FINAL LAB: SOUND TRAINING GAME

ECE 4273: Digital Design Laboratory

David McKnight and Ben Kory in the House

Contents

Introduction

For this project, we endeavored to make an audio training game. Gameplay consists of players hearing musical notes on a pentatonic scale (the major pentatonic scale of A3, C4, D4, E4, G4, and A4) and playing the corresponding note up an octave higher in response (the notes A4, C5, D5, E5, G5, and A5).

Upon startup, a short piece of music is played to the user using the sounds that they'll be training on. The game then plays a note randomly selected from the scale; when the user correctly plays the note an octave above it in response, the game gives a confirmation sound effect and plays a new note. The process of the game playing a note and awaiting the proper response from the user continues until the end of the game.

Project Overview

Sine Wave Synthesis

To generate sine waves, we used the standard library's valarray<T> class, creating methods for generating a range of values at provided intervals, normalizing those values between zero and one, and rescaling them to the appropriate values for output.

Audio Output

For the start-up audio sequence and the sound-wave output, we used sine waves we generated using the methods described in "Sine Wave Synthesis" and output them to the LPCXpresso's Digital-to-Analog subsystem. We used a wait function we created in an earlier lab to space out the outputs to that subsystem (as well as changed the intervals of the sine wave samples) in order to achieve desired frequencies, which in our case were the notes of the major pentatonic scale consisting of A3, C4, D4, E4, G4, and A4.

Button Input and Response Wave Generation

Due to some issues with the consistency of reading in a planned PS/2 keyboard's inputs, we instead decided to use push-button inputs. These push-buttons also hit diode logic to display the appropriate note name on a 14-segment LED display and use 555 timers to output square waves to the speaker.

PS/2 Input and Game Start

The start of the game is triggered by receiving an input from the spacebar of a PS/2 protocol keyboard.

Audio Summation and Amplification

To make sure that all the wave generators could hook up to the same speaker without interfering with each other, we ran them through resistors before joining them at a single node. We then output the result at that node to an LM386 audio amplification circuit, which then outputs to a speaker.

Sine-Wave Synthesis

General Description

For creating sine waves, we have to consider that the LPCXpresso can't output a large number of data points in a short period of time. Thus, we need a way to prepare discrete sinusoid approximations with varying numbers of samples.

Initial Sine Wave

We begin by creating a range of values between 0 and 2pi (exclusive in the case of the latter) as part of a val_array<double>. We then apply the sin function to that val_array<double>.

Sine Wave Renormalization

To renormalize the sine wave in preparation for output to the digital-to-analog subsystem, we first bring the wave out of the negative-y half-plane. To do this, we subtract the minimum sample in the wave from every value in the wave. Next, we bring the wave between 0 and 1 (exclusive in the case of the latter) by dividing all elements in the val_array<double> by the largest value in the wave (post-minimum-sample-subtraction). Finally, we multiply the wave's values by 1023 and truncate the elements to form a new val_array<int> for output.

Audio Output

General Description

To output audio waves from the LPCXpresso, we output samples to the digital-to-analog subsystem. We achieve different frequencies by varying the number of those samples and the time we wait before changing them.

Digital-to-Analog Subsystem

utilize the digital-to-analog subsystem (DAC). The DAC allows us to approximate a continuous signal through outputting samples to pin enabled for such a process; more specifically, by setting values from 0 to 1023 in the appropriate register, the DAC outputs values between 0 and 3 volts to the associated pin.

Wave Requirements

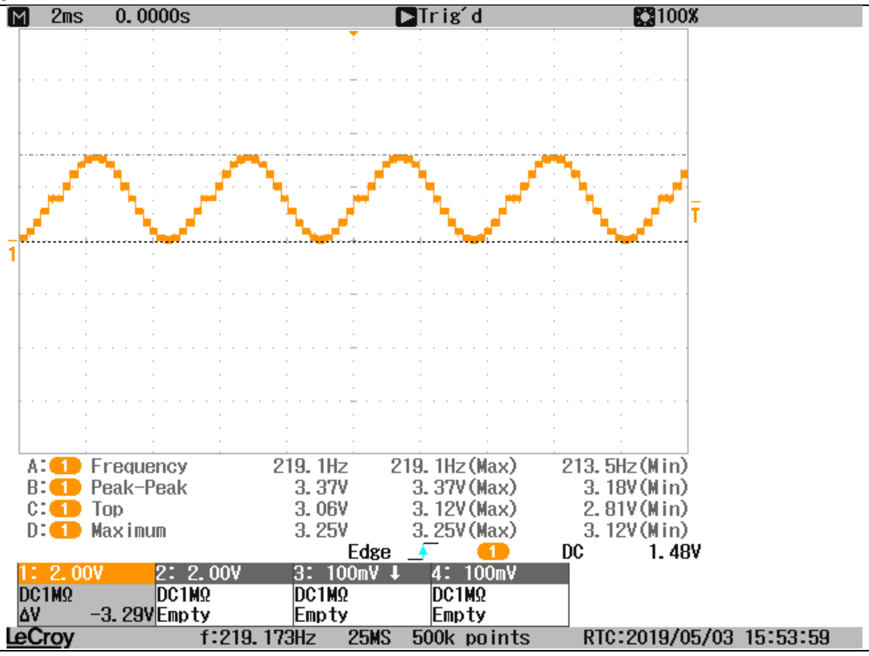
As mentioned previously, the waves output through the DAC must have values between 0 and 1023, inclusive. We therefore use the wave generation method described in the Sine-Wave synthesis section to produce DAC-compatible signals for output.

Synchronization/Frequency Notes

We desire waves in the pentatonic major scale A-C-D-E-G; specifically, we would like to generate A3, C4, D4, E4, and G4, which have frequencies 220Hz, 261.63Hz, 293.66Hz, 329.63Hz, 392Hz, and 440.00Hz, respectively.

Oscilloscope Images

A3 (220 Hz)

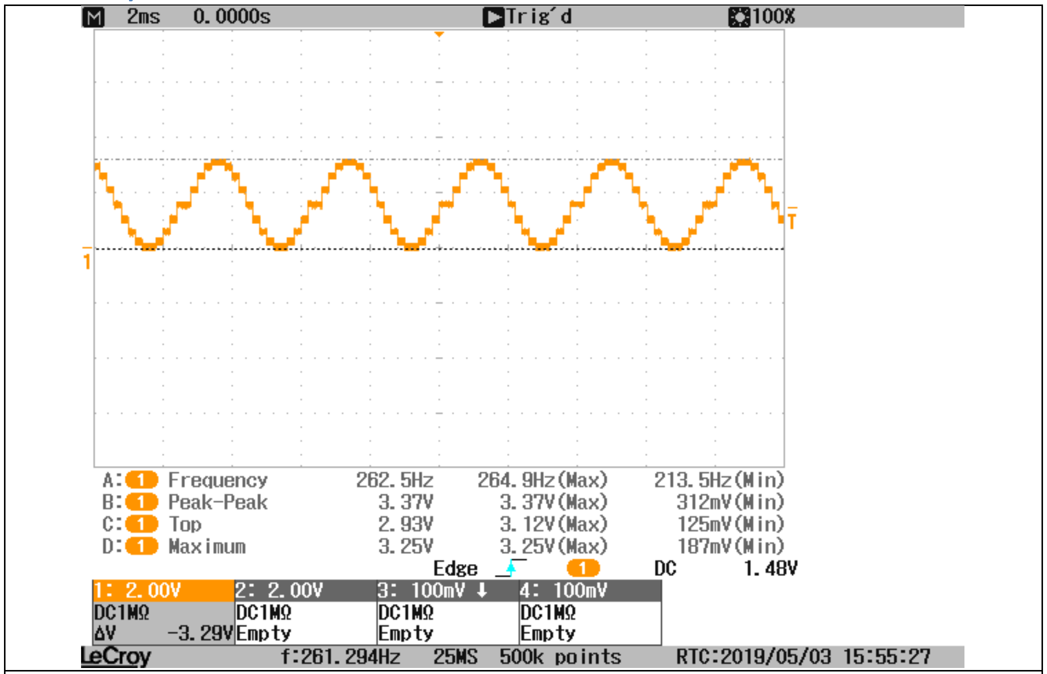


```
M  2ms    0.0000s                 Trig'd                100%

A: 1 Frequency     219.1Hz   219.1Hz(Max)   213.5Hz(Min)
B: 1 Peak-Peak       3.37V     3.37V(Max)     3.18V(Min)
C: 1 Top             3.06V     3.12V(Max)     2.81V(Min)
D: 1 Maximum         3.25V     3.25V(Max)     3.12V(Min)
                         Edge       1        DC      1.48V
1: 2.00V     2: 2.00V   3: 100mV   4: 100mV
DC1MΩ        DC1MΩ      DC1MΩ      DC1MΩ
ΔV   -3.29V  Empty      Empty      Empty
LeCroy     f:219.173Hz   25MS   500k points    RTC:2019/05/03 15:53:59
```

LPCXpresso-Generated A220Hz Wave Oscilloscope Capture

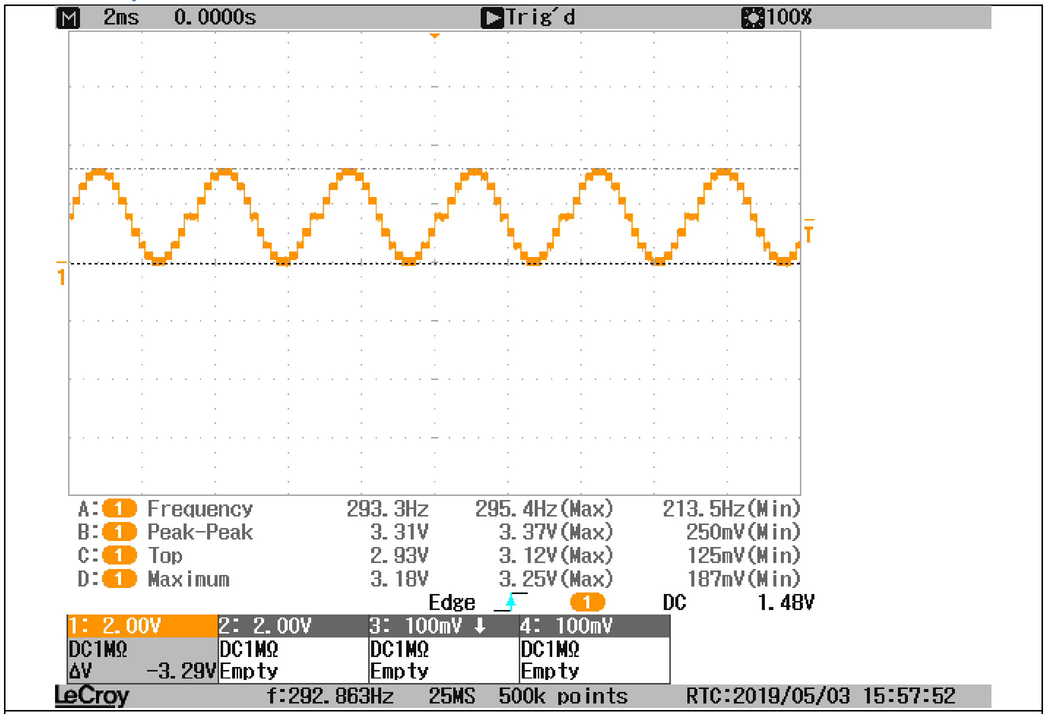
C4 (261.63 Hz)



```
M  2ms   0.0000s              Trig'd                    100%
A: 1 Frequency      262.5Hz   264.9Hz(Max)   213.5Hz(Min)
B: 1 Peak-Peak        3.37V     3.37V(Max)    312mV(Min)
C: 1 Top              2.93V     3.12V(Max)    125mV(Min)
D: 1 Maximum          3.25V     3.25V(Max)    187mV(Min)
                         Edge        1        DC       1.48V
1: 2.00V      2: 2.00V      3: 100mV      4: 100mV
DC1MΩ         DC1MΩ         DC1MΩ         DC1MΩ
ΔV    -3.29V  Empty         Empty         Empty
LeCroy        f:261.294Hz   25MS   500k points   RTC:2019/05/03 15:55:27
```

LPCXpresso-Generated C261.63Hz Wave Oscilloscope Capture

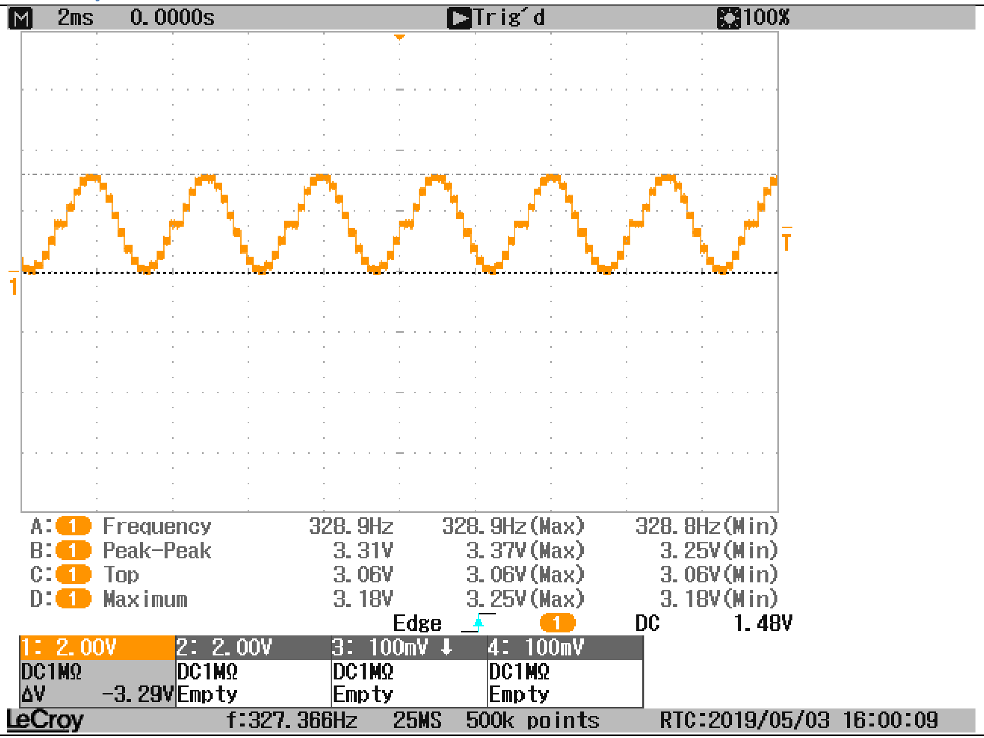
D4 (293.66 Hz)



```
M  2ms   0.0000s              Trig'd                    100%
A: 1 Frequency      293.3Hz   295.4Hz(Max)   213.5Hz(Min)
B: 1 Peak-Peak        3.31V     3.37V(Max)    250mV(Min)
C: 1 Top              2.93V     3.12V(Max)    125mV(Min)
D: 1 Maximum          3.18V     3.25V(Max)    187mV(Min)
                         Edge        1        DC       1.48V
1: 2.00V      2: 2.00V      3: 100mV      4: 100mV
DC1MΩ         DC1MΩ         DC1MΩ         DC1MΩ
ΔV    -3.29V  Empty         Empty         Empty
LeCroy        f:292.863Hz   25MS   500k points   RTC:2019/05/03 15:57:52
```

Image Caption

E4 (329.63 Hz)

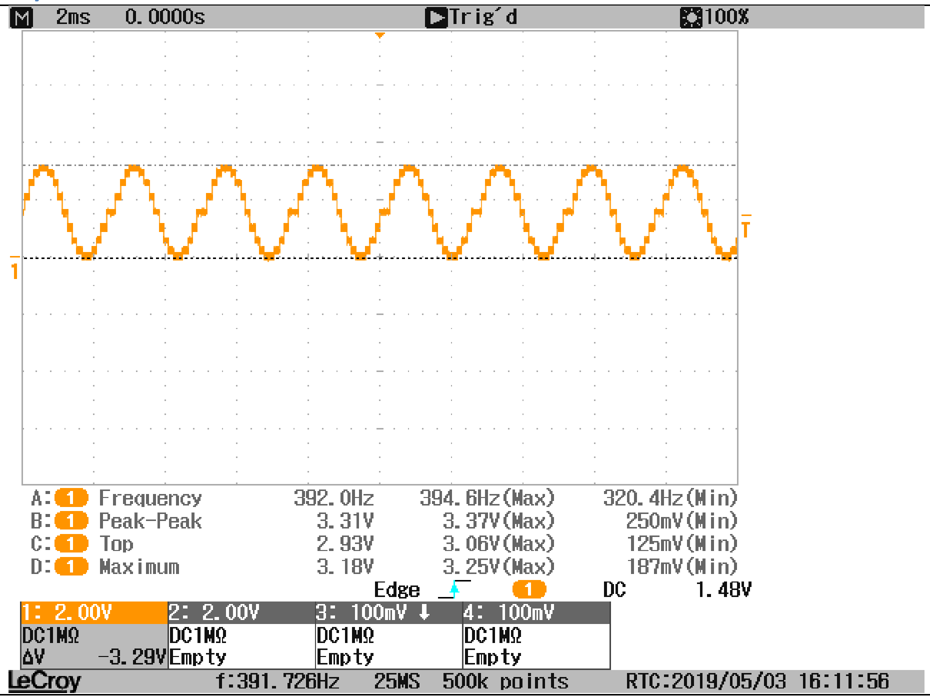


```
M  2ms   0.0000s                Trig'd                100%

A: 1  Frequency      328.9Hz    328.9Hz(Max)    328.8Hz(Min)
B: 1  Peak-Peak       3.31V      3.37V(Max)      3.25V(Min)
C: 1  Top             3.06V      3.06V(Max)      3.06V(Min)
D: 1  Maximum         3.18V      3.25V(Max)      3.18V(Min)
                          Edge        1      DC       1.48V
1: 2.00V         2: 2.00V   3: 100mV   4: 100mV
DC1MΩ            DC1MΩ      DC1MΩ      DC1MΩ
ΔV       -3.29V  Empty      Empty      Empty
LeCroy      f:327.366Hz   25MS   500k points   RTC:2019/05/03 16:00:09
```

LPCXpresso-Generated E329.63Hz Wave Oscilloscope Capture

G4 (392 Hz)

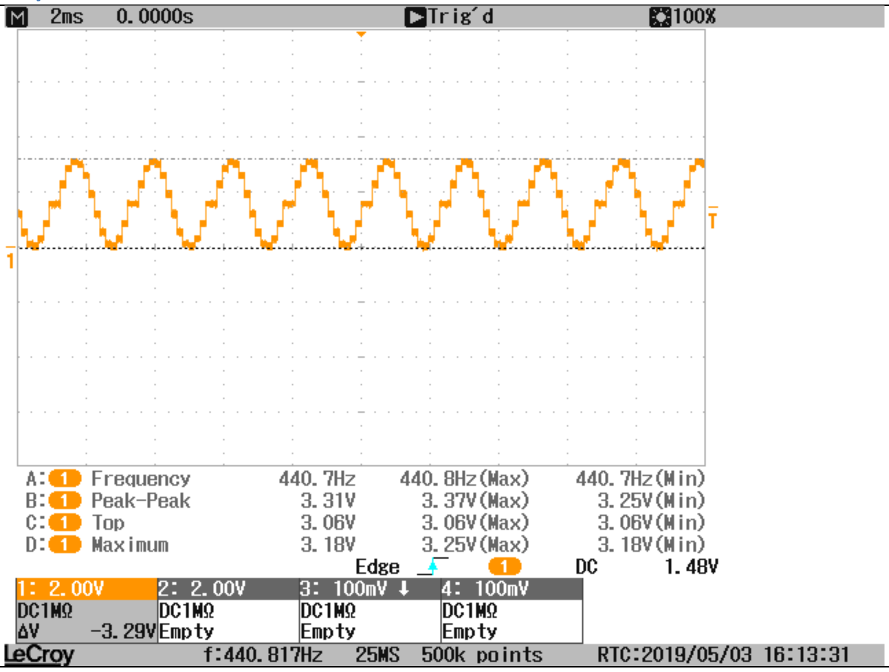


```
M  2ms   0.0000s                Trig'd                100%

A: 1  Frequency      392.0Hz    394.6Hz(Max)    320.4Hz(Min)
B: 1  Peak-Peak       3.31V      3.37V(Max)      250mV(Min)
C: 1  Top             2.93V      3.06V(Max)      125mV(Min)
D: 1  Maximum         3.18V      3.25V(Max)      187mV(Min)
                          Edge        1      DC       1.48V
1: 2.00V         2: 2.00V   3: 100mV   4: 100mV
DC1MΩ            DC1MΩ      DC1MΩ      DC1MΩ
ΔV       -3.29V  Empty      Empty      Empty
LeCroy      f:391.726Hz   25MS   500k points   RTC:2019/05/03 16:11:56
```

LPCXpresso-Generated G392Hz Wave Oscilloscope Capture

A4 (440.0 Hz)



```
M  2ms   0.0000s              Trig'd              100%

A: 1 Frequency        440.7Hz    440.8Hz(Max)    440.7Hz(Min)
B: 1 Peak-Peak          3.31V      3.37V(Max)      3.25V(Min)
C: 1 Top                3.06V      3.06V(Max)      3.06V(Min)
D: 1 Maximum            3.18V      3.25V(Max)      3.18V(Min)
                         Edge         1       DC        1.48V
1: 2.00V      2: 2.00V    3: 100mV    4: 100mV
DC1MΩ         DC1MΩ       DC1MΩ       DC1MΩ
ΔV    -3.29V  Empty       Empty       Empty
LeCroy        f:440.817Hz   25MS   500k points   RTC:2019/05/03 16:13:31
```

LPCXpresso-Generated A440Hz Wave Oscilloscope Capture

Initial Start-Up Audio Recording

OU_Bells_Startup_Sequence.wav

OU Bells Startup Sequence

Button Input and Response Wave Generation

General Overview

Due to some issues with synchronization affecting the consistency of the PS/2 keyboard inputs, we used push-buttons to track responses instead. Additionally, we used 555-timer outputs tied to those push-buttons to output players' response sounds due to concerns about processing speed vis-à-vis outputting to an LCD screen, which we originally planned to do.

PS/2 Input

General Overview

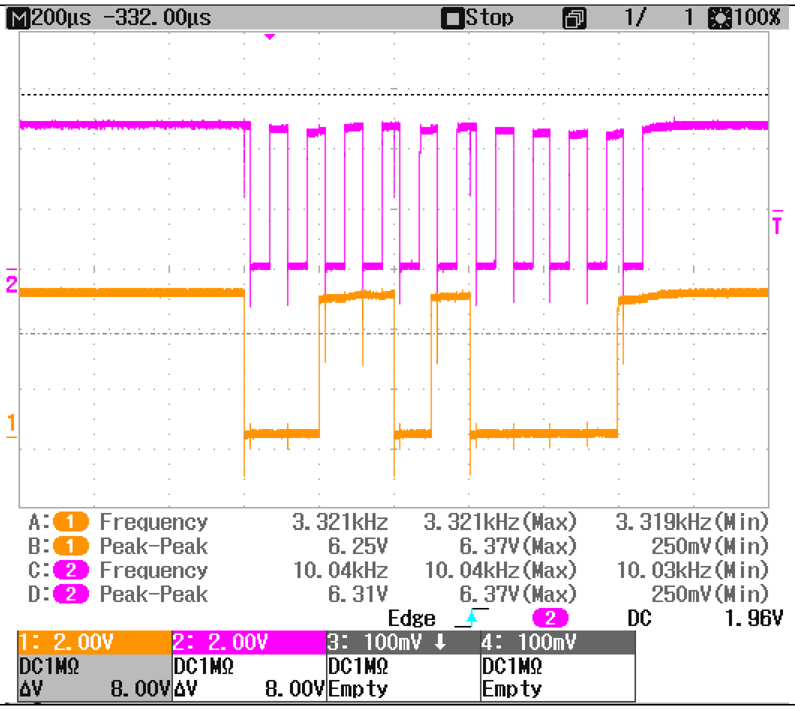
The PS/2 protocol utilizes a device (keyboard) generated clock and data stream, along with standard power/ground rails. Each key on a PS/2 keyboard is assigned an 8-bit value, which is then preceded by a single start bit (always 0), and succeeded by a parity bit and a stop bit (always 1), forming an 11-bit frame to be transmitted. The main game loop, prior to actually beginning the loop itself, waits until a recognized input from the PS/2, that corresponding to a certain key, is received.

Implementation

The PS/2 keyboard was physically implemented by splicing its cable and attaching via solder breadboard-capable wires. The data and clock signals from the keyboard were then read in as inputs to GPIO pins. In terms of software, the PS/2 protocol was read in by sampling the incoming data after confirming that a signal had been sent. This could be confirmed by starting sampling whenever the clock was pulled low as the clock's default state was high. Because the CPU operates at 4 MHz, which is somewhat greater than the 15 kHz of the keyboard, the sampling was reasonably accurate, sufficient enough to provide a start command to the game loop.

Oscilloscope Observations

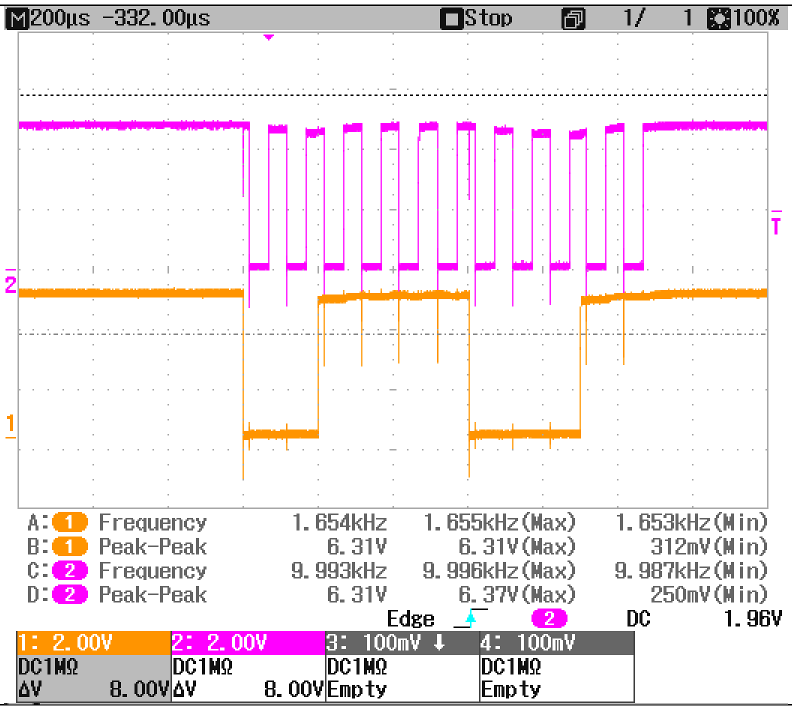
Button '1'



```
A: 1 Frequency   3.321kHz   3.321kHz(Max)   3.319kHz(Min)
B: 1 Peak-Peak      6.25V      6.37V(Max)      250mV(Min)
C: 2 Frequency   10.04kHz   10.04kHz(Max)   10.03kHz(Min)
D: 2 Peak-Peak      6.31V      6.37V(Max)      250mV(Min)
                         Edge    2     DC      1.96V
1: 2.00V      2: 2.00V      3: 100mV     4: 100mV
DC1MΩ         DC1MΩ         DC1MΩ        DC1MΩ
ΔV    8.00V   ΔV    8.00V   Empty        Empty
```

Oscilloscope Reading for PS/2 Keyboard Input '1'
Corresponding byte is 01101000

Button ‘2’

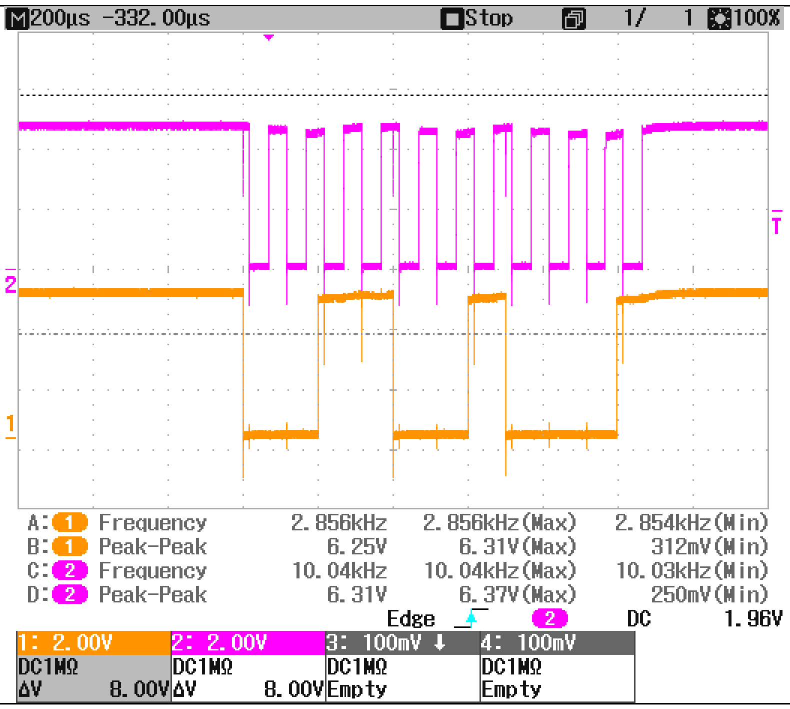


```
M200µs -332.00µs        Stop        1/   1   100%

A: 1 Frequency     1.654kHz   1.655kHz(Max)   1.653kHz(Min)
B: 1 Peak-Peak        6.31V      6.31V(Max)     312mV(Min)
C: 2 Frequency     9.993kHz   9.996kHz(Max)   9.987kHz(Min)
D: 2 Peak-Peak        6.31V      6.37V(Max)     250mV(Min)
                        Edge        2        DC       1.96V
1: 2.00V        2: 2.00V        3: 100mV        4: 100mV
DC1MΩ           DC1MΩ           DC1MΩ           DC1MΩ
ΔV       8.00V  ΔV       8.00V  Empty           Empty
```

Oscilloscope Reading for PS/2 Keyboard Input ‘2’
Corresponding byte is 01111000

Button ‘3’

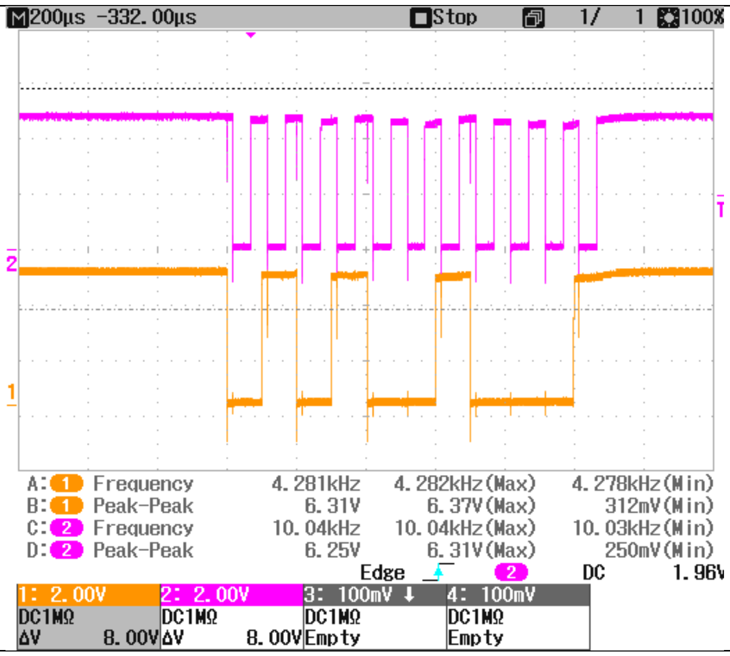


```
M200µs -332.00µs        Stop        1/   1   100%

A: 1 Frequency     2.856kHz   2.856kHz(Max)   2.854kHz(Min)
B: 1 Peak-Peak        6.25V      6.31V(Max)     312mV(Min)
C: 2 Frequency     10.04kHz   10.04kHz(Max)   10.03kHz(Min)
D: 2 Peak-Peak        6.31V      6.37V(Max)     250mV(Min)
                        Edge        2        DC       1.96V
1: 2.00V        2: 2.00V        3: 100mV        4: 100mV
DC1MΩ           DC1MΩ           DC1MΩ           DC1MΩ
ΔV       8.00V  ΔV       8.00V  Empty           Empty
```

Oscilloscope Reading for PS/2 Keyboard Input ‘3’
Corresponding byte is 01100100

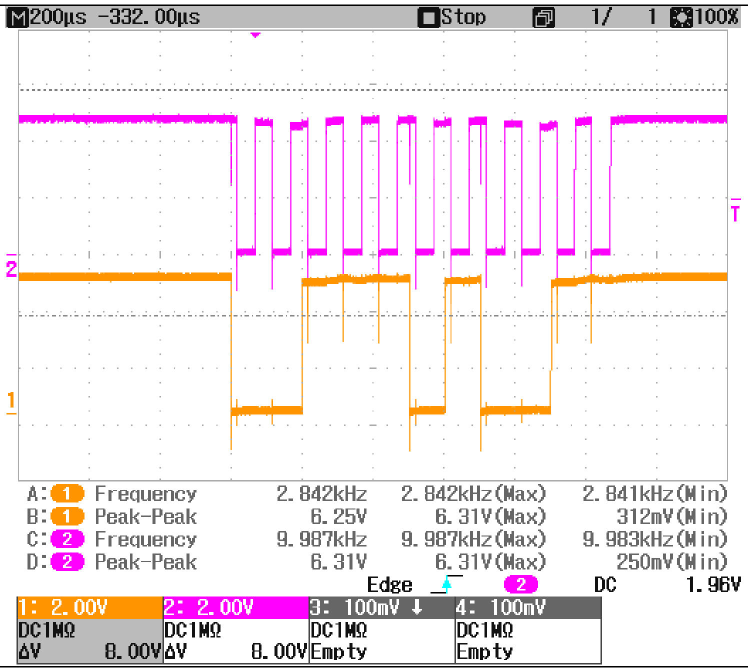
Button ‘4’



```
A: 1 Frequency   4.281kHz   4.282kHz(Max)   4.278kHz(Min)
B: 1 Peak-Peak   6.31V      6.37V(Max)      312mV(Min)
C: 2 Frequency   10.04kHz   10.04kHz(Max)   10.03kHz(Min)
D: 2 Peak-Peak   6.25V      6.31V(Max)      250mV(Min)
                       Edge     2     DC     1.96V
1: 2.00V        2: 2.00V        3: 100mV      4: 100mV
DC1MΩ           DC1MΩ           DC1MΩ         DC1MΩ
ΔV      8.00V   ΔV      8.00V   Empty         Empty
```

Oscilloscope Reading for PS/2 Keyboard Input ‘4’
Corresponding byte is 10100100

Button ‘5’



```
A: 1 Frequency   2.842kHz   2.842kHz(Max)   2.841kHz(Min)
B: 1 Peak-Peak   6.25V      6.31V(Max)      312mV(Min)
C: 2 Frequency   9.987kHz   9.987kHz(Max)   9.983kHz(Min)
D: 2 Peak-Peak   6.31V      6.31V(Max)      250mV(Min)
                       Edge     2     DC     1.96V
1: 2.00V        2: 2.00V        3: 100mV      4: 100mV
DC1MΩ           DC1MΩ           DC1MΩ         DC1MΩ
ΔV      8.00V   ΔV      8.00V   Empty         Empty
```

Oscilloscope Reading for PS/2 Keyboard Input ‘5’
Corresponding byte is 01110100

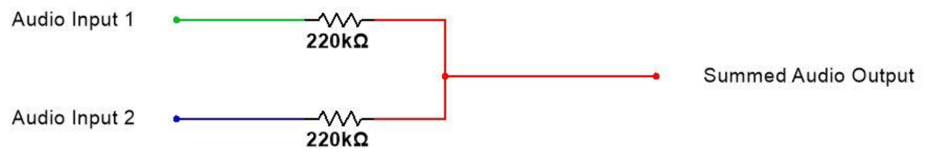
Audio Summation and Amplification

Audio Summation

Audio Summation Description

Connecting the outputs of the audio waves directly resulted in feedback distortion. As such, we connect them through parallel resistors; we selected 220kOhm resistors for this, as lower values were not enough to avoid distortion.

Audio Summation Schematic



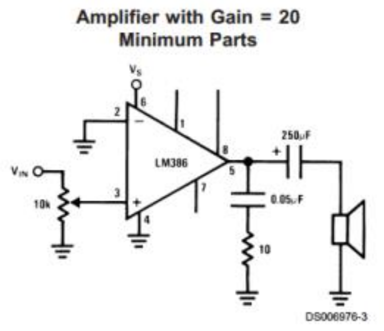
A rudimentary signal summing amplifier with Gain <1. In this way, all 6 notes were summed.

Audio Amplification

Audio Amplification Description

The 220kOhm resistors that we used dampened the audio considerably; as such, and because the output from the LPCXpresso can’t source enough current to provide more than a tiny output from the speaker, we utilize an LM386 to construct an amplifier for the signal.

Audio Amplification Schematic



For this project, Vs was +5V, and Vin was the summed audio signal

Un-/Partially Implemented Features

LED Screen Output

PS/2 Button Inputs

As was mentioned above, the relative closeness between the CPU clock and the keyboard's clock meant that whenever the incoming data was sampled, errors occurred at a rate of around 20%, which made the control system somewhat unreliable. A better method, namely the use of external interrupts was attempted, as this would allow the CPU to focus exclusively on the data sampling briefly, allowing for more accurate sampling.

DMA Subsystem for Color LCD Screen Use

Initially, because this project had originally been intended to incorporate a colored LCD screen, the use of the DMA subsystem was attempted. Specifically, the hope was to offset some of the work done by the CPU in transferring well-developed sinusoids to the DAC, in order to allow the CPU to focus more exclusively on the graphical requirements. An attempt is included below.

Code

Main.h

```
/*Includes*/
#include <deque>
#include <valarray>
#include <numeric>
#include <cmath>
#include <climits>
#include <stdlib.h>
#include "../src/pin.h"
#include "../src/timer.h"
#include "wave_creation.h"
#include "note_generation.h"
#include "ps2_input.h"

#define PINSEL1 (*(volatile int*)0x4002C004)
#define PINMODE1 (*(volatile int*)0x4002C044)
#define PINMODE0 (*(volatile int*)0x4002C040)
#define PINMODE3 (*(volatile int*)0x4002C04C)

//Used for setting the Digital-to-Analog subsystem values.
#define DACR (*(volatile unsigned int *)0x4008C000)

/*Main function*/
/**
 * LPC1769 Final Project: Audio Training Game
 * Generates audio signals on a pentatonic scale and awaits the proper input 
responses.
```

```
 */
int main()
{

	//Uses PS/2 Keyboard Input to begin program
	//Also keeps a counter that is used for the random number generation
seed
	bool program_start = false;
	for (int random_number_generation_seed = 0;
random_number_generation_seed++; !program_start)
	{
		//Reads in input from keyboard.
		//If a key is being pressed (that is, the interpreted keycode is
not 0), then the program will start
		if (keyInput() > 0)
		{
			program_start = true;
		}
		random_number_generation_seed %= INT_MAX;
	}

	//Seed the random number generation LCG
	srand(0);

	//The number of seconds to wait between rounds of the game
	int ticks_to_wait = seconds_to_ticks(0.001);
	//The number of seconds
	int ticks_to_wait_interior = seconds_to_ticks(0.01);

	PINSEL1 |= 1 << 21;
	PINSEL1 &= ~(1 << 20);

	PINMODE1 |= 0b11 << 23;

	//Configure response buttons
	pin a3_in = pin(0, 6);
	pin c4_in = pin(0, 1);
	pin d4_in = pin(0, 17);
	pin e4_in = pin(0, 16);
	pin g4_in = pin(0, 24);
	pin a4_in = pin(1, 31);

	//Set input buttons to have no pull-up or pull-down resistors
	PINMODE0 |= (1 << 13);
	PINMODE0 |= (1 << 3);
	PINMODE1 |= (1 << 3);
	PINMODE1 |= (1 << 1);
	PINMODE1 |= (1 << 17);
	PINMODE3 |= (1 << 31);

	//Configure response buttons to have pull-down resistors
	PINMODE1 |= (0b11 << 18);
	PINMODE1 |= (0b11 << 14);
	PINMODE0 |= (0b11 << 30);
```

```
PINMODE1 |= (0b11 << 4);
PINMODE0 |= (0b11 << 0);
PINMODE0 |= (0b11 << 14);

//Play OU bells starting music sequence
play_a4(200);
play_g4(100);
play_e4(100);
play_d4(200);
play_g4(200);
play_e4(150);
play_c4(150);
play_d4(150);
play_g4(300);

wait_ticks(seconds_to_ticks(2));

play_e4(350);
play_c4(350);
play_d4(350);
play_g4(350);
wait_ticks(seconds_to_ticks(1));
play_g4(350);
play_d4(350);
play_e4(350);
play_c4(350);

wait_ticks(seconds_to_ticks(2));

//Generate random note and receive response
while (true)
{
       //Choose a random note to play and play it
       int note_to_play = rand() % 6;
       switch (note_to_play)
       {
       case 0:
              play_a3();
              break;
       case 1:
              play_c4();
              break;
       case 2:
              play_d4();
              break;
       case 3:
              play_e4();
              break;
       case 4:
              play_g4();
              break;
       case 5:
              play_a4();
              break;
```

```
default:
        break;
}

//Stall until player selects the right note
bool wrong_entry = true;
while (wrong_entry)
{
        switch (note_to_play)
        {
        case 0:
                if (a3_in)
                {
                        wrong_entry = false;
                }
                break;
        case 1:
                if (c4_in)
                {
                        wrong_entry = false;
                }
                break;
        case 2:
                if (d4_in)
                {
                        wrong_entry = false;
                }
                break;
        case 3:
                if (e4_in)
                {
                        wrong_entry = false;
                }
                break;
        case 4:
                if (g4_in)
                {
                        wrong_entry = false;
                }
                break;
        case 5:
                if (a4_in)
                {
                        wrong_entry = false;
                }
                break;
        default:
                break;
        }

        pin(0, 25) = a3_in;
        pin(0, 23) = c4_in;
        pin(0, 15) = d4_in;
        pin(0, 18) = e4_in;
```

```
				pin(0, 0) = g4_in;
				pin(0, 7) = a4_in;

				wait_ticks(ticks_to_wait_interior);
			}

			wait_ticks(ticks_to_wait);

			//The player has gotten it right by this point. Play the
"success" jingle and start again.
			play_d4(100);
			play_g4(100);

			wait_ticks(seconds_to_ticks(1));
		}

		return 0;
}
```

pin.h

```
/*
 * Pin.cpp
 *
 *  Created on: Feb 6, 2019
 *      Author: David "Dawn" Estes McKnight
 */

#ifndef PIN_H_
#define PIN_H_

#include <string>
#include "../src/FIOprep.h"

/**
 * A class to simplify the usage of pins for DDL projects.
 */
class pin
{

public:

	/**
	 * Default constructor for the pin class.
	 */
	pin();

	/**
	 * Copy constructor for the pin class.
	 * @param orig The pin to be copied.
	 */
	pin(const pin& orig);

	/**
```

```
         * Constructs a pin object that refers to the given bit/number
corresponding to the provided port number and bit number.
         * @param port_location The number of the port.
         * @param bit_number The offset of the bit from the start of the port.
         */
        pin(unsigned int port_number, unsigned int bit_number);

        /**
         * Constructs a pin object that refers to the given bit/number
corresponding to the provided port number and bit number.
         * Also sets the pin high (value is true) or low (value is false).
         * @param port_location The number of the port.
         * @param bit_number The offset of the bit from the start of the port.
         * @param value Whether to set the pin high (true) or low (false).
         */
        pin(unsigned int port_number, unsigned int bit_number, bool value);

        /**
         * Calls write on the value
         * @param value the value to be written to this pin
         */
        void operator = (bool value);

        /**
         * Writes the value (true = high, false = low) to this pin.
         * @param value The value to write to the pin (true = high, false =
low).
         */
        void write(bool value);

        /**
         * Casts the value of this pin to the read-in value.
         */
        operator bool();

        /**
         * Reads the value (high = true, low = false) from this pin.
         * @return The value of the pin (high = true, low = false).
         */
        bool read();


protected:

        /**
         * The port number of this pin
         */
        unsigned int _port_number;

        /**
         * The bit offset of this pin (the pin number within the port).
         */
        unsigned int _bit_number;
};
```

```
#endif
```

pin.cpp

```
/*
 * Pin.cpp
 *
 *  Created on: Feb 6, 2019
 *      Author: David "Dawn" Estes McKnight
 */
#include "../src/pin.h"

pin::pin()
{
	_port_number = 0;
	_bit_number = 0;
}

pin::pin(const pin& obj)
{
	_port_number = obj._port_number;
	_bit_number = obj._bit_number;
}

pin::pin(unsigned int port_number, unsigned int bit_number)
{
	_port_number = port_number;
	_bit_number = bit_number;
}

pin::pin(unsigned int port_number, unsigned int bit_number, bool value) :
pin(port_number, bit_number)
{
	write(value);
}


void pin::operator=(bool value)
{
	write(value);
}

void pin::write(bool value)
{
	FIO[_port_number].FIODIR |= 1<<_bit_number;
	FIO[_port_number].FIOPIN &= ~(1<<_bit_number);
	FIO[_port_number].FIOPIN |= value<<_bit_number;
}

pin::operator bool()
{
	return read();
}
```

```
bool pin::read()
{
	FIO[_port_number].FIODIR &= ~(1<<_bit_number);
	return (FIO[_port_number].FIOPIN >> _bit_number) & 1;
}
```

timer.h

```
/*
 * timer.h
 *
 *  Created on: Feb 6, 2019
 *      Author: David "Dawn" Estes McKnight and Benjamin Korty
 */

#ifndef TIMER_H_
#define TIMER_H_

/**
 * Delays the program by the provided number of decrementation and
comparison operations.
 * @param count The number of decrementation and comparison operations
to perform.
 */
void wait_ticks(int count)
{
    volatile int ticks = count;
    while (ticks>0)
        ticks--;
}

/**
 * Converts the desired number of seconds to comparison+decrementation
ticks for wait_ticks
 * @param seconds the desired number of seconds for wait_ticks to wait
 * @return the number of seconds for wait_ticks to wait
 */
int seconds_to_ticks(double seconds)
{
    double ticks_unrounded = (seconds * 1000000 - 9.896)/3.464;

    //round the ticks and return
    return (int)(ticks_unrounded + 0.5);
}

#endif
```

wave_creation.h

```
/*
 * wave_creation.h
 *
 *  Created on: May 2, 2019
 *      Author: DEMcKnight
 */
#include <valarray>
```

```
#ifndef WAVE_CREATION_H_
#define WAVE_CREATION_H_

/**
 * Truncates a valarray of doubles to a valarray of ints
 * @param untruncated_array The untruncated doubles array
 * @return An integer array of the truncated values of the provided double
array
 */
std::valarray<int> truncated_array(std::valarray<double> untruncated_array)
{
	std::valarray<int> truncated_array = std::valarray<int>(
			untruncated_array.size());
	for (int index = 0; index < untruncated_array.size(); index++)
	{
		truncated_array[index] = (int) untruncated_array[index];
	}

	return truncated_array;
}

/**
 * Generates a valarray with values ranging from start through end, with the
given step size.
 * @param start The lowest value of the generated valarray.
 * @param end The upper bound on the highest value of the generated valarray.
 * @param step The spacing between members of the generated valarray
 * @return a valarray with a range of values specified by the parameters
 */
std::valarray<double> generate_range(double start, double end, double step)
{
	std::valarray<double> return_array((end - start) / step + 1);
	for (int i = 0; i < return_array.size(); i++)
	{
		return_array[i] = start + i * step;
	}
	return return_array;
}

/**
 * Renormalizes a provided wave from 0 to 1 (inclusive)
 * @param initial_wave The wave to normalize
 * @return The provided wave, renormalized from zero to one (inclusive)
 */
std::valarray<double> normalize_from_zero_to_one(
		std::valarray<double> initial_wave)
{
	//Normalize above/including 0
	std::valarray<double> nonnegative_wave = initial_wave -
initial_wave.min();

	//Normalize from 0 to 1 (inclusive)
	std::valarray<double> normalized_wave = nonnegative_wave
```

```
                    / (nonnegative_wave.max() - nonnegative_wave.min());

        return normalized_wave;
}

#endif /* WAVE_CREATION_H_ */
```

note_generation.h

```
#ifndef NOTE_GENERATION_H_
#define NOTE_GENERATION_H_
#define DACR (*(volatile unsigned int *)0x4008C000)

#include "wave_creation.h"

/**
 * Generates a ready-for-output-on-the-LM386 audio wave with the
 * given number of samples
 * @param high_value
 * @return
 */
std::valarray<int> generate_audio_wave(int num_samples)
{
        std::valarray<double> raw_sin_values = sin(
                        generate_range(0, 2 * M_PI, 2 * M_PI / (double)
num_samples));
        std::valarray<double> normalized_sin_values =
normalize_from_zero_to_one(
                        raw_sin_values);
        normalized_sin_values *= 1023;
        std::valarray<int> output_wave = truncated_array(normalized_sin_values);
        return output_wave;
}

/**
 * Plays the provided wave with the provided waittime between each sample
output
 * @param wave
 * @param waittime
 */
void play_wave_with_wait(std::valarray<int> wave, double waittime,
                int num_notes)
{
        int ticks_to_wait = seconds_to_ticks(waittime);

        int notecount = 0;
        int count = 0;
        while (notecount < num_notes)
        {
                DACR = (int) (wave[count]) << 6;
                count++;
                if (count == wave.size())
                {
                        count = 0;
                        notecount++;
```

```
            }

            wait_ticks(ticks_to_wait);
        }
}

const std::valarray<int> a_220_sin_values = generate_audio_wave(20);
/**
 * Method for playing num_notes A3 waves
 * @param num_notes the number of A3 waves to produce
 */
void play_a3(int num_notes = 250)
{
        play_wave_with_wait(a_220_sin_values, 0.0001915, num_notes);
}

const std::valarray<int> c_261_63_sin_values = generate_audio_wave(18);
/**
 * Method for playing num_notes C4 waves
 * @param num_notes the number of C4 waves to produce
 */
void play_c4(int num_notes = 250)
{
        play_wave_with_wait(c_261_63_sin_values, 0.000174, num_notes);
}

const std::valarray<int> d_293_66_sin_values = generate_audio_wave(18);
/**
 * Method for playing num_notes D4 waves
 * @param num_notes the number of D4 waves to produce
 */
void play_d4(int num_notes = 250)
{
        play_wave_with_wait(d_293_66_sin_values, 0.000153, num_notes);
}

const std::valarray<int> e_329_63_sin_values = generate_audio_wave(17);
/**
 * Method for playing num_notes E4 waves
 * @param num_notes the number of E4 waves to produce
 */
void play_e4(int num_notes = 250)
{
        play_wave_with_wait(e_329_63_sin_values, 0.000142, num_notes);
}

const std::valarray<int> g_392_sin_values = generate_audio_wave(18);
/**
 * Method for playing num_notes G4 waves
 * @param num_notes the number of G4 waves to produce
 */
void play_g4(int num_notes = 250)
{
        play_wave_with_wait(g_392_sin_values, 0.000108, num_notes);
```

```
}

const std::valarray<int> a_440_sin_values = generate_audio_wave(13);
/**
 * Method for playing num_notes A4 waves
 * @param num_notes the number of A4 waves to produce
 */
void play_a4(int num_notes = 250)
{
	play_wave_with_wait(a_440_sin_values, 0.000135, num_notes);
}

#endif /* NOTE_GENERATION_H_ */
```

ps2_input.h

```
/*
 * ps2_input.h
 *
 *  Created on: May 3, 2019
 *      Author: Ben Korty and DEMcKnight
 */

#ifndef PS2_INPUT_H_
#define PS2_INPUT_H_


//#include "timer.h"
#ifndef TIMER_H_
#include <iostream>
#include "timer.h"
#endif

#include "pin.h"

//Sample code for reading in PS/2 input

#define PINMODE0 (*(volatile int*)0x4002C040)


int keyInput()
{
	PINMODE0 |= (1<<17)|(1<<19);
	int input =0;


	bool lastClk = true; //The state of the last PS/2 clock read (starts
high)
	int ps2Bits = 0;    //The PS/2 read-in (we start with nothing)
	int numBits = 0;    //The number of bits we have read into ps2Bits. We
have no bits to start;
	pin ps2DataLine = pin(0, 8);     //the ps/2 data line pin
	pin ps2Clk = pin(0, 9);          //the ps/2 clock pin
	int ps2BitsFinal = 0;				//where to store a full read
	bool fullRead = false;

	pin one pressed = pin (0,7);
	pin two pressed = pin (0,0);
	pin three pressed = pin(0,18);
	pin four pressed = pin(0,15);
	pin five pressed = pin(0,23);

	bool clockChanged = ((bool)ps2Clk.read()!=lastClk);
	if (clockChanged) //If the clock changed
	{
		while(numBits<11){
			lastClk = !lastClk;		//the clock has changed
			if (!(bool)ps2Clk.read()) //if the clock is low, read in
new bit into ps2Bits
			{
				ps2Bits = ps2Bits << 1;
	//make new room for new bit
				ps2Bits |= ps2DataLine.read();	//insert new bit
with bitwise or
				numBits++;
	//we've read in one more bit, so increase this counter
			}

			if (numBits == 11)
			{
				fullRead = true;
				ps2BitsFinal = ps2Bits;				//Store
fully-read bits in ps2BitsFinal
				ps2Bits = 0;					//Clear
ps2Bits
				numBits = 0;					//Clear the
number
				ps2BitsFinal = ps2BitsFinal >> 2;
				ps2BitsFinal = ps2BitsFinal & 0b11111111;
				return ps2BitsFinal;
			}
		}
	}
	return 0;
}

#endif

/* PS2_INPUT_H_ */
```

DMAtransfer.h

```
/*
 * DMAtransfer.h
 *
 *  Created on: May 1, 2019
 *      Author: kort8201
 */

#ifndef DMATRANSFER_H_
#define DMATRANSFER_H_

#include "../src/pin.h"

#define PCONP (*(volatile int*)0x400FC0C4)
#define PINSEL1 (*(volatile int*)0x4002C004)
#define PINMODE1 (*(volatile int*)0x4002C044)
#define DACCTRL (*(volatile int*)0x4008C004) //DAC control register. Used to
request data transfer using DMA
#define DACR (*(volatile unsigned int *)0x4008C000)
#define DMACConfig (*(volatile unsigned int *)0x50004030) //Used to activate
DMA controller
#define DMACIntTCClear (*(volatile unsigned int *)0x50004008) //Used to clear
Terminal Interrupts
#define DMACIntErrClear (*(volatile unsigned int *)0x50004010)//Used to clear
Error based Interrupts
#define DMACC0SrcAddr (*(volatile unsigned int *)0x50004100)//Used to indicate
to DMA controller the location in memory to read from
#define DMACC0DestAddr (*(volatile unsigned int *)0x50004104)//Used to
indicate to DMA controller the location in the DAC subsystem being written to
#define DMACC0LLI (*(volatile unsigned int *)0x50004108)//Used to indicate to
DMA controller the location of next linked list item used.
                                                        //In this case, no
other items are used per transaction, so register will
#define DMACC0Control (*(volatile unsigned int *)0x5000410C)//Used to indicate
the transfer size of the upcoming transaction (in shorts)
#define DMACC0Config (*(volatile unsigned int *)0x50004110)//Used to indicate
the transfer size of the upcoming transaction (in shorts)

int DMAtransfer(int inputInts[], int numInputs)
{
	//Setup for DAC
	PINSEL1 |= 1 << 21;			//Select appropriate mode for output
pin for DAC out
	PINSEL1 &= ~(1<<20);

	//Setup for DMA
	PCONP |= 1 << 29;			//Power the DMA controller
	DMACConfig |= 1;			// Enable the DMA controller

	//DMA Channel 0 Initiation
	DMACIntTCClear |= 1;			//Clear Interrupts on Channel 0
	DMACIntErrClear |= 1;
	DMACC0SrcAddr = (int)inputInts;		//Write memory location of data
to be transferred as SOURCE location
	DMACC0DestAddr = 0x4008C000;		//Write DACR as location to be
transferred to aka DESTINATION location
	DMACC0LLI = 0;				//Point next linked list item
address to 0
```

```
	//DMA Channel 0 Control
	DMACC0Control |= numInputs;		//Sets the number of shorts to be 
transferred.  4 bytes per int.
	DMACC0Control &= ~(0b111<<12);	//Sets source burst size to 1. Burst 
is number of transfer per transaction.
	DMACC0Control &= ~(0b111<<15);	//Sets destination burst size to 1. 
Burst is number of transfer per transaction.
	DMACC0Control |= (1<<19);		//Sets source transfer width to 32 
bits (0b010 for 32 bits)
	DMACC0Control |= (1<<22);		//Sets destination transfer width 
to 32 bits (0b010 for 32 bits)
	DMACC0Control |= 1<<26;			//Enables source address 
incrementation
	DMACC0Control &= ~(1<<27);		//Disables destination address 
incrementation

	DMACC0Config;

	//DMA Channel 0 Configuration
	DMACC0Config = 1<<15;			//Masks TC Interrupt
	DMACC0Config |= (7<<6)|(1<<11);		//Sets DAC peripheral as transfer 
destination (7), and transfer type as memory to peripheral (0b001)
	DMACC0Config |= 1;			//Enables Channel 0. Channel is now 
active.

	//Request a DMA transfer
	DACCTRL |= 1 <<3;			//Enable DMA Burst Requests for DAC 
to DMA

	while((DMACC0Config&1)==1)
	{
		DMACC0Config;
		DACR;
	}

	int output = DACR>>6;
	output = output&0b1111111111;
	return output;
}

#endif /* DMATRANSFER_H_ */
```