"Ziggy"

The Autonomous Robot

Ed Carstens

EEL 5934
Professor Keith Doty
May 1, 1995

Table of Contents

Abstract

Ziggy is a mobile robot which consists of a variety of behaviors in order to carry out a specific goal. That goal is to serve as a moving vending machine -- one that only distributes items. Its behaviors are straight line motion, playing music, and sound-following. The purpose of the music is to attract people to Ziggy just like an ice cream truck attracts children with its music. The straight line and object avoidance behaviors help Ziggy navigate through passageways and avoid collisions with objects. Sound-following helps Ziggy locate people.

Executive Summary

Ziggy consists of a plywood platform, wheels, motors, and sensors -- all of which tie in to the Motorola 6811, the brains of the robot. Two reversible dc motors are mounted to the platform via a custom-designed metal bracket. The motors are highly efficient in terms of the current drawn and output torque. The emitter-detectors are taped to the metal bracket. A hole in the bracket serves to block ambient light from the detector, which peeks out through the hole from inside. A 64-segment shaft encoder is supposed to tell Ziggy how much each wheel turns as it moves. (This has not yet worked successfully.) An algorithm can then keep Ziggy going in a straight line. Cellophane (Scotch) tape holds 32 thin white strips to the shaft encoder wheel.

Ziggy has two IR sensors in front positioned cross-eyed and one IR sensor on each side. A 390 chip used as a divide-by-50 on the E-clock generates a 40 kHz square wave propogated through four LED's -- one for each sensor. Most collision avoidance is done by the two front IR sensors. The side sensors let Ziggy know when a wheel is pinned against an object. They also put him into and out of test mode by triggering both sides at once. In test mode, his motors are disabled and music is stopped by the software, but all other processes continue as if nothing happened.

Ziggy has a right and a left microphone allowing him to hear loud noises such as claps. The sound registers as a tiny fluctuation in voltage at the output of the piezoelectric sound sensor. It runs to an LM386 op amp which amplifies the signal enough to sense loud noises as small fluctuations about a dc level. This voltage is compared with a threshold set manually by a potentiometer. Two flip flops

hold a trigger and a direction bit. If either microphone hears something, the first flip flop is set and clocks the second flip flop which is set or cleared depending on which microphone heard the noise first.

Introduction

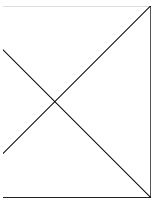
Autonomous mobile robots are not very smart. They have limited senses and do not comprehending anything. They are mobile computers, carrying out programs written by their creators, emulating only the simplest of behaviors. How is it then that they seem to exhibit some kind of basic intelligence, leading us to anthropomorphize (i.e. name) them? Jones and Flynn's book, "Mobile Robots", discusses one such autonomous robot called "Rug Warrior" which seems to have its own personality. It is capable of straight-line motion, wall-following, light following, person-following, and hiding. This book is a good starting point for anyone who wants to build their own mobile robot.

Ziggy's goal in life is simply to dispense as many goodies to as many people as possible. Ziggy, the moving vending machine, will have a variety of behaviors designed to produce the overall goal of finding people or having people find Ziggy. To this end, Ziggy must be capable of following people, hearing sounds, and playing music (to get people's attention). It will also need a collision avoidance algorithm and a wall-following behavior (in case it gets lost).

System Components

The following is a list of components from which Ziggy was created.

- plywood platform
- two black rubber 3" diameter model airplane wheels
- caster wheel mounted to block of wood glued to platform
- Motorola (M68HC11EVBU) board
- power bus consisting of a 5V voltage regulator (7805A), 470 uF capacitor, 1k ohm resistor, LED, and two AA battery 4-packs



re : Power Bus

- 32K RAM chip (HY62256A)
- motor driver chip (L293NE)
- 3-input nand gate chip (CD74HCT10EX)
- latch (74HC573N)
- hex inverter chip (CD4049UBE)
- two SFH900 Siemens sensors and sensor circuits, each consisting of 290 and 33k ohm resistors
- two cylindrical variable dc motors
- motor mount assembly including screws and hose clamps
- two toggle switches (for power and chip mode)
- four analog IR sensors
- IR emitter circuit consisting of four LED's in series with 270 ohm resistor, MC74HC390N chip
- PKM44EW-10 piezoelectric buzzer
- two op amp circuits each consisting of a QMB-01 STAR 901 piezoelectric sound sensor; an LM386 op amp; 1000 uF-, 10 uF-, and 0.001 uF capacitors; and a 2.2k resistor

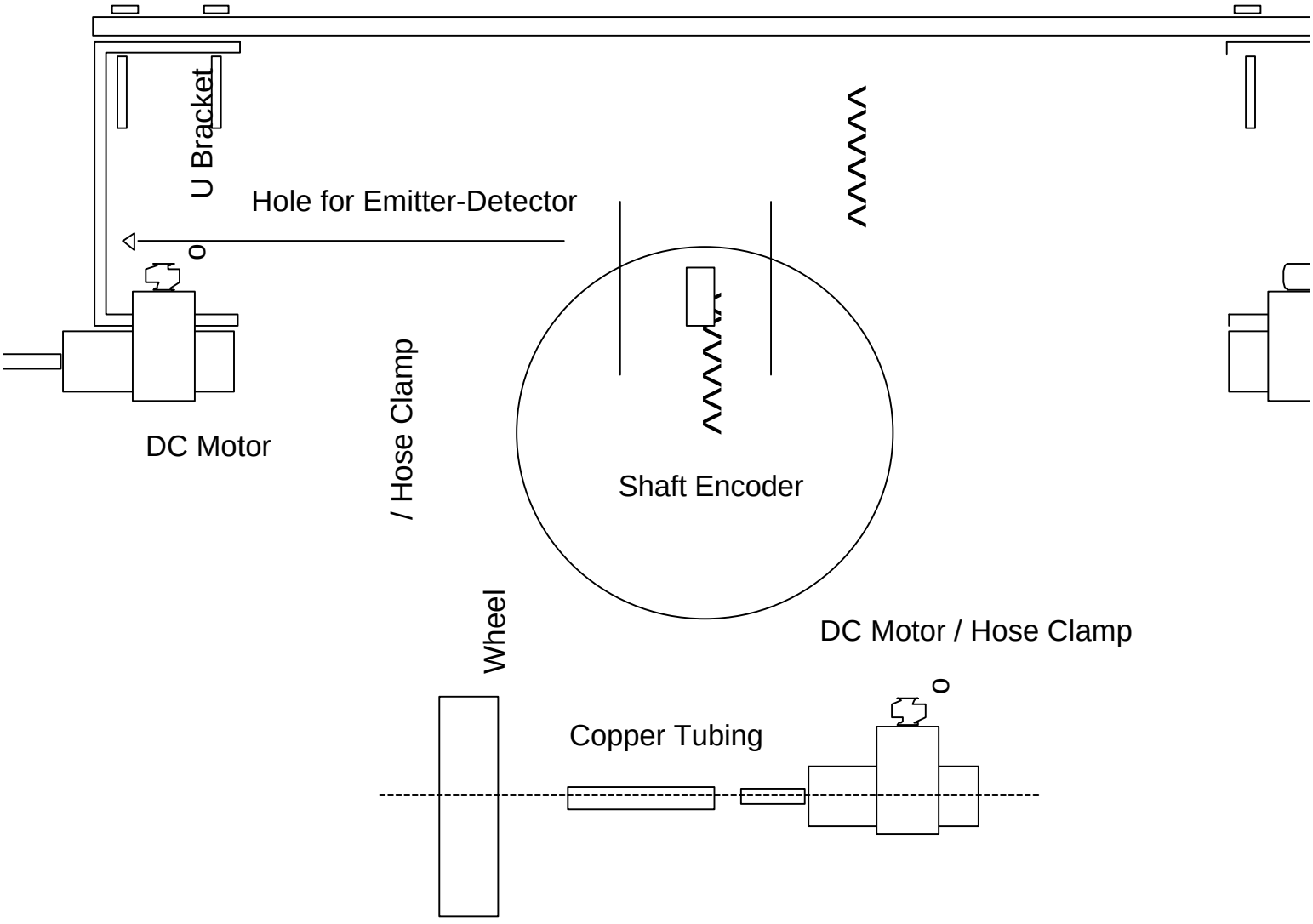
- one sound direction capture circuit consisting of two 10k potentiometers, a 339 comparator, a 74LS32 chip (4 OR gates), and a 74S112 dual negative edge triggered JK-flip flop.

Mobile Platform

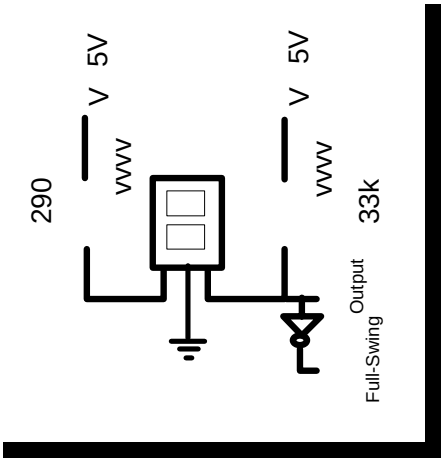
The robot platform, built from inexpensive yet sturdy plywood, takes a fair amount of abuse so sturdiness is advantageous. While lego robots have the advantage of easy modification, they tend to break apart more easily. The base consists of two thin 12"x8" boards glued together for a total thickness of 1/4 inch. The rear corners of the platform are cut off, permitting Ziggy to more easily avoid getting caught against walls as it swivels. The rear caster wheel fastens to a block of wood glued to the platform with Zap-a-Gap. The block of wood, 1.5 inches in height, keeps the platform level. Two switches are present on the platform. The right rocker switch turns power on and off and the left one toggles between bootstrap and single-chip mode.

Motor and Wheel Assembly

The motors are a couple of Swiss variable 90 rpm dc motors from American Science and Surplus. These motors are extremely efficient, as they produce much torque and draw little power even when the motor reverses directions. The motors are 5/8" dia. x 1-3/4" long body. Two custom-made metal U brackets (2"x2.25"x2") attach the motors to my platform. The motor-mounting assembly is shown in Figure 2. A hose clamp holds each motor. The model airplane-type wheels are three inches in diameter and each is glued to a piece of copper tubing glued to the motor shaft.



re : Motor mount assembly and Shaft Encoder Hole



re : Shaft Encoder Circuit

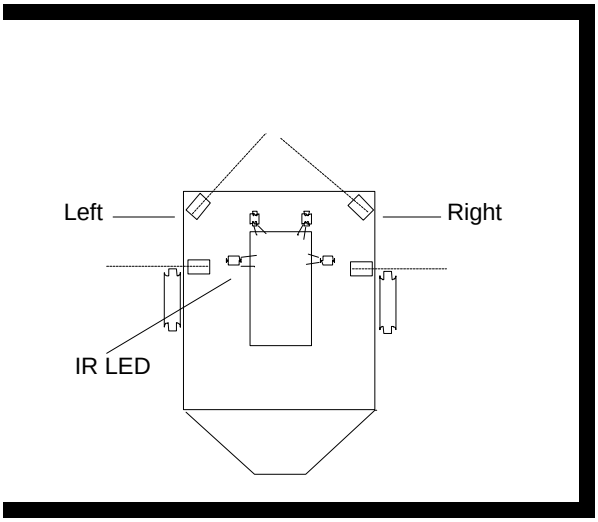
Shaft Encoders

The shaft encoder circuit never worked reliably so it is not presently incorporated into Ziggy. This circuit finds out how much each wheel is turning by using a shaft encoder wheel. For each side, a rectangular hole was drilled and filed and a Siemens SFH900 emitter-detector was positioned right up next to and looking out through the hole. The direct output voltage ranges from about 0.2 V to 3 V. This signal then runs through an inverter to produce a clean digital waveform (0 or 5V) and is wired to the pulse accumulator on the Motorola M68HC11EVBU board. It may be necessary to add a capacitor to eliminate spurious output signals.

A 64-segment shaft encoder wheel is used. Originally, the paper template was pasted to some cardboard, but the black portions of paper did not absorb the light as expected. It reflected about as well as white paper. With narrow (1/16" wide) strips of white paper glued directly to the black rubber tires, the shaft encoder circuit catches nearly all of the 64 segments. The reason for the missed segments is probably that the strips of paper are spaced too close to one another in conjunction with the fact that the curvature of the tires tends to admit ambient light to the sensors. Yet another problem develops when Ziggy moves about. The strips of paper eventually start falling off the tires. With the same narrow strips glued to a flat black plastic top from a tennis ball container, less ambient light is admitted to the sensors, enhancing performance. However, Zap-a-gap or Elmers glue fails to hold the paper strips permanently to the smooth plastic. The strips are now held on permanently

with Scotch tape, but this reduced performance of the shaft encoder circuit since the tape is sometimes a little bit reflective.

Infrared Proximity Sensors

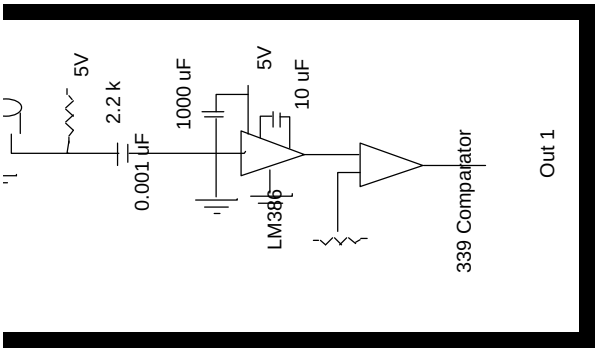


re : Aerial View of Ziggy

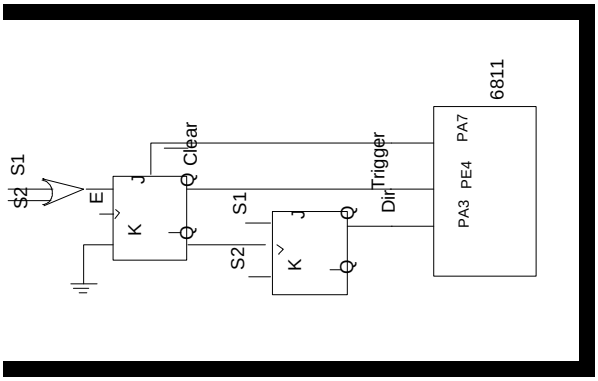
A "hack" described in class for modifying the digital Sharp IR sensors to produce an analog output is employed. The sensors were successfully tested on a protoboard before adding them to Ziggy. Four IR-emitting LED's wired in series with a 270-ohm resistor reside on the 6811 board as shown in Figure 4. The two cross-eyed sensors in front are used for nearly all obstacle avoidance. The cross-eyed configuration is recommended (over straight-line sight) because it enhances general performance. The side sensors also aid in obstacle avoidance occasionally. Notice each side sensor is positioned just in front of each wheel. When Ziggy's wheel gets pinned against something (as it often does), the side sensor lets Ziggy know which wheel is pinned so Ziggy can move clear of the obstacle.

Sound Sensors and Related Circuits

Ziggy has two "ears" for hearing and detecting the direction of the sound. The right microphone faces out to hear noises from Ziggy's right side. Similarly, the left one hears noises from Ziggy's left side. Sound wave vibrations trigger tiny voltage fluctuations at the output of each microphone. Th LM386 op amp amplifies this. For each ear, the amplified waveform is fed to a 339 comparator. If the waveform rises above the threshold manually set by the potentiometer, the 339 comparator output goes high. Otherwise, it remains low. The two outputs from the two 339 comparators are OR'ed together and fed to the J input of the first JK flip flop, which stores the sound trigger. The second JK flip flop, clocked by the sound trigger, reads the two 339 comparator outputs into the J and K inputs. This sets or resets the direction bit. In this, way Ziggy can tell where the sound is coming from by which sensor detected the sound first.



re : Sound Amplification and Comparison

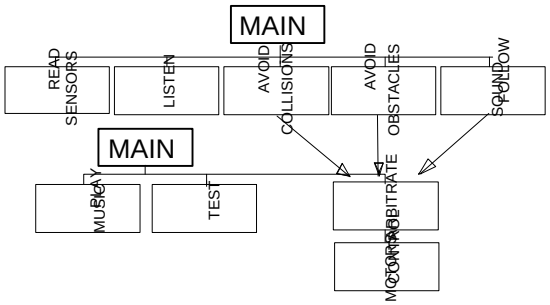


re : Directional Stage

Behaviors

Ziggy has four main behaviors. They are straight-line motion, obstacle avoidance, sound-following, and playing music. If the IR proximity sensors do not detect anything and there are no loud sounds to trigger the sound-following behavior, Ziggy continues in a straight line. This is the lowest priority behavior. When Ziggy encounters an obstacle it may turn to

avoid it, or if it is closer it may spin right or left to avoid it. When Ziggy hears something, it goes into its sound-following behavior. The architecture is not truly subsumption because all behaviors, once started, run perpetually. The arbitration process, which alone controls the motors, decides which behavior to obey. Each behavior sends its recommendation for right and left motor speeds along with a priority to the arbitration routine (through global arrays). The highest priority behavior is always active. A selection of tunes continually plays with 30 second pauses between each selection. Having other behaviors themselves trigger different tunes is fun too, and especially useful for debugging. However, it does get tiring listening to the same tune for very long. Ziggy plays four tunes: "Charge!", "Deep Space Nine Theme ", "Pink Panther", and a classical number. Ziggy used to play the theme for final jeopardy when it got trapped. The piezoelectric speaker connects to PA3 on the 6811. Interactive C comes with several tunes and routines to play them.

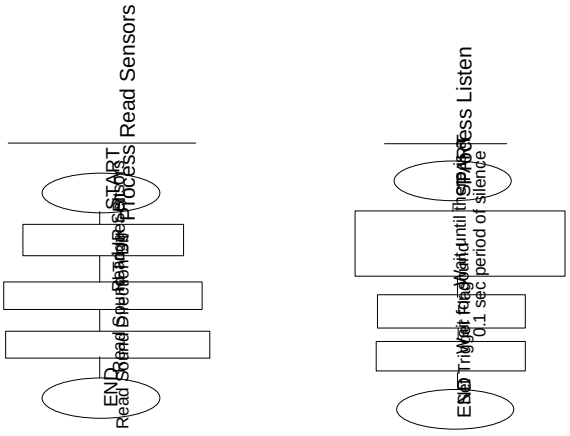


re : Structure Chart

Structure Chart

The main module initializes eight processes: Read Sensors, Listen, Avoid Collisions, Avoid Obstacles, Follow Sound, Play Music, Test, and Arbitrate. The arrows pointing from the Avoid Collisions, Avoid Obstacles, and Follow Sound processes show that they are each making recommendations of motor speeds and sending priorities to the Arbitrate process. The Arbitrate process simply executes the motor speed commands of highest priority by calling the Control Motors procedure. Test mode is a special case where the motors stop and are inhibited within the Control

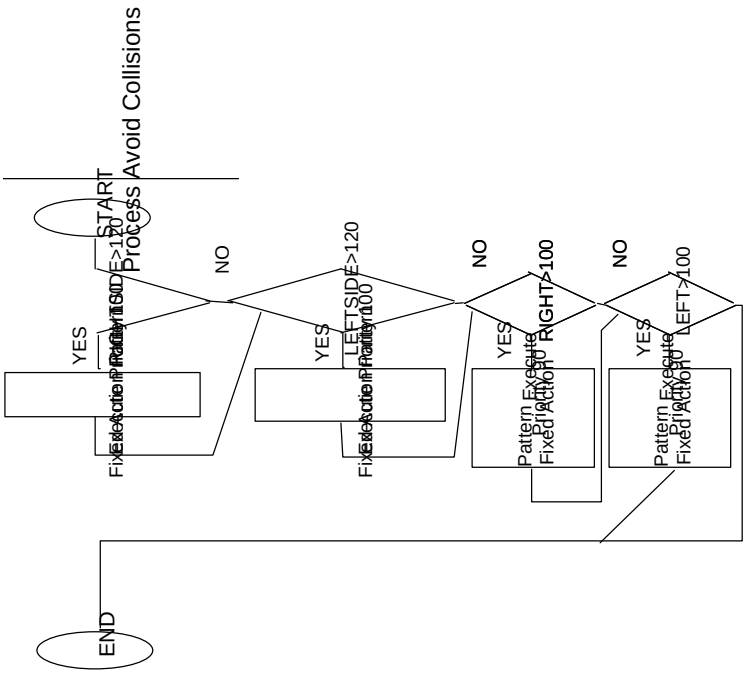
Motors procedure. Further music selections also abate. Everything else runs as if nothing happened. Test mode activates when both side sensors read greater than 120 and deactivates the same way.



re : Read Sensor and Listen Flowcharts

Sensor Algorithms

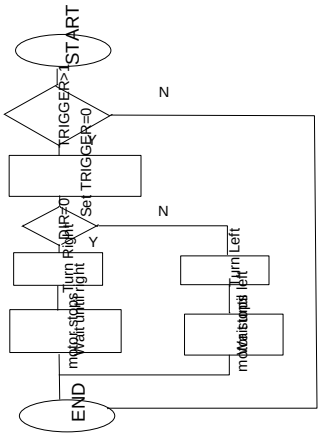
One process reads all sensors, storing them in global variables. The Listen process takes care of reading a sound, getting a direction and setting a global 'trigger' flag to activate the sound following behavior. The direction returned from the flip flop would be dubious if the first flip flop were cleared while a sound was occurring. To avoid this, the Listen process waits until there is a 0.1 second period of silence. Then it waits for and processes the next sound it hears.



re : Collision Avoidance Flowchart

Avoidance Algorithms

The highest priority behavior is found in the process, Avoid Collisions. This algorithm first checks to see if one wheel is pinned against an object. If so, it executes a level 100 priority fixed action pattern. Suppose the right wheel is pinned against a wall. Presumably, RIGHTSIDE will be greater than 120. Ziggy first tries to run the right motor forward for a certain amount of time. Then he stops the right one and moves the left forward for the same amount of time. At this point Ziggy checks RIGHTSIDE again to see if he is clear of the wall. If not, he runs the right motor in reverse, followed by the left, and exits the fixed action pattern. The two front sensors (RIGHT and LEFT) are compared with 100. If the right front sensor (RIGHT) is greater than 100, Ziggy is told to run the right motor in reverse. Likewise, if the LEFT is greater than 100, the left motor runs in reverse. The obstacle avoidance algorithm is similar, but is of lower priority because obstacles are more distant so a collision is not imminent. Ziggy does not spin in place, but turns more gradually at this distance.



re : Sound Following Flowchart

Sound Following Algorithm

The sound following algorithm is very simple. It is the Listen process that bears the responsibility of determining a valid direction for a given sound. If TRIGGER is 2 then TRIGGER is set to zero so that the Listen can set TRIGGER while the fixed action is taking place. Ziggy turns right if the sound direction (DIR) bit indicates the sound arrived at Ziggy's right ear first. The priority level of sound following is 70 so it will dominate far collision avoidance, but will be inactive if a collision is imminent or a wheel gets pinned.

Experimental Layout and Results

Before writing the program to carry out obstacle avoidance, the following measurements were taken from the IR sensors. The Jones and Flynn book was used as an object to test the sensors.

Table I.

Distance From Sensor (inches)	Side Sensor (book)	Side Sensor (white paper)	Front Right (white paper)	Front Left (white paper)	Front Right (book)	Front Left (book)
infinity	85	85	85	85	85	85
10	87	90	91	90	87	87
8	88	94	94	95	88	88
6	89	101	102	101	92	91

4	90	113	111	111	97	100
3	94	117	118	117	106	109
2	104	120	122	121	116	111
1	106	124	126	123	121	113
0	113	125	113	87		86

Each column in the table represents a particular sensor being tested with either a book or a white sheet of paper. The numbers ranging from 85 to 126 are the digital readings of the A/D ports corresponding to the sensors. The environment plays a big role in Ziggy's perception. It is practically impossible for Ziggy to determine the color of the object it is sensing with the IR sensors it has. The first threshold of 115 was too high for the poorer reflecting surfaces and Ziggy kept crashing into things. After several tries, the threshold of 100 seemed to work best for my environment. Narrow passageways with highly reflective surfaces may necessitate a higher threshold. Otherwise, Ziggy would think he was surrounded and would either continually back up or spin.

Ziggy's ability to follow sound is dependent on the voltage thresholds set by the two potentiometers. If one ear is more sensitive than the other, Ziggy always turns the same way no matter where the sound is coming from. If either ear is too sensitive, Ziggy responds to vibrations or sounds caused by its own motors. There is always a probability of error due to the thresholding problem. To try to alleviate this, Ziggy does not respond until he hears two claps in a row from the same direction.

Hearing tests show that Ziggy is able to guess the right direction about 80% of the time, assuming the potentiometers are adjusted correctly.

Conclusions

Drilling rectangular holes in the platform for the switches, and wire wrapping and soldering the switches took a couple of hours, but made it much easier to initialize P-code and download programs. Before, two jumpers were constantly being placed on J3 and J4 on the 6811 board and then removed. Now it is done with a single throw of a switch! I recommend that everybody do this in the beginning because it saves you time and energy in the long run.

The infrared sensors work great as analog sensors for detecting objects, especially if the objects are good reflectors (such as white walls). A disadvantage is that the threshold may have to be tweaked depending on the environment and some objects simply cannot be detected because they absorb IR too well. Ziggy can almost accomplish collision avoidance with only two front sensors positioned cross-eyed. There are some pitfalls to this minimalistic approach. First, because Ziggy only has no middle front IR sensor, he may run directly into narrow objects such as chair legs. There is also no test for height clearance. The side sensors solve the problem of one wheel getting pinned against a wall.

Sound following seems to be so full of difficulties, it is amazing Ziggy is able to get 80% success rate. Besides the tedium of setting the potentiometer thresholds, a more subtle problem is the fact that the second flip flop is not clocking in the same two signals that triggered it because there is a very short delay. Then there are acoustical properties of the environment which can affect Ziggy (such as echoing). For example, if Ziggy is near a wall, the echo off the wall may actually trigger the wrong sensor.

Caveats for future students include double checking wiring diagrams because it is difficult to change wire wraps, run tests (such as continuity) whenever possible. This may seem like extra work, but it is worth the time in the long run. If I were to start this project from scratch, I would use a circular platform because it can work in tight spaces more easily and is in my opinion more aesthetic than the rectangular platform. I learned a lot in this class and created a "moving vending machine" that turned out to be pretty cool.

Bibliography

J. L. Jones and A. M. Flynn, "Mobile Robots: Inspiration to Implementation," A. K. Peters, Ltd., Wellesley, MA, 1993.

F. G. Martin, "The 6.270 Robot Builder's Guide," F. G. Martin, 1992.

Motorola, M68HC11EVBU User's Manual, Motorola, Inc., 1992.

Appendix

```
int RSPDINT=0;
int LSPDINT=0;
float RSPD=0.;
float LSPD=0.;
float LDELTA=1.;
float RDELTA=1.;
int RIGHTSIDE;
int LEFTSIDE;
int LEFT;
int RIGHT;
int RIGHTSPEED[10];
int LEFTSPEED[10];
float MINDELTA;
int MAXSPD;
int TESTMODE;
int HEAR;
int VOL;
int p1;
int p2;
int p3;
int p4;
int p5;
int p6;
int p7;
int p8;
int ALLSTOP=1;
int SPINDELAY=600;
int TURNDELAY=800;
int BACKUPDELAY=500;
int GOSTRAIGHTDELAY=400;
int SIDEWAYSDELAY=1700;
int TRIGGER;
int DIR;
int OLDDIR;
int PRIORITY[10];
int NB;
int BEHAVIOR;

void wait(int msecs)
```

```
{
  long timer_a;
  timer_a=mseconds()+(long) msecs;
  while(timer_a>mseconds()) {
    defer();
  }
}

void init_globals()
{
  MINDELTA=0.5;
  MAXSPD=10;
  TESTMODE=0;
  PRIORITY[1]=0;
  PRIORITY[2]=0;
  PRIORITY[3]=0;
  PRIORITY[4]=0;
}

int sign(int x)
{
  if (x>0) return 1;
  else if (x<0) return -1;
  else return 0;
}

int abs(int x)
{
  if (x<0) return -x;
  else return x;
}

int random()
{
  return peek(0x100f)&1;
}

void stop()
{
  motor(0,0);
  motor(1,0);
```

```
}

void control_motors(int B)
{

  /* if speed is small, go by MINDELTA */
  /* If actual speed opposite in sign to desired, decrease by 80% */

  if ((abs(RSPDINT)>6) && (sign(RIGHTSPEED[B])==-sign(RSPDINT))) RDELTA=-0.8*RSPD;
  else RDELTA=MINDELTA*(float) sign(RIGHTSPEED[B]-RSPDINT);

  if ((abs(LSPDINT)>6) && (sign(LEFTSPEED[B])==-sign(LSPDINT))) LDELTA=-0.8*LSPD;
  else LDELTA=MINDELTA*(float) sign(LEFTSPEED[B]-LSPDINT);

  /* Change speed closer to desired by adding DELTA */
  RSPD=RSPD+RDELTA;
  LSPD=LSPD+LDELTA;
  LSPDINT=(int) LSPD;
  RSPDINT=(int) RSPD;

  /* Keep speed from going over MAXSPD */
  if (abs(RSPDINT)>MAXSPD) RSPDINT=sign(RSPDINT)*MAXSPD;
  if (abs(LSPDINT)>MAXSPD) LSPDINT=sign(LSPDINT)*MAXSPD;
  if (TESTMODE==0) {
    motor(1,RSPDINT);
    motor(0,LSPDINT);
  }
}

void arbitrate()
{
/* Arbitration: Decide which recommendation for motor
   control has highest priority and execute control_motors
*/
int i;
int j;
while (1) {
  j=1;
  for (i=2; i<=NB; i++) {
    if (PRIORITY[j]<PRIORITY[i]) j=i;
  }
```

```
  BEHAVIOR=j;
  control_motors(j);
  defer();
}
}

void reset_ears()
{
  /* Clear interrupt */
  bit_clear(0x1022,128);
  /* Set directions */
  bit_set(0x1026,128);
  bit_clear(0x1026,4);
  /* Reset JK Flip Flop for sound */
  bit_clear(0x1000,128);
  bit_set(0x1000,128);
  /* Reset Hear Flag */
  HEAR=0;
}


void read_sensors()
{
  while (1) {
    LEFTSIDE=analog(0);
    RIGHTSIDE=analog(1);
    LEFT=analog(3);
    RIGHT=analog(2);
    if ((HEAR==0)&&(analog(4)>128)) {
      DIR=peek(0x1000) & 4;
      HEAR=1;
      }
    defer();
  }
}

void spin(int d, int b)
{
/* Spin right or left depending on d */

int r;
```

```
int l;

  if (d>0) {
    r=MAXSPD;
    l=-MAXSPD;
    }
  else {
    r=-MAXSPD;
    l=MAXSPD;
    }
  RIGHTSPEED[b]=r;
  LEFTSPEED[b]=l;
  wait(SPINDELAY);
}

void turn(int d, int b)
/* Turn right or left depending on d */
{
int r;
int l;

  if (d>0) {
    r=MAXSPD;
    l=0;
    }
  else {
    r=0;
    l=MAXSPD;
    }
  RIGHTSPEED[b]=r;
  LEFTSPEED[b]=l;
  if (b!=3) wait(TURNDELAY);
  if ((b!=3)&&(LEFT<=87)&&(RIGHT<=87)) {
    RIGHTSPEED[b]=MAXSPD;
    LEFTSPEED[b]=MAXSPD;
    wait(GOSTRAIGHTDELAY);
    RIGHTSPEED[b]=MAXSPD-r;
    LEFTSPEED[b]=MAXSPD-l;
    wait(GOSTRAIGHTDELAY);
  }
}
```

```
void avoid_collisions()
{
/* Behavior #1
   This behavior is always high priority --
   2nd only to test() behavior
   because it checks for wheels stuck (side)
   or imminent collisions
*/
while (1) {
  if (RIGHTSIDE>120) {
    PRIORITY[1]=100;
    RIGHTSPEED[1]=-MAXSPD;
    LEFTSPEED[1]=0;
    wait(SIDEWAYSDELAY);
    if ((RIGHTSIDE>120)&&(LEFTSIDE>120)) TESTMODE=1;
    RIGHTSPEED[1]=0;
    LEFTSPEED[1]=-MAXSPD;
    wait(SIDEWAYSDELAY);
    if (RIGHTSIDE>120) {
      PRIORITY[1]=80;
      RIGHTSPEED[1]=MAXSPD;
      LEFTSPEED[1]=0;
      wait(SIDEWAYSDELAY);
      RIGHTSPEED[1]=0;
      LEFTSPEED[1]=MAXSPD;
      wait(SIDEWAYSDELAY);
    }
    PRIORITY[1]=0;
  }

  if (LEFTSIDE>120) {
    PRIORITY[1]=100;
    RIGHTSPEED[1]=0;
    LEFTSPEED[1]=-MAXSPD;
    wait(SIDEWAYSDELAY);
    if ((RIGHTSIDE>120)&&(LEFTSIDE>120)) TESTMODE=1;
    RIGHTSPEED[1]=-MAXSPD;
    LEFTSPEED[1]=0;
    wait(SIDEWAYSDELAY);
    if (LEFTSIDE>120) {
```

```
      PRIORITY[1]=80;
      RIGHTSPEED[1]=0;
      LEFTSPEED[1]=MAXSPD;
      wait(SIDEWAYSDELAY);
      RIGHTSPEED[1]=MAXSPD;
      LEFTSPEED[1]=0;
      wait(SIDEWAYSDELAY);
    }
    PRIORITY[1]=0;
  }

  if (RIGHT>100) {
    PRIORITY[1]=90;
    RIGHTSPEED[1]=-MAXSPD;
  }

  if (LEFT>100) {
    PRIORITY[1]=90;
    LEFTSPEED[1]=-MAXSPD;
  }

  if ((LEFT<=100)&&(RIGHT<=100)) PRIORITY[1]=0;
  defer();
}
}

void avoid_obstacles()
{
/* Behavior #2
   This behavior checks for far obstacles
*/
int d;

while (1) {
  if ((RIGHT>87)&&(LEFT>87)) {
    if (RIGHT>LEFT) d=0;
    else if (LEFT>RIGHT) d=1;
    else d=random();
    PRIORITY[2]=50;
    spin(d,2);
    while ((RIGHT>87)&&(LEFT>87)) ;
```

```
    PRIORITY[2]=0;
  }

  if ((RIGHT>87)&&(LEFT<=87)) {
    PRIORITY[2]=50;
    turn(0,2);
  }

  if ((RIGHT<=87)&&(LEFT>87)) {
    PRIORITY[2]=50;
    turn(1,2);
  }

  if ((RIGHT<=87)&&(LEFT<=87)) {
    PRIORITY[2]=10;
    RIGHTSPEED[2]=MAXSPD;
    LEFTSPEED[2]=MAXSPD;
  }

  defer();
}
}

void listen()
/* Behavior #4: Does not affect motors directly
*/
{
long curtim;
int READY;
while (1) {
  /* Wait for 0.1 sec period of silence */
  READY=0;
  while (READY==0) {
    READY=1;
    reset_ears();
    curtim=mseconds();
    while (mseconds()<curtim+(long)100) {
      if (HEAR>0) READY=0;
    }
  }
```

```
  /* Wait for next sound */
  while (HEAR==0) ;

  /* Reset ears before triggering fixed action pattern */
  reset_ears();
  if ((TRIGGER==1)&&(DIR==OLDDIR)) TRIGGER=TRIGGER+1;
  else if (TRIGGER==1) OLDDIR=DIR;
  else {
    OLDDIR=DIR;
    TRIGGER=TRIGGER+1;
  }
  defer();
}
}

void follow_sound()
/* Behavior #3: Priority 70
   Ziggy follows claps
*/
{
while (1) {
  if (TRIGGER>1) {
    TRIGGER=0;
    /* Turn towards sound and beep */
    if (DIR==0) {
      play("1a");
      PRIORITY[3]=70;
      turn(0,3);
      /* Wait for motor_control routine to
         catch up */
      while (RSPDINT!=0) ;
      PRIORITY[3]=0;
    }
    else {
      play("1c");
      PRIORITY[3]=70;
      turn(1,3);
      /* Wait for motor_control routine */
      while (LSPDINT!=0) ;
      PRIORITY[3]=0;
    }
```

```
  }
  defer();
}
}

void play_music()
{
while (1) {
  if (TESTMODE==0) charge();
  wait(30000);
  if (TESTMODE==0) ds9();
  wait(30000);
  if (TESTMODE==0) pp();
  wait(30000);
  if (TESTMODE==0) classics();
  wait(30000);
}
}

void test()
{
/* TESTMODE inhibits the motors (see control_motors)
   and runs this process until Ziggy is set free again
*/
while (1) {
  if (TESTMODE>0) {
    stop();
    while ((RIGHTSIDE>120)&&(LEFTSIDE>120)) defer();
    while ((RIGHTSIDE<=120)||(LEFTSIDE<=120)) defer();
    TESTMODE=0;
  }
  defer();
}
}

void main()
{
  init_globals();

/* Senses */
  p1=start_process(read_sensors());
```

```
  p2=start_process(listen());

/* Behaviors */
  p3=start_process(avoid_collisions());
  p4=start_process(avoid_obstacles());
  p5=start_process(follow_sound());
  NB=3;

/* Test Mode */
  p6=start_process(test());

/* Arbitration */
  p7=start_process(arbitrate());

/* Music */
  p8=start_process(play_music());
}
```