SoundCube

Marisol Beck, Kathleen Kohl, and Sakshi Shah
E155 – Fall 2016

Abstract

A Cyclone 4 FPGA and Raspberry Pi microcontroller were used to build a wireless, portable, musical instrument called a SoundCube. The FPGA takes in keypad inputs corresponding to five-sound datasets and transmits this information via Bluetooth to a Raspberry Pi inside the SoundCube. The Raspberry Pi uses an SPI protocol to receive accelerometer data from an E80 IMU4 Breakout Board via a dual-channel analog-to-digital converter. Acceleration data is used to determine the orientation of the SoundCube, and a different note is programmed onto each face of the cube. The volume of the sound changes based on the velocity experienced by the SoundCube in the z-direction. The SoundCube allows users to play simple five-note songs or allows a more experienced user to switch between different five-sound datasets to play more complicated songs.

1 Introduction

1.1 Project Overview

This project involves a wireless, portable plexiglass musical instrument called the "SoundCube." The SoundCube generates a sound based on which side of the cube is facing up and manipulates the volume of the sound based on its velocity in the z-direction (i.e. when it is lifted off a surface). The SoundCube can be programmed to have several sound datasets, each containing one sound per face of the cube. A user selects which sound dataset they intend to program onto the cube by pressing a button on a keypad that corresponds to their desired sound dataset. The keypad is connected to an FPGA, which transmits the selected sound dataset to a Raspberry Pi 3 microcontroller via a UART communication protocol over an SPP Bluetooth channel.

The high level block diagram for the SoundCube is shown in Fig. 1. Outside the SoundCube, keypad data travels from the 4x4 matrix keypad used in Lab 3 [4] to the MuddPi Mark IV utility board containing an Cyclone 4 FPGA. This keypad data is sent through a serial UART transmission line from the FPGA to a BlueSMiRF Bluetooth module. The BlueSMiRF Bluetooth module transmits the ASCII information of the key that has been pressed to a Raspberry Pi microcontroller inside the SoundCube. Based on the keypress data, the Raspberry Pi picks a corresponding sound dataset and programs a sound onto each of the faces of the SoundCube. A BMA145 accelerometer is used to detect the motion of the cube, which also determines the volume control functionality. A speaker circuit inside the cube, modified from Lab 6 [5], plays the sounds in real time. In addition, all hardware inside the SoundCube is battery-powered to ensure that the cube is portable.

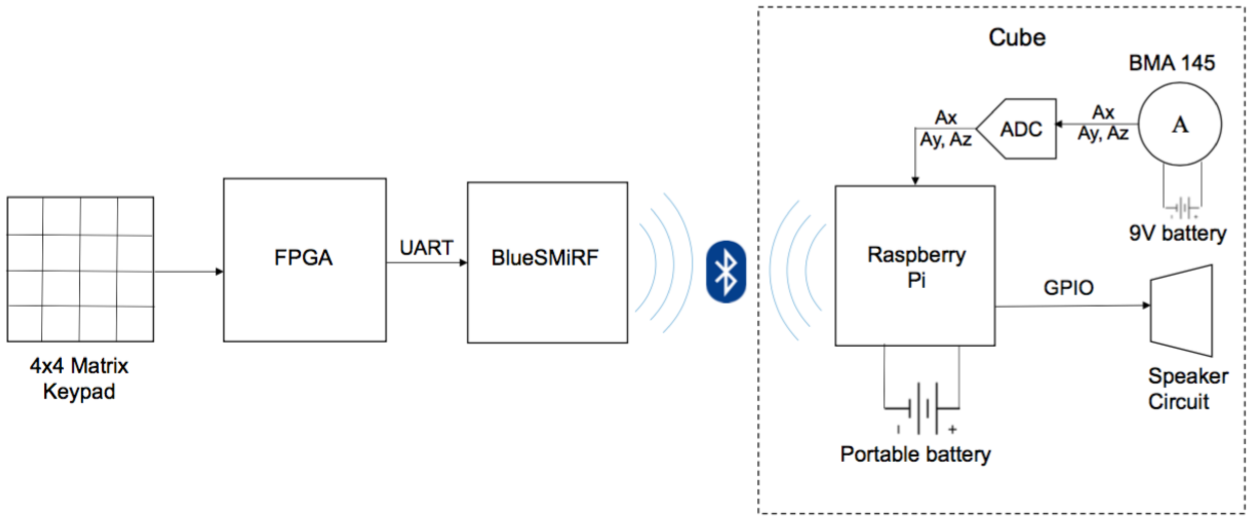


Figure 1: A block diagram demonstrating the various components required for the SoundCube, and interfaces between them.

1.2 Deliverables

This section outlines the specific functionality of the SoundCube.

- The SoundCube is wireless, battery powered, and portable.
- Each face of the SoundCube corresponds to a different sound. A BMA145 accelerometer is used to detect which face of the cube is up. The sound corresponding to that face is then played by the SoundCube.
- When the SoundCube is moved along the z-axis (i.e., lifted off the surface), volume changes based on the velocity at which it's moving. This velocity is also detected by the accelerometer.
- The 4x4 matrix keypad connected to the FPGA allows a user to choose between 4 sets of pre-programmed 6-sound datasets which can be played on the SoundCube. One of the sounds in each dataset is silence.
- The FPGA interfaces, via a UART protocol, with a BlueSMiRF Bluetooth module in order to send sound datasets via Bluetooth to the Raspberry Pi.
- The Raspberry Pi interfaces via SPI with two analog-to-digital converters (MCP3002) which in turn receive voltage information from the accelerometer.

2 Schematics

The FPGA on the breadboard communicates with the SoundCube over an SPP Bluetooth channel, thus the wiring diagrams are split into Fig. 2 and Fig. 3 respectively. These correspond to the wiring outside and inside the cube.

2.1 Outside the Cube

Outside of the cube, a 4x4 matrix keypad is connected to the FPGA in the same way as for E155's keypad lab [4]. The rows are powered sequentially and the columns are scanned to detect a keypress. Then, the detected keypress is routed via a UART protocol to a BlueSMiRF, a small Bluetooth module that had not been previously used in class. More information about the internal functionality in the FPGA follows in Section 4. Information about the BlueSMiRF follows in Section 3.1.

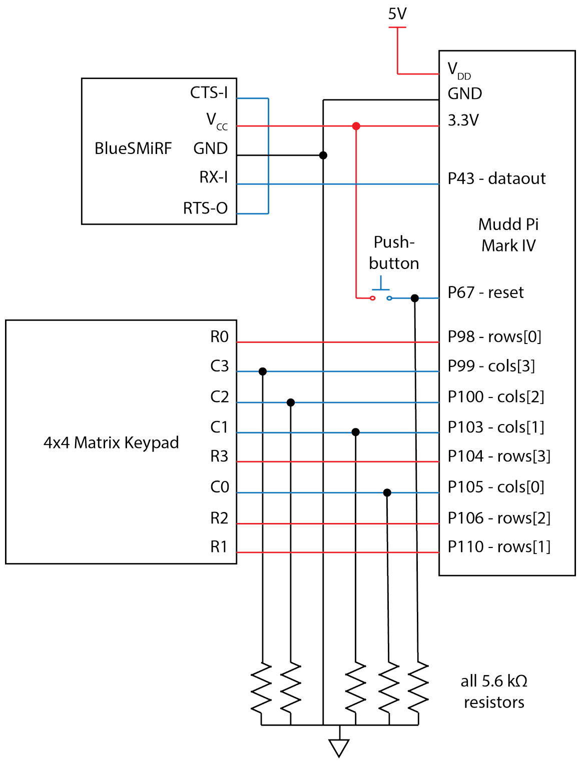
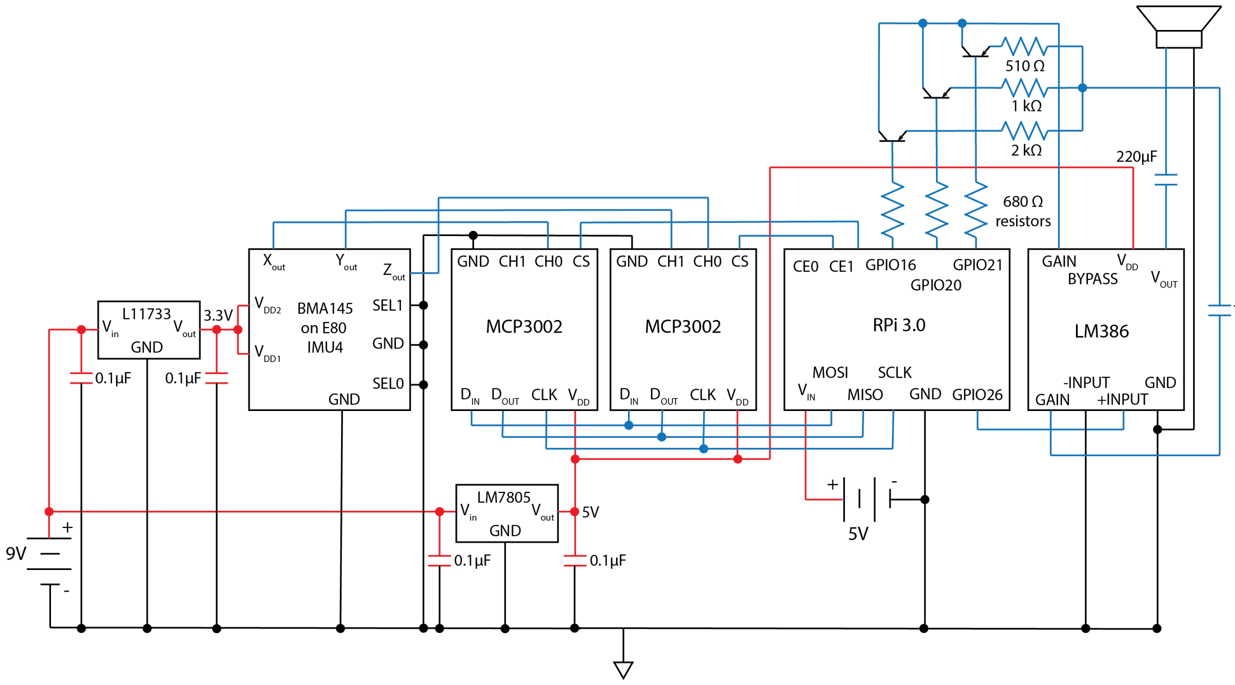


Figure 2: Wiring diagram of the circuits on the breadboard. This wiring shows the interface between the keypad, the FPGA, and the BlueSMiRF.

2.2 Inside the Cube

All of the remaining hardware is housed inside a plexiglass cube. See Appendix D for the dimensions and assembly instructions of the cube. One cube face has a circle cut out of it, which is where an 8 Ω speaker is mounted. Inside the cube, a Raspberry Pi is powered using a 5V cell phone battery charger with a micro-USB cable. The Pi is connected to a circuit which controls the audio output of the cube as well as a circuit which reads in the acceleration data from the accelerometer. The overall layout can be seen in Fig. 3. The speaker's wiring and the accelerometer wiring are independent of one another, so they are each discussed in separate sections.



510 Ω
1 kΩ
2 kΩ
220µF
680 Ω resistors
100µF
L11733 V_{in} V_{out} GND
3.3V
0.1µF
0.1µF
X_{out} Y_{out} Z_{out}
V_{DD2} BMA145 on E80 IMU4 SEL1 GND SEL0 V_{DD1} GND
GND CH1 CH0 CS MCP3002 D_{IN} D_{OUT} CLK V_{DD}
GND CH1 CH0 CS MCP3002 D_{IN} D_{OUT} CLK V_{DD}
CE0 CE1 GPIO16 GPIO21 GPIO20 RPi 3.0 MOSI SCLK V_{IN} MISO GND GPIO26
GAIN V_{DD} BYPASS V_{OUT} LM386 -INPUT GND GAIN +INPUT
LM7805 V_{in} V_{out} GND
5V
0.1µF
0.1µF
9V
5V

Figure 3: Wiring diagram of the Raspberry Pi, accelerometer, and speaker.

2.2.1 Speaker Circuit

Square waves of varying frequencies are output from pin 26 of the Raspberry Pi. The waves go through the LM386 amplifier used in Lab 6, and are then sent through a 220 μF capacitor to stabilize the output sound [5]. The SoundCube requires a digitally-controllable volume output, so some wiring not required in Lab 6 was added across the gain pins (1 and 8) of the LM386 [6]. From pin 1, a 100 μF capacitor is wired to three resistors in parallel, which are each wired in series with a 2N3906 bipolar junction transistor (see the upper right section of Fig. 3), which then goes back to pin 8 of the amplifier. The base pin of each transistor is wired through a 680Ω resistor in order to limit the base current on the transistors ($\frac{3.3\text{V}-0.7\text{V}}{3\text{mA}} = 650\Omega$, where 0.7V is due to the voltage drop across a diode, and 4mA is the average current driven by the Pi's output pins. The closest resistor value from the stockroom was 680Ω).

The resistors which control the gain on the amplifier are 510Ω, 1000Ω, and 2000Ω resistors.

This allows for potentially up to 8 different volumes to be selected depending on which transistors are powered off and on (and thus which resistors current flows through). Based on the LM386 datasheet, a lower resistor value ought to lead to higher gain (and thus higher volume) [6]. However, when testing the circuit, sending the current through lower resistors actually led to a lower volume output from the speakers. The rest of the circuit was found to be self-consistent when checked with a multimeter, so lower resistor values were chosen to create lower volume outputs. The outputs sent to the base pins of the transistors are configured in an array at the end of Appendix C.4 in the file `audio.h`. Only six volume levels were actually implemented, as the selection of all three resistors (510Ω, 1000Ω, and 2000Ω in parallel, with $R = 289\Omega$) does not result in a noticeably different volume than the second lowest option (510Ω and 1000Ω in parallel, with $R = 338\Omega$); and the selection of no resistors was not used because it resulted in a less consistent output sound.

2.2.2 Accelerometer Circuit

The BMA145 accelerometer on the E80 IMU4 Breakout board outputs voltages based on the acceleration each axis is experiencing, so it requires a very stable voltage input in order to maintain stable output values. A 9V battery was attached to a surface mount 3.3V voltage regulator. Several TO-92 voltage regulators were compared but all the output voltages were well below 3V, whereas the L11733 surface mount voltage regulator gave a consistent 3.3V output, so wires were soldered to the surface mount regulator. Bypass capacitors of 0.1 μF were added to the V_{in} and V_{out} connections of the voltage regulator in order to reduce noise, as well as to create a low-impedance path to ground for the power supply. This stable 3.3V signal was then used to power the BMA 145 accelerometer. More information about the specific functionality and pinouts of the breakout board can be found in Section 3.2.

There are three output analog voltages from the BMA145 corresponding to x, y, and z acceleration. These needed to be sent through an analog to digital converter. The MCP3002 10-bit dual-channel ADC was chosen because they were readily available and had previously been used in Lab 6 [5]. Each ADC can take in two different channels of data, so two ADCs were necessary in order to transmit all three voltages. Each ADC interfaces with the Raspberry Pi via an SPI protocol, the details of which can be found in Section 5.

The MCP3002 has a 200 ksps maximum sampling rate at 5V, so an LM7805 5V voltage regulator (powered by the 9V battery) was used to power the ADCs, with the same 0.1 μF bypass capacitor wiring as for the 3.3V regulator. Since the ADCs were powered with a 5V regulator, up to 5V were sent into the Raspberry Pi's GPIO pins. The Pi has no over-voltage protection, which means that inputting 5V into its GPIO pins could result in unpredictable behavior or could fry the Pi. However, since it functioned throughout the testing and prototyping process, this problem was not caught until the cube was first demoed, which is why the wiring diagram in Fig. 3 shows wires that go directly from the ADC outputs to the Pi. If this project were to be attempted again, some suggested alternatives include: (a) powering the MCP3002 with a 3.3V regulator and testing whether the input voltages can be sampled quickly enough, or (b) implementing a voltage divider [9].

3 New Hardware

3.1 BlueSMiRF Bluetooth Module

The BlueSMiRF Bluetooth Module communicates with the FPGA via a UART protocol and sends data to the Pi over an SPP Bluetooth Channel. The BlueSMiRF uses an 8N1 UART protocol at a default 115200 bps baud rate, which means it expects 8 data bits, no parity bit, and one stop bit (see Fig. 6). ASCII characters are sent to the BlueSMiRF bit-by-bit least-significant bit first. The UART protocol is explained further in Section 4.2. In addition, the BlueSMiRF transmits Bluetooth data as ASCII characters, which means the FPGA UART transmission must also represent keypad data in ASCII.

Configuring Bluetooth settings on the BlueSMiRF can be done through a serial UART link (using UART code provided in the `EasyPIO.h` file on the E155 website), however this proved to be extremely tricky. Using Command Mode as outlined in the Advanced User Manual [7] enabled the visible connection of one BlueSMiRF module to another, but the Pi was unable to detect a serial response from the BlueSMiRF. Modifying configuration settings and requesting two BlueSMiRF modules to pair with each other is not recommended without further investigation into the serial UART link between the Raspberry Pi and BlueSMiRF that is required to access Command Mode.

3.2 Inertial Motion Unit and BMA145 Accelerometer

A BMA 145 accelerometer was used to determine the orientation of the cube. This accelerometer was wired on an E80 IMU breakout board and the pinouts of the breakout board was found on the Spring 2016 E80 website [2]. The BMA 145 accelerometer requires a very stable voltage source, as mentioned in Section 2.2.2. An 3.3V voltage regulator connected to a 9V battery was used as a voltage source, with 0.1 μF bypass capacitors connecting V_{in} and V_{out} to ground. Because the AD22280-R2 chip on the E80 IMU4 board was not used, a 5V power supply does not need to be connected.

Values corresponding to the orientation of the accelerometer were found in the BMA145 datasheet and shown in Table 1. However, these values are provided for a supply voltage of 3.0V. Because the supply voltage used for the SoundCube was 3.3V, acceleration values were experimentally found to be consistently 0.1V higher than stated on the datasheet. Experimentally-determined voltage thresholding was used to determine the orientation of the SoundCube. Table 2 shows the thresholding values found to be most consistent.

Table 1: Output values of the accelerometer sensor at rest or at uniform motion from the BMA145 datasheet. The datasheet only specified values with two significant digits.

Orientation	+x	-x	+y	-y	+z	-z
A_x	1.8V	1.2V	1.5V	1.5V	1.5V	1.5V
A_y	1.5V	1.5V	1.8V	1.2V	1.5V	1.5V
A_z	1.5V	1.5V	1.5V	1.5V	1.8V	1.2V

The BMA145 accelerometer has a "stand-by" setting that is activated when the SEL0 and SEL1 pins are connected to V_{DD} and turned off when the pins are grounded. These pins should be grounded to receive constant acceleration data.

4 FPGA Design

Outside the SoundCube, the FPGA performs the following tasks: interpreting input data from the matrix keypad, constructing a decoder module that converts keypad data into an ASCII representation according to ASCII-Code's online reference manual [1], and assembling this data into a 10-bit UART packet to be transmitted to the BlueSMiRF Bluetooth module. The high-level FPGA schematic is shown in Fig. 4, while the schematics for specific modules are described in Appendix A. Hardware was synthesized in SystemVerilog, compiled in Quartus II, and simulated using ModelSim. Verilog files for the synthesized hardware can be found in Appendix B.1 and B.2.

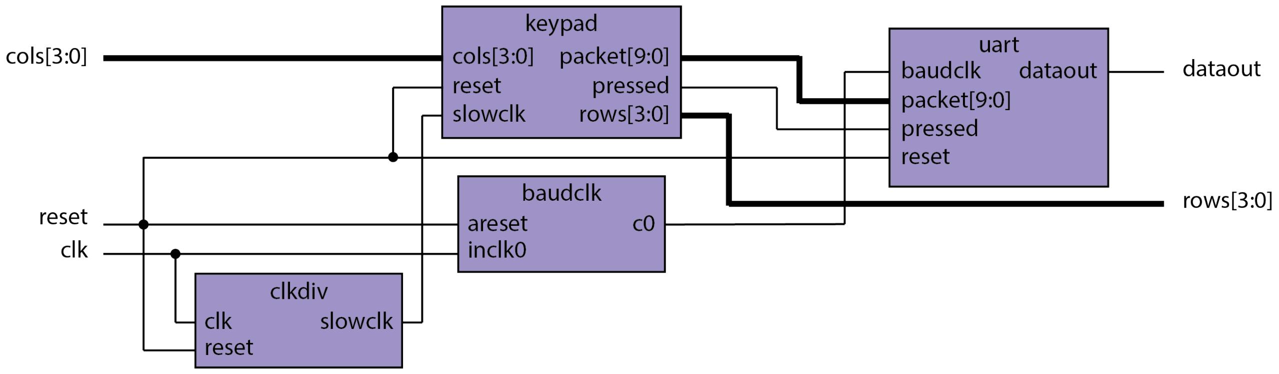


Figure 4: High-level schematic describing the datapath of the FPGA hardware. Keypad data arrives as an input from an asynchronous matrix keypad on the breadboard and is converted into a 10-bit `packet`. This packet is outputted bit-by-bit, labeled `dataout` on the diagram.

The FPGA design contains four modules of importance: a clock divider for switch-debouncing the keypad inputs (`clkdiv`), a keypad decoder (`keypad`), a baud rate generator module (`baudclk`), and a module that transmits UART data (`uart`). Several modules are clocked on different clocks out of necessity. The `keypad` module, which contains a keypad scanner circuit, has to be synchronized with the experimentally-determined switch-debouncing clock generated in the `clkdiv` module in order to be able to detect keypresses with certainty. However, the `uart` module has to run on a `baudclk` which is required by the BlueSMiRF Bluetooth module. In order to combat this asynchronicity, a synchronizer is placed before blocks of logic that require a clock rate change. This report will provide detail on the `keypad` module and the UART transmitter implementation.

4.1 Keypad Scanner Circuit

The `keypad` module contains a scanning finite state machine (see Fig. 5) that powers each of the rows on the matrix keypad on the positive edge of a 150 Hz `slowclk`, created by the `clkdiv` module.

This keypad module was modified from the keypad decoder constructed in Lab 3 [4]. If a column input is HIGH during a row output, that row-column combination is registered as a keypress. The bitwise-OR of all the column inputs is also an output of this module (pressed), a control signal that is indicative that a key press has occurred. Using a case statement, the keypress data is converted into a 10-bit UART packet with a structure as described in Fig. 6. Appendix A contains schematics for the keypad module.

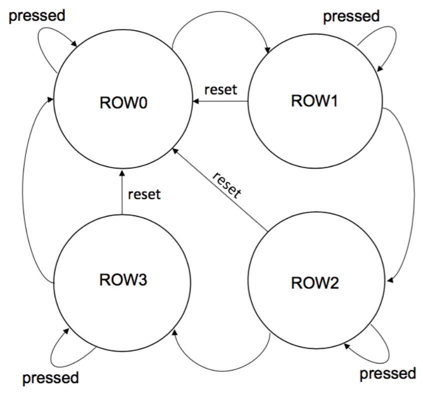


Figure 5: The keypad module contains a keypad scanner finite state machine that powers each row of the 4x4 matrix keypad one by one in order to determine which key is pressed.

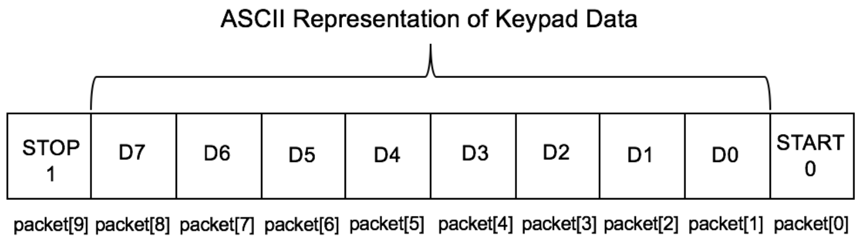


Figure 6: 10-bit UART packet, an output signal of the keypad module.

4.2 UART Transmitter Implementation

The uart module contains the following inputs and outputs: an input asynchronous clock that decides the baud rate for the UART link, an asynchronous reset, the 10-bit packet from the keypad module as an input, and dataout which is the serial transmission line. As shown in Fig. 7, UART transmission is implemented using a finite state machine.

The UART finite state machine transitions on the positive edge of baudclk, which is a 115200 Hz clock. This baud rate clock must be precise, which is why it is generated by a phase-locked loop on Quartus II's Megafunction Wizard tool. Upon receipt of a pressed signal, the uart module stops idling HIGH and starts serially transmitting the 10-bit UART packet. Before transmission, the packet is synchronized to the baudclk. In addition, the packet is transmitted in a least-significant bit first format. The dataout output of this FSM results in the timing diagram in Fig. 8. Appendix B contains SystemVerilog files for both the upper level module and the phase-locked loop that generates the UART baud rate, while Appendix A provides a schematic for the `uart` module.

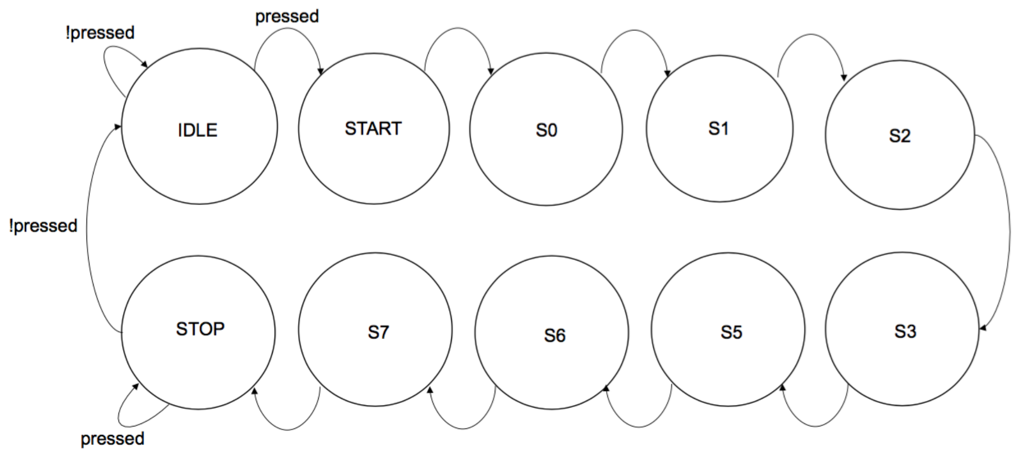


Figure 7: Finite State Machine to send UART data to the BlueSMiRF Bluetooth Module.

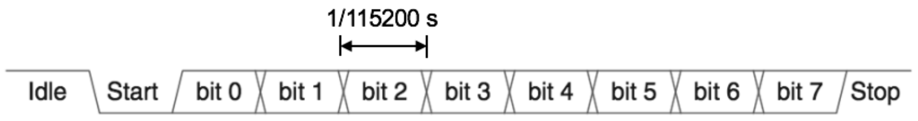


Figure 8: Timing diagram that demonstrates how the `dataout` signal is constructed, based on an 8N1 UART protocol. Modified from [3].

5 Microcontroller Design

The microcontroller design is divided into four main components. First, `EasyPIO.h` handles initializing the Pi's memory-mapped I/O, as well as setting up the SPI0 port that is used to communicate with two analog-to-digital converters. Second, `accel.h` receives acceleration data in the x, y, and z-directions from the ADCs via SPI. Third, `motion.h` contains the face-detection algorithm and the numerical integration necessary for velocity-based volume control. The `main` function is contained in `audio.c`, which selects the appropriate sound dataset (the `toneset` variable) based on Bluetooth information from the BlueSMiRF, plays a note based on the orientation information collected in `motion.h`, and writes to the appropriate pins to implement volume control. Appendix C provides all the relevant C code for the microcontroller design of the SoundCube.

5.1 SPI with the Accelerometer

SPI communication to retrieve accelerometer data is handled in the `accel.h` header file. A block diagram describing the subsystem for retrieving acceleration data from the accelerometer is shown in Fig. 9. The BMA145 accelerometer provides acceleration data in three axes, labeled `Ax`, `Ay`, and

`Az`. These wires go through two dual-channel ADCs that are then received by the SPI0 port on the Raspberry Pi. `EasyPIO.h` initializes the SPI0 port using a generic SPI protocol that sends and receives 8-bit `chars`. However, the MCP3003 ADCs provide 10 bits of serial data, which is why two calls to the `SPIsendReceive` function are made to assemble 10 bits of acceleration data from the accelerometer.

The signals that need to be received from the accelerometer (via ADCs) are voltages from 0 to 3V corresponding to acceleration in the x, y, and z-directions. In order to receive all three signals using only the SPI0 port on the Raspberry Pi, `Az` and `Ay` were connected to both channels of one MCP3002 and retrieved using time-multiplexing techniques and switching between Channels 0 and 1 on the ADC. `Ax` data was received by time-multiplexing between the Chip Select 0 and Chip Select 1 pins on the Pi's SPI0 port. Thus, a single SPI port on the Pi was able to receive three sets of acceleration values, each corresponding to a different direction in space. Another thing to note about the SPI algorithm is that some bits in the `SPI0CS` register needed to be manipulated outside of `EasyPIO.h` in order to make looping SPI calls, most notably the `TA` bit. Please refer to Appendix C for more information about the SPI implementation.

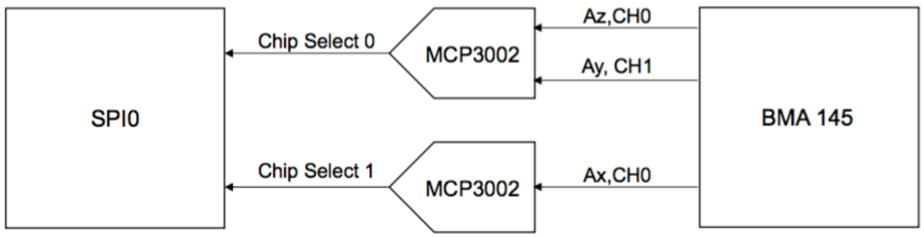


Figure 9: A block diagram representing the architecture of the SPI subsystem. Acceleration data flows from the BMA145 accelerometer to the SPI0 port on the Raspberry Pi, via two 10-bit dual channel ADCs.

5.2 Face Detection Algorithm

The algorithm that detects which side of the cube is facing up (relative to the ground or a surface) is implemented in the `motion.h` file. Voltage values corresponding to accelerations in each direction were received through SPI and used to determine the orientation of the cube. According to the BMA145 datasheet [8], each axis of the accelerometer can output a voltage corresponding to when it experiences -1g, 0g, or 1g of acceleration (please refer to Table 1). Once the accelerometer was mounted on the E80 IMU4 circuit board, the outputs increased by 0.1V. When a certain axis experienced acceleration $\geq$ g, while the other axes were explicitly not experiencing g-acceleration, that axis was used to determine the upward-facing side of the cube. Thresholds were experimentally and individually set for each axis to ensure clean transitions between cube face. These thresholds are outlined in Table 2. When a new face is set, the global variable `NEWFACE` is triggered and when the cube is oriented in between faces, the face variable is set to zero (`face = 0`) to indicate that no sound should play during face transitions. Note that there is a range of angles for which the cube should play no sound, to allow for smoother transitions between faces.

Table 2: Thresholds for the accelerometer sensor at rest.

Orientation	+x	-x	+y	-y	+z	-z
A_x	> 1.7V	< 1.49V	1.45 - 1.69V	1.45 - 1.69V	1.45 - 1.7V	1.45 - 1.7V
A_y	1.45 - 1.67V	1.45 - 1.67V	> 1.78V	< 1.49V	1.45 - 1.65V	1.45 - 1.65V
A_z	1.45 - 1.65V	1.45 - 1.65V	1.45 - 1.69V	1.45 - 1.69V	> 1.7V	< 1.35V

5.3 Volume Control Algorithm

The acceleration voltages and `face` variable are used to determine the velocity at which the cube is moving. This velocity is then used to control the volume output based on the range the velocity is in. The velocity is zeroed in several situations: if the cube is experiencing ±1g (which is most likely to happen when the cube is still, this catches a significant portion of velocity drift); if the cube is in the middle of a face transition (i.e., `face = 0`); or if noise has been integrated up such that velocity is greater than 20 (an empirically-determined cutoff). Otherwise, the value for `face` indicates which side is up and thus which acceleration should be used to calculate velocity, e.g., if `face` corresponds to ±z, then integration is done with the z-acceleration voltage. If a face change has been detected (that is, the control signal `NEWFACE = 1`), then the previous velocity is zeroed.

To determine velocity values from acceleration, Euler's method of numerical integration was utilized because it only requires storing one previous acceleration value and one previous velocity value. The current velocity is calculated like so

$$\text{velocity} = \text{previous velocity} + \alpha \times |(\text{acceleration}) - (\text{previous acceleration})| \times \mathrm{d}t$$

where dt is approximately equal to the time spent in the overall while loop (determined by the duration that a note is played), and α is an experimentally determined multiplier which gives greater resolution to the output velocity values.

The cube has six volume settings controlled by resistors connected to transistors, which are set as described in Section 2.2.1 by writing 0s or 1s to GPIO pins 16, 20, and 21 (see Figure 3). By varying which resistors the current flows through, the gain (and thus the volume) can be digitally controlled based on empirically determined thresholds for velocity. The lowest velocity was chosen to correspond to highest volume so that moving the cube would not be annoying to users.

6 Results

The SoundCube successfully plays different notes depending on which face is up. It is wireless, and it also implements limited volume control based on motion in the z-direction. When turning the cube, there are breaks of silence to allow for nicer transitions between one note and the next. This also allows a user to turn the cube back and forth between a single note and silence to get a repeated note that is not sustained. The SoundCube also responds immediately to a keypress, changing the sound dataset as long as the cube is within communication range of the BlueSMiRF.

If not in range, the cube will continue to play the last sound dataset selected by the user. The SoundCube is completely wireless and portable, and the portable battery power lasts several hours.

6.1 Room for Improvement

The face detection algorithm was written to be simple and understandable, however it is not as robust as it could be. When the cube is held at an angle which triggers a `face = 0` output, the noise in the voltage values from the accelerometer causes the sound to turn on and off rapidly. This makes unpleasant sounds not dissimilar to croaking. One idea to remedy this was to implement hysteresis. Unfortunately this solution required a lot of experimentation to determine the appropriate outer and inner thresholds that would give smooth transitions for each face. Hysteresis was implemented for one face but there was not enough time left to finish the implementation.

Numerical integration of acceleration was also done very simply. Unprocessed voltages from the BMA145 were used to determine velocity data instead of converting to acceleration values, which meant values weren't exact but relative behavior was still as expected. Additionally, the velocity was calculated using Euler's Method to integrate "acceleration," but Euler's method is not very good, as it only takes in values from the previous iteration and doesn't store data from any farther back.

6.2 Changes from Original Deliverables

Originally, the deliverables for the SoundCube stated that an Adafruit Feather M0 Bluefruit LE would replace the Raspberry Pi as the microcontroller in this project, as the Feather is significantly smaller and requires less power than the Pi. However, the Bluetooth module on the Feather is a Bluetooth Low Energy module, which utilizes a completely different communication protocol than the Classic Bluetooth protocol performed by the BlueSMiRF. Thus, there was no way for the chosen Bluetooth devices to communicate with each other.

The first alternative considered was to simply utilize another BlueSMiRF that could receive data from the first BlueSMiRF and would interface with the Feather via a UART protocol. However, as explained in section 3.1, the second BlueSMiRF module was not able to transmit data to the microcontroller. Given these difficulties, and since that deliverable could not be perfectly met anyway, a Raspberry Pi was used instead of the Feather since the Pi has a classic Bluetooth module and had been previously used in several labs.

Another departure from the initial deliverables is using the E80 IMU4 with the BMA145 accelerometer rather than using an LSM6DS3 gyroscope/accelerometer sensor on the Sparkfun 6 degrees of freedom breakout board. The LSM6DS3 was initially chosen for its low power consumption and SPI serial interface. However, the Raspberry Pi was never able to receive any data back from the LSM6DS3, even the `WHO_AM_I` register. It was determined to be more important to get a functioning cube than to work specifically with the LSM6DS3, so the E80 IMU4 was used instead.

References

[1] "ASCII-Code." *The Extended ASCII Table.* [Online]. Available: http://www.ascii-code.com/

[2] M. Cardenas, and E. Spjut, "Accelerometer and Gyroscope Calibration." *E80 Spring 2016,* published online: Spring 2016.

[3] D. M. Harris and S. L. Harris, "I/O Systems," in *Digital Design and Computer Architecture: ARM Edition*, 1st ed. Burlington, MA: Morgan Kaufmann, 2015, ch. 9, pp. 531.e9.

[4] D. M. Harris, and M. Spencer, "Lab 3: Keypad Scanner." *E155: Microprocessor-Based Systems,* published online: Fall 2016.

[5] D. M. Harris, and M. Spencer, "Lab 6: Internet of Things." *E155: Microprocessor-Based Systems,* published online: Fall 2016.

[6] "LM386 Low Voltage Audio Power Amplifier," *LM386 datasheet*, National Semiconductor Corporation. 2000.

[7] Roving Networks, Los Gatos, CA, USA. *Roving Networks.* (2009) [Online]. https://www.sparkfun.com/datasheets/Wireless/Bluetooth/rn-bluetooth-um.pdf.

[8] "Triaxial, Analog Acceleration Sensor." *BMA145 datasheet*, Bosch Sensortec. Feb. 2010.

[9] "Voltage Dividers," SparkFun. [Online]. Available: https://learn.sparkfun.com/tutorials/voltage-dividers

[10] "2.7V Dual Channel 10-Bit A/D Converter with SPI Serial Interface," *MCP3002 datasheet*, Microchip Technology Inc. 2011.

Parts List

Part	**Source**	**Part Number**	**Price**
Cyclone 4 FPGA	Stockroom		
Raspberry Pi 3.0	Stockroom		
BlueSMiRF	Micro-Ps Lab		
E80 IMU4 Breakout Board	E80 Lab		
Speaker	Micro-Ps Lab		
5V Voltage Regulator	E80 Lab	LM7805	
Analog to Digital Converter	Micro-Ps Lab	MCP3002	
9V Battery	E80 Lab		
IXCC 3400 mA/hours Powerbank	Amazon		$14.99
12"x36"x1/8" Acrylic Frosted Plexiglass	Amazon		$17.16

Appendices

Appendix A - Detailed FPGA Schematics

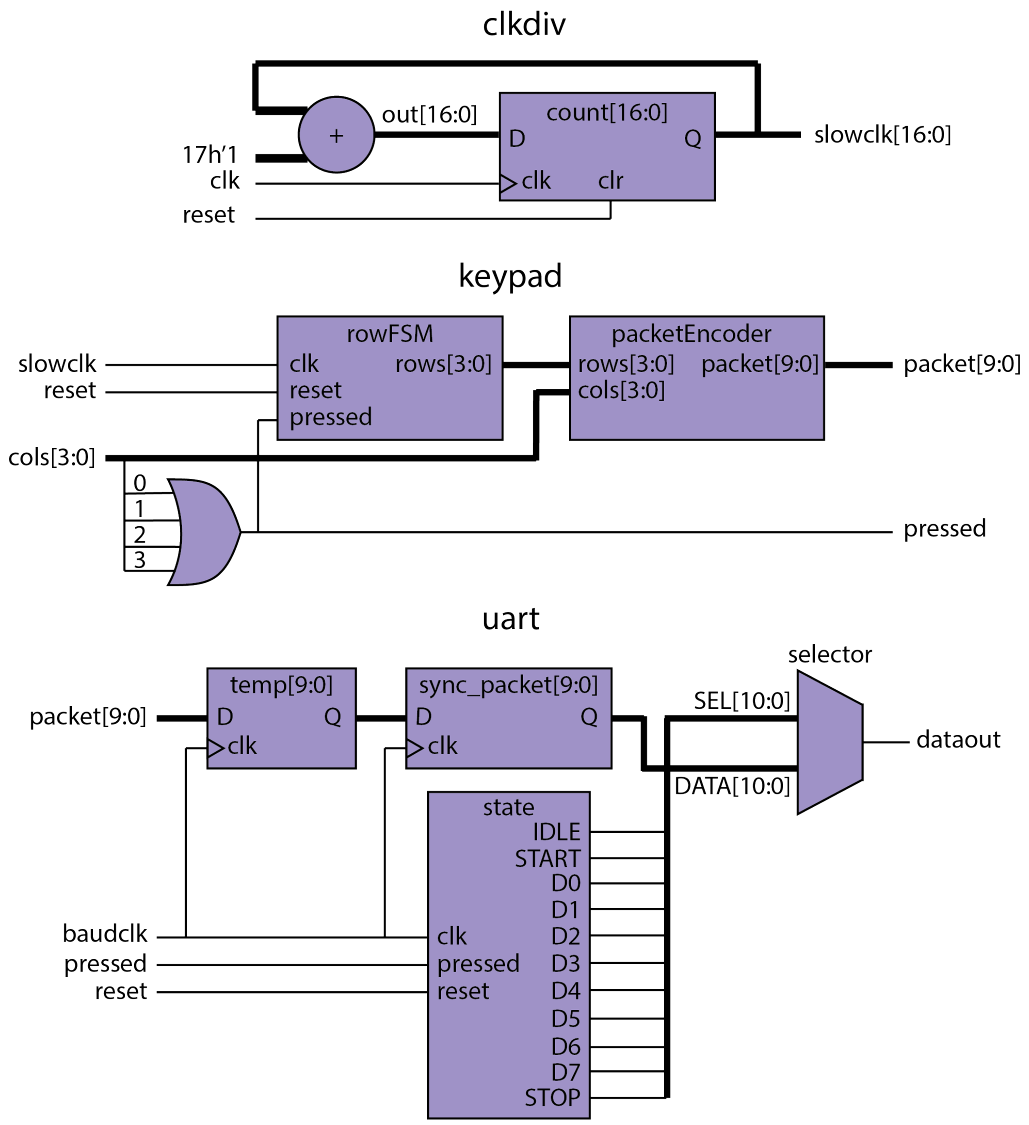


Figure 10: Schematics of FPGA modules. The `clkdiv` module generates the `slowclk` that controls the row switching in the `keypad` module. The `keypad` module controls the row outputs and takes the column inputs to determine the key that's pressed and the UART packet to be sent. The `uart` module synchronizes the the `packet` to the `baudclk` and controls the UART FSM that sends data to the BlueSMiRF.

Appendix B - Verilog Files for FPGA Hardware

B.1 finalproject.sv

```
/*
 * finalproject.sv
 * Authors: Sakshi Shah, sashah@g.hmc.edu
 *          Marisol Beck, mbeck@g.hmc.edu
 * Last modified: December 8, 2016
 *
 * This SystemVerilog file synthesizes the FPGA hardware to perform
 * a keypad scanner circuit and transmit the data of which key is pressed
 * to a BlueSMiRF bluetooth module via a UART communication protocol
 */

/*
 * finalproject module
 * Authors: Sakshi Shah, sashah@g.hmc.end
 *          Marisol Beck, mbeck@g.hmc.edu
 * Last Modified: December 8, 2016
 *
 * The upper level module for the FPGA hardware. A phase-locked loop is used
 * to create a UART baudrate clk with a baudrate of 115200 bps
 * A keypad scanner circuit takes in keypress input data and performs
 * switch debouncing. This data is assembled in the form of a 8N1 UART packet
 * and is outputted from the MuddPi utility board.
 */
module finalproject(input logic clk, reset,  // Utility board clk, asynchronous reet
                    output logic [3:0] rows, // For keypad scanner
                    input logic [3:0] cols,  // For keypad scanner
                    output logic dataout);   // UART TX data

    // Initializing internal wires, necessary for other modules
    logic baudrate, locked, slowclk, pressed;
    logic [9:0] packet; // Initializing 10-bit 8N1 UART packet

    // baudclk module creates a baudrate of 115200 bps using a PLL
    baudclk baudclk(reset, clk, baudrate, locked);
    // clkdiv module
    clkdiv  clkdiv(clk, reset, slowclk);
    keypad  keypad(slowclk, reset, cols, pressed, rows, packet);
    uart    uart(baudrate, reset, pressed, packet, dataout);
endmodule


/*
```

```
 * uart module
 * Authors: Sakshi Shah, sashah@g.hmc.end
 *          Marisol Beck, mbeck@g.hmc.edu
 * Last Modified: December 8, 2016
 *
 * Takes in an input baudclk, generated by a phase-locked loop
 * in the baudclk module, and a control signal that indicates
 * a key has been pressed on the keypad. Outputs a data packet
 * of 10 bits that contains information about which key has been pressed
 * in LSB first format.
 * Each bit in the data packet is output at a rate of 115200 bps.
 * Data packet format (LSB first): START(0), Data[0],...,Data[7],STOP(1)
 */
module uart(input logic baudclk, reset, pressed, // pressed comes from keypad module
            input logic [9:0] packet,            // 10 bit packet assembled in keypad
            output logic dataout);               // 10 bit packet transmitted bit by bit

logic [9:0] temp, sync_packet;                   // Asynchronous keypad data synchronized to baudclk

// Before doing anything with packet, synchronize it to the baudclk
always_ff@(posedge baudclk)
    begin
        temp <= packet;
        sync_packet <= temp;
    end

// A finite state machine handles the UART transmission
typedef enum logic [4:0] {IDLE, START, D0, D1, D2, D3, D4, D5, D6, D7, STOP} statetype;
statetype state, nextstate;

// FSM State register
always_ff @(posedge baudclk, posedge reset)
    if (reset) state <= IDLE;
    else       state <= nextstate;

// Nextstate logic
always_comb
    case(state)
        IDLE: if (pressed)  nextstate = START; // Stop idling when a keypress
              else          nextstate = IDLE;  // has been registered
        START:              nextstate = D0;
        D0:                 nextstate = D1;
        D1:                 nextstate = D2;
        D2:                 nextstate = D3;
        D3:                 nextstate = D4;
        D4:                 nextstate = D5;
        D5:                 nextstate = D6;
        D6:                 nextstate = D7;
        D7:                 nextstate = STOP;
        STOP: if (!pressed) nextstate = IDLE; // Only move off STOP
              else          nextstate = STOP; // if no button pressed
        default:            nextstate = IDLE;
    endcase

// Output logic
// UART data is transmitted LSB first
always_comb
    case(state)
        IDLE:     dataout = 1'b1; // idling high
        START:    dataout = sync_packet[0];
        D0:       dataout = sync_packet[1];
        D1:       dataout = sync_packet[2];
        D2:       dataout = sync_packet[3];
        D3:       dataout = sync_packet[4];
        D4:       dataout = sync_packet[5];
        D5:       dataout = sync_packet[6];
        D6:       dataout = sync_packet[7];
        D7:       dataout = sync_packet[8];
        STOP:     dataout = sync_packet[9];
        default:  dataout = 1'b1; // default is idle
    endcase

endmodule

/*
 * keypad module
 * Authors: Sakshi Shah, sashah@g.hmc.end
 *          Marisol Beck, mbeck@g.hmc.edu
 * Last Modified: December 8, 2016
 *
 * This module constructs a keypad scanner circuit that powers
 * each row sequentially and scans each column to detect a keypress.
 * When a keypress has been detected, we generate a control signal "pressed"
 * and also construct a 10 bit 8N1 UART packet that contains a start bit, 8 bits
 * of data, and a stop bit. The 8 bits of data correspond to the ASCII
 * value (in binary) of the character that represents the pressed key.
 */
module keypad(input logic slowclk, reset,
              input logic [3:0] cols,
              output logic pressed,
              output logic [3:0] rows,
              output logic [9:0] packet);

    // Scanning FSM trasnsitions on a slowclk to debounce asynchronous keypad data
    always_ff@(posedge slowclk or posedge reset)
        if (reset) begin
            pressed <= 0;
            rows <= 4'b0001;
        end else if (~(|cols)) begin      // No keypress detected for this col, keep scanning
            pressed <= 0;
            rows <= {rows[0], rows[3:1]}; // Shift rows right to power the next row
        end else if (~pressed) begin
            pressed <= 1;                 // pressed is the bitwise OR of all the columns
        end                               // Generates a pulse upon keypress
        // otherwise wait until all keys are released before continuing

        // Conversion from input keypad data to 10 bit UART packet
        // The numbers corresponding to keypad inputs are converted to ASCII
        // Because UART data is transmitted to a BlueSMiRF, which sends data
        // in ASCII form.
        // Packet looks like this: stop_ascii_start
        always_comb
            case({rows,cols})
                8'b0001_0001: packet <= 10'b1_0011_0001_0; // 1
                8'b0001_0010: packet <= 10'b1_0011_0010_0; // 2
                8'b0001_0100: packet <= 10'b1_0011_0011_0; // 3
                8'b0001_1000: packet <= 10'b1_0100_0001_0; // A
                8'b0010_0001: packet <= 10'b1_0011_0100_0; // 4
                8'b0010_0010: packet <= 10'b1_0011_0101_0; // 5
                8'b0010_0100: packet <= 10'b1_0011_0110_0; // 6
                8'b0010_1000: packet <= 10'b1_0100_0010_0; // B
                8'b0100_0001: packet <= 10'b1_0011_0111_0; // 7
                8'b0100_0010: packet <= 10'b1_0011_1000_0; // 8
                8'b0100_0100: packet <= 10'b1_0011_1001_0; // 9
                8'b0100_1000: packet <= 10'b1_0100_0011_0; // C
                8'b1000_0001: packet <= 10'b1_0100_0101_0; // E
                8'b1000_0010: packet <= 10'b1_0011_0000_0; // 0
                8'b1000_0100: packet <= 10'b1_0100_0110_0; // F
                8'b1000_1000: packet <= 10'b1_0100_0100_0; // D
                default:      packet <= 10'b1_0011_0000_0; // default is 0
            endcase
endmodule

/*
 * clkdiv module
 * This module the clk by 2^17z: 20MHz/2^17 = 150 Hz
 * in order to perform switch debouncing on the asynchronous keypad inputs
 */
module clkdiv(input logic clk, reset,
                  output logic slowclk);
    logic [16:0] count;

    // This synthesizes to an asynchronously resettable counter.
    // The reset line is tied to the global reset/reset line of the FPGA
    always_ff@(posedge clk or posedge reset)
        if (reset) count <= 17'b0;
        else count <= count + 17'b1;

    assign slowclk = count[16];
endmodule
```

B.2 baudclk.v

```
// megafunction wizard: %ALTPLL%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altpll

// ============================================================
// File Name: baudclk.v
// Megafunction Name(s):
//          altpll
//
// Simulation Library Files(s):
//          altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 15.0.2 Build 153 07/15/2015 SJ Web Edition
// ************************************************************


//Copyright (C) 1991-2015 Altera Corporation. All rights reserved.
//Your use of Altera Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Altera Program License
//Subscription Agreement, the Altera Quartus II License Agreement,
//the Altera MegaCore Function License Agreement, or other
//applicable license agreement, including, without limitation,
//that your use is for the sole purpose of programming logic
//devices manufactured by Altera and sold by Altera or its
//authorized distributors.  Please refer to the applicable
//agreement for further details.
 n

// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module baudclk (
    areset,
    inclk0,
    c0,
    locked);

    input     areset;
    input     inclk0;
    output    c0;
    output    locked;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri0      areset;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [0:0] sub_wire2 = 1'h0;
    wire [4:0] sub_wire3;
    wire  sub_wire5;
    wire  sub_wire0 = inclk0;
    wire [1:0] sub_wire1 = {sub_wire2, sub_wire0};
    wire [0:0] sub_wire4 = sub_wire3[0:0];
    wire  c0 = sub_wire4;
    wire  locked = sub_wire5;

    altpll  altpll_component (
                .areset (areset),
                .inclk (sub_wire1),
                .clk (sub_wire3),
                .locked (sub_wire5),
                .activeclock (),
                .clkbad (),
                .clkena ({6{1'b1}}),
                .clkloss (),
                .clkswitch (1'b0),
                .configupdate (1'b0),
                .enable0 (),
                .enable1 (),
                .extclk (),
                .extclkena ({4{1'b1}}),
                .fbin (1'b1),
                .fbmimicbidir (),
                .fbout (),
                .fref (),
                .icdrclk (),
                .pfdena (1'b1),
                .phasecounterselect ({4{1'b1}}),
                .phasedone (),
                .phasestep (1'b1),
                .phaseupdown (1'b1),
                .pllena (1'b1),
                .scanaclr (1'b0),
                .scanclk (1'b0),
                .scanclkena (1'b1),
                .scandata (1'b0),
                .scandataout (),
                .scandone (),
                .scanread (1'b0),
                .scanwrite (1'b0),
                .sclkout0 (),
                .sclkout1 (),
                .vcooverrange (),
                .vcounderrange ());
    defparam
        altpll_component.bandwidth_type = "AUTO",
        altpll_component.clk0_divide_by = 3125,
        altpll_component.clk0_duty_cycle = 50,
        altpll_component.clk0_multiply_by = 9,
        altpll_component.clk0_phase_shift = "0",
        altpll_component.compensate_clock = "CLK0",
        altpll_component.inclk0_input_frequency = 25000,
        altpll_component.intended_device_family = "Cyclone IV E",
        altpll_component.lpm_hint = "CBX_MODULE_PREFIX=baudclk",
        altpll_component.lpm_type = "altpll",
        altpll_component.operation_mode = "NORMAL",
        altpll_component.pll_type = "AUTO",
        altpll_component.port_activeclock = "PORT_UNUSED",
        altpll_component.port_areset = "PORT_USED",
        altpll_component.port_clkbad0 = "PORT_UNUSED",
        altpll_component.port_clkbad1 = "PORT_UNUSED",
        altpll_component.port_clkloss = "PORT_UNUSED",
        altpll_component.port_clkswitch = "PORT_UNUSED",
        altpll_component.port_configupdate = "PORT_UNUSED",
        altpll_component.port_fbin = "PORT_UNUSED",
        altpll_component.port_inclk0 = "PORT_USED",
        altpll_component.port_inclk1 = "PORT_UNUSED",
        altpll_component.port_locked = "PORT_USED",
        altpll_component.port_pfdena = "PORT_UNUSED",
        altpll_component.port_phasecounterselect = "PORT_UNUSED",
        altpll_component.port_phasedone = "PORT_UNUSED",
        altpll_component.port_phasestep = "PORT_UNUSED",
        altpll_component.port_phaseupdown = "PORT_UNUSED",
        altpll_component.port_pllena = "PORT_UNUSED",
        altpll_component.port_scanaclr = "PORT_UNUSED",
        altpll_component.port_scanclk = "PORT_UNUSED",
        altpll_component.port_scanclkena = "PORT_UNUSED",
        altpll_component.port_scandata = "PORT_UNUSED",
        altpll_component.port_scandataout = "PORT_UNUSED",
        altpll_component.port_scandone = "PORT_UNUSED",
        altpll_component.port_scanread = "PORT_UNUSED",
        altpll_component.port_scanwrite = "PORT_UNUSED",
        altpll_component.port_clk0 = "PORT_USED",
        altpll_component.port_clk1 = "PORT_UNUSED",
        altpll_component.port_clk2 = "PORT_UNUSED",
        altpll_component.port_clk3 = "PORT_UNUSED",
        altpll_component.port_clk4 = "PORT_UNUSED",
        altpll_component.port_clk5 = "PORT_UNUSED",
        altpll_component.port_clkena0 = "PORT_UNUSED",
        altpll_component.port_clkena1 = "PORT_UNUSED",
        altpll_component.port_clkena2 = "PORT_UNUSED",
        altpll_component.port_clkena3 = "PORT_UNUSED",
        altpll_component.port_clkena4 = "PORT_UNUSED",
        altpll_component.port_clkena5 = "PORT_UNUSED",
        altpll_component.port_extclk0 = "PORT_UNUSED",
        altpll_component.port_extclk1 = "PORT_UNUSED",
        altpll_component.port_extclk2 = "PORT_UNUSED",
        altpll_component.port_extclk3 = "PORT_UNUSED",
        altpll_component.self_reset_on_loss_lock = "OFF",
        altpll_component.width_clock = 5;

endmodule
```

Appendix C - Software

C.1 audio.c

```
/* SOUNDCUBE PROJECT
 *
 * audio.c
 * Sakshi Shah   sashah@g.hmc.edu
 * Kathleen Kohl kkohl@g.hmc.edu
 * Marisol Beck  mbeck@g.hmc.edu
 * Created: November 19, 2016
 * Modified: December 8, 2016
 *
 * The main file which initializes the SoundCube and runs the infinite
 * while loop which runs the program for the cube.
 *
 * This includes modified functionality from K. Kohl's files for
 * E155 Lab 5.
 *
 */

#include "EasyPIO.h"
#include "audio.h"
#include "accel.h"
#include "motion.h"


int main() {

    ////////////////////// INITIALIZATION ////////////////////////////

    /* The initialization of the SoundCube requires the following steps:
     *
     * - Initialize the memory mapping, IO pins, and SPI of
     * the Raspberry Pi, as well as setting the pins required
     * for volume and sound control.
     *
     * - Open and adjust settings of the file reading in Bluetooth
     * data from the FPGA via a BlueSMiRF (which must be connected
     * to separately).
     *
     * - Set initial values for the face, toneset, velocity,
     * and accelerations.
     *
     * While we initially thought to put this in a header file,
     * it is mostly defining local variables which get used in a
     * while loop, so it needs to live here.
     */

    // Initialize memory mapping, IO pins, SPI
    pioInit();
    int frequency = 122000;
    int cpha = 0b0;
    int cpol = 0b0;
    int cs = 0b00; // set chip select to CE0 for default
    spiInit(frequency, cpha, cpol, cs);

    // Open the Bluetooth communication port as a readable file
    // Create a file structure to manipulate the file as a C object
    FILE *fp;
    // The bluetooth file to read in
    fp = fopen("/dev/rfcomm0","r");

    /* The bluetooth file is a "blocking" file, which means
     * that if we try to read in from the file when there hasn't
     * been a new keypress, it gets stuck and won't read anymore.
     * This sets some settings which turns it into a "non-blocking"
     * file, which allows us to "read" from it even when there is no
     * new character, and still catches the next keypress.
     * For more information, visit http://tinyurl.com/non-blocking-file
     *
     * Thank you to Eric Mueller for pointing us towards this fix!
     */
    int file_descriptor = fileno(fp);
    int flags = fcntl(file_descriptor, F_GETFL, 0);
    fcntl(file_descriptor, F_SETFL, flags | O_NONBLOCK);

    // Define the pins that configure volume changes
    int volumePin1 = 21; // controls transistor for the larger resistor
    int volumePin2 = 20; // controls transistor for the medium resistor
    int volumePin3 = 16; // controls transistor for the smallest resistor
    int volumeLevel = 6; // start on default volume
    // set default volume
    ChangeVolume(volumeLevel, volumePin1, volumePin2, volumePin3);

    // Initialize default values for face and toneset
    toneset *out = &majorscale; // pointer to a toneset, default majorscale
    int face = 0;
    int prevFace = 0;
    int changeFace = 0;         // Face hasn't changed
    int soundPin = 26;          // Write to pin 26 on the Pi
```

```
toneset *temp;              // output of GetKeypadChar
int duration = 5000;        // duration in microseconds

// Initialize default values for acceleration
float zaccelV = zAccelV();
float yaccelV = yAccelV();
float xaccelV = xAccelV();
float vel;
float velPrev = 0.0;
float xaccelVPrev = 1.5;
float yaccelVPrev = 1.5;
float zaccelVPrev = 1.5;

////////////////////// MAIN WHILE LOOP ///////////////////////////

/*
 * The main functionality is in an infinite while loop.
 *
 * It reads in the latest keypress data via bluetooth
 * to pick a toneset. It also reads in acceleration
 * data to determine which face is up, and to calculate
 * the velocity in order to set the volume of the
 * output sound.
 *
 */
while(1) {
    // As long as there is a viable character to get
    // get a new character and so get a new toneset
    temp = GetKeypadChar(fp);

    // If a viable character was inputted, the first
    // value in the array of the toneset will be nonzero
    // (i.e., it didn't choose the "noTone" set).
    if (temp->notes[0] != 0) {
         out = temp;
    }

    // Read in acceleration voltage values
    // Uses functions in accel.h
    xaccelV = xAccelV();
    yaccelV = yAccelV();
    zaccelV = zAccelV();
    // Figure out what face to play
    face = newFace(xaccelV, yaccelV, zaccelV, prevFace);
    prevFace = face;
```

```
        // Determine velocity
        vel = velocity(velPrev, xaccelV, yaccelV, zaccelV, xaccelVPrev,
                       yaccelVPrev, zaccelVPrev, face);
        xaccelVPrev = xaccelV;
        yaccelVPrev = yaccelV;
        zaccelVPrev = zaccelV;
        velPrev = vel;

        // Assign a new volume level based on velocity
        volumeLevel = volume(vel, face);

        if (volumeLevel == 0) {
            PlayNote(noTone, face, duration, soundPin);
        } else {
            ChangeVolume(volumeLevel, volumePin1, volumePin2, volumePin3);
            PlayNote(*out, face, duration, soundPin);
        }

    } // end of the while loop

    return 0;
}


/************************************************************************
                           HELPER FUNCTIONS
 ************************************************************************/


/* A function which reads in keypad data from the file coming in via
 * Bluetooth. Reads a character from the file and picks a toneset
 * based on the data. If a user choose a key that does not correspond
 * to a toneset, or if no new key has been pressed, we choose the
 * "noTone" toneset.
 *
 * Input:
 *    *fp: the address in memory of the file which contains the
 *         Bluetooth input.
 *
 * Output:
 *    out: a pointer to the associated toneset.
 *
 */
toneset* GetKeypadChar(FILE *fp) {
    // fgetc gets the latest character from the file specified by fp
    char key;
    key = fgetc(fp);

    // Pick a toneset based on the key pressed.
    // This is done with if/else rather than case statements to
    // avoid breaks.
    toneset* out;

    if  (key == 'A') {
        out = &majorscale;
    } else if (key == 'B') {
        out = &majorpent;
    } else if (key == 'C') {
        out = &minorpent;
    } else if (key == 'D') {
        out = &jingle;
    } else {
        out = &noTone;
    }

    return out;
}


/* A function which plays an individual note in a specific
 * toneset for a specified duration. The note played depends
 * on the input face (which should be the face which is currently
 * up).
 *
 * Inputs:
 *    set: the toneset to play from, based on keypad input
 *    face: which face of the cube is up
 *    duration: duration of the note in microseconds
 *    pin: which pin to write to on the Pi
 *
 */
void PlayNote(toneset set, int face, unsigned int duration, int pin) {

    // Ensure the pin we want to write to is an output
    pinMode(pin, OUTPUT);

    unsigned int freq; // pitch in Hz that we want to play
    // If we haven't detected a face, play no sound
    if (face == 0) {
        freq = 0;
    } else {
        // access face-1 because the faces are numbered 1-6
        freq = set.notes[face-1];
    }

    // If there is a nonzero frequency, play a note
    // Otherwise treat it as a musical rest
    if (freq != 0) {
        // Half the period of the pitch (in ms)
        unsigned int halfPeriod = 1000000 / (2*freq);
        // The number of cycles of a sound wave with freq
        // that should play within the specified duration
        unsigned int numCycles = duration / (2*halfPeriod);

        unsigned int j; // index to play note for numCycles cycles
        for (j = 0; j < numCycles; j++) {
        // Constructing a sound wave of appropriate frequency
        // by alternating writing 1s and 0s to pin
        // with delays corresponding to the frequency
        digitalWrite(pin, 1);
        delayMicros(halfPeriod);
        digitalWrite(pin, 0);
        delayMicros(halfPeriod);
        }
    // else no pitch to play, just rest
    } else {
        delayMicros(duration);
    }
}

/* A function that allows us to control the volume based on the
 * z-acceleration the cube experiences.
 *
 * Inputs:
 *    volumeLevel: how loud the output should be. This input should
 *                 be an int between 1 and 6.
 *    volumePin1: the GPIO pin which controls the base terminal of the
 *                largest resistor in the circuit. In our case, this is
 *                a 2k ohm resistor.
 *    volumePin2: the GPIO pin which controls the base terminal of the
 *                middle resistor in the circuit. In our case, this is
 *                a 1k ohm resistor.
 *    volumePin3: the GPIO pin which controls the base terminal of the
 *                smallest resistor in the circuit. In our case, this is
 *                a 510 ohm resistor.
 *
 */
void ChangeVolume(int volumeLevel, int volumePin1, int volumePin2, int volumePin3) {

    // Ensure that the pins we want to write to are outputs
    pinMode(volumePin1, OUTPUT);
    pinMode(volumePin2, OUTPUT);
    pinMode(volumePin3, OUTPUT);

    int index = volumelevel - 1;

    /* volumeControl pins is an array defined in audio.h.
     * It contains the values to write to volumePin1, 2,
     * and 3 respectively based on an input volume level.
     * The index in the array corresponds to (volume level - 1).
     */
    digitalWrite(volumePin1, volumeControlPins[index][0]);
    digitalWrite(volumePin2, volumeControlPins[index][1]);
    digitalWrite(volumePin3, volumeControlPins[index][2]);

}
```

C.2 accel.h

```
/* SOUNDCUBE PROJECT
 *
 * accel.h
 * Sakshi Shah   mbeck@g.hmc.edu
 * Kathleen Kohl kkohl@g.hmc.edu
 * Marisol Beck  sashah@g.hmc.edu
 * Created: November 30, 2016
 * Modified: December 8, 2016
 *
 * The header file which encodes the SPI protocols required to extract
 * the accelerometer data from our MCP3002 10-bit ADC.
 *
 * This code runs best with an SPI frequency of 122 kHZ, CPHA = 0, and
 * CPOL = 0.
 */

#ifndef ACCEL
#define ACCEL

/* A general function which gives the voltage from an accelerometer
 * which is converted from an analog to digital signal through SPI
 * of a 10-bit ADC.
 *
 * Inputs:
 *     spiData: a ten bit number which tells the ADC what data to send.
 *              The normal format should be: 0b011_x_1000 where x is
 *              the channel select on the ADC (0 or 1)
 *     chipSelect: a two bit number, choosing either CE0 (0b00) or
 *                 CE1 (0b01) on the Pi
 *
 * Output:
 *     accelerometerVoltage: a float that gives the accelerometer voltage
 *
 * Note that this function simply outputs the voltage coming out of
 * the accelerometer and not the specific acceleration it corresponds
 * to. However, since the voltage scales linearly with acceleration and
 * this code doesn't interface with anything that requires physically
 * accurate values, the conversion is unnecessary.
 */
float AccelV(int spiData, int chipSelect) {

    SPI0CSbits.CLEAR = 0b11;
    SPI0CSbits.CLEAR = 0b00;
    SPI0CSbits.CS = chipSelect;
    SPI0CSbits.TA = 1;

    /* The ADC provides 10 bits of data but spiSendReceive only
     * takes in chars (8 bits). In each acceleration call, we
     * configure the ADC to break up the data and send it
     * sequentially.
     * See Section 6.1 of the MCP3002 datasheet for reference.
     */
    int A1 = (int) spiSendReceive(spiData);
    int A2 = (int) spiSendReceive(0b00000000);
    SPI0CSbits.TA = 0;

    A1 &= 0b11; // only want the last 2 bits of A1
    A1 = A1 << 8; //shift it up for bits 8, 9
    int A = A1 | A2; // make it {A1}{A2}

    float Vin = 5.0*A/1024.0;
    return Vin;
}

/* A function which gives the voltage from a z-accelerometer
 * which is converted from an analog to digital signal through SPI
 * of a 10-bit ADC.
 *
 * Output:
 *     zaccelVoltage: float that gives the z-accelerometer voltage
 */
float zAccelV() {

    // z accelerometer should be on CH0 of an ADC connected to CE0
    return AccelV(0b01101000, 0b00);
}

/* A function which gives the voltage from a y-accelerometer
 * which is converted from an analog to digital signal through SPI
 * of a 10-bit ADC.
 *
 * Output:
 *     zaccelVoltage: float that gives the y-accelerometer voltage
 */
float yAccelV() {

    // y accelerometer should be on CH1 of an ADC connected to CE0
    return AccelV(0b01111000, 0b00);
```

```
}

/* A function which gives the voltage from an x-accelerometer
 * which is converted from an analog to digital signal through SPI
 * of a 10-bit ADC.
 *
 * Output:
 *     zaccelVoltage: float that gives the x-accelerometer voltage
 */
float xAccelV() {

    // x accelerometer should be on CH0 of an ADC connected to CE1
    return AccelV(0b01101000, 0b01);
}

#endif
```

C.3 motion.h

```
/* SOUNDCUBE PROJECT
 *
 * motion.h
 * Marisol Beck   mbeck@g.hmc.edu
 * Kathleen Kohl  kkohl@g.hmc.edu
 * Sakshi Shah    sashah@g.hmc.edu
 * Created: November 30, 2016
 * Modified: December 8, 2016
 *
 * The header file which encodes the motion of the SoundCube.
 * It utilizes input accelerations to calculate the velocity
 * and thus to determine the volume the output sound should be
 * played at, as well as to determine which side of the cube
 * is facing up.
 *
 * The acceleration being felt by the BMA145 in the cube
 * outputs approximately the following voltages:
 * <-1g  ->  <1.3V
 *  -1g  ->  1.3V
 *   0g  ->  1.6V
 *   1g  ->  1.9V
 *  >1g  ->  >1.9V
 *
 * We use thresholds to say we're "close enough" to a stable
 * orientation. Additionally, each axis is calibrated slightly
 * differently, so individual thresholds were set empirically.
 */

#ifndef MOTION
#define MOTION

#include <math.h> // in order to use fabs (float absolute value)

// dt is one unit of time to perform the numerical integration
#define DT    0.005

// Control signal to detect that a new face is up
int NEWFACE = 0;

/*
 * A function which utilizes Euler's method to find the approximate
 * current velocity of the cube in the horizontal axis of the user of
 * the cube. The velocity resets to 0.0 in the following cases:
```

```
 *     - the cube is experiencing "no face" - i.e., there is not enough
 *       data to determine which face is facing up
 *     - the cube is experiencing +/- g acceleration, which is most likely
 *       to occur when the cube is sitting on a surface.
 *
 * This does not account for a user moving the cube at constant or near
 * constant velocity. However, this was a conscious UI choice so that
 * users can hold the cube and have it stay at a constant volume unless
 * they move it up and down more decisively.
 *
 * Inputs:
 *     velPrev: the previous calculated velocity
 *     accelx: current x-acceleration
 *     accely: current y-acceleration
 *     accelz: current z-acceleration
 *     accelPrevX: previous x-acceleration
 *     accelPrevY: previous y-acceleration
 *     accelPrevZ: previous z-acceleration
 *     face: the side currently facing up
 *
 * Output:
 *     vel: the newly calculated velocity
 *
 */
float velocity(float velPrev, float accelx, float accely, float accelz,
               float accelPrevX, float accelPrevY, float accelPrevZ,
               int face) {
    float vel;
    float accel;
    float accelPrev;
    float dt = DT; // from the constant above

    // If we detect a face change, set previous velocity to 0
    // to restart the integration calculation.
    if (NEWFACE) {
        velPrev = 0.0;
    }

    // If the up-face is +z or -z, integrate z-acceleration
    if (face == 1 || face == 6) {
        accel = accelz;
        accelPrev = accelPrevZ;

    // If the up-face is +y or -y, integrate y-acceleration
    } else if (face == 2 || face == 5) {
        accel = accely;
        accelPrev = accelPrevY;

    // If the up-face is +x or -x, integrate x-acceleration
    } else {
        accel = accelx;
        accelPrev = accelPrevX;
    }

    float accelDiff; // variable to hold accel-accelPrev

    /* Refresh the velocity to be 0 when:
     * - the cube is experiencing +/- g velocity, which we assume
     *   occurs most often when the cube is still (rather than moving
     *   at constant velocity).
     * - the cube is in the middle of a face transition (that is, it
     *   is "no face," or face = 0).
     * - the previous velocity has drifted too much
     */
    if ( face == 0 ||                                   // no face or
         (accel     > 1.25 && accel     < 1.35 &&       // it's experiencing -g accel
          accelPrev > 1.25 && accelPrev < 1.35 ) ||     // or
         (accel     > 1.85 && accel     < 1.97 &&       // it's experiencing g accel
          accelPrev > 1.85 && accelPrev < 1.97 ) ||     // or
          velPrev > 20.0) {                             // the velocity drifted
        vel = 0.0;
    // If none of the above cases happened, calculate velocity:
    } else {
        float accelDiff = accel - accelPrev;
        // The 10000x multiplier was determined experimentally in order
        // to have greater resolution on the velocity values.
        vel = velPrev + 10000.0 * fabs(accelDiff) * dt;
    }
    return vel;
}


/*
 * A function which picks a volume based on the velocity range. Higher
 * velocities correspond to lower volumes and vice versa.
 *
 *
 * Inputs:
 *     vel: the current velocity
 *     face: the current side facing up, used to ensure that there
 *           is a face selected.
 *
 * Output:
 *     volume: the selected volume level
 */
int volume(float vel, int face) {

    int volume;

    // This is done with if/else rather than case statements to
    // avoid breaks.
    if ( face == 0 ) {
        volume = 0;
    } else if ( vel <= 1.0 ) {
        volume = 6;
    } else if ( vel > 1.0 && vel <= 4.0 ) {
        volume = 5;
    } else if ( vel > 4.0 && vel <= 7.0 ) {
        volume = 4;
    } else if ( vel > 7.0 && vel <= 10.0 ) {
        volume = 3;
    } else if ( vel > 10.0 && vel <= 15.0 ) {
        volume = 2;
    } else {
        volume = 1;
    }

    return volume;

}


/*
 * A function which determines the side which is facing up based on
 * the x, y and z accelerations being experienced.
 *
 * It also sets the NEWFACE control signal to 1 if there was a change
 * in face.
 *       _ _ _
 *    /  1   /|  Opposite sides must add to 7.
 *   /_ _ _ / |  +z is normal to 1.
 *   |     | 2/  +y is normal to 2.
 *   |  3  | /   +x is normal to 3.
 *   |_ _ _|/    Face '0' means play nothing.
 *
 * Inputs:
 *     xaccelV: the current x-acceleration voltage
 *     yaccelV: the current y-acceleration voltage
 *     zaccelV: the current z-acceleration voltage
 *     prevFace: the previous face which was up
 *
 * Output:
 *     face: the new face which is up
 */
int newFace(float xaccelV, float yaccelV, float zaccelV,
            int prevFace) {

    int face;

    // If neither x or y axes are experiencing g
    // Pick one of the faces that depends on z acceleration
    // Or pick "no face" face
    if ( (xaccelV >= 1.45 && xaccelV <= 1.7) && (yaccelV >= 1.45 && yaccelV <= 1.65) ) {
        if (zaccelV > 1.7) {
            face = 1; // z = 1g
        } else if (zaccelV < 1.35) {
            face = 6; // z = -1g
        } else {
            face = 0; // can't determine a face
        }

    // If neither x or z axes are experiencing g
    // Pick one of the faces that depends on y acceleration
    // Or pick "no face" face
    } else if ( (xaccelV >= 1.45 && xaccelV <= 1.69) &&
                (zaccelV >= 1.45 && zaccelV <= 1.69) ) {
        if (yaccelV > 1.78) {
            face = 2; // y = 1g
        } else if (yaccelV < 1.49) {
            face = 5; // y = -1g
        } else {
            face = 0; // can't determine a face
        }

    // If neither y or z axes are experiencing g
    // Pick one of the faces that depends on x acceleration
    // Or pick "no face" face
    } else if ( (yaccelV >= 1.45 && yaccelV <= 1.67) &&
                (zaccelV >= 1.45 && zaccelV <= 1.65) ) {
        if (xaccelV > 1.7) {
            face = 3; // x = 1g
        } else if (xaccelV < 1.49) {
            face = 4; // x = -1g
        } else {
            face = 0; // can’t determine a face
        }
    } else {
        face = 0; // can’t determine a face
    }

    if (prevFace == face) {
        NEWFACE = 0;
    } else {
        NEWFACE = 1;
    }

    return face;
}

#endif
```

C.4 audio.h

```
/* SOUNDCUBE PROJECT
 *
 * audio.h
 * Kathleen Kohl kkohl@g.hmc.edu
 * Sakshi Shah   sashah@g.hmc.edu
 * Marisol Beck  mbeck@g.hmc.edu
 * Created: November 29, 2016
 * Modified: December 8, 2016
 *
 * The header file containing the tonesets for the SoundCube,
 * as well as the initialization function.
 */

#ifndef AUDIO_H
#define AUDIO_H

// Forward declaration of helper functions used in audio.c
toneset* GetKeypadChar(FILE *fp);
void PlayMusic(toneset notes, int face, int pin);
void PlayNote(toneset notes, int face, unsigned int duration, int pin);
void ChangeVolume(int volumeLevel, int volumePin1, int volumePin2,
              int volumePin3);

/* A struct which is a 1D array of musical notes.
 * The array contains the frequency in Hz of each note
 * within the toneset.
 *
 * While this struct is not strictly necessary (as we could
 * simply use an array), renaming it "toneset" helps to understand
 * the meaning of the code in audio.c
 */
typedef struct{
    int notes[6];
} toneset;

// G Major Scale
toneset majorscale = {{
 395, // note is G4
 // coded as 395 > 392 Hz because it sounded flat
 440, // A4
 494, // B4
 523, // C5
 587, // D5
 0};  // rest

// Eb Major Pentatonic Scale
//Eb - F - G - Bb - C - Eb
toneset majorpent = {{
 311, // Eb4
 349, // F4
 395, // G4
 466, // Bb4
 523, // C5
 0};  // rest

// Gb Minor Pentatonic Scale
toneset minorpent = {{
 370, // Gb4
 440, // A4
 595, // B4
 554, // Db5
 659, // E5
 0};  // rest

// Notes required for Jingle Bells
toneset jingle = {{
 330,  // E4
 395,  // G4
 262,  // C4
 294,  // D4
 349,  // F4
 0};   // rest

// null toneset
toneset noTone = {{
 0,
 0,
 0,
 0,
 0,
 0};

/* An array which contains the values to write to volumePin1,
 * 2, and 3 respectively based on a volume level. The index of
 * the values corresponds to (volume level - 1).
 */
int volumeControlPins[6][3] =
{ {1, 0, 0}, // select 510 and 1k ohm resistors; R = 338  ohms
```

```
  {0, 1, 0}, // select 510 and 2k ohm resistors; R = 406  ohms
  {0, 0, 1}, // select 510 ohm resistor;         R = 510  ohms
  {1, 1, 0}, // select 1k and 2k ohm resistors;  R = 667  ohms
  {1, 0, 1}, // select 1k ohm resistor;          R = 1000 ohms
  {0, 1, 1}, // select 2k ohm resistor;          R = 2000 ohms
}


#endif
```

C.5 EasyPIO.h

```
/* EasyPIO.h
 * Created:         8 October 2013
 *                      Sarah_Lichtman@hmc.edu & Joshua_Vasquez@hmc.edu
 * Last Modified:   4 April 2014
 *                      Sarah_Lichtman@hmc.edu & Joshua_Vasquez@hmc.edu
 *                  15 August 2014
 *                      David_Harris@hmc.edu  (simplify pinMode)
 *                  8 December 2016
 *                      mbeck@g.hmc.edu & kkohl@h.hmc.edu & sashah@g.hmc.edu
 *
 * Library to simplify memory access on Raspberry Pi (Broadcom BCM2835).
 * Must be run with root permissions using sudo.
 * Functionality not required for this project has been removed.
*/

#ifndef EASY_PIO_H
#define EASY_PIO_H

// Include statements
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

/////////////////////////////////////////////////////////////////////
// Constants
/////////////////////////////////////////////////////////////////////

// GPIO FSEL Types
#define INPUT  0
#define OUTPUT 1
#define ALT0   4
#define ALT1   5
#define ALT2   6
#define ALT3   7
#define ALT4   3
#define ALT5   2

// Clock Manager Bitfield offsets:
#define PWM_CLK_PASSWORD 0x5a000000
#define PWM_MASH 9
#define PWM_KILL 5
```

```
#define PWM_ENAB 4
#define PWM_SRC 0

// PWM Constants
#define PLL_FREQUENCY 500000000 // default PLLD value is 500 [MHz]
#define CM_FREQUENCY 25000000   // max pwm clk is 25 [MHz]
#define PLL_CLOCK_DIVISOR (PLL_FREQUENCY / CM_FREQUENCY)

////////////////////////////////////////////////////////////////////
// Memory Map
////////////////////////////////////////////////////////////////////

// These #define values are specific to the BCM2835, taken from "BCM2835 ARM Peripherals"
//#define BCM2835_PERI_BASE        0x20000000
// Updated to BCM2836 for Raspberry Pi 2.0 Fall 2015 dmh
#define BCM2835_PERI_BASE        0x3F000000

#define GPIO_BASE               (BCM2835_PERI_BASE + 0x200000)
#define UART_BASE               (BCM2835_PERI_BASE + 0x201000)
#define SPI0_BASE               (BCM2835_PERI_BASE + 0x204000)
#define PWM_BASE                (BCM2835_PERI_BASE + 0x20c000)

#define SYS_TIMER_BASE          (BCM2835_PERI_BASE + 0x3000)
#define ARM_TIMER_BASE          (BCM2835_PERI_BASE + 0xB000)

#define CM_PWM_BASE             (BCM2835_PERI_BASE + 0x101000)

#define BLOCK_SIZE (4*1024)

// Pointers that will be memory mapped when pioInit() is called
volatile unsigned int *gpio; //pointer to base of gpio
volatile unsigned int *spi;  //pointer to base of spi registers
volatile unsigned int *pwm;

volatile unsigned int *sys_timer;
volatile unsigned int *arm_timer; // pointer to base of arm timer registers

volatile unsigned int *uart;
volatile unsigned int *cm_pwm;

////////////////////////////////////////////////////////////////////
// GPIO Registers
////////////////////////////////////////////////////////////////////

// Function Select
```

```
#define GPFSEL    ((volatile unsigned int *) (gpio + 0))
typedef struct {
    unsigned FSEL0      : 3;
    unsigned FSEL1      : 3;
    unsigned FSEL2      : 3;
    unsigned FSEL3      : 3;
    unsigned FSEL4      : 3;
    unsigned FSEL5      : 3;
    unsigned FSEL6      : 3;
    unsigned FSEL7      : 3;
    unsigned FSEL8      : 3;
    unsigned FSEL9      : 3;
    unsigned            : 2;
}gpfsel0bits;
#define GPFSEL0bits (*(volatile gpfsel0bits*) (gpio + 0))
#define GPFSEL0 (*(volatile unsigned int*) (gpio + 0))

typedef struct {
    unsigned FSEL10      : 3;
    unsigned FSEL11      : 3;
    unsigned FSEL12      : 3;
    unsigned FSEL13      : 3;
    unsigned FSEL14      : 3;
    unsigned FSEL15      : 3;
    unsigned FSEL16      : 3;
    unsigned FSEL17      : 3;
    unsigned FSEL18      : 3;
    unsigned FSEL19      : 3;
    unsigned             : 2;
}gpfsel1bits;
#define GPFSEL1bits (*(volatile gpfsel1bits*) (gpio + 1))
#define GPFSEL1 (*(volatile unsigned int*) (gpio + 1))

typedef struct {
    unsigned FSEL20      : 3;
    unsigned FSEL21      : 3;
    unsigned FSEL22      : 3;
    unsigned FSEL23      : 3;
    unsigned FSEL24      : 3;
    unsigned FSEL25      : 3;
    unsigned FSEL26      : 3;
    unsigned FSEL27      : 3;
    unsigned FSEL28      : 3;
    unsigned FSEL29      : 3;
    unsigned             : 2;
}gpfsel2bits;
#define GPFSEL2bits (* (volatile gpfsel2bits*) (gpio + 2))
#define GPFSEL2 (* (volatile unsigned int *) (gpio + 2))

typedef struct {
    unsigned FSEL30      : 3;
    unsigned FSEL31      : 3;
    unsigned FSEL32      : 3;
    unsigned FSEL33      : 3;
    unsigned FSEL34      : 3;
    unsigned FSEL35      : 3;
    unsigned FSEL36      : 3;
    unsigned FSEL37      : 3;
    unsigned FSEL38      : 3;
    unsigned FSEL39      : 3;
    unsigned             : 2;
}gpfsel3bits;
#define GPFSEL3bits (* (volatile gpfsel3bits*) (gpio + 3))
#define GPFSEL3 (* (volatile unsigned int *) (gpio + 3))


typedef struct {
    unsigned FSEL40      : 3;
    unsigned FSEL41      : 3;
    unsigned FSEL42      : 3;
    unsigned FSEL43      : 3;
    unsigned FSEL44      : 3;
    unsigned FSEL45      : 3;
    unsigned FSEL46      : 3;
    unsigned FSEL47      : 3;
    unsigned FSEL48      : 3;
    unsigned FSEL49      : 3;
    unsigned             : 2;
}gpfsel4bits;
#define GPFSEL4bits (* (volatile gpfsel4bits*) (gpio + 4))
#define GPFSEL4 (* (volatile unsigned int *) (gpio + 4))

typedef struct {
    unsigned FSEL50      : 3;
    unsigned FSEL51      : 3;
    unsigned FSEL52      : 3;
    unsigned FSEL53      : 3;
    unsigned             : 20;
}gpfsel5bits;
#define GPFSEL5bits (* (volatile gpfsel5bits*) (gpio + 5))
```

```
#define GPFSEL5 (* (volatile unsigned int *) (gpio + 5))

// Pin Output Select
#define GPSET    ((volatile unsigned int *) (gpio + 7))
typedef struct {
    unsigned SET0       : 1;
    unsigned SET1       : 1;
    unsigned SET2       : 1;
    unsigned SET3       : 1;
    unsigned SET4       : 1;
    unsigned SET5       : 1;
    unsigned SET6       : 1;
    unsigned SET7       : 1;
    unsigned SET8       : 1;
    unsigned SET9       : 1;
    unsigned SET10      : 1;
    unsigned SET11      : 1;
    unsigned SET12      : 1;
    unsigned SET13      : 1;
    unsigned SET14      : 1;
    unsigned SET15      : 1;
    unsigned SET16      : 1;
    unsigned SET17      : 1;
    unsigned SET18      : 1;
    unsigned SET19      : 1;
    unsigned SET20      : 1;
    unsigned SET21      : 1;
    unsigned SET22      : 1;
    unsigned SET23      : 1;
    unsigned SET24      : 1;
    unsigned SET25      : 1;
    unsigned SET26      : 1;
    unsigned SET27      : 1;
    unsigned SET28      : 1;
    unsigned SET29      : 1;
    unsigned SET30      : 1;
    unsigned SET31      : 1;
}gpset0bits;
#define GPSET0bits (* (volatile gpset0bits*) (gpio + 7))
#define GPSET0 (* (volatile unsigned int *) (gpio + 7))

typedef struct {
    unsigned SET32       : 1;
    unsigned SET33       : 1;
    unsigned SET34       : 1;
    unsigned SET35       : 1;
    unsigned SET36       : 1;
    unsigned SET37       : 1;
    unsigned SET38       : 1;
    unsigned SET39       : 1;
    unsigned SET40       : 1;
    unsigned SET41       : 1;
    unsigned SET42       : 1;
    unsigned SET43       : 1;
    unsigned SET44       : 1;
    unsigned SET45       : 1;
    unsigned SET46       : 1;
    unsigned SET47       : 1;
    unsigned SET48       : 1;
    unsigned SET49       : 1;
    unsigned SET50       : 1;
    unsigned SET51       : 1;
    unsigned SET52       : 1;
    unsigned SET53       : 1;
    unsigned             : 10;
}gpset1bits;
#define GPSET1bits (* (volatile gpset1bits*) (gpio + 8))
#define GPSET1 (* (volatile unsigned int *) (gpio + 8))

// Pin Output Clear
#define GPCLR    ((volatile unsigned int *) (gpio + 10))
typedef struct {
    unsigned CLR0       : 1;
    unsigned CLR1       : 1;
    unsigned CLR2       : 1;
    unsigned CLR3       : 1;
    unsigned CLR4       : 1;
    unsigned CLR5       : 1;
    unsigned CLR6       : 1;
    unsigned CLR7       : 1;
    unsigned CLR8       : 1;
    unsigned CLR9       : 1;
    unsigned CLR10      : 1;
    unsigned CLR11      : 1;
    unsigned CLR12      : 1;
    unsigned CLR13      : 1;
    unsigned CLR14      : 1;
    unsigned CLR15      : 1;
    unsigned CLR16      : 1;
    unsigned CLR17      : 1;
    unsigned CLR18      : 1;
    unsigned CLR19      : 1;
    unsigned CLR20      : 1;
    unsigned CLR21      : 1;
    unsigned CLR22      : 1;
    unsigned CLR23      : 1;
    unsigned CLR24      : 1;
    unsigned CLR25      : 1;
    unsigned CLR26      : 1;
    unsigned CLR27      : 1;
    unsigned CLR28      : 1;
    unsigned CLR29      : 1;
    unsigned CLR30      : 1;
    unsigned CLR31      : 1;
}gpclr0bits;
#define GPCLR0bits (* (volatile gpclr0bits*) (gpio + 10))
#define GPCLR0 (* (volatile unsigned int *) (gpio + 10))

typedef struct {
    unsigned CLR32       : 1;
    unsigned CLR33       : 1;
    unsigned CLR34       : 1;
    unsigned CLR35       : 1;
    unsigned CLR36       : 1;
    unsigned CLR37       : 1;
    unsigned CLR38       : 1;
    unsigned CLR39       : 1;
    unsigned CLR40       : 1;
    unsigned CLR41       : 1;
    unsigned CLR42       : 1;
    unsigned CLR43       : 1;
    unsigned CLR44       : 1;
    unsigned CLR45       : 1;
    unsigned CLR46       : 1;
    unsigned CLR47       : 1;
    unsigned CLR48       : 1;
    unsigned CLR49       : 1;
    unsigned CLR50       : 1;
    unsigned CLR51       : 1;
    unsigned CLR52       : 1;
    unsigned CLR53       : 1;
    unsigned             : 10;
}gpclr1bits;
#define GPCLR1bits (* (volatile gpclr1bits*) (gpio + 11))
#define GPCLR1 (* (volatile unsigned int *) (gpio + 11))
```

```
// Pin Level
#define GPLEV    ((volatile unsigned int *) (gpio + 13))
typedef struct {
    unsigned LEV0       : 1;
    unsigned LEV1       : 1;
    unsigned LEV2       : 1;
    unsigned LEV3       : 1;
    unsigned LEV4       : 1;
    unsigned LEV5       : 1;
    unsigned LEV6       : 1;
    unsigned LEV7       : 1;
    unsigned LEV8       : 1;
    unsigned LEV9       : 1;
    unsigned LEV10      : 1;
    unsigned LEV11      : 1;
    unsigned LEV12      : 1;
    unsigned LEV13      : 1;
    unsigned LEV14      : 1;
    unsigned LEV15      : 1;
    unsigned LEV16      : 1;
    unsigned LEV17      : 1;
    unsigned LEV18      : 1;
    unsigned LEV19      : 1;
    unsigned LEV20      : 1;
    unsigned LEV21      : 1;
    unsigned LEV22      : 1;
    unsigned LEV23      : 1;
    unsigned LEV24      : 1;
    unsigned LEV25      : 1;
    unsigned LEV26      : 1;
    unsigned LEV27      : 1;
    unsigned LEV28      : 1;
    unsigned LEV29      : 1;
    unsigned LEV30      : 1;
    unsigned LEV31      : 1;
}gplev0bits;
#define GPLEV0bits (* (volatile gplev0bits*) (gpio + 13))
#define GPLEV0 (* (volatile unsigned int *) (gpio + 13))


typedef struct {
    unsigned LEV32       : 1;
    unsigned LEV33       : 1;
    unsigned LEV34       : 1;
    unsigned LEV35       : 1;
    unsigned LEV36       : 1;
    unsigned LEV37       : 1;
    unsigned LEV38       : 1;
    unsigned LEV39       : 1;
    unsigned LEV40       : 1;
    unsigned LEV41       : 1;
    unsigned LEV42       : 1;
    unsigned LEV43       : 1;
    unsigned LEV44       : 1;
    unsigned LEV45       : 1;
    unsigned LEV46       : 1;
    unsigned LEV47       : 1;
    unsigned LEV48       : 1;
    unsigned LEV49       : 1;
    unsigned LEV50       : 1;
    unsigned LEV51       : 1;
    unsigned LEV52       : 1;
    unsigned LEV53       : 1;
    unsigned             : 10;
}gplev1bits;
#define GPLEV1bits (* (volatile gplev1bits*) (gpio + 14))
#define GPLEV1 (* (volatile unsigned int *) (gpio + 14))

////////////////////////////////////////////////////////////////////
// SPI Registers
////////////////////////////////////////////////////////////////////

typedef struct {
    unsigned CS         :2;
    unsigned CPHA       :1;
    unsigned CPOL       :1;
    unsigned CLEAR      :2;
    unsigned CSPOL      :1;
    unsigned TA         :1;
    unsigned DMAEN      :1;
    unsigned INTD       :1;
    unsigned INTR       :1;
    unsigned ADCS       :1;
    unsigned REN        :1;
    unsigned LEN        :1;
    unsigned LMONO      :1;
    unsigned TE_EN      :1;
    unsigned DONE       :1;
    unsigned RXD        :1;
    unsigned TXD        :1;
    unsigned RXR        :1;
    unsigned RXF        :1;
    unsigned CSPOL0     :1;
    unsigned CSPOL1     :1;
    unsigned CSPOL2     :1;
    unsigned DMA_LEN    :1;
    unsigned LEN_LONG   :1;
    unsigned            :6;
}spi0csbits;
#define SPI0CSbits (* (volatile spi0csbits*) (spi + 0))
#define SPI0CS (* (volatile unsigned int *) (spi + 0))

#define SPI0FIFO (* (volatile unsigned int *) (spi + 1))
#define SPI0CLK (* (volatile unsigned int *) (spi + 2))
#define SPI0DLEN (* (volatile unsigned int *) (spi + 3))

////////////////////////////////////////////////////////////////////
// System Timer Registers
////////////////////////////////////////////////////////////////////

typedef struct {
    unsigned M0     :1;
    unsigned M1     :1;
    unsigned M2     :1;
    unsigned M3     :1;
    unsigned        :28;
}sys_timer_csbits;
#define SYS_TIMER_CSbits (*(volatile sys_timer_csbits*) (sys_timer + 0))
#define SYS_TIMER_CS    (* (volatile unsigned int*)(sys_timer + 0))

#define SYS_TIMER_CLO   (* (volatile unsigned int*)(sys_timer + 1))
#define SYS_TIMER_CHI   (* (volatile unsigned int*)(sys_timer + 2))
#define SYS_TIMER_C0    (* (volatile unsigned int*)(sys_timer + 3))
#define SYS_TIMER_C1    (* (volatile unsigned int*)(sys_timer + 4))
#define SYS_TIMER_C2    (* (volatile unsigned int*)(sys_timer + 5))
#define SYS_TIMER_C3    (* (volatile unsigned int*)(sys_timer + 6))

////////////////////////////////////////////////////////////////////
// ARM Timer Registers
////////////////////////////////////////////////////////////////////

#define ARM_TIMER_LOAD (* (volatile unsigned int *) (arm_timer + 256))
//TODO: make timer control struct
#define ARM_TIMER_CONTROL  (* (volatile unsigned int *) (arm_timer + 258))
```

```
#define ARM_TIMER_IRQCLR (* (volatile unsigned int*) (arm_timer + 259))
#define ARM_TIMER_RAWIRQ (* (volatile unsigned int *) (arm_timer + 260))
#define ARM_TIMER_RELOAD (* (volatile unsigned int *) (arm_timer + 262))
#define ARM_TIMER_DIV (* (volatile unsigned int *) (arm_timer + 263))

/////////////////////////////////////////////////////////////////////
// Clock Manager Registers
/////////////////////////////////////////////////////////////////////

typedef struct {
    unsigned SRC        :4;
    unsigned ENAB       :1;
    unsigned KILL       :1;
    unsigned            :1;
    unsigned BUSY       :1;
    unsigned FLIP       :1;
    unsigned MASH       :2;
    unsigned            :13;
    unsigned PASSWD     :8;
}cm_pwmctl_bits;
#define CM_PWMCTLbits (* (volatile cm_pwmctl_bits *) (cm_pwm + 40))
#define CM_PWMCTL (* (volatile unsigned int*) (cm_pwm + 40))

typedef struct {
    unsigned DIVF       :12;
    unsigned DIVI       :12;
    unsigned PASSWD     :8;
} cm_pwmdivbits;
#define CM_PWMDIVbits (* (volatile cm_pwmdivbits *) (cm_pwm + 41))
#define CM_PWMDIV (*(volatile unsigned int *)(cm_pwm + 41))

/////////////////////////////////////////////////////////////////////
// General Functions
/////////////////////////////////////////////////////////////////////

// TODO: return error code instead of printing (mem_fd, reg_map)
void pioInit() {
    int  mem_fd;
    void *reg_map;

    // /dev/mem is a psuedo-driver for accessing memory in the Linux filesystem
    if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
          printf("can't open /dev/mem \n");
          exit(-1);
    }
```

```
reg_map = mmap(
  NULL,             //Address at which to start local mapping (null means don't-care)
  BLOCK_SIZE,       //Size of mapped memory block
  PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
  MAP_SHARED,       // This program does not have exclusive access to this memory
  mem_fd,           // Map to /dev/mem
  GPIO_BASE);       // Offset to GPIO peripheral

if (reg_map == MAP_FAILED) {
  printf("gpio mmap error %d\n", (int)reg_map);
  close(mem_fd);
  exit(-1);
}

gpio = (volatile unsigned *)reg_map;

reg_map = mmap(
  NULL,             //Address at which to start local mapping (null means don't-care)
  BLOCK_SIZE,       //Size of mapped memory block
  PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
  MAP_SHARED,       // This program does not have exclusive access to this memory
  mem_fd,           // Map to /dev/mem
  SPI0_BASE);       // Offset to SPI peripheral

if (reg_map == MAP_FAILED) {
  printf("spi mmap error %d\n", (int)reg_map);
  close(mem_fd);
  exit(-1);
}

spi = (volatile unsigned *)reg_map;

reg_map = mmap(
  NULL,             //Address at which to start local mapping (null means don't-care)
  BLOCK_SIZE,       //Size of mapped memory block
  PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
  MAP_SHARED,       // This program does not have exclusive access to this memory
  mem_fd,           // Map to /dev/mem
  PWM_BASE);       // Offset to PWM peripheral

if (reg_map == MAP_FAILED) {
  printf("pwm mmap error %d\n", (int)reg_map);
  close(mem_fd);
  exit(-1);
}

pwm = (volatile unsigned *)reg_map;

reg_map = mmap(
  NULL,             //Address at which to start local mapping (null means don't-care)
  BLOCK_SIZE,       //Size of mapped memory block
  PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
  MAP_SHARED,       // This program does not have exclusive access to this memory
  mem_fd,           // Map to /dev/mem
  SYS_TIMER_BASE);       // Offset to Timer peripheral

if (reg_map == MAP_FAILED) {
  printf("sys timer mmap error %d\n", (int)reg_map);
  close(mem_fd);
  exit(-1);
}

sys_timer = (volatile unsigned *)reg_map;

reg_map = mmap(
  NULL,             //Address at which to start local mapping (null means don't-care)
  BLOCK_SIZE,       //Size of mapped memory block
  PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
  MAP_SHARED,       // This program does not have exclusive access to this memory
  mem_fd,           // Map to /dev/mem
  ARM_TIMER_BASE);       // Offset to interrupts


if (reg_map == MAP_FAILED) {
  printf("arm timer mmap error %d\n", (int)reg_map);
  close(mem_fd);
  exit(-1);
}

arm_timer = (volatile unsigned *)reg_map;

reg_map = mmap(
  NULL,             //Address at which to start local mapping (null means don't-care)
  BLOCK_SIZE,       //Size of mapped memory block
  PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
  MAP_SHARED,       // This program does not have exclusive access to this memory
  mem_fd,           // Map to /dev/mem
  UART_BASE);       // Offset to UART peripheral
```

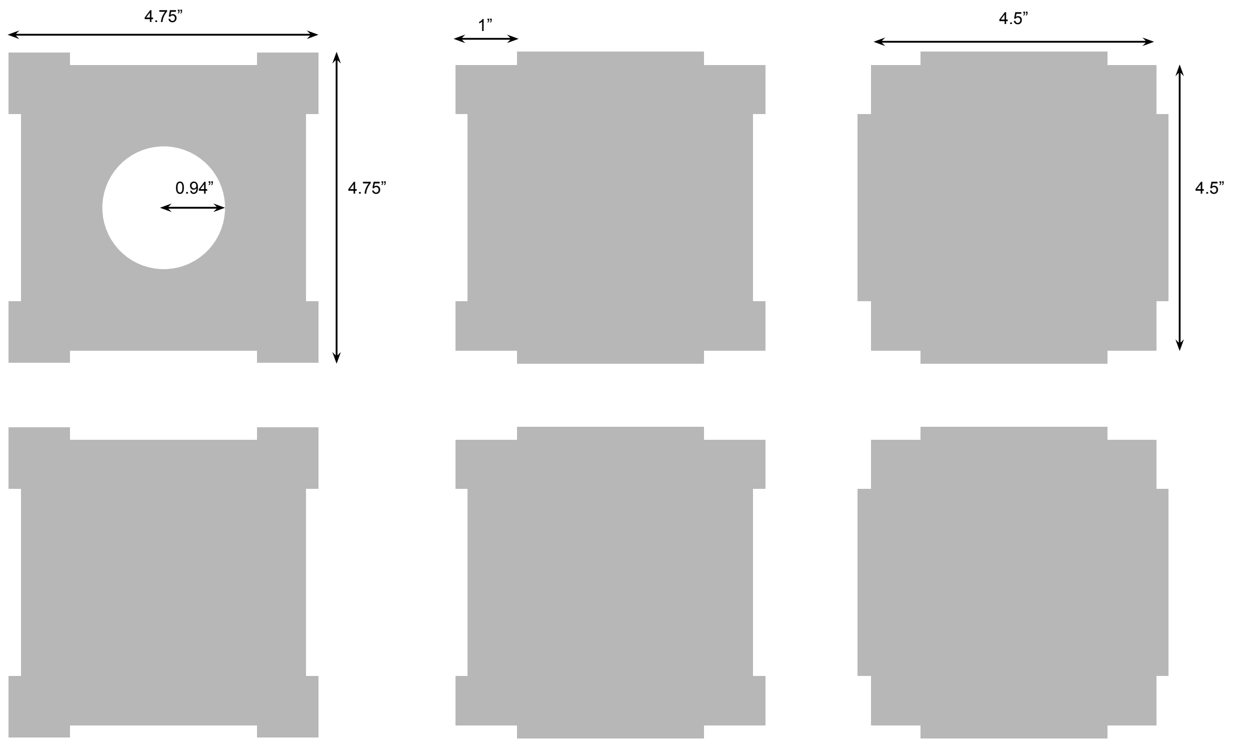
```
    if (reg_map == MAP_FAILED) {
      printf("uart mmap error %d\n", (int)reg_map);
      close(mem_fd);
      exit(-1);
    }

    uart = (volatile unsigned *)reg_map;

    reg_map = mmap(
      NULL,             //Address at which to start local mapping (null means don't-care)
      BLOCK_SIZE,       //Size of mapped memory block
      PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
      MAP_SHARED,       // This program does not have exclusive access to this memory
      mem_fd,           // Map to /dev/mem
      CM_PWM_BASE);       // Offset to ARM timer peripheral

    if (reg_map == MAP_FAILED) {
      printf("cm_pwm mmap error %d\n", (int)reg_map);
      close(mem_fd);
      exit(-1);
    }

    cm_pwm = (volatile unsigned *)reg_map;
    close(mem_fd);
}

/////////////////////////////////////////////////////////////////////
// GPIO Functions
/////////////////////////////////////////////////////////////////////

void pinMode(int pin, int function) {
    int reg      =  pin/10;
    int offset   = (pin%10)*3;
    GPFSEL[reg] &= ~((0b111 & ~function) << offset);
    GPFSEL[reg] |=  ((0b111 &  function) << offset);
}

void digitalWrite(int pin, int val) {
    int reg = pin / 32;
    int offset = pin % 32;

    if (val) GPSET[reg] = 1 << offset;
    else     GPCLR[reg] = 1 << offset;
}
```

```
int digitalRead(int pin) {
    int reg = pin / 32;
    int offset = pin % 32;

    return (GPLEV[reg] >> offset) & 0x00000001;
}

void pinsMode(int pins[], int numPins, int fxn) {
    int i;
    for(i=0; i<numPins; ++i) {
        pinMode(pins[i], fxn);
    }
}

void digitalWrites(int pins[], int numPins, int val) {
    int i;
    for(i=0; i<numPins; i++) {
        digitalWrite(pins[i], (val & 0x00000001));
        val = val >> 1;
    }
}

int digitalReads(int pins[], int numPins) {
    int i, val = digitalRead(pins[0]);

    for(i=1; i<numPins; i++) {
        val |= (digitalRead(pins[i]) << i);
    }
    return val;
}

////////////////////////////////////////////////////////////////////
// Timer Functions
////////////////////////////////////////////////////////////////////

// RPi timer peripheral clock is 1MHz.
// M0 and M3 are used by the GPU, so we must use M1 or M2

void delayMicros(int micros) {
    SYS_TIMER_C1 = SYS_TIMER_CLO + micros;   // set the compare register
    // 1000 clocks per millisecond
    SYS_TIMER_CSbits.M1 = 1;                 // reset match flag to 0
    while(SYS_TIMER_CSbits.M1 == 0);         // wait until the match flag is set
}
```

```
void delayMillis(int millis) {
    delayMicros(millis*1000);               // 1000 microseconds per millisecond
}

/////////////////////////////////////////////////////////////////////
// SPI Functions
/////////////////////////////////////////////////////////////////////

void spiInit(int freq, int cpha, int cpol, int cs) {
    // set GPIO 8 (CE0), 9 (MISO), 10 (MOSI), 11 (SCLK) alt fxn 0 (SPI0)
    // and GPIO 7 (CE1) for a second chip select
    pinMode(7, ALT0);
    pinMode(8, ALT0);
    pinMode(9, ALT0);
    pinMode(10, ALT0);
    pinMode(11, ALT0);

    //Note: clock divisor will be rounded to the nearest power of 2
    SPI0CLK = 250000000/freq; // set SPI clock to 250MHz / freq
    SPI0CSbits.TA = 0;
    SPI0CSbits.CPHA = cpha;
    SPI0CSbits.CPOL = cpol;
    SPI0CSbits.CSPOL = 0;     // we want CS active low
    SPI0CSbits.CS = cs;
}

char spiSendReceive(char send) {
    SPI0FIFO = send;
    while(!SPI0CSbits.DONE);
    return SPI0FIFO;
}

short spiSendReceive16(short send) {
    short rec;
    SPI0CSbits.TA = 1;
    printf("Right before sending anything, SPI0CS is %x\n",SPI0CS);
    rec = spiSendReceive((send & 0xFF00) >> 8); // send data MSB first
    rec = (rec << 8) | spiSendReceive(send & 0xFF);
    SPI0CSbits.TA = 0;
    printf("Right after sending, set TA = 0,  SPI0CS is %x\n", SPI0CS);
    return rec;
}

#endif
```

Appendix D - Cube Assembly

The plexiglass used was 1/8" thick and was cut using a laser cutter to the dimensions in Figure 11. Five of the cube faces were "welded" together using acetone. One face was left unglued so that the batteries inside the cube could be replaced. For the prototype, that side is simply affixed with clear tape. The circle cut out of one side of the cube is where the speaker was mounted using hot glue.



All the grooves are ⅛" either in or out from the edge, as the Plexiglass used was ⅛" thick.

Figure 11: The dimensions that the plexiglass was cut to in order to create the sides of the cube.

The hardware housed inside the cube was wired on three small breadboards except for the Raspberry Pi. The breadboards have sticky backs which were affixed to the sides of the cube. The Pi, 5V power supply, and 9V battery were all affixed to the sides using duct tape for easy removal when needed.