Music Memory

Delainey Ackerman

March 2017

1 Abstract

Music Memory is a make your own music box consisting of three preset melodies and three custom tracks for the user to input, record, and play music. Using components and an Arduino Uno micro-controller from the Arduino Starter Kit, Music Memory is designed so users can select, play, stop any song, and create custom tracks with up to 33 notes per track using a 7 note keyboard. Music Memory will maintain note order and note duration, storing the tune in persistent memory for playback across sessions.

Figure 1: Music Memory box for song recording and playback

2 Design

The initial concept arose from a personal desire to play the Jurassic Park theme song at any time. This developed into the more interesting idea of allowing user input to create, store, and play custom melodies. The primary inputs are a potentiometer which allows the user to select one of six melodies to play through 30° segments of the pot's range of motion, and a seven note piano-style keyboard.

The initial design created the piano keys with photoresistors as a touch input, however inconsistent input and a lack of analog input pins on the Arduino forced an alternative design. I explored using capacitive touch sensors, but ultimately decided to limit the number of required input pins on the Arduino by using a resistor ladder circuit design with push buttons.

3 Planning & Materials

Using ny design concept, I separated Music Memory into its core functionality and created a plan to prototype these functions progressing from simple to complex both in terms of circuit design and associated code.

The three main parts in order of simplicity are playing a melody, choosing a melody, and recording a melody. Additional planning involved researching iconic melodies, relearning sheet music reading, and testing which melodies came through the piezo well only playing one note at a time.

3.1 Materials

Arduino Kit Includes:

- Arduino Uno
- LCD
- Multi-Color LED
- Piezo
- Potentiometer x 2
- Push Button x 9
- Resistors
  - $220\Omega\ x\ 4$
  - $560\Omega\ x\ 1$
  - $4.7k\Omega\ x\ 1$
  - $1k\Omega\ x\ 1$
  - $10k\Omega\ x\ 4$
  - $1M\Omega\ x\ 1$
- Wiring

Additional Materials Required:

- 9V Battery
- M2 x 15mm screws & nuts
- Strip board (7 x 5cm) x 4
- Toggle Switch

4 Prototyping

Using the Arduino Starter Kit components and breadboard I tested the design and implementation of each aspect of Music Memory.

4.1 Playing a Melody

The first aspect of Music Memory is the ability to play a melody. This involves storing a playable representation of the song and sending frequencies through a piezo speaker at specific intervals. To test this functionality I used a push button to initialize play, an LED to light up during play, and a piezo speaker to play musical notes.

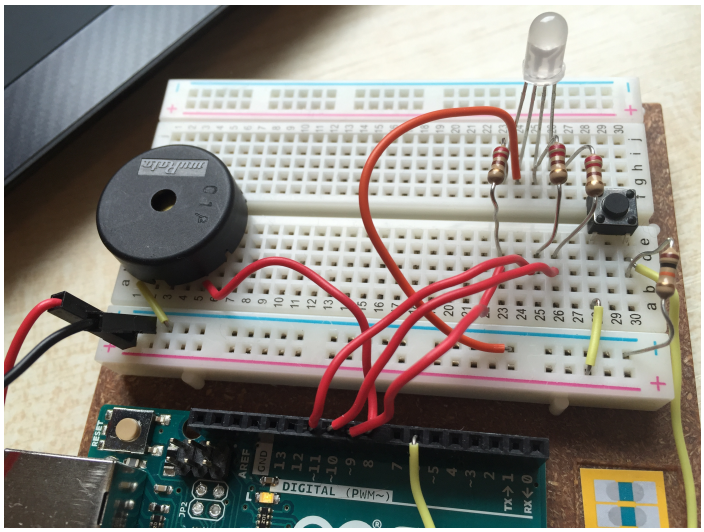

Figure 2: Push button to turn on green LED and play stored melody. Light turns off and sound stops after last note is played

I chose to represent a melody as a pair of arrays where the first array contains note frequencies and the second array holds note durations. A single incrementing index could then be used to access a pair of associated frequencies and durations and send each frequency in order to the piezo speaker at the corresponding intervals.

```
// Frequencies for associated notes are stored as const ints
// ex. const int A = 220;
int notes[] = {B, B, A, B, B, A, B, C, C, E, E, D, B, C, A, 175,
    D, B, C, F, B, E, D, D, C, C};
int noteDurations[] = {12, 4, 4, 12, 4, 4, 8, 4, 8, 4, 12, 4, 4, 8, 4, 8,
    4, 4, 12, 4, 4, 8, 4, 8, 4, 12};
```

```
void loop() {
    ...
    // If the button has been pressed, play is enabled
    if (playEnabled) {
        // Turn on green LED
        greenValue = 255;
        analogWrite(greenLEDPin, greenValue);
                ...

        // For each stored note
        for (int i = 0; i < sizeof(notes) / sizeof(notes[0]) ; i++) {
            // Find note frequency and note duration and
            // send to piezo through Digital 8
            tone(8, notes[i], noteDurations[i] * 50);

            // Pause between notes, length of pause
            // determined by length of previous note
            delay(1.3 * 50 * noteDurations[i]);
            noTone(8);
        }

        // Turn play and green LED off after melody finishes playing
        playEnabled = false;
        greenValue = 0;
    }
    ...
}
```

4.2 Selecting a Melody

The next challenge of selecting a single melody to play from a list of possibilities was primarily a coding problem. The physical aspect is simply dividing the 180° range of a potentiometer into 30° segments with each segment representing one of six possible tunes.

```
void loop() {
    ...
    // Read potentiometer to determine current melody
    potVal = analogRead(potPin);

    // Constrain analog input to a 0-179 degree angle reading
    potZone = map(potVal, 0, 1023, 0, 179);

    // Assign angle to zone index 0-5
    potZone /= 30;
```

```
    . . .
}
```

This implementation was confirmed using serial print to see zones increment from 0 to 5 as the potentiometer twists across its full range. I then prototyped how to store multiple pairs of arrays and maintain the association between a single note frequency and its play duration.

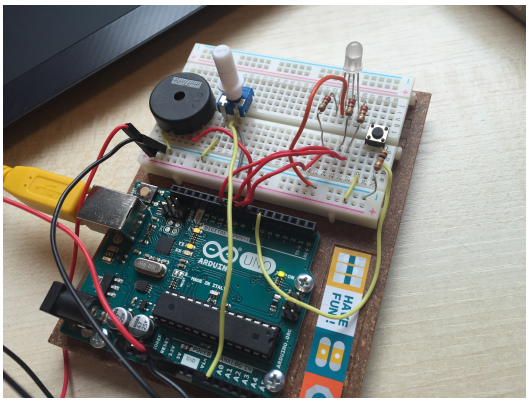

Figure 3: Potentiometer twists to select one of six zones; each zone represents a unique melody to play

I settled on using a 2D array as a database of all note frequencies and note durations. The two arrays comprising a melody are stored consecutively in the 2D array database so that database[0][0] is the first note frequency of the first tune and database[1][0] is the associated duration of the first note in the first tune.

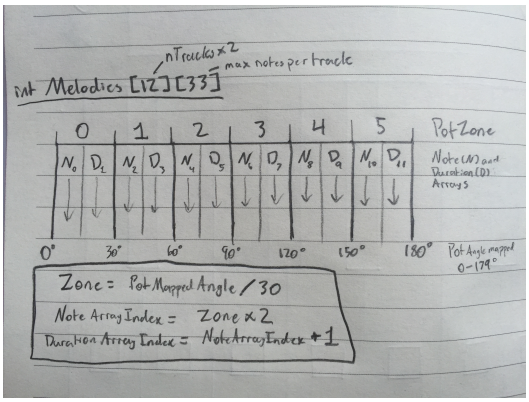

Figure 4: 2D Array of melody arrays. Each potentiometer zone covers two melody arrays, one for note frequencies and one for durations.

That solved storing different melodies, the next challenge was selecting and playing a tune matching the current potentiometer zone. I used two arrays to hold the current tune's frequencies and durations, and then each time the zone changed, read from the 2D array database at the appropriate zone and overwrite the two current arrays with the new zone's database tune.

```
// 2D array of 6 pairs of note frequency and duration arrays
int melodies[12][MAX_NOTES] = {
    // Jurassic Park
    { B, B, A, B, B, A, B, C, C, E, E, D, B, C, A, 175, D,
        B, C, F, B, E, D, D, C, C, 0, 0, 0, 0, 0, 0, 0 },
        { 12, 4, 4, 12, 4, 4, 8, 4, 8, 4, 12, 4, 4, 8, 4, 8, 4,
            4, 12, 4, 4, 8, 4, 8, 4, 12, 0, 0, 0, 0, 0, 0, 0 },

    // Lord of the Rings: The Shire
    { C, D, E, G, E, D, C, E, G, 440, 523, 493, G, E, F, E,
        D, C, D, E, G, E, D, E, D, C, E, G, 440, 440, G, E, D },
        { 4, 4, 8, 8, 8, 4, 12, 4, 4, 8, 4, 4, 4, 8, 4, 4,
            8, 4, 4, 8, 4, 4, 8, 2, 2, 12, 4, 4, 8, 4, 4, 4, 16 },
    ...
}

// Current zone note frequency and durations
int notes[MAX_NOTES];
int noteDurations[MAX_NOTES];

void loop() {
    ...
    // Assign angle to zone index 0-5
    potZone /= 30;
    // Each zone has indices for the
    // note frequency and duration arrays
    noteIndex = potZone * 2;
    durationIndex = noteIndex + 1;

    // Load note frequency arrays and durations from
    // current zone into the playable track array
    for (int i = 0; i < MAX_NOTES; i++) {
        notes[i] = melodies[noteIndex][i];
        noteDurations[i] = melodies[durationIndex][i];
    }
    ...
}
```

With the representation and access to multiple melodies implemented, the last piece of selecting a melody was hooking up an LCD to visualize for the user which track is selected. The screen uses a simple switch case statement based

on the pot zone index 0 - 5 to print appropriate text.

```
void loop() {
    ...
    switch (potZone)
    {
    case 0:
        lcd.print("Jurassic Park");
        break;
        ...
    case 2:
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Play Song: How");
        lcd.setCursor(0, 1);
        lcd.print("to Save a Life");
        break;
        ...
    }
}
```

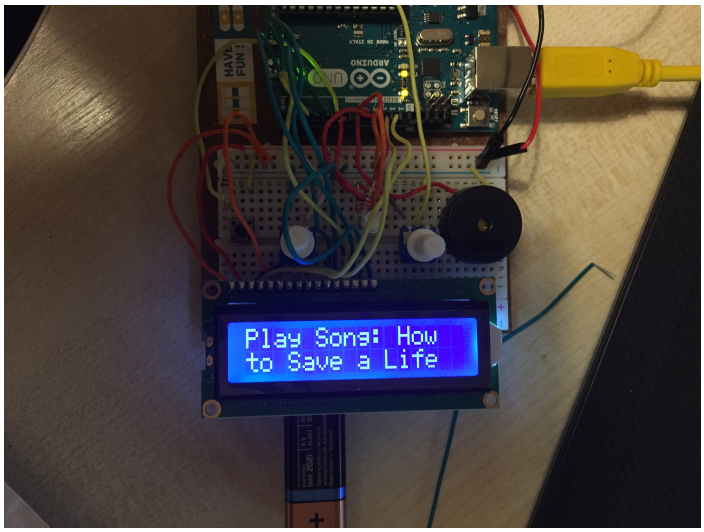

Figure 5: Potentiometer currently set to zone 2, will play preset song from The Fray

4.3 Recording a Melody

The previous prototypes provided the infrastructure enabling the implementation of the core functionality of Music Memory, the ability to store and play

custom tunes. The next functionality requires user input to create and record custom tunes. In addition to the physical components to allow user input across a range of frequencies, the Arduino has to track and store to the 2D database array of melodies the input in terms of number of inputs, order of input frequencies, and the duration the user pressed each frequency.

After testing using a photoresistor as a touch sensor, I wasn't satisfied with the quality and accuracy of the sensor to produce a consistent, smooth note. Instead, I prototyped a seven step resistor ladder using push buttons which provided a much better result and the additional benefit of using only a single input pin on the Arduino. With the keyboard integrated into the prototype, each of the seven buttons could be pressed and when in a custom track zone (zones 3-5) the associated musical frequency plays through the piezo.

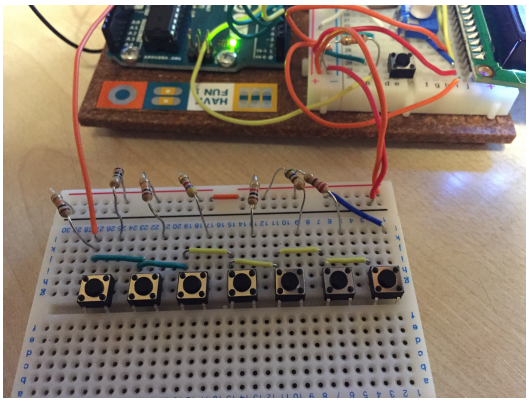

Figure 6: Resistor ladder for piano-style user input recording custom melodies

The next step was enabling the Arduino to store each key press and the length for which it was pressed. I decided to store this information by directly overwriting any previous data stored in the 2D database array at the current custom zone.

I used a boolean to ensure that each button press was stored as a single note input rather than a note being added each time the loop function iterated while the button was pressed. The first loop iteration where a key is pressed, the boolean is flipped to indicated the key is currently down, the current time is stored, and the note frequency is added in the appropriate zone and note index. Until the the key is released, the Arduino simply increments the difference between the current time and the time the note started playing.

```
// Record note in custom track and increment
// timer determining note duration
void recordNote(int note) {
    // On key down
    if (!isKeyDown) {
        isKeyDown = true;
```

```
        // store note start time
        lastNoteMillis = millis();

        // Save note
        melodies[noteIndex][recordNoteIndex] = note;
    }
    // While key is pressed
    else {
        // increment by difference between last loop and now
        currNoteMillis += millis() - lastNoteMillis;
        // store current time
        lastNoteMillis = millis();
    }
}
```

Then when the key is released, the note duration is stored in the paired array, the note index is incremented, and duration counter is reset. As the new notes are stored directly in the 2D database array, the custom track is loaded and can be played whenever the user enters the recorded zone.

```
// On key up
if (isKeyDown) {
    // Store note duration as time difference
    // between note start and end
    // scaled by a constant modifier
    melodies[durationIndex][recordNoteIndex] =
        currNoteMillis / MILLIS_TO_NOTE_DUR;
    // Increment index for next note to record
    recordNoteIndex++;

    // reset timer and boolean for key press
    currNoteMillis = 0;
    isKeyDown = false;
}
```

With the ability to record and playback custom notes, I added a record on/off button and set the LED to blink red while record mode is on for a more user friendly design. Similar to the timer used to store note duration, the blinking light is triggered by a timer and set interval so as not to prevent the Arduino from properly recording notes if the blink was created through the delay function.

```
// If record button has been pressed
if (recordEnabled) {
    // Store start time
    unsigned long currMillis = millis();

    // Blink light while recording
```

```
    if (currMillis - prevMillis >= blinkInterval) {
        prevMillis = currMillis;    // Increment timer

        // Turn on light
        if (redValue != 255) {
            // set light to record color
            redValue = 255;

            // Turn on record light
            analogWrite(redLEDPin, redValue);
            analogWrite(greenLEDPin, greenValue);
        }
        // Turn off light
        else {
            // reset light color
            redValue = 0;

            // Turn off record light
            analogWrite(redLEDPin, redValue);
            analogWrite(greenLEDPin, greenValue);
        }
    }
    ...
}
```

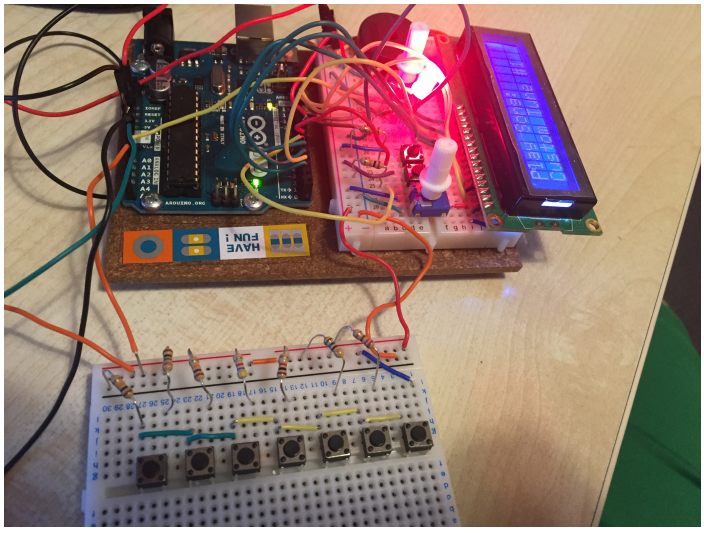

Figure 7: User pressed record button to enable custom input, LED blinks red while recording mode is active

4.4 Persistent Memory

With the major prototypes implemented, I moved on to less foundational aspects of the design and focused on improving the design from the perspective of usability. One area of user frustration was losing a custom recording each time the Arduino was unplugged or reset.

In order to elevate Music Memory beyond a short-term echo box, I implemented EEPROM persistent memory to store and load custom tracks when the Arduino is unplugged and restarted. For additional usability, I also set up the Arduino to receive power from a 9V battery through a toggle switch to eliminate the reliance on a USB cable to a computer.

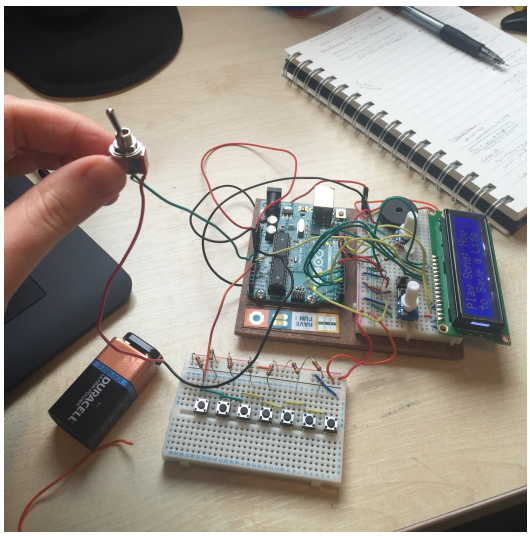

Figure 8: Toggle switch providing an ON/OFF for Music Memory through a 9V battery

To implement EEPROM memory I defined a structure to store a pair of arrays defining the melody and an integer indicating which potentiometer zone the melody belongs in.

```
typedef struct {
    // zone for corresponding pot position,
    // custom tracks are zones 3-5
    int trackZone;

    // array of frequencies played for track
    int notes[MAX_NOTES];
    // array of durations for each stored frequency
    int noteDurs[MAX_NOTES];
} CustomTrack;
```

Knowing that Music Memory permits the final three zones to create and store custom tracks, I could store three CustomTrack structs in EEPROM memory accessing the zone 3 track at EEPROM address 0, with the next two tracks incrementing address size in memory by the size of the CustomTrack struct.

$$AddressInEEPROM = (TrackZone - 3) * sizeof(CusstomTrack\ struct)$$

Custom tracks are stored into EEPROM memory when record mode is switched off, and the custom tracks are loaded into the 2D database array from EEPROM in startup with the Arduino is powered on.

```
if (onOffRecordSwitchState != previousOnOffRecordSwitchState) {
    if (onOffRecordSwitchState == HIGH) {
        ...
        // Save track to persistent memory on finish
        else {
            // Initialize struct to store track
            CustomTrack newTrack{
                potZone,
                0,
                0
            };

            // Use notes just recorded to fill struct arrays
            for (int i = 0; i < MAX_NOTES; i++) {
                newTrack.notes[i] = notes[i];
                newTrack.noteDurs[i] = noteDurations[i];
            }

            // Store recorded track in persistent memory
            // at address corresponding with current zone
            EEPROM.put((potZone - 3) * sizeof(CustomTrack), newTrack);
        }
        ...
    }
}

...

// Load custom recorded tracks - called from startup()
void loadCustomTrackFromMemory() {
    CustomTrack track1;
    CustomTrack track2;
    CustomTrack track3;

    // Read from memory
```

```
    // Look in first CustomTrack bucket for potZone 3 track
    if (EEPROM.read(0) != 255) {
        EEPROM.get(eeAddress, track1);
    }
    // Look in first CustomTrack bucket for potZone 4 track
    if (EEPROM.read(sizeof(CustomTrack)) != 255) {
        EEPROM.get(sizeof(CustomTrack), track2);
    }
    // Look in first CustomTrack bucket for potZone 5 track
    if (EEPROM.read(2 * sizeof(CustomTrack)) != 255) {
        EEPROM.get(2 * sizeof(CustomTrack), track3);
    }

    // If CustomTrack was found in persistent memory,
    // load into playable 2D array containing all tracks
    for (int i = 0; i < MAX_NOTES; i++) {
        // PotZone 3
        if (EEPROM.read(0) != 255) {
            melodies[track1.trackZone * 2][i] = track1.notes[i];
            melodies[track1.trackZone * 2 + 1][i] = track1.noteDurs[i];
        }

        // PotZone 4
        if (EEPROM.read(sizeof(CustomTrack)) != 255) {
            melodies[track2.trackZone * 2][i] = track2.notes[i];
            melodies[track2.trackZone * 2 + 1][i] = track2.noteDurs[i];
        }

        // PotZone 5
        if (EEPROM.read(2 * sizeof(CustomTrack)) != 255) {
            melodies[track3.trackZone * 2][i] = track3.notes[i];
            melodies[track3.trackZone * 2 + 1][i] = track3.noteDurs[i];
        }
    }
}
```

4.5 Circuit Design

With each aspect of Music Memory prototyped, I iterated through circuit designs to eliminate bugs. This involved evaluating each component and its connections and ensuring that no incidental connections existed to create unexpected behavior.

One of the problems I ran into was the potentiometer which selects one of six songs was accidentally connected to a pin on the LCD. This error and a few other misplaced connections resulted in corrupted characters being printed

to the screen. After I relocated the pot and checked the rest of the prototype, it was time to move to full implementation of soldering components to a final circuit designed to fit inside a laser cut wooden enclosure.

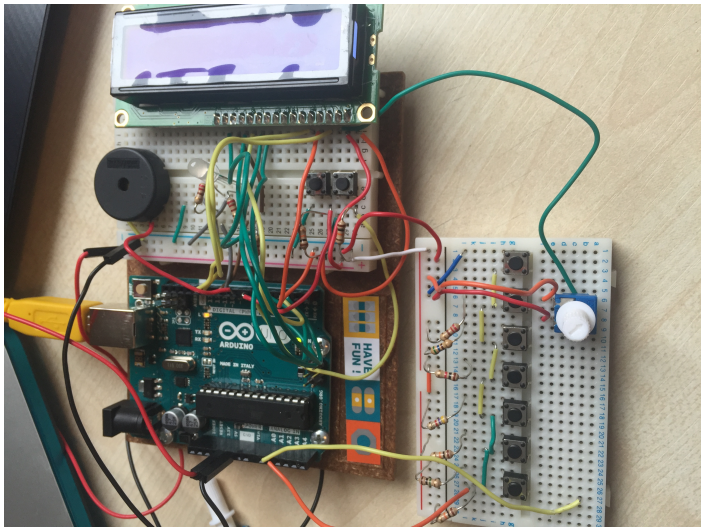

Figure 9: Testing different circuit connections to eliminate button presses and potentiometer twists from corrupting LCD display and piezo frequency output

5 Enclosure & Final Circuit

5.1 Enclosure Prototype

I used component sizes and initial concept sketches to layout and design user access to the appropriate components. After testing different layouts, I determined that 6" x 6" x 1" enclosure worked best for fitting all the components and providing enough spacing for easy usability. I sketched out this design on paper and then built a rough cardboard mock-up as proof of concept before moving on to designing the laser cut schematic.

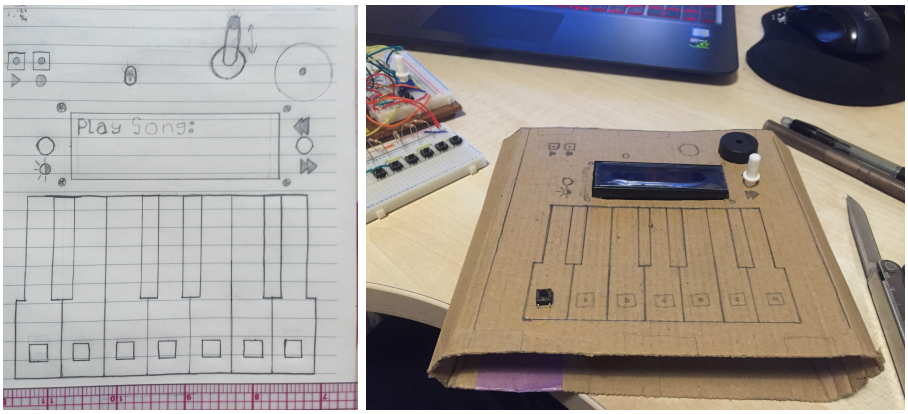

Figure 10: 1:1 scale (Left) sketch of component layout and UI engraving, (Right) cardboard prototype

5.2 Circuit

After designing the enclosure concept and layout, I began planning my final circuit design. This included how best to solder each component to a copper strip board and attach the board to the underside of the enclosure. I determined that I could use four 5cm x 7cm strip boards to get all the components properly connected to power, ground, and the Arduino.

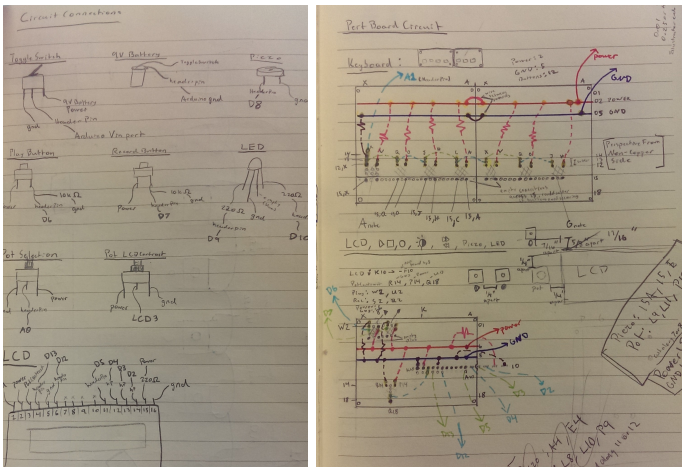

Figure 11: (Left) Component connections and (Right) circuit designs for individual strip boards

I soldered power and ground lines across the strip boards so that each component would have local access and then wires between each board would continue and complete the circuit.

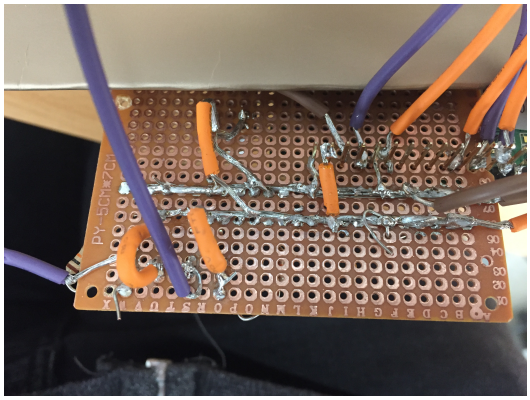

Figure 12: Power (orange) and ground (brown) solder lines running down the center of a strip board containing the LCD, potentiometer, and two push buttons

Two adjacent boards hold the 7 push buttons comprising the piano keyboard, one board contains the LCD, contrast potentiometer, and the play and record push buttons, and the final board holds the piezo and song selection potentiometer. The final two components are the multi-color LED and toggle switch which are housed directly in the enclosure with wires connecting to the appropriate circuit locations.

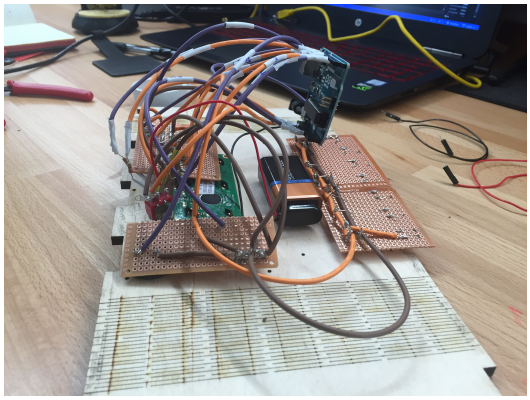

Figure 13: Complete circuit with all components soldered and connected to the Arduino

5.3 Fabrication

Using the initial enclosure design and measurements from the completed circuit, I created a laser cut design in Illustrator. After iterating the laser cut design on cardboard, the design was ready to cut on 6mm wood for the final enclosure. I also knew that each of the push buttons were too shallow in depth to properly extend through the 6mm wood. To allow full user interaction with these important components, I added small rectangular pieces to the laser cut design to create a wooden extension for each button.

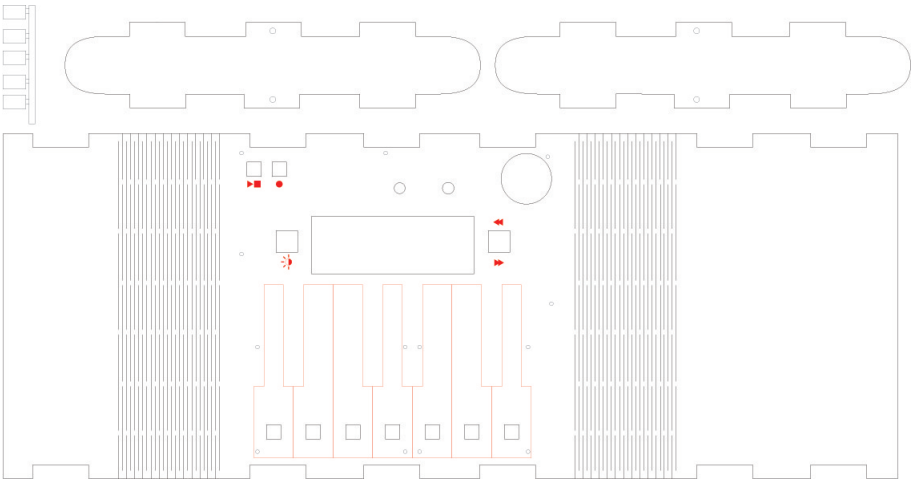

Figure 14: Illustrator design for laser cutting enclosure. Red lines indicate paths for engraving, black indicates cutting.

After cutting the wood, I secured each strip board to the enclosure by slotting each accessible component through its designed hole and using M2 screws to hold the strip board against the underside of the top of the enclosure. The main challenge in completing the enclosure was fitting everything, including the Arduino and the battery, into the short height I'd designed for the enclosure. After many iterations of redesigning the Arduino and battery placement, redoing soldering to all pins connected into the Arduino to eliminate unnecessary height, and using all available crevices to tightly pack in the wiring, I was able to finalize the interior design to allow for easy closure.

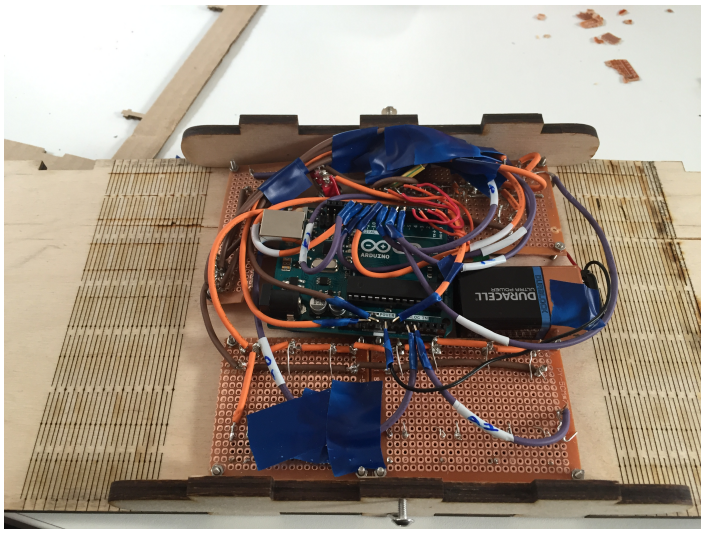

Figure 15: Final interior layout with tightly packed wiring

With each aspect of the project complete, I assembled the enclosure and slotted in the wooden extensions for each push button.

Figure 16: Music Memory fully assembled

6 Reflection

To further improve Music Memory I would increase the height of the enclosure by about 1/2" to better fit both the components and wiring, and add additional functionality to increase the musical range of user input. This second

improvement could be achieved by adding two buttons placed on either side of the existing piano keyboard which would shift the note frequencies down of up an octave.

Overall the prototyping and implementation of Music Memory successfully creates all core functionality and the aesthetic and structure of the enclosure was fully realized. Full source code available at `https://github.com/dack91/Music-Box`. Special thanks to Konstantine for help throughout the lab and using the laser cutter.

References

[1] Arduino AG. "Arduino - EEPROM." Arduino - Learning. N.p., n.d. ¡https://www.arduino.cc/en/Reference/EEPROM¿.

[2] Arduino AG. "Play a Melody Using the Tone() Function." Arduino - Tutorials - ToneMelody. N.p., n.d. https://www.arduino.cc/en/Tutorial/toneMelody.

[3] King, Joseph, and Isaac Slade. *How to Save a Life*. The Fray. Aaron Johnson, 2005. http://www.musicnotes.com/sheetmusic/mtd.asp?ppn=MN0054526.

[4] Michigan Technological University. "Frequencies of Musical Notes." Physics of Music. N.p., n.d. http://www.phy.mtu.edu/ suits/notefreqs.html.

[5] Musicnotes. "How to Read Sheet Music: Step-by-Step Instructions." Musicnotes Blog. N.p., 11 Apr. 2014. http://www.musicnotes.com/blog/2014/04/11/how-to-read-sheet-music/.

[6] Shore, Howard, orch. Cond. Howard Shore. *The Lord of the Rings The Fellowship of the Ring: The Shire*. New Line Cinema, 2001. $http://alcaeru.weebly.com/uploads/7/8/6/0/786082/fotr_complete_transcription.pdf$.

[7] Virtual Piano. "Virtual Piano." Virtual Piano. N.p., n.d. http://virtualpiano.net/.

[8] Williams, John, perf. *Theme from Jurassic Park*. Universal Music Publishing Group, 1992. http://pop-sheet-music.com/Files/aa83e57d18950f7b37be07e6ee421480.pdf.