Abstract

BuggyBot is a robot that runs to its darkened home when it sees an unexpected light source. BuggyBot avoids obstacles and finds its home using a microcontroller brain and several types of sensors. It moves on two wheels in the front, each driven by a hacked servo, with a caster in the back.

Executive Summary

BuggyBot is an autonomous robot that mimics the insect behavior of running from unexpected light sources. When such a light source hits BuggyBot, it searches for north, moves forward when it has the appropriate heading, and stops when it has found the darkness of its home colony. It accomplishes this behavior through an ATXmega128 board interfaced with a sensor suite including three infrared sensors, two bump sensors, a CdS cell, and a 3D compass. BuggyBot moves with a two-wheel, single caster setup. Hacked servos drive the two wheels, situated in the front of the robot, with the caster in the back. The electronics are mounted on a small wooden platform. The overall system for this project includes not only BuggyBot but also an arena to give it limited space to move around, a garage at the north end of the arena to provide the home/safe zone with enough darkness, and an external light source such as a flashlight.

Introduction

BuggyBot was inspired by a very common Florida phenomenon: turning on the kitchen light in the middle of the night and seeing any bugs in the area scatter to darker places. Given this inspiration, BuggyBot's objective is highly behaviorally oriented. It does not look like an insect, but it can act like one in one respect: fear of light.

Insects are intelligent creatures and react in many ways to variations in their environment. BuggyBot will, like most bugs, react to obstacles, dangers, and safe zones. Obstacle avoidance, safe zone detection, and danger detection will come from standard sensors. What BuggyBot does when threatened will be implemented with a special system. BuggyBot has a simple mechanical structure. It moves about its environment looking for the darkest place it can find (i.e., a good place for a nocturnal bug to hang out). When threatened, it will locate its home and scurry to safety.

First, this paper will provide a complete description of the system and describe how it meets the above specifications. The next topic is the mobile platform and how it relates to the objectives. The paper then covers BuggyBot's actuation and how it is achieved. Following will come sections describing sensors used and behaviors, followed by a section on experimental data. Finally, the paper will conclude with a summary of the work accomplished and limitations encountered. Documentation and appendices with supplemental material are provided following the conclusion.

Integrated System

BuggyBot mimics insect behavior, maneuvering about its environment and finding a dark place to rest at its home when threatened. The system consists of BuggyBot, an arena with a darkened area, and an external light source (other than ambient light).

The arena fully contains BuggyBot and close it off from outside world objects; only light from the outside would affect it. The covered area needs to be dark enough for BuggyBot to detect it as a safe zone, and it should lie roughly in the north part of the arena. BuggyBot's sensors allow it to collect data on the light level in the environment. If a light more intense than the ambient light hits it, it will stop and wait for the light to disappear. BuggyBot should then spin in a circle until it finds north, the direction of its home. Next BuggyBot would run to its home, avoiding obstacles and rediscovering north when it loses track, until it finds a place dark enough to call safe. Here, BuggyBot should stop and wait to be reset.

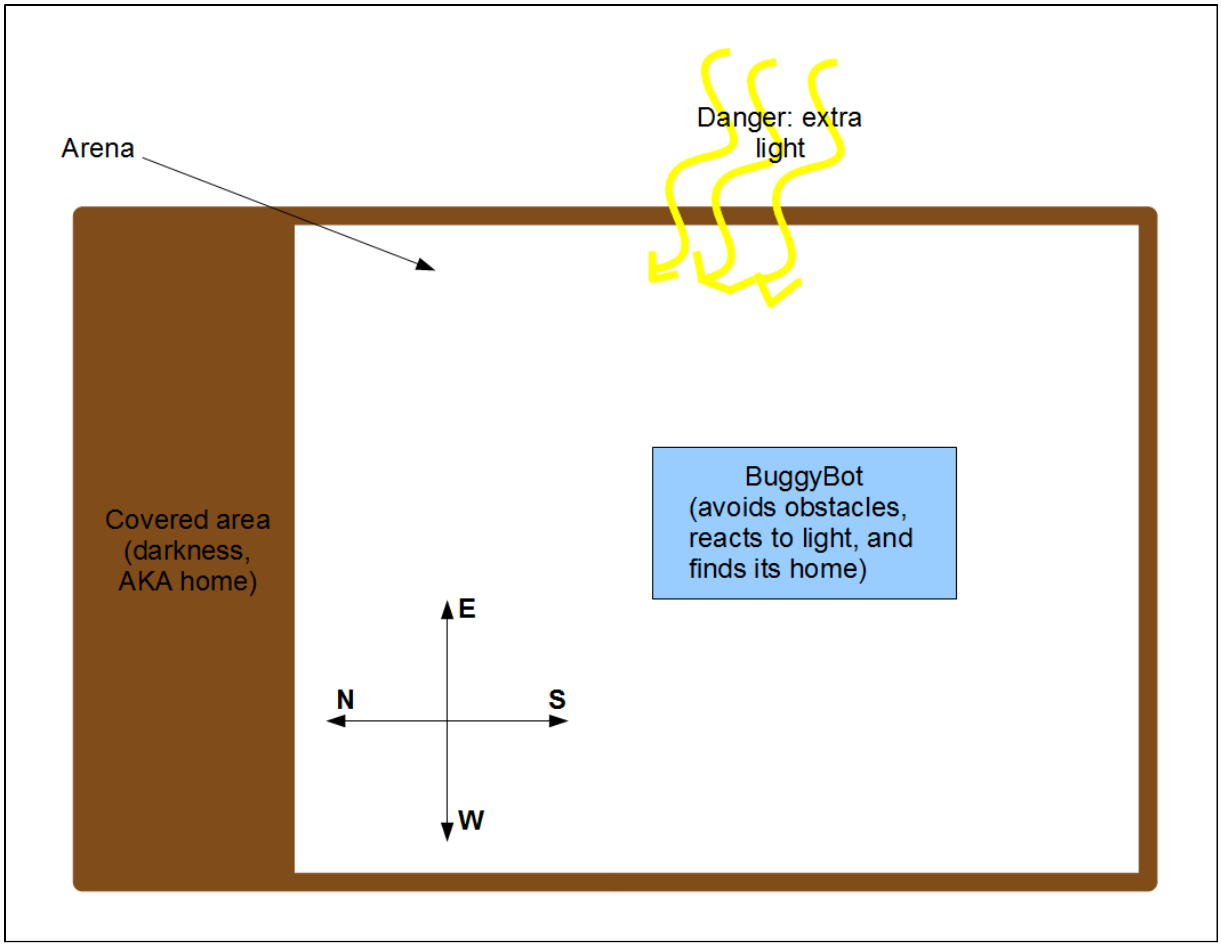


Figure IS1: Overall System Block Diagram

Inside the overall integrated system is the system that is BuggyBot. This system consists of many non-electronic parts (including the wooden platform, servo wheels, a caster, mounting brackets, and various screws), two Hitec HS-311 standard servos, three Sharp GP2D120XJ00F short-range infrared sensors, one CdS cell circuit, two bump switches, one LSM303DLH compass (accelerometer not used), one PVR Xmega microcontroller board, an LCD for information, and the on/off switch. See Figure IS2 for a block diagram of the electronic parts of the system.

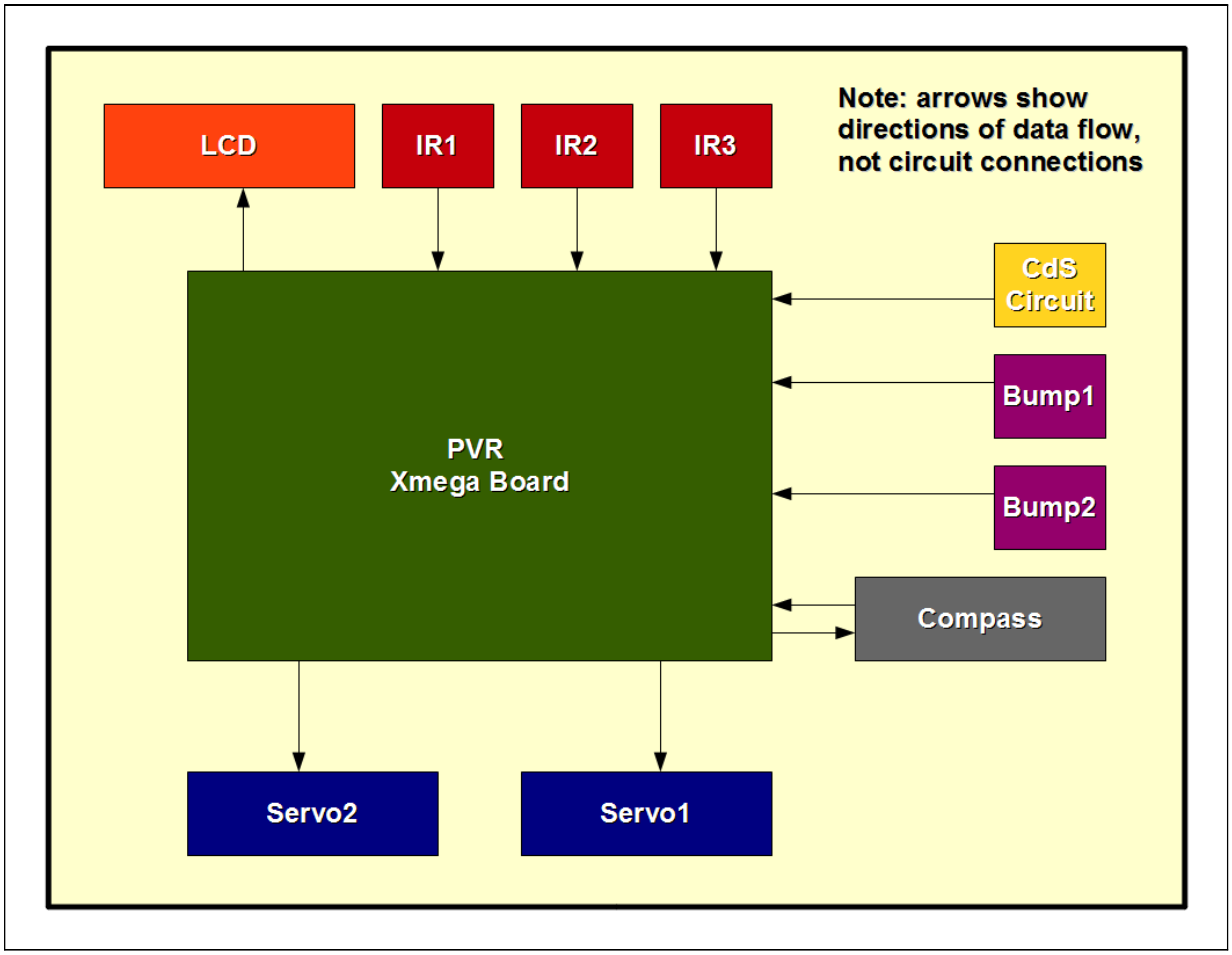


Figure IS2: Block diagram of electronic system

Mobile Platform

The mobile platform for this project is simple and flexible since the objectives are primarily behavioral rather than mechanical. It consists of a rounded, wooden base, approximately 8 inches in diameter, on two front wheels with a ball caster behind. Hacked servos drive the wheels. I learned that simpler platforms are great if you don't know exactly how you want to mount everything from the start. I had to flip my platform upside down and turn it around, but it worked out and I didn't need to re-make any wooden parts since my design wasn't so complicated that either of these things would affect it. See Figure MP1 for the final platform with electronics mounted. The platform was originally designed in Solidworks.

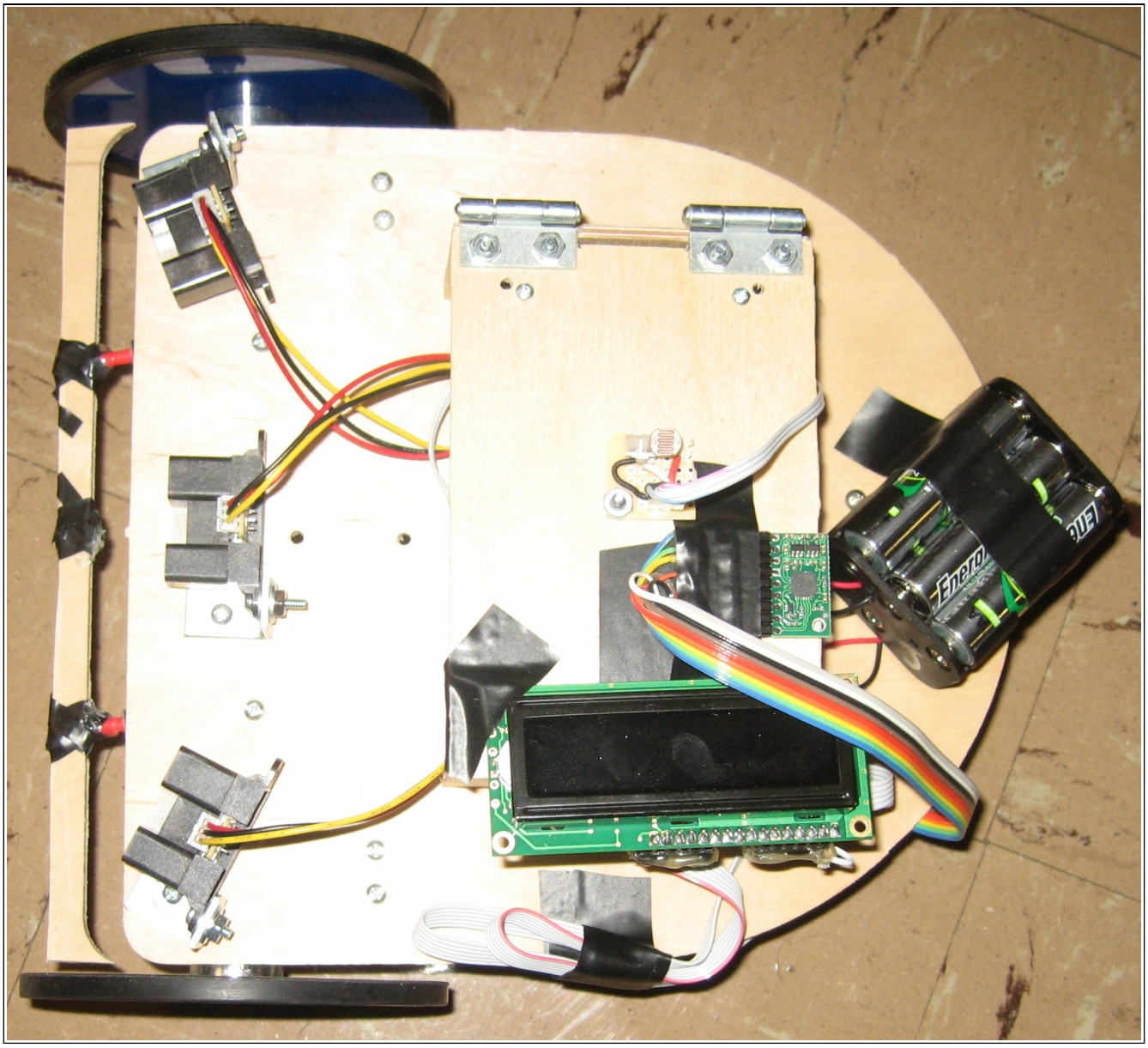


Figure MP1: Top view of completed platform

Actuation

BuggyBot's actuation comes from two hacked servos mounted to the front of the base, which are driven by the PVR Xmega board. BuggyBot navigates its environment on wheels, with sensory input telling the servos which way the robot should turn to reach its goal. One thing I learned was that when hacking servos, it is a good idea to make the potentiometer accessible from the outside. This has allowed me to adjust the servos so that when I tell them to stop, they can always find the zero position.

Hacked servos were chosen because they are less expensive and easier to use than motors. Also, the fact that BuggyBot is lightweight and its movements are not mechanically complicated contributed to this choice. One other factor is the torque needed for BuggyBot to move. Based on the formula $T = \frac{Wr\mu}{n}$, with W = weight = 1kg, r = radius of a wheel = 5.08 cm, mu = coefficient of friction = .3,

and n = number of wheels = 2, the necessary torque was found to be approximately .075 N*m, or 10.6 oz-in. The HS-311 has a torque of 42 oz-in when powered with 4.8 V, which makes it more than enough to drive BuggyBot.

As BuggyBot receives input from the infrared sensors, CdS cell, bump sensors, and compass, its microcontroller brain will react to these stimuli by telling the servos how to turn. For example, if the infrared sensors report an obstacle in one direction, the microcontroller will tell the servos how to turn to avoid the obstacle through pulse width modulation. Also, if the CdS cell has seen a strong light and BuggyBot is fleeing, when the dark area is reached the servos will be told to go to the zero position so that BuggyBot will stop moving. A high input from the bump sensors will illicit a similar response to the servos, but BuggyBot's servos will have to back it up before it can turn away. In the event that while BuggyBot is running away it loses its northerly heading, the servos receive signals that turn BuggyBot in a circle until it sees north again.

For the wheels to both be the same distance from the front of the robot, one of the servos had to be mounted upside down compared to the other. This means that when moving forward, one servo receives one direction to go to, while its counterpart receives the opposite. The two servos are connected to ports C0 and D0 on the PVR board. See Figure A1 for the basic actuation algorithm. See Appendix A for the C code relating to actuation.

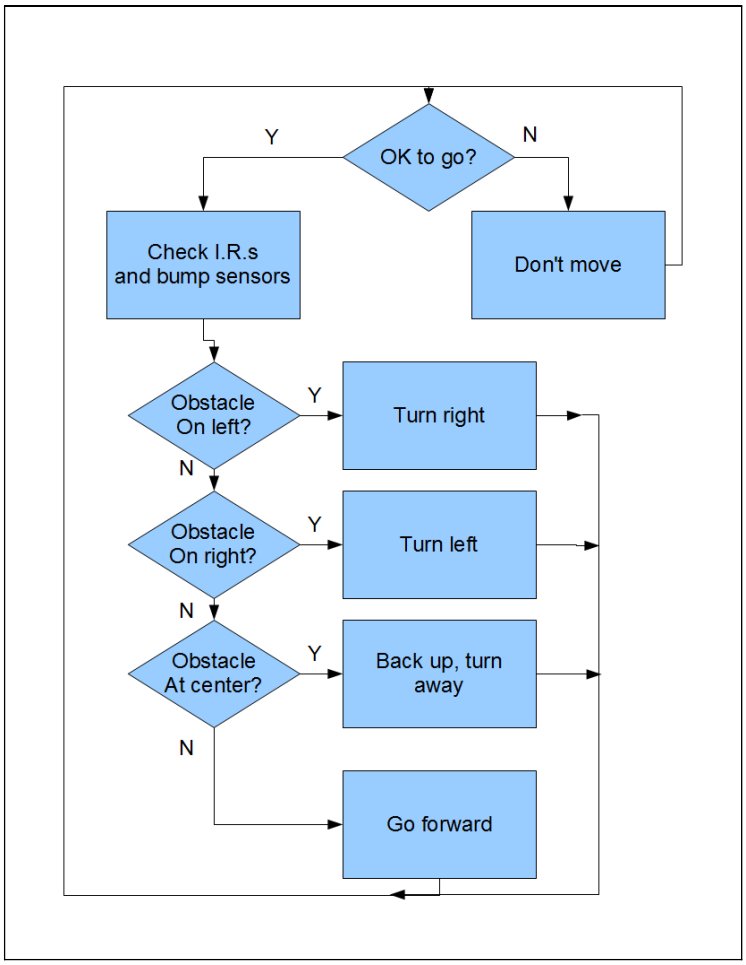


Figure A1: Actuation flow chart

Sensors

BuggyBot has three standard sensors: infrared for obstacle avoidance (Sharp GP2D120XJ00F, obtained from Sparkfun at www.sparkfun.com), CdS cells (purchased from RadioShack) for light detection, and bump sensors (part number KSM06330, from Jameco at www.jameco.com) for sensing touch. BuggyBot's special sensor is an LSM303DLH compass (from Pololu at www.pololu.com. See Appendix A for the actual code relating to these sensors.

The bump sensors related to obstacle detection are made with tactile switches attached to a protruding piece of wood that is the same width as the robot. Mounted on the front, these switches tell BuggyBot when it runs into something and give it a direction to move in based on whether the impact was on the left, the right, or in the center.

The first sensor test performed for this project was a plot of the voltages for each of three Sharp IR sensor modules. For each sensor, a ruler was taped to the table and the IR sensor was placed at a height similar to where it would be when mounted on BuggyBot. Then, readings were taken with a sheet of paper moved two centimeters further away from the last measurement. See Figures S1, S2, and S3 for the results for IR's 1, 2, and 3. These IR's are mounted, when facing BuggyBot's front, at the left, center, and right, respectively.

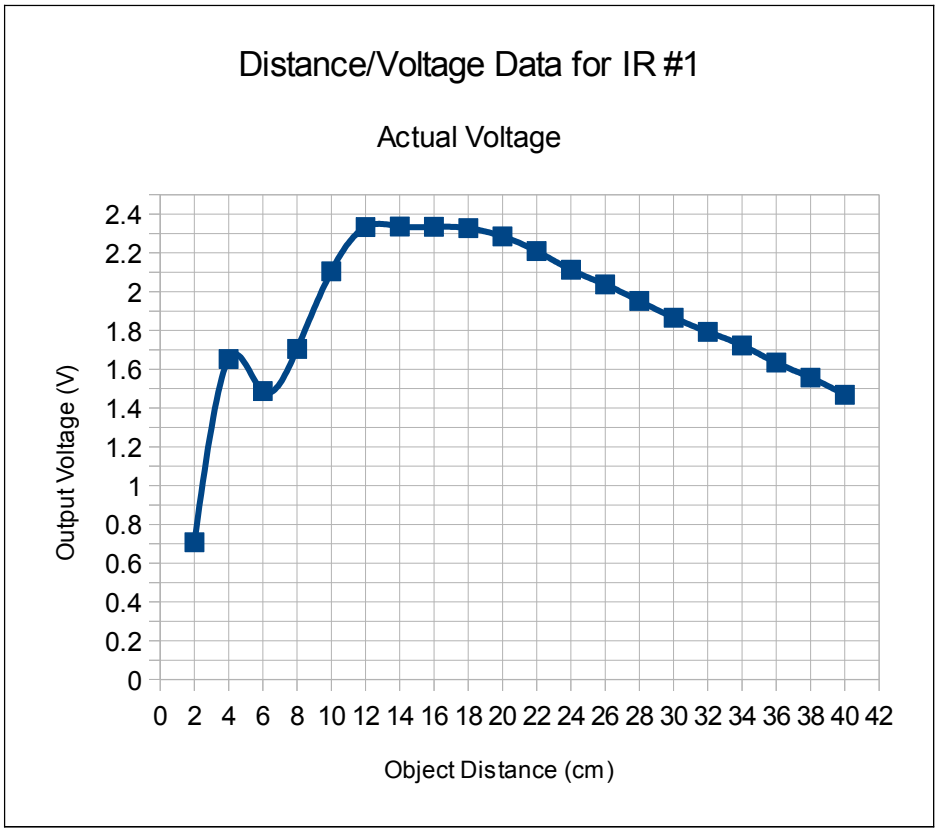


Figure S1: IR #1 distance-voltage data

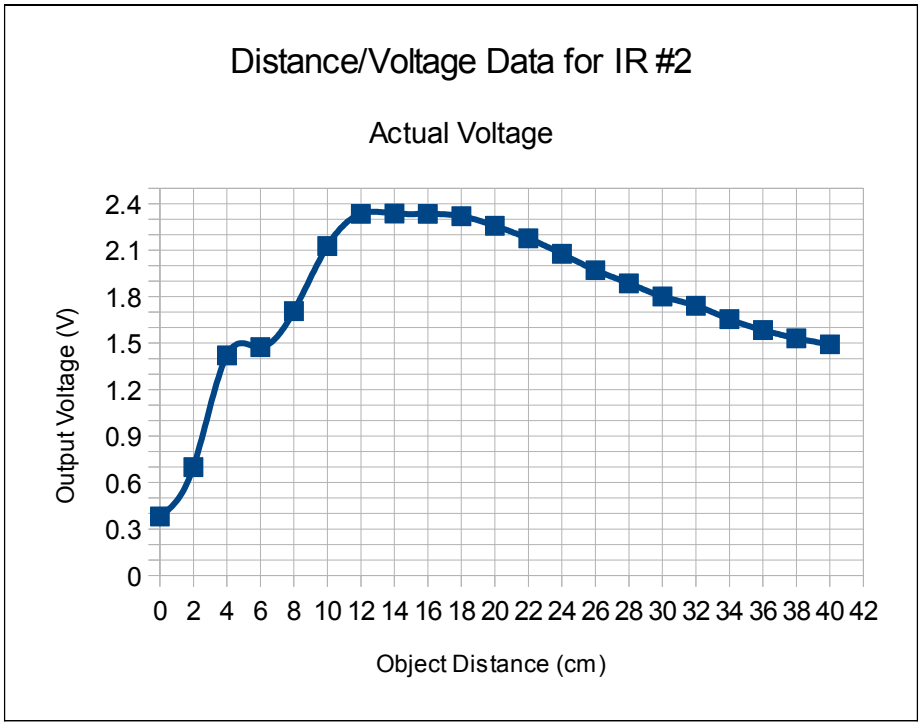


Figure S2: IR #2 Distance/Voltage Data

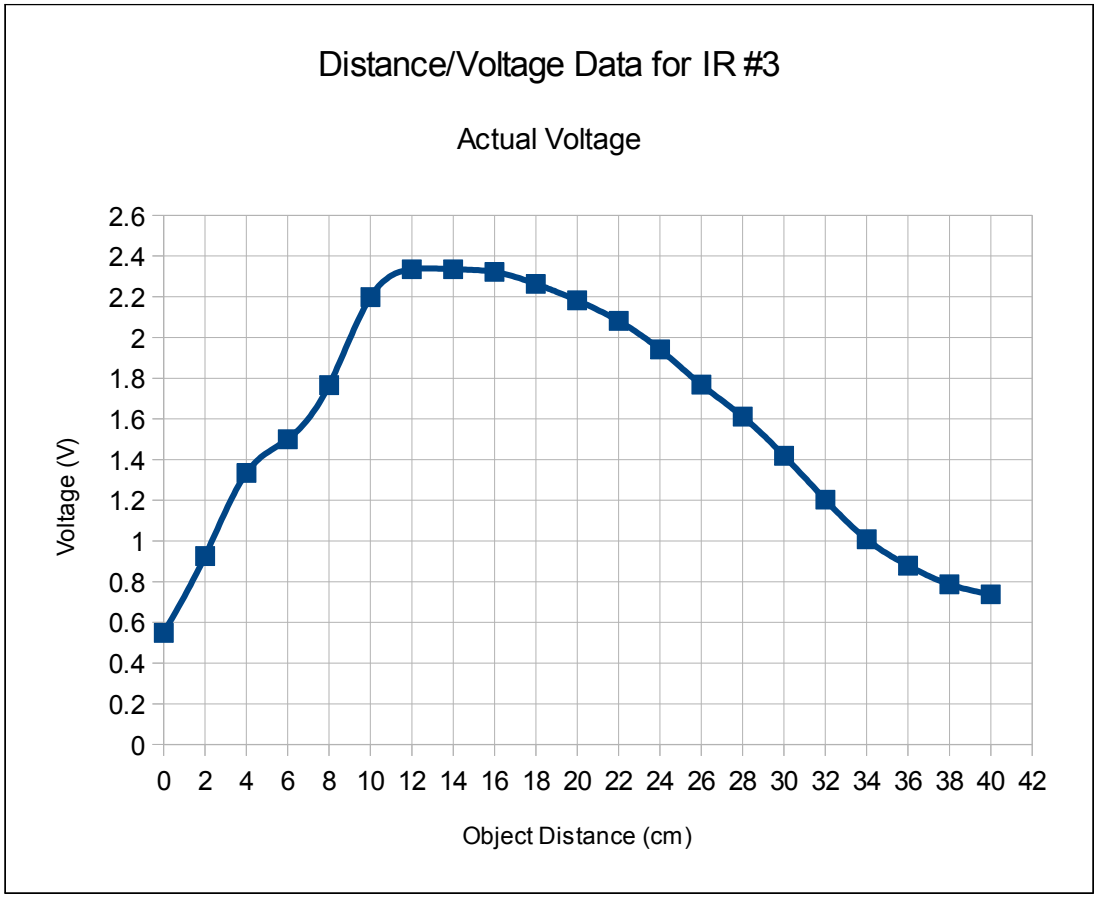


Figure S3: IR #3 Distance/Voltage data

The next sensor test performed was a test of the CdS cell. An average of the voltages found at various light levels for Vcc = 3.3 V is shown in Table S1.

Table S1: Voltage ranges for some levels of light on CdS cells	
Ambient	1.76-1.82 V
With Flashlight	2.55-3.20 V
Medium Dark (Hand over cell)	1.06-1.22 V
Cell completely covered	0.56-0.72 V

Note that when sectioning off the ranges for BuggyBot's reactions to light, the ranges of values not included in the table above were divided between the two measured ranges surrounding them. See Figure S4 for the CdS cell circuit used in the project. The trim pot is used to adjust the CdS cell's sensitivity to light.

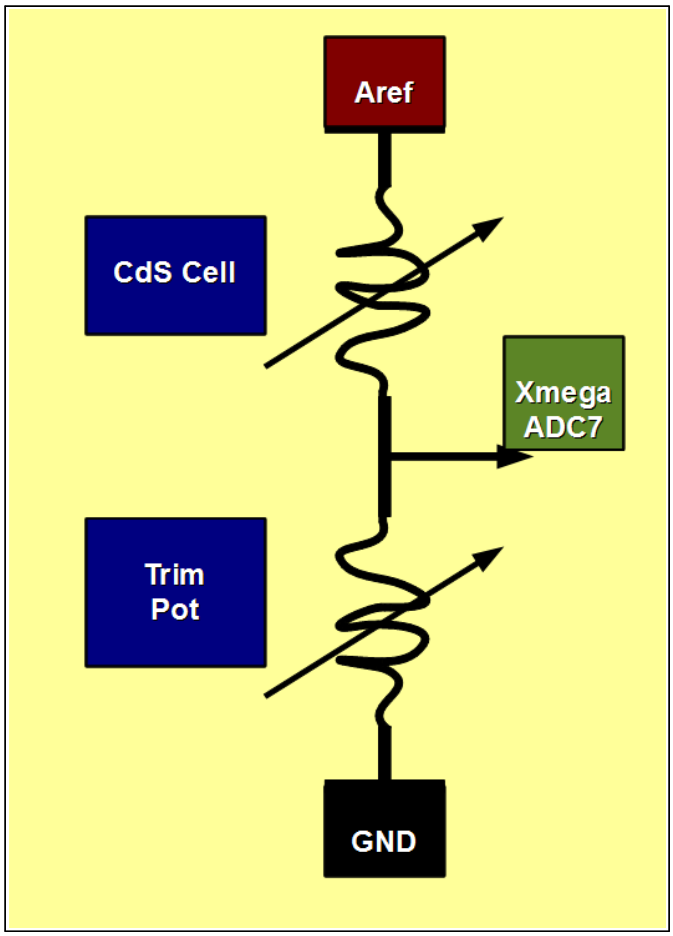


Figure S4: CdS circuit

To read from the photocell, one must first set the ADC channel mux to the correct value to select for the pin the photocell is connected to. Next, it is as simple as fetching the data from the ADC result

register. See Appendix A for the code to do this. Note that the algorithm for reading from the IR sensors is the same with the mux values set differently.

Originally, the special system for this project was going to be a radio transmitter and receiver pair that could tell BuggyBot where its colony is. The transmitter beacon would have been an oscillator with a simple whip antenna attached to make an omnidirectional transmitter. The receiver, then, would have been a directional antenna attached to BuggyBot through a filter. When the receiver could see the transmitted signal, BuggyBot would know where it needed to go to escape danger.
Several different approaches to designing this system were discarded, until finally it was concluded that in the time available, this special sensor just was not feasible. Originally, the circuit in Figure S5 was the transmitter. It is a very simple oscillator, made to transmit a sinewave in the megahertz range to a receiver within several feet. The frequency changes based on the value of R. L is a home made inductor with 15 turns on the short part and 30 turns in the long part. In the circuit actually used, R1 was a 100kohm potentiometer and C1 was 1 microfarad.

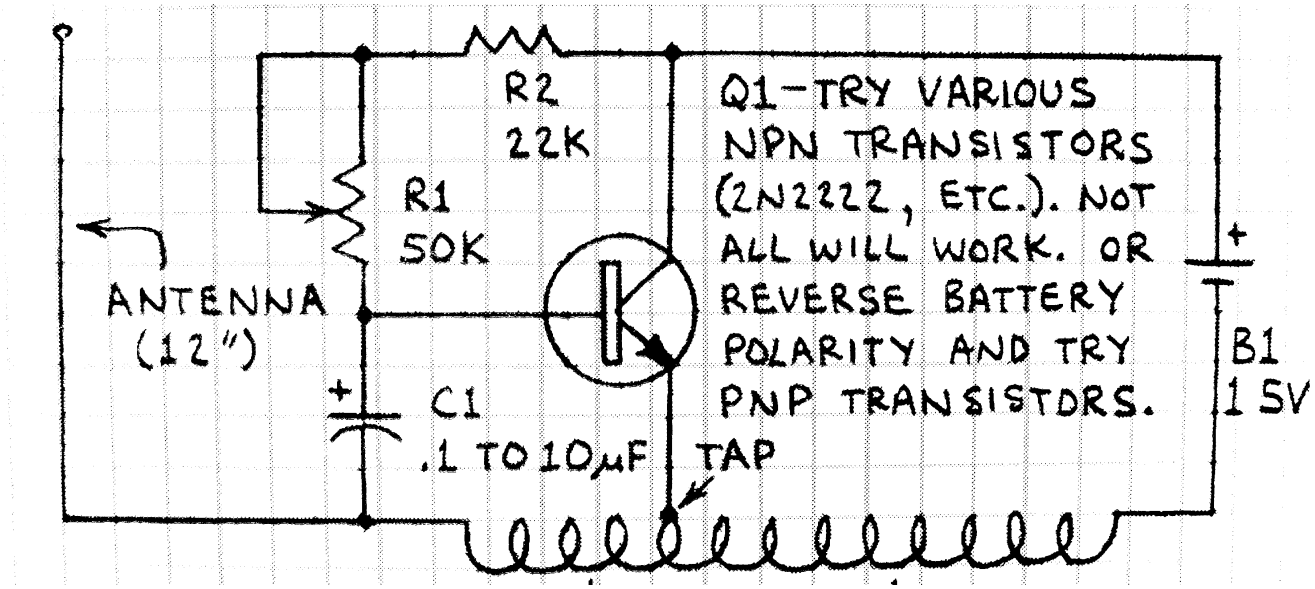


Figure S5: Oscillator circuit. [1]

Figure S6 shows the oscilloscope output R1 = close to 50 kohms, which gave a 252 mV, 7.924 MHz sinewave. Other values of R1 change the frequency and voltage, but it never leaves the MHz range.

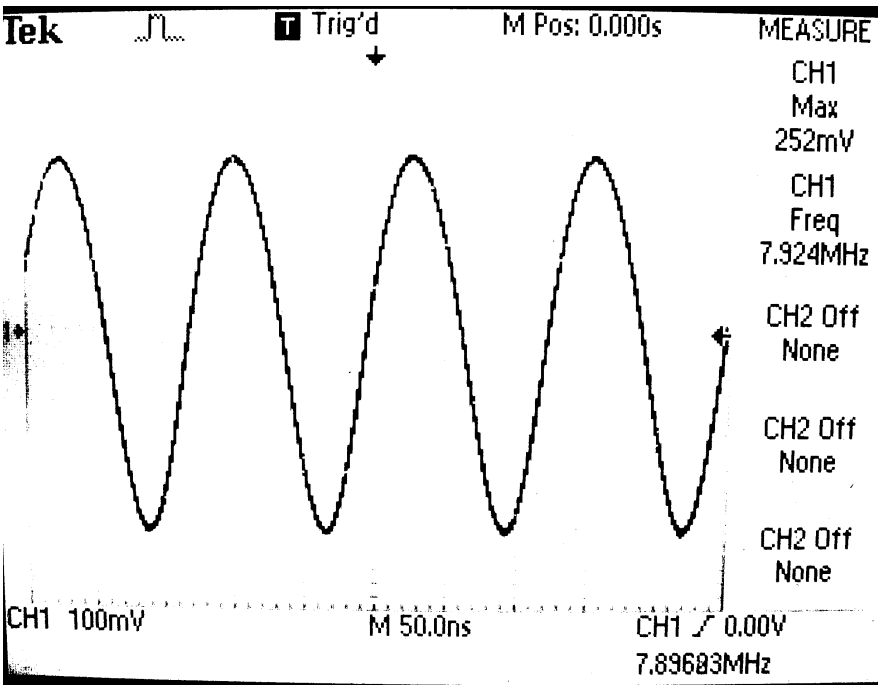


Figure S6: Oscilloscope output for oscillator circuit (R1 = 50 kohms)

The issue with the frequencies coming off of this circuit lies with the antenna that would be mounted to BuggyBot. Originally, BuggyBot was set to have a highly directional receiver antenna known as a Yagi-Uda antenna. After finding a lookup table [2] for various elements of this type of antenna, it was calculated that using the oscillator at the frequencies it was transmitting would require a Yagi antenna larger than the robot platform. Another method involved designing the antenna first so that its dimensions would allow it to fit on BuggyBot, finding the frequency the transmitter would need to send at, and tweaking the oscillator circuit to generate a signal at this frequency. However, the frequency calculated for a Yagi antenna small enough to mount on the robot was close to 100 MHz, which was entirely too high.

Loop antennas were also considered since they are often used in direction finding and low frequency, short range applications [3], and they can be small. However, at this point, it was too late to finish this design.

A compass was chosen for the new special sensor because it would allow BuggyBot's response to danger to stay the same. With the compass, BuggyBot finds its home by locating north and traveling that way until it hits a dark place. The LSM303DLH communicates with the Xmega through a two-wire interface, or I^2C. See Table S2 for calibration information along the compass' x-axis.

Table S2: LSM303DLH Information			
Direction	**X (hex)**	**Y (hex)**	**Z (hex)**
North	880	7fd	670
Northeast	860	8a0	670
East	7b0	900	670

Southeast	6a0	8e0	670
South	650	840	670
Southwest	680	790	670
West	760	700	670
Northwest	810	730	670

Since the x-axis hits its maximum when it is pointing north, the compass was positioned with the positive x-axis pointing to BuggyBot's front. This way, when the robot circles to find north, it will already be facing that direction when it finds it. Note that the values in Table S2 were converted from signed to unsigned values by adding 0x800 for convenience of calculation and display during debugging.

The algorithm for checking the compass heading is simply using an I2C read (with the Xmega as the master and the compass as the slave) six times in a row to get the x, y, and z high and low bytes, then doing some math on the numbers to make them easier to deal with. The I2C read and write algorithms are shown in Figure S6, which came from the LSM303DLH datasheet. See Appendix A for the code to accomplish this.

Table 11. Transfer when master is writing one byte to slave

Master	ST	SAD + W		SUB		DATA		SP
Slave			SAK		SAK		SAK	

Table 12. Transfer when master is writing multiple bytes to slave

Master	ST	SAD + W		SUB		DATA		DATA		SP
Slave			SAK		SAK		SAK		SAK	

Table 13. Transfer when master is receiving (reading) one byte of data from slave

Master	ST	SAD + W		SUB		SR	SAD + R			NMAK	SP
Slave			SAK		SAK			SAK	DATA		

Figure S6: LSM303DLH datasheet's tables showing I2C read and write

The two bump sensors were connected directly to ground on one pin and to an Xmega I/O pin on the other. The internal pullups were activated for the two I/O pins the bump sensors were on, and during obstacle avoidance, the values of these pins were polled. If the bump sensor were activated, the value would go from high to low, and BuggyBot would react accordingly. See Appendix A for the C code.

Behaviors

BuggyBot was programmed in C using the PVR Xmega board. It receives input from the sensors, and following the aforementioned algorithms, it determines where it should go to reach an area with the desired darkness. If it comes too close to an obstacle, the infrared sensors alert it to turn in the appropriate direction. BuggyBot knows when it is dark enough to stop based on input from a CdS cell.

The CdS cell also tells BuggyBot if a light has been directed at it, giving the signal to go away from the light. It then spins in a circle until it finds the north heading, at which point it will pause and go forward in that direction until it loses its heading or reaches the safety of the colony. Should it lose its heading, BuggyBot will stop and look for north again. The darkness level of the colony will tell BuggyBot when it has reached safety. It will pause here until it is reset.

When threatened, BuggyBot's colony-finding behavior takes priority over wandering. The rest of the time, BuggyBot ignores the heading from the compass and meanders about its arena until a light tips it off to danger.

See Figure B1 for the behavior algorithm. C code can be found in Appendix A.

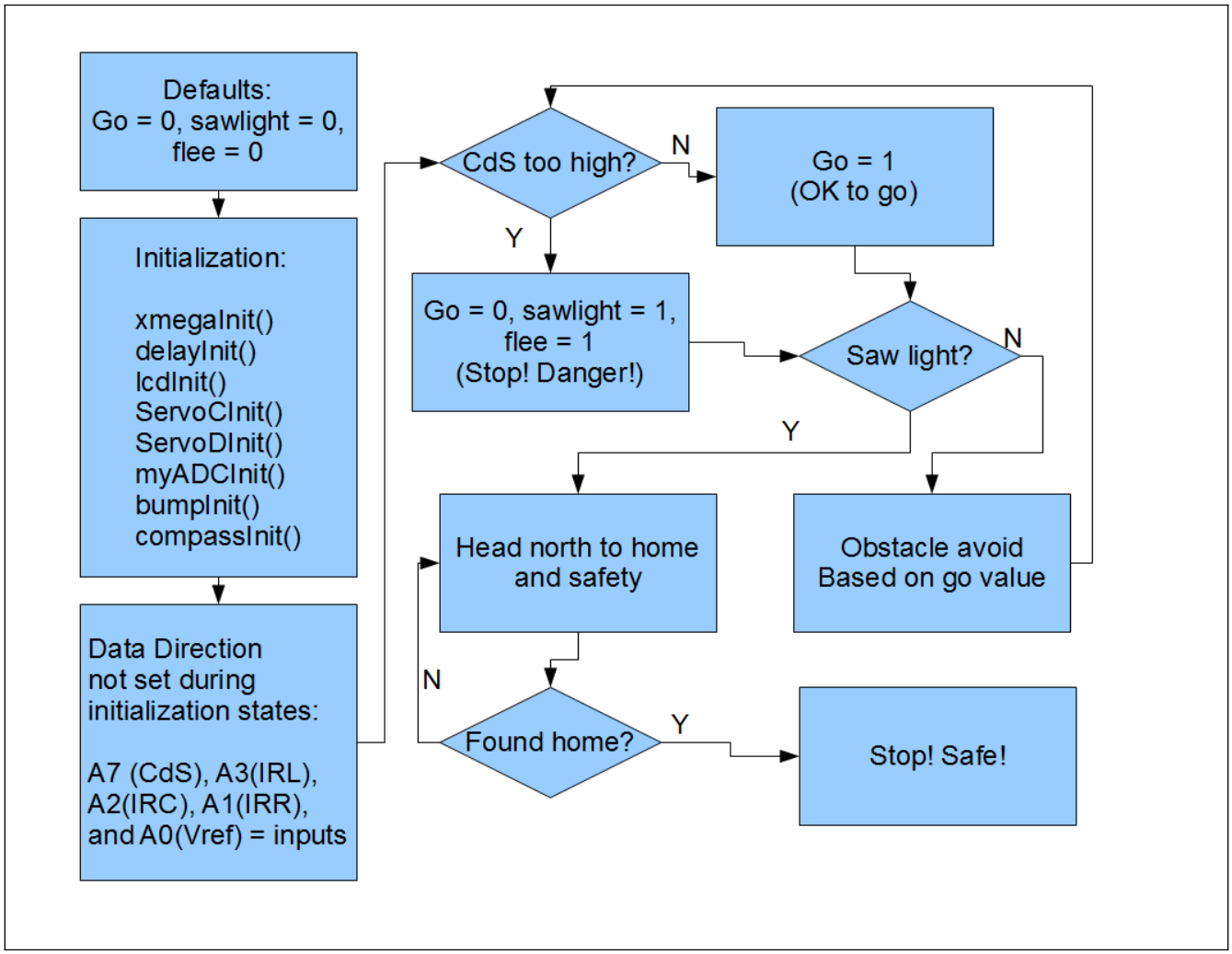


Figure B1: Overall behavior flow chart

Experimental Layout and Results

Most of the experimentation for this project was associated with calibrating the sensors. Since the experimental information related to the sensors has been stated in the "Sensors" section, this section discusses methods used to make sure BuggyBot's systems worked.

The first test involved the hacked servos. The servos were set to stop, and when the PVR board was programmed, a test was done to see if adjusting the potentiometer brought out during the hacking process would help keep the servos from drifting. This worked, and now the potentiometers are checked before any demo.

The first sensor integrated was the CdS cell. The first experiment tested to see if BuggyBot's servos could be stopped as a response to a change in the light hitting the photocell. BuggyBot was placed on a box so that its wheels could spin without the robot going anywhere, and a flashlight was turned on above the photocell. The servos did stop, so next a check was performed to make sure the sensitivity potentiometer on the photocell did its job. This also worked, making CdS cell adjustment another priority before any demo. This allows BuggyBot to perform well in many different levels of lighting.

A check on the bump sensors was performed next. BuggyBot was programmed to not move, but to output to the LCD as a response to a bump. It detected the three possible bumps, so this system was integrated into the behavior.

Finally, the compass was tested. Here, BuggyBot was programmed to only output the headings to the LCD and not move (this was the step used to obtain data in the "Sensors" section). The robot was turned manually to find the values for each direction.

The last test was the final behavior algorithm, which interfaces all of the working systems. Based on these experiments, it seems like a good decision to get all of the pieces of your system working before trying to piece them together. It made debugging things a lot simpler than it would have been otherwise.

Conclusion

This project started out being fairly complicated. Ultimately, limitations on my own capabilities and the time constraints caused the project to end up being much simpler. However, that does not diminish the amount of work accomplished.

Four different kinds of sensors, along with servos, were interfaced with a microcontroller board. A platform was designed in Solidworks and put together, at which point the electronics were mounted. After this was accomplished, the behavioral code fell into place and BuggyBot was finished, albeit in a simpler form than when the semester started. Despite these accomplishments, there were some limitations. For a while, a lot of research time was put into antennas. It turned out that time was a huge limitation to this part of the work. As such, the antenna was dropped in favor of something faster (the compass). Though this didn't end up as part of the design, however, I was still able to make an oscillator circuit, which I count an accomplishment.

While there were limitations, there were also some areas that exceeded expectations. The compass, for

example, proved to be much more accurate than I was anticipating. Having the potentiometers available to adjust on the servos and CdS cell, which were both done as something of an afterthought, became invaluable for keeping BuggyBot behaving as it should. Areas that could be improved lie primarily in the realm of behavior. I really wanted to make BuggyBot much more bug-like than it is.

Although some aspects of BuggyBot did not meet my expectations, I did learn a lot from this project. The biggest thing I learned is that unless you have a lot of experience with analog, do not try to start an antenna project from scratch and finish it in a semester. It is extremely difficult. Another caveat for students to follow would be to make good cables. It was helpful to not have to worry about bad connections. Testing every component used in the final system before putting the final system together is also important. Talking to other students, TA's, and the professors when you are stuck on something is a good idea as well. Sometimes other people have a new perspective on an issue that you hadn't even thought about.

If I could start the project over, I would not try to make an antenna. I would go with something simpler like a compass for safe zone discovery from the start, which would allow me to add more sensors and make the behaviors more complex. This would allow me to enhance BuggyBot's insect behaviors and make it more bug-like.

Documentation

(Note: datasheets not directly used for this paper are included here because without them, the paper would not be possible)

[1] *Science and Communication Circuits & Projects,* P. 134, F. M. Mims, III.

[2] http://www.antenna-theory.com/antennas/travelling/yagi3.php

[3] http://en.wikipedia.org/wiki/Loop_antenna#cite_note-0

[4] *ATXmega128 datasheet* (Available at: http://www.atmel.com/dyn/products/product_docs.asp?category_id=163&family_id=607&subfamily_id=1965&part_id=4298)

[5] *ATXmega128 user manual* (Available at: http://www.atmel.com/dyn/products/product_docs.asp?category_id=163&family_id=607&subfamily_id=1965&part_id=4298)

[6] *LCD datasheet (*Available at: http://www.sparkfun.com/products/9054)

[7] *LSM303DLH datasheet* (Available at: http://www.pololu.com/catalog/product/1250)

[8] *LSM303DLH application note* (Available at: http://www.pololu.com/catalog/product/1250)

[9] *Sharp GP2D120XJ00F datasheet* (Available at: http://www.sparkfun.com/products/8959)

[10] *PVR board manual.* M. Pridgen and T. Vermeer.

[11] *PVR Sample Code.* M. Pridgen and T. Vermeer.

[12] *Sharp IR Sensor Comparison.* (Available at: http://www.mil.ufl.edu/imdl/handouts.htm)

[13] Sample I2C code from Kris Brosch

Appendices

Appendix A: C Code

1. behavior_system.c

```
#include "PVR.h"
#include "jdg_header.h"
#include "compass.h"
#include <avr/io.h>
#include <stdlib.h>
#include <stdio.h>

int main(){
  int go = 0;
  int sawlight = 0;
  int flee=0;
  xmegaInit();
  delayInit();
  lcdInit();
  ServoCInit();
  ServoDInit();
  myADCinit();

  PORTA_DIR &= 0xE0; //Set A7(CDS), 3(IRL), 2(IRC), 1(IRR),
0(Vref) to input

  bumpInit();
  CompassInit();
```

(behavior_system.c continued)

```
  while(1){
    lcdGoto(0,0);
    lcdChar((go == 0) ? '0' : '1');
    //Freeze in the light at first
    if(CDScheck() > (3000)){
      go = 0;
      sawlight = 1;
      flee = 1;
    }
    else{
      go = 1;
    }

    if(sawlight == 0) {
      obstacle_avoid(go);
    }
    else {
      goNorth(go);
    }

    if((flee == 1) && (CDScheck() < 500)){
      while(1){
     ServoC0(0);
     ServoD0(0);
     lcdGoto(1,0);
              //0123456789abcdef
     lcdString("Safe!            ");
     flee = 0;
      }
    }

  }
}
```

2. jdg_header.c

```
//C file with functions I've written for BuggyBot. Some
//code borrowed from PVR.c

#include <avr/io.h>
#include <avr/interrupt.h>
#include "PVR.h"
#include "jdg_header.h"
#include "compass.h"

void ADCToString(int x){

  unsigned char num[5] = {'0','0','0','0','\0'};
  unsigned long a;
  int i;

  a = (unsigned long)x;
  for (i = 3; i >= 0; i--){
     num[i] = ((char)((int)(a % 10))) + 0x30;
     a = a/10;
  }

  //lcdString(num);
}

int IRcheck(int ir){
  int adc;
  if(ir == 1)
    ADCA_CH0_MUXCTRL = 0x08;
  else if (ir == 2)
    ADCA_CH0_MUXCTRL = 0x10;
  else
    ADCA_CH0_MUXCTRL = 0x18;

  ADCA_CTRLA |= 0x04; //start conversions...
  while((ADCA_CH0_INTFLAGS & 0x01) != 0x01);
  ADCA_CH0_INTFLAGS |= 0x01; //clear conv. flag

  adc = ADCA_CH0_RES;
  ADCToString(adc);
  //lcdString("  ");
  return adc;
}
```

(jdg_header.c continued)

```
int CDScheck(){
  int adc;
  ADCA_CH0_MUXCTRL = 0x38;
  ADCA_CTRLA |= 0x04;
  while((ADCA_CH0_INTFLAGS & 0x01) != 0x01);
  ADCA_CH0_INTFLAGS |= 0x01;
  adc = ADCA_CH0_RES;
  return adc;
}

void myADCinit(){
  ADCA_CTRLB = 0x00; //12-bit, right adjusted
  //ADCA_REFCTRL = 0x10; //Vref = Vcc/1.6
  ADCA_REFCTRL = 0x20; //Vref = Aref = 3.3V
  ADCA_CH0_INTCTRL = 0x00; //set flag @ conv. complete
  ADCA_CH0_CTRL = 0x01; //single ended
  ADCA_PRESCALER = 0x03; //divide analog clk by 32...?
  ADCA_CTRLA |= 0x01; //enable
}

int obstacle_avoid (int go){
  unsigned int IR1, IR2, IR3, IR1_val, IR2_val, IR3_val, straight,
turn, force_right, force_left, bump, force_back;

  ServoC0(-50);
  ServoD0(50);

//  turn = (rand() & 0x7f);
  straight = 0;
  force_left = 0;
  force_right = 0;
  force_back = 0;
  IR2 = 0;
  IR3 = 0;
  if(go == 1){
   // lcdGoto(1,0);
    //lcdInt(turn);
      //lcdString("   ");
    IR1_val = IRcheck(1); //right
    IR2_val = IRcheck(2); //center
    IR3_val = IRcheck(3); //left
    bump = bumpCheck();
```

(jdg_header.c continued)

```
if(IR1_val > 2000)
  IR1 = 1;
else
  IR1 = 0;
if (IR2_val > 2000)
    IR2 = 1;
else
    IR2 = 0;
if(IR3_val > 2000)
    IR3 = 1;
else
    IR3 = 0;

if(bump == bump_left) {
  ServoC0(50);
  ServoD0(-80);
}

else if(bump == bump_right){
  ServoC0(80);
  ServoD0(-50);
}

else if(bump == bump_center){
  ServoC0(50);
  ServoD0(-50);
  force_back  = 2;
}

if (force_back > 0){
  ServoC0(50);
  ServoD0(-50);
  force_back--;
}
else if(force_right > 0) {
    ServoC0(-100);
    ServoD0(-10);
    force_right--;
}
else if(force_left > 0){
  ServoC0(10);
  ServoD0(100);
  force_left--;
}
```

(jdg_header.c continued)

```
    else if(IR1){
      ServoC0(-100);
      ServoD0(-10);
      force_right= 1;
    }

    else if(IR3){
      ServoC0(10);
      ServoD0(100);
      force_left = 1;
    }

    else if(IR2){
      ServoC0(50);
      ServoD0(-50);
    if(IR1_val > IR3_val){
      force_left =2;
    }


    else{
      force_right = 2;
    }
   }

   else{

    ServoC0(-50);
    ServoD0(50);
   }

  }

  else {
    ServoC0(0);
    ServoD0(0);
  }

  if((force_left > 0) || (force_right > 0)) {
    return 1;
  }
  else {
    return 0;
  }
}
```

(jdg_header.c continued)

```
void goNorth(int go) {
  int avoiding = 0;
  int avoidcount = 0;
  if(go == 1) {
    if((compassCheck() == 1) || (avoidcount > 0)) {
      avoiding = obstacle_avoid(go);
      if(avoiding != 0) {
        avoidcount = 10;
      }
      if(avoidcount > 0) {
        avoidcount--;
      }
    }
    else {
      ServoC0(10);
      ServoD0(10);
    }
  }
  else {
    ServoC0(0);
    ServoD0(0);
  }
}

void bumpInit(){
  PORTJ_DIR &= 0xF8; //J(2 downto 0) <= input
  PORTJ_PIN0CTRL  = 0x58;  //set internal pull ups
  PORTJ_PIN1CTRL  = 0x58;  //and inverters
  PORTJ_PIN2CTRL  = 0x58;
  return;
}

int bumpCheck(){
  int bump;
  bump = (PORTJ_IN & 0x07); //fetch bump sensor vals
  return bump;
}
```

(jdg_header.c continued)

```
int compassCheck(){
  uint8_t x_h, x_l, y_h, y_l, z_h, z_l, r, dbg;
  int16_t x,y,z;
  char headings[17];
  int north_found = 0;
  lcdGoto(1,0);
           //0123456789abcde
  lcdString("Finding north...");

  x_h = compassRead(comp_XH);
  x_l = compassRead(comp_XL);
  y_h = compassRead(comp_YH);
  y_l = compassRead(comp_YL);
  z_h = compassRead(comp_ZH);
  z_l = compassRead(comp_ZL);
  x = (((uint16_t)x_h << 8) | x_l) + 0x800;
  y = (((uint16_t)y_h << 8) | y_l) + 0x800;
  z = (((uint16_t)z_h << 8) | z_l) + 0x800;

  if(x > 0x786){
    north_found = 1;
             //0123456789abcdef
    lcdGoto(1,0);
    lcdString("North found!     ");

  }

  return north_found;
}
```

3. jdg_header.h

```
#ifndef  __jdg_header_h__
#define  __jdg_header_h__

#include <avr/io.h>
#include <avr/interrupt.h>
#include "jdg_header.h"
#include "PVR.h"

#define bump_left 4
#define bump_right 2
#define bump_center 6
#define bump_behind 1
#define no_bump 0

void ADCToString(int x);
int IRcheck(int ir);
int CDScheck();
void myADCinit();
int obstacle_avoid();
void bumpInit();
int bumpCheck();
int compassCheck();

#endif
```

4. compass.c

```
//Compass function file. Based in part on code from Kris Brosch and
//sample code from pololu.com

#include "compass.h"
#include <avr/io.h>
#include "PVR.h"

void compassInit(){
  //xmega twi init setup
  TWIF_MASTER_BAUD = 155; //f_twi = 100kHz
  TWIF_MASTER_CTRLA |= 0x08; //enable TWIF as master
  TWIF_MASTER_STATUS = 0x01; //set bus state to idle

  //compass twi setup
  compassWrite(0x00, 0x14); //min. data output rate = 30 Hz
  compassWrite(0x01, 0x20); //gain = 1.3
  compassWrite(0x02,0x00); //write 0x00 to MR_REG_M for cont. conv. mode
  return;

}

//send one byte of data
void compassWrite(uint8_t address, uint8_t data){
  TWIF_MASTER_ADDR = write_addr; //tell device we want to write to it
  while (WIF == 0); //wait for device to ack, transmission to finish
  TWIF_MASTER_DATA = address; //send address where we want to write to
  while (WIF == 0); //wait for device to ack, transmission to finish
  TWIF_MASTER_DATA = data; //send data to write to address
  while (WIF == 0); //wait for device to ack, transmission to finish
  TWIF_MASTER_CTRLC |= 0x03; //send stop condition...
}

//read one byte of data...DON'T FORGET TO INC THE ADDRESS EVERY PASS!
//(only for this device)
uint8_t compassRead(uint8_t address){
  uint8_t compass_data;
  TWIF_MASTER_ADDR = write_addr; //tell device we're writing to it
  while(WIF == 0); //wait for device to ack
  TWIF_MASTER_DATA = address; //tell device we want to read from output
  while(WIF==0); //wait for device to ack
  TWIF_MASTER_ADDR = read_addr; //send read addr to device
  while((WIF == 0) && (RIF == 0)); //wait for ack
  compass_data = TWIF_MASTER_DATA; //fetch data
  TWIF_MASTER_CTRLC |= 0x04; //send nack
  TWIF_MASTER_CTRLC |= 0x03; //send stop
  return compass_data;
}
```

5. compass.h

```
#ifndef __compass_h__
#define __compass_h__

#include <avr/io.h>
#include "jdg_header.h"
#include "PVR.h"

#define write_addr 0x3C
#define read_addr 0x3D
#define MR_REG_M 0x02
#define comp_XH 0x03
#define comp_XL 0x04
#define comp_YH 0x05
#define comp_YL 0x06
#define comp_ZH 0x07
#define comp_ZL 0x08
#define RIF (TWIF_MASTER_STATUS & 0x80)
#define WIF (TWIF_MASTER_STATUS & 0x40)

void compassInit();
void compassWrite(uint8_t address, uint8_t data);
uint8_t compassRead(uint8_t address);

#endif
```

6. Other code files used but not included here due to length

- PVR.c (included with PVR board)
- PVR.h (included with PVR board)
- iox128a1.h (C header file from Atmel)

Appendix B: Parts Description

- LCD: LCD-09054, Sparkfun, $14.95

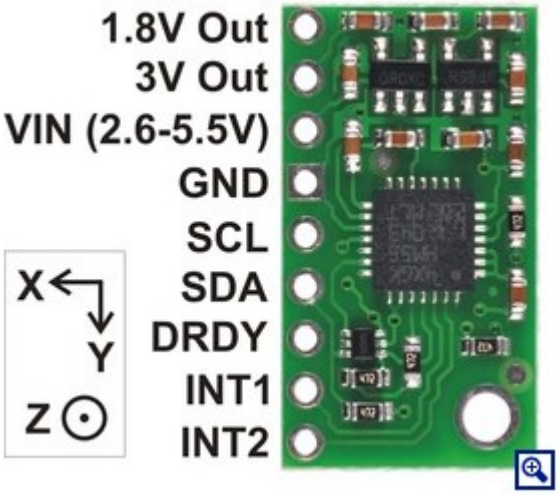
- Servos (2) : Hitec HS-311 Standard, ServoCity, $7.99

- IR Sensors (3): Sharp GP2D120XJ00F, Sparkfun, $13.95

- CdS Cell: 276-1657, Radioshack, $.60

- Bump switches (2): KSM06330 , Jameco, $.19

- Compass: LSM303DLH, Pololu, $29.95



- PVR Board: $79.69
- Wheels (pair): 4.00ACR, ServoCity, $3.00
- Servo hubs (2): SH503H, ServoCity, $9.99
- Screws and nuts (many) : Lowes
- 1/8" wood: provided in lab