret en spænding på 2538mV og på figur ?? kan man se at ADC'en modtog en værdi på 1595mV, man skal huske på at der en forskel på 1.515 i mellem potentiometeret og ADC #ref Reference til DesignOgImplementering . Som forventet kunne motoren kun køre i begge retninger.



Figur 49: måleresultat fra analog Discovery, ved tredje del af test

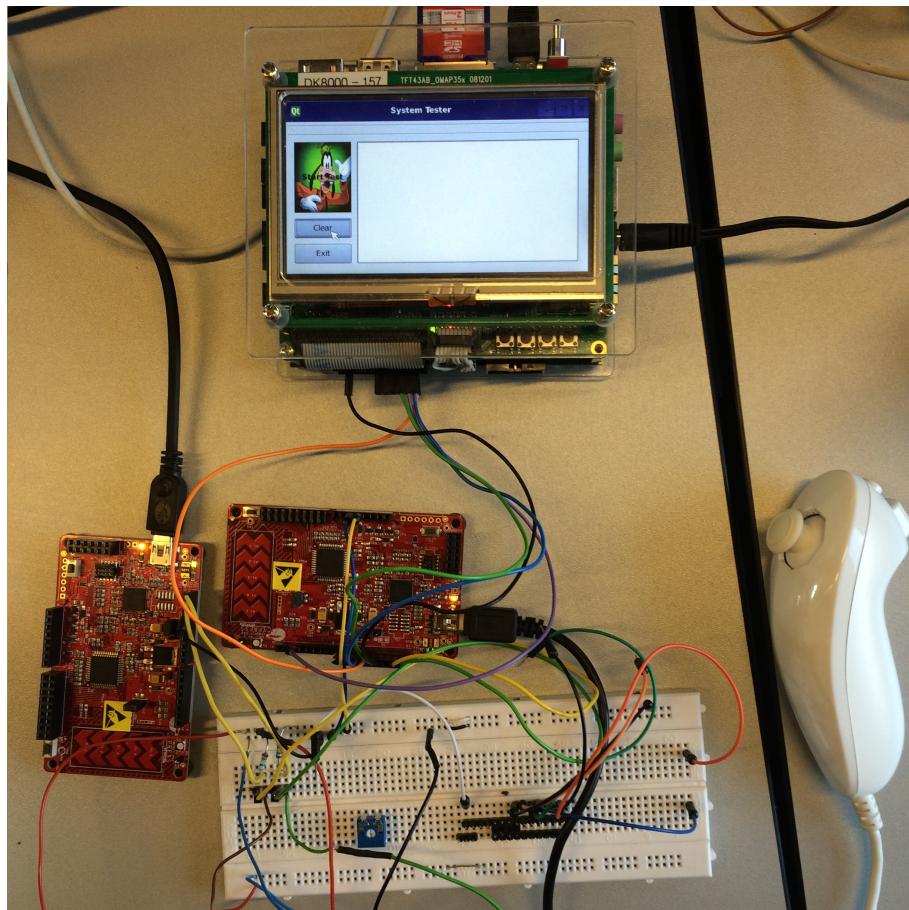


Figur 50: måleresultat fra ADC, ved tredje del af test

Tredje del: Som det fremgår på figur ?? bliver der sendt fra potentiometeret en spænding på 4909mV og på figur ?? kan man se at ADC'en modtog en værdi på 3252mV, man skal huske på at der en forskel på 1.515 i mellem potentiometeret og ADC #ref Reference til DesignOgImplementering . Som forventet kunne motoren kun køre mod højre.

### 5.3 Integrationtest - Use case 2

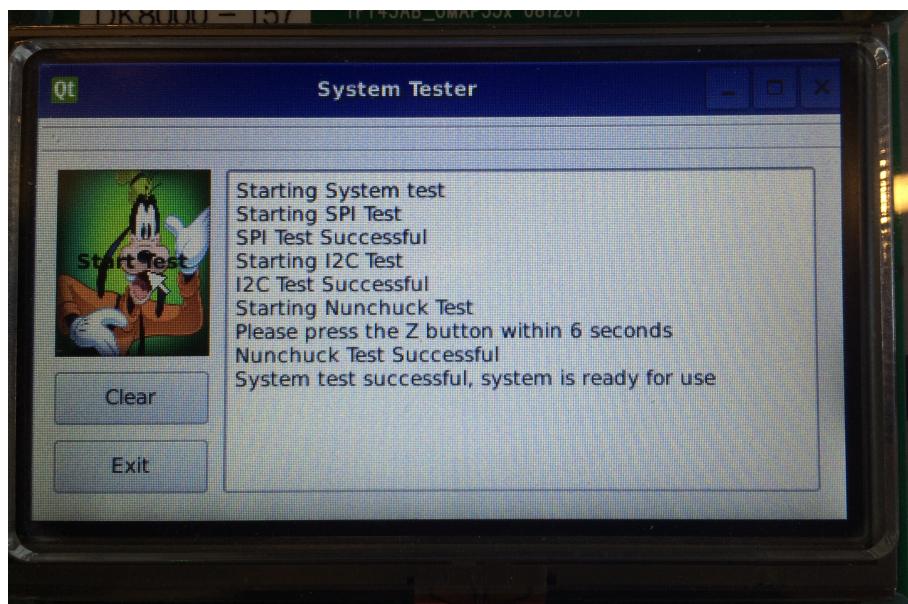
For at verificere at use case 2 fungerer når det sammensættes til en enkelt enhed, er der lavet en integrationstest. Testen er lavet ud fra et 'black-box' princip, hvilket vil sige, at der kun evalueres ud fra systemets funktionalitet, og ikke på den interne struktur. Testen udføres ved, at klikke på 'Start-test', og bagefter observeres udskriften på terminal-vinduet på brugergrænsefladen. På figur 51 ses opstillingen for integrationstesten.



Figur 51: Integrationstest opstilling

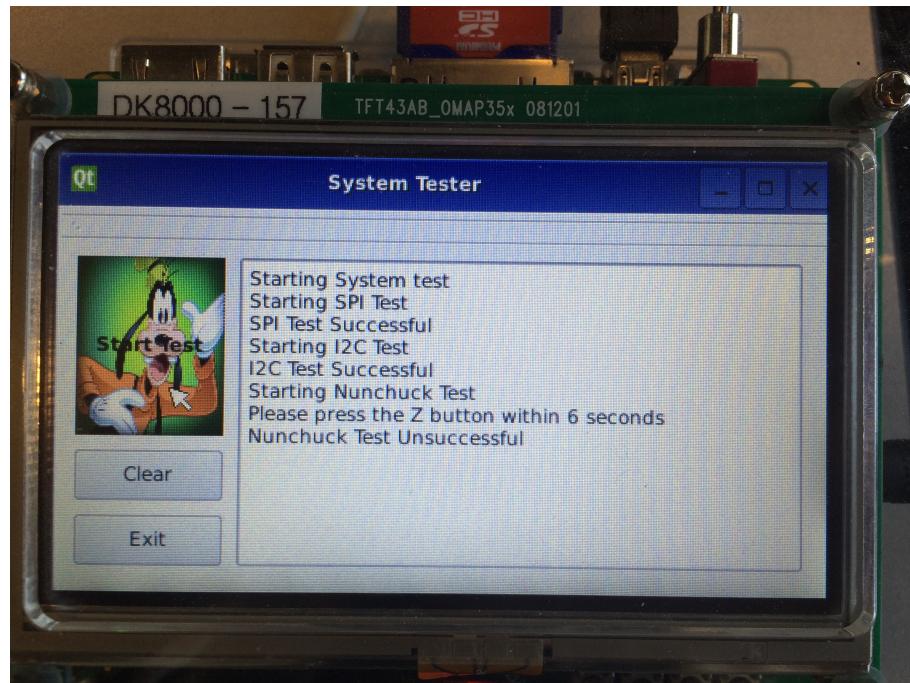
Opstillingen viser, at Nunchuck'en og de to PSoC's er forbundet til samme I2C-netværk via fumlebrættet. PSoC0(PSoC'en til højre) er også forbundet til DevKit8000 via SPI, ledt igennem fumlebrættet. På Devkittet ses brugergrænsefladen, hvor testen initieres ved at klikke på "Start test". Brugergrænsefladen har også et terminalvindue, hvor testens status bliver udprintet.

Selv testen gennemføres, ved at der klikkes "Start test" på brugergrænsefladen, og derefter følges evt. instruktioner der vises på brugergrænsefladen. Det forventes, at efter testen, vil brugergrænsefladen fortælle, at testen blev gennemført uden fejl, og systemet er klar til brug. Figur 52 viser brugergrænsefladen efter endt system test.



Figur 52: Resultat af integrationstesten

Som det ses, er resultatet af testen som forventet, og testen er gennemført. Under testen bliver brugeren bedt om at trykke på nunchucken's 'z' knap. Hvis brugeren ikke klikker på knappen indenfor det angivne stykke tid, forventes det at testen vil fejle. Figur 53 viser netop dette.



Figur 53: Resultat hvis man ikke klikker på nunchuck

Som det ses, fejlede testen, hvilket stemte overens med det forventede resultat. Derved kan det konkluderes at systemet opfører sig som forventet.

#### 5.4 Accepttest

## **6 Referencer**