# CircuitPython Audio FX

Created by Jeff Epler

https://learn.adafruit.com/circuitpython-audio-fx

Last updated on 2024-10-17 11:42:22 AM EDT

# Table of Contents


Overview 3

- Parts

Coding CircuitPython Audio FX 5

- Upload the Code, Sound Effects and Libraries to the Raspberry Pi Pico 2

Using CircuitPython Audio FX 10

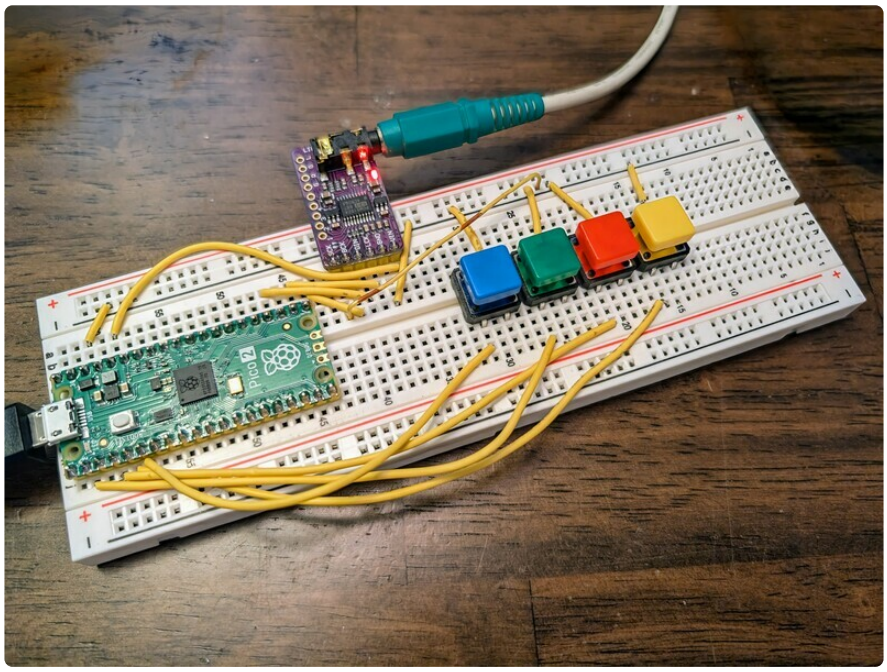
- Preparing Audio Files

Code Walkthrough 13

CircuitPython Audio FX Monophonic 16

- Convert your OGG files to MP3 and save to CIRCUITPY
- Coding CircuitPython Audio FX
- Upload the Code, Sound Effects and Libraries to the Raspberry Pi Pico 2

# Overview



I love the Adafruit Audio FX family of boards based on the VS1000 WAV/OGG decoder chip, but let's face it: they're getting a bit long in the tooth.

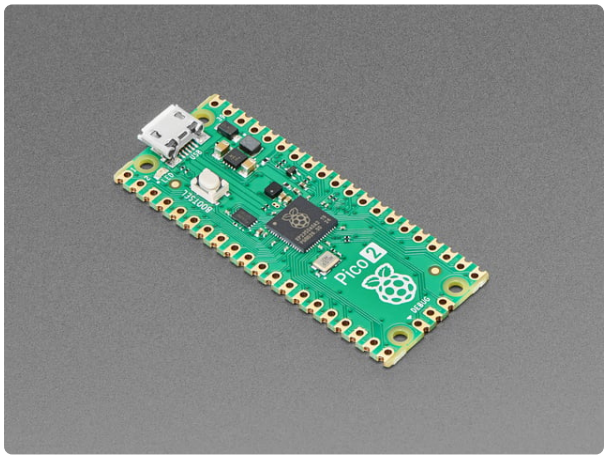
Inspired by the way the Audio FX made it easy to create interactive sound experiences, I created CircuitPython Audio FX.

Like the VS1000 Audio FX, sounds are triggered by pressing a button or closing a switch; the behavior is controlled by the name of the sound file, so you don't have to write any Python code.

Since the MP3 codec patents have all expired, CircuitPython Audio FX supports MP3 and WAV files, rather than OGG files.

CircuitPython Audio FX also supports polyphony: On the Raspberry Pi Pico 2 with RP2350 microcontroller, you can play 4 simultaneous low-bitrate MP3 files from the internal flash memory!

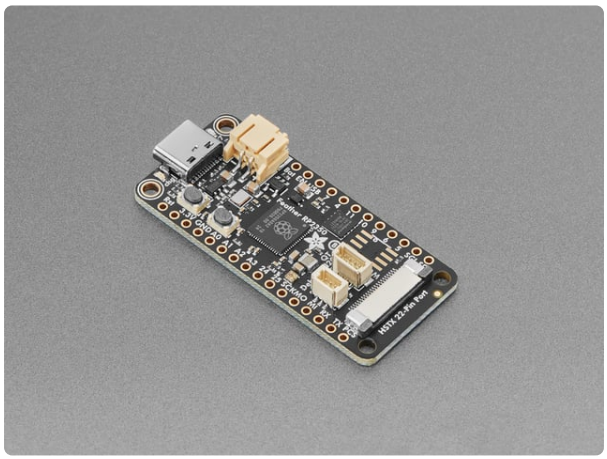
## Parts



Raspberry Pi Pico 2 - RP2350

Raspberry Pi Pico 2 is Raspberry Pi Foundation's update to their popular RP2040-based Pico board, now built on RP2350: their new...

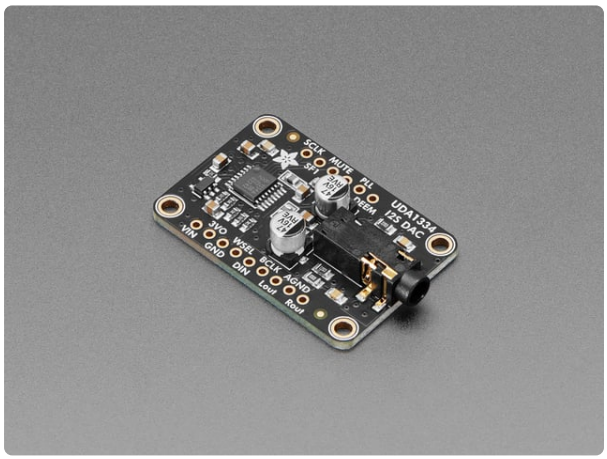
https://www.adafruit.com/product/6006



Adafruit Feather RP2350 with HSTX Port

RP2350 flies high with the Feather format - now you can use any FeatherWings with this battery-powered dev board. It comes with 8MB of flash, 22pin HSTX output port,...

https://www.adafruit.com/product/6000



Adafruit I2S Stereo Decoder - UDA1334A Breakout

This fully-featured UDA1334A I2S Stereo DAC breakout is a perfect match for any I2S-output audio interface. It's affordable but sounds great! The NXP UDA1334A is a...

https://www.adafruit.com/product/3678

**1 x** USB Cable — https://www.adafruit.com/product/592

USB A to Micro-B - 3 foot long for Pico 2

---

**1 x** USB Cable — https://www.adafruit.com/product/4474

USB Type A to Type C Cable - 1 meter / 3 ft long for Feather RP2350

---

# Coding CircuitPython Audio FX

Click on the **Download Project Bundle** button in the window below. It will download to your computer as a zipped folder.

```
# SPDX-FileCopyrightText: Copyright 2024 Jeff Epler for Adafruit Industries
# SPDX-License-Identifier: MIT

import os
import collections
import io
import random

import board
import keypad
import audiobusio
import audiocore
import audiomp3
import audiomixer

# Configure the pins to use -- earlier in list = higher priority
pads = [
    board.GP0, board.GP1, board.GP2, board.GP3,
    board.GP4, board.GP5, board.GP6, board.GP7,
    board.GP8, board.GP9, board.GP10, board.GP11,
    board.GP12, board.GP13, board.GP14, board.GP15
]

# Configure max voices to play at once
# (No matter what, at most 4 MP3 decoders)
# If set this number too high, playback will stutter. use lower bit rates or fewer
voices
#
# when the number of active samples being played back exceeds the number of voices,
# the top numbered playing sample is stopped. There is no logic to restore a sample
that
# got stopped in this way.
#
# (this may not be the same as the old FX board logic)
max_simultaneous_voices = 2
audiodev = audiobusio.I2SOut(
    bit_clock=board.GP16, word_select=board.GP17, data=board.GP18
)

# This is enough to register as an MP3 file with mp3decoder!, allows creating a
decoder
# without "opening" a "file"!
EMPTY_MP3_BYTES = b"\xff\xe3"

def exists(p):
    try:
        os.stat(p)
        return True
    except OSError:
        return False


def random_choice(seq):
    return seq[random.randrange(len(seq))]


# There's no notification when something finishes playing. So, first loop over
# all triggers; if they're not playing, then calling force_off() doesn't actually
# stop any audio (it's already stopped) but it DOES mark the voice & decoder as
# available. Otherwise, we might needlessly stop some other sample.
```

```
def free_stopped_channels():
    for i in triggers:
        if i.voice and not i.playing:
            print("fst")
            i.force_off()


# iterating on reversed triggers gives priority to **lower** numbered triggers
def ensure_available_decoder():
    if available_decoders:
        return available_decoders.popleft()

    for i in reversed_triggers:
        i.force_off()
        if available_decoders:
            break

    return available_decoders.popleft()


def ensure_available_voice():
    if available_voices:
        return available_voices.popleft()

    for i in reversed_triggers:
        i.force_off()
        if available_voices:
            break

    return available_voices.popleft()


class TriggerBase:
    def __init__(self, prefix):
        self._decoder = None
        self.voice = None
        self.filenames = list(self._gather_filenames(prefix))

    def _gather_filenames(self, prefix):
        for stem in self.stems:
            name_mp3 = f"{prefix}{stem}.mp3"
            if exists(name_mp3):
                yield name_mp3
                continue
            name_wav = f"{prefix}{stem}.wav"
            if exists(name_wav):
                yield name_wav
                continue

    def get_sample(self, path):
        if path.endswith(".mp3"):
            self._decoder = ensure_available_decoder()
            self._decoder.open(path)
            return self._decoder
        else:
            return audiocore.WaveFile(path)

    def play(self, path, loop=False):
        self.force_off()
        free_stopped_channels()
        sample = self.get_sample(path)
        self.voice = ensure_available_voice()
        self.voice.play(sample, loop=loop)

    def force_off(self):
        print("force off", self)
        voice = self.voice
        if voice is not None:
            print(f"return voice {id(voice)}")
```

```
            self.voice = None
            voice.stop()
            available_voices.append(voice)
        decoder = self._decoder
        if decoder is not None:
            print(f"return decoder {id(decoder)}")
            self._decoder = None
            print(list(available_decoders), end=" ")
            available_decoders.append(decoder)
            print("->", list(available_decoders))

    @property
    def playing(self):
        return False if self.voice is None else self.voice.playing

    @classmethod
    def matches(cls, prefix):
        stem = cls.stems[0]
        name_mp3 = f"{prefix}{stem}.mp3"
        name_wav = f"{prefix}{stem}.wav"
        return exists(name_wav) or exists(name_mp3)

    def __repr__(self):
        return f"<{self.__class__.__name__} {self.filenames}{' playing' if
self.playing else ''}>"


class NopTrigger(TriggerBase):
    """Does nothing."""

    stems = [""]

    def on_press(self):
        pass

    def on_release(self):
        pass


class BasicTrigger(TriggerBase):
    """Plays a file each time the button is pressed down"""

    stems = [""]

    def on_press(self):
        self.play(self.filenames[0])

    def on_release(self):
        pass


class HoldLoopingTrigger(TriggerBase):
    """Plays a file as long as a button is held down"""

    stems = ["HOLDL"]

    def on_press(self):
        self.play(self.filenames[0], loop=True)

    def on_release(self):
        self.force_off()


class LatchingLoopTrigger(TriggerBase):
    """Toggles playing each time the button is pressed"""

    stems = ["LATCH"]

    def on_press(self):
```

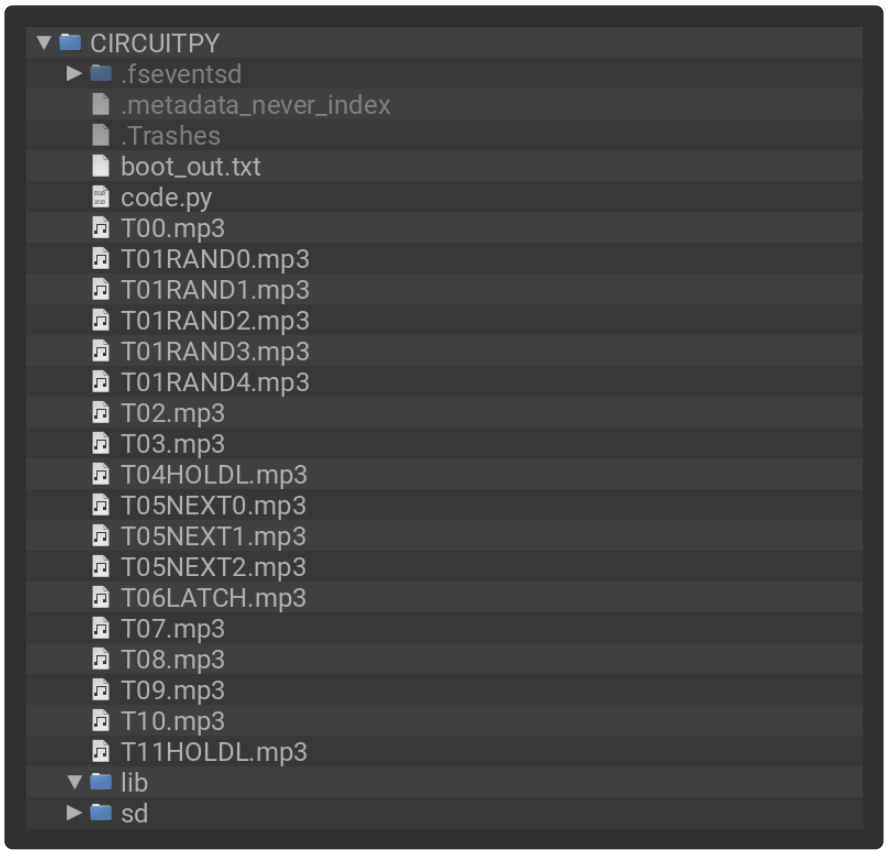
```
        if self.playing:
            self.force_off()
        else:
            self.play(self.filenames[0], loop=True)

    def on_release(self):
        pass


class PlayNextTrigger(TriggerBase):
    stems = [f"NEXT{i}" for i in range(10)]

    def __init__(self, prefix):
        super().__init__(prefix)
        self._phase = 0

    def on_press(self):
        self.play(self.filenames[self._phase])
        self._phase = (self._phase + 1) % len(self.filenames)

    def on_release(self):
        pass


class PlayRandomTrigger(TriggerBase):
    stems = [f"RAND{i}" for i in range(10)]

    def on_press(self):
        self.play(random_choice(self.filenames))

    def on_release(self):
        pass


trigger_classes = [
    BasicTrigger,
    HoldLoopingTrigger,
    LatchingLoopTrigger,
    PlayNextTrigger,
    PlayRandomTrigger,
]


def make_trigger(i):
    prefix = f"T{i:02d}"

    for cls in trigger_classes:
        if not cls.matches(prefix):
            continue
        return cls(prefix)

    return NopTrigger(prefix)


# No matter what, at most 4 MP3 decoders
decoders = [
    audiomp3.MP3Decoder(io.BytesIO(EMPTY_MP3_BYTES))
    for _ in range(min(4, max_simultaneous_voices))
]
print(decoders)
available_decoders = collections.deque(decoders, len(decoders))
print(list(available_decoders))

keys = keypad.Keys(pads, value_when_pressed=False)

triggers = [make_trigger(i) for i in range(len(pads))]


def playback_specs(sample):
```

```
    return dict(
        channel_count=sample.channel_count,
        sample_rate=sample.sample_rate,
        bits_per_sample=sample.bits_per_sample,
    )


def check_match_make_mixer(dev):
    all_filenames = []
    for i in triggers:
        all_filenames.extend(i.filenames)

    if not all_filenames:
        raise RuntimeError("*** NO AUDIO FILES FOUND ***")

    if max_simultaneous_voices == 1:
        return [dev]

    first_trigger = triggers[0]

    mixer_buffer_size = (1152 * 4) * 4

    specs = None
    for filename in all_filenames:
        sample = first_trigger.get_sample(filename)
        new_specs = playback_specs(sample)
        if specs is None:
            specs = new_specs
        else:
            if specs != new_specs:
                print("*** Audio file specs don't match ***")
                print("{all_filenames[0]}: {specs}")
                print("{filename}: {specs}")
                raise RuntimeError("*** WITH POLYPHONY, ALL MUST MATCH ***")
        first_trigger.force_off()

    print(f"audio specs: {specs}")
    samples_signed = specs["bits_per_sample"] == 16
    mixer = audiomixer.Mixer(
        voice_count=max_simultaneous_voices,
        buffer_size=mixer_buffer_size,
        samples_signed=samples_signed,
        **specs,
    )
    dev.play(mixer)

    return list(mixer.voice)


print(triggers)
print(list(available_decoders))

reversed_triggers = list(reversed(triggers))

voices = check_match_make_mixer(audiodev)
print(list(available_decoders))
available_voices = collections.deque(voices, len(voices))

while True:
    if e := keys.events.get():
        print("event", e)
        print("available decoders", *(id(i) for i in available_decoders))
        print("available voices", *(id(i) for i in available_voices))
        trigger = triggers[e.key_number]
        if e.pressed:
            trigger.on_press()
        else:
            trigger.on_release()
        print(triggers)
```

## Upload the Code, Sound Effects and Libraries to the Raspberry Pi Pico 2

After downloading the Project Bundle, plug your Pico 2 into the computer's USB port with a known good USB data+power cable. You should see a new flash drive appear in the computer's File Explorer or Finder (depending on your operating system) called **CIRCUITPY**. Open up the zipped file, go to the the appropriate folder inside (e.g., **circuitpython-audio-fx/polyphonic/CircuitPython 9.x** if you are using CircuitPython 9.x) and copy the items in that folder directly onto your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should look like this after copying the sound effects and the **code.py** file.




---

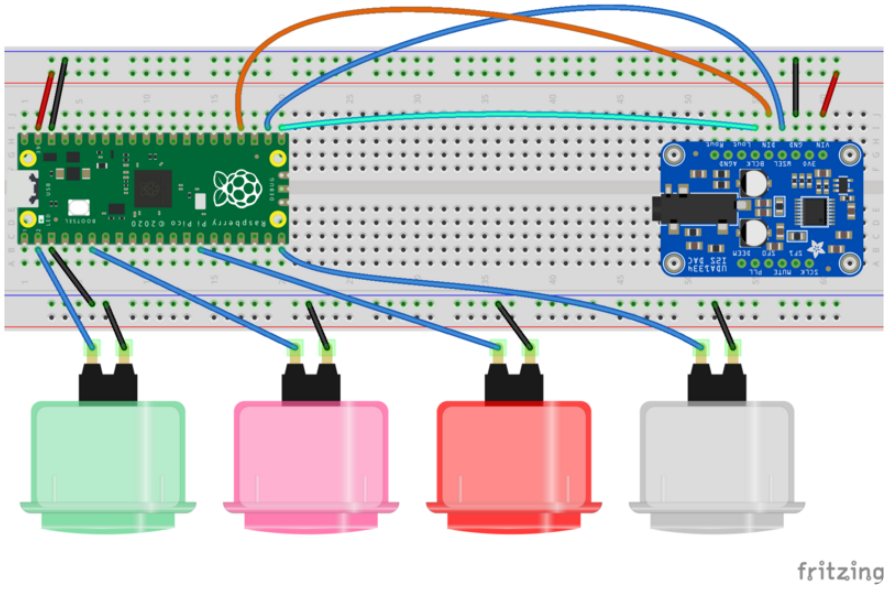
# Using CircuitPython Audio FX

### Project Wiring

Connect your Pico 2 board to an I2S DAC or I2S Amplifier using the following pins:

- Connect Pico GP16 to I2S Bit Clock (BCLK)
- Connect Pico GP17 to I2S Word Select (WSEL)
- Connect Pico GP18 to I2S Data (DAT)
- Connect Pico GND to GND

- Connect your I2S board to appropriate power & speakers or headphones




Next, connect at least one but up to 16 trigger buttons or switches. All of the pins from GP0 to GP15 are for audio triggers. For each button or switch:

- Connect one side of the switch to one of the GPx pins
- Connect the other side of the switch to GND

The sample project includes sound files for 12 different triggers on pins GP0 through GP11. The diagram only shows a few buttons, but any or all of the first 16 GPIO pins GP0 through GP15 can be used.

## Preparing Audio Files

You can use the software of your choice to prepare audio files for CircuitPython Audio FX. If you don't already have software for recording & saving sound samples, you might want to try the open source Audacity (https://adafru.it/1a8o), which runs on Linux, Mac and Windows.

For a given project, all the audio files have to have the same key specifications:

- sample rate (e.g., 8kHz, 11.025kHz, 16kHz)
- number of samples (mono or stereo)
- bit depth (8-bit unsigned or 16-bit signed)

While WAV files are supported, the Pico 2's **CIRCUITPY** drive has a capacity of only about 3 megabytes, so you will probably want to load up with compressed MP3 files instead.

The amount of audio you can load in MP3 format depends on the bit rate. For 128 kbit/second "CD quality" MP3 files (44.1kHz, stereo, 16-bit), you can store about a total of 3 minutes of audio. If you go for a lower quality file such as 32kbit/second (16kHz, mono, 16-bit) you can have over 10 minutes of sound files.

The MP3 **bitrate** is separate from the **sample** rate. A higher **bitrate** means the file takes up more kB per second of audio, but the audio fidelity improves with a higher bitrate. Within a project, MP3 files can have different **bitrates**. For example, your project can include a file with a 16kHz sample rate encoded at 32kbit/s bitrate alongside a file with a 16kHz sample rate and a 64kbit/s bitrate.

A project can mix MP3 files and WAV files but the channel count and sample rate must be the same, and the wave files must be 16-bit signed files.

The included sample files are all 16kHz mono 16-bit MP3s encoded at 32 kbit/s.

## Naming audio files

The name of your audio file controls when and how it will be played.

The name of an audio file is divided into several parts:

- The trigger number, **T00** through **T15**
- The kind of trigger:
    - Empty, for a basic trigger (starts playing once when its trigger is pressed)
    - **HOLDL** for a hold looping trigger (plays in a loop while trigger is pressed)
    - **LATCH** for a toggle looping trigger (starts playing in a loop when trigger is pressed, stops next time trigger is pressed)
    - **NEXT#** for a sequential trigger (plays **NEXT0**, then **NEXT1**, up to **NEXT9**, then back to **NEXT0** each time its trigger is pressed)
    - **RAND#** for random trigger (randomly chooses among **RAND0**, **RAND1** up to **RAND9**, each time its trigger is pressed)
- The file extension, **.WAV** or **.MP3**

Filenames are not case dependent, so you can have **T00RAND9.MP3** or **T00rand9.Mp3** and it will work the same way.

If several kinds of trigger all appear for the same trigger number, only one of them will be used. For example if you have both **T15HOLDL.mp3** and **T15.mp3** (two different trigger types for one trigger number), one or the other will be used, but not both. Similarly, if you have both **T14.mp3** and **T14.wav**, one or the other will be used, but not both (two different file extensions for the same file).

The demonstration files show all the different kinds of triggers, including:

- T00.mp3: A basic trigger on pin GP0
- T01RAND0.mp3 through T01RAND4.mp3: A random trigger on pin GP1
- T04HOLDL.mp3: A hold looping trigger on pin GP4
- T05NEXT0.mp3 through T05NEXT2.mp3: A sequential trigger on pin GP5
- T06LATCH.mp3: A latching looping trigger on pin GP6

### Polyphony & Precedence

In **code.py** there is a configurable number for the maximum **polyphony**, or the maximum number of different sound files that can be played simultaneously. There's a second limit for the number of simultaneous **mp3** files that can be played, because each "MP3 Decoder" object requires a substantial amount of RAM.

When a new sound file is triggered, but the maximum number are already playing, here's what happens: Of the currently playing sounds **the one with the highest trigger number** is stopped, and then the new sound is started.

So imagine that you have polyphony of 2 (the default), and sounds T05 and T06 are playing. When you trigger another channel, whether it's T02 or T15, then the T06 sound is stopped and the new sound file is started.

### Audio file maximums

The maximum audio files that can be played depend on

- format: wav vs mp3
- bit rate & sample rate
- audio volume & clipping when mixing many samples

As a guideline a non-overclocked RP2350 can successfully play 2 320kbit/s MP3 files or 8 32kbit/s MP3 files from internal flash. With overclocking, you can go further.

There's room in the RP2350 RAM for up to 9 MP3 decoder objects, but this has to be reduced somewhat to allow for a complex **code.py** program. The soft maximum of 4 in the **code.py** is probably a bit too restrictive.

---

## Code Walkthrough

There's a fair amount of code in this project! Let's look at the highlights.

You can reconfigure the pins used for triggers by updating this block. The number of triggers can be fewer or more than the 16 shown here.

```
# Configure the pins to use -- earlier in list = higher priority
pads = [
    board.GP0, board.GP1, board.GP2, board.GP3,
    board.GP4, board.GP5, board.GP6, board.GP7,
    board.GP8, board.GP9, board.GP10, board.GP11,
    board.GP12, board.GP13, board.GP14, board.GP15
]
```

You can select a greater number of simultaneous voices (polyphony). Further down there's an additional limit of at most 4 simultaneous MP3s.

```
# Configure max voices to play at once
# (No matter what, at most 4 MP3 decoders)
# If set this number too high, playback will stutter. use lower bit rates or fewer
voices
#
# when the number of active samples being played back exceeds the number of voices,
# the top numbered playing sample is stopped. There is no logic to restore a sample
that
# got stopped in this way.
#
# (this may not be the same as the old FX board logic)
max_simultaneous_voices = 2
```

You can create an I2SOut on different pins, or change to a different kind of audio output altogether (such as `PWMAudioOut`)

```
audiodev = audiobusio.I2SOut(
    bit_clock=board.GP16, word_select=board.GP17, data=board.GP18
)
```

Each kind of trigger derives from TriggerBase and defines several items:

- The filename fragments associated with this kind of trigger. This is the part of the filename immediately after the **T##** trigger number. It should be a list with at least one string in it. The kind of trigger is chosen based on matching the first stem in the list, so each trigger needs to have a unique stem compared to all the others.
- The action to take when the associated trigger is pressed
- The action to take when the associated trigger is released

Usually one trigger will call **self.play**. If the play is looping, then the other trigger will probably call **self.force_off**.

If no action is called for in a press or release, provide a method that just says "pass".

Each kind of trigger also needs to be added to the list of **trigger_classes**.

The basic stem doesn't loop, so it only needs to do something on press:

```
class BasicTrigger(TriggerBase):
    """Plays a file each time the button is pressed down"""

    stems = [""]

    def on_press(self):
        self.play(self.filenames[0])

    def on_release(self):
        pass
```

The HoldLoopingTrigger demonstrates that samples can be looped and shows use of **force_off** in the **on_release** method:

```
class HoldLoopingTrigger(TriggerBase):
    """Plays a file as long as a button is held down"""

    stems = ["HOLDL"]

    def on_press(self):
        self.play(self.filenames[0], loop=True)

    def on_release(self):
        self.force_off()
```

The **PlayRandomTrigger** class shows how to select one out of a list of different files. Notice how **stems** will be a list of the strings **RAND0**, **RAND1**, ..., **RAND9**. It uses **random_choice**, a function that works similar to standard Python's **random.choice** function:

```
class PlayRandomTrigger(TriggerBase):
    stems = [f"RAND{i}" for i in range(10)]

    def on_press(self):
        self.play(random_choice(self.filenames))

    def on_release(self):
        pass
```

In the TriggerBase class, the play function starts playing the new sample, after stopping another channel from playing if necessary (inside **ensure_available_voice**; similarly, an MP3 channel is freed if necessary by **ensure_available_decoder**):

```
class TriggerBase:
    ...
    def get_sample(self, path):
        if path.endswith(".mp3"):
            self._decoder = ensure_available_decoder()
            self._decoder.open(path)
            return self._decoder
        else:
            return audiocore.WaveFile(path)

    def play(self, path, loop=False):
        self.force_off()
        free_stopped_channels()
        sample = self.get_sample(path)
```

```
        self.voice = ensure_available_voice()
        self.voice.play(sample, loop=loop)
```

The main loop just checks for key events and calls **on_press**/**on_release** as needed (it also spews some debugging information to the serial console):

```
while True:
    if e := keys.events.get():
        print("event", e)
        print("available decoders", *(id(i) for i in available_decoders))
        print("available voices", *(id(i) for i in available_voices))
        trigger = triggers[e.key_number]
        if e.pressed:
            trigger.on_press()
        else:
            trigger.on_release()
        print(triggers)
```

---

# CircuitPython Audio FX Monophonic

Polyphonic Audio FX is pretty cool, but suppose you have an existing project for the original Audio FX and you want to "port" it to a newer board using CircuitPython instead of VS1000. We got you! Load up your CircuitPython with the monophonic version of the script and you'll get behavior a lot closer to the VS1000-based Audio FX boards. (not working quite right? Please drop us some feedback using the link in the sidebar!)

Wiring is the same as shown under Using CircuitPython Audio FX (https://adafru.it/1a93).

## Convert your OGG files to MP3 and save to CIRCUITPY

Using software of your choice, such as the free and open source Audacity (https://adafru.it/1a8o) which is available on Linux, Windows & Mac computers. Open each file and then export it as an MP3 file. Keep the first part of the filename (e.g., T01RAND0 or T11LATCH) but choose the right file extension (.mp3)

The RP2350 has approximately 3MB of flash available for the CIRCUITPY drive, so if you were using a VS1000 with a larger capacity you will have to choose a higher compression ratio (lower bit rate) when performing the conversion.

If you still have original wave files with a .WAV extension, it's better to use these files for MP3 conversion, because the WAV file is lossless. Doing repeated lossy audio conversions can reduce the quality of the final audio file, the audio version of those weird crinkly edges and smudgy indistinct areas that JPEG files can get.

Unlike the polyphonic version, your files can have different sample rates, you can mix mono & stereo, etc. For more information about naming and converting files see the

headings "Preparing Audio Files" and "Naming Audio Files" under Using CircuitPython Audio FX (https://adafru.it/1a93). For information about how each trigger type works, read about it in the original Audio FX guide (https://adafru.it/pqb).

## Coding CircuitPython Audio FX

Click on the **Download Project Bundle** button in the window below. It will download to your computer as a zipped folder.

```
# SPDX-FileCopyrightText: Copyright 2024 Jeff Epler for Adafruit Industries
# SPDX-License-Identifier: MIT

# pylint: disable=no-self-use

import os
import io
import random

import board
import digitalio
import keypad
import audiobusio
import audiocore
import audiomp3

# Configure the pins to use -- earlier in list = higher priority
pads = [
    board.GP0, board.GP1, board.GP2, board.GP3,
    board.GP4, board.GP5, board.GP6, board.GP7,
    board.GP8, board.GP9, board.GP10, board.GP11,
    board.GP12, board.GP13, board.GP14, board.GP15
]

# Configure the audio device
audiodev = audiobusio.I2SOut(
    bit_clock=board.GP16, word_select=board.GP17, data=board.GP18
)

led = digitalio.DigitalInOut(board.LED)
led.switch_to_output(False)

# This is enough to register as an MP3 file with mp3decoder!, allows creating a
decoder
# without "opening" a "file"!
EMPTY_MP3_BYTES = b"\xff\xe3"

# Create the MP3 decoder object
decoder = audiomp3.MP3Decoder(io.BytesIO(EMPTY_MP3_BYTES))

def exists(p):
    try:
        os.stat(p)
        return True
    except OSError:
        return False


def random_shuffle(seq):
    for i in range(len(seq)):
        j = random.randrange(0, i+1)
        if i != j: # Chance an item remains in same location
            seq[i], seq[j] = seq[j], seq[i]

def random_cycle(seq):
```

```
    while True:
        random_shuffle(seq)
        yield from seq

def cycle(seq):
    while True:
        yield from seq

class TriggerBase:
    def __init__(self, prefix):
        self._filenames = list(self._gather_filenames(prefix))
        self._filename_generator = type(self).generate_filenames(self._filenames)
        self.wants_to_play = False

    # Can be cycle or random_cycle
    generate_filenames = cycle

    def on_press(self):
        self.wants_to_play = True

    def on_release(self):
        self.wants_to_play = False

    def on_activate(self):
        self.play_wait()

    def _gather_filenames(self, prefix):
        if self.stems is None:
            return
        for stem in self.stems:
            name_mp3 = f"{prefix}{stem}.mp3"
            if exists(name_mp3):
                yield name_mp3
                continue
            name_wav = f"{prefix}{stem}.wav"
            if exists(name_wav):
                yield name_wav
                continue

    def _get_sample(self, path):
        if path.endswith(".mp3"):
            decoder.open(path)
            return decoder
        else:
            return audiocore.WaveFile(path)

    def play(self, loop=False):
        audiodev.stop()
        path = next(self._filename_generator)
        sample = self._get_sample(path)
        audiodev.play(sample, loop=loop)

    def play_wait(self):
        self.play()
        while audiodev.playing:
            poll_keys()

    def stop(self):
        audiodev.stop()

    @classmethod
    def matches(cls, prefix):
        stem = cls.stems[0]
        name_mp3 = f"{prefix}{stem}.mp3"
        name_wav = f"{prefix}{stem}.wav"
        return exists(name_wav) or exists(name_mp3)

    def __repr__(self):
        return (f"<{self.__class__.__name__} {' '.join(self._filenames)}" +
```

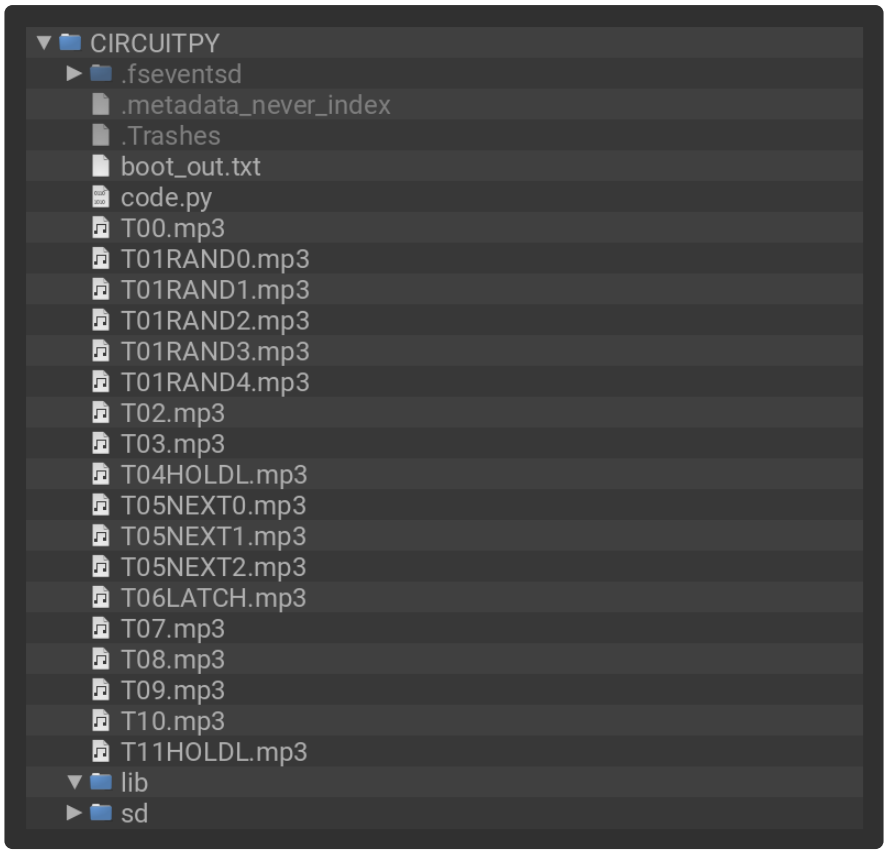
```
                f"{' ACTIVE' if self.wants_to_play else ''}>")


class NopTrigger(TriggerBase):
    """Does nothing."""

    stems = None

    def on_activate(self):
        return

class BasicTrigger(TriggerBase):
    """Plays a file each time the button is pressed down"""

    stems = [""]

class HoldLoopingTrigger(TriggerBase):
    """Plays a file as long as a button is held down

    This differs from the basic trigger because the loop stops as soon as the button
    is released """

    stems = ["HOLDL"]

    def on_activate(self):
        self.play(loop=True)
        while audiodev.playing:
            poll_keys()
            for trigger in triggers:
                if trigger is self:
                    break
                if trigger.wants_to_play:
                    self.wants_to_play = False
            if not self.wants_to_play:
                audiodev.stop()

class LatchingLoopTrigger(HoldLoopingTrigger):
    """Plays a file until the button is pressed again

    When the button is pressed again, stops the loop immediately."""

    stems = ["LATCH"]

    def on_press(self):
        if self.wants_to_play or not audiodev.playing:
            self.wants_to_play = not self.wants_to_play

    def on_release(self):
        pass # override default behavior


class PlayNextTrigger(TriggerBase):
    stems = [f"NEXT{i}" for i in range(10)]
    _phase = 0


class PlayRandomTrigger(TriggerBase):
    stems = [f"RAND{i}" for i in range(10)]

    generate_filenames = random_cycle



trigger_classes = [
    BasicTrigger,
    HoldLoopingTrigger,
    LatchingLoopTrigger,
    PlayNextTrigger,
    PlayRandomTrigger,
```

```
]


def make_trigger(i):
    prefix = f"T{i:02d}"

    for cls in trigger_classes:
        if not cls.matches(prefix):
            continue
        return cls(prefix)

    return NopTrigger(prefix)


keys = keypad.Keys(pads, value_when_pressed=False)

triggers = [make_trigger(i) for i in range(len(pads))]

def poll_keys():
    while e := keys.events.get():
        trigger = triggers[e.key_number]
        if e.pressed:
            trigger.on_press()
        else:
            trigger.on_release()
        print(e.pressed, trigger)

print(triggers)

reversed_triggers = list(reversed(triggers))

while True:
    poll_keys()
    for t in triggers:
        if t.wants_to_play:
            print(t)
            t.on_activate()
            break
```

## Upload the Code, Sound Effects and Libraries to the Raspberry Pi Pico 2

After downloading the Project Bundle, plug your Pico 2 into the computer's USB port with a known good USB data+power cable. You should see a new flash drive appear in the computer's File Explorer or Finder (depending on your operating system) called **CIRCUITPY**. Open up the zipped file, go to the the appropriate folder inside (e.g., **circuitpython-audio-fx/polyphonic/CircuitPython 9.x** if you are using CircuitPython 9.x) and copy the items in that folder directly onto your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should look like this after copying the sound effects and the **code.py** file.




At this point, CircuitPython will automatically restart and run the Audio FX code. If you activate the digital inputs (for instance, by pressing a connected button) a sound effect will play. If you run into trouble, you can use the CircuitPython serial connection, sometimes call the REPL, to troubleshoot.