Hacking the HERO: Development of a Game-Playing Robot

Drew Bagnell
EEL 5666
Dr. Keith L. Doty
TA: Scott Jantz
Final Report, 12/9/96

Abstract

This paper is on the design and construction of an autonomous robotic platform for EEL5666 - Intelligent Machine Design Lab at the University of Florida. The goal of the project was to create a robust robotic platform to implement a game-playing autonomous agent. The agent, named Marco, says the word “Marco” and the person playing with the robot replies “Polo”. The robot attempts to locate the person from the noise they make. The first critical goal was to development of the robotic platform. The mechanics of the robot were developed from a Heathkit HERO. The author developed motor and stepper control circuitry and software to implement the agent. The author also developed the required sound generation and sound localization circuitry. Finally, the author developed the code necessary to implement the central behavior. The elements of this paper describe the hardware and software systems developed to attain this goal.

Executive Summary

The motivation for this project was the creation of an autonomous agent capable of playing a game. In particular, the robot is designed to simulate the children's game "Marco, Polo". The agent says "Marco!", and listens for a response ("Polo!") from the other player. The robot then tries to find the other player.

The development of a robust platform was critical for successful completion of this project. The chassis and motors of the robot came from a Heathkit HERO robot. The electronics to drive the mechanics were developed specifically for this project.

The autonomous agent has a complete sensor suite consisting of four analog IR sensors, two sound intensity sensors, and a stepper limit switch compose the sensor suite of the robot. The development of the "ears" of the robot, the sound intensity sensors, was essential to the implementation of the game playing behavior. .

The "brain" of the agent is a memory and I/O expanded MC68HC11E9 evaluation board from Motorola. The software, including both drivers and behavior code is written entirely in the Image Craft 'C' compiler for the HC11. Includes a master process that occurs at a regular interval to insure consistency in sensor reading and actuation. A variety of behaviors are implemented in a subsumption architecture. The following simple behaviors are implemented: phonotaxis, phonophobia, collision avoidance, and resetting the stepper. Finally, the composite behaviors of playing the game "Marco Polo", and feeling "spurned" were generated from the other behaviors.

Introduction

The autonomous agent Marco was designed around the theme of game playing robot. All the systems incorporated within the robot are necessary to the fulfill this aim. The agent is designed to play a variation on a children's game called "Marco, Polo." In this game, two players are in a confined area. One player, the "seeker", and in this case the robot, is blinded (in this model the robot has no complex vision system, and is effectively blind already). The seeker says "Marco" and the other player(s) must respond by saying "Polo". From the reply, the seeker tries to find the other player or players.

Integrated System

The robot's hardware systems are integrated as depicted in Figure 1. A "master task" clocked off of the output compare hardware of the HC11 reads the sensor suite, can arbitrate process, and determines the next action of the actuators.

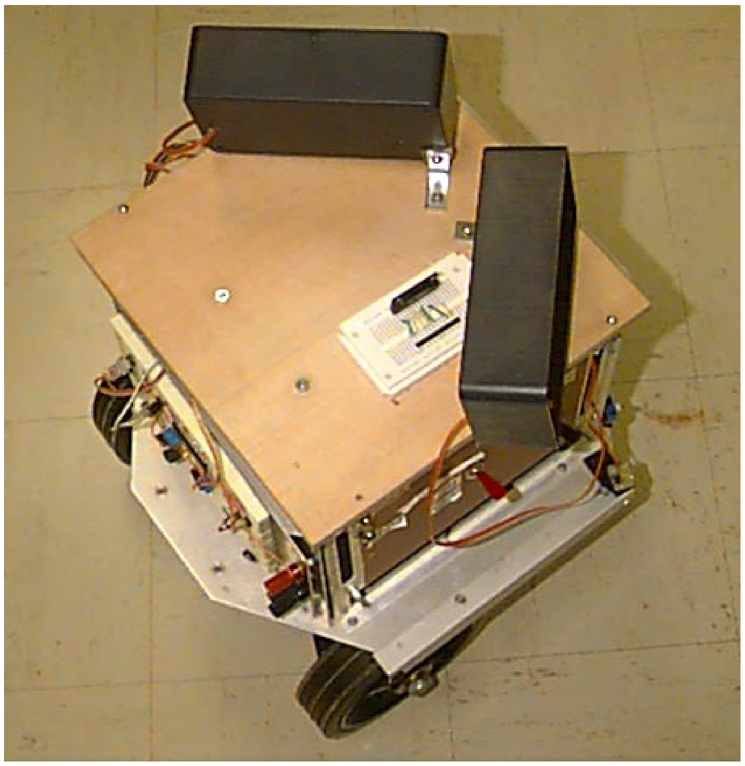

Figure 1: Autonomous agent Marco (top view)

The sensor values are stored in a sensor blackboard that the behaviors have access to. The arbitration of behaviors is determined from a rule-based subsumption architecture. Further, the "master task" implements a smoothing function on the motor so that is not forced to rapidly and instantaneously change direction

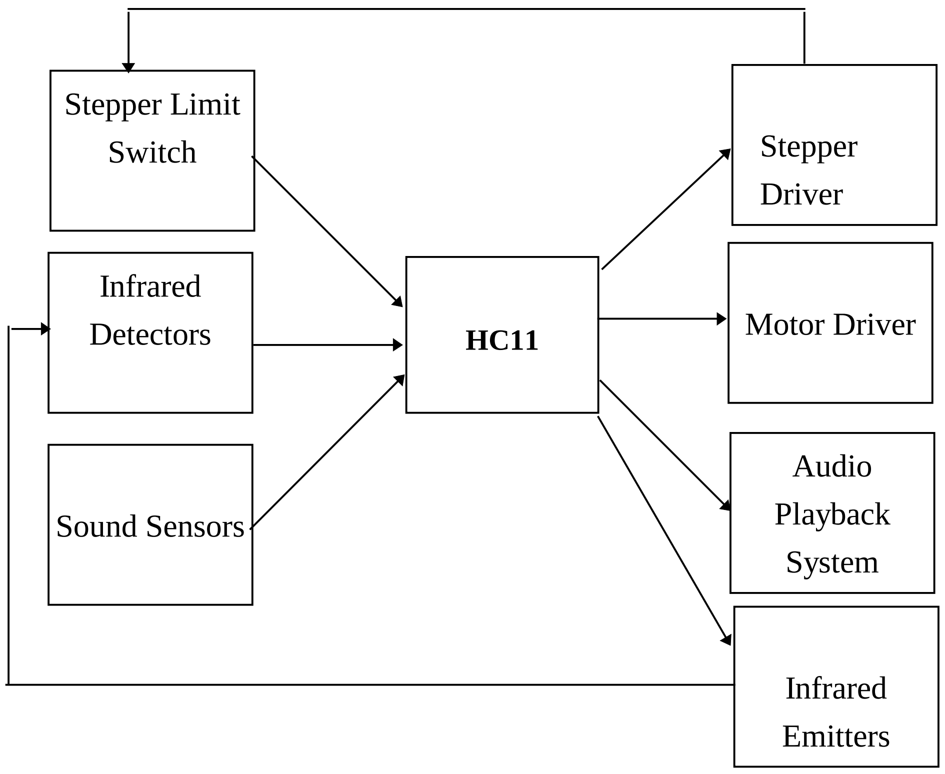

Figure 2: Block Diagram of Agent's Hardware

Each of the behaviors is described more completely starting on page 18. All of the software running on the agent, including behaviors and driver code is written for the ImageCraft HC11 'C' compiler. [7]

Mobile Platform

Power and MCU Systems

Power to the robot is provided by two onboard lead-acid gel-cell batteries connected in series, providing 12V at 5Ah. The batteries provides both drive motor and MCU power (via a voltage regulator). The batteries can be recharged through a set of jacks mounted on the side of the robot. The battery power provides several hours of charge.

At the heart of Marco is a MC68HC11E9 microcontroller that has been expanded with a Novasoft ME11 and MSX11 sensor board. These boards provide expanded memory, expanded analog input capabilities of the microprocessor, and memory mapped I/O.

Mobile Chassis

Marco is based mechanically on a Heathkit HERO robot, his chassis and motors having come from one. He measures 15’’ x 14’’ at his widest points and stands 14.5’’ high without his “ear” sensors. The robot has three ground contacts in a tripod arrangement. The single front wheel is driven by a DC motor and turned directionally by a stepper motor.

Propulsion

The combination of DC motor and stepper allow the robot a great deal of flexibility and smoothness. Marco is capable of forward and backward motion, and of spinning on his axis. This makes possible escape from tight quarters. Furthermore, the agent is quite fast, reaching speeds of approximately 1mi/hr. The design and realization of a system for locomotion was an involved and critical part of this project.

Sensor/Actuator Suite

Sharp IR Sensors

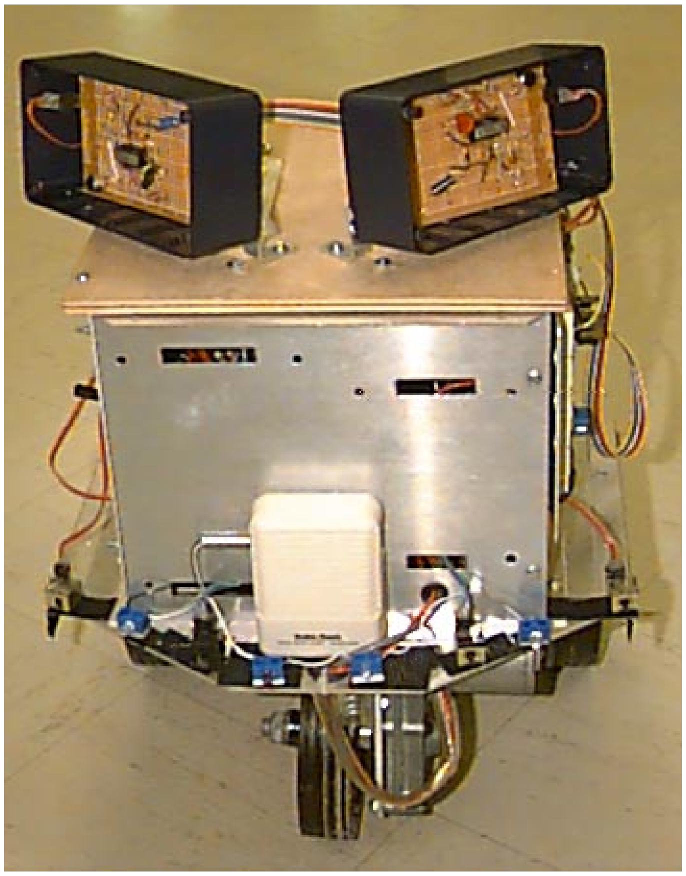
Figure 3: Autonomous agent (front view) showing IR emitter/detector pairs, ear sensors(blue bloxes), and speaker(white box).

Marco has a battery of 4 analog-modified Sharp IR sensors mounted facing to the front of the robot on the front bumper (Figure 3.) These pick up reflections from the four 40-KHz modulated IR LED's mounted on the platform. The four LED's are connected in series and driven from a latch on the ME11.

This sensor arrangement provides necessary information about nearby objects critical to the implementation of a collision avoidance behavior. This is the method used throughout the EEL 5666 class, and has proven quite successful. Given reasonable constraints on the size of the objects (must be higher than the bumper of the robot, and of significant size) and reflectiveness of the objects, the agent is capable of detecting obstacles with more than enough time to prevent collisions. The primary actuation/arbitration routine reads these sensors on a regular basis (every 32ms) and stores the result in a "sensor blackboard" that all behaviors have access to.

Sound detection and location system

As the primary behavior of this robot is a variation of phono-taxis, is is necesssary for the robot to both detect the presence of sound, and to directionalize the sound. A number of techniques were considered for this. First, a pattern recognition technique wherein the agent would find the peak of incoming signals and then calculate time differences between the arrival of the peak to sensors at different locations on the robot body was considered. This algorithm proved to be extremely processor intensive, and the author thought would be difficult to implement successfully using an HC11.

Ultimately, a simpler, more biologically-based, and in some regards more elegant approach was considered that would simply hold the peak value of two sensors mounted on the right and left sides of the agents and compare intensity differences. This was

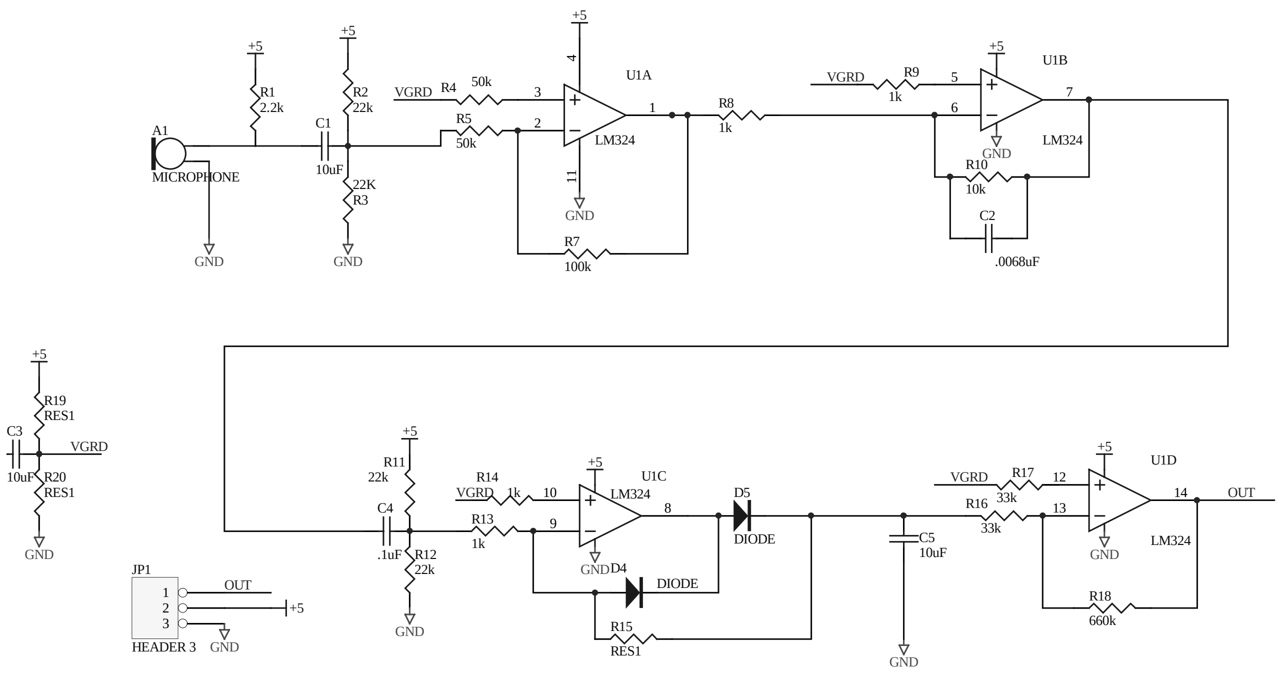

Figure 4: Peak detect and hold schematic for "ear" sensors

implemented using analog circuitry. Each ear circuit is essentially an amplifier and peak rectifier[9] that provides as output a slowly time-varying voltage from 0 to 2.5 volts that is a representation of the intensity of the signal. The lower the voltage the greater the intensity of the peak value of the sound heard. The outputs of these circuits are fed into PE4 and PE5 of the HC11 were they are converted to digital representations. The implemented software successively reads the analog ports over a period of time, usually approximately 2 seconds. It stores the smallest value (greatest intensity) for both the left and right sensors. From this Marco can determine the approximate location of the loudest sound source.

The circuit boards are mounted on a soft, sound absorbent foam to minimize vibration due to the motion of the platform. The circuit board are then mounted in blue plastic boxes that help to directionalize the sound

The author did encounter some problems in the development of the sensor. First, noise was a significant problem. High frequency noise on signals from the microphone was effectively eliminated with a low-pass filter having a corner at 10kHz. The mike also generated a large amount of low frequency noise. This was reduced through a small coupling capacitor (high pass). The net effect of the circuit is to act as a bandpass in the approximate frequency spectrum of voice.

Another problem was the very large dynamic range of the sounds encountered. The gain of the amplifiers were empirically chosen to provide the necessary sensitivity to be able to localize sound. However, very loud sounds, like claps close to the robots ears saturate the amplifiers and the agent is unable to tell where the sound originated.

Given sounds of moderate intensity, however, the robot is generally quite successful in determining which side the sound is coming from.

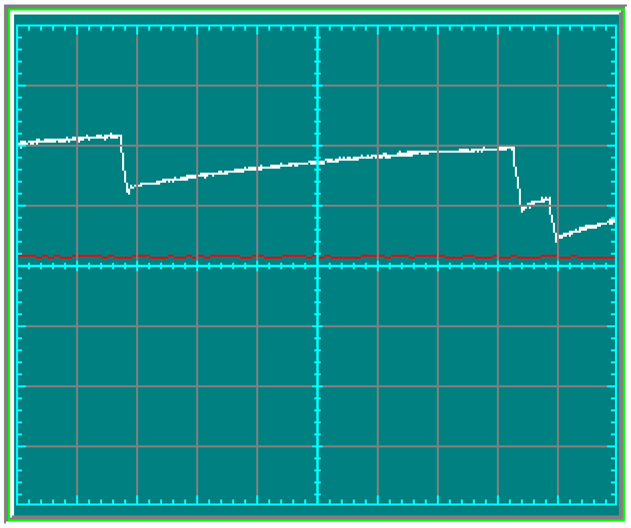
Figure 5: Oscilloscope output from "ear" circuit. Author is saying the word "test". Horizontal axis is time (ms) Vertical axis is voltage (V) .

Audio Record/Playback system

The design goal of the agent required the development of a system that would allow the device to say the word “Marco”. A simple circuit using an ISD1000A Voice Recording and Playback chip (similar to those used in certain answering machines) allows the robot to play back a recorded voice. The circuit has been simplified for the purposes of this

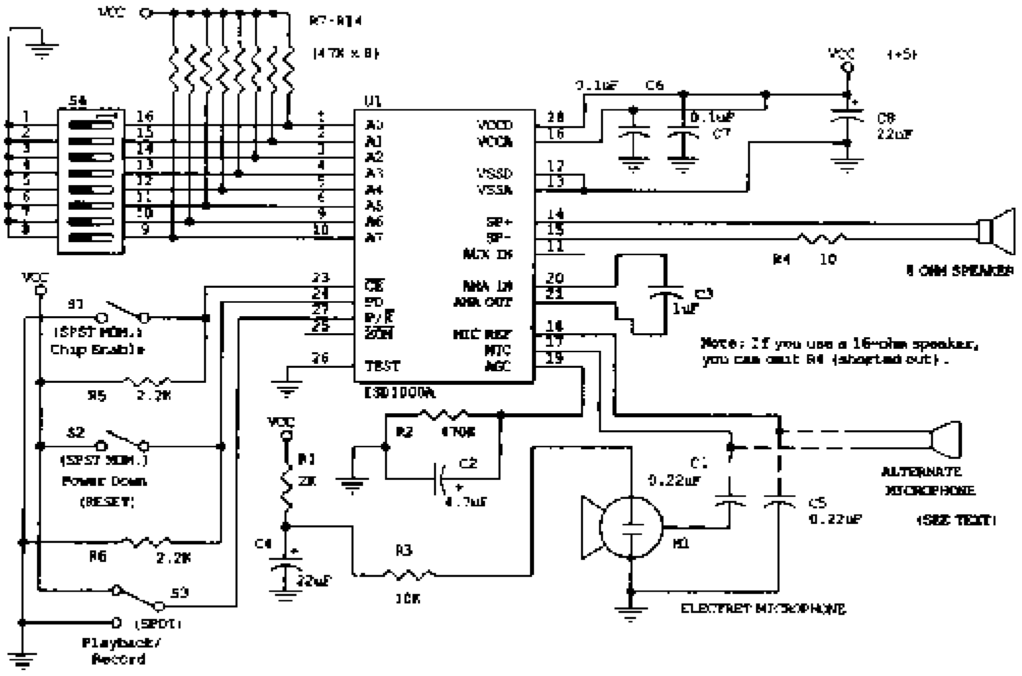

Figure 6: Circuit used for Audio Record/Playback. (ISD1000A Databook, 1992).

robot. Since only two samples are needed, the address lines of the chip are all grounded except A6, which is driven off a latch on the sensor board. After recording a sample, the circuit is switched into playback mode. The chip enable input is driven off a bit of a latch mapped at 0x6000. When a behavior requires the robot to play back a noise it simply toggles this bit low (sending a short duration, active low pulse) and the chip plays the first message recorded in memory. This circuit also makes future expansion relatively easy. Another memory mapped latch could be added that would allow all the address bits to be changed. Then more samples, up to a total of 20 seconds of recording, could be played back for different events. The setup also allows the chip to be recorded while plugged into the robot.

The speaker outputs of the chip provide a signal that is load enough to drive an 8 ohm speaker, but it is typically not loud enough and tends to distort the sample. Thus, the output of the chip on the agent is taken to a 386-based audio amplifier that drives a speaker. This proved more than loud enough, and prevents the distortion of the signal.

Locomotion

The front wheel is rotated by a stepper motor through a gear train that increases the motor's torque, and is driven with a DC motor. The stepper is bolted onto the base of the chassis of the robot.

Stepper Hardware

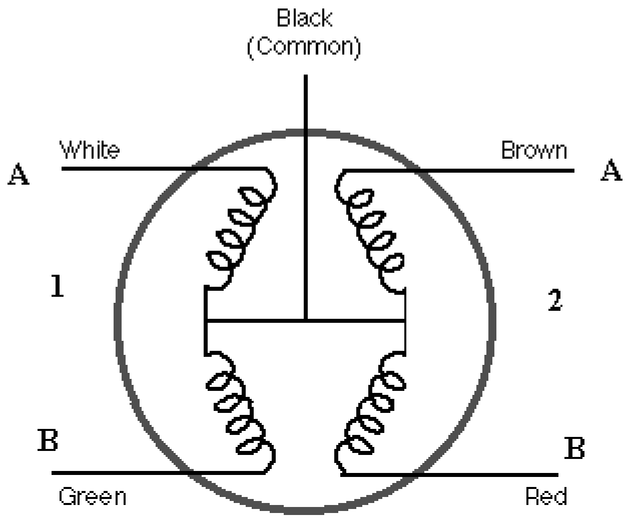

Figure 7: 4 phase stepper [1]

The stepper motor is best though of as an electric motor without a commutator. The rotor is made of a permanent magnetic material, and the windings of the electromagnets are all in the stator. The particular stepper in this autonomous agent has a very common wiring scheme [Figure 1].

To make the motor turn, the coils must be excited in a particular sequence. One sequence turns it clockwise, and the sequence in reverse turns it counterclockwise. Since the motor is stepped at essentially audio frequencies, the author concluded the microcontroller on the autonomous agent would only be moderately loaded by interrupt required to do the stepping. This decision allows for great flexibility in controlling the stepper, as different sequences and different speeds are easily implemented.

Some hardware was necessary to interface the microcontroller to the stepper. First, a 74HC574 latch is mapped using the ME11 expansion board to the memory location 0x6000. This latch holds the last value written. The lower four bits of the latch are used for the stepping. Each of these lower four output bits connects through a 1kohm resistor to the base of a TIP100 NPN transistor. These transistors are actually high-current Darlington pairs, with protection diodes built in to handle the large back electromotive force generated when the coils are turned off. The center tap of the coils is pulled to 12 volts, and the emitter of each transistor is grounded.

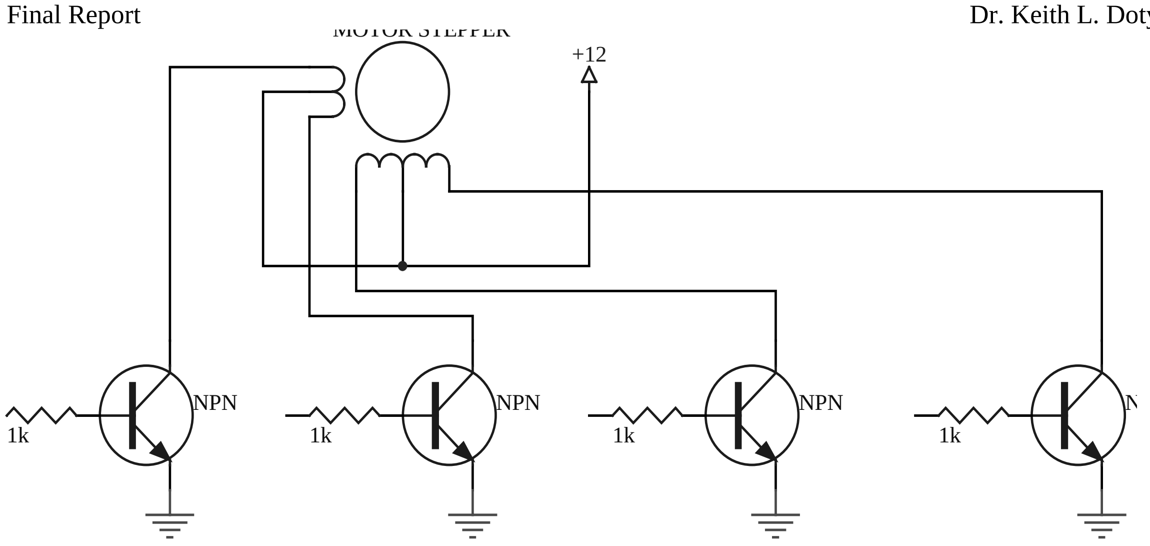

Figure 8: Stepper Controller. Inputs are driven from lower bits of latch.

When a latch output is at logic zero, zero volts at the transistor base put the transistor in the cutoff mode of operation, and no current flows through the coil. When the latch output is at logic one, five volts is applied at the base-emitter junction, and the transistor enters saturation, and current flows through the coil. The current drawn by the coil, assuming the stepper is not stalled is approximately 400mA.

This approach is simple, but has a couple of drawbacks. The first is that there is no current limiting on the coil, other than its own internal resistance. Testing demonstrated that leaving a coil on for 15 minutes made it noticeable hot, but did no permanent damage. Another problem is related to the speed of operation. The time to energize the coil for a given voltage is a function of the series inductance-resistance time constant $\tau = L/R$. By adding a small value resistance in series with the coil, one can lower this time constant, and hence raise the maximum operating speed of the stepper.

Adding resistance is inefficient in that more of the power delivered by the source is converted to heat, and both efficiency and overheating are concerns on the autonomous agents. Further, the coil has less current through it, and consequently generates less torque. Evaluation of the performance of the circuit without inline resistance suggested that it was fast enough, and generated a reasonable amount of torque. However, if performance must be increased, other techniques that involve intelligently applying higher voltages for shorter periods of time can maintain a high torque and increase speed, at a minimal cost in efficiency. The only drawback is circuit complexity and cost.

Stepper Software

The job of the stepping software driver is quite simple to describe. It must take as an input the number of steps to make, and output the appropriate bit patterns to the memory mapped latch, with due pauses in between updates. I am aware of three bit patterns that will step a motor of the type diagrammed above. The two more common stepping patterns are listed below. The first energizes a single coil at a time, while the second energizes two coils at a time. The second pattern provides approximately 40% more torque, but requires twice the power [1]. In the autonomous agent the author implemented, the first pattern generated enough torque for the smooth surface it was being run on, and concerns of power consumption and coil-winding overheating were

	Normal	High Torque
Winding 1a	1000100010001000	1100110011001100
Winding 1b	0010001000100010	0011001100110011
Winding 2a	0100010001000100	0110011001100110
Winding 2b	0001000100010001	1001100110011001

Figure 9: Winding bit patterns [1]

deemed more pressing than torque. If the surface to be run on changes to a higher friction one like carpeting, the pattern can easily be adjusted. There is also a less common pattern that generates only slightly more torque at the price of three coils being active at once.
The stepper driver was implemented as an interrupt service routine written in ICC11. The driver is clocked off of Output Compare 3. Every 5mS (a frequency of 200Hz) the driver interrupts the processor. The driver maintains a number of global variables. The first is one called LATCH6000, which is simply the bit pattern currently loaded into the memory-mapped latch. This is maintained in a variable because the latch can not be read back: reading memory location 0x6000 returns 0. All operations are performed on the global variable and then the new value of LATCH6000 is written to 0x6000. The other globals maintained are the current location of the stepper (variable STEP) and the location the stepper should be moving towards (NEXT_STEP), and finally where within the step the motor is (which of the four bit patterns) (SUBSTEP). The location of the wheel facing forward is defined as STEP=5000, and counterclockwise is referenced in the positive direction.

The bit patterns are picked deliberately in the order required such that if the software asks the stepper to change direction in mid-step, it will not lose a step. Even if this were to happen, it would result in a loss of at most one step, and assuming a random direction, the resulting loss of step should be the same as the "random-walk" problem— the differential from true position goes as the root of n, where n is the number of times a step is lost. With steps as small (well under a degree) as they are, this problem should be negligible .

A more serious problem is that under the circumstance that NEXT_STEP and STEP are equal (that is, the motor need not move) the driver turns all the coils off. This keeps power consumption down, and minimizes the heating effect on the stepper. The motor's holding torque with no current applied is quite low however. I have successfully run the autonomous agent on a smooth floor for 10 minutes with no measurable loss in step position, but further tests need to be conducted. Furthermore, the problem may be much worse on a surface of higher friction.

Stepper Position Sensor

A stepper, unlike a servomotor, is run without any feedback circuitry. This can be problematic in two instances. First, when the autonomous agent is initialized, the starting position of the stepper is unknown. Furthermore, the potential exists for the motor to lose steps (two such circumstances are described above), and this cannot be detected.

The author decided that hand-centering the wheel, each time the agent needed to be run was unacceptable. The solution to both problems involves closing the feedback with a limit switch.

A micro-switch is mounted on the underside of the robot, such that when the agent turns its wheel approximately 80 degrees (135 steps) from the center in the positively referenced direction(counterclockwise), it closes the switch. This switch is wired such that when closed it grounds the second bit of Port E of the HC11.

To solve the initialization problem, a routine reset_stepper() was written that simply moves the stepper counterclockwise, and then checks this switch. If the switch is closed, the stepper is stopped, and its current location is defined as 135 steps greater than the center position. Finally, the stepper is told to go the center location. The code to do this follows:

```
int
reset_stepper() {

    int bump_switch;

    do {
        bump_switch=analog(1);
        /*write_int(bump_switch);*/
        NEXT_STEP=STEP+2;
        while(STEP!=NEXT_STEP){} /*wait for hardware to catch up....*/
    }while(bump_switch > 80)

    INTR_OFF();
    STEP=5135;
    NEXT_STEP=5000;
    INTR_ON();

    while(STEP!=NEXT_STEP){} /*wait for hardware to catch up....*/

    return(0);
}
```

Because the possibility of lack of synchronization between stepper and software, due to collision or rough surfaces, a behavior is implemented that resets the stepper on a regular basis. This subsumes other behaviors and occurs approximately once a minute. The necessary frequency of resets is obviously a function the robot is running in, and will be fine-tuned to match that.

Motor Hardware

The front wheel of the autonomous agent is driven with a DC motor mounted through a gearbox to the wheel. The motor is designed to be run with at 24 volts. Experiments showed, however, that it runs with very acceptable performance at 12 volts. This DC motor must be run in both directions and must be interfaced to the microcontroller. The power is delivered to the motor load with an integrated circuit H-bridge, the Motorola 3008. The 3008 can handle up to 16 amps continuous current (well above the peak draw of this DC motor), and provides internal diode protection to prevent back EMFs from destroying the MOSFET driving the circuit. A circuit was constructed based on one designed by Erik de la Iglesia in his design for the autonomous submarine project. This design is robust, in its inclusion of photo-transistors to isolate logic and power, and in its guaranteeing that both gates on a side of the bridge can never be on. It is also very efficient in terms of components as some of the selection and inversion logic is implemented in the opto-isolators. The opto-isolators also force a gate delay between turning on the gates on a side of the bridge. This keeps the bridge from shorting to ground for even the nanoseconds of switching time of the FETs.

The inputs to this circuit are a direction and an enable bit. These bits are written to at bits 5 and 6 of the latch memory mapped at 0x6000. The software takes care not to interfere with the bits dedicated to the stepper.

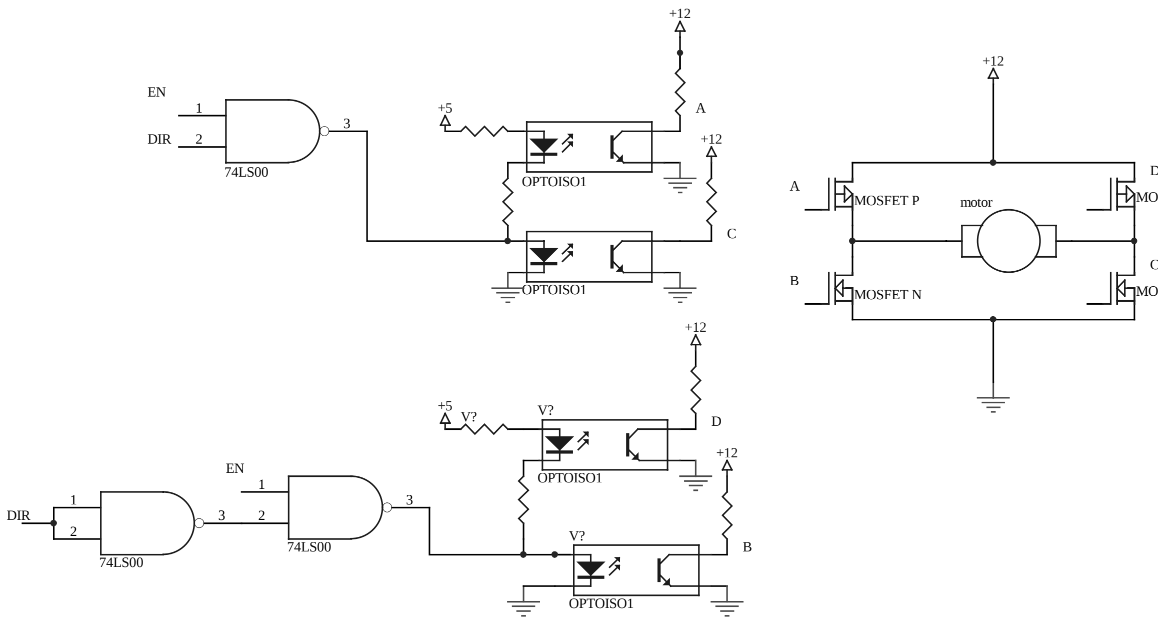

Figure 10: DC motor controller hardware (modified from design by E. de la Iglesia)

Motor Control Software

The motor driver software is very similar to that written in for the Talrik motors. Pulse-width modulation of the enable bit is accomplished through Output Compare 3. The motor is run at frequency of 30.52 Hz (the rollover rate of free running counter). Interface to the interrupt service routine is provided through a duty-cycle global variable. A routine motor() is written as user-level code that accepts as input the pulse-width in percentage, and the direction (as the sign of the percentage). The motor() routine also directly writes the direction bit at 0x6000. In this implementation, motor() is only directly addressed by the arbitration/actuation process. This allows control of actuation to remain solely in the domain of the master process, and makes smoothing the commands sent to the motor much more simple.

Behaviors

A number of simple behavior routines were written to take advantage of the sensors and actuators implemented on the agent. Two compound behaviors, “Marco, polo” and “Spurned” were also implemented.

Reset Stepper

This simple behavior stops forward motion, and then moves the stepper until it hits a limit switch in its fully counter-clockwise position. It then returns the stepper back to the center position. This behavior happens on a regular basis (approximately once every 2 minutes). Also, if the limit switch is pressed, the robot will reset the stepper as this should not occur during normal running of the agent. This is the most fundamental behavior as it success is critical to nearly all the other behaviors.

Collision Avoidance

This behavior is the essential element of the robots self-preservation instinct. This behavior overrides the others except for the stepper resetting behavior. The behavior is simple and implemented using switching functions. The author hopes to improve the implementation using a dynamics algorithm similar to that of dynam11.

Phonotaxis

When this behavior is activated the robot tries to head in the direction of the loudest sound it detects. It samples both the right and left ear quickly for approximately 2 seconds, and then heads in the direction of the loudest returned noise. If neither noise is louder than a threshold it does not change course, and is they are approximately equal in intensity in continues to go straight.

Phonophobia

This behavior is the dual of the previous behavior. It is implemented to simulate biological behavior. The agent moves away from loud noises.

Sound generation

In this simple behavior the robot is “bored” and plays backs the recorded sound in its sound chip.

“Marco! Polo!”

This behavior is in a fact the integration of two distinct behaviors. The first is that of the sound generation. The robot turns on its sound chip and plays back the recorded message. It then executes the phonotaxis behavior and heads for the loudest source of sound, making the assumption that the sound is its partner in the game replying “Polo”. In this way the robot plays the game “Marco, polo.”

“Spurned”

This is also a compound behavior. If the agent does not get an audible response to its “Marco, polo” behavior it will try again 3 times. After that, the robot becomes “spurned” and activates the sound generation saying “No Marco for you!”. Then the agent uses a the behaviors phonophobia and collision avoidance to stay away from sound and objects.

Arbitration System

The robot stays in collision avoidance during most of its running time. This gives a preference for moving forward. The robot resets is stepper on a regular basis, and in response to the limit switch on the stepper closing. Other behaviors are entered as a function of time: the robot will begin playing “Marco, polo” for a period of time, and then will switch to “spurned”.

Conclusion

The development of Marco has reached the stage where it satisfies the criterion set forth in the EEL 5666 syllabus. The robot has a complement of 3 sensors: analog IR detectors, analog sound detectors, and a stepper limit switch. Further, it has 3 actuators: audio playback, DC motor driver hardware and software, and stepper driver hardware and software. Finally, seven behaviors have been successfully implemented on the platform: collision avoidance, phonotaxis, phonophobia, sound generation, “Marco, polo!”, “spurned”, and resetting the stepper.

A number of improvements could be made to the hardware of the robot. First, a more complete array of IR detector/emitter pairs should be placed on the robot and integrated into the behaviors. These should be mounted to keep the robot from hitting its top, particularly the ears, on low hanging surfaces. It would also be advantageous to mount additional sensors in such a way as to see obstacles below the bumper, and to detect drop-offs in the surface ahead of the agent. A bump-sensor should also be implemented to allow the agent to detect non-reflective surfaces, and to implement learning algorithms for collision avoidance.

The code written for the agent is not yet fully mature. It could be improved in a number of ways. First, the motor control and collision avoidance routines could be improved to take advantage of the work done by Dr. Keith Doty and Scott Jantz in non-linear dynamics control of small autonomous agents. Further, the arbitration network could be improved by the integration of the neural network work of Lee Rossey and Pedro Kulzar. Also, implementing a simple local mapping algorithm would dramatically increase the sophistication of the agents game playing abilities. The robot should be able to maneuver around an obstacle in its path to reach it partner in the game.

Works Cited

[1] Douglas Jones. Control of Stepping Motors, a tutorial.
http://www.cis.uiowa.edu. 1995

[2] Erik de la Iglesia. Final Report: R2, Real Time Contouring.
IMDL Final papers, http://www.mil.ufl.edu. 1995

[3] M68HC11 Reference Manual
Motorola 1991

[4] MC68HC11E9 Technical Data
Motorola 1991

[5] High-Speed CMOS Data
Motorola 1993

[6] Paul Horowitz and Winfield Hill. The Art of Electronics. Cambridge University Press, 1989.

[7] Barbara Webb, “Using robots to model animals: a cricket test” from Robotics and Autonomous Systems 16, 1995

[8] Chirstina J. Willrich, ImageCraft 68HC11 C Compiler and REXIS User Manual. ImageCraft, 1995.

*** Special thanks to Iván Ricardo Zapata, and Christopher Beatty for being the voices of Marco.**

Appendix A- Software

dday.c

```
/* Next working version of MARCO-POLO playing routine for robot */
/* This is avoid9*/
/* D-day version */


#include <serial.h>
#include "step.c"
#include <motor.h>
#include <analog.h>

#define IR *(unsigned char *)0x7000
/*extern unsigned char LATCH6000;*/


char clear[]={0x1b,0x5B,0x32,0x4A,0x04};  /* clear screen */
char place[]={0x1b,'[','2',';','2','H'};

int left_ir,right_ir;
int next_speed=50,current_speed=0;
int main_counter;
int limit_switch=0;

extern int STEP;

/*Prototypes*/
int phonophobia(void);
void avoid(void);
void init_arbitrator();
void arbitrator(void);
int game(void);
int msleep(int tenths_of_seconds);
#pragma interrupt_handler arbitrator

int
main(void) {


      int
i,avoid_counter,DONE_PLAYING,SPURNED,spurned_start_time,spurn_counter;
      init_serial();
      init_analog();
      init_stepper();
      init_motors();

      /*clear SPURNED*/
      SPURNED=0;
      spurn_counter=0;

      /*turn off latch*/
      LATCH6000 |= 0X40;
      /*make 4000 latch 0's*/
      ADDR4=0;

      /*turn ir on*/
      IR=0xff;
      /* required initialization */
      reset_stepper();
      write("stepper reset");
      init_arbitrator();
      write("arbitrator initialized");

    /* Run behaviors forever */
    while(1){

      /*Check if limit switch... highest priority behavior*/
        if(limit_switch) {
            write("resetting stepper!");
            msleep(30);
            reset_stepper();
            limit_switch=0;
        }

        if((main_counter % 5000) == 0) {
          reset_stepper();
            limit_switch=0;
        }


      if(!SPURNED) {
            write("not spurned!\n");
            /* Start playing game */
          DONE_PLAYING=0;
          spurn_counter=0;
            do{
                  write("playing game!\n");
                DONE_PLAYING=game();
                if(!DONE_PLAYING) spurn_counter++;
                if(spurn_counter > 3) {
                  DONE_PLAYING=1;
                  SPURNED=1; write("spurned has been set!\n");
                        /*INTR_OFF();*/
                        ADDR4=0x40;
                        /*Have robot speak */
                        LATCH6000 &= 0xbf;
                        ADDR6=LATCH6000;
                        msleep(2);        /*width of pulse*/
                        LATCH6000 |= 0X40;      /*bring latch back
high*/
                        ADDR6 = LATCH6000;
                        ADDR4=0x00;
                        write("survived thus far\n");
                        /*INTR_ON();                  */
                }
            }while(!DONE_PLAYING);

            write(clear);

            /*Otherwise perform collision avoidance behavior */
            write("game collision avoidance\n");
            avoid_counter=main_counter;
                  while(main_counter < avoid_counter + 50) {
                        avoid();
                }
        }

        else /*robot has been snubbed */ {
                write("robot feeling snubbed");
                spurned_start_time = main_counter;
                while((spurned_start_time + 600) > main_counter) {
                        if((left_ir > 104) || (right_ir > 104))   {
                                avoid();
                                write("avoiding stuff in snubbed behavior\n");
                        }
                        else {
                                phonophobia();
                                write("feeling phonophobic\n");
                        }
                }
                SPURNED=0;
        }

       }
    return(0);
}





void
init_arbitrator() {

        INTR_OFF();
        /* Set OC4 PSUEDO vector to stepper_ISR */

        *(unsigned char *)(0x00d6)= 0x7e;
        *(void(**)())0x00d7 = arbitrator;     /*MUST go with OC4*/


        /* Initialize globals */
        main_counter=0;
        left_ir=largest(analog(2),analog(6));
        right_ir=largest(analog(3),analog(7));
        next_speed=50;


        /* Enable OC4*/
        SET_BIT(TMSK1,0x10);

        INTR_ON();

}


void
arbitrator(void) {

    int limit_switch;

    /*clear flag*/
    CLEAR_FLAG(TFLG1,0x10);


      /*Update analog sensors*/
      left_ir=largest(analog(2),analog(6));
      right_ir=largest(analog(3),analog(7));

      /*Update motors*/
      current_speed=(next_speed+current_speed)/2;
    motor(current_speed);


      /*Update counter*/
      main_counter++;

    /* Check stepper*/
    ADDR4=0x00;     /* CDS 2 cells, reset switch */
    if(analog(1) > 80) limit_switch=1;

      return;
}

int
largest(int value1,int value2) {
    if(value1 > value2) return value1;
    else return value2;
}


void
avoid(void) {
      if((left_ir > 125) || (right_ir > 125)) {
            direction(5050);
            next_speed=-50;

      }
      else {
                  if((left_ir < 107) && (right_ir < 107)){
                        /* go mostly foward */

                  next_speed=50;
                        if(left_ir > (right_ir +3)) direction(4970);
                  else if(right_ir > (left_ir+3)) direction(5030);
                  else direction(5000);

            }

                  /*condition that theres stuff in front of him on both
sides*/

                   if((left_ir >= 114) && (right_ir >= 114)) {
                        direction(5070);
                        next_speed=-50;

                  }

                  else if((left_ir >= 107) && (right_ir >= 107)) {
                        /* chill for a moment */
                        if(STEP != 5120) next_speed=0;
                        msleep(6);
                          direction(5120);
                        next_speed=30;
                  }


                  if((left_ir >= 107) && (right_ir < 107)) {
                   /*try going right*/
                        direction(4900);
                        next_speed=30;
                  }

                  if((left_ir < 107) && (right_ir >= 107)) {
                  /*try going left*/
                  direction(5100);
                  next_speed=30;
                  }
      }
}


/*sleep routine in approximately 100ms increments*/
int
msleep(int tenths_of_seconds) {
    int temp;
    temp=main_counter;
    temp+=3*tenths_of_seconds;
    while(main_counter <= temp) { /* chilling */ }
    return(0);
}


int
game(void) {
      int low_analog4,low_analog5,current_analog4,current_analog5,i;

      /* clear low readings */
      low_analog4=low_analog5=255;


      /* stop motors */
      next_speed=0;
    direction(5000);


      write(clear);
      write("MARCO!!!!\n");

      /*Have robot speak */
      LATCH6000 &= 0xbf;
      ADDR6=LATCH6000;
      msleep(2);        /*width of pulse*/
      LATCH6000 |= 0X40;      /*bring latch back high*/
      ADDR6 = LATCH6000;

      IR=0x00;
      msleep(17);             /*wait 1sec or so until done speaking*/


      /*Turn interrupts off... absolutely necessary, or reading are too
low!?! */

       INTR_OFF();
```

```
	/*Sample*/
	for(i=0;i<10000;i++){

		current_analog4=analog(4);
		current_analog5=analog(5);


		if(current_analog4 < low_analog4){
			low_analog4=current_analog4;
	}

		if(current_analog5 < low_analog5) {
			low_analog5=current_analog5;
		}

	}

	IR=0xff;	/*Turn IR back on*/
	INTR_ON();		/*Restore interrupts*/


	/*Establish the direction the sound came from, then turn towards
it*/
	if(((low_analog5 < 110) || (low_analog4 < 110)) && ((low_analog5
>= 2) || (low_analog4 >= 2))) {

		/*Output his readings*/

		write("Right ear minimum: ");
		write_int(low_analog4);
		write("\nLeft ear minimum: ");
		write_int(low_analog5);
		write("\n");
		msleep(10);

		if(low_analog4 < low_analog5) {
			write("On my right\n");
			if(low_analog4 < (low_analog5 - 25)) {

					direction(4940);
					write("hard right");
					msleep(15);
					next_speed=50;
					msleep(4);

			}
			else if(low_analog4 < (low_analog5 - 15)) {
					write("med right");
					direction(4940);
					msleep(15);
					next_speed=50;
					msleep(4);
			}
			else if(low_analog4 < (low_analog5 - 7)) {
					write("soft right");
					direction(4940);
					msleep(15);
					next_speed=50;
					msleep(3);
			}
			else if(low_analog4 < (low_analog5 - 2)) {
```

```
                        write("v soft right");
                        direction(4970);
                        msleep(15);
                        next_speed=50;
                        msleep(3);
                  }
                  return(1);
            }

            if(low_analog5 <= low_analog4) {
                  write("On my left\n");
                  if(low_analog5 < (low_analog4 - 25)) {

                        direction(5060);
                        write("hard left");
                        msleep(15);
                        next_speed=50;
                        msleep(4);

                  }
                  else if(low_analog5 < (low_analog4 - 15)) {
                        write("med left");
                        direction(5060);
                        msleep(15);
                        next_speed=50;
                        msleep(4);
                  }
                  else if(low_analog5 < (low_analog4 - 7)) {
                        write("soft left");
                        direction(5060);
                        msleep(15);
                        next_speed=50;
                        msleep(3);
                  }
                  else if(low_analog5 < (low_analog4 - 2)) {
                        write("v soft left");
                        direction(5030);
                        msleep(15);
                        next_speed=50;
                        msleep(3);
                  }
                  return(1);
            }



      }
      else return(0);
 }

int
phonophobia(void) {
      int low_analog4,low_analog5,current_analog4,current_analog5,i;

      /* clear low readings */
      low_analog4=low_analog5=255;

      /*Sample*/
      for(i=0;i<200;i++){
            INTR_OFF();
            current_analog4=analog(4);
            current_analog5=analog(5);
                  /*Restore interrupts*/
            INTR_ON();
            if(current_analog4 < low_analog4){
                  low_analog4=current_analog4;
            }

            if(current_analog5 < low_analog5) {
                  low_analog5=current_analog5;
            }

      }



      if(((low_analog5 < 100) || (low_analog4 < 100)) && ((low_analog5
>= 2) || (low_analog4 >= 2))) {

            /*Output his readings*/
      /*
            write("Right ear minimum: ");
            write_int(low_analog4);
            write("\nLeft ear minimum: ");
            write_int(low_analog5);
            write("\n");
            msleep(10);   */


            /*same routines as phonotropism, opposite directions*/
            if(low_analog4 < low_analog5) {
                  write("on my right");
                  if(low_analog4 < (low_analog5 - 25)) {

                              direction(5060);
                              /*write("hard left");   */
                              msleep(15);
                              next_speed=50;
                              msleep(4);

                  }
                  else if(low_analog4 < (low_analog5 - 15)) {
                              /*write("med left");*/
                              direction(5060);
                              msleep(15);
                              next_speed=50;
                              msleep(4);
                  }
                  else if(low_analog4 < (low_analog5 - 7)) {
                              /*write("soft left");*/
                              direction(5060);
                              msleep(15);
                              next_speed=50;
                              msleep(3);
                  }
                  else if(low_analog4 < (low_analog5 - 2)) {
                              /*write("v soft left");*/
                              direction(5030);
                              msleep(15);
                              next_speed=50;
                              msleep(3);
                  }
                  return(1);



            }

            if(low_analog5 <= low_analog4) {
                  write("On my left\n");
                  if(low_analog4 < (low_analog4 - 25)) {

                              direction(4940);
                              /*write("hard right");  */
                              msleep(15);
                              next_speed=50;
                              msleep(4);

                  }
                  else if(low_analog5 < (low_analog4 - 15)) {
                              /*write("med right");*/
                              direction(4940);
                              msleep(15);
                              next_speed=50;
                              msleep(4);
                  }
                  else if(low_analog5 < (low_analog4 - 7)) {
                              /*write("soft right");*/
                              direction(4940);
                              msleep(15);
                              next_speed=50;
                              msleep(3);
                  }
                  else if(low_analog5 < (low_analog4 - 2)) {
                              /*write("v soft right");*/
                              direction(4970);
                              msleep(15);
                              next_speed=50;
                              msleep(3);
                  }
                  return(1);

            }




      }

      else {

            direction(5000);
            next_speed=50;
            return(0);
      }
 }
```

step.c

```
/**************** Includes *************************************/
#include <hc11.h>
#include <serial.h>
#include <mil.h>


/**************** Constants ************************************/
#define PERIODM 65,500
#define PERIOD_1PC 655
#define ADDR6 *(unsigned char *)(0x6000)

extern unsigned char LATCH6000;

#pragma interrupt_handler stepper_ISR
void stepper_ISR(void);


/* Required global varaibles */

int STEP,NEXT_STEP,SUBSTEP;



/* Stepper code initialization */
int
init_stepper() {

      INTR_OFF();

      /* Set OC3 PSUEDO vector to stepper_ISR */

      *(unsigned char *)(0x00d9)= 0x7e;
      *(void(**)())0x00da = stepper_ISR;     /*MUST go with OC3*/



      /* Clear latch and clear memory rememberance of the Latch */
       ADDR6=0;
       LATCH6000=0;

      /* Initialize globals */

       STEP=5000;
     NEXT_STEP=5000;
     SUBSTEP=0;

      /* Enable motor interrupts on OC3*/
      SET_BIT(TMSK1,0x20);

      INTR_ON();

}


/*      This next function counts on the analog subsystem to be enabled
        It moves the stepper in the positive (CCW) direction until it
senses the bump
        switch activation. Then it moves the stepper back to the zero
position, calling
```

```
        5000 zeroed out.
*/


int
reset_stepper() {

    int bump_switch;

    do {
        bump_switch=analog(1);
        /*write_int(bump_switch);*/
        NEXT_STEP=STEP+2;
        while(STEP!=NEXT_STEP){} /*wait for hardware to catch up....*/
    }while(bump_switch > 80)

    INTR_OFF();
    STEP=5135;
    NEXT_STEP=5000;
    INTR_ON();

    while(STEP!=NEXT_STEP){} /*wait for hardware to catch up....*/

    return(0);
}




/* Stepper ISR */

void
stepper_ISR(void) {

      TOC3 += 10000;  /*next time*/

    /* Clear oc3 flag */

    CLEAR_FLAG(TFLG1,0x20)

      if(NEXT_STEP == STEP) {
        LATCH6000 &= 0xf0;   /*leave motor the same but kill stepper
driver stuff*/
        ADDR6=LATCH6000;
    }

    if(NEXT_STEP < STEP) { /* going clockwise */

            /* Do SUBSTEPping */
        LATCH6000 &= 0xf0;  /*Clear only stepping stuff*/

        if(SUBSTEP==1) {

            LATCH6000 |= 2;
            ADDR6=LATCH6000;

        }
        if(SUBSTEP==2) {

            LATCH6000 |= 4;
            ADDR6=LATCH6000;
        }
        if(SUBSTEP==3) {

            LATCH6000 |= 1;
            ADDR6=LATCH6000;

        }
        if(SUBSTEP>=4) {

            LATCH6000 |= 8;
            ADDR6=LATCH6000;
            STEP--;
            SUBSTEP=0;
        }

        SUBSTEP++;
  }

     if(NEXT_STEP > STEP) { /* going counterclockwise */

            /* Do SUBSTEPping */
        LATCH6000 &= 0xf0;  /*Clear only stepping stuff*/

        if(SUBSTEP==1) {

            LATCH6000 |= 8;
            ADDR6=LATCH6000;

        }
        if(SUBSTEP==2) {

            LATCH6000 |= 1;
            ADDR6=LATCH6000;

        }
        if(SUBSTEP==3) {

            LATCH6000 |= 4;
            ADDR6=LATCH6000;

        }
        if(SUBSTEP>=4) {

            LATCH6000 |= 2;
            ADDR6=LATCH6000;
            STEP++;
            SUBSTEP=0;
        }

        SUBSTEP++;
  }

}


int
direction(int step) {
    NEXT_STEP=step;
    return 0;
}
```

motor.c

```
/* Marco’s Motor Driver Routines
   based loosely on TALRIK routines
   and modified by Drew Bagnell
*/

/**************** Includes *************************************/
#include <hc11.h>
#include <serial.h>
#include <mil.h>

/**************** Constants ************************************/
#define PERIODM 65,500
#define PERIOD_1PC 655
#define ADDR6 *(unsigned char *)(0x6000)

#pragma interrupt_handler motor0
void motor0();


/********************* Data ************************************/
int duty_cycle;  /* Specifies the PWM duty cycle for two motors */
unsigned char LATCH6000;

/************** Functions ***************************************/
void init_motors(void)
/* Function: This routine initializes the motors
 * Inputs:   None
 * Outputs:  None
 * Notes:    This routine MUST be called to enable motor operation!
 */

{

  INTR_OFF();

 /* Clear latch and clear memory rememberance of the Latch */
    ADDR6=0;
    LATCH6000=0;


/* Set OC2 and OC3 to output low */
/*  SET_BIT(TCTL1,0xA0);
  CLEAR_BIT(TCTL1,0x50);*/

/* Set PWM duty cycle to 0 first */
   duty_cycle = 0;

/* Associate interrupt PSEUDO-vectors with motor routines */
  /* *(void(**)())0xFFE6 = motor0;    */
   *(unsigned char *)(0x00dc)= 0x7e;     /*MUST go with OC2*/
   *(void(**)())0x00dd = motor0;     /*MUST go with OC2*/


/* Enable motor interrupts on OC2*/
```

```
  SET_BIT(TMSK1,0x40);


/* Specify PD4 and PD5 as output pins.
 * PD4 controls direction of Motor 1 and PD5 the direction of Motor 0.
 */

  /*SET_BIT(DDRD,0x30);*/
  INTR_ON();
}

void motor(int per_cent_duty_cycle)
/* Function: Sets duty cycle and direction of motor specified by index
 * Inputs:   index in [0,1]
 *         -100% <= per_cent_duty_cycle <= 100%
 *         A negative % reverses the motor direction
 * Outputs:  duty_cycle[index]
 *           0 <= duty_cycle[index]<= PERIOD (Typically, PERIOD =
65,500)
 * Notes:    Checks for proper input bounds
 */
{
  if (per_cent_duty_cycle < 0)
     {
        per_cent_duty_cycle = -per_cent_duty_cycle; /* Make positive */
     /* Set negative direction of motors */
     /* Bit 5 is direction bit */
      LATCH6000 |= 0x10;
      ADDR6     =LATCH6000;
     }
  else
     {
     /* Set positive direction of motors */

      LATCH6000 &= 0xef;
      ADDR6   = LATCH6000;

     }

/* At this point per_cent_duty_cycle must be a positive number less
 * than 100. If not make it so.
 */
  if (per_cent_duty_cycle > 100) per_cent_duty_cycle = 100;
  duty_cycle = per_cent_duty_cycle*PERIOD_1PC;

}

void motor0 ()
/* Function: This interrupt routine controls the PWM to motor0 using OC2
 * Inputs:   duty_cycle[0] (global)
 * Outputs:  Side effects on TCTL1, TOC2, TFLG1.
 * Notes:    init_motors() assumed to have executed
 */

{
/* Keep the motor off if no duty cycle specified.*/

  if(duty_cycle == 0)
   {
    CLEAR_BIT(LATCH6000,0x20);
    ADDR6 = LATCH6000   ;
}
  else
    if(LATCH6000 & 0x20)
     {
        TOC2 += duty_cycle;             /* Keep up for width */
                                    /*  Set to turn off */
      CLEAR_BIT(LATCH6000,0x20);      /*  Set to turn off */
        ADDR6=LATCH6000;
     }
    else
     {
            TOC2 += (PERIODM - duty_cycle);
                    /* Set to raise signal */
            SET_BIT(LATCH6000,0x20);
      ADDR6=LATCH6000;
     }
  CLEAR_FLAG(TFLG1,0x40);                         /* Clear OC2F
interrupt Flag */
}
```