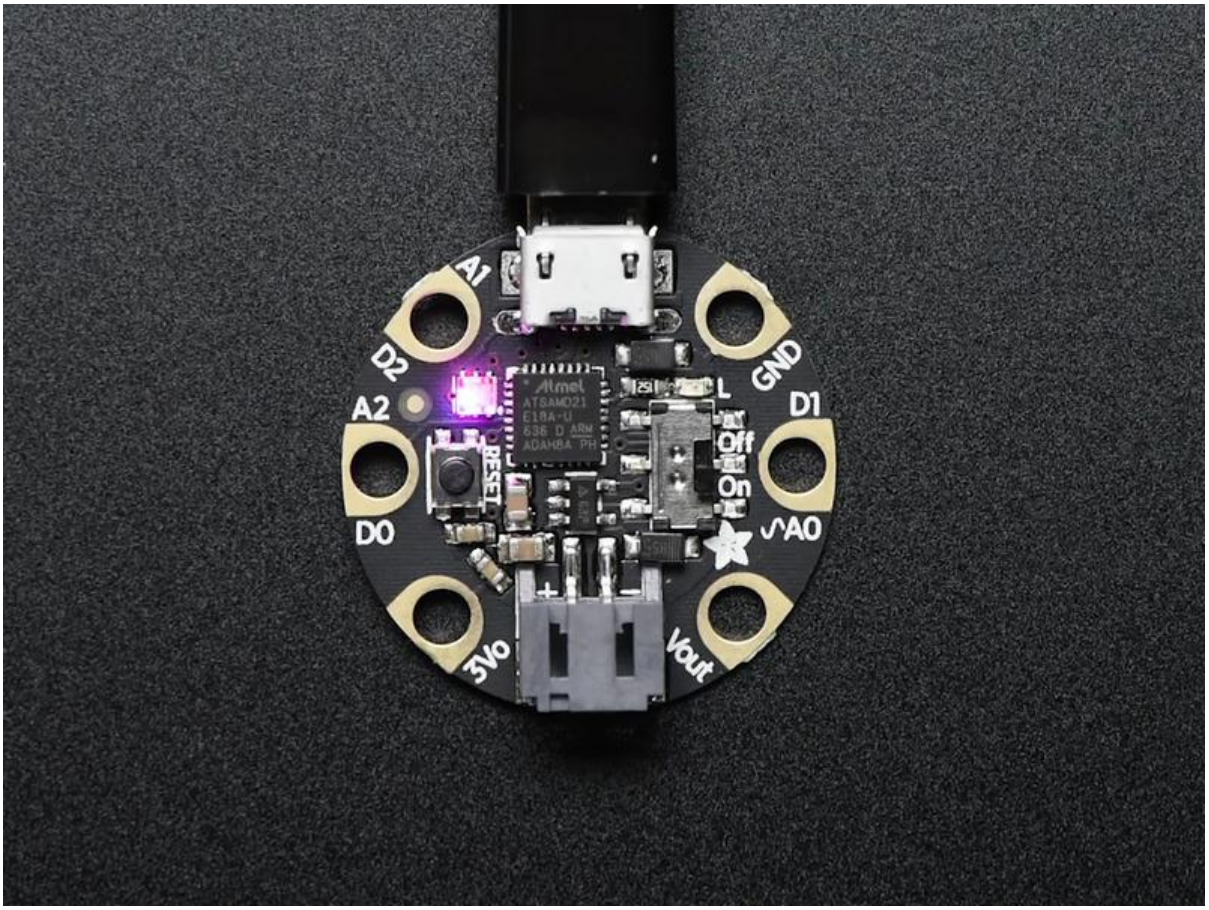
Gemma LightTouch

Created by Kattni Rembor



https://learn.adafruit.com/gemma-lighttouch

Last updated on 2023-08-29 03:42:30 PM EDT

Table of Contents

Red light, green light, blue light!

CircuitPython is Python that runs on microcontrollers, including your Gemma board! To learn more about CircuitPython, check out the Welcome to CircuitPython () and CircuitPython Essentials () guides. This guide uses CircuitPython and Gemma.

This project is a simple one that requires touch interaction from you! There are three touch pads on your Gemma, and it just so happens that there are three colors in the built-in RGB LED! So, we've written up a program using CircuitPython that allows you to control the red level from the first pad, the green level from the second pad, and the blue level from the third to make any color in the rainbow!

RGB LED colors are set using a combination of red, green, and blue, in the form of an (R, G, B) tuple. Each member of the tuple is set to a number between 0 and 255 that determines the amount of each color present. Red, green and blue in different combinations can create all the colors in the rainbow! So, for example, to set the LED to red, the tuple would be (255, 0, 0), which has the maximum level of red, and no green or blue. Green would be (0, 255, 0), etc. For the colors between, you set a combination, such as cyan which is (0, 255, 255), with equal amounts of green and blue. If you increase all values to the same level, you get white!

Let's take a look at the code!

Code Walkthrough

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""Touch each pad to change red, green, and blue values on the LED"""
import time

import adafruit_dotstar
import board
import touchio

led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)
touch_A0 = touchio.TouchIn(board.A0)
touch_A1 = touchio.TouchIn(board.A1)
touch_A2 = touchio.TouchIn(board.A2)

r = g = b = 0

while True:
    if touch_A0.value:
        r = (r + 1) % 256
    if touch_A1.value:
        g = (g + 1) % 256
    if touch_A2.value:
        b = (b + 1) % 256
```

```
    led[0] = (r, g, b)
    print((r, g, b))
    time.sleep(0.01)
```

First we import our libraries: `time`, `touchio`, `adafruit_dotstar`, and `board`.

Next, we create the LED and touch objects.

DotStar LEDs () use SPI, but the DotStar on the Gemma has its own unique pin assignments. So we assign `led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)` to tell it which pins to use. The `1` tells the code that we are using a single LED.

If you look at your board, you'll see three pads that have A0, A1 and A2 next to them. Those are the touch pads on your Gemma. We have three touch pads, so we create three touch objects. To create a touch object, you need to provide the pin you plan to use. We start with `touch_A0 = touchio.TouchIn(board.A0)` and then use the same concept for `touch_A1` and `touch_A2`.

```
led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)
touch_A0 = touchio.TouchIn(board.A0)
touch_A1 = touchio.TouchIn(board.A1)
touch_A2 = touchio.TouchIn(board.A2)
```

Since we're going to be using r, g, b color values, we need to initialise the variables for later use. So we set `r = g = b = 0`.

We create our main loop with `while True:`.

Within the loop, we have three `if` statements. Each are the same concept, using a touch pad to change a color value.

We start with `if touch_A0.value:`, which says, "if this touch pad is touched, do the following:". We are going to use touch pad A0 to change the red color level. So, we have `r = (r + 1) % 256`. This says every time you touch the pad, increase the red value by 1 up to a maximum value of 255, and loop once the maximum is reached. This cycles from 0 to 255 and then back to 0. However, you don't need to touch the pad 256 times to get through the cycle! The touch pads respond if you touch and hold, so you can place your finger on A0 and leave it, and it will go through the entire cycle.

```
    if touch_A0.value:
        r = (r + 1) % 256
```

Then we do the same for A1 and A2, where A1 affects the green value and A2 affects the blue value.

```
    if touch_A1.value:
        g = (g + 1) % 256
    if touch_A2.value:
        b = (b + 1) % 256
```

Then, we set `led[0] = (r, g, b)`. This set the LED to the color that matches the current values that you've set by touching the different pads! Note that we have 1 LED, however we've set `led[0]`. This is because Python starts counting at 0. So our first LED is number 0 according to the code.

Then we print the color values to the serial console with `print((r, g, b))`. If you are connected to the serial console, you'll see the values change live as you touch the pads!

Last, we include a `time.sleep(0.01)` to include a tiny delay. Otherwise the colors can change so quickly you'll pass right by the color you were trying to set! You can change this if you'd like to make the values change more quickly.

You don't have to change each value separately! You can touch any number of pads at the same time to affect multiple color values at the same time.

That's all there is to it! Simple code with a little math, and you've got a way to use the touch pads on your Gemma to show off any color you'd like!

Fancy an interactive light?

CircuitPython is Python that runs on microcontrollers, including your Gemma board! To learn more about CircuitPython, check out the Welcome to CircuitPython () and CircuitPython Essentials () guides. This guide uses CircuitPython and Gemma.

This project gets super fancy! We're using the LED and the touch pads, like we do in the Red light, green light blue light project. However, we've added some very different functionality to this CircuitPython program. We'll have solid colors like the first one, but we also have some Python-colored blinky fun and an awesome rainbow! As well, you'll be able to change the speed of the blinking and the rainbow, and change the brightness of the LED. We're going to use helper functions, generators (), dictionaries (), and state machines () to make this happen, all while keeping the code non-blocking ().

All of the concepts used in this program are introduced and explained in the Hacking Ikea Lamps with Circuit Playground Express Guide (). Each concept is linked to the applicable section of the guide. If you'd like more detailed explanations of everything we use here, check it out!

We have three different inputs, the three touch pads on your Gemma board. These inputs will control different modes, speeds, brightness.

We'll use:

- Touch pad A0 to change color modes
- Touch pad A1 to change speeds
- Touch pad A2 to change brightness

The five different modes that touch pad A0 will cycle through are:

- a smooth rainbow cycle
- a blinking blue and yellow Python-colored blinking mode
- three static solid colors: red, green and blue.

Let's take a look at the code!

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""Interactive light show using built-in LED and capacitive touch"""
import time
from rainbowio import colorwheel
import adafruit_dotstar
import board
import touchio

led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)
touch_A0 = touchio.TouchIn(board.A0)
touch_A1 = touchio.TouchIn(board.A1)
touch_A2 = touchio.TouchIn(board.A2)


def cycle_sequence(seq):
    """Allows other generators to iterate infinitely"""
    while True:
        for elem in seq:
            yield elem


def rainbow_cycle(seq):
    """Rainbow cycle generator"""
    rainbow_sequence = cycle_sequence(seq)
    while True:
        # pylint: disable=stop-iteration-return
        led[0] = (colorwheel(next(rainbow_sequence)))
        yield
```

```
def brightness_cycle():
    """Allows cycling through brightness levels"""
    brightness_value = cycle_sequence([1, 0.8, 0.6, 0.4, 0.2])
    while True:
        # pylint: disable=stop-iteration-return
        led.brightness = next(brightness_value)
        yield


color_sequences = cycle_sequence(
    [
        range(256),  # rainbow_cycle
        [50, 160],  # Python colors!
        [0],  # red
        [85],  # green
        [170],  # blue
    ]
)

cycle_speeds = cycle_sequence([0.1, 0.3, 0.5])

brightness = brightness_cycle()

CYCLE_SPEED_INITIAL = 0.3
cycle_speed_start = time.monotonic()
cycle_speed = cycle_speed_start + CYCLE_SPEED_INITIAL

rainbow = None
touch_A0_state = None
touch_A1_state = None
touch_A2_state = None

while True:
    now = time.monotonic()

    if not touch_A0.value and touch_A0_state is None:
        touch_A0_state = "ready"
    if touch_A0.value and touch_A0_state == "ready" or rainbow is None:
        rainbow = rainbow_cycle(next(color_sequences))
        touch_A0_state = None

    if now >= cycle_speed:
        next(rainbow)
        cycle_speed_start = now
        cycle_speed = cycle_speed_start + CYCLE_SPEED_INITIAL

    if not touch_A1.value and touch_A1_state is None:
        touch_A1_state = "ready"
    if touch_A1.value and touch_A1_state == "ready":
        CYCLE_SPEED_INITIAL = next(cycle_speeds)
        cycle_speed_start = now
        cycle_speed = cycle_speed_start + CYCLE_SPEED_INITIAL
        touch_A1_state = None

    if not touch_A2.value and touch_A2_state is None:
        touch_A2_state = "ready"
    if touch_A2.value and touch_A2_state == "ready":
        next(brightness)
        touch_A2_state = None
```

Setup

First we import our libraries: `time`, `touchio`, `adafruit_dotstar`, and `board`.

```
import time
import adafruit_dotstar
import touchio
import board
```

Next, we create the LED and touch objects.

DotStar LEDs use SPI, but the DotStar on the Gemma has its own unique pin assignments. So we assign `led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)` to tell it which pins to use. The `1` tells the code that we are using a single LED.

```
led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)
```

If you look at your board, you'll see three pads that have A0, A1 and A2 next to them. Those are the touch pads on your Gemma. We have three touch pads, so we create three touch objects. To create a touch object, you need to provide the pin you plan to use. We start with `touch_A0 = touchio.TouchIn(board.A0)` and then use the same concept for `touch_A1` and `touch_A2`.

```
touch_A0 = touchio.TouchIn(board.A0)
touch_A1 = touchio.TouchIn(board.A1)
touch_A2 = touchio.TouchIn(board.A2)
```

Helper Function

To learn more about `wheel()`, check out this page ().

```
def wheel(pos):
    """ Input a value 0 to 255 to get a color value.
    The colours are a transition r - g - b - back to r."""
    if pos &lt; 0 or pos &gt; 255:
        return 0, 0, 0
    if pos &lt; 85:
        return int(255 - pos * 3), int(pos * 3), 0
    if pos &lt; 170:
        pos -= 85
        return 0, int(255 - pos * 3), int(pos * 3)
    pos -= 170
    return int(pos * 3), 0, int(255 - (pos * 3))
```

Our helper function is called `wheel()`, and is necessary for the rainbow cycle to work. `wheel()` allows us to specify any color using a single number by requiring a position, `(pos)`. Since we're already using it for the rainbow, we'll also be using it for all of our colors.

Generators

To learn more about these generators, check out this page ().

Our first generator is called `cycle_sequence()` and is designed to allow the other generators to iterate infinitely instead of stopping after the initial set of values.

```
def cycle_sequence(seq):
    """Allows other generators to iterate infinitely"""
    while True:
        for elem in seq:
            yield elem
```

Next we have `rainbow_cycle()` which iterates through the entire set of rainbow colors and starts again to create our rainbow cycle. Normally the rainbow cycle must complete before any changes can be made, but the generator allows us to provide input at any time. That way we can change the mode or the brightness at any time without waiting until the cycle is complete!

```
def rainbow_cycle(seq):
    """Rainbow cycle generator"""
    rainbow = cycle_sequence(seq)
    while True:
        led[0] = (wheel(next(rainbow)))
        yield
```

The `brightness_cycle()` generator iterates though the different brightness levels. Brightness is assigned using any number `0` through `1`, which represents 0-100%. So, a brightness level of `0.3` is 30% brightness, and a brightness of `0.07` is 7% brightness. We've included brightness levels of `0.2`, `0.4`, `0.6`, `0.8`, and `1`, beginning with `1` and decreasing each time.

```
def brightness_cycle():
    """Allows cycling through brightness levels"""
    brightness_value = cycle_sequence([1, 0.8, 0.6, 0.4, 0.2])
    while True:
        led.brightness = next(brightness_value)
        yield
```

The next two generators use `cycle_sequence` to iterate through a list of values. The first, `color_sequences`, is a list containing the different `(pos)` position values that will be provided to `wheel`. The second generator, `cycle_speeds`, contains the speed of our modes in seconds. To be clear, this is not the speed to cycle between modes - that will be done with user input. This affects the speed of the rainbow and Python-color blink mode.

```
color_sequences = cycle_sequence(
    [
        range(256),  # rainbow_cycle
        [50, 160],  # Python colors!
        [0],  # red
        [85],  # green
        [170],  # blue
    ]
)

cycle_speeds = cycle_sequence([0.1, 0.3, 0.5])
```

We assign `brightness = brightness_cycle()` so we can use the `brightness_cycle()` generator later.

Time

To learn more about non-blocking time code, check out this page ().

```
cycle_speed_initial = 0.3
cycle_speed_start = time.monotonic()
cycle_speed = cycle_speed_start + cycle_speed_initial
```

Here is where we begin our non-blocking time code. `time.monotonic()` is a time in seconds since your board was last power cycled. To use `time.monotonic()`, you must assign it to variables so you can compare different points in time to figure out when to allow the code to continue. So, you find the time delta by subtracting a start time from a current time. Here we have `cycle_speed_initial`, the initial cycle speed constant of `0.3` seconds. We have `cycle_speed_start`, the start speed assigned to `time.monotonic()`, and a `cycle_speed` that subtracts the initial speed from the start speed to create a delay of 0.3 seconds in a non-blocking variable available for use later.

Variables

We have four variables we initialise for use later.

```
rainbow = None
touch_A0_state = None
touch_A1_state = None
touch_A2_state = None
```

`rainbow` is so we can call the `rainbow_cycle` generator. The three `touch_state` variables are for use in the state machines that we'll use to keep the touch pads from spamming touch values when you touch and hold on any given pad.

Main Loop

We begin our main loop with `while True:`.

First we create a current time for comparison by assigning `now = time.monotonic()`.

In the first project, we learned that we can touch and hold on the touch pad and they will continue to respond. For this project, that won't work. We are cycling through different modes and would like to be able to control which mode at which we would like to stop. So, we're going to create a state machine that causes the touch pad to respond once each time we touch it. That way we can cycle through one mode at a time, regardless of how long we touch the touch pad.

Our first state machine utilises touch pad A0.

```
    if not touch_A0.value and touch_A0_state is None:
        touch_A0_state = "ready"
```

We're going to use our touch_A0_state variable. Remember, we assigned it to None before our loop. Here we're saying, "if we haven't touched touch pad A0, and `touch_A0_state is None`, then assign `touch_A0_state = "ready"`."

```
    if touch_A0.value and touch_A0_state == "ready" or rainbow is None:
        rainbow = rainbow_cycle(next(color_sequences))
        touch_A0_state = None
```

Then we use that state to determine that we're ready to accept touch input. After we accept the single touch input, we call our rainbow generator,. Then we reassign `touch_A0_state = None` so we can begin again.

Inside our state machine, we called the rainbow generator. Generators are used by calling `next`. The way the `rainbow_cycle` generator works is by calling `next` on the `color_sequences` generator, which provides `rainbow_cycle` with `wheel()` positions.

Our rainbow_cycle generator accepts ranges, static colors and groups of colors, by utilising wheel(). wheel() allows for a position of range(256) which cycles through all available colors, single numbers, which returns static colors, and groups of colors, which allows for blinking.

We use the next section of code to determine the speed at which we are calling `next` on `rainbow`.

```
    if now &gt;= cycle_speed:
        next(rainbow)
        cycle_speed_start = now
        cycle_speed = cycle_speed_start + cycle_speed_initial
```

This is where we set the speed of each color mode. Solid colors don't care about speed and simply aren't affected. This speed is important to the rainbow and blink modes.

Next, we have another state machine, with exactly the same syntax and concept as the first, used with touch pad A1.

```
    if not touch_A1.value and touch_A1_state is None:
        touch_A1_state = "ready"
    if touch_A1.value and touch_A1_state == "ready":
```

Touch pad A1 is used to change the speed of the blinking and the rainbow. So we have a similar set of `time` code as we had in our setup.

```
        cycle_speed_initial = next(cycle_speeds)
        cycle_speed_start = now
        cycle_speed = cycle_speed_start + cycle_speed_initial
```

Then we assign `touch_A1_state = None` so the state machine is ready to begin again.

The last bit of code is the third state machine used with touch pad A2.

```
    if not touch_A2.value and touch_A2_state is None:
        touch_A2_state = "ready"
    if touch_A2.value and touch_A2_state == "ready":
```

We use touch pad A2 to cycle through the brightness levels by calling `next` on the `brightness` generator.

```
        next(brightness)
```

And with that, we reach the end of our program! Put it all together, and you get a fancy interactive tough light using your new Gemma!

While you're welcome to edit any part of the program, there's a few things in here that you can easily change up for different effects. Let's take a look!

Change it up!

Colors and Blinks

You can change the blink modes or the static colors by adding more data to the list contained within the `color_sequences` generator.

To add another static color mode, you would add another line similar to the red, green or blue lines. For example, if you would like to include orange, you would add another line containing `[10],`, which is the `wheel()` position that returns orange. Adding orange might look like the following example.

```
color_sequences = cycle_sequence(
    [
        range(256),  # rainbow_cycle
        [50, 160],  # Python colors!
        [0],  # red
        [10],  # orange
        [85],  # green
        [170],  # blue
    ]
)
```

The code on this page () has seven different static colors. Check it out to get some ideas of what colors are in different positions in `wheel()`!

To add another blinking mode, you would provide another line similar to the Python colors blink line. For example, if you would like to have a cyan and purple blinking mode, you would add another line containing `[137, 213],`, which includes the two `wheel()` positions that return cyan and purple. It may look like the following.

```
color_sequences = cycle_sequence(
    [
        range(256),  # rainbow_cycle
        [50, 160],  # Python colors!
        [137, 213]  # Cyan and purple
        [0],  # red
        [85],  # green
        [170],  # blue
    ]
)
```

You can blink as many different colors in one mode as you'd like. To add a mode that blinks seven different colors, you'd add a line containing `[0, 10, 30, 85, 137, 170, 213],`. Adding it may look like the following.

```
color_sequences = cycle_sequence(
    [
        range(256),  # rainbow_cycle
```

```
        [50, 160],  # Python colors!
        [0],  # red
        [85],  # green
        [170],  # blue
        [0, 10, 30, 85, 137, 170, 213], # Party mode!
    ]
)
```

It's really easy to add all kinds of different modes to your Gemma light!

Brightness

You can change the available brightness levels. Remember, brightness is assigned by using a number from 0-1 to represent 0-100%. So to change the brightness levels that the code cycles through, you simply need to add to or remove some of the brightness levels in `brightness_cycle` generator.

For example, if you would like to have a dimmer option than is already available, you could add `0.1,` into the the list in the line reading `brightness_value = cycle_sequence([0.1, 1, 0.8, 0.6, 0.4, 0.2])`.

If you would rather have less brightness options, remove some of the numbers in the list. For example, `[0.1, 0.5, 1]` cycles between 10%, 50% and 100% without all the steps in between.

As well, if you would like to change the order, you can move the numbers around in the list so it cycles in a different order. That might look like `[0,2, 0.4, 0.6, 0.8, 1]`. You can do it however you'd like!

Speeds

You can change the available speeds. The speed generator works exactly like the brightness generator in terms of making alterations to it. You will add to, remove from, or change the order of the current list of numbers in the line reading `cycle_speeds = cycle_sequence([0.1, 0.3, 0.5])`. The current available speeds are 0.1 seconds, 0.3 seconds and 0.5 seconds. Remember, these are the delays between changes in the modes that change, such as the rainbow mode and the blink mode.

For example, if you'd like to add a slower option, your new list might look like `[0.1, 0.3, 0.5, 0.7]`. If you wanted to add a faster option, your new list might look like `[0.05, 0.1, 0.3, 0.5]`. If you wanted to change the order that the speeds cycle, your new list might look like `[0.5, 0.3, 0.1]`.

You can also change the speed it starts at by changing `CYCLE_SPEED_INITIAL = 0.3` to be a different number of seconds. If you'd like it to start out faster, try `0.1` seconds. If you'd like it to start out slower, try `0.5`. You can change it to any number of seconds you'd like. It doesn't have to be a number within the `cycle_speeds` list.

It's super simple to customise the speed options!