CAN-styrd regulator för kylvätskeventil till hybridbatterier

Examensarbete inom högskoleingenjörsprogrammet Elektroingenjör

ARVID ZIEMANN

Institutionen för Signaler och system
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2016

Förord

Examensarbetet har utförts som ett avslut på elektroingenjörsprogrammet 180 HP vid Chalmers tekniska högskola. Arbetet omfattar 15 HP och har genomförts som ett uppdrag av Volvo Group Trucks Technology under vårterminen 2016.

Hos Volvo GTT vill jag tacka Mikael Svensson som ställt upp som handledare samt Jan Grundberg som varit behjälplig under arbetets gång. Därutöver vill jag tacka Göran Hult och Manne Stenberg på Chalmers som ställt upp som handledare respektive examinator.

Sammanfattning

Temperaturkontroll av hybridbatterier spelar en avgörande roll för batteriernas livslängd. Hybridbatterierna är vätskekylda och celltemperaturen kan kontrolleras genom att med en trevägsventil reglera kylvätskeflödet genom batterierna. Ventilen har ingen egen intelligens utan kräver ett externt drivsteg samt en regulator för att få den önskvärda funktionen.
En ventilregulator med CAN-kommunikation har konstruerats med hjälp av mikrokontrollern Arduino. Regulatorn har dimensionerats genom stegsvarsanalyser samt en god förståelse av vilken effekt de olika regulatorparametrarna P, I och D har på ett system. Resultatet är presenterat i form av en låda där nödvändig kontaktering för spänningsmatning och CAN-kommunikation till regulatorn har installerats. Slutprodukten har testats genom att med CAN-kommunikation skicka stegsvars ändringar och sedan bevaka dem i datorprogrammet CANalyzer. Dessutom har den testats skarpt i provrum på Volvo GTT.

Abstract

Temperature control of hybrid batteries plays a decisive role for the battery life. The hybrid batteries are liquid cooled and the cell temperature is controlled by a threewayvalve, regulating the coolant flow through the batteries. The valve has no intelligence of it its own thus it requires an external driver and a regulator to obtain the desired function. A Valve regulator with CAN-communication has been constructed using the microcontroller Arduino. The regulator has been dimensioned by analyzing step responses and obtaining a good understanding of the impact the different controller parameters P, I and D has on a system. The result is presented in form of a box where necessary contacting for power supply and CAN-communication for the controller has been installed. The final product has been tested by sending various step responses using CAN-communication and then observing the expected response in CANalyzer. In addition, the controller has been sharply tested at Volvo GTT.

Innehåll

1 Inledning ... 1
1.1 Bakgrund ... 1
1.2 Syfte ... 1
1.3 Avgränsningar ... 1
1.4 Precisering av frågeställningen ... 1
1.5 Kravspecifikation ... 1
2 Teknisk bakgrund ... 2
2.1 Controller Area Network ... 2
2.2 PID regulator ... 3
2.3 Hårdvara ... 4
2.3.1 Trevägsventil ... 4
2.3.2 Arduino Uno ... 5
2.3.3 CAN-bus-sköld ... 5
2.3.4 Motor-sköld R3 ... 5
2.4 Mjukvara ... 6
2.4.1 Arduino ... 6
2.4.2 CANalyzer ... 6
3 Metod ... 7
4 Genomförande ... 8
4.1 Val av mikrokontroller ... 8
4.2 CAN-kommunikation ... 8
4.2.1 Arduino ... 8
4.2.2 CANalyzer ... 9
4.3 Ventilstyrning ... 9
4.3.1 Modifikation av motor-sköld R3 ... 9
4.3.2 Drift av ventilmotor ... 10
4.3.3 Mätningar av ventilmotorn ... 11
4.3.4 Ventilens lägesgivare ... 12
4.4 Ventilregulatorns huvuddelar ... 13
4.4.1 PI-regulator ... 13
4.4.2 Autokalibrering av ventil ... 15
4.4.3 DR (Wake-up) signal ... 16

4.4.4 Byte av CAN-ID ... 16
4.4.5 CAN-meddelande ... 16
4.4.6 Watchdog ... 17
4.4.7 Flödesschema ... 18
4.5 Spänningsmatning ... 19
4.5.1 Arduino Uno och sköldar ... 19
4.5.2 Spänningsreglering ... 19
4.6 Byggnation av låda ... 20
4.6.1 Val av låda ... 20
4.6.2 Ritningar för borrhål ... 20
4.6.3 Montering och koppling ... 20
5 Resultat ... 22
5.1.1 CAN-kommunikation ... 22
5.1.2 Regulator ... 22
5.1.3 Fungerande produkt ... 22
6 Slutsats ... 24
6.1 Diskussion ... 24
6.1.1 CAN-kommunikation ... 24
6.1.2 Regulator och dimensionering ... 24
6.2 Vidareutveckling ... 24
Referenser ... 25

BILAGA 1 – Pin konfiguration

BILAGA 2 – Kopplingsschema

BILAGA 3 – Mekaniska ritningar

BILAGA 4 – Produkt/material lista

BILAGA 5 – Källkod

1 Inledning

1.1 Bakgrund

Hybridbatterier är väldigt känsliga mot överhettning. Temperaturkontroll av batterier är därför avgörande för deras hållbarhet och prestanda. På Volvo GTT används hybridbattier med lithium-jon-teknik. Batterierna är kylvätskekylda som i hybridbussar kyls av en luftpåblåst radiator. Kylningen av batterierna bestäms av mängden kylvätska som passerar radiatorn. Den mängd kylvätska som passerar radiatorn styrs av en 3-vägs proportionalventil. Vid kylbehov skickas en viss mängd kylvätska till radiatorn, och när inget kylbehov förekommer så recirkuleras kylvätskan utan att passera radiatorn.

1.2 Syfte

Syftet med examensarbetet är att bygga ett drivsteg samt en CAN-styrd regulator för en kylvätskeventil som är tänkt att kunna användas för att kontrollera celltemperaturen på hybridbatterier.

1.3 Avgränsningar

Vilket läge ventilen skall reglera in sig till beroende på set-temperatur hos hybridbatteriet hanteras av en extern riggdator. Denna reglering ingår inte i projektet. Däremot ska ventilen kunna anta ett börvärde i form av ventilläge 0-100 %.

1.4 Precisering av frågeställningen

Ventilregulatorn ska konstrueras med hjälp av en mikrokontroller som ska kunna ta emot ett önskat ventilläge via CAN-kommunikation. Mikrokontrollern skall sedan reglera ventilöppningen beroende av vilket ventilläge som tagits emot på CAN-bus. Ventilen drivs av en dc-motor och har en återkopplad signal som indikerar ventilens position, denna signal kommer utnyttjas till regulatorn och skall dessutom skickas till användaren på CAN-bus

1.5 Kravspecifikation

De kravspecifikationer som sattes upp för ventilregulatorn är följande:

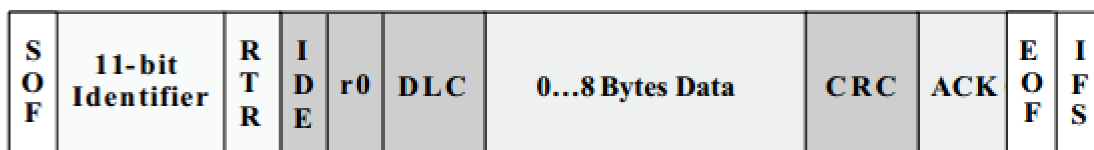
- Ventilregulatorn skall konstrueras med hjälp av en mikrokontroller.
- All kommunikation till ventilregulatorn skall ske på CAN-bus med en hastighet på 500 kbps.
- Ventilen skall kunna anta ett ventilläge från 0-100% (0 decimalers noggrannhet) som skickas med CAN-kommunikation. Ventilregulatorn skall dessutom rapportera tillbaka sitt läge på CAN-bus.
- Ventilregulator skall kunna växla mellan 4 olika CAN-adresser.

2 Teknisk bakgrund

2.1 Controller Area Network

Controller Area Network (CAN) som framställdes av Bosch är en databus avsedd för att låta flera noder eller styrenheter i fordon att skicka meddelanden till varandra. CAN kan sända med hastigheter upp till 1 Mbit/s och är vanligt förkommande i fordon där kraven på snabbhet är höga. Det kan bland annat användas för bromssystem, antisladdsystem och motorstyrning.

I ett CAN nätverk är alla noder ihopkopplade till ett och samma tvinnade kabelpar. Varje nod avgör själv om meddelandena på CAN-bussen är relevanta. Eftersom noderna i CAN nätverket kan sända samtidigt finns det ett CAN-ID i varje meddelande som kännetecknar dess prioritet. CAN-ID:et kan antingen bestå av 11 eller 29 bitar och kallas för CAN-standard respektive CAN-extended. Om två meddelanden skickas samtidigt så kommer det meddelande som har lägst värde på sitt CAN-ID ha högst prioritet [1]. Ett CAN meddelande kan skicka upp till 8 bytes med data, hur ett CAN meddelande i standard format ser ut visas i figur 1.



Figur 1. Bit strukturen av ett CAN-standard meddelande.

- **SOF** – Denna bit kallas för ”Start Of Frame” och markerar starten av ett meddelande och används för att synkronisera alla noder.
- **Identifier** – Anger vilken prioritet meddelandet skall ha. Det finns två olika bit längder på identifiern: CAN-standard (11 bitar) och CAN-extended (29 bitar). Ju lägre värde desto högre prioritet.
- **RTR – ”**Remote Transmission Request**”** Denna bit är ’1’ när information tas emot från en annan nod.
- **IDE** – ”Identifier Extension” anger om CAN-standard eller CAN-extended skall användas.
- **DLC** – **”**Data Length Code” bestämmer hur många bytes data fältet innehåller.
- **Data field** – Innehåller all data, 0 till 8 bytes.
- **CRC** – **”**Cyclic Redundancy Check” innehåller kontrollsumman av CAN meddelandet för att säkerställa att alla bitar har skickats korrekt.
- **ACK** – Alla noder som korrekt mottar ett meddelande skriver en logisk ”1” till denna bit. Detta är för att indikera att meddelandet har mottagits utan fel. Är denna bit en logisk ”0” så raderas meddelandet och skickas på nytt.
- **EOF – ”**End Of Frame” indikerar slutet av CAN-meddelandet[2].

2.2 PID regulator

PID(Proportional, integral och derivata) reglering är en vanligt förkommande metod för reglering av diverse system. Algoritmen är populär därför att den fungerar på majoriteten av alla reglersystem och är lätt att förstå och använda.

En PID regulator använder sig av ett återkopplat system där utsignalen kopplas tillbaks och summeras eller subtraheras med insignalen. I ett reglersystem är insignalen det värde som önskas erhållas på utgången, detta värde kallas ofta för börvärde. Utsignalen är det aktuella värdet som brukar kallas för ärvärde. PID regulatorn räknar kontinuerligt fram reglerfelet genom att subtrahera börvärdet med ärvärdet. Regulator försöker sedan minska detta reglerfel genom att summera och räkna ut de tre olika termerna P, I och D [3][4].

$$u(t) = K_P e(t) + K_i \int e(t)dt + K_d \frac{d}{dt} e(t) \qquad (1)$$

Där $u(t)$ är utsignalen och $e(t)$ reglerfelet. K_P, K_i och K_d är parametrarna som dimensioneras för att få en så bra reglering som möjligt.

P-del
Proportional delen är beroende av reglerfelet och parametern K_P. Denna term bestämmer systemets snabbhet men ett för högt K_P värde leder oftas till att systemet börjar oscillera. En av nackdelarna med att bara använda sig av en P-del i en regulator är att statiska reglerfel kan uppstå. Dessa statiska reglerfel uppstår när reglerfelet är så litet att produkten av reglerfelet och K_P parametern inte blir tillräckligt stort för att tvinga systemet att reglera in sig den sista biten [3][4].

I-del
Den integrerande delen summerar alla reglerfel med tiden. Detta betyder att de statiska reglerfelen som uppstår i en P-regulator kan elimineras eftersom även det minsta lilla reglerfelet kommer att ackumuleras upp tills det är tillräckligt stort för att systemet ska klara att reglera in sig den sista biten [3][4].

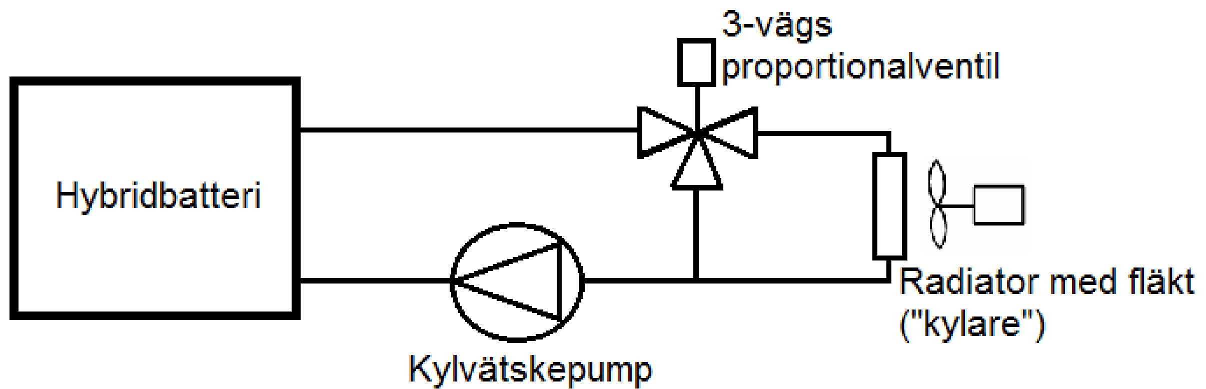
D-del
Den deriverande delen är proportionell mot derivatan av reglerfelet. D-delen används sällan man kan vara aktuell i vissa reglersystem där man vill förbättra stabiliteten [3][4].

Dimensionering
För att få fram ett bra reglersystem måste parametrarna för varje del P, I och D bestämmas noggrant. Det finns ett antal olika metoder för att bestämma dessa parametrar.
En manuell metod har använts i detta projekt som kan beskrivas med de engelska orden ”guess and check”. Enkelt beskrivet så ändrar man parametrarna till genomtänkta värden sedan observerar man hur systemet reagerar till förändringarna genom att tillföra ett stegsvar, denna procedur repeteras sedan tills ett stabilt system har erhållits [3].

2.3 Hårdvara

2.3.1 Trevägsventil

Ventilen som kommer att användas för att reglera kylvätskeflödet till hybridbatterierna är en 3-vägs proportionalventil. Vid kylbehov passerar kylvätskan en radiator och när inget kylbehov förekommer recirkuleras kylvätskan utan att passera radiatorn.



Figur 2. Förenklad skiss över kylvätskekrets till hybridbatterier.

Ventilen har ingen egen intelligens vilket betyder att ett externt drivsteg samt reglering behöver konstrueras. Ventilöppningen styrs med en dc-motor som sitter inuti ventilen, där finns också en lägesgivarpotentiometer som anger ventilläge.

Tabell 1. Trevägsventil pinout.

Ventil Pin	Beskrivning
1	GND (lägessgivarpotentiometer)
2	Utgång (lägesgivarpotentiometer)
3	+5 V (lägessgivarpotentiometer)
4	Ventil dc-motor -
6	Ventil dc-motor +

Figur 3. Trevägsventil

2.3.2 Arduino Uno

Utvecklingskortet Arduino Uno är grunden till regulatorn och kommer att programmeras för att sköta reglering av ventil, kommunikation och en del andra finesser. Uno-kortet är baserad på Atmega328 processorn och har 14 digitala in/utgångar, 6 analoga ingångar samt en 16 MHz kvartsoscillator [5].

2.3.3 CAN-bus-sköld

CAN-bus-skölden möjliggör CAN-kommunikation hos Arduinon. Den använder sig av CAN kontrollern MCP2551 och CAN tranceivern MCP2551 för att samverka med Arduinons SPI anslutningar och därmed ge Arduinon CAN-bus kapabilitet. CAN-kortet monteras ovanpå Arduino Uno-kortet och har en skruvterminal där CAN-High respektive CAN-Low ansluts [6].

Tabell 2. CAN-bus-sköld pinanvändning

Arduino Pin	Beskrivning
D2	Interrupt
D9	SPI_CS(standard)
D10	SPI_CS(valbar)
D11	SPI_MOSI
D12	SPI_MISO
D13	SPI_SCK

2.3.4 Motor-sköld R3

Motor-skölden är baserad på dubbla fullbryggedrivaren L298 som har möjlighet att styra upp till två induktiva laster separat på kanalerna A och B. Genom att programmera Arduinon kan riktning samt hastighet bestämmas på var och en av de två kanalerna. Skölden ger också möjligheten att läsa av strömförbrukningen hos varje motor [7].

L298 som är monterad på motor-skölden har två separata strömanslutningar en för motorförsörjning och en för logik. Då motorströmmen oftast överskrider den ström som Arduinon kan leverera måste motor-skölden matas med en extern ström källa. Detta görs via motor-sköldens Vin ingång på skruvplinten. Om dc-motorn kräver mer än 9 volt rekommenderas det att separera den andra Vin anslutningen(inte den vid skruvplinten) från Arduino kortet för att undvika eventuella skador. Detta görs genom att klippa bort Vin anslutningen på baksidan av skölden [7].

Tabell 3. Motor-sköld R3 pinanvändning

Arduino pin	Kanal	Beskrivning
D13	B	Riktning
D12	A	Riktning
D11	B	PWM
D9	A	Broms
D8	B	Broms
D3	A	PWM
A0	A	Strömavkänning
A1	B	Strömavkänning

2.4 Mjukvara

2.4.1 Arduino

Arduino programmeras i ett C-liknande språk och uppladdning av kod till Arduinon sker via en USB kabel. Ett Arduino program utgår från två grundfunktioner:

- setup() – Denna funktion körs bara en gång i början av programmet och kan till exempel användas för att konfigurera in/utgångar.
- loop() – Denna funktion körs hela tiden så länge Arduinon är igång.

2.4.2 CANalyzer

CANalyzer används för att analysera CAN-meddelanden samt kontrollera ifall och vilken kommunikation som sker på bussen. Det kan också användas för att skicka eller logga data på bussen. Ett CAN-kort som sätts in i datorn används för att ansluta sig på CAN-nätverket [8]. För att strukturera och modifiera data används databasfiler som skapas i ett program som heter CANdb++. Databasfilerna kan man sedan associera med CANalyzer och få sin önskade struktur över alla CAN-meddelanden [9].

I detta projekt används CANalyzer för att skicka ventilläge till regulatorn samt ta emot och analysera all data som skickas från regulatorn.

3 Metod

Under den första veckan skrevs en planeringsrapport där arbetet strukturerades upp genom att sätta upp olika delmål som skulle följas under arbetets gång. Delmålen ändrades en del under arbetets gång och resulterades tillslut upp i följande delmål:

Förstå problemet & Studera litteratur
Det första som gjordes var att få en överblick av problemet som skulle lösas. Därefter kunde relevant litteratur studeras. Eftersom det fanns begränsade kunskaper inom CAN-kommunikation lades en stor vikt på att förstå hur det fungerar.

Val av mikrokontroller
Nästa mål var att välja en mikrokontroller. Det viktigaste i åtanke här var att mikrokontrollern skulle ha stöd för CAN-kommunikation. Dessutom fanns det i åtanke att på ett enkelt med sätt kunna kontrollera hastighet och riktning hos en dc-motor.

CAN-kommunikation.
Det tredje delmålet var att få CAN-kommunikation med mikrokontroller att fungera. Det vill säga att kunna ta emot och sända ett CAN-meddelande. Som kommunikationspartner med mikrokontroller användes CANalyzer. För att lära sig hur CANalyzer fungerade söktes information på internet samt användes ”hjälp” fönstret i CANalyzer.

A/D omvandling samt PWM
Nästa delmål som sattes upp var att få mikrokontroller att kunna generera en PWM signal samt A/D omvandla ett spänningvärde. PWM signalen studerades med oscilloskop.

Ventilstyrning
I det här delmålet studerades ventilen. Målet var att kunna justera ventilöppningen genom att ändra hastighet och riktning på ventilens dc-motor. Relevanta mätningar så som dc-motorns strömförbrukning studerades med ett oscilloskop. Utöver detta lästes ventilläget av genom att A/D omvandla utsignalen av ventilens lägesgivare.

Regulator
Efter att bekantat sig med ventilen konstruerades en PI-regulator för ventilen. För att åstadkomma detta användes tidigare kunskaper inom området. Regulatorn testades sedan genom att undersöka stegsvar ändringar.

Montering
Det sista steget var att montera ventil regulatorn i en låda. En låda införskaffades där hål för montering av mikrokontroller och kontaktdon mättes upp och borrades ut. Nödvändiga komponenter löddes fast på ett kretskort dessutom tillverkades kablage för CAN-kommunikation och spänningsmatning.

4 Genomförande

4.1 Val av mikrokontroller

Det främsta kravet som ställdes på mikrokontrollern var stöd för CAN-kommunikation. Mikrokontrollern Arduino Uno har valts att användas främst därför att Arduino är populärt och tillåter öppen källkod vilket gör det lätt att hitta information på internet. En annan anledning till valet av Arduino är att det finns färdiga kretsar med specifika funktioner som kallas för ”Arduino Shields” som kan monteras ovanpå Arduino Uno kortet. I detta projekt används en CAN-bus-sköld som möjliggör CAN-kommunikation hos mikrokontrollern. Dessutom används en motor-sköld där det finns en fullbryggedrivare som kommer att utnyttjas för att driva dc-motorn i ventilen.

4.2 CAN-kommunikation

4.2.1 Arduino

Tanken är att regulatorn ska ta emot ett börvärde via CAN-kommunikation, dessutom ska regulatorn rapportera tillbaka sitt ventilläge samt en del andra saker som är relevanta att skicka.

För att möjliggöra CAN-kommunikation hos Arduinon användes en CAN-bus-sköld som monterades ovanpå Arduino Uno-kortet, sedans anslöts CAN-H och CAN-L till respektive ingång på CAN-bus-skölden.

En del programmering behövs göras för att Arduino enheten ska bli fullt kapabel till att sända och skicka CAN-meddelanden. Till CAN-bus-skölden finns det ett tillgängligt bibliotek där det finns ett antal olika funktioner. De som använts i detta projekt är:

- **CAN.begin(*can_speed*);**
  Initierar CAN bussen och sätter buadrate.
  ”can_speed” representerar buadrate.
- **CAN.init_mask(unsigned char num, unsigned char ext, unsigner char ulData)**
  CAN.init_filt(unsigned char num, unsigned char ext, unsigner char ulData)
  Dessa två funktioner används för att endast ta emot data från särskilda CAN-IDs.
  ”num” respresenterar vilka register som ska användas.
  ”ext” anger om CAN meddelandet är av typ standard ’0’ eller extended ’1’.
  ”ulData” representerar innehållet av masken i filtret.
- **CAN.checkRecieve( );**
  Söker efter inkommande data.
- **CAN.readMSGBuf(unsigned char len, unsigned char buf);**
  Läser av data på CAN bussen.
  ”len” representerar datalängden.
  ”buf” är där all data lagras.
- **CAN.sendMsgBuf(INT8U id, INT8U ext, INT8U len, data_buf);**
  Sänder ut data på bussen.
  ”id” representerar meddelandets CAN ID.

”ext” anger om CAN meddelandet är av typ standard ’0’ eller extended ’1’.
”len” representerar längden på datan som skickas.
”data_buf” representerar den data som skickas.

CAN bussen initieras först med en viss baudrate i detta fall 500 kbps, detta görs med funktionen ”CAN.begin”. Sedan går det att välja om man endast vill ta emot data från specifika CAN-IDs. I ett CAN-nätverk skickas data till alla noderna och det är nodernas uppgift att besluta om datan är relevant. I stora CAN-nätverk kan det därför vara bra att bara ta emot data som är relevant, detta görs med funktionerna ”CAN.init_mask” och ”CAN.init_filt”.

För att ta emot ett CAN-meddelande används först funktionen ”CAN.checkrecieve” som kollar efter inkommande data, därefter används ”CAN.readMSGBuf” som läser av all data i CAN-meddelandet och lagrar det i en array av storlek antal bytes data meddelandet innehåller. Sändning av CAN-meddelanden görs med funktionen ”CAN.sendMsgBuf”.

4.2.2 CANalyzer

Som kommunikationspartner med Arduinon användes programmet CANalyzer. Med hjälp av detta program kunde all datatrafik på CAN-bussen spåras och tolkas vilket gjorde det enkelt att se om CAN-meddelanden från Arduinon sändes korrekt.

Det går att sända meddelanden ifrån CANalyzer genom att skapa ett så kallat IG Block(Interactive Generator Block). Med detta IG block kunde CAN-meddelanden till Arduinon skickas.

4.3 Ventilstyrning

4.3.1 Modifikation av motor-sköld R3

Motor-skölden används för att styra ventilmotorn och monteras ovanpå CAN-bus-skölden som redan har monterats på Arduino Uno kortet. En dubbel fullbryggedrivare L298P är installerad på motor-skölden som har möjlighet att styra två motorer en på kanal A och en på kanal B. Bara kanal B har valts att användas eftersom endast en motor ska drivas.

En del modifikationer gjordes på motor-skölden för att få den att fungera ihop med CAN-bus-skölden. Både pin 11 och 13 på Arduinon används för styra kanal B på motor-skölden, problemet är att CAN-bus-skölden också använder sig av dessa två pins. Denna pinkonflikt löstes genom att klippa av pin 11 och 13 från baksidan av motor-skölden så att dem inte konflikterar med CAN-bus-skölden. Sedan kopplades pin 6 och 7 till pin 11 respektive pin 13 på motor-skölden.

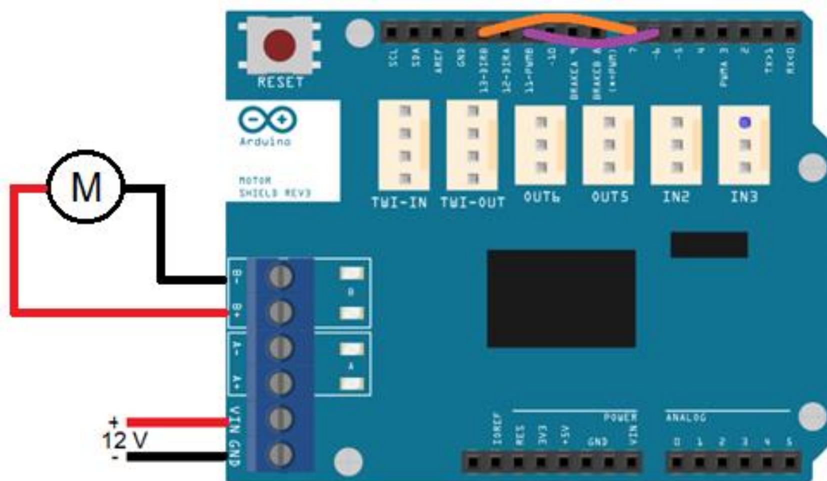
Tabell 4. Pin användning efter modifikation av motor-sköld

Arduino Pin	CAN-bus-sköld	Motor-sköld
D6	Ej använd	Pwm, kanal B (ansluten till pin 11)
D7	Ej använd	Riktning, kanal B (ansluten till pin 13)
D11	SPI_MOSI	Pwm, Kanal B
D13	SPI_SCK	Riktning, kanal B

En annan modifikation som gjordes med motor-skölden var att separera ”Vin anslutningen” mellan motor-skölden och CAN-bus/Arduino Uno kortet genom att klippa av pin ”Vin” från baksidan av motor-skölden. Anledningen till att detta gjordes var att motor-sköldens fullbryggedrivare matas externt med 12 volt för att spänningsförsörja ventilens dc-motor. Detta görs via ”Vin” ingången därmed finns det risk att skada Arduino Uno kortet.

4.3.2 Drift av ventilmotor

Ventilmotorn drivs med 12 volt likspänning därmed matades fullbryggedrivaren på motor-skölden med 12 volt via skruvterminalens Vin ingång. Därefter kunde ventilmotorn anslutas till kanal B på motor-skölden enligt figur 4 nedan.



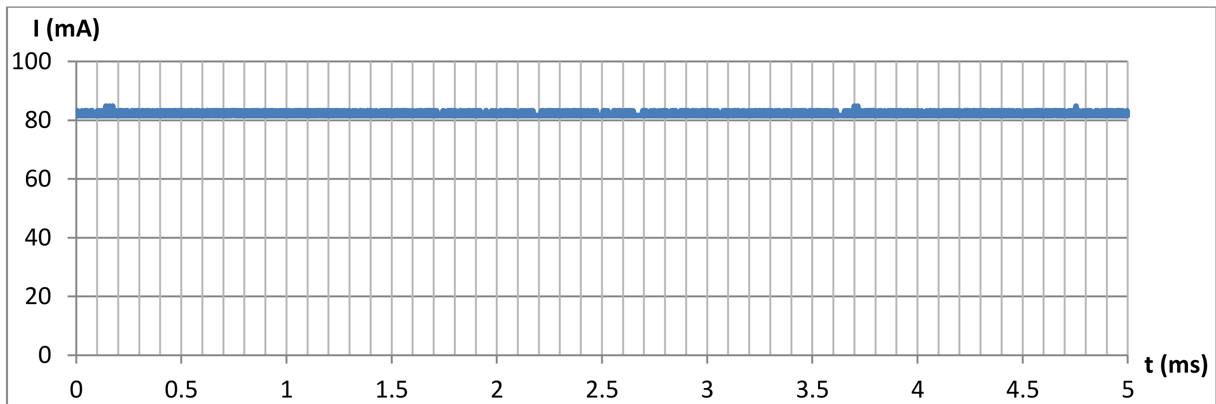
Figur 4. Bild över hur motorskölden kopplades till ventilmotorn. Dessutom visas att pin 6 och 7 har kopplats till pin 11 respektive 13 som beskrivs i avsnittet ”4.3.1 Modifikation av motor-sköld R3”.

Genom att skriva en etta eller nolla till pin 7 kan strömmens riktning på kanal B ändras. Pulskvoten hos kanal B ändras genom att skriva ett värde mellan 0 (0 % pulskvot) och 255(100 % pulskvot).

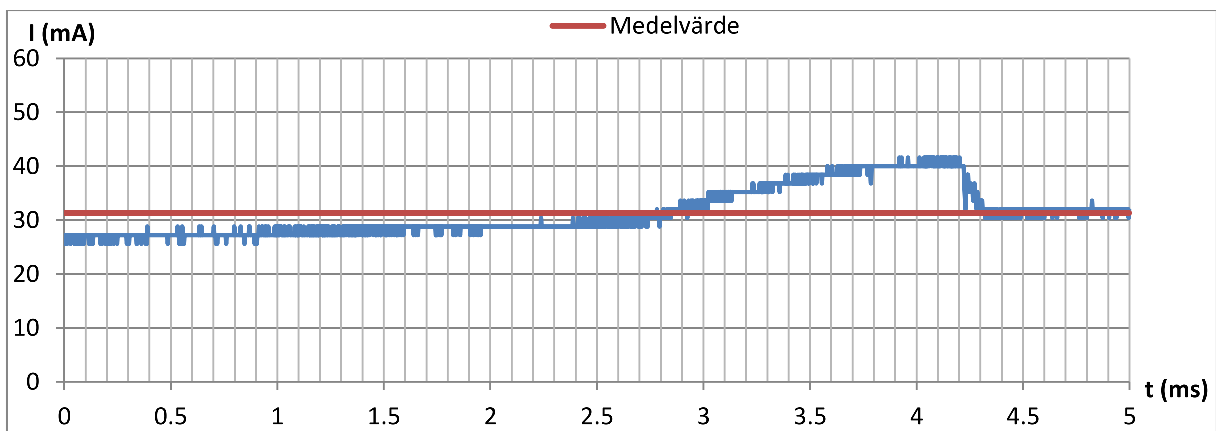
För att verifiera ändring av riktning samt pulskvot hos den PWM modulerade fullbryggedrivaren användes en potentiometer som kopplades in i en av de analoga ingångarna på mikrokontrollern. Mikrokontrollern programmerades så att vid avläst mittenläge på potentiometern så skickas 0 % pulskvot ut på h-bryggan (motor står still). Om man sedan reglerar potentiometern till dess top- eller botten läge så ökar pulskvoten gradvis till 100 % samt så byter h-bryggan riktning beroende på om potentiometern regleras till sitt top- eller botten läge. På så sätt var det enkelt att verifiera ändring av riktning samt reglering av hastighet hos dc-motorn.

4.3.3 Mätningar av ventilmotorn

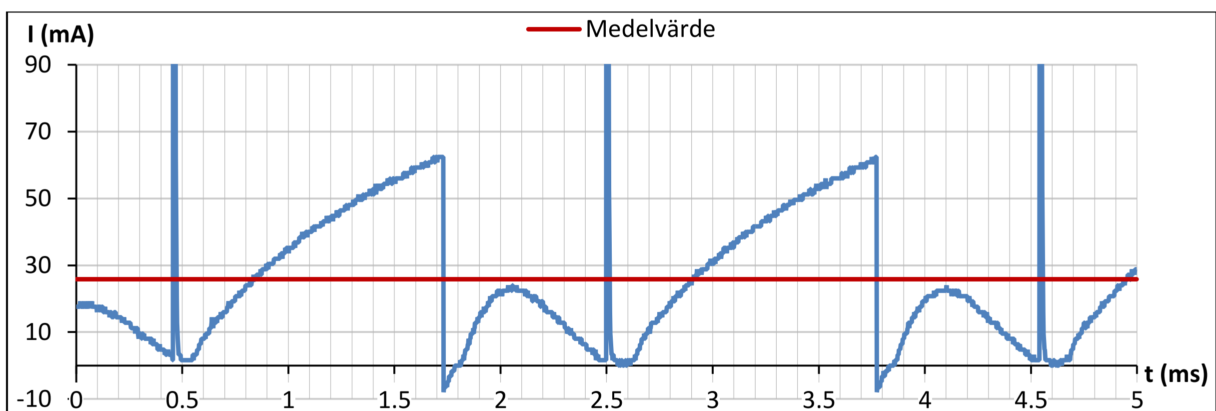
Ventilmotorns strömförbrukning har undersökts med ett oscilloskop vid olika vridmoment.



Figur 5. Strömförbrukning hos ventilens dc-motor, då motorn ger maximalt vridmoment.



Figur 6. Strömförbrukning hos ventilens dc-motor, då motorn roterar fritt med 100 % pulskvot.



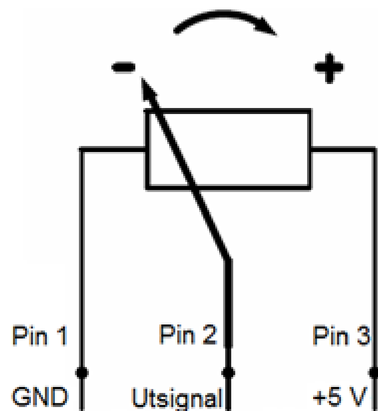
Figur 7. Strömförbrukning hos ventilens dc-motor, då motorn roterar fritt med 63 % pulskvot.

Från mätningarna av ventilmotorn kan det noteras att dc-motorn i värsta fall drar runt 80 mA, men i normala fall kommer strömförbrukningen vara mindre än så. Eftersom motorn sällan kommer ge max vridmoment.

En annan nämnvärd mätning som gjordes var att mäta den minimala pulskvot som behövdes för att dc-motorn skulle röra sig. Detta är bra att veta så att motorn inte drar onödig effekt utan att röra sig. Denna pulskvot uppmättes till 55 %.

4.3.4 Ventilens lägesgivare

Ventilen har en lägesgivare i form av en potentiometer som indikerar ventilöppningens läge. Denna lägesgivarpotentiometer matades med +5 volt och utsignalen kopplades till en av de analoga ingångarna på mikrokontrollern.



Figur 8. Koppling över ventilens lägesgivarpotentiometer.

Något nämnvärt som upptäcktes var att när ventilen är i sitt bottenläge eller toppläge så visar lägesgivaren inte 0 % respektive 100 %. Istället visar lägesgivaren ca 25 % när ventilen börjar öppnas och ca 90 % när ventilen närmar sig helt öppen. Eftersom ventilen är skalad till 0-100% så måste den skalas om till 25-90%.

4.4 Ventilregulatorns huvuddelar

4.4.1 PI-regulator

Regulatorn konstruerades genom att först läsa av ärvärdet samt ett börvärdet. Ärvärdet läses av från ventilens lägesgivare och A/D omvandlas genom en av de analoga ingångarna på Arduinon. Börvärdet tas emot på CAN-bus och är 10 bitar långt liksom ärvärdet som A/D omvandlas med 10 bitars upplösning. Reglerfelet kan därefter räknas ut:

$$Reglerfel = Börvärde - Ärvärde \tag{2}$$

Ventilmotorns riktning bestäms beroende på om reglerfelet är negativt eller positivt:

$$Reglerfel > 0 => BYT\ RIKTNING$$
$$Reglerfel < 0 => BYT\ RIKNTING$$

En viktig del i konstruktionen av regulatorn är att räkna ut alla parametrar för regulatorn i ett bestämt regelbundet intervall. Den främsta anledningen till detta är att det blir lättare att räkna fram parametrarna som är beroende av förändring i tid, det vill säga både den integrerande och deriverande delen. Efter beslut har det dock valts att utesluta den deriverande delen då styrsystemet fungerar utmärkt med endast en proportional och integrerande del.

Proportionaldelen räknades ut genom att först ta absolutbeloppet av reglerfelet, detta för att om ärvärdet är större än börvärdet fås ett negativt reglerfel vilket ställer till det när utsignalen ska beräknas. Sist multipliceras absolutbeloppet av reglerfelet med en konstant, K_p enligt ekvation 3:

$$P = abs(reglerfel) * K_p \tag{3}$$

Eftersom regulator parametrarna har valts att samplas i ett intervall T_s, blir matematiken lättare i processen för att räkna ut den integrerande delen. Integraldelen räknades ut genom att summera föregående reglerfel multiplicerat med en konstant K_i samt en samplings tid T_s:

$$I = \int (abs(reglerfel) * K_i * T_s) \tag{4}$$

Nästa steg är att summera P och I termerna för att generera en PWM utsignal. PWM signalen kan anta värden mellan 0 (0 % pulskvot) och 255 (100 % pulskvot). Från tidigare mätningar hos ventilmotorn noterades det att det krävdes cirka 55 % pulskvot innan ventilmotorn började röra sig. För att ventilmotorn inte ska dra onödig effekt utan att röra sig adderades ett värde 120, vilket gav en grundpulskvot på 47 %. Vid reglerfel generas då en PWM signal mellan värdena 120 (47 % pulskvot) och 255 (100 % pulskvot). Slutgiltiga pulskvoten beräknas därefter genom att addera proportional delen samt integral delen:

$$pwmOUT = 120 + P + I \tag{5}$$

För att få en bra reglering på systemet dimensionerades parametrarna Kp och Ki. Ingen speciell regel som till exempel Ziegler-Nicholasmetoden har använts för att välja parametrarna. Istället dimensionerades parametrarna manuellt genom att undersöka hur systemet reglerar in sig vid olika stegsvar. Stegsvaren analyserades genom att plotta börvärdet och ärvärdet i CANalyzer.

Först dimensionerades systemet med endast P-delen det vill säga bara förstärkningen av reglerfelet. Systemet lyckades bli någorlunda stabilt med endast en P-del, men i vissa fall upptäcktes det att när systemet närmade sig börvärdet så ”orkade” systemet inte reglera in sig den sista biten. Detta statiska reglerfel uppstår när reglerfelet är så litet att produkten av proportional förstärkningen och reglerfelet inte blir tillräckligt stort för att tvinga in systemet den sista biten.

För att åtgärda detta problem lades I-delen till som summerar ihop reglerfelen med tiden. Detta gör att även det minsta reglerfelet kommer att ackumuleras tills att det är tillräckligt stor för att tvinga in systemet till det önskade börvärdet. Ett nämnvärt problem som stöttes på vid implementationen av den integrerande delen var att vid stora reglerfel (systemet startar långt ifrån börvärdet) blir I-delen väldig stor på grund av summationen av de stora reglerfelen. För att undvika detta problem programmerades ventilregulatorn till att nollställa I-delen tills dess att reglerfelet var litet. Slutligen dimensionerades K_i parametern tills systemet blev stabilt. De slutgiltiga parametrarna som använts är:

$$K_p = 0.2; \;\; K_i = 0.5$$

Kod 1.PI regulator för ventilstyrningen

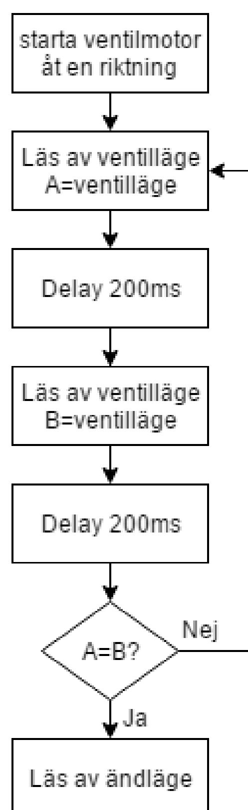
```
void PIcalc(float K_P, float K_I){
  float sampletime = 100; // sample time in ms
  unsigned long int current_time = millis();
  long int time_change = current_time - last_time;

  if(time_change >= sampletime) //sample every 100ms
  {
    P_term = abs(errorSignal)*K_P; //calculate P term
    I_term += abs((errorSignal*K_I))*(sampletime/1000); //calculate I term
    if(I_term >= 135){I_term = 135;} //integration limit

    pwmOut = 120 + P_term + I_term;
    if(pwmOut > 255){pwmOut = 255;}

    last_time = current_time;
  }
}
```

4.4.2 Autokalibrering av ventil

Ventilen är skalad till 0-100% läge, men slår i botten runt 25 % och 90 %. Därmed måste ventilen skalas om till 25-90%. Detta kan självklart göras manuellt i koden men eftersom regulatorn skall användas för flera ventiler och ventilerna förmodligen inte har exakt samma ändlägen blir det tidskrävande att manuellt kalibrera ventilerna var för sig. Genom att göra en autokalibrering skulle man slippa allt detta. Dessutom är det bra för ventilen att ”motioneras” det vill säga att den gör en rörelse mellan fulla ändlägen för att motverka att ventilen kör in sig inom ett visst arbetsområde. Om man på någon sätt automatiskt kan detektera respektive ändläge för ventilen kan en autokalibrering göras.

Den första tanken som kom upp för att detektera ventilens ändlägen var att mäta strömmen hos ventilmotorn. När ventilmotorn går i botten ger den maximalt vridmoment och drar mer ström än under normal drift, på så sätt kan man detektera när ventilen är i ett ändläge. Men denna idé grundade sig på att använda sig av motorsköldens strömavkännings pin som har en upplösning på 1.65V/A. Då ventilmotorn max drar 80 mA är detta alldeles för dålig upplösning, därmed beslutades det att använda en annan metod.

Den andra metoden för att detektera respektive ändläge är relativt lätt att implementera. Med denna metod så startar man ventilmotorn åt ett håll och läser av ventilens läge i intervall om 200 ms. Om sedan två avläsningar är samma betyder det att ventilen är i ett ändläge. Efter avläsning av första ändläget byter motorn riktning och gör samma procedur för detektera det andra ändläget. Proceduren för att detektera ett ändläge visas i figur 9.



Figur 9. Flödesschema över detektering av ändläge.

4.4.3 DR (Wake-up) signal

Ventilregulatorn använder sig av en ”wake-up” signal även kallad ”DR” signal som kopplas in i en av de digitala ingångarna på Arduinon. Denna signal kan beskrivas som ”tändning av” eller ”tändning på” det vill säga den avgör om regulatorn ska var inaktiv eller aktiv.

- DR = 0 => Ventilen går till 0 % ventilöppning oavsett CAN-kommando
- DR = 1 => Ventilen väntar på CAN-kommando och reglerar sedan in sig till en viss ventilöppning.
- Tillslag av DR => Om DR signalen ändrar värde från 0 till 1 så motioneras ventilen det vill säga den gör en rörelse mellan fulla ändlägen och sedan återgår till sitt ursprungliga läge.

4.4.4 Byte av CAN-ID

Då det önskas att styra upp till fyra ventiler på samma CAN-bus behöver man kunna sätta ett enskilt CAN-ID för varje ventilregulator. Detta har valts att implementeras hårdvarumässigt genom att ha en bcd-omkopplare som är kopplad till två av de digitala ingångarna på Arduinon. Programmet är uppbyggt så att CAN-ID bara sätts en gång i början av programmet, för att byta CAN-ID måste man starta om regulatorn och ändra läge på bcd-omkopplaren. Vilket CAN-ID som sätts beroende på bcd omkopplarens läge beskrivs i tabellen nedan.

Tabell 5. Tabell över vilket CAN-ID som sätts beroende på bcd omkopplarens läge.

BCD-omkopplare läge (dec)	Kontroller	CAN-ID TX (hex)	CAN-ID RX (hex)
0	Nr 1	0x19000011	0x19000012
1	Nr 2	0x19000021	0x19000022
2	Nr 3	0x19000031	0x19000032
3	Nr 4	0x19000041	0x19000042

4.4.5 CAN-meddelande

Ventilregulatorn skickar ut ventilläge, spänningsvärde(från ventilens lägesgivarpotentiometer) samt en ”heartbeat” signal på CAN-bus. Heartbeat signalen skiftar värde mellan 0 och 255 med en frekvens på 1 Hz för att kunna se om regulatorn är igång och fungerar.
I ett CAN-meddelande kan totalt 8 bytes skickas. Nedan visas vad som skickas i varje byte ifrån ventilregulatorn. Observera att spänningsvärdet från ventilens lägesgivare är 10 bitar långt vilket kräver två bytes.

Tabell 6. Överblick av vad varje byte i CAN-meddelandet som skickas ifrån ventilregulatorn innehåller.

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Bit 0-8	Bit 8 -18		Bit 24-32				
Ventil position Procent 0=0% 255=100%	Ventil Position Volt 0=0 V 1023= 5V		heartbeat signal	Ej använ d	Ej använd	Ej använd	Ej använd

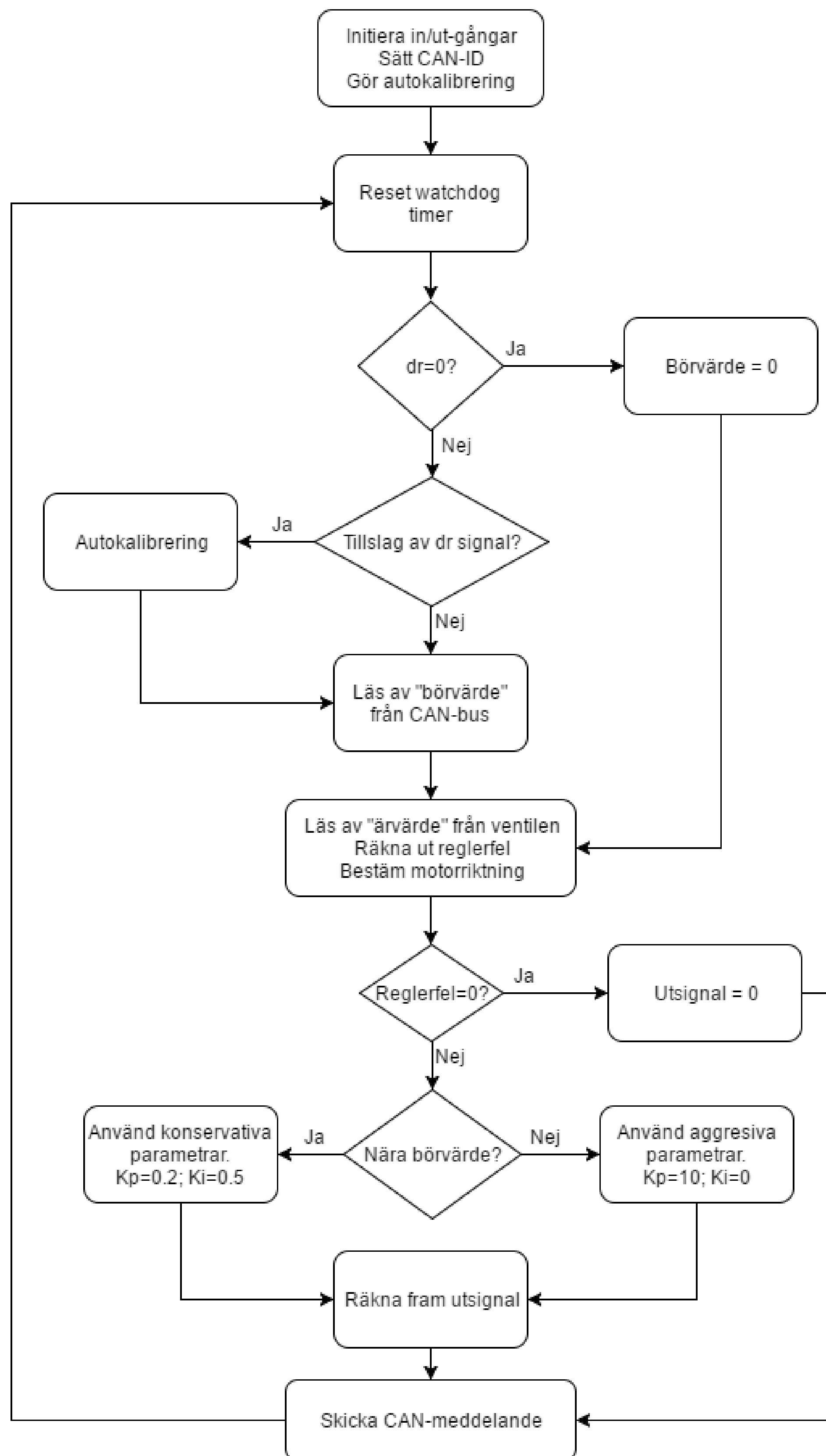
4.4.6 Watchdog

Då regulatorn kan komma att stå obevakad under en längre tid vid prov kan det vara bra att ha en vakthund som ser till att processorn arbetar korrekt. Oväntade fel i programmet kan uppstå, till exempel kan programmet fastna i en oändlig loop eller något liknande. När ett sådant fel uppstår utför vakthunden en reset som kan få processorn att arbeta korrekt igen.

Vakthunden använder sig av en timer som är inställd på ett visst antal sekunder, i detta fall 8 sekunder. Delar av programmet har till uppgift att återställa denna timer. Om programmet hamnar i ett tillstånd där timer inte kan återställas kommer timern att hinna räkna färdigt därmed vet vakthunden att ett fel har inträffat och utför då en reset.

4.4.7 Flödesschema

Hur programmet är uppbyggt och fungerar visas i flödesschemat nedan.



Figur 10. Flödesschema över programmet.

4.5 Spänningsmatning

4.5.1 Arduino Uno och sköldar

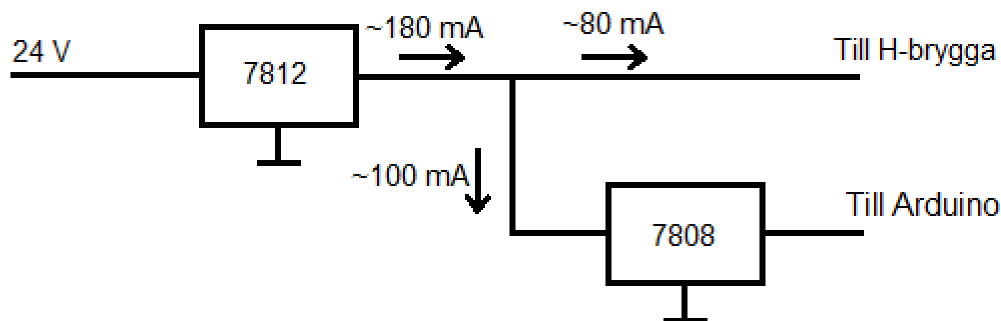
Arduino Uno – Det finns några olika möjligheter att spänningsförsörja Arduino kretsen. Den metoden som valts i detta projekt är att spänningsmata genom Arduinons Vin pin. Vin anslutningen är kopplad till en 5 volt regulator som sedan spänningsförsörjer Arduino kretsen. Eftersom 5 volt regulator som sitter inuti Arduinon medför ett visst spänningsfall valdes det att mata Vin anslutningen med 8 volt.

CAN-bus -sköld – Denna sköld spänningsmatas från Arduino Uno-kortet.

Motor-sköld – Denna sköld matas externt med 12 V via skruvterminalens Vin ingång för att driva ventilmotorn.

4.5.2 Spänningsreglering

I provrum finns det bara tillgång till 24 volt. Spänningsregulatorerna LM7812 samt LM7808 har använts för att reglera ner spänningen till 12 volt för matning av H-brygga respektive 8 volt för matning av Arduino krets. Nedan visas hur spänningsregulatorerna är kopplade samt hur mycket ström som beräknas passera regulatorerna.



Figur 11. Kopplingschema över hur spänningsregulatorerna LM7812 och LM7808 är kopplade för att spänningsförsörja Arduino kretsen samt motor-sköldens h-brygga.

Efter beräkningar och mätningar uppskattades det att Arduino kretsen samt dc motorn kommer dra runt 100 mA respektive 80 mA. Då fås en ström på 180 mA genom regulatorn LM7812 som visas i figur 11. För att säkerställa att regulatorn inte överhettas gjordes följande beräkningar:

LM7812 regulatorn reglerar ner spänningen från 24 volt till 12 volt det ligger då ett spänningsfall på 12 volt över regulatorn. Med en rumstemperatur $T_A = 30℃$ och $\mathrm{R}_{\theta\mathrm{JA}} = 50$ ℃/W fås en chip temperatur på:

$$T_j = T_A + P_D * \mathrm{R}_{\theta\mathrm{JA}} = 30℃ + 12V * 180mA * 50 ℃ / \mathrm{W} = 138 ℃ \quad (6)$$

T_{j-max} ligger på 150 ℃ därmed klarar sig regulatorn utan kylfläns. Dock är det aldrig bra att ligga allt för nära gränsvärdet vilket ledde till ett beslut att montera fast LM7812 regulatorn i byggnationslådan aluminiumplåt vilket gav lite extra kylning.

4.6 Byggnation av låda

4.6.1 Val av låda

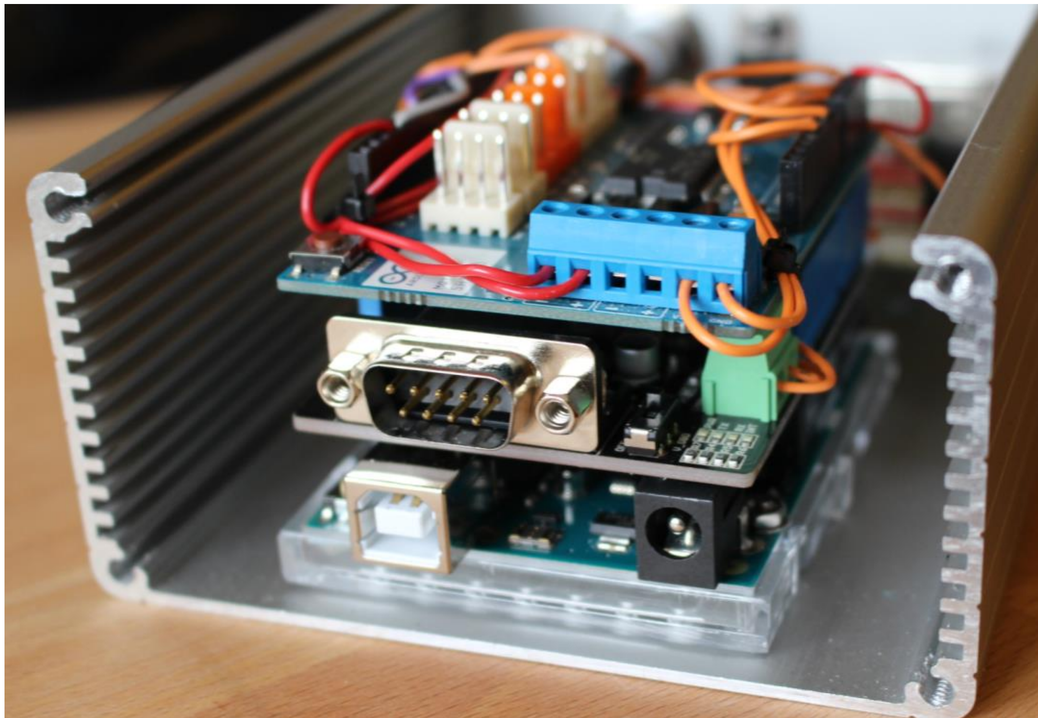
När funktionaliteten hos ventilregulatorn var som önskad monterades alla bestående delar av regulatorn in i en låda för att enkelt kunna testas och användas i provrum. Vid val av låda gjordes en skiss på hur alla delar ska placeras i lådan samt vilka dimensioner på lådan som krävs för att allting ska få plats.

4.6.2 Ritningar för borrhål

Det första som gjordes med lådan var att mäta upp och borra nödvändiga hål för montering av Arduino, d-sub kontakt samt lemo kontakt. Hålet för d-sub kontakten gjordes med hjälp av en d-sub punsch. Dessutom mättes ett hål upp för att montera fast regulatorn LM7812 i lådans sidoplåt för kylning. Mekaniska ritningar finns tillgängliga i bilaga 3.

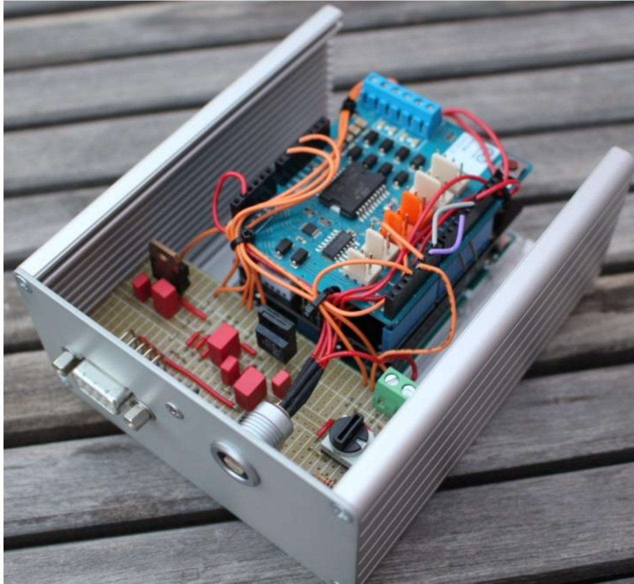
4.6.3 Montering och koppling

Arduino Uno kortet monterades fast i respektive hål som hade borrats ut för Arduinon. CAN-bus-skölden samt motor-skölden monterades ovanpå Uno-kortet enligt figur 12.



Figur 12. Bild över hur CAN-bus-skölden och motor-skölden har monterats på Arduino Uno kortet. Motor-skölden(högst upp i bild) är monterad ovanpå CAN-bus-skölden som i sin tur är monterad på Arduino Uno kortet.

Nästa steg var att löda fast diverse komponenter på ett kretskort. Kretskortet klipptes ut till dimensionerna 100x40 mm för att passa in i lådan. Regulatorer, bcd-omkopplare samt d-sub kontakten löddes sedan fast i kretskortet enligt kopplingsschema i bilaga 2. Därefter tillverkades kablage för lemo och d-sub kontaktaterna. Slutligen monterades alla delar fast och kablage drogs till respektive pin ingång på Arduinon enligt pin konfigurationen som visas i bilaga 1.



Figur 13. Inuti färdig monterad låda.

5 Resultat

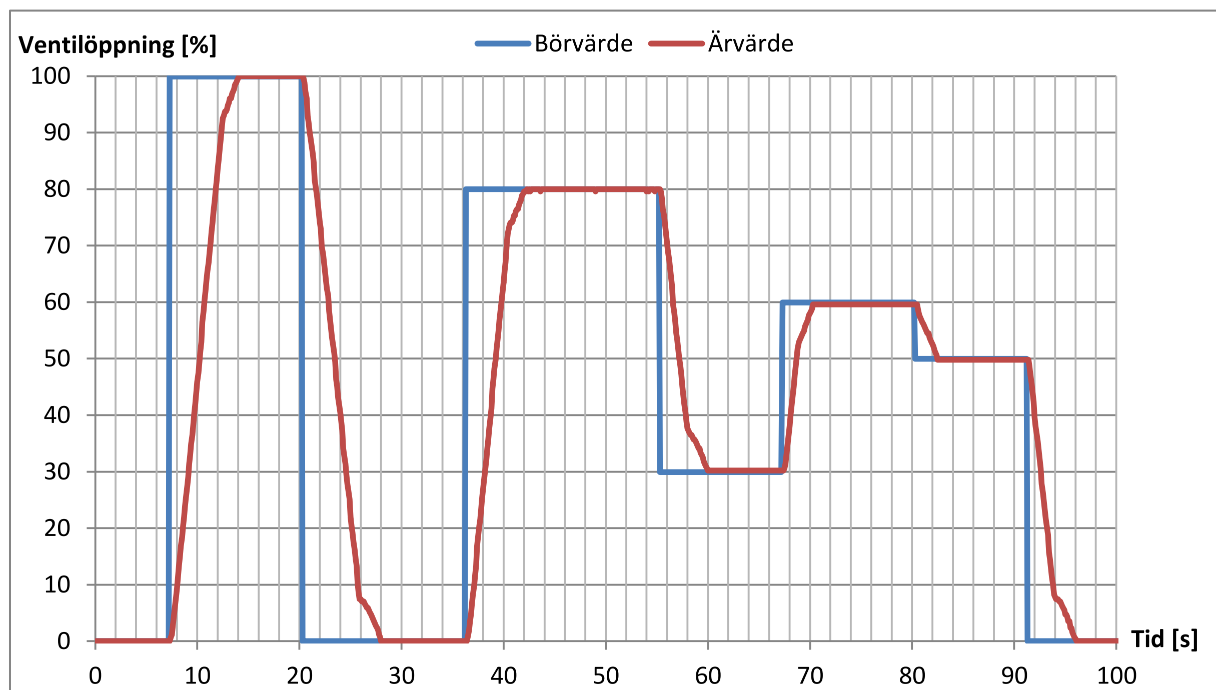
De kravspecifikationer som beskrivs i kapitel 1.5 har uppnåtts med ett tillfredställande resultat.

5.1.1 CAN-kommunikation

Innan projektets start fanns det begränsade kunskaper och erfarenheter inom CAN-kommunikation, därmed blev en av de större utmaningarna att få ventilkontrollern att fungera i ett CAN-nätverk. Denna utmaning har klarats av och en god förståelse över hur ett CAN-nätverk är uppbyggt och fungerar har åstadkommits. Programmet CANalyzer har använts för att kommunicera med ventilregulatorn samt har en CAN-databas för ventilstyrningen definierats och byggts upp.

5.1.2 Regulator

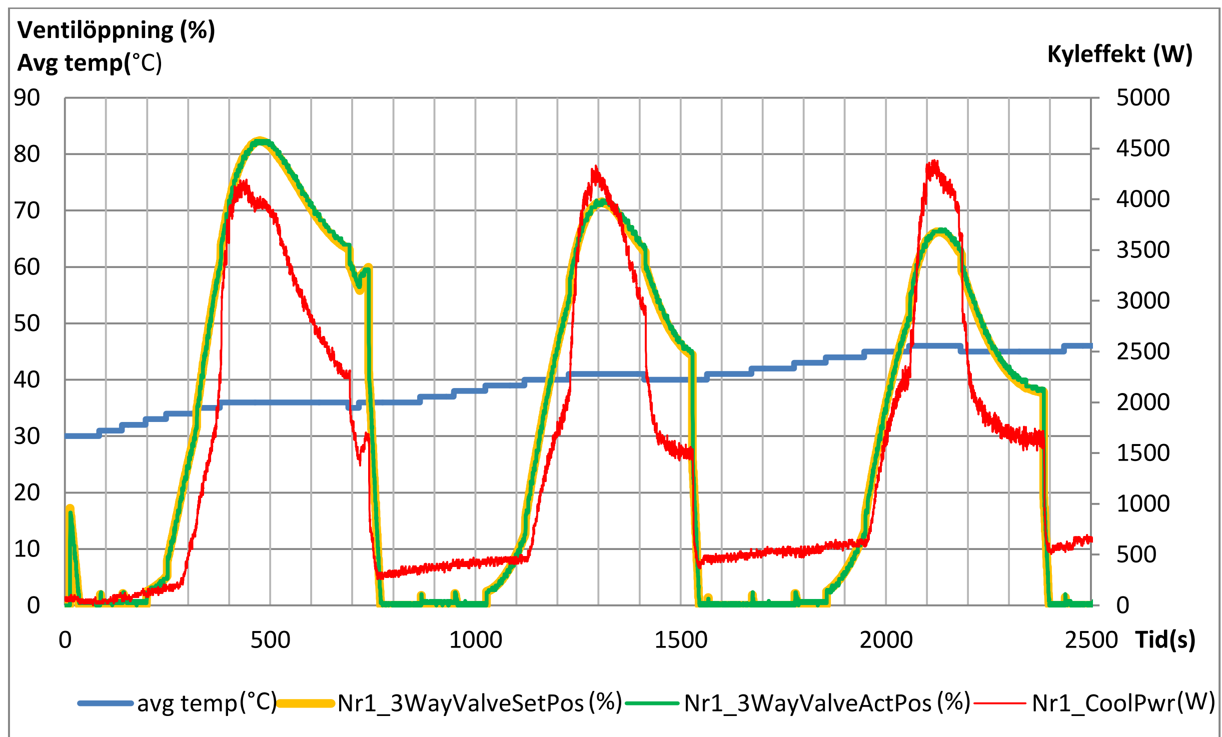
En fungerande PI-regulator för reglering av en trevägsventil har konstruerats med hjälp av en Arduino som programmerats i ett C-liknande språk. Regulatorn har dimensionerats genom att förstå vilken uppgift respektive term i en PI-regulator har. Dessa kunskaper har sedan använts för att analysera stegsvar och göra nödvändiga parametrar ändringar för att få ett så bra reglersystem som möjligt.



Figur 14. Resultat över hur ventilregulatorn reglerar in sig till ett antal olika börvärden.

5.1.3 Fungerande produkt

Regulatorn har byggts ihop i en låda där diverse komponenter har lödats fast i ett kretskort samt har nödvändig kontaktering för spänningsförsörjning och CAN-kommunikation installerats. Slutprodukten har testats i provrum där dess uppgift var att reglera kylvätskeflödet för temperaturkontroll av ett hybridbatteri.



Figur 15. Resultat från provrum. Systemet reglerar in sig på 3 olika set-temperaturer. Vilket börvärde ventilen skall ha med avseende på set-temperatur sköts av en extern riggdator och skickas kontinuerligt till ventilregulatorn via CAN-kommunikation. Ventilens ärvärde (grön) följer ventilens börvärde (orange) väldigt väl.

6 Slutsats

6.1 Diskussion

Projektet har varit väldigt lärorikt och intressant. Det har varit upplysande att få utnyttja tidigare kunskaper men också att lära sig helt nya saker. De större utmaningarna med projektet var CAN-kommunikation och mjukvaruutvecklingen av programmet för ventilstyrningen. Genom att lägga ner en större proportion av tiden på detta så har jag lyckats med utmaningarna och åstadkommit en väl fungerande regulator med CAN-kommunikation vilket jag känner mig nöjd med.

6.1.1 CAN-kommunikation

I stora drag gick implementeringen av CAN-kommunikationen bra. När regulatorn testades i ett mindre CAN-nätverk fungerade allt som det var tänkt. Ett av det mer tidskrävande och svårtolkade problemet uppstod när regulatorn testades i ett större CAN-nätverk. Problemet som uppstod var att CAN-kommunikationen hos regulatorn plötsligt slutade fungera. Efter en hel del felsökning hittades orsaken till problemet, det visade sig att regulatorn inte kunde hantera all trafik på CAN-bussen. Detta löstes genom att sätta upp ett filter som filtrerar bort irrelevanta CAN-meddelanden. Tillslut fungerade CAN-kommunikationen felfritt.

6.1.2 Regulator och dimensionering

Implementationen av regulator gick bra. När det kom till att dimensionera regulator parametrarna användes en manuell metod och stegsvaren studerades i realtid med CANalyzer.

Man skulle kunna analysera stegsvaren på andra sätt. Som exempel kan man bestämma överföringsfunktionen för systemet och sedan använda matlab för att analysera stegsvaren. Sedan kan man använda sig av andra metoder som till exempel Ziegler-Nicholasmetoden för att bestämma regulator parametrarna. Men den metodiken som använts för detta system fungerade bra och gav ett tillfredsställande resultat.

6.2 Vidareutveckling

Denna CAN-styrda ventilregulator för temperaturkontroll av hybridbatterier skulle kunna vidare utvecklas på ett eller annat sätt. I nuvarande system sköter en extern riggdator hur ventilen ska regleras med avseende på set-temperatur. Denna reglering skulle kunna implementeras i mikrokontrollern. En annan förbättring som skulle kunna realiseras är att skicka ut en felsignal på CAN-bussen när något fel inträffar hos regulatorn. Detta kan vara behändigt att ha vid eventuell felsökning.

Referenser

[1] National Instruments. (2014). Controller Area Network (CAN) Overview. http://www.ni.com/white-paper/2732/en/ (hämtad 2016-05-10)

[2] Steve Corrigan. (2008). Introduction to the Controller Area Network (CAN). http://www.ti.com/lit/an/sloa101a/sloa101a.pdf (hämtad 2016-05-10)

[3] National Instruments. (2011). PID Theory Explained. http://www.ni.com/white-paper/3782/en/ (hämtad 2016-05-20)

[4] Thomas, Bertil. 2008. *Modern Reglerteknik*. 4. uppl. Stockholm 113 98: Liber AB.

[5] Arduino. Arduino Uno. https://www.arduino.cc/en/main/arduinoBoardUno (hämtad 2016-02-10)

[6] Seeeduino. CAN-BUS Shield V1.2. http://www.seeedstudio.com/wiki/CAN-BUS_Shield_V1.2 (hämtad 2016-02-10)

[7] Arduino. Arduino Motor Shield R3. https://www.arduino.cc/en/Main/ArduinoMotorShieldR3 (hämtad 2016-03-10)

[8] Vector. CANalyzer. http://vector.com/vi_canalyzer_en.html (hämtad 2016-05-15)

[9] Vector. DBC Communication Database for CAN. http://vector.com/vi_candb_en.html (hämtad 2016-05-15)

BILAGA 1 – Pin konfiguration

Tabell 7. Arduino pin användning. (Gråmarkerat=kabelanslutna).

Arduino Pin	Beskrivning
D0	Serial Com (RX)
D1	Serial Com (TX)
D2	Recieve interrupt
D3	CANID bcd-omkopplare (hög vid dec: 1, 3)
D4	CANID bcd-omkopplare (hög vid dec: 2, 3)
D5	DR(”Wake-up”)
D6	PWM (CH B) *(Ansluten till pin 11)
D7	Riktning (CH B)* (Ansluten till pin 13)
D8	Broms (CH B)
D9	NC (default)
D10	SPI_CS(selectable)
D11	SPI_MOSI*
D12	SPI_MISO
D13	SPI_SCK*
A0	Current sense (CH A)
A1	Current sense (CH B)
A2	NC
A3	NC
A4	Ventil lägesgivare
A5	NC
Motorsköld Vin (skruvterminal)	12 V
Kanal B motor-sköld (+)	Ventilmotor
Kanal B motor-sköld (-)	Ventilmotor

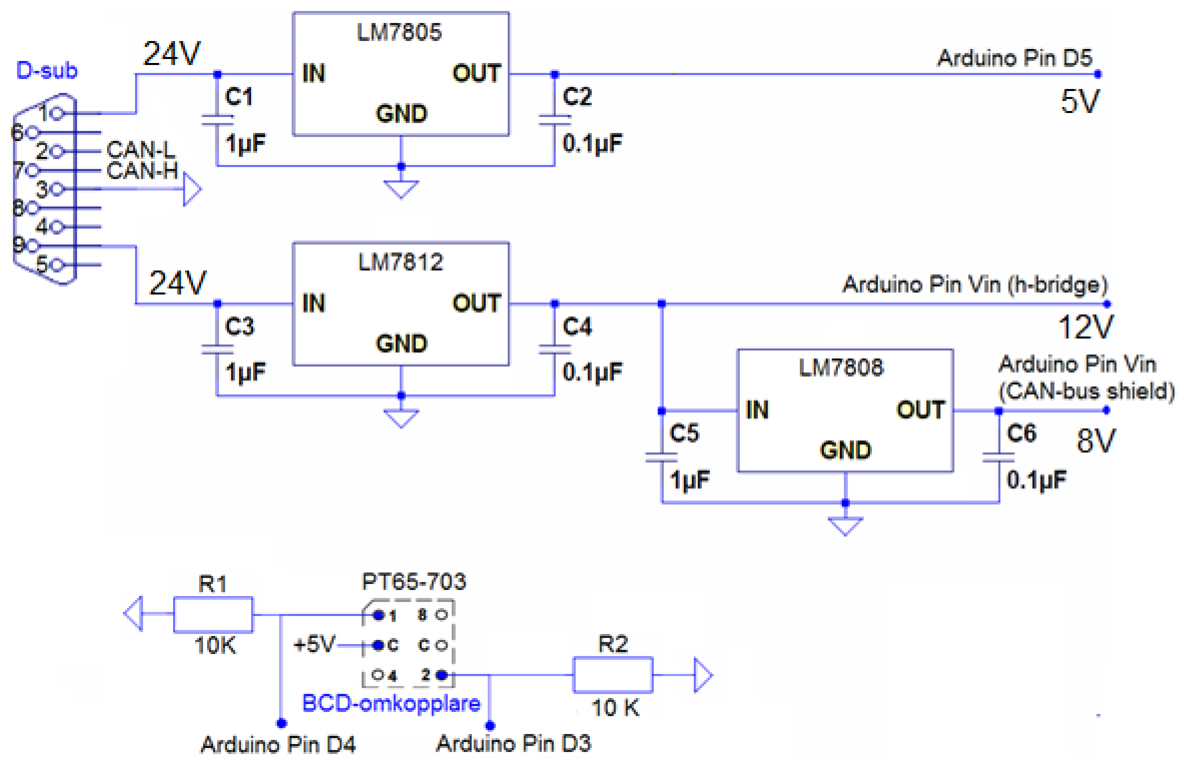
Tabell 8. Lemo kontakt pinkonfig.

Lemo Pin	**Beskrivning**	**Kopplas till Ventil Pin**
1	Kanal B motor-sköld (+)	6
2	Kanal B motor-sköld (-)	4
3	GND	1
4	Ventil lägesgivare	2
5	+5V	3
6	NC	NC

Tabell 9. D-sub kontakt pinkonfig.

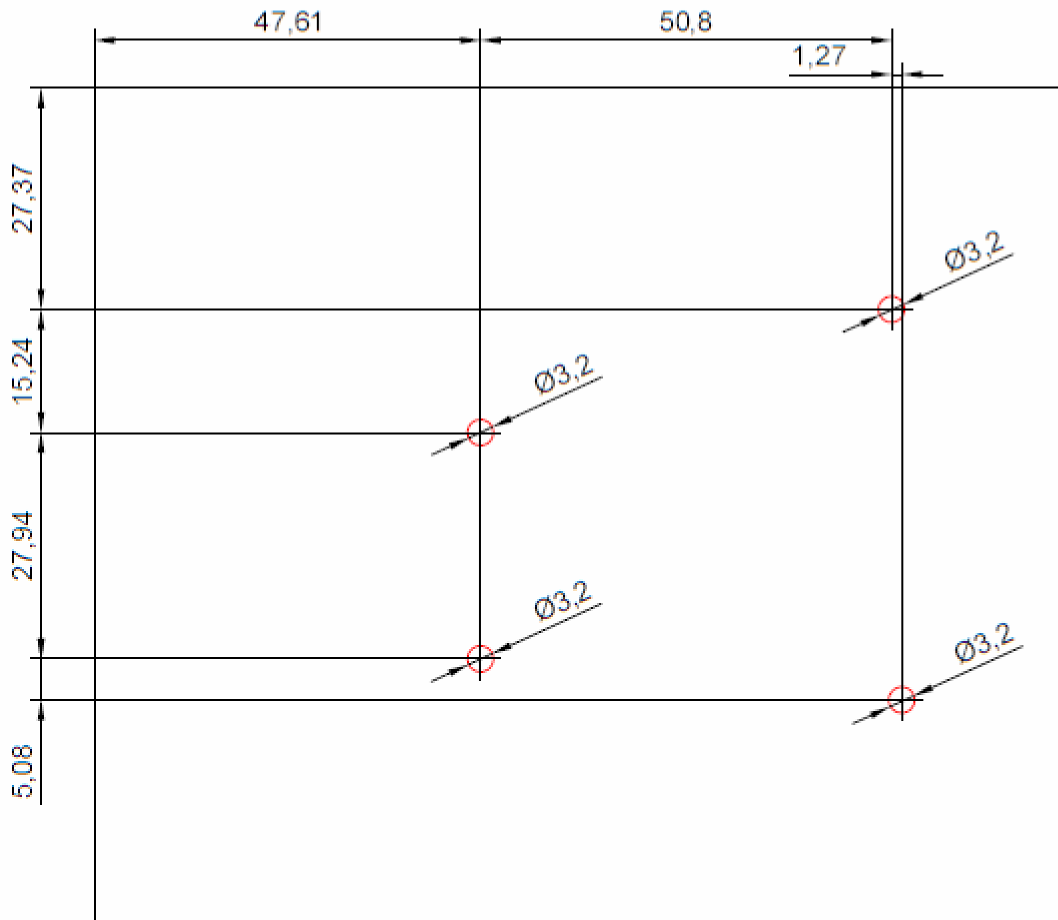
D-sub Pin	**Beskrivning**
1	DR (”wake-up”)
2	CAN-L
3	NC
4	NC
5	NC
6	NC
7	CAN-H
8	NC
9	B+ Spänningsmatning

BILAGA 2 – Kopplingsschema

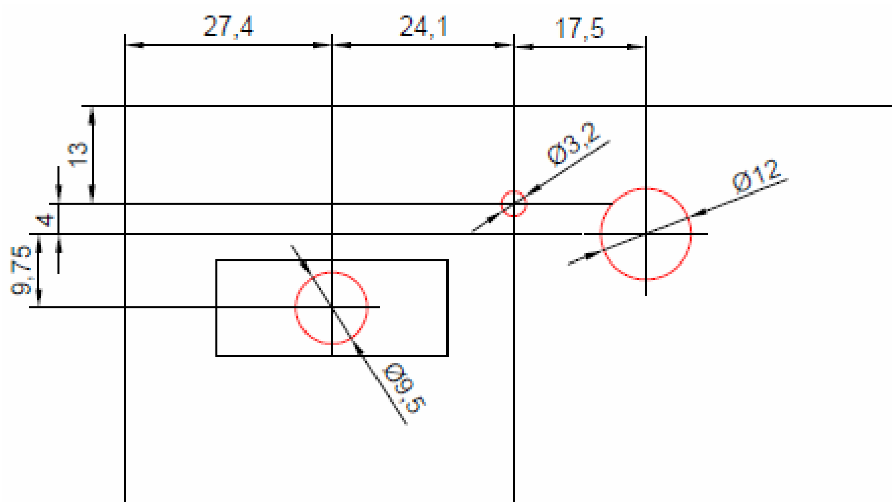


Figur 16. Kopplingsschema över spänningsregulatorer samt bcd-omkopplare.

BILAGA 3 – Mekaniska ritningar



Bottenvy



Sidovy

BILAGA 4 – Produkt/material lista

Produkt/material	**Antal**	**Leverantör**	**Art.nr.**	**Pris/st**
Arduino Uno	1 st	ELFA	11038919	185 kr
Arduino Motorsköld R3	1 st	ELFA	11038926	185 kr
Arduino CAN-BUS Shield V1.2 (Seeduino)	1 st	Lawicel	113030021	259 kr
Profillåda Metall/matt 120x103x53mm	1 st	ELFA	150-43-856	136 kr
PCB kretskort 100x40 mm	1 st			
BCD-omkopplare PT65-703	1 st			
Resistor 10k	2 st			
Kondensator 1 uF	3 st			
Kondensator 0.1 uF	3 st			
Spänningsregulator LM78012	1 st			
Spänningsregulator LM7808	1 st			
Spänningsregulator LM7805	1 st			
Lemo-kontaktdon 6 pol	1 st			
D-SUB-kontaktdon hona PCB 9 pol	1 st			
Skruv M2.5x12 inkl. mutter och bricka	4 st			
Skruv M3x9 inkl. mutter och bricka	1 st			
Skruv för d-sub kontakt	2 st			

BILAGA 5 – Källkod

```
#include <SPI.h>
#include "mcp_can.h"
#include <avr/wdt.h>
//watchdogtimer approx 8 seconds
/*
------------------------GLOBAL VARIABLES------------------------
*/
unsigned char len = 0;
unsigned char buf[8];

const int SPI_CS_PIN = 9; //CAN PIN
const int dirPin = 7; //direction of the motor channel B
const int brakePin = 8; // brake pin for the motor channel B
const int pwmPinMotor = 6; // pwm pin for the motor channel B;
const int sensorPinA4 = A4; // Valvefeedback potentiometer pin.
const int drPin = 5; // "wake-up" signal
const int ID_Pin_1 = 3;
const int ID_Pin_2 = 4;

float ValvePos = 0;
float valvePosVoltage = 0;
float ToCANValvePos = 0;
float valveFeedback = 0; //the valve opening 0-255. 0 = 0%,  255 = 100%
float readValve = 0;
float pwmOut = 0; // The output signal (0-255. where 0 is 0% dutycycle and 255 is 100%
dutycycle)
float setpoint = 0; // the setpoint of the valveopening. recieved from CAN-bus
float errorSignal = 0;
float P_term = 0;
float I_term = 0;
float k_p = 0;
float k_i = 0;
float ValveSenseMin = 0;
float ValveSenseMax = 0;
float UpperLimit = 0;
float LowerLimit = 0;
boolean dr = 0; //if 0 no CAN communication and valveposition = 0%, if 1 enable can comm.
boolean drFlag = 0;
unsigned long int last_time = 0;
unsigned long int last_time_CAN = 0;
unsigned long int last_time_heartbeat = 0;
boolean laststate = 0;
byte heartbeat = 0xFF;
boolean a = 0;
boolean b = 0;

INT32U CANID_TX_PosValve = 0;
INT32U CANID_RX_SetValvePos = 0;
```

```
/*
------------------------GLOBAL VARIABLES------------------------
*/

/*
------------------------FUNCTIONS------------------------
*/
MCP_CAN CAN(SPI_CS_PIN);

void PIcalc(float K_P, float K_I); //Calculates proportional(P) and integral(I) terms.
void CANsendInterval(int sendInterval); //send canmsg with a specific interval.
void MotorStopOrChangeDirection(); //automatically changes direction or stops the valve-
motor depending on the error signal.
void DetectUpperAndLowerLimit();//forces the valve in both end positions where it sets
upper and lower limit.
float SetUpperLimit(float upperlimit); //allows one to change the upper limit (0-100%)
float SetLowerLimit(float lowerlimit); //allows one to change the lower limit (0-100%)
void setCANID(); //sets a specific CAN-id for the controller.
/*
------------------------FUNCTIONS------------------------
*/

void setup() //SETUP runs only 1 time
{
  Serial.begin(9600);
  watchdogSetup();

  START_INIT:

  if(CAN_OK == CAN.begin(CAN_500KBPS)) // init can bus : baudrate = 500kbps
  {
     Serial.println("CAN BUS Shield init ok!");
  }
  else
  {
     Serial.println("CAN BUS Shield init fail");
     Serial.println("Init CAN BUS Shield again");
     delay(100);
     goto START_INIT;
  }

  CAN.init_Mask(0, 1, 0xfffffff);// there are 2 mask in mcp2515, you need to set both of them
  CAN.init_Mask(1, 1, 0xfffffff);
  pinMode(ID_Pin_1,INPUT);
  pinMode(ID_Pin_2, INPUT);
  setCANID();
  pinMode(drPin, INPUT);
  pinMode(sensorPinA4, INPUT);
  pinMode(dirPin, OUTPUT);
  pinMode(brakePin, OUTPUT);
```

```
  pinMode(pwmPinMotor, OUTPUT);
  digitalWrite(brakePin, LOW); //disable brake
  DetectUpperAndLowerLimit();
}


void loop() //Main loop
{
  wdt_reset(); //reset watchdogtimer
  dr = debounce(digitalRead(drPin), 30); //read dr ("wake-up") signal
  if(dr == 0) // if "wake-up" signal is ZERO go to 0% valve opening
  {
    setpoint = 0;
    drFlag = 1;
  }
  if(drFlag == 1 && dr == 1) //when switching dr from 0 to 1 do autocalibrate
  {
    DetectUpperAndLowerLimit();
    drFlag = 0;
  }

  if(CAN_MSGAVAIL == CAN.checkReceive() && dr == 1) // check for data on CANBUS
  {
    CAN.readMsgBuf(&len, buf); // length of message = len, data = buf

    INT32U CANID = CAN.getCanId();

    if(CANID_RX_SetValvePos == CANID)
    {
      setpoint = buf[0] | (buf[1] << 8);
    }
  }

  readValve = analogRead(sensorPinA4); //read from valve feedback potentiometer
  ToCANValvePos = map(readValve, LowerLimit, UpperLimit, 0, 255);
  ValvePos = map(readValve, LowerLimit, UpperLimit, 0, 100);
  valveFeedback = map(readValve, LowerLimit, UpperLimit, 0, 1023);

  errorSignal = setpoint - valveFeedback; //calculate the error
  MotorStopOrChangeDirection();
  if(errorSignal != 0)
  {
    if(abs(errorSignal) < 100) //when close to setpoint use appropiate parameters
    {
      k_i = 0.5;
      k_p = 0.2;
    }
    else //go full speed when far away from setpoint
    {
      k_i = 0;
```

```
      k_p = 10;
    }
    PIcalc(k_p, k_i); //calculate the output.
  }

  analogWrite(pwmPinMotor, pwmOut);

  if(dr==1)
  {
     CANsendInterval(10);
  }
}

void CANsendInterval(int sendInterval)
{
  unsigned long int current_time = millis();
  long int time_change_CAN = current_time - last_time_CAN;
  long int time_change_heartbeat = current_time - last_time_heartbeat;
  if(time_change_heartbeat >= 500)
  {
    heartbeat = ~heartbeat;
    last_time_heartbeat = current_time;
  }

  if(time_change_CAN >= sendInterval)
  {
    if(ToCANValvePos > 255){ToCANValvePos=255;}
    else if(ToCANValvePos < 0){ToCANValvePos=0;}

    unsigned char canMSG[8] = {ToCANValvePos,(int(readValve) & 255), (int(readValve)
>> 8), heartbeat, 0, 0, 0, 0};
    CAN.sendMsgBuf(CANID_TX_PosValve, 1, 4, canMSG); //CAN.sendMsgBuf(INT8U
id, INT8U ext, INT8U len, data_buf)
    last_time_CAN = current_time;
  }
}


boolean debounce(boolean state, int debouncedelay)
{
  if(state != laststate)
  {
    delay(debouncedelay);
    state = digitalRead(drPin);
  }
  laststate = state;
  return state;
}

void PIcalc(float K_P, float K_I)
```

```
{
  float sampletime = 100; // sample time in ms
  unsigned long int current_time = millis();
  long int time_change = current_time - last_time;

  if(time_change >= sampletime) //sample every 100ms
  {
    I_term += abs((errorSignal*K_I))*(sampletime/1000);//calculate I term
    P_term = abs(errorSignal)*K_P; //calculate P term
    if(I_term >= 135){I_term = 135;}//integration limit

    pwmOut = 120 + P_term + I_term;
    if(pwmOut > 255){pwmOut = 255;}

    last_time = current_time;
  }
}

void MotorStopOrChangeDirection()
{
  if(errorSignal > 0) //change direction
  {
    digitalWrite(dirPin, LOW);
  }
  else if(errorSignal < 0 )//change direction
  {
    digitalWrite(dirPin, HIGH);
  }

  if(errorSignal <= 3 && errorSignal >= -3 ) //when error signal is between this boundry stop
motor
  {
    errorSignal = 0;
    I_term = 0;
    pwmOut = 0;
  }
}

void DetectUpperAndLowerLimit()
{
  int readValve1 = 0;
  int readValve2 = 0;
  int countCal = 0;
  //----------DETECT UPPER LIMIT----------
  digitalWrite(dirPin, LOW); //apply the direction to detect upper limit
  analogWrite(pwmPinMotor, 255);//start motor (towards upper limit)
  while(countCal < 2)
  {

    readValve1 = analogRead(sensorPinA4);//read valve feedback potentiometer
```

```
    delay(200); // wait 200ms
    readValve2 = analogRead(sensorPinA4); //read valve feedback potentiometer AGAIN
    delay(200);// wait 200ms
    if(readValve1 == readValve2) //if these are equal the motor is at endposition
    {
      countCal++;
    }

  }
  wdt_reset();
  analogWrite(pwmPinMotor, 0); //stop motor
  delay(100);
  ValveSenseMax =  analogRead(sensorPinA4); //read max valve pos

  //----------DETECT LOWER LIMIT----------
  digitalWrite(dirPin, HIGH); //change direction to detect lower limit
  analogWrite(pwmPinMotor, 255); //start motor (towards lower limit)
  countCal = 0;

  while(countCal < 2)
  {
    readValve1 = analogRead(sensorPinA4);
    delay(200);
    readValve2 = analogRead(sensorPinA4);
    delay(200);
    if(readValve1 == readValve2)
    {
      countCal++;
    }
  }
  wdt_reset();
  analogWrite(pwmPinMotor, 0); //stop motor
  delay(100);
  ValveSenseMin = analogRead(sensorPinA4); //read min valve pos

  UpperLimit = SetUpperLimit(97);
  LowerLimit = SetUpperLimit(3);
}

float SetUpperLimit(float upperlimit)
{
  return (ValveSenseMax - (ValveSenseMax-ValveSenseMin)*((100-upperlimit)/100));
}

float SetLowerLimit(float lowerlimit)
{
  return (ValveSenseMin + (ValveSenseMax-ValveSenseMin)*(lowerlimit/100));
}

void setCANID()
```

```
{
 a = digitalRead(ID_Pin_1);
 b = digitalRead(ID_Pin_2);
 if(a == 0 && b == 0)
 {
  CANID_TX_PosValve = 0x19000011;
  CANID_RX_SetValvePos = 0x19000012;
  CAN.init_Filt(0, 1, 0x19000012);
 }
 else if(a == 0 && b == 1)
 {
  CANID_TX_PosValve = 0x19000021;
  CANID_RX_SetValvePos = 0x19000022;
  CAN.init_Filt(0, 1, 0x19000022);
 }
 else if(a == 1 && b == 0)
 {
  CANID_TX_PosValve = 0x19000031;
  CANID_RX_SetValvePos = 0x19000032;
  CAN.init_Filt(0, 1, 0x19000032);
 }
 else if(a == 1 && b == 1)
 {
  CANID_TX_PosValve = 0x19000041;
  CANID_RX_SetValvePos = 0x19000042;
  CAN.init_Filt(0, 1, 0x19000042);
 }
}

void watchdogSetup(void)
{
cli();  // disable all interrupts
wdt_reset(); // reset the WDT timer
/*
 WDTCSR configuration:
 WDIE = 0: Interrupt Disable
 WDE = 1 :Reset Enable
 WDP3 = 1 :For 8000ms Time-out
 WDP2 = 0 :For 8000ms Time-out
 WDP1 = 0 :For 8000ms Time-out
 WDP0 = 1 :For 8000ms Time-out
*/
//Watchdog settings:
WDTCSR |= (1<<WDCE) | (1<<WDE);

 WDTCSR = (0<<WDIE) | (1<<WDE) | (1<<WDP3) | (0<<WDP2) | (0<<WDP1) |
(1<<WDP0);
sei();
}
```