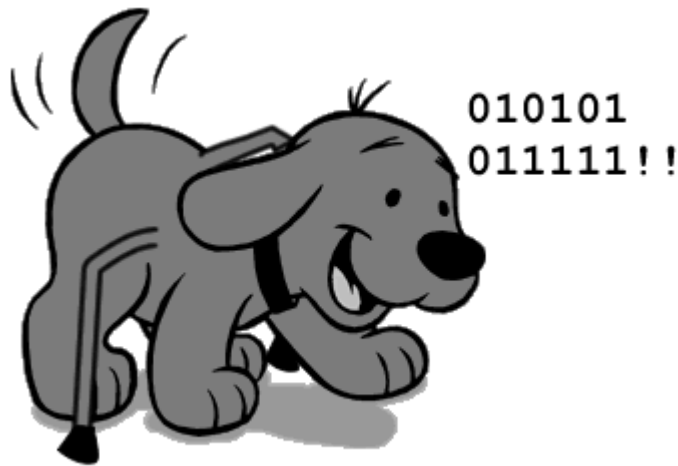# University of Florida
# Department of Electrical and Computer Engineering
# EEL 5666 - Summer 2005
# Intelligent Machines Design Laboratory

Instructor: A. Arroyo
TAs: William Dubel, Steven Pickles

# T.E.R.E.S.A.

**(Tyrannous Effeminate Rambling Esoteric Sybaritic Arthopod)**

## Six-Legged Robopuppy

## FINAL REPORT

Blake C. Sutton

**Table of Contents**

# Abstract

Teresa the Six-Legged Robopuppy is a hexapod entertainment robot that mimics a pet. She is equipped with a pyroelectric sensor to seek out humans, ultrasound ranging for proximity detection, contact switches or photoreflectors for edge-of-the-world detection, and contact switches to track the pull of a leash. She can walk forward and backward and turn using three servo motors attached to her legs by control wires.

Teresa's behavior includes avoiding obstacles, detecting and demanding attention from a playmate (a human or animal), walking on a leash, and dancing. This report includes a detailed description of all of Teresa's sensors, actuation, design, and software algorithms.

# Executive Summary

Teresa's behavior includes avoiding obstacles, detecting and demanding attention from a playmate (a human or animal), walking on a leash, and dancing. An integrated system of four sensors, a microcontroller board, and a six-legged mobile platform bring Teresa to life. Sensors and motion control are arbitrated by the central microcontroller—Teresa's brain. The six-legged platform allows Teresa to walk forward and backward and turn via servo-connected control wires on her legs. Teresa's brain, body, muscles (actuators), senses (sensors), and behavior will be discussed in detail in following sections.

Teresa's integrated system consists of a microcontroller board, a six-legged physical platform, three servo motors, a pyroelectric sensor, a hacked sega genesis controller, and two sonar sensors. The servo motors used to realize Teresa's motion are controlled from her brain (the microcontroller board). Teresa's brain also periodically polls all sensors to inform her decisions. The microcontroller board selected for Teresa is the Mavric-IIB. Three servo motors move Teresa's six legs in a modified tripod gait determined by software.

Teresa's body is based on a pre-designed six-legged platform. Teresa's actuators (muscles) consist of servo motors. Motion is achieved by attached control wires between servo cranks and

Teresa's legs. To turn, Teresa will walk forward with one side while reverse-walking on the other. To achieve the dancing effect, Teresa wobbles from side to side by shifting her center legs.

Teresa's ultrasound sensor(s) give her vision in the sense of proximity detection. One ultrasound sensor is mounted on the front of Teresa's body, enabling her to steer clear of obstacles. The other is mounted on the bottom, enabling her to detect the edge of her world (cliffs). Teresa's pyro sensor gives her the ability to see humans as heat sources. One is mounted on the front of her body near her ultrasound sensor. A sega genesis controller provides a joystick for control via a leash as well as buttons to press to scratch her in her "special place" and start and pause motion in the robot at any time.

## Introduction

Teresa the Six-Legged Robopuppy is an entertainment robot that imitates the playfulness and short attention span of a puppy. Meant to be a pet substitute for people who cannot have a living puppy due to constraints on time or space, Teresa will notice humans and dance in happiness and demand attention when she finds one. She will also have the ability to be walked on a leash. Like any animal Teresa will avoid obstacles in her path, including walking off the edge of a table. This combination of characteristics, in addition to her novel six-legged looks, makes Teresa the ideal apartment pet! Cute, fun, and interactive, Teresa is sure to be a hit with anyone she meets.

An integrated system of four sensors, a microcontroller board, and a six-legged mobile platform will bring Teresa to life. Sensors and motion control will be arbitrated by the central microcontroller—Teresa's brain. All programs will be written in the C language. The six-legged platform will allow Teresa to walk forward and backward and turn via servo-connected control wires on her legs. Teresa's body will be based on Cricket the Robot, a fully planned and tested

platform described in [3]. Teresa's brain, body, muscles (actuators), senses (sensors), and behavior will be discussed in detail in following sections.

In planning this project, two books and many people were helpful. [1] is a good introduction to the concepts of motors, servos, and weight as applied to robotics. [2] provided a thoughtful treatment of common problems and solutions in robot behavior. Background information from William Dubel, Michael Ihms (designer of ELSI), countless IMDL final reports, and several websites made up the backbone of the research for this project.

However, when it came down to the practical aspects of this project (including make the thing actually walk), the help of Marco Downs (UF architecture student) was indispensable. He constantly supplemented my meager construction experience with new ideas and encouragement and lent me tools I would never have thought to buy.

## Integrated System

Teresa's integrated system consists of a microcontroller board, a six-legged physical platform, three servo motors, a pyroelectric sensor, a hacked sega genesis controller, and two sonar sensors. These represent, respectively, Teresa's brain, body, muscles, heat sensing, touch, and vision. The servo motors used to realize Teresa's motion are controlled from her brain (the microcontroller board). Teresa's brain also periodically polls all sensors to inform her decisions.

Like human or animal brains, the central microcontroller board serves as the hub where all activity and ability is generated. The microcontroller board selected for Teresa is the Mavric-IIB. Based on the Atmega128 chip, the Mavric-IIB has 128K FLASH memory, 4K Static RAM, and 4K EEPROM. With its small size (2.2 x 3.6 inches) and ease of use, the Mavric-IIB seems ideal for Teresa's brain.

The programs which determine Teresa's everyday behavior are written as C modules. All programs were compiled with WinAVR and downloaded to the board with PonyProg 2000. An ISP cable was used for the physical interface.

Three servo motors move Teresa's six legs in a modified tripod gait determined by software. In the same way, feedback from Teresa's sensors motivate behavior changes such as avoiding an obstacle, avoiding an edge, or dancing in front of a person.

## Mobile Platform

Teresa's body is based on the six-legged platform designed in [3]. However, certain modifications were necessary based on the available materials and tools.

Using this platform, Teresa can walk forward and backward and turn in any direction. With the modified tripod gait in software, at least three legs are always on the ground to maintain stability, but the maximum pace is faster than the maximum speed when using other gaits. However, Teresa is not capable currently of traversing rough, uneven surfaces, due to the way the walking method works. Instead, she is limited to small irregularities on any relatively flat, smooth surface. After all, wood flooring, glass, and linoleum, are Teresa's native environment.

The body construction consists of six legs, control wires, servo motor brackets, and a chassis consisting of a carapace and two side pieces. The chassis is made of 1/8" balsa wood cut in the laboratory. Specific information and diagrams for the construction of the rest of the body can be found in [3].

Leg construction and attachment to the main body proved difficult for me, especially since I am a beginner at building. As I was unable to find aluminum tubing with sufficient wall thickness to bend without kinking and breaking, I chose to nest consecutive sizes of thin-walled tubing to

achieve the desired strength.  Next, the issue of how to attach the tubing to the hinged metal standoffs used for Teresa's swinging joints was particularly vexing.  Using superglue, as suggested in the Cricket design, would not hold up to the amount of weight the joints must bear.  Finally, I decided to use straight ¼" brass pipe fittings to attach the tubing to the standoffs, although the extra weight from the six brass fittings is not ideal.   The next issue was the standoffs themselves, which consist of thin, flimsy pins that constantly come loose. After constantly tightening and replacing Teresa's joints, I enlarged the pin hole and inserted size 2 screws with nuts to hold the standoffs together.  This new method has worked much better than the original pins inserted into the standoffs, and has the additional benefit that the standoffs can be taken apart for tightening and assembly.

If I were rebuilding the platform one last time, I would pay much more attention to weight. I would eliminate the pipe fittings and any other extra weight, possibly by shortening the legs.  I would also eliminate the brass standoffs, which were a constant problem to orient, tighten, and attach to the legs.  Instead, I would create my own, better joints by directly sandwiching aluminum tubing between brackets, held together by a screw.  This would provide much stiffer, better, more accessible swiveling joints which are also quite light, since brackets could be made from aluminum as well.  This would require a slight relocation of the servo mounts (to allow extra space), and also a slight widening of the body's side pieces.

In addition, I would also make the chassis wider to accommodate more electronics.  I had to be relatively creative to fit everything I wanted on the platform (sacrificing a dvd case to make a second story on which to mount more components).

Finally, the platform does not yet allow completely foolproof edge detection, as the sonar can miss an edge that a leg can fall off.  The weight of the robot is such that no leg can go off an

edge without the whole robot being compromised. This could be avoiding by rebalancing the robot and installing edge detection on the bottom of the feet, or installing additional sonar units on either side of the robot, closer to the actual spread of the front legs.

## Actuation

Teresa's actuators (muscles) consist of servo motors. Three standard servo motors control her six legs: one for her two center legs, one for the back and front legs on her left side, and one for the back and front legs on her right side. This three-servo scheme is one of the most efficient methods of achieving fast, flat-surface walking in a six-legged robot. In this scheme, the center legs move up and down while the front and back legs on each side move forward and backward. This motion is achieved by attached control wires between servo cranks and Teresa's legs.

The algorithm (implemented in software) which allows Teresa to walk is a modified tripod gait. In the tripod gait, three legs are always on the ground to ensure stability. In this modified tripod gait, the front and back legs on each side are tied together and move at the same time. To walk, Teresa will repeatedly go through the following steps (alternating sets of legs in the air):

1. Lift center legs, tilting body to one side.

2. Move right set of legs forward.

3. Move left set of legs backward.

4. Set down center legs.

To turn, Teresa will walk forward with one side while reverse-walking on the other. The number of steps to achieve various turns and distances will be determined from observation and

manual calibration. To achieve the dancing effect, Teresa wobbles from side to side by shifting her center legs.

Beyond the high-level walking algorithm, the servos are physically controlled by pulse-width modulation (PWM) from the microcontroller. During a 20 ms period, the control signal is set high for a specific time to indicate to which position the servo should move. In C code, I set up the Atmega128's PWM unit to generate PWM on various output pins with specific settings. To change a servo's value, I simply modify the output compare value in the register associated with the output pin attached to the particular servo.

All three of the servos used are relatively high-torque, about double the power of standard servos. The higher torque was needed due to the high weight of the finished robot, since the whole walking process rests on the ability to rock the robot from side to side and lift one set of legs off the ground.

I wish I had bought the stronger servos earlier in the semester, since they are only slightly heavier and much more powerful. When you can't cut down on weight very much, a stronger servo can save you from a lot of problems you might not even realize you have unless you try to get by on less power. On the other hand, the stronger servos probably also burn up batteries more quickly. I have learned that low battery power causes extremely slow and unpredictable behavior from servos—after a while I realized that the servos were not broken but that the batteries simply needed changing.

## Sensors

Teresa's senses (sensors) provide her brain with the feedback on the outside world required to make decisions. She is equipped with two ultrasound rangers, a pyroelectric sensor, and a sega

controller.  These sensors enable her to see obstacles in her path, find humans (as heat sources), know when there is ground in front of her, know when someone is scratching her ear, and know which direction her leash is pulled when she is walked.

*Ultrasound / Sonar*

Teresa's ultrasound sensor(s) give her vision in the sense of proximity detection.  By periodically emitting an ultrasound pulse and measuring the time it takes to receive the return pulse, the distance to the reflecting object (in the direction of the pulse) can be determined.  Unlike infra-red sensors intended for the same purpose, an ultrasound sensor continues to function outdoors (with high ambient IR).  An ultrasound sensor's operation is also independent of the surface the pulse reflects from, unlike IR.  For this reason, ultrasound sensors are a better choice.

One ultrasound sensor is mounted on the front of Teresa's body, enabling her to steer clear of obstacles.  Another is mounted on the bottom of her body to enable her to see when the ground has dropped away and she should back up to avoid falling off.

For both purposes, the Devantech SRF04 was used.  Both were ordered from Acroname (http://www.acroname.com).  This sensor has five pins--+5 volts, ground, pulse_trigger, and echo_out.  Pulse_trigger is set to high when an SRF04 pulse is desired, and echo_out holds the signal which remains high for a period indicating the sonar reading.

To get a sonar reading in software, a timed interrupt is used to poll the SRF04's echo output pin and check if the pulse is still high.  An interrupt fires every 67 us, so that is the resolution of each ultrasound ranger.  Every time the interrupt fires, the interrupt service routine checks if the echo output pin is still high—if it is, it increments the current ultrasound reading count by one, otherwise it saves the current count to a globally accessible sensor reading variable for the main

loop to access.  Pulses are fired continuously—as soon as one pulse is received back, the signal is sent out for another pulse to be activated.

Please see my Sensor Report for this project to find data and a graph for the data from the sonar unit.

*Pyroelectric*

The purpose of Teresa's pyroelectric sensor is to give her the capability of finding humans or animals to interact with.  The sensor is sensitive to infra-red and motion, and gives feedback of different kinds when these things are detected.  The sensor chosen was the Eltec R3-PYRO1, purchased from Acroname (http://www.acroname.com).  The kit came with a lens and cardboard cone housing, meant to screen all kind of radiation except the most commonly emitted by human presence.

I used the pyro sensor in conjunction with the ultrasound ranger to detect when a human is in a certain range in front of the robot, so that she would know when to beg for attention.  The pyro sensor gives reading within a certain range of values when it detects motion in either direction.  In software, I used the analog to digital converter to read the pyro sensor's value constantly and compare it to these values.  See the Pyro.c source code in the appendix for details.

Please see my Sensor Report for data and a graph showing the pyro sensor's reaction to various stimuli.

*Sega Genesis Controller / Joystick*

The Sega Genesis Controller consists of simple touch switches which ground when pressed. The unit runs off of 5 volts on the microcontroller board, which powers the switches feeding into the input port on the board.  To make a joystick (for the leash to pull) from the standard direction pad, I epoxied a plastic furniture tack into the center of the pad.  I then screwed a wooden dowel into the tack and tied a ribbon through the dowel to make a crude joystick.

To discover the connections internal to the Sega, I took it apart and found which connections went to which pins on the standard 9-pin Dsub connector, then cross-referenced that with a handy pinout I found online at (http://pinouts.ru).  The Sega works by simply grounding switches when they are pressed, with the additional complication that an internal 2 to 1 multiplexer is used to switch between the A and B buttons and the Start and C buttons.  I simply set the Select line to a constant value and used the A and Start buttons only.

After dissecting the input and output wires to the controller, using it was simply a matter of connecting the wires to a port on the microcontroller board and polling that port at strategic points in the code.  See the Appendix for actual source code.

## Behaviors

It is Teresa's behavior, in conjunction with her moods, that brings her to life.  To seem like a pet she must act like a pet, and her behaviors serve this purpose.   While not as complex as a real animal, Teresa's behavior allow her to mimic some of the most salient characteristics of a rambunctious puppy.  No, not the obvious one—Teresa does not leave mysterious yellow puddles on the floor!

*Obstacle Avoidance / Edge of the World Detection*

Although Teresa is meant to mimic a rambunctious puppy, she should not crash into things and damage herself.  As a result, Teresa detects obstacles and avoids them.  When faced with an obstacle (an object within a certain distance of her front), Teresa will back up a short distance and turn in a randomly chosen direction.  In this way, she will avoid hitting tables, clutter on the floor, people, and angry hamsters.

In addition, Teresa detects and avoids falling off the edge of her world.  Since, like humans, she is not equipped to walk on thin air, she will constantly check that there is a solid surface in front of her as she walks forward.  When she realizes a foot has gone off the edge she will back away and turn away.

*Dance For Humans*

When Teresa sees a person in front of her (using a combination of her pyro sensor and her ultrasonic ranger), she will dance for attention until she is scratched (her "A" button is pressed).

*Walking on a Leash*

Like any reasonably trained pet, Teresa is trained to walk on a leash.  Like most dogs, this training is somewhat sketchy—she only responds to two directions of pulling on the leash (from side to side), and her response is sometimes delayed due to her interest in her surroundings.

*Likes being scratched*

When users press the "A" button on her sega controller, Teresa reacts by stretching her legs and printing a happy message on her LCD screen.  Like puppies, she has a special spot she likes people to touch!

## Experimental Layout and Results

I tested Teresa informally in a variety of situations. One thing I noticed immediately is that Teresa performs differently on different surfaces—while she walks forward on a stickier surface like linoleum, she walks slower and does not bounce. On glass, on the other hand, her walking is bouncy and relatively crisp.

I tested her obstacle avoidance and response time by holding a hand in front of her to trigger her avoidance routine. I tested her edge detection by allowing her to walk into an edge (but planning to catch her if she fell). I found the response on both sensors to be quite fast.

The pyro / proximity detection used to detect humans in a certain range was tested separately, by waving a hand in range in front of the detector while the robot stood stationary. It proved to be relatively reliable—I can't tell how reliable it really is due to the delays needed to display the LCD while I work. However, from the testing of the final integrated version of the robot, the response is quick enough for the purpose of the behavior.

## Conclusion

By far the most difficult part of this project was the mechanical aspect—building the platform and making Teresa walk. I spent the entire semester working on different ways to build Teresa's legs, make rigid but flexible swiveling joints without a metal shop, cut down on her weight, and mount components. Once a reasonably walking platform was accomplished (and it is still much heavier and more awkward than I desire), the software was simple for me. The truth is that the vast majority of the work I did this semester was ultimately worthless, when I threw out one design in favor of another that worked slightly better. If I had somehow been able to find the

right ideas to focus on for the walking platform in the first few weeks, Teresa would have much more sophisticated behaviors and she would certainly be making sounds.

At the moment Teresa avoids obstacles and edges to the best of her ability based on how her sonar units are mounted. She will fall off edges if she approaches them from the wrong angle, so that the bottom-mounted SRF04 does not go off the edge before the legs do. However, if she approaches an edge relatively straight, she will find the edge and back away before falling off. Again, to me, my biggest accomplishment this semester is making a platform that actually walks forward.

If I could start the project over again, I would throw out my platform and make an improved but similar design that is much lower in weight. I would widen the carapace and make custom joints, and probably also find light, rechargeable lithium batteries to cut down on weight. Now that I have experience putting things together and experimenting with what works for a walking platform, I could put the new platform together relatively quickly and then concentrate on improving the software. In addition, I would work on a better way to detect edges, either using different sensors or using additional, better-placed SRF04 units. I would also add the sound unit I didn't have time to hack and incorporate, and I would add many random factors to make the puppy more lifelike. I left out many random factors in Teresa's current incarnation in order to be able to demonstrate behaviors directly.

## Documentation

[1] D. Clark, M. Owings, Building Robot Drive Trains, New York: McGraw-Hill, 2003.
[2] J. L. Jones, Robot Programming, New York: McGraw-Hill, 2004.
[3] H. Arnold, "Cricket the Robot" [Online document], Available HTTP:
http://home.earthlink.net/~henryarnold/

# Appendix A: Source Code

Ultrasound.h:

```c
/*
 * Title: Ultrasound
 *
 * Author: Blake C. Sutton
 *
 * This is a set of functions used to test and read input from
 * a SRF04 ultrasound ranger connected to a port on the Mavric IIB.
 * It will have functions to tell the SRF04 to send a pulse out,
 * read the data back (by counting the cycles the SRF04 output pin
 * is high ), and be able to display this value on the LCD
 *
 * Pulse Trigger input (to SRF04) : Port D, pin 0
 * Echo output (to Mavric II) : Port D, pin 1
 */

#ifndef __Ultrasound_h__
#define __Ultrasound_h__

#include "LCD_Routines.h"

//Input and output pins used by the front-mounted sonar
//(for detecting obstacles in front of Teresa)
#define PULSE_TRIGGER_FRONT (0x01)
#define ECHO_OUTPUT_FRONT   (0x02)

//Input and output pins used by the bottom-mounted sonar
//(for detecting unstable / nonexistent footing )
#define PULSE_TRIGGER_BOTTOM (0x40)
#define ECHO_OUTPUT_BOTTOM   (0x80)

//Length of array of sonar readings (for both sensors)
#define SONAR_ARRAY_LENGTH    1

//Constants defining thresholds of closeness to an object
//for front-mounted sonar.  Bottom-mounted sonar should be
//self-calibrating in final version
#define VERY_NEAR             0x0003
#define NEAR                  0x0008

#define MAX_HEIGHT            0x0006
```

```c
/* 16-bit unsigned integers used for ultrasound timing */
volatile uint16_t front_ultrasound_count;
volatile uint16_t bottom_ultrasound_count;

volatile uint16_t front_ultrasound;
volatile uint16_t bottom_ultrasound;

/* 16-bit unsigned integer incremented every time the 67 us interrupt
   is fired.  Useful for small delays, not used it anything but spin
   loops
 */
volatile uint16_t timed_interrupt_count;

/* Array holding the last ARRAY_LENGTH readings from the front SRF04 */
volatile uint16_t front_sonar_readings [SONAR_ARRAY_LENGTH];
volatile uint8_t front_sonar_readings_ct;

/* Array holding the last ARRAY_LENGTH readings from the bottom SRF04 */
volatile uint16_t bottom_sonar_readings [SONAR_ARRAY_LENGTH];
volatile uint8_t bottom_sonar_readings_ct;

//Internal use only
void init_ultrasound_timer(void);

//Use this with other port initializing functions
void init_ultrasound_port(void);

//Internal use only
void send_pulse_front(void);

//Internal use only
void send_pulse_bottom(void);

//Call this in the main loop to keep updating the sonar with
//pings and readings back.  Results are added to the sonar_readings
//arrays above.
void update_sonar(void);

void show_sonar(void);

#endif
```

Ultrasound.c

```c
/*
 * $Id: LCD_Routines.c,v 1.3 2005/03/22 18:51:42 bsd Exp $
```

```
 *
 */

/*
 * Ultrasound
 *
 * This is a set of functions used to test and read input from
 * a SRF04 ultrasound ranger connected to a port on the Mavric IIB.
 * It will have functions to tell the SRF04 to send a pulse out,
 * read the data back (by counting the cycles the SRF04 output pin
 * is high ), and be able to display this value on the LCD
 *
 * FRONT MOUNTED SRF04:
 * Pulse Trigger input (to SRF04) : Port D, pin 0
 * Echo output (to Mavric II) : Port D, pin 1
 *
 * BOTTOM MOUNTED SRF04:
 *  Pulse Trigger input (to SRF04) : Port D, pin 6
 * Echo output (to Mavric II) : Port D, pin 7
 */

#include "Ultrasound.h"

/*
 * initialize timer 1 to generate an interrupt every millisecond.
 */
void init_ultrasound_timer()
{
  //disable interrupts for the setup process
  cli();

  //Clear timer
  TCNT1 = 0x0000;

  //Flag register
  //TIFR = OCR2 TOV2 ICF1 OCF1A OCF1B TOV1 OCF0 TOV0
  TIFR |= 0x10;

  //Enable interrupt
  //TIMSK = OCIE2 TOIE2 TICIE1 OCIE1A OCIE1B TOIE1 OCIE0 TOIE0
  TIMSK |= 0x10;

  //TCCR1A = COM1A1 COM1A0 COM1B1 COM1B0 COM1C1 COM1C0 WGM11
WGM10
  //TCCR1A = 0x00;
```

```c
//TCCR1B: ICNC1 ICES1 - WGM13 WGM12 CS12 CS11 CS10
//COMA1 and COMA0 = 0 (no output pin needed on output compare)
//WGM3:0 = 0100, for CTC mode with the compare value in OCR1A
//TCCR1B = 0x09;
TCCR1B |= 0x09;

//Output compare value for a time of 50us

//Leads to 15 counts in a 1ms delay, so fires every
//67 us = resolution of ultrasound
OCR1A  = 1020;

//TCCR3A = COM3A1 COM3A0 COM3B1 COM3B0 COMB3C1 WGM31 WGM30
//TCCR3B: ICNC3 ICES3 - WGM33 WGM32 CS32 CS31 CS30

//Allow interrupts again
sei();
}

/*
 * Output Computer 1 A interrupt vector
 */
SIGNAL(SIG_OUTPUT_COMPARE1A )
{
  timed_interrupt_count++;

  //If front echo_output pin is high, bump counter
  if(  (PIND & ECHO_OUTPUT_FRONT) == ECHO_OUTPUT_FRONT ) {
        front_ultrasound_count++;
  }

  //If bottom echo output pin is high, do the same for its respective count
  if( (PIND & ECHO_OUTPUT_BOTTOM) == ECHO_OUTPUT_BOTTOM ) {
        bottom_ultrasound_count++;
  }

  //Clear flag -- needed?
  TIFR |= 0x10;
}

//Set up data direction for pins used for SRF04
void init_ultrasound_port() {

  /* Set up Port D as inputs/outputs (lowest two bits used for SRF04)
   * Bit 0 = pulse out to SRF04
   * Bit 1 = response back from SRF04
```

```
  * Bit 6 = output
  * Bit 7 = input
  */

 //1 = output, 0 = input
 DDRD = 0x7D;   // 0111 1101
 PORTD = 0x00;

 front_sonar_readings_ct = 0;
 bottom_sonar_readings_ct = 0;

 timed_interrupt_count = 0;

 //Set up timed interrupt once and use forevah
 init_ultrasound_timer();
}

//Tells the front-mounted SRF04 to send out a pulse for ranging
void send_pulse_front() {

 front_ultrasound_count = 0;

 //Set pin high and hold for 1ms
 PORTD |= PULSE_TRIGGER_FRONT;

 //2 * 67 us delay
 timed_interrupt_count = 0;
 while( timed_interrupt_count < 2 )
         ;

 //Bring pin low again, triggering sonar's signal back, so
 //start timing!
 PORTD &= (~PULSE_TRIGGER_FRONT);

 //Enable interrupt, which will automatically
 //turn off once it detects the echo back pin has gone low


 //Delay time for echo pin to go high
 while( (PIND & ECHO_OUTPUT_FRONT) != ECHO_OUTPUT_FRONT )
        ;

}

//Tells the bottom-mounted SRF04 to send out a pulse for ranging
void send_pulse_bottom() {
```

```c
  bottom_ultrasound_count = 0;

  //Set pin high and hold for 1ms
  PORTD |= PULSE_TRIGGER_BOTTOM;

  //2 * 67 us delay
  timed_interrupt_count = 0;
  while( timed_interrupt_count < 2 )
        ;

  //Bring pin low again, triggering sonar's signal back, so
  //start timing!
  PORTD &= (~PULSE_TRIGGER_BOTTOM);

  //Enable interrupt, which will automatically
  //turn off once it detects the echo back pin has gone low

  //Delay time for echo pin to go high
  while( (PIND & ECHO_OUTPUT_BOTTOM) != ECHO_OUTPUT_BOTTOM )
        ;
}

//This is a sensor update routine to be called in the main robot loop
//It checks if the last sonar pulse is finished, if so fires the next one
//and also updates the globally available sonar reading array for use elsewhere.
void update_sonar() {

        //If front-mounted sonar pulse has dropped (sonar reading is done)
        if( (PIND & ECHO_OUTPUT_FRONT) != ECHO_OUTPUT_FRONT ) {

                //Read value set by timed interrupt, now guaranteed to be finished
                //Screen the value for general limits

                if( front_ultrasound_count > 0 ) {

                        front_ultrasound = front_ultrasound_count;
                }
                //Use this later?  For now just use raw front_ultrasound_count value
                /*if( front_ultrasound_count > 0 ) {

                        //Add to globally available SRF04 array for averaging
                        //by decision-making functions
                        if( (front_sonar_readings_ct >= SONAR_ARRAY_LENGTH) ||
(front_sonar_readings_ct <= 0) ) {
                                front_sonar_readings_ct = 0;
```

```c
                  }
                  front_sonar_readings[front_sonar_readings_ct] =
front_ultrasound_count;
                  front_sonar_readings_ct++;

            }*/

            /* Debug to print current reading
            print_byte(interrupt_flag, "hex");

            lcd_byte = 0x20;
            write_data_byte();
            */

            //Since the current sonar pulse is done, trigger the next one
            send_pulse_front();
      }

      if( (PIND & ECHO_OUTPUT_BOTTOM) != ECHO_OUTPUT_BOTTOM ) {

            //Read value set by timed interrupt, now guaranteed to be finished
            //Screen the value for general limits

            if( bottom_ultrasound_count > 0 ) {

                  bottom_ultrasound = bottom_ultrasound_count;
            }
            /*if( bottom_ultrasound_count > 0 ) {

                  //Add to globally available SRF04 array for averaging
                  //by decision-making functions
                  if( (bottom_sonar_readings_ct >= SONAR_ARRAY_LENGTH) ||
(bottom_sonar_readings_ct <= 0) ) {
                        bottom_sonar_readings_ct = 0;
                  }
                  bottom_sonar_readings[bottom_sonar_readings_ct] =
bottom_ultrasound_count;
                  bottom_sonar_readings_ct++;
            }*/

            /* Debug to print current reading
            print_byte(interrupt_flag>>8, "hex");
            print_byte(interrupt_flag, "hex");

            lcd_byte = 0x20;
            write_data_byte();
```

```c
                */

                //Since the current sonar pulse is done, trigger the next one
                send_pulse_bottom();
        }

}

//Debug routine to show both sonar readings on the LCD
//Front sonar = top line, bottom sonar = 2nd line
void show_sonar(void) {

                //Print current reading
                print_byte(front_ultrasound, "hex");

                //Next line
                lcd_byte = 0xC0;
                write_control_byte();

                //Print current reading
                print_byte(bottom_ultrasound, "hex");

}


/*int main(void)
{
  // initialize LCD, with proper delays
  init_LCD();

  init_ultrasound_port();

  write_string("testing sonar...");

  uint8_t ct = 0;

  while(1) {

        update_sonar();

        //Print sonar readings for top and bottom
        clear_home();
        ct = 0;

        //Print current reading
        print_byte(front_ultrasound, "hex");
```

```
        lcd_byte = 0x20;
        write_data_byte();

        //Next line
    lcd_byte = 0xC0;
        write_control_byte();

        //Print current reading
        print_byte(bottom_ultrasound, "hex");

        lcd_byte = 0x20;
        write_data_byte();

    ms_sleep(150);

        //For obstacle avoidance main loop, add
        //threshold stuff to randomly turn
        //(print "turn" + direction ) to test

  }

  return 0;
}
*/
```

ATD_Routines.h

```
/* Basic interfacing functions for the pyroelectric sensor
 * which uses the A/D Converter, channelo (PORTF0 pin)
 *
 * Following functions/header file is an extra commented version
 * of BDMicro sample code for the atd
 */

#ifndef __ATD_Routines_h__
#define __ATD_Routines_h__

#include <avr/io.h>

/*Sets up A/D for use in the program -- run this once
 * 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)
 */
void    adc_init(void);
```

```c
/* Select the specified A/D channel for the next conversion
 * Pyro sensor uses channel 0
 */
void    adc_chsel(uint8_t channel);

/* Loop until A/D conversion is complete
 */
void    adc_wait(void);

/* Start an A/D conversion on the selected channel
   (selected with adc_chsel function above)
 */
void    adc_start(void);

/* Read the currently selected A/D Converter channel.
 */
uint16_t adc_read(void);

/* Read the specified A/D channel 'n' times and return the average of
 * the samples.
 * This function (along with adc_init earlier) is the only function that
 * will be actually called internally most of the time.
 */
uint16_t adc_readn(uint8_t channel, uint8_t n);

uint16_t adc_read_one(uint8_t n);
#endif
```

ATD_Routines.c

```c
#include "ATD_Routines.h"

/*
 * ATmega128 A/D Converter utility routines, from BDMicro
 */

/*
 * adc_init() - 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)
 */
void adc_init(void)
```

```c
{
  /* configure ADC port (PORTF) as input */
  DDRF  = 0x00;
  PORTF = 0x00;

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


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


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


/*
 * adc_start() - A/D start conversion
 *
 * Start an A/D conversion on the selected channel
 */
void adc_start(void)
{
  /* clear conversion, start another conversion */
  ADCSR |= _BV(ADIF);
}
```

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


uint16_t adc_read_one(uint8_t channel) {

  uint16_t t;

  //Set channel
  adc_chsel(channel);

  //Start conversion
  adc_start();

  //Loop until conversion complete
  adc_wait();

  //Start conversion
  adc_start();

  /* sample selected channel n times, take the average */
  t = 0;
  adc_wait();
  t = adc_read();

  return t;
}

/*
 * 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 adc_readn(uint8_t channel, uint8_t n)
{
  uint16_t t;
```

```c
  uint8_t i;

  //Set channel
  adc_chsel(channel);

  //Start conversion
  adc_start();

  //Loop until conversion complete
  adc_wait();

  //Start conversion
  adc_start();

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

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

Delay_Routines.h:

```c
/****************************************************************************
  Name: Delay_Routines
  Purpose: Gathers functions used to create delays and timed effects, such
         delaying a number of milliseconds or causing an interrupt to fire
                 every 50us.
  ****************************************************************************/
#ifndef __Delay_Routines_h__
#define __Delay_Routines_h__

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

#include <inttypes.h>

/* #define NAME (value) -- constants go here */
```

```c
/* 16-bit unsigned integer used for ms_sleep() */
volatile uint16_t ms_count;

/* 16-bit unsigned integer used for set_servo_delay() */
volatile uint16_t servo_count;

/* flag variable that should be checked from the outside.  Used to
   keep servo delays in the background (no spin)
 */
volatile uint8_t servo_delay_expired;

/* Variable for the output compare value passed in to the
 * set_servo_delay function below
 */
volatile uint16_t servo_delay_output_compare;

/*
 * initialize timer 0 and timer 2 to generate an interrupt every millisecond.
 * this should be called for setup whenever you anticipate needing the ms_sleep
 * function (e.g., whenever the LCD will be used!)
 */
void init_timer(void);

/*
 * ms_sleep() - delay for specified number of milliseconds
 * Note: Uses timer 0!
 */
void ms_sleep(uint16_t);

/*
 * Like ms_sleep(), specifies a delay in milliseconds, but
 * instead of spinning, it sets a flag (servo_delay_expired) to 1
 * when the delay expires.
 */
void set_servo_delay(uint16_t);

#endif
```

Delay_Routines.c:

```c
#include "Delay_Routines.h"

/*
 * ms_sleep() - delay for specified number of milliseconds
 */
void ms_sleep(uint16_t ms)
```

```c
{
  TCNT0  = 0;
  ms_count = 0;
  while (ms_count != ms)
    ;
}

void set_servo_delay(uint16_t amount ) {

   //Reset old value of flag
      servo_delay_expired = 0;

      TCNT2 = 0;
      servo_count = 0;
      //servo_delay_output_compare = amount;
      while( servo_count != amount )
              ;
}


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

/*
 * millisecond counter (timer 2) interrupt vector
 */
SIGNAL(SIG_OUTPUT_COMPARE2) {

  servo_count++;
  //If the servo count matches the compare value, set
  //flag variable to 1!
  if( servo_count == servo_delay_output_compare ) {

      servo_delay_expired = 1;

      //Make it so that the output compare value doesn't get hit
      //until output compare is set to valid value by calling
      //set_servo_delay again
      servo_delay_output_compare = -1;
  }
}
```

```c
/*
 * initialize timer 0 and timer 1 to generate an interrupt every millisecond.
 * this should be called for setup whenever you anticipate needing the ms_sleep
 * function (e.g., whenever the LCD will be used!)
 */
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);
  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 */

  //Set up timer 2 in an identical fashion for use by the rolling servo delay function
  TIFR |= _BV(OCIE2);
  TCCR2 = _BV(WGM21)|_BV(CS21) | _BV(CS20); // CTC, prescale = 64
  TCNT2 = 0;
  TIMSK |= _BV(OCIE2);

  //Half of 125 because prescale is twice as much (so clock is half as fast)
  OCR2 = 250;

  //Set initial value of flag to 0
  servo_delay_expired = 0;

  //Before output compare is initialized by calling set_servo_delay()
  //make it so match will never happen
  servo_delay_output_compare = -1;
}
```

LCD_Routines.h:

```
/*
 * Title: LCD_Routines
 *
 * Author: Blake C. Sutton
 *
 * Description: This is a set of functions used to initialize and write to the
 * LCD screen in 4-bit mode.  The LCD uses Port B in the following port format:
```

```
 *
 * Lower nibble (Port B 0 to 3) represents the data sent (upper nibble on LCD).
 *
 * Bits 6, 5, and 4 represent RS RW, and E, respectively.
 *
 * This can also be used to test your LCD to verify that you didn't burn it...
 * which if you are Blake can be a very useful thing :D
 *
 */

#ifndef __LCD_Routines_h__
#define __LCD_Routines_h__

#include "Delay_Routines.h"

#define SPACE      0x20

/* 8-bit unsigned integer used for passing LCD bytes
 * without parameter overhead
 */
volatile uint8_t lcd_byte;

/* 8-bit unsigned inteteger used for passing LCD output bytes
 * Without parameter overhead.  Consists of nibble + RS + RW + E bits.
 */
volatile uint8_t lcd_nibble;

/* Writes one nibble of raw information to LCD,
 * once it has been properly encoded by write_data_byte
 * or write_control_byte.
 *
 * Parameter: lcd_nibble (global variable, 8-bit unsigned integer)
 */
void write_nibble(void);

/* Writes one control byte (clear screen, next line, etc.) in 4-bit mode.
 * This should not be used for data -- use write_data_byte.
 *
 * Parameter: lcd_byte (global variable, 8-bit int)
 */
void write_control_byte(void);

/* Writes one byte of data (one ASCII char) to the LCD in 4-bit mode.
 * Thus the upper nibble is written, then
 * the lower nibble.
 *
```

```
 * Parameter: lcd_byte (global variable, 8-bit int)
 */
void write_data_byte(void) ;

/* Clears LCD screen and returns cursor to home
 */
void clear_home(void);

/* Internal Function -- do not call this, call init_LCD()
 */
void init_lcd_ports_and_timer(void) ;

/* Initialize LCD for 4-bit mode, 2-line mode.
 * When this routine returns, LCD is ready to display
 * characters.
 */
void init_LCD(void);

/*
 * Writes out the parameter string to the LCD.
 * LCD must be initialized first.
 */
void write_string(char input[]);

/* Function that takes in an 8 bit value and displays it on the LCD
 * according to the options -- ghetto printf.
 *
 * display_options is a string, as follows:
 * decimal -- display as base 10 integer NOT YET IMPLEMENTED
 * hex -- display as base 16 (hexadecimal) number (0xNN)
 * binary -- display as binary (XXXXXXXX )
 * char -- display as ASCII char (a, b, c, etc.)
 */
void print_byte(uint8_t value, char display_options[] );

#endif
```

LCD_Routines.c:

```
/*
 * $Id: LCD_Routines.c,v 1.3 2005/03/22 18:51:42 bsd Exp $
 *
 */

/*
 * Title: LCD_Routines
```

```
 *
 * Author: Blake C. Sutton
 *
 * Description: This is a set of functions used to initialize and write to the
 * LCD screen in 4-bit mode.  The LCD uses Port B in the following port format:
 *
 * Lower nibble (Port B 0 to 3) represents the data sent (upper nibble on LCD).
 *
 * Bits 6, 5, and 4 represent RS RW, and E, respectively.
 *
 * This can also be used to test your LCD to verify that you didn't burn it...
 * which if you are Blake can be a very useful thing :D
 *
 */

#include "LCD_Routines.h"

/* Writes one nibble of raw information to LCD,
 * once it has been properly encoded by write_data_byte
 * or write_control_byte.
 *
 * Parameter: lcd_nibble (global variable, 8-bit unsigned integer)
 */
void write_nibble() {
        //Pull E bit (bit 4) low and send to LCD port
        PORTB = lcd_nibble & 0xEF;

        //Pulse E high (set bit 4)
        PORTB |= 0x10;

        ms_sleep(1);

        //Pulse E back low (bit 4)
        PORTB &= 0xEF;
}

/* Writes one control byte (clear screen, next line, etc.) in 4-bit mode.
 * This should not be used for data -- use write_data_byte.
 *
 * Parameter: lcd_byte (global variable, 8-bit int)
 */
void write_control_byte() {
        //Special characteristics of control bytes:
        //      1. RS = 0
        //  2. Need a 5 ms delay between nibbles (less?)
```

```
        //First send most significant nibble
        //Shift into lower nibble for output to port
        lcd_nibble = lcd_byte >> 4;

        //Set RS = 0 for control byte (leave as is because 0 was shifted in)

        //Write nibble to LCD, with E pulse, delay.
        write_nibble();

        //Delay between nibbles needed
        ms_sleep(5);

        //Now get lower nibble (no shifting necessary, just mask off)
        lcd_nibble = lcd_byte & 0x0F;
        //Set RS = 0 for control byte (leave as is because ANded it to 0)
        write_nibble();

        ms_sleep(1);
}

/* Writes one byte of data (one ASCII char) to the LCD in 4-bit mode.
 * Thus the upper nibble is written, then
 * the lower nibble.
 *
 * Parameter: lcd_byte (global variable, 8-bit int)
 */
void write_data_byte() {
        //Special characteristics of control bytes:
        //      1. RS = 1

        //First send most significant nibble
        //Shift into lower nibble for output to port
        lcd_nibble = lcd_byte >> 4;

        //Set RS = 1 for data byte
        lcd_nibble |= 0x40;

        //Write nibble to LCD, with E pulse, delay.
        write_nibble();

        //Delay between nibbles needed?
        ms_sleep(5);

        //Now get lower nibble (no shifting necessary, just mask off)
        lcd_nibble = lcd_byte & 0x0F;
```

```
        //Set RS = 1 for data byte
        lcd_nibble |= 0x40;

        write_nibble();

        ms_sleep(1);
}

/* Clears LCD screen and returns cursor to home
 */
void clear_home() {
        lcd_byte = 0x01;
        write_control_byte();
}

void init_lcd_ports_and_timer() {
 /* Init needed for ms_sleep function */
  init_timer();

  /* enable interrupts */
  sei();

  /* Set up Port B (used for LCD) as outputs) */
  DDRB = 0xFF;
}

/* Initialize LCD for 4-bit mode, 2-line mode.
 * When this routine returns, LCD is ready to display
 * characters.
 */
void init_LCD() {

        init_lcd_ports_and_timer();

        ms_sleep(1000);

        //LCD Init
        lcd_byte = 0x33;
        write_control_byte();

        //Finish LCD Init, then set 4-bit mode
        lcd_byte = 0x32;
        write_control_byte();

        //Set to 2-line mode
        lcd_byte = 0x2C;
```

```c
        write_control_byte();

        //Turn on screen, turn on cursor, cursor blink
        lcd_byte = 0x0F;
        write_control_byte();

        //Clear home
        clear_home();
}

/*
 * Writes out the parameter string to the LCD.
 * LCD must be initialized first.
 */
void write_string(char input[]) {

  int i = 0;
  //Look for terminating null character.
  while( input[i] != '\0' ) {

    //Set global var to current char for
        //following function to access.

        //Question: does this automatically convert char to ASCII code?
    lcd_byte = input[i];
        write_data_byte();

        i++;
  }
}

/* Function that takes in an 8 bit value and displays it on the LCD
 * according to the options -- ghetto printf.
 *
 * display_options is a string, as follows:
 * decimal -- display as base 10 integer NOT YET IMPLEMENTED
 * hex -- display as base 16 (hexadecimal) number (0xNN)
 * binary -- display as binary (XXXXXXXX )
 * char -- display as ASCII char (a, b, c, etc.)
 */
void print_byte(uint8_t value, char display_options[] ) {

        if( display_options[0] == 'd' ) {

                //Not yet implemented
        }
```

```c
if( display_options[0] == 'h' ) {
        //Translate each nibble to corresponding ascii decimal digit
        //Set up 5 char array (no null until added manually)
        char hex_string[3];
        //hex_string[0] = '0';
        //hex_string[1] = 'x';

        uint8_t nibble = value >> 4;
        if( nibble > 9 ) {
                //A = 65 in decimal, so adjust to display ascii chars A to F
                hex_string[0] = ( 65 + (nibble - 10) );
        }
        else {
                //Else is a digit from 0 to 9, so simply add 0x30 to get ASCII
                hex_string[0] = ( 0x30 + nibble );
        }

        nibble = value & 0x0F;
        if( nibble > 9 ) {
                //A = 65 in decimal, so adjust to display ascii chars
                hex_string[1] = ( 65 + (nibble - 10) );
        }
        else {
                //Else is a digit from 0 to 9, so simply add 0x30 to get ASCII
                hex_string[1] = ( 0x30 + nibble );
        }

        hex_string[2] = '\0';

        write_string( hex_string );
}
if( display_options[0] == 'b' ) {

        uint8_t mask = 0x80;

        //Set up 8 char array (no null until added manually)
        char binary_string[9];

        int ct;
        for(ct = 0; ct < 8; ct++ ) {

                //Go from right (MSB) to left (LSB)
                //Shift mask as you go

                //If the bit is set (equal to mask), add a "1" to the string
```

```c
                        //Else add a "0"
                        if( (value & mask) == mask ) {
                                binary_string[ct] = 0x31;
                        }
                        else {
                                binary_string[ct] = 0x30;
                        }

                        mask = mask >> 1;
                }
                binary_string[ct] = '\0';

                write_string(binary_string);
        }
        if( display_options[0] == 'c' ) {
                //Char is already fine for sending out.
                lcd_byte = value;
                write_data_byte();
        }
}

//Quick test program for LCD
/*int main(void)
{
  // initialize LCD, with proper delays
  init_LCD();

  uint8_t temp;

  //Test display of strings, hex, and binary
  write_string("testing...");
  temp = 0xCA;

  ms_sleep(2000);
  clear_home();
  write_string( "Marco V. Downs ");

  ms_sleep(2000);
  clear_home();
  print_byte(temp, "hex");

  ms_sleep(2000);
  clear_home();
  print_byte(temp, "binary");

  return 0;
```

```
}
*/
```

Sega.h:

```
/* For the sega controller,
 */
#ifndef __Sega_h__
#define __Sega_h__

#include <avr/io.h>
#include "Delay_Routines.h"
#include "LCD_Routines.h"
#include "Pyro.h"

//When a sega button is pressed, it grounds, causing a logical 0



//Inputs (buttons / dpad switches)
//These are the locations of the bits, for comparison with the port
#define START        ~0x01
#define BUTTON_A ~0x08

#define LEASH_LEFT       ~0x04
//#define UP            ~0x02
//#define DOWN        ~0x04
#define LEASH_RIGHT      ~0X02

void init_sega_port(void);

#endif
```

Sega.c:

```
/*
 * Sega Controller
 *
 * This is a set of functions for interfacing the Sega Genesis controller / joystick
 * with the rest of the program
 *
 *
 */

#include "Sega.h"
```

```c
void init_sega_port(void) {

        DDRC =  0x00; // 0000 0000
}

/*
int main(void) {

        init_LCD();
        adc_init();
        init_sega_port();

        ms_sleep(5000);
        uint16_t current_sega_reading;

        while(1) {

                clear_home();

                print_byte(PINC, "binary");

                lcd_byte = 0xC0;
                write_control_byte();

                //Read and avg samples from pyro channel
                current_sega_reading = adc_readn(3, NUMBER_SAMPLES);

                print_byte(current_sega_reading, "hex");

                lcd_byte = 0x20;
                write_data_byte();

                current_sega_reading = adc_readn(2, NUMBER_SAMPLES);
                print_byte(current_sega_reading, "hex");

                ms_sleep(250);

        }

        return 0;
}*/
```

Pyro.h:

/* This is a set of routines for reading data from the

```
 * pyroelectric sensor, using a configurable channel of the on-board
 * ATD.
 *
 * To access and set up the ATD, BDMicro's ATD library functions
 * are used.  The most important ones for use in this module are
 * adc_init(), called once to initialize the ATD, and adc_readn(channel, n),
 * which returns the average of n readings converted from the channel parameter.
 *
 */

#ifndef __Pyro_h__
#define __Pyro_h__

#include <avr/io.h>
#include "Delay_Routines.h"
#include "LCD_Routines.h"
#include "ATD_Routines.h"
#include "Servo.h"

//If the channel is changed from 0, need to also physically change the pin
//that the pyro output comes in on, and also makes a few adjustments to
//adc_init() in ATD_Routines
#define PYRO_CHANNEL 0
#define NUMBER_SAMPLES  8

void read_pyro(void);

#endif
```

Pyro.c:

```
#include "Pyro.h"

void read_pyro(void) {

        pyro = adc_readn(PYRO_CHANNEL, NUMBER_SAMPLES);

        if( (pyro >= 0x219) && (pyro <= 0x300) ) {
                person_found = 0xFF;
        }
        else if( (pyro >= 0x100) && (pyro <= 0x1E7) ) {
                person_found = 0xFF;
        }
        else {
                person_found = 0x00;
        }
```

```
}
//Test program for ATD
/*int main(void)
{

        init_LCD();
        write_string("dude?");

        ms_sleep(5000);

        //Initialize ATD
        adc_init();

        uint16_t current_pyro_reading;

        while(1) {
                clear_home();

                //Read and avg samples from pyro channel
                current_pyro_reading = adc_readn(PYRO_CHANNEL,
NUMBER_SAMPLES);

                //Test idea -- if reading is >= 300 or <= 200, display something different
                if(current_pyro_reading >= 0x219 ) {
                        write_string("motion left!");
                }
                else if( current_pyro_reading <= 0x1E7 ) {
                        write_string("motion right!");
                }
                else {

                        write_string("nada!");
                }
                        lcd_byte = SPACE;
                        write_data_byte();
                        //Display reading average (in hexadecimal) on LCD
                        print_byte(current_pyro_reading>>8, "hex");
                        print_byte(current_pyro_reading, "hex");

                //Sleep to avoid flicker
                ms_sleep(50);
        }

        return 0;
}
*/
```

Servo.h:

```
/* For each servo, the pulse width is set as follows corresponding to
 * the output compare register:
 *
 *     Pulse Width    OCR3x/OCR1x
 *     1.0 ms        2000
 *     1.5 ms        3000
 *     2.0 ms        4000
 *
 */

#ifndef __Servo_h__
#define __Servo_h__

#include <avr/io.h>
#include "Delay_Routines.h"
#include "LCD_Routines.h"
#include "Ultrasound.h"
#include "Sega.h"

#define LEFT_SERVO   OCR3A
#define RIGHT_SERVO   OCR3B
#define CENTER_SERVO   OCR3C

#define SERVO1   OCR3A
#define SERVO2   OCR3B
#define SERVO3   OCR3C

#define SERVO_MID 3000
#define SERVO_MIN 2000
#define SERVO_MAX 4000

//Higher numbers move the right servo toward the front of hte robot
//Lower numbers move it back
#define RIGHT_CENTER   2600
#define RIGHT_FORWARD       3200
#define RIGHT_BACKWARD      2000


//Higher numbers move the left servo toward the back of the robot
//lower numbers move it to the front
#define LEFT_CENTER    3700
#define LEFT_FORWARD        3100
#define LEFT_BACKWARD   4300
```

```
#define CENTER_CENTER      4200
#define RIGHT_LEGS_UP   4760
#define LEFT_LEGS_UP   3600

uint8_t stopped;
uint8_t person_found;
uint8_t dancing;
uint16_t pyro;

void respond_to_buttons(void);
void servo(int16_t s1, int16_t s2, int16_t s3);

//Sets up servo pwm.  After this point, servos can be controlled by writing
//different values to the OCR3 registers, aliased above as SERVO1, SERVO2, SERVO3
//Note the servo control goes out on the pins attached to OC3 -- Port E pins 3, 4, and 5
void init_servo_pwm(void);

void walk_backward(void);
void turn_right(void);
void turn_left(void);
void itch(void);
void look_for_people(void);

//Testing and/or novelty functions

#endif


Servo.c:


#include "Servo.h"

void init_servo_pwm(void) {

        // set up timer3 (16 bit) to act as a triple channel PWM generator
        // we want OC3A, B, and C to be set on reaching BOTTOM, clear on reaching
        // compare match, use ICR1 as TOP and have a prescale of 8 (clock goes to
2Mhz).
        TCCR3A = _BV(COM3A1) // set OC3A/B/C at TOP
                | _BV(COM3B1) // clear OC1A/B/C when match
                | _BV(COM3C1)
                | _BV(WGM11); // mode 14 (E) (fast PWM, clear TCNT1 on match ICR1)

        TCCR3B = _BV(WGM33)
```

```
                | _BV(WGM32)
                | _BV(CS31); // timer uses main system clock with 1/8 prescale

        ICR3 = 40000; // used for TOP, makes for 50 hz PWM @ 2MHz clock

        //Note that for the Mavric II B board, OC3A/B/C are at pins 3,4,5 of Port E
        DDRE   |= _BV(PORTE3) | _BV(PORTE4) | _BV(PORTE5);

        LEFT_SERVO = LEFT_CENTER;
        RIGHT_SERVO = RIGHT_CENTER;
        CENTER_SERVO = CENTER_CENTER;
}

//Tests left servo: loops moving from center to forward and backward positions
//Calibration works only for left servo
void test_left_servo(void) {

        int center = LEFT_CENTER;
        int full_backwards = LEFT_BACKWARD;
        int full_forwards = LEFT_FORWARD;

        LEFT_SERVO = center;
        ms_sleep(1000);

        while(1) {

                //Move legs forward
                LEFT_SERVO = full_forwards;

                //Wait a second
                ms_sleep(1000);

                LEFT_SERVO = center;

                //Wait
                ms_sleep(1000);

                LEFT_SERVO = full_backwards;

                ms_sleep(1000);

                LEFT_SERVO = center;

                ms_sleep(1000);
        }
}
```

```c
//Tests right servo: loops moving from center to forward and backward positions
//Calibration works only for left servo
void test_right_servo(void) {

        int center = RIGHT_CENTER;
        int full_forwards = RIGHT_FORWARD;
        int full_backwards = RIGHT_BACKWARD;

        RIGHT_SERVO = center;
        ms_sleep(1000);

        while(1) {

                //Move legs forward
                RIGHT_SERVO = full_forwards;

                //Wait a second
                ms_sleep(1000);

                RIGHT_SERVO = center;

                //Wait
                ms_sleep(1000);

                RIGHT_SERVO = full_backwards;

                ms_sleep(1000);

                RIGHT_SERVO = center;

                ms_sleep(1000);
        }
}

//Moves side servos sort of in synch (1 second pauses between changes)
void test_side_servos(void) {
        //Note: left servo goes to channel A (pin 4)
        //      right servo goes to channel B (pin 5)

        LEFT_SERVO = LEFT_CENTER;
        RIGHT_SERVO = RIGHT_CENTER;
        ms_sleep(1000);

        while(1) {
```

```c
        //Move right side forward and left side backward
        RIGHT_SERVO = RIGHT_FORWARD;
        LEFT_SERVO = LEFT_BACKWARD;

        //Wait a second
        ms_sleep(1000);

        //Center again
        LEFT_SERVO = LEFT_CENTER;
        RIGHT_SERVO = RIGHT_CENTER;

        //Wait
        ms_sleep(1000);

        //Move right side backward and left side forward
        RIGHT_SERVO = RIGHT_BACKWARD;
        LEFT_SERVO = LEFT_FORWARD;

        ms_sleep(1000);

        //Center
        LEFT_SERVO = LEFT_CENTER;
        RIGHT_SERVO = RIGHT_CENTER;

        ms_sleep(1000);
    }
}

//Moves side servos to mimic walking (1 second pauses between changes)
void test_stride(void) {
    //Note: left servo goes to channel A (pin 4)
    //      right servo goes to channel B (pin 5)

    LEFT_SERVO = LEFT_CENTER;
    RIGHT_SERVO = RIGHT_CENTER;
    ms_sleep(1000);

    while(1) {

        //Move right side forward and left side backward
        RIGHT_SERVO = RIGHT_FORWARD;
        LEFT_SERVO = LEFT_BACKWARD;

        //Wait a second
        ms_sleep(1000);
```

```c
        //Move right side backward and left side forward
        RIGHT_SERVO = RIGHT_BACKWARD;
        LEFT_SERVO = LEFT_FORWARD;

        ms_sleep(1000);
    }
}


void test_center(void) {

        //Move legs up
        CENTER_SERVO = RIGHT_LEGS_UP;

        if( (PINC | BUTTON_A ) == BUTTON_A ) {

                clear_home();
                write_string("mmmmmmm...");

                itch();

                dancing = 0x00;
                stopped = 0x00;
                return;
        }

        //Wait a second
        ms_sleep(250);

        if( (PINC | BUTTON_A ) == BUTTON_A ) {

                clear_home();
                write_string("mmmmmmm...");

                itch();

                dancing = 0x00;
                stopped = 0x00;
                return;
        }

        ms_sleep(250);

        if( (PINC | BUTTON_A ) == BUTTON_A ) {

                clear_home();
```

```
                        write_string("mmmmmmm...");

                        itch();

                        dancing = 0x00;
                        stopped = 0x00;
                        return;
}

        ms_sleep(250);

                        if( (PINC | BUTTON_A ) == BUTTON_A ) {

                        clear_home();
                        write_string("mmmmmmm...");

                        itch();

                        dancing = 0x00;
                        stopped = 0x00;
                        return;
}

        ms_sleep(250);

        //CENTER_SERVO = CENTER_CENTER;
        //Wait
        //ms_sleep(1000);

        CENTER_SERVO =  LEFT_LEGS_UP;

        if( (PINC | BUTTON_A ) == BUTTON_A ) {

                clear_home();
                write_string("mmmmmmm...");

                itch();

                dancing = 0x00;
                stopped = 0x00;
                return;
}

        ms_sleep(250);

        if( (PINC | BUTTON_A ) == BUTTON_A ) {
```

```c
            clear_home();
            write_string("mmmmmmm...");

            itch();

            dancing = 0x00;
            stopped = 0x00;
            return;
        }

        ms_sleep(250);

        if( (PINC | BUTTON_A ) == BUTTON_A ) {

            clear_home();
            write_string("mmmmmmm...");

            itch();

            dancing = 0x00;
            stopped = 0x00;
            return;
        }

        ms_sleep(250);
        if( (PINC | BUTTON_A ) == BUTTON_A ) {

            clear_home();
            write_string("mmmmmmm...");

            itch();

            dancing = 0x00;
            stopped = 0x00;
            return;
        }

        ms_sleep(250);

        //CENTER_SERVO = CENTER_CENTER;
        //ms_sleep(1000);
}

void avoid_obstacle(void) {
```

```
        //clear_home();
        //write_string("avoiding...");

        walk_backward();
        walk_backward();

        if( (TCNT0 & 0x00) == 0x00 ) {
                turn_right();
                turn_right();
                turn_right();
                turn_right();
        }
        else {
                turn_left();
                turn_left();
                turn_left();
                turn_left();
        }
}

void check_for_obstacle(void) {

        //Section where sonar is updated and checked.
        update_sonar();

        //Section where pyro is updated and checked
        read_pyro();

        //Print sonar readings for top and bottom
        if( bottom_ultrasound > MAX_HEIGHT ) {
                clear_home();
                write_string("Cliff! ");
                print_byte(bottom_ultrasound, "hex");

                if( stopped == 0x00 ) {
                        avoid_obstacle();
                }
        }

        if(front_ultrasound < NEAR ) {
                clear_home();
                write_string("Too close! ");
                print_byte(front_ultrasound, "hex");

                if( stopped == 0x00 ) {
                        avoid_obstacle();
```

```c
                }
        }

        respond_to_buttons();

        look_for_people();
}

void respond_to_buttons() {

        if(  (PINC | START ) == START ) {
                //If robot is not stopped, pauses, otherwise resumes
                clear_home();
                write_string("stopped");
                stopped = ~stopped;
        }

        //Temporary value of flag
        uint8_t old_stopped = stopped;

        //Leash pull to right
        if( (PINC | LEASH_RIGHT) == LEASH_RIGHT ) {

                clear_home();
                write_string("pull right");

                stopped = 0x00;
                turn_right();
                stopped = old_stopped;
        }

        //Leash pull to left
        if( (PINC | LEASH_LEFT) == LEASH_LEFT ) {

                clear_home();
                write_string("pull left");

                stopped = 0x00;
                turn_left();
                stopped = old_stopped;
        }

        //Scratch behind the ear
        //She responds even if stopped
        if( (PINC | BUTTON_A ) == BUTTON_A ) {
```

```
                clear_home();
                write_string("mmmmmmm...");

                /*CENTER_SERVO = RIGHT_LEGS_UP;
                ms_sleep(3000);
                CENTER_SERVO = CENTER_CENTER;
                ms_sleep(1000);
                */
                itch();
        }
}


//Walks forward w/ trigate gait
void walk_forward(void) {

        //250 works
   int delay = 100;
        int delay2 = 250;

        if( stopped == 0xFF ) {
                return;
        }

                //Tilts to side so right legs are in the air (or loose)
                CENTER_SERVO = RIGHT_LEGS_UP;

                //Push right set of legs forward
                RIGHT_SERVO = RIGHT_FORWARD;

                //Push left set of legs backwards (pushing robot forward)
                LEFT_SERVO = LEFT_BACKWARD;

                ms_sleep(delay2);

                //If not stopped, check for obstacle
                if( stopped == 0xFF ) {
                        return;
                }

                        check_for_obstacle();

                        //All legs back on the ground
                        CENTER_SERVO = CENTER_CENTER;
                        ms_sleep(delay);
```

```c
        if( stopped == 0xFF ) {
            return;
        }

            check_for_obstacle();

            //Tilts to side so left legs are in the air (or loose)
            CENTER_SERVO = LEFT_LEGS_UP;

            //Push left set of legs forward
            LEFT_SERVO = LEFT_FORWARD;

            //Push right set of legs backwards (pushing robot forward)
            RIGHT_SERVO = RIGHT_BACKWARD;

            ms_sleep(delay2);


        if( stopped == 0xFF ) {
            return;
        }
            check_for_obstacle();

            //All legs back on the ground
            CENTER_SERVO = CENTER_CENTER;

            ms_sleep(delay);

        if( stopped == 0xFF ) {
            return;
        }

        check_for_obstacle();

}

//Brings all servos to their respective center positions (neutral)
void center_all_servos(void) {

    CENTER_SERVO = CENTER_CENTER;
    LEFT_SERVO = LEFT_CENTER;
    RIGHT_SERVO = RIGHT_CENTER;
}

void walk_backward(void) {
```

```
if( stopped == 0xFF ) {
        return;
}
        //clear_home();
        //write_string("back up");

        int delay = 100;
        int delay2 = 250;

        //Tilts to side so right legs are in the air (or loose)
        CENTER_SERVO = RIGHT_LEGS_UP;

        //Push right set of legs forward
        RIGHT_SERVO = RIGHT_BACKWARD;

        //Push left set of legs backwards (pushing robot forward)
        LEFT_SERVO = LEFT_FORWARD;

        ms_sleep(delay2);

        if( stopped == 0xFF ) {
                return;
        }

        //All legs back on the ground
        CENTER_SERVO = CENTER_CENTER;

        ms_sleep(delay);

        if( stopped == 0xFF ) {
                return;
        }

        //Tilts to side so left legs are in the air (or loose)
        CENTER_SERVO = LEFT_LEGS_UP;

        //Push left set of legs forward
        LEFT_SERVO = LEFT_BACKWARD;

        //Push right set of legs backwards (pushing robot forward)
        RIGHT_SERVO = RIGHT_FORWARD;

        ms_sleep(delay2);

        if( stopped == 0xFF ) {
                return;
```

```
        }

        //All legs back on the ground
        CENTER_SERVO = CENTER_CENTER;

        ms_sleep(delay);

}

//Has robot turn left one pace (left side moves backward, right side moves forward)
void turn_left(void) {
/* Turn Left Sequence
, LEFT_LEGS_UP, CENTER_CENTER, RIGHT_LEGS_UP, CENTER_CENTER
, RIGHT_FORWARD, RIGHT_FORWARD, RIGHT_BACKWARD, RIGHT_BACKWARD
, LEFT_FORWARD, LEFT_FORWARD, LEFT_BACKWARD, LEFT_BACKWARD
*/
        if( stopped == 0xFF ) {
                return;
        }

                int delay = 100;
                int delay2 = 250;

                //clear_home();
                //write_string("turn left");

                CENTER_SERVO = LEFT_LEGS_UP;
                RIGHT_SERVO = RIGHT_FORWARD;
                LEFT_SERVO = LEFT_FORWARD;

                ms_sleep(delay2);

                if( stopped == 0xFF ) {
                        return;
                }

                check_for_obstacle();

                CENTER_SERVO = CENTER_CENTER;

                ms_sleep(delay);

                if( stopped == 0xFF ) {
                        return;
                }
```

```c
        check_for_obstacle();

        CENTER_SERVO = RIGHT_LEGS_UP;
        LEFT_SERVO = LEFT_BACKWARD;
        RIGHT_SERVO = RIGHT_BACKWARD;

        ms_sleep(delay2);

        if( stopped == 0xFF ) {
                return;
        }

        check_for_obstacle();

        CENTER_SERVO = CENTER_CENTER;

        ms_sleep(delay);

        if( stopped == 0xFF ) {
                return;
        }

        check_for_obstacle();
}

//Has robot turn right one pace (right side moves backward, left side moves forward)
void turn_right(void) {
/*, LEFT_LEGS_UP, CENTER_CENTER, RIGHT_LEGS_UP, CENTER_CENTER
, RIGHT_BACKWARD, RIGHT_BACKWARD, RIGHT_FORWARD, RIGHT_FORWARD
, LEFT_BACKWARD, LEFT_BACKWARD, LEFT_FORWARD, LEFT_FORWARD
*/
        if( stopped == 0xFF ) {
                return;
        }

                int delay = 100;
                int delay2 = 250;

                //clear_home();
                //write_string("turn right");

                CENTER_SERVO = LEFT_LEGS_UP;
                RIGHT_SERVO = RIGHT_BACKWARD;
                LEFT_SERVO = LEFT_BACKWARD;

                ms_sleep(delay2);
```

```
        if( stopped == 0xFF ) {
                return;
        }

        check_for_obstacle();

        CENTER_SERVO = CENTER_CENTER;

        ms_sleep(delay);

        if( stopped == 0xFF ) {
                return;
        }

        check_for_obstacle();

        CENTER_SERVO = RIGHT_LEGS_UP;
        LEFT_SERVO = LEFT_FORWARD;
        RIGHT_SERVO = RIGHT_FORWARD;

        ms_sleep(delay2);

        if( stopped == 0xFF ) {
                return;
        }

        check_for_obstacle();

        CENTER_SERVO = CENTER_CENTER;

        ms_sleep(delay);

        if( stopped == 0xFF ) {
                return;
        }

        check_for_obstacle();
}

void test_person_detection(void) {
        update_sonar();
        read_pyro();

        clear_home();
        if( (person_found == 0xFF) && (front_ultrasound < 0x1A ) ) {
```

```c
                        write_string("hi!");
                }
                else {
                        write_string("i'm lonely");
                }

                lcd_byte = 0xC0;
                write_control_byte();

                print_byte( pyro>>8, "hex");
                print_byte( pyro, "hex");

                lcd_byte = 0x20;
                write_data_byte();

                print_byte(front_ultrasound_count, "heX");
}

//Call this function everywhere you call check_for_obstacle
void look_for_people(void) {

                read_pyro();
                update_sonar();

                if( (person_found == 0xFF) && (front_ultrasound < 0x1A ) &&
(front_ultrasound > 0x06 ) ) {
                        clear_home();
                        write_string("hi!");

                        //Stop and put in dancing mode
                        stopped = 0xFF;
                        dancing = 0xFF;
                        return;
                }
                else {
                        dancing = 0x00;
                        stopped = 0x00;
                }
}

void itch( void ) {

        //Go up on right legs and waggle right leg servos twice
        CENTER_SERVO = RIGHT_LEGS_UP;

        ms_sleep(1000);
```

```c
        RIGHT_SERVO = RIGHT_BACKWARD;

        ms_sleep(500);

        RIGHT_SERVO = RIGHT_CENTER;

        ms_sleep(500);

        RIGHT_SERVO = RIGHT_BACKWARD;

        ms_sleep(500);

        RIGHT_SERVO = RIGHT_CENTER;

        ms_sleep(1000);

        CENTER_SERVO = CENTER_CENTER;

}

//Test program for walking forward on a sequence only
int main(void)
{

        //Initialization
        init_LCD();
        init_sega_port();
        adc_init();
        init_ultrasound_port();
        init_servo_pwm();

        //Initialize modes
        stopped = 0x00;
        dancing = 0x00;
        person_found = 0x00;

        //Initial lcd string
        clear_home();
        write_string("start me up ;)");

        //Wait til Start button is pressed
        while( (PINC | START ) != START )
                ;

        while(1) {
```

```
        update_sonar();

        //Always respond to buttons, even if stopped
        //This includes the leash button
        respond_to_buttons();

        //Looks for people in front of it to dance for
        //Once in dance mode, does not stop until A button is pressed
        look_for_people();

        //Walk forward (and do other movement)
        //only if not currently stopped
        if( stopped == 0xFF ) {
        }
        else {
                walk_forward();
        }

        //If in dance mode, dance until
        //A button is pressed
        if( dancing == 0xFF ) {

                //Lift right legs and waggle
                //check often for pressing of A button
                test_center();
        }

    }

    return 0;
}
```