Jonathan Brophy & Matt Crussell

CPE 329 – 03/04

Spring 2012

Final Project

8-Bit Harmony

6/1/12

Professor Oliver

Purpose: The purpose of this experiment was to generate sine waves at different frequencies, corresponding to eight keys in a musical octave (seven keys in a musical octave from C to B, plus one extra C in the next octave). We could set each frequency to an individual button, making that button act like a key. We could then output the waveform through a DAC, and then amplify the signal and output the result through a speaker, creating our synthesized keyboard.

Design-Process:

System Requirements: This project aims to create a synthesized keyboard by generating a sine wave at different frequencies and outputting that sine wave to a speaker to generate a tone. We will have eight buttons that manipulate the frequency of the sine wave, and the wave will only be generated while the button is pressed. These buttons correspond from C3 to C4 in musical notes. We have also implemented a separate button that shifts the all the notes of the buttons up one octave, so now the same buttons will correspond to notes C4 to C5; if that separate button is pressed again, the notes will return to C3 to C4. These notes can only be played one tone at a time since we are only using one DAC and outputting one sine wave.

System Specification: In order to generate these sine waves, we populated an array with values for a sine wave and then used SPI to transmit a sine wave with our microcontroller similar to what we did for our waveform generator in project two. We had to once again use a DAC to output this sine wave. We then connected a speaker to the output of our DAC, but realized that the tone being generated was too quiet. We only used a five-volt supply from our microcontroller and so our input signal was much too small to hear for a proper demo, but did work very well for ear-bud type headphones. We then ran the output from our DAC through an audio amplifier (LM386) to our much bigger speaker getting a gain of approximately twenty through our amplifier. This gain was still too large even for our larger speakers; we implemented a voltage divider at the input to get a smaller input signal. After this was accomplished and we had a decent volume for our sized speaker, we were able to implement eight buttons that would act as keys on a keyboard. Our musical notes range from C3 to C4 from left to right, and then a separate button is toggled to change the range from C4 to C5 again looking left to right. We correspond each button press to a sine wave being transmitted at a particular frequency that corresponds to the musical note it is placed at. This octave of notes can be toggled back and forth using that separate button.

System Architecture: Figure 1 below shows how the Arduino is wired to the external DAC, audio amplifier, and speaker; as well as to the button circuitry as well, which can be found in more detail in Figure 2.

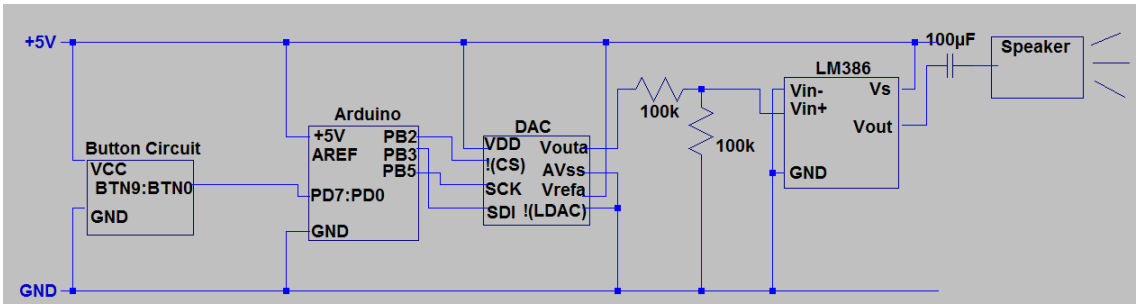

Figure 1 Top-Level Design Block Diagram

Component Design: The following wiring tables can be used to find exactly which pins on the Arduino Uno were used and hooked up to either an external button or digital to analog converter, as outlined in Table 2.

Table 1. Wiring between external LCD Screen and the external DAC

DAC	**Arduino Uno**
!(CS)	PORTB: PB2
SDI	PORTB: PB3
SCK	PORTB: PB5
VDD/Vrefa	+5V
AVss/!(LDAC)	GND
Vout	LM386

Table 1 (above) represents the wiring configurations used when hooking up the external DAC to our Arduino Uno in conjunction with a small breadboard, as Table 2 (below) shows the external buttons (keyboard) used in this project.

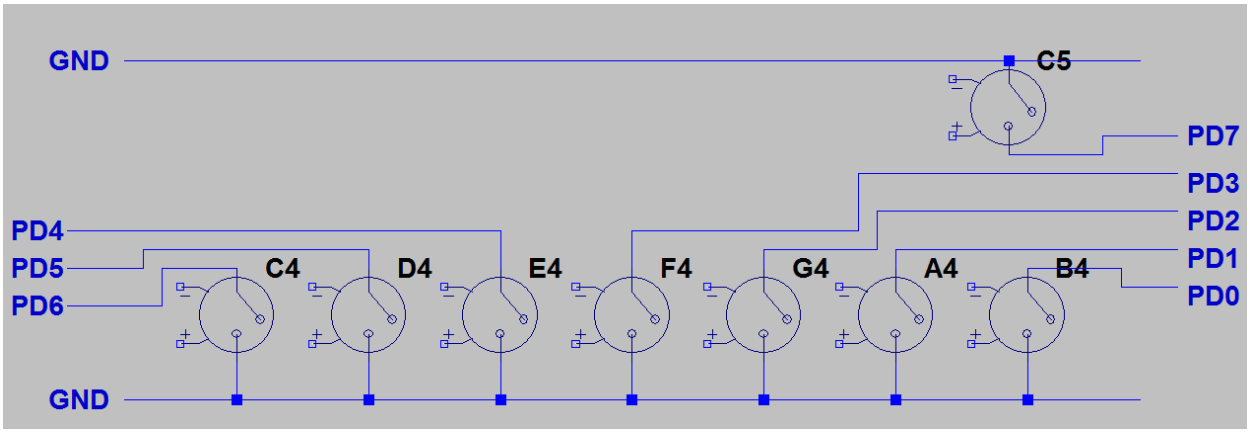

Figure 2 External Button Circuit for our Keyboard

Table 2. Wiring for external buttons (keyboard)

Buttons	**Arduino Uno**
Button 1 (C4)	PORTD: PD6
Button 2 (D4)	PORTD: PD5
Button 3 (E4)	PORTD: PD4
Button 4 (F4)	PORTD: PD3
Button 5 (G4)	PORTD: PD2
Button 6 (A4)	PORTD: PD1
Button 7 (B4)	PORTD: PD0
Button 8 (C5)	PORTD: PD7

//System Integration: For this device we used AVR Studio 4 to code and build, then AVR Dude to program the Arduino Board.

The first component we needed to test for proper functionality was the DAC. When we were programming the Function Genarator, we first implemented the DAC code. The task was to transmit a simple square wave onto the oscilloscope. When we tested the DAC, we noticed that our voltages were a tenth of the value (5V) that they should be. We discovered that the Oscilloscope probe was not resting properly.

The next components we tested were the sine and sawtooth wave codes. We performed a non-interrupt-manual cycle of each wave to verify their proper outputs. We again had an issue with the Peak-to-Peak voltage, but quickly realized that we accidently set our peak voltage to 0xAA instead

of 0xAAA.

At this point we calibrated our Interrupt so that the Function Generator worked properly at 100Hz. By varying the number of points displayed, we could easily change between frequencies.

While implementing our buttons, we had several issues with timing. The buttons switched too fast, so we implemented sub-timers that counted up to a value (10,000) before changing the respective values i.e. voltage, frequency, wave-type, and duty cycle. After adding the button codes, our timer needed to be recalibrated to the 100Hz mark.

Conclusion: Getting the right tones to output to our speaker took most of the time when working on this project. We were able to generate the sine waves fairly easily since we already did that in our second project, but then implementing them to each individual button and working constantly with button de-bouncing became a real issue. We finally managed to come up with a solution that used precision to take care of button de-bouncing and we manipulated the code to play the tone of the button that was last pressed; so if you were to hold down one button and toggle another button while still holding down the original, you would hear the tone go back and forth from whatever button was pressed and held last.

One of the other difficulties we had was creating a tone that would be loud enough to demo with. For most of our project, we used ear bud headphones to test our system and practice with our keyboard, we knew this would not be suitable during a demo and so we implemented an audio amplifier (LM386) to the output of the DAC. We changed out our ear bud headphones with much larger speakers that were able to handle this much larger output, but it was still too large and our output was clipping, which in turn affected our sine wave and gave us terrible sounding tones. We used a simple voltage divider at the input of the amplifier as well as diving the input of the sine wave in our SPI transmit software code in order to receive an amplified output that would be at a suitable volume with the speaker we were using for the demo we would perform. After this was accomplished we added a special feature that consisted of a separate button that would change all of the notes on our keyboard to one octave lower, and if toggled again would change back to the original octave. Working through these other obstacles, we came to the conclusion that fine-tuning our notes and keeping our sine wave smooth and unclipped was the most difficult part of creating this keyboard, especially transitioning between notes or frequencies.

//Appendices:

C File:

```
/*
 * Project #2:
 * Waveform Generator C file that creates a waveform generator and
interface three buttons and one switch
 *
 * Created by Jonathan Brophy and Matt Crussell
 *
```

```
 */

#define F_CPU 16000000              //define internal CLK speed
#define MOSI 3                    // PB pin 3
#define SCK  5                    // PB pin 5
#define SS   2                    // PB pin 2
#define WTYPE 1                       // PB pin 1 chooses wave type
#define FREQ 0                        // PB pin 0 choose frequency
#define DUTY 4                        // PB pin 4 choose duty cycle
#define VOLT 2                        // PD pin 2 choose volt vals
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <math.h>

void Initialize_SPI_Master(void);
void Transmit_SPI_Master(int Data);
void initTimer0(void);
void populateSine(void);
void populateSawtooth(void);
void handleFreq(void);
void handleWave(void);
void handleDuty(void);
void handleVolt(void);
int wave, duty, freq, volt;
int waveIter, freqTimer, dutyTimer, waveTimer, voltTimer;
int sine[360]; // Sineval of N degrees multipled by 0xAA dc shifted
0xAA/2
int sawtooth[360]; //Sawtooth Vals
int main(void)
{
   waveIter = 0;
   //volt = 1;
   wave = 1;
   duty = 180;
   initTimer0();
   freq = 1;
   DDRB = 1<<MOSI | 1<<SCK | 1<<SS | 1 <<WTYPE;   // make MOSI, SCK
and SS outputs
   DDRB &= ~(1<<FREQ | 1 << WTYPE | 1 << DUTY);
   DDRD = 1 << VOLT;
   PORTD |= 1 << VOLT;
   PORTB |= 1 << FREQ;
   PORTB |= 1 << WTYPE;
   PORTB |= 1 << DUTY;
   populateSine();
   populateSawtooth();
   Initialize_SPI_Master();
   sei();
```

```
   while(1) // :)
   {
   }
   return 0;
}  // end main


void Initialize_SPI_Master(void)
{
   SPCR = (0<<SPIE) |         //No interrupts
   (1<<SPE) |              //SPI enabled
   (0<<DORD) |             //shifted out LSB
   (1<<MSTR) |             //master
   (1<<CPOL) |             //rising leading edge
   (0<<CPHA) |             //sample leading edge
   (0<<SPR1) | (0<<SPR0) ;     //clock speed
   SPSR = (0<<SPIF) |         //SPI interrupt flag
   (0<<WCOL) |             //Write collision flag
   (0<<SPI2X) ;             //Doubles SPI clock
   PORTB = 1 << SS;          // make sure SS is high


}

void Transmit_SPI_Master(int Data) {

    PORTB = 0 << SS;                  //Assert slave select
    SPDR = ((Data >> 8) & 0xF) | 0x70;     //Attach configuration Bits
onto MSB
    while (!(SPSR & (1<<SPIF)));
     SPDR = 0xFF & Data;
    while (!(SPSR & (1<<SPIF)));
     PORTB = 1 << SS;
}
void initTimer0(void)
{
    OCR2A = 165; //Set number to Count To
    TCCR2A |= (1 << WGM21);// Set to CTC Mode
    TIMSK2 |= (1 << OCIE2A);//Set interrupt on compare match
    TCCR2B |= (1 << CS21);// set prescaler to 64 and starts PWM

}
void populateSine()
{
    int i;
    for(i = 0; i < 360; i++)
     sine[i] = 0xFFF/2 * (sin(((double)i) * M_PI / 180)) + 0xFFF/2;//
Shift 0xAAA/2 = 0x55
}
void populateSawtooth()
```

```
{
     int i;
     for(i = 0; i < 360; i++)
      sawtooth[i] = (int)(0xFFF - (0xFFF*((double)i))/360);
}

ISR(TIMER2_COMPA_vect)
{
     handleVolt();
     handleFreq();
     handleWave();
     handleDuty();
     if(wave == 0) //Sine Wave
      Transmit_SPI_Master(sine[3* freq * waveIter++]/volt);
     else if(wave == 1)//Square Wave
     {
      if(waveIter++ < duty/freq/3)
            Transmit_SPI_Master(0xFFF/volt);
         else
            Transmit_SPI_Master(0x000);
     }
     else if(wave == 2)//Sawtooth
      Transmit_SPI_Master(sawtooth[3* freq * waveIter++]/volt);
     if(waveIter >= 360/freq/3)
     {
      waveIter = 0;
     }
}
void handleVolt()
{
     if(voltTimer == 10000)
     {
      if(volt++ == 5)
            volt = 1;
      voltTimer = 0;
     }
     if(bit_is_clear(PIND, VOLT))
     {
      PORTD |= 1 << FREQ;
      voltTimer++;
     }
}
void handleFreq()
{
     if(freqTimer == 10000)
     {
      if(freq++ == 5)
            freq = 1;
      freqTimer = 0;
     }
```

```
     if(bit_is_clear(PINB, FREQ))
     {
      PORTB |= 1 << FREQ;
      freqTimer++;
     }
     else
      freqTimer = 0;

}
void handleWave()
{
     if(waveTimer == 10000)
     {
      if(wave++ == 2)
            wave = 0;
      waveTimer = 0;
     }
     if(bit_is_clear(PINB, WTYPE))
     {
      PORTB |= 1 << WTYPE;
      waveTimer++;
     }
     else
      waveTimer = 0;

}
void handleDuty()
{
     if(dutyTimer == 10000)
     {
      if(duty == 360)
            duty = 0;
      else
            duty+=36;
      dutyTimer = 0;
     }
     if(bit_is_clear(PINB, DUTY))
     {
      PORTB |= 1 << DUTY;
      dutyTimer++;
     }
     else
      dutyTimer = 0;

}
```

References:

Atmega 328P Datasheet: *http://www.atmel.com/Images/8271S.pdf*

Button Debouncing Information:
http://www.newbiehack.com/UnderstandingButtonDebouncing.aspx