GitHub Actions Status Tower Light

Created by Kattni Rembor

https://learn.adafruit.com/github-actions-status-tower-light

Last updated on 2024-06-03 03:39:09 PM EDT

Table of Contents

Overview

GitHub is a web service that keeps track of code and files, designed to enable folks to collaborate on projects. Nearly all of Adafruit's coding projects and hardware files are hosted on GitHub (we have ~1600 repos!) (https://adafru.it/aYH). Since Adafruit publishes open source hardware and software, this works great to share the designs and also get feedback and improvements from the community. Community members can even find bugs or add new features, and submit those back to Adafruit so that everyone can benefit from the effort!

Many open source projects (or projects in general!) have standards and formatting styles to which all submitted code must adhere. The process of submitting bug fixes or new features to an existing GitHub repository often involves creating a pull request. Continuous Integration (CI) enables you to verify on GitHub that incoming code follows these standards. One form of CI is called GitHub Actions.

GitHub Actions is a Continuous Integration / Continuous Delivery (CI/CD) platform (https://adafru.it/-Ze) that, among many other things, allows you to automate testing and building incoming code. You create workflows that build and test every pull request. When a pull request is submitted, it automatically triggers an Actions run. Actions goes through all of the checks within the workflows. If there are errors, it will fail. If there are no errors, it will pass. Once passing, a maintainer will be able to merge your PR into their project.

As it is, to keep track of the Actions status of a PR, you would need to keep it open in a visible tab in your browser, and check on it periodically to find out where it's at. Not anymore!

You can easily set up a **GitHub Actions Status Tower Light** to place on your desk. **No soldering or wiring is required for this project** thanks to a handy USB-powered tower light we stock (http://adafru.it/5125) that works with any/all computers and operating

systems. Simply plug in the tower light, run the code on your computer, and you're ready to go!

- It lights up blinking yellow when the Actions run is queued
- It lights up solid yellow once Actions is running
- When it completes, it will beep.
- If it fails, it will light up red along with the beep.
- If it passes, it will light up green along with the beep.
- If it is cancelled, it will blink red along with the beep.

Of course, all these can be configured to your desire! This guide will show you how to set up the tower light, as well as walk you though the code needed to keep track of your Actions status.

There are a few prerequisites for this guide.

GitHub Requirements

The repository you intend to track must have some sort of GitHub Actions workflow enabled on it. The program used in this project uses the GitHub API to access workflow statuses. You can update which workflow file will trigger the light, but you must have at least one enabled.

Required Software

This guide assumes you have Python installed on your computer, including `pip` for installing the required libraries.

It is also assumed that you have a basic understanding of how to run Python programs from your computer.

Required Hardware

Tri-Color USB Controlled Tower Light with Buzzer

With this Tri-Color USB Controlled Tower Light with Buzzer, you can easily monitor and alert humanoids as to the status of a project, machine, or even if the bathroom...

https://www.adafruit.com/product/5125

Tower Light Set Up

There are two steps needed to get started with the tower light: installing the USB serial driver, and plugging in the light.

USB Serial Driver Installation

Inside the tower light is a microcontroller connected over a CH43x USB-to-UART chip. To access the tower light over USB serial, you will need to install the CH43x driver.

The driver is available for Mac and Windows. It is already built into Linux.

Download the driver below for your operating system and install it. If you would like more detail, check out the guide on installing this driver (https://adafru.it/-f8).

Click here to download the Windows driver

https://adafru.it/-f9

Click here to download the Mac driver

https://adafru.it/-ed

Plug in the Tower Light

Physical set up of the tower light is incredibly simple. The tower light is controlled over USB via a serial port, directly from your computer.

To get started, simply plug the USB cable attached to the tower light into a USB port on or attached to your computer.

That's it!

Code Requirements

There are a couple of things that you need to do before you can run the code for this project. This page covers how to complete the requirements.

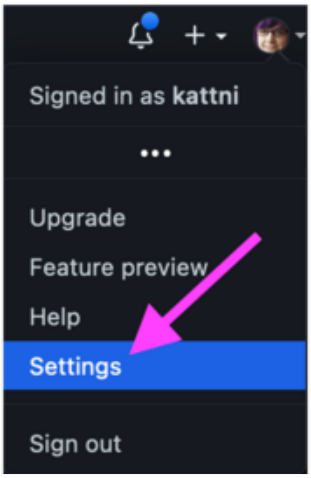
GitHub Personal Access Token

To access the Actions status through the GitHub API, you must have a GitHub Personal Access Token. This section will show you how to generate a new token, and how to make it available to your code.

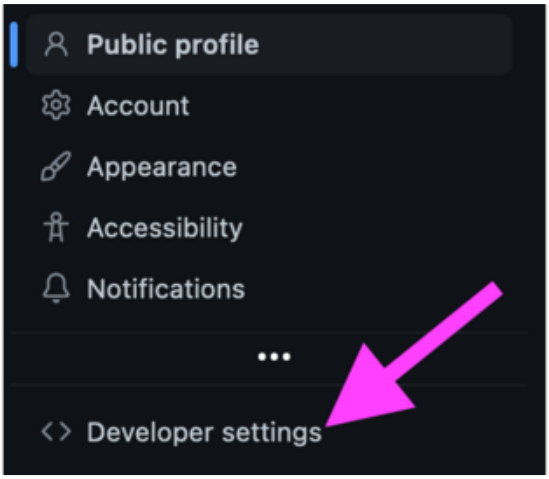
Generate Your Personal Access Token

Visit GitHub (https://adafru.it/d6C) while logged into your account. Then follow the instructions below.

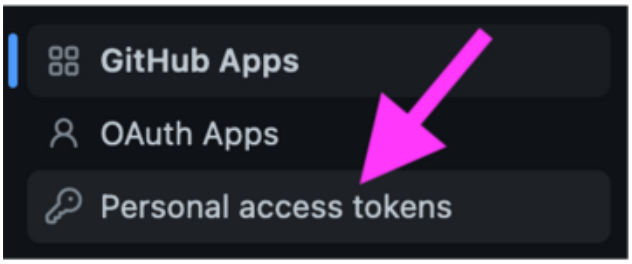
The "..." in the first two images represents the middle sections of rather lengthy menus. You will need to scroll past menu items not included in the images to get to the desired menu items.



In the upper right corner, you'll find a **small version of your avatar that is a menu**. Click the dropdown menu.
Towards the bottom of the menu, you'll find a **Settings** button. Click **Settings**.



The left sidebar begins with Public Profile selected. Scroll down the page to the bottom of the sidebar. The last item in the list is **Developer settings**. Click **Developer Settings**.



The sidebar in Developer Settings is quite short. Click the last item in the sidebar, **Personal access tokens**.

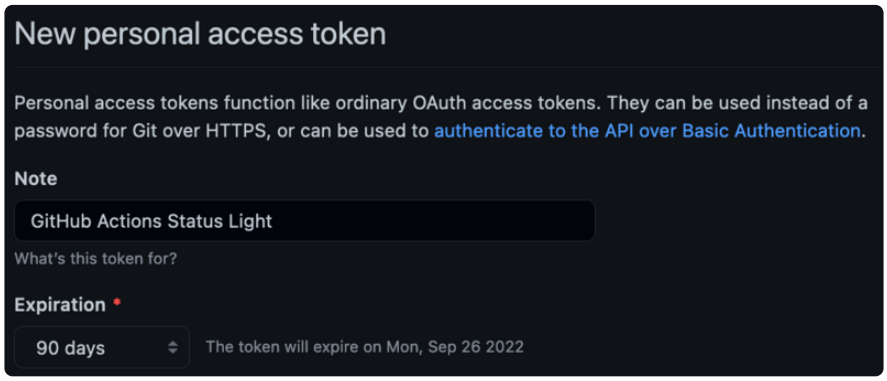
Once you're on the Personal access tokens page, click **Generate new token**.

You may be asked to verify your access to this account via password or, if enabled, two-factor authentication. Simply verify to continue.

The New Personal Access Token page contains a couple of items for you to update.

- **Add a Note** - Though not necessary, you can provide a short note as to what the token is for. This helps differentiate it from other tokens. In this case, **GitHub Actions Status Light** would be sufficient. Feel free to change it up to whatever works for you.
- **Verify the Expiration** - The token is set to expire in 30 days by default. Depending on your needs, you can increase or decrease the expiration by clicking the dropdown menu and choosing a different expiration option, or clicking **Custom...** and providing an expiration date.

While it is possible to set the token to never expire, it is not recommended. The safer option is to set an expiration date, and follow the steps to add a new personal access token to your project once it expires.

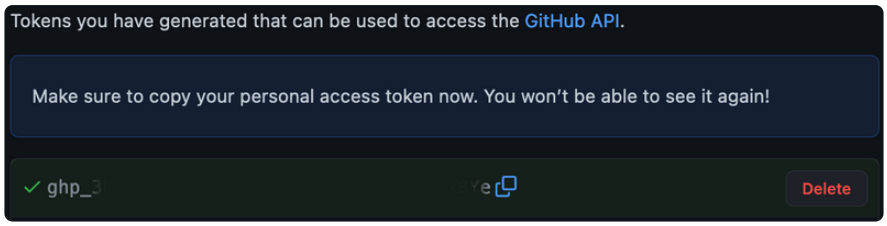


The next section allows you to set specific scopes for your token. In some cases, you will need to allow certain scopes for your token to work with your project. As this project is written, a personal access token with no scope provided will work fine. Skip all the checkboxes.

At the bottom of the page, click the green **Generate token** button.

GitHub will generate your personal access token, return you to the Personal Access Tokens page, and display your new token for you to copy.

Copy your new personal access token immediately. This is the only time you can view the token itself. When you visit this page again, the token will no longer be displayed.



Never share your personal access tokens with others! Personal access tokens should be kept private and local to your computer at all times.

Now that your token is generated and copied to your clipboard, it's time for the next step.

Create and Update Your Token Environment Variable

Follow the steps based on which operating system you're using.

Mac OS / Linux

In the directory from which you intend to run the code, create a file called **.env.sh**. Being in the same directory as the code is not a requirement. It does, however, make it quicker to `source` the file when you're ready to run your code, as you'll already be in the directory containing your code.

Open the **.env.sh** file in an editor of your choice (a text editor will be sufficient), and add the following line to the top of the file.

```
export GITHUB_API_TOKEN="ghp_yourtokenvaluehere"
```

Replace `ghp_yourtokenvaluehere` with your newly generated personal access token, retaining the quotation marks around it.

Save your changes to the file.

From a command line in a terminal window, run the following:

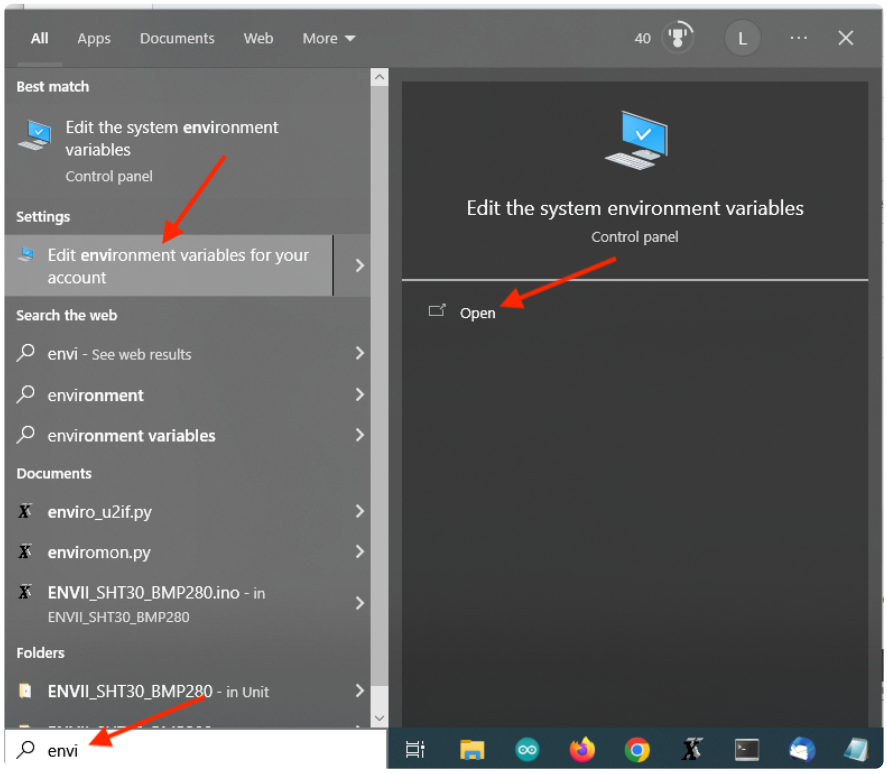
```
source .env.sh
```

```
10012 kattni@robocrepe:GitHub_Actions_Status_Tower_Light (towerlight) $ source .env.sh
10013 kattni@robocrepe:GitHub_Actions_Status_Tower_Light (towerlight) $
```

You've now made your personal access token available to the code.

Windows

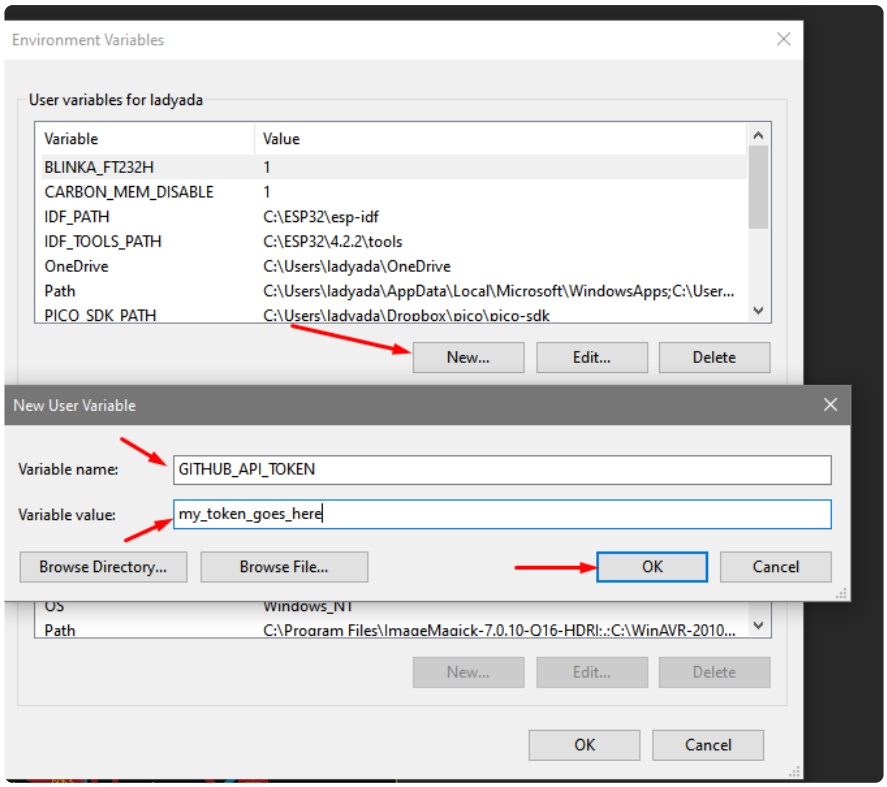
Begin typing **environment** into the Windows search. You'll find **Edit the system environment variables**. Click it in the search list, and click **Open** to continue.



Click the **New...** button to open the **New User Variable** window. Fill in the following:

- **Variable name:** = **GITHUB_API_TOKEN**
- **Variable value:** = **ghp_yourtokenvaluehere**

Then click **OK**.



You've now made your personal access token available to the code.

You're now ready to run the code and access the GitHub API successfully!

Code Configuration

You'll begin by copying the following code to a file on your computer called, for example, **actions_status.py**. (The filename does not really matter, but this page will assume the file is named **actions_status.py**.)

Click the **Copy code** link at the top left of the code element below, and paste it into the new file on your computer. For your convenience, this new file should be in the same directory as your **.env.sh** file, if you used that method to create your personal access token environment variable.

```
# SPDX-FileCopyrightText: 2022 Tim C for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
GitHub Actions Status Tower Light
"""
import json
import time
import os
import serial
import requests

# Customisations for this program
# Update with the URL from any repo running Actions to get the status. Defaults to
CircuitPython.
REPO_WORKFLOW_URL = "https://api.github.com/repos/adafruit/circuitpython/actions/
```

```
workflows/build.yml/runs"  # pylint: disable=line-too-long
# The rate at which the program will query GitHub for an updated status. You can
increase or
# decrease the delay time to fit the duration and frequency of Actions runs on your
repo.
# Defaults to 3 minutes.
POLL_DELAY = 60 * 3  # 3 minutes
# The length of time in seconds the LED remains on once the Actions run has
completed.
# Defaults to 30 seconds.
COMPLETION_LIGHT_TIME = 30  # seconds
# The length of time in seconds the buzzer beeps once the actions run has completed.
# Set this to 0 to disable the buzzer. Defaults to 1 second.
COMPLETION_BUZZER_TIME = 1  # seconds
# Determines whether the code sends commands to the tower light. Set it to False to
disable the
# tower light code and run this example without requiring the tower light. Defaults
to True.
ENABLE_USB_LIGHT_MESSAGES = True

# Serial port. Update the serial port to match the port of the tower light on your
computer.
# Windows will be a COM** port. If you are on Windows, comment out the Mac/Linux
line, and
# uncomment the line immediately below.
# serial_port = "COM57"
# Mac/Linux will be a /dev/** path to the serial port. If you're having trouble
finding it,
# check the contents of the /dev/ directory with the tower light unplugged and
plugged in.
serial_port = "/dev/tty.usbserial-144430"

# USB Tower Light constants
RED_ON = 0x11
RED_OFF = 0x21
RED_BLINK = 0x41

YELLOW_ON = 0x12
YELLOW_OFF = 0x22
YELLOW_BLINK = 0x42

GREEN_ON = 0x14
GREEN_OFF = 0x24
GREEN_BLINK = 0x44

BUZZER_ON = 0x18
BUZZER_OFF = 0x28
BUZZER_BLINK = 0x48

# Baud rate for serial communication
baud_rate = 9600


def send_command(serialport, cmd):
    serialport.write(bytes([cmd]))


def reset_state():
    # Clean up any old state
    send_command(mSerial, BUZZER_OFF)
    send_command(mSerial, RED_OFF)
    send_command(mSerial, YELLOW_OFF)
    send_command(mSerial, GREEN_OFF)


def buzzer_on_completion():
    if COMPLETION_BUZZER_TIME > 0:
        send_command(mSerial, BUZZER_ON)
        time.sleep(COMPLETION_BUZZER_TIME)
```

```
        send_command(mSerial, BUZZER_OFF)


already_shown_ids = []

headers = {'Accept': "application/vnd.github.v3+json",
           'Authorization': f"token {os.getenv('GITHUB_API_TOKEN')}"}


mSerial = None
if ENABLE_USB_LIGHT_MESSAGES:
    print("Opening serial port.")
    mSerial = serial.Serial(serial_port, baud_rate)

print("Starting Github Actions Status Watcher.")
print("Press Ctrl-C to Exit")
try:
    while True:
        print("Fetching workflow run status.")
        response = requests.get(f"{REPO_WORKFLOW_URL}?per_page=1", headers=headers)
        response_json = response.json()
        with open("action_status_result.json", "w") as f:
            f.write(json.dumps(response_json))

        workflow_run_id = response_json['workflow_runs'][0]['id']
        if workflow_run_id not in already_shown_ids:
            status = response_json['workflow_runs'][0]['status']
            conclusion = response_json['workflow_runs'][0]['conclusion']
            print(f"Status - Conclusion: {status} - {conclusion}")

            if status == "queued":
                print("Actions run status: Queued.")
                if ENABLE_USB_LIGHT_MESSAGES:
                    print("Sending serial command 'YELLOW_BLINK'.")
                    send_command(mSerial, YELLOW_BLINK)

            if status == "in_progress":
                print("Actions run status: In progress.")
                if ENABLE_USB_LIGHT_MESSAGES:
                    print("Sending serial command 'YELLOW_ON'.")
                    send_command(mSerial, YELLOW_ON)

            if status == "completed":
                print(f"Adding {workflow_run_id} to shown workflow IDs.")
                already_shown_ids.append(workflow_run_id)

                if conclusion == "success":
                    print("Actions run status: Completed - successful.")
                    if ENABLE_USB_LIGHT_MESSAGES:
                        send_command(mSerial, YELLOW_OFF)
                        print("Sending serial command 'GREEN_ON'.")
                        send_command(mSerial, GREEN_ON)
                        buzzer_on_completion()
                    time.sleep(COMPLETION_LIGHT_TIME - COMPLETION_BUZZER_TIME)
                    if ENABLE_USB_LIGHT_MESSAGES:
                        print("Sending serial command 'GREEN_OFF'.")
                        send_command(mSerial, GREEN_OFF)

                if conclusion == "failure":
                    print("Actions run status: Completed - failed.")
                    if ENABLE_USB_LIGHT_MESSAGES:
                        send_command(mSerial, YELLOW_OFF)
                        print("Sending serial command 'RED_ON'.")
                        send_command(mSerial, RED_ON)
                        buzzer_on_completion()
                    time.sleep(COMPLETION_LIGHT_TIME - COMPLETION_BUZZER_TIME)
                    if ENABLE_USB_LIGHT_MESSAGES:
                        print("Sending serial command 'RED_OFF'.")
                        send_command(mSerial, RED_OFF)
```

```
                if conclusion == "cancelled":
                    print("Actions run status: Completed - cancelled.")
                    if ENABLE_USB_LIGHT_MESSAGES:
                        send_command(mSerial, YELLOW_OFF)
                        print("Sending serial command 'RED_BLINK'.")
                        send_command(mSerial, RED_BLINK)
                        buzzer_on_completion()
                    time.sleep(COMPLETION_LIGHT_TIME - COMPLETION_BUZZER_TIME)
                    if ENABLE_USB_LIGHT_MESSAGES:
                        print("Sending serial command 'RED_OFF'.")
                        send_command(mSerial, RED_OFF)

            else:
                print("Already followed the current run.")
            time.sleep(POLL_DELAY)

    except KeyboardInterrupt:
        print("\nExiting Github Actions Status Watcher.")
        reset_state()
```

Code Configuration

Early in the code you'll find a series of options for you to configure. This section will walk through each one and provide details about its purpose, and how you can configure it to fit your needs.

Workflow URL

You can configure the repo workflow URL, which is saved to `REPO_WORKFLOW_URL` in the code. This is the URL to Actions runs for a specific workflow file on a specific repository. It defaults to the Adafruit CircuitPython repository **build.yml** file.

```
REPO_WORKFLOW_URL = "https://api.github.com/repos/adafruit/circuitpython/actions/
workflows/build.yml/runs"
```

There are three elements within the URL that you will likely want to update:

- User
- Repository
- Workflow file

Below is the same line of code with the configurable sections identified.

```
REPO_WORKFLOW_URL = "https://api.github.com/repos/{USER}/{REPOSITORY}/actions/
workflows/{WORKFLOW_FILE}/runs"
```

To follow Actions statuses on a repository on your own account, you want to replace `{USER}` with your GitHub user name, `{REPOSITORY}` with your local repository, and

`{WORKFLOW_FILE}` with the specific workflow file you'd like to use to trigger the status light.

Remember, you must have at least one GitHub Actions workflow enabled on the given repo for this project to work. Workflow files can be found within your repo in the **/.github/workflows/** directory.

API Query Interval Duration

You can choose the interval at which the program will query the GitHub API for updates. `POLL_DELAY` is a duration in seconds. As shown below, it defaults to `60 * 3` seconds, or three minutes.

```
POLL_DELAY = 60 * 3
```

You can increase or decrease this value to fit the typical duration and frequency of Actions runs on your chosen repository. For example, if you usually have short Actions runs that occur every few minutes, you might want to decrease this to 15 or 30 seconds to ensure you catch the statuses of the runs while they're active. If your Actions runs typically take an hour, and occur a few times per day, you may want to increase this to 20 minutes. You can tweak it to figure out what works best for you.

Status Light Duration

You can choose the duration the light will remain on after getting a **completed** status response from the GitHub API query. `COMPLETION_LIGHT_TIME` is a duration in seconds, which, as shown below, defaults to `30`.

```
COMPLETION_LIGHT_TIME = 30
```

When a completed status is returned from the API for the current Actions run, the light will respond by turning on red or green depending on the conclusion. The red or green light will stay on for the duration you choose here.

Increase or decrease this value to fit what works best for you.

Buzzer Sound Duration

You can choose the duration the buzzer will sound after getting a **completed** status response from the API query. `COMPLETION_BUZZER_TIME` is a duration in seconds, which, as shown below, defaults to `1`.

```
COMPLETION_BUZZER_TIME = 1
```

The buzzer is incredibly loud! Be prepared for this when running this code. You can set this variable to 0 to disable the buzzer completely.

When a completed status is returned from the API for the current Actions run, the buzzer will sound for both passing or failing conclusions, e.g. the buzzer sounds when the light turns on red or green. The duration of the buzzer is configurable by you.

You can set this variable to 0 to disable the buzzer completely, if you'd rather.

Enable USB Tower Light

This code is intended to be run with a USB Tower Light attached to your computer. However, it was written to be able to run without the hardware involved. The code defaults to expecting the hardware, with the following variable being set to `True`.

```
ENABLE_USB_LIGHT_MESSAGES = True
```

If you simply want to test whether your setup is working to enable you access to the GitHub Actions API, you can set `ENABLE_USB_LIGHT_MESSAGES = False` to disable sending the light commands in the code. Statuses will be printed to the serial console if this is disabled.

Serial Port

When you connect the tower light to your computer via USB (as long as you've installed the drivers!), it will show up on a serial port. You'll want to identify what that port is, and update `serial_port` to match.

Windows

On Windows, the serial port will be `COM**`, where `**` is a number. Click here (https://adafru.it/AAH) if you need help figuring out where to find the COM port. Once identified, you'll need to do two things in the code. First, **comment out** the Mac/Linux `serial_port =` line by adding a `#` to the beginning of the line. Second, **uncomment** the Windows `serial_port =` line, and update it to match the COM port from your tower light.

Your code would look similar to the following.

```
serial_port = "COM57"
# Mac/Linux will be a /dev/** path to the serial port. (...)
# serial_port = "/dev/tty.usbserial-144430"
```

MacOS/Linux

On MacOS and Linux, the serial port will be a **/dev/**** path. If you're struggling to find it, check the contents of **/dev/** with the tower light unplugged, and then check them again with the light plugged in. The new item in **/dev/** will most likely be the tower light serial port. Once Identified, update the serial_port = line to match the serial port from your tower light.

On MacOS, your code would look similar to the default code.

On Linux, your code may look similar to the following.

```
serial_port = "/dev/USBserial0"
```

That's it to configure! Now you're ready to run the code.

Code Usage

Before continuing, ensure that you completed the Code Requirements (https://adafru.it/-YE). If you haven't already saved the file, return to the beginning of the Configuration (https://adafru.it/-YF) page and follow the instructions there.

Open a terminal program on your computer, and navigate to the directory in which you saved **actions_status.py** during Code Configuration (https://adafru.it/-YF).

From the command line, run the program. When first running the program, your serial console should look something like the following.

```
10002 kattni@robocrepe:GitHub_Actions_Status_Tower_Light (towerlight) $ python actions_status.py
Opening serial port.
Starting Github Actions Status Watcher
Press Ctrl-C to Exit
Fetching workflow run status.
```

Workflow Run Statuses

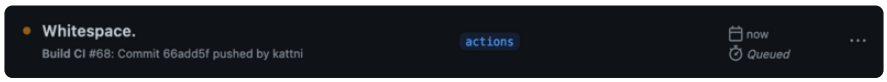
There are four possible responses when the workflow run status is fetched:

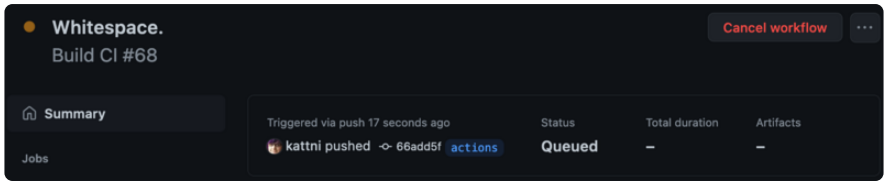
1. queued
2. in progress

3. completed - success
4. completed - failure
5. completed - cancelled

Each response triggers a different set of actions for the tower light.

Queued



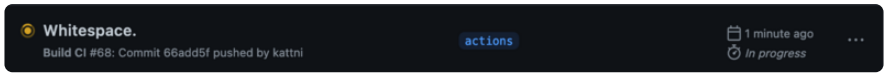


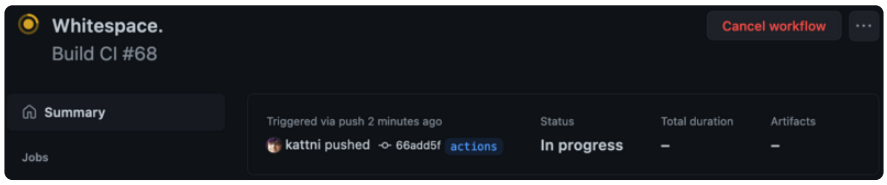
When the status is queued, it will print to the serial console.

```
Status - Conclusion: queued - None
Actions run status: Queued.
Sending serial command 'YELLOW_BLINK'.
```

The light will begin blinking yellow. This will continue as long as the current Actions run remained queued.

In Progress



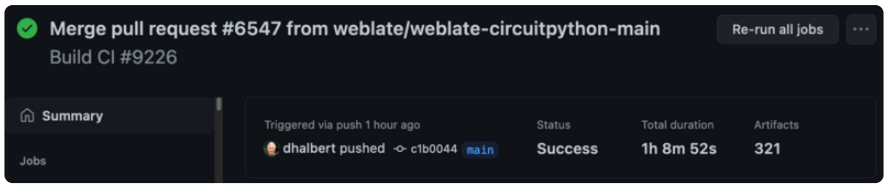


When the status is in progress, it will print to the serial console.

```
Status - Conclusion: in_progress - None
Actions run status: In progress.
Sending serial command 'YELLOW_ON'.
```

The light will turn on solid yellow. This will continue as long as the current Actions run remained in progress.

Completed - Success

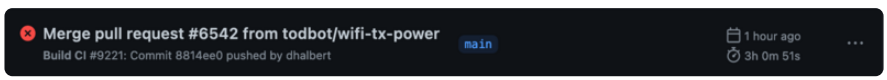


When the status is completed - success, the ID for that Actions run will be added to the already-shown workflow ID list.

```
Status - Conclusion: completed - success
Adding 2600100156 to shown workflow IDs.
Actions run status: Completed - successful.
Sending serial command 'GREEN_ON'.
```

The light will turn green and the buzzer will sound. The length of time for the light to remain on and the buzzer to sound was determined by you in configuration. After the chosen duration, each will turn off.

Completed - Failure

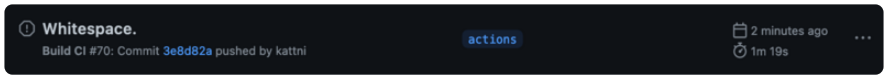


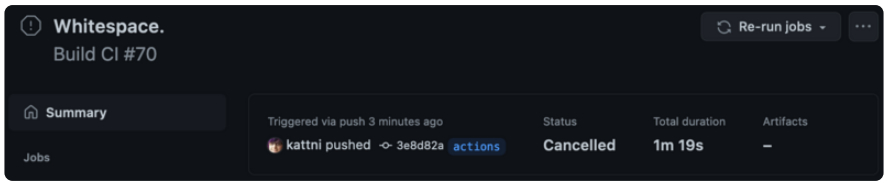
When the status is completed - failure, the ID for that Actions run will be added to the already-shown workflow ID list.

```
Status - Conclusion: completed - failure
Adding 2618212549 to shown workflow IDs.
Actions run status: Completed - failed.
Sending serial command 'RED_ON'.
Sending serial command 'RED_OFF'.
```

The light will turn red and the buzzer will sound. The length of time for the light to remain on and the buzzer to sound was determined by you in configuration. After the chosen duration, each will turn off.

Completed - Cancelled





When the status is completed - cancelled, the ID for that Actions run will be added to the already-shown workflow ID list.

```
Status - Conclusion: completed - cancelled
Adding 2618479273 to shown workflow IDs.
Actions run status: Completed - cancelled.
Sending serial command 'RED_BLINK'.
Sending serial command 'RED_OFF'.
```

The light will blink red and the buzzer will sound. The length of time for the light to remain on and the buzzer to sound was determined by you in configuration. After the chosen duration, each will turn off.

Already Followed

If the latest Actions run reached either completed status, and the code fetches again, it will simply print to the serial console, "Already followed the current run." This will continue until a new Actions run is triggered, at which point, it will return one of the four statuses above, and the light (and maybe the buzzer) will turn on as explained.

```
Fetching workflow run status.
Already followed the current run.
```

Exiting the Code

Press CTRL+C to exit the code. You will see the following printed to the serial console.

```
^C
Exiting Github Actions Status Watcher.
```

Code Walkthrough

You've gotten the code up and running. Excellent! However, you may be wondering exactly what's going on with the code. This page will walk you through the code, section by section, and explain what's happening.

Set Up Code

First, you'll go through the sections of code related to set up, found at the beginning of the file.

Imports and Configuration

The code begins with importing the necessary libraries and modules.

```
import json
import time
import serial
import requests
import os
```

Imports are followed by the configuration section explained in the previous page of this guide. You should have already configured this section to meet your needs and your choice of projects.

```
REPO_WORKFLOW_URL = "https://api.github.com/repos/adafruit/circuitpython/actions/
workflows/build.yml/runs"
POLL_DELAY = 60 * 3
COMPLETION_LIGHT_TIME = 30
COMPLETION_BUZZER_TIME = 1
ENABLE_USB_LIGHT_MESSAGES = True

# serial_port = "COM57"
serial_port = "/dev/tty.usbserial-144430"
```

Tower Light Constants and Baud Rate

The USB tower light is controlled over a serial connection from your computer. To turn on the lights and make sound with the buzzer, you need to send a command using a hex value constant. Assigning these constants to variables makes it much easier to identify which value does what, and makes the code significantly more readable. Therefore, the code includes a list of those constants and their variables.

```
RED_ON = 0x11
RED_OFF = 0x21
RED_BLINK = 0x41

YELLOW_ON = 0x12
YELLOW_OFF = 0x22
YELLOW_BLINK = 0x42

GREEN_ON = 0x14
GREEN_OFF = 0x24
GREEN_BLINK = 0x44

BUZZER_ON = 0x18
BUZZER_OFF = 0x28
BUZZER_BLINK = 0x48
```

Serial communication requires you to set a baud rate, which is the speed of the serial communication. This is done here.

```
baud_rate = 9600
```

Convenience Functions

There are three functions utilised in this example.

The first function enables you to send a command. It requires you to provide a serialport (defined later in this example) and a command.

```
def send_command(serialport, cmd):
    serialport.write(bytes([cmd]))
```

The second function turns off all of the lights and the buzzer, allowing you to reset the state of the tower light where needed.

```
def reset_state():
    # Clean up any old state
    send_command(mSerial, BUZZER_OFF)
    send_command(mSerial, RED_OFF)
    send_command(mSerial, YELLOW_OFF)
    send_command(mSerial, GREEN_OFF)
```

The third function includes code to turn the buzzer on for a period of time defined in the configuration section, if the buzzer is not disabled (e.g. `COMPLETION_BUZZER_TIME` set to 0 in configuration). This function is included simply to avoid duplicate code where the buzzer is used.

```
def buzzer_on_completion():
    if COMPLETION_BUZZER_TIME &gt; 0:
        send_command(mSerial, BUZZER_ON)
        time.sleep(COMPLETION_BUZZER_TIME)
        send_command(mSerial, BUZZER_OFF)
```

Workflow ID List

The code keeps track of completed workflows by adding the workflow ID number into a list, and querying that list so that it doesn't repeatedly notify you about completed workflows. You must create an empty list before you can add content to it.

```
already_shown_ids = []
```

HTTP Header for GitHub API

This code is required to be able to request data from the GitHub API. Your GitHub API Token should have been made available when you configured the environment variable. If you opted not to do this, you can paste it directly into the code, replacing `GITHUB_API_TOKEN` with your personal access token.

Including your personal access token directly in your code is not recommended. It is much safer to use environment variables.

```
headers = {'Accept': "application/vnd.github.v3+json",
           'Authorization': f"token {os.getenv('GITHUB_API_TOKEN')}"}
```

Serial Connection

As this code can be run without the hardware (if you set `ENABLE_USB_LIGHT_MESSAGES = False` in the configuration stage), you do not necessarily want to create the serial connection if you're not going to be using it. So, to ensure the code runs, the `mSerial` variable is initially created as `None`. Then, if the tower light communication is enabled, it initializes the serial connection as `mSerial` for use later.

```
mSerial = None
if ENABLE_USB_LIGHT_MESSAGES:
    print("Opening serial port.")
    mSerial = serial.Serial(serial_port, baud_rate)
```

Starting Status Watcher

Finally, immediately before the main loop, two lines are printed to the serial console, informing you of the status watcher starting, and how to exit the program when you desire. The code will run in an infinite loop until you interrupt it by pressing **CTRL+C** on your keyboard.

```
print("Starting Github Actions Status Watcher")
print("Press Ctrl-C to Exit")
```

Main Loop

This section walks through all the code found in the main loop in this example.

Immediately before the main loop begins, you'll find a `try`.

```
try:
    while True:
```

This is part of a `try`/`except` block surrounding the entire main loop. It will be explained in full at the end of this page.

Fetch Workflow Data

Before the fetch begins, "Fetching workflow run status." is printed to the serial console, to let you know what comes next.

Then, you perform a GET request to the GitHub API to pull the current Actions workflow data.

This data can be returned in multiple ways. This example will use JSON, so following the request for current data, we access the response as JSON data.

Next, you open a file named **actions_status_result.json** to which to write the results, write the results to the newly opened file, and then close the file to complete the file write.

```
[...]
        print("Fetching workflow run status.")
        response = requests.get(f"{REPO_WORKFLOW_URL}?per_page=1", headers=headers)
        response_json = response.json()
        f = open("action_status_result.json", "w")
        f.write(json.dumps(response_json))
        f.close()
```

Actions Workflow ID Lookup

The next step is to look up the ID number for the current Actions workflow from the JSON data, and save it to a variable. This is necessary to keep track of workflow IDs later in the code.

```
[...]
    workflow_run_id = response_json['workflow_runs'][0]['id']
```

Workflow Status and Conclusion

Later in the code, if an Actions run is completed, the ID is added to the `already_shown_ids` list that was created towards the beginning. This is where that list is checked, to determine whether the latest Actions workflow ID is in that list.

If the current workflow ID is not on the already shown list, the program continues on to the rest of the code nested beneath.

The code then looks up the current workflow's `status` and `conclusion`. The status can be **queued**, in **progress** or **completed**. The conclusion applies only to the completed status, and can be either **success** or **failure**. (Conclusion is `None` for queued and in progress.)

It then outputs the status and conclusion by printing to the serial console.

```
[...]
        if workflow_run_id not in already_shown_ids:
            status = response_json['workflow_runs'][0]['status']
            conclusion = response_json['workflow_runs'][0]['conclusion']
            print(f"Status - Conclusion: {status} - {conclusion}")
```

Status: Queued

If the status is queued, it prints the active status to the serial console. If the tower light is enabled, it prints the serial command being sent, and sends the serial command

`YELLOW_BLINK`. This makes the tower light blink yellow. The yellow blinking will continue as long as the workflow is queued.

```
[...]
            if status == "queued":
                print("Actions run status: Queued.")
                if ENABLE_USB_LIGHT_MESSAGES:
                    print("Sending serial command 'YELLOW_BLINK'.")
                    send_command(mSerial, YELLOW_BLINK)
```

Status: In Progress

If the status is in progress, it prints the active status to the serial console. If the tower light is enabled, it prints the serial command being sent, and sends the serial command `YELLOW`. This makes the light turn yellow. The yellow light will continue as long as the workflow is in progress.

```
[...]
            if status == "in_progress":
                print("Actions run status: In progress.")
                if ENABLE_USB_LIGHT_MESSAGES:
                    print("Sending serial command 'YELLOW_ON'.")
                    send_command(mSerial, YELLOW_ON)
```

Status: Completed

If the status is completed, the first thing that happens is printing to the serial console that the workflow ID is being added to the already shown IDs list, and the workflow ID number is added to the list.

```
[...]
            if status == "completed":
                print(f"Adding {workflow_run_id} to shown workflow IDs.")
                already_shown_ids.append(workflow_run_id)
```

Conclusion: Success

If the completed status conclusion is success, it prints this info to the serial console. If the tower light is enabled, it first sends the command `YELLOW_OFF`, in the event that the yellow light is still on. Next, it prints to the serial console the next command being sent, and sends the command `GREEN_ON`.

Following that, it runs the `buzzer_on_completion()` function to turn the buzzer on for the duration you previously configured.

Then the code pauses for the duration of the buzzer subtracted from the duration you configured the light to remain on. This ensures that the light will remain on as long as you configured it for, regardless of how long you turn on the buzzer.

Finally, when the completion light time duration is up, it prints to the serial console the command it's sending, and sends the command `GREEN_OFF`.

```
[...]
                if conclusion == "success":
                    print("Actions run status: Completed - successful.")
                    if ENABLE_USB_LIGHT_MESSAGES:
                        send_command(mSerial, YELLOW_OFF)
                        print("Sending serial command 'GREEN_ON'.")
                        send_command(mSerial, GREEN_ON)
                        buzzer_on_completion()
                    time.sleep(COMPLETION_LIGHT_TIME - COMPLETION_BUZZER_TIME)
                    if ENABLE_USB_LIGHT_MESSAGES:
                        print("Sending serial command 'GREEN_OFF'.")
                        send_command(mSerial, GREEN_OFF)
```

Conclusion: Failure

If the completed status conclusion is failure, it follows the same set of steps as the success conclusion. The differences are the info printed to the serial console, and the commands being sent to the light. The printed info indicates the failure conclusion. The commands sent to the light are `RED_ON`, and the configured amount of time later, `RED_OFF`.

```
[...]
                if conclusion == "failure":
                    print("Actions run status: Completed - failed.")
                    if ENABLE_USB_LIGHT_MESSAGES:
                        send_command(mSerial, YELLOW_OFF)
                        print("Sending serial command 'RED_ON'.")
                        send_command(mSerial, RED_ON)
                        buzzer_on_completion()
                    time.sleep(COMPLETION_LIGHT_TIME - COMPLETION_BUZZER_TIME)
                    if ENABLE_USB_LIGHT_MESSAGES:
                        print("Sending serial command 'RED_OFF'.")
                        send_command(mSerial, RED_OFF)
```

Conclusion: Cancelled

If the completed status conclusion is failure, it follows the same set of steps as the success and failure conclusions. The differences are the info printed to the serial console, and the commands being sent to the light. The printed info indicates the cancelled conclusion. The commands sent to the light are `RED_BLINK`, and the configured amount of time later, `RED_OFF`.

```
[...]
                if conclusion == "cancelled":
                    print("Actions run status: Completed - cancelled.")
                    if ENABLE_USB_LIGHT_MESSAGES:
                        send_command(mSerial, YELLOW_OFF)
                        print("Sending serial command 'RED_BLINK'.")
                        send_command(mSerial, RED_BLINK)
                        buzzer_on_completion()
                    time.sleep(COMPLETION_LIGHT_TIME - COMPLETION_BUZZER_TIME)
                    if ENABLE_USB_LIGHT_MESSAGES:
                        print("Sending serial command 'RED_OFF'.")
                        send_command(mSerial, RED_OFF)
```

Already Followed and Query Delay

The code ends with an `else` block that is matched up with the `if workflow_run_id not in already_shown_ids:` line from the Workflow Status and Conclusion section above. If the `workflow_run_id` is in the `already_shown_ids` list, then the code in the `else` block is run. It simply prints to the serial console, "Already followed the current run." to let you know that no new Actions run has occurred since the completion of the previous one.

Then, no matter what, the code sleeps for the `POLL_DELAY` duration. This is the length of time between queries to the GitHub API, configured in the Configuration section of this guide.

```
[...]
        else:
            print("Already followed the current run.")
        time.sleep(POLL_DELAY)
```

`try` / `except` Block

As mentioned at the beginning of the Main Loop section, the entire main loop is wrapped in a `try` / `except`. If you recall, you must press CTRL+C to exit the program. This causes a `KeyboardInterrupt`, which can be caught by an `except`, enabling you to run code when exiting the program.

This block tries to run the main loop, and if a `KeyboardInterrupt` is detected, it instead runs the code within the `except`. This code prints the exiting status to the serial console, and runs the `reset_state()` function to turn off all the possible tower light actions (i.e. the LEDs and the buzzer).

```
try:
[...]  # Main loop.
except KeyboardInterrupt:
```

```
    print("\nExiting Github Actions Status Watcher.")
    reset_state()
```

In this case, the problem was that without this block, if you exited the program while the tower light was doing something (e.g. the yellow light blinking, or green light and buzzer going off), the light would continue its current state after the program had exited. To terminate the continued tower light state, you would be required to unplug the light from your computer (to power it off) and plug it back in to be ready for the next time you run the code.

This solves that problem by using the `reset_state()` function to send the `*_OFF` commands for all possible light functionality. When you use CTRL+C to exit the program, the tower light will stop any current actions as well.

3D Printed Stand

You may have noticed that the tower light base doesn't have clearance for the USB cable to come out of the side of it, causing it to sit on an angle if placed on your desk or other flat surface. There are many ways to resolve this problem (including cardboard!). As a bonus, a 3D printed bolt-on stand is included here as an option.

Thank you to Rose Hooper (https://adafru.it/-Za) for designing this 3D printed stand!

The STL file can be downloaded by clicking the button below.

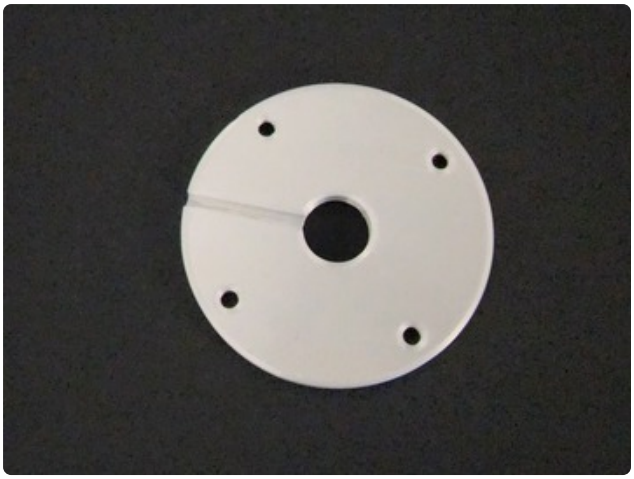
USB Tower Light Stand

https://adafru.it/-Zb

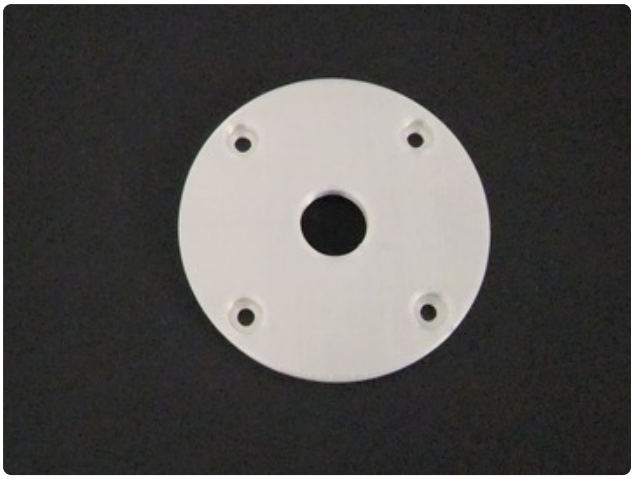
Suggested Hardware

- M4 x 8mm nuts x 4
- M4 bolts x 4

You can use any size fasteners that work for you. These are what were used in the assembled images below.



The top of the stand prints with a track for the USB cable to fit underneath the tower light base, and the tops of four holes through which to fit fasteners to attach it to the base of the light.



The bottom of the stand prints with four recessed holes to allow for fasteners to be attached without sticking out below the stand.

In this image, M4 x 8mm nuts were used, as they fit perfectly without sticking out the top of the M4 nuts. You can use any size that works for you.

Once assembled, the USB cable will sit in the stand track and no longer interfere with stability of the tower light!