Juntao Wang & Wei Wu

CS 281, Spring 2019

Dr. Thomas Bressoud

5/6/2019

Final Project: Multi-tone Arduino Piano

Introduction

We made an Arduino piano. It has eight buttons to play keys ranging from C4 to C5, and each button has a corresponding white LED. There are three buzzers with corresponding LEDs to play the notes, but each of them are controlled individually using “Tone.h” library. This feature allows the piano to play chords instead of single notes. The piano also has a music player mode, which can be toggled with a button. In this mode, a green LED will be turned on, and the piano will play one pre-loaded song while all the piano keys are locked. In order to control large quantity of buttons and LEDs, the CD4021BE chip and the 74HC595 chip are used. In our circuit, six pins are used to control eight inputs and eight outputs. Although the hardware side of this project is not complex, it takes more effort to write code that allows all the components to work together as desired.

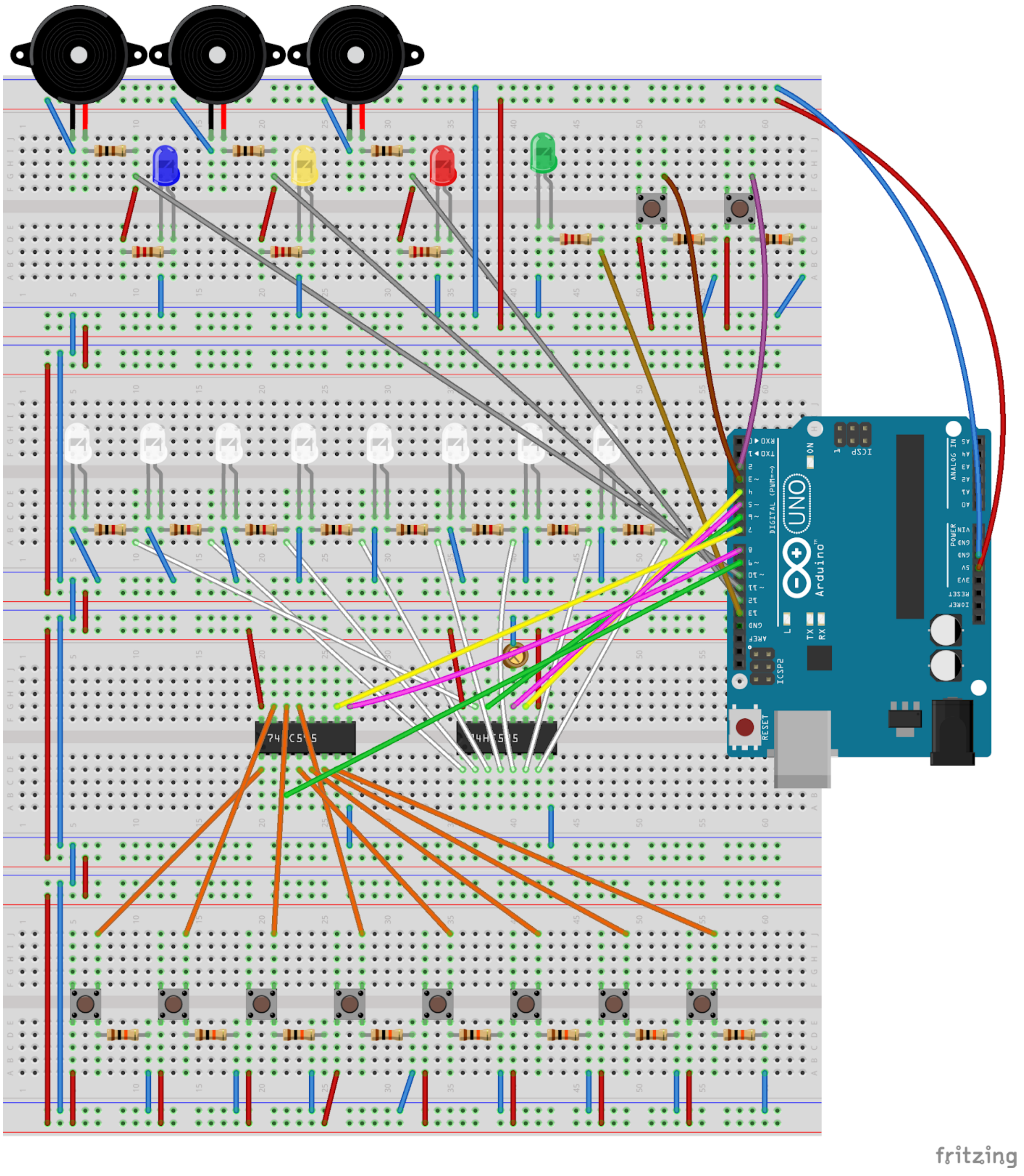
Hardware Design

The circuit has the following components: ten buttons, three buzzers, eight white LEDs, one red LED, one yellow LED, one blue LED, one green LED, ten 10K ohm resistors, eight 1K resistors, four 220 ohm resistors, three 100 ohm resistors, one 0.1 uf ceramic capacitor, one CD4021BE shift in register, one 74HC595 shift out register, one Arduino UNO, and wires.

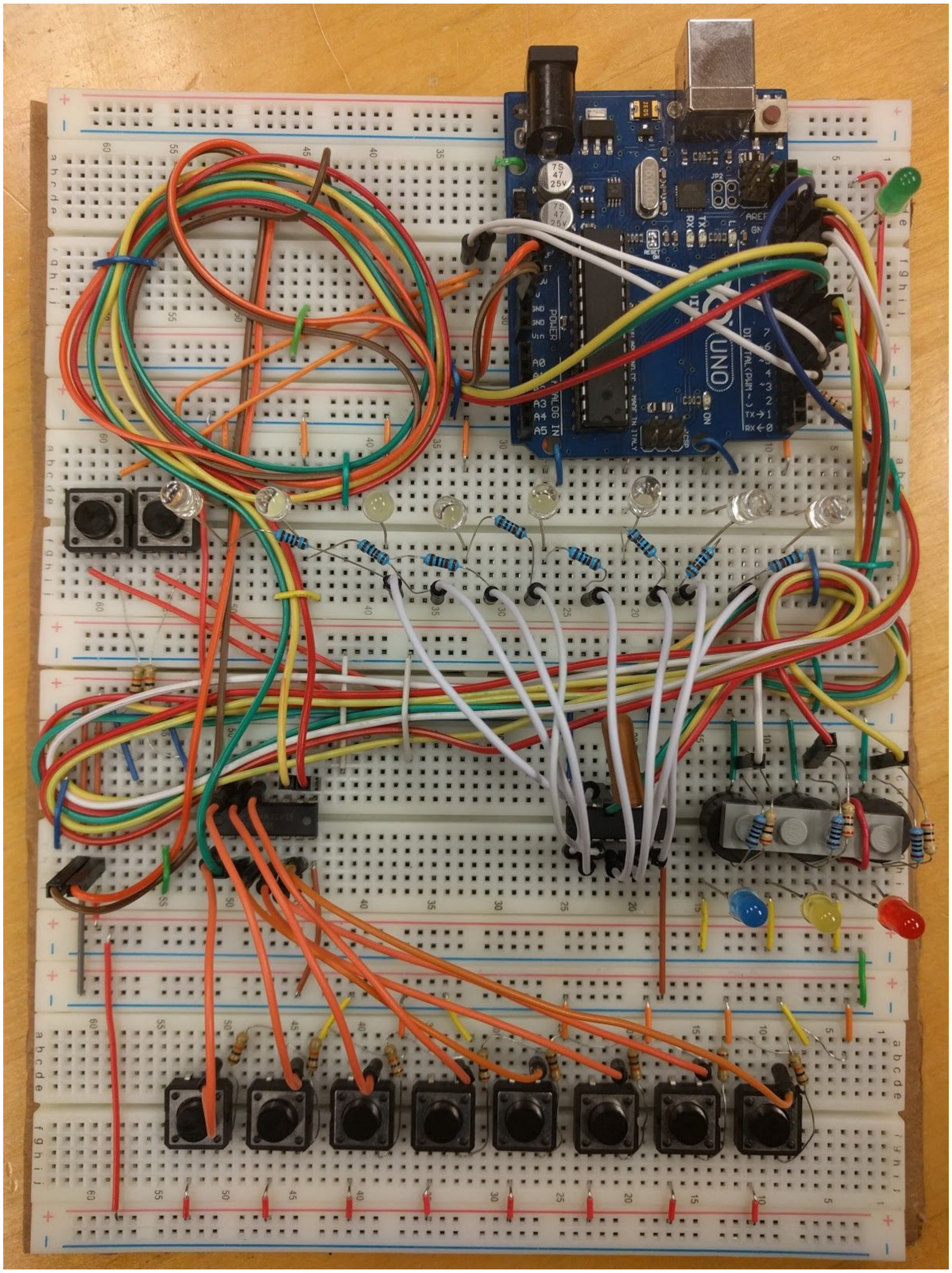
The CD4021BE shift in register takes inputs from eight buttons and transmit an 8-bit pattern to the Arduino. The 74HC595 shift out register will then take the 8-bit pattern as input to give appropriate outputs to LEDs. The three buzzers are directly connected to the Arduino. When only one note is pressed, the buzzer with red LED will play the note. When two keys are pressed simultaneously, buzzers with red and yellow LEDs will play the notes. When three keys are pressed, all buzzers will play the notes, and all LEDs light up. If a fourth key is pressed, the first three keys in MSB order will be played. There are two buttons for the music player mode, and both are directly connected to the Arduino. If the mode switch button is pressed, a green LED will light up, and all buttons connected to the CD4021BE chip will be locked. The other button is the play button of the music player. Both music player buttons take 1 press to change state. For example, if the mode switch is not pressed, the piano will not be in music player mode, and the play button will not be working.

The diagram of the circuit is shown below. Note that in the diagram there are two 74HC595 chips. Since there is no image for CD4021BE chip, the 74HC595 on the left is used as a placeholder, but the wiring is the same as the actual CD4021BE chip.

In the diagram, red wires are connected to +5V, blue wires are connected to the ground. Orange wires transmit inputs from buttons to the shift in register. White wires transmit outputs to the white LEDs. Grey wires connect buzzers to the Arduino. For both shift registers, yellow wires connect to clock pins, green wires connect to data pins, and pink wire connect to latch pins. The two buttons in the top right corner of the breadboard are music player buttons. The 0.1 uf capacitor is located above the right side shift register.


74HC595
74HC595
UNO
Arduino
RESET
ICSP
ICSP2
DIGITAL (PWM=~)
ANALOG IN
POWER
fritzing

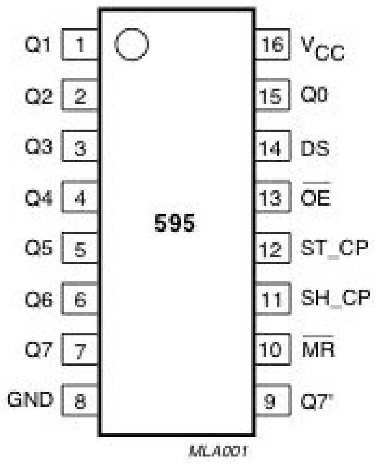
Hardware Implementation

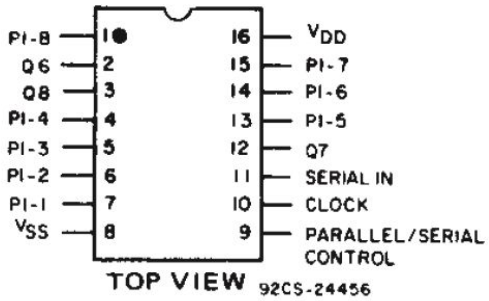


The image above is the actual circuit of the Arduino piano, which is almost the same as the diagram. In this circuit, the Arduino UNO is tied to the breadboard, the top left corner is used for cable management, the buzzers are next to the 74HC595 chip, the music player buttons are to the left of white LEDs, and the mode switch LED is located in the top right corner. In addition, the left music player button is the play button, and the right one is the mode switch. There is no strict color coding for wires, but latch pins are red, clock pins are yellow, and data pins are green. The buzzer LEDs share power with the buzzers. In this way, if the buzzers are on, the LEDs will also be on, and the program does not need to control them. The LEGO block is used to change the timbre of the buzzers. Extra short wires are used to tie the long wires.

The circuit is in piano mode by default, and you can play up to three notes at the same time. To enter music player mode, press the first button to the left of white LEDs, and the green LED will be turned on. To play the song, press the leftmost button. The song can be paused by pressing the leftmost button again. To exit music player mode, press the first button to the left of white LEDs again, and the green LED will be turned off. The song progress will be reset after exiting music player mode. The length of a pre-loaded song is restricted because the Arduino UNO only has 2 Kilobytes of dynamic memory.

The two images below are details of the 74HC595 chip and the CD4021BE chip. For the 74HC595 chip, pin 1-7 and 15 are used as outputs, pin 11 is the clock pin, pin 12 is the latch pin, and pin 14 is the data pin. For the CD4021BE chip, pin 1, pin 4-7, and pin 13-15 are used as inputs, pin 9 is the latch pin, pin 10 is the clock pin, and pin 2 is the data pin. The 0.1 uf capacitor is used to debounce the latch pin (pin 12) of the 74HC595 chip. Pin 10 of the 74HC595 chip (master reclear) also need to be connected to +5V for the chip to work properly.





TERMINAL DIAGRAM
CD4014B, CD4021B

Software Design

The program needs to read inputs from the CD4021BE chip to decide what buttons are pressed. The inputs will form an 8-bit pattern, and each bit represents one button. A button press will turn the corresponding bit to 1, and turn back to 0 on release. The binary pattern is directly sent to the 74HC595 chip to manipulate the white LEDs. The eight buttons are given keys from C4 to C5, and a loop will be used to record the first three bits that is 1. In each iteration, if the bit is 1, the index will be recorded in an array. Since there are at most three items in this array, each item is assigned to a buzzer. The “Tone.h” library will be used to control three buzzers at the same time, which allows us to play three different notes simultaneously. The music player mode will play a song by reading note arrays from a header file. In the header file, there are three note arrays and three duration arrays. Each buzzer will use one note array and one duration array. Besides arrays, there are variables for the song’s number of measures, time signature, beats per minute, and the precision (divide a quarter note by this value). All the arrays should have the same length, which can be calculated by multiplying measure, time signature, and precision. The music player buttons are programmed for state change detection, and it takes one key press to change state. In each program cycle, the mode switch reading will first decide the piano's mode. If the music player mode is not on, the program will execute the code for piano buttons, including shift in, shift out, and playing notes. If the music player mode is on, the code for piano keys will be ignored. In this mode, the play button reading will decide whether the preloaded song should be played. In addition, in each program cycle, only one set of notes (three notes at the same time) is played for both piano mode and music player mode. Outside the loop, there should be a counter to record progress of the song.

Software Implementation

```
/* main program code */
#include <Tone.h>            // allows control of three buzzers at the same time
#include "acat.h"            // preloaded song, portion of "A Cruel Angel's Thesis"

#define Buzz1 12
#define Buzz2 11
#define Buzz3 10
#define clockPin 9           // shift in register
#define latchPin 8           // shift in register
#define dataPin 7            // shift in register
#define clockPinL 6          // shift out register, L stands for LED
#define latchPinL 5          // shift out register
#define dataPinL 4           // shift out register
#define playButton 3
#define modeSwitch 2
#define LED 13               // signal LED for mode switch

Tone tone1;                  // tone objects from "Tone.h"
Tone tone2;
Tone tone3;
int mode = 0;                // mode flag
int playButtonCounter = 1;   // playButtonCounter % 2 == 0
int modeSwitchCounter = 1;   // so initial value cannot be 0
int playButtonState = 0;
int modeSwitchState = 0;
int lastPlay = 0;
int lastMode = 0;
int arrayLen = 0;               // music array length
int noteLength = 0;             // single note length
long noteIndex = 0;             // progress of a song
byte pattern = 72;              // 1001000, default pattern

// assign notes to each piano key
const int keys[] = {
        NOTE_C5,NOTE_B4,NOTE_A4,NOTE_G4,NOTE_F4,NOTE_E4,NOTE_D4,NOTE_C4};


void setup(void){
        // shift in register
        pinMode(latchPin, OUTPUT);
        pinMode(clockPin, OUTPUT);
        pinMode(dataPin, INPUT);
        // shift out register
        pinMode(latchPinL, OUTPUT);
        pinMode(clockPinL, OUTPUT);
        pinMode(dataPinL, OUTPUT);

        pinMode(playButton, INPUT);
        pinMode(modeSwitch, INPUT);
        pinMode(LED, OUTPUT);
        tone1.begin(Buzz1);
        tone2.begin(Buzz2);
        tone3.begin(Buzz3);
}

void loop(){
        // state change detection
        modeSwitchState = digitalRead(modeSwitch);
        if (modeSwitchState != lastMode){
                if (modeSwitchState == HIGH){
                        // increase if button is pressed and released
                        // holding button will not increase this value
                        modeSwitchCounter++;
                }
        }
        lastMode = modeSwitchState;          // update lastMode for next iteration
        // 1 key press and release to change state
        if (modeSwitchCounter % 2 == 0){
                mode = 1;                    // mode switch pressed, enter music player mode
                digitalWrite(LED, HIGH);     // turn green LED on
                // measures, timeSig, and precision come from the header file
                arrayLen = measures * timeSig * precision;  // calculate music and note length
                noteLength = (60000/(bpm*precision));       // 1 minute has 60000 milliseconds
        }
```

```
	else{
		mode = 0;			// do not enter music player mode
		digitalWrite(LED, LOW);
		noteIndex = 0;			// resets song progress
		lastPlay = 0;			// as well as play button states
		playButtonState = 0;
		playButtonCounter = 1;
	}
	lastPlay = playButtonState;
	if (mode){
		// state change detection for play button
		// if music player mode is not on
		// play button is locked
		playButtonState = digitalRead(playButton);
		if (playButtonState != lastPlay){
			if (playButtonState == HIGH){
				playButtonCounter++;
			}
		}
		lastPlay = playButtonState;
		if (playButtonCounter % 2 == 0){
			// if music is not over
			if (noteIndex < arrayLen){
				musicPlayer(Sa,Sb,Sc,Pa,Pb,Pc);
				noteIndex++;
				myDelay(noteLength);		// delay to play the whole note
			}
		}
	 }
	else{
		// stays in piano mode
		digitalWrite(latchPin,1);	// set latch pin to 1 to collect parallel data
		delayMicroseconds(20);	// debounce
		digitalWrite(latchPin,0);	// set it to 0 to transmit data serially
		pattern = shiftIn(dataPin, clockPin);	// collect each bit into a byte in MSB order
		digitalWrite(latchPinL,0);	// set latch pin to 0 so the LEDs don't change
		shiftOut(dataPinL, clockPinL, LSBFIRST, pattern); // shift out bits in LSB order
		digitalWrite(latchPinL,1);	// set latch pin to 1 to turn on LEDs
		int pressed[] = {0, 0, 0};
		checkKey(pattern, pressed, keys);	// record up to three pressed key
		playNotes(pressed);		// play the recorded keys
	}
}

/*
 * Purpose: check what keys are played and record up to three keys
 * Parameters:
 *		byte pattern: bit representation of keys
 *			int pressed[]: array of pressed keys
 *			const int keys[]: assigned notes
 * Return:
 *			None
 */
void checkKey(byte pattern, int pressed[], const int keys[])
{
	int i = 0;
	int j = 0;
	// while bit checking not finished
	// and recorded keys are not full
	while ((i<8) && (j<3)){
		if (pattern & (1 << i)){
			pressed[j] = keys[i];		// use assigned keys as reference
			j++;
		}
		i++;
	}
}

/*
 * Purpose: play notes using buzzers
 * Parameters:
 *			int pressed[]: notes to be played
 * Return:
 *			None
 */
void playNotes(int pressed[])
{
	// check each position in the array
	// and play if there is a value
	if (pressed[0]){
		tone1.play(pressed[0]);
	}
	else{
		tone1.stop();
	}
	if (pressed[1]){
		tone2.play(pressed[1]);
	}
	else{
		tone2.stop();
	}
	if (pressed[2]){
		tone3.play(pressed[2]);
	}
	else{
		tone3.stop();
```

```
        }
}

/*
 * Purpose: play a preloaded song
 * Parameters:
 *              int notes1[]: track 1, played by buzzer 1, main melody recommended
 *              int notes2[]: track 2, played by buzzer 2, harmony or bass recommended
 *              int notes3[]: track 3, played by buzzer 3, percussion recommended
 *              int8_t pause1[]: note length data for track 1, use signed byte to save space
 *              int8_t pause2[]: note length data for track 2
 *              int8_t pause3[]: note length data for track 3
 * Return:
 *              None
 */
void musicPlayer(int notes1[], int notes2[], int notes3[], int8_t pause1[], int8_t pause2[], int8_t
pause3[])
{
   if (notes1[noteIndex]){
              // the play function takes frequency and duration
              // adding 10 milliseconds to noteLength can connect two notes
              // adding -10 milliseconds can disconnect two notes clearly
              tone1.play(notes1[noteIndex], noteLength+pause1[noteIndex]);
   }
   if (notes2[noteIndex]){
              tone2.play(notes2[noteIndex], noteLength+pause2[noteIndex]);
   }
   if (notes3[noteIndex]){
              tone3.play(notes3[noteIndex], noteLength+pause3[noteIndex]);
   }
}

/*
 * Purpose: shift in input bits to form a byte, replace default shiftin function
 * Parameters:
 *              int DataPin: data pin number
 *              int ClockPin: clock pin number
 * Return:
 *              byte representation of input
 */
byte shiftIn(int DataPin, int ClockPin){
        int one = 0;                                        // data pin value
        byte pattern = 0;                                   // return value
        // in MSB order
        for (int i= 7; i>=0; i--){
              // set clock pin to 0
              // to change data pin state
              digitalWrite(ClockPin, 0);
```

```
                delayMicroseconds(0.2);
                one = digitalRead(DataPin);
                if (one){
                        // if data pin has value
                        // add the bit to pattern
                        pattern = pattern | (1 << i);
                }
                digitalWrite(ClockPin, 1);
                // high to low drop of the clock pin
                // will cause the data pin to change state
                // based on the value of the next bit in serial flow.
        }
        return pattern ;
}

/*
 * Purpose: delay for given milliseconds, replace default delay function
 * Parameters:
 *              int ms: duration of delay in milliseconds
 * Return:
 *              None
 */
void myDelay(int ms)
{
        for (int i = 0; i < ms; i++){
                delayMicroseconds(1000);
        }
}

/* end of main program, start of header file example */
// NOTE_C4 and other notes are integers defined in "Tone.h"
int Sa [] = {NOTE_C4, NOTE_D4}  // song A, track 1 notes
int Sb [] = {NOTE_F4, NOTE_G4}  // song B, track 2 notes
int Sc [] = {NOTE_C3, NOTE_C3}  // song C, track 3 notes
int8_t Pa [] = {-10,-10}        // pause A, track 1 note duration
int8_t Pb [] = {-10,-10}        // pause B, track 2 note duration
int8_t Pc [] = {10,-10}         // pause C, track 3 note duration
int measures = 1;               // total measures of the song
int timeSig = 2;                // time signature, number of quarter notes each measure
int bpm = 60;                   // beats per minute, for speed control
int precision = 1;              // how many notes in the array represent a quarter note
```

The musicPlayer() function and the header file need more explanation. The key idea is to use the main loop of the Arduino to iterate through an array of frequency values, so that the tone object can read and play the note. In order to play a three-track song (melody, bass, and percussion), the arrays for each track need to be synced with the biggest common divisor note. For example, if the shortest note in a song is the quarter note, each item in the arrays should be treated as a quarter note. To play a half note in this configuration, two quarter notes are needed, and we also need two iterations of the main loop. If the shortest note is sixteenth note, we will need eight of them and eight iterations to play a half note.

The tone object has a "play()" method that takes frequency and duration as arguments. To play a note with a given length (500 milliseconds for example), we not only need to call "tone.play(frequency, 500)", but also need to call a delay function to delay for 500 milliseconds, so that the note is fully played before the next iteration. By adding 10 milliseconds to the duration parameter and keep delay time the same, we are able to connect two consecutive notes in the array (same frequency), because the buzzer gets 10 more milliseconds to play in the next iteration. By adding or subtracting 10 milliseconds from the duration parameter, we can play group notes in the array together to play long notes, or cut each note's length for staccato.

The musicPlayer() function takes 3 arrays of notes and 3 arrays of duration offsets (10, -10, and 0) as arguments, and all the arrays are defined in the header file. All the arrays have the same length. There are three additional variables: measures, timeSig, and precision. The type of note in the array determined by precision. For example, precision 4 means we need four iterations to construct a quarter note, and therefore the shortest note is the sixteenth note. To calculate the duration in milliseconds of the shortest note, we use the formula:

$$note\ length\ = milliseconds\ per\ minute \div (bpm \cdot precision)$$

To calculate the length of each array, we use the formula:

$$\textit{array length} = \textit{measures} \cdot \textit{time signature} \cdot \textit{precision}$$

For example, if a song has 40 measures, each measure has 3 quarter notes, and the shortest note is the thirty-second note (precision 8), the array length will be 960, and we will have six arrays of the same length. In practice, the “acat.h” contains only 12 measures of music, but the array length is 192, and 96% of dynamic memory of the Arduino UNO is used after compiling. To save space, the type “int8_t” is used for note duration offsets, because this data type can represent integer from -128 to 127. In order to play longer music with the Arduino, we will need a better algorithm that can dynamically allocate note length.

Other

The preloaded song for this project is specifically arranged for the Arduino. As the first step, we need to get the sheet music or midi file of the song we want to play. If the song is a solo without any background instruments, we can turn the melody into one array of frequencies easily. However, most songs will have more than three tracks, so the second step is to rearrange the music into a three-track version. A midi editor will get the job done, but we need to keep the balance between the quality of music and the amount of notes when shrinking tracks. The third step is to turn the three-track version of the song into C++ arrays, which takes a lot of time to do by hand (4 measures per hour). We can also write a program to read the midi file and generate the header file directly, but since this project will not be graded on this program, it is not worth the time to do so. Therefore, we come up with a quick and dirty solution: use a python script to read user inputs and generate array strings. For example, in each session of the script, we only need to type the name of the note (C4 for example) and the length of the note, and then the script will write the formatted arrays of notes and duration offsets into a header file. The following function demonstrates how a single note is formatted in to array strings.

```
def makeNote(note, length):
        noteString = note + ",”
        offsets = "-10,"
        filler = "10,"
        if note == "0":              # if there is no note
                filler = "0,"
                offsets = "0,"
        for i in range(length-1):
                noteString = note + "," + noteString
                offsets = filler + offsets
        return noteString, offsets
```

Although we constructed the array representation of “A Cruel Angel’s Thesis” (1 minute and 30 seconds, 49 measures) with this script, only the last 12 measures are used due to memory limitation of the Arduino UNO.

Conclusion

The most important parts of this project are the 74HC595 chip, the CD4021BE chip, and the “Tone.h” library. Using bit patterns to give inputs and outputs is unconventional considering the projects we have done throughout the semester, but in practice it is more efficient to use these two chips than using digital pins. If I can do this project again, touch sensors will be a better option than buttons for representing piano keys (and they are more expensive). A better algorithm for the music player is also needed to longer songs while using less SRAM. In addition, a separate program that can generate C++ array representation of a midi file would be useful for future Arduino music projects. At the end, the piano circuit works smoothly even if we abuse the buttons.