Advanced Microcontroller Audio Workshop

This workshop is intended to introduce you to the Teensy Audio Library and its audio system design tool. If you're accustomed to 8 bit microcontrollers, you will be pleasantly surprised how much capability a modern 32 bit processor provides.

For the latest copy of this document, parts list and more details, please visit:
https://hackaday.io/project/8292-microcontroller-audio-workshop-had-supercon-2015

Section 1: Install Software and Play Music

In this first section, you set up software on your computer, make Teensy play music, and make Teensy perform other tasks while the music is playing.

Part 1-1: Install Arduino & Teensyduino 1.26

Your workshop kit includes a SD card reader and a Micro SD card with all the required files.

Step #1: Plug the card into your laptop and copy all of its files to a folder on your hard drive.

Step #2: Verify or Install Arduino & Teensyduino. This workshop requires Arduino 1.6.5 or 1.6.6 and Teensyduino 1.26.
To verify which versions are installed:
If using Windows or Linux, click **Help > About**
If using Macintosh, click **Arduino > About**

If your Arduino and Teensyduino versions are correct, **skip to Step #3**.

If your copy of Arduino is not one of these versions, you need to install Arduino and Teensyduino. Simply extract the .ZIP or .TGZ file. *Remember the location on your computer where it is extracted.*

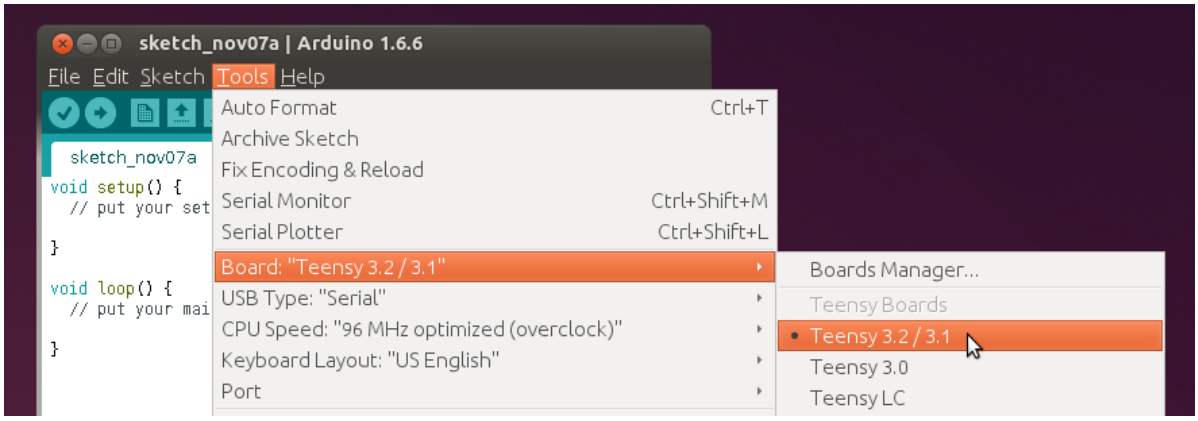
Windows: If Arduino is installed in C:\Program Files (x86), you may need to **right-click** the Teensyduino installer and select **Run as administrator**.
Macintosh: For Mavricks or later, run Arduino at least once before installing Teensyduino.
Linux: The **49-teensy.rules** file needs to be copied to **/etc/udev/rules.d**

Quit Arduino if it's still running. Then run the Teensyduino installer. It will ask for Arduino's location.

We will be using the **Audio, SerialFlash** and **ILI9341_t3** libraries in this workshop. You must keep these selected at the step to install additional libraries.

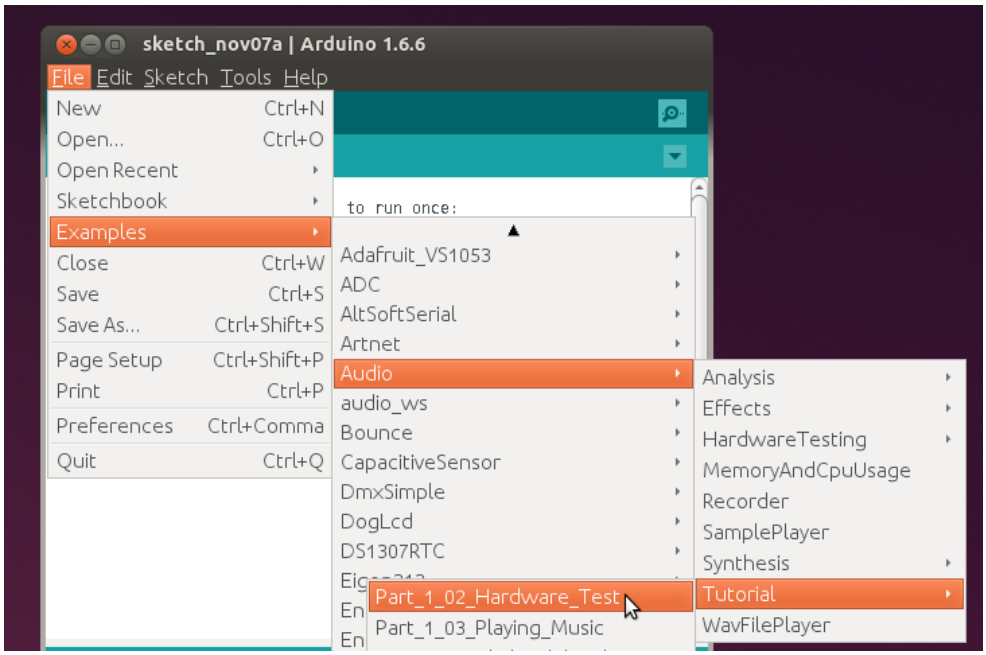
Continue the install until completed.

Step #3: Configure Arduino for Teensy 3.2. In Arduino select **Tools > Boards > Teensy 3.2/3.1**

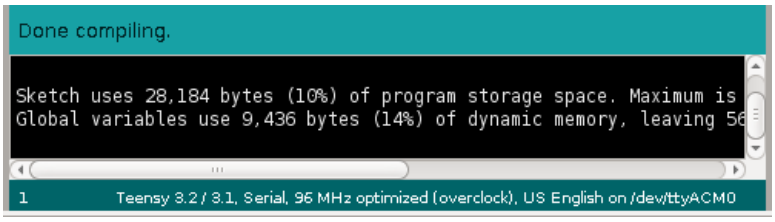


Step #4: Verify an example to make sure Arduino can compile the workshop files.

Open the example file, from **File > Examples > Audio > Tutorial > Part_1_02_Hardware_Test**



Click the **Verify** button. If Arduino and Teensyduino and libraries are installed properly, after several seconds the you should see this message.

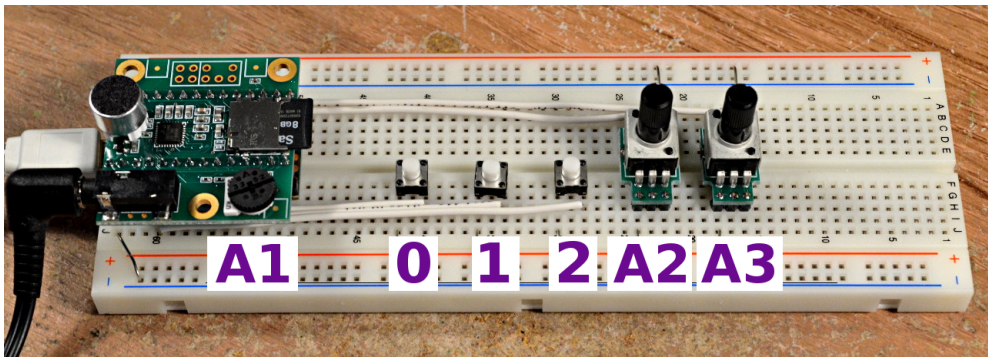


This workshop uses the examples from **File > Examples > Audio > Tutorial**

Part 1-2: Test Hardware

Your workshop package contains a breadboard with the Teensy, Audio Shield, and push buttons and pots already assembled.

Step #1: Gently plug in the USB cable and headphones
Step #2: Plug the USB cable into your computer

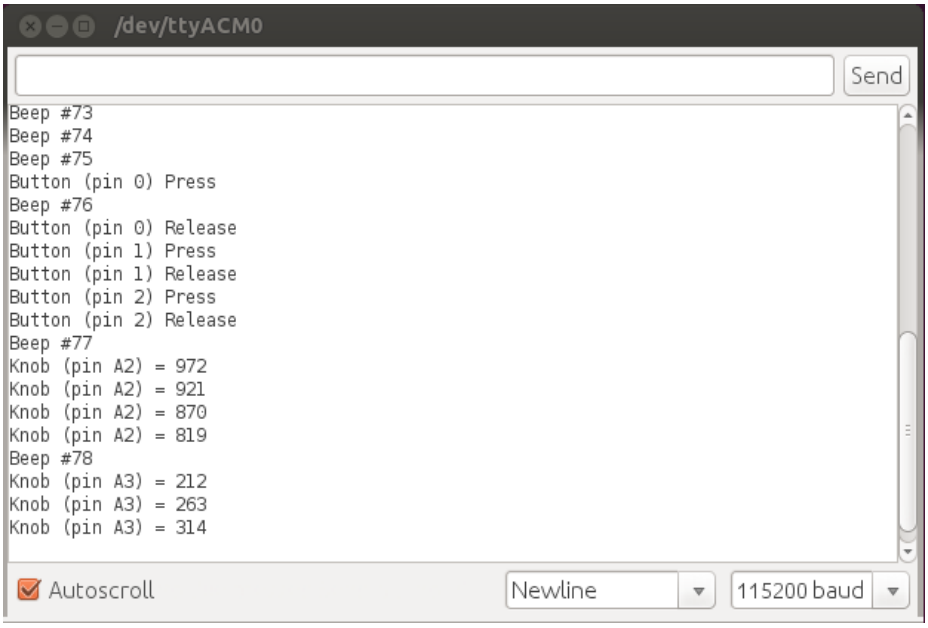


You should immediately hear a regular beep on your headphones, once every 2 seconds.

If Windows shows detecting new hardware, DO NOT cancel. Allow it to complete!

Step #3: Use Arduino's **Tools > Ports** to select the Teensy serial port. If you are unsure which is Teensy, you may unplug the USB cable (but not while Windows is detecting new hardware!) and try the **Tools > Ports** menu again to see which port vanished.

Step #4: With the proper port selected, open the Arduino Serial Monitor, using the **Tools > Serial Monitor** menu.



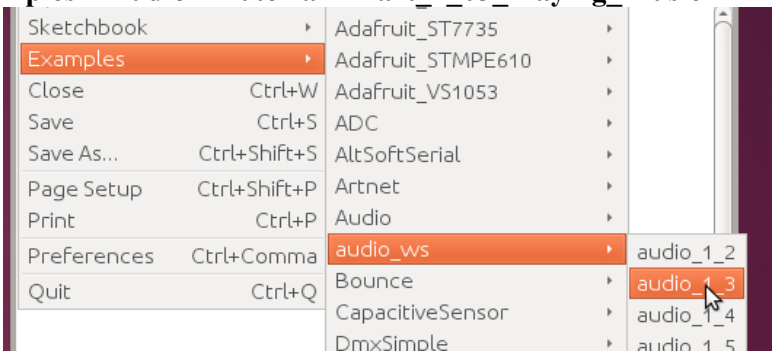
You should see a Beep message for each headphone beep. Pressing the buttons and turn the 2 pots on the breadboard. All should respond. You will need these, so it's important to verify they work before proceeding.

Part 1-3: First Program, Play Music

With everything installed, you're ready to replace that beeping with music.

Step #1: Open the example sketch for Part 1-3.

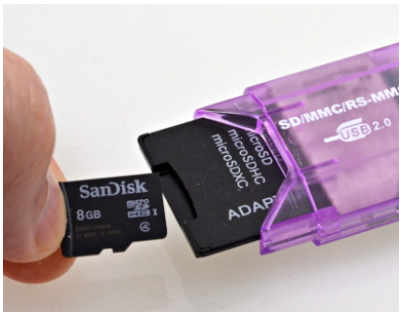
File > Examples > Audio > Tutorial > Part_1_03_Playing_Music

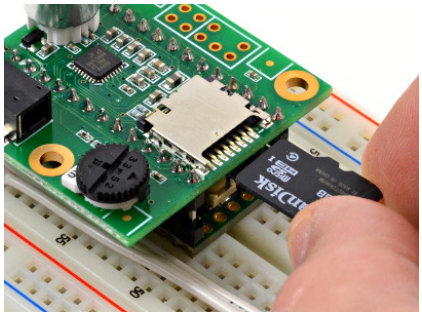


Errata: this image is incorrect. The text above is correct.

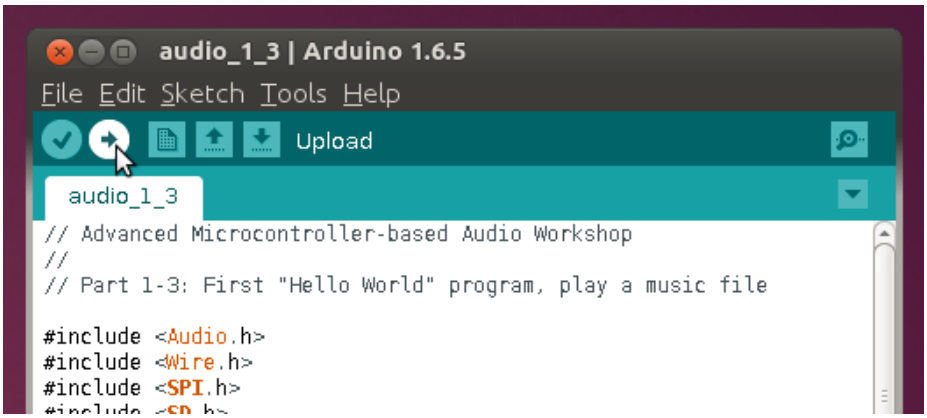
Step #2: You should click **Verify** now, *but do not Upload yet*. Arduino can take quite a while to compile the entire audio library, so start the verify step now.

Step #3: Insert the Micro SD card from the card reader and into the Teensy Audio Shield *before you Upload the program.* The music files are stored on this MicroSD card.





Step #4: When the Micro SD card is in place, click the **Upload** button.



You should now be hearing music rather than simple beeping.

Part 1-4: Blink LED while Playing Music

Real projects need to do more than just play music. Great sound isn't so great if it imposes difficult requirements on the rest of your program. Let's try doing some simple things with the audio plays.

Step #1: Open the Part 1-4 example
File > Examples > Audio > Tutorial > Part_1_04_Blink_While_Playing

Step #2: **Verify** and **Upload** this sketch.

> ***Special Tip:***
> If **Verify** or **Upload** is very slow to compile on your laptop, you can use a little trick to speed things up. If you have a previously used window, delete all its code (**CTRL-A** and **Del**), then copy and paste a new program into the old window. Close the new window. When you **Verify** or **Upload**, Arduino will reuse all the compile work previously done for the old window, which dramatically speeds things up!

When you run this example, you'll see the messages about playing appear in the Arduino Serial Monitor. If you look between the Audio Shield and Teensy, you can see the orange LED blinking.

The key point is your Arduino code can use delay(), Serial.print() and other common Arduino features. The audio keeps playing during those delays.

Step #3 (optional): Uncomment the volume knob code near the end of this sketch and **Upload**. When you turn the knob, there's a lag before the volume changes. In Part 1-5 we will look at a way to prevent this problem.

Step #4 (optional): What happens if you reduce the delays? Or if you completely remove them? Or print even more text? Can you get extremely fast transmission to the Arduino Serial Monitor to make the audio stutter?

Section 1 – Achievement Unlocked

Congratulations! You've completed the essential material of Part 1. If you have time, keep exploring more of Section 1, or skip ahead to start Section 2.

Part 1-5 : Do More While Playing Music

In the previous part, we saw you can use delay() without disrupting the audio library, but it does prevent your program from rapidly responding to user actions. In this part, we'll blink the LED without using delay, so our sketch remains responsive. We'll also look in more detail at reading the knobs and buttons to do more things while the music plays, and mention some of the finer details of the Arduino sketch example code.

Step #1: Open the Part 1-5 example
File > Examples > Audio > Tutorial > Part_1_05_Do_More_While_Playing

Step #2: **Verify** and **Upload** this sketch.

As this sketch runs, you should see the LED blink. The knob controls the volume immediately, regardless of where the LED is within its blinking cycle. The left and right pushbuttons allow you to cycle through the 4 songs.

The blinking without delay uses a special elapsedMillis variable, which acts like a normal 32 bit integer, but it automatically increments 1000 times per second.

```
elapsedMillis blinkTime;

void loop() {

  if (playSdWav1.isPlaying() == false) {
```

This makes blinking without delay very simple. If the “blinkTime” variable is still under 250, the LED should be off. If it's incremented to between 250 to 500, the LED needs to be on. If it's gone past 500, simply set it back to zero to restart the blink cycle.

```
// blink the LED without delays
if (blinkTime < 250) {
  digitalWrite(13, LOW);
} else if (blinkTime < 500) {
  digitalWrite(13, HIGH);
} else {
  blinkTime = 0; // start blink cycle over again
}
```

The pushbuttons are read using the Bounce library. While you could use Arduino's digitalRead() function to directly read the pins, Bounce automatically handles mechanical chatter in the button, and it does this without requiring a delay which could slow our response to other user input.

The Bounce library is easy to use, but it does require a few small pieces of code. First, a Bounce object is created for each button. The digital pin number and worst-case mechanical chatter time are given.

```
Bounce button0 = Bounce(0, 15);
Bounce button2 = Bounce(2, 15);  // 15 = 15 ms debounce time
```

Inside setup(), the digital pins need to be configured with INPUT_PULLUP mode. The buttons connect the pin to ground, so the pullup causes the pin to be high when the button isn't pressed.

```
  }
  pinMode(13, OUTPUT); // LED on pin 13
  pinMode(0, INPUT_PULLUP);
  pinMode(2, INPUT_PULLUP);
  delay(1000);
}
```

Inside loop(), the buttons are checked using update() and fallingEdge(). Because the button connects to ground, the moment the button is pressed is called the falling edge. The Bounce library allows us to reliably detect the falling and rising edge, even if the button has chatter, and without ever delaying our sketch.

```
// read pushbuttons
button0.update();
if (button0.fallingEdge()) {
  playSdWav1.stop();
}
button2.update();
if (button2.fallingEdge()) {
  playSdWav1.stop();
  filenumber = filenumber - 2;
  if (filenumber < 0) filenumber = filenumber + 4;
}
```

All the examples in this tutorial use Bounce to read the pushbuttons.

This final piece of code, which we'll see repeated throughout this workshop, reads one of the knobs. The Arduino analogRead() function gives an integer between 0 to 1023.

```
  // read the knob position (analog input A2)
  int knob = analogRead(A2);
  float vol = (float)knob / 1280.0;
  sgtl5000_1.volume(vol);
  //Serial.print("volume = ");
  //Serial.println(vol);
}
```

Most audio library functions need a floating point number between 0 to 1.0, so throughout this workshop you will see equations which convert the 0-1023 integer. In this case, volume more than 0.8 is much too loud. Each example varies slightly, but these basic concepts are used throughout the example code in this workshop.

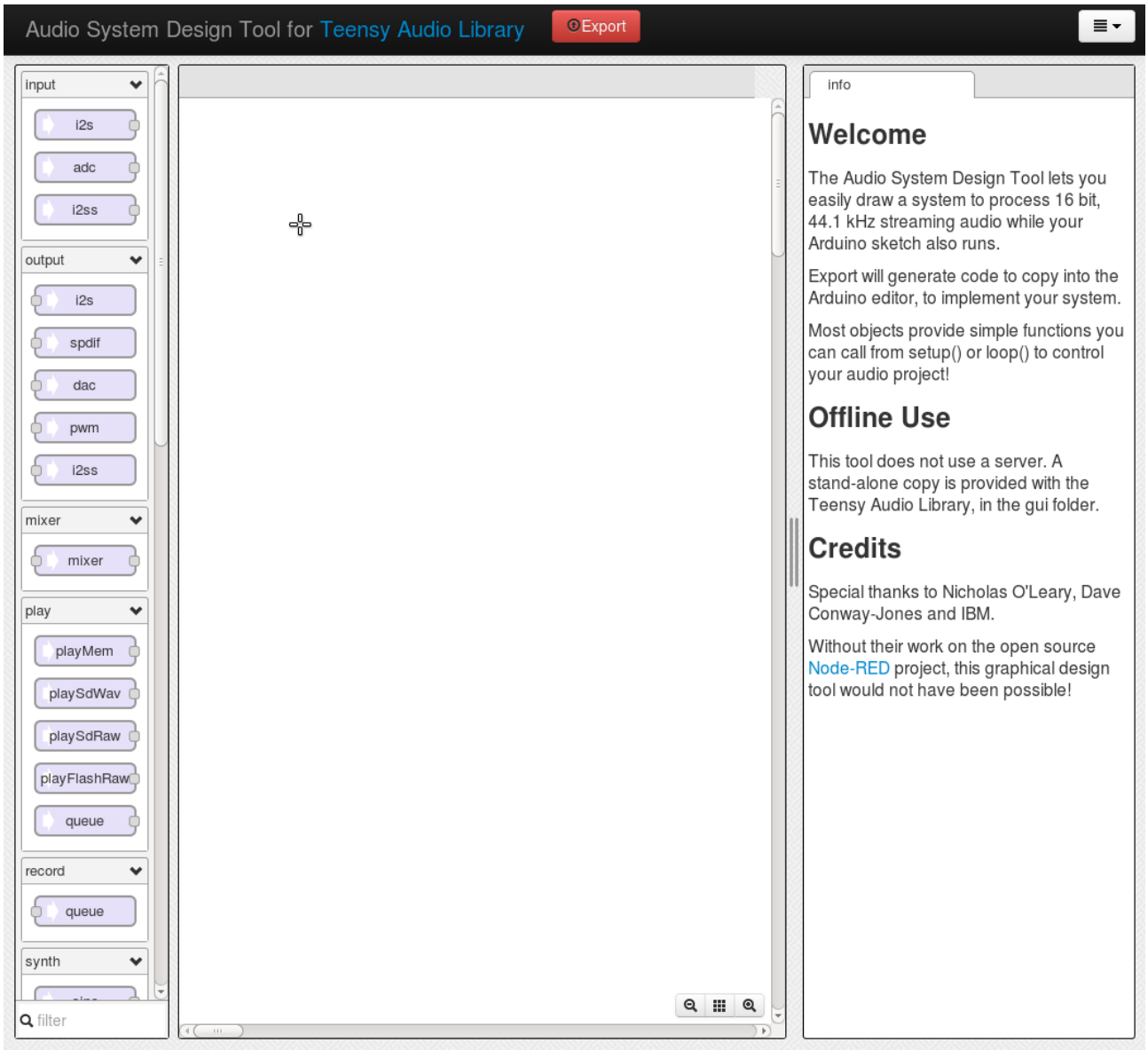
Section 2: Creating Audio Systems

The previous examples had collections of objects to play music. In this section, we'll explore how to use the design tool to create our own audio systems.

Part 2-1: First Design Tool Use

Step #1 (with Internet connection):
To run the design tool, open a web browser and go to http://www.pjrc.com/teensy/gui

When the design tool first appears, you'll see a blank canvas in the center panel. The left side is a list of all the audio objects grouped into functional categories, and the right side shows documentation.



Step #1 (without Internet access, using Windows or Linux):

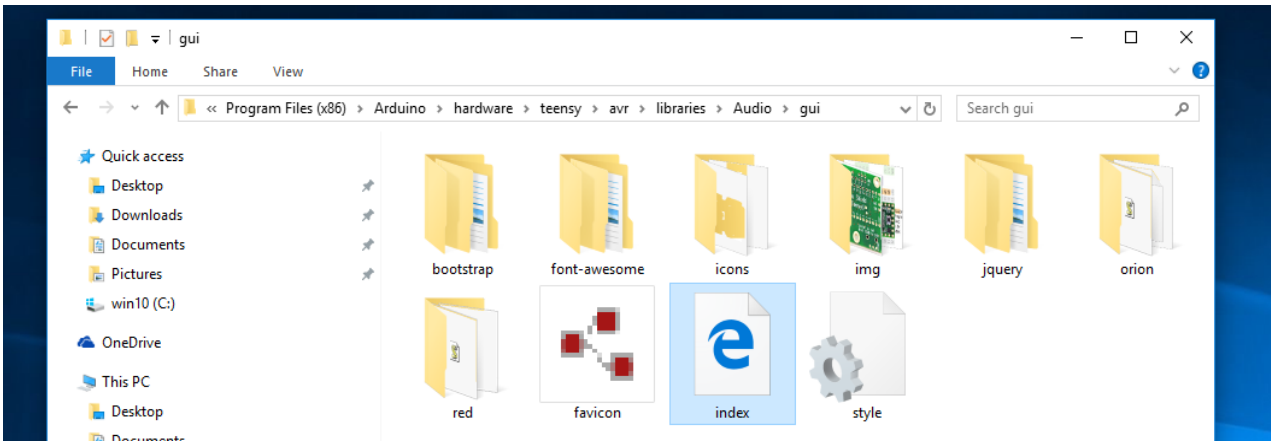
The design tools files are located deep within Arduino. On Windows and Linux, Arduino is a hierarchy of many folders and files. First, find your Arduino folder.

On Windows, the default Arduino location is **C:\Program Files (x86)\Arduino**.

From the main Arduino folder, navigate to these folders:

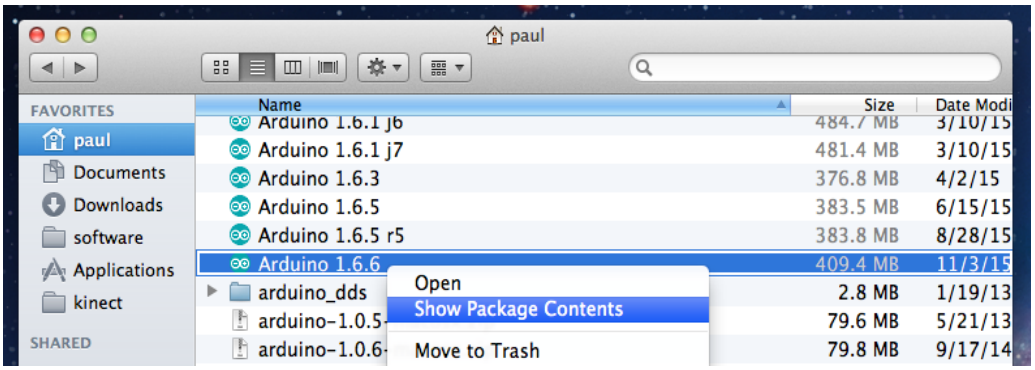
hardware / teensy / avr / libraries / Audio / gui

Inside the **gui** folder, open **index.html**.



Step #1 (without Internet access, using Macintosh):

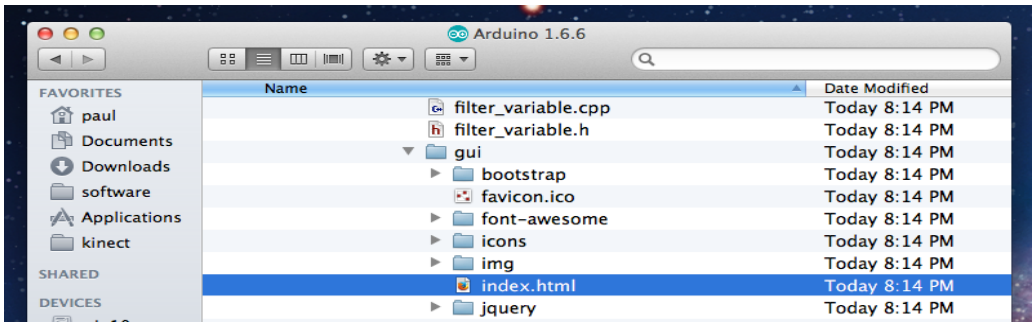
The design tools files are located deep within Arduino. On Macintosh, Arduino is a special application bundle. Hold the **Control** key and **click Arduino**, and choose **Show Package Contents**.



From the new windows which opens, navigate to these folders:

Contents / Java / hardware / teensy / avr / libraries / Audio / gui

Then open **index.html**

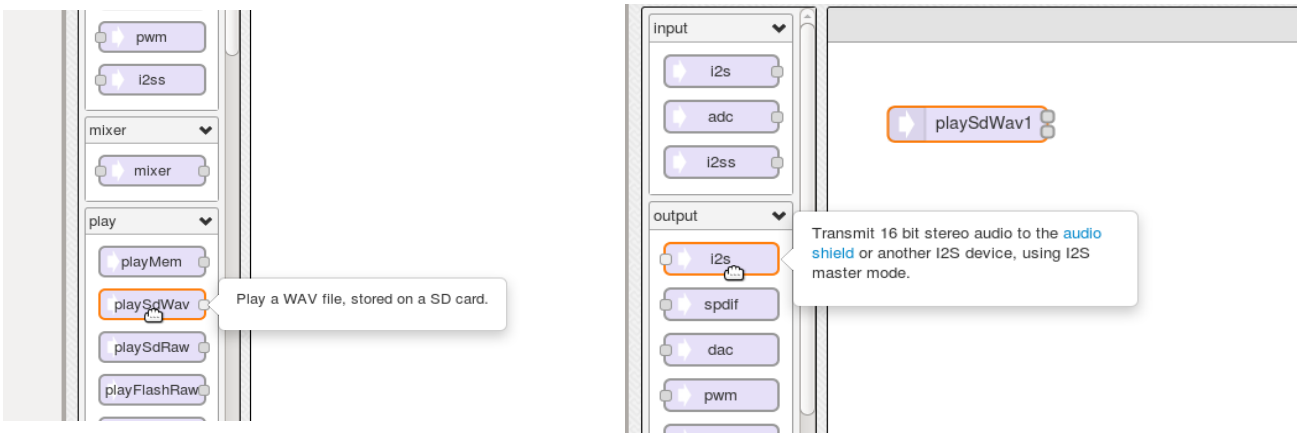


Step #2: Create an audio system

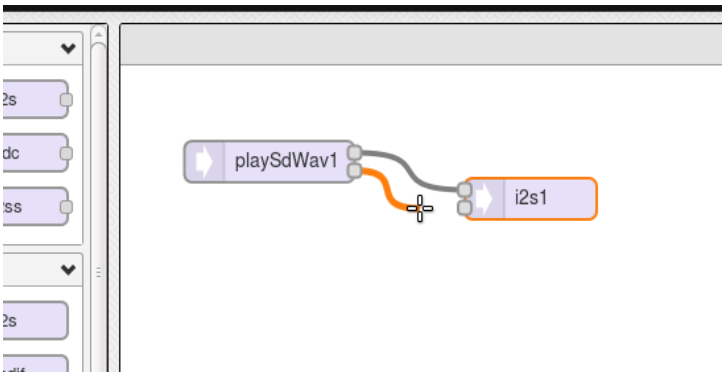
Scroll down the left panel. Objects are grouped by types. In the "**play**" section is **playSdWav**. Drag this object onto the upper left corner of the canvas.

Then locate the **i2s** object in the "**output**" section. This **i2s** object sends digital audio data from Teensy to the Audio Shield (I2S is a technical term for the signals which communicate digital stereo sound). Drag **i2s** onto the canvas, to the right of the **playSdWav** object.

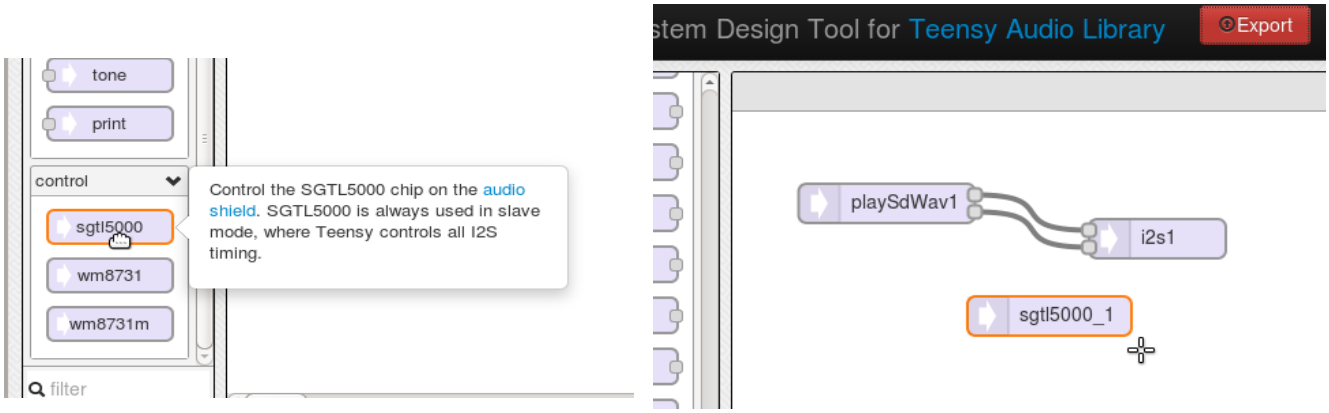
Special Note: do not use the **i2ss** object. Make sure you use **i2s** object.



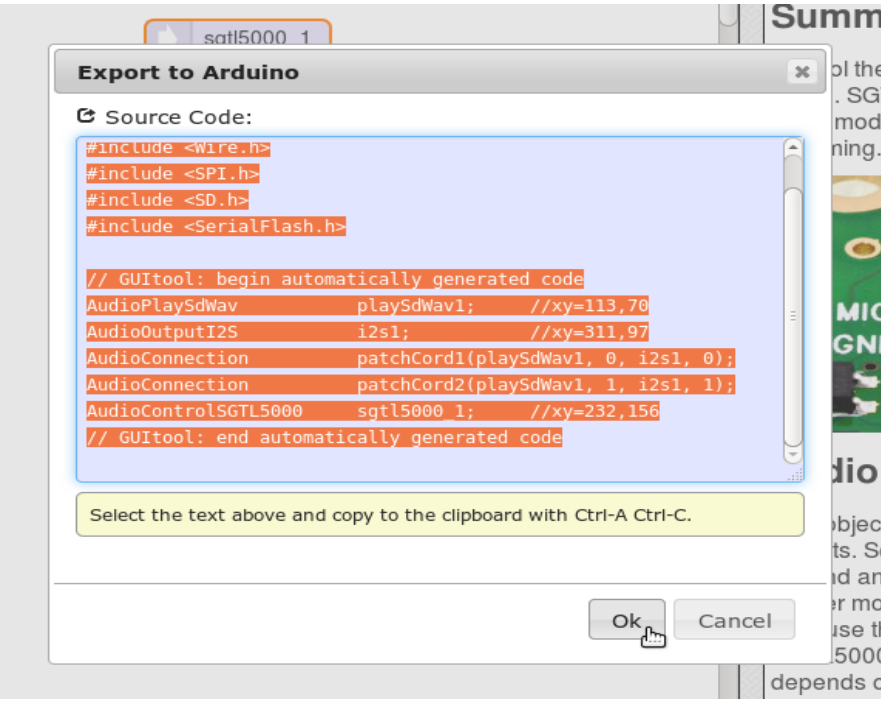
With both objects on the canvas, click (and hold) on their small squares, and drag your mouse to draw wires. Each wire causes a stream of 16 bit, 44.1 kHz audio data to automatically flow between the objects.



One more object is needed to control the audio shield. Scroll down to the end of the object list, then drag the **sgtl5000** object onto the canvas. This is a special hardware control object, *which does not have audio connections*, but must be present to control the Audio Shield hardware.



Step #3: When you've created an audio system, click the red **Export** button. The design tool will create code which you copy (**Ctrl-C**) from its export window and paste into Arduino.



Step #4: In Arduino, open the Part 2-1 example

File > Examples > Audio > Tutorial > Part_2_01_First_Design_Tool_Use

and paste the code from the design tool into the commented section *copy the Design Tool code here*

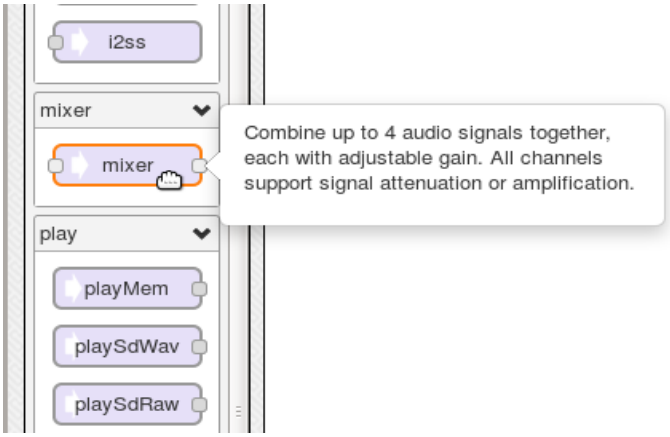
Step #5: **Verify** and **Upload** after you've copied the design tool's exported code. This program is the same basic song player from Part 1, but this time you've created the audio objects!

Part 2-2: Mixers & Playing Multiple Sounds

Playing one stereo WAV file can be very useful in many projects, but we can do so much more! Let's use mixers to combine sounds.

Step #1: Create an audio system in the design tool

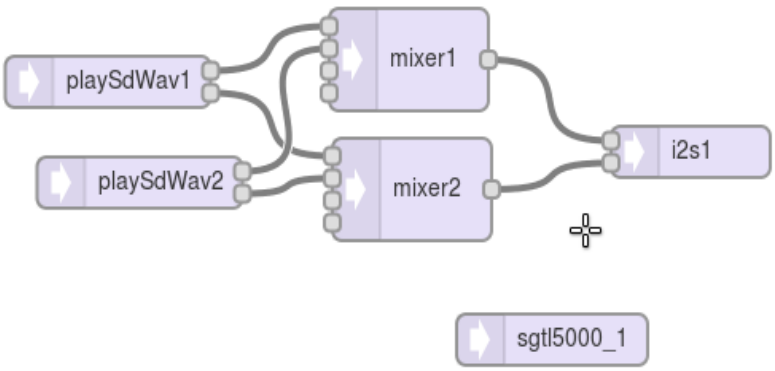
First, find the **mixer** object.



When you click any object, the right side updates with documentation.

Most objects have functions which you can call from your Arduino sketch, to configure or change what the object does. For the mixer, each input channel has adjustable gain. These can be set once, or changed at any time. In just a moment, you'll use the gain() function to switch or fade between 2 songs. The design tool serves as a handy reference manual when you write your code in Arduino.

To complete this example, draw a second **playSdWav** object onto the canvas and connect the wires as shown. Each **playSdWav** has 2 outputs, for left and right channels. **mixer1** combines both left channels and connects the left channel input of **i2s1**, and **mixer2** combines both right channels and feed the right channel of **i2s1**.



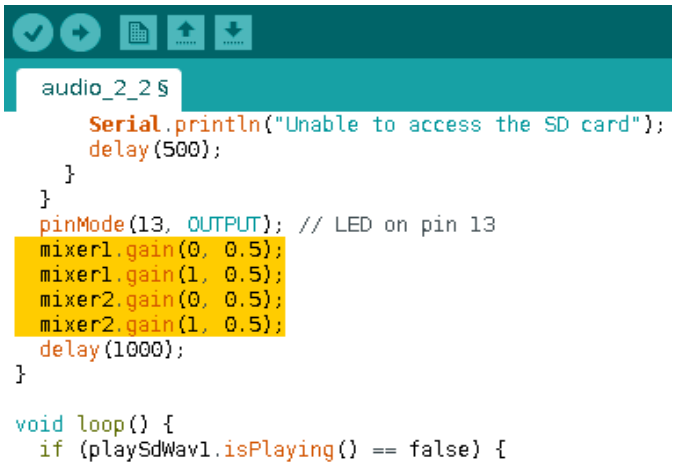
Step #2: When the audio system is designed **Export** and copy the code to paste in the Arduino sketch

Step #3: In Arduino, open the Part 2-2 example
File > Examples > Audio > Tutorial > Part_2_02_Mixers and paste the code from the design tool into the commented section *copy the Design Tool code here*

Step #4: To hear 2 songs playing together, **Verify** and **Upload** to Teensy. You should hear a jumble of music, playing 2 songs of very different genres.

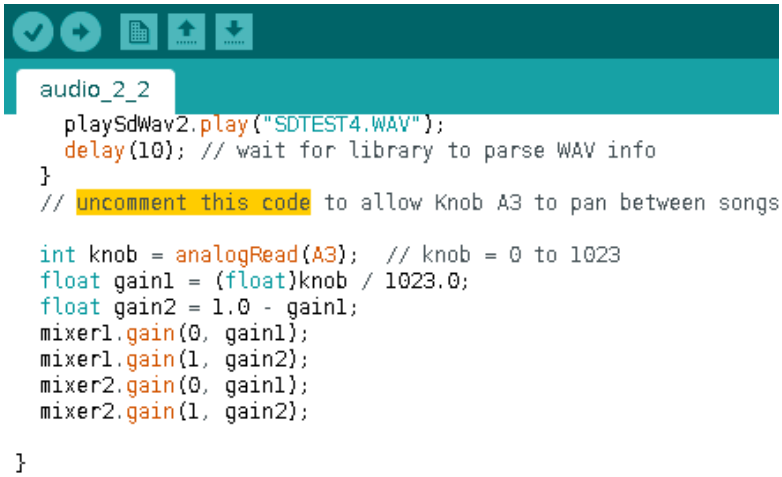
The design tool allows you to easily compose complex audio systems. Every object you place onto the canvas continuously processes CD quality sound. Each wire causes a stream of digital (44.1 kHz, 16 bit) audio to automatically flow between 2 objects.

This code near the end of setup() is responsible for the not-so-pleasing mashup of these dissimilar songs. This is the gain() function we saw documented in the design tool. Each channel is set to 0.5. If both signals happen to be at maximum of +1.0 or -1.0 signal, the worst case is adding together to 1.0 or -1.0.



Signals are always be between -1.0 to 1.0. If you mix signals to a sum beyond 1.0, the result is limited or "clipped" to 1.0. Clipping causes harmonic distortion. Gains are usually set to prevent clipping. If you have extra time, try adjusting the gains to experience clipping distortion. Much of the practical experience of audio design is familiarity with the sounds caused by common mistakes.

Near the end of this code is a section to uncomment. It will allow the A3 Knob to adjust the mixer gains at both songs play. Many objects can be put to creative uses by calling their functions from Arduino code while they're processing live audio.



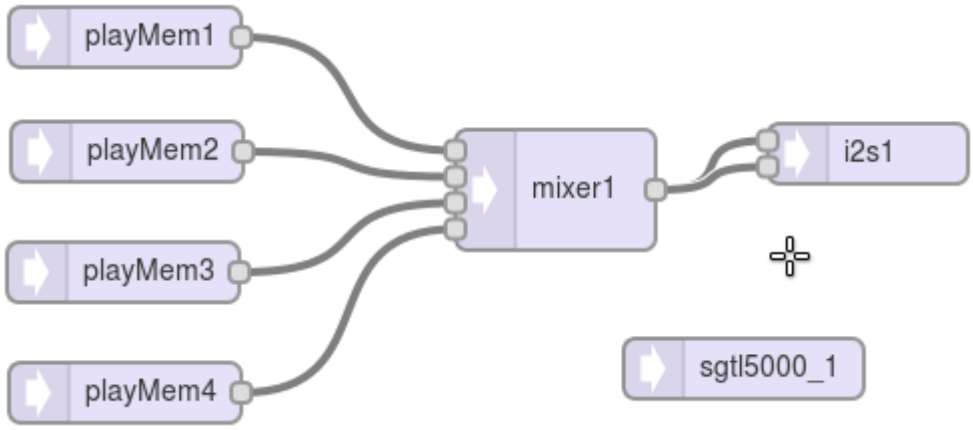
Part 2-3: Playing Samples (Short Sound Clips)

For very short sounds, you can place the sound data directly into your program. Playing sounds from Teensy's memory has 2 huge advantages:

1. It's much more efficient, allowing dozens of sounds to play simultaneously without reaching microcontroller resource limits.
2. The SD card isn't needed. If you use the DAC output instead of I2S, a very small project can be made without the large audio shield.

Step #1: Design and audio system

Draw this simple 4 sound player, using **playMem** objects. This system uses a connection type we haven't seen before, where 1 object connects to 2 others. This is perfectly fine when 1 output transmits to 2 or more inputs. However, you can't connect 2 signals to the same input. To feed more than 1 signal to any input, you must use mixers to combine them, which exactly what this system does for the 4 **playMem** objects.



Step #2: When the audio system is designed **Export** and copy the code to paste in the Arduino sketch

Step #3: In Arduino, open the Part 2-3 example
File > Examples > Audio > Tutorial > Part_2_03_Samples and paste the code from the design tool into the commented section *copy the Design Tool code here*

Step #4: **Verify** the sketch and **Upload** it to Teensy.

> ***Special Tip:***
> If you have been using copy & paste to an old window for faster Verify & Upload (as mentioned in Part 1-4), that trick will not work with this example. This sketch has several extra files, which you can't easily copy and paste.

Tapping the 3 buttons will play drum and cymbal sounds.

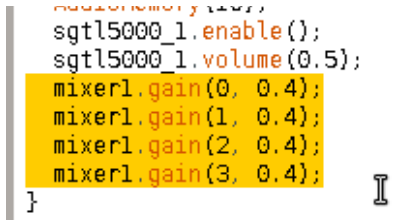
Step #5 (optional): Enable Extra Sounds

Near the end of loop() you can find more complex code which allows the 3 buttons to play 6 sounds, depending on the position of the A3 knob.

The gong and cash register sounds play for longer times. If you look at the code, the center button uses **playMem2** in one case and **playMem4** in the other. If you tap and turn quickly, can you get all 4 **playMem** objects making sound at the same moment?

Step #6 (optional): Experiment with mixer gain

While frantically tapping buttons, you might consider the mixer gain settings in this example. If all 4 sounds simultaneously produce a full 1.0 amplitude signal, we could end up with a sum to 1.6, which could cause clipping and massive harmonic distortion.



Can you notice any distortion by listening as you press all 3 buttons together? How about while the gong still rings? (odds are any distortion will be extremely difficult to hear in this scenario)

If you have a strong background in software or engineering, you may be accustomed to rules based on solid mathematical principles. Try adjusting and experimenting with the gain settings, perhaps **set the gains to 0.9** and **Upload**. Even with all 4 sounds able to add to 3.6, the louder output may still sound better or more dramatic, even with some distortion possibly occuring.

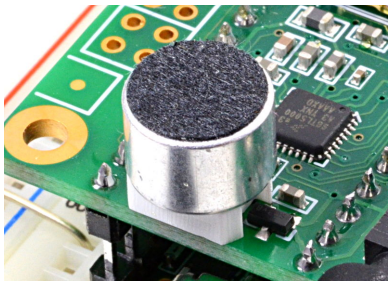
Audio design tends to be very subjective. Experimenting and listening is the best way to learn.

In this workshop example, six sound samples were provided as additional files. As you can see in the comments, these extra files are created by a program called “wav2sketch”. This is a command line utility, which must be run from a Terminal (Mac, Linux) or Command Prompt (Windows).

Using wav2sketch isn't part of this workshop, but when you need to create your own samples, the code can be found in the “extras” folder within the audio library. It's also available on Github, using this link: https://github.com/PaulStoffregen/Audio/tree/master/extras/wav2sketch

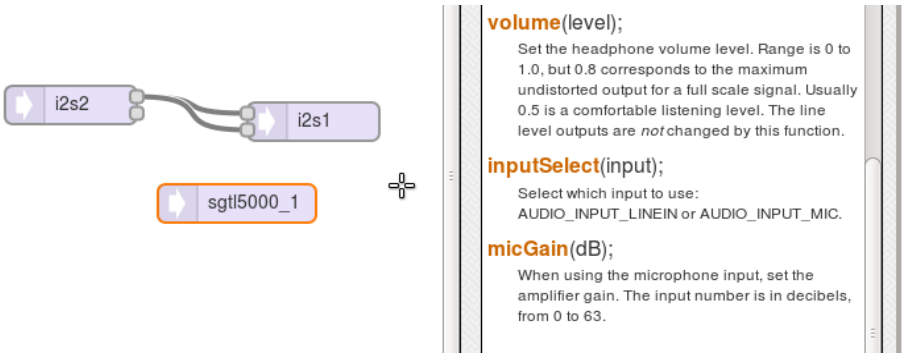
Part 2-4: Using the Microphone

Now it's time to try using the microphone. These are inexpensive mics, so you can expect telephone quality sound.



For a first microphone test, use this simple system. The **i2s** input object provides stereo data, but when the mic is in use both channels are the same data. Of course you want to connect to both outputs, because listening with only 1 ear is unpleasant.

Step #1: Design an audio system



Step #2: When the audio system is designed **Export** and copy the code to paste in the Arduino sketch

Step #3: In Arduino, open the Part 2-4 example
File > Examples > Audio > Tutorial > Part_2_04_Microphone_Check and paste the code from the design tool into the commented section *copy the Design Tool code here*

Step #4: **Verify** the sketch and **Upload** it to Teensy.

Until now, every example has simply enabled the SGTL5000 and set the headphone volume. If you review the SGTL5000 documentation, it has a dizzying number of configurable features. By default, the I2S input will use the line level input pins on the top edge of the audio shield. The inputSelect() function is needed to use the mic.

```
  sgtl5000_1.enable();
  sgtl5000_1.volume(0.5);
  sgtl5000_1.inputSelect(AUDIO_INPUT_MIC);
  sgtl5000_1.micGain(32);
  delay(1000);
}

void loop() {
```

Step #5: Experiment to find comfortable micGain setting.

The main setting to adjust is the micGain(). You can adjust between 0 to 63. Zero will not let you hear anything with this mic, and 63 will amplify a lot of static and probably be far too sensitive.

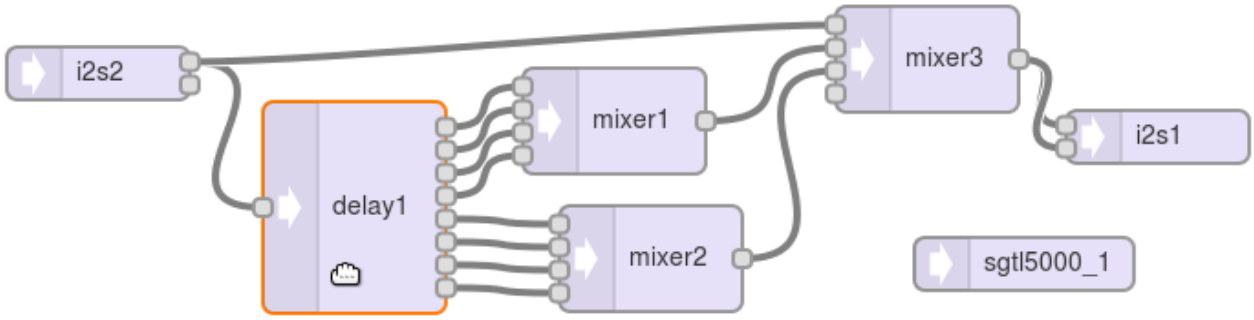
In the following examples using the mic, edit the micGain() to whatever setting is comfortable.

Part 2-5: Simple Delay

You can easily apply real-time effects using the design tool. In this part, we'll try the delay object.

Step #1: Design an audio system

When drawing this system, be careful to put **mixer3** in the correct location. The example code configures its gain settings differently than **mixer1** and **mixer2**. Also, be careful to use **delay**, not **delayExt**. The audio shields in this workshop are not equipped with RAM chip needed by **delayExt**.



Step #2: When the audio system is designed **Export** and copy the code to paste in the Arduino sketch

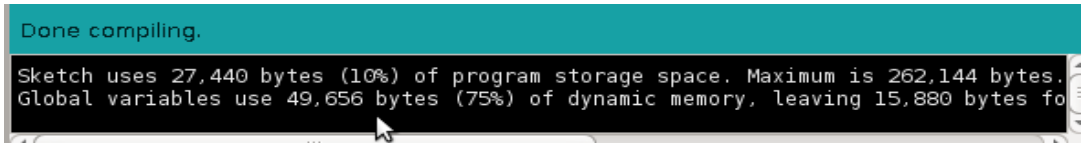
Step #3: In Arduino, open the Part 2-5 example
File > Examples > Audio > Tutorial > Part_2_05_Simple_Delay and paste the code from the design tool into the commented section *copy the Design Tool code here*

Step #4: **Verify** the sketch and **Upload** it to Teensy.

When you load this into Teensy you should hear your voice very noticeably delayed.

Until now, we've ignored the AudioMemory() line in the setup() function of every example. AudioMemory() allocates the all memory the Teensy Audio Library uses. The number you give to AudioMemory() is the total number of 128 sample buffers. Normally very little memory is required, so 10 to 20 buffers are usually plenty. Each buffer consumes 260 bytes of Teensy's RAM. You can see the impact of allocating more memory in Arduino's console window.

```
void setup() {
  Serial.begin(9600);
  AudioMemory(160);
  sgtl5000_1.enable();
  sgtl5000_1.volume(0.6);
  sgtl5000_1.inputSelect(AUDIO_INPUT_MIC);
  sgtl5000_1.micGain(32);
```



Step #5 (optional): Experiment with delay tap settings

In setup() are 8 lines which set all 8 delay outputs to 400 ms. Edit these to hear multiple delays!

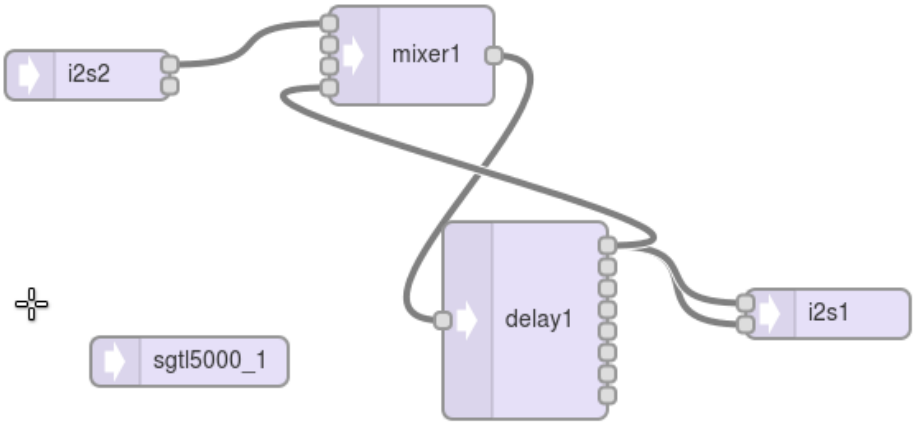
Section 2 - Achievement Unlocked

Congratulations! You've completed all of the essential material of Part 2. If you have time, keep exploring the more of Part 2, or skip ahead to start Part 3.

Part 2-6: Feedback (Echo) Delay

Most audio systems connect all their wires in forward (left to right) direction. However, backward feedback (right to left) wires are possible, especially to turn a delay into a repeating echo.

Step #1: Design an audio system



Step #2: When the audio system is designed **Export** and copy the code to paste in the Arduino sketch

Step #3: In Arduino, open the Part 2-6 example
File > Examples > Audio > Tutorial > Part_2_06_Feedback_Delay and paste the code from the design tool into the commented section *copy the Design Tool code here*

Step #4: **Verify** the sketch and **Upload** it to Teensy.

When you run the example, you'll clearly hear a repeating echo. The amount of feedback at channel 3 on mixer1 controls how much weaker the signal becomes each time it echoes, which in turn affects the number of echoes heard.

```
sgtl5000_1.inputSelect(AUDIO_INPUT_MIC);
sgtl5000_1.micGain(32);
mixer1.gain(0, 0.5);
mixer1.gain(3, 0.7);
delay1.delay(0, 400);
```

Step #5 (optional): Experiment with different feedback gain.

Uncomment two blocks of code near the end of loop(). The first part allows the A3 knob to adjust the feedback level. If you turn the knob all the up (clockwise), the feedback will be very close to 1.0. Each time the sound circulates back through the loop, it is reduced only slightly, so it keep echoing for a very long time.

Feedback signals are risky, if the total signal gain around the loop is ever more than 1.0. If the signal becomes stronger each time it travels around the loop, it will grow without bound, until clipping. This is called an unstable feedback loop.

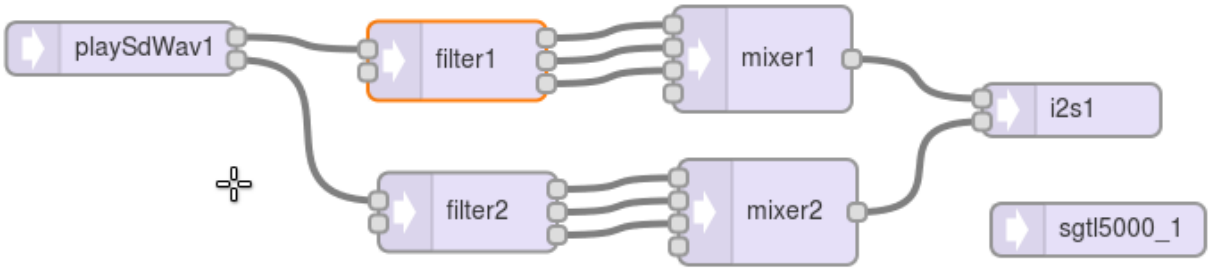
To experiment with an unstable loop, hold the left button to double the feedback. If the knob it turned up past half, the result is an unstable loop.

In natural echo, the sound decreases but also changes as it echoes. You can create a similar effect by adding a filter object in the feedback path. Each pass around the loop will decrease the signal and also decrease its high frequencies even more, to achieve a very natural echo effect. Combinations of looped delay (a loop within the loop) can create rich reverb effects.

Part 2-7: Filters

Filters allow a portion of the audio spectrum to pass, while reducing the rest. The simplest filter to use in the Teensy Audio library is the state variable filter, labeled simply **filter** in the "filter" section. In this part, you will listen to filtered music, to hear the effect each type of filtering.

Step #1: Design an audio system



These mixers as a signal routing switch. One channel will be set to 1.0 gain, to pass the signal straight through so we can hear the effect of the filter. The other mixer channels will set to zero to block those signals. The mixers won't actually be doing any "mixing", just routing one of the filter's outputs.

Step #2: When the audio system is designed **Export** and copy the code to paste in the Arduino sketch

Step #3: In Arduino, open the Part 2-6 example
File > Examples > Audio > Tutorial > Part_2_07_Filters and paste the code from the design tool into the commented section *copy the Design Tool code here*

Step #4: **Verify** the sketch and **Upload** it to Teensy.

Step #5: Listen to each filter type. Vary the filter frequency to hear how the filter changes sound.

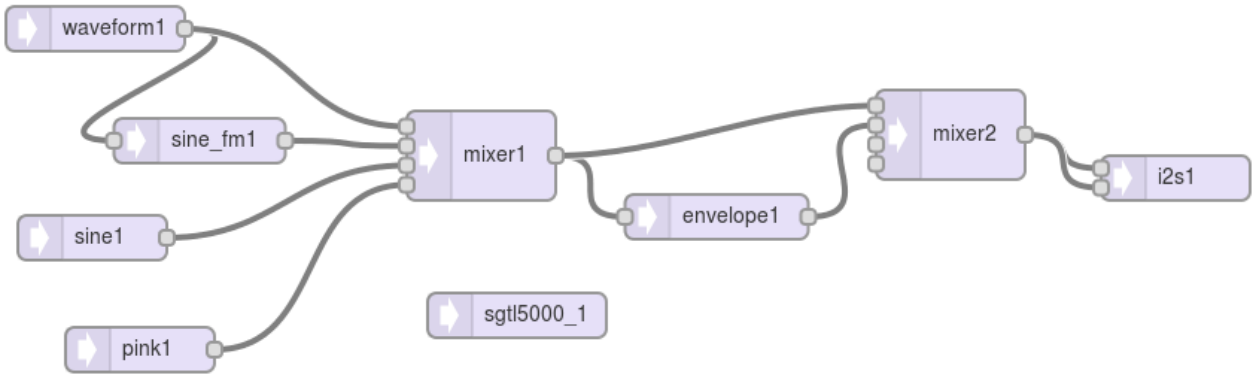
The example program uses the 3 buttons to reconfigure the mixers, and the A3 pot to allow you to vary the filter frequency as the music plays. The serial monitor window will show the actual frequency setting as you turn the knob. On the low-pass and high-pass signals, the music will seem to get quieter as you filter away more of its spectrum.

You may have noticed the filter objects have 2 inputs. The second input allows another audio signal to vary the filter frequency automatically. Details and functions to tune parameters can be found in the design tool documentation. Strange effects can be achieved by using an oscillator to "sweep" the filter frequency rapidly.

Part 2-8: Oscillators and Envelope

Synthesis is the process of creating sounds. In this part, you'll listen to some of the fundamental syntheis building blocks.

Step #1: Design an audio system



In this system, waveform1 is the "control waveform". It is (usually) a relatively low frequency which controls another signal. Like in the previous example, these mixers are used as switches to allow us to hear one of the 4 synthesized sounds, and to hear it directly or modified by the envelope1 object.

Step #2: When the audio system is designed **Export** and copy the code to paste in the Arduino sketch

Step #3: In Arduino, open the Part 2-8 example
File > Examples > Audio > Tutorial > Part_2_08_Oscillators and paste the code from the design tool into the commented section *copy the Design Tool code here*

Step #4: **Verify** the sketch and **Upload** it to Teensy. Open the **Serial Monitor** window.

When this sketch runs, the buttons and knobs have these functions:

- Button 0 (left): Cycles through the 5 control waveform types
  - Sawtooth
  - Sine
  - Square
  - Triangle
  - Pulse
- Button 1 (middle): Cycles through the 4 mixer1 signals
  - Control Waveform
  - Frequency Modulated Sine
  - Pure Sine
  - White Noise
- Button 2 (right): Activate the envelope
  - Press = Note Begin
  - Release = Note End
  - (after 4 seconds) = Revert to steady tone from mixer1
- Knob A2 (left): Varies control waveform frequency
- Knob A3 (right) Varies sine_fm1 & sine1 frequency

Step #5: Listen to the five control waveform types

If you've already fiddled with the settings, press the middle button until mixer1 is passing the control oscillator. Then turn knob A2 clockwise, until the control oscillator is a recognizable tone.

Press the left button and listen to each control waveform. The serial monitor will tell you which one is playing. Each has a distinctive sound.

Step #6: Listen to the modulated waveform with each control type

Turn the A2 knob fully counterclockwise. This will configure the control oscillator to a very low frequency, far below human hearing. If sawtooth, square or pulse are selected, you may hear occasional clicking, but the tone of the waveform itself is inaudible.

Press the middle button to configure mixer1 to pass the modulated sine wave. Adjust A3 for a comfortable tone range.

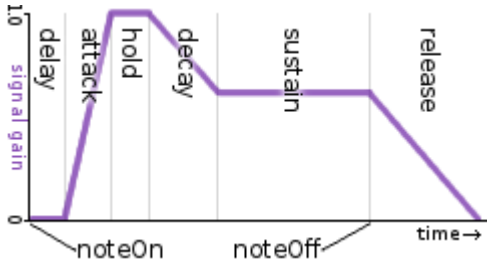
Listen to the modulated sine with each control waveform (left button). You should be able to hear the effect of the control waveform shape. Sawtooth causes a gradual rise in tone, the sudden drop back down. Sine varies the tone back and forth. Square will alternate between 2 tones. Triangle should be similar to sine, but the tone changes less smoothly. Pulse should be similar to square, except you will hear one tone nearly all the time and the other only briefly.

Step #7: Experiment with faster control waveforms

While listening to the modulated waveform, increase the control waveform (knob A2). You should here funny sounds. With sawtooth or triangle control, you should hear strange tones, perhaps "metallic" sounding, as the control frequency rises much higher.

Step #8: Listen to envelope effect

Adjust the settings for a distinctive sound, and then repeatedly tap the right button to activate the envelope. Tap it to a simple rhythm.



Envelope tries to transform a tone into the intensity profile of a musical note. It rapidly changes the signal level in several phases

Consider a flute. The attack phase is similar to initially blowing into the tube, causing pressure to build before any substantial sound is hear. The decay phase emulates the equalizing of pressure between the instrument and musician. The sustain phase is the steady sound heard as the musician continues blowing in the tube. The final release phase emulates the gradual decrease in pressure & sound after the musician stops applying air pressure.

These oscillators and envelopes are some of the basic building blocks for synthesis.

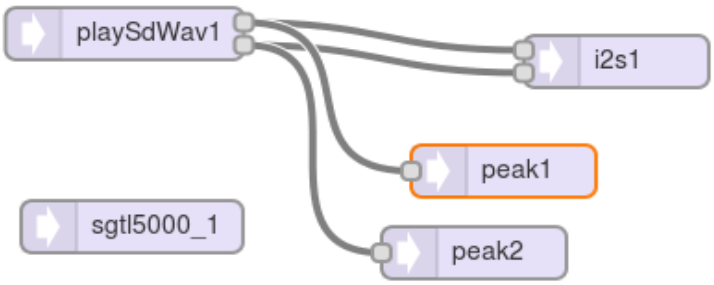
Section 3: Audio Analysis

We played, created and processed sound in the prior sections, which required only simple control from Arduino sketches to start playing or configure settings. In this section, we'll explore more sophisticated ways for Arduino code to interact with the audio system. We'll also look at the limitation of processing audio on a single chip microcontroller.

Part 3-1: Peak Detection

For some projects, like sound reactive costumes and DJ stage lights, your Arduino sketch needs information about the sound, rather than just allowing the audio system to process it. The analysis objects fill this role.

Let's start with **peak**, the simplest analysis object.



Step #1: Design an audio system

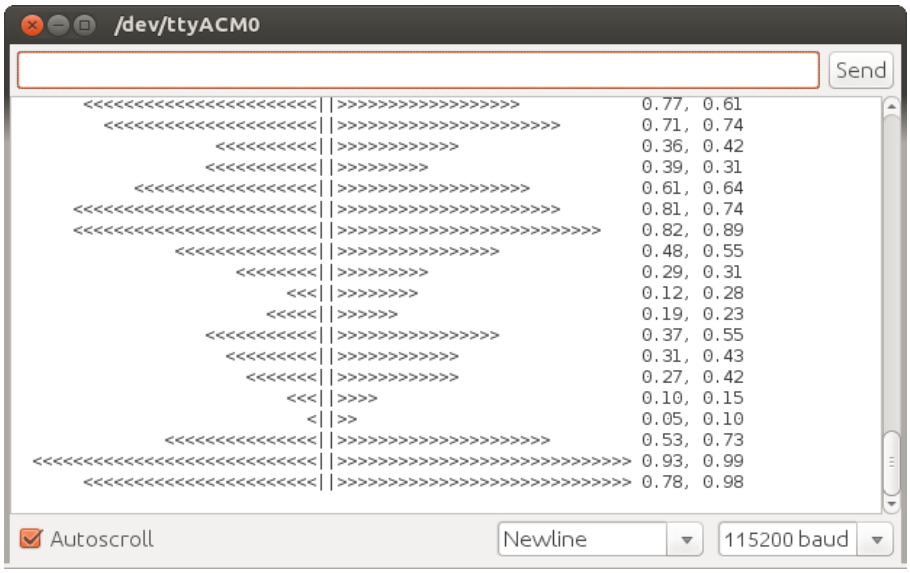
The audio **playSdWav1** is sent to both the audio shield and the **peak1** & **peak2** objects. A subtle issue is the need for the **i2s1** object. All audio systems require at least one physical input or output, due to a technical requirement of the Teensy Audio Library.

Step #2: When the audio system is designed **Export** and copy the code to paste in the Arduino sketch

Step #3: In Arduino, open the Part 3-1 example
File > Examples > Audio > Tutorial > Part_3_01_Peak_Detection and paste the code from the design tool into the commented section *copy the Design Tool code here*

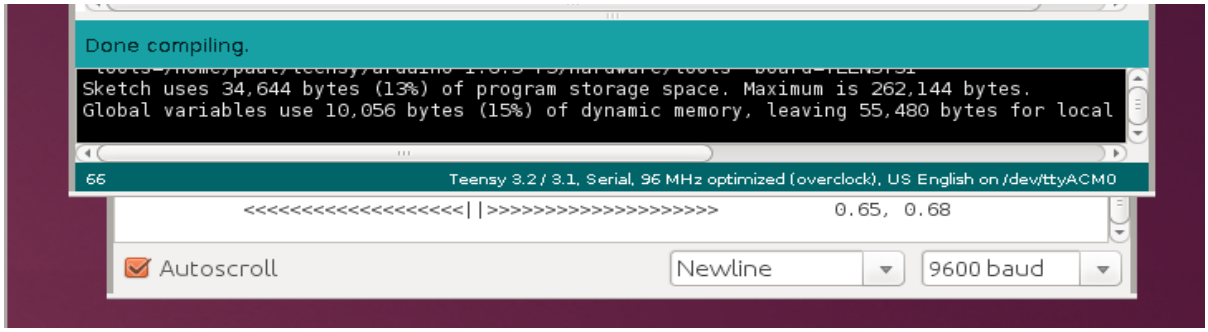
Step #4: **Verify** the sketch and **Upload** it to Teensy.

When you run this code, you should see this display in the Arduino Serial Monitor.



The peak objects give numbers between 0 to 1.0, representing the largest amplitude peak of the audio signal during the time elapsed since the previous reading.

To see the ASCII graphic as it might appear in LEDs on a stereo display, move another window on top of the Arduino Serial Monitor, to block all but the last line.



Looking at the code, you can see the peak objects have functions very similar to other Arduino libraries. The available() function tells you when the you can read more data, and the read() functions give you the data. These read() give floating point numbers, rather than data bytes.

```
if (msecs > 40) {
  if (peak1.available() && peak2.available()) {
    msecs = 0;
    float leftNumber = peak1.read();
    float rightNumber = peak2.read();
    int leftPeak = leftNumber * 30.0;
```

These familiar-looking functions do have subtle but important differences from Arduino's normal usage. With incoming Serial data, available() and read() access a queue, where available() tells you how many items are currently in the queue, and read() removes 1 item from the queue for you. They are designed to prevent data loss. You want to read the oldest data first, even it if actually arrived some time ago.

Audio analysis objects are designed to provide only recent information. The available() returns only true or false. When new information is available, only the most recent data can be read. Audio analysis is designed to lose old information. Some analysis, like peak, gives you the cumulative effect since your last read(). Information about smaller peaks is automatically lost as larger peaks occur. Other types of analysis give you information from a recent window in time, where results from previous windows are automatically discarded. Details are in each object's documentation in the design tool.

This example has code to lower the printing speed. It uses an elapseMillis variable, which is a convenient Teensyduino feature. elapseMillis are integers which automatically increment 1000 times per second.

```
// for best effect make your terminal/moni

elapsedMillis msecs;

void loop() {
  if (playSdWav1.isPlaying() == false) {
```

The example only prints if msecs has incremented past 40. Try reducing this number. If you use 0, it will print every time new peak analysis is available. Though very fast, this speed is 1 update for every 128 audio samples, or about 344 Hz.

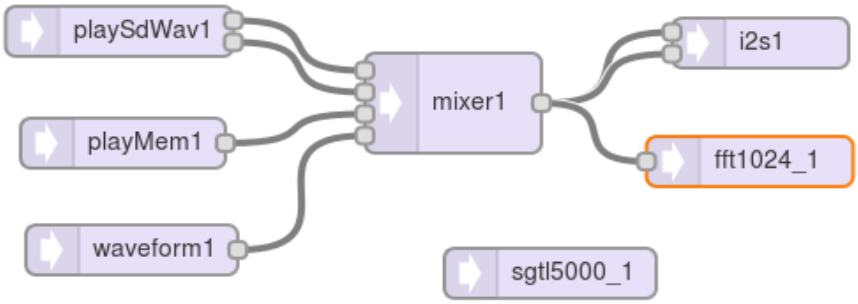
Part 3-2: Fourier Transform

Fourier transform gives you detailed analysis of the audio spectrum. It turns data in time to data in frequency, which is really useful for building an art installation or costume which reacts uniquely to sounds in different frequency ranges.

The acronym FFT means Fast Fourier Transform, which refers to a mathematical optimization. FFTs have two types of output. Complex output gives you 2 numbers per frequency, and is generally needed if you will turn the frequency data back to an audio signal. The Teensy Audio Library provides the simpler Real output, where you get a single number per frequency. The numbers tell you the amount of signal found at each frequency, without any information about its phase shift.

To begin exploring Fourier transform, draw this simple system. Even with the FFT optimization, a 1024 point analysis (which provides detailed results) will tax Teensy's computational power, so only a single **fft1024** object is used. The stereo WAV file is mixed to mono for the sake of analysis.

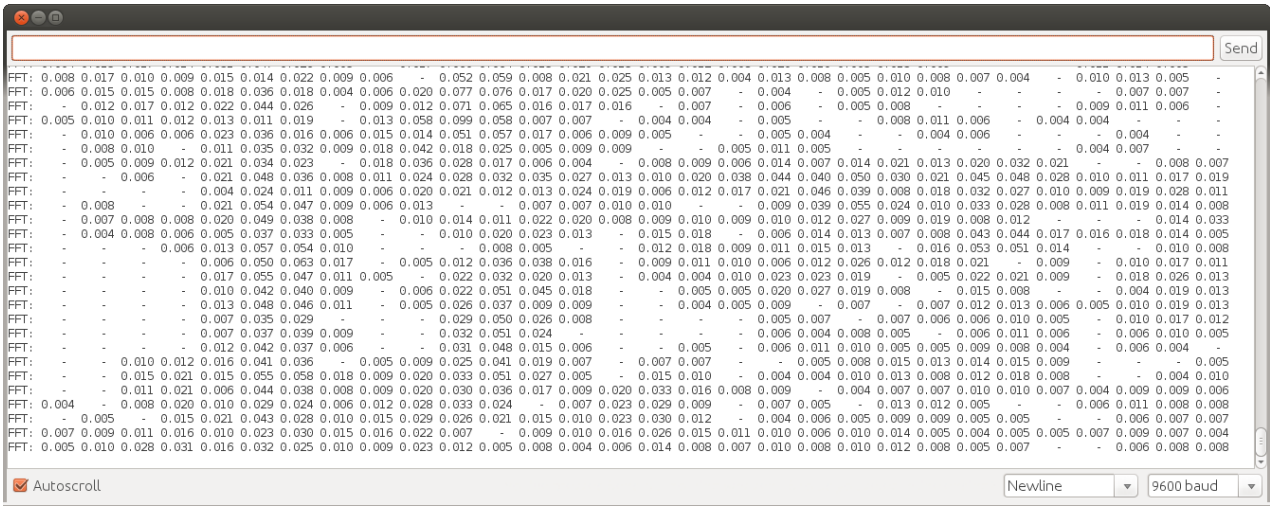
Step #1: Design an audio system



Step #2: When the audio system is designed **Export** and copy the code to paste in the Arduino sketch

Step #3: In Arduino, open the Part 3-2 example
File > Examples > Audio > Tutorial > Part_3_02_XXX and paste the code from the design tool into the commented section *copy the Design Tool code here*

Step #4: **Verify** the sketch and **Upload** it to Teensy. When run the code, this should appear in the Arduino Serial Monitor.



The FFT analysis produces a tremendous amount of data. This rapidly scrolling window shows only the first 30 of the 512 frequency "bins". However, these 30 are much more important than the remaining 482. Later we'll look at why that is. For now, let's concentrate on understanding these numbers.

Each frequency bin represents the amount of signal found at a particular frequency. The bins are 43 Hz apart.

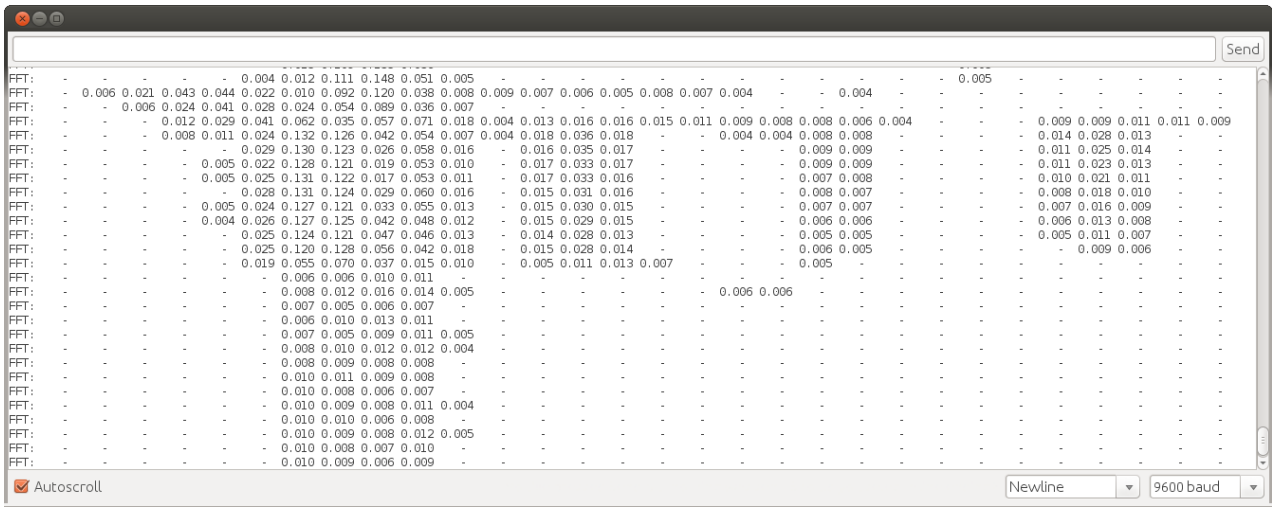
```
Bin 0 = DC or zero Hz component
Bin 1 = 43 Hz
Bin 2 = 86 Hz
Bin 3 = 129 Hz
Bin 3 = 172 Hz
Bin 4 = 215 Hz
Bin 5 = 258 Hz
Bin 6 = 301 Hz
Bin 7 = 344 Hz
etc....
```

The numbers are all typically small. You may recall from Part 2-2 Mixers that signals range from -1.0 to 1.0. The FFT considers only the total amount, so you get only positive numbers. If a signal total is 1.0, meaning it's actually oscillating between -1.0 to 1.0, the FFT will report it as 1.0, likely spread across many bins if it's composed of many frequencies.

For example, if a signal with total amplitude of 0.6 which is a mixture of 25% 120 Hz and 75% 1 kHz, bins clustered around 3 (129 Hz) and around 23 (989 Hz) will indicate the amount of each part. Bin 3 and nearby bins will sum to 0.15, and pin 23 and nearby bins will sum to 0.45. The sum of all bins always equals the total amount of the signal. There are some finer details to consider, but first let's do some listening tests to see the data corresponding to real sounds.

Step #5: Guitar listening test

If you press the 3 buttons, you'll hear each activates one of the 3 signal sources you drew in the design tool. When you listen to the guitar sample from the *middle* button, you should see this:



After the sound stops, it's easy to scroll up and see the notes. In this screenshot, you can see the moment the string is plucked, which results in a few lines with rapidly changing numbers in many frequency bins. Then the guitar string vibrates with at least 4 frequency ranges (visible in this data). Soon the higher frequency components fade, as the guitar string vibration settles towards the wavelength matching the string length between the player's finger and the bottom of the instrument.

Step #6: Experiment with thresholds for data visualization

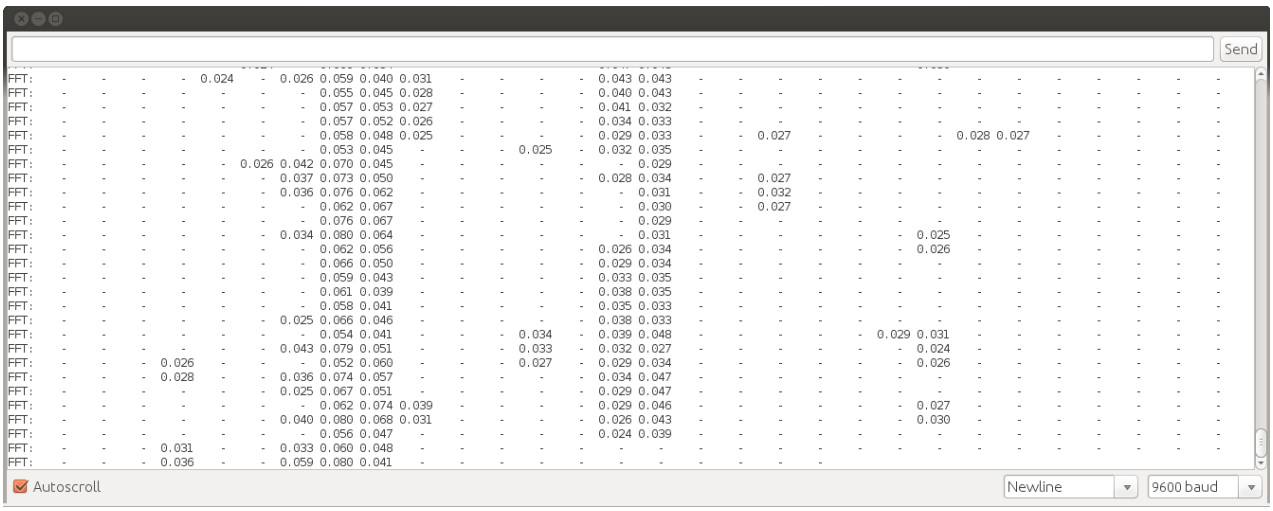
Visually detecting patterns in FFT data of composed music is much harder than a single instrument. The many separate sounds often overlap in frequency bins. Human hearing and the brain's ability to discern complex combinations of sounds is pretty amazing.
To improve things a bit, find the printNumber() function near the end of the example.

```
void printNumber(float n) {

  if (n >= 0.004) {
    Serial.print(n, 3);
    Serial.print(" ");
  } else {
    Serial.print("   -  "); // don't print "0.00"
  }
```

Editing the threshold. Increase it from 0.004 to 0.024. This will print less detail, allowing you to more easily see the numbers for only the stronger sounds.

The first song that automatically plays is “Where You Are Now” featuring WolfSky singing. She has a strong voice, which usually shows up as 1 or 2 columns of numbers scrolling by, as in this screenshot.



Unlike the acoustic guitar piece from the middle button, the guitars and other instruments in this music have many complex effects applied. Those effects which produce rich and complex sounds tend to scatter the instrument's numbers across many frequency bins. Because the bins add up to the signal's total, the numbers in most bins are lower, mostly below this higher 0.024 threshold.

If you have extra time, perhaps try adjusting the threshold to see if other types of sounds become noticeable in the scrolling data. The printNumber() function also has alternate ASCII art code you can try, and perhaps even extend or redesign with other patterns to better see the relative signal strength in each bin?

Scrolling numbers probably won't impress many people, but perhaps instead of printing to the serial monitor, you could turn LED on/off, animate RGB LEDs to different colors, or control solenoids and motors to visualize the music?

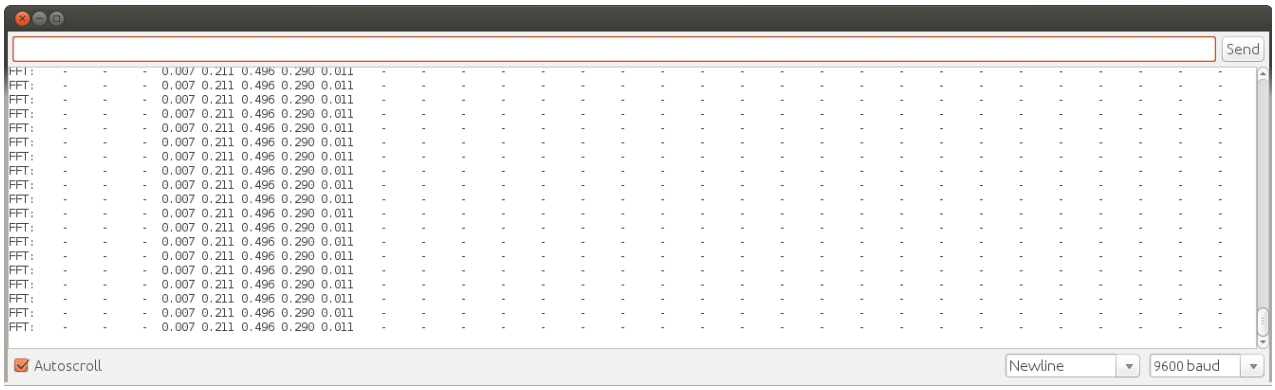
Step #7 (optional): Understand FFT mathematical limitations (or skip to Part 3-3)

The *right* button plays pure sine waves. Each time to press it, you'll hear a higher pitch, until it cycles through all 12 musical notes. The exact 12 frequencies it plays are from this array.

```
int noteNumber = 0;
const float noteFrequency[12] = {
  220.00,  // A3
  233.08,  // A#3
  246.94,  // B3
  261.63,  // C4
  277.18,  // C#4
  293.66,  // D4
  311.13,  // D#4
  329.63,  // E4
  349.23,  // F4
  369.99,  // F#4
  392.00,  // G4
  415.30   // G#4
};
```

If you previously edited the printNumber() threshold, restore it to 0.004.

Press and hold the right button. When Teensy plays 220 Hz (musical note A3), you will see this.



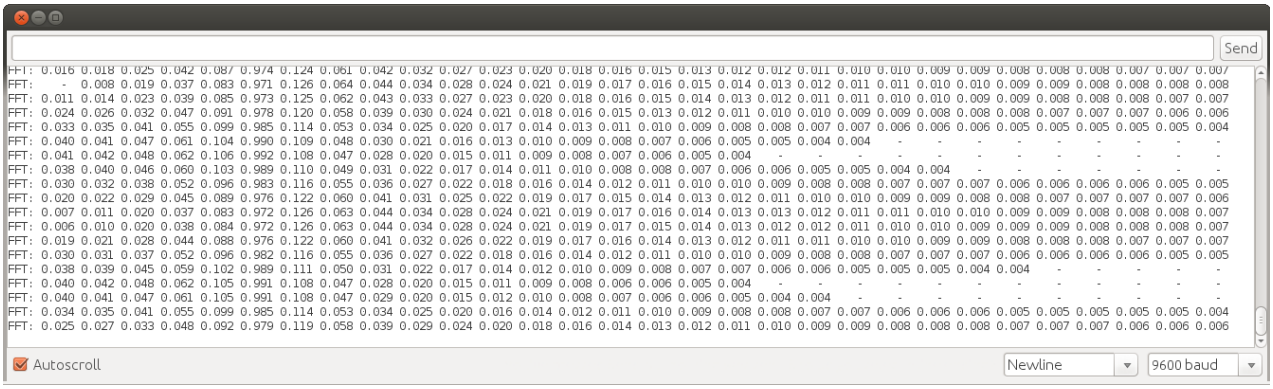
This example shows a limitation of the Fourier transform. For signal frequencies which don't align perfectly on the FFT bin frequencies, you can never have ideal performance.

By default, the **fft1024** object uses a Hanning window, which causes frequencies not exactly on a FFT bin to cluster in the close-by bins, but it also smears the data across several nearby bins. Ideally, you'd like your 220 Hz frequency to mostly appear in the 215 Hz bin, with some perhaps in the 258 Hz bin. The Hanning window gets you close, with 0.496 in the 215 Hz bin, and 0.290 in the 258 Hz bin. But 0.211 ends up in the 172 Hz bin, and small amounts go to the 129 and 301 Hz bins, which isn't so desirable. They still add up to 1.0, the original signal size, but they're smeared across 5 bins.

You can disable the Hanning window by editing this code. Just uncomment the line which sets the window to NULL. Your signal will be processed by the FFT without any window processing.

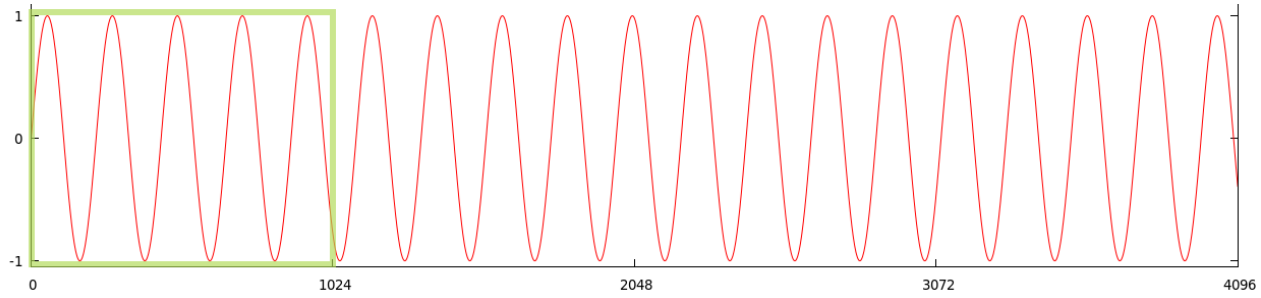
```
mixer1.gain(3, 0.0);
// Uncomment one these to try other window functions
 fft1024_1.windowFunction(NULL);
// fft1024_1.windowFunction(AudioWindowBartlett1024);
// fft1024_1.windowFunction(AudioWindowFlattop1024);
delay(1000);
```

When you press the button for 220 Hz sine wave tone, you should see this with the window disabled.



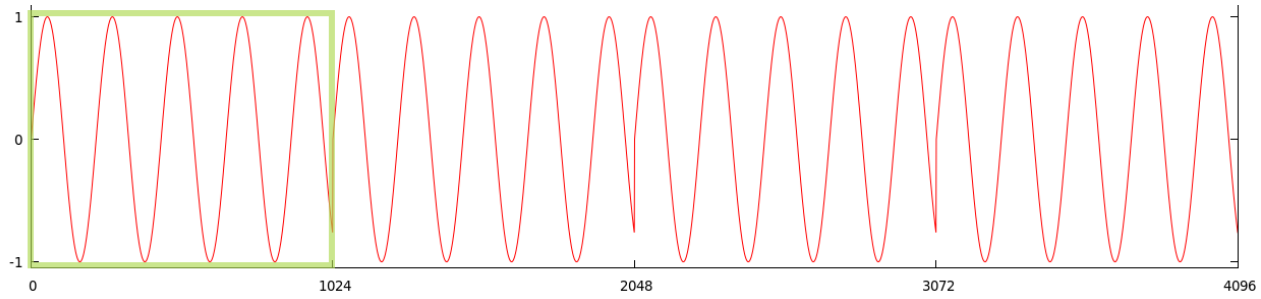
At first glance, this looks pretty horrible. But if you read the actual numbers, you'll see the 215 Hz bin has numbers between 0.971 to 0.992. The 258 Hz bin is between 0.108 to 0.126. So almost all of our 220 Hz sine wave did go into the 2 desired bins. However, some of it got scattered across almost all the other bins. This problem is called Spectral Leakage. The windows are meant to prevent spectral leakage, containing all the results to only nearby frequency bins.

To understand how a pure 220 Hz sine wave becomes data in all those other frequency bins, consider this plot of 4096 points of a 220 Hz sine wave, sampled at 44.1 kHz.

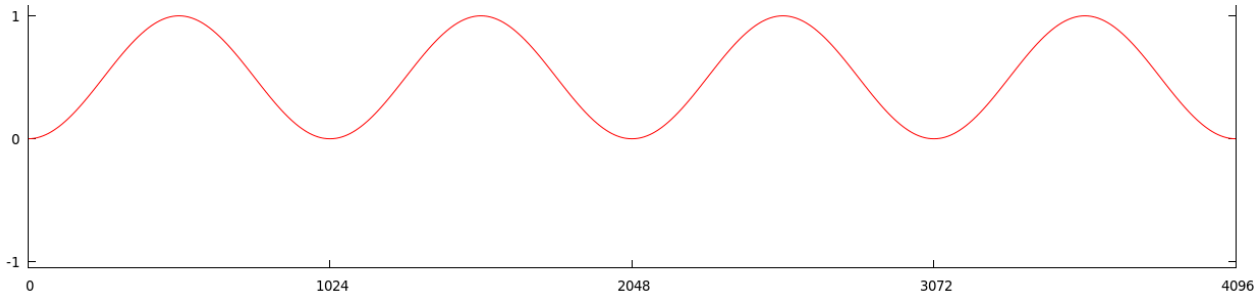


The green box is 1024 points where the FFT analyzes the spectrum. The FFT reports the spectrum based on the assumption the waveform is periodic, that it repeats indefinitely.

Here is the same 220 Hz waveform with the first 1024 point section repeated 4 times. Without a window, the FFT is returning the spectrum of this waveform, which differs from the intended 220 Hz pure sine wave.

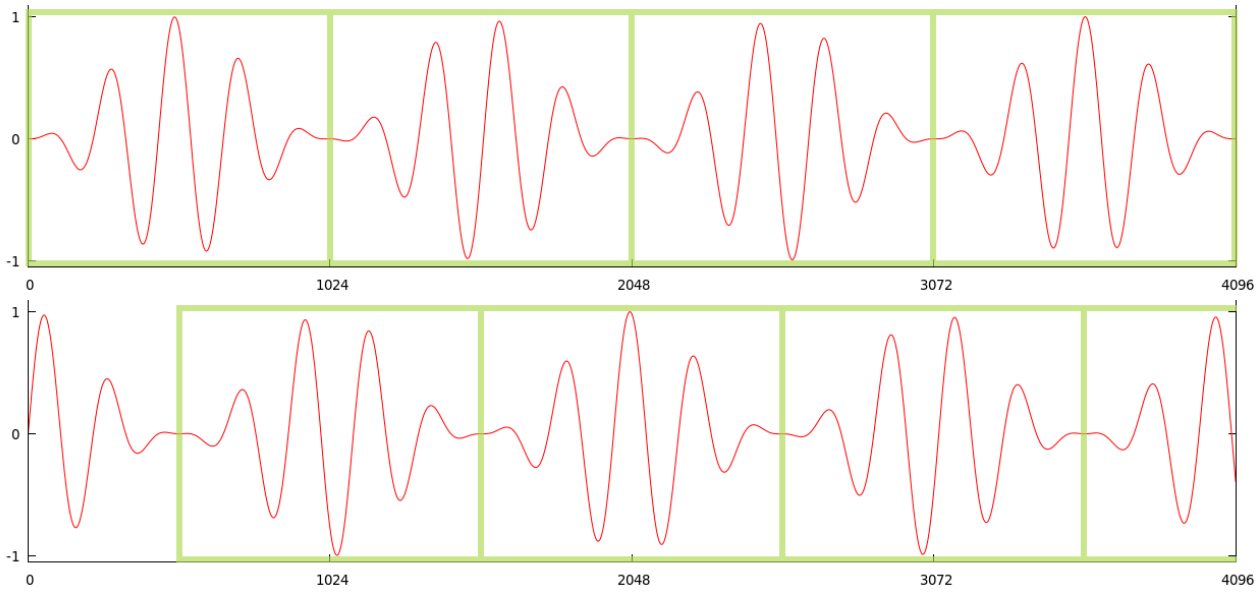


The window function is just another waveform which the signal is multiplied by, before the FFT. The Hanning window is basically just an offset sine wave, which multiplies the original waveform by zero at the beginning and end of the window, and by 1 in the middle.



Window functions destroy about half of the original data, where they multiply by zero or very small numbers. For this reason, the Teensy Audio Library uses 50% overlap in its **fft1024** object. Twice as many 1024 point FFTs are computed, where the second set uses the window function offset by 512 points.

With the pure 220 Hz tone, this graph shows the actual windows inputs to each 1024 point Fourier transform.



A new 1024 point FFT is completed every 512 samples, because they are performed at twice the rate on 50% overlapping data after windows are applied.

Because of the 50% overlap, fft1024_1.available() will return true 86 times per second. Each new update of **fft1024** data represents the prior 1024 samples (approximately 23.2 ms) with a window applied, so it was most sensitive to the sound in the center of those 23.2 ms.

Many different window shapes are available in the library, which trade off spectral leakage versus smearing of the frequency bins. There is no magic solution if your signals have frequencies not perfectly aligned onto the FFT bins. But despite these limitations, the FFT works great for sound reactive projects.

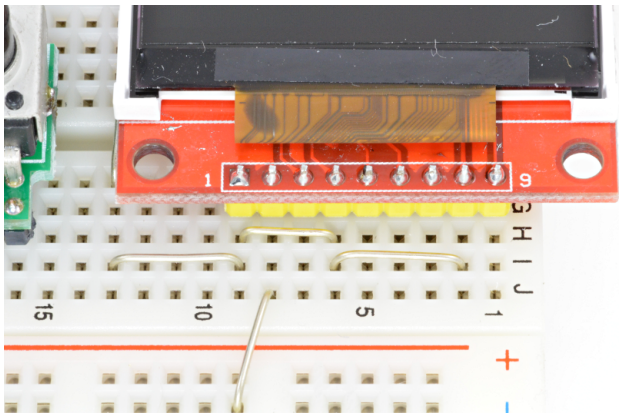
Part 3-3: Add a TFT Display

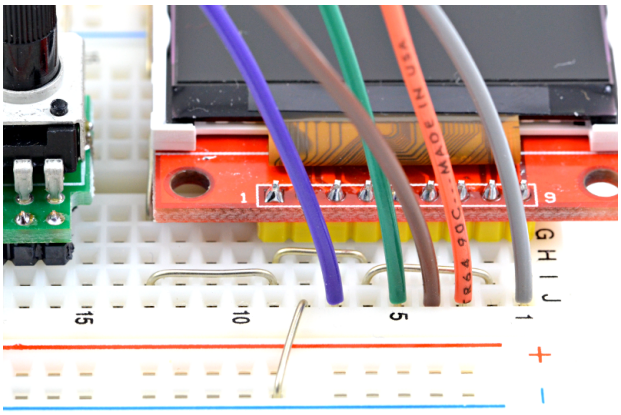
In this last part, you'll connect a TFT display for visualization of the peak detect analysis.

Step #1: Add the TFT display and connect 5 signals

Disconnect your Teensy from the USB cable before changing breadboard wiring.

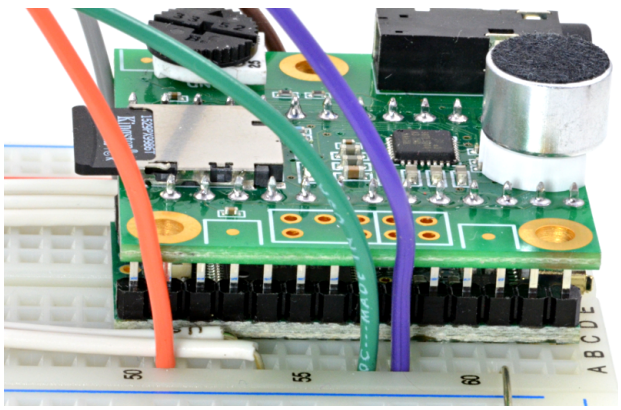
Place the TFT display on the last 9 holes of the breadboard, locations 9-G to 1-G.

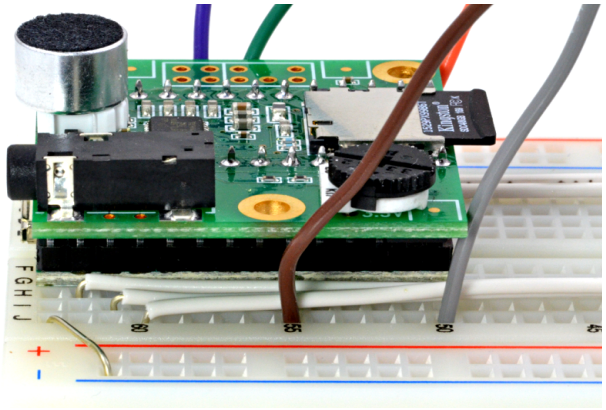




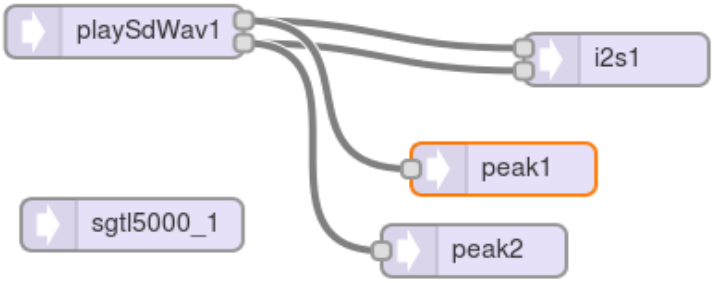
Your breadboard already has the 3 VCC and 1 GND connections installed. Five colored wires are needed to complete the data path.

Signal	Color		Teensy Pin	Breadboard		Display Pin	Breadboard
VCC	-		3.3V			1	
GND	-		GND			2	
CS	Purple		21	58-A		3	7-J
RESET	-		3.3V			4	
D/C	Green		20	57-A		5	5-J
MOSI	Brown		7	55-J		6	4-J
SCK	Orange		14	51-A		7	3-J
LED	-		3.3V			8	
MISO	Gray		12	50-J		9	1-J





Step #2: Design an audio system. This is the same system from Part 3-1.

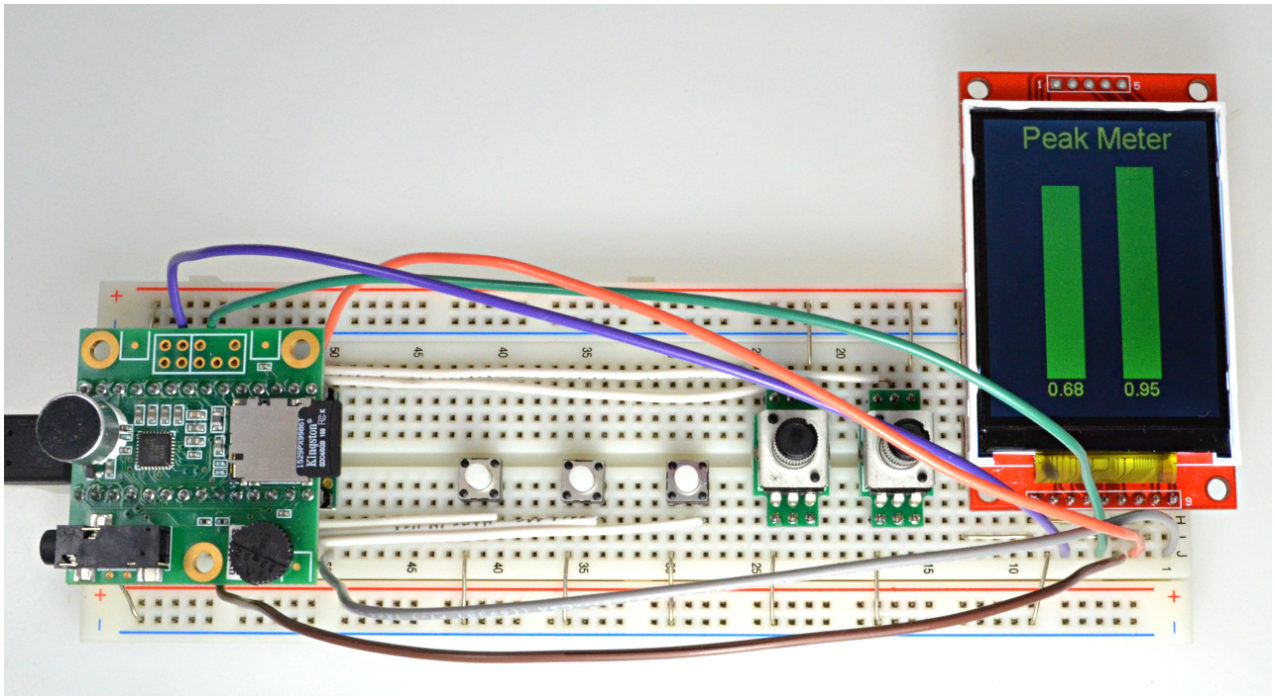


Step #3: When the audio system is designed **Export** and copy the code to paste in the Arduino sketch

Step #4: In Arduino, open the Part 3-3 example
File > Examples > Audio > Tutorial > Part_3_03_TFT_Display and paste the code from th design tool into the commented section *copy the Design Tool code here*

Step #5: Reconnect the USB cable, **Verify** the sketch and **Upload** it to Teensy.

When Teensy begins running this sketch, you should see the display show a simple visualization of the peek level meter.



Hopefully this rapidly updated TFT example may inspire you to continue exploring with the Teensy Audio library.