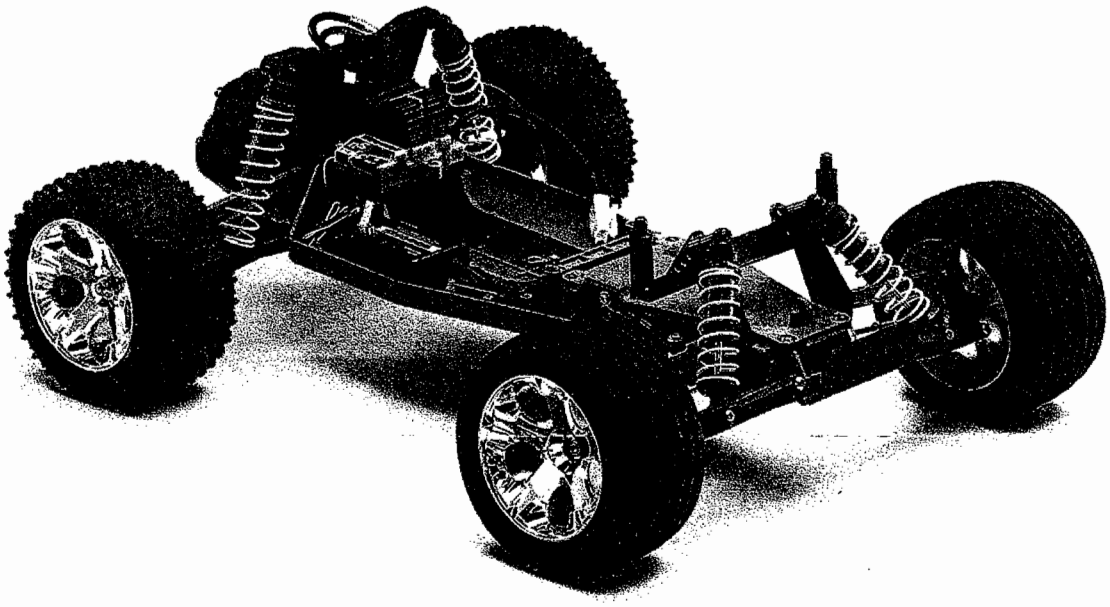
"GNAV"

A+

GPS Navigated Autonomous Vehicle

EECS 129B
Professor Klefstad

Chris Abernethy
Jason Meachum
Michael Soliman



The goal of this project is to design and construct an automated vehicle which is guided by GPS and other sensory inputs, capable of traveling from a given location to a chosen destination unassisted by human intervention. The path that the vehicle will take will be predetermined using set waypoints for the vehicle to travel. To accomplish this we will be using a small scale electronically controlled and powered four wheeled vehicle.

Guardall
Shelter

Description and Purpose

According to the National Highway Traffic Safety Administration, in 2005 the United States incurred 43,443 traffic fatalities, 2,699,000 injuries, and an estimated $230.6 billion economic cost as the result of traffic related accidents. Human error and poor judgment lead not only to traffic accidents, but to inefficient use of our roadways in general. Every day millions of people needlessly spend hours in traffic or otherwise risk their lives by putting faith in the competence of other motorists.

We envision a future without these problems. If it would be possible to remove the human factor from public roads, the world would be a much safer and more efficient place to live in. Impaired driving would be non existent and all vehicles would be able to travel quickly, efficiently, and safely even in the most congested of traffic. While the technology required to implement this vision is still in its infancy, the goal of our project will be to take a step in the direction needed to accomplish this goal.

By using a small scale electronic vehicle, we hope to be able to automate its travel between two locations from a given origin to a desired destination along a predetermined path. The vehicle will be free of human control or intervention and guided by way of the Global Positioning System along a path of assigned waypoints. The vehicle that will serve as a base for our modifications will be a Traxxas brand Rustler model radio controlled electric car. We will also be using a Navman Jupiter 30 GPS receiver to provide our navigational solution. A Navman Mobile Data Terminal will be used in order to receive data from the receiver and convert the latitude and longitude readings into a degree that spans from 0 to 359. Two Arduino microcontroller boards will also be utilized for vehicle operation. Additionally, we will be using 2 sonar sensors to detect objects that may be in the vehicles path of travel. A simple obstacle avoidance algorithm will be used to navigate around objects which may be in that path.

In regards to the functionality of our vehicle, the GPS receiver will constantly be calculating our vehicle's position and sending a string of data to the dumb terminal. The terminal will then spit out the next desired heading to Arduino 1. Arduino1 will be the master. This Arduino will be responsible for overseeing the execution of all vehicle functions and RS-232 communication with the dumb-terminal. It will be in charge of executing vehicle turn commands it receives from the mobile data terminal by means of outputting a pulse-width-modulation (PWM) signal to the vehicle's steering servo. In addition to all of this, it will also take on the task of sending a digital-pin signal to the slave Arduino for the execution of the throttle and stop commands. Arduino 2 will be the slave. Its jobs will include executing throttle commands, holding the center steering position, continuously monitoring the sonar sensors for obstacles, and overriding the system to avoid an obstacle if one is detected.

The intended target consumer of this product would ideally be everyone, but we realize that implementing this in commercial motor vehicles is a long ways off. More immediate uses of this product would most likely be for military purposes such as surveillance or as a weapons delivery device.

Software Description

Jupiter-to-MDT-to-Buffer

The Navman Jupiter 30 GPS receiver kit will be transmitting a NMEA GPRMS string to the Navman Mobile Data Terminal once a second at 9600 baud. The GPRMS string contains the validity of the GPS position, latitude, longitude, date, and time among other things. The MDT is responsible for parsing this GPRMS string and extracting these values for use in the navigational algorithm.

Yet the MDT will only begin the navigational algorithm and the parsing of the GPRMS string if and only if a designated "go" button has been pressed. When this button is pressed a transmission to the master Arduino controller occurs, which signals the motor to drive the vehicle forward. During this time the Jupiter should be able to obtain a fairly accurate heading. This initial heading is what we use for our first **expected heading**. After three seconds navigational data used for debugging appears on the screen, and a navigational solution is considered.

Assuming the vehicle is still on track and within the initial expected heading the MDT begins to calculate the navigational solution to the first waypoint. This is done using the properties of arctangent and some ingenuity to figure a new heading, which is a degree of difference from north, based upon the difference of latitude (rise) and longitude (run) between the waypoint and the current position of the vehicle. Knowing the current heading and the newly desired heading we can calculate a desired turn we wish the vehicle to make. This desired turn we wish to make is transmitted to the Arduino as a value between 0-360. This angle is the (clockwise) angle away from our current heading and will be translated into a degree of turn to the left or right by the software on the master Arduino.

The MDT assumes the vehicle has properly executed the last turn it was instructed to turn, but it sets an updated **expected** heading based upon the heading reported by the GPS receiver and the turn the vehicle has just been commanded to make. This is used for the next navigational calculation, just in case the vehicle has been thrown off course by either poor steering, terrain or some other factor as well as to check the validity of the reported heading by the Jupiter for the next iteration.

A navigational solution only occurs when **new** GPS data arrives, which is set as a parameter of the Jupiter GPS receiver to be once per second. A variable **count** is used to limit navigational solutions to occur at most once every other second. This is to insure that the receiver has had ample time and covered a sufficient distance to be able to accurately report a heading. Without this limit, there would be a greater possibility of a false heading being reported to the receiver.

One of the most important aspects of the navigational algorithm is the accepted range of heading. Not only are navigation reports limited to at most, once every two seconds, but they are limited by the fact that they can only be calculated if and only if the heading being reported by the GPS receiver is within a tolerated threshold of the **expected** heading set by the last turn. This acceptable heading is initially limited to be +/- 45 degrees. So if the vehicle is off course because of naturally occurring error and deviation in the GPS system, we are sure to throw out the navigational solution which would be based upon that error. Yet, to account for the case that the vehicle truly is off course from our expected heading, we broaden the acceptable error in the heading by +/- 20 degrees for every successive report by the receiver until the vehicle's heading lies within an acceptable threshold. At a worst case scenario the heading would be +/- 180 degrees away from our expected heading, and it would take about 7 seconds to 'believe' the receiver and recalculate a navigational solution. In this way we limit the number of inaccurate or poor heading reports to the MDT by the Jupiter receiver.

With all iterations, before we even consider heading, we consider position. Unlike heading, position is accurate regardless of movement. This is because the Jupiter requires a vehicle to move, and by taking averages in position samples it is able to acquire a fairly accurate heading. But if the vehicle is not moving, naturally occurring 'jumps' or inaccuracies in position cause errors and likewise 'jumps' in heading. We consider position first because we would like to know if the vehicle has reached the waypoint destination. We check to see if the vehicle has reached its destination by setting a threshold in difference of latitude and longitude. In this way an effective positional 'box' is formed, and if the vehicle falls within this 'box' it has reached the waypoint. If the vehicle does indeed fall upon the current waypoint, a built in buzzer is sounded, and the waypoint counter for the waypoint array is incremented. If this counter reaches the end of the waypoint array, we make sure to transmit to the Arduino a command to **stop** the vehicle. We are also sure to reset all variables, so the vehicle is ready to go again.

Arduino1: MASTER

Arduino1 is the master. It is primarily responsible for overseeing the execution of all vehicle functions and RS-232 communication with the dumb-terminal. Directly, Arduino1 is responsible for executing vehicle turn commands it receives from the mobile data terminal by means of outputting a pulse-width-modulation (PWM) signal to the vehicle's steering servo. Indirectly, it is responsible for execution of throttle and stop commands by sending a digital-pin signal to the slave Arduino. It is also responsible for the initialization of the MaxSonar EZ1 sensors.

Initialization:

Upon startup, the master Arduino will first initialize the state of the MaxSonar EZ1 sensors. It will do this by holding a digital pin (that is branched to both sensors) high for 20us. After this is done, the sonar sensors will begin pinging on their own every 50ms and reporting analog signals to the slave Arduino with no further input required. Next, the master Arduino will execute a steering diagnostic test that includes turning the wheels

100% left and then 100% right. (The re-center diagnostic test will be covered by the slave Arduino.) It then initializes throttle to FALSE.

RS-232 communication:

Arduino1 is responsible for communication with the dumb-terminal in the RS-232 standard so that it may accurately receive navigational commands. The RS-232 data signal is sent from the dumb-terminal to the Max3323E buffer. This buffer scales down the RS-232 voltages (which range from about 15-3v) to TTL voltages (5-0v). The master Arduino then reads the data out of this buffer in the following manner:

RS-232 protocol consists of a start bit (0), followed by 1 byte of data (8bits), and then a stop bit (1). The stop bit (1) is held high until the next transmission. When Arduino1 calls SWread( ), it initially enters a while loop that loops until it finds a 0 start bit signaling that a byte is ready to be read. Once the start bit is found, it delays for 100us and then enters a "for" loop that will loop 8 times (for one 8-bit byte). Each time a bit is read from the buffer, it delays for 84us, which is the period consistent with the 9600 baud rate setting of the dumb-terminal. Each bit extracted is then shifted proportionally and appended to a byte variable. After all 8 bits have been read, the function delays for two 84us periods (waiting for the stop bit) and then returns the byte.

In our void loop function, this SWread( ) function is called in a while loop, appending each byte to a character array named "command" until it finds our chosen end command character 'D'.

Throttle:

The first command that the master Arduino should receive from the dumb-terminal is 'G' for "go". It will execute this command indirectly by setting digital-pin 6 to HIGH. This digital signal instructs the slave Arduino to begin moving the vehicle.

Turning:

Next are the turn commands. Turn commands will be streaming from the dumb-terminal on 2-to-7 second intervals. These turn commands are integers ranging from 0 to 359. Note that the dumb terminal's commands are all considered to be ***right*** turns. It is the master Arduino's job to take this "right turn", convert it into an appropriate left-or-right turn, and then execute the turn via a PWM signal to the vehicle's steering servo.

Upon receiving this turn command integer, the steering algorithm first determines if the turn should be a left or a right. Naturally, if the turn degree is greater than 180 degrees, it would be better to make a left turn. If the turn should be to the left, it converts the angle to a left turn angle by subtracting it from 360. After the turn direction is determined, the turn algorithm theory is as follows:

The magnitude of steering, or the angle at which the wheels will be cocked, is determined by a pulse width modulation signal. This signal is of a 20ms period, with a high-time ranging from 1-2ms. A 1ms high-time would correspond to a 100% right turn, a 1.5ms high-time to center, and a 2ms high-time to 100% left. The vehicle's turn is a

function of the PWM magnitude and the length of time for which this turn is held. After much testing, we determined that the smoothest turning across all angles is achieved as follows: For angles < 90 degrees, the turn time is held constant at the 90degree constant time and the PWM signal is varied by an amount proportional to the angle / 90. For angles > 90 degrees, the PWM is held constant at max and the turn time is varied by an amount proportional to the angle / 180. In the course of implementing these ideals, much floating-point arithmetic is required. Unfortunately, the Arduino is limited to 7kb of memory and the use of any floating point arithmetic is almost impossible. As such, we were forced to approximate this calculation with integers in order to save memory.

After the PWM and time are determined, the pulse width modulation signal is output by holding a digital pin high for the PWM time, then low for the remainder of the 20ms period. This period is repeated in the appropriate number of "for" loops for the turn angle time, resulting in the steering servo conducting the desired turn.

It should be noted that before a steering PWM is output, the master Arduino must first set the digital-pin 7 "steering Command" high to tell the slave Arduino to stop holding center. It then must change its digital-pin 11 "steering PWM" from an input to an output. The reason it is default as an input is so that it does not ground the linked PWM hold-center signal of the slave Arduino.

Arduino2: SLAVE

The jobs of the slave Arduino include executing throttle commands, holding the center steering position, continuously monitoring the sonar sensors for obstacles, and overriding the system to avoid an obstacle if one is detected.

Initialization:

Upon initialization, the slave Arduino will enter a "for" loop that waits for the master Arduino to set throttle to false. This is to account for the fact that when the system is powered up, the digital "throttle command" pin from the master Arduino is temporarily held high, which would cause the vehicle to lurch forward.

Obstacles:

In the slave Arduino's void loop function, the first thing checked is the Max Sonar EZ1 sensors. If an obstacle is detected, it will hold the obstacle override pin HIGH to signal to the master Arduino that it has taken control so that the master Arduino will reset its turn-PWM-pin to an "input". It will then execute the obstacle avoidance algorithm.

Throttle / Hold Center:

Next, the slave Arduino checks the throttle digital pin to determine if it should hold "throttle" or "stop". It also checks the steering digital pin to determine if the master Arduino is currently executing a turn. If no turn is currently in execution, the slave Arduino holds the center steering position to prevent the vehicle's wheels from swaying off course due to minor bumps and deviations in the terrain.

The combination of these two functions results in one of 4 possible states: Throttle while holding center, Throttle while allowing the master to turn, Stop while holding center, and Stop while allowing the master to turn. As such, the situation arises where this Arduino is required to execute two PWM signals at the same time. To do this, we hold two digital pins high, and then lower them at their respective correct times, making sure to encompass both signals within the 20ms period. The same holds true for the obstacle avoidance algorithm.

Obstacle Avoidance Algorithm:

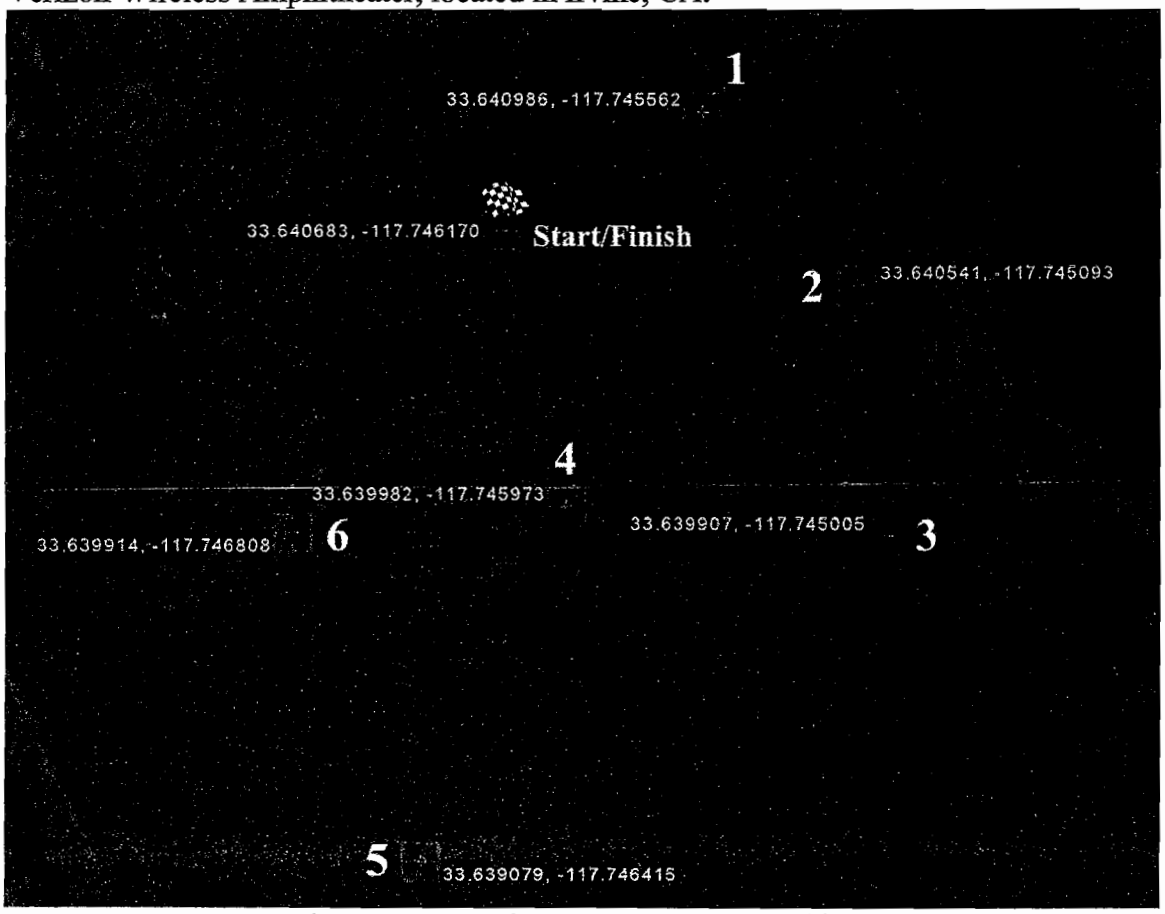
In order to determine when an obstacle is truly in the path of the vehicle, the algorithm will require that a certain number of successive sonar pings from BOTH sensors register below the set threshold. This is to account for the outlier deviations inherent to ultrasonic pings. Once an obstacle is detected, it will determine if the obstacle is right-of-center or left-of-center based on which of the sensors is outputting the lower analog value. After this, the Arduino will output a throttle / turn PWM combo in the direction opposite of the obstacle for every 20ms period in which an obstacle is still detected. Once an obstacle is no longer detected, the navigational algorithm in the dumb-terminal will regain control and correct the vehicle's direction to point it back on course.

Unfortunately, while testing the sensors we ran into the problem of a great instability in the sensors ability to detect objects in its path at random times. Through much analysis and testing of this problem, we found that it was mainly the time of day, temperature, barometric pressure, and wind factor that was affecting the functionality of the sensors. While we were still able to get our car to successfully navigate around obstacles in its path, we feel that the obstacle avoidance capabilities of our vehicle need further research to become acceptably stable in all temperature scenarios for full time implementation.

System Test Plan

The test plan for the GNAV's navigational capabilities can be conducted in a simple, straightforward, and intuitive manner. A course of successive waypoints should be programmed into the MDT, and the navigational algorithm should attempt to traverse the course sequentially. Assuming that the course has sufficient variety and tests all aspects of the navigational algorithm (i.e. the vehicle should be made to make turns from 0 to nearing 180 degrees to the left or right), if the vehicle is able to consistently complete this course then it can be readily assumed that the navigational algorithm is correct and functional.

To accomplish this evaluation one would need a clear, safe, open environment in which to conduct the test. For our testing of the vehicle we chose the parking lot for the Verizon Wireless Amphitheater, located in Irvine, CA.



As one can see the course contains a variety of turns which we feel represent a fairly complete assessment of vehicle functionality. At the **start** one must allow the GPS receiver to sufficiently acquire the GPS almanac and signal and determine its position, as well as allow it to acquire as many satellites as possible. This can take upwards of 12 minutes if it is the first time the receiver has been powered for the day. For our testing we enlisted the support of the Differential GPS system known as WAAS for greater GPS

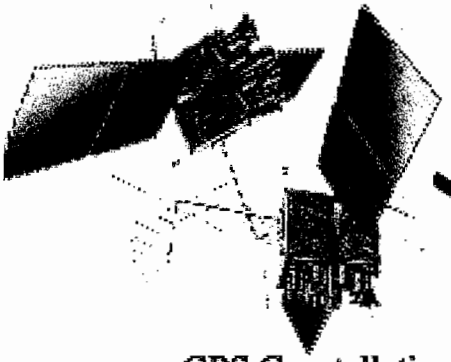
accuracy. WAAS capability was enabled on the receiver prior to testing, and substantially reduces the inherent error in the GPS system.

When the button assigned to transmit the 'go' signal to the Arduino controllers has been pressed, the vehicle should proceed to the first waypoint. We chose to limit the time between the start of the vehicle's travel to first navigation to three seconds, in order to give the receiver sufficient time and distance to acquire a good initial heading. Because of inherent variations in reported position and heading by the GPS receiver, the navigational algorithm will appear to 'snake' its way toward the waypoint. If the threshold for the allowable heading is appropriate, this 'snaking' should be minimized. Should a (incorrect) position report that the vehicle has somehow traveled astray, the navigational algorithm would correct for this unquestioningly if this threshold was not considered. It is to safeguard against this inherent inaccuracy that the threshold for acceptable heading be implemented.

It is also important that a reasonable threshold be set for determining if the vehicle has come close enough to the waypoint to have that waypoint be considered as being 'reached.' If one picks a threshold that is too low, the vehicle could conceivably travel through the waypoint between navigational considerations, miss the waypoint, and have to circle back in order for the algorithm to consider it as 'reached.' If one picks a threshold that is too large, the vehicle would make a very sloppy run of the course, and in the spirit of GPS, accuracy is paramount for this project.

The vehicle should make its way between waypoints and upon reaching the final waypoint, come to a stop and reinitialize itself for another run. For our testing we chose to have the finish line be the origin of the course. Upon the final build of our navigational and steering algorithms it should be noted that the vehicle was tested a half dozen times with a perfect track record.

System Level Data Path

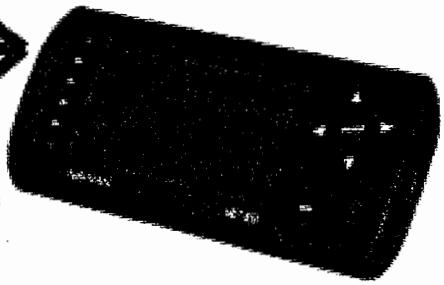


GPS Constellation

"Jupiter 30" GPS Receiver

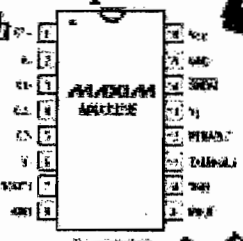
NMEA protocol's GPRMS Stream

The dumb terminal is responsible for parsing the GPRMS stream, navigational calculations, and sending vehicle movement / steering commands to the master arduino.



RS-232 Command

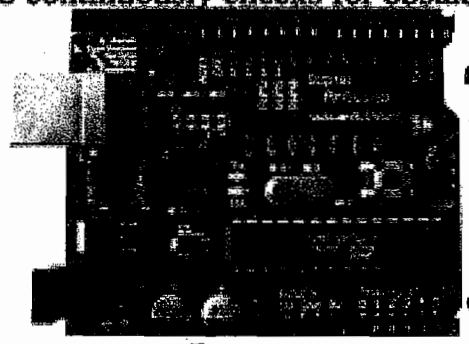
MAX3323E-
A buffer specially designed for RS-232 protocal. It takes in the RS-232 data (which ranges from 15-3v) and it scales it down to TTL values (between 5-0v) for the Arduino to read.



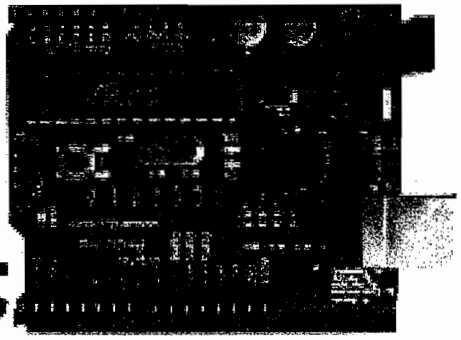
Arduino: MASTER
This arduino is in charge of rs-232 communication and implementing most vehicle operations. It directly outputs PWM turn signals to the servo and it sends digital throttle commands to the salve arduino.

RS-232 Voltage Scaled

Arduino: SLAVE
This arduino transmits PWM's to hold throttle and center steering based off of the master's commands. It also continuously checks for obstacles.



Boolean Digital Com



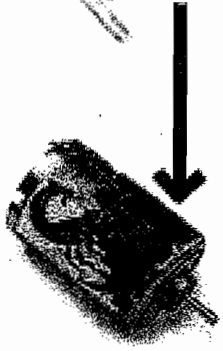
Throttle PWM

IPC Digital Proportional Electronic Speed Control

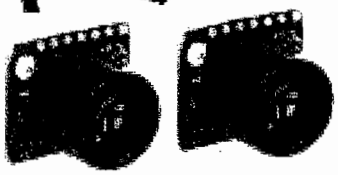
Hold Center PWM and Obstacle Override

Turn PWM

Analog



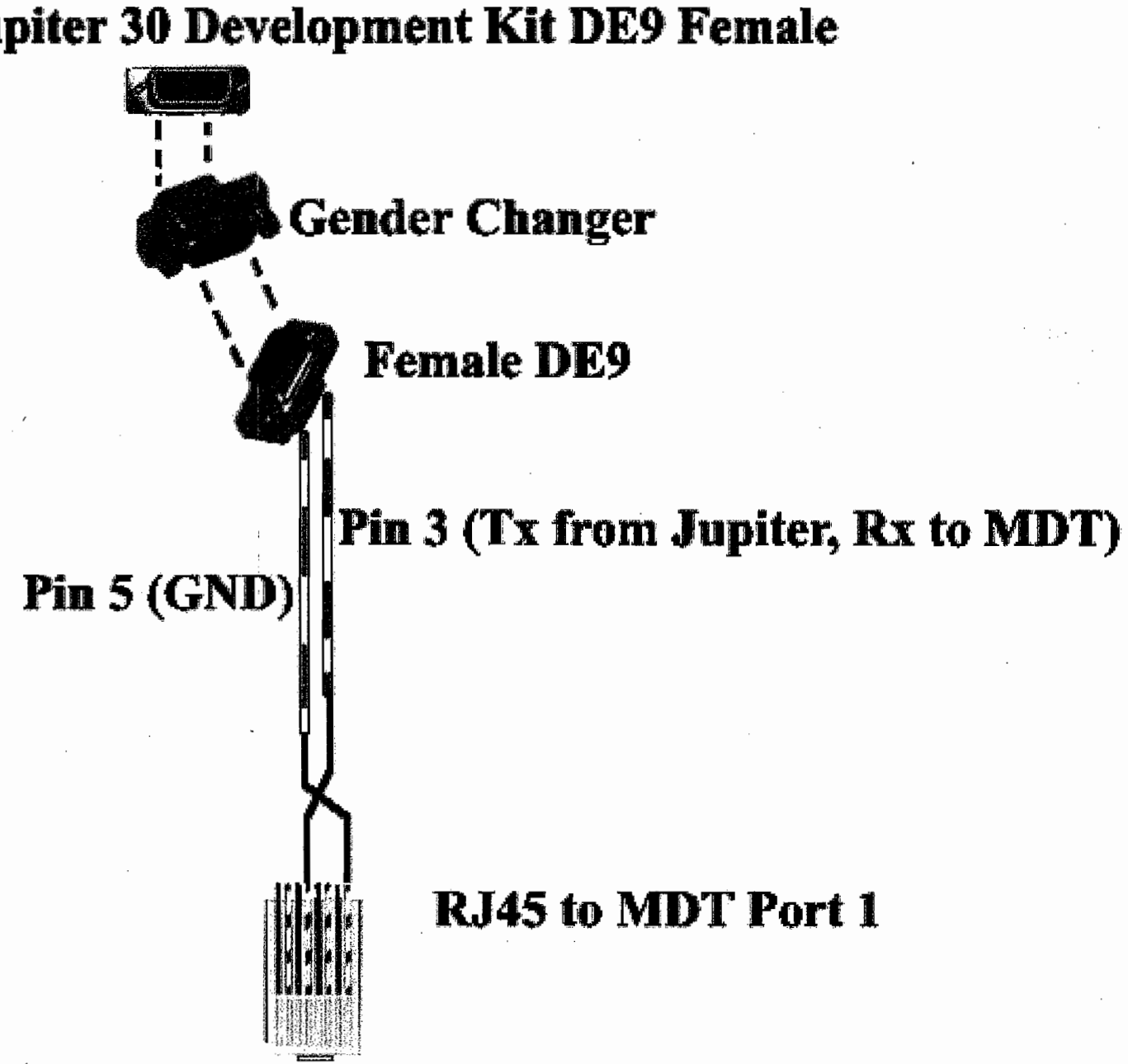
Electric Motor



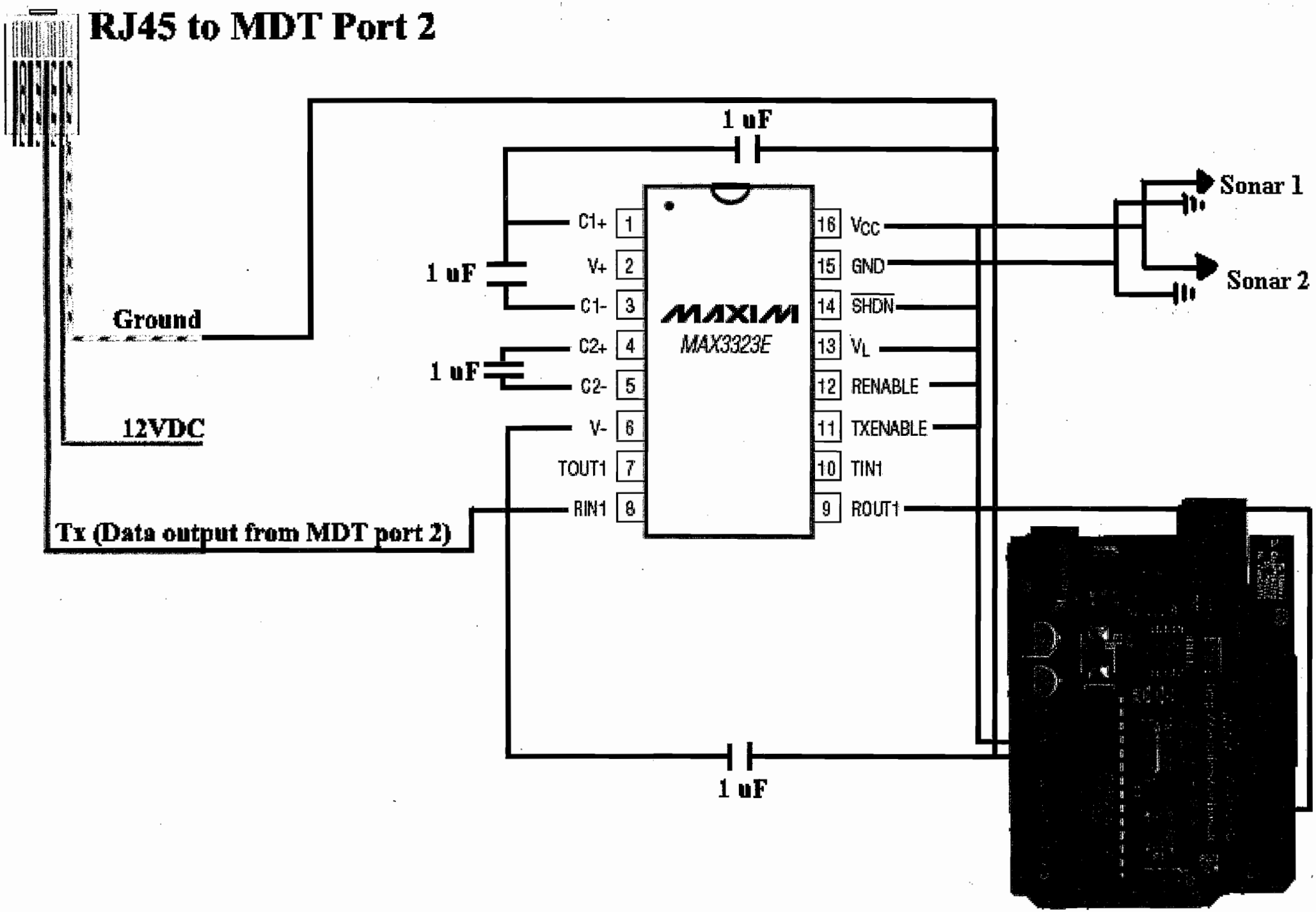
MAX SONAR EZ1
Obstacle Detection via ultrasonic ping

Traxxas High Torque Steering Servo

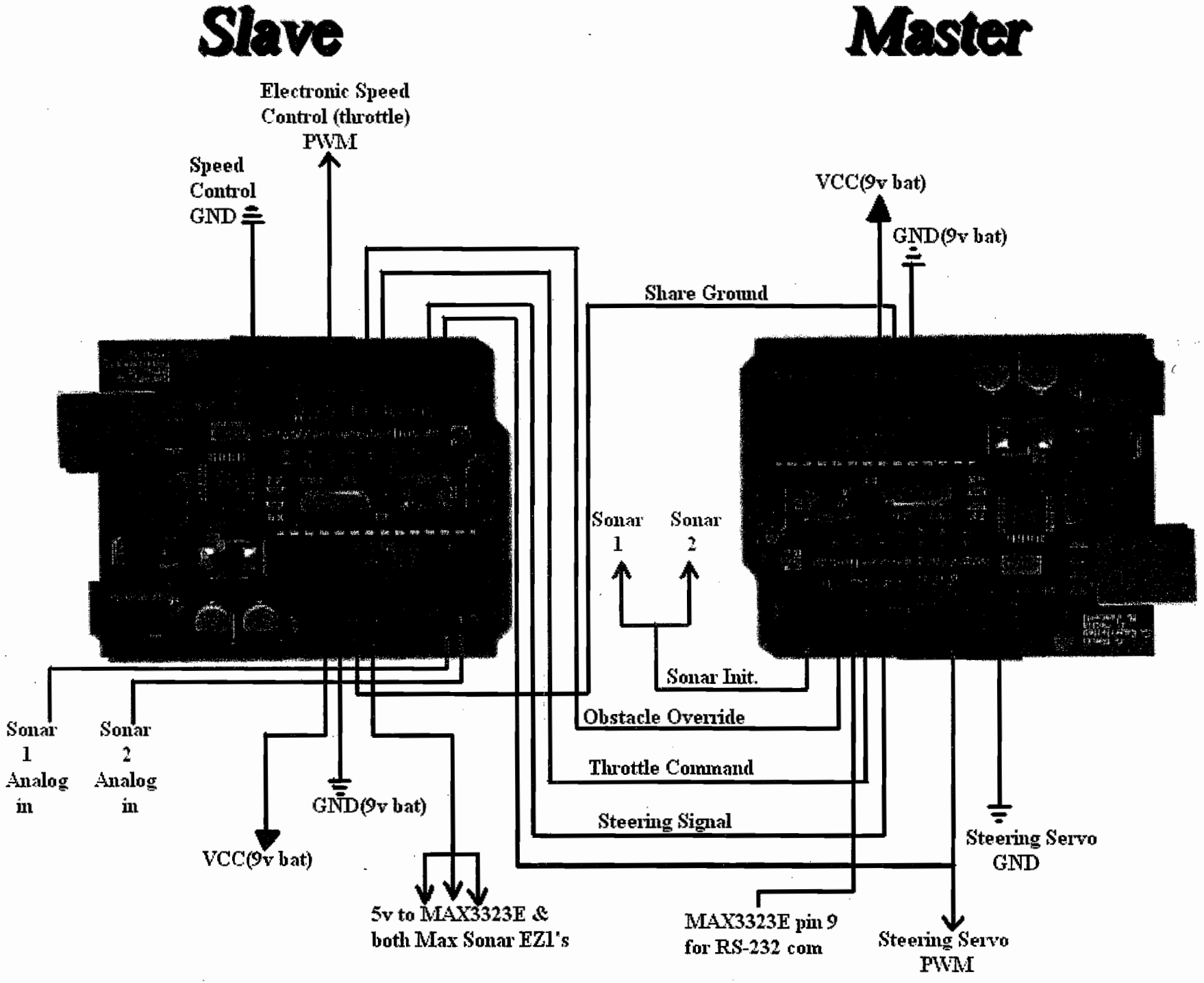
Jupiter 30 (DE9) to MDT 850 (RJ45) Wiring



MDT 850 (RJ45) to MAX3323E (RS232 Buffer) to Arduino (Master)



ARDUINOS


Slave
Master
Electronic Speed Control (throttle) PWM
Speed Control GND
VCC(9v bat)
GND(9v bat)
Share Ground
Sonar 1
Sonar 2
Sonar Init.
Obstacle Override
Throttle Command
Steering Signal
Steering Servo GND
Steering Servo PWM
Sonar 1 Analog in
Sonar 2 Analog in
GND(9v bat)
VCC(9v bat)
5v to MAX3323E & both Max Sonar EZ1's
MAX3323E pin 9 for RS-232 com

Navigational Code:

```
/*************************************************************
* Copyright (c) 2007, Navman
*************************************************************
*
* Jason Meachum, January 2007
* Demonstrates:
* GPS NMEA parsing functionality
* Navigational Heading calculation
*
*************************************************************/
/********************  Includes ****************************/
#include <stdio.h>
#include <string.h>
#include <math.h>

#include "system.h"
#include "mdt800.h"
#include "Nmea.h"
#include "app.h"
#include "appMain.h"

/********************** Defines ****************************/
typedef struct {
  BYTE boGPSFix;
  BYTE UTCTime[20];
  BYTE UTCDate[20];
  double Lat;              //Latitude
  double Lon;              //Longitude
  double heading;
  BYTE boUpdate;
  BYTE boGPRMCDecoded;
  BYTE str[100];
  WORD cntDec;
  WORD cntTmr;

  BOOL clearStatus;
} GPS_DATA;                        //Holds GPS data

#define GPS_PORT 0         //MDT port 1
#define GPS_BAUD 9600      //GPS operational baud rate
#define SERIAL_PORT 1      //MDT port 2 (communicates with Arduino)
#define SERIAL_BAUD 9600 //Baud rate for communication with Arduino
#define BUF_SZ 250         //Buffer size
/******************** Global Vars ************************/
//Navigation (7 waypoints)
int currentWaypoint = 0;   //keeps track of current waypoint
BOOL firstRun = true;              //keeps track of the travel to the first
waypoint
int expectedHeading;               //tracks the expected heading of the vehicle
                                          //after it his completed a turn
int count = 1;                           //counts the cycles between navigation
reports
int halfRange = 45;                //1/2 the threshold of acceptable heading in
degrees
double rise;                       //difference in latitude of the waypoint and
current position
double run;                              //difference in longitude of waypoint
and the current position
int turn_command;                  //angle of turn to be transmitted to the
arduino
```

```
//Pi
double Pi = 3.14159265358979;

//General
BYTE KeyPressed;                    //holds value of which key has been pressed
BYTE Volume = 255;                  //system volume for piezzo buzzer
BYTE Backlight_Level = 160;         //intensity of the MDT backlight

BOOL go = false;                    //algorithm operates only if go = true

//GPS
static GPS_DATA GPSData;   //struct to hold the GPS data
static BYTE GPSCBKTimer;   //countback timer to detect if data stream drops or
fails
static char NMEABuffer[200];      //buffer for NMEA data string
static char SerialBuffer[100];    //buffer for the serial transmission to the
Arduino

WORD datalen, cnt;

//Serial
BYTE TempChar;                        //input
static WORD _len;          //for length of msg output
static BYTE _str[BUF_SZ]; //string for msg output
static BYTE _buf[40];        //buffer for output

/********************* Forward Declarations ***************/
void GPSvalues(void);
void decodeNMEA(void);
static void Write_Serial_String(BYTE *, WORD);
double abs(double);
void navigate(double, double);
/*******************  Functions *************************/
#if !defined(CROSS_COMPILE)
void main(void)
{
      MDT_Init(appMain,NULL);
}
#endif

//Takes in the current Latitude and Longitude
//and calculates a corrected heading to the next waypoint
//and transmits a turn command to the Arduino.
//An expected heading is determined to use as a standard
//for the next heading calculation as well.
void navigate(double currentLat, double currentLon)
{
      double raw_angle;   //stores result of the arctan of rise/run
      double angle_from_north = 0;      //stores the computed heading

      //BEGIN ALGORITHM
      rise = currentLat - GPSData.Lat; //calculates "rise"
      run = currentLon - GPSData.Lon;         //calculates "run"

      if(run == 0){              //protects from division by zero
                  if (rise > 0){       //directly north
                        angle_from_north = 0;
                  }
                  else{                //directly south
                        angle_from_north = 180;
                  }
      }
```

```
	else if (run > 0){  //if the waypoint is east of current position
		if(rise != 0){
			raw_angle = atan(rise/run); //RADIANS
			raw_angle = raw_angle * 180/Pi; //CONVERT TO DEGREES!
			angle_from_north = 90 - raw_angle;
		}
		else{			//directly east
			angle_from_north = 90;
		}
	}
	else if (run < 0){  //if the waypoint is west of the current position
		if (rise != 0){
			raw_angle = atan(rise/run);  //RADIANS
			raw_angle = raw_angle * 180/Pi; //CONVERT TO DEGREES!
			angle_from_north = 270 - raw_angle;
		}
		else {			//directly west
			angle_from_north = 270;
		}
	}

	//If the heading of the vehicle is greater than the calculated heading
	//of the next waypoint then
	if(GPSData.heading > angle_from_north){ //LEFT turns
		turn_command = 360 - GPSData.heading + angle_from_north;
	}
	else{  //RIGHT turns
		turn_command = angle_from_north - GPSData.heading;
	}

	//Transmit turn command
	_len = 0;
	_len += (WORD)sprintf((BYTE *)&_str[_len], "%iD", turn_command);
	Write_Serial_String(_str, _len);

	//set the expected heading of the next navigation
	//we will use this expected heading as a standard
	//to judge incoming heading readings
	expectedHeading = GPSData.heading + turn_command;
	if(expectedHeading >= 360) //handle overflow of degrees from north
		expectedHeading = expectedHeading - 360;
}

//returns the absolute value of an input double
double abs(double value)
{
	if (value > 0)
		return value;
	else
		return -value;
}

//initializes the data terminal
void appMainInit(void)
{
    /* Start up everything */
    Display_Init();		//initializes LCD
    Keypad_Init();		//initializes Keypad on faceplate
    Kbd_Init();			//(not used) Initializes keyboard PS/2
    Basicio_Init();		//initializes basic input/output functions
    I2C_Init();			//initialization of the I2C module
    Serial_Init();		//initializes serial ports
```

```
      //configure the GPS communications port
    Serial_Config(GPS_PORT, 8, 1, 0, (long)GPS_BAUD);   //uses port '1' for
data TX/RX
    Serial_Set_RX_Buff(GPS_PORT, SerialBuffer, sizeof(SerialBuffer));
      //configure the serial communications port
      Serial_Config(SERIAL_PORT, 8, 1, 0, (long)SERIAL_BAUD);  //uses ISP port
for IO
      Serial_Set_TX_Buff(1, (BYTE *)_buf, sizeof(_buf));

    Timer_Init(); //initialize the internal timer
    enable;

    load_display_data();    //Load fonts and images into ROM

      //set display properties (contrast/brightness/font)
    Display_Set(16, 16);
    Display_Set(12, 00);
    Display_Set(13, 11);
    Display_Set(14, 20);
    Display_Set(15, 31);
    Basicio_Backlight(Backlight_Level);
    Display_Font(FONT_NOKIA);

   memset((BYTE *) &GPSData, 0x00, sizeof(GPS_DATA)); //allocate memory for
structure

    return;
}

void appMain(void)
{
      WORD startTime = 0; //initialize counter for sentinal

          appMainInit();  //initialize the MDT

          Display_Cls();  //clears the screen
          Display_TextCol(LCD_Black, LCD_White);     //set color properties of
text

      Display_Font(0);    //set Font style
      Display_Pos(57, 1); //print to the screen
      printf("GPS Receiver - Com 1");

      while(TRUE)   //main loop of the system (always running)
      {
            //if the button assigned to 'go' has been pressed
            //and three seconds has passed, begin accepting GPS
            //signals and calculating navigational solution
            if((go)&&((signed)(TimeElapsed_Sec(startTime)) >= 3)){
                  GPSData.clearStatus = false;

                  //Gather GPS Data and calculate navigational solution once
a second!
                  decodeNMEA();

                  //if GPS data is inturupted
                  if(TimeElapsed_mS(GPSData.cntTmr) > 5000){
                        Display_Box(0, 10, 0, 159, 69, 1); //clear the screen
                        Display_Pos(10, 10);
                        printf("Invalid signal.");
                        Display_Pos(10, 20);
                        printf("Check connection and baud rate.");
                        GPSData.clearStatus = false;
```

```
                }
            }

        // If key pressed then perform operation
        if(Keypad_Query()) {
                KeyPressed = Keypad_Get();
                if (KeyPressed & 0x10) {  // Only use key down events
                      KeyPressed &= 0x0F;
                      switch (KeyPressed) {
                            case 1:
                                        go = true; //GO!
                                        //SIGNAL car to start with
transmission to Arduino
                                        _len = 0;
                                        _len += (WORD)sprintf((BYTE
*)&_str[_len], "GD");
                                        Write_Serial_String(_str, _len);

                                        //but WAIT 3 seconds to acquire
accurate heading.
                                        startTime =
Timer_Tick_Read_Sec();
                                        //startTime holds timer value
used as a standard
                                        //for use later (3 seconds
later).
                                  break;

                                  case 2:
                                        _len = 0; //Debug value to
demonstrate steering.
                                        _len += (WORD)sprintf((BYTE
*)&_str[_len], "45D");
                                        Write_Serial_String(_str, _len);
                                  break;

                                  case 3:
                                        _len = 0; //Debug value to
demonstrate steering.
                                        _len += (WORD)sprintf((BYTE
*)&_str[_len], "180D");
                                        Write_Serial_String(_str, _len);
                                  break;

                                  case 4:
                                        go = false; //STOP!
                                        //SIGNAL car to stop with a
transmission
                                        _len = 0;
                                        _len += (WORD)sprintf((BYTE
*)&_str[_len], "SD");
                                        Write_Serial_String(_str, _len);
                                  break;

                                  case 15:    // "Power button", cycles
backlight
                                      if (Backlight_Level == 0)
                                            Backlight_Level = 160;
                                      else
                                            Backlight_Level = 0;
                                     Basicio_Backlight(Backlight_Level);
                                  break;
                            }
```

```
waypointLon[4] = -117.746415;
waypointLon[5] = -117.746808;
waypointLon[6] = -117.746170;

                    //Do we have a fix?
                    if(GPS_Get_Valid())
                          GPSData.boGPSFix = true;
                    else
                          GPSData.boGPSFix = false;

                    GPSData.Lat = GPS_Get_Lat();  //sets latitude
                    GPSData.Lon = GPS_Get_Lon();  //sets longitude

                    GPSData.heading = (double) GPS_Get_Dir(); //sets current
heading
                    //on the first run our expectedHeading IS our reported
heading.
                    //after which our expectedHeading is based upon what is
expected
                    //after making turns within the navigational algorithm
                    if(firstRun == true)
                    {
                          expectedHeading = GPSData.heading;
                          firstRun = false;
                    }

                    //------DISPLAY----------------------
                    //Primarily for debugging, these values
                    //print to the screen of the MDT
                    if(GPSData.boGPSFix == true){

                    Display_Pos(10, 20);
                    printf("Valid Position Fix");

                    //LAT/LONG
                    Display_Pos(10, 30);
                    printf("Lat: %f", GPSData.Lat);
                    Display_Pos(10, 40);
                    printf("Lon: %f", GPSData.Lon);

                    //HEADING
                    Display_Pos(10, 50);
                    printf("Heading: %.1f", GPSData.heading);

                    Display_Pos(10, 60);
                    printf("LatDif: %.6f   Turn: %i", abs(rise), turn_command);
                    Display_Pos(10, 70);
                    printf("LonDif: %.6f   WP#: %i", abs(run),
currentWaypoint);
                    //-----------------------------------

                    //GPS DATA ACQUIRED... NOW TO NAVIGATE!
                    //set the thresholds of our expected heading
                    //in degrees from north.
                    min = expectedHeading - halfRange;
                    if(min < 0){min = min + 360;}
                    max = expectedHeading + halfRange;
                    if(max >= 360){max = max - 360;}

                    //find differences of latitude and longitude
                    //of the next waypoint and vehicles current position
                    rise = waypointLat[currentWaypoint] - GPSData.Lat;
                    run = waypointLon[currentWaypoint] - GPSData.Lon;
```

```
                    //Check if the waypoint has been reached!
                    if ((abs(run)< thresh)&&(abs(rise)< thresh)){//WAYPOINT
REACHED!
                          Basicio_Buzzer(2000, 250, 255); //sound the buzzer
                          currentWaypoint++;                    //increment
the waypoint

                          //If it's reached all waypoints END sim
                          //and reset variables for another run!
                          if(currentWaypoint >= 1){
                                go = false;
                                firstRun = true;
                                _len = 0;
                                _len += (WORD)sprintf((BYTE *)&_str[_len],
"SD");
                                Write_Serial_String(_str, _len);
                                currentWaypoint = 0;
                          }
                    }

                    //count is used to make sure the vehicle has a chance to
accurately
                    //acquire a heading between reports... because the reciever
is transmitting
                    //once a second, this means that we limit the number of
navigational corrections
                    //to once every two seconds.
                    //We then check to see if the heading that is being
reported is within our expected
                    //threshold of heading, and if so calculate a new
navigational solution for
                    //the next turn.
                    //If we have fallen outside the range of expected heading
we have either been drawn
                    //off course, or the GPS receiver is reporting an
inaccurate heading.  We hold out
                    //another second and broaden the acceptable heading
threshold for the next heading
                    //report.  After many seconds the threshold of acceptable
heading is ~360 degrees
                    //and we give the receiver the benefit of doubt and accept
its heading report
                    //over what we expect as our heading, meaning we have been
seriously thrown off course
                    if( ( (count >= 1)&&(((min <=
GPSData.heading)&&(GPSData.heading <= max))  || (  (min > max)&&( (min <=
GPSData.heading)||(GPSData.heading <= max)   )  ))   )){
                          navigate(waypointLat[currentWaypoint],
waypointLon[currentWaypoint]);
                          halfRange = 45; //reset
                          count = 0;
                    }
                    else{
                          if (count>= 1){halfRange = halfRange + 20;}
//increment acceptable heading
                          count++;
                    }
              }
              else{
                    Display_Box(1, 20, 0, 150, 9, 1);
                    Display_Pos(10, 20);
                    printf("Invalid Fix");
```

```
            }
}

//write to the output buffer and transmit to the Arduino
void Write_Serial_String(BYTE *str, WORD len)
{
      #ifdef CROSS_COMPILE
            WORD temp = 0;

            temp += len;

            SYSERROR(str);
      #else
            while (Serial_TX_Query(SERIAL_PORT)); //nothing
                  Serial_Write_Buffer(SERIAL_PORT, len, str);
      #endif
}
```

Arduino: MASTER

```
//***** Arduino: Master ****************************
//EECS 129B Senior Project
//Project: GNAV
//Chris Abernethy
//*************************************************
#include <stdio.h>
#include <stdlib.h>

#define bit9600Delay 84
#define startDelay 100
byte incByte;

//***** Digital Pins ******
int rx = 5;
int steeringPWM = 11;
int steeringCommand = 7;
int throttleCommand = 6;
int sonarSignal = 2;
int obstacleSignal = 4;

//***** Steering PWM Constants *****
int left = 2000;
int right = 1000;
int center = 1500;
int leftCorrection = 1345;
int steeringTimeConstant = 2;

int i;

char command[4];       //For RS-232 commands

void setup()
{
 pinMode(steeringCommand, OUTPUT);
 pinMode(throttleCommand, OUTPUT);
 pinMode(sonarSignal, OUTPUT);
 pinMode(rx, INPUT);
 pinMode(obstacleSignal, INPUT);

 //***** Initialize the sonar sensors ******
 digitalWrite(sonarSignal, HIGH);
 delayMicroseconds(20);
 digitalWrite(sonarSignal, LOW);
```

```
 //***** Run diagnostic steering tests ******
 digitalWrite(steeringCommand, HIGH);
 delayMicroseconds(20000);
 pinMode(steeringPWM, OUTPUT);
 for(i=0; i<250; i++){
     digitalWrite(steeringPWM,HIGH);
     delayMicroseconds(left);
     digitalWrite(steeringPWM, LOW);
     delayMicroseconds(20000-left);
     }
  for(i=0; i<250; i++){
     digitalWrite(steeringPWM,HIGH);
     delayMicroseconds(right);
     digitalWrite(steeringPWM, LOW);
     delayMicroseconds(20000-right);
     }
  pinMode(steeringPWM, INPUT);
  digitalWrite(steeringCommand, LOW);

 throttle(false);                //Set throttle to false
 delay(2000);

}

void loop()
{
//******* Get command from the MDT ************
    for(i=0; i<3; i++)
     {command[i] = '\0';}     //Clearing the command string

    i=0;
    Serial.println("Attempting to read: ");
    while(true)                  //Reading the RS-232 command
     {  incByte = SWread();
      if((char)incByte == 'D'){break;}    //D is our enD command character
      command[i] = incByte;
      i++;
     }

//****** Execute Command **********************
   if(command[0] == 'G')
    {  throttle(true);  }          //GO
   else if(command[0] == 'S')       //STOP
    {  throttle(false); }
   else
    {  turnVehicle(atoi(command));    }    //TURN X degrees
```

```
}

//*** Serial RS-232 Communication ****
int SWread()
{
 byte val = 0;
 while (digitalRead(rx));                //Wait for start bit
 if (digitalRead(rx) == LOW) {           //get rid of the start bit
  delayMicroseconds(startDelay);
  for (int offset = 0; offset < 8; offset++) {
   delayMicroseconds(bit9600Delay);      //Read 8 bits into a byte variable
   val |= digitalRead(rx) << offset;
  }
  delayMicroseconds(bit9600Delay*2);     //wait for stop bit + extra
  return val;
  }
}

//**** Digital Throttle Control Signal *****
void throttle(boolean movement)
{
 if(movement)
  {digitalWrite(throttleCommand, HIGH);}   //Gas
 else
  {digitalWrite(throttleCommand, LOW);}    //Breaks
}

//**** Pulse Width Modulation(PWM) Vehicle Steering Algorithm *****
void turnVehicle(int turnAngle)
{
 int turnTime, PWM, reCenter;
 //***** Left Turns *********************************
 if(turnAngle > 180){
    turnAngle = 360-turnAngle;
    reCenter = leftCorrection;              //Correcting for poor left turn mechanical
advantage
  if(turnAngle < 90)
    {
     turnTime = 90*steeringTimeConstant;
     PWM = 1655 + (38*turnAngle)/10;        //MIN left = 1655 us HIGH;  PWM =
1655 + 345*(turnAngle/(double)90)
    }                                       //Approximated with int's to save memory (7kb limit on
Arduino)
  else
    {
     turnTime = turnAngle*steeringTimeConstant;
```

```
      PWM = left;                          //MAX left = 2000 us HIGH;
    }
  }
  //***** Right Turns ******************************
  else{
    reCenter = center;                     //Center = 1500 us HIGH;
    if(turnAngle < 90)
    {
     turnTime = 90*steeringTimeConstant;
     PWM = 1345 - (38*turnAngle)/10;       //MIN right = 1345 us HIGH
    }
    else
    {
     turnTime = turnAngle*steeringTimeConstant;
     PWM = right;                          //MAX right = 1000 us HIGH
    }
  }

  //******* Send the PWM signal *********************
  digitalWrite(steeringCommand, HIGH);     //Telling the slave arduino to stop
holding center
  delayMicroseconds(20000);                //Wait 1 period
  int obstacleFound = digitalRead(obstacleSignal); //Check if the SLAVE is avoiding an
obstacle, if so wait
  //pinMode(steeringPWM, OUTPUT);

  //*** PWM Signal Sent ****
  for(i=0; i<turnTime; i++){               //Execute turn
    while(obstacleFound = HIGH)
    {
     obstacleFound = digitalRead(obstacleSignal);
    }

    pinMode(steeringPWM, OUTPUT);

    digitalWrite(steeringPWM,HIGH);
    delayMicroseconds(PWM);
    digitalWrite(steeringPWM, LOW);
    delayMicroseconds(20000-PWM);

    pinMode(steeringPWM, INPUT);           //Sets the PWM pin to input so as to not
interfere
    }                                      //with the slave-arduino's pwm

  for(i=0; i<75; i++){                     //Re-center the wheels (overcorrects on left turns due
to  75
```

```
    while(obstacleFound = HIGH)
    {
     obstacleFound = digitalRead(obstacleSignal);
    }

    pinMode(steeringPWM, OUTPUT);

    digitalWrite(steeringPWM,HIGH);        // poor servo mechanical advantage on re-
centering after lefts).
    delayMicroseconds(reCenter);
    digitalWrite(steeringPWM, LOW);
    delayMicroseconds(20000-reCenter);

    pinMode(steeringPWM, INPUT);
    }
 digitalWrite(steeringCommand, LOW);       //Telling the slave arduino to resume
holding center
}
```

Arduino: SLAVE

```
//*********************************
//GNAV
//Arduino: SLAVE
//Chris Abernethy , 2007
//*********************************

//***** Digital Pins ********
int throttlePWM = 9;
int throttleControl = 6;
int steeringPWM = 2;
int steeringControl = 3;
int obstacleSignal = 7;

//**** PWM Constants ********
int forward = 1525;
int centered = 1500;
int stop = 1000;

//***** Global Variables ********
int speed, turning, object1Distance, object2Distance, i;
int objectPingCount = 0;
int turnCounter = 0;
int object1Total = 0;
int object2Total = 0;
```

```
int count = 2;
int threshold = 110;
int pingNo = 4;

//***** Analog Pins *******
int sonar1 = 3;
int sonar2 = 4;

void setup()
{
 pinMode(throttlePWM, OUTPUT);
 pinMode(obstacleSignal, OUTPUT);
 pinMode(throttleControl, INPUT);
 pinMode(steeringControl, INPUT);
 pinMode(sonar1, INPUT);
 pinMode(sonar2, INPUT);

 digitalWrite(obstacleSignal, LOW);

 speed = digitalRead(throttleControl);  //Wait for the Master Arduino to set throttle to
FALSE
 while(speed == HIGH)                  //This prevents the vehicle from briefly lurching
forward on startup.
 {
  speed = digitalRead(throttleControl);
 }
}

void loop()
{
    speed = digitalRead(throttleControl);
    turning = digitalRead(steeringControl);

    if(speed == HIGH)
    {
      //**** Check for obstacle ******************************
      count--;
      if(count == 0)                    //Only check the sonars every other cycle (20ms
cycle)
       {
        count=2;
        object1Distance = analogRead(sonar1);
        object2Distance = analogRead(sonar2);
        if((object1Distance < threshold) && (object2Distance < threshold))  //If both see
an object:
         {
```

```
          objectPingCount ++;                    //Increment the objectPinged variable
          object1Total+= object1Distance;        //Add distance to determine which
sonar is closer
          object2Total+= object2Distance;
        }
        else{
          objectPingCount = 0;
          object1Total = 0;
          object2Total = 0;
        }
      }

      if(objectPingCount >= pingNo)          //If both sonars ping an object 4 times in a
row, we
      {                                      //can be reasonably sure that there actually is an object
there.
        avoidObstacle();                     //run avoidance algorithm
        objectPingCount = 0;
        object1Total = 0;
        object2Total = 0;
        count = 2;
      }
      //*********************************************************
      else if(turning == LOW)                //If the arduino Master is not executing a turn
      {
        forwardMovementCentered();           //Send out forward and hold_center PWM
signals
      }
      else
      {
        forwardMovementTurning();            //If Master is turning, Send out Forward
PWM signal only
      }
    }
    else
    {
      if(turning == LOW)                     //These are the cases for no movement
      {
        stopMovementCentered();
      }
      else
      {
        stopMovementTurning();
      }
    }
}
```

```
//**** OBSTACLE AVOIDANCE ************************************
void avoidObstacle()
{
 digitalWrite(obstacleSignal, HIGH);          //Signal the master arduino
 delay(20);
 pinMode(steeringPWM, OUTPUT);
 //*** Dodge Left **********************
 if(object1Total > object2Total)              //If object is right of center
 {
  while(objectPingCount != 0)
  {
   evasionLeft();             //Send out one Left PWM
   turnCounter++;              //Keep track of how many turn PWM's sent
   count--;
   if(count == 0)             //Only check sonars every other cycle (20ms cycles)
   {
    count=2;
    object1Distance = analogRead(sonar1);
    object2Distance = analogRead(sonar2);
    if((object1Distance > threshold) && (object2Distance > threshold))   //If Both
DONT ping an obstacle
    {
     objectPingCount --;                    //Decrement the objectPingCount
    }
    else{objectPingCount=pingNo;}                 //This essentially turns until it pings
   }                                    //no obstacle for 4 pings in a row.
  }
  for(i=0; i<75; i++)                   //Re-center the wheeles
  {
   forwardMovementCentered();
  }

  count = 2;
  while(turnCounter!=0)                 //Counter turn back on course
  {
   count--;
   if(count==0)
   {
    count=2;
    object2Distance = analogRead(sonar2);    //Every other cycle, check if there is an
obstacle
   }                                //in the way of the turn.

   if(object2Distance > threshold)          //If there is no obstacle, counter turn
   {
```

```
    evasionRight();
    turnCounter--;
   }
   else{                          //If there still is an obstacle, turn away further
    evasionLeft();
    turnCounter++;
   }
  }

  for(i=0; i<75; i++)                  //Re-center wheeles
  {
   forwardMovementCentered();
  }
 }
 //***** Dodge Right
**********************************************************************
 else{
  while(objectPingCount != 0)        //Same thing as above except for the case where
  {                             //the obstacle is left of center.
   evasionRight();
   turnCounter++;
   count--;
   if(count == 0)
   {
    count=2;
    object1Distance = analogRead(sonar1);
    object2Distance = analogRead(sonar2);
    if((object1Distance > threshold) && (object2Distance > threshold))
    {
      objectPingCount --;
    }
    else{objectPingCount=pingNo;}
   }
  }
  for(i=0; i<75; i++)
  {
   forwardMovementCentered();
  }

  count = 2;
  while(turnCounter!=0)
  {
   count--;
   if(count==0)
   {
    count=2;
```

```
      object1Distance = analogRead(sonar1);
     }

     if(object1Distance > threshold)
     {
      evasionLeft();
      turnCounter--;
     }
     else{
      evasionRight();
      turnCounter++;
     }
    }

    for(i=0; i<75; i++)
    {
     forwardMovementCentered();
    }
   }
   pinMode(steeringPWM, INPUT);
   digitalWrite(obstacleSignal, LOW);
}

//******* PWM signal combination cases ************
void forwardMovementCentered()
{
    pinMode(steeringPWM, OUTPUT);
                          //Outputing 2 PWM's separately in the same 20ms period
    digitalWrite(throttlePWM,HIGH);       //Begin throttle PWM high period
    digitalWrite(steeringPWM,HIGH);       //Begin steering PWM high period
    delayMicroseconds(centered);
    digitalWrite(steeringPWM, LOW);       //End steering PWM high period
    delayMicroseconds(forward-centered);
    digitalWrite(throttlePWM, LOW);       //End throttle PWM high period
    delayMicroseconds(20000-forward);     //Delay for the remainder of the 20ms period

    pinMode(steeringPWM, INPUT);
}

void forwardMovementTurning()
{                         //Single pwm for throttle
    digitalWrite(throttlePWM,HIGH);
    delayMicroseconds(forward);
    digitalWrite(throttlePWM, LOW);
    delayMicroseconds(20000-forward);
}
```

```
void stopMovementCentered()
{
    pinMode(steeringPWM, OUTPUT);
                            //Pwm's for stop and hold-center
    digitalWrite(throttlePWM,HIGH);
    digitalWrite(steeringPWM,HIGH);
    delayMicroseconds(stop);
    digitalWrite(throttlePWM, LOW);
    delayMicroseconds(centered-stop);
    digitalWrite(steeringPWM, LOW);
    delayMicroseconds(20000-centered);

    pinMode(steeringPWM, INPUT);
}

void stopMovementTurning()
{                           //Single pwm for stop
    digitalWrite(throttlePWM,HIGH);
    delayMicroseconds(stop);
    digitalWrite(throttlePWM, LOW);
    delayMicroseconds(20000-stop);
}

void evasionRight()
{                           //Output two PWM's
     digitalWrite(throttlePWM,HIGH);        //One for throttle
     digitalWrite(steeringPWM,HIGH);        //One for turn Right
     delayMicroseconds(1000);
     digitalWrite(steeringPWM, LOW);
     delayMicroseconds(forward-1000);
     digitalWrite(throttlePWM, LOW);
     delayMicroseconds(20000-forward);
}

void evasionLeft()
{                           //Output two pwm's
     digitalWrite(throttlePWM,HIGH);        //One for throttle
     digitalWrite(steeringPWM,HIGH);        //One for turn Left
     delayMicroseconds(forward);
     digitalWrite(throttlePWM, LOW);
     delayMicroseconds(2000-forward);
     digitalWrite(steeringPWM, LOW);
     delayMicroseconds(18000);
}
```

Cost Analysis

Part Name	Manufacturer	Model #	Quantity	$ / Item
RC Car	Traxxas Rustler	3710	1	$149.99
Electronic Speed Control	IPC	?	1	$89.99
Battery Charger	Prophet Sport Dynamite	DYN4056	1	$39.95
Ni-MH Battery	Venom 4600	VEN-1547	2	$29.95
9V Battery	Duracell	MN1604	2	$3.99
AA Battery	Duracell	MN1500	8	$1.95
1uF Capacitor	NTE	N/A	4	$0.99
RS-232 Buffer	MAXIM	MAX3323E	1	$3.95
Through Hole	SchmartBoard	201-0001-01	1	$5.00
Microcontroller	Arduino	NG4	2	$31.95
GPS Dev. Kit w/ Jupiter 30	Navman	AA003029	1	$299.00
MDT 850 SDK	Navman	AA003028	1	$245.00
MaxSonar EZ1	MaxBotix	S-10-EZ1	2	$29.95
			Total:	**$1,044.12**

**Note: All of the above analysis excludes the cost of replacement parts, construction materials and manufacturing overhead.

Project Summary

To sum it all up, we feel that we truly did accomplish all of the goals that we initially set about to achieve. In going about completing this project we broke it up into many little subtasks. We felt that by focusing on perfecting little tasks and then integrating them we would be successful when the end of the quarter neared. Our success confirms that we were right in pursuing our goals in such a manner. In looking back upon the road that we took to get here, one would find a variety of different obstacles that we overcame in a timely and efficient manner. Many obstacles were quite small and a few others were relatively big.

Between weeks one and two of this quarter we set about confirming the functionality of our GPS unit and our RC car. In order to get a fully functioning car in the end, 100% reliability of these two units was essential. The GPS unit turned out not only to be great, but a much better unit than we had expected to acquire. We encountered a minor hindrance with the RC car, in that we accidentally fried one of our electronic speed controls. In the end, this turned out to be a good thing because we ended up buying a much better speed control to replace the one we had.

Between weeks three and four, we decided that instead of having one Arduino do all the navigational work that we would have the Navman Mobile Data Terminal handle the navigational algorithms. After that decision we then decided that Arduino 1(Master) would only be in charge of vehicle functionality and that Arduino 2(Slave) would control throttle and hold center steering based off the master Arduino's commands. We also decided that this Arduino would be in charge of constantly monitoring the sonar sensors for any obstacle that may be in the vehicle's immediate path.

This now brings us to weeks five and six where much coding went into all the individual units. All individual components such as the two microcontrollers and the Navman Mobile Data Terminal became individually fully functioning. It would then become the obstacle of weeks seven and eight to fully integrate them with each other. Before going on to explain this however, it is necessary to note what the status of our obstacle avoidance algorithm was. Due to the fact that there were many problems with getting the sonar sensors shipped out in time we were not able to focus our full attention on obstacle avoidance until week 10 and though we have come up with a simple obstacle avoidance algorithm we would have liked to pursue more in depth algorithms.

Now, in shifting our attention back to weeks seven and eight we found that the problem that presented itself to us was the transfer of data from the terminal to the master Arduino. We thought long and hard about this and realized that the best way to do it would be to use a buffer that would enable us to take in the data from the terminal which was being output in RS-232 format. After assembling the buffer on our Through Hole SchmartBoard, we then were able to fully integrate the GPS receiver and terminal with the microcontrollers. Needless to say, after this part was done, we breathed much easier.

Week nine then came along and in that week we found ourselves at the Verizon Amphitheatre on a daily basis testing the functionality of our car. We set our waypoints at both the Verizon Amphitheatre and near our home so that we could test functionality without having to drive every single time we needed to perform a simple test. Our first run was moderately successful but then came the infamous second run where we found quite a few difficulties to overcome. In fact, we suffered a rather damaging vehicle crash in the process which resulted in a day's worth of repairs. After some hard thinking and incredible ingenuity we found ourselves completely successful with every next run. Week ten quickly dawned upon us and we found ourselves working a great deal on obstacle avoidance and repeatedly adjusting the threshold of the sonar sensors until we finally achieved victory in getting the vehicle to navigate around an array of cones. After all work had been done we then video taped our vehicle in action and prepared our documentation.

After all is said and done, we believe that this Senior Design Project course is the one class that we benefited the most from in the EECS department here at UCI. We learned a great deal of organizational, planning and implementation skills. These skills are quite invaluable and will be of assistance to us when we get out into the real world. Watching our ideas that first started out as diagrams and words on simple pieces of paper, become a fully functioning GPS Self-Navigating vehicle was a complete thrill. It really is hard to put into words the sense of joy we felt when we succeeded in getting the car to get to every distinct waypoint that was set. No matter what grade we get we feel as if we have already been rewarded by seeing our project progress from its early inception in our minds, to a very real and physical element.