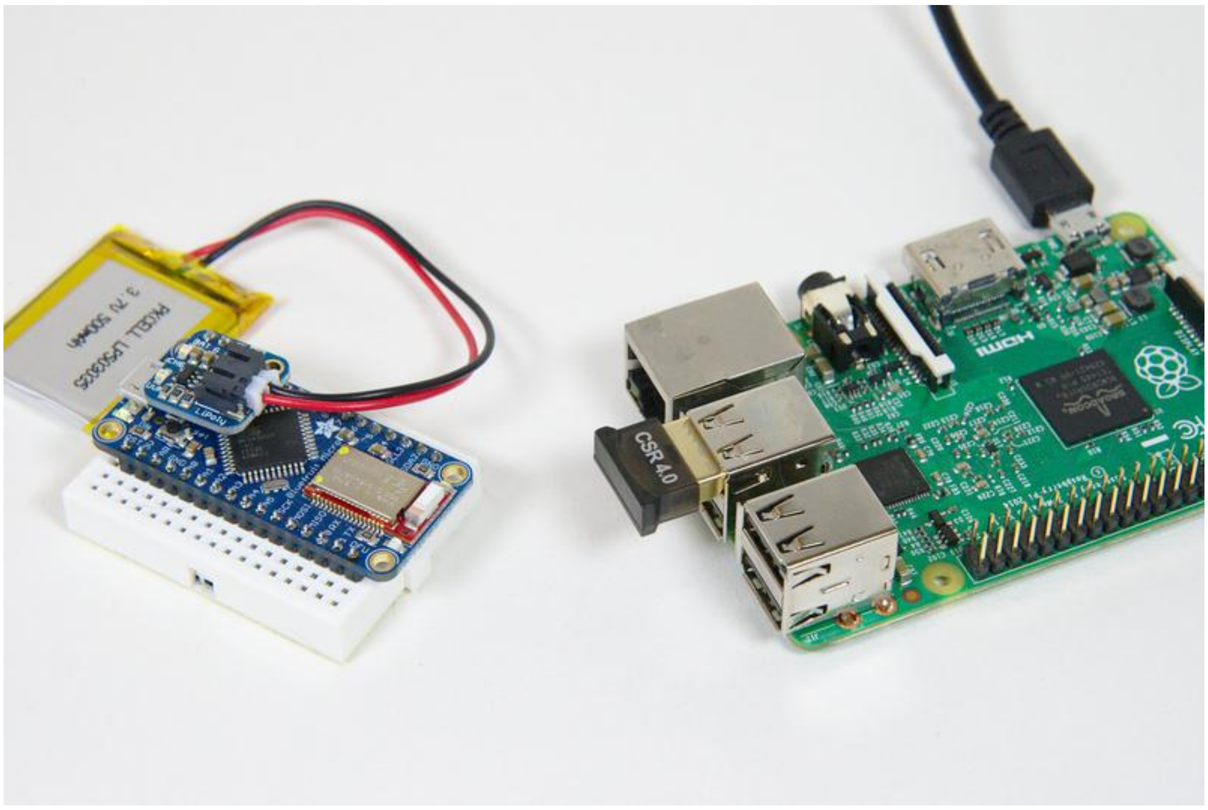
Bluefruit LE Python Library

Created by Tony DiCola



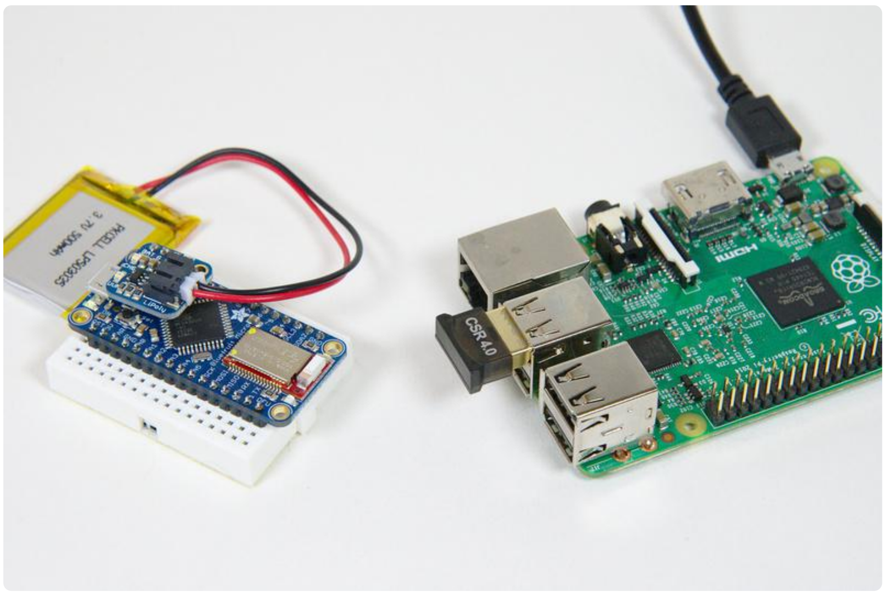
https://learn.adafruit.com/bluefruit-le-python-library

Last updated on 2022-12-01 02:35:56 PM EST

Table of Contents

Overview

This library is deprecated in favor of the Adafruit_Blinka_bleio library. See this guide for more information: https://learn.adafruit.com/circuitpython-ble-libraries-on-any-computer



So you've got a Bluefruit LE device () that's ready to be the next awesome wireless IoT gadget, but how do you actually talk to it from your computer? The Bluefruit LE Python library () is just what you need to write code that reads and writes data with a Bluefruit LE device! This Python library allows you to write simple code to talk to a Bluefruit LE UART from a Mac OSX computer or Linux machine, like a Raspberry Pi. This library is great for logging sensor data, controlling your device, and much more through the wireless magic of Bluetooth low energy!

To use this library you'll need to be running a Mac OSX or Linux machine with a Bluetooth 4.0 low energy adapter. Sorry Windows is not yet supported because Bluetooth low energy support is still a little bit too new to the platform (only the very latest Windows 10 release exposes enough BLE APIs to completely control a device). In the future Windows support might be added, but for now stick with a Linux machine, like a Raspberry Pi, or a Mac OSX desktop/laptop.

You'll also want to be somewhat familiar with the Python programming language and Bluetooth low energy. The Hitchhiker's Guide to Python has a great learning python section () with links to books and free resources for learning the language. For Bluetooth low energy be sure to read this excellent intro guide () and even consider picking up a book on the topic (http://adafru.it/1978).

When you're ready continue on to learn about what hardware you'll need to use the Bluefruit LE Python library.

Hardware

This library is deprecated in favor of the Adafruit_Blinka_bleio library. See this guide for more information: https://learn.adafruit.com/circuitpython-ble-libraries-on-any-computer

To talk to a Bluefruit LE from your Linux or Mac computer you'll need to make sure you have a Bluetooth 4.0+ low energy (sometimes called Bluetooth Smart) adapter.

For Mac OSX you don't have many options since Bluetooth support is built in to the hardware. Make sure you're using a device that has Bluetooth 4.0 low energy support. Most Mac laptops made since ~2012 should have Bluetooth 4.0 low energy support.

For a Linux machine like a Raspberry Pi this CSR8510 Bluetooth 4.0 USB dongle () is exactly what you want to use. In general if BlueZ () supports your Bluetooth hardware and it's 4.0 low energy then it should work (but no guarantees, we've only tested with the CSR8510 dongle).

Finally you'll need a Bluefruit LE device to talk to when using the library. Some of the options include:

- Bluefruit LE UART () or Bluefruit LE SPI friend (http://adafru.it/2633) - These devices connect to an Arduino through a UART or SPI connection respectively and allow the Arduino to expose itself as a BLE UART or other peripheral.
- Bluefruit LE Micro () - This is an all-in-one Bluefruit SPI friend connected to a ATmega32u4 processor (like a FLORA) and is a great option for a small BLE project.
- Bluefruit LE USB friend () - This is a Bluefruit LE UART friend that's connected to a serial to USB converter so it can be accessed by a computer. This is a good option if you have a computer like a Raspberry Pi that you want to turn into a BLE peripheral.

If you aren't sure which one to pick, consider using the Bluefruit LE SPI friend (http://adafru.it/2633) if you already have an Arduino or a Bluefruit LE Micro () if you don't have an Arduino.

Continue on to learn how to install the Python library that will talk to the Bluefruit LE.

Installation

This library is deprecated in favor of the Adafruit_Blinka_bleio library. See this guide for more information: https://learn.adafruit.com/circuitpython-ble-libraries-on-any-computer

Follow the appropriate steps below depending on your platform to install the Bluefruit LE Python library.

Mac OSX

On Mac OSX nothing extra needs to be installed to use the library. The library makes use of the PyObjC library () that Apple includes with the version of Python installed in OSX.

Note that if you're using a non-Apple version of Python, like one installed with Homebrew, you might need to manually install PyObjC ()!

Skip down to the Library Installation section () at the bottom to continue.

Linux & Raspberry Pi

The Raspberry Pi 3, 4, and Pi Zero W include Bluetooth capability. Recent versions of Raspbian include the bluez package. If it is not already installed, you can install it by doing:

```
apt list bluez  # See whether bluez is already installed.
# If not, do this:
sudo apt update
sudo apt install bluez
```

After you have installed bluez, you need to add yourself to the bluetooth user group:

```
sudo useradd -G bluetooth $USER
```

Then reboot.

Note the Python Bluetooth LE library below only supports Python 2.7 right now and not yet Python 3.x.

Library Installation

To install the library globally, use `pip`:

```
pip install --user Adafruit-BluefruitLE
```

Alternatively, you can clone it from its home on GitHub () and then run its setup.py as below. Assuming you have git () installed you can run the following in a terminal to clone the library and install it:

```
git clone https://github.com/adafruit/Adafruit_Python_BluefruitLE.git
cd Adafruit_Python_BluefruitLE
sudo python setup.py install
```

That's it, the library should be installed globally and ready to use with any Python script on your system.

Alternatively you can install with `sudo python setup.py develop' to put the library into develop mode where changes to the code are immediately reflected without the need to reinstall. This is handy if you're modifying the code or updating it frequently from GitHub.

Continue on to learn about the examples included with the library and how to use it in your own code.

Library Usage

This library is deprecated in favor of the Adafruit_Blinka_bleio library. See this guide for more information: https://learn.adafruit.com/circuitpython-ble-libraries-on-any-computer

Examples

To demonstrate the usage of the library look at the included examples in the example s subdirectory:

- list_uarts.py - This example will watch for any BLE UART devices and print out their name and ID when found. This is a good example of searching for devices.
- uart_service.py - This is an example of talking to a BLE UART device. When run the example will connect to the first BLE UART device it finds, send a string, and then wait to receive a string.
- low_level.py - This is similar in functionaliy to uart_service.py but implemented using a lower level direct interaction with BLE GATT services and characteristics. This is a good starting point for interacting with a custom BLE service or device.
- device_info.py - This example will connect to the first found BLE UART and print out information from its device info service, like serial number, hardware version, etc. Note this example only works on Mac OSX--a bug/issue in BlueZ currently prevents access to the device info service.

Since all of the examples deal with a BLE UART device you'll want to make sure you first have a Bluefruit LE device running as a BLE UART:

- Bluefruit LE UART friend - Connect it to an Arduino and run its BLEUart example (). Make sure its switch is in the CMD position if you're using the bleuart_cmdmo de example, and DATA position if you're using bleuart_datamode.
- Bluefruit LE SPI friend - Connect it to an Arduino and run its BLEUart example ().
- Bluefruit LE micro - Load the BLEUart example ().
- Bluefruit LE USB friend - Make sure the switch is in the DATA position, connect the device to your computer, and open its serial port with a terminal ().

Make sure to open the serial terminal in the Arduino IDE if you're running the BLEUart example. You'll want the terminal open so you can see the data received and send data to the BLE UART example code in the next step.

To run an example you just need to invoke it using the Python interpretor. On Linux make sure to run as root using sudo, and on Mac OSX it's not necessary (but will still work fine) to run with sudo. For example to run the uart_service.py example open a terminal, navigate to the examples directory of the library, and run:

```
sudo python uart_service.py
```

After a short period you should see the example run and start printing status messages out like the following:

```
Using adapter: BlueZ 5.33
Disconnecting any connected UART devices...
Searching for UART device...
Connecting to device...
Discovering services...
Sent 'Hello world!' to the device.
Waiting up to 60 seconds to receive data from the device...
```

Note if you see an error 'org.freedesktop.DBus.Error.UnknownMethod: Method "Set" with signature "ssb" on interface "org.freedesktop.DBus.Properties" doesn't exist' run the example again. It appears this is a transient or one time error from setting up the DBus connection with BlueZ.

The example will attempt to connect to the first BLE UART device it finds and send & receive data with it. On Linux with BlueZ be patient as the searching for UART device and the discovering services phase can take around 30-60 seconds the first time it runs. On Mac OSX BLE actions are generally quite fast and happen in a few seconds.

Once you see the example print "Waiting up to 60 seconds to receive data from the device..." then check the serial terminal of the Bluefruit LE device. You should see it received the string 'Hello World!' (without quotes). For example the Bluefruit LE UART / SPI / micro terminal will show:

```
H [0x48] e [0x65] l [0x6C] l [0x6C] o [0x6F]   [0x20] w [0x77] o [0x6F] r [0x72] l
[0x6C] d [0x64] ! [0x21]  [0x0D]
 [0x0A]
```

Now while the uart_service.py example is still running enter some text in the serial terminal and press send. You should see the text received and the uart_service.py example exit:

```
Received: test
```

Congrats, you've successfully run the uart_service.py example!

If you run into issues with the example first make sure the Bluefruit LE Python library was successfully installed from the previous page. Also make sure only one BLE UART device is running--if more than one are running the code might get confused and connect to a device you don't expect (it just connects to the first device it finds).

You can run the other 3 examples in the same way as you ran the uart_service.py example. Note that the device_info.py example only runs on Mac OSX right now. There's a bug or issues with BlueZ that prevents accessing the device information service.

Usage

To understand how to use the library to send and receive data with a BLE UART I'll walk through the uart_service.py example below. Open the file () in a text editor and follow along:

```
import Adafruit_BluefruitLE
from Adafruit_BluefruitLE.services import UART
```

The first part of the code is the import section that pulls in the Bluefruit LE library. Notice that both the main library is imported with the first line, and a special UART service implementation is imported with the second line.

```
# Get the BLE provider for the current platform.
ble = Adafruit_BluefruitLE.get_provider()
```

Next a BLE provider is created by calling the Adafruit_BluefruitLE.get_provider() function. This will grab the appropriate BLE provider for your platform, either Linux or Mac OSX. You only need to call this once at the very start of your program to get the provider that will be used in future BLE calls.

After the provider is created you'll notice a main function is defined. We'll actually skip this function for now and come back to it. Scroll down past the main to the end of the file so you can see where execution of the script starts:

```
# Initialize the BLE system.  MUST be called before other BLE calls!
ble.initialize()

# Start the mainloop to process BLE events, and run the provided function in
# a background thread.  When the provided main function stops running, returns
# an integer status code, or throws an error the program will exit.
ble.run_mainloop_with(main)
```

First the code calls the initialize() function on the provider to setup BLE communication. This is required before you make any other calls to the library.

Next the code calls run_mainloop_with() and passes it the main function that was created above. One very important thing to realize with BLE on desktop platforms is that it generally requires a full GUI event loop to run in the background. Actions like connecting to a device, searching for a device, etc. are actually asyncronous and start and stop at different times. An event loop is necessary to make sure these asyncronous events are processed. However an event loop complicates your program, especially if you just want to write a simple script that performs syncronous actions against BLE devices.

The Bluefruit library attempts to hide all of this event loop logic and just present an easy to use syncronous interface. By calling the run_mainloop_with function the library will setup all the platform-specific event loop logic and then invoke the provided function in a background thread. The event loop runs in the main program thread and processes BLE events and your code runs in a background thread.

A side effect of your code running in a background thread is that you need to be careful that any callbacks from the Bluefruit library are made to be thread safe. In normal interaction with the UART service you don't need to worry about thread safety as the service takes care of moving data between the main and background thread with a queue. However if you dig into lower level direct access with BLE characteristics be aware that their callbacks won't be performed on the same thread as your main code.

It's also good to realize that the Bluefruit library is really targeted at simple scripts that interact syncronously with a BLE UART. For example a script that polls a BLE device to read sensor data and save it to a file or push to a web service is a good example of what the library can do. If you're building a GUI application that presents a user interface then the library might not be the best option for you. When you build a GUI app you want control over the main event loop yourself and likely want to choose how BLE events are processed, like in a background thread. You also likely want calls to the BLE device to happen asyncronously, that is in the background instead of blocking execution until they finish. When a GUI application uses a syncronous or blocking API like this Bluefruit library exposes it can be a poor user experience with a slow GUI. If you're building a GUI application consider talking directly to your platform's BLE APIs (like CoreBluetooth () on OSX or BlueZ () on Linux).

Now let's go back up and look at the main function that was defined earlier:

```
# Main function implements the program logic so it can run in a background
# thread.  Most platforms require the main thread to handle GUI events and other
# asyncronous events like BLE actions.  All of the threading logic is taken care
# of automatically though and you just need to provide a main function that uses
# the BLE provider.
def main():
    # Clear any cached data because both bluez and CoreBluetooth have issues with
    # caching data and it going stale.
    ble.clear_cached_data()
```

The function will run in a background thread and performs all of the main logic of the script.

To start the main function calls the clear_cached_data() function to reset any platform BLE state. Unfortunately this is a bit of a hack that's necessary on most platforms as BLE support is still somewhat immature.

On Linux with BlueZ this function will remove any known BLE devices so they can be rediscovered again fresh. This is necessary because of bugs and issues with heavy caching of devices (for example if a BLE UART is turned off BlueZ will still remember it for some time and confuse your program).

On Mac OSX with CoreBluetooth this function will perform these steps () to clear its cache. Again this is necessary because CoreBluetooth has heavy caching which can confuse your program when devices appear and disappear.

```
    # Get the first available BLE network adapter and make sure it's powered on.
    adapter = ble.get_default_adapter()
    adapter.power_on()
    print('Using adapter: {0}'.format(adapter.name))
```

Next the BLE adapter for the platform is retrieved with the get_default_adapter() function, and it's powered on by calling the power_on() function. These are necessary to make sure the computer has Bluetooth turned on and ready to find devices.

```
    # Disconnect any currently connected UART devices.  Good for cleaning up and
    # starting from a fresh state.
    print('Disconnecting any connected UART devices...')
    UART.disconnect_devices()

    # Scan for UART devices.
    print('Searching for UART device...')
    try:
        adapter.start_scan()
        # Search for the first UART device found (will time out after 60 seconds
        # but you can specify an optional timeout_sec parameter to change it).
        device = UART.find_device()
        if device is None:
            raise RuntimeError('Failed to find UART device!')
    finally:
        # Make sure scanning is stopped before exiting.
        adapter.stop_scan()
```

The next block of code deals with searching for a UART device. First any connected UARTs are disconnected by calling the UART.disconnect_devices() function. This is necessary because a BLE central device (your computer) can only be connected to a single BLE peripheral (your BLE UART devices) at a time. If your computer was still connected to a BLE UART then it would fail to find any new ones.

After disconnecting UART devices the code calls adapter.start_scan() to turn on device scanning. Then the UART.find_device() function is called to find the first available UART device.

Note that if you'd like to find a specific UART device, perhaps one that has a different name, you can instead call UART.find_devices() (plural) and it will immediately return a list of all known UART devices. Enumerate the list and look for a device that you'd like to connect to, or keep calling the find_devices function periodically in a loop to scan for new devices (see the list_uarts.py example for this kind of logic).

Finally notice the block ends with a finally block that ensures the adapter.stop_scan() function is called. This is good practice to make sure the adapter doesn't stay in a scanning mode if the program should exit prematurely. Also before you connect to a device (the next code) you'll want to make sure scanning is turned off.

```
    print('Connecting to device...')
    device.connect()  # Will time out after 60 seconds, specify timeout_sec
parameter
                      # to change the timeout.
```

Now a connection is created with the device that was found by calling its connect() function. This function will wait up to 60 seconds for the device to respond and connect (note you can change this timeout by passing in a timeout_sec parameter with a new value).

```
    # Once connected do everything else in a try/finally to make sure the device
    # is disconnected when done.
    try:
        # Wait for service discovery to complete for the UART service.  Will
        # time out after 60 seconds (specify timeout_sec parameter to override).
        print('Discovering services...')
        UART.discover(device)

        # Once service discovery is complete create an instance of the service
        # and start interacting with it.
        uart = UART(device)
```

Next a new try/finally block is created to ensure the device connection is closed, and the main UART logic starts to run.

First the UART services are discovered by calling UART.discover() and passing it the connected device. When connecting to a device you must ensure all the services you'll use have been discovered so that the platform knows what characteristics, etc. are available on the service.

After the service discovery completes a UART object is created by passing it the connected device. This UART object implements a simple send and receive interface to interact with the connected UART.

```
        # Write a string to the TX characteristic.
        uart.write('Hello world!\r\n')
        print("Sent 'Hello world!' to the device.")
```

Data is sent to the UART device by calling the UART object's write function. Pass the function a string and it will ensure the data is written to the connected device--easy!

```
        # Now wait up to one minute to receive data from the device.
        print('Waiting up to 60 seconds to receive data from the device...')
        received = uart.read(timeout_sec=60)
        if received is not None:
            # Received data, print it out.
            print('Received: {0}'.format(received))
        else:
            # Timeout waiting for data, None is returned.
            print('Received no data!')
```

Data is received by calling the UART object's read function. Note that a timeout of 60 seconds is passed in which tells the function to wait up to 60 seconds for data to be received. If something is received within that time then it will be returned, otherwise the None result is returned which signifies the timeout elapsed before data was received.

If you don't want any timeout just call read with no extra parameters. The default timeout value of 0 will be used which means it does a quick check for any received data and instantly returns. In the background the UART object uses a queue to keep track of data received from the UART device so you don't need to worry about constantly calling read and potentially missing data. Just make sure to call read when you're ready to get data.

```
    finally:
        # Make sure device is disconnected on exit.
        device.disconnect()
```

The very last part of the function is the finally block that disconnects the device. It's good practice to make sure the device is disconnected at the end of the program execution by putting it in a finally block.

Note that on some platforms like Linux if you stop a program with Ctrl-C it might not actually invoke this finally block. The reason is with the main thread being used to process GUI events. As a side effect this means Ctrl-C will stop the main thread and the entire program so your finally block in a background thread won't run. You can work around this by using Python's atexit module () to register a cleanup function that's called when your program ends. See the list_uarts.py example for a demonstration of atexit for cleanup.

In general it's best to not rely on Ctrl-C to stop a program using the library, intead use an explicit action like an input from a user to end the program or just run for a specific amount of time. Also be aware Ctrl-C to exit can be slow on OSX, perhaps taking 30 seconds or more to shut down the main loop (and even throwing crash errors on Mac OSX Yosemite).

That's all there is to using the Bluefruit LE Python library to talk to a BLE UART device! The library works great for connecting to a BLE UART to read sensor data, send it actions, etc--the sky is the limit with what you can do using a Bluefruit LE device and your computer!