CycleFlow Integration Guide

The CycleFlow Team
cycleflow@squircle.ca

Version 1.0

Important Notes

CycleFlow was not designed to replace any safety-critical systems. CycleFlow's use should be limited to augmenting existing traffic signals, not as a substitute. All hardware and software implementations of the CycleFlow protocol **must** degrade safely.

CycleFlow is a fourth-year engineering design project. It was not developed by licensed professional engineers, nor has it been scrutinized by any regulatory bodies.

This guide is written for users who wish to use the CycleFlow protocol with hardware or software other than the reference implementations provided at `https://github.com/squircle/cycleflow`. It describes the communication protocol in detail, and outlines general considerations for development.

This is **not** a step-by-step development guide. It exists as a reference to aid interoperability of CycleFlow deployments on diverse platforms.

1 System Overview

CycleFlow is a system that transmits traffic signal timing information to cyclists so they can preserve momentum and move easily through urban areas. The protocol uses standard Bluetooth technology that can be implemented on a wide range of client and base station hardware.

A high-level overview of CycleFlow in action is the following:

1. Base station receives timing information from traffic signal controller
2. Base station updates Bluetooth broadcast with updated information

3. Client receives and decodes Bluetooth broadcast
4. Client determines which intersection user is approaching and their ETA
5. Based on this data, client notifies user of speed recommendation

The reference implementation is based on Nordic Semiconductor's nRF52 Bluetooth platform (base station) and the Android smartphone platform (client). CycleFlow should work on any platform that meets the hardware requirements detailed below; it was designed to be easily integrated as it has no hardware-specific dependencies.

> This is only a brief synopsis of CycleFlow's operation. For more information, please see the overview documentation on the project's site.

2 Hardware Requirements

The CycleFlow base station must support, at minimum:

- Bluetooth Low Energy (known as Bluetooth 4.0, Bluetooth LE, BLE, or Bluetooth Smart)
  - Non-connectable, undirected Generic Access Profile (GAP) advertising
  - Fully customizable GAP advertising packets with Manufacturer Specific Data type (`0xFF`)
- Traffic data exchange interface
  - Generally UART (serial)
  - Requires custom translation layer for each type of traffic signal controller

The following features may be desirable for real-world implementations:

- External antenna support
- Uninterruptable power or battery backup
- Weather sealing or ruggedizing
- Debugging or logging interface (*e.g.* display, SD card, etc.)

The reference implementation consists of the following hardware:

- Nordic Semiconductor nRF52832 BLE SoC
  - ARM Cortex M4F processor
  - Bluetooth 5 and BLE compliant radio
  - Nordic S132 SoftDevice Bluetooth-qualified protocol stack
  - Hardware UART (requires level shifting for RS-232 integration)
  - GPIO (for monitoring state, and future integrations)
  - Adjustable transmit power (+4 to -20 dBm and external antenna support with impedance matching network)
- Android 5.0 Lollipop smartphone platform
  - Full BLE protocol stack support (with compliant hardware)
  - GPS using Google Play Services Location API

⚠ Hardware that meets the above requirements on paper may not function as expected in the field. Nothing can replace real-world integration testing to discover quirks or incompatibilities.

3 CycleFlow Protocol

3.1 Bluetooth Advertising Format

The key interchange format of CycleFlow is BLE advertisement frames. In each advertising interval, up to two frames are broadcast on the dedicated broadcast channels. Each frame has a payload of up to 31 B, depending on

- **Advertising frame:** contains safety-critical information like position, light timing, etc.
- **Scan response frame:** contains the friendly (human-readable) intersection name (UTF-8 format)

The scan response frame is only sent in response to a scan request from a client BLE device. In practice, this is usually handled by the BLE client when scanning for peripherals, but may depend on its API.

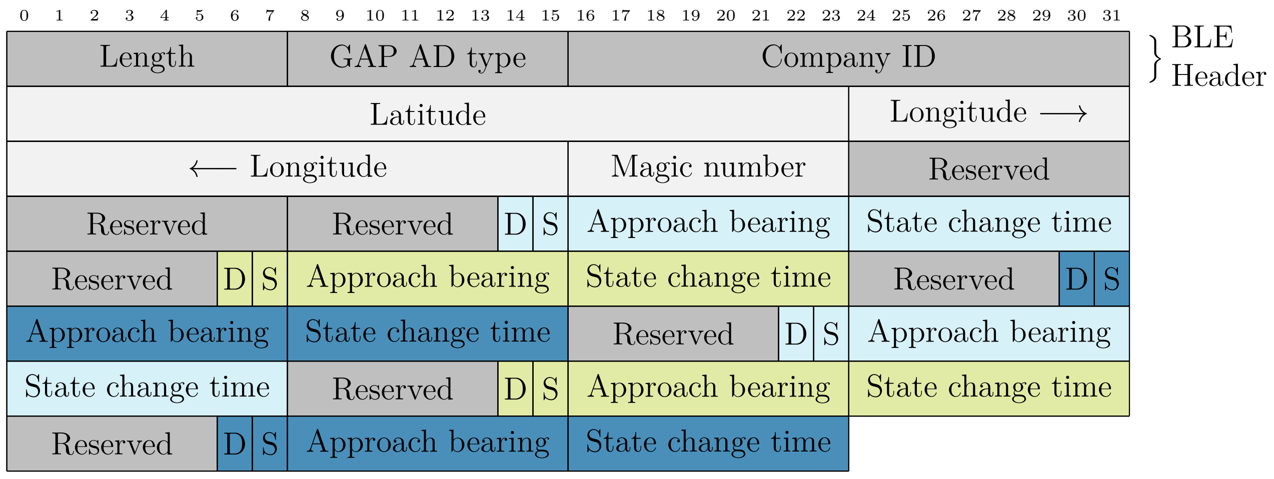
Minimum BLE advertising interval (by BLE specification): 100 ms
Recommended BLE advertising interval: 100 ms

3.1.1 Advertising Frame

The advertising frame payload has a length of $13 + (3 \times n)$ bytes, where n is the number of intersection entrances.

A single advertising frame can broadcast up to 6 entrances, sufficient for even the most complex entrances. However, if more than 6 entrances are required, multiple base stations can be situated at an intersection. The disambiguation between base stations is performed by the client.

The structure of the advertising frame is as follows (*n.b.* the diagram is 32 bits wide and contains 31 bytes in total):



Each field is described below:

- **Length**

  Payload length in bytes, excluding the length field. Minimum value `0x0F` for one entrance, increasing by increments of 3 to a maximum value of `0x1E` for six entrances.

  > Many Bluetooth hardware APIs calculate this automatically, so it doesn't need to be explicitly included.

- **GAP AD type**

  `0xFF`, a constant, indicating manufacturer-specific data follows (*i.e.* not following a standard Bluetooth SIG-defined data layout)

- **Company ID**

  A constant field, identifying the manufacturer that defines the format for the following data. `0xFFFF` is reserved for testing/internal use. CycleFlow has not yet obtained a permanent ID from Bluetooth SIG.

- **Latitude**

  A 24-bit big-endian integer containing an encoded representation of the base station's latitude. See Section 3.1.2 below for encoding details.

- **Longitude**

  A 24-bit big-endian integer containing an encoded representation of the base station's longitude. See Section 3.1.2 below for encoding details.

- **Magic Number**

  `0xCF`, a constant, positively identifying a CycleFlow base station until a permanent Company ID is obtained. This field may be repurposed in the future, once an ID is obtained.

- **Entrance Data**

  Each entrance data block – identified by a colour in the bitfield above – begins with 6 reserved bits, then follows this format:

- **D** – a bit indicating whether the traffic signal facing this entrance is timer-based (`0`) or requires a demand trigger (`1`). If the signal is demand-based (`1`), a value of 255 (`0xFF`) in the *State change time* field signifies no demand, and should be interpreted as an absolute stop until this bit changes.
- **S** – a bit indicating whether the traffic signal facing this entrance is currently in a blocked (`0`, *i.e.* amber or red) or permissive (`1`, *i.e.* green).
- **Approach bearing** – an octet containing the heading for this entrance (*i.e.* the client's heading as they approach this entrance), used to determine which signals apply. This value is normalized in 240 steps of 1.5°. Valid values are 0-239 (`0x00-0xEF`), and 255 (`0xFF`) for omnidirectional. Values between 240 and 254 (`0xF0-0xFE`) should not be used.
- **State change time** – an octet containing the time to the next state (*i.e.* change in the D bit) in seconds. The maximum value 255 (`0xFF`) should be interpreted as infinity.

3.1.2 Latitude & Longitude Calculation

Due to space constraints in the protocol, geographic coordinates must be normalized. However, storing each as a 24-bit integer is still sufficiently precise to disambiguate intersections, as the minimum precision (at the equator) is approximately 3 m. Since the position of a base station shouldn't change under normal operation, none of these floating-point calculations are required on the base station.

Constants Each calculation involves the use of five constants, given below in both C and Java format:

```
const unsigned long LATLONG_INT_OFFSET = 8388540;

const unsigned long LATITUDE_TO_INT_FACTOR = 93206;
const unsigned double LATITUDE_TO_FLOAT_FACTOR = 0x1.6800bf40659a3p-17;

const unsigned long LONGITUDE_TO_INT_FACTOR = 46603;
const unsigned double LONGITUDE_TO_FLOAT_FACTOR = 0x1.6800bf40659a3p-16;
```

```
public static final long LATLONG_INT_OFFSET = 8388540;

public static final int LATITUDE_TO_INT_FACTOR = 93206;
public static final double LATITUDE_TO_FLOAT_FACTOR = 0x1.6800bf40659a3p-17;

public static final int LONGITUDE_TO_INT_FACTOR = 46603;
public static final double LONGITUDE_TO_FLOAT_FACTOR = 0x1.6800bf40659a3p-16;
```

The floating-point factors are stored as IEEE 754 double-precision constants to ensure consistency. No interpretation is required, as these are exact definitions.

Conversion to Integer Format Use this conversion method when you have identified the geographic coordinates of the intersection, and wish to encode them for use in the base station.

Latitude

1. **Multiply** floating-point latitude by `LATITUDE_TO_INT_FACTOR`.
2. **Round** at the units place (removing the fractional part).
3. **Add** `LATLONG_INT_OFFSET` to the result.

Longitude

1. **Multiply** floating-point latitude by `LONGITUDE_TO_INT_FACTOR`.
2. **Round** at the units place (removing the fractional part).
3. **Add** `LATLONG_INT_OFFSET` to the result.

Conversion to Floating-Point Format Use this conversion method in the client implementations, to translate an encoded lat/long pair to proper coordinates.

Latitude

1. **Subtract** `LATLONG_INT_OFFSET` from the integer latitude.
2. **Signed multiply** the difference by `LATITUDE_TO_FLOAT_FACTOR`.
3. **Round** the result to 5 decimal places or fewer.

Longitude

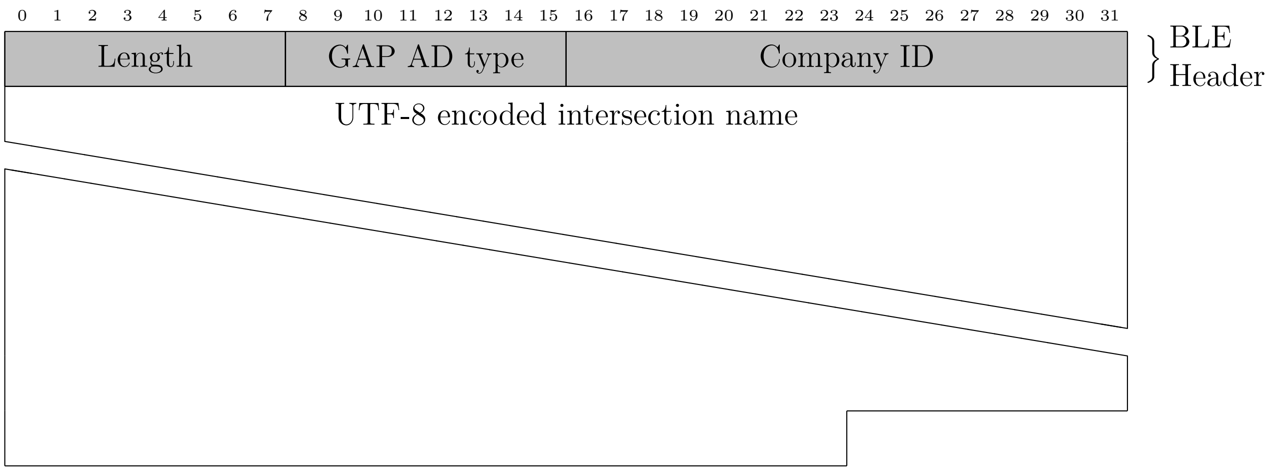
1. **Subtract** `LATLONG_INT_OFFSET` from the integer longitude.
2. **Signed multiply** the difference by `LONGITUDE_TO_FLOAT_FACTOR`.
3. **Round** the result to 5 decimal places or fewer.

A sample Python script to perform the float-to-integer conversion follows:

```python
#!/usr/bin/env python3

import sys

print("Usage: latlong.py latitude longitude")

LATLONG_INT_OFFSET = 8388540
LATITUDE_TO_INT_FACTOR = 93206
LATITUDE_TO_FLOAT_FACTOR = float.fromhex("0x1.6800bf40659a3p-17")
LONGITUDE_TO_INT_FACTOR = 46603
LONGITUDE_TO_FLOAT_FACTOR = float.fromhex("0x1.6800bf40659a3p-16")

latitude = float(sys.argv[1])
longitude = float(sys.argv[2])

i_latitude = round(latitude * LATITUDE_TO_INT_FACTOR) + LATLONG_INT_OFFSET
i_longitude = round(longitude * LONGITUDE_TO_INT_FACTOR) + LATLONG_INT_OFFSET

print("Input: {}, {}".format(latitude, longitude))
print("Output: {} ({}), {} ({})".format(i_latitude, hex(i_latitude),
                                        i_longitude, hex(i_longitude)))
```

3.1.3 Scan Response Frame

The scan response frame header is four bytes, plus up to 27 bytes of UTF-8 encoded text (for a maximum length of 31 bytes). This text communicates the friendly name of the intersection to the user. The format of the frame is as follows:



The format of the BLE header is the same as the advertisement frame.

Many client implementations will present both the advertisement and scan response frames as a single bitstream. Consider using the length fields to determine where the two frames are divided.

Client implementations – especially those written in memory-unsafe languages – should implement strict bounds checking to prevent against buffer overruns by malicious BLE advertisements.

3.2 Reference Implementation

The CycleFlow Github repository (`https://github.com/squircle/cycleflow`) contains a reference implementation for a base station (using a Nordic Semiconductor nRF52-DK development kit) and a client (using Android).

The base station implementation has two modes of operation: static demo, and timer-based demo. The implementation could be easily extended to interface with a traffic signal controller using the pre-built Nordic UART module.

Nordic's development kit supports multiple methods of development. The reference implementation uses GCC/Clang and the ARM SDK, but there are multiple fully-featured IDEs and debugging tools available for free or at nominal cost. See `https://www.nordicsemi.com/DocLib/Content/User_Guides/getting_started/latest/UG/gs/develop_sw` for more information on software development.