Drivsystem för ett lätt eldrivet fordon

Jonas Pettersson

Arcada – Nylands svenska yrkeshögskola
Elektroteknik

Helsingfors 2009

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Elektroteknik
Identifikationsnummer:	2318
Författare:	Jonas Pettersson
Arbetets namn:	Drivsystem för ett lätt eldrivet fordon
Handledare:	Rene' Herrmann
Uppdragsgivare:	Arcada, Mikael Paronen

Sammandrag:

Detta slutarbete handlar om planeringen av ett drivsystem för ett lätt eldrivet fordon.

Drivsystemet innehåller ett ackumulatorpaket, en trefas frekvensomvandlare och en trefas permanentmagnetiserad synkronmotor. Det drivande hjulet är direktdrivet alltså är det frågan om en navmotor.

Ackumulatorpaketet som har valts är på 48V och består av 4 st slutna blyackumulatorer. Kapaciteten är i denna prototyp vald till 9Ah av säkerhetsskäl.

Frekvensomvandlaren bygger på mosfet teknik och är planerad för höga strömmar och låga spänningar. DsPIC30f2010 används som styrlogik i frekvensomvandlaren. Mikrokontrollern använder SVM för att matematiskt modelera trefasspänningen och spänningen moduleras till elmotorn genom PWM.

Frekvensomvandlaren är optimerad för hög effektivitet. Transistorerna är överdimensionerade för effekten, detta minskar på ledningsförlusterna. Även kopplingstiden för transistorerna är så liten som teknisk möjligt. Kort kopplingstid medför dock problem med överspänningar som måste filtreras bort.

Mätning för förbrukad effekt är utförd i arabiastrandens parkeringshall, resultaten visar att det krävs en motor med nominaleffekten på ca 1kW för att uppnå 40km/h utan att överbelasta motorn.

Effektiviteten för frekvensomvandlaren är beräknad till 99,59 vid ca 1kW belastning. Effektivitetsmätningar är ej utförda som skulle bevisa beräkningarna.

Drivsystemet använder sig av regenerativ bromsning för att ta tillvara kinetisk energi vid inbromsningar med motorn.

Nyckelord:	SVM, Frekvensomvandlare, Navmotor, PMAC, Regenerativ bromsning.
Sidantal:	43
Språk:	Svenska
Datum för godkännande:	

DEGREE THESIS	
Arcada	
Degree Programme:	Electrotechnics
Identification number:	2318
Author:	Jonas Pettersson
Title:	Drive unit for a light electrically powered vehicle
Supervisor:	Rene' Herrmann
Commissioned by:	Arcada, Mikael Paronen

Abstract:

This thesis is about the planning of a drive system for a lightweight electric vehicle.

The drive system includes a batterypack, a three-phase frequency converter and a three-phase permanent magnet synchronous motor. The driving wheel is directly driven, therefore the motor is of hubmotor type.

The batterypack which has been chosen to 48V and is made up of 4 pieces of sealed lead-acid batteries. The capacity of the batterypack of this prototype is selected to 9Ah for security reasons.

The frequency converter is based on MOSFET technology and is designed for high currents and low voltages. DsPIC30f2010 is used as control logic in the frequency converter. The microcontroller uses SVM to mathematically handle the sinusoidal three phase voltage. The voltage is modulated to the electric motor through PWM.

The frequency converter is optimized for high efficiency. Transistors are oversized for the calculated effect, thus reducing the conduction losses. Switching time of the transistors is as small as technically possible. Short switching time though pose a problem with surges that must be filtered out.

Measurement of consumed power is carried out in the Arabianranta parking hall, the results demonstrate the need for an motor with nominal power of about 1kW to reach 40km / h without overloading the engine.

The effectiveness of the frequency converter is estimated at 99.59 at about 1kW load. Efficiency measurements are not performed that would prove the calculations.

The drive system uses regenerative braking to charge the batterypack.

Keywords:	SVM, Frequencyconverter, Hubmotor, PMAC, regenerative braking.
Number of pages:	43
Language:	Swedish
Date of acceptance:	

FÖRORD

Jag vill tacka Ville Schütt som har hjälpt mig genom att låta mig installera ett drivsystem på hans moped samt gett värdefull feedback från testkörningar samt agerat testförare för testfordon. Jag hoppas innerligt att drivsystemet jag har utvecklat kan vidareutvecklas vid Arcada och kommer till användning.

Jonas Pettersson 11.11.2009

Innehållsförteckning

BEGREPP OCH TERMER SOM ANVÄNDS

Jag använder i min text en del speciella termer och begrepp som kanske inte alla är bekanta med.

- FET Field Effect Transistor, Fälteffekttransistor.
  En transistor som styrs genom att man laddar upp transistorns grind. Laddningen resulterar i ett fält som ger upphov till en ledande kanal i transistorn där ström kan flyta. N kanal transistorer leder då de har en positiv laddning, P kanal transistorer leder då de har en negativ laddning. Till fälteffekttransistorernas egenskaper hör: snabb koppling, låg ledningsresistans, hög strömtålighet men liten spänningstålighet.

- MOSFET Metallized Oxide on Silicon Field Effect Transistor
  En typ av fälteffekttransistor.

- PWM Pulse Width Modulation
  Ett modulationsätt med vilket man kan uppnå en energireglering. Tex 50% på av full tid ger teoretiskt halv effekt.

- IGBT Isolated Gate Bipolar Transistor
  En hybrid mellan bipolär transistor och fälteffekttransistor. Till egenskaperna hör: hög kopplingstid (långsam), hög spänningstålighet och effekttålighet.

- SVM Space Vector Modulation
  Ett modulationssätt som inte betraktar 3-fas spänningarna som enskilda objekt utan innehåller alla faser i ett koordinatsystem. Man kan genom matematiska formler bestämma påtiden för två faser och en ”nollvektor” för en fas. Då en elektrisk cykel kan ses som 360º kan man för godtycklig punkt räkna ut tiderna samt sedan skala ner resultatet med önskad spänningsamplitud. Denna resultatid för de olika faserna kan sedan moduleras över motorlindningarna med PWM. Den kombinerade tekniken kallas för SVPWM i facklitteratur.

1 INTRODUKTION

Enligt finsk lag är en moped ett fordon som har en maximal hastighet av 45km/h och om mopeden är el driven får motoreffekt vara maximalt 4kW. Om någon av dessa begränsning överskrids är fordonet en motorcykel.

Mopeder och motorcyklar som drivs med el motor har funnits länge men har nu på de sista 2-3 åren börjat slå igenom då samhället börjar visa större mån om miljön och det finns en vilja att minska på koldioxid utsläppen.

Genom den senaste tidens framsteg inom ackumulatortekniken har ackumulatorernas kapacitet ökats och vikten minskats. Detta medför att man nu kan lagra den effekt i ackumulatorer som krävs för att en moped eller motorcykel skall kunna vara användbar som färdmedel.

Detta slutarbete omfattar val av ackumulatorteknik samt motorteknik för ett lätt eldrivet fordon för att uppnå en så hög verkningsgrad som möjligt för drivsystemet.

Kommersiellt finns det på marknaden 3-5 tillverkare som är har en eldriven moped som är väglaglig i Finland. Dessa tillverkare använder sig av permanentmagnetiserade borstlösa likströmsmotorer eller likströmsmotorer med borstar.

Orsaken till att få utvecklat ett eget drivsystem är att få så hög effektivitet som möjligt och att få drivsystemet fullt programmerbart och möjliggöra en större grad av integrering i det eldrivna fordonets övriga system.

Drivsystem som utvecklas i detta slutarbete kunde jämföras med det som hittas i de versionerna av eldrivna mopeder som har en permanentmagnetiserad synkronmotor.

2 ÖVERSIKT AV DRIVSYSTEMET

Figur 1: Blockshema för drivsytemet.

I ovanstående figur 1 presenteras ett principiellt blockschema över drivsystemet innehållande en ackumulator som lagrar energi, en frekvensomvandlare som omvandlar spänningen från ackumulatorn till en form som kan matas i motorn och motorn omvandlar el energin till kinetisk energi. Drivsystemet skall även ha egenskapen att kunna ta till vara kinetisk energi då det eldrivna fordonet saktar in.

Dimensioneringen för drivsystemet är gjort för en moped med förare som kan antas få en totalmassa på 140 kg, mopeden 60-70kg och föraren 70-80kg. Hastigheten är antagen att vara i området för en moped som är 0-45km/h.

I alla omvandlingar sker det förluster och målet är att minimera dessa förluster så långt som det är ekonomiskt och tekniskt lönsamt. Detta innebär att effektiviteten skall vara hög på motorn och frekvensomvandlaren samt att ackumulatorerna skall kunna avge effekten utan större interna förluster. Dessutom måste kabeldragningar vid förverkligande vara dimensionerade så att stora överföringsförluster inte sker.

3 ACKUMULATORER

Ackumulatorerna spelar en stor roll för vilken prestanda ett eldrivet fordon har. Räckvidden och vikten är beroende av ackumulatorerna. Ackumulatorerna inverkar även på köregenskaperna för fordonet, placeras ackumulatorerna rätt blir fordonet stabilt.

Ackumulatorerna har även stor betydelse för servicekostnaden för ett eldrivet fordon då ackumulatorerna är i praktiken den enda delen som måste bytas ut o de har blivit slitna. I ett ungefär tål ackumulatorerna 2000-3000 upp- och urladdningar oberoende typ.

3.1 ACKUMULATOR TYPER

- NiMH

Nickel Metall Hydrid ackumulatorerna används i applikationer där viktigaste faktorerna är hög ström jämfört med laddningskapaciteten och drift i ett stort område av temperaturer. Framförallt största delen av vanliga laddningsbara batterier tillhör denna kategori. NiMH används också i bärbara el-verktyg. Även så använder Toyota NiMH ackumulatorer i sin hybridbil Prius.

- Lithium

Lithium teknologin gör stora framsteg framförallt genom att det utförs undersökning i lithium ackumulatorteknologin mest pga. mobiltelefonernas utveckling. Lithium används mest i applikationer som kräver låg ström en längre tid. Det finns även olika variationer av litium ackumulatorer som har olika egenskaper. Men de mesta variationer måste ha elektroniska skydd så att överladdning, överhettning eller total urladdning inte sker.

- Bly

Blyackumulatorer har funnits länge och även här har framsteg gjorts med tanke på kapacitet och hållbarhet. För applikationer som kräver hög ström och stryktålighet är blyackumulatorer nästan den ända praktiskt och ekonomiskt lönsamma ackumulatortyp.

3.2 PRESTANDA FÖR OLIKA ACKUMULATOR TYPER

Nedan finns presenterat en tabell med viktigaste egenskaperna för denna applikation för olika ackumulatortyper.

Tabell 1: Ackumulatordata.

Ackumulatortyp	NiMH	Bly	Li
Gravimetrisk energi kapacitet (Wh/kg)	60-120	30-50	100-190
Livslängd, urladdningar till 80%	300-500	200-300	300-1500
Urladdningsström Topp	5 x Kap.	5 x Kap.	15-20 x Kap.
Optimal	0.5 x Kap.	0.2 X kap.	10 x Kap.
Överladdningstålighet	Låg	Hög	Obefintlig

Värden är tagna ur Yuasas datablad (Yuasa Battery Inc. 2009) och från batteryuniversity.com (Buchman, Isidor. 2003).

3.3 VAL AV ACKUMULATORER

Inbromsningen är ett problem då den matar ström tillbaka till ackumulatorerna och beroende på hastigheten kan det vara en ganska stor strömpuls på en relativt kort tid.

Med blyackumulatorer innebär detta att inte optimal laddning uppnås och en del av energin går förlorad i ackumulatorerna då de hettas upp.

Med litium ackumulatorer så kan en stor strömpuls betyda att en cell kan bli överladdad fast ett övervakningssystem försöker balansera laddningen över alla celler. Om en litium cell överladdas kan den i värsta fall avge vätgas som kan fatta eld då litium cellen redan är överhettad i det fallet. Slutresultatet är ofta att överladdade litiumceller fattar eld och brinner explosionsartat.

Till detta projekt har jag valt att använda blyackumulatorer då de är billiga och tål att urladdas och laddas ganska fritt. Slutna blyackumulatorer är även ganska säkra att använda då syran inte lätt kan läcka ut. Det bästa med anseende på vikten skulle vara att använda litium ackumulatorer då de har högsta energitätheten, men då måste man ha ett övervakningssystem som ser till att ackumulator cellerna inte överhettas.

Kapaciteten för ackumulatorerna bör vara så att man kan köra med dimensionerad effekt i 1 timme med 40km/h hastighet. Här bör ses över det aktuella elfordonets bärförmåga och hur stort utrymme kan användas för ackumulatorer.

Ackumulatorernas kapacitet anges ofta i 20h urladdning så då är en 20Ah ackumulator dess specificerade kapacitet vid en urladdning på 1A. Om man urladdar en ackumulator snabbare sjunker dess kapacitet avsevärt. Ofta meddelar tillverkare även olika urladdningskurvor för sina ackumulatorer där man kan söka upp rätt storleks ackumulator.

Blyackumulatorer finns i olika versioner för olika ändamål. För elfordonsbruk är ackumulatorer som är avsedda för cykliskt bruk och lämpliga för djupurladdning de som bör väljas.

I testfordonet används slutna ackumulatorer så att möjliga syraspill elimineras. Kapaciteten på dessa ackumulatorer är 7,5Ah(20h). Det kunde vara större ackumulatorer men dessa blev valda för att de har en relativt låg kortslutningsström som är att föredra då det är frågan om en prototyp där saker kan och kommer att gå fel.

4 MOTOR

4.1 MOTORTYP

Jag valde att använda en permanentmagnetiserad 3-fas synkronmotor. Denna motortyp har hög effektivitet, i normala fall 80% men upptill 90% är möjligt vid optimala förhållanden. Även en faktor som bidrog till valet var att alla navmotortillverkare har börjar flytta över till permanentmagnetiserade navmotorer.

Motorn har inte en rotor som roterar inuti en stator som normala elmotorer har, utan en rotor som roterar runt en stator. Detta innebär att statorn sitter på axeln och rotorn är skalet av motorn. Detta möjliggör att motorn är monterad inne i en fälg som eliminerar de förluster som sker i kraftöverföringen mellan motorn och drivande hjul. Denna typ av motorn kallas allmänt för navmotor.

Den permanentmagnetiserade motorn kräver en 3-fas frekvensomvandlare som kör motorn. Då motelektromotoriska kraften är sinusformad får man bästa effektivitet genom att använda sinusformad matning till motorn.

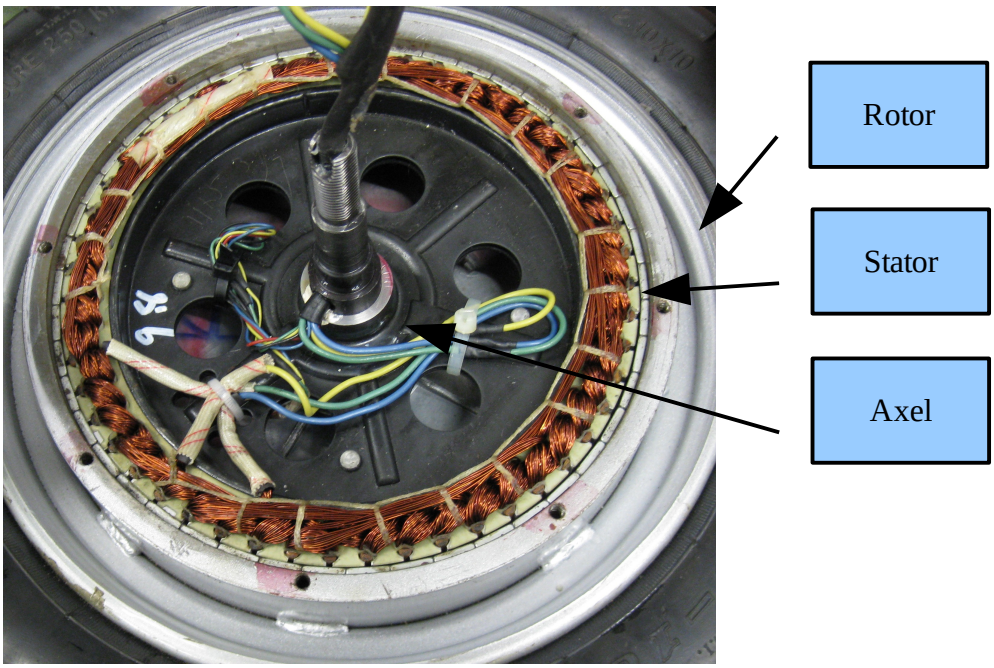


Bild 1: Navmotorns uppbyggnad.

På bild 1 ser man hur motorn är uppbyggd med statorlindningarna på axeln och permanentmagneter i fälgen runtom. Motorn har en nominell effekt på 500W.

Räknar man ut varvtalet med synkrona varvtalets formel som lyder n_s=120 *f/p [1] (Alfredsson, alf et al. s. 84) där f är frekvensen för matningen och p är antalet poler. Som 48 polig motor vid 50Hz blir synkrona varvtalet 115 varv/min.

Hjulet har en fälg som är 10 tum i diameter samt däcket som är 3 tum. Detta ger en effektiv diameter på ca 40cm vilket resulterar i en omkrets på ca 127 cm. Skall man med denna omkrets uppnå en hastighet på 45km/h krävs ett varvtal på ca 610 varv per minut. Detta betyder att frekvensomvandlaren måste klara av att ge ut en frekvens som motsvarar nominella varvtalet 610 varv per minut. Skriver man på formel [1] så att man löser ut frekvensen ur formeln lyder formeln som följande f= (n_s*p)/120. Då man matar in i denna formel att poltalet är 48 får man till resultat att maxfrekvensen blir 244 Hz för sinusspänningen vid maximal hastighet.

4.1.1 ÖVERSYNKRON BROMSNING

Då motorn är en synkronmotor kan man använda sig av översynkron bromsning. Detta innebär att om rotorn drivs av yttre krafter till en högre hastighet än synkrona varvtalet för dess matning, kommer motorn att köras som en generator och mata effekt tillbaka i ackumulatorerna.

Översynkron bromsning sker på grund av att fasförskjutningen mellan rotor och det roterande magnetfältet blir negativt i jämförelse med fasförskjutningen vid motordrift som är positivt. En negativ fasförskjutning ger upphov till en högre spänning i statorlindningarna och effekt matas tillbaka till drivande systemet då motorns mot elektromotoriskakraft är större än drivsystemets driftspänning. Detta kan även bevisas matematiskt med formeln.

$$M=k * B_{res} * B_r * \sin \gamma \qquad [2]$$

Där M=axelmoment, k=en konstant för maskinen i fråga, B_{res}=statorflödets flödestäthet, B_r=rotorflödets flödestäthet och γ = fasförskjutningsvinkeln mellan stator och rotorflödet.

Denna formel [2] gäller i motordrift men också i generatordrift för synkronmotorer. Vid motordrift är γ vinkeln positiv men i generatordrift är den negativ, då alla andra variabler är oförändrade blir resultatmomentet negativt vid generatordrift. Då effekten som motorn konsumerar är direkt proportionerlig till momentet blir också effekten negativ då momentet blir negativt. (Alfredsson, alf et al. s. 86, 90-91)

4.2 ROTORPOSITIONSBESTÄMNING

För synkrona motorer som är elektroniskt kommuterade är det viktigt att veta rotorns postition i förhålland till statorn så att man kan magnetisera statorn rätt. I den motorn jag har valt att jobba med så är detta gjort så att det i statorn finns 3st inbyggda hall-sensorer. Med hjälp information man får från hall sensorerna kan man magnetisera motorn rätt vid start och vid varierande lastförhållanden. Då motorn till funktionsprincip är en synkronmotor så roterar rotorn med samma hastighet som magnetfältet. Hall-sensorerna används alltså för att bestämma var rotorn befinner sej mekaniskt och för att räkna ut frekvensen som skall matas till motorn så inte rotorn tappar låsningen till fältet.

5 FREKVENSOMVANDLAREN

Frekvensomvandlaren är till sin funktionsprincip som en vanlig trefas frekvensomvandlare. Men i detta fall för låga spänningar och höga strömmar.

I vanliga frekvensomvandlare används IGBT kraftelektronik komponenter men i denna version används MOSFET komponenter som ger en snabbare koppling och högre strömtålighet.

Frekvensomvandlaren består fysiskt av en spänningsmodulerare som körs av en drivkrets. Drivkretsen får sin styrning från en mikrokontroller. Programmet i mikrokontrollern använder sig av SVM för att få en matematisk modell för trefassystemet och omvandlar sedan det beräknade riktningen och storleken för varje fas till en PWM modulerad styrsignal anpassad för spänningsmodulatorn.

Strömbegränsning i drivsystemets första version är implementerad i formen av en enkel säkring som kapar matningen till drivsystemet om motorn konsumerar en för stor ström. Säkringen måste ha en diod som tillåter ström flyta tillbaka till ackumulatorerna från motorn då drivsystemet vid inbromsning annars förstörs då energin som återvinns får en hög spänning som övergår transistorernas spänningstålighet.

5.1 DIMENSIONERING

Effekten som krävs för att flytta på ett fordon består av olika delar. Effekten måste övervinna följande storheter: luftmotstånd, rullningsmotstånd, effekten som krävs för acceleration och effekten som krävs för att bestiga backar. För att köra på ett plant underlag så är de största krafterna man måste övervinna rullningsmotståndet och luftmotståndet. Om det inte finns speciella krav på accelerationstider och hastighet för bestigning av backar kan man räkna ut vad tiden blir för acceleration om man bara har en motor som är dimensionerad för att köra på plan mark. I andra fall stiger effektkraven betydligt om man vill bestiga backar med samma hastighet som man kör på plan väg.

Ekv.2: $P_{lms} = 0.5 * C_d * A_f * \rho * v^3$ [3]

Där C_d = Koefficient för luftmotstånd (1,8 enhetslös), A_f = Frontalarea av fordon (0,6 m²), ρ = Densiteten av luften (1,22 kg/m³) och v = hastigheten(m/s)

Ekv.3: $P_{rull} = C_r * m * g * v$ [4]

Där C_r = Koefficient för rullmotstånd, m = massan (kg), g = gravitationskonstanten 9,81(m/s²) och v = hastigheten(m/s)

Ekv.4: $P_{backe} = m * g * v * (B_{\%}/100)$ [5]

Där m = massan (kg), g = gravitationskonstanten 9,81(m/s²), v = hastigheten(m/s) och $B_{\%}$= backen beskriven i % (100*höjning(m)/plana sträckan för höjningen(m))

Ekv.5: $P_{acc} = m * a * v_{avg}$ [6]

Där m = massan (kg), a = acceleration $\Delta v/\Delta t$ (m/s²) och v_{avg} = Medelhastighet vid accelerationen ($0.5*(v_1+v_0)$)

Jag har baserat mina uträkningar på att fordonet väger 60kg och föraren 80kg som kan anses rimligt. Jag har räknat fordonets silhuettyta till 0.6 m² och anser detta rimligt för en moped samt förare. Nedan finns det ett diagram där effektbehovet presenteras (Basic Robotics Inc. 2008).

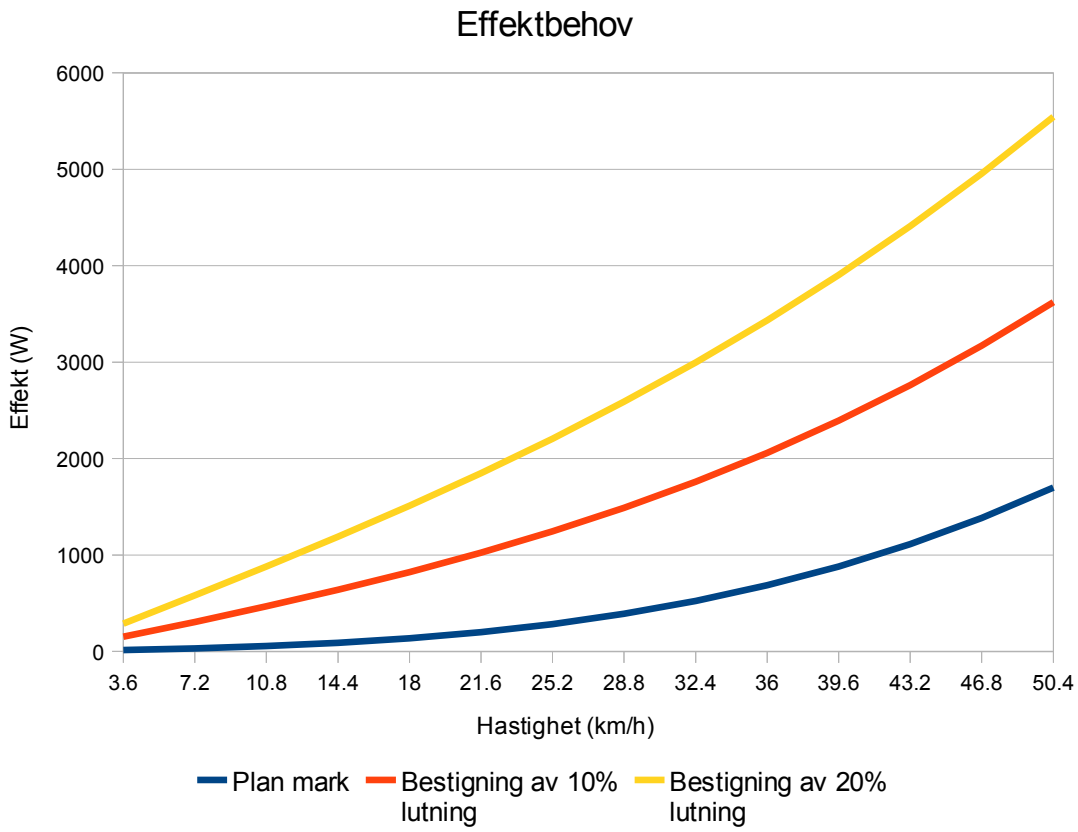


Diagram 1: Krävd effekt per hastighet vid 140kg totalmassa.

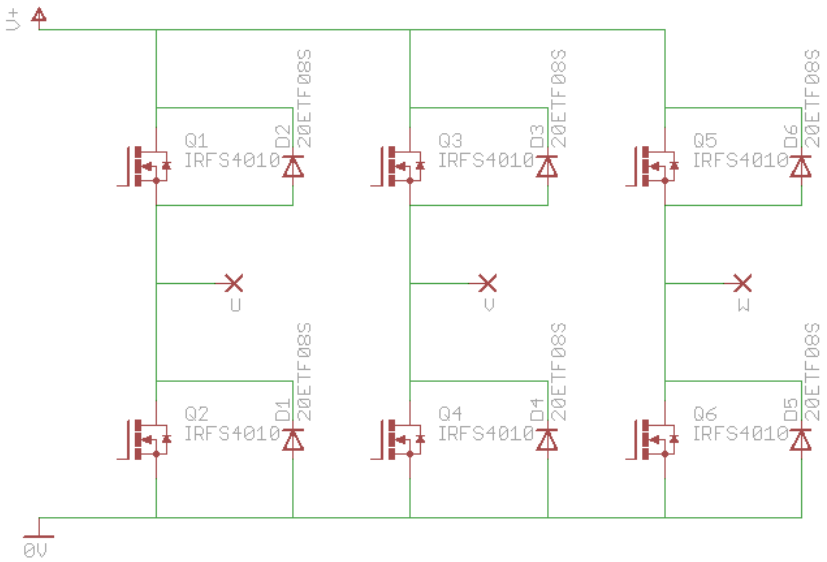
I planeringsskedet utgick jag från att den nominella effekten av drivsystemet skulle vara en kilowatt vilket ger ca 20A kontinuerlig ström genom kretsen. Skall man bestiga en backe med 20% lutning med en kilowatt så uppnår man bara en hastighet på ca 11km/h.

Om man strukturerar om ekv. 6 så att man får reda på tiden istället för krävd effekt blir formeln följande: $\Delta t = (\Delta v * m * (v_1 + v_0))/2 * P_{acc}$ [7]. Jag gjorde en uträkning i excel där jag räknar accelerationen i steg med effekten som kvarstår av 1kW vid aktuella hastighetens effektbehov på plan mark. Med denna metod fick jag tiden för acceleration från 0-40km/h att bli ca 25sec. Har man inte en begränsning på effekten kommer accelerationen att ske snabbare då motorns motelektromotoriska kraft är mindre än drivspänningen, och därmed konsumerar motorn en större effekt än vid en balanserad drift som sker då man kör med konstant hastighet.

5.2 PLANERING AV KRETSSCHEMA OCH LAYOUT AV KRETSKORTET

5.2.1 SPÄNNINGSMODULATORN

Uppbyggnaden av frekvensomvandlarens spänningsmodulator är tagen från Sähkömoottorinkäytön digitaalinen ohjaus (Niiranen Jouko: 1999:48-52). Med tillägg av dioder kopplade parallellt med transistorerna så att man kan skydda transistorerna från tillbakakoppling från motorn och också att man kan likrikta växelspänning som kommer från motorn vid översynkron bromsning.



Figur 2: Principbild för 3 fas växelriktare.

Kretskortlayouten är planerad så att de ytor som leder hög ström är så korta som möjligt för att minska på magnetfält som uppstår vid ledare som leder höga strömmar, strömpulserna kan ge upphov till störningar och icke önskat beteende i vissa delar av frekvensomvandlaren. Även att hålla delar korta som leder en hög ström minskas ledningsförluster som gör att kretskortet hettas upp.

Kretskortets kopparbanor förstärks genom att löda fast koppartrådar och kopparbanor med mindre ström förtennas. Detta för att minska på risken att kopparbanorna bryts eller förångas då det går en större strömpuls igenom dem.

Kraftelektronikkomponenterna löds fast på ett speciellt kretskort som består av en aluminiumskiva och isolerade kopparytor för komponenterna. Då används komponenternas kylyta som ledare och minskar chansen för att något ben skall förångas vid en hög strömpuls. Kretskortet har en koppartjocklek av 35µm men förstärks med 1millimeters kopparplattor och tennlod.

Då komponenterna är ytmonterade så är de svåra om inte nästan omöjliga att byta utan specialverktyg, så kraftdelen kommer att i sin helhet vara utbytbar för att underlätta service.

För kretsen i testfordonet använde jag mig av annan kapsling för transistorerna då jag ville kunna byta ut komponenter utan att demontera och bygga upp hela kretsen pånytt.

5.2.2 STYRDELEN

Grundstrukturen är tagen ur Microchips applikationsbeskrivning AN1017 (IRF:2005). Uppbyggnaden dvs kretskortlayouten för styrdelen är gjord så att kopparlinjer skulle vara så korta som möjligt för att styrelektroniken kan utsättas för elektromagnetisk strålning från ledare som går från kraftelektronikdelen. Detta varierande magnetiska fält kan inducera störningar i kopparlinjer om de är för långa. Till de första testerna har jag använt en annan layout än för den sista versionen då en mera öppen och utspridd layout gör det enklare att byta komponenter som kan gå sönder.

Schematiska bilden av kretsen återfinns i bilaga 1.

5.3 VAL AV KOMPONENTER

5.3.1 MIKROKONTROLLER

Det valdes en mikrokontroller dsPIC30F2010 som är utvecklad för motorstyrning av permanentmagnetiserade trefas-motorer. Den behöver några få komponenter som finns specificerade i dess datablad för att funktionera. Kodningen av programmet sker i C-språk och kan såtillvida användas på olika hårdvaruplattformar då alla mera avancerade mikrokontrollers ofta har

en c-kompilator. Detta möjliggör att den tekniska lösningen kan adapteras till andra tillverkares mikrokontroller och programmet kan användas efter små ändringar pågrund av att hårdvarunamn och adresser ändrar med tillverkarbyte.

Nedan är presenterade de viktigaste egenskaperna.

- Operationshastighet på 20 miljoner instruktioner per sekund, med 5MHz kristall
- 16 bits mikrokontroller, vilket ger större upplösning på variablerna
- DSP egenskaper, vilket betyder att det finns avancerade matematiska funktioner
- Hårdvarufärdighet för UART, SPI och I2C kommunikation
- 6 pulsbreddsmodulerade utgångar som mitt- eller kantjusterade
- ICSP, In Circuit Serial Programming. Som är PIC mikrokontrollernas vanliga programmeringssätt, som också tillåter att programmera kontrollern då den är i kretsen den skall styra.
- Packningstypen är 28 pin SOIC vilken är 10,34 mm * 17.87 mm

5.3.2 FÄLTEFFEKT TRANSISTOR DRIVKRETS

Jag hade tidigare erfarenheter av International Rectifier och fann ur deras produktsortiment en tvåpulsbrygg drivare IR2110 som jag fick varuprov av genom Flinkenberg Oy. En tvåpulsbrygga mosfet grinddrivar IC innehåller drivketsarna för att styra en n-kanal mosfet på jordsidan och driva för att styra en n-kanal mosfet på plus sidan av utgången. Den gör det också möjligt att styra högre spänningar då IR2110 tål 500V. Så kan man genom att byta ut transistorerna jag har valt till denna version, koppla högre spänningar. IR2110 kan även köra IGBT tvåpulsbryggor vilket ger utökat spänningsområde om man byter typen av kraftelektronikkomponenter. Drivkretsarna kräver en del kondensatorer som säkerställer att inte drivkretsen matningsspänning sjunker vid stora strömspikar. Det kan gå relativt stora strömmar genom drivkretsen då transistorerna snabbt styrs av och på.

Den flytande delen av drivkretsen dvs ”highside driver” har även en kondensator som laddas upp då transistorn som kopplar utgången till jord är på. Denna sk. ”bootstrap”-kondensator måste räknas ut. Den är beroende av kopplingsfrekvensen och komponentens gateladdning. I detta fall räknade jag ut komponenten till en lägre kopplingsfrekvenser då man även kan använda den storlekens

kondensatorer för större kopplingssfrekvenser. De är då bara överdimensionerade. Jag har använt mig av följande formel. (Adams, Jonathan, International Rectifier. 2001. s 2-3)

$$C_{bs} \geq [2*(2*Q_g + I_{qbs\,(max)} / f + Q_{ls} + I_{Cbs(leak)} / f)] / V_{cc}-V_f-V_{LS}-V_{min} \qquad [8]$$

Där: Q_g = Gate laddningen av höga sidans transistor = 230 nC (International Rectifier 2008 s. 2), f = kopplingsfrekvens = 2kHz, $I_{qbs\ (max)}$ = gatedriverns strömförbrukning = 230µA (International Rectifier 2005 s. 3), $I_{cbs(leak)}$ = Kondensatorns läckström = 1µA, Q_{ls} = Laddning krävd för nivåskjutning per cykel = 5nC, V_f = lednings-spänningsfallet över ”bootstrap” dioden = 0.7V, V_{LS} = Spänningsfallet över lägre transistorn som bootstrapkondensatorn laddas upp genom = Id*Ron = 20A*0.003Ω = 0,06V, V_{Min} = Minimala spänningen som är tillåten mellan VB och VS (drivspänningen för höga sidan) = 11V, Vcc = drivspänningen för drivkretsen = 15V

$C_{bs} \geq [2*(2*0.000\,000\,230C + 0.000230A / 2\,000 + 0.000\,000\,005C + 0.000\,001A / 2\,000)] /15 – 0.7 – 0.06V – 11V$

$C_{bs} \geq 0.000000358$ F

OBS. värdet som man får för kondensatorn av ovanstående ekvation är det krävda absolut minimivärdet av ”bootstrap” kondensatorn. En för liten kondensator kan orsaka rippelspänningar i drivpänningen till höga sidans driver, samt kan det även uppstå spänningstoppar som kan skada drivkretsen. För att minimera rippelspänning och överladdning av kondensatorn skall resultatet från formeln multipliceras med 15 (riktgivande). (Adams, Jonathan, International Rectifier. 2001 s. 3)

Multipliceras man värdet för C_{bs} med 15 får man till resultat 5,375µF. Jag har använt 10µF i prototypen så att det säkert är tillräckligt, då inte storleken eller priset var ett hinder att använda större kondensator.

5.3.3 TRANSISTORERNA

IRFS4010-7PPbF valdes då den motsvarar transistorn som jag hade i testversionen, men den jag hade tillverkas inte i ytmonterat utförande. Dessutom så håller detta utförande mera ström då den har mera ledaryta genom att komponenten har 6 ben som leder till kollektorn och själva komponenten har emittern i kylytan som kan effektivt utnyttjas på kyllösningen som jag använder. Testversionen hade det problemet att transistorernas ben kunde förångas fast komponentens kärna tålde strömmängden.

5.3.4 ÖVRIGA KOMPONENTER

I denna version tas styrspänningen till logiken från ackumulatorerna mellan andra och tredje ackumulatorn. Detta leder till att styrspänningen hålls relativt stabilt på 24volt. Denna spänning regleras sedan ner till 15volt, 5 volt och 3.3 volt med vanliga spänningsregulatorer.

Kretsen innehåller dock överspänningsskydd och en diod som hindar att strömmen matas tillbaka mot ackumulatorerna. Spänningen sjunker vid en hastig acceleration och om ackumulatorerna är delvis urladdade kan de falla under 15 och då blir spänningsförsörjningen genom regulatorn instabil.

Kondensatorerna i kretsen är överdimensionerade både i kapacitans och spänningstålighet för att öka på livslängden och för att vara säker på att inte spänningstoppar förstör kondensatorerna. Motstånden är dimensionerade så de inte skall förorsaka stora förluster men ändå hålla spänningen vid önskad nivå vid tex. pulldown sammanhang.

5.4 FÖRLUSTBERÄKNINGAR

Förlustberäkningarna utförs för att kunna bestämma effektiviteten för frekvensomvandlaren och styret. I effektiviteten beaktas inte förluster i motorn utan bara förlusterna i drivsystemet.

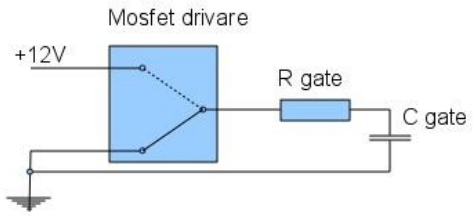
Mikrokontrollen skall ha en operational strömförbrukning enligt tillverkarens datablad (Microchip 2005b s. 151) på max 59mA men detta exkluderar belastningar på mikrokontrollerns ben. Vilket ger att maximala effektförlusten i mikrokontrollern är 0,195W då den drivs på 3,3Vdc.

Hall-givaren har jag ingen data för, men motsvarande komponenter har en medeltalig max-förbrukning på 8mA. Med en matningsspänning på 5 volt är effektförlusten i en hall-givare till 0,04W då där finns tre stycken givare blir totala förbrukningen 0,12W. Då hall givarna är aktiva så dras det även ström genom dess pullup motstånd på 2,2kohm. Hall givarna är aktiva en tredjedel av ett varv av magnetfältet. I kretsen finns det tre givare, och då kan man räkna att det konstant dras ström genom ett pulldown motstånd, $P=U^2/R$ $P=3.3^2/2200$. Detta ger ungefärliga effektförlusten i motstånden till 0,005W

Mosfetdriv-kretsarna tar även en del matningsström från logiksidan som räknas med i denna del av förlusterna. De tar även en ström till logiska ingångarna då de är på. Där finns 6 st logiska ingångar och 3 spänningsmatningar som drar ungefärliga 30uA per styck, så drivarnas effektförluster på logiksidan blir 1,35 mW

Summerat är Effektförlusten på logiksidan i klass med 0,32W och då en ungefärlig strömförbruking på 100mA på 3,3volts matningen.

Vid beräkningar av förluster i gatedrivarna har jag använt mig av följande formler som jag har hittat i applikationsnoteringen 898 från microchip. (Microchip. 2003:AN898, sida 11)

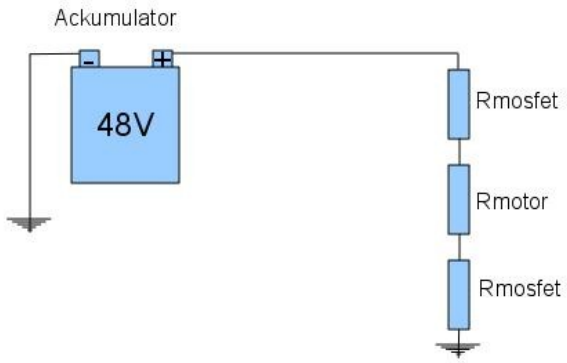


Figur 3: Ekvivalent shema för mosfetuppladdning och urladdning

Effekt som krävs för att ladda up effektransistorns styre Pgate = 0.5 Cgate * Vgate^2 * f där f är kopplingsfrekvensen som i denna krets är max 20kHz. Om man matar in värden som hittas i tillverkarens datablad för effektranistorn IRFS4010 så är Cgate = ca 10nF = 0,00000001F och Vgate torde vara ca 11V. Då blir effektförbrukningen i transistorns styre ca 12mW. Det finns 6st

transistorer blir totala förlusten ca 73mW. Denna effektförlust sker i drivaren då mosfet drivaren uppladdar och urladdar styret genom sig.

Förlusterna i kopplingsdelen består av två delar: Den förlust som sker då transistorn är på och den som sker då transistorn kopplas på och av. Förlusten som sker då transistorn är på kallas ledningsförlust och räknas ut enligt ohms lag. Formeln blir enligt följande $P_{ledning}=I_{rms}^2*R_{ledning}$ där I_{rms} är den beräknade konstanta strömmen och $R_{ledning}$ transistorns ledningsresistans. Om man sätter in värden från datablad och beräknad effekt i formeln blir resultatet $P_{ledning}$=1,2W, och om man räknar med att tre transistorer leder kontinuerligt blir resultatet för ledningsförlusterna maximalt 3,6W vid en kW belastning. Vid påkoppling och avstängning av transistorerna sker även en förlust som heter kopplingsförlust. Denna förlust är beroende på komponentens kopplingstid och kopplingsfrekvensen som transistorn kopplas med. Dvs om man har en kopplings frekvens på 20kHz kopplar transistorerna av och på 40 000 gånger per sekund. Vilket innebär en högre kopplingsförlust än vid tex 8kHz då transistorerna byter läge 16 000 gånger per sekund. Jag mätte påkopplingstiden till 70ns och avkopplingstiden till 80ns med oscilloskop.



Figur 4: Ekvivalent schema för transistorförluster.

Med följande formel kan förlusterna vid koppling beräknas.

$$P_s=(V_s*f_s)/2 * (T_{on} + T_{off}) * I_{on} \quad [9]$$

Lägger jag in värden från min krets ser formeln ut enligt följande:

P_s=(52V*20 000Hz)/2 * (0,000 000 07s + 0,000 000 008s) * 20A

Som reslutat fås att förlusterna i en transistor blir 0,156W. Om man lika som med ledningsförlust antar att tre transistorer kan anses kopplade vid alla tillfällen så är då kopplinsförlusterna 3 * 0,156W vilket ger 0,468 W totala kopplinsförluster.

Effektförlusten i kraftelektroniken blir totalt 4,068W. Detta resulterar i en verkningsgrad för kraftelektroniken enligt följande formel.

Eff = (Pin – Pfl) / Pin *100% [10]

Lägger jag in de värden som jag räknat ut ser formeln ut enligt följande.

Eff= (1000W – 4,068W) / 1000W *100%

Enligt dessa beräkningar är effektiviteten för frekvensomvandlaren 99,59 % Vid 20A belastning.

5.5 PROGRAMMET FÖR MIKROKONTROLLERN

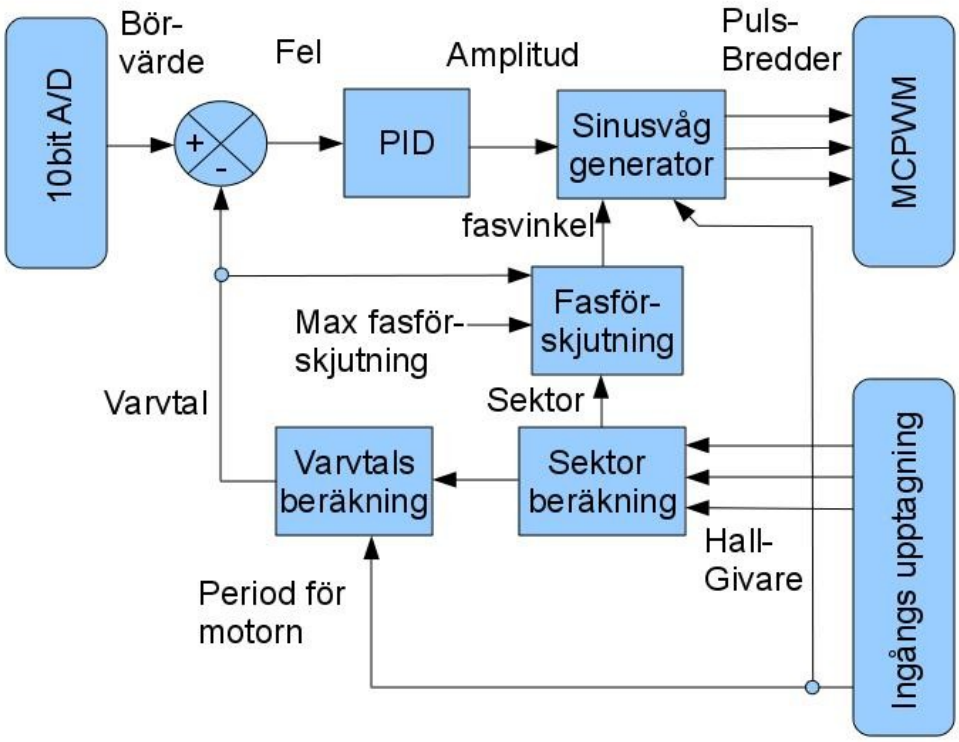
Som utvecklingsplattform använder jag Microchips MPLAB IDE program som därtill har C30 modulen. MPLAB med C30 modulen tillåter utveckling av program till dsPIC seriens mikrokontrollers med Ansi C programmerings språk.

För mjukvara i drivsystemet har jag använt mig av Microchips demonstrationsprogram. Mjukvaran fungerar på principen att en permanentmagnetiserad synkronmotor kan ses som en vanlig permanentmagnetiserad likströmsmotor. I permanentmagnetiserade likströmsmotorn kan man reglera motorns varvtal genom att öka eller minska på dess spänning.

I detta fall bara ökar och sänker man på sinusspännigens amplitud och ställer frekvensen för sinusvågorna enligt med vilket vartal motorn roterar. Man kan välja i programmet om man vill ha PID reglering av hastigheten av motorn och börvärdet avläses från ett gasreglage (potentiometer). Eller bara så att önskad amplitud av sinusspänningen är inmatad från gasreglaget. Jag har använt mig av att bara mata in önskad amplitud från en potentiometer. Nedan finns det presenterat

principbilden (Illustration 6) för mjukvaran som styr motorn (Microchip: 2005: 5). Jag har använt mig av att ha PID regulatorn avstängd då det gav direkt kontroll av motorn genom att ha amplituden direkt styrd från potentiometern.

Programmets operation är avbrott-styrd, så principbilden är inte beskrivande för själva flödet i programmet men ger en bra överblick över vad programmet gör.



Figur 5: Blockschema för mjukvaran.

Jag har modifierat programmet en del för att passa till detta ändamål. Jag har bifogat källkoden som bilaga 2. Men även den orörda koden fungerar på denna hårdvara men om motorn har något annat poltal än den som är använd i exempelprogrammet (Microchip 2005c) kommer motorn inte att fungera som den skall.

Koden är modifierad så att drivsystemet startar då man vrider på gasen. Mjukvaran är också ändrad så man inte kan få motorn att starta baklänges eller att byta riktning när man kör. Detta skulle medföra en enorm effektförbrukning då motorn skulle köras i motsatt riktning till dess rotation.

Troligtvis skulle kraftelektronik komponenterna förstöras om motorn skulle byta riktning under körning.

Källkoden i sin helhet kan återfinnas i bilaga 3 och bilaga 4. Det skulle kunna skriva om hela koden men för testning och verifiering av hårdvaran har det ingen betydelse om man har halva eller skalan av potentiometern till sitt förfogande.

5.5.1 OLIKA ARBETSSKEDEN FÖR PROGRAMMET

Programmets olika arbetskeden går att skilja på som "tomgång", acceleration, hålla en konstant hastighet samt att bromsa in. Kodmässigt använder alla förutom tomgång samma kodsnuttar. Det är bara yttre variabler som inverkar på utmatningen av sinusvågen.

Börvärdet under startgränsen:

Mjukvaran cirkulerar bara samplingen av ingången för börvärde och väntar på att börvärdet skall stiga över startgränsvärdet då pwm utgångarna aktiveras och motorn börjar köras.

Acceleration:

Motorn körs och när gasen ökas, ökas amplituden av sinusspänningen och frekvensen ändras baserat på hall sensorernas information, så att när motorn får en högre spänning börjar motorn accelerera och i princip tappar låsningen med frekvensen som matas ut. Frekvensen korrigeras sedan för att motsvara rotorns rotationshastighet.

Konstant hastighet:

Om motorn har ett konstant moment håller mjukvaran frekvensen och amplituden av sinus-spänningarna konstant. Om momentet ökar eller minskar måste amplituden på sinusvågen ökas eller minskas för att hålla konstant hastighet. Men frekvensen för sinusvågen skall hållas konstant.

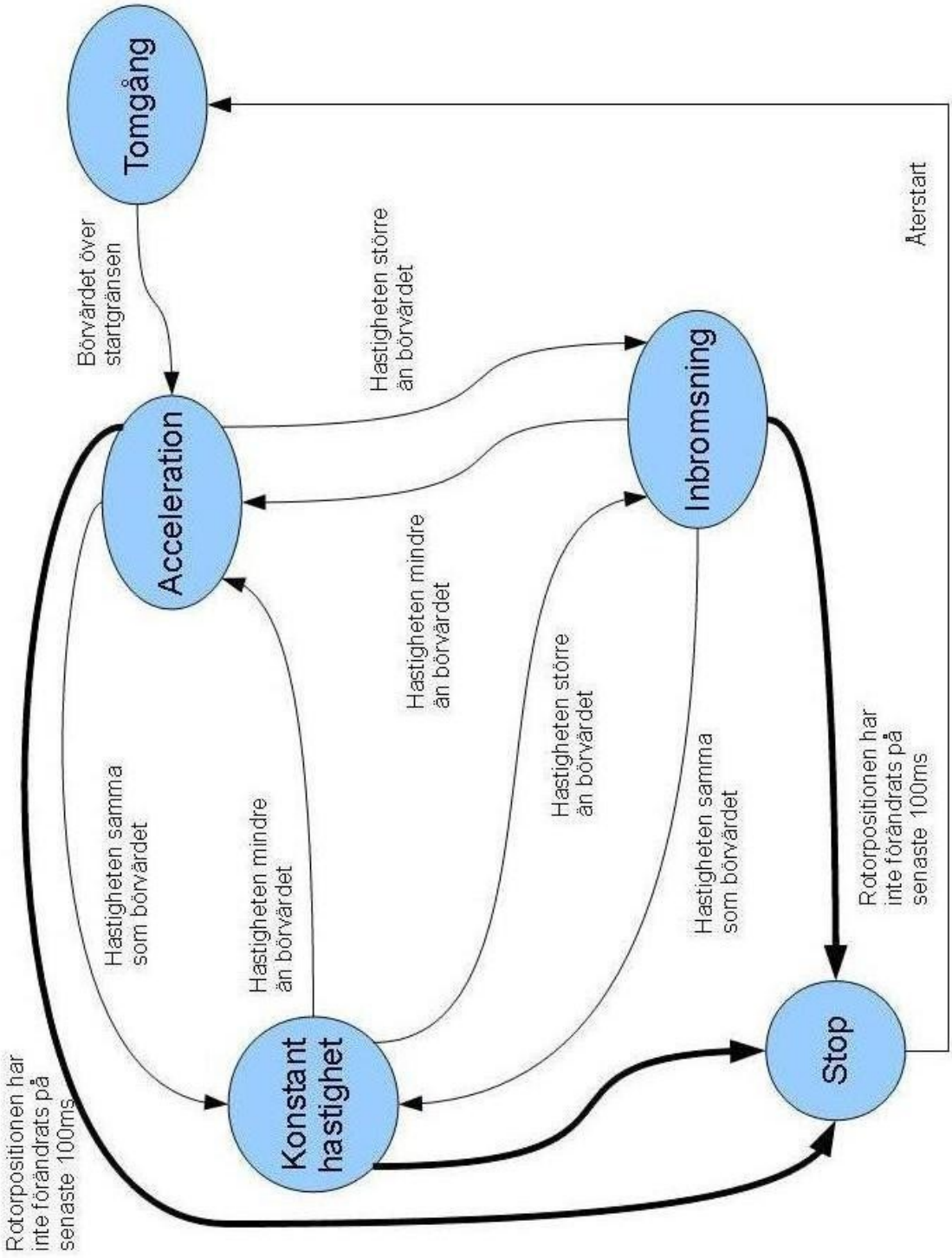
Retardation:

Om man har mindre amplitud kommer frekvensen att sänkas då rotorn saktar in. Detta ger upphov till översynkron bromsning (se kapitel 4.1.1).

Avstängning av drivsystemet:

I testfordonet har mjukvaran varit så att motorn inte mera körs då motorn har varit låst i mera än 100ms. Detta har varit praktiskt taget så att man justerar gasen till minsta möjliga och står på stället så har mjukvaran stängt av motorn pågrund av låst rotor.

På följande sidan finns ett flödesschema (figur 6) som beskriver funktionen av mjukvaran. Mjukvaran är interruptstyrd så när nånting händer beror detta på yttre omständigheter och inte på en förutbestämd ordning.



Figur 6: Arbetsskeden flödesschema.

6 BYGGANDE AV KRETSEN

Jag använde mig av Cadsoft:s Eagle layout editor (www.cadsoft.de) som jag härefter bara refererar till som Eagle. Jag använde mig av programmet i uppbyggandet av kretsschemat och kretskortlayouten. Layouten för kretskorten kan återfinnas i bilaga 2.

Vid kretskortstillverkningen använde jag mig av Arcadas resurer då jag etsade kretskorten och lödde dem. Vid Arcada används litografisk process för att göra kretskort. Detta innebär att kretskortets layout ("kopparlinjer") från kretskortsplaneringsprogrammet skrivs ut på en plastfilm. Kretskortet som är lackat med positivt lack belyses med UV ljus genom denna plastfilm. Då lacket är positivt så blir det lack kvar som inte har exponeras för UV ljus. Kretskortet framkallas sedan i en lutlösning som löser upp lacket som blivit utsatt för UV ljus. Efter detta nedsänkes kretskortet i en frätande vätska som fräter bort kopparen som inte är skyddad av lack.

Vid kretskorttillverking måste man ofta borra hål om man har komponenter som har ben som skall gå genom kretskortet. Då detta kretskort är till 100% ytmonterade komponenter är detta behov minimalt och kräver liten tid. Sedan är det bara att putsa bort lacket med tex aceton innan man löder fast komponenterna. I fallet av detta kretskort var det fråga om relativt stora komponenter men ändå var bruket av förstoringsglas samt pincett nödvändigt.

Drivsystemet består av 2 kretskort, styrelektronikkretskortet samt effektdelens kretskort. Nedan följer en beskrivning av dessa kretskort.

6.1 STYRELEKTRONIKEN

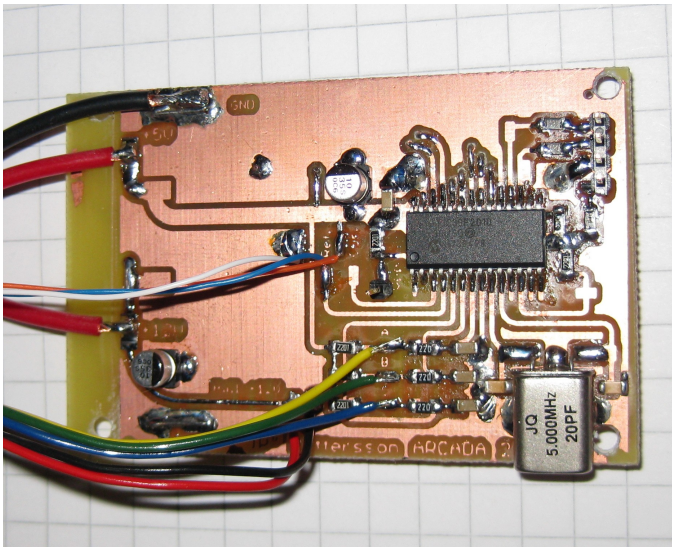


Bild 2: Styrkortets övre sida

Styrelektroniken är placerat på ett skilt kretskort från spänningsmodulatorn. Detta för att minska på störningar som kan induceras i styrelektronikens kopparbanor som kan störa eller i värsta fall söndra drivsystemet. Den känsligaste elektroniken (Bild 2) som omfattar mikrokontrollern samt de komponenter som krävs för dess funktion och även alla givaringångar, finns på ena sidan av det dubbelsidiga styrelektronikkretskortet.

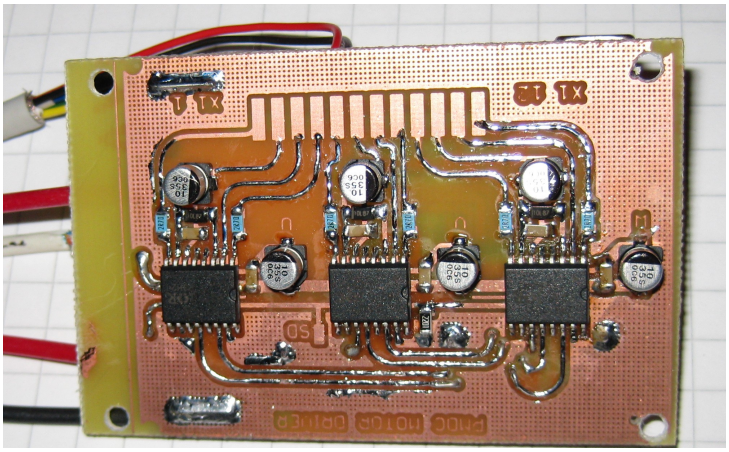


Bild 3: Styrkortets undre sida med drivkretsarna för transistorerna

På andra sidan av kretskortet finns grinddrivkretsarna (Bild 3) för effektransistorerna. Dessa är placerade där dels för att spara utrymme och göra kretskortet och layouten mera kompakt, men även för att minska på störningar som kan komma från drivkretsarna. Det går ändå en relativt stor ström

till grinddrivarna då de kopplar av och på effekttransistorerna jämfört med strömförbrukningen av mikrokontrollern. Störningarna som kan uppstå från grinddrivkretsarna kan ha störande effekt på mikrokontrollern då den är av storleksordning kopplingsfrekvensen för effekttransistorerna. I detta drivsystem mellan 4kHz och 20kHz. Även högre frekvenser är möjliga men är i frekvensomriktarsammanhang inte till stor nytta. Högre kopplingsfrekvenser ger upphov till större värmeförlust i effektransistorerna i form av kopplingsförluster. Därtill filtreras de övertonerna som bildas i den uppbygda sinusvågen bort av motorn och omvandlas till värme. Detta i sin tur leder till att verkningsgraden försämras för motorn.

6.2 SPÄNNINGSMODULATORN

Spänningsmodulatorn är uppbyggd så att det skall gå ström genom så korta ledare som möjligt för att reducera elektromagnetisk strålning som utstrålas från ledare där en ström flyter igenom. Det visade sig vara problem att få alla komponenter lödda på samma gång; Detta löstes provisoriskt med att placera kretskortet med tennlod och komponenter på en spisplatta (bild 4). Då kretskortet var etsat på en aluminiumplatta ledde det bra över värmen och lödningen gick enkelt. En lödugn skulle vara idealisk men då detta inte fanns på Arcada måste det improviseras.

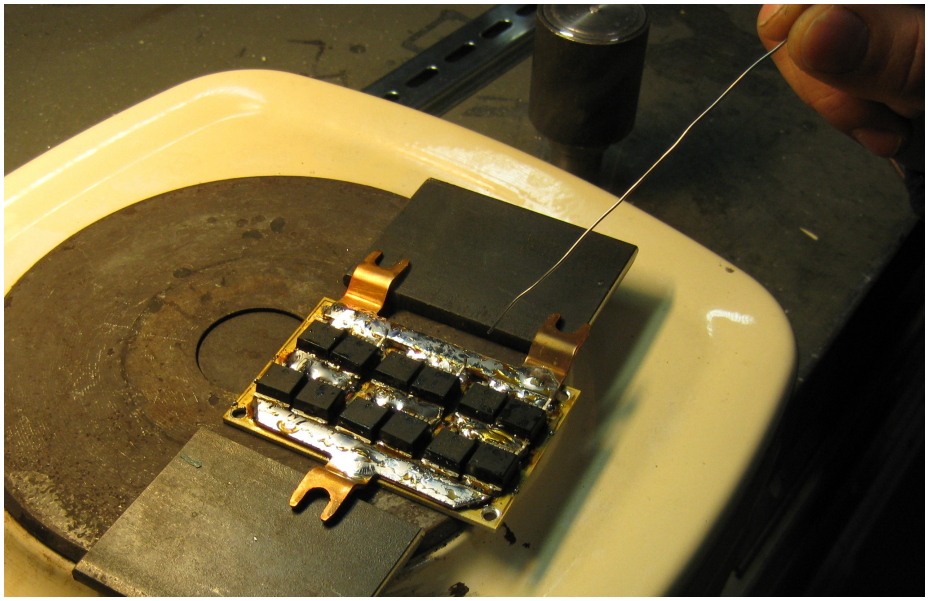


Bild 4: Lödning av effektdelen på en kokplatta.

6.3 FUNKTIONSBESKRIVNING AV HÅRDVARAN

6.3.1 INLÄSNING AV BÖRVÄRDE

Börvärdet fås genom spänningsdelning över en lineär 5kΩ potentiometer som är monterad på fordonets gasreglage. Det kunde även användas såkallade hall-effekt gasreglagen genom en spänningsdelare, då mikrokontrollern körs på 3.3Vdc och de gasreglagen som finns på marknaden ger ut 0-5Vdc. Mikrokontrollen samplar denna ingång varje mikrosekund för ett börvärde.

6.3.2 HALL-SENSORER

Hallgivarna fungerar så att de jordar utgången då magnetflödet når ett gränsvärde vid hallgivaren och utgången återgår till flytande då magnetfältet faller under avstängingsgränsen. Hall sensorerna är kopplade till mikrokontrollen genom ett lågpassfilter bestående av en kondensator på 0.1uF och ett seriemotstånd på 220Ω. Filtret har en filtreringsfrekvens på 7,2 kHz. För att mikrokontrollen skall kunna läsa tillståndet för hall sensorerna krävs ett pullup motsånd som drar upp spänningen till en logisk etta då inte hallsensorens utgång är kopplad till jorden. Detta pullup motstånd är i denna krets 1.5kΩ för att ge en tillräckligt snabb tillståndsändring men ändå inte ha stora effektförsluster då det finns logikens matninsspänning över motstånden då hall givarens utgång är aktiv.

6.3.3 MOSFET GRIND DRIVARE

Mosfet grind drivarna kopplar sina utgångar till antigen plus eller minus beroende på insignalen som mikrokontrollen styr dem med. I min krets används en drivare (bild 5) som är planerad för anvädning i tvåpulskoppling. Detta medför att samma chip innehåller två grind drivare varav den ena är isolerad då den måste arbeta i en potential som är högre än matningsspänningen till frekvensomvandlaren.

Transistorns styre har en gateresistor som skall begränsa strömmen som går till styret i transistorn då styret uppladdas och urladdas. Jag har använt 2,7 ohms motstånd som begränsar strömmen minimalt. Detta ger en snabb av- och påkoppling av transistorn som i sin tur leder till minskade kopplingsförluster.

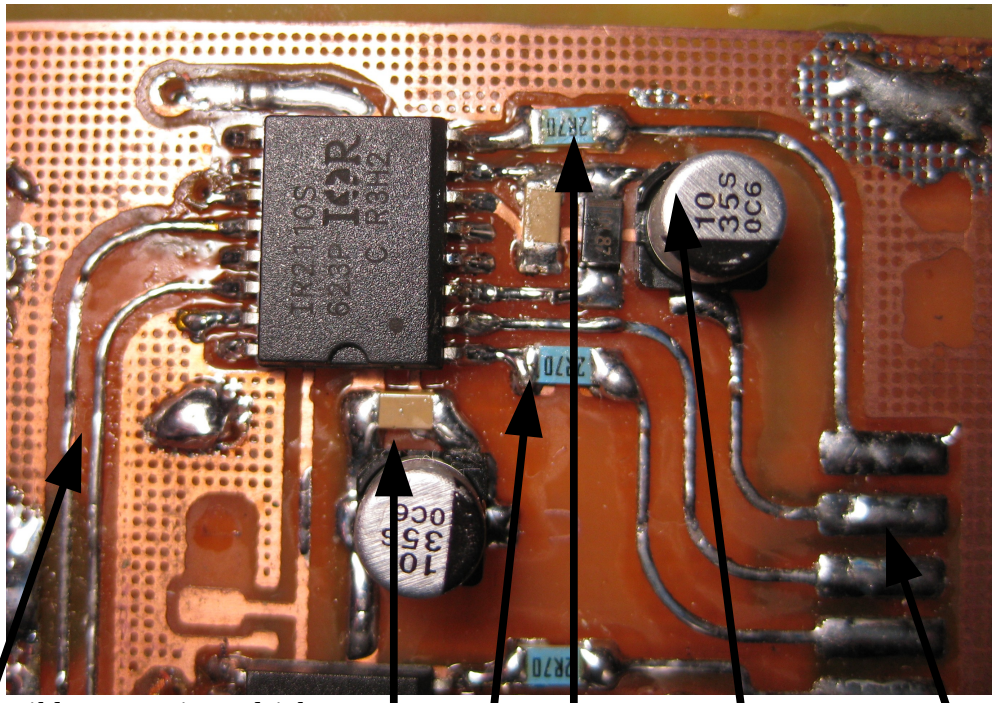


Bild 5: Transistor drivkrets.

Låga sidans drivare är kopplad till grinden på transistorn som sitter i nedre delen av H-bryggan. Transistorn sitter således mellan belastningen och minus. Höga sidans grind drivare är kopplad till grinden på transistorn som sitter i övre delen av H-bryggan, mellan + och belastningen. Båda styrda transistorerna är av N-kanal typ mosfet. N-kanal mosfet är att föredra över P-kanal mosfet för att N-kanal transistorn kopplar snabbare och har mindre ledningsresistans, vilket innebär mindre förluster.

Spänningsförsörjningen för höga sidans drivare är konstruerad enligt ”bootstrap” principen. Detta möjliggör att den spänning som krävs för att styra höga sidans transistorn fås genom att ladda upp en kondensator då låga transistorn leder. Uppladdningen sker vid det tillfälle då utgången är kopplad till jord genom lägre transistorn. Vid uppladdingen ladda ”bootstrap” kondensatorn upp till en spänning på 12Vdc som sedan används för att köra grind drivaren samt ladda upp transistorns styre. Höga sidans ”bootstrap” kondensator laddas upp igen vid nästa koppling av låga transistorn.

Fördelen med ”bootsrap” tekniken är att man eliminerar krångliga flytande spänningskällor och minskar på antalet komponenter som krävs för funktion. Med mindre komponentantal finns det också ett mindre antal komponenter som kan gå sönder.

6.3.4 PROGRAMMERING AV MIKROKONTROLLERN

"In Circuit Serial Programming" gör det möjligt att programmera mikrokontrollern då den är lödd på kretskortet. Då mikrokontrollen är ytmonterad är den krånglig att programmera före den löds fast. För programmeringen (bild 6) krävs det bara kontakter på kretskortet som man kan koppla fast programmeraren i. Jag har försökt att inte använda mikrokontrollerns programmeringsben till någon annan funktion än detta då det kan medföra problem vid omprogrammering. Om programmeringsbenen har en annan funktion i kretsen och om kretsen är igång när den programmeras kan det i värsta fall förstöra nån del av kretsen eller skada programeringsenheten.

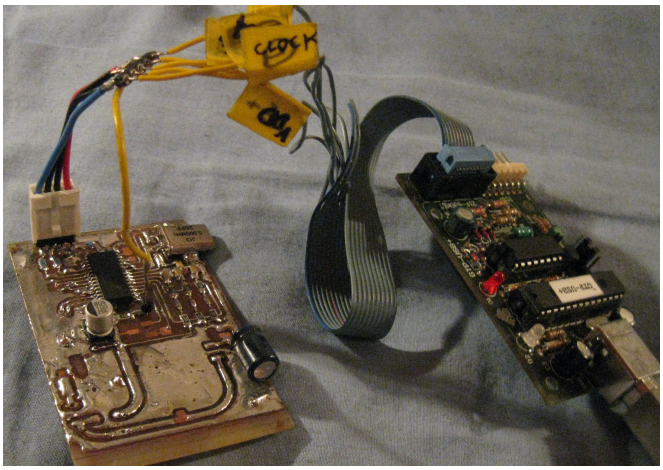


Bild 6: Programering av mikrokontrollern.

Jag använder mig av WINPIC GTP+ USB programmerare för att programmera mikrokontrollen.

6.4 TESTFORDON

Bild 7: Testfordon

Som testfordon (Bild 7) fungerade en gammal moped som jag fått av en vän som donation samt delar av en gammal cykel som jag svetsade fast på mopedramen. Denna prototyp hade som syfte att testa funktionsprincipen och få riktgivande testdata. Bakgaffeln från mopeden var inte direkt passande till mitt drivande hjul, utan bakgaffeln måste modifieras och sänkas lite för att ge rum för en större hjuldiameter än originalet. Elektroniken var i testskedet utspritt på en plywoodskiva som monterats på bakdelen av ramen. I prototypen är transistorerna och skyddsdioderna monterade på aluminiumbitar som kyl ner komponenterna med fartvinden. Ackumulatorerna som används är 12V 9Ah slutna blyackumulatorer. De är kopplade 4 st i serie för att uppnå nominella spänningen på 48V. Underdimensionerade ackumulatorer används för att begränsa kortslutningsströmmen och maximala strömmen då de har större intern resistans än större ackumulatorer.

Gashandtaget är borttaget och ersatt med en potentiometer som är kopplad till mikrokontrollern. Mekaniskt fungerar gasen som en tumgas. Axeln av potentiometern är fäst i en platta som är fjäderbelastad och har mekaniska begränsningar för ändlägen. Mopeden har sin ursprungliga

frambroms av säkerhetskäl. Bakbromsen är borttagen och har inte kunnats ersätta då navmotorn saknar trumbroms eller fästen för skivbroms.

Köregenskaperna för testfordonet är dåliga på grund av sliten fjädring samt att tyngdpunkten är högt uppe på grund av trampgeneratorns (ej behandlad i detta slutarbete) och ackumulatorpaketets placering.

Testerna är körda med en annan kretslayout och transistorer än i den slutliga versionen av kretskortet som finns presenterat i detta slutarbetets bilagor. Men prestandamässigt är de motsvarande.

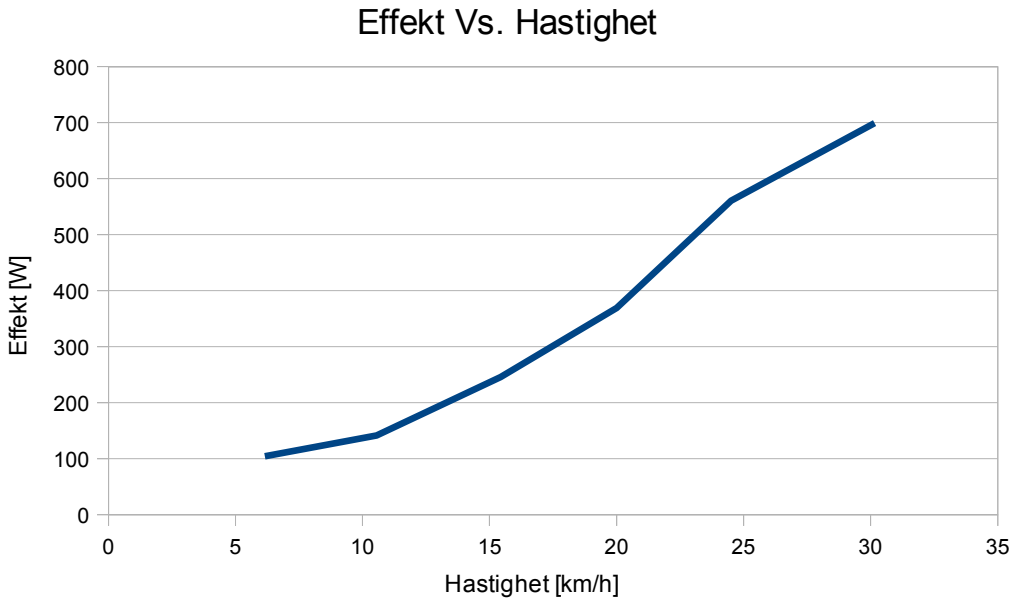
7 TESTNING OCH VERIFIERING AV KRETSEN

För verifieringen att drivsystemet uppfyller kriterierna som ställs på det har det gjorts en serie mätningar. En mätning har gjorts som mäter förbrukad effekt för en viss hastighet. Med denna mätning kan man verifiera om drivsystemet är rätt dimensionerat. En mätning för spänningen som frekvensomvandlaren matar ut har även utförts för att demonstrera pulsbreddsmodulerad sinusvåg.

7.1 MÄTRESULTAT

7.1.1 RESULTAT FRÅN TESTFORDONKÖRNINGEN

Denna test utfördes med testfordonet i Arabiastrandens köpcentrums parkeringshall då inte Arcadas parkeringshall var tillräckligt stor för att uppnå en hastighet vars mätdata har relevans för drivsystemets tänkta användningsområde. Resultaten är förevisade i figur 3.



Figur 2: Förhållande mellan effekt och hastighet på en plan köryta utan vind.

Testerna kan anses relativt pålitliga då testerna utfördes vid olika tillfällen på samma plats med liknande förhållanden. Högre hastigheter uppnåddes men mätresultaten kan inte anses jämförbara med de andra mätvärden då det ingick acceleration i effektförbrukningen. Raksträckan i Arabiastrandens parkeringshall räckte inte till för att hålla en konstant topphastighet.

Den uppmätta effekten är som bekant el effekt, och motorer är angivna i mekanisk effekt. Men det framgår tydligt att en 500W motor är underdimensionerad. Största effekterna är uppmätta vid acceleration till 4kW. Men vid testkörningen har det försökts hållas en konstant hastighet. Mätningarna utfördes genom att ha en spänningsmätare kopplad till ackumulatorpaketet för att övervaka spänningen och strömmen är uppmätt genom att mäta spänningsfallet över ett 0.01 ohm effektmotstånd. Hastigheten är uppmätt med en cykelmätare från bakhjulet. Alla dessa tre mätningar filmades under körning med en digitalkamera och resultatvärden är sedan inmatade i en tabell ur filmen.

7.1.2 UTMATAD SPÄNNING

Spänningen som matas ut på en fas är inte en sinusspännig i sig utan en komponent som resulterar i en sinusspännig i förhållande till en annan fas. Nedan presenteras bild 5 som är tagen med digitalkamera av ett oscilloskops mätresultat mellan två faser som demonstrerar hur huvudspänningen ser ut. Mätningen är utförd med bara en ackumulator kopplad för att minimera störningar på oscilloskopet.

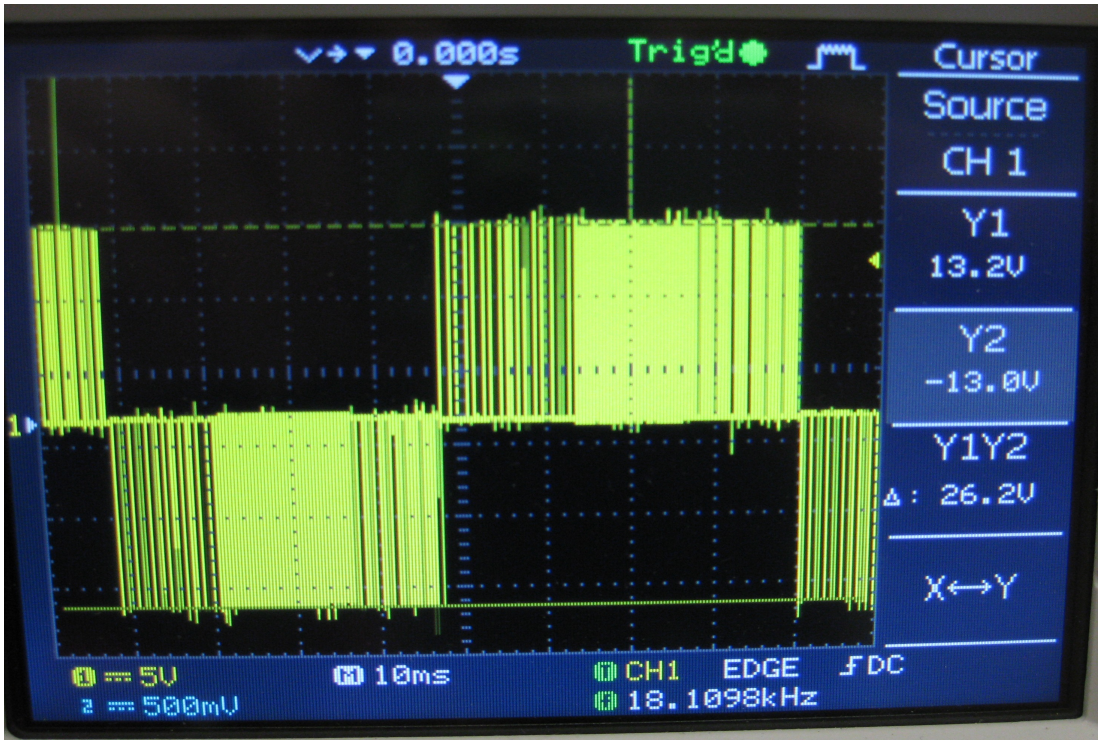


Bild 8: Huvudspänningen mellan två faser, ackumulatorspänning 12V.

Man ser tydligt hur sinusvågen består av pwm signal och att man får full huvudspänning på sinusvågen av 12 volt.

8 SLUTDISKUSSION

Den eldriva mopeden som blev resultatet av detta slutarbete är otroligt trevlig att köra då gasen har snabbt svar och vridet är där direkt. Inget varvande, växlande och väntande eller liknande som det är med vanliga mopeder. Av de personer som har provkört prototypen har största delen uttryckt sig i stilen med att de vill ha en sån moped.

Med eldrivna mopeder kan man även köra inomhus vilket kunde ge nya tillämpningar i tex sjukhus eller stora maskinhallar.

Vad gäller frekvensomvandlarens effektivitet stöder egna iakttagelser de beräknade resultaten. Prototypen har bara aluminiumplåtbitar av storlek ca 30 * 50 mm och tjocklek av 1,5mm. Med mycket accelerering och bromsning blir skivorna uppvärmda till ca 50ºC. Detta skulle tyda på att effektförlusterna är små. Motorns anslutningskabel uppvärms i högre grad än transistorernas kylplattor, vilket också tyder på att förlusterna är små i transistorerna.

Jag har tyvärr bara beräkningar att stöda effektiviteten av kretsen på. En mätning av frekvensomvandlarens effektivitet skulle ha varit en värdefull information, också med tanke på möjliga framtida förbättringsprojekt.

Men jag ser det som helt möjligt att man kan börja se mera och mera elfordon i gatubilden under de kommande årtionden. Ackumulatorerna blir bättre för varje år och även motorernas prestanda kan stiga ännu en aning om storskalig produktion startas så att bättre magneters pris fås ned.

8.1 PROBLEM VID FÖRVERKLIGANDE

Största problemet vid förverkligandet låg i att anslutningskablarna mellan ackumulatorpaketet och frekvensomvandlaren genererade höga spänningstoppar, dvs. transientter då stor ström snabbt kopplades av och på. Spänningstopparna övergick transistorernas spänningstålighet på 100V redan vid en drivspänning på 12V. Det fanns 68volts överspänningsskydd kopplade men de kunde bara absorbera en viss energimängd.

Problemets lösning var att öka på tiden som transistorerna kopplar spänningen på och av på. Genom att öka på resistansen i gate drivkretsen saktar uppladdningen och urladdningen in av styret på transistorn. Detta gjorde att förändringen av ström per tid minskade, dvs $\Delta I / \Delta T$ minskade. Också lades en snubber så nära transistorerna som möjligt. Snubbern bestod av 3st 10uF plastfilmskondensatorer samt 3 mindre 2.2uF plastfilmskondensatorer vilka har låg inre

serieresistans samt en 4700uF elektrolytkondensator för att jämna ut strömmen så det inte blev stora och snabba strömförändringar i matningsledningen.

8.2 FÖRBÄTTRINGS- OCH UTVECKLINGSFÖRSLAG

8.2.1 PROGRAMVARAN

Jag har nu använt ett exempelprogram från microchip (www.microchip.com) för att köra motorn, men programmet är inte helt så optimerat för detta ändamål som det skulle kunna vara. Men oavsett av hur programmet ser ut och vilken modulationsteknik det använder så är ändå hårdvaran samma. Så min hårdvara som jag utvecklat kan köras med ett program tex framtaget i Arcada. Detta kan ge möjlighet för flera slutarbeten som kan fortsätta utveckla programmet.

8.2.2 HÅRDVARAN

Med tanke på att drivsystemet skall användas i verkliga världen bör även vissa feltillstånd beaktas och om möjligt göra drivsystemet så att det kan klara av tänkbara fel som kan uppstå. Största felet som kan uppstå på en elmotor är att rotorn är låst eller att lindningarna kortsluts. För kortslutning är ett analogt och snabbt skydd ett måste.

Skyddet kunde kombineras till en startlåsning som skulle hindra ytterligare skador på drivsystemet om transistorerna har en kortslutning i sig som resultat av tidigare fel. Skyddet skulle då förhindra att strömmen slås på till transistorerna. Effektbehovet och krav på hög verkningsgrad för drivsystemet utesluter mätning över ett shuntmotstånd, vilket skulle ha varit lätt att utföra. Det krävs en strömmätning som är snabb och tillräckligt noggrann för att kunna skilja åt kortslutning och vanlig drift. Strömmätning med hjälp av hall givare kunde vara ett alternativ. Samma givare kunde också användas för att begränsa strömmen till motorn i normal drift.

Användning av IGBT kunde utforskas då det har visat sig att induktansen i ledningarna mellan frekvensomvandlaren och ackumulatorerna ger upphov till spänningstoppar på över 100volt om det inte finns relativt stora kondensatorer mycket nära effektdelen som filtrerar bort spänningstoppar.

Men även ett bättre överspännings skydd för drivsystemet kunde utvecklas så man kan köra kretsen med en högre ackumulatorspänning än nu är möjligt.

En utredning om vilken kopplingsfrekvens som ger bästa effektiviteten för motorn kunde även utföras. Eller om det skulle vara bättre med en varierande kopplingsfrekvens beroende på motorns hastighet, då lindningarna i motorn beter sig på olika sätt beroende på rotorns hastighet.

Det kunde även vara av värde att göra en jämförelse av effektiviteten av motorn om man kör den med högre spänning och mindre ström eller mindre spänning och högre ström. Om körhastigheten är låg vore det kanske effektivare att ha ackumulatorpaketet på 36V spänning och inte 48V som det är nu. Eller om högre spänning vore bättre för effektiviteten då strömmen minskar i kraftelektronikkomponenterna.

IRF har lanserat en ny mosfetdrivare ämnad för fordonsbruk. Denna drivare som har mindre kapsling kunde göra layouten betydligt kompaktare, som kanske skulle möjliggöra att ha drivaren direkt vid transistorerna, det skulle minska risken för störningar och överhopp i styrkretskortet t.ex. p.g.a. fukt eller dylikt.

KÄLLFÖRTECKING

Adams, Jonathan, International Rectifier. 2001: Bootstrap Component Selection For Control IC's. Tillgänglig: http://www.irf.com/technical-info/designtp/dt98-2.pdf. Hämtat 21.11.2007.

Alfredsson, Alf. 1994. Elkraft. Liber Utbildning. 344 s. ISBN 91-634-0951-8

Alfredsson, Alf, Jacobsson, Karl Axel, Rejminger, Anders, Sinner, Bengt. 1996. El kraft handboken Elmaskiner. Liber AB. 470 s. ISBN 91-47-00066-X

Basic Robotics Inc. 2008: Help - Vehicle Power calculator and visualizer. Tillgänglig: http://powcal.basic-robotics.com/help.htm Hämtat 25.09.2009.

Buchman, Isidor. 2003: What's the best battery? Tillgänglig: www.batteryuniversity.com/partone-3.htm hämtat 10.10.2009

Höst, Martin Regnell, Björn Runeson, Per. 2006. Att genomföra examensarbete. Lund: Studentlitteratur. 153 s. ISBN 91-44-00521-0.

International Rectifier 2001: DT98-2 Tillgänglit: http://www.irf.com/technical-info/designtp/dt98-2.pdf Hämtat 21.11.2007

International Rectifier 2005: IR2110(S)PbF/IR2113(S)PbF Datasheet. Tillgänglig: www.irf.com/product-info/datasheets/data/ir2110.pdf. Hämtat: 21.01.2009.

International Rectifier 2008: IRFS4010-7PPbF Tillgänglit: http://www.irf.com/product-info/datasheets/data/irfs4010-7ppbf.pdf Hämtat 20.01.2009.

Niiranen, Jouko. 1999. Sähkömoottorikäytön digitaalinen ohjaus. Yliopistokustannus/Otatieto. 379 s. ISBN 951-672-270-9.

Microchip 2003: AN898 Tillgänglit: http://ww1.microchip.com/downloads/en/AppNotes/00898a.pdf Hämtat 08.10.2007.

Microchip 2005: AN1017 Tillgänglit: http://ww1.microchip.com/downloads/en/AppNotes/01017A.pdf Hämtat 09.10.2007

Microchip 2005b: Datasheet for dsPIC30f2010 Tillgänglig: ww1.microchip.com/downloads/en/DeviceDoc/70118g.pdf Hämtat: 09.10.2007.

Microchip 2005c: CE003_Sinusoidal_BLDC Tillgänglig: http://ww1.microchip.com/downloads/en/AppNotes/AN1017_30F2010_V1.zip Hämtat: 09.10.2007.

Mogensen, Hans. 1989. El maskiner. Almqvist & Wiksell Förlag. 384 s. ISBN 91-21-12148-6.

Valentine, Richard. 1998. Motor control electronics handbook. McGraw-Hill Handbooks. 704 s. ISBN 0-07-066810-8

Von Hertzen, Maria, Stolt , Kerstin. 2009: Skrivguide 2009 (version1). Tillgänglig: http://studieguide.arcada.fi/webfm_send/434 Hämtat 23.11.2009.

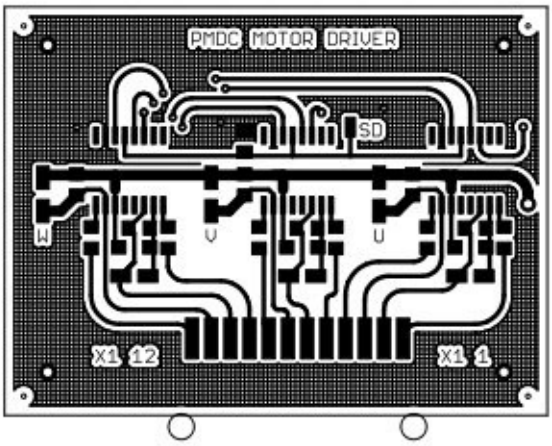
Yuasa Battery Inc. 2009: NP 24-12 Datasheet. Tillgänglig: http://www.yuasabatteries.com/pdfs/NP_24_12_DataSheet.pdf Hämtat 10.10.2009.

+24V
C18
IC6
15V
+12V
C15
10uF
AGND
+5VHALL
IC5
LP2950CDT-5.0
C16
10uF
IC7
7806DT
VI
VO
GND
C17
10uF
VDD
C6
0.1uF
C7
0.1uF
C14
0.1uF
IC4
RB0/CN2/AN0/EMUD3/VR+
RB1/CN3/AN1/EMUC3/VR-
RB2/CN4/AN2/SS1
RB3/CN5/AN3/INDX
RB4/CN6/AN4/QEA/IC7
RB5/CN7/AN5/QEB/IC8
RC13/CN1/EMUD1/ATX/T2C
RC14/CN0/EMUC1/ARX/T1C
RC15/OSC2/CLKO
OSC1/CLKI
RD0/INT1/EMUC2/OC1/IC1
PWM1L/RE0
PWM1H/RE1
PWM2L/RE2
PWM2H/RE3
PWM3L/RE4
PWM3H/RE5
INT0/FLTA/SCK1/OCFA/RE8
PGC/EMUC/RX/SDI1/SDA/RF2
PGD/EMUD/TX/SDO1/SCL/RF3
VPP/MCLR
INT2/EMUD2/OC2/IC2/RD1
DSPIC30F3010/SO
5Mhz
Q1
C2
15pF
C1
15pF
R13
10k
R1
10k
R6
2k
RUN
CURRENT
Gasreglage
SPDREF
D4
Hall sensor ingångar
GNDHALL
R15
2k
R3
20
C3
0.1uF
HALLA
R17
2k
R4
20
C4
0.1uF
HALLB
R19
2k
R5
20
C5
0.1uF
HALLC
I2CCLOCK
I2CDATA
R16
10
R18
10
R2
10
PGGND
PGVDD
PGVPP
PGD
PGC
ICSP kontakt
SHUTDOWN
2k
R14
R7
10
R8
10
R9
10
R10
10
R11
10
R12
10
D1
D2
D3
IC1
IC2
IC3
HO
VDD
VB
HIN
VS
SD
LIN
VCC
VSS
COM
LO
IR2113SPBF
C13
0.1uF
C12
0.1uF
C11
0.1uF
C10
0.1uF
C9
0.1uF
C8
0.1uF
Header
X1-1
X1-2
X1-3
X1-4
X1-5
X1-6
X1-7
X1-8
X1-9
X1-10
X1-11
X1-12

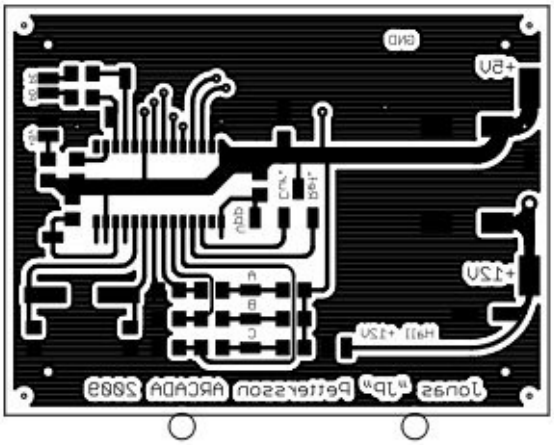
Jonas Pettersson	
3fas växelriktare	FC_v4_A3
Slutarbete 2008	18.12.2009 04:38:00
Max Pin 80V, 100A	Sheet: 1/1

Bilaga 2

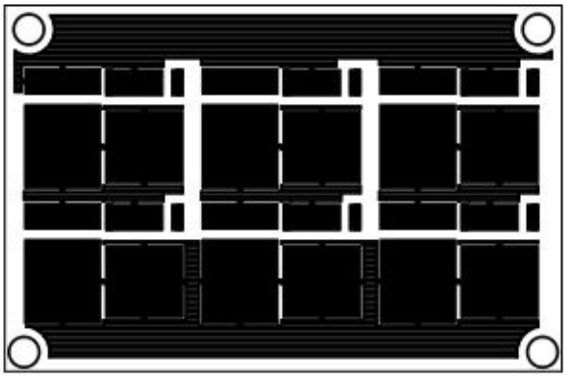
Styrkortets övre lager



Styrkortets bottenlager



Kraftelektronik kortets övre lager



Bilaga 3

```
//------------------------------------------------------------------
//                          Software License Agreement
// The software supplied herewith by Microchip Technology Incorporated
// (the "Company") for its PICmicro® Microcontroller is intended and
// supplied to you, the Company's customer, for use solely and
// exclusively on Microchip PICmicro Microcontroller products. The
// software is owned by the Company and/or its supplier, and is
// protected under applicable copyright laws. All rights are reserved.
//  Any use in violation of the foregoing restrictions may subject the
// user to criminal sanctions under applicable laws, as well as to
// civil liability for the breach of the terms and conditions of this
// license.
//
// THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES,
// WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED
// TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
// PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT,
// IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR
// CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
//
//------------------------------------------------------------------
//        File:             sinusoidalBLDC v1.2.c
//
//        Written By:    Jorge Zambada, Microchip Technology
//
// The following files should be included in the MPLAB project:
//
//                sinusoidalBLDC v1.2.c        -- Main source code file
//                SVM.c                        -- Space Vector Modulation file
//                SVM.h
//                p30f2010.gld                 -- Linker script file
//------------------------------------------------------------------
//
// Revision History
//
// July/5/2005 -- first version
// November/18/2005 -- Review
//------------------------------------------------------------------

#include "p30f2010.h"
#include "svm.h"

//-------------------------Device Configuration-----------------------
_FOSC(CSW_FSCM_OFF & XT_PLL16);
_FWDT(WDT_OFF);
_FBORPOR(PBOR_OFF & BORV_20 & PWRT_64 & MCLR_EN);
//------------------------------------------------------------------
```

Bilaga 3

```
//        Hurst Motor Terminals | MC LV PICDEM Board Connection
// ----------------------|--------------------------------
//        Ground Phase ---------|-- G
//        Phase Red    ---------|-- M1
//        Phase Black  ---------|-- M2
//        Phase White  ---------|-- M3
//        Hall White   ---------|-- HA
//        Hall Brown   ---------|-- HB
//        Hall Green   ---------|-- HC

typedef signed int SFRAC16;

#undef CLOSED_LOOP      // if defined the speed controller will be enabled
#define PHASE_ADVANCE    // for extended speed ranges this should be defined

#define FCY  20000000          // xtal = 5Mhz; PLLx16 -> 20 MIPS
#define FPWM 20000             // 20 kHz, so that no audible noise is present.
#define _10MILLISEC            10        // Used as a timeout with no hall effect sensors
                                         // transitions and Forcing steps according to the
                                         // actual position of the motor
#define _100MILLISEC 100               // after this time has elapsed, the motor is
                                         // consider stalled and it's stopped

// These Phase values represent the base Phase value of the sinewave for each
// one of the sectors (each sector is a translation of the hall effect sensors
// reading
#define PHASE_ZERO         57344
#define PHASE_ONE((PHASE_ZERO + 65536/6) % 65536)
#define PHASE_TWO          ((PHASE_ONE + 65536/6) % 65536)
#define PHASE_THREE        ((PHASE_TWO + 65536/6) % 65536)
#define PHASE_FOUR         ((PHASE_THREE + 65536/6) % 65536)
#define PHASE_FIVE         ((PHASE_FOUR + 65536/6) % 65536)

#define MAX_PH_ADV_DEG  35 // This value represents the maximum allowed phase *standard 40*
                                        // advance in electrical degrees. Set a value from
                                        // 0 to 60. This value will be used to calculate
                                         // phase advance only if PHASE_ADVANCE is defined

// This is the calculation from the required phase advance to the actual
// value to be multiplied by the speed of the motor. So, if PHASE_ADVANCE is
// enabled, a certain amount of shit angle will be added to the generated
// sine wave, up to a maximum of the specified value on MAX_PH_ADV_DEG. This
// maximum phase shift will be present when the MeasuredSpeed variable is a
// fractional 1.0 (for CW) or -1.0 (for CCW).
#define MAX_PH_ADV                 (int)(((float)MAX_PH_ADV_DEG / 360.0) * 65536.0)
```

Bilaga 3

```
#define HALLA        1        // Connected to RB3
#define HALLB        2        // Connected to RB4
#define HALLC        4        // Connected to RB5
#define CW    0                // Counter Clock Wise direction
#define CCW  1                 // Clock Wise direction
#define SWITCH_S2  (!PORTCbits.RC14) // Push button S2
#define CRUISE_SWITCH   (!PORTCbits.RC13) // Push button S2

// Period Calculation
// Period = (TMRClock * 60) / (RPM * Motor_Poles)
// For example>
// Motor_Poles = 12
// RPM = 600 (Max Speed)
// Period = ((20,000,000 / 64) * 60) / (1200 * 12) = 1304 (600rpm 2604)
// RPM = 1 (Min Speed)
// Period = ((20,000,000 / 64) * 60) / (1 * 12) = 1562500

#define MINPERIOD 3500    // For 1200 max rpm and 12 poles motor 1304
#define MAXPERIOD          1562500         // For 1 min rpm and 12 poles motor

// Use this MACRO when using floats to initialize signed 16-bit fractional
// variables
#define SFloat_To_SFrac16(Float_Value)     \
      ((Float_Value < 0.0) ? (SFRAC16)(32768 * (Float_Value) - 0.5) \
      : (SFRAC16)(32767 * (Float_Value) + 0.5))

void InitADC10(void);          // Initialization of ADC used for Speed Command
void InitMCPWM(void);       // Initialization for PWM at 20kHz, Center aligned,
                  // Complementary mode with 500 ns of deadtime
void InitTMR1(void); // Initialization for TIMER1 used for speed control
                  // and motor stalled protection
void InitTMR3(void); // Initialization for TIMER3 used as a timebase
                  // for the two input capture channels
void InitUserInt(void);          // This function initializes all ports
                  // (inputs and outputs) for the application
void InitICandCN(void);       // Initializes input captures and change notification,
                  // used for the hall sensor inputs
void RunMotor(void);// This function initializes all variables
                  // and interrupts used for starting and running
                  // the motor
void StopMotor(void);          // This function clears all flags, and stops anything
                  // related to motor control, and also disables PWMs
void SpeedControl(void);     // This function contains all ASM and C operations
                     // for doing the PID Control loop for the speed
void ForceCommutation(void); // When motor is to slow to generate interrupts
                     // on halls, this function forces a commutation
void ChargeBootstraps(void); // At the begining of the motor operation, the
```

Bilaga 3

```
                    // bootstrap caps are charged with this function

// Constants used for properly energizing the motor depending on the
// rotor's position
int PhaseValues[6] __attribute__((far,space(auto_psv)))=
{PHASE_ZERO, PHASE_ONE, PHASE_TWO, PHASE_THREE, PHASE_FOUR, PHASE_FIVE};

// In the sinewave generation algorithm we need an offset to be added to the
// pointer when energizing the motor in CCW. This is done to compensate an
// asymetry of the sinewave
int PhaseOffset = 0; //4100

// Flags used for the application
struct
{
        unsigned MotorRunning       :1;  // This bit is 1 if motor running
        unsigned unused                 :15;
}Flags;

unsigned int Phase;     // This variable is incremented by the PWM interrupt
                // in order to generate a proper sinewave. Its value
                // is incremented by a value of PhaseInc, which
                // represents the frequency of the generated sinewave
signed int PhaseInc; // Delta increments of the Phase variable, calculated
                 // in the TIMER1 interrupt (each 1 ms) and used in
                 // the PWM interrupt (each 50 us)
signed int PhaseAdvance; // Used for extending motor speed range. This value
                    // is added directly to the parameters passed to the
                    // SVM function (the sine wave generation subroutine)
unsigned int HallValue;          // This variable holds the hall sensor input readings
unsigned int Sector;  // This variables holds present sector value, which is
                 // the rotor position
unsigned int LastSector; // This variable holds the last sector value. This
                    // is critical to filter slow slew rate on the Hall
                    // effect sensors hardware
unsigned int MotorStalledCounter = 0; // This variable gets incremented each
                                // 1 ms, and is cleared everytime a new
                                // sector is detected. Used for
                                // ForceCommutation and MotorStalled
                                // protection functions

// This array translates the hall state value read from the digital I/O to the
// proper sector.  Hall values of 0 or 7 represent illegal values and therefore
// return -1.
char SectorTable[] = {-1,4,2,3,0,5,1,-1};

unsigned char Current_Direction;      // Current mechanical motor direction of
```

Bilaga 3

```
                        // rotation Calculated in halls interrupts
unsigned char Required_Direction;    // Required mechanical motor direction of
                        // rotation, will have the same sign as the
                                                        // ControlOutput variable from the
Speed
                        // Controller
unsigned char Cruise_Active=0;       // Cruise control on or off


// Variables containing the Period of half an electrical cycle, which is an
// interrupt each edge of one of the hall sensor input
unsigned int PastCapture, ActualCapture, Period;
// Used as a temporal variable to perform a fractional divide operation in
// assembly
SFRAC16 _MINPERIOD = MINPERIOD - 1;

SFRAC16 MeasuredSpeed, RefSpeed;         // Actual and Desired speeds for the PID
                // controller, that will generate the error
SFRAC16 ControlOutput = 0;          // Controller output, used as a voltage output,
                // use its sign for the required direction

SFRAC16 Cruise = 0;// Value that is locked with a switch for cruise control operation

// Absolute PID gains used by the controller. Position form implementation of
// a digital PID. See SpeedControl subroutine for details
SFRAC16 Kp = SFloat_To_SFrac16(0.01);   // P Gain
SFRAC16 Ki = SFloat_To_SFrac16(0.01);  // I Gain
SFRAC16 Kd = SFloat_To_SFrac16(0.000); // D Gain

// Constants used by the PID controller, since a MAC operation is used, the
// PID structure is changed (See SpeedControl() Comments)
SFRAC16 ControlDifference[3] \
      __attribute__((__space__(xmemory), __aligned__(4)));
SFRAC16 PIDCoefficients[3]   \
      __attribute__((__space__(ymemory), __aligned__(4)));

// Used as a temporal variable to perform a fractional divide operation in
// assembly
SFRAC16 _MAX_PH_ADV = MAX_PH_ADV;

SFRAC16 Temp;       // Temporal Variable



void __attribute__((interrupt, no_auto_psv)) _T1Interrupt (void)
{
        IFS0bits.T1IF = 0;
```

Bilaga 3

```
        Period = ActualCapture - PastCapture;  // This is an UNsigned substraction
                                // to get the Period between one
                                // hall effect sensor transition

    // These operations limit the Period value to a range from 60 to 6000 rpm
        if (Period < (unsigned int)MINPERIOD)  // MINPERIOD or 6000 rpm
                Period = MINPERIOD;
        else if (Period > (unsigned int)MAXPERIOD) // MAXPERIOD or 60 rpm
                Period = MAXPERIOD;

    // PhaseInc is a value added to the Phase variable to generate the sine
    // voltages. 1 electrical degree corresponds to a PhaseInc value of 184,
    // since the pointer to the sine table is a 16bit value, where 360 Elec
    // Degrees represents 65535 in the pointer.
    // __builtin_divud(Long Value, Int Value) is a function of the compiler
    // to do Long over Integer divisions.
        PhaseInc = __builtin_divud(512000UL, Period);      // Phase increment is used
                                                                                    // by
the PWM isr (SVM)

    // This subroutine in assembly calculates the MeasuredSpeed using
    // fractional division. These operations in assembly perform the following
    // formula:
    //                 MINPERIOD (in fractional)
    //  MeasuredSpeed = --------------------------
    //                  Period (in fractional)
    //

        __asm__ volatile("repeat #17\n\t"
                "divf %1,%2\n\t"
                "mov w0,%0" : /* output */ "=g"(MeasuredSpeed)
                        : /* input */ "r"(_MINPERIOD),
                                "e"(Period)
                        : /* clobber */ "w0");

    // MeasuredSpeed sign adjustment based on current motor direction of
    // rotation
        if (Current_Direction == CCW)
                MeasuredSpeed = -MeasuredSpeed;

    // The following values represent the MeasuredSpeed values from the
    // previous operations:
    //
        // CONDITION        RPM          SFRAC16      SINT      HEX
        // Max Speed CW  -> 6000 RPM  -> 0.996805  -> 32663  -> 0x7F97
        // Min Speed CW  -> 60 RPM    -> 0.009984  -> 327    -> 0x0147
        // Min Speed CCW -> -60 RPM   -> -0.009984 -> -327   -> 0xFEB9
```

Bilaga 3

```
        // Max Speed CCW -> -6000 RPM -> -0.996805 -> -32663 -> 0x8069

        SpeedControl(); // Speed PID controller is called here. It will use
              // MeasuredSpeed, RefSpeed, some buffers and will generate
              // the new ControlOutput, which represents a new amplitude
              // of the sinewave that will be generated by the SVM
              // subroutine.

#ifdef PHASE_ADVANCE
         // Calculate Phase Advance Based on Actual Speed and MAX_PH_ADV define
      // The following assembly instruction perform the following formula
      // using fractional multiplication:
      //
      // PhaseAdvance = MAX_PH_ADV * MeasuredSpeed
      //

#if !defined(__C30_VERSION__) || (__C30_VERSION__ < 200) || defined(TEST_ASM)
      {  register int wreg4 asm("w4") = _MAX_PH_ADV;
        register int wreg5 asm("w5") = MeasuredSpeed;

        asm volatile("mpy %0*%1, A" : /* no outputs */
                                : "r"(wreg4), "r"(wreg5));
        asm volatile("sac A, %0" : "=r"(PhaseAdvance));
      }
#else
      {  register int a_reg asm("A");

        a_reg = __builtin_mpy(_MAX_PH_ADV, MeasuredSpeed, 0,0,0,0,0,0);
        PhaseAdvance = __builtin_sac(a_reg,0);
      }
#endif

#endif

        MotorStalledCounter++;        // We increment a timeout variable to see if the
                    // motor is too slow (not generating hall effect
                    // sensors interrupts frequently enough) or if
                    // the motor is stalled. This variable is cleared
                    // in halls ISRs
   if ((MotorStalledCounter % _10MILLISEC) == 0)
        {
              ForceCommutation(); // Force Commutation if no hall sensor changes
                    // have occured in specified timeout.
        }
        else if (MotorStalledCounter >= _100MILLISEC)
        {
              StopMotor(); // Stop motor is no hall changes have occured in
```

```
                // specified timeout
        }
        return;
}



void __attribute__((interrupt, no_auto_psv)) _CNInterrupt (void)
{
        IFS0bits.CNIF = 0;     // Clear interrupt flag
        HallValue = (unsigned int)((PORTB >> 3) & 0x0007);        // Read halls
        Sector = SectorTable[HallValue];       // Get Sector from table

   // This MUST be done for getting around the HW slow rate
        if (Sector != LastSector)
        {
                // Since a new sector is detected, clear variable that would stop
      // the motor if stalled.
                MotorStalledCounter = 0;

                // Motor current direction is computed based on Sector
                if ((Sector == 5) || (Sector == 2))
                        Current_Direction = CCW;
                else
                        Current_Direction = CW;

      // Motor commutation is actually based on the required direction, not
      // the current dir. This allows driving the motor in four quadrants
                if (Required_Direction == CW)
                {
                        Phase = PhaseValues[Sector];
                }
                else
                {
                        // For CCW an offset must be added to compensate difference in
        // symmetry of the sine table used for CW and CCW
                        Phase = PhaseValues[(Sector + 3) % 6] + PhaseOffset;
                }
                LastSector = Sector; // Update last sector
        }

        return;
}


void __attribute__((interrupt, no_auto_psv)) _IC7Interrupt (void)
{
```

Bilaga 3

```
        IFS1bits.IC7IF = 0;     // Cleat interrupt flag
        HallValue = (unsigned int)((PORTB >> 3) & 0x0007);        // Read halls
        Sector = SectorTable[HallValue];       // Get Sector from table

    // This MUST be done for getting around the HW slow rate
        if (Sector != LastSector)
        {
                // Calculate Hall period corresponding to half an electrical cycle
                PastCapture = ActualCapture;
                ActualCapture = IC7BUF;
                IC7BUF;
                IC7BUF;
                IC7BUF;

                // Since a new sector is detected, clear variable that would stop
      // the motor if stalled.
                MotorStalledCounter = 0;

                // Motor current direction is computed based on Sector
                if ((Sector == 3) || (Sector == 0))
                        Current_Direction = CCW;
                else
                        Current_Direction = CW;

      // Motor commutation is actually based on the required direction, not
      // the current dir. This allows driving the motor in four quadrants
                if (Required_Direction == CW)
                {
                        Phase = PhaseValues[Sector];
                }
                else
                {
                        // For CCW an offset must be added to compensate difference in
        // symmetry of the sine table used for CW and CCW
                        Phase = PhaseValues[(Sector + 3) % 6] + PhaseOffset;
                }
                LastSector = Sector; // Update last sector
        }

        return;
}


void __attribute__((interrupt, no_auto_psv)) _IC8Interrupt (void)
{
        IFS1bits.IC8IF = 0;     // Cleat interrupt flag
        HallValue = (unsigned int)((PORTB >> 3) & 0x0007);        // Read halls
```

Bilaga 3

```
        Sector = SectorTable[HallValue];       // Get Sector from table

   // This MUST be done for getting around the HW slow rate
        if (Sector != LastSector)
        {
                // Since a new sector is detected, clear variable that would stop
      // the motor if stalled.
                MotorStalledCounter = 0;

                // Motor current direction is computed based on Sector
                if ((Sector == 1) || (Sector == 4))
                        Current_Direction = CCW;
                else
                        Current_Direction = CW;

      // Motor commutation is actually based on the required direction, not
      // the current dir. This allows driving the motor in four quadrants
                if (Required_Direction == CW)
                {
                        Phase = PhaseValues[Sector];
                }
                else
                {
                        // For CCW an offset must be added to compensate difference in
        // symmetry of the sine table used for CW and CCW
                        Phase = PhaseValues[(Sector + 3) % 6] + PhaseOffset;
                }
                LastSector = Sector; // Update last sector
        }

        return;
}



void __attribute__((interrupt, no_auto_psv)) _PWMInterrupt (void)
{
        IFS2bits.PWMIF = 0; // Clear interrupt flag

        if (Required_Direction == CW)
        {
                if (Current_Direction == CW)
                        Phase += PhaseInc;    // Increment Phase if CW to generate the
                      // sinewave only if both directions are equal
                // If Required_Direction is CW (forward) POSITIVE voltage is applied
                #ifdef PHASE_ADVANCE
                SVM(ControlOutput, Phase + PhaseAdvance);         // PhaseAdvance addition
```

Bilaga 3

```
                                                                                                    //
produces the sinewave
                                                                                                    //
phase shift
                #else
                SVM(ControlOutput, Phase);
                #endif
        }
        else
        {
                if (Current_Direction == CCW)
                        Phase -= PhaseInc;      // Decrement Phase if CCW to generate
                                                                        // the sinewave only if both
                                                                        // directions are equal
                // If Required_Direction is CCW (reverse) NEGATIVE voltage is applied
                #ifdef PHASE_ADVANCE

                if (MeasuredSpeed > 0,3)
                {
                        SVM(-(ControlOutput+1), Phase + PhaseAdvance);
                }

                else
                {
                        SVM(-(ControlOutput+1), Phase);// PhaseAdvance addition
                }                                                                                   //
produces the sinewave
                                                                                                    //
phase shift
                #else
                SVM(-(ControlOutput+1), Phase);
                #endif
        }
        return;
}



void __attribute__((interrupt, no_auto_psv)) _ADCInterrupt (void)
{
        IFS0bits.ADIF = 0;     // Clear interrupt flag
        RefSpeed = ADCBUF0; // Read POT value to set Reference Speed

        if(RefSpeed!=0)
        RefSpeed = -RefSpeed;
```

Bilaga 3

```
        if (!Flags.MotorRunning)
        {
                if (RefSpeed < 50)
                RunMotor();

        }



        return;
}


int main(void)
{
        InitUserInt();   // Initialize User Interface I/Os
        InitADC10();  // Initialize ADC to be signed fractional
        InitTMR1();             // Initialize TMR1 for 1 ms periodic ISR
        InitTMR3();             // Initialize TMR3 for timebase of capture
        InitICandCN();          // Initialize Hall sensor inputs ISRs
        InitMCPWM();            // Initialize PWM @ 20 kHz, center aligned, 500 ns of
              // deadtime
        for(;;)
        {
                /*if ((SWITCH_S2) && (!Flags.MotorRunning))
                {
                        while(SWITCH_S2);
                        RunMotor();    // Run motor if push button is pressed and motor is
                // stopped
                }
                //else if ((SWITCH_S2) && (Flags.MotorRunning))
                {
                        while(SWITCH_S2);
                        StopMotor();// Stop motor if push button is pressed and motor is
                // running
                }*/
        }
        return 0;
}


void ChargeBootstraps(void)
{
        unsigned int i;
        OVDCON = 0x0015;  // Turn ON low side transistors to charge
        for (i = 0; i < 33330; i++) // 10 ms Delay at 20 MIPs
                ;
```

Bilaga 3

```
        PWMCON2bits.UDIS = 1;
        PDC1 = PTPER;        // Initialize as 0 voltage
        PDC2 = PTPER;        // Initialize as 0 voltage
        PDC3 = PTPER;        // Initialize as 0 voltage
        OVDCON = 0x3F00; // Configure PWM0-5 to be governed by PWM module
        PWMCON2bits.UDIS = 0;
        return;
}


void RunMotor(void)
{
        ChargeBootstraps();
        // init variables
        ControlDifference[0] = 0;      // Error at K     (most recent)
        ControlDifference[1] = 0;      // Error at K-1
        ControlDifference[2] = 0;      // Error at K-2 (least recent)
        PIDCoefficients[0] = Kp + Ki + Kd;  // Modified coefficient for using MACs
        PIDCoefficients[1] = -(Kp + 2*Kd);  // Modified coefficient for using MACs
        PIDCoefficients[2] = Kd;                        // Modified coefficient for using MACs

        TMR1 = 0;                       // Reset timer 1 for speed control
        TMR3 = 0;                       // Reset timer 3 for speed measurement
        ActualCapture = MAXPERIOD;      // Initialize captures for minimum speed
                                                                //(60 RPMs)
        PastCapture = 0;

        // Initialize direction with required direction
        // Remember that ADC is not stopped.
        HallValue = (unsigned int)((PORTB >> 3) & 0x0007);        // Read halls
        LastSector = Sector = SectorTable[HallValue];         // Initialize Sector
                                        // variable

        // RefSpeed's sign will determine if the motor should be run at CW
   // (+RefSpeed) or CCW (-RefSpeed) ONLY at start up, since when the motor
   // has started, the required direction will be set by the control output
   // variable to be able to operate in the four quadrants
        if (RefSpeed < 0)
        {
                ControlOutput = 0;     // Initial output voltage
                Current_Direction = Required_Direction = CCW;
                Phase = PhaseValues[(Sector + 3) % 6] + PhaseOffset;
        }
        /*else
        {
                ControlOutput = 0;     // Initial output voltage
                Current_Direction = Required_Direction = CW;
```

Bilaga 3

```
                Phase = PhaseValues[Sector];
        }*/

        MotorStalledCounter = 0;       // Reset motor stalled protection counter
        // Set initial Phase increment with minimum value. This will change if a
    // costing operation is required by the application
        PhaseInc = __builtin_divud(512000UL, MAXPERIOD);

        // Clear all interrupts flags
        IFS0bits.T1IF = 0;      // Clear timer 1 flag
        IFS0bits.CNIF = 0;     // Clear interrupt flag
        IFS1bits.IC7IF = 0;    // Clear interrupt flag
        IFS1bits.IC8IF = 0;    // Clear interrupt flag
        IFS2bits.PWMIF = 0; // Clear interrupt flag

        // enable all interrupts
        __asm__ volatile ("DISI #0x3FFF");
        IEC0bits.T1IE = 1;     // Enable interrupts for timer 1
        IEC0bits.CNIE = 1;    // Enable interrupts on CN5
        IEC1bits.IC7IE = 1;    // Enable interrupts on IC7
        IEC1bits.IC8IE = 1;    // Enable interrupts on IC8
        IEC2bits.PWMIE = 1;// Enable PWM interrupts
    DISICNT = 0;

        Flags.MotorRunning = 1;       // Indicate that the motor is running
        return;
}


void StopMotor(void)
{
        OVDCON = 0x0000;  // turn OFF every transistor

        // disable all interrupts
        __asm__ volatile ("DISI #0x3FFF");
        IEC0bits.T1IE = 0;     // Disable interrupts for timer 1
        IEC0bits.CNIE = 0;    // Disable interrupts on CN5
        IEC1bits.IC7IE = 0;    // Disable interrupts on IC7
        IEC1bits.IC8IE = 0;    // Disable interrupts on IC8
        IEC2bits.PWMIE = 0;// Disable PWM interrupts
    DISICNT = 0;

        Flags.MotorRunning = 0;       // Indicate that the motor has been stopped
        return;
}
```

Bilaga 3

```
#if !defined(__C30_VERSION__) || (__C30_VERSION__ < 200)
void SpeedControl(void)
{       register SFRAC16 *ControlDifferencePtr asm("w8") = ControlDifference;
        register SFRAC16 *PIDCoefficientsPtr asm("w10") = PIDCoefficients;
        register SFRAC16 x_prefetch asm("w4");
        register SFRAC16 y_prefetch asm("w5");

        CORCONbits.SATA = 1;       // Enable Saturation on Acc A
        // Calculate most recent error with saturation, no limit checking required
        __asm__ volatile ("LAC %0, A" : /* no outputs */ : "r"(RefSpeed));
        __asm__ volatile ("LAC %0, B" : /* no outputs */ : "r"(MeasuredSpeed));
        __asm__ volatile ("SUB A");
        __asm__ volatile ("SAC A, [%0]" : /* no outputs */ :
                                "r"(ControlDifferencePtr));
        // Prepare MAC Operands
        __asm__ volatile ("MOVSAC A, [%0]+=2, %2, [%1]+=2, %3" :
                        /* outputs */ "+r"(ControlDifferencePtr),
                                "+r"(PIDCoefficientsPtr),
                                "=r"(x_prefetch),
                                "=r"(y_prefetch));
        __asm__ volatile ("LAC %0, A" : /* no outpus */ : "r"(ControlOutput));                          //
Load Acc with last output
        // Perform MAC
        __asm__ volatile ("REPEAT #2\n\t"
                        "MAC %2*%3, A, [%0]+=2, %2, [%1]+=2, %3" :
                        /* outputs */ "+r"(ControlDifferencePtr),
                                "+r"(PIDCoefficientsPtr),
                                "+r"(x_prefetch),
                                "+r"(y_prefetch));
        // Store result in ControlOutput with saturation
        __asm__ volatile ("SAC A, %0" : "=r"(ControlOutput));
        CORCONbits.SATA = 0;       // Disable Saturation on Acc A
        // Store last 2 errors
        ControlDifference[2] = ControlDifference[1];
        ControlDifference[1] = ControlDifference[0];

        // If CLOSED_LOOP is undefined (running open loop) overide ControlOutput
        // with value read from the external potentiometer
        #ifndef CLOSED_LOOP
                ControlOutput = RefSpeed;
        #endif

        // ControlOutput will determine the motor required direction
        if (ControlOutput < 0)
                Required_Direction = CCW;
        //else
        //      Required_Direction = CW;
```

```
        return;
}
#else
        // In version 2.0 and greater; there is no need to claim specific
        // registers, new constraint letters allow the compiler to make a
        // choice (also, new builtins mean you don't have to resort to
        // assembly for this kind of operation, see the next function
void SpeedControl_using_inline_asm(void)
{       SFRAC16 *ControlDifferencePtr = ControlDifference;
      SFRAC16 *PIDCoefficientsPtr = PIDCoefficients;
      register SFRAC16 x_prefetch asm("w4");
      register SFRAC16 y_prefetch;

      CORCONbits.SATA = 1;    // Enable Saturation on Acc A
      // Calculate most recent error with saturation, no limit checking required
      __asm__ volatile ("LAC %0, A" : /* no outputs */ : "r"(RefSpeed));
      __asm__ volatile ("LAC %0, B" : /* no outputs */ : "r"(MeasuredSpeed));
      __asm__ volatile ("SUB A");
      __asm__ volatile ("SAC A, [%0]" : /* no outputs */ :
                              "r"(ControlDifferencePtr));
      // Prepare MAC Operands
      __asm__ volatile ("MOVSAC A, [%0]+=2, %2, [%1]+=2, %3" :
                        /* outputs */ "+x"(ControlDifferencePtr),
                                    "+y"(PIDCoefficientsPtr),
                                    "=z"(x_prefetch),
                                    "=z"(y_prefetch));
      __asm__ volatile ("LAC %0, A" : /* no outpus */ : "r"(ControlOutput));                // Load Acc
with last output
      // Perform MAC
      __asm__ volatile ("REPEAT #2\n\t"
                  "MAC %2*%3, A, [%0]+=2, %2, [%1]+=2, %3" :
                        /* outputs */ "+x"(ControlDifferencePtr),
                                    "+y"(PIDCoefficientsPtr),
                                    "+z"(x_prefetch),
                                    "+z"(y_prefetch));
      // Store result in ControlOutput with saturation
      __asm__ volatile ("SAC A, %0" : "=r"(ControlOutput));
      CORCONbits.SATA = 0;    // Disable Saturation on Acc A
      // Store last 2 errors
      ControlDifference[2] = ControlDifference[1];
      ControlDifference[1] = ControlDifference[0];

      // If CLOSED_LOOP is undefined (running open loop) overide ControlOutput
      // with value read from the external potentiometer
      #ifndef CLOSED_LOOP
            ControlOutput = RefSpeed;
      #endif
```

Bilaga 3

```
    // ControlOutput will determine the motor required direction
    if (ControlOutput < 0)
        Required_Direction = CCW;
    else
        Required_Direction = CW;
    return;
}

void SpeedControl(void) {
    SFRAC16 *ControlDifferencePtr = ControlDifference;
    SFRAC16 *PIDCoefficientsPtr = PIDCoefficients;
    SFRAC16 x_prefetch;
    SFRAC16 y_prefetch;

      register int reg_a asm("A");
      register int reg_b asm("B");

      CORCONbits.SATA = 1;    // Enable Saturation on Acc A
      reg_a = __builtin_lac(RefSpeed,0);
      reg_b = __builtin_lac(MeasuredSpeed,0);
      reg_a = __builtin_subab();
      __asm__ volatile ("PUSH W0");
      __asm__ volatile ("SAC A, #0, W0");
      __asm__ volatile ("MOV W0, _Temp");
      __asm__ volatile ("POP W0");
      *ControlDifferencePtr = Temp;
      reg_a = __builtin_movsac(&ControlDifferencePtr, &x_prefetch, 2,
                               &PIDCoefficientsPtr, &y_prefetch, 2, 0);
      reg_a = __builtin_lac(ControlOutput, 0);
      reg_a = __builtin_mac(x_prefetch,y_prefetch,
                &ControlDifferencePtr, &x_prefetch, 2,
                &PIDCoefficientsPtr, &y_prefetch, 2, 0);
      reg_a = __builtin_mac(x_prefetch,y_prefetch,
                &ControlDifferencePtr, &x_prefetch, 2,
                &PIDCoefficientsPtr, &y_prefetch, 2, 0);
      reg_a = __builtin_mac(x_prefetch,y_prefetch,
                &ControlDifferencePtr, &x_prefetch, 2,
                &PIDCoefficientsPtr, &y_prefetch, 2, 0);
            __asm__ volatile ("PUSH W0");
            __asm__ volatile ("SAC A, #0, W0");
            __asm__ volatile ("MOV W0, _Temp");
            __asm__ volatile ("POP W0");
            ControlOutput = Temp;
    CORCONbits.SATA = 0;    // Disable Saturation on Acc A
    // Store last 2 errors
    ControlDifference[2] = ControlDifference[1];
```

Bilaga 3

```
	ControlDifference[1] = ControlDifference[0];

	// If CLOSED_LOOP is undefined (running open loop) overide ControlOutput
	// with value read from the external potentiometer
	#ifndef CLOSED_LOOP
		ControlOutput = RefSpeed;
	#endif

	// ControlOutput will determine the motor required direction
	if (ControlOutput < 0)
		Required_Direction = CCW;
	//else
	 //	Required_Direction = CW;
	return;
}
#endif


void ForceCommutation(void)
{
	HallValue = (unsigned int)((PORTB >> 3) & 0x0007);	// Read halls
	Sector = SectorTable[HallValue];	// Read sector based on halls
	if (Sector != -1)		// If the sector is invalid don't do anything
	{
		// Depending on the required direction, a new phase is fetched
		if (Required_Direction == CW)
		{
			// Motor is required to run forward, so read directly the table
			Phase = PhaseValues[Sector];
		}
		else
		{
			// Motor is required to run reverse, so calculate new phase and
	// add offset to compensate asymmetries
			Phase = PhaseValues[(Sector + 3) % 6] + PhaseOffset;
		}
	}
	return;
}


void InitADC10(void)
{

	ADPCFG = 0x0038;				// RB3, RB4, and RB5 are digital
	ADCON1 = 0x036E;				// PWM starts conversion
							// Signed fractional conversions
```

Bilaga 3

```
        ADCON2 = 0x0000;

        ADCHS = 0x0002;                                 // Pot is connected to AN2

        ADCON3 = 0x0003;
        IFS0bits.ADIF = 0;                      // Clear ISR flag
        IEC0bits.ADIE = 1;                      // Enable interrupts

        ADCON1bits.ADON = 1;                    // turn ADC ON
        return;
}


void InitMCPWM(void)
{
        TRISE = 0x0100;       // PWM pins as outputs, and FLTA as input
        PTPER = (FCY/FPWM - 1) >> 1;     // Compute Period based on CPU speed and
                        // required PWM frequency (see defines)
        OVDCON = 0x0000;  // Disable all PWM outputs.
        DTCON1 = 0x0008;   // ~500 ns of dead time 8 orginal
        PWMCON1 = 0x0077;         // Enable PWM output pins and configure them as
                // complementary mode
        PDC1 = PTPER;           // Initialize as 0 voltage
        PDC2 = PTPER;           // Initialize as 0 voltage
        PDC3 = PTPER;           // Initialize as 0 voltage
        SEVTCMP = 1;            // Enable triggering for ADC
        PWMCON2 = 0x0F02;         // 16 postscale values, for achieving 20 kHz
        PTCON = 0x8002;               // start PWM as center aligned mode
        return;
}


void InitICandCN(void)
{
        //Hall A -> CN5. Hall A is only used for commutation.
        //Hall B -> IC7. Hall B is used for Speed measurement and commutation.
        //Hall C -> IC8. Hall C is only used for commutation.

        // Init Input change notification 5
        TRISB |= 0x38;                  // Ensure that hall connections are inputs
        CNPU1 = 0;              // Disable all CN pull ups
        CNEN1 = 0x20;                   // Enable CN5
        IFS0bits.CNIF = 0;    // Clear interrupt flag

        // Init Input Capture 7
        IC7CON = 0x0001;    // Input capture every edge with interrupts and TMR3
        IFS1bits.IC7IF = 0;    // Clear interrupt flag
```

Bilaga 3

```
	// Init Input Capture 8
	IC8CON = 0x0001;    // Input capture every edge with interrupts and TMR3
	IFS1bits.IC8IF = 0;    // Clear interrupt flag

	return;
}



void InitTMR1(void)
{
	T1CON = 0x0020;			// internal Tcy/64 clock
	TMR1 = 0;
	PR1 = 313;			// 1 ms interrupts for 20 MIPS
	T1CONbits.TON = 1;		// turn on timer 1
	return;
}



void InitTMR3(void)
{
	T3CON = 0x0020;			// internal Tcy/64 clock
	TMR3 = 0;
	PR3 = 0xFFFF;
	T3CONbits.TON = 1;		// turn on timer 3
	return;
}


void InitUserInt(void)
{
	TRISC |= 0x6000;	// S2/RC14 as input
	// Analog pin for POT already initialized in ADC init subroutine
	PORTF = 0x0008;		// RS232 Initial values
	TRISF = 0xFFF7;		// TX as output
	return;
}

// End of SinusoidalBLDC v1.2.c
```

Bilaga 4

```
/*********************************************************************
*                                                                    *
*                    Software License Agreement                      *
*                                                                    *
*   The software supplied herewith by Microchip Technology           *
*   Incorporated (the "Company") for its dsPIC controller            *
*   is intended and supplied to you, the Company's customer,         *
*   for use solely and exclusively on Microchip dsPIC                *
*   products. The software is owned by the Company and/or its        *
*   supplier, and is protected under applicable copyright laws. All  *
*   rights are reserved. Any use in violation of the foregoing       *
*   restrictions may subject the user to criminal sanctions under    *
*   applicable laws, as well as to civil liability for the breach of *
*   the terms and conditions of this license.                        *
*                                                                    *
*   THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION.  NO           *
*   WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING,    *
*   BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND    *
*   FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE     *
*   COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL,  *
*   INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.  *
*                                                                    *
*********************************************************************/

/*********************************************************************
*                                                                    *
*   Author: Steve Bowling                                            *
*                                                                    *
*   Filename:       svm.c                                            *
*   Date:           3/2/04                                           *
*   File Version:   1.00                                             *
*                                                                    *
*   Tools used:    Compiler  -> 1.10                                 *
*                                                                    *
*   Linker File:    p30f2010.gld                                     *
*                                                                    *
*                                                                    *
*********************************************************************
*       Code Description
*
*  This file implements 3-phase space vector modulation.
*********************************************************************/

#include <p30fxxxx.h>
#include <svm.h>

//-----------------------------------------------------------------
```

Bilaga 4

```
// These are the definitions for various angles used in the SVM
// routine.  A 16-bit unsigned value is used as the angle variable.
// The SVM algorithm determines the 60 degree sector
#define     VECTOR1     0                       // 0 degrees
#define     VECTOR2     0x2aaa          // 60 degrees
#define     VECTOR3     0x5555                  // 120 degrees
#define     VECTOR4     0x8000                  // 180 degrees
#define     VECTOR5     0xaaaa          // 240 degrees
#define     VECTOR6     0xd555                  // 300 degrees
#define     SIXTY_DEG   0x2aaa

//---------------------------------------------------------------------

// This is the maximum value that may be passed to the SVM
// function without overmodulation.  This limit is equivalent
// to 0.866, which is sqrt(3)/2.
#define VOLTS_LIMIT     30000 //28300 orginalvärde

// This sinewave lookup table has 171 entries.  (1024 points per
// electrical cycle -- 1024*(60/360) = 171)
// The table covers 60 degrees of the sine function.

int sinetable[] __attribute__((far,space(auto_psv)))=
{0,201,401,602,803,1003,1204,1404,1605,1805,
2005,2206,2406,2606,2806,3006,3205,3405,3605,3804,4003,4202,4401,4600,
4799,4997,5195,5393,5591,5789,5986,6183,6380,6577,6773,6970,7166,7361,
7557,7752,7947,8141,8335,8529,8723,8916,9109,9302,9494,9686,9877,10068,
10259,10449,10639,10829,11018,11207,11395,11583,11771,11958,12144,
12331,12516,12701,12886,13070,13254,13437,13620,13802,13984,14165,
14346,14526,14706,14885,15063,15241,15419,15595,15772,15947,16122,
16297,16470,16643,16816,16988,17159,17330,17500,17669,17838,18006,
18173,18340,18506,18671,18835,18999,19162,19325,19487,19647,19808,
19967,20126,20284,20441,20598,20753,20908,21062,21216,21368,21520,
21671,21821,21970,22119,22266,22413,22559,22704,22848,22992,23134,
23276,23417,23557,23696,23834,23971,24107,24243,24377,24511,24644,
24776,24906,25036,25165,25293,25420,25547,25672,25796,25919,26042,
26163,26283,26403,26521,26638,26755,26870,26984,27098,27210,27321,
27431,27541,27649,27756,27862,27967,28071,28174,28276,28377};

//---------------------------------------------------------------------
// The function SVM() determines which sector the input angle is
// located in.  Then, the modulation angle is normalized to the current
// 60 degree sector.  Two angles are calculated from the normalized
// angle.  These two angles determine the times for the T1 and T2
// vectors.  The T1 and T2 vectors are then scaled by the modulation
// amplitude and the duty cycle range.  Finally, the T0 vector time
// is the time left over in the PWM counting period.
```

Bilaga 4

```
// The SVM() function then calculates three duty cycle values based
// on the T0, T1, and T2 times.  The duty cycle calculation depends
// on the
// appropriate duty cycle values depending on the type of SVM to be
// generated.
//---------------------------------------------------------------------

void SVM(int volts, unsigned int angle)
{
// These variables hold the normalized sector angles used to find
// t1, t2.
unsigned int angle1, angle2;

// These variables hold the space vector times.
unsigned int half_t0,t1,t2,tpwm;

// Calculate the total PWM count period, which is twice the value
// in the PTPER register.
tpwm = PTPER << 1;

// Limit volts input to avoid overmodulation.
if(volts > VOLTS_LIMIT) volts = VOLTS_LIMIT;

if(angle < VECTOR2)
        {
        angle2 = angle - VECTOR1;            // Reference SVM angle to the current
                        // sector
        angle1 = SIXTY_DEG - angle2;       // Calculate second angle referenced to
                        // sector

        t1 = sinetable[(unsigned char)(angle1 >> 6)];          // Look up values from
                                        // table.
        t2 = sinetable[(unsigned char)(angle2 >> 6)];

        // Scale t1 to by the volts variable.
        t1 = ((long)t1*(long)volts) >> 15;
        // Scale t1 for the duty cycle range.
        t1 = ((long)t1*(long)tpwm) >> 15;
        // Scale t2 time
        t2 = ((long)t2*(long)volts) >> 15;
        t2 = ((long)t2*(long)tpwm) >> 15;

        half_t0 = (tpwm - t1 - t2) >> 1;                  // Calculate half_t0 null time from
                                // period and t1,t2

        // Calculate duty cycles for Sector 1  (0 - 59 degrees)
        PDC1 = t1 + t2 + half_t0;
```

```
        PDC2 = t2 + half_t0;
        PDC3 = half_t0;
        }

else if(angle < VECTOR3)
        {
        angle2 = angle - VECTOR2;           // Reference SVM angle to the current
                        // sector
        angle1 = SIXTY_DEG - angle2;        // Calculate second angle referenced to
                        // sector

        t1 = sinetable[(unsigned char)(angle1 >> 6)];          // Look up values from
                                    // table.
        t2 = sinetable[(unsigned char)(angle2 >> 6)];

        // Scale t1 to by the volts variable.
        t1 = ((long)t1*(long)volts) >> 15;
        // Scale t1 for the duty cycle range.
        t1 = ((long)t1*(long)tpwm) >> 15;
        // Scale t2 time
        t2 = ((long)t2*(long)volts) >> 15;
        t2 = ((long)t2*(long)tpwm) >> 15;

        half_t0 = (tpwm - t1 - t2) >> 1;                  // Calculate half_t0 null time from
                                // period and t1,t2

        // Calculate duty cycles for Sector 2  (60 - 119 degrees)
        PDC1 = t1 + half_t0;
        PDC2 = t1 + t2 + half_t0;
        PDC3 = half_t0;
        }

else if(angle < VECTOR4)
        {
        angle2 = angle - VECTOR3;           // Reference SVM angle to the current
                        // sector
        angle1 = SIXTY_DEG - angle2;        // Calculate second angle referenced to
                        // sector

        t1 = sinetable[(unsigned char)(angle1 >> 6)];          // Look up values from
                                    // table.
        t2 = sinetable[(unsigned char)(angle2 >> 6)];

        // Scale t1 to by the volts variable.
        t1 = ((long)t1*(long)volts) >> 15;
        // Scale t1 for the duty cycle range.
        t1 = ((long)t1*(long)tpwm) >> 15;
```

Bilaga 4

```
        // Scale t2 time
        t2 = ((long)t2*(long)volts) >> 15;
        t2 = ((long)t2*(long)tpwm) >> 15;

        half_t0 = (tpwm - t1 - t2) >> 1;                    // Calculate half_t0 null time from
                                    // period and t1,t2

        // Calculate duty cycles for Sector 3  (120 - 179 degrees)
        PDC1 = half_t0;
        PDC2 = t1 + t2 + half_t0;
        PDC3 = t2 + half_t0;
        }

else if(angle < VECTOR5)
        {
        angle2 = angle - VECTOR4;             // Reference SVM angle to the current
                            // sector
        angle1 = SIXTY_DEG - angle2;          // Calculate second angle referenced to
                            // sector

        t1 = sinetable[(unsigned char)(angle1 >> 6)];           // Look up values from
                                        // table.
        t2 = sinetable[(unsigned char)(angle2 >> 6)];

        // Scale t1 to by the volts variable.
        t1 = ((long)t1*(long)volts) >> 15;
        // Scale t1 for the duty cycle range.
        t1 = ((long)t1*(long)tpwm) >> 15;
        // Scale t2 time
        t2 = ((long)t2*(long)volts) >> 15;
        t2 = ((long)t2*(long)tpwm) >> 15;

        half_t0 = (tpwm - t1 - t2) >> 1;                    // Calculate half_t0 null time from
                                    // period and t1,t2

        // Calculate duty cycles for Sector 4  (180 - 239 degrees)
        PDC1 = half_t0;
        PDC2 = t1 + half_t0;
        PDC3 = t1 + t2 + half_t0;
        }

else if(angle < VECTOR6)
        {
        angle2 = angle - VECTOR5;             // Reference SVM angle to the current
                            // sector
        angle1 = SIXTY_DEG - angle2;          // Calculate second angle referenced to
                            // sector
```

Bilaga 4

```
        t1 = sinetable[(unsigned char)(angle1 >> 6)];          // Look up values from
                                        // table.
        t2 = sinetable[(unsigned char)(angle2 >> 6)];

        // Scale t1 to by the volts variable.
        t1 = ((long)t1*(long)volts) >> 15;
        // Scale t1 for the duty cycle range.
        t1 = ((long)t1*(long)tpwm) >> 15;
        // Scale t2 time
        t2 = ((long)t2*(long)volts) >> 15;
        t2 = ((long)t2*(long)tpwm) >> 15;

        half_t0 = (tpwm - t1 - t2) >> 1;                  // Calculate half_t0 null time from
                                // period and t1,t2

        // Calculate duty cycles for Sector 5  (240 - 299 degrees)
        PDC1 = t2 + half_t0;
        PDC2 = half_t0;
        PDC3 = t1 + t2 + half_t0;
        }

else
        {
        angle2 = angle - VECTOR6;             // Reference SVM angle to the current
                        // sector
        angle1 = SIXTY_DEG - angle2;          // Calculate second angle referenced to
                        // sector

        t1 = sinetable[(unsigned char)(angle1 >> 6)];          // Look up values from
                                        // table.
        t2 = sinetable[(unsigned char)(angle2 >> 6)];

        // Scale t1 to by the volts variable.
        t1 = ((long)t1*(long)volts) >> 15;
        // Scale t1 for the duty cycle range.
        t1 = ((long)t1*(long)tpwm) >> 15;
        // Scale t2 time
        t2 = ((long)t2*(long)volts) >> 15;
        t2 = ((long)t2*(long)tpwm) >> 15;

        half_t0 = (tpwm - t1 - t2) >> 1;                  // Calculate half_t0 null time from
                                // period and t1,t2

        // Calculate duty cycles for Sector 6  ( 300 - 359 degrees )
        PDC1 = t1 + t2 + half_t0;
        PDC2 = half_t0;
```

Bilaga 4

```
        PDC3 = t1 + half_t0;
        }
}                               // end SVM()
```