# Table of Contents

## Overview
- Parts

## Build the 16-Step Drum Sequencer
- Solder KB2040 Pins
- Prepare the AW9523
- Jumper Soldering on AW9523
- Play/Pause Switch
- Step Switches
- Step Switch Breakout Soldering
- Breadboard Wiring
- Switch Wiring
- Switch Wiring Continued
- Play Button and Switch LED Wiring
- Top Row Switches
- Add the Display
- Add Display
- Add Rotary Encoder

## CircuitPython
- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

## Code the 16-Step Drum Sequencer
- Text Editor
- Download the Project Bundle
- Upload the Code and Libraries to the KB RP2040
- How it Works
- MIDI, Drum, Pattern Setup
- Functions
- print_sequence()
- Set Initial LEDs
- Display Setup
- Main Loop

## Use the 16-Step Drum Sequencer
- Software Instrument Track
- Drum Kit Select
- Play!
- Mix, Effects, EQ
Overview

Build a Roland 808-style step sequencer to trigger multiple drum tracks in your digital audio workstation (DAW), such as GarageBand or Ableton, over USB MIDI. All built in CircuitPython, and easy to customize.

Parts

Adafruit KB2040 - RP2040 Kee Boar Driver
A wild Kee Boar appears! It’s a shiny KB2040! An Arduino Pro Micro-shaped board for Keebs with RP2040. (#keeblife 4 evah) A lot of folks like using Adafruit... https://www.adafruit.com/product/5302
Adafruit AW9523 GPIO Expander and LED Driver Breakout

Expand your project possibilities, with the Adafruit AW9523 GPIO Expander and LED Driver Breakout - a cute and powerful I2C expander with a lot of tricks up its...

https://www.adafruit.com/product/4886

Quad Alphanumeric Display - Red 0.54" Digits w/ I2C Backpack

Display, elegantly, 012345678 or 9! Gaze, hypnotized, at ABCDEFGHIJKLMNOP - well it can display the whole alphabet. You get the point. This is a nice, bright alphanumeric display that...

https://www.adafruit.com/product/1911
A couple of months ago we started carrying packs of colorful step switches reminiscent of the
https://www.adafruit.com/product/5631

1 x **Large Solderless Breadboard**
Large Solderless Breadboard

1 x **Hook-up Wire Spool Set**
Hook-up Wire Spool Set - 22AWG Solid Core - 10 x 25ft

6 x **Step Switch w LED**
Three Pack PB86

1 x **Colorful 12mm Square Tactile Button Switch Assortment**
15 pack

1 x **Pink and Purple Woven USB A to USB C Cable**
1 meter long

https://www.adafruit.com/product/443
https://www.adafruit.com/product/3174
https://www.adafruit.com/product/5499
https://www.adafruit.com/product/1010
https://www.adafruit.com/product/5153
1 x STEMMA QT / Qwiic JST SH 4-pin Cable
100mm Long

2 x STEMMA QT / Qwiic JST SH 4-Pin Cable
50mm Long

Build the 16-Step Drum Sequencer
Solder KB2040 Pins
Solder header pins to the Kee Boar (similar to this Feather configuration ()), then press it into the breadboard as shown in the diagram.
Prepare the AW9523

Jumper Soldering on AW9523

Solder header pins onto the AW9523. Connect it to the KB2040 with a short STEMMA QT cable, then press it into the breadboard as shown.

Solder both address jumpers on the AW9523 in order to prevent the LEDs from all lighting at startup. Soldering the jumpers sets the I2C address on the board to \textbf{0x5B} which sets the chip's initial boot state.
Play/Pause Switch
Add the tactile switch to the lower right corner of the board as shown. You’ll use this to start and stop your sequencer.

You’ll wire this to GND and a KB 2040 IO pin later.

Step Switches
Each PB-86 step switch has pins for the switch (both Normally Open and Normally Closed) as well as the LED anode and cathode, and these pins won’t work on a breadboard, due to their arrangement. So, we made a convenient breakout PCB set for it.

You can see the full pinout info here.
Step Switch Breakout
Soldering

Snap off the breakout PCBs.

Then, cut your header strip into five-pin lengths.

Place the header pins into the breadboard as shown, then solder the breakout PCBs onto them.
Place the switch into the PCB as shown and then solder all six legs.
Repeat this process for the remaining fifteen switches, then you'll add them to the breadboard and start wiring them as shown in the steps below.

**Breadboard Wiring**

**Switch Wiring**
Start with the GND and LED + connections to the breadboard power rails -- making short wire "staples" helps to keep things neat.
Switch Wiring Continued

Wire the KB 2040 GND and 3v3 pins to the breadboard power rails as shown. Remember to connect all of the breadboard power rails to each other.

Wire the bottom row of switches as shown here, connecting their normally open (N.O.) switch contacts to the GPIO pins of the KB 2040.

Be methodical and careful about it, since there is quite a lot of wiring it can become confusing to troubleshoot later!
Play Button and Switch LED Wiring

Connect the play button as shown here.

Wire the LED- pins from the switches to their associated pins on the AW9523. These will drive the current for the LEDs.

Top Row Switches

Next, you can add the top row of switches and wire them the same way you did with the bottom row, but all going to their own pins on the KB 2040 and AW9523.
Add Display

Plug the 14-segment LED backpack into the AW9523 with a short STEMMA QT cable. Also, plug in a longer cable at this time that will connect to the rotary encoder breakout.

Due to the position of the STEMMA QT connectors, the board can't lay flat on the breadboard, so you'll mount it on the overhang of the breadboard.

In order to secure it to the board, create some "staples" with pairs of pin headers modified as shown. You could alternatively use double-stick foam tape, hot glue, or another method.
Add Rotary Encoder

Connect the rotary encoder to the STEMMA QT cable coming from the 14-segment LED backpack.

In order to secure it to the breadboard, solder pin headers and press it into the breadboard as shown.
Your build is complete -- this is a good time to use a continuity tester to double-check that you don't have any shorts from power to ground. Next, move on to coding the Drum Sequencer!

---

**CircuitPython**

*CircuitPython* is a derivative of *MicroPython* designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the CIRCUITPY drive to iterate.

**CircuitPython Quickstart**

Follow this step-by-step to quickly get CircuitPython running on your board.

Download the latest version of CircuitPython for this board via circuitpython.org
Click the link above to download the latest CircuitPython UF2 file.

Save it wherever is convenient for you.

To enter the bootloader, hold down the BOOT/BOOTSEL button (highlighted in red above), and while continuing to hold it (don't let go!), press and release the reset button (highlighted in blue above). Continue to hold the BOOT/BOOTSEL button until the RPI-RP2 drive appears!

If the drive does not appear, release all the buttons, and then repeat the process above.

You can also start with your board unplugged from USB, press and hold the BOOTSEL button (highlighted in red above), continue to hold it while plugging it into USB, and wait for the drive to appear before releasing the button.

A lot of people end up using charge-only USB cables and it is very frustrating! Make sure you have a USB cable you know is good for data sync.
You will see a new disk drive appear called RPI-RP2.

Drag the adafruit_circuitpython_etc.uf2 file to RPI-RP2.

The RPI-RP2 drive will disappear and a new disk drive called CIRCUITPY will appear.

That's it, you're done! :)

Safe Mode

You want to edit your code.py or modify the files on your CIRCUITPY drive, but find that you can't. Perhaps your board has gotten into a state where CIRCUITPY is read-only. You may have turned off the CIRCUITPY drive altogether. Whatever the reason, safe mode can help.
Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode bypasses any code in boot.py (where you can set CIRCUITPY read-only or turn it off completely). Second, it does not run the code in code.py. And finally, it does not automatically soft-reload when data is written to the CIRCUITPY drive.

Therefore, whatever you may have done to put your board in a non-interactive state, safe mode gives you the opportunity to correct it without losing all of the data on the CIRCUITPY drive.

**Entering Safe Mode**

To enter safe mode when using CircuitPython, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 1000ms. On some boards, the onboard status LED (highlighted in green above) will blink yellow during that time. If you press reset during that 1000ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

**In Safe Mode**

If you successfully enter safe mode on CircuitPython, the LED will intermittently blink yellow three times.

If you connect to the serial console, you’ll find the following message.

```
Auto-reload is off.
Running in safe mode! Not running saved code.
CircuitPython is in safe mode because you pressed the reset button during boot.
Press again to exit safe mode.
Press any key to enter the REPL. Use CTRL-D to reload.
```

You can now edit the contents of the CIRCUITPY drive. Remember, your code will not run until you press the reset button, or unplug and plug in your board, to get out of safe mode.

**Flash Resetting UF2**

If your board ever gets into a really weird state and doesn't even show up as a disk drive when installing CircuitPython, try loading this 'nuke' UF2 which will do a 'deep
clean' on your Flash Memory. You will lose all the files on the board, but at least you'll be able to revive it! After loading this UF2, follow the steps above to re-install CircuitPython.

Download flash erasing "nuke" UF2

**Code the 16-Step Drum Sequencer**

**Text Editor**

Adafruit recommends using the Mu editor for editing your CircuitPython code. You can get more info in [this guide](https://example.com).

Alternatively, you can use any text editor that saves simple text files.

**Download the Project Bundle**

Your project will use a specific set of CircuitPython libraries, and the code.py file. To get everything you need, click on the Download Project Bundle link below, and uncompress the .zip file.

Drag the contents of the uncompressed bundle directory onto your KeeBoar board's CIRCUITPY drive, replacing any existing files or directories with the same names, and adding any new ones that are necessary.

**Upload the Code and Libraries to the KB RP2040**

After downloading the Project Bundle, plug your KB2040 into the computer USB port. You should see a new flash drive appear in the computer's File Explorer or Finder (depending on your operating system) called CIRCUITPY. Unzip the folder and copy the following items to the KB2040's CIRCUITPY drive.

- lib folder
- code.py

```python
# SPDX-FileCopyrightText: 2022 John Park for Adafruit Industries
#
# SPDX-License-Identifier: MIT
# Drum Trigger Sequencer 2040
# Based on code by Tod Kurt @todbot https://github.com/todbot/picostepseq
# Uses General MIDI drum notes on channel 10
```

©Adafruit Industries
# Range is note 35/B0 - 81/A4, but classic 808 set is defined here

```python
import time
from adafruit_ticks import ticks_ms, ticks_diff, ticks_add
import board
from digitalio import DigitalInOut, Pull
import keypad
import adafruit_aw9523
import usb_midi
from adafruit_seesaw import seesaw, rotaryio, digitalio
from adafruit_debouncer import Debouncer
from adafruit_ht16k33 import segments

# define I2C
i2c = board.STEMMA_I2C()

num_steps = 16  # number of steps/switches
num_drums = 11  # primary 808 drums used here, but you can use however many you like
# Beat timing assumes 4/4 time signature, e.g. 4 beats per measure, 1/4 note gets
the beat
bpm = 120  # default BPM
beat_time = 60/bpm  # time length of a single beat
beat_millis = beat_time * 1000  # time length of single beat in milliseconds
steps_per_beat = 4  # subdivide beats down to to 16th notes
steps_millis = beat_millis / steps_per_beat  # time length of a beat subdivision,
e.g. 1/16th note

step_counter = 0  # goes from 0 to length of sequence - 1
sequence_length = 16  # how many notes stored in a sequence
curr_drum = 0
playing = False

# Setup button
start_button_in = DigitalInOut(board.A2)
start_button_in.pull = Pull.UP
start_button = Debouncer(start_button_in)

# Setup switches
switch_pins = (board.TX, board.RX, board.D2, board.D3,
               board.D4, board.D5, board.D6, board.D7,
               board.D8, board.D9, board.D10, board.MOSI,
               board.MISO, board.SCK, board.A0, board.AI)

switches = keypad.Keys(switch_pins, value_when_pressed=False, pull=True)

# Setup LEDs
leds = adafruit_aw9523.AW9523(i2c, address=0x5B)  # both jumperes soldered on board
for led in range(num_steps):  # turn them off
    leds.set_constant_current(led, 0)
leds.LED_modes = 0xFFFF  # constant current mode
leds.directions = 0xFFFF  # output

# Values for LED brightness 0-255
offled = 0
dimled = 2
midled = 20
highled = 150
for led in range(num_steps):  # dramatic boot up light sequence
    leds.set_constant_current(led, midled)
    time.sleep(0.05)
    time.sleep(0.5)

# STEMMA QT Rotary encoder setup
rotary_seesaw = seesaw.Seesaw(i2c, addr=0x36)  # default address is 0x36
encoder = rotaryio.IncrementalEncoder(rotary_seesaw)
```
last_encoder_pos = 0
rotary_seesaw.pin_mode(24, rotary_seesaw.INPUT_PULLUP)  # setup the button pin
knobbutton_in = digitalio.DigitalIO(rotary_seesaw, 24)  # use seesaw digitalio
knobbutton = Debouncer(knobbutton_in)  # create debouncer object for button
encoder_pos = -encoder.position

# MIDI setup
midi = usb_midi.ports[1]

"Bass", "Snar", "LTom", "MTom", "HTom",
"Clav", "Clap", "Cowb", "Cymb", "OHat", "CHat"
]
drum_notes = [36, 38, 41, 43, 45, 37, 39, 56, 49, 46, 42]  # general midi drum notes matched to 808

# default starting sequence needs to match number of drums in num_drums
sequence = [
    [1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0], # bass drum
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], # snare
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0], # low tom
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0], # mid tom
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], # high tom
    [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], # rimshot/claves
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0], # handclap/maracas
    [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0], # cowbell
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], # cymbal
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], # hihat open
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  # hihat closed
]

def play_drum(note):
midi_msg_on = bytearray([0x99, note, 120])  # 0x90 is noteon ch 1, 0x99 is noteon ch 10
midi_msg_off = bytearray([0x89, note, 0])
midi.write(midi_msg_on)
midi.write(midi_msg_off)

def light_steps(step, state):
    if state:
        leds.set_constant_current(step, midled)
    else:
        leds.set_constant_current(step, offled)

def light_beat(step):
    leds.set_constant_current(step, highled)

def edit_mode_toggle():
    # pylint: disable=global-statement
    global edit_mode
    # pylint: disable=used-before-assignment
    edit_mode = (edit_mode + 1) % num_modes
    display.fill(0)
    if edit_mode == 0:
        display.print(bpm)
    elif edit_mode == 1:
        display.print("{}
        for k in range(num_drums):
            print("[" + ",".join('1' if e else '0' for e in sequence[k]) + "]", #
        drum_names[k])
            print("\n# set the leds
for j in range(sequence_length):
    light_steps(j, sequence[curr_drum][j])
display = segments.Seg14x4(i2c, address=(0x71))
display.brightness = 0.3
display.fill(0)
display.print(bpm)
display.show()

edit_mode = 0  # 0=bpm, 1=voices
num_modes = 2

print("Drum Trigger 2040")

while True:
    start_button.update()
    if start_button.fell:  # pushed encoder button plays/stops transport
        if playing is True:
            print_sequence()
        playing = not playing
        step_counter = 0
        last_step = int(ticks_add(ticks_ms(), -steps_millis))
        print("*** Play:", playing)
        if playing:
            now = ticks_ms()
            diff = ticks_diff(now, last_step)
            if diff >= steps_millis:
                late_time = ticks_diff(int(diff), int(steps_millis))
                last_step = ticks_add(now, - late_time/2)
                light_beat(step_counter)  # brighten current step
                for i in range(num_drums):
                    if sequence[i][step_counter]:  # if there's a 1 at the step for the
                        play_drum(drum_notes[i])
                        light_steps(step_counter, sequence[curr_drum][step_counter])  # return
                        led to step value
                        step_counter = (step_counter + 1) % sequence_length
                        encoder_pos = -encoder.position  # only check encoder while playing

    knobbutton.update()
    if knobbutton.fell:
        edit_mode_toggle()  # switches add or remove steps
    switch = switches.events.get()
    if switch:
        if switch.pressed:
            i = switch.key_number
            sequence[curr_drum][i] = not sequence[curr_drum][i]  # toggle step
            light_steps(i, sequence[curr_drum][i])  # toggle light
if encoder_pos != last_encoder_pos:
    encoder_delta = encoder_pos - last_encoder_pos
    if edit_mode == 0:
        bpm = bpm + encoder_delta  # or (encoder_delta * 5)
        bpm = min(max(bpm, 10), 400)
        beat_time = 60/bpm  # time length of a single beat
        beat_millis = beat_time * 1000
        steps_millis = beat_millis / steps_per_beat
        display.fill(0)
        display.print(bpm)
    if edit_mode == 1:
        curr_drum = (curr_drum + encoder_delta) % num_drums
        # quickly set the step leds
        for i in range(sequence_length):
            light_steps(i, sequence[curr_drum][i])
        display.print(drum_names[curr_drum])
last_encoder_pos = encoder_pos

How it Works

Libraries

The Drum Trigger Sequencer 2040 code first imports a number of libraries used for counting accurate time, reading switches, lighting LEDs using the AW9523 driver board, sending MIDI messages, writing to the display, and using the rotary encoder via SeeSaw.

```python
import time
from adafruit_ticks import ticks_ms, ticks_diff, ticks_add
import board
from digitalio import DigitalInOut, Pull
import keypad
import adafruit_aw9523
import usb_midi
from adafruit_seesaw import seesaw, rotaryio, digitalio
from adafruit_debouncer import Debouncer
from adafruit_ht16k33 import segments
```

Step and Time Setup

After setting up the I2C bus, the code defines a number of variables related to steps (the sixteen divisions of the four beat measure), drum tracks (a.k.a. "drum voices"), tempo (bpm), and variables for state, such as the step_counter, sequence_length and playing state.

```bash
# define I2C
i2c = board.STEMMA_I2C()

num_steps = 16  # number of steps/switches
num_drums = 11  # primary 808 drums used here, but you can use however many you like
# Beat timing assumes 4/4 time signature, e.g. 4 beats per measure, 1/4 note gets the beat
bpm = 120  # default BPM
beat_time = 60/bpm  # time length of a single beat
```
beat_millis = beat_time * 1000  # time length of single beat in milliseconds
steps_per_beat = 4  # subdivide beats down to 16th notes
steps_millis = beat_millis / steps_per_beat  # time length of a beat subdivision, e.g. 1/16th note

step_counter = 0  # goes from 0 to length of sequence - 1
sequence_length = 16  # how many notes stored in a sequence
curr_drum = 0
playing = False

# Setup button
start_button_in = DigitalInOut(board.A2)
start_button_in.pull = Pull.UP
start_button = Debouncer(start_button_in)

Switch Setup

The next section of code sets up the switches using the keypad library and the switch LEDs using the AW9523 library in constant current mode.

```python
# Setup switches
switch_pins = (board.TX, board.RX, board.D2, board.D3,
               board.D4, board.D5, board.D6, board.D7,
               board.D8, board.D9, board.D10, board.MOSI,
               board.MISO, board.SCK, board.A0, board.A1)
switches = keypad.Keys(switch_pins, value_when_pressed=False, pull=True)

# Setup LEDs
leds = adafruit_aw9523.AW9523(i2c, address=0x5B)  # both jumperes soldered on board
for led in range(num_steps):  # turn them off
    leds.set_constant_current(led, 0)
leds.LED_modes = 0xFFFF  # constant current mode
leds.directions = 0xFFFF  # output

# Values for LED brightness 0-255
offled = 0
dimled = 2
midled = 20
highled = 150

for led in range(num_steps):  # dramatic boot up light sequence
    leds.set_constant_current(led, dimled)
time.sleep(0.05)
time.sleep(0.5)
```

Rotary Encoder Setup

Then, the rotary encoder knob and push button are set up using the seesaw library.

```python
# STEMMA QT Rotary encoder setup
rotary_seesaw = seesaw.Seesaw(i2c, addr=0x36)  # default address is 0x36
encoder = rotaryio.IncrementalEncoder(rotary_seesaw)
last_encoder_pos = 0
rotary_seesaw.pin_mode(24, rotary_seesaw.INPUT_PULLUP)  # setup the button pin
knobbutton_in = digitalio.DigitalIO(rotary_seesaw, 24)  # use seesaw digitalio
knobbutton = Debouncer(knobbutton_in)  # create debouncer object for button
encoder_pos = -encoder.position
```
MIDI, Drum, Pattern Setup

MIDI is set up over `usb_midi`.

The `drum_names` list is used to store the strings that are displayed on the 14-segment LED backpacks.

The related `drum_notes` list defines which General MIDI note numbers correlate to each drum track (e.g., Note 36 is Bass Drum, Note 42 is a Closed Hi-Hat).

The `sequence` list stores the default 16-step pattern for each of the drum tracks. A 1 means there is a trigger at that step, a 0 means there is not. So, `[1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]` would trigger a drum on every quarter note of the measure (a four-on-the-floor kick drum pattern).

```python
# MIDI setup
midi = usb_midi.ports[1]
drum_notes = [36, 38, 41, 43, 45, 37, 39, 56, 49, 46, 42]  # general midi drum notes matched to 808

# default starting sequence needs to match number of drums in num_drums
sequence = [
    [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0], # bass drum
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], # snare
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], # low tom
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], # mid tom
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], # high tom
    [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], # rimshot/claves
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], # handclap/maracas
    [0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0], # cowbell
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], # cymbal
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], # hihat open
    [0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0], # hihat closed
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]  # default starting sequence
```

Functions

A number of functions are created as convenient reusable code sections:

```python
# Functions

def play_drum(note):
    # play_drum(note) function is called to send a MIDI NoteOn and NoteOff message for a given note number. These are sent as a byteterm for optimal speed.
```

© Adafruit Industries
def play_drum(note):
    midi_msg_on = bytearray([0x99, note, 120])  # 0x90 is noteon ch 1, 0x99 is noteon ch 10
    midi_msg_off = bytearray([0x89, note, 0])
    midi.write(midi_msg_on)
    midi.write(midi_msg_off)

light_steps() & light_beat()

The `light_steps(step, state)` function is called to toggle an LED on or off when a step switch is pressed.

The `light_beat(step)` is called once per 16th note when the sequencer is playing in order to show where the current step is in the pattern -- think of it sort of like the bouncing ball on a Karaoke machine.

def light_steps(step, state):
    if state:
        leds.set_constant_current(step, midled)
    else:
        leds.set_constant_current(step, offled)
def light_beat(step):
    leds.set_constant_current(step, highled)

edit_mode_toggle()

Since we only have one knob, we need to use it for a couple of different functions. Pressing the encoder knob toggles between the two modes so the knob can be used for either changing the tempo or picking among the drum tracks.

The `edit_mode_toggle()` function is called to toggle the sequencer between editing the tempo (BPM) with the knob and picking the tracks with the knob.

def edit_mode_toggle():
    global edit_mode
edit_mode = (edit_mode + 1) % num_modes
display.fill(0)
if edit_mode == 0:
display.print(bpm)
elif edit_mode == 1:
display.print(drum_names[curr_drum])

print_sequence()

This is a convenience function -- it is called when you press the Start/Stop button to stop playback and it prints the current sequence to the REPL/serial monitor. The
format is the same as the `sequence[]` list in the code, so you can copy and paste to change the default sequence.

Want to enhance the sequencer's functionality? The `print_sequence()` function could lead to more advanced things such as saving sequences to a text file automatically, or storing multiple sequences!

```python
def print_sequence():
    print("sequence = [ ")
    for k in range(num_drums):
        print(" [" + ','.join('1' if e else '0' for e in sequence[k]) + "], ", #
        drum_names[k])
    print("]")
```

**Set Initial LEDs**

Next, the first drum track's step sequence LEDs are lit.

```python
for j in range(sequence_length):
    light_steps(j, sequence[curr_drum][j])
```

**Display Setup**

The 14-segment LED display is set up on I2C, and then runs a set of marquees to print "Drum Trigger 2040" followed by "BPM" and the initial value of "120".

```python
display = segments.Seg14x4(i2c, address=(0x71))
display.brightness = 0.3
display.fill(0)
display.show()
display.print(bpm)
display.show()

edit_mode = 0  # 0=bpm, 1=voices
num_modes = 2
print("Drum Trigger 2040")

display.fill(0)
display.show()
display.marquee("Drum", 0.05, loop=False)
time.sleep(0.5)
display.marquee("Trigger", 0.075, loop=False)
time.sleep(0.5)
display.marquee("2040", 0.05, loop=False)
time.sleep(1)
display.marquee("BPM", 0.05, loop=False)
time.sleep(0.75)
display.marquee(str(bpm), 0.1, loop=False)
```
Main Loop

The main loop of the program first checks to see if the start button has been pressed. This toggles the playing state, and if it is being stopped, calls the `print_sequence()` function. Otherwise, if it is playing, it resets the sequence to the first step and sets the `last_step` variable based on ticks.

```python
start_button.update()
    if start_button.fell:  # pushed encoder button plays/stops transport
        if playing is True:
            print_sequence()
        playing = not playing
        step_counter = 0
        last_step = int(ticks_add(ticks_ms(), -steps_millis))
        print("*** Play:", playing)
```

Playing

When the sequence is playing, the timing is counted accurately based on ticks, the current step LED is flashed, and any drum track steps that are active are played via MIDI.

Between steps the encoder button and knob are checked, and if so, the button will switch edit modes and the knob value is stored as `encoder_pos` for later action.

```python
if playing:
    now = ticks_ms()
    diff = ticks_diff(now, last_step)
    if diff >= steps_millis:
        late_time = ticks_diff(int(diff), int(steps_millis))
        last_step = ticks_add(now, -late_time//2)
        light_beat(step_counter)  # brighten current step
        for i in range(num_drums):
            if sequence[i][step_counter]:  # if there's a 1 at the step for the
                play_drum(drum_notes[i])
            light_steps(step_counter, sequence[curr_drum][step_counter])  # return
        seq, play it
        step_counter = (step_counter + 1) % sequence_length
        encoder_pos = -encoder.position  # only check encoder while playing
        led to step value
        knobbutton.update()
        if knobbutton.fell:
            edit_mode_toggle()
        else:  # check the encoder all the time when not playing
            encoder_pos = -encoder.position
            knobbutton.update()
        if knobbutton.fell:  # change edit mode, refresh display
            edit_mode_toggle()
```
Switch Check

The switches are all checked with `switches.events.get()` to see if anything is pressed. If a switch is pressed it's light is toggled and its value in the `sequence[]` list is flipped.

```python
switch = switches.events.get()
if switch:
    if switch.pressed:
        i = switch.key_number
        sequence[curr_drum][i] = not sequence[curr_drum][i]  # toggle step
        light_steps(i, sequence[curr_drum][i])  # toggle light
```

Knob Change

If the encoder position was changed (due to knob twiddling) the tempo or track will change, depending on the current `edit_mode` state.

If in tempo mode the BPM value is increased or decreased (minimum 10, maximum 400!) and the `steps_millis` value is recalculated. The 14-segment LED display is updated to read out the current tempo.

Otherwise, in track edit mode, the knob will switch among the eleven drum tracks. Each has their own step patterns, which update the LEDs on the fly as you rotate the knob. Also, the 14-segment display shows the current track drum name.

```python
if encoder_pos != last_encoder_pos:
    encoder_delta = encoder_pos - last_encoder_pos
    if edit_mode == 0:
        bpm = bpm + encoder_delta  # or (encoder_delta * 5)
        bpm = min(max(bpm, 10), 400)
        beat_time = 60/bpm  # time length of a single beat
        beat_millis = beat_time * 1000
        steps_millis = beat_millis / steps_per_beat
        display.fill(0)
        display.print(bpm)
    if edit_mode == 1:
        curr_drum = (curr_drum + encoder_delta) % num_drums
        for i in range(sequence_length):
            light_steps(i, sequence[curr_drum][i])
        display.print(drum_names[curr_drum])
    last_encoder_pos = encoder_pos
```

Use the 16-Step Drum Sequencer

Most any DAW (digital audio workstation) software will allow you to trigger drum kits using the standardized General MIDI (GM) drum note assignments.
Here is the standard percussion sound set assignment list per MIDI note sent over channel 10:

31 Sticks
32 Square Click
33 Metronome Bell
34 Metronome Click
35 Acoustic Bass Drum
36 Electric Bass Drum
37 Side Stick
38 Acoustic Snare
39 Hand Clap
40 Electric Snare
41 Low Floor Tom
42 Closed Hi-hat
43 High Floor Tom
44 Pedal Hi-hat
45 Low Tom
46 Open Hi-hat
47 Low-Mid Tom
48 High-Mid Tom
49 Crash Cymbal 1
50 High Tom
51 Ride Cymbal 1
52 Chinese Cymbal
53 Ride Bell
54 Tambourine
55 Splash Cymbal
56 Cowbell
57 Crash Cymbal 2
58 Vibraslap
59 Ride Cymbal 2
60 High Bongo
61 Low Bongo
62 Mute High Conga
63 Open High Conga
64 Low Conga
65 High Timbale

While we normally use MIDI notes to indicate semitone pitches just like on a piano keyboard, the General MIDI standard reserves notes sent on MIDI channel 10 to play different percussion sounds per note.
GarageBand () is an excellent, free option for MacOS users, so we'll take a look at how to set it up here.

Software Instrument Track

Install GarageBand if it isn't already on your machine and launch it.

Create a new song and then pick Software Instrument from the Chose a track type dialogue box. Then click Create.
Drum Kit Select
Now you can select the drum kit for your new software instrument track. Both the Drum Kit and Electronic Drum Kit categories will work well with the Drum Trigger 2040. Here, you can see the Modern 808 has been picked.

Play!

Now, press the yellow Start/Stop button on your 16-Step Drum Sequencer -- it will start running, sending MIDI note messages to play the drum kit you selected!

Try turning the encoder knob to change the tempo.

Then, click and hold the encoder knob button to switch into track mode. The display will show the name of your current drum track -- initially Bass.

Click the step switches to change which steps will trigger the bass/kick drum.

Turn the encoder knob and you can move among the preset drum tracks. Try changing the step patterns on each track!

These are the tracks:

1. Bass
2. Snare
3. Low Tom
4. Medium Tom
5. High Tom
6. Clave
7. Clap
8. Cowbell
9. Cymbal
10. Open hi-hat
11. Closed hi-hat
Mix, Effects, EQ

Press the Play button on your 16-Step Drum Sequencer -- you'll hear your rad drum pattern playing!

You can fine tune the mix of the different drum voices, adjust effects, and fine tune the EQ (equalization) from the Control and EQ panels in GarageBand.