Driving Miss Daisy
EEL 5666 IMDL
Final Report

Moishe Groger
TA: Mike Pridgen, Thomas Vermeer
Instructors: Dr. A. Antonio Arroyo, Dr. Eric M. Schwartz

Abstract

Driving Miss Daisy (DMD) is an autonomous wheelchair designed to assist with day-to-day transportation, including mobility inside a house or apartment. DMD is targeted for the elderly or disabled who have limited coordination and have difficulty manipulating a joystick but can be used by any wheelchair user for safety and convenience. DMD will use several sonar sensors for obstacle avoidance, a series of focused IR sensors for line tracking, a sip/puff controller and a joystick for user interaction.

Executive Summary

DMD may be used in self-navigating or manual modes. While self-navigating, DMD uses the custom line tracker to track a tape line to the appropriate stop. The stop is selected using a sip/puff switch and the LCD screen.

While in manual mode the user can drive the DMD as they wish using the joystick. In either modes sonar sensors are used for obstacle avoidance. The base DMD platform was a box-shaped FIRST robot used in a 2001 high school robotics competition. The platform was modified into a wheelchair by adding foot rests and a chair back. An arm was added on the side to hold the joystick, sip/puff controller and LCD screen.

The platform was modified further to improve traction and general drivability. The front wheels and motors were replaced with a basic suspension system and casters. The casters were mounted to a arm that can travel several inches up and down on either side, mounted in the middle to the rest of the frame. The suspension system and casters allow DMD to drive easily on uneven terrain which was not possible in the original design. It also allows for much easier turning.

Bicycle tread was adding to the rear wheels to improve traction. The front and rear wheels that came with the platform were made by gluing several sheets of plywood together to create a solid wood wheel. A timing belt was then glued around the driving surface for traction. To further improve traction, I added bicycle tread around the outside of the wheels.

Introduction

DMD is a mouth-controlled (using a sip/puff sensor) autonomous robot that can self-navigate to any number of destinations. DMD allows the user to select from a list of possible destinations on the LCD screen. Once a destination is selected, DMD uses a custom line-tracking board to follow a black line to the appropriate position and stop. Sonar sensors are used for obstacle avoidance and a joystick is also available for manual control as well as off-track navigation.

Originally, DMD was going to use a grid system for navigation using a series of IR sensors and a compass to determine current position. This approach would have been more flexible but it would have been much more difficult to implement and demonstrate. A line-tracking demo could be more interesting since the tape lines could lead to actual destinations in the building, not limited by the practical size of a grid which would have to be small relative to the DMD.

Also, the DMD was going to use a wireless interface with a laptop to map the current position and allow the user to select a new destination. Since the DMD is designed to assist the mobility-impaired, Dr. Schwartz suggested that a better control system might utilize the users breath.

Integrated System

As mentioned previously, DMD operates in one of two states - self navigating or manual control. In both states DMD simply stops if a potential collision is detected so I won't discuss the obstacle avoidance further in this section.

DMD starts in the manual control state. The user can drive pretty much however they want (as long as they don't run into anything). The menu of possible destinations is shown on the LCD screen.

If the user wants to navigate to one of the predefined destinations the user must first position DMD somewhere on the first section of line. Then, the sip/puff control may be used to switch into self navigating mode. Individual puffs are used to scroll through the different options for destinations. Then, a sip is used to select the currently displayed destination and transition into self navigating mode.

In self-navigating mode DMD attempts to follow the line until it achieves the proper intersection or destination. The actual decision process is somewhat complicated since the line tracker readings were somewhat glitchy and prone to errors. The code tries to average out and error check as much as possible to make intelligent decisions with noisy data.

While self-navigating a puff will pause self-navigating mode and switch back to manual mode without resetting the current position on the line or the desired destination. A sip will reset both and switch back to manual mode. If self-navigation mode isn't interrupted it will continue to follow the line past the correct number of intersections until the current position matches the desired position. When the correct destination is attained DMD will switch back into manual mode so the user can either drive using the joystick or select another destination.

Mobile Platform

The platform is based on an old FIRST competition robot donated by Andy Thon. The platform structure was constructed using 1/4" polycarbonate plastic sheets and aluminum bars. After removing the pneumatic system an office chair back and a gator cushion was added to the top to make a seat and an arm was added to side with a platform for the joystick, the LCD and the sip/puff switch. Footrests were added, also constructed from polycarb and aluminum.

The wheels and motors in the front were replaced with casters mounted on an axle that allows the casters to move up and down to adjust to uneven ground.

Actuation

The platform came complete with four 60A motor drivers driving two drill motors in the back and two barbie car motors in the front. The two motors on the left side are controlled by one PWM signal and the two on the right are controlled by another. The platform now uses just two motor drivers - one for each of the rear drill motors. The drill motors are hacked 12V Bosch cordless drills with a maximum current draw of 58 Amps. Each drill motor is on a 30A fuse and they share a 60A breaker so the maximum power draw on the platform is well under 30 Amps. Using the high-torque gearbox (64:1) a stall torque of 29 N-m is achieved. One of the drill motors seems to put out more power than the other but they seem to both be in good shape.

The motors and motor drivers have been tested with the microcontroller and have been calibrated to stop using the appropriate PWM signal. Full forward and full reverse PWM signals have also been calibrated. I included an opto-coupler on my analog multiplexer board to isolate the motor drivers from the microcontroller and sensors. It turns out that with my Victor 883 motor drivers this wasn't necessary as the motor drivers have their own built-in optocoupling.

Sensors

Line Following

DMD uses a custom 6-sensor IR board that I designed and built. The IR board includes six potentiometers for threshold adjustment. The selected threshold is compared with the IR sensor output and a digital value for each sensor is sent to the microcontroller.

The custom 6 sensor IR board includes potentiometers and analog comparators for threshold adjustment. I used the OPB745 reflective sensors for the IR sensing. I also added a red LED at each digital output so I can watch the line tracking in real time to verify threshold values. The OPB745 sensors are very sensitive to vertical positioning. I mounted them at about 1/2" off the ground where they worked quite well on a perfectly flat surface. Unfortunately, the surfaces in NEB and Benton vary by at least 1/8" which can accumulate to 1/4" across the width of DMD even with the new suspension system. Thankfully, the floor in NEB is much flatter than the linoleum in Benton - instead of going both up and down by 1/8" it only seems to have 1/8" dips.

The OPB745 sensors seem to work well overall in NEB using a matte black tape. Unfortunately, they are still prone to glitches to due high differences and the black areas of the floor cause false positives in the evening when more artificial lighting is used.

For the first few days that I tried to line follow I kept loosing the line. I thought that it was because the wheels didn't have enough traction so I added the bicycle tread. That wasn't it. Then I thought it was because all my turns in the code were pretty hard and it was causing DMD to jerk too hard and loose the line. Without motor encoders it was impossible to make a slight turn due to the unpredictable nature of the turns - sometimes just going straight would veer hard to one side or the other. I even thought at one point that one of the drill motors was not running well and I took it apart and cleaned it with some WD-40.

Well.. it was none of those. Certainly they all had an effect but the real reason I was having trouble following lines turned out to be the LCD. Updating the LCD every time I looped through the code used up 99% of the available execution time and the line tracking code was only running about twice a second (2Hz). By removing that code and only updating the LCD minimally the loop could run hundreds of times in a second and was able to actually respond to the environment.

Sonar (5)

I designed and built a custom 8:1 analog multiplexer board with five headers for the five Maxbotix LV-MaxSonar-EV0 sonar sensors and a game port connector for the joystick. I'm using the analog outputs from the sonar sensors since its faster and easier to read in the values. Also, I'll be using the microcontroller to control the operation of the sonar sensors so that they run in a synchronized fashion and don't interfere with each other.

When I first wrote the code for the sonar I used timer 0 in a similar fashion to the sleep timer on timer 2 provided by the Mike and Thomas. I don't know if there's a difference in the clocking options between timer 0 and timer 2 but with identical settings my timer 0 interrupt ran at 1/10 ms until I saw the problem and changed the output compare from 12 to 120. I used the resulting 1 ms timer interrupt to keep track of time for the main program loop and pull in new values from the ADC every 5ms. Once every 5ms a new value is read from the ADC and the external analog mux is switched to the next channel so that the data will be ready 5ms later when the ADC is read again. So an individual sonar or joystick reading is updated every 40ms. The timer interrupt is also used to trigger the sonar sensors so that they fire simultaneously and don't interfere with each other.

Sip/Puff

I purchased one pressure/vacuum switch for use a sip/puff but realized that I would need two of them to make a full sip/puff as one switch can only be used in sip or puff mode.

Before ordering another switch to complete the design I found a full sip/puff switch at a lower price from http://quadcontrol.com/. The sip/puff system I purchased included the sip/puff switches in a box, additional push button switches to simulate a sip or puff and a short section of hose.

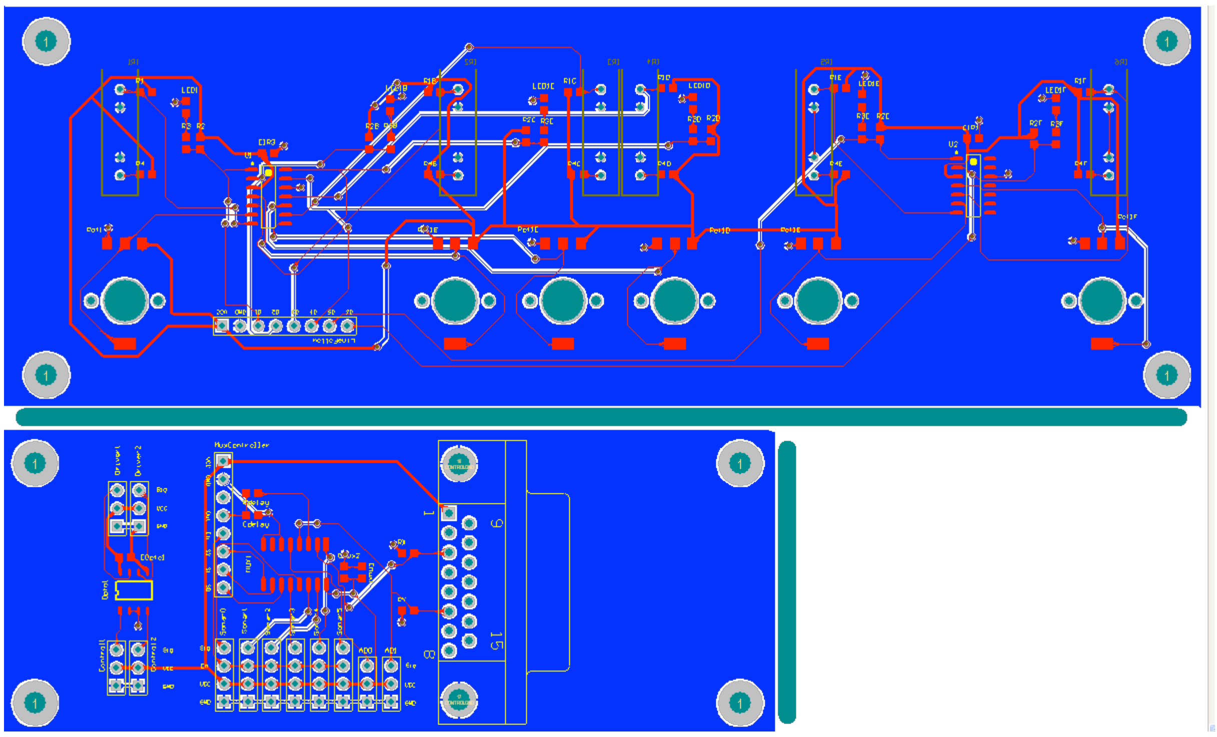
A list of possible destinations will be displayed on the LCD screen. Puffs will be used to scroll through the list of options and a sip will be used to select the current option. Once a option is selected the DMD will start navigating towards the selected destination by following the line and keeping track of intersections.

Joystick

The joystick was donated along with the FIRST platform. It includes potentiometers for adjusting the center values so I'll calibrate the center each time DMD is turned on. The analog multiplexer board I made includes a game port connector, and two pull-up resistors to read the X/Y potentiometer values coming from the joystick.

When using the joystick the DMD will allow the user to pick a desired direction on the fly (along with speed) using the joystick.

Originally, if a potential collision was detected, DMD was going to slow down and help the user turn around the object. Unfortunately, with a platform of this size it can be difficult to add enough sonar and/or IR sensors for obstacle avoidance around the entire robot. Without full coverage it is not possible to know where obstacles are located while trying to drive around them so the "intelligent" portion of the joystick control had to be abandoned. With the current design, DMD simply stops if the user is driving backward and a close object is detected to the rear. Similarly, DMD will stop if the user is driving forward and a close object is detected ahead.



The top board is the 6 IR line tracker. The bottom board has a two-channel optocoupler on the left, then the 8:1 analog multiplexer with headers for sonar and a gameport and pull-up resistors for a joystick connection.

Conclusion

Overall, I'm really happy with the result of all my hard work this semester.

I don't think I got my money's worth with the sonar. I spent about $120 on sonar sensors and wasn't able to do a whole lot with them due to the size of my robot. I think I would rather get a whole bunch of cheaper IR sensors so I can actually cover the entire perimeter of the robot and do some interesting things with the readings instead of just stopping when something gets too close. That said, the sonar sensors did work well - I just wasn't really able to take full advantage of them because I didn't have enough of them.

I think the OPB745 sensor was a bad choice for a robot of this size. It is simply too difficult to keep the sensors at the precise height that they seem to need. The output from the sensor is virtually digital even before the comparator with a voltage range of 0.80 V to 3.5V or so which means the potentiometer is pretty much useless as any value between 1V and 3V will work pretty well. This is good in a way but it's also bad because you can't really adjust for conditions - it either works or it doesn't.

Finally, I think that knowing the actual speed of the wheels would be very helpful when trying to smoothly track a line. Predictable turns were difficult on this platform and I found that while sometimes DMD would go straight with no problems sometimes it would veer hard to the left or hard to the right. This might be attributed to the small casters or uneven floor or even the drive train. Whatever the reason, it was pretty much impossible to code slight turns - all turns had to be somewhat abrupt making the line tracking jerky.

Before this class I knew nothing about motors and suspension and I never heard of PWM signals. I'd never tapped aluminum for a bolt or tried to follow a line. Needless to say I've learned a lot about robotics but I've also learned a surprising amount about mechanical and electrical issues and I have a new appreciation for robots that actually work.

Appendix 1 – Code

Robot.c

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "ADC.h"
#include "PVR_Servos.h"
#include "LCD.h"
#include "sleep.h"

#define JOYSTICKCENTER          190
#define LT_PORT                 (PIND & 0x3F)
#define TRUE 1
#define FALSE      0

#define INTERSECTION 0x1E
#defineFL          0x20
#define FLL        0x30
#defineL           0x10
#define LML        0x18
#defineML          0x08
#defineMR          0x04
#define M          0x0C
#define RMR        0x06
#defineR           0x02
#define FRR        0x03
#define FR         0x01

#define NavJoystick        0
#define NavLine                 3

/* #define STRAIGHT        25, 25
#define SOFT_LEFT  10, 30
#define MID_LEFT   7, 35
#define HARD_LEFT4, 35
#define SOFT_RIGHT         30, 10
#define MID_RIGHT 35, 7
#define HARD_RIGHT         35, 4*/

/*#define STRAIGHT         24, 24
#define SOFT_LEFT  8, 27
#define MID_LEFT   5, 30
#define HARD_LEFT4, 30
#define SOFT_RIGHT         27, 8
#define MID_RIGHT 30, 5
```

```
#define HARD_RIGHT         30, 4 */

#define STRAIGHT   25, 25
#define SOFT_LEFT  15, 40
#define MID_LEFT   15, 55
#define HARD_LEFT10, 65
#define SOFT_RIGHT         40, 15
#define MID_RIGHT 55, 15
#define HARD_RIGHT         65, 10

#define sonar_fl        adcvalues[4]
#define sonar_fr        adcvalues[3]
#define sonar_l                 adcvalues[2]
#define sonar_fm        adcvalues[1]
#define sonar_b                 adcvalues[0]

inline void updateMotors(int left, int right);
char startLCD();
void FlashLight(int x);

volatile unsigned char trackToLine();

//variables
volatile unsigned char NavState = NavJoystick;
volatile int rightMotor = 0;
volatile int leftMotor = 0;

//sip/puff switches
// 0: no sip, 1: sipping, ... 200: sipping, 0: no sip.
volatile unsigned char sipState = 0;
volatile unsigned char puffState = 0;
volatile unsigned char sipComplete = FALSE;
volatile unsigned char puffComplete = FALSE;

volatile unsigned int lcd_update = 0;
volatile unsigned int ms_clock = 0;
volatile int adcvalues[8];

char destinations[3][10] = {
        "Home     ",
        "Bath     ",
        "Kitchen  "};
volatile unsigned char selDestIndex = 1;
volatile unsigned char currPosIndex = 0;

//debug.....
```

```
volatile unsigned char trackresult;
//unsigned char debounceIntersection;
volatile unsigned char inIntersection = 0;

//keep track of older line tracking readings
//unsigned char ltreadings[8];
//unsigned char ltindex = 0;
unsigned char lostline = 0;
volatile unsigned int lostlineCount = 0;
volatile unsigned int intersectionCount = 0;

//freq
unsigned volatile int countExec = 0;
unsigned volatile int totalExec = 0;

int main(void)
{
        initSleep();
        FlashLight(5);
        lcdInit();
        lcdString("Moishe Groger");

        //ms_sleep(2000);
        initServo();
        initADC();

        lcdClear();

        //setup input pins for sip/puff
        DDRA &= ~0x03;
        PORTA |= 0x03; //enable pull-up for A1, A0

        //setup input pins for line tracker
        DDRD &= ~0x3F;
        PORTD &= ~0x3F; //disable pull-up for D5 .. D0.

        //for (ltindex = 0; ltindex < 8; ltindex++) {
        //        ltreadings[ltindex] = 0;
        //}

        ms_sleep(500);

        while (1) {

                countExec++;
```

```
if ((PINA & 0x03) == 0x00) {
        puffState = 205;
        sipState = 205;

} else if ((PINA & 0x03) != 0x03) {

        //sip/puff switches
        // 0: no sip, 1: sipping, ... 3: sipping, 0: no sip.
        if ((PINA & 0x01) == 0x00) {
                //lcdClear();
                //lcdString("puff");
                if (puffState < 5) puffState++;
                //go while button pushed
                //updateMotors(SOFT_LEFT);
        }
        if ((PINA & 0x02) == 0x00) {
                //lcdClear();
                //lcdString("sip");
                if (sipState < 5) sipState++;
                //go while button pushed
                //updateMotors(SOFT_RIGHT);
        }
} else {
        if (puffState > 4) {
                puffComplete = TRUE;
        }
        if (sipState > 4) {
                sipComplete = TRUE;
        }

        puffState = 0;
        sipState = 0;
}



//if (startLCD()) {

// output raw joystick signals
/*lcdString("joystick y: ");
lcdInt(adcvalues[7]);
lcdGoto(1,0);
lcdString("joystick x: ");
lcdInt(adcvalues[6]);*/

// output pwm signals
```

```
		/*lcdString("left: ");
		lcdInt(leftMotor);
		lcdString("     ");
		lcdGoto(1,0);
		lcdString("right: ");
		lcdInt(rightMotor);
		lcdString("     ");*/

		//output sonar readings
		/*lcdString("l:");
		lcdInt(sonar_fl);
		lcdGoto(0,5);
		lcdString("m:");
		lcdInt(sonar_fm);
		lcdGoto(0,10);
		lcdString("r:");
		lcdInt(sonar_fr);
		lcdGoto(1,0);
		//lcdString("l:");
		//lcdInt(sonar_l);
		//lcdGoto(1,5);
		lcdString("b: ");
		lcdInt(sonar_b);
		lcdString("exe: ");
		lcdInt(totalExec); */

		//}

		if (NavState == NavJoystick) {
			//output nav menu
			if (startLCD()) {
				lcdString("Select Dest: ");
				//lcdInt(totalExec);
				lcdGoto(1,0);
				lcdInt(selDestIndex + 1);
				lcdString(": ");
				lcdString(destinations[selDestIndex]);
			}

			//outputting this to screen constantly slows exec from 12 Hz to 5
Hz.

			if (puffComplete == TRUE) {
				//process puff
				selDestIndex++;
				if (selDestIndex == 3) selDestIndex = 0;
		}
		if (sipComplete == TRUE) {
			//process sip, switch to line tracking mode.
			NavState = NavLine;

			lcdClear();
			lcdString("Sip to Cancel");
			lcdGoto(1,0);
			lcdString("Puff to Pause");
		}

	} else if (NavState == NavLine) {

		/*if (startLCD()) {
			lcdString("Sip - Cancel ");
			lcdInt(trackresult);
			lcdGoto(1,0);
			lcdString("freq: ");
			lcdInt(totalExec);
			//lcdString("Puff - ?");
		}*/

		trackresult = trackToLine();

		if (trackresult == 0x00) {
			//lost line
			if (lostline == 0) {
				lostline = 1;
				lostlineCount = 0;
			}

			if (lostlineCount > 700 && lostlineCount <= 900) {
				//slow down and turn more.
				if (leftMotor > rightMotor) {
					updateMotors(40, -10);
				} else {
					updateMotors(-10, 40);
				}
			} else if (lostlineCount > 900) {
				//slow down and turn more.
				if (leftMotor > rightMotor) {
					updateMotors(30, -25);
				} else {
					updateMotors(-25, 30);
				}
			}
```

```
				//wait for the line to be lost for 3.5 seconds
				if (lostlineCount > 3500) {
					/*if (leftMotor > rightMotor) {
						updateMotors(10, 70);
					} else {
						updateMotors(70, 10);
					}*/

					NavState = NavJoystick;
					lcdClear();
					//reset count, start searching from scratch
					lostline = 0;
				}
			} else {
				lostline = 0;
			}

			if (leftMotor == 0x00 && rightMotor == 0x00) {
				//start moving if glitches or something else
				// stop movement, or prevent from starting
				updateMotors(STRAIGHT);
			}

			if ((trackresult & INTERSECTION) == INTERSECTION) {
				if (inIntersection == 0 && intersectionCount > 3000) {
					//just got to intersection, haven't been to an
intersection in awhile
					inIntersection = 1;
					intersectionCount = 0;
				} else if (inIntersection == 1) {   // && intersectionCount >
5
					currPosIndex++;
					if (currPosIndex == 3) currPosIndex = 0;
					inIntersection = 0;  //done with intersection
					//wait at least 3 seconds before allowing another
intersection
					intersectionCount = 0;
				}
			} else {
				//not in intersection anymore
				inIntersection = 0;
			}

			if (currPosIndex == selDestIndex || sipComplete == TRUE) {
				NavState = NavJoystick;
				selDestIndex = 1;
				currPosIndex = 0;
				lcdClear();
			} else if (puffComplete == TRUE) {
				NavState = NavJoystick;
				lcdClear();
			}
		}

		if (NavState == NavJoystick) {
			//Joystick driver
			//137, 190, 255 full forward (y)		-53, +65
			//141, 190, 255 full left (x)		-49, +65
			//get x, y positions
			int ypos = adcvalues[7] - JOYSTICKCENTER;
			int xpos = adcvalues[6] - JOYSTICKCENTER;
			if (ypos < 0) {  //scale forward speed to 50%
				ypos = (ypos * ypos) / -50;
			} else {
				ypos = (ypos * ypos) / 84;
			}
			if (adcvalues[6] - JOYSTICKCENTER < 0) {  //scale turning to
50%
				xpos = (xpos * xpos) / -50;
			} else {
				xpos = (xpos * xpos) / 84;
			}

			//if ((PINA & 0x03) == 0x03) {
				leftMotor = ypos - xpos;
				rightMotor = ypos + xpos;
			//}

		}

		//avoid obstacles - only while line tracking?
		if ((sonar_fl < 5 || sonar_fr < 5 || sonar_fm < 5) &&
			leftMotor > 2 && rightMotor > 2) {

			//obstacle ahead
			updateMotors(0, 0);
		}
		if (sonar_b < 6 && leftMotor < -5 && rightMotor < -5) {
			//obstacle behind
			updateMotors(0, 0);
		}
		//Change motor speed, cap at 100%
		if (leftMotor < -100) {
			leftMotor = -100;
		} else if (leftMotor > 100) {
			leftMotor = 100;
		}
		if (rightMotor < -100) {
			rightMotor = -100;
		} else if (rightMotor > 100) {
			rightMotor = 100;
		}
		moveServo1(rightMotor);
		moveServo2(leftMotor);

		//FlashLight(5);
		//ms_sleep(10);

		puffComplete = FALSE;
		sipComplete = FALSE;
	}

	return 0;

}

unsigned char trackToLine()
{
	static unsigned char prevDir = 0;
	static unsigned char prevDir2 = 0;
	static unsigned char prevDir3 = 0;
	static unsigned char prevDir4 = 0;
	static unsigned char prevDir5 = 0;
	unsigned char currDir = prevDir & prevDir2 & prevDir3 & prevDir4 & prevDir5
& LT_PORT;  //simple filter

	if (currDir == FL) {
		updateMotors(HARD_LEFT);
	} else if (currDir == L || currDir == FLL) {
		updateMotors(MID_LEFT);
	} else if (currDir == ML || currDir == LML) {
		updateMotors(SOFT_LEFT);
	} else if (currDir == M) {
		updateMotors(STRAIGHT);
	} else if (currDir == MR || currDir == RMR) {
		updateMotors(SOFT_RIGHT);
	} else if (currDir == R || currDir == FRR) {
		updateMotors(MID_RIGHT);
	} else if (currDir == FR) {
		updateMotors(HARD_RIGHT);
	} else if ((currDir & INTERSECTION) == INTERSECTION) {
		updateMotors(MID_LEFT);
		//slow down near intersection even if not stopping
	} else if (currDir == 0x00) {
		//just keep going in current direction for awhile (2 seconds)
	} else {
		char lineOnLeft = 0;
		char lineOnRight = 0;
		char turn;
		if (currDir & FL == FL) lineOnLeft++;
		if (currDir & L == L) lineOnLeft++;
		if (currDir & ML == ML) lineOnLeft++;
		if (currDir & MR == MR) lineOnRight++;
		if (currDir & R == R) lineOnRight++;
		if (currDir & FR == FR) lineOnRight++;

		turn = lineOnLeft - lineOnRight;
		if (turn > 2) {
			updateMotors(SOFT_LEFT);
		} else if (turn < 2) {
			updateMotors(SOFT_RIGHT);
		} else {
			updateMotors(STRAIGHT);
		}
	}

	prevDir5 = prevDir4;
	prevDir4 = prevDir3;
	prevDir3 = prevDir2;
	prevDir2 = prevDir;
	prevDir = LT_PORT;

	//ltreadings[ltindex] = LT_PORT;
	//ltindex++;
	//if (ltindex > 7) ltindex = 0;

	return currDir;
}

inline void updateMotors(int left, int right) {
	leftMotor = left;
	rightMotor = right;
```

```
}

char startLCD() {
	//only allow update every 200 ms
	if (ms_clock > lcd_update + 200 || ms_clock < lcd_update) {
		if (ms_clock < 800) lcd_update = ms_clock;
		else lcd_update = ms_clock - 800;
		lcdGoto(0, 0);
		return 1;
	}
	return 0;
}

void FlashLight(int x){
   DDRB = 0b00000001;
	int i;
	for (i=0;i<x;x--){
		ms_sleep(100);
		PORTB = 0b11111111;
		ms_sleep(100);
		PORTB = 0b00000000;
		ms_sleep(100);
	}
}
```

ADC.c

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "ADC.h"
#include "sleep.h"

extern volatile unsigned int ms_clock;
extern volatile int adcvalues[8];
extern volatile unsigned int countExec;
extern volatile unsigned int totalExec;
extern volatile unsigned int lostlineCount;
extern volatile unsigned int intersectionCount;

unsigned volatile char sonar_enable = 0;
unsigned volatile char sonar_clk = 0;
unsigned volatile int adc_channel = 0;
int counter = 0;

unsigned long tempsonar = 0;
unsigned volatile char sonar_fl_index = 0;
unsigned volatile char sonar_fr_index = 0;
unsigned volatile char sonar_l_index = 0;
unsigned volatile char sonar_fm_index = 0;
unsigned volatile char sonar_b_index = 0;
int sonarreadings[20];

void initADC(void)
{
//init port F for AD
        DDRF = 0x00;            //set as input
        PORTF = 0;
//set to use internal ARef, and put value in ADCH
        ADMUX = 0b01100000;
        ms_sleep(100);          //wait for power up
//set to on, free running at 1/2 times CLK
        ADCSRA = ~(_BV(ADIE));
        //ADCSRA |= (1 << ADIE);  //enable ADC interrupt
        //ADCSRA |= (1 << ADSC);  //start adc conversions
        ms_sleep(100);          //wait for initialization

        //setup timer
        TIFR  |= _BV(OCIE0);
        TCCR0  = _BV(WGM01)|_BV(CS02)|_BV(CS00); /* CTC, prescale = 128 */
        TCNT0  = 0;
        TIMSK |= _BV(OCIE0);    /* enable output compare interrupt */
        OCR0   = 120;          /* match in 1 ms */
        sei();

        //setup output pins for adc mux
        DDRA |= 0xF0;
        //en (7), s2, s1, s0
        PORTA |= 0xF0; //enable mux, channel 8
}

/*
 * 1 millisecond counter interrupt vector
 */
SIGNAL(SIG_OUTPUT_COMPARE0)
{
        if (ms_clock % 5 == 0) {
                // average values if coming from sonar
                switch (adc_channel) {

                        case 4:
                                //front left
                                sonarreadings[4*4 + sonar_fl_index] = ADCH;
                                sonar_fl_index++;
                                if (sonar_fl_index == 4) sonar_fl_index = 0;
                                tempsonar = 0;
                                for (int i = 16; i < 20; i++) {
                                        tempsonar = tempsonar + sonarreadings[i];
                                }
                                adcvalues[adc_channel] = tempsonar >> 2;
                                break;

                        case 3:
                                //front right
                                sonarreadings[3*4 + sonar_fr_index] = ADCH;
                                sonar_fr_index++;
                                if (sonar_fr_index == 4) sonar_fr_index = 0;
                                tempsonar = 0;
                                for (int i = 12; i < 16; i++) {
                                        tempsonar = tempsonar + sonarreadings[i];
                                }
                                adcvalues[adc_channel] = tempsonar >> 2;
                                break;

                        case 2:
                                //left
                                sonarreadings[2*4 + sonar_l_index] = ADCH;
                                sonar_l_index++;
                                if (sonar_l_index == 4) sonar_l_index = 0;
                                tempsonar = 0;
                                for (int i = 8; i < 12; i++) {
                                        tempsonar = tempsonar + sonarreadings[i];
                                }
                                adcvalues[adc_channel] = tempsonar >> 2;
                                break;

                        case 1:
                                //front middle
                                sonarreadings[1*4 + sonar_fm_index] = ADCH;
                                sonar_fm_index++;
                                if (sonar_fm_index == 4) sonar_fm_index = 0;
                                tempsonar = 0;
                                for (int i = 4; i < 8; i++) {
                                        tempsonar = tempsonar + sonarreadings[i];
                                }
                                adcvalues[adc_channel] = tempsonar >> 2;
                                break;

                        case 0:
                                //rear
```

```
                    sonarreadings[0*4 + sonar_b_index] = ADCH;
                    sonar_b_index++;
                    if (sonar_b_index == 4) sonar_b_index = 0;
                    tempsonar = 0;
                    for (int i = 0; i < 4; i++) {
                        tempsonar = tempsonar + sonarreadings[i];
                    }
                    adcvalues[adc_channel] = tempsonar >> 2;
                    break;

                default:
                    //joystick
                    adcvalues[adc_channel] = ADCH;

            }

            //switch channel
            adc_channel = adc_channel + 1;
            if (adc_channel == 8) adc_channel = 0;
            //enable should be high for 20ms, low for 40ms
            if (sonar_clk == 4) sonar_enable = 0b00000000;
            if (sonar_clk == 12) sonar_enable = 0b10000000;
            //control analog mux
            PORTA = ((adc_channel & 0b00000111) << 4) | sonar_enable |
                (PORTA & 0b00001111);
            //increment variables
            sonar_clk++;
            if (sonar_clk == 13) sonar_clk = 1;
        }

        lostlineCount++;
        if (intersectionCount < 4000) intersectionCount++;
        ms_clock++;

        if (ms_clock == 1001) {
            ms_clock = 1;
            totalExec = countExec;
            countExec = 0;
        }

}

int adcZero(void)
{
        ADMUX = 0b01100000;
        ms_sleep(1);
```

```
	return ADCH;
}

int adcOne(void)
{
	ADMUX = 0b01100001;
	ms_sleep(1);
	return ADCH;

}

int adcTwo(void)
{
	ADMUX = 0b01100010;
	ms_sleep(1);
	return ADCH;
}

int adcThree(void)
{
	ADMUX = 0b01100011;
	ms_sleep(1);
	return ADCH;
}
```