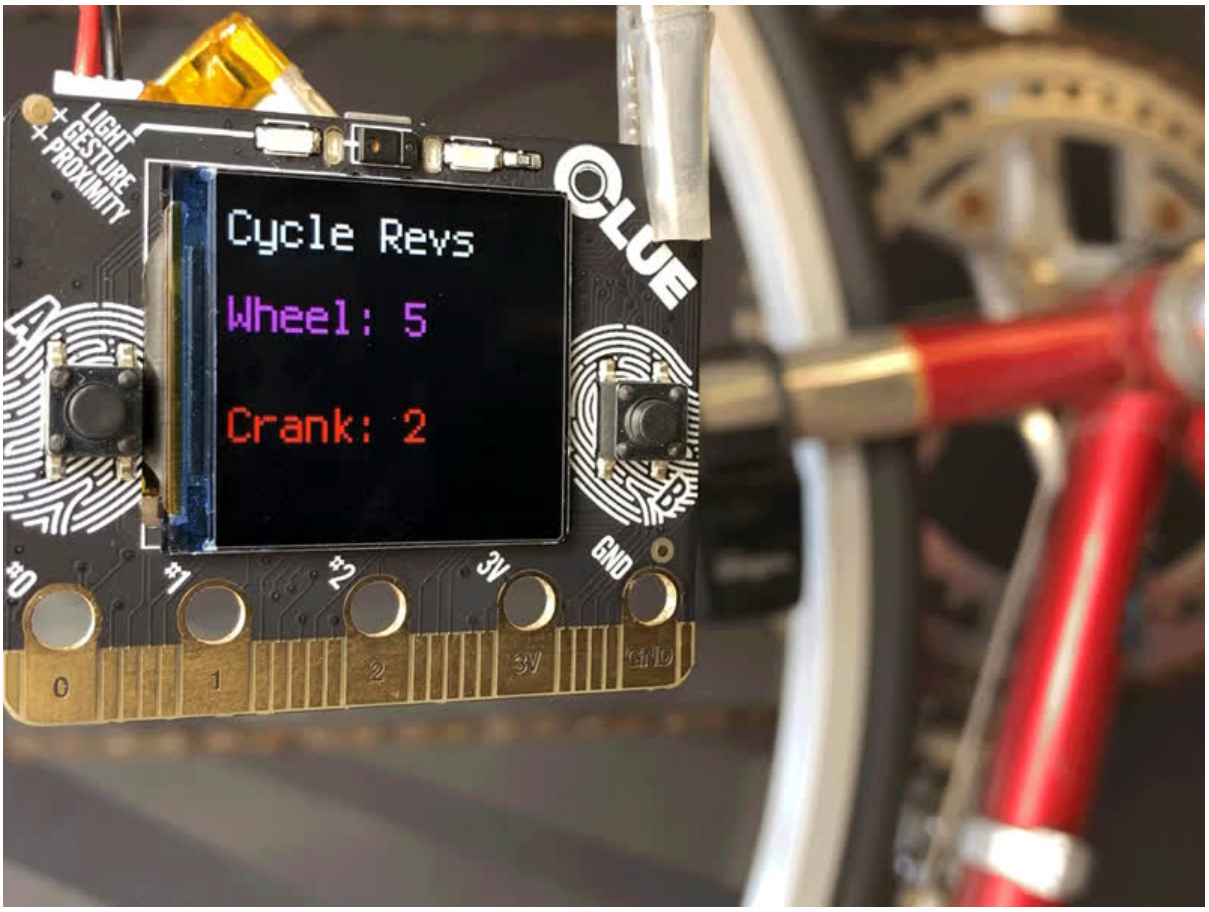
Bluetooth Cycling Speed & Cadence Sensor Display with Clue

Created by John Park



https://learn.adafruit.com/bluetooth-bicycle-speed-cadence-sensor-display-with-clue

Last updated on 2024-06-03 03:02:33 PM EDT

Table of Contents

Overview

Learn to read cycling speed and cadence sensor data over Bluetooth LE and display data on your CLUE board's screen!

Using our Bluefruit libraries and CircuitPython, it's now possible to connect to multiple devices, in the case of separate speed and cadence sensors, as well as dual sensor devices that present as a single BLE peripheral.

The **adafruit_clue** library is used to help keep things simple, so you can write the wheel and crank revolution text to the display without diving into complicated display code.

In a future project, we'll cover the math needed to convert these raw revolution values into speed and cadence like you'll see on a typical bike computer or cycling app. This project will get you familiar with reading and displaying data directly onto the CLUE.

Parts

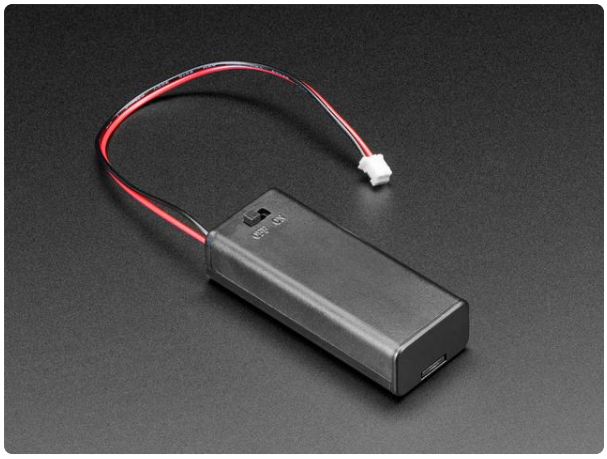
Cycling Speed & Cadence Sensor

Bluetooth LE compatible, such as the Wahoo Fitness Blue SC (https://adafru.it/IXA). This type of device is two sensors in one package and typically reads speed and cadence revolutions based on a pair of magnets affixed to a spoke and crank.

You can also use individual speed and/or cadence sensors that use an IMU rather than a magnet to sense revolutions, such as the Wahoo RPM Speed (https://adafru.it/IXB) and RPM Cadence (https://adafru.it/IXC).

Be sure that your sensors use Bluetooth LE and not Ant+ or some other radio standard. (Some use both, which is fine.)

[Adafruit CLUE - nRF52840 Express with Bluetooth LE]
Do you feel like you just don't have a CLUE? Well, we can help with that - get a CLUE here at Adafruit by picking up this sensor-packed development board. We wanted to build some...
https://www.adafruit.com/product/4500



[2 x AAA Battery Holder with On/Off Switch & JST PH Connector]
This battery holder connects 2 AAA batteries together in series for powering all kinds of projects. We spec'd these out because the box is nicely slim, and 2 AAA's add up to...
https://www.adafruit.com/product/4191

[Alkaline AAA batteries - 2 pack]
Battery power for your portable project! These batteries are good quality at a good price, and work fantastic with any of the kits or projects in the shop that use AAA's. This...
https://www.adafruit.com/product/617

[USB cable - USB A to Micro-B]
This here is your standard A to micro-B USB cable, for USB 1.1 or 2.0. Perfect for connecting a PC to your Metro, Feather, Raspberry Pi or other dev-board or...
https://www.adafruit.com/product/592

Understanding BLE

BLE Basics

To understand how we communicate between the MagicLight Bulb and the Circuit Playground Bluefruit (CPB), it's first important to get an overview of how Bluetooth Low Energy (BLE) works in general.

The nRF52840 chip on the CPB uses Bluetooth Low Energy, or BLE. BLE is a wireless communication protocol used by many devices, including mobile devices. You can communicate between your CPB and peripherals such as the Magic Light, mobile devices, and even other CPB boards!

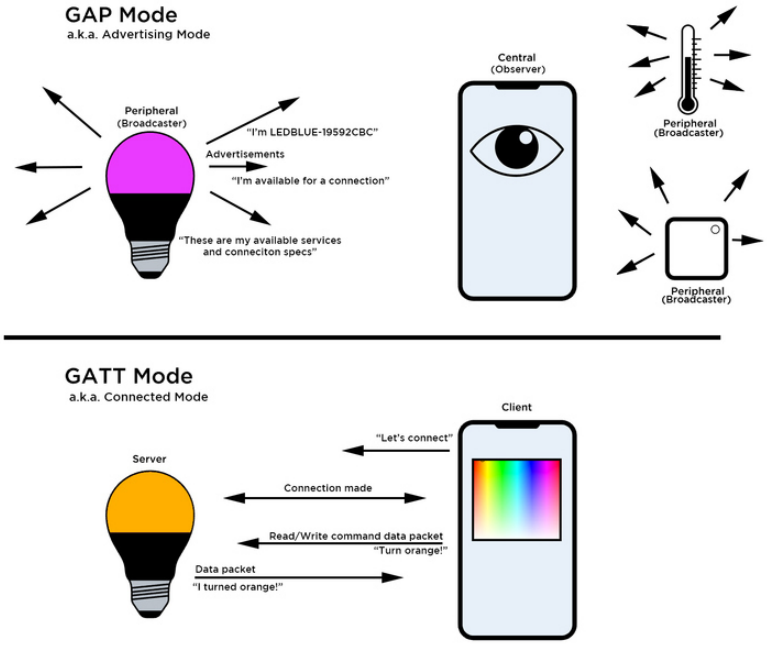
There are a few terms and concepts commonly used in BLE with which you may want to familiarize yourself. This will help you understand what your code is doing when you're using CircuitPython and BLE.

Two major concepts to know about are the two modes of BLE devices:

- Broadcasting mode (also called **GAP** for **G**eneric **A**ccess **P**rofile)
- Connected device mode (also called **GATT** for **G**eneric **ATT**ribute Profile).

GAP mode deals with broadcasting peripheral advertisements, such as "I'm a device named LEDBlue-19592CBC", as well as advertising information necessary to establish a dedicated device connection if desired. The peripheral may also be advertising available services.

GATT mode deals with communications and attribute transfer between two devices once they are connected, such as between a heart monitor and a phone, or between your CPB and the Magic Light.



Bluetooth LE Terms

GAP Mode

Device Roles:

- **Peripheral** - The low-power device that broadcasts advertisements. Examples of peripherals include: heart rate monitor, smart watch, fitness tracker, iBeacon, and the Magic Light. The CPB can also work as a peripheral.
- **Central** - The host "computer" that observes advertisements being broadcast by the Peripherals. This is often a mobile device such as a phone, tablet, desktop or laptop, but the CPB can also act as a central (which it will in this project).

Terms:

- **Advertising** - Information sent by the peripheral before a dedicated connection has been established. **All** nearby Centrals can observe these advertisements. When a peripheral device advertises, it may be transmitting the name of the device, describing its capabilities, and/or some other piece of data. Central can look for advertising peripherals to connect to, and use that information to determine each peripheral's capabilities (or Services offered, more on that below).

GATT Mode

Device Roles:

- **Server -** In connected mode, a device may take on a new role as a **Server**, providing a Service available to clients. It can now send and receive data packets as requested by the Client device to which it now has a connection.
- **Client -** In connected mode, a device may also take on a new role as **Client** that can send requests to one or more of a Server's available Services to send and receive data packets.

NOTE: A device in GATT mode can take on the role of both Server and Client while connected to another device.

Terms:

- **Profile -** A pre-defined collection of **Services** that a BLE device can provide. For example, the Heart Rate Profile, or the Cycling Sensor (bike computer) Profile. These Profiles are defined by the Bluetooth Special Interest Group (SIG). For devices that don't fit into one of the pre-defined Profiles, the manufacturer creates their own Profile. For example, there is not a "Smart Bulb" profile, so the Magic Light manufacturer has created their own unique one.
- **Service** - A function the Server provides. For example, a heart rate monitor armband may have separate Services for **Device Information**, **Battery Service**, and **Heart Rate** itself. Each Service is comprised of collections of information called **Characteristics**. In the case of the Heart Rate Service, the two Characteristics are **Heart Rate Measurement** and **Body Sensor Location.** The peripheral advertises its services.
- **Characteristic -** A Characteristic is a container for the value, or attribute, of a piece of data along with any associated metadata, such as a human-readable name. A characteristic may be readable, writable, or both. For example, the Heart Rate Measurement Characteristic can be served up to the Client device and will report the heart rate measurement as a number, as well as the unit string "bpm" for beats-per-minute. The Magic Light Server has a Characteristic for the RGB value of the bulb which can be written to by the Central to change the color. Characteristics each have a Universal Unique Identifier (UUID) which is a 16-bit or 128-bit ID.
- **Packet** - Data transmitted by a device. BLE devices and host computers transmit and receive data in small bursts called packets.

This guide (https://adafru.it/iCS) is another good introduction to the concepts of BLE, including GAP, GATT, Profiles, Services, and Characteristics.

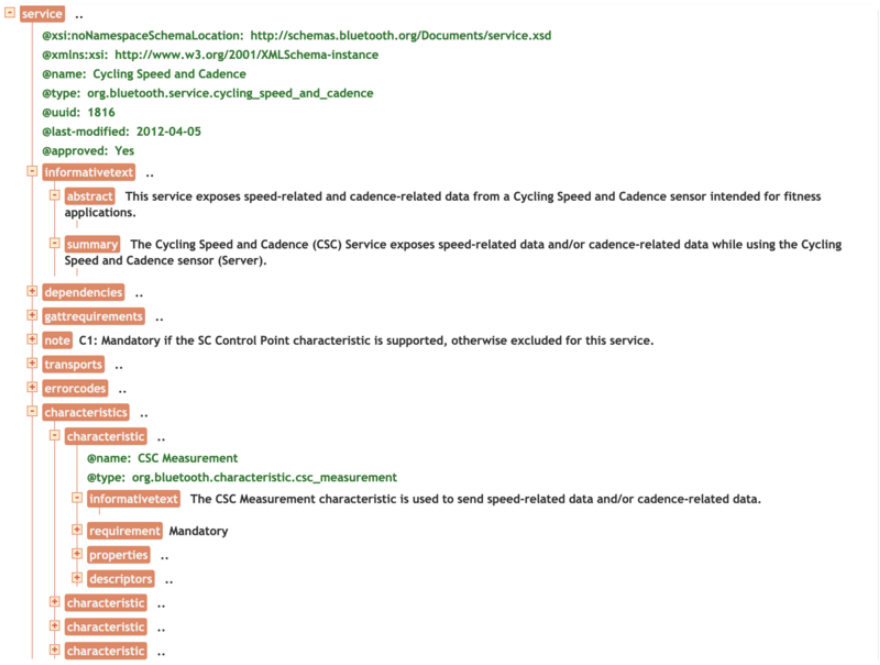
Cycling Speed and Cadence Service

The Bluetooth Special Interest Group has a standardized GATT (**G**eneritt **ATT**ribute Profile) for cycling speed and cadence sensors called the Cycling Speed and Cadence (CSC) profile. (You can see a list of all the GATT services here (https://adafru.it/IEF).)

This defines the commands and data that can be exchanged between bike sensor devices and the client device such as a phone, tablet, or BLE capable microcontroller (like we'll use in our project).

If you want to see how the Bluetooth SIG defines a GATT, such as the Cycling Speed and Cadence Profile (https://adafru.it/ITF), you can look at the official XML file here (https://adafru.it/IUa).

Even better, run that URL through a code beautifier, such as **codebeautify.org** for a more human-readable version.



Cycling Speed & Cadence Characteristics

The CSC service defines characteristics that can be served from the sensors to a connected device.

CSC Measurement

The most important for most needs is the CSC Measurement characteristic which serves up the following information:

- **Wheel revolutions** - used to calculate **speed** of a known wheel diameter
- **Last wheel event time**
- **Crank revolutions** -- used to indicate pedaling cadence
- **Last crank event time**

Not all Cycling sensor support all of the above characteristics. Some individual sensors will report just speed or just cadence.

Sensor Location

CSC sensors will also include a characteristic for the intended location of the cadence sensor, typically the right crank. This is built into the sensor firmware, not something that the device is determining on the fly!

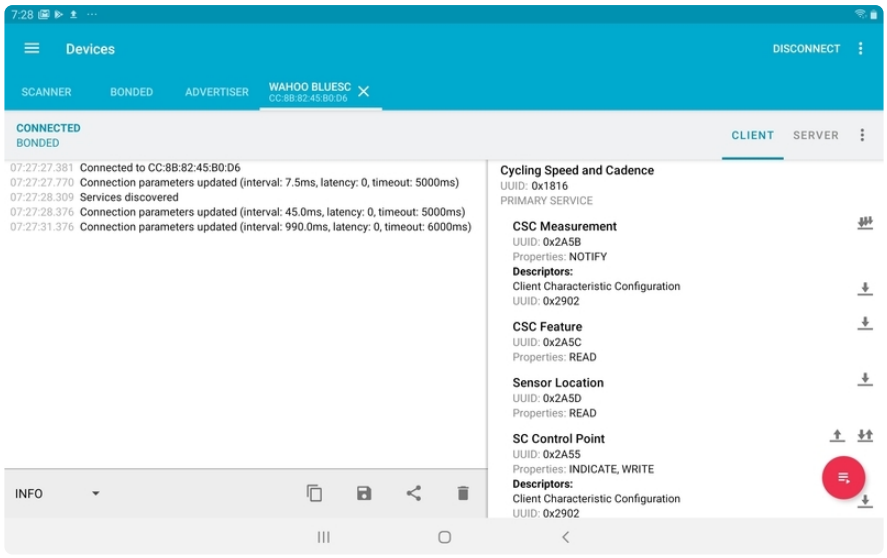
SC Control Point

The speed/cadence (SC) control point characteristic is used to allow the client device to write control points to the CSC in order to initiate sensor calibration.

nRF Connect View

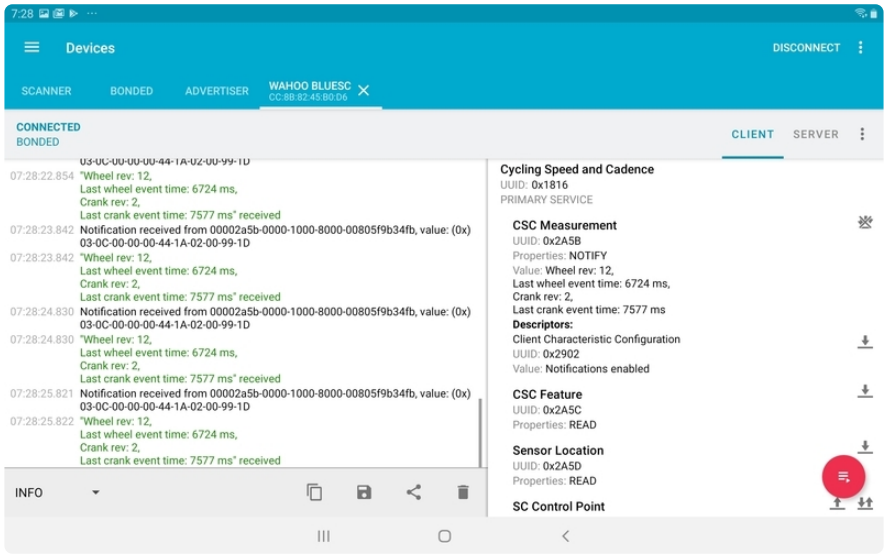
We can use the nRF Connect app from Nordic on iOS (https://adafru.it/IcD) and Android (https://adafru.it/eDw) to connect to a CSC sensor and look at the service, characteristics, and data.

Here we see data advertised including the device name, available services, connection parameters, and more.



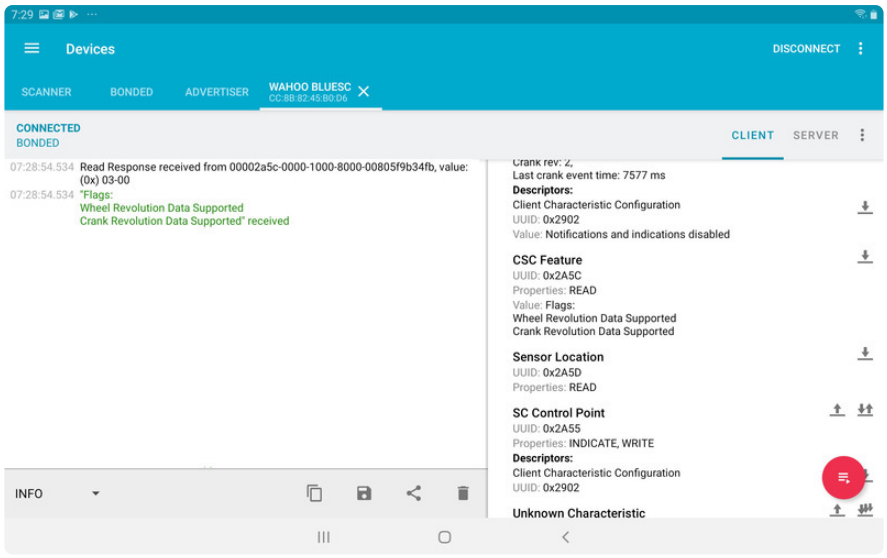
CSC Measurement

In this image, we can see the CSC device (WAHOO BLUESC) has been connected, and the Cycling Speed and Cadence characteristic is reporting its data.

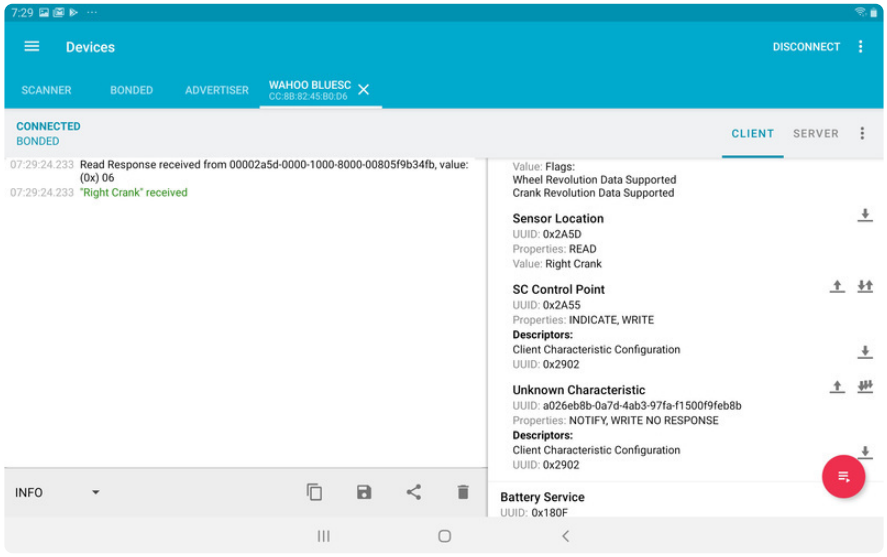


Supported Features

By requesting a read of the CSD Feature, we can see that both Wheel Revolution Data and Crank Revolution Data are supported.



If we request a read of the Sensor Location characteristic, we receive it as shown here, indicating "Right Crank":



CircuitPython on CLUE

CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** flash drive to iterate.

The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

Set up CircuitPython Quick Start!

Follow this quick step-by-step for super-fast Python power :)

Download the latest version of CircuitPython for CLUE from circuitpython.org

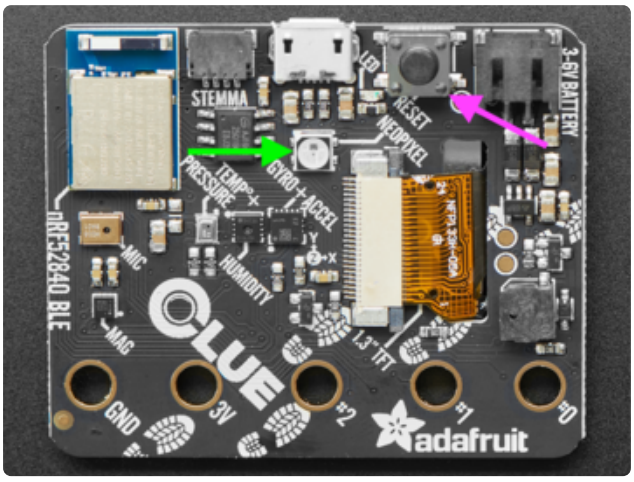
https://adafru.it/IHF

Click the link above to download the latest version of CircuitPython for the CLUE.

Download and save it to your desktop (or wherever is handy).

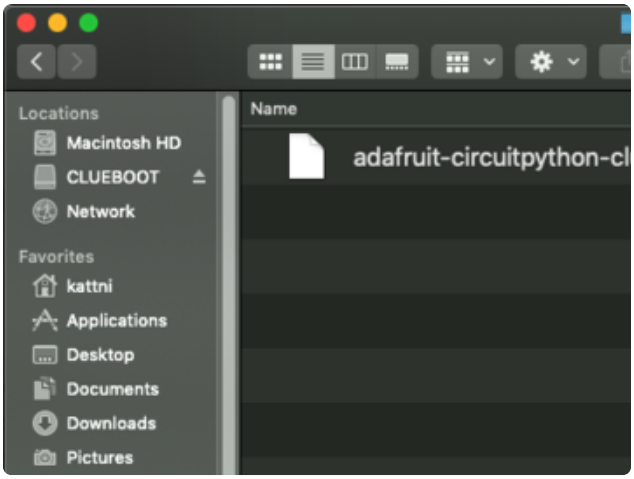
Plug your CLUE into your computer using a known-good USB cable.

A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

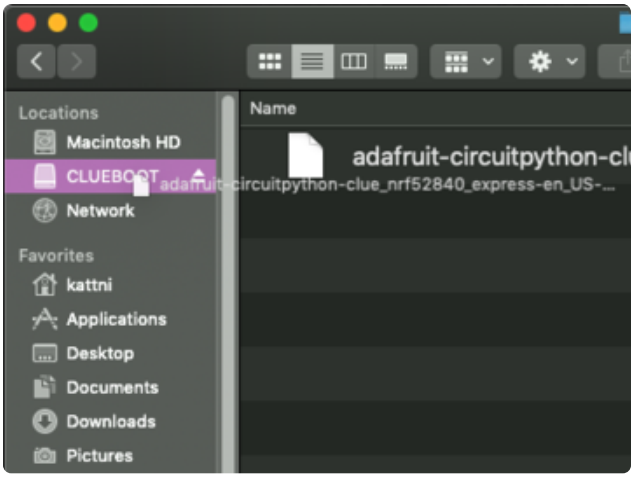


Double-click the **Reset** button on the top (magenta arrow) on your board, and you will see the NeoPixel RGB LED (green arrow) turn green. If it turns red, check the USB cable, try another USB port, etc. **Note:** The little red LED next to the USB connector will pulse red. That's ok!

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!

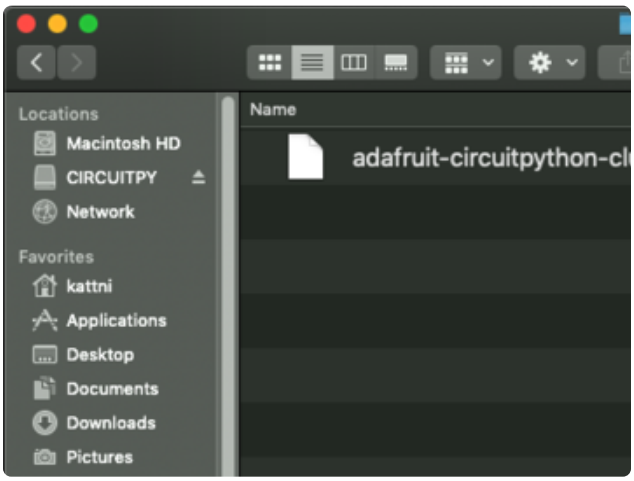


You will see a new disk drive appear called **CLUEBOOT**.



Drag the **adafruit-circuitpython-clue-etc.uf2** file to **CLUEBOOT.**

The LED will flash. Then, the **CLUEBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.



If this is the first time you're installing CircuitPython or you're doing a completely fresh install after erasing the filesystem, you will have two files - **boot_out.txt**, and **code.py**, and one folder - **lib** on your **CIRCUITPY** drive.

If CircuitPython was already installed, the files present before reloading CircuitPython should still be present on your **CIRCUITPY** drive. Loading CircuitPython will not create new files if there was already a CircuitPython filesystem present.

That's it, you're done! :)

Code the CLUE in CircuitPython

Text Editor

Adafruit recommends using the Mu editor for using your CircuitPython code with the Feather boards. You can get more info in this guide (https://adafru.it/ANO).

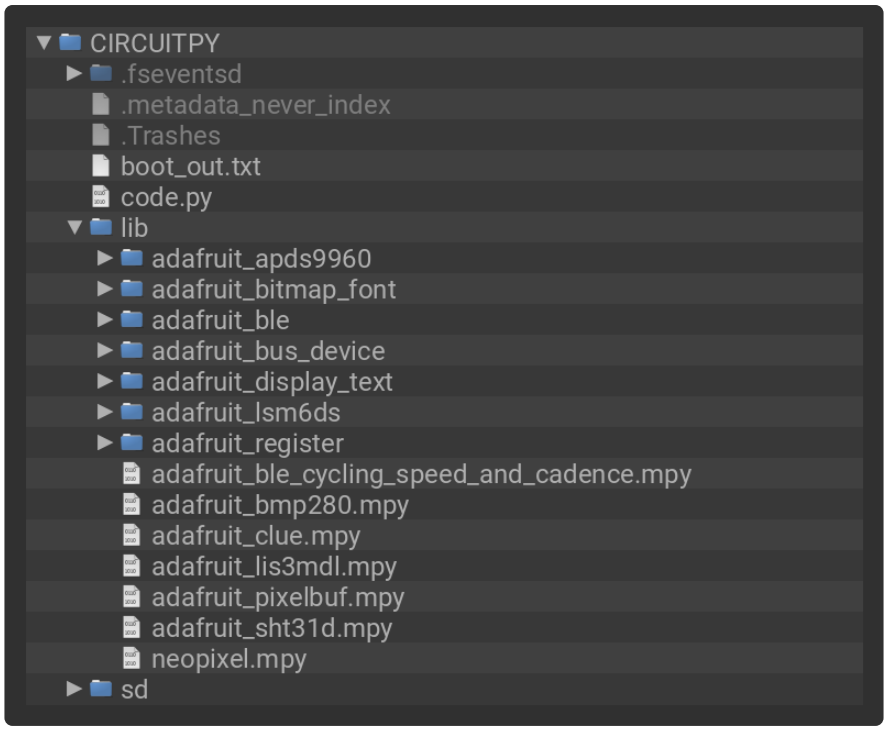
Alternatively, you can use any text editor that saves files.

Installing Project Code

To use with CircuitPython, you need to first install a few libraries, into the lib folder on your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CLUE_Cycling_Simple/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:



```
# SPDX-FileCopyrightText: 2020 John Park for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
Read cycling speed and cadence data from a peripheral using the standard BLE
Cycling Speed and Cadence (CSC) Service.
Works with single sensor (e.g., Wahoo Blue SC) or sensor pair, such as Wahoo RPM
"""

import time
from adafruit_clue import clue
import adafruit_ble
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services.standard.device_info import DeviceInfoService
from adafruit_ble_cycling_speed_and_cadence import CyclingSpeedAndCadenceService

clue_data = clue.simple_text_display(title="Cycle Revs", title_scale=1,
text_scale=3)

# PyLint can't find BLERadio for some reason so special case it here.
ble = adafruit_ble.BLERadio()    # pylint: disable=no-member


while True:
    print("Scanning...")
    # Save advertisements, indexed by address
    advs = {}
    for adv in ble.start_scan(ProvideServicesAdvertisement, timeout=5):
        if CyclingSpeedAndCadenceService in adv.services:
            print("found a CyclingSpeedAndCadenceService advertisement")
            # Save advertisement. Overwrite duplicates from same address (device).
            advs[adv.address] = adv

    ble.stop_scan()
    print("Stopped scanning")
    if not advs:
        # Nothing found. Go back and keep looking.
        continue

    # Connect to all available CSC sensors.
    cyc_connections = []
    for adv in advs.values():
        cyc_connections.append(ble.connect(adv))
        print("Connected", len(cyc_connections))

    # Print out info about each sensors.
    for conn in cyc_connections:
        if conn.connected:
            if DeviceInfoService in conn:
                dis = conn[DeviceInfoService]
                try:
                    manufacturer = dis.manufacturer
                except AttributeError:
                    manufacturer = "(Manufacturer Not specified)"
                print("Device:", manufacturer)
            else:
                print("No device information")

    print("Waiting for data... (could be 10-20 seconds or more)")
    # Get CSC Service from each sensor.
    cyc_services = []
    for conn in cyc_connections:
        cyc_services.append(conn[CyclingSpeedAndCadenceService])
    # Read data from each sensor once a second.
    # Stop if we lose connection to all sensors.

    while True:
```

```
        still_connected = False
        wheel_revs = None
        crank_revs = None
        for conn, svc in zip(cyc_connections, cyc_services):
            if conn.connected:
                still_connected = True
                values = svc.measurement_values
                if values is not None:  # add this
                    if values.cumulative_wheel_revolutions:
                        wheel_revs = values.cumulative_wheel_revolutions
                    if values.cumulative_crank_revolutions:
                        crank_revs = values.cumulative_crank_revolutions

        if not still_connected:
            break
        if wheel_revs:   # might still be None
            print(wheel_revs)
            clue_data[0].text = "Wheel: {0:d}".format(wheel_revs)
            clue_data.show()
        if crank_revs:
            print(crank_revs)
            clue_data[2].text = "Crank: {0:d}".format(crank_revs)
            clue_data.show()
        time.sleep(0.1)
```

Code Explainer

The code is doing a few fundamental things.

First, it loads the `time` and `board` libraries, as well as the necessary libraries to use **BLE** in general, and the `adafruit_ble_cycling_speed_and_cadence` library in specific.

We also load the `adafruit_clue` library so we can take advantage of convenient commands that simplify using the CLUE's display.

The `clue_data` variable is created to instantiate the CLUE display object for simple text and titles.

We also set up the `BLERadio` so it can be used to communicate with the sensor.

```
import time
from adafruit_clue import clue
import adafruit_ble
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services.standard.device_info import DeviceInfoService
from adafruit_ble_cycling_speed_and_cadence import CyclingSpeedAndCadenceService

clue_data = clue.simple_text_display(title="Cycle Revs", title_scale=1,
text_scale=3, num_lines=3)

# PyLint can't find BLERadio for some reason so special case it here.
ble = adafruit_ble.BLERadio()    # pylint: disable=no-member
```

Connection

Next, we scan for a BLE peripheral device advertising that it has the Cycling Speed & Cadence service.

When it is found, the CLUE will connect to it and then display the device name and other info. These are regular `print()` statements that show up in the REPL or other serial display, including the CLUE's display.

```
while True:
    print("Scanning...")
    # Save advertisements, indexed by address
    advs = {}
    for adv in ble.start_scan(ProvideServicesAdvertisement, timeout=5):
        if CyclingSpeedAndCadenceService in adv.services:
            print("found a CyclingSpeedAndCadenceService advertisement")
            # Save advertisement. Overwrite duplicates from same address (device).
            advs[adv.address] = adv

    ble.stop_scan()
    print("Stopped scanning")
    if not advs:
        # Nothing found. Go back and keep looking.
        continue

    # Connect to all available CSC sensors.
    cyc_connections = []
    for adv in advs.values():
        cyc_connections.append(ble.connect(adv))
        print("Connected", len(cyc_connections))

    # Print out info about each sensors.
    for conn in cyc_connections:
        if conn.connected:
            if DeviceInfoService in conn:
                dis = conn[DeviceInfoService]
                try:
                    manufacturer = dis.manufacturer
                except AttributeError:
                    manufacturer = "(Manufacturer Not specified)"
                print("Device:", manufacturer)
            else:
                print("No device information")

    print("Waiting for data... (could be 10-20 seconds or more)")
    # Get CSC Service from each sensor.
    cyc_services = []
    for conn in cyc_connections:
        cyc_services.append(conn[CyclingSpeedAndCadenceService])
    # Read data from each sensor once a second.
    # Stop if we lose connection to all sensors.
```

Receiving Revolution Values

Once connected, we create a pair of variables to store the wheel and crank revolution values as `wheel_revs` and `crank_revs`

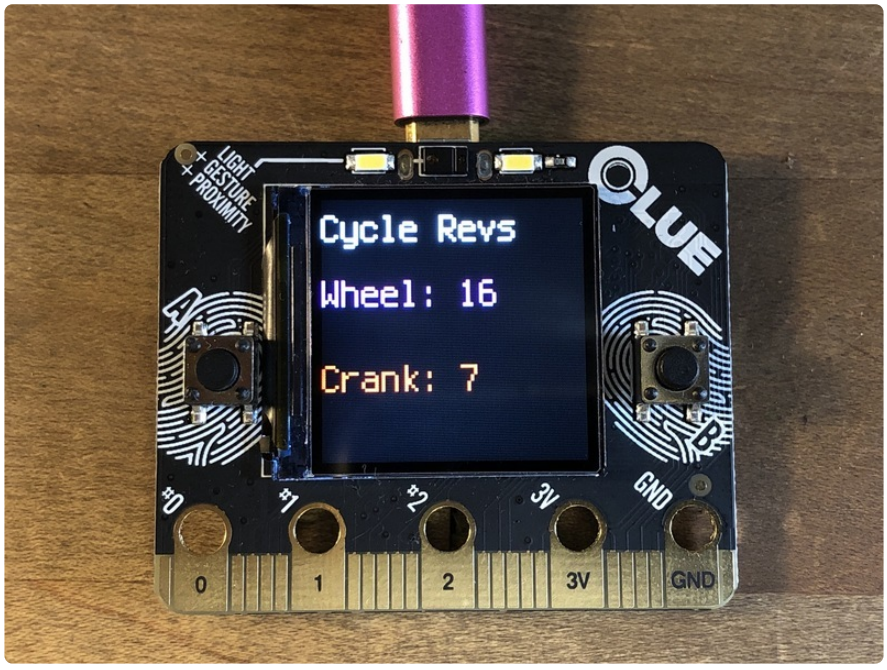
When we receive values for these two attributes they are displayed on the CLUE screen using the `clue_data[].text` command, and `clue_data.show()` to update the display.

```
while True:
        still_connected = False
        wheel_revs = None
        crank_revs = None
        for conn, svc in zip(cyc_connections, cyc_services):
            if conn.connected:
                still_connected = True
                values = svc.measurement_values
                if values is not None:  # add this
                    if values.cumulative_wheel_revolutions:
                        wheel_revs = values.cumulative_wheel_revolutions
                    if values.cumulative_crank_revolutions:
                        crank_revs = values.cumulative_crank_revolutions

        if not still_connected:
            break
        if wheel_revs:   # might still be None
            print(wheel_revs)
            clue_data[0].text = "Wheel: {0:d}".format(wheel_revs)
            clue_data.show()
        if crank_revs:
            print(crank_revs)
            clue_data[2].text = "Crank: {0:d}".format(crank_revs)
            clue_data.show()
        time.sleep(0.1)
```

Let's check it out in action!

Use the CLUE Cycling Display



Install Sensors

Follow the manufacturer's directions to install your speed and cadence sensors. Here you can see a single unit with a magnet attached to a rear wheel spoke for wheel revolution sensing, and a crank mounted magnet for cadence sensing.

The unit detects each magnet with a different Hall effect sensor and then sends revolution data over BLE.

Use the CLUE Display

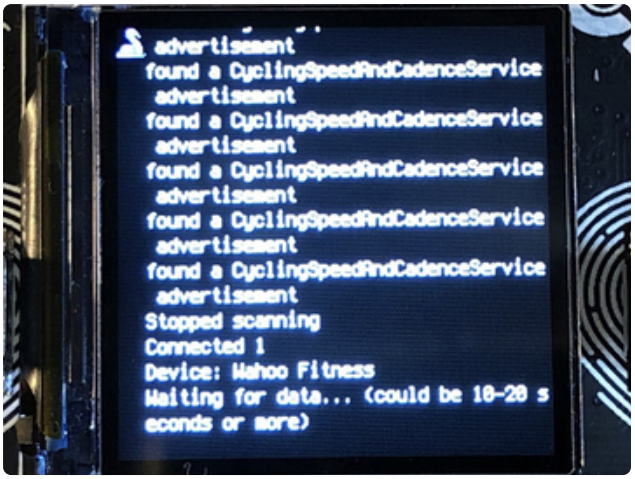
Plug in power to the CLUE, either with a battery pack or over USB.

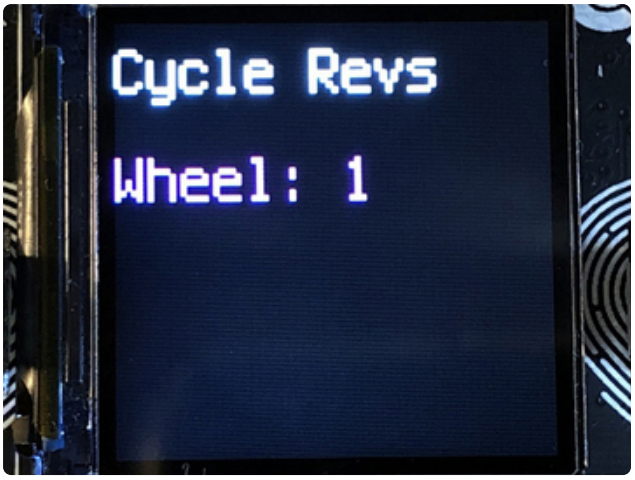
The CLUE will begin to search for the cycling speed & cadence sensor.

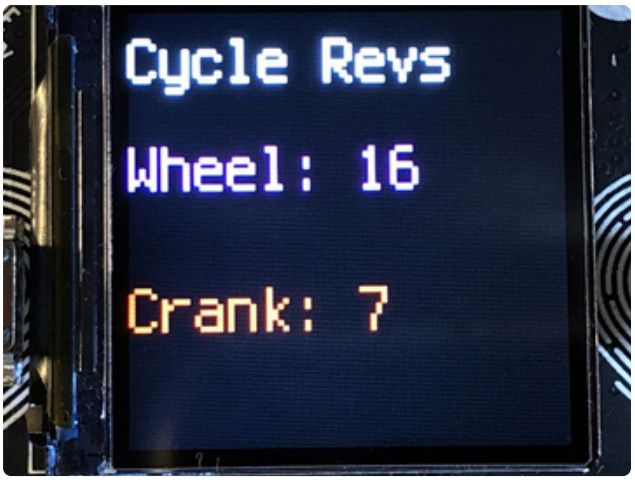
Turn the crank to wake up the sensor in case it has gone into power saving mode.

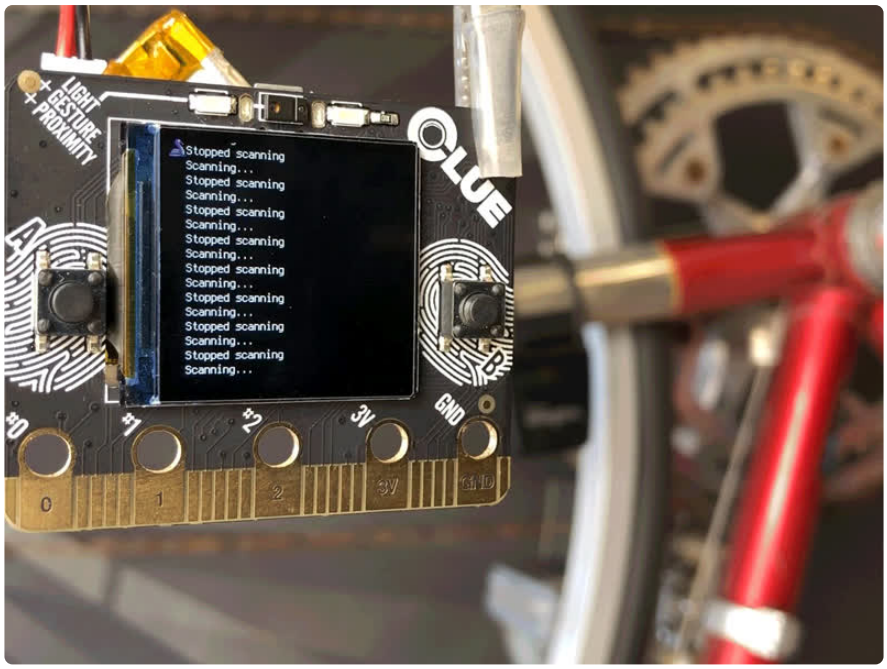
When the CLUE finds the CSC sensor, they will negotiate a connection.

Now, you can give the wheel a spin and crank the pedal. You'll see the cumulative revolutions of each displayed on the CLUE!


advertisement
found a CyclingSpeedAndCadenceService advertisement
found a CyclingSpeedAndCadenceService advertisement
found a CyclingSpeedAndCadenceService advertisement
found a CyclingSpeedAndCadenceService advertisement
found a CyclingSpeedAndCadenceService advertisement
Stopped scanning
Connected 1
Device: Wahoo Fitness
Waiting for data... (could be 10-20 seconds or more)


Cycle Revs
Wheel: 1


Cycle Revs
Wheel: 16
Crank: 7


LIGHT
+ GESTURE
+ PROXIMITY
CLUE
Stopped scanning
Scanning...
Stopped scanning
Scanning...
Stopped scanning
Scanning...
Stopped scanning
Scanning...
Stopped scanning
Scanning...
Stopped scanning
Scanning...
Stopped scanning
Scanning...
Stopped scanning
Scanning...
A
B
#0
#1
#2
3V
GND
0
1
2
3V