




Degree Programme
Systems Engineering

Major Infotronics

Bachelor's thesis
Diploma 2020

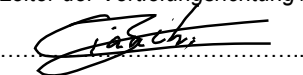
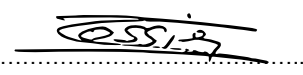
Rossier Yoan

IoT bike gateway for Swiss Cycling

-  *Professor*
Rieder Medard
-  *Expert*
Diethelm Bruno
-  *Submission date of the report*
14.08.2020

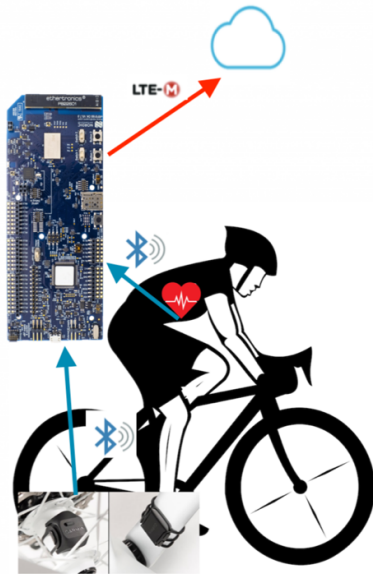
Filière / Studiengang SYND	Année académique / Studienjahr 2019/20	No TD / Nr. DA it/2020/58
Mandant / Auftraggeber <input type="checkbox"/> HES—SO Valais <input checked="" type="checkbox"/> Industrie <input type="checkbox"/> Etablissement partenaire Partnerinstitution	Etudiant / Student Yoan Rossier Professeur / Dozent Medard Rieder	Lieu d'exécution / Ausführungsort <input checked="" type="checkbox"/> HES—SO Valais <input type="checkbox"/> Industrie <input type="checkbox"/> Etablissement partenaire Partnerinstitution
Travail confidentiel / vertrauliche Arbeit <input type="checkbox"/> oui / ja ¹ <input checked="" type="checkbox"/> non / nein	Expert / Experte (données complètes) Hauptexperte Bruno Diethelm , Nationalcoach bruno.diethelm@swiss-cycling.ch Freienhofgasse 12, 3600 Thun Nebenexperte Lucas Schmid, Ausbildungsverantwortlicher: lucas.schmid@swiss-cycling.ch	

Titre / Titel IoT bike gateway for Swiss Cycling
Description / Beschreibung The goal of this project is to develop a tool to help cyclists during training. At this time, the analysis of the metrics collected from the sensors on the bike and on the cyclist is only possible at the end of the session. Getting a feedback from the coach during the session would significantly improve the quality of the training. The tool is a gateway that collects metrics from the sensors mounted on the bike and on the body during a short segment. It uploads those metrics to a Web platform during the training. Right after the segment is cleared by the athlete, the coach has access to all the metrics. A detailed project specification will be handed out to the student at the begin of the bachelor work. Must goals: 1. Design and implement a wearable gateway able to a. collect metrics from the sensors over BLE (two metrics out of these four: speed, cadence, power, heart rate) b. upload timestamped metrics to a cloud system over LTE-M 2. Design a cloud system: a. configure / implement a basic cloud system (time series database) b. integrate an existing time measurement system into the cloud system Optional goals: 3. Design and implement a gateway configuration web page 4. Correlate time series data from the time measurement system and the metrics of the gateway 5. Implement all four metrics Deliverables: Project report in English language, SW & HW documentation, project presentation

Signature ou visa / Unterschrift oder Visum Responsable de l'orientation / filière Leiter der Vertiefungsrichtung / Studiengang:  ¹ Etudiant / Student : 	Délais / Termine Attribution du thème / Ausgabe des Auftrags: 25.05.2020 Présentation intermédiaire / Zwischenpräsentation Semaine / Woche 26 (22.06 – 26.06.2020) Remise du rapport / Abgabe des Schlussberichts: 14.08.2020, 12:00 Exposition / Ausstellung der Diplomarbeiten: 28.08.2020 (si autorisé / falls genehmigt) Défense orale / Mündliche Verfechtung: Semaine / Woche 36 (31.08 – 04.09.2020)
---	--

¹ Par sa signature, l'étudiant-e s'engage à respecter strictement la directive DI.1.2.02.07 liée au travail de diplôme.
Durch seine Unterschrift verpflichtet sich der/die Student/in, sich an die Richtlinie DI.1.2.02.07 der Diplomarbeit zu halten.

IoT bike gateway for Swiss Cycling



Bachelor's Thesis
| 2020 |

Degree programme
Systems Engineering

Field of application
Major Infotronics

Supervising professor
Rieder Medard
medard.rieder@hevs.ch

Partner
Swiss Cycling

Graduate

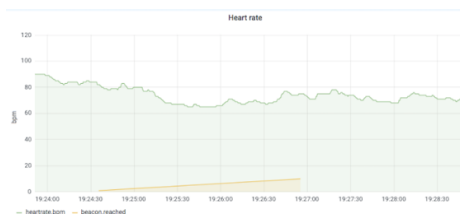
Rossier Yoan

Objectives

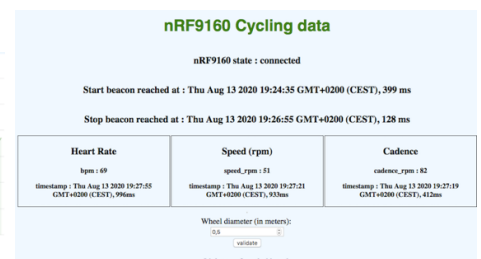
Developing a tool tracking cyclist during short training segment. This tool will allow trainer to see metrics, from sensors on bike and on cyclist, in real time. So, he can give feedback right after the training segment.

Methods | Experiences | Results

To reach the goal, the nRF9160 DK from Nordic Semiconductor is used. This board allows collecting data from the sensors through Bluetooth. Then, the board can connect to the cellular network and send the collected data to the cloud. When data are in the cloud, a simple web server get them and insert them into a database (Influx DB). After data are in the database, the Grafana tool allows to see them as a graph. It's also possible to see the Grafana graphs from the web page. In order to synchronize data with the beginning and the end of the training segment, beacons are placed at the beginning and the end of the segment. With this, the beginning and the end of the segment can be seen on the graph.



Grafana graph of some collected metrics.



Simple web page to get data from MQTT broker.

IoT bike gateway for Swiss cycling : Report

Rossier Yoan

August 2020

Contents

1	Introduction	1
1.1	Context	1
1.2	Goals	1
1.3	In practice	1
2	Material	2
2.1	Overview	2
2.2	nRF9160DK	3
3	Development environment	4
3.1	Programming language	4
3.2	Nordic environment	4
3.3	Web and database environment	5
4	Global development	5
4.1	Deployment	5
4.2	Implementation overview	6
5	Gateway start	9
5.1	LEDs status	9
5.2	LTE-M connection	9
5.3	Getting timestamp	9
5.3.1	Test on timestamps	10
5.4	Classes initialization	11
6	MQTT part	11
6.1	LTE-M and the nrf9160 SiP	11
6.2	About MQTT	11
6.3	MQTT state machine	12
6.3.1	ST_MQTT_INIT	13
6.3.2	ST_CONNECTED	14
6.3.3	ST_DISCONNECTED	14
6.3.4	ST_PING_ACK and ST_PUB_ACK	14
6.4	Application start and MQTT init	14
6.4.1	The XF	14
6.5	Incoming messages from the MQTT broker	14
6.6	Analyse with MQTT explorer	16
6.7	MQTT tests	16
6.7.1	MQTT state machine test	16
6.7.2	Test reconnection to the MQTT server	17
6.7.3	Test if the "will" message is well sent	18
6.7.4	Test PING response	19

7	BLE part	19
7.1	Bluetooth Low Energy	20
7.1.1	GAP Layer	21
7.1.2	GATT Layer	21
7.1.3	Cycling sensors	22
7.2	HCI	23
7.2.1	Definitions	23
7.2.2	On the nRF9160DK	23
7.3	BLE state diagram	24
7.3.1	Initialization state	26
7.3.2	Scan state to subscribe done	26
7.3.3	Waiting for the beacons states	27
7.3.4	New value from the sensor state	27
7.4	From the BLE initialization to the first notification in details	27
7.5	New value from CSC cadence sensor	30
7.5.1	CSC sensor notification	30
7.5.2	Calculate rpm for CSC	31
7.6	BLE beacons reached	33
7.7	BLE_controller tests	33
7.7.1	Connect and subscribe to sensors	33
7.7.2	Receive notification from sensor	34
7.7.3	Detect beacons	35
8	BLE and MQTT : global behavior	36
9	Web server part	38
9.1	Collect data from the MQTT broker	38
9.2	InfluxDB	41
9.2.1	Database and measurements for this project	41
9.3	Grafana	42
9.3.1	Panel configuration	42
9.3.2	<iframe>	42
9.4	Web page	43
9.5	Calcul of speed	44
9.6	Tests	45
9.6.1	One hour test	45
9.6.2	Beacons test	46
10	Embedded board	47
10.1	nRF9160DK power supply	47
10.2	Powerbank	47

11 Conclusion	47
11.1 Final state	47
11.2 Things to improve	47
11.3 Acknowledgements and final words	48
12 Annexes	48
12.1 Installation of project components	48
12.1.1 Install nRF connect SDK and set up environment	48
12.1.2 install Grafana and Influx	48
12.1.3 Webstorm	48
12.1.4 Build and flash on the nRF9160DK	48
12.2 Start Grafana and InfluxDB	49
12.3 How to run application	49
12.4 Diagrams	49
12.4.1 Complete nRF9160DK class diagram	50
12.4.2 Complete web server class diagram	51

1 Introduction

1.1 Context

Nowadays, when cyclists train, coaches can only obtain measurements at the end of the training session from sensors located on the bike and on the cyclist. If they could collect metrics in live during the session, it would allow them to give a direct feedback to the cyclist right after the session. Therefore ,Swiss Cycling¹, the national cycling federation, is looking for a way to resolve this issue for short segment training.

1.2 Goals

According to the diploma thesis guidelines, the main goals are the following :

- Collect metrics from sensors over bluetooth low energy (BLE).
- Send these collected metrics to a cloud system.
- Implement a basic cloud system (time series database).
- Integrate an existing time measurement system into the cloud system.

1.3 In practice

First, to reach these goals, a board that can do BLE and cloud connection is needed. In that aim, the nRF9160 Development Kit from Nordic semiconductor is appropriate, because it contains two chips : the nRF9160 SiP(System in Package) for cellular connection and the nRF52840 SoC (System on Chip) for BLE scanning and connection.

Second, collected metrics need to be sent to a cloud, which can then allow a web server to collect the data. An IoT way to do that is to send data through MQTT, a lightweight messaging protocol that allows to publish or subscribe messages to a topic on a MQTT broker² and then the web server can get these data from the broker.

Third, one needs a way to store and visualize the data in the web server side. To achieve this, a time series database (TSDB) and a tool allowing to visualize the data are required. The chosen TSDB is InfluxDB and the tool to visualize the stored data is Grafana.

Fourth, to synchronize the collected data to the short segment, a time measurement system is needed. For that purpose, a BLE beacon is chosen. Thanks to this, the collected data have the same source of timestamp as the beacon and coaches can see the beginning and the end of the segment directly on the same graph as the measurements.

¹<https://www.swiss-cycling.ch/de/swiss-cycling/>

²broker.hivemq.com or mqtt.eclipse.org

All these steps can be seen in Figure 1.

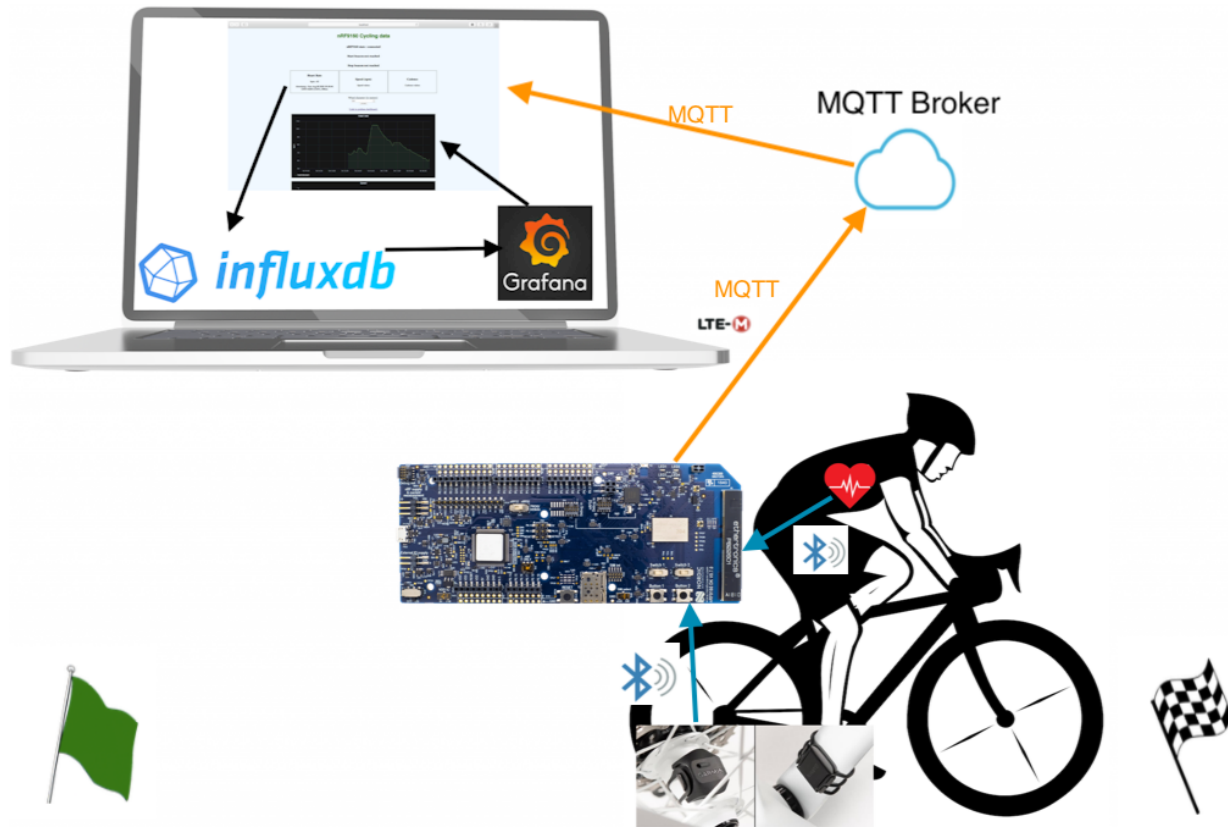


Figure 1: Summary of IoT bike gateway for Swiss cycling. The beacons are represented by flags.

2 Material

2.1 Overview

The following material has been used in the project :

- nRF9160DK V0.8.5 from Nordic Semiconductor³ :
 - nRF9160 SiP (modem firmware V1.2.0).
 - nRF51840 SoC.
- SIM card from Swisscom.
- BLE sensors from Garmin :
 - Heart rate monitor HRM-DUAL.

³<https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF9160-DK>

- Speed sensor 2 and cadence sensor 2.
- Two BLE beacons nRF51 DK, V1.1.0 and V1.2.2, containing both a nRF51422 SoC. (nRF51 DK with the Beacon Transmitter Sample Application⁴).
- A 6000 mA Power bank.

2.2 nRF9160DK

The nRF9160DK⁵ is the development kit for the nRF9160 SiP and contains two chips :

- nRF9160 SiP.
- nRF51840 SoC.

The nRF9160 SiP (Figure 2) is able to communicate through cellular network (LTE-M) when a SIM card has been included. There are two parts in this chip : the application part with the application code and the modem part, which contains the modem firmware. Thus, the application and the modem have both their processor⁶. Other functionalities, such as a GPS antenna, exist but have not been used in this project.

The other chip on the board is the nRF52840 SoC, which is able to communicate through BLE with a 2.4GHz antenna. In the DK, this chip is the board controller, which is able to route some pins to others. This has been used to link the two chips together to communicate.

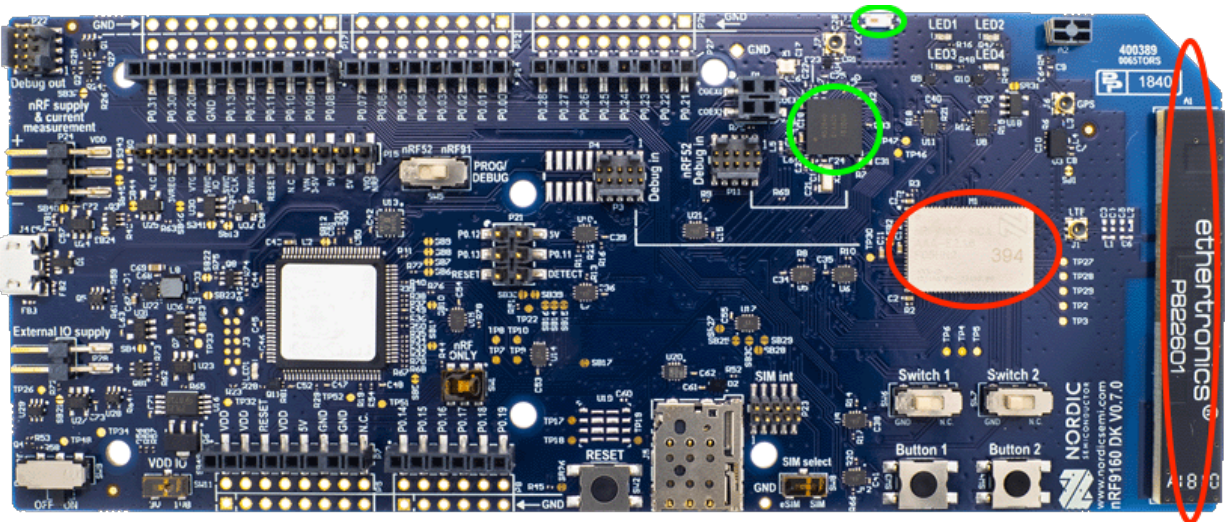


Figure 2: nRF9160DK : **nRF52840** chip with the 2.4 GHz antenna(A3) in green and the **nRF9160** chip with the LTE-M antenna(A1) in red.

⁴https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v15.0.0%2Fble_sdk_app_beacon.html

⁵<https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF9160-DK>

⁶For application part : Cortex M33

The nRF9160DK carry also some LEDs and buttons to help with the development. In this project, the LEDs are used to show the state of the running application.

3 Development environment

3.1 Programming language

The programming language is C on the nRF9160DK, typescript⁷ in the web page and InfluxQL⁸ for the database queries.

3.2 Nordic environment

The environment used to develop on this board is the *nRF connect SDK*⁹(Software Development Kit v1.3.0). This SDK provides all the necessary tools for programming on the nRF9160DK. For example, it includes many libraries and APIs that allow:

- to manage LTE-M connection and modem.
- to manage BLE (scanning device, service discovering, ...).
- to configure and abstract pins, buttons, and LEDs.

It also includes a lot of examples that can directly be loaded on the board, as well as the tiny embedded OS *Zephyr RTOS*(V2.1.99), which contains many libraries to manage sockets, MQTT or BLE for example. Furthermore, this SDK contains tools to manage all the things related to the modem, like the modem firmware, and has the possibility to include certificates (for secure connection) directly on the board.

The environment from Nordic is very convenient. There is a web platform, Nordic Devzone¹⁰, with many tutorials to get started or to learn how to use the provided APIs. On this platform, Nordic staff and other Nordic users are available to give answers to issues that people have. Often, issues have already been encountered by someone who found and shared a solution. For example, questions about Zephyr RTOS have many responses on this web platform.

In addition to this, Nordic and Zephyr provide a well-supplied documentation¹¹ with references to all libraries and APIs, configuration options and other tools.

⁷<https://www.typescriptlang.org>

⁸https://docs.influxdata.com/influxdb/v1.8/query_language/spec/

⁹<https://www.nordicsemi.com/Software-and-Tools/Software/nRF-Connect-SDK>

¹⁰<https://devzone.nordicsemi.com>

¹¹https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/index.html

3.3 Web and database environment

To develop the web server, the *WebStorm IDE* ¹²(V11.0.7) has been used. This IDE allows to manage HTML, CSS and JavaScript files in one project. It also includes an integrated web server.

Concerning the time series database, *InfluxDB* ¹³(V1.8.1) is used. This database provides some API to query and insert data. In addition to this, the *Grafana* ¹⁴(V7.1.3) tool allow to visualize data from *InfluxDB*. This tool allows to define queries to display data in a graph and to export this graph in another web-page.

4 Global development

4.1 Deployment

According to the goals, the provided material and the development environment, the following deployment diagram can be made :

¹²<https://www.jetbrains.com/webstorm/>

¹³<https://www.influxdata.com/products/influxdb-overview/>

¹⁴<https://grafana.com>

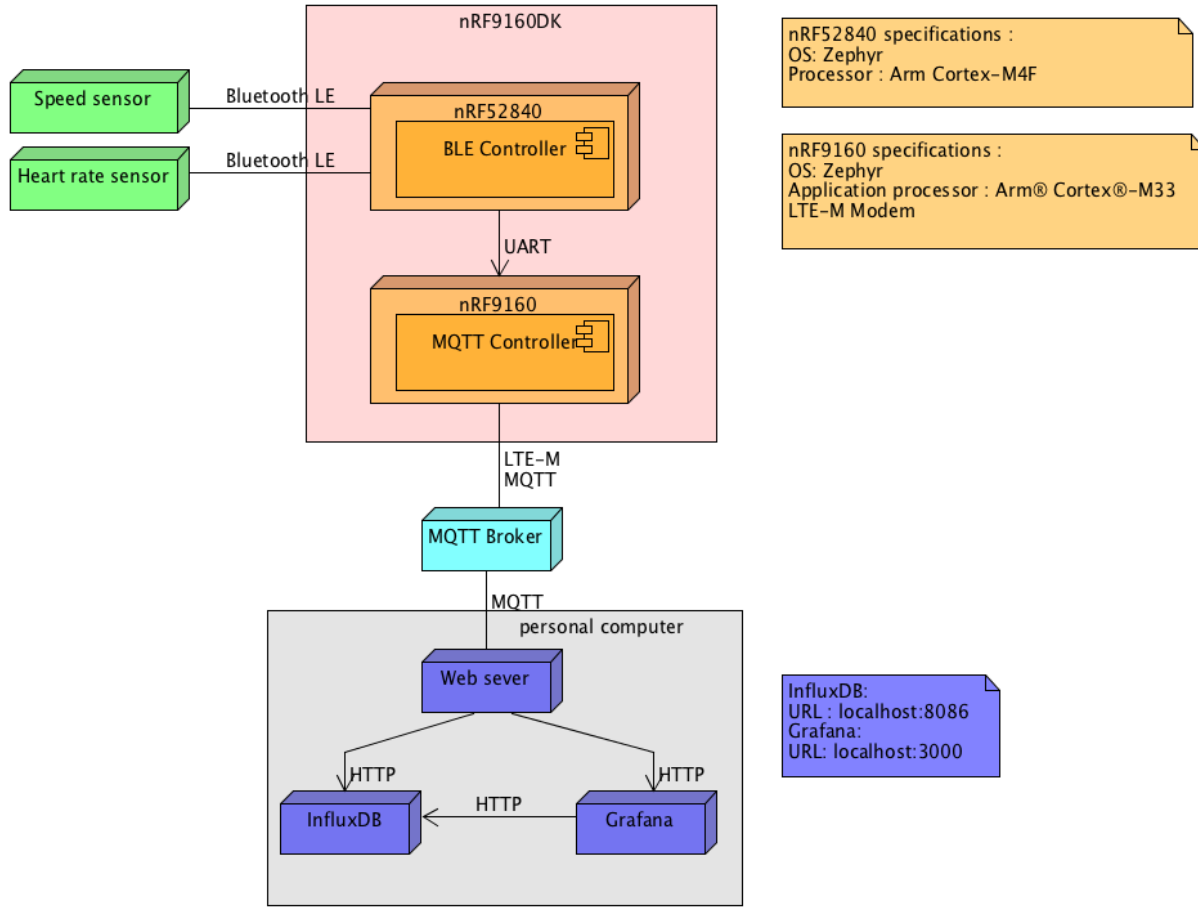


Figure 3: Deployment diagram.

The information that one still needs to specify is the way the *nRF52840* and the *nRF9160* chips communicate together. This is resolved by routing some pins from the *nRF52840* to the *nRF9160* and by using UART¹⁵. Another element that is how the web server communicates with the *InfluxDB*. *InfluxDB* provides an HTTP API to query and insert data in the database with HTTP POST and GET.

4.2 Implementation overview

In order to realize the implementation on the gateway, two main parts are needed : one for the BLE management and one for the MQTT management. This leads to the class *BLE_controller* and *MQTT_controller*.

In summary, the class *BLE_controller* manages scanning, connection and measurement notifications from the BLE devices (sensors, beacons). The *MQTT_controller* manages the MQTT connection to the broker and the sending of data. Previously, the LTE-M connection must be done from the *main function*.

¹⁵Further information comes in the BLE part section

In addition to the *BLE_controller* class, some tiny classes are developed to handle addresses and data from the BLE devices. These classes are *BT_device* and *Data_buffer*. To handle the specific data from the *Cycling Speed and Cadence service*(CSCS), the *CSC_data* was created¹⁶.

The application has been developed in a state machine way. Therefore, the XF package is needed. This package manages the event between two states and the progression of the state machine.

In the web side, a class is created, which manages the collection of data from the MQTT broker and the insertion of these data in the database. This class is called *Data_manager*.

All these elements are described in the following sections and can be seen in Figures 4 and 5 . The complete class diagrams can be found in annex 12.4.1.

The rest of this document describes each part of the implementation in more details. It will begin to describe the first step to get connected to the cellular network and to initialize the system. Then, it will explain how the MQTT code works and how the data sent through MQTT are collected (BLE part). Finally, details about the server side, including Grafana and InfluxDB, will be provided.

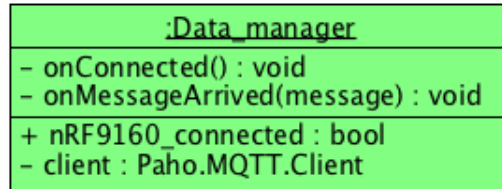


Figure 4: Class diagram of the web page.

¹⁶Further information comes in the BLE part section

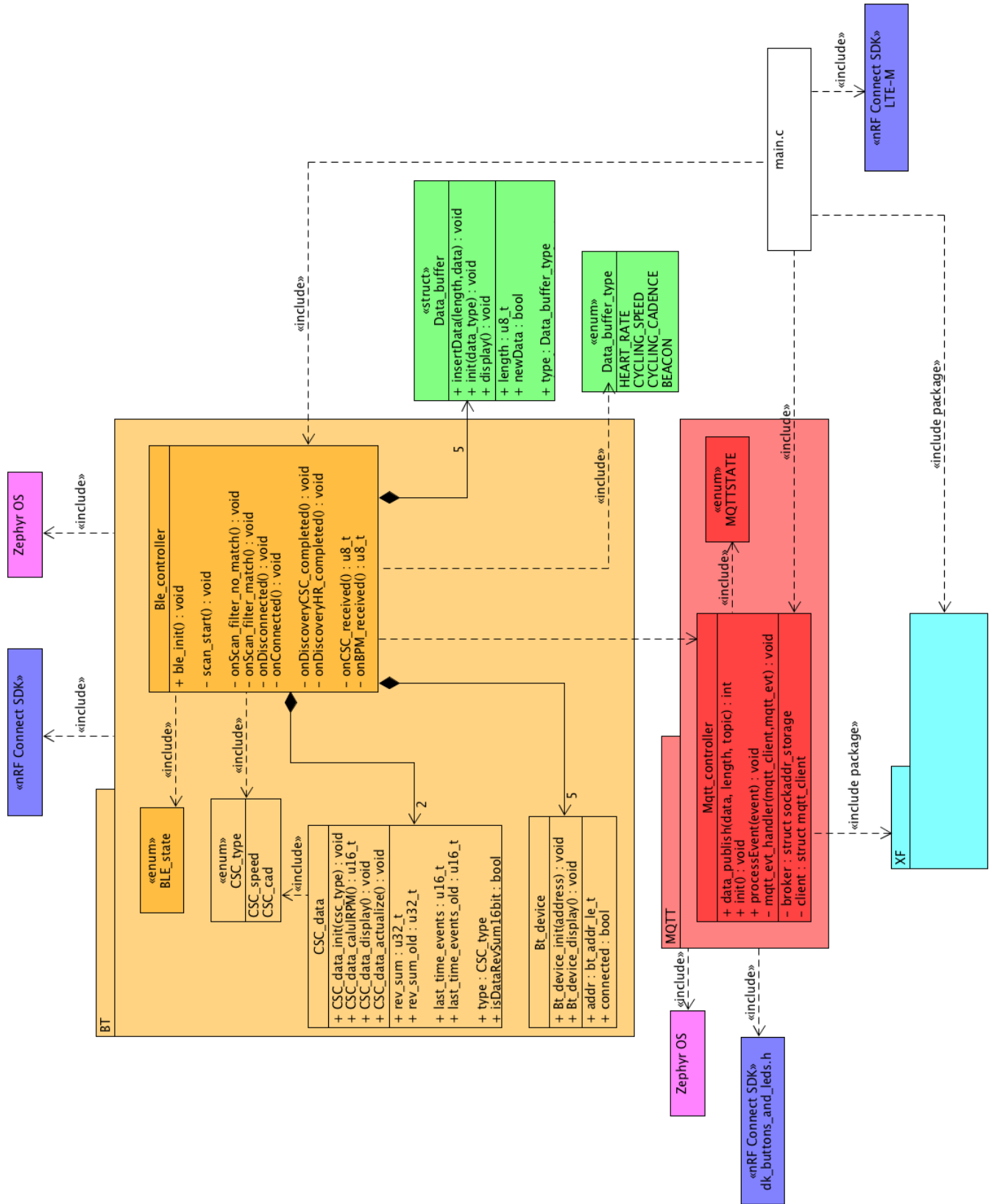


Figure 5: Class diagram of the nRF9160 implementation.

5 Gateway start

5.1 LEDs status

LEDs are used to show some information about the state of the application :

- LED1 heart rate sensor connected.
- LED2 speed sensor connected.
- LED3 cadence sensor connected.
- LED4 MQTT client connected to broker.

To turn these LEDs on or off, the *dk_buttons_and_leds* from nRF is used. First, the LED module must be initialize with *dk_leds_init()*.

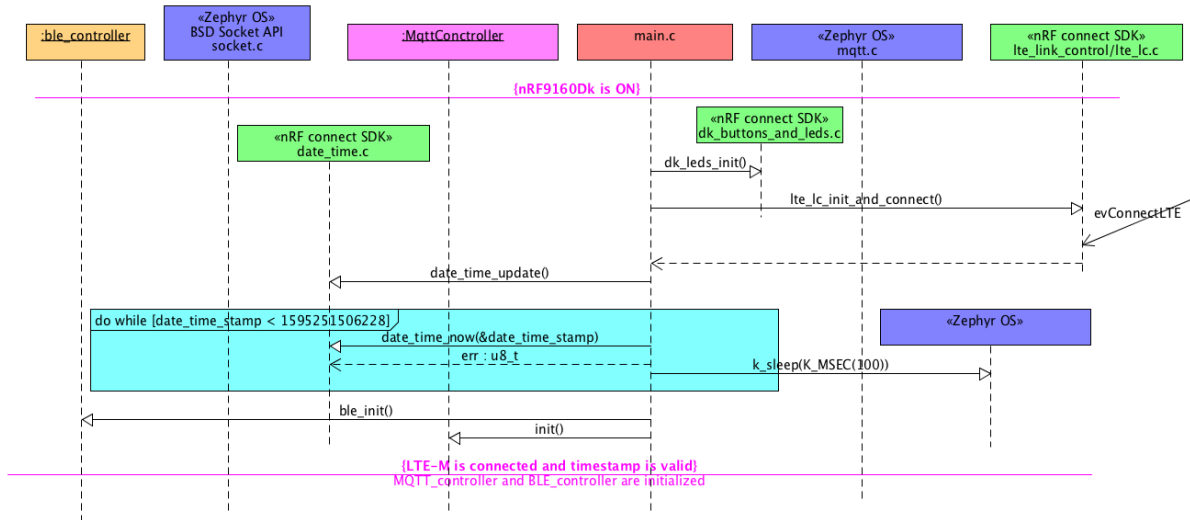


Figure 6: Start of the nRF09160 application.

5.2 LTE-M connection

There are several ways to connect to the cellular network. The one used in this project calls the blocking function *lte_lc_init_and_connect()* from the LTE link controller library from Nordic¹⁷. This function blocks until the connection to the network is set.

5.3 Getting timestamp

Timestamps are used to know at which moment data are collected from the sensors. To get timestamps, the *date-time* library¹⁸ provides some functions. This library requests time

¹⁷https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/include/modem/lte_lc.html

¹⁸https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/include/date_time.html

information first from the cellular network and then, if it has no time response, from a list of *Network Time Protocol server (NTP)*.

The first function used is the `date_time_update()` in order to request a first timestamp asynchronously. This function is needed because the date-time module requests the time information periodically each `CONFIG_DATE_TIME_UPDATE_INTERVAL_SECONDS`¹⁹ because no request is done before this constant is reached.

Then, a loop on `date_time_now(&date_time_stamp)`, which returns the timestamp in the variable passed as argument, is made until a valid value is obtained. Non-valid values are the time elapsed from the start of the gateway. Indeed, the function `date_time_now()` uses `k_uptime_get()` from Zephyr that returns the time elapsed from the start of the gateway and adds this to the time delivered from the network. This, as long as no time is obtained from the network, the `date_time_now()` function returns a non-valid timestamp.

5.3.1 Test on timestamps

Below are two examples on getting timestamps from the used library : one with the time information from the NTP and one from the cellular network.

```
Main -> Date time: 000000000000f24
, time 200
[00:00:03.663,146] .[0m<dbg> date_time.new_date_time_get: Updating date time UTC....[0m
[00:00:03.663,146] .[0m<dbg> date_time.current_time_check: Date time never set.[0m
[00:00:03.663,146] .[0m<dbg> date_time.new_date_time_get: Current time not valid.[0m
[00:00:03.663,177] .[0m<dbg> date_time.new_date_time_get: Fallback on cellular network time.[0m
[00:00:03.663,208] .[1;31m<err> date_time: Valid time not currently available, requesting time.[0m
[00:00:03.669,616] .[0m<dbg> date_time.time_modem_get: Could not get cellular network time, error: -8.[0m
[00:00:03.669,616] .[0m<dbg> date_time.new_date_time_get: Not getting cellular network time.[0m
[00:00:03.669,647] .[0m<dbg> date_time.new_date_time_get: Fallback on NTP server.[0m
[00:00:03.769,592] .[1;31m<err> date_time: Valid time not currently available, requesting time.[0m
[00:00:03.876,098] .[1;31m<err> date_time: Valid time not currently available, requesting time.[0m
[00:00:03.945,037] .[0m<dbg> date_time.time_NTP_server_get: Got time response from NTP server ntp.uio.no.[0m
[00:00:03.945,068] .[0m<dbg> date_time.new_date_time_get: Time from NTP server obtained.[0m
[00:00:03.945,068] .[0m<dbg> date_time.new_date_time_get: Updating date time UTC....[0m
Main -> Date time: 000001736c65f90d
, time 300
45,068] .[0m<dbg> date_time.new_date_time_get: Time successfully obtained.[0m
Main -> Date time: 000001736c65f981
```

Figure 7: Getting time from NTP server²⁰

```
:20.219,696] .[0m<dbg> date_time.new_date_time_get: Current time not valid.[0m
[00:00:20.219,696] .[0m<dbg> date_time.new_date_time_get: Fallback on cellular network time.[0m
[00:00:20.226,287] .[0m<dbg> date_time.time_modem_get: Response from modem: +CCLK: "20/07/20,13:25:05+08.[0m
[00:00:20.226,318] .[0m<dbg> date_time.new_date_time_get: Time from cellular network obtained.[0m
Main -> Date time: 000001736c663bcb
```

Figure 8: Getting time from cellular network

¹⁹can be found in the `kconfig` file

²⁰This screen and all similar screens show the *CoolTerm* application for MAC that allows to receive UART messages on the USB port. Thus, the displayed text is sent from the nRF9160DK `printf(text:string)` function through the MCU interface.

5.4 Classes initialization

Finally, the *BLE_controller* and *MQTT_controller* classes are initialized. In the *MQTT_controller*, this is done by fixing the first state of this class, but in the *BLE_controller* this starts a series of callback functions that will be detailed in the BLE part.

6 MQTT part

The MQTT part is managed by the *MQTT_controller* class. The class diagram can be found in annex 12.4.1.

The main tasks that the MQTT part has to do is to initialize the MQTT client, connect to the broker and publish some data to the MQTT broker. Before, the LTE-M connection must be set.

6.1 LTE-M and the nrf9160 SiP

As mentioned in the material section, the nRF9160DK contains a full LTE-M modem and supports the SIM for the network operator authentication²¹.

”LTE-M is the abbreviation for LTE Cat-M1 or Long Term Evolution (4G), category M1. This technology is for Internet of Things devices to connect directly to a 4G network, without a gateway and on batteries.”²²

In other words, this allows to connect to the cellular network, in response to the needs of IoT. Moreover, LTE-M devices have a long battery life and are cheaper than devices that use traditional LTE technology.²³ However, the data speed is smaller than traditional LTE but fit right to the IoT needs.

In the nRF9160 SiP, the application processor is the master that controls the modem. The control of the modem is done by the LTE link controller library.²⁴

6.2 About MQTT

MQTT²⁵ is a protocol based on top of the TCP/IP layer, which is used to publish, subscribe data to a specific topic on a MQTT server called MQTT broker. An example is shown in the figure below :

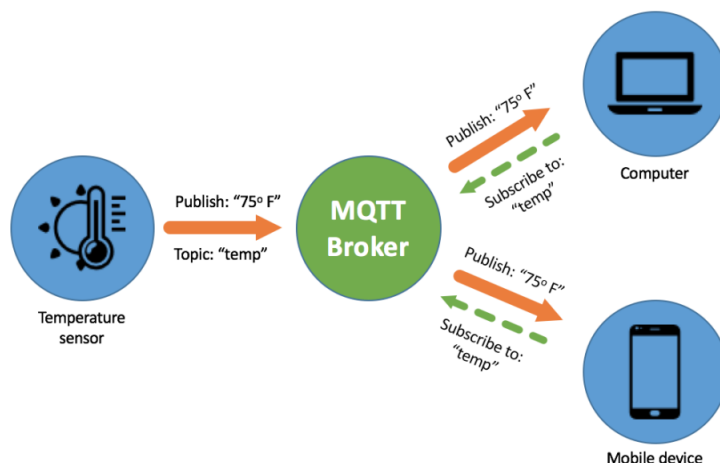
²¹<https://www.nordicsemi.com/Products/Low-power-cellular-IoT/nRF9160>

²²<https://www.link-labs.com/blog/what-is-lte-m>

²³<https://www.rfwireless-world.com/Terminology/Difference-between-LTE-and-LTE-M.html>

²⁴https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/include/modem/lte_lc.html

²⁵<http://mqtt.org>

Figure 9: MQTT example ²⁶.

In Zephyr RTOS, the MQTT client library is built on top of the BSD socket²⁷, which represents the connection between two applications.

Due to the light weight of the MQTT messaging transport, it is ideal for the IoT communication. In the IoT gateway bike, this amounts to publish the collected data from the sensors to their dedicated topic. Here is the list of topics :

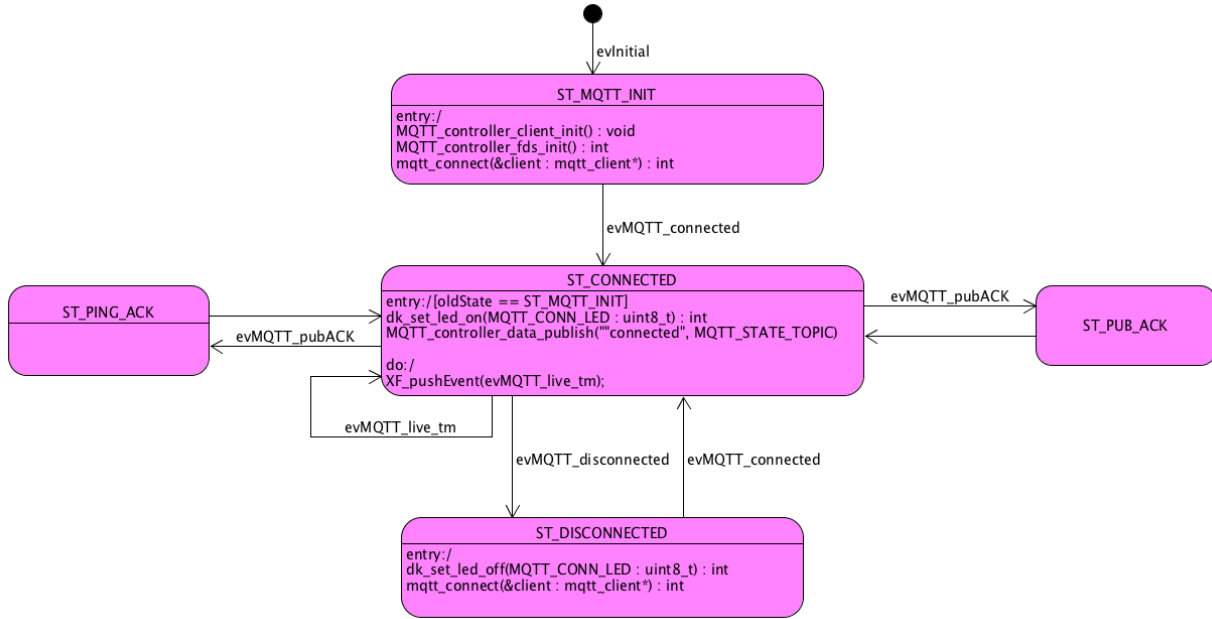
- 9160GW4SC/sensors/heartRate.
- 9160GW4SC/sensors/speed.
- 9160GW4SC/sensors/cadence.
- 9160GW4SC/beacon.
- 9160GW4SC/nRF9160State.

6.3 MQTT state machine

The figure below represents the implemented MQTT state machine :

²⁶https://8bitwork.com/wp-content/uploads/2019/03/MQTT_1-1024x604.png

²⁷http://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/reference/networking/mqtt.html

Figure 10: MQTT state machine²⁸

6.3.1 ST_MQTT_INIT

To use MQTT, after the LTE-M connection is set, one needs to initiate the client, the broker and the file descriptor.

The **client** is initialized in the the following way :

1. Set the callback function when Zephyr MQTT events happen to the client.
2. Set an unique ID to the client.
3. Set the "will" message configuration (message sent when a disconnection happens).
4. Set buffer to send and receive messages.
5. Set the MQTT transport type (here : non-secure).

One of the initialization processes for the **broker** is done by resolving the broker hostname and by setting ip type to IPV4.

File descriptor is a small integer that represents the socket. This is used by the blocking `poll()` (see subsection : Application start and MQTT init) function that waits for the corresponding file descriptor to arrive. This happens when a new message arrives from the broker. The configuration is done by setting a number to the file descriptor.

6.3.2 ST_CONNECTED

When this state comes for the first time, the first action is to publish the connected status of the gateway to the broker. Then, in order to keep the connection alive, the gateway sends periodically PING requests to the broker (sending data and wait for the echo). To re-enter in this state periodically and send PING requests, an event (`evMQTT_live_tm`) is emitted periodically.

Note that in all states, except for `ST_disconnect` and `ST_MQTT_init`, the `data_publish(message,topic)` can be called to send data from the broker.

6.3.3 ST_DISCONNECTED

When the broker disconnects or the connection is lost for any reason, the state machine comes in this state. The purpose of this state is to retry a new connection.

6.3.4 ST_PING_ACK and ST_PUB_ACK

The state machine goes in these states when a message is sent with a QoS 1 (at least once). In this project, it happens with Ping requests and with connection requests. Indeed, all the data collected from the sensors are published with QoS 0 (at most once).

6.4 Application start and MQTT init

This section is there to explain in more details how the XF package is used and how the incoming messages are managed. All the presented elements can be seen in Figure 11.

6.4.1 The XF

The XF is a tool that allows to execute a state machine. The main components are the events, the event queue and the event dispatcher. Indeed, events can be created with a unique id and are then inserted in the event queue with the `XF_pushEvent(event)` function. Then, the XF dispatches the events to a dedicated state machine. In order to dispatch the events, the XF must run. This is done with `XF_run()`, which is actually a forever loop checking if an event is in the event queue. When an event is in the event queue, it is dispatched to the corresponding state machine. The state machine is set to the event by giving a reference to the function that implements it. The state machine is named `processEvent(event)` and is implemented according to the *double switch* pattern : one *switch-case* on the current state to progress the state machine and another to do some actions.

6.5 Incoming messages from the MQTT broker

In order to wait for an incoming message, two functions must be used. The first is the `poll()` function that was explained before, which blocks until the corresponding file descriptor arrives, which happens when one waits for a MQTT message. The second function is the `mqtt_input(&client)` to manage the incoming message and to transmit the right MQTT_event to the callback function (`mqtt_evt_handler(event)`).

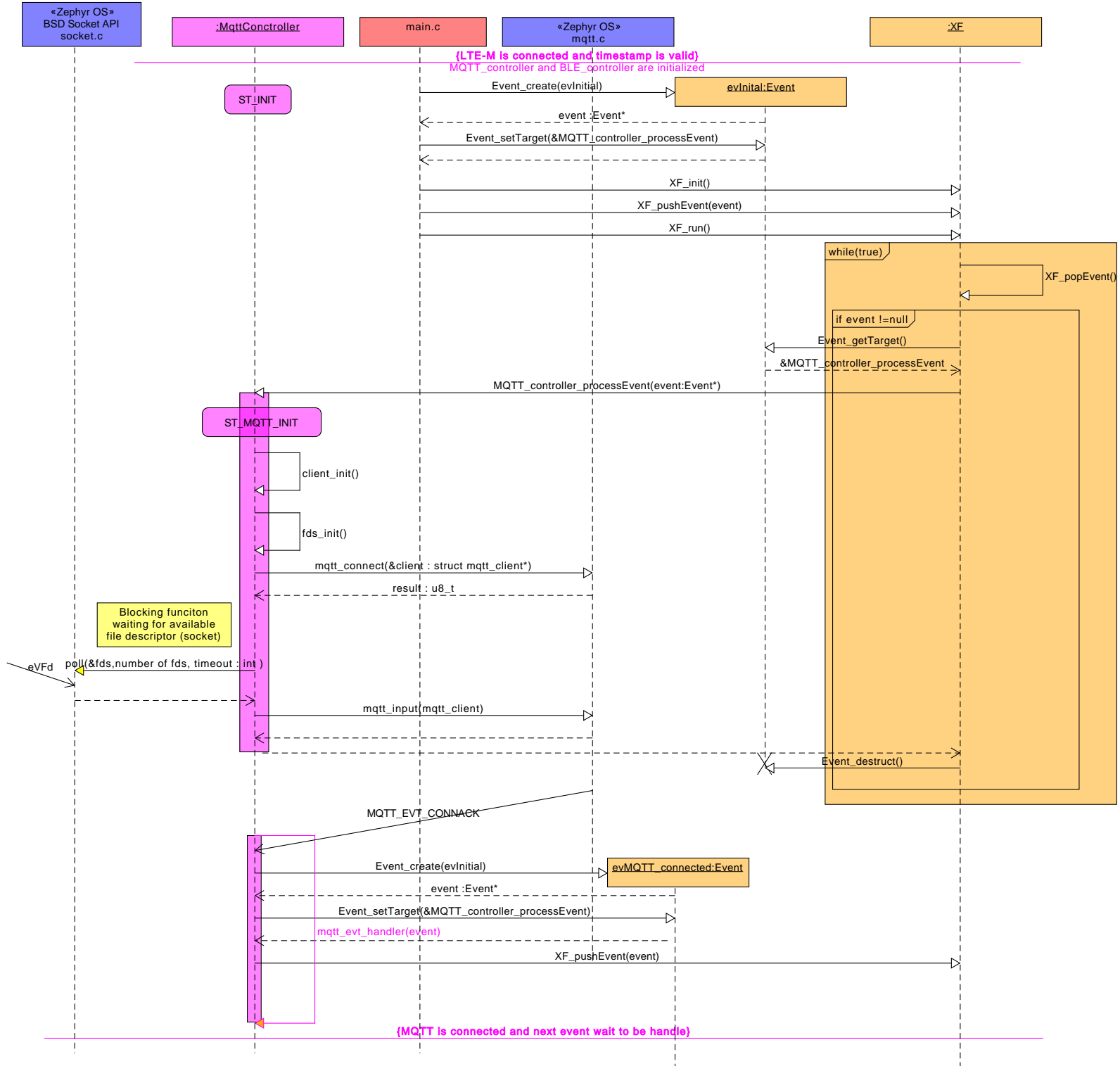


Figure 11: XF and MQTT start.

6.6 Analyse with MQTT explorer

MQTT explorer is a client able to visualize the data published on the topics.²⁹ With this tool, the topics and the published messages can be controlled.

When the data have been collected from the sensors, they are sent to the broker on a specific topic. The specific topics used in this project can be seen in the figure below:

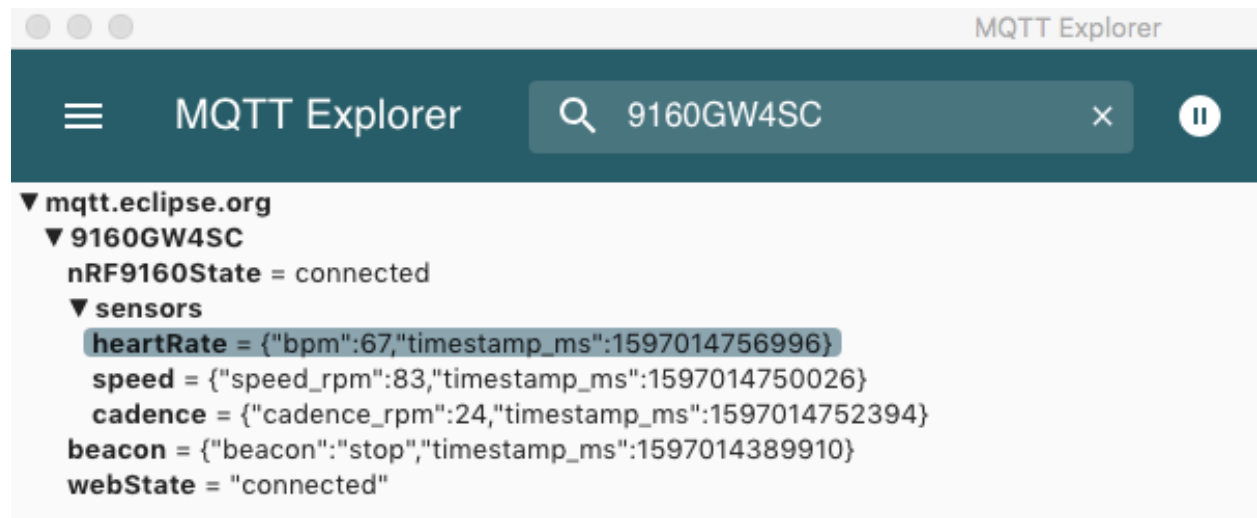


Figure 12: Topics on the broker. The format of the data can also be seen. It is made up from a timestamp and a specific value.

6.7 MQTT tests

6.7.1 MQTT state machine test

Goal This test is done to validate the correct progression of the MQTT state machine. The connection is also tested.

Initial condition To run this test, only the *MQTT_controller* class and the XF package have been used.

²⁹<http://mqtt-explorer.com>

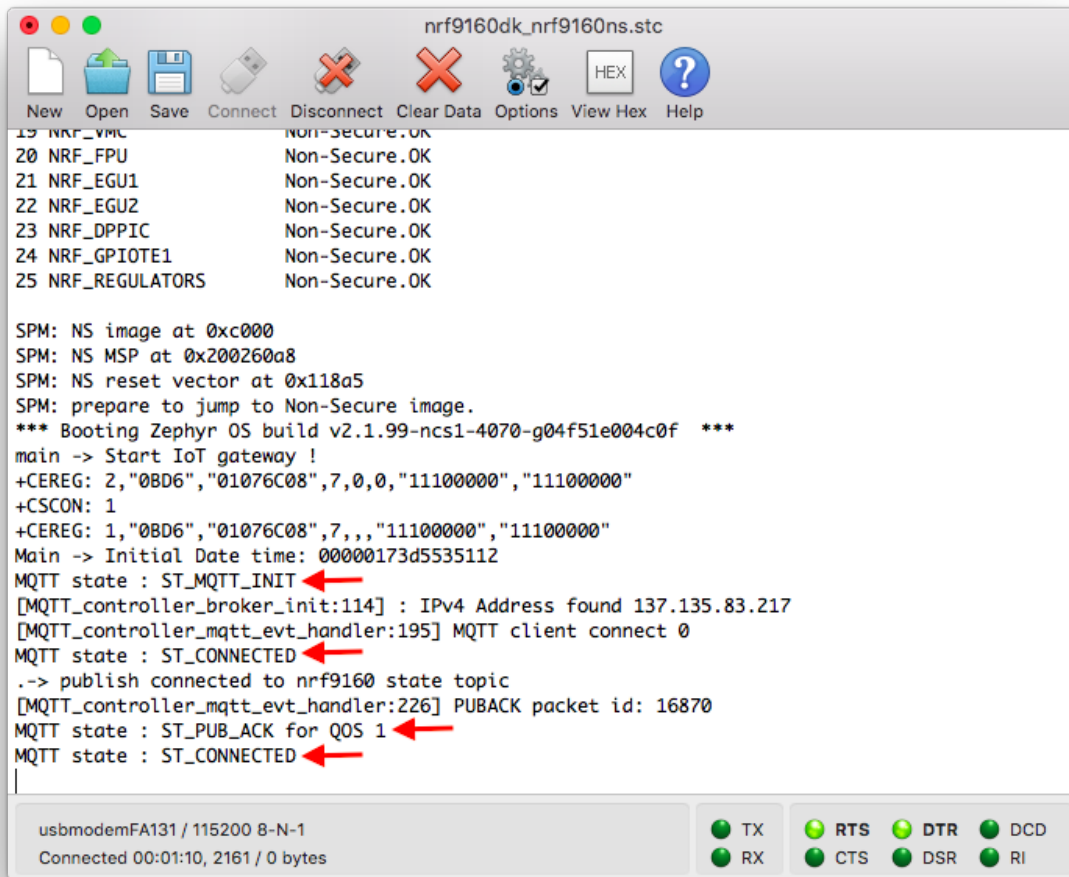


Figure 13: Progression of the MQTT state machine.

Results The state machine progresses in a correct way. The connection is done and the confirmation message is well received.

6.7.2 Test reconnection to the MQTT server

Goal This was not a test, but it can demonstrate the behavior of an unexpected disconnection.

Initial condition The system was running.

```

MQTT state : 1
MQTT state : ST_MQTT_INIT
BLE_controller -> Bluetooth ready
BLE_controller -> scan_start()
BLE_controller -> Scanning for heart rate sensor.
Bt_device_display -> ADDR : ec:68:be:8a:cc:f7 (random)
IPv4 Address found 35.156.251.58
MQTT state : 2
MQTT state : ST_CONNECTED
.-> publish connected to nrf9160 state topic
BLE_controller -> Device found: ec:68:be:8a:cc:f7 (random), RSSI : -46
BLE_controller -> HR Connected: ec:68:be:8a:cc:f7 (random)
BLE_controller -> HRS : Service discovery completed
BLE_controller -> scan_start()
BLE_controller -> Scanning for speed sensor.
Bt_device_display -> ADDR : d9:3f:f2:d1:0b:1b (random)
[MQTT_controller_mqtt_evt_handler:238] MQTT client disconnected -128
Main -> data_publish err=-128
MQTT state : 3
MQTT state : ST_DISCONNECTED
.-> Retry new connection
+CSCON: 0
+CSCON: 1
MQTT state : 2
MQTT state : ST_CONNECTED
.-> publish connected to nrf9160 state topic

```








Figure 14: Reconnection to the broker.

Results After the loss of the connection, it reconnects successfully.

6.7.3 Test if the "will" message is well sent

Goal This test is done to confirm that the "will" message is well sent to the corresponding topic when the connection is closed on the client side.

Initial condition To run this test, only the *MQTT_controller* class and the XF package have been used. Then, the nRF9160DK has been turned off.

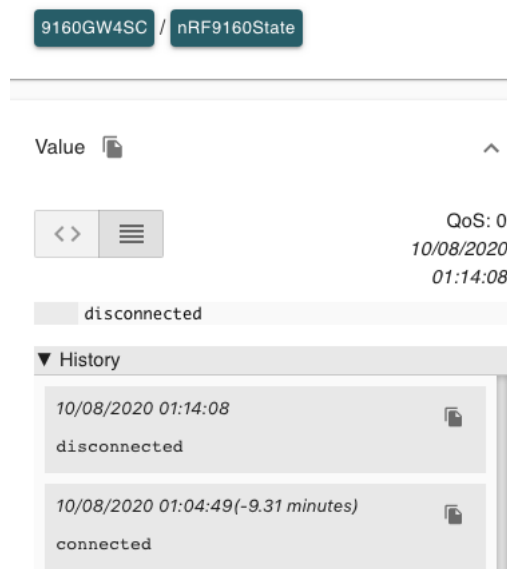


Figure 15: nRF9160 last will.

Results The last will successfully arrived to the broker.

6.7.4 Test PING response

Goal This test is done to see if the PING ACK is well received and if the event is well handled.

Initial condition To run this test, only the *MQTT_controller* class and the XF package have been used.

```
[MQTT_controller_mqtt_evt_handler:259] Ping response from server
MQTT state : ST_PING_ACK
MQTT state : ST_CONNECTED
[MQTT_controller_mqtt_evt_handler:259] Ping response from server
MQTT state : ST_PING_ACK
MQTT state : ST_CONNECTED
```

Figure 16: PING response.

Results The last will successfully arrived to the broker.

7 BLE part

The BLE part is managed by the *BLE_controller* class. A class diagram can be found in annex 40.

The tasks that the BLE part must do :

- Initialize the *BLE_controller* class.

- Scan for sensors on the bike and on the cyclist (speed, cadence and heart rate).
- Discover the BLE services that are included in the BLE devices.
- Subscribe to the measurements' characteristic included in the wanted service (Heart rate service, Cycling speed and cadence service).
- Send the collected data to the broker with the `data_publish(message,topic)` from the `MQTT_controller` class.
- Scan for the beacon start and stop delimiting the segment.

7.1 Bluetooth Low Energy

Bluetooth low energy (BLE) is a wireless technology. It is optimized for low power consumption. It allows to transmit data from a device to another, like transmitting the heart rate from a sensor to another device that collects these data. The BLE architecture consists of several layers(Figure 17).

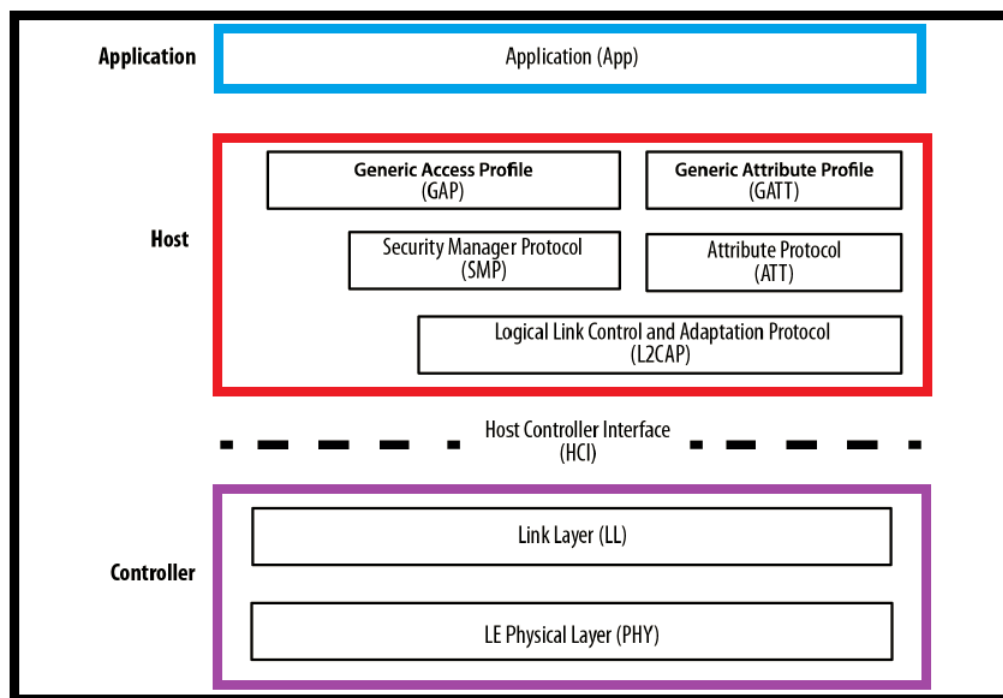


Figure 17: BLE protocol stack ³⁰.

Next sections describe some of these layer.

³⁰<https://i2.wp.com/www.embetronicx.com/wp-content/uploads/2017/07/bluetooth-low-energy-ble-protocol-stack.png>

7.1.1 GAP Layer

The definition of the GAP layer is the following :

”GAP is an acronym for the Generic Access Profile, and it controls connections and advertising in Bluetooth. GAP is what makes your device visible to the outside world, and determines how two devices can (or can’t) interact with each other.” ³¹

BLE devices can have two roles : central or peripheral. In this project, the nRF9160DK has the central role and the sensors and the beacons the peripheral one. This means that the sensors and the beacons are advertising, i.e. broadcasting information about themselves, and that the nRF9160 is scanning for devices. When the wanted device is scanned, the nRF9160DK can connect to it by fixing the connection parameters and by making a connection request. It is noteworthy to mention that it is not possible to connect to a beacon.

7.1.2 GATT Layer

The GATT³² layer definition is the following :

”GATT is an acronym for the Generic Attribute Profile, and it defines the way that two Bluetooth Low Energy devices transfer data back and forth using concepts called Services and Characteristics. It makes use of a generic data protocol called the Attribute Protocol (ATT), which is used to store Services, Characteristics and related data in a simple lookup table using 16-bit IDs for each entry in the table.” ³³

Thus, from the GATT point of view, a device has a server role, i.e. it holds the data, and another device has the client role that wants to access these data. To hold these data, services are used :

”Services are used to break data up into logic entities, and contain specific chunks of data called characteristics. A service can have one or more characteristics, and each service distinguishes itself from other services by means of a unique numeric ID called a UUID,[...].” ³⁴

Thus, characteristics contain the data that the client would want to access. There is the possibility to be notified about the changes of values. This is done by writing a specific value in a characteristic field named CCCD (Client Characteristic Configuration Descriptor) to configure the characteristic.

³¹<https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gap>

³²<https://www.bluetooth.com/specifications/gatt/services/>

³³<https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt>

³⁴<https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt>

7.1.3 Cycling sensors

Heart rate service The heart rate service can be seen in Figure 18. This service contains a characteristic named *Heart rate measurement*, which contains the bpm (beats per minute). This characteristic contains other values that are not needed in this project.

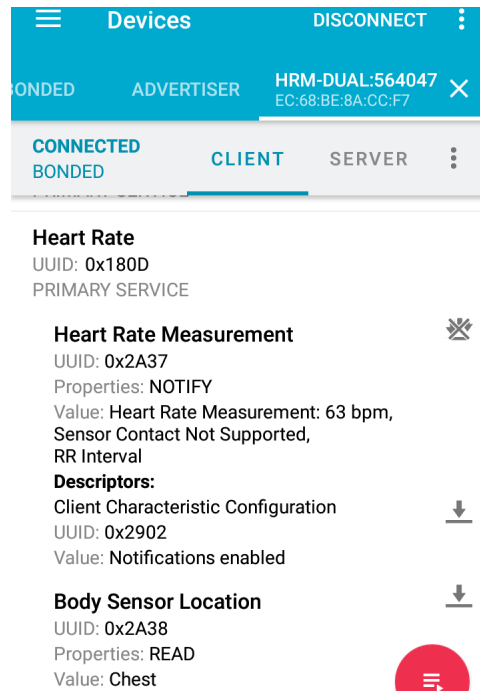


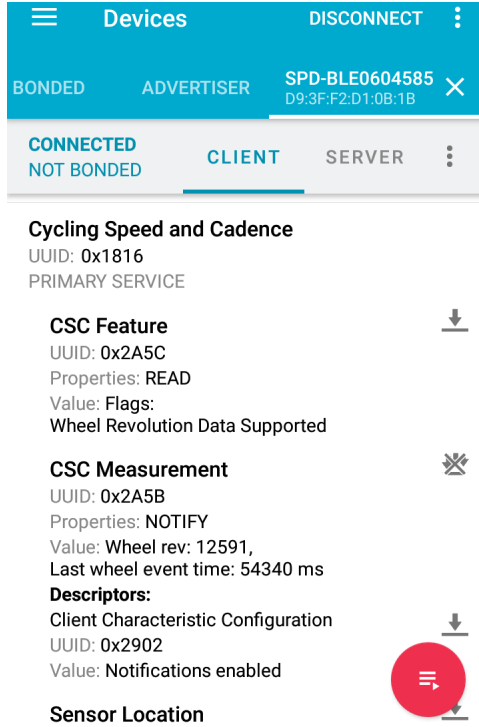
Figure 18: Heart rate service ³⁵. The CCCD can also be seen (here with enabled notification).

Cycling speed and cadence service The cycling speed and cadence service contains a characteristic named *CSC measurement*. This characteristic can contain :

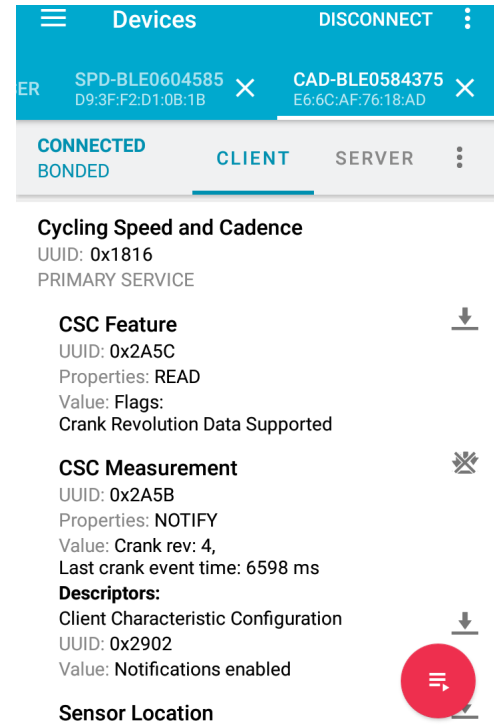
- only data related to the speed sensor.
- only data related to the cadence sensor.
- data from the speed and the cadence sensor.

In this project, the Garmin sensors contain specific data. The specific data are the number of revolutions (for the wheels or for the cadence) and the last time at which a new value is measured (the resolution is defined as second/1024). Thus, with this, the revolution per minute can be calculated. Additionally, for the speed sensor, the speed can be calculated with a wheel diameter. **In this project, the speed is calculated on the web page.**

³⁵The mobile application used to discover and connect to the BLE device is *nRF connect for mobile*.



(a) CSC speed service.



(b) CSC crank service.

7.2 HCI

7.2.1 Definitions

The BLE protocol consists of two main parts : the *Host* part and the *controller* part (Figure 17).

The host part manages the upper layers (GAP, GATT, ...) and it is generally controlled by the application layer.

The controller part manages the lower layers (Physical layer, ...) and it is linked to an antenna, since this part manages the sending and the reception of the BLE frame.

The two parts of the stack protocol communicate through the HCI (Host Controller Interface), which can be implemented with many physical transports (SPI, UART, ...).

7.2.2 On the nRF9160DK

For this project, the controller part is loaded into the nRF51840 SoC, which controls the 2.4GHz antenna. Then, it is exposed to the nRF9160 SiP over UART. This is done with a provided sample from Zephyr : *hci_uart*³⁶. This sample can be found to the Zephyr tree in : `samples/bluetooth/hci_uart`.

³⁶https://docs.zephyrproject.org/latest/samples/bluetooth/hci_uart/README.html

To link the two chips, some pins from the nRF52840 must be linked to the nRF9160. This is done by routing several pins with the board controller (which is the nRF51840 on the gateway).

This means that the application layer could be in the nRF9160 SiP, able to manage the BLE. The advantages are : all the applications of the gateway will be in the same chip and it is not needed to define a way to communicate between the two chips.

7.3 BLE state diagram

The *BLE_controller* class is implemented to :

- connect to the sensors speed, cadence, heart rate.
- subscribe for the value notification.
- send the collected data to the MQTT broker.
- wait to reach the beacons.

All these steps result in a state machine. The Figure 20 represents the different states of the *BLE_controller* class :

Figure 20: *BLE_controller* state machine.

The following sections describe the action performed in the states. Then, more details regarding the main part of the implementation will be given with sequences and activity diagrams. Note that, unlike the *MQTT_controller* class, this step machine is not run with the *XF package*. The states progress through the code depending on events received from the bluetooth callback functions.

7.3.1 Initialization state

The initialization state, *ST_BLE_INIT*, initializes all attributes of the *BLE_controller* and enables the BLE. These attributes include one *Bt_device* instance for the bluetooth devices, one *Data_buffer* instance for the bluetooth devices (for more details, see complete class diagram in annex 40).

Bt_device This class contains the fixed address and the connected status of each bluetooth device. The address is fixed in the beginning to scan and connect only the specific sensors. This is also used to detect the specific beacons. The initialization sets the previously configured addresses (these addresses can be found and modified in the *bt_device.h* file).

Data_buffer This class is used to store and display the notification messages. The initialization is made by setting a type (HEART_RATE,CYCLING_SPEED,CYCLING_CADENCE,BEACON) to the instances.

When the BLE is enabled, the corresponding callback function is executed and the state machine progresses to the next state.

7.3.2 Scan state to subscribe done

When the Bluetooth is enabled, the *BLE_controller* can start scanning for sensors whose notifications are needed. Since the scan is done on a specific sensor with the address of it (filter on specific address is enabled), an order must be defined to scan the next device. It has been decided to scan, connect and subscribe to the sensors in the following order :

1. heart rate sensor.
2. speed sensor.
3. cadence sensor.

Another choice that has been made is to connect to the next sensor after subscribing to the notification of the previous one (in contrast to connect to all sensors and then subscribing to all sensors notification). This choice has been made because of the Zephyr and Nordic Bluetooth libraries. Indeed, only one scan³⁷ can be performed at one time, and only one service discovery procedure can be started at one time³⁸.

³⁷https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/include/bluetooth/scan.html

³⁸https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/include/bluetooth/gatt_dm.html

In order to subscribe to the notification, it is needed to write *notify* value in the CCCD (See GATT section 7.1.2) of the measurement characteristic (from HR or CSC service). Since this characteristic is included in a service, it is needed to discover it when the connection is done. Discovering these services is performed by the *bt_gatt_dm_start(Unique_identifier_of_the_service)* of the discovery manager API from Nordic.

According to the choices, the states to subscribe are : *ST_HR_SCANNING*, *ST_HR_CONNECTED* and *ST_HR_DISC_COMPLETED* (similar for other sensors).

7.3.3 Waiting for the beacons states

When the subscription to all the measurement characteristics is done, the gateway can start scanning for beacons. This time, the detection of the beacon is not made with a filter on a specific address. Indeed, it compares all the discovered devices around the gateway to the beacons addresses. This comparison is done in the callback function : *onScan_filter_no_match()*. When the first beacon matches, it goes to the next waiting state, looking for the next beacon (stop beacon). The states corresponding to this are : *ST_WAIT_START*, *ST_WAIT_STOP* and *ST_STOP*.

7.3.4 New value from the sensor state

When a notification message arrives, the timestamp is obtained from the *date-time library*. Then, these values (the timestamp and the measures from the sensors) are integrated in a JSON³⁹ message to be sent to the MQTT broker. The sending is done with the *data_publish(data,topic)* from *MQTT_controller class*, as it will be explained in section 7.5. The states corresponding to a new notification are : *ST_NEW_HR_VAL*, *ST_NEW_SP_VAL* and *ST_NEW_CAD_VAL*.

7.4 From the BLE initialization to the first notification in details

The following sequence diagram (Figure 21) shows with more details the process of :

- Scanning with a filter on the sensor address.
- Discovering the sensor service.
- Subscribing to the measurement characteristic notifications.

Scanning The diagram shows that previous filters are removed before a new scan is performed. This is done to be sure that the gateway only scans for the wanted device.

Connection When connected, a dedicated LED is turned on to show the state of connection of the sensor.

³⁹Javascript object notation

Subscribing The process of subscribing is shown and corresponds to the GATT section 7.1.2 as followed : discovering service, get measurement characteristic, get the descriptor and finally CCCD⁴⁰. When the CCCD is obtained, the subscription can be done.

⁴⁰Client Characteristic Configuration Descriptor

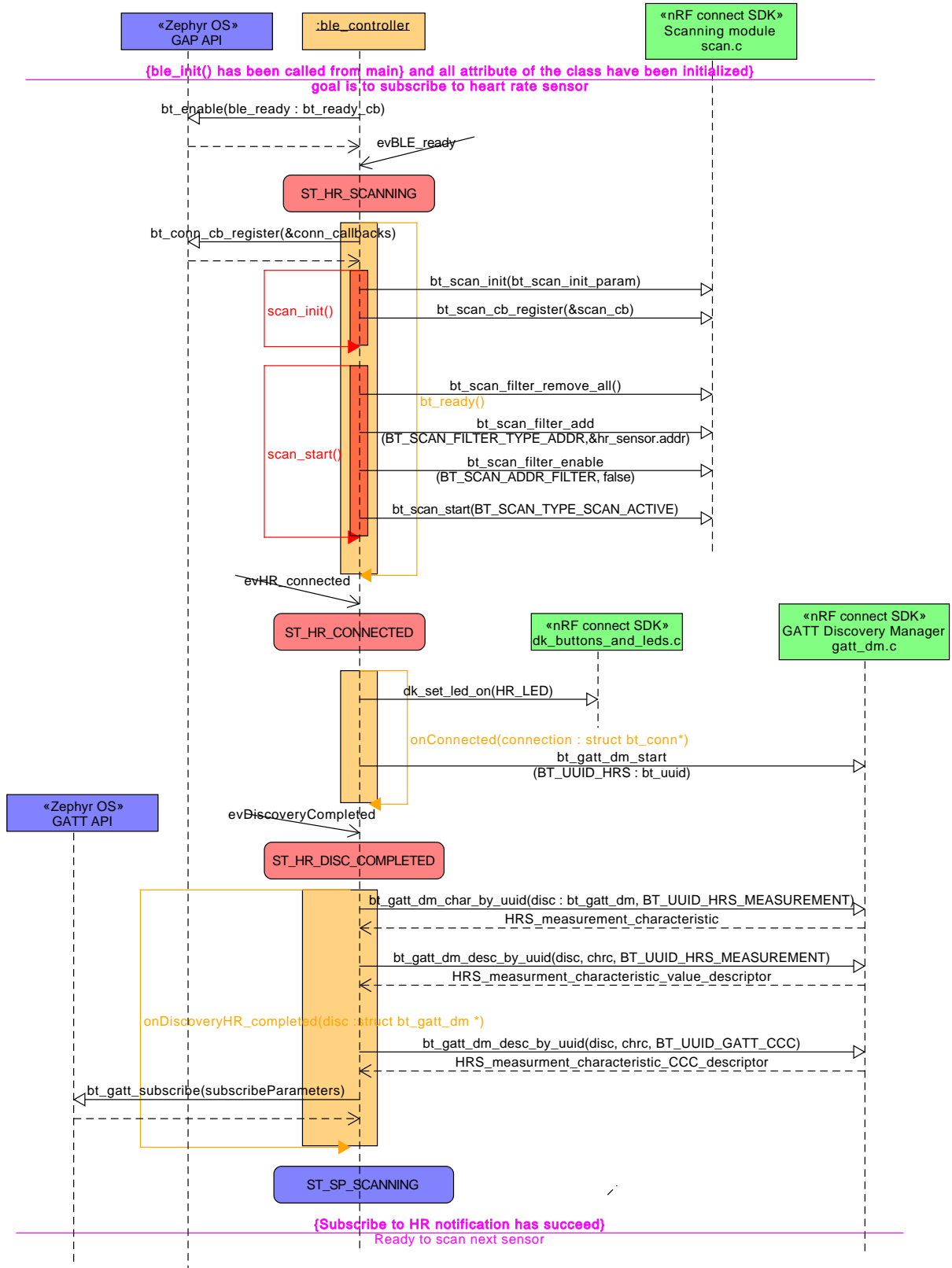


Figure 21: Subscribe to heart rate notification.

7.5 New value from CSC cadence sensor

7.5.1 CSC sensor notification

The following sequence diagram (Figure 22) shows in more details the process of :

- Getting data from the notification (in the `u8_t` data array).
- Calculate the revolution per minute with the collected data (See section 7.5.2)
- Converting the measures and the timestamps to a JSON message.
- Publishing data on the broker.
- Actualize `cadence_data:CSC_Data`

Getting data from notification Data are stored in a little endianness way. To get correct data, the function `sys_get_le_16(data*)` is called and returns the corresponding value. Additionally, the data are taken according to the CSC measurement characteristics ⁴¹, which is 16 bits for the last time event and 16 bits for the revolution sum (note that for the speed, the revolution sum is on 32 bits).

Publishing data on the broker Publishing data on the broker is done only if the `MQTT_controller` is in the `ST_CONNECTED` state. As notifications arrive each 500 ms and as the lost data are not significant, it was decided to send the data according to QoS 0.

Actualize cadence_data After data have been sent, the revolution per minute and the last time event must be stored in order to calculate the rpm on the next notification.

Note that the notifications of the new values of the other sensors (heart rate and speed) are done in the same way.

⁴¹https://www.bluetooth.com/blog/part-2-the-wheels-on-the-bike-are-bluetooth-smart-bluetooth-smart-b?utm_campaign=connected-device&utm_source=internal&utm_medium=blog&utm_content=the-wheels-on-the-bike-are-bluetooth-smart

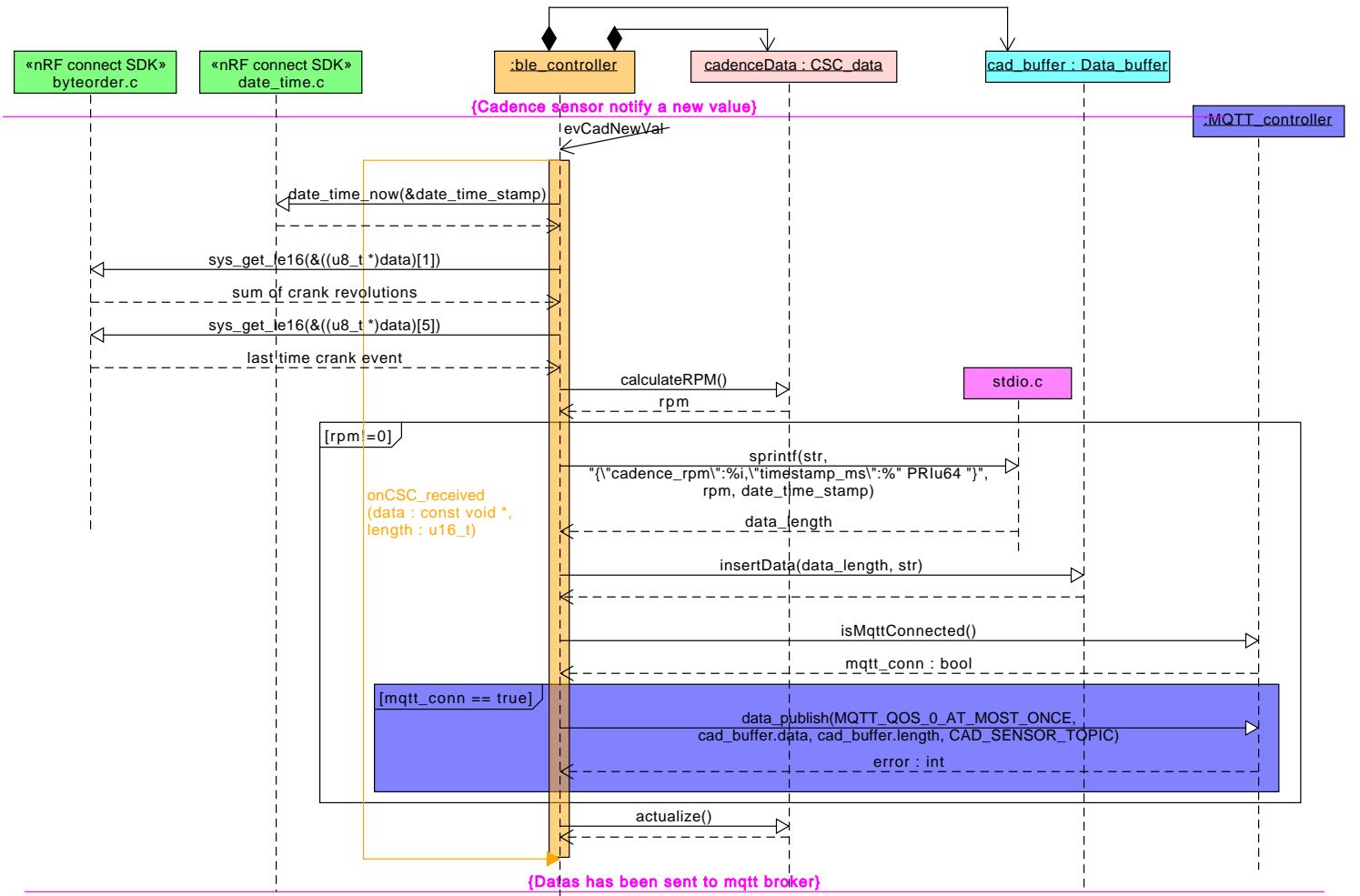


Figure 22: New value from CSC sensor.

7.5.2 Calculate rpm for CSC

The next activity diagram (Figure 23) of the `calculate_rpm()` from the `CSC_data` class shows how the revolutions per minute are calculated from the data collected from the CSC sensors. The only subtlety is when a overflow arrives on the revolution sum or the last time events: this is detected when the difference between the new and old value is negative.

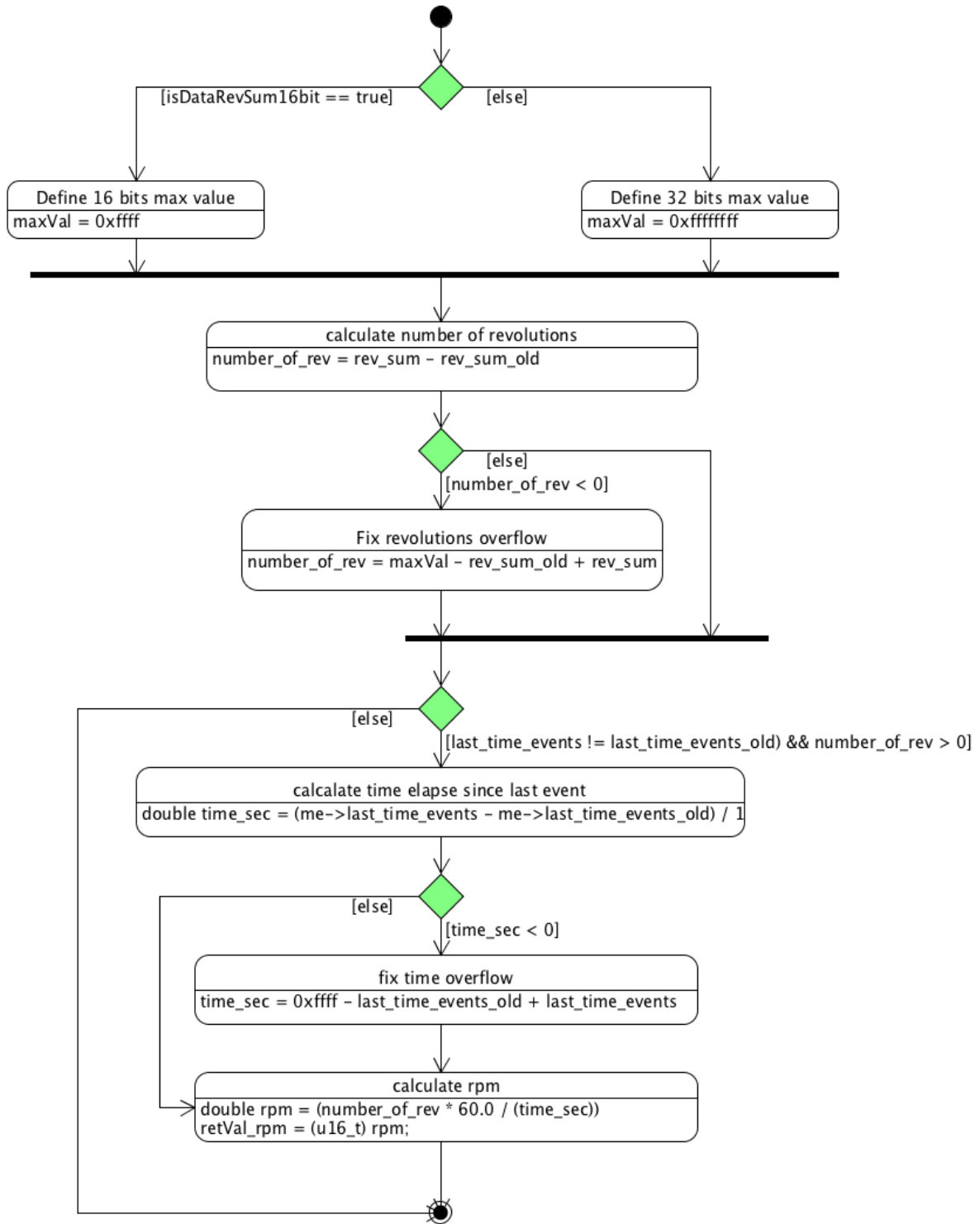


Figure 23: Calculate revolutions per minute.

7.6 BLE beacons reached

The following sequence diagram (Figure 24), which is similar to Figure 22, shows in more details the process of the detection of the beacons.

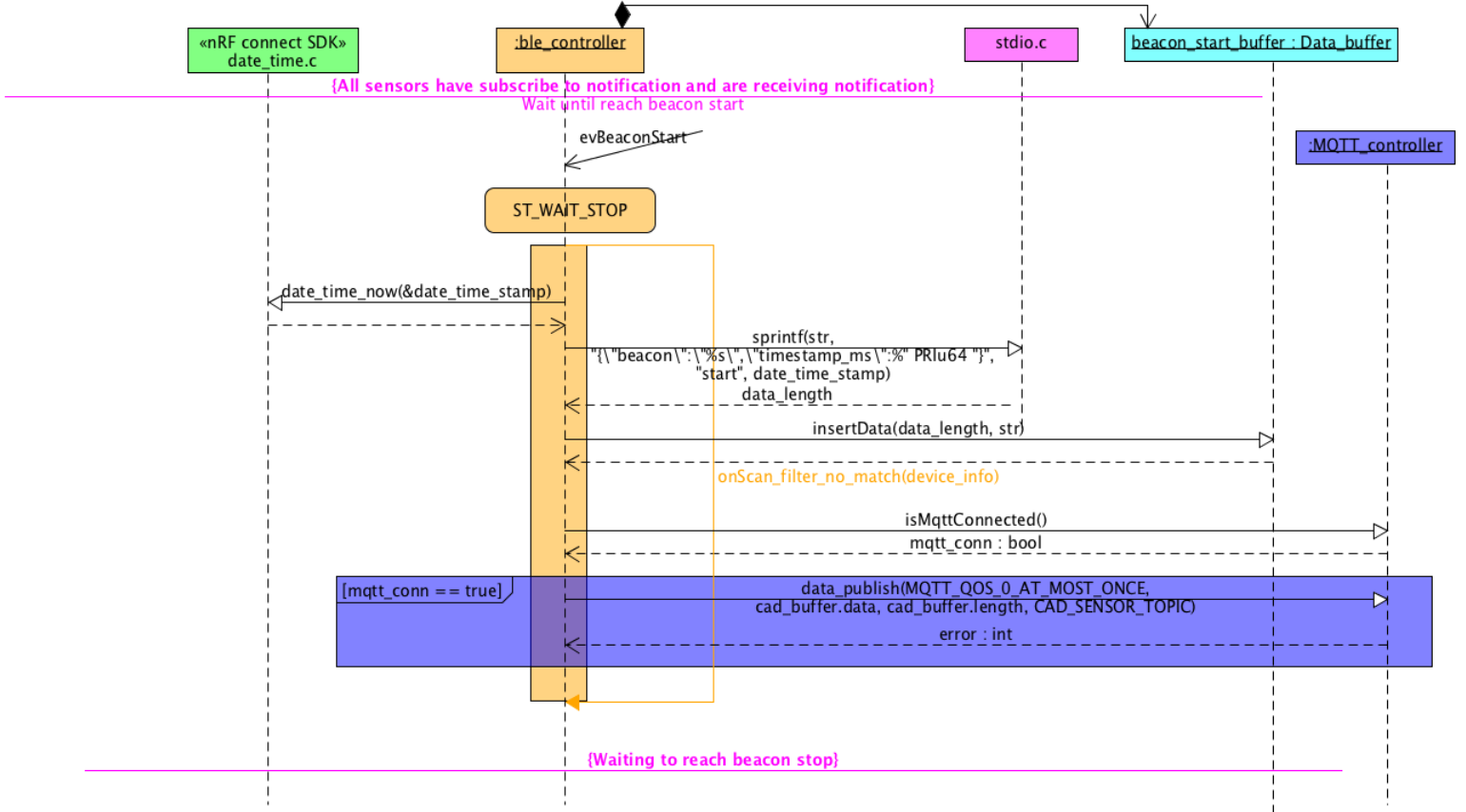


Figure 24: Beacon start reached.

7.7 BLE_controller tests

7.7.1 Connect and subscribe to sensors

Goal Connect and subscribe to all sensors, check the correct progress of the state machine.

Initial condition *ble_init()* from *BLE_controller* is called from main. LTE-M is disabled.

```

nrf9160dk_nrf9160ns.stc
New Open Save Connect Disconnect Clear Data Options View Hex Help

BLE state : ST_BLE_INIT
BLE_controller -> Initializing Bluetooth..
BLE_controller -> Init beacon start
Bt_device_init -> Init bluetooth device, ADDR : ec:2b:77:ef:93:60 (random)
BLE_controller -> Init beacon stop
Bt_device_init -> Init bluetooth device, ADDR : ea:91:d7:6e:dd:2c (random)
BLE_controller -> Init heart rate sensor
Bt_device_init -> Init bluetooth device, ADDR : ec:68:be:8a:cc:f7 (random)
BLE_controller -> Init speed sensor
Bt_device_init -> Init bluetooth device, ADDR : d9:3f:f2:d1:0b:1b (random)
BLE_controller -> Init cadence sensor
Bt_device_init -> Init bluetooth device, ADDR : e6:6c:af:76:18:ad (random)
BLE state : ST_HR_SCANNING
BLE_controller -> Bluetooth ready
BLE_controller -> scan_start()
BLE_controller -> Scanning for heart rate sensor.
Bt_device_display -> ADDR : ec:68:be:8a:cc:f7 (random)
BLE_controller -> Device found: ec:68:be:8a:cc:f7 (random), RSSI : -59
BLE_controller -> HR Connected: ec:68:be:8a:cc:f7 (random)
BLE state : ST_HR_CONNECTED.
BLE_controller -> HRS : Service discovery completed
BLE state : ST_HR_DISC_COMPLETED
BLE state : ST_SP_SCANNING
BLE_controller -> scan_start()
BLE_controller -> Scanning for speed sensor.
Bt_device_display -> ADDR : d9:3f:f2:d1:0b:1b (random)
BLE_controller -> Device found: d9:3f:f2:d1:0b:1b (random), RSSI : -58
BLE_controller -> SPEED Sensor Connected: d9:3f:f2:d1:0b:1b (random)
BLE state : ST_SP_CONNECTED
BLE_controller -> CSC : Service discovery completed
BLE state : ST_SP_DISC_COMPLETED
BLE state : ST_CAD_SCANNING
BLE_controller -> scan_start()
BLE_controller -> Scanning for cadence sensor.
Bt_device_display -> ADDR : e6:6c:af:76:18:ad (random)
BLE_controller -> Device found: e6:6c:af:76:18:ad (random), RSSI : -60
BLE_controller -> Cadence Sensor Connected: e6:6c:af:76:18:ad (random)
BLE state : ST_CAD_CONNECTED
BLE_controller -> CSC : Service discovery completed
BLE state : ST_CAD_DISC_COMPLETED
BLE state : ST_SCAN_END
BBLE state : ST_WAIT_START
BLE_controller -> scan_start()
BLE_controller -> Scanning for beacon
BLE_controller -> CSC-Cadence : New value.
----- Data buffer -----

```

Figure 25: Progression of the MQTT state machine.

Results The *BLE_controller* state machine progresses in the correct state depending on the received events. All the sensors have been connected and subscribed to the respective notification. The first value is received from the CSC cadence sensor.

7.7.2 Receive notification from sensor

Goal Verify that the CSC sensor notification is well received from the gateway.

Initial condition All the sensors have subscribed to the notification. Only the CSC cadence notification display is enabled.

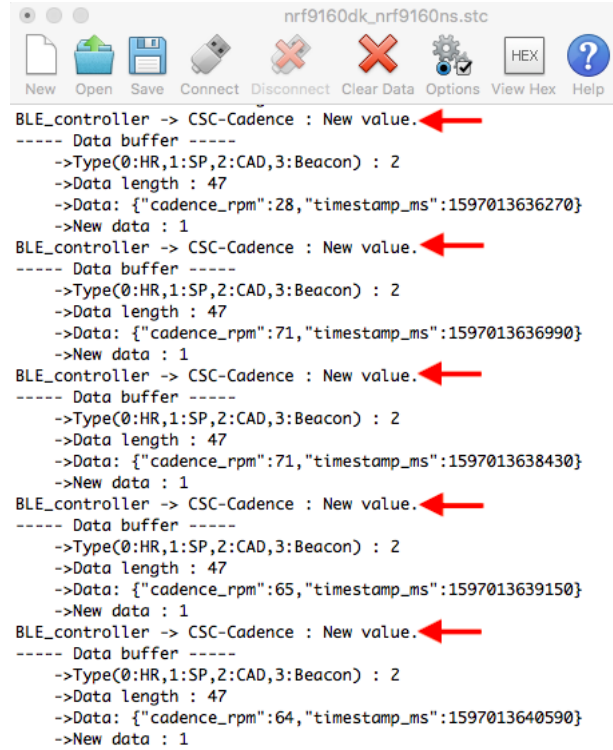


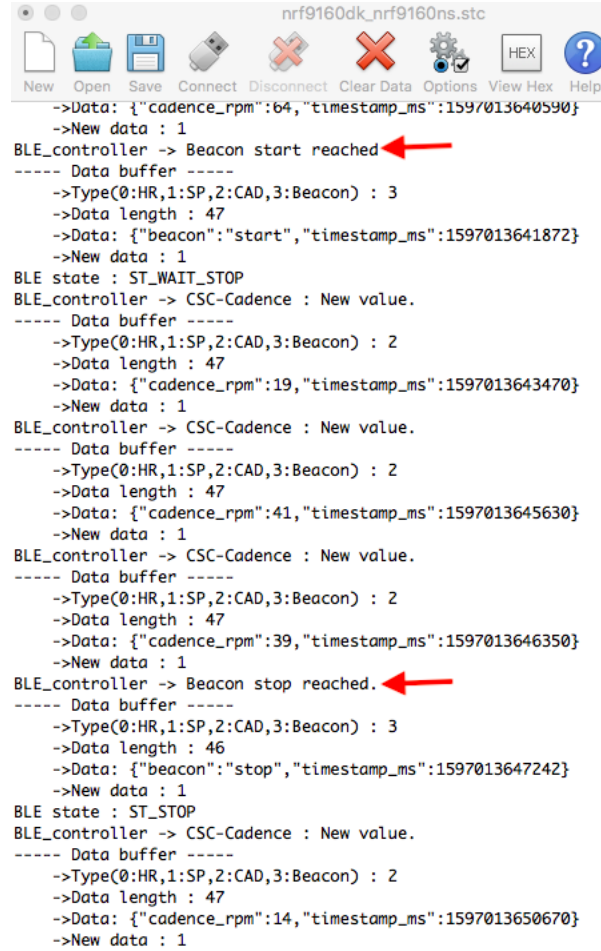
Figure 26: Notification from the CSC cadence sensor.

Results The gateway receives all the notifications from the CSC cadence sensor. The notifications from the other sensors are also received but are not visible on the debug screen.

7.7.3 Detect beacons

Goal Verify that the BLE beacons are detected.

Initial condition All the sensors have subscribed to the notification and the state is ST_WAIT_START.



```

nrf9160dk_nrf9160ns.stc
New Open Save Connect Disconnect Clear Data Options View Hex Help
->Data: {"cadence_rpm":64,"timestamp_ms":1597013640590}
->New data : 1
BLE_controller -> Beacon start reached
----- Data buffer -----
->Type(0:HR,1:SP,2:CAD,3:Beacon) : 3
->Data length : 47
->Data: {"beacon":"start","timestamp_ms":1597013641872}
->New data : 1
BLE state : ST_WAIT_STOP
BLE_controller -> CSC-Cadence : New value.
----- Data buffer -----
->Type(0:HR,1:SP,2:CAD,3:Beacon) : 2
->Data length : 47
->Data: {"cadence_rpm":19,"timestamp_ms":1597013643470}
->New data : 1
BLE_controller -> CSC-Cadence : New value.
----- Data buffer -----
->Type(0:HR,1:SP,2:CAD,3:Beacon) : 2
->Data length : 47
->Data: {"cadence_rpm":41,"timestamp_ms":1597013645630}
->New data : 1
BLE_controller -> CSC-Cadence : New value.
----- Data buffer -----
->Type(0:HR,1:SP,2:CAD,3:Beacon) : 2
->Data length : 47
->Data: {"cadence_rpm":39,"timestamp_ms":1597013646350}
->New data : 1
BLE_controller -> Beacon stop reached.
----- Data buffer -----
->Type(0:HR,1:SP,2:CAD,3:Beacon) : 3
->Data length : 46
->Data: {"beacon":"stop","timestamp_ms":1597013647242}
->New data : 1
BLE state : ST_STOP
BLE_controller -> CSC-Cadence : New value.
----- Data buffer -----
->Type(0:HR,1:SP,2:CAD,3:Beacon) : 2
->Data length : 47
->Data: {"cadence_rpm":14,"timestamp_ms":1597013650670}
->New data : 1

```

Figure 27: Beacons start and stop reached.

Results Beacons are detected in the correct order.

8 BLE and MQTT : global behavior

This section demonstrates the correct behavior of the nRF9160DK.

The Figure 28 shows the initialization of both the *MQTT_controller* and the *BLE_controller*. It can also show that the collected data (in blue) are sent to the broker only when the connection is set (in red).

The Figure 29 shows that the collected data are well sent to the broker (in blue). Then, the detection of the beacons, when reached, is sent to the MQTT broker.

```

nrf9160dk_nrf9160ns.stc
New Open Save Connect Disconnect Clear Data Options View Hex Help
Bt_device_init -> Init bluetooth device, ADDR : d9:3f:f2:d1:0b:1b (random)
BLE_controller -> Init cadence sensor
Bt_device_init -> Init bluetooth device, ADDR : e6:6c:af:76:18:ad (random)
BLE state : ST_HR_SCANNING
MQTT state : ST_MQTT_INIT
BLE_controller -> Bluetooth ready
BLE_controller -> scan_start()
BLE_controller -> Scanning for heart rate sensor.
Bt_device_display -> ADDR : ec:68:be:8a:cc:f7 (random)
[MQTT_controller_broker_init:114] : IPv4 Address found 137.135.83.217
BLE_controller -> Device found: ec:68:be:8a:cc:f7 (random), RSSI : -49
BLE_controller -> HR Connected: ec:68:be:8a:cc:f7 (random)
BLE_controller -> HRS : Service discovery completed
BLE state : ST_SP_SCANNING
BLE_controller -> scan_start()
BLE_controller -> Scanning for speed sensor.
Bt_device_display -> ADDR : d9:3f:f2:d1:0b:1b (random)
BLE_controller -> Device found: d9:3f:f2:d1:0b:1b (random), RSSI : -50
BLE_controller -> SPEED Sensor Connected: d9:3f:f2:d1:0b:1b (random)
BLE_controller -> New heart rate value.
BLE_controller -> CSC : Service discovery completed
BLE state : ST_CAD_SCANNING
BLE_controller -> scan_start()
BLE_controller -> Scanning for cadence sensor.
Bt_device_display -> ADDR : e6:6c:af:76:18:ad (random)
BLE_controller -> New heart rate value.
BLE_controller -> New heart rate value.
BLE_controller -> CSC-Speed : New value.
BLE_controller -> New heart rate value.
BLE_controller -> New heart rate value.
BLE_controller -> CSC-Speed : New value.
BLE_controller -> New heart rate value.
BLE_controller -> New heart rate value.
BLE_controller -> CSC-Speed : New value.
BLE_controller -> CSC-Speed : New value.
[MQTT_controller_mqtt_evt_handler:195] MQTT client connect 0
MQTT state : ST_CONNECTED
-> publish connected to nrf9160 state topic
BLE_controller -> New heart rate value.
BLE_controller -> Send heart rate value to broker !
[MQTT_controller_mqtt_evt_handler:226] PUBACK packet id: 34920
MQTT state : ST_PUB_ACK for QOS 1
MQTT state : ST_CONNECTED
BLE_controller -> CSC-Speed : New value.
BLE_controller -> Send speed value to broker !

```

Figure 28: BLE and MQTT behavior.

```

nrf9160dk_nrf9160ns.stc
New Open Save Connect Disconnect Clear Data Options View Hex Help
BLE_controller -> CSC-Speed : New value.
BLE_controller -> Send speed value in rpm to broker !
BLE_controller -> New heart rate value.
BLE_controller -> Send heart rate value to broker !
BLE_controller -> CSC-Cadence : New value.
BLE_controller -> Send cadence value to broker !
BLE_controller -> New heart rate value.
BLE_controller -> Send heart rate value to broker !
BLE_controller -> CSC-Speed : New value.
BLE_controller -> Send speed value in rpm to broker !
BLE_controller -> CSC-Cadence : New value.
BLE_controller -> Send cadence value to broker !
BLE_controller -> New heart rate value.
BLE_controller -> Send heart rate value to broker !
BLE_controller -> CSC-Speed : New value.
BLE_controller -> Send speed value in rpm to broker !
BLE_controller -> New heart rate value.
BLE_controller -> Send heart rate value to broker !
BLE_controller -> CSC-Cadence : New value.
BLE_controller -> Send cadence value to broker !
BLE_controller -> New heart rate value.
BLE_controller -> Send heart rate value to broker !
BLE_controller -> New heart rate value.
BLE_controller -> Send heart rate value to broker !
BLE_controller -> Beacon start reached
----- Data buffer -----
->Type(0:HR,1:SP,2:CAD,3:Beacon) : 3
->Data length : 47
->Data: {"beacon":"start","timestamp_ms":1597014389279}
->New data : 1
BLE_controller -> Send beacon start to broker !
BLE state : ST_WAIT_STOP
BLE_controller -> CSC-Speed : New value.
BLE_controller -> Send speed value in rpm to broker !
BLE_controller -> New heart rate value.
BLE_controller -> Send heart rate value to broker !
BLE_controller -> CSC-Cadence : New value.
BLE_controller -> Send cadence value to broker !
BLE_controller -> Beacon stop reached.
----- Data buffer -----
->Type(0:HR,1:SP,2:CAD,3:Beacon) : 3
->Data length : 46
->Data: {"beacon":"stop","timestamp_ms":1597014389910}
->New data : 1
BLE_controller -> Send beacon stop to broker !
BLE state : ST_STOP

```

Figure 29: BLE and MQTT behavior.

9 Web server part

The web server tasks are the following :

- Collect data from MQTT broker.
- Display received values to the web page.
- Push Data on the database.
- Show graph about the stored values.

9.1 Collect data from the MQTT broker

The data are collected from the MQTT broker with the *Paho MQTT client*⁴². The first thing to do is to connect to the broker in order to be notified for new values on the topic. Then, when the connection is done, the subscription to the wanted topic can be done. Finally, the web server will be notified when a message arrives on the broker. These steps are shown in Figure 30.

⁴²<https://www.eclipse.org/paho/>

When a new message arrives, the callback function *onMessageArrived(message)* is called. In this function, the topic of the incoming message is compared to the subscribe topic in order to execute the corresponding code section. In this code section, the *JSON.parse(message)* is used to parse the message into an object. This allows to get the timestamp and the value of the message. When the timestamp and the measures have been received, they can be inserted in the right measurement in InfluxDB(section 9.2) with a Javascript API⁴³.

⁴³<https://github.com/boynet/Influxdb-js-client>

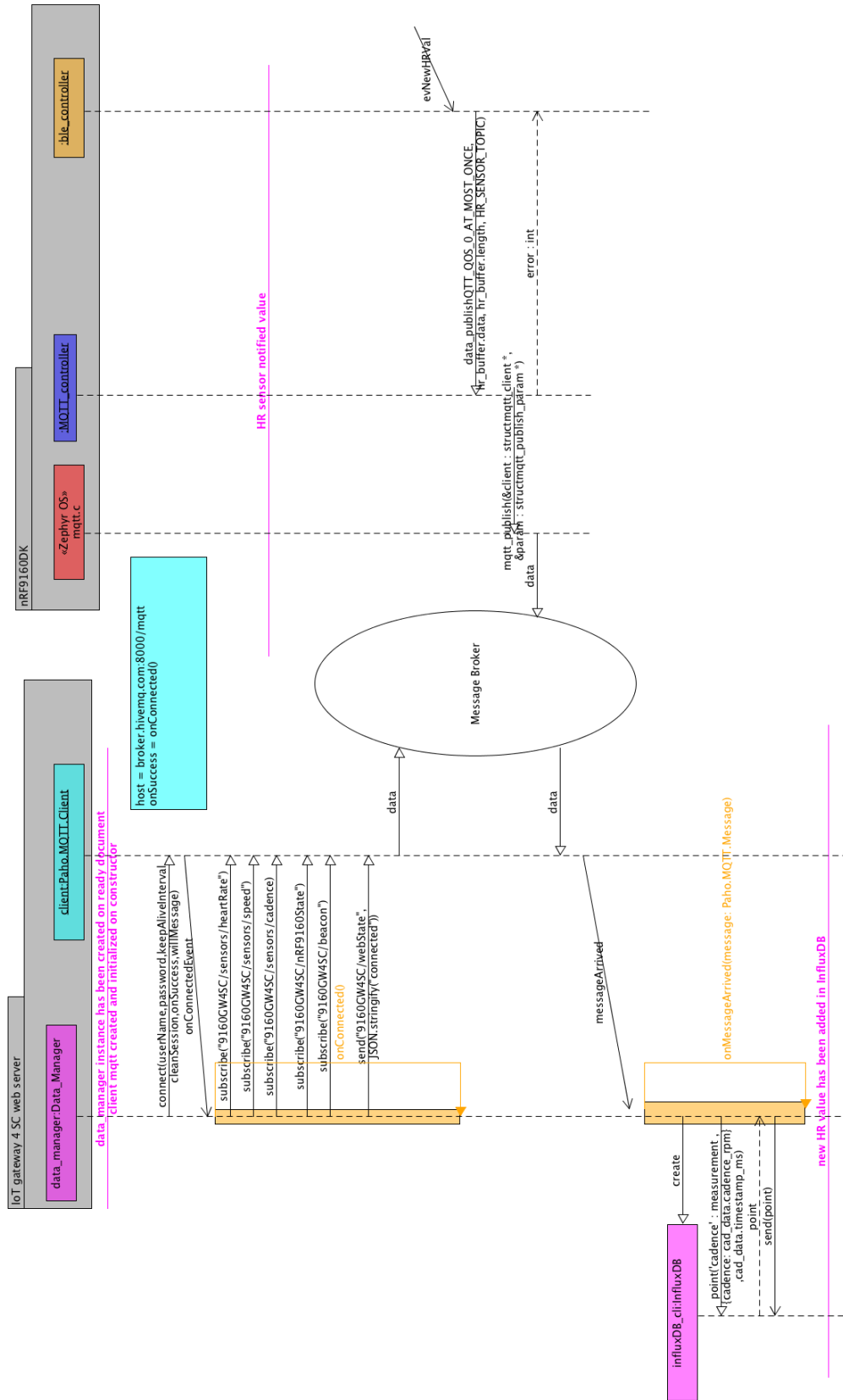


Figure 30: Way data are collected from the broker.

9.2 InfluxDB

InfluxDB⁴⁴ is a time-series database. It allows, for example, to create a new database, new measurements (tables), to insert and query data. There are many ways to manage the database. For example, one can use the Command Line Interface (CLI⁴⁵) with the InfluxQL (Query Language) or directly use the HTTP API⁴⁶.

The InfluxDB server default runs on *localhost:8086*.

9.2.1 Database and measurements for this project

In order to insert data in a database, such a database has been created : *ioiotgateway4sc_db*. This is shown in Figure 31. This request has been made with the CLI.

```
MacBook-Air-de-Rossier:IoTGateway Yoan$ influx
Connected to http://localhost:8086 version v1.8.1
InfluxDB shell version: v1.8.1
> show databases
name: databases
name
----
_internal
testdb
iotgateway4sc_db
> use iotgateway4sc_db
Using database iotgateway4sc_db
```

Figure 31: Start influxDB Command Line Interface and set database.

In the database, measurements have been created to dedicated sensor and beacon data (Figure 32).

```
> show measurements
name: measurements
name
----
beacon
cadence
heartrate
speed
```

Figure 32: List of measurements in *iotgateway4sc_db*.

The Figure 33 shows an example of query.

```
> select bpm from heartrate where time > 1597015872997000000
name: heartrate
time                bpm
----                ---
1597015873497000000  61
1597015873997000000  61
1597015874497000000  61
1597015874997000000  61
1597015875497000000  61
1597015875997000000  62
```

Figure 33: Select query example.

⁴⁴<https://www.influxdata.com/products/influxdb-overview/>

⁴⁵<https://www.docs.influxdata.com/influxdb/v1.8/tools/shell/>

⁴⁶<https://docs.influxdata.com/influxdb/v1.8/tools/api/#influxdb-1-x-http-endpoints>

9.3 Grafana

Grafana⁴⁷ allows to visualize data on a time-series database, including InfluxDB. The Grafana server default runs on *localhost:3000*. The way to visualize data is to configure queries that will be display on a graph. It is possible to configure a dashboard (set of panel) to see multiple graphics or multiple representations of the same query.

9.3.1 Panel configuration

The Figure 34 shows how a panel can be configured. In addition to query, it is possible to set a refresh time, which will query periodically. It is also possible to change the representation of the graph or to show multiple queries on the same graph. This will be done to display the segment on the same graph as the BLE sensor measurements.

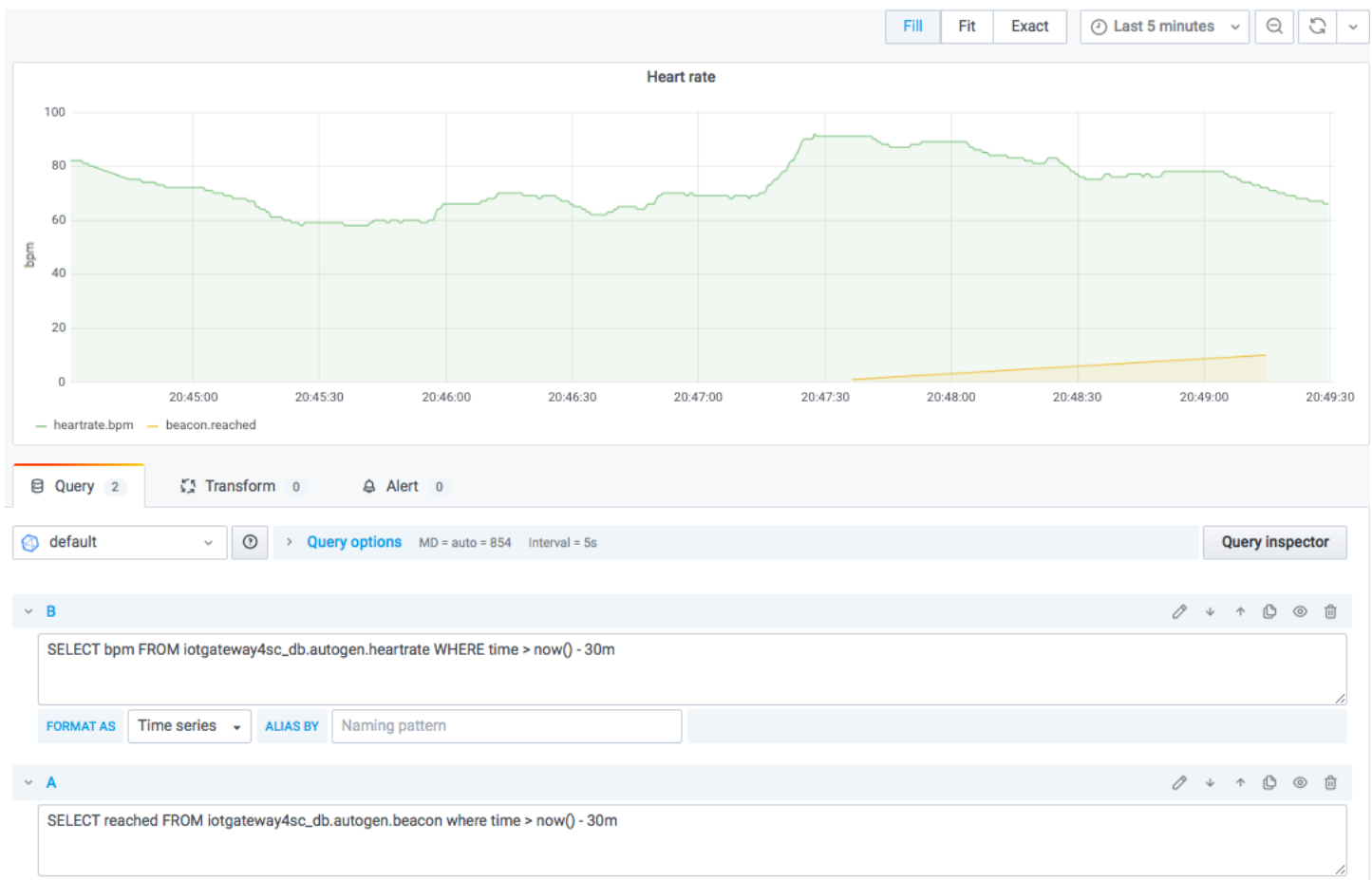


Figure 34: Make graph with InfluxDB query .

9.3.2 <iframe>

This is the definition of the <iframe> tag :

⁴⁷<https://grafana.com>

”The `<iframe>` tag specifies an inline frame. An inline frame is used to embed another document within the current HTML document⁴⁸.”

Thus, with this tag included in the HTML file, a panel from Grafana can be integrated. This embed link can be found by a right click on the panel and then share.

The Figure 35 specifies the `<iframe>` included in the web page of this project. One can see that there is a relative time graph that shows the data from : now-5minutes.

```

<!--To get live graph from Grafana, must be log before on grafana.-->
<iframe src="http://localhost:3000/d-solo/LF9tY6Mz/cycling-data?orgId=1&refresh=5s&from=now-5m&to=now&panelId=2" width="900" height="400" frameborder="0"></iframe>
<br>
<iframe src="http://localhost:3000/d-solo/LF9tY6Mz/cycling-data?orgId=1&refresh=5s&from=now-5m&to=now&panelId=4" width="900" height="400" frameborder="0"></iframe>
<br>
<iframe src="http://localhost:3000/d-solo/LF9tY6Mz/cycling-data?orgId=1&refresh=5s&from=now-5m&to=now&panelId=6" width="900" height="400" frameborder="0"></iframe>
/html>

```

Figure 35: Embed panels (speed, cadence and heart rate data) in a personal web page.

9.4 Web page

The web page is designed in a simple way. It displays :

- The nRF9160Dk status connection to the broker.
- The times when the beacons are reached.
- The received values in live from the MQTT broker.
- The possibility to enter a wheel diameter to calculate the speed from the rpm.
- A link to the Grafana Dashboard.
- The embedded configured graphs from Grafana.

The Figure 36 displays the web page.

⁴⁸https://www.w3schools.com/tags/tag_iframe.asp

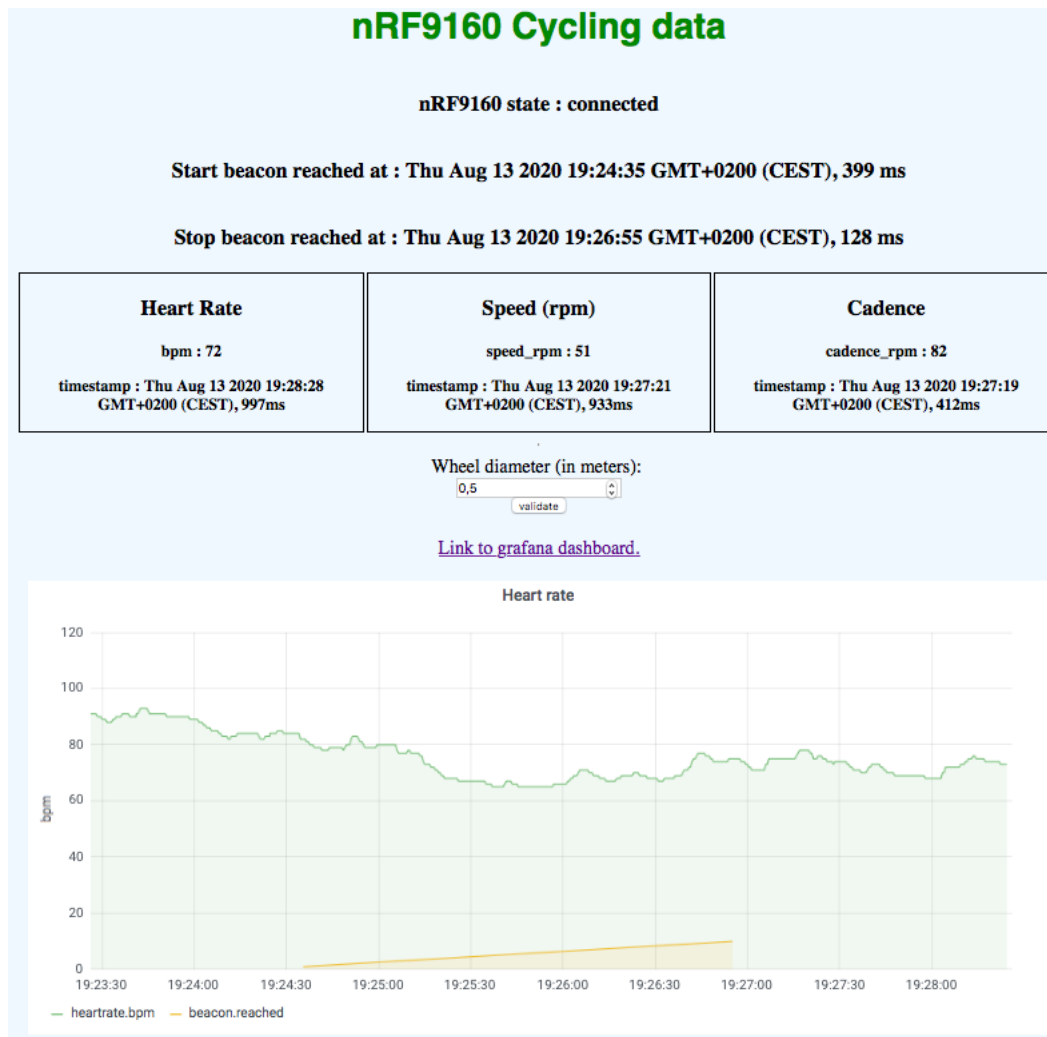


Figure 36: Web page.

9.5 Calcul of speed

In order to calculate the speed, the possibility to enter a wheel diameter is given. The new wheel diameter is set by clicking on the validate button. Thus, it is possible to change the wheel diameter at any time, as shown in Figure 37, in which the wheel diameter suddenly passes from 0.5 to 5 meters.

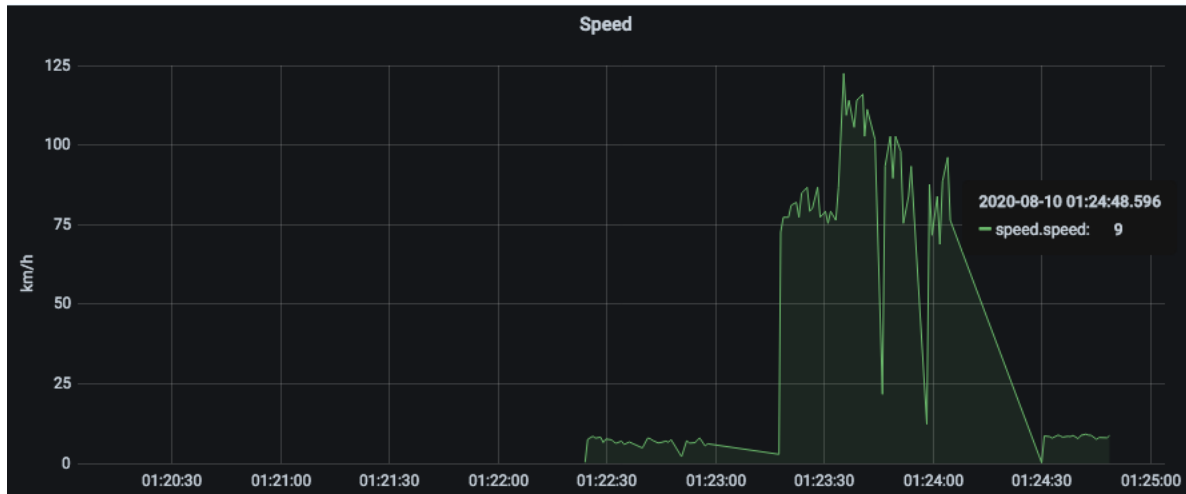


Figure 37: Reactive speed on wheel diameter changes

9.6 Tests

9.6.1 One hour test

Goal Verify that all system holds in time and don't crash

Initial condition The whole system was running : nRF9160DK, web-server, InfluxDB and Grafana. Note that only heart rate measurement was connected all time due to is easy wearability.

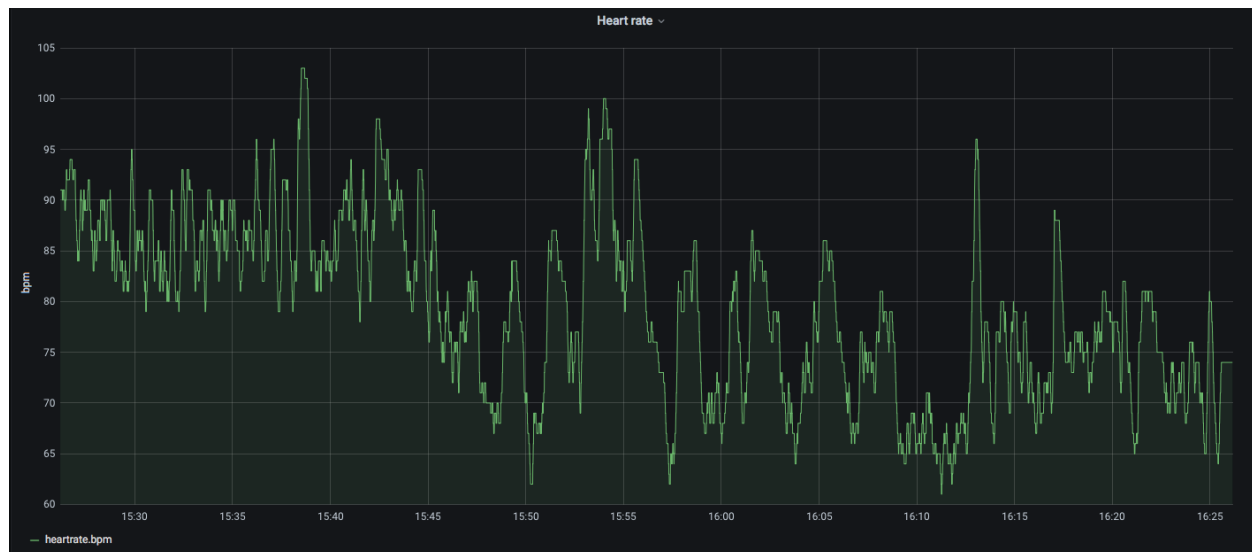


Figure 38: One hour test.

9.6.2 Beacons test

Goal Verify that the beacons are detected, well inserted in the database and displayed on the same graph as the BLE data sensor.

Initial condition The whole system is running : nRF9160DK, web-server, InfluxDB and Grafana.



Figure 39: Beginning and end of the segment on the graphs

Results The segment appears directly on the graph as expected.

10 Embedded board

10.1 nRF9160DK power supply

There are three ways to power supply the gateway :

1. From USB.
2. From P15.
3. From P28.

10.2 Powerbank

To embedded the gateway, a power bank was used. Things to be aware with the used power bank is that it turns off if not enough current is drawn. Solution to this issue was to change the voltage on the nRF9160DK pins. Thus, the voltage have been increase from 1.8V to 3V.

11 Conclusion

11.1 Final state

This project demonstrates the capability to collect data from BLE sensors located on the cyclist and on the bike and how to expose them to other people in realtime. Developing this to a final product may improve the cyclist training session and open a new way to train by directly giving feedbacks to the cyclist. This will save time and improve the cyclist performances during each training session.

11.2 Things to improve

There are several things that can still be improved in this project:

1. The implementation : The whole project is not coded in the same way. Some parts of the code are set to follow the oriented object pattern and some are not.
2. Unexpected events : If the BLE sensors disconnect, nothing is done. Trying to reconnect could be an improvement.
3. Sequential state machine : All the sensors must be advertising because the connection to the BLE sensors is made one after the other. For example, if the heart rate sensor is out of power, the other sensors won't be able to connect to the gateway.
4. Pre defined addresses : All the sensors that the gateway must connect need to have their addresses set previously in the gateway. A solution to this issue could be to filter the sensors with the UUID of a specific service. However will retained the gateway to connect to another device, which is not the wanted device.

5. Nowadays, all the server are running locally and aren't available from another computer or a mobile phone.
6. The web page is created in an elementary form and is not very user friendly.

11.3 Acknowledgements and final words

I want to extend a huge thanks to all people associated to this diploma thesis specially: M. Rieder, M. Mottier for their support during the whole work on the thesis.

Finally, this project allowed me to discover a part of the Nordic environment, LTE-M connection and more details about bluetooth. This project was very motivating and allowed me to progress in my engineering skills

12 Annexes

12.1 Installation of project components

12.1.1 Install nRF connect SDK and set up environment

The *nRF9160 DK Getting Started*⁴⁹ web page from the *Nordic infocenter* provides all steps to set the development environment, configure the nRF9160DK and get a first sample run on the board.

12.1.2 install Grafana and Influx

The *Install Grafana*⁵⁰ web page from *Grafana* provides all the steps, according to the user OS, to install Grafana. The *Downloads*⁵¹ web page from *InfluxData* on section InfluxDB v1.x provides all the steps, according to the user OS, to install InfluxDB.

12.1.3 Webstorm

The *Download WebStorm*⁵² from *JetBrains* provides all the steps to install the Webstorm IDE.

12.1.4 Build and flash on the nRF9160DK

The following commands must be type in the console in order to build and the flash application :

- export ZEPHYR_BASE= path to Zephyr Base
- export ZEPHYR_TOOLCHAIN_VARIANT=gnuarmemb

⁴⁹https://infocenter.nordicsemi.com/index.jsp?topic=%2Fug_nrf91_dk_gsg%2FUG%2Fnrf91_DK_gsg%2Fintro.html

⁵⁰<https://grafana.com/docs/grafana/latest/installation/>

⁵¹<https://portal.influxdata.com/downloads/>

⁵²<https://www.jetbrains.com/webstorm/download/#section=mac>

- export GNUARMEMB_TOOLCHAIN_PATH= path to the gnuarmemb
- west build -p -b nrf9160_pca10090ns (for the nRF9160 SiP) or west build -p -b nrf9160dk_nrf52840 (for the nRF52840 SoC)⁵³.

12.2 Start Grafana and InfluxDB

Start Grafana If Grafana is install with *HomeBrew*⁵⁴ the following command line start and stop Grafana :

- brew services start grafana
- brew services stop grafana

Start InfluxDB To start InfluxDB the *influxd* command line must be typed in console.

12.3 How to run application

To run application following steps must be performed :

- Load *hci_uart* sample from Zephyr on the nRF52840 chip.
- Load application code on the nRF9160DK.
- Turn on all sensors (heart rate, speed, cadence)
- Start InfluxDB
- Start Grafana
- Launch the Webstorm project and run it.

When the four LEDs are on on the gateway value should be appear in the web page.

12.4 Diagrams

⁵³Don't forget top put the sw5 switch to the correct position

⁵⁴<https://brew.sh>

12.4.1 Complete nRF9160DK class diagram

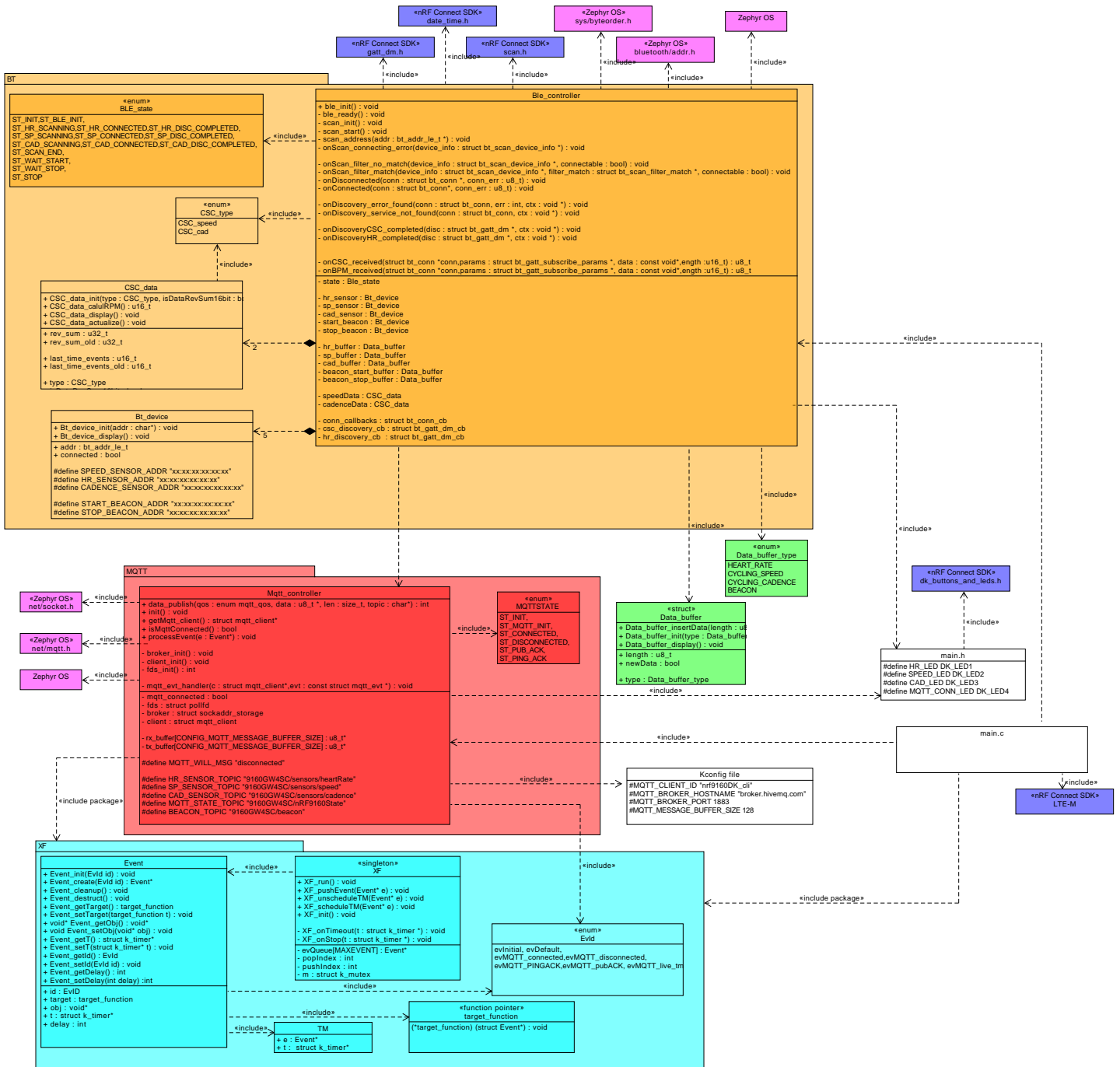


Figure 40: nRF9160DK complete class diagram

12.4.2 Complete web server class diagram

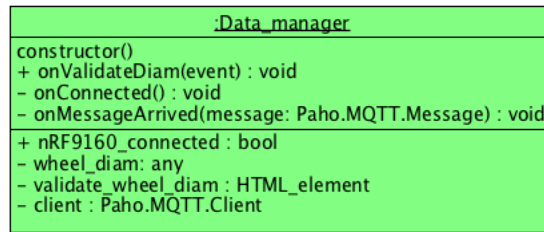


Figure 41: Web server class diagram