Make your own 8-bit synths with Arduino

Getting started with the Mozzi library to get your Arduino wailing

Chris Ball

@ChrisBallMidi

Chris Ball is a technologist working in Manchester, UK. He has worked on a variety of interactive art installations. You can visit his site at **chrisballprojects.co.uk**

YOU'LL NEED

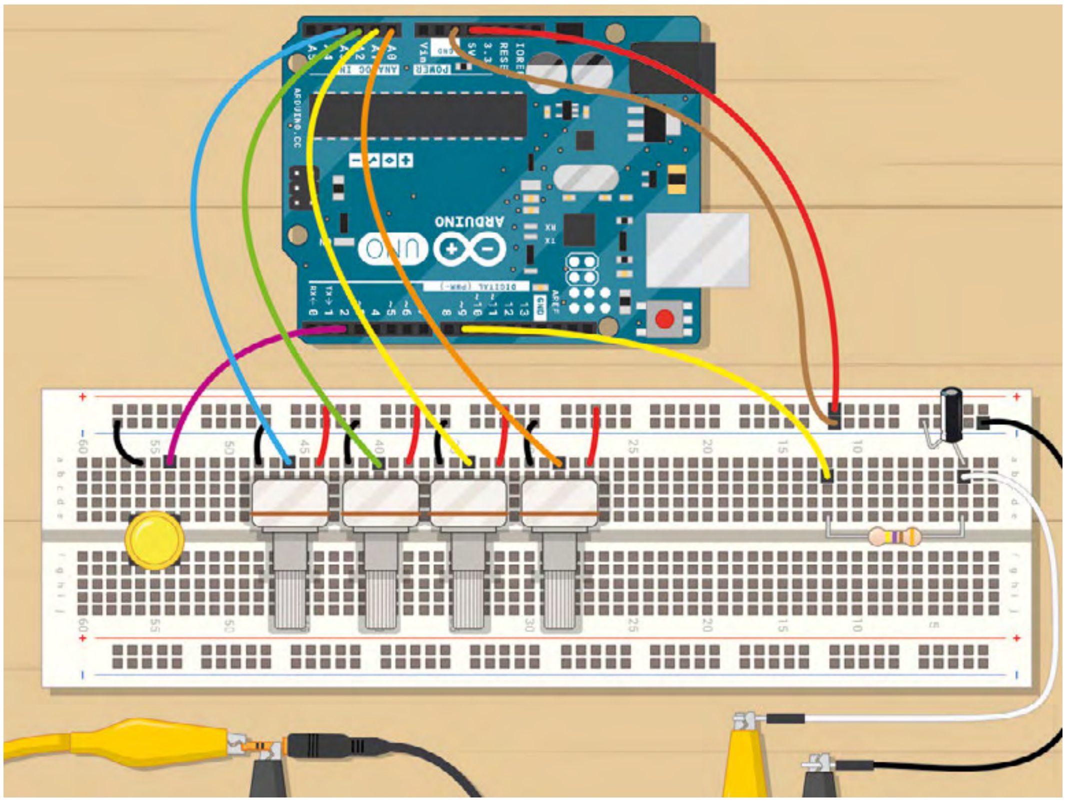
- **An Arduino** (Preferably Uno, although others are possible)
- **Breadboard**
- **470 Ω resistor**
- **Tactile button**
- **4 × 10 kΩ linear potentiometers** (usually marked B10K)

So, you've got your first Arduino, and you've tried a few basic projects. Maybe you've got an LED blinking and now you're struggling to find a project that's a little more creative. Look no further, we've got you covered! You may have achieved some basic bleeps and bloops with the built in Tone() function, but we'll be doing some much more advanced digital synthesis.

Digital synthesizers are very different from their analogue counterparts. Instead of a complex collection of diodes, amplifiers, oscillators, and other esoteric audio electronics, they mainly use processing power to generate waveforms and effects. Digital synths have other benefits too, but their main strength is that once set up, they're extremely reconfigurable; you don't need to rebuild your synth to change its sound, just reprogram it.

Throughout this tutorial we'll be using the Mozzi library to create a variety of sounds. The library is capable of generating complex waveforms, audio effects, and playing short samples, all from the modest hardware in an Arduino. We'll be using it to create a basic FM (frequency modulation) synthesizer.

We'll get started with the absolute bare minimum for a Mozzi-based sketch. Make sure you've installed



Right Our final synth, with the four potentiometers we need to play with to create the sound of the future the Mozzi library, then start your Arduino environment and open the example under File > Examples > Mozzi > Basics > Sinewave. This is a sine wave generator, which is pretty much the digital audio equivalent of a 'Hello, World!' program.

Here you'll see the basics of a Mozzi program, and you might notice it has a slightly more complex structure than your usual Arduino sketch. Let's ignore that for now, and get making some sound. Upload the code to your Arduino. If all is well, a sine wave will be generated on pin 9, and we just need to listen to it.

To connect the Arduino to our amplifier/earphones we need to connect the following:

Arduino pin 9 → 470 Ω resistor → Audio jack tip (the resistor is to help protect pin 9)
Arduino GND → Audio jack base

If all's well, you should hear a sine wave at 440 Hz. If you have no sound, check your volume, connections, and that the sketch has uploaded successfully.

If you've had some success, we'd recommend at this point that you take a look at some of the other examples the Mozzi library has to offer. This will give you an idea of what it's capable of, but bear in mind that some examples expect extra hardware.

Back to the sine wave generator: we'll be breaking down the elements of this sketch fairly thoroughly, as knowing the basics of how Mozzi works will enable you to make more exciting changes later.

First, we'll take a look at the includes. This is where we add in the required files from the Mozzi library, and you should see three includes: 'MozziGuts.h', 'Oscil.h', and 'tables/sin2048_int8.h'.

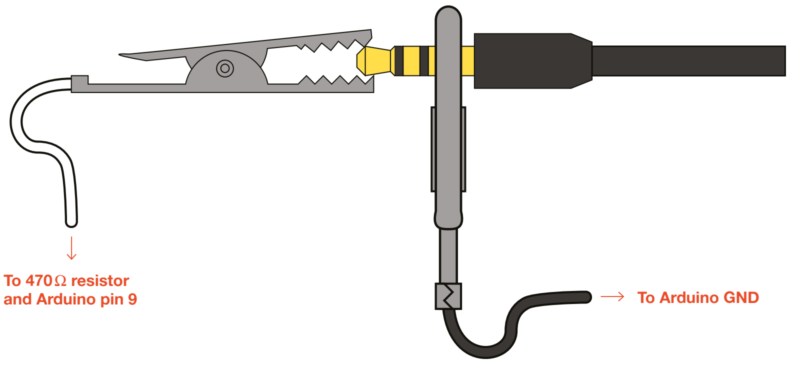
MozziGuts.h is the main library required for doing anything with Mozzi. This file will adapt your Arduino for use as a synth, by taking over some timers and setting up some fast sampling methods.

Oscil.h is simply a template for an oscillator. Any sound requires a repeated change in voltage, or air pressure; an oscillation. This file tells Mozzi how to create an oscillator from a lookup table.

tables/sin2048_int8.h is the lookup table we'll be using to make a sine wave. A lookup table is often used where calculating the values of a function (in this case, a sine wave) would take too long. We simply pre-calculate all the values and store them in memory. When we need them, we can simply 'look them up', hence the name lookup table.

We then have a line:

```
Oscil <SIN2048_NUM_CELLS, AUDIO_RATE>
aSin(SIN2048_DATA);
```



Above How to connect crocodile clips to an audio jack

This is a little like saying, "Create a sine wave oscillator called aSin, using the table I mentioned before." We also have the line:

```
#define CONTROL_RATE 64
```

Which means we intend to update our controls (our potentiometers and buttons) 64 times per second. Mozzi asks for control rates to be powers of two (e.g. 2, 4, 8, 16, …)

To continue to our main functions: in **`setup()`** you'll see two commands. The first, **`startMozzi(CONTROL_RATE)`**, will start the Mozzi engine, and the second, **`aSin.setFreq(440)`**, will set the frequency (or pitch) of our oscillator. 440 Hz is middle A (so if you only get this far, at least you can get your band in tune).

Typically, when writing a Mozzi sketch, you'll avoid putting anything in **`loop()`**, except the function **`audioHook()`**. This function will calculate samples (little chunks of audio data) ready to be written to our output. So where do we put our code? You'll notice, apart from the usual **`setup()`** and **`loop()`** functions that we have two more: **`updateControl()`** and **`updateAudio()`**.

`updateControl()` is where we put the changes we want to happen at our control rate (64 times per second). This will be things like reading our potentiometer values, button states and other tasks that don't need to happen too often. In this sketch, nothing like this is required, so the function is empty.

`updateAudio()` is the function that **`audioHook()`** will run repeatedly – it calculates our audio samples and stores them in a buffer to be sent to pin 9 later. You can see within this sketch the code: →

INSTALLING THE MOZZI LIBRARY

From **sensorium.github.io/Mozzi** you can click on the download link. This will take you to an optional donation page where you can help the author of the library out, if you choose. You'll then be taken to the GitHub project page for Mozzi. Click the 'Source Code .zip' link, download the zip file, and extract the contents to your **Arduino/libraries** directory (this is usually in your **documents** folder).

Your file structure should then look like this:
Documents/Arduino/Libraries/Mozzi-1.0.3 (although you may have a different version).

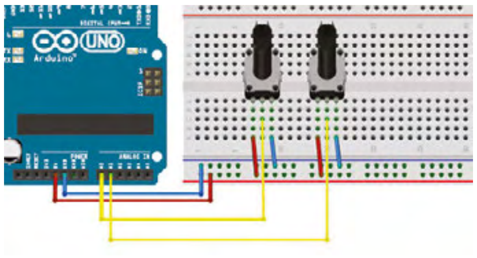


Figure 1 Connecting two pots to your Arduino

`return aSin.next();` which simply means to send the next sample for this oscillator to the buffer.

Let's make a couple of changes to the way this works. We'll add one pot (potentiometer) to control frequency, and a second pot to control volume.

Connect two pots to your Arduino (**Figure 1**). Each pot will have one side connected to 5V, the other side connected to GND and the middle (wiper) to an analogue input. We'll use analogue inputs A0 and A1.

Add the following lines of code before `void setup()`:

```
int pot0, pot1;
int volume,frequency;
```

These will be the variables where we'll store the pot values, and the frequency and volume values they will control.

Add the following lines of code inside your `updateControl()` function:

```
pot0 = mozziAnalogRead(A0);
pot1 = mozziAnalogRead(A1);
frequency = pot0 + 50;
volume = map(pot1, 0, 1023, 0, 255);
aSin.setFreq(frequency);
```

The first two lines will store our pot voltages as variables, `pot0` and `pot1`.

The third stores the value of `pot0` + 50 to a variable called `frequency`. We've added the +50 to prevent the frequency becoming too low to hear.

The fourth line will store the value of `pot1` to a variable called `volume`, but will scale it in the process to be between 0 and 255 (instead of 0 and 1023).

The last line will set the frequency of our oscillator to the value in the `frequency` variable

This covers changing our frequency, but we need to make one last change in `updateAudio()` for the volume control to work.

Change the line:

```
return aSin.next();
```

to:

```
return (aSin.next()*volume)>>8;
```

This line may look confusing, but it's very similar to multiplying the output by a value between 0 and 1. It's good to get used to calculating this way as it's significantly faster with integer values on an Arduino, and we need speed to calculate all our sample values.

If you upload these changes, you now have a basic synthesizer! You should be able control pitch with pot 0 and volume with pot 1.

So perhaps you've played that for a while and become bored already. This was bound to happen – it's only a simple synthesizer. Let's try adding another sine wave oscillator, and another potentiometer to control it. To add another potentiometer, you can repeat the connection pattern as before, with our middle wiper pin wired to A2 on the Arduino. We already have the sine wave lookup table we need, so we can do this simply by duplicating the line:

```
Oscil <SIN2048_NUM_CELLS, AUDIO_RATE>
aSin(SIN2048_DATA);
```

You'll need to give our oscillators distinct names, so we should change this to:

```
Oscil <SIN2048_NUM_CELLS, AUDIO_RATE>
aSin1(SIN2048_DATA);
Oscil <SIN2048_NUM_CELLS, AUDIO_RATE>
aSin2(SIN2048_DATA);
```

We'll add and change some variables too:

```
int pot0,pot1,pot2;
int frequency1,frequency2,volume;
```

Our `updateControl()` function will become:

```
pot0=mozziAnalogRead(A0);
pot1=mozziAnalogRead(A1);
```

QUICK TIP

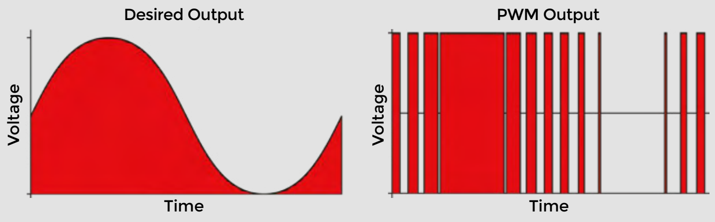
The '>>' and '<<' symbols are called bitshift operators, and they are a very fast way of dividing or multiplying by 2. The '>>8' is a little like saying "divide by 2, 8 times". If our volume value was 200, you could think of this line as Output × (200/256).

DIGITAL TO ANALOGUE WITH PWM

You might have realised that we are using pin 9, a digital pin, to do the job of an analogue output – how does this work? We are using pulse-width modulation (PWM). Simply put, if we want to approximate 2.5V with a 5V digital output, we switch the digital pin high for 50% of the time. 1V would be 20%, 2V 40%, and so on.

PWM is most commonly used for making lights (particularly LEDs) appear at different brightnesses or motors run at different speeds, all by switching a constant voltage on or off.

This approach does have significant downsides, though – mainly that it will introduce a lot of noise at your modulation frequency. Not a problem for motors or LEDs, but your ears will probably notice straight away.



```
pot2=mozziAnalogRead(A2);
frequency1=pot0+50;
frequency2=pot1+50;
volume=map(pot2, 0, 1023, 0, 255);
aSin1.setFreq(frequency1);
aSin2.setFreq(frequency2);
```

And our `updateAudio()` code will be changed also:

```
return volume*((aSin1.next()+aSin2.next())>>1)>>8;
```

Our two sine waves, when added together, could add up to a number higher than our PWM output can reproduce. In audio circles this is called 'clipping' and is generally avoided (unless you're intentionally after a distorted sound). We've prevented this here by dividing the output by two.

The above changes should result in two controllable sine waves on pots 0 and 1. You may even be able to get some interesting 'throbbing' if you pitch the notes close together – this is called 'beating' and is caused by interference between the two frequencies.

To develop the synth further, we'll introduce frequency modulation (FM). This means we'll use the output of one sine wave to control the frequency of another, resulting in varied timbres.

We'll also be making some changes to our hardware: adding another potentiometer; and introducing a push button to trigger the audio.

If you make these changes to the circuit, and upload the code from **hsmag.cc/JPNNBP**, you should have yourself an FM synthesizer!

The magic happens in two lines. This one, in `updateControl()`:

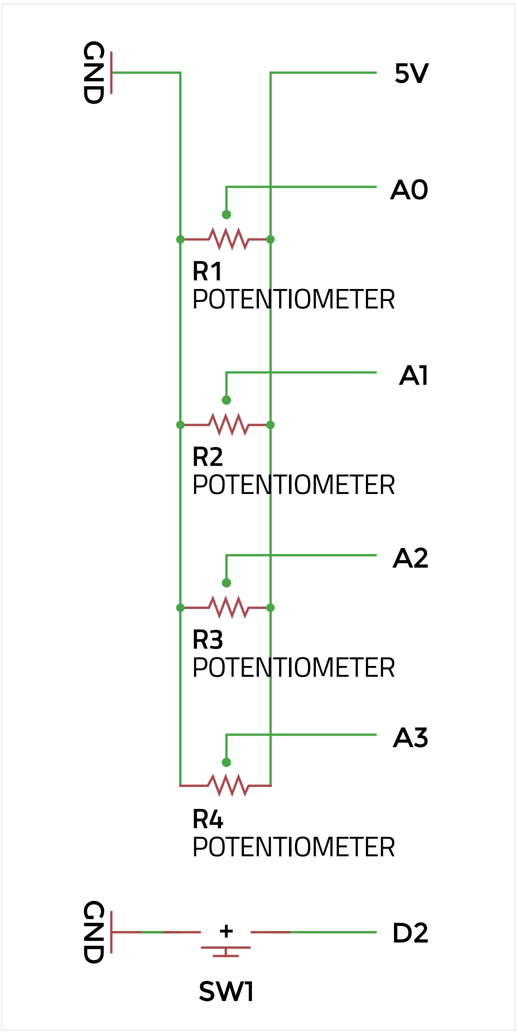
```
aSin2.setFreq(frequency2);
```

And this line, in `updateAudio()`:

```
aSin1.setFreq(frequency1+(amount*(aSin2.
next())>>8));
```

The first sets the frequency of our modulation, and the second uses that to control the frequency of our main waveform. There is also an `amount` control that will multiply our modulation, with some interesting effects. Remember, now you'll need to push the trigger button to hear sound! Try changing some of the numbers in this code and see how they affect the output.

So, you should have a basic 8-bit synthesizer, but more importantly, an idea of how to use the Mozzi library to develop it further. Mozzi has a huge selection of basic waveforms, some audio effects, and it's extremely well documented, with great examples. If you feel lost at any point, you can always check on the website. □



QUICK TIP

Having problems getting the first example working? There's a much more thorough walkthrough at **hsmag.cc/AwksSP**

Left The final circuit diagram for the breadboard

OTHER ARDUINO AUDIO PROJECTS

ElectroSmash PedalShield: This is a kit designed to sit on top of an Arduino Due and turn it into a general-purpose guitar effects pedal. It has some basic examples available, and a forum with many more. **Electrosmash.com/pedalshield**

Ardutouch: International hacker Mitch Altman has created an Arduino-based synth project called Ardutouch, built on a fantastic library by himself and Bill Alessi. The library by itself is great to mess around with, although it may require an experienced Arduino user. **cornfieldelectronics.com/cfe/projects.php**

Teensy Audio Board: This hardware for the Teensy 3.1/3.2 and the accompanying audio library get an honourable mention simply because it's so fully featured. Not strictly Arduino, but Arduino-like. **pjrc.com/teensy/td_libs_Audio.html**

There are many more useful libraries in the Arduino Library List (playground.arduino.cc/Main/LibraryList) under the 'audio' section.