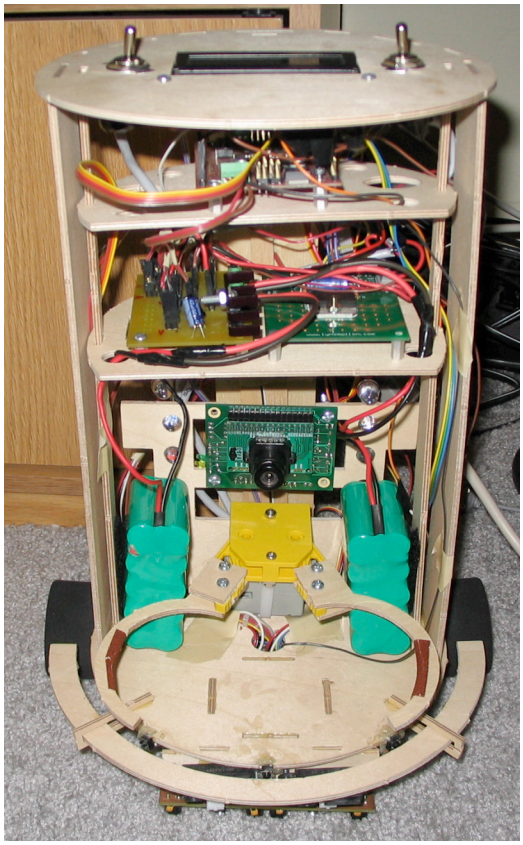
BARGHLES:
Final Report



Instructors: Dr. A. A. Arroyo & Dr. Schwartz

TAs: Sara Keen & Adam Barnett

Student: Gorang Gandhi

Date: 4/25/2006

Table of Contents:

Abstract:

Barghles is an autonomous soda retrieving robot. Based on a remote control command it follows a line, while avoiding obstacles, to bring back a soda from the kitchen. It chooses out of three different types of sodas based on color.

Executive Summary:

This paper will cover the entire development of the robot from concept to creation. It will begin with the premise for building the robot and then give a general overview of the entire system. Then it will describe the platform design and sensor suite. Additionally the different behaviors of the robot will be described, and experimental results will be given. Finally, a conclusion of the project is given, along with appropriate reference material.

Introduction:

Barghles is a robot designed to accommodate the typical laziness demonstrated by most of us in today's society. Any sort of excess physical exertion must be kept to the absolute minimal. This laziness is the catalyst for many robotic and electronic applications. One of these daunting tasks is getting up to get a drink from the kitchen. This is the problem that Barghles is designed to solve.

Barghles will take a remote control signal, travel to kitchen, and return a soda to the user. The robot will choose out of three different types of sodas based on their color. It will also have to travel on a high contrast line to travel from the user to the kitchen and back. Finally, the sodas will have to be placed on the floor because the robot will not have the capacity to open the refrigerator door.

Integrated System:

The micro controller board chosen was the MAVRIC-II board from bdmicro.com. The board uses an ATMEGA128 microprocessor. The board will control two DC motors, a motor controller, two IR range finders, bump sensors, a AVRCam, four photoreflectors, a IR can, a LCD screen, a speaker, and a servo. Power will be supplied using two 12 V 2000 mAh NiMH battery packs connected to a separate power board. The complete functional diagram is show below in Fig. 1.

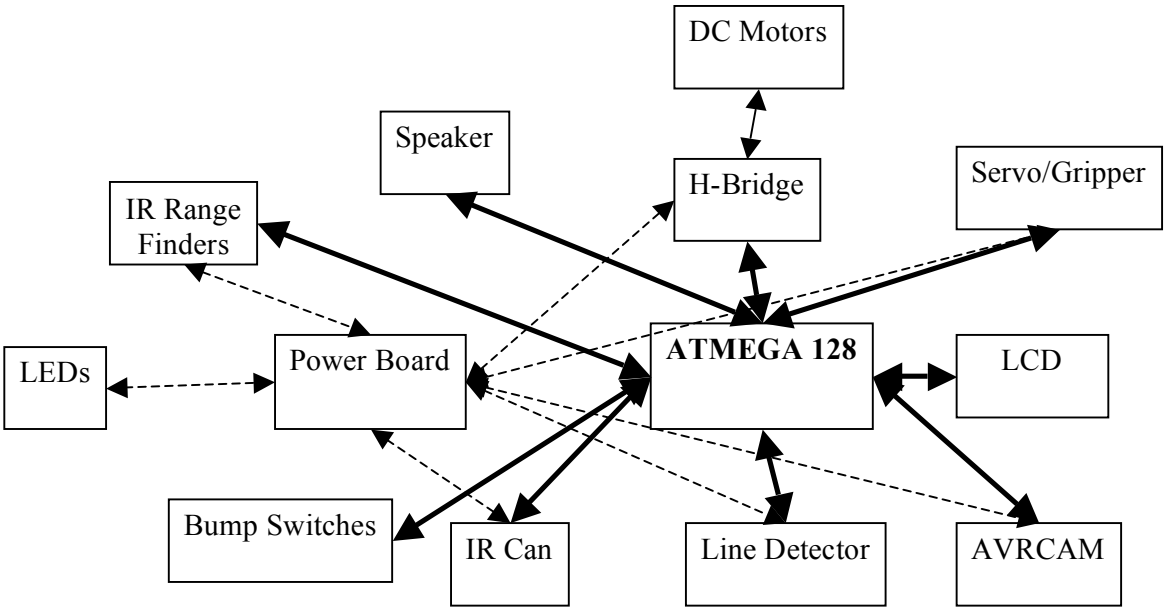


Fig. 1 Functional Diagram

The two DC motors will be driven by a dual H-Bridge motor driver to provide the robot's motion. Obstacle avoidance will be achieved using the two IR range finders and three switches coupled with a bump ring. The AVRCam will be used to distinguish colors for choosing the appropriate soda can. The photoreflectors will be used to follow a line made out of black electrical tape. The IR Can will be used to receive the remote control signals. The LCD screen will be used for debugging purposes, and will also display messages to the user. The speaker will play a different tone for each remote key pressed, and will alert the user if an obstacle is in the path or if the sequence is complete. Finally, the servo motor will be used to control the gripping motions of the robot. The power board will include a regulator for 5 volts, and connections for power switches.

Mobile Platform:

The body of the robot is cylindrical in shape to resemble that of a can of soda. It is 11" high and has 6.5" diameter. The bottom circular surface is surrounded by a thin 0.5" bump ring so the robot can detect when it runs into objects. The motors are placed towards the back of the robot so that it will not tip after grabbing the soda can. Also, a caster is placed in the front of the robot to increase stability. The servo gripper is attached to a camera and LED holder. The gripping mechanism grabs the cans at about 2" from the bottom of the soda can. Also, custom fingers were made and attached to the gripper to fit snugly around a soda can when the gripper is fully closed. A 2.1 inch clearance was needed at the bottom of the robot for the line following sensors. These sensors were placed at the front of the robot to enable the smoothest line following. Additionally, two IR range finders are mounted on the bottom of the robot. The bump ring and IR range finders are placed on the bottom so that short objects can be detected accurately.

Actuation:

There are two types of actuation including one for motion and one for grabbing the soda can. Motion is achieved using a Dual H-Bridge motor driver and two DC gear head motors. These motors rotate at 120 rpm and can produce 123 oz-in of torque. After referencing the Snackbot's final report, I knew that the robot would need at least 100 oz-in of torque to carry a can of soda. Also, the stall current of these motors are 1.5 A, and the motor driver can handle up to 2 A. Thus, the motor drivers meet their current requirements. Additionally the motor drivers can handle up to 12 V, which is what the

motors operate at. The motor controller has three inputs: A(enable), A(+), and A(-) to determine the direction and speed of the motor. See Fig. 2 below for the truth table.

A Enable	A (+)	A(-)	Motor Status
L	X	X	Power Off
H	L	L	Stop (Brake)
H	H	L	Rotate CW (Fast)
H	L	H	Rotate CCW (Fast)
H	H	H	Stop (Brake)
P	H	L	Rotate CW (Slow)
P	L	H	Rotate CCW (Slow)

H = +5V, L = 0 V, P = Pulse, X = Don't Care

Fig. 2

I dedicated PORTB (pin1-4) and PORTE (pin3,4) of the MAVRIC board to control these inputs. As can be seen by the truth table, if zero volts is applied to the A(enable) pin then the motor will stop. Conversely, if 5 volts is applied to the pin then the motor will run at full speed. Finally, if a pulse voltage between 0 and 5 volts is applied, then the motor will run at a speed determined by the average voltage of the wave form. Thus, a PWM signal can be sent to the A(Enable) input to regulate the speed of the motors. I used the output compare feature of the ATMEGA processor to send this PWM signal. I used a 50 Hz PWM signal to match that needed by the servo motor. I created defined constants in my code of FORWARD, REVERSE, STOP, LEFT, and RIGHT to be applied to port connected to the motor driver. I also defined a motor ramping function to allow for smoother motion.

The second type of actuation includes that of the servo motor used for the gripping. I used a servo/gripper kit (JM-GRP-01) attached to custom made grippers that fit perfectly around a soda can. This servo required a 50 Hz PWM signal, with a pulse width of 1.375 ms to nearly close the gripper, and 2 ms to open it completely.

Sensors:

IR Can:

The first sensor needed is the IR Sharp can used for detecting the remote control signal. General remotes send out an infrared serial data stream utilizing a 40 kHz modulating square wave. The signal is modulated because of ambient infrared radiation that could interfere with the signal. I also surrounded the IR Can with heat shrink to block out ambient interference. This is especially prevalent with fluorescent lighting. The IR Can consists of a sensitive transistor and demodulator. The IR Can outputs a simple demodulated on/off data stream. This data stream is connected to PORTD (pin 0) of the MAVRIC board. The signal is composed of on, off, and a start bit. I experimented with remotes and choose one that had a constant pulse width and a distinguishable start bit.

IR Range Finders:

Obstacle avoidance will be accomplished using two Sharp GP2D12 IR sensors, and three bump switches. The IR sensors will be arranged in the front of the robot 0.7" apart from each other. They are placed slightly angled inward at a ten degree angle from horizontal. They were placed on the bottom of the robot so that short objects will be detected, and so they will never have to be turned off during operation.

The IR sensors have three pins: power, ground, and output signal. The output signal is analog voltage between 0 and 5 V. The higher the intensity of reflected IR light, the higher the analog voltage and the closer the detected object. The IR sensors are connected to pins 0,6 on PORTF of the MAVRIC board. The analog to digital conversion takes the analog voltages and converts them to a 10 bit binary number.

Bump Switches:

The three bump switches are connected to a 0.5" thin bump ring surrounding the front half of the robot. When the robot comes in contact with an object the active low switch is closed and is tied to ground. The pins connected to the switches (PORTG pin0-2) are pulled up to 5V by enabling the pull up resistors of the ATMEGA128. Thus, the pins will be high when the robot has not hit an object, and low when the object is hit.

Photoreflectors:

Line following is accomplished using four Hamamatsu photoreflector sensors in the formation of Fig. 1, with the inner two sensors separated by 0.3" and the outer sensors 1.3" from the inner ones. These detectors include an IR emitter, photodiode, amplifier, and Schmitt trigger all in one package. Thus, not much external circuitry is needed. The sensor outputs high if a no line is detected and low if the line is detected. The sensors are placed 1 cm from the surface with heat shrink surrounding them to block outside ambient light. The sensors utilize the circuit outlined in Fig. 2.

The robot will stay on the line based on the inner two sensors; if either sensor no longer detects the line then the robot moves to center the robot over the line. The outer two sensors will be used to detect an intersection and sharp turns. If both outer sensors detect a line at the same time then it must be an intersection.

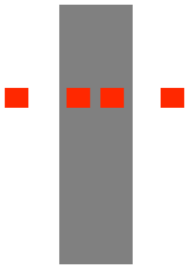


Fig. 1: Arrangement of photoreflectors

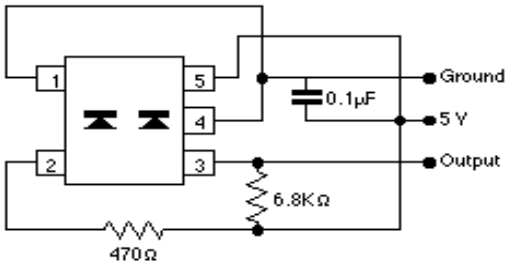


Fig. 2: Photoreflector circuits

AVRCAM:

The final sensor used will be the AVRCam to distinguish various colors. This camera can detect up to eight different colors. Thus, each type of soda will be detected by its unique color. The AVRCam will communicate serially with the ATMEGA128 UART1 at a baud rate of 115 Kbps at 30 frames/sec. The serial data uses 8 data bits, 1 stop bit, and no parity. It can transmit using either TTL or RS-232 voltage levels, but I will use RS-232 to communicate with the MAVRIC board. If the camera is set to Color Tracking mode it will continually send hex data through its serial port concerning the current viewed image. This is accomplished by sending "ET/r" to the camera. The bytes are sent in the format of Fig. 3 below. Based on byte 2,7,12,etc of the data, the color of the object tracked, the robot can determine which soda can it is facing. Twenty tracking samples are read and then averaged to increase the accuracy of color detection. This helped when a soda can had multiple colors on it.

Mnemonic	Enable color-blob tracking
OpCode	ET
Payload	<none>
Response	ACK – if command successfully received NCK – if command not successfully received
Supplemental Response Packets	<The camera begins to generate color-tracking packets in **hex** to indicate what color-blobs are found in each frame. There are NO spaces between bytes in a color-tracking packet. The current color map will be used to map the sampled pixel values into actual colors.> Byte 0: 0x0A – Indicating the start of a tracking packet Byte 1: Number of tracked objects (0x00 – 0x08 are valid) Byte 2: Color of object tracked in bounding box 1 Byte 3: X upper left corner of bounding box 1 Byte 4: Y upper left corner of bouding box 1 Byte 5: X lower right corner of boudning box 1 Byte 6: Y lower right corner of boudning box 1 Byte 7: Color object tracked in bound box 2 …. Byte x: 0xFF (indicates the end of line, and will be sent after all tracking info for the current frame has been sent)

Fig. 3: Format of AVRCam data in Color Tracking Mode

The camera also includes AVRCam View software which can be run on any PC. In this software the user can define RGB boundary values for each desired image. The decided upon RGB values are summarized in Table 2,3,4. The camera can store up to eight RGB ranges in memory, which index number determine byte 2 of Fig. 3. The valid color index values are 0-7, which wasn't completely evident be reading the camera's manual. The software also includes options for Florescent Light Filtering, Auto White Balance, and Auto Adjusting modes, which help eliminate some of the excess flooding of red in the camera display. Six white LEDs are also incorporated to have additional control over the lighting conditions of the camera.

Coke		
Red	Green	Blue
144 - 240	16 - 64	16 - 64
Pepsi		
Red	Green	Blue
16 - 80	16 - 48	48 - 240
Mt. Dew		
Red	Green	Blue
16 - 80	112 - 240	16 - 48

Table 2: RGB values for my room

Coke		
Red	Green	Blue
144 - 240	16 - 64	16 - 64
Pepsi		
Red	Green	Blue
16 - 80	16 - 48	112 - 240
Mt. Dew		
Red	Green	Blue
16 - 80	80 - 240	16 - 96

Table 3: RGB values for IMDL room

Coke		
Red	Green	Blue
144 - 240	16 - 64	16 - 64
Pepsi		
Red	Green	Blue
16 - 80	16 - 80	80 - 240
Mt. Dew		
Red	Green	Blue
16 - 80	112 - 240	16 - 96

Table 4: RGB values for NEB rotunda

AVRCAM vs CMUCAM:

The following table provides a superficial comparison of the AVRCAM and CMUCAMs. Note that I do not have any actual experience with the CMUCAM and that my data is based on what I could find online.

	AVRCAM:	**CMUCAM1:**	**CMUCAM2:**
Price	$99 (unassembled)	$109 (assembled)	$169 (assembled)
Camera	OV6620	OV6620	OV7620
Current	57 mA	200 mA	200 mA
Frames/Sec	30	17	50
Size	2.4”x1.9”	2.25”x1.75”	?
Number of Tracked Objects	8	1	1
Baud Rates (kbps)	115.2	9.6,19.2,38.4,115.2	1.2,2.4,4.8,9.6,19.2,38.4,57.6,115.2
Resolution	88x144 pixels	80x144 pixels	176x255 pixels
Servo Outputs	0	1	5
Online Forum	Yes	On Acroname	On Acroname
Open Source	Yes	Supposedly	Supposedly

Table 1: AVRCAM vs. CMUCAM comparison

Some positive aspects of the AVRCAM are that the online forum is very useful, and most of the questions are answered by the maker of the camera. I posted several questions and received answers quite promptly. Also, all of the software is completely open source and the camera utilizes an ATMEGA8 processor. I am not sure how the support for the CMUCAM is, nor do I know if it is indeed open source. Both cameras

have some trouble with red flooding the images. The AVRCAM can track up to eight objects at the same time, but does not return the mean RGB values of an object and does not have a direct servo output. Also, it is hard to adapt to various ambient lighting conditions without changing the RGB ranges of the colors to be detected. Thus, each time I changed environments I had to open up the camera's PC software and change the RGB ranges.

Behaviors:

The six behaviors include: Receiving a signal command from a remote, avoiding obstacles, following a line, distinguishing a color of a soda can, gripping the chosen soda can, and playing tones with the speaker.

The remote key pressed will determine which soda is retrieved. Remote key 1 is for Coke, key 2 for Pepsi, and key 3 for Mt. Dew. I detected the start bit of the remote data stream by waiting for a falling edge and then looking for a 2-4 ms time period before the signal went high again. Next, each pulse that was greater than 1 ms was stored as 1 bit, and anything else was stored as a 0 bit. Finally, after four bits were stored, this number was converted to the binary number of the button pressed on the remote. Additionally, I took ten samples each time a remote key was pressed and averaged the result.

Obstacle avoidance was accomplished using the IR range finders and bump ring. When an object is detected by IR or if hit by the bump ring, the robot will back up and decelerate, and display an error message to the user. This is accomplished by checking if

either of the IR detectors are over a certain threshold or if any of the switches are pressed. The robot does not proceed forward until the object is removed from its path.

Line following was accomplished using the four photoreflectors mounted 1 cm from the surface towards the front of the robot. The algorithm simply turned left if either of the right sensors were no longer above a line, and turned right if the left sensors were no longer above the line. Also, varying speeds and motor ramping was implemented for smoother motion. Finally, if all four of the sensors detected a line, the robot stops. Functions were also implemented to turn onto a line and off of line using these sensors.

Color detection was accomplished using the AVRCAM. The camera was put into tracking mode by sending "ET/r" to the camera via UART1. Then 50 samples were read into a temp character array. Next, this temp array was parsed until 0x0A was found. This indicates the start of a tracking packet. Additionally, the next 13 bytes were copied to another char array called AVRCAMdata. Finally, byte 2 of AVRCAMdata was read in, which is the index of the tracked color, and if it was one of the valid color indexes, 0-2, the value was added to a global variable called AVRCAM_AVG. Twenty samples were taken in this manner and the color detected was determined by the value of AVRCAM_AVG. If the value of AVRCAM_AVG was less than 6, then it was a Coke can, if it was between 7 and 31 then it was a Pepsi, and if it was between 32 and 40 it was a Mt. Dew.

The next behavior was grabbing the soda can which was accomplished through the servo/gripper kit. A 50 Hz PWM signal was sent to the servo through pin 5 of PORTE. A pulse width of 1.370 ms was used to close the gripper and a 1.7 ms pulse was used to partially open the gripper. Finally, the cans were mounted on cardboard holders

and included a line intersection below them so that the robot could stop in the correct position to grab the soda can.

The last behavior was playing tones of the speaker. When a key was pressed a different tone at set frequency was played for each key pressed. Additionally, when an object was detected by the IR range finders a regular beeping pattern was played. Finally, when Barghles returned with the soda can the speakers played 5 beeps before turning around and restarting the cycle. The speaker has two wires with one connected to ground and the other connected to PORTB (pin5), which is Output Compare 1A (OC1A). The speaker played a tone simply by toggling a bit at a certain frequency. This was accomplished using a PWM signal that toggled OC1A each time TCNT1 reached the value of ICR1. Thus, ICR1 simply had to be set to a different value for each desired frequency, and was set to zero to stop playing the tone.

Experimental Layout and Results:

Experiment 1: Varying Ambient Light:

IR Can:

With the IR Can I simply varied the ambient lighting and noted whether it affected the accuracy of detecting the key pressed. I set a test program that displayed the key pressed to the LCD. I observed that only fluorescent lighting had an effect on the accuracy of the readings. To alleviate this problem I surrounded the IR Can with heat shrink.

IR Range Finder:

I set up a simple program to display the analog values of the two IR range finders to the LCD. I took 20 samples and then averaged them. I noticed that sensor readings only varied slightly with changes in ambient lighting.

Photoreflectors:

I set up a simple program to display the value of the pins connected to the four photoreflectors. I did notice that if too much ambient light is introduced then the readings are no longer accurate. The sensors should display a 0 if over a line and a 1 otherwise, but this was not the case if there was too much outside light. I alleviated this problem by surrounding the sensors with heat shrink to block the outside ambient light. After doing this I observed complete accuracy of results regardless of the ambient lighting condition.

AVRCAM:

In this experiment the lighting was varied between low, medium, and high using a 3 lamp light and opening the shades. Experimentation was also performed with the presence and absence of the 6 white LEDs pointing at the desired image. The soda cans experimented on were: Coke, Pepsi, Sprite, Celeste Orange Soda, and Bargs. It was concluded during this experimentation that the most consistent color is achieved when the white LEDs are not pointed directly at the object, but to the sides. Also, as indicated in the below tables, there was less variance in the main color when using white LEDs. Also, the camera performed better at lower lighting conditions when using white LEDs. The Bargs and Celeste sodas had silver and orange colored cans respectively, and ended up interfering too much with the other cans color maps. I confirmed this when tracking

the other cans with all of the color maps added. Thus, only Coke, Pepsi, and Mt. Dew/Sprite are being used. A lot of tweaking was performed with these three sodas to make sure that the color maps did not interfere with each other when in color tracking mode. The following table summarizes the RGB ranges that worked the best for Pepsi, Coke, and Sprite. The valid ranges for RGB values are 0-240.

Soda	**Red Range**	**Green Range**	**Blue Range**
Coke	240-240	16-32	16-32
Pepsi	16-48	48-64	208-240
Sprite	16-48	96-240	64-96

Table 5: RGB ranges used for Coke, Pepsi, Sprite

The following table outlines the RGB ranges obtained in various lighting conditions with the absence of white LEDS.

Coke			
	Red	Green	Blue
Low	80-144	16-48	16-48
Medium	236-240	16-40	16-32
High	240-240	16-40	16-32
Pepsi			
	Red	Green	Blue
Low	40-56	32-64	32-52
Medium	64-104	44-72	48-104
High	80-112	64-112	80-148
Sprite			
	Red	Green	Blue
Low	32-56	32-64	32-56
Medium	32-64	64-80	64-92
High	48-64	32-112	72-112

Table 6: AVRCAM RGB readings w/ No LEDS

Additionally, huge variances were observed in the RGB values of the Bargs soda can when not using white LEDs. The following table outlines the RGB ranges obtained in various lighting conditions with the use of white LEDS.

Coke			
	Red	Green	Blue
Low	64-240	16-32	16-32
Medium	240-240	16-48	16-32
High	240-240	16-48	16-32
Pepsi			
	Red	Green	Blue
Low	24-64	48-80	120-216
Medium	48-100	64-112	84-224
High	64-128	64-128	136-240
Sprite			
	Red	Green	Blue
Low	16-48	80-240	80-240
Medium	16-80	96-240	96-240
High	48-92	96-240	96-240

Table 7: AVRCAM RGB readings w/ LEDS

Experiment 2: Distance Measurements:

IR Range Finders:

I varied the distance between an object and the IR sensor and recorded the analog values. I noticed that even with 20 samples being taken that the values still varied while maintaining a constant distance. The following table and graph summarizes the results.

Distance (in)	Analog IR_L	Analog IR_R
1	175	175
2	325	255
3	532	490
4	480	509
5	415	445
6	344	366
7	310	324
8	278	285
9	240	240
10	212	226
11	204	208
12	184	197
13	174	184
14	157	168
15	145	159

Table 8: IR range finder readings

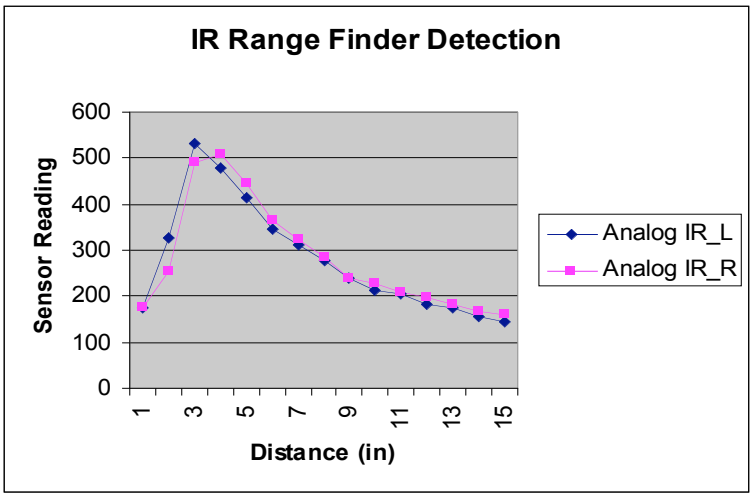


Fig 4: IR range finder reading

Photoreflectors:

This experiment consisted of varying the distance between the photoreflector and the line that it is detecting. It was found that the sensor works in the range of 0.3 - 1.5 cm. I chose to place the sensor at a distance of 1 cm from the surface of the line. It was also found that the sensor works best with no angle.

AVRCAM:

The camera images were displayed to the PC software and the distance of the object was varied. It was concluded that accurate readings for RGB values can be obtained within 15" of the camera, and there was not too much variance in RGB values with distance.

IR Can:

A test program was set up to display what key has been pressed, while the distance between the sensor and remote was varied. Accuracy of the results remained the same as the distance increased. Also, the remote did not have to be pointed directly at the sensor to function properly.

Experiment 3: Functionality:

The functionality of the AVRCAM, IR Range Finders, IR Can, and Photoreflectors were all verified will performing Experiment 1 and 2.

Bump Switches:

A simple program was set up to continually display the contents of the pins connected to the bump switches. These are pins 0-2 on PORTG. It was verified that when the switches were not pressed then the pins values were high, and when pressed the pins values were low.

Conclusion:

In summary, Barghles is an autonomous robot that retrieves soda cans for a user based on remote commands. These commands correspond to the desired soda can. The robot follows a line to the where the sodas are placed, grabs the can, and returns it to the user.

Overall I am very pleased with the results of my robot. It performed all of the tasks with great accuracy. Every so often there was a soda color detection misread, but it was rare. Of course the varying colors of each soda can made some portions of the can easier to detect than others. Adapting the robot to various ambient lighting conditions was a bit tedious with the AVRCAM, but possible. Additionally, lighting conditions flooded the camera with either red or green light, even with the use of 6 white LEDs. Also, heat shrink had to be wrapped around the IR Can to block outside ambient lighting. This was especially prevalent with fluorescent lighting. Twisting in wires caused bad

connections, and several hours of frustration. I have concluded that sloppy hardware is the easiest way to ruin your robot functioning.

If I were able to redo Barghles I would definitely make a sturdier frame. I would also make the robot more physically appealing and do a much better job wiring and soldering. Almost every problem I had was due to hardware issues. I would also have my robot actually leave the line and require the line to avoid an object, rather than simply stopping and displaying an error message. Finally, I would lift the soda can vertically to serve as a coaster for the user.

Documentation:

SnackBot Final Report:

http://www.mil.ufl.edu/courses/eel5666/papers/IMDL_Report_Summer_05/shepherd-chris/snackbot.pdf

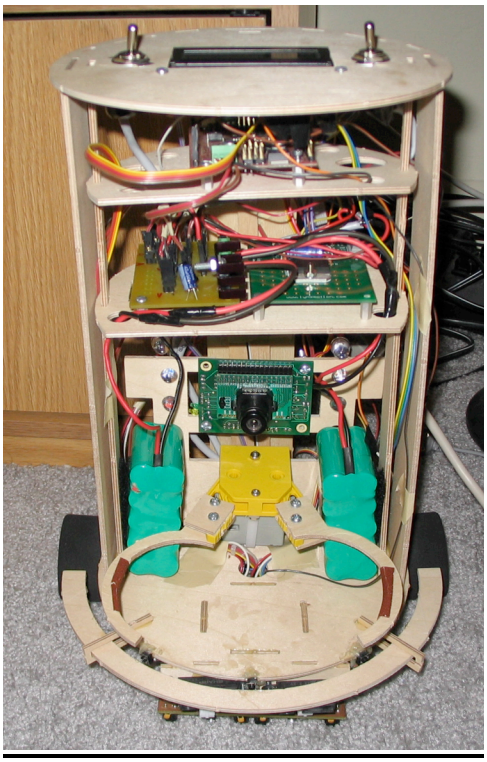
William Dubel's Line Following Report:

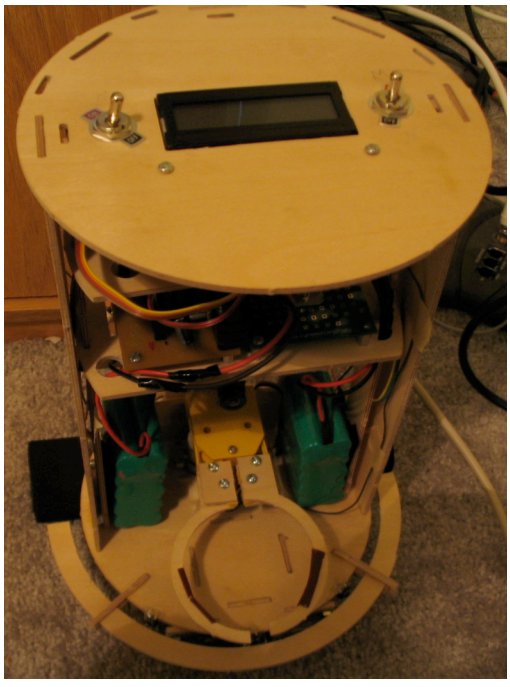
http://www.mil.ufl.edu/courses/eel5666/handouts/lt.doc

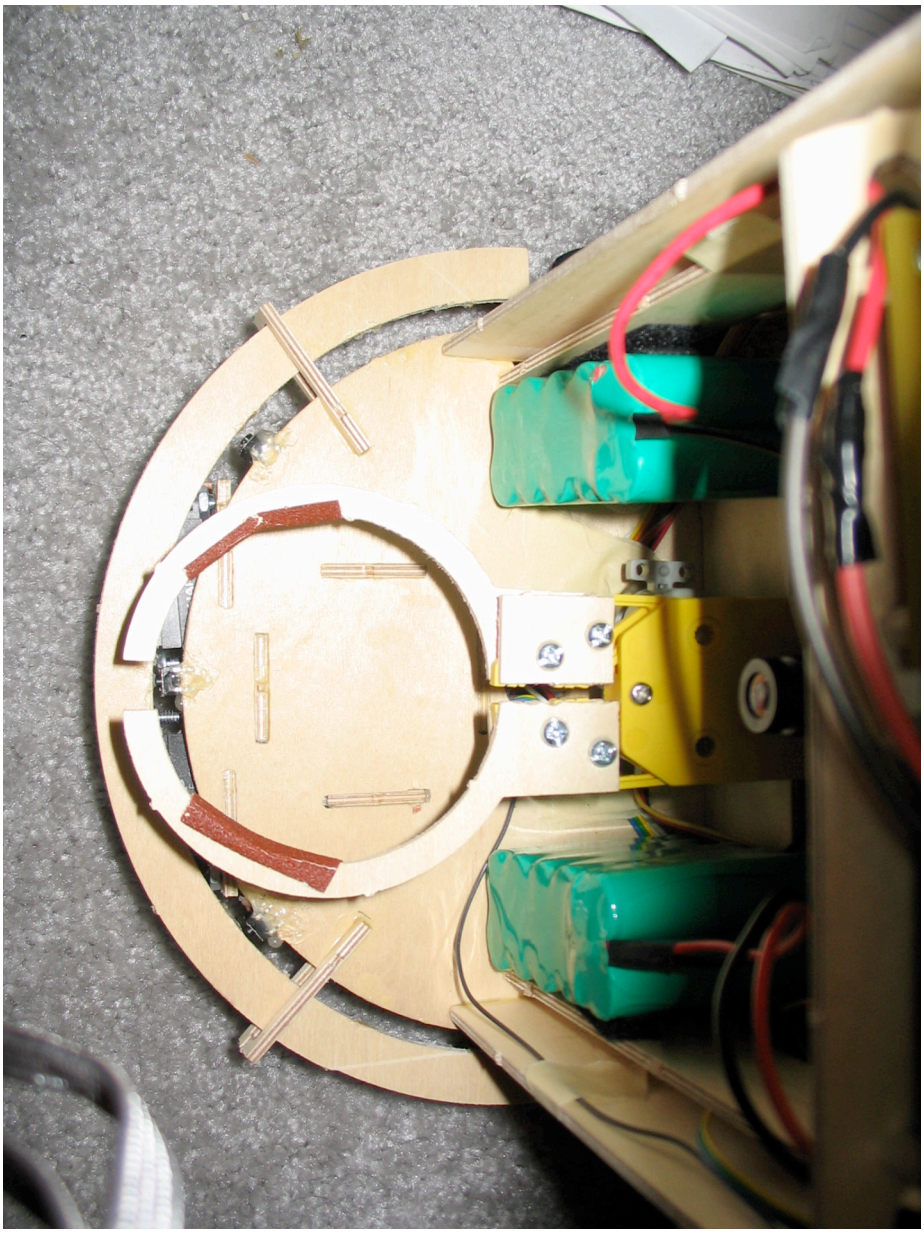
AVRCAM Manual v1.4:

http://www.jrobot.net/Download/AVRcam_Users_Manual_v1_4.pdf

Appendix A- Pictures:

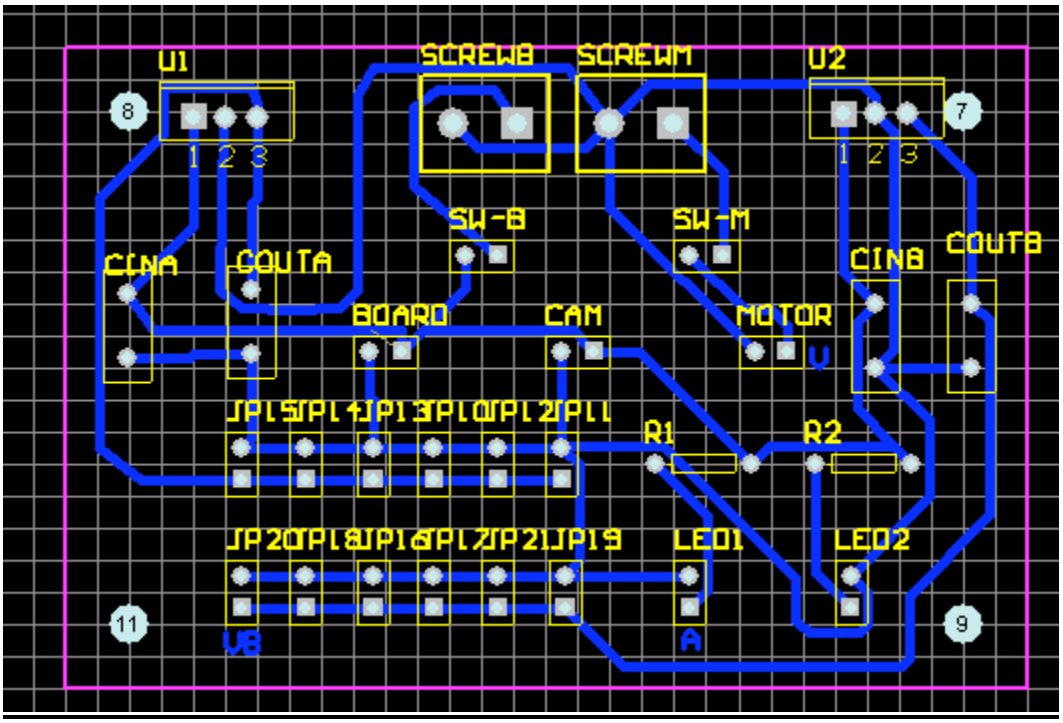




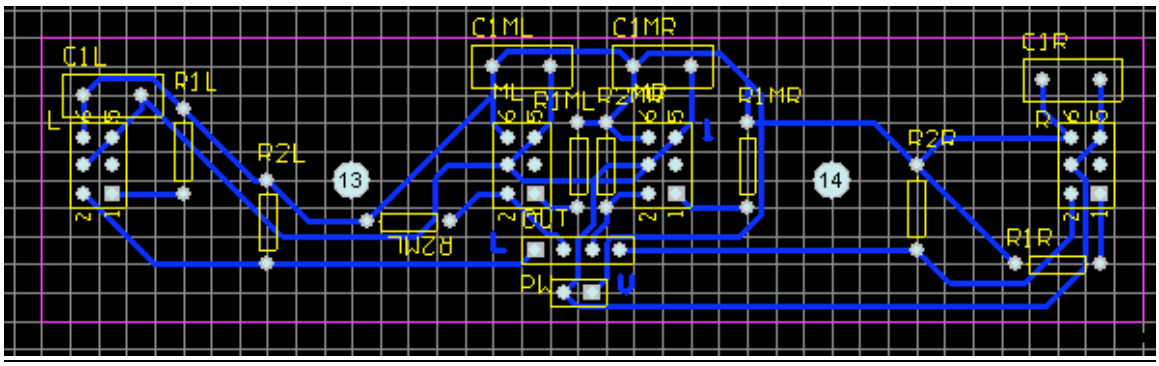


Appendix B: Boards:

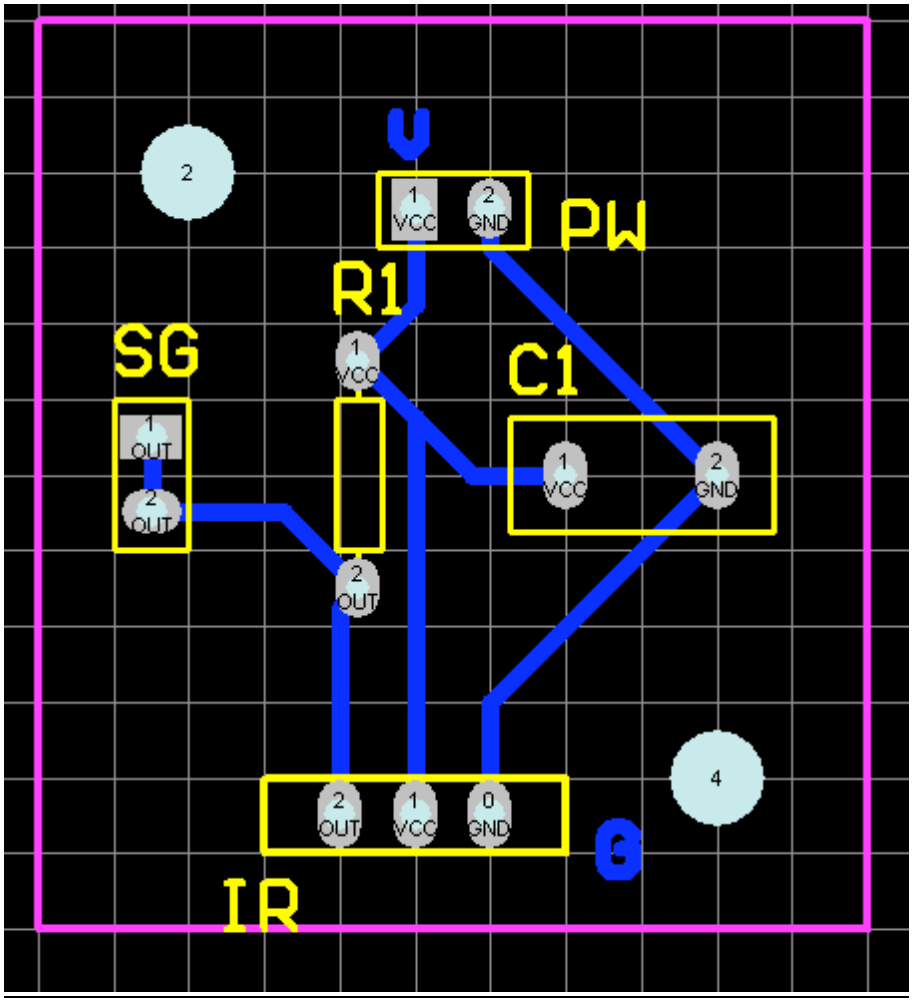
Power Board:



Line Follower Board:



IR Can Board:


2
U
1
VCC
2
GND
PW
R1
SG
1
VCC
C1
1
OUT
2
OUT
1
VCC
2
GND
2
OUT
4
2
OUT
1
VCC
0
GND
G
IR

Appendix C- Code:

```
/*****************************************************************************
 * MAIN
 * Created By: Gorang Gandhi
 * 4/24/06
 * Atmega 128
 *****************************************************************************
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>
#include <avr/pgmspace.h>
#include <stdio.h>
#include <stdlib.h>

//============================================================================
============
// PORTC- LCD (pin0-6)
// PORTF- IR Range finders (pin0,6)
// PORTA- PHOTO (pin0-3)
// PORTE- Output Compares (pin3-5)
// PORTB- Motors (pin1-4), Speaker (pin5)
// PORTD- IR Can (pin0)
// PORTG- Bump Switches (pin0-2)
// PORTD- UART1- AVRCam (pin3,2)
//============================================================================
============

// LCD Screen
//==========================================================================
#define LCD PORTC
#define LCD_DDR DDRC

// IR
//==========================================================================
=======
#define IR PORTF
#define IR_DDR DDRF

// Photoreflector
//========================================================================
#define PHOTO PORTA
#define PHOTO_DDR DDRA
#define PHOTO_PIN PINA&0x0F // Only least significant nibble

//
Motors=====================================================================
=========
// PortB- B0B1B2B3 - A+A-B+B- - A is left motor, B is right motor
#define MOTOR PORTB
#define MOTOR_DDR DDRB
#define MOTOR_L OCR3A       // For PWM of Aenable of motor driver- left motor- pin3 port E
#define MOTOR_R OCR3B       // For PWM of Aenable of motor driver- right motor- pin4 port E
```

```
#define FORWARD 0x14          // 0001 0100
#define BACKWARD 0X0A         // 0000 1010
#define LEFT 0x0C             // 0000 1100
#define RIGHT 0x12            // 0001 0010
#define STOP 0x00             // 0000 0000
#define SLOW 5000
#define MEDIUM 10000
#define FAST 15000

//
Servo==========================================================================
==========
#define SERVO OCR3C                     // For PWM of servo motor- pin5 port E
#define S_CLOSE 1370 // Closes servo gripper
#define S_OPEN 2000  // Fully opens servo gripper
#define S_MIDDLE 1700 // middle servo gripper position

// Bump
Switches=======================================================================
====
#define BUMP PORTG // Bump switches will bo on PORTG
#define BUMP_DDR DDRG
#define BUMP_PIN PING

// IR
Can============================================================================
========
#define IRCAN PORTD // Pin0
#define IRCAN_DDR DDRD
#define IRCAN_PIN PIND

//
Speaker========================================================================
==========
#define SPEAKER OCR1A // Only pin5 of PORTB
#define SPEAKER_DDR DDRB
#define SPEAKER_F ICR1


volatile uint16_t ms_count;
volatile uint8_t IR_flag;
volatile uint8_t BUMP_flag;
volatile uint8_t PHOTO_flag;
volatile uint8_t IR_output_d;
volatile uint8_t BUMP_output_d;
volatile uint8_t PHOTO_output_d;
volatile uint8_t IR_output_s;
volatile uint16_t BUMP_output_s;
volatile uint16_t PHOTO_output_s;
volatile uint8_t motor_input_d;
volatile uint16_t motor_input_s;
volatile uint8_t MODE; // MODE = 0 going to/from soda pickup; MODE = 1 picking up soda
volatile uint8_t SODA; // 1-Coke, 2-Pepsi, 3-Sprite
volatile uint8_t FIRST_PASS; // 0 if first pass, 1 if not
volatile uint8_t AVRCAM_AVG;
volatile uint8_t CAM_counter;
```

```
volatile uint8_t IR_val;
volatile uint8_t PATH; // 1-Left path, 2 Center Path, 3 Right Path


// TIMER
=============================================================================
=========
//=============================================================================
================

void ms_sleep(uint16_t ms)
// ms_sleep() - delay for specified number of milliseconds
{
            TCNT0  = 0;
            ms_count = 0;
            while (ms_count != ms)
                    ;
}

SIGNAL(SIG_OUTPUT_COMPARE0)
/*
 * millisecond counter interrupt vector
 */
{
          ms_count++;
}

void init_timer(void)
{
  /*
   * Initialize timer0 to generate an output compare interrupt, and
   * set the output compare register so that we get that interrupt
   * every millisecond.
   */
            TIFR  |= _BV(OCIE0); // Clears Interupt flag
            TCCR0  = _BV(WGM01)|_BV(CS02)|_BV(CS00); /* CTC, prescale = 128 */
            TCNT0  = 0;
            TIMSK |= _BV(OCIE0);    /* enable output compare interrupt */
            OCR0   = 125;          /* match in 1 ms */
}

// LCD
=============================================================================
===========
//=============================================================================
================
void lcd_init(void)
{
/*   Intializes the LCD
 */

            ms_sleep(20);
            LCD = 0x13;
            LCD = 0x03;
            ms_sleep(5);
            LCD = 0x13;
```

```
            LCD = 0x03;
            ms_sleep(1);
            LCD = 0x13;
            LCD = 0x03;
            ms_sleep(5);
            LCD = 0x12;
            LCD = 0x02; // 4 Bit Operation
            ms_sleep(1);

            LCD = 0x12;
            LCD = 0x02; // $28- Two Line
            LCD = 0x18;
            LCD = 0x08;
            ms_sleep(2);

            LCD = 0x10;
            LCD = 0x00; // $0F- Display, Cursor, Blink
            LCD = 0x1F;
            LCD = 0x0F;
            ms_sleep(2);

            LCD = 0x10;
            LCD = 0x00; // $01- Clear screen, cursor home
            LCD = 0x11;
            LCD = 0x01;
            ms_sleep(2);

            LCD = 0x10;
            LCD = 0x00; // $06- Increment cursor to right
            LCD = 0x16;
            LCD = 0x06;
            ms_sleep(2);

}

int lcd_char(char character)
 // Displays a character to the LCD
{
        uint8_t MS_Nibble;
        uint8_t LS_Nibble;

        MS_Nibble = (character&0xF0)>>4; // 0000xxxx- MS Nibble
        LS_Nibble = character&0x0F; // 0000xxxx- LS Nibble

        LCD = MS_Nibble|0x50;           // RS,E = 1
        LCD = MS_Nibble|0x40;  // RS = 1, E = 0
        ms_sleep(2);
        LCD = LS_Nibble|0x50;
        LCD = LS_Nibble|0x40;
        ms_sleep(2);

        return character;
}

 void lcd_string(char message[])
// Displays a string to the LCD
```

```
{
	int k;

	for(k=0;k<21;k++)
	{
		if(message[k] == '\0' || message[k] == '\n')
		{
			break;
		}

		lcd_char(message[k]);
	}

}

void lcd_clear_display(void)
// Clears the entire LCD display
{
	 LCD = 0x10;
	 LCD = 0x00; // $01- Clear screen, cursor home
	 LCD = 0x11;
	 LCD = 0x01;
	 ms_sleep(2);
}

void lcd_goto_line(int line)
// Goes to line 1 or 2 specified by line
{
	switch(line)
	{
		case(1):
		{
			LCD = 0x10;
			LCD = 0x00; // $02- cursor home
			LCD = 0x12;
			LCD = 0x02;
			ms_sleep(2);
		}
		case(2):
		{
			LCD = 0x1C;
			LCD = 0x0C; // $C0- second row
			LCD = 0x10;
			LCD = 0x00;
			ms_sleep(1);
		}
	}
}

void lcd_goto_pos(int pos)
// Goes to horizontal position. 0-19
{
	int i;

	 LCD = 0x10;
	 LCD = 0x00; // $02- cursor home
```

```
            LCD = 0x12;
            LCD = 0x02;
            ms_sleep(2);

           for(i=0;i<pos;i++)
           {
                    LCD = 0x11;
                    LCD = 0x01; // $14- shift cursor right
                    LCD = 0x14;
                    LCD = 0x04;
                    ms_sleep(1);
           }
}

void banner(void)
// Displays a beggining banner
{
           lcd_clear_display();

           printf("BARGHLES!!!!");

           ms_sleep(3000);

}

//
ADC=============================================================================
====================
//
=============================================================================
====================
void adc_init(void)
 /* initialize A/D converter
 *
 * Initialize A/D converter to free running, start conversion, use
 * internal 5.0V reference, pre-scale ADC clock to 125 kHz (assuming
 * 16 MHz MCU clock)
 */
{
  IR = 0x00;

  ADMUX = _BV(REFS0);
  ADCSR = _BV(ADEN)|_BV(ADSC)|_BV(ADFR) | _BV(ADPS2)|_BV(ADPS1)|_BV(ADPS0);
}


void adc_chsel(uint8_t channel)
 /*A/D Channel Select
 *
 * Select the specified A/D channel for the next conversion
 */
{
  ADMUX = (ADMUX & 0xe0) | (channel & 0x07);
}

void adc_wait(void)
```

```
 /* adc_wait() - A/D Wait for conversion
 *
 * Wait for conversion complete.
 */
{
  /* wait for last conversion to complete */
  while ((ADCSR & _BV(ADIF)) == 0)
    ;
}


void adc_start(void)
 /* adc_start() - A/D start conversion
 *
 * Start an A/D conversion on the selected channel
 */
{

  ADCSR |= _BV(ADIF); /* clear conversion, start another conversion */
}

uint16_t adc_read(void)
/* adc_read() - A/D Converter - read channel
 *
 * Read the currently selected A/D Converter channel.
 */
{
  return ADC;
}


uint16_t adc_readn(uint8_t channel, uint8_t num)
/*
 * adc_readn() - A/D Converter, read multiple times and average
 *
 * Read the specified A/D channel 'n' times and return the average of
 * the samples
 */
{
  uint16_t t;
  uint8_t i;

  adc_chsel(channel);
  adc_start();
  adc_wait();

  adc_start();

  /* sample selected channel n times, take the average */
  t = 0;
  for (i=0; i<num; i++)
  {
    adc_wait();
    t += adc_read();
    adc_start();
  }
```

```
  /* return the average of n samples */
  return t / num;
}

//
Motor==========================================================================
=========================
//==============================================================================
============================

void motor_init(void)
// Initializes motors: Enable OC3A, OC3B, OC3C.  3 Mhz frequency.
// ICR3 will be the TOP.  clk(io)/8.
{
        TCCR3A = 0xA8; //10101000- Clear OCA,B,C on compare match- non-inv PWM
        TCCR3B = 0x12; //00010010- Mode 8 - PWM Phase/Freq Correct, clk(io)/8
        ICR3 = 20000; // PWM = 50 Hz. 1/(16 MHz/8)*2*20000 = 1/50
        TCNT3 = 0x00; // Used to be TCNT2?

        MOTOR_L = 0; // Not moving
        MOTOR_R = 0;
        SERVO = S_MIDDLE; // sets OC3C to 1700

        PORTE = 0x00;
}

void motor_move(uint8_t direction, uint16_t speed)
// Moves in a certain direction at a certian speed.
// Directions- 1-FW,2-BW,3-L,4-R, Speeds- 0 to 20000
{
        int increment_l = 0;
        int increment_r = 0;

        if(MOTOR_L > speed)
        {
                increment_l = -1;
        }
        if(MOTOR_L < speed)
        {
                increment_l = 1;
        }
        if(MOTOR_L == speed)
        {
                increment_l = 0;
        }
        if(MOTOR_R > speed)
        {
                increment_r = -1;
        }
        if(MOTOR_R < speed)
        {
                increment_r = 1;
        }
        if(MOTOR_R == speed)
        {
```

```
                increment_r = 0;
        }

        if(direction == 1)
        {
                MOTOR = FORWARD;
        }
        if(direction == 2)
        {
                MOTOR = BACKWARD;
        }
        if(direction == 3)
        {
                MOTOR = LEFT;
        }
        if(direction == 4)
        {
                MOTOR = RIGHT;
        }

        while(MOTOR_L != speed || MOTOR_R !=speed)
        {
                if(MOTOR_L != speed)
                {
                        MOTOR_L = MOTOR_L + increment_l;
                }

                if(MOTOR_R != speed)
                {
                        MOTOR_R = MOTOR_R + increment_r;
                }

                ms_sleep(1); // 3 too long
        }

}


// Speaker
================================================================================
==================
//==============================================================================
============================

void speaker_init(void)
// Initializes speakers: Enable OC1A. clk(io)/8.
{

        TCCR1A = 0x54; //01010100- Toggle OCA on compare match
        TCCR1B = 0x1A; //00011010- Mode 12 - CTC- Clear counter on match w/ ICR1, clk(io)/8
        TCNT1 = 0x00;
        SPEAKER_F = 0x00;     // Sets ICR1 to 0
}
```

```
// IR
Can=============================================================================
========================
//=============================================================================
============================

void falling_edge(void)
// Waits for a falling edge on pin0 of PORTD
{
	int i;

	while(1)			// Loops till pin is high
	{
		if(IRCAN_PIN&0x01)
		{
			break;
		}
	}

	for(i=0;i<3;i++);

	while(1)			// Loops till pin is low
	{
		if(!(IRCAN_PIN&0x01))
		{
			break;
		}
	}

	//printf("falling edge");
}

void rising_edge(void)
// waits for a rising edge on pin0 of PORTD
{
	int i;

	while(1)			// Loops till pin is high
	{
		if(IRCAN_PIN&0x01)
		{
			break;
		}
	}

	for(i= 0;i<3;i++);

	//printf("rising edge");
}

uint8_t remote(void)
// Reads in a remote signal. Returns zero when a key is pressed.
{
	uint16_t time; // Holds the time of the pulse
	uint8_t value; // Holds the 4 bit data number
	uint8_t adder;
```

```
	int n;

	value = 0;
	adder = 0x01;

		do                                          // Gets start bit which is b/t 2 - 4 ms
		{
			falling_edge();
			TCNT0 = 0;
			ms_count = 0;
			rising_edge();
			time = ms_count;

		}while((time<2)||(time>4));

		for(n=0;n<4;n++)                    // Get the rest of the data bits
		{
			falling_edge();
			TCNT0 = 0;
			ms_count = 0;
			rising_edge();
			time = ms_count;

			if(time>0)                  // If time >= 1 ms, then place a 1
			{
				value += adder;
			}

			adder = adder<<1; // Shifts 1 to the left
		}

	value += 0x01; // The key 1 on the remote sends binary zero, so I must offset by 1.

	return value;

}

uint8_t remote_avg(void)
// Reads in a remote signal. Returns zero when a key is pressed.
{
	uint8_t k;
	IR_val = 0;

	lcd_clear_display();

	printf("Press a remote key:");

	for(k=0;k<10;k++)
	{
		IR_val = remote() + IR_val;
	}

	if(IR_val <= 15)
	{
		lcd_goto_line(2);
		printf("1 key: Coke");
```

```
                SPEAKER_F = 1136; // freq= 1760 Hz. 1/(16 MHz/8)*1136 = 1/1760
                ms_sleep(1000);
                SPEAKER_F = 0;

                SODA = 1;
                return 0;
        }

        else if(IR_val >= 16 && IR_val <= 25)
        {
                lcd_goto_line(2);
                printf("2 key: Pepsi");

                SPEAKER_F = 2025; // freq= 987.7 Hz. 1/(16 MHz/8)*2025 = 1/987.7
                ms_sleep(1000);
                SPEAKER_F = 0;

                SODA = 2;
                return 0;
        }

        else if(IR_val >= 26)
        {
                lcd_goto_line(2);
                printf("3 key: Mt. Dew");

                SPEAKER_F = 956; // freq= 2093 Hz. 1/(16 MHz/8)*956 = 1/2093
                ms_sleep(1000);
                SPEAKER_F = 0;

                SODA = 3;
                return 0;
        }

        return 1;
}

//=============================================================================
==========================
// AVRCAM
//=============================================================================
==========================

void UART1_init(void)
{
        UCSR1A |= 0x02; // Reduces divisor of baud rate to 8- doubles transfer rate, U2X = 1
        UCSR1B = 0x18; // Enables the Reciever and Transmitter
        UCSR1C = 0x06; // Asychronous operation; No parity; 1 stop bit; 8 bits
        UBRR1H = 0x00;
        UBRR1L = 0x10; // UBRR = 16, 115.2 kbps baud rate
}

void UART1_transmit(char data[])
// Transmits a char array over UART1
{
```

```
	int k;

	for(k=0;k<5;k++)
	{

		while( !(UCSR1A & (1<<UDRE1)) ) // Wait for a empty transmit buffer
			;

		UDR1 = data[k];
	}

}

unsigned char UART1_receive(void)
// Receives data overy UART1 and returns it
{
	while( !(UCSR1A & (1<<RXC1)) ) // Wait for data to be recieved
		;

	return UDR1;
}

void AVRCAM_track_on(void)
// Turns the AVRCam on
{
	UART1_transmit("ET\r"); // Enable Color Tracking mode
}

void AVRCAM_track_off(void)
// Turns the AVRCam off
{
	UART1_transmit("DT\r"); // Disable Color Tracking mode
}

void AVRCAM_data(void)
// Displays the color found- byte 2- 00-07.
// Make averaging better....
{
	unsigned char AVRCAMdata1[13];
	int k;

	unsigned char temp[50];
	int i;

	lcd_clear_display();

	for(i=0;i<50;i++) // Reads in everything, including ACK/r
	{
		temp[i] = UART1_receive();

	}

	k = 0;

	while(temp[k] != 0x0A) // Increments till the beginning of the packet stream
	{
```

```
                k++;
        }

        if(temp[k+1] != 0xFF)     // If dont have temp[]= 0A,FF,0A,...
        {
                for(i=0;i<13;i++)
                {
                        AVRCAMdata1[i] = temp[k+i];
                }


                if(AVRCAMdata1[2] <= 3)                 // If a valid color
                {
                        CAM_counter++;

                        if(AVRCAMdata1[2] == 0x00)
                        // If color 1 displays it to LCD
                        {
                                AVRCAM_AVG = AVRCAM_AVG + 0;
                        }
                        else if(AVRCAMdata1[2] == 0x01)
                        // If color 2 displays it to LCD
                        {
                                AVRCAM_AVG = AVRCAM_AVG + 1;
                        }
                        else if(AVRCAMdata1[2] == 0x02)
                        // If color 3 displays it to LCD
                        {
                                AVRCAM_AVG = AVRCAM_AVG + 2;
                        }
                }

        } // Ends if(temp[k+1] != 0xFF)

}

uint8_t AVRCAM_data_avg(void)
// Returns an average AVRCAM color reading. Prints it to LCD and returns 1 when done.
{

        AVRCAM_AVG = 0;
        CAM_counter = 0;

        while(CAM_counter < 20)         // Takes 20 samples
        {
                AVRCAM_data();
        }

        lcd_clear_display();

        printf("Color Avg = %i",AVRCAM_AVG);
        lcd_goto_line(2);

        ms_sleep(2000);

        lcd_clear_display();
```

```
	if(AVRCAM_AVG <= 6)
	{
		printf("Color1: Coke");

		if(SODA == 1) // If Coke was selected by remote
		{
			return 1; // In pick up soda mode
		}

		else
		{
			lcd_goto_line(2);
			printf("Not this one");
			return 0;
		}
	}
	else if(AVRCAM_AVG >= 7 && AVRCAM_AVG <= 31)
	{
		printf("Color2: Pepsi");

		if(SODA == 2) // If Pepsi was selected by remote
		{
			return 1; // In pick up soda mode
		}

		else
		{
			lcd_goto_line(2);
			printf("Not this one");
			return 0;
		}
	}
	else if(AVRCAM_AVG >= 32 && AVRCAM_AVG <= 40)
	{
		printf("Color3: Mt. Dew");

		if(SODA == 3) // If Sprite was selected by remote
		{
			return 1; // In pick up soda mode
		}

		else
		{
			lcd_goto_line(2);
			printf("Not this one");
			return 0;
		}
	}

	printf("No Color Match");
	return 0; // If not in any of the ranges
}

//=====================================================================
==================
```

```
// BEHAVIORS
//=============================================================================
==================

void avoid_ir(void)
// Obstacle Avoidance
{
	uint16_t ir_l;
	uint16_t ir_r;

	if(IR_flag == 1) // If flag was set already
	{
		ms_sleep(3000); // waits 3 seconds
		lcd_clear_display();
	}


	ir_l = adc_readn(0,20); // pin0, 5 samples, takes the average

	ir_r = adc_readn(6,20); // pin6. pin7 does not work.

	if(ir_l > 400 || ir_r > 400) // 450 works, may be too high
	{
		if(IR_flag == 0) // If the message was not already displayed
		{
			lcd_goto_line(2);
			printf("IR: Remove object");
		}

		IR_output_d = 2; // Backward
		IR_output_s = 0;
		IR_flag = 1;

		SPEAKER_F = 1136; // freq= 1760 Hz. 1/(16 MHz/8)*1136 = 1/1760
		ms_sleep(500);
		SPEAKER_F = 0;
		ms_sleep(100);

	}

	else
	{
		IR_flag = 0;
	}
}

void avoid_bump(void)
// Bump ring obstacle advoidance
{
	if(BUMP_flag == 1) // If the bump was hit last time in this fnc
	{
		ms_sleep(3000); // waits 3 seconds

		lcd_clear_display();
	}
```

```
	if((BUMP_PIN & 0x01) == 0 || (BUMP_PIN & 0x02) == 0 || (BUMP_PIN & 0x04) == 0)
	{
		BUMP_flag = 1;
		BUMP_output_d = 2; // Backward
		BUMP_output_s = 0;

		lcd_goto_line(2);
		printf("B: Remove object");

	}

	else
	{
		BUMP_flag = 0;
	}

}

unsigned char line_tracking(void)
// 2200 slow, 2400 medium speed
{
		switch(PHOTO_PIN)
		{
			case 0x0F:		// 1111
			{
				//Do nothing. Turning messes it up. May try stop/forward/back.

				break;
			}
			case 0x0E:		// 1110
			{

				PHOTO_output_d = 4;			// Right
				PHOTO_output_s = 2400;			//	2500 may be
too high
				PHOTO_flag = 1;

				break;
			}
			case 0x0D:		// 1101
			{
				PHOTO_output_d = 4;		// Right
				PHOTO_output_s = 2200;		// good for now
				PHOTO_flag = 1;

				break;
			}
			case 0x0C:		// 1100
			{

				PHOTO_output_d = 4;		// Right
				PHOTO_output_s = 2400;		// Good for now
				PHOTO_flag = 1;

				break;
			}
```

```
case 0x0B:              // 1011
{
        PHOTO_output_d = 3;         // Left
        PHOTO_output_s = 2200;      // 2200 is good
        PHOTO_flag = 1;

        break;
}
case 0x0A:              // 1010
{
        // Do nothing.

        break;
}
case 0x09:              // 1001
{
        PHOTO_output_d = 1;         // Forward
        PHOTO_output_s = 2400;      // 2500 may be too fast
        PHOTO_flag = 1;

        break;
}
case 0x08:              // 1000
{
        PHOTO_output_d = 4;         // Right
        PHOTO_output_s = 2400;      //
        PHOTO_flag = 1;

        break;
}
case 0x07:              // 0111
{

        PHOTO_output_d = 3;         // Left
        PHOTO_output_s = 2400;      // 2500 may be too high
        PHOTO_flag = 1;

        break;
}
case 0x06:              // 0110
{
        // Do nothing

        break;
}
case 0x05:              // 0101
{

        // Do nothing.

        break;
}
case 0x04:              // 0100
{
        // Do nothing.
```

```
                        break;
                }
                case 0x03:              // 0011
                {

                        PHOTO_output_d = 3;             // Left
                        PHOTO_output_s = 2400;          //
                        PHOTO_flag = 1;

                        break;
                }
                case 0x02:              // 0010
                {
                        // Do nothing.

                        break;
                }
                case 0x01:              // 0001
                {
                        PHOTO_output_d = 3;             // Left
                        PHOTO_output_s = 2400;          //
                        PHOTO_flag = 1;

                        break;
                }
                case 0x00:              // 0000
                {
                        // At intersection

                        PHOTO_flag = 1;

                        break;
                }
        }

        return PHOTO_PIN;
}

void turn_left_line()
// Turns left until over a line
{
        motor_move(3,2200);             // Turns left, 2200 too slow

        while(PHOTO_PIN&0x02); // Breaks once right inner sensor over line

        MOTOR = STOP;

}

void turn_right_line()
// Turns right until over a line
{
        motor_move(4,2200);             // Turns right, 2200 too slow

        while(PHOTO_PIN&0x04); // Breaks once left inner sensor over line
```

```
	MOTOR = STOP;

}

void forward_line()
// Move forward untill on a line
{
	motor_move(1,2200);

	while(PHOTO_PIN&0x06);		// Breaks once both inner sensors over a line

	MOTOR = STOP;
}

void turn_left_off_line()
// Turns left until off a line
{
	motor_move(3,2200);		// Turns left, 2200 too slow

	while(1) // Breaks once none of the sensors are over the line
	{
		if((PHOTO_PIN&0x0F) == 0x0F)
		{
			break;
		}
	}

	MOTOR = STOP;

}

void turn_right_off_line()
// Turns right until off a line
{
	motor_move(4,2200);		// Turns right, 2200 too slow. 2500 too fast

	while(1) // Breaks once none of the sensors are over the line
	{
		if((PHOTO_PIN&0x0F) == 0x0F)
		{
			break;
		}
	}

	MOTOR = STOP;

}

void pickupsoda_front(void)
// Moves robot to pick up the front soda can and returns robot to return line
{
	lcd_clear_display();
	printf("OFF1");

	motor_move(1,2500);			// Move over first intersection
	ms_sleep(750);			// 500 too short
```

```
	MOTOR = STOP;

	while(line_tracking() != 0) // While not at a intersection, track to get to soda can
	{
		printf("LT");
		line_tracking();

		motor_move(PHOTO_output_d,PHOTO_output_s);
	}

	MOTOR = STOP;		// Stop at intersection

	ms_sleep(1000);			// Close claw
	SERVO = S_CLOSE;
	ms_sleep(1000);

	lcd_clear_display();
	printf("OFF2");

	//turn_right_off_line();	// Return robot to return line
	motor_move(4,2200);		// Right
	ms_sleep(1500);
	turn_right_line();
}

void pickupsoda_right(void)
// Moves robot to pick up the right soda can and returns robot to return line
{
	while(line_tracking() != 0) // While not at a intersection, track to get to soda can
	{
		printf("LT");
		line_tracking();

		motor_move(PHOTO_output_d,PHOTO_output_s);
	}

	MOTOR = STOP;		// Stop at intersection

	ms_sleep(1000);			// Close claw
	SERVO = S_CLOSE;
	ms_sleep(1000);

	lcd_clear_display();
	printf("OFF2");

	ms_sleep(1000);

	motor_move(2,2400); // Backward
	ms_sleep(1200);			// Used to be 2000

	motor_move(4,2200);		 // Turns Right
	ms_sleep(500);
	turn_right_off_line();	// Return robot to return line
	turn_right_line();
}
```

```
void pickupsoda_left(void)
// Moves robot to pick up the left soda can and returns robot to return line
{
	while(line_tracking() != 0) // While not at a intersection, track to get to soda can
	{
		printf("LT");
		line_tracking();

		motor_move(PHOTO_output_d,PHOTO_output_s);
	}

	MOTOR = STOP;		// Stop at intersection

	ms_sleep(1000);			// Close claw
	SERVO = S_CLOSE;
	ms_sleep(1000);

	lcd_clear_display();
	printf("OFF2");

	ms_sleep(1000);

	motor_move(2,2400); // Backward
	ms_sleep(1000);			// Used to be 2000

	motor_move(3,2200);		// Turns left
	ms_sleep(500);
	turn_left_off_line();	// Return robot to return line
	turn_left_line();
}

void arbitrate()
// Behavior arbitrator
{
	if(PHOTO_flag == 1)
	{
		motor_input_d = PHOTO_output_d;
		motor_input_s = PHOTO_output_s;

	}

	if(IR_flag == 1)
	{

		motor_input_d = IR_output_d;
		motor_input_s = IR_output_s;

	}

	if(BUMP_flag == 1) // Highest priority
	{

		motor_input_d = BUMP_output_d;
		motor_input_s = BUMP_output_s;

	}
```

```
}

//=============================================================================
==========================
// MAIN
//=============================================================================
==========================

int main(void)
{
	int k;
	LCD_DDR= 0xFF; // Set all bits of port C for output
	IR_DDR= 0x00; // Set all bits of port F for input
	PHOTO_DDR = 0x00; // Set all bits of port A for input
	MOTOR_DDR = 0x0F; // Set all bits of port B for output
	SPEAKER_DDR = 0XF0;
	IRCAN_DDR = 0x00; // Set for input- IR can
	DDRE = 0x38; //00111000- Sets OCA,B,C to outputs
	BUMP_DDR= 0x00; // Set bump switches to inputs

	BUMP = 0xFF; // Enables internal pull up resistors of port

	IR_flag = 0;
	BUMP_flag = 0;
	PHOTO_flag = 0;
	MODE = 0;			// To/From soda pickup
	FIRST_PASS = 0;
	SODA = 0;
	PATH = 0;

	init_timer();

	sei();  // enable interrupts

	lcd_init();
	adc_init();		// For IR rangefinders
	UART1_init();	// Initializes UART1
	motor_init();
	speaker_init();

	fdevopen(lcd_char,NULL,0);	// Sets up to print to lcd using printf


	while(1)
	{

		if(FIRST_PASS == 0 && MODE == 0)
		{
			banner(); // displays banner

			while(remote_avg()); // Loops till a key is pressed

			ms_sleep(1000);
		}
```

```
if(MODE == 0) // If going to/from soda pickup
{
	lcd_clear_display();
	printf("MODE 0");

	while(line_tracking() != 0) // Break out once hit a intersection
	{
		avoid_bump();
		avoid_ir();
		line_tracking();
		arbitrate();

		motor_move(motor_input_d,motor_input_s);
	}

	MOTOR = STOP;		// Stop b/c at intersection

	if(FIRST_PASS == 0)
	{
		FIRST_PASS = 1;
		MODE = 1;	// Go to soda detection mode
	}

	else	// Coming back with soda can
	{
		lcd_clear_display();	// Sequence is done
		printf("Done!");

		SERVO = S_MIDDLE;

		for(k=0;k<5;k++)
		{
			SPEAKER_F = 2025; // freq= 987.7 Hz. 1/(16
MHz/8)*2025 = 1/987.7
			ms_sleep(750);
			SPEAKER_F = 0;
			ms_sleep(250);
		}

		// Turn around robot before restarting:
		if(PATH == 1) // Left Path
		{
			turn_right_off_line();
			forward_line();
		}
		if(PATH == 2) // Center Path
		{
			//turn_right_off_line();
			motor_move(4,2200); // right
			ms_sleep(2500);
			turn_right_line();
		}
		if(PATH == 3) // Right Path
		{
			turn_left_off_line();
			forward_line();
```

```
                        }

                        FIRST_PASS = 0;         // Restart cycle
                }

        }       // Ends if MODE == 0

        if(MODE == 1) // If in soda pick up mode
        {
                lcd_clear_display();
                printf("MODE 1");
                ms_sleep(5000);

                AVRCAM_track_on();

                if(AVRCAM_data_avg() == 1) // If the soda that is being looked at is the
one desired
                {
                        ms_sleep(2000);

                        AVRCAM_track_off();

                        pickupsoda_front();

                        PATH = 2; // Center path

                        MODE = 0; // Back to to/from soda pick up mode
                }

                else
                {
                        turn_right_off_line();      // Move to check right soda can
                        turn_right_line();

                        ms_sleep(1000);

                        if(AVRCAM_data_avg() == 1) // If the soda that is being looked
at is the one desired
                        {
                                ms_sleep(2000);

                                AVRCAM_track_off();

                                pickupsoda_right();

                                PATH = 1; // Left path

                                MODE = 0;       // Back to to/from soda pick up mode
                        }

                        else
                        {

                                turn_left_off_line();       // Move to check left soda can
                                turn_left_line();
                                turn_left_off_line();
```

```
                                        turn_left_line();

                                        ms_sleep(1000);

                                        if(AVRCAM_data_avg() == 1) // If the soda that is being
looked at is the one desired
                                        {
                                                ms_sleep(2000);

                                                AVRCAM_track_off();

                                                pickupsoda_left();

                                                PATH = 3; // Right path

                                                MODE = 0;       // Back to to/from soda pick up
mode
                                        }

                                        else
                                        {
                                                ms_sleep(3000);

                                                lcd_clear_display();
                                                printf("None detected"); // Displays if none of the
3 cans is the one desired

                                                AVRCAM_track_off();

                                                turn_left_off_line();       // Move return home w/
no soda can. :(
                                                turn_left_line();

                                                PATH = 3; // Right path

                                                MODE = 0;

                                        }       // Ends else3
                                } // Ends else2
                        } // Ends else1

                } // End if Mode = 1

        } // Ends main while loop


} // Ends main()
```