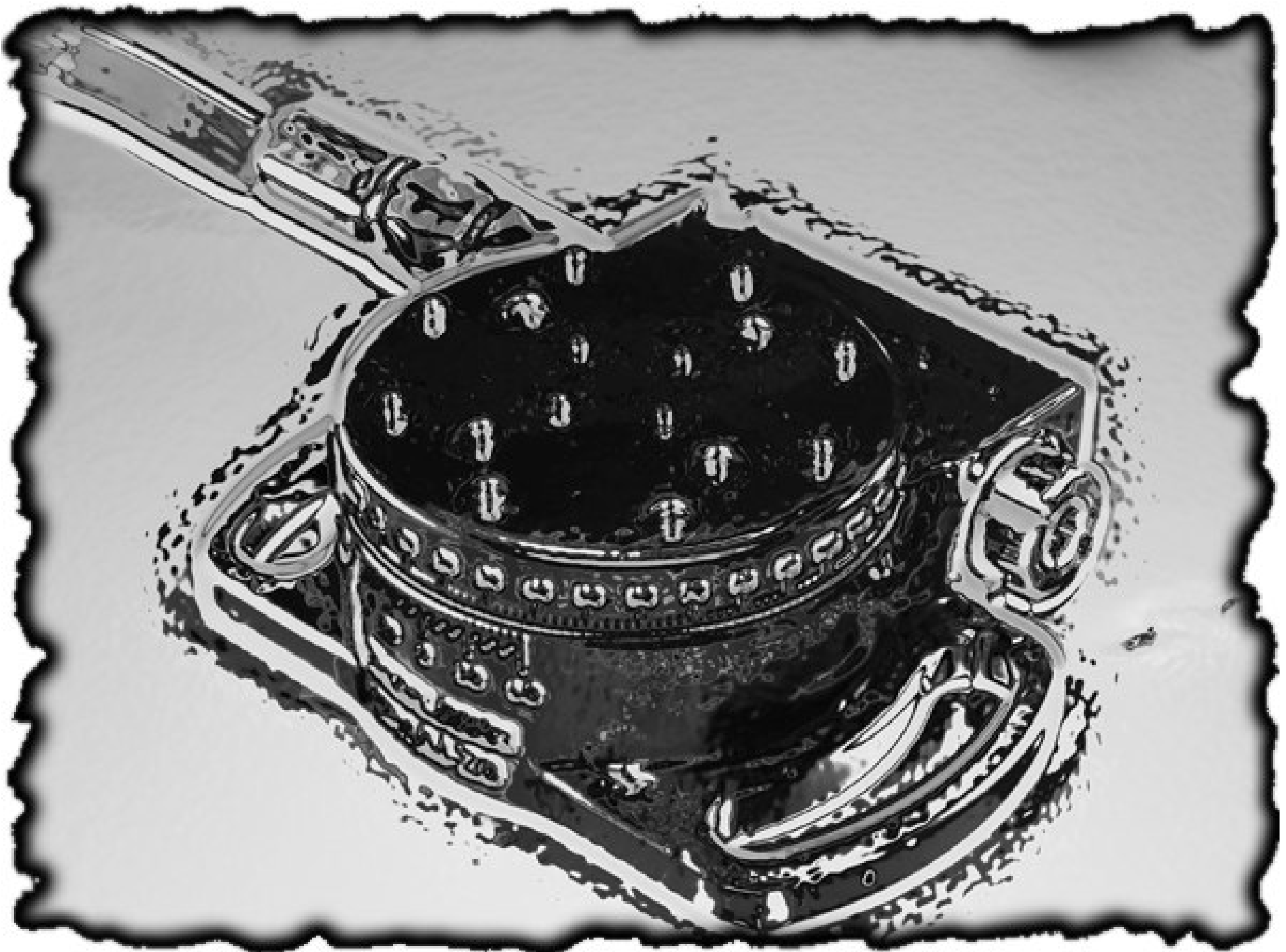
DTU

oticon
PEOPLE FIRST

Software API til at styre et array af DC servo motorer



af: Allan Frølund Riley (s001951)
Jakob Menander (s010811)

DTU vejleder: Stig Høgh
Oticons vejleder: Søren Valentin Stentoft

Afleveringsdato: 10. april 2006
Antal sider: CII

IMM – B. Eng. – 2006 – 16

Abstrakt

Vi har fået til opgave af Oticon at lave et api til styring og kalibrering af et array af motorer. Projektet er skrevet i Oticons domicil i Måløv og vi har i mindre omfang samarbejder med Oticons medarbejdere.

Api'et er skrevet i C++ og udviklet i Microsoft Visual Studio og ligger oven på det api der fulgte med hardwaret.

Undervejs i projektet har vi overvejet, designet og implementeret software såvel som hardware. Det er endt ud med at vi har lavet et api, som kan kalibrerer motorerne ved at tage et digitalt input fra en sensor. Ud fra det kan vi genererer et nulpunkt, som motorerne kan bevæge sig ud fra.

Api'et er lavet som en dll-fil, efter ønske fra Oticon. For at bruge api'et, skal man også bruge en lib-fil, to header-filer, samt det medfølgende api.

Vi syntes selv at det er gået ret godt. Vi har arbejdet godt sammen og er kommet frem til et godt resultat.
Api'et har nogle fejl, men vi har dokumenteret dem, og det burde nemt at rette i en videreudvikling. Til gengæld ligger præcisionen for kalibreringen ligger 125 gange under den ønskede præcision, hvilket vi er ret stolte af.

Alt i alt et spændende projekt, med et godt resultat.

Indholdsfortegnelse

ABSTRAKT 3

INDHOLDSFORTEGNELSE 4

BILLEDINDEKS 8

INDLEDNING 9

TAK TIL 9

MEDARBEJDERE PÅ OTICON 9

OVERVEJELSER 10

TIDSPLAN 10
DESIGN 10
MOVE/POSITION 10
KONFIGURERING 10
KALIBRERING 10
TRÅDE 10
DLL 11
TEST 11
FÆRDIG API 11
DEMO 11
PRINTMONTAGE 11
STATUS 11
ERROR 11
HARDWARE 11
MOTORER OG PCI100 BOARD 11
Slidestage 12
Turnstage 14
SENSOR 15
Krav 15
Test 15
Photointerrupter *15*
Reflective *16*
Valg 16
FORSTÆRKERKREDSLØB 16
UDVIKLINGSMILJØ 18
SOFTWARE 18
STAGE 18
BEVÆG STAGE 19
AKSESAMMENLÆGNING 19
BEVÆG AKSE 20
KALIBRERING 24

TRÅDE 26
ERROR OG ERROR HANDLING 26
DATASTRUKTURER 27
KONFIGURERING 27
STATUS 29
DLL 29

DESIGN 30

HARDWARE 30
FORSTÆRKERKREDSLØB 30
SENSORHOLDER 30
SOFTWARE 31
KODEOPBYGNING 31
KLASSER 32
OVERORDNET SOFTWARE DESIGN 34

IMPLEMENTERING 36

HARDWARE 36
PRINTET 36
SENSORHOLDER 36
SOFTWARE 36
INTERFACE 37
STRUKTURER 37
MOTOR KONFIGURATION 37
MESSAGES 37
TRÅDE 38
STATUS 39
BEVÆG EN STAGE 40
KALIBRERING 41
Kalibreringsalgoritme 41
BEVÆG EN AKSE 42
ERRORS 43

KLASSEBESKRIVELSER 44

MOTORCONTROLLER (INTERFACET) 44
STARTUPCONTROLTHREAD 44
CALIBRATESTAGE 44
CALIBRATEAXIS 44
SETCALIBRATION 45
SETCONFIGDEFAULT 45
SETCONFIG 45
GETCONFIG 45
SETCONFIGGUI 45
ENABLESTAGE 45
SHOWCONFIGDATA 46
MOVEAXIS 46

MoveAxisXYZA 46
MoveStageAbsolute 46
MoveStageRelative 47
MoveStageDegreeAbsolute 47
MoveStageDegreeRelative 47
GetStatus 47
GetStatusAxis 48
SetStatus 48
ResetStatus 48
MotorStructure 48
Arg 48
MotorConfiguration 48
SetDefault 48
SetConfig 50
GetConfig 51
SetConfigGUI 52
EnableStage 53
ShowConfigData 53
Messages 54
AddItem 54
TakeOutItemBack 55
Display 56
DisplayBack 56
Show 57
MotorThreads 57
T_ControlThread 57
T_MotorMove 59
T_MotorCalibration 60
MotorStatus 61
StatusContainer 61
GetStatus 62
GetStatusAxis 63
SetStatus 64
ResetStatus 64
MotorMove 65
MoveCalibration 65
MoveToX 69
MoveArcToDegree 70
MoveArcRelativeDegree 71
MoveRelative 71
MoveRelativeAjusted 73
MotorCalibration 74
CalibrateStage 74
CalibrateAxis 75
SetCalibration 76
AxisMove 77
CalculateNewPosition 77
MoveAxis 79
MoveAxisXYZA 81
ErrorManager 83
Translate 83

TEST 84

HARDWARE 84
STAGE PRÆCISION 84
FORSTÆRKNING AF SENSOR 84
SOFTWARE 85
PÅLIDELIGHED 85
REPETERBARHEDEN 85
REPRODUCERBARHEDEN 93

FORBEDRINGER OG FREMTIDIGE ASPEKTER 94

ERRORMANAGER 94
STAGEKØRSEL 95
MESSAGES 95
DATAINTEGRITET 96
BEVÆG SINGLE STAGE, MENS AKSEN KØRER 96
VED FEJL BLIVER STATUS IKKE ORDENTLIG RESAT 96
AUTOMATISK KALIBRERING AF TURNSTAGE 96
ABSOLUTE-ABSOLUTE SYSTEM. 97
DESTRUCTOR 97
SENSORHOLDER 97
SENSOR 97
TURNSTAGE 97
SAVE/LOAD KONFIGURATIONER 98
STATISK STATUS 98
STATUS POSTER 98
GRÆNSER FOR RELATIV KØRSEL 98
INTERRUPTS 98

KONKLUSION 100

LITTERATURLISTE 102

BØGER 102
HJEMMESIDER 102

SOFTWARE 102

BILAG 102

1. PROJEKTBESKRIVELSE 102
2. BRUGSANVISNING 102
3. API 102
4. KODE 102
5. HARDWARE 102

Billedindeks

Figur 1 - Tidsplan over projektet. 10
Figur 2 - PCI100 board med tilhørende servomotorercontrollere. 12
Figur 3 - Slide stage, med påmonteret slæde og sensorholder. 13
Figur 4 - Aktuatorens målemekaniske 13
Figur 5 - Aktuatoren. 14
Figur 6 - Turnstage. 14
Figur 7 - Til venstre kan man se reflektoren. Til højre ses photointerrupteren, som vi har valgt at bruge til vores kalibreringer. 15
Figur 8 - Diagram over forstærkerkredsløbet 17
Figur 9 - Primitiv montering af forstærkerkredsløbet. 17
Figur 10 - Stages sammensat til en akse. Den ene af de to stages vil have en negativ kørselsretning. 20
Figur 11 - JSP diagram af metode 2. Udregner nye positioner for stagene på en akse. 22
Figur 12 - Diagram over en tråds ”liv”. 26
Figur 13 - Forslag til en sensorholder. 31
Figur 14 - Klassediagram over vores api. 33
Figur 15 - Funktionsbeskrivelse af vores api. 34
Figur 16 - Færdigt monteret forstærkerkredsløb til otte sensorer. 36
Figur 17 - FIFO liste. 38
Figur 18 - Glasskalaen taget med zoom gennem mikroskop uden kryds, da krydset er monteret i linsen som kameraet erstatter 84
Figur 19 - 100 målinger. 86
Figur 20 - 100 målinger. 87
Figur 21 - 100 målinger. 87
Figur 22 - 100 målinger 88
Figur 23 - Histogrammer over målepunkter. 88
Figur 24 - 100 målinger af baggrundsbelysningen til venstre og steps til højre. 89
Figur 25 - 100 målinger med varierende baggrundsbelysning. 90
Figur 26 - Den varierende baggrundsbelysning. 90
Figur 27 - 100 målinger i total mørke. 91
Figur 28 - 51 kalibreringer. 92
Figur 29 - 32 kalibreringer i total mørke. 92
Figur 30 - Løsningsforslag til en errormanager. 94

Indledning

Vi har i forlængelse af vores praktik hos Oticon, fået til opgave at løse et problem[1] omkring styring og kalibrering af nogle servomotorer.

Oticon er en verdensomspændende virksomhed, der har specialiseret sig inden for høreapparater og har afdelinger i flere lande. Vi ved ikke hvad projektet skal bruges til og indeholder ikke nogle direkte referencer til høreapparater.

Kort fortalt er problemet, at det eksisterende stykke software der følger med motorerne, ikke gemmer positionerne når man slukker for systemet. Vores opgave er at gøre det muligt at fortsætte med at arbejde fra det samme udgangspunkt efter at systemet bliver genstartet. Systemet består af et antal motorer fra Thorlabs af typen Z6 med tilhørende DCX-MC110B boards tilsluttet et DCX - PCI100 kontroller board. Motorerne er monteret i stages[2] af typen CR1 eller PT1, hvor CR1 er en turnstage (drejestage) og PT1 er en slidestage (glidestage). En stage er det hus som motoren en monteret på. En turnstage er et hus med en skive på, som drejer når motoren kører. En slidestage er et hus med en slæde monteret på lejer inden i. Slæden kan bevæge sig frem og tilbage. Slæden bliver skubbet frem af motorens pal. Et sæt af fjedre sørger for at slæden sidder tæt på palen, og virker også til at trække slæden tilbage, når palen kører ind.

De samlede krav til systemet er at det skal kunne kalibrere en motor med en præcision på 0,1 millimeter inden for at begrænset tidsrum, alle otte motorer skal kalibreres simultant uden at belaste systemet betydeligt (der skal kunne udføres andre ting på computeren samtidigt).

For at kunne lave en kalibrering skal vi have et fysisk forhold til hvor stagen befinder sig – en måde at fortælle softwaren hvilken position stagen har, så den kan fortælle stagen dens position. En positionssensor med et interface til softwaren er det mest oplagte. Dette interface skal foregå gennem det medfølgende hardware. På DCX -PCI100 er der en digital input/output port. Hvis vi vil bruge denne, skal vi bruge en sensor der genererer et digitalt signal. Til styring og brug af denne sensor bliver vi nød til at have et sæt funktioner, et api der udvider funktionaliteten af det api som Thorlabs leverer med deres hardware. Det er vores opgave.

Tak til

Stig Høgh – For vejledning igennem projektet.
Søren Valentin Stentoft – For vejledning igennem projektet og formulering af tests.

Medarbejdere på Oticon

Jens Christian Sahl	–	For vejledning og problemstilling af projekt og motorer. Har desuden været vores kontaktperson omkring kravene til api'et.
Kasper Seidler	–	For hjælp til at designe og teste forstærkerkredsløbet mellem computer og sensor.
Flemming Dahl	–	For mekanisk løsning af fastgørelse af sensor på stages.
Ole Munk Plum	–	For hjælp til at finde brugbare sensorer.
VQS	–	For deres interesse omkring praktik og projekt, samt gode råd og vejledning.

[1] Se projektbeskrivelsen på bilag 1.
[2] Et navn for huset som motoren er monteret i.

Overvejelser

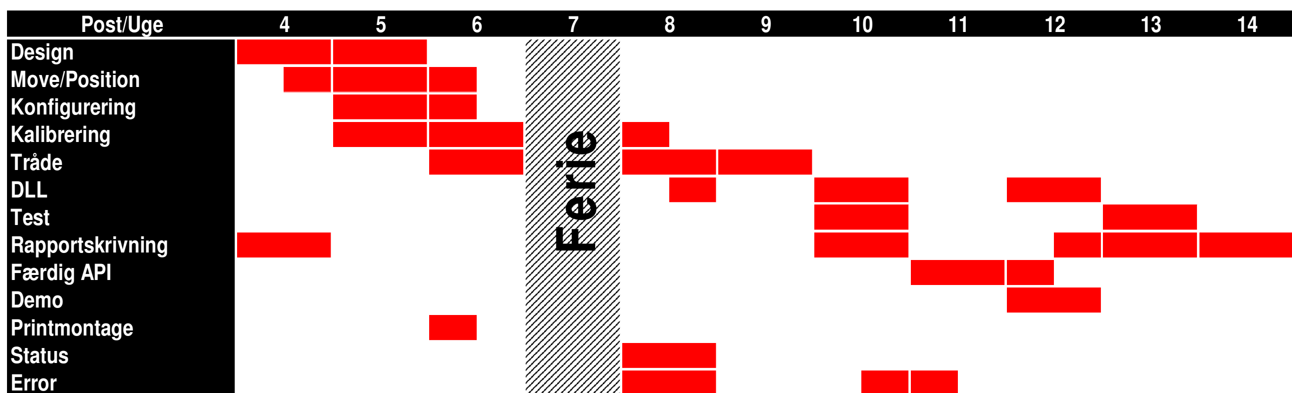
Det har været vigtigt for os at slå fast, at det er et api vi skal udvikle og ikke et program. Det betyder at vi ikke skal ende med et produkt man installere og så virker det hele. Et api skal bruges af programmøren, til at hjælpe ham med funktioner han ellers selv skulle bruge tid på at skrive. Med andre ord; Vi skal lave et api og programmøren skal derefter lave programmet.

Vores api skal endvidere bygge på det api, der følger med hardwaren.

Tidsplan

Vi har fra start lagt en tidsplan for projektet. Tidsplanen er delt ind i forskellige poster, som vi mener vi kan opdele projektet i. Vi har stort set fulgt tidsplanen, men har set os nødsaget til at lave små ændringer i tidsplanen undervejs. Selvom at vi har fulgt tidsplanen har den skredet nogle gange, mens vi andre gange har været lidt foran.

Ud over en tidsplan over hvornår vi skal påbegynde og afslutte delmål i projektet, har vi også aftalt møder med henholdsvis DTU og Oticon. Hver anden uge var der møde med DTU og hver anden uge var der møde med Oticon.



Figur 1 - Tidsplan over projektet.

Design

Design af klasser.

Move/Position

Skrive en klasser til at bevæge stages og akser.

Konfigurering

Skrive en klasse til at konfigurerer stagesne.

Kalibrering

Skrive en klasse til at kalibrerer stages og akser.

Tråde

At skrive tråde ind i vores andre funktioner og få dem til at virke.

DLL

At ligge alt kode ind i en dll-fil.

Test

Test af kode og pålidelighed.

Færdig API

Skrive softwaret sammen til ét api.

Demo

Lave et lille demoprogram der bruger vores api.

Printmontage

Få monteret komponenterne på printet.

Status

At skrive en statusklasse og implementere den i koden.

Error

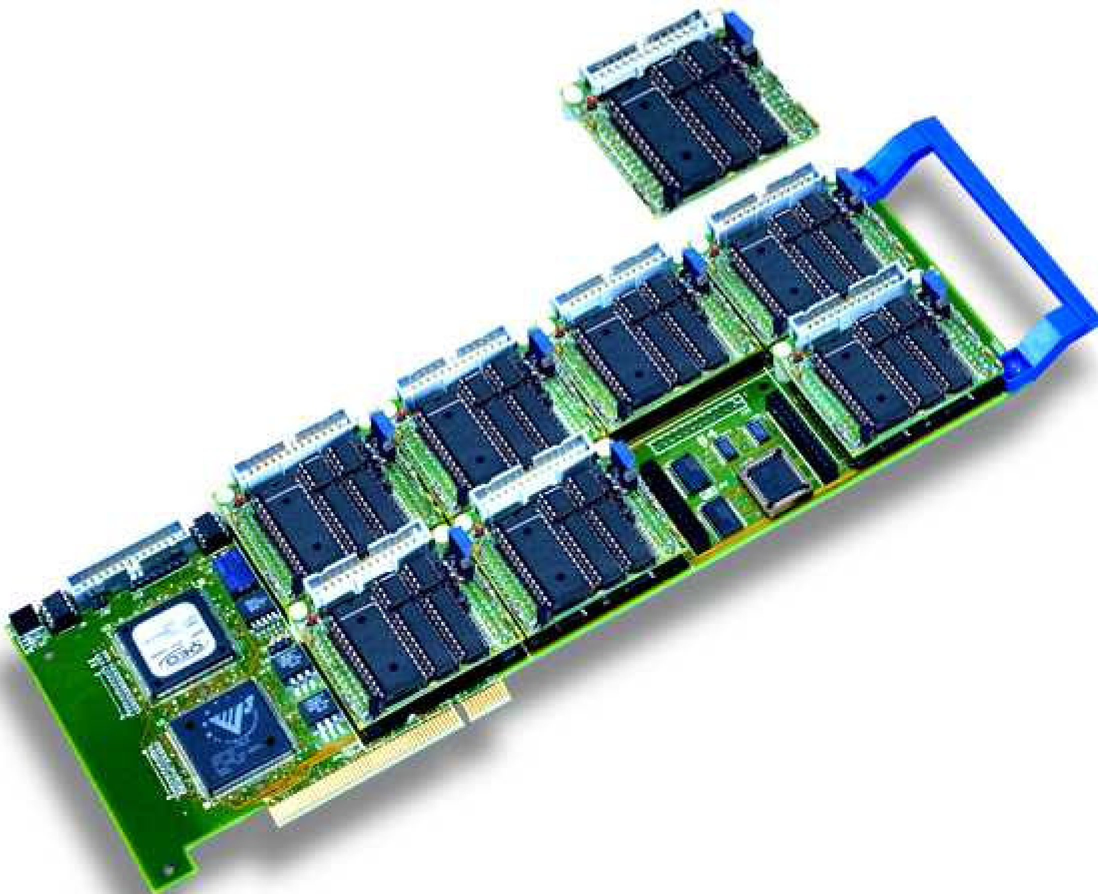
Lave en error-handler, som kan håndtere eventuelle fejl.

Hardware

Hardwaret består af et board til en pc, to typer stage, med tilhørende motorcontrollere, et forstærkerkredsløb, sensorer og sensoreholdere.

Motorer og PCI100 board

Dette er et indstiks kort til en pc, via en pci sokkel kan det tilsluttes en pc og styres via det tilhørende api, enten ved at skrive software til det eller med de medfølgende programmer, Til dette tilsluttes op til otte servo kontrol moduler.

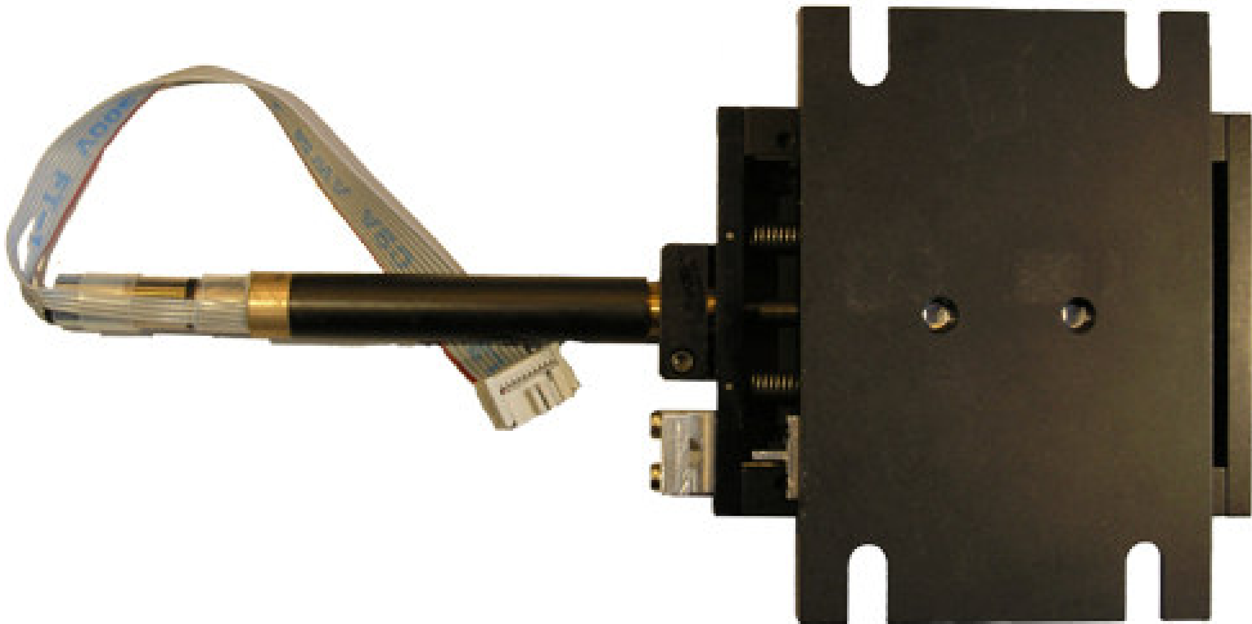


Figur 2 - PCI100 board med tilhørende servomotorercontrollere.

Den digitale I/O port er den som er situeret i øvre venstre hjørne på dette billede. Der er otte input og otte output ben, plus jord og VCC. Det medfølgende api indeholder funktioner til styring af de tilsluttede kontrolmoduler, servomotorer og digitale porte.

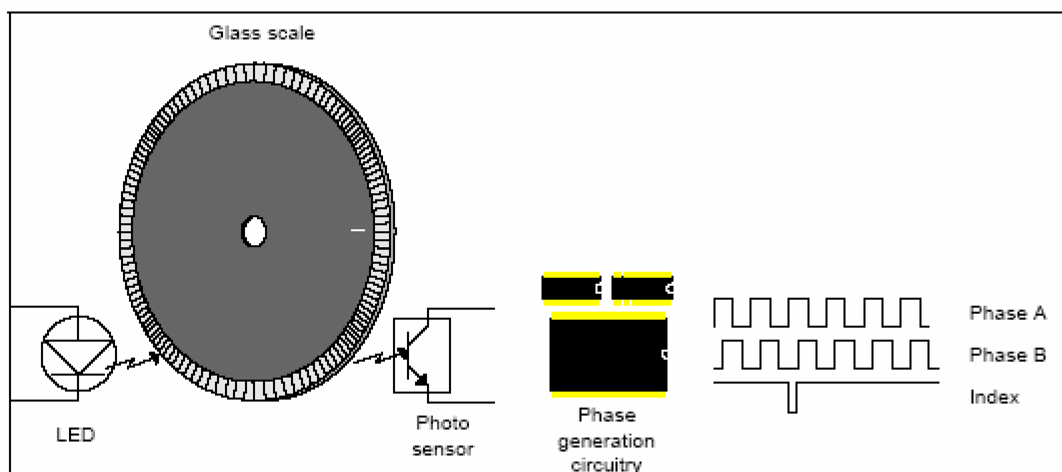
Slidestage

Den ene type servomotor vi har til rådighed, sidder på vores slidestage og har navnet Z625B.
En slidestage fungerer ved at aktuatoren drejer rundt og skubber en pal ud. Denne pal skubber en slæde frem. To fjedre sørger for at holde blokken helt op af palen og får den til at køre tilbage igen. Hvis modstanden på aktuatoren bliver større end den kan klare, slår den fra og skal slås til igen for at kunne køre videre.



Figur 3 - Slide stage, med påmonteret slæde og sensorholder.

Aktuatoren fungerer sådan at en sensor monteret på en glasskive genererer tre signaler. To signaler kommer med samme frekvens bare forskudt, bruges til at bestemme hvilken retning motoren bevæger sig i. Det tredje bruges til at bestemme "home" med, som indikerer at glasskalaen har drejet en hel omgang.



Figur 4 - Aktuatorens målemekaniske

Afhængigt af hvilket signal, A eller B, der kommer først, kan retningen bestemmes.
Med den indbyggede glasskala opnår denne at der er 12.288 steps og 0,5 millimeter når motoren har bevæget sig 1 omgang. Dette giver ≈ 0,00004069 mm per step[3].

[3] Cirka 40 nanometer.

Figur 5 - Aktuatoren.

Turnstage

Turnstagen har nummeret CR1/M-Z6. Motoren virker på samme måde som på slidestagen. Den store forskel på turnstagen og på slidestegen, ud over at den ene drejer og den anden kører frem og tilbage, er at turnstagen kan fortsætte den ene kørselsretning ud i det uendelige.

Figur 6 - Turnstage.

Turnstagen er delt ind i grader. Som det fremgår af bilag 5, er et enkelt step 2,19 arc sec. Da et arc sec er det samme som 1/3600 grader, betyder det at turnstagen drejer 0,0006083 grader pr step. Dette giver igen at der skal bruges 591.780,822 steps, for at dreje alle 360 grader. Da man kun kan rykke hele steps og ikke kan rykke en stage 0,822 step, må man tage stilling til om man skal runde op eller ned.

Sensor

For at kunne kalibrere noget som helst, skal vi bruge en sensor til at detektere stagens position. Dette skal være en binær sensor, der kan bruges sammen med den digitale port på kortet. Vi kiggede på et færdig sensor kit, men disse var for avancerede til vores formål og vi blev anbefalet at bruge en optisk sensor af Ole Munk Plum[4].

Hos RS og Farnell fandt vi to typer sensorer der kunne bruges til formålet. Den ene er en photointerrupter (RS: 455-0925), en såkaldt "gaffel"-sensor, mens den anden er reflektor (RS: 307-913), som detekterer et objekt ved at sende en stråle ud, som bliver reflekteret.
Refleksionssensoren reagerer på lys reflekteret tilbage til dens optiske transistor fra en diode og er meget følsom overfor baggrundsbelysning og farveskift. Hvorimod photointerrupteren reagerer på at en lysstråle mellem en diode og en transistor bliver brudt.

Krav

Motoren skal kunne kalibreres inden for 0,1 mm (hvilket svarer 2457,6 step for motoren). Sensoren skal derfor have en præcision på mindre end 0,05 mm, så der er noget at give af til hver side.

Test

Vi har testet to slags sensorer.
Testene gik forud for projektet og lavet med et primitivt forstærkerkredsløb.

Figur 7 - Til venstre kan man se reflektoren. Til højre ses photointerrupteren, som vi har valgt at bruge til vores kalibreringer.

Photointerrupter

Ved test med en gaffelsensor placeret i "absolut" mørke, har sensoren en præcision på ca. 5,6 µ meter. Derved er vi en faktor 10 under kravspecifikationen. Tillader man derimod at der er lys, vil resultatet variere alt efter hvor stor lysintensiteten er. Præcisionen daler markant.
Dermed kan sensoren godt bruges, forudsat at der ikke kommer lys til sensoren, når tappen skal blokere i gaflen.

[4] Medarbejder i Oticon.

Reflective

Testen med reflektoren viste at den var ubrugelig til formålet. Den er nemlig meget afhængig af overfladen der reflekterer og af omgivelsernes lys. Desuden er den ikke så god til at afstandsbedømme objekter, der nærmer eller fjerner sig. Den reagerer bedre på objekter der glider ind forbi sensoren, hvilket gør den praktisk talt umulig at montere på en stage.

Valg

Refleksionssensoren er altså meget ustabil og kan ikke give et brugbart resultat, mens photointerrupteren troligt skifter state når lyset blev brudt, og det er endda ved en lavere spænding end den ifølge databladet skal bruge.

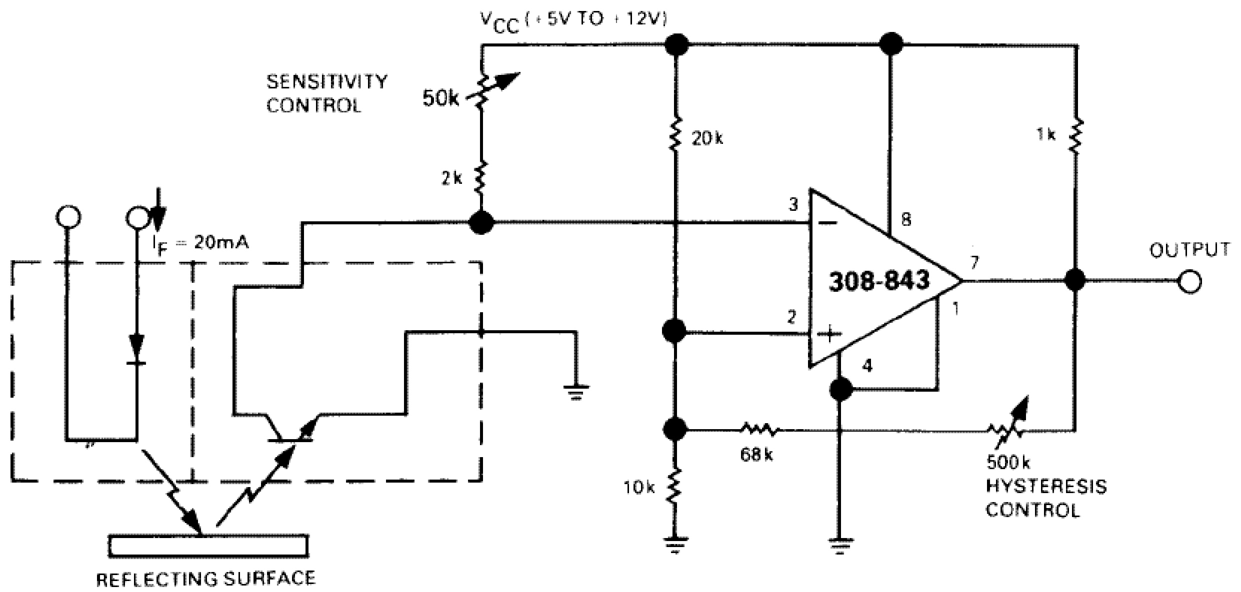
Vi har på baggrund af deraf valgt at bruge photointerrupteren, da den egner sig bedre til vores behov end reflektoren.

Efter at have samlet et testkredsløb, som er beskrevet i afsnittet herunder, og med en opstilling med photointerrupteren tapet fast til den faste del af stagen og en papirklips tapet til den bevægelige del, opnåede vi en præcision på ca. 120 nanometer med en spænding leveret fra boardet på 3.3 V. Sensoren skal egentligt bruge 5V.
Derfor vælger vi at fortsætte med denne, da vi antager at vi med en bedre montering og forstærkerkredsløb kunne komme i nærheden af den præcision vi gerne vil.
Dette viser at vi har taget det rigtige valg og at interrupteren er egnet til kalibreringen.

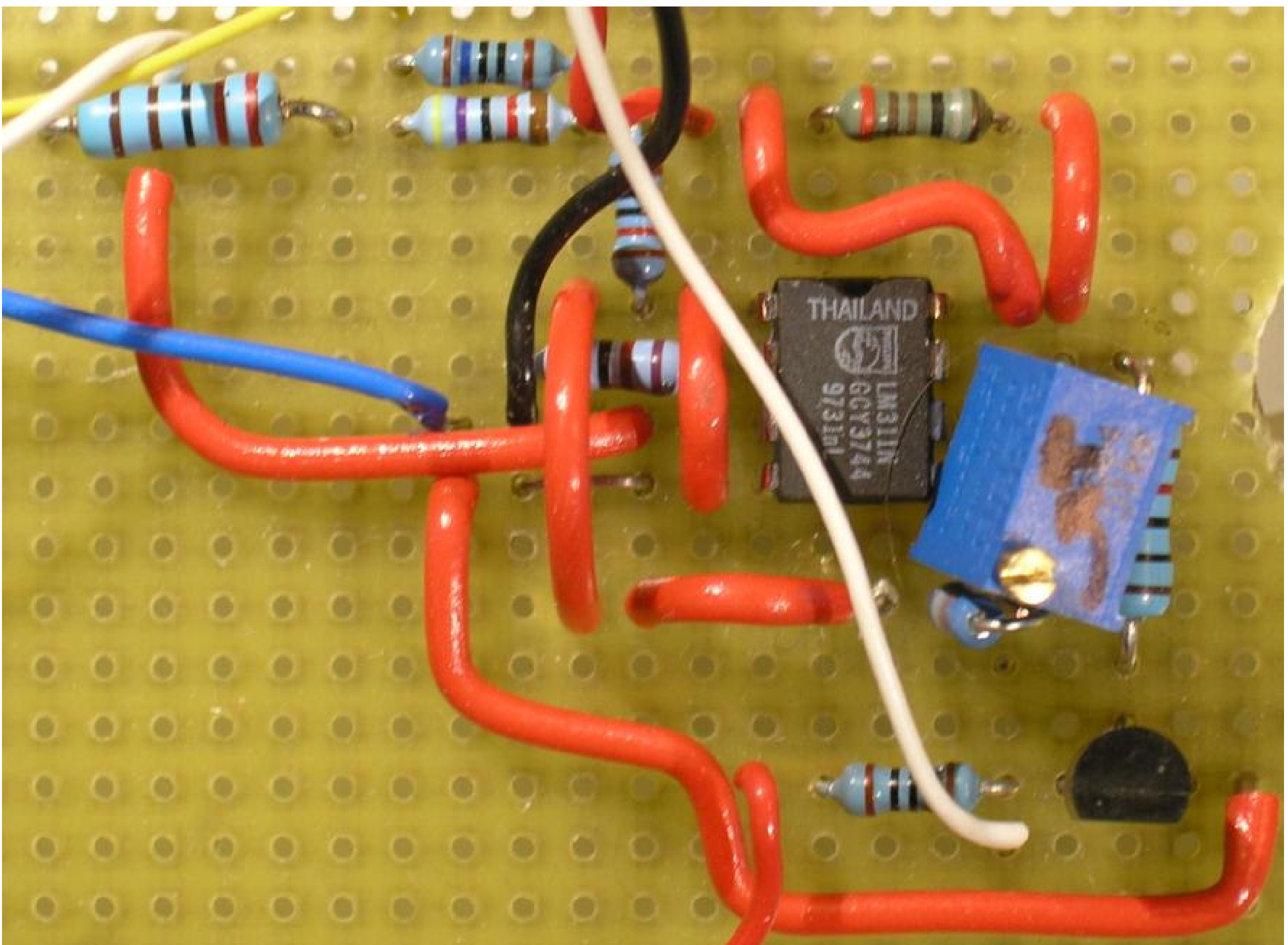
Forstærkerkredsløb

Til kalibreringsfunktionen skal der tages input fra en sensor monteret på stagen. Det medfølgende PCI100 board til motorstyringen har en digital port med otte input og otte output der kan bruges til formålet og det medfølgende api'et indeholder funktioner til at aflæse disse porte. For at få en aflæsning til kalibreringen skal signalet fra sensoren passe ind i det givne signal styrker for den digitale port, 0 volt og 3.3 volt for henholdsvis lav og høj på porten. Til dette skal der bruges et forstærker kredsløb.

Til reflektoren fandtes der et diagram til et forstærkerkredsløb, som kan bruges til begge sensorer. Vi har ud fra diagrammet, selv lavet et forstærkerkredsløb. Ved test og brug har vi fundet ud af at kredsløbet er tilfredsstillende, og vi har derfor haft fat i en af Oticons medarbejder, Kasper Seidler, som har hjulpet os med at optimere forstærkerkredsløbet og udlagt designet til en printplade.
Vi skal have monteret kortet og testet det, når det engang bliver færdigt.



Figur 8 - Diagram over forstærkerkredsløbet



Figur 9 - Primitiv montering af forstærkerkredsløbet.

Udviklingsmiljø

Oticon ønskede et api, som skal køre på en Windows maskine og være pakket i en dll. Api'et skal kunne bruges til at kalibrerer og styre deres motorer. Derfor lå valget af udviklingssprog på enten C, C++ eller C#, da vi ellers ikke vil have mulighed for at lave en dll-fil..

Da vi kiggede på det underlæggende api, som fulgte med boardet, kunne vi vælge mellem fire forskellige programmeringssprog: C, C++, Visual Basic eller Pascal. Valget var derfor ikke så svært at tage, eftersom ønsket var en dll-fil og vi desuden aldrig har arbejdet med VB eller Pascal. Vi gjorde os hurtig klart, at vi ville gøre brug af tråde og objekt orienteret programmering. Så der blev også C udelukket. Da api'et ikke understøtter C#, faldt valget naturligt på C++.

Valget var nok faldet på C++ alligevel, eftersom at vi er blevet oplært med C++. Heldigvis viste det sig også, at der fulgte flere programeksempler med til C/C++, end der gjorde til VB eller Pascal.

Valget af C++ betyder at vi har rettigheder til at skrive direkte i adresselageret, i modsætning til C# eller Java. Med andre ord, vi vil bruge unmanaged kode og ikke managed kode.
Managed kode er når der befinder sig et lag mellem det program du afvikler og cpu'en, en såkaldt fortolker. I tilfældet med C# er det .Net virtual machine der eksekverer koden. Denne er garant for hvad den kode der afvikles foretager sig. Fortolkeren indlægge forskellige sikkerhedsforanstaltninger i koden, så systemet holdes stabilt.
Fortolkeren laver koden om til noget der kan afvikles på det aktuelle system, med garanti for den kode der eksekveres er sikker.
Unmanaged kode er en færdigkompileret binær fil, der læses direkte ned i hukommelsen og afvikles. Når operativsystemet sætter programcounteren til at afvikle programmet holder operativsystemet ikke længere øje med hvad programmet laver.

Udviklingsmiljøet som vi vil udvikle i, bliver Microsoft Visual Studio. Da vi før har udviklet i version 6, var det naturligt for os at bruge Microsoft Visual Studio, blot i en nyere version.
Oticon bruger selv alle deres licenser, og vi vil derfor bruge en fuld 90 dages prøve version af Microsoft Visual Studio 2005 Professional Edition.
Vi har også haft kig på en fri Microsoft Visual Studio 2005 Express Edition, som godt nok ikke er en fuld version. Men da den ikke kan oprette dll-projekter, vil vi bruge prøveversionen. De 90 dage overskrider dog vores projektperiode.
Vi vil dog starte med at udvikle i Express Edition, eftersom Professional Edition skal bestilles hjem og først kommer 2-3 uger inde i projektet.

Software

Vi har diskuteret alle tænkelige aspekter omkring hvilke funktioner og hvad de skulle indeholde, for at kunne opfylde vores krav.

Stage

Stagen skal indeholde information om sin min og max værdi, udregnet ud fra kalibreringen af stagesne (i tilfælde med en turnstage er det fra 0° – 360°). Derudover skal den indeholde en retning på stagen i forhold til aksen, så hvis to stages sidder modsat hinanden på samme akse vil de stadig bevæge sig i samme retning. En stage der sidder i modsat retning vil eventuelt have et max der er 0 og en minusværdi som minimum.

Eksempel på de data en stage kan indeholde:

```
int stagenr;
int retning;
int type;
char axis;
double max;
double min;
```

Bevæg stage

Man skal både kunne rykke en stage absolut og relativt. Vi har overvejet at bruge det medfølgende api's funktion til at bevæge en stage absolut. Dette vil vi dog ikke gøre, da vi har en mistanke om at deres funktion alligevel bygger på deres funktion til at rykke relativt. Vi vil også gerne have en funktion, som skal korrigere, hvis stagen ikke har rykket den ønskede distance eller rykket for langt. Vi vil derfor lave en funktion der flytter en stage relativt, og som samtidig korrigere eventuelle fejlbevægelser. Det kan den eventuelt gøre ved at lave et rekursivt kald, hvor distancen den sender med, er den distance den mangler at køre.

For at rykke absolut, skal en funktion udregne distancen mellem den nuværende position og den ønskede position, for derefter at kalde den relative funktion.

Distancen og positionerne vi arbejder med her, er i steps. Vi vil dog gøre en undtagelse med turnstagesne. Til dem vil vi nemlig lave to funktioner der tager grader i stedet for steps. De skal så omregne graderne til steps og bruge den føromtalte funktion til at køre korrigeret relativt. Måden der skal omregnes på, skal være ved at gange konstant på den værdi man ønsker at omregne. Omregningskonstanten er fundet ved hjælp af de informationer, der fremgår i overvejelserne i hardwaren[5].

Vi vil også have en sidste funktion, som køre relativt. Den skal dog bare ikke korrigere, da vi vil bruge den til at kalibrerer med. Grunden til at kalibreringen ikke har brug for at korrigere dens position, er fordi det ikke er kritisk at kalibreringen kører hen til den korrekte position. Kalibreringen skal nemlig først bruge positionen, når det digitale input går højt.

Aksesammenlægning

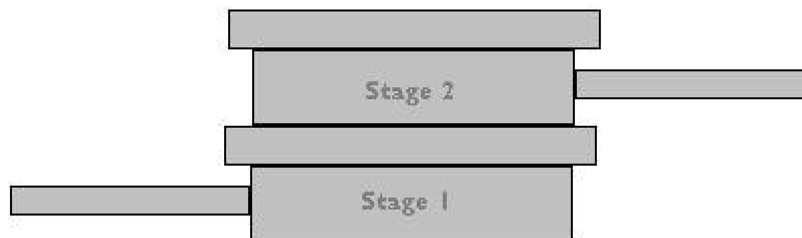
Sammenlægning af stages til akser, giver et mere dynamisk system at arbejde med. En akse består af en eller flere stages, der er koblet sammen. Vi har overvejet om vi skulle opfatte systemet som et rum bestående af et XYZ-plan, eller om vi skulle opfatte det som uafhængige akser, hvor kun brugeren kender aksernes indbyrdes forhold.
Fordelen ved at opfatte systemet som et XYZ-plan, er at man ikke behøver at fortælle hvor meget hver akse skal rykke sig, men kan nøjes med at sætte et bestemmelsessted bestående af en X-, Y-, Z-koordinat. Det vil være umuligt at gøre, hvis akserne er uafhængige af hinanden.
På den negative side ved at opfatte systemet som et XYZ-plan, er at det vil kræve en stor og kompleks udregning at holde styr på alle stagesne og positionere dem alle til bestemmelsesstedet. Systemet vil desuden sætte krav til rækkefølgen af stages. I den simple version, består systemet kun af slidestages. Her er det eneste man skal undgå, akser der er positioneret i to planer. Akserne skal være positioneret i X-, Y- eller Z-planet. Indgår der turnstages i opstillingen, skal de være placeret

[5] Se tidligere afsnit.

efter slidestagesne. Ellers vil turnstagesne kunne rotere akserne med turnstagesne, og yderligere komplicere systemet.

Ved nærmere overvejelse, og med vejledning fra de af Oticons medarbejdere der skal bruge systemet, har vi droppet ideen med at opfatte systemet som et XYZ-plan.
Vi vil i stedet opfatte systemet som et vilkårligt antal akser, der er uafhængige af hinanden. Dette giver også brugeren en større frihed til at definere de akser som han vil have, og en mindre risiko for at konfigurere systemet forkert.

En akse skal altså bestå af et antal stages. Disse stages skal kunne adderes sammen til akser, så to stages der bevæger sig i samme retning (eller modsatte) udgør en akse.



Figur 10 - Stages sammensat til en akse. Den ene af de to stages vil have en negativ kørselsretning.

Det vil være nødvendigt for brugeren manuelt at fortælle hvilken stage der hører til hvilken akse, dens orientering i forhold til de andre stages på aksen, dens minimum og maksimum, samt om det er en slidestage eller en turnstage.

Informationerne skal lægges som en datastruktur i en liste eller i et array. Bruger vi en liste, skal vi selv holde styr på pointerne der skal bruges til at lave listen. Hvorimod vi med array reducere os til et maksimalt antal stages.
Vi har holdt begge løsninger op mod hinanden og fundet løsningen med et array mest fordelagtig. Arrayet skal have størrelsen otte, da det maksimalt kan være otte motorcontrollere på et board. Dermed reducerer vi også vores brug af unmanaged kode mest muligt, da vi ikke skal have styr på pointerne i en liste. Problemet kan dog opstå, hvis vi kun ligger fire stages i array der er otte stort. Vi skal sørge for at man ikke kan vælge plads fem i arrayet, da der ikke ligger nogle stage. Dette kan undgås vel at bruge en counter, der fortæller hvor mange stages der ligger i arrayet, og så aldrig overskride den counter.
Ud fra arrayet skal man så udtage de relevante informationer, hver gang der skal udføres noget på en akse. Vi vil dermed ikke have et array for hver akse, men have ét array med alle stagesne i. Skal man derfor bevæge en akse, må man først finde alle stages der tilhøre aksen, for derefter at flytte dem.

Bevæg akse

Når man skal bevæge en akse, skal man finde ud af hvor langt hver enkelt stage i aksen skal bevæge sig. Det er umiddelbart tre teorier man kan vælge at implementere.

Den ene går ud på at fordele afstanden ud på motorerne på aksen. Derved vil man med sikkerhed holde sig inden for motorernes ydergrænser, og meget sjældent nå helt derud. En matematisk forklaring illustrerer nok bedre fremgangsmåden.

Lad os antage at vi har en akse bestående af tre stages. Alle tre stages har en maksimal position på 10 og en minimum position på 0. Aksen kan derfor befinde sig i intervallet [0:30]. Vi har valgt at lade de tre stages starte på positionerne: 2, 6, 9. Aksen befinder sig dermed på position 17. Den ønskede position for aksen sætter vi til 27. Regnestykket vil så se således ud:
Stagesnes grænseværdier:

$$p_{\min} = 0$$

$$p_{\max} = 10$$

Stagesnes positioner:

$$p1_{start} = 2$$

$$p2_{start} = 6$$

$$p3_{start} = 9$$

Akse positionen:

$$p_{start} = 17$$

$$p_{slut} = 27$$

Udregning af X (distance per 'enhed'):

$$(p_{\max} - p1)X + (p_{\max} - p2)X + (p_{\max} - p3)X = p_{slut} - p_{start} \Leftrightarrow$$

$$8X + 4X + X = 27 - 17 \Leftrightarrow$$

$$X = \frac{10}{13}$$

Udregning af de nye positioner for stagesne:

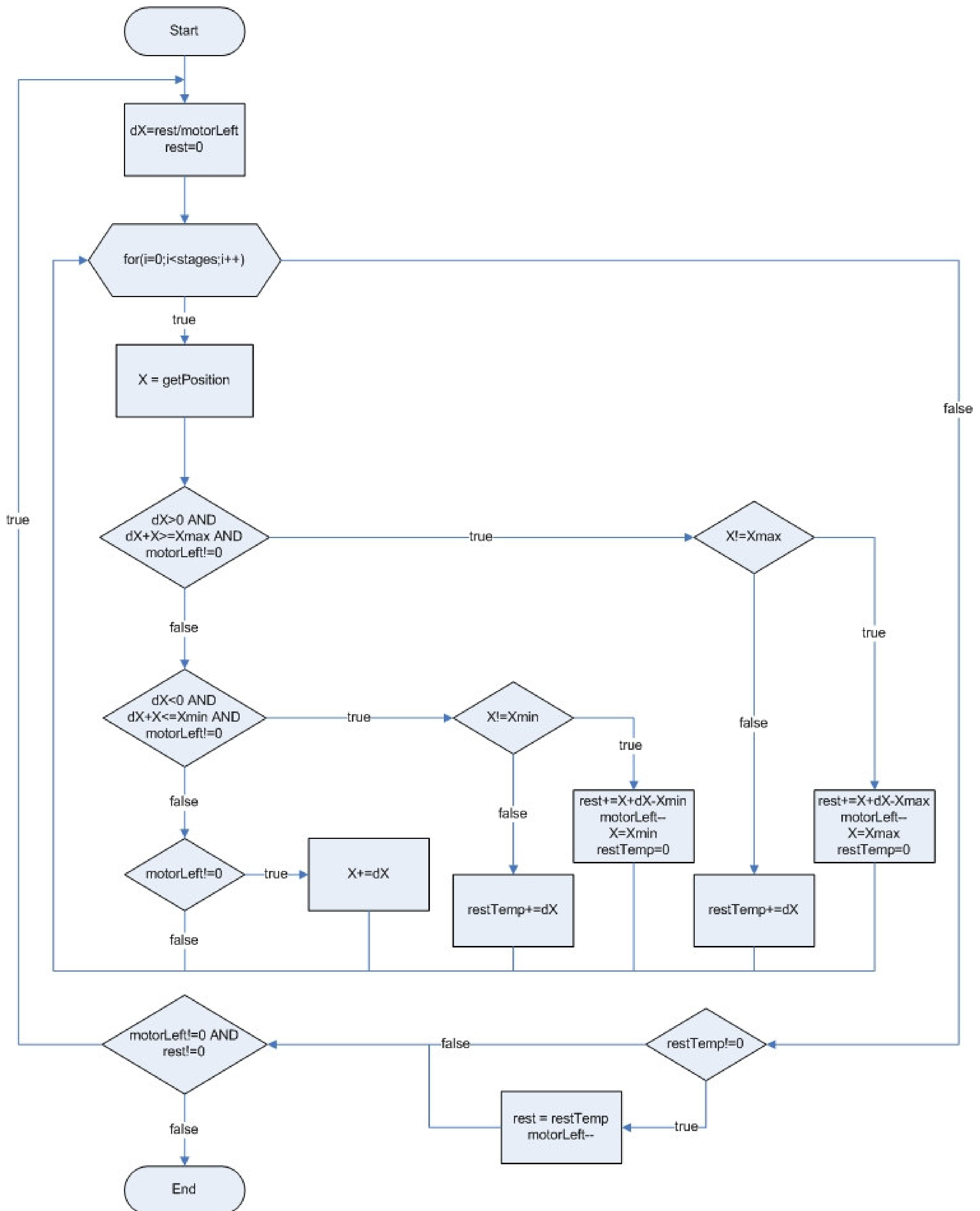
$$p1_{ny} = p1 + (p_{\max} - p1)X = 8,\overline{153846}$$

$$p2_{ny} = p2 + (p_{\max} - p2)X = 9,\overline{076923}$$

$$p3_{ny} = p3 + (p_{\max} - p3)X = 9,\overline{769230}$$

Vi skal dermed rykke stagesne til positionerne 8,15, 9,08 og 9,77 for at opnå aksepositionen 27. Derudover kan det også ses at stage 1 har rykket sig 6,15 enheder, stage 2 har rykket sig 3,08 enheder og stage 3 har rykket sig 0,77 enheder.

En anden metode man kan bruge, er at fordele distancen ligeligt mellem motorerne og lade hver motor køre lige langt, eller indtil de møder deres ydergrænse. Støder man på en ydergrænse, fordeles den manglende distance ligeligt på de resterende motorer. Fordelen ved at gøre det på den måde, er at man vil komme til den ønskede position, hurtigst muligt.



Figur 11 - JSP diagram af metode 2. Udregner nye positioner for stagene på en akse.

En matematisk gennemgang ud fra billedet, kunne se sådan ud: Vi antager at vi bruger stagene fra før, med samme positioner og ønskede position. Vi vil dog kun regne med en enkelt decimal, for at gøre udregningen mere simpel.
Først skal vi have rest, motorLeft og dX:

$$rest = p_{slut} - p_{start} = 10$$

$$motorLeft = 3$$

$$dX = rest / motorLeft = 3{,}3$$

Nu kan vi følge JSP diagrammet og får at positionerne bliver (⇒ er den distance der skal rykkes; = er den position der skal rykkes hen til):

$$p1 \Rightarrow 3{,}3$$

$$p1 = 5{,}3$$

$$p2 \Rightarrow 3{,}3$$

$$p2 = 9{,}3$$

$$p3 \Rightarrow 1$$

$$p3 = 10$$

Her ses det at p1 og p2 kan rykke deres andel af spektret, mens p3 kun kan rykke en enkelt enhed.
Vi kan nu regne rest, motorLeft og dX ud igen:

$$rest = 2{,}3$$

$$motorLeft = 2$$

$$dX = 1{,}2$$

Den resterende del som p3 ikke kunne rykke fordeles nu ud på p1 og p2.

$$p1 \Rightarrow 3{,}3 + 1{,}2 = 4{,}5$$

$$p1 = 6{,}5$$

$$p2 \Rightarrow 3{,}3 + 0{,}7 = 4$$

$$p2 = 10$$

$$p3 \Rightarrow 1 + 0 = 1$$

$$p3 = 10$$

$$rest = 0{,}5$$

$$motorLeft = 1$$

$$dX = 0{,}5$$

Da p2 ikke kan indeholde dens del af rest, som ses herover, må man igennem én gang til:

$p1 \Rightarrow 3{,}3 + 1{,}2 + 0{,}5 = 5$

$p1 = 7$

$p2 \Rightarrow 3{,}3 + 0{,}7 + 0 = 4$

$p2 = 10$

$p3 \Rightarrow 1 + 0 + 0 = 1$

$p3 = 10$

$rest = 0$

$motorLeft = 1$

$dX = 0$

Nu er rest nul, hvilket får løkken til at afslutte. Man kan se at den endelige akseposition er den ønskede position 27. Yderligere kan man se at den maksimale afstand der skal tilbagelægges er 5 enheder, hvilket er 1,15 enheder mindre end metode et. Dette betyder at flytningen tager kortere tid.

En sidste metode man kan bruge er, at lave en statistisk udregning over hvilke punkter der oftest bruges. Det kan måske være en fordel at lade en motor køre i modsat retning og lade de andre motorer køre lidt længere, hvis det bringer motorerne i en mere fordelagtig position til senere kørsler. Her har vi intet eksempel, da vi ikke har erfaringen med at bruge stagesne, eller tiden til at analysere bevægelsesmønstrene.

Om de tre scenarier, kan man sige at den første, nok er den mest sikre, da vi meget sjældent når ud til ydergrænserne. Scenario to er isoleret set for en enkelt kørsel den hurtigste, da hver stage kører sin optimale distance. Det sidste scenario er måske det hurtigste set over mange kørsler, da den vil prøve at optimere stangenes position på aksen.
Ud fra kendsgerningerne vil vi derfor prøve at implementere metode to. Vi ser det ikke som et problem at en stage når sin grænse, da vi vil sætte vores grænser et lille stykke fra stagens fysiske grænser. Vi tror heller ikke at vi kan nå at få en tilstrækkelig viden til at implementere metode tre.

For at kompensere for at vi ikke opfatter det som et rum, vil vi lave en (x,y,z,α)-funktion, som tager tre akser og en turnstage. Funktionen skal være meget enkel, da den bare skal kalde funktionen til at rykke en akse tre gange, og funktionen til at køre en turnstage én gang.

Kalibrering

Formålet med kalibrering er at stagen fra en tilfældig position skal kunne komme ned til et fast nulpunkt. Dette nulpunkt skal med minimal afvigelse være det samme for hver kalibrering. Til dette har vi monteret en sensor på slæden af slidestagen. Sensoren bruger et digitalt input på motorstyringskortet. Dette digitale input kan man tilgår via det api der følger med kortet fra producentens side.

Der er umiddelbart tre mulige måder at implementere det på. Enten ved polling, polling med Sleep() eller ved et hardwareinterrupt.

Ren polling er hvor vi hele tiden kalder en funktion, der henter staten på sensoren. Dette kunne ske i en while-løkke, som hele tiden henter informationer om sensoren. Hvis denne så har ændret sig skal programmet reagere. Et problem med denne metode er, at den kommer til at tage mange resurser fra

pc'en. Hvis man holder det sammen med at vi gerne vil køre flere kalibreringstråde samtidigt, vil de belaste systemet endnu mere.

Ved brug af funktionen Sleep(), kan man tvinge tråden til at afgive processoren. Derved opnår vi at computeren ikke bliver belastet på samme måde og vi kan køre flere tråde samtidigt uden at systemet låser for os.
Sleep ændrer trådens state til suspended i et givent antal millisekunder. Når den tid så er gået bliver tråden lagt over på waiting og eksekveret når kernen skifter tråd. Derved opnår vi at systemet ikke bliver belastet på samme måde som hvis vi kørte uden Sleep.

Den tredje mulighed er at få et hardwareinterrupt. Desværre understøtter hverken hardwaret eller det medfølgende api interrupts.
Havde vi haft adgang til et hardwareinterrupt når sensoren skiftede state ville vores software kunne bygges helt anderledes op. Vi ville så ikke være afhængige af polling men ville få at vide ligeså snart at sensoren ændrede state. Man kunne så lade interruptet gå direkte til den tråd der var ved at kalibrere stagen.

Den største prioritet ved interrupts, ville være at stoppe stagen ligeså snart sensoren ændrede state. Helt hvordan dette bedst kunne gøres er ikke til at sige da vi ikke har haft mulighed for at udføre test med interrupt. Så for at få adgang til test af dette skal der designes noget hardware der kan dette.

Ud fra det, vil vi prøve at implementere polling, hvor vi frigiver resurser med Sleep().

Vi har primær to forskellige stages og to forskellige måder at kalibrerer på. Slidestagen skal kunne kalibreres både automatisk og manuelt, mens turnstagen kun behøves at kunne kalibreres manuelt og har derfor ikke en sensor påmonteret.

For slidestagen skal det være muligt at kalde en funktion, som automatisk kan kalibrere og enten blive i det kalibrerede punkt, eller vende tilbage til udgangspunktet.

Eftersom vi har placeret sensoren for enden af slidestagen, må det ved automatisk kalibrering, være en fordel at kører hurtigt til sensoren. Når den når sensoren, skal den langsommere og mere præcist, køre ud og ind til og fra sensoren. Ved at logge hver gang sensoren aktiveres, kan man lave et gennemsnit af sensorens position, og derved få det mest præcise nulpunkt for stagen.

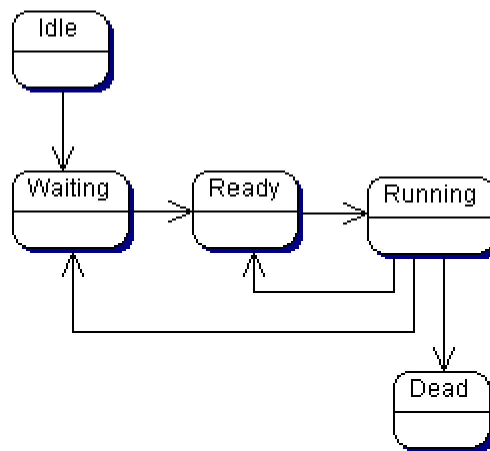
Kalibreringens succeskriterium er, at systemet har fået defineret et nulpunkt for systemet. Dette nulpunkt skal være det samme sted for hver kalibrering. Sensoren er monteret i samme side på alle stagesne. Stagesne skal derfor alle køre i samme retning ned mod sensoren og stoppe når de når sensoren. Hvis dette stop sker præcist samme sted hver gang, er kalibreringssystemet godt. Præcisionen er afhængig af om sensorern skifter state ved det samme punkt hver gang og om systemet er lige godt til at stoppe stagen når sensoren skifter.

Både slidestagen og turnstagen skal kunne kalibreres manuelt. Dette gøres ved at tilbyde brugeren en funktion til at sætte stagens nuværende position til en valgfri værdi.

Tråde

For ikke at blokere for programmet og nye kommandoer, skal vi bruge tråde i vores design. Vi skal have en main tråd til at holde styr på alle stagene. Derudover skal vi bruge to tråde, én til kalibreringen og én til at flytte stagesne på en akse. Dermed får vi et system, som kan håndtere flere stages på én gang. Vi skal dog være opmærksomme på at de enkelte tråde ikke optager så mange resurser, at der ikke er nok cpu kraft til de andre tråde.

Når en tråd ikke kører, ligger den enten som waiting eller suspended. Hvis den er waiting er den klar til at kører lige så snart kernen vil skifte. Ligger den som suspended vil den ikke komme til at køre før den bliver lagt som waiting.



Figur 12 - Diagram over en tråds "liv".

Interfacet må ikke være låst af funktionskald ind i api'et, men samtidig må et nyt kald ikke afbryde en kørende funktion. En mulighed vil være at lave maintråden som en form for shell, der fortolker et indgående funktionskald og starter funktionen i sin egen tråd. Via en status variabel, skal den følge med i hvor langt den er nået, før den begynder at udføre den næste kommando (en undtagelse er en eventuel stop funktion, der skal stoppe den igangværende funktion).

En anden måde er at lade interfacet have direkte adgang til funktionerne og så lade dem køre i hver sin tråd. Vi skal så sørge for at trådene er synkroniserede ved at bruge eksempelvis Windows semaphore og mutex. Igen skal det ikke være muligt at udføre en ny kommando på en stage før den er færdig med det den er i gang med (eller stoppet af brugeren)og igen skal det overvejes om tråden skal køre konstant, eller den skal oprettes hver gang noget skal udføres.

Videre test vil vise hvilken løsning der er den optimale.

Error og error handling

Hvis der undervejs i afviklingen af api'et sker en fejl, skal api'et reagere på denne fejl. Api'et skal enten rette den og/eller returnere at der er sket en fejl og informere brugeren om fejlen.

Returneringen skal foregå uanset om fejlen rettes, så brugeren kan se at der er sket en fejl og hvilken fejl der er tale om. Den mest optimale mulighed i vores øjne er, at returnere en boolean fra

funktionen der er opstået en fejl i. Returværdien fortæller at der er sket en fejl og så overfører selve fejlmeddelelsen via en pointer.
En eksempelkode, hvor fejlbeskrivelsen vil ligge i &fejlkode, kan se sådan her ud:

```
BOOLEAN succes Funktion(var1,var2,var3, &fejlkode);
```

På den måde kan vi teste på om funktionen er succesfuldt afviklet med en sand eller falsk værdi, og få selve fejlmeddelelsen til videre behandling som en tekststreng.

Det er vigtigt at der gives en fejlmeddelelse hvis programmet ikke kan udføre den givne operation og ikke kan fortsætte. Hvis det enten er en move funktion eller en kalibrering der ikke kan udføres, skal programmet fortælle brugeren at det ikke kan udføres og give grunden hertil. Hvis motorstyring ikke er åbnet eller stagen er optaget skal operationen ignoreres og brugeren skal have at vide hvad der skal til for at det kan udføres.
En fejlmeddelelse kan for eksempel se sådan her ud:

> ”operation kan ikke udføres, initialiser stage og prøv igen”
> ”operationen kan ikke udføres, stagen er i gang med noget andet, annuller, eller vent til den er færdig og prøv igen”

Datastrukturer

Vi har bestemt os for at samle vores data i strukturer og i stedet for at hardcode værdier i funktioner, bruge defines til værdier der skal bruges flere forskellige steder. Datastrukturer er både nødvendige og hjælper os en del.
Nødvendigeheden ligger i at den funktion vi bruger til at lave nye tråde kun tager pointers af typen void. Vi kan heldigvis typecaste denne type pointer til hvad vi gerne vil. Da vi skal bruge mere end et argument i de nye tråde, skal pointeren indeholde mere end værdi. Dette gøres ved at vi typecaster en strukturpointer til void, når vi opretter tråden. Inde i tråden kaster vi den så tilbage til den type vi kom fra, for at få adgang til dataene strukturen indeholder.

Vi har brug for at holde styr på en del information om de stages der er tilsluttet systemet. Den akse de tilhører, deres id nr. i forhold til hardwaren osv. Dette data kunne vi holde styr på ved at lave et array pr. variable der var otte stort så indeholdt data for stage nr. 1 og 2 osv.
Denne løsning ville være ret irriterende at arbejde med da vi, hver gang vi skulle bruge information om stagesne i en funktion skulle have alle disse arrays med over som argumenter. I stedet samler vi dem i en datastruktur som vi så laver et array af. Længden af arrayet har en fast værdi som vil gå igen flere gange i programmet, blandt andet i gennemløb af arrayet med for-løkker.
Derfor bliver dette defineret som en fast variabel vi vil kalde STAGE_ARRAY_MAX. På samme måde definerer vi andre værdier der vil gå igen gennem programmet. Vi behøver derfor kun ændre den ét sted, hvis vi vil ændre en af værdierne. Det kan spare os for en masse problemer med sammenligninger og ukorrekte datalængder.

Konfigurering

For at stagesne kan bevæge sig, skal de først konfigureres. Gør man ikke det, sker der intet når man sender kommandoer til dem. Da vi ikke laver et program men et api, skal vi ikke selv konfigurere stagesne, men gøre det muligt for brugeren at gøre det igennem vores api.

Brugeren skal have mulighed for at kunne konfigurerer de enkelte stages og få konfigurationen fra de enkelte stages. Vi vil yderligere gøre det muligt for brugeren af api'et, at gemme og læse konfigurationerne i en konfigureringsfil.

Konfigurationen direkte til motoren består af tre datastrukturer. Vi samler det hele i en struktur, der kommer til at indeholde alle de variable der skal indstilles for at motoren kører.

Disse data skal indstilles:

```
MCMOTIONEX motion;
motion.Acceleration = 100000;
motion.Deceleration = 0;
motion.Velocity = 9000;
motion.Torque = 0;
motion.MinVelocity = 0;
motion.Direction = 2;
motion.StepSize = 1;
motion.HardLimitMode = 3;
motion.SoftLimitMode = 3;
motion.SoftLimitHigh = 0;
motion.SoftLimitLow = 0;

MCFILTER filter;
filter.AccelGain = 0;
filter.DecelGain = 0;
filter.DerivativeGain = 2000;
filter.DerSamplePeriod = 0.001364;
filter.FollowingError = 1024.000000;
filter.IntegralGain = 40;
filter.IntegrationLimit = 50.000000;
filter.VelocityGain = 0;

MCSCALE scale;
scale.Constant = 0;
scale.Offset = 0;
scale.Rate = 1;
scale.Scale = 1;
scale.Time = 1;
scale.Zero = 0;

MCSetGain(hCtlr,axis,600);
```

Planen er at ligge dette ind i én datastruktur og derved kunne tilgå det på en nem og overskuelig måde.

Denne datastruktur skal så bruges som argument i en funktion der lægger det hele ned i motorstyringen. Udover dette, indeholder det medfølgende api et funktionskald til et grafisk interface til rette i de samme datastrukturer. Vi har bestem at for at gøre det muligt at bruge begge dele.

Det grafiske interface kræver at bruger selv går ind og ændre værdierne i det vindue den åbner og lukker det bagefter, hvorimod vores version kan hardcodes. Brugeren kan selv skrive funktioner til at bruge den.

Status

Som en service til brugeren og som en sikkerhed for os, har vi valgt at lave en statusklasse. Status klassen skal kunne gemme og fortælle om en motors status. For eksempel skal vi bruge en status, der fortæller om motoren er kalibreret. Dette kan være en nyttig viden for brugeren af api'et og skal desuden forhindre, at man starter motoren uden at kalibrerer først.

Af andet vi gerne vil have en status om, kunne for eksempel være: Om motoren er i gang med at kalibrere, om den bevæger sig eller positionen for motoren.
Vi skal også bruge en status for aksen. Kører en stage på akse X, betyder det at aksen bevæger sig og at man ikke skal flytte de andre stages. Man skal dog ikke kunne gemme status for en akse, da det gøres af den enkelte stage. Modtagelsen af status, skal ske ved at tage data fra de enkelte motorer der tilhøre aksen og sammenligne dem med hinanden.

For at gemme dataene, kan man enten lave en database eller gemme dem i en fil. Vi vil dog prøve noget helt andet. Eftersom det er et api vi skriver, vil vi ikke bede brugeren om at oprette en database eller tilføje en fil på harddisken. Derfor vil vi lave status statisk. Det vil betyde, at hver gang vi går ind i status funktionen, vil vi få den seneste værdi at arbejde med.

DLL

Et af ønskerne fra Oticon, er at api'et bliver pakket ned i en dll-fil. Vi har dog et problem med Express versionen af Visual Studio, da den ikke kan oprette dll-projekter. Problemet kan løses på to måder: Få en fuld version af Visual Studio, eller få oprettet et dll-projekt fra en fuld version og bruge den på vores Express version. Det fandt vi ud af, godt kunne lade sig gøre.
Vi tror det er fordi at Express versionen indeholder en stort set fuld C++, men at man bare har valgt at spærre nogle af valgmulighederne. Men man kan altså sagtens oprette et dll-projekt i en fuld version (2003 eller 2005) og hente den over i Express versionen. Vi vil dog prøve at få fat i den fulde prøveversion, som Microsoft stiller til rådighed.

At lave selve dll-filen, er ikke så svært. Hvis indstillingerne i udviklingsværktøjet står rigtigt (og det burde det gøre når man opretter et dll-projekt), skal vores interface klasse blot være af typen: `class __declspec(dllexport)`. Dette sørger for at vi har adgang til alle funktionerne i klasserne, når dll-filen er lavet.
Når man laver en dll-fil, vil der samtidig blive lavet en lib-fil. Denne lib-fil skal brugeren af vores dll-fil bruge, når han vil lave sit eget projekt. lib-filen skal altså bruges i hans projekt og dll-filen skal han bruge sammen med den exe-fil han vil få bygget.

Design

Designdelen af projektet har været vigtig for os. Det har hjulpet os til at kunne overskue projektet og strukturer vores arbejde. Eftersom vi arbejder to sammen har designet været vigtigt for at vi ikke bevægede os i hver sin retning, men holdt os på samme spor hele vejen igennem projektet.

Hardware

Der har ikke været så meget at designe på hardwaresiden, da det meste hardware var givet på forhånd og dette er jo også et softwareprojekt. Vi skulle dog alligevel overveje og designe hardwaret omkring sensoren, da det ikke følger med motorerne.

Forstærkerkredsløb

Med udgangspunkt i diagrammet på figur 8, har vi i samarbejde med en af Oticons medarbejdere, Kasper Seidler, udarbejdet et printkort med smd komponenter. Samarbejdet lå i at vi specificerede vores ønsker, og Kasper så designede printet til os.

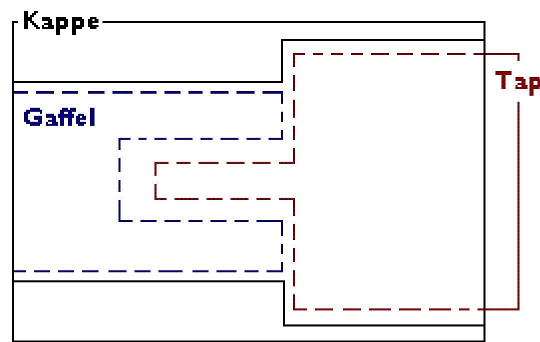
Vores ønsker til printet, lå i at få samlet otte forstærkerkredsløb, et til hver motor/sensor, på ét enkelt print. Kablerne mellem computer/print, strømforsyning/print og sensor/print, skulle ikke være loddet fast på printet, men kunne kobles af og på, for bedre mobilitet.
Derudover ville det være ønskværdigt med nogle lysdioder, som kunne indikere hvornår sensoren var aktiveret. Vi ville desuden gerne have at printet bestod af smd komponenter, da det ser pænere ud og ikke fylder så meget på bagsiden af printet.
Da DCX boardet kun leverer 3.3 V og sensoren skal bruge 5 V til dioden, har vi valgt at trække strøm fra en ekstern strømforsyning. Det digitale input bliver forbundet med de 3.3V hvis sensoren går høj, hvorimod resten af boardet bruger den eksterne strømforsyning.

Kasper forslog at han kunne tilføje en spændingsbegrænser, så systemet ikke led overlast. Samtidig ville han også tilføje nogle tomme komponentpladser på printet, hvor man eventuelt ville kunne forbedre kredsløbet. Forbedringerne ligger i at gøre operationsforstærkerne mere stabile og præcise, men det ville samtidig gøre dem langsommere. Med de ekstra komponentpladser, behøver man ikke at lave et helt nyt print.

Sensorholder

Det første udkast til en sensorholder, bestod af dobbeltklæbende tape og nogle bøjede papirklips. Denne løsning var langtfra optimal, så vi gik til Flemming Dahl[6] med et basis designkoncept. Vi gav ham frie hænder til at lave en holder og en tap til at bryde lyset i sensoren med.
Da belysning af tappen kan snyde sensoren til ikke at skifte stadie, tænker vi at man kan lave en kappe til sensoren, således at der ikke kommer lys ind til sensoren. En kappe kunne se således ud:

[6] En af Oticons mekanikere.



Figur 13 - Forslag til en sensorholder.

Gaflen ligger inde i en kappe, som går ud over tappen, så minimalt lys slipper ind og forstyrrer målingen. Kappen og tappens "krop" skal være minimum lige så lang som selve tappen. Dette vil forhindre at tappen når gaflen, før lyset er blevet afskærmet.
Den eneste ulempe kan dog være at gaflen skal sidde i yderpositionen af motoren, hvor motoren måske ikke er så præcis - eller sidde lidt inde, så man ikke udnytter hele motorens spekter (selvom det ikke er så meget man går glip af).

En anden mulighed er at lade tappen glide forbi gaflen. En kappe her vil ikke kunne lukke helt af for lyset. Til gengæld vil man kunne udnytte hele motorens spekter.

Software

Softwaredesignet er super vigtigt. Med et godt design og en god tidsplan, er man kommet et godt stykke af vejen.

Kodeopbygning

For at gøre koden mere overskuelig, har vi valgt at alle vores klasser skal deles op i en cpp-fil og en h-fil. h-filen skal indeholde klassebeskrivelsen, samt inkludere alle relevante filer. Selve funktionerne derimod skal ligge i cpp-filen. Før funktionerne i cpp-filen, skal vi lavet en kommentar, der fortæller lidt om variablerne funktionen tager med over, hvad funktionen returnerer og hvad funktionen gør.

Det er ikke alt vi vil ligger i klasser. I de tilfælde vi ikke gør det, har vi lagt dem i en h-fil. Det der ikke ligger i klasser er definitioner, strukturer og tråde.

For at holde rede på de forskellige navnetyper, som for eksempelvis funktions- eller variabelnavne. Har vi valgt at lave et lille regelsæt for navnens opbygning.
Klasse- og funktionsnavne har vi valgt skulle bestå af små bogstaver. Undtagelsesvis skal forbogstavet og nye ord starte med et stort bogstav. Navnet skal også hænge sammen, uden brug af bindestreg. Eksempel: TestNavn.

Variabelnavne skal ligne klasse- og funktionsnavnene, med undtagelse af at de skal starte med et lille bogstav. Eksempel: testNavn.

Strukturer og definitioner skal udelukkende bestå af store bogstaver og eventuelle nye ord skal skilles med et underscore. Eksempel: TEST_NAVN.

Tråde ligner funktionerne, de har bare fået tilføjer et ”T_” foran navnet. Eksempel: T_TestNavn.

Et eksempel for klassen ”TestKlasse”, med funktionen ”Test”, ville se sådan her ud:

TestKlasse.h:

```
class TestKlasse
{
private:
public:
    int Test(bool);
};
```

TestKlasse.cpp:

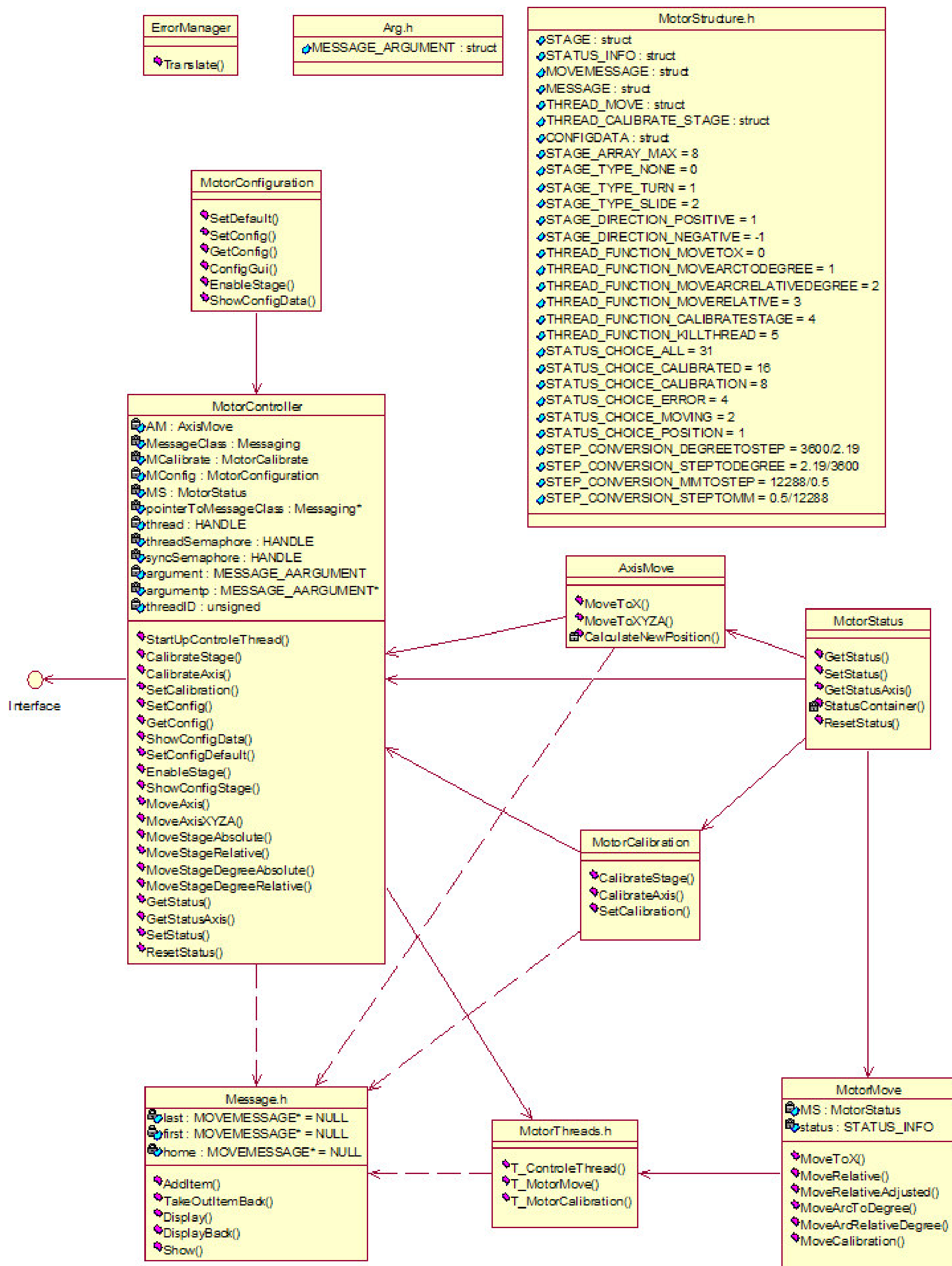
```
#include "TestKlasse.h"

/******************
** TestKlasse(bool xVar)
** bool xVar: an integer
** return: An integer value
**
** Calculate the power of x
******************/
int TestKlasse::Test(bool xVar)
{
    return xVar*xVar;
}
```

For at sikre os at vi ikke inkludere klasser, funktioner, strukturer eller definitioner flere gange, skal vi pakke dem ind i nogle ’#ifndef’[7]. På denne måde sikrer vi os, at vi kun inkluderer tingene én gang.

Klasser

Klassedesignet består primært af et klassediagram.

[7] ”If not defined”



Figur 14 - Klassediagram over vores api.

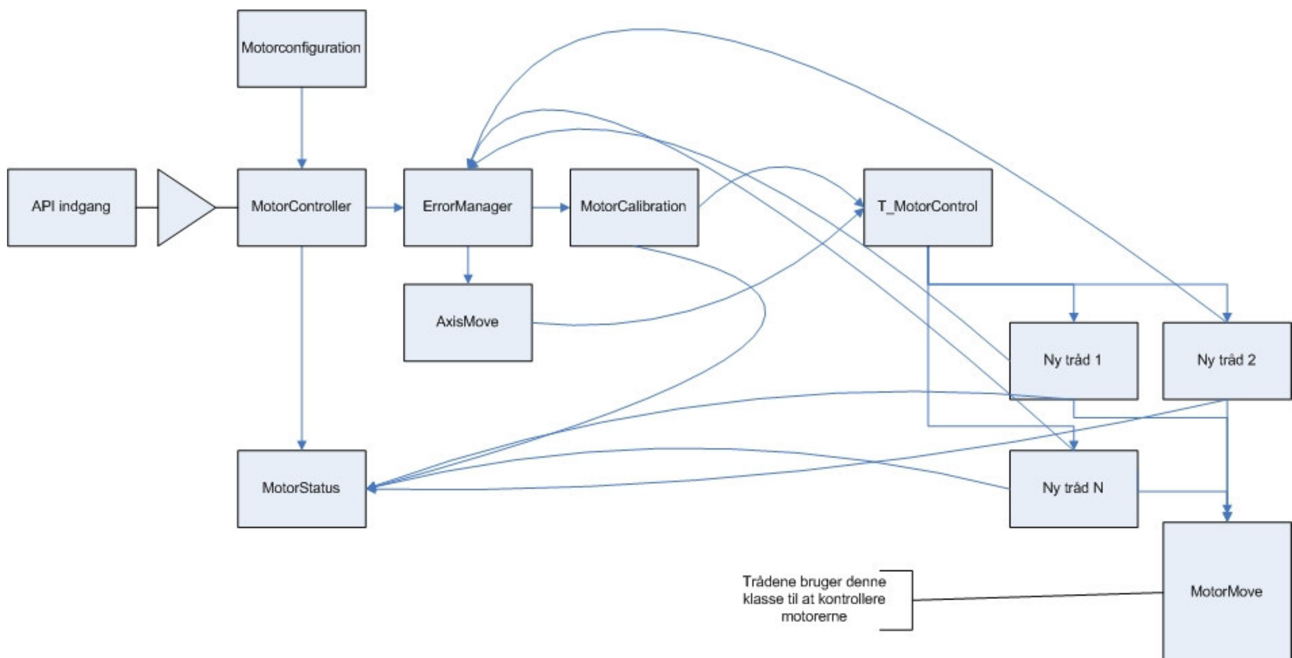
Klasserne som skal bruge funktioner og strukturer fra det medfølgende api, skal inkludere disse tre headerfiler:

```
#include <windows.h>
#include "mcapi.h"
#include "mcdlg.h"
```

Det medfølgende api ligger i mcapi.h og mcdlg.h, mens windows.h bliver inkluderet for at stille definitioner til rådighed for de to andre header-filer.

Overordnet software design

Dette er en kort beskrivelse af hvordan softwaren bliver opbygget. Vi vil ikke gå i detaljer med de enkelte funktionaliteter. Dette er altså beregnet til at give læseren et overblik over hvordan softwaren fungerer, for at gøre det nemmere at læse og forstå koden.



Figur 15 - Funktionsbeskrivelse af vores api.

Dette skema skal ses fra venstre mod højre hvor brugeren har adgang til systemet gennem api'et. Funktionerne i MotorController giver adgang MotorConfiguration og MotorStatus og error manager. Konfigurationsklassen indeholder funktioner til at konfigurere de forskellige stages, eller kan returnere en konfiguration til brugeren. Motorstatus initialiserer et statisk array af datastrukturer som indeholder statusen for stagesne. Denne bliver kontrolleret når en funktion prøver at få adgang til at bruge en stage.

ErrorManager er beregnet til at håndtere fejl, men fungerer ikke helt som tiltænkt på dette billede, da api'et ikke kan vente på, en funktion som kalibrering da det så ikke ville være muligt at sætte systemet til at kalibrere en anden stage eller akse mens den første er i gang.
Derfor afsluttes kaldene, før funktionen er udført (funktionerne bliver udført i en anden tråd, uafhængigt af den kaldene funktion), den kaldene funktion sender en besked til T_MotorControl,

med alle de nødvendige argumenter til afvikling af den ønskede funktion. T_MotorControl opretter så en ny tråd hvor den ønskede funktion afvikles i.

For at finde ud af om funktionen har lavet fejl, eller er i gang med en stage, sættes der variable i MotorStatus. Hvis der sker en fejl bliver det sendt til ErrorManager. Fejlen skal så sendes tilbage til api'et for at brugeren kan gøres opmærksom på, at der er sket en fejl.

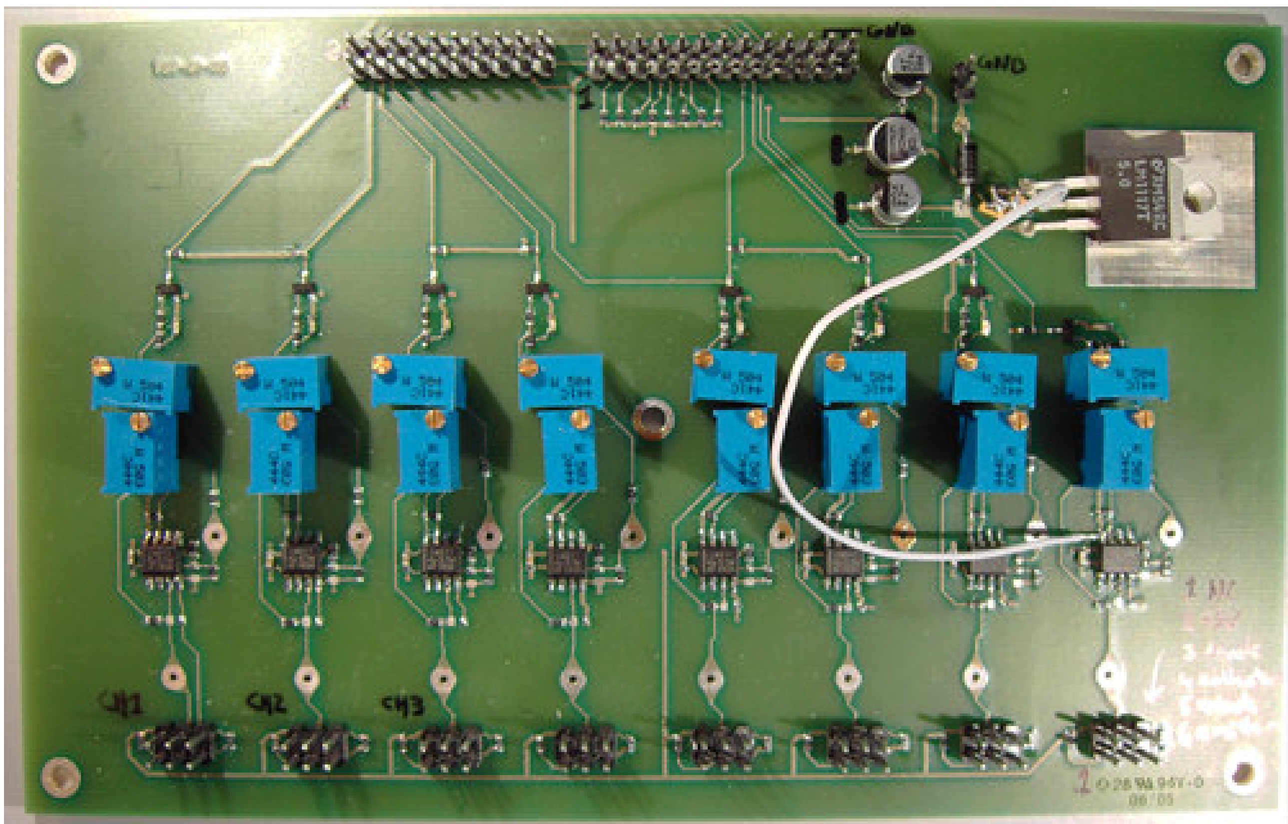
Implementering

Hardware

Implementeringen af hardwaren, kan begrænses til montering af printet og sensorholderen.

Printet

Vi har selv skulle loddet smd-komponenterne på printet.
Printet indeholder en lille fejl i, hvilket gør at de 5 volt fra den eksterne strømforsyning, ikke når ud i hele kredsløbet. Vi har derfor loddet en lus på, som forbinder "IC1" ben to med "U8" ben otte[8].
Fejlen er sket opstået i Kasper Seidlers design, som følge af en lille fejl i navngivningen.



Figur 16 - Færdigt monteret forstærkerkredsløb til otte sensorer.

Sensorholder

Sensorholderen er blevet fræset ud og sat på slidestagene af Flemming Dahl fra maskinværkstedet.

Software

Under implementeringen af softwaret, har vi siddet ved hver vores computer og udviklet. Vi har løbende skrevet vores kode sammen, hvis vi var afhængige af hinandens kode. Selve den egentlige sammenskrivning, er sket i slutningen af forløbet.

[8] Se eventuelt bilag 5.

Interface

Vores interface[9], består af de funktioner vi har vurderet til at være relevante for brugeren. Der sker som sådan ikke noget i funktionerne og størsteparten videresender brugeren ned i systemet. Der er dog undtagelser i de funktioner, der skal servicere brugeren med muligheder for at flytte en enkelt stage. Her bliver ikke kaldt en funktion som sådan, men oprettet en message som systemet skal fange og behandle.

Vi har også valgt at lade kontroltråden oprette i interfaceklassens konstruktør, samt at resætte alt i status til nul. At vi opretter kontroltråden her, betyder at brugeren starter kontroltråden, når han opretter en instans af klassen.
Klassen er omklamret af en `__declspec(dllexport)` for at lave en indgang fra dll-filen.

Strukturer

Vi har valgt at samle strukturer og defines i samme h-fil. For at beskytte koden mod multiple inkludering, har vi valgt at ligge en beskyttelse rundt om strukturerne og definesne.
En fordel ved at definere strukturer og defines uden for funktionerne, er at man så kan ændre koden et enkelt sted i koden, som efterfølgende har indflydelse på resten af koden. Alternativet ville være, at man skulle rette et utal af steder i koden.

Motor konfiguration

For at stagene skal kunne køre, skal de først konfigureres. Dette kan gøres gennem det api der følger med hardwaren, men der skal man kalde flere forskellige funktioner. Vi har valgt at samle disse funktioner i en, for at gøre det nemmere for os selv at konfigurere systemet og tilbyde brugeren denne mulighed. Af samme grund har vi lavet en funktion der indlæser nogle default værdier som vi ved at stagesne kan køre med, så vi ikke behøver at kode dette hver gang.

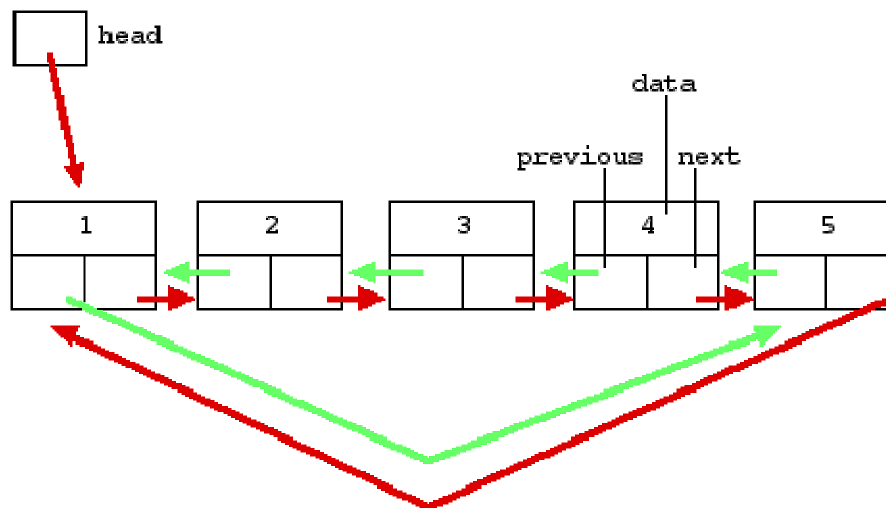
Messages

Denne klasse skal lægger beskeder i en hægtet liste og tager dem ud fra den hægtede liste igen. Vores liste er dobbelthægtet så man både kan tage data ud af listen forfra og bagfra. En hægtet liste kan indeholde forskellige typer for data, alt efter hvordan man laver den.

Vi har valgt at vores skal indeholde en datastruktur. Dette gør implementeringen nemmere, da vi kan ændre på indholdet af listen ét sted og ikke skal ind og rette flere steder i koden. Vi skal heller ikke tænke på antallet af argumenter når vi skal tilføje data til listen. Returnering af indholdet bliver også tilsvarende simplere.

Vi har brug for en såkaldt FIFO liste[10]. Det betyder at det vi første vi ligger ind i listen, også er det første vi skal tager ud. Vi har derfor lavet en dobbelt hægtet liste efter denne model.

[9] Interfaceklasse: MotorController
[10] " First In First Out"



Figur 17 - FIFO liste.

Listen har en begrænset størrelse så hvis der sker et problem i systemet kan den ikke vokse så meget, at den fylder hukommelsen op med beskeder der ikke bliver fjernet.

Funktioner:
Additem():Lægger et nyt objekt ind forrest i listen. Pointerne opdateres alt efter om listen er tom eller om der er noget i den i forvejen. Den ene pointer peger på det sidste objekt i listen og den anden pointer peger på den første.

TakeOutItemBack(): Returnerer og fjerner det sidste objekt i liste efter FIFO princippet

Display() og DisplayBack(): Udskriver pointernes værdier og bruges til at debugge. Debuggingen består af at man kan se om nye objekter bliver lagt på og taget ud af listen på korrekt vis, og om listen holder sin integritet under disse operationer.

Show(): Viser indholdet af det første objekt på listen. Bruges til debugging når dataene skal følges gennem systemet.

Tråde

Denne fil indeholder ikke en klasse men tre funktioner, der bruges til at afvikle tråde i. Grunden til at disse funktioner ikke er en del af nogle klasser, er at funktionen der laver nye tråde, _beginthreadex, skal have en pointer til adressen på funktionen. Af grunde vi ikke har fundet ud af, vil den ikke tage adressen til funktionen når den er en del af en klasse. Derfor er det implementeret på denne måde.

Api'et er bygget op over klasser der sender beskeder til kontroltråden om hvilke funktioner den gerne vil have udført. Disse funktioner starter kontroltråden så i deres egne tråde. Vi har konstrueret api'et på denne måde, for at undgå at brugeren skal vente på afviklingen af funktionerne. Her bliver der overhovedet ikke ventet på funktionerne, men kontrol tråden opretter bare tråde og venter så på den næste besked.

Når kontrol tråden modtager en besked, tager den variablen der bestemmer hvilken funktion der skal afvikles og hvilken "case" der skal udføres. Afhængigt af "casen", ligges variable i en datastruktur og en ny tråd starter med funktionen og argumenterne.

Kalibreringstråden frigiver synkroniseringssemaforen så der kan blive lagt en ny besked ind i messageklassen; men først efter argumenterne er blevet lagt ind i en lokal variabel.
Så kaldes kalibreringsfunktionen, som bliver udført i tråden. Når kalibreringen er færdig, succesfuld eller med en fejl, bliver tråden nedlagt.

Motormovetråden lægger også argumenterne over i en lokal variabel og frigiver derefter synkroniseringssemaforen. Den kalder den ønskede move-funktion med argumenterne, når funktionen er færdig nedlægges tråden.

Status

Som vi var inde på i vores overvejelser, har vi valgt at lave vores status statisk. Da man ikke kan lave en klasse statisk og vi havde problemer med at gøre funktionerne statiske, har vi valgt al lave en funktion, der indeholder en statisk variabel.
Funktionen fungerer som en container, der bliver kaldt hver gang man vil have status ud eller vil sætte status ind. Brugeren har ikke direkte adgang til denne container funktion, som er blevet gjort 'private' i klassen.

Fire andre funktioner, servicere brugeren og de andre klasser med ydelser omkring status. Vi har en funktion til at få status for en enkelt stage og en for en akse.
For at få status for en akse, har vi lagt alle stages på aksen, sammen med en række logiske operationer.
For at få statusen for om aksen er kalibreret har vi brugt &, da aksen først er kalibreret, når alle stages er kalibreret. Omvendt har vi brugt | når vi vil vide om aksen er i gang med at kalibrerer, om der er en fejl eller om aksen bevæger sig. Positionen for aksen findes ved at lægge alle stage positionerne sammen. Brugeren får statusen returneret igennem en struktur, som man så kan skille ad bagefter.
For at sætte en post i status, sender man en struktur ned med de ønskede ændringer. For at sikre os at man kun ændre på det relevante og ikke alt der ligger i strukturen, skal man også sende en integer værdi med. For at undgå at man sender en forkerte integrer værdi med, har vi lavet nogle defines, der beskriver det valg man tager. De mulige værdier ses herunder:

```
#define STATUS_CHOICE_ALL 31
#define STATUS_CHOICE_CALIBRATED 16
#define STATUS_CHOICE_CALIBRATION 8
#define STATUS_CHOICE_ERROR 4
#define STATUS_CHOICE_MOVING 2
#define STATUS_CHOICE_POSITION 1
```

Ønsker man at sætte flere poster i status, skal man blot lægge værdierne sammen. Ønsker man at sætte alle poster, kan man nøjes med at sætte `STATUS_CHOICE_ALL`.
Som man kan se er værdierne "binære" værdier, forstået på den måde, at de stiger som i det binære system. Dette hjælper os til at få fat i de rigtige ønsker. Ønsker brugeren nemlig at sætte posterne `STATUS_CHOICE_CALIBRATION`, `STATUS_CHOICE_MOVING` og

STATUS_CHOICE_POSITION. Bliver den samlede værdi 11. Vi kan nu finde ud af, hvilke poster der skal sættes, ved at se om posterne går op i værdien. For dette eksempel kan vi se at 16 ikke går op i 11 og derfor skal posten der fortæller at stagen er kalibreret ikke sættes. Derimod skal posten der fortæller at stages er i gang med at blive kalibreret sættes, da 8 går op i 11. Vi trækker nu de 8 fra de 11, og har dermed 3 tilbage.
Vi forsætter nu fremgangsmåden og ser at 'error' ikke skal sættes, men det skal 'moving' og 'position'. Vi har valgt ikke at lave en funktion til at sætte status for en akse, men kun for de enkelte stages. Det har vi gjort, for at holde det så simpelt som muligt.

En sidste funktion sørger for at sætte alt i status til nul. Dette gør vi brug af i vores interface, da det samtidig sørger for at der ikke ligger en tilfældig værdi i status.

Som det fremgår af kildekoden[11], har vi ingen pointere som argument i container-funktionen. Vi returnerer heller ikke status, men i steder om der er sket en fejl. Derfor undrer vi os også lidt over, at det alligevel virker. Vi har både prøvet at lade funktionen være statisk, og sende pointeres med som argumenter, men ingen af tingene lykkedes. Derfor har vi valgt at bibeholde den nuværende løsning, selvom at vi ikke er sikre på hvad det er, der får den til at virke.

Bevæg en stage

Vi har implementeret muligheder for at rykke en slidestage eller turnstage relativt eller absolut i steps, eller for at rykke en turnstage relativt eller absolut i grader. For at gå mellem steps og grader, har vi lavet nogle defines, som kan ses under strukturer i kildekoden.

Som det nederste lag har vi to funktioner, MoveRelative() og MoveRelativeAjusted(). Fælles for dem begge er, at de rykker en stage relativt. Forskellen ligger i at MoveRelativeAjusted() korrigerer dens position, og bruges til at køre en stage efter den har været kalibreret. MoveRelative() korrigerer ikke og bruges når man kalibrerer.

Da funktionen ikke bruges til andet end ved kalibreringen, er den også blevet specificeret til kalibrering. Blandt andet tager den en variabel. Den bestemmer om man skal vente på at stagen er færdig med at køre, eller om man skal forsætte i koden mens stagen kører. Funktionen tjekker også om stagen kører, når den bliver kaldt. Gør den det, returneres en fejl, ellers reserveres stagen ved at sætte move-posten i status og slippes igen i slutningen af funktionen.

MoveRelativeAjusted() justerer dens position, så man er sikker på at man når den ønskede position. Vi havde overvejet at lade den bruge et rekursivt kald, men da vi implementerede statustjek fungerede det ikke. Vi har derfor ændret det til en do-while-løkke, som kører så længe man er mere end ét step fra den endelige position. Vi starter med at tjekke om den pågældende stage er i brug, hvilket er grunden til at det rekursive kald ikke virker. Dette gør vi ved at tjekke status. Er den i brug, returnerer vi 1 for en fejl. Ellers reserverer vi stagen, og slipper den først igen efter do-while-løkken.
For at være sikker på at stagen er stoppet inden vi går videre, står vi og venter i en while-løkke (inden i do-while-løkken). Hvis stagen stadigvæk kører, lader vi den sleepe i et sekund, hvorefter vi tjekker igen. Skulle det ske at stagen ved en fejl står af, tjekker vi også for det, og returnerer 1 for fejl.

[11] Se bilag 4.

Vores funktion til at køre til en absolut position, MoveToX(), er ret simpel. Det eneste den gør, er at tjekke om den ønskede position, ligger inden for stagens grænser og finde den distance der skal køres, for at nå til den ønskede position. Funktionen bruger i sidste ende MoveRelativeAjusted() til at rykke stagen.

De to sidste funktioner til at bevæge en stage, er til at bevæge en turnstage. MoveArcToDegree() bevæger en stage absolut til en gradværdi. Den minder meget om MoveToX(), og den største forskel er da også at den tager en gradværdi med, som den regner distancen ud fra. Dette gør vi ved at lade gradværdien gange med konverteringsværdien[12] og trække den nuværende position fra. For ikke at flytte stagen med et kommatal, rundes distancen af til nærmeste heltal. MoveArcRelativeDegree(), som flytter en turnstage et relativt antal grader, er stort set identisk med MoveArcToDegree(). Små undtagelser gør at man kan rykke relativt i stedet for absolut. Begge funktioner bruges lige som MoveToX(), MoveRelativeAjusted() til at flytte stagen med i sidste ende.

Kalibrering

Der skal kunne kalibreres stages og akser af begge typer. Det er dog ikke et at der skal være automatisk kalibrering af turnstages. Det skal være muligt at sætte status for alle stages til kalibreret uden at køre kalibreringen, da det ikke er muligt at rykke dem med funktionerne før de er kalibrerede.

En kalibrering foregår sådan at stagen bevæger sig hen til et nulpunkt og stoppe der. Dette nulpunkt skal være det samme for hver gang systemet kalibreres med en præcision på 0,1 millimeter. Dette må ikke tage for lang tid, da det hurtigt kan blive et irritationsmoment for brugeren.

Vi kom frem til et maksimum på to. minutter før end alle stages i systemet skal være kalibreret. Dette er max kravene til systemet, disse skal overholdes for at systemet kan siges at være acceptabelt, vi vil selvfølgeligt gerne selv gøre det bedre end dette, med en minimal kalibrerings tid og en præcision på èt step, svarene til 0,00004 millimeter.
For at opnå dette, har vi have en kalibreringsalgoritme som bevæger stagen mod nulpunktet og stoppe stagen når den står præcis ved nulpunktet. Til dette har vi en optisk sensor som er monteret på stagen og tilsluttet på det digitale input på DCX kortet (nærmere beskrivelse af denne opsætning kan findes andet steds i rapporten).

Kalibreringsalgoritme

Algoritmen har to succes kriterier. Hvor lang tid det tager og hvor præcis den er. Med det hardware vi har udviklet, er det nødvendigt aktivt i softwaren at spørge om staten på sensoren. Vi er kommet frem til den løsning vi syntes bedst opfylder disse kriterier.

Algoritmen er bygget op så den starter med at undersøge om stagen er klar eller om den er optaget af fra andre funktioner. Er den ikke det optager algoritmen stagen til kalibreringen og er den undersøger det digitale input. Hvis staten på sensoren er høj, kan det betyde at stagen står så den afbryder sensoren, og derfor køres væk fra sensoren. Der køres en afstand så vi er sikre på at stagen

[12] Konverteringsværdien ligger i MotorStructures.h og er defineret til 3600/2,19 for at konvertere fra grader til step og 2,19/3600 for at konvertere fra step til grader.

ville være lav når kørslen er færdig. I samme løkke undersøges der om stagen overhovedet bevæger sig. Hvis den ikke gør det bliver der rapporteret en fejl og funktionen går ud. På samme måde hvis stagen kører færdig og staten på sensoren ikke er gået lav, antages det at der må være en fejl i opstillingen og der meldes fejl.
Hvis staten på sensoren fra start er lav eller den første løkke har kørt uden fejl begynder den første del af kalibreringen. Stagen sættes til at køre mod den ende vi ved at sensoren befinder sig i. Mens den kører undersøges det om sensoren skifter state. Det undersøges også løbende om stagen bevæger sig og om den er enabled. Når sensoren har skiftet state, stoppes stagen og der gøres klar til anden kørsel.
Stagen bevæges tilbage et stykke, så vi er sikre på at sensorens state har ændret sig til lav igen. Hastigheden sættes ned, og vi kører tilbage mod sensoren igen. Vi undersøger med samme tidsinterval om staten på sensoren ændres, og hvis den gør stopper stagen. Når stagen er stoppet er vi klar til tredje kørsel
Der rykkes kortere tilbage da anden kørsel giver et mere præcist stoppunkt end den første. Derfor behøver vi ikke bevæge os så langt væk fra sensoren for at være sikre på at dens state er lav. Hastigheden sættes igen ned så stagen bevæger sig endnu langsommere. Når stagen stoppes af sensoren, tages positionen og lægges sammen med de forrige målinger. Til sidst er alle målingerne lagt sammen i denne variabel, der så divideres med antallet af gentagelser.
Dette er det nulpunkt stagen skal have efter kalibreringen. Derfor rykkes stagen dertil og positionen for den sættes til nul. Nu kan brugeren så vælge om stagen skal køres tilbage til det punkt som den stod i før kalibreringen. Når dette er udført sættes hastighed og acceleration tilbage til de værdier de havde før kalibreringen begyndte. Status sættes til 'calibrated' og algoritmen er nu færdig.

Denne algoritme ligger til grundlag for kalibreringen, der kalibrerer en stage eller en akse; som jo bare er flere stages der skal kalibreres samtidigt.

Kalibrer stage:
Fungerer ved at lægge argumenterne fra funktionskaldet ind i et besked, lægge den på køen og frigive en semafor, så kontroltråden kan hente beskeden og kalde kalibrerings algoritmen.

Kalibrer akse:
Gør det samme som kalibrer stage, bare for alle de stages der ligger på den akse der skal kalibreres. Lægger argumenter over i en besked, ligger den på køen og frigiver semaforen. Dette gør den så for alle de stages, der tilhører den akse der skal kalibreres.

Sæt status for stage:
Da man ikke kan kalibrere en dreje stage eller det nogen gange kan være nødvendigt at køre stages uden at kalibrere dem, kan denne funktion ændre status til at være kalibreret og så sætte positionen.

Bevæg en akse

Vi har valgt at dele bevægelsen af en akse op i to dele. Den ene håndtere aksen overordnet, MoveAxis(), mens den anden, CalculateNewPosition(), finder ud af hvor meget den enkelte stage på aksen skal rykke sig, for at aksen når den ønskede position.
Efter at havde oprettet variable og initialiseret dem, tjekker vi om aksen er fri. Er den det, går vi videre og finder ud af hvilke stages der tilhører aksen. Ellers returnerer vi 1 for fejl. Samtidig med at vi finder de stages der ligger på aksen og lægger dem over i et lokalt array, tjekker vi også at det

kun er én type af stages der ligger på aksen. Der kan altså ikke både være slidestages og turnstages på samme akse.
Herefter regner vi de nye positioner ud for aksens stages. Hvis der altså overhoved er nogle stages på aksen. Det gør vi med udgangspunkt i figur 11. Før vi kommer ind på start i figuren, finder vi grænseværdierne for aksen og tjekker at vi ligger inden for. Vi finder samtidig ud af hvor langt der skal rykkes, førend aksen er på den ønskede position. Nu bliver den nye position (som hver stage på aksen skal rykke til) udregnet efter figur 11. Når udregningerne er færdige, har man altså et array med de positioner som stagesne skal rykkes, for at aksen får den ønskede position.

Nu laver vi et argument til hver enkel stage og opretter derefter en message ud fra argumentet. Grunden til at vi ikke lægger dataene direkte i en message, er at stagenes positioner ligger i et pointerarray og messagen også er et pointerarray. Vi er derfor nød til at gemme pointeren til positionen i en almindelig variabel, for at tildele pointeren til message værdien af den almindelige variabel. Vi sørger nu for en synkronisering, lægger messagen i en kø og frigiver en semaphore. Der vil nu blive oprettet en tråd, som får stagen til at køre igennem klassen MotorMove.
Vi har også implementeret en mulighed for at rykke tre akser og en turnstage på én gang. Brugeren kan selvfølgelig altid, via vores interface, skrive en kode og rykke flere akser op én gang. Det kan han gøre ved at fortælle systemet, at nu skal akse X rykke derhen og akse Y skal rykke derhen. Og faktisk er det lige præcist det vi har gjort. Vi kalder MoveAxis() tre gange og opretter en message til en turnstage.

Errors

De parametre der kan resultere i en fejl, er hvis der ikke er nogle stages der passer til aksen eller hvis den ønskede position ligger udenfor aksens maksimum/minimum. Sker der en fejl, returnerer vi 'NULL' og ligger en fejlmeddelelse i '&error'. Status skal sættes korrekt hvis en funktion ikke bliver færdig pga. fejl.

Klassebeskrivelser

Her vil vi beskrive klasserne linie for linie er opbygget. Se eventuelt også kommentarer i kildekoden på bilag 4.

MotorController (interfacet)

MotorController klassen er api'ets interface til brugeren. Det er de funktioner der ligger i denne klasse, man har adgang til efter at det er blevet til en dll-fil. Derfor er klassen defineret som: `class __declspec(dllexport) MotorController`, som betyder også at klassen bliver indgangen til dll-filen.
Klassen inkludere følgende filer, for at kunne tilgå funktioner, strukturer og defines:

```
#include <windows.h>
#include <process.h>    /* _beginthread, _endthread */
#include "Arg.h"
#include "AxisMove.h"
#include "Message.h"
#include "MotorCalibration.h"
#include "MotorConfiguration.h"
#include "MotorStatus.h"
#include "MotorStructure.h"
```

Vi vil ikke beskrive interfacets funktioner i dybden, da størstedelen af funktionerne refererer videre ind i api'et, hvor funktionerne bliver beskrevet.

StartUpControlThread

```
void StartUpControlThread()
```

Starter kontroltråden, som kommunikerer beskeder rundt i systemet. Det er kontroltråden der opretter undertråde til at bevæge motoren.

CalibrateStage

```
bool CalibrateStage(
         HCTRLR hCtlr,
         int stage,
         bool retur
)
```

Dirigerer brugeren længere ind i systemet med **`CalibrateStage()`**.

CalibrateAxis

```
bool CalibrateAxis(
         HCTRLR hCtlr,
         char axis,
         bool retur,
         STAGE stages[STAGE_ARRAY_MAX]
)
```

Dirigerer brugeren længere ind i systemet med **`CalibrateAxis()`**.

SetCalibration

```
bool SetCalibration(
         HCTRLR hCtlr,
         int stage,
         double position
)
```

Dirigerer brugeren længere ind i systemet med **SetCalibration()**.

SetConfigDefault

```
bool SetConfigDefault(
         HCTRLR hCtlr,
         int stage
)
```

Dirigerer brugeren længere ind i systemet med **SetDefault()**.

SetConfig

```
bool SetConfig(
         HCTRLR hCtlr,
         int stage,
         CONFIGDATA &data
)
```

Dirigerer brugeren længere ind i systemet med **SetConfig()**.

GetConfig

```
bool GetConfig(
         HCTRLR hCtlr,
         int stage,
         CONFIGDATA &data
)
```

Dirigerer brugeren længere ind i systemet med **GetConfig()**.

SetConfigGUI

```
bool ConfigGUI(
         HCTRLR hCtlr,
         int stage
)
```

Dirigerer brugeren længere ind i systemet med **SetConfigGUI()**.

EnableStage

```
bool EnableStage(
         HCTRLR hCtlr,
         int stage
)
```

Dirigerer brugeren længere ind i systemet med **EnableStage()**.

ShowConfigdata

```
void ShowConfigdata(
         HCTRLR hCtlr,
         int stage
)
```

Dirigerer brugeren længere ind i systemet med **ShowConfigdata()**.

MoveAxis

```
bool MoveAxis(
         HCTRLR hCtlr,
         char axis,
         double position,
         STAGE stages[STAGE_ARRAY_MAX]
)
```

Dirigerer brugeren længere ind i systemet med **MoveAxis()**.

MoveAxisXYZA

```
bool MoveAxisXYZA(
         HCTRLR hCtlr,
         char XAxis,
         double XPosition,
         char YAxis,
         double YPosition,
         char ZAxis,
         double ZPosition,
         char AAxis,
         double APosition,
         STAGE stages[STAGE_ARRAY_MAX]
)
```

Dirigerer brugeren længere ind i systemet med **MoveAxisXYZA()**.

MoveStageAbsolute

```
bool MoveStageAbsolute(
         HCTRLR hCtlr,
         STAGE stage,
         double destination
)
```

hCtlr: Et handle til at kunne tilgå stagesne.
stage: Den stage der skal flyttes.
destination: Den position stagen skal flyttes til.

Funktionen opretter en message , med de medsendte argumenter og lægger den i messagekøen.

```
MESSAGE beMessage;
MESSAGE* pointerToMessage;
pointerToMessage = &beMessage;

pointerToMessage->hCtlr = hCtlr;
```

```
        pointerToMessage->stageOnAxis = stage;
        pointerToMessage->destination = destination;
        pointerToMessage->function = THREAD_FUNCTION_MOVETOX;
        WaitForSingleObject(syncSemaphore,INFINITE);
        pointerToMessageClass->AddItem(pointerToMessage);

        ReleaseSemaphore(threadSemaphore,1,NULL);

        return 0;
```

MoveStageRelative

```
bool MoveStageRelative(
        HCTRLR hCtlr,
        int stage,
        double distance,
        HANDLE syncSemaphore
)
```

Opretter en message og lægger den i messagekøen. Se MoveStageAbsolute(). Eneste forskel er funktionsværdien.

MoveStageDegreeAbsolute

```
bool MoveStageDegreeAbsolute(
        HCTRLR hCtlr,
        int stage,
        double destination,
        HANDLE syncSemaphore
)
```

Opretter en message og lægger den i messagekøen. Se MoveStageAbsolute(). Eneste forskel er funktionsværdien.

MoveStageDegreeRelative

```
bool MoveStageDegreeRelative(
        HCTRLR hCtlr,
        int stage,
        double degree,
        HANDLE syncSemaphore
)
```

Opretter en message og lægger den i messagekøen. Se MoveStageAbsolute(). Eneste forskel er funktionsværdien.

GetStatus

```
bool GetStatus(
        HCTRLR hCtlr,
        int stageId,
        STATUS_INFO* status
)
```

Dirigerer brugeren længere ind i systemet med **`GetStatus()`**.

GetStatusAxis

```
bool GetStatusAxis(
         HCTRLR hCtlr,
         char axis, STAGE
         stages[STAGE_ARRAY_MAX],
         STATUS_INFO* status
)
```

Dirigerer brugeren længere ind i systemet med **GetStatusAxis()**.

SetStatus

```
bool SetStatus(
         int stageId,
         STATUS_INFO status,
         int choice
)
```

Dirigerer brugeren længere ind i systemet med **SetStatus()**.

ResetStatus

```
bool ResetStatus()
```

Dirigerer brugeren længere ind i systemet med **ResetStatus()**.

MotorStructure

Vi vil ikke gå i detaljen omkring opbygningen af strukturer og defines, da koden kun består af oprettelse og initialisering af variabler i en h-fil. Ønsker man at se koden, henviser vi til bilag 4.

Arg

Indeholder også en struktur.
Grunden til at den ikke kan ligge i ’MotorStructure’, er at ’Arg’ indeholder pointer til message-klassen, som inkluderer ’MotorStructure’.

MotorConfiguration

```
#include "MotorStructure.h"
#include "ErrorManager.h"
```

Inkludere to headerfiler for at tilgå strukturer, defines og funktioner.

```
class MotorConfiguration{
```

Der er ikke noget i oprettelse af klassen udover definitionen af funktionerne.

SetDefault

```
bool SetDefault(
         HCTRLR hCtlr,
         int stage,
         LPCTSTR &error
)
```

Denne funktion læser et sæt hardcodede værdier ned i motoren

hCtlr: Er handlet til at få fat i motorcontrolleren. Den skal bruges når der skal skrives/læses fra og til motorerne.
stage: Den motor der skal konfigureres.
error: Pointer til en tekststreng til at returnere fejl med.

```
MCMOTIONEX motion;

           long errorcode;
           bool boolerror;

           CONFIGDATA data;

           data.motion.Acceleration = 10000;
           data.motion.Deceleration = 0;
           data.motion.Velocity = 8000;
           data.motion.Torque = 0;
           data.motion.MinVelocity = 0;
           data.motion.Direction = 1;
           data.motion.StepSize = 1;
           data.motion.HardLimitMode = 3;
           data.motion.SoftLimitMode = 3;
           data.motion.SoftLimitHigh = 100000;
           data.motion.SoftLimitLow = -600000;
           data.motion.cbSize = sizeof( data.motion );

           data.filter.AccelGain = 0;
           data.filter.DecelGain = 0;
           data.filter.DerivativeGain = 2000;
           data.filter.DerSamplePeriod = 0.001364;
           data.filter.FollowingError = 1024.000000;
           data.filter.IntegralGain = 40;
           data.filter.IntegrationLimit = 50.000000;
           data.filter.VelocityGain = 0;

           data.scale.Constant = 0;
           data.scale.Offset = 0;
           data.scale.Rate = 1;
           data.scale.Scale = 1;
           data.scale.Time = 1;
           data.scale.Zero = 0;

           data.gain = 600;

           MCSetMotionConfigEx(hCtlr,stage, &data.motion);

           errorcode = MCSetFilterConfig(hCtlr,stage, &data.filter);
           em.Translate(errorcode,error);

           boolerror = (bool)MCSetScale(hCtlr,stage, &data.scale);

           errorcode = MCSetGain(hCtlr,stage,data.gain);
           em.Translate(errorcode,error);

           MCSetVelocity(hCtlr, stage, data.motion.Velocity );
```

```
            MCSetAcceleration(hCtlr,stage,data.motion.Acceleration);

            return 1;
}
```

Der oprettes variable til at modtage og fortolke fejl fra funktionerne der bruges til at skrive data til motorerne. Desuden oprettes der en forekomst af CONFIGDATA. Strukturen ligger ikke i denne klasse, men er en datastruktur der indeholder alle de data der er nødvendige for at konfigurere en motor.

```
struct CONFIGDATA
{
            MCMOTIONEX motion;
            MCFILTER filter;
            MCSCALE scale;
            double gain;
};
```

En dybdegående beskrivelse af indholdt i strukturen, kan det læses i MCAPI referencen.

De variable der er nødvendige for at køre stagen sættes. Funktioner til at skrive dem til motoren kaldes. Gain, acceleration og velocity indlæses i funktioner hver for sig, da vi har haft problemer med at få det til at virke eller.

Der er ikke taget højde for fejl, da errormanageren ikke er implementeret færdig. Men for hver funktion der bliver kaldt i funktionen, skal der være en reaktion hvis de ikke returner succes.

SetConfig

```
bool SetConfig(
            HCTRLR hCtlr,
            int stage,
            CONFIGDATA &data,
            LPCTSTR &error
)
```

Denne funktion tager en datastruktur og skriver den til motorerne.

hCtlr: Handlet til at få fat i motorcontrolleren. Det skal bruges når der skal skrives/læses fra og til motorerne.
stage: Den motor der skal konfigureres.
data: Indeholder de data der skal skrives til motorerne.
error: Pointer til en tekststreng til at returnere fejl med.

```
            int errorcode;
            ErrorManager em;
            if (errorcode = MCSetFilterConfig(hCtlr,stage, &data.filter) != 0)
            {
                  if (errorcode < 0)
                  {
                        em.Translate(errorcode,error);
                        return 0;
```

```
            }
        }
        if (errorcode = MCSetMotionConfigEx(hCtlr,stage, &data.motion)!= 0)
        {
            if (errorcode < 0)
            {
                em.Translate(errorcode,error);
                return 0;
            }
        }
        if (errorcode = MCSetScale(hCtlr,stage, &data.scale)!= 0)
        {
            if (errorcode < 0)
            {
                em.Translate(errorcode,error);
                return 0;
            }
        }
        if (errorcode = MCSetGain(hCtlr,stage,data.gain)!= 0)
        {
            if (errorcode < 0)
            {
                em.Translate(errorcode,error);
                return 0;
            }
        }

        return 1;
}
```

Vi opretter en integer til fejlkoder, og en forekomst af klassen errormanager. Herefter indlæses de ønskede datastrukturer til motoren. Hvis funktionerne returnerer en værdi der er forskellige fra nu betyder dette at den ikke er afviklet korrekt. Den værdi der returneres bliver oversat til en tekststreng der beskriver fejlen og funktionen forlades. Dette er ikke den færdige implementering af error, men det kunne godt være noget i denne retning. Ved at lade error tekststrengen vandre gennem systemet hvis der så sker en fejl, fungerer den som en form for global variabel der er initialiseret i interface klassen. Samtidigt skal systemet på en eller anden måde gøre brugeren opmærksom på der er sket en fejl.

Hvis alle ”Set” funktionerne er succesfulde returnerer funktionen 1.

GetConfig

```
bool GetConfig(
        HCTRLR hCtlr,
        int stage,
        CONFIGDATA &data,
        LPCTSTR &error
)
```

Denne funktion tager en pointer til en datastruktur som den lægger konfigurationen ind i.

hCtlr: Handlet til at få fat i motorcontrolleren. Skal bruges når der skal skrives/læses fra og til motoreren.

stage: Den motor der skal konfigureres.
data: Pointer til at indeholde data, der læses fra hardwaren.
error: Pointer til en tekststreng til at returnere fejlmeddelelser med.

```
int errorcode;
ErrorManager em;

if (errorcode = MCGetFilterConfig(hCtlr,stage, &data.filter)) {
     if (errorcode < 0)
     {
          em.Translate(errorcode,error);
          return 0;
     }
}
if (errorcode = MCGetMotionConfigEx(hCtlr,stage, &data.motion)){
     if (errorcode < 0)
     {
          em.Translate(errorcode,error);
          return 0;
     }
}
if (errorcode = MCGetScale(hCtlr,stage, &data.scale)){
     if (errorcode < 0)
     {
          em.Translate(errorcode,error);
          return 0;
     }
}
if (errorcode = MCGetGain(hCtlr,stage,&data.gain)){
     if (errorcode < 0)
     {
          em.Translate(errorcode,error);
          return 0;
     }
}
return 1;
```

Der oprettes en integer til at gemme fejlkoden i og en forekomt af errormanagerklassen. Derefter hentes opsætningen ud af motorerne og gemmes i datastrukturen. Sker der en fejl returneres 0 fra funktionen. Hvis der ikke sker nogen fejl retureres 1.

SetConfigGUI

```
bool SetConfigGUI(
          HCTRLR hCtlr,
          int stage,
          LPCTSTR &error
)
```

Denne funktion åbner et vindue til at ændre i værdierne direkte i motorerne.

hCtlr: Handlet til at få fat i motorcontrolleren. Skal bruges når der skal skrives/læses fra og til motorerne.

stage: Den motor der skal konfigureres.
error: Pointer til en tekststreng til at returnere fejl med.

```
int errorcode;
ErrorManager em;

LPCSTR title;
title = "configuration for stage " + stage;
errorcode = MCDLG_ConfigureAxis( NULL, hCtlr,
stage,MCDLG_CHECKACTIVE||MCDLG_PROMPT, title);
if (errorcode < 0)
{
      em.Translate(errorcode,error);
      return 0;
}
return 1;
```

Der oprettes en integer til at gemme fejlkoden i. Der laves en titel, og åbnes en GUI til konfigurering. Hvis fejlkoden er under 0, ligges en tekstbeskrivelse af fejlen i error, og der returners 0. Hvis det går godt returneres 1.

EnableStage

```
bool EnableStage(
          HCTRLR hCtlr,
          int stage,
          LPCTSTR &error
)
```

hCtlr: Handlet til at få fat i motorcontrolleren. Skal bruges når der skal skrives/læses fra og til motorerne.
stage: Den motor der skal konfigureres.
error: Pointer til en tekststreng til at returnere fejl med.

Denne funktion fungerer ved at den kalder MCEnablesAxis(), der ligger i producentens api. Men vores hedder stage i stedet for axis, det er for at prøve at reducere forvirringen ved at bruge samme term for forskellige ting. En akse er i vores forståelse flere stages lagt sammen, hvor det i deres er referer til som en stage.

```
MCEnableAxis(hCtlr,stage,TRUE);
return 1;
```

funktionen kaldes og der returneres.

ShowConfigdata

```
void ShowConfigdata(
          HCTRLR hCtlr,
          int stage
)
```

hCtlr: Handlet til at få fat i motorcontrolleren. Skal bruges når der skal skrives/læses fra og til motorerne.
stage: Den motor der skal konfigureres.

Denne funktion er kun beregnet til debugging. Den henter værdierne i motoreren og skriver dem med en printf funktion til skærmen.

Messages

Inkluderer kun MotorStructure.h, for at tilgå strukturer og defines.

```
#include "MotorStructure.h"
```

Klassen indeholder ligesom interface-klassen, __declspec(dllexport). Det har vi været nød til, da intarface-klassen bruger strukturer, som indeholder pointere til denne klasse.
Når klassen oprettes indeholder den tre pointere til movemessage. Der sættes til null i default konstruktøren.

```
class __declspec(dllexport) Messaging // message klasse, til push/pop beskeder i den dobbelhægtet liste
{
private:
		MOVEMESSAGE* last; // pointer til den sidste besked i listen
		MOVEMESSAGE* first; // pointer til den første besked
		MOVEMESSAGE* home; // pointer til hjem
public:

		Messaging() // default konstruktør
		{
			first = NULL; // alle 3 pointers sættes til  null
			last = NULL;
			home = NULL;
		}
```

Disse pointere bruges i funktionerne til at få adgang til de forskellige objekter i den hægtede liste.

```
struct MOVEMESSAGE
{
		MOVEMESSAGE* next;
		MOVEMESSAGE* prev;
		MESSAGE* pointerToMessage
};
```

Indeholder pointere til den foregående og den næste besked i den hægtede liste, og en pointer til datastrukturen den skal indeholde.

Additem

```
bool AddItem(
		MESSAGE* pointerToMessageClass
)
```

pointerToMessageClass: Pointer til messageklassen.

Funktionen returnere en boolean, true hvis den udføres korrekt og en false hvis den ikke gør. Som argument tager den en Message pointer, det er en pointer til den datastruktur objektet i listen skal indeholde. I funktionen oprettes der en ny movemessage.

```
		MOVEMESSAGE* newMoveMessage = new MOVEMESSAGE;
```

Den message som movemessage indeholder sættes til at være lig med den som ligger i argumentet.

```
newMoveMessage->pointerToMessage = pointerToMessage;
```

Her kan der opstå et problem i programmet hvis AddItem() bliver kaldt med samme pointer som argument flere gange. Så vil flere objekter indeholde en pointer til den samme struktur og hvis man skriver i den ene vil man skrive i dem alle samtidigt. Dette er på nuværende tidspunkt ikke et problem, da det ikke kan forekomme. Men man skal være opmærksom på det hvis der skal videreudvikles på api'et.

```
if (first == NULL)
        {
            last = newMoveMessage;
        }
```

Hvis first-pointeren er lig med null betyder det at listen er tom, og last pointeren skal pege på den nye besked.

```
newMoveMessage->next = first;
```

Next-pointeren skal altid pege på den besked som first peger på, for at den kan sættes ind imellem first og evt. objekter i listen. Hvis first er null kommer next til at pege på null. Dette er en indikator på at det er det sidste objekt i listen.

```
if(first != NULL)
        {
            newMoveMessage->next->prev = newMoveMessage;
        }
```

Hvis der ligger andre objekter i listen, skal det der kommer til at ligge efter det objekt vi sætter ind, pege tilbage på det.

```
newMoveMessage->prev = home;
```

Den nye beskeds prev–pointer, skal pege på det foregående objekt. I dette tilfælde hvor den bliver sat ind som det første, vil dette være 'home'. Hvis prev-pointeren peger på home, der har værdien null som sat i default konstruktøren, betyder det at den er den første besked i listen.

```
first = newMoveMessage;
```

First-pointeren sættes til at pege på den nye besked. Det afslutter indsættelsen af den nye besked i listen.
First peger på den nye besked, den nye besked peger på den besked first pegede på før. Den nye beskeds prev-pointer peger på 'home', og den besked der ligger efter den nye peger tilbage på den nye.

TakeOutItemBack

MOVEMESSAGE TakeOutItemBack()*
Når der skal fjernes en besked fra listen skal det i vores tilfælde gøres fra slutningen af listen. Listen fungerer som en FIFO liste. Listen er lavet så den uden videre problemer kan laves om til FILO eller løbes igennem for at finde præcis det objekt i beskederne man skal bruge. Derfor indeholder

den flere pointere end det egentligt er nødvendigt for vores brug, men dette ser vi ikke som et problem.

```
MOVEMESSAGE* Messaging::TakeOutItemBack()
{
            if (last != NULL)
            {
```

Funktionen returnere en pointer til typen movemessage. Man kunne nok nøjes med at returneres en pointer til en message, da det ikke er vigtigt for at modtage de to pointere til den foregående og den næste besked i listen.

Der undersøges om listen er tom

```
}
            else
            {
                  return NULL;
            }
```

Hvis den er tom returneres null. Men hvis den ikke er tom oprettes en ny pointer der sættes lig med last, der jo peger på den sidste besked i listen. Last sættes så til at pege på den foregående pointer, som jo bliver den sidste når denne besked er taget ud af listen.
Hvis den foregående besked er lig med null, er den besked vi har fat i den første og eneste i listen, da værdien null jo er den værdi der er i 'home'. Derfor skal first pege på null igen da der jo ikke er andre beskeder i listen at pege på. Ellers skal den forrige beskeds next pointer pege på det som den nuværende beskeds next pointer peger på. Så er beskeden fjernet fra listen og vi kan returnere pointeren til den og forlade funktionen.

```
MOVEMESSAGE* current = last;
last = current->prev;

if (current->prev == NULL )
{
    first = NULL;
}
else
{
    current->prev->next = current->next;
}
return current;
```

Display

void Display()
Udskriver pointernes værdier. Bruges til at debugging for at se om nye objekter bliver lagt på og taget ud af listen på korrekt vis, samt om listen holder sin integritet under disse operationer

DisplayBack

void DisplayBack()
Udskriver pointernes værdier bagfra. Bruges til at debugging for at se om nye objekter bliver lagt på og taget ud af listen på korrekt vis, og om listen holder sin integritet under disse operationer.

Show

```
void Show()
```

Viser indholdet af det første objekt på listen. Bruges til debugging når dataene skal følges gennem systemet.

MotorThreads

Dette er ikke en klasse, men en enkelt headerfil.
Inkluderer følgende filer, for at tilgå funktioner, strukturer og defines:

```
#include "MotorStructure.h"
#include "MotorMove.h"
#include <process.h>    /* _beginthread, _endthread */
#include "MotorStatus.h"
#include "Message.h"
#include "Arg.h"
```

T_ControlThread

```
unsigned _stdcall T_ControlThread(
         void* arg
)
```

arg: Pointer der peger på argumentet, hvori der gemmer sig en datastruktur.

Funktionen returnerer en unsigned _stdcall som er påkrævet for at den kan køre med '_beginthreadex'. Argumentet er en pointer til void. For at kunne have mere end et argument, har vi brugt en pointer til denne datastruktur.

```
struct MESSAGE_ARGUMENT
{
         Messaging* pointerToMessageClass;
         HANDLE threadSemaphore;
         HANDLE syncSemaphore;

};
```

Funktionen indeholder en pointer til klassen messaging, for at få adgang til den hægtede liste med beskeder. Den første semafor skal sørge for at tråden venter på at der er lagt noget i listen før den prøver at læse fra den. Den anden giver tråden videre til de tråde den laver, den sørger for at dataene i argument strukturerne forbliver det samme, indtil de er lagt over i lokale variable.

```
         MESSAGE_ARGUMENT argument;
         argument = *(MESSAGE_ARGUMENT*) arg;
```

Da vi ikke direkte kan tilgå dataene i en void pointer, typecaster vi den til den type den blev lagt ind som..

```
         MOVEMESSAGE* pm;

         HANDLE threadhandle[STAGE_ARRAY_MAX];
         unsigned threadID[STAGE_ARRAY_MAX];
         int NrOfThreads = 0;
```

```
THREAD_CALIBRATE_STAGE calistarg;
THREAD_CALIBRATE_STAGE* pcalistarg;
pcalistarg = &calistarg;
pcalistarg->syncSemaphore = argument.syncSemaphore;

THREAD_MOVE movearg;
THREAD_MOVE* pmovearg;
pmovearg = &movearg;
pmovearg->syncSemaphore = argument.syncSemaphore;
```

Der oprettes en movemessage-pointer til at pege på den movemessage der skal tages ud af listen. Denne metode virker ikke med at vi sletter beskeden når vi har returneret den, da denne pointer så ville pege på ingenting. Sammen med den omskrivning der er omtalt under messages implementeringen skal dette skrives om så den holdes lokalt her. Dette kan betyde at den skal returneres anderledes fra messaging klassen, og bruges på en anden måde. Men det forhindrer det memory leak vi har i øjeblikket hvor stacken bliver fyldt op med messages der ikke bliver slettet.

Der laves handles til oprettelse af tråde samt en integer. Grunden til at disse er arrays, er at vi overgang overvejede at lade tråden skifte mellem de handles den brugte til at oprette trådene. Dette er i midlertidig ikke nødvendigt, så det er bare et spørgsmål om at omskrive dette. Det betyder at vi ikke kan få fat i trådene fra denne funktion når først handlet er overskrevet med en ny værdi fra et nyt _BeginThreadEx kald, men det har vi heller ikke brug for.

De to datastrukturer er argumenter til tråde som bliver oprettet med pointere. Begge to bliver sat til at indeholde synkroniseringssemaforen. Det er nok ikke nødvendigt at have pointers til, da man kan bruge adressen til den og derved ”spare” en variabel.

```
do{
     WaitForSingleObject(argument.threadSemaphore,INFINITE);
     pm = argument.pointerToMessageClass->TakeOutItemBack();

     switch(pm->pointerToMessage->function)
```

Når variablene er oprettet kommer vi til den løkke som tråden skal køre i indtil programmet lukkes. Der ventes på semaforen. Den bliver frigivet i en anden tråd, når der bliver lagt noget ind i listen, og der bliver taget et objekt ud af listen. Der bliver så lavet en switch på den variable der beskriver hvilken funktion der skal afvikles.

```
case THREAD_FUNCTION_MOVETOX:
case THREAD_FUNCTION_MOVEARCTODEGREE:
case THREAD_FUNCTION_MOVEARCRELATIVEDEGREE:
case THREAD_FUNCTION_MOVERELATIVE:
     // Create arguments to send with the thread
     movearg.destination = pm->pointerToMessage->destination;
     movearg.function = pm->pointerToMessage->function;
     movearg.hCtlr = pm->pointerToMessage->hCtlr;
     movearg.stageOnAxis = pm->pointerToMessage->stageOnAxis;

     // Create a move thread
     threadhandle[NrOfThreads] = (HANDLE)_beginthreadex( NULL, 0,
          &T_MotorMove,(void *)(&movearg), 0,
```

```
                &threadID[NrOfThreads]);
          NrOfThreads++;
          break;
          // Create calibrations threads
     case THREAD_FUNCTION_CALIBRATESTAGE:
          // Create arguments to send with the thread
          pcalistarg->error = pm->pointerToMessage->error;
          pcalistarg->hCtlr = pm->pointerToMessage->hCtlr;
          pcalistarg->retur = pm->pointerToMessage->retur;
          pcalistarg->stage = pm->pointerToMessage->stage;

          // Create a calibration thread
                threadhandle[NrOfThreads] = (HANDLE)_beginthreadex( NULL,
                0, &T_MotorCalibration,(void *)  (pcalistarg), 0,
                &threadID[NrOfThreads] );
          NrOfThreads++;
          break;
          // Kill controle thread
     case THREAD_FUNCTION_KILLTHREAD:
          _endthreadex(0);
          break;
          default: return 1;
          }
```

Hvis det er en move funktion, tager den de variable move bruger og lægger dem over i move argumentet. Hvis det er en kalibreringsfunktion, lægges argumenterne over i den datastuktur og opretter den nye tråd til at køre funktionen i.

Hvis det er en besked om at lukke programmet, dræbes tråden. Alle tråde skabt ud af denne dræbes samtidigt.

```
                while(true);
                return 0;
```

Løkken skal køre indtil tråden dræbes, og der skal returneres en værdi fra funktionen. Ellers vil kompileren ikke køre.

T_MotorMove

```
unsigned _stdcall T_MotorMove(
          void* arg
)
```

arg: Pointer der peger på argumentet, hvori der gemmer sig en datastruktur.

For at kunne bruge den i den funktion der opretter tråden skal funktionen returnere en unsigned og tage argumentet på den måde. Argumentet er en datastruktur der indeholder flere variable til alle typer funktioner der skal afvikles i tråden.

```
MotorMove MM;

// Casting the argument to a THREAD_MOVE staucture
THREAD_MOVE stage = *(THREAD_MOVE*)arg;
ReleaseSemaphore(stage.syncSemaphore,1,NULL);
// Determine the function to reach
switch(stage.function)
```

Der oprettes en forekomst af motermove og gemmer argumentet lokalt. Derefter kommer der en switch af funktionsvariablen.

```
{
          case THREAD_FUNCTION_MOVETOX:
               MM.MoveToX(stage.hCtlr, stage.stageOnAxis, stage.destination);
               break;
          case THREAD_FUNCTION_MOVEARCTODEGREE:
               MM.MoveArcToDegree(stage.hCtlr, stage.stageOnAxis.stageId,
                    stage.destination);
               break;
          case THREAD_FUNCTION_MOVEARCRELATIVEDEGREE:
               MM.MoveArcRelativeDegree(stage.hCtlr, stage.stageOnAxis.stageId,
                    stage.destination);
               break;
          case THREAD_FUNCTION_MOVERELATIVE:
               MM.MoveRelative(stage.hCtlr, stage.stageOnAxis.stageId,
                    stage.destination, stage.delay);
               break;
          default: return 1;
}

          // End/close the thread
          _endthreadex(0);

          return 0;
```

Funktionen kaldes med de korrekte argumenter hvis funktionsvariablen passer. Når funktionen er færdig nedlægges tråden.

T_MotorCalibration

```
unsigned _stdcall T_MotorCalibration(
          void* arg
)
```

arg: Pointer der peger på argumentet, hvori der gemmer sig en datastruktur.

For at kunne bruge den i den funktionen der opretter tråden, skal funktionen returnere en unsigned og tage argumentet på følgende måde. Argumentet er en datastruktur der indeholder flere variable til alle typer funktioner der skal afvikles i tråden.

```
THREAD_CALIBRATE_STAGE stage;  // instance af datastrukturen til at
modtage argumenterne
stage = *(THREAD_CALIBRATE_STAGE*)arg; // argumenterne ligges over i
instancens, i værdier
ReleaseSemaphore(stage.syncSemaphore,1,NULL);
MotorMove MM;
MotorStatus ms;
STATUS_INFO Sinfo;

MM.MoveCalibration(stage.hCtlr,stage.stage,stage.retur,stage.error);
// kalder kaligreringalgoritmen

_endthreadex(0);
```

```
        return 0;
```

Argumenterne gemmes lokalt, og synkroniseringssemaforen tælles op, der oprettes en forekomst af motormove til at kalde funktionen med. Funktionen kaldes og tråden nedlægges, når funktionen er færdig.

MotorStatus

Inkluderer følgende for at kunne få fat i stagesne, strukturer og defines.

```
#include <windows.h>
#include "mcapi.h"
#include "mcdlg.h"
#include "MotorStructure.h"
```

StatusContainer

```
bool StatusContainer(
        bool get,
        int stageId,
        STATUS_INFO stageStatus[STAGE_ARRAY_MAX],
        int choice
)
```

get: Er 'true' hvis man ønsker at læse i status og 'false' hvis man ønsker at skrive i status.
stageId: Nummeret på den stage man ønsker at gøre status på.
stageStatus: Det array statusen skal lægges over i.
choice: De poster man ønsker at skrive ind i status.

StatusContainer'en skal kun bruges af de andre funktioner i klassen. Derfor har vi valgt at gøre den privat. Der er én lokal variabel i funktionen. Den er statisk og er et array af struktur til at indeholde status. Det er denne variabel, der sørger for at vores status bliver gemt.

```
        static STATUS_INFO stageStatusLocal[STAGE_ARRAY_MAX];
```

Derefter burger vi en switch til enten at modtage status eller sætte status. Hvis man skal modtage status, ligger vi 'stageStatusLocal' over i den medsendte array. Vi bruger en for-løkke til at lægge værdierne fra det ene array over i det andet array.

```
        switch(get)
        {
        case true:
            for(int k=0; k<STAGE_ARRAY_MAX; k++)
            {
                stageStatus[k] = stageStatusLocal[k];
            }
            break;
```

Skal man derimod sætte status, analyserer vi hvad det er man vil sætte. Det gør vi i en række if-sætninger, som ser på hvilke ønsker brugeren har sendt med over. Ønskerne ligger som "binære" tal, i variablen 'choice', og vil derfor være unikke.

```
        case false:
```

```
          if(choice/STATUS_CHOICE_CALIBRATED >= 1){
               stageStatusLocal[stageId].calibrated =
                    stageStatus[stageId].calibrated;
               choice -= STATUS_CHOICE_CALIBRATED;
          }
          if(choice/STATUS_CHOICE_CALIBRATION >= 1){
               stageStatusLocal[stageId].calibration =
                    stageStatus[stageId].calibration;
               choice -= STATUS_CHOICE_CALIBRATION;
          }
          if(choice/STATUS_CHOICE_ERROR >= 1){
               stageStatusLocal[stageId].error =
                    stageStatus[stageId].error;
               choice -= STATUS_CHOICE_ERROR;
          }
          if(choice/STATUS_CHOICE_MOVING >= 1){
               stageStatusLocal[stageId].moving =
                    stageStatus[stageId].moving;
               choice -= STATUS_CHOICE_MOVING;
          }
          if(choice/STATUS_CHOICE_POSITION >= 1){
               stageStatusLocal[stageId].position =
                    stageStatus[stageId].position;
               choice -= STATUS_CHOICE_POSITION;
          }
```

Hvis 'choice' ikke er nul til sidst, er der blevet sendt en forkert værdi med over, og der returneres 1 for fejl.

```
          if(choice!=0) return 1;
          break;
     }
     return 0;
```

GetStatus

```
bool GetStatus(
          HCTRLR hCtlr,
          int stageId,
          STATUS_INFO* status
)
```

hCtlr: Handle til at kommunikere med stagesne.
stageId: Nummeret på den stage man ønsker at gøre status på.
status: Pointer til en status-stauktur.

Vi starter med at oprette en lokal variable, som vi kan hente statusen over i. Derefter henter vi statusen fra StatusContainer().

```
     static STATUS_INFO stageStatus[STAGE_ARRAY_MAX];
     if(StatusContainer(true, NULL, stageStatus, NULL)){
          return 1;
     }
```

Derefter lægger statusen over i pointeren der er sendt med kaldet. Positionen modtager vi direkte fra stagen.

```
status->calibrated = stageStatus[stageId].calibrated;
status->calibration = stageStatus[stageId].calibration;
status->error = stageStatus[stageId].error;
status->moving = stageStatus[stageId].moving;
MCGetPositionEx(hCtlr, stageId, &status->position);
return 0;
```

GetStatusAxis

```
bool GetStatusAxis(
        HCTRLR hCtlr,
        char axis,
        STAGE stages[STAGE_ARRAY_MAX],
        STATUS_INFO* status
)
```

hCtlr: Handle til at kommunikere med stagesne.
axis: Den akse man ønsker at få status af.
stages: Array indholdene alle stages.
status: Pointer til en status-stauktur.

Vi starter med at oprette og initialisere tre lokale variable og hente status over i den ene af dem.

```
static STATUS_INFO stageStatus[STAGE_ARRAY_MAX];
int counter = 0;
double stagePosition=0;
StatusContainer(true, NULL, stageStatus, NULL);
```

Da vi har fået status for alle stages, finder vi nu dem der tilhører aksen. Det gør vi med en for-løkke, der kører alle stages igennem, og en if-sætning, der tjekker om stagen tilhører aksen.

```
for(int i=0; i<STAGE_ARRAY_MAX; i++)
{
     if(stages[i].axis == axis)
     {
```

Den første stages der bliver fundet, får lagt statusen over som i GetStatus(). Eneste ændring er at vi tæller en counter op efterfølgende.

```
if(counter==0)
{
     status->calibrated = stageStatus[i].calibrated;
     status->calibration = stageStatus[i].calibration;
     status->error = stageStatus[i].error;
     status->moving = stageStatus[i].moving;
     MCGetPositionEx(hCtlr, stages[i].stageId, &stagePosition);
     status->position = stagePosition;
     counter++;
}
```

De efterfølgende stages der bliver fundet, bliver sammenlignet med de forrige for at opnå det rigtige resultat for aksen. Det gør vi ved enten at and'e eller or'e dem.

```
else {
     status->calibrated = status->calibrated &
          stageStatus[i].calibrated;
     status->calibration = status->calibration |
          stageStatus[i].calibration;
     status->error = status->error | stageStatus[i].error;
     status->moving = status->moving | stageStatus[i].moving;
     MCGetPositionEx(hCtlr, stages[i].stageId, &stagePosition);
     status->position += stagePosition;
     counter++;
}
```

Til sidst returner vi 0.

```
return 0;
```

SetStatus

```
bool SetStatus(
          int stageId,
          STATUS_INFO status,
          int choice
)
```

stageId: Nummeret på den stage man ønsker at gøre status på.
status: En status-stauktur, som skal skrives ind i status.
choice: De poster man ønsker at skrive ind i status.

For at gemme statussen, opretter vi først en lokal variabel. Vi kopiere derefter statussen der er sendt med som argument over i variablen.

```
static STATUS_INFO stageStatus[STAGE_ARRAY_MAX];
stageStatus[stageId].calibrated = status.calibrated;
stageStatus[stageId].calibration = status.calibration;
stageStatus[stageId].error = status.error;
stageStatus[stageId].moving = status.moving;
stageStatus[stageId].position = status.position;
```

Til sidst sender vi den lokale variabel og variablen 'chice' ned i containeren.

```
StatusContainer(false, stageId, stageStatus, choice);
return 0;
```

ResetStatus

```
bool ResetStatus()
```

For at resætte status, sætter vi det hele til nul. Det gør vi ved oprette en lokal variabel, som vi fylder op med nuller. I en for-løkke sørger vi for at statussen for alle stages bliver resat.

```
static STATUS_INFO stageStatus[STAGE_ARRAY_MAX];
for(int i=0; i<STAGE_ARRAY_MAX; i++)
{
     stageStatus[i].calibrated = 0;
```

```
            stageStatus[i].calibration = 0;
            stageStatus[i].error = 0;
            stageStatus[i].moving = 0;
            stageStatus[i].position = 0;
            StatusContainer(false, i, stageStatus, STATUS_CHOICE_ALL);
        }
        return 0;
```

MotorMove

MotorMove får den enkelte stage til at bevæge sig.

Inkludere følgende for at kunne tilgå stagesne, strukturer, defines og status:

```
#include <windows.h>
#include <math.h> /* ceill(), floorl()*/
#include "mcapi.h"
#include "mcdlg.h"
#include "MotorStructure.h"
#include "MotorStatus.h"
```

MoveCalibration

```
bool MoveCalibration(
          HCTRLR hCtlr,
          int stage,
          bool retur,
          LPCTSTR &error
)
```

hCtlr: Handle til motorstyringen som algoritmen skal bruge til at få adgang til det digitale I/O og stagen.
stage: Er nummeret på den stage der skal kalibreres.
retur: Den variable der bestemmer om stagen skal retur til sin udgangsposition efter kalibreringen.

De variable der initialiseres til funktionen, har alle en god beskrivelse tilknyttet, så det vil vi ikke uddybe nærmere.

```
double repeatitions = 5; //antal gentagelse (max størrelsen på resultarray)
double resultarray[20]; //array der henvises til i kommentaren lige over
double oldacceleration; // bruges til at gemme accelerationen
double result = 0; //bruges til udregning af gennemsnit
double oldvelocity; // er den velocity der skal sættes tilbage til når
funktionen returnerer

double position; //position der læses fra motoren og lægges over i resultarray
double startPosition; // den position som motoren står i før kalibrering
foretages, bruges til at køre retur til den gamle position

double positionDifference; // bruges til udregning af forskellen på nulpunktet
og startPosition

double oldPosition = 0; //bruges til at forsikre om stagen bevæger sig
double newPosition = 1; //bruges til at forsikre om stagen bevæger sig
MCSTATUSEX stagestatus; // bruges til at dekode status for stages,
```

```
STATUS_INFO status; //status objekt til at get/set status for stagen
MotorStatus MS; // klasse objekt til manipuilation af ovenstående objekt
```

I det første tjek undersøger vi om stagen er optaget af at kalibrere eller at køre. Hvis den er optaget, sættes der en fejl og returneres. Hvis den ikke er optaget sættes den til at være optaget med at kalibrere.
Desværre er fejl håndtering/videresendelse ikke færdigt implementeret, så dette er ikke den færdige kode for fejl, men beskrivelsen vil være den korrekte.

```
MS.GetStatus(hCtlr,stage,&status);
if(status.moving == TRUE ||status.calibration == TRUE)
{
        printf("Stage %d is currently active from,
        kalibreringsalgoritme", stage);

        error = (LPCTSTR)"Stage is currently active";
        return 1;
}
else
{
        status.calibration = TRUE;
        MS.SetStatus(stage,status, STATUS_CHOICE_CALIBRATION);
}
```

Herefter aktiveres det digital I/O. Startpositionen, hastigheden og accelerationen hentes og gemmes. Hastigheden og accelerationen sættes til nye værdier, som vi gerne vil have dem til at være under kalibreringen. Disse værdier er valgt ud fra vores kendskab til systemet, som de hurtigste vi har kørt med uden at få problemer med nogle af stagesne.

```
MCEnableDigitalIO((HCTRLR)hCtlr,stage,TRUE); // aktiverer den
digitale port
MCGetPositionEx( hCtlr, stage , &startPosition ); // positionen
hentes

//denne if sætning undersøger staten på sensoren og rykker ud til et
punkt hvor sensoren burde være lav
//hvis dette ikke sker melder den en fejl og returerer
MCGetVelocityEx(hCtlr,stage,&oldvelocity);
MCGetAccelerationEx(hCtlr,stage, &oldacceleration);
MCSetVelocity(hCtlr,stage,8000); // stagen sættes til at køre med den
højeste hastighed vi atm er sikre på de kan klare
MCSetAcceleration(hCtlr,stage,10000);
```

En if-sætning kører hvis staten på sensoren er lav. Det kan der være flere grunde til som if-sætningen prøver at finde ud af. Først bevæges stagen tilbage for at vi kan være sikre på, at den er fri af sensoren, så undersøges der om stagen er slået til. Her kunne der argumenteres for, at det der skulle forsøges at slå den til og så køre med den, hvis den ikke var slået til. Det gør vi ikke i denne version, i stedet returnere vi.
Så undersøger vi igen om staten på sensoren er lav, hvis ja returneres der. Det kan være mange grunde til, for eksempel hvis opstillingen er tilsluttet forkert eller der er en korsslutning på sensoren. Igen er fejlreturneringen ikke lavet færdig. Når den returnerer skal status for sensoren sættes tilbage til ikke optaget, og brugeren skal informeres om fejlen.

```
if(MCGetDigitalIO(hCtlr,stage))
            {
                  MoveRelative(hCtlr, stage, -50000, true);

                  MCGetStatusEx( hCtlr, stage, &stagestatus );
                  if (!MCDecodeStatusEx( hCtlr, &stagestatus, MC_STAT_MTR_ENABLE
                        )){
                  error = (LPCTSTR) "Stage is not  enabled, tjeck configuration
                        and limit";
                  return 0;
            }

            if(MCGetDigitalIO(hCtlr,stage))
            {
                  error = (LPCTSTR) "The digital input do not change, make sure
                        the sensor and board is on and working";
                  return false;
            }
}
```

Herefter står en do-while-løkke for første gennemkørsel. Før vi kommer ind i den, sættes stagen til at køre 7.000.000 steps mod sensoren. Da stagen kun har ca. 600.000 steps mellem dens maksimum og minimum, er vi sikre på at den inden for denne kørsel, vil komme til sensoren, hvis der ikke opstår problemer undervejs.
Derfor undersøger vi løbende om stagen stadig bevæger sig, eller om den slår fra. Er den det, gøres funktionen ikke færdig, men melder en fejl.

Når sensoren skifter state stoppes stagen i nærheden af nulpunktet. Præcisionen ved den første måling er ca. 2500 steps ved den valgte hastighed og med det tidsinterval vi tester ved.
Tidsntervallet er de 50 millisekunder der er angivet i SleepEx().

```
MoveRelative(hCtlr, stage, 7000000, false);
            do
            {
                  SleepEx(50, FALSE);
                  MCGetPositionEx(hCtlr,stage,&newPosition);
                  if(newPosition == oldPosition)
                  {
                        error = (LPCTSTR) "Stage is not movin, tjeck
                              configuration";
                        return false;
                  }
                  else
                  {
                        newPosition= oldPosition;
                  }

                  MCGetStatusEx( hCtlr, stage, &stagestatus );
                  if (!MCDecodeStatusEx( hCtlr, &stagestatus, MC_STAT_MTR_ENABLE
                        )){
                        error = (LPCTSTR) "Stage is not  enabled, tjeck
                              configuration and limit";
                        return 0;
                  }
            }while(!MCGetDigitalIO(hCtlr,stage));
```

```
MCStop(hCtlr,stage);
```

Stagen rykkes tilbage så vi er fri af sensoren og hastighed sættes ned så vi kan få en bedre præcision ved samme tidsinterval.

```
MoveRelative(hCtlr, stage, -5000, true);


MCSetVelocity(hCtlr,stage,1000);
MoveRelative(hCtlr, stage, 10000, false);

do
{
      SleepEx(50, FALSE);
}while(!MCGetDigitalIO(hCtlr,stage));

MCStop(hCtlr,stage);
while(!MCIsStopped(hCtlr,stage,0)){SleepEx(50,FALSE);}
```

Så kører stagen anden runde af kalibreringen, efter samme princip som første vi kører bare kortere da vi nu har en god ide om hvor sensoren er.

```
for(int i = 0;i < repeatitions;i++){

        MCSetVelocity(hCtlr,stage,8000);
        MoveRelative(hCtlr, stage, -250, true);

        MCSetVelocity(hCtlr,stage,100);
        MoveRelative(hCtlr, stage, 10000, false);

        do
        {
              SleepEx(50, FALSE);
        }while(!MCGetDigitalIO(hCtlr,stage));

        MCStop(hCtlr,stage);
        while(!MCIsStopped(hCtlr,stage,0)){SleepEx(50,FALSE);}

        MCGetPositionEx( hCtlr, stage , &position
        resultarray[i] = position;
        result = result + position
}
```

Med denne for-løkke kommer vi frem til det endelige nulpunkt. Det gør vi ved at køre løkken gentagne gange og udregne et gennemsnit. Det punkt som udregningen kommer frem til er det absolutte nulpunkt.

Først rykkes stagen tilbage med høj hastighed, så langsomt frem. Det giver os en høj præcision, samtidigt med det ikke tager så lang tid. For at opnå en endnu højere præcision kan hastigheden sættes yderligere ned, hvilket gør at denne løkke tager længere tid at køre. Samtidigt kan dette dog betyde at antallet af gentagelser også kan sættes ned, hvilket sparer tid.
Tidsintervallet kan også sættes ned hvilket igen gør at vi kan opnå en større præcision, men belaster computeren mere. De to værdier er vi er kommet frem til for hastigheden og tidsintervallet, giver os

en kalibrering, som foregår på en tilfredsstillende tid og med en tilfredsstillende præcision. Mere om dette i test afsnittet.

Den tredje gennemkørsel foregår på samme måde som de to foregående på nær at vi tager en aflæsning på det nulpunkt vi kommer frem til og gemmer det.
Resultatet regnes ud og stagen rykkes til det punkt som algoritmen er kommet frem til er nulpunktet. Vi har ikke fået lavet nok test til at kunne vurdere om dette er den bedste måde at gøre det på. Man kunne måske helt droppe gentagelserne til sidst uden at miste præcision, eller måske lave en gentagelse til ved endnu lavere hastighed.

```
result = result/repeatitions; //gennemsnittet udregnes
MoveRelative(hCtlr, stage, result, true);
```

Positionen sættes til 0 i nulpunktet og hvis brugeren har bedt om at få rykket stagen tilbage til udgangspunktet, udregnes dette og stagen bliver kørt tilbage til dette.

```
MCSetPosition(hCtlr,stage,0);
if(retur == 1){
     positionDifference = startPosition - result;
     MoveRelative(hCtlr, stage, positionDifference, true);
}
```

Accelerationen og hastigheden sættes tilbage til den værdi de havde før kalibreringen. Det burde måske være en komplet konfiguration der blev gemt til at starte med og indlæst igen til slut, da der er flere ting vi ikke tager højde for. Blandt andet gain og grænser.

```
MCSetVelocity(hCtlr,stage,oldvelocity); //hastigheden sættes tilbage
til det den var da kalibreringen startede
MCSetAcceleration(hCtlr,stage,oldacceleration);
```

Status opdateres til at stagen ikke er optaget og den er kalibreret.

```
MS.GetStatus(hCtlr,stage,&status);
status.calibration = FALSE;
status.calibrated = TRUE;
MS.SetStatus(stage,status,(STATUS_CHOICE_CALIBRATED
+STATUS_CHOICE_CALIBRATION));
```

MoveToX

```
bool MoveToX(
          HCTRLR hCtlr,
          STAGE stage,
          double destination
)
```

hCtlr: Handle til at kommunikere med stagesne.
stage: Struktur med stage-informationer.
destination: Det ønskede slutposition.

Vi starter med at oprette tre variable, position, x_max og x_min. Positionen bliver sat til nul, for at være sikker på at den har en værdi, når vi senere skal bruge den til at få stagens nuværende position. For at sætte x_max og x_min, gør vi som vist herunder:

```
double position=0;
double x_max =
     (stage.max*stage.direction>=stage.min*stage.direction)?
     stage.max*stage.direction : stage.min*stage.direction;
double x_min =
     (stage.max*stage.direction<stage.min*stage.direction)?
     stage.max*stage.direction : stage.min*stage.direction;
```

Grunden til at vi gør det på denne måde, er at x_max og x_min skal bruges til at bestemme grænseværdierne, og vi bliver derfor nødt til at tage højde for om en stage vender modsat de andre stages i en akse. Altså om stagen har en negativ kørselsretning. Ved at gange kørselsretningen på stagens maksimum og minimum værdi, kan man bestemme om man skal bytte rundt på maksimum og minimum. Lad os i et eksempel antage at maksimum er 10, minimum er -2 og orienteringen er negativ. I det tilfælde bliver maksimum 2 og minimum -10.
Vi tjekker herefter i en if-sætning om den ønskede destination ligger inden for grænserne. Gør den ikke det, sætter vi posten error i status til 'true' og returnere 1.

```
if(destination > x_max || destination < x_min){
     status.error = 1;
     MS.SetStatus(stage.stageId, status, STATUS_CHOICE_MOVING);
     return 1;
}
```

Ellers henter vi positionen fra stagen og bruger den til at finde den distance der skal rykkes. Lykkes det ikke, registrerer vi at der er en fejl, og returnere en fejl. Det der ellers bliver returneret, er desuden om MoveRelativeAjusted() lykkes. MoveRelativeAjusted() er funktionen der endeligt rykker stagesne.

```
if(MCGetPositionEx(hCtlr, stage.stageId, &position){
     status.error = 1;
     MS.SetStatus(stageId, status, STATUS_CHOICE_ERROR);
     return 1;
}
return MoveRelativeAdjusted(hCtlr, stage.stageId, destination-
     position);
```

MoveArcToDegree

```
bool MoveArcToDegree(
          HCTRLR hCtlr,
          WORD stageId,
          double destination
)
```

hCtlr: Handle til at kommunikere med stagesne.
stageId: Stagenummeret man ønsker at rykke.
destination: Den ønskede slutposition.

Her starter vi med at oprette variable og tjekke om den ønskede destination, som er i grader, ligger inden for grænserne. Gør den ikke det, sætter vi posten error i status til 'true' og returnere 1.

```
double distance;
double position=0;
```

```
if(destination<0 || destination>360){
     status.error = 1;
     MS.SetStatus(stageId, status, STATUS_CHOICE_ERROR);
     return 1;
}
```

Derefter henter vi positionen fra stagen. Lykkes det ikke, registrerer vi at der er en fejl, og returnere en fejl.

```
if(MCGetPositionEx(hCtlr, stageId, &position)){
     status.error = 1;
     MS.SetStatus(stageId, status, STATUS_CHOICE_ERROR);
     return 1;
}
```

Distancen bliver nu regnet ud, ved at gange en konverteringsværdi på den ønskede destination. Vi får så destinationen i steps og så trække den nuværende position fra. For ikke at sende et komma tal med til stagen, runder vi den endelige distance af til et heltal. Til sidst bruger vi MoveRelativeAjusted() til at rykke stagen og returnere 0.

```
distance = destination * STEP_CONVERSION_DEGREETOSTEP - position;
distance = (ceill(distance)-distance <= distance-floorl(distance)) ?
     ceill(distance) : floorl(distance);
MoveRelativeAdjusted(hCtlr, stageId, distance);
return 0;
```

MoveArcRelativeDegree

```
bool MoveArcRelativeDegree(
          HCTRLR hCtlr,
          WORD stageId,
          double degree
)
```

hCtlr: Handle til at kommunikere med stagesne.
stageId: Stagenummeret man ønsker at rykke.
degree: Det ønskede antal grader man vil rykke.

Denne funktion, som rykker en turnstage relativt, ligner MoveArcToDegree() så meget, at vi vil nøjes med at beskrive hvordan man finder grænserne. Til forskel fra MoveArcToDegree(), er vi nød til at finde grænserne ved at bruge vores nuværende position og gange en konverteringsværdi på, for til sidst at ligge den ønskede distance til.

```
if(position * STEP_CONVERSION_STEPTODEGREE + degree > 360 || position
     * STEP_CONVERSION_STEPTODEGREE + degree<0){
     status.error = 1;
     MS.SetStatus(stageId, status, STATUS_CHOICE_ERROR);
     return 1;
}
```

MoveRelative

```
bool MoveRelative(
          HCTRLR hCtlr,
```

```
            WORD stageId,
            double distance,
            bool delay
)
```

hCtlr: Handle til at kommunikere med stagesne.
stageId: Stagenummeret man ønsker at rykke.
distance: Den distance der ønskes rykket.
delay: 'true' hvis man ønsker at vente til stagen har kørt færdig, 'false' hvis man ønsker at returnere førend stagen har kørt færdig.

Efter at havde oprettet tre variable, tjekker vi om stagen er reserveret. Det gør vi ved at kalde GetStatus() fra status-klassen. Er stagen i brug, returnerer vi 1 og går ud af funktionen. Ellers reservere vi stagen ved at sætte posten moving i status til 'true'.

```
MotorStatus MS;
STATUS_INFO status;
MCSTATUSEX stageStatus;
MS.GetStatus(hCtlr, stageId, &status);
if(status.moving == true)
{
     return 1;
}
status.moving = true;
MS.SetStatus(stageId, status, STATUS_CHOICE_MOVING);
```

Derefter lader vi stagen køre ved at kalde MCMoveRelative() fra det medfølgende api.

```
MCMoveRelative((HCTRLR)hCtlr, stageId, distance);
```

Da funktionen er optimeret til kalibrering, kan man selv vælge om man vil vente på at stagen har kørt færdig, eller om man vil fortsætte i koden. Hvis 'delay' er 'true', startes en while-løkke, som tjekker om stagen er stoppet. Er den ikke det sleepes et sekund og der tjekkes igen. Samtidig tjekkes der også for om stagen er stået af, eller stadig er tilgængelig.

```
if(delay==true){
     while(!MCIsStopped(hCtlr, stageId, 0.01)){
          Sleep(1*1000);
          MCGetStatusEx( hCtlr, stageId, &stageStatus );
          if (!MCDecodeStatusEx( hCtlr, &stageStatus,
              MC_STAT_MTR_ENABLE )){
              return 1;
          }
     }
}
```

Til sidst frigives stagen ved at sætte posten 'moving' i status til 'false'.

```
status.moving = false;
MS.SetStatus(stageId, status, STATUS_CHOICE_MOVING);
return 0;
```

MoveRelativeAjusted

```
bool MoveRelativeAdjusted(
         HCTRLR hCtlr,
         WORD stageId,
         double distance
)
```

hCtlr: Handle til at kommunikere med stagesne.
stageId: Stagenummeret man ønsker at rykke.
distance: Den distance der ønskes rykket.

Vi starter med at initialisere nogle variable, for derefter at tjekke om stagen er reserveret, om den er kalibreret, eller om den er i gang med at blive kalibreret. Er den ikke det, reservere vi den, så andre ikke kan rykke stagen, mens vi gør det.

```
double positionBefore=0, positionAfter=0;
MCSTATUSEX stageStatus;
MS.GetStatus(hCtlr, stageId, &status);
if(status.moving == true  || status.calibration == true ||
     status.calibrated == false)
{
     return 1;
}
status.moving=true;
MS.SetStatus(stageId, status, STATUS_CHOICE_MOVING);
```

Derefter looper vi i en do-while-løkke, indtil vi er inden for ét step af destinationen. I løkken henter vi stage-positionen over i variablen positionBefore. Samtidig tjekker vi for om der skete en fejl.

```
do{
  if(MCGetPositionEx(hCtlr, stageId, &positionBefore)){
       status.error = 1;
       MS.SetStatus(stageId, status, STATUS_CHOICE_MOVING);
       return 1;
  }
```

Derefter rykker vi stagen, indtil den står stille. Undervejs tjekker vi for, om vi stadigvæk har forbindelse til stagen. Mister vi forbindelsen, returnere i 1 for fejl.

```
MCMoveRelative((HCTRLR)hCtlr, stageId, distance);
while(!MCIsStopped(hCtlr, stageId, 0.01)){
     Sleep(1*1000);
     MCGetStatusEx( hCtlr, stageId, &stageStatus );
     if(!MCDecodeStatusEx( hCtlr, &stageStatus, MC_STAT_MTR_ENABLE)){
         return 1;
     }
}
```

Efter at havde kørt stagen, henter vi nu positionen for stagen ind igen, og lægger værdien i variablen 'positionAfter'.

```
if(MCGetPositionEx(hCtlr, stageId, &positionAfter)){
     status.error = 1;
```

```
            MS.SetStatus(stageId, status, STATUS_CHOICE_MOVING);
            return 1;
        }
```

Derefter regner en ny distance ud. Ved at tage positionen før vi begyndte at rykke stagen og lægge den sammen med den distance vi rykkede stagen, for derefter at trække positionen efter aktionen fra, får vi den distance der mangler eller som der er rykket for meget. Optimalt vil denne nye distance være nul. Hvis den nye distance er større end ±1, køres do-while-løkken igen, med den nye distance.

```
            distance = positionBefore + distance - positionAfter;
        }while(distance <= -1 || distance >= 1);
```

Til sidst frigives stagen og der returneres 0.

```
        status.moving = false;
        MS.SetStatus(stageId, status, STATUS_CHOICE_MOVING);
        return 0;
```

MotorCalibration

Definitionen af klassen er der ikke rigtigt noget ved, der er ikke nogen private variable eller funktioner, kun de tre public funktioner der bliver beskrevet.

Inkludere følgende, for at kunne tilgå strukturer, defines, messages og status.

```
#include "MotorStructure.h"
#include "message.h"
#include "MotorStatus.h"
```

CalibrateStage

```
bool CalibrateStage(
         HCTRLR hCtlr,
         int stage,
         bool retur,
         LPCTSTR &error,
         HANDLE threadSemaphore,
         HANDLE syncSemaphore,
         Messaging* pointerToMessageClass
)
```

hCtlr: Er et handle til motorstyringen, som algoritmen skal bruge til at få adgang til det digitale I/O og stagen.
stage: Er nummeret på den stage, der skal kalibreres.
retur: Bestemmer om stagen skal køre retur til den position den startede i. Den skal være 'true' hvis den skal køre tilbage og 'false' hvis den skal blive i nulpunktet.
error: Er en pointer til en tekst streng til at indeholde fejlbeskrivelsen.
threadSemaphore: Semafor som når den bliver frigivet, sætter kontroltråden i gang med at hente en besked fra message klassen.
pointerToMessageClass: En pointer til message klassen, som skal bruges til at lægge en besked ind i listen.

syncSemaphore: Løser et problem vi havde med at dataene i argumenterne i kontroltråden, blev korrupte hvis vi læste en ny besked før den forrige var lagt korrekt ind i en lokal variabel i den næste tråd. Skader ikke funktionaliteten af programmet, men er ikke en ”pæn” løsning.

Der oprettes en ny besked og en pointer til den, pointeren sættes til at pege på beskeden.

```
MESSAGE besked;
MESSAGE* pbesked;
pbesked = &besked;
```

Herefter lægges variablerne over i en besked. `THREAD_FUNCTION_CALIBRATESTAGE` er en defineret integer, der fortæller kontroltråden hvilken funktion der skal afvikles.

```
pbesked->function = THREAD_FUNCTION_CALIBRATESTAGE;
pbesked->stage = stage;
pbesked->hCtlr = hCtlr;
pbesked->retur = retur;
pbesked->error = error;
```

Der laves en wait på synkroniseringssemaforen, som tillader én adgang til det område den omslutter ad gangen. Det vil sige at hvis denne funktion bliver kaldt med det samme igen, vil den stoppe her indtil semaforen er blevet frigivet, hvilket den gør andetsteds.
AddItem() lægger beskeden over i den liste som message-klassen administrerer, derefter frigives ’threadSemaphore’, som kontroltråden venter på. ’threadSemaphore’ giver kontroltråden lov til at gå ind og hente beskeden fra message-klassen.

```
WaitForSingleObject(syncSemaphore,INFINITE);
pointerToMessageClass->AddItem(&besked);

ReleaseSemaphore(threadSemaphore,1,NULL);

return 1;
```

CalibrateAxis

```
bool CalibrateAxis(
        HCTRLR hCtlr,
        char axis,
        bool retur,
        STAGE stages[STAGE_ARRAY_MAX],
        LPCTSTR &error,
        HANDLE threadSemaphore,
        HANDLE syncSemaphore,
        Messaging* pointerToMessageClass
)
```

hCtlr: Er et handle til motorstyringen, som algoritmen skal bruge til at få adgang til det digitale I/O og stagen.
axis: Er en char der bestemmer den akse som skal kalibreres. Den skal sammenholdes med de stages der ligger i et argumentet ’stages’. Hver enkelt stage har en char der fortæller hvilken akse de tilhører.

retur: Bestemmer om stagen skal køre retur til den position den startede i. Den skal være 'true' hvis den skal køre tilbage og 'false' hvis den skal blive i nulpunktet.
stages[STAGE_ARRAY_MAX]: Et array af datastrukturen 'STAGE' der indeholde data om de stages der er tilsluttet kontrolleren og oprettet i systemet.
error: Er en pointer til en tekst streng, som indeholder fejlbeskrivelsen.
threadSemaphore: Semafor som når den bliver frigivet, sætter kontroltråden i gang med at hente en besked fra message klassen.
pointerToMessageClass: En pointer til message klassen, som skal bruges til at lægge en besked ind i listen.
syncSemaphore: Synkroniserings semafor, forsikre at dataene ikke når at blive overskrevet inden funktionen der skal bruge dem har fået gemt dem lokalt

Vi opretter en besked med tilhørende pointer og to andre variable til funktionen. Den første tæller vi op for hver gang der er en stage på aksen der skal kalibreres og den anden tjekker stagetypen. Denne funktion skal kun bruges til slidestages, da turnstages ikke kan kalibreres. Vi har desuden valgt ikke at have akser af turnstages.

```
MESSAGE besked;
MESSAGE* pbesked;
pbesked = &besked;

int NumberOfStagesonAxis = 0;
int stageType = STAGE_TYPE_SLIDE;
```

En for-løkke løber hele arrayet med stages igennem, for at finde dem der ligger på den akse der ønskes kalibreret, Lægger derefter på samme måde som i CalibrateStage() variabler over i en besked og sende dem til message klassen.

```
for(int i=0; i<STAGE_ARRAY_MAX; i++)
        {
            if( stages[i].axis == axis && stageType != stages[i].type)
            {
                error = (LPCTSTR)"Only one type of stage allowed on axis";
                break;
            }
            if(stages[i].axis == axis)
            {
                pbesked->function = THREAD_FUNCTION_CALIBRATESTAGE;
                pbesked->hCtlr = hCtlr;
                pbesked->retur = retur;
                pbesked->stage = stages[i].stageId;
                pbesked->error = error;
                WaitForSingleObject(syncSemaphore,INFINITE);
                pointerToMessageClass->AddItem(pbesked);

                ReleaseSemaphore(threadSemaphore,1,NULL);
            }
        }
```

SetCalibration

```
bool SetCalibration(
        HCTRLR hCtlr,
```

```
        int stage,
        double position
)
```

hCtlr: Er et handle til motorstyringen som algoritmen skal bruge til at få adgang til det digitale I/O og stagen.
stage: Er nummeret på den stage der skal sættes som kalibreret.
position: Er den position stagen skal sættes til at have, for at være kalibreret. For eksempel en dreje stage der står på 20 grader skal sættes til at have positionen 20*STEP_CONVERSION_DEGREETOSTEP[13] = 32.876.

Vi opretter en forekomst af klassen 'MotorStatus' og strukturen 'STATUS_INFO', positionen for stagen sættes til at være den ønskede. Status for stagen hentes i klassen. Den sættes til at ikke være i gang med at kalibrere og være kalibreret, statusen lægges tilbage i klassen.

```
MotorStatus MS;
STATUS_INFO status;

MCSetPosition(hCtlr,stage,position);

MS.GetStatus(hCtlr,stage,&status);
status.calibration = FALSE;
status.calibrated = TRUE;
MS.SetStatus(stage,status,(STATUS_CHOICE_CALIBRATED
+STATUS_CHOICE_CALIBRATION));
```

AxisMove

AxisMove får akserne til at bevæge sig ved at finde alle de stage der sidder på aksen og derefter finde ud af hvor langt den enkelte stage skal rykke sig, førend aksen når den ønskede position.

Inkludere følgende, for at kunne tilgå stages, strukturer, defines, messages og status.

```
#include "MotorStructure.h"
#include <windows.h>
#include "mcapi.h"
#include "mcdlg.h"
#include "Message.h"
#include "MotorStatus.h"
```

CalculateNewPosition

```
bool CalculateNewPosition(
        HCTRLR hCtlr,
        double position,
        int stageCounter,
        STAGE stagesOnAxis[STAGE_ARRAY_MAX],
        double* stagePosition[STAGE_ARRAY_MAX]
)
```

hCtlr: Handle til at tilgå stagesne.
position: Positionen aksen skal bevæge sig til.

[13] Defineret til at være: 3600/2,19

stageCounter: Antallet af stages på aksen.
stagesOnAxis: Array indholdene alle stagesne på aksen.
stagePosition: Kommer til at indeholde stagesnes nye positioner.

Dette er funktionen, som udregner de nye positioner for stagesne på en akse. Da funktionen kun skal bruges af MoveAxis(), har vi valgt at gøre den privat.
Efter at havde oprettet og initialiseret variablerne der skal bruges, finder vi aksens grænser og hvor langt aksen skal bevæge sig for at opnå den ønskede position.

```
int x_left = stageCounter;
double rest = position;
double restTemp = 0;
double x_max = 0;
double x_min = 0;
double
stagePositionGet[STAGE_ARRAY_MAX],stagePositionLocalTemp[STAGE_ARRAY_
MAX];
for(int i=0; i<stageCounter; i++)
{
     x_max += (stagesOnAxis[i].max*stagesOnAxis[i].direction >=
          stagesOnAxis[i].min*stagesOnAxis[i].direction)?
          stagesOnAxis[i].max*stagesOnAxis[i].direction :
          stagesOnAxis[i].min*stagesOnAxis[i].direction;
     x_min += (stagesOnAxis[i].max*stagesOnAxis[i].direction <
          stagesOnAxis[i].min*stagesOnAxis[i].direction)?
          stagesOnAxis[i].max*stagesOnAxis[i].direction :
          stagesOnAxis[i].min*stagesOnAxis[i].direction;
     if(MCGetPositionEx(hCtlr, stagesOnAxis[i].stageId,
          &stagePositionGet[i])) return 1;
     stagePosition[i] = &stagePositionGet[i];
     rest = rest - *stagePosition[i];
}
```

Variablerne 'x_max' og 'x_min', bruges til tjekke at den ønskede position ligger inden for aksens grænser. Variablen 'rest' er det der mangler at blive rykket, førend aksen står på den ønskede position.
Er vi ikke inden for grænserne, returnere vi en fejl.

```
if(position>x_max || position<x_min)
{
     return 1;
}
```

Nu er vi klar til udregne hver stages nye position. Dette gør vi i en do-while-løkke, der kører indtil at hele den manglende distance er fordelt på stagene, eller indtil at der ikke er flere stages tilbage at fordele på.
Da proceduren er beskrevet på figur 11, vi har givet en matematisk gennemgang i overvejelserne, samt kommenteret kildekoden, vil vi henvise hertil og nøjes med at vise selve råkoden her:

```
do{
     double dx = rest/x_left;
     rest = 0;
```

```
            for(int j=0; j<stageCounter; j++)
            {
                x_max = (stagesOnAxis[j].max * stagesOnAxis[j].direction >=
                    stagesOnAxis[j].min * stagesOnAxis[j].direction)?
                    stagesOnAxis[j].max * stagesOnAxis[j].direction :
                    stagesOnAxis[j].min * stagesOnAxis[j].direction;
                x_min = (stagesOnAxis[j].max * stagesOnAxis[j].direction <
                    stagesOnAxis[j].min * stagesOnAxis[j].direction)?
                    stagesOnAxis[j].max * stagesOnAxis[j].direction :
                    stagesOnAxis[j].min * stagesOnAxis[j].direction;
                stagePositionLocalTemp[j] = *stagePosition[j];
                if(dx>0 && stagePositionLocalTemp[j]+dx>=x_max &&
                    x_left!=false){
                    if(*stagePosition[j]!=x_max){
                        rest += (*stagePosition[j] + dx - x_max);
                        x_left--;
                        *stagePosition[j] = x_max;
                        restTemp=0;
                    }else restTemp+=dx;
                }else if(dx<0 && stagePositionLocalTemp[j]+dx<=x_min &&
                    x_left!=false){
                    if(*stagePosition[j]!=x_min){
                        rest += (*stagePosition[j] + dx - x_min);
                        x_left--;
                        *stagePosition[j] = x_min;
                        restTemp=0;
                    }else restTemp+=dx;
                }else if(x_left!=0){
                    *stagePosition[j] = dx + stagePositionLocalTemp[j];
                }
            }
            if(restTemp!=0){
                x_left--;
                rest=restTemp;
            }
        }while(rest!=0 && x_left!=false);

        return 0;
```

Alle stagesnes nye positioner ligger nu i pointerarrayet 'stagePosition'.

MoveAxis

```
bool MoveAxis(
        HCTRLR hCtlr,
        char axis,
        double position,
        STAGE stages[STAGE_ARRAY_MAX]
)
```

hCtlr: Handle til at tilgå stagesne.
axis: Den akse man vil rykke.
position: Positionen aksen skal bevæge sig til.
stagesOnAxis: Array indholdene alle stagesne.

Vi starter med at oprette og initialisere variable. Her bruger vi en for-løkke til at tildele pointerarrayet adresser.

```
MotorStatus MS;
STATUS_INFO status;
STAGE stagesOnAxis[STAGE_ARRAY_MAX];
int stageCounter = 0;
double* stagePosition[STAGE_ARRAY_MAX]={0,0,0,0,0,0,0,0};
int stageType = STAGE_TYPE_NONE;
MESSAGE beMessage[STAGE_ARRAY_MAX];
MESSAGE* pointerToMessage[STAGE_ARRAY_MAX];

for(int h=0; h<STAGE_ARRAY_MAX; h++)
     pointerToMessage[h] = &beMessage[h];
```

Vi tjekker herefter om aksen er i brug, og går kun videre hvis alle stages på aksen er frie.

```
MS.GetStatusAxis(hCtlr, axis, stages, &status);
if(status.moving==1) return 1;
```

Nu finder vi alle stages, som der tilhører aksen. Det gør vi i en for-løkke, som kører lige så mange gange der er potentiale for stages. Stagene består jo af en struktur, hvori der ligger information om hvilken akse den tilhører, og ligger i et array.
Vi tjekker dog først om stagetypen varierer. Gør den det returner vi 1 for fejl. Ellers tjekker vi om stagen tilhører aksen. Gør den det, ligger vi stagen over i et lokalt array, som kun består at stages der tilhøre aksen og tæller en counter op. Counteren bruger vi til at vide, hvor mange stages der ligger i det lokale array.

```
for(int i=0; i<STAGE_ARRAY_MAX; i++)
{
     if(stageCounter>0 && stages[i].axis == axis && stageType !=
          stages[i].type)
     {
          return 1;
     }
     if(stages[i].axis == axis)
     {
          stagesOnAxis[stageCounter] = stages[i];
          stageCounter++;
          stageType = stages[i].type;
     }
}
```

Hvis der er stages der tilhører aksen, bruger vi en switch til bestemme om det er en akse af slidestages eller at turnstages. Vi udfører dog intet, hvis det er en akse bestående af turnstages. Det gør vi ikke, da man fra Oticons side ikke mente at der ville være behov for det. Derfor står den åben for eventuelle fremtidig implementering. Består aksen derimod af slidestages, starter vi med at få udregnet stagesnes fremtidige positioner med CalculateNewPositions().

```
if(stageCounter!=0)
{
     switch(stageType)
     {
     case STAGE_TYPE_TURN:
          break;
```

```
                case STAGE_TYPE_SLIDE:
                     CalculateNewPosition(hCtlr, position, stageCounter,
stagesOnAxis, stagePosition);
```

Herefter ligger vi alle stagesne ind i et argument. Argumentet bliver derefter lagt over i et messagepointerarray. Vi synkronisere, ved at vente på synkroniserings semaphoren, og lægger så messagepointerarrayet op som en message. Til sidst fortæller vi systemet at der er lagt en ny message i messagekøen, ved at tælle tråd semaphoren op.

```
                     THREAD_MOVE ar[STAGE_ARRAY_MAX];
                     THREAD_MOVE *arr[STAGE_ARRAY_MAX];
                     for(int j=0; j<stageCounter; j++)
                     {
                         ar[j].hCtlr = hCtlr;
                         ar[j].stageOnAxis = stagesOnAxis[j];
                         ar[j].destination = *stagePosition[j];
                         ar[j].function = THREAD_FUNCTION_MOVETOX;
                         arr[j] = &ar[j];
                     }
                     for(int k=0; k<stageCounter; k++)
                     {
                         pointerToMessage[k]->hCtlr = arr[k]->hCtlr;
                         pointerToMessage[k]->retur = NULL;
                         pointerToMessage[k]->stage = arr[k]->
                             stageOnAxis.stageId;
                         pointerToMessage[k]->stageOnAxis =arr[k]->stageOnAxis;
                         pointerToMessage[k]->destination = arr[k]->destination;
                         pointerToMessage[k]->function =
                             THREAD_FUNCTION_MOVETOX;
                         WaitForSingleObject(syncSemaphore,INFINITE);
                         pointerToMessageClass->AddItem(pointerToMessage[k]);
                         ReleaseSemaphore(threadSemaphore,1,NULL);
                     }
                     break;
                default:
                     break;
                }
           }
           return 0;
```

MoveAxisXYZA

```
bool MoveAxisXYZA(
           HCTRLR hCtlr,
           char XAxis,
           double XPosition,
           char YAxis,
           double YPosition,
           char ZAxis,
           double ZPosition,
           char AAxis,
           double APosition,
           STAGE stages[STAGE_ARRAY_MAX]
)
```

hCtlr: Handle til at tilgå stagesne.
XAxis: Den akse man vil rykke.
XPosition: Positionen aksen skal bevæge sig til.
YAxis: Den akse man vil rykke.
YPosition: Positionen aksen skal bevæge sig til.
ZAxis: Den akse man vil rykke.
ZPosition: Positionen aksen skal bevæge sig til.
AAxis: Den akse man vil rykke.
APosition: Positionen aksen skal bevæge sig til.
stagesOnAxis: Array indholdene alle stagesne.

MoveAxisXYZA() gør egentlig bare det, at den kalder MoveAxis() tre gange og opretter en message om at bevæge en turnstage.
Vi kalder altså MoveAxis() tre gange, efter at havde oprettet diverse variable. Hvis MoveAxis() lykkes går vi videre i koden, ellers returnere 1 for fejl.

```
THREAD_MOVE ar;
THREAD_MOVE *arr;
MESSAGE beMessage;
MESSAGE* pointerToMessage;
pointerToMessage = &beMessage;

if(MoveAxis(hCtlr,XAxis,XPosition,stages,threadSemaphore,
     syncSemaphore, pointerToMessageClass)) return 1;
if(MoveAxis(hCtlr,YAxis,YPosition,stages,threadSemaphore,
     syncSemaphore, pointerToMessageClass)) return 1;
if(MoveAxis(hCtlr,ZAxis,ZPosition,stages,threadSemaphore,
     syncSemaphore, pointerToMessageClass)) return 1;
```

Nu mangler vi bare at fortælle, at turnstagen skal bevæge sig. Som de andre gange, opretter vi først et argument, for derefter at ligge det over i en messagepointer. Vi bruger en for-løkke og en if-sætning til at finde den rigtige stage.

```
ar.hCtlr = hCtlr;
for(int i=0;i<STAGE_ARRAY_MAX;i++)
     if(AAxis == stages[i].axis)
          ar.stageOnAxis = stages[i];
ar.destination = APosition;
ar.function = THREAD_FUNCTION_MOVEARCTODEGREE;
arr = &ar;

pointerToMessage->function = arr->function;
pointerToMessage->destination = arr->destination;
pointerToMessage->hCtlr = arr->hCtlr;
pointerToMessage->stageOnAxis =  arr->stageOnAxis;
```

Vi mangler nu kun at synkronisere og ligge messagepointeren op som en message.

.

```
WaitForSingleObject(syncSemaphore,INFINITE);
pointerToMessageClass->AddItem(pointerToMessage);
ReleaseSemaphore(threadSemaphore,1,NULL);
return 0;
```

ErrorManager

Kan oversætte fejlbeskeder fra de medfølgende api. Klassen bliver ikke brugt i vores api, da error håndteringen er mangelfuld og ukonsekvent. Er mest taget med, i tilfælde at en videreudvikling af api'et.

Translate

```
Translate(
          long errorCode,
          LPCTSTR &error
)
```

errorCode: Fejlens værdi.
error: Kommer til at indeholde fejloversættelsen.

Indeholder en switch til at oversætte fejlbeskeden. Vi vil ikke gå i detaljer, da det fremgår tydeligt af kildekoden hvad der sker.

Test

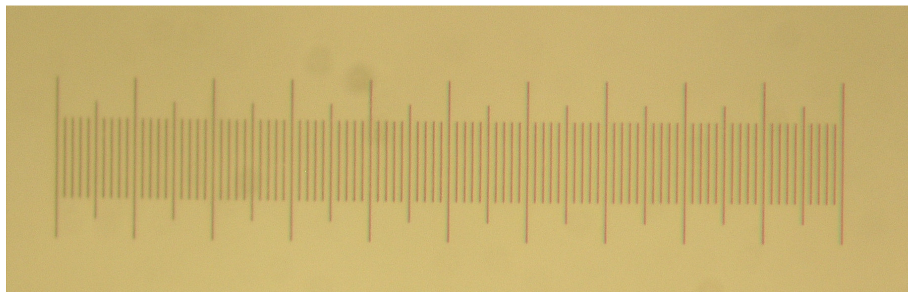
Vi har lavet test af både hardware og software, samt lavet pålidelighedstests af vores målinger.

Hardware

Testene af hardwaren har været begrænset til at teste om stagesne var så præcise som lovet, samt forstærkningen af sensoren.
Test med to sensortyper op mod hinanden, er desuden beskrevet under overvejelserne af sensoren.

Stage præcision

For at være sikre på at længden af et step passede med det der var angivet i dokumentationen, lavede vi en opstilling med en glasskala og et mikroskop. Skalaen er en millimeter med 100 opdelinger det svarer til 0,01 millimeter per opdeling. Til mikroskopet fik vi indkøbt et glaskryds som vi kunne måle op mod glasskalaen.



Figur 18 - Glasskalaen taget med zoom gennem mikroskop uden kryds, da krydset er monteret i linsen som kameraet erstatter.

For at undersøge om præcisionen er som angivet, monterede vi skalaen på en slidestage og observerede skalaen bevæge sig i forhold til krydset. For at observere en bevægelsen på denne skulle vi minimum bevæge stagen 0,01 millimeter. Da stagen er angivet 12.288 steps på 0,5 millimeter i manualen, placerede vi krydset ud for den ene ende på skalaen og rykkede så stagen 24576 step. Krydset burde nu være ud for den anden ende af skalaen, hvilket den også var.

Vi vil dog lige gøre opmærksom på, at testen ikke har kunne teste helt ned i nanometer, som er den skala stagen opererer på.
Testen er altså baseret på et mikroskop der ikke forstørrer tilstrækkeligt til, at det er muligt at se om krydset er præcis ud for skalaen. Desuden spiller den menneskelige faktor ind her. Det ville være mere optimalt, hvis vi havde haft mekanik, der kunne måle i nanometer.

Med turnstagen lavede vi nogenlunde den samme test. Her regnede vi i stedet ud, hvor mange steps der ville gå på en hel omgang og så efterfølgende om det passede. Også denne test viste stor præcision.

Forstærkning af sensor

Det første kredsløb vi selv loddede sammen, havde en præcision målingerne imellem, på ca. 4000 steps med en sensorholder bestående af papirklips og tape.
Det nye forstærkerkredsløb, hvor der var blevet regnet på modstandende til følsomhed og hysterese i kredsløbet og den nye sensorholder, kom vi ned på den præcision som ses i måletestene. Her

sammenligner vi resultaterne for en hel kalibrering med tilbagekørsel. Det er muligt at man med en videre udvikling af forstærkerkredsen og sensorholderen (inklusiv tappen der afbryder sensoren), kan få en endnu bedre præcision. Vi har ikke mulighed for at videreudvikle det.

Software

Testen af softwaret er sket løbende igennem projektet. Vores tests er mest brugt i forbindelse med at fejlfinde i systemet. Da det er et api vi har lavet, har vi ikke lavet egentlige sluttest.
Vi har primært gjort brug at to fremgangsmåder; Visual Studios indbyggede debugger og ved udskrivning af variabler.

Så længe der ikke var tale om at teste kode der indeholdt tråde, kunne vi gøre brug af Visual Studios indbyggede debugger. Med debuggeren har vi kunne følge eksekveringen af kode, og på den måde kunne finde fejl i vores kode.

Når vi skulle teste kode med tråde i, har vi kun kunne bruge debuggeren begrænset. Debuggeren er nemlig ikke så godt at bruge til tråde. I stedet har vi gjort brug af printf(), til at skrive værdier ud på skærmen. Disse udskrivninger er fjernet i den endelige version, da de kun var beregnet til vores debugging.
Det var dog ikke altid at vi kunne bruge printf(), da printf() af ukendte årsager gjorde vores pointere ubrugelige.

Enkelte gange har vi lavet direkte funktioner til at debugge med. Det er blandt andet i testene omkring messages.

Pålidelighed

Pålidelighedstestene går ud på to ting; at teste repeterbarheden og at teste reproducerbarheden. Det er to tests som også bliver brugt ude i erhvervslivet, herunder Oticon.
Test af repeterbarheden fortæller noget om hvor pålidelig systemet er ved gentagende forsøg. Reproducerbarheden derimod fortæller noget om hvor pålideligt systemet er, hvis to forskellige personer skal samle opstillingen af stages og herefter lave hver deres måling.

Repeterbarheden

For at få en ide om præcisionen på sensorerne, har vi lavet et sæt målinger under forskellige forhold. Det har vi gjort for at se om vi kan opretholde de krav vi har opsat. Vi kan ikke umiddelbart lave en optimal nulpunktsmåling, da vi ikke har eksterne måleapparater til at observere de afstande som stagen kan rykke i. Systemet bevæger sig med 0,00004 mm per step og den mindste skala vi har nem adgang til, ligger på 0.01 mm. Selv med denne skala og et mikroskop med otte gange forstørrelse kan øjet ikke opfatte under 200 steps.

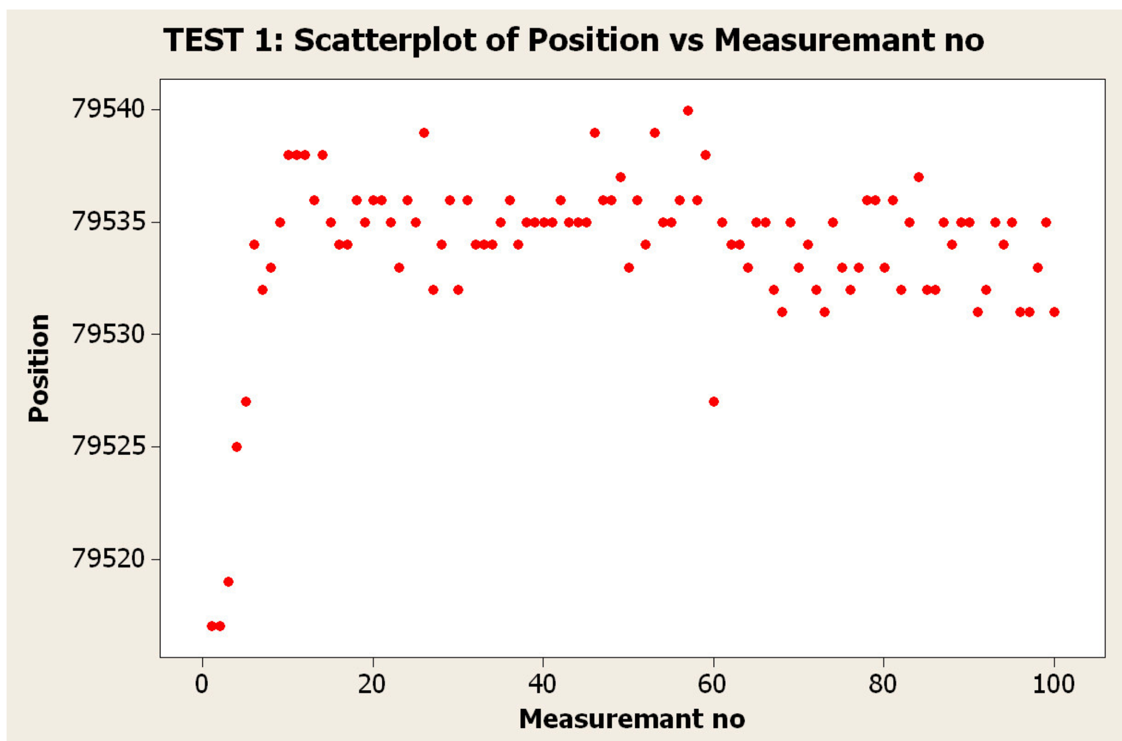
Vi vil ikke angive vores præcisionsmålinger i mm, men i stedet for bruge steps. Vi kan heller ikke måle nulpunktet i forhold til noget fysisk, men kun i forhold til det antal steps som sidste måling kom frem til. Målingerne vil derfor være serier af målinger taget med samme nulpunkt på stagen, så det punkt vi kommer frem til er angivet i et antal steps. Hver gang målingen gentages, vil det sted som sensoren har angivet til nul, ikke nødvendigvis være det samme. Men da det er forholdet mellem hver måling vi skal tage hensyn til og ikke antallet af steps vi kører fra et arbitrært nulpunkt, er dette fint nok.

Vi har lavet ni serier af målinger taget under forskellige forhold. Nogen er dog taget under samme forhold, det er for at se om der er en sammenhæng mellem målingerne; et mønster man kan slå ned på som typisk for systemet.

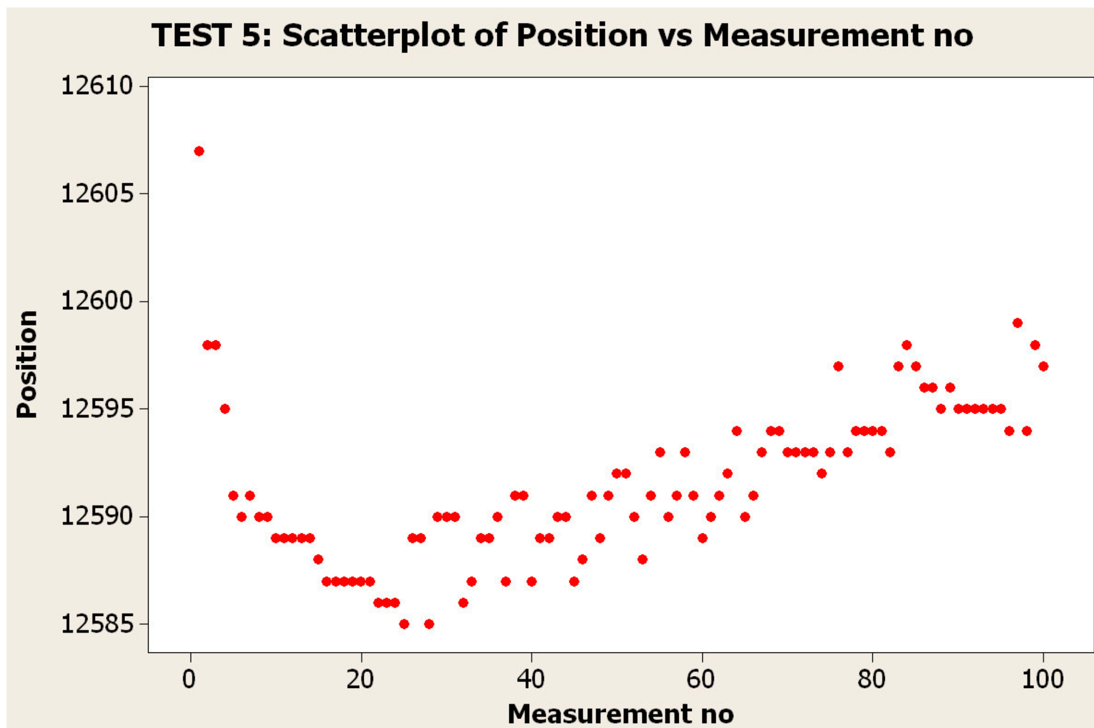
1. 100 enkelte målinger.
2. 51 hele kalibreringer med 10 gentagelser pr. styk med tilbagekørsel.
3. 32 hele kalibreringer med 10 gentagelser pr stk. uden tilbagekørsel i en skuffe (reducering af lysindfald).
4. *Findes ikke.*
5. 100 enkelte målinger.
6. 100 enkelte målinger med lux.
7. 100 enkelte målinger med lux.
8. 100 enkelte målinger i skuffe.
9. 100 enkelte målinger med skiftende lux, angivet i cirka tal.

Der er lavet serier af 100 enkelte målinger hvor stagen har kørt 100 gange i den sidste løkke af kalibreringen. Der er lavet komplette kalibreringer hvor der med ti gentagelser er udregnet et gennemsnit. Dette gennemsnit er angivet som det målte nulpunkt.

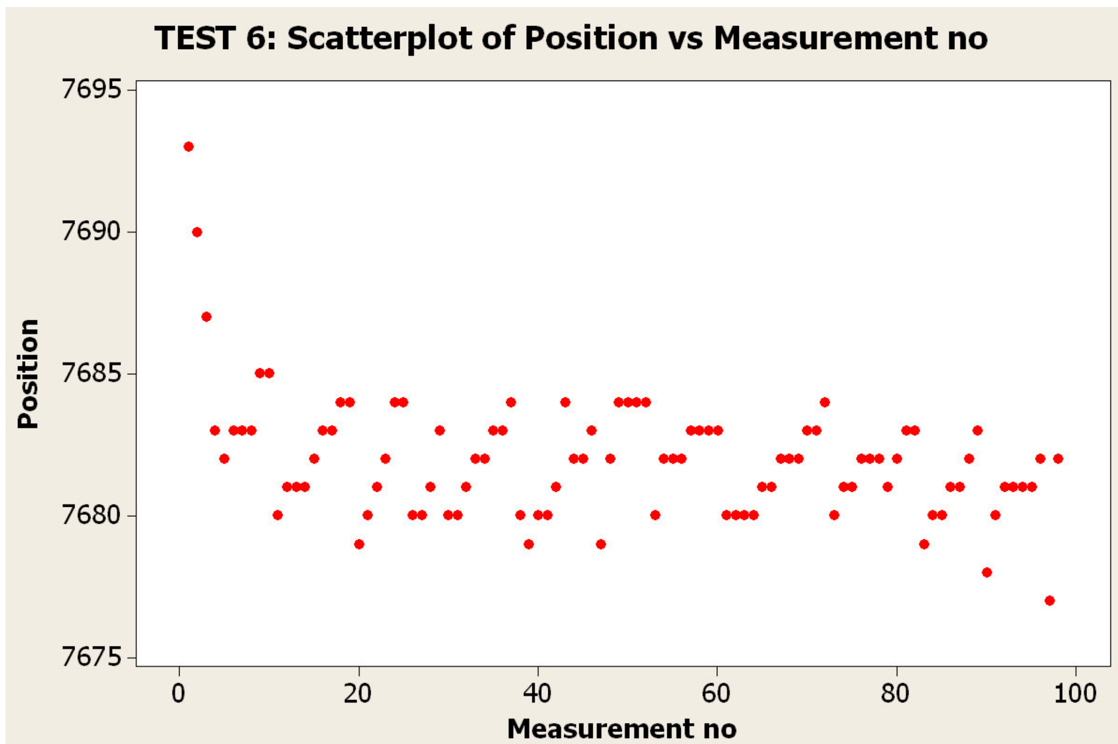
Målingerne 1, 5, 6 og 7, er taget under ens forhold, ved normal belysning som den er ved vores skrivebord. Ud fra disse fire serier, er der ikke umiddelbart noget mønster vi kan komme frem til. Udsvinget på målingerne ligger på ca. ti steps, hvis man ser på målingerne samlet. Der er nogle få målinger der ligger længere fra. Hvor det procentvis kan se ud til at være langt, er det afstandsmæssigt kun ca. 0,0004 mm hvis man ser på fordelingen af serierne.



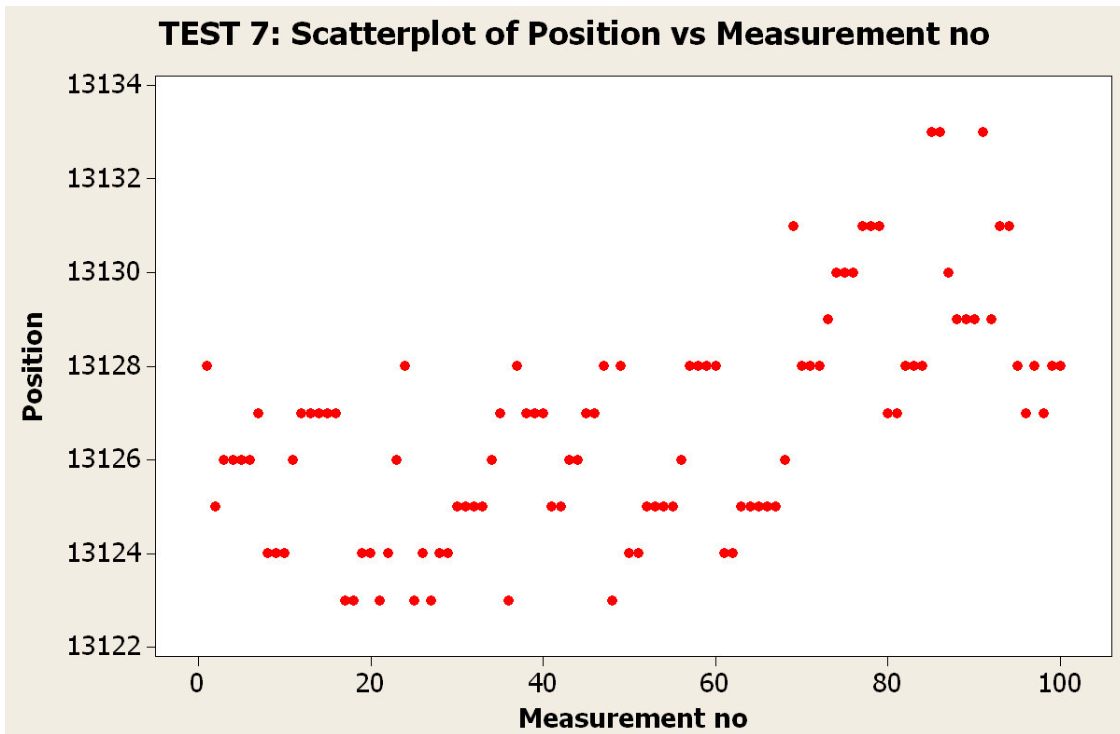
Figur 19 - 100 målinger.



Figur 20 - 100 målinger.

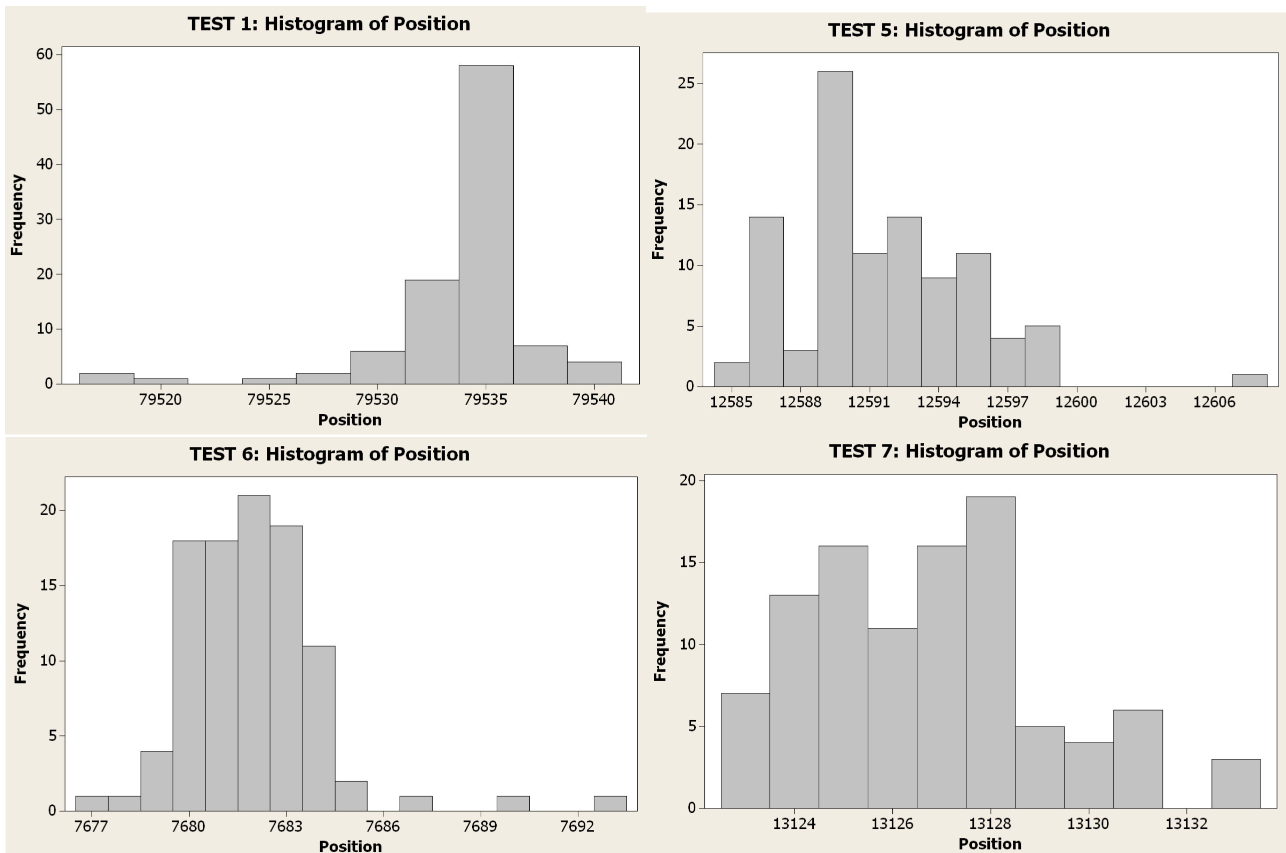


Figur 21 - 100 målinger.



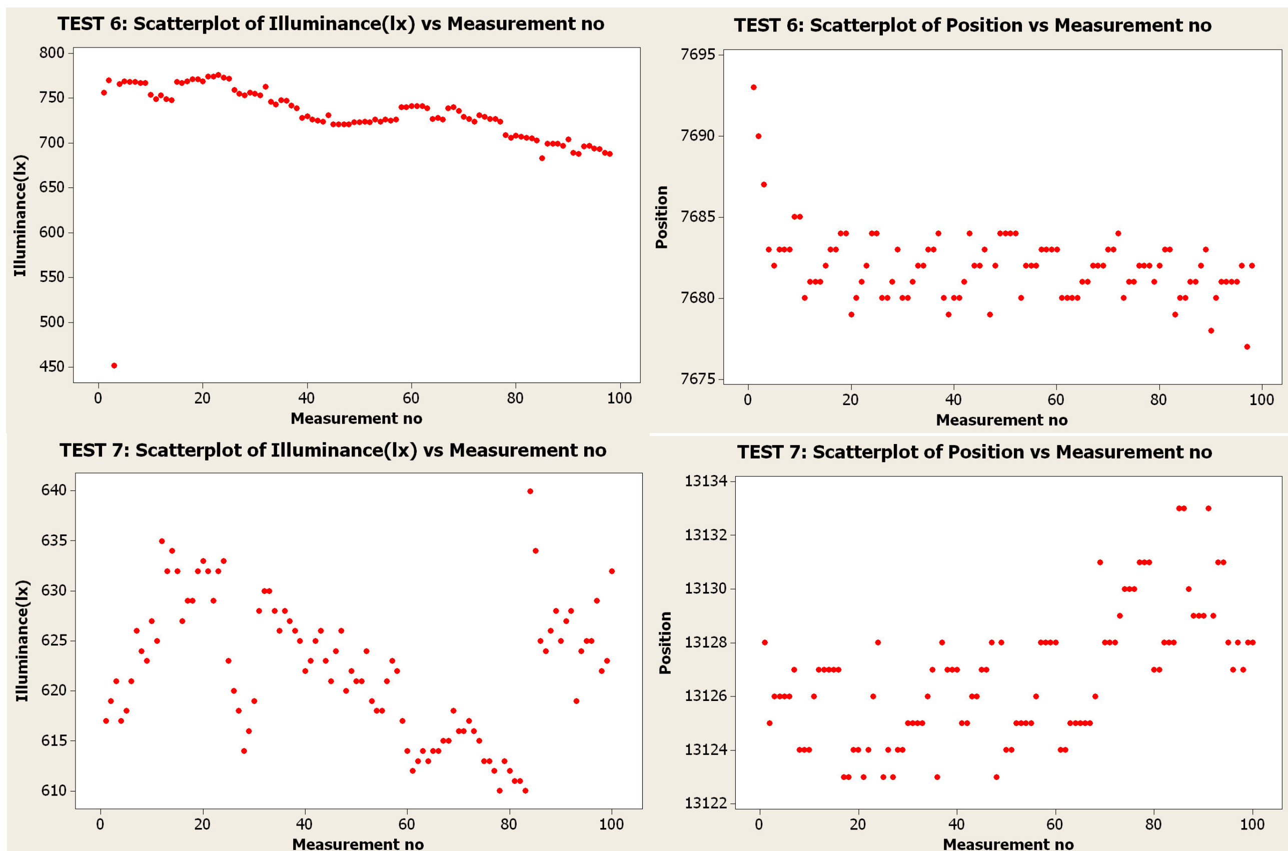
Figur 22 - 100 målinger

På histogrammerne kan man ses hvordan målinger samler sig. Hvorfor nogen af målingerne falder langt ved siden af kan være svært at sige.



Figur 23 - Histogrammer over målepunkter.

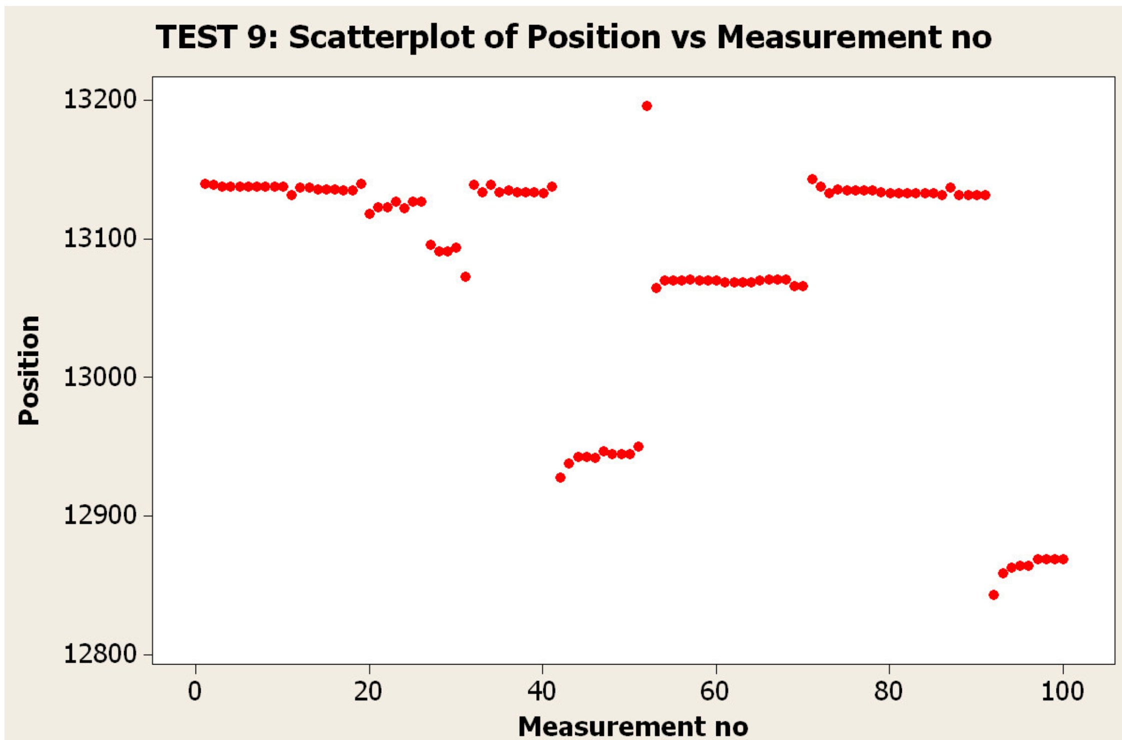
På to af serierne har vi målt baggrundsbelysningen, for at se om der kan være et sammenfaldende udsving i baggrundsbelysningen og i steppositionerne.



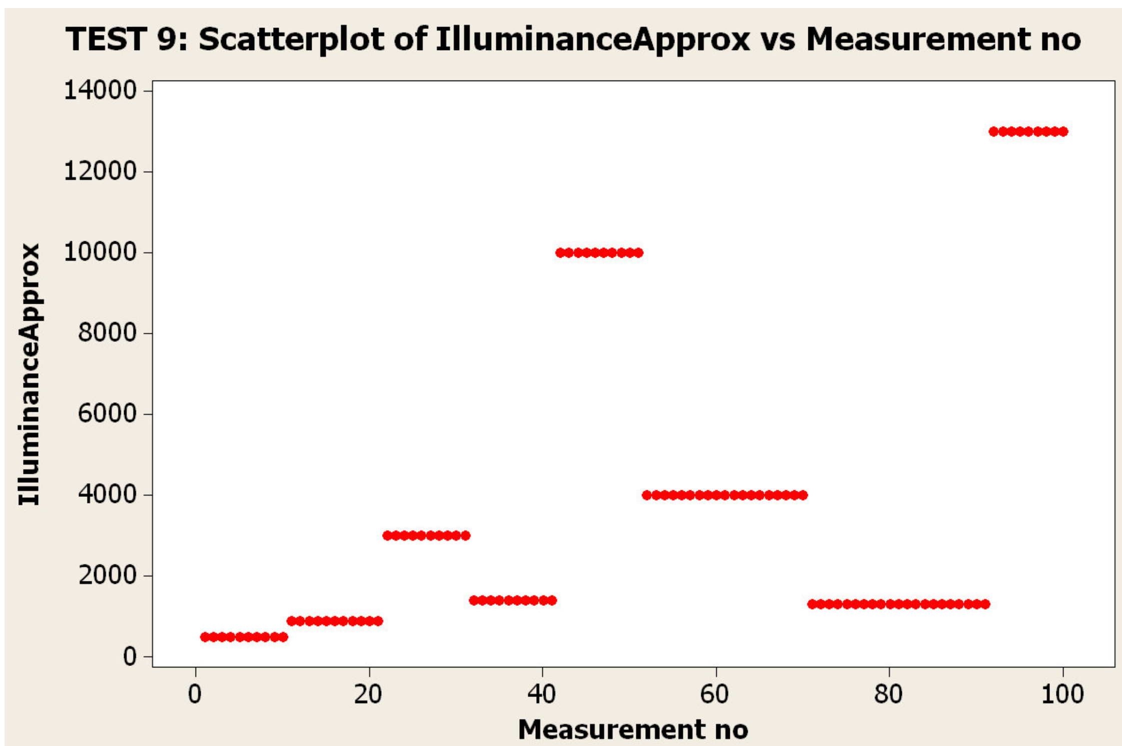
Figur 24 - 100 målinger af baggrundsbelysningen til venstre og steps til højre.

Der er ikke nogen umiddelbar sammenhæng mellem målingen af lyset og udsvinget på lyset i forhold til udsvinget af steps. Her skal dog tages højde for, at hvor målingen af steps er automatisk (aflæst af softwaren når stagen er stoppet), er målingen af lux manuel taget nogenlunde samtidigt som målingen af steps og skrevet ind i et ark. Disse målinger er taget ved normale lysforhold ved vores skrivebord og giver ikke rigtigt noget godt billede af sammenhæng mellem lys og step. Derfor har vi lavet målinger med ekstreme lysforhold.

I test ni er det til at se den påvirkning, ændrede lysforhold har på antallet af steps. Men testen er også lavet under ekstreme lys udsving. Positionen hvor det målte nulpunkt befinder sig, rykker sig sammen med at baggrundsbelysningen ændrer sig.

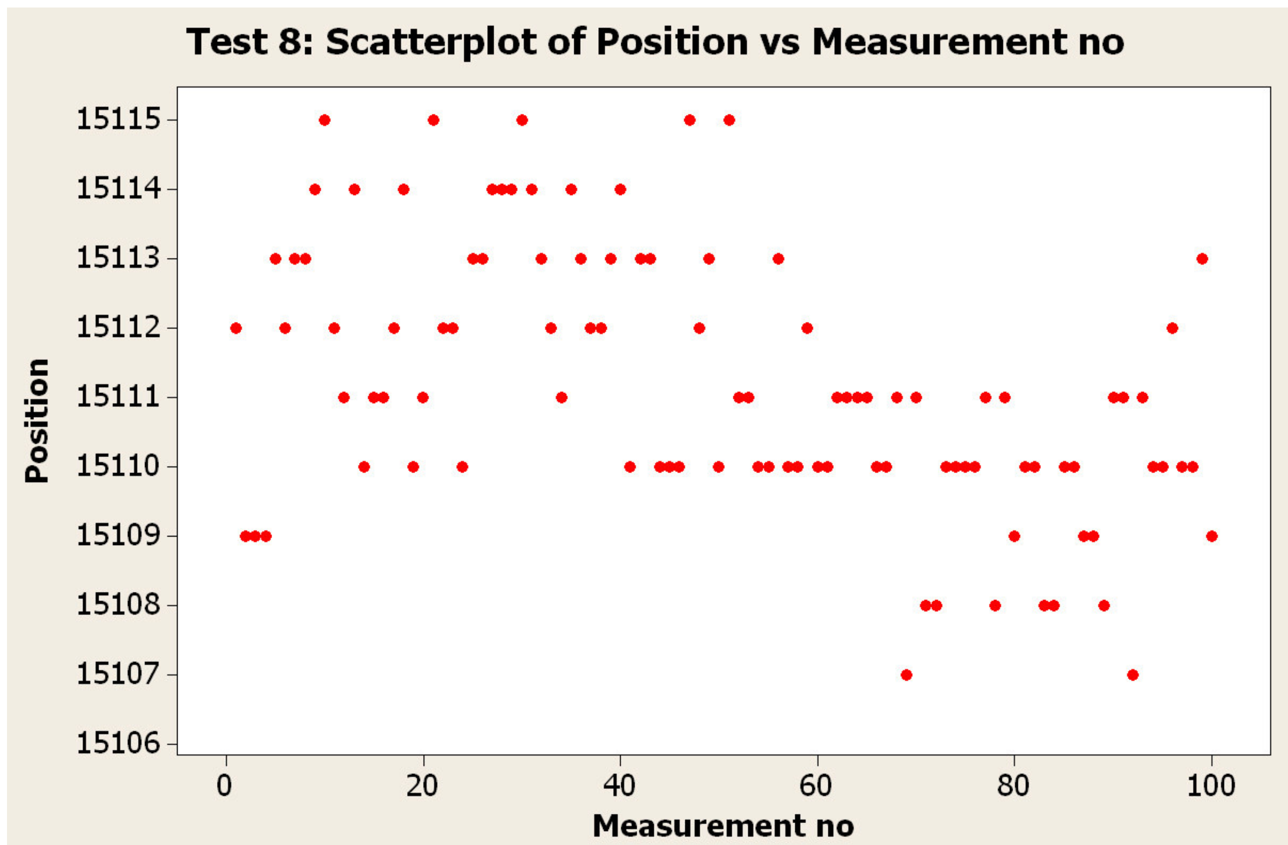


Figur 25 - 100 målinger med varierende baggrundsbelysning.



Figur 26 - Den varierende baggrundsbelysning.

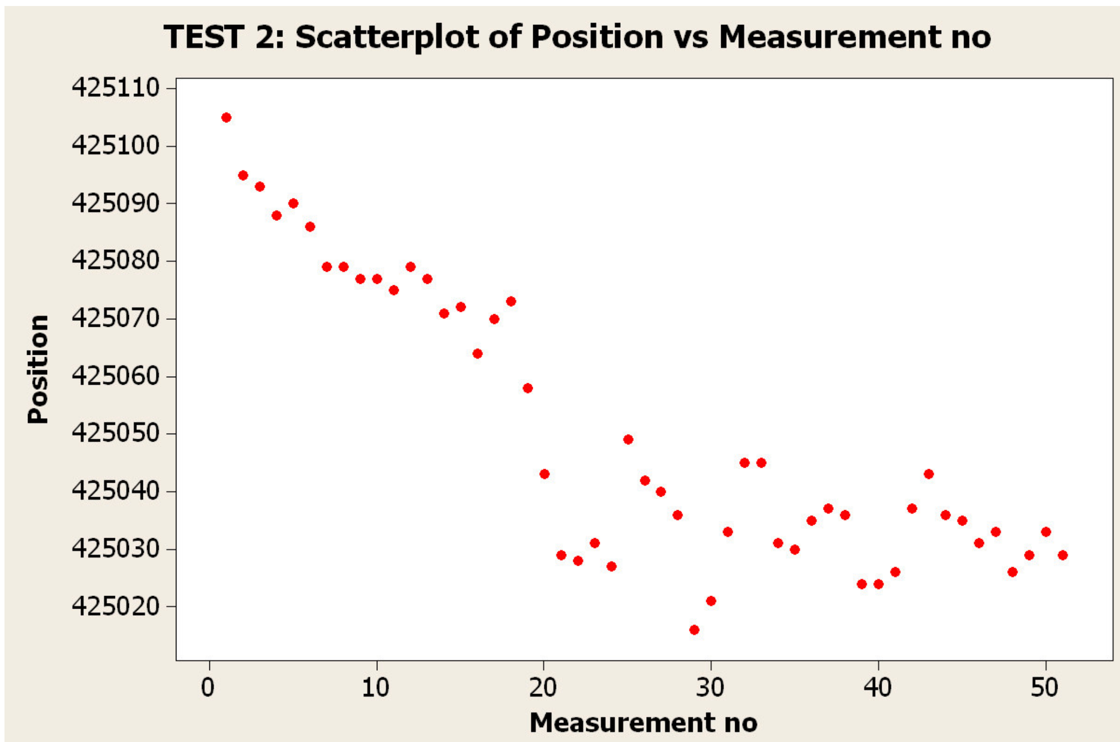
Vi har også foretaget målinger hvor baggrundsbelysningen er forsøgt minimeret, for at se hvordan systemet opfører sig under disse forhold. Det er blandt andet gjort i test 3 og 8.



Figur 27 - 100 målinger i total mørke.

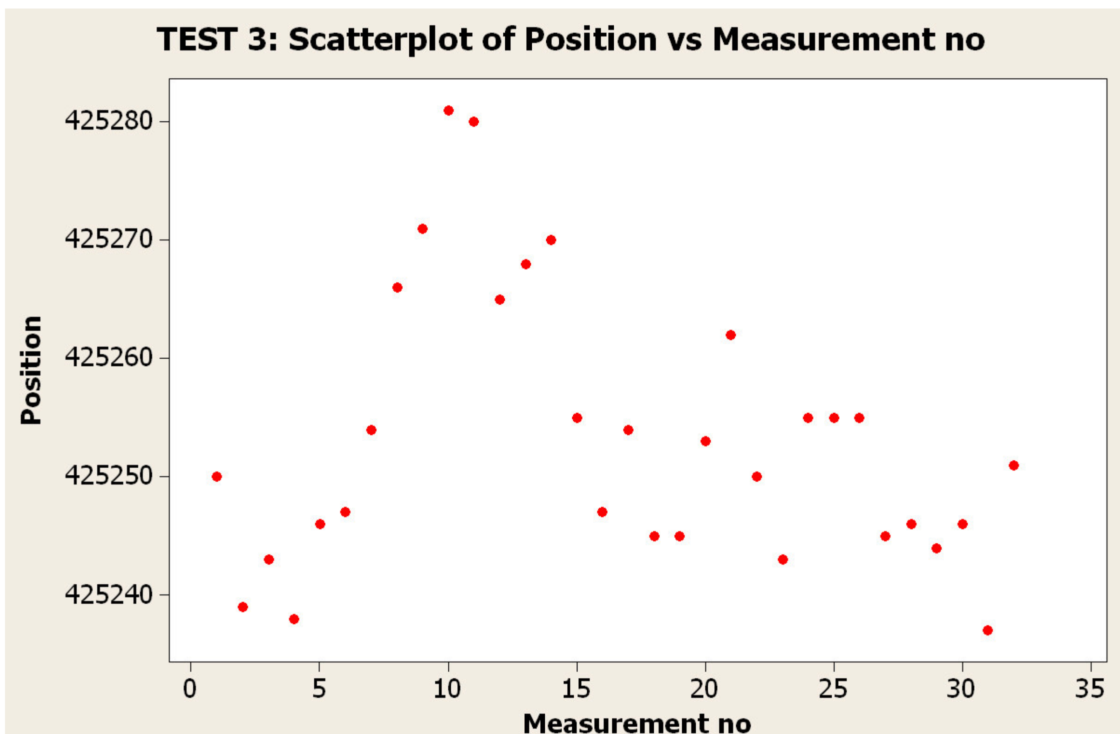
Udsvinget er ikke reduceret noget særligt, men de store udsving finder ikke sted mere.

Da en hel kalibrering foregår på en lidt anden måde (der tages ti målinger og regnes et snit ud), har vi også foretaget serie af målinger på denne måde. Disse serier af 100 målinger er foretaget ved at stagen har bevæget sig 250 steps tilbage og så bevæget sig langsomt mod sensoren igen hver gang. Ved en komplet kalibrering vil kun ti målinger i træk være lavet på denne måde, og så vil stagen køre tilbage til hvor den startede og starte en ny kalibrering.



Figur 28 - 51 kalibreringer.

51 kalibreringer med tilbagekørsel, stagen bevæger sig ca.50.000 steps væk fra sensoren før den begynder næste kalibrering.



Figur 29 - 32 kalibreringer i total mørke.

32 kalibreringer uden tilbagekørsel foretaget i en skuffe for minimering af baggrundsbelysning. Samlet kan vi ud fra vores test drage den konklusion, at vores system er præcist nok til at opfylde vores krav. Under de værst tænkelige lysforhold. Med store udsving, vil forskellen på den højeste og den laveste måling ikke komme over 400 steps, hvilken svarer til 0,016 mm. Man skal dog være opmærksom på at jo større udsving der er i lysforholdene mellem kalibreringerne, jo større forskel vil der være på placeringen af nulpunkterne.

Reproducerbarheden

Reproducerbarheden skal vise om opstillingen kan opnå den samme præcision, efter at have været taget ned og sat op igen. Vi har ikke lavet denne test da den ikke er nødvendig. Systemet afhænger af om sensorerne er placeret på samme sted i forhold til stagen, for at det samme nulpunkt kan findes.

Det ville være utopi at tro at man kan montere sensorerne på samme sted, med så stor en præcision med den nuværende opstilling. Selv om man kunne, ville det ikke være muligt for os at se om det kunne lade sig gøre, da vi er afhængige af det antal steps som stagesne kommer frem til fra det nuværende nulpunkt. Dette nulpunkt forsvinder når systemet tages fra og vi vil derfor miste udgangspunktet for målingerne.

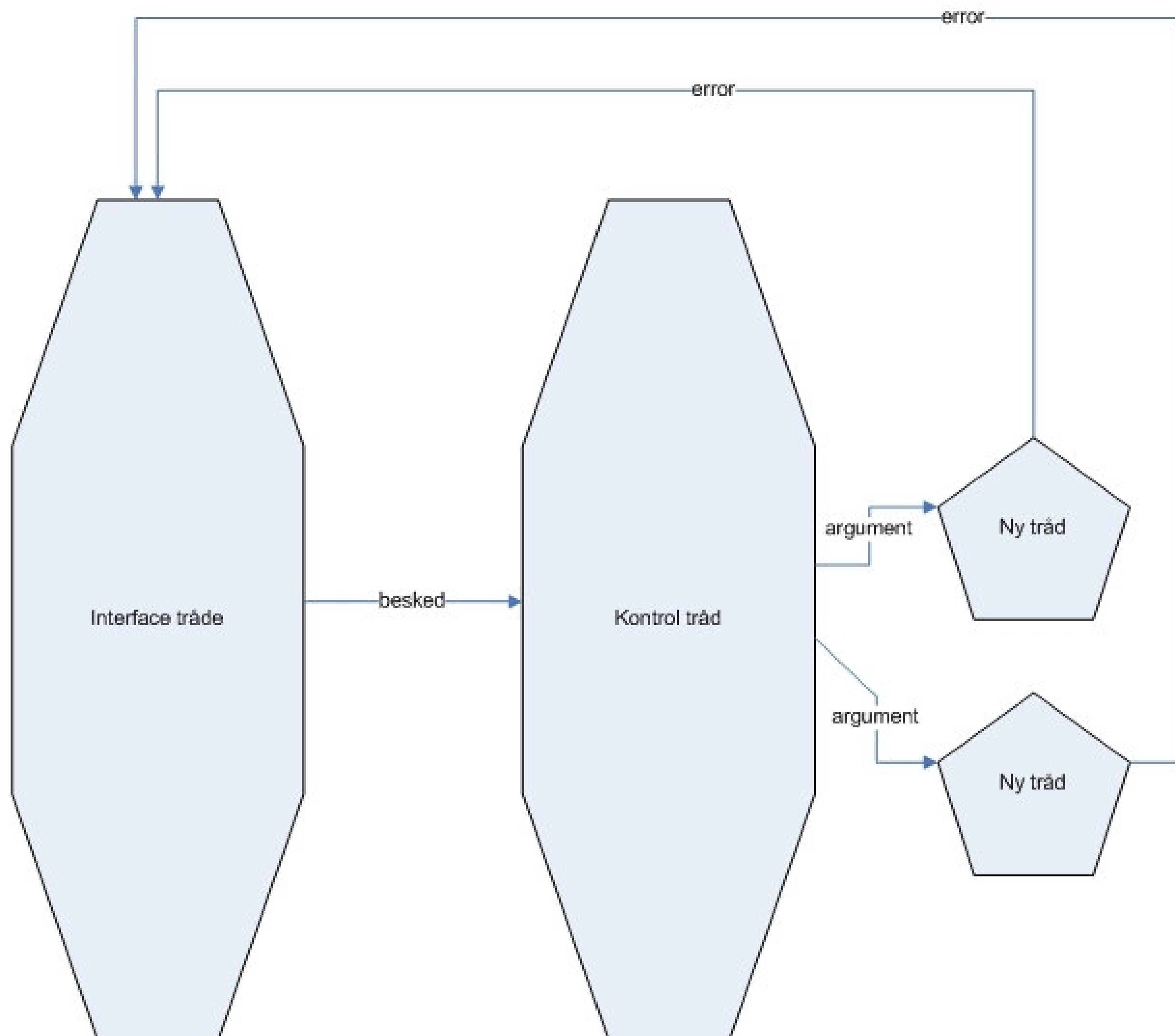
Testen er desuden urelevant, da vores præcision kun er interessant mellem kalibreringerne. Vi har fået til opgave at lave et api, for at genoprette systemet til samme nulpunkt hver gang systemet startes. Derfor er reproducerbarhedstesten urelevant, da man ville opnå en ny opstilling uden forudgående kalibreringer.
Skulle testen have relevans, skulle man have en færdig opstilling, med det udstyr der skulle bruges udover stages og sensorer.

Forbedringer og fremtidige aspekter

Errormanager

Hvis der under afviklingen af en funktion sker en fejl, skal systemet informere brugeren om dette. Som det er lige nu, returnerer vi om der er sket en fejl, sættes status for stagen og gemmer/returnerer fejlbeskrivelsen i en pointer[14].

Et løsningsforslag kunne være som følger: Programmet kører i to forskellige tråde. I den ene kalder brugeren en funktion, der sender en besked til den anden tråd om hvilken funktion brugeren gerne vil have kørt. Den modtagende tråd afvikler så den ønskede funktion. Hvis der under afviklingen skulle ske en fejl, skal der sendes en besked den anden vej med en beskrivelse af fejlen. Brugeren skal så gøres opmærksom på fejlen og eventuelt på hvordan han kan løse den.



Figur 30 - Løsningsforslag til en errormanager.

[14] Ikke konsekvent implementeret, da det ikke virker i praksis.

På grund af opbygningen af programmet med flere tråde skal en pointer til de to besked typer eksistere i de to tråde. Så hvis der lægges et item til listen i den ene tråd, kan man få fat i den anden tråd, eventuelt med en tilhørende semaphore til at overføre information om hvornår der ligger noget i listen. Denne liste skal så være tilgængelig for brugeren, så han kan se når der er noget på listen – altså sket en fejl.

Stagekørsel

Hvis stagen slår fra, rykker den ikke længere og hopper ud af funktionen. Hvis der skal køres en længerevarende test, hvor det ikke er muligt at fysisk at overvåge opstillingen, ville det være en fordel at stagen overvåger om den slår fra og hvorfor. Hvis den kunne analysere årsagen til fejlen, kunne den eventuelt slå sig selv til igen og prøve at rykke det manglende antal steps, så kørslen kan blive færdig og testen kan foresætte.

Messages

Der er flere problemer med den nuværende implementering og den burde omskrives. Det største problem er at når vi returnerer fra funktionen, returnerer vi en pointer vi har oprettet i funktionen. Denne bliver aldrig slettet, så umiddelbart vil vi for hver gang vi fjerner et objekt fra listen efterlade en forekomst af movemessage på stacken, uden at have styr på hvor eller hvordan den skal fjernes igen.
Problemet med at lave dette om med den nuværende implementering, er at når vi returnerer pointeren forlader vi også funktionen. Derfor burde funktionen i stedet tage en movemessage eller bare en message som argument. Som så kunne sættes lig med indholdet af beskeden og så først returnere når beskeden der skulle fjernes var korrekt slettet med et delete kald, altså putte værdierne fra beskeden over i en variabel der var oprettet i den kaldene funktion, og så slette alle lokale variable før vi forlader TakeOutItemBack().

Så ville funktionen komme til at se nogenlunde sådan her ud; hvor værdierne i den message vi skulle hente er lagt over i en variable der er oprettet i den kaldene funktion. Derefter er det objekt der blev oprettet med new kommandoen i AddItem(), fjernet med delete og belaster derfor ikke den tilgængelige mængde hukommelse på stacken efter vi er færdige med at bruge den.

```
Bool Messaging::TakeOutItemBack(MESSAGE mess)
{
		if (last != NULL)
		{
			MOVEMESSAGE* current = last;
			last = current->prev;

			if (current->prev == NULL )
			{
				first = current->next;
			}
			else
			{
				current->prev->next = current->next;
			}
			mess = *current->pointerToMessage;
			delete current;
			return true;
```

```
            }
            else
            {
                 return false;
            }
}
```

Denne rettelse i koden har meget høj prioritering ved videre udvikling, da det er uforsvarligt at gå videre med et program der tager mere og mere af stacken jo længere tid det kører, for til sidst at løbe tør for plads.

Dataintegritet

Når vi opretter en ny tråd skal den have argumenterne som en pointer. Vi bruger en pointer til en datastruktur som indeholder de argumenter den nye tråd skal bruge. Denne pointer bliver så brugt til at køre den funktion, der skal udføres i tråden. Et stort problem for os er, at kontrol tråden som lægger argumenterne ind i denne pointer, godt kan overskrive de værdier som pointeren indeholder før de er blevet lagt ind i en lokal variable i den næste tråd. Dette er et hul i vores program som vi indtil videre har løst med en semafor, der sørger for at der ikke bliver frigivet nogen beskeder, førend de er blevet lagt over i en lokal variabel i den nye tråd. Denne løsning er langtfra optimal da det lidt ødelægger ideen bag besked systemet. Det vi kunne gøre er, at i stedet for at tage beskeden ud af listen i kontroltråden, så bare returnerer funktions variablen og oprette en ny tråd baseret på denne. I den nye tråd kan vi så fjerne beskeden fra listen og bruge argumenterne til at kalibrere eller rykke en stage.

Bevæg single stage, mens aksen kører

Som det ser ud lige nu, kan man bevæge en enkelt stage på en akse, mens aksen er i bevægelse. Det kan man i det tilfælde, at den pågældende stage står stille, mens andre stages på aksen kører. Aksen reserverer nemlig ikke stagesne på aksen, men kun når den skal bruge dem. Det vil kun være et problem i det tilfælde at man rykker enkelte stages, og ikke når man rykker hele akser. Vi vil dog mene at det er en fejl, som bør rettes i eventuelle fremtidige versioner.

Ved fejl bliver status ikke ordentlig resat

Sker der en fejl i systemet, bliver fejlen sat i status. Den bliver dog ikke resat igen og kan derfor blokere for videre brug af systemet.

Automatisk kalibrering af turnstage

Det har ikke været en del af vores opgave, at lave en automatisk kalibrering af turnstagen. Alligevel må man regne det for en udvidelse, der er værd at overveje. Skal man kalibrerer en turnstage, kan man ikke direkte bruge den eksisterende kalibreringsalgoritme. Men bliver man nød til at nøjes med at bruge dele af den eller udtænke en ny kalibreringsalgoritme. Grunden er at turnstagen kan fortsætte rundt i det uendelige og sensoren derfor ikke kan sidde i enden af turnstagen som på slidestagen. En af de ting man skal tage højde for, er at man ikke drejer den forkerte vej rundt. Dette kunne resultere i, at ledningerne på det instrument der måtte sidde på turnstagen, bliver snoet og viklet sammen. Det kunne eventuelt løses ved at have tre tapper i stedet for én. Hvis man placerer en tap på 0° og to tæt på hinanden ved 180°, vil man kunne detektere hvilken ”ende” af cirklen vi er i.

Om der skulle være andre faldgrupper eller hvor og hvordan sensoren skulle sidde, har vi ikke tænkt yderligere over, da det ikke er en del af vores opgave.

Absolute-absolute system.

Eftersom at vi har implementeret en kalibreringsalgoritme, har vi fået et absolut system. Vi kan kalibrere vores stages og få et nulpunkt. Ud fra nulpunktet, kan vi køre til et absolut punkt. Men tænker man over det, kan man sige at vi har lavet et absolut-relativt system. Det skal forstås på den måde, at vi godt nok rykker vores stages absolut, men akserne har ingen indbyrdes relationer. Vi kan altså rykke to eller flere akser, uden at systemet ved hvor vi er. Brugeren skal altså selv have styr på aksernes relationer til hinanden. Skulle man have et absolut-absolut system, skulle man kunne nøjes med at sige at man ville til position XYZ, og systemet ville derefter sørge for at der blev kørt og drejet de rigtige distancer. Vi ville derned opfatte det som et rum med planer, i stedet for uafhængige akser.
Det har aldrig været et krav at opfatte det som andet end uafhængige akser. Og vi ved heller ikke om det ville være hensigtsmæssigt at implementere, eftersom vi ikke ved hvad programmet i sidste ende skal bruges til.

Destructor

Når man ikke bruger vores interface længere, bliver tråde og klasser ikke ordentligt nedlagt. Dette kunne løses ved at have en destructor i interfaceklassen, som kunne dræbe alle tråde og standse eventuelle kørende motorer.

Sensorholder

Sensorholderen er i og for sig udmærket, men vi ser tre punkter hvori den kan forbedres.
Sensorholderen er ikke lukket, og tillader derfor baggrundslys at komme ind. Dette kan være med til at flytte kalibreringspunktet en lille smule, som også kan ses under testene. Med en lukket holder, ville man kunne reducere mængden af lys.
Sensoren sidder lige nu klemt fast i sensorholderen. Det betyder at man kan komme til at rykke sensoren mellem kalibreringerne og derved igen flytte kalibreringspunktet. Man kunne eventuelt spænde sensoren fast, eller lave den sådan at den ikke kunne rykke sig.
Det sidste der kunne forbedres, er at få lavet en tegning over sensorholderen. Lige nu bliver den lavet på skøn, hvilket kan føre til variationer. Det ville være oplagt at få lavet en tegning, så man fik den samme holder hver gang.

Sensor

Vi syntes selv at sensoren vi har fundet er ret god til formålet, hvilket vores tests også har vist. Men vi har ikke ret meget forstand på sensorer, og det må være muligt at kunne finde en bedre. Dette ville dog betyde at man grundigt skulle scanne marked for sensorer og de er formentlig ikke besværet værd.

Turnstage

Vi har ikke implementeret mulighed for at lave en akse af turnstages. En akse bestående at turnstages, giver heller ikke rigtigt nogen mening, da en turnstage jo kan køre ud i det uendelige. Det vil to turnstages jo ikke lave om på. Alligevel burde brugeren have mulighed for det, da vi jo ikke kan vide hvad api'et eksakt skal bruges til.

Vi har heller ikke implementeret en orientering for turnstagesne, hvilket vil sige en positiv eller negativ omløbsretning. Man kan sætte orienteringen, men vi tager ikke højde for den. I en fremtidig version, ville det være hensigtsmæssigt også at tage højde for orienteringen på turnstages.

Save/load konfigurationer

En fremtidig forbedring, kunne være en service hvor man kunne gemme og loade konfigurationerne for stagesne.

Statisk status

Vores statusklasse gør ikke brug af statiske funktioner. Faktisk gør den kun brug af lokale statiske variable. Vi havde problemer med at få lavet en global statisk variabel, samt med at lave statiske funktioner. Derfor har vi lavet en løsning med en lokal statisk variabel. Da vi ydermere havde problemer med at lægge værdien i en pointer, droppede vi pointeren. Vi er ikke helt sikre på hvad det er der får statusklassen til at virke, og i en fremtidig version af api'et, vil det være ønskværdigt at rette koden til, så man ved hvad der sker. Der er umiddelbart ingen fejl ved koden, men det er altid en fejlkilde, når man ikke er sikker på hvad der sker.

Status poster

I vores status-struktur, kan man sætte en given stages position. Vi bruger dog aldrig posten, da vi ved læsning af status, henter stagens position direkte fra stagen. Det vil nok være en ide, helt at fjerne positions-posten fra status, da positionen jo ikke rigtigt er status berettiget. Det ville den være, hvis stagen kun kunne indtage et fåtal af positioner. I stedet skulle brugeren gå uden om vores api og bruge producentens api direkte.
En post der derimod kunne være gavnlig at tilføje er en occupation-post. Denne skulle erstatte move-, calibrated- og calibration-posten. Ikke at man skulle fjerne de tre poster, da de kan være informative for brugeren. Occupation-posten skulle derimod erstatte dem, når vi optager motoren. På den måde ville vi kunne få et mere alsidigt system, der kunne optage motorene, uden at rykke eller kalibrerer dem. Det ville også løse det føromtalte problem, med at rykke en single stage i en akse, selvom at aksen var i gang med at flytte sig.

Grænser for relativ kørsel

Vi har implementeret grænser for at rykke en stage eller en akse. Vi har ikke taget højde for grænser, hvis man kører en single stage relativt. Der er dog kun tale om stages der rykkes relativt i steps. Alle andre former at rykke på, inklusiv at rykke relativt i grader, tager højde for grænserne. Dette vil være nemt at implementere, og bør derfor stå højt på listen over forbedringer. Hertil skal det da lige nævnes, at hvis man kører ud over grænserne, vil der ikke ske andet end stagen disabler og der sendes en fejlmeddelelse igennem systemet.

Interrupts

DCX PCI-100 kortet kan ikke generere et interrupt på pci bussen. Så når der skal tages input fra sensorer gennem dette kort, bliver det nød til at ske gennem polling. Man kunne formodentligt få en hurtigere og bedre kalibrerings algoritme ved at sensorene kunne generere et interrupt direkte til softwaren. Dette kunne opnås ved at bruge en anden måde at få signalet ind i systemet på, eventuelt ved usb eller ved seriel overførsel. Ved at lægge en processor ned på forstærkerboardet, med otte eksterne interrupts og usb interface, kunne kan lade det stå og reagere på sensorenes states og via usb sende beskeder om ændringer til systemet. Derved kunne kalibreringen omskrives til at reagere

på disse interrupts, i stedet for den nuværende polling hver 50. millisekund. Reaktionstiden fra stateskifte på sensorene ville kunne nedsættes betydeligt og derved kunne vi hurtigere opnå en mere præcis kalibrering uden at belaste computeren yderligere.

Konklusion

Konklusionen på projektet er, at det er gået rigtigt godt. Vi har udarbejdet et api, som opfylder alle de krav der var omkring det. Derudover har vi fundet og fået udviklet hardware, som er essentielt for at kunne kalibrerer.

Vores samarbejde med Oticon, har været godt. Vi har haft møder med de folk der kommer til at skulle bruge systemet. Hvis der er noget vi har følt vi ikke selv har kunnet klare tilfredsstillende, er vi blevet henvist til personer der har hjulpet os. Her tænker vi især på hardwaredelen. Vi har også haft en god dialog med DTU igennem hele projektet.

Vi har haft god gavn af at være to om projektet og har arbejdet godt sammen. Vi har hjulpet og suppleret hinanden og opdelt arbejdet på en fornuftig måde imellem os.

Vi har lavet en tidsplan, som vi har holdt os inden for. Til tider har vi været lidt bagud, mens vi til andre tider har været foran. Hver uge har vi haft et møde med enten Oticon eller DTU.
Det har været en rigtig god måde at gøre det på, da vi dermed aldrig er røget af sporet. Har der været spørgsmål omkring funktionaliteten, har vi kunne spørge Oticon, mens programmeringsrelevante spørgsmål har vi kunne rette til DTU.

Vi har lavet et fornuftigt design. I projektets start prøvede vi at danne os et overblik over hvad klasserne skulle indeholde og hvad de skulle gøre. Vi gjorde det også klart, at det var et api vi skulle skrive og ikke et program som sådan. Derfor har vi valgt at se bort fra klassiske use-cases, og holdt os til klassediagrammet. Vi har derudover lavet et diagram over hvordan api'et virker, som vi syntes er beskrivende og har hjulpet os igennem projektet.
Både klassediagrammet og diagrammet over api'et har vi ændret undervejs. Det har vi gjort, når vi undervejs i projektet har fundet bedre måder at gøre tingene på, eller er stødt på problemer som krævede ændringer i designet.

Det kan ikke undgås at løbe ind i problemer, og det har vi da også gjort. Et at de store problemer vi stødte på undervejs i projektet, var omkring trådene. Midtvejs i projektet måtte vi ændre vores tråd behandling til at tage messages.
Det har dog ikke gået ud over tidsplanen, da vi havde forudset at trådene kunne give problemer, og derfor sat lidt ekstra tid af til trådene. Denne nye implementering medførte dog et memoryleak som vil få programmet til at blive ustabil efter en længere periode, dette er vigtigt at få rettet i en videre udvikling.

Vi mangler også at implementere en fornuftig fejlbehandling, så man ved en eventuel fejl, kan se hvilken fejl der er tale om og hvor den er sket. Vi har dog implementeret en vag fejlbehandling, som er ukonsekvent og varierer funktionerne imellem.

Vi har opfyldt kravene fra specifikationerne, fået implementeret en kalibreringsalgoritme og en algoritme til at bevæge flere stages på en akse. Kalibreringen tager mellem et halvt til halvanden minut, hvilket vi syntes er acceptabelt. Specielt når man tænker på at præcisionen gennem testene ligger på ca. 20 steps[15], hvilket svare til 0,8µm og er 125 gange under kravet på 0,1 mm. Når man

[15] Et step svarer til 40 nanometer.

har kalibreret og kører en akse, opnår man en præcision på ét step. Vi har altså afprøvet api'et, og fundet det meget præcist.

Vi syntes derfor selv at vi er nået frem til et rigtig godt resultat, som vi er stolte af. Der er plads til forbedringer. Med den beskrivelse af mangler og måder at løse dem på, vil det være let for de folk der skal implementere det endelige system, at gøre disse ting færdige.

Til sidst vil vi bare sige tak for opholdet hos Oticon og samarbejdet med DTU. Vi føler selv at vi har lært en del af dels praktikken, men også projektet. Vi håber at Oticon vil få brug for vores api, og at dokumentationen gør det let at videreudvikle.

Litteraturliste

Bøger

Robert Lafore; Object-Oriented Programming in C++ 3rd Edition; Sams Publishing
Craig Larman; Applying UML and patterns 2nd edition; Prentice Hall PTR

Hjemmesider

http://www.thorlabs.com – Leverandør af hardware.
http://www.pmccorp.com – Leverandør af api og testsoftware.
http://msdn.microsoft.com – C++ api.
http://www.codeproject.com – Små kodeeksempler.

Software

Microsoft Visual Studio Professional Edition (90-Day Trial).
Medfølgende software; Precision MicroControl Corp. (PMC).
Rational Rose Enterprise Edition; Rational Software Corporation.

Bilag

1. Projektbeskrivelse

Beskrivelse af projektet.

2. Brugsanvisning

En brugsanvisning til hvordan man bruger vores api.

3. API

En beskrivelse af vores api.

4. Kode

Kildekoden for projektet.

5. Hardware

For specifikation for sensor, forstærkerkredsløb, stages og pci-board, manualer til PCI100 api'et, henviser vi til medfølgende cd.