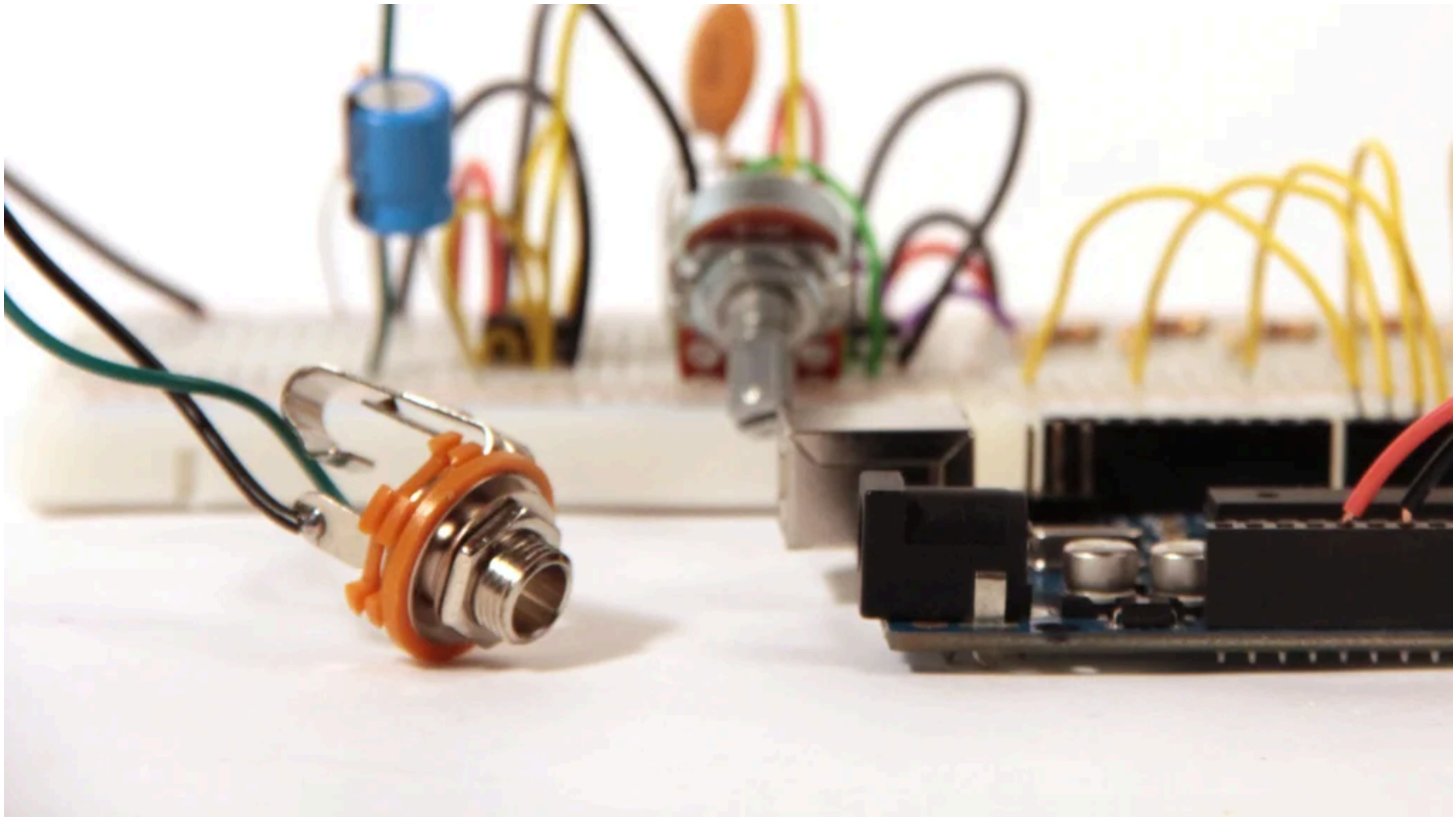
AUTODESK Instructables

Arduino Audio Output

By amandaghassaei in CircuitsArduino

Introduction: Arduino Audio Output



Generate sound or output analog voltages with an Arduino. This Instructable will show you how to set up a really basic digital to analog converter so you can start generating analog waves of all shapes and sizes from a few digital pins on an Arduino. (This article is a companion to another Instructable I've written about sending audio into an Arduino, find that here)

Some ideas that come to mind:

sample based instrument- store samples on the Arduino or on an SD card and trigger playback with buttons or other types of controls. Check out my Arduino drum sampler for an idea of how to get started.
digital synthesizer- make saw, sine, triangle, pulse, or arbitrary waveshapes- check out my waveform generator to get started
MIDI to control voltage module/ MIDI synthesizer- receive MIDI messages and translate them into a voltage so you can control an analog synthesizer with MIDI, or use the MIDI data to output audio of a certain frequency
analog output- you may find yourself needing to generate analog voltages from your Arduino at some point, maybe to communicate with an analog device
effects box/digital signal processing- in combination with a microphone/audio input you can

perform all kinds of digital signal manipulations and send the processed audio out to speakers. Check out my vocal effects box for an example.
audio playback device- make your own ipod. With the addition of an SD shield you could create your own Arduino mp3 player (check out the wave shield documentation for an idea of how to get started with the code). The circuits and code provided here are compatible with SD shields that communicate via SPI.

Feel free to use any of the info here to put together an amazing project for the DIY Audio Contest! We're giving away an HDTV, some DSLR cameras, and tons of other great stuff! The contest closes Nov 26.

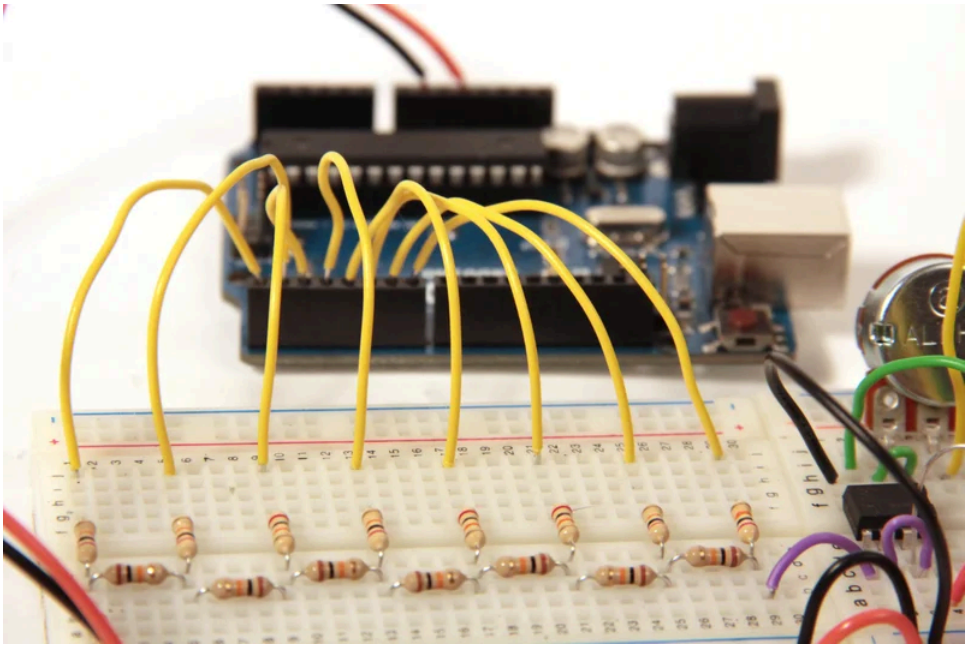
PartsList:

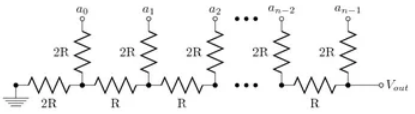
(x9) 1/4 Watt 20kOhm Resistors Digikey 0KQBK-ND
(x7) 1/4 Watt 10kOhm Resistors Digiikey CF14JT10K0CT-ND
(x2) TS922IN Digikey 497-3049-5-ND I like these because they can be powered off the Arduino's 5V supply (one 924 works too, but they don't seem to be available on digikey at the moment)
(x1) 10kOhm potentiometer linear Digikey 987-1308-ND
(x1) 0.01uF capacitor Digikey 445-5252-ND
(x1) 220uF capacitor Digikey P5183-ND
(x1) 0.1uF capacitor Digikey 445-5303-ND
(x1) 1/4 Watt 3kOhm Resistor Digikey CF14JT3K00CT-ND
(x1) 1/4 Watt 10Ohm Resistor Digikey CF14JT10R0CT-ND
(x1) Arduino Uno Amazon

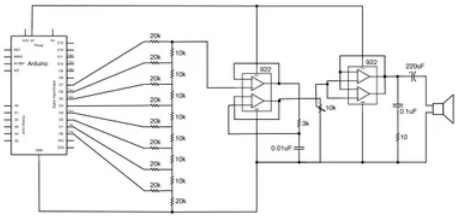
Additional Materials:

(1x) usb cable Amazon
(1x) breadboard (this one comes with jumper wires) Amazon
(1x) jumper wires Amazon

Step 1: Digital to Analog Converter







DAC stands for "digital to analog converter." Since the Arduino does not have analog out capabilities, we need to use a DAC to convert digital data (numbers/ints/bytes) to an analog waveform (oscillating voltage). A simple, easy to program, and cheap way to do this is to use something called an R2R resistor ladder. Essentially, it takes incoming digital bits (0V and 5V from Arduino), weights them, and sums them to produce a voltage between 0 and 5 volts (see the schematic in fig 2, taken from the Wikipedia resistor ladder page). You can think of a resistor ladder as a multi-leveled voltage divider.

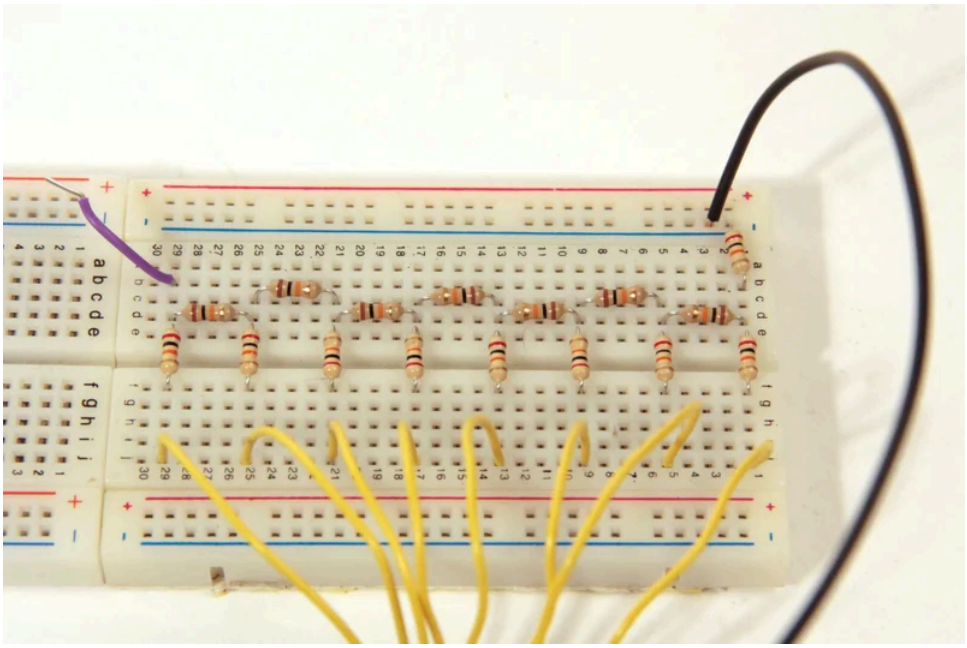
The resistor ladder I'll be demonstrating in this tutorial is an 8-bit DAC, this means it can produce 256 (2^8) different voltage levels between 0 and 5v. I connected each of digital pins 0-7 to each of the 8 junctions in my 8 bit DAC (shown in figs 1 and 3).

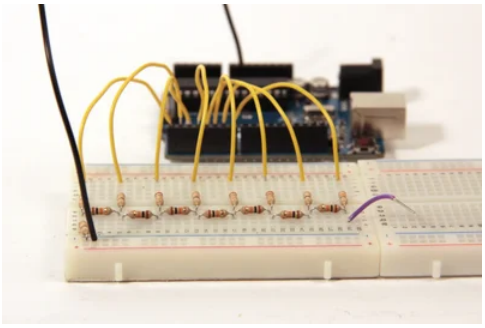
I like using these resistor ladder DACs because I always have the materials around, they're cheap, and I think they're kind of fun, but they will not give you the highest quality audio. You can buy a chip that works in the exact same was as an R2R DAC (and will work with all the code in this instructable), but has internal, highly-matched resistors for better audio quality, I like this one bc it runs off a single 5V supply (you can even do stereo audio with it), but there are many more available, look for "parallel input, 8 bit, dac ic".

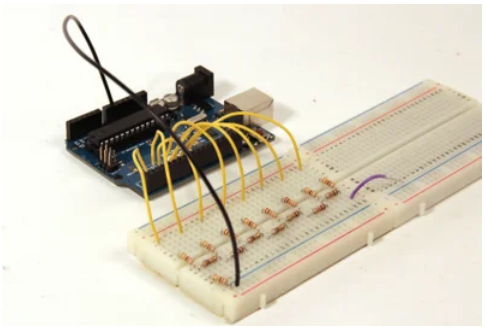
Alternatively, there are chips that take in serial data to perform digital to analog conversion. These chips are generally higher fidelity (definitely better quality that the resistor ladder DAC) and they only use two or three of the Arduino's output pins (as opposed to 8). Downsides are they are a little more challenging to program, more expensive, and will not work with the code in this Instructable, though I'm sure there are some other tutorials available. After a quick search on digikey, these looked good, for Arduino, try to find something that will run off a single 5V supply.

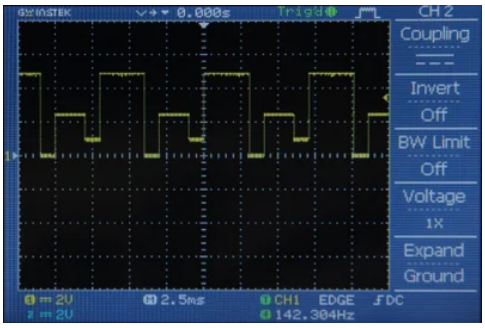
One more note - there seems to be kind of a misconception abut 8 bit audio- that it always has to sound like the sounds effects from a Mario game- but 8bit audio with this really basic DAC can actually replicate the sounds of people's voices and instruments really well, I'm always amazed at the quality of sound that can come from a bunch of resistors.

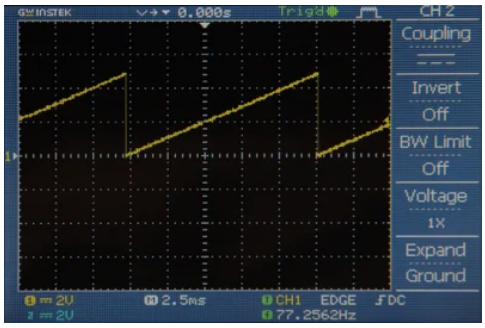
Step 2: Set Up DAC and Test

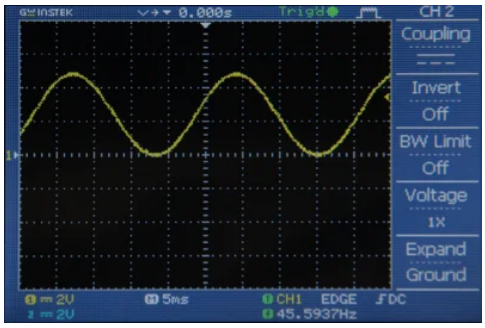


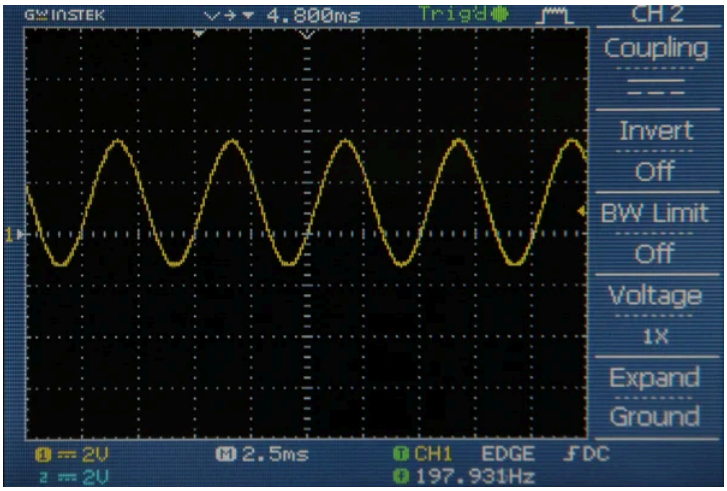


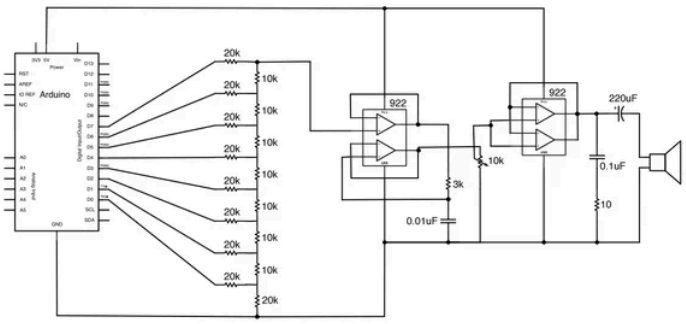












I constructed my DAC on a breadboard (figs 1-3). The schematic is given in fig 8. Below are a few pieces of sample code that generate the waveforms shown in figs 4-7. In the following pieces of code I send a value between 0 and 255 to "PORTD" when I want to send data to the DAC, it looks like this:

PORTD = 125;//send data to DAC

This is called addressing the port directly. On the Arduino, digital pins 0-7 are all on port d of the Atmel328 chip. The PORTD command lets us tells pins 0-7 to go HIGH or LOW in one line (instead of having to use digitalWrite() eight times). Not only is this easier to code, it's much faster for the Arduino to process and it causes the pins to all change simultaneously instead of one by one (you can only talk

to one pin at a time with digitalWrite()). Since port d has eight pins on it (digital pins 0-7) we can send it one of 2^8 = 256 possible values (0-255) to control the pins. For example, if we wrote the following line:

PORTD = 0;

it would set pins 0-7 LOW. With the DAC set up on pins 0-7 this will output 0V. if we sent the following:

PORTD = 255;

it would set pins 0-7 HIGH. This will cause the DAC to output 5V. We can also send combinations of LOW and HIGH states to output a voltage between 0 and 5V from the DAC. For example:

PORTD = 125;
125 = 01111101 in binary. This sets pin 7 low (the msb is 0), pins 6-2 high (the next five bits are 1), pin 1 low (the next bit is 0), and pin 0 high (the lsb is 1). You can read more about how this works here. To calculate the voltage that this will output from the DAC, we use the following equation:

voltage output from DAC = [(value sent to PORTD) / 255] * 5V
so for PORTD = 125:
voltage output from DAC = (125 / 255) * 5V = 2.45V

The code below sends out several voltages between 0 and 5V and holds each for a short time to demonstrate the concepts I've described above. In the main loop() function I've written:

```
PORTD = 0;//send (0/255)*5 = 0V out DAC
delay(1);//wait 1ms
PORTD = 127;//send (127/255)*5 = 2.5V out DAC
delay(2);//wait 2ms
PORTD = 51;//send (51/255)*5 = 1V out DAC
delay(1);//wait 1ms
PORTD = 255;//send (255/255)*5 = 5V out DAC
delay(3);//wait 3ms
```

The output is shown on an oscilloscope in fig 4. The center horizontal line across the oscilloscope represents 0V and each horizontal line represents a voltage increase/decrease of 2V. The image notes on fig 4 show the output of each of the lines of code above, click on the image to view the image notes.

```
*/

void setup(){
  //set digital pins 0-7 as outputs
  for (int i=0;i<8;i++){
    pinMode(i,OUTPUT);
  }
}

void loop(){
  PORTD = 0;//send (0/255)*5 = 0V out DAC
  delay(1);//wait 1ms
  PORTD = 127;//send (127/255)*5 = 2.5V out DAC
  delay(2);//wait 2ms
  PORTD = 51;//send (51/255)*5 = 1V out DAC
  delay(1);//wait 1ms
  PORTD = 255;//send (255/255)*5 = 5V out DAC
  delay(3);//wait 3ms
}
```

The code below outputs a ramp from 0 to 5V. In the loop() function, the variable "a" is incremented from 0 to 255. Each time it is incremented, the value of "a" is sent to PORTD. This value is held for 50us before a new value of "a" is sent. Once "a" reaches 255, it gets reset back to 0. The time for each cycle of this ramp (also called the period) takes:

period = (duration of each step) * (number of steps)
period = 50us * 256 = 12800us = 0.0128s

so the frequency is:
frequency of ramp = 1/0.0128s = 78Hz

The output from the DAC on an oscilloscope can be seen in fig 5.

```
//Ramp out
//by Amanda Ghassaei
//https://www.instructables.com/id/Arduino-Audio-Output/
//Sept 2012

/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
*/

void setup(){
  //set digital pins 0-7 as outputs
  for (int i=0;i<8;i++){
    pinMode(i,OUTPUT);
  }
}
```

The code below outputs a sine wave centered around 2.5V, oscillating up to a max of 5V and a min of 0V. In the loop() function, the variable "t" is incremented from 0 to 100. Each time it is incremented, the expression:
127+127*sin(2*3.14*t/100)
is sent to PORTD. This value is held for 50us before "t" is incremented again and a new value is sent

out to PORTD. Once "t" reaches 100, it gets reset back to 0. The period of this sine wave should be:

period = (duration of each step) * (number of steps)
period = 50us * 100 = 5000us = 0.005s

so the frequency should be:
frequency of ramp = 1/0.005s = 200Hz

```
//Sine out
//by Amanda Ghassaei
//https://www.instructables.com/id/Arduino-Audio-Output/
//Sept 2012

/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
*/

void setup(){
  //set digital pins 0-7 as outputs
  for (int i=0;i<8;i++){
    pinMode(i,OUTPUT);
  }
}
```

But this is not the case, the output from the DAC is shown in fig 6. As indicated in the image notes, it does not have a frequency of 200hz, its frequency is more like 45hz. This is because the line:

PORTD = 127+127*sin(2*3.14*t/100);
takes a very long time to calculate. In general multiplication/division with decimal numbers and the sin() function take the Arduino a lot of time to perform.

One solution is to calculate the values of sine ahead of time and store them in the Arduino's memory. Then when the Arduino sketch is running all the Arduino will have to do is recall these values from memory (a very easy and quick task for the Arduino). I ran a simple Python script (below) to generate 100 values of 127+127*sin(2*3.14*t/100):

```
import math
for x in range(0, 100):
print str(int(127+127*math.sin(2*math.pi*x*0.01)),)+str(","),
```

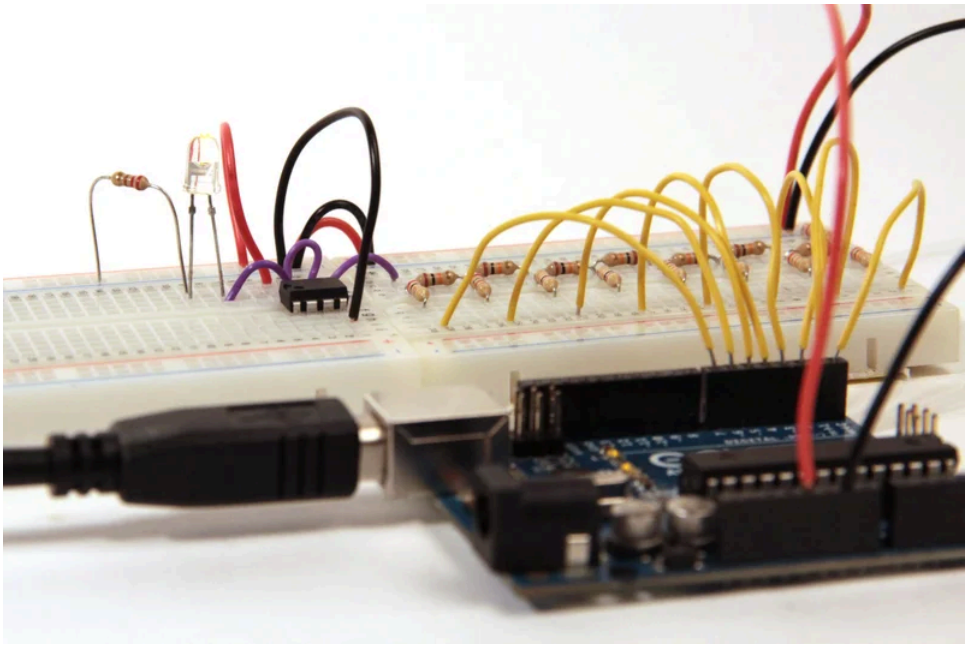
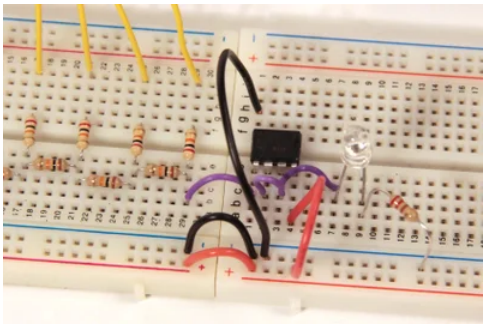
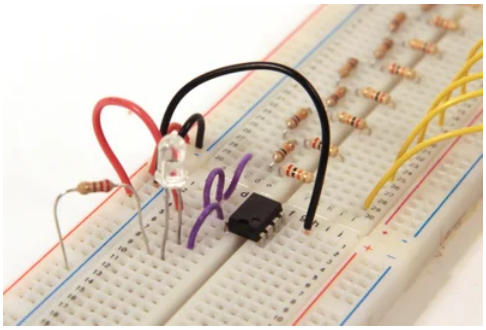
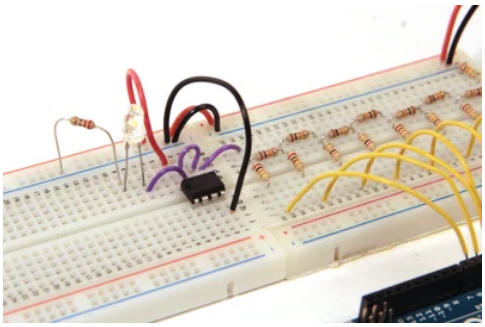
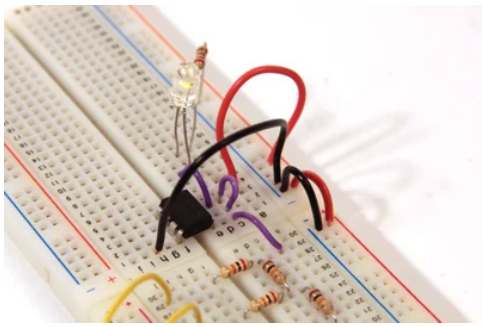
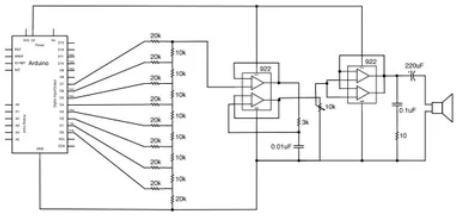
I stored these values in an array called "sine" in the Arduino sketch below. Then in my loop, for each value of "t" I sent an element of sine[] to PORTD:

PORTD = sine[t];

The output from this DAC for this sketch is shown in fig 7. You can see that it outputs a sine wave of 200hz, as expected.

```
//Sine out with stored array
//by Amanda Ghassaei
//https://www.instructables.com/id/Arduino-Audio-Output/
//Sept 2012

/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
*/

byte sine[] = {127, 134, 142, 150, 158, 166, 173, 181, 188, 195, 201, 207, 213,
219, 224, 229, 234, 238, 241, 245, 247, 250, 251, 252, 253, 254, 253, 252, 251,
250, 247, 245, 241, 238, 234, 229, 224, 219, 213, 207, 201, 195, 188, 181, 173,
166, 158, 150, 142, 134, 127, 119, 111, 103, 95, 87, 80, 72, 65, 58, 52, 46,
40, 34, 29, 24, 19, 15, 12, 8, 6, 3, 2, 1, 0, 0, 0, 1, 2, 3, 6, 8, 12, 15, 19,
24, 29, 34, 40, 46, 52, 58, 65, 72, 80, 87, 95, 103, 111, 119,};
```

Step 3: DAC Buffer

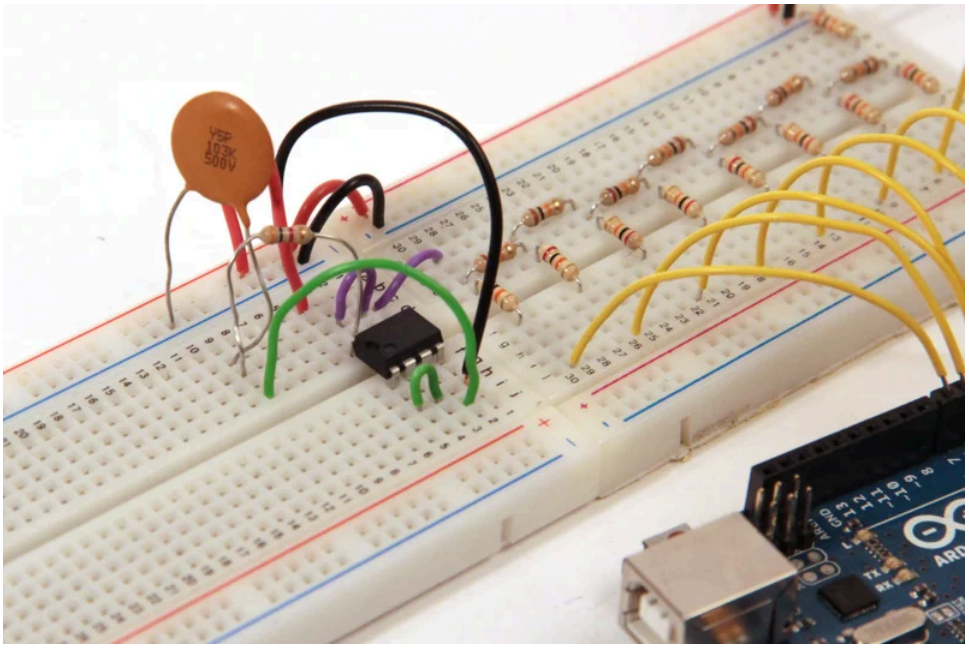
Now that we have a good signal coming out Arduino, we need to protect it. The R2R DAC is very sensitive to any loads put on it, so trying to drive speakers directly from the DAC will distort the signal heavily. Before doing anything with the signal you need to set up some kind of buffer circuit. I set up one of the op amps in the TS922 dual op amp package as a voltage follower to buffer my DAC from the rest of my circuit (see schematic in fig 6, be sure to power the op amp with 5V and ground).

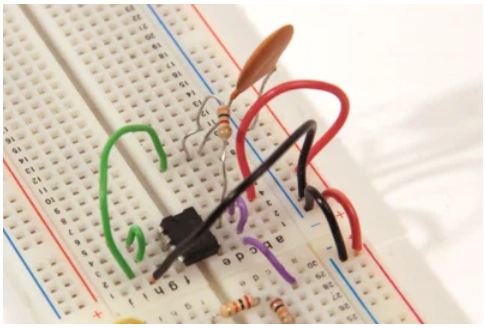
Once this was set up I wired an LED and 220ohm resistor in series between the output of the op amp and ground. The sketch below outputs a slow ramp out the DAC so you can actually see the LED get brighter as the ramp increases in voltage. The period of the ramp is:

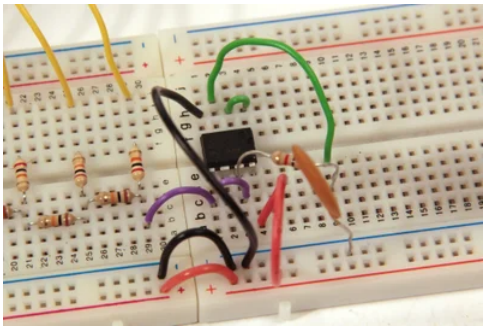
period = (duration of each step) * (number of steps)
period = 5ms * 256 = 1280ms = 1.28s
so the LED takes 1.28 seconds to ramp up from off to full brightness.

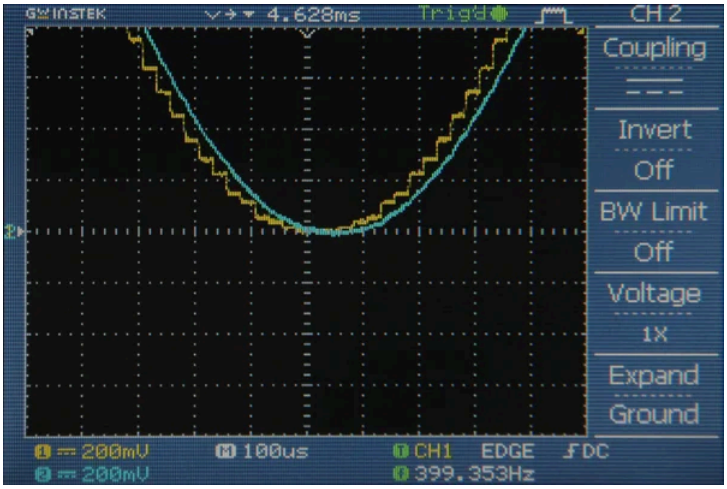
```
//Slow Ramp
//by Amanda Ghassaei
//https://www.instructables.com/id/Arduino-Audio-Output/
//Sept 2012

/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
*/

void setup(){
  //set digital pins 0-7 as outputs
  for (int i=0;i<8;i++){
    pinMode(i,OUTPUT);
  }
}
```

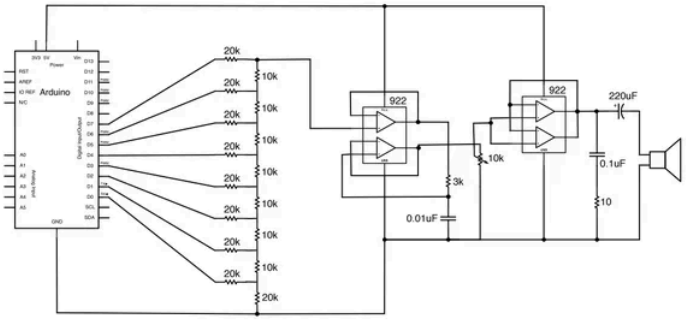
Step 4: Low Pass Filter











The purpose of a low pass filter is to smooth out the output of the DAC in order to reduce noise. By using a low pass filter on the signal, you can smooth out the "steps" in your waveform while keeping the overall shape of the waveform intact (see fig 4). I used a simple RC flow pass filter to achieve this: a resistor and a capacitor in series to ground. Connect the resistor to the incoming signal and the capacitor to ground, the signal coming from the junction between these two components will be low pass filtered. I sent this filtered signal into another buffer circuit (I wired an op amp in a voltage follower configuration) to protect the filtered signal from any loads further down in the circuit. See the schematic in fig 5 for more info.

You can calculate the values of the capacitor and resistor you need for a low pass filter according to the following equation:

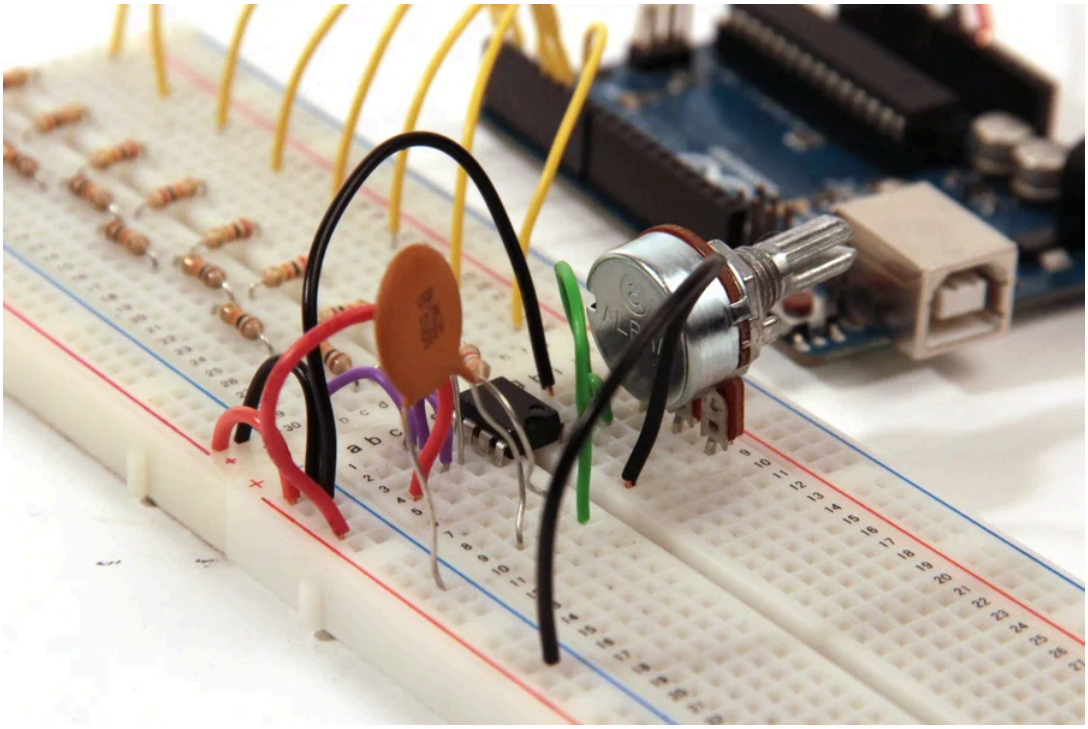
cutoff frequency = 1/ (2*pi*R*C)

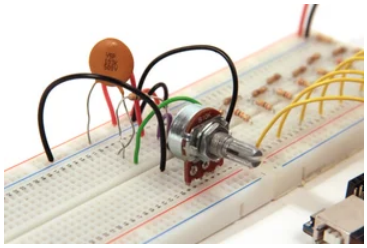
Nyquist's Theroum states that for a signal with a sampling rate of x Hz, the highest frequency that can be produced is x/2 Hz. You should set your cutoff frequency to x/2Hz (or maybe slightly lower depending on what you like). So if you have a sampling rate of 40kHz (standard for most audio), then the maximum frequency you can reproduce is 20kHz (the upper limit of the audible spectrum), and the cutoff frequency of your low pass filter should be around 20kHz.

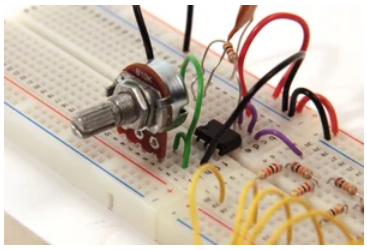
For a cutoff frequency of 20,000Hz and 1kOhm resistor:
20000=1/(2*3.14*1000*C)
C =~ 8nF

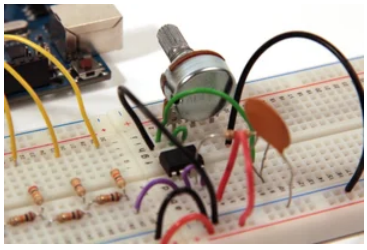
since 8nF capacitors are hard to come by I rounded up to 0.01uF. This gives a cutoff frequency of about 16kHz. You can mess around with different values and see what you like best, I tend to like heavier filtering because it removes more unwanted noise.

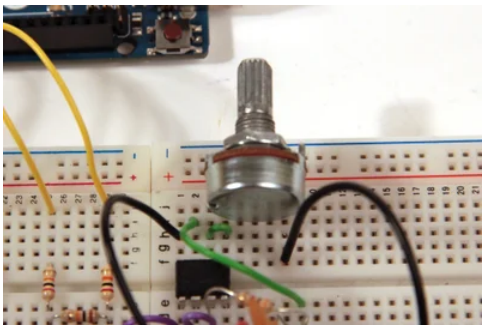
Step 5: Signal Amplitude

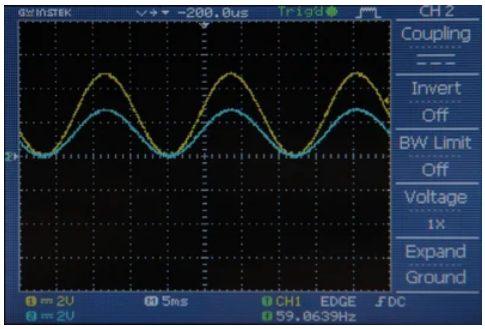


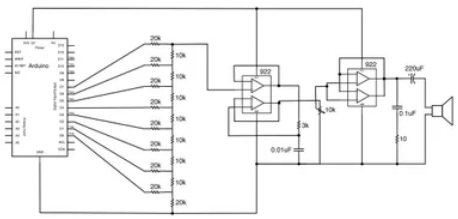






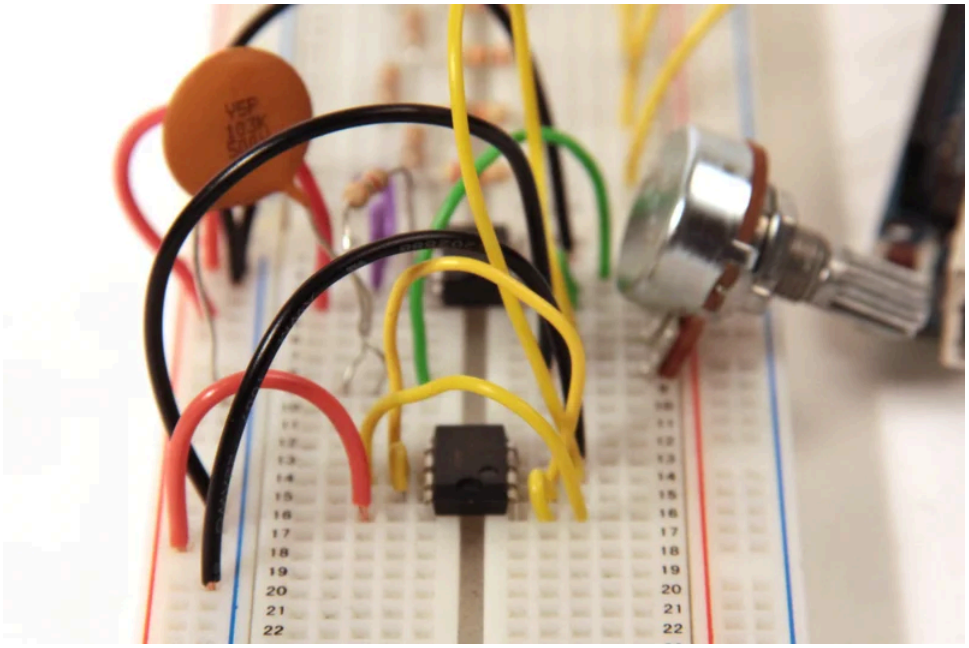


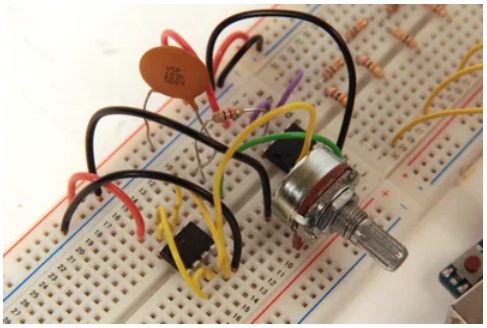


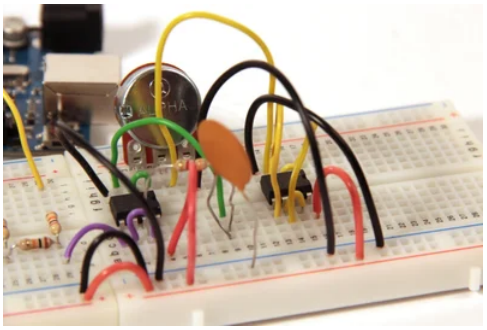


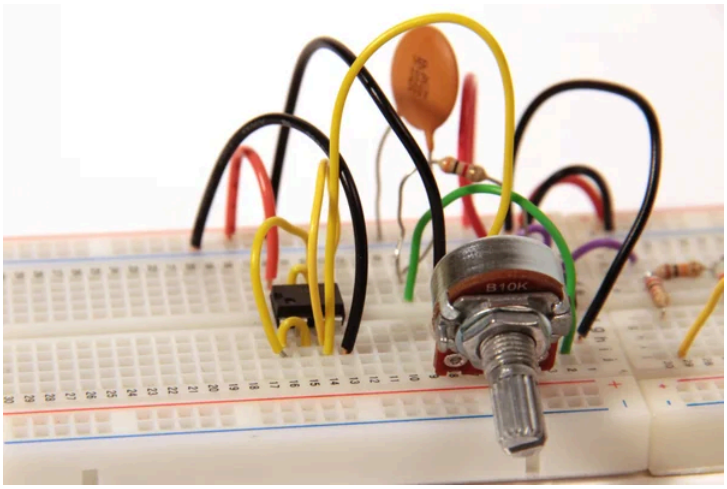
Next I added a potentiometer to control the amplitude of my signal. To do this I wired the output from the 2nd voltage follower to one side of a 10k potentiometer. The I wired the other side of the pot to ground. The signal coming out from the middle of the pot has an adjustable amplitude (between 0 and 2.5V) depending on where the pot is turned. See the schematic (fig 7) for more info. You can see the output of the signal before the pot and after the pot (when turned to halfway point) in fig 6.

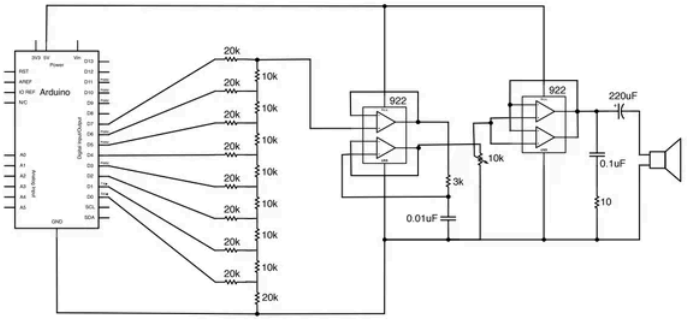
Step 6: Amplifier





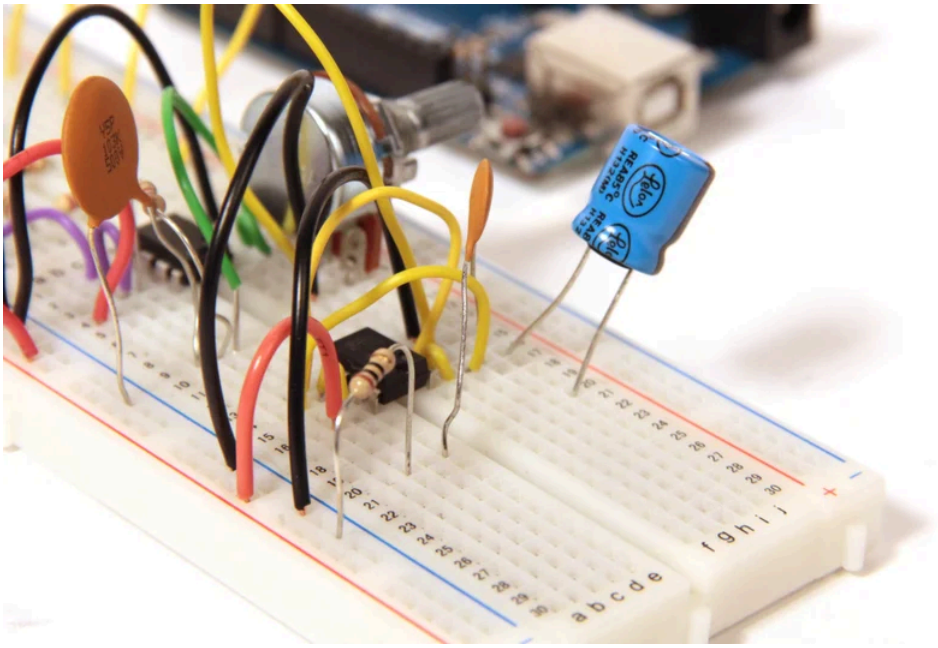


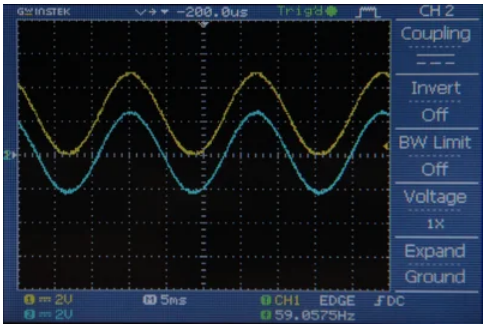


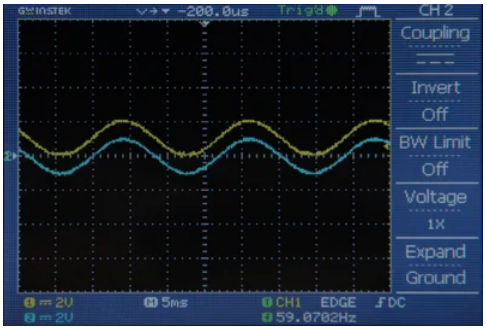


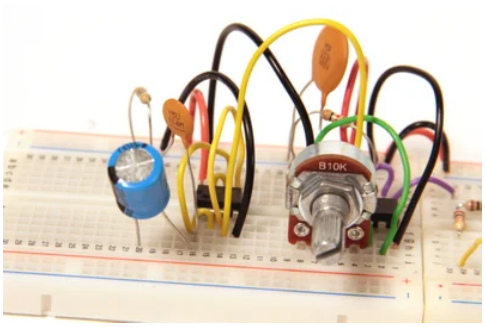
Many times when we talk about amplifiers we think about circuits which increase the amplitude of a signal. In this case I'm talking about increasing the current of the signal so that it can drive a load (like a speaker). In this stage of the circuit I set up both op amps on one TS922 package as parallel voltage followers. What this means is I sent the output from the amplitude pot to the non-inverting input of both op amps. Then I wired both op amps as voltage followers and connected their outputs to each other. Since each op amp can source 80mA of current, combined they can source 160mA of current.

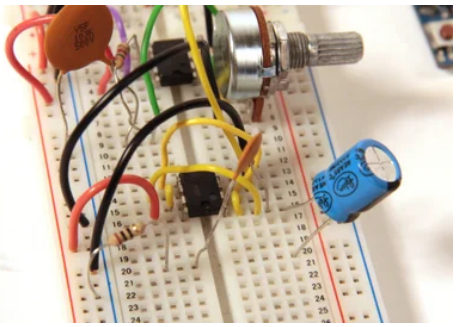
Step 7: DC Offset

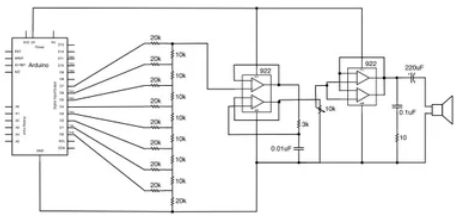






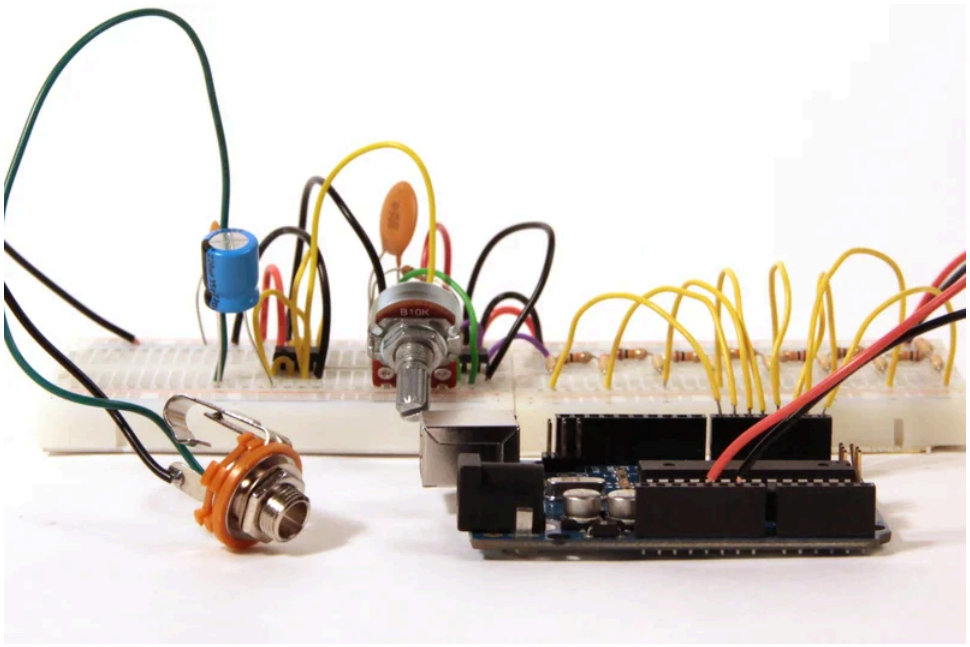


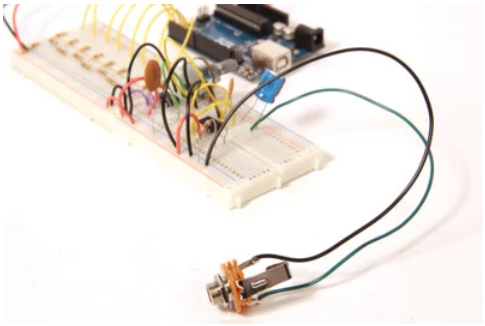


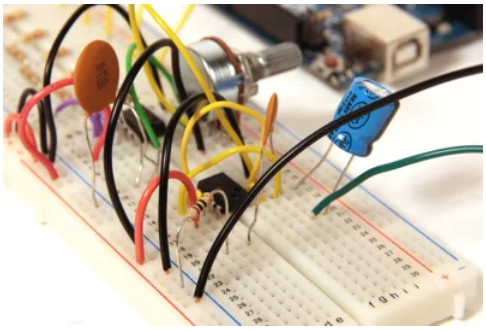


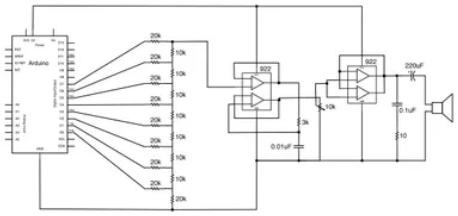
Before sending a signal to speakers, you want to make sure it is oscillating around 0V (typical of audio signals). So far, the Arduino DAC output we've been dealing with is oscillating around 2.5V. To fix this we can use a big capacitor. As indicated in the schematic, I used a 220uF capacitor to DC offset my signal so that it oscillates around 0V. The output of the DC offset signal (blue) and un-offset signal (yellow) for two different amplitudes can be found in figs 2 and 3.

Step 8: Output



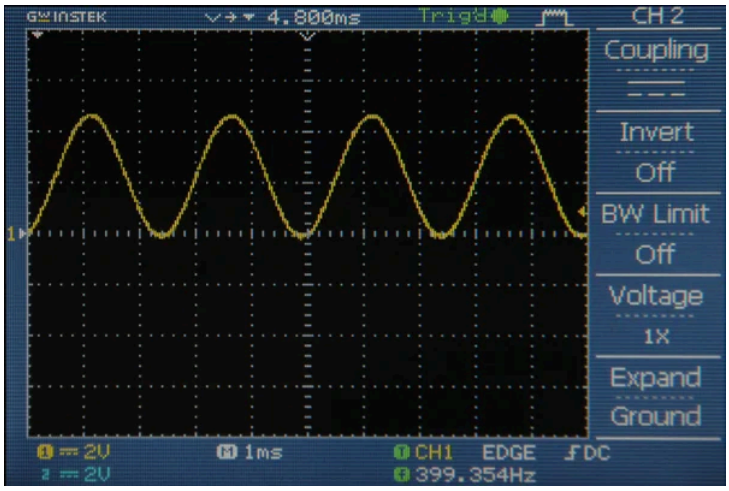


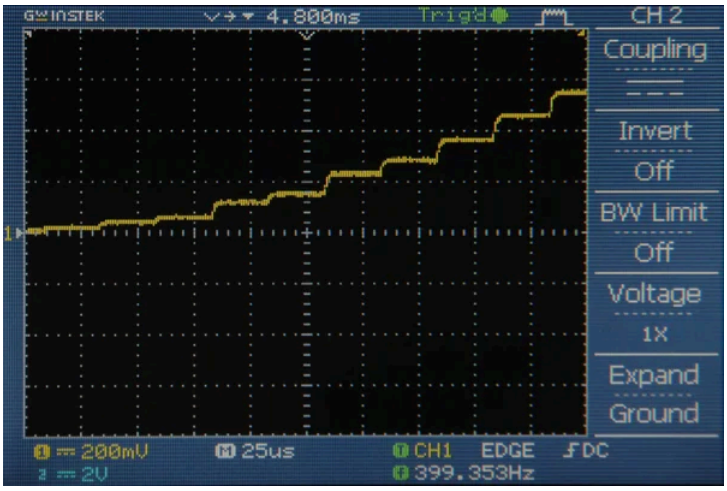




Finally, I wired up a 1/4" mono jack with two wires. I connected the ground lead to the Arduino's ground and the signal lead to the negative lead of the 220uF capacitor. The ground pin is usually the larger pin on the jack, test for continuity with the threaded portion of the jack to make sure that you have located the ground pin correctly (see fig 5). The signal pin will be continuous with the clip that extends out from the jack (fig 5). See the schematic for more info.

Step 9: 40kHz Sampling Rate





For those of you who are interested in producing audio at 40kHz sampling rate, here is some code that uses timer interrupts to let you do that. Arduino timer interrupts allow you to pause what you are doing in your main loop() function and jump to a special function called an "interrupt routine." Once this routine is done you come back to where you left off in the loop(). You set up and specify the frequency of these interrupts in the setup() part of your code. You can learn the specifics of setting up interrupts here, but if you are only interested in 40kHz interrupts, then you can just copy parts of the code below.

To set up the interrupt you need to copy the following lines into your setup() function:

```
cli();//disable interrupts
//set timer0 interrupt at 40kHz
TCCR0A = 0;// set entire TCCR0A register to 0
TCCR0B = 0;// same for TCCR0B
TCNT0  = 0;//initialize counter value to 0
// set compare match register for 40khz increments
OCR0A = 49;// = (16*10^6) / (40000*8) - 1 (must be <256)
// turn on CTC mode
TCCR0A |= (1 << WGM01);
// Set CS11 bit for 8 prescaler
TCCR0B |= (1 << CS11);
// enable timer compare interrupt
TIMSK0 |= (1 << OCIE0A);
sei();//enable interrupts
```

the contents of the interrupt routine are encapsulated in the following function:

```
ISR(TIMER0_COMPA_vect){ //40kHz interrupt routine
}
```

You want to keep the interrupt routine as short as possible, only the necessities. You can do all of your other tasks (checking on buttons, turning on leds, etc) in the loop(). Also keep in mind that setting up interrupts may affect other Arduino functions such as analogWrite and delay.

In the code below, I use the interrupt function to send a new value of sine[] to PORTD at a rate of 40kHz and increment the variable "t." Figs 1 and 2 show the (unfiltered) output of the code on an oscilloscope. We can calculate the expected frequency as follows:

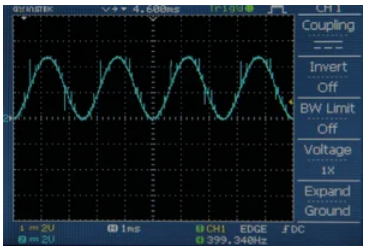
frequency = (sampling frequency) / (steps per cycle)
frequency = 40,000 / 100 = 400hz

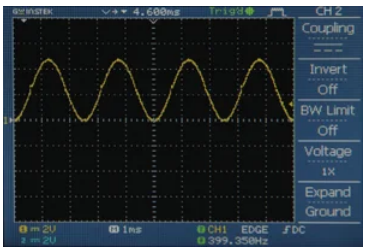
at a sampling frequency of 40kHz we expect the duration of each step to be:

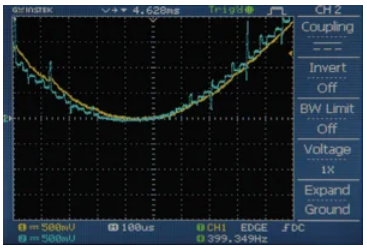
duration of each sample step = 1/(sampling frequency)
duration of each sample step = 1/40,000 = 25us

```
//Sine out w/ 40kHz sampling rate
//by Amanda Ghassaei
//https://www.instructables.com/id/Arduino-Audio-Output/
//Sept 2012

/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
*/

byte sine[] = {127, 134, 142, 150, 158, 166, 173, 181, 188, 195, 201, 207, 213,
219, 224, 229, 234, 238, 241, 245, 247, 250, 251, 252, 253, 254, 253, 252, 251,
250, 247, 245, 241, 238, 234, 229, 224, 219, 213, 207, 201, 195, 188, 181, 173,
166, 158, 150, 142, 134, 127, 119, 111, 103, 95, 87, 80, 72, 65, 58, 52, 46,
40, 34, 29, 24, 19, 15, 12, 8, 6, 3, 2, 1, 0, 0, 0, 1, 2, 3, 6, 8, 12, 15, 19,
24, 29, 34, 40, 46, 52, 58, 65, 72, 80, 87, 95, 103, 111, 119,};
int t = 0;//time
```

Step 10: Extra Tips







This DAC uses quite a bit of the Arduino's available digital pins, including some that are normally used for serial communications and PWM, so here are a few tips that will help you deal with pin conflicts.

If you want to do serial communication: Software Serial is an Arduino library that allows you to turn any of the Arduino's pins into serial pins. Usually when you are doing an Arduino project that requires serial communication, you avoid using digital pins 0 and 1 because they need to be free to send serial data. I like to use them for the 8 bit DAC because pins 0-7 are all part of PORTD on the Arduino's Atmel328 chip, this allows me to address all of them in a single line of code. PORTB only has 6 pins (digital pins 8-13) and PORTC only has 6 pins (analog pins 0-5), so you cannot construct an 8 bit DAC with these ports alone.

If you need to use the PWM pins, or otherwise need to use different pins as the DAC: If you must use the PWM pins you can use bit manipulation to free up pins 3, 5, and 6 and replace them with pins 8, 12, and 13. Say you want to send the number 36 to PORTD. You can use the following lines:

```
//define variables:
boolean bit3state;
boolean bit5state;
boolean bit6state;

//in your main loop():

bit3state = (36 & B00001000)>>3;//get the third bit of 36
bit5state = (36 & B00100000)>>5;//get the fifth bit of 36
bit6state = (36 & B01000000)>>6;//get the sixth bit of 36

//send data to portd w/o disrupting pins 3, 5, and 6
PORTD |= (36&B10010111);//set high pins high using the number 36 with zeros replacing bits 3, 5, and 6
PORTD &= (36|B01101000);//set low pins low using the number 36 with ones replacing bits 3, 5, and 6
```

//send data to portb w/o disrupting pins 9, 10, and 11
PORTB |= 0 | (bit3state) | (bit5state<<4) | (bit6state<<5);//set high pins
PORTB &= 255 & ~(1-bit3state) & ~((1-bit5state)<<4) & ~((1-bit6state)<<5);//set low pins

be sure to keep these PORTD and PORTB lines right next to each other in your code, you want the pins on port d and port b to switch at as close to the same time as possible.

Here is the code from the previous step, edited so that it does not use any PWM pins. As you see in fig 1, the unfiltered output from the DAC has many discontinuities caused by the lag between sending data to port d and port b, as well as splitting up the commands for setting pins high and low. You can get rid of most of these discontinuities with the low pass filter (fig 2). If you wanted to use this technique you might consider increasing the cutoff frequency of your low pass filter. If you wanted to make this really good, you could send your 5 most significant bits to port d and your 3 least significant bits to port b. This would decrease the amplitude of some of the discontinuities, reducing the magnitude of the noise. I'll let you figure that one out on your own.

```
//Sine out, 40kHz sampling rate, w/o using PWM pins
//by Amanda Ghassaei
//https://www.instructables.com/id/Arduino-Audio-Output/
//Sept 2012

/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
*/

byte sine[] = {127, 134, 142, 150, 158, 166, 173, 181, 188, 195, 201, 207, 213,
219, 224, 229, 234, 238, 241, 245, 247, 250, 251, 252, 253, 254, 253, 252, 251,
250, 247, 245, 241, 238, 234, 229, 224, 219, 213, 207, 201, 195, 188, 181, 173,
166, 158, 150, 142, 134, 127, 119, 111, 103, 95, 87, 80, 72, 65, 58, 52, 46,
40, 34, 29, 24, 19, 15, 12, 8, 6, 3, 2, 1, 0, 0, 0, 1, 2, 3, 6, 8, 12, 15, 19,
24, 29, 34, 40, 46, 52, 58, 65, 72, 80, 87, 95, 103, 111, 119,};
int t = 0;//time
```

If you run out of digital pins and need more: Remember you can always use your analog pins as Digital I/O. Try out the following functions, they work just like you are dealing with a regular digital pin.

digitalWrite(A0,HIGH);//set pin A0 high
digitalWrite(A0,LOW);//set pin A0 low
digitalRead(A0);//read digital data from pin A0

Otherwise, try using a multiplexer. If you need more digital outputs, the 74HC595 allows you to turn three of the Arduino's digital pins into 8 outputs. You can even daisy chain multiple 595's together to create many more outputs pins. You could set up your whole DAC on one of these chips if you wanted (though it would take a few lines of code to address it and might slow you down too much for higher sampling rates). The Arduino website is a good place to start learning about how to use the 595.

If you need more digital inputs, the 74HC165 or CD4021B let you turn three of the Arduino's digital pins into 8 inputs. Again, the Arduino website is a good place to start learning how to use these chips.

If you want to use the info in this Instructable with the Mega or other boards: In this Instructable I talked exclusively about the Arduino Uno with Atmel328. The same code will run fine on any board with an Atmel328 or Atmel168 chip on it. You can also use the same ideas with a Mega. You should

try to attach your DAC to any port that has 8 available pins, that way you can address your DAC with one line of code ("PORTD =") On the Uno, the only port that has 8 available pins is port d. This picture indicates that the Mega has several ports with 8 pins: ports a, b, c, and l are the obvious choices. If you don't care about wasting analog pins you could also use ports f or k.