Formal Report

Nicholas Wulf

MickeyBot

EEL 5666: Intelligent Machines Design Laboratory
Instructors: Dr. A. Antonio Arroyo, Dr. Eric M. Schwartz
TAs: Adam Barnett, Mike Pridgen, Sara Keen

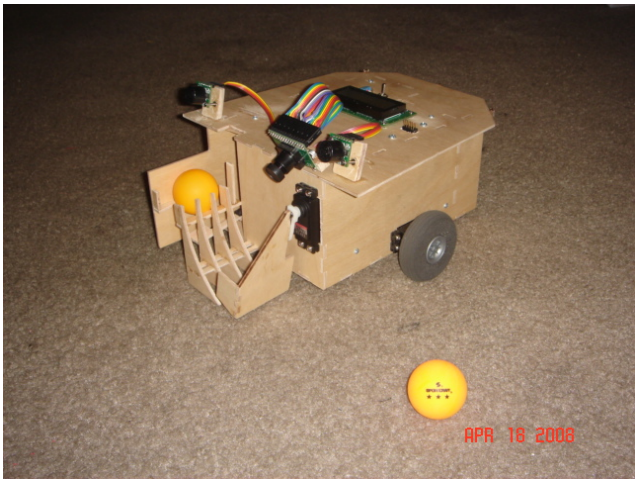


Table of Contents:

Abstract:

The purpose of this report is to detail the design and operations of my robot MickeyBot, which is named after my dog. Once turned on, the user will present MickeyBot with a ball, and MickeyBot will then go out find a ball of the same color and bring it back to the base location. Once it arrives at the base with a ball, MickeyBot will eject the ball into the air so that the user can catch it. MickeyBot uses a MAVRIC IIB board, 3 servos for locomotion and ball grabbing, an LCD for outputting to the user, 2 sonar sensors for detecting any obstacles, a CMUCam2 for identifying colored objects, and a solenoid for ejecting the ball.

Executive Summary:

MickeyBot's purpose is to locate a ball in the surrounding environment, bring it back to the user, and launch the ball into the air so the user can catch it.

MickeyBot is controlled by a MAVRIC IIB board which houses an ATMega128 microprocessor. Also, many device connections in the robot are routed to the MAVRIC IIB board through a custom designed board, which I designed to simplify connections.

Two sonar sensors are placed on the front of the robot and are used for detecting objects before MickeyBot has a chance to run into them.

The special sensor for the robot is a CMUCam2 which is used to track the position of the colored ball and base. This camera has also been placed on the top of the robot and tilted down so that the robot can detect how far away a given target is.

2 servos are directly attached to wheels so as to allow the robot to move around the environment. An additional servo is placed on the front of the robot and controls the paddle which is responsible for capturing balls.

A solenoid is used to launch the ball vertically into the air. A lever is used to convert the torque generated by the solenoid from high force low velocity to low force high velocity, which is what is required to launch the ball up high. A 0.1 Farad capacitor is used to store the energy needed for the solenoid.

An LCD is attached to the top of the robot so as to provide the user with information during operations. This was also an essential part for initial debugging.

8 1.2V NiMH rechargeable batteries are used to power the robot. This should produce 9.6V but in actuality produces around 11V.

MickeyBot's platform is made out of 1/8 in birch wood that was cut and provided by the IMDL lab.

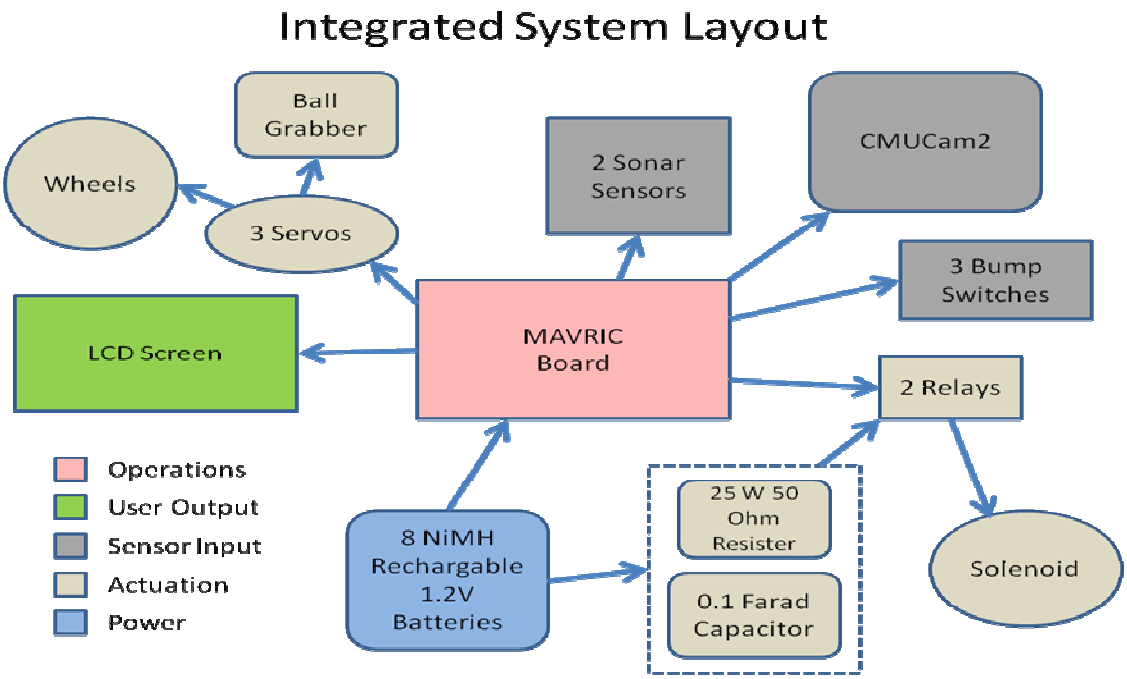
Introduction:

Beer-pong happens to be a popular game at college for some reason, and as such I figured that it would be worth my time to do a little practice by myself so that I could better impress my friends later during actual play. It was a fairly boring activity that consisted of setting up a table next to a wall with cups of water on it, standing about 5 to 7 feet away from the cups, and trying to throw a ping-pong ball into the cups. However, the worst part of the "training" was that after I would throw the balls at the cups, I would have to go around and under the table to pick up balls that I missed with. This is the inspiration behind my robot MickeyBot, which I named after my childhood doggy friend. MickeyBot's purpose will be to retrieve these ping-pong balls on the ground and retrieve them for me while I spend my time more efficiently honing my skills. It will most likely not be used regularly during actual competitive play. Beer-pong participants usually aren't so good about watching where they put there feet, and coincidentally one of the most dangerous natural hazards for small robot dogs happens to be misplaced steps.

The rest of the paper will go into the details of individual systems used for actuation and sensing, the integration of all of these systems, the physical structure of the robot, the behavior of the robot, results of the final design, and ending conclusions.

Integrated System:

Below is a diagram of the different parts used in the robot (other than the wood cutouts for the platform itself) and how they all interact.



The processor I'm using is an Atmel ATMega 128 which is interfaces and is housed on the BDMicro MAVRIC IIB board. This is the brains of the robot. All sensor data and actuation commands are processed and controlled through here. I've also made my own board (shown in the appendix) that helps the MAVRIC board to more easily interface with the sensors and actuators.

The LCD screen is a standard 16-pin backlit LCD from Microtips Technology. It can operate in 8-bit or 4-bit data mode and can output any combination of 16 ASCII characters on each of its 2 lines for a total of 32 simultaneous characters. I've set it to 4-bit operation mode so as to reduce the number of pins used. The LCD was critical for debugging purposes, and functions as a basic process updater to the user in its final design.

For sensing the environment, I will be using 2 sonar sensors, 3 bump switches, and a CMUcam2. The sonar sensors are MaxSonar-EX0 sensors and are used for detecting objects in front so that no frontal collisions occur. The bump sensors are mounted on the rear and are used for detecting rear-end collisions. Finally, the CMUcam2 is mounted on the front and is used for detecting where the ball is located.

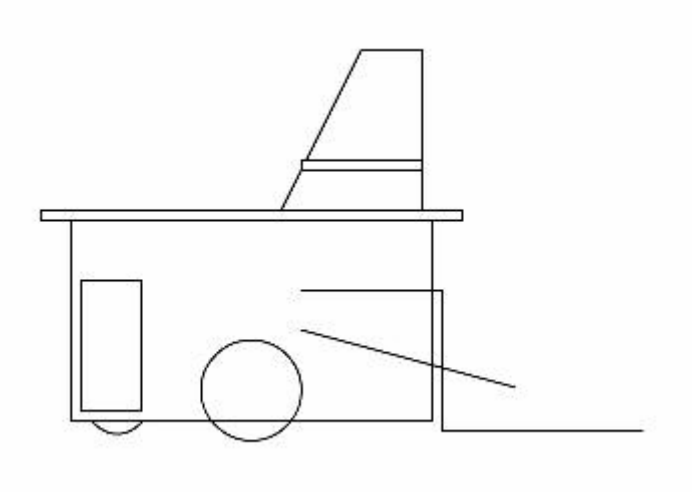
The 3 main actuation functions are locomotion, ball pickup, and ball firing. Locomotion is handled by two hacked servos, each directly controlling a wheel. The servos have been hacked so that they can rotate freely in 360 degrees and send no positional feedback. Picking up the ball is handled by an un-hacked positional servo. This servo sits on the front and controls the ball lift, which moves the ball up the ramp and loads it for firing. The firing mechanism is slightly more complicated. The main focus here is the solenoid that actually provides the torque needed for launching the ball. However, the solenoid requires a large current spike in order to activate, so instead of powering the solenoid

directly from the batteries, a 0.1 F capacitor is charged instead and is used for supplying the power to the solenoid at the appropriate time.

For the robot's power source, I've chosen to use 8 NiMH batteries. I chose NiMH because they were readily available (purchased Energizer batteries and charger at Wal-Mart) and because they apparently don't have the memory loss that many other rechargeable batteries have. Since each battery is a 1.2 cell, then all 8 of them in series should produce around 9.6V total. However, in actuality, a full charge produces closer to 11 V.

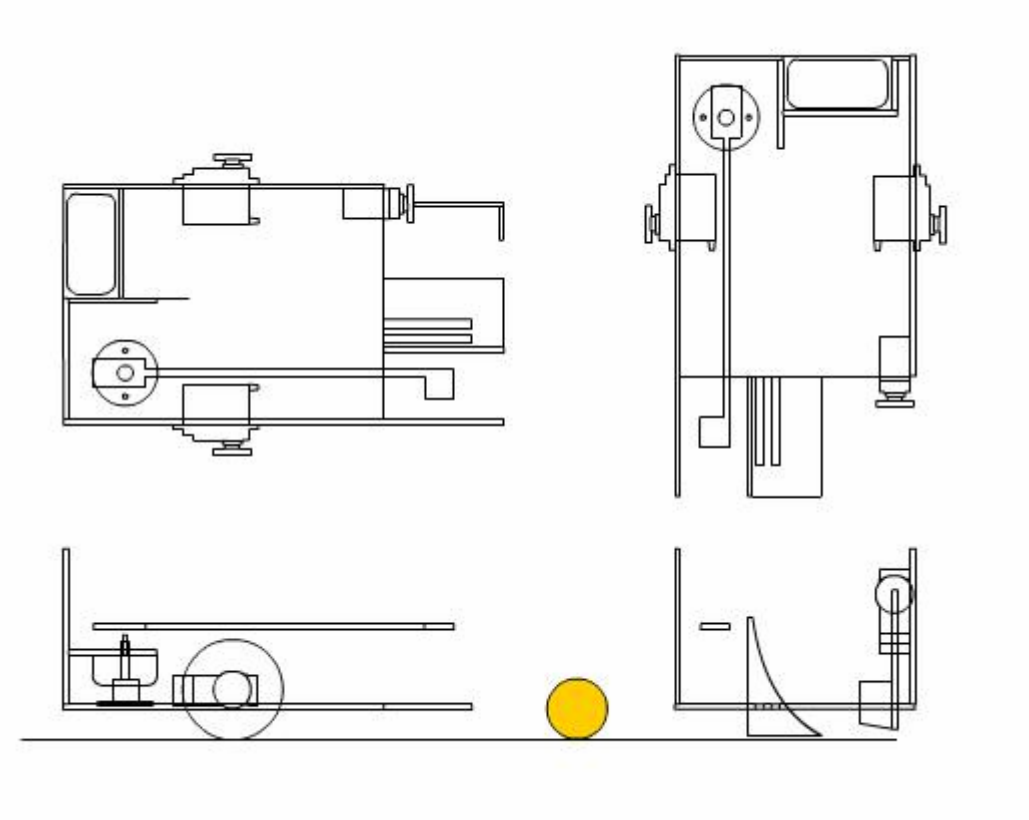
Mobile Platform:

The platform is constructed out of 1/8 inch birch wood and was supplied and cut by the IMDL lab and TAs. The platform is effectively a box that houses almost all components internally. The only pieces that are mounted outside are the sensors, the LCD screen, and the wheels. Additionally, the ball loader sticks out in front mainly so that the ball doesn't collide with the platform when it fires.



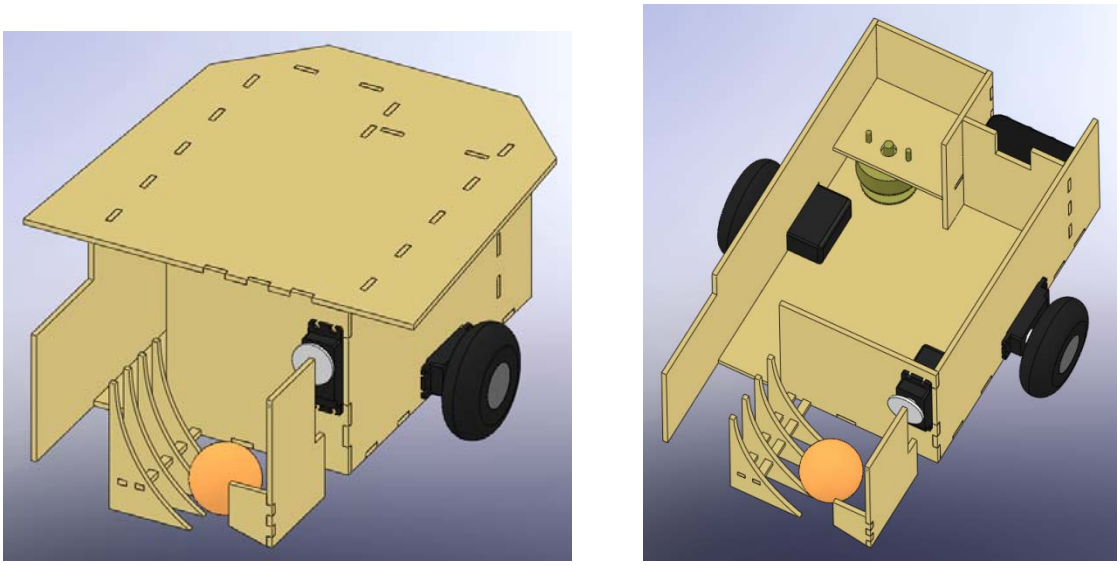
Initial Platform Design

This was my initial sketch for what my robot would look like. I had ambitiously decided to mount the sensors on the elevated piece above the main body. I also had not really worked out a well working ball loader or launcher.



Final Platform Design

Above are my final sketches for the platform design. This only shows off the bottom part of the robot without all the sensors. However, it contains all necessary actuator designs which were a priority at the time.



3D platform representation

These are some screenshots of the main platform modeled in SolidWorks. It still does not show any sensors or circuitry. This was used primarily to ensure that my AutoCAD wood cutout designs were correct and wouldn't conflict with each other in unforeseeable ways. These AutoCAD designs can be found in the appendix.

Actuation:

MickeyBot is actuated by 3 systems. The locomotion system allows for movement around the environment. The grabber system allows the robot to pick a ball off the ground and load it. Finally, the firing system is used to launch the loaded ball into the air so that the user can easily catch it.

Locomotion is achieved through directly controlling 2 servos with the ATMega's built in PWM generator. The MAVRIC board also comes with servo headers, so it was not necessary to connect the servos through my custom built board. The servos also needed to be hacked in order to use them properly. The original un-hacked servo is meant to move the gear head to a specific position. It does this by obtaining electrical feedback of the current position through a potentiometer connected directly to the gear head, and then uses this feedback in a linear control feedback loop to adjust the gears to the proper position. In order to hack the servo, this feedback was cutoff and the potentiometer glued into the neutral position. There was also a stopper that was removed, which kept the servo from rotating past 90 degrees on either side. Once these two modifications are made, the servo is constantly fooled into thinking that it is always in the neutral position. Due to the internal control system, the further an input is from neutral, the faster the gear will turn in that direction. Attaching wheels directly onto the gear head gave me a wheel whose rotation I could easily control the direction and speed of. By using two of these servo-wheel constructions and placing them on the left and right sides of the platform, the robot can easily go forward, stop, turn, and backup, all at varying speeds. The servos are HiTec HS-422 standard deluxe servos. They run off of 4.8V to 6V and produce a stall torque of about 48oz.in.

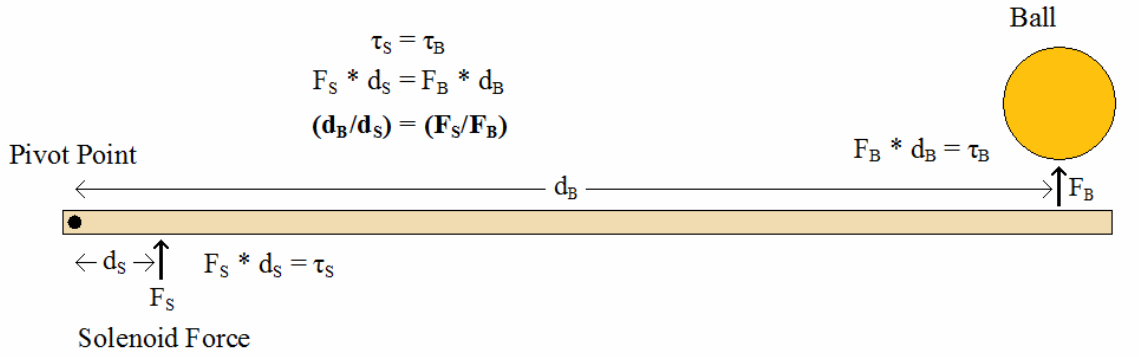
Servo #	Min-1000us (rpm)	Max-2000us (rpm)
1	49.5	49.75
2	47.5	50

Since the servos will have to be in the opposite orientation from each other (since there's one on either side), any forward motion of the robot will involve one servo in forward motion and the other in reverse. Thus, in order to optimize forward motion to be as straight as possible, I chose the closest matching forward-reverse pair. This turns out to be the reverse speed of servo 1 and the forward speed of servo 2. In the final design, for forward motion, servo 2 is given a high pulse width and servo 1 is sent a low pulse width.

Grabbing the ball was initially planned out as a more complicated process involving two servos, one to move the ball into position and one to lift it up so as to be fired, as well as

other gears and gear shafts. The final design is much more simplistic and just as effective if not more so. This design involves only one servo that is directly attached to a wide paddle. Once the ball is somewhere in front of the robot the paddle will swing down and capture the ball. A stopper is placed on the end of the paddle to make sure that the ball doesn't roll out. The paddle then continues along its rotation and lifts the ball up a curved ramp. At the top of the ramp, the ball rolls out and into the loading compartment where it waits to be fired.

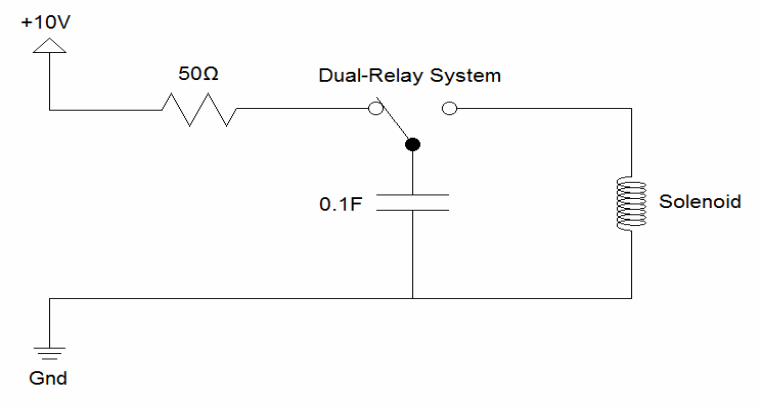
The Launcher turned is the most electrically complicated of the actuation systems. The idea behind it was to use a solenoid to launch the ball from a stationary position on the ground to about 5 feet in the air where it can be easily grabbed. However, a couple issues had to be dealt with for this system to function properly. The solenoid is effectively a long conductive material that is looped many times around a free moving magnetic core. When current is passed through this conductive material, a magnetic field is induced inside the loop. The magnetic core then aligns with this induced field and is quickly pulled inward. A pin on the forward moving side of the core pushes through a hole in the solenoid and pops out, hitting anything in its way. However, during initial testing, I discovered that the solenoid wouldn't be able to launch the ball high enough with a direct hit. To overcome this problem, I took the advice of Dr. Schwartz and created a mechanism as shown below.



The solenoid generates a large force (much larger than what is needed to move a hollow plastic ball) yet it can't generate enough speed to hit the ball high enough. By implement the above ball batter, the force of the solenoid at the base of the batter is transformed into a torque, which is then transferred down the batter to the ball. Since the ball is much farther away from pivot point than the solenoid, the torque generated by the solenoid produces a far less force at the ball's location. However, since the ball is so far away from the pivot point, the batter will hit the ball much faster and transfer more speed to the ball than it would have if the ball were closer. In this way, the system trades force for speed, and the result is a ball that soars much higher (5 to 6 feet) than it would have otherwise.

Another problem was the fact that the solenoid draws a large amount of current when it is activated. If I were to power the solenoid directly from the battery source, this large spike in current could cause a momentary drop in voltage, possibly resetting important

devices. Thus, I decided to use a 0.1 F capacitor to store up power and then release it through a series of relays. I also used a 50Ω resistor rated for 25W. The resistor is used to slow down the charge time of the capacitor so that not too much current is drained from the batteries at once. However, during the beginning of a charge the capacitor can draw around 200mA, causing the resistor to consume about 2W. Although the normal resistors provided by the IMDL lab may have been able to handle this for a few seconds, I decided to err on the side of caution and use resistors that I knew would work. The firing mechanism circuitry is shown below.



Sensors:

MickeyBot uses two sonar sensors, 3 bump switches, and a CMUcam2 in order to navigate through its environment and retrieve the ball.

The 3 bump switches are placed on the rear of the robot and are used to detect if the robot backs into an obstacle. Bump detection is adequate here since the robot has no delicate parts on its rear. The signal from the switch is pulled high normally, and when the switch is pressed, a connection to ground is made that pulls the signal low. Thus the microprocessor interprets a high signal as no collision and a low signal as a hit.

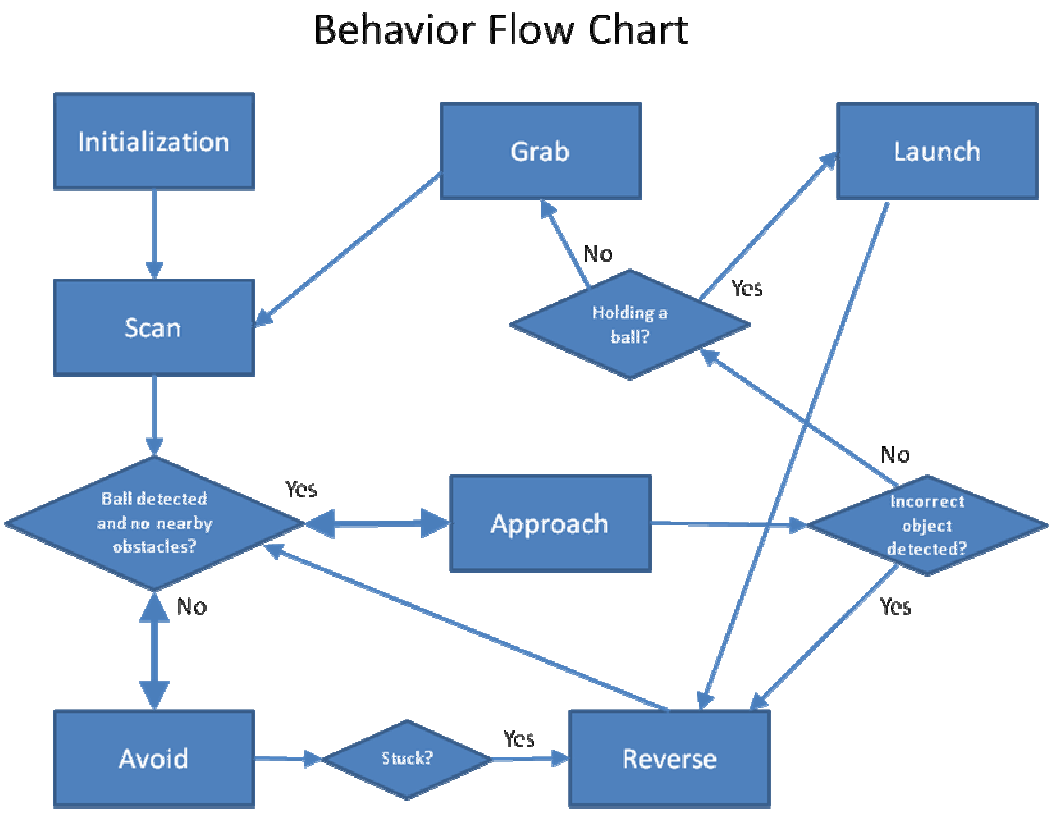
The 2 sonar sensors are MaxSonar-EX0 models. The sensors are designed to detect objects within a distance of 0 to 24 feet. They are provided with +5V power and ground from the MAVRIC board. The sensors relay information to the microprocessor by way of a single analog signal which is set to a value of 9.8mV/in, which is interpreted by the microprocessor's analog to digital converter. They are also capable of sending the same distance data through PWM signals or serial RS232. However, these methods are not used for simplicity's sake. These sensors are mounted on the front of the robot on the left and right sides and are used for obstacle avoidance. It is important that the robot perform non-contact obstacle avoidance in the front since it will have more easily breakable parts here. After extensive testing of three sonar sensors, two were found to match remarkably well and these two are the ones used in the final design. A more extensive description of the testing process can be found in the Experimental Layout and Results section. In

summary, the sensors have a viewing angle of 60 degrees, detect a beer bottle just as easily as a wall, and produce a consistent and linear output voltage based on distance.

The CMUCam2 captures images of its environment in real time and computes convenient and relevant data about these images through the use of its on board processor. It can be hooked up through serial to a computer and used to output images such as the one below, or it can be directly hooked up to the MAVRIC board and communicate through the use of RS232 serial. MickeyBot will use the CMUCam2 in order to locate the ball and then the user on the return trip. By specifying a min and max value for each red, green, and blue color component, a range of colors can be specified to the camera, which will then search for the location of any pixels in said range and return the pixel location for the average of these positives. Thus, MickeyBot will receive two values from the camera in real time that will specify where any objects of interest may be.

Behaviors:

The below chart shows the basic flow of logic that controls the behavior of MickeyBot. The boxes represent the different states that MickeyBot can be in at anytime and each will be discussed below.



The initialization state takes care of initializing all of the different systems such as the pwm generators and A/D converters. This is also where the CMUCam2 is initialized, allowed to adjust to the ambient light, and grab the color of the target ball.

The scan state makes MickeyBot turn in place in a random direction. If a target is detected at any time during this state then the turning will stop and the state will switch to approach. If the scan state completes without detecting a target, the robot will go to the avoid state. The duration of the scan state lasts for at least one full 360 degree turn so as to make sure that there are no surrounding targets. Then the robot will spin anywhere between 0 and 360 degrees which is also selected randomly. This ensures that if the robot does not immediately detect a target, it will seek out the target in a random direction every time it does a scan. The random number generator used here is based on values collected from a sonar sampling which tend to be somewhat noisy and random in the lower bits.

The avoid state takes care of the obstacle avoidance within the system. This state is entered whenever no target is detected or if there is an object close to the front of the robot. The obstacle avoidance algorithm is simple yet effective. There are two sonar sensors at the front of the robot on the left and right that measure the distance to the closest object in front of them. Each sonar is individually responsible for controlling the speed of the wheel on the opposite side, causing the wheel to slow down and eventually reverse as objects get closer and closer. For example, if there is an object up ahead that is slightly left of the robot, the left sensor will report a lesser distance than the right. This

will cause the right wheel to slow down and the robot will make a slight right turn while still going forward at almost full speed. If however, an object is placed right in front of the robot, both sensors will report a low distance and both wheels will move in reverse. In addition to this simple avoidance system, if the robot should ever get stuck for a few seconds, such as in a corner where it is difficult to tell where to turn, then the robot will enter into the reverse state to try to free itself from the situation.

The approach state is used for tracking and approaching targets and is entered when a target is detected and there are no nearby obstacles. Once an object is detected by the CMUCam2, the ball coordinates relative to the camera are sent to the robot. The target's position along the x-axis is used for approaching the target, whereas the y-axis position and size of target are used for identifying the target once it is reached. While approaching the target, if the more the x-value differs from center, the faster the robot will turn to face the ball and the slower it will move forward. Since the camera is positioned about 5 inches off of the ground and is tilted downwards, a lower y-value indicates a closer target. Once a certain y-value is reached, the robot will know that it is close to the target, at which point it uses the target size to identify the target as either the ball or the base. If the robot already has a ball and detects a base, it will enter into the launch state. If the robot has no ball and detects one in front of it, it will enter into the grab state. If however none of these conditions is true and the object is not the correct target, the robot will enter into the reverse state.

The grab state is used when the robot has no ball and one is detected on the ground from the approach state. Once in this state, the robot will stop, open it's grabber paddle, move forward, and move the grabber paddle in to capture the ball and transfer it to the launching bay. The robot will then enter into scan mode in order to search for the base.

The reverse state is similar to the scan state except for two key differences. First, the robot will always turn 180 degrees in the absence of a detected target. Second, a special blind mode can be activated which causes the robot to ignore any detected targets for the first 80 degrees of the turn. This is important for when the robot approaches an incorrect object and then needs to turn around. If it was not initially blinded then it would immediately hone in on the incorrect target again as soon as it started trying to turn. The only time that the blind mode is not used is when the robot performs a reverse due to getting stuck while in obstacle avoidance.

The launch state is used to eject the ball into the air once a base has been approached. Afterwards the robot enters into the reverse state rather than the scan state, so as to not accidentally detect the base again. For more details on the launching mechanism, see the Actuation section above.

Experimental Layout and Results:

Here, I will go into detail on the various tests that were performed on the different actuators and sensors that were used in the robot.

Sonar Sensors:
This was the most extensively tested component of the system. Thanks to the recommendations of the teachers and TAs, I purchased 3 of these sensors even though I knew I only needed two. There were two main tests that were run on each of the sensors. The first tested the values obtained from looking at a wall. The sensors were positioned about 3 inches off of the ground and a large flat box was used to simulate the wall. 20 measurements were taken from 1 to 30 inches. The second test was identical except that instead of using a large box, I used a regular 12oz beer bottle which was meant to simulate any kind of thin-like obstacle the robot might encounter. Also, the angle of detection was determined with this test.

In total there were 120 different readings taken! Average deviations were then taken of various combinations of data, which helped to identify which two sensors were most consistent with each other and which sensors were most consistent between their wall and bottle readings.

The results were quite favorable. There turned out to be very little difference between the wall and bottle readings, and two of the sensors were determined to match very well (numbers 1 and 2). All results data and graphs can be found in the Appendices section.

CMUCam2:
The CMUCam2 was tested extensively with the provided GUI and HyperTerminal serial communications. The GUI that came with the camera was very nice to have since it showed exactly what the robot would be seeing and helped to easily identify any possible viewing problems that might arise. This was also instrumental in making sure that I mounted the camera at the correct angle so that the robot could see the horizon as well as the ground below it. The HyperTerminal serial communications testing was also vital in that it allowed me to simulate the interactions between the MAVRIC board and the camera, while letting me see exactly what is being communicated between the two devices. Setting up and debugging the RS-232 code would have been much more difficult without the use of HyperTerminal.

Servos:
As discussed in the Actuation section, I tested the two servos that I was using for locomotion in both the 100% forward and reverse settings. For exactly one minute, I counted the cycles of the rotating servo gear heads and used these values to determine the correct orientation for the servos that would produce the most straight forward motion.

Solenoid/Launcher:
The launcher testing was probably the most haphazard test that I performed, yet ended up producing a successful system. The first test that I ran was to determine if the solenoid could directly hit the ball and launch it high enough. It was quite difficult to actually engage the solenoid with the batteries and hold the solenoid and ball at the same time (the solenoid has nothing for your hands to really grab onto and can easily pinch your fingers

hard if held improperly). However, after enough testing, it was determined that the solenoid could only launch the ball about 6 inches upward, which was far from adequate.

Based on the advice of the teachers, I decided to implement a lever system (discussed in far more detail in the Actuators section) that would convert the high force low velocity of the solenoid to a much more favorable low force high velocity hit. Initial tests however were discouraging. The lever/hinge system used was a plastic 6 inch ruler that was attached to two L-brackets through the use of nuts and bolts. In retrospect this was a horribly designed contraption, yet I was eager to test my launcher with whatever was available at the time. With the help of my roommates, I was able to hold the solenoid under the ruler, hold down the L-brackets with my tape, hold the ball above the ruler and then engage the solenoid. It was an awfully awkward setup that produced only slightly better results than the direct solenoid hits. However, I recognized the fact that some of the energy was most likely lost to the fact that my hand was pushed down by the solenoid every time it activated. In the end, I decided to just continue with the launcher construction as planned and to just hope that the final system would perform better. I had placed my faith in what I knew should work in theory rather than what I was seeing to work in practice. Surprisingly, this approach paid off. The final design is a success and launches the ball upwards of 4 feet into the air!

Conclusion:

I'm proud to say that MickeyBot is a success! Never before have I been so challenged and worked so hard as in this class. The end result is something I can truly be proud of as well. Not only do I now have a fun robot, but I have gained an immense amount of knowledge and experience. This was my last semester as an undergrad, and I feel that it has served as the perfect climax for my four years of education in that it effectively put to use the culmination of my last four years of study as an electrical engineer. This was a great opportunity to forego raw theory and to actually create something.

I feel that I was adequately prepared to understand almost everything I needed for this project due to my previous class EEL 4744 (Microprocessors and Applications). This is not to say that I did not learn anything new, since I most definitely did. However, I think that more important than the knowledge gained, is the confidence that this course builds. At the beginning of the semester, I was intimidated by the idea of building a robot and saw the robots of others in past semesters as being something that I could never achieve. Now I am filled with confidence in this area and am thinking of new possible robots that I could build in the future.

For future IMDL students I have the following advice: Start planning early and order your parts as soon as you can, especially if you want to enclose your design within as small an area as possible. Until you know the sizes of the parts you are going to use, you can't know how to build your platform, and until you build your platform and mount your devices and board, it's hard to really start testing system integration. My mistake was waiting until Spring break until I had even designed the platform in SolidWorks and had

all of my important parts. This left me around a month and a half to do what should have taken a whole semester, and my social life took a brutal hit because of this. Also make sure that for most of your parts, you order one or two more than you actually need. There were several cases where my progress could have been greatly hindered if I had not heeded the advice of the teachers and TAs and ordered more than I needed. Also a hard learned lesson for myself, don't mess around with the fuse bits on the MAVRIC board unless you are absolutely sure of what you are doing. Doing so could permanently lock you out of your board and cost you $140 to overnight ship another.

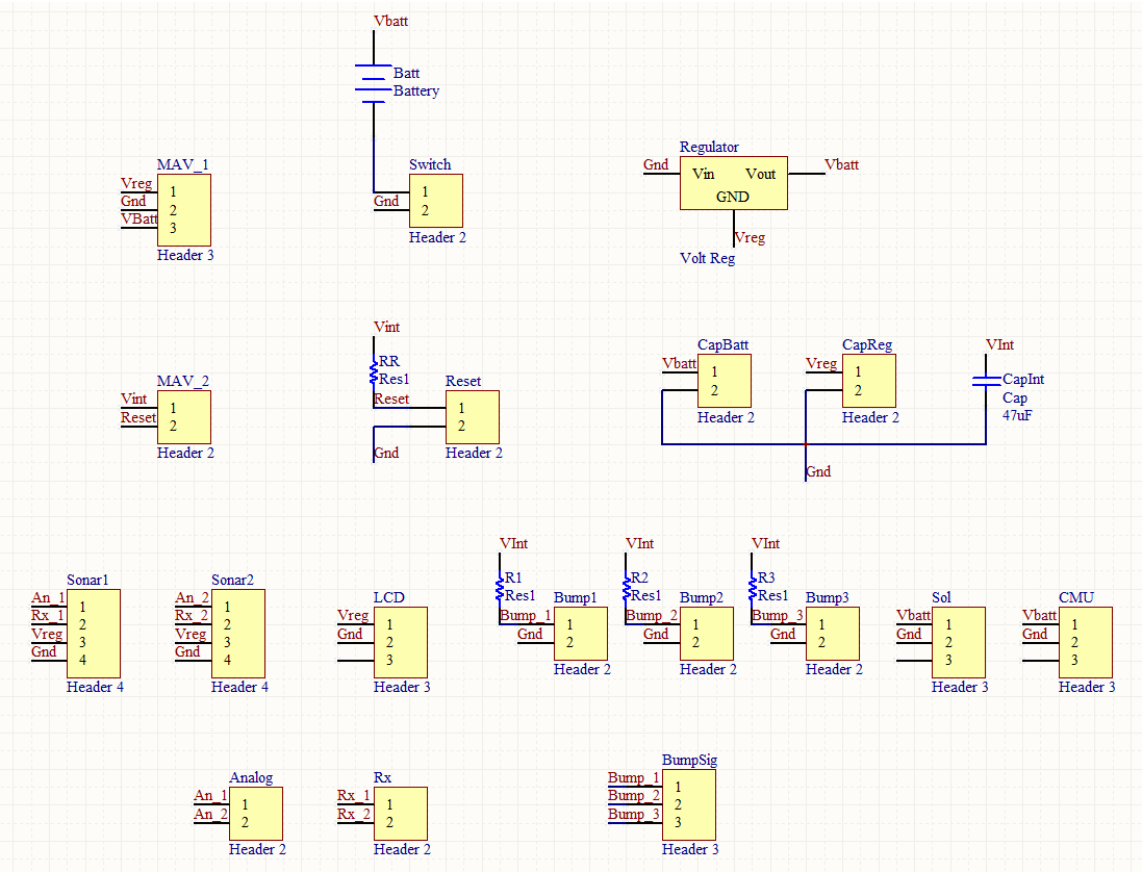
As for what I would do differently next time, my robot turned out much better than I ever thought it would and I can't really think of too much that I could do to improve it except paint it and make it a flashier design.

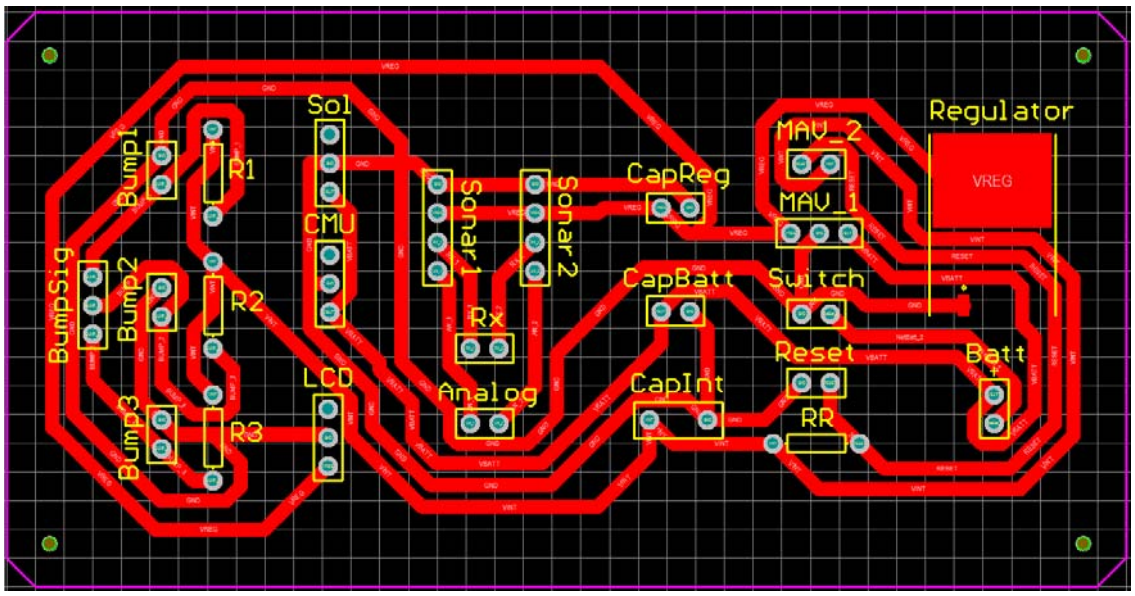
Documentation:

Other than the data sheets provided with the parts used in the robots, the only document that I used in creating the robot and code was AVR GCC tutorial written by Leitner Harald and provided on the IMDL website. I would also like to thank the teachers for their initial guidance and the TAs for providing help and for holding extra lab hours. Mike Pridgen was also a great help in letting me temporarily use an extra board of his when mine broke. Finally I would like to thank Adam Barnett for the LCD code that he provided on the website to all the students at the beginning of the semester.

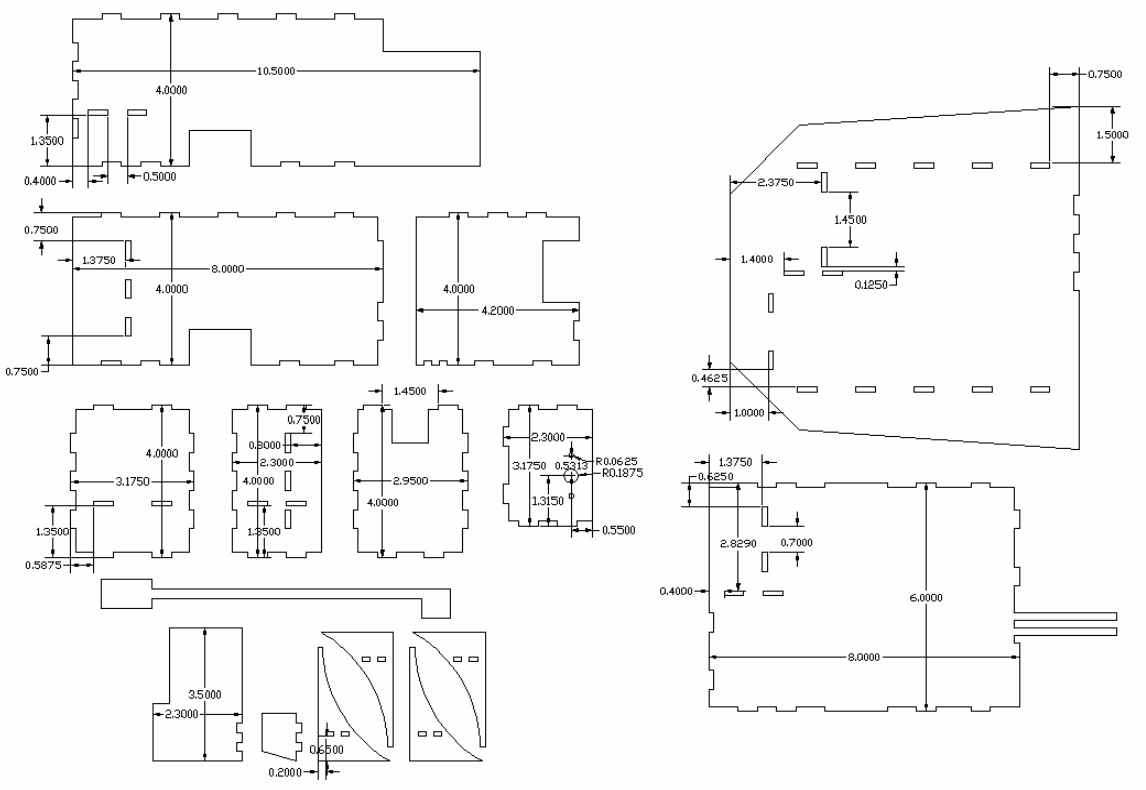
Appendices:

Altium Protel Schematics and Circuit Design





AutoCAD Wood Cutouts:

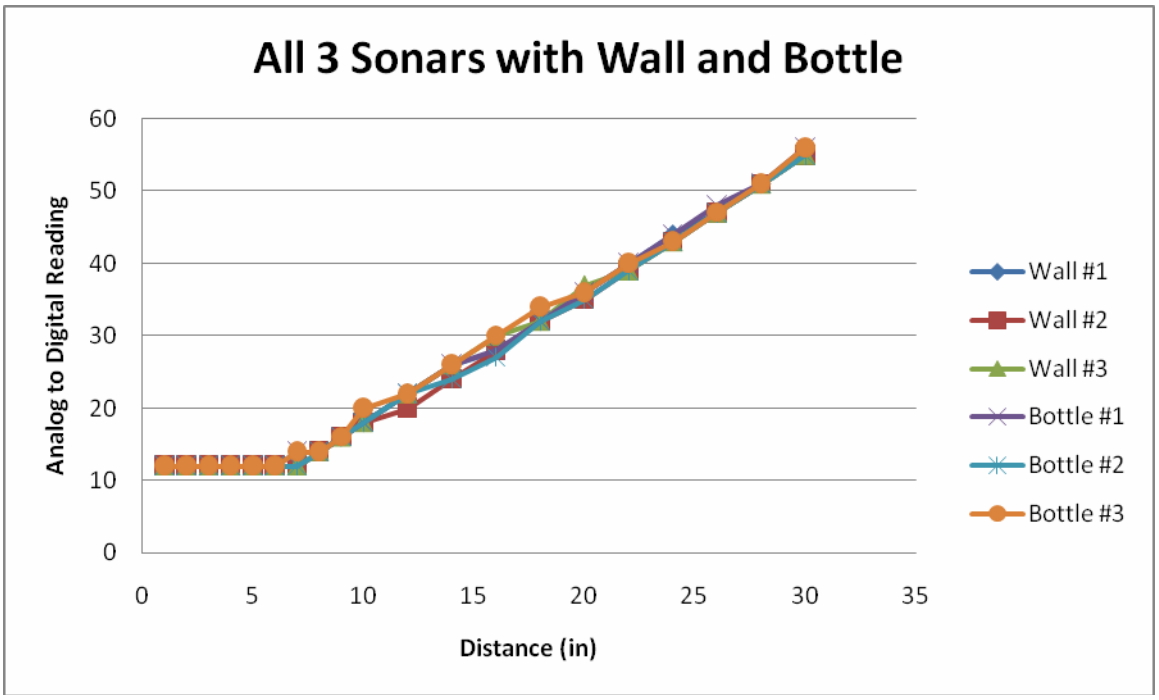


Colors are inverted to increase visibility

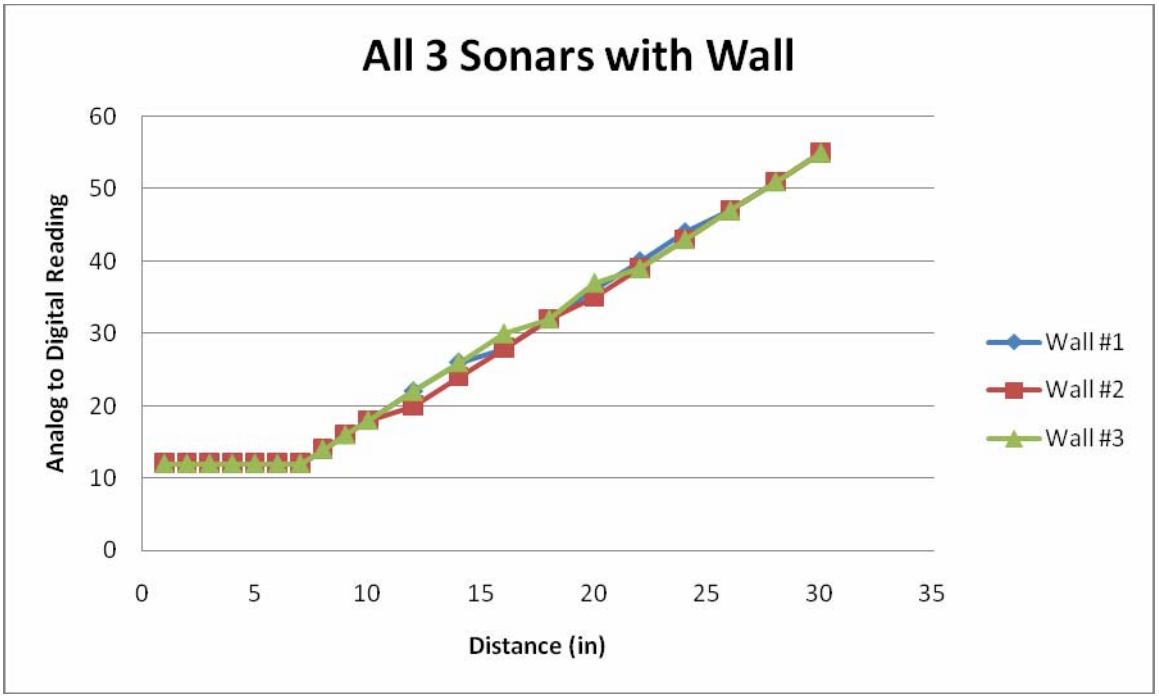
Sonar Test:

Sonar Test - Wall			
2.25in above ground			
Distance (in)	#1	#2	#3
1	12	12	12
2	12	12	12
3	12	12	12
4	12	12	12
5	12	12	12
6	12	12	12
7	12	12	12
8	14	14	14
9	16	16	16
10	18	18	18
12	22	20	22
14	26	24	26
16	28	28	30
18	32	32	32
20	36	35	37
22	40	39	39
24	44	43	43
26	47	47	47
28	51	51	51
30	55	55	55

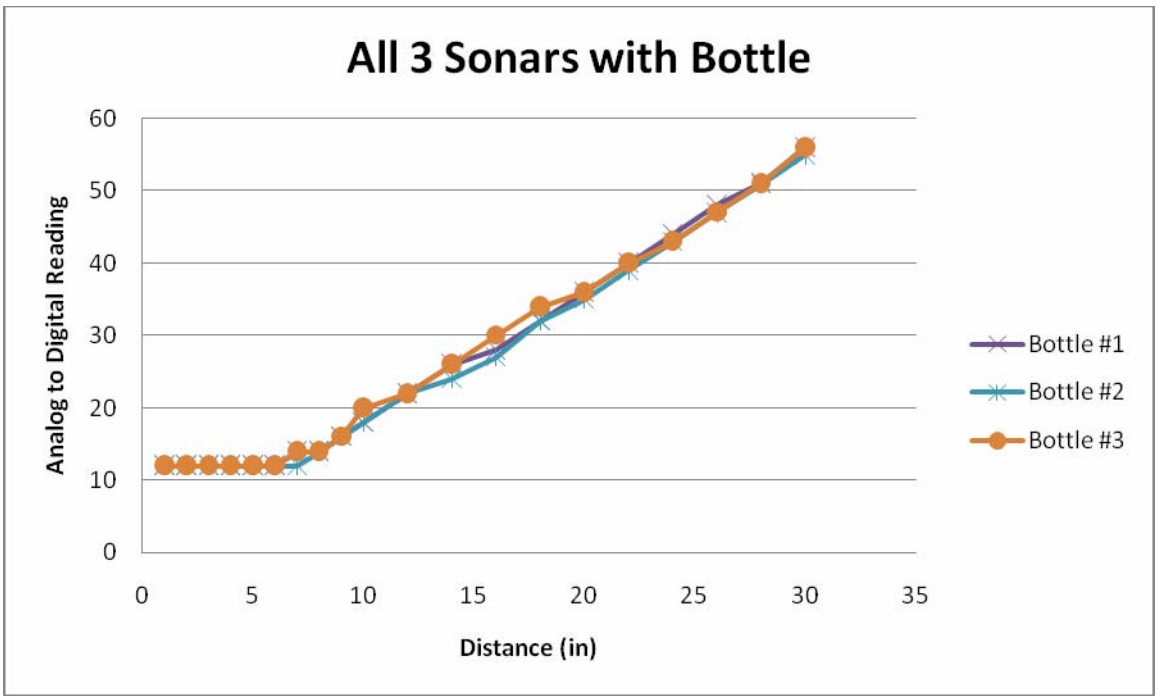
Sonar Test - Beer Bottle			
2.25in above ground, 12oz, 9in tall, 2.4in diameter base, dark glass			
Distance (in)	#1	#2	#3
1	12	12	12
2	12	12	12
3	12	12	12
4	12	12	12
5	12	12	12
6	12	12	12
7	14	12	14
8	14	14	14
9	16	16	16
10	18	18	20
12	22	22	22
14	26	24	26
16	28	27	30
18	32	32	34
20	36	35	36
22	40	39	40
24	44	43	43
26	48	47	47
28	51	51	51
30	56	55	56
Detection Angle on either side of center (1ft dis)	~ 30°	~ 33°	~ 18°



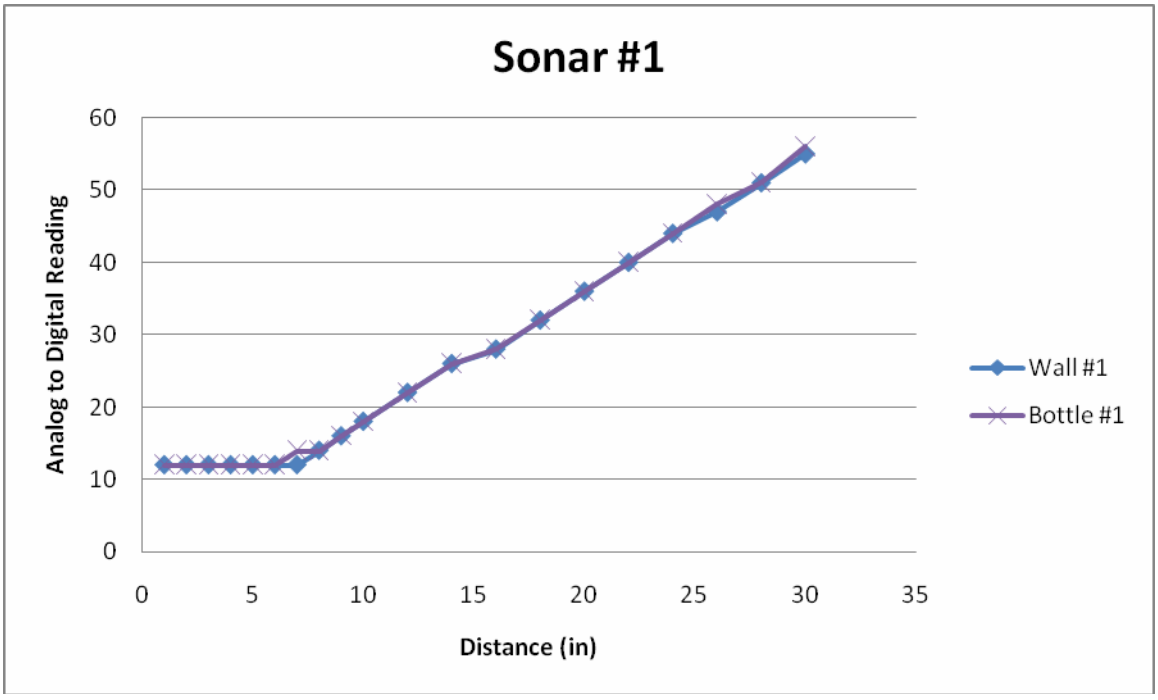
Slope = 0.543



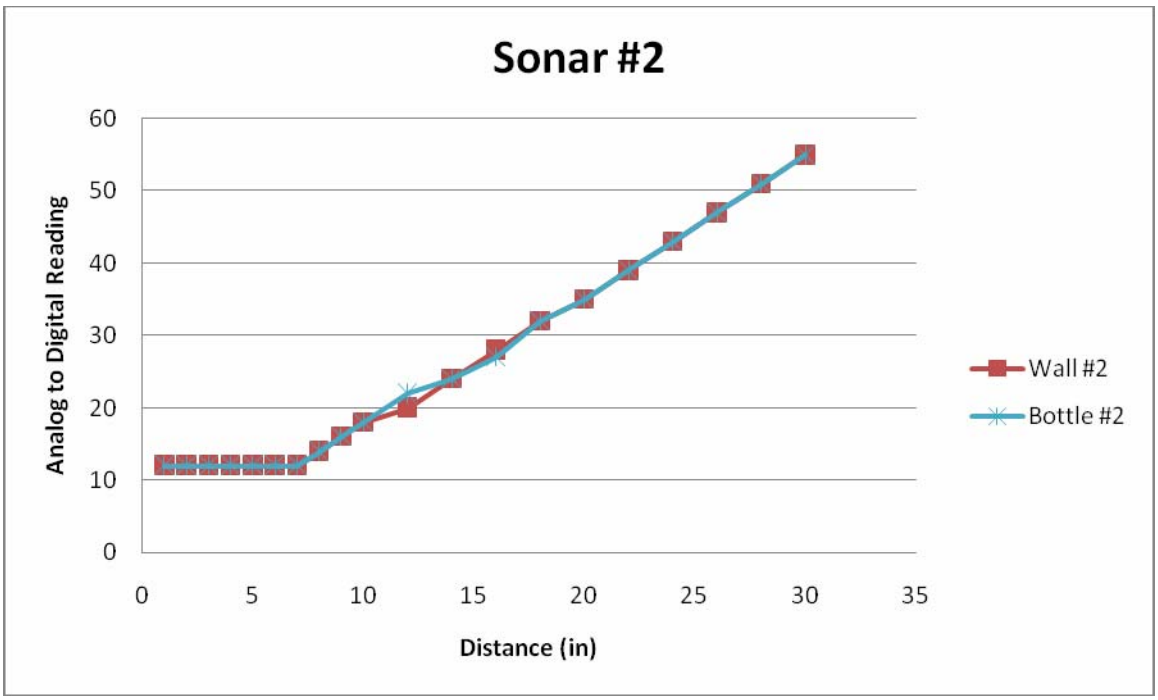
Average Deviations: #1 – 0.17, #2 – 0.25, #3 – 0.22



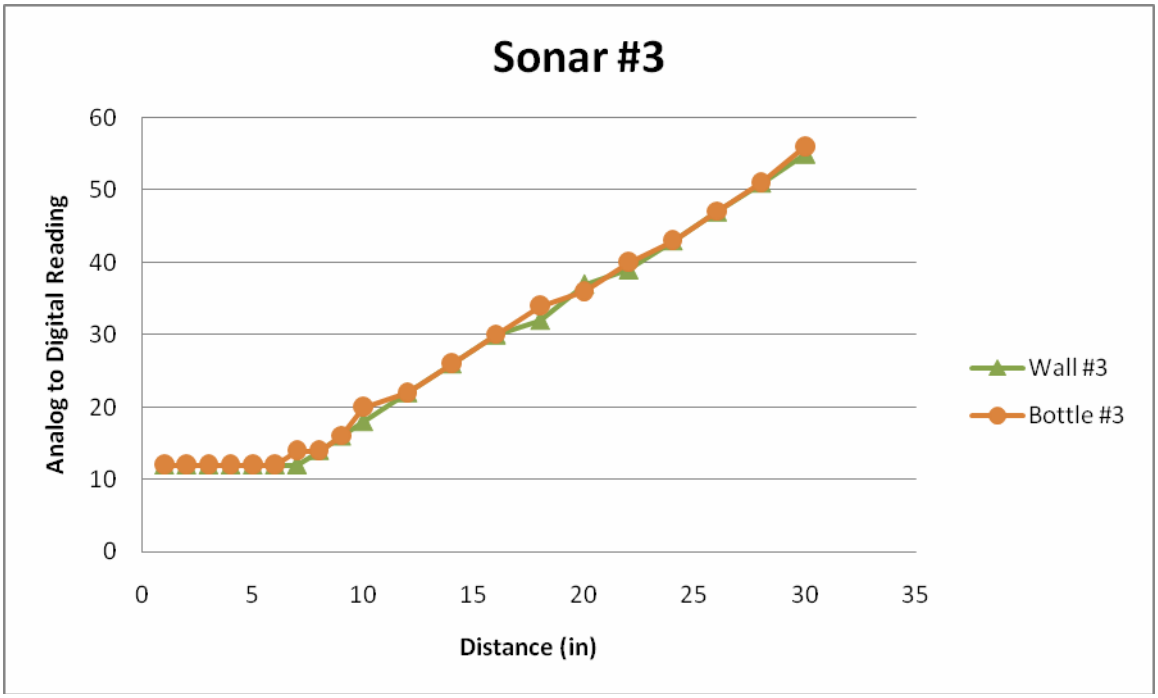
Average Deviations: #1 – 0.37, #2 – 0.27, #3 – 0.53



Average Deviation: 0.1



Average Deviation: 0.08



Average Deviation: 0.23

Final Code:

```
/*********************************************************************
 *                                                                   *
 * Title:           FinalProgram.c                                   *
 * Programmer:      Nicholas Wulf (LCD code provided by Adam Barnett)*
 * Date:            April 15, 2008                                   *
 * Version:                 1.0                                      *
 *                                                                   *
 * Description:                                                      *
 *        This is the code for the final MickeyBot design            *
 *                                                                   *
 *********************************************************************/


/************************** Includes *******************************/


#include <avr/io.h>
#include <avr/interrupt.h>

void launch();           // launches the ball into the air
void reverse();          // do a 180
void avoid();            // move forward while avoiding objects
void filter1();          // filter sonar1 value
void filter2();          // filter sonar2 value
void adc_init();         // initialize A/D converter
void approach();         // turn the robot so that it faces and moves towards the ball
void grab();             // move forward and grab the ball
void scan();             // turn around in place looking for ball
void pwm_init();         // initialize pwm generator for servos
void interrupt_init();         // initialize interrupt routines
void cam_string();       // extracts data from CMUcam2 T strings
void cam_value();        // extracts the next value from lineR starting with extractPos
void cam_init();         // perform CMUCam2 initialization process
void port_init();        // initialize I/O of ports
void serial_init();      // initialize serial communication
void lcd_delay();        // short delay (50000 clocks)
void lcd_init();         // sets lcd in 4 bit mode, 2-line mode, with cursor on and set to blink
void lcd_cmd();          // use to send commands to lcd
void lcd_disp();         // use to display text on lcd
void lcd_clear();        // use to clear LCD and return cursor to home position
void lcd_row(int row);   // use to put the LCD at the desired row
void lcd_int(int value); // write interger to LCD

/* IMPORTANT!

Before using this code make sure your LCD is wired up the same way mine is, or change the

code to
match the wiring of your own LCD.  My LCD is connected to PortC of the At-Mega128 in the

following manner:

PortC bit 7   :   LCD data bit 7 (MSB)
PortC bit 6   :   LCD data bit 6
PortC bit 5   :   LCD data bit 5
PortC bit 4   :   LCD data bit 4 (LSB)

PortC bit 3   :   (not connected)
PortC bit 2   :   LCD enable pin (clock)
PortC bit 1   :   LCD R/W (read / write) signal
PortC bit 0   :   LCD RS (register select) pin

Also remember you must connect a potentiometer (variable resistor) to the vcc, gnd, and

contrast pins on the LCD.
The output of the pot (middle pin) should be connected to the contrast pin.  The other two

can be on either pin.

*/


void lcd_delay()      // delay for 1000 clock cycles
{
   long int ms_count = 0;
   while (ms_count < 500)
   {
      ms_count = ms_count + 1;
   }
}

void lcd_cmd( unsigned int myData )
{

   /* READ THIS!!!

   The & and | functions are the BITWISE AND and BITWISE OR functions respectively.  DO NOT
   confuse these with the && and || functions (which are the LOGICAL AND and LOGICAL OR functions).

   The logical functions will only return a single 1 or 0 value, thus they do not work in this scenario
   since we need the 8-bit value passed to this function to be preserved as 8-bits
```

```
   */


   unsigned int temp_data = 0;

   temp_data = ( myData | 0b00000100 );           // these two lines leave the upper nibble as-is, and set
   temp_data = ( temp_data & 0b11110100 );  // the appropriate control bits in the lower nibble
   PORTC = temp_data;
   lcd_delay();
   PORTC = (temp_data & 0b11110000);              // we have written upper nibble to the LCD

   temp_data = ( myData << 4 );                   // here, we reload myData into our temp. variable and shift the bits
                                                            // to the left 4 times.  This puts the lower nibble into
the upper 4 bits

   temp_data = (temp_data & 0b11110100);   // temp_data now contains the original
   temp_data = (temp_data | 0b00000100);   // lower nibble plus high clock signal

   PORTC = temp_data;                    // write the data to PortC
   lcd_delay();
   PORTC = (temp_data & 0b11110000);      // re-write the data to PortC with the clock signal low (thus creating the
falling edge)
   lcd_delay();

}

void lcd_disp(unsigned int disp)
{

   /*

   This function is identical to the lcd_cmd function with only one exception.  This least significant bit of
   PortC is forced high so the LCD interprets the values written to is as data instead of a command.

   */

   unsigned int temp_data = 0;

   temp_data = ( disp & 0b11110000 );
   temp_data = ( temp_data | 0b00000101 );
   PORTC = temp_data;
   lcd_delay();
   PORTC = (temp_data & 0b11110001);
   lcd_delay();                   // upper nibble

   temp_data = (disp << 4 );
   temp_data = ( temp_data & 0b11110000 );
   temp_data = ( temp_data | 0b00000101 );
   PORTC = temp_data;
   lcd_delay();
   PORTC = (temp_data & 0b11110001);
   lcd_delay();                   // lower nibble

}


void lcd_init()
{
   lcd_cmd(0x33);      // writing 0x33 followed by
   lcd_cmd(0x32);      // 0x32 puts the LCD in 4-bit mode

   lcd_cmd(0x28);      // writing 0x28 puts the LCD in 2-line mode

   lcd_cmd(0x0F);      // writing 0x0F turns the display on, curson on, and puts the cursor in blink mode

   lcd_cmd(0x01);      // writing 0x01 clears the LCD and sets the cursor to the home (top left) position

   //LCD is on... ready to write

}

void lcd_string(char *a)
{

   /*

   This function writes a string to the LCD.  LCDs can only print one character at a time so we need to
   print each letter or number in the string one at a time.  This is accomplished by creating a pointer to
   the beginning of the string (which logically points to the first character).  It is important to understand
   that all strings in C end with the "null" character which is interpreted by the language as a 0.  So to print
   an entire string to the LCD we point to the beginning of the string, print the first letter, then we increment
   the pointer (thus making it point to the second letter), print that letter, and keep incrementing until we reach
   the "null" character".  This can all be easily done by using a while loop that continuously prints a letter and
   increments the pointer as long as a 0 is not what the pointer points to.

   */

   while (*a != 0)
   {
      lcd_disp((unsigned int) *a);   // display the character that our pointer (a) is pointing to
      a++;                     // increment a
   }
   return;

}
```

```
void lcd_int(int value)
{

   /*
   This routine will take an integer and display it in the proper order on
   your LCD.  Thanks to Josh Hartman (IMDL Spring 2007) for writing this in lab
   */

   int temp_val;
   int x = 10000;         // since integers only go up to 32768, we only need to worry about
                          // numbers containing at most a ten-thousands place

   while (value / x == 0)   // the purpose of this loop is to find out the largest position (in decimal)
   {                        // that our integer contains.  As soon as we get a non-zero value, we know
      x/=10;                    // how many positions there are int the int and x will be properly initialized to the
largest
   }                                  // power of 10 that will return a non-zero value when our integer is divided by
x.

   if (value==0) lcd_disp(0x30);

   else while (x >= 1)           // this loop is where the printing to the LCD takes place.  First, we divide
   {                          // our integer by x (properly initialized by the last loop) and store it in
      temp_val = value / x;     // a temporary variable so our original value is preserved.Next we subtract the
      value -= temp_val * x;    // temp. variable times x from our original value.  This will "pull" off the most
      lcd_disp(temp_val+ 0x30); // significant digit from our original integer but leave all the remaining digits
alone.
                                        // After this, we add a hex 30 to our temp. variable because ASCII values for
integers
      x /= 10;               // 0 through 9 correspond to hex numbers 30 through 39.  We then send this value to the
   }                         // LCD (which understands ASCII).  Finally, we divide x by 10 and repeat the process
                                  // until we get a zero value (note: since our value is an integer, any decimal
value
   return;                        // less than 1 will be truncated to a 0)

}

void lcd_clear()      // this function clears the LCD and sets the cursor to the home (upper left) position
{
   lcd_cmd(0x01);

   return;
}

void lcd_row(int row)   // this function moves the cursor to the beginning of the specified row without changing
{                          // any of the current text on the LCD.

   switch(row)
   {

      case 0: lcd_cmd(0x02);
      case 1: lcd_cmd(0xC0);

   }

   return;
}


//////// Define globals //////////////
// used for soft calibrating servos
#define servo1Neutral 1497
#define servo2Neutral 1498

// used for transmitting
int lineTPos = 0;
char *lineT;

// used for receiving
int lineRPos = 0;
int lineRChoose = 0;
char lineR[30];
char linePrint[30] = "Nothing Here                 ";

// used for extracting cam data
int valueTemp, ballX, ballY, ballPix, ballCon;
int extractPos;

// used for smoothing servos
int servo1 = servo1Neutral;
int servo2 = servo2Neutral;
int servo3 = 1500;

// used for sonar data capturing
int sonar1;                        // reading of sonar 1
int sonar2;                        // reading of sonar 2
int sonarRead = 0;         // tells the adc interrupt routine which value it's writing to
int sonarState = 1;

// used for filtering sonar data
int sonar1Filt = 0;
int sonar1Mass = 0;
int sonar2Filt = 0;
```

```
int sonar2Mass = 0;

// used for general operations
int state = 0;      // 0=initializing, 1=scan, 2=avoid, 3=approach, 4=grab, 5=reverse, 6=launch
int ballHold = 0;   // 0 = not holding ball, 1 = have ball & looking for base
int stuckCount = 0;           // increments when not much movement is occuring in state 2
int startLoop = 1;                  // controls the flow of the main program while loop
int newTalk = 0;              // holds the value of the new string that needs to be printed
int oldTalk = 0;              // holds the value of the current printed string
int blind = 0;                // forces blindness for the first 80 degrees of a reverse

////////////////////////////////////

//////// Main routine ////////////////
int main (void) {

          adc_init();                           // initialize A/D conversions

          port_init();                  // set port directions

          lcd_init();     // set lcd in 4 bit mode, 2-line mode, with cursor on and set to blink

          serial_init();                // initializes and activates interrupt routines

          pwm_init();                           // initializes pwm generators

          interrupt_init();             // initializes and activates interrupt routines

          cam_init();

          state = 1;
          newTalk = 1;
          PORTD &= 0b01011111;          // start charging capacitor
          PORTD |= 0b01010000;

          while (1) {                                                   // runs about once every 15ms

                    while (startLoop == 0) ;                // do nothing until startLoop = 1;
                    startLoop = 0;

                    if (newTalk != oldTalk) {
                              oldTalk = newTalk;
                              lcd_clear();
                              if (newTalk == 1) {
                                        lcd_string("  *Sniff*");
                                        lcd_row(1);
                                        lcd_string("       *Sniff*");
                              }
                              else if (newTalk ==2) {
                                        lcd_string(" *Pant* *Pant*");
                                        lcd_row(1);
                                        lcd_string("  *Pant* *Pant*");
                              }
                              else if (newTalk ==3) {
                                        lcd_string(" Grrrrrr...");
                              }
                              else if (newTalk ==4) {
                                        lcd_string("  BARK!");
                                        lcd_row(1);
                                        lcd_string("      BARK!");
                              }
                              else if (newTalk ==5) {
                                        lcd_string("  *Whine*");
                              }
                              else if (newTalk ==6) {
                                        lcd_string(" Rruff");
                                        lcd_row(1);
                                        lcd_string("      Rruff");
                              }
                    }

                    if (state == 1) scan();
                    else if (state == 4) grab();
                    else if (state == 5) reverse();
                    else if (state == 6) launch();

          }

          return 0;

}

SIGNAL(SIG_UART0_DATA) {
          if (lineTPos < 15) {
                    UDR0 = lineT[lineTPos];
                    if (lineT[lineTPos] == 13 || lineTPos == 14) {    // if character is carriage return
                              UCSR0B &= 0b11010111;                   // then turn off interrupt for transmission
                    }
                    lineTPos++;
          }
}

SIGNAL(SIG_UART0_RECV) {
          if (lineRPos < 30) {
                    lineR[lineRPos] = UDR0;
```

```
                    if (lineR[lineRPos] == 13 || lineRPos == 29) {    // if character is carriage return
                              if (lineR[0] == 'T') {
                                        cam_string();
                              }
                              lineRPos = 0;
                    }
                    else lineRPos++;
          }
}

SIGNAL (SIG_ADC) {
          if (sonarRead == 0) {
                    sonar1 = ADCL;                // read the lowest 8 bits
                    filter1();
          }
          if (sonarRead == 1) {
                    sonar2 = ADCL;                // read the lowest 8 bits
                    filter2();
          }
          int temp;                               // discard the upper 2 bits
          temp = ADCH;                            // ADC will not update again until ADCH is read
          if (state == 2) avoid();
          else if (state == 3 && (sonar1Filt < 25 || sonar2Filt <25)) avoid();
}

SIGNAL (SIG_OVERFLOW0) {      // used to control main program while loop
          startLoop = 1;                          // start another loop
          TCNT0 = 256-216;                        // this will cause an interupt about every 15 ms
}

SIGNAL (SIG_OVERFLOW2) {      // use for smoothing pwm
          if (OCR1A < servo1-3) OCR1A += 3;
          else if (OCR1A > servo1+3) OCR1A -= 3;
          else OCR1A = servo1;

          if (OCR1B < servo2-3) OCR1B += 3;
          else if (OCR1B > servo2+3) OCR1B -= 3;
          else OCR1B = servo2;

          if (OCR1C < servo3-1) OCR1C += 1;
          else if (OCR1C > servo3+1) OCR1C -= 1;
          else OCR1C = servo3;

          TCNT2 = 0xE0;                           // this will cause an interupt about every 2.2 ms
}

SIGNAL (SIG_OVERFLOW3) {
          if (sonarState == 1) {
                    sonarState = 2;                         // 2) deactivate sonar 1, wait 100us
                    PORTA &= 0b10111111;          // deactivate sonar 1
                    TCNT3H = 0xFF;                          // initialize counter to 2 * 1024 = 2K
                    TCNT3L = 0xFE;                          // this will give 68ns * 2K = 140us second delay
          }
          else if (sonarState == 2) {
                    sonarState = 3;                         // 3) activate sonar 2, start reading sonar 1, wait 100 us
                    PORTA |= 0b10000000;          // activate sonar 2
                    sonarRead = 0;                          // set up reading of sonar 1, analog channel 0
                    ADMUX = 0b01000000;           // 5V reference, select channel 0 (pin F0)
                    ADCSRA |= 0b01000000;         // start a/d conversion
                    TCNT3H = 0xFD;                          // need 717 counts for 50 ms so set TCNT to 256^2 - 717
                    TCNT3L = 0x33;                          //  = 64819 or 0xFD33
          }
          else if (sonarState == 3) {
                    sonarState = 4;                         // 4) deactivate sonar 2, wait 100us
                    PORTA &= 0b01111111;          // deactivate sonar 2
                    TCNT3H = 0xFF;                          // initialize counter to 2 * 1024 = 2K
                    TCNT3L = 0xFE;                          // this will give 68ns * 2K = 140us second delay
          }
          else if (sonarState == 4) {
                    sonarState = 1;                         // 1) activate sonar 1, start reading sonar 2, wait 100 us
                    PORTA |= 0b01000000;          // activate sonar 1
                    sonarRead = 1;                          // set up reading of sonar 2, analog channel 1
                    ADMUX = 0b01000001;           // 5V reference, select channel 1 (pin F1)
                    ADCSRA |= 0b01000000;         // start a/d conversion
                    TCNT3H = 0xFD;                          // need 717 counts for 50 ms so set TCNT to 256^2 - 717
                    TCNT3L = 0x33;                          //  = 64819 or 0xFD33
          }
}

void scan() {                 // scan for the ball

          int random;
          random = (sonar1Filt + sonar2Filt)/10;                    // add together the sonars and
          random = (sonar1Filt + sonar2Filt) - random * 10; // extract the ones digit

          if (random < 5) {
                    servo1 = servo1Neutral + 250;           // turn left
                    servo2 = servo2Neutral + 250;
          }
          else {
                    servo1 = servo1Neutral - 250;           // turn right
                    servo2 = servo2Neutral - 250;
          }

          random = (sonar1Filt + sonar2Filt)/10;                    // add together the sonars and
```

```
		random = (sonar1Filt + sonar2Filt) - random * 10; // extract the ones digit

		int turnCount = 4000 + random * 400;	// 4000 is a full 360
		while (turnCount > 0 && state == 1) {		// if a ball is detected this will break
			turnCount--;
			lcd_delay();
		}

		if (turnCount <= 0) {
			state = 2;				// if no ball detected then just wander
		}
}

void reverse() {	// do a 180
		int random;
		random = (sonar1Filt + sonar2Filt)/10;			// add together the sonars and
		random = (sonar1Filt + sonar2Filt) - random * 10; // extract the ones digit

		if (random < 5) {
			servo1 = servo1Neutral + 250;		// turn left
			servo2 = servo2Neutral + 250;
		}
		else {
			servo1 = servo1Neutral - 250;		// turn right
			servo2 = servo2Neutral - 250;
		}

		int turnCount = 2000;
		while (turnCount > 0) {
			if (turnCount <= (2000-800)) {			// if robot has turned for more than 80 degrees
				blind = 0;					// start looking for ball
			}
			turnCount--;
			lcd_delay();
		}

		if (turnCount <= 0) {
			state = 2;				// if no ball detected then just wander
		}
}

void launch() {		// launch the ball into the air
		int i;

		servo1 = servo1Neutral;		// stop
		servo2 = servo2Neutral;

		PORTD &= 0b10101111;		// disconnect launcher from platform
		PORTD |= 0b10100000;

		for (i = 0; i < 500; i++) {
			lcd_delay();
		}

		PORTA &= 0b11011111;		// engage solenoid
		PORTA |= 0b00010000;

		for (i = 0; i < 200; i++) {
			lcd_delay();		//delay to read LCD (humans reading)
		}

		PORTA &= 0b11101111;		// disengage solenoid
		PORTA |= 0b00100000;

		for (i = 0; i < 500; i++) {
			lcd_delay();
		}

		PORTD &= 0b01011111;		// start recharging capacitor
		PORTD |= 0b01010000;

		blind = 1;
		state = 5;
		ballHold = 0;
}

void grab() {
		int i;

		servo1 = servo1Neutral;
		servo2 = servo2Neutral;
		servo3 = 1000;

		for (i = 0; i < 1000; i++) {
			lcd_delay();
		}

		servo1 = servo1Neutral - 60;	// robot will naturally turn a little to the left otherwise
		servo2 = servo2Neutral + 50;

		for (i = 0; i < 1000; i++) {
			lcd_delay();		//delay to read LCD (humans reading)
		}

		servo3 = 2300;
```

```
          for (i = 0; i < 500; i++) {
                    lcd_delay();        //delay to read LCD (humans reading)
          }

          servo1 = servo1Neutral;
          servo2 = servo2Neutral;

          for (i = 0; i < 2000; i++) {
                    lcd_delay();        //delay to read LCD (humans reading)
          }

          servo3 = 1500;

          state = 1;
          newTalk = 1;
          ballHold = 1;
}

void approach() {
          if (ballX != 0 && ballY != 0 && (state == 1 || state == 2 || state == 3 || (state == 5 && blind == 0))) {
                    state = 3;
                    newTalk = 3;
                    stuckCount = 0;
                    int temp = -2 * (ballX - 50);
                    if (temp > 250) temp = 250;
                    if (temp < -250) temp = -250;
                    servo1 = servo1Neutral + temp;                                                  // this turns if
not looking at ball
                    servo2 = servo2Neutral + temp;

                    int diff = ballX - 50;
                    if (diff < 0) diff = -diff;             // get the absolute value
                    if (diff > 20) diff = 20;               // set a max limit at 20
                    servo1 -= 8 * (20 - diff);              // this moves forward if looking at ball
                    servo2 += 8 * (20 - diff);

                    if (ballHold == 0) {          // if no ball is held
                              if (ballY >= 135 && ballPix < 90) {
                                        state = 4;                    // and a ball is detected then pick it up
                                        newTalk = 4;
                              }
                              else if (ballY >= 85 && ballPix >= 90) {
                                        blind = 1;
                                        state = 5;          // and a base is detected then turn around
                                        newTalk = 5;
                              }
                    }
                    else {                                          // if a ball is being held
                              if (ballY >= 135 && ballPix < 90) {
                                        blind = 1;
                                        state = 5;                  // and a ball is detected then turn around
                                        newTalk = 5;
                              }
                              else if (ballY >= 85 && ballPix >= 90) {
                                        state = 6;          // and a base is detected then fire the ball
                                        newTalk = 6;
                              }
                    }
          }
          else {
                    if (state == 3) {
                              state = 2;
                    }
          }
}

void avoid() {
          state = 2;
          newTalk = 2;

          if (sonar2Filt <= 15) servo1 = servo1Neutral + 250;
          else if (sonar2Filt <= 25) servo1 = servo1Neutral + 250 - 25*(sonar2Filt - 15);
          else if (sonar2Filt <= 50) servo1 = servo1Neutral - 10*(sonar2Filt - 25);
          else servo1 = servo1Neutral - 250;

          if (sonar1Filt <= 15) servo2 = servo2Neutral - 250;
          else if (sonar1Filt <= 25) servo2 = servo2Neutral - 250 + 25*(sonar1Filt - 15);
          else if (sonar1Filt <= 50) servo2 = servo2Neutral + 10*(sonar1Filt - 25);
          else servo2 = servo2Neutral + 250;

          if ((servo1 >= servo1Neutral-40 && servo1 <= servo1Neutral+40) && (servo2 >= servo2Neutral-40 && servo2 <=
servo2Neutral+40)) stuckCount++;
          else stuckCount--;
          if (stuckCount < 0) stuckCount = 0;
          if (stuckCount >= 200) {
                    stuckCount = 0;
                    state = 5;
                    newTalk = 5;
          }
}

void filter1() {
          int error = 100*sonar1 - sonar1Mass;  // scale by 100 to increase precision
          sonar1Mass += error/5;
```

```
          sonar1Filt = sonar1Mass/100;            // unscale by 100
}

void filter2() {
          int error = 100*sonar2 - sonar2Mass;  // scale by 100 to increase precision
          sonar2Mass += error/5;
          sonar2Filt = sonar2Mass/100;            // unscale by 100
}

void cam_string() {
          extractPos = 0;
          cam_value();
          ballX = valueTemp;
          cam_value();
          ballY = valueTemp;
          cam_value();
          ballPix = valueTemp;
          cam_value();
          ballCon = valueTemp;
          approach();
}

void cam_value() {
          char tempC;
          int tempI = 0;
          long tenPower = 1;
          int pos = 0;
          // find the next number
          tempC = lineR[extractPos];
          while (!(tempC>47 && tempC<58) && extractPos<30) {          // while tempC is not a number and is in bounds
                    extractPos ++;                                                                                    // then
increment extractPos
                    tempC = lineR[extractPos];                                                  // and reset tempC
          }
          // find how many digits it is (tempI)
          while ((tempC>47 && tempC<58) && (extractPos+tempI)<=30) {  // while tempC is a number and is in bounds
                    tempI ++;                                                                                         // then
increment tempI
                    tenPower *= 10;                                                                                   //
increment tenPower by a power of ten
                    tempC = lineR[extractPos + tempI];                          // and reset tempC
          }
          // extract this number to valueTemp
          valueTemp = 0;
          while (pos < tempI) {                                                 // for as many digits as are in the
number
                    tenPower /= 10;                                                             // reduce tenPower by a
factor of ten
                    tempC = lineR[extractPos + pos];        // reset tempC
                    tempC -= 48;                                                                // convert tempC from ASCII
to decimal
                    valueTemp += (tenPower * tempC);        // place the value in tempC as the correct digit in
valueTemp
                    pos ++;
          }
          extractPos += tempI;
}

void port_init() {
          DDRC = 0xFF;                            // set portC to output  (could also use DDRC = 0b11111111)
          DDRB = 0xFF;                            // set portB to output for PWM generator
          DDRE = 0xFF;                            // set portE to output
          DDRA = 0b11111000;            // set portA bits 2-0 to input
          DDRD = 0b11110000;            // set portD bits 7-4 to output
          DDRF = 0b00000000;            // set port F to all input
}

void serial_init() {
          UCSR0B = 0b00000000;                    // don't enable anything
          UCSR0C = 0b00000110;                    // asynch, no parity, 1 stop bit, 8 bit characters
          UBRR0L = 23;                                      // initialize transfer speed for 38400 baud
          sei();                                            // enable all interupts
}

void pwm_init() {
          TCCR1A = 0b10101000;          // ......00:P&FC PWM // 11......,..11....:OC1A,OC1B,OC1C inv
          TCCR1B = 0b00010010;          // ...10...:P&FC PWM // .....010:bclk/8 (~8kHz)
    ICR1=20000;
          TCNT1=0x0000;
          OCR1A = servo1Neutral;                    // ~2.5v dc level, motor 1 on PB5 (OC1A)
          OCR1B = servo2Neutral;                    // ~2.5v dc level, motor 2 on PB6 (OC1B)
          OCR1C = 1500;                             // ~2.5v dc level, motor 3 on PB7 (OC1C)
}

void interrupt_init() {
          ETIMSK = 0b00000100;                    // enable T/C3 overflow interrupt

          TCCR3A = 0b00000000;          // regular counter mode, no PWM
          TCCR3B = 0b00000101;          // prescale clk/1024
          TCNT3H = 0xFD;                            // need 717 counts for 50 ms so set TCNT to 256^2 - 717
          TCNT3L = 0x33;                            //  = 64819 or 0xFD33

          TIMSK = 0b01000001;           // enable T/C2 and T/C0 overflow interrupt

          TCCR0 = 0b00000111;           // regular counter mode, no PWM, prescale clk/1024
```

```
        TCNT0 = 0x28;                           // this will cause an interupt about every 15 ms

        TCCR2 = 0b00000101;           // regular counter mode, no PWM, prescale clk/1024
        TCNT2 = 0xE0;                           // this will cause an interupt about every 2.2 ms


        sei();                                                  // enable all interupts
}

void adc_init() {
        // Note: when JTAGEN fuse is set, F4 - F7 don't work
        ADMUX = 0b01000000;           // 5V reference, select channel 0 (pin F0)
        ADCSRA |= 0b10001111;         // turn on ADC, don't start conversions
                                                                        // no free running, interupt enabled, divide
clock by 128
}

void cam_init() {
        int i, j;
        lcd_clear();
        lcd_string("Setting up");
        lcd_row(1);
        lcd_string(" CMUCam2");

        for (i = 0; i < 1000; i++) {
                lcd_delay();        //delay to read LCD (humans reading)
        }

        if (1) {
                char newLine[15] = "OM 0 195       ";
                newLine[8] = 13;    // insert carriage return
                lineT = newLine;
                lineTPos = 0;
                UCSR0B |= 0b00101000;         // turn on transmission

                for (i = 0; i < 500; i++) {
                        lcd_delay();        //delay to read LCD (humans reading)
                }
        }

        if (1) {
                char newLine[15] = "CR 18 44       ";
                newLine[8] = 13;    // insert carriage return
                lineT = newLine;
                lineTPos = 0;
                UCSR0B |= 0b00101000;         // turn on transmission
        }

        lcd_clear();
        lcd_string("Adjusting white");
        for (j = 5; j > 0; j--) {
                lcd_row(1);
                lcd_string(" balance...");
                lcd_int(j);
                for (i = 0; i < 1000; i++) {
                        lcd_delay();        //delay to read LCD (humans reading)
                }
        }

        if (1) {
                char newLine[15] = "CR 18 40       ";
                newLine[8] = 13;    // insert carriage return
                lineT = newLine;
                lineTPos = 0;
                UCSR0B |= 0b00101000;         // turn on transmission
        }

        lcd_row(1);
        lcd_string(" balance...Done");

        for (i = 0; i < 1000; i++) {
                lcd_delay();        //delay to read LCD (humans reading)
        }

        if (1) {
                char newLine[15] = "PS 4           ";
                newLine[4] = 13;    // insert carriage return
                lineT = newLine;
                lineTPos = 0;
                UCSR0B |= 0b00101000;         // turn on transmission
        }

        lcd_clear();
        lcd_string("Acquiring ball");
        for (j = 3; j > 0; j--) {
                lcd_row(1);
                lcd_string(" color in ");
                lcd_int(j);
                for (i = 0; i < 1000; i++) {
                        lcd_delay();        //delay to read LCD (humans reading)
                }
        }

        if (1) {
                char newLine[15] = "TW             ";
```

```
                newLine[2] = 13;    // insert carriage return
                lineT = newLine;
                lineTPos = 0;
                UCSR0B |= 0b00101000;         // turn on transmission
        }

        lcd_row(1);
        lcd_string(" color done");
        for (i = 0; i < 1000; i++) {
                lcd_delay();        //delay to read LCD (humans reading)
        }
        UCSR0B |= 0b10010000;                 // turn on recieve
}
```