Bricktunes: LEGO Synthesizer Glove

Created by John Park

https://learn.adafruit.com/bricktunes-lego-glove-synthesizer

Last updated on 2023-10-21 02:39:35 PM EDT
# Table of Contents

**Overview** 3  
- Parts & Materials

**Assemble the Bricktunes Circuit** 8  
- Connections

**CircuitPython** 9  
- CircuitPython Quickstart  
- Safe Mode  
- Flash Resetting UF2

**Code the Bricktunes Synth** 13  
- Text Editor  
- Download the Project Bundle  
- How It Works

**Build the Bricktunes Glove** 21  
- Attach Feather to Glove  
- Sensor Positioning and Cabling  
- Sew the Sensor  
- Mount the Speaker  
- Connect Battery

**Play Music with your LEGO Bricks** 27

**Optional LEGO Bricktune Platform Build** 28
Overview

Point at a LEGO part and make music with the Bricktunes synthesizer.

Bricktunes is a LEGO color sensing glove-mounted synth. It uses an eight channel color sensor to distinguish between bricks and plays synthesized chime arpeggios with synthio in CircuitPython running on a Feather Prop-Maker RP2040.

This project was inspired by an MIT Media Lab project by friend-of-Adafruit Jay Silver ( ).

LEGO Colors

Many different LEGO colors were tested during the development of Bricktunes. While I was able to have the sensor distinguish between over 25 different colors (of the 64 I tested) -- there are trade-offs to be made between speed and accuracy. Ultimately, I settled on 12 colors that are distinct enough to register quickly with very few false reads.

Here's a helpful guide () on the currently produced solid brick colors.
Parts & Materials

Adafruit RP2040 Prop-Maker Feather with I2S Audio Amplifier
The Adafruit Feather series gives you lots of options for a small, portable, rechargeable microcontroller board. By picking a feather and stacking on a FeatherWing you can create...
https://www.adafruit.com/product/5768

Adafruit AS7341 10-Channel Light / Color Sensor Breakout
The Adafruit AS7341 10-Channel Light / Color Sensor Breakout is a multi-channel spectrometer, which is a special type of light sensor that is able to detect not only...
https://www.adafruit.com/product/4698

Mono Enclosed Speaker with Plain Wires - 3W 4 Ohm
Listen up! This single 2.8" x 1.2" speaker is the perfect addition to any audio project where you need 4 ohm impedance and 3W or less of power. We...
https://www.adafruit.com/product/4445
STEMMA QT / Qwiic JST SH 4-Pin Cable - 400mm long
This 4-wire cable is a little over 400mm / 15.7" long and fitted with JST-SH female 4-pin connectors on both ends. Compared with the chunkier JST-PH these are 1mm pitch instead of...
https://www.adafruit.com/product/5385

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

Black Nylon Machine Screw and Stand-off Set – M2.5 Thread
Totaling 380 pieces, this M2.5 Screw Set is a must-have for your workstation. You'll have enough screws, nuts, and hex standoffs to fuel your maker...
https://www.adafruit.com/product/3299
Glove
Any glove should work fine, however these work gloves are especially nice for mounting parts, thanks to their rubber patches. I got a pair at my local hardware store for about $20.

LEGO Plates
Not all LEGO parts come in every color. Instead of bricks we're technically using plates. You can order these from the LEGO Pick a Brick shop, or a seller through Brinklink or other online stores.

You can use as many pieces as you like. Four each of the 2x4 plates in the following colors is a great starting point. Note: the names used are from BrickLink's Studio and Marketplace, with the official LEGO name in parentheses where different.

Blue (Bright Blue)
Bright Green
Bright Light Orange (Flaming Yellowish Orange)
Bright Pink (Light Purple)
Coral
Dark Purple (Medium Lilac)
Dark Turquoise (Bright Bluish Green)
Lavender
Lime (Bright Yellowish Green)
Red (Bright Red)
Sand Green
Yellow (Bright Yellow)
Why are there multiple color names? The LEGO fan community came up with names for colors before LEGO had decided to reveal their official internal color names!

More Color Info

The last time LEGO published an official color chart was 2016, see the image below. Since then a few new colors have been released -- Coral, Neon Yellow, Medium Tan, and Medium Brown -- and the Dark Turquoise color was brought back, so there have been community efforts to produce updated charts, such as the one found here.
Assemble the Bricktunes Circuit
Connections
This build is plug-and-play, no soldering required. Before assembling into the glove, you can build and code the Feather, color sensor, and speaker to make sure everything's working well.

First, plug the AS7341 sensor into the Feather using the STEMMA QT cable.

Then, screw the speaker wires into the Feather's terminal block:

- speaker red wire to Feather amplifier +
- speaker black wire to Feather amplifier -

You can also plug in the LiPoly battery to the Feather's battery JST connector.

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 Bricktunes Synth

Text Editor

Adafruit recommends using the Mu editor for editing your CircuitPython code. You can get more info in this guide.

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.

Connect your computer to the Feather board via a known good USB power+data cable. A new flash drive should show up as CIRCUITPY.

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

```
# SPDX-FileCopyrightText: 2023 John Park for Adafruit
# SPDX-License-Identifier: MIT

# Bricktunes LEGO Color Synth
# Feather RP2040 Prop-Maker + AS7341 Color Sensor
# Color comparison code and chime library by CGrover

import time
import math
import board
import digitalio
import audiobusio
```
from adafruit_as7341 import AS7341, Gain
import audiomixer
from cedargrove_chime import Chime, Voice, Material, Striker

DEBUG = False  # Useful for tuning reference color values by printing them
TOLERANCE = 800  # The color matching tolerance index (0 to 8 * max_sensor_count)

sensor = AS7341(board.STEMMA_I2C())
sensor.astep = 128  # (999) The integration time step size in 2.78 microsecond increments
sensor.atime = 50  # The integration time step count.
sensor.gain = Gain.GAIN_256X
sensor.led_current = 4  # increments in units of 4
sensor.led = True
max_sensor_count = (sensor.astep + 1) * (sensor.atime + 1)

# ===================================================
# color lists as 8-channel tuples (channels[0:8])
brick_full_spectrum_values = [
    (94, 1310, 1736, 1075, 592, 437, 497, 383),  # Blue
    (148, 324, 838, 2577, 2363, 1259, 929, 819),  #
    (381, 576, 850, 1619, 3688, 5532, 6291, 4250),  # Bright Green
    (404, 2300, 2928, 2385, 2679, 3804, 5576, 4284),  #
    (545, 1276, 1513, 1178, 2291, 6579, 6579, 6486),  # Bright Lt Orange
    (136, 1055, 1223, 745, 748, 768, 1205, 1100),  # Bright Pink
    (85, 731, 1375, 1604, 1019, 557, 533, 370),  # Dark Purple
    (451, 2758, 3786, 2880, 3007, 3064, 4539, 3656),  # Turquoise
    (214, 300, 771, 1811, 3245, 2897, 2051, 1392),  # Lavender
    (188, 341, 435, 507, 625, 1703, 4361, 3692),  # Lime
    (182, 870, 1455, 1799, 2149, 1879, 1702, 1273),  # Sand Green
    (461, 497, 878, 2412, 4699, 5935, 6579, 4677)  # Yellow
]
brick_color_names = [
    "Blue",
    "Bright Green",
    "Bright Light Orange",
    "Bright Pink",
    "Coral",
    "Dark Purple",
    "Dark Turquoise",
    "Lavender",
    "Lime",
    "Red",
    "Sand Green",
    "Yellow"
]
brick_states = [False] * (len(brick_color_names))
gap_state = False

# ===================================================
# audio setup
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)
audio_output = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)
mixer = audiomixer.Mixer(sample_rate=11020, buffer_size=4096, voice_count=1,
channel_count=1)
audio_output.play(mixer)
mixer.voice[0].level = 0.50  # adjust this for overall volume

brickscale = [
        "C5", "D5", "E5", "F5", "G5", "A5", "B5",
        "C7", "D7", "E7", "F7", "G7", "A7", "B7",
    ]

# Instantiate the chime synthesizer with custom parameters
chime = Chime(
    mixer.voice[0],
    scale=brickscale,
    material=Material.Brass,  # SteelEMT, Ceramic, Wood, Copper,
    Aluminium, Brass
    voice=Voice.Tubular,  # bell, perfect, tubular
    scale_offset=-16
)

# Play scale notes sequentially
for index, note in enumerate(chime.scale):
    chime.strike(note, 1.0)
    time.sleep(0.1)
    time.sleep(1)

def compare_n_channel_colors(color_1, color_2, tolerance=0):
    """Compares two integer multichannel count tuples using an unweighted linear
    Euclidean difference. If the color value difference is within the tolerance
    band of the reference, the method returns True.
    The difference value index 'tolerance' is used to detect color similarity.
    Value range is an integer value from 0 to
    (maximum_channel_count * number_of_channels). Default is 0 (detects a
    single color value)."""
    # Create list of channel deltas using list comprehension
deltas = [(((color_1[idx] - count) ** 2) for idx, count in enumerate(color_2)]
    # Resolve squared deltas to a Euclidean difference
    # pylint: disable=c-extension-no-member
    delta_color = math.sqrt(sum(deltas))
    return bool(delta_color <= tolerance)

print("Bricktunes ready")

while True:
    sensor_color = sensor.all_channels
    # don't bother to check comparison when we're looking at a gap between bricks
    if sensor_color[0] <= 70:  # this checks for a minimum value on one channel
        if gap_state is False:
            print("no brick...")
            for i in range(len(brick_color_names)):
                brick_states[i] = False
            gap_state = True
        else:
            if DEBUG:
                print(sensor_color)
            for i in range(len(brick_full_spectrum_values)):
                color_match = compare_n_channel_colors(
                    sensor_color, brick_full_spectrum_values[i], TOLERANCE
                )

                if color_match is True:
                    if brick_states[i] is False:
                        for n in range(5):
                            chime.strike(chime.scale[i+(n*2)], 1.0)
How It Works

The code has two key functions:

- check the color sensor to see if there is a match (within specified tolerance) to a reference color
- play a synthio arpeggio based on the matched color

Import Libraries

First, you'll import the necessary libraries:

- **time**: used to create delays between notes
- **math**: used for some of the color comparison functions
- **board**: provides pin mappings for the Feather board
- **digitalio**: used to power on the extra peripherals of the Feather Prop-Maker (in this case the I2S amp)
- **audiobusio**: for sending audio to the I2S amp output
- **adafruit_as7341**: the color sensor library
- **audiomixer**: used for setting the audio output volume
- **cedargrove_chime**: this library from the community bundle adds note overtones and envelopes from tubular bell/chime algorithms to synthio

```python
import time
import math
import board
import digitalio
import audiobusio
```
from adafruit_as7341 import AS7341, Gain
import audiomixer
from cedargrove_chime import Chime, Voice, Material, Striker

User Parameters

The **DEBUG** variable can be set to True or False to switch on and off printing to the serial output of the color values. This is used for calibration.

The **TOLERANCE** variable sets the color matching tolerance index. It defines how close the detected color needs to be to a reference color to trigger a musical note. A higher value increases tolerance.

```
DEBUG = False  # Useful for tuning reference color values by printing them
TOLERANCE = 800  # The color matching tolerance index (0 to 8 * max_sensor_count)
```

Color Sensor Setup

This configures the AS7341 color sensor. It specifies integration time step, sensor gain, LED current, and whether the LED should be on or off.

```
sensor = AS7341(board.STEMMA_I2C())
sensor.astro = 128  # (999) The integration time step size in 2.78 microsecond increments
sensor.atime = 50  # The integration time step count.
sensor.gain = Gain.GAIN_256X
sensor.led_current = 4  # increments in units of 4
sensor.led = True
max_sensor_count = (sensor.astro + 1) * (sensor.atime + 1)
```

Brick Color Values & Names

Here we create two lists: measured color values of each brick on the eight channels, and the corresponding color names.

You can replace these with your own brick colors and names by copying values from the REPL.

```
brick_full_spectrum_values = [
    (94, 1310, 1736, 1075, 592, 437, 497, 383),  # Blue
    (148, 324, 838, 2577, 2363, 1259, 929, 819),  # Bright Green
    (381, 576, 850, 1619, 3688, 5532, 6291, 4250),  # Bright Lt Orange
    (404, 2300, 2928, 2385, 2679, 3804, 5576, 4284),  # Bright Pink
    (545, 1276, 1513, 1178, 2291, 6579, 6579, 6486),  # Coral
    (136, 1055, 1223, 745, 748, 1205, 1100),  # ©Adafruit Industries
```
Dark Purple
(85, 731, 1375, 1604, 1019, 557, 533, 370), # Dark
Turquoise
(451, 2758, 3786, 2880, 3007, 3064, 4539, 3656), #
Lavender
(214, 300, 771, 1811, 3245, 2897, 2051, 1392), #
Lime
(188, 341, 435, 507, 625, 1703, 4361, 3692), # Red
Sand Green
(461, 497, 878, 2412, 4699, 5935, 6579, 4677) #
Yellow
]
brick_color_names = [
  "Blue",
  "Bright Green",
  "Bright Light Orange",
  "Bright Pink",
  "Coral",
  "Dark Purple",
  "Dark Turquoise",
  "Lavender",
  "Lime",
  "Red",
  "Sand Green",
  "Yellow"
]

Brick States

brick states is a list of boolean values that keeps track of whether a specific LEGO brick color is currently detected.

gap state keeps track of when no brick is currently detected.

brick states = [False] * (len(brick_color_names))
gap state = False

Audio Setup

Configuration for audio output and mixer, which is used to set the overall volume. We first have to enable external power via digitalio on the Feather Prop-Maker RP2040 (this is a power saving feature).

```python
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)
audio_output = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)
mixer = audiomixer.Mixer(sample_rate=11020, buffer_size=4096, voice_count=1, channel_count=1)
audio_output.play(mixer)
mixer.voice[0].level = 0.50  # adjust this for overall volume
```
Chime Setup

We'll create a scale specified in note names and octaves called `brickscale` and then instantiate the `Chime` object.

You can adjust the chime's parameters for material, stricker, voice, and `scale_offset`.

We then play through the scale as a startup sound.

```python
brickscale = [
    "C5", "D5", "E5", "F5", "G5", "A5", "B5",
    "C7", "D7", "E7", "F7", "G7", "A7", "B7",
]

# Instantiate the chime synthesizer with custom parameters
chime = Chime(
    mixer.voice[0],
    scale=brickscale,
    material=Material.Brass,  # SteelEMT, Ceramic, Wood, Copper, Aluminum, Brass
    voice=Voice.Tubular,  # bell, perfect, tubular
    scale_offset=-16
)

for index, note in enumerate(chime.scale):
    chime.strike(note, 1.0)
    time.sleep(0.1)
```

Color Compare Function

The function `compare_n_channel_colors` compares the color detected by the sensor with reference color values. It calculates the Euclidean difference between two color values and checks if it's within the specified tolerance.

From CGrover()'s function notes:

> Compares two integer multichannel count tuples using an unweighted linear Euclidean difference. If the color value difference is within the tolerance band of the reference, the method returns `True`.

The difference value index 'tolerance' is used to detect color similarity. Value range is an integer value from 0 to `(maximum_channel_count * number_of_channels)`. Default is 0 (detects a single color value).

```python
def compare_n_channel_colors(color_1, color_2, tolerance=0):
    # Create list of channel deltas using list comprehension
```
deltas = [((color_1[idx] - count) ** 2) for idx, count in enumerate(color_2)]
# Resolve squared deltas to a Euclidean difference
# pylint: disable=c-extension-no-member
delta_color = math.sqrt(sum(deltas))
return bool(delta_color <= tolerance)

You can learn more about the color comparison function in this TV Backlight Playground post.

Main Loop

The main loop continuously reads color data from the sensor. If the color reading indicates a gap between LEGO bricks (based on the value of the first channel), it resets the state of all brick colors to "not detected."

If a color is detected, it compares the detected color to the reference colors.

If there's a match within the tolerance, it plays a musical chime note set corresponding to the detected color.

```python
while True:
    sensor_color = sensor.all_channels
    # don't bother to check comparison when we're looking at a gap between bricks
    if sensor_color[0] <= 70:  
        # this checks for a minimum value on one channel
        if gap_state is False:
            print("no brick...")
            for i in range(len(brick_color_names)):
                brick_states[i] = False
            gap_state = True
        else:
            if DEBUG:
                print(sensor_color)
                for i in range(len(brick_full_spectrum_values)):
                    color_match = compare_n_channel_colors(
                        sensor_color,
                        brick_full_spectrum_values[i],
                        TOLERANCE
                    )
                if color_match is True:
                    if brick_states[i] is False:
                        for n in range(5):
                            chime.strike(chime.scale[i+(n*2)], 1.0)
                            time.sleep(0.1)
                        brick_states[i] = True
                        gap_state = False
                        print("sensor color:", sensor_color, "| ref:",
                            brick_full_spectrum_values[i])
                        print(brick_color_names[i])
                        break
                      ```
Build the Bricktunes Glove

Attach Feather to Glove

Lay the Feather onto the back of the glove and drive an awl or a small screwdriver through the rubber padding to create a hole corresponding to each Feather mounting hole.

Drive an M2.5 x 10mm screw up from inside the glove as shown for each of the holes. (I skipped the fourth hole due to space constraints with the screw terminal block, but three hold it just fine.)
Affix a nut and 6mm F-F hex standoff to each screw.

Use M2.5 x 4mm screws to attach the Feather to the standoffs.
Sensor Positioning and Cabling

Dry fit the sensor position to find a good mounting point and cable run path.

With this particular glove it worked out well to remove a few stitches from the palm patch and run the STEMMA QT cable through it.
Sew the Sensor
Plug the STEMMA QT cable into the board.

Use a needle and thread to sew the sensor board into place on the finger tip.
Mount the Speaker

You can mount the speaker using a similar technique to the Feather.

light_sensors_bricketunes-5026.jpg
Connect Battery
Nothing too fancy here! Use a hair band or other wrap to secure the battery under the Feather and then plug it in as shown.

Play Music with your LEGO Bricks
Grab your bricks in the specific colors we're using and lay them out on a table.
All you have to do is point your finger right above a LEGO piece and the chimes will play!

For a different sound, you can look at the [synthio Fundamentals](#) guide for info on creating different synthio waveforms.

Optional LEGO Bricktune Platform Build

If you'd like to build the over-engineered LEGO color testing platform, go ahead and download the .pdf below.

Here's the [build on Bricklink](#), which makes it easy to order any parts you need as well.

Bricktunes Instructions