LUNDS UNIVERSITET
Lunds Tekniska Högskola

Digitala Projekt(EITF40) – Robert Linjeföljaren

Handledare – Bertil Lindvall

Jonathan Åkerblom, ie09jak

Tobias Olsson, ie09to3

Abstract

The purpose of this project was to build a robot with the ability to follow a black taped line on a white background. This was going to be implemented through IR emitting diodes and by sampling with NPN photo transistors. Using an Atmel AVR mega16 we could sample analog values from these sensors and convert them into manageable digital values. Having these we could steer a pair of DC motors with PWM according to our own algorithm. This was implemented together with an H-bridge and enabled us to set individual velocities on the motors to make the robot turn.

In the end our algorithm worked successfully, although the motors didn't due to their lack of strength. That setback also kept us from making our code more detailed and strongly calibrated since it couldn't be tested enough.

Contents

Inledning

Syftet med det här projektet var att få lära sig att utveckla en prototyp utifrån en idé och att fördjupa kunskaperna inom digitalteknik. Att bygga någon form av robot var något som tidigt lockade och slutligen bestämde sig gruppen för att utveckla en robot som på egen hand skulle kunna följa en svart, tejpad linje ovanpå en vit yta. Då ingen av oss besatt några kunskaper inom ämnet sedan tidigare krävdes ordenligt förarbete med läsning på internet och i aktuella komponenters datablad. En linjeföljare kan implementeras på olika sätt, men vår idé var att lösa det med hjälp av någon form av optisk sensor som kan skilja på ljus och mörk miljö. Då tänkte vi att IR-ljus skulle passa bra, eftersom mängden IR-ljus som reflekteras från en yta varierar beroende på färgen. Detta ledde i sin tur fram till en idé om en vektor uppbyggd av parvisa dioder, där den ena skickar ut IR-ljus och den andra på något sätt fångar upp reflekterat IR-ljus. På så sätt skulle man då kunna avgöra var roboten befinner sig i förhållande till en svart linje och styra den därefter.

Kravspecifikation

Prototypen skall kunna:

- Köra rakt fram.
- Följa en svart linje/bana tejpad ovanpå en vit bakgrund.
- Svänga höger och vänster med olika hastigheter beroende på hur snett den har kommit i förhållande till linjen.
- Komma ihåg åt vilket håll den ska köra för att hitta linjen igen om den råkar köra utanför.
- Sättas igång med en strömbrytarknapp.
- Visa status genom en lysdiod om den är på eller av.
- Visa status genom lysdioder beroende på om de tolkar sitt synfält som svart eller vitt.

Teori

Komponenter som använts vid byggandet av prototypen:

J-TAG

För att integrera processorn med utvecklingsmiljön AVR-Studio och överföra C-koden till processorn används en J-TAG.

Processor

Projektet bygger på en bas bestående av mikroprocessorn Atmel AVR Atmega16. Processorn har 40 pinnar och ett 16KB stort minne för ett förprogrammerat program. Av dessa är 32 stycken programmerbara I/O-pinnar. Dessa är uppdelade i de fyra portarna A-D. Fem av de åtta pinnarna med möjlighet till A/D-omvandling, port A, används för att omvandla inhämtad data från sensorerna från analog till digital. Fyra av pinnarna på port C används av J-taggen. De två motorerna använder två pinnar var fördelade på port C och D. De har även vars en Enable Input som båda ligger på port D. Ytterligare sex pinnar används till kontrollampor för felsökning och övervakning under körtid.

Motorkontrollerare

Motorerna krävde en starkare spänning än vad som kan fås ut ur processorn. För att uppnå detta användes en L298 H-brygga som i själva verket består av två H-bryggor, en för varje motor.

Motorer
För att kunna förflytta roboten användes två likströmsmotorer som via motorkontrolleraren (Enable A/B) och processorn (OC1A/B) drevs av den medelspänning vi skickade in till dem i form av olika långa pulser, vilket i sin tur bestämde deras hastigheter.

Spänningsregulator
Spänningskällan var ett batteri på 6V, då vissa komponenter tålde högst 5V krävdes en spänningsregulator som gjorde den nödvändiga omvandlingen från 6 till 5 Volt. Vi använde oss av en LP3852.

Sensor
Vår sensor utgjordes av fem stycken ”par” som bestod av en IR-emittor, TSUS5400, och en NPN fototransistor, TLP321. Till dessa kopplades motstånd på 220 ohm respektive 1 Mohm enligt kopplingsschemat. Se Bilaga 1. Fototransistorerna kopplades till vars en ingång till AD-omvandlaren för att göra om den mängd ström som leds genom fototransistorerna till digital data.

Motstånd och kondensatorer
Motstånd och kondensatorer kopplades enligt kopplingsschemat. Se Bilaga 1. Hur dessa skulle kopplad kunde erhållas ur de övriga komponenternas datablad.

Utförande
Projektet delades ursprungligen upp i fyra faser; planering, montering, kodning och testning. Dessa faser överlappade dock varandra allteftersom diverse problem hopade sig.

Under planeringen skrevs först kravspecifikationen och därefter började ett kopplingsschema att ritas upp. För att kunna göra detta krävdes mycket läsning bland äldre projekt, på internet och i tillgängliga komponenters datablad. Även handledarens expertis var till stor nytta.

Själva monteringen skedde förhållandevis smärtfritt med en del felsökningar av hårdvaran. Efter att ha stött på lite problem med för klena kablar och felaktigt kopplade motstånd och kondensatorer kunde kodningen sättas igång.

Kodandet var nog det momentet som var mest tidsödande då det krävdes en enorm mängd efterforskning om både hårdvara och språk för att ta sig runt problemen och alla de underliga fenomen som uppstod.

Testningen skedde parallellt med kodningen där vi gick från att försäkra oss om hårdvarans funktion till kalibrering av dessa.

Rent intuitivt blev vårt projekt uppdelat i två delar, där sensorerna stod för ena delen och motorstyrningen stod för den andra delen. Vi utvecklade dem till att börja med helt separerade ifrån varandra för att sedan synkronisera dem enligt våra krav och preferenser.

Sensorvektorn är som tidigare nämnt uppbyggd med hjälp av parvisa IR-emittorer och NPN fototransistorer. De analoga utsignalerna från sensorerna leder rakt in i AD-omvandlaren på processorn, där signalernas storlek beror helt på hur mycket IR-ljus som absorberas av fototransistorn. Då det är en NPN-fototransistor vi har vid varje par innebär det att ju mindre IR-ljus som absorberas (mörk yta), desto svagare signal fås ut. Detta beror på att den vid avsaknad av ljus har ett väldigt högt motstånd. Den analoga signalen från varje fototransistor omvandlas av ADC’n till

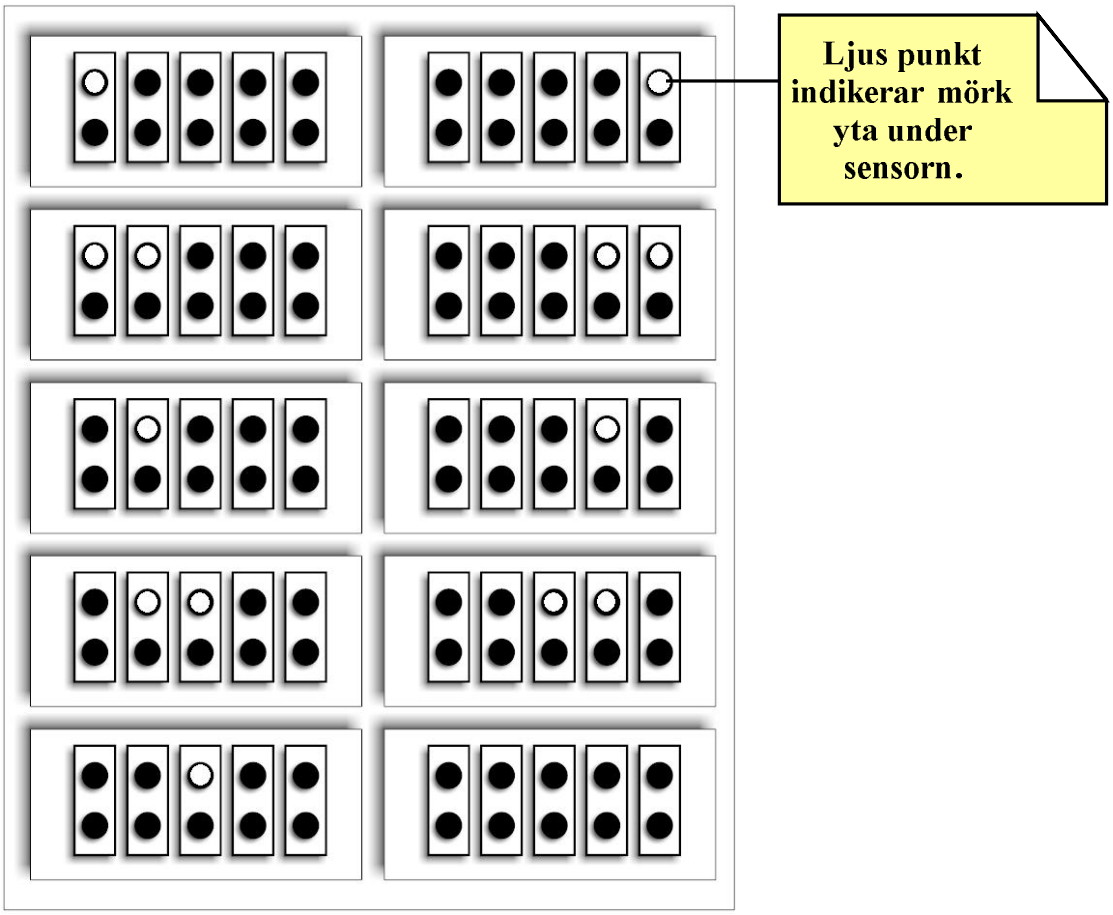
en 8-bitars digital signal. Det medför att man tillåts utläsa värden från ett spänningsintervall med 256 olika värden.

Tricket med en sådan här signalomvandling är att man har ett rimligt spänningsintervall. För att styra intervallet kalibrerade vi fram ett rimligt motstånd med hjälp av ett variabelt motstånd. På så sätt kunde vi se till att vi hamnade kring en maximal spänning över fototransistorerna på 2,6-3,1 V. Sedan stod vi inför valet med att bestämma en referensspänning, som i sin tur bestämmer den maximala spänningen i ett intervall. Vår processor hade ett alternativ som innebar en intern referensspänning på 2,56 V. Genom att välja den här kunde vi skära av värden som våra sensorer hade klarat av. På så sätt blev sensorerna mer jämlika, vilket också kan förklaras genom att vissa sensorer aldrig hade kunnat demonstrera vissa värden i intervallet om man hade låtit andra sensorer, som kan ge ut en högre spänning än andra, bestämma den maximala spänningen. Därmed utsatte vi oss också för risken att en del sensorer toppade värden, men våra tester visade sedan att vi aldrig fick ut några värden som indikerade mättning.

Vidare i arbetets gång med sensorvektorn implementerade vi sampling av värden genom att för varje sensor starta en omvandling. En sådan omvandling tar 13 cyklar innan en avbrottsflagga dyker upp. Denna fångas i sin tur upp av en ISR-funktion och tillåter en att läsa av ett fryst digitalt värde ur ADCH. Sedan ändrar man helt enkelt kanal i ADMUXen vilket bestämmer vilken sensor man vill läsa av. Vår idé var att göra repetera den här proceduren fem gånger konsekutivt tills man samplat värden från alla sensorerna. På så sätt fick man alla värdena med så lite tidsskillnad som möjligt. Eftersom ADC-kretsen kräver en 50kHz-200kHz klocksignal ställde vi in en pre-scaler på 64 för att på så sätt få en frekvens på 125kHz, vilket är den snabbaste frekvensen vi kan få med en 8MHz systemklocka och ändå hålla oss inom intervallet. Efter de fem omvandlingarna hade man en vektor med fem 8-bitars värden att tillgå, som sedan fick användas i main-metoden till att styra motorerna. När motorernas fart ställts in följaktligen, kunde fem nya omvandlingscyklar initieras igen och på så sätt fortsätter det.

Vad gäller förvaltningen av de fem nästintill realtidssamplade värdena kom vi fram till att det lättaste sättet att styra efter dessa värden var om man på något sätt kunde med säkerhet säga om en diod just nu befinner sig över en svart linje eller inte. Antingen eller, alltså. Eftersom varje diod blev på något sätt unik i vektorn på grund av vår amatörmässiga design, var det ju ingen chock att man fick olika värden. Med lite testning kunde vi få fram en gräns där man med säkerhet kunde säga att sensorn befann sig över en svart linje. Med den här gränsen kunde vi då göra om en mängd 8-bitars värden till fem nollor och ettor.

Med hjälp av den kraftiga nerkortningen av samplingen kunde en styrningsalgoritm implementeras lättare. Vad vi gjorde då var att ställa upp tio olika scenarion. De kan bli många fler men eftersom vår bil inte ens kunde rulla med den motor vi hade, så kändes det onödigt att skriva extremfallskod som vi inte ens kan testa. I bilden nedan ses de tio olika fallen och de ligger uppradade utefter grad av svängningshastighet på motorn. Desto längre ut på vardera sida sensorn linjen befinner sig, desto mer vill motorn svänga. Undantaget i bildens upplägg är den där ingen sensor befinner sig över en linje. Det innebär att bilen har slagit över, vilket vi löste med att komma ihåg vilken sida den tappade bort sig på och sedan dra på en extra kraftig sväng tillbaka åt rätt håll. Det enda scenariot som inte innebär en sväng är den där endast mittensensorn befinner sig över linjen. Då är det gasen i botten rakt fram.



En sväng motsvarar i teorin antingen att de två hjulen har olika hastigheter i samma eller olika riktning. Vi lät bara ena motorn ha varierande och även lägre hastighet än den andra.

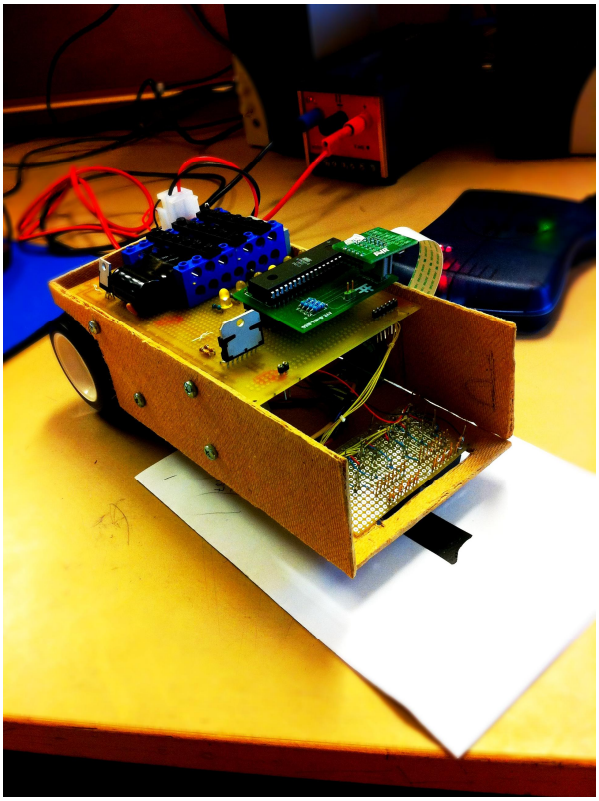
För att implementera en motor med variabel hastighet, där motorn i sig styrs av storleken på spänningen som ligger på dess Enable-utgångar, måste man på något sätt kunna ge den spänningen man vill ha från processorn. Detta är endast genomförbart genom PWM – Pulse Width Modulation. Det innebär i praktiken att man, med hjälp av en pulserad konstant in-spänning med olika förhållanden mellan pulsernas period (tid på + tiden av) och pulsernas bredd (tid på), kan få ut olika medelspänningar. Förhållandet kallas också ’Duty cycle’ eller ’pulskvot’. Desto kortare puls desto mindre blir kvoten och således även medelspänningen ut. 100% pulskvot innebär att man har maximal spänning ut. 50% pulskvot innebär halva spänningen ut.

Det sättet vi genomförde det här på var att använda oss av Timer1 och ställde in den på 8-bitar, 1MHz frekvens och använde OC1A/B som utgång (de är Enable A/B till motorn). För att sedan reglera pulskvoten använde man OCR1A/B som variabel. 8-bitarsinställningen gav en talmängd på 0 till 255, där ungefär 90 var det lägsta som motorn klarade av, vilket satte vår lägsta-hastighet. OCR1A/B satt till 255 gav maxhastighet. Däremellan definierade vi fyra hastigheter med godtyckliga mellanrum, som sedan fick representera varsitt scenario hos styralgoritmen.

Resultat och diskussion

De tidigare formulerade kraven uppfylls alla bortsett från en mindre detalj. Motorerna och hjulen kan inte driva bilen. Då hjulen har varit med ett antal år har det tagit ut sin rätt på dem. Hjulen kunde inte ens snurra med motorernas vikt så därför har vi fått nöja oss med att simulera körningen med hjulen i luften och visa på hur deras hastighet varierar enligt vår styrningsalgoritm. Vi planerar dock att på egen hand införskaffa en ny uppsättning motorer och hjul för att på riktigt kunna få se resultatet av vårt projekt. En annan sak som påverkar mycket är att när vi använder oss av batteri så blir spänningen väldigt ojämn, vilket i sin tur påverkar styrningsalgoritmen väldigt påtagligt.
Med spänningsaggregat eller ett välfungerande batteri och hjulen uppallade uppfylls de tidigare nämnda kraven. När strömbrytarknappen slås på börjar lysdioden, som signalerar att roboten är på, att lysa. När ett vitt papper med en svart tejpad linje förs under sensorerna lyser dioderna som representerar de sensorer som befinner sig över den svarta tejpen. Då endast den mittersta dioden lyser rullar båda hjulen med högsta fart. Då pappret och den svarta linjen flyttas i sidled minskas det ena hjulets hastighet proportionellt mot hur långt ut på sensorn linjen är. Då den helt kommer utanför svänger roboten kraftigt för att söka upp linjen igen. De olika svänghastigheterna skulle med största sannolikhet behöva justeras med fungerande motorer och hjul för att optimera robotens förmåga att följa linjen med minsta möjliga avvikelse.

I efterhand kan vi tycka att det skulle varit en bra idé att kontrollera mer än att bara hjulen kan snurra, utan också kan driva en prototyp. Detta är något som hade kunnat göras i ett tidigt stadie och då hade man också haft tid att beställa hem nya och välfungerande motor- och hjulset. I övrigt är vi väldigt nöjda med resultatet då roboten med fungerande motorer och hjul och justering av svänghastigheter med största sannolikhet skulle uppfylla alla de krav som satts upp i början av projektet.

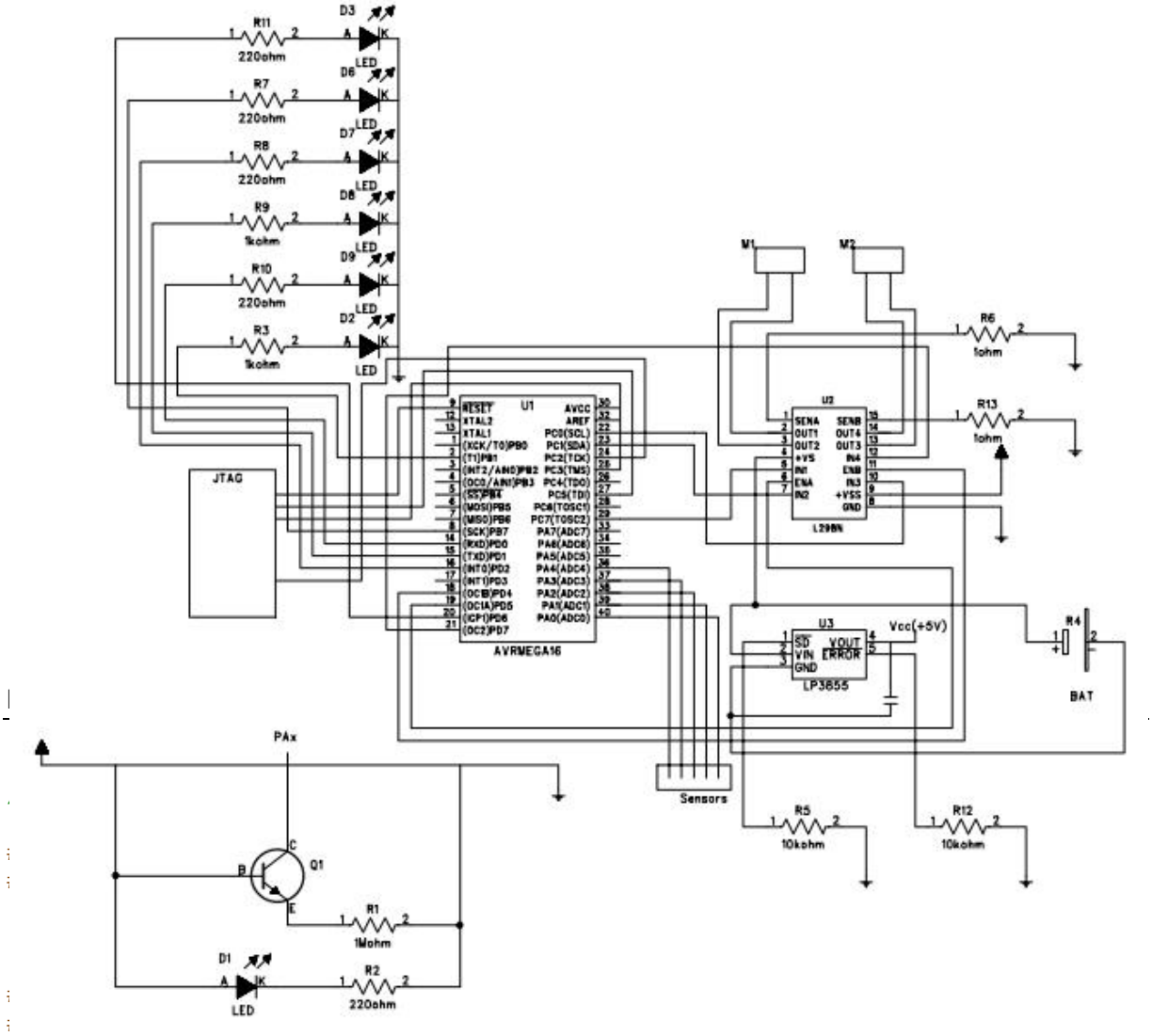


Källförteckning

ATMEL ATmega16 Datasheet.
http://www.eit.lth.se/fileadmin/eit/courses/edi021/datablad/Processors/ATmega16.pdf

Duel Full Bridge Driver L298 Datasheet.
http://www.eit.lth.se/fileadmin/eit/courses/edi021/datablad/Analog/linear/l298.pdf

Bilaga 1: Kopplingsschema



Bilaga 2: Källkod

```
/* <-------- Defines ---------> */

#define sbi(x,y) x |= _BV(y)                    //'Set bit'-function
#define cbi(x,y) x &= ~(_BV(y))                 //'Clear bit'-function
#define NUM 5                                   //Num steers array spacing
                                                //depending on number of sensors
                                                //we want to use.
#define IN 1          //Duh?
#define OUT 0

#define    LEFTSPEED OCR1AL      //OCR1AL's value determines the speed of
                                 //the left motor.
#define RIGHTSPEED OCR1BL        //OCR1BL's value determines the speed of
                                 //the right motor.
#define V1 90                    //Lowest speed
#define V2 110
#define V3 130
#define V4 170
#define V5 210
#define V6 255                   //Highest speed

/* <--- Method Prototyping ---> */

                     //Diodes (x => off):
void d1(void);       //1. PB0
void d1x(void);
void d2(void);       //2. PD0
void d2x(void);
void d3(void);       //3. PD1
void d3x(void);
void d4(void);       //4. PD2
void d4x(void);
void d5(void);       //5. PD3
void d5x(void);
void d6(void);       //6. PD6
void d6x(void);

void initPorts(void);            //Initiate all in- and output
void initPWM(void);              //Initiate Pulse Width Modulation,
                                 //enabling motor steering.
void initADC(void);              //Initiate the settings for the
                                 //AD-converter.


                                 //Left motor:
void leftForward(void);          //Forward
void leftBack(void);             //Back     (Not used in this edition)
void leftStop(void);             //Stop     (Not used in this edition)

                                 //Right motor:
void rightForward(void);         //Forward
void rightBack(void);            //Back     (Not used in this edition)
void rightStop(void);            //Stop     (Not used in this edition)


          //Calculate where the line is at
          //and set the 'states'-array with
          //1's and 0's.
void calculateStates(void);
```

```
		//Algorithm that from the values in
		//'states' steers the motors.
void steerAccordingToStates(void);

/* <---- Global Variables ----> */

short int sensor;		//Periodically incrementable variable
				//to keep track of the current sensor
				//that is to be read.
short int conversionCycle;	//Keep track of current conversion cycle
				//nbr.

unsigned int values[NUM];	//Int values from ADCH (sensors)
short int states[NUM];		//States of sensors in 0's or 1's. 1 being
				//above line.
char lastSensorOnSide;		//Keep track of which outermost sensor
				//that was last to read the line.

int s1;
int s2;
int s3;
int s4;
int s5;

/* <---------- Main ----------> */

int main(void)
{
		d6();			//Turns on diode, indicating power is on.

		sensor = 1;		//Start at the first sensor.
		conversionCycle = 1;

		initPorts();
		initPWM();
		initADC();
		sei();			//Enables interrupts
		sbi(ADCSRA, ADSC);	//ADC Start Conversion

		while(1) {
			//If we have scanned all the 5 sensors
			//we want to transform the values to 1's and 0's
			//and then set each motor's speed according to these.
			//Ultimately we start a new series of conversions
			//to collect new values.
			if (conversionCycle%NUM == 0 & (ADCSRA&0x10) == 0) {
				calculateStates();
				rightForward();
				leftForward();
				steerAccordingToStates();
				sbi(ADCSRA, ADSC);//ADC Start Conversion
			}
		}
		return 1;
}

void calculateStates(void) {
		int i = 0;
		unsigned int k;
		while (i < NUM) {
			k = values[i];
```

```
                        switch ( i ) {
                                case 0:
                                        if (k < 150) {
                        //'k' = 150 happened to be a good limit value,
                        //but should actually be calibrated according
                        //to each sensor's conditions.
                                                states[i] = IN;
                                                d5();
                                        } else {
                                                states[i] = OUT;
                                                d5x();
                                        }
                                        break;
                                case 1:
                                        if (k < 150) {
                                                states[i] = IN;
                                                d4();
                                        } else {
                                                states[i] = OUT;
                                                d4x();
                                        }
                                        break;
                                case 2:
                                        if (k < 150) {
                                                states[i] = IN;
                                                d3();
                                        } else {
                                                states[i] = OUT;
                                                d3x();
                                        }
                                        break;
                                case 3:
                                        if (k < 150) {
                                                states[i] = IN;
                                                d2();
                                        } else {
                                                states[i] = OUT;
                                                d2x();
                                        }
                                        break;
                                case 4:
                                        if (k < 150) {
                                                states[i] = IN;
                                                d1();
                                        } else {
                                                states[i] = OUT;
                                                d1x();
                                        }
                                        break;
                        }
                        i++;
                }
}

void steerAccordingToStates(void) {
                s1 = states[1];       //Position above line expr. in 1's & 0's.
                s2 = states[2];
                s3 = states[3];
                s4 = states[4];
                s5 = states[5];
```

```
if (s1 == 1) {
                    //If the rightmost sensor is above line,
          lastSensorOnSide = 'r';          //we remember that.
} else if (s5 == 1) {                      //Else if the
                                           //leftmost sensor is
                                           //above line
          lastSensorOnSide = 'l';          //we remember that.
}

                                           //Position meaning:
if ( (s1 + s2 + s3 + s4 + s5) == 0) {
                                           //Overshoot

          switch ( lastSensorOnSide ) {
                    case 'l':
                              LEFTSPEED = V1;
                              RIGHTSPEED = V6;
                              break;
                    case 'r':
                              LEFTSPEED = V6;
                              RIGHTSPEED = V1;
                              break;
          }
} else if ( (s3 == 1) && ((s2 + s4) == 0) ) {
                              //Perfectly aligned
          RIGHTSPEED = V6;
          LEFTSPEED = V6;
} else if ( (s2 + s3) == 2 && ((s1 + s4) == 0) ) {
                              //Slightly to the left
          RIGHTSPEED = V5;
          LEFTSPEED = V6;
} else if ( (s3 + s4) == 2 && ((s2 + s5) == 0) ) {
                              //Slightly to the right.
          RIGHTSPEED = V6;
          LEFTSPEED = V5;
} else if ( (s2 == 1) && ((s1 + s3) == 0) ) {
                              //More to the left.
          RIGHTSPEED = V4;
          LEFTSPEED = V6;
} else if ( (s4 == 1) && ((s3 + s5) == 0) ) {
                              //More to the right.
          RIGHTSPEED = V6;
          LEFTSPEED = V4;
} else if ( (s1 + s2) == 2 && ((s3 + s4) == 0) ) {
                              //Even more to the left.
          RIGHTSPEED = V3;
          LEFTSPEED = V6;
} else if ( (s4 + s5) == 2 && ((s2 + s3) == 0) ) {
                              //Even more to the right.
          RIGHTSPEED = V6;
          LEFTSPEED = V3;
} else if ( (s1 == 1) && ((s2 + s3) == 0) ) {
                              //Outermost position on left
                              //side before overshoot.
          RIGHTSPEED = V2;
          LEFTSPEED = V6;
} else if ( (s5 == 1) && ((s3 + s4) == 0) ) {
                              //Outermost position on right
                              //side before overshoot.
          RIGHTSPEED = V6;
          LEFTSPEED = V2;
```

```
            }
}

ISR(ADC_vect) {

            values[sensor-1] = ADCH;          //Get the upper 8-bits

            if (sensor == NUM) {              //Incrementing 'sensor' until it
                        sensor = 1;           //reaches '6', then set to '1'.
            } else {
                        sensor++;
            }

            ADMUX &= ~(0x1F);                 //Selects Analog Channel 0
                                              //or in other words it erases
                                              //whatever bit that was checked
                                              //before this point.

            switch ( sensor ) {               //Switches the Analog Channel
                        case 2:               //depending on what sensor we
                                              //want to read.
                                    sbi(ADMUX, MUX0);
                                    break;
                        case 3:
                                    sbi(ADMUX, MUX1);
                                    break;
                        case 4:
                                    sbi(ADMUX, MUX0);
                                    sbi(ADMUX, MUX1);
                                    break;
                        case 5:
                                    sbi(ADMUX, MUX2);
                                    break;
            }


            if(conversionCycle % NUM != 0) { //This ensures 5 consecutive
                        conversionCycle++;    //conversions.

                        sbi(ADCSRA, ADSC);    //ADC Start Conversion

            } else {
                        conversionCycle = 1;
            }
}

ISR(BADISR_vect)                  //This vector is only triggered if an ISR
{                                 //fires with no accompanying ISR handler.
            sbi(PORTB, PB0);      //(i.e. debugging reasons)
}

void initPorts(void) {

            sbi(DDRD, PD4);                   //
            sbi(PORTD, PD4);                  //Enable B
            sbi(DDRD, PD5);                   //
            sbi(PORTD, PD5);                  //Enable A

            sbi(DDRD, PD7);                   //
            sbi(DDRC, PC0);                   //Left motor
            sbi(DDRC, PC1);                   //
```

```
            sbi(DDRC, PC6);                 //Right motor

                                            //Diode:
            sbi(DDRB, PB0);                 // 1.
            sbi(DDRD, PD0);                 // 2.
            sbi(DDRD, PD1);                 // 3.
            sbi(DDRD, PD2);                 // 4.
            sbi(DDRD, PD3);                 // 5.
            sbi(DDRD, PD6);                 // 6.
}

void initADC(void) {

            /* <---ADMUX---> */
            sbi(ADMUX, REFS1);    //
            sbi(ADMUX, REFS0);    //Voltage Reference: Internal 2,56V

            sbi(ADMUX, ADLAR);    //Left adjust

            ADMUX &= ~(0x1F);     //Just declaring the start at sensor 1.

            /* <---ADCSRA---> */
            sbi(ADCSRA, ADIE);    //ADC Interrupt Enable

            cbi(ADCSRA, ADPS0);   //
            sbi(ADCSRA, ADPS1);   //
            sbi(ADCSRA, ADPS2);   //ADC Prescaler div. factor: 64 (=>125kHz)

            sbi(ADCSRA, ADEN);    //ADC Enable (Should be last)
}

void initPWM(void) {//Settings of Timer1
            sbi(TCCR1A, WGM10);   //Fast PWM mode, 8-bit (look below as
                                  //well).
            sbi(TCCR1A, COM1A1);  //Clear OC1A on compare match.
            sbi(TCCR1A, COM1B1);  //Clear OC1B on compare match.

            sbi(TCCR1B, CS11);    //Prescaler of 8 => 1MHz clock frequency.
            sbi(TCCR1B, WGM12);   //This together with 'WGM10' sets the Fast
                                  //8-bit PWM mode.
}
```

Steering.c

```
#define sbi(x,y) x |= _BV(y)
#define cbi(x,y) x &= ~(_BV(y))

void leftForward(void) {
            cbi(PORTC, PC0);
            sbi(PORTD, PD7);
}

void rightForward(void) {
            sbi(PORTC, PC1);
            cbi(PORTC, PC6);
}
```

Diodes.c

```
#define sbi(x,y) x |= _BV(y)
#define cbi(x,y) x &= ~(_BV(y))

void d1(void) {                  //Diod1 På
           sbi(PORTB, PB0);
}

void d1x(void) {                 //Diod1    Av
           cbi(PORTB, PB0);
}

void d2(void) {                  //Diod1 På
           sbi(PORTD, PD0);
}

void d2x(void) {                 //Diod2    Av
           cbi(PORTD, PD0);
}

void d3(void) {                  //Diod3 På
           sbi(PORTD, PD1);
}

void d3x(void) {                 //Diod3    Av
           cbi(PORTD, PD1);
}

void d4(void) {                  //Diod4 På
           sbi(PORTD, PD2);
}

void d4x(void) {                 //Diod4    Av
           cbi(PORTD, PD2);
}

void d5(void) {                  //Diod5 På
           sbi(PORTD, PD3);
}

void d5x(void) {                 //Diod5    Av
           cbi(PORTD, PD3);
}

void d6(void) {                  //Diod6 På
           sbi(PORTD, PD6);
}

void d6x(void) {                 //Diod6    Av
           cbi(PORTD, PD6);
}
```

Bilaga 3: Bilder