BACHELOROPPGAVE:

Sikker styring av nøkkelskap ved hjelp av USB

Safe management of key-locker using USB

FORFATTER(E): MAGNUS KOPPERUD, ANDREAS WAAL

Dato: 25.05.09

SAMMENDRAG AV BACHELOROPPGAVEN

Tittel:	Sikker styring av nøkkelskap ved hjelp av USB	Nr. : 1
	Safe management of key-locker using USB	Dato : 25.05.2009
Deltaker(e):	Andreas Waal ,Magnus Kopperud	
Veileder(e):	Høgskolelektor Halgeir Leiknes, Høgskolen i Gjøvik	
Oppdragsgiver:	Electric Time Car AS	
Kontaktperson:	Dag L. Solhaug	
Stikkord (4 stk)	USB kommunikasjon, Microchip PIC18, kretskort produksjon, kretskort utlegg	

Antall sider: 77	Antall bilag: 6	Tilgjengelighet (åpen/konfidensiell): Åpen

Kort beskrivelse av bacheloroppgaven:

Prosjektet er utført på oppdrag fra bedriften Electric Time Car(ETC). ETC har et system for administrasjon av bilparker som bl.a. benytter seg av elektronisk styrte nøkkelskap. Disse skapene har tidligere vært styrt av den eldre serielle standarden RS232.

Vi skal i denne oppgaven overføre denne styringen til den nyere og mer aktuelle standarden Universal Serial Bus(USB). Fordelen med USB fremfor RS232 er mange, der de viktigste er økt sikkerhet og mulighet for tilbakemeldingen fra skapet.

Rapporten følger hele utviklingsprosessen fra valg av komponenter, koding av software og firmware, PCB-utlegg og over til et endelig utlegg ferdigstilt for produksjon.

Du vil med andre ord i denne rapporten få informasjon om hva som skal til for og få en USB- kommunikasjon til og virke helt i fra grunnen av. Kritiske koblinger, viktige initsialiseringer og en god teoretisk innsikt i hvordan en Human Interface Device enhet opererer.

Forord

Vi skulle våren 2009 gjennomføre vår avsluttende bacheloroppgave. Totalt elleve oppgaver kom inn fra diverse bedrifter. Vi valgte etter nøye omtanke en oppgave gitt av bedriften Electric Time Car. Dette var en oppgave knyttet til deres system for håndtering av bilnøkler, CarAdmin. Hovedfokuset i oppgaven var på mikrokontrollere.
Grunnen til at vi valgte denne var muligheten til å jobbe med alle aspekter i en konstruksjonsprossess, helt fra valg av komponenter til kretskortutlegg. Dette var noe vi så på som meget motiverende da det ga mulighet for læring innenfor mange aspekter av en utviklingsprosess.

Vi vil gjerne takke:

- Veileder Halgeir Leiknes
- Oppdragsgiver ETC v/Dag Solhaug
- Vegard Johansen for utlån av USB demobrett, samt veiledning
- Arne Wold
- Lab ingeniør John Elvesveen for bestilling av komponenter
- Prototyp-ansvarlig TOPRO, Tom Andrè Skogsrud

Gjøvik, 25. Mai 2009

Magnus Kopperud

Andreas Waal

Innhold

Figurliste

Tabelliste

Formelliste

1 Innledning

1.1 *Bakgrunn for oppgaven*

Electric Time Car AS(ETC) har utviklet et system for administrering av nøkler i større bil parker. Dette systemet bruker et program utviklet av ETC med navnet BilPool. Dette programmet holder styr på alle bilene i en bilpark, der all informasjon om bilene ligger lagret. Denne informasjonen kan være km. stand, størrelse på bilen og lånehistorien på denne bilen.

Alle som skal benytte seg av systemet har hvert sitt unike brukernavn og passord. Når en person logger inn i programmet kan får de en bil tildelt ut i fra de ønsker de måtte ha om biltype. Videre tildeler programmet en bil til bruker ut i fra forhåndssatte algoritmer slik som km-stand. Dette hjelper til med å holde oversikt over alle bilene, hvem som har kjørt dem til enhver tid og ikke minst vil alle bilene kjøre tilnærmet like langt.

Dette systemet benytter av nøkkelskap i mettall produsert av Creone AB. Disse skapene har seks luker på hvert skap som kan innehold en nøkkel hver. Skapene har også en luke der det er mulighet til å plassere styringselektronikk.

Kommunikasjonen mellom PC-en hvor programmet kjører på og skapet skjer ved hjelp av standarden RS232.

1.2 *Organisering av rapporten*

Oppgaven er organisert i totalt 13 kapitler. Av disse er det tre kapitler som tar for seg utviklingsprosessen underveis i oppgaven.

Software – Dette kapitelet tar for seg alt som har med verts siden av systemet. Dette innebærer alt som skjer på PC siden, hvordan programvare er oppbygd og hvordan denne snakker med mikrokontrolleren.

Firmware – Dette kapitlet tar for seg alt vedrørende mikrokontrolleren. Dette innebærer teori rundt mikrokontrolleren, koding av mikrokontroller og hvordan denne kommuniserer med PCen.

Hardware – Dette kapitlet forklarer alle detaljer rundt det kretstekniske i prosjektet. Dette innebærer valg av komponenter, enkelte koblinger og kretskort utlegg.

Komplett liste over alle referanser finnes i kappitlet ”Bibliografi”.

1.3 *Oppgavebeskrivelse*

Systemet til ETC bruker RS232 grensesnitt for å styre skapene. Løsningen basert på RS232 har en rekke bakdeler. De tre største bakdelene er sikkerhet, tilbakemelding og styringsenhetens avhengighet av en PC.

RS232 er en standard som ikke krever noen form for oppsett mellom vert og slave før kommunikasjon kan starte. Dette gjør at man relativt enkelt kan manipulere signaler å åpne et nøkkelskap.

RS232 løsningen ETC har gir heller ingen tilbakemelding fra skapet etter utført handling. Problemet med dette er at etter avsendt kommando har man ingen anelse om åpningen av skapet gikk slik det skulle. Dersom noe går galt er det derfor umulig å finne ut om problemet sitter mellom PC og kontroller, eller kontroller og skapkrets.

Alle instruksjoner som har med styringen av skapet å gjøre blir med dagens system sendt i fra PC-en i nærheten av skapet. Dette er en løsning som gjør skapet veldig avhengig av denne PC-en.

1.3.1 **Oppgavemål**

I denne oppgaven skal vi konstruere et nytt kort for styring av skapene.

Dette kortet skal gjøres uavhengig av en PC. Det skal ha en egen strømforsyning å gå i en default mode som sørger for jevnlig nullstilling av samtlige skap. Sikkerhets og tilbakemeldingsproblemene skal løses ved at kortet vil benytte USB-kommunikasjon for å motta instruksjoner fra en PC. Vi skal forsøke å fremstille en komplett løsning som inneholder alt ETC trenger for å installere kortet i sine skap, dette innebærer software, firmware og hardware.

Hvis vi får god tid med å ferdigstille første delen av oppgaven, ønsker ETC å se på mulighetene for å gjøre systemet trådløst. På denne måten ville de kunne sende data for åpning av skap direkte i fra sin server, å slippe å gå om moder PC-en plassert ved skapene. Vurdering rundt løsingen av denne problemstillingen må tas på et senere tidspunkt når vi vet hvor mye tid som eventuelt ville bli til overs.

1.4 *Målgruppe*

Denne rapporten har to målgrupper.

Den først er oppdragsgiver. Denne rapporten skal være et komplett oppslagsverk å gjøre det enkelt for ETC å sette seg inn i hvordan systemet fungerer ved eventuelle endringer.

Den vil også være av stor interesse for personer som ønsker å konstruere en krets som benytter seg av USB-kommunikasjon og en mikrokontroller fra Microchip.

Rapporten er skrevet på en måte som krever teknisk forståelse rundt kretskonstruksjon og programmering, hovedsakelig i C.

1.5 *Arbeidsform*

Arbeidet ble utført i perioden 5. Januar 2009 til 24. Mai 2009. I denne perioden ble det jobbet alle ukedager og ofte helger. Studentgruppen utførte oppgaven parallelt med to fag, slik at arbeidsmengden på hovedoppgaven ble tilpasset disse fagene.

Gruppen har under hele arbeidsprosessen jobbet på elektrolaboratoriet (rom B104) ved HiG.

Det har vært møter med arbeidstaker med jevne mellomrom, hovedsakelig har møte vært blitt holdt hver 3. uke. Veileder har ved flere anledninger også deltatt på disse møtene.

Møter med veileder har blitt gjort fortløpende da gruppen og veileder har holdt til i samme bygg under hele prosjektperioden.

1.6 *Utstyr*

Følgende utstyr ble brukt til løsning av prosjektet.

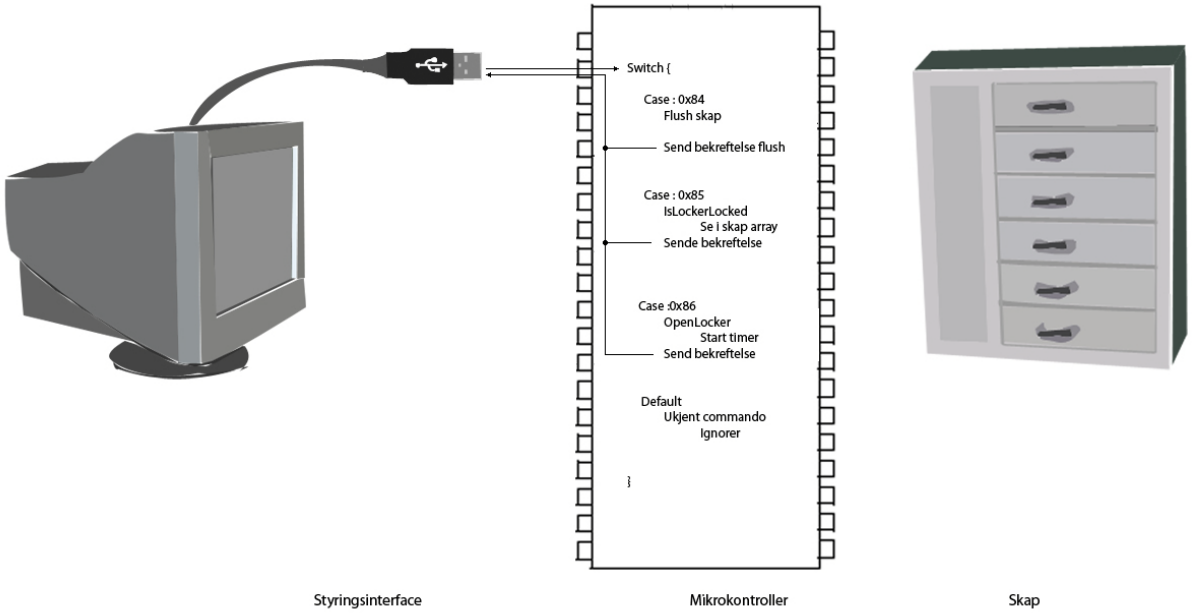
- Thurlby 30V-2A PL320 - Strømforsyning
- Philips PM3208 20MHz – Analogt Oscillator
- Microchip PICDEM Full Speed USB - Utviklingskort
- Microchip ICD2 – Programmer/debugger
- Fluke 37 Multimeter
- System for kretskortetsning, Elfa

2 Totalbildet

Electric Time Car har utviklet et system for administrasjon av bilparker kalt CarAdmin som de selger som en komplett løsning. Kundene er i dag for det meste kommunale avdelinger med flere biler som de må holde oversikt over. Systemet har som mål å gi enkel tilgang for brukerne, god oversikt over bruksdata og god sikring av bilene.

Løsningen baserer seg på metallskap produsert av Creone AB. Disse skapene har seks separate rom med hver sin elektronisk styrte lås. Til venstre på skapet er et rom for plassering av styringselektronikk. Selve åpningsmekanismen er styrt av et skiftregister, så dersom en kunde har behov for mer en seks låsbare skapplasser kan de ganske enkelt seriekople flere skap til de oppnår ønsket antall skapplasser.

Videre brukes et program kalt Bilpool som kjøres på en PC terminal i nærheten av skapet til å hente ut nøkler. Brukerne registrer seg da med brukernavn/nr og passord, ev. magnetstripekort og passord. I tillegg må de registrere forventet brukstid og spesielle bilønsker (varebil, personbil o.l.). I dagens system skjer kommunikasjonen mellom PC-en og kontrolleren i skapet via et serielt grensesnitt (RS232). Ulempene med dette er at PC-en ikke får noen tilbakemelding fra kontrolleren om utført oppgave, dersom noe skulle gå galt er det derfor veldig vanskelig å finne ut om feilen sitter før eller etter kontrolleren. Et serielt grensesnitt er også rimelig lett å manipulere, hvis man vet hva som skal sendes er det bare å koble seg inn på lederen til skapet å sette i gang. ETC har derfor ytret et ønske om å få denne kommunikasjonen over på USB. På den måten vil man være kvitt begge problemene rundt RS232. USB åpner for kommunikasjon mellom pc og enhet, og åpner i tillegg for enkel programvareoppdatering. Det er utviklingen av denne styringsenheten vi tok på oss i forbindelse med vår bacheloroppave.



Figur 1 - Flytskjema for kommunikasjon mellom vert og mikrokontroller

3 Nøkkelskapet

Skapet som ble brukt i oppgaven er det ETC bruker i sitt system. Valg av skapet hadde vi ikke noe med, vi måtte kun forholde oss til det slik det var.

3.1 *Spesifikasjoner*

- Leverandør: Creone AB
- Størrelse : 25cm x 28cm
- 7 luker
    - 6 stk - 14cm x 4cm tiltenkt nøkler
    - 1 stk – 5cm x 25cm tiltenkt styringskretser

3.2 *Styringskrets*

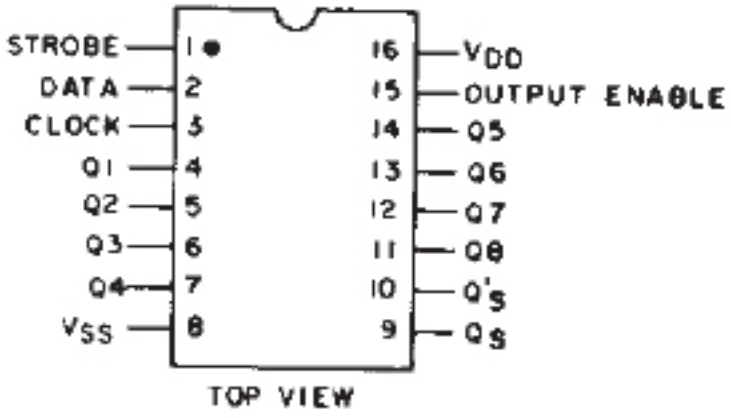
I skapet er det en krets som står for åpningen av luke tiltenkt nøkler. Dette skjer ved hjelp av et skiftregister og en forsterkerkrets som aktiverer en reele ved luken som skal åpnes.

For at skapet skal være operativt trenger det spenning for å drive styringskretsen og spenning til reelene. I tilegg må skiftregistret få pulstog som indikerer hvem luke som skal åpnes på skapet. Dette er dekket grundigere i firmware kapitelet.

3.2.1 **Skiftregisteret**

Skiftregistret er av typen CD4094B fra Texas Instruments. Dette registret har følgende inngange som vi måtte forholde oss til.

- Strobe – Når høy, flyttes data i minnet til utgangene
- Data – Data inngang
- Clock – Klokke inngang
- VDD – Drivspenning, 5 V
- VSS - Jord
- Output enable – Aktiverer utgangene på skiftregistre, 5V



Figur 2 – Pin diagram for skiftregistret, skjema henter fra (SCHS063, Texas Instrument Datasheet, 1998).

3.2.2 **Tilkobling til skapet**

Skapet blir tilkoblet ved hjelp av en 2x5 pins krysset parallell kabel. Hvordan denne er koblet kan sees i kapittelet som omtaler ”Hardware”.

Da det er mulig å seriekoble flere skap har styringskretsen i skapet både inngang og utgangs port. Seriekoblingen fungerer ved ta man kobler utgangen på et skap inn på inngangen til neste. Fordelen ved at man kun trenger en styringskrets på et system.

4 USB

I dette kapittelet skal vi prøve å redegjøre litt for hva USB er, teoriene bak USB-kommunikasjon, og vi skal se på de viktigste detaljene rundt denne formen for dataoverføring. Vi har i all hovedsak benyttet oss av to bøker og USB.org for å lære oss hvordan denne kommunikasjonen fungerer, det er også disse kildene som ligger til grunne for det følgende kapittelet. (Hyde, 2001) (Axelson, 2001)

4.1 *Fakta om USB:*

Universal Serial Buss er en seriell buss-standard for å koble enheter til en vert, standarden består av fire tilkoblingspunkter: spenning(Vcc), jord, og to datalinjer.

USB ble for første gang introdusert i 1994 av bla. Intel, Compaq, Microsoft, Digital, IBM og Northern Telecom. De forskjellige IT-aktørene samarbeidet om å utvikle en ny standard som skulle overta for datidens kaotiske system med RS232 og mange andre parallellsystemer.

Den offisielle lanseringen av USB1.0 kom i januar 1996, denne første utgaven kunne overføre med en hastighet på opp til 1,5 Mbit/s. I 1998 ble denne standarden utvidet og USB 1.1 var et faktum, endringene innebar ”full speed” noe som tilsier hastigheter på opp til 12Mbit/s. I tillegg ble det luket vekk en del problemer i forhold til å koble flere enheter til verten igjennom huber.

I april 2000 var det klart for den standarden som er mest utbredt i dag, USB 2.0 kan overføre med hastigheter på opp til 480Mbit/s. Disse standardene har i dag blitt omdøpt og vi kjenner de som:

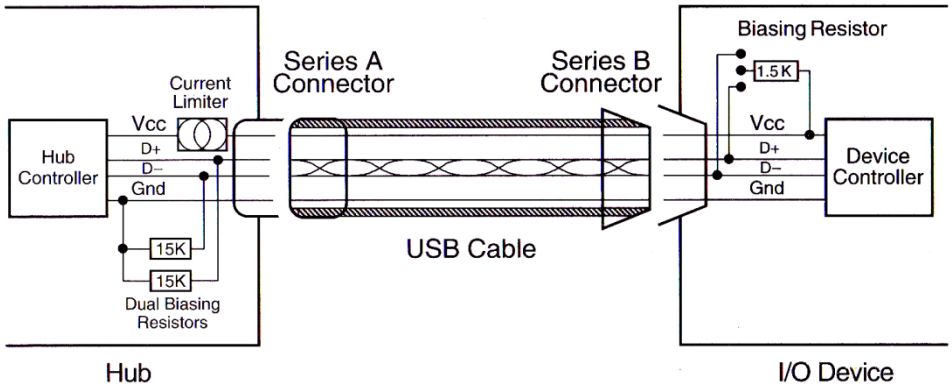
- USB 1.1 – 1.5Mbits/s
- USB 2.0 (full-speed) – 12Mbits/s
- USB 2.0 (High-speed) – 480Mbits/s

I november 2008 ble USB 3.0 (Super Speed) offisielt lansert som en standard, og overført til USB-IF. Denne standarden har en hastighet på 5Gbit/s og kapasitet på opp til 500Mbyte/s! De første enhetene som støtter denne standarden er ventet å komme i løpet av 2009-2010.

4.2 *Hvordan USB-kommunikasjon virker:*

4.2.1 **Tilkobling av en USB-enhet**

For å få til USB-kommunikasjon trengs en vert og en USB-enhet. Verten går og poller for å se etter nivåendringer på datalinjene, disse er i frakoblet tilstand lagt til jord igjennom to motstander på 15k. På USB-enheten er en av datalinjene trukket opp til VCC igjennom en motstand på 1.5k, hvis D+ er trukket opp betyr det at enheten er en høyhastighetsenhet og D- betyr lavhastighet. Når pluggen kobles til vil nivået på en av datalinjene på verten stige og tilkoblingen blir registrert.



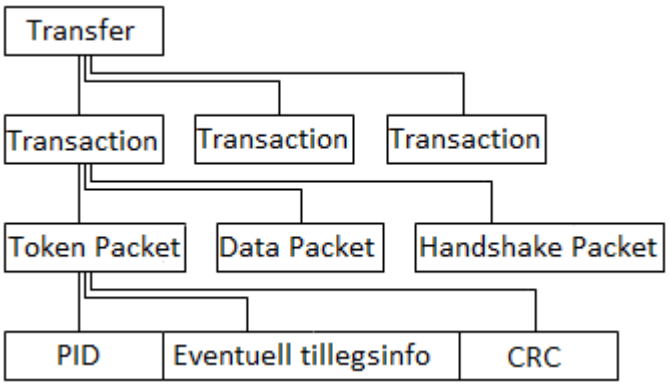
Figur 3- Bildet illustrerer en USB-tilkobling (Hyde, 2001) side 50

USB-enheten vil nå få tildelt adresse 0 av verten, og vil få tilsendt en etterspørsel om å sende sin device descriptor. Dette er en standardisert datapakke på 18 byte som inneholder informasjon om blant annet USB-versjon, Klasse, Produkt ID og Serienummer. Etter dette vil verten tildele USB-enheten en adresse, å sende en etterspørsel om device descriptor en gang til i påvente av eksakt samme svar som tidligere.

For å avslutte denne tilkoblingsprosedyren vil verten sende en etterspørsel om en konfigurasjons descriptor. USB-enheten svarer nå med alle de descriptorene som blir brukt for å gi en funksjonsbeskrivelse, og forteller verten nøyaktig hva som skal til for at enheten skal virke. Verten vil nå lete igjennom driverbiblioteket for å finne en driver som passer til enheten. I vårt tilfelle har vi valgt å programmere opp mot en Human Interface Device (HID) driver som ligger inne som standard i både Windows og OsX. Når alle disse stegene er fullført vil enheten være klar til bruk.

4.2.2 **Dataoverføring**

Selve dataoverføringen kan bestå av flere transaksjoner, en transaksjon består igjen av opp til tre pakker(Token, Data, Handshake).



Figur 4 - Flytskjema som viser en dataoverføring med USB. (Axelson, 2001)

Hver eneste av disse pakkene inneholder en packet identifier (PID), denne forteller hva slags pakke det er snakk om. Token-pakker blir brukt når verten ønsker å sette opp kommunikasjon med USB-enheten. PID vil i disse tilfellene inneholde informasjon om hvilken vei overføringen skal gå, om det er fra vert til enhet, enhet til vert, eller setup. Hvis det skal overføres flere datapakker vil også disse datapakkene inneholde sine PIDer. I disse tilfellene brukes PIDen til å fortelle hvilken del av en større

datarekke denne pakken tilhører. Den siste pakketypen som blir brukt i en transaksjon er handshake, denne pakken inneholder en statusrapport som forteller om forsendelsen var vellykket.

4.3 *Overføringsmetoder*

USB har fire forskjellige overføringsmetoder for å kunne oppfylle et hvert krav en USB-enhet måtte ha, om det er å overføre store datamengder, eller ha rask respons i forhold til eksterne kilder. De fire metodene er control, bulk, interrupt og asynkron overføring, og vi skal nå se litt på de forskjellige metodene.

4.3.1 **Control:**

Control basert overføring er den mest komplekse overføringsmetoden. Det er denne overføringen som sørger for å initsialisere enheten som skal tilkobles. Den består av tre ledd, setup, data og status. For hver eneste pakke som skal overføres må overføringen gå igjennom disse tre stegene, noe som gir veldig god kontroll på dataoverføringen. Den aller første pakken som sendes er en setup-pakke. Denne pakken er høyprioritert, og verten må svare på denne forespørselen selv om det innebærer å avbryte et påbegynt svar på en annen control-forespørsel. I denne setup sekvensen blir det bestemt hvor mye data som må overføres for at verten skal kunne tildele en driver. Dersom det er nødvendig med mer informasjon enn det som kan bli overført i data leddet i denne forsendelsen, vil det bli satt opp en overføring med flere pakker. Den siste pakken i en slik serie vil bli signalisert ved at denne datapakken er mindre en de andre. Siden denne overføringen er nødvendig for å sette opp USB-kommunikasjon er det avgjørende at alle USB-enheter har støtte for denne overføringstypen.

4.3.2 **Bulk:**

Denne overføringsmetoden er lavprioritert på bussen, dette betyr at hvis det er andre medier på bussen som har fastsatt overføringsrate må en overførsel sendt med bulk vente. Bulk opererer med garanti på overføringen av en pakke, noe som betyr at alle pakkene blir overført helt uten feil. Denne formen for overføring egner seg derfor bra til datatyper der det ikke er kritisk med overføringshastighet, men derimot at all data som kommer frem er lik den som ble sendt. Eksempler på dette kan være når vi sender en fil til en printer, eller mottar filer fra en skanner.

4.3.3 **Interrupt:**

Når vi overfører data ved hjelp av interrupt vil verten gå og polle på USB-enheten. Interruptstyrt overføring støtter alle USB-hastigheter og egner seg bra til enheter der det stadig skjer endringer. Størrelsen på filene som kan overføres er begrenset til 1024 byte i high-speed, så ved bruk av interrupt til dataoverføring er det viktig å begrense datamengden som overføres. Eksempler på ideelle former for dataoverføring av denne typen er ved overføring av tastetrykk på et tastatur, eller bevegelser på en mus.

4.3.4 **Asynkron overføring:**

Asynkron overføring er den eneste overføringsmetoden som har fastsatt leveringstid. Dette gjør den velegnet for datatyper man ønsker å overføre i ”real time”, en ulempe med dett er at det ikke er noen form for feilsjekk med mulighet for retting, så hvis man skal bruke denne typen overføring må man godta enkelte feil. Et bra eksempel på datatyper der det kan være aktuelt å bruke asynkron overføring er lyd og video filer.

4.3.5 **Klassifisering av en USB-enhet**

For at verten skal kunne tildele den tildele USB-enheten riktig driver er enhetene oppdelt i forskjellige klasser. Driverne kan ses på som en verktøykasse som forteller verten hvordan den skal snakke med den tilkoblede enheten. Denne klasseinndelingen er standardisert slik at driverutviklere kan se på klassekoden til en enhet å vite hvordan denne enheten overfører data uansett hvilken produsent som står bak.

Nr	Kode	Deskriptor som blir brukt	Type Enhet	Eksempel på enhet
1	00h	Enhet	Henviser til interface	-
2	01h	Grensesnitt	Lyd	Mikrofon
3	02h	Begge	Kommunikasjon og CDC	Ethernet, serielt
4	03h	Grensesnitt	HID	Mus, tastatur
5	05h	Grensesnitt	Fysisk	-
6	06h	Grensesnitt	Bilde	Webkamera
7	07h	Grensesnitt	Printer	Printer
8	08h	Grensesnitt	Masse lagringsenhet	USB-pinne, ekstern disk
9	09h	Enhet	USB-hub	USB-hub
10	0Ah	Grensesnitt	CDC data	Brukes sammen med nr 3
11	0Bh	Grensesnitt	Smart kort	Smart-kort leser
12	0Dh	Grensesnitt	Innholdssikkerhet	-
13	0Eh	Grensesnitt	Video	Webkamera
14	0Fh	Grensesnitt	Personlig helse	-
15	DCh	Begge	Diagnose enhet	-
16	E0h	Grensesnitt	Trådløs kontroller	Wi-fi, Bluetooth
17	EFh	Begge	Åpen enhet	-
18	FEh	Grensesnitt	Applikasjon-spesifikk	-
19	FFh	Begge	Leverandør-spesifikk	-

Tabell 1-Oversikt over klasser og klassekode (USB.org, 2009)

4.3.6 **Klasse forklaring:**

I dette avsnittet skal vi prøve å forklare de forskjellige klassene i tabellen over.

1. Den første descriptoren som blir tilkalt i fra verten er device descriptoren, men som vi ser av tabellen bruker mange av enhetene interface (grensesnitt) descriptoren til å fortelle verten hvilken klasse de tilhører. Denne klassen brukes derfor til å fortelle verten at informasjonen om enheten kommer i interface descriptoren.
2. Denne klassen er tilpasset overføring av lyd.
3. Communication device class (CDC) er en klasse som blir brukt til enheter som driver en eller annen form for kommunikasjon, eksempler kan være Ethernet, telefon, eller et modem. Denne klassen kan ta i bruk begge klasser for kommunikasjon, både device og interface.
    - Device class brukes av verten til å identifisere en kommunikasjonsenhet som benytter flere forskjellige grensesnitt.
    - Interface class brukes når man ønsker å ta i bruk de typene kommunikasjon som USB tilbyr, du vil igjennom denne klassen få tilgang til samtlige.
4. Human Interface Device (HID) er en klasse som brukes til å registrere en brukers bevegelser, dette være seg tastatur, mus eller en joystick.
5. Physical Interface Device (PID) er en utvidelse av HID, den bruker samme grensesnitt, men er beregnet for å kunne overføre data fra en trykksensitiv bryter, altså ikke bare ett trykk, men også hvor hardt trykket er.
6. Denne klassen er definert for overføring av bilder.
7. Klassen er beregnet for kommunikasjon med printere, eller liknende enheter.
8. Standard utviklet for ekstern tilkobling av harddisker. I utgangspunktet var det tiltenkt magnetiske disker, men standarden har blitt utvidet til å støtte mange flere standarder der flash er den største.
9. Klasse for direkte videreføring av et signal, når verten går videre inn i klassesystemet (base class, sub class, protocol) vil den få beskjed om hva slags hub som er tilkoblet, og hvilke hastigheter denne huben kan operere med.
10. Se klasse 3.
11. Denne klassen er utviklet med tanke på overføring av data fra et ICC(integrated chip card) igjennom en kortleser. Verten vil først opprette kontakt med kortleseren, og etter dette vente på at et ICC-kort blir satt inn. Et eksempel på slike kort er bankkort med integrert chip eller et SIM-kort.
12. Klassen tilbyr en beskyttet og kontrollert overføring av data igjennom et fellesbibliotek av forskjellige descriptorer og forsendelses anmodninger. Hvilke descriptorer som brukes for hver forsendelse avgjøres før forsendelsen. På denne måten vil dataene bli beskrevet separat, men samtidig over en felles lest.
13. Denne klassen er beregnet for overføring av video.
14. Personlig helse (Personal hygiene) klassen er utviklet mot bruk i helsesektoren, for eksempel ved å koble en blodtrykksmåler til en PC eller direkte til et medisinsk apparat.
15. Finn ut mer
16. Denne klassen har et brukergrensesnitt tilpasset trådløs kommunikasjon.
17. Som navnet tilsier er dette en åpen klasse. Den viser til forskjellig funksjonalitet alt etter hva som bestemmes av underklasse og protokoll.

18. Dette er en klasse som brukes til bla. test av applikasjon og firmware oppgraderinger, også i dette tilfellet bestemmes funksjonaliteten av underklasser og protokollen.
19. Denne klassen kan leverandørene bruke som de vil.

Av alle disse klassene måtte vi velge en som ville passe til vårt prosjekt. Dataene som skal sendes i mellom PC-en og skapet er begrenset til skap nummer, dør nummer, hvor lenge skapet skal stå oppe, status og manuell flush. Dette er veldig små datamengder, og siden det ikke ble stilt noen andre krav fra oppdragsgiver enn at det hele skulle fungere på Windows uten videre modifikasjoner lyste HID klassen seg tidlig ut som en sterk kandidat for den kommunikasjonen vi var ute etter. Denne klassen dekker veldig mange områder innen USB-kommunikasjon, som tidligere nevnt er både tastatur, mus og joystick alle eksempler på enheter som bruker dette grensesnittet. I tillegg til disse har vi også flere typer sensorer for lys, lyd og trykk (PID) som bruker eksakt samme grensesnitt. Selv om navnet på driveren får det til og høres slik ut er ikke grensesnittet avhengig av noen form for menneskelig berøring, så lenge datamengden som sendes ikke er for stor eller har for høy klokkefrekvens kan de fleste enheter benytte dette grensesnittet. HID grensesnittet er også det enkleste og ha med å gjøre i denne sammenhengen, og krever lite initialisering i forhold til mange av de andre klassene.

4.4 *HID descriptor*

PC-en har nå lest av device og interface deskriptoren, den vet hvilken klasse enheten tilhører og vil nå søke etter en HID deskriptor og en rapport deskriptor.

HID

Lengde = 9
Type = 21H
HID Versjon
Språk kode
HID descriptorer
Rapport = 22H
Total Rapportlengde =J

Rapport

Rapport 0
Rapport 1
Rapport 2
-
Rapport J-1

Tabell 2- Oversikt over HID descriptoren

Tabell 3 - Oversikt over rapportdescriptoren

HID descriptoren har i utgangspunktet fastsatte verdier for de fire første blokkene (9, 21H, 100H, 0). Men dersom man skal lage en språkavhengig enhet, for eksempel et tastatur skal koden for det respektive landet legges i den fjerde boksen merket språk kode. Boksen for HID descriptorer(5) inneholder antallet HID descriptorer som følger, en mus kan eksempelvis inneholde flere bevegelsessensorer. Trykknapper, skrolle-hjul, og optiske sensorer har alle sine descriptorer. Boks

nummer 6 har fastsatt verdi til 22H, noe som betyr at det følger en eller flere rapport descriptorer. Den siste boksen forteller hvor mange rapportdescriptorer som følger.

Rapport descriptoren er en viktig brikke for å få til USB-kommunikasjon, Denne bestemmer nemmelig hvilken protokoll som skal brukes for dataene som sendes. Protokollene som brukes er ofte fastsatt i operativsystemet som blir brukt, og utfordringen blir i disse tilfellene å sende data på den måten verten forventer å motta den.

I vårt tilfelle var vi så heldige at Microchip allerede hadde utviklet et rammeverk for HID grensesnittet, så vi slapp å sette opp alle disse tingene manuelt. Dette var til betydelig hjelp da det ga oss umiddelbart innsyn til USB-kommunikasjon, og hjalp oss til å komme fort i gang med denne delen av utviklingen. ETC hadde noen bekymringer angående tilbakemeldingene når man tar i bruk HID grensesnittet, disse gikk ut på at det var mulighet for at tilbakemeldingene til PC-en ville bli sendt til det vinduet som måtte stå oppe. Vi utførte flere tester for å se om dette var tilfellet, men fant ingen problemer med dette.

5 Mikrokontrolleren

5.1 *Produsentvalg*

Vi hadde fra tidligere prosjekter jobbet mye med Microchip sine kontrollere. Det var derfor et naturlig valg for oss å undersøke muligheten for å bruke en av deres kontrollere i dette prosjektet. Våre tidligere erfaringer tilsa at hvis muligheten til fritt valg av kontroller produsent er til stedet, så er Microchip et godt valg. De er en av de største aktørene innenfor mikrokontrollere, noe som tilsier at det finnes store mengder ressurser å hente, både igjennom internett og annen litteratur.

Av andre produsenter vi hadde i tankene var Atmel og Nordic, da disse produsentene var produsenter vi hadde hørt mye om.

Det første vi måtte avgjøre når det kom til valget av kontrolleren var størrelsen i form av kompleksitet. Både Atmel og Microchip leverer alt fra små 8-bit kontrollere til større 32-bits kontrollere. Til vårt formål var det kravet om USB som var hovedkriteriet.

Etter en innledende runde med forskning kom vi frem til at både Atmel og Microchip hadde 8-bits kontrollere som vi kunne benytte oss av. Disse 8-bits kontrollerne er små effektgjerrige chiper som leverer hastigheter opp mot 20MIPS (Mega Instruksjoner Per Klokketakt), til en pris som gjør dem attraktive for design av den typen vi skal lage. Dette er opp mot større 32-bits kontrollere ikke spesielt imponerende hastigheter, men til vårt bruk var ikke hastigheten det essensielle.

Etter å ha undersøkt nærmere kom vi frem til to alternative chiper til vårt prosjekt.

Atmel AVR AT900USB1286

- 48 pins
- 16 MHz - 16 MIPS
- Innebygd USB kontroller
- 8/16 bit timers
- USART, SPI, I2C

Microchip PIC18F4550

- 44 pins
- 48 MHz- 12 MIPS
- Innebygd USB kontroller
- USART, SPI, I2C

De to alternativene er nærmest identiske, og siden vi igjennom tidligere prosjekter har mye erfaring med Microchip sine kontrollere der spesielt deres dokumentasjon har vært utmerket valgte vi PIC18F kontrolleren.

5.2 *Microchip PIC18F4550*

PIC18F4550 er som nevnt over en kontroller fra Microchip sin 8-bits serie. PIC18F serien er Microchip sin kraftigste i 8-bit klassen og inneholder en rekke forskjellige chiper som tilbyr alt fra helt simple kontrollere med inn og utganger, til innebygde ZigBee, Ethernet og USB kontrollere.

Blant de aktuelle chipene var det to serier som pekte seg ut. Det var 18FXX5X serien som består av 2455,2550,4455 og 4550 eller 18FXXJ50 serien. Av disse to seriene kom vi frem til at 18F4550 var det beste alternativet for oss.

Grunnen til dette er at Microchip har et utviklingsbrett til USB som er basert på denne chipen. Dette ga oss mulighet til å teste ut firmware og oppkoblingsalternativer før vi lagde et kretskort. Dette gjorde også at vi kunne begynne utviklingen av drivere og firmware parallelt med arbeidet rundt kortutlegg.

5.2.1 **Microchip PIC18F4550**

- USB V2.0 Low Speed (1.5 Mb/s) og Full Speed (12 Mb/s)
- Støtte for Control, Interrupt, Asynkron og Bulk overføring
- Klokkehastighet opp til 48Mhz noe som tilsvarer 12 MIPS.
- 35 Input/output porter
- Seriell kommunikasjon i form av SPI, I2C og USART
- 2 komparatorer
- 8/16 bit timere – Totalt 4 timer moduler
- 10-bit A/D konverter
- Mulighet for ICSP (In Circuit Serial Programming) via to pinner

5.3 *Microchip USB rammeverk*

For at en enhet skal kunne kobles til en PC via USB trenger man en USB stack. En USB stack tar seg av kommunikasjonen mellom enhet og driver på det laveste nivået. Det vil si hvilke data som går ut av kontrolleren. Da USB har en kompleksitet som går langt over andre standarder tradisjonelt brukt i mikrokontrollere trengs det også mye mer for å få opp kommunikasjonen.

Der man for eksempel i I2C kun trengte å senke datalinjen på et bestemt sted i en klokketakt, så krever USB flere steg med initialisering (Token, data, handshake). Der man ved I2C trengte to linjer med kode for å starte en sending, så inneholder USB stacken på kontrolleren sin side flere sider med kode kun for initialisering og oppsett. Når man skal sette opp det samme på PC-en for å få til kommunikasjon tilbake sier det seg at kompleksiteten begynner å bli rimelig stor.

Heldigvis har Microchip som leverandør av kontrollerne et USB rammeverk som det er mulig å benytte seg av. Det gjør at man som designer av en krets kan konsentrere seg om sendingen av data på et høyere nivå. Microchip sitt rammeverk har blitt utviklet i mange år nå og har når dette blir skrevet kommet til versjon 2.3. Rammeverket inneholder eksempler på mange forskjellige USB klasser, deriblant Generic Device klasse, HID device klasse og Mass Storage Device klasse.
Som tidligere nevnt fant vi ut at HID var den klassen som var mest aktuell for oss, dette grunnet den gode implementeringen av HID drivere i operativsystemer.

Dette rammeverket er et rammeverk som er laget til å passe til alle Mikrochip sine kontrollere med USB, den store utfordringen for oss var rett og slett å luke ut det som ikke var aktuelt å bruke i vårt prosjekt.

Selve rammeverket består av to hoveddeler, den første er den tidligere nevnte USB stacken. Denne tar seg av selve oppkoplingen og sendingen av data. Den andre delen er den logiske strukturen til rammeverket. Dette er den delen som tar seg av sendingen av selve bruker dataen. Det er denne delen av rammeverket vi har jobbet direkte med. Da stacken kun legger grunnlaget for resten av sendingen var det ikke behov for modifikasjon av denne.

Rammeverket legger opp til bruk av USB 2.0 Full Speed. Vi valgte derfor å fortsette med dette.

6 Software

Med software menes den programvaren som er på PC siden av applikasjonen. I vårt tilfelle vil vi ikke selv stå for selve softwaren da vårt system skal implementeres i en allerede eksisterende software. Vår oppgave ble dermed å legge til rette for denne implementeringen.

6.1 *ETC sine ønsker om software*

ETC har som nevnt tidligere et program som heter CarAdmin som står for selve styringen av skapene. Dette programmet er et komplekst program som tildeler bruker bilnøkler ut i fra bestemte algoritmer. Da omfanget på dette systemet er meget stort måtte vi finne en måte å få inn de rutiner som trengtes for å styre kortet på en så enkel og ryddig måte som mulig.

Vi kom frem til at den beste løsningen på dette var å flytte alt inn i en. DLL fil.

DLL(Dynamically Linked Library) er et eksternt bibliotek som man kan hente mange forskjellige rutiner fra. I vårt tilfelle var vi interessert i å hente funksjoner som sto for styring og initialisering av skapet. Det som skiller en DLL fra for eksempel en header er at .DLL filer er dynamiske. Dette vil si at man laster inn. DLL filen under kjøring av programmet i motsetning til statiske biblioteker der man kompilerer biblioteket inn i selve programmet. Fordelen med dette er at .DLL filer kan ligge separert som en selvstendig fil.

6.2 *ETC sine ønsker om styringsinterface*

Selve softwaren skulle fra vår side som nevnt over bli plassert i en .DLL fil. Denne .DLL filen skulle inneholde alle de funksjonene som ETC trengte for å styre skapet. Totalt fire funksjoner trengtes for at ETC skulle kunne gjøre det de ønsket.

int init();

Sammendrag	*:*	*Initialiserer Nøkkelskapsgrensesnittet.*
Parametere	*:*	*Ingen*
Returverdi	*:*	*Ingen*

bool openLocker(int locker, int door, int duration);

Sammendrag	*:*	*Åpner et nøkkelskap*
Parametere	*:*	*Skapnummer, dørnummer, tid skapet er åpent*
Returverdi	*:*	*Sann hvis skapet ble åpnet, usann hvis noe gikk galt*

bool flushLockers();

Sammendrag	*:*	*Tving alle dører til å lukke ved å sette 0 i hele shiftregisteret*
Parametere	*:*	*Ingen*
Returverdi	*:*	*Sann hvis flush gikk ok, usann hvis noe gikk galt*

bool isLockerClosed(int locker, int door);

Sammendrag	*:*	*Ser om en bestemt dør i et bestemt skap er lukket*
Parametere	*:*	*Skapnummer, dørnummer*
Returverdi	*:*	*Sann hvis dør er lukket, usann hvis dør er åpen*

6.3 *Microchip rammeverk*

Som nevnt har Microchip et rammeverk til USB-design. Dette rammeverket inneholder også eksempler på applikasjoner som kan brukes på PC siden av designet.

Dataene vi sender til og fra kontrolleren er kun enkeltstående bytes som blir tolket ut i fra de forhåndsbestemte reglene vi har satt. Softwaren sin oppgave blir å sende kommandoer til kontrolleren, for så å videreformidle tilbakemeldingen fra kontrolleren til bruker.

Det programmet som følger med rammeverket inneholder rutiner for å gjøre nettopp dette. Det eneste problemet vårt var at til tross for at rutinene for sendingen var tilstedet, så var disse viklet inn i et allerede eksisterende program. Dette programmet var et Visual C++ program med Windows GUI. Utfordringen vår på denne siden var å luke ut den koden vi trengte for vår kommunikasjon, konvertere det fra Visual C++ med Windows GUI og over til et en ren DLL fil som kan kalles fra et annet program.

Dette programmet inneholder også en annen meget essensiell del som vi trengte for å få i gang vår USB kommunikasjon. I Windows styres mange av de fundamentale delene i operativsystemet av API(Application Programming Interfaces) funksjoner. Det er disse funksjonene som setter opp kommunikasjonen med HID-driveren som kontrolleren benytter til og kommunisere med PC-en.

Selve sendingen av data foregår ved hjelp av to arrayer, som hver er på 65 bytes, der den første byten ikke blir sendt.

```
unsigned char OutputPacketBuffer[65];
unsigned char InputPacketBuffer[65];
```

Et eksempel på en forenklet sending kan være at man først skriver til Output arrayen. Her starter man alltid med å skrive en 0 til plassering [0] i arrayen. Etter dette skriver man de reelle data til plassering [1] og utover. Etter at man har skrevet de data man ønsker kaller man funksjonen som står for selve sendingen av dataen. Denne heter WriteFile(..). Eksempel på kall av denne kan ses under.

```
WriteFile(WriteHandle, &OutputPacketBuffer, 65, &BytesWritten, 0);
```

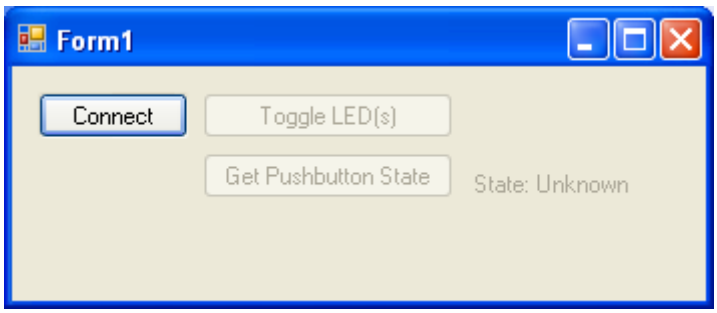
Funksjonen tar seg av sendingen, med parametrene som sier hva man skal sende, hvilke buffer man skal sende og lengden av det man skal sende.

På grunn av høy hastighet i tillegg til små datapakker kan man umiddelbart lese av resultatet man får tilbake fra kontrolleren. Dette gjøres ved følgende funksjon:

```
ReadFile(ReadHandle, &InputPacketBuffer, 65, &BytesRead, 0);
```

Denne funksjonen leser fra USB enheten og plasserer rusultatet i *InputPacketBuffer* arrayen. Den mottatte dataen må nå behandles på et vis. Det vanlige vil da være å ha en switch eller en else if som handler ut i fra det man forventer å motta.

Det programmet som følger med rammeverket er et enkelt program som kun består av tre knapper. Den første av dem setter opp selve koblingen mellom programmet og driveren, blant annet ved å benytte seg av de tidligre nevnte API funksjonene. De to andre knappene er for sending av data.



Figur 5 - Microchip eksempelprogram sett fra brukers perspektiv

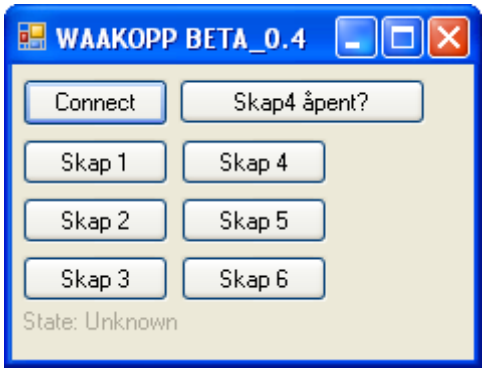
Under utviklingen hadde vi behov for fortløpende å teste softwaren. Siden vi på dette tidspunktet ikke hadde en DLL fil klar, modifiserte vi Microchip sitt test program til å sende de data vi selv ønsket. Da vi allerede hadde rammeverket til programmet innebar dette å legge til flere knapper som sendte andre data.

Før vi kunne starte testingen måtte vi lage et regelsett for tyding av kommandoer. Dette vil si at vi måtte bestemme hva en byte skulle inneholde for at f. eks et skap skulle åpnes.

Kommando	**Beskrivelse**
0x84	FlushLocker function
0x85	IsLockerClosed function
0x86	OpenLocker function

Tabell 2- Kommandoliste

Kommandoene over er de kommandoene som Mikrokontrolleren er satt opp til å se etter. Dette innebærer at alle andre kommandoer utenom de tre over vil bli ignorert. Det vi gjorde videre nå var å lage et testprogram som hadde mulighet til å åpne alle luker på test-skapet vårt ved hjelp av funksjoner med faste parametere.



Figur 6 - Testprogrammet brukt under utviklingen

Dersom en av knappene markert med skap1-6 blir trykket på vil programmet tilkalle funksjonen openLocker med skapnummer, dørnummer, og en int som bestemmer hvor lenge skapet skal stå oppe.

```
EXPORT bool openLocker(int locker, int door,int duration){

   DWORD BytesWritten = 0;                  // Variabel for å holde styr på bytes skrevet/lest
   DWORD BytesRead = 0;
   unsigned char OutputPacketBuffer[65];    // Array for sending av data, data her BLIR SENDT
   unsigned char InputPacketBuffer[65];     // Array for mottak av data.
   OutputPacketBuffer[0] = 0;               // Alltid '0' i  denne, blir IKKE sendt
   OutputPacketBuffer[1] = 0x86;            // Første byte som blir sendt. 0x86 = ÅPNE SKAP
   OutputPacketBuffer[2] = locker;          // Hvilket skap. Pga få skap: OK med dec. verdi
   OutputPacketBuffer[3] = door;            // Hvilken dør/luke på dette skapet
   OutputPacketBuffer[4] = duration;        // Hvor lenge, I SEKUNDER

WriteFile(WriteHandle, &OutputPacketBuffer, 65, &BytesWritten, 0);
                                            // Writefile. Tar seg av SENDINGEN.
ReadFile(ReadHandle, &InputPacketBuffer, 65, &BytesRead, 0);
                                            // ReadFile. Tar seg av MOTTAK. SKjer umudelbart
    if(InputPacketBuffer[2] == 0x01)        // Leser av mottatt data fra kontroller.

   return true;                             // Første data av interesse er [2] i arrayen.
                                            // [0] er report ID, bryr oss ikke.
                                             // [1] er ekko av sendt kommando.

    else                                    // Skap ikke åpnet OK. Returner false.

    return false;

        }
```

Som man kan se i koden over så er siste delen av funksjonen en "if else" løkke. Denne tar i mot svaret fra kontrolleren, og da vi vet at kontrolleren sender resultatet "0x01" ved positivt resultat kan vi trekke konklusjonen at alle andre svar er feil for oss. De øvrige funksjonene vi benytter for å gi kommandoer til kontrolleren fungerer på samme måte som vist over, eneste forskjell er at de tar i bruk andre parametere. Vi velger derfor å hendvise til vedlegg D for å se kodingen rudt disse funksjonene.

6.4 *Konvertering til DLL fil*

Konverteringen til DLL var det vi kanskje sleit mest med rundt software delen. Ikke fordi dette hadde en alt for høy kompleksitet, men rett og slett fordi dette var noe vi ikke hadde noe erfaring med. Derfor ble det brukt en god del tid for å lese seg opp rundt dette. På internett fant vi flere måter å lage DLL filer, men den greieste måten å gjøre det på fant vi på forumet (Techguy.org), for URL se kildehenvisning.

Essensen i det å lage en DLL fil er meget enkel. Etter kompilering ender man opp med en .Dll fil i stedet for en programfil.
Da en DLL er et bibliotek så er det allikevel noen små endringer som må gjøres i programkoden. En veldig viktig ting å tenke på er at alle funksjoner skal eksporteres ut. Derfor må man deklarere funksjoner som eksport funksjoner. Ved deklarasjon av den tidligere nevnte openLocker funksjonen ble nå deklarasjonen som følgende.

```
   EXPORT bool openLocker(int locker, int door,int duration);
```

Dette må gjøres ved alle funksjoner, slik at kompilatoren vet at denne funksjonen skal være mulig å kalle utenfra. Koden for hele DLL filen kan sees i vedlegget merket D.

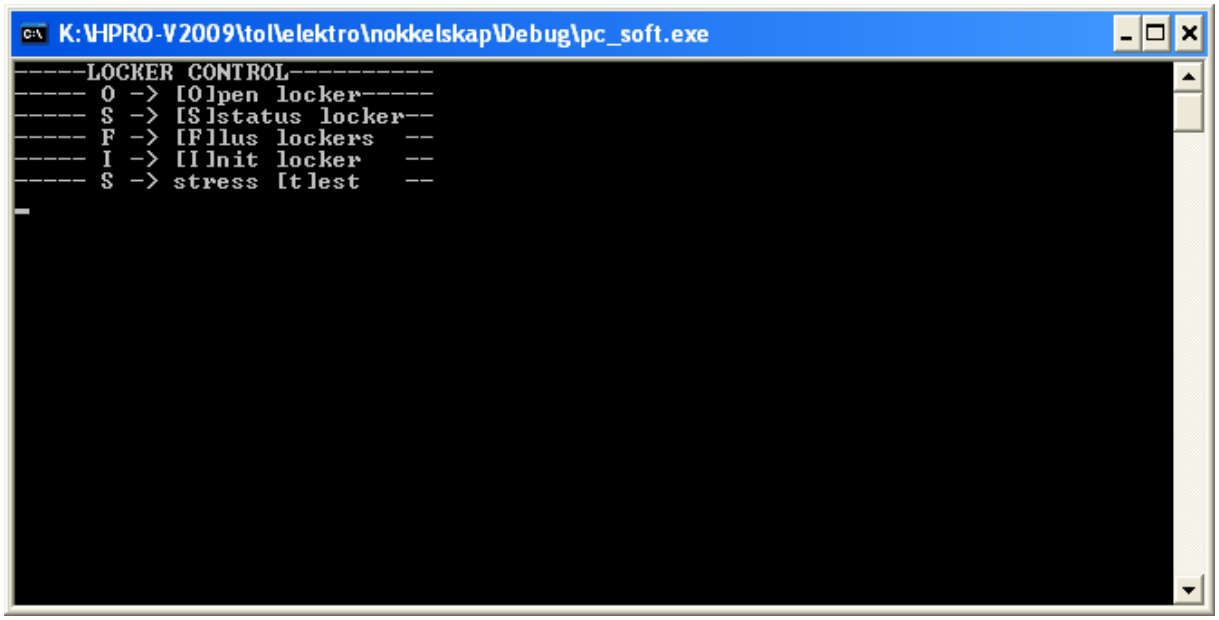
6.5 *Importering av DLL fil*

Når vi nå hadde fått til å lage en .DLL fil var det kritisk at vi også lærte oss å bruke den, dette var den eneste måten vi kunne kvalitetssikre filen å forsikre oss om at den fungerte slik ETC ønsket. Det testprogrammet vi hadde brukt frem til dette tidspunktet hadde en stor ulempe, det var hardkoda og kunne derfor kun åpne de første seks skapene. For å tilnærme oss en mer realistisk bruk av programmet besluttet vi derfor å lage et lite ”console” test program. Dette testprogrammet skulle tilkalle alle funksjonene som blir benyttet fra .DLL-filen vi hadde laget, i tillegg skulle det motta kommandoer i fra tastaturet slik at vi kunne bestemme hvilket skap, hvilken dør, og hvor lenge det skulle stå åpent ved hjelp av noen enkle tastetrykk.

Den første utfordringen i forhold til dette ble å importere funksjonene fra .DLL-filen inn i programmet. For å fortelle til prosjektet vårt hvor det kunne finne .DLL-filen måtte vi i Visual C++ legge den inn under prosjektegenskaper. Etter dette må man vite navnet på funksjonene man ønsker å importere, når vi nå vet hvilke funksjoner vi ønsker å benytte må disse defineres.

```
extern EXPORT bool isLockerClosed(int locker, int door);
extern EXPORT bool flushLockers();
extern EXPORT bool openLocker(int locker, int door,int duration);
```

Når dette var gjort kunne vi ganske enkelt skrive de funksjonene som måtte til for å få det brukergrensesnittet vi ønsket, funksjonene over kunne nå enkelt tilkalles som en del av programmet.



Figur 7 - Console testprogram til skap

7 Firmware

Med firmware menes den programvaren som styrer en elektronisk enhet. Dette kan dreie seg om alt fra kalkulatorer, fjernkontroller til vårt eksempel en PIC mikrokontroller. Dette kapitlet vil ta for seg alt som har med selve programmeringen av mikrokontrolleren. Dette innebærer innstillinger av kontrolleren, styring av skapet, samt styring av diverse moduler på mikrokontrolleren.

7.1 *Utviklingsverktøy for mikrokontrolleren*

Utviklingsverktøy innebærer utviklingsprogrammet og kompilatoren. I vårt tilfelle er man avhengig av at begge deler har støtte opp mot Microchip sine kontrollere, og helst også støtte for vår kontroller 18F4550.

7.1.1 **Utviklingsprogram – MPLAB IDE**

Når man skal utvikle en mikrokontroller fra Microchip så er det eneste reelle alternativet etter vår mening Microchip sin egen software MPLAB IDE. Dette er et program som Microchip har utviklet siden 2001 og stadig kommer med nye forbedringer og oppdateringer til. Dette har god støtte for alle deres kontrollere samt en rekke programmeringsmoduler og feilsøkingsmoduler. Det har også støtte for de fleste kompilatorer innenfor C, Basic og assembly.

I vår utviklingsprosses benyttet vi oss av versjon 8.10 av programmet.

7.1.2 **Kompilator**

Innenfor kompilatorer er det flere alternativer. Vi visste fra før av at vi ønsket å programmere på høy nivå, da helst C eller C++. Her finnes det flere gode kompilatorer å velge mellom, de mest relevante alternativene for oss var B Knudsen CC8E, CSS C og Microchip C18 kompilator.

En av de kritiske faktorene ved dette prosjektet var muligheten til å benytte seg av Microchip sin USB stack. For at dette skulle være mulig var vi avhengig av å benytte oss av Microchip sin egen C18 kompilator, dermed ble kompilator valget også veldig enkelt for oss.

7.1.3 **Microchip C18 kompilator**

Fordelen med denne kompilatoren er som med de fleste andre ting fra Microchip meget god dokumentasjon. Microchip tilbyr både en oppstarts guide samt en bruker guide som totalt inneholder nærmere tre hunder sider med informasjon rundt kompilatoren.

Denne finnes i to versjoner, en kommersiell versjon samt en gratis student versjon. Student versjonen er i utgangspunktet identisk med den kommersielle versjonen, men etter seksti dagers bruk mister man deler av kode optimaliseringen ved kompilering. Dette betyr at firmwaren man har rett og slett vil ta opp mer plass på kontrolleren. Så hvis man har en mikrokontroller med lite programminne ville dette vært kritisk, i vårt tilfelle har vi veldig god plass så dette var ikke noe problem.

En annen meget god ting med denne kompilatoren er Microchip sitt bibliotek med styringsfunksjoner for de forskjellige mulighetene kontrolleren har, slik som timere, bus systemer, komparatorer osv. Dette gjør at man har mulighet til å aktivere f. eks en timer å være sikker på at dette blir gjort riktig. Man får da luket bort et feil element og kan konsentrere seg om selve programmeringen.

7.2 *Programmering av mikrokontrolleren*

Med programmering av mikrokontrolleren mener man selve programmeringsprosessen. Dette innebærer hvordan man får firmwaren over på kontrolleren. Det er flere metoder å gjøre dette på og det finnes et stort antall av forskjellige moduler som gjør jobben.

7.2.1 **In Circuit Serial Programming (ICSP)ved hjelp av ICD 2**

Den vanligste måten å programmere en PIC kontroller under en utviklingsperiode er ved hjelp av ”In Circuit Serial Programming” fra nå av omtalt som ICSP.

Dette gjøres på en meget enkel måte ved å koble seg opp på to programmerings pinner på kontrolleren.

Dette er på vår kontroller pinnene PGC og PGD. Som er forkortelser for ”Programming Clock” og ”Programming Data”.

For å kunne programmere ICSP er man avhengig av en modul som står for kommunikasjonen mellom PC og kontrolleren. I vårt tilfelle benyttet vi oss av Microchip sin egen programmering og feilsøkingsmodul ICD2. ICD2 står for ”In Circuit Debugging 2”, men til tross for navnet så er man ikke begrenset til feilsøking, man kan også programmere. Dette er kun en av flere moduler som finnes på markedet, men vi valgte denne grunnet gode erfaringer.

Til tross for alt selve programmeringen kun skjer over to pinner, så skal ICD2 enheten kobles til kontrolleren ved hjelp av seks linjer. Disse blir vanligvis koblet til ved hjelp av en RJ45 kontakt eller en RJ11 kontakt, men kan også kobles til på andre måter.

7.2.2 **USB bootloader**

En annen aktuell måte å programmere kontrolleren på, er med hjelp av en USB bootloader. Dette vel og merke hvis man har en kontroller med USB funksjonalitet. En bootloader er opprinnelig et lite program som kun har jobb å laste selve operativsystemet eller selve systemets firmware.

I våres tilfelle snakker vi om en bootloader hvis jobb er å laste inn firmwaren på kontrolleren. Dette blir gjort ved at man legger inn en liten programsnutt i en reservert minneplassering. Ved oppstart ser kontrolleren etter hvorvidt bruker ønsker å gå inn i bootloader minnet eller kun gå videre til regulær minneplassering. Slik som vår USB bootloader fungerer kan man ved hjelp av en bootloader koden legge inn firmware på kontrolleren, uten behov for eksterne programmeringsenheter av typen ICD2.

Fordelen med dette er flere, man kan i et design kutte ut hele tilkoblingskontakten tilhørende ICSP, i tillegg slipper man å forholde seg til en ICD2 enhet. I stedet for å programmere ved hjelp av programmeringsenheten kan man nå programmere ved hjelp av et bootloader program på PC-en. Av andre fordeler kan man trekke frem at programmeringshastigheten er mye raskere når man programmerer ved hjelp av bootloader. Det er også meget tidsbesparende under utvikling av et USB design og kun å ha USB kontakten å forholde seg til.

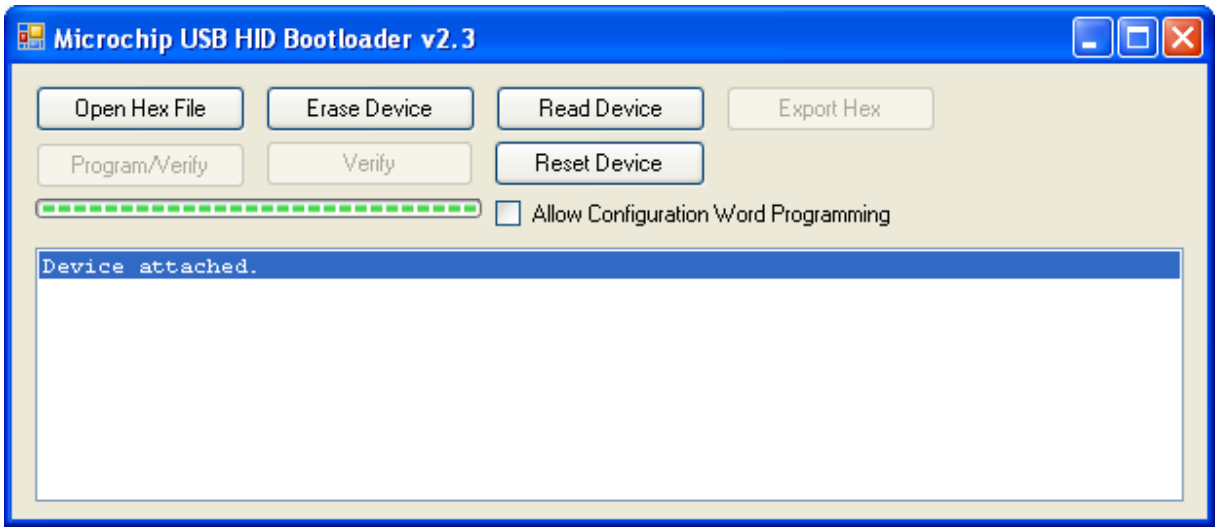
Bakdeler med en USB bootloader er også tilstedet. Siden bootloaderen ikke ligger inne på mikrokontrollerene som standard så må dette legges inn manuelt. Det finnes heldigvis flere måter å løse dette på. Man kan spesialbestille kontrollere som har bootloader lagt inn. Eller så kan man legge dette inn ved hjelp av ICSP. Problemet er da at man allikevel må programmere kontrolleren ved hjelp av eksterne kilder, dette gjør at en USB bootloader egner seg best for oppdatering av firmware.

I vårt tilfelle løste vi dette ved å lage et eget kort for programmering av kontrolleren.

Forhåpentligvis vil Microchip med tiden levere sine mikrokontrollere som har USB støtte med bootloader lagt inn.

Hovedgrunnen til vårt valg om å benytte USB bootloader på dette prosjektet var ikke kun de fordelene nevnt over, men også av hensyn til oppdragsgiver. ETC ønsket mulighet til så enkelt som mulig å kunne gjøre endringer i firmware ved en senere anledning. Den enkleste måten å gjøre dette på er utvilsomt en bootloader.

Selve bootloader softwaren og firmwaren vi benyttet oss av under utviklingen var Microchip sin egen, som er hentet fra Microchip sin USB rammeverk. Microchip har to bootloadere tilgjengelig, en som bruker "HID Class" USB driver og en som bruker "Vendor Class" USB driver. Vi valgte å bruke HID versjonen av bootloaderen. Dette består av en .hex fil man legger inn på kontrolleren, og et lite program på PC siden. Dette programmet er meget enkelt og gir mulighet for å legge inn en .hex fil på mikrokontrolleren, lese ut fra mikrokontrolleren samt andre valg slik som reset av kontrolleren. På figuren under kan man se programmet sett fra brukers perspektiv.



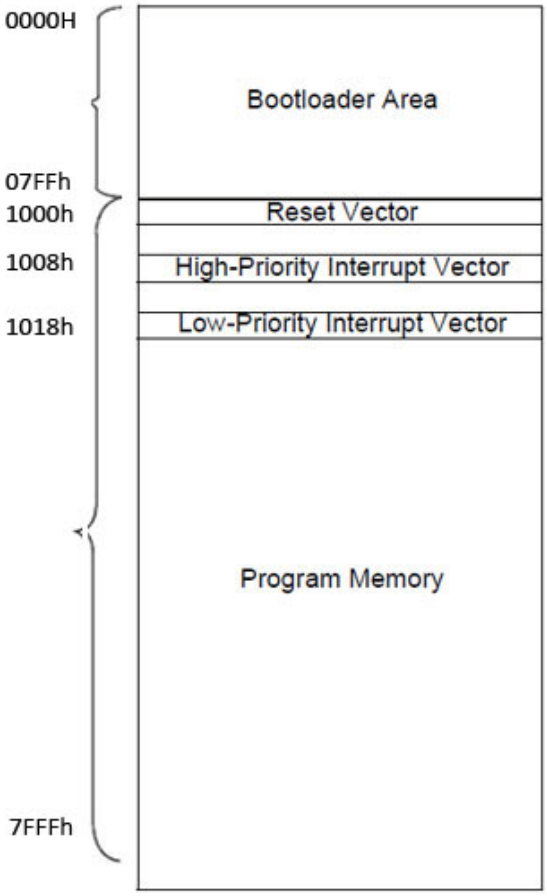
Figur 8 - Microchip HID bootloader med mikrokontroller tilkoblet

For at man skal kunne programmere ved hjelp av USB bootloaderen må man sette mikrokontrolleren i bootloader modus. Dette gjøres ved å holde nivået på en bestemt pin lavt etter reset av mikrokontrolleren.

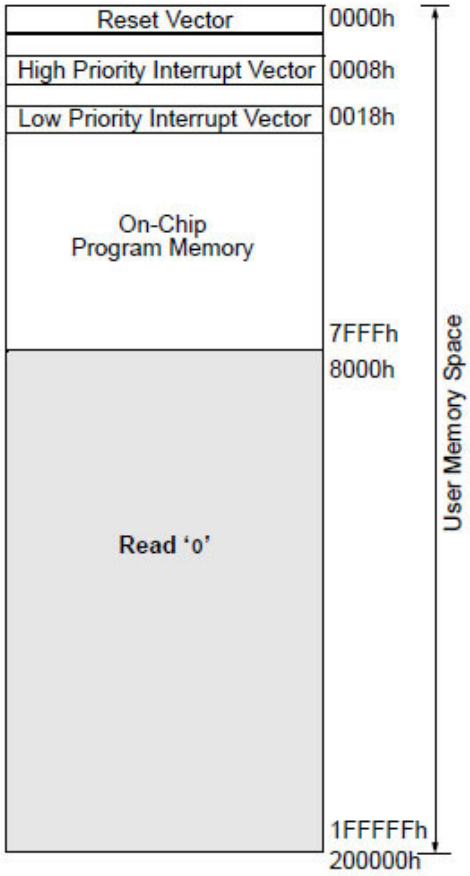
På PIC18 kontrollere har man tre minne vektorer som angir plassering på kritiske elementer i kontrolleren. Disse tre er vektorer for "Reset", "Low interrupt" og "High Interrupt". En minne vektor angir forhåndsbestemt minneplassering, så når kontrolleren detekterer en "Low interrupt" vil den da gå til plasseringen angitt av vektoren knyttet opp til "Low interrupt" å utføre de kommandoer som er der. Disse tre vektorene har plasseringen 0000h for reset vektoren, 0008h og 0018h for høy og lav interrupt. Disse vektorene er skrivebeskyttet og kan ikke endres.

Som nevnt tidligere fungerer bootloaderen ved at man plasserer bootloader koden i en reservert plassering i programminnet. Denne er 0000h til 07FFh på kontrolleren. Dette er en plassering som i utgangspunktet inneholder de nevnte vektorene for "Reset" samt interruptene. Dette kan løses i software ved hjelp av en omadressering bort fra de opprinnelige vektorene. Man setter da ikke nye adresser på vektorene, men man plasserer en omadressering ved den opprinnelige vektoren.

De nye minnelokasjonene er 1000h for reset, 1008h for høy interrupt og 1018h for lav interrupt. Under kan man se figurer av programminnet med og uten bootloader. Figurene er hentet fra (DS39632D, Microchip Datasheet, 2007) og (DS51526B, Microchip Datasheet, 2008).



Figur 9 - Programminnet med bootloader



Figur 10 – Programminnet uten bootloader

Koden for selve omadresseringen kan sees under.

```
#pragma code HIGH_INTERRUPT_VECTOR = 0x1008 // Setter ny adresse på high_interrupt vectoren
void High_ISR (void)                        // Funksjonen som styrer høy interrupt
{
_asm goto hoy_interrupt _endasm      }      // Leder videre til interrupt funksjonen med
                                            //--interrupt rutinen


#pragma code LOW_INTERRUPT_VECTOR = 0x1018  // Setter ny adresse på lav_interrupt vectoren
void Low_ISR (void)                         // Funksjonen som styrer lav interrupt
{
_asm goto lav_interrupt _endasm      }      // LEder videre til interrupt funksjonen med

                                            //--interrupt rutinen
```

7.3 *Oppsett av mikrokontrolleren*

Oppsettet av mikrokontrolleren gjøres ved å sette flere registre på kontrolleren. Vi skal her se på de registrene vi har satt for å få mikrokontrolleren til å oppføre seg slik vi ønsker.

7.3.1 **Oscillator innstillinger**

En av de mest kritiske innstilingene på alle mikrokontrollere og spesielt mikrokontrollere som skal bruke USB kommunikasjon er oscillator innstillingene, det er disse innstillingene som bestemmer hastigheten på mikrokontrolleren. Ringvirkningene av kontrollrens hastighet går igjen i hele systemet, enten det er snakk om tiden en tidtakings modul bruker, eller tiden en analog til digital konvertering tar.

I vårt tilfelle var det kritiske at hastigheten på kontrolleren tilfredsstilte de kravene stilt for å få USB kommunikasjonen til å fungere.
USB kontrollen på kontrolleren er avhengig av en frekvens på 48Mhz for å fungere. For at kontrolleren skal kunne oppnå denne hastigheten har den interne funksjoner som hjelper til med dette.

Den har mekanismer for både oppskalering og nedskalering av klokkehastigheten. Alle disse tingene må manuelt stilles innetter klokkehastigheten til det enkelte krystall. I vårt tilfelle valgte vi å benytte et krystall som ga en frekvens på 20Mhz.

For å skalere frekvensen slik vi ønsker må følgende bit settes.

- FOSC
  - FOSC bitene består av fire bit, som alle er en del av CONFIG1H registret. Disse sier kontrolleren hvilken type oscillator som er tilkoblet. I våres tilfelle ønsker vi å bruke et krystall på 20Mhz. Fra databladet til kontrolleren finner vi at denne skal konfigureres som HS oscillator. Vi vet i tilegg at vi skal bruke PLL for å få frem ønsket klokketakt med tanke på USB kommunikasjonen.
    Derfor skal FOSC bitene settet som følgende:
    ```
    #pragma config FOSC      = HSPLL_HS
    ```
- PLLDIV
  - Disse bitene sin jobb er å nedeskalere inngangsfrekvensen fra krystallet ned til 4Mhz, dette for at modulen som oppskalerer til 96Mhz skal fungere slik den skal.
    I våres tilfelle er inngangsfrekvensen 20Mhz, vi må derfor nedskalere fem ganger.
    PPLDIV består av tre bit, som er en del av CONFIG1L registret.

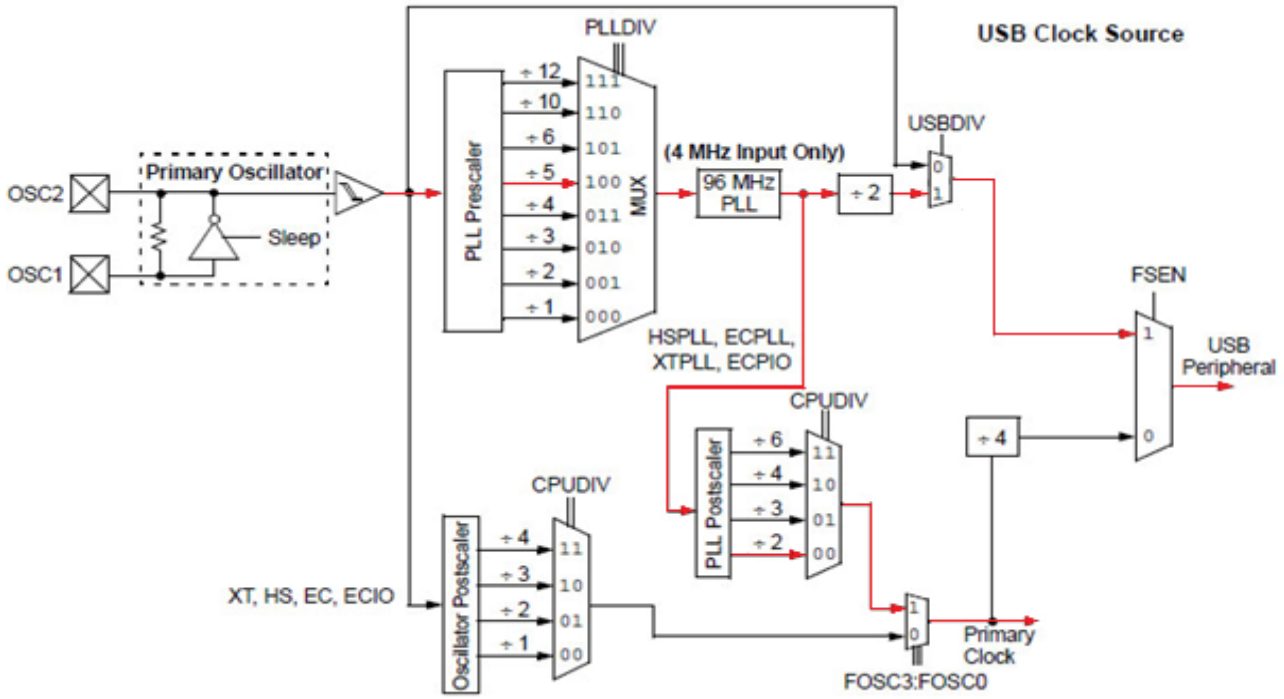
    Dette blir satt med kommandoen:

```
#pragma config PLLDIV   = 5
```

- USBDIV
  - Dette bitet bestemmer hvor USB kommunikasjonen henter sin klokketakt fra. Hvis kontrolleren er klokket av et krystall på 48Mhz, slik at man ikke trenger å modifisere kølokketakten. I våres tilfelle har vi modifisert frekvensen opp til 96Mhz. Denne ønske vi nå å dele opp i to, slik at USB kommunikasjonen får riktig frekvens. Dette gjøres ved følgende kommando :
    ```
    #pragma config USBDIV    = 2
    ```

- CPUDIV
  - Dette registret skal sette den endelige klokketakten ved hjelp av en nedskalering. CPUDIV består av to bit, som begge er i CONFIG1L registret. Innstillingene i disse bit-ene oppfører seg forskjellig ut i fra hvilken oscilator man har bestemt seg for i FOSC bitene. I våres tilfelle er dette HSPLL oscilatoren.
    vårt tilfelle er frekvensen 96MHz før nedskaleringen i CPUDIV bit-ene. Vi ønsker en endelig frekvens 48MHz på kontrolleren og velger derfor 2 ganger nedskalering, som gjøres med kommando :
    `#pragma config CPUDIV = OSC1_PLL2`

På figuren under, hentet fra (DS39632D, Microchip Datasheet, 2007) kan man se et diagram som illustrerer de forskjellige registrene knytet til oscillatoren.



Figur 11 - Registre knyttet til oscillatoren på mikrokontrolleren

7.3.2 Kritiske innstillinger på mikrokontrolleren

Det er også en del andre bit man må stille inn for å få kontrolleren til å fungere slik man ønsker.

- VREGEN
  - Dette bittet styrer den innebygde 3.3V spenningsregulatoren knyttet til USB kommunikasjonen. Denne regulatoren har kun i oppgave levere spenning til USB-modulen. Når denne er aktivert vil det ligge 3.3V på VUSB pinnen. Alternativet til å aktivere denne er å ha en ekstern spenning på 3.3V på VUSB pinnen på kontrolleren.
    Et meget viktig poeng å få frem er at VDD må til enhver tid være høyere enn VUSB, dette gjelder også når regulatoren er deaktivert.

Regulatoren kan aktiveres med følgende kommando:

```
#pragma config VREGEN    = ON
```

- MCLRE
  - Dette bit-et bestemmer om mikrokontrolleren har Master Clear aktivert, altså hoved nullstillings mulighet. Hvis dette er deakmtivert vil man få en ekstra pinne som kan brukes til andre ting. MCLRE bitet finner man i CONFIG3H registret.
    I våres tilfelle er denne aktivert med kommando:

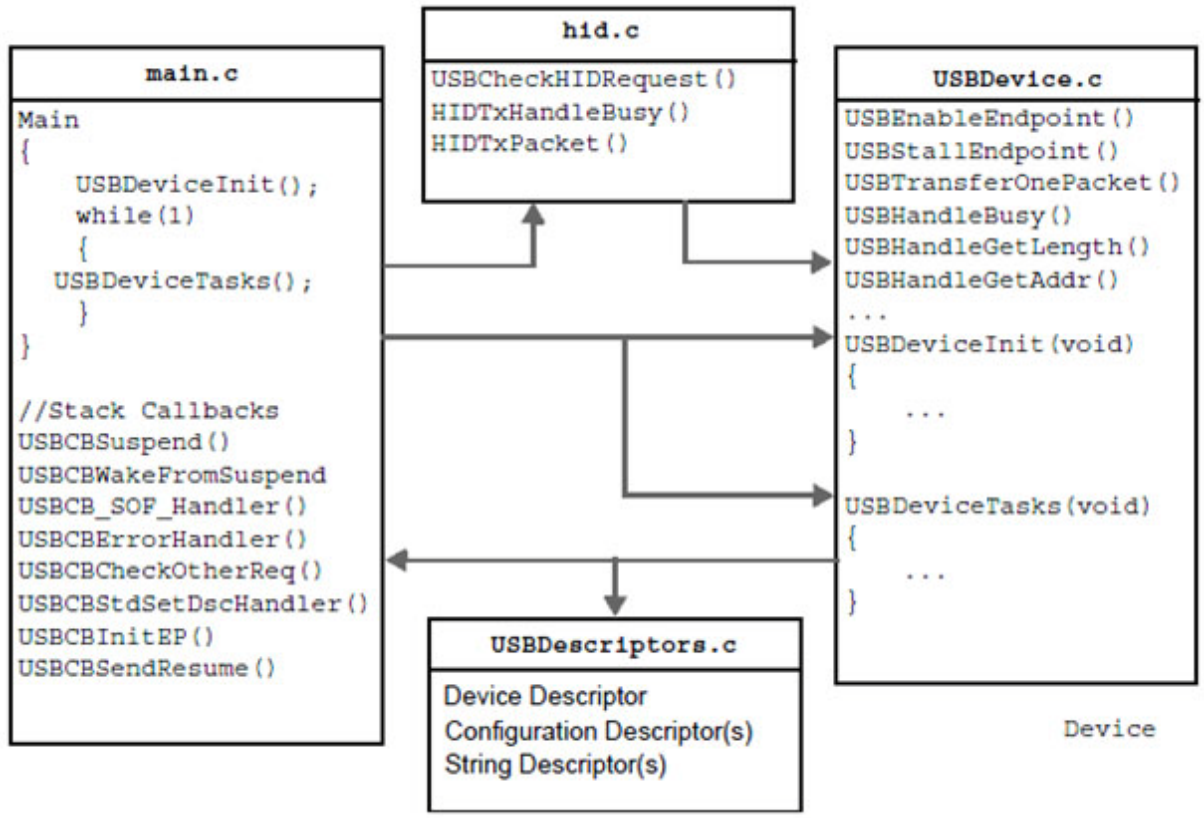
```
#pragma config MCLRE     = ON
```

- STVREN
  - Dette bitet er "Stack Full Reset Enable Bit". Det vil si at hvis hvis stacken i kontrolleren skulle bli full, så vil dette bit-et nullstille kontrolleren.
    Vi har valgt å aktivere dette ved hjelp av kommandoen:

```
#pragma config STVREN    = ON
```

7.4 *USB rammerverket*

Selve firmware koden som styrer kontrolleren består av flere deler. Den delen som har med selve kommunikasjonen med USB å gjøre stammer fra Microchip sitt USB rammeverk som vi tidligere har omtalt i mikrokontroller kapittelet. På figuren under hentet fra (DS51679B, Microchip Datasheet, 2008) kan man se flytskjema som viser hvordan den logiske strukturen er bygd opp og fungerer.



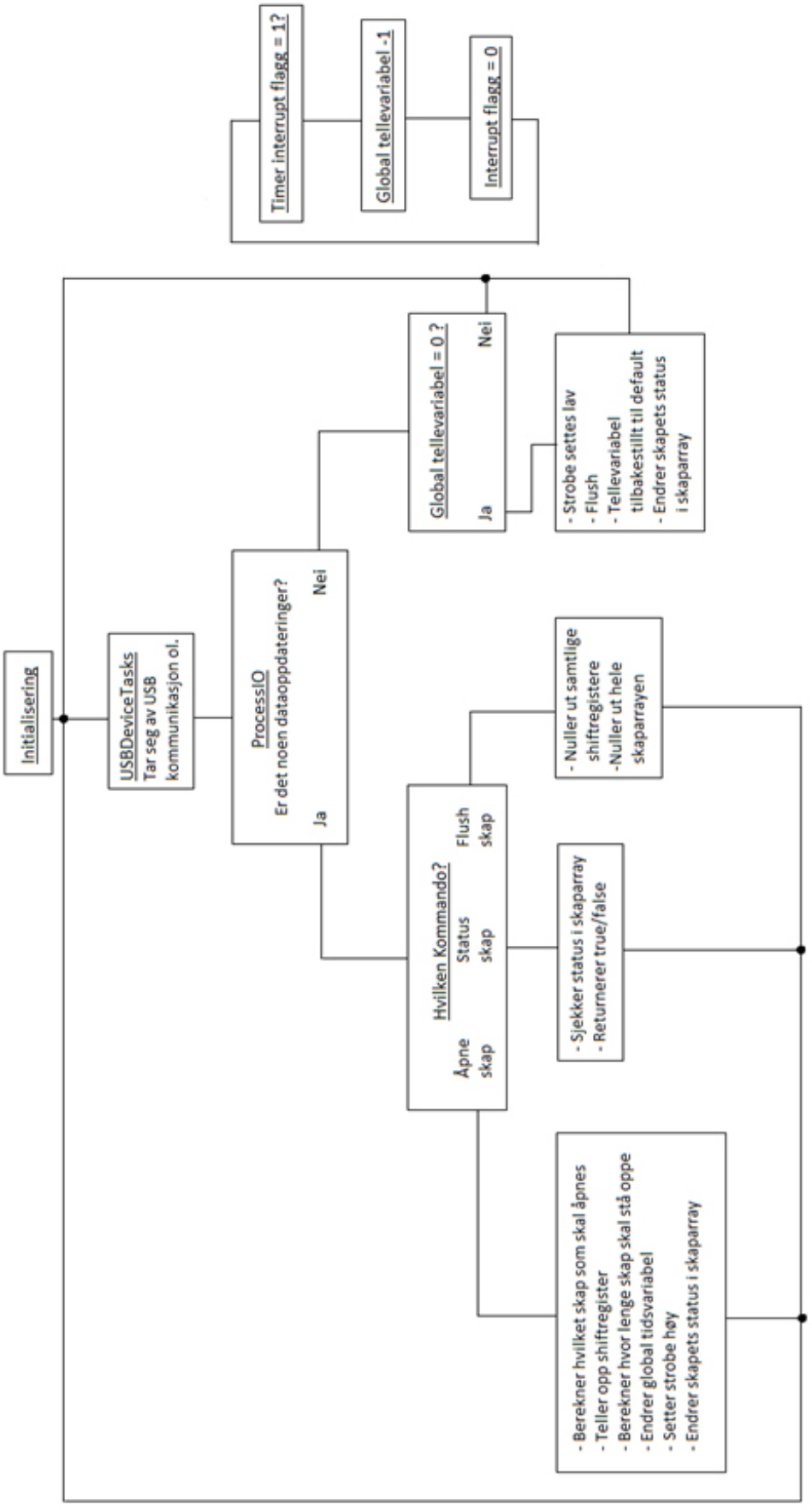
Figur 12 - Logiske strukturen av Microchip USB rammeverk

USB kommunikasjonen er delt opp i totalt fire filer. Disse er følgende:

- Main.c
  Er hovedfilen til programmet. Denne filens oppgave er å kalle på funksjonene som initialiserer og aktiverer USB kommunikasjonen. Dette er også filen som inneholder resten av programmet, inkludert styringsfunksjonene. Den inneholder også noen funksjoner som tar for seg mer avanserte deler av stacken for eks feilbehandlig, men dette er funksjoner vi ikke bruker i vårt prosjekt.
- Hid.c
  Inneholder de delene som har med oppsett mot HID driver å gjøre. Dette er dekket kapitelet om USB.
- USBDevice.c
  Inneholder funksjoner for oppsett av kommunikasjonen. Her finner man også de to hovedfunksjonene USBDeciceInit og USBDeviceTask.
- USBDecriptors.c
  Denne filen inneholder USB descriptorene til mikrokontrolleren. Dette er filene som forteller USB verten om seg selv under en enumerasjons prosess. Dette er dekket kapitelet om USB.

7.5 *USB funksjoner*

USB rammeverket inneholder en rekke funksjoner, noen som påvirker bruker direkte samt andre som kun blir brukt til oppsett av USB kommunikasjonen. Vi skal gå igjennom de funksjonene vi benytter oss direkte av her.



Figur 13 - Flytskjema firmware

7.5.1 **USBDeviceTask(void)**

Denne funksjonen er tilstandsmaskinen til USB stacken. Dette innebærer at denne ser om det er endringer på USB buss-en, i form av data sendt fra kontrolleren eller data mottatt fra USB verten.

Vi poller denne som en del av den evige løkken som holder kontrolleren sin firmware i gang.

7.5.2 **ProcessIO(void)**

Denne funksjonen tar seg av videreformidling av data som blir sendt og mottatt fra brukeren. Hvis den mottar eller sender ny data sier den fra om dette, slik at USBDeviceTask igjen kan ta seg av sendingen.

Sendingen av data foregår ved hjelp av to arrayer med data, begge med lengde på 65 bytes. Hver gang man sender eller mottar data blir disse arrayene oppdatert og hele dens innhold blir sendt.

Som et resultat av at kontrolleren kun skal oppføre seg etter et fastsatt mønster inneholder denne funksjonen en switch med alle mulige alternativer for mottatt data.

Når man i softwaren sender arrayen med data vil man på kontrolleren motta denne i variabelarrayen med navn RecivedDataBuffer. Her vil kommandoen som bestemmer hva kontrolleren skal gjøre legge seg i plassering [0]. Ut i fra denne reagerer kontrolleren ved å utføre de forhåndsbestemte opperasjonene knyttet til denne kommanoden. Da kontrolleren har gjort det den skal, vil den sende tilbake svar til styringsinterfacen. Dette svaret består alltid av to bytes med data som plasseres først i arrayen med navnet ToSendDataBuffer. Den første av bytesene er et ekko av det softwaren sendte til kontrolleren, mens byte nummer to er svaret fra kontrolleren. Dette svaret er forhåndsbestemt slik at softwaren vet hva den skal se etter i statussjekken, for så å melde fra til bruker. Under kan man se koden for mottak og sending når kontrolleren får kommandoen 0x85 som er kommandoen for å se om et skap er åpent.

```
case 0x85:                    // Case for å se om skap er åpent
help_var = skap_apent(ReceivedDataBuffer[1], ReceivedDataBuffer[2]);
// Ser om mottat skap og luke er åpen, hvis tifeller blir help_var == 1
ToSendDataBuffer[0] = 0x85;   // Ekko av kommando til vert
if(help_var == 1)
        ToSendDataBuffer[1] = 0x01;// Forteller til vert at skap er åpent
else
        ToSendDataBuffer[1] = 0x00;// Forteller at skap er lukket

if(!HIDTxHandleBusy(USBInHandle))// Funksjonen som sender og forteller hva som skal sendes
{
        USBInHandle = HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],64);
}
break;
```

7.5.3 **USBDeviceInit(void)**

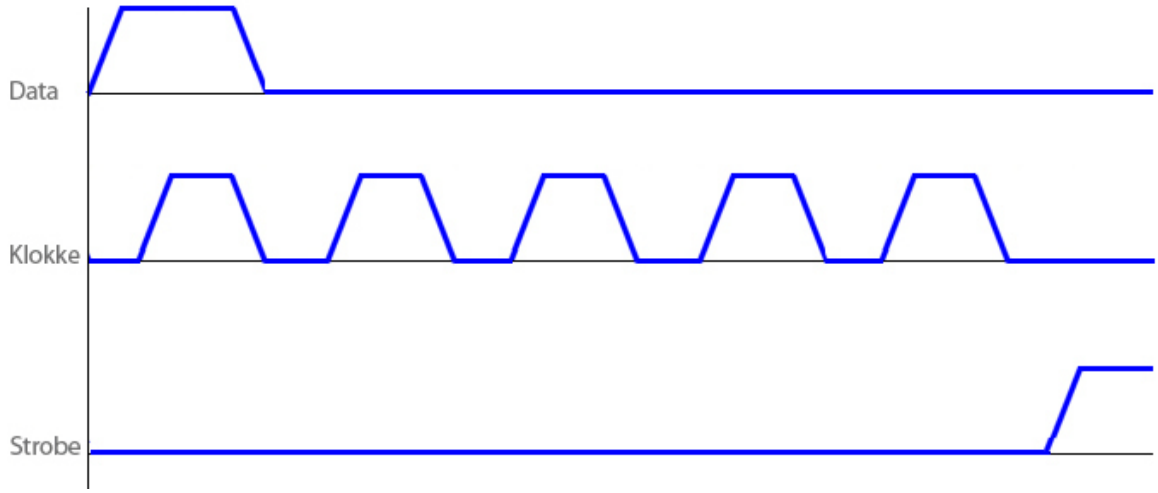
Denne funksjonen initialiserer device stacken til sin opprinnelige tilstand. Denne blir påkalt en gang etter reset.

7.6 ***Styringsfunksjoner nøkkelskap***

Skapet inneholder et skiftregister som styrer releene som åpner lukene. Selve styringen av skapet skjer ved hjelp av tre linjer. En klokkelinje, en datalinje og en strobelinje. I tilegg krever skiftregistret en linje til ”output enable”. Denne linjen skal ligge høy når registret er aktivert. Siden registeret til en hver tid skal ligge høyt når kortet er tilkoblet har vi valgt og legge den høy direkte i fra regulatoren.

Data linjen sin jobb er å sende ut en høy puls til skiftregisteret, denne pulsen blir så klokket utover i skiftregistret av klokkelinjen. Når man har klokket så mange plasseringer som ønsket, settes strobelinjen høy. Så lenge strobelinjen er høy vil luken med denne plasseringen i skiftregistret stå åpen. En viktig ting å få med seg rundt skiftregistret er at på starten av hvert skiftregister er det to plasseringer som ikke går ut til skapet. Dette fordi skapene har seks luker, mens skiftregistret har åtte utganger. Dette innebærer ved åpning av en luke at man må legge til to klokketakter for hvert skiftregister(skap) som klokkes igjennom.

På figuren under kan man se hvordan de forskjellige signalene går ved åpning av luke nummer to i det første skapet i systemet.



Figur 14 - Datasignaler ved åpning av luke 4 på skap nr1

7.6.1 Konstruksjon av signaler for styring av skiftregister

For å lage signaler for styring av skiftregistret må vi manuelt senke og heve linjer med en fast forsinkelse. For å få en fast forsinkelse har vi benyttet oss av en funksjon som er hentet fra C18 kompilatoren sitt sett med funksjoner. Ved å legge til filen ”delays.h” fra C18 biblioteket kan man benytte seg av funksjonen ” `Delay100TCYx(..)`”. Denne funksjonen venter 100 instruksjonstakter multiplisert med parameteret sendt med.

For å styre data og klokkelinjen laget vi en funksjon som styrte begge disse to. Funksjonen ”`bit_send(int)`” sender ut enten en høy eller lav puls på datalinjen, samtidig med at den genererer en puls ut på klokkelinjen. Koden for denne funksjonen kan sees under.

```
int bit_send(int hoy_lav){          // Har definert skap_tris og skap_com i top.
                                    // -- vikig at data og klokke er på samme
                                    // -- port slik at systemet skal virke.
skap_tris = 0x00;                   // Setter port defintert til datasending til utgang

if(hoy_lav == 1){                   // Hvis mottatt 1 som parameter sender det
                                    //--ut høy datapuls
skap_com = 0x04;                    // Sender ut høy data, lav klokke
Delay100TCYx(75);                   // Delayer for å skape en halv takt

skap_com = 0x06;                    // Setter klokken høy for å skape høye delen av periode
Delay100TCYx(75);                   // Delay for andre halvdel av perioden
```

```
}
Else                                    // Ikke mottatt 1 som parameter, dermed er det snakk om
                                        // -- lav periode
{
skap_com = 0x00;                        // Begge linjer ut lav
Delay100TCYx(75);                       // Delay for halve perioden

skap_com = 0x02;                        // Setter Klokke høy, data lav. For å skape høye delen av
                                        //--perioden
Delay100TCYx(75);                       // Delay for andre halvdel av perioden
}
```

Som man kan se av koden har vi definert to navn som brukes til settingen av linjer. Disse er ”skap_com” og ”skap_tris”. Vi har gjort det på denne måten for å gjøre det enkelt å bytte utgang på kontrolleren. Vi bruker på vårt endelige design PORTD på kontrolleren. Hvis vi derimot skulle ønske å endre på dette, f. eks pga lettere tilgang ved utlegg, kunne vi gjort dette enkelt ved å endre hva portene er definert som.

Funksjonen som brukes har et parameter som sendes med. Hvis tallet sendt med er ”1” vil man sende ut en høy puls. Settingen av linjene foregår ved hjelp av hex verdier. Dette fordi man med hex verdier har mulighet til å sette et helt register i en operasjon. Man slipper dermed en forsinkelse mellom settingen av data og klokke linjen.

Vi har i vår design brukt RD1 og RD2 til henholdsvis til klokke og data. Disse bitene er en del av PORTD registret. I tabellen under kan man se de forskjellige bitene.

Navn	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit2**	**Bit 1**	**Bit0**
PORTD	RD7	RD6	RD5	RD4	RD3	***RD2 - Data***	***RD1 – Klokke***	RD0
0x00	0	0	0	0	0	0	0	0
0x02	0	0	0	0	0	0	**1**	0
0x04	0	0	0	0	0	**1**	0	0
0x06	0	0	0	0	0	**1**	**1**	0

Tabell 3 - PORTD register med hex verdier

7.6.2 Åpning av luker

Funksjonen for sending av bit er grunnlaget for to andre viktige funksjoner i firmwaren, nemlig åpning og lukking av luker.

Styringsinterfacet fra software henviser til skap x og luke nr y. Da alle lukene styres av seriekoblete skiftregistre er det derfor mye greiere og kun å arbeide med luke nummer.

For å beregne hvilket lukenummer vi skulle frem til ut fra disse to parameterene trengte vi en algoritme. For dette lagde vi funksjonen ”skap_nummer(int,int)”. Denne funksjonen får in skap og lukenummer som parametere, og returnerer denne lukens nummer.

```
int skap_nummer(int skap_num, int dor_nr){
                                        // Funksjon for å finne hvem plassering i luke
```

```
                                    //--arrayen et skap og luke tilsvarer.
int help_var = 0;                   //-- Hjelpevariabel

help_var = skap_num -1 ;            // Trekker fra 1 på skapnummer, pga siste skapet i rekken
                                    //-- ikke skal telles igjennom alle lukene, men kun til
                                    //-- den luken som er sendt med
help_var *= 6;                      // Ganger opp alle "hele" skap med seks,
                                    //--altså alle skap før det skapet en luke skal åpnes i

help_var += ( skap_num -1 )* 2;     // Legger til 2 ekstra per skiftregister for å
                                    //-- fylle ut skiftregistrene
help_var += dor_nr;                 // Legger på luken på siste skapet, for å få endelig luke
                                    //-- nummer
return help_var;
}
```

Under kan man se eksempel på algoritmen ved åpning av skap 4, luke 6.

$$Helpvar = 4 - 1 => 3$$

$$Helpvar = 3 * 6 => 18$$

$$Helpvar = 18 + \big((4 - 1) * 2\big) => 24$$

$$Helpvar = 24 + 6 => 30$$

Ved hjelp av funksjonen over, samt ”bit_send” funksjonen har vi laget en rutine som automatiserer dette. Denne rutinen er en del av ”ProssessIO” funksjonen som er omtalt tidligere.

```
bit_send(1);                        // Sender først et høyt bit
help_var = skap_nummer(ReceivedDataBuffer[1], ReceivedDataBuffer[2]);
                                    //Finner hvem skap nr det er snakk om

for(j = 0; j < help_var+1; j++)     //Klokker den høye takten ut til     luken som skal åpnes

bit_send(0);
skap_nr[help_var] = 1;              //Setter skap til åpent i arryen som tar seg av dette

PORTD = 0x01;                       // Setter stoben høy
timer_delay(ReceivedDataBuffer[3]); // Setter verdi i "tid" variabel, det er da denne som
                                    //-- blir telt ned.  Når denne er ferdig blir skap døren
                                    //-- lukket.
ToSendDataBuffer[0] = 0x86;         //Ekko av kommandoen tilbake til verten
ToSendDataBuffer[1] = 0x01;         //Sier fra at man har mottatt data og apen skap
```

Når en luke har blitt åpnet av kontrolleren sender den bekreftelse på dette tilbake til software. Denne bekreftelsen er en byte med verdien ”0x01”.

7.6.3 **Status på luker**

Hvorvidt en luke er åpen eller ikke blir tatt hånd om av arrayen med navnet ”skap_nr[]”. Denne arrayen har en plassering for hver luke og når en luke blir åpnet vil dens respektive plass i arrayen bli satt til ”1”. Når USB verten spør hvorvidt en luke er åpen eller ikke vil, kontrolleren gå inn i denne arrayen å se. Den rutinen som ser etter hvorvidt en luke er åpen finner man også i ”ProsessIO” funksjonen til kontrolleren. Hvis luken er lukket vil mikrokontrolleren sende tilbake byten 0x00.

7.6.4 **Lukking av luker**

Den siste kritiske styringsfunksjonen er den tidligere omtalte ”flush” funksjonen. Denne benytter seg også av ”bit_send(int)” funksjonen. Denne funksjonen fungerer ved at man sender en klokketakt uten datatakt for hver plassering i skiftregistret man ønsker og sette blank. Ved å gjøre dette for alle plasseringene i skiftregistret etterfulgt av heving og senking av strobe linjen vil alle lukene på skapet lukke seg.

```
int flush(){                        // Funksjon som lukker skap ved å sende klokkesignaler
                                    //-- uten datasingaler
int j = 0;
TRISD = 0x00;                       // Setter portD til utgang

for(j = 0; j <100; j++)             // For løkke som bestemmer antall lave datatakter som
bit_send(0);                        // --  skal sendes. 100 er et "satt" antall, som sikkert
                                    // -- kan være mindre. men pga. hastighet på kontroller
                                    // -- har vi latt denne være.

PORTD = 0x01;                       // Setter strobe høy
Delay100TCYx(100);                  // Delay for at man skal registrere at stroben går høy
PORTD = 0x00;                       // Strobe lav


for(j=0; j<101; j++){               // Resetter array som holder styr på hvem skap som er
                                    //--åpent
skap_nr[j]=0;
};
```

Antall luker i et system kan variere sterkt. Vi har derfor valgt å sette 100 luker som standard for systemet. Dette vil si at man uansett klokker 100 plasseringer i skiftregistret. Da kontrolleren jobber veldig raskt så er ikke dette noe bruker merker. Hvis man derrimot skal ha flere luker enn 100 i systemet vil man få behov for å øke antall klokkesignaler i flush rutinen. Dette kan man gjøre ved å øke variablen som blir telt opp i for() løkken i flush rutinen.

7.7 *Timer med interrupt*

En av de funksjonene som blir benyttet mest på kontrolleren vår i tillegg til USB er muligheten for tidtaking ved hjelp av kontrollerens Timer modul. PIC18F4550 har totalt fire tidtakingsmoduler moduler.

7.7.1 **Tidtakingsmodulen Timer0**

I vårt arbeid fant vi ut at Timer0 var den modulen som best utførte den jobben vi hadde behov for. Timer0 har følgende egenskaper.

- 8 bit eller 16 bit register for tidtaking eller telling
- Nedskalering med mulighet for opptil 1:256 nedskalering
- Interrupt når tellerregistret er fult

Det vi ønsket å bruke tidtakingsfunksjonen på kontrolleren til var å holde styr på hvor lenge et skap skulle holdes åpent. De forskjellige tidtakingsmodulene på en mikrokontroller blir typisk brukt til små tids intervaller. Når man som i vårt tilfelle skal ta tiden på lengre intervaller enn et telleregister, så er det to måter å gjøre dette på.

Man kan la programmet gå inn i en loop rutine som man går i til man er ferdig med tellingen. Dette er en meget enkel måte å gjøre det på, som egner seg i de fleste helautomatiske systemer med mikrokontrollere. Dette fordi man som regel ikke er avhengig av respons fra enheten under

tidtakingsprosessen. I vår tilfelle var dette ikke tilfelle, vi fant derfor raskt ut denne metoden ikke var mulig å benytte seg av.

Den andre måten å løse tidtakingen på var ved hjelp av en tidtakingsmodul som er knyttet opp mot interrupt. Dette vil si at man teller opp antall ganger tiidtakingsmodulen har fylt registret og forårsaket interrupt i systemet. Hvis man vet hvor lang tid denne interrupten tar, kan man ut i fra dette ta tiden meget nøyaktig.

For å finne ut hvor lang tid et interrupt i systemet tar er det flere faktorer som spiller inn. Ved hjelp av formelen under kan man finne ut dette.

$$Interrupt\ tid = \frac{Instruksjoner\ pr.klokketakt*\ Register\ størrelse * Nedskalerings\ faktor}{Klokkefrekvens}$$

Formel 1 - Interrupt tid for Timer modul

Ved bruk av Timer0 med 16-bit register og en nedskalering på 1:64, får man følgende.

$$\frac{4 * 2^{16} * 64}{48Mhz} = 0.3495\ sekunder$$

Altså vil det ta 0.3495 sekunder mellom hvert interrupt signal. For å regne ut hvor mange interrupt rutiner som tilsvarer et sekund kan man bruke følgende formel.

$$Antall\ \ Interrupt\ rutiner = \frac{Sekunder}{Interrupt\ tid}$$

Formel 2 - Antall interrupt rutiner for et gitt antall sekunder

$$\frac{30\ sekunder}{0.3495\ sekunder} = 85\ interrupt\ rutiner$$

Ved et interrupt hvert 0.3495 sekund vil dette også tilsi at vi får en feilmulighet på 0.3495 sekunder, dette fordi man kun kan telle ”hele” interrupt. For å få ned denne feilmarginen kan man senke nedskalerings faktoren, og på denne måten få kortere tid mellom hvert interrupt. Man kan tenke på nedskaleringen som et verktøy for å bestemme oppløsningen på tidtakingen. I vårt tilfelle der vi skulle ta tiden på noe som ikke hadde behov for større nøyaktighet enn +/- 1 sekunder valgte vi å bruke 16-bits register med nedskalering på 1:64.

7.7.2 Oppsett av Timer0 modul med interrupt

Oppsett av Timer0 gjøres ved å stille inn de forskjellige bittene som finnes i registret med navn T0CON (Timer0 Control Register”). Dette inneholder totalt 8-bit, som gjør følgende.

- *Bit 7 : TMR0ON - 1/0 = Skrur timer på/av*
- *Bit 6 : T08Bit - 1/0 = Timer0 bruker 8/16-bit register for telling*
- *Bit 5 : T0CS - 1/0 = Velger klokkekilde. Klokke fra T0CKI pinne/Intern*

- *Bit 4 : T0SE - 1/0 = Ved ekstern klokke, telle på høy-lav/lav-høy overgang*
- *Bit 3- Bit 0 - Velger nedskaleringsfaktor. 110 – Tilsvarer 1:128*

For at timeren skal kunne styre interrupt må dette også settes i registrene INTCON og INTCON2.

Følgende bit må settes i INTCON registeret.

Bit 7 og Bit 6 i INTCON registret har forskjellige egenskaper ut i fra om bitet med navnet IPEN er høyt eller lavt. IPEN bitet finner man i RCON registret, og bestemmer hvorvidt man benytter seg av prioritet på interrupt på kontrolleren. Dette benytter vi oss ikke av.

- *Bit 7: GIE - Global Interrupt Enable Bit, aktiverer interrupt på kontrolleren.*
- *Bit 6: PEIE - Peripheral Interrupt Enable Bit, aktiverer interrupt på alle umarkerte moduler.*
- *Bit 5: TMR0IE - Timer0 Overflow Interrupt Enable Bit, aktiverer interrupt på Timer0 modulen på kontrolleren.*
- *Bit 2: TMR0IF - Timer0 Overflow Interrupt Flag Bit, bit som går høy når interrupt har forekommet.*

C18 kompilatoren vi benyttet oss av har en rekke medfølgende styringsfunksjoner for enkelt å kunne benytte seg av de innebygde modulene på kontrolleren. Dette er også tilfelle for Timer0 modulen. Vi kunne derfor benytte oss av denne fremfor å gjøre alt manuelt. Dette gjorde det mulig for oss å benytte oss av en kode som garantert fungerer, noe som igjen gjorde at vi kunne spare tid rundt dette.

For å kunne benytte seg av Microchip sine styringsfunksjoner for Timer moduler må man inkludere filen ”timers.h” som følger med Microchip sin C18 kompilator. For styringen av Timer0 modulen er det to funksjoner, en for start med navnet OpenTimer0(..) og en for stopp CloseTimer0(..).

De forskjellige funksjonene har en rekke parametere man kan kalle for å få Timer0 modulen til og oppføre seg slik man ønsker. I vårt tilfelle ønsker vi å bruke 16-bit register med en nedskalering på 1:128 samt interrupt aktivert. For å gjøre dette brukte vi følgende kode.

```
OpenTimer0(TIMER_INT_ON & T0_16BIT & T0_SOURCE_INT & T0_PS_1_64);
```

Her kaller vi OpenTimer0 funksjonen med parameterne som betyr følgende.

```
TIMER_INT_ON      – Timer0 har interrupt aktivert

T0_16BIT          - Timer0 jobber i 16-bit modus

T0_SOURCE_INT     - Velger intern klokke

TO_PS_1_64        - Velger nedskalering på 1:64
```

Videre må vi aktivere følgende bit for å få interrupt til å skje slik det skal. Dette gjøres ved å sette tidligre omtalte GIE bit.

```
INTCONbits.GIE = 1;
```

Det som skjer når et interrupt forkommer er som følgende.

- Timer0 sitt interrupt flag blir heis, med andre ord interrupt bit satt til 1.
- Programmet går til adressen gitt av interrupt vektoren til kontrolleren. Dette er omtalt mer detaljert i kapitelet som omtaler bootloaderen.
- Programmet går så til funksjonen High_ISR(void).
- Fra High_ISR() blir man igjen ledet videre til den endelige interrupt funksjonen med navnet hoy_interrupt().
- Funksjonen utfører det den skal, for så å sette Timer0 interrupt Flag bitet til 0. Man er da klar til neste interrupt.

For å sette tidtakings og interrupt rutinene i system bruker vi en global variabel. Denne holder styr på hvor mange tidsenheter som til enhver tid er gjenstående.

Slik programmet vårt fungerer blir denne telt ned med en for hver gang interrupt rutinen blir kjørt. Når denne når null vil kontrolleren først senke strobe linjen som holder lukene åpne, for så å sende ut en nullstillings rutine til skapet. Den vil nå sette den globale tidsvariablen til en standard verdi, som er på 30 sekunder. Denne standardverdien sørger for at skapet jevnelig blir nullstilt.

Det er totalt tre funksjoner som gjør dette. Interrupt funksjonen med navnet ”`hoy_interrupt`” som står for nedtellingen. Funksjonen ”`Lukk_skap` ” står for nullstilling av variabel samt nullstilling av skap og funksjonen ”`timer_delay` ” står for utregning av tidsvariabler fra sekunder.

8 Hardware

Under konstruksjonen av styringskortet var det mange kritiske faktorer å ta hensyn til. USB er en ganske følsom form for dataoverføring, små detaljer hardwaremessig kan føre til at ingen ting virker. Og når ingen ting virker er det veldig vanskelig å finne frem til årsaken. I dette kapittelet skal vi fortelle litt om vegen frem til det endelige kortutlegget, kritiske komponenter, og utfordringene vi hadde underveis.

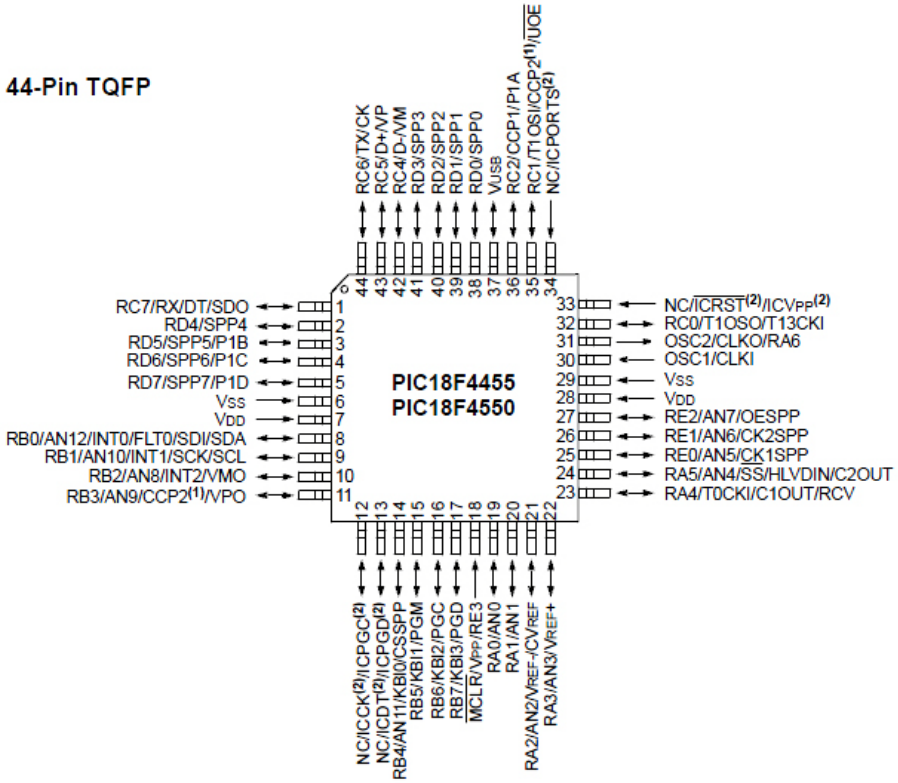
8.1 *Kritiske komponenter*

8.1.1 **Mikrokontrolleren**

Som fortalt tidligere valgte vi mikrokontrollren PIC18F4550 fra Microchip. Denne finnes i flere pakker, både hullmontert og overflate montert. Under utviklingsprosessen fant vi det gunstig å jobbe med en PDIP pakke, da denne er støttet av Microchips utviklingsmodul PICDEM 2 PLUS. Ved hjelp av PICDEM 2 PLUS modulen kunne vi programmere kontrolleren under de tidlige stadiene av testkort.

I forhold til oppkoblingen av chipen er dette den komponenten som bestemmer hvordan resten av kretsen må bli. Det som er veldig viktig å huske på i forhold til kontrollerkoblingen er koblingskondensatorer mellom VSS og VDD på begge sidene av kontrolleren, dette for å stabilisere spenningen på VDD.

På den endelige kretsen endte vi opp med en TQFP pakke på kontrolleren. Koblingskjema for denne kan sees under.

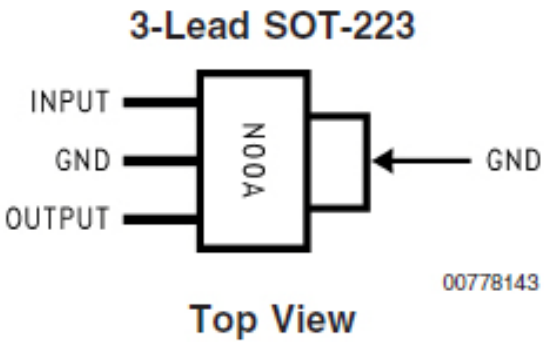


Figur 15 - Pin diagram for PIC18F4550 med TQFP pakke hentet fra (DS39632D, Microchip Datasheet, 2007).

8.1.2 **Regulator**

I valget av regulator hadde vi flere muligheter, valget sto i første omgang mellom en effektgjerrig switch-regulator eller en helt enkle lineære regulator. Ulempene med en switch-regulatore er at det er veldig begrenset hvor mye strøm de klarer å levere, i tillegg er de veldig dyre. Valget falt derfor på en lineær regulator til tross for at denne har større effekttap. Kravene som ble stilt til regulatoren var i utgangspunktet at den måtte levere 5V til drift av krets, samt elektronikken i skapet. Vi valgte til slutt en LM340MP, dette er en regulator som leverte veldig jevne test resultater i tillegg til at den er i stand til å levere opp til 1,0A noe som er mer en nok til kretsen vår og det skapet trekker.

Regulatoren er meget enkel å koble opp da den ikke har behov for noen eksterne komponenter. Den har kun 3 ben.



Figur 16 - LM340MP pin diagram hentet fra (DS007781, National Semiconductor Datasheet, 2006).

Den største bekymringen rundt regulatoren var varme. Den regulatoren vi brukte under utviklingsprosessen var en hullmontert regulator som hadde gode kjøleegenskaper. Denne sto fritt og hadde ingen komponenter som var plassert kritisk nærme. På det endelige kretskortet benyttet vi oss av en overflatemontert pakke på regulatoren. Denne pakken var av typen SOT223.

Regulatoren vi har valgt her tåler opp til 125 grader, for å se hvorvidt det gikk an å bruke denne med de spesifikasjoner vi krevde satte vi opp regnestykket. (DS007781, National Semiconductor Datasheet, 2006)

$$c = temp + (Vinn - Vut) * A * C/W$$

Formel 3- Beregning av temperatur i en lineær regulator

De forskjellige verdiene i formelen er :

c	= temperatur på regulatoren
temp	= temperatur på omgivelsen regulatoren skal operere i
Vinn	= Spenning inn på regulatoren
Vut	= Spenningen regulatoren skal levere ut
A	= Strømmen regulatoren skal levere
C/W	= Temperaturkonstant knyttet opp mot pakken på regulatoren

Fra databladet til regulatoren vet vi at C/W på regulatoren er 174. (DS007781, National Semiconductor Datasheet, 2006)

Når vi setter inn de verdiene vi har målt får vi

$c = 25 + (12 - 5) * 0.033 * 174 = 65.194$ grader

Fra databladet til mikrokontrolleren vet vi at den kan trekke opp til 50mA avhengig av oppgaver, så vi beregnet også for dette tilfellet. (DS39632D, Microchip Datasheet, 2007)

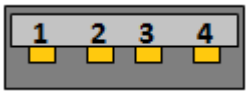
$c = 25 + (12 - 5) * 0.050 * 174 = 85$ grader

Begge tilfellene ser vi altså at er godt innenfor grensene for hva regulatoren tåler.

For å finne ut hvordan kretsene i skapet reagerte i forholdt til varmeutviklingen valgte vi å konstruere en stresstest, denne gikk ut på at vi lagde et program som åpnet et skap så fort et annet ble lukket. Dette er ikke en realistisk bruk av systemet, men et worst case senario der vi ville få en indikasjon på hvordan kretsene reagerte. Etter at stresstesten hadde gått i flere timer uten problemer besluttet vi at varmeutviklingen i fra regulatoren ikke var et problem.

8.1.3 **USB-kontakten**

En viktig komponent for å få dette til og fungere er USB-pluggen. USB har fire ledere Data-, data+, VDD, og VSS. Kontrolleren vi valgte har ferdig innebygd USB stack, så alt vi trengte å gjøre var og trekke baner fra kontakten til kontrolleren. Dette høres kanskje enkelt nok ut, men det er av like vel nok av fallgruver å gå i, og hvis kommunikajonen ikke virker kan det som sagt være problematisk og finne ut hvorfor. På det første testkortet benyttet vi oss av en USB-B kontakt, dette er kontakten vi kjenner igjen som en sekskantet inngang på baksiden av mange printere, skannere eller eksterndisker. Denne løsningen virket veldig bra helt til vi skulle testmontere kortet i skapet, pluggen var nemmelig større en åpningen på baksiden av skapet og vi måtte derfor tre enden med A-pluggen ut fra innsiden. Noe som ville være alt for tungvint med tanke på en eventuell produksjon. Vi gikk derfor over til en USB-A kontakt på kortet på den endelige kortet. Dette fordi man da kunne ta i bruk helt standard kabler til kommunikasjonen, noe som ville bli kostnadsbesparende i tillegg til at denne pluggen går inn i skapet uten problemer. Et noe uventet problem vi fikk med dette var tilgangen på kabler, IT-tjenesten hadde ikke kabel av typen A(male) - A(male), og nettbutikkene til Biltema og Clas Ohlson hadde heller ikke denne kabeltypen. Dette kan kanskje tyde på at denne kabelstandarden er på veg ut? Dette problemet er uansett minimalt da disse kablene vil bli bestilt sammen med kortene i en eventuell produksjon.



Figur 17 - USB-kontakt (Hunn)

1	VDD
2	D-
3	D+
4	VSS

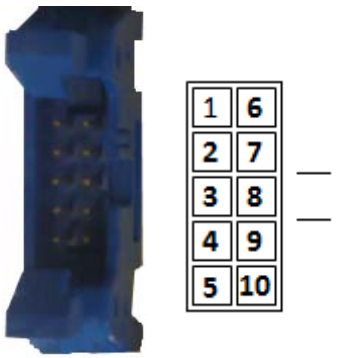
Tabell 4 - Signalhenvisning USB-kontakt

I tillegg til banene som trekkes inn til kontrolleren var det veldig viktig med en koblingskondensator på 1µF i mellom VDD og VSS, denne er veldig viktig for å stabilisere spenningen mellom verten og enheten vår.

8.1.4 **Skap-kontakten**

Skap-kontakten er en annen kritisk komponent i vårt design, det er i gjennom denne kommunikasjonen går videre opp til skapet slik at de forskjellige kommandoene blir utført. Dette skjer igjennom en 10 pins parallellkabel som også foretar kommunikasjonen videre i mellom skapene hvis flere skap er koblet i serie. Et umiddelbart problem vi fikk i forhold til denne pluggen var at vi ikke hadde noen form for datablad på hvordan den skulle kobles. I rapport fra tidligere prosjekter (Mølster & Olsen, 2008) fant vi bilder av, men ingen god henvisning på hvordan koblingen skulle være. Vi måtte derfor sette oss ned å måle på kretsen i skapet for å finne ut hvordan den fungerte før vi kunne prøve å koble til via pluggen. Dette kan i utgangspunktet virke enkelt nok, men når det er snakk om 10 pinner kan man fort bli litt forvirret å begynne å rote litt. Vi klarte til slutt og få koblet det hele riktig slik at vi fikk igjennom det vi ville til rett plass.

Et annet problem vi støtte på rundt kontakten var at den kabelen vi hadde benyttet oss av fra starten av prosjektet ikke var den samme som ble benyttet av ETC. Grunnen til dette var at kabelen vi hadde fått med skapet i utgangspunktet var feil, heldigvis kunne problemet løses ved å snu kontakten 180 grader.



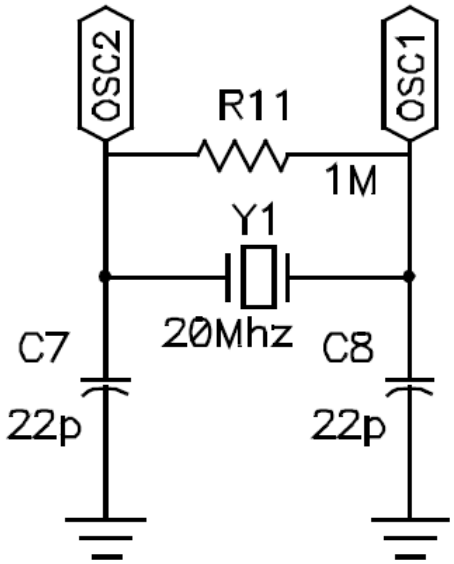
Figur 18 - Kobling av plugg til skapkommunikasjon

1	12V
2	-
3	VSS
4	VSS
5	VDD
6	12V
7	Strobe
8	Clk
9	Data
10	-

Tabell 5 – Signalhenvisning

8.1.5 Krystallet

For at kontrolleren vår skal få et fast referansepunkt i forhold til tid benytter vi oss av ett krystall, et krystall er rett og slett en stein som har kjent svingningstid når den blir påført strøm, og sender et elektrisk signal med en veldig nøyaktig frekvens. Kontrolleren vi bruker har allerede et innebygd krystall, men klokkehastigheten som USB-kommunikasjonen krever for å virke er høyere en hva dette krystallet er i stand til å levere, og vi ble derfor nødt til å benytte et eksternt krystall i stedet. Figuren under viser hvordan vi koblet opp krystallet til kontrolleren.



Figur 19 - Kobling av krystall hentet fra (DS51526B, Microchip Datasheet, 2008)

8.1.6 Bryteren

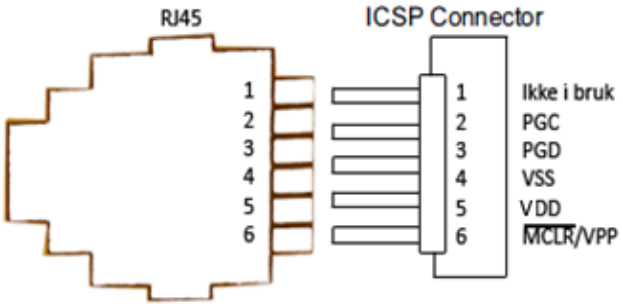
På kortet vårt er det er to brytere, Master clear og Bootloader. Begge disse knappene virker på samme måte igjennom at man legger en av inngangene på kontrolleren til jord.

Master clear er en reset, noe som gjør at kontrolleren starter opp på nytt. Mens Bootloaderen muliggjør programmering av kontrolleren igjennom USB (for mer info se kapittel som omtaler Bootloader).

8.1.7 Programmerings -kontakt

Denne kontakten brukes til å programmere mikrokontrolleren ved hjelp av Microchip sin ICD2 programmerings enhet. Denne er dekket i kapittelet som omtaler ”Firmware”. Selve pluggen er av typen RJ45, med seks pinner.

Tilkoblingen rundt programmeringspluggen er ganske enkel. Den kobles direkte til programmeringsdata og programmeringsklokke på kontrolleren samt til jord og spenning. Det eneste man må være observant på er at linjen som skal kobles til MCLR på kontrolleren må ha en motstand til VDD. Denne burde være i størrelses orden 10kohm. Grunnen til dette er at kontrolleren er operativ når denne pinnen er høy, ved resett eller når vi ønsker å programmere kontrolleren trekkes denne pinnen til jord.



Figur 20 - Pin diagram programmeringspluggen

8.1.8 Seriell port – USART

Vi hadde ved oppgavestart bestemt at dersom tiden tillot det skulle vi se på muligheter for å gjøre systemet trådløst. Da vi forsto at vi ikke rakk å fullføre en trådløs del, ble vi enig med oppdragsgiver at vi i stedet skulle legge til en mulighet for å koble seg in på kortet ved en senere anledning.

Etter en søkerunde på Elfa, fant ut at det er mulig å få kjøpt ferdige GSM modem som sender videre den informasjonen de blir tilsendt ved hjelp av USB eller RS232. Siden RS232 er en enkel seriell form for kommunikasjon valgte vi derfor å klargjøre den endelige prototypen for dette. Mikrokontrolleren vi benyttet oss av hadde mulighet for RS232 igjennom en modul som heter USART. Selve modulen er meget enkel da den kun består av fire pinner.

- RX – linje for mottak av seriell data
- TX – linje for sending av seriell data
- VDD – Spening, 5 V
- VSS – Jord

Da vi ikke vet hvordan en eventuell ekspansjon vil være, kom vi frem til at å legge ved en stiftlist på fire pinner ville være en god løsning. Med denne kan man koble seg rett inn på mikrokontrolleren, på de pinnene som trengs for dataoverføring via USART.

Denne porten er markert J6 på utlegget i vedlegg F.

Figur 21 - USART tilkobling

1	VDD, 5V
2	VSS
3	RX
4	TX

Tabell 6 - Signalhenvisning USART

8.2 *Kretskort*

8.2.1 **Software for kretskort utlegg**

Til kretskortutlegg valgt vi å benytte os av programmvare pakken fra Labcenter Electronics som heter Proteus. Denne programvare pakken inneholder programmer for både skjemategninger og kretskort utlegg. Vi benyttet oss av følgende programmer:

- ISIS Professional – Release 7.4(Build 6792)
  - Program for skjemategninger
- ARES professional – Release 7.4(Build 6792) – Level 2
  - Program for kretskort utlegg

8.2.2 **Skjemategning - ISIS**

ISIS har en rekke funksjoner som gjør at jobben med skjemategning gå greit. For det første linker det automatisk over til ARES slik at man har kontroll på alle koblinger når man driver med utleggget.

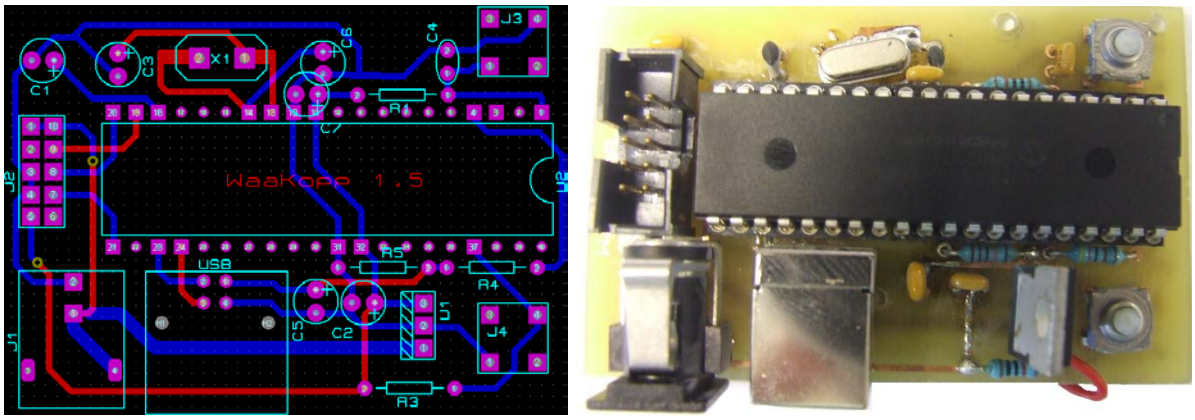
De aller fleste komponentene som vi trengte til dette prosjektet lå alle rede inne i ISIS-biblioteket, noe som gjør hele prosessen mye enklere. I denne delen av prosessen benyttet vi oss for det meste av datablader for å finne frem til hvordan de forskjellige koblingene skulle være.

8.2.3 **Kretskortutlegg – ARES**

Etter at alle koblingene var ferdig satt opp i ISIS var det på tide å linke det hele over til ARES slik at vi kunne begynne med kortutlegget. Når det kommer kortutlegg er det noen retningslinjer som det er anbefalt å følge (Hjelmstad & Juvstad, 2009).

- Til å begynne med er det en veldig god ide å legge ut de store komponentene først, på denne måten kan man tilpasse med mindre komponenter rundt slik at kortet blir kompakt.
- For å unngå støy er det en god ide å prøve å trekke banene i samme retning på samme side av kortet.
- Man bør også unngå 90 graders hjørner på banene da disse vil fungere som antenner å overføre signal til nærliggende baner, ideelt sett 45 grader.

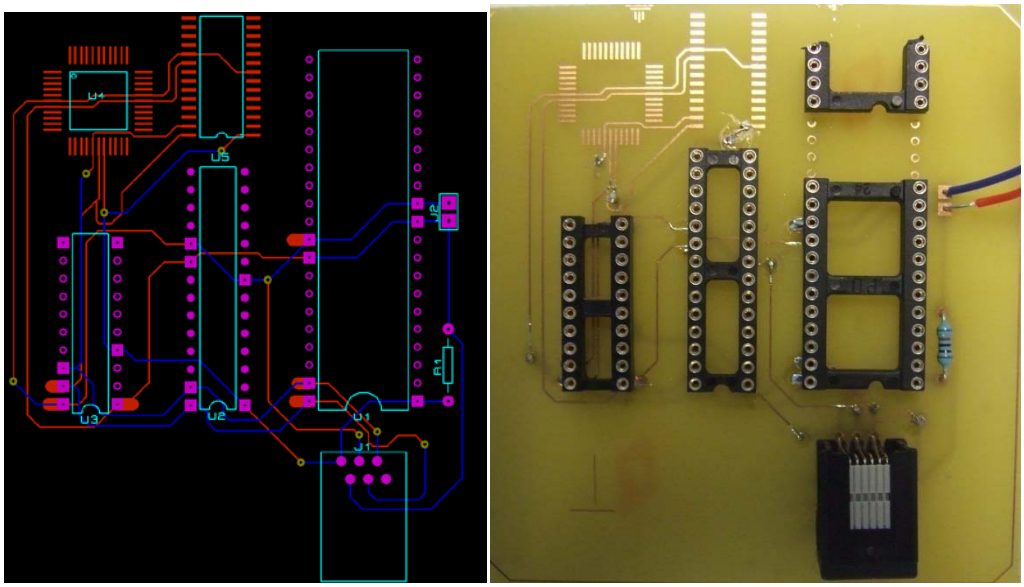
Til tross for timer med research og planlegging av kretsen er det så godt som umulig og få igjennom første kort uten feil. Siden vi hadde et meget begrenset antall nøkkelkomponenter valgte vi å lage det første testkortet socketmontert, noe som førte til at alle komponentene måtte hullmonteres til å begynne med. Dette er en tungvint løsning, men den vinner på fleksibilitet og gode muligheter for testing. En annen ting som vi måtte ta høyde for i forhold til dette kortet var plassen, kortet skulle monteres i et skap der det var noe begrenset plass til elektronikk. Det var derfor veldig viktig å planlegge plassering av komponentene, for eksempel å sørge for at USB og power-utgangene pekte i samme retning, og at det var mulig å komme til knappene på kortet når det var montert i skap. Vi begynte med å sette på de største komponentene, dette innebar chip, USB-kontakt, powerplugg, og plugg for skapkom. Vi brukte en god del tid på å finne en god plassering av disse komponentene, men fant til slutt frem til en layout som fungerte veldig fint. Kretskort utlegget ligger med som vedlegg E.



Figur 22 - Det første kortutlegget som virket slik vi ønsket.

8.3 *Programmeringskortet*

Vi forsto tidlig at dette kunne bli et prosjekt der vi kunne bli nødt til å programmere flere forskjellige kontrollere. Som et lite sideprosjekt satte vi derfor opp et lite programmeringskort der vi kunne programmere mange forskjellige chipper, deriblant den overflatemonterte chippen i det endelige kortet. Dette kortet var helt enkelt og besto kun av en RJ-45-port og en motstand, men virket utrolig bra til vårt formål. Det mangler mulighet for debugging, men siden dette er en funksjon vi ikke behøvde å benytte oss av valgte vi å utelate denne funksjonen i dette kortet. Kortet hjalp til med å gi oss god trening med tanke på in circuit programming, som var noe vi var usikre på før prosjektet startet.



Figur 23 - Utlegg programmeringskort

9 Testing

Underveis i utviklingsprosessen har det blitt testet kontinuerlig. Da vi hadde mulighet til å teste fortløpende kom vi aldri i en situasjon der det var veldig usikkert hva som gjorde at enheten ikke fungerte slik den skulle.

Vi hadde under oppgaveperioden en dag vi var på ekstern lokasjon for testing. Denne dagen dro vi på besøk til ETC, der vi fikk mulighet til å teste systemet på flere skap. Vi fikk på denne måten kvalitetssikret produktet utover de rammer som vi hadde tilgjengelig på elektro laboratoriet. Denne dagen er dekket i vedlegg B.

10 Produksjon

Det var meningen at kortet våres skulle erstatte et eksisterende krets med RS232-grensesnitt, det ville derfor bli aktuelt med en produksjon dersom vi fikk kortet og kommunikasjonen til og virke.

Da vi var ferdig med den første testversjon kom vi sammen med ETC frem til at dette testkortet fungerte så godt at vi skulle begynne å jobbe mot produksjon. Vi fikk oppgaven å sette i gang dialog med noen som kunne produsere kortet..

Vi valgte å ta kontakt med den Gjøvik baserte bedriften TOPRO. Der fikk vi kontakt med prototyp ansvarlig Tom Andre Skogsrud, som ble vår kontaktperson på TOPRO.

10.1 *Prototype*

Vi kom sammen med TOPRO og ETC frem til at vi i første omgang skulle produsere en prototype på 3-5 kort. Sammen med TOPRO kom vi også frem til de endringene vi behøvde å gjøre for at kortet kunne produseres.

Hullmonterte kort er dyre i produksjon, de må ofte håndloddes og komponenter kan i mange tilfeller være utdaterte, så vi kom fort frem til at vi måtte lage et nytt kort med overflatemonterte komponenter.

Ettersom dette ble det andre utlegget av samme kort ble prosessen denne gangen noe enklere. Det som måtte gjøres nå var å gå inn i ISIS å forandre pakkene til de forskjellige komponentene, å legge opp kortet på nytt. I forhold til overflatemonterte kort er det enda noen punkter man bør se på når det kommer til kortutlegg (Hjelmstad & Juvstad, 2009).

- Pads og baner må ikke gå for nære hverandre.
- Tykkelsen på banene bør være mindre en tykkelsen på paden.

Sammen med TOPRO kom vi også frem til følgende punkter som måtte utbedres.

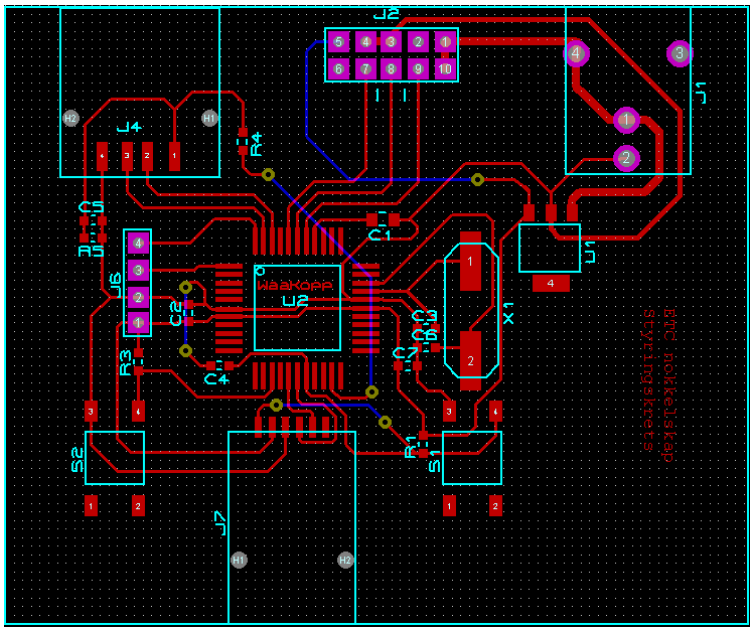
- Vi ønsket og få så mange komponenter som mulig overflatemontert, dette stilte større krav til plassering av komponenter for at banene skulle kunne trekkes så rett som mulig.
- Det ble fremmet ønske om og ha mulighet for å koble til en RJ-45 programmeringsport slik at TOPRO ikke måtte sende chippene til brenning, men at man hadde mulighet for og programmere ved hjelp av icd 2, noe vi valgte å inkludere.

10.2 *Utlegg av prototype*

Når vi startet arbeidet med utlegg av prototypen støtte vi raskt på noen problemer vi ikke hadde erfart med utlegget for testkortet. Det første var i forhold til stor mangel på overflatemonterte pakker i ARES. Dette førte til at vi ved hjelp av datablader måtte lage pakkene til de fleste komponentene selv.

Et annet problem vi støtte på var ARES sin dårlige støtte for mm mål, dette gjorde prosessen med å lage pakkene utrolig mye vanskeligere en det kunne ha hvert.

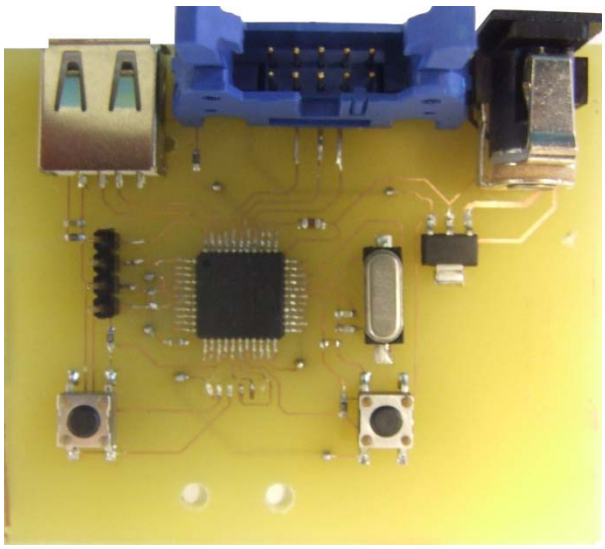
Takket være god støtte fra TOPRO samt mye arbeid fikk vi ferdig et utlegg for prototypen. Større versjon av utlegget kan sees vedlagt som vedlegg F.



Figur 24 – Kretskort utlegg prototype

10.3 *Ferdigstilt prototype*

Etter å ha fått kretskortutlegg godkjent for produksjon av TOPRO fikk vi laget 5 prototyper. Disse prototypene hadde en feil vi var nødt til å rette opp på. Det viste seg at vi hadde feiltolket databladet på knappene, noe som førte til at knappene ledet feil. Dette kunne heldigvis rettes opp relativt enkelt. Da dette var gjort fungerte prototypen akkurat slik den skulle.



Figur 25 - Ferdigstilt prototype

11 Diskusjon

11.1 *Mikrokontrolleren*

Valget av denne komponenten var det som farget mye av prosjektet. Det er ikke tvil om at i denne oppgaven var firmware koden en meget essensiell del. Hvordan det kodes varierer mye fra produsent til produsent. Like mye varierer hvor mye ekstern støtte det finnes rundt området.

Vi hadde lenge lyst til å velge en mikrokontroller fra Atmel. Grunnen til dette var rett og slett at vi da ville fått erfaring med en kontroller fra en av de andre store leverandørene av mikrokontroller, som i tilegg har en stor avdelig i Norge. Atmel har som nevnt i kapittelet om mikrokontrolleren en serie som er tilnærmet lik Microchip sin PIC serie. Det som hindret oss var to hovedgrunner. Den første var vårt førsteinntrykk av dokumentasjonen. Der Microchip har datablader som dekker absolutt alle aspekter rundt mikrokontrollerene, var det hos Atmel vanskelig å finne den informasjonen man trengte. Da dette var informasjon om de mest grunnleggende tingene begynte vi å lure på hvordan støtten ville være rundt mer snevre områder, som f. eks USB.

Den andre hovedgrunnen til at vi gikk for Microchip var rett og slett tidligere erfaring. Da begge på gruppen hadde jobbet med Microchip tidligere var dette et argument som veide tungt for oss. Som studenter ved HiG hadde vi under hele perioden også ressurspersoner som har stor kompetanse rundt Microchip sine kontrollere. Noe som vi også satte stor pris på.

Arbeidet med mikrokontrolleren gikk egentlig overraskende greit. Vi hadde som nevnt erfaring med denne kontroller-serien fra før av. Dette gjorde at vi visste hvordan ting fungerte og vi slapp dermed å bruke tid på mange av de feilene man gjerne gjør med et nytt system.

Det var knyttet stor spenning til hvorvidt USB kommunikasjonen var gjennomførbar. Heldigvis viste det seg at dette var mye greiere enn vi hadde trodd. For det første skal Microchip ha ære for det arbeid de legger i dokumentasjon. Alle aspekter av kontrolleren var dokumentert. Og all dokumentasjon var i tilegg meget god. Når Microchip i tilegg er så store som de er og PIC serien så populær som den har blitt, finnes det store mengder med ressurser på internett. Nesten alle problemene vi møtte på var det noen andre som hadde hatt før, noe som resulterte i at det ble spart mye tid rundt dette emnet.

Hvorvidt andre leverandører kan tilby like god støtte rundt USB vet vi ikke, men etter et halvt år med lesing på diskusjonsforumer og websider virker det som om Microchip er den mest brukte og foretrukne leverandøren.

11.2 *Software*

Denne delen av oppgaven var kanskje den vi hadde minst kunnskap med fra før av. Selv om gruppen hadde noen erfaringer med programmering av software, så hadde vi ikke vært i nærheten av å sette opp en kommunikasjon mot USB. Heldigvis inneholdt Microchip sitt USB rammeverk litt hjelp på dette området. Rammeverket hadde et eksempel program som gjorde mye likt det vi skulle gjøre. Dette gjorde at vi sparte mye tid ved å bruke initialiserings prosessen herfra i vår egen software.

Denne delen undervurderte vi nok omfanget av. Dette grunnet vår manglende erfaring med software programmering knyttet til USB. Til tross for at rammeverket hadde kildekode som gjorde mye av det samme, måtte det gjøres mye arbeid her. Når vi skulle redigere kildekode i et programmeringsspråk vi ikke hadde alt for mye erfaring med. Modifisere denne til en DLL fil, en filtype vi heller ikke hadde erfaring med sier det seg selv at mye kan gå galt.

I ettertid kunne vi kanskje med fordel ha latt oppdragsgiver stått for software programmeringen, da de har flere programmerere med langt høyere kompetanse enn oss. Dette ville da spart oss for mye tid som vi kunne brukt på selve produktutviklingen i stedet.

På den annen side ville vi da vert en erfaring fattigere. Det kan også tenkes at siden vi nå kjenner til både elektroingeniørens og programmererens perspektiv på en slik utvikling, at vi står bedre rustet til å takle liknende problemstillinger senere.

Når det gjelder softwaren er vi veldig godt fornøyd med at vi greide det til slutt, spesielt at vi greide å implementere DLL filen i vårt eget testprogram. Dette testprogrammet kom meget godt til nytte, da det var veldig fleksibelt. På grunn av dette sparte vi masse tid under feilsøkingsprosessen, spesielt den dagen vi var på besøk hos ETC for å teste systemet med flere skap.

11.3 *Hardware*

I forhold til denne oppgaven var de fleste premissene allerede satt, vi måtte begrense størrelsen på kortet innen skapets begrensninger. Systemet for overføring av signaler i fra kontroller til skap var allerede fastsatt, og de øvrige komponentene var allerede bestemt i fra datablader. Det eneste vi fikk friheten til var ved plassering av komponentene. Siden vi hadde et krav til størrelsen på det endelige kortet ble utfordringen å gjøre dette så kompakt som mulig.

Kortet er nok den delen vi ville endret hvis vi skulle gjort oppgaven på nytt i dag. Grunnen til dette er fordi vi først lagde et kretskort med hullmonterte komponenter, og socketer.

Vi har underveis i prosjektet erfart hvor lett ICSP fungerer, vi kunne gått rett til den overflatemonterte TQFP pakken, noe som hadde spart oss for en lang prosess med utlegg, etsing, boring og montering.

Det kan selvfølgelig diskuteres at hullmonterte kort er mer fleksible slik at man slipper å bruke opp kontrollere på kort som ikke virker. Men vi velger av like vel å påstå at dette ville vært både enklere og tidsbesparende. Hvis vi ser på det økonomiske aspektet av dette ville det nok også vært billigere, kjøpe inn komponenter til et produkt som ikke virker kan være dyrt. Men og ha to ingeniører til og utvikle et testkort som uansett må endres senere er nok enda dyrere.

11.4 *Produksjon*

Da vi valgte oppgaven visste vi at det kunne bli aktuelt med klargjøring til produksjon. Dette var en av de fordelene vi så med å gjøre en ren utviklings oppgave kontra en mer forskningsbasert oppgave.

Da vi var ferdig med første testkort og hadde demonstrert dette for oppdragsgiver fikk vi forespørsel om vi ønsket å være med på en prosess med produksjon av en liten serie med prototyper. Dette hadde vi i utgangspunktet ikke beregnet inn tid til, men da vi så på dette som en stor mulighet for å lære enda mer takket vi ja.

Forskjellen mellom å klargjøre til en fiktiv produksjon og det å forbrede en faktisk produksjon fant vi fort ut at er to helt forskjellige ting. Vi måtte nå jobbe ut i fra absolutte krav satt av de som skulle utføre produksjonen, en prosess vi fant særdeles lærerik.

Siden vi i denne oppgavedelen måtte ta hensyn til både TOPROs anbefalinger i forhold til hvilke pakker som var gunstig å bruke, og ETCs ønske om så billige kort som mulig. Ble utfordringen her å oppfylle disse betingelsene. Dette var helt på slutten av våres prosjektperiode, så vi fikk også erfaring med arbeid under kraftig tidspress.

Prototypen ble ferdig og fungerte, noe vi er veldig fornøyde med. Det er allikevel et par ting vi kunne gjort annerledes. Under prosessen med prototypen fikk vi diverse tilbakemeldinger og råd fra TOPRO som sto for produksjonen. Da TOPRO i utgangspunktet kan produsere nesten alle forskjellige typer kretskort, var det opp til oss å optimalisere kortet så mye som mulig for og få ned prisen. Her kom vi til et punkt hvor TOPRO sa seg fornøyd med kortet. Da prototypene var ferdig produsert fikk vi videre beskjed om at produksjonen hadde gått fint, men det var allikevel noen små elementer som måtte rettes opp før en større produksjon. Hvis vi skulle gjort det igjen burde vi stilt høyere krav i forhold til tilbakemeldingene fra TOPRO, på denne måten hadde vi fått løst alle de ”små” problemene først som sist. Dette ville spart ETC for en ny prosess med endringer før en større produksjon, både med tanke på kostnadene, og usikkerheten som alltid følger å gjøre endringer på et design.

12 Konklusjon

Gruppen er enigenår det kommer til konklusjonen for denne oppgaven, dette har vært en fantastisk utviklingsprosess der vi har lært utrolig mye. Vi fikk være med på en fullstendig utviklingsprosess der vi startet opp med lesing av USB-teorti og endte opp i produksjonsmøter og fremstilling av en prototyp.

Gruppen opparbeidet seg flere nyttige erfaringer i løpet av denne utviklingsprosessen.

- Nøye planlegging av utlegg lønner seg veldig
- Utvikling av Software
- Utvikling av Firmware
- Benytte teoretiske beregninger i praksis
- Research og innhenting av relevant informasjon
- Godt samarbeid er en av de sterkeste ressursene som finnes i en utviklingsprosess

Det å fremstille et produkt ut i fra konkrete spesifikasjoner gitt av industrien var veldig utfordrende, og samtidig veldig motiverende. I vår oppgave hadde ETC flere krav til produktet vi skulle utvikle for dem.

Kortet vårt skulle:

- Utføre kommando gitt fra PC-ens software
- Gi tilbakemelding om utført oppgave
- Inneholde en periodisk sikkerhetsprosedyre som lukker alle skap
- Kortet måtte passe inn i et skap med begrenset plass til ektronikk
- Mulighet for tilkobling av ekstern seriell enhet

Den endelige prototypen vi endte opp med oppfylte alle disse kravene, dette er et resultat som gruppen ser seg veldig fornød med at vi oppnådde.

Vi har i løpet av denne oppgaven opparbeidet oss en god innsikt til hvordan USB-kommunikasjon virker. I tillegg har vi fått erfaring med programmering både på kontroller og datasiden av et USB-interface. Utvikling av hardware var en del vi hadde brukbare kunnskaper om allerede før vi startet prosjektet, men også her lærte vi veldig mye om teknikk for utvikling av et kortutlegg. Vi fikk erfare hvordan samarbeidet mot industrien fungerer, og satte i sammen en endelig prototype etter TOPRO og ETCs krav ved hjelp av datablader. Dette er en veldig realistisk setting i forhold til det virkelige liv som ingeniør, og gruppen er meget takknemmelige for at vi fikk ta del i prosessen.

13 Bibliografi

Axelson, J. (2001). *USB Complete Second Edition.* Lakeview Research.

DS007781, National Semiconductor Datasheet. (2006). LM340/LM78XX Series 3-Terminal Positive Regulators.

DS39632D, Microchip Datasheet. (2007). Microchip Datasheet PIC18F 2550/4550/4455.

DS51526B, Microchip Datasheet. (2008). PICDEM FS USB Demonstration Board User`s Guide.

DS51679B, Microchip Datasheet. (2008). Microchip USB Device Firmware Framework User`s Guide.

Hjelmstad, A., & Juvstad, O. (2009). *Møterapport HAPRO.* Gjøvik: HiG.

Hyde, J. (2001). *USB Design by Example.* Intel Press.

Mølster, L., & Olsen, T. Ø. (2008). *Objekt Identifisering.* Gjøvik: HiG.

SCHS063, Texas Instrument Datasheet. (1998). Datasheet CD4094B Types.

Techguy.org, F. (u.d.). *Forum Techguy.org*. Hentet fra http://forums.techguy.org/6326442-post6.html

USB.org. (2009, 04). Hentet fra http://www.usb.org/developers/defined_class

Vedlegg A – Komponentliste

Styringskort nøkkelskap, ETC AS

Komponent referanse	Komponent	Verdi/ Type	Leverandør : Varenummer	Antall	Pris
C2,C4,C5,C7	Kondensator	0,1UF/50V 0603	Farnell	4	0.5,-
C1	Kondensator	0,47UF/50V 0805	Farnell	1	0.5,-
C3,C6	Kondensator	15PF/50V 0603	Farnell	2	0.5,-
R1,R3	Motstand	10k 0603	Farnell	2	0.5,-
R5	Motstand	100k 0603	Farnell	1	0.5,-
R4	Motstand	4K7 0603	Farnell	1	0.5,-
S1,S2	Bryter	FSM2JSMATR	Farnell : 1570392	2	5.29,-
J4	USB Type A		Elfa : 42-709-14	1	
U1	Regulator 5V, 1A	LM340MP-5,0	Farnell : 1469097	1	9.25 ,-
J2	Parallel plugg 10 pins	CWN-552-10-0021	Elfa : 43-682-05	1	34.60,-
J6	Stiftlist 4 pin	0-0826629-4	Elfa : 43-716-21	1	2.38,-
J1	Power Jack	ROKA 520 2550	Elfa : 42-047-15	1	19.60,-
X1	Krystall	20,0MHZ LF A147E	Farnell : 9713352	1	8.38,-
U2	Mikrokontroller	PIC18F4550	Farnell : 9321365	1	51,64,-
J7	RJ25	5557314-1	Farnell : 1522225	1	11,08,-
				Total pris =	153.72,-

Vedlegg B – Rapport systemtest

Rapport systemtest nøkkelskap

Andreas Waal, Magnus Kopperud

21.04.09

Innledning

Tirsdag den 21.4.09 var gruppen på besøk hos oppdragsgiver ETC for å teste den utviklede styringskretsen. Tidligere hadde gruppen kun fått testet styringskretsen på et skap, noe som hadde gått så bra at de nå kunne teste på et større system. I anledning denne dagen sto ETC til disposisjon med syv skap som gruppen fikk mulighet til å kople opp å teste på.

Oppkobling

Skapene som gruppen hadde til disposisjon var i utgangspunktet identiske. Av disse syv var det to som var litt annerledes enn de fem resterende. Et av skapene var det test skapet som gruppen hadde i utgangspunktet. Dette skapet var av litt mindre fysisk størrelse, men innehold den samme kretsen for den fysiske åpningen av lukene. Det andre skapet som skilte seg fra mengden, var et skap benyttet av ETC for demonstrasjon av systemet sitt. Dette hadde en annen krets for styring tilkoplet, men denne kunne gruppen lett kople fra for å kople på sitt eget system. De fem resterende av skapene var nye skap som kom rett fra produsenten.

Selve oppkoblingen av skapene går ut på at skap nr 1 i rekken får styringskretsen montert, mens de resterende seks blir koblet i serie etter dette. I alle skapene er det totalt seks luker, hvor hver av disse har et reel som står for åpning av lukene. Disse reelene blir igjen styrt av et skift register i hvert av skapene, dette skift registret fungerer slik at for hvert ”skift” blir neste luke åpnet. Disse registrene som nevnt over seriekoblet, som vil så at når man overflooder et register, altså skifter forbi de seks lukene som skapet har, så flytter man seg til neste skap.

Det viktige å notere seg rundt skiftregistrene er at på starten av hvert register er det to tomme plasser, noe som vil si at ved skifte av skap må man gå to ”skift” før man kommer til den delen av skiftregistre som styrer lukene.

Oppkoblingen av skapet førte til flere problemer. For å luke ut feil på en så enkel måte som mulig, så startet gruppen med å kople opp et skap om gangen. Allerede her støtte man på problemer. Fra før av hadde gruppen brukt et test skap, som alt hadde fungert meget godt på. Når de nå koblet opp skapet, valgte man og starte testingen på et av de nye skapene.

Når det første skapet ble koblet opp og strømmen ble slått på, så trakk kretsen med en gang mot sitt maksimale strømtrekk. Dette var et symptom som man hadde sett før, og ved tildigre anledninger kom dette av en feil ved tilkobling mellom styringskresten og skiftregistret. Etter litt feilsøking fant man et meget kritisk punkt for resten av prosjektet.

Gruppen hadde fra før av designet sin styringskrets mot en parallell kabel som de hadde fått utlevert av ETC. Denne kabelen var en parallell kabel av normal type, mens de kablene som ble benyttet i skapene var av typen parallell vridd. Det viste seg at kabelen som gruppen fikk utdelt av ETC ved prosjektstart var en kabel som gruppen som året før hadde hatt oppgave for ETC hadde laget. Denne kabelen var da av feil type, noe verken ETC eller vi fant ut av før denne dagen. Dette problemet fikk vi inntil videre løst ved å benytte oss av den ”gale” kabelen.

Når vi hadde fått systemet opp på et skap begynte vi å kople opp de resterende skapene. Dette gikk meget greit og rutinen vår på µkontrolleren som hadde ansvaret for å tømme alle skift registrene ved tilkobling av strøm fungerte meget bra. Bildet av skapene koblet opp kan ses under, merket *figur 1.*

Figur 1 – Skapene oppkoblet

Testing av systemet

Etter at alt var koblet opp kunne vi starte selve testingen av skapet. De punktene vi skulle teste var følgende :

1. Får vi åpnet alle skap?
2. Får vi åpnet riktig luke i alle skap?
3. Får vi tilbakemelding?
4. Virker flush rutinen vår slik vi ønsker?

Allerede ved punkt en støtte man på problemer. Når vi prøvde å åpne skap etter nummer en så åpnet ikke luken på skapet seg. Kontrollen ga ut riktig signaler samt ga tilbakemelding til programvaren om at luken var åpnet. Dette problemet gjorde at gruppen måtte nok en gang gå over systemet å se etter feil. Siden skap en i rekken fungerte slik det skulle, så var problemer med

oppkoblingen utelukket. Det som da sto igjen var firmware relaterte problemer. Da de elementene som innegår i styring av skapet er begrenset til å sende pulser til skiftregistret så gjorde dette prosessen med feilsøking relativt enkel. Det første vi tenkte var at vi rett og slett hadde for høy hastighet på sending av pulser. Vi tok da og minsket frekvensen på pulsene vi sendte, noe som ordnet problemet for oss.

Det neste vi skulle teste var hvorvidt den matematiske algoritmen som regnet ut hvor mange pulser som skulle sender fungerte slik vi ønsket. Etter en kjapp test av skapet kunne vi konkludere med at dette virket på de tre første skapene, men fra og med skap fire og utover bommet man på en luke. Hvis man ønsket åpne f. eks luke fire på skap fem, så endte man opp med luke tre i stedet. Dette var en feil som for oss ga veldig lite mening, siden i utgangspunktet så skulle alle skapene være identiske. Så hvorfor dette var som det var, ga veldig lite mening for oss. Måten vi angrep feilsøkingen av dette problemet var å åpne en og en luke for å se hvor ”steg” i skiftregistret ble borte. Det viste seg at på skap nr fire i rekken så var det kun et ”skift” på starten av skiftregistret og ikke to som det skulle være. Dette skapet var det skapet brukt av ETC til demonstrasjoner av systemet sitt. Så vi konkluderte ut i fra dette med at det måtte være noe galt med dette skapet og ikke systemet. Vi fjernet da dette skapet fra systemet. Når dette var fjernet og alle skap i systemet virket slik som de skulle, så gjorde også styringskretsen jobben slik den skulle.

Når det gjaldt tilbakemeldinger fra kontrolleren så fungerte dette utmerket under alle tester utført. Noe vi så oss meget fornøyd med.

Flush rutinen viste det seg også å være en god del problemer. Her også fungerte denne kun på det første skapet i rekken. Dette gjorde at man ikke fikk lukket andre luker enn de på det første skapet. Da dette problemet var noe lignende det vi alt hadde opplevd med åpning av skapet så fant vi raskt en løsning på dette. Ved å redusere frekvensen på flush rutinen så virket alt slik det skulle.

Konklusjon

Denne dagen med testing på større system betydde meget mye for prosjektet vårt. For det første fikk vi mulighet til å teste ut at systemet faktisk fungerte på et større system, samt vi fikk rettet en god del kritiske feil i systemet. Dette var ikke feil av stor karakter, men allikevel av så stor betydning at systemet ikke hadde fungert uten dette.

Vedlegg C – Firmware Kode

```
/********************************************************************
 FileName:      main.c
 Processor:           PIC18 or PIC24 USB Microcontrollers
 Beskrivelse:  Denne koden er en modifisering av Microchip sin main.c hentet
               fra Microchip USB Framework 2.3.Den er modifisert til å styre kommunikasjon
               mellom en PIC18F4550 mikrokontroller og ETC sitt nøkkelskap.
               Alle filer som blir inkludert finnes på CD medfølgende rapport
 Complier:     Microchip C18

/** INCLUDES *******************************************************/
#include "delays.h"                                 //Delays for bit send funksjon
#include "timers.h"                                 //Timer funksjoner
#include "GenericTypeDefs.h"
#include "Compiler.h"
#include "usb_config.h"
#include "./USB/usb_device.h"
#include "./USB/usb.h"

#include "HardwareProfile.h"

#include "./USB/usb_function_hid.h"

#define skap_tris TRISD
#define skap_com PORTD

        #pragma config PLLDIV   = 5         // (20 MHz crystal on PICDEM FS USB board)
        #pragma config CPUDIV   = OSC1_PLL2
        #pragma config USBDIV   = 2         // Clock source from 96MHz PLL/2
        #pragma config FOSC     = HSPLL_HS
        #pragma config FCMEN    = OFF
        #pragma config IESO     = OFF
        #pragma config PWRT     = OFF
        #pragma config BOR      = ON
        #pragma config BORV     = 3
        #pragma config VREGEN   = ON      //USB Voltage Regulator
        #pragma config WDT      = OFF
        #pragma config WDTPS    = 32768
        #pragma config MCLRE    = ON
        #pragma config LPT1OSC  = OFF
        #pragma config PBADEN   = OFF
//      #pragma config CCP2MX   = ON
        #pragma config STVREN   = ON
        #pragma config LVP      = OFF
//      #pragma config ICPRT    = OFF       // Dedicated In-Circuit Debug/Programming
        #pragma config XINST    = OFF       // Extended Instruction Set
        #pragma config CP0      = OFF
        #pragma config CP1      = OFF
//      #pragma config CP2      = OFF
//      #pragma config CP3      = OFF
        #pragma config CPB      = OFF
//      #pragma config CPD      = OFF
        #pragma config WRT0     = OFF
        #pragma config WRT1     = OFF
//      #pragma config WRT2     = OFF
//      #pragma config WRT3     = OFF
        #pragma config WRTB     = OFF       // Boot Block Write Protection
        #pragma config WRTC     = OFF
//      #pragma config WRTD     = OFF
        #pragma config EBTR0    = OFF
        #pragma config EBTR1    = OFF
//      #pragma config EBTR2    = OFF
//      #pragma config EBTR3    = OFF
        #pragma config EBTRB    = OFF


/** VARIABLES ******************************************************/
#pragma udata
BYTE old_sw2,old_sw3;
BOOL emulate_mode;
BYTE movement_length;
BYTE vector = 0;

int ant_skap;
```

```
int ant_luker;
int tid;
int default_tid = 20;
unsigned char skap_nr[100];

#pragma udata USB_VARIABLES=0x500

unsigned char ReceivedDataBuffer[64];
unsigned char ToSendDataBuffer[64];
#pragma udata

USB_HANDLE USBOutHandle = 0;
USB_HANDLE USBInHandle = 0;

static void InitializeSystem(void);
void ProcessIO(void);
void UserInit(void);
void hoy_interrupt();
void lav_interrupt();

void timer_delay(float);
void skap_init();
void Lukk_skap(void);
int skap_apent(int skap, int dor);
int skap_nummer(int skap_num, int dor_nr);


/*************** VECTOR REMAPPING ***********************************************/
#if defined(__18CXX)

        #define REMAPPED_RESET_VECTOR_ADDRESS                       0x1000

        extern void _startup (void);        // See c018i.c in your C18 compiler dir
        #pragma code REMAPPED_RESET_VECTOR = REMAPPED_RESET_VECTOR_ADDRESS
        void _reset (void)
        {           _asm goto _startup _endasm       }


//*************HER KOMMER DEFINERINGEN AV INTERRUPT VECTORENE - PGA BOOTLOADER
        #pragma code HIGH_INTERRUPT_VECTOR = 0x1008
        void High_ISR (void)
        {
         _asm goto hoy_interrupt _endasm     }


        #pragma code LOW_INTERRUPT_VECTOR = 0x1018
        void Low_ISR (void)
        {
        _asm goto lav_interrupt _endasm      }


//**************DE REEELTE INTERRUPT RUTINENE KOMMER UNDER HER**************
        #pragma code
        #pragma interrupt hoy_interrupt
        void hoy_interrupt()
        {
                tid--;
                INTCONbits.TMR0IF = 0;
        }

        #pragma interruptlow lav_interrupt
        void lav_interrupt()
        {}

int main(void){

InitializeSystem();
skap_init();                    // Funksjon initsialiserer skap
        while(1)
    {

         USBDeviceTasks();     // Funksjon som blir pullet for å se om det er endringer på USB
                ProcessIO();          // Funksjon som tar seg databehandlingen inn og ut USB
                Lukk_skap();          // Funksjon som puller på tid, hvis tid = 0, så flushes
det og skap array lukkes
```

```
                                                    // -- tiden denne puller på, er enten satt
skap åpen tid eller fast rutine hvert minutt
    }//end while
}//end main


/********************************************************************
 * Function:        static void InitializeSystem(void)

 * Overview:        InitializeSystem is a centralize initialization
 *                  routine. All required USB initialization routines
 *                  are called from here.
 *
 *                  User application initialization routine should
 *                  also be called from here.
 *
 * Note:            None
 *******************************************************************/
static void InitializeSystem(void)
{
    #if (defined(__18CXX) & !defined(PIC18F87J50_PIM))
        ADCON1 |= 0x0F;                 // Default all pins to digital

    #endif


//      The USB specifications require that USB peripheral devices must never source
//      current onto the Vbus pin.  Additionally, USB peripherals should not source
//      current on D+ or D- when the host/hub is not actively powering the Vbus line.
//      When designing a self powered (as opposed to bus powered) USB peripheral
//      device, the firmware should make sure not to turn on the USB module and D+
//      or D- pull up resistor unless Vbus is actively powered.  Therefore, the
//      firmware needs some means to detect when Vbus is being powered by the host.
//      A 5V tolerant I/O pin can be connected to Vbus (through a resistor), and
//      can be used to detect when Vbus is high (host actively powering), or low
//      (host is shut down or otherwise not supplying power).  The USB firmware
//      can then periodically poll this I/O pin to know when it is okay to turn on
//      the USB module/D+/D- pull up resistor.  When designing a purely bus powered
//      peripheral device, it is not possible to source current on D+ or D- when the
//      host is not actively providing power on Vbus. Therefore, implementing this
//      bus sense feature is optional.  This firmware can be made to use this bus
//      sense feature by making sure "USE_USB_BUS_SENSE_IO" has been defined in the
//      HardwareProfile.h file.
    #if defined(USE_USB_BUS_SENSE_IO)
    tris_usb_bus_sense = INPUT_PIN; // See HardwareProfile.h
    #endif

//      If the host PC sends a GetStatus (device) request, the firmware must respond
//      and let the host know if the USB peripheral device is currently bus powered
//      or self powered.  See chapter 9 in the official USB specifications for details
//      regarding this request.  If the peripheral device is capable of being both
//      self and bus powered, it should not return a hard coded value for this request.
//      Instead, firmware should check if it is currently self or bus powered, and
//      respond accordingly.  If the hardware has been configured like demonstrated
//      on the PICDEM FS USB Demo Board, an I/O pin can be polled to determine the
//      currently selected power source.  On the PICDEM FS USB Demo Board, "RA2"
//      is used for    this purpose.  If using this feature, make sure
"USE_SELF_POWER_SENSE_IO"
//      has been defined in HardwareProfile.h, and that an appropriate I/O pin has been mapped
//      to it in HardwareProfile.h.
    #if defined(USE_SELF_POWER_SENSE_IO)
    tris_self_power = INPUT_PIN;     // See HardwareProfile.h
    #endif

    USBDeviceInit();  //usb_device.c.  Initializes USB module SFRs and firmware
                                    //variables to known states.
    UserInit();

}//end InitializeSystem

void UserInit(void)          // FUnksjon i forhold til demobrett leds
{
   //initialize the variable holding the handle for the last
    // transmission
    USBOutHandle = 0;
    USBInHandle = 0;
```

```
}

int bit_send(int hoy_lav){                  // Har definert skap_tris og skap_com i top. Vikig
at data og klokke er på samme
                                                        // -- port slik at systemet
skal virke.
skap_tris = 0x00;

        if(hoy_lav == 1){
                skap_com = 0x04;
                Delay100TCYx(75);

                skap_com = 0x06;
                Delay100TCYx(75);
        }
        else                    {
                skap_com = 0x00;
                Delay100TCYx(75);

                skap_com = 0x02;
                Delay100TCYx(75);
        }


return 0;
}

int skap_nummer(int skap_num, int dor_nr){
        int help_var = 0;

        help_var = skap_num -1 ;

        help_var *= 6;

        help_var += ( skap_num -1 )* 2;
        help_var += dor_nr;                             //   trekker fra 1, for å gange med 6
return help_var;
}


int skap_apent(int skap, int dor){
        int help_var;

        help_var = skap_nummer(skap,dor);

                if(skap_nr[help_var]==1)
                        return 1;
                else
                        return 0;

}

void Lukk_skap(void){
        int i;

        if(tid==0){
                flush();
                tid=default_tid;


        }//end if
}



void timer_delay(float enheter){

        TRISD = 0x00;
        tid = enheter * 2.84;
}

int flush(){
        int j = 0;

                        TRISD = 0x00;
```

```
                for(j = 0; j <100; j++)
                        bit_send(0);

                        PORTD = 0x01;
                        Delay100TCYx(100);
                        PORTD = 0x00;


                        for(j=0; j<99; j++){
                                skap_nr[j]=0;
                                };

return 0;
}

void skap_init(){
        unsigned i = 0;
        tid = default_tid;

        for(i=0; i<99; i++)
                skap_nr[i]=0;

        OpenTimer0(TIMER_INT_ON & T0_16BIT & T0_SOURCE_INT & T0_PS_1_64);
        INTCONbits.GIE = 1;

        flush();


}
/********************************************************************
 * Function:        void ProcessIO(void)


 * Overview:        This function is a place holder for other user
 *                  routines. It is a mixture of both USB and
 *                  non-USB tasks.
 ********************************************************************/
void ProcessIO(void)
{
    int help_var = 0;
        unsigned open = 0;
        unsigned i = 1;
        unsigned j = 0;

    // User Application USB tasks
    if((USBDeviceState < CONFIGURED_STATE)||(USBSuspendControl==1)) return;

    if(!HIDRxHandleBusy(USBOutHandle))                          //Check if data was received
from the host.
    {


        switch(ReceivedDataBuffer[0])                       //Look at the data the host sent,
to see what kind of application specific command it sent.
         {


                        case 0x84:
                                flush();

                                ToSendDataBuffer[0] = 0x84;                         //Echo back
to the host PC the command we are fulfilling in the first byte.  In this case, the Get
Pushbutton State command.

                                ToSendDataBuffer[1] = 0x01;                         //Sier fra at
man har mottatt data

                                if(!HIDTxHandleBusy(USBInHandle))
                {
                   USBInHandle = HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],64);
                        }
                                break;


                        case 0x85:                          // Case for å se om skap er åpent
```

```
                                help_var = skap_apent(ReceivedDataBuffer[1],
ReceivedDataBuffer[2]);

                        ToSendDataBuffer[0] = 0x85;                         //Echo back to the
host PC the command we are fulfilling in the first byte.  In this case, the Get Pushbutton
State command.

                        if(help_var == 1)
                                        ToSendDataBuffer[1] = 0x01;
        //Sier fra at man har mottatt data
                        else
                                        ToSendDataBuffer[1] = 0x00;

                                if(!HIDTxHandleBusy(USBInHandle))
                {
                   USBInHandle = HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],64);
                    }
                                break;


                case 0x86:                              // Case for åpning av skap


                        for(i=0; i<99; i++){
                                if(skap_nr[i] == 1 ){
                                        open = 1;

                                        }       // end if
                        }       // end for

                        if(open == 0) {
                                        TRISD = 0x00;
                                        PORTD = 0x88;

                                        bit_send(1);
                                        PORTD = 0x88;

                                        help_var = skap_nummer(ReceivedDataBuffer[1],
ReceivedDataBuffer[2]);       //Finner hvem skap nr det er snakk om

                                        for(j = 0; j < help_var+1; j++)

                                        bit_send(0);

                                        skap_nr[help_var] = 1;
        //Setter skap til åpent

                                        PORTD = 0x01;

                                        timer_delay(ReceivedDataBuffer[3]);         //Setter
verdi i "tid" variabel, det er da denne som blir telt ned.

                                                //-- når denne er ferdig blir skap døren lukket.
                                        ToSendDataBuffer[0] = 0x86;
        //Echo back to the host PC the command we are fulfilling in the first byte.  In this
case, the Get Pushbutton State command.
                                        ToSendDataBuffer[1] = 0x01;
        //Sier fra at man har mottatt data og apen skap

                                } // end if open
                                else{
                                        ToSendDataBuffer[0] = 0x86;
        //Echo back to the host PC the command we are fulfilling in the first byte.  In this
case, the Get Pushbutton State command.
                                        ToSendDataBuffer[1] = 0x02;
        //Sier fra at man har mottatt data og skap ikke kan apnes

                                        }
                                                if(!HIDTxHandleBusy(USBInHandle))
                                         {
                                                        USBInHandle =
HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],64);
                                          }

                                        break;
```

```
            }
        //Re-arm the OUT endpoint for the next packet
        USBOutHandle = HIDRxPacket(HID_EP,(BYTE*)&ReceivedDataBuffer,64);
    }


}//end ProcessIO


//******************************************************************************\\
//***** HER FØLGER DIVERSE USB HANDEL FUNKSJONER; SE MICROCHIP DOKEMENTASJON ***\\
//******************************************************************************\\

void USBCBSuspend(void)
{
    #if defined(__C30__)
    #if 0
        U1EIR = 0xFFFF;
        U1IR = 0xFFFF;
        U1OTGIR = 0xFFFF;
        IFS5bits.USB1IF = 0;
        IEC5bits.USB1IE = 1;
        U1OTGIEbits.ACTVIE = 1;
        U1OTGIRbits.ACTVIF = 1;
        TRISA &= 0xFF3F;
        LATAbits.LATA6 = 1;
        Sleep();
        LATAbits.LATA6 = 0;
    #endif
    #endif
}


#if 0
void __attribute__ ((interrupt)) _USB1Interrupt(void)
{
    #if !defined(self_powered)
        if(U1OTGIRbits.ACTVIF)
        {
            IEC5bits.USB1IE = 0;
            U1OTGIEbits.ACTVIE = 0;
            IFS5bits.USB1IF = 0;

            //USBClearInterruptFlag(USBActivityIFReg,USBActivityIFBitNum);
            USBClearInterruptFlag(USBIdleIFReg,USBIdleIFBitNum);
            //USBSuspendControl = 0;
        }
    #endif
}
#endif


void USBCBWakeFromSuspend(void)             // FUNKSJON FOR USB WAKEUP; SE DOCEMENTASJON FOR
INSTRUKSER
{}

void USBCB_SOF_Handler(void)         // FUNKSJON FOR USB SOF_HANDLER; SE DOCEMENTASJON FOR
INSTRUKSER
{}

void USBCBErrorHandler(void)         // FUNKSJON FOR USB ERROR HANDLER; SE DOCEMENTASJON FOR
INSTRUKSER
{    }


void USBCBCheckOtherReq(void)
{
    USBCheckHIDRequest();
}

void USBCBStdSetDscHandler(void)
{}


void USBCBInitEP(void)
```

```
{
    //enable the HID endpoint

USBEnableEndpoint(HID_EP,USB_IN_ENABLED|USB_OUT_ENABLED|USB_HANDSHAKE_ENABLED|USB_DISALLOW_SET
UP);
    //Re-arm the OUT endpoint for the next packet
    USBOutHandle = HIDRxPacket(HID_EP,(BYTE*)&ReceivedDataBuffer,64);
}


void USBCBSendResume(void)
{
    static WORD delay_count;

    USBResumeControl = 1;                // Start RESUME signaling

    delay_count = 1800U;                // Set RESUME line for 1-13 ms
    do
    {
        delay_count--;
    }while(delay_count);
    USBResumeControl = 0;
}


#endif
```

Vedlegg D – Software kode

```
/*****************************************************************
 FileName:      dllmain.cpp
 Beskrivelse:  Denne koden genererer en DLL med de funksjoner som blir brukt
               til å styre ETC sitt nøkkelskap. Oppsett for å lage DLL finnes i rapporten.
               Prossessen med oppsett av USB er hentet fra Microchip sitt
               USB Framework 2.3, og programmet "GenericHIDsimpleDemo".
               Alle filer som blir inkludert finnes på CD medfølgende rapport.
 Complier:     Visual C++
*****************************************************************************/
#define DLL_EXPORT
#include "header_dll.h"
#include <iostream>

#pragma once

#include <Windows.h>  //Definitions for various common and not so common types like DWORD,
PCHAR, HANDLE, etc.
#include <setupapi.h> //From Platform SDK. Definitions needed for the SetupDixxx() functions,
which we use to
#using <mscorlib.dll>                        //find our plug and play device.
#define MY_DEVICE_ID  "Vid_04d8&Pid_003F"



       using namespace System;
       using namespace System::Runtime::InteropServices;

       #ifdef UNICODE
       #define Seeifdef       Unicode
       #else
       #define Seeifdef      Ansi
       #endif

       //Returns a HDEVINFO type for a device information set (USB HID devices in
       //our case).  We will need the HDEVINFO as in input parameter for calling many of
       //the other SetupDixxx() functions.
       [DllImport("setupapi.dll" , CharSet = CharSet::Seeifdef,
EntryPoint="SetupDiGetClassDevs")]
       extern "C" HDEVINFO  SetupDiGetClassDevsUM(
              LPGUID  ClassGuid,                                    //Input: Supply the class
GUID here.
              PCTSTR  Enumerator,                                   //Input: Use NULL here, not
important for our purposes
              HWND  hwndParent,                                     //Input: Use NULL here, not
important for our purposes
              DWORD  Flags);                                        //Input: Flags describing
what kind of filtering to use.

       //Gives us "PSP_DEVICE_INTERFACE_DATA" which contains the Interface specific GUID
(different
       //from class GUID).  We need the interface GUID to get the device path.
       [DllImport("setupapi.dll" , CharSet = CharSet::Seeifdef,
EntryPoint="SetupDiEnumDeviceInterfaces")]
       extern "C" WINSETUPAPI BOOL WINAPI  SetupDiEnumDeviceInterfacesUM(
              HDEVINFO  DeviceInfoSet,                      //Input: Give it the HDEVINFO we
got from SetupDiGetClassDevs()
              PSP_DEVINFO_DATA  DeviceInfoData,    //Input (optional)
              LPGUID  InterfaceClassGuid,                   //Input
              DWORD  MemberIndex,                                   //Input: "Index" of the
device you are interested in getting the path for.
              PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData);//Output: This function fills
in an "SP_DEVICE_INTERFACE_DATA" structure.

       //SetupDiDestroyDeviceInfoList() frees up memory by destroying a DeviceInfoList
       [DllImport("setupapi.dll" , CharSet = CharSet::Seeifdef,
EntryPoint="SetupDiDestroyDeviceInfoList")]
       extern "C" WINSETUPAPI BOOL WINAPI  SetupDiDestroyDeviceInfoListUM(

              HDEVINFO  DeviceInfoSet);                     //Input: Give it a handle to a
device info list to deallocate from RAM.
```

```
	//SetupDiEnumDeviceInfo() fills in an "SP_DEVINFO_DATA" structure, which we need for
SetupDiGetDeviceRegistryProperty()
	[DllImport("setupapi.dll" , CharSet = CharSet::Seeifdef,
EntryPoint="SetupDiEnumDeviceInfo")]
	extern "C" WINSETUPAPI BOOL WINAPI  SetupDiEnumDeviceInfoUM(
		HDEVINFO  DeviceInfoSet,
		DWORD  MemberIndex,
		PSP_DEVINFO_DATA  DeviceInfoData);

	//SetupDiGetDeviceRegistryProperty() gives us the hardware ID, which we use to check to
see if it has matching VID/PID
	[DllImport("setupapi.dll" , CharSet = CharSet::Seeifdef,
EntryPoint="SetupDiGetDeviceRegistryProperty")]
	extern "C"     WINSETUPAPI BOOL WINAPI  SetupDiGetDeviceRegistryPropertyUM(
		HDEVINFO  DeviceInfoSet,
		PSP_DEVINFO_DATA  DeviceInfoData,
		DWORD  Property,
		PDWORD  PropertyRegDataType,
		PBYTE  PropertyBuffer,
		DWORD  PropertyBufferSize,
		PDWORD  RequiredSize);

	//SetupDiGetDeviceInterfaceDetail() gives us a device path, which is needed before
CreateFile() can be used.
	[DllImport("setupapi.dll" , CharSet = CharSet::Seeifdef,
EntryPoint="SetupDiGetDeviceInterfaceDetail")]
	extern "C" BOOL SetupDiGetDeviceInterfaceDetailUM(
		HDEVINFO DeviceInfoSet,
			//Input: Wants HDEVINFO which can be obtained from SetupDiGetClassDevs()
		PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
	//Input: Pointer to an structure which defines the device interface.
		PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData,	//Output:
Pointer to a strucutre, which will contain the device path.
		DWORD DeviceInterfaceDetailDataSize,
	//Input: Number of bytes to retrieve.
		PDWORD RequiredSize,
		//Output (optional): Te number of bytes needed to hold the entire struct
		PSP_DEVINFO_DATA DeviceInfoData);
	//Output


//  Variables that need to have wide scope.
	HANDLE WriteHandle = INVALID_HANDLE_VALUE;  //Need to get a write "handle" to our
device before we can write to it.
	HANDLE ReadHandle = INVALID_HANDLE_VALUE;   //Need to get a read "handle" to our
device before we can read from it.

EXPORT void init(){

/*
		Before we can "connect" our application to our USB embedded device, we must
first find the device.
		A USB bus can have many devices simultaneously connected, so somehow we have to
find our device, and only
		our device.  This is done with the Vendor ID (VID) and Product ID (PID).  Each
USB product line should have
		a unique combination of VID and PID.

		Microsoft has created a number of functions which are useful for finding plug
and play devices.  Documentation
		for each function used can be found in the MSDN library.  We will be using the
following functions:

		SetupDiGetClassDevs()				//provided by setupapi.dll,
which comes with Windows
		SetupDiEnumDeviceInterfaces()		//provided by setupapi.dll, which
comes with Windows
		GetLastError()					//provided by
kernel32.dll, which comes with Windows
		SetupDiDestroyDeviceInfoList()		//provided by setupapi.dll,
which comes with Windows
		SetupDiGetDeviceInterfaceDetail()	//provided by setupapi.dll, which
comes with Windows
		SetupDiGetDeviceRegistryProperty()	//provided by setupapi.dll, which
comes with Windows
```

```
		malloc()                                                        //part of C
runtime library, msvcrt.dll?
		CreateFile()                                                    //provided by
kernel32.dll, which comes with Windows

		We will also be using the following unusual data types and structures.
Documentation can also be found in
		the MSDN library:

		PSP_DEVICE_INTERFACE_DATA
		PSP_DEVICE_INTERFACE_DETAIL_DATA
		SP_DEVINFO_DATA
		HDEVINFO
		HANDLE
		GUID

		The ultimate objective of the following code is to call CreateFile(), which
opens a communications
		pipe to a specific device (such as a HID class USB device endpoint).
CreateFile() returns a "handle"
		which is needed later when calling ReadFile() or WriteFile().  These functions
are used to actually
		send and receive application related data to/from the USB peripheral device.

		However, in order to call CreateFile(), we first need to get the device path
for the USB device
		with the correct VID and PID.  Getting the device path is a multi-step round
about process, which
		requires calling several of the SetupDixxx() functions provided by
setupapi.dll.
		*/


		//Globally Unique Identifier (GUID) for HID class devices.  Windows uses GUIDs
to identify things.
		GUID InterfaceClassGuid = {0x4d1e55b2, 0xf16f, 0x11cf, 0x88, 0xcb, 0x00, 0x11,
0x11, 0x00, 0x00, 0x30};

		HDEVINFO DeviceInfoTable = INVALID_HANDLE_VALUE;
		PSP_DEVICE_INTERFACE_DATA InterfaceDataStructure = new
SP_DEVICE_INTERFACE_DATA;
		PSP_DEVICE_INTERFACE_DETAIL_DATA DetailedInterfaceDataStructure = new
SP_DEVICE_INTERFACE_DETAIL_DATA;
		SP_DEVINFO_DATA DevInfoData;

		DWORD InterfaceIndex = 0;
		DWORD StatusLastError = 0;
		DWORD dwRegType;
		DWORD dwRegSize;
		DWORD StructureSize = 0;
		PBYTE PropertyValueBuffer;
		bool MatchFound = false;
		DWORD ErrorStatus;

		String^ DeviceIDToFind = MY_DEVICE_ID;

		//First populate a list of plugged in devices (by specifying "DIGCF_PRESENT"),
which are of the specified class GUID.
		DeviceInfoTable = SetupDiGetClassDevsUM(&InterfaceClassGuid, NULL, NULL,
DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

		//Now look through the list we just populated.  We are trying to see if any of
them match our device.
		while(true)
		{
			InterfaceDataStructure->cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);
			if(SetupDiEnumDeviceInterfacesUM(DeviceInfoTable, NULL,
&InterfaceClassGuid, InterfaceIndex, InterfaceDataStructure))
			{
				ErrorStatus = GetLastError();
				if(ERROR_NO_MORE_ITEMS == ErrorStatus)	//Did we reach the
end of the list of matching devices in the DeviceInfoTable?
				{	//Cound not find the device.  Must not have been
attached.
					SetupDiDestroyDeviceInfoListUM(DeviceInfoTable);
	//Clean up the old structure we no longer need.
					return;
```

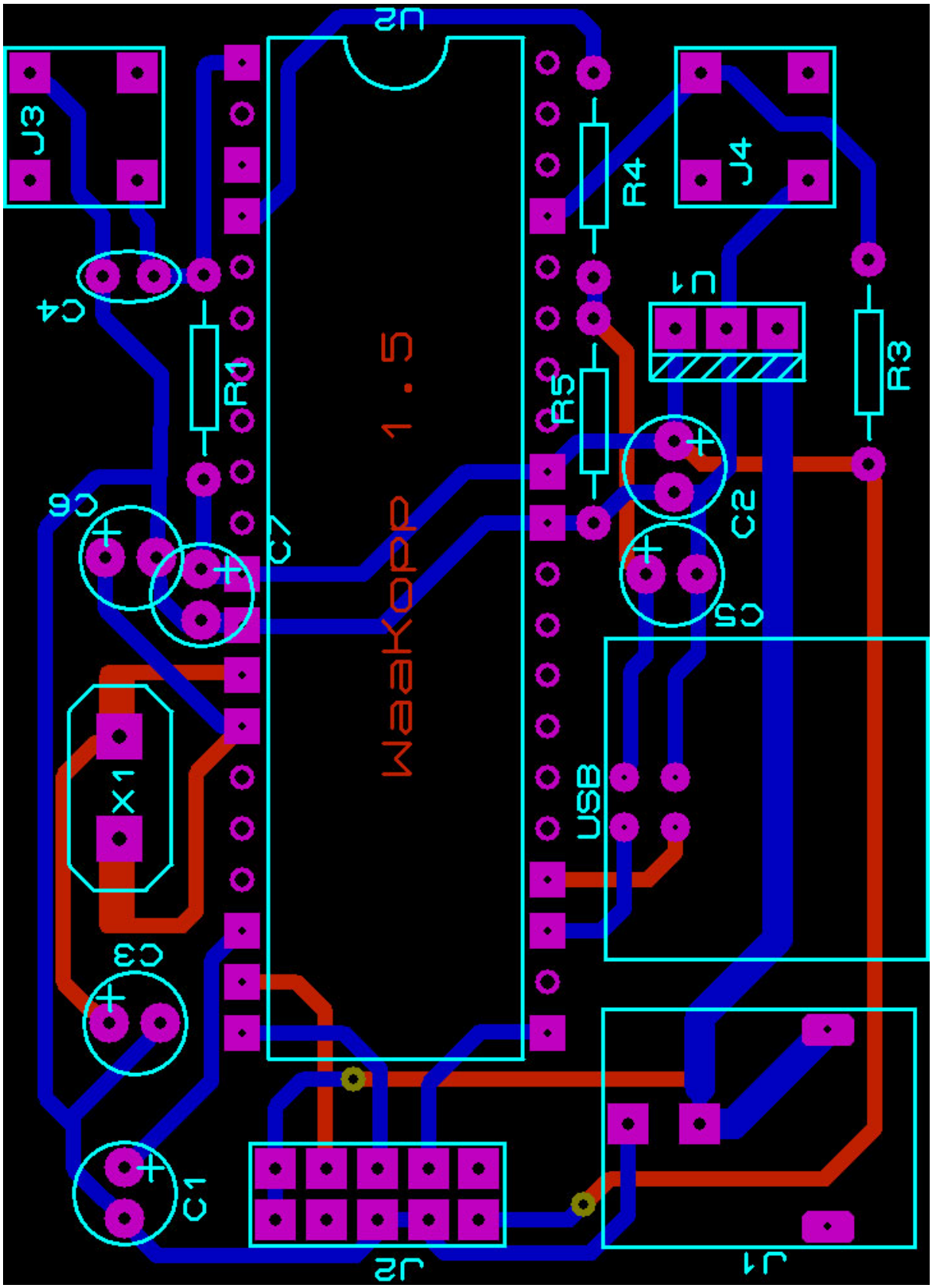
```
                        }
                }
                else    //Else some other kind of unknown error ocurred...
                {
                        ErrorStatus = GetLastError();
                        SetupDiDestroyDeviceInfoListUM(DeviceInfoTable);    //Clean up
the old structure we no longer need.
                        return;
                }

                //Now retrieve the hardware ID from the registry.  The hardware ID
contains the VID and PID, which we will then
                //check to see if it is the correct device or not.

                //Initialize an appropriate SP_DEVINFO_DATA structure.  We need this
structure for SetupDiGetDeviceRegistryProperty().
                DevInfoData.cbSize = sizeof(SP_DEVINFO_DATA);
                SetupDiEnumDeviceInfoUM(DeviceInfoTable, InterfaceIndex, &DevInfoData);

                //First query for the size of the hardware ID, so we can know how big a
buffer to allocate for the data.
                SetupDiGetDeviceRegistryPropertyUM(DeviceInfoTable, &DevInfoData,
SPDRP_HARDWAREID, &dwRegType, NULL, 0, &dwRegSize);

                //Allocate a buffer for the hardware ID.
                PropertyValueBuffer = (BYTE *) malloc (dwRegSize);
                if(PropertyValueBuffer == NULL)      //if null, error, couldn't allocate
enough memory
                {       //Can't really recover from this situation, just exit instead.
                        SetupDiDestroyDeviceInfoListUM(DeviceInfoTable);    //Clean up
the old structure we no longer need.
                        return;
                }

                //Retrieve the hardware IDs for the current device we are looking at.
PropertyValueBuffer gets filled with a
                //REG_MULTI_SZ (array of null terminated strings).  To find a device, we
only care about the very first string in the
                //buffer, which will be the "device ID".  The device ID is a string
which contains the VID and PID, in the example
                //format "Vid_04d8&Pid_003f".
                SetupDiGetDeviceRegistryPropertyUM(DeviceInfoTable, &DevInfoData,
SPDRP_HARDWAREID, &dwRegType, PropertyValueBuffer, dwRegSize, NULL);

                //Now check if the first string in the hardware ID matches the device ID
of my USB device.
                #ifdef UNICODE
                String^ DeviceIDFromRegistry = gcnew String((wchar_t
*)PropertyValueBuffer);
                #else
                String^ DeviceIDFromRegistry = gcnew String((char
*)PropertyValueBuffer);
                #endif

                free(PropertyValueBuffer);          //No longer need the
PropertyValueBuffer, free the memory to prevent potential memory leaks

                //Convert both strings to lower case.  This makes the code more
robust/portable accross OS Versions
                DeviceIDFromRegistry = DeviceIDFromRegistry->ToLowerInvariant();
                DeviceIDToFind = DeviceIDToFind->ToLowerInvariant();

                //Now check if the hardware ID we are looking at contains the correct
VID/PID
                MatchFound = DeviceIDFromRegistry->Contains(DeviceIDToFind);

                if(MatchFound == true)
                {
                        //Device must have been found.  Open read and write handles.  In
order to do this, we will need the actual device path first.
                        //We can get the path by calling
SetupDiGetDeviceInterfaceDetail(), however, we have to call this function twice:  The first
                        //time to get the size of the required structure/buffer to hold
the detailed interface data, then a second time to actually
                        //get the structure (after we have allocated enough memory for
the structure.)
```

```
                                DetailedInterfaceDataStructure->cbSize =
sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
                                //First call populates "StructureSize" with the correct value
                                SetupDiGetDeviceInterfaceDetailUM(DeviceInfoTable,
InterfaceDataStructure, NULL, NULL, &StructureSize, NULL);
                                DetailedInterfaceDataStructure =
(PSP_DEVICE_INTERFACE_DETAIL_DATA)(malloc(StructureSize));        //Allocate enough memory
                                if(DetailedInterfaceDataStructure == NULL)  //if null, error,
couldn't allocate enough memory
                                {       //Can't really recover from this situation, just exit
instead.
                                        SetupDiDestroyDeviceInfoListUM(DeviceInfoTable);
        //Clean up the old structure we no longer need.
                                        return;
                                }
                                DetailedInterfaceDataStructure->cbSize =
sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
                                 //Now call SetupDiGetDeviceInterfaceDetail() a second time to
receive the goods.
                                SetupDiGetDeviceInterfaceDetailUM(DeviceInfoTable,
InterfaceDataStructure, DetailedInterfaceDataStructure, StructureSize, NULL, NULL);

                                //We now have the proper device path, and we can finally open
read and write handles to the device.
                                //We store the handles in the global variables "WriteHandle" and
"ReadHandle", which we will use later to actually communicate.
                                WriteHandle = CreateFile((DetailedInterfaceDataStructure-
>DevicePath), GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, 0);

                                ErrorStatus = GetLastError();

                                ReadHandle = CreateFile((DetailedInterfaceDataStructure-
>DevicePath), GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, 0);
                                ErrorStatus = GetLastError();
                                if(ErrorStatus == ERROR_SUCCESS)
                                {


                                }
                                SetupDiDestroyDeviceInfoListUM(DeviceInfoTable);    //Clean up
the old structure we no longer need.
                                return;
                        }

                        InterfaceIndex++;
                        //Keep looping until we either find a device with matching VID and PID,
or until we run out of items.
                }//end of while(true)
        }
```

```
EXPORT void foo () {

        std::cout << "This is the DLL!" << std::endl;
}

EXPORT int sum (int j, int i ){

        std::cout << j << "+" << i << "=" << i+j ;

        return i+j;


}

EXPORT bool openLocker(int locker, int door,int duration){

        DWORD BytesWritten = 0;
                DWORD BytesRead = 0;
                unsigned char OutputPacketBuffer[65];      //Allocate a memory buffer equal to
our endpoint size + 1
```

```
                unsigned char InputPacketBuffer[65];        //Allocate a memory buffer equal to
our endpoint size + 1

                OutputPacketBuffer[0] = 0;                  // Alltid 0 først fordi denne ikke
blir sendt
                OutputPacketBuffer[1] = 0x86;               // Command for pulstog
                OutputPacketBuffer[2] = locker;
                OutputPacketBuffer[3] = door;
                OutputPacketBuffer[4] = duration;
                WriteFile(WriteHandle, &OutputPacketBuffer, 65, &BytesWritten, 0);
        //Blocking function, unless an "overlapped" structure is used
                ReadFile(ReadHandle, &InputPacketBuffer, 65, &BytesRead, 0);

                if(InputPacketBuffer[2] == 0x01)
                                return true;
                else //InputPacketBuffer[2] must have been == 0x00 instead
                                return false;


}


EXPORT bool flushLockers(){

        DWORD BytesWritten = 0;
                DWORD BytesRead = 0;
                unsigned char OutputPacketBuffer[65];       //Allocate a memory buffer equal to
our endpoint size + 1
                unsigned char InputPacketBuffer[65];        //Allocate a memory buffer equal to
our endpoint size + 1

                OutputPacketBuffer[0] = 0;                  // Alltid 0 først fordi denne ikke
blir sendt
                OutputPacketBuffer[1] = 0x84;               // Flush

                WriteFile(WriteHandle, &OutputPacketBuffer, 65, &BytesWritten, 0);
        //Blocking function, unless an "overlapped" structure is used
                ReadFile(ReadHandle, &InputPacketBuffer, 65, &BytesRead, 0);

                if(InputPacketBuffer[2] == 0x01)
                                return true;
                else //InputPacketBuffer[2] must have been == 0x00 instead
                                return false;

}
EXPORT bool isLockerClosed(int locker, int door){


                DWORD BytesWritten = 0;
             DWORD BytesRead = 0;
                unsigned char OutputPacketBuffer[65];       //Allocate a memory buffer equal to
our endpoint size + 1
             unsigned char InputPacketBuffer[65];

                OutputPacketBuffer[0] = 0;                  // Første er alltid 0, blir ikke
sendte
                OutputPacketBuffer[1] = 0x85;               // Command for å se om skap åpent
                OutputPacketBuffer[2] = locker;             // skap nr
                OutputPacketBuffer[3] = door;               // luke nr

                WriteFile(WriteHandle, &OutputPacketBuffer, 65, &BytesWritten, 0);
        //Blocking function, unless an "overlapped" structure is used
                ReadFile(ReadHandle, &InputPacketBuffer, 65, &BytesRead, 0);

                //InputPacketBuffer[0] is the report ID, which we don't care about.
                //InputPacketBuffer[1] is an echo back of the command.
                if(InputPacketBuffer[2] == 0x00)
                   return true;

                else
                        return false; }
```

Vedlegg E – Utlegg testkort



Vedlegg F – Utlegg prototyp