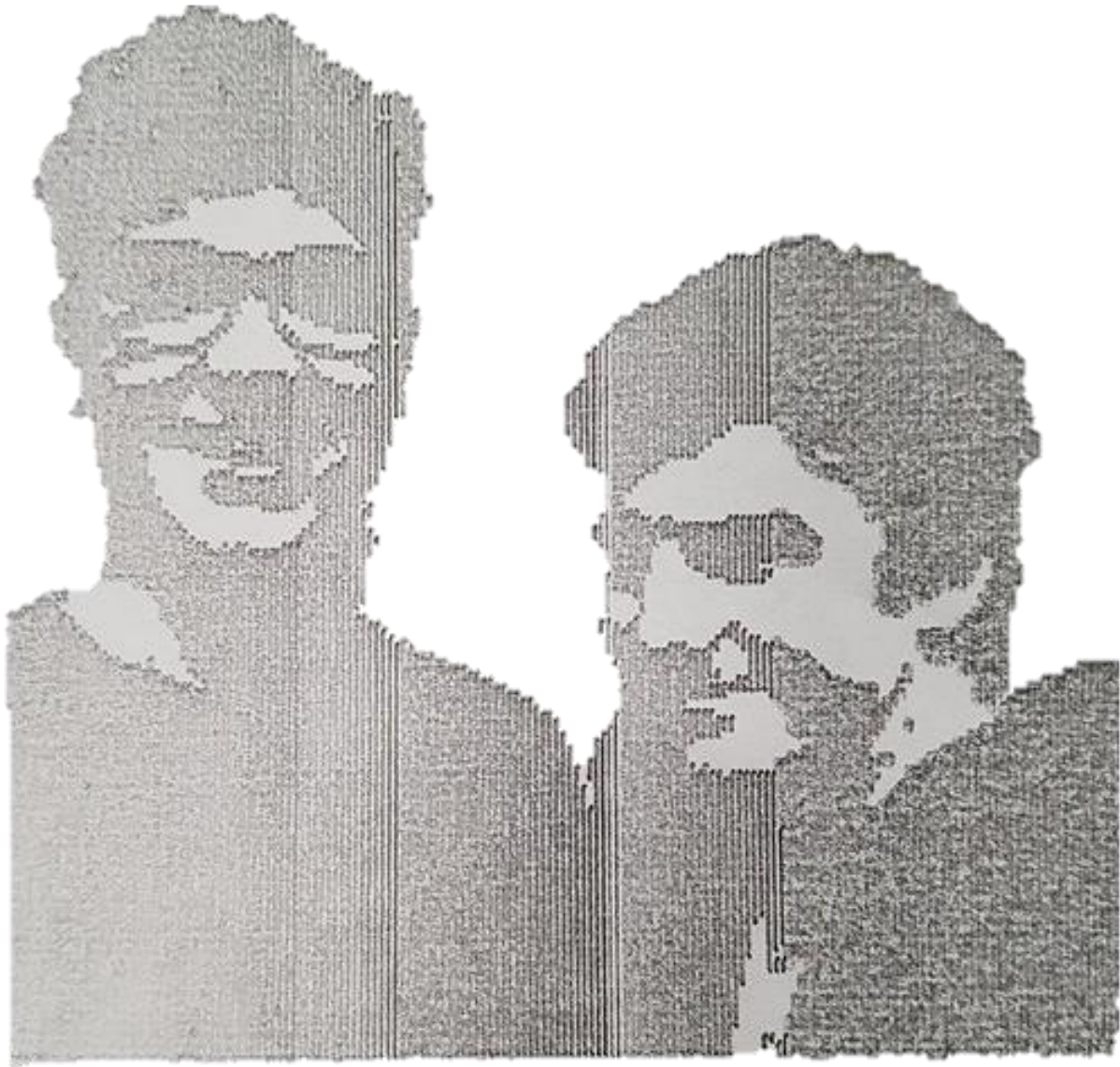


Semesterprojekt

**Gruppemedlemmer:**

Anders Elian 051196
Gustav Nobel 171296
Quan Nguyen 290996
Caroline Kragh 290994
Mads Østergaard 080896
Mads Nielsen 110896
Peter Nielsen 111196

Fødselsdato:**Vejleder:**

Jacob Wilm

Underviser:

Preben Hagh Struge Holm

Projektaflevering:

15 december 2017

Kursuskode:

RD1-PRO

Institution:

SDU 

Resume

Í dette projekt var formålet, at bygge noget software, som skal kunne konvertere et billede om til kode, hvorefter en PLC skal kunne omdanne dette til en tegning ved hjælp af en robot. Derudover skal robotten spidse blyanten automatisk. Vi har omdannet billedet til kode ved hjælp af et java-program, og brugt en PLC til at styre robotten, og genere en bane for robotten at følge. Vi er kommet frem til, at kunne lave et billede om til kode, og efterfølgende få PLC'en til at genere en bane for robotten og tegne ved hjælp af streger. Det blev også muligt at få spidset blyanten automatisk.

Forord

Denne rapport er skrevet af gruppe 8 fra studiet diplomingeniør i robotteknologi, ved Syddansk Universitet, i forbindelse med det givne semesterprojekt i perioden mellem den 12 oktober 2017 til 15 december 2017.

Denne rapport beskriver de problemstillinger, der er dukket op under projektforløbet, med en begrundelse for de valg og løsningsforslag der er blevet foretaget. I rapporten bliver der også beskrevet om samarbejdet i gruppen, og hvordan det har påvirket arbejdet i projektet, samt en konklusion der beskriver slutresultatet af projektet, m.m.

Underskrift

_____	_____
_____	_____
_____	_____

Indholdsfortegnelse

Resume.....	1
Forord	1
Underskrift.....	1
Indholdsfortegnelse	2
Indledning.....	4
Problemformulering.....	4
Tegnerobot	4
Elektronik	5
Driverboard	5
Opsætningen.....	6
Java	9
PictureinOOP	9
Andre klasser.....	11
Image.....	11
Color	12
Message	12
PartImage.....	13
Scale	13
Forbedringer og løsninger	14
Struktureret tekst	18
Opbygning af programmer	18
Globale variabler	18
Reset-program	19
TCP	21
DRAW.....	22
Emergency-program.....	23
HMI - Human Machine Interface	24
Forbedringer.....	25

Resultater.....	26
Tids- og arbejdsplaner	26
Konklusion.....	27
Litteraturliste.....	28

Indledning

I dette projekt er formålet at importere et billede (rgb/gråtone og/eller vektorgrafisk) til et java-program og herfra automatisk generere en bane for robotten at følge. Med en computer skal der kommunikeres med PLC'en og PLC'en får robotten til at bevæge sig. Ud fra denne kode skal robotten generere en tegning af hele eller dele af det oprindelige billede. Når robotten så har tegnet i noget tid bliver blyanten slidt, og det skal derfor være muligt for robotten selv at spidse blyanten.

Det er altså et krav for hele projektet at der bliver brugt et java-program, en computer der kan kommunikere med en PLC og herved tegne et billede. Vi vil have robotten til at tegne stregtegninger og spidse blyanten efter hvor mange steps den har taget på papiret.

Gennem hele projektførløbet har vi arbejdet i hold af 2 og 3 personer og løbende holdt møder for at evaluere fremgangen af de forskellige emner.

Problemformulering

Det primære fokus for dette projekt vil være, at kunne importere digitale billeder, der skal konverteres til data, via et selvlavet program der programmeres i sproget Java. Når dette er gjort, skal den genererede data kunne læses af et andet program. Samme program skal også kunne styre robotten og dermed tegne med en blyant. Desuden vil det andet program blive skrevet i struktureret tekst i Automation Studio. Til sidst i rapporten vil der vurderes og diskuteres, om de problemstillinger der fremkommer af vores løsningsforslag til hver problemstilling.

Tegnerobot

Tegnerobotten er opbygget på den måde, at den arbejder indenfor et tredimensionelt rum. Der bliver anvendt i alt tre stepper-motorer¹ som, hver opererer henholdsvis X-, Y-, og Z-aksen. Robotten har en blyantsholder som kan holde en blyant lodret. Derudover har den også en plade med et stykke papir ovenpå, der ligger vandret. Robotten bliver desuden holdt fast ved hjælp af skinner der fungerer som et skelet for robotten.

Kigger man på robotten forfra, så er X-aksen den akse som bevæger blyantholderen (med blyanten) til højre og venstre side. Z-aksen bevæger blyantholderen (med blyanten) op og ned. Og Y-aksen kan kun bevæge pladen (med papiret) frem og tilbage. På den måde kan man altså tegne på et stort område af papiret.

Til sidst har robotten en blyantspidser, hvor blyanten kan blive spidset med selvbestemte intervaller.

¹ http://www.motechmotor.com/products_detail.php?id=155&cid=71&page=1

Elektronik

Da dette emne er nyt for de fleste i gruppen og ikke er et kursus for dette semester, vil dette afsnit blive mere praktisk end teoretisk. Så i det følgende afsnit gennemgår vi opsætningen af driver-boardet og PLC'en

Driverboard

Driver boardet (Illustration 1 : Driver-board) som vi benytter har 10 porte og 9 knapper. Portene har til formål at forsyne driverboardet, styre en stepper-motor (Motechmotor, 2017) og få signaler fra PLC'en (Akizukidenshi.com, 2017).

Driver boardet har 4 porte til en A og B fase, som bliver tilkoblet til en motor.

Forbindelsen mellem driver boardet og PLC'en sker gennem de 3 porte STEP, DIR, EN som vi forbinder til PLC'ens X3 modul. Der er også en +5V port i samme sektion, som vi kobler til 24V fra strømforsyningen.

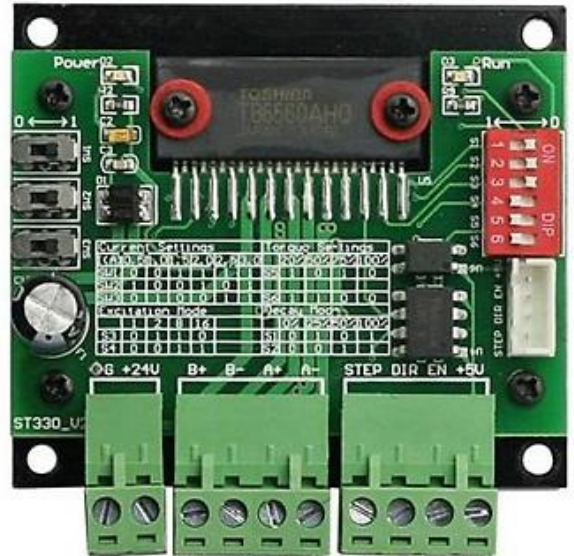


Illustration 1 : Driver-board

De 9 knapper på driver-boardet kan justere indstillinger på driver-boardet. Knapperne hedder som følgende, SW1 – SW3 og S1 – S6 og kan have værdien 0 eller 1.

SW1 – SW3 bestemmer hvor mange ampere driverboardet giver. Vi har valgt at køre med 1 ampere, og det gør vi ved at stille SW1 og SW2 på 0 og SW3 på 1. Så får vi en output current på 1 ampere.

De stepper-motorer vi bruger kan køre med op til 2.8 ampere, men gennem en række test, har vi fået bedst resultat ved at køre med 1 ampere.

S5 og S6 hænger sammen og definerer vores "Static current" som bestemmer, hvor hurtigt vores motor arbejder. Da vi gerne vil have et billede inden for en rimelig tid er S5 og S6 0 da motoren så kører på 100%.

S3 og S4 hænger sammen og definerer "Segments" som bestemmer, hvor præcis og detaljeret tegningen bliver. S3 og S4 har vi givet værdien 0, så vores motor kører med 200 steps på en rotation, og dermed 1,8° per step.

S1 og S2 definerer "Decay mode". S1 og S2 har fået værdien 1, som sætter "decay mode" til 100% hvilket giver den bedste kørsel med motoren.

Opsætningen

Vi bruger tre driver-boards, da vi skal kontrollere tre stepper-motorer, hvor opsætningen for hvert board er identisk, men med forskellige indgange. I tabellen under viser vi, hvor de 3 porte fra driver-boardet bliver tilkoblet til PLC'ens X3 modul (B&R, 2017). Da hvert driver-board styrer hver deres motor, nævner vi også hvilken akse hvert board styrer. Hele X3 modulet bruger vi som digital output (DO) og forsyning af PLC'en.

X3 modul	
DO1	Board x DIR
DO2	Board x EN
DO3	Board y DIR
DO4	Board y EN
DO5	Board z DIR
DO6	Board z EN
DO9 (HIGH-SPEED)	Board x STEP
DO10 (HIGH-SPEED)	Board y STEP
DO11 (HIGH-SPEED)	Board z STEP
24V	Forsyning til PLC
24V	Forsyning til PLC
GND	GND til PLC
GND	GND til PLC

Det elektroniske kredsløb er der gjort få tanker om, da holdet ikke har meget kendskab til elektronik. Primært skal der opstilles en forbindelse mellem PLC'en og de tre driver-boards, som skal forbindes til de tre stepper-motor. Det har vi gjort med følgende opstilling (Illustration 2: Opsætning af robot og Illustration 3: Elektrisk kredsløb).

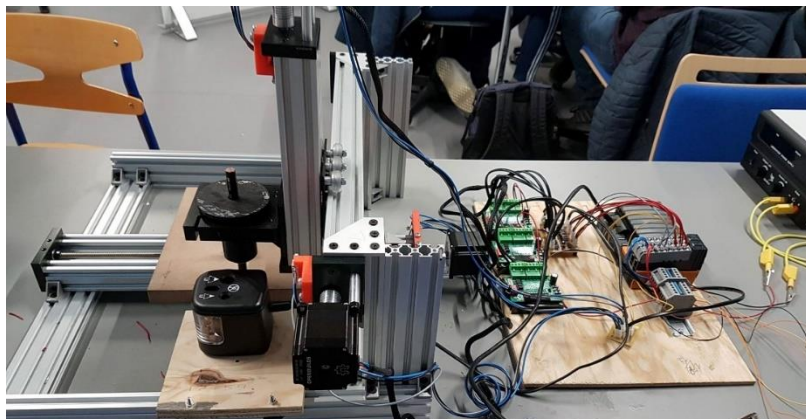


Illustration 2: Opsætning af robot

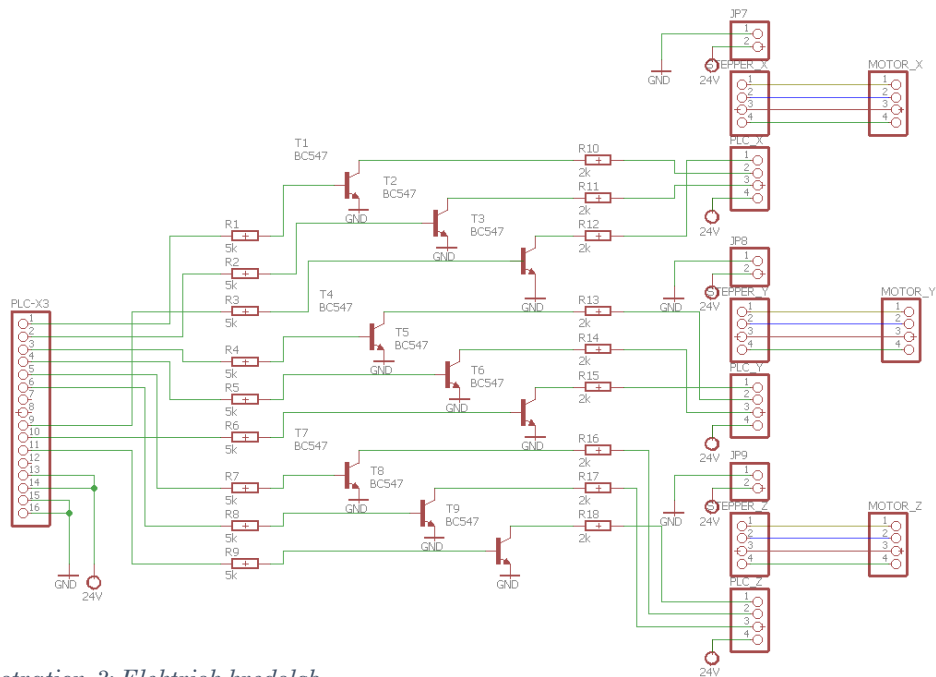


Illustration 3: Elektrisk kredsløb

Vi har fået fortalt at de transistorer vi bruger, godt kan klare 5 millieampere, og da vi ved at PLC'en giver 24 volt, bruger vi Ohms lov til at beregne modstanden mellem PLC'en og transistoren. En transistor indeholder en diode og har dermed et spændingsfald på 0.7 volt som skal med i vores modstandsberegning.

$$\frac{24V - 0.7V}{0,005A} = 4660 \text{ Ohm}$$

Ud fra udregningen har vi overdimensioneret og valgt en 5K ohm modstand. Efter transistoren har vi sat en 2K ohm modstand, som er et krav ved brug af 24 volt, som er beskrevet i databladet. Dette er gentaget for alle de signaler vi benytter.

De 3 stepper-motorer har 4 ledninger gul, blå, rød og grøn, som er i 2 par af gul og blå, og rød og grøn. Den gule og blå ledning har vi tilsluttet B fasen, hvor den gule går til b+ og den blå til b-. På samme måde har vi tilsluttet rød og grøn til A fasen, hvor den røde bliver tilsluttet til a+ og den grønne til a-.

For at vi altid har samme startpunkt efter hver kørsel, har vi valgt at bruge kontakter på de tre akser. Kontakternes funktion er at give PLC'en besked om, at blyantsholderen er kørt tilbage i startposition, og derved skal de tilsluttes til nogle af PLC'ens digitale input (DI). PLC'en har DI på X2 modulet, hvor vi har brugt følgende porte.

X2 modul	
DI1	switch_X
DI2	switch_Y
DI3	switch_Z

For opsætningen af kontakterne, som nævnt tidligere i elektroniksektionen, har vi gjort følgende.

Kontakterne har 3 ledninger (Illustration 4: Kredsløb for knapper). har vi 3 kontakter, en til X-, Y- og Z-aksen. Kontakterne tilsluttes alle 24 volt og GND. Herudover har de hver deres indgang til PLC'en som vist i tabellen over.

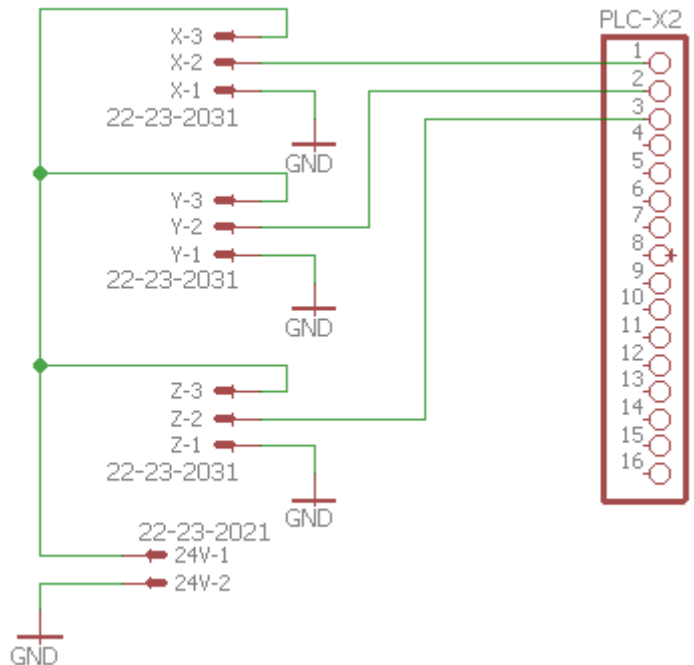


Illustration 4: Kredsløb for knapper

En del af projektet er også at få robotten til at spidse blyanten automatisk, så der skal også monteres en blyantspidser.



Illustration 5 : Kredsløb til blyantspidser

Da blyantspidseren kun kan klare 6 volt, bliver vi tvunget til at bruge en spændingsregulator. Så vi tilslutter spændingsregulatoren til 24 volt og til ground. Dermed får vi 6 volt på "output-benet", som vi kan tilslutte til blyantspidseren. Blyantspidseren skal også tilsluttes GND. Derudover skal der også placeres to kondensatorer. Den ene på input-benet, og den anden på output-benet (Illustration 5 : Kredsløb til blyantspidser).

Java

Java delen af projektet skal kunne læse en billedfil fra en given destination på computeren, hvorefter billedet vil blive lavet om til en String og efterfølgende sendt videre til vores PLC program, som styrer tegnerobotten. Java-koden gennemgås i denne del af rapporten med fokus på kodens main class, *PictureinOOP*, hvor der vil blive henvist til de andre classes i koden. Alle henvisninger til linjer i koden hører til hver anviste klasseoverskrift.

PictureinOOP

I starten af main-klassen befinder hele koden sig inde i et `while(true)`-loop for at den kan køres gentagende gange, hvis ønsket af brugeren. Det første der sker inde i dette loop er kreering af et objekt *sc*, af klassen *scanner*, som stammer fra det importerede bibliotek *java.util.Scanner* fra linje 17 i klassen. Dette bliver benyttet til at tage input fra brugeren.

Kodeblokken på linje 19-32 læser en billedfil (Oracle, LoadImageApp, 2017) (Oracle, Reading/Loading an Image, 2017) fra projektmappen og konverterer filen til et objekt af *BufferedImage*, via et import af biblioteket *java.awt.image.BufferedImage*. Kodeblokken starter med at deklarere variabelen *name* af typen String. Derefter deklareres et objekt af *BufferedImage*, *cImage* initialises med værdien null. *fileName* og *cImage* deklareres her, da de skal bruges i koden senere, og initialises med en brugbar værdi i et `while(true)`-loop derefter. I det førnævnte `while(true)`-loop prompter programmet til at skrive et filnavn tilhørende et billede i projektmappen. *fileName* bliver her sat lig det input, og via en try multi-catch læses filen. I kodens try-del af statementet sætter vi vores før-deklarerede *cImage* lig en billedfil med samme navn som der er blevet skrevet i brugerens input tidligere. Dette bliver gjort via importering af *java.io.file* og *javax.imageio.ImageIO*. Kodens catch tager højde for *IOExceptions* og *IllegalArgumentExceptions*, da de to metoder fra det importerede bibliotek kan kaste dem. Via *ImageIO*'s statiske metode *read* (Oracle, Read, 2017) læses en billedfil der inde i metoden *read* kreeres, som et objekt af *File* (Oracle, File, 2017), hvor *fileName* er et parameter. *File* laver *fileName* om til en fil og *read* kan derfra læse sagte fil, *read* returnerer *BufferedImage* og *cImage* kan altså sættes lig det *read* returnerer. *ImageIO* kan læse .PNG og .JPEG, som blev benyttet til at teste programmet og robotten, men den kan også læse .BMP, .WBMP og .GIF filer. *File* kan kaste en *nullPointerException* hvis *fileName* er lig null, men da vi giver den en værdi via brugerinput, kan den ikke have det på det tidspunkt i programmet, altså gribes den exception ikke i catch-statementen. For at kunne gribe *IOExceptions* er *java.io.IOException* importeret. *IOExceptions* gribes hvis *cImage* bliver sat lig null og *IllegalArgumentExceptions* gribes hvis en error sker i løbet af læsningen af billedet. I denne kode kunne det fx ske, hvis noget forkert er indtastet af brugeren ved brugerinput.

pictureArray, et multi-array af typen byte, kreeres på linje 36 i koden, hvis længde er sat lig bredden af vores valgte billede i pixels. På linje 40 til 45, laves der et objekt af klassen *Image*, og der benyttes en af dens metoder, *assembleImage*, der udfylder det nævnte multi-array (*pictureArray*).

På linje 48 printes der på konsollen til brugeren, hvilket limit der er sat for mængden af pixels, et billede må have før det skaleres, samt hvor mange pixels det valgte billede har. Brugeren bliver promptet i linjerne 51-85 til at vælge imellem det fulde billede, eller en udvalgt del af billedet. Hvad end billedet skal skaleres eller ej, promptes der om det samme. Uden at brugeren nødvendigvis ved det, skaleres billedet, hvis det overskrider billedets pixel *limit*. Limit forklares yderligere i afsnittet Forbedring.

På linje 88-90 deklarereres og initialiseres objektet *roboC* af klassen *RobotClient*. Via variablerne *port* af typen *int* som er sat lig 12345 og *hostName* af typen *String* som er sat lig "192.168.0.103" *RobotClient* er et library givet af vores lektor. Setupet af denne server og TCP forklares i PLC delen af rapporten.

På linje 92 kreeres et objekt, *scaledImage* af klassen *Scale*, så objektet kan benyttes længere nede i koden, inde i de følgende if-statements til at skalere billedet.

På linje 94-221, afhængigt af om brugerens input på linje 51-85 er FP (*full picture*) eller SP (*specific part of picture*), og om billedets originale mængde af pixels overskrider det satte limit, kan der forekomme fire forskellige udfald (Oracle, compareTo, 2017). Et-tallet bagved FP og SP står for det originale billede, hvor et to-tal står for det skalerede billede.

Det første mulige udfald, her for forklaringens skyld kaldt FP1, konverterer hele billedet til en *String* via objektet *code* af klassen *Message*, via metoden *convertToMessage*. Derefter tegner den, for brugerens skyld, billedet i hhv. 1'ere og 0'ere, med mellemrum bagved hvert tegn, via objektet *photo*, og printer omtalte *String*. Endvidere forbinder den til PLC'en, via en server og sender billedets *String*, derefter hvis den er forbundet til serveren, bryder den forbindelsen afhængigt af brugerinput.

På linjerne 104-117 sendes det fulde billede til PLC'en, I denne kode blok sendes den sammensatte *String* der sammensættes i metoden *convertToMessage* fra klassen *Message* til PLC'en. Metoden *connect* forbinder til serveren og videre til PLC'en via objektet *roboC*.

Endvidere, såfremt programmet fortsat er forbundet til PLC'en, prompter det brugeren til at bestemme, hvorvidt den skal afbryde forbindelsen, og såfremt brugeren skriver alt andet end "YES", prompter den brugeren igen via dens *while(true)* loop. Såfremt brugeren skriver "YES", bliver forbindelsen afbrudt, og via et *break* hopper koden ud af *while(true)*-loopet.

I det tredje mulige udfald her kaldt SP1, prompter koden brugeren til at specificere hvilket område af billedet som skal tegnes, og via objektet *part* af klassen *PartImage*, sættes variablen *partMessage* af typen *String* til billedets kode for det specifikke område. Endvidere forbinder den til PLC'en, sender billedets *String*, og via brugerinput afbryder den.

Et fjerde udfald er FP2, hvor koden skalerer billedet og beregner forholdet hvorved der skal skaleres via objektet *scaledImage*, der blev kreeret på linje 92 i koden. Ved hjælp af *scaledImage* tegnes billedet for brugeren med hhv. 1 og 0, med mellemrum bagved, hvorefter billedet bliver konverteret til en *String*, og denne *String* udskrives til brugeren. Endvidere forbinder programmet til PLC'en, sender billedets *String*, og via brugerinput afbryder den.

Det sidste udfald, kaldt SP2, skalerer koden billedet og beregner forholdet hvorved der skal skaleres via objektet *scaledImage*, der blev kreeret på linje 92 i koden. Brugeren promptes til at specificere det område af billedet som skal tegnes, og via objektet *scaledImage* tegnes billedet i hhv. 1 og 0, med mellemrum bagved samtlige tegn, og variabelen *scaledMessagePart* af typen String sættes lig billedets kode for det specifikke område og printes til brugeren. Endvidere forbinder programmet til PLC'en, sender billedets String, og via brugerinput afbryder den.

Ved linje 224-232 bliver brugeren promptet om hvorvidt de vil fortsætte programmet eller lukke programmet, og ud fra hvorvidt brugerens input er "NO" eller alt andet, hopper processen ud af *while true*-loopet som hele koden er i, eller programmet starter loopet forfra.

Andre klasser

Image

Klassen *Image* indeholder 2 attributter, 1 constructor og 2 metoder. De to private attributter er *iHeight* og *iWidth* af typen int, som i constructoren, respektivt bliver sat lig med parametrene *newIheight* og *newIwidth* begge af typen int, som i programmets main, *PictureinOOP*, respektivt bliver sat til billedets højde og bredde.

Denne klasse indeholder metoden *assembleImage*, der benytter parametrene byte *pictureArray*[][] af typen byte, og *image* af typen *BufferedImage*, til at udfylde et multi-array med hhv. 0 og 1, castet til typen byte, for henholdsvis hvid og sort. *pictureArray* er af typen byte, for at spare på pladsen, da der ikke er brug for mere plads for et 1 og 0. Metoden begynder med at kreere objektet *ofPicture* af klassen *color*, dette gøres for at gemme værdierne et samlet sted. Et dobbelt for-loop påbegyndes for at trappesere *pictureArray* fra main (Oracle, Load black and white image, 2017). Inde i det ydre for-loop, der gennemgår billedets bredde, kolonne for kolonne af pixels, sætter et nyt array med en størrelse lig højden af billedet ind i hvert indeks, så *pictureArray*, værdimæssigt, kan fyldes ud. For hvert y- og x-værdi der er indenfor grænserne sat af billedets bredde (y) og højde (x), sker der følgende:

Color har attributterne *rgb*, *red*, *green*, *blue*, og de har hver deres accesmetoder. *rgb* sættes via dens accensmetode med *getRGB(BufferedImage)*, som et parameter, der giver en farvekode for farven af et pixel, eller her, et index i multi-arrayet. Et eksempel på hvordan de tre farver sættes er, "*ofPicture.red = (ofPicture.rgb>>16)&0xff;*" (Classroom, 2017). Man bruger >> til bitvis at gå gennem farvekoden. Den specifikke farve værdi kan findes bytevis og koden gennemgås bitvis derfor med 0, 8 og 16 til at finde værdien for farverne rød, grøn og blå i hvert pixel i billedet. Af disse farver findes gennemsnittet, via objektet *ofPicture*, så vi kan se billedets pixel i hvor "lyse" de er, og dermed ikke i farver. Ud fra dette sættes en ny *RGB*, som er gennemsnittet for hvert pixel, via *getRGB*, en metode fra *BufferedImage*. På linje 43 til 47 sættes hvert pixel/index lig 0 eller 1 castet til byte, afhængig af om gennemsnittet er lig med eller højere end 160, eller om det er lavere. Hvis det er lavere, er farven tæt nok på sort til at koden konvertere det til 1 og omvendt hvis det er over eller lig med 160, konverteres det til 0.

Metoden `drawImage` har parameteren `pictureArray` af typen `byte` og står på linje 54 til 65 i denne klasse. Den benytter et dobbelt for-loop til at trappesere `pictureArray` og printer et 0 eller 1 (visuel reportage af pixels og dets farve) i konsollen, afhængig af arrayets værdi i det givne index i arrayet. For det ydre for-loop gælder det, at `int y` får den givne startværdi 0, og såfremt dens værdi er lavere end højden af billedet, så vil der blive lagt +1 oveni værdien for `y`. Såfremt disse kriterier bliver opfyldt, vil det indre for-loop blive gennemgået. For det indre for-loop gælder det samme, blot med værdien for `int x` og bredden af billedet, i stedet for `y` og højden af billedet.

På linje 57, såfremt at værdien i indekset som processen er nået til, er lig 0, så vil der blive printet et 0 efterfulgt af et mellemrum og såfremt dette ikke er tilfældet, vil der blive printet et 1, ligeså efterfulgt af et mellemrum.

Til sidst på linje 63, sørger koden for, at efter hver række af pixels der er gennemgået, hoppes der ud til det ydre for-loop igen. Her sørger koden for at der printes på næste linje af konsollen og dobbelt for-loopet er færdig med en række. Heraf tegnes billedet række for række.

Color

Klassen `Color` indeholder 5 attributterne `rgb`, `red`, `green`, `blue` og `average`, som står på linje 5-9 og alle af typen `int` og alle er `private`. Klassen indeholder desuden en `no-arg` constructor, samt 6 metoder, 2 accessmetoder og 4 mutatormetoder.

Accessmetoderne i denne kode er som følger: `getAverage` der returnerer attributen `average`, da den er `private`. Derimod lidt atypisk for accessmetoder, beregner den `average` i metoden, og på en måde er den dermed også en mutatormetode. En anden metode er `getCrgb` der returnerer attributen `rgb`, da dette er en `private` attribute. `Crgb` står for RGB værdien i klassen `Color`, og benyttes da det importerede library, `java.awt.image.BufferedImage`, allerede har en `getRGB` metode der bruges i klassen `Image`.

De 4 mutatormetoder gør det muligt at sætte attributterne `rgb`, `red`, `green` og `blue` til en værdi indenfor grænserne af typen `int`, hvilket benyttes i klassen `Image`.

Message

Klassen `Message` indeholder 2 `private` attributter på hhv. linje 6-7, som lyder som følgende:

`upOrDown` af typen `char`, og `message` af typen `String`. Derudover indeholder klassen også en `no-arg` constructor på linje 9-10, som sætter attributten `message` lig en tom `String`.

Klassen indeholder også metoden `convertToMessage` på linje 13, med parametrene `byte pictureArray` og `BufferedImage image`.

`convertToMessage` sætter en `String` sammen, baseret på `pictureArray` fra `main` og kombinationen af følgende bogstaver: 'D', 'U', 'N' og 'Q'.

Den sætter sagte `String` sammen på følgende måde. Metoden trappeserer `pictureArray`, via et dobbelt for-loop der er på linje 14-31. Det ydre for-loop gennemgår billedets pixels via variabelen af typen `int`. Billedet gennemgås række for række, en række for hver `y`-værdi, `y`'s maksimum værdi er altså billedets

højde. For hver række af pixels eller for hver y -værdi, gennemgår det indre for-loop, via variabelen x af typen `int`, hver kolonne, der tilsammen er bredden af billedet, hvilket altså er x 's maximum.

Inde i det indre for-loop sker følgende: Hvis x er under højden af billedet, og x er under bredden af billedet, så sættes attributen *upOrDown* lig enten 'U' for "Up", eller 'D' for "Down", afhængig af om det givende indeks, som koden er nået til, er lig 0 eller ej. Derefter bliver *upOrDown* sat i slutningen af attributten *message*. Dette har en begrænsning via et if-statement for at undgå en *OutOfBoundsException* når billedet ikke har lige lange sider. Altså kan koden kun sende en String for enten et kvadratisk billede, eller kun et kvadratisk stykke af billedet, hvis størrelse afhænger af dets bredde eller højde, afhængigt af hvilken der er mindst. I det ydre for-loop, efter gennemgangen af det indre for-loop, sættes der et 'N' på enden af *message*, som står for at robotten skal tegne næste linje. Efter hele dobbelt for-loopet sættes et 'Q' i enden af *message*, hvilket står for at robotten skal stoppe.

PartImage

Klassen *PartImage* har 2 private attributes: *message* af typen `String` og *upOrDown* af typen `char`. Klassen indeholder også 1 no-arg constructor der sætter *message* lig en tom String og 2 metoder. På linje 13-21 står metoden *messagePart* der via parametrene `int x1`, `int y1`, `int x2`, `int y2` og `byte[][] pictureArray` sætter en String sammen ud fra samme bogstavs kombination som metoden *convertToMessage* fra klassen *Message*. Den sammensætter koden på samme måde, men med ændringen, at y og x i dobbelt for-loopet har startværdier og slutværdier der er sat via brugerinput i `main`. Metoden *drawImage* har samme ændringer som *convertToMessage*, og gør altså det samme som *drawImage* fra klassen *Image*, men ud fra to intervaller af x - og y -værdier.

Scale

Denne klasse består bl.a af 7 private attributes, 3 af typen `int`, 2 af typen `long`, *upOrDown* af typen `char` og *message* af typen `String`. Klassen består også af en constructor og 8 metoder.

Constructoren på linje 18-23 benytter sig af parameteren *BufferedImage image* til at sætte attributen *iHeight* og *iWidth* af typen `long`, til respektivt billedets højde og bredde. Constructoren sætter attributen *message* af typen `String` til en tom String, sådan at den har en værdi før den benyttes i metoder længere nede i koden. Constructoren beregner også mængden af pixels i billedet, ved at gange *iHeight* og *iWidth*.

Metoden *amountOfScaling* på linje 40-56 udregner forholdet hvormed billedet skal skales og sætter attributten n til denne værdi, sådan at resten af de metoder der er i denne klasse har lettere adgang til den. Metoden beregner n ved at dividere mængden af pixels i billedet med et variabel der skjuler attributten n , altså en variabel med samme navn. Derefter hvis *spAmount*, som afhænger af variabelen n , er under det limit vi har på mængden af pixels, eller hvis n er lig 2, så er variabelens værdi godkendt. *spAmount* afhænger af n da den beregnes ved at trække *pAmount* divideret med n fra *pAmount*.

Variablens værdi starter på 1000 og formindskes med 1 for hver gennemgang af det `while(true)`-loop koden er indkapslet i, indtil det bliver godkendt. Attributten n sættes lig variabelen n når værdien er

godkendt og der hoppes ud ad `while(true)`-loopet via et `break`. n kan være lig 2 og blive godkendt i `amountOfScaling` fordi at enten er 2 det eneste tal der går op i billedets pixels, eller ingen gør, med undtagelse af 1, og 2 sættes dermed som standard for billedet.

`getValueN` er en accesmetode der returnerer n , så der er adgang til den i `main`, og derfra via metodernes parametre, har de også adgang.

`getValueSPAmount` er en metode der returnerer attributten `spAmount`, og ligeså returnerer `getValuePAmount` attributten `pAmount`.

På linje 58-76 er der en metode, `drawImage`, der printer billedet ud i 1'er og 0'er, med et mellemrum bagpå, ligesom `drawImage` i `Image`-klassen, men dette sker i forhold til n . Når `pictureArray` trappeseres via et dobbelt for-loop, gennemgås det indre for-loop, hvis y divideret med n ikke har en rest på 0.

Yderligere køres koden inde i det indre for-loop kun, hvis x divideret med n ikke giver 0. Derved fås den effekt at alle rækker hvor n går op i y og alle de kolonner hvor n går op i x , forsvinder.

På linje 78-104 står metoden `convertToMessage`. Den sætter en `String` sammen ligesom `convertToMessage` i klassen `Message`, men med brug af n på samme måde som denne klasses `drawImage`.

På linje 109-131 står metoden, `scaledMessagePart`, der sætter en string sammen ligesom `messagePart` i `PartImage` klassen, men med samme ændringer som i ovenstående metode, `convertToMessage` i denne klasse.

På linje 134-149 står der metoden, `drawImage`, som er en overloading af metoden `drawImage` længere oppe i klassen på linje 58-76. Denne metode printer et billede på samme måde som `drawImage` fra klassen `PartImage`, men med samme ændringer som `drawImage` på på linje 58-76.

Forbedringer og løsninger

Java kodens programmeringsfase startede før der var styr på hvad og hvordan abstrakte klasser og interfaces fungerer, derfor er de ikke inkorporeret. Fokus kunne være blevet sat på interfaces, da koden har metoder som `drawImage` der benyttes i næsten alle klasser. Det kunne også hjælpe på at få sat et system op for programmets klasser. Mange af klasserne kunne have været sat sammen, eller en klasse som `Scale` kunne være blevet splittet op, da der som det er nu, er klasser med få metoder og klasser med mange, og rent ordensmæssigt er det lidt et rod. Her kunne interface og abstrakte klasser have sat et hierarki og dermed gjort det lettere og mere naturligt at få holdt en hvis orden på klasserne.

Vedrørende orden på klasser, så er der en klasse, `Color`. Den blev skrevet, da den første virkende version af koden i starten af projektet, ikke specifikt var objekt orienteret, og derfor skulle omskrives. Det første forsøg på det, var `Color`. Den blev senere unødvendig og blev til intet andet end en samling af farve-værdier og beregning af gennemsnittet af farven i et pixel. Den er altså ikke nødvendig, men

forblev, da tiden ikke var til at fjerne den fra koden og da den ingen negativ virkning har, udover øgelse af dårlig klasseorden som er omtalt længere oppe i diskussionen.

Måden skaleringen i programmet fungerer, er ved at sammensætte en String til robotten og tegne på konsollen, hvor pixels hoppes over i processen og derved forsvinder rækker og kolonner af pixels i billedet. En anden måde at gøre dette, som bibeholder formen af motivet i det originale billede sammenlignet med den nuværende metode, der direkte kun fjerner pixels, er hvor der tages højde for de omkringliggende pixels. Det pixel der er i fokus får sin farve sat i forhold til de omkringliggende pixels som i stedet fjernes.

I den nuværende måde at skalere på, er det ikke muligt at skalere billeder til mindre end halv størrelse. Når et billede bliver skaleret til halv størrelse, fjernes der hver anden værdi i arrayet på y, hvorefter de bliver fjernet på x. Det er ikke muligt for koden at fjerne f.eks. to af tre pixels på både x og y. Den størst mulige skalering ved nuværende metode, hvor koden kun køres en gang, er halv størrelse. Hvis et billede skulle skales yderligere ville det være ved at skalere to gange. Det ville muligvis kræve at der laves et array ud fra det skalerede billede for at kunne skalere mere end en gang. Altså ville man skalere en gang med det originale array og igen med det nye array. Afhængigt af hvor mange gange der skal skales, skal der laves samme mængde nye arrays.

Når programmet tegner et billede i konsollen, ligger billedet på siden. Dette kunne sandsynligvis rettes ved at ændre på rækkefølgen af gennemgangen af indekser. Derudover tegnes billedet via robotten og via konsollen, spejlvendt. Grunden kan ikke siges med sikkerhed, men da det ikke er meningen, er det sandsynligt når billedet læses i main, at det spejles. Dette kunne sandsynligvis rettes ved at gennemgå rækkerne af pixels den modsatte vej i alle String sammensætnings- og tegnemetoder. Altså ved at lade x i de indre for-loops gå fra maksimumværdi mod 0, i stedet for at gå mod maksimum, som koden gør nu.

Endnu et problem koden har er at, af grunde ikke helt klare, kan metoderne der sammensætter en String på baggrund af et helt billede når "FP" er valgt af brugeren i main på linje 57-85, ikke håndterer et billede der ikke er kvadratisk, som ønsket. Det vil sige at de dobbelt for-loops gennemgår kun en kvadratisk del af billedet, for ikke at få en *ArrayIndexOutOfBoundsException*, når der af en ukendt grund, spørges efter et indeks ude for grænserne sat af billedets højde og bredde. Altså har programmet en løsning, men en bedre en, som kunne indføres ville være at gennemgå resten af billedet kvadratisk, stykke for stykke, indtil hele billedet er gennemgået. Hvis brugeren valgte "SP" så kan koden godt håndtere ikke kvadratiske billeder, og dette valg kan benyttes hvis brugeren kender til billedets længde og bredde, eller som anden forsøg hvis koden ikke gav det ønskede output.

"SP" valget kan benyttes som en løsning i ovenomtalte problem, "SP" har dog et problem da, hvis det bliver valgt, så promptes brugeren til at indskrive værdier af typen int, til at vælge hvilken del af billedet der skal tegnes. Hvis værdierne enten ikke er valgt inde for billedets grænser (højde og bredde)

eller hvis de ikke er af typen af int, kan det skabe problemer, som fx kan det kaste en *ArrayIndexOutOfBoundsException*. Dette kunne fixes med et while(true)-loop eller et try-catch-statement.

Til sidst i main, når brugeren promptes til om programmet skal køre forfra eller slukke på linje 244-250, kan et problem opstå afhængigt af om der vælges FP1, FP2, SP1 eller SP2. Programmet venter ikke på brugerens input, men starter automatisk programmet forfra, hvis SP1 er valgt og selvom det ikke er testet, kan SP2 have samme problem.

Byte benyttes som typen på arrayet *pictureArray* på grund af at arrayets størrelse ikke behøvede at være større for at indeholde 1 og 0, castet til byte og derfor er der ikke en grund til at benytte større datatyper. Det var tanken bag det, men et problem opstod i forhold til at programmet ikke kunne håndtere et billede når dets breddes eller længdes størrelse er omkring 256 eller højere. Der var på grund af andre problemer der dukkede op på samme tid, ikke tid til at teste præcis det maksimum der er, men det formodes at hvis de arrays der ligger i multi-arrayet *pictureArray* er større end 256, så er størrelsen af byte ikke stor nok. Altså hvis billedets højde er længere end 256 pixels, så kan et array på den længde ikke være i multi-arrayet.

Det limit, programmet har, for om billedet skal skaleres, er specifikt sat til 257 gange 257. Dette er en sat værdi, som skulle være blevet brugt til at teste den limit der sættes af byte i *pictureArray*. Meningen er dog, at limit afhænger af regnestykket, 65535 - antal af 'N'er i dens sammensatte String minus 1 for 'Q'. Altså for hver række af pixels i billedet skal der fjernes et pixel fra det maksimum limit har, sat af TCP, når der sendes til PLC'en, og da 'Q' står for at robotten stopper, fjernes der også et pixel for det. Da limit er sat til at et billede kan have 257 som højden, så er der et hul i programmet hvor at et billede programmet ikke kan håndtere ikke skaleres til et billede der kan håndteres. Det samme kan ske, hvis et billede skaleres, men ikke skaleres til under 256 gange 256 fordi det originalt er for stort. Altså kan programmet komme i problemer afhængig af størrelsen af billedet eller højden af billedet.

Koden kan forbedres på forskellige måder, her er nogle af de forbedringer der med mere tid ville have blevet indført.

Greyscale delen af koden kan udvides. *assembleImage* definerer farven ved en gennemsnitlig RGB værdi. Hvid defineres med et RGB gennemsnit på 161-255 og sort med et RGB gennemsnit på 0-160. Det ville være muligt at opdele farverne i flere kategorier, hvor opdelingerne er udsnit af RGB gennemsnittet. Herefter ville det være muligt at tildele de nye farveværdier til et sæt nye trykværdier til PLC'en. Hvilket ville gøre det muligt at skabe billeder med flere grå nuancer i stedet for kun sort og hvid. Dog ville det kræve at der blev opstillet flere tegn i vores String, for at definere hvilken grå værdi der skal gives videre til PLC'en, 'U' og 'D' ville ikke længere være nok.

Robotten bruger nogle gange unødvendig tid på at gennemgå billedet. En forbedring ville altså være at mindske denne tid. Et billede har hele rækker eller store dele af rækker, hvor dets pixels er hvide, så

spilder robotten en masse tid på at gennemgå disse pixels, som ikke skal tegnes. Det ville derfor være en optimering at robotten skiftede til næste pixel når der kun er 'U'er tilbage, før det næste 'N' i den String der sendes til PLC'en. På denne måde kan hele rækker erstattes med at gå til næste linje og det samme for det resterende af en række pixels, hvor resten er hvide. Dette ville dog hjælpe mere på Java siden af processen, da der så skal sendes en meget mindre String og altså kan maksimum på billedets størrelse øges. Man ville gøre dette ved den String der sendes til PLC'en bliver sat sammen baseret på bl.a om der, hen til slutningen af en række, fra det nuværende indeks, kun er 0'er. Hvis det er sandt, så sættes der kun et 'N' og ikke et stort og unødvendigt antal 'U'er.

Struktureret tekst

Opbygning af programmer

Samlet set har vi valgt, at lave fire forskellige programmer. Dette har vi valgt at gøre, for at holde det overskueligt, så vi nemt kan finde et bestemt stykke kode. Vi har lavet de fire programmer, som vil blive beskrevet senere i dette afsnit.

- Reset
- TCP
- Draw
- EmergencyStop

Vi har benyttet B&R's hjælpeprogram til at finde informationer omkring struktureret tekst. Dette har været en god hjælp, hvis der er noget man har været i tvivl om.

Globale variabler

Hvis der bliver henvist til en variabel i teksten og den ikke står som en lokal variabel står den her.

Navn	Type	Navn	Type
dirX	BOOL	quit	BOOL
dirY	BOOL	sharpen	UINT
dirZ	BOOL	status	USINT
enabX	BOOL	testEnab	BOOL
enabY	BOOL	z_move_down	BOOL
enabZ	BOOL	z_move_up	BOOL
input	USINT [0..65535]	y_move_right	BOOL
posX	UDINT	y_move_left	BOOL
posY	UDINT	x_move_right	BOOL
posZ	UDINT	x_move_left	BOOL
reset	BOOL	newLineX	UDINT
stepX	BOOL	ValueZ	UINT
stepY	BOOL	penLength	REAL
stepZ	BOOL	placeInArray	UDINT
switch_X	BOOL	placement	UINT
switch_Y	BOOL;	newLineHelper	UDINT
switch_Z	BOOL		

Reset-program

Dette program er lavet til at sætte robotten i den samme startposition, hver gang vi enten starter robotten eller af en eller anden grund ønsker at 'genstarte' programmet. Ud over at sætte robotten i startposition findes der også en række 'testfunktioner', som bruges til at køre med robotten på de forskellige akser. Reset-programmet benytter de knapper der er blevet monteret på de tre akser, som vi kalder x, y og z.

I dette program benytter vi en lille række lokale variable, som kan ses i nedenstående tabel.

Navn	Datatype
resetX	USINT
resetY	USINT
resetZ	USINT
timer	TON

Størstedelen af programmet er indkapslet i et if-statement, som spørger, om *reset = true* og om *testEnab = false*. Dette sørger for, at programmet kører, når knappen reset er trykket ind, og dermed true. Herefter køres der igennem koden fra top til bund. Hvis disse betingelser er opfyldte begynder den på reset-sekvensen, som kører alle akserne tilbage til udgangspunktet. Dette gøres igen ved et if-statement, som spørger om en knap er trykket ind eller ej. Hvis knappen ikke er trykket ind sættes retningen af motoren mod den ønskede position hvorefter den kører med motoren indtil knappen bliver trykket ind. Umiddelbart efter, spørges der om knappen er trykket ind og kører herefter ud igen, indtil der ikke længere trykkes på knappen. Dette kan ses på **Fejl! Henvisningskilde ikke fundet.** Dette gentages for hver af de tre akser, så vi ender ud med et udgangspunkt for resten af programmerne, så vi altid ved, at vi starter i samme position, hver gang vi kører programmet. Herefter sættes *posX*, *posY* og *posZ* til nul, så de kan benyttes senere i koden. Dette gøres ved endnu et if-statement, som spørger om alle akserne er blevet nulstillet, efterfulgt af en lille pause på 100ms, hvorefter værdierne sættes.

```
IF switch_X = TRUE AND resetX = 0 THEN
    enabX:= FALSE;
    dirX:= FALSE;
    stepX := NOT stepX;
ELSIF switch_X = FALSE THEN
    resetX := 1;
    dirX:= TRUE;
    stepX := NOT stepX;
END_IF

IF switch_Y = TRUE AND resetY = 0 THEN
    enabY:= FALSE;
    dirY:= FALSE;
    stepY := NOT stepY;
ELSIF switch_Y = FALSE THEN
    resetY := 1;
    dirY:= TRUE;
    stepY := NOT stepY;
END_IF

IF switch_Z = TRUE AND resetZ = 0 THEN
    enabZ:= FALSE;
    dirZ:= FALSE;
    stepZ := NOT stepZ;
ELSIF switch_Z = FALSE THEN
    resetZ := 1;
    dirZ:= TRUE;
    stepZ := NOT stepZ;
END_IF
```

Illustration 6: Udklip af kode

Derudover er der et stykke kode, som kører robotten hen i den bestemte startposition, som sættes i gang, når der bliver trykket på knappen 'Start'. Som det kan ses på Billede 2 er der opsat et if-statement, som spørger om *quit = false*(startknap) og om *resetX = 2*. Dette sørger for, at der ikke kan trykkes start, før akserne er blevet nulstillet. Herefter køres robotten i position ved hjælp af et if-statement på x-aksen. Der køres 1900 'steps' ud af x-aksen.

Til sidst i dette program har vi seks if-statements, som hver gør det muligt at vi kan styre de forskellige akser. Disse har vi brugt i testfasen, hvor vi evt. skulle finde ud af, hvor robotten skulle starte, hvor blyantspidseren er placeret osv. Vi benytter et boolesk udtryk

```
// if start-button pressed, then go to start position
IF quit = FALSE AND resetX = 2 THEN
  IF posX < 1900 THEN
    dirX := TRUE;
    stepX := NOT stepX;
    IF (stepX = FALSE) THEN
      posX := posX +1;
    END_IF
  ELSE
    reset:= FALSE;
    resetX := 0;
    resetY := 0;
    resetZ := 0;
    newLineX := 2294967295;
    newLineHelper := 2294967295;
  END_IF
END_IF
END_IF
```

Illustration 7: Udklip af kode

```
// move x-axis left
IF (x_move_left AND testEnab) THEN
  stepX := NOT stepX;
  dirX := TRUE;
  IF stepX = FALSE THEN
    posX := posX +1;
  END_IF
END_IF

// move x-axis right
IF (x_move_right AND testEnab) THEN
  stepX := NOT stepX;
  dirX := FALSE;
  IF stepX = FALSE THEN
    posX := posX -1;
  END_IF
END_IF
```

Illustration 8: Udklip af kode

TCP

TCP-programmet er vores modtage-program/TCPserver, der over en internet- eller LAN-forbindelse kan modtage et array fra en anden enhed. Dette array bliver sendt som en streng, men bliver lavet om så det modtages som et array (input) med 8 bit på hver plads. Selve koden er genbrug fra en opgave B&R har lavet med os. Dette betyder også, at selvom der er en del af koden der kan sende data, benyttes det ikke. Når strengen bliver modtaget af PLC'en lægges der et U, D, N eller Q ind på hver plads i arrayet, men da det er et USINT array lægges disse ind som decimaltal. Karaktererne i strengen bliver til nedenstående ascii kode.

$$U = 85, \quad D = 68, \quad N = 78, \quad Q = 81$$

Tallene bliver selvfølgelig sat ind i arrayet fra plads 0. Dette gør TCP'en for de pakker der modtages. Der er så det problem, at som programmet virker kan der kun modtages 1 pakke på 64kb eller mindre, hvilket betyder hvis vores billedetil bliver større end 64kb vil den starte på pakke nummer 2, som den så lægger ind i arrayet fra plads 0 igen derfor begrænses arrayet til 65536 hvilket er en plads mere end hvad en TCP pakke skal kunne håndtere. Dette gøres for at være sikker på, at det er hele pakke 1 der kommer med.

Af de variabler der bruges i denne del af programmet kan der ses en liste nedenfor:

Navn	Datatype
tcp1	TcpOpen
tcp2	TcpServer
tcp3	TcpRecv
tcp4	TcpSend
tcp5	TcpClose
state	UINT

TcpOpen, TcpServer, TcpRecv, TcpSend og TcpClose er alle funktionsblokke fra biblioteket AsTCP.

I starten af programmet åbnes porten og ip-adressen bliver sat. Dette gøres med *TcpOpen*. Herefter starter serveren hvor ip-adressen sættes ind. Dette er så funktionsblokken *TcpServer* der bruges for at starte den. Efter serveren er startet, vil serveren vente på at den modtager en pakke igennem den port der er blevet åbnet, her bruges *TcpRecv*.

I slutningen af programmet lukkes der for serveren. Dette gøres med *TcpClose*, for at der senere kan modtages ny data, uden at PLC'en skal genstartes. Dette kan gøres da serveren åbnes igen med det samme, men venter på at modtage ny data inden den lukker serveren igen.

Serveren bruger ip-adressen som PLC'en er blevet tildelt i konfigurationen. I dette setup bruges der en router så der er mulighed for at tilslutte flere enheder. Dette betyder så også, at der bliver brugt en ip ud fra routerens ip-range, i dette tilfælde bruges 192.168.0.103. der ud over bruges en port, som bliver defineret i programmet under variabel tcp1. Her bruges port 12345. Der kunne bruges en hel række andre porte, dog ikke porte der allerede er i brug som 5900 som bruges af visu-delen(Interfacet).

DRAW

Draw programmet er den del der læser arrayet, og ud fra hvad der sker i arrayet *input* sørger for at robotten bevæger sig.

I programmet bruger vi en række lokale variabler, som kan ses neden for.

Navn	Datatype
Activator	BOOL
tempX	UDINT
Timer	TON
sharpenderLen	UINT
i	UDINT

Først i koden testes der, om robotten er blevet nulstillet

```
IF (reset = FALSE AND testEnab = FALSE AND penLength > 5 ) THEN
```

Illustration 9: Udklip af kode

”reset”, og om der er blevet

sat en længde på blyanten ”penLength”. Den tester også om variablen ”testEnab” er falsk, hvis alle disse er sande i forhold til det if-statement de ligger i, vil den gå ind og teste, om der er nogle af knapperne ude i akserne der er blevet trykket ned. Hvis dette er tilfældet stoppes robotten, da dette er et tegn på at der er en akse der er kørt for langt i en retning den ikke skulle køre i. Herefter sættes variablen ”status” da denne viser, i visuen, at programmet kører.

Efter alt dette bliver der testet om blyanten skal spidises på variablen ”sharpen”. Hvis variablen er nået til 30000 vil den flytte blyanten over og spidse denne. Dette gør den ved først at flytte blyanten op af z-aksen. Hver gang den tegner sætter den variablen ”tempX” som bruges for at robotten kan huske, hvor den er noget til på papiret. Dette udnyttes i den næste del, hvor robotten bevæger sig hen af X-aksen mod blyantspidseren. Da blyantspidseren er placeret under nulpunktet ved robotten, hvor den skal køre hen af x-aksen for at spidse blyanten. Nå den så er nået til nulpunktet bliver blyanten sænket ned i spidserne. Hvor langt blyanten bliver sænket ned, kommer an på, hvor lang blyanten er sat i ”penLength”. Her bruges formlen $\frac{45000}{penLength}$ som giver antallet af steps den skal gå ned, plus en værdi der er sat til 1500 steps fra 0 på z-aksen. Denne værdi er sat ud fra at det er den længst mulige blyant der er vurderet til at kunne blive spidset. Når blyanten har været nede i spidseren i 6 sekunder, hvilket er

bestemt af variabelen "timer" vil den blive taget op af spidseren og bevæge sig tilbage til der, hvor den stoppede.

Hvis variabelen "sharpen" ikke er nået til 30000, vil den begynde at tegne. Her bruger den så arrayet "input". For at holde styr på hvor den er nået til i arrayet bruges variabelen "placeInArray" hvor den så lægger, hvad der er på den plads i arrayet, ind i variabelen "placement". "placeInArray" ændres hver gang variabelen "i" når 30, hvilket den gør når der er taget 15 steps på enten x- eller y-aksen hvilket sker mens den tegner. Dette gøres for at kunne bestemme størrelsen af de pixels der tegnes. Grunden til hver pixel kun er 15 step store selvom *i* er 30, er fordi stepper motorerne reagerer på et højt signal, altså når variabelen for step bliver boolsk lav, på grund af transistorerne som inverterer signalet. I programmet bruges "i" kun sammen med variablerne "stepX" og "stepY" da der tegnes i 2 dimensioner.

```
IF (placement = 68 AND i < 30) THEN
  IF (posZ < (3200-ValueZ)) THEN
    IF dirZ THEN
      stepZ := NOT stepZ;
      IF (stepZ = FALSE) THEN
        posZ := posZ + 1;
      END_IF
    ELSE
      dirZ := TRUE;
    END_IF
  ELSE
    IF dirX THEN
      stepX := NOT stepX;
      i := i + 1;
      IF (stepX = FALSE) THEN
        posX := posX + 1;
        newLineX := newLineX + 1;
        sharpen := sharpen + 1;
        tempX := posX;
      END_IF
    ELSE
      dirX := TRUE;
    END_IF
  END_IF
```

Illustration 10: Udklip af kode

For at tegne pixels bruges "U" og "D" som i ascii er 85 og 68. "U" betyder at der ikke skal tegnes, mens "D" betyder at der skal tegnes. Så det første der sker er, at der bliver testet om blyanten er nede eller oppe. Hvis højden ikke stemmer overens med den nuværende plads i arrayet, altså "U" eller "D", vil den først bevæge sig af z-aksen, enten op eller ned, hvorefter den vil bevæge sig af x-aksen for en pixel altså 15 steps.

Y-aksen bevæger sig kun når "placement" er blevet ascii for "N", altså 78, hvilket betyder, at der skal laves en ny linje.

```
IF placement = 78 THEN
  newLineHelper := newLineHelper - 17 ;
  posX := posX + 17;
END_IF
```

Illustration 11: Udklip af kode

Den starter med at z-aksen køre op til 2900 minus variabelen "ValueZ". (ValueZ er sat ud fra penLength igennem ligningen $penLength^{1.5}$ denne virker desværre ikke med alle længder, så der skal findes en anden konstant) Efter blyanten er rykket op, bevæger robotten sig tilbage af x-aksen ud fra variablerne "newLineX" og "newLineHelper" disse 2 variabler sørger for, at hver linje starter det samme sted. Dette gøres bl.a. ved at trække 17 steps fra "newLineHelper". Hver gang der laves en ny linje springes der ca. 17 steps over, og derfor lægges der 17 til variabelen "posX" igen for at kompensere for motorerne. På grund af alt dette starter både "newLineX" og "newLineHelper" meget højt, så der ikke laves et overflow.

Nå den så er flyttet tilbage af x-aksen rykker den y-aksen 1 pixel altså 15 steps ved hjælp af *i*.

Emergency-program

Vi har valgt at lave et dedikeret program til robotstens 'Emergency Stop'. Vi har gjort dette, da vi vil være sikker på, at motorerne stopper, når man trykker på knappen. Ved denne løsning køres

programmet hvert 100ms og dermed får vi en hurtig reaktion når der trykkes på knappen. Som det kan ses på billede 4, er programmet ret simpelt. Det eneste der sker er, at et if-statement spørger efter variabelen 'emergencyStop', og hvis denne er true, så sættes enabX, enabY og enabZ til true, hvilket stopper stepper-motorerne.

```

IF (emergencyStop) THEN // emergency button
    enabX := TRUE;      // disables x-axis stepper
    enabY := TRUE;      // disables y-axis stepper
    enabZ := TRUE;      // disables z-axis stepper
    status := 0;

END_IF

```

Illustration 12: Nødstop

Derudover findes der et stykke kode der styrer knappen 'reset'. Hvis denne bliver trykket sættes variabelen 'reset' til true. Dette gør, at programmet 'Reset' starter med at køre, og sætter robotten tilbage til startpositionen. Derudover sættes en række andre variable, som er nødvendige for at kunne starte reset-programmet.

```

IF (reset_quit) THEN // Reset button
    reset := TRUE;      // starts the reset sequence in reset-program
    quit := TRUE;
    status := 1;        // sets status light to 1
    reset_quit := FALSE;
    placeInString := 0; //resets place in array |

END_IF

```

Illustration 13: Startknap

HMI - Human

Machine Interface

Vi har valgt at lave et grafisk interface til at styre tegnerobotten. Vi har lavet dette fordi det giver et rigtigt godt overblik over de forskellige variable og på den måde kan vi følge med i, hvad robotten gør. Vi har valgt at lave tre forskellige sider, med hver deres funktion. På hver af disse sider findes der et 'Common layer'. Dette lægges som et lag ovenpå de enkelte sider. Her har vi placeret de vigtigste ting, såsom 'nødstop', 'reset' og 'start'. Derudover er der også en inputboks, hvori man indtaster den længde der stikker ud af blyantsholderen. Dette benyttes i koden i programmet 'Draw'. Til sidst findes der tre knapper, som fører hen til de sider, der findes i interfacet (Kan ses i Bilag 1).

Vi har lavet siderne 'Information' og 'Konfiguration' til fremtidige tilføjelser. Disse er tiltænkt, som de beskriver, forskellige informationer om position, variables værdier, evt. billedet den er ved at tegne osv. Derudover var ideen, at man kunne sætte nogle indstillinger i konfigurationen, om evt. blyanten man benytter, tiden der skal spidses osv. Disse sider er endnu blanke, men kan ses i Bilag 2 og 3.

Til sidst er der en side dedikeret til testning. Her kan der ændres på forskellige variable, som det kan ses i bilag 4. Derudover er der knapper, som hver kan køre robotten i forskellige retninger. Til sidst er

der indsat variable for positionerne på de forskellige akser. Denne side har været rigtig brugbar gennem størstedelen af forløbet.

Forbedringer

Da dette programmeringssprog er nyt for os alle har vi nogle ting, vi ikke har haft mulighed og tid til at få lavet. Disse ting er alle løsninger der hver løser mindre problemer eller gør tegningen af billedet mere effektiv.

Den ting, som vi ser som den største forbedring vi kan lave er, at få robotten til at tegne hurtigere. Som det ser ud nu, så tegner robotten en linje, hvorefter den kører tilbage til start og tegner endnu en streg. Derfor ville det være en fordel at få robotten til at tegne billedet både på vej frem og tilbage. Dette kan gøres på flere forskellige måder. En måde at gøre det på er, at lave om i java-koden, sådan at den sammensætter strengen, der bliver sendt videre, skiftevis fra venstre til højre og derefter fra højre til venstre. Dette gøres ved at gennemgå bredden af billedet, skiftevis fra højre til venstre og venstre til højre. Derudover kræver det også en smule ændringer i PLC-programmet. Dette ville gøres ved ændre koden for linjeskift. Dette gøres ved kun at flytte y-aksen og at have en variabel der kan fortælle, om man er i den ene eller den anden ende af billedet.

En anden væsentlig ting ved vores program er, at alle de billeder vi tegner bliver spejlvendte. Dette kan dog ændres ret nemt ved at sætte robotten til at starte i det modsatte hjørne. Vi har valgt ikke at gøre det i programmet, da vi mener det ikke betyder så meget. Dette problem kommer først rigtigt frem, hvis man tegner billeder med tekst på. Her vil teksten altså blive spejlvendt i både den ene og den anden retning.

Derudover overvejede vi metoder der kunne tegne forskellige nuancer af grå, som ville gøre de billeder vi tegnede mere detaljerede. Dette kunne gøres ved forskellige grader af spidsning af blyanten.

Derudover kan det også gøres ved at tegne ovenpå billedet flere gange, så man får mørkere grå de ønskede steder. Vi valgte ikke at lave dette, da det ville blive meget komplekst og desuden ikke var et krav.

Vi har også tænkt en del på, hvordan vi kunne udregne længden af blyanten. Problemet opstår efter der er blevet tegnet et stykke tid. Det er tid til at spidse blyanten, hvorefter blyanten bliver kortere. Dette vil påvirke længden blyanten skal sænkes ned mod papiret, men også længden der skal køres ned for at spidse. Vi har forsøgt at udregne forholdet mellem længden af blyanten og antallet af 'steps' der skal køres ned til papiret. Dette er vigtigt i forhold til, hvor lang der skal køres ned for både at spidse og tegne.

Til sidst er der nogle småting, som kunne optimeres i programmet. Vi ville gerne have nogle flere elementer i interfacet. Her skulle man gerne kunne ændre forskellige indstillinger og kunne se forskellige variables værdier. Derudover resetter vi de tre akser på samme tid. Da vi gerne vil undgå, at

blyanten kolliderer med blyantspidseren eller tegner streger på vej op i startpositionen, ville vi gerne starte med at køre z-aksen helt i nul-position.

TCP-programmet kunne forberedes ved at modtage flere pakker. Vi har dog ikke haft mulighed for dette på grund af manglende viden og information omkring TCP-biblioteket.

Resultater

Ud fra de krav der er stillet til projektet har vi med de koder der er blevet lavet fået robotten til at tegne et billede der først bliver lagt ind i java og bliver lavet om til en sting som PLC'en så modtager og tegner ud fra. Mens den tegner, vil den med mellemrum spidse blyanten. (se eventuelt timelaps hvor robotten tegner under bilag) Skaleringen i java virker afhængig af billedet. Vi kan kommunikere mellem PLC'en og en pc via TCP.

Tids- og arbejdsplaner

I dette projekt har vi arbejdet sammen i et hold på syv. Dette gør at vi kan dele projektet på i forskellige dele. Vi har delt gruppen på i 2-3 mands hold, og sørget for, at alle har arbejdet med noget de ikke var alt for gode til. Dette er med til at give noget ekstra indlæring på de forskellige emner. I tabellen nedenfor, kan arbejdsfordelingen ses.

Arbejdsopgave	Ansvarlig
Indledning/Problemformulering	Anders Elian og Quan Nguyen
Elektronik	Anders Elian, Quan Nguyen og Peter Nielsen
Struktureret tekst og beskrivelse	Gustav Nobel og Peter Nielsen
Java-program og beskrivelse	Mads Østergaard, Caroline Kragh og Mads Benjamin Nielsen
Opsætning af rapport	Quan Nguyen
Finpudsning og korrekturlæsning	Gustav Nobel, Anders Elian, Quan Nguyen, Mads Østergaard, Mads Benjamin Nielsen, Caroline Kragh og Peter Nielsen

Når man arbejder i grupper, er det vigtigt, at holde styr på, hvad der bliver lavet på forskellige tidspunkter. Dette har vi løst bl.a. ved hjælp af en tidsplan ugentlige møder og kommunikation via internettet. Derudover er det også vigtigt at alle i gruppen ved hvad der bliver lavet og, at vi ved hvor langt man er nået i projektet.

For bedst mulig planlægning har vi lavet en tidsplan, som vist nedenfor.

Task Name	Start	End	Duration (days)
Projektbeskrivelse	13-10-2017	06-11-2017	24
Tidsplan	30-10-2017	31-10-2017	1
Java	23-10-2017	15-11-2017	23
Structured text	30-10-2017	15-11-2017	16
Elektronik	30-10-2017	08-11-2017	9
Test af software	15-11-2017	17-11-2017	2
Rettelser af software	17-11-2017	24-11-2017	7
Rapport	30-10-2017	08-12-2017	39
Beskrivelse af elektronik	08-11-2017	17-11-2017	9
Beskrivelse af software	15-11-2017	27-11-2017	12
Indledning	30-11-2017	08-12-2017	8
Diskussion	30-11-2017	08-12-2017	8
Konklusion	30-11-2017	08-12-2017	8
Bilag	30-11-2017	08-12-2017	8
Rettelse af rapport	08-12-2017	14-12-2017	6

Konklusion

Til sidst i forløbet, er vi kommet frem til, at vores løsning virker ud fra de fleste af vores krav. Der vil altid være noget man kunne gøre bedre og mere effektivt, og hvis vi havde haft tiden, ville vi selvfølgelig fortsætte med forbedringer.

Projektet har lært os en masse, både indenfor programmering, men også sammenhold og samarbejde. Derudover har vi haft diskussioner om forskellige løsningsforslag, som selvfølgelig er løst på bedst mulig måde.

Vi er rigtigt godt tilfredse med vores resultater, og måden vi har løst opgaven på.

Litteraturliste

- Akizukidenshi.com. (13. 12 2017). Hentet fra ST330 Stepper-Driver:
http://akizukidenshi.com/download/ds/sainsmar/ST330_Stepper_Motor_Driver_Board_User_Manual.pdf
- B&R. (13. 12 2017). *X20CP13XX*. Hentet fra http://www.br-automation.com/downloads_br_productcatalogue/BRP4440000000000000493282/X20CP13xx-RT-ENG_V1.16.pdf
- Classroom. (12. Dec 2017). *Convert to grayscale*. Hentet fra <https://www.dyclassroom.com/image-processing-project/how-to-convert-a-color-image-into-grayscale-image-in-java>
- Motechmotor. (13. 12 2017). *Nema 23 Stepper Motor*. Hentet fra http://www.motechmotor.com/products_detail.php?id=155&cid=71&page=1
- Oracle. (12. Dec 2017). *compareTo*. Hentet fra [https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#compareTo\(java.lang.String\)](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#compareTo(java.lang.String))
- Oracle. (12. Dec 2017). *File*. Hentet fra [https://docs.oracle.com/javase/7/docs/api/java/io/File.html#File\(java.lang.String\)](https://docs.oracle.com/javase/7/docs/api/java/io/File.html#File(java.lang.String))
- Oracle. (12. Dec 2017). *Load black and white image*. Hentet fra <https://stackoverflow.com/questions/5925426/java-how-would-i-load-a-black-and-white-image-into-binary>
- Oracle. (12. Dec 2017). *LoadImageApp*. Hentet fra <https://docs.oracle.com/javase/tutorial/displayCode.html?code=https://docs.oracle.com/javase/tutorial/2d/images/examples/LoadImageApp.java>
- Oracle. (12. Dec 2017). *Read*. Hentet fra [https://docs.oracle.com/javase/7/docs/api/javax/imageio/ImageIO.html#read\(java.io.File\)](https://docs.oracle.com/javase/7/docs/api/javax/imageio/ImageIO.html#read(java.io.File))
- Oracle. (12. Dec 2017). *Reading/Loading an Image*. Hentet fra <https://docs.oracle.com/javase/tutorial/2d/images/loadimage.html>