Arduino Synthesizer
Far-out Frequency Frazzelator

ECE 437 Final Project

By: Alex Faron & Thomas Frazel

Table of Contents

Description

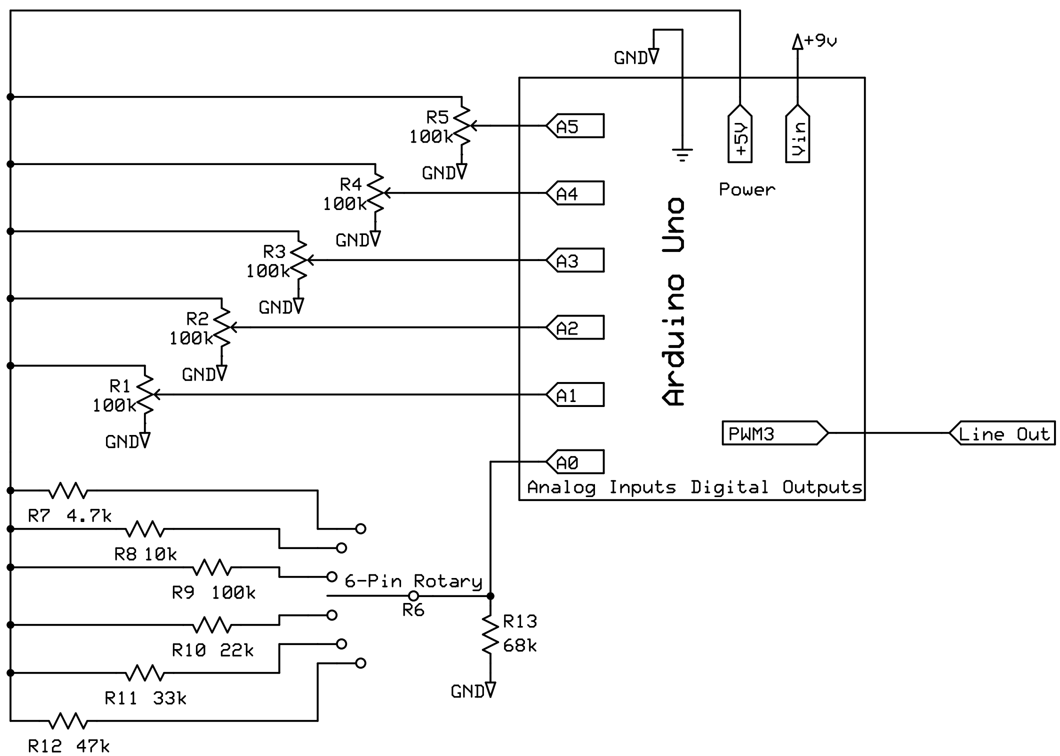
Our synthesizer uses the Arduino Uno microprocessor to generate a PWM triangle wave output in one of 5 different musical scales or a linear frequency mapping, selected using the selector knob. Additionally, the synthesizer has two different grain volumes—summed at the output—and each grain has a dedicated delay. The output frequency knob changes the note (or frequency) being played. The synthesizer is fully reprogrammable, allowing the user to change which musical scales are programmed for the synthesizer and change which control is mapped to which knob. Below is a table of included controls.

Controls (default, clockwise from center)

Output Mode: C, dm, E♭, F#, G, Smooth-mapping
Grain 1 Volume
Grain 1 Decay
Frequency
Grain 2 Decay
Grain 2 Volume

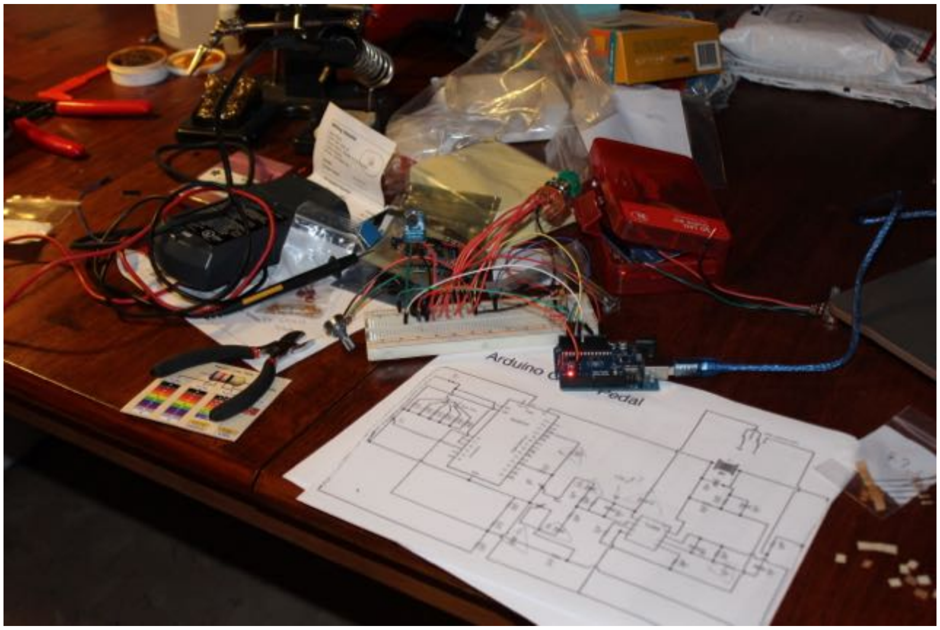
Design Process

Our design began as a guitar effects pedal that would use a rotary switch to choose between various Arudino audio effects. We morphed our design into a synthesizer that used the rotary switch to change between different output modes (different scales or a smooth frequency mapping); we chose to change the design because we wanted a project more focused on DSP and less focused on the electronics that the pedal required. We based our design on a project we found online (mentioned in the acknowledgements section), but added our own twist with the rotary switch to implement a fully re-programmable scale selector (also included is custom C++ code to generate the scales for the synthesizer) and made the output stereo. We drew out a circuit; programmed the Arduino; soldered and wired all the inputs, resistors, and the rotary switch; and encased the synthesizer to make it a playable instrument. All of these steps are documented in greater detail below.

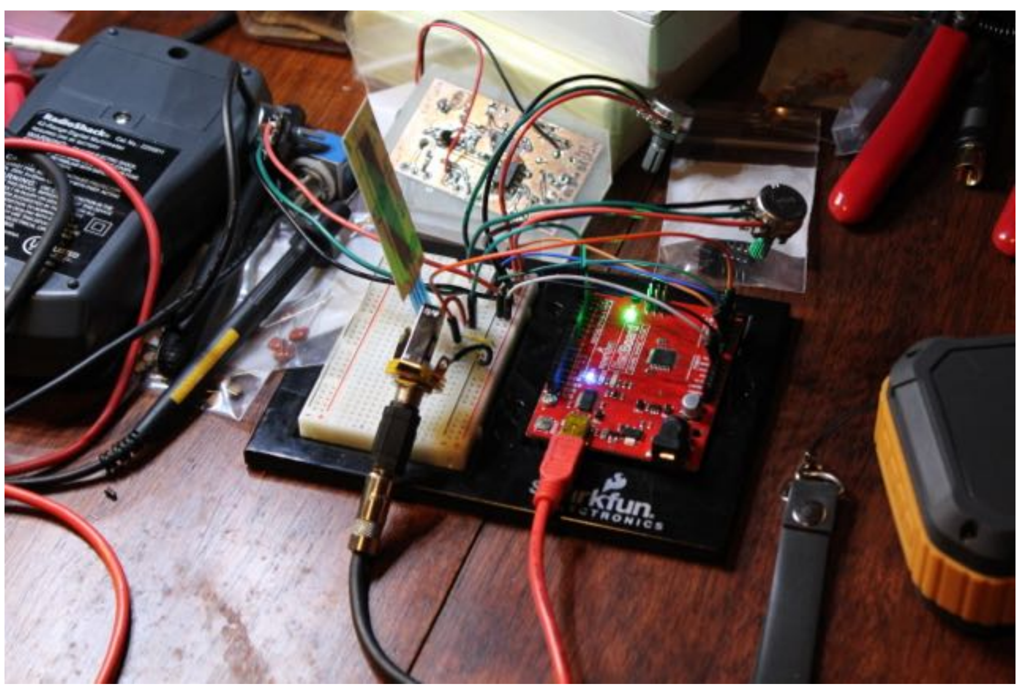


University of Miami		
Arduino Synthesizer		
Alex Faron Thomas Frazel	Rev 1.0 12/10/2015	ECE 437

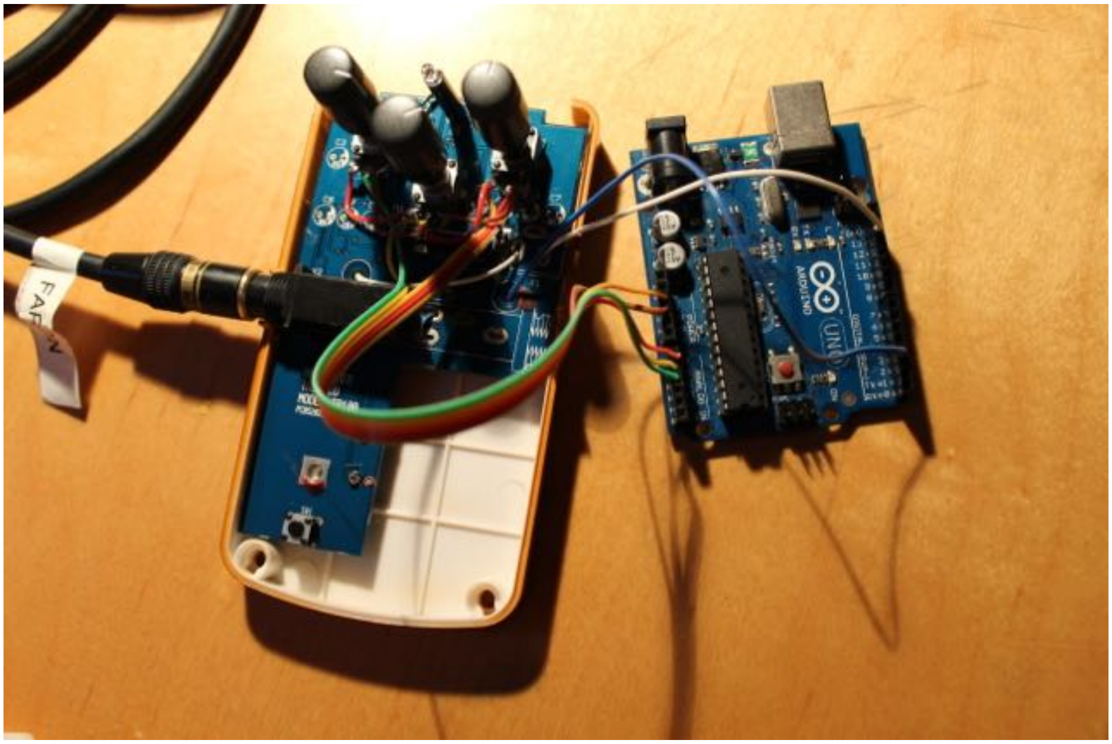
Construction



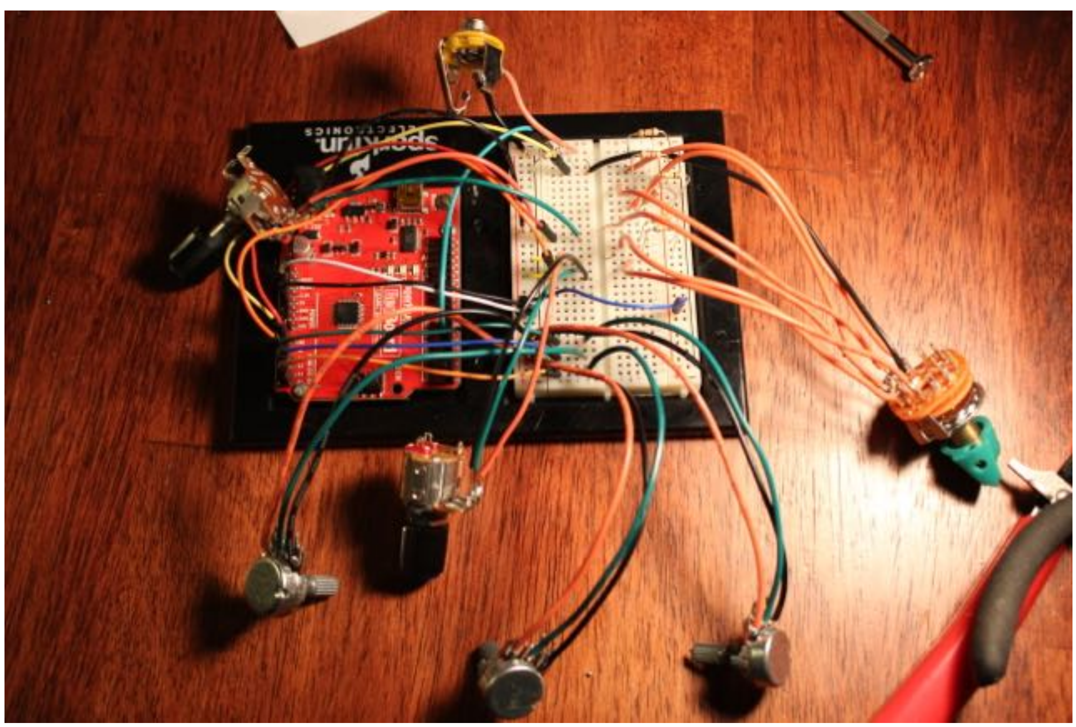
Phase 1: Attempt at making a guitar pedal



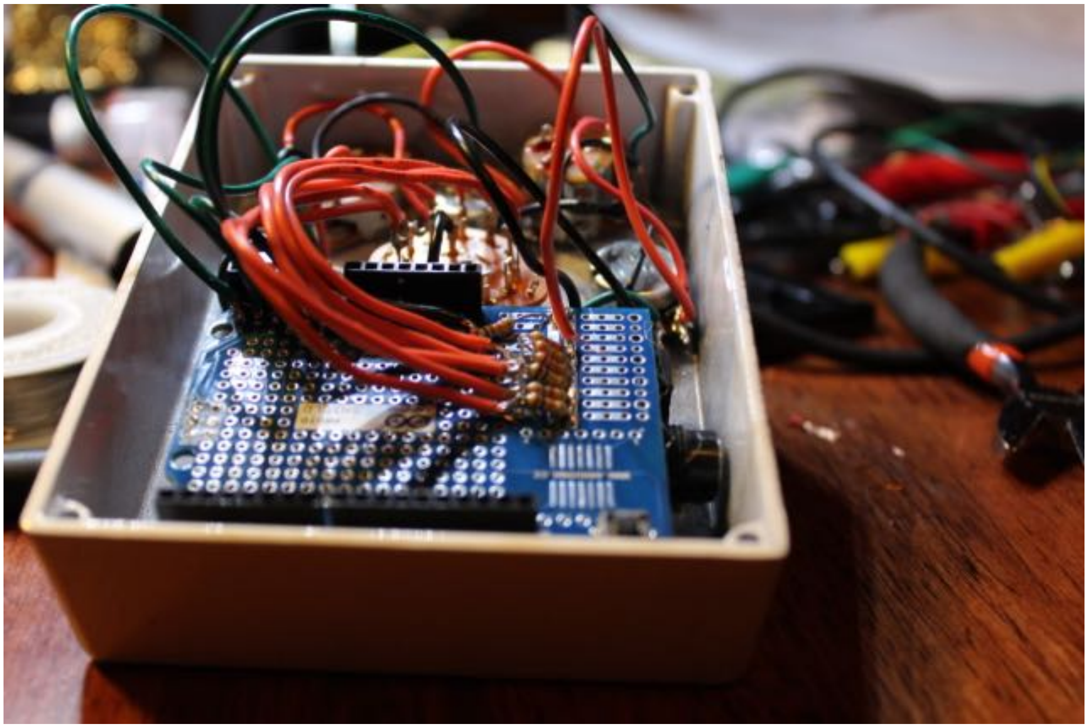
Phase 2: Synthesizer prototype 1



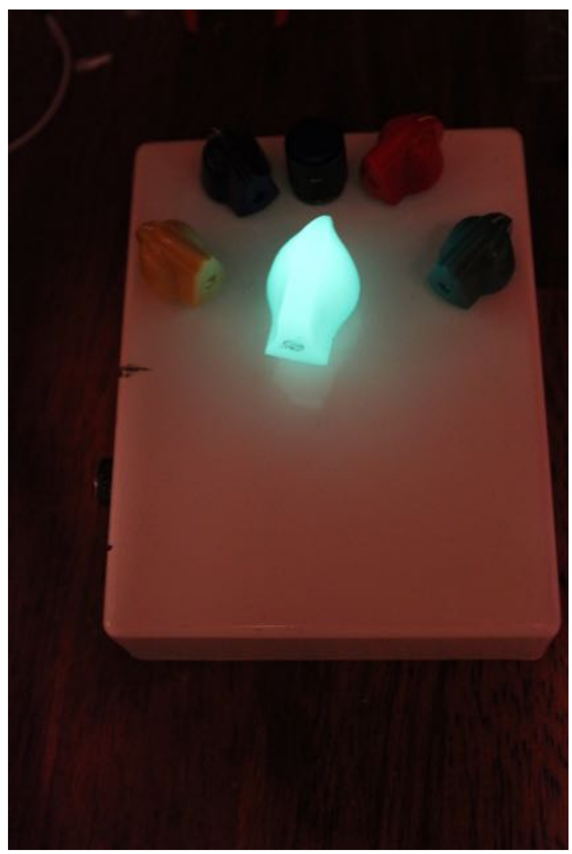
Phase 3: Synthesizer prototype 2



Phase 4: Final Synthesizer



Phase 5: Pictures of the final synthesizer in casing



Code

Arduino Code

```
// Far-out Frequency Frazzelator
// By: Alex Faron and Thomas Frazel
//

// Analog in 0: Output Mode Selector
// Analog in 1: Grain 1 pitch
// Analog in 2: Grain 2 decay
// Analog in 3: Grain 1 decay
// Analog in 4: Grain 2 pitch
// Analog in 5: Grain repetition frequency
//
// Digital 3: Audio out
//

#include <avr/io.h>
#include <avr/interrupt.h>

uint16_t syncPhaseAcc;
uint16_t syncPhaseInc;
uint16_t grainPhaseAcc;
uint16_t grainPhaseInc;
uint16_t grainAmp;
uint8_t grainDecay;
uint16_t grain2PhaseAcc;
uint16_t grain2PhaseInc;
uint16_t grain2Amp;
uint8_t grain2Decay;

// Map Analogue channels
#define MODE_SELECT          (0)
#define GRAIN_PITCH_CONTROL   (1)
#define GRAIN2_PITCH_CONTROL  (5)
#define GRAIN_DECAY_CONTROL  (2)
#define GRAIN2_DECAY_CONTROL (4)
#define SYNC_CONTROL         (3)

#define PWM_PIN_L     3
#define PWM_VALUE_L   OCR2B
#define PWM_PIN_R     11
#define PWM_VALUE_R   OCR2A
#define LED_PIN       13
#define LED_PORT      PORTB
#define LED_BIT       5
#define PWM_INTERRUPT TIMER2_OVF_vect

// Smooth logarithmic mapping
//
uint16_t antilogTable[64] = {
```

```
64830,64132,63441,62757,62081,61413,60751,60097,59449,58809,58176,5754
9,56929,56316,55709,55109,

54515,53928,53347,52773,52204,51642,51085,50535,49991,49452,48920,4839
3,47871,47356,46846,46341,

45842,45348,44859,44376,43898,43425,42958,42495,42037,41584,41136,4069
3,40255,39821,39392,38968,

38548,38133,37722,37316,36914,36516,36123,35734,35349,34968,34591,3421
9,33850,33486,33125,32768
};

uint16_t mapPhaseInc(uint16_t input) {
  return (antilogTable[input & 0x3f]) >> (input >> 6);
}

// b minor
//
uint16_t midiTableb[64] = {
18,21,22,24,28,31,33,37,41,44,49,55,62,65,73,82,87,98,110,123,131,147,
165,175,196,220,247,
262,294,330,349,392,440,494,523,587,659,698,784,880,988,1047,1175,1319
,1397,1568,1760,1976,
2093,2349,2637,2794,3136,3520,3951,4186,4699,5274,5588,6272,7040,7902,
8372,9397
};

uint16_t mapMidi_b(uint16_t input) {
  return (midiTableb[(1023-input) >> 4]);
}

// Eb Major
//
uint16_t midiTableEb[64] = {
19,22,24,26,29,33,37,39,44,49,52,58,65,73,78,87,98,104,117,131,147,156
,175,196,208,233,262,
294,311,349,392,415,466,523,587,622,698,784,831,932,1047,1175,1245,139
7,1568,1661,1865,2093,
2349,2489,2794,3136,3322,3729,4186,4699,4978,5588,6272,6645,7459,8372,
9397,9956
};

uint16_t mapMidi_Eb(uint16_t input) {
  return (midiTableEb[(1023-input) >> 4]);
}

// F Major
//
uint16_t midiTableFs[64] = {
23,26,29,31,35,39,44,46,52,58,62,69,78,87,92,104,117,123,139,156,175,1
85,208,233,247,277,311,
```

```
349,370,415,466,494,554,622,698,740,831,932,988,1109,1245,1397,1480,16
61,1865,1976,2217,2489,
2794,2960,3322,3729,3951,4435,4978,5588,5920,6645,7459,7902,8870,9956,
11175,11840
};

uint16_t mapMidi_Fs(uint16_t input) {
  return (midiTableFs[(1023-input) >> 4]);
}

// G Major
//
uint16_t midiTableG[64] = {

24,28,31,33,37,41,46,49,55,62,65,73,82,92,98,110,123,131,147,165,185,1
96,220,247,262,294,

330,370,392,440,494,523,587,659,740,784,880,988,1047,1175,1319,1480,15
68,1760,1976,2093,

2349,2637,2960,3136,3520,3951,4186,4699,5274,5920,6272,7040,7902,8372,
9397,10548,11840,12544
};

uint16_t mapMidi_G(uint16_t input) {
  return (midiTableG[(1023-input) >> 4]);
}

// C Major
//
uint16_t midiTableC[64] = {
33,37,41,44,49,55,62,65,73,82,87,98,110,123,131,147,165,175,196,220,24
7,262,294,330,349,
392,440,494,523,587,659,698,784,880,988,1047,1175,1319,1397,1568,1760,
1976,2093,2349,2637,
2794,3136,3520,3951,4186,4699,5274,5588,6272,7040,7902,8372,9397,10548
,11175,12544,14080,
15804,16744
};

uint16_t mapMidi_C(uint16_t input) {
  return (midiTableC[(1023-input) >> 4]);
}

// Map the selector knob
//
uint16_t mapControl(uint16_t input){
  if (input < 500){
    return 0;
  }
  else if (input < 640){
    return 1;
  }
```

```
  else if (input < 730){
    return 2;
  }
  else if (input < 840){
    return 3;
  }
  else if (input < 925){
    return 4;
  }
  else{
    return 5;
  }
}

// Turn audio on
//
void audioOn() {
  // Set up PWM to 31.25kHz, phase accurate
  TCCR2A = _BV(COM2B1) | _BV(WGM20);
  TCCR2B = _BV(CS20);
  TIMSK2 = _BV(TOIE2);
}

// Setup
//
void setup() {
  pinMode(PWM_PIN_L,OUTPUT);
  pinMode(PWM_PIN_R,OUTPUT);
  audioOn();
  pinMode(LED_PIN,OUTPUT);
}

// Loop
//
void loop() {

  // Determine the output mode
  //
  switch(mapControl(analogRead(MODE_SELECT))){
    case 0: // Smooth Mapping
      syncPhaseInc = mapPhaseInc(analogRead(SYNC_CONTROL)) / 4;
      break;
    case 1: // bm Scale
      syncPhaseInc = mapMidi_b(analogRead(SYNC_CONTROL));
      break;
    case 2: // Eb Scale
      syncPhaseInc = mapMidi_Eb(analogRead(SYNC_CONTROL));
      break;
    case 3: // Fs Scale
      syncPhaseInc = mapMidi_Fs(analogRead(SYNC_CONTROL));
      break;
    case 4: // G Scale
      syncPhaseInc = mapMidi_G(analogRead(SYNC_CONTROL));
```

```
      break;
    case 5: // C Scale
      syncPhaseInc = mapMidi_C(analogRead(SYNC_CONTROL));
      break;
  }

  // Update the grain parameters
  //
  grainPhaseInc  = mapPhaseInc(analogRead(GRAIN_PITCH_CONTROL)) / 2;
  grainDecay     = analogRead(GRAIN_DECAY_CONTROL) / 8;
  grain2PhaseInc = mapPhaseInc(analogRead(GRAIN2_PITCH_CONTROL)) / 2;
  grain2Decay    = analogRead(GRAIN2_DECAY_CONTROL) / 4;
}

// Interupt
//
SIGNAL(PWM_INTERRUPT)
{
  uint8_t value;
  uint16_t output;

  syncPhaseAcc += syncPhaseInc;
  if (syncPhaseAcc < syncPhaseInc) {
    // Time to start the next grain
    grainPhaseAcc = 0;
    grainAmp = 0x7fff;
    grain2PhaseAcc = 0;
    grain2Amp = 0x7fff;
    LED_PORT ^= 1 << LED_BIT;
  }

  // Increment the phase of the grain oscillators
  grainPhaseAcc += grainPhaseInc;
  grain2PhaseAcc += grain2PhaseInc;

  // Convert phase into a triangle wave
  value = (grainPhaseAcc >> 7) & 0xff;
  if (grainPhaseAcc & 0x8000) value = ~value;
  // Multiply by current grain amplitude to get sample
  output = value * (grainAmp >> 8);

  // Repeat for second grain
  value = (grain2PhaseAcc >> 7) & 0xff;
  if (grain2PhaseAcc & 0x8000) value = ~value;
  output += value * (grain2Amp >> 8);

  // Make the grain amplitudes decay by a factor every sample
(exponential decay)
  grainAmp -= (grainAmp >> 8) * grainDecay;
  grain2Amp -= (grain2Amp >> 8) * grain2Decay;

  // Scale output to the available range, resolve clipping
  output >>= 9;
```

```
  if (output > 255) output = 255;

  // Output to PWM
  PWM_VALUE_L = output;
  PWM_VALUE_R = output;

}
```

C++ Code To Generate Scales

```
#include <iostream>
#include <math.h>

using namespace std;

void generateFrequencies(int v[]){
 for (int i = -64; i < 64; i++)
  v[i+64] = (int)(pow(2,((double)i/12))*440+0.5);
}

void printAllFrequencies(int v[]){
 for (int i = 0; i<128; i++)
  cout<<v[i]<<",";
}

void printMajorScale(int v[], int start){
 if (start > 19){
  cout<<"Improper usage\n";
  return;
 }
 int j;
 for(j = start; j <(start+9*12); j+=12)

cout<<v[j]<<","<<v[j+2]<<","<<v[j+4]<<","<<v[j+5]<<","<<v[j+7]<<","<<v
[j+9]<<","<<v[j+11]<<",";
 cout<<v[j]<<endl;
}

void printMinorScale(int v[], int start){
 if (start > 19){
  cout<<"Improper usage\n";
  return;
 }
 int j;
 for(j = start; j <(start+9*12); j+=12)

cout<<v[j]<<","<<v[j+2]<<","<<v[j+3]<<","<<v[j+5]<<","<<v[j+7]<<","<<v
[j+9]<<","<<v[j+10]<<",";
 cout<<v[j]<<endl;
}

void main(){
 int v[128];
```

```
 generateFrequencies(v);
 //printAllFrequencies(v);
 // Respective Scale Values:
 //   D:9, Ds/Eb:10, E/Fb:11, Es/F:12, Fs/Gb:13, G:14,
 //   Gs/Ab:15, A:16, As/Bb:17, B/Cb:18, C/Bs:19
 // the number in the both print frequencies is the tonic of the scale
 // choice major or minor
 printMajorScale(v,13);
 //printMinorScale(v,9);
}
```

What We Learned

Through building a reprogrammable synthesizer using an Arduino, we reinforced many digital signal processing concepts. The biggest challenge to overcome was to learn how to generate tones and perform DSP using the hardware. The sample rate of the Arduino must be considered in order to know what frequencies can accurately be reproduced; the Arduino can produce PWM at a max frequency of 62.5kHz. Reading and writing frequency-tables was necessary in order to vary the sounds synthesized. We also used concepts from senior project such as time management and teamwork skills to divide tasks and work in conjunction. Assembling the house required drilling, dremeling, and soldering. Experimenting with Stereo TRS vs. Mono TS jack wiring resulted in stereo jack output for the final.

Acknowledgments

> http://www.instructables.com/id/Arduino-Guitar-Pedal/?ALLSTEPS
This tutorial covered how to build a reprogrammable guitar pedal using a TL082 IC as the pre-amp for the input and output of the signal.

> https://code.google.com/p/tinkerit/wiki/Auduino
A great tutorial on making a granular synthesizer using the Arduino.

> http://makezine.com/projects/make-35/advanced-arduino-sound-synthesis/
A reference for Arduino sound synthesis parameters.