

# *Pole Position*

## *AUR2: Gruppe 2*

Uddannelse og semester:

*Robotteknologi - 2. semester*

Dato for aflevering:

*27. Maj 2015*

Vejleder:

*Preben Hagh Strunge Holm*

Gruppe medlemmer:

*Joakim Grøn, Anders Fredensborg Rasmussen,*

*Daniel Holst Hviid, Jonas Alexander Lundberg Andersen,*

*Kristian Hansen & William Bergmann Børresen*



Teknisk Fakultet

Syddansk Universitet

# Indholdsfortegnelse

<b>1</b>	<b>Indledning</b>	<b>4</b>
1.1	Problemformulering . . . . .	4
1.2	Kravspecifikation . . . . .	4
1.3	Projektafgrænsning . . . . .	5
1.4	Tidsplan . . . . .	6
<b>2</b>	<b>Hardware</b>	<b>7</b>
2.1	Accellerometer . . . . .	7
2.1.1	Logic level shifter . . . . .	7
2.2	Lap sensor . . . . .	8
2.3	Tachometer . . . . .	9
2.4	Bremse . . . . .	10
2.5	Elektromagnet . . . . .	11
2.6	Print . . . . .	12
<b>3</b>	<b>Software</b>	<b>14</b>
3.1	Generel struktur . . . . .	14
3.1.1	Retningslinjer . . . . .	14
3.1.2	Eksempler på udførsel . . . . .	15
3.2	Underbyggende funktionaliteter . . . . .	16
3.2.1	Time . . . . .	16
3.2.2	Delay . . . . .	16
3.2.3	I2C . . . . .	17
3.2.4	USART . . . . .	17
3.3	Kommunikations Protokol . . . . .	18
3.3.1	Beskrivelse af telegram . . . . .	18
3.3.2	Implementering . . . . .	20
3.4	Sensorer . . . . .	21
3.4.1	Tachometer . . . . .	21
3.4.2	Lap sensor . . . . .	23
3.5	Hastighedskontrol . . . . .	25
3.5.1	PWM . . . . .	25
3.5.2	Bremse . . . . .	26
3.6	GUI . . . . .	27
<b>4</b>	<b>AI</b>	<b>28</b>
4.1	Indledning . . . . .	28
4.2	Runder . . . . .	28
4.2.1	Preround . . . . .	29
4.2.2	Mapping round . . . . .	29
4.2.3	Run time . . . . .	30
4.3	Funktioner . . . . .	30
4.3.1	Hall-interrupt funktion . . . . .	30
4.3.2	Hall-interrupt ved Preround . . . . .	30
4.3.3	Hall-interrupt ved Mapping round . . . . .	32

4.3.4	Hall-interrupt ved Run time . . . . .	32
4.4	Lap-interrupt Funktion . . . . .	32
4.4.1	Lap-interrupt efter Preround . . . . .	33
4.4.2	Lap-interrupt efter Mapping round . . . . .	33
4.4.3	Lap-interrupt Run time . . . . .	33
4.4.4	Skift-test funktion . . . . .	33
<b>5</b>	<b>Test/Resultater</b>	<b>36</b>
5.1	Bremse-test . . . . .	36
5.2	Databehandling vha. bluetooth . . . . .	37
5.3	Mapping-test . . . . .	37
<b>6</b>	<b>Diskussion</b>	<b>39</b>
6.1	Fejlkilder . . . . .	39
<b>7</b>	<b>Konklusion</b>	<b>39</b>
<b>8</b>	<b>Litteraturliste</b>	<b>40</b>
<b>9</b>	<b>Bilag</b>	<b>41</b>

# 1 Indledning

Denne rapport er udarbejdet af 6 studerende, der læser til civilingeniør i robotteknologi på SDU, det tekniske fakultet.

Det vil være en fordel hvis læseren af denne rapport har en grundlæggende kendskab til programmering og elektronik. Dette skyldes at projektet hovedsageligt går ud på at benytte elektronik og programmering til at løse opgaven.

I dette projekt er formålet at finde en metode til at opnå den hurtigste omgangstid på en vilkårlig bane. Der vil tages højde for hastighed i sving og på langside. Bilen udstyres med forskellige sensorer for at kunne kortlægge banen og vide hvornår målstregen passerer. Til sidst er der foretaget en vurdering af projektet i helhed.

Projektet er lavet i en gruppe, hvor arbejdsopgaver fordeles. Hvert medlem har dog ansvar for at sætte sig ind i dele, de ikke selv har været en del af. Gruppen har haft møde en mindst gang ugentligt, hvor der blev fremlagt fremskridt, diskussion af problemer undervejs samt uddelegering af nye arbejdsopgaver. Gruppen har i forløbet haft flere møder med vejleder, for at diskutere retning og eventuelle spørgsmål er blevet afklaret. Rapporten er opbygget kronologisk og de forskellige kapitler, der har relevans for hinanden, vil ligge samlet. Således er der en rødtråd igennem rapporten.

## 1.1 Problemformulering

Vi ønsker at ombygge en scalextric bil til at kunne kører autonomt på en ukendt bane. Den skal selv formå at opmåle banen og derudfra regulere sine egne parametre, med det formål at opnå den hurtigst mulige omgangstid.

- Hvordan får vi bilen til at bremse ?
- Hvordan får vi bilen till at holden en jævn hastighed ?
- Hvordan får vi mappet banen op ?
- Hvordan får bilen til at reagere på det givne map ?

## 1.2 Kravspecifikation

Vi har fået stillet disse følgende oblikatoriske krav til projektet

- Den udleverede bil skal benyttes.
- Der skal anvendes en ATmega32 microcontroller i forbindelse med projektet.
- Kommunikationsprotokollen skal overholdes af hensyn til test (se nærmere i den efterfølgende gennemgang) – Det er dog tilladt at lave tilføjelser til protokollen. Det udleverede trådløse Bluetooth-modul skal benyttes.
- Der skal gøres et reelt forsøg på at gennemføre banen på den bedste tid – det er med andre ord ikke nok at lave et projekt, hvor bilen kører banen rundt med konstant fart.
- Kommunikation mellem PC og bil kan foregå via et terminalprogram, eller man er velkommen til egenhændigt at udvikle et PC-baseret program (C++, C#, Java, etc.) til formålet.
- Der skal anvendes en elektromagnetisk sensor og/eller aktuator i projektet. Der må ikke benyttes permanente magnetter.

### 1.3 Projektafgrænsning

Vi har afgrænset os til at programmere i assembly, udover dette har vi begrænset os til at bruge vores eget accelerometer med gyro (MPU5050). Vi vil fremstille en bil der skal kunne køre udlukkende af sig selv, ved hjælp af den kode der bliver skrevet til den. Som en del af vores projekt har vi stillet nogle krav som vi vil opfylde.

- Bilen skal kunne køre autonomt
- Bilen skal kunne forsøge at sætte den hurtigste omgangstid
- Bilen skal kunne opretholde sin funktion ved kortvarige udfald på strømforsyningen(Blackout og Brownout)
- Der skal laves et monitoreringssuite til pc der kan overvåge bilens performance
- Bilen skal kunne bremse
- Vi vil kunne se forskel på store og små sving

Vi har tænkt os lave nogle forskellige test på bilen for at kunne optimere på dens performance. En af de test vi har tænkt os at lave er se på dens bremselængde ved at lave en kvantitativ test.

De teori områder vi kommer indpå har vi tænkt os at dele op i to områder Hardware og software

## Hardware

- Accelerometer
- Lap sensor
- Tachometer
- Bremse
- elektromagnet

## Software

- Strukture af kode
- Interfacing af hardware
- Kommunikations protokol
- GUI (Graphical user interface)

## 1.4 Tidsplan

	Periode:		
		Start:	Slut:
		02-02-2015	27-05-2015
Mål:	Forventet:	Aktuel:	Milepæl:
Micotriontrolleren interface med bluetooth	23-03-2015	23-03-2015	
Sensor montering	01-04-2015	22-05-2015	
Bilen skal overholde de obligatoriske krav	20-05-2015	25-05-2015	
Projekt status	Uge 18	Uge 18	
Hardware implementering i rapport	20-05-2015	27-05-2015	
Software implementering i rapport	20-05-2015	27-05-2015	
Aflevering af Rapport	20-05-2015	27-05-2015	

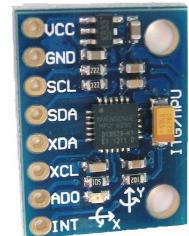
Figur 1: Her kan vi se vores tidsplan

## 2 Hardware

### 2.1 Accellerometer

Til projektet er der valgt at bruge en gyro, til at bestemme hvilken type sving bilen er i. Til projektet var der mulighed for at få udleveret et analogt og et digitalt accelramometer.

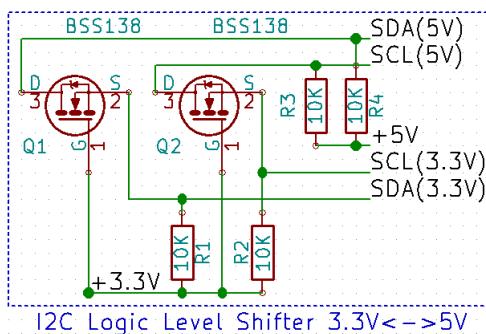
Det digitale var et LIS35DE, som kan måle  $\pm 2G$  eller  $\pm 8G$  på 3 akser, og kan lave målinger op til 400 gange i sekundet. Det analoge var et MMA1270KEG med kun en akse, der kan måle op til  $\pm 2,5G$ . Den akse det kan måle på ligger vinkelret på printet, som chippen bliver placeret på.



Figur 2: MPU-6050 Breakout Board

I gruppen havde vi en MPU-6050 tilrædighed, som har 3 akset accelramometer, som sender 16bit data ud for hver akse, det er her muligt at sætte følsomheden til:  $\pm 2G$ ,  $4G$ ,  $8G$  eller  $16G$ . Men uover at have et accelramometer så har den også en 3 akset gyro, som vi i gruppen havde en ide om, kunne gøre det nemmere at se forskel på, om det er et stort sving eller lille sving. I gruppen valgte vi at bruge MPU-6050, da det gav os mulighed for at bruge både gyro og accelramometer, så vi ikke skulle lave noget om, hvis vi ikke fik nogle brugbare værdier fra gyroen.

#### 2.1.1 Logic level shifter



Figur 3: Logic level shifter

MPU-6050 breakout boardet fungerer kun med 3,3V, og der er derfor behov for en logic level shifter, som skal omsætte 3,3V I<sup>2</sup>C til 5V I<sup>2</sup>C. I<sup>2</sup>C fungere på den måde, at linjerne

altid er trukket til 5V eller 3,3V ved hjælp af pull up modstande, og når der så bliver sendt data, så bliver linjen trukket til ground for et ditaliet 1. På figur 3 kan man se de 2 5V pull up modstændte, de 2 3,3V pull up modstændte og så de 2 MOSFET's som får 3,3V

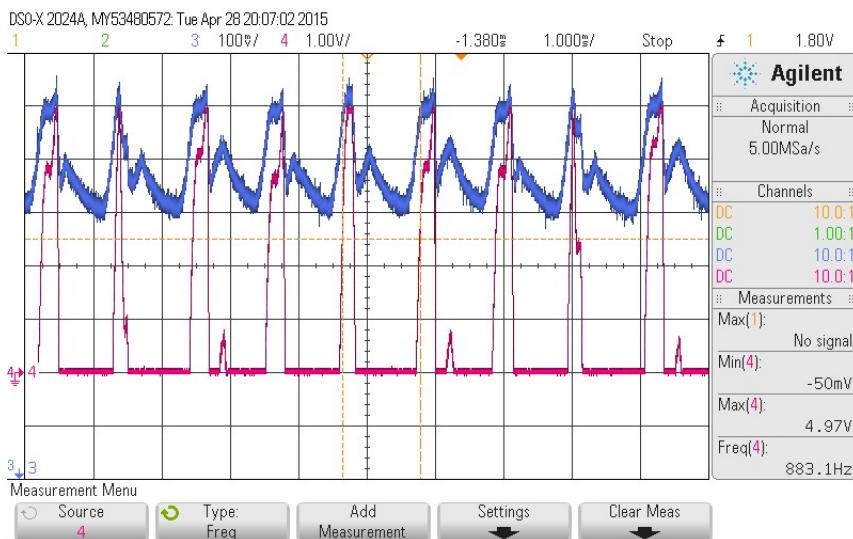
## 2.2 Lap sensor

Til at holde øje med når bilen krydsede målstregen, faldt valget på en CNY70 optisk sensor. Den består af en infrarød lysdiode og en phototransistor bygget ind i samme hus. Dioden sender hele tiden infrarødt lys ud gennem et lille vindue, og hvis der er en flade tæt på til at reflektere lyset, vil det blive opfanget af phototransistoren, der sidder ved siden af dioden bag et lysfilter. Mængden af lys der bliver reflekteret tilbage afhænger af afstanden til fladen, og typen af materiale der er på overfladen - en hvidt malet streg vil for eksempel reflektere mere lys tilbage end en sort materet overflade.

For at gøre signalet fra sensoren brugbart for microcontrolleren, ville vi bruge en komparator til at digitalisere outputtet. Oprindeligt var planen at bruge en LM311 komparator, men det viste sig at den interne komparator i microcontrolleren, lige så nemt kunne bruges til det formål, og så ville der også skæres ned på antallet af eksterne komponenter og derved mindske pladsforbruget på vores print. En anden fordel ved at bruge den interne komparator, er at det bliver muligt at bruge den interne spændingsreference på 2.56 V som input til komparatoren. Så er det bare et spørgsmål om at dimensionere outputtet fra CNY70 sensoren med en modstand, så spændingen, der kommer fra den sorte overflade på banen, ligger under 2.56V, og spændingen fra den hvide målstreg ligger over. Microcontrolleren aflæser komparatoren ved at se på outputtet fra den, og det kan sættes op, så den sender et interrupt, når der kommer en rising eller falling edge, eller når outputtet skifter tilstand. Med denne løsning bruges der kun tre eksterne komponenter - en modstand hver til henholdsvis lysdioden og phototransistoren samt CNY70 sensoren. Modstanden til lysdioden blev sat til 150 ohm, så der løber godt 30 mA i det kredsløb. For at opfylde spændingskravene til komparatoren blev der brugt en 22 kilo-ohms modstand - det gav en spænding på banen omkring 0.8 V og en spænding på målstregen på ca 4.5 V.

## 2.3 Tachometer

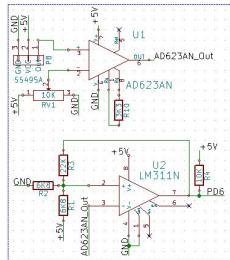
Til at måle omdrejningshastigheden på bilen, samt at opfylde kravet om en elektrofysisk sensor/aktuator, blev der valgt en analog hall sensor som monteredes på motoren. Sensoren har tre ben - et til 5 V, et til stel og et output. Outputtet ligger og svinger omkring 2.5 V og ændrer sig i positiv eller negativ retning alt afhængigt af, hvordan den magnetiske flux ændrer sig i nærheden af sensoren. Når motoren kører, vil man på hall sensorens output, så kunne måle, når polerne passerer tæt forbi sensoren. Det giver et svagt og noget støjjet signal, der skal behandles for, at microcontrolleren kan bruge det til noget. Det rå signal havde en peak-to-peak spænding på 100-200 mV, så det var ret vigtigt først at få det forstærket op. Det havde også den tilføjede bonus at skære noget af den mere højfrekvente støj fra, fordi den forstærker vi valgte, en instrumenteringsforstærker AD623, havde en knækfrekvens på godt og vel 90 kHz, under de forhold vi arbejdede under.



Figur 4: Her ses signalet fra hall sensoren før (blå) og efter (lyserød), det er blevet behandlet af forstærkeren.

Ud fra de første målinger, viste det sig at den lave del af signalet, var meget støjfyldt og svær at behandle, så vi valgte at skære den del af signalet fra med et offset på forstærkeren se figur 4. Offsettet blev styret af et potentiometer, der blev brugt som en slags justerbar referencespænding.

For at digitalisere signalet, så microcontrolleren kunne behandle det, blev der brugt en komparator - denne gang som en schmitt trigger for at formindske fejl. Det var ikke nødvendigt at designe spændingstærsklerne på schmitt triggeren så præcist. De skulle



Figur 5: Diagram over schmitt trigger kredsløbet.

bare ligge relativt langt fra hinanden, så eventuel støj ikke kunne få den til at skifte tilstand utilsigtet. Ud fra målinger med oscilloskop viste det sig, at et spænd på over 0.5 V var rigeligt til at forhindre støj i at få schmitt triggeren til at skifte tilstand.

$$V_{TL} = \frac{R_2 || R_3}{R_2 || R_3 + R_1} \cdot 5V \quad (1)$$

$$V_{TH} = \frac{R_2}{R_2 + R_1 || (R_3 + R_4)} \cdot 5V \quad (2)$$

Til at beregne spændingstærsklerne skal man finde spændingen på det ikke-inverterende ben, når komparatoren er åben, og når den er lukket. Så ender man med to ligninger med to ubekendte, hvis man fastsætter to af modstandene og spændingstærsklerne på forhånd. Modstandsværdierne blev dikteret af udvalget på komponentlageret, og ved hjælp af ligning (1) og (2) designede vi spændingstærskler på henholdsvis 2.2 V og 2.8 V.

## 2.4 Bremse

For at bremse bilen kortsluttes motorterminalerne. En kortslutning bremser motoren ved at lade et magnetfelt, der modsætter sig motorens bevægelse, blive induceret i spolerne. Hvis kortslutningen ikke er til stede, kan der ikke løbe nogen strøm og derfor heller ikke dannes noget modsatrettet magnetfelt. Selve kortslutningen bliver udført af 5 volts relæ, der bliver styret af en MOSFET forbundet til et IO-ben på microcontrolleren. Se figur 7.

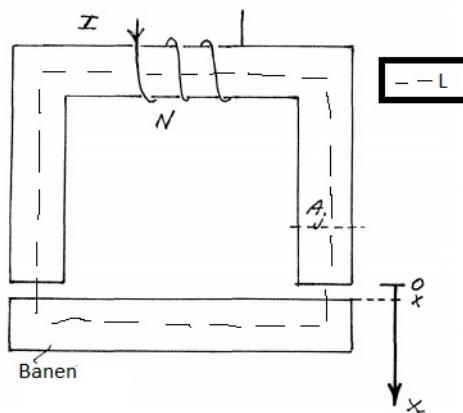
## 2.5 Elektromagnet

I dette projekt har vi valgt at se på en elektromagnet, for at se om det kunne svare sig at implementere en elektromagnet, for at kunne bremse/holde bilen på Banen.

En elektromagnet er en spole med tilhørende kerne som laver et magnetiskfelt ved at sende strøm igennem en spole. Der er flere forskellige måder, man kan påvirke dette felt på for eksempel ved at sende en kraftigere strøm igennem spolen eller have flere vindinger på spolen. Du kan også ændre den styrke ved at tilføje en kerne til elektromagneten. Det denne kerne gør at den magnetiske felter nemmer, kan trænge igennem kernen, end de kan i luft. Disse kerner bliver oftest lavet af jern. Når elektromagneten ikke er tændt vil alle de forskellige magnetiske felter være spredt ud i alle retninger, og når du så tilføjer en strøm vil de all sammen blive ensrettet, og derved bliver elektromagneten stærkere.

Disse elektromagnet bliver brugt mange forskellige steder, så som i industrien, hvor de bliver brugt til løfte meget tunge magnetiske ting, som for eksempel biler.

For at regne på en elektromagnet bliver designet valgt således, at man regner magneten, som et stykke. Hvor vi så regner på den magnetiske energi i forhold til, hvor langt de er væk for hinanden, dette kan nemmere ses, hvis det bliver illustreret i figur 6.



Figur 6: Her ses en illustration af en U-formet elektromagnet med  $N$  antal vindinger.

Ud fra amperes lov fås følgende

$$H_j \cdot L + 2 \cdot H_s \cdot x = N \cdot I$$

Hvor dette er gældende

$$H_j = \frac{B_j}{\mu_0 \cdot K_m}, H_s = \frac{B_s}{\mu_0}$$

Her er  $B_j$  det magnetiske felt for kernen, og  $B_s$  er det magnetiske felt for det tommerum mellem banen og magneten.  $H_j$  og  $H_s$  er de til hørende elektriske felter. Til sidste har vi fejlen som i det enkle materiale  $K_m$

Da vi går ud fra at der ikke er nogen spredning bliver  $B_j = B_s$  og derved får vi dette

$$B_j(x) = \frac{\mu_0 \cdot I \cdot N}{\left(\frac{L}{K_m} + 2 \cdot x\right)}$$

Ud fra dette kan vi finde den magnetiske energi og derefter finde kraften ved

$$F_{mag} = -\frac{dU_B}{dx}$$

så får vi følgende udtryk, som bruges til at beregne kraften af en U formet elektromagnet

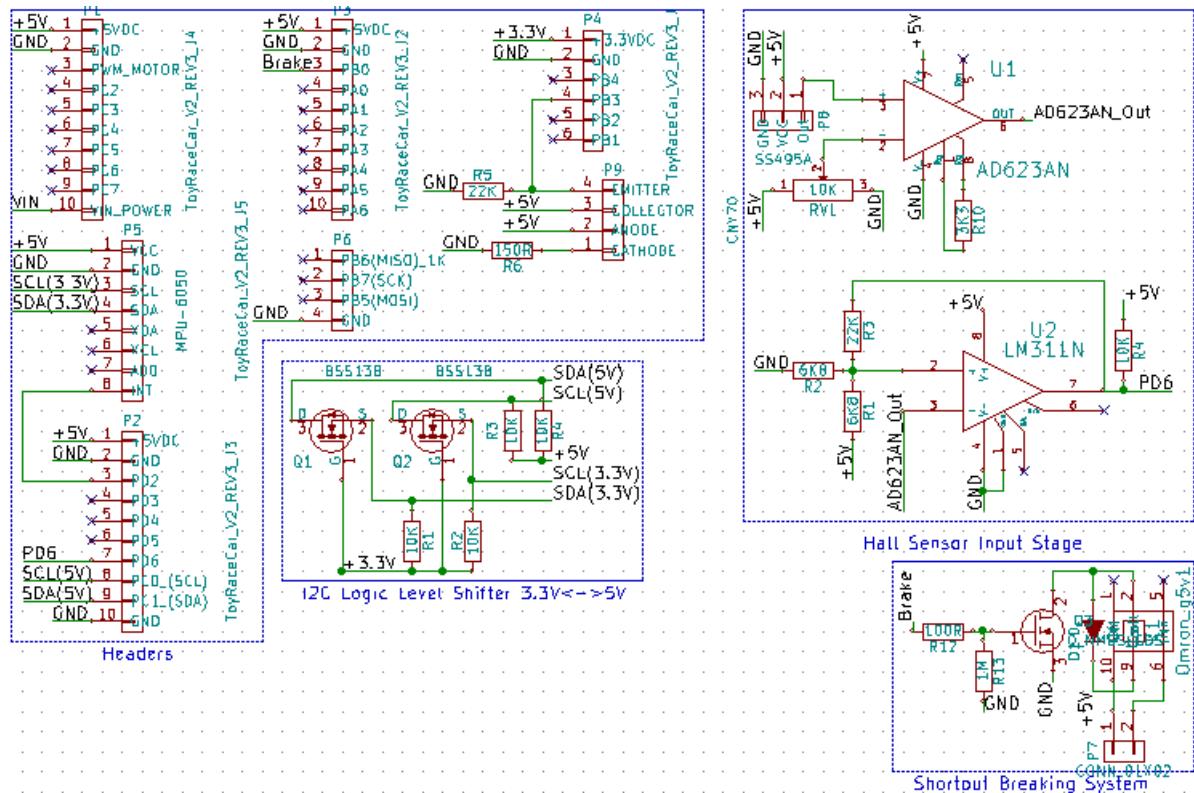
$$F_{mag} = -\frac{B_j^2}{2 \cdot \mu_0} \cdot A_j \quad (3)$$

Hvor vi går ud fra at fejlen  $\frac{L}{K_m} + 2 \cdot x = 2 \cdot x$  hvis  $K_m$  er meget stor, og x er meget lille, som i de fleste tilfælde er rigtigt, da kernen af elektromagnet vil være lavet af jern og da kraften skal være så stor som mulig, vil den også være så tæt på banen som mulig.

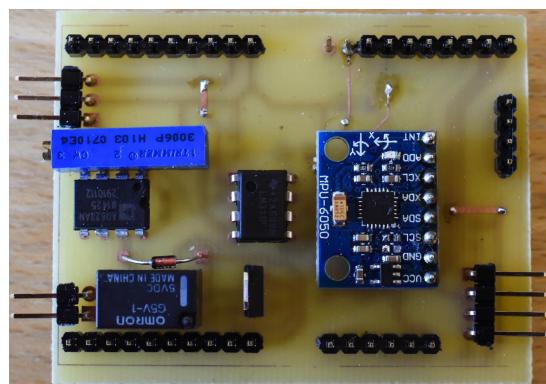
Vi valgt ikke at gå videre med at bruge en elektromagnet, da vi fandt en anden måde at bremse på som der kan læses om i sektion 2.4. Det ville også blive meget svært at få plads til elektromagneten, da den ville blive forholdsvis stor, hvis den skulle være effektiv nok. Ud fra den forudsætning at bilen skulle veje, så lidt som mulig og have det lavest mulige tyngdepunkt.

## 2.6 Print

Dette afsnit beskriver kort printet som der er blevet lavet til bilen. Vi har lavet det som et skjold der monteres ovenpå bilen i de headers som der allerede er på det print vi har fået udleveret. Figur 7 viser hele det skematiske overblik over printet. Og figur 8 viser det fremstillede print. Vi har designet det sådan at alle headers sidder så de ikke kommer i kambullage med karrosseriet på bilen.



Figur 7: Her ses hele schematic til det print der er blevet lavet til bilen. De individuelle dele er beskrevet i de relevante hardware afsnit



Figur 8: Her ses det færdige print

## 3 Software

I dette afsnit beskrives al det software som er blevet skrevet til microcontrolleren til at styrer de forskellige interne moduler samt de eksterne sensorer vi har sat på bilen. Selve styringen af hvordan bilen håndterer banen findes i sektion ??.

### 3.1 Generel struktur

I dette afsnit beskrives der, hvordan koden er struktureret, og hvilke tanker der er blevet gjort omkring opsætningen af koden inden den blev skrevet.

#### 3.1.1 Retningslinjer

Da vi gik i gang med at skrive kode, valgte vi lige at bruge lidt tid på at sætte os ned og opsætte nogle retningslinjer for, hvordan kodes skulle skrives. Dette gjorde vi for at få en konsekvent og let læselig kode. Samtidig med at debugging af eventuelle fejl skulle være så nem som mulig. Følgende kriterier blev opsat:

- Main filen skal være så ”ren” som muligt dvs. at den skal bruges som et samlingspunkt for al koden og ikke som et sted, hvor der reelt kører kode.
- Alle funktionaliteter skal skrives ind i separate asm filer, som så inkluderes i Main filen.
- Alle funktionaliteter indeholder en init macro i formatet ”funktionalitet\_Init”, som står for at initialiserer alt som funktionen skal bruge.
- Alle init macroer ”kaldes” fra ”setup.asm”, som er den sidste inkludering i main filen.
- Alle labels og definitioner skal have prefix efter funktionaliteten. Så der ikke overstår brug af de samme labels på tværs af asm filer.
- Alle interrupt vektorer defineres i en separat ”Interrupt\_Mapping.asm” fil, så det er nemt og overskueligt at holde styr på

- Alle definitioner af SRAM adresser foretages i filen ”SRAM\_Mapping.asm” så der altid er styr på, hvad der er tildelt hvilke SRAM-adresser.

Ud fra disse definitioner er al kode mere eller mindre skrevet. Hvilket har gjort det meget læseligt og nemt at debugge.

### 3.1.2 Eksempler på udførsel

Figur 9 viser et eksempel på, hvordan alle moduler er skrevet ud i individuelle asm-filer, og inkluderet ind i main filen. Dette gør, at det er nemt at slå et modul fra, hvis man skal teste noget og holde styr på, hvad der står hvor. Figur 10 viser, hvordan alle SRAM-definitioner er skrevet ind i den samme asm-fil så der ikke er nogen der kommer til at definerer de samme adresser til forskellige ting. Dette ville nemlig være en svær fejl at opdage, da compileren ikke vil smide en fejl på det.

```
;Library includes
.include "Macros.asm"
.include "Math.asm"
.include "SRAM-Mapping.asm"
.include "USART_Library.asm"
.include "WheelSpeed.asm"
.include "Delays.asm"
.include "Motor_Control.asm"
.include "I2C.asm"
.include "MPU-6050.inc"
.include "MPU-6050.asm"
.include "Time.asm"
.include "AI.asm"
.include "LapCounter.asm"
.include "Communication_Protocol.asm"
.include "Speed.asm"
.include "Setup.asm"
```

Figur 9: Viser hvordan alle moduler in individuelt inkluderet i main filen

```
;Gyro dataset from the MPU6050
.equ GYRO_XOUT_H = 0x0068
.equ GYRO_XOUT_L = 0x0069
.equ GYRO_YOUT_H = 0x006A
.equ GYRO_YOUT_L = 0x006B
.equ GYRO_ZOUT_H = 0x006C
.equ GYRO_ZOUT_L = 0x006D

;Time0 variables
.equ Timer_1ms_H = 0x006E
.equ Timer_1ms_M = 0x006F
.equ Timer_1ms_L = 0x0070

;Finish line time stamp
.equ Time_Stamp_H = 0x0071
.equ Time_Stamp_M = 0x0072
.equ Time_Stamp_L = 0x0073
```

Figur 10: Viser hvordan alle SRAM adresser er defineret i den samme fil for at kunne holde styr på, hvad der er brugt.

## 3.2 Underbyggende funktionaliteter

I dette afsnit beskrives nogle af de underbyggende funktionaliteter, der er inkorporeret i softwaren. Dette er ting, som ikke er programmerings mæssigt store nok, til at de kræver et underafsnit for sig selv, men som alligevel har dannet grundlag for nogle af de andre funktioner i koden.

### 3.2.1 Time

*Forefindes i filen "Time.asm"*

Time biblioteket blev lavet for at have en måde hvorpå bilen kan holde øje med, hvor lang tid der er gået, fra den er sat på banen. Dette bliver brugt flere stedet i koden til f.eks. at sende ud, hvor lang tid en omgang har taget, eller hvis der skal inkorporeres et delay, der ikke opholder bilen, imens den venter.

Det består af en init macro og en interrupt funktion. Init macroen sætter Timer0 op, til at kører i CTC-mode med en prescaler på 64. Dette giver Timer0 en clock på  $\frac{16.000.000Hz}{64} = 250.000Hz$  som svarer til at den tæller en op hvert  $4\mu s$ . Hvis der ikke står nogen værdi i OCR0 registret, så vil timeren også sætte et interrupt, hver gang at den tæller op. Ved at sætte OCR0 registret til en værdi kan man styre, hvor mange gange Timer0 skal tælle op, før den sætter et interrupt, og begynder forfra med at tælle. Ved at sætte OCR0 til 249 så opnås der et interrupt, hvert milisekund. Hvis man siger  $4\mu s \cdot 249$  så får man dog kun  $0.996ms$ , men timeren bruger lige en extra cycle, når der er overflow, og dette giver så i sidste ende en værdi på 250, som ganget sammen med cycliden bliver  $4\mu s \cdot 250 = 1ms$

Selve interruptet bliver håndteret af funktionen "Timer0\_Update", som sørger for at opdaterer en 24-bit værdi i SRAM, der indeholder tiden i ms, fra bilen blev sat på banen til der, hvor man læser værdien. 24-bit giver en maksimal tid på  $2^{24}ms \approx 4,6\text{timer}$  hvilket er forholdsvis lang tid, men en 16-bit værdi ville kun give en maksimal tid på  $2^{16}ms \approx 1\text{minut}$ , hvilket den meget hurtigt vil kunne opnå, hvis der køres en masse runder for at teste bilen af.

### 3.2.2 Delay

*Forefindes i filen "Delays.asm"*

Dette bibliotek indeholder en delay funktion, ”Delay\_MS”, som kan bruges til at indsætte et delay i en funktion uden at opholde bilen, mens der ventes. Det vil sige, at interrupts stadigvæk kan fungerer, som normalt imens delayet kører. Dette gøres ved at sætte et timestamp der repræsenterer tiden, hvor delayet blev aktiveret. Og så derefter loope indtil at forskellen mellem det timestamp og den nuværende tid er lig med det delay der ønskes. Imens der loopes er det stadigvæk muligt for interrupts at blive aktiveret. Denne funktion bruges bl.a. til at vente et vidst stykke tid, med at starte bilens AI fra den er blevet sat på banen

### 3.2.3 I2C

*Forefindes i filen ”I2C.asm”*

Dette bibliotek indeholder en række funktioner, der forsøger at gøre I2C/TWI-kommunikation meget mere overskueligt. Der er en init macro der sørger for at starte TWI-modulet op og sætte det op til at kører med en brugervalgt prescaler og TWBR-værdi som man kan tilføje som argument til macroen. Dette giver en mulighed for hurtigt at sætte den frekvens, modulet skal kører ved. Resten af funktionerne i filen er bare kald der gør ens kode meget mere overskuelig. I stedet for at skrive ”LDIR16, (1 << TWINT)|(1 << TWSTA)|(1 << TWEN)” og loade det ud til modulet for derefter at vente på at det sender en start kommando, så kan der blot skrives ”call I2C\_Start” og så bliver den korrekte værdi sendt til modulet, og loopet køres. Det samme gør sig gældende for alle andre I2C-kommandoer såsom Stop, Read og Write. Denne implementation, gør dog ikke brug af fejlkoder, og antager at modulet, sender kommandoerne uden fejl, hvilket det burde gøre i en single-master situation.

### 3.2.4 USART

*Forefindes i filen ”USART\_Library.asm”*

Dette bibliotek står for at implementerer alle funktioner, der har med at sende kommunikation over USART at gøre. Der er en init macro der sørger for at tænde USART-modulet og sætte dataformatet op som 8 data-bits, 1 stop-bit, no parity. Derudover kan man give UBBRH og UBBRL med som argument som baudraten så bliver sat efter. De fleste funktioner i biblioteket står primært for at sende forskellige data formater over serial i et menneskeligt læseligt format. Såsom funktionen ”Decimal\_S16”

der kan tage en 16-bit signederet værdi og skrive det ud som en serie af ASCII-karakterer. Eller funktionen ”Binary” der kan tage en 8-bit værdi og sende det ud som en serie af ASCII 0-og 1-taller. Håndteringen af modtagen data over USART foretages via et interrupt i kommunikations protokollen.

### 3.3 Kommunikations Protokol

I oplægget er der specificeret en kommunikationsprotokol, som bilen skal overholde. Protokollen fungerer ved hjælp af telegrammer. I afsnittet ”Beskrivelse af telegram” forklares der, hvad et telegram er. Og i afsnittet ”Implementering” beskrives der, hvordan protokollen kodemæssigt er implementeret i microcontrolleren.

#### 3.3.1 Beskrivelse af telegram

Protokollen er baseret på ”telegrammer” bestående af 3 bytes der henholdsvis repræsenterer en type, kommando og et parameter.

Af typer er der specificeret 3 forskellige:

- SET - 0x55 - Bruges til at sætte en værdi i bilen
- GET - 0xAA - Bruges til at hente en værdi fra bilen
- REPLY - 0xBB - Bruges til når der returneres en værdi, som svar på et get/set telegram

Det har ikke været nødvendigt at implementerer yderligere typer, så dette er også de eneste tre typer, som implementeringen gør brug af.

Af kommandoer er der fra oplæggets side sat et krav om to som bilen som minimum skal overholde, disse er som følger:

- Start - 0x10 - Bruges til at starte bilen med en procentdel af maxspændingen ud fra parametret
- Stop - 0x11 - Bruges til at stoppe bilen med

Ud over dette har vi specificeret yderligerer to set kommandoer:

- Speed\_H - 0x12 - Bruges til at sætte de øverste 8 bit af en 16-bit værdi, der essentielt bestemmer hvilken hastighed, hastighedskontrollen forsøger at holde
- Speed\_L - 0x13 - Bruges til at sætte de nederste 8 bit af en 16-bit værdi, der essentielt bestemmer hvilken hastighed, hastighedskontrollen forsøger at holde

Disse to kommandoer er essentielt en kommando, men skal sendes som to separate kommandoer, da et telegram kun kan bestå af en type, kommando og et parameter. De er blevet brugt meget til at teste hastighedskontrollen af og fastsætte max hastigheder rundt i forskellige sving.

Fra oplæggets side er der ikke specificeret nogle get kommandoer, men vi har implementeret de følgende:

- Xaccel\_H - 0xA1 - Returner de øverste 8 bit af den signerede 16-bit rå værdi fra acceleration af x-aksen på sensoren
- Xaccel\_L - 0xA2 - Returner de nederste 8 bit af den signerede 16-bit rå værdi fra acceleration af x-aksen på sensoren
- Zgyro\_H - 0xA3 - Returner de øverste 8 bit af den signerede 16-bit rå værdi af vinkelhastigheden om z-aksen på sensoren
- Zgyro\_L - 0xA4 - Returner de nederste 8 bit af den signerede 16-bit rå værdi af vinkelhastigheden om z-aksen på sensoren
- Ticks\_H - 0xA5 - Returner de øverste 8 bit af den 16-bit værdi der fortæller, hvor mange ”ticks” bilen har kørt på daværende tidspunkt
- Ticks\_L - 0xA6 - Returner de nederste 8 bit af den 16-bit værdi der fortæller, hvor mange ”ticks” bilen har kørt på daværende tidspunkt
- LapTime\_H - 0xA7 - Returner de øverste 8 bit af den 16-bit værdi der fortæller, hvor mange millisekunder banen tog at gennemfører
- LapTime\_L - 0xA8 - Returner de nederste 8 bit af den 16-bit værdi der fortæller, hvor mange millisekunder banen tog at gennemfører
- LapTicks\_H - 0xA9 - Returner de nederste 8 bit af den 16-bit værdi der fortæller, hvor mange ”ticks” der har været på en omgang af banen

- LapTicks\_L - 0xAA - Returner de nederste 8 bit af den 16-bit værdi der fortæller, hvor mange ”ticks” der har været på en omgang af banen
- Data - 0xAB - Bliver brugt til at få bilen til, at returnerer flere data-værdier, uden at skulle sende et get telegram for hver enkelt. Justeres ind alt efter hvad, man gerne vil have tilbage.

De fleste af kommandoerne består af to telegrammer da mange af værdierne, som er interessante at hente ud af bilen er i et 16-bit format.

Til sidst i telegrammet sendes der altid et parameter med, også selvom en funktion ikke gør brug af et parameter.

### 3.3.2 Implementering

Implementeringen kan deles op i to dele, en del der står for at behandle de modtagne bytes. Og en del der står for at eksekverer kommandoerne, når et helt telegram er modtaget.

**Modtagelse** *Figur 11 Beskriver modtagelsen som et flowchart.*

Når USART modulet modtager en byte, vil der blive kaldt et interrupt, der springer til plads 0x1A, hvor kommandoen ”jmp Comm\_Received” står. Dette er indgangen til modtagelsesdelen af implementeringen.

Først indlæses værdien Byte\_Num fra SRAM. Den holder styr på, hvilken af de 3 bytes der er modtaget. Den første byte, altså typen, har værdien et. Ud fra denne værdi springer programmet så til håndteringen af en modtaget type, modtaget kommando eller en eksekvering af det modtagende telegram. Håndteringen af typer og kommandoer er den sammen. Den modtagende type/kommando holdes op mod de typer/kommandoer, som microcontrolleren kender, og hvis det er en kendt type/kommando, så gemmes den i SRAM og Byte\_Num inkrementeres med en. Hvis det derimod ikke er en kendt type/kommando, så sættes Byte\_Numb til en, og derved smides telegrammet væk, hvis ikke det er af en type, som der er implementeret i microcontrolleren. Den første implementering af kommunikations protokollen indeholdt ikke dette check for om det var en godkendt type allerede i modtagelsen. Dette gjorde at hvis der på en eller anden måde fik sneget sig en byte ind imellem modtagelsen af type, kommando og parameter. Så ville microcontrolleren skulle modtage 3 bytes før fejlen bliver opdaget. Dette

resulterede i at efterfølgende telegrammer, ville være forskubbet med en byte og konsekvent ville fejle. Med den nuværende implementering vil microcontrolleren opdage fejlen med det samme, og efterfølgende telegrammer vil blive opfanget korrekt. Grunden til reimplementeringen var, at når man sendte telegrammer hurtigere end microcontrolleren kunne håndtere dem så, begyndte dens rx-buffer at blive overskrevet, og dette fik meget hurtigt ødelagt kommunikationen pga. forskubbede telegrammer. Den nuværende implementering lader ikke under dette, og vil blot ikke svare på en kommando, hvis den oplever et overflow, men svarer efterfølgende igen, når dens buffer ikke oplever et overflow længere. Dette gør protokollen meget mere solid.

#### **Eksekvering** *Figur 12 Beskriver eksekveringen som et flowchart.*

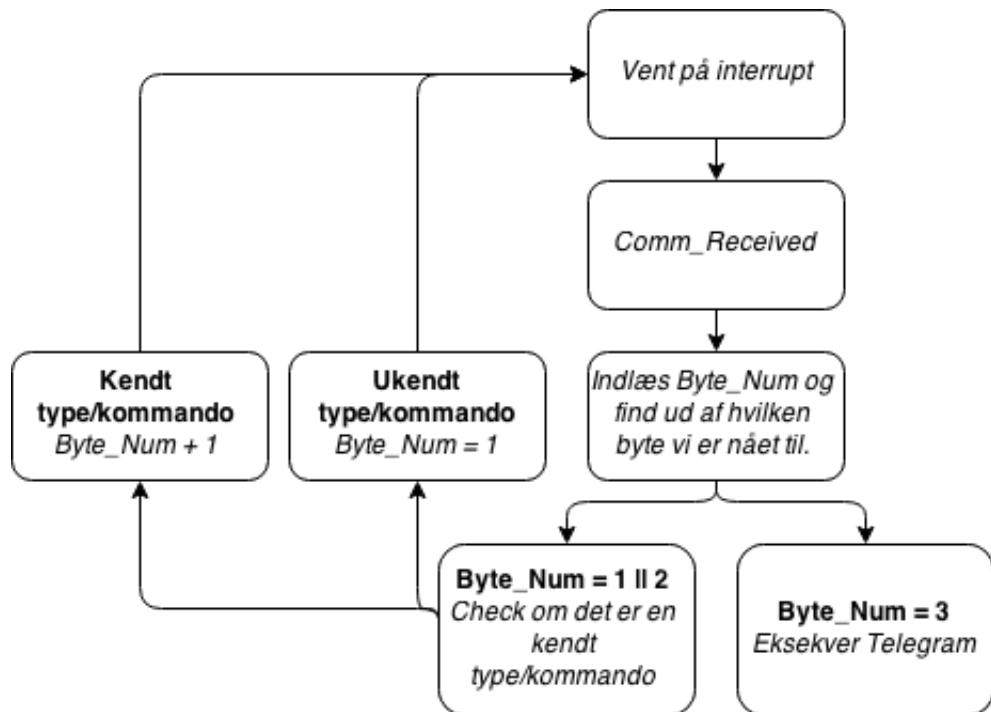
Eksekveringen bliver kørt når byte nummer 3, parametret, bliver modtaget. Her bliver typen og kommandoen igen tjekket for at finde ud af, hvilken kommando der skal eksekveres. Hvis der af en eller anden grund er en fejl på de SRAM addresser der indeholder typen og kommandoen. Så vil microcontrolleren ikke genkende det gemte telegram og selv resette igen, Så kommunikationen kan fortsætte bagefter, uden at den hænger eller forskyder telegrammerne. Efter eksekveringen af kommandoen er endt så vil Byte\_Num blive sat til en igen, og det hele vil begynde forfra.

### **3.4 Sensorer**

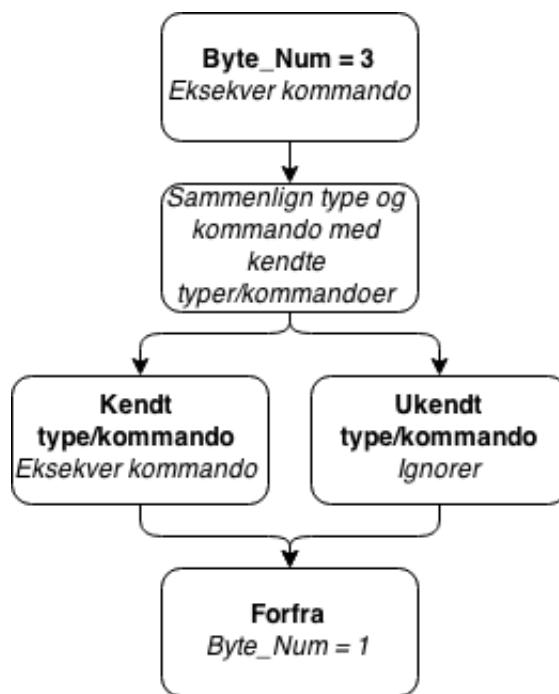
I denne sektion vil der kort blive beskrevet, hvordan vores tachometer og lap sensor interfaerer med microcontrolleren.

#### **3.4.1 Tachometer**

Til at behandle signalet fra hall sensoren, benyttes input capture funktionen til Timer 1. Input capture funktionen har sit eget interrupt og fungerer ved automatisk at læse værdien af Timer 1's timer/counter registre (TCNT1A/TCNT1B) over i to input capture registre (ICR1L/ICR1H), når der kommer et interrupt. Input capture er den sekundære funktion til PD6, og sættes op i Timer 1's kontrol registre (TCCR1A/TCCR1B). Vi valgte at sætte den op med en prescaler på 1/8 og en falling edge interrupt trigger. Det giver en opdateringsfrekvens på  $0.5 \mu s$  og en maksimal tid på 32.5 ms, inden timer/counter registrene overfloder. Det er fint nok, til at kunne rumme tiden imellem



Figur 11: Flowchart over kommunikations protokollen ved modtagelsen af en byte)



Figur 12: Flowchart over kommunikations protokollen ved eksekveringen af en byte)

flere interrupts. I interruptrutinen beregnes tiden imellem 4 interrupts for at finde ud af, hvor lang tid motoren er om at dreje en omgang (Der sidder tre poler på motoren, så 4 interrupts vil svare til en hel omgang for motoren). Denne tid kan så bruges som et udtryk for hastigheden, da de to ting er omvendt proportionale. Bilens reelle hastighed kan også beregnes, men på grund af proportionaliteten er det ligegyldigt fra microcontrolleren's synspunkt, om man bruger det ene eller det andet - man skal bare huske at en høj hastighed, vil give en lav tid og omvendt, når man skriver programmet, der skal bearbejde hastigheden. Jo færre cycles man bruger på at få noget brugbart ud i den anden ende jo bedre i næsten alle tilfælde, når der også skal laves andre ting sideløbende med input capture interruptet.

For at beregne tiden skal der holdes styr på pulserne - der skal bruges en tid fra den første puls og en tid fra den sidste puls. Det letteste er at tælle en variabel op, hver gang der modtages et interrupt således at, når variablen er 1 så læser man den første tid, og når variablen er 4 så læses den sidste tid. Derefter trækkes de to tider fra hinanden for at finde forskellen. Alle værdier og variable gemmes i SRAM-hukommelsen på tildelte addresser, indtil de skal bruges igen.

Det viste sig, at hall sensoren gav mange udslag, når bilen holdt stille, og en af polerne var lige på grænsen til at passere sensoren. Vi antager, at det er, fordi feltet er meget ustabilt i grænseområdet.

### 3.4.2 Lap sensor

For at initialisere microcontrolleren's komparator skal den sættes op i dens eget status register (ACSR) og i Special Function IO Registret (SFIOR). Man kan vælge flere forskellige inputs til komparatoren - PB2(AIN1) og PB3(AIN0) er henholdsvis ikke-inverterende og inverterende inputs som standard. Alternativt kan hele PORTA bruges til det inverterende og den interne bandgap reference på 2.56 V til det ikke-inverterende. ACME bit'en i SFIOR enabler ADC multiplexeren, så den styrer hvilket ben på PORTA, der bliver brugt. ACBG bit'en enabler bandgap referencen.

Man har også mulighed for at vælge, hvornår komparatoren skal sende et interrupt. Interruptet bliver sendt som en reaktion på, hvad der sker på komparatorens output-ben. Vi har sat den op til at sende et interrupt, når outputtet toggler tilstand, men der er også mulighed for at køre på enten falling eller rising edge. Det betyder i vores tilfælde ikke noget, hvilken metode man vælger på grund af måden koden bearbejder interruptet, men

**Table 21-1.** Analog Comparator Multiplexed Input

ACME	ADEN	MUX2:0	Analog Comparator Negative Input
0	x	xxx	AIN1
1	1	xxx	AIN1
1	0	000	ADC0
1	0	001	ADC1
1	0	010	ADC2
1	0	011	ADC3
1	0	100	ADC4
1	0	101	ADC5
1	0	110	ADC6
1	0	111	ADC7

Figur 13: I tabellen kan man se, at ACME bit'en lader multiplexeren vælge hvilket ben på PORTA, der er input til komparatorens inverterende ben. Medmindre ADC'en er slæt til - så bliver inputtet taget fra PB2(AIN1)

mere om det senere.

I interruptruten til lap sensoren beregnes omgangstiden. Hver gang målstregen krydses læses tiden fra tre SRAM addresser, hvor tiden microcontrolleren har været tændt, (i millisekunder) er gemt i 24 bits. Interruptruten sørger for at gemme et 24 bits time stamp i SRAM'en, hver gang den køres. Med den totale tid og et time stamp fra det forrige interrupt, er det bare et spørgsmål om at trække de to fra hinanden for at finde omgangstiden. For at forhindre ugyldige omgangstider, hvis der kommer mere end et interrupt på samme målstreg, kontrolleres det om omgangstiden er længere end 512 millisekunder se figur 14.

```

sub      R0, R3
sbc      R1, R4
sbc      R2, R5
; Difference between current time and last time stamp

cpi R1, 2
brlo Lap_Time_End

```

Figur 14: Her er den nuværende tid gemt i R0, R1, R2 og time stampet i R3, R4, R5. Ved at kontrollere, om den mellemste byte i R1 er under 2, kan det afgøres, om der er gået nok tid siden sidste time stamp, til at anse interruptet som gyldigt.

Det er også efter denne kontrol, at man kan ændre på alle de parametre, der kun skal ændres en gang når målstregen krydses. Inden man forlader interruptruten, er det vigtigt at resette comparator interrupt flaget i ACSR. Hvis flaget er sat, når det globale interrupt bliver enablet efter rutinen, anser microcontrolleren det som om, den har modtaget et nyt interrupt, og så vil den køre hele rutinen igen.

### 3.5 Hastighedskontrol

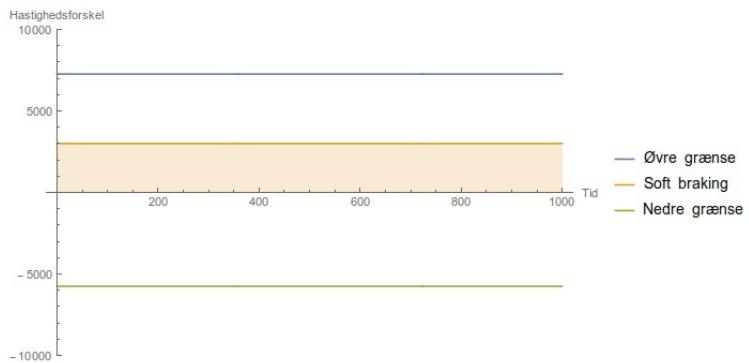
For at designe en hastighedskontrol, der selv regulerer PWM og bremser alt efter hvor langt, den målte hastighed er fra den ønskede, valgte vi at designe en progressiv kontrol, hvorved jo større forskellen imellem den ønskede og målte hastighed er, jo kraftigere vil microcontrolleren forsøge at justere. Tanken er at bruge den forskel som et direkte input til OCR2 eller bremsen, således at jo tættere bilen kommer på den ønskede hastighed jo lavere bliver værdien i OCR2, og lige sådan omvendt. Det medfører at det er umuligt for bilen, at stabilisere sig selv på den ønskede hastighed, men det gør ikke det store, da den ønskede hastighed er i forvejen en arbitrer værdi, som man skal eksperimentere sig lidt frem til. Det er både fordi, det ikke kan vides på forhånd, hvilke hastigheder der er ideelle i alle situationer, men også fordi den værdi microcontrolleren justerer efter, ikke er en rigtig hastighed, men tiden pr. motoromdrehning. Hele forløbet kan følges på flow chartet på figur 16

Hastighedskontrollen er implementeret som en call-funktion, der bliver kaldt hver gang, hastigheden bliver opdateret i input capture interruptrutinen. Det har den ulempe, at hvis bilen går i stå, af en eller anden grund, så der ikke kommer flere interrupts, så kan den ikke starte igen af sig selv, hvis værdien i OCR2 er 0.

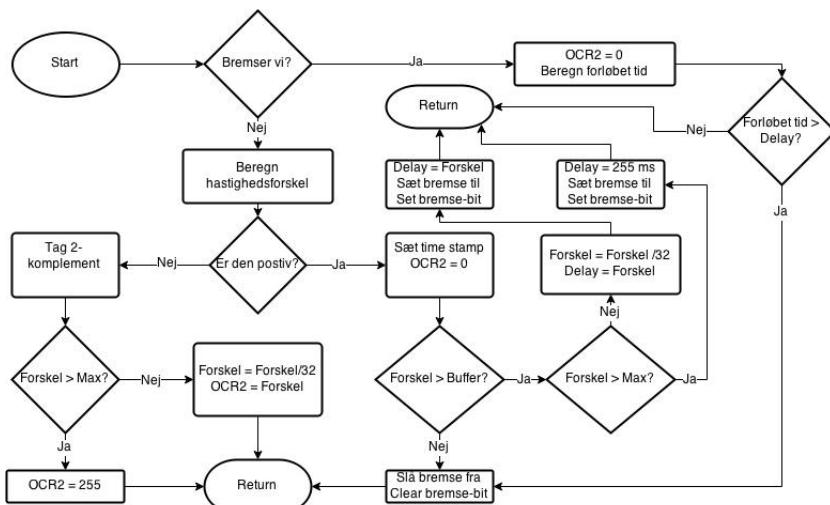
#### 3.5.1 PWM

Det første der sker er at den ønskede, og den målte hastighed læses ind fra SRAM'en, og derefter findes forskellen mellem dem. For at afgøre om bilen kører for hurtigt eller for langsomt, kontrolleres fortegnet på den udregnede forskel - det kan gøres nemt ved at kigge på N-flaget i statusregistret. Hvis bilen kører for langsomt er forskellen negativ, og for at kunne arbejde med det bruges tallets to-komplement. Hvis forskellen er større end en på forhånd bestemt størrelse, så sættes OCR2 til 255 så duty cycle er på 100 procent. Hvis den er under læses forskellen direkte ind i OCR2. Eller det ville vi gøre, hvis hastigheden var en 8-bits værdi - vi gemmer hastigheden som en 16 bits størrelse, så derfor er vi nødt til at dividere tallet ned, så det passer i et 8 bits register. Så længe man bibeholder proportionaliteten, gør det ikke noget, at tallet bliver divideret ned. For at spare på cycles i en interruptroutine, der bliver kaldt forholdsvis ofte, divideres der kun med potenser af 2 - det kan laves nemmere ved at bruge logic shifts eller rotations (det svarer til at flytte kommaet, når man ganger eller dividerer med 10 i base-10 talstemsler). Ved at dividere tallet ned giver det også muligheden for nemt at sætte et

maksimum på værdien, der kan læses ind i OCR2, når forskellen bliver brugt direkte. Hvis man eksempelvis sætter max-forskellen til 5760, så kan der højest læses 180 ind i OCR2, inden microcontrolleren regulerer med fuld kraft, idet  $180 \cdot 32 = 5760$ . Jo større tal man dividerer med, jo blødere reguleres motorkraften og jo større skal forskellen også være, for at der gives fuld kraft på motoren - vi fandt en udemærket balance med skaleringen på 32.



Figur 15: Ned til den nedre grænse bruge forskellen direkte i OCR2. I soft braking zonen læses 0 ind i OCR2 - derfra og op til den øvre grænse bruges bremsen i en tid bestemt af hastighedsforskellen. Hvis forskellen er større end de to grænsler, bremses der eller gasses op for fuld kraft.



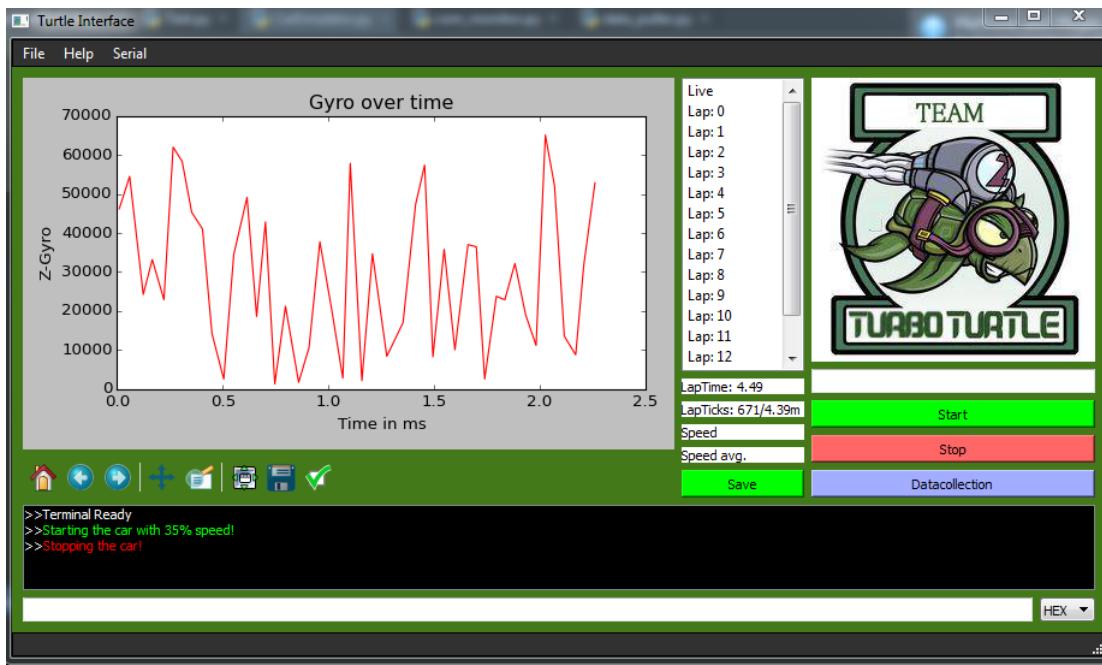
Figur 16: Her ses flow chartet til hele hastighedskontrollen. Med buffer menes grænsen imellem soft braking zonen og området hvor forskellen bliver brugt direkte som delay.

### 3.5.2 Bremse

Bremsen fungerer efter samme princip, men den regulerer bare bremsen med en tid i millisekunder. Når forskellen bliver positiv, læses der som standard 0 ind i OCR2 så duty

cycle også bliver 0 - der sættes også en bit på en SRAM-addresse, som fortæller, at bilen er igang med at bremse. Helt i starten af hastighedskontrolen checkes denne bit, og hvis den er sat, springer program counteren til en anden funktion, der holder øje med, hvor lang tid der har været bremset i. Funktionen fungerer meget på samme måde som, den der holder øje med omgangstider - der sættes et time stamp, lige når bremsen bliver sat til, og så holder microcontrolleren øje med om der forløbet nok tid, til at den kan slå bremsen fra igen. Tiden bremsen er sat til i, bliver på samme måde som med PWM, når bilen kører for langsomt, ved at bruge hastighedsforskellen som direkte parameter. Den maksimale tid der bremses i med er 255 ms - det viste sig under tests at en bremsetid på mere end 255 ms, ikke har ret meget effekt. Samtidig kan man genbruge 8 bits formatet fra tidligere. Bilen finder selv ud af om der er brug for at bremse mere, efter de 255 ms er gået. Der er indbygget en lille buffer, så bremsen ikke bliver brugt, lige så snart forskellen bliver positiv - op til en vis grænse er der en slags soft braking zone se figur 15, hvor OCR2 bare er sat til 0.

### 3.6 GUI



Figur 17: Her ses det GUI vi har fremstillet i Python til at styre bilen og lave dataopsamling. Der er mulighed for at starte og stoppe bilen samt indsamle komplette dataset der kan gemmes. Programmet kan beregne gennemsnitshastighed samt øjeblikshastighed af bilen. Der er også en terminal med indbygget farvekodning for kendte kommandoer

I forbindelse med udviklingen af bilen har vi også udviklet et GUI. Dette har givet os mulighed for at monitorere bilen i real-time. Ved opstart kommer der en dialog der giver mulighed for at vælge com-port og oprette forbindelse til bilen. Derefter kan bilen startes enten via knapper eller via terminal direkte med kommandoer. Når dataopsamling er slæt til vil grafen liveopdaterer med de data vi har sat den til at vise. Og for hver gang bilen passere målstregen vil data blive gemt i datasæt som man kan vælge at kigge på senere. Her er der mulighed for at zoome ind og gemme data hvis man har lyst. Selve GUI'et viser altid den nyeste laptid og beregnet gennemsnitshastighed for den lap, samt viser bilens øjeblikshastighed.

## 4 AI

### 4.1 Indledning

Med de sensorer vi benytter i det her projekt, er det besluttet at den databehandling, som microcontrolleren laver fungerer bedst som interrupt funktioner. På denne måde benytter vi, at data bliver opdateret i interrupt funktioner, og derfor bliver behandlet med det samme.

Det vigtigste interrupt er Hall-interruptet, se figur 18. Denne sker hver gang, hjulet er kørt en tolvtedel rundt, og giver derfor en afstand. Da der måles hvor lang tid, der går mellem disse, to fås derfor også en periode, som så benyttes som en psuedo hastighed. Det er i dette interrupt, at det meste af koden ligger.

Det andet interrupt er Lap-interruptet, se figur 19. Denne skifter mellem bane omgangene, og laver de beregninger, der skal, ske når vi går fra mapping runden til den første runde, samt resetter det map vi laver.

Koden laver dermed intet i tidsrummet mellem interrupts, hvilket gør det muligt at lave Bluetooth kommunikation.

### 4.2 Runder

Da bilen først skal mappe banen op, og derefter køre efter dette map, er der tre runde-tilstande:

- Preround: Bilen er sat ned på banen, og venter på at ramme målstregen.
- Mapping: Bilen er ved at sætte et kort, eller et “map” op af banen.
- Run time: Bilen er ved at køre efter det målte map.

#### 4.2.1 Preround

I preround kører bilen blot med jævn hastighed, indtil den rammer målstregen. Der er ikke nogen grund til, at den skulle gøre noget, før dette sker, og det eneste relevante interrupt er lap sensoren, som i dette tilfælde blot tæller lap counteren op til 1.

#### 4.2.2 Mapping round

I mapping round laver bilen et map af banen. For at mappe banen op, og benytte dette map til at optimerer bilens omgangstid, benyttes to sensorer. Den ene giver vinkelhastigheden rundt om Z-aksen, hvilket benyttes til at bestemme, om bilen befinder sig på et lige stykke bane, eller i et sving. Den anden er en hall sensor, som laver et interrupt, når den måler en vis værdi, hvilket sker tolv gange på en hjulomdriftning, hvilket giver en afstand samt en “hastighed”.

Når der kommer et Hall-interrupt i mapping serien, starter vi med at se hvilken banetype, gyroen siger, at vi befinner os på, se figur figur 20. Vi sammenligner så denne med den banetype, vi befandt os på i det forrige Hall-interrupt. Hvis disse matcher, eller hvis denne kun viser en forskel på et lille eller et stort sving, så inkrementerer vi længden af det nuværende banestykke med én. Hvis ikke, så gemmer vi længden og typen af banestykket i rammene, skifter banestykke, og sætter længden til 1. På denne måde får vi så længden og typen af alle banestykker i rækkefølge, hvilket nemt kan bruges til at følge banen i de øvrige runder.

Derefter benytter vi hastighedskontrollen, som bliver beskrevet i afsnit 3.5.

Når mapping runden er færdig skal Lap-interruptet gemme de nuværende værdier af afstanden og banetypen i rammene.

#### 4.2.3 Run time

Når bilen har lavet et map af banen, skal den kører efter det. Dette gøres ved i starten af runden, med Lap-interruptet, at indlæse bane type og længde fra toppen af listen. Når der sker et Hall-interrupt, dekrementerer vi den nuværende del af bane længen. Når denne går i nul indlæses de næste emner fra listen, banetype og længde. På denne måde ved vi altid, hvor på banen vi befinder os.

Ved så at branche ud alt efter vores nuværende banetype kan vi så sætte den ønskede "hastighed", som så gives videre til hastigheds-kontrol funktionen.

Der er to special-tilfælde. På vej ud af et sving sættes "hastigheden" til det samme som på et lige banestykke, og på vej ind i et sving bremses der en fast afstand fra et sving, bestemt fra hvor lang tid der højst skal bremses fra fuld hastighed til den hastighed vi kører med i et sving.

### 4.3 Funktioner

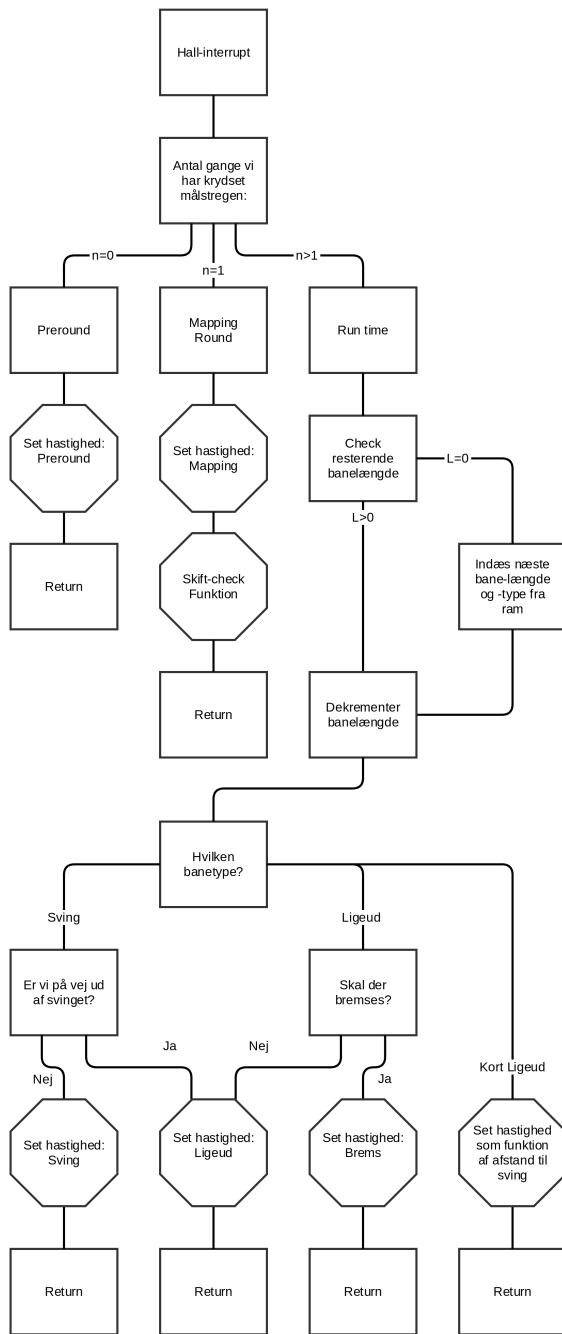
Dette afsnit beskriver de forskellige interrupt funktioner som AI'en har. Disse er implementeret som funktionskald inde i de funktioner der står får at håndterer de forskellige interrupts. Og kan derved nemt slås fra hvis noget skal testes uden AI.

#### 4.3.1 Hall-interrupt funktion

Hall-interrupt funktionen er delt op i tre dele: Preround, Mapping round og Run time. I alle delene benytter vi Hastigheds funktionen, som sætter hastigheden, såfremt den får den ønskede hastighed, samt den aktuelle hastighed som input.

#### 4.3.2 Hall-interrupt ved Preround

I Preround skal bilen bare holde en stabil hastighed, så her aktiverer vi blot vores Hastigheds Call-funktion.



Figur 18: Flowchart der beskriver Hall-interrupt funktionen. De ottekantede kasser er funktioner der er beskrevet i andre flowcharts.

#### 4.3.3 Hall-interrupt ved Mapping round

Når Hall sensoren laver et interrupt, ved microcontrolleren at der er kørt en tolvtedel af en hjulomdrehning. Dette betyder at vi meget præcist, i forhold til blot at bruge hele, halve, eller kvarte hjulomdrehninger, kan mappe længden af forskellige bane elementer op. Vi bestemmer, ved brug af gyroen, den vinkelhastighed vi har i øjeblikket, og sammenligner denne med de satte værdier for de forskellige banetyper. Hvis resultatet er det samme, som den gemte banetype inkrementerer vi blot banelængden. Ellers undersøger vi om skiftet er mellem et lille og et stort sving. Da vinkelhastigheden er større for et lille sving, vil bilen altid tro, at der er tale om et stort sving på vej ind og ud af et sving. Derfor gemmer vi ikke vejlængden, hvis der er tale om et skift mellem et stort og lille sving, men ændrer blot typen til at være et stort sving. Hvis skiftet er mellem et stort sving og et lige banestykke, gemmer vi det forrige banestykke som længde og type i rammene, og sætter længden på det nye stykke til at være 1. Se figur 20 for visuel repræsentation.

#### 4.3.4 Hall-interrupt ved Run time

I Main round benytter vi vores map til at bestemme, hvilken hastighed bilen skal køre med. Ved først at bestemme om mappet siger at bilen er på et lige banestykke, et lille sving eller et stort sving, kan vi se hvilken hastighed, bilen burde køre med. Vi ser så, om vi er på vej ind eller ud af et sving. Hvis vi er på vej ind i et sving bremses der. Hvis vi er på vej ud af et sving sættes "hastigheden" til at være det samme som på et lige banestykke.

Når vi kører ind i et lige stykke, checker vi om der er tale om et stykke, hvor der er tid til at gasse nok op, til at der skal bremses ned. Hvis der ikke er, sætter vi en hastighed, der afhænger af afstanden til det næste sving.

Derefter sendes der en tilsvarende ønsket hastighed til Hastigheds Call funktionen.

### 4.4 Lap-interrupt Funktion

Lap-interruptet inkrementerer rundetælleren, som vi bruger, til at se om vi er i Preround, Mapping round, den første Run time, eller de resterende.

#### **4.4.1 Lap-interrupt efter Preround**

Her sker intet andet, end at vi går videre til Mapping round.

#### **4.4.2 Lap-interrupt efter Mapping round**

Her går vi fra mapping round til Main round. Når dette sker sammenligner vi det første og det sidste banestykke. Hvis de begge er et sving, eller de begge er et lige stykke, lægges deres længder sammen før det sidste stykke gemmes. Ellers gemmes det sidste stykke først, hvorefter det første stykke gemmes efter det. På denne måde sikre vi os, at bilen ikke løber tør for map, hvis længden driver, samt at den ved om banestykket fortsætter forbi målstregen.

Derefter nulstilles mappet, så vi kan læse fra det i den rigtige rækkefølge.

#### **4.4.3 Lap-interrupt Run time**

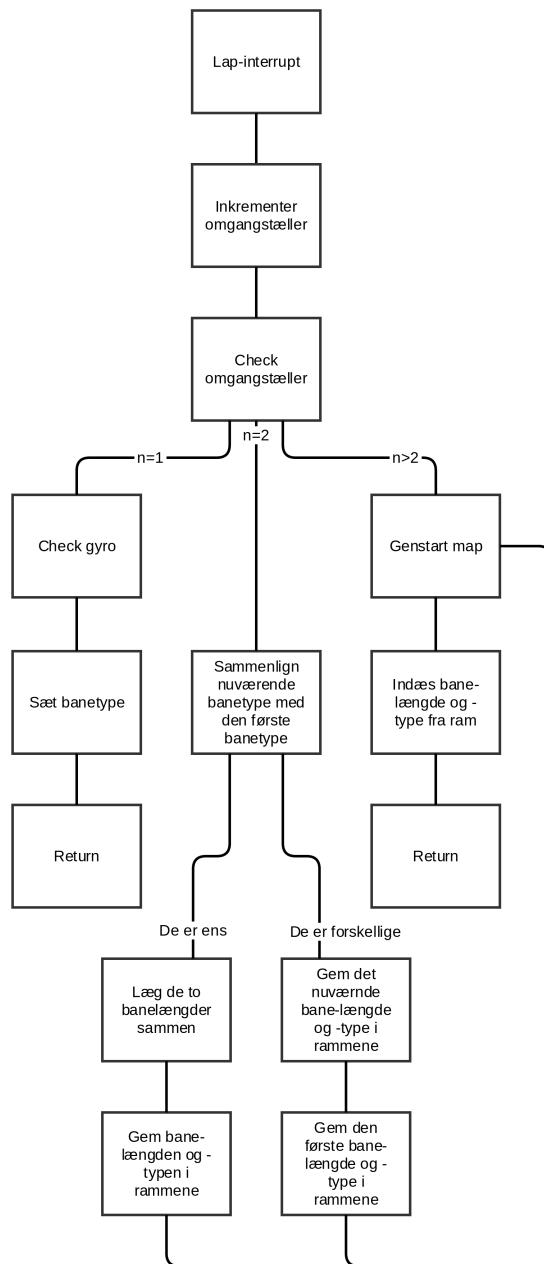
I alle resterende Lap-interrupts nulstilles mappet blot.

#### **4.4.4 Skift-test funktion**

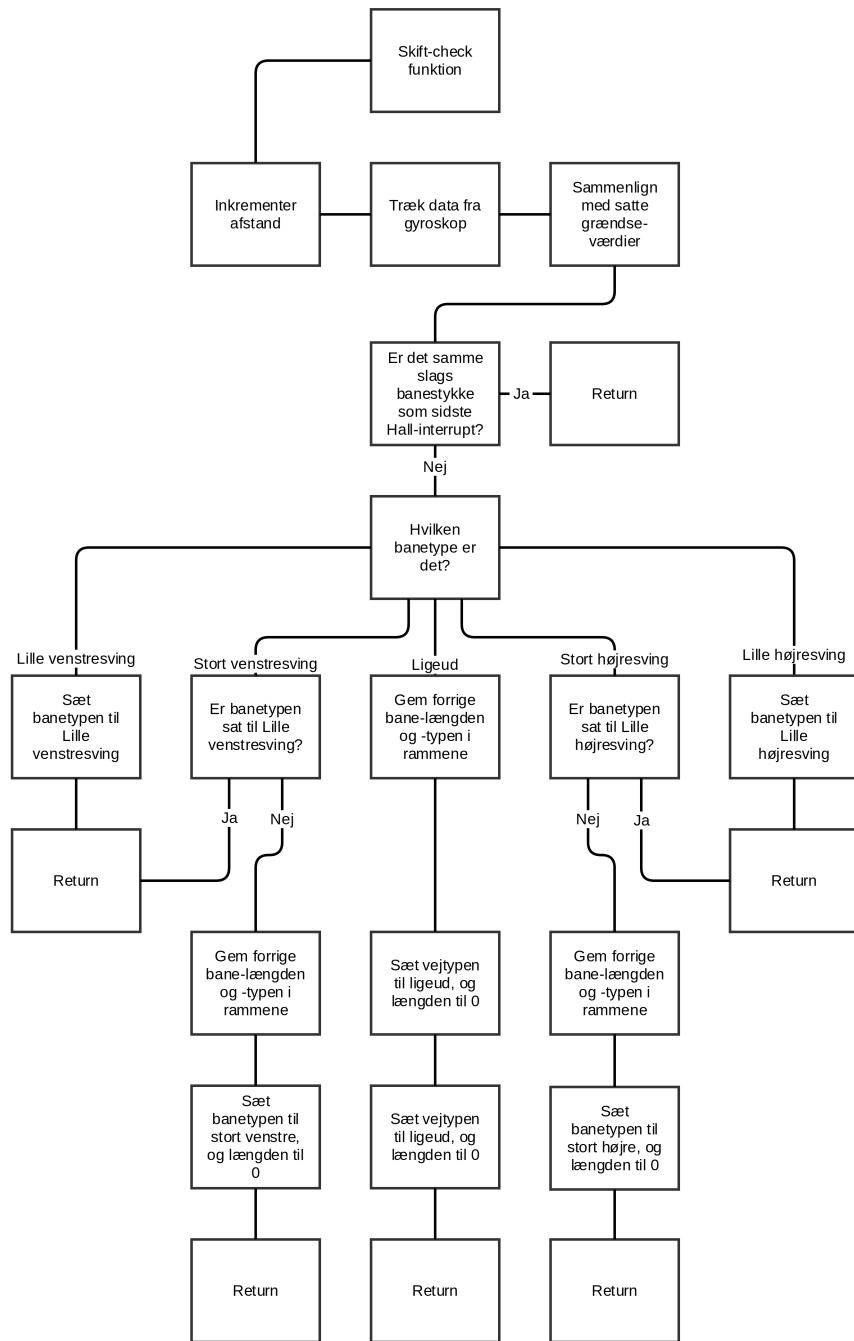
Når vi kører mapping runden, checker vi ved hvert Hall-interrupt, om vinkelhastigheden fra gyroen antyder, at der er sket et skift mellem de forskellige banetyper. Undtagelsen er, hvis der er sket et skift mellem store og små sving, da usikkerheden på gyroen er relativt stor, at det kan ske at den kan se et stort sving som et lille, eller omvendt. Derfor antager koden at alle sving der har haft blot én gyro-værdi der antyder et lille sving, er et lille sving, da det er bedre at antage at et stort sving er et lille sving end omvendt.

Hvis der er sket et skift i vinkelhastigheden, betyder det at vi har skiftet banetype. Derfor gemmes længden og typen af det gamle banestykke i RAM'ene, og vi begynder at tælle det nye banestykke op. Da dette ikke vil ske når vi krydser målstregen, sker dette også når Lap-interruptet siger at mapping runden er slut.

For visuel repræsentation, se figur 20



Figur 19: Flowchart over Lap-interrupt funktionen.



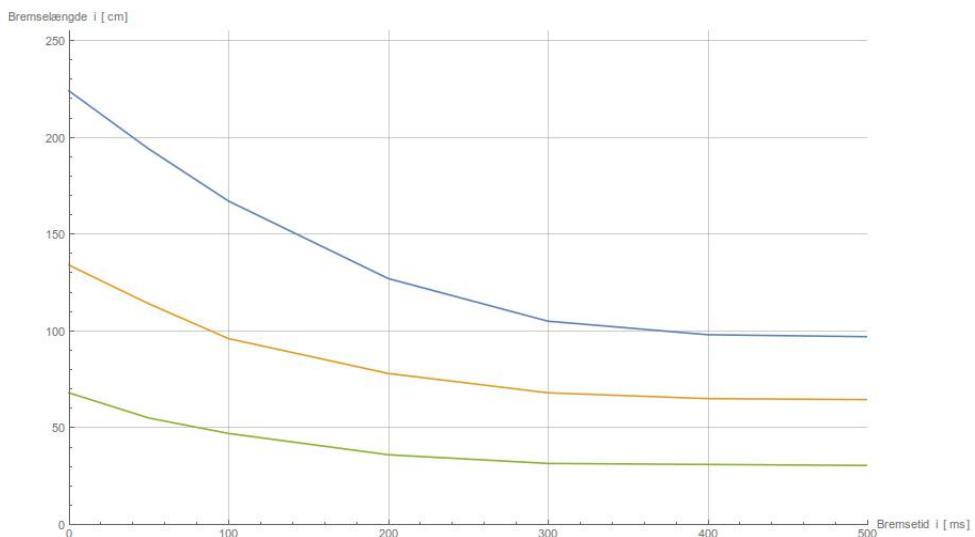
Figur 20: Flowchart over Skift-test funktionen.

## 5 Test/Resultater

I dette afsnit beskrives der nogle af de tests, vi lavede for både at debugge, men også for at indsamle data, der skulle bruges for at komme videre med programmeringen.

### 5.1 Bremse-test

For at teste effekten af bremsen, opstillede vi en lang lige bane, for at afprøve bilens bremselængde. Bremselængde skulle gerne variere alt efter, hvor lang tid motoren var kortsluttet. Vi satte bilen igang med samme hastighed, og varierede kortslutningsperioderne i intervallerne 0, 50, 100, 200, 300, 400 og 500 millisekunder. Når bilen krydsede målstregen, slukkedes motoren, og bremsen blev tilsluttet. Så lod vi bilen køre indtil den stoppede, og målte afstanden til målstregen. Hver periode blev kørt fem gange, og så beregnede vi gennemsnittet. Testen blev lavet for tre forskellige værdier i OCR2 - 255, 180 og 120 svarende til duty cycles på henholdsvis 100 %, 70,6 % og 47 % (se figur 21).

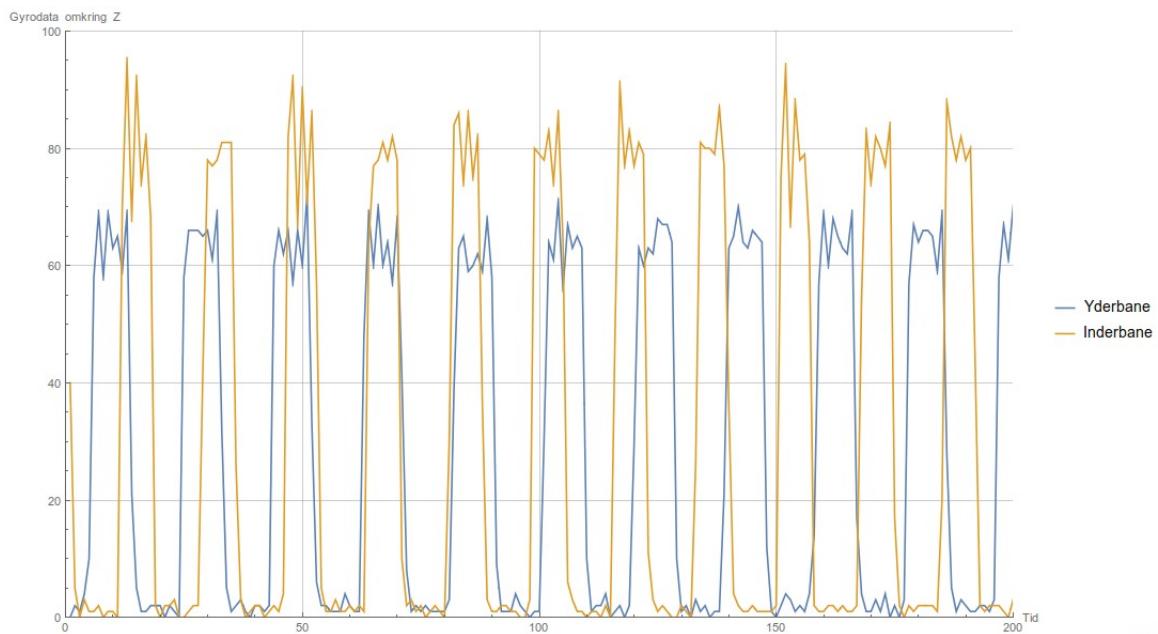


Figur 21: Her ses bremselængden i forhold til bremsetiden, med bremselængden i cm op ad y-aksen, og bremsetiden i ms ud ad x-aksen

Ud fra testen kan vi konkludere, at bremsen har størst effekt ved korte bremsetider under 300 ms.

## 5.2 Databehandling vha. bluetooth

Vi lavede også en række tests, hvor vi brugte bluetooth modulet til at sende data til et terminalprogram, så vi kunne plotte eksempelvis gyrodata i en graf (se figur 22). Ud fra en graf var det nemmere at afgøre helt præcist, hvordan dataen skulle behandles i programmet. Vi lavede også tests, hvor vi plottede OCR2 sammen med den målte ”hastighed”, for at evaluere hastighedskontrollen tidligt i udviklingen.



Figur 22: Her ses forskellen mellem vinkelaccelerationen på yder- og inderbanesving ved nogenlunde konstant hastighed

## 5.3 Mapping-test



Figur 23: Her ses vores debugging-LED'er monteret på bilen.

Til at fintune gyro-værdierne for sving og afprøve om baneelementerne blev mappet korrekt, fastmonterede vi fem LED'er, vi kunne tænde individuelt så, det var muligt at se direkte, om microcontrolleren reagerede, som vi ville have den til (se figur 23).

Hver LED fik sit eget IO-ben, og så blev de tændt alt efter hvilken værdi, gyroen sendte tilbage, således at de røde LED'er repræsenterede

yderbanesving, de gule: inderbanesving og den grønne: ligeud. På den måde kunne man nemt se, hvordan bilen tolkede banen under mappinggrunden, og om værdierne blev gemt rigtigt i SRAM-hukommelsen. Eksempelvis havde vi et problem, hvor microcontrolleren gemte det sidste baneelement på den forkerte RAM-addresse - det var nemt at se fordi, den lige i det tilfælde, konsekvent troede at der var et lige baneelement et sted på banen, hvor der var et sving. Derudover kunne LED'erne bruges til at teste andre funktionaliteter, som for eksempel hvornår bremserne blev slået til, hvornår der blev givet fuld gas osv.

## 6 Diskussion

### 6.1 Fejlkilder

## 7 Konklusion

I dette projekt integrerede vi en ATmega32A microcontroller i en Scalextric slot car, således den selv kunne kortlægge en vilkårlig bane, og vælge den optimale hastighed på de forskellige baneelementer. En optisk lap sensor, et 3 akset accellerometer/gyro og en hall sensor anvendt som tachometer blev implementeret som inputs til microcontrolleren, så den kunne finde ud af, hvor på banen den var og hvor hurtigt den kørte. Derudover installerede vi en elektrisk bremse, der på signal fra ATmega'en kortsluttede motorterminalerne.

Oprindeligt var planen at designe en elektromagnet til at bremse bilen, men det viste sig at være meget besværligt at implementere, da den ville blive relativt tung og bruge meget strøm for at være effektiv. En kortslutningsbremse var meget mere elegant og behagende for alle der havde æren af at overvære dens under. Det har vist sig at være svært at differentiere imellem inder- og yderbanesving, så vi har været nødsaget til at sørge for at lave en slags fail safe, så bilen aldrig tror den befinner sig i yderbanen, når den egentlig er i inderbanen. Det omvendte scenarie er ikke så stort et problem da bilen ikke falder af banen, hvis den kører efter den langsomme inderbanehastighed i yderbanen.

## 8 Litteraturliste

### Figure 2

- [www.playground.arduino.cc/uploads/Main/mpu-6050.jpg](http://www.playground.arduino.cc/uploads/Main/mpu-6050.jpg)
- Dato: 25/05/2015

### Figure 13

- [www.atmel.com/images/doc2503.pdf](http://www.atmel.com/images/doc2503.pdf)
- Dato: 13/03/2015

### Principles of Physics

- 10th Edition
- David Halliday, Robert Resnick & Jearl Walker
- ISBN-1 978-1-118-23074-9

### Intelligent Mechatronic System

- Modeling, Control and Diagnosis
- <https://books.google.dk/books?id=k81ECeMxyk8C&pg=PA404&dq=ferromagnetic+electromagnet&hl=en#v=onepage&q&f=false>
- Rochdi Merzouki, Arun Kumar Samantaray, Pushparaj Mani Pathak & Belkacem Ould Bouamama
- ISBN-978-1-4471-5 (E-bog)

### Elektrofysik (E-ANA 1)

- Supplerenede noter
- Efterår 1997
- Søren Hassing & René Skov Hansen
- Ved hørende PDF (CD)

### Electrical Engineering

- Principles and Applications
- 6th Edition
- Allan R. Hambley
- ISBN-13 978-0-273-79325-0

### Atmega32A-PU Datasheet

- [www.atmel.com/images/doc2503.pdf](http://www.atmel.com/images/doc2503.pdf)
- Dato: 13/03/2015

### **AVR 8bit Instruction set**

- [www.atmel.com/images/doc0856.pdf](http://www.atmel.com/images/doc0856.pdf)
- Dato: 13/03/2015

## **9 Bilag**