Synth8
From Outer Space

By Josh Natis

Operating Manual For the Interested User

Hello user, thanks for your interest! Thankfully, the device (henceforth referred to as *synth8*) is not very difficult to use.

To turn it on, you'll need to procure a USB Type A Male to USB Type B Male connector -- the Type B end goes into the Arduino's port situated on the top right. You can also try your luck with a 9V AC ⅂ DC adaptor, if you'd like.

To make sound, press a button! The buttons are mapped to the standard Western 12-tone equal temperament scale (as in, the leftmost one is C, then C sharp, then D...), and are arranged somewhat in the style of a piano keyboard. By default, the notes will be from octave 5, but you can use one of the knobs (to be discussed) to change the octave. *Synth8* is polyphonic, capable of playing up to 4 voices at once. If you pass that limit (i.e. press 5 buttons at once), the note pressed longest ago will stop playing.

The lone knob on the left adjusts the volume (it gets pretty loud, so be careful). As the

volume increases, the tones will begin to clip/distort, and vice versa (i.e. at lower volume the sound will be smooth and nicely filtered). The knobs on the right serve the following functions: from left to right

(1) Change the waveform of the sound (`SINE, TRIANGLE, SQUARE, SAW, RAMP, NOISE`),
(2) Pitch-bend the sound, i.e. fine tuning (`0-100HZ`),
(3) Change the octave of the notes, i.e. coarse tuning (`octaves 1-7`),
(4) Change the modulation property of the sound (`0-127`)

Effects #1, #2, and #4 are LIVE, as in they affect the sound as you twist the knobs. Effect #3 is applied the next time you press a button. A neat trick: if you want tones from multiple octaves, simply play the tones from one octave, continue to hold the buttons down, change the octave using knob #3, then press the buttons for the other tones you want to play. Voila!

That's all there is to it, enjoy!

How It Was Made

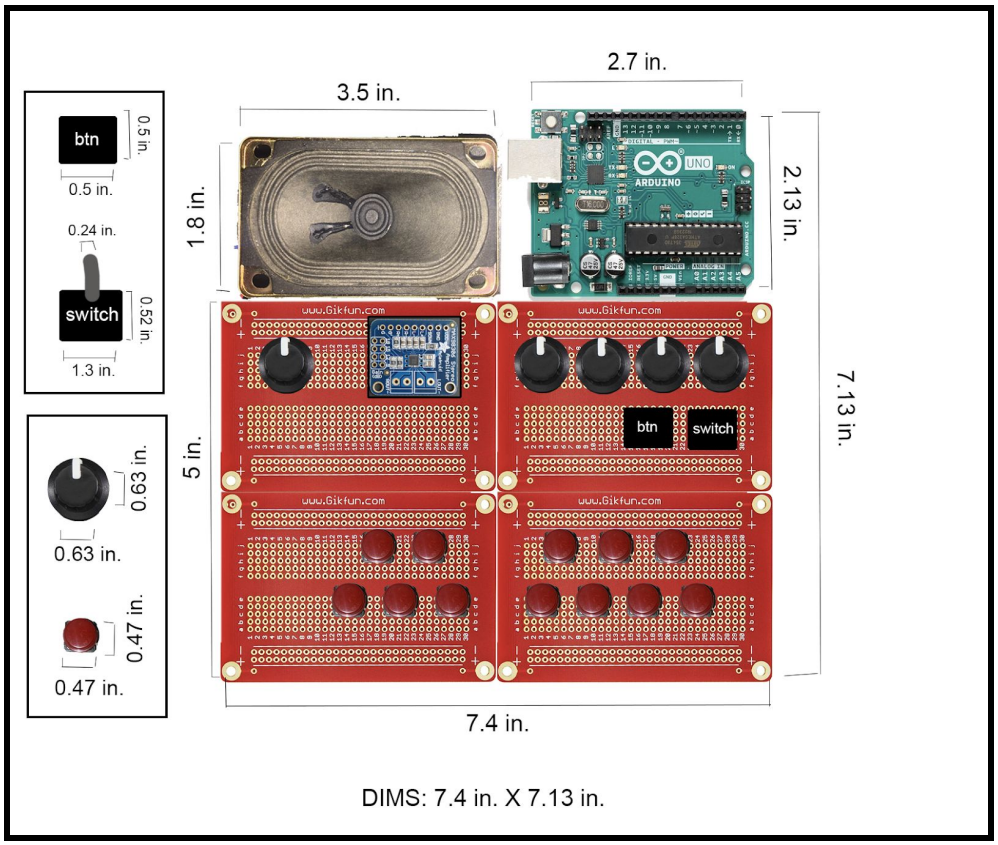
At its heart, *Synth8* is four components in correspondence with one another -- a bunch of inputs (buttons and knobs), an audio signal generating library, a speaker and amplifier, and a microcontroller loaded with a program to mediate it all.

The audio signal generating **library** was written by a man who goes by the name of Dzl TheEvilGenius. Its source code can be found here: github.com/dzlonline/the_synth. Although it is very (very) poorly documented, it works like a champ and is not very computationally expensive. Its usage can be summarized by this code block:

```
synth winona;
winona.begin();

/*            0-3            0-127           0-127  0-127 */
/* setupVoice(voice, waveform, pitch, envelope, length, mod) */

/* waveforms: SINE, TRIANGLE, SQUARE, SAW, RAMP, NOISE */
/* envelopes: ENVELOPE0, ENVELOPE0, ENVELOPE1, ENVELOPE2, ENVELOPE3 */

winona.setupVoice(1, TRIANGLE, 0, ENVELOPE1, 127, 64);
winona.setupVoice(2, SINE    , 0, ENVELOPE3, 127, 64);
winona.trigger(1, C4);
winona.trigger(2, E5);
```

That's literally it, I didn't use any other functions from the library (largely because it doesn't offer any... haha). But, as you can see, it provides a pretty logical and unencumbered way to play sounds, leaving me room to implement the logic and hardware components of the synth.

Next, I'll discuss how I set up the **hardware components**. Here's what the synth looks like under the shell (disregard the switch and button, those are remnants from another life):



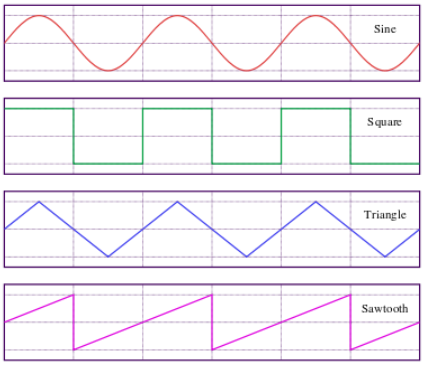
The **buttons** can be in one of two states: pressed or not pressed (i.e. on/off, 1/0, HIGH/LOW, beep/boop, encendido/apagado, etcetera). To detect and respond to such events, I connected the buttons to the digital (read: on/off) input pins of the Arduino. Buttons are just another type of switch -- you provide 5V of electricity to one of the pins, and it flows through to the other pin only when the button is pressed. Therefore, hooking up said other button pin to a digital input pin of the Arduino will allow us to test for whether the button is pressed -- if there is voltage (the read is HIGH), then the button is pressed.

The only thing we have to watch out for are floating inputs, where the ambient electricity in the air (or on your body, when you move your fingers near the device) may

make the button read as if it is pressed, even if it isn't. To counter this, we can use a "pull-down resistor" connecting the button pin to ground, assuring the button will only read as high when it's actually pressed. This resistor can be of any sufficiently high value -- I used ones that were rated as 10K Ω. Repeat this simple setup for all 12 of the buttons and you're golden.

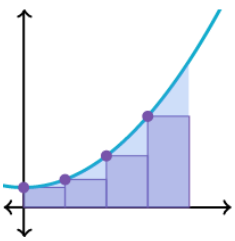
The treatment of the **knobs** (or, potentiometers) is not so different, except for the fact that they are analog devices. As you can see from the figure on the right, potentiometers have 3 pins. One of these is connected to ground (on the side where you want the pot to read as zero when it's turned there), and another to a power source (on the side where you want the pot to read as the maximum value when it's turned there). The middle pin will be your signal -- its output depends on where the pot's knob has been turned. Since the Arduino is a digital device, somewhere in the process of reading the pot the actual value has to be converted to an integer (the `analogRead()` function responsible for this has 10 bits of precision, so it'll yield a value between 0 and 1024). If we connect this middle pin to the analog input pins of the Arduino (labeled as A0, A1, .. A5), we can read the current position of the knobs at will.

We've covered the inputs, now we need to talk about the **output**. Dzl TheEvilGenius's synth library reserves pin 11 as its output (though it allows us the option to use pin 3 instead if we choose to). Notice that these are both PWM-capable pins. This is because the synth library uses PWM (pulse-width modulation) to shape its signals. Remember that the Arduino is a digital device, it can only send signals of on or off. How is this device supposed to create something like a sine wave, which has a smooth curve from off to on?



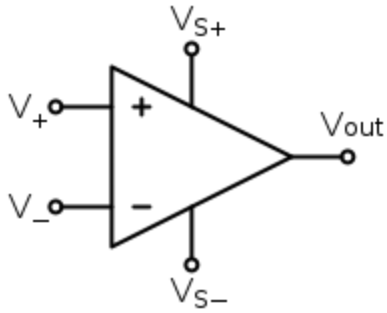
The Arduino can only really create square/rectangular waves

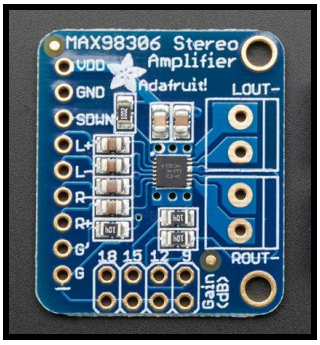
If you know anything about calculus and integrals, then you already know the essence of the answer to this riddle:



Basically, you estimate the signal by emitting rectangular waves at a certain frequency -- by varying the ratio of how long the wave stays on and how long it stays off, along with changing the frequency, you can emulate many wave shapes. Hence pulse-width, and modulation, because you are modulating the width of the pulse. And that's what this library does!

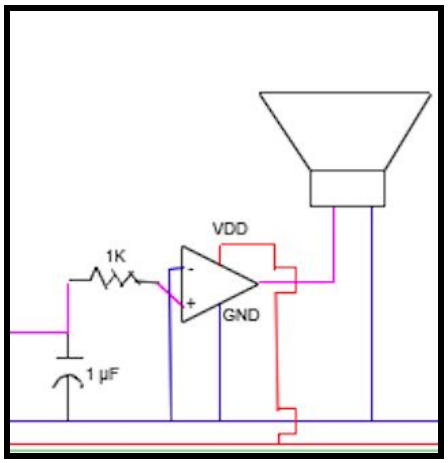
So, we have this PWM signal coming from pin 11 of the Arduino. The easiest thing to do is to connect it directly to a speaker, and voila! And yes, if you're wondering, that's what I did at first too -- the only problem is that you can barely hear anything. You'll need much more current to drive a speaker, so an amplifier is needed. Amplifiers are circuits in and of themselves, they're not fundamental electronic parts (though any part has a level of internal complexity). One way to amplify your signal is by using a transistor, which I tried to do, but it didn't have a profound enough effect. So, I invested in a *Stereo 3.7W Class D Audio Amplifier*, which, as you can see by the colorful text and Comic Sans font, is very fancy. It's nothing crazy, though, you just feed it 5V and ground into the VDD and GND, and the audio signal as well as ground into L+ and L-, and the output should come out much louder at the other end!





There's one last hitch -- amplifying the signal also amplifies any noise that comes with it. You know, the annoying *bzzzzzzzzzzzzzzzzzzz*, which is annoying. Here in computer science we like Fizz and Buzz but no *bzzzzzzzz*, understand? To fix this, I created a lowpass filter (which, as the name implies, allows low frequencies to pass through while blocking high ones). There are many types of lowpass filters -- the one I employed was an RC filter, so called because it's implemented using a resistor and a capacitor. The values of the resistor and capacitor dictate exactly which frequencies get filtered out, and how strongly.

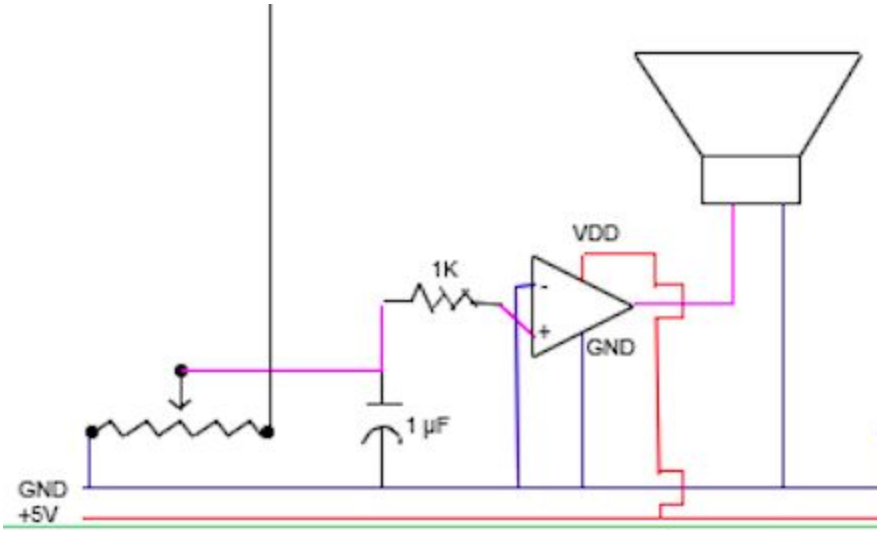
Here's the circuit:



You basically feed the signal between a capacitor and resistor, and then get the filtered output out the other end of the resistor -- not so hard (the pink line is the signal).

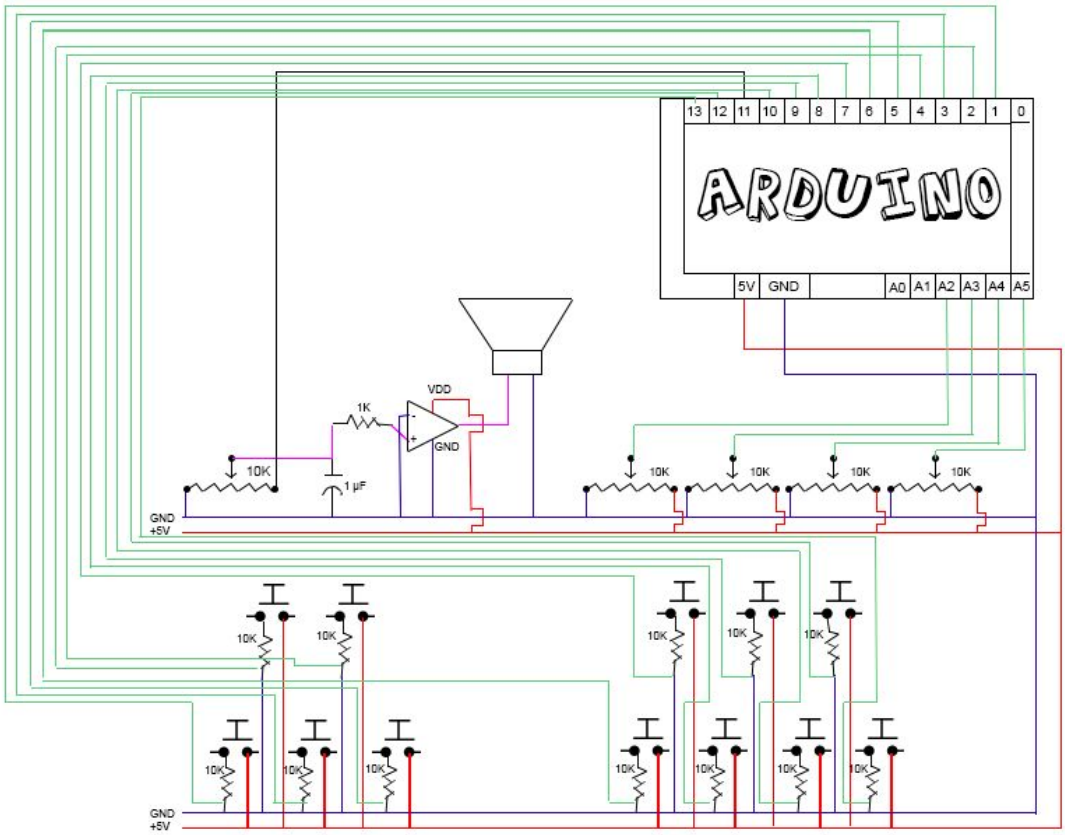
The only thing left to do is to install the potentiometer that allows us to adjust the volume. This potentiometer won't be an analog input to the Arduino; rather, we'll be using its

variable resistance (which is what a pot is…) to adjust the strength of the signal.
Instead of feeding our audio signal directly to the filter, we'll first feed it into the volume potentiometer (as one of its side pins, the other being connected to ground), and then feed the output signal of that (now reduced in volume) into the amplifier, which finally goes into the speaker. Here's a schematic of that:

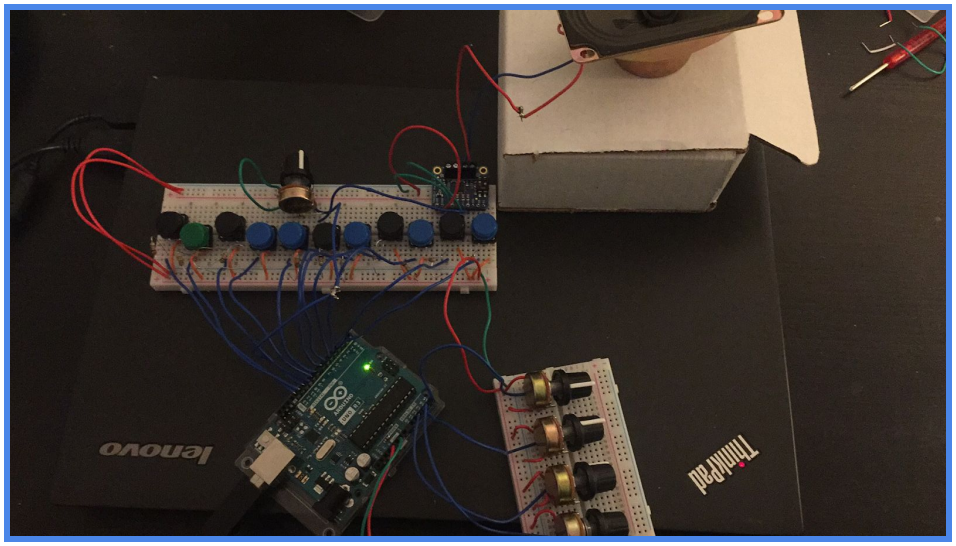


The black line is the audio output coming from the Arduino

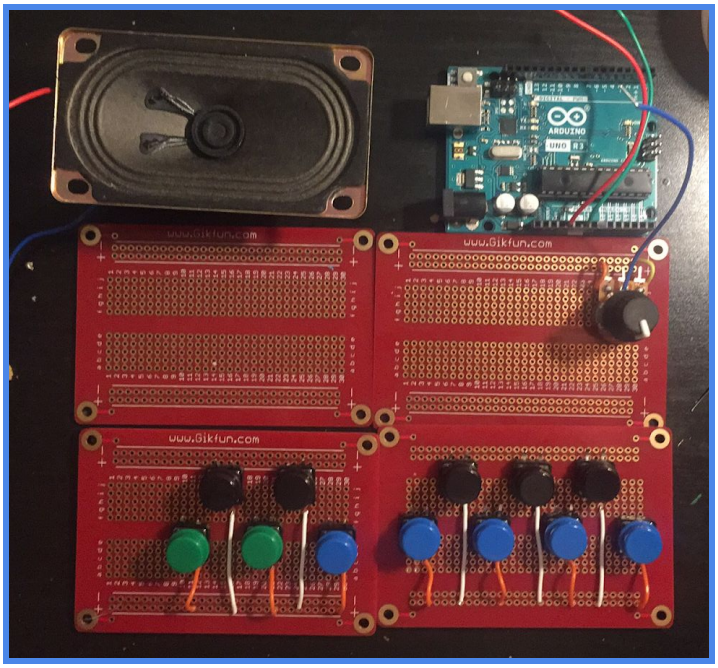
In fact, I think you're ready to see the entire schematic now, as we're done covering how this thing is set up.



Here are some bonus pictures of the thing at various stages of the building process:



Oh, Le Olde Breadboard Times. How I miss them :'(



Soldering is hard, man...

Oh, and one last note before we continue -- if you are going to get potentiometers and try to use them as knobs in this manner, make sure you get ones which are meant to face upwards! The ones I got face sideways, so I had to solder on wires to each pin (that's 15 pins in total...) in order to get them to a state where they can stand upwards. I ended up overheating like >4 pots this way, and I wouldn't recommend it to anyone. While you're at it, make sure you buy long ass wires -- I only had short wires so I had to create my own long wires by trimming and tying and soldering and repeating over and over again (I needed like 12 of them).

We've arrived at the **program** portion of this document -- what does the Arduino need to do to facilitate the communication of all of these parts? At its most basic, this is the algorithm:

1. check which buttons are pressed
2. read the values of the potentiometers
3. play the notes corresponding to buttons that are pressed, with the specific sound settings as determined by the potentiometers

Of course it's never actually *that* simple. Firstly, neither the Arduino nor the library has any conception of a "note" -- the library allows us to specify the frequency we want to play, though, so we can work with that. Here's a solution:

```
#define AS4 466
#define B4  494

#define C5  523
#define CS5 554
#define D5 587
#define DS5 622
#define E5 659
#define F5 698
#define FS5 740
#define G5  784
#define GS5 831
#define _A5  880
#define AS5 932
#define B5  988

#define C6  1047
#define CS6 1109
#define D6  1175
```

I shoved this in a file called *notes.h* and `#included` it into our main file. Now we can refer to notes as if they exist. Oh, why is A5 preceded by an underscore, you ask? Remember how the Arduino has analog inputs A0 through A5... yeah, that happened. Our `#define` took precedence over the A5 as defined by the Arduino, and thus broke the process of reading from the potentiometers. This is literally the worst bug ever, and I think it's hilarious. I also have no idea how I realized that this was the case.

Similarly, there's no such thing as an octave in Arduino-land, so I created an "octave table". If we read a value between 0 and 6 from the octave knob, we can use that as an index to this table, and determine what note to play that way.

```
const int OCTAVE_TABLE[7][12] = {
  {C1, CS1, D1, DS1, E1, F1, FS1, G1, GS1, _A1, AS1, B1},
  {C2, CS2, D2, DS2, E2, F2, FS2, G2, GS2, _A2, AS2, B2},
  {C3, CS3, D3, DS3, E3, F3, FS3, G3, GS3, _A3, AS3, B3},
  {C4, CS4, D4, DS4, E4, F4, FS4, G4, GS4, _A4, AS4, B4},
  {C5, CS5, D5, DS5, E5, F5, FS5, G5, GS5, _A5, AS5, B5},
  {C6, CS6, D6, DS6, E6, F6, FS6, G6, GS6, _A6, AS6, B6},
  {C7, CS7, D7, DS7, E7, F7, FS7, G7, GS7, _A7, AS7, B7}
};
```

The next trivial-seeming thing which actually isn't trivial is reading buttons. Firstly, if we just check whether a button is currently pressed, that's all we get -- whether it is being pressed, not whether it has *just been pressed*. We only need to perform an action when the button's state goes from being unpressed to being pressed, so we need some kind of way to detect that. My solution: use a "cache" which holds what the button's state was previously -- if the button's cache is LOW but it's current read is HIGH, it was just pressed. Solved! Another problem, though, is that sometimes signals linger for a brief but detectable moment, so you can misread things as being pressed when they're not (yes, this is distinct from floating inputs). To fix this, we need to employ a technique called

debouncing. There are many ways to do this, some with hardware and others with software. The method I went with can be summed up as “read the button, then read it again 10 milliseconds later”. The specifics of the technique can be found at http://gammon.com.au/switches.

Now we’re talking! Thus our main loop can be expressed as follows:

```
void loop():
    read_button_states()
    read_pot_states()
    apply_pot_settings()

    for each button:

        int note = OCTAVE_TABLE[INPUT_OCTAVE][button]

        if is_pressed(button):
            last_debounce_time = current_time_in_milliseconds()
            play(note);

        else if is_released(button):

            last_debounce_time = current_time_in_milliseconds()
            silence(note)

    cache_button_states()
```

This is pseudocode, by the way, don’t try to run this or bad things will happen! JK. The complexity is nicely hidden away in all of the helper functions, but this is basically what’s happening. If you take a look at the code, you’ll see my annotations there for extra things I had to add because the world isn’t perfect and unfortunately neither is Dzl TheEvilGenius. Regardless, we’re done now! We’ve covered how this synth works, all the way from soup to nuts (yeah, that’s a saying, get over it).

Deviations From the Original Idea

Sadly some of the things I originally planned to implement did not make their way into the final build. For the most part, this was because I underestimated how long things would take, so I ran out of time (other projects for school didn't help with my time management either…). Here's what went missing:

1. A battery
2. An arpeggiator mode, with a 7-segment screen to represent that
3. A 3.5mm jack, with a switch to route audio there instead

#3 would've been trivial to implement, I even had it all ready until the last moment but I just couldn't get to it in time. #1 would've been possible (but not the complex rechargeable battery scheme I was planning, probably just a regular old battery kinda deal). #2 is really two separate parts -- the arpeggiator mode and the 7-segment display. Creating an arpeggiator mode would've been very simple, it would be a matter of checking whether a switch is flipped, and calling a different function if it is. In fact, I could even create a "dumb" arpeggio just by changing the length of the notes (that way there'd be a gap between each). With no arpeggiator, there was no need for an LED display to tell us what mode we're in. Either way, for the screen I would've needed like 10 more pins, but I only had 1 or 2 remaining. I was considering using a multiplexor and all that fancy stuff, but that went out the window when I ran out of time.

Parts List

12 Gikfun Buttons -- Amazon [link] Part #LYSB01E38OS7K-ELECTRNCS
12 10K Resistors -- Sparkfun Resistor Kit [link] COM-10969
1 1K Resistor -- Sparkfun Resistor Kit [link] COM-10969
1 1uF Polarized Capacitor -- Sparkfun Capacitor Kit [link] Kit #13698
5 10K Linear Potentiometers -- Adafruit [link] Part #P160KN-0QC15B10K
5 Potentiometer Knobs -- Adafruit [link] Product ID #2047
1 4Ω Impedance Speaker -- salvaged from broken set of speakers
1 Stereo 3.7W Class D Audio Amplifier -- Adafruit [link] Part #MAX98306
30+ Jumper Cables
1 Arduino Uno Rev3 -- Arduino [link] Barcode #7630049200050
4 Gikfun Solder-able Breadboards -- Amazon [link]
1 WYCTIN 60-40 Tin Lead Rosin Core Solder Wire -- Amazon [link]

You also probably need some breadboards, a soldering iron, wire cutters, wire strippers, an LED for testing stuff, and a USB Type A Male to USB Type B Male connector. Note: these parts don't include what was used for the case -- for that you need a 3D printer, ABS filament, hot glue, and probably some cardboard hahaha.