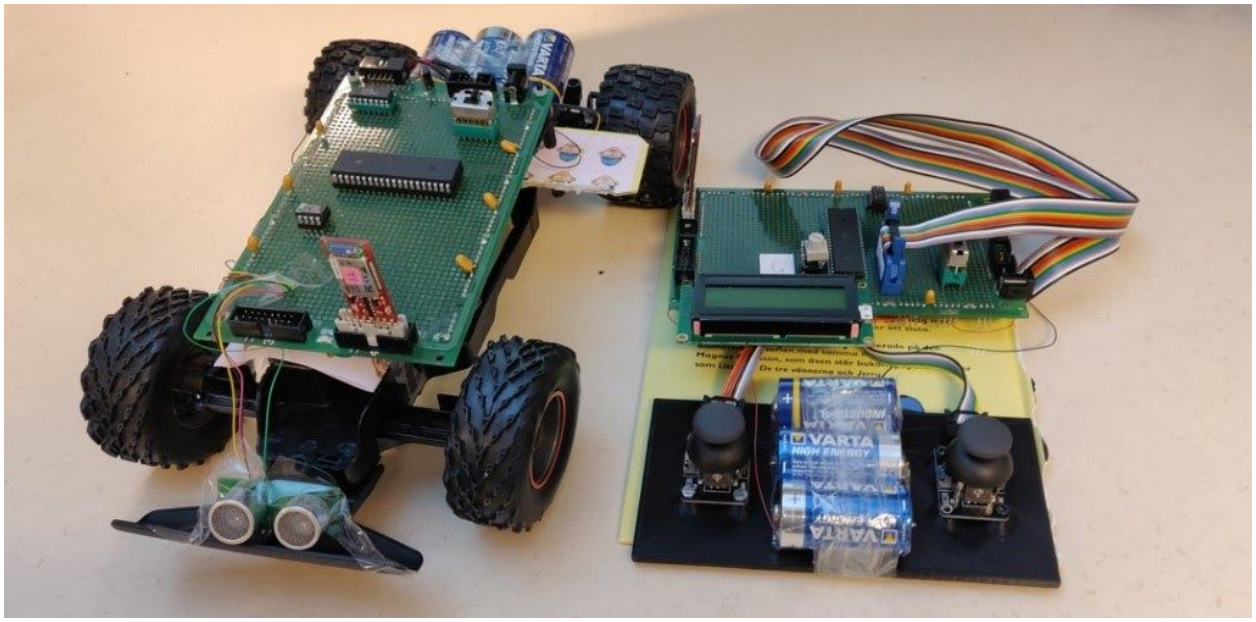
Bluetooth-styrd bil

Mikrodatorprojekt - Grupp 14



Dutsadi Bunliang (dutbu586@student.liu.se),
Sebastian Börjeteg (sebbo952@student.liu.se),
Hamid Gholami (hamgh740@student.liu.se),
Wiktor Isaksson (wikis954@student.liu.se)

Linköpings universitet

VT -19

Sammanfattning

Denna rapport handlar om ett projekt, rapporten förklarar hur man bygger en Bluetooth-styrd bil med tillhörande kontroll som kommunicerar med varandra med hjälp av bluetooth. Rapporten inkluderar det som gjorts under projektets gång och går igenom alla delar i projektet.

Rapporten kommer beskriva viktiga komponenter och hur dessa fungerar. Under denna del kommer det även finnas en kort genomgång om vad man kan göra med komponenterna, alltså komponenternas funktionalitet. Här kan man även hitta motiveringar till varför vissa komponenter används och vad deras syfte är. ATmega16 är en viktig komponent men kommer inte täckas helt under rapporten.

Vår kod är en stor del av projektet och kommer förklaras med bl.a. JSP-diagram. Koden är till största del uppdelad efter komponenterna, dvs varje komponent har en del kod som hanterar den. Rapporten täcker även de två huvudprogram som innehåller allt som krävs vid sammanställning av kod-delen av projektet. Själva koden finns i bilagor och är uppkallad efter respektive enhet.

Avslutningsvis kommer rapporten att gå igenom upplägget och fördelningen som användes för att strukturera upp projektet under de första veckorna. Det sista rapporten kommer behandla är en genomgång av hur projektet har sett ut, hur det slutade och vad man har får ut av detta projekt.

Innehållsförteckning

1. Inledning

Den här rapporten beskriver hur det gick till att bygga en radiostyrd bil och en kontroll som styr denna bil. Rapporten kommer täcka alla delar i projektet, hårdvara som mjukvara.

1.1 Bilen

Bilen består av 6 komponenter. Dessa komponenter är en ultraljudssensor som används för avståndsmätning, en bluetooth-modul som är parad till en annan bluetooth-modul som är på kontrollen, 2 st motorer där ena motorn används för styrning och den andra används för hastigheten. Sedan finns det även en ATmega16 mikroprocessor på bilen som kopplar ihop alla komponenter och fungerar som hjärnan bakom allt, den tar hand om t.ex. avkodning av styrsignaler.

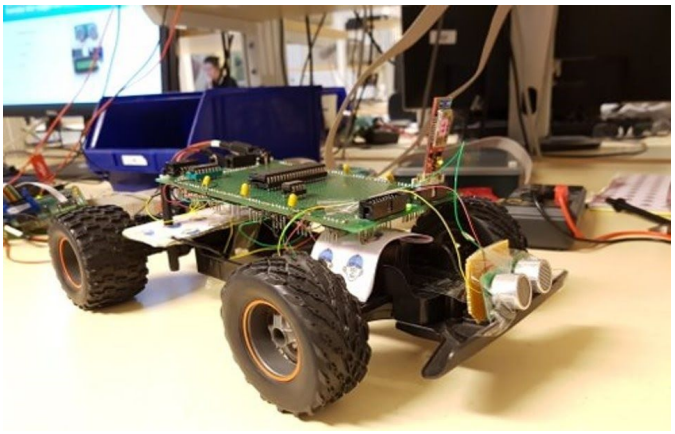


Bild 1: Bild på bilen.

1.2 Kontrollen

Kontrollen används som styrenhet för den bluetooth-styrda bilen. Kontrollen består av många olika komponenter, *Bild 2* visar vilka komponenter översiktligt.

LCD displayen har syftet att förmedla information från bilen, den visar avstånd till närmaste hinder framför bilen, på LCD:n visas hur mycket som vi kör framåt och hur mycket vi svänger.

1. LCD
2. Joysticks
3. Bluetooth-modem
4. Atmega16-mikroprocessor
5. 14.75 MHz kristallklocka
6. Potentiometer
7. På/Av knapp
8. Joystick-adapter

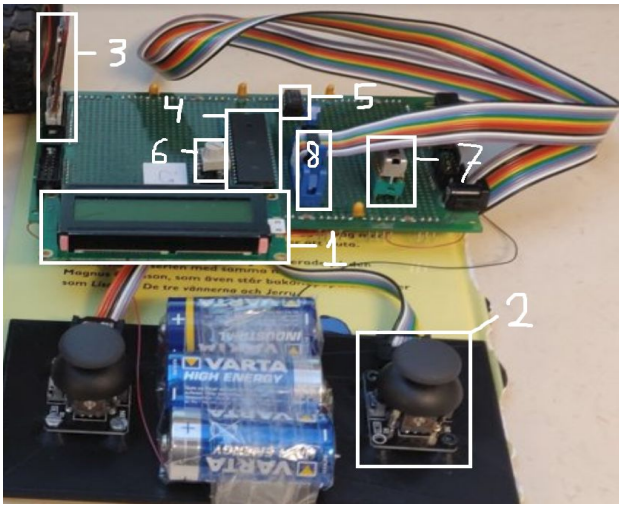


Bild 2: Visar några av komponenterna som sitter på kontrollen.

1.3 Blockschema

BIld 3 och *4* visar översiktligt hur komponenterna är sammanlänkade.

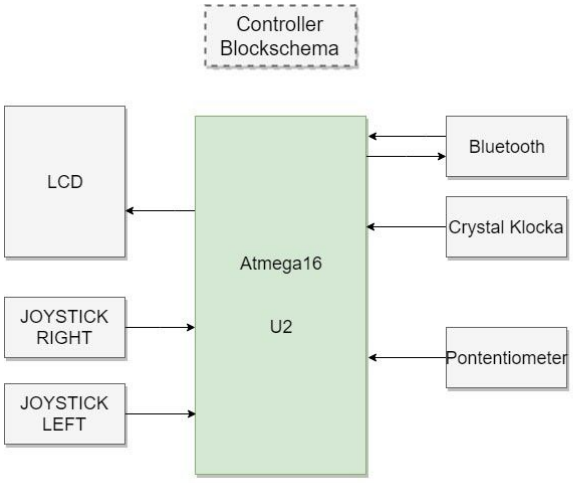


Bild 3: Blockschema för kontroller.

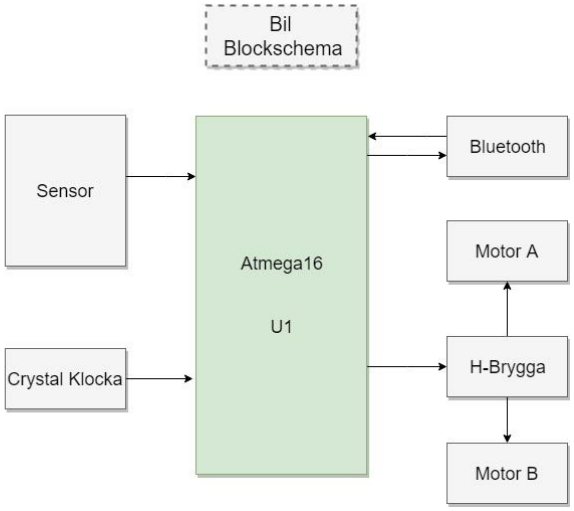


Bild 4: Blockschema för bilen.

1.4 Syfte

Syftet med detta projekt var att få en djupare förståelse inom ämnet och bli mer insatta i språket assembler, men även att utveckla de kunskaperna från föregående kurs (TSIU02). En till stor del var felsökning, eftersom att projektet var så pass stort bidrar detta till en stor del av projektet. Projektet ger även en bra insikt i hur det är att planera och förbereda ett projekt, jobba i grupp och att arbeta utifrån kravspecifikation.

1.5 Kravspecifikation

Kraven för projektet delas i två kategorier. Första kategorin är skallkrav och den andra är börkrav. Projektkrav måste vara färdigt implementerat i slutprodukten. Däremot är börkrav inte lika strikta, då de kan implementeras i mån av tid.

Skallkrav

- Bilen skall kunna styras via en bluetooth kontroller
- Bilen skall använda sensorer för att upptäcka hinder
- Kontrollen skall kunna varna användaren om hinder
- Kontrollen skall ha två joystickar

Börkrav

- Kontrollen bör kunna spela ljud
- Kontrollen bör ha en display

2. Hårdvara

Under den här delen kommer all hårdvara som används tas upp och hur allt är sammankopplat hos både kontrollen och bilen.

2.1 ATmega16

En Atmega16 är en mikroprocessor utrustad med 40 pinnar varav 32 pinnar kan användas som I/O (input/output) pinnar. Mikroprocessorn har många olika inbyggda funktioner. I denna projekt använder vi bland annat USART(*Universal Synchronous and Asynchronous Receiver and Transmitter*) för kommunikation mellan kontrollen och bilen. Processorn kan kontrolleras med hjälp av kodning i språket assembler. Den är hjärtat av vårt projekt och styr alla komponenter. Bilen och kontrollen har en processor var.

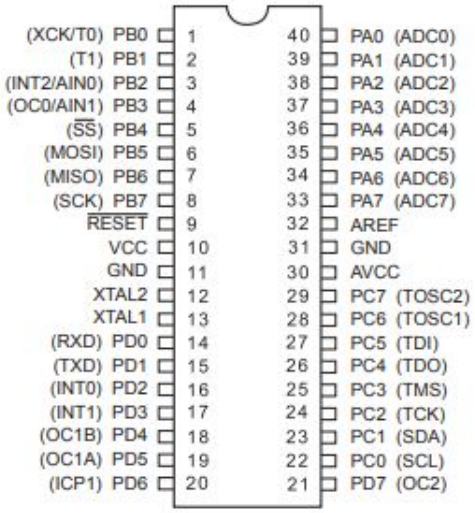


Bild 5: Pinupplägg för ATmega16 (Atmel)

2.1.2 JTAG

JTAG används som en länk mellan datorn som man programmerar på och processorn, den används för att få koden på datorn in i processorn.

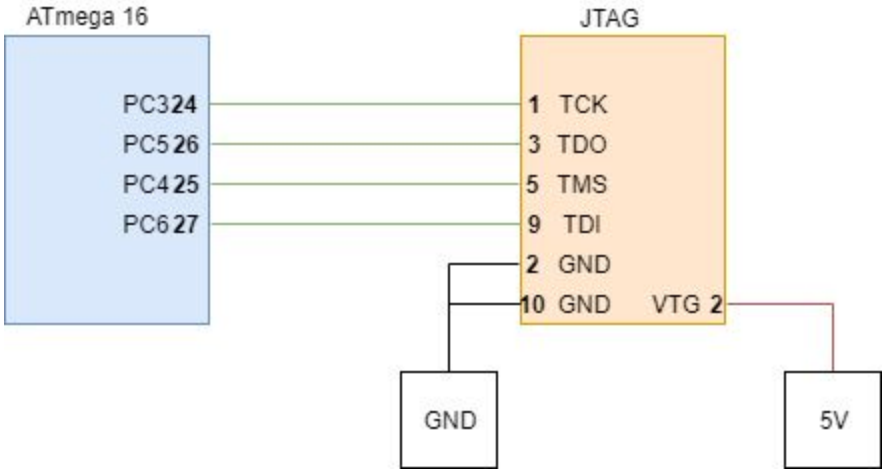


Bild 6: Kopplingsschema för JTAG-gränssnitt med ATmega16. Med hjälp av JTAG ICE 3 och programvaran Atmel Studio kan ATmega16 programmeras.

2.2 LCD

För att kunna visa sensorns data används en 2x16 display. Displayen består av en del olika ingångar, RS avgör om man ska skicka instruktioner eller data och R/W avgör om det ska läsas av displayen eller skickas till displayen. De åtta databitar på LCD:n används för att välja instruktion och skicka in våra databitar. LCD:n kräver att man ger den signaler i en viss ordning för att kunna skriva till den, lite som ett kodlås. Alltså varje gång man ska göra något med displayen måste man göra detta. Detta gäller för både instruktionen och skrivningen. Funktion fungerar på detta sätt:

1. Sätt RS och R/W. Beroende på situation ändras RS, R/W kommer förbli låg då projektet aldrig kräver att läsning används
2. Skicka in data-bitarna
3. Delay
4. Sätt Enable till ett
5. Delay
6. Sätt Enable till noll (här tar displayen in data-bitarna)
   Varje gång något ska skrivas till displayen måste denna funktion användas för att få in vår data till displayen.

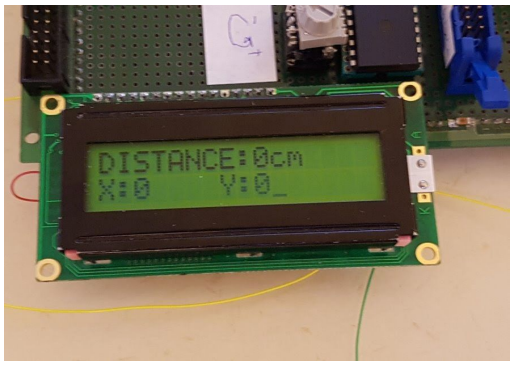


Bild 7: Bild på LCD.

Displayen är kapabel att utföra tio olika instruktioner och kunna skriva till displayen. De olika instruktionerna säger hur displayens pekare och liknande ska fungera. Endast sju av dessa instruktioner användes under projektets gång. Här är de förklarade:

Tabell 1: Visar funktioner för LCD-displayen.

Namn	**Beskrivning**
clear display	Den rensar skärmen och ställer pekaren på förstaplatsen
return home	Flyttar pekaren till början utan att rensa skärmen
display on/off	Denna används för att initiera displayen alltså slå igång den, sätta igång pekaren och välja om den ska blinka eller inte
cursor or display shift	I denna instruktion bestäms hur pekare och display ska bete sig när man byter rad på displayen
function set	Här bestäms saker som 8-bitars mode, vilket betyder att alla 8 ingångar på displayen istället för att använda 4 och köra seriellt, sen bestäms även 2-line mode vilket säger helt enkelt att via har en 2x16 display.
set DDRAM address	Här sätts var displayen ska skriva
entry mode set	Här initieras pekaren

Och efter dessa finns de vanliga instruktionerna, skriva och läsa. Den enda skillnaden mellan hur man skickar en instruktion och hur man skickar in sina databitar är att man sätter RS till ett när man ska skicka datan.

Uppkoppling ser ut på detta vis. Här får man en övergripande bil på hur det ser ut.

Tabell 2: Pin-funktioner för LCD.

Namn	**Pin**	**Beskrivning**
GND	1	0 V
Vcc	2	5 V
V5	3	Ljusstyrka (med hjälp av en Potentiometer)
RS	4	Väljer om man ska skicka data eller instruktion
R/W	5	Read/Write bestämmer om man ska skriva till displayen eller ej
E	6	Enable (användning förklaras i mjukvaru delen)
Data-bitarna	7-14	8 st databitar
A	15	Anod till backlight
K	16	Katod till backlight

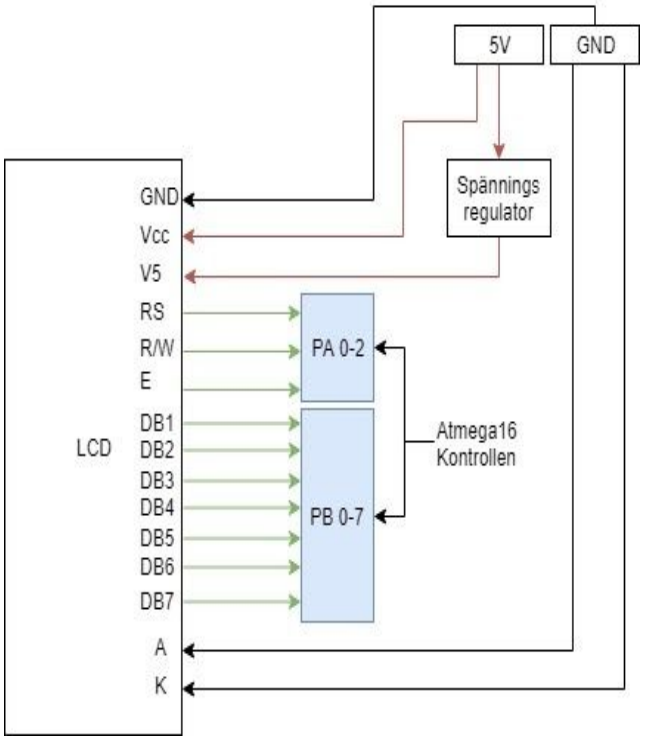


Bild 8: Kopplingsschema för LCD:n 2x16 respektive till ATmega16

2.3 Bluetooth

För att kunna kommunicera mellan kontrollen och bilen användes två parade bluetooth-modem. Bluetoothen använder processorns USART funktionalitet för att kommunicera via seriell kommunikation. För att få bluetoothen att fungera måste processorns inställningar för seriell kommunikation konfigureras korrekt och uppkopplingen mellan komponenterna vara som visas på kopplingsschema i bild 9.

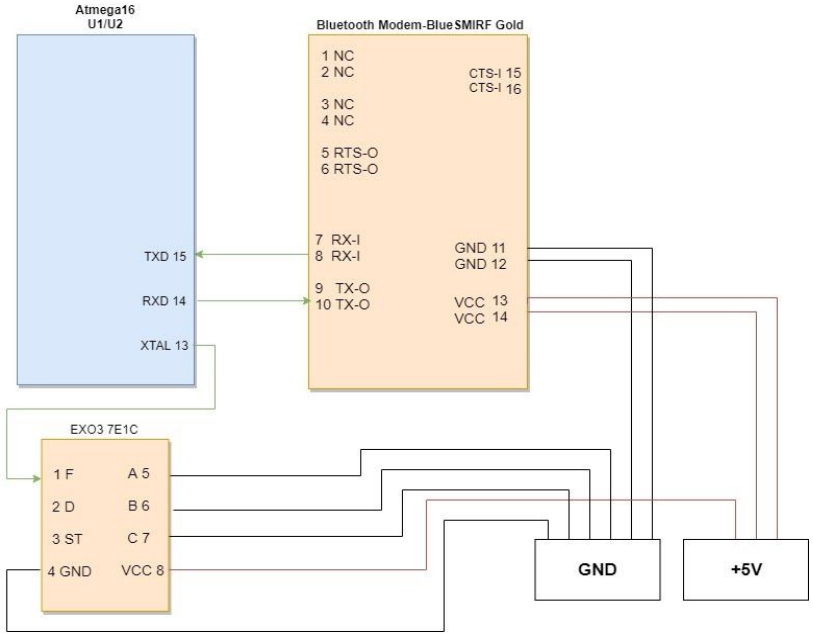


Bild 9: Kopplingsschema över bluetooth-modem, kristallklocka 14.75 Mhz respektive till ATmega16. Samma uppkoppling används på bilen och kontrollen.

För en djupare förståelse för hur bluetoothen fungerar ges en översikt för bluetooth-modemets pin-funktionalitet i tabell 3.

Tabell 3: Pin-funktioner för bluetooth-modem.

Namn	**Pin**	**Beskrivning**
NC	1, 2, 3, 4	Ingen anslutning(Används ej)
RTS-O	5, 6	Begär att skicka(Används ej)
RX-I	7, 8	Mottagning av seriell signal
TX-O	9, 10	Utskick av seriell signal
GND	11, 12	
VCC	13, 14	3.3-6 V
CTS-1	15, 16	Klart att skicka(Används ej)

Förutom uppkoppling enligt Bild 9 måste det även försäkras att processorn är inställd på samma baudrate som bluetooth-modemen. Baudraten bestämmer hur många bitar som kan skickas per sekund. Den är beroende på processorns systemklocka och kan konfigureras med hjälp av en prescaler register som heter UBRR, *Usart Baud Rate Register*.

Den interna systemklockan i Atmega16 mikroprocessorn är inte tillräckligt noggrann för att upprätthålla en konstant baudrate. Därför användes en extern kristallklocka, EXO 7E1C med frekvensen 14,75 MHz. Kristallklockan användes som systemklocka istället för den interna klockan.

När allt är korrekt konfigurerat fungerar bluetoothen som en kanal för seriell kommunikation där data man lägger i processorns UDR, *Usart Data Registren* skickas och tas emot trådlöst via bluetooth till den andra enheten.

I en seriell sändning skickas det först en startbit för att signalera start, sedan ett antal informationsbitar, en paritetsbit och sist skickas stoppbitar för att signalera slut, se *bild 10*.

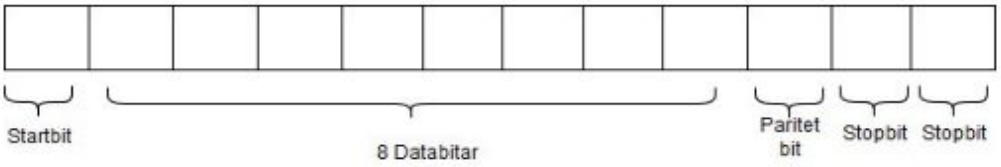


Bild 10: Visar hur seriell sändning ser ut.

Protokollet för vår konfiguration av seriell kommunikation är 8N2 med hastigheten 115,2 kbps. Protokollet beskriver hur överföringen av information sker. 8:an står för antalet informationsbitar, N står för ingen paritetskontroll och 2:an står för antalet stoppbitar som har valts.

En viktig notering är att det finns en buffer i bluetooth-modemet som sätter en gräns på hur snabbt *bytes* kan skickas efter varandra. För mycket trafik leder till att modemet inte hinner rensa sin interna buffer och därmed slutar att fungera.

2.4 PWM

Pulsbreddsmodulering, på engelskan *pulse width modulation* (PWM), är en funktion som finns i ATmega16. Hur PWM fungerar kan beskrivas med en fyrkantvåg, se *Bild 11*. En PWM-signal från ATmega16 genererar kontinuerligt en “fyrkantvåg” med en spänning mellan 0 och 5 volt. Perioden för“på” och “av” kallas cykeltid. Termen cykeltid använd för att beskriva en PWM-signals beteede och utalas oftas i procent. Exempelvis om en PWM-signal har en cykeltid på 33 procent menas att en tredjedel av dess periodtid är aktivt hög och två tredjedelar är aktivt låg. En cykeltid på 33 procent för en ATmega16:s PWM-pinne menas även att dess utspänning kommer att motsvara en tredjedel av 5 volt.

Formel för cykeltid är T_{on} / T_{period} där kvoten mellan T_{on} och T_{period} resulterar i en cykeltid i procent. I vissa lägen kan en PWM-signal utryckas i hertz. Frekvensen för en PWM-signal kan därmed beräknas med följande formel: $1 / T_{period.}$

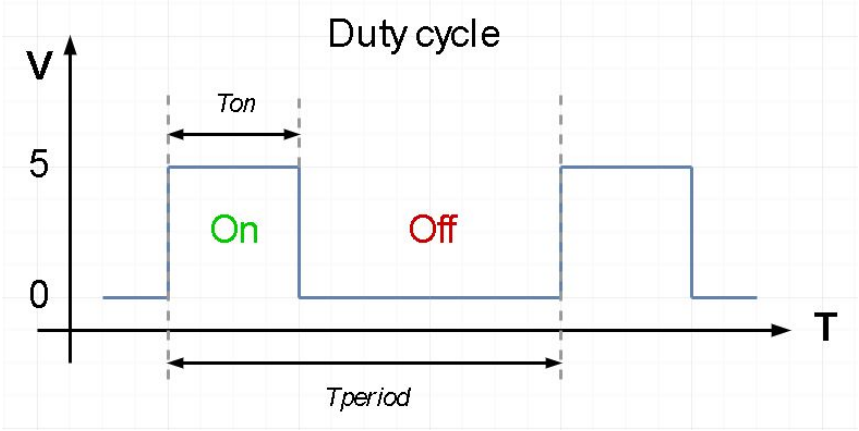


Bild 11: Cykeltid i en PWM-signal.

2.5 Styrkrets

Bild 12 visar uppkopplingen som användes av bilen för att kunna styra servomotorernas hastighet och riktning. I bilden är *Motor A* bilens frammotor och *Motor B* bilens bakmotor. Genom att ändra *Motor A* riktning kommer bilen att kunna svänga vänster och höger. Att ändra riktningen på *Motor B* kommer bilen att kunna åka framåt och bakåt.

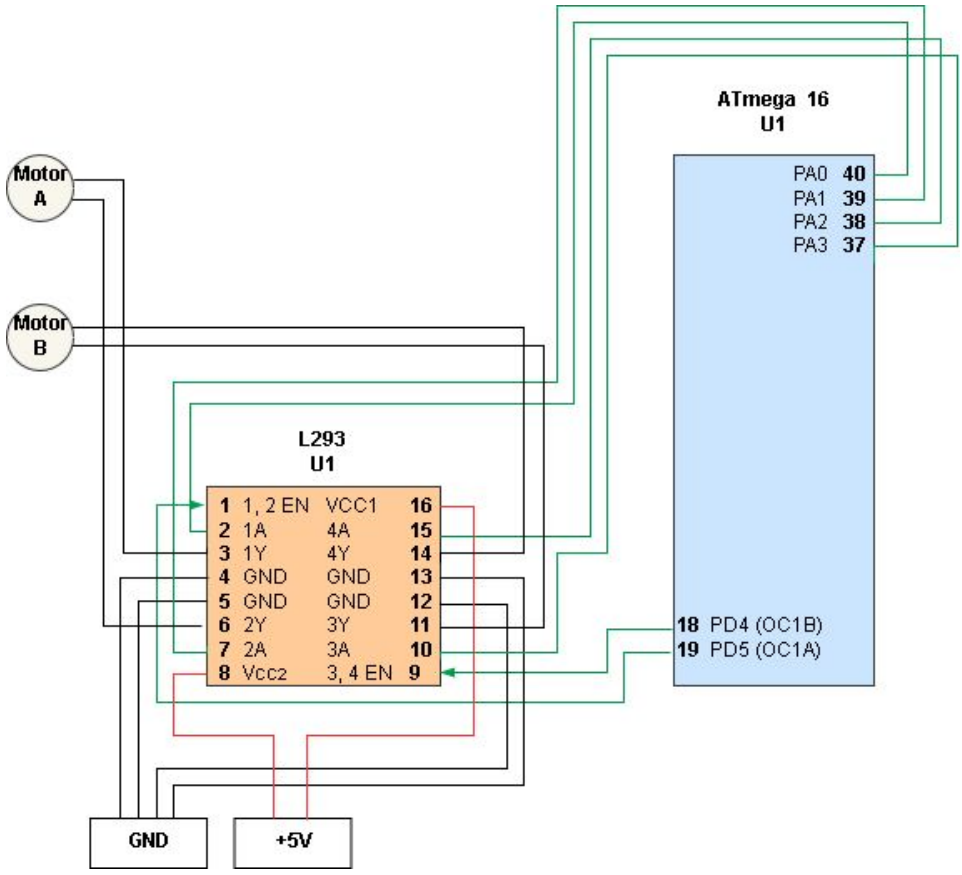


Bild 12: Kopplingsschema för L293 respektive till ATmega16 och servomotor. Kopplingspunkt med pil kopplas till ATmega16:s PWM-pinnar.

2.5.1 L293

Komponenten L293 användes med ATmega16 för att styra *Motor A* och *Motor B*. L293 har funktionalitet att tillåta enkelriktad eller dubbelriktad styrning av DC-motorer. Om enkelriktad styrning behövs kan L293 styra upp till fyra motorer. Medan om dubbelriktad styrning behövs kan L293 styra upp till två motorer. Under projektet användes L293 för dubbelriktad styrning. En annan funktionalitet som finns på L293 är spänningsreglering av dess drivpinnar. Denna funktion används för att styra servomotorernas hastighet. Se *Tabell 4* för att se L293:s drivpinnar.

Tabell 4: Pin-funktioner för L293.

Namn	**Pin**	**Beskrivning**
Vcc1	16	5 V
GND	4, 5, 12, 13	
Vcc2	8	4.5 V till 36 V
1,2 EN och 3,4 EN	1, 9	PWM-pinnar. 0 kHz till 5 kHz
1A, 2A, 3A, 4A	2, 7, 15, 10	Styrpinnar. Kopplas till ATmega16
1Y, 2Y, 3Y, 4Y	3, 6, 14, 11	Drivpinnar. Kopplas till DC-motor

2.5.2 Dubbel -och enkelriktad styrning

Bild 13 visar två sätt som DC-motorer kan kopplas till L293. Bildens vänstra sida visar en uppkoppling till en DC-motor som är kopplat till pinne 3 och pinne 6. Denna uppkopplings sätt tillåter en DC-motor att kunna rotera mot -och medurs, det vill säga en uppkoppling som tillåter dubbelriktad styrning av en DC-motor. När pinne 1 och 2 är aktivt hög kommer DC-motorn att rotera moturs. Medan när pinne 1 och 7 är aktivt hög kommer motorn att rotera medurs. Om pinne 1, 2 och 7 skulle vara aktivt höga samtidigt kommer DC-motorn att låsa sig och kommer inte att kunna rotera.

Bildens högra sida visar en uppkoppling sätt som tillåter enkelriktad styrning av två DC-motorer. För att få DC-motorerna att rotera medurs måste de kopplas enligt bilden. Däremot om moturs rotation önskas måste uppkopplingen på bilden spelas. När DC-motorerna ska rotera måste pinne 9 respektive pinne 11 och pinne 14 vara aktivt hög.

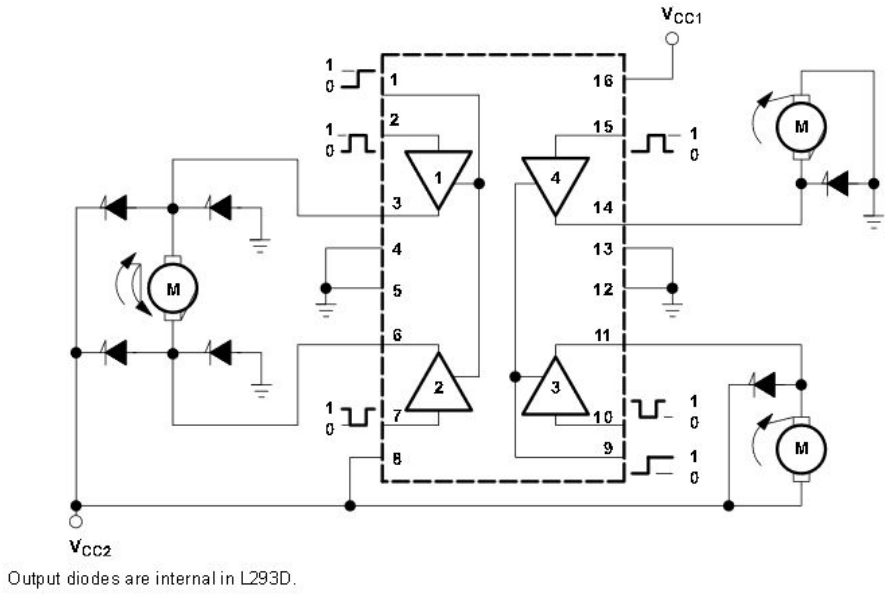


Bild 13: Funktionell block diagram på L293 (Texas Instrument).

2.5.3 Hastighet styrning

Pinne 1 och 9 på L293 har funktionen att bestämma utspänningen på dess drivpinnar. Genom att förse en av pinnarna med en spänning mellan 0 och 5 volt kommer vissa drivpinnar att få en viss utspänning. Exempelvis om pinne 1 är aktivt hög kommer endast drivpinnarna 3 och 6 att kunna få en viss utspänning. Om pinne 9 är aktivt hög kommer endast drivpinnarna 11 och 14 att kunna få en viss utspänning.

Utspänning på drivpinnarna är dock inte bara beroende på pinne 1 och 9 men även av pinne 8. Relationen mellan pinne 8 och pinne 1 och 9 kan beskrivas som en produkt där resultatet bestämmer drivpinnarnas utspänning. När pinne 1 och 9 kopplas till en PWM-signal blir drivpinnarnas utspänning beroende av PWM:ens cykeltid multiplicerad med pinne 8:s inspänning. Se *Bild 14.* Det vill säga att den procentuella cykeltiden bestämmer hur många procent av pinne 8:s inspänning ska släppas till drivpinnarna. Detta betyder även att drivpinnar inte kommer få en utspänning om pinne 1 och 9 är aktivt låg.

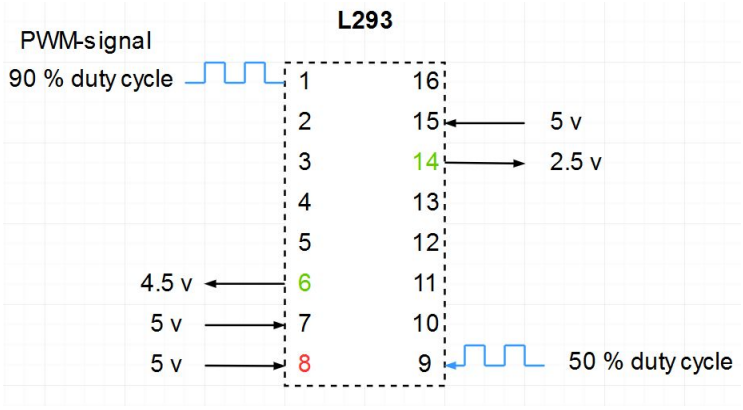


Bild 14: Bilden beskriver hur L293 kan reglerar utspänning på sina pinnar med hjälp av PWM.

2.6 Ultraljudssensor

Ultraljudssensorn är den komponent som används för att mäta ut avståndet mellan bilen och ett eventuellt föremål framför den.

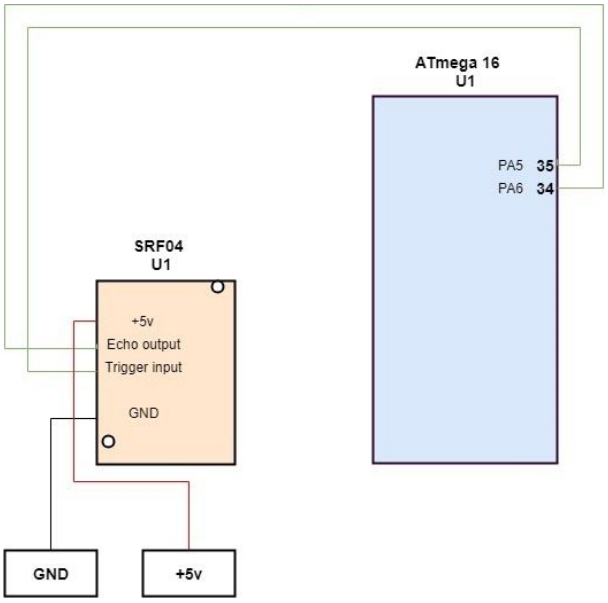


Bild 15: bild på kopplings schema mellan ultraljudssensorn SRF04.

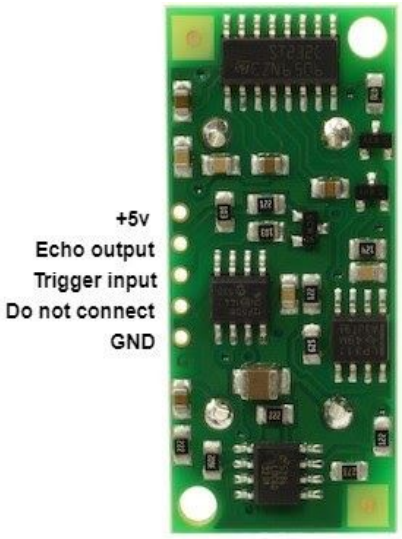


Bild 16: Bild på Ultraljudssensorn och ATmega 16.

Sensorn som används behöver en startpuls till sensorns Trigger-input-pinne på minst tio mikrosekunder. När den har fått in signalen blir den aktiverad. Därefter kommer det skickas ut en åtta-cykels signal av ultraljuden med en frekvens på 40 kHz. Sedan sätter sensorn Echo-output-pinnen till en logisk etta. Sensorn lyssnar sedan på ett eko från ultraljudet som skickades ut när trigger-input-pinnen får sin aktiveringssignal. Om sensorn inte tar upp ljudvågen kommer den sänka Echo output pinnen till en logisk nolla ändå efter cirka 36 millisekunder. När sensorn upptäcker ljudvågen sätts Echo output pinnen till en logisk nolla. När den logiska nollan satts behöver man vänta i minst tio mikrosekunder efter varje gång, för att kunna aktivera sensorn på nytt.

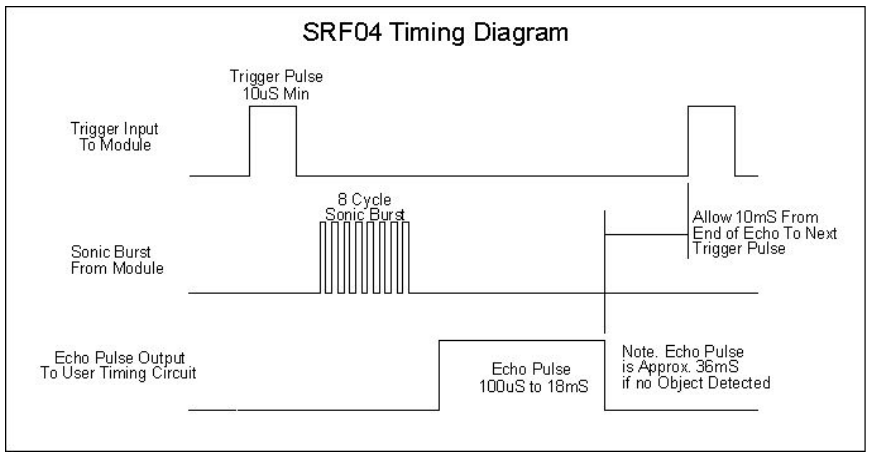


*Bild 17: Bild på SRF04 Tidsdiagram (*Robot Electronics*).*

Förklarar hur man startar ultraljudssensorn och hur det de olika pulser förhåller sig till varandra.

2.7 Joystick

Analoga PS2 Joysticks används för att kunna reglera hastighet och riktning på bilen. Det används två stycken joysticks ena för hastighet och andra för riktning. Joysticken är utrustad med två potentiometers och en knapp. De två potentiometrarna används för att kunna indikera vilket håll joysticken trycks åt i X- och i Y-led. Knappen är hög när den inte är intryckt och när knappen blir intryckt skickas det en låg eller en groundad signal på KEY-pinnen, men är inte användbar i fallet för att reglera hastighet och riktning.

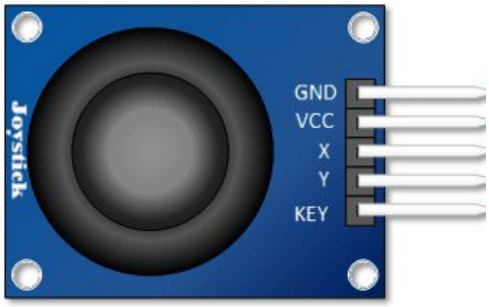


*Bild 18: Bild på Analog PS2 joystick (*Electro kit*).*

Tabell 5: Pin-funktioner till Analog PS2 Joystick.

Namn	**Pin**	**Beskrivning**
GND	1	0 V
VCC	2	5 V
X	3	Analog utsignal när X led inte är centrerad
Y	4	Analog utsignal när Y led inte är centrerad
KEY	5	Knappen skickar en låg signal när nedtryckt

3. Mjukvara

I den här delen beskrivs viktiga subrutiner och program som används av kontrollen och bilen. Bilens och kontrollens Kod kommer tas upp, men även LCD:n och sensorns kod då de är viktiga delar av huvudprogramen.

3.1 Motor

Bild 19 beskriver subrutinen som styr bilens servomotorer. Bilden beskriver en översikt av *motor_update* alla faser. Fasen *Get motor ID* bestämmer vilket servomotor som ska röras, fasen *Update speed* bestämmer servomotorns hastighet och fasen *Update direction* bestämmer servomotorns riktning.

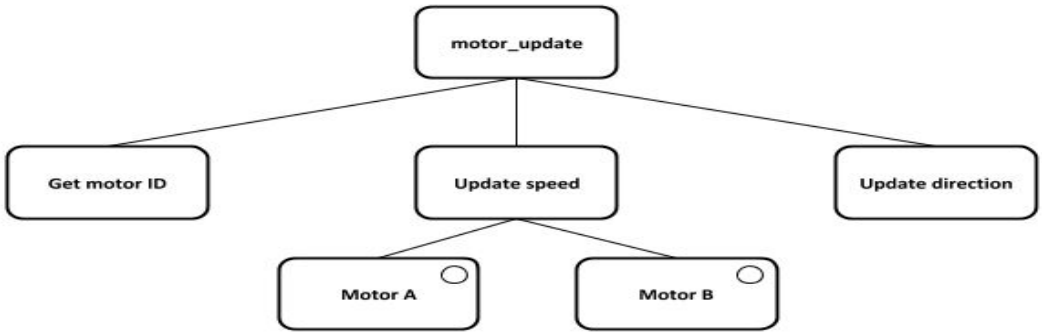


Bild 19: JSP-diagram för subrutin `motor_update`.

Programmet `motor_update` använder sig av tre argument för att uppdatera en motors hastighet och riktning. Dessa argument finns i SRAM och heter `MOTOR_ID`, `MOTOR_SPEED` och `MOTOR_STATE`, se *Bilaga C Kod C.1 Bilen*.

När `motor_update` kallas kommer rutinen att hämta `MOTOR_ID`. Här sker en kontroll om ett giltig motor-ID anges, om inte kommer rutinen att avbrytas omedelbart. Sedan hämtar rutinen `MOTOR_SPEED` som uppdaterar den specifika motorns hastighet. Här kontrolleras även inmatningsdatan.

Därefter läser rutinen hastighetsdatan och kontrollerar om det ligger i en tillåten intervall. Om hastighetsdatan skulle ligga utanför den tillåtna intervallen kommer den specifika motorns hastighet att sättas till noll. Kontroll av `motor_update` argumenten finns för att minimera användning av ATmega16:s resurser och motverka oönskade beteende vid signalfel. Under projektets gång har dessa beteende varit att motorerna rörde sig utan användarinput eller mindre reaktivt styrning av motorerna.

Slutligen hämtas `MOTOR_STATE` som utan kontroll uppdaterar motorns riktning. Varför tillståndsdatan inte kontrolleras beror på att motorernas rörelse är inte beroende av `MOTOR_STATE`.

3.2 Ultraljudssensor

Bild 20 beskriver subrutiner under `get_distance` som står för beräkningen av avstånd mellan ett objekt och bilen.

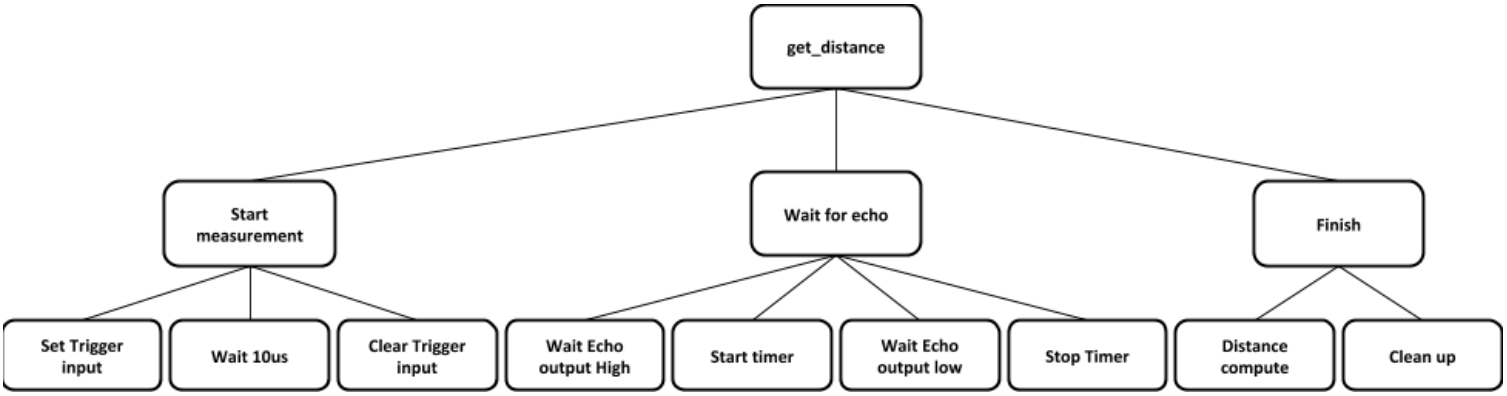


Bild 20: Ett JSP-diagram för kod till ultraljudssensorn SRF04. Där bilden visar hur avståndet mellan bilen och ett eventuellt föremål framför bilen räknas ut.

Programmet delas in i tre olika delar som syns i bilden *bild 20*, där `start measurement` tar hand om att aktivera sensorn. Där skickas en signal till sensorn i tio mikrosekunder för att aktivera den. När sensorn väl är aktiv, aktiveras en intern timer som sedan är igång tills Echo output pinnen går från en logisk etta till en logisk nolla. När den logiska nollan väl har upptäckts stoppas den interna timern. Efter den väl har stoppats skickas datan in i `distance_compute` för att räkna ut det eventuella avståndet till objektet. Efter `distance_compute` fått ut vilket avstånd det är till objektet skickas datan till kontrollern. Därefter rensas variablerna och gör det möjligt att köra den igen.

3.2 Bluetooth

I detta projekt finns det två separata implementationer av koden för bluetooth. Ena för kontrollen och den andra för bilen. De skiljer sig åt lite grann. Men huvudfunktionerna är i grunden detsamma. `TX_BUFFER` och `RX_BUFFER` är minnen i SRAM som används för att skicka respektive skicka data.

`Usart_transmit` är en subrutin som hanterar sändning av data. Funktionen kollar om UDRE (*Usart Data Register Empty*) flaggan i register UCSRA(*UART Control and Status Register A*) i processorn är satt och om registren är tom överförs 1 *byte* av data till UDR *(Usart Data Register)* registren för att sedan skickas via bluetooth. Detta fungerar väldigt bra för bilen eftersom sensordata får plats i 1 *byte*.

Från kontrollen ska det sändas 2 *bytes* av data, istället för att modifiera subrutinen `usart_transmit` lades det till en ny subrutin som heter `usart_buffer_transmit`. Funktionen itererar över `TX_BUFFER` och använder subrutinen `usart_transmit` för att sända data som finns i bufferten. Funktionen används för att överföra hela `TX_BUFFER` som håller 2 *byte* av data.

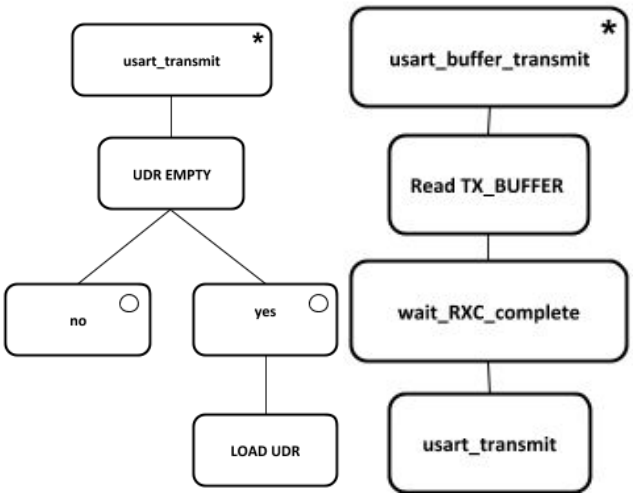


Bild 21: JSP-diagram för bluetooth-sändning

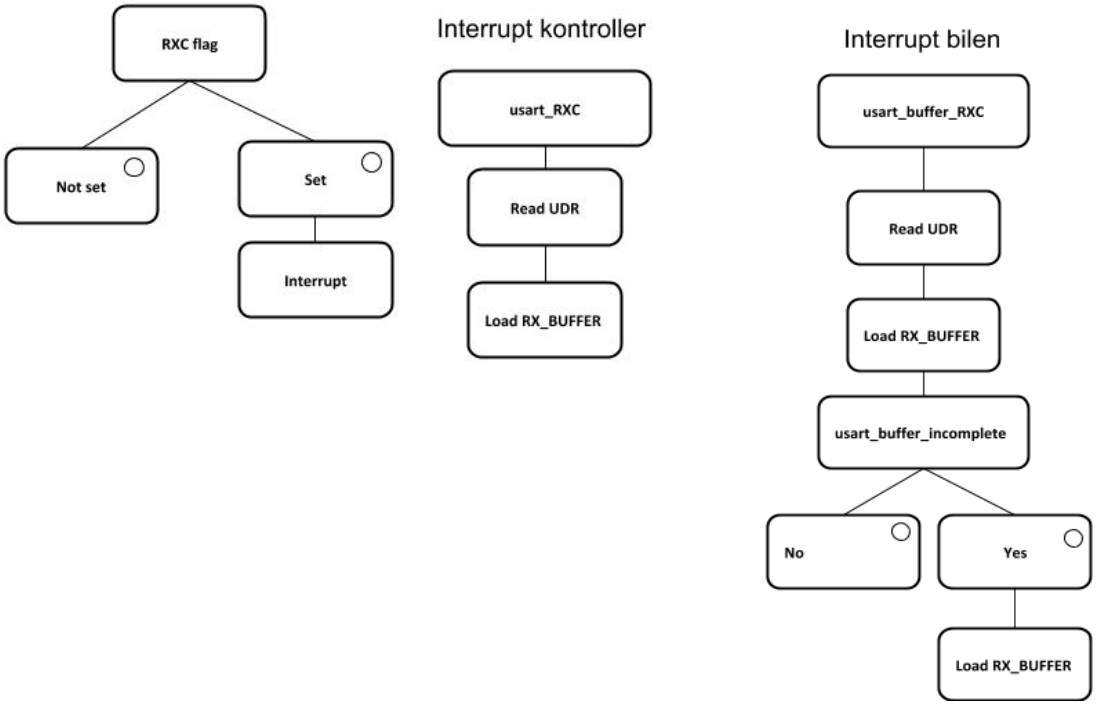


Bild 22: JSP-diagram för bluetooth-mottagning

När RXC (*Receive Complete*) flaggan i register UCSRA är satt körs en interrupt som hanterar inkommande dataströmmar. Interrupten lagrar data som tagits emot via bluetooth i `RX_BUFFER`. På kontrollen körs interrupten `Usart_RXC` som lagrar 1 byte av data. På bilen körs `Usart_buffer_RXC` som används för att lagra 2 bytes av data.

3.2 Kontrollen

I kontrollens huvudprogram finns två huvudfaser som heter *cold* och *start*. Fasen *cold* körs bara en enda gång medan fasen *start* kommer köras flera gånger. Fasen *start* är självaste kontrollprogrammet och kan ytterligare delas in i olika underfaser. Se *Bild 23.*

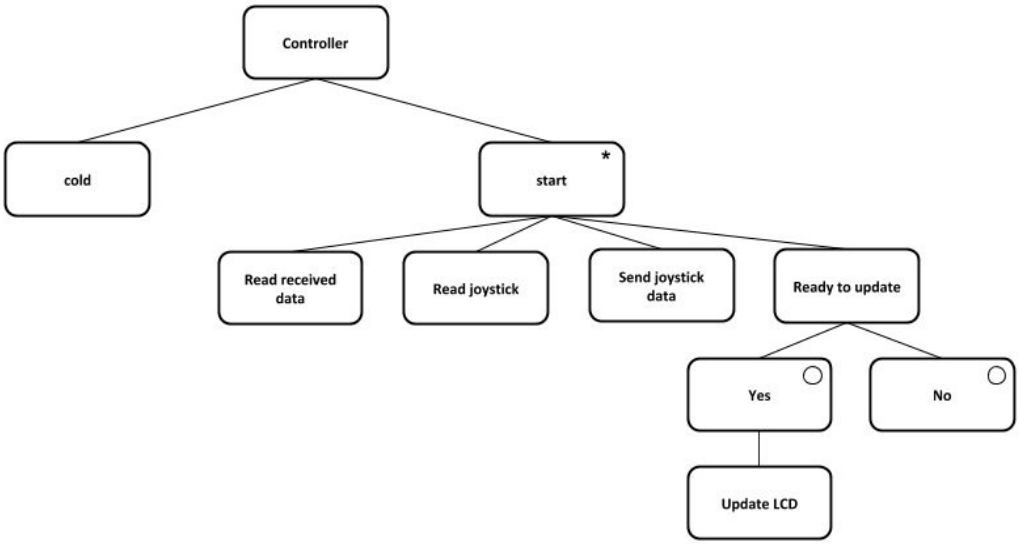


Bild 23: JSP-diagram för kontrollens huvudprogram.

Kontrollens huvudprogram börjar att utföra initialisering av alla hårdvarurutiner som finns i *cold*. Därefter kommer programmet att spendera resten av sin tid i *start*. I *start* körs subrutiner som hanterar användarinteraktion mellan kontrollen och bilen. För varje iteration börjar *start* med att läsa `RX_BUFFER_BEGIN` adressen i SRAM:et, denna adress innehåller avstånd datan som bilen har skickat. Sedan kommer avstånd datan att flytta `RECIEVED_DATA` adressen för att kunna uppdatera avståndet som visas på kontrollens display. Nästa fas i *start* är inläsning av kontrollens joysticks positioner. Inläsning av joystickarna kan delas i tre steg:

Första steget kommer *start* att filtrera analog signaler från joystickarna. När *start* läser insignaler från joystickarna startas en A/D-omvandling som returneras ett värde mellan 0 till 255. Beroende på vilken värde som returneras kan joysticken riktning bestämmas. Efteråt filtreras värden mellan 128 och 140 till noll. För värdena utanför 128 och 140 filtreras de till värden mellan 1 och 128 i stigande ordning.

Andra steget kommer *start* att kalla subrutinen *encode.* De *encode* gör är att den formaterar de filtrerade returvärdet till de format som används av bluetooth-koden.

Tredje steget kommer *start* att skriva joysticken returvärden till adresserna `JOYSTICK_X_DATA` och `JOYSTICK_Y_DATA`, adresserna används för att kunna uppdatera joystickarnas position på displayen. Därefter kommer *start* att skriva de formaterade joystick-data till `TX_BUFFER_BEGIN` och `TX_BUFFER_END` och kalla `usart_buffer_transmit` för att skicka joysticks positionerna till bilen. *Start* kommer att sedan repetera inläsning av avståndsdata, samt sändning av joystick-data i 256 repetitioner innan displayen kommer att uppdateras.

3.3 Bilen

Bilens huvudfaser fungerar som kontrollens huvudprogram. Fasen *cold* körs en enda gång och initierar hårdvarurutiner som används av bilen. Fasen *start* är bilprogrammet och kommer köras flera gånger. I fasen *start* finns även underfaser som körs av *start* tills den termineras. Se *Bild 24.*

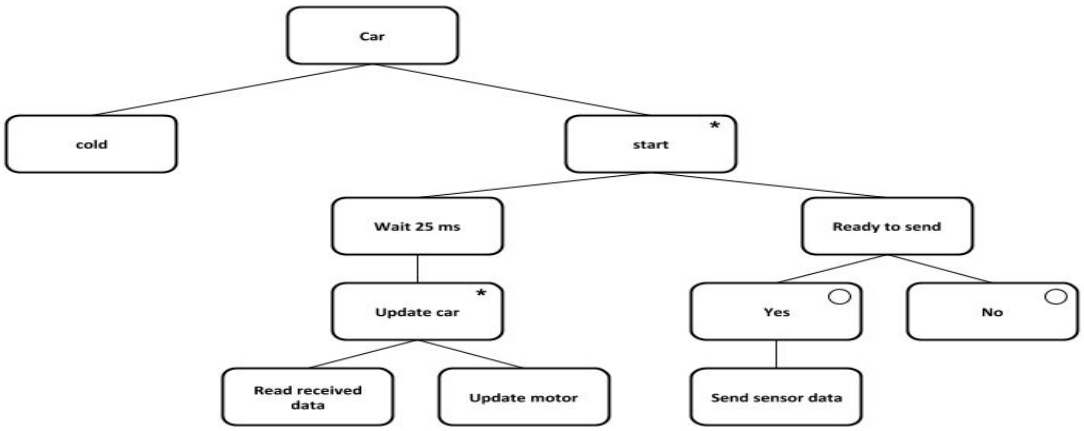


Bild 24: JSP-diagram för bilens huvudprogram.

Lik kontrollens *start* så tar bilens *start* hand om användarinteraktion mellan kontrollen och bilen. Det som urskiljer bilens huvudprogram med kontrollens huvudprogram är att bilens kod är anpassad för kontrollens kod. Detta på grund av att kontrollens kod måste bilen hinna att hantera 512 *byte* av inkommande data under en väldigt kort tid.

För att hantera de inkommande data delas bilens *start* i två faser. Första fasen är datahanterings fasen och det andra är datasändningsfasen. Under bil programmets livstid kommer majoritet av bilens resurser att gå till datahantering fasen, då den prioriterar hantering av data som sändes från kontrollen.

I datahanterings fasen hämtas `RX_BUFFER_BEGIN` och `RX_BUFFER_END` i SRAM:et, dessa adresser innehåller joysticks data som kontrollen har skickat till bilen. Sedan kallas subrutinen `status_update`. När `status_update` kallas kommer den att läsa in en joystick data och avkoda den med hjälp av *decode*. Efter avkodningen finns motor-ID, motorhastighet och motortillståndsdatan besparade i tre olika register. Dessa värden skrivs sedan till adresserna `MOTOR_ID`, `MOTOR_SPEED` och `MOTOR_STATE` för där sedan kalla `motor_update`.

Kallelsen av `status_update` upprepas två gånger, en gång för varje joystick data. Hela datahanterings fasen upprepas sedan under 25 millisekunder innan *start* går vidare. När 25 millisekunder har passerat kommer *start* att ytterligare repeterar fasen i 160 repetitioner. Totalt tar det 4 sekunder innan *start* hamnar i datasändnings fasen. I denna fas kallas `get_distance`, de returvärdet från `get_distance` skrivs sedan till `TX_BUFFER_BEGIN` adressen. Efter skrivningen kallar datasändningsfasen `usart_transmit` som skickar avståndsdatan till kontrollen.

3.4 LCD

Displayen består av en del olika ingångar, RS avgör om man ska skicka instruktioner eller data och R/W avgör om det ska läsas av displayen eller skickas till displayen. De åtta databitar på LCD:n används för att välja instruktion och skicka in våra databitar. LCD:n kräver att man ger den signaler i en viss ordning för att kunna skriva till den, lite som ett kodlås. Alltså varje gång man ska göra något med displayen måste man göra detta. Detta gäller för både instruktionen och skrivningen. Funktion fungerar på detta sätt:

1. Sätt RS och R/W. Beroende på situation ändras RS, R/W kommer förbli låg då projektet aldrig kräver att läsning används
2. Skicka in data-bitarna
3. Delay
4. Sätt Enable till ett
5. Delay
6. Sätt Enable till noll (här tar displayen in data-bitarna)

Varje gång något ska skrivas till displayen måste denna funktion användas för att få in vår data till displayen.

Under setup delen initieras alla delar i displayen för att möjliggöra skrivning och liknande till displayen. Man börjar med att sätta RS till noll då en instruktion ska skickas och R/W till noll då instruktionen ska skickas. Sedan under skrivdelen skickas vår data in. Nu kommer data skickas och då måste RS sättas till ett. R/W kommer här förbli noll då det skickas data. För att välja vilken instruktion man ska använda sätter man data-bitarna till olika kombinationer. displayen avgör vilken instruktion det är beroende på den vilken den största biten satt till ett. I setup-delen används dessa instruktioner:

1. `Function_set`: Här ställs displayen in på two-line mode, 5x11 mode och 8 bit mode vilket ställer in hur displayen kommer skicka och hur den kommer användas.
2. `Display_on`: Här sätts displayen på, positions-pekaren och pekarens blinkande
3. `Display_clear`: Här rensas displayen och återställs
4. `Entry_mode_set`: Här bestämmer man hur pekaren kommer funka

`Function_set`, `display_on` och `entry_mode_set` är instruktioner som måste göras för att displayen ska fungera. Dessa behöver bara göras en gång och därför lades de in under init-delen. `Display_clear` används vid initiering för att tömma portarna att det inte blir en krasch i programmet om något skulle ligga kvar från innan displayen startats om.

Sedan i skrivdelen ser det ut såhär:

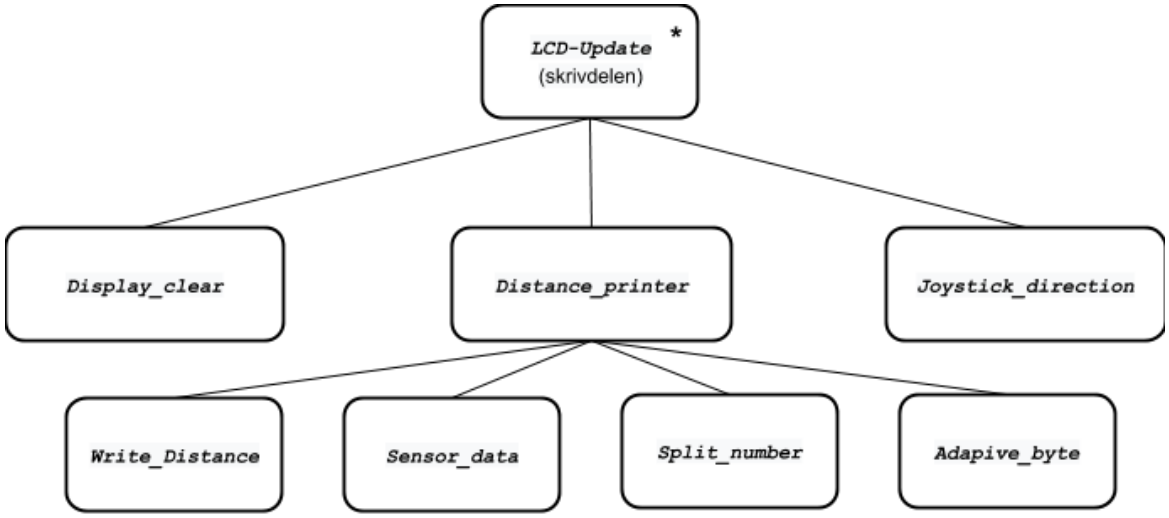


Bild 25: JSP-diagram på hur skrivdelen är upplagd.

Och nedan förklaras funktionernas användning, de ser ut såhär:

1. `Display_clear`: Eftersom skriv-funktionen kommer loopas kommer det som stod innan behöva tas bort, annars kommer det skrivas på rad.

2. `Distance_printer`: Detta är funktionen som kommer skriva på första raden av vår display, den består av fyra st separata funktioner:
   - `Write_distance`, som skriver ut ordet “Distance:”
   - `Senstor_data`, tar in ett tal och delar upp det i ental, tiotal och hundratal denna baseras på en funktion som heter `split_number` som delat upp ett tal till delar och är en generell funktion som funkar för alla tal.
   - `Adaptive_byte` som avgör vilka tal som ska skrivas ut på skärmen, den använder sig av uppdelning från `sensor_data` för att avgöra vilka tal som ska filtreras bort, t.ex. om man har talet 90 vill man inte skriva 090 istället delar man upp talet och kollar vilken första siffran är. Denna funktionen printar även ut på displayen.

3. `Joystick_direction`, fungerar väldigt likt `distance_printer` på så sätt att man här också tar in ett tal delar upp det filtrerar ut onödiga siffror. Eftersom detta görs på rad två måste det bestämmas att man ska börja med att skriva på rad två, detta görs med hjälp av instruktionen set DDRAM address. Sedan skrivs det ut “X:”, efter detta delar sedan upp dess värde och filtrerar bort det onödiga som vanligt och skriver ut. Innan Y skrivs ut sätts en ny plats att skriva på, detta gör att Y inte flyttar sig beroende på hur många siffror som blir av X. Även denna gång med samma instruktion som innan. Sedan görs sedan samma sak med Y. Det delas upp, filtreras bort och skrivs ut.

Våra andra viktiga funktioner är enable, *write_to_lcd* och *set_enable*, dessa funktioner är de som hanterar separata skrivningar till displayen. *Write_to_lcd* skriver data till displayen, medans *setup_enable* används när instruktioner ska skrivas till displayen. Dessa funktioner sätter vilka RS/RW som ska användas och sätter datan som ska skickas in. Sedan kallas vår enable-funktion som sköter alla skrivning, den sätter enable till ett sedan gör den till noll. När enable bli noll tar displayen in vår data. LCD:n är alltså väldigt komplicerad och invecklad, men så länge man gör alla initieringar och man skriver allting i ordning har man kommit långt.

4 Tillvägagångssätt och planering

Tillvägagångssättet under projektets gång kan delas upp i tre delar:

- Förberedelser och planering
- Sammankoppling och programmering
- Färdigställning

De första veckorna var förberedelser och planering av arbetet. Under denna tid diskuterades det om hur projektet skulle se ut, det bestämdes vilka projektkrav och börkrav som skulle ställas och vilka funktioner som skulle implementeras. Efter att projektkraven var upplagda valdes det lämpliga komponenter för att möjliggöra implementationen av de olika funktionerna.

För att underlätta arbetet tilldelades gruppmedlemmarna med vardera ansvarsområden, d.v.s. ansvar för komponenter till vissa funktioner från projektkraven. Därefter lästes datablad för respektive komponent för att förstå hur de olika komponenterna fungerar och för att kunna skapa kopplingsscheman. Efter detta sattes det upp en planering för hur arbetet skulle genomföras.

När all planering och förberedelser var färdiga påbörjades sammankoppling av hårdvaran utefter kopplingsscheman. När alla komponenter var ihopkopplade började arbetet på mjukvaran.

Mjukvaran för komponenterna utvecklades separata från varandra för att underlätta felsökning. Under denna del av projektet användes olika hjälpmedel för att kontrollera komponenternas funktionalitet. Logikanalysatorn är ett exempel på ett sådant hjälpmedel. Logikanalysatorn kan visa binära signaler från individuella pinnar på en komponent som höga eller låga signaler på en graf. Den användes flitigt för att bevaka logiska signaler och felsöka alla komponenter.

Den sista delen i projektet var färdigställningen. Efter att alla komponenters funktionalitet bekräftats och mjukvaran optimerats slogs mjukvaran för komponenterna ihop för att bilda den slutgiltiga produkten. Detta krävde ytterligare optimeringar i både mjukvaran och hårdvaran för att bekräfta korrekt samverkan mellan de olika delarna.

4.1 Planeringsschema

Här är planeringsschemat för projektet som visar vad som gjorts under veckorna. Scheman visar händelser från 2019-01-28 till 2019-03-22.

Tabell 6: Planeringsschema.

Vecka	**Händelse**
5	Projektstart Kravspecifikationen, komponentlistan och tidsplanet skrevs Tilldelade arbetsuppgifter Läste datablad för alla komponenter
6	Löste ut komponenter ATmega16 kopplades och kunde programmeras med JTAG L293 kopplades LCD kopplades
7	Ultraljudssensorn kopplades Bluetooth kopplades Motorkod skrevs och färdigt testat LCD-kod skrevs och färdigt testat Ultraljud-kod skrevs Bluetooth-kod skrevs
8	Joysticks kopplades Joystick-kod skrevs
9	Kontrollen byggdes Bilen byggdes Ultraljud-kod färdigt testat Joystick-kod färdigt testat
10	LCD-kod förbättrades och kunde visa joysticks positioner
11	Bluetooth-kod färdigt testat Sammansatte hårdvarukod
12	Buggfixade kod Huvudkod för bilen och kontrollen färdigställdes Test körde bilen Projektslut

6. Resultat

Alla funktioner och krav som lades upp under projektstart har färdigställts förutom ett av börkraven som var att kontrollen skulle kunna spela ljud. Den kunde inte implementeras på grund av tidsbrist. Slutliga produkten hade alla andra funktioner som krävdes i projektkraven och fungerade helt. Det vill säga att alla delar hade kopplats ihop till en produkt och bilen blev styrd av två joysticks, processorerna kommunicerade åt båda hållen via bluetooth kommunikation, sensorn fungerade korrekt och kunde varna om hinder samt att kontrollen hade en LCD.

Bilen och kontrollen var portabla. För att försörja alla komponenter med ström användes 3 stycken 1.5 V batterier för bilen respektive kontrollen. Bilen kunde varna om hinder på upp till 1 m avstånd. Användaren kunde bestämma hastigheten genom att kontrollera X och Y på LCD:n.

7. Diskussion

Till en början kan man helt klart säga att vi har lärt oss en hel del under projektet och man kan nu ta med sig mycket av det som man gjort under projektets gång. Vi har fördjupat oss inom ämnet och har nu lärt oss mycket mer om det vi jobbade med. Men vi har även lärt oss mer om assembler och processorn vi jobbade med.

När vi började detta projekt visste vi att det var en gammal radiostyrd bil som vi skulle använda oss av för att få tekniken att fungera, den styrs av motorer som går på en spänning som vi inte var säkra på vad det var. Detta märkte man då när vi satte i våra batterier gick den fruktansvärt långsamt. Detta mycket på grund av att batterierna väger mycket mer än resten av bilen totalt och på grund av det saktar ned den. Men vi misstänker även att motorerna kan ha varit gjorda för högre spänning än det batterierna hade och att det var därför den inte körde fort.

Med den planering som vi satte upp skulle vi varit klara efter tre veckor. Vi lyckades nästan detta eftersom vi började ganska tidigt. Men vi tänkte inte på att problem skulle komma upp och att sådana saker tar tid. Vi bytte under projektets gång bluetooth-modemen tre gånger och klockorna två gånger. Detta ledde till att vi var tvungna att ändra all kod till den nya klockfrekvensen.

När vi sedan var klara fick vi en del oväntade problem som kom av ihopsättning av de olika kod delarna och kommunikationen. Det som hände var att när vi skulle skicka sensor-datan var den prioriterad över annat. Detta gjorde att de annat kod inte körde lika ofta och då fungerade inget. Det enda som kontrollen hade tid för var att ta in sensorns data. Detta ledde till att vi var tvungna att dra ner på hur ofta den skulle skickas. Det gjorde dock att sensordatan inte var ändrad lika ofta och kunde göra att man inte hann att reagera om det kom en vägg.

Referenser

Atmel (Rev. 2466T–AVR–07/10). *8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash ATmega16 ATmega16L*[bild, www] <http://ww1.microchip.com/downloads/en/DeviceDoc/doc2466.pdf#G1.1058286> Hämtad 2019-02-04.

Electro kit (2019). *Analog Joystick Module*[bild, www] <https://www.electrokit.com/uploads/productfile/41015/41015703_-_Analog_Joystick.pdf> Hämtad 2019-04-05.

Robot Electronics (2003). *SRF04 - Ultra-Sonic Ranger Technical Specification*[bild, www] <https://www.robot-electronics.co.uk/htm/srf04tech.htm> Hämtad 2019-03-25.

Texas Instrument (2016). *L293x Quadruple Half-H Drivers*[bild, www] <http://www.ti.com/lit/ds/symlink/l293.pdf> Hämtad 2019-02-04.

VanHeden(2013-03-05). *Kom igång med JTAG ICE 3 för AVR*[www] <https://docs.isy.liu.se/bin/view/VanHeden/AVRJTAG3> Hämtad 2019-02-04.

VanHeden (2019-02-08). *SRF04 Ultrasonic Range Finder 30th September 2003*[www] <https://docs.isy.liu.se/pub/VanHeden/DataSheets/srf04.pdf> Hämtad 2019-02-04.

VanHeden (2019-02-08). *SPECIFICATION Character Type Dot Matrix LCD Module JM162A*[www] <https://docs.isy.liu.se/pub/VanHeden/DataSheets/jm162a.pdf> Hämtad 2019-02-04.

Wikipedia (6 May 2019, at 18:19 (UTC)). *Frekvens*[www] <https://en.wikipedia.org/wiki/Frequency> Hämtad 2019-05-25.

Wikipedia (26 November 2018, at 17:12 (UTC)). *Duty cycle*[www] <https://en.wikipedia.org/wiki/Duty_cycle> Hämtad 2019-05-25.

Wikipedia (10 May 2019, at 15:23 (UTC)). *Pulse-width modulation*[www] <https://en.wikipedia.org/wiki/Pulse-width_modulation> Hämtad 2019-05-25.

Bilagor

Här hittar komponentlista och kod för projektets delar.

Bilaga A Komponentlista

I denna del har vi en lista på de komponenter som användes på bilen och kontrollen.

Tabell 7. Komponentlista.

Komponent	**Antal**
ATmegaL16	2
JTAG ICE 3	2
L293	1
Servomotor	2
LCD JM162A	1
Potentiometer	1
SRF04 (Ultraljudssensor)	1
Analog PS2 Joystick	2
EXO3 7E1C (Kristall klocka 14.75MHz)	2
BlueSMIRF Gold (Bluetooth-modem)	2

Bilaga B Kod

Här är koden som användes till projektet. Koden är skriven i AVR assembler. Under B.1 Bilen finns koden för hur bilens komponenter skulle agera.I B.2 Kontrollen finns de koden för komponenterna på kontrollen.

B.1 Bilen

```
;
; car_main_v3.asm
;
; Created: 2019-03-12 20:06:02
;

; ---------------------------------------
; --- INC START: usart.inc
; ---------------------------------------

      .equ         UBRR_VALUE = 7
      .equ         BUFFER_SIZE = 2

; ---------------------------------------
; --- INC END: usart.inc
; ---------------------------------------

; ---------------------------------------
; --- INC START: motor.inc
; ---------------------------------------

      ; Instruction
      .equ         MOTUR = $02
      .equ         MEDUR = $01
      .equ         FAST_STOP = $03
      .equ         FREE_STOP = $00
      ; PWM output frequency
      .equ         ICR1_TOP = 64
      .equ         TOV1_PERIOD = 1039
      ; Motor ID
      .equ         MIDA = 0
      .equ         MIDB = 1

; ---------------------------------------
; --- INC END: motor.inc
; ---------------------------------------
```

```
; ---------------------------------------
; --- INC START: ultrasound.inc
; ---------------------------------------

      .equ         DIST_COUNT_TOP = 255
      .equ         OCR0_TOP = 23

; ---------------------------------------
; --- INC END: ultrasound.inc
; ---------------------------------------

      .org   0
      jmp    cold
      .org   URXCaddr
      jmp    usart_buffer_RXC
      .org   OC0addr
      jmp    timer0_OCF

      .dseg
      .org   $60

; ---------------------------------------
; --- SRAM START: usart.asm
; ---------------------------------------

      TX_BUFFER_POS:
            .byte  1
      TX_BUFFER_BEGIN:
            .byte  1
      TX_BUFFER_END:
            .byte  1
      ;
      RX_BUFFER_POS:
            .byte  1
      RX_BUFFER_BEGIN:
            .byte  1
      RX_BUFFER_END:
            .byte  1

; ---------------------------------------
; --- SRAM END: usart.asm
; ---------------------------------------

; ---------------------------------------
; --- SRAM START: motor.asm
; ---------------------------------------

      MOTOR_ID:
```

```
		.byte  1
	MOTOR_SPEED:
		.byte  1
	MOTOR_STATE:
		.byte  1

; ---------------------------------------
; --- SRAM END: motor.asm
; ---------------------------------------

; ---------------------------------------
; --- SRAM START: ultrasound.asm
; ---------------------------------------

	DIST_COUNT:
		.byte  1

; ---------------------------------------
; --- SRAM END: ultrasound.asm
; ---------------------------------------

	.cseg

; ---------------------------------------
; --- START: car_main.asm
; ---------------------------------------

cold:
	ldi    r16, LOW(RAMEND)
	out    SPL, r16
	ldi    r16, HIGH(RAMEND)
	out    SPH, r16
	call   usart_hw_init
	call   motor_hw_init
	call   ultrasound_hw_init
	sei

start:
	; Read the buffer transmited from controller
	ldi    YL, LOW(RX_BUFFER_BEGIN)
	ldi    YH, HIGH(RX_BUFFER_BEGIN)
	ld     r16, Y+
	push   r16
	ld     r16, Y
	call   status_update
	pop    r16
	call   status_update
	; Has 25 ms passed?
```

```
	ldi	YL, LOW(DIST_COUNT)
	ldi	YH, HIGH(DIST_COUNT)
	ld	r16, Y
	cpi	r16, DIST_COUNT_TOP
	brne	start
	; Remove bufferoverflow
	inc	r19
	cpi	r19, $A0
	brne	start
	clr	r19
	; Transmit distance data
	call	get_distance
	call	usart_transmit
	jmp	start

; ---------------------------------------
; --- status_update: Update motors on the car.
; --- Argument: r16 (encoded data)
; --- Return: None
; --- Uses: r16, r17, r18, Y, Z
; --- Usage:
; --- ldi	YL, LOW(RX_BUFFER_BEGIN)
; --- ldi	YH, HIGH(RX_BUFFER_BEGIN)
; --- ld	r16, Y
; --- call	status_update
; ---------------------------------------
status_update:
	call	decode
	ldi	YL, LOW(MOTOR_ID)
	ldi	YH, HIGH(MOTOR_ID)
	st	Y+, r16
	st	Y+, r17
	st	Y, r18
	call	motor_update
	ret

; ---------------------------------------
; --- decode: Decode a byte.
; --- Argument: r16 (encoded data)
; --- Return: r16 (motor id), r17 (motor speed), r18 (motor state)
; --- Uses: r16, r17, r18
; --- Usage:
; --- ldi	YL, LOW(RX_BUFFER_BEGIN)
; --- ldi	YH, HIGH(RX_BUFFER_BEGIN)
; --- ld	r16, Y
; --- call	decode
; ---------------------------------------
decode:
```

```
       mov    r17, r16
       mov    r18, r16
       andi   r16, 0b00000011 ; id
       andi   r17, 0b11110000 ; speed
       swap   r17
       andi   r18, 0b00001100 ; state
       lsr    r18
       lsr    r18
       ret

; ---------------------------------------
; --- END: car_main.asm
; ---------------------------------------

; ---------------------------------------
; --- START: usart.asm
; ---------------------------------------

; ---------------------------------------
; --- usart_transmit: Transmit a byte with USART.
; --- Argument: r16 (data)
; --- Return: None
; --- Uses: r16
; --- Usage:
; --- ldi    r16, $F0
; --- call    usart_transmit
; ---------------------------------------
usart_transmit:
    sbis    UCSRA, UDRE
    rjmp    usart_transmit
    out       UDR, r16
    ret

; ---------------------------------------
; --- usart_buffer_RXC: An interrupt that handle incoming datastream from the
controller.
; --- Argument: None
; --- Return: None
; --- Uses: r16, Y
; ---------------------------------------
usart_buffer_RXC:
    push    r16
    in        r16, SREG
    push    r16
    push    YL
    push    YH
    ;
    ldi       YL, LOW(RX_BUFFER_POS)
```

```
    ldi       YH, HIGH(RX_BUFFER_POS)
    ld        r16, Y
    inc       r16
    cpi       r16, BUFFER_SIZE + 1
    brlo    usart_buffer_incomplete
    ldi       r16, 1
    st        Y, r16
usart_buffer_incomplete:
    st        Y, r16    ; Update buffer pos
    add       YL, r16
    clr       r16
    adc       YH, r16
    in        r16, UDR
    st        Y, r16 ; Store value in right position
    ;
    pop       YH
    pop       YL
    pop       r16
    out       SREG, r16
    pop       r16
    reti

; ---------------------------------------
; --- usart_hw_init: Init. resources used by USART.
; --- Argument: None
; --- Return: None
; --- Uses: r16, Y
; --- Usage:
; --- call    usart_hw_init
; ---------------------------------------
usart_hw_init:
    ; Init. port
    in        r16, DDRD
    ldi       r16, (1 << DDD1)
    out       DDRD, r16
    ; Init. USART
    ldi       r16, HIGH(UBRR_VALUE)
    out       UBRRH, r16
    ldi       r16, LOW(UBRR_VALUE)
    out       UBRRL, r16
    ;
    ldi       r16, (1 << RXCIE) | (1 << RXEN) | (1 << TXEN)
    out       UCSRB, r16
    ;
    ldi       r16, (1 << URSEL) | (0 << UMSEL) | (1 << USBS) | (1 << UCSZ1) | (1 <<
UCSZ0)
    out       UCSRC, r16
    ;
```

```
    clr       r16
    ldi       YL, LOW(TX_BUFFER_POS)
    ldi       YH, HIGH(TX_BUFFER_POS)
    st        Y, r16
    ldi       YL, LOW(RX_BUFFER_POS)
    ldi       YH, HIGH(RX_BUFFER_POS)
    st        Y, r16
    ret

; ---------------------------------------
; --- END: usart.asm
; ---------------------------------------

; ---------------------------------------
; --- START: motor.asm
; ---------------------------------------

; ---------------------------------------
; --- motor_update: Move motor A and B using motor id, motor state and motor speed in
SRAM.
; --- Argument: None
; --- Return: None
; --- Uses: Y, Z
; --- Usage:
; --- call motor_update
; ---------------------------------------
motor_update:
       ldi    ZH, HIGH(MOTOR_ID)
       ldi    ZL, LOW(MOTOR_ID)
       ld     YL, Z+ ; Read motor_id
       ld     YH, Z+ ; Read motor_speed
       ;
       cpi     YL, MIDA
       brne   PC + 2
       jmp    case_a
       ;
       cpi      YL, MIDB
       brne   PC + 2
       jmp    case_b
       jmp    update_done
case_a:
       call   cycle_compute
       cli
       out    OCR1AH, YH
       out    OCR1AL, YL
       sei
       call   wait_next_tov1
       ; Clear prev. state
```

```
	cbi	PORTA, 0
	cbi	PORTA, 1
	; Write state
	ld	YL, Z ; Read motor_state
	jmp	update_state
case_b:
	call	cycle_compute
	cli
	out	OCR1BH, YH
	out	OCR1BL, YL
	sei
	call	wait_next_tov1
	; Clear prev. state
	cbi	PORTA, 2
	cbi	PORTA, 3
	; Write state
	ld	YL, Z ; Read motor_state
	lsl	YL
	lsl	YL
update_state:
	in	YH, PORTA
	or	YL, YH
	out	PORTA, YL
update_done:
	ret

; ---------------------------------------
; --- cycle_compute: Return a duty cycle constant.
; --- Argument: YL (motor_speed)
; --- Return: Y (duty cycle constant)
; --- Uses: Y, Z
; --- Usage:
; --- in	ZL, LOW(MOTOR_SPEED)
; --- in	ZH, HIGH(MOTOR_SPEED)
; --- ld	YL, Z
; --- call	cycle_compute
; --- mov	r16, YL
; --- mov	r17, YH
; ---------------------------------------
cycle_compute:
	push	ZH
	push	ZL
	; Check valid speed
	cpi	YH, 16
	brlo	cycle_in_range
	ldi	YH, 0
cycle_in_range:
	ldi	ZL, LOW(DUTY_CYCLE*2)
```

```
	ldi	ZH, HIGH(DUTY_CYCLE*2)
	add	ZL, YH
	clr	YH
	adc	ZH, YH
	lpm	YL, Z
	;
	pop	ZL
	pop	ZH
	ret

; ---------------------------------------
; --- wait_next_tov1: Ensure that one TOV1 period has occurred.
; --- Argument: None
; --- Return: None
; --- Uses: Y
; --- Usage:
; --- call   wait_next_tov1
; ---------------------------------------
wait_next_tov1:
	ldi	YH, HIGH(TOV1_PERIOD)
	ldi	YL, LOW(TOV1_PERIOD)
wait_next0:
	sbiw	Y, 1
	brne	wait_next0
	ret

; ---------------------------------------
; --- pwm_start: Start timer1 in fast pwm mode.
; --- Argument:    None
; --- Return: None
; --- Uses: Y
; --- Usage:
; --- call   pwm_start
; ---------------------------------------
pwm_start:
	in	YL, TCCR1B
	ori	YL, (0 << ICNC1) | (0 << ICES1) | (0 << 5) | (1 << WGM13) | (1 << WGM12)
| (0 << CS12) | (1 << CS11) | (1 << CS10)
	out	TCCR1B, YL
	;
	in	YL, TCCR1A
	ori	YL, (1 << COM1A1) | (1 << COM1A0) | (1 << COM1B1) | (1 << COM1B0)  | (0
<< FOC1A) | (0 << FOC1B) | (1 << WGM11) | (0 << WGM10)
	out	TCCR1A, YL
	ret

; ---------------------------------------
; --- motor_hw_init: Init. resources used by motor.
```

```
; --- Argument:     None
; --- Return: None
; --- Uses: r16
; --- Usage:
; --- call   motor_hw_init
; ---------------------------------------
motor_hw_init:
       ; Init. motor_state port
       in     r16, DDRA
       ori    r16, $0F
       out    DDRA, r16
       ; Init. PWM port
       in     r16, DDRD
       ori    r16, (1 << PORTD4)|(1 << PORTD5)
       out    DDRD, r16
       ; Setup idle
       in     r16, PORTA
       ori    r16, $0F
       out    PORTA, r16
       ;
       ldi    YH, HIGH(ICR1_TOP)
       ldi    YL, LOW(ICR1_TOP)
       out    ICR1H, YH
       out    ICR1L, YL
       ;
       ldi    YH, HIGH(ICR1_TOP-1)
       ldi    YL, LOW(ICR1_TOP-1)
       out    OCR1BH, YH
       out    OCR1BL, YL
       ;
       out    OCR1AH, YH
       out    OCR1AL, YL
       call   pwm_start
       ret

; ---------------------------------------
; --- FLASH: Duty cycle constant
; --- It is important that OCRnX its not equal 0 and ICRI_TOP, because it will lead
to interrups crashes. Read about Timer1 and PWM in ATmega16 datasheet!.
; ---------------------------------------
DUTY_CYCLE:
       .db    120, 112, 104, 96, 88, 80, 72, 64, 56, 48, 40, 32, 24, 16, 8, 1

; ---------------------------------------
; --- END: motor.asm
; ---------------------------------------

; ---------------------------------------
```

```
; --- START: ultrasound.asm
; ---------------------------------------

; ---------------------------------------
; --- get_distance: Start the ultrasound module and return a distance between 3-100
cm.
; --- Argument:     none
; --- Return: r16 (distance in cm)
; --- Uses: r16, Y, Z
; --- Usage:
; --- call   get_distance
; ---------------------------------------
get_distance:
       ; Setup
       cbi    PORTA, 6
       ; Start ultrasound sensor
       sbi    PORTA, 6
       ldi    r16, 52
wait_10us:
       dec    r16
       brne   wait_10us
       cbi    PORTA, 6
wait_echo_high:
       sbis   PINA, 5
       rjmp   wait_echo_high
       cli
       call   dist_count_reset
       sei
wait_echo_low:
       sbic   PINA, 5
       rjmp   wait_echo_low
       call   distance_compute
       push   r16
       cli
       call   dist_count_reset
       sei
       pop    r16
       ret

; ---------------------------------------
; --- distance_compute: Use DIST_COUNT to get the distance.
; --- Argument:     DIST_COUNT
; --- Return: r16 (distance in cm)
; --- Uses: r16, Y, Z
; --- Usage:
; --- call   distance_compute
; ---------------------------------------
distance_compute:
```

```
	ldi	YL, LOW(DIST_COUNT)
	ldi	YH, HIGH(DIST_COUNT)
	ldi	ZL, LOW(DISTANCE_TABLE*2)
	ldi	ZH, HIGH(DISTANCE_TABLE*2)
	ld	r16, Y
	cpi	r16, 67
	brlo	not_max_dist
	ldi	r16, 66
not_max_dist:
	add	ZL, r16
	clr	r16
	adc	ZH, r16
	lpm	r16, Z
	ret

; ---------------------------------------
; --- dist_count_reset: Reset DIST_COUNT to zero.
; --- Argument:	None
; --- Return: None
; --- Uses: r16, Y
; --- Usage:
; --- cli
; --- call	dist_count_reset
; --- sei
; ---------------------------------------
dist_count_reset:
	ldi	YL, LOW(DIST_COUNT)
	ldi	YH, HIGH(DIST_COUNT)
	clr	r16
	st	Y, r16
	ret

; ---------------------------------------
; --- timer0_start: Start timer0.
; --- Argument:	None
; --- Return: None
; --- Uses: r16
; --- Usage:
; --- call	timer0_start
; ---------------------------------------
timer0_start:
	ldi	r16, (0 << COM01) | (0 << COM00) | (1 << WGM01) |(0 << WGM00) | (0 <<
CS02) | (1 << CS01) | (1 << CS00)
	cli
	out	TCCR0, r16
	sei
	ret
```

```
; ---------------------------------------
; --- timer0_OCF: An interrupt that increment DIST_COUNT by one every 100
microsecond. The increment is stop when DIST_COUNT reach DIST_COUNT_TOP, DIST_COUNT
will not change until it is cleared.
; --- Argument:     None
; --- Return: None
; --- Uses: r16, Y
; ---------------------------------------
timer0_OCF:
       push   r16
       in     r16, SREG
       push   r16
       push   YL
       push   YH
       ;
       ldi    YL, LOW(DIST_COUNT)
       ldi    YH, HIGH(DIST_COUNT)
       ld     r16, Y
       cpi    r16, DIST_COUNT_TOP
       brne   timer0_inc
       cbi    PINA, 5
       rjmp   timer0_done
timer0_inc:
       inc    r16
       st     Y, r16
timer0_done:
       ;
       pop    YH
       pop    YL
       pop    r16
       out    SREG, r16
       pop    r16
       reti

; ---------------------------------------
; --- ultrasound_hw_init: Init. resources used by ultrasound.
; --- Argument:     None
; --- Return: None
; --- Uses: r16
; --- Usage:
; --- call   ultrasound_hw_init
; ---------------------------------------
ultrasound_hw_init:
       ; Init. port
       in     r16, DDRA
       ori    r16, (1 << DDA6) | (0 << DDA5)
       out    DDRA, r16
       ;
```

```
	ldi	r16, OCR0_TOP
	out	OCR0, r16
	; Init. timer0
	in	r16, TIMSK
	ori	r16, (1 << OCIE0)
	out	TIMSK, r16
	;
	call	dist_count_reset
	call	timer0_start
	ret

; ---------------------------------------
; --- FLASH: Distance constant in cm.
; ---------------------------------------
DISTANCE_TABLE:
	.db	3, 3, 3, 5, 5, 5, 10, 10, 10, 15, 15, 15, 20, 20, 20, 25, 25, 25, 30,
30, 30, 30, 30, 30, 40, 40, 40, 40, 40, 40, 50, 50, 50, 50, 50, 50, 60, 60, 60, 60,
60, 60, 70, 70, 70, 70, 70, 70, 80, 80, 80, 80, 80, 80, 90, 90, 90, 90, 90, 90, 90,
100, 100, 100, 100, 100, 100, $00

; ---------------------------------------
; --- END: ultrasound.asm
; ---------------------------------------
```

B.2 Kontrollen

```
;
; con_main_v3.asm
;
; Created: 2019-03-12 20:08:49
;

; ---------------------------------------
; --- INC START: usart.inc
; ---------------------------------------

	.equ	UBRR_VALUE = 7
	.equ	BUFFER_SIZE = 2

; ---------------------------------------
; --- INC END: usart.inc
; ---------------------------------------

; ---------------------------------------
; --- INC START: motor.inc
; ---------------------------------------
```

```
	; Instruction
	.equ	MOTUR = $02
	.equ	MEDUR = $01
	.equ	FAST_STOP = $03
	.equ	FREE_STOP = $00
	; PWM output frequency
	.equ	ICR1_TOP = 64
	.equ	TOV1_PERIOD = 1039
	; Motor ID
	.equ	MIDA = 0
	.equ	MIDB = 1

; -------------------------------------
; --- INC END: motor.inc
; -------------------------------------

; -------------------------------------
; --- INC START: joystick.inc
; -------------------------------------

	.equ JOYID_RIGHT_X = (1 << ADLAR) | PINA6
	.equ JOYID_RIGHT_Y = (1 << ADLAR) | PINA5
	;
	.equ JOYID_LEFT_X = (1 << ADLAR) | PINA4
	.equ JOYID_LEFT_Y = (1 << ADLAR) | PINA3

; -------------------------------------
; --- INC END: joystick.inc
; -------------------------------------

; -------------------------------------
; --- INC START: lcd.inc
; -------------------------------------

	.equ	LCD_RS = 2
	.equ	LCD_RW = 1
	.equ	LCD_E = 0

; -------------------------------------
; --- INC END: lcd.inc
; -------------------------------------

	.org	0
	jmp	cold
	.org	URXCaddr
	jmp	usart_RXC

	.dseg
```

```
	.org	$60

; ---------------------------------------
; --- SRAM START: usart.asm
; ---------------------------------------

	TX_BUFFER_POS:
		.byte	1
	TX_BUFFER_BEGIN:
		.byte	1
	TX_BUFFER_END:
		.byte	1
	;
	RX_BUFFER_POS:
		.byte	1
	RX_BUFFER_BEGIN:
		.byte	1
	RX_BUFFER_END:
		.byte	1

; ---------------------------------------
; --- SRAM END: usart.asm
; ---------------------------------------

; ---------------------------------------
; --- SRAM START: lcd.asm
; ---------------------------------------

	RECIEVED_DATA:
		.byte	1
	JOYSTICK_X_DATA:
		.byte	1
	JOYSTICK_Y_DATA:
		.byte	1

; ---------------------------------------
; --- SRAM END: lcd.asm
; ---------------------------------------

	.cseg

; ---------------------------------------
; --- START: con_main.asm
; ---------------------------------------

cold:
	ldi	r16, LOW(RAMEND)
	out	SPL, r16
```

```
	ldi	r16, HIGH(RAMEND)
	out	SPH, r16
	call	joy_hw_init
	call	usart_hw_init
	call	lcd_init
	clr	r20
	sei

start:
	; Read distance data from car
	ldi	XL, LOW(RX_BUFFER_BEGIN)
	ldi	XH, HIGH(RX_BUFFER_BEGIN)
	ld	r16, X
	ldi	XL, LOW(RECIEVED_DATA)
	ldi	XH, HIGH(RECIEVED_DATA)
	st	X+, r16 ; Update lcd distance
	; Read joysticks status
	ldi	r16, MIDA
	ldi	YL, JOYID_LEFT_X
	call	read_adc
	call	encode
	push	r16
	st	X+, YH ; Update lcd x
	;
	ldi	r16, MIDB
	ldi	YL, JOYID_RIGHT_Y
	call	read_adc
	call	encode
	st	X, YH ; Update lcd y
	; Transmit buffer
	ldi	YL, LOW(TX_BUFFER_BEGIN)
	ldi	YH, HIGH(TX_BUFFER_BEGIN)
	st	Y+, r16
	pop	r16
	st	Y, r16
	call	usart_buffer_transmit
	; Has x passed?
	inc	r20
	cpi	r20, $FF
	brne	start
	clr	r20
	call	LCD_update
	jmp	start

; ---------------------------------------
; --- encode: Encode motor id, motor state and motor speed into a byte.
; --- Argument: r16 (motor id), YH (speed)
; --- Return: r16
```

```
; --- Uses: r16, r17, r18
; --- Usage:
; --- ldi    r16, MIDB
; --- ldi    YL, JOYID_RIGHT_Y
; --- call   read_adc
; --- call   encode
; ---------------------------------------
encode:
       mov    r18, YH      ; speed
       cpi    r18, 116
       brlo   encode_motur
       cpi    r18, 140
       brsh   encode_medur
       ldi    YH, 128
encode_idle:
       clr    r18
       ldi    r17, FAST_STOP
       rjmp   encode_start
encode_motur:
       com    r18
       subi   r18, -128
       ldi    r17, MOTUR
       rjmp   encode_start
encode_medur:
       subi   r18, -128
       ldi    r17, MEDUR
encode_start:
       lsl    r17
       lsl    r17
       or     r16, r17
       lsr    r18
       lsr    r18
       lsr    r18
       swap   r18
       or     r16, r18
       ret

; ---------------------------------------
; --- END: con_main.asm
; ---------------------------------------

; ---------------------------------------
; --- START: usart.asm
; ---------------------------------------

; ---------------------------------------
; --- usart_buffer_transmit: Iterate over tx_buffer in SRAM and transmit the data
with USART. Start at tx_buffer_begin and stop at tx_buffer_end.
```

```
; --- Argument: SRAM, whole tx_buffer
; --- Return: None
; --- Uses: r16, Y
; --- Usage:
; ---     ldi        YL, LOW(TX_BUFFER_BEGIN)
; ---     ldi        YH, HIGH(TX_BUFFER_BEGIN)
; ---     st         Y+, r16
; ---     pop        r16
; ---     st         Y, r16
; ---     call    usart_buffer_transmit
; ---------------------------------------
usart_buffer_transmit:
    ldi       YL, LOW(TX_BUFFER_BEGIN)
    ldi       YH, HIGH(TX_BUFFER_BEGIN)
usart_buffer_loop0:
    ld        r16, Y+
    call    usart_transmit
    call    wait_RXC_complete
    cpi       YL, LOW(TX_BUFFER_END + 1)
    brne    usart_buffer_loop0
    cpi       YH, HIGH(TX_BUFFER_END + 1)
    brne    usart_buffer_loop0
    ret

; ---------------------------------------
; --- wait_RXC_complete: Wait a period of an usart_buffer_RXC interrupt.
; --- Argument: None
; --- Return: None
; --- Uses: r16
; --- Usage:
; --- call    wait_RXC_complete
; ---------------------------------------
wait_RXC_complete:
    ldi       r16, 10
wait_loop0:
    dec       r16
    brne    wait_loop0
    ret

; ---------------------------------------
; --- usart_transmit: Transmit a byte with USART.
; --- Argument: r16 (data)
; --- Return: None
; --- Uses: r16
; --- Usage:
; --- ldi    r16, $F0
; --- call    usart_transmit
; ---------------------------------------
```

```
usart_transmit:
    sbis    UCSRA, UDRE
    rjmp    usart_transmit
    out       UDR, r16
    ret

; ---------------------------------------
; --- usart_RXC: An interrupt that handle incoming data from the car.
; --- Argument: None
; --- Return: None
; --- Uses: r16, Y
; ---------------------------------------
usart_RXC:
    cbi       PORTB, 1
    push    r16
    in        r16, SREG
    push    r16
    push    YL
    push    YH
    ;
    in        r16, UDR
    ldi       YL, LOW(RX_BUFFER_BEGIN)
    ldi       YH, HIGH(RX_BUFFER_BEGIN)
    st        Y, r16
    pop       YH
    pop       YL
    pop       r16
    out       SREG, r16
    pop       r16
    reti

; ---------------------------------------
; --- usart_hw_init: Init. resources used by USART.
; --- Argument: None
; --- Return: None
; --- Uses: r16, Y
; --- Usage:
; --- call    usart_hw_init
; ---------------------------------------
usart_hw_init:
    ; Init. port
    in        r16, DDRD
    ldi       r16, (1 << DDD1)
    out       DDRD, r16
    ; Init. USART
    ldi       r16, HIGH(UBRR_VALUE)
    out       UBRRH, r16
    ldi       r16, LOW(UBRR_VALUE)
```

```
    out       UBRRL, r16
    ;
    ldi       r16, (1 << RXCIE) | (1 << RXEN) | (1 << TXEN)
    out       UCSRB, r16
    ;
    ldi       r16, (1 << URSEL) | (0 << UMSEL) | (1 << USBS) | (1 << UCSZ1) | (1 <<
UCSZ0)
    out       UCSRC, r16
    ;
    clr       r16
    ldi       YL, LOW(TX_BUFFER_POS)
    ldi       YH, HIGH(TX_BUFFER_POS)
    st        Y, r16
    ldi       YL, LOW(RX_BUFFER_POS)
    ldi       YH, HIGH(RX_BUFFER_POS)
    st        Y, r16
    ret

; ---------------------------------------
; --- END: usart.asm
; ---------------------------------------

; ---------------------------------------
; --- START: joystick.asm
; ---------------------------------------

; ---------------------------------------
; --- read_adc: Read ADC pin and return digital representation.
; --- Argument YL (ADC pin)
; --- Return: Y (data)
; --- Uses: Y
; --- Usage:
; --- ldi    YL, JOYID_LEFT_X
; --- call   read_adc
; ---------------------------------------
read_adc:
       out    ADMUX, YL
       sbi    ADCSRA, ADSC
read_adc_wait0:
       sbic   ADCSRA, ADSC
       rjmp   read_adc_wait0
       in     YL, ADCL
       in     YH, ADCH
       ret

; ---------------------------------------
; --- joy_hw_init: Init. resources used by joystick.
; --- Argument: None
```

```
; --- Return: None
; --- Uses: r16
; --- Usage:
; --- call   joy_hw_init
; ---------------------------------------
joy_hw_init:
       ; Init. port
       in     r16, DDRA
       andi   r16, (1 << DDA7) | (0 << DDA6) | (0 << DDA5) | (0 << DDA4) | (0 << DDA3)
| (1 << DDA2) | (1 << DDA1) | (1 << DDA0)
       out    DDRA, r16
       ; Init. ADC
       ldi    r16, (1 << ADEN) | (1 << ADSC)| (1 << ADPS2) | (1 << ADPS1) | (1 <<
ADPS0) //125kHz
       out    ADCSRA, r16
       ret

; ---------------------------------------
; --- END: joystick.asm
; ---------------------------------------

; ---------------------------------------
; --- START: lcd.asm
; ---------------------------------------

; ---------------------------------------
; --- lcd_init: initierar lcd.
; --- Argument r16 (ADC pin)
; --- Return: none (data)
; --- Uses: 16
; ---------------------------------------
lcd_init:
       ldi    r16, $50 ;start delay
       call   delay
       ldi    r16, $FF
       out    DDRB, r16
       ldi    r16, $07
       out    DDRA, r16
       call   setup
       call   function_set
       call   display_on
       call   display_clear
       call   entry_mode
       ret

; --- lcd_update: skriver allt på skärmen, main loopen som kallar allt annat.
lcd_update:
       call   display_clear
```

```
	call	distance_printer
	call	joystick_direction
	ret


; ------------------------------------
; --- joystick_direction: denna kod hanterar joystick printing.
; --- Argument: none (ADC pin)
; --- Return: none (data)
; --- Uses: r21
; ------------------------------------
joystick_direction:
	ldi	r21, $C0
	call	setup_enable
	call	X_print
	call	joystick_X
	call	adaptive_byte
//	ldi	r21, $20 ; Write space between Y and X
//	call	write_to_lcd
	ldi	r21, $C7
	call	setup_enable
	call	Y_print
	call	joystick_Y
	call	adaptive_byte
	ret

; ------------------------------------
; ---distance_printer: denna kod hanterar distans printning.
; --- Argument: none (ADC pin)
; --- Return: none (data)
; --- Uses: none
; ------------------------------------
distance_printer:
	call	write_distance
	call	sensor_data
	call	adaptive_byte
	call	cm_print
	ret

; ------------------------------------
; --- function_set: set 2-line mode, sets transfer to 8-bit and 5x11 mode
; --- Argument: none
; --- Return: none
; --- Uses: r21
; ------------------------------------
function_set:
	ldi	r21, $3F ; function set
	call	Setup_Enable
```

```
	nop
	ret

; -------------------------------------
; --- display_on: turns on the display
; --- Argument: none
; --- Return: none
; --- Uses: r21
; -------------------------------------
display_on:
	ldi	r21, $0E ; display and cursor working
	call	Setup_Enable
	nop
	ret

; -------------------------------------
; --- display_clear: clears the display
; --- Argument: none
; --- Return: none
; --- Uses: r21, r16
; -------------------------------------
display_clear:
	ldi	r21, $01 ; display clear
	call	Setup_Enable
	ldi	r16, $2A ; 1,53ms delay
	call	delay
	ret

; -------------------------------------
; --- entry_mode: cursor mode set
; --- Argument: none
; --- Return: none
; --- Uses: r21, r16
; -------------------------------------
entry_mode:
	ldi	r21, $06 ; entry mode set
	call	Setup_Enable
	ldi	r16, $A0 ; lång delay innan skrivning
	call	delay
	ret

; -------------------------------------
; --- setup_enable: ska göras alltid när man ska skriva till displayen
; --- Argument: none
; --- Return: none
; --- Uses: none
; -------------------------------------
Setup_Enable:
```

```
	call	setup
	call	enable
	ret



; ------------------------------------
; --- write_distance: print word from .db
; --- Argument: none
; --- Return: none
; --- Uses: r21, Z
; ------------------------------------
write_distance:
	push	r21
	push	ZH
	push	ZL
	ldi	ZH, HIGH(DISTANCE*2)
	ldi	ZL, LOW(DISTANCE*2)
write_distance_inner:
	lpm	r21, Z
	cpi	r21, 0
	breq	finished
	lpm	r21, Z+
	call	write_to_lcd
	jmp	write_distance_inner
finished:
	pop	ZL
	pop	ZH
	pop	r21
	ret

; ------------------------------------
; --- write_to_lcd: denna funktion sköter skrivning till displayen
; --- Argument: none
; --- Return: none
; --- Uses: r16
; ------------------------------------
write_to_lcd:
	call	write
	call	enable
	ldi	r16, $10
	call	delay
	ret

; ------------------------------------
; --- write: ställer in skrivmode som sätter att vi vill skicka data
; --- Argument: none
```

```
; --- Return: none
; --- Uses: r21
; -------------------------------------
write:
       sbi    PORTA, LCD_RS ; RS hög
       cbi    PORTA, LCD_RW ; RW låg
       out    PORTB, r21    ; DATA UT
       ret

; -------------------------------------
; --- setup: ställer in setupmode som sätter att vi vill skicka instruktioner
; --- Argument: none
; --- Return: none
; --- Uses: r21
; -------------------------------------
setup:
       cbi    PORTA, LCD_RS ; RS låg
       cbi    PORTA, LCD_RW ; RW låg
       out    PORTB, r21    ; DATA UT
       ret

; -------------------------------------
; --- enable: the sequense to do something with the display
; --- Argument: none
; --- Return: none
; --- Uses: r21
; -------------------------------------
enable:
       nop
       sbi    PORTA, LCD_E ; E hög
       ldi    r16, $04
       call   delay
       cbi    PORTA, LCD_E ; E låg
       nop
       cbi    PORTA, LCD_RS ; RS låg
       cbi    PORTA, LCD_RW ; RW låg
       ret

; -------------------------------------
; --- delay: delay function
; --- Argument: none
; --- Return: none
; --- Uses: r16, r17
; -------------------------------------
delay:
       push   r17
inner_delay1:
       ldi    r17, $FF
```

```
inner_delay2:
       dec    r17
       brne   inner_delay2
       dec    r16
       brne   inner_delay1
       pop    r17
       ret

; ------------------------------------
; --- split_number: tar ett tal och delar upp det
; --- Argument: none
; --- Return: r26, r25, r24
; --- Uses: r26, r26, r25, r23, Z
; ------------------------------------
joystick_X:
       push   ZH
       push   ZL
       ldi    ZH, HIGH(JOYSTICK_X_DATA)
       ldi    ZL, LOW(JOYSTICK_X_DATA)
       rjmp   split_number
joystick_Y:
       push   ZH
       push   ZL
       ldi    ZH, HIGH(JOYSTICK_Y_DATA)
       ldi    ZL, LOW(JOYSTICK_Y_DATA)
       rjmp   split_number
Sensor_data:
       push   ZH
       push   ZL
       ldi    ZH, HIGH(RECIEVED_DATA)
       ldi    ZL, LOW(RECIEVED_DATA)
split_number:
       clr    r26 ;hundra-delar
       clr    r25 ;tio-delar
       clr    r24 ;entals-delar
       clr    r23
       ld     r23, Z
hundred_loop:
       cpi    r23, 100
       brlo   ten_loop
       inc    r26
       subi   r23, 100
       rjmp   hundred_loop

ten_loop:

       cpi    r23, 10
       brlo   singular_loop
```

```
	inc	r25
	subi	r23, 10
	rjmp	ten_loop

singular_loop:
	cpi	r23, 0
	breq	done
	inc	r24
	subi	r23, 1
	rjmp	singular_loop
done:
	pop	ZL
	pop	ZH
	ret
; -------------------------------------
; --- adaptive_byte: använder split number och filtrerar bort onödiga siffror
; --- Argument: r26, r25, r24
; --- Return: none
; --- Uses: r26, r25, r24, r21, r19
; -------------------------------------
adaptive_byte:
	cpi	r26, $00
	brne	first_byte
	cpi	r25, $00
	brne	second_byte
	rjmp	third_byte
first_byte:
	mov	r19, r26
	call	write_byte
second_byte:
	mov	r19, r25
	call	write_byte
third_byte:
	mov	r19, r24
	call	write_byte
	ret

write_byte:
	subi	r19, -$30 ; lägg till addi till assembler SNÄLLA
	mov	r21, r19
	call	write_to_lcd
	ret

; -------------------------------------
; --- cm_print: skriver ut cm på skärmen
; -------------------------------------
cm_print:
	ldi	r21, $63
```

```
	call	write_to_lcd
	ldi	r21, $6d
	call	write_to_lcd
	ret
; ------------------------------------
; --- Y_print: skriver ut Y på skärmen
; ------------------------------------
Y_print:
	ldi	r21, $59
	call	write_to_lcd
	ldi	r21, $3A
	call	write_to_lcd
	ret

; ------------------------------------
; --- X_print: skriver ut X på skärmen
; ------------------------------------
X_print:
	ldi	r21, $58
	call	write_to_lcd
	ldi	r21, $3A
	call	write_to_lcd
	ret
; ------------------------------------
; --- DISTANCE: tabell som innehåller ordet distance
; ------------------------------------
DISTANCE:
	.db	"DISTANCE:", $00

; ------------------------------------
; --- END: lsd.asm
; ------------------------------------
```