ISSN 0280-5316
ISRN LUTFD2/TFRT--5743--SE

ABS på radiostyrd bil

Arne Hörberg

Department of Automatic Control
Lund Institute of Technology
May 2005

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
May 2005

Document Number
ISRN LUTFD2/TFRT--5743--SE

Author(s)
Arne Hörberg

Supervisor
Anders Blomdell and Karl-Erik Årzén at Department of Automatic Control in Lund

Sponsoring organization

Title and subtitle
ABS på radiostyrd bil (ABS for a Radiocontroled Car)

Abstract
This master thesis concerns the analysis and implementation of a tire slip controller for an off-the-shelf RC-model car. The Anti-lock brake system was implemented using two Atmel microcontrollers programmed in C, and sensors on all four wheels in additional to the standard electronics of the model car. The control strategy used was PI-methodology.'
The car is able to accelerate and brake at a specified wheel slip. However satisfying performance has not been reached in all different cases of operation due to limitations in the hardware of the car.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
Swedish

Number of pages
68

Recipient's notes

Security classification

Innehåll

Kapitel 1

Bakgrund

Ett av de absolut viktigaste säkerhetssystemen i hjulburna fordon är bromssystemet, och en av de viktigaste aspekterna för att säkerställa en effektiv bromsning på hala och våta underlag är att förhindra att hjulen låser sig. Friktionskoefficienten mellan däck och underlag är oftast betydligt mindre på ett låst hjul än på ett som fortfarande rullar och, kanske ännu viktigare; när hjulen låser sig blir bilen omöjligt att styra och föraren kan helt tappa kontrollen över fordonet [Drakunov et al.,1995]. För att förhindra hjullåsning utvecklades Antilock Braking System (ABS).

1.1 ABS

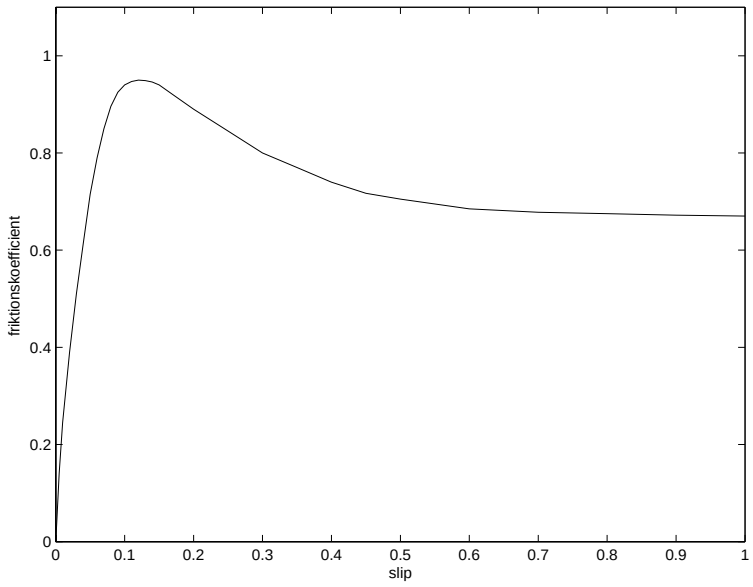
De första ABS systemen kom ut på marknaden på 1970-talet och idag är det standardutrustning i de flesta bilar. Förutom att förhindra att hjulen låser sig strävar ABS-system efter att hålla däckglidningen inom ett bestämt interval omkring friktionskkoefficientens maxima, se figur 1.1. Glidfaktorn, λ, är i praktiken svår att bestämma eftersom det inte finns något enkelt och praktiskt sätt att mäta fordonets lineära hastighet. Bromsregleringen bygger därför ofta på variabler som är enkla att mäta som exempelvis hjulens vinkelhastighet och vinkelacceleration samt bilens lineära acceleration [Wong, 2001].

Reglerstrategier som används inom ABS kan delas upp i två kategorier; Reglering av hjulaccelerationen och reglering av glidfaktor [Solyom, 2002].

Reglering av hjulaccelerationen är den absolut vanligaste och enklaste. Vanligtvis mäts hjulens acceleration och regulatorn agerar vid vissa tröskelvärden. De flesta ABS-system på marknaden är uppbyggda kring hydrauliska bromsar där bromstrycket regleras genom att öppna eller stänga ventiler. Regulatorn har då tre möjliga alternativ i varje cykel; öka trycket, minska trycket eller hålla kvar samma tryck [Kiencke och Nielsen, 2000]. Nackdelarna med att reglera hjulaccelerationen är att det ger kraftiga vibrationer vid bromsning eftersom glidfaktorn oscillerar kring maximat på friktionskurvan, samt att det kräver stora tabeller för olika bromsscenario vilket gör systemen svårhanterliga och nästan omöjliga att överblicka [Solyom, 2002].

Reglering av glidfaktorn är ett intressant alternativ till traditionel ABS med flera fördelar. Systemen är modellbaserade och blir därför lättare att analysera och och överblicka. Om slipvärdet kan hållas så att friktionen alltid är

maximal, istället för att oscillera kring maximat, kan bromssträckan minimeras. Med elektromekaniska bromsar finns möjligheten att ge kontinuerligt varierande bromsverkan som dessutom kan sättas olika på alla fyra hjul, vilket gör slipreglering möjlig [Solyom, 2002].



Figur 1.1: Typiskt utseende på hur friktionskoefficienten, μ, beror på glidfaktorn (*slipvärdet*), λ

1.2 Traction control

Traction control är närbesläktat med ABS och har många gemensamma delar. Tekniken används för att förhindra att hjulen slirar vid acceleration. För drivaxlar med enkla differentialer fördelas motormomentet lika mellan båda hjulen. Om friktionskoefficienten under det ena däcket är mycket mindre än under det andra kommer det att leda till att hjulet med låg friktion till underlaget spinner och tappar väggreppet. Problemet skulle kunna lösas med differentialspärrar eller med traction control då det hjul som snurrar fortast, bromsas för att återfå väggreppet. Om båda hjulen slirar eller vid långvarig körning på hala underlag regleras också motormomentet, för att förhindra att bromsarna blir överhettade. För tvåhjulsdrivna fordon är det lättare att mäta hjulglidning eftersom de frirullande hjulens hastighet kan användas som referens.

Eftersom ABS- och Traction Control-system har många gemensamma delar, som sensorer, kontrollenhet och bromstrycksmodulator, är dessa ofta integrerade i samma enheter i dagens fordon. [Wong, 2001]

1.3 Problemformulering

Uppgiften i examensarbetet bestod i att implementera en hjul-slip-reglering på en kommersiell radiostyrd modellbil.

Som tidigare nämnts är slip-reglering intressant ur många aspekter särskillt då allt fler bilar utrustas med elektromekaniska bromsar och brake by wire-system som möjliggör betydligt exaktare styrning av bromsverkan.

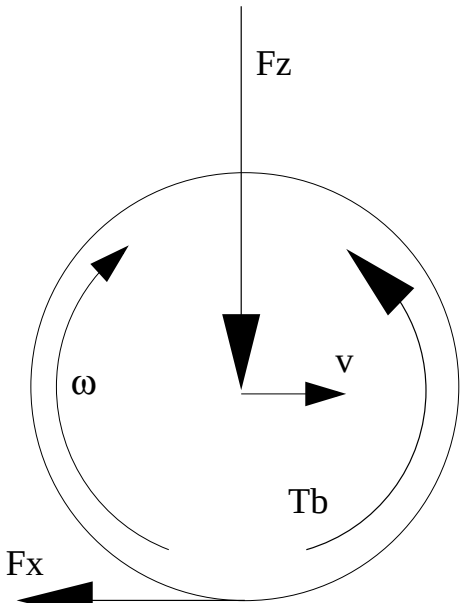
1.3.1 Begränsningar av uppgiften

Uppgiften begränsades till att bara omfatta drivning och bromsning på bakre hjulparet; den främre motorn plockades bort och användes ej.

Kapitel 2

Matematisk modell

Som modell används kvarts-bil-modellen dvs en modell av ett enda hjul som belastas med en massa, se figur 2.1.



Figur 2.1: Kvartsbil

Detta objekts rörelseekvationer ges av:

$$J\dot{\omega} = rF_x - T_b \tag{2.1}$$

$$m\dot{v} = -F_x$$

där:

J - tröghetsmoment hos hjulet, motorn samt alla roterande massor i kraftöverföringen däremellan.

ω - hjulets vinkelhastighet

r - hjulets radie

F_x - friktionskraft mellan däck och underlag

T_b - bromsande moment

m - massan hos kvartsbilen

v - bilens longitudinella hastighet

F_z - vertikal kraft

Den longitudinella däckglidningen definieras som:

$$\lambda = \frac{v - \omega r}{v} \tag{2.2}$$

vilket medför att $\lambda = 0$ när hjulet rullar fritt och att $\lambda = 1$ när hjulet är låst ($\omega = 0$). Friktionskraften mellan däcket och underlaget, F_x bestäms av:

$$F_x = F_z \mu(\lambda, \mu_H, \alpha, F_z, v)$$

där $\mu(\lambda, \mu_H, \alpha, F_z, v)$ är friktionskoefficienten mellan däck och underlag och beror på bland annat på glidningsfaktorn, λ och styrvinkel, α. Denna modell av friktionen tar dock inte hänsyn till eventuella krängande rörelser hos bilen.

Med insättning av ekvation 2.2 i ekvation 2.1 kan rörelseekvationerna uttryckas:

$$\dot{\lambda} v = -\frac{F_z \mu}{J} r^2 - \frac{F_z \mu}{m}(1 - \lambda) + \frac{r}{J} T_b \tag{2.3}$$

$$\dot{v} = -\frac{F_z \mu}{m}$$

Bilens hastighet ändras långsammare än övriga variabler och däckglidningen kan därför beskrivas som:

$$\dot{\lambda} v = -\frac{F_z r^2}{J} \mu + \frac{r}{J} T_b \tag{2.4}$$

Kapitel 3

Hårdvara

Figur 3.1: Bilen

3.1 Wild Dagger

Bilen som användes för projektet var Tamiya "Wild Dagger" skala 1/10. Den var utrustad med två stycken DC-motorer typ 540 med 27-varvslindningar, en motor för framhjulen och en för bakhjulen. Dessa motorer används förutom till själva framdrivningen också till att bromsa bilen. Motorerna kunde köras individuellt via varsitt fartreglage och möjliggjorde på så vis olika bromsning för fram- respektive bakhjulen. För att förenkla uppgiften användes bara en motor och ett fartreglage. Utöver detta utrustades bilen med följande:

- Fyra hjulhastighetsgivare för att mäta varje hjuls rotationshastighet individuellt.

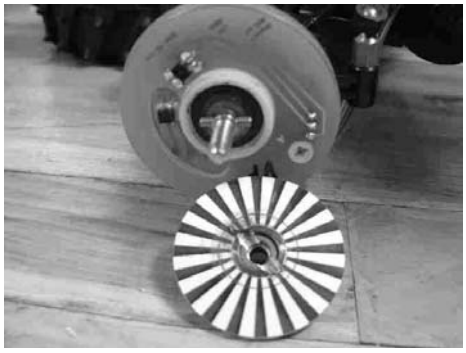
- Två identiska processorkort med en Atmel AVR microcontroller ATmega16 på vadera.
- Extra utsmyckningar som bromsljus och blinkers.

Bilen hade kraftiga däck med gummidubbar som gav ett allt för bra väggrepp för att kunna demonstrera fördelarna med slip-reglering. Däcken på de drivande hjulen förseddes därför med textilband för att minska friktionen mellan däck och underlag.

3.2 ATmega16

Processorerna som används är av typen atmel ATmega16 microcontroller. ATmega16 är en 8-bitars enchipsdator med 16kbyte programmerbart flashminne. Den klockas med 14.7MHz

3.3 Hjulhastighetsgivare

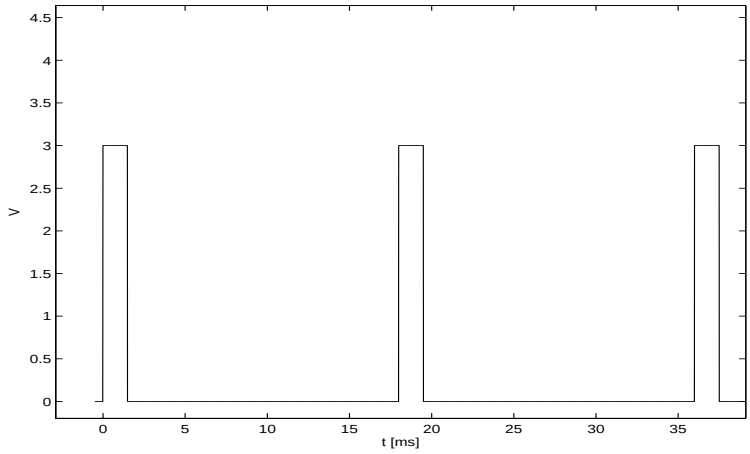


Figur 3.2: Hjulhastighetsgivare

Givarna på hjulen består av två optiska reflexgivare och en kodskiva. Kodskivan är uppdelad i 20 svarta och 20 vita cirkelsektorer och roterar med hjulet. När hjulet roterar ger reflexgivarna en sinusformad spänning vars frekvens beror på hjulets hastighet. Reflexgivarna är monterade på ett sådant sätt att signalerna från dem är fasförskjutna 90 grader i förhållande till varandra. Detta gör att även hjulets rotationsriktning kan bestämmas.

3.4 Radiosystemet

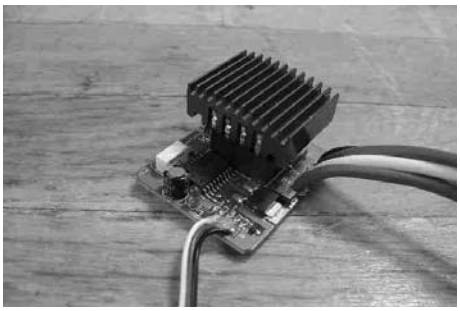
Radiosystemet använder frekvensmodulering med en bärvåg på 40.735MHz. Systemet har fyra kanaler varav tre användes i projektet. Bilens radiomottagare genererar styrsignaler till fartreglage och styrservo i form av en puls var 18:e millisekund. Bredden på pulsen varierar mellan 1.2ms och 1.9 ms och bestämmer styrsignalens storlek. En pulsbredd på 1.5 ms motsvarar neutralläge, se figur 3.3.



Figur 3.3: Principiellt utseende på signalen från en kanal på radiomottagaren.

3.5 Fartreglage

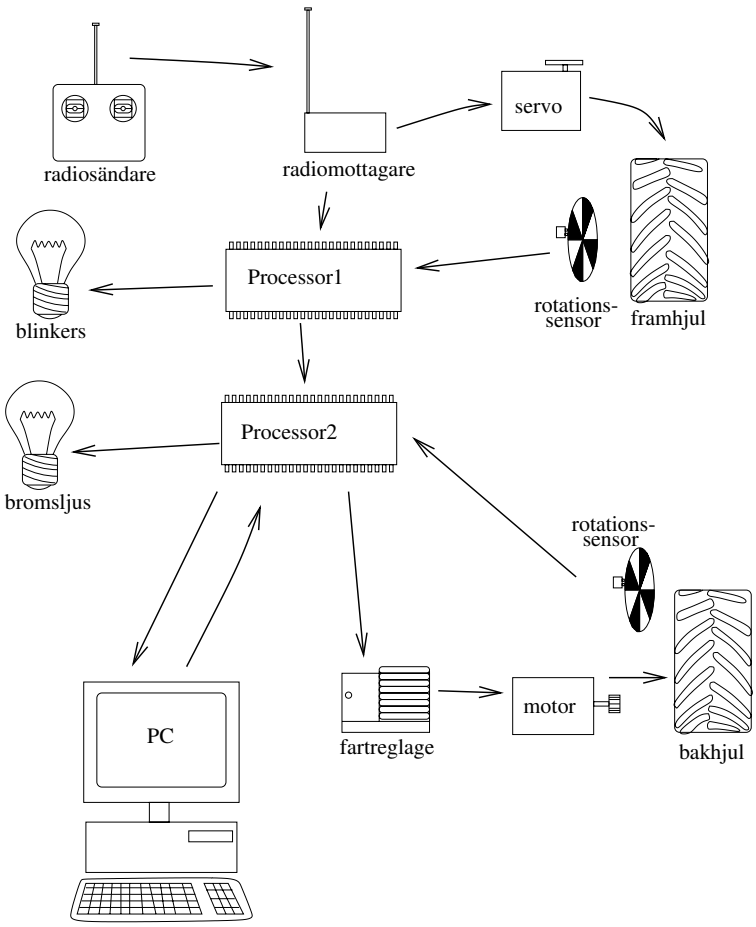
Som standardutrustning till bilen fanns ett mekaniskt fartreglage som bestod av en strömställare och ett variabelt motstånd. Detta gav två hastighetslägen framåt och ett bakåt. Eftersom det gav liten möjlighet att reglera, skaffades istället ett elektroniskt fartreglage, se figur 3.4 som gav möjlighet att ändra motorspänningen i betydligt fler steg.



Figur 3.4: Elektroniskt fartreglage

3.6 Sammankoppling av hårdvaran

I figur 3.5 visas hur de olika hårdvarudelarna är sammankopplade. Processor2 har en ömsesidig kommunikation med en PC, men i övrigt är alla signaler enkelriktade.



Figur 3.5: Schema över hur de olika hårdvarudelarna är sammankopplade. Pilarna beskriver signalflödet.

3.7 Begränsningar och olineäriteter i hårdvaran

Den största begränsningen fanns i fartreglaget. Detta är tillverkat för att ge föraren bästa möjliga köregenskaper vilket innebär att det innehåller en hel del olineäriteter:

- När insignalen till reglaget går från positiv till negativ ger det först en proportionell bromsverkan under ca 0.5s innan det ställer ut negativ spänning. När insignalen går från negativ till positiv finns dock ingen sådan bromsfunktion.
- Reglaget har en dödzon och ger ingen utsignal förrän insignalen överstiger ca 10 procent av sitt maximala värde. Denna dödzon är mindre i omvänd riktning, dvs om insignalen redan befinner sig i det positiva eller negativa intervallet och utsignal ges, kan insignalen sjunka något under 10 procent av sitt maximala värde innan utsignalen blir 0.
- För att undertrycka störningar och ge jämna köregenskaper lågpassfiltrerar reglaget insignalen.

Dessa egenskaper upptäcktes inte förrän mot slutet av arbetet då reglaget började användas till reglering av hjulhastighet.

Övriga begränsningar bestod i att fördelningen av motormomentet på höger respektive vänster hjul inte kunde påverkas eftersom hjulaxlarna var utrustade med differentialaxlar, samt att det i kraftöverföringen mellan motor och hjul finns ett glapp på ca 10 grader.

Kapitel 4

Mjukvara

Två stycken ATMega16 exekverar varsitt program samtidigt. Anledningen till att två processorer behövdes var främst för att kunna sampla signalerna från hjulhastighetsgivarna tillräckligt snabbt. Programmen är skrivna i C och avbrottsbaserade. Fullständig källkod återfinns i bilaga B.

4.1 Avbrottsrutiner

När en viss händelse sker begärs ett avbrott. Processorn sparar programräknarens värde på stacken och en avbrottsrutin kopplad till den specifika händelsen körs. Avbrottsrutinen är ett stycke programkod, liknande en funktion, men som anropas av hårdvaran. Om avbrottsrutinen deklareras med `INTERRUPT` sker exekveringen som vanligt och fler avbrott kan komma att ske. Om den däremot deklareras med `SIGNAL` kan den inte avbrytas av andra avbrott. Detta kan användas för att garantera ömsesidig uteslutning vid åtkomst av gemensama resurser.

När en avbrottsrutin har exekverats klart hämtas data från stacken och programmet fortsätter köra den kod som exekverades när avbrottet skedde.

De avbrottsrutiner som används i programmen är:

`SIGNAL(SIG_INTERRUPT0){}`

Anropas när en förändring i spänningsnivå registreras på pinnen PD2. Används i processor1 för att mäta en signal från radiomottagaren. 13 rader kod i processor1.

`SIGNAL(SIG_INTERRUPT1){}`

Anropas när en förändring i spänningsnivå registreras på pinnen PD3. Används i processor1 för att mäta en signal från radiomottagaren. 13 rader kod i processor1.

`SIGNAL(SIG_OVERFLOW0){}`

Anropas när Timer0 har räknats upp till sitt maximala värde 255 och börjar om på 0. Används i processor1 för att tända och släcka blinkers med passande tidsintervall. I processor2 används avbrottsrutinen som en säkerhetskontroll som sätter utsignalen till noll om inte data har mottagits från processor 1. Detta för att inte bilen skall skena om radiokontakten av någon anledning skulle förloras. 24 rader kod i processor1 och 12 rader i processor2.

`SIGNAL(SIG_UART_RECV){}`

Anropas när en byte data har mottagits på serieporten. Sparar datan och sätter ihop två byte till en 16-bitars integer som sparas i rätt parameter. Om kommando för överföring av loggad data har mottagits, skickas datan från denna rutinen i en lång loop och blockerar alltså processorn under en längre tid. 41 rader kod i processor2.

`SIGNAL(SIG_2WIRE_SERIAL){}`

Anropas då TWI-enheten behöver betjänas av programvara. 70 rader kod i processor1 och 25 rader i processor2.

`SIGNAL(SIG_ADC){}`

Identisk kod i båda processorerna, 19 rader lång. Exekveras då ADC-enheten är klar med en AD-omvandling. Från denna avbrottsrutin anropas en funktion för att beräkna hjulhastigheten.

`SIGNAL(SIG_INPUT_CAPTURE1){}`

Funktion för att mäta tidsvaraktighet, exempelvis pulsbredd. Används av processor1 för att mäta gaspådrag från radiomottagaren. 24 rader lång.

4.2 Fixpunktsaritmetik

I programmen används fixpunktsaritmetik istället för flyttal för att öka snabheten. Fixpunktsaritmetik är ett sätt att representera decimaltal endast med heltal, där decimaltal multipliceras med ett fixt värde (256 i de flesta fall i denna tillämpningen) och sedan trunkeras till heltal. I beräkningar används de uppskalade parametrarna och sedan divideras resultatet med samma fixvärde. Detta är en vanlig metod i små processorer med kort ordlängd och ger en hyfsad noggranhet i beräkningarna utan att ta upp för mycket minne eller processortid.

Uppgifterna är fördelade mellan processorerna på följande sätt:

Processor1

- Mäta framhjulens vinkelhastighet
- Mäta signalerna från radiomottagaren
- Skicka de uppmätta värdena till Processor2
- Tända blinkers

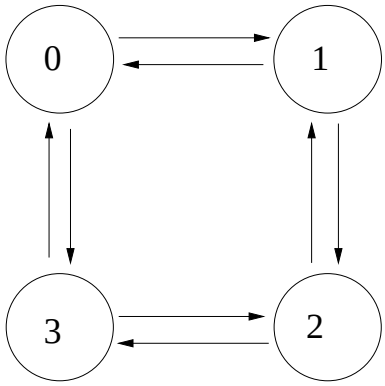
465 rader programkod.

Processor2

- Mäta bakhjulens vinkelhastighet
- Beräkna lämplig utsignal (PI-reglering)
- Ställa ut utsignal till fartreglaget
- Spara data från testkörningar
- Kommunicera med PC
- Tända bromsljus

770 rader programkod.

4.3 Mätning av hjulhastighet



Figur 4.1: Tillståndsmaskinen i vinkelmätningen

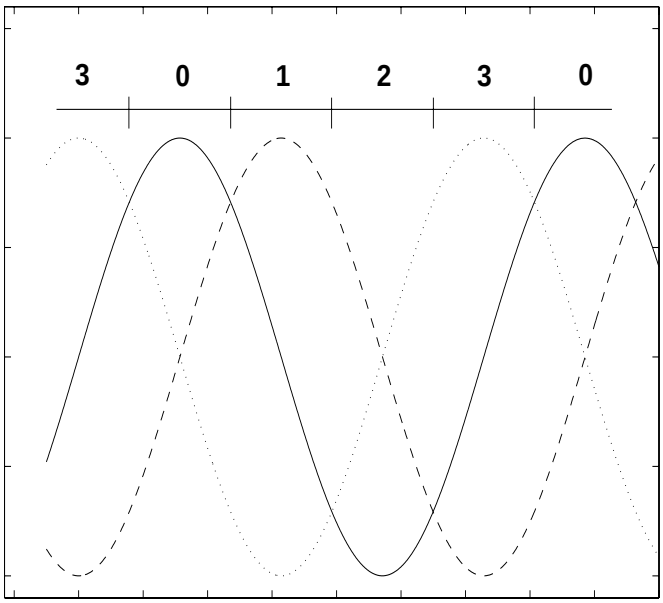
På varje hjul sitter en sensor som, när hjulet roterar, ger två sinusformade signaler. Dessa signaler samplas var för sig varpå en avkodning sker. Avkodningen kan liknas vid en tillståndsmaskin med fyra tillstånd som bestäms av vilken signal som har störst absolutvärde, se figur 4.1 och 4.2. Beroende på i vilket tillstånd avkodningsalgoritmen befinner sig divideras antingen insignal1 med insignal2 eller tvärtom. Arcustangens av kvoten ger ett värde på vinkeln. Arcustangens beräknas inte utan läses från en tabell i programkoden. Vinkelberäkningen ger en teoretisk upplösning på 20480 punkter per varv, men den verkliga noggranheten blir sämre pga mätbrus.

För att få en korrekt mätning av vinkeln krävs att avkodning sker minst en gång i varje kvadrant eller tillstånd. Avkodning görs med en frekvens på 2000Hz, vilket ger en teoretiskt största mätbara vinkelhastighet på 157 rad/s. Vinkelhastigheten beräknas från vinkeln genom en differensbildning mellan vart tjugonde sample dvs med 100Hz. Detta ger betydligt större noggrannhet och mindre kvantiseringsbrus än om differensen hade beräknats mellan varje sample.

Mätningen av hjulhastigheten är den mest tidskrävande proceduren i programmen och samma kod körs i bägge processorerna. Avkodningsfunktionen: `int encoder_resolve(int sin_ch, int cos_ch, struct encoder *enc) {}` är 200 rader lång och anropas från `SIGNAL(SIG_ADC){}`. Eftersom funktionen anropas från en avbrottsrutin är avbrott "disabled". `SIGNAL(SIG_ADC){}` är avbrottsrutinen som anropas varje gång processorns ADC-enhet är klar med en omvandling. I rutinen sparas det samplade värdet i en global variabel och ADC-enheten sätts att sampla en annan inpinne nästa gång. Varannan gång avbrottsrutinen körs anropas avkodningsfunktionen. ADC-enhetens prescaler är satt till 64 vilket är det snabbaste läget med full upplösning på tio bitar. Rutinen är 19 rader lång.

4.4 Mätning av signaler från radiomottagaren

De signaler som mäts från radiomottagaren är



Figur 4.2: Signaler från hjulhastighetsgivarna och de olika tillstånden i tillståndsmaskinen. Den heldragna kurvan är "cosinussignalen", de streckade och prickade linjerna motsvarar "sinussignalen" vid medurs respektive moturs rotation.

- "gaspådrag"
- "rattutslag"
- "mode"- fram, back eller manuel körning utan reglering

För att mäta pulsbredd finns det i Atmega16 en InputCapture-enhet. InputCapture kan sättas att generera avbrott när en förändring i spänningsnivå sker på en viss inpinne. När en viss nivåförändring sker kopieras värdet från Timer1 till ett register och kan läsas av programvaran t.ex. med en avbrottsrutin. I varje processor finns bara möjlighet till att mäta en signal med inputCapture och i processor2 används Timer1 till att generera en pulsbreddsmodulerad utsignal. Därför kunde bara en signal mätas på detta sätt. Förarens "gaspådrag" är den signal från radiomottagaren som kräver störst noggranhet, därför används inputCapture till den. I `SIGNAL(SIG_INP_CAPTURE){}` läses värdet från ICP1 och enheten sätts att reagera på negativ flank nästa gång. Nästa gång avbrottsrutinen anropas beräknas differensen mellan det nya värdet och det gamla. De övriga signalerna mäts mha external interrupt där värdet på en räknare läses med programvara och sedan trösklas till ett av tre möjliga värden. Rattutslag mäts bara för att kunna tända blinkers vid rätt tillfälle, signalen går opåverkad vidare till styrservot som svänger framhjulen.

4.5 Kommunikation mellan processorerna

Processorerna kommunicerar via Two Wire Serial Interface (TWI) som bygger på protokollet I^2C. Som framgår av namnet används bara två ledare för kommunikationen; en ledare för klocksignal och en för data. Processor1 agerar i "master transmitter mode" och är alltså den som initierar en dataöverföring. Processor2 verkar således i "slave reciever mode" och tar emot data på kommando från processor1. När processor1 har mätt pulslängden på radiomottagarens "gaspådragssignal" skickar den över sin data till processor2. Datan består av fyra variabler; hastigheten för vänster respektive höger framhjul, förarens gaspådrag samt "mode".

4.6 Reglering

Samplingsfrekvensen till regulatorn styrs av radiosystemets pulser. När processor1 har mätt pulslängden på radiomottagarens "gaspådragssignal" skickar den över sin data till processor2 som då kör ett varv i regulatoralgoritmen. Detta ger ett samplingsintervall på 18 ms, vilket är ett naturligt val eftersom fartreglaget endast kan påverkas så ofta.

Börvärdet ω_{ref} till regulatorn ges vid bromsning av de frirullande hjulens hastighet ω_{fr} enligt

$$\omega_{ref} = (1 - \lambda) * \omega_{fr}$$

och vid acceleration

$$\omega_{ref} = (1 + \lambda) * \omega_{fr}$$

λ kan varieras i steg om jämna åttondelar. Vid acceleration slås regulatorn till om föraren försöker ge ett gaspådrag som skulle medföra större slipfaktor än det inställda värdet. Samma sak gäller vid bromsning; om föraren försöker bromsa hjulen hårdare än det inställda värdet på λ kopplas regulatorn in.

För att bilen skall kunna accelerera med hjälp av denna slipreglering krävs en viss initialhastighet. Detta löstes genom att ställa ut en konstant spänning till motorn så fort föraren ger något gaspågrag och sedan öka spänningen tills hastigheten överstiger ett visst tröskelvärde då regulatorn slås på. Regulatorns integratordel sätts till ett värde proportionellt mot motorspänningen vid tiden då regleringen startar, för att åstadkomma en jämn övergång.

Eftersom beräkning av hjulhastigheten görs med ett bestämt tidsintervall oberoende av regulatoralgoritmen medför det en varierande tidsfördröjning från det att mätning görs tills utsignal presteras. För att inte göra denna tidsfördröjning ännu större beräknas utsignalen enligt följande princip [Årzén 2003]:

```
u=piCalculateOutput(d3*lambdaBrake,d1);
v=limit(u,390,0);
Out(v);
piUpdateState(d3*lambdaBrake,d1,v,u);
```

Styrsignalen ställs ut i funktionen `Out(v);`, och först efter det räknas integraldelen upp i `piUpdateState();`

AntiWindup har implementerats genom tracking för att unvika att integratordelen växer när styrsignalen mättar.

4.6.1 Åtgärder för att kringå begränsningar i fartreglaget

För kunna använda fartreglaget trots dess olineäriteter och begränsningar gjordes olika försök att anpassa styrsignalen. Generellt begränsades utsignalen till antingen det positiva eller negativa intervallet för att ta bort inverkan av bromsfunktionen som aktiverades då styrsignalen gick från positiv till negativ. För att eliminera påverkan från reglagets dödzon adderades motsvarande konstanta term till utsignalen.

Om föraren bromsar när bilen kör framåt ger regulatorn en negativ styrsignal som fartreglaget svarar på med en negativ utspänning. I [Bengtsson, Solyom 2001] reglerades bromsning av en legobil framgångsrikt på detta sätt, men motorn i RC-bilen ger betydligt större vridmoment och förmår hjulen att rotera åt motsatt håll om negativ spänning verkar på motorn under någon längre tid. Därför slås regulatorn av så fort slipvärdet överstiger det inställda, och aktiveras igen när det understiger börvärdet.

Vid inbromsning från en hastighet bakåt används en annan strategi; För att kringå att reglagets inbyggda bromsfunktion bara är aktiv i ca 0.5s sattes styrsignalen att växla tecken till det positiva området under några samples, utan att överstiga reglagets dödzon. På detta sätt befinner sig reglaget i det positiva intervallet utan att någon utspänning ges. Under denna period bromsas inte hjulen utan rullar fritt och regulatorn är då avstängd. När sedan regulatorn aktiveras igen och ger en negativ styrsignal kommer reglagets bromsfunktion att aktiveras på nytt.

4.7 Generering av utsignal

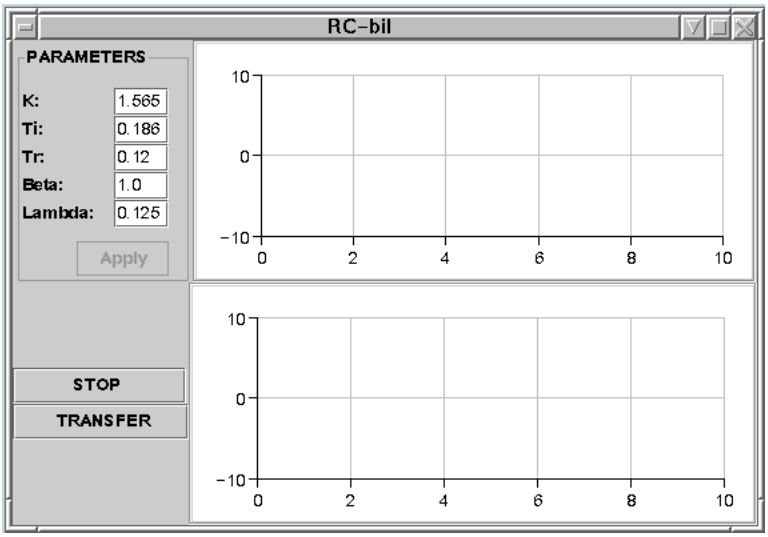
Styrsignalen till fartreglaget ska vara av samma typ som signalerna från radiomottagaren, se figur1. Detta ordnades med PWM-modulen som finns i ATMega16. Periodtiden för pulserna ställdes in för att stämma överens med radiosystemet.

4.8 Loggning av data

Börvärdet, mätvärdet och styrsignalvärdet loggas under en inbromsning och kan sedan föras över till PC:n. När en ny inbromsning görs försvinner den gamla loggen och nya värden sparas.

4.9 Kommunikation mellan processor2 och PC

Processorkorten i bilen är utrustade med portar för seriell kommunikation (RS232) och kan kommunicera via dessa med en operatörsdator. Över serieporten skickas en byte i taget och ett enkelt protokoll, där datapaket sätts samman av tre byte, har därför utvecklats. Varje paket består av en header som talar om vilken variabel paketet innehåller och en datadel som består av en 16-bitars integer.



Figur 4.3: Grafiskt användargränssnitt

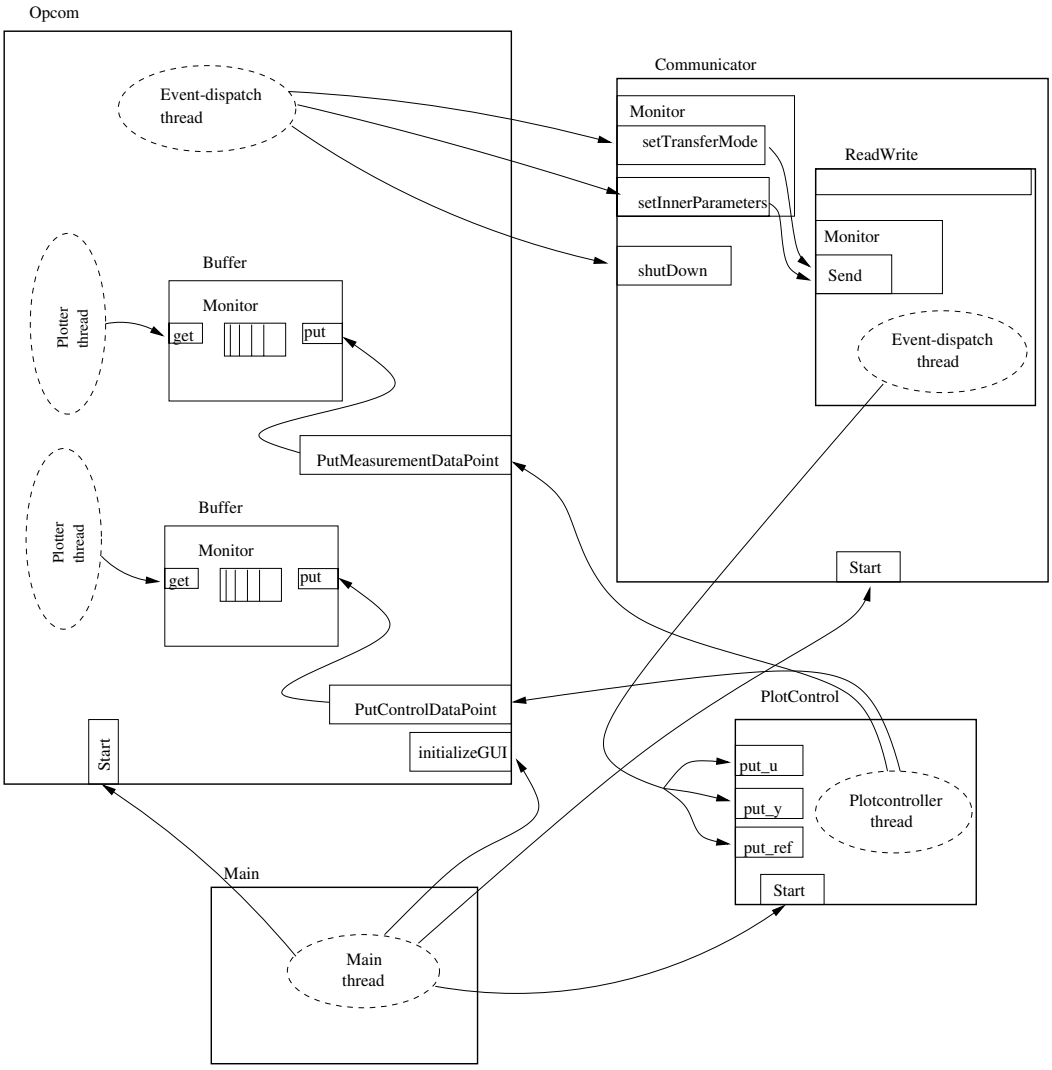
4.10 PC-program

Det grafiska användargränssnittet som körs i operatörsdatorn är implementerat i Java med Java-Swing. För seriell kommunikation används "javax.comm extension package" Från programmet kan regulatorparametrarna ändras, hastigheten från hjulen kan plottas i realtid och överföring av loggad data kan göras.

Programmet består av följande klasser;

- Communicator.java
- Main.java
- Opcom.java
- PI.java
- PIGUI.java
- PIParameters.java
- PlotControl.java
- PlotData.java
- ReadWrite.java

Figur 4.4 visar hur de olika klasserna är förhåller sig till varandra.



Figur 4.4: Strukturdiagram över Java-klasserna i PC-programmet. Varje modul representeras av en rektangel med tjock linje. Publika metoder representeras av rektanglar i linje med modulens kant. Monitorer inne i en modul ritas som en rektangel med tunn linje inne i modulen och monitorns synkroniserade metoder som rektanglar i linje med monitorns kant. Trådar representeras med streckade elipser.

Kapitel 5

Resultat

Vid körning med reglering kopplas regleringen in om föraren gasar eller bromsar så kraftigt att glidfaktorn, λ överstiger det tillåtna. Övergången sker mycket jämnt och ger bra köregenskaper för föraren.

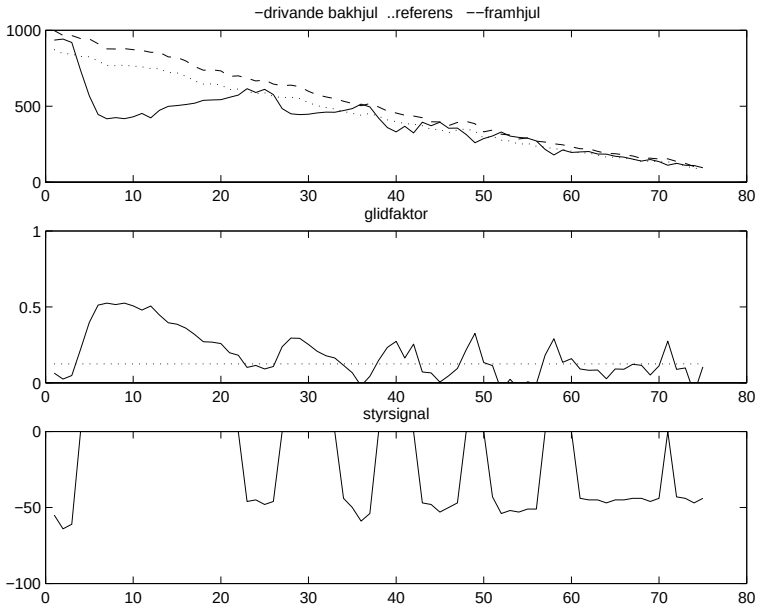
På grund av de begränsningar som finns i fartreglaget har någon verklig slipreglering inte kunnat genomföras i alla olika fall. Vid acceleration framåt eller bakåt regleras slipvärdet med en PI-algoritm. Vid inbromsning under körning framåt däremot fungerar systemet mera som ett traditionellt ABS som reglerar bromsningen genom att slå på och av bromsverkan, vilket ger kraftiga vibrationer i bilen under inbromsningen på samma sätt som i vanliga kommersiella ABS-lösningar. Detta syns tydligt i figur 5.1 och figur 5.2; Styrsignalen slår till och från flera gånger per sekund. Även i dessa fall kan bilen dock bromsa utan att hjulen låser sig, mer än i korta intervall, eller att kontrollen över fordonet går förlorad. Vid bromsning från hastighet bakåt utnyttjas fartreglagets inbyggda bromsfunktion. Denna är endast aktiv under en kort tid då insignalen till reglaget går från positiv till negativ och för att kringgå denna begränsning sätts styrsignalen till att växla tecken till det positiva området. Detta är alltså anledningen till styrsignalens utseende i figur 5.9, 5.10, 5.11 och 5.12 Anledningen till att styrsignalen växlar tecken till det positiva området under kortare intervaller är för att kringgå denna begränsning.

I figurerna nedan är hjulens vinkelhastighet givna i den enhet som de beräknas i processorerna. Då differensbildningen av två vinkelvärden sker med 100 Hz och ett varv motsvarar 20480 punkter motsvarar ett hastighetsvärde på 1000 en verklig vinkelhastighet på ca 30 rad/s och om hjulen inte slirar är detta detsamma som en linjär hastighet på ca 2 m/s. Tidsaxlarna är givna i antal sample, där 55 sample motsvarar 1 s. På grund av det begränsade minnet i ATMega16 har bara 75 mätvärden kunnat sparas åt gången, vilket innebär att testkörningarna i figurerna nedan visar ett tidsförlopp på ca 1,4 s.

Den heldragna kurvan visar ärvärdet och den prickade visar referensvärdet.

Vid acceleration når regulatorn snabbt sitt referensvärde, se figur 5.7, men glidfaktorn, λ oscillerar tyvärr en del kring sitt referensvärde. Detta kanske skulle kunna åtgärdas medbättre regulatorparametrar, men det skulle också kunna vara en följd av olineäriteter i hårdvaran.

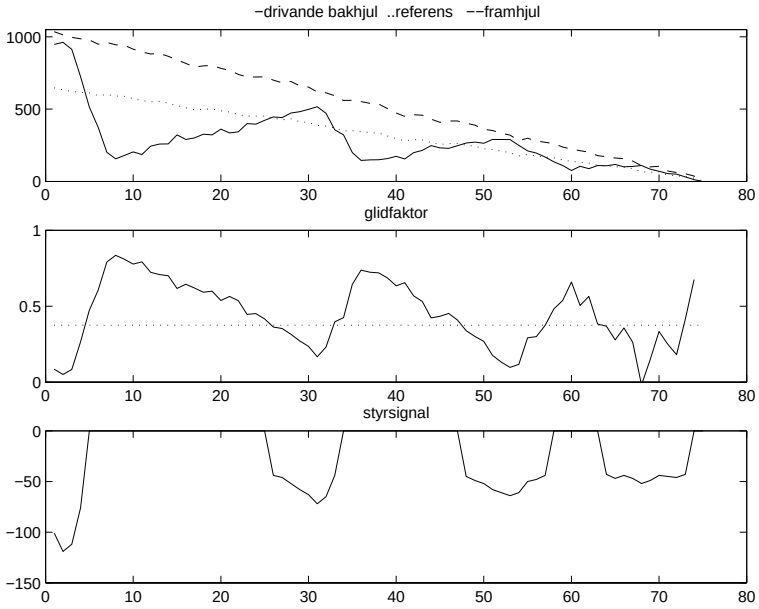
I figur 5.12 framgår det att försöket att *lura* fartreglaget inte riktigt fungerar eftersom bromsverkan avtar mot slutet trots att styrsignalen växer nära sitt



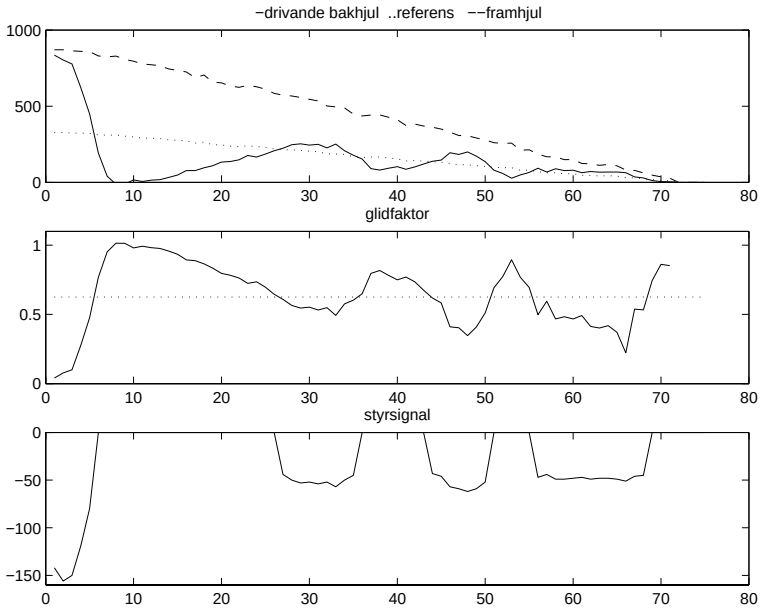
Figur 5.1: Bromsning, $\lambda = 1/8$

maxima. När hastigheten minskar krävs det mindre vridmoment för att bromsa hjulen och följdaktligen borde styrsignalen också minska när hastigheten sjunker. Detta är också fallet vid bromsning från hastighet frammåt, se figur 5.10. Att styrsignalen istället växer i figur 5.12 beror på att fartreglagets bromsfunktion avtar.

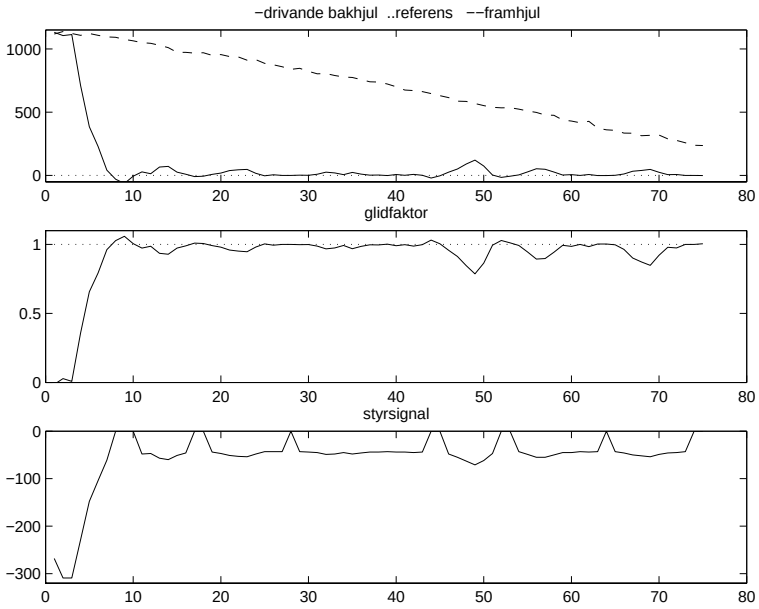
Även om detta försök att kringgå fartreglagets begränsningar inte är en rimlig metod i en slutgiltig produkt visar det ändå på fördelarna med slipreglering. Vid en jämförelse mellan figur 5.11 och figur 5.3 framgår det att under den korta tid då fartreglagets bromsfunktion verkligen är aktiv oscillerar λ betydligt mindre i figur 5.11.



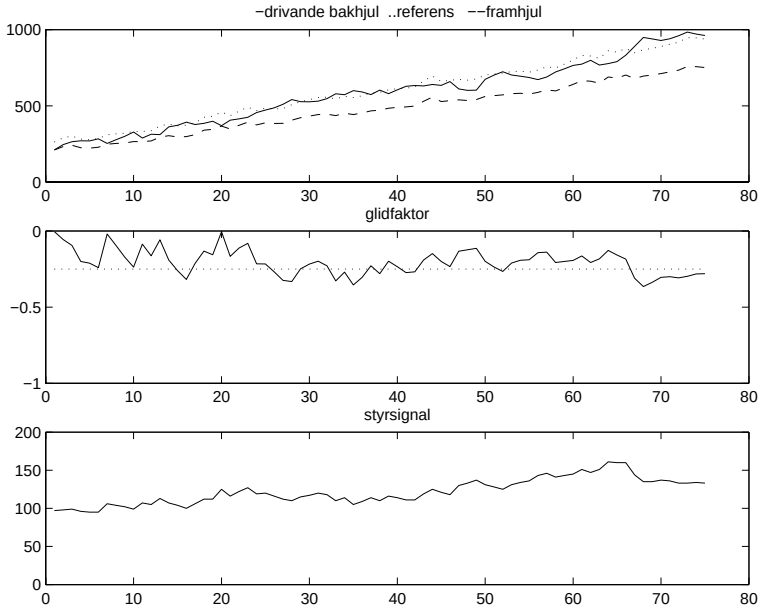
Figur 5.2: Bromsning, $\lambda = 3/8$



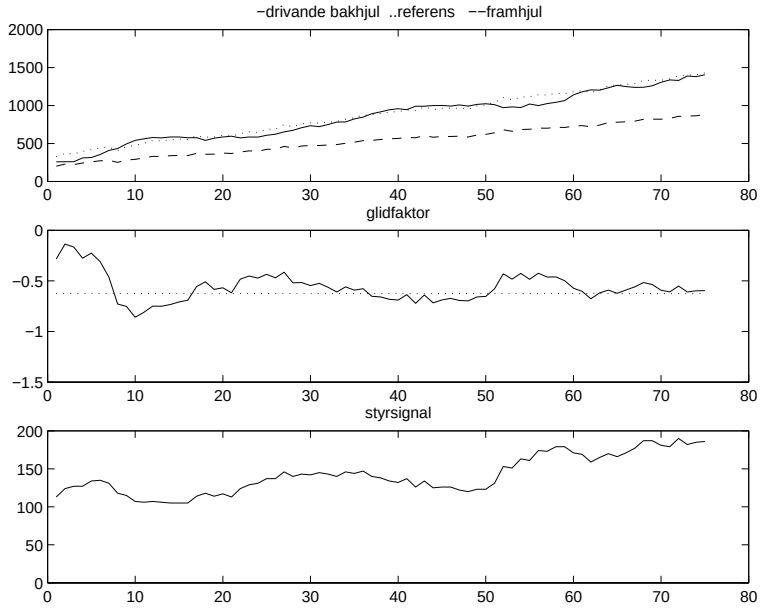
Figur 5.3: Bromsning, $\lambda = 5/8$



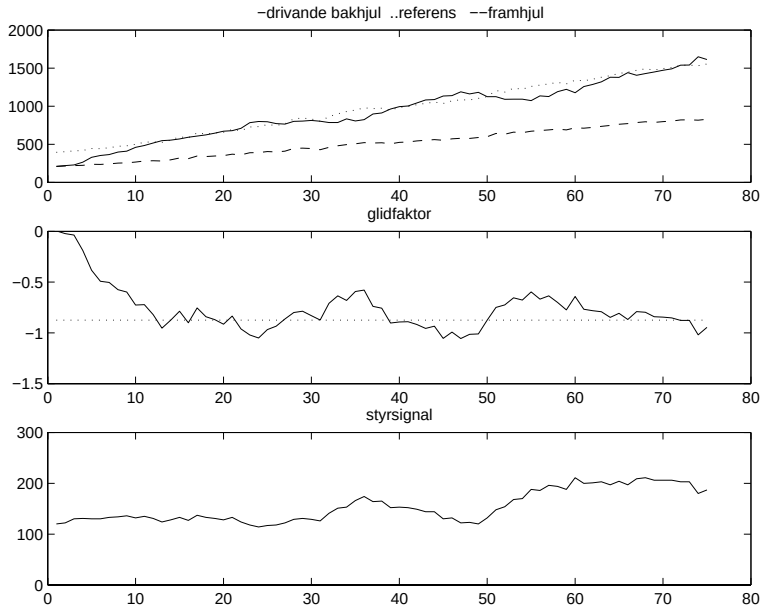
Figur 5.4: Bromsning, $\lambda = 1$



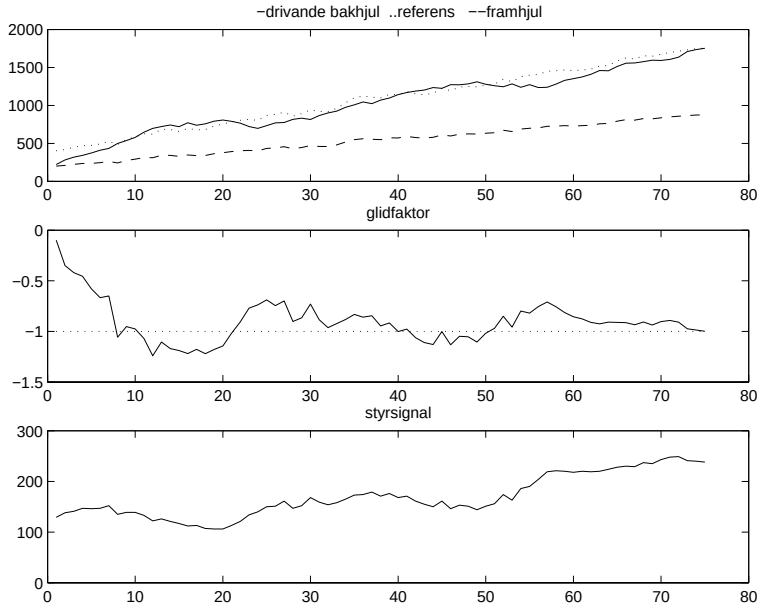
Figur 5.5: Acceleration, $\lambda = 2/8$



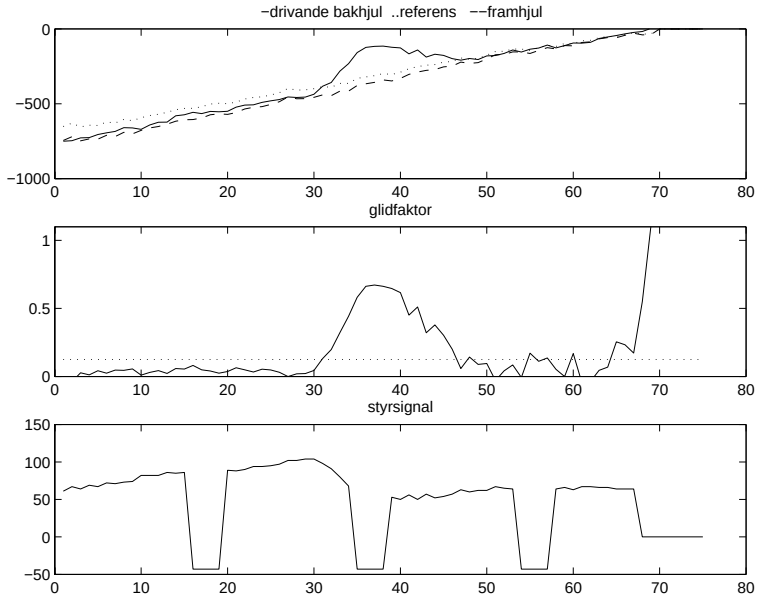
Figur 5.6: Acceleration, $\lambda = 5/8$



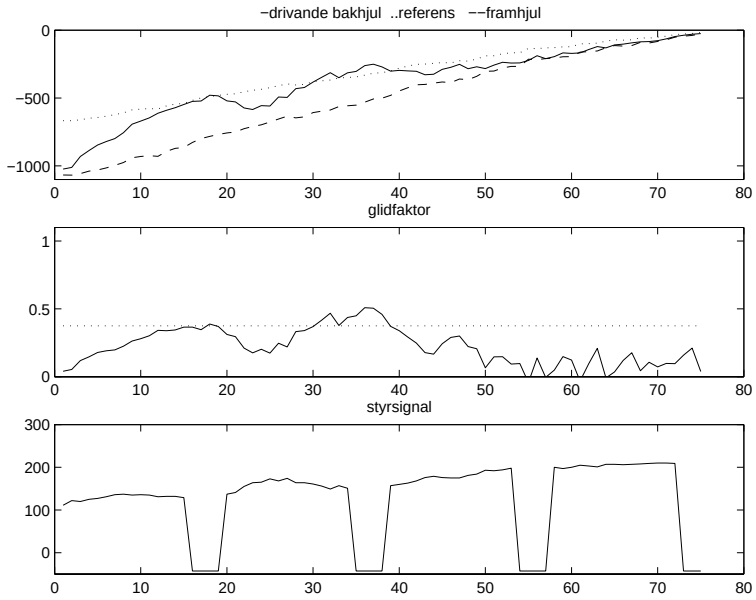
Figur 5.7: Acceleration, $\lambda = 7/8$



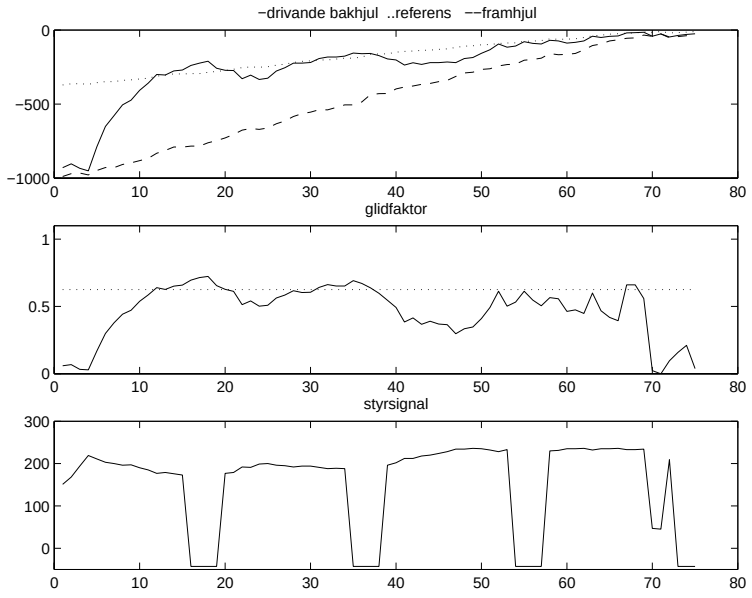
Figur 5.8: Acceleration, $\lambda = 1$



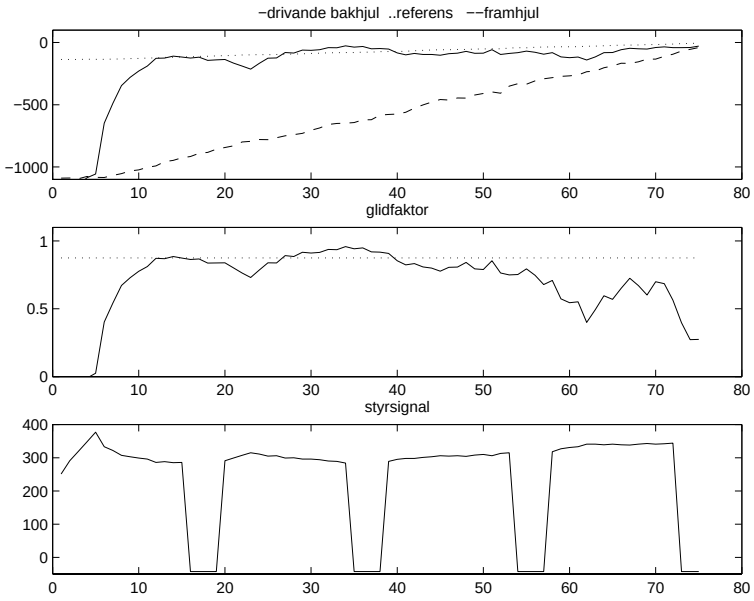
Figur 5.9: Bromsning från hastighet bakåt, $\lambda = 1/8$



Figur 5.10: Bromsning från hastighet bakåt, $\lambda = 3/8$



Figur 5.11: Bromsning från hastighet bakåt, $\lambda = 5/8$



Figur 5.12: Bromsning från hastighet bakåt, $\lambda = 7/8$

Kapitel 6

Framtida arbete

Den största begränsningen i detta arbetet har varit fartreglaget. Om ett sådant skulle specialtillverkas, som ger utspänning eller bromsverkan direktproportionell mot insignal, skulle regleringen kunna göras avsevärt bättre.

För bättre prestande borde olika regulatorparametrar väljas för reglering av bromsning respektive acceleration. Som det ser ut nu används samma parametrar i alla olika fall. För att underlätta tuning skulle systemidentifiering och modellering kunna användas i betydligt större utsträckning.

En annan förbättring skulle vara att köra med två motorer och uppskatta hastigheten med hjälp en accelerometer. Problem som uppstår då är att accelerometern även registrerar tyngdacceleration vilket gör att accelerometersignalen inte kan integreras direkt för att få hastigheten. En lösning kan vara att använda ett komplementärt filter och blanda signalen från accelerometern med exempelvis den från hjulsensorerna. Snabba förändringar i hastigheten fås då från accelerometern och vid långsamma förlopp ges större tillförlit till hjulhastigheten. På detta sätt kan man få det bästa ur båda sensorerna; integrering av tyngdaccelerationen filtreras bort och likaså blir inverkan av att hjulen ibland slirar respektive låser sig liten.

Litteraturförteckning

[1] Sefan Solyom. *Synthesis of a Model-based Tire Slip Controler* Department of Automatic Control Lund Institute of Technology 2002.

[2] J. Y. Wong *Theory of Ground Vehicles.* Third edition. Wiley (2001).

[3] U. Kiencke L. Nielsen *Automotive Control Systems.* Springer Verlag 2000.

[4] ATMEL *ATMega16 Data*

[5] S. Drakunov, U. Ozguner, P. Dix,B. Ashrafi. *ABS control using optimum search via sliding modes.* Control Systems Technology, IEEE Transactions on ,Volume: 3 , Issue: 1 , March 1995

[6] Johan Bengtsson, Stefan Solyom *ABS and Anti-skid on a LEGO car -A projekt in Embedded Systems.*

[7] Karl-Erik Årzén *Real-Time Control Systems* Department of Automatic Control Lund Institute of Technology 2003.

[8] Per Foreby *Att skriva rapporter med LaTeX* 2003.

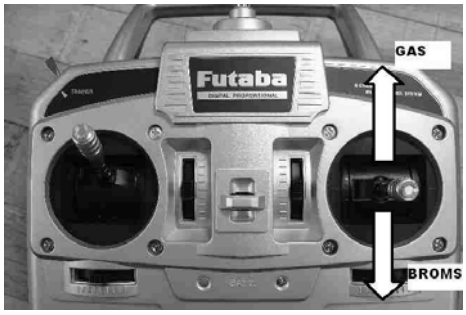
Bilaga A

Liten manual till bilen

Slå först på bilens strömställare framför bilens vänstra bakhjul, och kontrollera att processorkorten också är spänningsmatade så att de gröna lysdioderna på korten lyser. Koppla därefter in NiCd-batteriet till fartreglaget, kontakten sitter på bilens högra sida. Lysdioden på fartreglaget skall då slockna. Det är viktigt att detta görs i rätt ordning eftersom fartreglagets nollnivå programmeras när batterispänningen kopplas till. Slå till sist på radiosändaren.

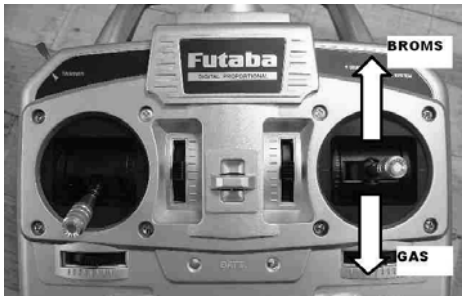
Den vänstra spaken på radiosändaren används som lägesväljare och kan ställas i ett av tre lägen. Den högra styrspaken används för att gasa och bromsa samt att svänga åt höger eller vänster.

Framåt med reglering



Figur A.1: Framåt med reglering

Detta läge väljs genom att ställa den vänstra styrspaken i sitt ändläge framåt. I detta läge accelereras fordonet när den högra styrspaken förs framåt och bromsas när den förs bakåt, acceleration och bromsning begränsas till det angivna slipvärdet, dvs regulatorn slås på om föraren försöker accelerera eller bromsa mer än vad väggreppet tillåter.



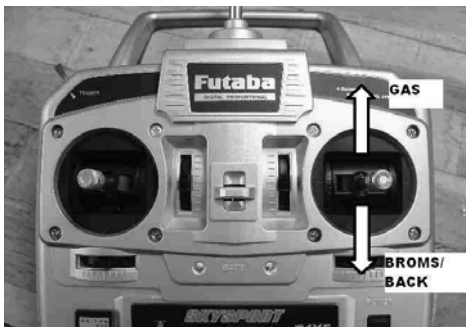
Figur A.2: bakåt med reglering

Bakåt med reglering

I detta läget accelereras fordonet bakåt när den högra styrspaken förs bakåt och bromsas när den förs framåt. Acceleration och bromsning begränsas till det angivna slipvärdet.

Läget väljs genom att ställa den vänstra styrspaken i sitt ytterläge bakåt.

Manuell körning



Figur A.3: utan reglering

När den vänstra styrspaken står i sitt mittenläge accelereras fordonet framåt när den högra styrspaken förs framåt och bromsas respektive backar när den förs bakåt. Acceleration och bromsning är inte begränsad av regulatorn.

Gui startas med kommandot:

```
>java Main
```

Från programmet kan slipvärdet λ och regulatorparametrarna ändras om den nedre processorn har sin serieport kopplad till PC:n. När radiosändaren är på skickas kontinuerligt hjulhastigheten för bakhjulen och rítas som kurvor i GUI-fönstret. När seriekabeln skall kopplas in bör radiosändaren vara avstängd för att undvika problem med kommunikationen. Protokollet som används förutsätter att den första byte som tags emot är en *header byte*. När knappen Transfer

trycks in skickas data från den senaste bromsningen eller accelerationen. Data loggas endast när bilen körs i läge *framåt med reglering* eller *bakåt med reglering* och bromsas eller accelereras. Om lägesväljaren sätts till *manuell körning* kan datan sparas över nästa gång något av de andra lägena väljs.

Bilaga B

Källkod

```
/*top.c            Processor1      */

#include <avr/io.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
volatile  uint8_t intrptFlag=0;
volatile  short mode=0;
volatile  short blinkers=0;
volatile  uint8_t index=0;
volatile  uint8_t TWIdata[4];
volatile  uint8_t TWIstatus[10];
volatile  uint16_t wheel[5];
volatile  short encDeriv[2];
volatile  short ref=20;
volatile short counter2=0;
volatile short count_blinkers=0;
volatile short time=0;

struct encoder {
   signed int sin_min, sin_max,
     cos_min, cos_max, state, oldPos,counter, derivata, oldDerivata;
   char sin_gtz, cos_gtz;
   signed int offset, pos_interp;
   signed int pos_base;
};

struct encoder encod[2];
 /*initial values for encoders*/
 /*these values are different for each sensor*/

const struct encoder encoder2_init={105,270,240,470,0,0,0,0,0,0,0,0,0,0};
const struct encoder encoder3_init={145,366,176,400,0,0,0,0,0,0,0,0,0,0};

const arctan[128] =  {0, 1, 3, 4, 5, 6, 8, 9, 10, 12, 13, 14, 15, 17,18,
                      19, 20, 22, 23, 24, 25, 27, 28, 29, 30,32, 33, 34,
```

```
                        35, 37, 38, 39, 40, 41, 43, 44, 45, 46,47, 49, 50,
                        51, 52, 53, 54, 55, 57, 58, 59, 60, 61,62, 63, 64,
                        66, 67, 68, 69, 70, 71, 72, 73, 74, 75,76, 77, 78,
                        79, 80, 81, 82, 83, 84, 85, 86, 87, 88,89, 90, 91,
                        92, 93, 93, 94, 95, 96, 97, 98, 99, 100,100, 101,
                        102, 103, 104, 105, 105, 106, 107, 108,109, 109,
                        110, 111, 112, 113, 113, 114, 115, 116,116, 117,
                        118, 119, 119, 120, 121, 121, 122, 123,123, 124,
                        125, 125, 126, 127, 127, 128};
#define ENCODER_N 10/* number of segments for a half turn of the wheel
                        encoder disc*/
#define ENCODER_DELTA 256
const long res = 4*ENCODER_DELTA*ENCODER_N;
//resolution, 1/2 turn

int abs(int x){
   return x < 0 ? -x : x;
}

short int sign(int x){
   if (x == 0)
     {
       return 0;
     }
   else
     {
       return x < 0 ? -1 : 1;
     }
}

/*Function used for calculating the angular velocity of the wheels
from the sensor values*/
 short encoder_resolve(int sin_ch, int cos_ch, struct encoder *enc) {
  unsigned char state;
  long pos_interp, pos;
  short derivata, derivReturn;
  int sin_mid, cos_mid, sin_amp, cos_amp, sin_norm, cos_norm;

  sin_mid = (enc->sin_max + enc->sin_min)/2;
  cos_mid = (enc->cos_max + enc->cos_min)/2;
  sin_amp = (enc->sin_max - enc->sin_min)/2;
  cos_amp = (enc->cos_max - enc->cos_min)/2;

  sin_norm = ((sin_ch - sin_mid) *128)/sin_amp;// sin_norm is between [-128, 128]
  cos_norm = ((cos_ch - cos_mid) *128)/cos_amp;
  if (abs(sin_norm) < abs(cos_norm))
    {
      if (cos_norm > 0) // state 0
        {
          state = 0;
```

```
          if (enc->sin_gtz != (sin_norm > 0))
/*detects when sine-signal goes through zero*/
            {
              enc->sin_gtz = (sin_norm > 0);
              enc->cos_max = cos_ch;
            }
          enc->pos_interp = sign(sin_norm)*arctan[abs((sin_norm*127)/cos_norm)];
          switch (enc->state)
            {
            case 1:
              enc->pos_base -= ENCODER_DELTA;
              break;
            case 3:
               enc->pos_base += ENCODER_DELTA;
               break;
            }
        }
      else // state 2
        {
          state = 2;
          if (enc->sin_gtz != (sin_norm > 0))
            {
              enc->sin_gtz = (sin_norm > 0);
              enc->cos_min = cos_ch;
            }
          enc->pos_interp = -sign(sin_norm)*arctan[abs((sin_norm*127)/cos_norm)];
          switch (enc->state)
            {
            case 1:
              enc->pos_base += ENCODER_DELTA;
               break;
            case 3:
              enc->pos_base -= ENCODER_DELTA;
              break;
            }
        }
    }
  else
    {
      if (sin_norm > 0) // state 1
         {
           state = 1;
           if (enc->cos_gtz != (cos_norm > 0))
             {
               enc->cos_gtz = (cos_norm > 0);
                enc->sin_max = sin_ch;
             }
           enc->pos_interp = -sign(cos_norm)*arctan[abs((cos_norm*127)/sin_norm)];
           switch (enc->state)
             {
```

```
            case 0:
              enc->pos_base += ENCODER_DELTA;
              break;
            case 2:
             enc->pos_base -= ENCODER_DELTA;
              break;
            }
         }
      else // state 3
        {
          state = 3;
          if (enc->cos_gtz != (cos_norm > 0))
             {
               enc->cos_gtz = (cos_norm > 0);
               enc->sin_min = sin_ch;
             }
          enc->pos_interp = sign(cos_norm)*arctan[abs((cos_norm*127)/sin_norm)];
          switch (enc->state)
            {
            case 0:
              enc->pos_base -= ENCODER_DELTA;
              break;
            case 2:
               enc->pos_base += ENCODER_DELTA;
               break;
             }
        }
    }
  enc->state = state;
  pos = enc->pos_base + enc->pos_interp + enc->offset;
 enc->counter++;
  if(enc->counter>20){
/*angular velocity is derived from the angle every twentieth time */
    derivata=pos - enc->oldPos;
  }

/*check if more than half a turn has passed*/
  if (pos < -res/4) {
    enc->pos_base = enc->pos_base + res;
    pos = enc->pos_base + enc->pos_interp + enc->offset;
    enc->oldPos=enc->oldPos+res;
  }
  if (pos > 3*res/4) {
    enc->pos_base = enc->pos_base - res;
    pos = enc->pos_base + enc->pos_interp + enc->offset;
    enc->oldPos=enc->oldPos-res;
  }
  enc->oldDerivata=enc->derivata;

  if(enc->counter>20){
```

```
    enc->counter=0;
    enc->oldPos=pos;
    enc->derivata=derivata;
  }

  return enc->derivata;
 }


void encoder_reset(struct encoder *enc) {
  enc->offset = -(enc->pos_base + enc->pos_interp);
}

/* interrupt routine for measuring the throttle signal from radio reciever*/
SIGNAL(SIG_INPUT_CAPTURE1){
  uint8_t lbyte,hbyte;
  static unsigned int oldtime;

  lbyte=inp(ICR1L);
  hbyte=inp(ICR1H);

  if(inp(TCCR1B)&0x40){
     oldtime=(hbyte<<8) | lbyte;
    outp(TCCR1B & 0xBF, TCCR1B);
    outp(TIFR & 0xDF,TIFR);

  }
  else{
    time=2780-((((hbyte<<8) | lbyte)-oldtime)/2);

    outp(TCCR1B | 0x40, TCCR1B);
    outp(TIFR & 0xDF,TIFR);
    outp(0x00,TCNT1H);
    outp(0x00,TCNT1L);

    TWIdata[0]=(time>>8);
    TWIdata[1]=time&0x00ff;
    outp(BV(TWSTA)|BV(TWINT)|BV(TWEA)|BV(TWEN)|BV(TWIE),TWCR);
    intrptFlag=1;
  }
}

/* interrupt routine for measuring the steering signal from radio reciever*/
SIGNAL(SIG_INTERRUPT0){
  if((inp(MCUCR)&0x03)==3){
    outp(0x00,TCNT2);
    outp(BV(CS22)|BV(CS20), TCCR2); //TCNT0=CLK/1024
    outp(MCUCR&0xFE,MCUCR);
  }
  else{
```

```
    count_blinkers=inp(TCNT2);
    if(count_blinkers>188)blinkers=2;
    else if(count_blinkers<158)blinkers=1;
    else blinkers=0;
    outp(MCUCR|0x03,MCUCR);
    }
}
/* interrupt routine for measuring the mode-select from radio reciever*/
SIGNAL(SIG_INTERRUPT1){
  if((inp(MCUCR)&0x0c)==12){
    outp(0x00,TCNT2);
    outp(BV(CS22)|BV(CS20), TCCR2); //TCNT0=CLK/256
    outp(MCUCR & ~BV(ISC10),MCUCR); // negative edge trigger
  }
  else{
    mode=inp(TCNT2);//counter2;
    if(mode>190)TWIdata[2]=2;
    else if(mode<150)TWIdata[2]=1;
    else TWIdata[2]=0;
    counter2=0;
    outp(MCUCR|BV(ISC10),MCUCR); //  positive edge trigger
  }
}

/* interrupt routine for switching blinkers on and off with certain timing*/
SIGNAL(SIG_OVERFLOW0){
  static short ijk=0;
  switch(blinkers){
  case 0:
    ijk=0;
    outp(0xDF&PORTB,PORTB);//  PB5=0
    outp(0xEF&PORTB,PORTB);//  PB4=0
    break;
  case 1://Right turn
    outp(0xEF&PORTB,PORTB);//  PB4=0
    ijk++;
    if(ijk<13)outp(0x20|PORTB,PORTB);//PB5=1
    else outp(0xDF&PORTB,PORTB);//  PB5=0
    if(ijk>30)ijk=0;
    break;
  case 2://Left turn
    outp(0xDF&PORTB,PORTB);//  PB5=0
    ijk++;
    if(ijk<13)outp(0x10|PORTB,PORTB);//PB4=1
    else outp(0xEF&PORTB,PORTB);//  PB4=0
    if(ijk>30)ijk=0;
    break;
  }
}
```

```
/*functions for serial communications via RS232. Used only in
developement for debugging purposes*/
SIGNAL(SIG_UART_RECV){
  char ch = inp(UDR);
}
static void putchar(unsigned char ch){
  while ((inp(UCSRA) & 0x20) == 0) {};
  outp(ch, UDR);
  while ((inp(UCSRA) & 0x20) == 0) {};
}

static void sendData(short data, uint8_t id){
  putchar(id);
  putchar(data>>8);
  putchar(data);
}

/*Interrupt routine for analog to digital conversions*/
/*Five different signals are converted but only four is used. The
reason for the fifth conversion is simply to reduce the sampling
frequency slightly*/
SIGNAL(SIG_ADC){
  static short i=0;
  uint8_t hbyte,lbyte;
  lbyte=inp(ADCL);
  hbyte=inp(ADCH);
  wheel[i]=(hbyte<<8)| lbyte;

/*When both channels from one wheel has been converted a call to
encoder_resolve is made*/

  if(i%2==1){
    encDeriv[i/2] = encoder_resolve(wheel[i-1],wheel[i],&encod[i/2]);
  }
  i++;
  if(i>4){
      i=0;
  }
  outp(BV(REFS0)|i,ADMUX); //next interrupt ADC[i]
  outp(BV(ADEN)|BV(ADSC)|BV(ADIE)|BV(ADPS2)|BV(ADPS1),ADCSR);
  // Internal Vref, right adjust, Enable ADC interrupts, Clock/64
}

/*interrupt routine used for serial data transfer to processor2*/
SIGNAL(SIG_2WIRE_SERIAL){ //Master_transmitter_mode
  static uint8_t i=0;

   if(index<10){
   TWIstatus[index]=inp(TWSR)&0xf8;
   index++;}
```

```
 switch(inp(TWSR)&0xf8){
 case 0x08:
   outp(0x34,TWDR);//SLA+W
   outp((BV(TWINT)|BV(TWEN)|TWCR)&~BV(TWSTO)&~BV(TWSTA),TWCR);
   break;
case 0x18:
  outp(TWIdata[i],TWDR);
  i++;
  if(i>3)i=0;
  outp((BV(TWINT)|BV(TWEN)|TWCR)&~BV(TWSTO)&~BV(TWSTA),TWCR);
  break;
 case 0x28:
   switch(i){
   case 0:
     outp(TWIdata[i],TWDR);
     i++;
     outp((BV(TWINT)|BV(TWEN)|TWCR)&~BV(TWSTO)&~BV(TWSTA),TWCR); break;
   case 1:
     outp(TWIdata[i],TWDR);
     i++;
     outp((BV(TWINT)|BV(TWEN)|TWCR)&~BV(TWSTO)&~BV(TWSTA),TWCR); break;
     break;
   case 2:
     outp(TWIdata[i],TWDR);
     i++;
     outp((BV(TWINT)|BV(TWEN)|TWCR)&~BV(TWSTO)&~BV(TWSTA),TWCR); break;
     break;
   case 3:
     TWIdata[i]=encDeriv[0]>>8;
     TWIdata[i+1]=encDeriv[0]&0x00ff;
     outp(TWIdata[i],TWDR);
     i++;
     outp((BV(TWINT)|BV(TWEN)|TWCR)&~BV(TWSTO)&~BV(TWSTA),TWCR); break;
     break;
   case 4:
     outp(TWIdata[i],TWDR);
     i++;
     outp((BV(TWINT)|BV(TWEN)|TWCR)&~BV(TWSTO)&~BV(TWSTA),TWCR); break;
     break;
   case 5:
     TWIdata[i]=encDeriv[1]>>8;
     TWIdata[i+1]=encDeriv[1]&0x00ff;
     outp(TWIdata[i],TWDR);
     i++;
     outp((BV(TWINT)|BV(TWEN)|TWCR)&~BV(TWSTO)&~BV(TWSTA),TWCR); break;
     break;
   case 6:
     outp(TWIdata[i],TWDR);
     i++;
```

```
      outp((BV(TWINT)|BV(TWEN)|TWCR)&~BV(TWSTO)&~BV(TWSTA),TWCR); break;
      break;
    case 7:
      i=0;
      outp((BV(TWINT)|BV(TWEN)|TWCR|BV(TWSTO))&~BV(TWSTA),TWCR);
      break;
    }
    break;
  default:
    outp(BV(TWINT),TWCR);  //disenable TWI temporarly
    // outp(BV(TWEA)|BV(TWINT)|BV(TWEN)|BV(TWIE),TWCR);
    break;
  }
}

short main(){
  int i,dataToSend,refToSend;

  outp(BV(ICES1)|BV(CS12)|BV(CS10),TCCR1B);//counter1 = clk/1024
  outp(BV(TOIE0)|BV(TICIE1),TIMSK); // Enable TCNT0  and Output Compare Match Interrupt
  outp(0x00,TCNT0);      // Reset TCNT00
  outp(0x00,TCNT2);      // Reset TCNT02
  outp(BV(CS02)|BV(CS00), TCCR0); //TCNT0=CLK/1024



  outp(0x11,TWBR);                        //TWI: BitRate TWI
  outp(BV(TWPS1),TWSR);                   //TWI: prescaler=16
  outp(0x32,TWAR);                        //TWI: slave-adress
  outp(BV(TWEA)|BV(TWEN)|BV(TWIE),TWCR); //TWI: Master_transmitter_mode


  outp(0x00, UCSRA);     // USART:
  outp(0x98, UCSRB);     // USART: RxIntEnable|RxEnable|TxEnable
  outp(0x86, UCSRC);     // USART: 8bit, no parity
  outp(0x00, UBRRH);     // USART: 38400 @ 14.7456MHz
  outp(23,   UBRRL);     // USART: 38400 @ 14.7456MHz


                                  // Enable ADC interrupts, Clock/64
  outp(BV(ADEN)|BV(ADSC)|BV(ADIE)|BV(ADPS2)|BV(ADPS1),ADCSRA);
  outp(BV(REFS0),ADMUX);            //ADC0 first time

  encod[0] = encoder2_init;
  encod[1] = encoder3_init;


  outp(BV(DDB4)|BV(DDB5),DDRB);       //define PortB4 and B5 as output
  outp(0xE7&PORTB,PORTB);             //write 0 to PortB4 and B5
```

```
/*rising edge of INT0 (PD2), and  INT1 (PD3), generates an interrupt request*/
  outp(BV(ISC11)|BV(ISC10)|BV(ISC01)|BV(ISC00),MCUCR);

/*external interrupt request enable*/
  outp(BV(INT0)|BV(INT1),GICR);
  i=0;

  sei();                            //set global interrupts enable
  while(1){
    cli();                          //disable global interrups
    if(intrptFlag==1){
      dataToSend=encDeriv[1];
      refToSend=encDeriv[0];
      sei();

/*sending some data to PC. This has no real purpose in the full system
but can be used for debugging*/
      sendData(dataToSend,2); //sends velocity of wheel1 to PC
      sendData(dataToSend,0);
      sendData(refToSend,1); //sends velocity of wheel0 to PC
      cli();
      intrptFlag= 0;
      }
    sei();
  }
}
```

```
/*bottom.c              Processor2    */

#include <avr/io.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
#define SKALA 8
#define INTGRSKALA 16
#define vecLen 75 number of samples in datalogg
volatile  uint8_t intrptFlag=0;
volatile  uint8_t safetyFlag=0;
volatile uint16_t wheel[5];
volatile  uint8_t TWIstatus=0;
volatile  uint8_t TWIdata[8];
volatile short Reference[vecLen];
volatile short Meassure[vecLen];
volatile short Control[vecLen];
volatile short encDeriv[4];
volatile long pi_I=0;
volatile short pi_K=50; //scaled by 2^8
volatile short pi_kb=50;
volatile short pi_integratorOn=1;
volatile short pi_cd=2500; //scaled by 2^16
volatile short pi_Tt_inv=20000;          // 1/Tt
volatile short lambdaBrake=7;
volatile short lambdaAcc=9;
volatile short *params[8];
volatile short pi_v=0;

struct encoder {
   signed int sin_min, sin_max,
     cos_min, cos_max, state, oldPos,counter,derivata, oldDerivata;
   char sin_gtz, cos_gtz;
   signed int offset, pos_interp;
   signed int pos_base;
};

struct encoder encod[2];
 /*initial values for encoders*/
 /*these values are different for each sensor*/
const struct encoder encoder0_init ={100,300,126,475,0,0,0,0,0,0,0,0,0,0};
const struct encoder encoder1_init ={207,385,192,446,0,0,0,0,0,0,0,0,0,0};

const arctan[128] =  {0, 1, 3, 4, 5, 6, 8, 9, 10, 12, 13, 14, 15, 17,18,
                      19, 20, 22, 23, 24, 25, 27, 28, 29, 30,32, 33, 34,
                      35, 37, 38, 39, 40, 41, 43, 44, 45, 46,47, 49, 50,
                      51, 52, 53, 54, 55, 57, 58, 59, 60, 61,62, 63, 64,
                      66, 67, 68, 69, 70, 71, 72, 73, 74, 75,76, 77, 78,
                      79, 80, 81, 82, 83, 84, 85, 86, 87, 88,89, 90, 91,
                      92, 93, 93, 94, 95, 96, 97, 98, 99, 100,100, 101,
                      102, 103, 104, 105, 105, 106, 107, 108,109, 109,
```

```
                    110, 111, 112, 113, 113, 114, 115, 116,116, 117,
                    118, 119, 119, 120, 121, 121, 122, 123,123, 124,
                    125, 125, 126, 127, 127, 128};

#define ENCODER_N 10
#define ENCODER_DELTA 256
const long res = 4*ENCODER_DELTA*ENCODER_N;

int abs(int x){
   return x < 0 ? -x : x;
}

short int sign(int x){
   if (x == 0)
     {
       return 0;
     }
   else
     {
       return x < 0 ? -1 : 1;
     }
}

/*Function used for calculating the angular velocity of the wheels
from the sensor values. This function is identical in both programs.*/

 short encoder_resolve(int sin_ch, int cos_ch, struct encoder *enc) {
  unsigned char state;
  long pos_interp, pos;
  short derivata, derivReturn;
  int sin_mid, cos_mid, sin_amp, cos_amp, sin_norm, cos_norm;

  sin_mid = (enc->sin_max + enc->sin_min)/2;
  cos_mid = (enc->cos_max + enc->cos_min)/2;
  sin_amp = (enc->sin_max - enc->sin_min)/2;
  cos_amp = (enc->cos_max - enc->cos_min)/2;

  sin_norm = ((sin_ch - sin_mid) *128)/sin_amp;
  cos_norm = ((cos_ch - cos_mid) *128)/cos_amp;
  if (abs(sin_norm) < abs(cos_norm))
    {
      if (cos_norm > 0) // state 0
        {
          state = 0;
          if (enc->sin_gtz != (sin_norm > 0))
            {
              enc->sin_gtz = (sin_norm > 0);
              enc->cos_max = cos_ch;
            }
          enc->pos_interp = sign(sin_norm)*arctan[abs((sin_norm*127)/cos_norm)];
```

```
        switch (enc->state)
          {
          case 1:
            enc->pos_base -= ENCODER_DELTA;
            break;
          case 3:
             enc->pos_base += ENCODER_DELTA;
             break;
          }
      }
    else // state 2
      {
        state = 2;
        if (enc->sin_gtz != (sin_norm > 0))
          {
            enc->sin_gtz = (sin_norm > 0);
            enc->cos_min = cos_ch;
          }
        enc->pos_interp = -sign(sin_norm)*arctan[abs((sin_norm*127)/cos_norm)];
        switch (enc->state)
          {
          case 1:
            enc->pos_base += ENCODER_DELTA;
             break;
          case 3:
            enc->pos_base -= ENCODER_DELTA;
            break;
          }
      }
  }
else
  {
    if (sin_norm > 0) // state 1
       {
         state = 1;
         if (enc->cos_gtz != (cos_norm > 0))
           {
             enc->cos_gtz = (cos_norm > 0);
              enc->sin_max = sin_ch;
           }
         enc->pos_interp = -sign(cos_norm)*arctan[abs((cos_norm*127)/sin_norm)];
         switch (enc->state)
           {
           case 0:
             enc->pos_base += ENCODER_DELTA;
             break;
           case 2:
            enc->pos_base -= ENCODER_DELTA;
             break;
           }
```

```
       }
    else // state 3
      {
        state = 3;
        if (enc->cos_gtz != (cos_norm > 0))
           {
             enc->cos_gtz = (cos_norm > 0);
             enc->sin_min = sin_ch;
           }
        enc->pos_interp = sign(cos_norm)*arctan[abs((cos_norm*127)/sin_norm)];
        switch (enc->state)
          {
          case 0:
            enc->pos_base -= ENCODER_DELTA;
            break;
          case 2:
             enc->pos_base += ENCODER_DELTA;
             break;
           }
      }
   }
enc->state = state;
pos = enc->pos_base + enc->pos_interp + enc->offset;
enc->counter++;

if(enc->counter>20){
  derivata=pos - enc->oldPos;
}
if (pos < -res/4) {
  enc->pos_base = enc->pos_base + res;
  pos = enc->pos_base + enc->pos_interp + enc->offset;
  enc->oldPos=enc->oldPos+res;
}
if (pos > 3*res/4) {
  enc->pos_base = enc->pos_base - res;
  pos = enc->pos_base + enc->pos_interp + enc->offset;
  enc->oldPos=enc->oldPos-res;

}


enc->oldDerivata=enc->derivata;
if(enc->counter>20){
  enc->counter=0;
  enc->oldPos=pos;
  enc->derivata=derivata;
}
```

```
  return enc->derivata;
}

void encoder_reset(struct encoder *enc) {
  enc->offset = -(enc->pos_base + enc->pos_interp);
}

/*Interrupt routine for safety check. If nothing has been received
from Processor1 in a certain time ,because of radio failure or other
problems, output to motor is put to zero, preventing the car from
running away*/
SIGNAL(SIG_OVERFLOW0){
  static short i=0;
  i++;
  if(i>5){
    i=0;
    if(!safetyFlag){
      safetyFlag=1;
    }
    else {
      Out(0);
    }
  }
}

/*Saturation function*/
int limit(int k,int pos_lim,int neg_lim ){
  if(k>pos_lim)return pos_lim;
  if(k<neg_lim) return neg_lim;
  else return k;
}
/*Function used for setting correct output signal to speed controller*/
void Out(int k){
  unsigned char lowbyte,highbyte;
  k=-k+1390;
  lowbyte=(unsigned char)k;
  highbyte=(unsigned char)(k>>8);
  outp(highbyte,OCR1BH);
  outp(lowbyte,OCR1BL);
}

/*Interrupt routine called when data is received on serial port RS232*/
SIGNAL(SIG_UART_RECV){
  static short i=0;
  static uint8_t id=0;
  static short temp=0;
  switch(i){
  case 0:
    id=inp(UDR); //First byte= header
    i++;
```

```
    break;
  case 1:
    temp= inp(UDR);
    i++;
    break;
  case 2:
    temp= (temp<<8)|inp(UDR);
    if(id==99){   // User has requested data transfer
      int k=0;
      sendData(-1024,5);
      for(k=0;k<vecLen;k++){
        sendData(Reference[k],4);
      }
      sendData(-1024,6);
      k=0;
      for(k=0;k<vecLen;k++){
        sendData(Meassure[k],4);
      }
      sendData(-1024,7);
      k=0;
      for(k=0;k<vecLen;k++){
        sendData(Control[k],4);
      }
    }
    else{
      *params[id]=temp;
    }
    i=0;
    break;
  }
}
/*Function to write one byte to the serial port RS232*/
static void putchar(unsigned char ch){
  while ((inp(UCSRA) & 0x20) == 0) {};
  outp(ch, UDR);
  while ((inp(UCSRA) & 0x20) == 0) {};
}
/*Function used to send one byte header and one 16-bit integer data on RS232*/
static void sendData(short data, uint8_t id){
  putchar(id);
  putchar(data>>8);
  putchar(data);

}
/*Interrupt routine to handle serial communication between Processor1 and 2*/
SIGNAL(SIG_2WIRE_SERIAL){
  static uint8_t i=0;
  TWIstatus=inp(TWSR)&0xf8;
  switch(inp(TWSR)&0xf8){
  case 0x60:
```

```
    outp((BV(TWEA)|BV(TWINT)|TWCR)&~BV(TWSTO),TWCR);
   break;
  case 0x80:
   TWIdata[i]= inp(TWDR);
   i++;
   if(i>6){
     i=0;
     intrptFlag=1;
     safetyFlag=0;
   }
   outp((BV(TWEA)|BV(TWINT)|TWCR)&~BV(TWSTO),TWCR);
   break;
  case 0xA0:
    outp((BV(TWEA)|BV(TWINT)|TWCR)&~BV(TWSTO),TWCR);
   break;
  default:
    outp(BV(TWEA)|BV(TWINT)|BV(TWEN)|BV(TWIE),TWCR);
    break;
  }
}

/*Function used to initialize the PWM unit*/
void initDAC() {
  outp(BV(DDB4)|BV(DDB5)|BV(DDB2)|0x07,DDRB);
  outp(BV(DDD7)|BV(DDD5)|BV(DDD4),DDRD);
  outp(0xFE&PORTB,PORTB);

  outp(BV(COM1A1)|BV(COM1B1),TCCR1A);
  outp(BV(CS11)|BV(WGM13),TCCR1B);//prescale: clk_io/8

/*Sets frequency equal to the frequency of the radioreceivers outputs~55 Hz*/
  outp(0x40,ICR1H);
  outp(0x9c,ICR1L); //sets frequency equal to the frequency of the radioreceiver

  outp(0x05,OCR1BH);
  outp(0x6e,OCR1BL);    //neutral value 0x056e=1390 max=0x0708=1800, min=0x03de=990
  outp(0x05,OCR1AH);
  outp(0x6e,OCR1AL);

}

/*Function used to calculating control signal*/
int piCalculateOutput(short ref, short mess){
  long t;
  int v;

  t=(pi_kb*(long)ref)/8-(long)(pi_K*(long)mess);
  v=(int)(t>>SKALA);
   if (pi_integratorOn) {
     v = v +(int)(pi_I>>INTGRSKALA);
```

```
  }
  pi_v=v;
  return v;
}

/*Funtion used to update state in the controller
(i.e increasing integrator part)*/
void piUpdateState(short ref, short mess,short v,short u){
  pi_I=pi_I+(pi_cd*(long)ref)/8-pi_cd*(long)mess+pi_Tt_inv*(long)v-pi_Tt_inv*(long)u;
}

/*Funtion used for approximating output*/
int approximateOutput(short ref, short mess){
return (int)((((pi_kb*(long)ref)/8-(long)(pi_K*(long)mess))>>SKALA)
               +((pi_cd*(long)ref)/8-pi_cd*(long)mess)>>INTGRSKALA);
}


/*Interrupt routine for analog to digital conversions*/
/*Five different signals are converted but only four is used. The
reason for the fifth conversion is simply to reduce the
sampling frequency slightly*/
SIGNAL(SIG_ADC){
  static short i=0;
  uint8_t hbyte,lbyte;
  lbyte=inp(ADCL);
  hbyte=inp(ADCH);
  wheel[i]=(hbyte<<8)| lbyte;

/*When both channels from one wheel has been converted a call to
encoder_resolve is made*/

  if(i%2==1){
    encDeriv[i/2] = encoder_resolve(wheel[i-1],wheel[i],&encod[i/2]);
  }
  i++;
  if(i>4){
      i=0;
  }
  outp(BV(REFS0)|i,ADMUX); //next interrupt ADC[i]
  outp(BV(ADEN)|BV(ADSC)|BV(ADIE)|BV(ADPS2)|BV(ADPS1),ADCSR);
  // Internal Vref, right adjust, Enable ADC interrupts, Clock/64
}

void initADC(){
  int i;
  uint8_t hbyte,lbyte;
  outp(0x10|PORTB,PORTB);//PB4=1
  outp(BV(REFS0),ADMUX);
  outp(BV(REFS0),ADMUX);//ADC0 first time
```

```
  // Enable ADC interrupts, Clock/64
  outp(BV(ADEN)|BV(ADSC)| BV(ADIE) |BV(ADPS2) |BV(ADPS1),ADCSRA);
 }

short main(){
  int i,dataToSend,refToSend;
  static int d0,d1,d2,d3=0;
  static int j=0;  // index for datalogg
  static int h=0; // indicates if the car has any high velocity
  static int m=0; //index to count samples when braking backwards
  static int k=110;
//startvalue for controlsignal when accelerating from standstill

  static int v=0;
  uint8_t brake=0;//when braking brake=1
  short R=0;
  unsigned char lowbyte,highbyte;
  int u,driverRef,lambda,min,max;
  uint8_t mode=0;
  i=0;
  params[0]=&pi_K;
  params[1]=&pi_kb;
  params[2]=&pi_integratorOn;
  params[3]=&pi_cd;
  params[4]=&pi_Tt_inv;
  params[5]=&lambdaBrake;
  params[6]=&lambdaAcc;

 outp(0x00, UCSRA);     // USART:
 outp(0x98, UCSRB);     // USART: RxIntEnable|RxEnable|TxEnable
 outp(0x86, UCSRC);     // USART: 8bit, no parity
 outp(0x00, UBRRH);      // USART: 38400 @ 14.7456MHz
 outp(23,   UBRRL);      // USART: 38400 @ 14.7456MHz

 outp(0x11,TWBR);        //BitRate TWI
 outp(BV(TWPS1),TWSR);   //prescaler=16
 outp(0x34,TWAR);        //slave-adress
 outp(BV(TWEA)|BV(TWEN)|BV(TWIE),TWCR);

 initDAC();

 encod[0] = encoder0_init;
 encod[1] = encoder1_init;

 initADC();
 outp(BV(TOIE0),TIMSK);
// Enable TCNT0  and Output Compare Match Interrupt
 outp(0x00,TCNT0);      // Reset TCNT00
 outp(0x00,TCNT2);
 outp(BV(CS02)|BV(CS00), TCCR0); //TCNT0=CLK
```

```
 outp(BV(CS21), TCCR2);

 //rising edge of INT0 (PD2) generates an interrupt request
 outp(BV(ISC01)|BV(ISC00),MCUCR);
 //external interrupt request enable
 outp(BV(INT0),GICR);
 sei();

 while(1){
   cli();
   i=intrptFlag;
   sei();
   if(i){
     cli();
     d0=-encDeriv[0];
     d1=(-encDeriv[1]+d0+d1)/3;
//average speed of rear wheels, LP-filtered
     d2=-((TWIdata[3]<<8)|TWIdata[4]);
     d3=(-((TWIdata[5]<<8)|TWIdata[6])++d2+d3)/3;
     //average speed of front wheels, LP-filtered

     driverRef=(TWIdata[0]<<8)|TWIdata[1];
//driverRef is between 990-1800, neutral=1390
     mode=TWIdata[2];

     sei();//interrupts enable
     R=driverRef-1390;
     switch(mode){

  /*Manual drive, controller off*/
     case 0:
       j=0;
       Out((driverRef-1390));
       brake=0;
       R=0;
       cli();
       pi_I=0;
       sei();
       h=0;
       k=110;
       m=0;
       break;

  /*Forward with controller*/
    case 2:
       if(d3< -25){ /*if the car is travelling backwards,
                brake to standstill before going forward*/
          if(m<15){
            u=piCalculateOutput(d3*lambdaBrake,d1);
            if(u>0){
```

```
            u=u+42;
            cli();
            pi_integratorOn=1;
            /* pi_I=0; */
            sei();
          }
          v=limit(u,390,0);
          Out(v);
          piUpdateState(d3*lambdaBrake,d1,v,u);
          m++;
        }
        else if(m<18){
          v=-43;
          Out(v);
          m++;
        }
        else{
          v=-43;
          Out(v);
          m=0;
        }
        brake=1;
        h=0;
        k=110;
        if(j<vecLen){
          Reference[j]=d3;
          Meassure[j]=d1;
          Control[j]=v;
          j++;
        }
        break;
      }
      else  if(abs(d3)<5){
//if the car is allmost att standstill
        h=0;
        m=0;
        brake=0;
        cli();
        pi_integratorOn=1;
        pi_I=0;
        sei();
      }
       if(abs(d3)<150 && !brake && !h){//starting to accelerate
        if(driverRef>1400){
          k++;
          u=k>>1;
          Out(u);
          pi_I=(long)(k*18000);
//required for a smooth transition
//when controller is switched on
```

```
          }
          else Out(0);
          m=0;
        }
        else{
          min=approximateOutput(d3*lambdaBrake,d1)-42;
          max=approximateOutput(d3*lambdaAcc,d1)+42;
          if(R>(u+10) && (R>max || d1>((lambdaAcc*d3)/8))){
/*If driver is trying to accelerate to fast*/
            u=piCalculateOutput(d3*lambdaAcc,d1);
            if(u>0){
              u=u+42;
              cli();
              pi_integratorOn=1;
              sei();
            }
            v=limit(u,R,45);
            Out(v);
            piUpdateState(d3*lambdaAcc,d1,v,u);
            brake=0;
            h=1;
            m=0;
            if((j<vecLen)&& (d3 > 200)){
//saving data from acceleration
              Reference[j]=d3;
              Meassure[j]=d1;
              Control[j]=v;
              j++;
            }
          }
         else if((R < -10)){// braking
           if(h){
             j=0;
             cli();
             pi_I=0;
             sei();
           }
           u=piCalculateOutput(d3*lambdaBrake,d1);
           if(u<0){
             u=u-42;//compensating for deadzone
             cli();
             pi_integratorOn=1;
             pi_I=0;
             sei();
           }
           else{
             cli();
             pi_integratorOn=0;
             sei();
           }
```

```
            v=limit(u, 0, -390);
            Out(v);
            piUpdateState(d3*lambdaBrake,d1,v,u);
            brake=1;
            h=0;
            k=110;
            m=0;
               if(j<vecLen){//datalogg
             Reference[j]=d3;
            Meassure[j]=d1;
            Control[j]=v;
            j++;
            }
          }
          else {
/*If the drivers demands is within the limits,
      output drivers control signal*/
            u=R;
            Out(u);
            pi_I=0;
            brake=0;
            k=110;
            m=0;
          }

        }
        break;

 /*Reverse with controller on*/
    case 1:
       if(d3>25){/*if the car is travelling forward,
                  brake to standstill before reversing*/
          u=piCalculateOutput(d3*lambdaBrake,d1);
          if(u<0){
            u=u-42;
            cli();
            pi_integratorOn=1;
            pi_I=0;
            sei();
          }
          else{
            cli();
            pi_integratorOn=0;
            sei();
          }
          v=limit(u,0,-390);
          Out(v);
          piUpdateState(d3*lambdaBrake,d1,v,u);
          h=0;
          brake=1;
```

```
  k=110;
  m=0;
  if(j<vecLen){
    Reference[j]=d3;
    Meassure[j]=d1;
    Control[j]=v;
    j++;
  }
  break;
}
else  if(abs(d3)<5){
  h=0;
  brake=0;
  cli();
  pi_integratorOn=1;
  pi_I=0;
  sei();
  m=0;

}
if(abs(d3)<150 && !brake && !h){
  if(driverRef<1380){
    k++;
    u=-k>>1;
    Out(u);
    pi_I=(long)(-k*18000);
  }
  else Out(0);
  m=0;
}
else{
  min=approximateOutput(d3*lambdaBrake,d1)+42;
  max=approximateOutput(d3*lambdaAcc,d1)-42;
  if(R<(u-10) && (R<max || d1<((lambdaAcc*d3)/8))){
    u=piCalculateOutput(d3*lambdaAcc,d1);
    if(u<0){
      u=u-42;
      cli();
      pi_integratorOn=1;
      sei();
    }
    v=limit(u,-45,R);
    Out(v);
    piUpdateState(d3*lambdaAcc,d1,v,u);
    brake=0;
    h=1;
    m=0;

    if(j<vecLen&& abs(d3)>200){
      Reference[j]=d3;
```

```
            Meassure[j]=d1;
            Control[j]=v;
            j++;
          }
        }
       else if((R > 10)){//Braking
         if(h){
           j=0;
           cli();
           pi_I=0;
           sei();
           m=0;
         }

         if(m<15){
           u=piCalculateOutput(d3*lambdaBrake,d1);
           if(u>0){
             u=u+42;
             cli();
             pi_integratorOn=1;
             sei();
           }
           v=limit(u,390,0);
           Out(v);
           piUpdateState(d3*lambdaBrake,d1,v,u);
           m++;
         }
         else if(m<18){
// trying to fool the speed controller by
//changing sign for a few samples.
           v=-43;
           Out(v);
           m++;
         }
         else{
           v=-43;
           Out(v);
           m=0;
         }
         brake=1;
         h=0;
         k=110;
          if(j<vecLen){//log data
          Reference[j]=d3;
          Meassure[j]=d1;
          Control[j]=v;
          j++;
          }
       }
        else {
```

```
            u=R;
            Out(u);
            pi_I=0;
            brake=0;
            k=110;
            m=0;
          }
        }
     break;
     }
     sendData(d1,2);
     sendData(d1,0);
     sendData(d0,1);
     if(brake) outp(0x20|PORTB,PORTB);//PB5=1 Brake lights on
     else outp(0xDF&PORTB,PORTB); //PB5=0  Brake lights off
     i=0;
     cli();
     intrptFlag=0;
     sei();
    }
  }
}
```

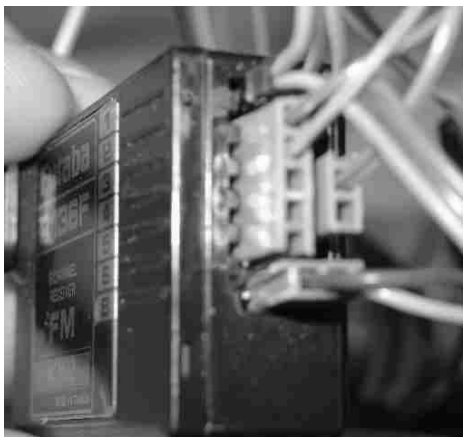
Bilaga C

Kopplingsschema

I tabellerna nedan finns beskrivning av hur processorkorten är sammankopplade med varandra och yttre enheter. J6(3)PD6(ICP1) betyder pinne 3 i kontakt J6 på processorkortet samt att denna på kretkortet är ansluten till PortD pinne 6 på processorn. (ICP1) står för Input Capture 1 och innebär att pinnen förutom att användas som generell digital I/O även kan användas till denna specialfunktion. Om det bara står t.ex. J5 innebär det att alla pinnarna i J5 skall kopplas till angiven kontakt i rätt inbördes ordning.

Pinnarna i kontakterna på kretskorten är indexerade från 1 men för processorns pinnar börjar indexeringen på 0.

undre processorkortet	*till kontakt*
J3 Power Out	J8 Power in på övre processorkortet
J5 TWI	J5 TWI på övre processorkortet
J6(1)GND	bromsljus GND
J6(2)GND	GND till fartreglaget, svart
J6(5)PD4(OC1B)	Styrsignal till fartreglaget, vit
J6(10)PB5	bromsljus +
J7(1)GND	GND på sensorkortet
J7(3)PA0(ADC0)	VB pulseA på sensorkortet
J7(4)PA1(ADC1)	VB pulseB på sensorkortet
J7(5)PA2(ADC2)	HB pulseA på sensorkortet
J7(6)PA3(ADC3)	HB pulseB på sensorkortet
J7(20)+5V	+5V på sensorkortet
J8(1) Power in GND	- på batterihållaren
J8(3) Power in +5V	+ på batterihållaren

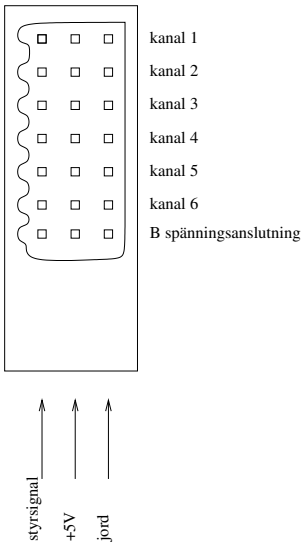


Figur C.1: Anslutningar till radiomottagaren

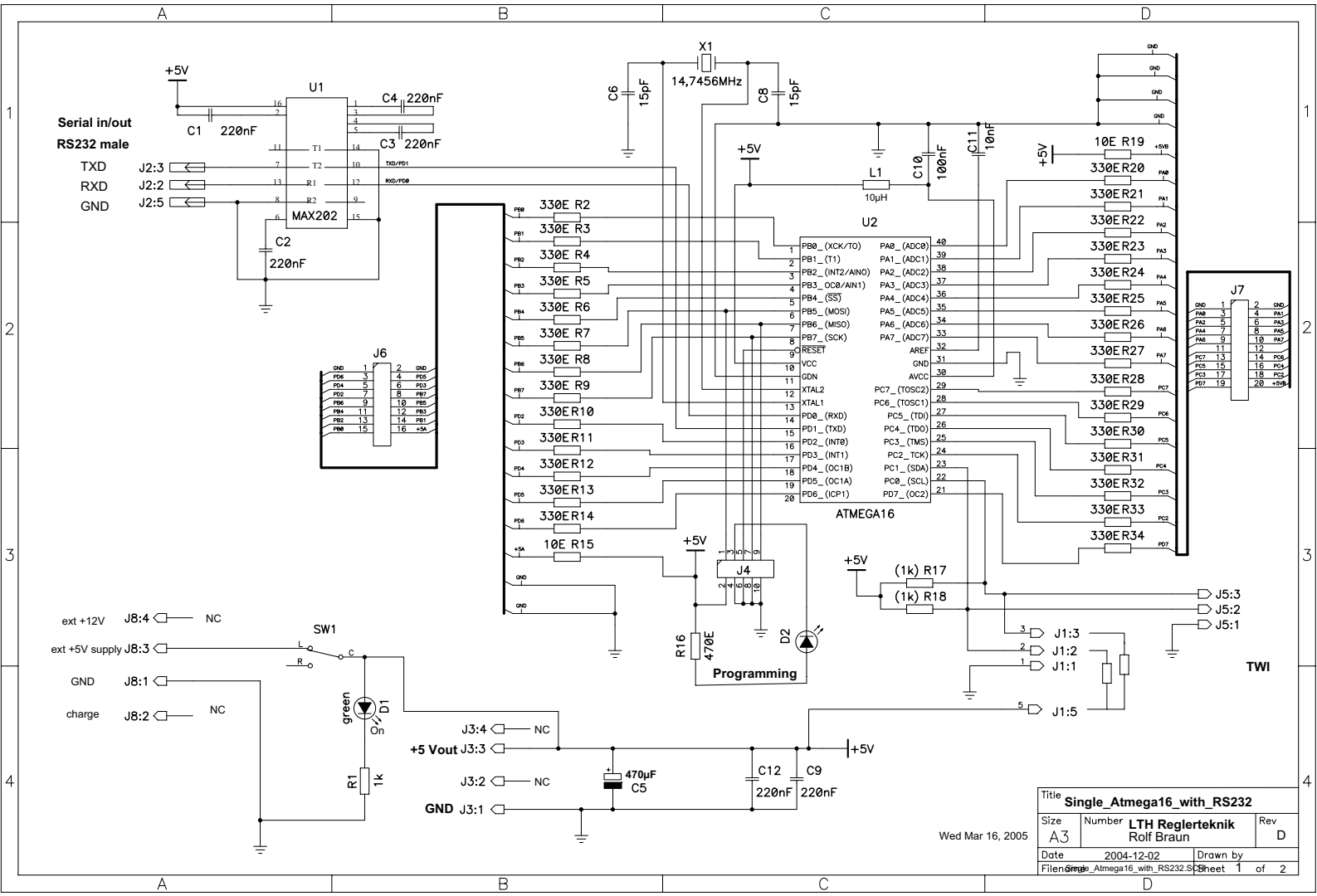
övre processorkortet	*till kontakt*
J3(1) Power Out GND	GND B spänningsanslutning radiomottagaren
J3(3) Power Out +5V	+5V B spänningsanslutning radiomottagaren
J5 TWI	J5 TWI på nedre processorkortet
J6(1)GND	GND radiomottagaren
J6(2)GND	GND höger och vänster blinkers
J6(3)PD6(ICP1)	styrsignal kanal 2 radiomottagaren
J6(6)PD3(INT1)	Styrsignal kanal 3 radiomottagaren
J6(7)PD2(INT0)	Styrsignal kanal 1 radiomottagaren, Kanal 1 kopplas även vidare till styrservot
J6(10)PB5	höger blinkers +
J6(11)PB4	vänster blinkers +
J7(3)PA0(ADC0)	VF pulseA på sensorkortet
J7(4)PA1(ADC1)	VF pulseB på sensorkortet
J7(5)PA2(ADC2)	HF pulseA på sensorkortet
J7(6)PA3(ADC3)	HF pulseB på sensorkortet
J8(1) Power in	J3 Power Out på nedre processorkortet

Sensorerna på hjulen kopplas till respektive kontakter på sensorkortet. Kontakterna ansluts så att de svarta markeringarna på stiften respektive hylslisterna sammanfaller.

Fartreglagets kablar ansluts till motorn med vit till grön och blå till gul så att polariteten blir rätt dvs att bilen kör framåt när styrspaken förs framåt vid *manuell körning*



Figur C.2: Principskiss över radiomottagarens kontakter sedd från samma sida som bilden i figur C.1. Den nedersta raden är kopplad till batterispänning, de övriga är utsignaler till servon. Kontakterna i den vänstra kollumnen är styrsignaler, den mittersta +5V och den högra jord


Serial in/out
RS232 male
TXD J2:3
RXD J2:2
GND J2:5
MAX202
U1
C1 220nF
C2 220nF
C3 220nF
C4 220nF
X1
14,7456MHz
C6 15pF
C8 15pF
C10 100nF
C11 10nF
L1 10µH
U2
ATMEGA16
330E R2
330E R3
330E R4
330E R5
330E R6
330E R7
330E R8
330E R9
330E R10
330E R11
330E R12
330E R13
330E R14
10E R15
10E R19
330E R20
330E R21
330E R22
330E R23
330E R24
330E R25
330E R26
330E R27
330E R28
330E R29
330E R30
330E R31
330E R32
330E R33
330E R34
(1k) R17
(1k) R18
R16 470E
D2
Programming
J4
J6
J7
J5:3
J5:2
J5:1
J1:3
J1:2
J1:1
J1:5
TWI
ext +12V J8:4 NC
ext +5V supply J8:3
GND J8:1
charge J8:2 NC
SW1
green D1
On
R1 1k
+5 Vout J3:3
J3:4 NC
J3:2 NC
GND J3:1
470µF C5
C12 220nF
C9 220nF
Wed Mar 16, 2005
Title Single_Atmega16_with_RS232
Size A3
Number LTH Reglerteknik Rolf Braun
Rev D
Date 2004-12-02
Drawn by
Filename Single_Atmega16_with_RS232.SCH
Sheet 1 of 2

I/O 1 J6 16 pin *

1 GND
2 GND
3 PD 6 (ICP1)
4 PD 5 (OC1A)
5 PD 4 (OC1B)
6 PD 3 INT 1)
7 PD 2 (INT 0)
8 PB 7 (SCK)
9 PB 6 (MISO)
10 PB5 (MOSI)
11 PB4 (SS)
12 PB3 (OC0/AIN1)
13 PB2 (INT2/AIN0)
14 PB1 (T1)
15 PB0 (XCK/T0)
16 +5V (10E)

I/O 1 J7 20 pin *

1 GND
2 GND
3 PA 0 (ADC0)
4 PA 1 (ADC1)
5 PA 2 (ADC2)
6 PA 3 (ADC3)
7 PA 4 (ADC4)
8 PA 5 (ADC5)
9 PA 6 (ADC6)
10 PA7 (ADC7)
11 NC
12 NC
13 PC7 (TOSC2)
14 PC6 (TOSC1)
15 PC5 (TDI)
16 PC4 (TDO)
17 PC3 (TMS)
18 PC2 (TCK)
19 PD7 (OC2)
20 +5V (10E)

TWI J5 3 pin **

1 gnd
2 SDA
3 SCL

Power in J8 4 pin

1 gnd
2 nc
3 + 5 V
4 nc

RS 232 J2 D-sub 9 pin

1 gnd (5)
2 RXD (2)
3 TXD (3)
4 NC

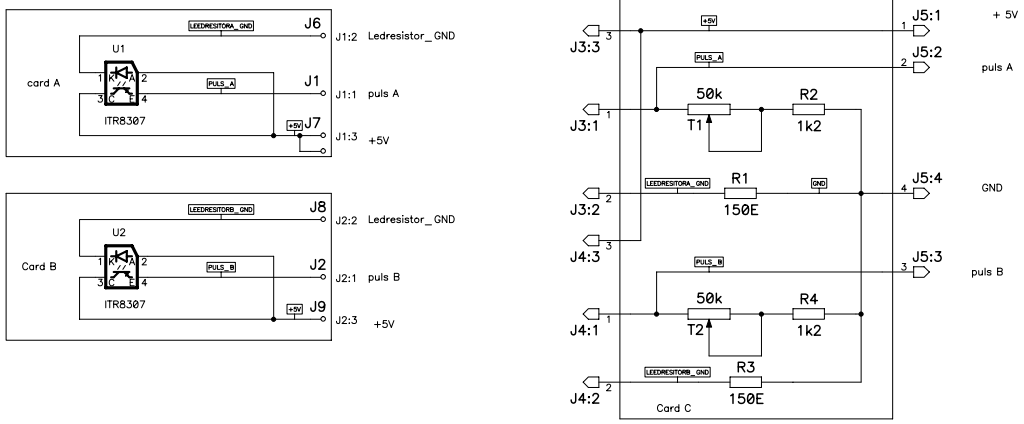
Power out J3 4 pin

1 gnd
2 nc
3 + 5 V
4 nc

* I/O ports have 330 ohm serial resistors. +5V 10 ohm.

** Only pull-up resistor mounted at the last card.

Title	**Single_Atmega16_with_RS232**	
Size A3	Number **LTH Reglerteknik** Rolf Braun	Rev D
Date 2004-12-02		Drawn by
Filename Single_Atmega16_with_RS232.SchDoc		Sheet 2 of 2



Title	Encoder8307 AWDcar		
Size A3	Number LTH Reglerteknik Rolf Braun		Rev A
Date	2004-10-01	Drawn by	
Filename	encoder8307_AWDcar.Sch1	Sheet 1 of 1	

Wed Mar 16, 2005