— El Diablo Studios —

師傅

Windows Embedded 1 Projekt

Martin Aksel Jensen, Kim Munk Petersen, Tor Ranfelt

04-06-2009

Dedikering

Til Angus Macgyver du viste os, at lige meget hvor svær situationen ser ud, så findes der altid en løsning.

Indholdsfortegnelse

Projektbeskrivelse

Til vores projekt vil vi gerne lave et lille spil, nærmere et kamp-spil hvor man skal kunne være en eller flere spillere, der skal kunne kæmpe mod hinanden. Spillet skal være animeret og man skal kunne bruge Tahoe2 knapper, såvel som Wii Classic Controller til styringen. Vi har flere ønsker for spillet, blandt andet multiplayer på samme board, såvel som trådløs med Zigbee moduler, lydeffekter og evt. også musik.

Hardware mål

Vores første hardware mål er rimeligt beskedne. Vi bør som minimum have drivers til Accelerometer, Touch-controller og Wii Classic Controllers. For to spillere på samme enhed, skal Wii controllerne virke på samme board.

Næste mål er at få ZigBee modulet til at kommunikere med et andet modul tilsluttet endnu et Tahoe2 board, således vi kan få lavet noget trådløs kommunikation mellem to enheder.

Sidste mål og måske (antager vi) det sværeste mål, er at få tilsluttet og benytte en MP3-decoder således vi kan få lydeffekter og måske baggrundsmusik.

Software mål

Første mål er at få lavet en lille grafikmotor til understøttelse af spillet. Denne skal håndtere både animationer, tekst, billeder mm. Der skal også laves driver-klasser til at kommunikere med diverse hardware komponenter samt en input abstraktion som kan benyttes af spillet til aflæsningen og callback for input fra enhederne tilsluttet.

Næste mål er Zigbee driver og en overlæggende kommunikations protokol der sikre at to spil instanser på hver deres board kan arbejder sammen i overensstemmelse med spillets logik. Akkurat som hvis et multiplayer spil foregik på et Tahoe2 board lokalt.

Sidste mål er MP3 afspilleren, der kan sende data til afspilning så spillet kan få lagt lydeffekter over sig. Igen her er der brug for et overlæggende niveau for at abstrahere fra MP3 dekoderens logik, og give en programmerings brugerflade der er mere passende i et egentlig programs kontekst (for os værende et spil).

Disse tre ovenstående mål er selvfølgelig kun fundamentet som selve spillet skal bygge oven på. Det øjeblik vores første mål er opfyldt med en fungerende grafikmotor kan vi faktisk gå i gang med udviklingen af selve spillet. Derfor er det vigtigt at værktøjerne vi udvikler udmunder i nogle simple API'er med det for øje.

Sifu (師傅)

Sifu som vores kampspil er døbt, skal være en kampstil lignende Streetfighter eller Tekken i sin aller simpleste form med kun få muligheder for at slag, spark og hop. Her nedenunder vil vi forklare om de elementer som indgik i at lave dette spil.

Arkitektur

Elementerne i arkitekturen er beskrevet i de efterfølgende afsnit, for opbygning og brug af elementerne som indgår i spillet.

Sprite Engine

Der er lavet en rendering engine (motor) til formålet at designe et spil til .NET microframeworket. Der findes i øjeblikket ingen andre projekter til at kunne lave spil med, som f.eks. XNA til det fulde .NET framework og PC. Denne motor er et meget simpelt threaded framework, der tegner sin spritessamling på et kanvas der efter hver rutine sender sit indhold til skærmen.

Motoren kan fodres med sprites som den ufortrødent renderer indtil de bliver fjernet igen. Funktionalitet der er nødvendig for at gøre dette muligt, såsom *suspend* og *resume* der gør det muligt at sikkert ændre på hvilke sprites der skal renderes ved hvert gennemløb imens motoren er suspenderet. Samtidig er det muligt at skrue ned for antallet af renderinger i sekundet og derved frigive mere processortid til resten af systemet. Dette gøres vha. *RenderWait,* der afgør tidsintervallet mellem hver rendering. Motoren er som sagt trådet, og det er den for at sikre at skærmen kan opdateres sideløbende med bruger-input og/eller programlogik der har ændret på det visuelle outputs forudsætninger, hvilket selvfølgelig igen giver sig i udtryk i et egentlig output.

For at gøre denne sprite håndtering mere systematisk og threadsafe har vi udvidet med et begreb vi kalder ”scener”. En scene er som navnet antyder en scene i spillet, værende det hovedmenuen eller kamparenaen der er spillet omdrejningspunkt. Det vigtige er at der kun kan være en aktiv scene af gangen. Dette håndteres af Game-klassen der arbejder med selve motoren, og som bl.a. tager sig af at tilsidesætte motoren, smide sprites ud fra den gamle scene, sætte gang i load scenen (load baren), dispose den gamle scene, give den nye scene besked på at tilføje sine sprites til motoren og bagefter lade motoren genoptage sin opgave. Med andre ord alt hvad der måtte være nødvendigt for at supporte scener.

Sprites

Den abstrakte sprite klasse, fra hvilken alle egentlige sprite klasser nedstammer, indeholder de basale ting vores motor har behov for når den skal positionere og rendere de sprites den bliver fodret. Derudover indeholder den ting som mange sprite klasser vil have gavn af (f.eks. Width og Height). Altså hvor en generalisering er passende.

Der er en række metoder som nedarvende sprites skal overskrive (og som de skal kalde tilbage til) der alle tager sig af aspekter som f.eks. rendering og disposing. Det er i selve renderingen at en sprite tegner sig selv på det kanvas der er blivet givet som argument. En sprite kan uden videre tegne som det passer den på kanvassen, men det er naturligvis meningen den skal arbejde med den givne position værende spritens øverste venstre punkt. Der er ikke noget i vejen for at en sprite kan være mere

kompleks end som så, hvorfor vi har en vifte fra noget så basalt som TextSprites, SolidColorSprite og BitmapSprite til noget så eksotisk som HealthBarSprite, StringMenuSprite, ProgressSpriteog KeyboardSprite.

Det er vigtigt at være opmærksom på det at sprites i vores motor ikke er givet hver deres afgrænsede område, men altså blot et udgangspunkt (dets position), hvorfra det i reglen kan gøre hvad det vil. Derfor er den rækkefølge som sprites bliver tilføjet motoren relevant, da det afgøre den rækkefølge de vil blive renderet i. Konsekvensen værende at den sidst tilføgne sprite tegner oven på de andre hvis flere prøver at arbejde med en given pixel.

De færreste sprites er statiske i deres indhold. Hermed ment at de ikke bare gives en position og tegner sig selv når de får besked på det. ProgressSprite, HealthBarSprig og KeyboardSprite er eksempler på mere komplekse sprites der er dynamiske i deres indhold. Det er klart at hver enkelt dynamisk sprite har sine egne megen individuelle egenskaber der kan modificeres, og derfor er det op til den enkelte sprite selv at tilbyde den nødvendige funktionalitet. Et eksempel er KeyboardSpriten hvor den aktuelt valgte knap kan ændres med metoderne moveLeft(), moveDown(), moveUp() og moveRight(), og hvad der står skrevet i keyboardets tekstfelt kan ændres af et applySelected() kald der sætter i værk hvad end den valgte knap har af semantisk betydning for spriten.

Vores motor og sprites har en smule redundant kode, såsom rendersquare og IsDirty, der er et levn fra vores oprindelige forestilling om at motoren udelukkende skulle beskæftige sig med at opdatere de dele af skærmen der var berørt af ændringer. Herved kunne der undgås spildt regnekraft på et i forvejen forholdsvis begrænset embedded system. Idéen var at kun hvis en sprite var i det givne rendersquare og var "forurenet" (altså dets forudsætninger havde ændret sig) skulle den gentegne sig selv. Den givne square ville selvfølgelig være blevet ryddet før spritesene var gået i gang med deres rendering. Det viste sig desværre at være meget ineffektivt, og i praksis var den brutale løsning hvor alle sprites gentegnes efter hver endt rutine i den grafiske motor den bedste. På papiret var efter behov løsningen altså bedre, men algoritmens konstanter var så meget værre at den teoretiske forbedring kun ville have gjort sig gældende med utrolig mange sprites. En mængde der med garanti ville have bragt vores embedded system i knæ lige meget hvor kreative vi havde været.

Scener

Sceners rolle er allerede blevet omtalt med hvordan de er den måde hvorpå vi strukturerer præsentationen af programmet (som et vindue i et traditionelt vindue program). En scene består af flere sprites som ved loading bliver tilføjet til motorens aktive sprites. Derudover bindes der under konstruktionen metoder til input events. Disse bliver fjernet igen når scenen bliver disposed. Det er vigtigt at være opmærksom på dette for at undgå at to scener begge reagerer på det samme input, hvis det er meningen kun én scene fortsat skal lytte på netop det input.

Som det altid er tilfældet med .NET Frameworks, sikrer Micro Frameworket, at user input event lyttere (som *OnKeyPress()*) bliver aktiveret sekventielt, men derudover bliver lytterne der venter det event at scenen er afsluttet sat i kø i vores UI workerthread. På den måde sikrer man dataens integritet for det tilfælde at en speciel slags scene måske vil erklære sig for færdig uden for en af de lyttere den selv har på egentlige user input events, og dermed måske arbejde asynkront (ikke noget der faktisk forekommer

i vores spil), og man undgår at RaiseSceneEnded ødelægger user input lytterens arbejdsflow ved aktiveringen af lytterne der venter på scenens afslutning.

Partielle scener

En partiel scene er en scene abstraktion ovenpå klassen Scene der muliggør dialogbokse. Vi tillader kun at en scene har én partiel scene af gangen (ikke at andet ville have været utænkeligt i vores framework, men vi fik bare aldrig brug for en sådan facilitet og valget simplificerede implementeringen).

Når en scene skal vise eller holde op med at vise en partiel scene, anvendes henholdsvis metoderne Show og Hide. Ved Show loades den partielle scene, men i stedet for at lægge dets sprites ind i den bagvedliggende scene bruger den bare load til at beregne sine sprites. Dvs. konstruere og konfigurere dem i forlængelse af hvordan den selv er blevet konfigureret. Bagefter henter den nu bagvedliggende scene de beregnede sprites fra partiel scenen og tilføjer dem til motorens. Omvendt når en partiel scene fjernes sker det ved at en scenes Hide metode fjerner den partiel scenes sprites fra motoren.

Ved konstruktionen af en partiel scene bliver der ligesom med en almindelig scene tilføjet de nødvendige lytter af bruger input. Derfor sker der det at en almindelige scene har mulighed for at fjerne sine input lytter når der ved kald af show lægges en partiel scene ovenpå, og omvendt tilføje dem igen når der kaldes Hide. På den måde undgås det at undgå den bagvedliggende scene uønsket reagere på bruger input der alene er tiltænkt den partielle scene.

Threads

Der bliver brugt mange tråde i vores system og hvad der er vigtigt når vi bruger så mange tråde er at man altid husker og sætte tråden til at sove en hvis ventetid og eller i en løkke huske at kalder Sleep(0) når vi ved der gerne lige må komme andre til. I .NET microframework er det lavet sådan at normalt får en tråde 25ms at arbejde i, før den næste tråd får et tidsrum at arbejde i. Når man også bruger mange tråde som vi gør i vores spil og kommer til at arbejde på højtryk, så kommer der meget sjældent et pusterum til at backend systemerne for den virtuelle maskine får tid tildelt, at kunne arbejde i. Dette har betydet at vi har været nød til helt specifikt kræve at garbage-collectoren, som er implementeret ved en simpel mark-and-sweep strategi, får lov at køre og samtidig at vi venter på at finalizeren-tråden færdiggøre alle objekter der markeret til fjernelse i hukommelsen.

Oprydning

Oprydning (eller disposing) er meget centralt vores spil, da vi ikke i dette aspekt af programmet kan forlade os på garbage-collectoren hvis det skal gå hurtigt. Udover konsekvenserne for performance er det simpelthen en nødvendighed kvag den måde vi har scener til at lytte efter brugerinput. Som sagt skal en scene ved dets konstruktion binde sig fast til diverse input kilder den måtte have behov for, men disse skal fjernes igen. Umiddelbart virker det oplagt at fjerne dem når scenen selv afgør den er færdig (RaiseSceneEnded()), men når en scene er færdig kan det eksempelvis meget vel være en scene der er en partial til en anden scene, hvoraf den sidste har brug for at hente information fra den ny-afsluttede partiel scene (såsom hvilken controller der blev valgt). Dette skal selvfølgelig gøres før den bliver disposed, hvorfor disposing er noget der både er nød til at ske eksplicit samt skal afgøres udenfor scenen selv.

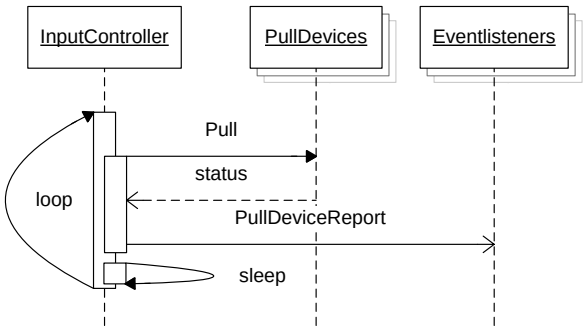
Timer

Fordi vi ville lave et spil, bliver vi ret afhængige af timers og for at få den bedste kontrol, lavede vi vores egen, sådan at vi kunne få den logik præcis som vi ønskede. Det kommer heller ikke uden sin pris, hvilket viser sig ved at hver timer-instans har en tråd hver og betyder at kontekst-switching når logikken på de forskellige tråde skal køres kan ende med at blive dyr i den sidste ende. Men også at vores timers måske ville bliver lidt mere upålidelige end den mulige implementation .NET Micro Frameworket benytter. Men vi har endnu ikke under udvikling og live tests mødt reelle problemer med vores egen implementerede timer.

Timerens loop der kører er meget simpelt, det kører men en variabel er *true*, hvilket gør vi kan lade tråde pænt terminere når en timer disposes eller ryddes op. Inde i det loop kaldes *WaitOne* på et AutoResetEvent, hvilket sætter den givne tråd til at vente, ligesom ved et Thread.Sleep, på at et event sker, eller at der sker et timeout. At denne ventemetode har to udfald, ekstern påvirkning eller timeout gør dette perfekt til vores timer, hvor vi ønsker at implementere et *reset*. For at kunne genstarte og starte ventetiden med det samme, i stedet for at vente på timer-trådens tilbagevenden, kan vi når eksternt påvirke det *AutoResetEvent* og på baggrund af dets udfald vælge at stoppe løkken.

InputController

Hele hjertet for at alt input sker igennem InputControlleren, hvor man kan tilslutte sig events og lytte på event på blandt andet knapperne på Tahoe kortet, TouchScreen og pull-devices. Wii enhedsdrivers er implementeret som at værende *IPullDevice*, som InputController både stiller mulighed for at lave forespørgelser på specifikke drivers og derefter tilføje dem til en observationsliste. Den har også en løkke som med et givent interval foretager et *Pull* på hver enhed, i observationslisten og hvis der er sket ændringer, vil InputControlleren meddele ændringer ved hjælp af *PullDeviceReport* begivenheden.



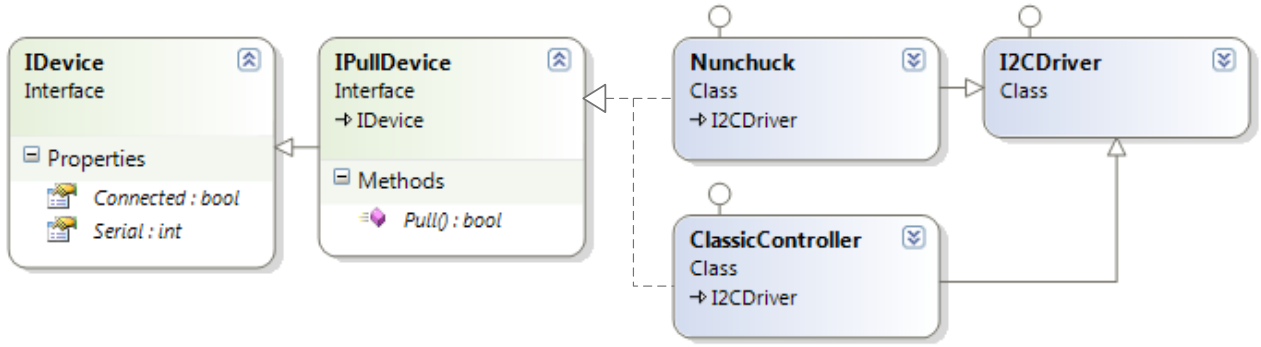
Under instantieringen af InputControlleren opretter den en masse driverinstanser som stiller flere inputenheder til rådighed. Blandt andet oprettes der et Accellerometer pulldevice, der oprettes med et normalt *I2CDevice.Configuration objekt,* og derudover oprettes der to *ChangableI2CConfiguration* instanser som er initialiseret med hver sin kanalværdi. Herefter oprettes der en Nunchuck og en ClassicController driver for hver buskonfiguration og hvis enhedens *Connected* egenskab returnere falsk, så udlades denne driver fra driverlisten. På den måde er der understøttelse får flere kombinationer af wii udstyr på de forskellige I^2C kanaler, så man fx kan have to samtidige ClassicControllers tilsluttet eller blandet ved en ClassicController og en Nunchuck.

Hardware Drivers

Projektet inkluderede brug af flere forskellige enheder, blandt andet Wii controllers og ZigBee enheder til netværkskommunikation. Derfor skulle der også udvikles drivere til disse enheder for at have support for brug af disse enheder. Rent faktisk har vi lavet drivere til stort set alle Tahoe'ens funktioner, fx touch screen, accelerometer, temperaturmåler og SD-kort.

Arkitektur

De eksterne enheder som vi skulle skrive drivers til, var passende alle input enheder. Enheder hvor man kunne læse en status for den specifikke enhed og det er alle enheder som man skal tilgå og trække dataene ud, de sender ikke individuelt data til enheden. På baggrund af dette lavede vi et interface kaldet IPullDevice, som implementerer nogle nødvendige funktioner og egenskaber for at kunne lave en abstraktionsmodel over dette. Som det kan ses i billedet nedenunder er interfacet en specialisering af IDevice som har et serielnummer til unik identificering i systemet og en *Connected* egenskab som angiver om en enhed er tilsluttet eller ej. Hvad IPullDevice bringer til bordet er en *Pull* metode som returnere en boolean værdi som indikere om der er sket en forskel i status siden sidste dataudtrækning.



På billedet kan man se at blandt andet vores ClassicController driver implementere *IPullDevice* da Wii Controllers fungerer på den måde at man henter enhedens statusrapport over I²C, så man kan hente og behandle rapporterne i det tempo man ønsker.

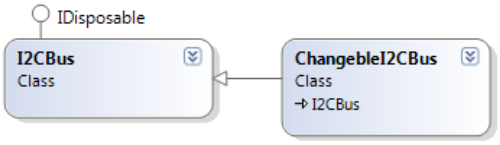
For at kunne benytte flere I2C enheder på en gang, har vi været nød til at lave en abstraktion over en I2C enhed, *I2CBus*, da IO-pindene der bruges til kommunikation bliver reserveret. Dette er derfor vi har lavet en abstraktion, som benyttes af alle vores device-drivers og er lavet thread-safe da denne klasse kan blive brugt af flere samtidige tråde.

Drivere vi har udviklet til brug som *Pull Devices* er, accellerometer, temperaturføler, touchscreen som enten direkte bliver brugt i spiller eller som man kan teste under *technical* menupunktet..

I²C Mux

I vores projekt har vi valgt at vi skulle kunne benytte mere end et Nintendo Wii udstyr, problemer opstår dog da Wii udvidelsesudstyr til Wiimoten alle har samme I²C-adresse. Kommunikation mellem udvidelsesudstyr og Wiimoten er tilsluttet via en I²C-bus. Det giver så et problem da for at kunne snakke med mere udvidelsesudstyr af gangen er vi nød til at få bussen delt op i flere, som vi via software kan skifte imellem. Til dette valgte vi at bruge en to-kanals multiplexer fra NXP, kaldet *PCA9542A* som

gentager data ud på dens forskellige busser alt efter hvilken kanal chippen er sat til at benytte på et givent tidspunkt.

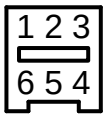


I2CBus klassen, som i samme stil med FusionWare's bibliotek, som kan findes på codeplex, laver en abstraktion som kan bruges til kommunikation med I²C enheder fra flere klasser på en threadsafe måde. Yderligere har vi så lavet en specialisering af denne klasse, *ChangableI2CBus*. Denne specialisering indeholder speciel logik til håndtering af endnu end klasse specialisering af *I2CDevice.Configuration,* kaldet *ChangeableI2CConfiguration*. Klassen indeholder information om, hvilken mux kanal denne konfiguration vil benytte, og når man eksekvere transaktioner på en *ChangeableI2CBus* vil den genkende konfigurationsklassen og sørge for at skifte MUX I²C kanal til at værende den tilsvarende beskrevet af konfigurationen. Dette giver os nu mulighed for at snakke med flere enheder med samme I²C adresse, men siddende på forskelle MUX kanaler.

Nintendo Wii Udstyr

Vi har implementeret mulighed for kommunikation til Nintendo Wii enheder, nærmere Nunchuck og Classic enhederne. Wii tilbehørs enheder tilsluttet wiimoten via et eget defineret stik, men kommunikationen foregår ved I2C kommunikationsprotokollen. Fra de informationer vi har, kunne opstøve, fra internettet ligner det at alle enheder har samme I2C adresse, 0x52, hvilket også giver mening i forhold til det reelt kun er muligt at tilslutte et enkelt stykke eksternt udstyr til en Nintendo Wiimote.

Nunchucken er en pind med monteret xy-joystick, to knapper samt et accelerometer. Classic controller er et gamepad som har 15 knapper, blandt andet krydsnavigationsknap, A, B, X, Y, to xy-joystick og to tryksensitive skulderknapper.



Figur 1 - Wii Stik

```
1 – SDA
3 – +3V3
4 – SCL
6 - Ground
```

Udvidelses-controllers til Nintendo Wiimoten tilsluttes dens I²C bus som de fleste DIY grupper tilslutter med en bus hastighed på 100KHz, det forlyder dog at hvis man tilslutter enhederne 5 volt i stedet for 3.3 kan man kommunikere med enheden med en bushastighed på 400KHz. På internettet anbefaler grupperne dog 3.3 volt og 100KHz bushastighed.

Nunchuck Controller

Kommunikationen mellem host og Nunchuck er endnu ikke fuldt forstået, men man har fundet ud af mere en rigeligt for at få et brugbart resultat i vores tilfælde. På nettet har personer fundet frem til at få læst status for knapper, joystick og accelerometer og ikke nok med det, så man også få læst nunchuckens on-board kalibreringsdata.

For at læse hvad der står i, hvad andre formoder er et, enhedsidentifikations register skal man først sende enheden en enkelt byte, 0xFA, og herefter læse 6 bytes sendt fra enheden. Enhedsidentifikationsnummeret for Nunchuck er

```
00: 00 00 A4 20 00 00
```

Før man kan læse nunchuckens status skal nunchucken være initialiseret først og det gøres ved at skrive to bytes til enheden, 0x40 0x00. Herefter kan man begynde at trække status pakker ud af enheden, ved først at sende enheden en byte, 0x00 og efterfølgende læse 6 bytes sendt fra enheden. Indholdet af disse seks bytes er beskrevet i følgende tabel:

		Bit							
		7	6	5	4	3	2	1	0
Byte	0	SX<7:0>							
	1	SY<7:0>							
	2	AX<9:2>							
	3	AY<9:2>							
	4	AZ<9:2>							
	5	AZ<1:0>		AY<1:0>		AX<1:0>		BZ	BC

Tabel 1 - Nunchuck Datapakke

Kalibreringsdata

Kalibrerings data kan hentes ved at læse 16 bytes fra 0x20 eller 0x30 registret. Kalibreringsdatapakken indeholder information om maksimum, minimum og center værdi for joysticket, Accelertions-værdier for x-, y- og z-aksen på 0G og 1G af kendt data. Et eksempel på data kan ses efterfølgende.

```
00: 29 7F 79 34 AD B2 AF 16
08: E3 20 84 DE 25 7B 23 78
```

For at trække denne information fra enheden skal man skrive en byte 0x20 eller 0x30 og derefter læse 16 bytes som indeholder kalibreringsinformationen.

Implementering

Nunchuck enhedsdriveren er implementeret som et Pull-device og er implementeret lignende denne pseudokode

```
Nunchuck Device

OnCreate:
```

```
  Check device-id
  if (correct device-id)
    Calibrate device
    Initialize device

OnPull:

  Send 0x00
  Receive 6 bytes
  Decode data
```

Classic Controller

Oversigt over status datapakken

		Bit							
		7	6	5	4	3	2	1	0
Byte	0	RX<4:3>		LX<5:0>					
	1	RX<2:1>		LY<5:0>					
	2	RX<0>	LT<4:3>		RY<4:0>				
	3	LT<2:0>			RT<4:0>				
	4	BDR	BDD	BLT	B-	BH	B+	BRT	1
	5	BZL	BB	BY	BA	BX	BZR	BDL	BDU

Tabel 2 - Classic Datapakke

Kalibreringsdata

Kalibreringsdata for Classic controller indeholder maksimum, minimum og center for de to joystick der er monteret, samt nulpunktet for venstre og højre skulderknapper. Joystick informationerne er lagt ud ligesom med Nunchucken, hvor første byte er maksimum, anden byte er minimum og tredje byte er centerpunktet. Hvert joystick har seks bytes data som er lagt i forlængelse af hinanden, startende fra og med første byte. Disse data skal justeres før de direkte kan benyttes og man skal derfor dividere dataene med 4 for venstre joystick da den består af en 6-bits opløsning og data for det højre joystick og skulderknapper skal divideres med 8 da de har en 5-bits opløsning. De sidste to bytes af det 16 bytes i kalibreringspakken er endnu ukendte.

```
Kalibreringspakke

00: E4, 1C, 7E, EB, 1E, 7E, E1, 1F
08: 89, E5, 15, 81, 13, 16, 87, DC

Afbalanceret
01: 39, 07, 1F, 3A, 07, 1F, 1C, 03
08: 11, 1C, 02, 10, 02, 02, 87, DC
```

I eksemplet ovenover kan vi her at LX løber fra 7 til 57 og med center position ved 31 og RY løber fra 7 til 58 og med center position ved 31. De højre joystick kan aflæses på samme måde og nulpunktet for skulderknapperne hver kan man se er 2.

Implementering

Classic enhedsdriveren er implementeret som et Pull-device og er implementeret lignende denne pseudokode

```
Classic Device

OnCreate:

  Check device-id
  if (correct device-id)
    Calibrate device
    Initialize device

OnPull:

  Send 0x00
  Receive 6 bytes
  Decode data
```

Zigbee Trådløs Kommunikation

Som også beskrevet i vores projektbeskrivelse ønsker vi mulighed for at kunne spille multiplayer over netværk og det valgte vi at bruge et XBee modul fra *Digi*, som baserer sig på ZigBee™ specifikationen.

Hardware

XBee modulet blev valgt da de har en god rækkevidde (ved indendørs brug op til 40 meter), lavt strømforbrug og en tilstrækkelig netværksoverførselshastighed på 250 kilobit i sekundet. Meget passende kommunikeres der med XBee modulerne over seriel-kommunikation og TahoeII brættet har en stiktilslutning til modulet direkte på sig. Dette gjorde det nemt at skrive noget software som kan kommunikere med modulet, sende og modtage data. Kommunikationshastigheden virkede også til at være tilstrækkelig for vores tilfælde hvor der kun skal sendes forholdsvis små datapakker ofte.

XBee har to forskellige firmware til modulerne, der giver dem forskellige fordele alt efter hvilken software de er flashet med. Den simple firmware, *Transparent Operation*, har fordelene ved at have et simpelt interface og gør en masse ting automatisk for en. Der er dog ulemper ved dens operationsmåde, der er blandt andet mindre egenkontrol og mere modtager-information. Operationsmåden vi valgte at benytte var *API Operation* som tilbyder andre fordele end *transparent operation* men der er også flere opgaver der bliver pålagt udvikleren. Blandt andet skal man selv udvikle understøttelse for tilmeldelse af netværk og tilmeldelser af enheder, hvor den anden måde håndterer dette for en. Til gengæld får man nu mere kontrol og information, blandt andet får man adressen på afsender af datapakker som et eksempel. Dette er noget vi får brug for, da man i et netværk gerne skulle kunne have flere sessioner af spil kørende samtidig i et netværk og derfor er nød til at kende afsender.

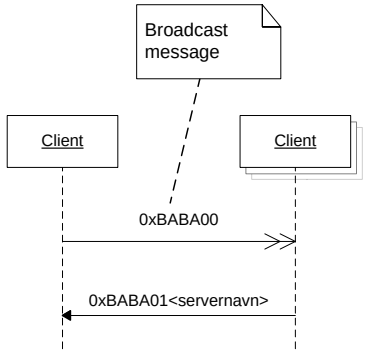
Software

Til brug i vores software, lavede vi vores egen mini arkitektur som bygger over .Net *SerialPort* klassen, som giver mulighed for at arbejde med objekter frem for rå byte-arrays fra data kommunikationen. Så for at lave dette har vi en eventlistener der lytter på *DataReceived* begivenheder og som samler ZigBee

transmissionspakker ind. Når en hel transmissionspakke er indsamlet gives dataene videre til en object-factory som dekoder dataene og omdanner pakkerne til objekter, som kan arbejdes med i på et højere niveau i koden. Som den er lige nu kan den dekode AT kommandoerne VR, HR og ND, samt *ZigBeeReceivePackerA00* og *ZigBeeTransmitStatus*. Til afsendelse af data er der lavet en metode, hvis parameter tager imod instanser af typen *XBeeCommand* som omdanner denne instans data til et byte-array der efterfølgende sendes ud på serielporten til XBee hardwaren til behandling.

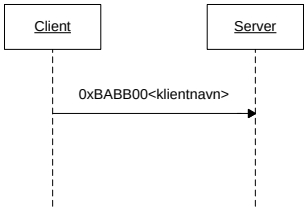
Multiplayer spil

Vi lavede vores egen lille kommunikationsprotokol til multiplayer, da det blandt andet skulle være muligt at have flere spil kørende på samme netværk. Derfor skulle der indbygges en mulighed for at spørge på netværket efter tilgængelige spil.



Som det måske bedst beskrives på tegningen, sender klienten en datapakke, ”BABA00”, ud på hele netværket som alle vil modtage. Herefter svarer servere som står klar til at acceptere klienter, med en datapakke der starter med ”BABA01” og efterfølgende sit servernavn i ASCII. Det er for at hjælpe spilleren til at identificere de tilgængelige netværksspil og vælge den rigtige modstander.

Network game-handshake udføres for endeligt at oprette en netværksspils session mellem to enheder og dette gøres ved først at sende en forbindelsesforespørgsel ved ”BABB00” efterfulgt at klientnavnet i ASCII kodning.



Efter dette er gjort antages det at en spilsession nu er oprettet og spillet laver sine controllers til sine spillere så der kan ske bevægelse på skærmen pr. input. Dette er ikke den bedste løsning idet der helt klart mangler noget tilbagemelding fra serveren, i stil med en TCP-handshake. Men pga. den begrænsede tid til rådighed var det et af punkterne hvor vi valgte at få lavet noget som ville virke, men havde sine begrænsninger og mangler.

Performance

Ved almindelige spil hvor vi benytter TahoeII hardwareknapperne virker multiplayer rigtig fint. Der sker ikke en så ofte dataudveksling og datapakkerne er så små at det hele kører flydende og ligner det kører samtidigt på begge klienter uden forsinkelse. Vi oplevede dog problemer med netværksspil når man valgte at benytte en Wii Controller, hvor vi pludselig kom til at opleve kæmpe forsinkelser i spillet. Det var selvom overførselshastigheden var mere end rigeligt, afsendelses hastigheden af datapakker til ZigBee enheden som ikke kunne håndtere det højere pres. Når et *PullDevice* benyttes som input enhed, betyder det også at transmissionsintervallerne kommer til at ske meget oftere, 20ms, end med hardware knapperne. InputController er som standard sat op til at læse enhedernes status hvert tyvende millisekund, hvilket gav en meget større throughput end XBee enheden kunne klare.

SD-kort

Den 28. april 2009, kom en firmware-opdatering til Tahoe-II som gav bedre support for SD-kort. Tidligere havde det været sådan at det kun var en meget lille mængde SD-kort der var kompatible med Tahoe'en. For at få SD-kortet til at virke, måtte det ikke indeholde en MBR (Master Boot Record). Med den nye firmware supporterer Tahoe'en SD-kort op til 2GB, med eller uden MBR.

Da vi havde flashet tahoe'en med den nye firmware, fandt vi et program til Micro Frameworket der kunne teste læse- og skrive-hastighed på et SD-kort. Desværre blev der kastet en IOException og testen fejlede derfor. Vi lavede derefter vores egne små tests og fik kortet til både at skrive og læse filer, samt oprette nye mapper.

Vi lavede klassen SD som bl.a. giver spillet mulighed for at gemme en high score-liste på SD-kortet og huske spillerens sidste navn så man slipper for at taste det ind ved hvert spil. High score-listen er ordnet efter tid således at det går ud på at få den laveste tid. Efter hvert single player spil bliver den automatisk opdateret. Hvis man i spillets hovedmenu vælger "High Score" punktet, får man high score listen vist. Desværre er SD-kortet ikke særligt stabilt, så det er ikke altid at det virker. Problemet som vi ser det, er at kortet ikke altid bliver mounted korrekt når man sætter det i Tahoe'en. Selvom man får et *RemovableMedia_Insert*-event der indikerer at kortet er sat i, kan der gå et tilsyneladende vilkårligt antal sekunder indtil kortet er klar til brug. Prøver man at tilgå det inden da, får man en IOException.

MP3 Afspilning

VS1053

VLSI VS1053 er en lyddekoder der kan afspille lyd og musik, bl.a. i form af mp3, wma, wav og midi. Lyddekoderen har to SPI-busser; SCI (Serial Control Interface) som bruges til opsætning af registre, og SDI (Serial Data Interface) der bruges til at overføre data til dekoderen. De to SPI-interfaces deler input og output, men har forskellige chip-selects. Måden den virker på er at man først konfigurerer den vha. SCI. Bl.a. sætter man dens mode og clockfrekvens. Breakout-boardet fra Sparkfun har et eksternt 12.288MHz krystal, men for at kunne afspille høje sample rates, skal man sætte en multiplier så den interne clock kommer op på ca. 40MHz.

Chippen har en FIFO-kø på 2048 bytes som man sender data til over SDI-interfacet. Når der ikke er plads til mere i denne kø, eller chippen er ved at udføre en kommando sendt over SCI, bliver chippens *data request*-pin (DREQ) lav hvorefter man skal vente med at sende data indtil den bliver høj igen.

Måden vi havde tænkte os at bruge chippen på var at sætte den i stream-mode og så sende lyd-data til den med et fast interval, hvilket i chippens Application Notes refereres til som *Reliable streaming*. Inden vi kastede os ud i det, prøvede vi at læse fra chippens registre vha. SCI-interface hvilket mislykkedes. Lige meget hvad vi gjorde, returnerede alle registre 0. Ved at skrive til et bestemt register, fik vi chippen til at lave et såkaldt "software reset". Dette verificerede vi ved at hovedtelefonerne gav et kort klik. Det lykkedes os også at få chippen til at udføre en test hvor den giver en konstant tone, ved først at skrive til et kontrol-register og derefter sende bestemte data til chipen. Da vi senere prøvede at streame musik til chippen, gav den desværre kun skrattende lyde, selvom vi fulgte alle reglerne og guidelines i databladet og application notes. Vi har selvfølgelig prøvet med flere forskellige lydformater og sample rates, men lige lidt hjælper det. En grund til at det ikke virker kan være at vi ikke får konfigureret chippens interne clockfrekvens, men idet vi ikke kan læse chippens registre, er vi ikke sikre.

Konklussion

Det endelige spil kørende på TahoeII-boardet er blevet ganske udmærket, som nok kan anses for at være i et beta-stadie, og vi har nu fået en bedre idé for mulighederne i .NET Micro Frameworket og performance på Meredian CPU'en der har en clockfrekvens på 100Mhz. Vi lærte også lidt om begrænsningerne ved kun at have 8MB ram til rådighed og gav os en bedre indsigt til hvilke andre ting, man skal tænke over når man udvikler software til indlejrede systemer.

Tidligt i forløbet fik vi brug for at udvikle vores egne emulator-værktøjer da Tahoe-II samt .NET Micro Framework emulatoren simpelthen var utilstrækkelige. Vi manglede blandt andet seriel port mapping til computerens fysiske COM-port, hvor vi var nødt til at skrive vores egen emulator-komponent der gjorde det muligt at mappe en serielport i emulatoren til computerens fysiske serielporte. Denne komponent viste sig uundværlig under udviklingen af ZigBee kommunikationen, da vi ikke havde flere Tahoe-II boards til rådighed. Det var heller ikke den eneste emulatorkomponent som blev udviklet, der gik lang tid inden vi kunne få Wii controllers i hænderne, så på baggrund af specifikationer udregnet af personer på internettet, skrev vi også komponenter til at emulere ClassicControlleren, samt Nunchucken.

Vi har også set at når teknologien er ny, Micro Frameworket, så støder man også på flere fejl og mangler, end når man arbejder med en modent og gennemtestet framework som det fulde .NET Framework til PC. Projektet her gav os rig mulighed for at grave i mange hjørner af frameworket for at skabe de muligheder vi havde brug for og her stødte vi så også på fejl og mangler af forskellige årsager.

Til projektet er der flere mangler og mulige fejl som der ikke har været tid til at nå og rette eller gøre op for. Men vi har nået rigtig meget og lært en masse og når man ser på det endelige produkt vi endte med, virker den funktionalitet som er opnået også ganske god.

Litteraturliste

Digi. *XB24-Z7CIT-004 Datablad.* 3. Marts 2009.

NXP, og Phillips. *PCA9542A Datablad.* 24. November 2008.

VLSI. *VS1053b Datablad.* 20. Oktober 2008.

WiiLi. http://www.wiili.com/index.php/Wiimote/Extension_Controllers (senest hentet eller vist den 4. April 2009).