Circuit Playground-O-Phonor

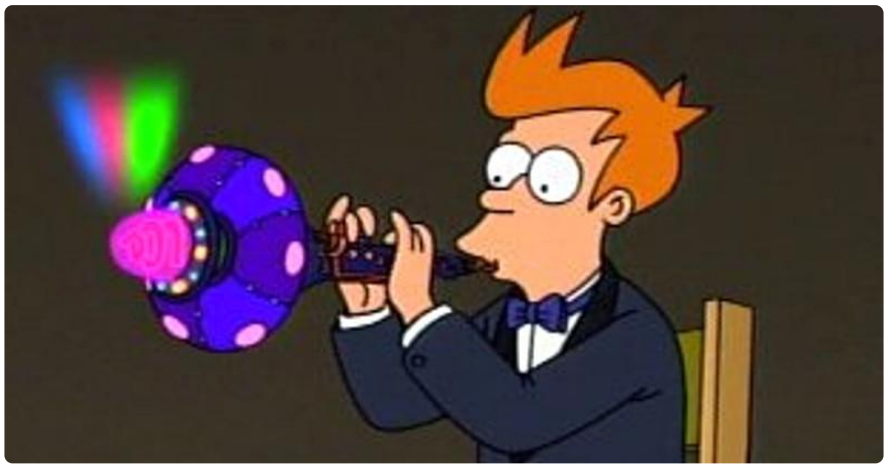
Created by Carter Nelson

https://learn.adafruit.com/circuit-playground-o-phonor

Last updated on 2023-08-29 04:18:54 PM EDT

Table of Contents

Overview



Remember the holophonor () from Futurama ()? It was (will be?) a musical instrument that would project holograms depending on the mood of the note played.

Being a futuristic instrument, the inner workings of the holophonor are unknown. However, the basic idea of being able to detect the frequency of musical notes is something we can possibly emulate. And that is what we'll explore in this guide using a fairly simple technique.

We'll also use the frequency detection to drive NeoPixels (our version of the holophonor) and also show how a TFT Gizmo () can be used to display the note information.

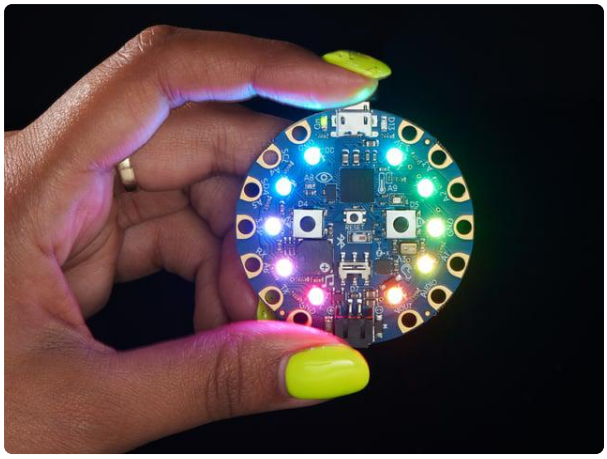
Which Circuit Playground?

The technique used here to estimate the audio frequency is fairly simple. It relies on constantly sampling the microphone and analyzing the results. Therefore, the more powerful the processor, the better.

We've found that this works best on the Circuit Playground Bluefruit. You can also run it on the Circuit Playground Express.

Parts

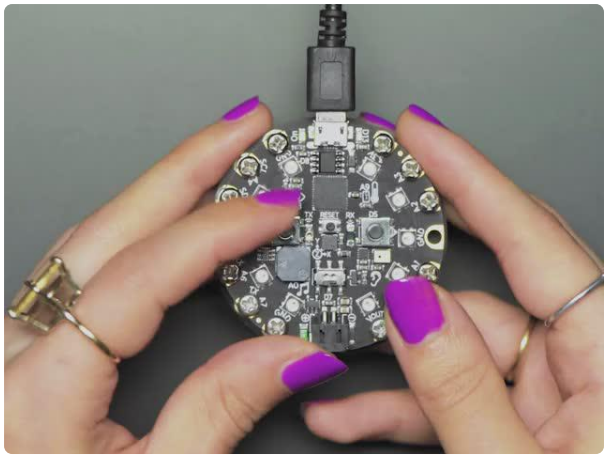
The following parts are used in this project:



Circuit Playground Bluefruit - Bluetooth Low Energy

Circuit Playground Bluefruit is our third board in the Circuit Playground series, another step towards a perfect introduction to electronics and programming. We've...

https://www.adafruit.com/product/4333



Circuit Playground TFT Gizmo - Bolt-on Display + Audio Amplifier

Extend and expand your Circuit Playground projects with a bolt on TFT Gizmo that lets you add a lovely color display in a sturdy and reliable fashion. This PCB looks just like a round...

https://www.adafruit.com/product/4367

USB cable - USB A to Micro-B

This here is your standard A to micro-B USB cable, for USB 1.1 or 2.0. Perfect for connecting a PC to your Metro, Feather, Raspberry Pi or other dev-board or...

https://www.adafruit.com/product/592

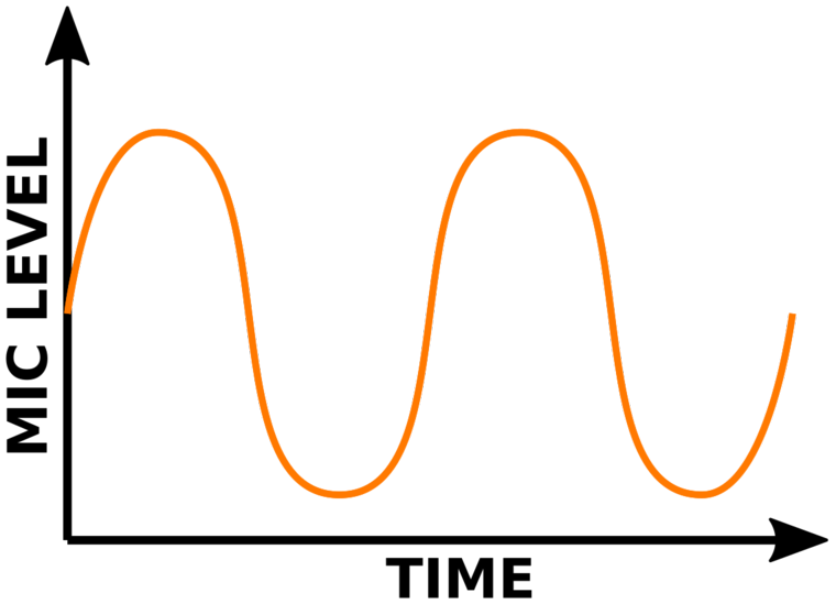
How It Works

The technique we use in this guide to determine the frequency is pretty straight forward. It's sometimes called "zero crossing" or "mean crossing" as we use the time between crossings of a zero or mean level to estimate the frequency.

Let's look at this in more detail.

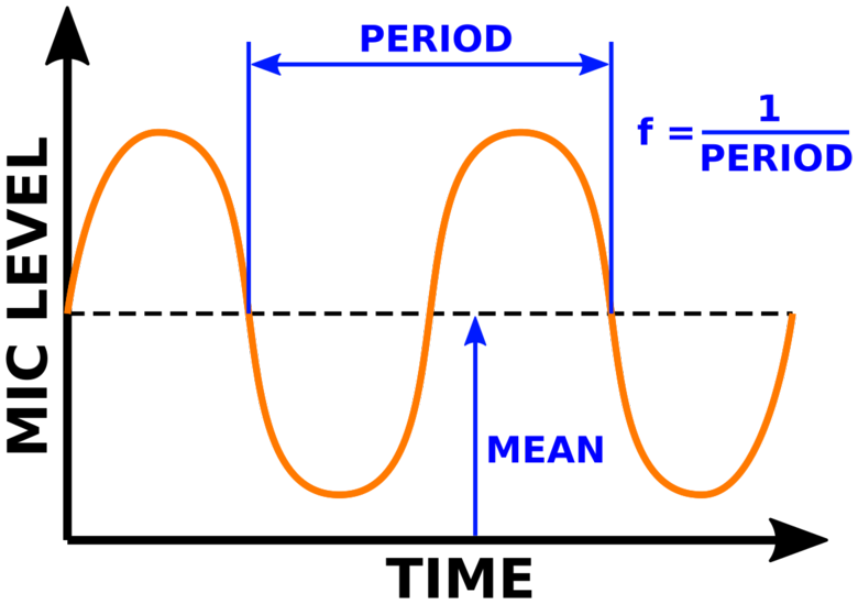
Pure Tone Basics

A fundamental assumption of this technique is that we are dealing with a sound which is a pure tone. That means it is a nice clean signal that just repeats over and over again at a single frequency. If we recorded such a signal with a microphone and plotted the results, it would look something like this:



This is an example of a signal with an offset, since the line is not crossing back and forth through the zero line on the graph. In this case, a zero level represents silence and all noises will have a positive value. So we end up with a signal oscillating back and forth around some offset.

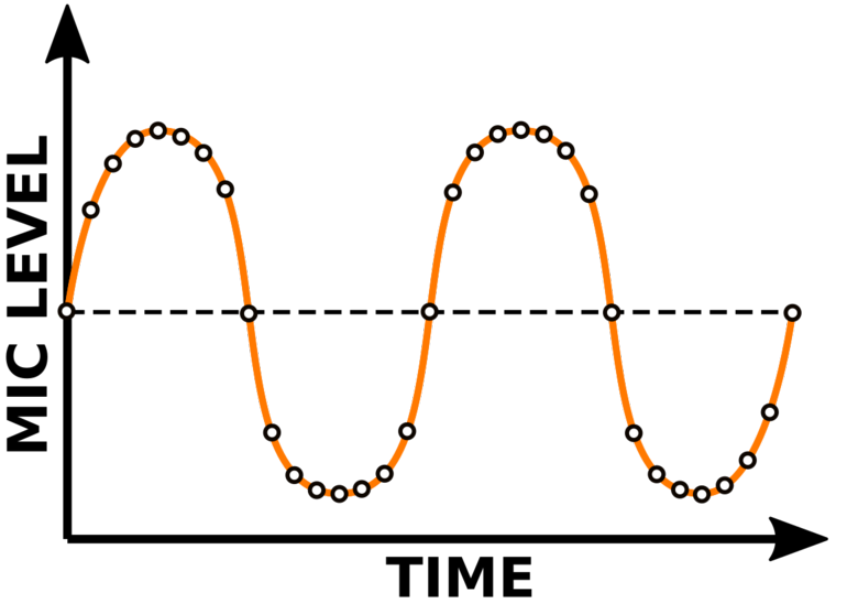
There are several key features to this signal as shown below. The offset mentioned above is called the MEAN.



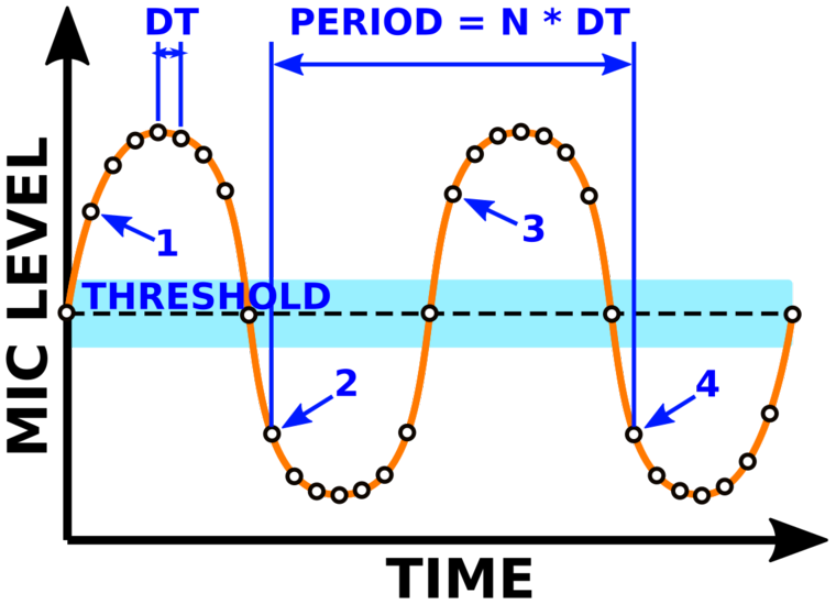
MEAN is just another word for average, and that is how this value can be determined - by averaging the entire sample. The PERIOD is the total time it takes to make one cycle of the pattern, which then repeats over and over. We are interested in the frequency, f, of this signal, which is simply the inverse of the PERIOD.

Computing Frequency From Mic Data

When we sample the microphone, we don't get the orange line above. Instead we get a bunch of individual data points, like the black circles below.



These will all fall on the orange line as shown. Just keep in mind that the microphone sample data is a series of discrete data points. We can then work with these data points to estimate the frequency.



Here's how the "mean crossing" technique works:

1. Find the first point which is greater than the MEAN value plus the THRESHOLD. Set a "has crossed" flag to True.
2. Find the first point after the "has crossed" flag has been set True that goes below the MEAN value. Store this location and clear the "has crossed" flag.
3. We now repeat the process, so this is the next point like 1.
4. And this is the next point like 2. Compute the delta between 2 and 4 and save it.

We do the above across the entire mic sample and then take an average of the deltas. The average delta is in terms of number of samples, N, in one cycle. The microphone sample rate sets the time, DT, between samples. So we can compute the PERIOD as simply N * DT. And the frequency of the mic data is then just the reciprocal of the PERIOD.

Frequency Basics

OK, let's actually implement the "mean crossing" technique and see if it works. We'll start simple and just print the computed results to the serial monitor. Here's the complete code:

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
import array
import board
import audiobusio

#---| User Configuration |-------------------------
SAMPLERATE = 16000
```

```
SAMPLES = 1024
THRESHOLD = 100
MIN_DELTAS = 5
DELAY = 0.2
#-----------------------------------------------------

# Create a buffer to record into
samples = array.array('H', [0] * SAMPLES)

# Setup the mic input
mic = audiobusio.PDMIn(board.MICROPHONE_CLOCK,
                       board.MICROPHONE_DATA,
                       sample_rate=SAMPLERATE,
                       bit_depth=16)

while True:
    # Get raw mic data
    mic.record(samples, SAMPLES)

    # Compute DC offset (mean) and threshold level
    mean = int(sum(samples) / len(samples) + 0.5)
    threshold = mean + THRESHOLD

    # Compute deltas between mean crossing points
    # (this bit by Dan Halbert)
    deltas = []
    last_xing_point = None
    crossed_threshold = False
    for i in range(SAMPLES-1):
        sample = samples[i]
        if sample > threshold:
            crossed_threshold = True
        if crossed_threshold and sample < mean:
            if last_xing_point:
                deltas.append(i - last_xing_point)
            last_xing_point = i
            crossed_threshold = False

    # Try again if not enough deltas
    if len(deltas) < MIN_DELTAS:
        continue

    # Average the deltas
    mean = sum(deltas) / len(deltas)

    # Compute frequency
    freq = SAMPLERATE / mean

    print("crossings: {}  mean: {}  freq: {} ".format(len(deltas), mean, freq))

    time.sleep(DELAY)
```

The part of the code that does the main work for the mean crossing estimation of frequency is this:

```
# Compute deltas between mean crossing points
    # (this bit by Dan Halbert)
    deltas = []
    last_xing_point = None
    crossed_threshold = False
    for i in range(SAMPLES-1):
        sample = samples[i]
        if sample &gt; threshold:
            crossed_threshold = True
        if crossed_threshold and sample &lt; mean:
            if last_xing_point:
```

```
            deltas.append(i - last_xing_point)
        last_xing_point = i
        crossed_threshold = False
```

Compare that to the explanation in the previous section. You can see how it is looping over the entire mic SAMPLES, checking for crossed_threshold and storing all of the d eltas.

Once that loop is complete, the actual frequency computation is straight forward. The deltas are averaged:

```
# Average the deltas
    mean = sum(deltas) / len(deltas)
```

And the frequency (freq) is then just:

```
# Compute frequency
    freq = SAMPLERATE / mean
```

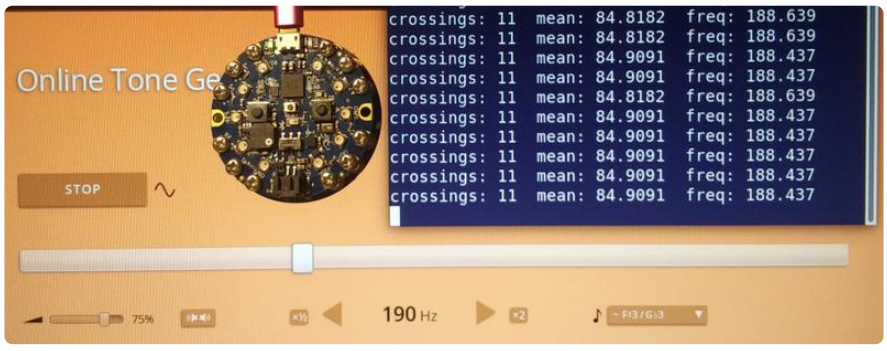
Tone Test

Here is a nice website that provides a pure tone generator we can use for testing:

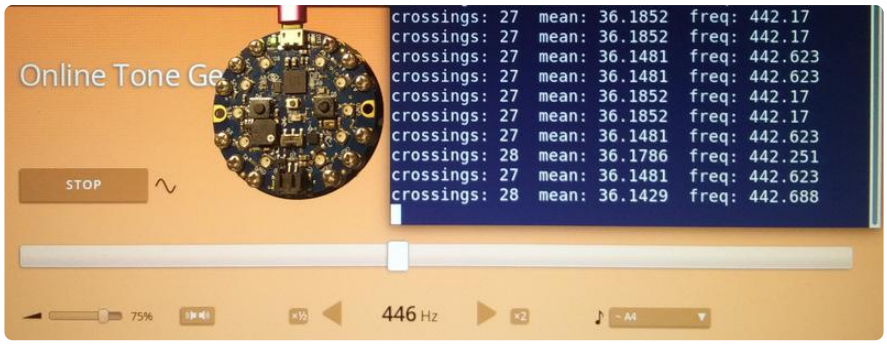
Online Tone Generator

Run the basic frequency example program above and watch the output on the serial monitor. Then use the website to generate some tones of various frequencies. Like this:

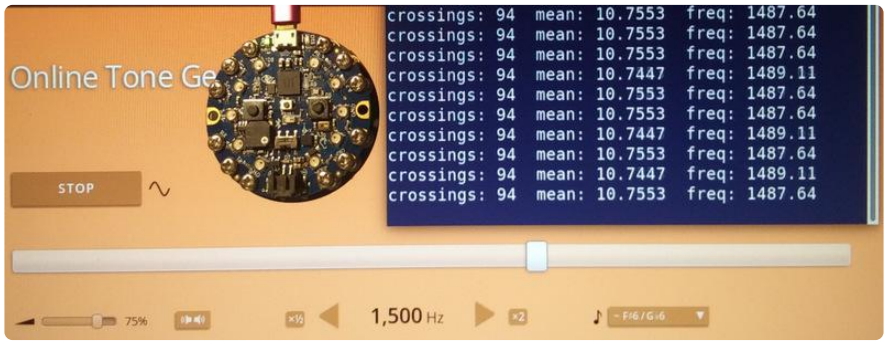
Here 190 is registering at about ~188:



Here 446 is registering at about ~442:



Here 1500 is registering at about ~1487:



The results are not exact, but pretty close. Not too bad for a technique this simple.

Now let's add some fun NeoPixel response based on the estimated frequency.

Frequency and NeoPixels

We'll look at determining musical notes in a bit. But for making the NeoPixels respond to the mic data, we can just work in terms of frequency. The idea is pretty simple - we'll set a LOW frequency and a HIGH frequency. The NeoPixels will then light up individually for any frequency in this range.

Here's the complete code:

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
import array
import board
import audiobusio
import simpleio
import neopixel

#---| User Configuration |--------------------------
SAMPLERATE = 16000
SAMPLES = 1024
THRESHOLD = 100
MIN_DELTAS = 5
DELAY = 0.2
```

```
FREQ_LOW = 520
FREQ_HIGH = 990
COLORS = (
    (0xFF, 0x00, 0x00) , # pixel 0
    (0xFF, 0x71, 0x00) , # pixel 1
    (0xFF, 0xE2, 0x00) , # pixel 2
    (0xAA, 0xFF, 0x00) , # pixel 3
    (0x38, 0xFF, 0x00) , # pixel 4
    (0x00, 0xFF, 0x38) , # pixel 5
    (0x00, 0xFF, 0xA9) , # pixel 6
    (0x00, 0xE2, 0xFF) , # pixel 7
    (0x00, 0x71, 0xFF) , # pixel 8
    (0x00, 0x00, 0xFF) , # pixel 9
)
#-----------------------------------------------------

# Create a buffer to record into
samples = array.array('H', [0] * SAMPLES)

# Setup the mic input
mic = audiobusio.PDMIn(board.MICROPHONE_CLOCK,
                       board.MICROPHONE_DATA,
                       sample_rate=SAMPLERATE,
                       bit_depth=16)

# Setup NeoPixels
pixels = neopixel.NeoPixel(board.NEOPIXEL, 10, auto_write=False)

while True:
    # Get raw mic data
    mic.record(samples, SAMPLES)

    # Compute DC offset (mean) and threshold level
    mean = int(sum(samples) / len(samples) + 0.5)
    threshold = mean + THRESHOLD

    # Compute deltas between mean crossing points
    # (this bit by Dan Halbert)
    deltas = []
    last_xing_point = None
    crossed_threshold = False
    for i in range(SAMPLES-1):
        sample = samples[i]
        if sample > threshold:
            crossed_threshold = True
        if crossed_threshold and sample < mean:
            if last_xing_point:
                deltas.append(i - last_xing_point)
            last_xing_point = i
            crossed_threshold = False

    # Try again if not enough deltas
    if len(deltas) < MIN_DELTAS:
        continue

    # Average the deltas
    mean = sum(deltas) / len(deltas)

    # Compute frequency
    freq = SAMPLERATE / mean

    print("crossings: {}  mean: {}  freq: {} ".format(len(deltas), mean, freq))

    # Show on NeoPixels
    pixels.fill(0)
    pixel = round(simpleio.map_range(freq, FREQ_LOW, FREQ_HIGH, 0, 9))
    pixels[pixel] = COLORS[pixel]
    pixels.show()
```

```
    time.sleep(DELAY)
```

This code is largely the same as the previous example. All that has been added is some NeoPixel output based on frequency. You can set the LOW and HIGH frequency and the NeoPixel COLORS at the top of the code in these lines:

```
FREQ_LOW = 520
FREQ_HIGH = 990
COLORS = (
    (0xFF, 0x00, 0x00) , # pixel 0
    (0xFF, 0x71, 0x00) , # pixel 1
    (0xFF, 0xE2, 0x00) , # pixel 2
    (0xAA, 0xFF, 0x00) , # pixel 3
    (0x38, 0xFF, 0x00) , # pixel 4
    (0x00, 0xFF, 0x38) , # pixel 5
    (0x00, 0xFF, 0xA9) , # pixel 6
    (0x00, 0xE2, 0xFF) , # pixel 7
    (0x00, 0x71, 0xFF) , # pixel 8
    (0x00, 0x00, 0xFF) , # pixel 9
)
```

The part of the code that does the actual NeoPixel lighting is at the bottom:

```
# Show on NeoPixels
    pixels.fill(0)
    pixel = round(simpleio.map_range(freq, FREQ_LOW, FREQ_HIGH, 0, 9))
    pixels[pixel] = COLORS[pixel]
    pixels.show()
```

It uses the simpleio.map_range() function to map the computed frequency to the range of NeoPixels. So the LOW frequency will light NeoPixel #0, the HIGH frequency will light NeoPixel #9, and any in between frequencies will light the others.

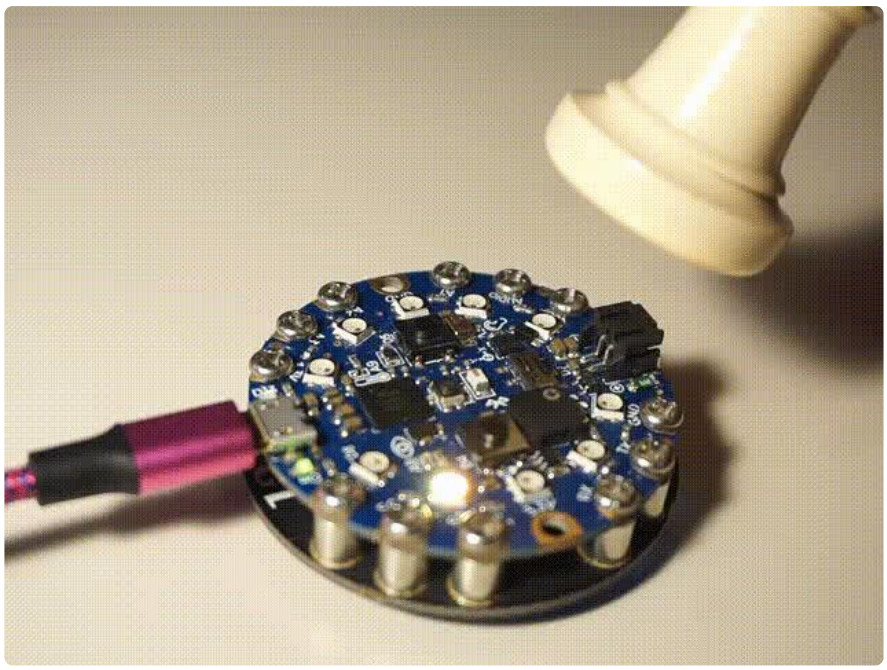
Circuit Playground-O-Phonor

We can use the above sketch to make our Circuit Playground-O-Phonor. As a substitute for a real holophonor, we'll use a very common flute like instrument called a recorder (). They look like this:

These come in different shapes and sizes with corresponding differences in their note ranges. You'll need to figure out what range of notes, and in what octave, your recorder produces. The code above has been setup to run from a 5th octave C (~520Hz) to a 5th octave B (~990Hz). If your recorder is different, change these two lines as needed:

```
FREQ_LOW = 520
FREQ_HIGH = 990
```

Once you've got that setup, run the code above and try playing notes on your recorder. You should get the NeoPixels to light up.



Musical Note Basics

So what about musical notes? The ones with letters, like ABCDEFG. Every Good Bird Does Fly. And all that.

Determining notes is essentially just a matter of mapping the computed frequency to specific notes. This mapping is well known and you can read more about it here:

Musical Notes and Frequencies

Here's the complete code that includes note detection:

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT
```

```
import time
import array
import board
import audiobusio

#---| User Configuration |-------------------------
SAMPLERATE = 16000
SAMPLES = 1024
THRESHOLD = 100
MIN_DELTAS = 5
DELAY = 0.2

#        octave = 1    2    3    4    5     6     7     8
NOTES = { "C" : (33,  65, 131, 262, 523, 1047, 2093, 4186),
          "D" : (37,  73, 147, 294, 587, 1175, 2349, 4699),
          "E" : (41,  82, 165, 330, 659, 1319, 2637, 5274),
          "F" : (44,  87, 175, 349, 698, 1397, 2794, 5588),
          "G" : (49,  98, 196, 392, 785, 1568, 3136, 6272),
          "A" : (55, 110, 220, 440, 880, 1760, 3520, 7040),
          "B" : (62, 123, 247, 494, 988, 1976, 3951, 7902)}
#-----------------------------------------------------

# Create a buffer to record into
samples = array.array('H', [0] * SAMPLES)

# Setup the mic input
mic = audiobusio.PDMIn(board.MICROPHONE_CLOCK,
                       board.MICROPHONE_DATA,
                       sample_rate=SAMPLERATE,
                       bit_depth=16)

while True:
    # Get raw mic data
    mic.record(samples, SAMPLES)

    # Compute DC offset (mean) and threshold level
    mean = int(sum(samples) / len(samples) + 0.5)
    threshold = mean + THRESHOLD

    # Compute deltas between mean crossing points
    # (this bit by Dan Halbert)
    deltas = []
    last_xing_point = None
    crossed_threshold = False
    for i in range(SAMPLES-1):
        sample = samples[i]
        if sample > threshold:
            crossed_threshold = True
        if crossed_threshold and sample < mean:
            if last_xing_point:
                deltas.append(i - last_xing_point)
            last_xing_point = i
            crossed_threshold = False

    # Try again if not enough deltas
    if len(deltas) < MIN_DELTAS:
        continue

    # Average the deltas
    mean = sum(deltas) / len(deltas)

    # Compute frequency
    freq = SAMPLERATE / mean

    print("crossings: {}  mean: {}  freq: {} ".format(len(deltas), mean, freq))

    # Find corresponding note
    for note in NOTES:
        for octave, note_freq in enumerate(NOTES[note]):
```

```
            if note_freq * 0.97 <= freq <= note_freq * 1.03:
                print("-"*10)
                print("NOTE = {}{}".format(note, octave + 1))
                print("-"*10)

    time.sleep(DELAY)
```

This also largely the same as our original code. The main new item is a table that maps NOTES at various OCTAVES to their corresponding frequency:

```
#        octave = 1    2    3    4    5     6     7     8
NOTES = { "C" : (33,  65, 131, 262, 523, 1047, 2093, 4186),
          "D" : (37,  73, 147, 294, 587, 1175, 2349, 4699),
          "E" : (41,  82, 165, 330, 659, 1319, 2637, 5274),
          "F" : (44,  87, 175, 349, 698, 1397, 2794, 5588),
          "G" : (49,  98, 196, 392, 785, 1568, 3136, 6272),
          "A" : (55, 110, 220, 440, 880, 1760, 3520, 7040),
          "B" : (62, 123, 247, 494, 988, 1976, 3951, 7902)}
```

And then once we have our frequency estimate, we just figure out where it is located in this table (coded to be plus or minus 3%):

```
# Find corresponding note
    for note in NOTES:
        for octave, note_freq in enumerate(NOTES[note]):
            if note_freq * 0.97 &lt;= freq &lt;= note_freq * 1.03:
                print("-"*10)
                print("NOTE = {}{}".format(note, octave + 1))
                print("-"*10)
```

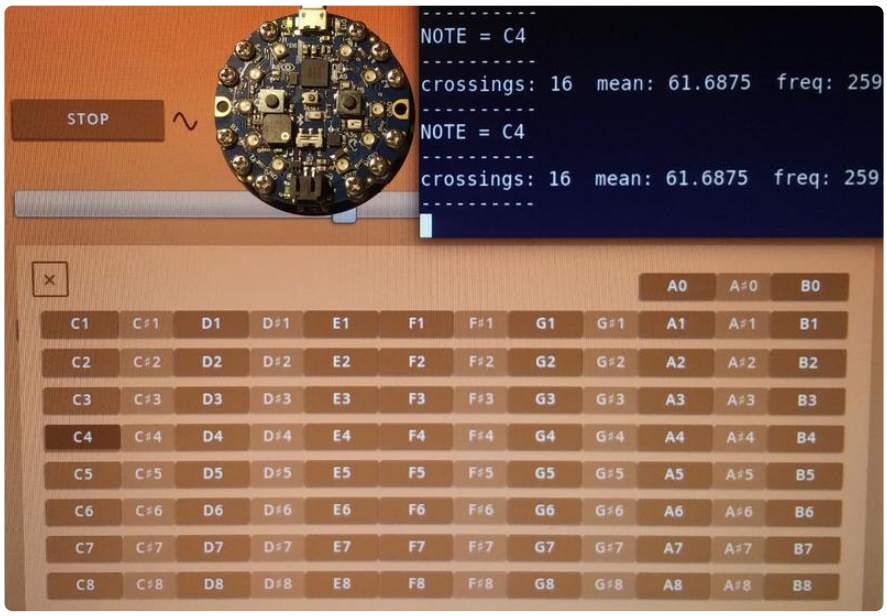
Notes Test

We can use the same website we used for the tones test to also test musical notes.

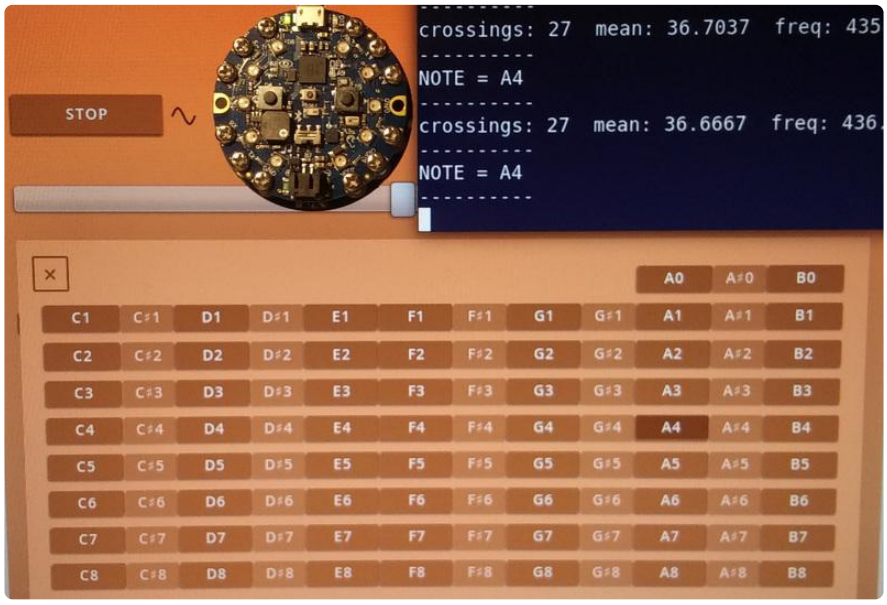
Musical Note Generator

The simple sketch above does not have entries for sharp and flat notes. So we'll test just the primary notes. To bring up a table of pre-defined musical note frequencies, click this button:

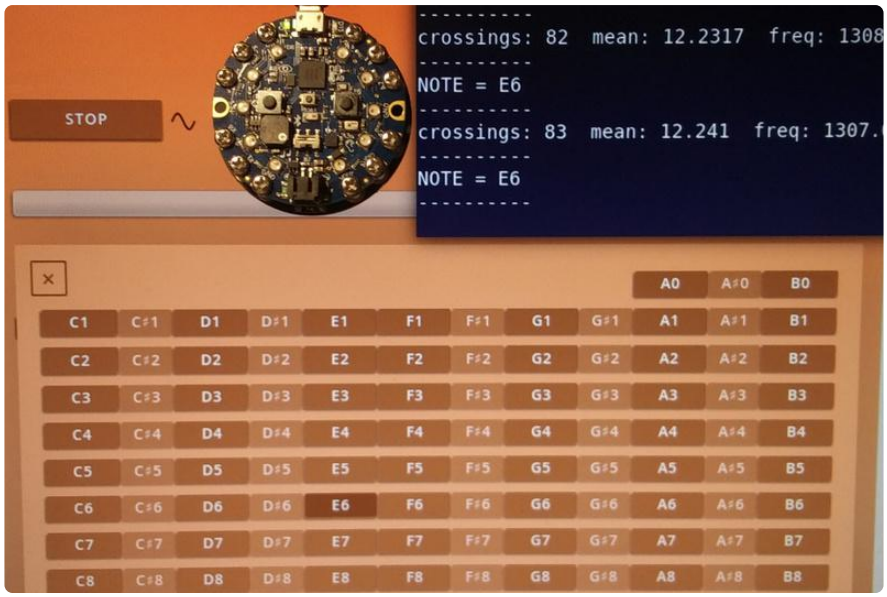
And then, with the above code running, try different notes. Here's C4:



Here's A4:



Here's E6:



Pretty good. As you get lower in octave (the number, like 4 and 6 above), you'll probably find it doesn't work quite as well. That's mainly due to the absolute frequency differences between the notes getting smaller in the lower octaves. And since our technique isn't super accurate, it has trouble distinguishing between consecutive note frequencies. It should get pretty close though.

Notes and TFT Gizmo

All of the examples so far have outputted to the serial monitor. If you have a TFT Gizmo (), we can use that instead to display information. That way you don't need to open a terminal to watch the serial output or be attached to a computer.

Let's adapt our basic notes program to output to the TFT Gizmo. Start by first going through this guide to get the TFT Gizmo setup and running:

Adafruit Circuit Playground TFT Gizmo

We'll also use some special fonts for the display. Here's a zip file containing the font files. Both of these font files were created from the Mono MMM 5 () font.

fonts.zip

Download that zip file, unzip the contents, and add the two font files to your CIRCUITPY drive which shows up when your board is connected to your computer via a known good data + power USB cable.

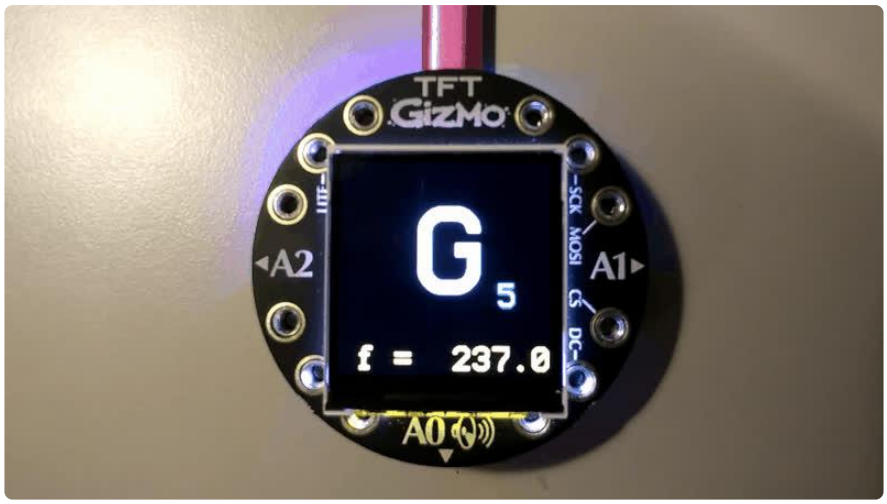
Here's the complete code for displaying note info to the TFT Gizmo:

```
# SPDX-FileCopyrightText: 2019 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
import array
import board
import busio
import audiobusio
import displayio
from adafruit_st7789 import ST7789
from adafruit_bitmap_font import bitmap_font
from adafruit_display_text import label

#---| User Configuration |-------------------------
SAMPLERATE = 16000
SAMPLES = 1024
THRESHOLD = 100
MIN_DELTAS = 5
DELAY = 0.2

#        octave = 1    2    3    4    5     6     7     8
NOTES = { "C" : (33,  65, 131, 262, 523, 1047, 2093, 4186),
          "D" : (37,  73, 147, 294, 587, 1175, 2349, 4699),
          "E" : (41,  82, 165, 330, 659, 1319, 2637, 5274),
          "F" : (44,  87, 175, 349, 698, 1397, 2794, 5588),
          "G" : (49,  98, 196, 392, 785, 1568, 3136, 6272),
          "A" : (55, 110, 220, 440, 880, 1760, 3520, 7040),
          "B" : (62, 123, 247, 494, 988, 1976, 3951, 7902)}
#-----------------------------------------------------

# Create a buffer to record into
samples = array.array('H', [0] * SAMPLES)

# Setup the mic input
mic = audiobusio.PDMIn(board.MICROPHONE_CLOCK,
                       board.MICROPHONE_DATA,
                       sample_rate=SAMPLERATE,
                       bit_depth=16)

# Setup TFT Gizmo
displayio.release_displays()

spi = busio.SPI(board.SCL, MOSI=board.SDA)
tft_cs = board.RX
tft_dc = board.TX
tft_backlight = board.A3

display_bus = displayio.FourWire(spi, command=tft_dc, chip_select=tft_cs)

display = ST7789(display_bus, width=240, height=240, rowstart=80,
                 backlight_pin=tft_backlight, rotation=180)

# Setup the various text labels
note_font = bitmap_font.load_font("/monoMMM_5_90.bdf")
note_text = label.Label(note_font, text="A", color=0xFFFFFF)
note_text.x = 90
note_text.y = 100

oct_font = bitmap_font.load_font("/monoMMM_5_24.bdf")
oct_text = label.Label(oct_font, text=" ", color=0x00FFFF)
oct_text.x = 180
oct_text.y = 150

freq_font = oct_font
freq_text = label.Label(freq_font, text="f = 1234.5", color=0xFFFF00)
freq_text.x = 20
freq_text.y = 220
```

```
# Add everything to the display group
splash = displayio.Group()
splash.append(note_text)
splash.append(oct_text)
splash.append(freq_text)
display.show(splash)

while True:
    # Get raw mic data
    mic.record(samples, SAMPLES)

    # Compute DC offset (mean) and threshold level
    mean = int(sum(samples) / len(samples) + 0.5)
    threshold = mean + THRESHOLD

    # Compute deltas between mean crossing points
    # (this bit by Dan Halbert)
    deltas = []
    last_xing_point = None
    crossed_threshold = False
    for i in range(SAMPLES-1):
        sample = samples[i]
        if sample > threshold:
            crossed_threshold = True
        if crossed_threshold and sample < mean:
            if last_xing_point:
                deltas.append(i - last_xing_point)
            last_xing_point = i
            crossed_threshold = False

    # Try again if not enough deltas
    if len(deltas) < MIN_DELTAS:
        continue

    # Average the deltas
    mean = sum(deltas) / len(deltas)

    # Compute frequency
    freq = SAMPLERATE / mean

    print("crossings: {}  mean: {}  freq: {} ".format(len(deltas), mean, freq))
    freq_text.text = "f = {:6.1f}".format(freq)

    # Find corresponding note
    for note in NOTES:
        for octave, note_freq in enumerate(NOTES[note]):
            if note_freq * 0.97 <= freq <= note_freq * 1.03:
                print("Note: {}{}".format(note, octave + 1))
                note_text.text = note
                oct_text.text = "{}".format(octave + 1)

    time.sleep(DELAY)
```

With this code running, try making some musical notes. Use either a musical instrument, like a recorder, or the tone generator website used previously.

It will display the note along with the octave (the subscript number). The actual frequency is also shown at the bottom:


TFT
GiZMO
G
5
f = 237.0
A2
A1
A0
SCK
MOSI
CS
DC