Utvikling av et system for automatisert posisjonering av hydrofon

Del-oppgave under prosjektet ”Kartlegging av lydfelter på grunt vann”

Erlend Langbach

Masteroppgave i
Elektronikk og Datateknologi
Fysisk institutt
Universitetet i Oslo
2005

Forord

Denne oppgaven avslutter min mastergrad i ”Elektronikk og Datateknologi” ved Fysisk Institutt, Universitetet i Oslo. Studiet har vært morsomt men samtidig ufattelig frustrerende til tider. Grunnen til at oppgaven lokket var at jeg fikk mulighet til å kunne fikle litt med mekaniske og elektroniske ting, i motsetning til en veldig teoretisk oppgave.

Siden denne oppgaven er en del av et prosjekt for kartlegging av lydfeltet på grunt vann, så håper jeg de som kommer etter meg får bruk for oppgaven når de skal koble opp posisjoneringsdelen sammen med datainnsamlingsdelen. Jeg har prøvd etter min beste evne å sette opp oppgaven systematisk slik at det skal være lett å finne frem til de forskjellige elementene.

Jeg vil takke mine veiledere Helge Balk og Torfinn Lindem for å ha stilt sine ressurser tilgjengelige for meg. De har latt meg jobbe fritt slik jeg liker det best, men når motbakkene har blitt for bratte har jeg fått den nødvendige puffen som var nødvendig. Det er mange som trenger en takk og en god klem, dette er først og fremst muttern og bruttern. Videre har jeg hatt mange gode pauser hvor mye rart har blitt diskutert sammen med både Ines Hafizovic og Eirik Sundve. Tidligere kontorkamerat Armen-Sjur Minassian har gitt et nytt innblikk i livet med mange ”interessante” teorier på flere områder deriblant damer, bilister og ”det spartanske levesettet”. Det er absolutt på sin plass med et stort kyss til kjæresten min Heidi som gir meg stor tro på meg selv og ett *lite* hint når ideene begynner å bli litt for virkelighets fjerne.

Sist takker jeg de støttespillerene som har stått meg absolutt nærmest under hele arbeidet, det er mine to platespillere, min mikser og mine kjære vinylplater. Uten dem ville aldri denne oppgaven blitt en realitet.

Erlend Langbach
Blindern, desember 2005

Innholdsfortegnelse

Innholdsfortegnelse

Figurliste

Innhold på CD-rom

Datablader

- ATMega16 datablad.pdf
- Steppermotor styrekrets.pdf
- Steppermotor 23HS3001datablad.pdf

Kildekode og programmering

- Stepper Controller Delphi program
- Mikrokontroller program for stepping
- Makefile

Ferdig oppgave

- Masteroppgave.pdf

Sammendrag

Denne oppgaven tar for seg design og implementasjon av et automatisert system for nøyaktig posisjonering av en hydrofon på grunt vann. Dette er viktig når lydfeltet utsendt fra et horisontalt skutt ekkolodd skal kartlegges. Man kan så stille seg spørsmålet hvorfor kartlegging av et slikt lydfelt er av interesse?, det er en god grunn til dette. Det har nemlig vist seg at lyden sendt ut fra et ekkolodd på grunt vann ikke sprer seg rett ut i en fin stråle, slik som man skulle kunne tro at man kunne anta ved første øyekast. Det bør nevnes tidlig at produsentene av ekkolodd antar nettopp en slik *fin* spredning. Det viser seg at lyden får en avbøyning enten oppover eller nedover i vannet. Denne avbøyningen har opphav i at lyden sprer seg med litt forskjellige hastigheter i de forskjellige vannlagene, som skyldes temperaturgradienter i vannet. Når man da i tilegg legger til at det kan forekomme refleksjoner fra bunn og/eller overflate, blir virvaret komplett. Et slikt tilfellet med avbøyning og refleksjon vil kunne føre til at lyden blir sperret inne i en såkalt lydkanal.

Det ligger sentralt i problematikken at man skal kunne benytte seg av et ekkolodd til å telle fisk på grunt vann, enten i elver eller innsjøer. Det vil da være vitalt at man får et så nøyaktig resultat som mulig. Hvis man derimot ikke har full kontroll på hvordan lyden sprer seg ut fra (inn til) ekkoloddet, vil man ikke ha kontroll på hvilke områder som dekkes av strålen ei heller om styrken av de returnerte ekkoene er riktig i forhold til hva størrelsen på fisken skulle gi. Dermed kan det hende at man sitter med en situasjon hvor man får feil størrelse på fisken samtidig som at posisjonen til fisken er heller usikker. I tillegg kan fisk som oppholder seg i områder man trodde man dekte med strålen ”unnslippe”. Det håpes nå at leseren har fått et kjapt innblikk i hvorfor kartlegging av lydfelter er ytterst viktig.

Nøyaktig posisjonering er første steg i utviklingen av et automatisert måleoppsett for kartlegging av lydfeltet. Design og implementasjon av den automatiserte posisjoneringsdelen er således denne oppgavens innhold. Posisjoneringsoppsettet er basert rundt et posisjoneringsstativ som var delvis utviklet før denne oppgaven ble påbegynt. Utfordringene bestod i å få en nøyaktig posisjonering ved hjelp av en steppermotor styrt av datamaskinen. For å få dette til, ble det brukt et ferdig utviklet steppermotor driverkort, en USB basert I/O løsning og en mikrokontroller. Programmeringen og implementasjonen av dette er detaljer som overlattes til selve oppgave

Kapittel 1

Innledning

Bakgrunn

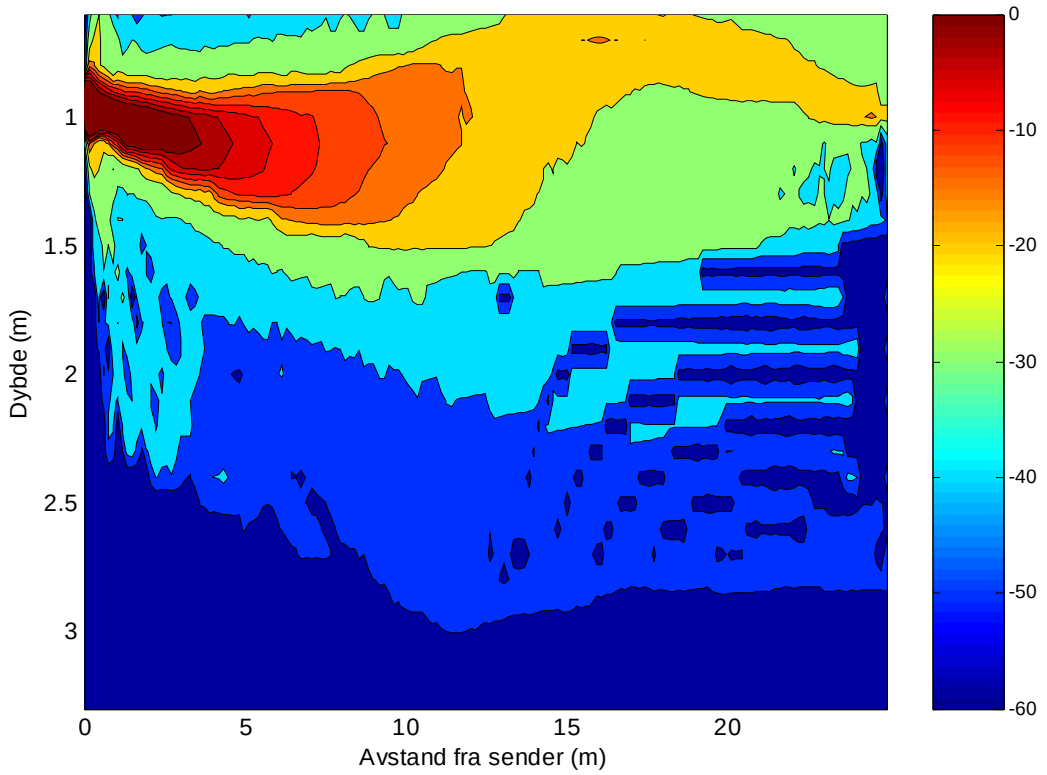
Sonargruppen ved Fysisk Institutt på Universitet i Oslo har i en årrekke arbeidet med bruk av undervannsakustikk til telling av fisk på *grunt vann.* Et ekkolodd/sonar sender en smal, horisontal lydpuls ut i vannet – denne lydstrålen blir justert slik at man får minst mulig refleksjon fra overflate og bunn. Mottatte ekkosignaler gir grunnlag for telling og størrelsesberegning av fisk som befinner seg i strålen på en avstand av opp til 40 meter. Dette er særdeles anvendelig i elver og grunne innsjøer hvor det er umulig å anvende konvensjonelle metoder. Et tradisjonelt ekkolodd - hvor man plasserer transduceren ved overflaten og skyter vertikalt ned mot bunnen blir ubrukelig når vanndypet er 1 til 2 meter. En horisontal lydutbredelse kan imidlertid lett bli forvrengt pga refleksjoner fra overflate og bunn. I tillegg vil en temperaturgradient i vannet fort medføre en avbøyning av lydstrålen. Alle disse feilkildene kan lett føre til stor usikkerhet i ekkoloddets registreringer og det blir meget viktig å kunne registrere strålegangen i detalj.

En SONAR sender ut en fokusert lydstråle i vannet. Når avstanden til et mål således skal beregnes, antar man at lyden sprer seg i rette baner ut fra svingeren. Dette er derimot bare tilfellet når temperaturen i vannet er konstant. Temperaturvariasjoner som er tilstedet i vannlagene gir opphav til en lydhastighetsprofil som fører til en avbøyning av lydstrålen, slik at lyden kan gå uventede retninger. På grunt vann vil dette fenomenet komme enda bedre tilsyne. Refleksjoner fra bunn eller overflate i sammenheng med avbøyningen kan da føre til at lyden blir sperret inne i såkalte lydkanaler. Dette var noe hovedfagstudentene Kai Morgan Kjølerbakken [12] og Vibeke Jahr [10] ved Fysisk Institutt, Universitetet i Oslo konkluderte med i sine oppgaver. Resultatene var basert på målinger og simuleringer.

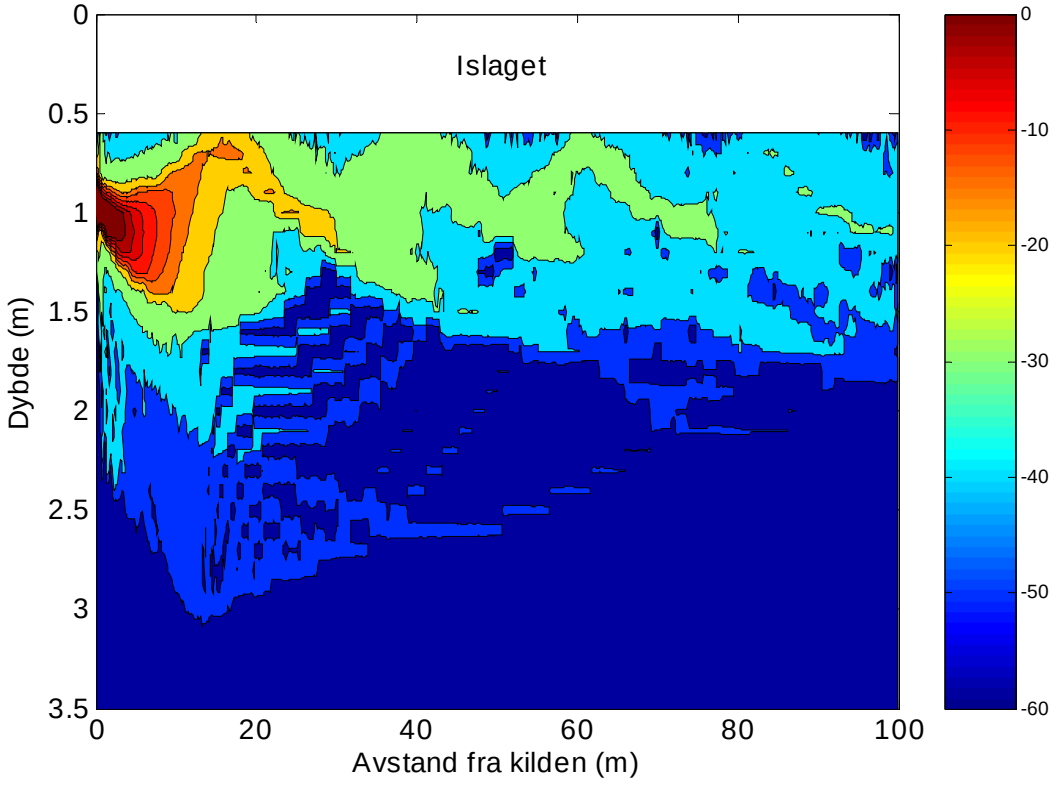
Produsenter av ekkolodd antar en sfærisk spredning av ekkoene i vannet, og dermed en avtagning av intensiteten som følger et $1/R^2$ forhold. Hvor R er avstanden fra kilden. Det har derimot vist seg at denne antagelsen ikke bestandig er korrekt. Videre konklusjoner fra Kjølerbakken og Jahr i tilfellet hvor lyden blir sperret i en kanal, er at lyden vil ligge nærmere en sylindrisk spredning slik at intensiteten vil avta mer i nærheten av ett 1/R forhold fremfor det oppgitte $1/R^2$ forholdet.

Det er hovedsakelig to grunner til at det er viktig å undersøke lydutbredelsen. Den første grunnen er at det er viktig å kjenne det området som dekkes av strålen til ekkoloddet, slik at man vet dekningsgraden på de utførte målingene. Den andre er at det er viktig å vite styrken på det returnerte ekkosignalet, siden dette direkte angir størrelsen på målet.

Videreføringen av simuleringsarbeidet ble gjort av Ines Hafizovic [8]. Hun utviklet programvare basert på ray-tracing til å estimere lydutbredelsen med den målte lydhastighetsprofilen som input. Figur 1 under viser den simulerte lydutbredelsen for islagt vann inntil bredden. Ser vi så videre i Figur 2 har det simulerte området blitt utvidet utover det området som lydhastigheten har blitt målte for. ”Vi ser at mesteparten av energien forblir under isen. Under slike forhold kan deteksjon av objekter som befinner seg dypere enn 1,5 meter under overflaten være vanskelig.” [8, side 102]. Figur 1 og Figur 2 er hentet fra oppgaven til Hafizovic.



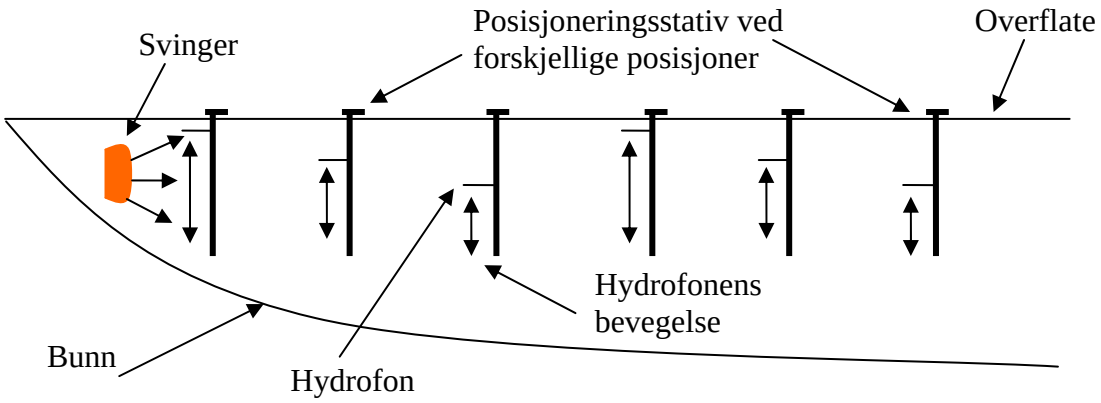
Figur 1 - Simulert intensitet ved islagt vann. Hele området.



Figur 2 - Simulert intensitet, ved islagt vann. Utvidet området.

Kjølerbakken og Jahr fikk laget et motorisert stativ som var tenkt benyttet til å posisjonere hydrofonen i dybderetningen. På denne måten kunne man få en nøyaktig posisjon på hydrofonen og dermed få kartlagt lydfeltet med god oppløsning. Grunnet tekniske problemer med stativet ble det ikke anledning til å ta stativet i bruk. Posisjoneringen av hydrofonen ble da gjort manuelt og måleserier ble gjort hver tiende centimeter i dybderetning og hver femte meter i lengderetning.

Sonar gruppa hadde et ønske om å muliggjøre bruken av det allerede produserte stativet for å oppnå mye høyere presisjon i målingene enn hva det manuelle oppsettet kunne gi, men da med forandringer på selve styringsdelen og motoren. På denne måten får man muligheten for å nøyaktig kartlegge lydfeltet på grunt vann for så å kunne sammenlikne dette med simuleringsmodellen som hadde blitt produsert. Slik man ønsket at systemet skulle fungere var at posisjoneringsstativet skulle settes ved forskjellige avstander fra svingeren og ved hver posisjon skulle det kjøres en automatisk måleserie. Se Figur 3 under for et overblikk over driften av posisjoneringsstativet.



Figur 3 – Overblikk over posisjoneringsstativets drift.

Mål/problemstilling

Målet med denne oppgaven er å lage et automatisert system for nøyaktig posisjonering av en hydrofon. Det skal lages et enkelt dataprogram som kommuniserer med systemet via

Universal Serial Bus (USB). For å få god nøyaktighet skal systemet bygges opp rundt en steppermotor.

Videre skal det legges tilrette for et enkelt datainnsamlingssystem hvor signalene fra hydrofonen kan registreres innenfor et dybdeområde på minst 1,5 meter. Problemstillingen i oppgaven er å kunne angi posisjonen til en hydrofon med tilstrekkelig nøyaktighet slik at lydfeltet fra en ”eliptisk split-beam transducer” kan kartlegges med en oppløsning bedre enn en bølgelengde. Med en operativ frekvens på 120kHz tilsier dette en oppløsning bedre enn 1cm.

Metoder

En steppermotor med driverkort skal brukes til å styre et stativ som fører hydrofonen opp og ned i vannlagene. Brukegrensesnitt for styring skal skje ved hjelp av programvare utviklet i Delphi. Til kommunikasjon mellom programvare og steppermotoren vil det bli benyttet et ferdigprodusert USB basert brett for analog og digital inn og ut, PMD-1208LS. Videre vil det bli benyttet et Atmel STK500 utviklingskort for å programmere en Atmel Mega16L mikrokontroller for å få høy nok frekvens til steppermotorens styringskort.

Verdt å merke seg

Det er inkludert en CD-rom plate med denne oppgaven. På den ligger den ferdig utviklede programvaren, samt de fleste datablader. I tillegg ligger selve oppgaven i pdf format. Videre er også makefilen[1] som benyttes under programmering av mikrokontrolleren inkludert. Se innholdsfortegnelse for fullstendig liste over innhold.

[1] Makefile er laget av Eric B. Weddington, Jörg Wunsch, et al.

Kapittel

2

Materialer og Metode

Målet med denne oppgaven var å lage et automatisert system for nøyaktig posisjonering av en hydrofon. Som utgangspunkt for arbeidet hadde vi en gammel målerigg som instrumentverkstedet og Elab ved Fysisk institutt hadde bygget for Lindem for noen år tilbake. Det viste seg dessverre at denne riggen fungerte dårlig under feltarbeid. Studentene som arbeidet med simulering av lydfelt på grunt vann fikk ikke måledata som til fulle kunne underbygge den teoretiske modellen. Det kom klart fram i hovedfagsoppgaven til Ines Hafizovic [8] at verifiserbare måleserier var nødvendig for videre teoretiske arbeider med simuleringsmodellen. Stort sett kan man si at ideen til denne oppgaven er å få liv i en ellers så død posisjoneringsrigg slik at nøyaktige måleserier kan bli gjennomført.

For å få dette til å fungere har det vært en del ting som måtte passe rimelig godt sammen. Det startes med å gi en gjennomgang av hvordan det var ønskelig at stativet skulle kunne opereres. Deretter vil det bli gitt en grundigere gjennomgang av hvert element under *Utstyr som ble brukt*. Avslutningsvis vil det bli gått grundig igjennom selve implementeringen av oppsettet.

Ønskelig drift og funksjonalitet

Den første ideen til hvordan stativet skulle kunne operere var at det skulle kjøres automatisk sammen med et eksternt datainnsamlingssystem. For at et slikt datainnsamlingssystem skulle kunne bli påbygd i etterkant ble det tatt høyde for at man trengte to signaler som kunne formidle kommunikasjonen mellom disse. Ett signal skulle formidle at posisjoneringssystemet hadde oppnådd den ønskelige

posisjonen og at det da var klart for datainnsamling. Det andre signalet skulle komme fra datainnsamlingssystem å gå til posisjoneringsdelen. Dette signalet skulle da si ifra at datainnsamlingsdelen var ferdig med å innsamle den nødvendige dataen og at det da var klart for eventuell videre forflytning av den påmonterte hydrofonen.

Etter hvert som ting begynte å fungere ble det innsett at man også måtte ha en måte å styre bevegelsene til posisjoneringsstativet manuelt på. Dette kom som et krav fordi det kunne oppstå situasjoner av forskjellige årsaker hvor en måleserie måtte avbrytes, dermed var det nødvendig å kunne komme tilbake til utgangspunktet. Videre var denne metoden også nødvendig når startposisjonen i vannlaget til hydrofonen skulle settes. Hvis man for eksempel var på en lokasjon hvor det var meget grunt vann kunne tilfellet oppstå hvor hydrofonen ble stående over vann. Dette ville da være et lite gunstig tilfelle, slik at muligheten til å justere startposisjonen for målingene var viktig.

Den transduceren som blir benyttet på Sonar gruppa ved Fysisk institutt er en såkalt ”eliptisk split-beam transducer” av typen ES 120-4. Denne opererer med en frekvens på 120kHz, noe som tilsvarer en bølgelengde rett i overkant av 12mm[1]. Det var ønskelig å oppnå en oppløsning på de vertikale måleseriene bedre enn en bølgelengde. Det ble således bestemt at det skulle foretaes en måling hver halve bølgelengde. I praksis betyr det at når man kjører i automatisk målemodus skal hydrofonen tilbake legge en distanse på 6mm mellom hver gang posisjoneringsdelen sier i fra at en måling kan gjennomføres.

Den totale høyden på stativet er litt over 2,2 meter. Selve rørene hvor hydrofonfesteplatene er påmontert er rett over 1,9meter derimot er det bare et totalt intervall på 1,55meter hvor festeanordningen faktisk kan bevege seg. Høyden på festeplatene til hydrofonen er cirka 35cm. For å forhindre at noe kunne gå galt måtte styringsprogrammet som skulle utvikles ha innebygde rutiner for at motoren ikke skulle kunne gå utenfor et forhåndsbestem lovlige intervall. Dette intervallet måtte da også ha litt slakk å gå på, slik at ingenting uforutsett kunne skje. Skulle det nå

[1] Hvor man antar en lydhastighet på 1466,5 m/s

allikevel oppstå noe uforutsett, ble det også tiltenkt at systemet måtte ha en måte for å bryte av selve steppingen. Dette ble gjort ved å lytte på reset signalet på mikrokontrolleren. Hvis reset signalet ble aktivert, ville styringsprogrammet stoppe en eventuell pågående måleserie.

Etter at posisjoneringssystemet hadde forflyttet hydrofonen en halv bølgelengde, skulle den generere et logisk signal som sa ifra til datainnsamlingsdelen at den kunne begynne. For at posisjoneringsdelen skulle forflytte seg videre etter at datainnsamlingsdelen var ferdig ble det påkrevd at posisjoneringssystemet fikk et logisk signal tilbake som sa ifra at datainnsamlingen var ferdig. Det ble da antatt at enten samplingsdelen genererte dette signalet, eller at man eventuelt kunne lytte på samplingsdelens analoge til digitale omvandler sitt *Conversion Flag Finished* eller ett tilsvarende signal.

En viktig del av oppsettet var at man skulle kunne vite med stor nøyaktighet hvor i vannlaget hydrofonen befant seg til en hver tid. For å kunne samkjøre posisjon og måling i ettertid ved hver målingsposisjon ble systemet konstruert slik at hver gang systemet ga signal om at måling skulle skje ble posisjon til hydrofonen skrivet til en tekstfil.

Overblikk over oppsettet

For å kunne holde følge når det senere blir gått igjennom de forskjellige elementene som posisjoneringsoppsettet består av, er det essensielt at man har et lite overblikk over hvordan ting henger sammen. Først og fremst så har vi to deler av systemet, et som står på land mens den andre delen er i vannet. Et generelt overblikk over oppsettet er gitt i Figur 1.

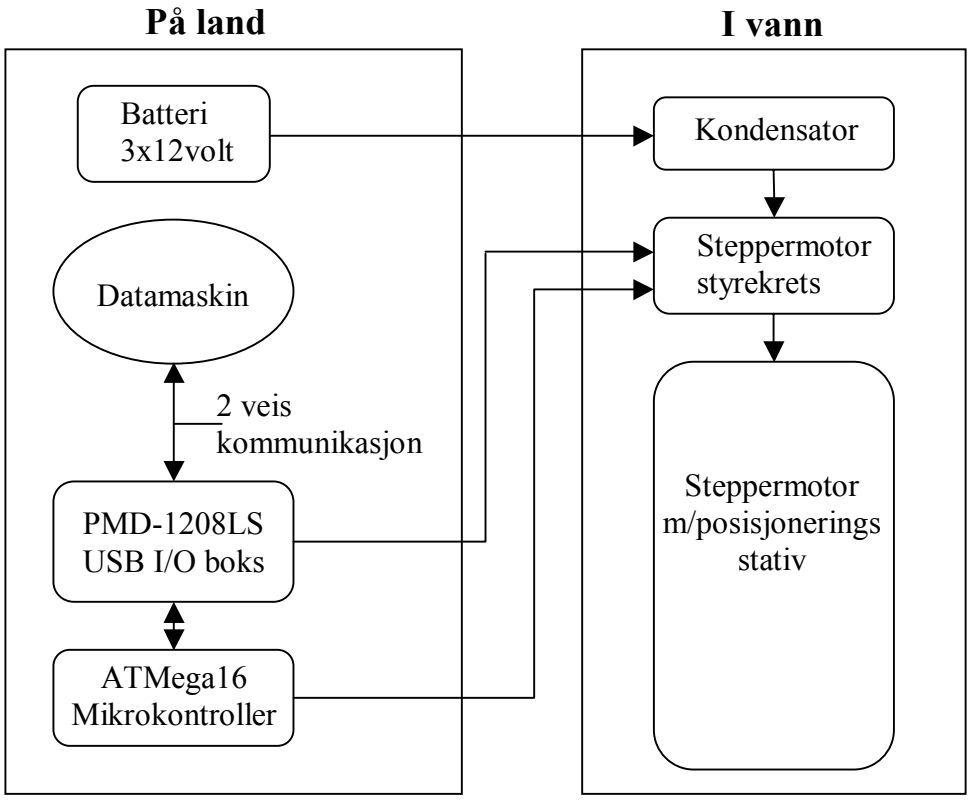
På land

Den delen av posisjoneringsoppsettet som er på land består av tre stykk vanlige 12 volts bilbatterier koblet i serie for å kunne gi posisjoneringsstativets motor nok spenning. Videre har vi en datamaskin som kjører det grafiske grensesnittet (GUI) som brukeren benytter for styring. Datamaskinen er forbundet med en ytre

tilkoblingsboks, PMD-1208LS [4], for inn og ut signaler via USB porten. For å assistere den tilkoblede USB boksen med selve steppingsprosessen er det koblet til en mikrokontroller. Grunnen er at USB boksen rett og slett ikke greier å steppe raskt nok. Oppgaven med å generere pulstog til steppermotorens styringsenhet er derfor gitt til mikrokontrolleren for å oppnå de nødvendige hastigheter som kreves.

I vann

Kontrollsignaler fra USB boksen overføres til steppermotorens styringsenhet som er lokalisert ute ved selve motoren. Dette er signaler som *enable*, altså aktivisering av motoren, samt retningssignalet. Signalet fra mikrokontrolleren brukes til genereringen av selve steppingen, altså til å sende forskjellige type pulstog til motorens styringsenhet i forhold til hva som brukeren vil at skal skje. For å være sikker på at motorens styringsenhet skal kunne trekke tilstrekkelig strøm under "steppingen" er det montert en stor kondensator ute på styringsstativet.



Figur 1 – Oversiktskart over posisjoneringsoppsett

Utstyr som ble brukt

Posisjoneringsstativet

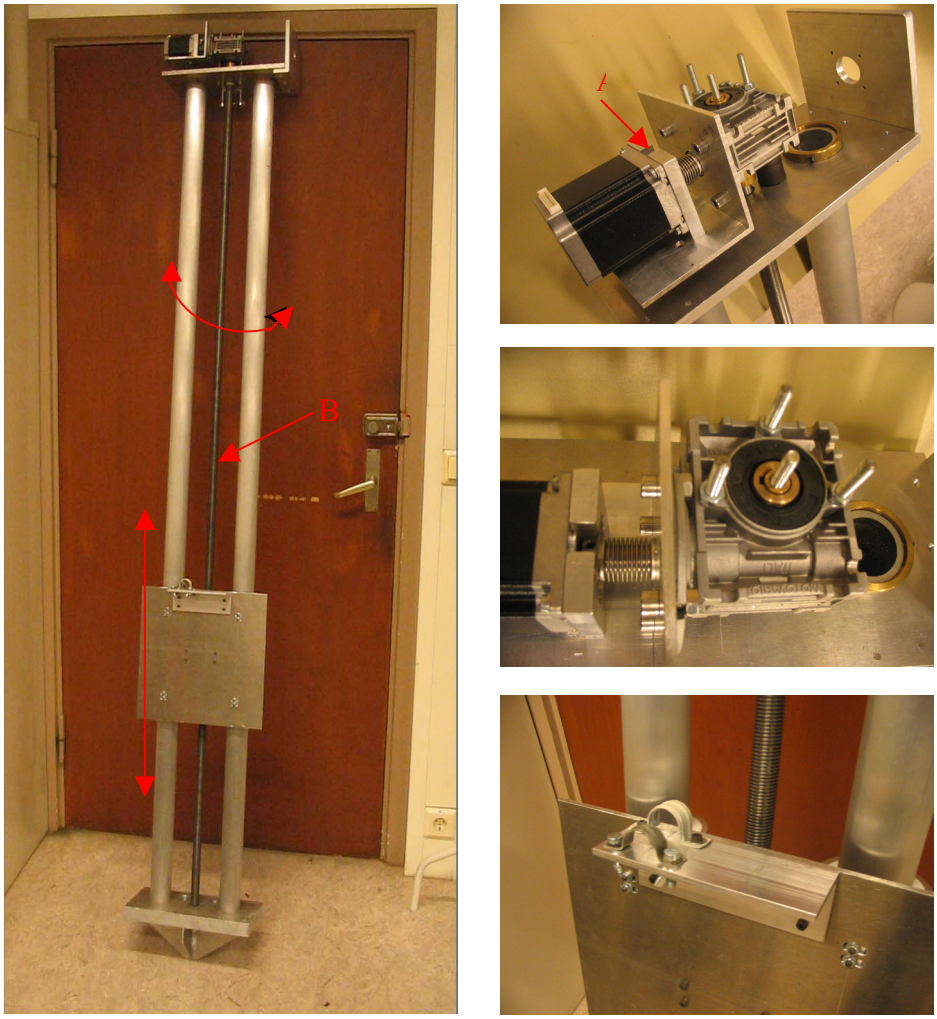
Bakgrunn

Det har blitt produsert et posisjoneringsstativ ved Fysisk Institutt, men som nevnt tidligere i oppgaven fungerte ikke dette helt som forventet. Det var opprinelig en 24 volts likestrømsmotor påmontert, se Figur 3. Denne skulle drive oppsettet og styres fra land via en kontroller. Problemet ble at det ble for stort spenningstap over kablene slik at måleoppsettet i praksis ikke kunne bli benyttet i feltarbeid. Det ble fremmet et ønske fra Sonar gruppen om at de gjerne ville få stativet til å fungere, slik at man kunne få kartlagt lydfeltet med tilstrekkelig nøyaktighet. På denne måten kunne man få data som indikerte om de simuleringsmodellene som var blitt produsert i de tidligere hovedfagsoppgaver ved gruppen var korrekte.

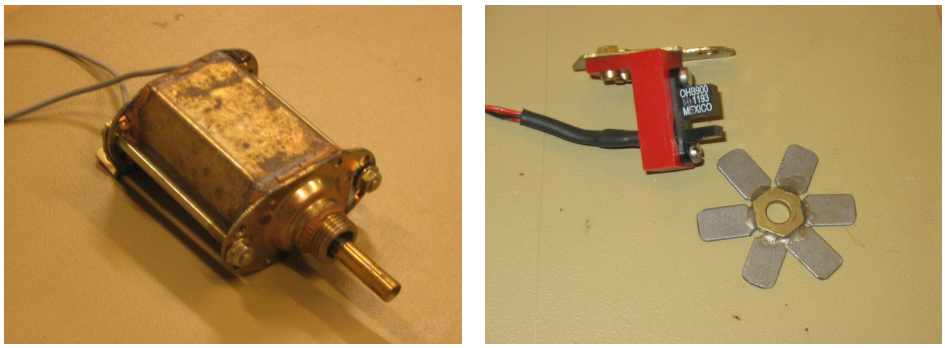
Original utførsel

Stativet var opprinnelig laget av to aluminiumsrør til selve konstruksjonen samt topp og bunn plater. Videre var det brukt en skrue, se merknad B i Figur 2, for selve mekanismen som styrte hevningen og senkningen av hydrofonen. I overgangen mellom motoren og skruen var det brukt en girboks[2] med en utveksling på 20:1. Dette vil med andre ord si at motoren måtte gå 20 ganger rundt for å oppnå en fullstendig rotasjon på skruen. I horisontal bevegelse, for en eventuelt påmontert hydrofon, vil dette tilsvare 2,5mm. På det opprinnelige stativet ble det brukt en *Hall effekt sensor* med en tilhørende sekskantet magnetisk tellemekanisme, se Figur 3, for å kunne ha kontroll over den horisontale bevegelsen til hydrofonen. Dette ble gjort ved å feste den sekskantede delen på toppen av skruen og Hall effekt sensoren slik at den sekskantede delen gikk i mellom sensoren. På denne måten ble det i teorien generert en puls for hver 1/6 rotasjon som skruen på stativet gjorde. Det skulle videre benyttes en lang metallstang til å feste hydrofonen et stykke bort fra selve stativet, slik at man unngikk problemer med refleksjon fra selve stativet ved datainnsamling.

[2] Motovario S.p.A Italy, type: NMRV-025.



Figur 2 – Modifisert posisjoneringsstativet til venstre. Øverst til høyre er den nye steppermotoren, deretter gir boksen og på bunnen deler avfestningsanordningen til hydrofon

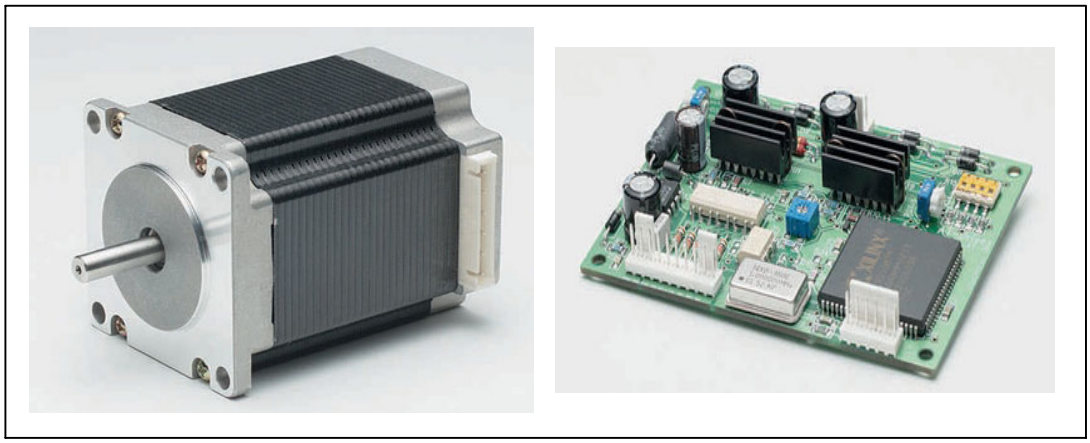


Figur 3 – Opprinelig motor og Hall Effekt tellemekanisme

Forbedringer på stativet

For å få posisjoneringsriggen til å fungere ble ideen om å bruke en steppermotor som man kunne kontrollere fra datamaskinen luftet. Se Figur 4 for bilde av steppermotoren. Siden en steppermotor ikke trekker konstant strøm fra en eventuell strømkilde, kunne man ved hjelp av en stor nok kondensator[3] ute ved motoren slippe problemet med spenningstapet over kablene fra land til stativet. Det ble da antatt at kondensatoren ville få muligheten til å lade seg opp imellom steppene og/eller pulstogene.

For å kunne få festet steppermotoren ble den gamle motoren og den tilhørende Hall effekt sensor tellemekanismen samt festeanordning tatt av. Det ble så laget en ny aluminiumsplate på det mekaniske verkstedet ved Fysisk institutt til å holde steppermotoren, se bildet øverst til høyre på Figur 2 merket med A. Ellers ble ikke stativet ytterligere modifisert.



Figur 4 – Fabr Moons 23HS3001-01 motor og tilhørende Fabr AT Drives AR15R05-5 driverkrets

[3] I størrelsesorden noen titusen μF.

Steppermotor og motorstyringskrets

Generell info

En steppermotor lar seg skille fra andre motorer ved at den opererer ved å dreie en fast vinkel per stepp eller puls den mottar, i kontrast til kontinuerlig drift slik som ved f.eks. en vanlig likestrømsmotor. Dette impliserer derimot ikke at en steppermotor ikke kan operere kontinuerlig, det er bare å sende motoren pulstog med tilstrekkelig høy frekvens. Motorene kommer i et vidt forskjellig spekter, alt fra små til store i størrelse samt et stort spekter av dreiningsvinkel per stepp. Noen vil ha et fast totalspenn på oppnåelig vinkel[4], mens andre kan dreie i det uendelige. Videre er det et relativt stort spekter av størrelsesordenen på dreimomentet som de forskjellige motorene har. For en grundig innføring i steppermotorer, dets operasjon og styringsdesign henvises leseren til materialet som *1. amanuensis*[5] *Douglas W. Jones* ved *University of Iowa* har publisert under tittelen "Jones on Stepping Motors" på internett [11].

I materialet som Douglas W. Jones har publisert står teorien bak hvordan fasen til pulsene skal styres og manipuleres for å oppnå den spesifikke ønskede driften av motoren. Steppermotorer har nemlig muligheten til å ikke steppe fulle steg, det kan da som oftest steppes i halv, kvart og åttendedels steg, samt selvfølgelig helsteg. Det er motorens styringskrets som regulerer alternativene som brukeren har, men det er en kombinasjon av motoren og styringskretsen som bestemmer hva som er oppnåelig for oppsettet. Styringskretser kan enten designes selv, eller kjøpes ferdig. For denne oppgavens valg av motor så var tilgjengeligheten av en ferdig laget styringskrets en sentral faktor.

Valg av motor og driverkrets

For i det hele tatt å få kommet i gang med oppgaven var det essensielt at en motor ble kjøpt inn, slik at testing av hvordan den fungerte kunne starte. Dette ble innsett etter at

[4] F.eks. servo motorer som blir brukt til å styre radiostyrte biler

[5] Associate Professor

alt for mye tid ble brukt på å sette seg inn i de forskjellige typene av steppermotorer og dets opperasjoner og styring. Valget av motor falt på steppermotoren *Fabr Moons 23HS3001-01* [7] og den tilhørende styrekretsen *AT Drives AT15R05-5* [1]. Motoren har en vekt på 1kg og et dreimoment på 1,25Nm. Om dette kom til å være tilstrekkelig var ikke helt klart i utvelgelsesfasen, men en plass måte man starte. Motoren er av typen bipolar tofasers hybrid steppermotor, den ble oppgitt til å være en robust og korrosjonsbeskyttet motor noe som passet ypperlig for denne oppgavens bruksfelt. Styringskretsen er operative mellom 12 og 48 volt og benytter 5volts TTL logikk for styringssignaler. For mer gjennomgående informasjon om motoren og styringskretsen henvises det til databladene for disse i appendiksen og CD-rom plata som er inkludert i oppgava. Bilder av motor og styringskretsen[6] er vist i Figur 4.

I/O signaler for steppingsenheten

Driverkretsen har 6 innganger og 4 utganger, pinnekonfigurasjonen er gitt i Figur 5. Pin1 og pin2 brukes respektivt til driftsspenning og jord for motoren. Spenningen må være mellom 12 og 42 volt, i denne oppgaven ble det valgt å bruke 32volts driftspenning. Dette ble klart etter litt testing i labben med forskjellige batteri konfigurasjoner. Det viste seg at med mindre enn tre seriekoblede 12 volts batterier greide man ikke å oppnå ønsket drift. Dette var da vel og merke uten en ekstra kondensator, slik planen var å ta i bruk. Pin3 er logisk jord, pin4 gir muligheten til å skifte retning. Videre er pin5 pinnen som skal motta pulstogene. For hver gang signalet inn på pin5 går høyt vil motoren gå en vinkel på 1.8 grader[7]. Pin6 er *enable* signalet, dvs. slår motoren av eller på. Ut i fra databladet til driverkretsen blir det oppgitt at den har en maksimal frekvens på 20kHz, slik at vi fint er innenfor denne grensen med frekvensen vi opererer på her[8].

[6] Bildene må ikke brukes til å sammenlikne størrelsen på komponentene.

[7] Så frem ikke mikrostepping er satt til annet enn helstepping.

[8] Maksimal frekvens er 1,85kHz.

PIN ANS.	FUNCTION FUNKTION	DESCRIPTION BENÄMNING
1	PWR MATNING	8-42VDC supply input. (Voltages under 12VDC and a very low pulse speed (pps) can cause disturbances). Matning 8-42VDC (Spänning under 12VDC och väldigt låg pulshastiget kan ge upphov till störningar).
2	GND	Ground 0V for input power. Jord 0V för matning.
3	IGND*	Ground for the logic signals 5VDC or 24VDC. Jord för logiksignaler 5VDC eller 24VDC.
4	DIR*	Signal goes high at (5VDC or 24VDC) and low (0VDC). The motor turns direction (CW/CCW). Signalen går hög vid (5VDC eller 24VDC) och låg vid (0VDC). Motorn vänder riktning (medsols/motsols).
5	CLK*	Step clock: The motor moves one step on the rising edge of this signal. Klocka in: Motorn tar ett steg på hög flank.
6	ENB*	Enable/Disable: Starts and stops the drivers (the motor).Signal goes high at 5VDC or 24VDC(Enable). When the signal goes 0VDC it goes low (Disable). På/Av: Stoppar och stannar drivkretsarna (motorn). När signalen är 5VDC eller 24VDC går signalen hög (På). När signalen är 0VDC går den låg (Av).
B-	Motor	Output/Utgång
B+	Motor	Output/Utgång
A-	Motor	Output/Utgång
A+	Motor	Output/Utgång

Figur 5 – Pin-konfigurasjonen til steppermotorens styrekrets.

Videre har vi utgangssignalene til motoren, B-, B+, A- og A+. Disse signalene ble koblet til motorens respektive innganger for å få fasene riktig. Tabell 1 nedenfor gir en oversikt over oppkobling av motoren til styrekretsen. Det er tatt utgangspunkt i de fargene på ledningene som kommer med motoren, men dette kan selvfølgelig skiftes forløpende så man bør ta utgangspunkt i pin-nummeret for å få riktig oppkobling.

Utgang fra driverkort	Motor pin#	Ledningsfarge til motoren
A+	3	Rød
A-	5	Blå
B+	7	Gul
B-	9	Hvit

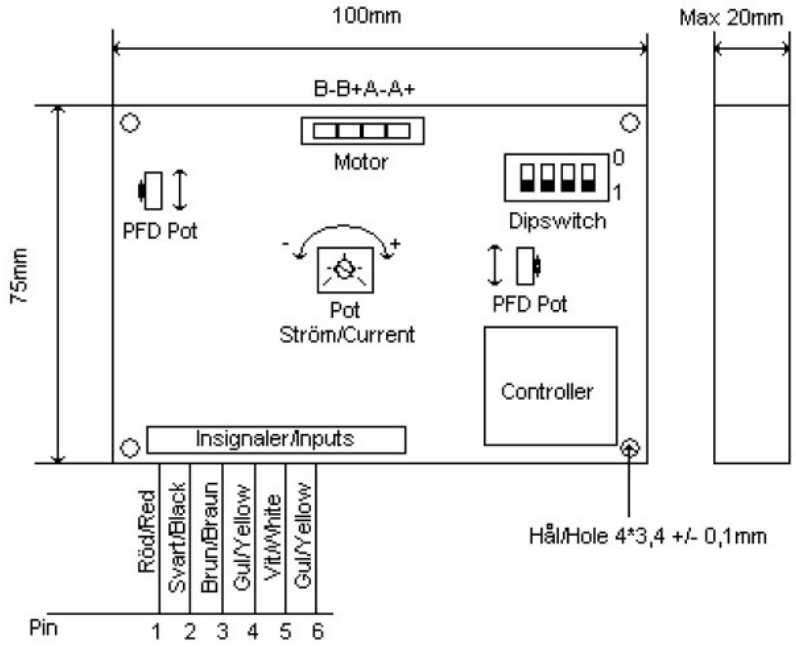
Tabell 1 - Oversikt over motor oppkobling

Figur 6 på neste side gir en liten oversikt over selve oppbygningen til driverkortet. Foruten om inn og utsignalene er det i tillegg fire brytere[9] samt to potentiometre. Bryterne 1 og 2 benyttes til å velge om man vil mikrosteppe[10], mens bryterne 3 og 4 i sammen med de to potentiometrene gir muligheter for justering av strøm og fase. For

[9] Merket "Dipswitch" på Figur 6

[10] Hel, halv, kvart eller åttendels steg kan benyttes.

detaljer angående setting av disse parametrene henvises leseren til databladet [1] til driverkortet.



Figur 6 - Styrekrets oversiktsdiagram

Ekstern I/O boks

Measurement Computing PMD-1208LS ekstern I/O boks

Measurement Computing produserer en rekke kort og eksterne brett/bokser for datainnsamling, blant annet en serie med USB baserte kommunikasjonsbokser til bruk både for digitale og analoge inn- og utsignaler. Disse boksene er primært rettet mot hobbybrukeren grunnet sitt relativt gunstige prissjikt, men andre kan også dra verdifull nyte av dem.

Figur 7 - PMD-1208LS[11]

Den nye standarden innenfor kommunikasjon med omverdenen blir mer og mer dreid bort i fra f.eks *Peripheral Component Interconnect(PCI)*-løsninger og mer over til USB og Firewire oppsett. Båndbredden på en PCI buss er fortsatt høyere enn på USB/Firewire løsningene, men siden brukeren får et atskillig lettere oppsett å forholde seg til så har de eksterne løsningene kommet for å bli. Siden det var et høyt ønske om å lage et posisjoneringsoppsett som var kompatibelt med fremtiden, var veien via USB oppkobling et naturlig valg.

Det ble relativt tidlig valgt å gå til innkjøp av en av disse boksene. Valget havnet på *PMD-1208LS Personal Measurement Device*. Denne boksen har 8 analoge inn kanaler og to stykker 8 bits digitale porter konfigurerbare til enten inn eller ut porter. Når innkjøpet ble gjort var det ikke helt klart hva som faktisk behøvdes av inn og utganger for å få prosjektet i havn. Siden boksen virket relativt universell så dette ut som en grei løsning man med letthet kunne jobbe videre med. Se Figur 7 for bilde av PMD-1208LS[11], og videre Figur 8 for relasjon mellom fysiske innganger og signal navn som brukes innad i boksen/programvare.

[11] Bildet er egentlig av en USB-1208LS, men disse er forstått til å være ekvivalente.

8-channel single-ended mode

Pin	Signal Name	Pin	Signal Name
1	CH0 IN	21	Port A0
2	CH1 IN	22	Port A1
3	GND	23	Port A2
4	CH2 IN	24	Port A3
5	CH3 IN	25	Port A4
6	GND	26	Port A5
7	CH4 IN	27	Port A6
8	CH5 IN	28	Port A7
9	GND	29	GND
10	CH6 IN	30	PC+5V
11	CH7 IN	31	GND
12	GND	32	Port B0
13	D/A OUT 0	33	Port B1
14	D/A OUT 1	34	Port B2
15	GND	35	Port B3
16	CAL	36	Port B4
17	GND	37	Port B5
18	TRIG_IN	38	Port B6
19	GND	39	Port B7
20	CTR	40	GND

Figur 8 - PMD-1208LS Portkonfigurasjon i Single-ended mode

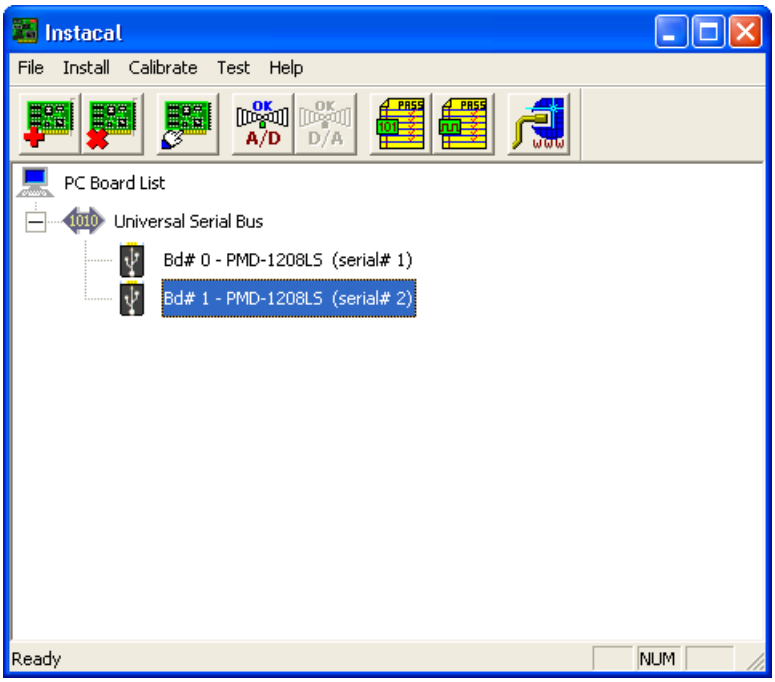
I første omgang var det tenkt at den eksterne USB boksen skulle styre alle signaler som var nødvendig for fullstendig kommunikasjon mellom datamaskinen og steppingsenheten[12]. Etter at en midlertidig styringsprogramvare var utviklet viste det seg etter mye tidkrevende testing i labben samt mye omprogrammering, at USB boksen ikke kunne oppnå en høyere steppingsfrekvens enn rundt ca 70Hz[13]. Videre ville også denne steppingsfrekvensen være avhengig av datamaskinens klokkehastighet[13]. En maksimal frekvens på 70Hz ble sett på som langt under hva som var nødvendig for at systemet skulle være tidsmessig forsvarlig å bruke ved feltarbeid. Kjappere er som oftest bedre, men ved å oppnå en steppingsfrekvens i underkant av 2 kHz vil stativet kunne bruke omkring 30 minutter på en måleserie med hydrofonen fra topp til bunn. Dette ble ansett som et greit tidsskjema. For å kunne oppnå dette måtte det introduseres en dedikert pulstog generator. Mer om dette under mikrokontroll delen.

[12] Styrekrets og motor.

[13] Etter korrespondanse med produsenten.

Measurement Computing Instacal

Med den eksterne I/O boksen følger det også med programvare kalt *Instacal*. Dette benyttes til å administrere boksene, og angi et spesifikt nummer til hver boks. Videre anvendes det til kalibrering av de analoge inngangene og dets analog til digitale omvandlere. Spesielt hendig blir denne programvaren når man skal benytte seg av mer enn en boks, siden programvaren har inkorporert en funksjon som gjør at man kan blinke med lysdioden som er lokalisert på brettet, se Figur 7 for bilde av boksen. Det å kunne blinke med en lysdiode høres kanskje ikke så hendig ut? Derimot for å kunne skille en boks fra de andre er lysdioden meget nyttig når styrings programmer skal programmeres. Et skjermbilde av Instacal programmet når to PMD-1208LS bokser er tilkoblet er vist under i
Figur 9.



Figur 9 - Skjermbilde av Instacal med 2 PMD-1208LS bokser tilkoblet

Measurement Computing Universal Library

I PMD-1208LS pakka medfølger det, i tillegg til boksen og Instacal, det meget hendige funksjonsbiblioteket *Universal Library* [5][6]. For å kunne programmere Measurement Computing sine eksterne I/O brett bør man absolutt bruke dette[14].

Funksjonsbiblioteket er kryss kompatibelt med de forskjellige boksene til Measurement Computing. Biblioteket er oppbygd av flere høynivåfunksjoner for de fleste vanlige operasjoner som boksene kan utføre, deriblant funksjoner for konfigurering av porter samt innlesning og utsetting av digitale signaler[15]. Hvis man ønsker å gå fra en boks til en annen, men har flere velfungerende programmer som man ønsker å ta med seg videre, så er ikke dette et problem. Selv om hardwaren inne i de forskjellige boksene fungerer forskjellig og dermed er bygd opp på forskjellige måter, så passer Universal Library på at ting blir kalt på riktig vis. Det ligger i bakgrunnen en antagelse om at man sjekker at de nye boksen man ønsker å bruke har muligheten til å utføre de kallene man tidligere har gjort. Det vil si at man ikke kan bruke en funksjon som de nye boksene ikke er kompatible med.

Universal Library er videre kompatibelt over en mengde forskjellige programmerings språk. Det vil si at identiske funksjoner med identiske parameterkall finnes for forskjellige programmeringsspråk, blant disse er *Delphi*, *Visual Basic* og *Borland C++*[16]. Dette betyr i praksis at går man fra et språk til et annet så vil ikke selve kallet fra funksjonsbiblioteket til Measurement Computing være annerledes.

Det følger videre med en rekke programmeringseksempler for de fleste kompatible språkene i pakka, slik at det å få startet med selve programmeringen blir hakket lettere.

[14] Så frem man ikke vil skrive alle driverne selv.

[15] I tillegg er det et hav av andre funksjoner for total liste se *Function Referance*.

[16] Fullstendig liste over kompatible språk: Universal Library User's Guide side 2.

Programmering av Delphi for operasjon med PMD-1208LS

Generelt

Det antas at leseren av oppgaven har programmeringserfaring, slik at ting som er helt basis for programmering ikke vil bli gjennomgått. Derimot vil de elementene som er spesielt for oppgaven gått igjennom. Dette vil da spesielt gjelde bruken og programmeringen av de metodene som er inkludert i Universal Library pakka. Leseren henvises til "Universal Library Function Reference" [6] for utfyllende informasjon angående de forskjellige funksjonene som blir nevnt og brukt i denne seksjonen. Programmet som ble benyttet til selve programmeringen av Delphi var Borland Delphi 6.0.

Feilhåndtering

Feilhåndtering har blitt inkludert i Universal Library. Dette vil slå ut hvis det skjer noe galt med selve boksen, eller at ting blir gjort som er i strid med den konfigurasjonen som har blitt satt. Man kan sette parametere for hvor strikt feilhåndteringen skal være. I denne oppgaven ble den satt til å rapportere og stoppe ved alle feil. Feilhåndteringen har i denne oppgaven slått ut når brett har blitt konfigurert feil i Instacal og når portene har blitt konfigurert på en måte som hardware ikke tillater.

Koden vist i Figur 10 ble brukt i Delphi for at programmet skulle rapportere alle feil og stoppe på alle feil. De to siste linjen i kodesnutten i Figur 10 gjør slik at hvis selve funksjonen som blir kalt, i dette tilfellet *cbErrHandling*, ikke greier å utføre ønsket oppdrag vil denne returnere noe annet enn 0 slik at programmet stopper opp. Dette er en standard måte å gjøre ting på når Universal Library blir brukt. Alle funksjoner som er definert igjennom Universal Library vil returnere en verdi som sier noe om den greide å utføre den ønskede operasjonen eller ikke. Så lenge det blir returnert 0 vil det ikke skje noen ting. Det har ikke blitt erfart at noen av funksjonene som har blitt kalt har returnert noe annet enn 0.

```
{oppsett for internal error handling for the Universal Library}
ErrReporting := PRINTALL;       {setter Universal Library til å skrive ut alle evt. errors}
ErrHandling := STOPALL;         {setter Universal Library til å stoppe på alle evt. errors}
ULStat := cbErrHandling(ErrReporting, ErrHandling);
if ULStat <> 0 then exit;
```

Figur 10 - Delphi kode, konfigurasjon av feilhåndtering

Konfigurasjon og initialisering av de digitale I/O portene

PMD-1208LS har to digitale I/O porter. Disse kan bli konfigurert som enten inn eller ut. Hvis man mot formodning velger å rekonfigurere en av portene til å ha den motsatte retningen av hva den opprinnelig ble konfigurert som, vil eventuelle verdier som den andre porten har bli satt til null. Dette er et stort poeng hvis man for eksempel har planer om å lage en 2-veis kommuniserende buss. Dette ble prøvd i denne oppgaven, men ble da umulig å implementere. Til opplysning så er det ikke alle av Measurement Computing sine bokser som oppfører seg på denne måten, en del av de greier å håndtere at portene blir rekonfigureret. Den slags informasjon står således i de respektive databladene.

Det som ble gjort for å konfigurere de digitale I/O portene er vist i kodesnutten under i Figur 11. Funksjonen *cbDConfigPort* tar brettnummeret, porttype og portretning som parametere. Brettnummer er en integer som i dette tilfellet satt til 0. Porttype er en integer[17] som enten kan være *FirstPortA* eller *FirstPortB*. I dette tilfellet er det satt til FirstPortA, men FirstPortB vil bli benyttet for den andre porten som skal brukes. Videre er parameteren *PortDirectionOut*[18] definert som integer og kan ta på seg verdiene *DIGITALIN* eller *DIGITALOUT*.

```
{konfigurere PORTA for utgang på brett #1}
ULStat := cbDConfigPort (BoardNumA, PortTypeA, PortDirectionOut);
if ULStat <> 0 then exit;
```

Figur 11 - Delphi kode, konfigurering av digital I/O porter

Det er alltid greit å initialisere en port. For å gjøre dette benyttes koden vist under i Figur 12. De to første parametrene i funksjonen *cbDOut* er de samme som benyttet for

[17] Funksjonen krever integer på denne parameter plassen. FirstPortA og FirstPortB er bare tall som er definert i form av ord slik at det blir lettere for brukeren å benytte seg av det.

[18] *PortDirectionOut* er brukt som navnet på parameteren, men bare *PortDirection* blir brukt i funksjonsbiblioteket. Så *Out* delen på slutten av den parameteren som er brukt har ingenting å si, kun et navn.

cbDConfigPort vist ovenfor i Figur 11. *DataValue* er definert som typen *Word*, som i Delphi er definert som 16bit. Siden det her snakkes om en fysisk 8 bits port så er det anbefalt å holde seg innenfor dette.

```
{initialisere port verdi}
DataValue := 0;
ULStat := cbDOut (BoardNumA, PortTypeA, DataValue);
if ULStat <> 0 then exit
```

Figur 12 - Delphi kode, initialisering av digital I/O port

Konfigurasjon av den analoge porten

Grunnen til at den analoge innporten blir benyttet i denne oppgaven er fordi PMD-1208LS boksen ikke har nok digitale porter. Det er nødvendig med kommunikasjon tilbake til USB boksen fra mikrokontrolleren. Siden det, som nevnt tidligere, ble problematisk å omdefinere retningen på de digitale portene til innganger etter at de allerede hadde blitt definert som utganger, var benyttelsen av den analoge porten vital.

PMD-1208LS tilbyr to forskjellige konfigurasjonsmåter for de analoge inngangene. Det kan enten settes opp for *differential mode* eller *single-ended mode*. I denne oppgaven ble single-ended mode brukt. Man skal være oppmerksom på at ved denne konfigurasjonen fungere det kun å bruke signaler som spenner mellom ±10 volt.

Tre av de inngående analoge portene på den eksterne I/O boksen ble benyttet. Disse tre var *CH0IN*, *CH1IN* og *CH2IN* plassert respektivt på de fysiske pinene 1, 2 og 3, se Figur 8 for utfyllende informasjon. Siden de analoge inngangene i prinssippet skulle benyttes akkurat som en digitalinngangsport ble det laget en funksjon ved navn *checkAnalogInput* som vist under i Figur 13. Den tok parameteren *Channel* i kallet. Channel er da en integer som forteller hvilken inngang det er snakk om, enten CH0IN, CH1IN eller CH2IN. Hvis inngangen er høy vil funksjonen returnere *True*, hvis ikke vil *False* bli returnert. For å sjekke selve verdien som ligger på pinen ble funksjonen *cbAIn* fra funksjonsbiblioteket benyttet. Denne funksjonen trenger brettnummer, kanal og range for å returnere dataverdien som ligger på den respektive pinen gitt av kanal parameteren. Range er måleområdet til den analoge til digitale omvandleren og ble her satt til *BIP10VOLTS*. Med andre ord vil dette tilsvare et spenn på ±10volt.

```
{funksjon som sjekker om det som kommer inn på analoginputen er høy eller lav}
{returnere True hvis den er høy og False hvis den er lav}
function TfrmStepper.checkAnalogInput(Channel: Integer): Boolean;
var
  ChannelTemp:    Integer;
  Temp:           Boolean;
begin
  Temp := True;
  ChannelTemp := Channel;
  ULStat := cbAIn(BoardNumA, ChannelTemp, Range, DataValue); if ULStat <> 0
then exit;
  if DataValue >= 2500 then
  begin
    Temp := True;
  end;
  if DataValue < 2500 then
  begin
    Temp := False;
  end;
  Result := Temp;
end;
```

Figur 13 - Delphi kode, analog inngangsfunksjon

Kodesnutten ovenfor i Figur 13 gjør slik at hvis signalet inn har en verdi på over eller lik 2500 vil signalet inn bli tatt som høy, dette vil da tilsvare over eller lik 2,5volt. Hvilken verdi som trengtes å bli benyttet som grenseverdi ble sjekket i labben siden dette ikke var tilstrekkelig dokumentert. Hvis pinen blir koblet til jord, gir funksjonen tilbake en verdi på cirka 2050. Kobles pinen til +5 volt gir den cirka 3080 tilbake. 2500 vil ligge midt mellom disse tallene å benyttes derfor for å skille et logisk høyt fra et lavt signal.

Mikrokontrolleren

Hvorfor mikrokontroller?

Siden *Measurement Computing* sin USB boks ikke greier å oppnå en høyere frekvens på pulstogene enn maksimalt 70Hz, måtte det benyttes en annen form for pulsgenerator. Det ble tenkt på litt forskjellige løsninger blant annet å bruke en eller annen form for analog krets, som en *Schmidt Trigger* eller liknende form for oscillator. Under testing i labben av posisjoneringsriggen ble det erfart at motoren

ikke taklet å få en steppingsfrekvens på mer enn noen hundre hertz uten at en form for akselerasjon av frekvensen var tilstede. Om dette var strømforsyningen som ble brukt eller motoren vites ikke, men det ble konkludert med at det var nødvendig å få en form for akselerasjon for å få god drift av motoren. Problemet gjorde seg gjeldende både ved bruk av vanlig likestrømskilde og ved tre seriekoblede batterier[19]. Det ble da ansett nødvendig å få anskaffet en akselererende funksjonsgenerator.

Kurset "Fys4240 PC-basert instrumentering og mikrokontrollere" som holdes av Ørjan G. Martinsen ved Fysisk Institutt tar for seg blant annet programmering av mikrokontrollere i C. Dette kurset har vært en del av det teoretiske pensumet for denne masteroppgaven, det ble derfor sett på som en overkommelig oppgave å få programmert en mikrokontroller til å fungere som en akselererende funksjonsgenerator. Det ble tatt utgangspunkt i et liknende oppsett som ble brukt i Fys 4240 kurset.

Atmel STK500 utviklingssett

Atmel STK500 settet er en fin begynnelse for å komme i gang med programmering av Atmel sine *flash* baserte mikrokontrollere. Det inneholder blant annet selve Atmel STK500 utviklingskortet, programvaren AVR Studio 4 og to mikrokontrollere, deriblant en ATMega16. Utviklingskortet passer for en mengde forskjellige typer mikrokontrollere fra Atmel, og det finnes ekspansjonskort for videre å åpne mulighetene. Se Figur 14 for detaljer.

[19] Tre seriekoblede 12 volts batterier var det som var planlagt å bli benyttet ved feltarbeid.

Figur 14 - Atmel STK500 utviklingssett

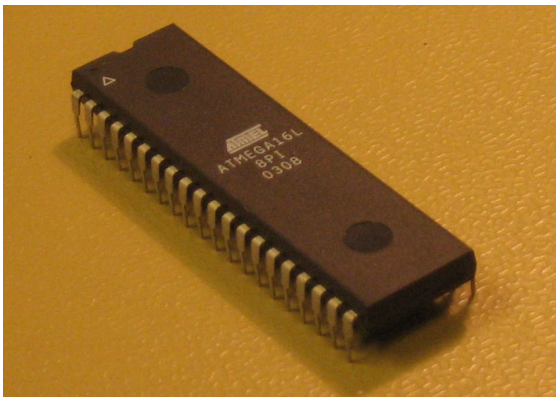
Et utviklingskort slik som Stk500 gjør det greit å komme i gang da man kjapt har tilgang til selve programmeringsdelen av mikrokontrolleren og de forskjellige portene for testing. Det er også inkorporert 8 trykknapper og 8 lysdioder for testing. I tillegg er det inkludert programvaren AVR Studio 4 for kode skrivning[20] og feilsøking, samt for kommunikasjon til kortet via serieporten for programmering av kontrolleren. Dette er relativt greit å sette opp, men det er absolutt anbefalt å lese bruksanvisningen [2] for kortet for å sette opp programmeringsdelen riktig samt for å få innblikk i de mange mulighetene utviklingskortet gir. Spesielt er det viktig at man får satt jumperene og diverse kabler opp riktig for den programmeringsmetoden som er ønskelig, eller så blir det fort krøll. I denne oppgaven ble det benyttet "In system programming" [2] (side 3-7).

ATMega16 mikrokontroller

Atmel sin ATMega16 mikrokontroll [3], se Figur 15 for bilde, er en av de som fulgte med kortet. Siden erfaring fra programmeringa av en ATMega32 er ervervet fra tidligere var det naturlig å benytte en tilsvarende kontroller, noe som ATMega16 er. Den er en mikrokontroller med 16Kbytes programminne, 1Kbyte SRAM innebygd og en 10 bits analog til digital omvandler. Videre har den fire 8 bits porter konfigurerbare

[20] Til kodeskrivningen ble "Programmers Notepad" brukt, men til *flashing* av mikrokontrolleren ble AVR Studio benyttet.

til inn eller ut. Disse portene er egentlig alt denne oppgaven trenger å ha tilgang til for å få styringssystemet til og fungerer slik som ønsket. Siden denne mikrokontrolleren allerede følger med STK500 pakka ble det ikke sett på som økonomisk fornuftig å finne en annen kontroller som kun hadde det man absolutt trengte av porter.



Figur 15 - Atmel ATMega16 Mikrokontroller

Programmeringsspråk og -verktøy

Når man skal programmere en mikrokontroller så er det vanlig å enten å benytte seg av Assembler språk eller C. I denne oppgaven er det valgt C, dette er fordi dette er et språk som er kjent fra før.

Programmers Notepad[21] er et gratis program for kodeskriving og kompilering av deriblant C-kode, slik som benyttet her. AVR Studio 4 kunne ha vært benyttet til dette, men tidligere erfaring har vist at Programmers Notepad er et veldig lett oversiktlig program spesielt når det gjelder skriving av løkker. Man har muligheten for å fjerne løkker, prosedyrer og funksjoner visuelt slik at man kun sitter igjen med navnet på metodene[22] og dets parametere, dette øker klart oversikteligheten i kodingen.

[21] Dokumentasjon av program finnes på URL http://www.pnotepad.org/ [hentet 30. nov. 2005] , samt nedlastningsmuligheter.

[22] Med metode i *dette* tilfellet menes det prosedyrer, funksjoner og løkker.

For å få kompilert C-koden i Programmers Notepad trengs det to metoder som brukeren må sette opp, nemlig *Make All* som kompilerer selve koden, og *Make Extcoff* som genererer en *Hex-fil* som benyttes senere til selve programmeringa av mikrokontrolleren i AVR Studio 4. For å kunne sette opp og benytte seg av disse metodene trengs det en *Makefile* [CD:\Kildekode\Makefile.txt]. Denne forteller kompilatoren alt den trenger å vite for å få kompilert.

Programmering av mikrokontrolleren

Programmering av mikrokontrollere i C er en veldig grei oppgave når man først har fått draget på et par små ting. Selvfølgelig må syntaksen i C være kjent og for å friske litt opp på dette kom boken ”C Pocket Reference” av Peter Prinz og Ulla Kirch-Prinz godt inn [14].

Når man skal programmere en mikrokontroller må man først sette seg litt generelt inn i databladet for kontrolleren. På denne måten får man et lite overblikk over funksjonalitetene som mikrokontrolleren har, og som kan benyttes. I denne oppgaven blir det benyttet avbruddsrutiner og inn og ut portene som ligger i Atmega16 kontrolleren. For å få dette til å fungere så må de riktige bibliotekene inkluderes i koden, diverse register må konfigureres og avbruddsrutiner må skrives. Det vil her bli gått imellom det som måtte settes for å få de enkelte elementene til å fungere i oppgaven. I *implementasjons* delen av oppgaven vil det bli gitt videre opplysninger om hvordan de forskjellige elementene ble koblet sammen for å fungere tilfredsstillende.

Inkludering av biblioteker

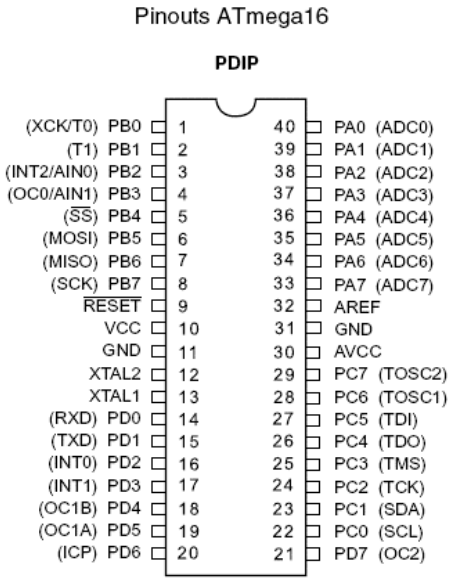
Med AVR Studio 4 fåes det også med en del biblioteker for selve programmeringen. Dette er alt fra helt standard funksjoner til forsinkelsesløkker samt definisjoner av porter etc. For å kunne bruke de i programmene så må dette inkluderes i C koden. For denne oppgaven gjelder det å få inkludert de bibliotekene som tar seg av avbrudd og generell I/O. Bibliotekene blir da inkludert som vist under i Figur 16.

```
#include <avr/io.h>
#include <avr/delay.h>
#include <avr/interrupt.h>
#include <avr/signal.h>
```

Figur 16 - C kode, inkludering av biblioteker

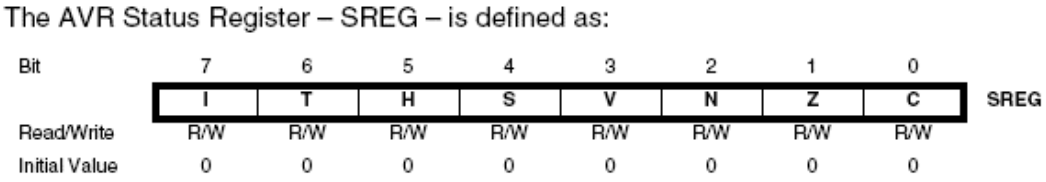
Konfigurering av registre

Det som kanskje kan virke litt uoverkommelig og uoversiktelig i begynnelsen ved programmering av mikrokontrollere, er konfigurering av registrene innad i mikrokontrolleren for å oppnå den driften som er ønskelig. Registrene forteller mikrokontrolleren hvilke funksjoner som skal brukes/aktiveres og hvordan disse eventuelt skal brukes. Hvordan man skriver til disse registrene og aktiv bruk av dem er en vital del i forståelsen av mikrokontrolleren, og drifting av den. Uten denne innsikten er det umulig å få til noe som helst med en mikrokontroller. Det blei derfor tilbrakt en del timer over datablader og gamle programmer for igjen å oppnå innsikten.



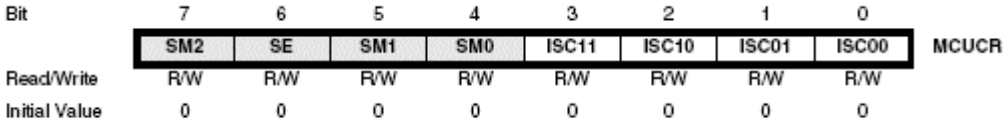
Figur 17 - Atmega16 pinnekonfigurasjon

Før vi går videre til avbrudd[23] og håndtering av de, la oss ta for oss et lite eksempel[24] på setting av registre for eksterne avbrudd[25]. La oss først finne ut hvilke registre som trenger eksplisitt konfigurasjon. Alt dette er oppnåelig informasjon fra databladet, men la oss ta en titt allikevel. Når man skal åpne for generering av eksterne avbrudd må man først sette *Status Register*(SREG) sitt *Global Interrupt Enable* bit, også kalt *I-bit*, høyt- se Figur 18 under. Det vi nå har gjort er å ha åpnet for bruk av avbrudd generelt sett, derav navnet globale avbrudd.



Figur 18 - Atmega16 Status register (SREG)

Videre må vi bestemme oss for hvordan vi vil at de eksterne avbruddene skal operere innad i systemet. I denne oppgaven blir det benyttet det eksterne avbruddet INT1 som genereres på pin17. Her kunne vi likeså ha benyttet INT0 eller INT2. *MCU Control Register*(MCUCR) gir oss mulighetene for å justere hvordan de eksterne avbruddene INT0 og INT1 skal trigges. Bitene og dets plassering i MCUCR er gitt i Figur 19. I alle registrene er det mange bit man kan sette eller manipulere med, her kommer vi derimot kun til å gå imellom de som er direkte relatert til oppgaven.



Figur 19 - Atmega16 MCU Control Register(MCUCR)

[23] Hvis leseren ikke har noen forhåndskunnskaper om avbrudd så anbefales det å hoppe videre til avbruddsdelen for så å gå tilbake å lese eksemplet med ny kunnskap innabords.

[24] Eksempel basert på Atmega16 mikrokontroller og dets konfigurasjon.

[25] Tre eksterne avbrudd er tilgjengelig: INT0, INT1 og INT2 på de respektive pinene 16, 17 og 3, se Figur 17 for pinnekonfigurasjonene til Atmega 16.

Ovenfor i Figur 19 er det kun de fire siste bitene som vi trenger å tenke på. Ser vi videre i Figur 20 ser vi de mulige valgene av bitene ISC11 og ISC10 i MCUCR. Bitene ISC01 og ISC00 vil sette akkurat det samme utfallet som ISC11 og ISC10 gjør, men for det eksterne avbruddet INT0 i stedet. Siden vi ikke skal benytte oss av INT0 trenger vi heller ikke tenke på dette. Det var ønskelig å få generert et avbrudd på stigende flanke av INT1. For å oppnå dette må vi sette både ISC11 og ISC10 høyt.

Interrupt 1 Sense Control

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

Figur 20 - INT1 avbruddsparametere

Det neste registeret som har med eksterne avbrudd å gjøre vil være *MCU Control and Status Register*(MCUCSR). Dette registeret tar hånd om eksterne avbrudd på INT2, men siden dette ikke skal benyttes trenger vi ikke tenkte mer på dette.

Det neste registeret er *Generell Interrupt Control Register*(GICR). Dette er i prinsippet et *enable* register, her setter vi hvilket av de eksterne avbruddene som mikrokontrolleren skal reagere på. Se nedenfor i Figur 21 for info angående bitenes plassering i registeret.

General Interrupt Control Register – GICR

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figur 21 - Atmega16 Global Interrupt Control Register (GICR)

Det er bitene INT1, INT0 og INT2 som vi trenger å se på. INT1 bitet står her for *External Interrupt Request1 Enable* [3][26] således er INT0 og INT2 enable bitene for de respektive eksterne avbruddene. Siden vi skal benytte hos av INT1, vil det også være dette bitet som må settes høyt. De andre holdes lave.

Det siste registret som brukes i håndtering av eksterne avbrudd vil være *General Interrupt Flag Register*(GIFR). Dette registeret benyttes til å holde styr på avbruddene for mikrokontrolleren, men siden dette ikke er noe som aktivt blir satt i denne oppgaven henvises leseren til databladet for mer inngående detaljer.

Nå har alle registre blitt sett på og verdiene til de forskjellige bitene har blitt bestemt, det eneste som gjenstår er å sette dette i C. Vi vil da for eksempel skrive følgende kode som vist i Figur 22 under.

```
//Setting av Status Register se side7 i ATmega16(L) datablad
SREG = 0x80;

//Rising Edge på eksternt avbrudd 1
//Setting av MCU Control Register se side 66-68 i ATmega16(L)
//datablad
MCUCR = (1<<ISC11)|(1<<ISC10)| (0<<ISC11)|(0<<ISC10);;

//Setting av MCU Control and Status Register se side 66-68 i
//ATmega16(L) datablad
//Her trengs faktisk ingen ting å settes
MCUCSR = 0x00;

//Setting av General Interrupt Control Register se side 66-68 i
//ATmega16(L) datablad
GICR = (1<<INT1)|(0<<INT0)|(0<<INT2);
```

Figur 22 - C kode for setting av avbruddsregistre

Ser vi litt nøyere på denne koden, ser vi på andre linje følgende: SREG = 0x80, siden det var I-bitet eller bit7 som skulle settes høyt her, skulle vi egentlig ha skrevet vanlig kode for *bit shifting* SREG = (1<<I). Dette lar seg derimot ikke gjøre av en eller annen bisarr grunn[27], derfor blir det mest signifikante bitet satt høyt ved å bruke hex-

[26] Side 67-68.

[27] Det kan hende *I* er reservert enten av C, eller AVR slik at vi ikke får brukt det.

koden 0x80 som vil være det samme som å sette det binære 8bits koden fra mest signifikant til minst signifikant: 10000000.

Konfigurasjon av I/O porter

For å få I/O portene på mikrokontrolleren til å fungere slik som ønskelig må også registre settes for dette. Selve settingen av registrene er en grei jobb, men håndtering av inn og utsignaler viste seg å ikke være like lett som antatt. Med litt famling i datablader og kompilatornotater samt konstruktiv bruk av internett ble det også funnet løsninger på dette.

Siden alle portene[28] på Atmega16 er porter som er definerbare som ut- eller innporter må det konfigureres registre for håndtering av dette. Hver port har et register som heter *Port x Data Direction Register(DDRx)* hvor x er den respektive porten. For PortA vil det da bli betegnet *DDRA*. Hvis registret settes høyt vil porten bli en utgående port, og hvis det settes lavt blir det en inngående port. I denne oppgaven blir alle portene benyttet. PortA brukes som inngang for overførsel av informasjon om hvor langt hydrofonen skal beveges i manuell modus. PortB benytter bare den ene pinen. Denne blir således benyttet til selve pulstoget som sendes til motorstyringskretsen. PortC anvendes også til utgang, men til kontrollsignaler tilbake til PMD-1208LS og datamaskinen. Kontrollsignaler inn til mikrokontrolleren fra PMD-1208LS og datamaskinen blir styrt inn på PortD. Se Figur 23 for detaljer for kode.

```
//PORTA settes som inngang
DDRA = 0x00;

//PORTB som utgang
DDRB = 0xFF;
```

Figur 23 - C-kode for setting av PORT konfigurasjon

[28] PortA, PortB, PortC og PortD

Registrene er nå satt for riktig konfigurasjon for inn- eller utsignaler. For å sette ut signaler ut eller for å lese inn må man benytte seg respektivt av PINA eller PORTA for port A. Det samme vil gjelde for de andre portene. For å kunne skrive ut hex-tallet 0x00 på port B, slik som vi blant annet gjør i denne oppgaven, må det skrives følgende enkle C kode linje som vist i Figur 24.

```
//Port B setter ut hex-tallet 0x00
PORTB = 0x00;
```

Figur 24 - C-kode, utsetting av verdi på port

For så å skrive ut på en av portene så var det nærliggende å tro at man kunne bruke samme kode som i Figur 24, men at porten var konfigurert for inngang. Dette viste seg å ikke være tilfellet. I stedet må man benytte seg av koden PINx[29]. Skal man så lese inn hva som er av verdier på port D, må man først definere en variabel som man lagrer det i, for så å sette det som ligger på porten til denne variabelen, se Figur 25.

```
unsigned char data;
data = PIND;
```

Figur 25 - C-kode, setting av verdi inn på port

I denne oppgaven er det nødvendig å lese spesifikke signaler som kommer inn på porten altså man trenger ikke å vite alt som ligger på porten, men kun noen av pinene. For å gjøre dette må man bruke en heller spesiell kode[30], se Figur 26.

```
unsigned char data;
data = (PIND & (1<<PD0));// & er bitand
```

Figur 26 - C-kode, lesing av en spesifikk pinne på inn port

PD0 i koden i Figur 26 referere til selve pin0 på port D, med andre ord den fysiske pin-14, se Figur 17 for detaljer. Det som blir gjort her er at vi tar det som ligger inn på port D, og bit skifter logisk høy PD0 ganger til venstre. Vi tar så en *bitand* [14]

[29] Hvor x representerer A, B, C eller D.

[30] Takker forumet på http://www.avrfreaks.org for denne kode snutten.

operasjon på disse to, slik at hvis det signalet som ligger inn på PD0 pinen er høy vil *data* variabelen i koden bli satt høy, er derimot signalet lavt vil data bli satt lavt. Dette er komplisert kode og vanskelig å holde styr på i hodet, men det viser seg å fungere.

Avbrudd og avbruddshåndtering

Avbrudd er lett sagt en måte å få programmet til å hoppe til en annen programsnutt å kjøre dette hver gang noe spesielt skjer, for så å gå tilbake til den opprinnelige[31] plassen etterpå. Det er definert prioritert for de forskjellige avbruddene i mikrokontrolleren, for eksempel er *Reset* det avbruddet med høyest prioritet. Dette vil si at uansett hva mikrokontrolleren gjør så vil den hoppe ut av det hvis reset blir kalt. Avbrudd og avbruddshåndtering er en viktig del av mikrokontrolleren. Man kan få gjort litt uten dette, men de fleste funksjoner vil være utenfor rekkevidde uten inngående kjennskap til avbrudd og dets håndtering. Det som er spesielt ved programmering av avbrudd er at hvordan dette gjøres er kompilator spesifikt.

Det ble gjennomgått hvordan man gjør mikrokontrolleren klar for eksterne avbrudd i *Konfigurering av registre* tidligere i oppgaven. Nå vil det bli sett spesifikt på selve håndteringen av avbruddene. Hvis vi vil at mikrokontrolleren skal reagere når en spesifikk avbruddssignal, også kalt avbruddsvektor, går høy, må mikrokontrollerne også kodes deretter. Når registrene er konfigurert slik som det har blitt gjort tidligere, vil det bli generert en avbruddsvektor når et signal går høyt på INT1, fysisk pin-17, se Figur 17. Denne vektoren blir kalt *SIG_INTERRUPT1*. For at mikrokontrolleren skal kunne reagere på dette trengs det følgende kode snutt som vist under i Figur 27.

```
SIGNAL(SIG_INTERRUPT1)
{
 //evt. kode for hva som skal skje når INT1 går høy
}
```

Figur 27 - C-kode, avbruddshåndtering

[31] Det kan ikke garanteres at man kommer akkurat tilbake ditt man var. Dette må sjekkes eksplisitt i databladet. Beste måte er å passe på at et ting alltid lar seg gjøre seg ferdig før et avbrudd evt. blir generert.

Selve koden *SIGNAL(...)* vil da være kompilator spesifikt, mens SIG_INTERRUPT1 vil være det samme så lenge AVR sine biblioteker er inkludert.

Akselererende funksjonsgenerator

Grunnen til at mikrokontrolleren ble i første omgang tatt i bruk var fordi PMD-1208LS boksen ikke greide å generere kjapp nok frekvens til steppermotorens styringskrets. Den siste biten som trengs å gå imellom når det gjelder mikrokontrolleren vil da være den akselererende funksjonsgeneratoren, før selve implementasjonen blir tatt opp.

For å konstruere en akselererende funksjonsgenerator ble det først laget en funksjonsgenerator prosedyre som mottok to parametere i kallet. Den første var av typen *integer*, og bestemte antall pulser som skulle genereres i pulstoget. Neste parameter var også av typen integer, denne skulle derimot bestemme lengden på hver enkelt puls slik at vi kunne generere den ønskede frekvensen på pulstoget. Respektivt ble disse kalt *teller* og *delay_constant*. Se Figur 28 under for C kode.

```
void funksjonsgenerator(int teller,int delay_konstant)
{
 for(int i = 0; i < teller; i++)
 {
  PORTB = 0x01;
        _delay_loop_2(delay_konstant);
        PORTB = 0x00;
        _delay_loop_2(delay_konstant);
 }
}
```

Figur 28 - C kode, funksjonsgenerator

Det blir kalt en prosedyre *_delay_loop_2* i koden gitt i Figur 28, dette er en forsinkelsesløkke som er innebygd i AVR biblioteket som gir en viss forsinkelse gitt av parameteren som vi sender med. Verdiene som trengtes å benyttes her ble funnet på den gode gamle prøv og feil metoden, slik at det ble testet med forskjellige verdier for å oppnå riktig timing.

Med dette ble det oppnådd å ha produsert en firkantpuls generator, hvor man kunne spesifisere antall pulser som skulle bli generert og også hvilken frekvens som skulle genereres. Frekvensen som ble generert ved forskjellige delay_constant verdier måtte sjekkes manuelt i labben. Det viste seg at med en delay_constant verdi lik 1000, ble det generert en frekvens tilsvarene ca 125Hz. Videre med delay_constant lik 100 fikk vi 1,25kHz osv. På denne måten var det greit å regne ut cirka hvilken frekvens som ble generert ved forskjellige delay_constant verdier. Nå var det ikke noe mål i seg selv å måte vite nøyaktig hva frekvensen var så vi lot det ligge ved det.

For å overkomme problemet som hadde oppstått ved å sette en høy frekvens rett på motorens styringskrets ble bare funksjonsgenerator prosedyren kalt flere ganger med forskjellige verdier for parametrene. Etter hvert som frekvensen økte, ble teller variabelen økt slik at det ble flere pulser. Dette ble gjort for å få en mer mykere akselerasjon. På denne måten ble det skapt en akselererende funksjonsgenerator. Det totale spennet på generatoren ble fra 125Hz til cirka 1,85 kHz.

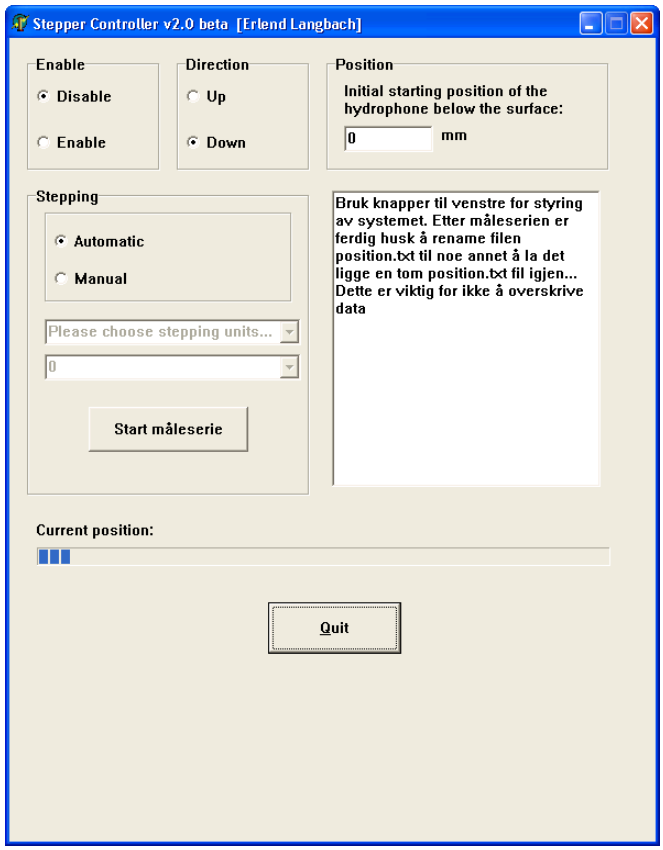
På samme måte som for akselerasjonen ble en deakselerasjons prosedyre laget, slik at motoren fikk en rolig nedbremsning.

Grafisk brukergrensesnitt

For å kunne gi brukeren av systemet et greit brukergrensesnitt å forholde seg til, var programmeringen av dette også en viktig del av oppgaven. Det er mange programmeringsspråk som lar seg anvende til dette. Programmeringserfaring som var oppnådd i koding av grafisk grensesnitt fra universitetet var kun i form av språket *Java*. Siden Measurement Computing sin eksterne USB boks ikke støtter dette språket måtte det taes fatt på oppgaven med å sette seg inn i noe nytt. Det falt seg naturlig å velge *Delphi*[32], grunnen til dette er flere. En faktor var at en av mine veiledere, Helge Balk, bruker dette språket til daglig slik at hjelp, om nødvendig, bestandig ville være i nærheten. Den aller viktigste grunnen til valget var derimot at Delphi gir

[32] Borland Delphi 6.0

programmereren en veldig kjapp måte å lett generere intuitive grafiske grensesnitt på. Figur 29 gir en liten smakebit på hvordan det ferdig utviklede programvarens grensesnitt ser ut.



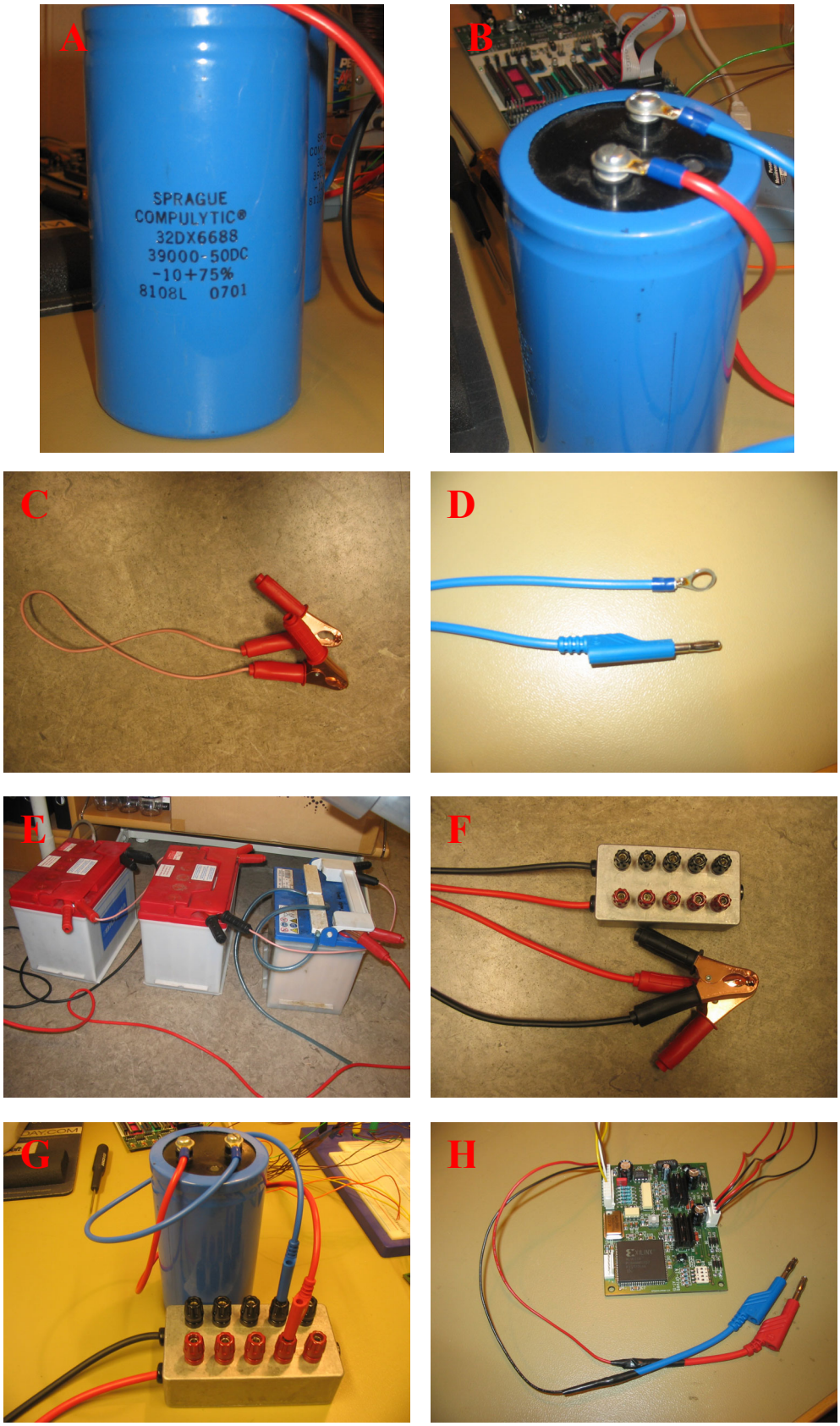
Figur 29 – Grafisk grensesnitt over egenutviklet styringsprogramvare

Spenningskilde og kabler

For å kunne gi en tilfredsstillende spenning til motorens driverkort, ble det valgt å benytte tre seriekoblede 12 volts bilbatterier og en stor kondensator i parallell til dette så nærme styringskretsen som mulig. To av bilbatteriene er fra samme innkjøp, det andre er av eldre dato. Under utviklingen av utstyret ble det observert at motoren kunne ha en tendens til å fuske litt, det ble på daværende tidspunktet tenkt at dette var relatert til en kombinasjon av ikke fullstendige oppladete batterier, og at ikke kondensatoren var tilstede i oppsettet. Denne ble nemlig ikke inkludert før siste testkjøringen, se mer om dette i kapittelet *Testkjøring og Resultater*.

Kondensatoren som ble benyttet var av typen *Sprague Compulytic 32DX6688 39000µF 50 volt*. Denne er ikke av det nyeste kaliberet, det ble derfor sjekket at den fungerte tilfredsstillende ved å kjøre en liten ladestrøm for så å se hvor bra den holdt spenningen. Den fungerte utmerket til dette.

I Figur 30 på neste side er det en bildeserie av utstyret benyttet rundt strømleveringen til oppsettet. A gir bildet av selve kondensatoren. Bilde B og D illustrerer kabelen som ble laget for tilkobling av kondensatoren i oppsettet slik at det var lett å koble den tilboksen som vist i G. Det ble videre loddet på kontakter på driverkortet slik som vist i H, slik at denne også kunne kobles til kontaktbrettet vist i G. C viser kabelen som ble benyttet til å seriekoble batteriene vist i E.



Figur 30 – A: Kondensatoren, B og D: Tilkobling av kondensatoren, C: Kabeltype benyttet til å seriekoble bilbatteriene, E: Bilbatteriene, F: Tilkoblingsboks som ble benyttet til å koble utstyret sammen, G: Kondensatoren tilkoblet, H: Spenningskabler til styrekort.

Implementasjon

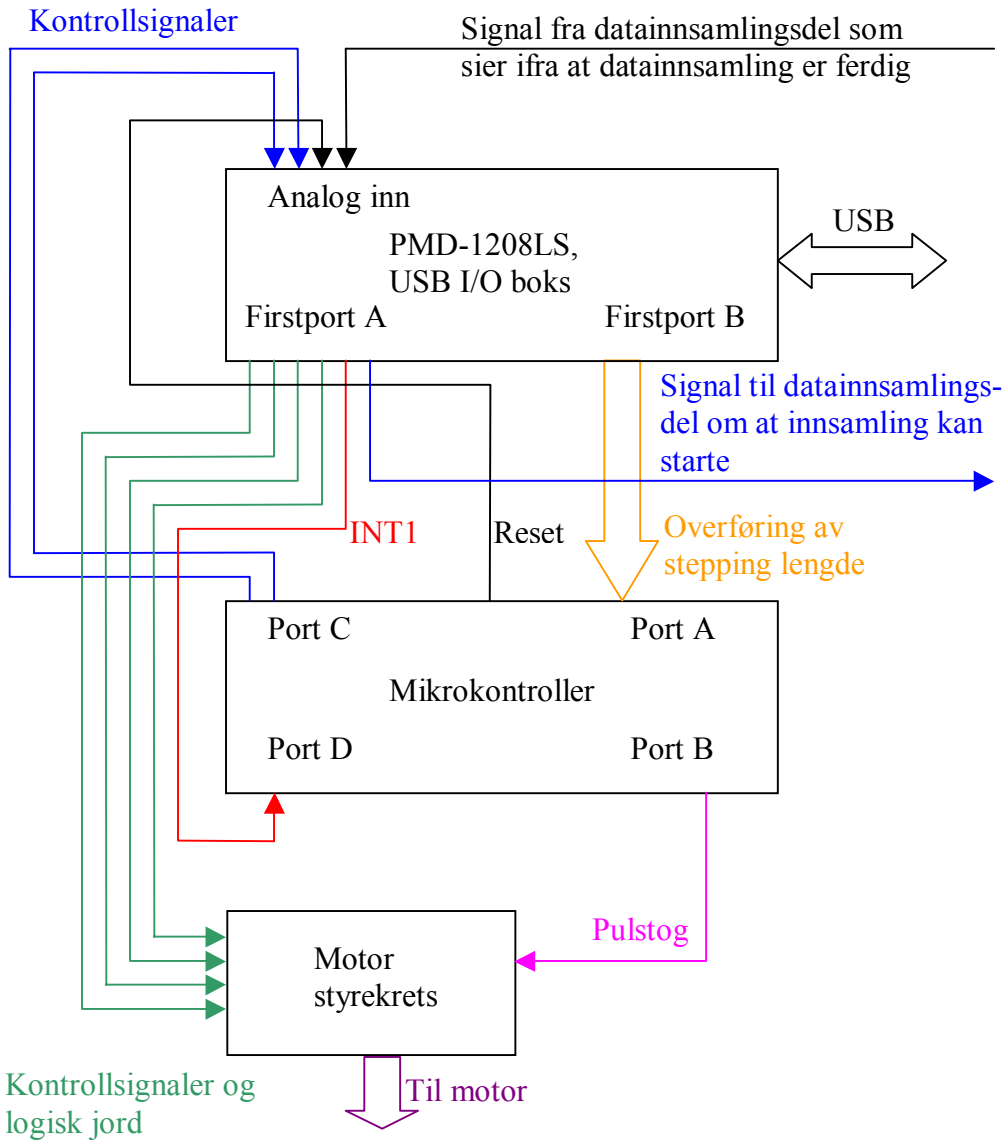
I de to foregående seksjonen ble det gitt et generelt overblikk over implementasjonen, og kjennskap til de delene som er nødvendige for sammenkjøring av posisjoneringsriggen. I denne delen vil det bli gitt en dyptgående gjennomgang av implementasjon samt dokumentasjon av den egenutviklet koden for både det grafiske grensesnittet og mikrokontroll.

Når man skal implementere et slikt system som i denne oppgaven, er det en del ting å ta hensyn til. Først og fremst må det genereres de signalene som de forskjellige elementene krever og trenger. Uten disse vil ikke systemet fungere tilfredsstillende. I tilegg er det til alles beste å kunne fremskaffe et brukergrensesnitt som både er lett å betjene, men også som har innebygd en del sikkerhetsfunksjoner. Disse sikkerhetstiltakene er der først og fremst for å sikre at avstandmålingen blir så nøyaktige som mulig. Slik at man oppnår det var utgangspunktet for å bygge om posisjoneringsstativet, nemlig å kunne ha muligheten til nøyaktig kartlegging av lydfeltet og dets utbredelse på grunt vann.

For at leseren skal ha muligheten til å følge oppgaveskriverens tankerekke, velges det å begynne med de logiske signalene som går mellom de forskjellige fysiske delene av oppsettet for så å bevege seg videre til selve programstrukturen.

Logiske signaler

Her vil det bli gått igjennom hvordan alle signalene ble rutet mellom de forskjellige fysiske boksene. Det vil da bli gitt den nødvendige informasjonen slik som fysisk pinnummer og pinnavn for at det er mulig å få koble dette greit opp i ettertid. Se under i Figur 31 for oversikt over hvordan den fysiske signalflyten fra og til de forskjellige fysiske elementene i oppsettet ble konfigurert. I Figur 31 vil det være spesifisert hvilken retning signalene har, dette vil det ikke bli gjort under de forskjellige delen i denne seksjonen. Så man må kryssreferere med Figur 31 for å få den totale oversikten. Se også Figur 5, Figur 8 og Figur 17 for de respektive detaljene angående pinene på styrekrets, USB I/O boks og mikrokontrolleren.



Figur 31 - Oversikt over signalflyt mellom de forskjellige elementene

Signaler mellom USB I/O boks og motordriverkrets

Signalene som går fra PMD-1208LS til motordriverkortet er *logisk jord*, *enable* og *direction*. Siden den eksterne I/O boksen har uttak for logisk jord blir denne pinen brukt som logisk jord for hele systemet. Se Tabell 2 nedenfor for utfyllende info om relasjon mellom pine, pinenavn og pinenummer.

	PMD-1208LS pin#		Driverkrets pin#	
Signalets betydning:	Navn	Fysisk pin	Navn	Fysisk pin
Logisk jord	GND	31	IGND	3
Enable	PORT A0	21	ENB	6
Direction	PORT A1	22	DIR	4

Tabell 2 - Pinekonfigurasjon for signaler mellom PMD-1208LS og AT Drives

Signaler mellom USB I/O boks og mikrokontroller

Fra PMD-1208LS boksen vil det gå en 8 bits buss til mikrokontrolleren, denne vil inneholde informasjon om antall steg som skal steppes. Det vil videre være et signal som går fra USB boksen til mikrokontrolleren som står for generering av eksterne avbrudd. PMD-1208LS vil sende ut et kontrollsignal for valg av steppingmodus. Mikrokontrolleren vil sende to signaler tilbake til USB boksen slik at den kan holde kontroll og oversikt over hvor mange pulstog mikrokontrolleren har sendt til motordriverkretsen. Se Tabell 3 for utfyllende info om pinekonfigurasjon.

	PMD-1208LS pin#		Atmega16	
Signalets betydning:	Navn	Fysisk pin	Navn	Fysisk Pin
+5 volt	PC +5V	30	VCC	10
Logisk jord	GND	31	GND	11
Buss for overføring av antall steps	FIRSTPORTB		PORTA	
	Port B0	32	PA0	40
	Port B1	33	PA1	39
	Port B2	34	PA2	38
	Port B3	35	PA3	37
	Port B4	36	PA4	36
	Port B5	37	PA5	35
	Port B6	38	PA6	34
	Port B7	39	PA7	33
Eksternt avbrudd	Port A2	23	INT1	17
Kontrollsignal for valg av manuell eller automatisk modus	Port A4	25	PD0	14
Reset	CH0 IN	1	RESET	9
Helt ferdig stepping	CH1 IN	2	PC0	22
Ferdig ½-bølgelengde, benyttes kun i manuell modus	CH2 IN	4	PC1	23

Tabell 3 - Pinekonfigurasjon for signaler mellom PMD-1208LS og Atmega16

Signaler mellom mikrokontroller og motordriverkrets

Fra mikrokontrolleren til driverkretsen for motoren går det bare et signal nemlig selve pulstog signalet som gir motoren antall steg og hastigheten på disse, se Tabell 4 for pinekonfigurasjon.

	Atmega16		Fabr AT Drives	
Signalets betydning:	Navn	Fysisk pin	Navn	Fysisk pin
Pulstog signal	PB0	1	SCLK	5

Tabell 4 - Pinekonfigurasjon for signaler mellom Atmega16 og AT Drives

Signaler mellom USB I/O boks og eksternt måleutstyr

For å holde kontroll på eksternt målesystem ble det tatt høyde for å bruke et signal som sier ifra til måledelen at den skal starte måleserie. For å vite når den eksterne måledelen har gjort jobben sin, ble det også implementert å bruke et signal som måledelen genererer. Dette kan enten være et signal av typen *AD_conversion_finished* eller ett tilsvarende flagg eller signal. Måledelen skal gi dette signalet enten via en utgang eller at vi kobler oss inn for å "lytte" på signalet. Se Tabell 5 under for oversikt over signalene mellom USB I/O boksen og det eksterne målesystemet.

	PMD-1208LS		Eksterne målesystem	
Signalets betydning:	Navn:	Fysisk pin:	Navn:	Fysisk pin
Signal som sier ifra at måling kan foretas	Port A3	24	Begynn måling	ukjent
Signal som sier ifra at målingen er ferdig	CH3 IN	5	Måling ferdig	ukjent

Tabell 5 - Pinekonfigurasjon for eksternt målesystem

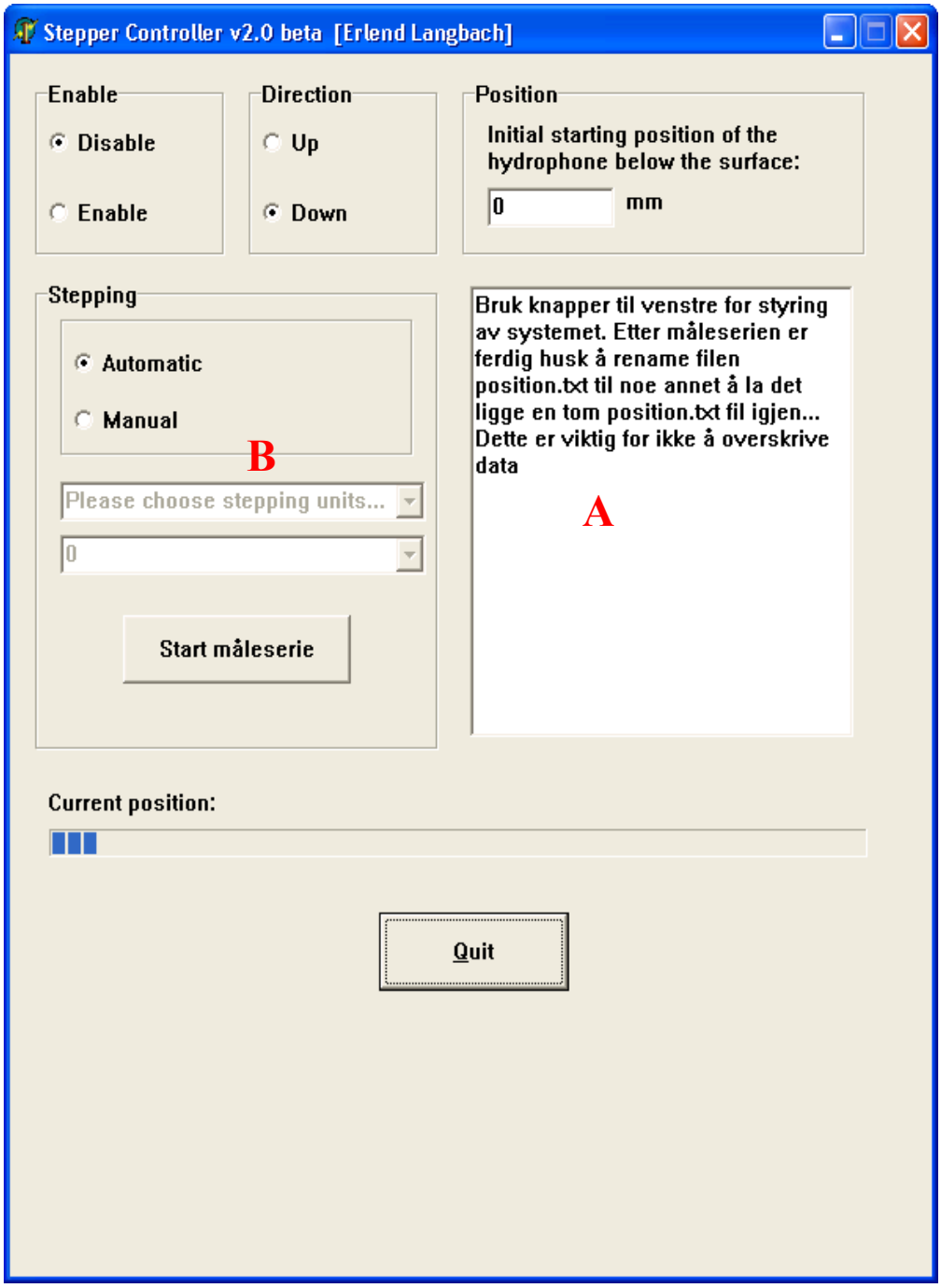
Programstruktur

Hvordan man skal programmere mikrokontrolleren i C, og hvordan man skal benytte seg av de rutinene som følger med Measurement Computing sitt funksjonsbibliotek ble gjennomgått under de respektive seksjonene *Programmering av mikrokontrolleren* og *Programmering av Delphi for operasjon med PMD-1208LS*. Her skal det derimot bli sett på hvordan selve programstrukturen og hvordan flyten i systemet er gjennomført. For å kunne tilegne seg implementasjonen greiest mulig begynner vi med å se på det grafiske grensesnittet som er utviklet og hvordan dette benyttes,

videre vil implementasjonen av mikrokontrolleren bli gått igjennom under *Programstrukturen i mikrokontrolleren.*

Programstrukturen i Delphi sammenkjørt med PMD-1208LS

Det egenutviklete programmet kodet i Delphi står for all kommunikasjon mellom brukeren og systemet. Det var derfor særdeles viktig at dette skulle fungere tilfredsstillende på en intuitiv og grei måte. Figur 32 under viser det grafiske brukergrensesnittet som ble utviklet. Alle de forskjellige elementene vil bli systematisk gjennomgått. For den fullstendige koden henvises det til appendiksen.



Figur 32 - Stepper Controller, grafisk brukergrensesnitt

Enable, Direction, infoboks og Quit

Enable er det første man trenger å sette for å få systemet til å fungere i det hele tatt. Hvis ikke denne er satt så kan man fortsatt benytte systemet, men motoren vil *ikke* forflytte seg. Dette skjer siden det å velge *Enable* eller *Disable* setter signalet ut fra pin 21 på PMD-1208LS og således inn på pin 6 på motordriverkortet til enten logisk høyt eller lavt, slik at man enabler kretsen eller ikke.

Med *Direction* radiogruppen kan man velge om stativet skal gå oppover eller nedover. Dette er en nyttig funksjon ikke bare ved manuell posisjonering, men også når man kjørerer automatiske måleserier, slik at etter en måleserie så kan man starte i motsatt retning slik at man sparer en del tid i felten.

Infoboksen, merket A i Figur 32, gir brukeren informasjon om hva man kan gjøre og hva man eventuelt må huske å gjøre. I dette tilfellet forteller den brukeren hvordan filen som holder oversikt over posisjons skal behandles for at det ikke skal mistes noe data. *Quit* knappen brukes enkelt og greit til å avslutte programmet.

Position, Current Position og filer til å holde kontroll på posisjonen

I *Position* boksen skrives det inn distansen mellom vannoverflaten og hydrofonen. Dette er for at programmet skal kunne estimere posisjonene til hydrofonen i vannlaget. Her må brukeren selv ta et valg om hvordan denne målingen mest hensiktsmessig skal taes. Dette benyttes kun til å estimere den faktiske distansen senere, i stedet for en relativ distanse.

For å holde rede på posisjonen til hydrofonen blir det i programmet benyttet to tekstfiler[33], *positionCheck.txt* og *position.txt* som skal ligge i samme mappe som selve programmet. Den første filen inneholder et tall mellom 0 og 250[34], dette tallet reflekterer antall halve bølgelengder[35] som hydrofonen har beveget seg vekk fra topp posisjonen. Grunnen til at 0 til 250 er valgt er at bussen som overførere antall halve bølgelengder som skal forflyttes er 8 bit. Bussen kan da maksimalt overføre tallet 255. Det kunne også blitt brukt seriell overføring, men siden 8 bit stemte fint overens med det maksimale heltallet som skulle overføres ble den parallelle overføringen benyttet.

[33] Fil av typen *.txt

[34] 6mm x 250 = 1500mm. Altså 1,5meter som er det totale intervallet som hydrofonfesteplata kan bevege seg. Se også fotnote 35.

[35] Halv bølgelgende tilsvarer cirka 6 millimeter.

Hver gang hydrofonen beveger seg en halv bølgelengde vil positionCheck.txt bli oppdatert enten med pluss eller minus en. Når filen har nådd enten 0 eller 250 vil programmet ikke tillate brukeren å forflytte hydrofonen lenger. Dette er for å sikre at motoren ikke prøver å komme utenfor det lovlige intervallet på 1,5 meter[36]. Topp posisjonen eller bunn posisjonen må settes manuelt. Dette gjøres ved å kjøre posisjonsriggen manuelt til den ønskede topp eller bunn plasseringen. Det må taes hensyn til at festeplatene skal kunne bevege seg 1,5 meter. Og deretter bytte ut hva enn som står i positionCheck.txt med 0 eller 250 for respektivt topp eller bunnposisjon. Når topp plasseringen blir satt vil programmet automatisk generere intervallet som er lovlig, og vica verca hvis bunn posisjon blir satt i stedet. Ved å bruke en fil til denne jobben vil det sikres mot eventuelle feil som oppstår, siden filen hele tiden blir oppdatert hver gang posisjonen blir forandret. På denne måten er det mulig å bruke systemet ved feltarbeid, og således benytte det ved en senere anledning uten å måtte sette topp eller bunn posisjon om igjen. Her ligger den en antagelse om at posisjonen til festeplatene til hydrofonen ikke blir forandret uten bruk av det grafiske grensesnittet. I prinsippet kan topp eller bunn posisjonene settes en gang i labben og da trengs den ikke å settes igjen.

Som nevnt ovenfor så er det en fil til, ved navn position.txt. Dette er filen som skal samkjøres med måledataen. Hver gang systemet sier ifra at måledelen kan gjøre målinger, vil posisjonen til hydrofonen bli skrevet til denne filen. På denne måten genereres det en tekstfil som det lett kan leses fra når man trenger posisjonsdataene. Det som er viktig her er at etter at en måleserie er kjørt må man kopiere filen og lagre den under et annet navn, for så å slette innholdet i den gamle filen. Hver gang man starter en måleserie ved en gitt distanse fra svingeren så finnes det en tom fil ved navn position.txt i samme mappe som programmet ligger.

Det siste som skal nevnes under denne delen er *Current Position*, dette er en status bar som forteller brukeren om hvor langt man er fra yterposisjonene. Når baren er helt til

[36] Det kan benyttes totalt 1,55 meter, men av sikkerhetsmessige hensyn blir bare et spenn på 1,5 meter benyttet.

venstre vil man være på toppen, med andre ord vil filen positionCheck.txt være 0. Er derimot posisjonen til hydrofonen helt i bunn, vil baren være fylt opp helt til høyre.

Stepping

Siste del av brukergrensesnittet er selve steppingsdelen som vist i merknad B i Figur 32. Et utsnitt av det aktuelle området er også vist under i Figur 33.

Figur 33 - Grafisk grensesnitt, utsnitt av steppingsdel

I denne delen velger man om man skal kjøre automatisk måleserie eller om man skal posisjonere manuelt. Hvis *Automatic* blir valgt, så vil programmet selv styre hvor langt posisjoneringsriggen forflytter seg, og det vil bli gitt signaler om at datainnsamlinger skal gjennomføres. Denne måten er den eneste for å få stativet til å fungere sammen med en eventuell ekstern datainnsamlingsdel, det vil ikke fungere med den manuelle operasjonen. Blir *Direction* satt til *Down*, og at verdien på positionCheck.txt er mindre enn 255 vil posisjoneringsstativet operere på følgende måte:

1. Si ifra til ekstern målingsdel at måling kan gjennomføres.
2. Vente på signal om at måling er gjennomført.

3. Sjekke fra positionCheck.txt om man faktisk skal bevege seg, hvis ikke avsluttes listen her. Det avsluttes her hvis positionCheck.txt er 255[37].
4. Forflytte hydrofonen en distanse tilsvarende ½ bølgelengde.
5. Oppdatering av filer. Nåværende posisjon i millimeter skrives til position.txt for deretter å oppdatere positionCheck.txt med en verdi 1 høyere enn den som var der.[38]
6. Si ifra til målingsdel at man har oppnådd ønsket posisjon, og at ny måling kan gjennomføres.
7. Gå så tilbake til punkt 2.

Velges det *Manual* i Figur 33 vil systemet styres på bakgrunn av mål som brukeren gir. Når dette velges vil flere valgmuligheter også komme til overflaten, de to rullegardinmenyene i Figur 33 vil bli mulig og bruke. ”*Please choose stepping units*...” gir brukeren en rullegardinmeny hvor det kan velges ”*half wavelength*”, ”*centimeters*”, ”*to top*” og ”*to bottom”*. Dette er da måter å steppe manuelt på. ”*Half wavelength*” gir muligheten til å forflytte en lengde i antall halve bølgelengder. Antallet sattes av brukeren i rullegardinmenyen under, enten ved å velge en av de oppgitte verdiene eller ved å sette inn en manuelt. Dette gjelder således for alle valgene som finnes under ”*Please choose stepping units*...” rullegardinmenyen. Halv bølgelengde steppings muligheten er vel mest anvendelig under testing av posisjoneringsstativet og ikke nødvendigvis under feltarbeid. Videre er det muligheten til å forflytte seg et vist antall centimeter. Eller helt til toppen ved ”*To top*”, eller til bunnen ved ”*To bottom*” valgene. I de to siste valgene vil menyen som gir muligheten til å sette inn antall bli umulig å bruke, siden det ikke gir mening å for eksempel kjøre to ganger til toppen. Alle måleenhetene regnes om til halve bølgelengder da det er dette målet som blir overført til mikrokontrolleren.

For å starte selve forflytningen må selvfølgelig ”*Start måleserie*” knappen benyttes.

[37] I tilfellet hvor man kjører i oppadgående modus vil tallet være 0 når man går ut.

[38] Hvis den opprinelige verdien var for eksempel 34 vil den nye verdien være 35 så lenge stativet går nedover, 33 hvis stativet opererte i oppadgående modus.

Programstrukturen i mikrokontrolleren

Siden systemet skal ha muligheten til å kjøre både i manuell og automatisk modus måtte dette også inkorporeres i mikrokontrolleren. Generelt så opererer koden i mikrokontrolleren på avbrudd, med en evig løkke gående i *Main* metoden. Dette er en vanlig måte å gjøre det på. For at noe skal skje i det hele tatt må det eksterne avbruddet på INT1, fysisk pin 17, gå høyt. Da sjekker mikrokontrolleren om signalet ut fra pin 25 på PMD-1208LS boksen og inn på mikrokontrolleren sin PD0 på PORTD med fysisk pin nummer 14, se Figur 17 tidligere i dette kapitelet for Atmega16 sin pinkonfigurasjon, gir mikrokontrolleren beskjed om systemet skal operere i manuell[39] eller automatisk modus[40]. For den fullstendige koden henvises det til appendiksen.

Automatisk modus

Når så mikrokontrolleren har valgt at man skal operere i automatisk modus kjøres metoden *autoStep()*. Det som så skjer er at metoden *akselerasjon()* blir kalt. Denne metoden genererer et akselererende pulstog opp til ca 1,9kHz. Når dette er gjort blir metoden *funksjonsgenerator()* kalt slik at man genererer et pulstog med konstant frekvens en viss tid, for så å kalle metoden *deakselerasjon()*. Totalt vil disse tre metodene gi en total forflytning på ½ bølgelengde. Når dette er gjort vil mikrokontrolleren gi beskjed at den er ferdig med forflytningen og at en eventuell måling kan foretas. Mikrokontrollerens avbruddsrutine er nå fullført og vil programmessig gå tilbake til den evige løkka i *Main()*.

Manuell

Er signalet inn på PD0 på mikrokontrolleren satt høyt vil metoden *manuelStep()* bli kalt av avbruddshåndteringen. Det som så skjer er at mikrokontrolleren leser inn hva

[39] Signalet er høyt.

[40] Signalet er lavt.

som ligger på PORTA[41]. Dette er antallet halve bølgelengder som det totalt skal forflyttes satt ut fra PMD-1208LS boksen. Det sjekkes så at det faktisk er et tall større en null som ligger inn på porten. Hvis det er tilfellet så kalles metoden *akselerasjon()* slik at motoren blir akselerert opp til ca 1.9kHz, videre vil det bli tilbakelagt en distanse på ¼ bølgelengde. Det vil så bli sjekket om motoren skal gå en distanse lengre enn totalt ½ bølgelengde, hvis så går stativet ¼ bølgelengde før den sier ifra at den har gått ½ bølgelengde tilbake til PMD-1208LS for så å faktisk gå den siste ¼ bølgelengden. Grunnen til at den opererer på denne måten er tosidig. Først og fremst for å få timingen i metoden *manuelStep()* riktig. Man oppnår således bedre presisjon på denne måten i tilfelle et reset signal eller tilsvarende skulle bli kalt. Hvis man hadde oppdatert PMD-1208LS, og dermed også positionCheck.txt, når intervallet på ½ bølgelengde var tilbakelagt i stedet for etter ¼, ville man ikke få registrert at man hadde tilbakelagt noen distanse i det hele tatt hvis et reset signal skulle bli kalt rett før intervallet var gjennomført. På denne måten kunne tellemekanismen i systemet blitt dårligere enn det strengt tatt trengte å bli. Hvis så, med måten det har blitt gjort på her, et reset blir kalt når man har tilbakelagt en distanse på mellom ¼ og ½ bølgelende vil dette bli registrert som å ha tilbakelagt ½ bølgelengde. I prinsippet blir den en måte å runde av på. Programmet vil så gå i en løkke hvor antallet er satt av hva som ble lest inn på PORTA litt tidligere. Når den så har gått den distansen som den fikk beskjed om vil metoden *deakselerasjon()* bli kalt og det vil bli sagt ifra at den ønskede avstanden er tilbakelagt. For å få den totale forståelsen av hvordan mikrokontrolleren opererer i manuell modus ansees kildekoden i appendiksen som god lesning.

[41] Hver oppmerksom på at kodemessig må PINA brukes når det leses fra port og ikke PORTA. PORTA brukes når det skal settes verdier *ut* på porten.

Kapittel 3

Testkjøring og Resultater

I denne delen ser jeg på resultatene som ble oppnådd ved den siste testkjøringen av utstyret. Utstyret er koblet opp slik som det har blitt gjennomgått i ”Materialer og Metode”-kapittelet. Kort fortalt ble det brukt 3 stykk 12 volts bilbatterier koblet i serie med en stor kondensator[1] i parallell. Kondensatoren var tilkoblet så nærme steppermotorens styringskrets som mulig. Videre ble posisjoneringsstativet, dets styringselektronikk og det selvutviklet styringsprogrammet benyttet. Før testingen ble startet ble det passet på at alle batteriene ble ladet opp.

Signaler

Det kan konstanteres at alle signaler fungerer slik som antydet i ”Materialer og metode” kapitelet. Det akselererende pulstog blir generert slik som forventet. Signalet som sier ifra til målingsdelen at måling kan foretas blir satt korrekt. Porten som deretter lytter etter signal om at AD-omvandlingen fra målingsdelen er ferdig, fungere også fint. Dette resulterer i at posisjoneringsdelen vil være lett å koble til et eksternt datainnsamlingssystem.

I selve testkjøringen blir signalet ut til målingsdelen og signalet inn fra målingsdelen koblet sammen. På denne måten oppnås det en drift av systemet hvor posisjoneringsdelen ”tror” at målingsdelen er tilstedet.

[1] 39000 μF.

Nøyaktighet av posisjonering

Hovedproblemstillingen i denne oppgaven var først og fremst å få til en nøyaktig posisjonering av hydrofonen. Så la oss først ta en titt på hva slags resultater det utviklede systemet ga med hensyn til nøyaktigheten på posisjoneringen.

Når systemet opererer i automatisk modus er det satt til å bevege seg en teoretisk distanse på 6mm før systemet gjør klart for et eksternt datainnsamlingssystem. Det som er interessant er om det er overensstemmelse mellom den faktiske forflytningen, og den teoretiske distansen på 6mm som programvaren opererer med. En måte å sjekke dette på er å kjøre en forflyttning[2] på stativet, for deretter å måle eventuell feil mellom den teoretiske og den forflytningen som ble oppnådd. Det å måle en distanse på 6mm relativt nøyaktig er en utfordring. Dette kan medføre større unøyaktigheter i målingene enn nødvendig. I stedet ble det valgt å kjøre en manuell serie på 20 forflytninger, noe som tilsvarer en teoretisk distanse på 120,0mm. For deretter å finne gjennomsnittet oppnådd for en forflytning.

For å måle den totale forflytningen ble det satt et merke på selve stativet ovenfor festet mellom aluminiumsrørene og festet til hydrofonfesteplata, se merknad A i Figur 1 på neste side. Deretter ble den ønskede distansen på 20 forflytninger kjørt, se Figur 2 for oppsett av programvare. En ny markering ble satt ved sluttposisjonen. Det ble brukt varsomhet når posisjonen ble markert, slik at den ble så nøyaktig som mulig. Hydrofonfesteplaten ble så forflyttet ut av veien slik at målingene med skyvelære kunne gjennomføres. Distansen ble målt[3] to ganger og ga de respektive målene 119,4mm og 119,2mm. I gjennomsnitt vil da posisjonsforflytningen ha vært 119,3mm, når den teoretiske forflytningen skulle være 120,0mm. Dette er således et negativt[4] avvik på 0,7mm. Avviket er dog over en distanse som tilsvarer 20 målinger. Slik at i realiteten vil avviket være 0,035mm ± 0,005mm[5] mellom den teoretisk og den oppnådde distansen per forflytning.

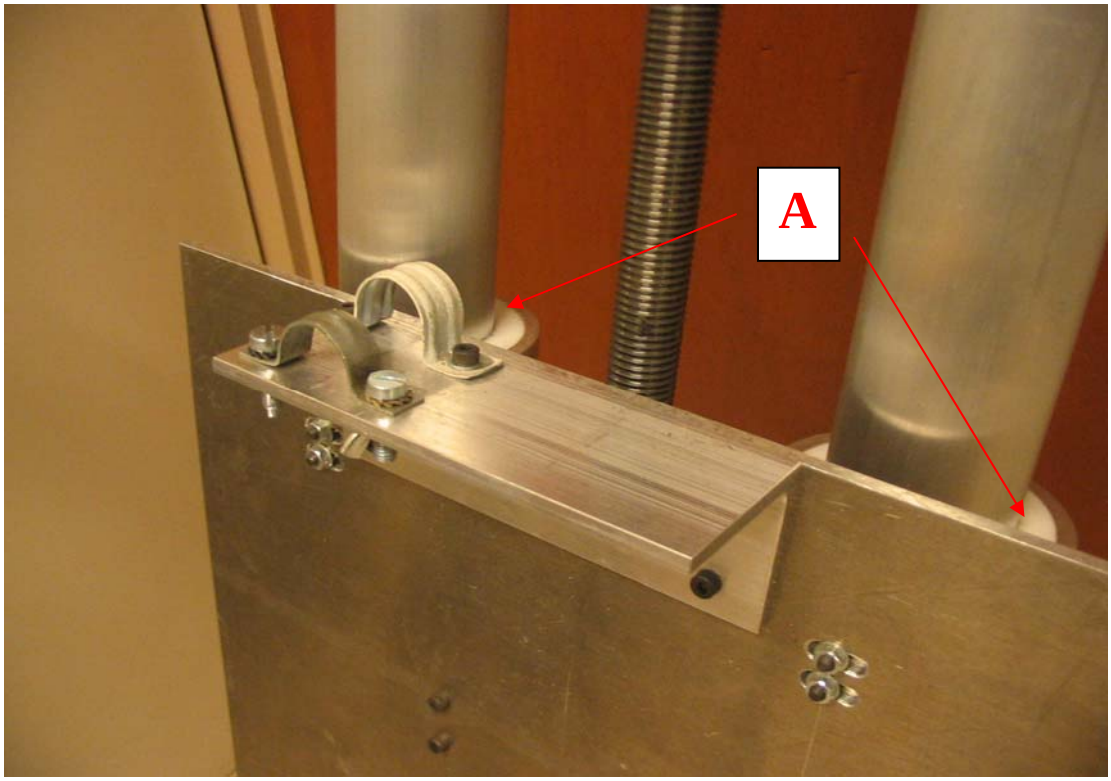
[2] En forflytning vil si ½-bølgelengde, noe som tilsvarer 6mm.

[3] Spesifiserer at det den forflyttede distansen ble målt 2 ganger, distansen ble ikke kjørt 2 ganger.

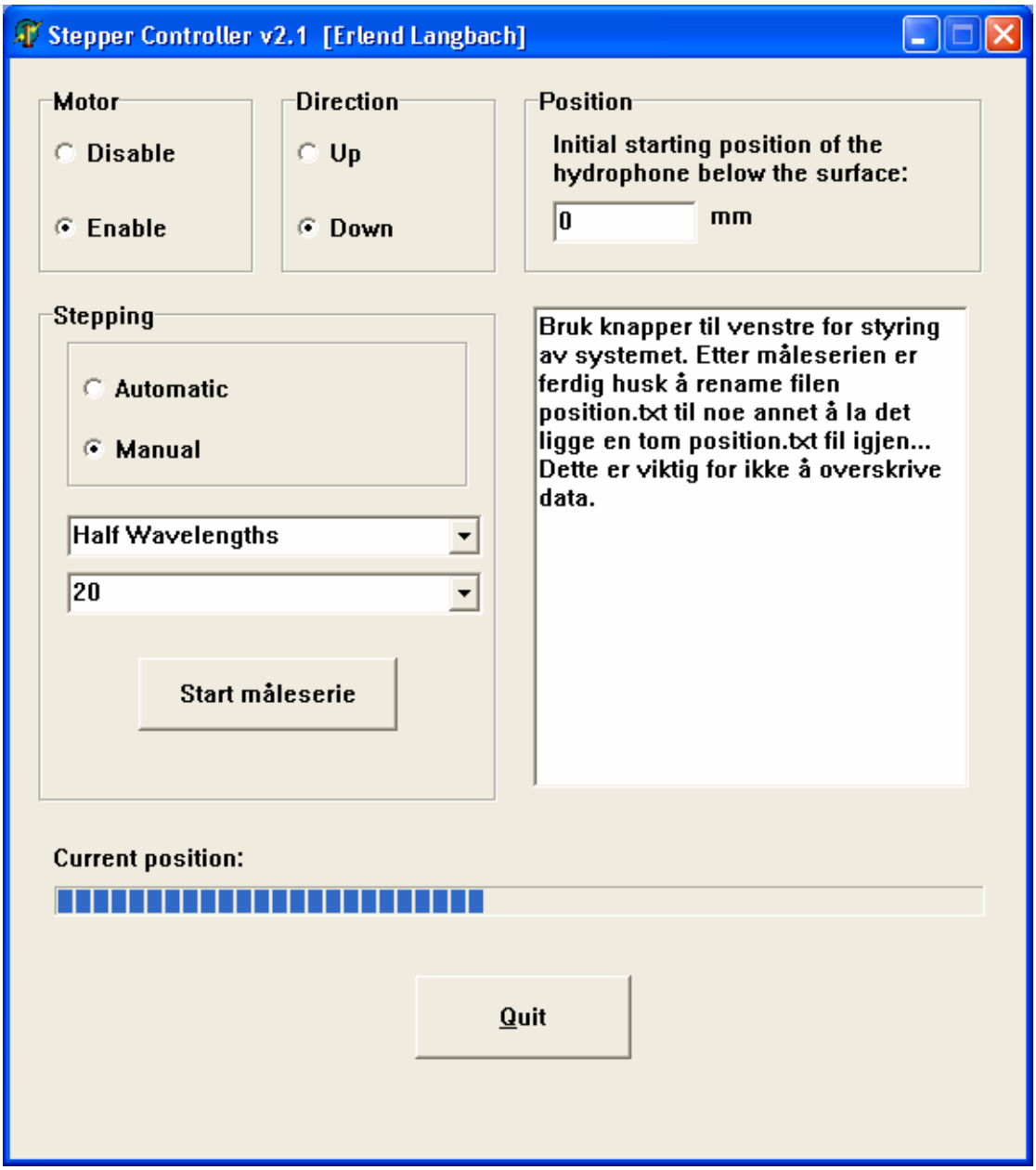
[4] Med negativt avvik menes det at systemet gikk mindre enn den teoretiske distansen.

[5] Forskjellen mellom det målte og teoretiske vil variere mellom 0,6mm og 0,8mm på de 20 målingene. Slik at faktisk avvik per måling blir mellom 0,03 og 0,04mm. Det skal merkes at en distanse på 0,03 ikke er en målbar størrelse i denne sammenhengen, men kun en kalkulert distanse.

Avviket i distansen kan det teoretisk sett kompenseres for i software hvis det er ønskelig. Variabelen *SampleInterval* som benyttes i Delphi programmet kan settes til den målte distansen i stedet for den teoretisk kalkulerte. Det er derimot en sterk anbefaling å teste ut systemet grundig før noe slikt gjøres, slik at man ikke ender opp med en situasjon som gir opphav til dårligere nøyaktighet enn hva som var ønskelig.



Figur 1 - Markering på stativ under testkjøring



Figur 2 - Innstilinger i "Stepper Controller" program under testkjøring av nøyaktighet

Resultater ved generell drift av systemet

Det ble merket at systemet ikke har kapasitet til å kjøre over en lengre periode slik som oppsettet er nå. Under den siste testkjøringen på labben ble det konstatert at systemet får problemer med å kjøre tilfredsstillende etter en times tid. Den høyeste frekvensen som blir benyttet inn på motorensdriverkort er 1,85kHz. Det har vist seg at motoren har en tendens til å låse seg når man akselererer opp til denne frekvensen. Det vil si at motoren slutter å gå rundt.

Dette ble utforsket litt nærmere og det kan tyde på at denne låsning skjer rundt omkring 1,3kHz. Eksakt frekvens hvor dette skjer er vanskelig å si.

Det ble prøvd å redusere maks frekvensen ned til 1,3kHz. Motoren låste seg ikke like hyppig da som når 1,85kHz ble benyttet, men det skjedde fortsatt innimellom. Det ble deretter prøvd å justere bryterne på motordriverkortet slik at "Standby current is on 20%"[6] og "Always fast current decay mode"[7][1] var aktivert, samt alle de innbyrdes mulighetene dette kunne gi. Dette ga derimot ingen merkbare resultater verken når 1,3kHz eller 1,85kHz ble brukt som maks frekvens.

Videre ble det prøvd å benytte halvstepping. Det vil si at motoren vil bevege seg halvparten av hva den vanligvis ville gjøre. Slik at for hvert stepp driverkortet får vil motoren gå 0.9° i stedet for 1.8°. Dette vil halvere hastigheten som hydrofonen beveges med. Halvstepping skrues på ved å aktivere bryter 2[8]. Under testkjøring med denne konfigurasjonen låste ikke motoren seg. Når derimot programvaren til mikrokontrolleren ble omprogrammert til å kompensere for hastighetstapet[9], endte man opp med at motoren låste seg slik som tidligere.

Det viste seg også at selve spenningen over batteriene droppet med nesten 1 volt under den 3 timer lange testing. Ved tidligere testing har det også blitt observert at spesielt det ene batteriet mister spenning relativt fort.

[6] Dipswitch 3, se datablad [1].

[7] Dipswitch 4, se datablad [1].

[8] Dipswitch 2, se datablad [1].

[9] Slik at frekvensen på pulstogene ble doblet.

Kapittel

4

Diskusjon og konklusjon

Ved Sonar gruppa blir det benyttet en ”eliptisk split-beam transducer” med en operativ frekvens på 120kHz. I denne oppgaven skulle det legges tilrette for at dette lydfeltet kunne kartlegges nøyaktig ved tilkobling til et enkelt datainnsamlingssystem. Slik at signalene fra hydrofonen registreres innenfor et dybdeområde på minst 1.5 meter. Problemstilingen var å angi posisjonen til hydrofonen med tilstrekkelig nøyaktighet slik at lydfeltet kunne kartlegges med en oppløsning bedre enn en bølgelengde. Det vil si bedre enn 1cm.

I denne oppgaven er det blitt produsert et system som fint leverer kravet til nøyaktighet. Det ble målt ved testkjøring at systemet har et negativt avvik på 0,035mm ± 0,005mm per forflytting på 6,0mm. Som spesifisert i foregående kapittel er dette avviket ikke målbart i seg selv, men kun et kalkulert avvik ut i fra en gjennomsnittskalkulasjon. Man kommer som oftest ikke unna en eller annen form for hysterese i en mekanisk overførsel. Slik at det lille avviket som faktisk oppstår kommer mest sannsynlig fra dette. Ut i fra dette og testkjøringen kan det konkluderes med at selve programmet, posisjonsestimeringen og overførselen av signaler mellom de forskjellige elementene fungerer ypperlig.

Når det kommer til selve utholdenheten og driftssikkerheten av systemet viste det seg at dette ikke på langt nær holdt mål. Motoren endte opp med å låse seg til tider. Det ble merket at batteriene, spesielt det ene, hadde et forholdsvis stort spenningstap over de timene systemet ble brukt. Dette kan tyde på at batteriene ikke er helt bra. Spesielt siden det ikke ble oppdaget noe som helst tull med at motoren låste seg før batterispenningen hadde tapt seg litt. En annen forklaring er at motoren ikke har tilstrekelig med dreimoment. Siden problemet oppstår kun etter at systemet har vært i drift noen timer, ser jeg det derimot rimelig å konkludere med at batterienes tilstand er grunnen til at systemet ikke fungere slik som ønsket.

Forbedringer og videreføring

Det hadde vært interessant og testkjørt stativet med tre splitter nye 12 volts batterier, for da å se hvordan systemet oppførte seg. Eventuelt kan det også prøves med fire batterier, siden motorstyringskretsen takler opp til 48 volt. På denne måten kunne man sjekke om det var problemer med batteriene som hemmet systemets drift.

PMD-1208LS boksen kunne også vært byttet ut med en annen boks som greier å håndtere avbrudd. Slik som systemet er nå så vil det egenutviklede *Stepper Controler* programmet stå å ”henge”. Det vil si at systemet kjører mellom en prosedyre eller rutine som hele tiden sjekker hva som ligger på porten, og når ønsket verdi kommer vil programmet gå videre. Det skal ikke kunne oppstå tilfeller hvor programmet låser seg på grunn av dette. Hvis så blir tilfellet kan man benytte seg av reset på mikrokontrolleren, da alle metodene hvor programmet lytter på en port har inkorporert å hoppe ut av rutinen hvis reset blir aktivert.

Hvordan Stepper Controler vil fungere sammen med eventuelt programvare for datainnsamling kjørt på samme datamaskin er uvisst. Windows støtter *multitasking*, men det er usikkert hvor mye av datamaskinens ressurser Stepper Controler tar grunnet denne ”hengingen”. Dette bør i så fall testes ut nøye. Slik som det er forstått så ser det ut som noen av Measurement Computing sine USB baserte bokser støtter en form for avbrudd. Hvis dette sees inn i og er tilfellet, kan Stepper Controler lett omgjøres til å benytte seg av dette da funksjonsbiblioteket som følger med Measurement Computing sine bokser er relativt universelle. Slik kan den allerede utviklede koden fint benyttes videre.

For at systemet så skal være helt operativt trengs det å testkjøre utstyret med samme lengde på kabler som skal benyttes i feltarbeidet. Det også nødvendig og teste hvor stor kondensatoren som står ute på stativet trenger å være. Alt som står ute på stativet bør pakkes inn i vanntette bokser, og elektronikken som står på land kan pakkes inn i en boks. Det er vitalt at det inkorporeres en reset knapp på boksen på land slik at man når som helst kan stoppe det som skjer ute på stativet.

Bibliografi

[1] AT Drives: *Stepper Driver Manual AT15R05*. [online] Elfa, 2002. Tilgjengelig fra http://www.elfa.se/pdf/54/05446406.pdf [hentet 30. nov. 2005] [CD:\Datablader\ Steppermotor styrekrets.pdf]

[2] Corporation Atmel: *AVR STK500 User Guide*, Atmel Corporation, 2004. [Tilgjengelig også elektronisk: http://www.atmel.com/dyn/resources/prod_documents/doc1925.pdf] [hentet 30. nov. 2005]

[3] Corporation Atmel. *8-bit AVR Microcontroller with 16K Bytes In-system Programmable FLASH. ATmega16. ATmega16L*. [online] Atmel Corporation, 2003. Tilgjengelig fra http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf [hentet 30. nov 2005] [CD:\Datablader\ATMega16 datablad.pdf]

[4] Corporation Measurment Computing: *PMD-1208LS User's Guide*, Measurment Computing Corporation, 2003. [Tilgjengelig også elektronisk: http://www.measurementcomputing.com/PDFManuals/ PMD-1208LS.pdf [hentet 30. nov. 2005]

[5] Corporation Measurment Computing: *Universal Library User's Guide*, Measurment Computing Corporation, 2003.

[6] Corporation Measurment Computing: *Universal Library Function Reference*, Measurment Computing Corporation, 2003.

[7] Faber Moons: *Hybrid Stepping motor type 23HS3001-01*. [online] Elfa, 2002. Tilgjengelig fra http://www.elfa.se/pdf/54/05446216.pdf [hentet 30. nov. 2005] [CD:\Datablader\ Steppermotor 23HS3001datablad.pdf]

[8] Hafizovic I: *Raytracingsmodul for simulering av lydutbredelse i vann*. Hovedoppgave i fysikk, Universitetet i Oslo, 2004.

[9] Hafting Y: *En undersøkelse av Universal Serial Bus, USB brukt til sanntisdatainnsamling*. Hovedoppgave i fysikk, Universitetet i Oslo, 2000.

[10] Jahr V: *Horisontal lydutbredelse i grunt vann. Målinger i elv og islagt vann. Simuleringer basert på ray tracing*. Hovedoppgave i fysikk, Universitetet i Oslo, 2003.

[11] Jones DW: *Control of Sepping Motors. A Tutorial* [online] University of Iowa, USA. Tilgjengelig fra http://www.cs.uiowa.edu/~jones/step/index.html [hentet 30. nov. 2005]

[12] Kjølerbakken KM: *Lydutbredelse i elv og på grunt islagt vann. Målemetode med hydrofon. Simuleringer basert på ray tracing.* Hovedoppgave i fysikk, Universitetet i Oslo, 2003.

[13] Lischner R: *Delphi in a Nutshell: A Desktop Quick Reference*, O'Reilly, 2000.

[14] Prinz P and Kirch-Prinz U: *C Pocket Reference*, O'Reilly, 2002.

[15] Samdal T: *Delphi Programmering – en innføring*, Samag, 2000.

Appendiks: Kildekode

”Stepper Controller” Delphi kildekode

```
unit Styring;
{
Styring_prosjekt.dpr
================================================================================
====

 File:                  Styring.pas

 Version:               v 2.1

 Purpose:               Software for controlling steppermotor, and for its
                        positining of a hydrophone.

 Demonstration:         Configures digital ports for output - PORTA is used
for
                        control signals to send to the MCU, while PORTB is
used
                        as a bus for sending 8bit to MCU.

                        Analog port is used for input, as a receiver of
signals
                        back from MCU. 2 signals for control/timing. 1
signal
                        to check MCU's reset signal.


 Upgrades from          Now incorporates the ability to detect MCU reset
 last version:          signals. We are then able to detect and able to go
                        out of loops to stop counting etc. This adds
increased
                        accuracy in counting and in controlling of
hydrophone's
                        position.
```

```
 Left to implement:     1.  Better information to user.
                        2.  Sampling part - where should signals be detected
                            from. MCU or PMD-1208LS? and should any signals
                            be sent back?
                        3.  Splash Screen

 Improvements:          It would be great to incorporate the ability to make
                        the program better at "multi-tasking". In other
words:
                        not the need for the program to "hang" as it does
today.

                        I'm not sure if this is possible with the current
board
                        which we are using today. We might need to upgrade
to
                        the PMD-1024LS board, but this must be looked into
                        if the necessity of it arises.

MCU = Micro controling unit, in this case an Atmel Mega16 (Atmega16)

================================================================================
====
}
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, cbw, ComCtrls;

type
  TfrmStepper = class(TForm)
    cmdQuit: TButton;
    MemoData: TMemo;
    rgpDirection: TRadioGroup;
    GroupBox1: TGroupBox;
    cbxStep: TComboBox;
    btnStartStep: TButton;
    rgpEnable: TRadioGroup;
    cbxStepUnit: TComboBox;
    rgpManuel: TRadioGroup;
    GroupBox2: TGroupBox;
    edtInitPos: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    pgbStatus: TProgressBar;
    Label4: TLabel;
```

```
    procedure cmdQuitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure rgpDirectionClick(Sender: TObject);
    procedure rgpEnableClick(Sender: TObject);
    procedure btnStartStepClick(Sender: TObject);
    procedure rgpManuelClick(Sender: TObject);
    procedure makeINT1;
    procedure waitMCU;
    procedure generateStartSamplingSignal;
    procedure waitForSamplingToBeDone;
    procedure appendPositionToFile(PositionCheckTemp: Integer);
    procedure appendPositionCheckToFile(PositionCheckTemp: Integer);
    procedure cbxStepUnitChange(Sender: TObject);

    function returnPositionCheckFromFile(): Integer;
    function returnNumberOfSampleIntervals(): Integer;
    function convertCmToSampleInterval(StepsNumberFromUser: String):
Integer;
    function checkOutOfRig(StepsFromUser: Integer): Integer;
    function checkMCUReset(): Boolean;
    function checkAnalogInput(Channel: Integer): Boolean;
    function updatePositionCheckTemp(PositionCheckTemp: Integer): Integer;

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmStepper: TfrmStepper;

implementation

{$R *.DFM}

var
   {Textfiles for appending position and temp. position}
   positionFile:           TextFile;
   positionCheckFile:      TextFile;

   {Variables used for board}
   ULStat:                 Integer;
   DataValue:              Word;

   {For internal errorhandling in Universal Library}
   ErrReporting:           Integer;
```

```
   ErrHandling:            Integer;
   RevLevel:               Single;

const
{*********************************************************}
{                 For Digital output                      }
{*********************************************************}

   {PMD-1208LS}
    BoardNumA:             Integer = 0;
    PortDirectionOut:      Integer = DIGITALOUT;

{*********************************************************}

   {PMD-1208LS PortA}
   PortTypeA:              Integer = FIRSTPORTA;

   {Name:                  Type:     Bit#}
   BitNumStepperEnable:    Integer = 0;
   BitNumDirection:        Integer = 1;
   BitNumInterupt:         Integer = 2;
   BitNumStartSampling:    Integer = 3;
   BitNumAutoOrMan:        Integer = 4;
   BitManControll:         Integer = 5;



{*********************************************************}

   {PMD-1208LS PortB}
   PortTypeB:              Integer = FIRSTPORTB;

   {Navn:                  Type:     Bit#}
   BitCont0:               Integer = 8; //LSB
   BitCont1:               Integer = 9;
   BitCont2:               Integer = 10;
   BitCont3:               Integer = 11;
   BitCont4:               Integer = 12;
   BitCont5:               Integer = 13;
   BitCont6:               Integer = 14;
   BitCont7:               Integer = 15; //MSB

{*********************************************************}
{                  For Analog input                       }
{*********************************************************}
  Range:                   LongInt = BIP10VOLTS;   {set input range to +/-
10V}
  AIOReset:                Integer = 0;
```

```
  AIOAllDone:              Integer = 1;
  AIOCycleDone:            Integer = 2;
  AIOSamplingDone:         Integer = 3;

{*********************************************************}

  SampleInterval:          Single = 6.0;
  MinSamples:              Word = 0;
  MaxSamples:              Word = 250;

procedure TfrmStepper.FormCreate(Sender: TObject);
begin
  {declare Revision Level}
  RevLevel := CURRENTREVNUM;
  ULStat := cbDeclareRevision(RevLevel);

  {Setting the internal error handling for the Universal Library}
  {Reporting ALL errors}
  ErrReporting := PRINTALL;
  {Stop on ALL errors}
  ErrHandling := STOPALL;
  {The usage of ULStat is a common way of checking for errors
  this will be used thruout the code}
  ULStat := cbErrHandling(ErrReporting, ErrHandling);
  if ULStat <> 0 then exit;

  {Configuring PORTA as an output}
  ULStat := cbDConfigPort (BoardNumA, PortTypeA, PortDirectionOut);
  if ULStat <> 0 then exit;
  {Initialising the portvalue}
  DataValue := 0; ULStat := cbDOut (BoardNumA, PortTypeA, DataValue);
  if ULStat <> 0 then exit;

  {Configuring PORTB as an output}
  ULStat := cbDConfigPort (BoardNumA, PortTypeB, PortDirectionOut);
  if ULStat <> 0 then exit;
  {Initialising the portvalue}
  ULStat := cbDOut (BoardNumA, PortTypeB, DataValue);
  if ULStat <> 0 then exit;


  {OBS OBS OBS OBS OBS OBS}
  {In english?}
  MemoData.Text := 'Bruk knapper til venstre for styring av systemet. Etter
måleserien er ferdig husk å rename filen position.txt til noe annet å la det
ligge en tom position.txt fil igjen... Dette er viktig for ikke å overskrive
data';
```

```
  {Setting up handles for the *.txt files to be used}
  {Data file for keeping track of position of where each sample has been
taken
  in millimeter}
  AssignFile(PositionFile, 'position.txt');

  {File to be used for keeping track of temp position in case of program
crash}
  AssignFile(PositionCheckFile, 'positionCheck.txt');

  {Configuring initial values for outputs according to GUI}
  {1. direction}
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumDirection,
rgpDirection.ItemIndex);
  if ULStat <> 0 then exit;
  {2. enable}
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumStepperEnable,
rgpEnable.ItemIndex);
  if ULStat <> 0 then exit;
  {3. auto or manuel stepping}
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumAutoOrMan,
rgpManuel.ItemIndex);
  if ULStat <> 0 then exit;

  {Setting the initial position of the status bar}
  pgbStatus.Position := returnPositionCheckFromFile();

end;

{Exit procedure}
procedure TfrmStepper.cmdQuitClick(Sender: TObject);
begin
  Close;
end;

{Setting of direction signal}
procedure TfrmStepper.rgpDirectionClick(Sender: TObject);
begin
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumDirection,
rgpDirection.ItemIndex);
  If ULStat <> 0 then exit;
end;

{Setting of enable signal}
procedure TfrmStepper.rgpEnableClick(Sender: TObject);
begin
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumStepperEnable,
rgpEnable.ItemIndex);
```

```
  If ULStat <> 0 then exit;
end;

{GUI update for switching between manual og automatic operation}
procedure TfrmStepper.rgpManuelClick(Sender: TObject);
begin
  {Setting BitNumAutoOrMan output to keep MCU updated}
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumAutoOrMan,
rgpManuel.ItemIndex);
  If ULStat <> 0 then exit;

  {GUI update for automatic}
  if rgpManuel.ItemIndex = 0 then
  begin
    cbxStepUnit.Enabled := false;
    cbxStepUnit.Text := 'Enhet:';
    cbxStep.Text := '0';
    cbxStep.Enabled := False;
    btnStartStep.Caption := 'Start måleserie';
  end;

  {GUI update for manuel}
  if rgpManuel.ItemIndex = 1 then
  begin
    cbxStepUnit.Enabled := true;
    btnStartStep.Caption := 'Start';
  end;

end;

{Starting stepping}
procedure TfrmStepper.btnStartStepClick(Sender: TObject);
var
  PositionCheckTemp:          Integer;
  NumberOfSampleIntervals:    Integer;
  Counter1:                   Integer;
  ButtonSelected:             Integer;
  Continue:                   Boolean;
  BooleanAllDone:             Boolean;

begin
  {Reads from file to set a temp position variable (PositionCheckTemp)}
  PositionCheckTemp := returnPositionCheckFromFile();

  {Regarding the variable "continue" used later. If this is set to logical
  false then the loop will be discontinued}

  if checkMCUReset() then ShowMessage('Reset is enabled. To continue please
```

```
set reset accordingly.')
  else
  begin
    if (PositionCheckTemp = MinSamples) and (rgpDirection.ItemIndex = 0)
then ShowMessage('The rig has reached its maximum top position. Please use
the direction button to change direction.')
    else
    begin
      if (PositionCheckTemp = MaxSamples) and (rgpDirection.ItemIndex = 1)
then ShowMessage('The rig has reached its maximum bottom position. Please
use the direction button to change direction.')
      else
      begin
        {*****|AUTOMATIC|*****}
        if rgpManuel.itemIndex = 0 then
        begin
          Continue := True;
          ButtonSelected := MessageDlg('Are you sure you wish to start a
measuring serie? this may take up to 45 minutes.',mtCustom, [mbYes, mbNo],
0);
          if ButtonSelected = mrYes  then Continue := True;
          if ButtonSelected = mrNo then Continue := False;

          if Continue then
          begin
            {Starts sampling}
            generateStartSamplingSignal();
            {Waiting for sampling to be done}
            waitForSamplingToBeDone();
            {Append current sampling position to file}
            AppendPositionToFile(PositionCheckTemp);

            {Updating the loop variable for next sampling position}
            PositionCheckTemp := updatePositionCheckTemp(PositionCheckTemp);

            {If the next sampling position is a LEGAL position, then start
            this while-loop...}
            while (PositionCheckTemp >= MinSamples) and (PositionCheckTemp
<= MaxSamples) and Continue do
            begin
              {Generate interupt to MCU to start stepping}
              makeINT1();
              {Waiting for MCU to finish stepping}
              waitMCU();

              {Checking the MCU reset signal}
              if checkMCUReset() then
              begin
```

```
                ShowMessage('Reset is enabled at the MCU, please fix this
before continuing. Accuracy might also have been lost due to the reseting of
the MCU');
                Continue := False;
              end
              else
              begin
                {Starts sampling}
                generateStartSamplingSignal();
                {Waiting for sampling to be done}
                waitForSamplingToBeDone();
                {Saving current temp. position to file}
                appendPositionCheckToFile(PositionCheckTemp);
                {Writing posistion in mm til position.txt file}
                appendPositionToFile(PositionCheckTemp);
                {Updating progression bar}
                pgbStatus.Position := PositionCheckTemp;
              end;

              {Updating the loop variable for next sampling position}
              PositionCheckTemp :=
updatePositionCheckTemp(PositionCheckTemp);
            end;
          end;{Done IF-CONTINUE loop}
        end;{Done AUTOMATIC loop}

        {*****|MANUAL|*****}
        {There is no need for sampling in MANUA,L-mode}
        if rgpManuel.itemIndex = 1 then
        begin
          {Checking how many 1/2-wavelengths the hydrophone should be moved}
          NumberOfSampleIntervals := returnNumberOfSampleIntervals();

          {Initialize values which have been used earlier}
          BooleanAllDone := False;
          Continue := True;

          if cbxStep.Text = '0' then ShowMessage('Please set the stepping
controls to apropriate values/units')
          else
          begin
            {Setting out a 8-bit number on PORTB in parallel
            to let the MCU know the distance it should travel}
            {Using a loop to make sure the value is set appropriatly}
            for Counter1 := 0 to 25 do
            begin
              ULStat := cbDOut (BoardNumA, PortTypeB,
NumberOfSampleIntervals);
              If ULStat <> 0 then exit;
```

```
            end;

            {Generate interupt to MCU to start stepping}
            makeINT1();

            {While-loop to wait for MCU to be finished with stepping.
However
            to loop also updates the files each time the MCU has gone a
total
            of 1/2 wavelength!!!}
            while Continue do
            begin
              {Checking if MCU is all done}
              BooleanAllDone := checkAnalogInput(AIOAllDone);

              if BooleanAllDone then Continue := False
              else
              begin
                {waitMCU also incorporates checkMCUReset(), so the case of
the
                program going into a latch should not happen}
                waitMCU();

                if checkMCUReset() then
                begin
                  ShowMessage('Reset has been enabled at MCU, please fix
this before continuing. Be aware that precision might have been lost!!!');
                  Continue := False;
                end
                else
                begin
                  BooleanAllDone := checkAnalogInput(AIOAllDone);

                  if BooleanAllDone = False then
                  begin
                    {Updating position variables}
                    PositionCheckTemp :=
updatePositionCheckTemp(PositionCheckTemp);

                    {Updating progresion bar}
                    pgbStatus.Position := PositionCheckTemp;

                    {Appending new position to check file}
                    appendPositionCheckToFile(PositionCheckTemp);
                  end;
                end;{Finished checkMCUReset}
              end;{Finished IF}
            end;{Finished WHILE}
          end;{Finished IF}
```

```
        end;{Done MANUAL}
      end;
    end;{Finished checkReset}
  end;
end;

{Generating an Interupt signal}
procedure TfrmStepper.makeINT1;
begin
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumInterupt, 0);
  If ULStat <> 0 then exit;
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumInterupt, 1);
  If ULStat <> 0 then exit;
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumInterupt, 0);
  If ULStat <> 0 then exit;
end;

{Generate signal to sampling part of the system}
{If the sampling part is interupt driven, the
last to lines could be uncoment. Else it might
be smart to leave it as it is now...}
procedure TfrmStepper.generateStartSamplingSignal;
begin
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumStartSampling, 0);
  If ULStat <> 0 then exit;
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumStartSampling, 1);
  If ULStat <> 0 then exit;
  {ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumStartSampling, 0);
  If ULStat <> 0 then exit;}
end;

{Wait for sampling to be done}
procedure TfrmStepper.waitForSamplingToBeDone;
var
  BooleanSamplingDone : Boolean;
  StillLoop : Boolean;
begin
  {Initializing}
  StillLoop := True;

  {Loop which will continue until a signal from the sampling part
  says it is done, will also exit on reset signal}
  while StillLoop do
  begin
    BooleanSamplingDone := checkAnalogInput(AIOSamplingDone);
    if BooleanSamplingDone then StillLoop := False
    else StillLoop := True;
    if checkMCUReset() then StillLoop := False;
```

```
  end;

  {Setting StartSampling signal to logical zero}
  {If the two lines in generateStarSamplingSignal is uncommented, then
  comment the two following lines out...}
  ULStat := cbDBitOut (BoardNumA, PortTypeA, BitNumStartSampling, 0);
  If ULStat <> 0 then exit;
end;


{Waiting for MCU to be finished}
{This procedure does incorporate reset checking!!!}
{so the posibility of the program going into a latch should not be present}
procedure TfrmStepper.waitMCU;
var
  MCU_done:  Integer;
  BooleanCycleDone: Boolean;
  BooleanAllDone: Boolean;
begin
  {OBS OBS OBS GJØRE OM MCU_done TIL BOOLEAN!!!!}
  MCU_done := 0;
  while MCU_done = 0 do
  begin
    BooleanCycleDone := checkAnalogInput(AIOCycleDone);
    if BooleanCycleDone then MCU_done := 0
    else MCU_done := 1;

    BooleanAllDone := checkAnalogInput(AIOAllDone);
    if BooleanAllDone then MCU_done := 1;

    if checkMCUReset() then MCU_done := 1;
  end;

  MCU_done := 0;
  while MCU_done = 0 do
  begin
    BooleanCycleDone := checkAnalogInput(AIOCycleDone);

    if BooleanCycleDone then MCU_done := 1
    else MCU_done := 0;

    BooleanAllDone := checkAnalogInput(AIOAllDone);
    if BooleanAllDone then MCU_done := 1;

    if checkMCUReset() then MCU_done := 1;
  end;
end;
```

```
{Returning PositionCheckTemp read from file}
function TfrmStepper.returnPositionCheckFromFile(): Integer;
var
  PositionCheckTemp:          Integer;
  PositionCheck:              String;
begin
  Reset(PositionCheckFile);
  Read(PositionCheckFile, PositionCheck);
  PositionCheckTemp := StrToInt(PositionCheck);
  CloseFile(PositionCheckFile);

  Result := PositionCheckTemp;
end;

{Append PositionCheckTemp to file}
procedure TfrmStepper.appendPositionCheckToFile(PositionCheckTemp: Integer);
var
  PositionCheck:              String;
begin
  PositionCheck := IntToStr(PositionCheckTemp);
  ReWrite(PositionCheckFile);
  WriteLn(PositionCheckFile, PositionCheck);
  CloseFile(PositionCheckFile);
end;

{Appending position of the hydrophone for each
sampling position in millimeters to file}
procedure TfrmStepper.appendPositionToFile(PositionCheckTemp: Integer);
var
  PositionLengthTest:         Single;
  InitPosition:               Single;
  PositionLength:             String;
begin
  InitPosition := StrToFloat(edtInitPos.Text);
  PositionLengthTest := PositionCheckTemp*SampleInterval + InitPosition;

  {Format float number to give the most correct datafile.
  Number is formated to give 4 digits in front, and 1 digit
  behind the comma. When the SampleInterval is set to 6.0
  which is the default(was set earlier) the comma will only
  result in a zero behind it in the data file. If however the
  SampleInterval variable is set to for example 5.97 then
  it will matter.}
  PositionLength := FormatFloat('####.0', PositionLengthTest);
  Append(PositionFile);
  WriteLn(PositionFile, PositionLength);
  CloseFile(PositionFile);
end;
```

```
{Returning number of sampleintervals}
{Basicly a conversion function, used for manual modus}
function TfrmStepper.returnNumberOfSampleIntervals(): Integer;
var
  NumberOfSampleIntervals:    Integer;
  StepsFromUser:              Integer;
begin
  NumberOfSampleIntervals := 0;

  {Checking what user wrote in cbxStep.Text box}
  try
    {Checking units, and then setting accordingly}
    case cbxStepUnit.ItemIndex of
    0:{Nothing to do...};
    1:{Half Wavelength}
      begin
        StepsFromUser := StrToInt(cbxStep.Text);
        NumberOfSampleIntervals := checkOutOfRig(StepsFromUser);
      end;
    2:{cm}
      begin
        StepsFromUser := convertCmToSampleInterval(cbxStep.Text);
        NumberOfSampleIntervals := checkOutOfRig(StepsFromUser);
      end;
    3:{To top of rig}
      begin
        rgpDirection.ItemIndex := 0;
        StepsFromUser := maxSamples;
        NumberOfSampleIntervals := checkOutOfRig(StepsFromUser);
        cbxStep.Enabled := True;
        cbxStepUnit.ItemIndex := 1;
        cbxStep.Text := IntToStr(NumberOfSampleIntervals);
      end;
    4:{To bottom of rig}
      begin
        rgpDirection.ItemIndex := 1;
        StepsFromUser := maxSamples;
        NumberOfSampleIntervals := checkOutOfRig(StepsFromUser);
        cbxStep.Enabled := True;
        cbxStepUnit.ItemIndex := 1;
        cbxStep.Text := IntToStr(NumberOfSampleIntervals);
      end;
    end;
  except
    cbxStep.Text := '0';
    NumberOfSampleIntervals := 0;
    ShowMessage('Please only use integer values when the half lambda option
```

```
is used');
  end;

  {Returning number of sample intervals}
  Result := NumberOfSampleIntervals;
end;

{Function used to check that the hydrophone
position stays within the legal interval.}
function TfrmStepper.checkOutOfRig(StepsFromUser: Integer): Integer;
var
  PositionCheckTemp:          Integer;
  NumberOfSampleIntervals:    Integer;
begin
  NumberOfSampleIntervals := 0;

  PositionCheckTemp := returnPositionCheckFromFile();

  {***Setting number of sampleintervals***}
  {Upward motion}
  if rgpDirection.ItemIndex = 0 then
  begin
    if PositionCheckTemp <= StepsFromUser then NumberOfSampleIntervals :=
PositionCheckTemp;
    if PositionCheckTemp > StepsFromUser then NumberOfSampleIntervals :=
StepsFromUser;
  end;
  {Downward motion}
  if rgpDirection.ItemIndex = 1 then
  begin
    if ((MaxSamples - PositionCheckTemp) <= StepsFromUser) then
NumberOfSampleIntervals := (maxSamples - PositionCheckTemp);
    if ((MaxSamples - PositionCheckTemp) > StepsFromUser) then
NumberOfSampleIntervals := StepsFromUser;
  end;
  Result := NumberOfSampleIntervals;
end;

{Converting from centimeters to half-lambda steps.
The number of half-lambda steps which will be used
will be rounded off to the nearest integer value.}
function TfrmStepper.convertCmToSampleInterval(StepsNumberFromUser: String):
Integer;
var
  UserFloat:                  Single;
  StepFloat:                  Single;
  StepFormated:               String;
begin
```

```
  UserFloat := StrToFloat(StepsNumberFromUser);
  StepFloat := ((UserFloat)*10)/sampleInterval;
  StepFormated := FormatFloat('0', StepFloat);

  Result := StrToInt(StepFormated);
end;

{Changing stepping unit}
procedure TfrmStepper.cbxStepUnitChange(Sender: TObject);
var
  ButtonSelected:             Integer;
begin
  case cbxStepUnit.ItemIndex of
  0: cbxStep.Enabled := False;
  1: cbxStep.Enabled := True;
  2: cbxStep.Enabled := True;
  3: cbxStep.Enabled := False;
  4: cbxStep.Enabled := False;
  end;

  if cbxStepUnit.ItemIndex = 3 then
  begin
    ButtonSelected := MessageDlg('Are you sure?, This may take as long as 45
minutes.',mtCustom, [mbYes, mbNo], 0);
    if ButtonSelected = mrYes  then cbxStepUnit.ItemIndex := 3;
    if ButtonSelected = mrNo then cbxStepUnit.ItemIndex := 0;
  end;

  if cbxStepUnit.ItemIndex = 4 then
  begin
    ButtonSelected := MessageDlg('Are you sure?, This may take as long as 45
minutes.',mtCustom, [mbYes, mbNo], 0);
    if ButtonSelected = mrYes  then cbxStepUnit.ItemIndex := 4;
    if ButtonSelected = mrNo then cbxStepUnit.ItemIndex := 0;
  end;
end;

{Function which checks the RESET signal of the MCU}
{If reset is enabled, TRUE is returned}
function TfrmStepper.checkMCUReset(): Boolean;
var
  MCU_reset:         Boolean;
  ResetReceived:     Boolean;
begin
  MCU_reset := False;
  ResetReceived := checkAnalogInput(AIOReset);
  if ResetReceived = True then MCU_reset := False;
```

```
  if ResetReceived = False then MCU_reset := True;

  Result := MCU_reset;
end;

{Function which checks if analog input is HIGH or LOW value}
{Returns TRUE if HIGH, and FALSE if LOW}
function TfrmStepper.checkAnalogInput(Channel: Integer): Boolean;
var
  ChannelTemp:     Integer;
  Temp:            Boolean;
begin
  Temp := True;
  ChannelTemp := Channel;
  ULStat := cbAIn(BoardNumA, ChannelTemp, Range, DataValue);
  if ULStat <> 0 then exit;
  if DataValue >= 2500 then
  begin
    Temp := True;
  end;
  if DataValue < 2500 then
  begin
    Temp := False;
  end;
  Result := Temp;
end;

{Function which updates the temporarly variable PositionCheckTemp}
function TfrmStepper.updatePositionCheckTemp(PositionCheckTemp: Integer):
Integer;
var
  Temp:          Integer;
begin
  Temp := 0;
  if rgpDirection.ItemIndex = 0 then Temp := PositionCheckTemp - 1;
  if rgpDirection.ItemIndex = 1 then Temp := PositionCheckTemp + 1;
  Result := Temp;
end;

{That's all folks...}
end.
```

Mikrokontroller kildekode

```
            /****************************************/
            /***************C kode til***************/
            /******Versjon 2.1 av stepperstyring******/
            /****************************************/
            /*******Kode i sin helhet laget av:*******/
            /*************Erlend Langbach*************/
            /****************2004-2005****************/
            /*********erlend.langbach@fys.uio.no******/
            /****************************************/

/*************************************************************/
/*Når funksjonsgenerator prosedyren blir kalt er dette for å   */
/*kjøre ønsket distanse. Det sendes derimot med et kryptisk    */
/*tall. Dette tallet sier noe om steps som faktisk skal kjøres */
/*Dette tallet må regnes ut. I denne oppgave benyttes det en   */
/*girboks på 1:20, samt at for hver runde vil selve hydrofonen */
/*gå 2,5mm. Det skal benyttes en transducer på 120kHz,og det er*/
/*ønskelig med en oppløsning på bedre enn en bølgelengde - slik*/
/*at vi ønsker sampling hver 1/2-bølgelengde. Dette tilsvarer  */
/*hver 6mm.
                */
/*Med helstepping beveger motoren seg 1,8grader per step. Slik */
/*at en for at motoren skal gå en hel runde trengs det 200 steg*/
/*For at hydrofoene skal bevege seg 6mm, trenger motoren å gå  */
/*48 ganger rundt. Dette vil da tilsvare følgende antall steps:*/
/*                                                           */
/* ---------------------------------------------------------- */
/* |  1/1 steg   |   1/2 steg   |   1/4 steg   |   1/8 steg  | */
/* ---------------------------------------------------------- */
/* | 9600 steps  | 19200 steps  | 38400 steps  | 76800 steps | */
/* ---------------------------------------------------------- */
/*                                                           */
/*Den ovenforstående tabellen er inkludert hvis at hvis det    */
/*det blir nødvendig å omprogrammere kontrolleren til mikro-   */
/*stepping så kan dette greit la seg gjøres.                   */
/*************************************************************/

/*Inkluderer en del biblioteksfiler*/
#include <avr/io.h>
#include <avr/delay.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

/*Definere Funksjonsprototyper*/
void init_system(void);
void funksjonsgenerator(unsigned int,int);
void manualStep(void);
```

```
void autoStep(void);
void akselerasjon(void);
void deakselerasjon(void);

/*Main metode*/
int main (void)
{
  /*Kaller prosedyre for initialisering av systemet*/
  init_system();

  /*Systemet går inn i en evig løkke,
  slik at programmet baserer seg på
  eventuelle avbruddsrutiner*/
  for( ; ; ) {}

  return 1;
}

/*Prosedyre for initialisering av systemet*/
void init_system(void)
{
  /*PORTA settes som inngang*/
  DDRA = 0x00;
  /*Oversikt over hva portens pinner benyttes til:
    PA0 = \
    PA1 =  \
    PA2 =   |
    PA3 =   |= Buss til å overføre antall 1/2-bølgelengder
    PA4 =   |= som skal steppes i manuelt modus.
    PA5 =   |
    PA6 =  /
    PA7 = /
  */

  /*PORTB settes som utgang*/
  DDRB = 0xFF;
  /*Oversikt over hva portens pinner benyttes til:
    PB0 = Pulstog utgang
    PB1 =
    PB2 =
    PB3 =
    PB4 =
    PB5 = avsatt til kontrollsignal til PMd-1208LS
    PB6 = avsatt til kontrollsignal til PMd-1208LS
    PB7 = avsatt til kontrollsignal til PMd-1208LS
  */

  /*PORTC settes som utgang*/
  DDRC = 0xFF;
  /*Oversikt over hva portens pinner benyttes til:
```

```
    PC0 = Ferdig all bevegelse.
    PC1 = Man modus: Ferdig en syklus.
    PC2 =
    PC3 =
    PC4 =
    PC5 =
    PC6 =
    PC7 =
  */

  /*PORTD settes som inngang*/
  /*dedikert til inngående kontrollsignaler og avbruddsrutiner*/
  DDRD = 0x00;
  /*Oversikt over hva portens pinner benyttes til:
    PD0 = Manuel/Automatisk styrings signal
    PD1 =
    PD2 =
    PD3 = INT1
    PD4 =
    PD5 =
    PD6 =
    PD7 =
  */

  /*Setting av Status Register se side7 i ATmega16(L) datablad.
    Setter I-bit høy. Følgende kode SREG = (1<<I) burte fungert,
       men det kan hende I er reservert på en eller annen måte*/
  SREG = 0x80;

  /*Rising Edge på SIG_INTERRUPT1 & SIG_INTERRUPT0 (external interrupt 1 og
0) gir oss avbruddet
  Setting av MCU Control Register se side 66-68 i ATmega16(L) datablad*/
  MCUCR = (1<<ISC11)|(1<<ISC10)|(1<<ISC01)|(1<<ISC00);

  /*Setting av MCU Control and Status Register se side 66-68 i ATmega16(L)
datablad
  Her trengs faktisk ingen ting å settes eksplisitt høyt*/

  /*Setting av General Interrupt Control Register se side 66-68 i
ATmega16(L) datablad.
  Gjør klar for at eksternt avbrudd på INT1 kan benyttes*/
  GICR = (1<<INT1)|(0<<INT0)|(0<<INT2);

  /*Initialiserer utgangene til null*/
  PORTB = 0x00;
  PORTC = 0x00;
}


/*********************/
```

```
/****Avbruddsrutine****/
/*********************/

/*SIGNAL(SIG_INTERRUPT1) starter enten prosedyrene manualStep() eller
autoStep(),
spørs hvordan bit på PD0 på PIND er satt fra PMD-1208LS*/
SIGNAL(SIG_INTERRUPT1)
{
  /*Setter alle utganger på PORTB og PORTC lav*/
  PORTB = 0x00;
  PORTC = 0x00;

  /*Setter verdiene på PD0 på PORTD til verdien data*/
  unsigned char data;
  data = (PIND & (1<<PD0));// & er bitand
                           //burte dette vært PIND0 eller PD0???

  /*Sjekker så om verdien er høy eller lav.
    Hvis verdien er høy betyr det at programmet
       skal kjøres i manuelt modus, ellers skal det
       kjøres i automatisk modus*/
  if (data == 1) //dvs hvis manual styring er brukt i delphi...
  {
    manualStep();
  }
  else
  {
      autoStep();
  }
}


////////////////////
//****Stepping****//
////////////////////

/*Prosedyre for manuell stepping*/
void manualStep(void)
{
  /*Leser inn fra PORTA hvor mange
  halvebølgelengder som systemet skal gå*/
  unsigned char antallHalfLambda;
  antallHalfLambda = PINA;
  //int teller = antallHalfLambda;
  /*Setter så antall halvebølgelengder til en teller variabel*/
  unsigned char teller = antallHalfLambda;

  /*Hvis teller er større enn null vil systemet starte
  å generere pulstog*/
```

```
  if(teller > 0)
  {
    akselerasjon();

       /*Sier ifra til PMD-1208LS at 1/2 bølgelengde er kjørt,
       selv om i virkeligheten bare 1/4 er tilbakelagt. Dette gjøres
       for at man skal ha få et så nøyaktig posisjon som mulig hvis
       reset blir benyttet eller tilsvarende. En må også gjøre det
       på denne måten da PMD-1208LS ikke har inkorporert en ekstern
       avbrudshåndtering, å istede ligge å må lytte på signalene.*/
       PORTC = (1<<PC1);
       funksjonsgenerator(4113,65);

       /*Nå er 1/4 bølgelengde tilbakelagt. Sjekker så om systemet
       skal gå lengre. Hvis så starter nedtelling i while() løkka.
       PC0 bittet blir benyttet til å fortelle PMD-1208LS at man har
       tilbakelagt 1/2 bølgelengde.*/
    while(teller > 1)
    {
         PORTC = (0<<PC1);
         funksjonsgenerator(4800,65);
         PORTC = (1<<PC1);
         funksjonsgenerator(4800,65);
         teller = teller-1;
    }
       /*Går ut av løkka når man er ferdig med hele intervallet og
       kjører så siste rest av den første delen før while() løkka.
       Benytter PC0 signalet til å fortelle PMD-1208LS at ønsket
       distanse er tilbakelagt. Dette gjøres nå for å få timing
       korrekt.*/
       PORTC = (1<<PC0)|(0<<PC1);
       funksjonsgenerator(4113,65);
    deakselerasjon();
       PORTC = (0<<PC0)|(0<<PC1);
  }
}

/*Prosedyre for automatisk stepping*/
void autoStep(void)
{
  akselerasjon();
  funksjonsgenerator(8226,65);
  deakselerasjon();

  /*Nå trengs det å si ifra til Delphi at forflytning er gjort for neste
målesserie*/
  PORTC = (1<<PC0);

}
```

```
//////////////////////
//Funksjonsgenerator//
//////////////////////
void funksjonsgenerator(unsigned int teller,int delay_konstant)
{
  for(int i = 0; i < teller; i++)
  {
    PORTB = 0x01;
       _delay_loop_2(delay_konstant);
       PORTB = 0x00;
       _delay_loop_2(delay_konstant);
  }
}

/*Prosedyre for akselerering
  Bruker totalt 687 steps for å gå igjennom hele prosedyren.
  Dette er viktig å vite når man skal regne ut hvor mange
  steps man skal sende med i funksjonsgenerator() prosedyren
  ellers i programmet.*/
void akselerasjon(void)
{
  funksjonsgenerator(10,1000);   //~125Hz
  funksjonsgenerator(14,750);
  funksjonsgenerator(18,500);
  funksjonsgenerator(20,250);    //~498Hz
  funksjonsgenerator(25,200);
  funksjonsgenerator(30,150);
  funksjonsgenerator(40,135);    //~921Hz
  funksjonsgenerator(50,115);
  funksjonsgenerator(60,105);
  funksjonsgenerator(70,95);
  funksjonsgenerator(80,85);
  funksjonsgenerator(85,80);
  funksjonsgenerator(90,75);
  funksjonsgenerator(95,70);
}

/*Prosedyre for deakselerering
  Bruker totalt 687 steps for å gå igjennom hele prosedyren.
  Dette er viktig å vite når man skal regne ut hvor mange
  steps man skal sende med i funksjonsgenerator() prosedyren
  ellers i programmet.*/
void deakselerasjon(void)
{
  funksjonsgenerator(95,70);
  funksjonsgenerator(90,75);
  funksjonsgenerator(85,80);
```

```
  funksjonsgenerator(80,85);
  funksjonsgenerator(70,95);
  funksjonsgenerator(60,105);
  funksjonsgenerator(50,115);
  funksjonsgenerator(40,135);    //~921Hz
  funksjonsgenerator(30,150);
  funksjonsgenerator(25,200);
  funksjonsgenerator(20,250);    //~498Hz
  funksjonsgenerator(18,500);
  funksjonsgenerator(14,750);
  funksjonsgenerator(10,1000);   //~125Hz
}
```