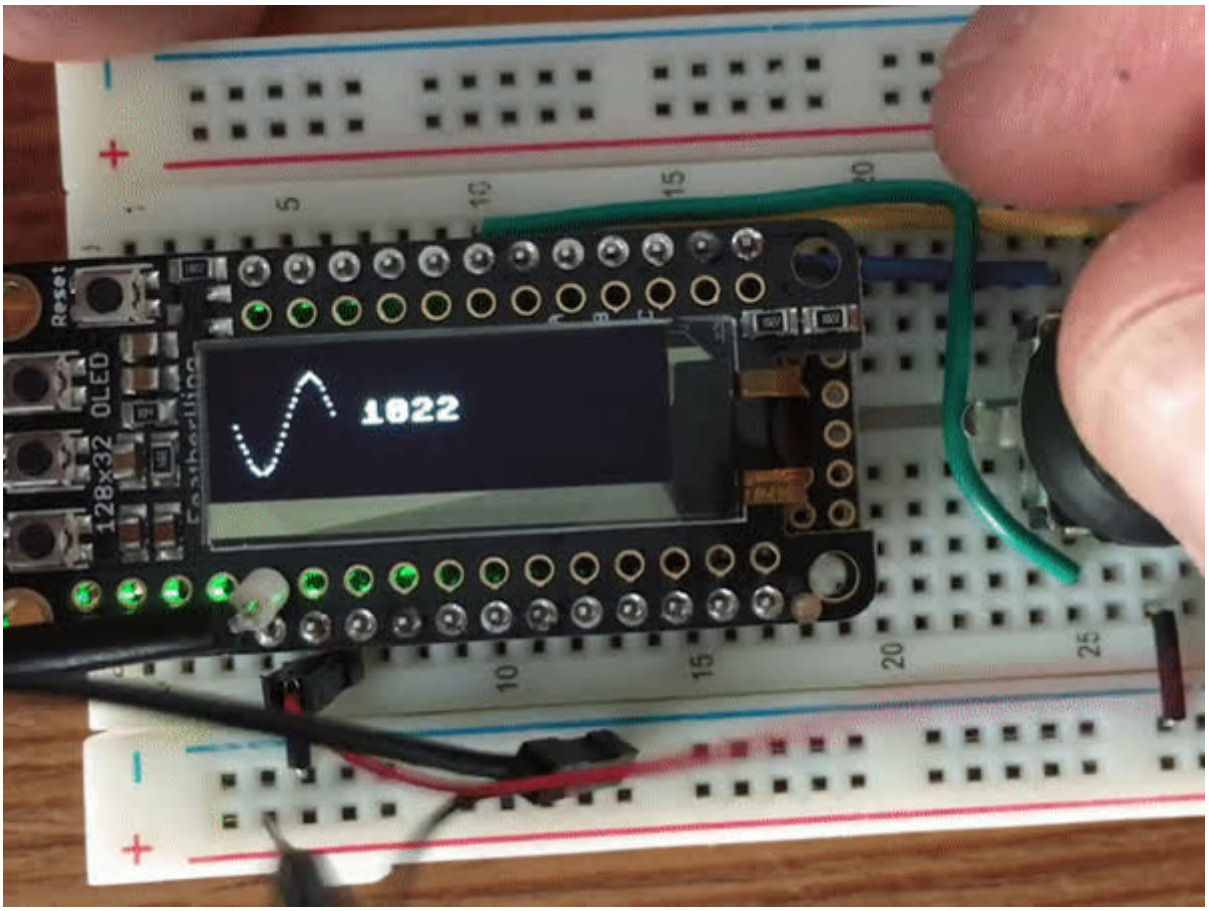
Feather Waveform Generator in CircuitPython

Created by Dave Astels

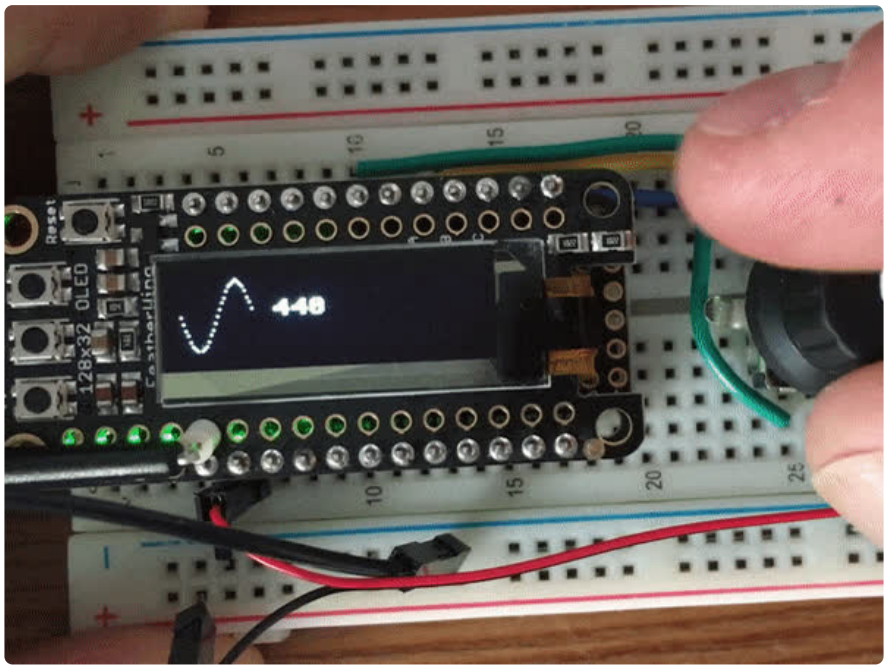


https://learn.adafruit.com/waveform-generator

Last updated on 2024-06-03 02:30:51 PM EDT

Table of Contents

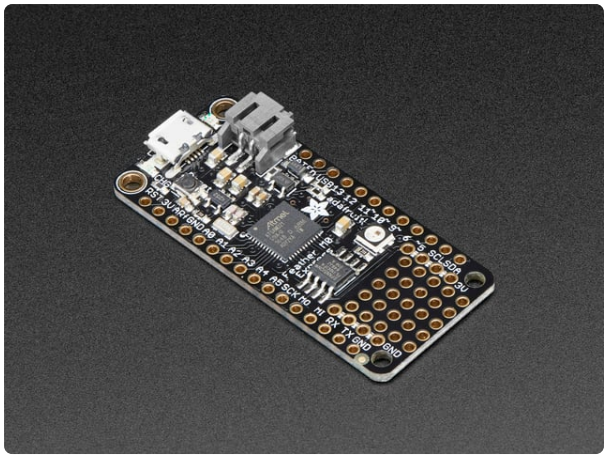
Overview



In this guide we'll take a few simple parts, add some not quite as simple code, and build an adjustable waveform generator (aka frequency generator). Using a rotary encoder push-switch, we'll select one of multiple waveforms, and using the rotation along with the three buttons on the OLED FeatherWing we'll adjust the frequency. We have great guides on the Feather M0 Express (https://adafru.it/B4b) and Feather M4 Express (https://adafru.it/CJN) as well as the OLED Featherwing (https://adafru.it/nek) to get you up to speed on this hardware.

A small speaker is useful to listen to the signal as a functional test. There's a nice one listed below but there are others; any 8 ohm speaker will work. The signal isn't very strong, so stick with a small one. You can easily power this by the USB port, but a LiPo battery will let you use it on it's own, untethered from a computer.

This project will work on either a Feather M0 Express or Feather M4 Express. Due to the additional performance and memory of the M4 board, capability of the code can be increased. This is pointed out in the code.

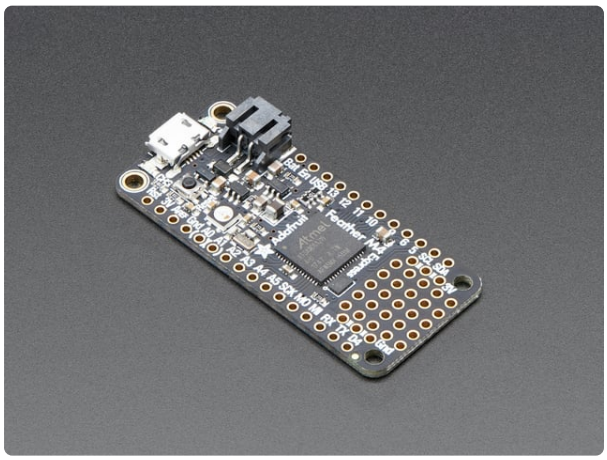


Adafruit Feather M0 Express

At the Feather M0's heart is an ATSAMD21G18 ARM Cortex M0+ processor, clocked at 48 MHz and at 3.3V logic, the same one used in the new

https://www.adafruit.com/product/3403

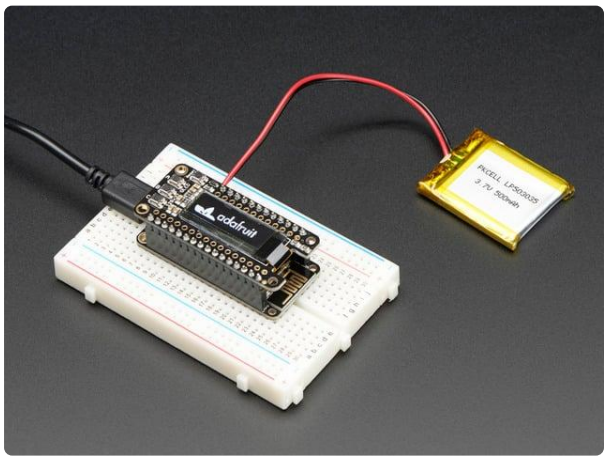
- or -



Adafruit Feather M4 Express - Featuring ATSAMD51

It's what you've been waiting for, the Feather M4 Express featuring ATSAMD51. This Feather is fast like a swift, smart like an owl, strong like a ox-bird (it's half ox,...

https://www.adafruit.com/product/3857



Adafruit FeatherWing OLED - 128x32 OLED Add-on For Feather

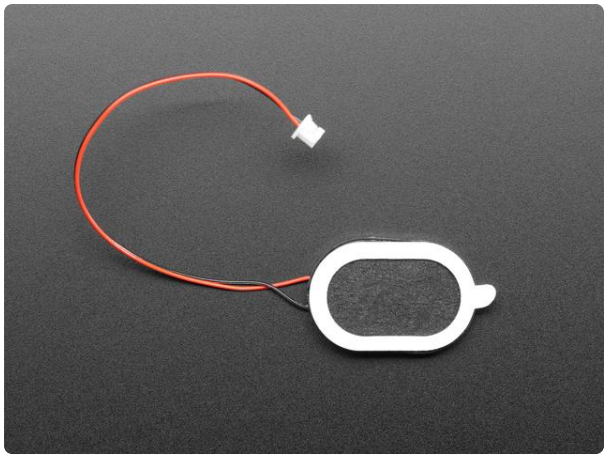
A Feather board without ambition is a Feather board without FeatherWings! This is the FeatherWing OLED: it adds a 128x32 monochrome OLED plus 3 user buttons to...

https://www.adafruit.com/product/2900

Rotary Encoder + Extras

This rotary encoder is the best of the best, it's a high-quality 24-pulse encoder, with detents and a nice feel. It is panel mountable for placement in a box, or you can plug it...

https://www.adafruit.com/product/377

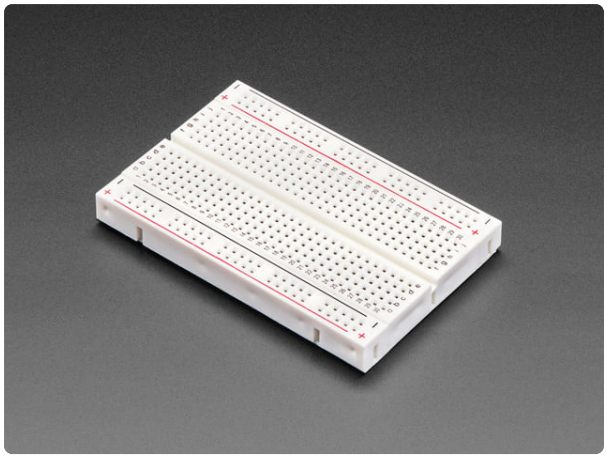


Mini Oval Speaker - 8 Ohm 1 Watt

Hear the good news! This wee speaker is a great addition to any audio project where you need 8 ohm impedance and 1W or less of power. We particularly like...

https://www.adafruit.com/product/3923

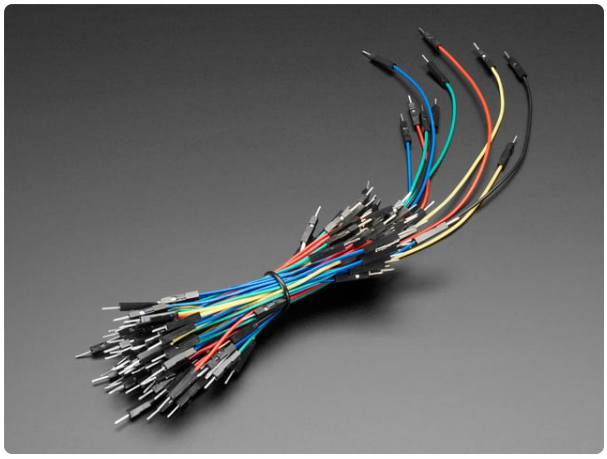
Other Parts and Tool



Half Sized Premium Breadboard - 400 Tie Points

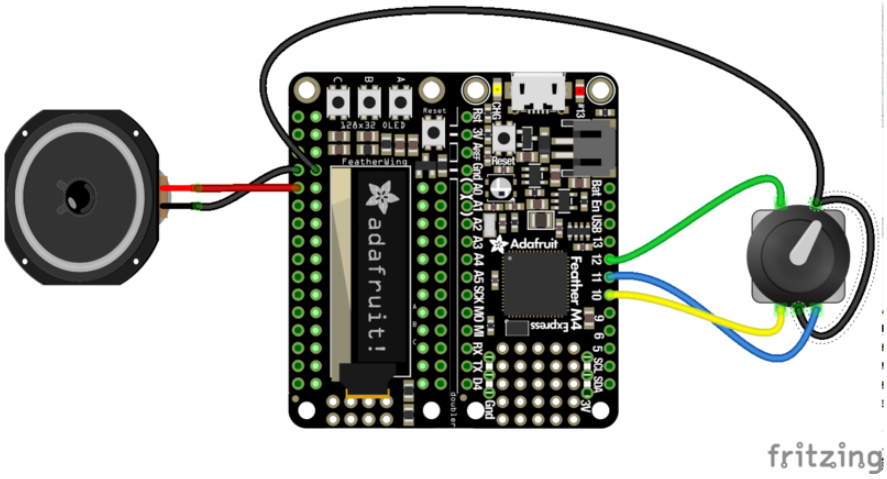
This is a cute, half-size breadboard with 400 tie points, good for small projects. It's 3.25" x 2.2" / 8.3cm x 5.5cm with a standard double-strip in the...

https://www.adafruit.com/product/64



Breadboarding wire bundle
75 flexible stranded core wires with stiff ends molded on in red, orange, yellow, green, blue, brown, black and white. These are a major improvement over the "box of bent...
https://www.adafruit.com/product/153

Hardware

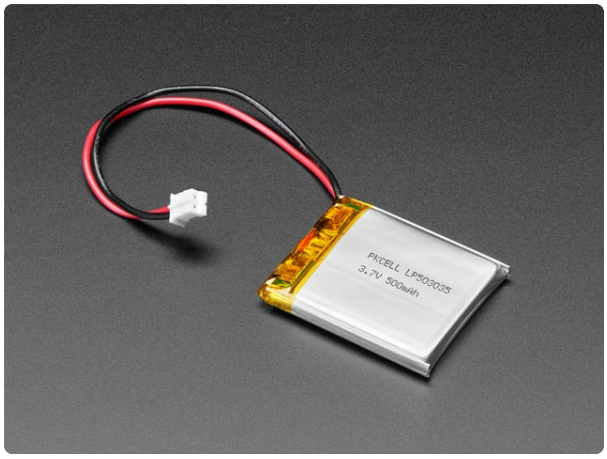


The hardware is pretty straight forward: A Feather M0 or M4 Express, an OLED FeatherWing, a speaker, and a rotary encoder. In the wiring diagram above we've put the Feather and wing on a feather doubler protoboard to show both, but you can use stacking headers to put the wing on top of the Feather. The Feather M4 is shown above, but the wiring is the same in either case.

Pushing the encoder will cycle through the different waveforms, while turning it will adjust the frequency. We'll use the A, B, and C buttons on the OLED wing to scale the frequency adjustment: A will change it by 1000 at a time, B by 100, and C by 10. Simply turning the encoder will adjust by 1 at a time.

A0, the true analog output pin on the Feather, will be used as the signal output.

You can power over USB or using a LiPo battery for a portable waveform generator!



[Lithium Ion Polymer Battery - 3.7v 500mAh]
Lithium-ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light, and powerful. The output ranges from 4.2V when completely charged to 3.7V. This...
https://www.adafruit.com/product/1578

Code

We'll be using CircuitPython for this project. Are you new to using CircuitPython? No worries, there is a full getting started guide here (https://adafru.it/cpy-welcome).

Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython. You can learn about Mu and its installation in this tutorial (https://adafru.it/ANO).

This project requires the latest CircuitPython 4.0 runtime and library bundle.

In `CIRCUITPY/lib` you will need to have:

- adafruit_bus_device
- adafruit_register
- adafruit_framebuf.mpy
- adafruit_ssd1306.mpy

- adafruit_debouncer.mpy

In addition to putting the project's python files on `CIRCUITPY`, you'll also need to copy the `font5x8.bin` from the project.

The basic approach to the user interface:

1. Check the inputs
2. Do something appropriate
3. Update the outputs
4. Go back to 1

More specifically:

1. Check the switches and encoder
2. If the encoder has been rotated change the frequency, possibly scaling based on OLED wing buttons
3. If the encoder was pushed, change the waveform
4. Update the display
5. Update the generated signal
6. Go back to 1

Since we are dealing with hardware buttons, it's a good idea to clean them up with a debouncer. We use the debouncer in the CircuitPython library bundle.

Using a SAMD21 board (in this case the Feather M0 Express) provides plenty of room for well structured CircuitPython code, but constrains how big of a sample buffer can be allocated. The code reflects this. Notes below indicate where the lower frequency limit and sample rate can be changed to take advantage of the SAMD51 chip on the Feather M4 Express.

In regards to well structured code, bundling the debouncer into a class is one example of this. This lets us create separate debouncer objects for the three wing buttons and the encoder's push switch in addition to putting all the code for the debouncer in one place.

To further separate code functionality, the display and signal generator can be placed into their own classes (which will be in separate files/modules). Everything needs to know what waveforms are supported, so that list can go into its own file/module as well.

The List of Shapes

Let's start with the simplest: the waveforms. This is imported by everything else (except the debouncer).

```
# SPDX-FileCopyrightText: 2018 Dave Astels for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
Signal generator wave shapes.

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""

SINE = 0
SQUARE = 1
TRIANGLE = 2
SAWTOOTH = 3
NUMBER_OF_SHAPES = 4
```

The Main Code

We'll structure things nicely and separate out areas of concern. There are two functions that handle changing the frequency and shape. Both of these let us ignore the details of how the changes occur when we write the main loop. It also puts those details in a clearly defined place. Do you want to change the frequency range? `change_frequency` is the logical place to do it. The change_frequency function is also responsible for limiting the frequency range. The lower limit of 150Hz prevents the sample buffer from becoming un-allocatably large on a Feather M0 board. If you are running this on a Feather M4 board you can safely decrease it to 10Hz.

```
def change_frequency(frequency, delta):
    return min(20000, max(150, frequency + delta))


def change_shape(shape):
    return (shape + 1) % shapes.NUMBER_OF_SHAPES
```

Similarly, we bundle the handling of the encoder into a function. It gets the new position from the encoder and computes the difference. That's the important part, as it determines how much to change the frequency by.

Note something else: The encoder and current position are passed in as parameters, and the new position and delta are returned. That new position gets stored by the main loop and passed back in next time. Doing this completely avoids having a global variable. Not only is this a good programming practice, but it also let's the code run significantly faster. This code is very user interface limited, so performance isn't much of a concern, but in a lot of embedded code it could be crucial. The reason is that referencing a global variable requires dictionary lookups, whereas using a local variable (or parameter) is a direct reference. A local/parameter also lives on the stack and not the heap, so there isn't allocation and garbage collection impact.

As you look through the code, you'll see that there are no global variables.

```
def get_encoder_change(encoder, pos):
    new_position = encoder.position
    if pos is None:
        return (new_position, 0)
    else:
        return (new_position, new_position - pos)
```

Now we have the main setup and loop code. This has been placed in a function. It starts by creating the variables it needs, initializing things, then entering the main loop.

The `Display` and `Generator` are created next. We'll look into these in detail later. Then the debouncers (see above) and rotary encoder are created. Next, a handful of state variables are allocated and initialized. Finally the display is given some initial content.

```
def make_debouncable(pin):
    switch_io = digitalio.DigitalInOut(pin)
    switch_io.direction = digitalio.Direction.INPUT
    switch_io.pull = digitalio.Pull.UP
    return switch_io

def run():
    display = Display()
    generator = Generator()
    button_a = Debouncer(make_debouncable(board.D9))
    button_b = Debouncer(make_debouncable(board.D6))
    button_c = Debouncer(make_debouncable(board.D5))
    encoder_button = Debouncer(make_debouncable(board.D12))
    encoder = rotaryio.IncrementalEncoder(board.D10, board.D11)

    current_position = None               # current encoder position
    change = 0                            # the change in encoder position
    delta = 0                             # how much to change the frequency by
    shape = shapes.SINE                   # the active waveform
    frequency = 440                       # the current frequency

    display.update_shape(shape)           # initialize the display contents
    display.update_frequency(frequency)
```

Now we have the main loop. First, all the debouncers are updated. This debounces the switches, giving us a stable value to use as well as detected rising and falling edges (switching being pushed or released). For the wing buttons, we just need button values and we can get that (logically enough) from the debouncer's `value` attribute. For the encoder switch, we want to know when it was pushed and don't care (in this case) when it's released or even what the value is. We can use the debouncer's `fell` attribute give us that information.

After updating the debouncers, we check the encoder to see if it has been rotated. If so, we use the values of the three wing buttons to scale the change by 10, 100, or 1000 (depending on which, if any, is pressed). If none are pressed, the change isn't scaled. Finally, if there was a change, the frequency is updated.

Next, if the encoder button was pushed since last time, the wave shape is changed.

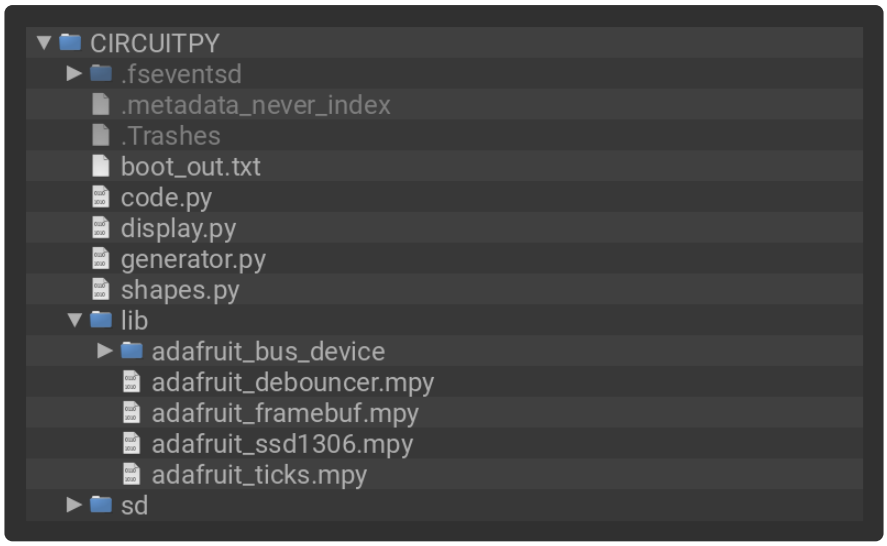
At the end of the loop, the display and generator are updated.

```
    while True:
        encoder_button.update()
        button_a.update()
        button_b.update()
        button_c.update()
        current_position, change = get_encoder_change(encoder, current_position)

        if change != 0:
            if not button_a.value:
                delta = change * 1000
            elif not button_b.value:
                delta = change * 100
            elif not button_c.value:
                delta = change * 10
            else:
                delta = change
            frequency = change_frequency(frequency, delta)

        if encoder_button.fell:
            shape = change_shape(shape)

        display.update_shape(shape)
        display.update_frequency(frequency)
        generator.update(shape, frequency)
```

Since the main code is in a function, all that's left is to execute it:

```
run()
```

Press 'Download Project Bundle' below, and copy the files over. After you've copied them over, it should look something like this:



Be sure that there isn't a code.py file on your Feather left from another project. If so, rename it or copy it off and delete the one on the Feather.

```
# SPDX-FileCopyrightText: 2018 Dave Astels for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
Main signal generator code.

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""

import rotaryio
import board
import digitalio
from display import Display
from adafruit_debouncer import Debouncer
from generator import Generator
import shapes



def change_frequency(frequency, delta):
    return min(20000, max(150, frequency + delta))


def change_shape(shape):
    return (shape + 1) % shapes.NUMBER_OF_SHAPES


def get_encoder_change(encoder, pos):
    new_position = encoder.position
    if pos is None:
        return (new_position, 0)
    else:
```

```
        return (new_position, new_position - pos)

def make_debouncable(pin):
    switch_io = digitalio.DigitalInOut(pin)
    switch_io.direction = digitalio.Direction.INPUT
    switch_io.pull = digitalio.Pull.UP
    return switch_io

def run():
    display = Display()
    generator = Generator()
    button_a = Debouncer(make_debouncable(board.D9))
    button_b = Debouncer(make_debouncable(board.D6))
    button_c = Debouncer(make_debouncable(board.D5))
    encoder_button = Debouncer(make_debouncable(board.D12))
    encoder = rotaryio.IncrementalEncoder(board.D10, board.D11)

    current_position = None               # current encoder position
    change = 0                            # the change in encoder position
    delta = 0                             # how much to change the frequency by
    shape = shapes.SINE                          # the active waveform
    frequency = 440                       # the current frequency

    display.update_shape(shape)           # initialize the display contents
    display.update_frequency(frequency)

    while True:
        encoder_button.update()
        button_a.update()
        button_b.update()
        button_c.update()
        current_position, change = get_encoder_change(encoder, current_position)

        if change != 0:
            if not button_a.value:
                delta = change * 1000
            elif not button_b.value:
                delta = change * 100
            elif not button_c.value:
                delta = change * 10
            else:
                delta = change
            frequency = change_frequency(frequency, delta)

        if encoder_button.fell:
            shape = change_shape(shape)

        display.update_shape(shape)
        display.update_frequency(frequency)
        generator.update(shape, frequency)


run()
```

The Display

The display functionality is encapsulated into the `Display` class which is instantiated (just a single instance) in the `run` function above. It's then used to show the user the current waveform and frequency.

The class has several instance variables: `I2C` and `SSD1306_I2C` instances for manipulating the physical display, and places to hold the current shape and frequency.

The constructor initializes the hardware and blanks the display.

```
    def __init__(self):
        self.i2c = busio.I2C(board.SCL, board.SDA)
        self.oled = adafruit_ssd1306.SSD1306_I2C(128, 32, self.i2c)

        self.oled.fill(0)
        self.oled.show()
```

There are methods to update the shape and frequency that are used from the main loop. These both operate similarly: they check that there actually was a change, update the appropriate instance variable, and call update to refresh the screen.

```
    def update_shape(self, shape):
        if shape != self.shape:
            self.shape = shape
            self.update()


    def update_frequency(self, frequency):
        if frequency != self.frequency:
            self.frequency = frequency
            self.update()
```

The `update` method and the `draw_*` methods do the work of putting pixels and text on the screen. This involves drawing the appropriate waveform and showing the selected frequency.

```
def draw_sine(self):
        for i in range(32):
            self.oled.pixel(i, int(math.sin(i/32 * math.pi * 2) * 16) + 16, 1)


    def draw_square(self):
        for i in range(16):
            self.oled.pixel(0, 32 - i, 1)
            self.oled.pixel(i, 31, 1)
            self.oled.pixel(31, i, 1)
            self.oled.pixel(15, 16 + i, 1)
            self.oled.pixel(15, i, 1)
            self.oled.pixel(16 + i, 0, 1)


    def draw_triangle(self):
        for i in range(8):
            self.oled.pixel(i, 16 + i * 2, 1)
            self.oled.pixel(8 + i, 32 - i * 2, 1)
            self.oled.pixel(16 + i, 16 - i * 2, 1)
            self.oled.pixel(24 + i, i * 2, 1)


    def draw_sawtooth(self):
        for i in range(16):
```

```
            self.oled.pixel(0, 16 + i, 1)
            self.oled.pixel(31, i, 1)
        for i in range(32):
            self.oled.pixel(i, 31 - i, 1)


    def update(self):
        self.oled.fill(0)
        if self.shape == shapes.SINE:
            self.draw_sine()
        elif self.shape == shapes.SQUARE:
            self.draw_square()
        elif self.shape == shapes.TRIANGLE:
            self.draw_triangle()
        elif self.shape == shapes.SAWTOOTH:
            self.draw_sawtooth()
        self.oled.text("{0}".format(self.frequency), 40, 10)
        self.oled.show()
```

```
# SPDX-FileCopyrightText: 2018 Dave Astels for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
Display code for signal generator.

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""

import math
import board
import busio
import adafruit_ssd1306
import shapes

class Display:
    """Manage the OLED Featherwing display"""


    i2c = None
    oled = None
    shape = None
    frequency = None

    def __init__(self):
        self.i2c = busio.I2C(board.SCL, board.SDA)
        self.oled = adafruit_ssd1306.SSD1306_I2C(128, 32, self.i2c)

        self.oled.fill(0)
        self.oled.show()


    def draw_sine(self):
        for i in range(32):
            self.oled.pixel(i, int(math.sin(i/32 * math.pi * 2) * 16) + 16, 1)


    def draw_square(self):
        for i in range(16):
```

```
            self.oled.pixel(0, 32 - i, 1)
            self.oled.pixel(i, 31, 1)
            self.oled.pixel(31, i, 1)
            self.oled.pixel(15, 16 + i, 1)
            self.oled.pixel(15, i, 1)
            self.oled.pixel(16 + i, 0, 1)


    def draw_triangle(self):
        for i in range(8):
            self.oled.pixel(i, 16 + i * 2, 1)
            self.oled.pixel(8 + i, 32 - i * 2, 1)
            self.oled.pixel(16 + i, 16 - i * 2, 1)
            self.oled.pixel(24 + i, i * 2, 1)


    def draw_sawtooth(self):
        for i in range(16):
            self.oled.pixel(0, 16 + i, 1)
            self.oled.pixel(31, i, 1)
        for i in range(32):
            self.oled.pixel(i, 31 - i, 1)


    def update(self):
        self.oled.fill(0)
        if self.shape == shapes.SINE:
            self.draw_sine()
        elif self.shape == shapes.SQUARE:
            self.draw_square()
        elif self.shape == shapes.TRIANGLE:
            self.draw_triangle()
        elif self.shape == shapes.SAWTOOTH:
            self.draw_sawtooth()
        self.oled.text("{0}".format(self.frequency), 40, 10, 1)
        self.oled.show()


    def update_shape(self, shape):
        if shape != self.shape:
            self.shape = shape
            self.update()


    def update_frequency(self, frequency):
        if frequency != self.frequency:
            self.frequency = frequency
            self.update()
```

The Generator

The generator is pretty simple. It uses the `audioio` library to play an array of samples.

The generator is in `generator.py` and implemented in the class: `Generator`.

There are a few instance variables to store the sample array, the sample player, and the current shape and frequency.

The constructor allocates and initializes the `AudioOut` driver that will be used to play the samples.

```
    def __init__(self):
        self.dac = audioio.AudioOut(board.A0)
```

If we jump down to the update method that's called from the main loop, we see that it first checks that there is a reason to update. If not, it immediately returns. This is an example of the guard clause pattern discussed in this guide (https://adafru.it/Czt).

If the frequency has changed, the sample array is reallocated: the size is the sample rate (64,000 in this case) divided by the frequency. The logic is that since we update the signal update 64,000 times per second (i.e. the sample rate) we have that many samples to use over the course of a second. For a 1 Hz signal we can use all 64,000 samples for one full cycle, but for a 1000 Hz signal we only have 64. That knowledge is encapsulated in the `length` function. The actual allocation of the array is done in the `reallocate` method. The sample rate of 32000 keeps the buffer small from being to large for a Feather M0 board. If running on a Feather M4 board you can increase it to 64000 without trouble.

```
def length(frequency):
    return int(32000 / frequency)


def reallocate(self, frequency):
    self.sample = array.array("h", [0] * self.length(frequency))
```

Back in the update method, the sample array is filled with a waveform determined by the shape parameter. Note that the sample array is always refilled. Either the frequency changed and is now a different size and thus the samples need to be regenerated, or the shape has changed and new samples are needed to reflect the new shape.

Finally the sample player is stopped and restarted with the new samples.

```
def update(self, shape, frequency):
        if shape == self.shape and frequency == self.frequency:
            return

        if frequency != self.frequency:
            self.reallocate(frequency)
            self.frequency = frequency

        self.shape = shape
        if shape == shapes.SINE:
            self.make_sine()
        elif shape == shapes.SQUARE:
            self.make_square()
        elif shape == shapes.TRIANGLE:
            self.make_triangle()
```

```
        elif shape == shapes.SAWTOOTH:
            self.make_sawtooth()

        self.dac.stop()
        self.dac.play(audiocore.RawSample(self.sample, channel_count=1,
sample_rate=64000), loop=True)
```

All that's left are the functions that generate the samples for each waveform. Sample values range from -32767 to +32767. These are the `2**15-1` values in the code The CircuitPython bytecode compiler will optimize these constant expressions to a constant value, so they don't have any runtime performance impact.

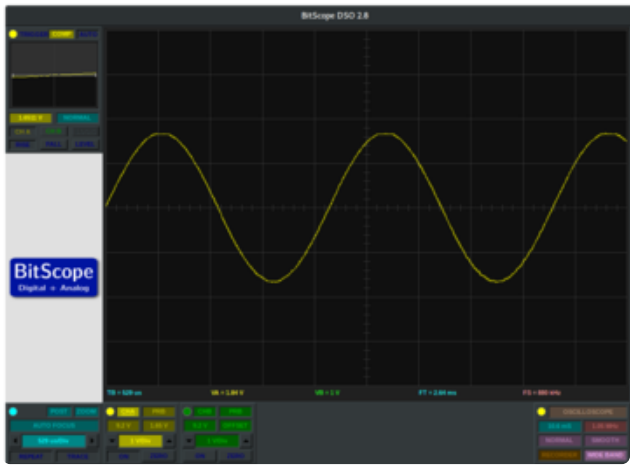
```
    def make_sine(self):
        l = len(self.sample)
        for i in range(l):
            self.sample[i] = min(2 ** 15 - 1, int(math.sin(math.pi * 2 * i / l) *
(2 ** 15)))


    def make_square(self):
        l = len(self.sample)
        half_l = l // 2
        for i in range(l):
            if i &lt; half_l:
                self.sample[i] = -1 * ((2 ** 15) - 1)
            else:
                self.sample[i] = (2 ** 15) - 1


    def make_triangle(self):
        l = len(self.sample)
        half_l = l // 2
        s = 0
        for i in range(l):
            if i &lt;= half_l:
                s = int((i / half_l) * (2 ** 16)) - (2 ** 15)
            else:
                s = int((1 - ((i - half_l) / half_l)) * (2 ** 16)) - (2 ** 15)
            self.sample[i] = min(2 ** 15 -1, s)


    def make_sawtooth(self):
        l = len(self.sample)
        for i in range(l):
            self.sample[i] = int((i / l) * (2 ** 16)) - (2 ** 15)
```

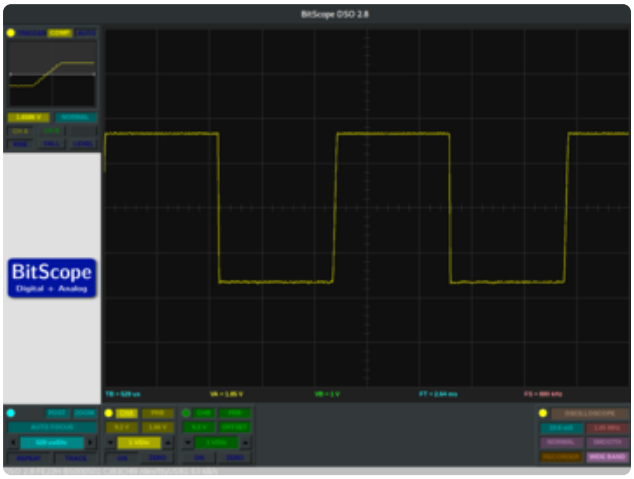
```
# SPDX-FileCopyrightText: 2018 Dave Astels for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
Outpout generator code for signal generator.

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""
```

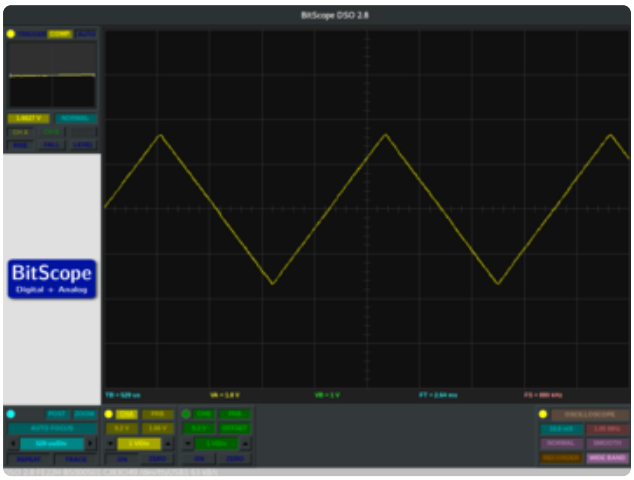
```
import math
import array
import board
import audioio
import audiocore
import shapes


def length(frequency):
    return int(32000 / frequency)



class Generator:

    sample = None
    dac = None
    shape = None
    frequency = None


    def __init__(self):
        self.dac = audioio.AudioOut(board.A0)


    def reallocate(self, frequency):
        self.sample = array.array("h", [0] * length(frequency))


    def make_sine(self):
        l = len(self.sample)
        for i in range(l):
            self.sample[i] = min(2 ** 15 - 1, int(math.sin(math.pi * 2 * i / l) * (2
** 15)))


    def make_square(self):
        l = len(self.sample)
        half_l = l // 2
        for i in range(l):
            if i < half_l:
                self.sample[i] = -1 * ((2 ** 15) - 1)
            else:
                self.sample[i] = (2 ** 15) - 1


    def make_triangle(self):
        l = len(self.sample)
        half_l = l // 2
        s = 0
        for i in range(l):
            if i <= half_l:
                s = int((i / half_l) * (2 ** 16)) - (2 ** 15)
            else:
                s = int((1 - ((i - half_l) / half_l)) * (2 ** 16)) - (2 ** 15)
            self.sample[i] = min(2 ** 15 -1, s)



    def make_sawtooth(self):
        l = len(self.sample)
        for i in range(l):
            self.sample[i] = int((i / l) * (2 ** 16)) - (2 ** 15)


    def update(self, shape, frequency):
        if shape == self.shape and frequency == self.frequency:
            return
```

```
        if frequency != self.frequency:
            self.reallocate(frequency)
            self.frequency = frequency

        self.shape = shape
        if shape == shapes.SINE:
            self.make_sine()
        elif shape == shapes.SQUARE:
            self.make_square()
        elif shape == shapes.TRIANGLE:
            self.make_triangle()
        elif shape == shapes.SAWTOOTH:
            self.make_sawtooth()

        self.dac.stop()
        self.dac.play(audiocore.RawSample(self.sample, channel_count=1,
sample_rate=64000), loop=True)
```

Operation

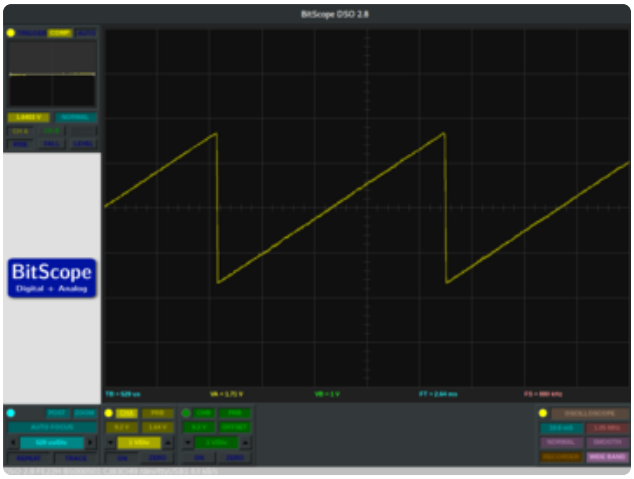




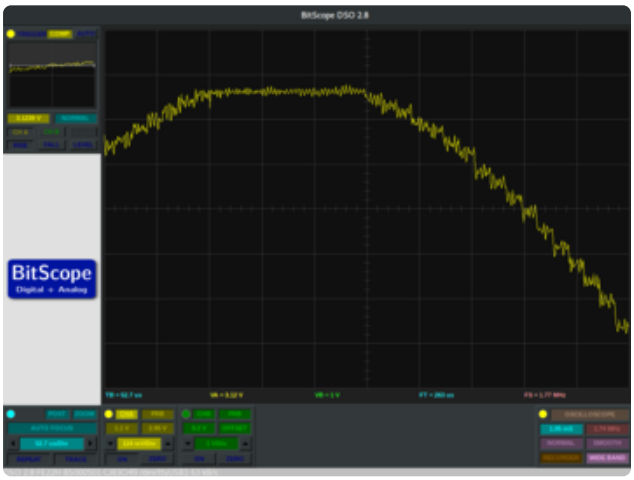


Using an oscilloscope, we can look at the output signal.

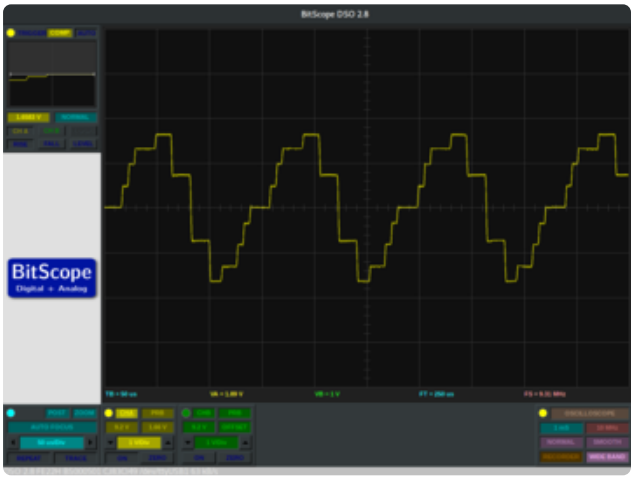
On the left are the four waveforms that can be generated. All are at 440 Hz. All four look relatively nice and smooth.



However, if we zoom in on the 440Hz sine wave we can see that it's not a smooth curve. Rather it's quite jaggy and noisy.



If the frequency is increased to 8440 the sine curve has degenerated into something very unlike a sine wave. This is because the sample rate is fixed, so as the frequency increases there are fewer samples available for a full cycle. That means fewer points on the signal, and that means a chunkier curve.



The easy answer to this is just to increase the sample rate. That, however, is limited by the speed of the underlying hardware. A more significant problem is the sample array. A higher sample rate means a bigger sample array at lower frequencies.

This is where the Feather M4 has an advantage over a Feather M0, it's faster and has more memory for more samples.

Speaker

You can hook the speaker to the output to hear the sounds of the various waveforms you generate. Without an amplifier, the volume is fixed and not very loud.

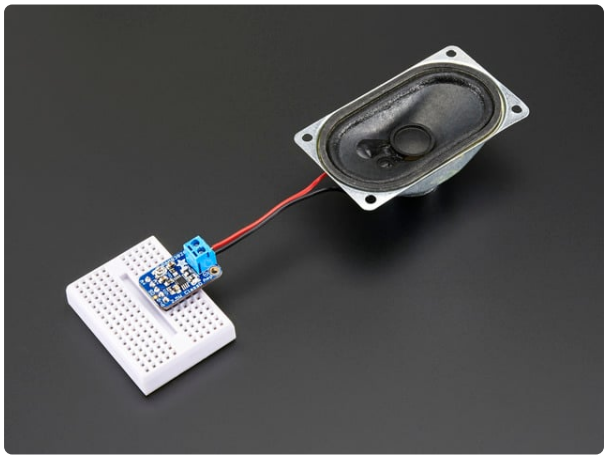
If you use a speaker with a JST connector on the end, use jumper wires to plug into the connector and the other end will plug into the breadboard easily.

Going Further

There are a variety of further steps you could take with this project that would make it even more capable. We'll look at a few here.

Amplifier

Adding a class D breakout would provide a stronger output signal. This breakout would do the job nicely. It's the amplifier used in the HalloWing (http://adafru.it/3900) and similar to the one in the Circuit Playground Express (http://adafru.it/3333).



Adafruit Mono 2.5W Class D Audio Amplifier - PAM8302
This super small mono amplifier is surprisingly powerful - able to deliver up to 2.5 Watts into 4-8 ohm impedance speakers. Inside the miniature chip is a class D controller, able to...
https://www.adafruit.com/product/2130

Low Pass Filter

Adding a low pass filter on the output to filter out some of the high frequency digital artifacts would produce a cleaner output signal.

More Waveforms

While the four provided waveforms are the standard ones, it could be useful to add more. Sine based signals with different sets of harmonics would be one possibility.

The current sawtooth wave ramps up slowly then drops. The opposite is also often present in signal generators: slowly ramping down before jumping up.

Waveforms as Classes

This would be an interesting as an exercise, and quite useful if you wanted to add to the selection of waveforms. Each waveform would be implemented by a separate class, possibly with a base class. Each of these classes would have methods to generate (x, y) pairs that define how to draw the thumbnail of the wave, and a method to generate the sample array.

There would need to be some way of storing the instances, and selecting from among them. This might require the Feather M4 due to memory.

Adjustable Duty Cycle

Currently square and triangle waves have a 50% duty cycle. Making that adjustable would be extremely useful. In fact, the sawtooth could be removed as a separate waveform (possibly completely, but probably just the implementation since it's common enough that you might want to keep it as a selection) since it can be considered to be a triangle wave with an extreme duty cycle (0% or 100%).

Dynamic Sample Rate

The current design uses a fixed sample rate and varies the number of samples based on frequency. A result of this is that high frequency signals have fewer samples and thus are much choppier, and have far more digital artifacts.

By keeping the number of samples fixed we can get good quality signals for all frequencies. To support that we need to vary the sample rate based on frequency.