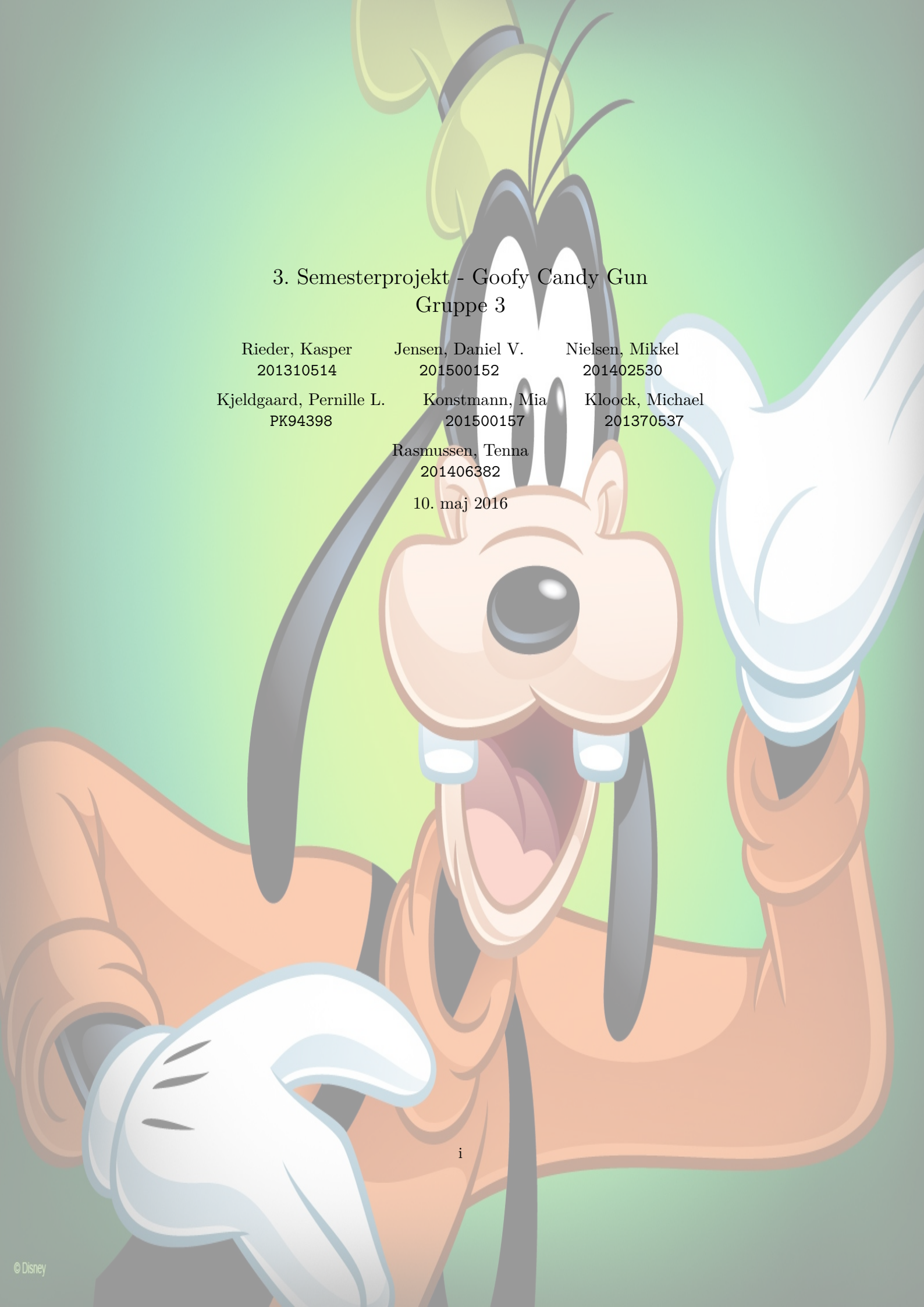


3. Semesterprojekt - Goofy Candy Gun
Gruppe 3

Rieder, Kasper
201310514

Jensen, Daniel V.
201500152

Nielsen, Mikkel
201402530

Kjeldgaard, Pernille L.
PK94398

Konstmann, Mia
201500157

Kloock, Michael
201370537

Rasmussen, Tenna
201406382

10. maj 2016

Indhold

Figurer

1 Resumé

2 Abstrakt

3 Indledning

4 Termliste

5 Projektformulering

5.1 Indledning

Ønsket med dette projekt er at udvikle en mekanisk enhed der kan styres med en håndholdt controller. Med dette udgangspunkt blev forskellige muligheder undersøgt som inspirationskilde, hvor valget herefter faldt på en kanon til affyring af slik. Ideen med en kanon der affyrer slik er ikke original, som det kan ses ud fra projektet *"The Candy Cannon"* som er fundet på YouTube: **https://www.youtube.com/watch?v=VgZhQJQnnqA**. Til forskel fra *The Candy Cannon* og lignende projekter som affyrer projektiler uden et egentlig formål, vil der i dette projekt blive udviklet en kanon til brug i et spil. Kanonen affyres og styres af spillerne via en controller. Altså skal projektet ende med en kanon som indgår i et to personersspil, f.eks. til brug ved fester og andre sociale begivenheder.

I dette projektet skal der altså udvikles en slikkanon til spillet *Goofy Candygun 3000*. Denne slikkanon skal kunne skyde med slik. Dette kunne for eksempel være M&M's eller Skittle's.

Goofy Candygun 3000 er et spil til to personer, hvilket gør det velegnet i sociale sammenhænge. Spillet går ud på at opnå flest point ved at ramme et mål. Hver spiller får et bestemt antal skud. Efter skuddene er opbrugt, er vinderen spilleren med flest point.

Et typisk brugerscenarie er, at spillerne bestemmer antallet af skud for runden. Når dette er gjort, er spillet igang. Herefter går Wii-nunchucken på skift mellem spillerne for hvert skud. Dette fortsættes indtil skuddene er opbrugt. Vinderen er spilleren med flest point. Spillets statistikker vises løbende på brugergrænsefladen.

Det endelige produkt omfatter:

- En brugergrænseflade, hvor spilstatistikker fremvises til deltagerne. Dette er blandt andet:

  Pointvisning

  Kanonens vinkel

  Antal resterende skud

- En motor, der drejer kanonen om forskellige akser

  Dette styres med en Wii-nunchuck

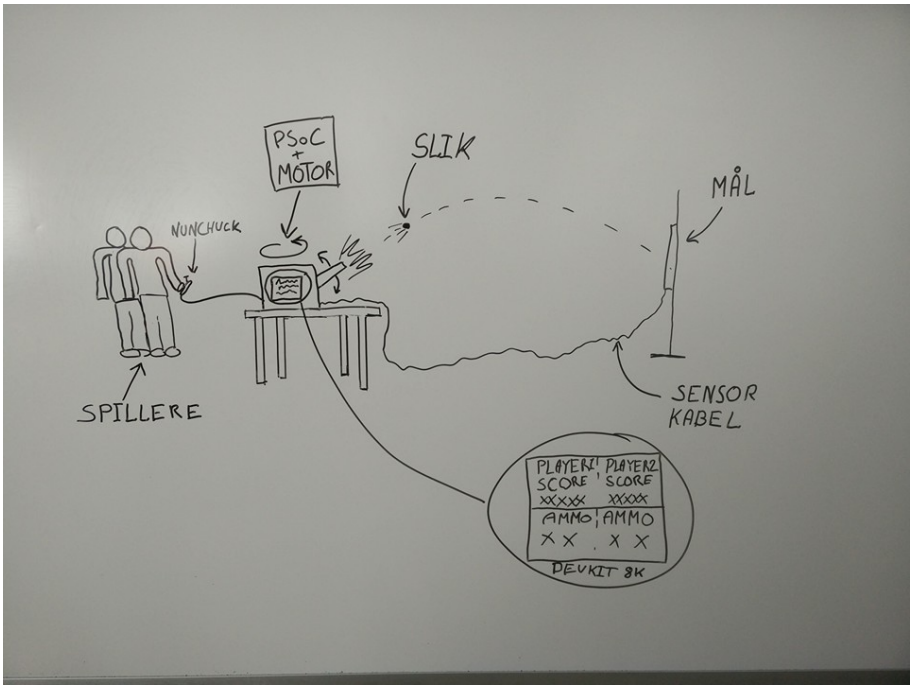
- Et mål, der kan registrere spillernes skud

Med baggrund i krav stillet af organisationen IHA, bliver produktet udviklet med følgende funktioner:

- DevKit 8000 som den indlejrede linux-platform til spillets grafiske brugergrænseflade.
- Motorer til styring af kanonen.
- Sensorer.
- PSoC 4, anvendt til styring af motorer og kommunikation med sensorer.

5.2 Rigt Billede

På figur 1 ses et rigt billede af det ønskede produkt. Billedet beskriver brugerscenariet.



Figur 1: Rigt Billede af det endelige produkt

5.3 MoSCoW

I forbindelse med projektet gøres der brug af MoSCoW-princippet (**https://en.wikipedia.org/wiki/MoSC** for at prioritere hvilke krav, der skal være implementeret ved projektets afslutning. Ifølge MoSCoW er prioriteringerne 'Must have', 'Should have', 'Could have' og 'Won't have'. Kravene er, som følger:

- Produktet must have:
  - En motor til styring af kanonen
  - En grafisk brugergrænseflade til visning af statistikker
  - En Wii-nunchuck til styring af motoren
  - En kanon med en afskydningsmekanisme
- Produktet should have:
  - Et mål til registering af point
  - En lokal ranglistestatistik

- Produktet could have:

  Partymode-indstilling til over to spillere

  Trådløs Wii-nunchuckstyring

  Afspilning af lydeffekter

- Produktet won't have:

  Et batteri til brug uden strømforsyning

  Online ranglistestatistik

5.4 Opdeling af gruppen

I løbet af projektet vil projektgruppen blive opdelt i to hovedgrupper - 'hardware' og 'software'. Softwaregruppen vil desuden stå for grænsefladeprogrammering mellem software og hardware. Disse grupper vil have til ansvar at designe og implementere hhv. hardware og software til projektet. Hardwaregruppen vil bestå af de personer, der læser til elektroingeniør (Mikkel Nielsen og Pernille Kjeldgaard). Softwaregruppen vil bestå af de personer, der læser til IKT-ingeniør (Kasper Rieder, Michael Kloock, Tenna Rasmussen, Mia Konstmann og Daniel Jensen).

6 Krav

Ud fra projektformuleringen er der formuleret en række krav til projektet. Disse indebærer to use cases og et antal ikke-funktionelle krav.

Indsæt figur her.

6.1 Aktørbeskrivelse

På figur 1 ses aktør-kontekst-diagrammet for systemet. På dette ses det, at der i systemet er en enkelt aktør og to use cases. Den ene aktør er brugeren som samtidig er den primære aktør. Brugeren kan sætte de to use cases i gang, hvilket indebærer, at kunne spille Goofy Candygun 3000 og sætte en test af systemet i gang.

6.2 Use case beskrivelse

6.2.1 Use case 1

Dette er den rigtige use case. Det er denne, der køres, når der skal spilles et spil på Goofy Candygun 3000, og den initieres af brugeren. Det første der sker i use case er, at brugeren skal vælge, hvilken type spil han/hun gerne vil spille. Det betyder, at det skal bestemmes, om det skal være et enpersonsspil, topersonersspil eller om det skal være party mode. Herefter skal der vælges, hvor mange skud et spil skal vare og disse skal puttes i magasinet. Når dette er gjort kan spillet gå i gang. Brugeren indstiller kanonen med Wii-nunchuck og affyrer den. Herefter lader systemet et nyt skud og samme procedure gentages. Til slut vises information om spillet på brugergrænsefladen, brugeren afslutter spillet ved at trykke på knappen på brugergrænsefladen og denne vender tilbage til starttilstanden.

6.2.2 Use case 2

Use case 2 skal kun bruges til at teste systemet og dets kommunikationsprotokoller. Den skal bruges til at finde ud af om systemet virker og hvis det ikke gør, hvad det så er, der er gået galt og hvor det er gået galt. Use casen initieres af brugeren, hvor der herefter bliver sendt informationer ud til de forskellige dele af systemet, som så gerne skal sende svar tilbage igen. Hvis det sker for alle dele, vil brugergrænsefladen til sidst meddele, at use casen er gennemført.

6.3 Ikke-funktionelle krav

De ikke-funktionelle krav siger noget om, hvordan systemet skal bygges, og hvilke specifikationer det skal have. I dette tilfælde er der udarbejdet syv krav, hvor der er et der siger noget om, hvordan kanonen skal kunne drejes. Nogle siger noget om kanonens størrelse, hvor der også er specificeret, hvor stort projektilet den skal affyre må være og et siger, hvor langt den skal kunne skyde. Der er et, der siger, hvor lang tid det må tage at skyde et projektil afsted og hvor hurtig den skal være til at lade kanonen igen. Endelig er der et, der specificerer, hvordan den grafiske brugergrænseflade skal se ud. Deciderede værdier for de enkelte krav kan læses i bilag.

7 Projektafgrænsning

Ud fra MoSCoW-princippet er der udarbejdet en række krav efter prioriteringerne 'must have', 'should have', 'could have' og 'won't have'. Dette er for at gøre det tydeligt, hvad der er vigtigt, der bliver udviklet først, og hvad der godt kan vente til senere. Disse krav er som følger:

- Produktet must have:
  - En motor til styring af kanonen
  - En grafisk brugergrænseflade til visning af statistikker
  - En Wii-nunchuck til styring af motoren
  - En kanon med en affyringsmekanisme
- Produktet should have:
  - Et mål til registering af point
  - En lokal ranglistestatistik
- Produktet could have:
  - En partymode-indstilling til over to spillere
  - Trådløs Wii-nunchuckstyring
  - Afspilning af lydeffekter
- Produktet won't have:
  - Et batteri til brug uden strømforsyning
  - Online ranglistestatistik

I forrige afsnit blev 'must have'-kravene beskrevet. Det er de krav, der har højst prioritering for at blive implementeret i projektet. Det vil altså sige, at kravene under punkterne 'should have' og 'could have' har lavere prioritet. For at kravene under punktet 'must have' er opfyldt, skal use case 1, Spil Goofy Candygun 3000 implementeres. Dette kan dog først lade sig gøre, når use case 2, Test Kommunikationsprotokoller er implementeret. Derfor blev prioriteringen i dette projekt, at use case 2 skulle implementeres til fulde, inden der kunne startes på at implementere use case 1. Det havde dog også høj prioritet at have et produkt, der kunne skyde, hvilket betød, at selvom affyringsmekanismen ikke indgår i use case 2, blev det alligevel prioriteret højt at få implementeret denne i systemet.

8 Metode

I arbejdet med projektet er det vigtigt at anvende gode analyse- og designmetoder. Dermed er det muligt at komme fra den indledende idé til det endelige produkt med lavere risiko for misforståelser og kommunikationsfejl undervejs. Det er også en stor fordel, hvis de metoder, der anvendes, er intuitive og har nogle fastlagte standarter. Det gør det muligt for udenforstående at sætte sig ind i, hvordan projektet er udviklet og designet. Dermed bliver projektet og dets produkt i højere grad uafhængigt af enkeltpersoner, og det bliver muligt at genskabe produktet. I forbindelse med udviklingen af Goofy Candygun 3000 er der anvendt use cases til at beskrive brugen af produktet. Gennem analyse af use cases er det muligt at skabe overblik over de nødvendige funktioner, som produktet skal have for at opfylde projektformuleringen. Som modellerings sprog til diagrammer er der anvendt SysML, som netop har faslagte standarter, der gør de forskellige diagrammer lettere at sætte sig ind i, og som dermed bidrager til at give en god forståelse for opbygningen og designet af produktet.

8.1 SysML

SysML er et modelleringssprog, der bygger videre på det meget udbredte modelleringssprog, UML. Men hvor UML hovedsagligt er udviklet til brug i objekt orienteret software udvikling, er SysML i højere grad udviklet med fokus på beskrivelse af både software- og hardwaresystemer. Det gør det særdeles velegnet til dette projekt, som netop består af både software- og hardwaredele. Det er derfor også anvendt i store dele af arbejdet med analyse og design af produktet.
SysML diagrammer kan overordnet inddeles i adfærdsdiagrammer og strukturdiagrammer. I analysen er der hovedsageligt anvendt adfærdsdiagrammerne; use case diagram og sekvensdiagram. Use case diagrammet giver et godt overblik over aktørenes roller i forhold til de forskellige anvendelser af produktet. Sekvensdiagrammerne er brugt analytisk til at udlede de metoder, de forskellige klasser og software enheder skal indeholde. Desuden er sekvensdiagrammer gode til at specificere grænseflader mellem delsystemer, da de afdækker kommunikationen mellem de forskellige enheder. I forhold til strukturdiagrammer benyttes "Internal Block Diagram" (IBD) og "Block Definition Diagram" (BDD). I BDD'et nedbrydes systemet i blokke, så der skabes et overblik over, hvad systemet består af hardwaremæssigt. IBD'et giver et overskueligt indblik i forbindelserne mellem de forskellige blokke og grænsefladerne mellem delene. Tilsammen danner de et godt grundlag for det videre design af produktet.
Når SysML benyttes til at beskrive software design, erstatter elementerne fra SysML blokdiagrammer det udbredte UML klassediagram. Der anvendes også et blokdiagram til at overskueliggøre de konceptuelle klasser i domænemodellen. Ud fra domænemodellen udledes klassediagrammer, som beskriver strukturen i software arkitekturen og designet.

8.2 Software Design Principper

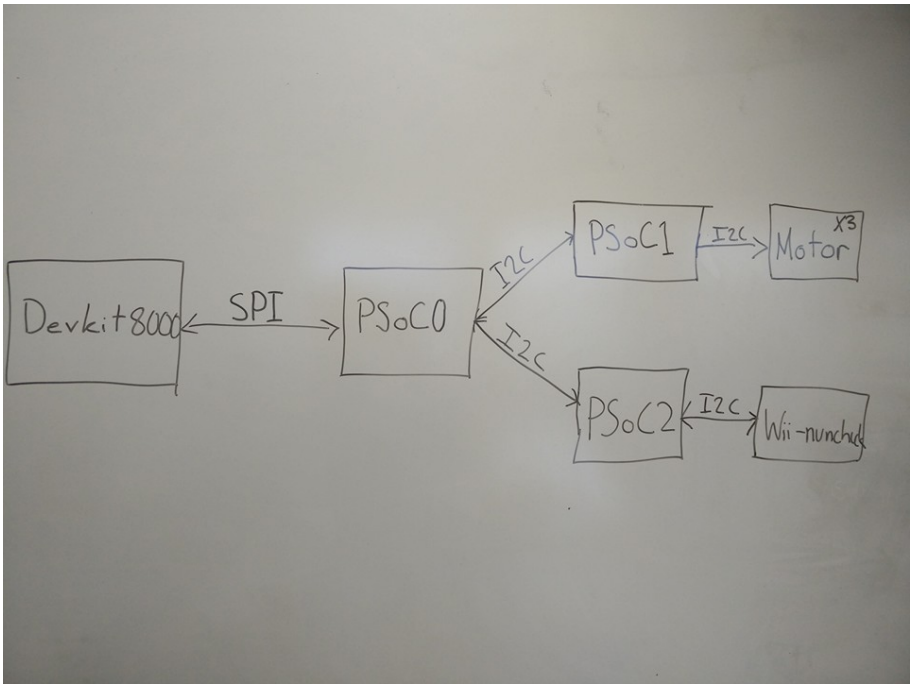
Cohesion, coupling

8.3 Designmetode

Afvigelser fra traditionelt vandfaldsmodel

9 Systembeskrivelse

Goofy Candygun 3000 er et underholdningssystem, som kan styres efter brugerønsker. Slikkanonen fungerer ved, at en bruger starter spillet på brugergrænsefladen på devkittet, derefter er spillet i gang. For at ændre sigteretningen for kanonen anvendes en Wii-nunchuck. Wii-nunchuck sender analogstikkets koordinater, via I2C kommunikation, til PSoC2. PSoC2 tolker dette data og sender kommandoen til PSoC0. PSoC0 videresender denne kommando til PSoC1. PSoC1 tolker denne kommando og udsender et PWM signal, som bevæger motorene efter den sendte kommando. Når der affyres et projektil sendes data om et knap tryk på Wii-nunchuck fra PSoC2 til PSoC1 via PSoC0. På figur 2 ses den overordnede struktur af systemet.



Figur 2: Illustation af Goofy Candy 3000 overordnet struktur

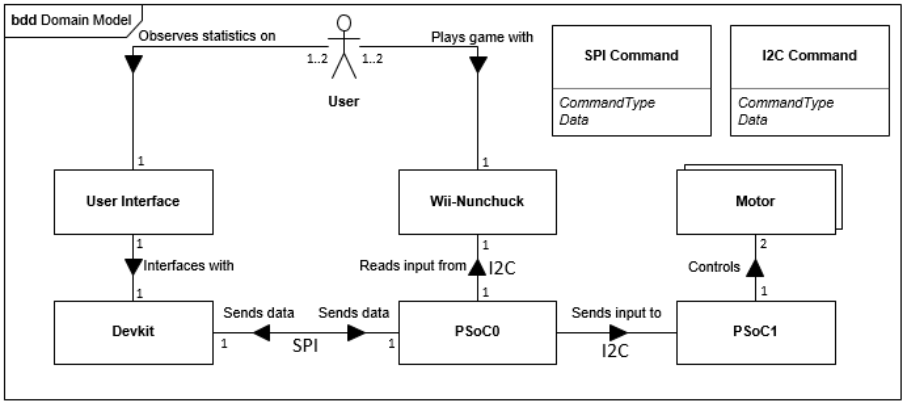
HUSK BILLEDE OG FORKLARING AF ENDELIG UDSEENDE.

10 Systemarkitektur

Dette afsnit præsenterer systemets arkitektur i en grad der gør det muligt at forstå sammensætningen mellem dets hardware og software komponenter.

10.1 Domænemodel

På figur 3 ses domænemodellen af systemet. Denne har til formål at præsentere forbindelserne mellem systemets komponenter, samt dets grænseflader.

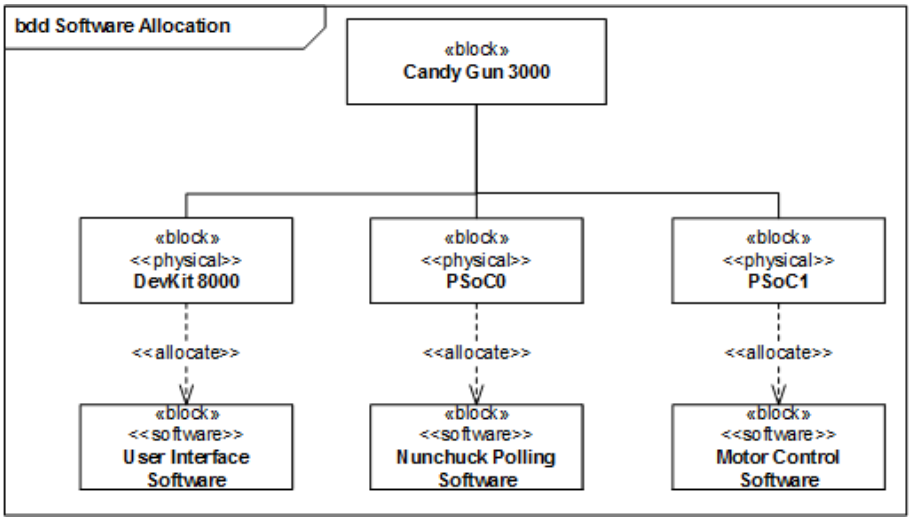


Figur 3: Systemets domænemodel

Her repræsenteres hardware som blokke forbundet med associeringer. Associeringerne viser grænsefladerne mellem de forbundne hardware komponenter (Enten *SPI* eller *I2C*), samt retningen af kommunikationen. Af modellen fremstår konceptuelle kommandoer for grænsefladerne, som beskriver deres nødvændige attributter.

10.2 Software Allokering

Domænemodellen i figur 4 præsenterer systemets hardwareblokke. På figur 4 ses et software allokeringsdiagram, som viser hvilke hardwareblokke der har softwaredele af systemet allokeret på sig.

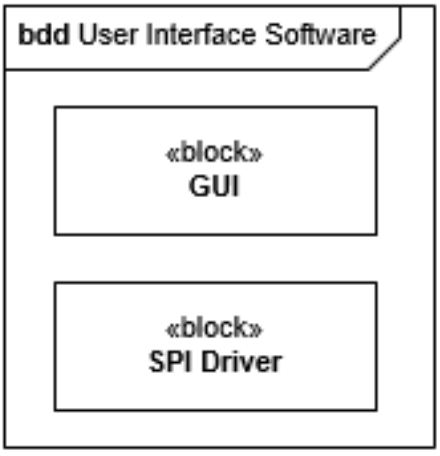


Figur 4: Systemets software allokeringer

Det kan her ses at systemet består af tre primære softwaredele: *User Interface Software*, *Nunchuck Polling Software*, *Motor Control Software*. Disse er fordelt over de tre viste CPU'er.

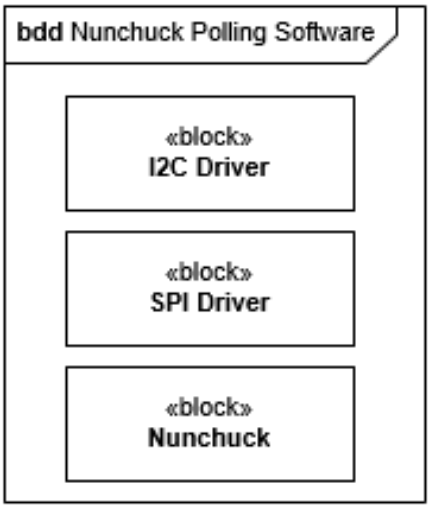
På figurerne: 5, 6, 7 ses et overblik over konceptuelle klasser der repræsenterer ansvarsområder de primære softwaredele får. For en beskrivelse af design og implementering af disse primære softwaredele henvises til **DESIGN OG IMPLEMENTERING #ref**.

På figur 5 ses det at *User Interface Software*, allokeret på DevKit 8000, har ansvar for brugergrænsefladen samt en SPI Driver til kommunikation med PSoC0.



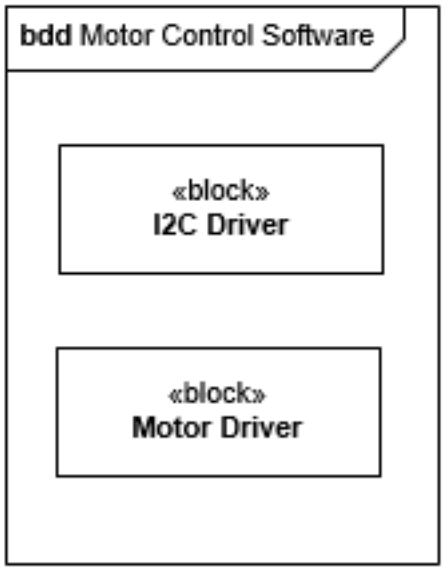
Figur 5: Systemets software allokeringer

På figur 6 ses det at *Nunchuck Polling Software*, allokeret på PSoC0, har ansvar for en I2C Driver, SPI Driver, samt en Nunchuck API. I2C Driveren skal bruges til kommunikation med den fysiske Nunchuck controller samt PSoC1. SPI Driveren skal bruges til kommunikation med DevKit 8000. Nunchuck API'en bruges for at tilgå data'en fra den fysiske Nunchuck controller.



Figur 6: Systemets software allokeringer

På figur 7 ses det at *Motor Control Software*, allokeret på PSoC1, har ansvar for en I2C Driver til kommunikation med PSoC0, samt en Motor Driver til motorstyring.

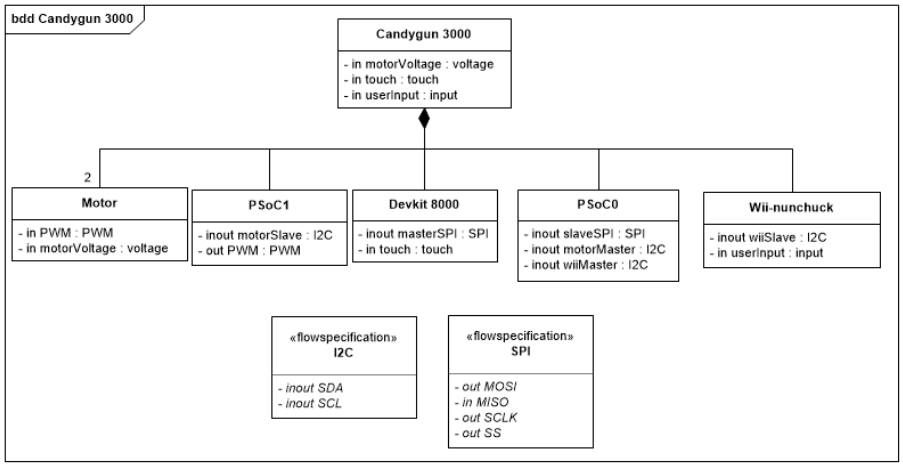


Figur 7: Systemets software allokeringer

10.3 Hardware

10.3.1 BDD

På figur 8 ses BDD'et for systemet.



Figur 8: BDD af systemets hardware

Her vises alle hardwareblokke fra domænemodellen (figur 3) med nødvændige indgange og udgange for de fysiske signaler. Yderligere ses det at flow specifikationer er defineret for de ikke-atomare forbindelser *I2C* samt *SPI*, da

disse er busser bestående af flere forbindelser. Der henvises til **IBD AFSNIT** for en detaljeret model af de fysiske forbindelser mellem hardwareblokkende.

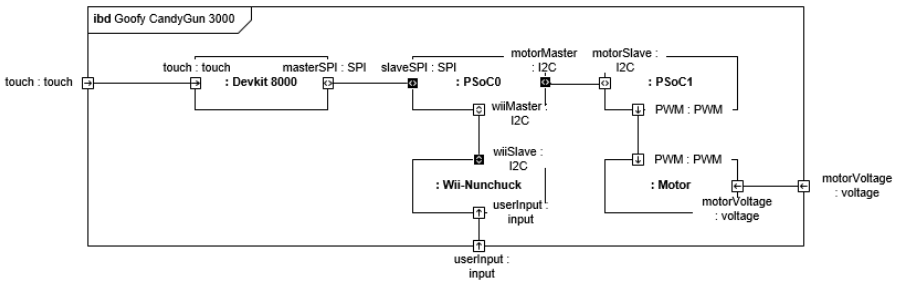
10.3.2 Blokbeskrivelse

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 kommunikatioon 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.

10.3.3 IBD

På figur 9 ses IBD'et for systemet.



Figur 9: IBD af systemets hardware

Her vises alle hardwareblokke med de fysiske forbindelser beskrevet i BDD'et (figur 8).

Det ses at systemet bliver påvirket af tre eksterne signaler: *touch*, *input*, samt *voltage*. *touch* er input fra brugeren når der interageres med brugergrænsefladen. *input* er brugerens interaktion med Wii-Nunchuk. *voltage* er forsyningsspænding til systemet.

10.3.4 Signalbeskrivelse

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 hvori 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 nunchuckinput 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% Spædingsniveau: 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 1: Tabel med signalbeskrivelse

10.4 Software

10.4.1 SPI Kommunikations Protokol

10.4.2 I2C Kommunikations Protokol

I afsnittet **IBD** ses det på IBD'et (figur 9) at tre hardwareblokke kommunikerer med hinanden via en I2C bus. Til denne I2C kommunikation er der defineret en protokol, som sætter en standard for hvordan modtaget data skal fortolkes. Denne protokol beskrives her.

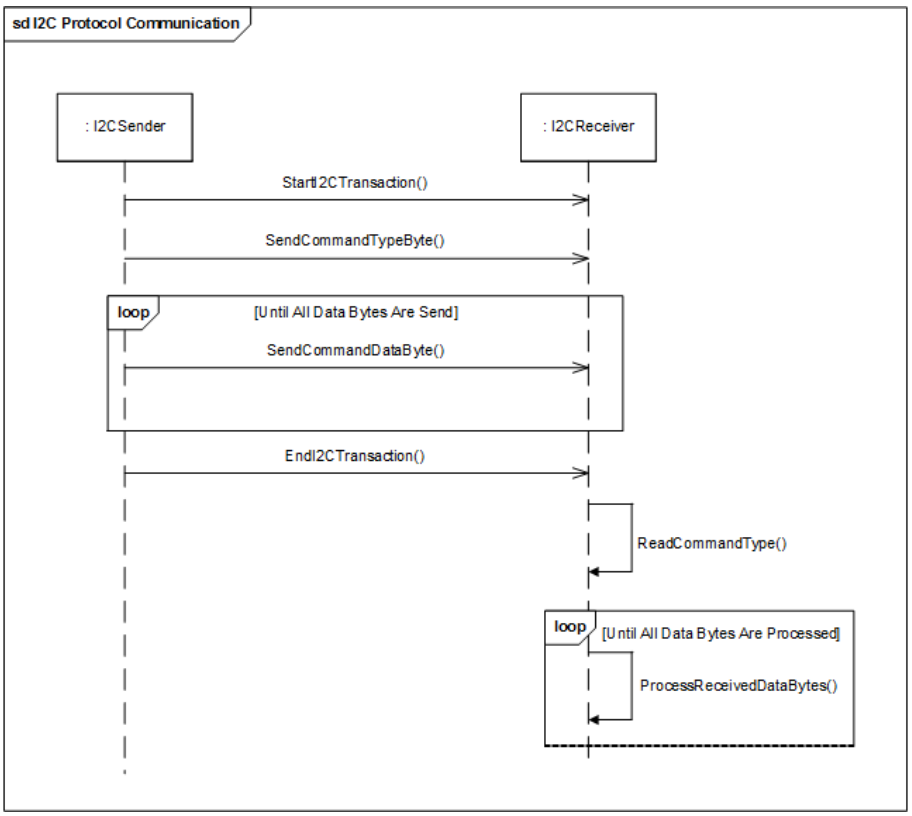
I2C gør brug af en indbygget protokol der anvender adressering af hardwareenheder for at identificere hvilken enhed der kommunikeres med. Derfor har hardwareblokkene som indgår i I2C kommunikation fået tildelt adresser. På tabel 2 ses adresserne tildelt systemets PSoCs.

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
Wii-Nunchuck	1	0	1	0	0	1	0	0/1

Tabel 2: Adresser der anvendes på I2C bussen

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 fortolkes. På figur 10 ses et sekvensdiagram der, med pseudo-kommandoer, demonstrerer forløbet mellem en I2C afsender og modtager ved brug af kommunikations protokollen.



Figur 10: Eksempel af I2C Protokol Forløb

På figur 10 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 3 ses de definerede kommandotyper og det tilsvarende antal af bytes der sendes ved dataveksling.

Kommandotype	Beskrivelse	Binær Værdi	Hex Værdi	Data Bytes
NunchchuckData	Indeholder aflæst 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 3: Kommandotyper der anvendes ved I2C kommunikation

Kolonnerne "Binær Værdi"og "Hex Værdi"i tabel 3 viser kommandotypens unikke tal-ID i både binær- og hexadecimalform. Denne værdi sendes som den første byte, for at identificere kommandotypen.

10.4.3 Specifikation og Analyse

11 Design og Implementering

11.1 Valg og Begrundelse

11.2 Hardware

11.2.1 Motorstyring

11.2.2 Affyringsmekanisme

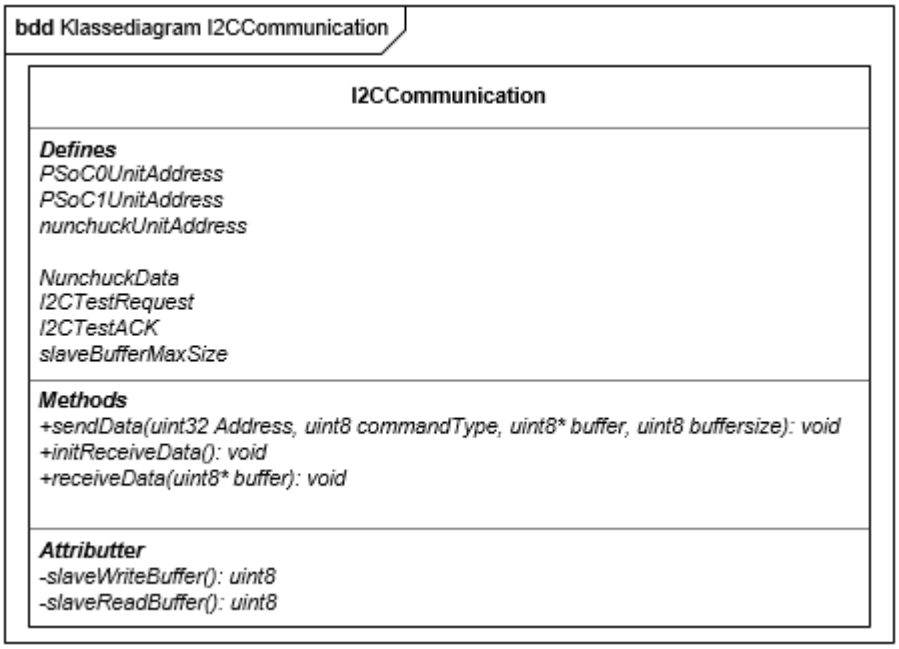
11.3 Software

11.3.1 I2C

I dette afsnit vil softwaren der omhandler I2C-kommunikation blive beskrevet. Dette inkluderer et klassediagram, samt en klassebeskrivelse.

Klassediagram

På figur 11 ses klassediagrammet for I2CCommunication.



Figur 11: Klassediagram for I2CCommunication klassen

Klassebeskrivelser

Som det ses på klassediagrammet figur 11 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.

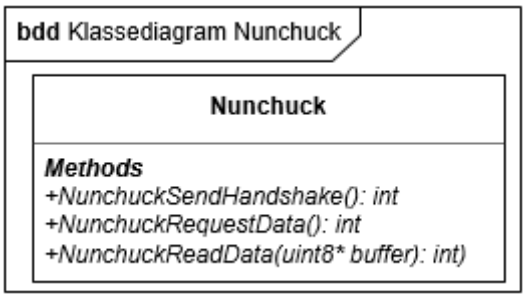
void receiveData(uint8* buffer)
Denne metode venter på at slaveRead bufferen er blevet fyldt. Når dette er sket, bliver slaveRead bufferen kopieret over i "buffer".

11.3.2 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 12 ses klassediagrammet for Nunchuck klassen.



Figur 12: Klassediagram for klassen Nunchuck

Klassebeskrivelser

Metoderne fra klassediagrammet figur 12 vil blive beskrevet i dette afsnit.
int NunchuckSendHandshake()
Denne metode sender et "handshake"(Se dokumentationen afsnit **INSERT AFSNIT 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.

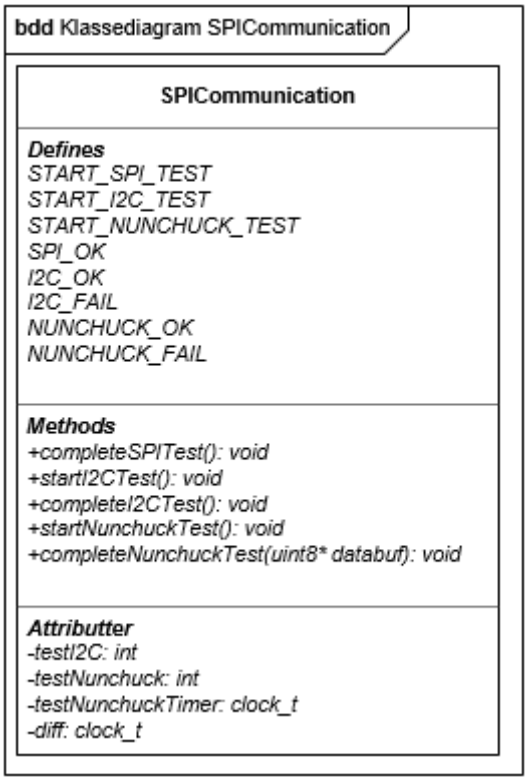
I klassediagrammet er der en sektion kaldet "Defines". Disse Defines bruges i implementeringen til forskellige formål. *PSoC0UnitAddress, PSoC1UnitAddres* og *nunchuckUnitAddress* bruges til at definere adresserne for I2C-nettets slaver. *NunchuckData, I2CTestRequest* og *I2CTestACK* er kommando typer 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 dokumentationen afsnit **INDSÆT REFERENCE TIL DOKUMENTATIONAFSNIT #ref**)

11.3.3 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 13 ses klassediagrammet over SPICommunication klassen.



Figur 13: 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.

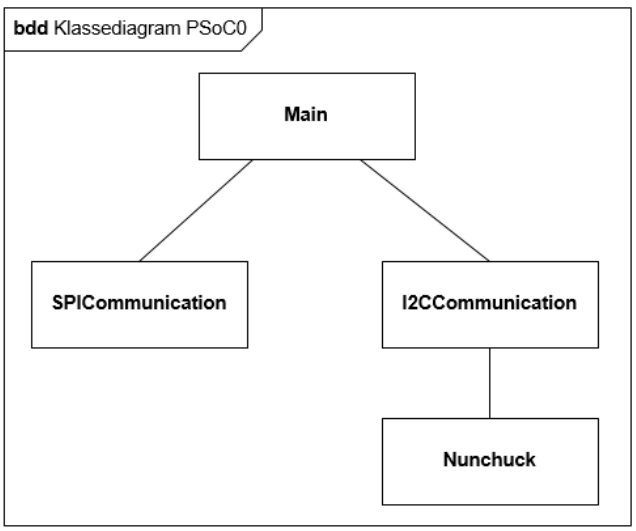
void completeI2CTest()
Denne metode gennemfører I2C-Testen. Dette gøres ved at der sendes en besked til alle enheder på I2C-nettet, og hvis der ikke registreres nogen fejl på denne besked, bliver *I2C_OK* gemt i PSoC'ens SPI-transfer buffer. Registreres der en fejl, bliver *I2C_FAIL* gemt i SPI-transfer 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.

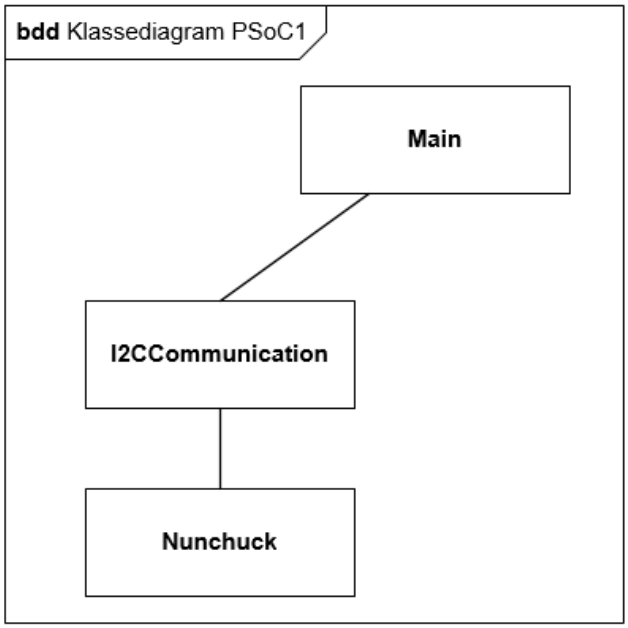
I klassediagrammet er der beskrevet en række "Defines". Disse defines bruges i klassen som unikke ID'er der indikerer om en test er gennemført OK eller om den er fejlet.

11.4 PSoC Software

PSoC0 og PSoC1 gør begge brug af den tidligere beskrevne software. De følgende klassediagrammer figur 14 og 15 giver et overblik over hvilke klasser der bliver gjort brug af på de forskellige PSoCs



Figur 14: Klassediagram oversigt for PSoC0



Figur 15: Klassediagram oversigt for PSoC1

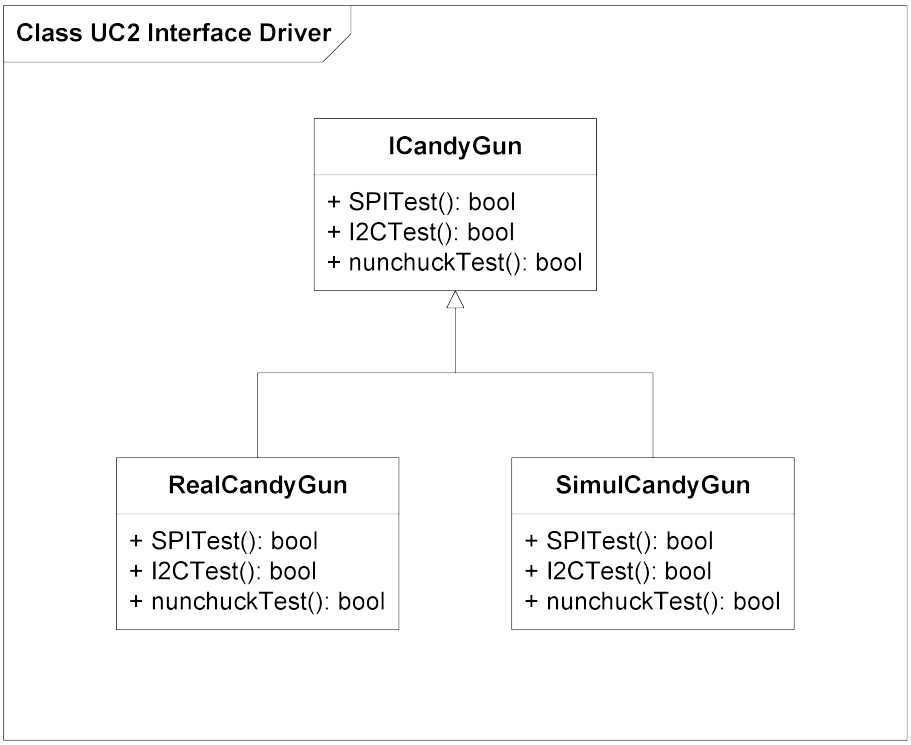
11.4.1 SPI - Devkit8000

Candydriveren sørger for SPI-kommunikationen fra Devkit8000 til PSoC0. Driveren er skrevet i c, hvilket er typisk for drivere til linuxplatforme.
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-relaterede 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.

11.4.2 Interface Driver

Interface driveren fungerer som bindeled mellem brugergrænsefladen og candydriveren på Devkit8000. Den indeholder tre funktioner. Funktionerne anvendes i use case 2 til at teste kommunikationsforbindelserne i resten af systemet. Interface driveren er designet og implementeret i C++ og gør brug af klasserelationen arv. Et klassediagram for interface driveren se på figur 16.



Figur 16: Interface driver for UC2

Basisklassen er ICandyGun. Det er en abstrakt klasse, da den udelukkende indeholder virtuelle metoder. Derudover er der to afledte klasser; SimulCandyGun og RealCandyGun. SimulCandyGun implementerer metoderne til at simulere respons fra Candydriveren. Dermed kan brugergrænsefladen testes uafhængigt af de resterende dele af systemet. Simuleringen er implementeret med *srand()*-funktionen fra *cstdlib*-biblioteket, som returnerer et tilfældigt tal, som her bliver mellem 0 og 1. I RealCandyGun-klassen er metoderne implementeret efter den reelle SPI-protokol og med de nødvendige funktioner til at skrive til et kernemodul. Fx open(), close(), read () og write(). Da interface driveren er implementeret med arv, skal der ikke foretages betydelige ændringer i brugergrænsefladen, når der skiftes mellem simuleringsklassen og den rigtige version. Dermed opnås lav kobling.
De tre funktioner som Interface driveren indeholder i forbindelse med use case 2 (test use casen) er: SPITest(), I2CTest(), NunchuckTest(). Hver af de tre funktioner anvendes til at starte en test af de forskellige kommunikationsforbindelser: SPI, I2C og brugerinputet fra nunchucken. Alle funktionerne returnerer en bool, som enten er true eller false, alt efter om testen var succesful eller ej. Når der skal startes en test, åbner den pågældende funktion filen *dev/candygun* og skriver SPI-kommandoen for *start test* til filen. Derefter venter funktionen ét sekund og læser så svaret fra filen. Da der i nunchucktesten ventes på et brugerinput, og brugeren skal have lidt tid til at trykke på nunchuck-knappen, er der oprettet en while-løkke, som tjekker flere gange om testen returnerer true. Hvis testen ikke returnerer true ved første check, venter funktionen atter et sekund og tjekker igen. Det gør den op til 15 gange og melder derefter om fejl, hvis ikke den returnerer true inden.
Brugergrænsefladen anvender interface driveren ved at inkludere headerfilerne og oprette en ICandyGun pointer, der peger på en instans af én af de to afledte klasser. Ved at pakke kommunikationen til kernemodulet for candydriveren væk i funktioner kan brugergrænsefladen anvende funktionerne uden at kende til SPI-protokollen. Det sikrer igen lav kobling, og i tilfælde hvor det kunne ønskes, at SPI-kommunikationen kan erstattes af en anden kommunikationsform, kan det gøres uden, at der skal foretages ændringer i brugergrænsefladen.

11.4.3 Brugergrænseflade

Brugergrænsefladen for UC2 er en simpel test-konsol. Den består af 3 knapper og et tekstvindue. Brugergrænsefladen interfacer med SPI-protokollen, gennem vores interface driver. Den første knap, Start test, initierer UC2. Efterhånden som testen løbes igennem kaldes test funktionerne, og ved hjælp af if-conditions, bliver der tjekket på retur-værdierne fra interface-funktionerne. Hvis retur-værdien er true, skrives der en --test successful"besked i tekstvinduet. og widgeten kører videre. Hvis retur-værdien er false, skrives der en --test unsuccessful"i tekstvinduet og widgeten returnerer til idle tilstand. Når alle test er successful, skrives "System test successful, system is ready for use"til tekstvinduet, og widgeten returnerer til idle tilstand. Den anden knap, Clear, clearer tekstvinduet til blank tilstand. Den tredje knap, Exit, lukker widgeten.

12 Udviklingsværktøjer

12.1 PSoC

12.2 DevKit 8000

13 Resultater og Diskussion

13.1 Perspektivering

13.2 Perspektivering til semesterets kurser

13.3 Ingeniørfaglige Styrker og Svagheder

14 Fremtidigt Arbejde

15 Fejl og Mangler

16 Referencer

Litteratur