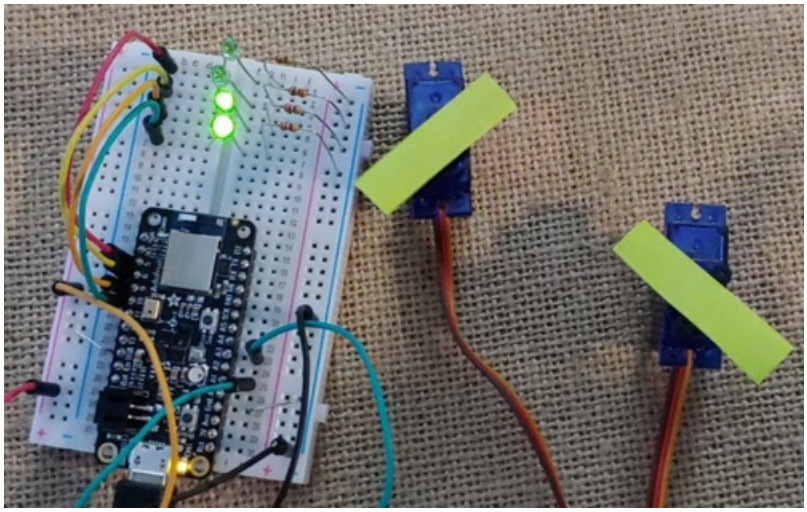
adafruit learning system

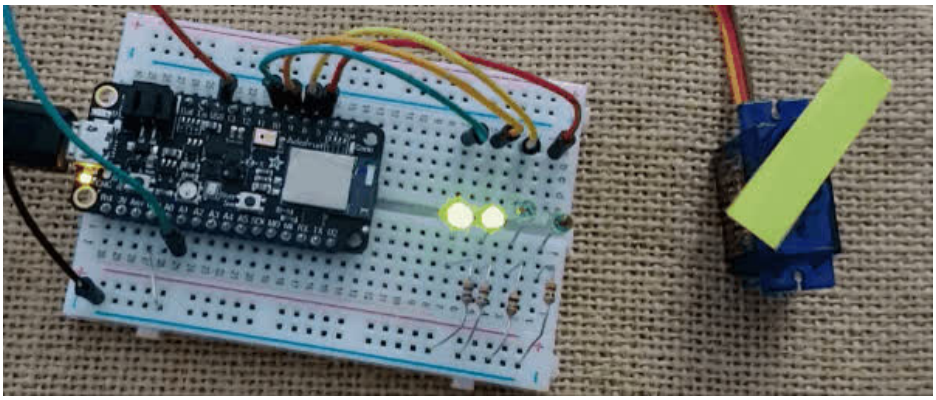
Multi-tasking with CircuitPython

Created by Tim C



Last updated on 2020-10-08 02:08:27 PM EDT

Overview



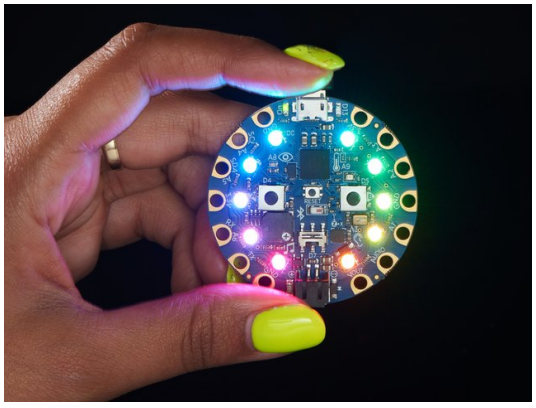
Once you've learned the basics of how to blink lights, move servos, and handle inputs, it often comes naturally to want make a larger project that does more than one thing -- more LEDs, more servos, and more buttons and other inputs.

One thing you'll quickly find is that as you add more things to your circuit and code, they are each delaying each other. So one blinking LED works fine, but when you add the second, their blinking patterns are always affecting each other. If you slow one down, the other slows as well.

If you add multiple servos, you might notice that you can't get them to move at the same time. One completes its full sweeping motion, and then the other completes its full sweep. They move sequentially, but never at the same time.

This guide will teach you a technique that you can use to manage multiple things happening at once in your CircuitPython project.

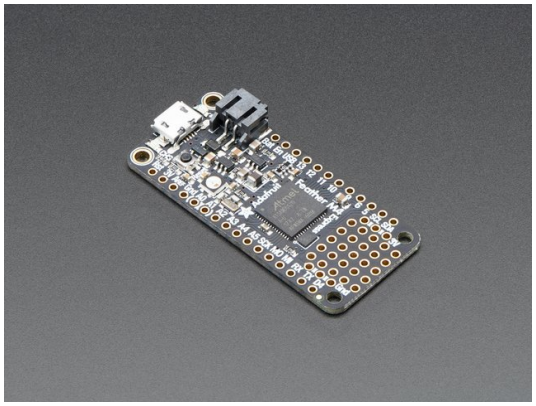
Parts



Circuit Playground Bluefruit - Bluetooth Low Energy

OUT OF STOCK

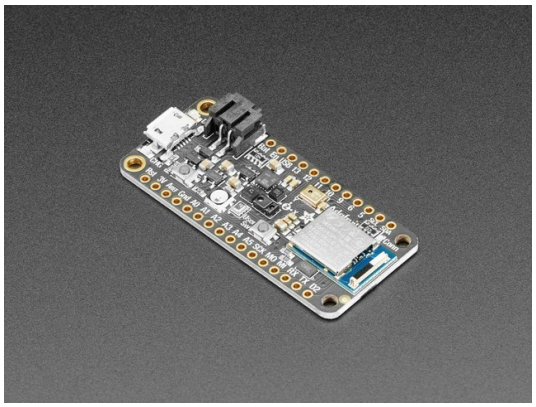
Out Of Stock



Adafruit Feather M4 Express - Featuring ATSAMD51

OUT OF STOCK

Out Of Stock



Adafruit Feather nRF52840 Sense

OUT OF STOCK

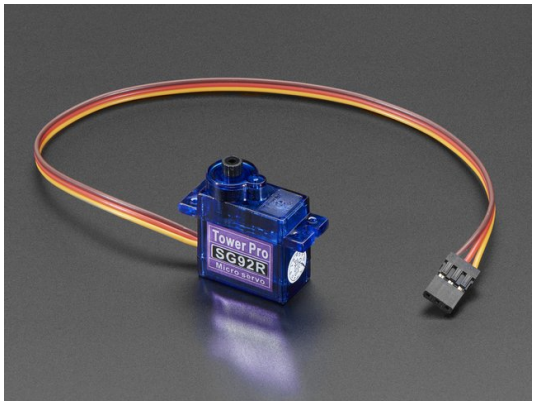
Out Of Stock

Adafruit Parts Pal

$19.95

IN STOCK

Add To Cart

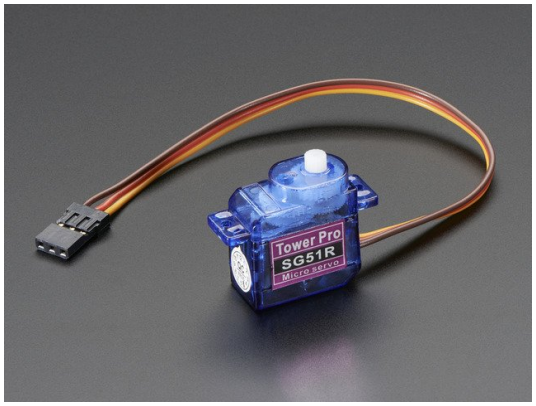


Micro servo

$5.95

IN STOCK

Add To Cart



Sub-micro Servo - SG51R

$5.95
IN STOCK

Add To Cart

USB cable - USB A to Micro-B

$2.95
IN STOCK

Add To Cart

No Sleeping

Blink

For many of us, the first program we see or run on a new platform is some variation of "Blink" to turn an LED on and off at some rate. It has become like the "Hello World" of microcontroller programming.

Here is a CircuitPython implementation:

```
import time
import digitalio
import board

# Setup the LED pin.
led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5) # Wait until we want to turn LED on.
    led.value = False
    time.sleep(0.5) # Wait until we want to turn LED off.
```

This will turn the LED on and then wait, or in other words, `time.sleep()` for some time and then turn the LED back off. We can even use different values for the on and off times by changing the `time.sleep()` parameters.

So what's the problem?

`time.sleep()` blocks the program while it is running. Nothing else can occur. In the basic blink example, this doesn't matter so much because we aren't trying to do anything else. But let's say we wanted to add a second LED and blink it at a different rate while still continuing to blink the first. Using `time.sleep()` will cause trouble because each LED will have no choice but to wait for the sleep calls that are blinking the other. Their blink rates won't ever be truly independent.

How can we work around this problem?

Is it time yet?

So we know that sleeping until it's time to act is causing trouble, but what we can we do to fix it? One possible solution is to change from sleeping to asking "is it time yet" over and over, just like an annoying sibling on a long car ride.

Our program already has a "main loop" that will execute over and over very fast, or at least it would if we allow it to by not taking up time with long-running blocking things like `time.sleep()`.

We can make use of `time.monotonic()` to check what time it is, then compare that against the last time we changed the LED to decide "is it time yet" to change the LED again.

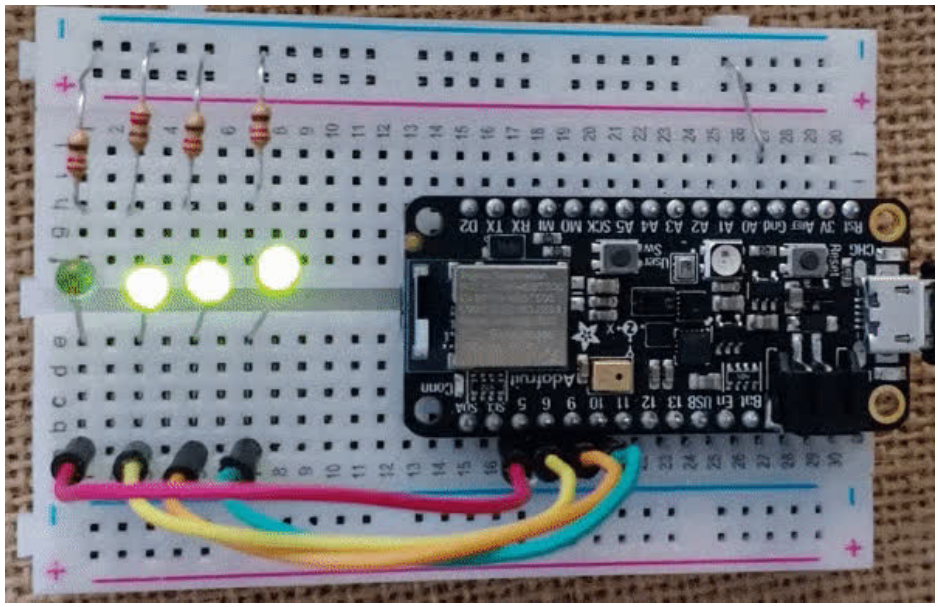
```
"""
Using time.monotonic() to blink the built-in LED.

Instead of "wait until" think "Is it time yet?"
"""
import time
import digitalio
import board

# How long we want the LED to stay on
BLINK_ON_DURATION = 0.5

# How long we want the LED to stay off
BLINK_OFF_DURATION = 0.25

# When we last changed the LED state
LAST_BLINK_TIME = -1

# Setup the LED pin.
led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

while True:
    # Store the current time to refer to later.
    now = time.monotonic()
    if not led.value:
        # Is it time to turn on?
        if now >= LAST_BLINK_TIME + BLINK_OFF_DURATION:
            led.value = True
            LAST_BLINK_TIME = now
    if led.value:
        # Is it time to turn off?
        if now >= LAST_BLINK_TIME + BLINK_ON_DURATION:
            led.value = False
            LAST_BLINK_TIME = now
```

By avoiding the use of `time.sleep()`, we've made it so that our main loop can complete each iteration *very quickly.* Now we check each time through the loop whether it's time to change the LED state or not. To determine this, we check the current time against a variable storing the last time we acted, along with a duration variable.

Once it is time to act, we change the LED state and update the variable, storing the last time we acted.

By using multiple duration variables, we can achieve different lengths of time for the various states. In this case we have the LED staying on for twice as long as it is off — `0.5` seconds ON and `0.25` seconds OFF.

Multiple LEDs



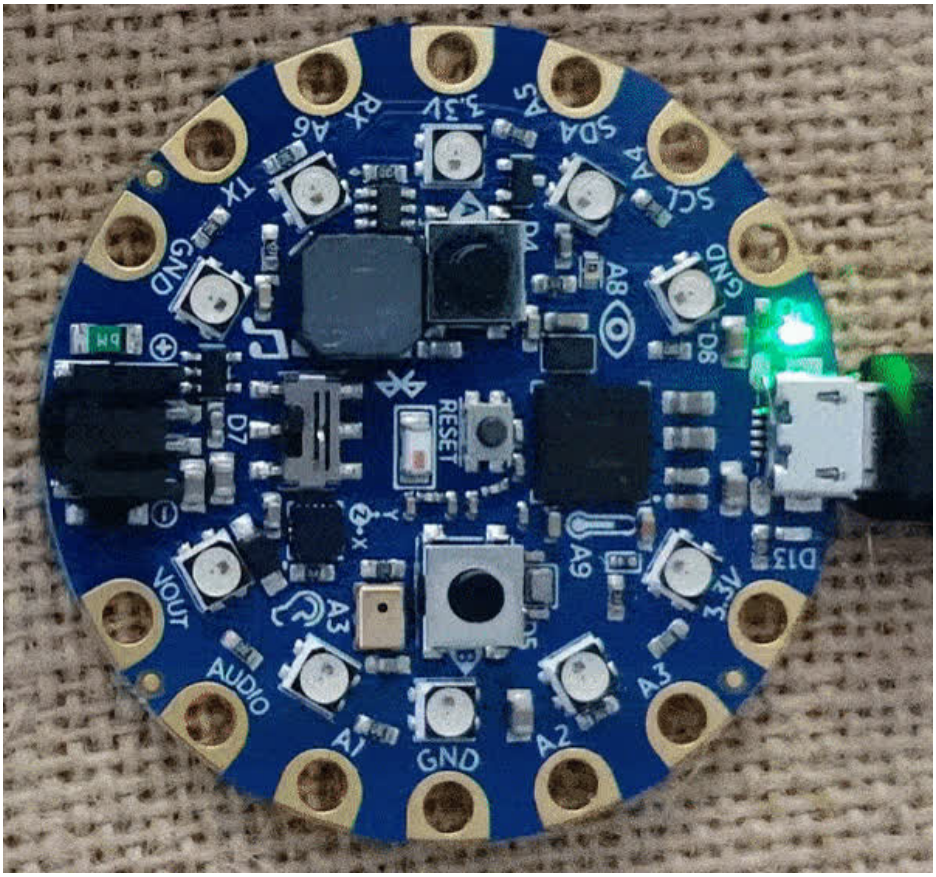
Okay, we've learned how to make our blink script without relying on `time.sleep()`, but if we only use one LED, then it hardly makes any difference to us really. Let's add some more LEDs into the mix so we can see the the major advantage of ditching the `time.sleep()` calls.

This script is made to run on a Feather M4 Express, Feather nRF52840, or Feather Bluefruit Sense. With minimal tweaking, it could be adjusted to work with the Metro M4 or ItsyBitsy M4 devices.

All of the pins and other information we need to manage the blinking schedule is packed into a list of dictionaries. This makes it easy for us to iterate over them and check each one, taking action as needed to turn LEDs ON or OFF.

```
"""
This example script shows how to blink multiple LEDs at different
rates simultaneously without each affecting the others.
"""

import board
import digitalio
import time

BLINK_LIST = [
    {
        "ON": 0.25,
        "OFF": 0.25,
        "PREV_TIME": -1,
        "PIN": board.D5,
    },
    {
        "ON": 0.5,
        "OFF": 0.5,
        "PREV_TIME": -1,
        "PIN": board.D6,
    },
    {
        "ON": 1.0,
        "OFF": 1.0,
        "PREV_TIME": -1,
        "PIN": board.D9,
    },
    {
        "ON": 2.0,
        "OFF": 2.0,
        "PREV_TIME": -1,
        "PIN": board.D10,
    }
]

for led in BLINK_LIST:
    led["PIN"] = digitalio.DigitalInOut(led["PIN"])
    led["PIN"].direction = digitalio.Direction.OUTPUT

while True:
    # Store the current time to refer to later.
    now = time.monotonic()

    for led in BLINK_LIST:
        if led["PIN"].value == False:
            if now >= led["PREV_TIME"] + led["OFF"]:
                led["PREV_TIME"] = now
                led["PIN"].value = True
        if led["PIN"].value == True:
            if now >= led["PREV_TIME"] + led["ON"]:
                led["PREV_TIME"] = now
                led["PIN"].value = False
```

Circuit Playground Bluefruit



In this script, we pack all of the blinking rules into a dictionary that we can iterate over to carry out the correct blink actions at the correct times and on the correct LEDs.

You can add more LEDs into the mix by adding new entries into the dictionary.

```
"""
Blinking multiple LEDs at different rates.

Circuit Playground Neopixels.
"""
import time
import digitalio
import board
from adafruit_circuitplayground import cp

BLINK_MAP = {
    "RED": {
        "ON": 0.25,
        "OFF": 0.25,
        "PREV_TIME": -1,
        "INDEX": 1,
        "COLOR": (255, 0, 0)
    },
    "GREEN": {
        "ON": 0.5,
        "OFF": 0.5,
        "PREV_TIME": -1,
        "INDEX": 3,
        "COLOR": (0, 255, 0)
```

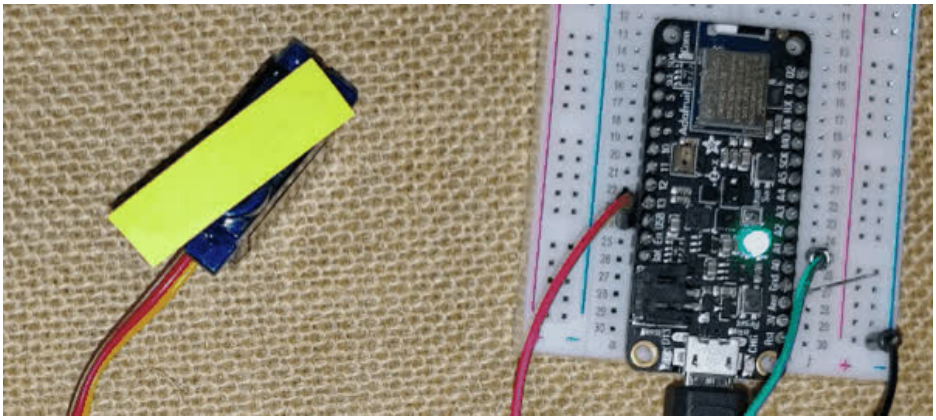
```
    },
    "BLUE": {
        "ON": 1.0,
        "OFF": 1.0,
        "PREV_TIME": -1,
        "INDEX": 6,
        "COLOR": (0, 0, 255)
    },
    "YELLOW": {
        "ON": 2.0,
        "OFF": 2.0,
        "PREV_TIME": -1,
        "INDEX": 8,
        "COLOR": (255, 255, 0)
    }
}

cp.pixels.brightness = 0.02

while True:
    # Store the current time to refer to later.
    now = time.monotonic()

    for color in BLINK_MAP.keys():

        # Is LED currently OFF?
        if cp.pixels[BLINK_MAP[color]["INDEX"]] == (0, 0, 0):
            # Is it time to turn ON?
            if now >= BLINK_MAP[color]["PREV_TIME"] + BLINK_MAP[color]["OFF"]:
                cp.pixels[BLINK_MAP[color]["INDEX"]] = BLINK_MAP[color]["COLOR"]
                BLINK_MAP[color]["PREV_TIME"] = now
        else:  # LED is ON:
            # Is it time to turn OFF?
            if now >= BLINK_MAP[color]["PREV_TIME"] + BLINK_MAP[color]["ON"]:
                cp.pixels[BLINK_MAP[color]["INDEX"]] = (0, 0, 0)
                BLINK_MAP[color]["PREV_TIME"] = now
```

If you want to do more advanced animations with NeoPixels, there is a library made especially for that check out this NeoPixel LED Animation Guide (https://adafru.it/NKa) to learn all about it.

Servos

Let's look at the most basic example from the Servo guide (https://adafru.it/DOf):

```
import time
import board
import pulseio
from adafruit_motor import servo

# create a PWMOut object on Pin A2.
pwm = pulseio.PWMOut(board.A2, duty_cycle=2 ** 15, frequency=50)

# Create a servo object, my_servo.
my_servo = servo.Servo(pwm)

while True:
    for angle in range(0, 180, 5):  # 0 - 180 degrees, 5 degrees at a time.
        my_servo.angle = angle
        time.sleep(0.05)
    for angle in range(180, 0, -5): # 180 - 0 degrees, 5 degrees at a time.
        my_servo.angle = angle
        time.sleep(0.05)
```



This works fine for us if we only have one servo, and we don't have anything else that we want to happen at the same time. But once we add in another thing, we'd have the same trouble as before — `time.sleep()` delaying our other actions unintentionally.

In the next version we adopt the "is it time yet" approach in order to avoid the use of `time.sleep()`:

```
"""
This example script shows how to sweep servo(s) without using
time.sleep().
"""

import board
import digitalio
import time
import pulseio
from adafruit_motor import servo

SERVO_LIST = [
    {
        "MAX_ANGLE": 180,
        "MIN_ANGLE": 0,
        "PREV_TIME": -1,
        "PIN": board.A1,
        "DELAY_BETWEEN": 0.05,
        "SERVO": None,
        "MOVE_BY": 5
    }
]

for cur_servo in SERVO_LIST:
    pwm = pulseio.PWMOut(cur_servo["PIN"], duty_cycle=2 ** 15, frequency=50)
    # Create a servo object.
    cur_servo["SERVO"] = servo.Servo(pwm)


while True:
    # Store the current time to refer to later.
    now = time.monotonic()
    for servo in SERVO_LIST:
        if now >= servo["PREV_TIME"] + servo["DELAY_BETWEEN"]:
            try:
                servo["SERVO"].angle += servo["MOVE_BY"]
            except ValueError as e:

                if servo["MOVE_BY"] > 0:
                    servo["SERVO"].angle = servo["MAX_ANGLE"]
                else:
                    servo["SERVO"].angle = servo["MIN_ANGLE"]

            if servo["SERVO"].angle >= servo["MAX_ANGLE"] or \
                servo["SERVO"].angle <= servo["MIN_ANGLE"]:

                    servo["MOVE_BY"] = -servo["MOVE_BY"]

            servo["PREV_TIME"] = now
```

This version also moves the configuration for the servo sweeping into a list of dictionaries so that additional servos with different sweeping effects can be added easily.

To add an additional servo, you would make a copy of the dictionary that is in `SERVO_LIST` and change any of the fields that we want to the new desired parameters:

```
{
    "MAX_ANGLE": 90,
    "MIN_ANGLE": 45,
    "PREV_TIME": -1,
    "PIN": board.A2,
    "DELAY_BETWEEN": 0.03,
    "SERVO": None,
    "MOVE_BY": 5
}
```

`SERVO` and `PREV_TIME` should be left as their default values. The rest of the code will fill them in when needed.

`MAX_ANGLE` and `MIN_ANGLE` are the degrees at which the servo will turn around to move the opposite direction.

`PIN` is the IO pin that the servo is connected to.

`DELAY_BETWEEN` is how long it will wait between each servo step. This is in seconds, and you likely want to keep it pretty small in most situations.

`MOVE_BY` is how many steps the servo will take each time it moves.

Here is an example of a list with two servos in it. You could continue adding more if you wanted.

```
SERVO_LIST = [
    {
        "MAX_ANGLE": 180,
        "MIN_ANGLE": 0,
        "PREV_TIME": -1,
        "PIN": board.A1,
        "DELAY_BETWEEN": 0.05,
        "SERVO": None,
        "MOVE_BY": 5
    },
    {
        "MAX_ANGLE": 90,
        "MIN_ANGLE": 45,
        "PREV_TIME": -1,
        "PIN": board.A2,
        "DELAY_BETWEEN": 0.03,
        "SERVO": None,
        "MOVE_BY": 5
    }
]
```

Buttons

Another place that we sometimes see `time.sleep()` used is for a sort of minimal "debouncing" effect with push buttons. Some code for that might look like this:

```
import time
import board
from digitalio import DigitalInOut, Direction, Pull

btn = DigitalInOut(board.SWITCH)
btn.direction = Direction.INPUT
btn.pull = Pull.UP

while True:
    if not btn.value:
        print("BTN is down")
    else:
        #print("BTN is up")
        pass

    time.sleep(0.1) # sleep for debounce
```

In this situation, the `time.sleep()` aims to reduce the speed at which the button will continue registering more actions if the button is held down.

Instead, we could store a variable containing the state of the button from the previous iteration, and then compare against it each time. That way, the button will only trigger a single action when it's pressed, or when it's released if we want that instead.

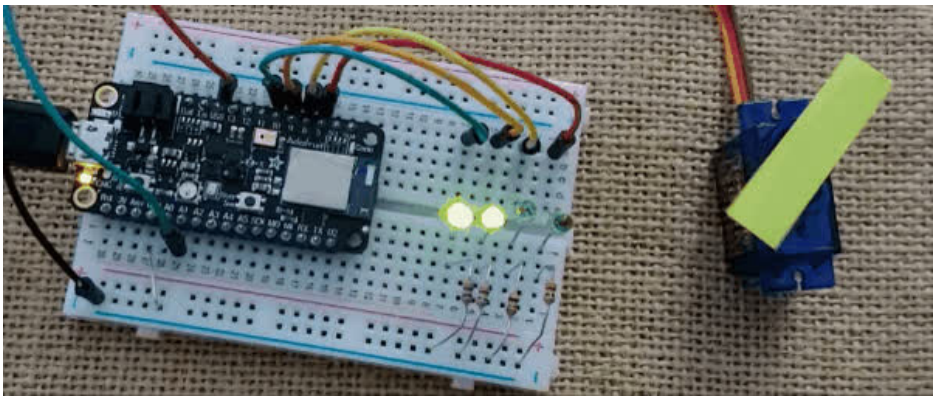
```
"""
This example script shows how to read button state with
debouncing that does not rely on time.sleep().
"""

import board
from digitalio import DigitalInOut, Direction, Pull

btn = DigitalInOut(board.SWITCH)
btn.direction = Direction.INPUT
btn.pull = Pull.UP

prev_state = btn.value

while True:
    cur_state = btn.value
    if cur_state != prev_state:
        if not cur_state:
            print("BTN is down")
        else:
            print("BTN is up")

    prev_state = cur_state
```

Now that this code no longer relies on `time.sleep()`, it can play nicely without interrupting LED animations, servos, and

other tasks that your Circuit Python device is handling. If we do it this way, we get a single action when the button is pressed, and another single action when the button is released.

This process of looking for for the change in button state is known as debouncing, and there is a CircuitPython library created to help make it easier called `adafruit_debouncer`. If you stick to using this library, you won't have to worry about the debouncing logic interfering with other tasks. This guide covers its usage: Python Debouncer Library for Buttons and Sensors (https://adafru.it/NKb)

All Together Now



This example brings together LEDs, button input, and a servo all operating at the same time without interfering with one another.

When you press the button it will switch to blinking the opposite pair of LEDs by swapping the dictionaries into and out of the `BLINK_LIST` using the `pop()` and `append()` functions.

```
"""
This example script shows the usage of servos, LEDs, and buttons all
used simultaneously without interrupting each other.
"""

import board
import digitalio
import time
import neopixel
import pulseio
from adafruit_motor import servo
from digitalio import DigitalInOut, Direction, Pull

btn = DigitalInOut(board.SWITCH)
btn.direction = Direction.INPUT
btn.pull = Pull.UP

prev_state = btn.value

pixels = neopixel.NeoPixel(board.NEOPIXEL, 1)
pixels[0] = (0, 0, 0)

BLINK_LIST = [
    {
        "ON": 0.5,
        "OFF": 0.5,
        "PREV_TIME": -1,
        "PIN": board.D5,
    },
    {
        "ON": 0.5,
        "OFF": 0.5,
        "PREV_TIME": -1,
```

```
        "PIN": board.D6,
    },
    {
        "ON": 0.5,
        "OFF": 0.5,
        "PREV_TIME": -1,
        "PIN": board.D9,
    },
    {
        "ON": 0.5,
        "OFF": 0.5,
        "PREV_TIME": -1,
        "PIN": board.D10,
    }
]

SERVO_LIST = [
    {
        "MAX_ANGLE": 180,
        "MIN_ANGLE": 0,
        "PREV_TIME": -1,
        "PIN": board.A1,
        "DELAY_BETWEEN": 0.05,
        "SERVO": None,
        "MOVE_BY": 5
    },
    {
        "MAX_ANGLE": 90,
        "MIN_ANGLE": 0,
        "PREV_TIME": -1,
        "PIN": board.A3,
        "DELAY_BETWEEN": 0.02,
        "SERVO": None,
        "MOVE_BY": 2
    }
]

for cur_servo in SERVO_LIST:
    pwm = pulseio.PWMOut(cur_servo["PIN"], duty_cycle=2 ** 15, frequency=50)
    # Create a servo object.
    cur_servo["SERVO"] = servo.Servo(pwm)


for led in BLINK_LIST:
    led["PIN"] = digitalio.DigitalInOut(led["PIN"])
    led["PIN"].direction = digitalio.Direction.OUTPUT

disabled_leds = []
# temporarily remove first two from the blink list
disabled_leds.append(BLINK_LIST.pop(0))
disabled_leds.append(BLINK_LIST.pop(0))
while True:
    # Store the current time to refer to later.
    now = time.monotonic()

    cur_state = btn.value
    if cur_state != prev_state:
        if not cur_state:
            print("BTN is down")
```

```
            # swap the LED Blink patterns to the opposite pairs of LEDs
            temp = []
            temp.append(BLINK_LIST.pop(0))
            temp.append(BLINK_LIST.pop(0))

            BLINK_LIST.append(disabled_leds.pop(0))
            BLINK_LIST.append(disabled_leds.pop(0))

            disabled_leds.append(temp.pop(0))
            disabled_leds.append(temp.pop(0))

        else:
            print("BTN is up")

    prev_state = cur_state

    for led in BLINK_LIST:
        if led["PIN"].value == False:
            if now >= led["PREV_TIME"] + led["OFF"]:
                led["PREV_TIME"] = now
                led["PIN"].value = True
        if led["PIN"].value == True:
            if now >= led["PREV_TIME"] + led["ON"]:
                led["PREV_TIME"] = now
                led["PIN"].value = False

    for servo in SERVO_LIST:
        if now >= servo["PREV_TIME"] + servo["DELAY_BETWEEN"]:
            try:
                servo["SERVO"].angle += servo["MOVE_BY"]
            except ValueError as e:

                if servo["MOVE_BY"] > 0:
                    servo["SERVO"].angle = servo["MAX_ANGLE"]
                else:
                    servo["SERVO"].angle = servo["MIN_ANGLE"]

            if servo["SERVO"].angle >= servo["MAX_ANGLE"] or \
                servo["SERVO"].angle <= servo["MIN_ANGLE"]:

                    servo["MOVE_BY"] = -servo["MOVE_BY"]

            servo["PREV_TIME"] = now
```