Sound-Seeking Robot

An E155 Design Project at
Harvey Mudd College (Claremont, California)
December 15, 2003
Alex Utter
Chris Wottawa

Abstract

A sound-seeking-robot was designed to navigate towards a specific alternating tone. The robot utilized analog signal processing, a PIC microcontroller, and an FPGA to locate and move towards its goal. The resulting robot was unable to reliably accomlish this task. The robot was able to identify the signal and locate it along a line, but was plagued by mechanical problems and frequently moved in precisely the opposite of the intended direction.

Table of Contents

Summary of Figures and Tables

1. Introduction

1.1. Motivation and Applications

Our final project was a sound-seeking mobile robot capable of locating and moving towards a specific sound signal. This project was chosen because it used many of the aspects of robotic navigation and electromechanical design applicable to more complicated robotic systems, but was still simple enough to complete in four weeks.

A sound seeking robot could be used in many applications, but more importantly, the sound-seeking-robot allows experimentation and testing on navigational algorithms that could be used as part of a more sophisticated, more practical robot.

1.2. Objectives

The completed robot should be able to:

- Identify a specific sound signal and find the direction to the source
- Activate motors to rotate the robot and move towards the source
- Stop movement on impact with an obstruction

1.3. Design Overview

The sound seeking robot contained a number of interconnected subsystems, as shown in figure 1.

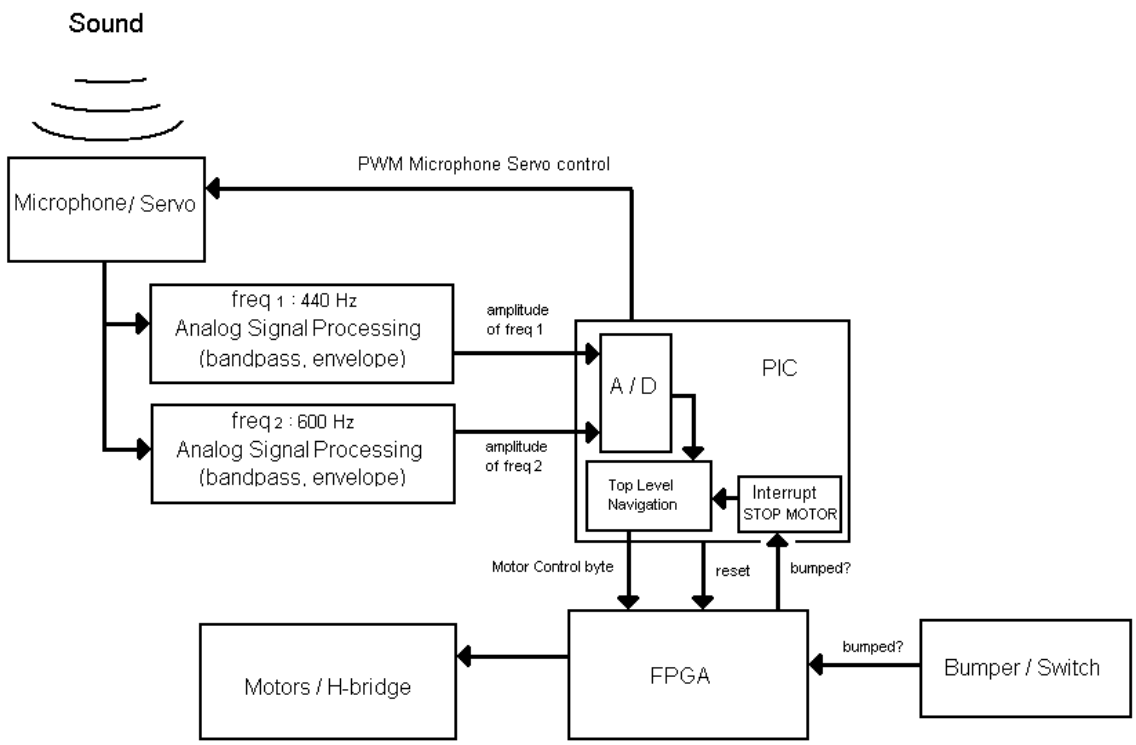


Figure 1: Block Diagram for Sound-seeking Robot

The sound that the robot sought was a signal that alternated between a 440 Hz tone and a 600 Hz tone at a rate of 4 Hz. This tone was chosen because a signal that alternates between two pure tones is easily processed and unlikely to be found from any undesired noise source. Therefore it would be easy to identify even in a relatively noisy environment. The robot used a directional microphone to detect this signal and determine a bearing to its source. This microphone was mounted on a servomotor which rotated in a search pattern.

Analog band-pass filters with envelope detectors processed the microphone signal, and the PIC microcontroller sampled these envelopes to determine whether the desired sound signal has been found.

For locomotion, the robot used two drive wheels with one motor each in a tank-style configuration, plus a caster wheel for support. These motors were driven by the FPGA through an H-bridge for reversible control. As the robot moved, it continued to use the microphone to get updated, more-accurate bearings on the source, adjusting its path accordingly. If it collided with an obstruction in its path, it simply stopped movement altogether. Collisions were detected using a front bumper that depressed a switch.

1.4. Digital Components and Functions

PIC Functions:

- A/D Conversion of sound signal envelopes (envelopes found using analog electronics)
- Sound signal identification
- PWM control of microphone-pointing servo
- Top level navigation logic
- Output status and A/D information

FPGA Functions:

- Motor control logic
- Bumper switch debounce and collision detection logic

2. New Hardware

In addition to the PIC Microcontroller and the Xilinx FPGA, additional hardware was needed to complete the sound-seeking robot.

2.1 Microphone

The microphone used was a standard consumer microphone for home audio use, and therefore included most of the circuitry necessary to output the sound signal. Before acquiring this microphone, we attempted to use a cartiridge microphone, which did not include the appropriate power and signal separation circuits. Despite many attempts to recreate the circuit shown in the cartridge microphone's datasheet, we were unable to ever detect any sort of useful signal; we believe that the microphones obtained from the stockroom may have "expired", having been left unused for so long that they lost the internal electrical charges necessary for proper functionality of electret microphones.

The commercially purchased microphone worked flawlessly from the moment we connected it to our processing circuit. To increase the directionality of our microphone to desired levels, a long paper tube was

2.2 Servomotor

For rapid and accurate pointing of the microphone independent of the movement of the robot, a servomotor was added. Standard hobby servomotors require power and a pulse-width-modulated signal that corresponds to a specific armature position. Once given a PWM command, a servomotor attempts to rotate to the specified angle and maintains that angle as long as the command is maintained. This robot's servomotor had a PWM period of 30 ms and a rotation range of about 180˚; periods corresponding to specific angles were determined empirically.

2.3 Robot Chassis

A photo of the robot is shown in figure 2. Additional images of the finished robot chassis is shown in Appendix A. The motors were connected to angle brackets with screws, through cuts in the angle brackets that allow easy adjustment of the motor's vertical position. The holding area of the robot had two floors. Three batteries were mounted on the lower floor. The analog signal processors, PIC, and FPGA were all mounted onto a Protoboard which rested securely on the top floor of the robot's chassis. The microphone and servomotor setup at the front of the robot allowed it to quickly point the microphone in different directions in order to best determine the location of the sound source. In front of the microphone and servo was a bumper; the front bumper needed to extend past the end of the microphone and past the wheels. When the robot came into contact with an object, the bumper closed a switch which was connected to the FPGA.

Figure 2: Robot chassis

2.4 Motor / Wheel Configuration

The motor gears are positioned directly against the wheels so that friction between the gear and the wheel moves the robot forward or backward, as shown in figure 3. The front two wheels are driven with motors and the back wheel is a free moving caster. For the robot to turn, one wheel needs to be driven forward and the other backwards. To move forward, both wheels are driven forward.

2.5 H-Bridge

In order to drive the large currents needed by the motors using the the small currents suppliable by the FPGA, the robot uses a device known as an H-bridge. The H-bridge allows for logic-level control of large currents and voltages, and is configured to drive the motor both forward and in reverse. The H-bridge used, the L293DNE, was a dual H-bridge with internal diodes that limit voltages developed by the motor that could damage the PIC and FPGA.

3. Schematics

Detailed electrical schematics are available in Appendix B. Descriptions of the circuits diagrammed follow:

3.1 Analog Signal Processing

The microphone output was fed into an amplifier which acted as a voltage follower and had a gain of 30. (The voltage follower was needed to supplement the microphone's very low output currents.) The amplified output was fed into a parallel pair of band pass filters with pass frequencies of 440 and 600 Hz and bandwidths of ± 20Hz, to effectively screen out background noise while still allowing adequate signal change rates. In series with each filter was a precision full wave rectifier and low-pass filter with time constant of 10 ms, to sense the signal envelope in each band. The output from each envelope detector is periodically sampled by the PIC's built-in A/D converters.

3.2 Power Supply

The sound-following robot had a large number of digital, analog, and mechanical components, each with

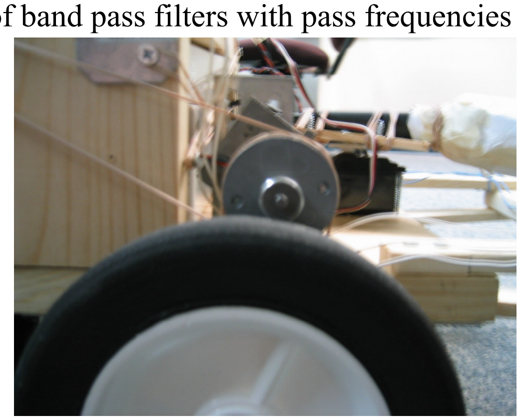


Figure 3: Motor drive system

unique voltage and current requirements. As a result, the power systems of the robot were very complex, requiring three separate batteries and five separate power buses.

The utility board and analog processing circuits were powered by a 12V battery. The utility board operated between 0 and +5V, maintained by an L7805CV voltage regulator. The analog signal processing circuits used operational amplifiers that normally require a ground, positive supply, and negative supply. Unfortunately, the analog outputs were required to operate in the same voltage range as the utility board, to allow the PIC's A/D converters to operate properly. As such, all analog signals used a "virtual ground" as the zero signal, which was maintained at +5V relative to the battery's ground. The op-amps then received 0 and +10V for their negative and positive supplies, respectively; the +10V power wass supplied by a pair of L7805CV regulators connected in series. To put the output in the right range, the analog envelope finders were operated in reverse, outputing the virtual ground of +5V for zero signal, with output voltage dropping for a stronger signal. All devices operated on the 0, +5V, and +10V supplies were sensitive to ripple, so all supply voltages were linked capacitively to maintain constant relative voltages in the face of rapidly fluctuating current loads.

A second, 6V battery supplied the microphone-pointing servomotor and a cooling fan. Originally, the servomotor was powered off of the +5V supply for the utility board, but the motor drew so much current that the utility board would frequently brown-out due to voltage sag, so it was moved to a separate supply. No regulation was needed because the motor was not sensitive to small voltage fluctuations.

Finally, the main drive motors were driven by a separate 12V battery. When running, the motors usually drew several amps of current, which would cause unacceptable voltage sag in the other 12V battery. These motors were originally powered by the 6V battery, but as the robot's weight increased, more power was needed to overcome static friction. Increasing the motor supply voltage to 12V battery was adequate to drive the motors.

3.3 PIC / FPGA interface

The division of tasks between the FPGA and the PIC required three signals connecting the two chips. The first was the motor control byte, which was sent serially over the PIC's SPI interface. The SPI interface consisted of serial clock and serial output signals which were connected to an appropriate decoder on the FPGA, as well as a serial input signal which was connected to ground, since there was no need for corresponding FPGA to PIC messaging.

The second signal was the bumped-flag, which was triggered and held high whenever the robot ran into an obstacle and depressed the bumper switches. Whenever the FPGA detected a bumper press, it would stop the drive motors and set the flag high until reset. This signal triggered a high-priority interrupt in the PIC, which stopped all other motors and sent the PIC into sleep mode until it was reset. The PIC's initialization routine triggered a reset in the FPGA (thus resetting the bumped flag); the reset wire is the third PIC to FPGA signal.

4. Microcontroller Design

The functions of the PIC microcontroller were:

- A/D Conversion of sound signal envelopes (envelopes found using analog electronics)
- Sound signal identification
- PWM control of microphone-pointing servo
- Top level navigation logic
- Output robot status and A/D information to LED indicators

The PIC assembly code was written in one module, called "navigation.asm", given in Appendix D. This PIC code does the following:

1) Outputs a PWM signal to the microphone-pointing servomotor

2) Reads the two Analog inputs and a reference voltage, and does A/D conversions
3) For each quarter second set of A/D results, computes TempResult (see below)
4) Uses TempResult score to send one of four possible motor control bytes to the FPGA. These control bytes consist of two signed four bit integers (-7 to +7) which correspond to the forward and backward speed of the left and right motors. The left motor's speed is specified by the first four bits, and the right motor's speed is specified by the last four bits. The most important control bytes are: 0x97 (turn left), 0x79 (turn right), 0x77 (go straight), or 0x00 (stop).
5) Receives the "bumped" input from the FPGA. If high, this triggers an interrupt. The PIC outputs a control byte to the FPGA that says the robot should stop moving, and then goes into sleep mode. RC6 is pulled high, turning on the red indicator light.

The following table gives the final pinout for the PIC microcontroller.

Table 1: PIC Microcontroller Pinout

Signal	From / To	I/O	Pin Used
440Hz Analog Input	Analog Signal Processor (440Hz)	Input	AN0 / RA0
600 Hz Analog Input	Analog Signal Processor (600Hz)	Input	AN1 / RA1
A/D Voltage Reference	Voltage Reference Circuit	Input	AN3 / RA3
Motor Control Byte	FPGA	Output	SPO / RC5
Serial Clock	FPGA	Output	SCLK / RC3
Bumped?	FPGA	Input	INT0 / RB0
Pulse	Microphone Servomotor	Output	CCP1 / RC2
Bumped Indicator	Red LED	Output	RC6
Normal Operation	Green LED	Output	RC7
Turning/Going Straight	Green LED	Output	RA5
Temp Result [9]	Yellow LED	Output	RB2
Temp Result [8]	Yellow LED	Output	RB3
Temp Result [7]	Yellow LED	Output	RB4
Temp Result [6]	Yellow LED	Output	RB5
Reset FPGA	FPGA	Output	RB1

Navigation was accomplished by a sort of state machine, as shown in figure 4. The robot starts in the "turn left" state. It continually turned left until the microphone detects a decrease in volume, indicating that it had found the sound source and turned slightly too far. (The "turn right" state functioned in the same manner.) The robot then went to the "move forward" state, and periodically scans regions directly ahead, slightly to the left, and slightly to the right. It continued to move forward until either the right or left sectors showed a higher signal amplitude, in which case it began turning in the direction of the strongest signal.

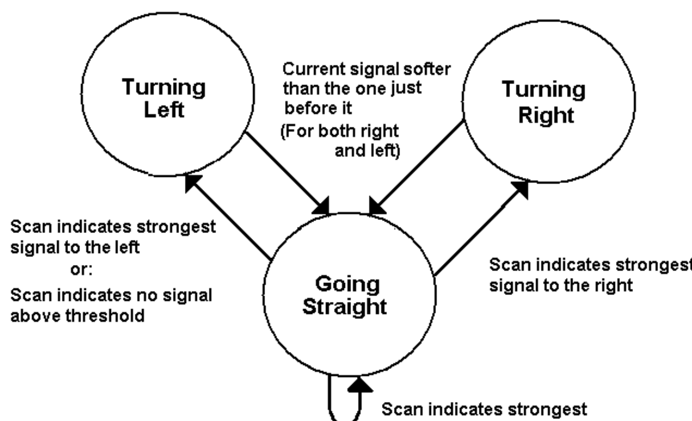


Figure 4 Navigation Algorithm

Movement was performed between microphone sampling times, to reduce the noise caused by motor vibrations. Input consisted of the two analog signal amplitudes (which started at 5.0V for no signal and dropped to as low as about 3.0 V for a very strong sound signal), and a "bumped" pin that was pulled high if the robot collided with an obstacle (indicating that the robot should stop as quickly as possible.) Output consisted of an SPI link for sending a motor control byte (MCB), a green LED that flashed to indicate normal operation, a green LED that turned on during forward motion, four yellow LEDS that represented the four most significant bits of the TempResult and a red LED that turned on if an emergency stop was initiated. The motor control byte consisted of two, signed 4-bit numbers (-8 to 7), corresponding to the speed of the left and right motors. The maximum forward speed was +7, and -7 was the maximum reverse speed; -8 is an error code.

To find the strongest signal, for every quarter second set of A/D samples, the largest 440Hz value and its corresponding 600Hz value were stored as well as the largest 600Hz value and its corresponding 440Hz value. The strongest signal was defined as the signal that has the largest result to the following equation:

$$\text{TempResult} = \left(MAX_{440} + MAX_{600}\right) - \left(COR_{600} + COR_{440}\right)$$

This way, the robot was more sensitive to the alternating tone and not as sensitive to other noises with high 440 Hz or high 600 Hz components.

In addition, if none of the TempResults were above a configurable threshold, the robot turned in counterclockwise circles until it found a strong enough signal. The pulse widths and corresponding motor control bytes were stored in a table so that it would be easier to add and adjust more microphone sample directions as needed.

5. FPGA Design

The field programmable gate array (FPGA) translated motor control information into pulse-width-modulated commands for the H-bridge, and debounced the bumper switches. The pinout of the FPGA is shown in Table 2.

Table 2: FPGA Pinout

Signal	From / To	I/O	Pin Used
Bumped	PIC	Output	P77
motorOut[3]	H-Bridge	Output	P40
motorOut[2]	H-Bridge	Output	P39
motorOut[1]	H-Bridge	Output	P38
motorOut[0]	H-Bridge	Output	P37
leds[7]	LED Array	Output	P10
leds[6]	LED Array	Output	P9
leds[5]	LED Array	Output	P8
leds[4]	LED Array	Output	P7
leds[3]	LED Array	Output	P6
leds[2]	LED Array	Output	P5
leds[1]	LED Array	Output	P4
leds[0]	LED Array	Output	P3
serIn	PIC	Input	P82
serClk	PIC	Input	P35
reset	PIC	Input	P62
clk	Clock	Input	P13
bumperSw[2]	Bumper switch	Input	P46
bumperSw[1]	Bumper switch	Input	P45
bumperSw[0]	Bumper switch	Input	P44

The motor control byte was periodically sent over the PIC's SPI interface, and received by a shift-register on the FPGA. This shift register counted incoming bits and updated a separate output register each time a complete byte was received. This output register allowed other modules to receive a clean and consistent motor-control-byte even during the transmission of the subsequent control byte.

The motor control byte consisted of a pair of four bit, two's complement binary numbers. For each number, the range of -7 to +7 corresponded to requests for speeds between full speed reverse and full speed forward, with 0 requesting a stop. The remaining possible input, -8, was reserved as a possible error code but was unused in this version of the robot. The four most significant bits controlled the left motor, and the four least significant bits controlled the right motor.

These speed requests were decoded into a corresponding PWM signal for the H-bridge, with a 14 ms cycle period broken into 7 segments of 2 ms each. The motors were activated in the requested direction for a number of segments equal to the requested speed, and stopped for the rest. Later experimentation showed that these speeds were from from linear, and that the motors would fail to start at all but the highest speed settings. Future versions of the robot might send a full speed signal to start the motors and drop the speed for more precise navigation, but these features were not needed in this version of the robot.

Bumper switch debouncing was accomplished by sampling the switch output on every cycle. If any switch was depressed for two subsequent cycles (1 microsecond), then the bumped flag was pulled high. This flag signaled the PIC to stop all motion, and also caused the FPGA to stop the drive motors without waiting for a stop command from the PIC; a faster stop means less likelihood of damage due to a collision.

Verilog for the modules responsible for these operations is given in Appendix C.

6. Results

6.1. Testing Procedure

The sound source was generated and saved in a .wav file. It was made using Matlab, and consists of a tone that alternates between 440 Hz and 600 Hz, cycling at a frequency of 4Hz. (The code for generating signals of this type is included in Appendix E.) This sound was played back through a personal computer. Most tests took place in the Microprocessors lab. Several tests were run with the robot positioned at different locations from one to eight feet away from the sound source. For each location, several tests were run with the robot initially facing in different directions relative to the source. Tests were completed successfully if the robot bumped into the sound source, or unsuccessfully if the robot bumped into another object or strayed too far away from the sound source (nine or more feet away).

6.2. Testing Results

The testing showed that the robot usually navigated to a location near the sound source, and occasionally navigated well enough to bump into the sound source, but this behavior was not very reliable. In most cases, once the robot was close to the sound and facing towards it, having acquired a solid "lock" on the sound's bearing, it moved towards the sound accurately enough to successfully contact the speaker. Ranges under one or two feet worked the most reliably. Occasionally, the robot "missed" and overshot the speaker, ending up in a sort of "orbit" around the sound source, but this problem was not as significant as many others.

Occasionally, the robot read an incorrect bearing to the sound source. The robot rarely went in a direction perpendicular to the sound source, instead favoring going directly towards or directly away. This behavior was the result of some combination of undesired microphone sensitivity, flaws in the robot's turning algorithm, and sound reflections. For a given source position, the microphone was most "sensitive" to the sound when facing either directly away or directly towards the sound. (See figure 5, where green represents weak signals and red represents strong signals.) This rearward sensitivity could have been due to a problem with our "unidirectional" microphone or with detection of reflections off of nearby walls.

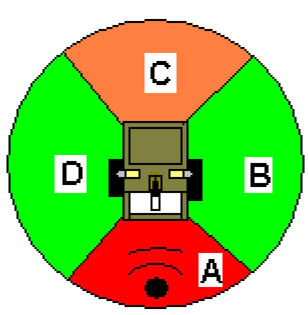


Figure 5: Tone intensities

In any case, the sensitivity had an adverse effect on the robot's behavior. When the robot was facing straight ahead, and had locked on to the signal, it continued towards it as much as the sticky left wheel allowed. However, if the sound source was not directly in front of the robot, problems arose. Suppose the robot is currently facing section B in the figure, such that the tone is just outside of the microphone's range to the right and the robot could not yet hear anything. According to the algorithm, this means that the robot should turn left. It would scan through section C, and realize that the strength of the signal has increased, though it would not stop turning until the signal intensity began to decrease. This would happen when the robot turns through the border between sections C and D. The robot would then think that the signal is in section C, so it would turn back one step and go straight ahead, directly away from the source. The robot effectively moves towards the local maxima in signal intensity, unaware of the stronger signal behind it.

Another significant problem involved the drive system. The motors usually maintained adequate traction with the wheels to accurately turn left or right. When attempting to move straight, however, the left motor had trouble breaking the static friction of the left wheel on the ground, so that the robot would veer to the left instead of moving forward. The inconsistency between the motor commands and the robot's behavior frequently caused the robot to become confused. Once confused, the robot rarely re-acquired its lock on the sound.

6.3. Future Improvements

There are a number of improvements that could be made to simplify the current robot design as well as improve the speed and reliability of its performance. The first obvious area in need of improvement is the motor mounting design. Even if the robot could perfectly sense the alternating tone and ignore arbitrary amounts of noise, if the wheels do not respond appropriately, the robot is not going to reliably succeed in its task. While this current motor design is very simple to build and works well under some applications, it would be better to use a consistently reliable and precise design. One such idea is to use a gear box to increase the motor output torque, and mount the output shaft directly to the axle of the wheels. This design would eliminate slippage and would probably greatly boost the motor's efficiency. The motors in this design were chosen because of their availability and price (Engineering stock room and free), with little regard for their appropriateness. It is very possible that there are more expensive motors which may be more difficult to find that are more suitable for this application.

Problems with rearward sensitivity to the signal (due to problems with the microphone and sound reflections) could be resolved by mounting an additional microphone, pointing opposite the direction of the original microphone. This rear-pointing microphone could then detect the stronger signal, indicating that the forward signal is not the correct one to follow. Unfortunately, this microphone would require an additional complete set of analog processing filters, making it very useful but impractical.

Another problem with the robot is its sensitivity to sounds which are not the intended signal. Stomping, clapping, vibrations, and even talking will often fool the robot into thinking that it has detected the alternating tone signal. This phenomenon is at least partly due to our processing system in the PIC, which uses only the maximum detected 440 Hz signal, the maximum of the 600 Hz signal, and the corresponding opposite signals. Since it does not take into account any of the other hundreds or thousands of data points at all, it discards a great deal of useful data. In addition, the "maximum" signal may frequently be due to a simple glitch or one-time spike, rendering the system very susceptible to random noise. The fact that the algorithm effectively uses the derivative of these maximums for its turning algorithm makes matters worse, since derivatives inherently accentuate high-frequency noise.

One way to solve these problems is to use more sophisticated processing algorithms in software. One such algorithm takes the mean of the data recorded over a quarter second, then averages values above and below this mean for each frequency band, computing the ranking score from these averages instead of the maximum values. This method would be complicated by the PIC's lack of a divide command, which makes it impossible to take the average of an variable quantity of values.

A simpler method might be to enhance the robot's analog signal processing. The current circuitry does not take advantage of the fact that the signal alternates frequencies at a rate of 4 Hz. Rather than sampling the output of the envelope detectors, an improved robot might include a 4 Hz band pass of the difference between the two envelope outputs, and then find the envelope of this signal. Any noise except our signal is very, very unlikely to alternate between 440 and 600 Hz at a rate of 4 Hz, so this filter would block nearly everything except the desired signal. With this filter in place, the PIC would only need to sample a single output signal which directly corresponds to the intensity of the desired signal.

In addition to the problems listed, a more useful robot of this type would need to be able to navigate through environments more complex than a large, flat, open room. Once navigation towards a sound is adequately refined, an important upgrade to the robot would include the ability to navigate around simple obstacles.

7. References

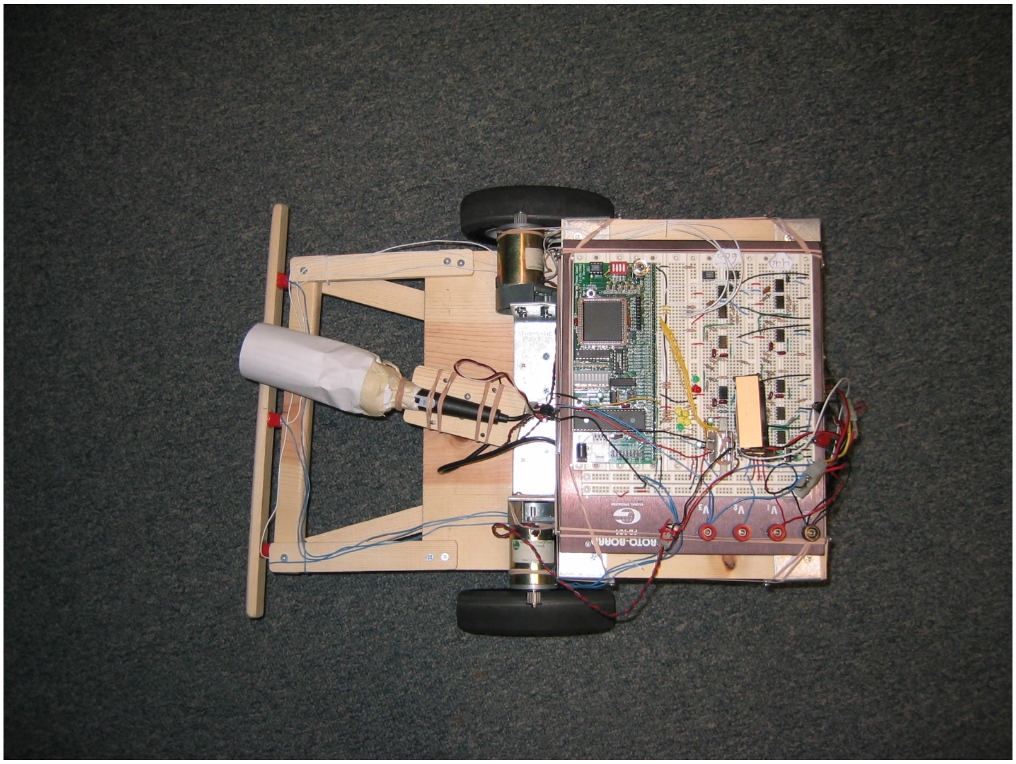
- Brown, Jim. "Brief H-Bridge Theory of Operation." 1998 April. http://www.dprg.org/tutorials/1998-04a/
- Doshi, Rishin. "Friction Drive Analysis". 1996 Spring. http://maeweb.ucsd.edu/~mae3/spring2002/mae3_17/web/Rishin/frictionanalysis.htm
- Fortner. "Laboratory Electronics 11 15.2". Date unknown. http://niufrm.physics.niu.edu/~labelec/lect/p475_lect152.pdf
- Mishra, Rohit and Wang, Mark. "Musical Instrument Tuner". 1999-12-9. http://odin.ac.hmc.edu/~harris/class/e155/projects99/instrumenttuner.pdf
- National Semiconductor. "LM741 Operational Amplifier". 2000 August. http://www.national.com/ds.cgi/LM/LM741.pdf
- Purdie, Ian C. "Active Bandpass Filters". 2002-1-20. http://www.electronics-tutorials.com/filters/active-bandpass-filters.htm
- ST Micro. "L293D Push-Pull Four Channel Driver with Diodes". 2003 July. http://www.st.com/stonline/books/pdf/docs/1330.pdf
- ST Micro. "L7800 Series Positive Voltage Regulators". 2003 November. http://www.stmicroelectronics.com/stonline/books/pdf/docs/2143.pdf

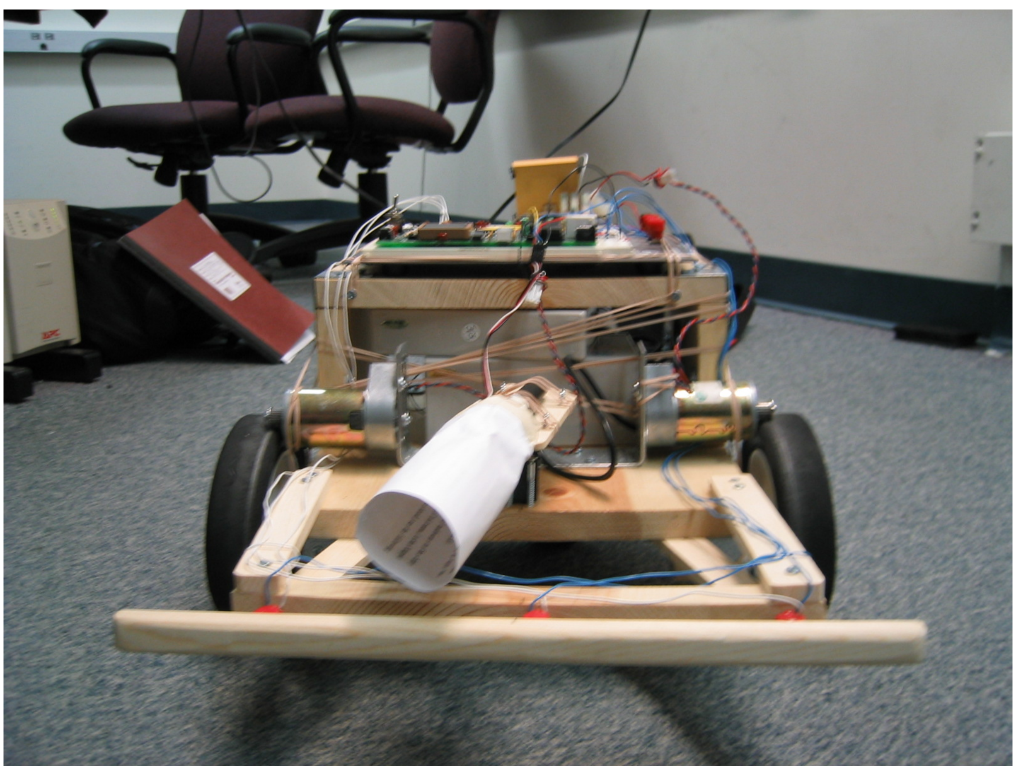
8. Parts List

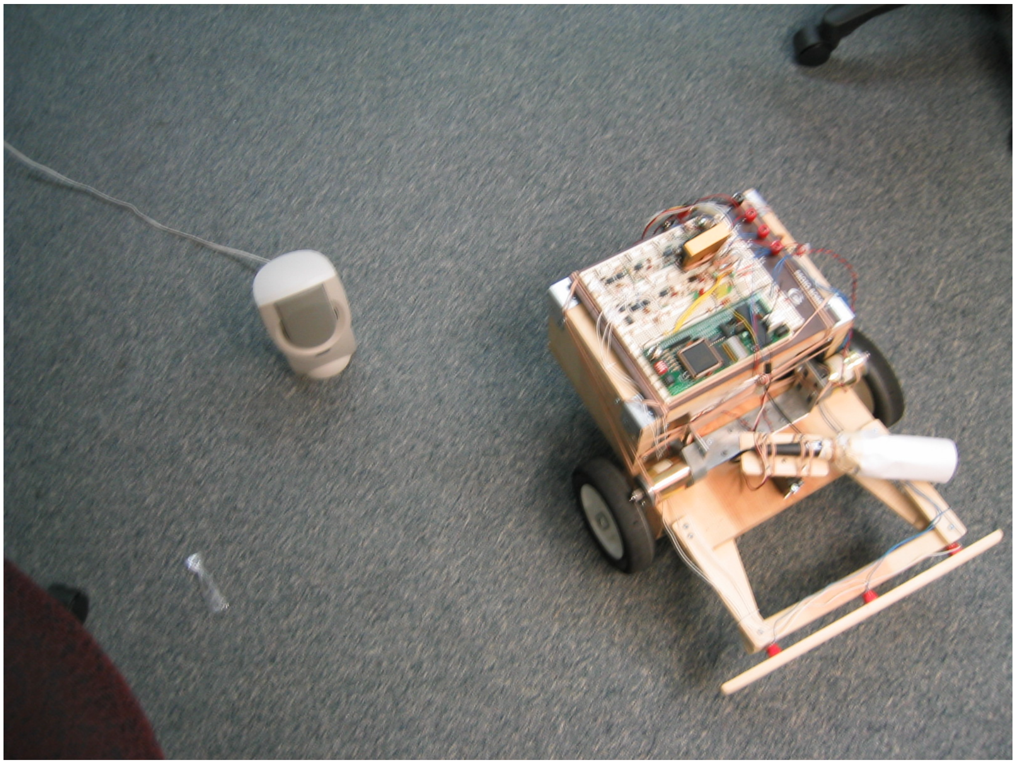
Table 3. Additional Parts and Budget

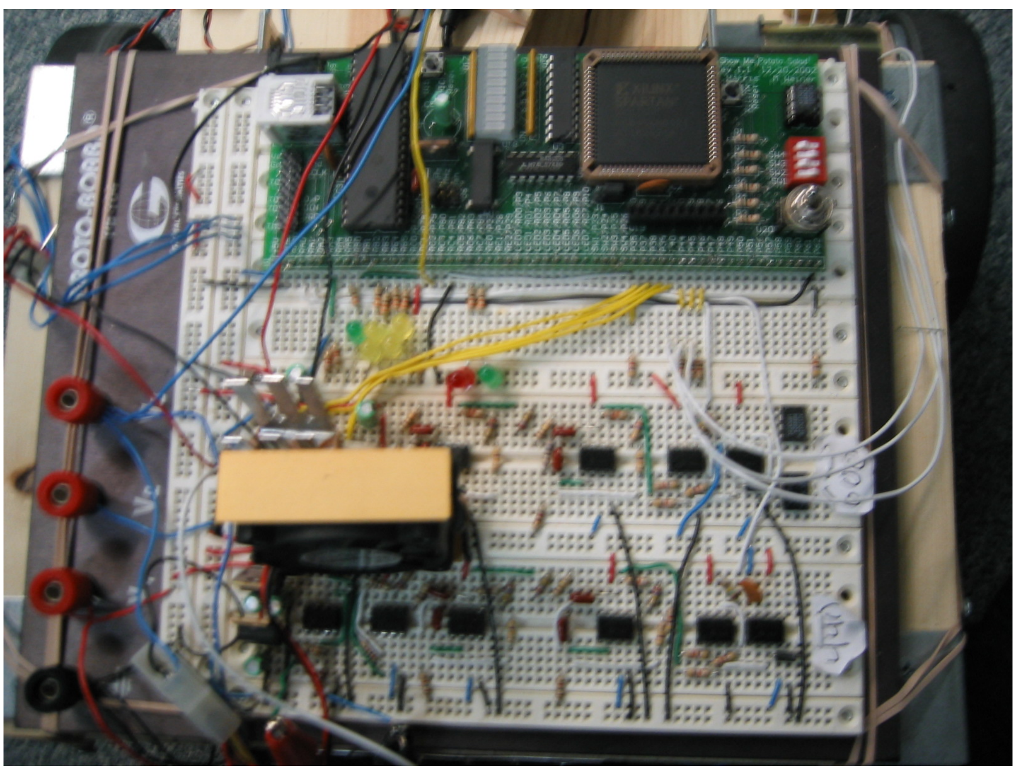
Part	Source	Vendor Part #	Price
Wood	East Dorm, HMC	----	----
Microphone	Radio Shack	----	$9.99
Servomotor	----	Hobbico CS-51	----
Cooling Fan	----	Topower	----
12V Battery	Radio Shack	Eveready	----
6V Rechargeable Battery	MarVac Electronics	PS-670	$15.75
12V Rechargeable Battery	MarVac Electronics	PS-1270F1	$28.50
6" Wheels (x2)	Home Depot	----	$9.92
Caster	Home Depot	----	$4.97
Angle Brackets	Home Depot	----	$6.63
Screws, Washers, Nuts, Bolts	Home Depot	----	$6.54
Motors (x2)	Stock	Bühler	----
Switches (x3)	Stock	----	----
Rubber Bands	----	----	----
Operational Amplifiers (x10)	Stock	HA17741	----
H-Bridge	DigiKey	L293DNE	----
Diodes (x2)	Stock	1N4002GP	----
Total Cost			$82.30

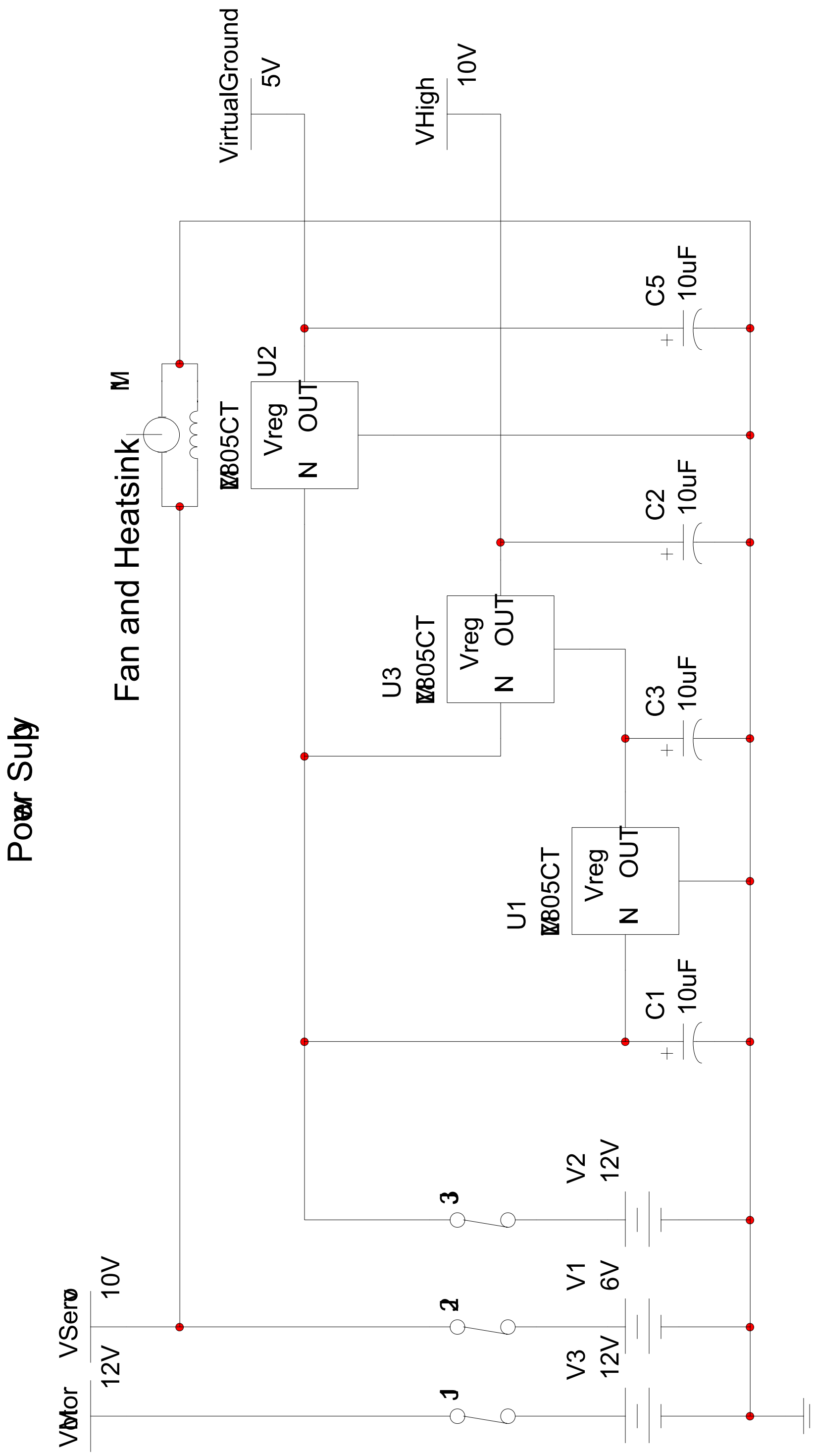
Appendix A: Robot Pictures



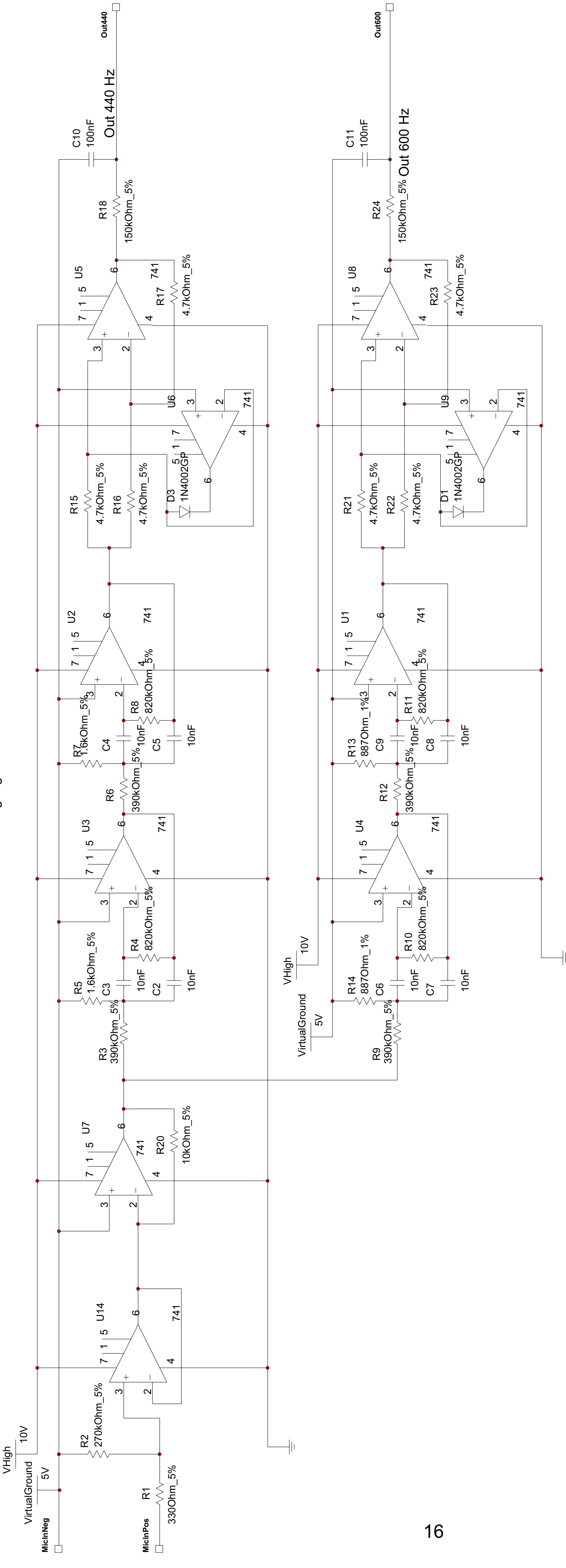




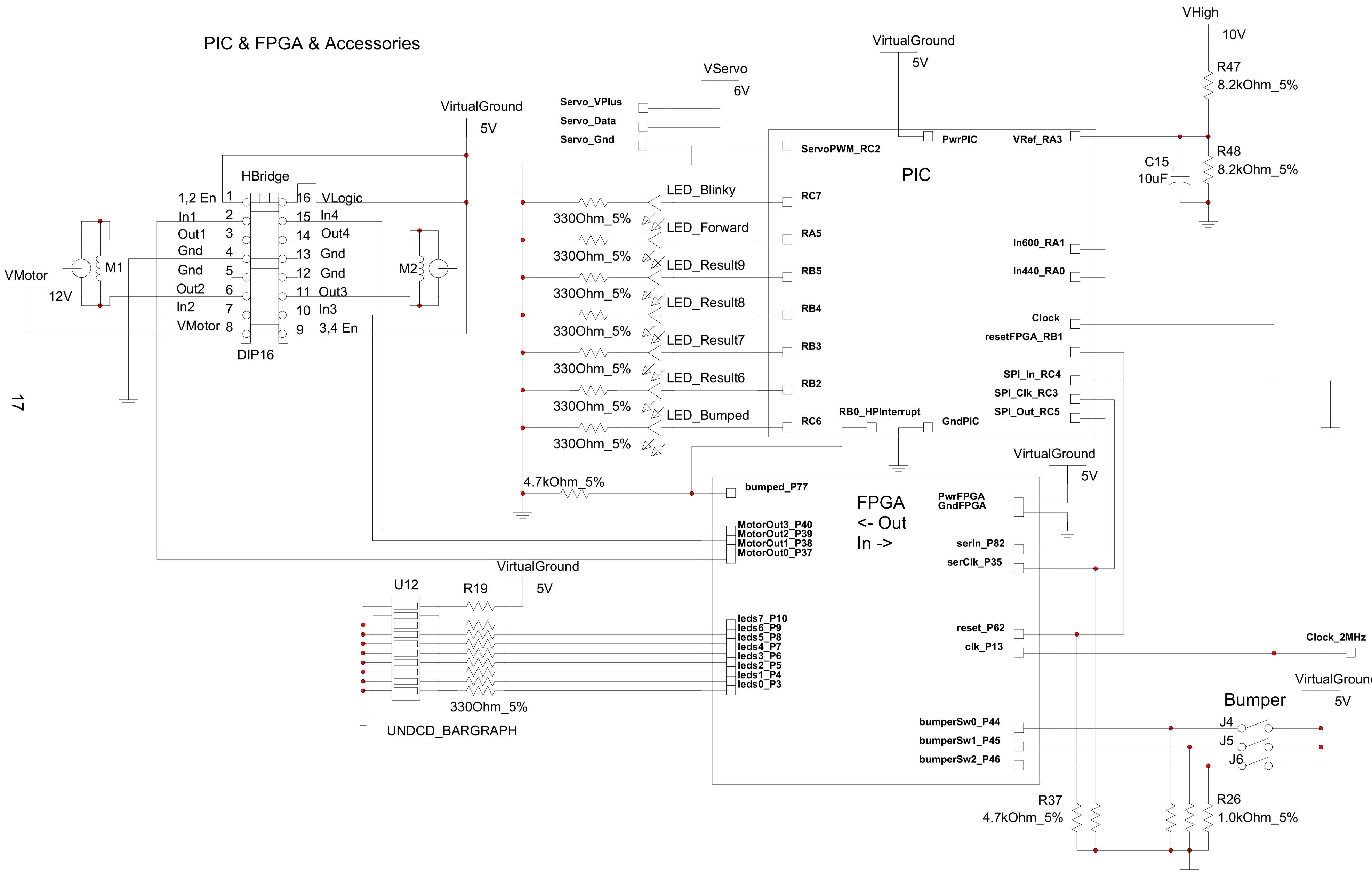



VirtualGround
5V
VHigh
10V
Fan and Heatsink
U2
Vreg
OUT
C5
10uF
C2
10uF
U3
Vreg
OUT
C3
10uF
U1
Vreg
OUT
C1
10uF
V2
12V
V1
6V
V3
12V
10V
12V

Analog Signal Processor



PIC & FPGA & Accessories



Appendix C: Verilog

All six Verilog modules for the sound-seeking robot were written by Alex Utter and Chris Wottawa, from 2003/11/21 through 2003/12/15.

```
// Sound-seeking robot, top-level module
// Takes a motor control byte from the PIC, and translates that into PWM
// control signals for the H-bridge.  Also debounces the bumper switches and
// reports the "bumped" flag back to the PIC.
module soundbot(clk, reset, serClk, serIn, bumperSw, motorOut,
     bumped, leds, slowClk);

     input clk, reset;
     input serIn, serClk;
     input [2:0] bumperSw;
     output [3:0] motorOut;
     output bumped;
     output [7:0] leds;
     output slowClk;

     wire slowClk;
     wire [7:0] command;
     wire bumped;

     // Get serial input from PIC
     // When a byte is ready, set that as the current command.
     ShiftReg serIn(serClk, reset, serIn, command);
     //assign command = 8'b01001100;
     assign leds[7:0] = command[7:0];
     //assign leds[7:0] = {~slowClk, slowClk, bumped&(~slowClk), bumped&slowClk,
     //                  motorOut[3], motorOut[2], motorOut[1], motorOut[0]};

     // If bumper ever touched, set bumped flag
     // Attempt to filter erroneous inputs
     SwitchDebounce bumperDebounce(clk, reset, bumperSw, bumped, slowClk);

     // Convert current command to PWM out
     ClockDiv clockDiv(clk, reset, slowClk);
     MotorControl motorControl(slowClk, reset, command, bumped, motorOut);

endmodule


// A standard clock-divider.  This one divides by 4000, to convert the 2 MHz
// clock to a more useful timescale at about 500 Hz.
module ClockDiv(clk,reset,slowClk);
     input clk;
     input reset;
     output slowClk;

     // Goal is to convert 2 Mhz clock to one with a period ~2 ms.
     // So divide by 4000
     reg [15:0] counter;
     wire [15:0] nextCounter;

     // Increment counter
     assign nextCounter = (counter==16'd3999) ? 0 : (counter + 1);
     always@(posedge clk or posedge reset) begin
          if(reset)
               counter <= 0;
          else
               counter <= nextCounter;
     end

     // If more than half done, slow-clock is high
     assign slowClk = counter > 16'd1999;
```

```
endmodule


// This module takes in a motor-control byte, and puts out a PWM signal for
// the H-bridge.  The MCB is split into two 4-bit numbers in two's
// complement form.  Full reverse is -7, stop is 0, and full forward is +7;
// -8 is reserved for later use as an error code.
module MotorControl(clk,reset,commandIn,fastStop,commandOut);
    input [7:0] commandIn;
    output [3:0] commandOut;
    input fastStop; // Emergency stop
    input clk;
    input reset;

    reg [6:0] L1, L2;
    reg [6:0] R1, R2;
    reg [2:0] counter;

    wire [6:0] nextL1, nextL2;
    wire [6:0] nextR1, nextR2;
    wire [2:0] nextCounter;

    // Decode -7 to +7 throttle to 7 PWM outputs
    ThrottleDecode throttleDecodeL(commandIn[7:4], nextL1, nextL2);
    ThrottleDecode throttleDecodeR(commandIn[3:0], nextR1, nextR2);

    // Keep counter going to count every 7 clock ticks
    assign nextCounter = (counter == 6) ? 0 : counter + 1;
    always@(posedge clk or posedge reset) begin
        if(reset)
            counter <= 3'b000;
        else
            counter <= nextCounter;
    end

    // At counter = 0, insert next command set
    // Otherwise, act as a shift register
    always@(posedge clk or posedge reset) begin
        if(reset) begin
            L1 <= 7'b0;
            L2 <= 7'b0;
            R1 <= 7'b0;
            R2 <= 7'b0;
        end else if(~(|counter)) begin
            L1 <= nextL1;
            L2 <= nextL2;
            R1 <= nextR1;
            R2 <= nextR2;
        end else begin
            L1 <= {L1[5:0], L1[6]};
            L2 <= {L2[5:0], L2[6]};
            R1 <= {R1[5:0], R1[6]};
            R2 <= {R2[5:0], R2[6]};
        end
    end

    // Construct the output control signal
    assign commandOut = (fastStop) ? (4'b0000) :
        ({L1[6], L2[6], R1[6], R2[6]});
endmodule


// Decodes the -7 to +7 throttle setting to a seven-segment PWM signal.
// Each segment is a set period (~2 ms or so), and switches the H-bridge
// to stop, forward, or reverse.  Other modules cycle through the series
// of seven time segments to create a PWM output.
module ThrottleDecode(command,outA,outB);
```

```
    input [3:0] command;
    output [6:0] outA;
    output [6:0] outB;

    reg [6:0] outA, outB;

    // Decode -7 to +7 command to forward/reverse/stop in PWM form
    // (-8 is an error code; stop if it is encountered)
    // AB bits for forward 10, reverse 01, stop 00
    always@(command)
        case(command)
            4'h0: begin outA <= 7'b0000000;
                        outB <= 7'b0000000; end
            4'h1: begin outA <= 7'b1000000;
                        outB <= 7'b0000000; end
            4'h2: begin outA <= 7'b1100000;
                        outB <= 7'b0000000; end
            4'h3: begin outA <= 7'b1110000;
                        outB <= 7'b0000000; end
            4'h4: begin outA <= 7'b1111000;
                        outB <= 7'b0000000; end
            4'h5: begin outA <= 7'b1111100;
                        outB <= 7'b0000000; end
            4'h6: begin outA <= 7'b1111110;
                        outB <= 7'b0000000; end
            4'h7: begin outA <= 7'b1111111;
                        outB <= 7'b0000000; end
            4'h8: begin outA <= 7'b0000000; // -8 = error
                        outB <= 7'b0000000; end
            4'h9: begin outA <= 7'b0000000; // -7
                        outB <= 7'b1111111; end
            4'hA: begin outA <= 7'b0000000; // -6
                        outB <= 7'b1111110; end
            4'hB: begin outA <= 7'b0000000; // -5
                        outB <= 7'b1111100; end
            4'hC: begin outA <= 7'b0000000; // -4
                        outB <= 7'b1111000; end
            4'hD: begin outA <= 7'b0000000; // -3
                        outB <= 7'b1110000; end
            4'hE: begin outA <= 7'b0000000; // -2
                        outB <= 7'b1100000; end
            4'hF: begin outA <= 7'b0000000; // -1
                        outB <= 7'b1000000; end
        endcase
endmodule


// A modified shift-register, for serial communication with the PIC.
// The PIC periodically sends a motor control byte via a single serial
// data pin and a serial clock.  A shift register holds this sequence of
// bits until all 8 bits have been received.  It then outputs that sequence
// of 8 bits until another set of 8 bits has been received, and so on.
module ShiftReg(clk,reset,d,out);
    input clk, reset;
    input d;
    output [7:0] out;

    reg [2:0] count;
    reg [7:0] temp;
    reg [7:0] out;

    wire [2:0] nextCount;
    wire [7:0] nextTemp;

    assign nextCount = count + 1;
    assign nextTemp = {temp[6:0], d};
```

```
    always@(posedge clk or posedge reset)
        if(reset) begin
            count <= 0;
            temp <= 0;
            out <= 0;
        end else begin
            count <= nextCount;
            temp <= nextTemp;
            if(nextCount == 0)
                out <= nextTemp;
        end
endmodule


// Debounce the bumper switches by ensuring that the "bumped" flag is not
// set unless a single switch has been held down for at least two
// successive clock cycles (1 microsecond).  If this occurs, set the
// bumped flag and hold it high until the FPGA is reset.
module SwitchDebounce(clk,reset,in,out, slowClk);
    input clk;
    input reset;
    input [2:0] in;
    input slowClk;
    output out;

    reg [2:0] prevIn;
    wire prevOut;
    reg out;
    reg startupDelay;

    assign prevOut = out;

    always@(posedge slowClk or posedge reset) begin
        if(reset)
        startupDelay <= 0;
       else
            startupDelay <= 1;
    end

    always@(posedge clk or posedge reset) begin
        if(reset) begin
            prevIn <= 0;
            out <= 0;
        end else begin
            prevIn <= in;
            // Output high if any switch held for two cycles
            out <= startupDelay & ( prevOut | ( |(prevIn & in)) );
        end
    end
endmodule
```

Appendix D: PIC assembly code

```
; navigation.asm
; Written 2003/11/21 - 2003/12/15
; autter@hmc.edu
; cwottawa@hmc.edu
;
; Sound Seeking Robot Navigation
;
; Navigation was accomplished by a sort of state machine.  The robot
; starts in the "turn left" state.  It continually turns left until the
; microphone detects a decrease in volume, indicating that it has found
; the sound source and turned slightly too far.  (The "turn right" state
; functions in the same manner.)  The robot then goes to the "move
; forward" state, and periodically scans regions directly ahead,
; slightly to the left, and slightly to the right.  It continues to move
; forward until either the right or left sectors show a higher signal
; amplitude, in which case it begins turning in the direction of the
; strongest signal. Input consists of the two analog signal amplitudes
; (which start at 5.0V for no signal and drop to as low as about 3.0 V
; for a very strong sound signal), and a "bumped" pin that is pulled
; high if the robot collides with an obstacle (indicating that the robot
; should stop as fast as possible.)  Output consists of an SPI link for
; sending a motor control byte (MCB), a green LED that flashes to
; indicate normal operation, a green LED that turns on during forward
; motion, four yellow LEDS that represent the four most significant bits
; of the TempResult and a red LED that turns on if an emergency stop is
; initiated.  The motor control byte consists of two, signed 4-bit
; numbers (-8 to 7), corresponding to the speed of the left and right
; motors.  The maximum forward speed is +7, and -7 is the maximum
; reverse speed; -8 is an error code.
;

; use the 18F452 PIC Microprocessor
    LIST p=18F452
    include "p18f452.inc"

; allocate variables
MCB equ 0x00                 ; store current MCB to be sent to FPGA

; NOTE:  PORTD is not being used by the FPGA now.

; need high and low, so can use 10-bit A/D
MAX440L equ 0x01             ; store maximum 440 result
MAX440H equ 0x11
COR600L equ 0x02             ; store corresponding 600 result
COR600H equ 0x12
MAX600L equ 0x03             ; store maximum 600 result
MAX600H equ 0x13
COR440L equ 0x04             ; store corresponding 440 result
COR440H equ 0x14
PRV440L equ 0x05             ; store previous samples
PRV440H equ 0x15             ; this is used for getting corresponding
PRV600L equ 0x06             ; amplitude of other tone.
PRV600H equ 0x16

MAXSUML equ 0x07             ; MAXSUM = MAX440 + MAX600
MAXSUMH equ 0x17
CORSUML equ 0x08             ; CORSUM = COR440 + COR600
CORSUMH equ 0x18
TEMPRESL equ 0x09            ; TEMPRES = MAXSUM - CORSUM
TEMPRESH equ 0x19

PRVRESL equ 0x0A             ; store previous result (only if turning)
PRVRESH equ 0x1A
```

```
MAXRESL equ 0x0B              ; store maximum result.
MAXRESH equ 0x1B

ZERO equ 0x30                 ; contains 0

AL equ 0x31                   ; 16-bit compare registers
AH equ 0x41
BL equ 0x32
BH equ 0x42
BisBigger equ 0x33            ; flag for 16-bit compare (0th bit only)

timerDone equ 0x0D            ; flag, use bit 0, assert if TMR0 is done
                              ; counting

; Motor Control Byte (MCB) to be sent serially
; Output protocol [Left Motor Thrust(4)][Right Motor Thrust(4)]
; using signed 2's complement numbers (negative means go backwards)
goleft equ 0x97
goright equ 0x79
gostraight equ 0x77
gostop equ 0x00

; Pulse Widths for Microphone Servo control
midp equ 0x28                 ; middle pulse
leftp equ 0x1A                ; left
rightp equ 0x38               ; right
midpl equ 0x27                ; slightly left of middle
midpr equ 0x29                ; slightly right of middle

; threshold value
thresh equ 0x30

; timer 0 starting values
; the lower the number the greater the time.
; When called for a delay, etc., the timer will be loaded with
; the given values in both the high and low counter registers.
; Don't use 0x00, or 0x80, because of PWM conflicts.
timerADRecord equ 0xD7        ; time to record a given A/D series
timerServoWait equ 0xDB       ; time to wait for the microphone servo
timerMotorForward equ 0xB0    ; time to allow forward movement before
                              ;     stopping and taking more samples
timerMotorTurn equ 0xE8       ; time for turning before stopping


; store servo positions and corresponding MCB in table
    org 0x400
    DB midp,gostraight,leftp,goleft,rightp,goright,0,0


; begin main program
    org 0

    bra setup

    org 0x08                  ; being bumped is high priority
    bra bumped

    org 0x18                  ; low priority finished collecting A/D
    bra lowP                  ; or PWM is done.

    org 0x30
setup
    ; I/O configuration

    setf TRISA                ; use RA0, RA1 for A/D input. RA3 for Vref
    clrf TRISB                ; use RB1 for FPGA reset.
    bsf PORTB,1               ; reset the FPGA
```

```
    setf TRISC
    bsf TRISB,0              ; use RB0 for INT0 (Bumped?)
    bcf TRISB,2              ; yellow LEDs for TEMPRES
    bcf TRISB,3              ; use RB2-RB5
    bcf TRISB,4
    bcf TRISB,5
    bcf TRISC,3              ; use RC3 for SCLK,
    bcf TRISC,5              ; use RC5 for SPO
    bcf TRISC,2              ; use RC2 for Pulse to Servo
    bcf TRISC,6              ; green LED, normal operation, toggle on
                             ; each compute
    bcf TRISC,7              ; red LED, bumped indicator
    bcf TRISA,5              ; yellow LED, straight or turning?
    bcf PORTB,1              ; clear FPGA reset

    setf TRISD               ; PORTD as input to not mess up FPGA
;   clrf TRISD               ; output to LEDs to help debug
;   clrf PORTD
;   bsf PORTD,6              ; pattern to indicate initial LED status
    bcf PORTC,6              ; green LED, normal op, initially off

    ; A/D configuration (see chapter 17 in PIC manual)
    movlw 0x85               ; right justified, Fosc/2, AN0 and AN1 input
    movwf ADCON1             ; Vref -> AN3
    movlw 0x01               ; power up, Fosc/2, initially use ch0
    movwf ADCON0             ; don't start taking data just yet

    ; SPI configuration
    clrf SSPCON1             ; for SPI need to clear [4:0]
    bsf SSPCON1,5            ; set SSPEN (turn on SPI)
    bcf SSPSTAT,7            ; must be cleared on SPI slave mode
    bsf SSPSTAT, 6           ; data on rising edge of clock

    ; PWM configuration
    movlw 0xFF               ; set PWM period (change to 11 ms)
    movwf PR2
    movlw leftp              ; set default PWM duty cycle
    movwf CCPR1L             ; always start in the center
    movlw 0x0C               ; PWM mode
    movwf CCP1CON

    ; Interrupt configuration (see chapter 8 of PIC manual)
    ; Interrupts are used for bumped condition and PWM module
    bsf RCON,7               ; enable priority levels
    movlw 0xF0               ; enable interrupts, timer and external
    movwf INTCON             ; and clear all interrupt flags
    bcf INTCON2,2            ; timer0 low priority
    bsf INTCON2,6            ; falling edge of INT0
    bcf PIR1,1               ; clear timer2 interrupt flag
    bsf PIE1,1               ; enable timer2 interrupts
    bcf IPR1,1               ; timer 2 interrupt low priority

    ; Timer2 Configuration (see chapter 12 of PIC manual)
    ; Timer2 is used in PWM module
    movlw 0x06               ; configure TMR2 (turn on, 16:1 prescale)
    movwf T2CON
    ; Timer0 used for A/D conversions
    movlw 0x84               ; enable, 16-bit, internal clock
    movwf T0CON              ; Use prescale(2:0) to set length of time
                             ; to wait and then sample
                             ; (1:8 is about one second)

   ; send range to servo to prevent freezing on start up.
    movlw leftp
    movwf CCPR1L
    movlw timerServoWait
    call delayAWhile
```

```
    movlw rightp
    movwf CCPR1L
    movlw timerServoWait
    call delayAWhile
    movlw midp
    movwf CCPR1L

    movlw goleft            ; initially go left
    movwf MCB
;   movwf PORTD
;   movwf SSPBUF            ; send MCB
    movlw 0x04
    movwf TBLPTRH           ; initialize table pointer to 000400
    clrf TBLPTRU

    movlw leftp             ; move to the left and take the initial
    movwf CCPR1L            ; sample, so that we start in a valid state
    movlw timerServoWait
    call delayAWhile
    call ADgo
    call compute
    movff MAXRESL,PRVRESL
    movff MAXRESH,PRVRESH


reinit                      ; reset defaults / table pointer
    clrf TBLPTRL
    movlw gostraight
    cpfseq MCB
    bra turnsetup

    clrf MAXRESH
    movlw thresh            ; threshold value to detect tone
    movwf MAXRESL
    bsf PORTA,5             ; yellow LED on!
    movlw goleft            ; turn left by default
    movwf MCB
goingstraight
    ; if going straight, scan microphone
    tblrd*+                 ; Pulse -> TABLAT -> W
    movf TABLAT,0
    bz output               ; send MCB out if at end of table
    movff TABLAT,CCPR1L     ; send current desired direction PWM module
    movlw timerServoWait
    call delayAWhile        ; wait for mic to get into position
    tblrd*+                 ; next potential best MCB -> TABLAT
    call ADgo               ; take a set of samples
    call compute            ; TEMPRES = (MAX440+MAX600)-(COR440+COR600)
    movff TEMPRESH,AH       ; set up for compare
    movff TEMPRESL,AL
    movff MAXRESH,BH
    movff MAXRESL,BL
    call compare16          ; 16-bit compare: TEMPRES > MAXRES (B)?
    btfsc BisBigger,0
    bra goingstraight       ; do this, if MAXRES bigger
newmaxres                   ; do this, if TEMPRES bigger
    movff TEMPRESH,MAXRESH  ; store new max
    movff TEMPRESL,MAXRESL
    movff TABLAT,MCB        ; and MCB
    bra goingstraight

output
    movlw gostraight        ; Check if still going straight
    cpfseq MCB
    bra outputDelayTurn     ; If turning, use turn delay
outputDelayForward
    movlw timerMotorForward
```

```
    bra outputOutput
outputDelayTurn
    movlw timerMotorTurn
outputOutput
;   movff MCB,PORTD          ; display on LEDs for debugging
    movff MCB,SSPBUF         ; Load MCB to Serial Port Buffer
    call delayAWhile         ; Wait for the time in W (turn or forward)
    movlw gostop             ; Then stop the motors
    movwf SSPBUF
    bra reinit               ; resets table pointer and defaults


; if the robot is turning, move the microphone to the middle and
; sample until the temporary result begins to go down
; recall that TEMPRES = MAX440 + MAX600 - COR440 - COR600
turnsetup
    movff MAXRESL,PRVRESL    ; just came from forward movement
    movff MAXRESH,PRVRESH    ; where MAXRES was the large signal
    clrf MAXRESH             ; that caused us to turn initially
    bcf PORTA,5              ; Turning/Going Straight LED off!
    movlw thresh             ; store the threshold value
    movwf MAXRESL            ; will use previous result register
    movlw goleft             ; check direction
    cpfseq MCB
    bra right
left
    movlw midpl              ; if turning left, send this PW
    bra turninit
right
    movlw midpr              ; if turning right, send this PW
turninit
    movwf CCPR1L             ; Send position to PWM out
    movlw timerServoWait
    call delayAWhile         ; Wait for mic to get into position
turn
    call ADgo
    call compute
    movff TEMPRESH,AH        ; set up for compare
    movff TEMPRESL,AL
    movff MAXRESH,BH         ; (MAXRES is threshold)
    movff MAXRESL,BL
    call compare16           ; 16-bit compare: TEMPRES > MAXRES (B)?
    btfss BisBigger,0
    bra aboveThreshold       ; if TEMPRES bigger, do nothing
    bra keepturning          ; If TEMPRES smaller, set to threshold
;   movff MAXRESH, TEMPRESH
;   movff MAXRESL, TEMPRESL
aboveThreshold
    movff PRVRESH,BH         ; set up for compare
    movff PRVRESL,BL
    call compare16           ; 16-bit compare: TEMPRES > PREVRES (B)?
    btfsc BisBigger,0
    bra stopturning          ; do this, if PRVRES bigger
keepturning                  ; do this, if TEMPRES bigger or equal
    movff TEMPRESH,PRVRESH   ; store current value
    movff TEMPRESL,PRVRESL
    movff MCB,SSPBUF         ; Send MCB
;   movff MCB,PORTD
    movlw timerMotorTurn     ; Wait a while...
    call delayAWhile
    movlw gostop             ; ...then stop the motors
    movwf SSPBUF
    bra turn                 ; and continue the turning process
stopturning
    movlw goleft             ; check direction
    cpfseq MCB
    bra goleftonce           ; if currently going right
```

```
gorightonce
    movlw goright
    bra goonce
goleftonce
    movlw goleft
goonce
    movwf SSPBUF
    movlw timerMotorTurn
    call delayAWhile
    movlw gostop
    movwf SSPBUF
    movlw timerMotorTurn
    call delayAWhile
    movlw gostraight
    movwf MCB
    bra output


; SUBROUTINES & INTERRUPTS!


; Set up the timer to trigger in a given amount of time, specified
; in the W register.  (See timer constants, above)
setupTimer
    movwf TMR0H
    movwf TMR0L
    bcf timerDone,0    ; Clear previous completion flags
    return


; Set up timer and busywait for 1/4 second (ish)
delayAWhile
    call setupTimer
waitMore
    btfss timerDone,0
    bra waitMore
    return


; the compute subroutine assumes that the MAX440, MAX600,
; COR440, and COR600 are all set accordingly.
; These values are used to compute TEMPRES
compute
    bsf PORTB,3             ; clear external TEMPRES LEDs
    bsf PORTB,4
    bsf PORTB,5
    bsf PORTB,2
    btg PORTC,7             ; Toggle the green LED; not related to
                            ; compute, but a convenient place to do so
    movf MAX440L,0          ; 16-bit add: MAXSUM = MAX440 + MAX600
    addwf MAX600L,0
    movwf MAXSUML
    movf MAX440H,0
    addwfc MAX600H,0
    movwf MAXSUMH
    movf COR440L,0          ; 16-bit add: CORSUM = COR440 + COR600
    addwf COR600L,0
    movwf CORSUML
    movf COR440H,0
    addwfc COR600H,0
    movwf CORSUMH
    movf CORSUML,0          ; 16-bit sub: TEMPRES = MAXSUM - CORSUM
    subwf MAXSUML,0
    movwf TEMPRESL
    movf CORSUMH,0
    subwfb MAXSUMH,0
    movwf TEMPRESH
```

```
    btfss TEMPRESL,6          ; output four most significant bits of
    bcf PORTB,2               ; to external LEDs
    btfss TEMPRESL,7
    bcf PORTB,3
    btfss TEMPRESH,0
    bcf PORTB,4
    btfss TEMPRESH,1
    bcf PORTB,5

    btfss TEMPRESH,7          ; if TEMPRES is negative, set it to zero.
    return
    clrf TEMPRESH             ; this makes 16-bit compare easier
    clrf TEMPRESL             ; and negative TEMPRES will never
                              ; be above the threshold anyway.
;   movff TEMPRESL,PORTD      ; display each temp result on LEDs
    return



; A/D subroutine (each call to this subroutine takes one "set" of
;  samples and saves max440, max600, cor440, and cor600
;      max440 is the maximum value from the 440 BP filter
;      cor600 is the corresponding value from the 600 BP filter
;      max600 is the maximum value from the 600 BP filter
;      cor440 is the corresponding value from the 440 BP filter
;  a "set" of samples is defined by the values in TMR0L, and TMR0H
;  when this subroutine is called. (This constant "timer" is above)
ADgo
    clrf MAX440H              ; starting a new A/D run..
    clrf MAX440L              ; clear out the old values
    clrf MAX600H
    clrf MAX600L
    clrf COR440H
    clrf COR440L
    clrf COR600H
    clrf COR600L
    clrf PRV600L
    clrf PRV600H
    clrf PRV440L
    clrf PRV440H
    movlw timerADRecord       ; Set to go off in 1/4 second
    call setupTimer           ; Initialize timer, but continue while waiting

    bsf ADCON0,3              ; Start by taking an initial 600 Hz reading
    bsf ADCON0,2              ; Start A/D converter
initialRead600
    btfsc ADCON0,2            ; poll A/D status
    bra initialRead600
    btg ADRESH,1              ; invert first 10 bits, since input is
    btg ADRESH,0              ; negative amplitude
    comf ADRESL,1
    movff ADRESH,PRV600H
    movff ADRESL,PRV600L

readAD
    btfsc timerDone,0         ; stop if we have taken enough samples
    return

    bcf ADCON0,3              ; use channel 0 (440Hz)
    bsf ADCON0,2              ; start A/D converter
read440
    btfsc ADCON0,2            ; pole A/D status
    bra read440
    btg ADRESH,1              ; invert first 10 bits, since input is
    btg ADRESH,0              ; negative amplitude
    comf ADRESL,1
```

```
    movff ADRESL,PRV440L       ; store A/D result in PRV
    movff ADRESH,PRV440H
    movff ADRESH,AH            ; set up A & B for compare
    movff ADRESL,AL
    movff MAX440H,BH
    movff MAX440L,BL
    call compare16             ; 16-bit compare: ADRES > MAX440?
    btfsc BisBigger,0
    bra setup600               ; if MAX440 is bigger
newmax440                      ; if ADRES is bigger
    movff ADRESH,MAX440H       ; store new max 440
    movff ADRESL,MAX440L
    movff PRV600L,COR600L      ; and corresponding 600Hz amplitude
    movff PRV600H,COR600L
setup600
    bsf ADCON0,3               ; use channel 1 (600Hz)
    bsf ADCON0,2               ; restart A/D converter
read600
    btfsc ADCON0,2             ; pole A/D status
    bra read600
    btg ADRESH,1               ; invert first 10 bits, since input is
    btg ADRESH,0               ; negative amplitude
    comf ADRESL,1
    movff ADRESL,PRV600L       ; store A/D result in PREV
    movff ADRESH,PRV600H
    movff ADRESH,AH            ; set up for compare
    movff ADRESL,AL
    movff MAX600H,BH
    movff MAX600L,BL
    call compare16             ; 16-bit compare: MAX600 (B) > ADRES?
    btfsc BisBigger,0
    bra readAD                 ; if MAX600 is bigger, move on
newmax600                      ; otherwise store new max
    movff ADRESH,MAX600H       ; store new max 600
    movff ADRESL,MAX600L
    movff PRV440L,COR440L      ; and corresponding 440Hz amplitude
    movff PRV440H,COR440H
    bra readAD


; compare16 compares two 16-bit values A & B,
; stored in memory locations AH,AL,BH,BL
; if B is Bigger, BisBigger,0 bit is set.
compare16                      ; 16-bit compare: A > B?
    clrf BisBigger
    movf AH,0
    cpfseq BH                  ; AH == BH?
    bra Hnotequal              ; high not equal
    bra Hequal                 ; high equal
Hnotequal                      ; if not equal, just compare
    cpfsgt BH                  ; If BH > AH, skip
    bra Abig
    bra Bbig
Hequal                         ; if equal need to check lower bits
    movf AL,0
    cpfsgt BL                  ; if BL > AL, skip
    bra Abig
Bbig
    bsf BisBigger,0
    return
Abig
    bcf BisBigger,0
    return

; low priority interrupt code
lowP
    btfsc INTCON,2             ; first check if TMR0 has overflowed
```

```
    bra timerDoneLbl
    bcf PIR1,1                 ; clear PWM interrupt flag
    retfie
timerDoneLbl
    bsf timerDone,0            ; set timerDone flag
    bcf INTCON,2               ; clear interrupt flag
    retfie

; bumped (high priority)
bumped
    movlw gostop               ; 00 is MCB for brake
    movwf SSPBUF               ; send to FPGA
;   movwf PORTD                ; and to PORTD for debugging
    bsf PORTC,6
    bcf PORTC,7
    sleep                      ; then go to sleep
    bra bumped
    end
```

Appendix E: Matlab Code

```
function y = makeAlternatingTone(f1,f2, fSwitch, dur, fs)
% y = makeAlternatingTone(f1,f2, fSwitch, dur, fs)
%  Makes a tone that alternates between f1 and f2, switching at the given
%  frequency.  The output will have the given duration and sampling
%  frequency.

% Allocate output space
ly = floor(dur*fs);
y = zeros(1,ly);

% Make a standard two-tone loop
oneLoop = [makeWindowedTone(f1, 1/(2*fSwitch), fs) makeWindowedTone(f2,
1/(2*fSwitch), fs)];
oll = length(oneLoop)-1;

% Repeat this single loop until near the end...
t = 1;
while(t + oll <= ly)
    y(t:t+oll) = oneLoop;
    t = t + oll;
end
% And leave the rest of y as zeros.



function y = makeWindowedTone(f, dur, fs)
% y = makeWindowedTone(f, dur, fs)
%  Makes a tone at frequency f with the given duration and sampling
%  frequency.  This tone is then windowed over the course of two
%  wavelengths at the beginning and end, to avoid clicking.

% Make the trapezoidal envelope
env = ones(1,floor(dur*fs));
twoWaveDur = floor(2*fs/f);
for n = 1:twoWaveDur
    env(n) = (n-1)/twoWaveDur;
    env(length(env)-n+1) = (n-1)/twoWaveDur;
end

% Make the signal y
t = ((1:floor(dur*fs))-1) / fs;
y = env .* sin(2*pi*f*t);
```