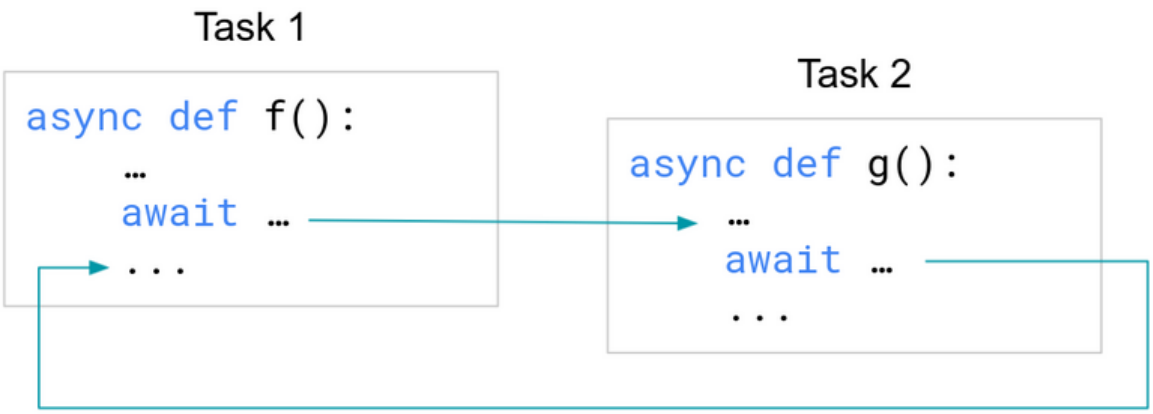
Cooperative Multitasking in CircuitPython with asyncio

Created by Dan Halbert



https://learn.adafruit.com/cooperative-multitasking-in-circuitpython-with-asyncio

Last updated on 2024-06-03 03:30:34 PM EDT

Table of Contents

Overview

This guide describes how to do cooperative multitasking in CircuitPython, using the `asyncio` library and the `async` and `await` language keywords. The `asyncio` library is included with CPython, the host-computer version of Python. MicroPython also supplies a version of `asyncio`, and that version has been adapted for use in CircuitPython.

asyncio is not supported on SAMD21 and a few other small boards due to lack of firmware and RAM space.

FAQ

Hey, why aren't you supporting preemptive hardware interrupts (irq)?

We looked at how MicroPython supports hardware interrupts and decided that the restrictions that are imposed make it harder to use and more prone to error than providing a better and more complete `asyncio` experience. Preemptive interrupts can come in at any time, which is hard to control in an interpreted language. In MicroPython, memory cannot be allocated in an interrupt handler, and there's a lot of things that Python does that allocate memory. And since there's a garbage collector, it isn't like interrupt latency can be promised.

Instead, we think `asyncio` + a "background task" that tracks GPIO change/fall/rise can be used to capture interrupts for when we are ready to process them. We have two native modules, `countio` and `keypad`, that can track your pin state changes in the background.

Also, we really, really want to keep CircuitPython code a true subset of CPython code so that examples can run on boards like Feather M4's or CircuitPlaygrounds as well as Raspberry Pi and desktop Python computers.

Hey, why aren't you supporting threads?

Python developers have dabbled with threads and pretty quickly determined that they are not a good way to have multiple tasks that can co-operate with shared memory. We believe that `asyncio` is a better way to have concurrent tasks that can share the same memory space safely - and without having to learn about and debug concurrent processes, something that is so notoriously hard that CS students have to take a course on how to do safely (or at least, they should if they don't!).

The vast majority of microcontrollers that are supported have a single core, and those that have dual core, like Espressif chipsets, would probably benefit from pinning certain background tasks like WiFi handling to a separate core, rather than trying to balance processing manually.

Cooperative Multitasking

Cooperative multitasking is a style of programming in which multiple tasks take turns running. Each task runs until it needs to wait for something, or until it decides it has run for long enough and should let another task run.

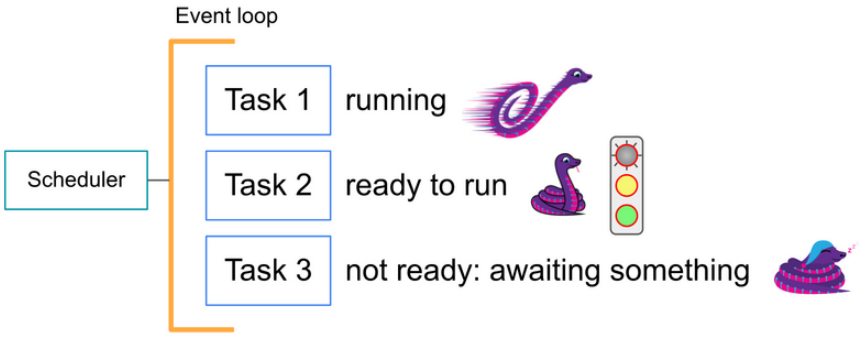
It's up to each task to decide when to yield control to other tasks, which is why it's cooperative. A task can freeze out other tasks, if it's not well behaved. This contrasts with preemptive multitasking, where tasks are interrupted without their knowledge to let other tasks run. Threads and processes are examples of preemptive multitasking.

Cooperative multitasking does not imply parallelism, where two tasks run literally simultaneously. The tasks do run concurrently. Their executions are interleaved: more than one can be active at a time.

In cooperative multitasking, a scheduler manages the tasks. Only one task runs at a time. When a task gives up control and starts waiting, the scheduler starts another task that is ready to run. The scheduler is fair and gives all tasks that are ready the chance to run. The scheduler runs an event loop which repeats this process over and over for all the tasks assigned to the event loop. You are already familiar with event loops, but you might not know the term. The `while True` loop that is the main part of nearly all CircuitPython programs often serves as an event loop, monitoring for button presses, or simply running some code periodically on a schedule. The `loop()` routine, a required part of every Arduino program, is also meant to be used as an event loop.

A task can wait for the completion of a sleep period, a network or file I/O operation, a timeout. It can also wait for some external asynchronous event such as a pin changing state.

The diagram below shows the scheduler, running an event loop, with three tasks: Task 1 is running, Task 2 is ready to run, and is waiting for Task 1 to give up control, and Task 3 is waiting for something else, and isn't ready to run yet.



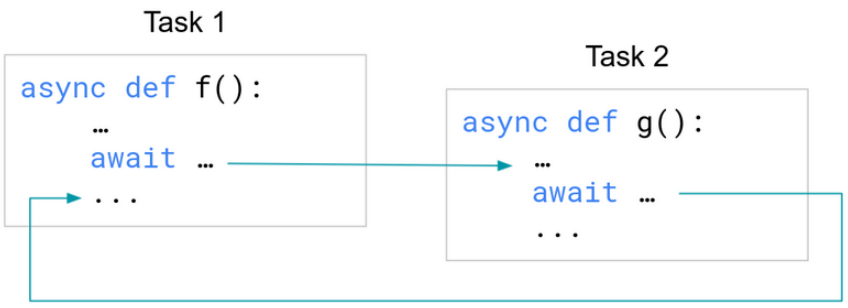
Coroutines

A task is a kind of coroutine. A coroutine can stop in the middle of some code and return back to its caller. When the coroutine is called again, it starts where it left off.

You may be familiar with generators in Python, which are also a kind of coroutine. You can recognize a generator because it includes one or more `yield` statements in it. When a coroutine gets to a `yield` statement, it returns to its caller, optionally passing back a value. When the coroutine is called again, it starts at the next statement after the `yield`. Early cooperative multitasking systems in Python took advantage of the generators mechanism, and used `yield` to indicate when a task was giving up control.

Later, Python added the `async` and `await` keywords specifically to support cooperative multitasking. A coroutine is declared with the keyword `async`, and the keyword `await` indicates that the coroutine is giving up control at that point.

The diagram below shows two coroutines, `f()` and `g()`, which are used as tasks. When Task 1 starts, it runs until it reaches an `await` statement where it needs wait for something. It gives up control at that point. The scheduler then looks for another task to run, and chooses Task 2. Task 2 runs until it reaches its own `await`. By that time, let's assume that whatever Task 1 was waiting for has happened. The scheduler sees that Task 1 is ready to run again, and starts up `f()` where it left off.



This guide will explain `async`, `await`, and their use with the `asyncio` library in more detail, in some simple examples, starting on the next page.

Installing the `asyncio` Library

Most of the examples in this guide require the CircuitPython version of the `asyncio` library. **The library is not built in to CircuitPython**; you need to copy it onto **CIRCUITPY** to use it. The `asyncio` library is available in the CircuitPython Library bundle (https://adafru.it/ENC), and is available on GitHub (https://adafru.it/X3c) as well. You can also use the circup tool (https://adafru.it/Tfi) to get the library and keep it up to date. If you find a problem with the library that you think is a bug, please file an issue (https://adafru.it/X3d).

The `asyncio` library uses the `adafruit_ticks` library internally. If you install `asyncio` by hand, install `adafruit_ticks` as well. The **circup** tool takes care of this for you automatically.

You may see mention of the `_asyncio` module, which is an internal helper module that is optionally used by the `asyncio` library, and is not meant for user by the end user. It is not a substitute for `asyncio`.

Not all CircuitPython boards provide async/await, because they do not have enough firmware space or RAM to use multitasking.

`async`/`await` is not Available on the Smallest Builds

`async` and `await` (and therefore `asyncio`) are available on most CircuitPython boards. SAMD21 boards, such as the Trinket M0, Metro M0, or Feather M0 boards do not have enough flash or RAM. A few nRF boards with internal flash only also do not have enough flash. RP2040, SAMD51, Espressif, and other ports are fine and do have `asyncio` capability.

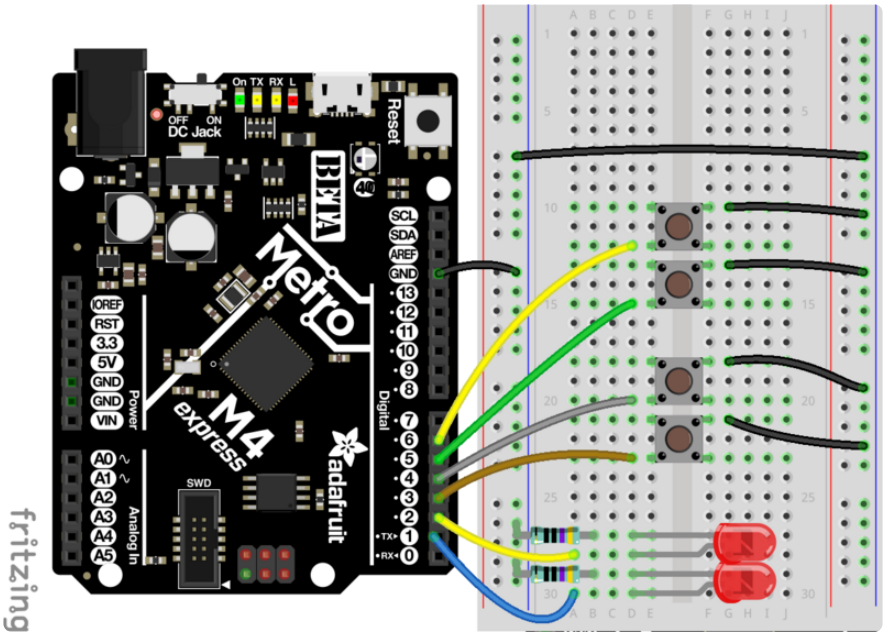
Future Enhancements

CircuitPython `asyncio` and related libraries and modules will be enhanced over time. For details, check out the issue list in Adafruit's asyncio library (https://adafru.it/X5f).

Concurrent Tasks

To demonstrate cooperative multitasking, the place to start is with simple examples of implementing one or two independently blinking LEDs. First there will be examples without asyncio, and then the guide will show how to use asyncio tasks to code the same examples.

You'll find it helpful to try these examples yourself, and experiment with modifying them a little. You just need to wire up a couple of LEDs with resistors to two pins. Later, you'll need some pushbuttons. Here's a typical wiring diagram for the Adafruit Metro M4 Express board, but you can use almost any board. See the Hardware (https://adafru.it/X3e) section on the previous page.



Blinking LEDs without asyncio

One LED

Suppose you want to make one LED blink, without using asyncio. The program below does that. It is fancier than the simplest blink example you may have seen, because it will be built upon as more examples are presented.

The example uses pin `board.D1`, because later examples use another pin for another LED. But use whichever pin you want, such as `board.LED` (if the board has such).

The program uses `with` for automatic `deinit()` (https://adafru.it/X3f) of `DigitalInOut`. It blinks the LED 10 times, and then it prints "**done**".

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time

import board
import digitalio


def blink(pin, interval, count):
    with digitalio.DigitalInOut(pin) as led:
        led.switch_to_output(value=False)
        for _ in range(count):
            led.value = True
            time.sleep(interval)
            led.value = False
            time.sleep(interval)


def main():
    blink(board.D1, 0.25, 10)
    print("done")


main()
```

Two LEDs

Now suppose you want to add another LED, blinking at a different rate, and blinking 20 times instead of 10. Both LEDs should start blinking at the same time, though they'll blink at different rates.

But if the program above is expanded in a thoughtless way, it won't work. Suppose just calling `blink()` a second time, after the first. The first `blink()` takes control and keeps control, even when it's sleeping with `time.sleep()`. The second `blink()`, for the second LED, only gets to run after the first `blink()` is all done. So this doesn't work.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# DOESN'T WORK
import time

import board
import digitalio


def blink(pin, interval, count):
    with digitalio.DigitalInOut(pin) as led:
        led.switch_to_output(value=False)
        for _ in range(count):
```

```
            led.value = True
            time.sleep(interval)
            led.value = False
            time.sleep(interval)


def main():
    blink(board.D1, 0.25, 10)
    # DOESN'T WORK
    # Second LED blinks only after the first one is finished.
    blink(board.D2, 0.1, 20)


main()
```

It turns out, it's quite a bit more complicated to get both LEDs to blink at the same time without using `asyncio`. The example below is just one way to solve the problem: there are many other ways. The program needs to check time constantly to see whether it's time for the LED to change state. A class was added to keep the necessary state. In essence, the program is using its own special-purpose task and event loop mechanism, just for this particular example.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time

import board
import digitalio


class Blinker:
    def __init__(self, led, interval, count):
        self.led = led
        self.interval = interval
        # Count both on and off.
        self.count2 = count * 2
        self.last_transition = 0

    def blink(self):
        """Return False when blinking is finished."""
        if self.count2 <= 0:
            return False
        now = time.monotonic()
        if now > self.last_transition + self.interval:
            self.led.value = not self.led.value
            self.last_transition = now
            self.count2 -= 1
        return True


def main():
    with digitalio.DigitalInOut(board.D1) as led1, digitalio.DigitalInOut(
        board.D2
    ) as led2:
        led1.switch_to_output(value=False)
        led2.switch_to_output(value=False)

        blinker1 = Blinker(led1, 0.25, 10)
        blinker2 = Blinker(led2, 0.1, 20)
        running1 = True
        running2 = True
        while running1 or running2:
```

```
            running1 = blinker1.blink()
            running2 = blinker2.blink()
        print("done")


main()
```

As you can see doing two tasks together, in harmony, is not nearly as straightforward as one might like. Now to see how asyncio can make this easier.

Blinking LEDs with asyncio

Now try the same examples like the ones above, but this time with asyncio.

One LED

Asyncio isn't needed to blink just one LED, but this example is written in asyncio style anyway. Notice that this example looks similar to the non-asyncio one-LED example above, but there are significant differences:

- The `time.sleep()` calls are replaced with `await asyncio.sleep()`.
- The `blink()` and `main()` functions are defined as `async def` instead of just `def`.
- A `Task` object is created (which starts it running) Then `await asyncio.gather()` will wait for the task to complete.
- Instead of just calling `main()`, call `asyncio.run(main())`.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import asyncio
import board
import digitalio


async def blink(pin, interval, count):  # Don't forget the async!
    with digitalio.DigitalInOut(pin) as led:
        led.switch_to_output(value=False)
        for _ in range(count):
            led.value = True
            await asyncio.sleep(interval)  # Don't forget the await!
            led.value = False
            await asyncio.sleep(interval)  # Don't forget the await!


async def main():  # Don't forget the async!
    led_task = asyncio.create_task(blink(board.D1, 0.25, 10))
    await asyncio.gather(led_task)  # Don't forget the await!
    print("done")
```

```
asyncio.run(main())
```

So what's going on here? First, every function or method that contains an `await` must be defined as `async def`, to indicate that it's a coroutine. Second, you can't call an `async` function directly from non-async code. Instead you must use `asyncio.run()` or a similar special function to bridge the gap between the non-async code (the mainline code in **code.py**) and async code.

What does `await` mean, anyway? It indicates a point in the code where the coroutine or task that is running gives up control to the scheduler, and waits for another async routine to complete. `await` means "I need to wait for something; let other tasks run until I'm ready to resume." In `blink()` above, it uses `await asyncio.sleep()`. When when the code goes to sleep, another task can be run. When the `sleep()` is over, this coroutine will resume.

In `main()`, we first create a `Task`. We instantiate the `blink()` coroutine by calling it with the arguments we want, and then we pass that coroutine to `asyncio.create_task()`. `create_task()` wraps the coroutine in a `Task`, and then schedules the task to run "soon". "Soon" means it will get a turn to run as soon other existing tasks have given up control.

Then the program uses `await asyncio.gather()`, which waits for all the tasks it's passed to finish. In this case, there's only one task to wait for.

Tasks and coroutines are both `Awaitable` objects, which means they can be `await`-ed. In fact, to get them to run, you have to await them.

Note that you have to use `await` to get a coroutine or a `Task` to do something. If you forget an `await`, nothing will happen, and you don't necessarily get an error immediately. Hence the cautionary comments above.

Two LEDs

The next example blinks two LEDs, again using asyncio. Note that this code is almost the same as the one-LED example above. The `blink()` function is exactly the same. This time two tasks in `main()` are created, one for each LED. `await asyncio.gather()` is used, but it is passed two tasks instead of one.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import asyncio
```

```
import board
import digitalio


async def blink(pin, interval, count):
    with digitalio.DigitalInOut(pin) as led:
        led.switch_to_output(value=False)
        for _ in range(count):
            led.value = True
            await asyncio.sleep(interval)  # Don't forget the "await"!
            led.value = False
            await asyncio.sleep(interval)  # Don't forget the "await"!


async def main():
    led1_task = asyncio.create_task(blink(board.D1, 0.25, 10))
    led2_task = asyncio.create_task(blink(board.D2, 0.1, 20))

    await asyncio.gather(led1_task, led2_task)  # Don't forget "await"!
    print("done")


asyncio.run(main())
```

Try this example, and see how the tasks appear to start at the same time. Both run to completion, and then you'll see "done" printed out.

Summary

Here are the key things to remember from these examples:

- Define a coroutine with `async def`.
- Give up control in a coroutine with `await`.
- Sleep in a coroutine with `await asyncio.sleep(interval)`.
- Create a task that will run soon with with `asyncio.create_task(some_coroutine(arg1, arg2, ...))`.
- Wait for tasks to finish with await `asyncio.gather(task1, task2, ...)`.
- Don't forget `await`.

Communicating Between Tasks

The LED blinking tasks discussed on the previous page (https://adafru.it/X3A) don't know about each other. In fact, that's almost the whole point: the tasks can run independently and still keep good time because they are using `asyncio.sleep()` to take turns running.

Control One Blinking LED

Now suppose you want to pass information to the LED tasks that affects what they do. For instance, suppose you want modify the blink rate, based on some button pushes.

You could monitor the buttons in the blink task, but that makes the blink task more complicated. This example makes a separate task that monitors the buttons. It will change a value in a shared object to tell the blink task what to do.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import asyncio
import board
import digitalio
import keypad


class Interval:
    """Simple class to hold an interval value. Use .value to to read or write."""

    def __init__(self, initial_interval):
        self.value = initial_interval


async def monitor_interval_buttons(pin_slower, pin_faster, interval):
    """Monitor two buttons: one lengthens the interval, the other shortens it.
    Change interval.value as appropriate.
    """
    # Assume buttons are active low.
    with keypad.Keys(
        (pin_slower, pin_faster), value_when_pressed=False, pull=True
    ) as keys:
        while True:
            key_event = keys.events.get()
            if key_event and key_event.pressed:
                if key_event.key_number == 0:
                    # Lengthen the interval.
                    interval.value += 0.1
                else:
                    # Shorten the interval.
                    interval.value = max(0.1, interval.value - 0.1)
                print("interval is now", interval.value)
            # Let another task run.
            await asyncio.sleep(0)


async def blink(pin, interval):
    """Blink the given pin forever.
    The blinking rate is controlled by the supplied Interval object.
    """
    with digitalio.DigitalInOut(pin) as led:
        led.switch_to_output()
        while True:
            led.value = not led.value
            await asyncio.sleep(interval.value)


async def main():
    # Start blinking 0.5 sec on, 0.5 sec off.
```

```
    interval = Interval(0.5)

    led_task = asyncio.create_task(blink(board.D1, interval))
    interval_task = asyncio.create_task(
        monitor_interval_buttons(board.D3, board.D4, interval)
    )
    # This will run forever, because neither task ever exits.
    await asyncio.gather(led_task, interval_task)


asyncio.run(main())
```

In the program above, the `led_task` and the `interval_task` share the `interval` object, but otherwise don't know about each other. You can change the details of one without the other one having to change.

There is one new interesting thing here. In `monitor_interval_buttons()`, it waits for keypresses in an infinite loop. Regardless of what happens, each time around the loop, it does an await `asyncio.sleep(0)`, which gives control back to the task scheduler. This is the standard way in asyncio of saying "I've run long enough, let other tasks run". If there's no other task ready to run, the scheduler will give back control immediately, since the sleep time is `0`.

Control Two Blinking LEDs

Now suppose you wanted to control two LEDs, with different buttons. That's easy: just create more tasks. Below is an example that does that. Only the `main()` has changed. The rest of the program stays the same.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import asyncio
import board
import digitalio
import keypad


class Interval:
    """Simple class to hold an interval value. Use .value to to read or write."""

    def __init__(self, initial_interval):
        self.value = initial_interval


async def monitor_interval_buttons(pin_slower, pin_faster, interval):
    """Monitor two buttons: one lengthens the interval, the other shortens it.
    Change interval.value as appropriate.
    """
    # Assume buttons are active low.
    with keypad.Keys(
        (pin_slower, pin_faster), value_when_pressed=False, pull=True
    ) as keys:
        while True:
            key_event = keys.events.get()
```

```
            if key_event and key_event.pressed:
                if key_event.key_number == 0:
                    # Lengthen the interval.
                    interval.value += 0.1
                else:
                    # Shorten the interval.
                    interval.value = max(0.1, interval.value - 0.1)
                print("interval is now", interval.value)
            # Let another task run.
            await asyncio.sleep(0)


async def blink(pin, interval):
    """Blink the given pin forever.
    The blinking rate is controlled by the supplied Interval object.
    """
    with digitalio.DigitalInOut(pin) as led:
        led.switch_to_output()
        while True:
            led.value = not led.value
            await asyncio.sleep(interval.value)


async def main():
    interval1 = Interval(0.5)
    interval2 = Interval(1.0)

    led1_task = asyncio.create_task(blink(board.D1, interval1))
    led2_task = asyncio.create_task(blink(board.D2, interval2))
    interval1_task = asyncio.create_task(
        monitor_interval_buttons(board.D3, board.D4, interval1)
    )
    interval2_task = asyncio.create_task(
        monitor_interval_buttons(board.D5, board.D6, interval2)
    )

    await asyncio.gather(led1_task, led2_task, interval1_task, interval2_task)


asyncio.run(main())
```

No Race Conditions

You might wonder if this technique of using shared data might run into race conditions, where data written by one task is in an inconsistent state when read by another task. As long as your task makes its data consistent before giving up control to the scheduler by using `await`, this won't happen. Since the tasks are cooperatively taking turns, one task cannot interrupt another to run.

To use jargon, the code between two `await` statements in a task is like a critical section: it can't be interrupted by another task.

(This is not true if you are using multiple event loops in multiple threads, but CircuitPython asyncio does not currently provide that.)

Controlling NeoPixels

Here's a more interesting example that uses tasks to control the direction and speed of a NeoPixel animation. This example has been tested on an Adafruit QT Py RP2040, with a 24-NeoPixel ring connected to `board.A0`. Three pushbuttons are connected to `board.A1`, `board.A2`, and `board.A3`; the other side of the buttons is grounded so that pressing the buttons brings the pins low.

Pressing button A1 will reverse the direction of the animation. Pressing A2 and A3 will slow down or speed up the animation by changing the delay between animation cycles.

As in the previous examples on the Communicating Between Tasks (https://adafru.it/XQE) page, a shared object is used to communicate between the task using the values and the tasks settings the values.

```
# SPDX-FileCopyrightText: 2022 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import asyncio

import board
import keypad
import neopixel
from rainbowio import colorwheel

pixel_pin = board.A0
num_pixels = 24

pixels = neopixel.NeoPixel(pixel_pin, num_pixels, brightness=0.03, auto_write=False)


class Controls:
    def __init__(self):
        self.reverse = False
        self.wait = 0.0


async def rainbow_cycle(controls):
    while True:
        # Increment by 2 instead of 1 to speed the cycle up a bit.
        for j in range(255, -1, -2) if controls.reverse else range(0, 256, 2):
            for i in range(num_pixels):
                rc_index = (i * 256 // num_pixels) + j
                pixels[i] = colorwheel(rc_index & 255)
            pixels.show()
            await asyncio.sleep(controls.wait)


async def monitor_buttons(reverse_pin, slower_pin, faster_pin, controls):
    """Monitor buttons that reverse direction and change animation speed.
    Assume buttons are active low.
    """
    with keypad.Keys(
        (reverse_pin, slower_pin, faster_pin), value_when_pressed=False, pull=True
    ) as keys:
```

```
        while True:
            key_event = keys.events.get()
            if key_event and key_event.pressed:
                key_number = key_event.key_number
                if key_number == 0:
                    controls.reverse = not controls.reverse
                elif key_number == 1:
                    # Lengthen the interval.
                    controls.wait = controls.wait + 0.001
                elif key_number == 2:
                    # Shorten the interval.
                    controls.wait = max(0.0, controls.wait - 0.001)
            # Let another task run.
            await asyncio.sleep(0)


async def main():
    controls = Controls()

    buttons_task = asyncio.create_task(
        monitor_buttons(board.A1, board.A2, board.A3, controls)
    )
    animation_task = asyncio.create_task(rainbow_cycle(controls))

    # This will run forever, because no tasks ever finish.
    await asyncio.gather(buttons_task, animation_task)


asyncio.run(main())
```

Handling Interrupts

The previous page, Communicating Between Tasks (https://adafru.it/X3B), showed how to handle button presses while blinking LEDs at the same time. A button press is an example of an asynchronous event, an event caused by something outside the running program, that can happen at any time.

Interrupts

Microcontrollers provide a hardware mechanism called interrupts for handling asynchronous events. A hardware interrupt can be generated when a pin changes state, when an internal timer triggers, when some hardware operation has completed, such as an I2C read or write, or for numerous other reasons. These events are usually asynchronous to the program being run, though sometimes interrupts are used to indicate that the program has caused an error, such as accessing a non-existent memory address.

When an interrupt occurs, the interrupt mechanism will call a routine called an interrupt handler. The currently running program is temporarily suspended and other interrupts of lower priority are blocked. The interrupt handler routine does something quickly and returns, and then the regular program (usually) resumes. An interrupt

handler is an example of preemptive multitasking, which was mentioned in the multitasking Overview (https://adafru.it/X3e) page.

For example, a pin connected to an external sensor may change, indicating that the sensor has new data. The interrupt handler itself could read the data and record it, but often the sensor would take too long to respond. So instead of reading the sensor data directly in the interrupt handler, the handler would set a flag to indicate that new data is available. Or, it may schedule a task to run later to read that data. Checking the flag or running the task occurs later, often inside an event loop.

The actual hardware interrupt is often called a hard interrupt, because it's generated and handled by the hardware. Acting on that interrupt later in an asynchronous fashion via software is often called handling a soft interrupt.

Polling

The alternative to interrupts is polling. When you check something over and over, waiting for a change, you are polling. For instance, you can monitor a `DigitalInOut.value` over and over in a loop. In the examples in this guide, you'll see a number of cases where some code checks for a condition, and then does an `asyncio.sleep()`. The code is polling, but in a controlled way, so that it doesn't block other code from running.

Handling Interrupts with `countio`

CircuitPython provides `countio`, a native module that counts rising-edge and/or falling-edge pin transitions. Internally, `countio` uses interrupts or other hardware mechanisms to catch these transitions and increment a count.

You can use `countio` with asyncio to catch interrupts and do something based on that interrupt. Here is a simple example using `countio` to monitor a pin connected to a push button, which will simulate a device interrupt. Note that the `countio` value is being polled in the task.

```
import asyncio
import board
import countio

async def catch_interrupt(pin):
    """Print a message when pin goes low."""
    with countio.Counter(pin) as interrupt:
        while True:
            if interrupt.count &gt; 0:
                interrupt.count = 0
                print("interrupted!")
            # Let another task run.
```

```
            await asyncio.sleep(0)


async def main():
    interrupt_task = asyncio.create_task(catch_interrupt(board.D3))
    await asyncio.gather(interrupt_task)

asyncio.run(main())
```

This program only has one task, so it's not that interesting. But you could use the techniques described on the Communicating Between Tasks (https://adafru.it/X3B) page in this guide to alert another task that the interrupt has happened.

The `countio` is good for catching pin transitions. But if you use it with mechanical switches, it will detect multiple counts due to switch bounce. Another good way is the `keypad` module, which does debouncing, and can handle multiple pins, switches, or buttons easily.

Handling Interrupts with `keypad`

The CircuitPython `keypad` module also provides a way of detecting pin transitions. It does not actually use hardware interrupts: instead it polls the pins every few milliseconds.

An example of using `keypad` was already presented in this guide on the Communicating Between Tasks (https://adafru.it/X3B) page. Here's another example, simplified to show just the transition detection.

```
import asyncio
import board
import keypad

async def catch_pin_transitions(pin):
    """Print a message when pin goes low and when it goes high."""
    with keypad.Keys((pin,), value_when_pressed=False) as keys:
        while True:
            event = keys.events.get()
            if event:
                if event.pressed:
                    print("pin went low")
                elif event.released:
                    print("pin went high")
            await asyncio.sleep(0)

async def main():
    interrupt_task = asyncio.create_task(catch_pin_transitions(board.D3))
    await asyncio.gather(interrupt_task)

asyncio.run(main())
```

Summary

- Monitor pin interrupts with asyncio by making a task that polls for asynchronous pin transitions.
- Detect asynchronous pin transitions with `countio` or `keypad.Keys`.