
POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

# Development of a synchronized acquisition system based on a BLE-sensor network for real time IoT applications

**Relatore**
Gianvito Urgese

**Candidato**
Andrea Pignata

October 2022

# Abstract

Nowadays, Internet of Things (IoT) hardware and technologies are getting more and more importance in our everyday lives: these small, low-cost devices, composed of sensors and wireless data transmitter, are gaining popularity. Despite all of this, these systems present some drawbacks and developers are forced to find compromises between measurement accuracy, battery duration and system costs, trying to optimize all these variables for deploying amazing smart applications.

Sensor fusion is one of the emerging techniques for analysing heterogeneous data coming from different sensors, for elaborating information to be exploited in several IoT applications. However, for the correct implementation of sensor fusion models, synchronization of the heterogeneous signals is a must. The literature presents some works on clock alignment, but those are intended for systems based on the same HW platform, require optimization of low-level connection layers (not available to developers in most cases), and are not always thought for wireless transmission. Hence, even if the presented results are promising, those methods can't be easily applied to heterogeneous, mass-produced devices.

In this thesis I present the system developed to synchronize the wireless-based acquisition of signals from four different sensor boards from ST Microelectronics and three from MBIENTLAB. I also explored the possibility to use the embedded microphones and the consequent audio data to synchronize and label data, finding valid ways to stream sound and implement Fast Fourier Transform (FFT) on the microcontrollers. I gave particular attention in getting the most out of the Bluetooth Low Energy protocol (BLE): it is a good solution from the energy-efficiency and compatibility points of view, but at the same time some optimization is necessary to keep a good throughput and maintain an efficient bi-directional connection. Finally, I developed time synchronization on the application layer of the BLE stack: it consists in the periodic exchange of timestamped packets between the client and the sensors. The client is then aware of the internal timestamp of the sensors in those re-synchronization moments, and can tag the samples with the precise moment in time they are acquired. The goals are a) development of firmware for the open-source ST devices, with a personalized time synchronization mechanism, b) design of clients for all the devices, formatting data in a consistent way and c) writing of MATLAB scripts to join together the resulting data.

The developed system behaves as expected in the experiments that have been carried out. I made initial tests to determine the best sample rate for the sensors, to guarantee the correct transmission through the BLE channel and to keep the packet loss/received ratio lower than 1%, which is 75 Hz. Then, I verified the behaviour of the time synchronization after 15 minutes of data streaming, showing a precision of around 70 ms, while the same setup without synchronization presents samples with delays of more than 1s. Finally, I performed an extensive test by installing the sensors in a car and collecting data in that environment, with the help of a camera and GPS devices. In this case, data have been labelled using sounds at predefined frequencies and elaborated at the node level. A calibration algorithm has been developed to rotate the sensor reference system for aligning it to the vehicle reference system. Results demonstrated that sensor information remains aligned in time and consistent with the recorded audio data. After the application of the calibration algorithm, data is also well aligned and consistent under this point of view.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The electronics market has been recently revolutionized with the massive manufacturing of the so-called Micro-Electro-Mechanical Systems (MEMS): micro-scale devices, with moving parts, that can be manufactured using some of the typical fabrication processes for Integrated Circuits (IC). Possible applications of these devices include a great number of different sensors, like Inertial Measurement Units (IMU), composed of accelerometers, gyroscope and magnetometer. Thanks to their already available assembly lines, silicon foundries started to produce these devices, making them small, easy to integrate, highly available and, most importantly, low cost.

In particular, small and low-cost are two important keywords when talking about Internet of Things (IoT). IoT is a term used to describe those objects composed of sensors, processing abilities, software and technologies for data transmission. Usually, their goal is fetching data from the environment using sensors, process that information and send it (raw or elaborated) to other devices, over the Internet or by custom connections. These devices are pretty easy to be found nowadays: lots of people have a smartwatch on their wrist, constantly fetching data about their physical activity and making users aware of their results via a smartphone application. Even homes are starting to be more IoT-friendly, with security cameras, alarm systems and thermostats that are connected to the Internet. As they are getting always smaller and cheaper, more and more people are buying them to make their lives easier. Another important field in which IoT is getting success is Industry 4.0: this word is used to group all the technological shifts that are interesting the industry domain in these days, including interconnection between different machines or assembly lines and smart automation. Also in this case the role of low-cost sensing and connected devices is important: different conditions during the manufacturing process can be easily monitored (sensors), fetched (interconnection) and elaborated, in order to understand how the processes currently used is working, if there is any improvement to be done, and eventually apply these changes automatically (smart automation). Figure 1.1 reports an example of a

configuration in which multiple IoT devices are used. Two nodes, represented in orange, are connected via a low-range transmission protocol (in this case Bluetooth) to an hub that can receive data from multiple devices. Then, the hub is responsible to encapsulate them and send them over WiFi to a centralized server. Another node, in green, overcomes the need of an intermediary and is capable of sending all the data via Internet. The centralized server is then used to receive all the data, perform elaboration on them, and generate some outputs that the final user can see on a website or on a smartphone application.

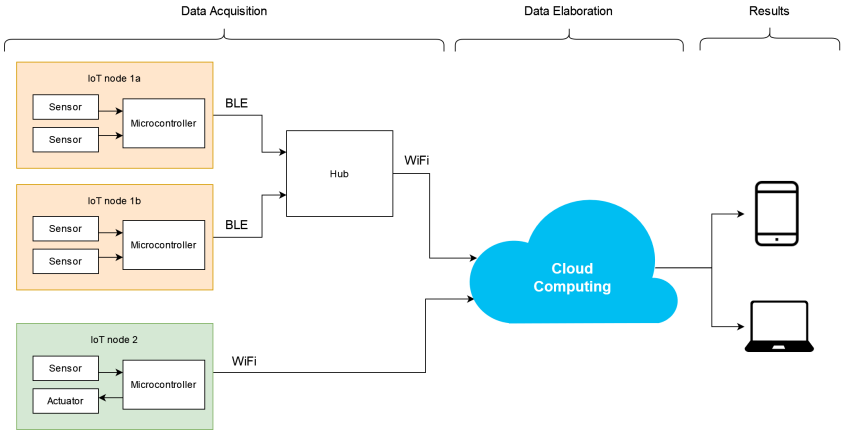



Figure 1.1: A configuration example for commercial IoT nodes

To sum it up, IoT requires a great number of sensors, and MEMS technologies can produce them easily and in a cheap way: their development is proceeding hand-in-hand, and nowadays different vendors, including ST Microelectronics and Bosch, have their own lines of micro sensors that can be implemented in a lot of applications. To improve their devices, they also embed functionalities other than the mere sensing ones, inside these MEMS: for example, some recent IMU devices from ST offer programmable Finite State Machines (FSM), that can generate interrupts when a particular condition is met. In this way, the microcontroller has less lines of code to execute, and doesn't have to continuously poll the sensor, with consequent energy savings.

For this thesis, I worked on some off-the-shelf boards that include different sensors. Some of them are evaluation boards, on which developers can deploy their firmware and verify if the available hardware is good for their goals. It's the case for the devices from ST Microelectronics, the SensorTile.box and the BlueTile. Other ones are ready-to-use solutions for motion and environmental data analysis: MetaMotionS boards from MBIENTLAB include all the hardware and the software for this scope.

These boards include different sensors:

- Environmental sensors: temperature, pressure, humidity and light sensors.
- IMU sensors: accelerometer, gyroscope, magnetometer.
- Microphone for audio data analysis (Fast Fourier Transform FFT) or transmission.

ST boards embed MEMS sensors from their own lineup, while MetaMotionS ones are primarily from Bosch.

Of course, cheapness and reduced dimensions have their own costs: the signal coming from the sensors is not high-precision, a lot of noise is present and output values are subject to bias. Better performing solutions are available, but only at much higher prices, they require more space and, last but not least important, more power. They also can't be wireless and battery-operated, that are the most important points of the above mentioned sensors boards.

My main goal has been working with these kind of sensors, trying to get the best out of them. In particular, I set up a data acquisition system in which multiple sensor boards, from different vendors, can be connected: in this way, it is possible to join the different information from the different sensors to get more accurate information. For example, sensors can be all placed on the same spot, to capture data on the same event happening on the same place, and signals can be compared to get rid of noise and bias. Another possibility is to place them on independent spots, capturing data of different nature. Then, with the help of Machine Learning algorithms, the fetched data can be joined to easily recognize different events: for example, putting sensors on more parts of the human body can help motion detection algorithms to return more accurate results. An example is reported in chapter 2.7.

Good part of the work has been the design of a mechanism to keep results synchronized over time. Each sensor board includes its own clock, subject to drift and offset: it is important that, in the final result, each sample is tagged with the moment of acquisition with the greatest precision possible, but more importantly that the timestamps obtained from the different devices are aligned as best as possible. In this way, the above mentioned sensor fusion algorithms can work correctly and produce coherent results.

As I have been working with low-cost IoT devices, with a very limited computational power, it is important that even this synchronization mechanism is low-cost and easy to implement: a good solution would be a trade-off between precision and complexity. Also the limitations of the Bluetooth Low Energy (BLE) protocol, used by all the considered boards, are taken into consideration.

The obtained result from this work is a system of 7 sensor boards, connected via BLE to a single client (a Linux computer). The client is responsible of collecting all

the data coming from the nodes, synchronize them over time and elaborate them to be further analyzed by other software solutions (like MATLAB) or algorithms (like sensor fusion ones). I tried to keep the time synchronization accuracy under 100 ms.

The system has been tested in different experiments: the first one has been carried out to understand the best data rate at which the boards can output data using Bluetooth Low Energy, to find an acceptable trade-off between the percentage of lost packets and the frequency at which data packets are generated. In this case, multiple situations have been tested, using different numbers of sensor boards and Bluetooth adapters. The second experiment, instead, had the goal of understanding how good the developed time synchronization is. In this case, all the devices streamed data for 15 minutes and, at the end, I gave them an acceleration on the Z-axis by making them fall (on a soft surface). Then, the delays between the accelerometer values are analyzed and discussed. The last experiment consisted of installing all sensors inside a car, collecting all the data from the sensors during a driving session. This was the most extensive test, and the system worked as expected. Some other simpler test have been performed during the development, including ones to verify the accuracy of the FFT, ones to test the audio streaming and ones to verify the direction of the axis for all the sensor boards.

To make the data acquisition campaign on the car, I also developed a calibration algorithm that is capable of aligning the sensor reference system to the vehicle reference system. In this way, even if sensors are not correctly aligned during the installation (as in our case), automation correction can be performed. Applying this algorithm to the data coming from the car experiment resulted in a correct rotation of the axes.

Possible applications of the developed system include human activity recognition for the electronics mass-market, and predictive maintenance for the industry. In my case, I explored the automotive domain during the last test: the resulting samples can then be labeled, and eventually a machine learning algorithm would be able to recognize in real time the different driving conditions (road bumps, left/right turns, and so on).

# Chapter 2

# Background

The most challenging part of this work has been dealing with low-cost sensors. These products are considered really interesting from different points of view, from the power consumption to the easiness of integration in PCBs, but at the same time some effort is required in order to get the most out from them. This is the subject of different studies, some of them focused on the performance/energy trade-off of the Bluetooth Low Energy standard, while others are trying to implement different mechanisms of time synchronization that could work between different sensors. Another field of exploration is the so-called sensor fusion: in these studies, more low-cost sensors are used in parallel, to join all their values and finally get more precise results by using adequate algorithms. In this background section, the goal is to sum up some of the most relevant studies and experiments that have been conducted on these fields, while exploring some state-of-the-art solutions that are available on the market.

## 2.1 Bluetooth Low Energy: standard overview and throughput analysis

The choice of radios and communication protocol is always a hard one when dealing with sensor networks. As different technologies offer different performances in different areas, the choice should be coherent with the kind of data we are dealing with. For example, some aspects that should be considered are throughput, latency, number of nodes and compatibility: while the first two could be critical in healthcare applications, compatibility could be crucial when dealing with sensor devices aimed at the mass market. In particular, the Bluetooth 4.0 standard (also called Bluetooth Low Energy) aims at maximizing device connectivity and energy efficiency, with the drawback of a limited throughput.

**The Bluetooth Application Layer.** When dealing with BLE chips, usually the final user can have control on the Application Layer of the communication protocol, while lower levels (Host and Controller) usually can't be optimized by the developer. In particular, the top level is composed of:

- Logical Link Control and Adaptation Protocol (L2CAP), takes care of protocol/channel multiplexing with error and flow control
- Security Manager, responsible for device pairing and key exchange
- Attribute Protocol, allows the device to expose a set of attributes and relative values
- Generic Attribute Profile (GATT), defines a way to interact with those attributes: devices can be clients or servers.
- Generic Access Profile (GAP), regulates the procedures of discovery and connection with other devices.

**Data Exchange.** When the connection completes, a device will assume the role of the GATT client, while the other one would be the GATT server. This means that data exchange happens only at the GATT level: the server will expose a number of *characteristics*, each one being an array of bytes (with the maximum set to 512 B), grouped into *services*. The GATT client can interact with those characteristics, depending on the *properties* of each one. Those properties can be:

- **Read**: the client explicitly requires the value of the characteristic, the server answers accordingly
- **Write**: the client writes the value of the characteristic
- **Notify/Indicate**: the server will inform the client of a new value of the characteristic, without the need of an explicit request. Indications also include an application level acknowledgement of the notification.

Usually notifications are preferred, as only one packet should be exchanged for each new value: no request and no acknowledgment are necessary.

A particular characteristic of the BLE protocol is the usage of the so-called Connection Events (CE). Data exchange can only happen during these events, and between them the radio goes to sleep: this duty cycled behaviour is one of the reasons of the low energy consumption of the standard. The time interval between two consecutive CEs is called Connection Interval (CI), with allowed values going from 7.5 ms to 4 s (at 1.25 ms steps). These parameters are negotiated at the connection setup, but they can be changed at any time during the connection.

**BLE Throughput.** The standard defines the data transfer rate to be 1 Mbit/s, but the effective value depends on link quality, layer overheads and hardware constraints. Some studies tried to calculate it, obtaining different results:

- Gomez (2011) [1] reports a maximum throughput of 236.7 kbit/s, by representing the connection through an analytical model. The main flaw of this approach is that it does not take into consideration the hardware constraints of low-cost BLE nodes.

- In [2], performances of Bluetooth Low Energy are compared to other wireless solutions, reporting a link layer throughput of 122.6 kbit/s: this value is not considering upper layers of the standard, hence it is not considering the packet overhead coming from the application layer.

- In [3] presents an overview on the standard, carrying out a practical experiment on throughput: two CC2540 devices, from Texas Instruments, are configured to send data at the maximum possible speed, with Connection Interval set to the minimum possible value. The resulting throughput is 58.48 kbit/s.

- [4] is probably the most extensive one, as multiple BLE modules are used, each one with its own stack and firmware. It underlines the importance of Connection Intervals in maximizing the throughput. To sum up, the study concluded that the maximum application level throughput that can be achieved is 64 kbit/s, by fine tuning the values of CIs and by using buffer-like structures to maximize the data to send out at each Connection Event.

## 2.2 Bluetooth performances in Sensor Networks

After understanding how BLE works and how to get the most out of it when dealing with a single node, I focused on more complex configurations. In [5], the goal is to set up a sensor network and to find the highest possible data rate, in order to use data coming from IMU devices for Human Activity Recognition (HAR) purposes. The researchers state that, for a good HAR analysis, they would require data coming from as many sensors as possible, with a sampling data rate between 25 ms (40 Hz) and 35 ms (28 Hz).

From the technical point of view, they decided to use notifications to reduce the number of exchanged packets to the bare minimum, left the original setting of the sensor node with regards to the Connection Interval and used a Python script to harvest all the data and perform the analysis on them. To make the performance analysis as complete as possible, they had 20 sensor nodes available, and used different computers as clients, each one with a different Bluetooth adapter. To be more precise, they used up to 20 *Arduino Nano 33 BLE Sense*, with 9-axis IMU,

barometer and temperature sensors connected at different computers, each one with its own Bluetooth Adapter.

The main test consisted of finding the maximum number of Bluetooth sensors that can be connected to a single client, varying the data rate between 25 ms and 35 ms and trying all the different computers. The results then were compared considering the percentage of lost packets. The results table in the paper makes it evident that the choice of the Bluetooth adapter is quite important: 10 sensors connected, with data rate of 35ms, resulted in 20% data loss for the Macbook Pro 2014, while the Intel NUC 8 shows a loss percentage of 0.07% only.

To sum up this experimental analysis and its conclusions:

- Reseachers showed that the best performance are obtained by using a DeLock Bluetooth Adapter, based on the Broadcom BCM20702A0 chip. It managed 13 simultaneous connections, with a loss percentage of 0.03% with the sensors set at 35ms data rate.

- Notifications perform better than indications in high performance sensor settings.

- There are big differences in performance between Bluetooth modules.

## 2.3 An example of time synchronization from National Instruments

National Instruments is a company that produces test equipment and virtual instrumentation software. One of the most important application for their products is data acquisition, and they are considered one of the leaders thanks to their experience in this field.

An article from their website [6] states that data should be appropriately synchronized in time to be accurately analyzed, and they propose a list of approaches to fulfill such goal using their products. In particular, they focus on their PXI platform.

**The PXI Platform** PXI is defined by National Instruments as the standard for automated test and measurements. It is a family of modular instruments and I/O elements that can work together, with specialized synchronization algorithms and software. The main core of the system is the chassis with its own controller, that accepts a certain number of measurement modules and provides the same functionalities to all of them.

In the latest version of the platform, called *PXI Express*, developers included different elements thought for precise time synchronization:

- A 10 MHz backplane clock, offered in parallel by the chassis to all connected component
- A single-ended trigger bus
- A star trigger signal, accurately developed such that the connection to all the modules have the same length
- 100 MHz differential clock and differential triggers, with increased noise immunity.

The user can then freely choose the way to share clock and trigger signals between all modules. Every option has its own advantages and disadvantages considering signal skew, signal integrity, compatibility with external modules and necessity for external controllers. For example, the 100 MHz clock provides a skew that is less than 250 ps, while the star trigger signal performs a skew that is less than 500 ps. Both of them provide the best signal integrity between all the solutions. Of course, the listed solutions are intended for modules connected to the same chassis, but some components can perform synchronization between different ones by using GPS or IRIG-B standards.

**Synchronization Techniques** In order to effectively perform time synchronization, National Instruments defines the differences between two types of devices:

- Sample Clock Timed Devices. These devices use a *sample clock* that controls when the samples are acquired. For example, SAR ADC converters usually have a dedicated convert line that, when toggled, triggers an immediate conversion.
- Oversample Clock Timed Devices. In this case, the device requires a free-running oversample clock that drives the conversion while keeping the ADC operating: the resulting samples are generated at a rate that is a division of the original clock. These devices can't be controlled by external signals, hence synchronization is more complicated.

To synchronize the sample clock, different options are possible:

- Share sample clock directly. A master device will share its sample clock to the slave ones, using one or more of the backplane of the PXI chassis.
- Derive the sample clock from a common reference clock. The 100 MHz differential clock is divided to generate the sample clock, distributed to all the devices. A dedicated software from NI is able to perform all the necessary configuration for the user.

- Share the trigger signal. This is the simplest but the loosest synchronization mechanism, as all the devices will only receive the same trigger signal. In this case, usually devices drift in time, as sample clocks are not locked together.

When dealing with oversample clock timed devices, the only possibility is the reference clock synchronization. First of all, the ADCs on the devices are reset and configured to generate samples after the same number of clock cycles. Then, the acquisition is started synchronously with a shared trigger signal. During the reset and sampling procedures, all the devices are locked to the 100 MHz clock signal provided by the PXI chassis.

**Conclusions** National Instruments solutions are, of course, the best synchronization mechanisms that I found during the research, offering extremely high performances and a great software support for easy configuration. All of this comes at certain costs:

- All the modules should be physically connected to the chassis, and no wireless connection is possible (it wouldn't support this kind of synchronization)
- The modules should support the National Instruments standard. Even if it is open and can be implemented by manufacturers quite easily, it is targeted at high-cost, high-precision measurement tools.
- The PXI system is extremely expensive: the cheapest controller has a cost that is around 1400€ (PXIe-1090), with no module installed.

## 2.4 Precise Sensor Synchronization in Robotic Computing

Paper [7] summarizes the development of time synchronization systems done by a development team interested in different commercial robotic and autonomous driving products. They underline the importance of time synchronization by presenting the example of a camera and a Laser Imaging Detection and Ranging (LiDAR) sensor: if the measurements are not aligned on time, those would cause inaccurate and ambiguous view of the environment, with the possibility of catastrophic outcomes.

**Addressed issues and goals** The main goal of the researchers has been to *ensure that sensor samples that have the same timestamps correspond to the same event*. To reach this objective, they developed two kinds of synchronization: intra-machine (synchronizing sensors within a particular machine) and inter-machine (synchronizing sensors across different machines). They reported some examples to motivate the need for synchronization:

- Perception tasks: in order to fuse 2D information from cameras and 3D information from LiDARs, their data must be synchronized (intra-machine).

- Localization tasks: autonomous machines use multi-sensor fusion algorithms to obtain accurate localization estimations, for example fusion of camera and IMU information (intra-machine). It was observed that a misalignment of 40 ms can cause a 10 m translational error with a 3° rotational error.

- Interaction with other machines: when dealing with robots (or other devices) collaborating together, their synchronization could be critical (inter-machine). The example of two Road Side Units is reported: they use the position of a vehicle with the corresponding timestamp when calculating its speed. If the timestamps are not coherent, the speed may vary significantly.

**Synchronization principles** In order to perform the synchronization, two requirements have been found: first, all sensors should be triggered simultaneously, and secondly, each sensor sample should be associated with a precise timestamp. This is challenging, as all sensor have different triggering and time-stamping mechanisms. Four (theoretical) principles have been found:

- Externally-triggered sensors should be triggered in hardware. Camera and IMUs can be easily triggered via software, but the driver stack may generate different latencies that can't be precisely predicted. The solution is the trigger through custom hardware.

- Use a shared timer to simultaneously initiate internally-triggered and externally-triggered sensors. Internally-triggered sensors, such as LiDARs, usually accept an external initialization signal, then they would generate samples at a defined rate. It is important to use a common timer to generate both this initialization signal for them and the trigger signal for the other devices.

- Synchronize machine timer with GPS global clock. If inter-machine synchronization is necessary, it is important that their time sources are aligned to a global high-precision atomic clock, like the time data transmitted by GPS satellites.

- Obtain the timestamp for a sensor sample as close to the sensor source as possible. Again, the easiest way would be to timestamp samples via software, but as explained before this can be not so precise. A better solution is to obtain the time information directly at the sensor interface, that is, when the sensor data is received by the SoC from the off-chip sensor.

**Adopted Solution** The researchers decided to adopt an FPGA-based solution to satisfy all the principles stated above. In particular, the Xilinx Zynq Board contains a general-purpose ARM CPU to run the software components, with a lot of I/O interfaces for all the connectivity necessities.

A serial port interface accepts GPS global data, distributing it to all hardware and software timers. Trigger and timers units, designed in hardware, manage the triggering and the initialization of the sensors. Finally, Serial port, CAN and MIPI interfaces are designed to automatically assign timestamps to the samples as soon as they're received from (respectively) the IMU, camera and radar sensors.

The obtained circuit is lightweight (6.9K LUTs and 7.1K Registers). Accuracy is particularly good, as timestamping is performed at cycle-level precision for most sensors, while the LiDAR sensor (internally triggered) showed variations within 100 $\mu$s.

## 2.5 Synchronization in Wireless Sensor Networks: Review

Paper [8] reviews synchronization mechanisms aimed at Wireless Sensor Networks, in which the single low-cost, low-power nodes should perform synchronization and transmission by themselves. It also underlines the importance of correlation between different reports from different devices to perform data fusion, time-division multiple-access scheduling and to use real-time applications, even if the delay variability of the wireless connection and the computation limitations of the nodes make the issue even more complex.

**General Concepts** The main idea regarding synchronization in this paper is to mathematically estimate the oscillators in each node, and then implement some sort of message exchange mechanism between them. The goal is to model the differences between the clocks in the devices and modify their behaviour accordingly.

The **clock models** analyzed, on which the synchronization algorithms will base their estimations, are:

- Offset-only model. The local clock of a node is defined as in 2.1

$$C_i = t + \theta_i \tag{2.1}$$

  where $t$ is an ideal clock and $\theta_i$ the offset relative to that ideal clock.

- Joint skew-offset model. This provides long-term estimators, at the cost of augmented calculation complexity. The local clock can be approximated as done in Equation 2.2

$$C_i = \alpha_i t + \beta_i \tag{2.2}$$

where $\alpha$ is the absolute skew and $\beta$ is the offset of the node clock.

Once the clock model has been defined, the **message exchange mechanism** should be chosen:

- Sender to Receiver: nodes periodically exchange timestamped synchronization messages. This happens in 4 steps. Node $n_1$ sends a packet with timestamp $t_1$. The second node $n_2$ receives it, with timestamp $t_2$, and replies with timestamp $t_3$. At the end, $n_1$ timestamps the reception of the reply with $t_4$.

- Receiver to Receiver: in this case, a reference (defined exploiting the property of the physical broadcast medium) is used to broadcast messages simultaneously to all devices, that will timestamp the reception event as $t_1$ and $t_2$. Those samples are the only ones used for synchronization purposes.

Timestamps obtained through these mechanisms are not totally accurate, as the messages go through a path that may vary nondeterministically to different factors: send time (packet generation and transmission time), medium access time (MAC level), propagation time, reception time (packet processing). The receiver to receiver method removes send and access time from this path.

Once the method has been defined, the paper reports three kind of synchronization categories: 1) event ordering, which is the simplest form, 2) synchronization to a reference and 3) relative clocks. The last one is the most used in the analyzed algorithms, and it consists in maintaining inside the node the information about relative skew and offset to other nodes.

**Main issues** The paper proceeds with a list of all the possible problems in maintaining a good time synchronization between the nodes:

- Fast drifting. This is considered the main issue, and it derives from the usage of cheap circuitry components (in particular, oscillators). Usually sensor nodes present two kind of oscillator, a high-speed one (usually with high drifting), and a lower speed one, used in energy-saving situations, that is usually more precise but provides less granularity.

  Of course, a fast drifting clock requires frequent re-synchronizations, that may result very expensive.

- Node limitation. If the synchronization requires some calculation at the node level, usually the calculation power at disposal is very low, and all of it should be optimized as more as possible. For example, some nodes may not dispose of FPU units, and FP calculation may be implemented at software level, causing delays.

- Delay Variation. This is intrinsic to the message exchange mechanism, as there is some nondeterminism when considering message delays. Usually, this issue can be overcome by timestamping packets at the MAC layer, but this solution is both not portable between different devices/protocols, and not applicable to devices whose software stack hides the lower level to the developer.
- Fault Tolerance. These synchronization mechanism neglect aspects like nodes failure or packet losses, and provide no recovery algorithm in those cases.
- Accuracy measurements. The accuracy of such protocols is extremely difficult to assess, as instantaneous local times from the nodes should be captured at the same physical time.
- Impact of empirical parameters. All the possible implementations relay on some parameters regarding the message exchange, in particular, the number of rounds before estimation $k$ and the period between rounds $T$.

**Conclusions** The main issue found in this paper with regards to time synchronization is, without doubts, the clock drifting issue. Two solutions have been found: frequent message exchange in the simpler offset-only clock model, or less messages using the more computationally expensive skew-offset model. The choice of the final model is up to the developer, that should choose the least expensive of the two considering the node and the connection protocol at their disposal.

The second big challenge is the delay of message exchange/handling. The usage of the Receiver to Receiver model will reduce that delay, but optimizations can be performed using a MAC-layer timestamping solution (when possible).

## 2.6 Bluesync: a Time Synchronization protocol for BLE

Paper [9] explains an interesting protocol that can be used for time synchronization purposes when dealing with Bluetooth Low Energy devices, like sensor network nodes. The article stresses the importance of a low-cost and high-efficiency mechanism that can work on these resource-constrained IoT nodes.

In particular, BlueSync is defined as a synchronization service for BLE beacon devices. BLE Beaconing is an advertising mode, supported by Bluetooth Low Energy, that enables the exchange of data between devices without the need of entering a bonded connection: the server device will *advertise* its data, while the client can read it doing a *scanning* operation. This protocol does not present any time synchronization service. BlueSync makes the system work in two different stages: first, there is a timeslot (few seconds) in which synchronization packages are

exchanged, then, the second timeslot will allow the execution of synchronous tasks. Of course, the time interval between the two stages should be decided with care, as precision will depend on that. The developers showed that the node can proceed with 10 minutes of synchronous tasks without the need of re-synchronization, still with a good time precision.

The devices tested in the work are from Nordic Semiconductor, in particular an nRF51 (16 MHz ARM Cortex-M0) and an nRF52 (64 MHz ARM Cortex-M4).

**Implementation** The implementation of the BlueSync protocol is presented in three steps, that are reported and explained as follows.

- Timestamping. The main problem encountered when dealing with BLE beacons is that they usually exchange data in three different channels, with non-predictable delays. With proper modifications of the registers of the Nordic devices, the developers managed to set the transmission (advertising) and the reception (scanning) of the data on a single channel, both on the server and the client devices. Once this has been resolved, the problem of delays is addressed and solved by precisely deciding the moment at which packets are timestamped. Also in this case, a particular function available on Nordic devices has been used: PPI, Programmable Peripheral Interconnect. It allows the execution of a particular task at the end of a transmission event in a very precise manner, generating a standard deviation in timestamping of less than 50 ns. Of course, the timestamps collected after the radio event are sent in the next packet, hence the synchronization stage requires more than one packet to be transmitted (to retrieve all the necessary timestamps).

- Frequency Drift Estimation. As soon as all synchronization packets have been received, the contained timestamps are used to estimate the offset and the drift of the nodes clocks. This is a calculation that is performed on the node itself, and should be optimized due to their hardware constraints and available functionalities (for example, the Floating Point Unit could be missing). Researchers tried two different algorithms: Linear Regression and Average Error. Both have been tested with either 8 or 16 timestamps, sent at regular intervals, with the latest packet initiating the execution of the synchronous task. The two algorithms show different performances on the two different processors, due to the different architectures: when dealing with heterogeneous hardware, it is important to fine-tune the choice of the algorithm.

- Tick Adjustment. The last part of the mechanism consists in applying the estimation calculated before to the number of ticks necessary to reach the next point in time to execute the synchronous task: the floating-point result should be converted to an integer and then set as timer interrupt. These adjustment

require clock cycles, and their impact should be taken into consideration, for example recalculating the ticks every 100 runs.

**Conclusions** To sum it up, BlueSync is a synchronization protocol aimed at BLE Sensor Networks, compatible with commercial SoCs. Its implementation consists mainly in improving timestamping of BLE packets and choosing the most optimized frequency-drift estimation algorithm. The resulting timing accuracies, in the tests performed in the laboratory, are around 320 ns for 60 s. They assure that the system can provide good accuracy with synchronization timeslots every 10 minutes, with synchronization windows of less than 2 seconds. Even if these results are particularly good, it is important to notice that they are possible only with the functionalities available on the particular Nordic devices used, hence they may not be applicable to other SoCs on the market.

## 2.7 An application example: IMU-Enabled Posture Monitor

Paper [10] reports an interesting application for synchronized sensor nodes that is worth analyzing in this thesis: a monitoring system for sitting posture that uses wireless motion sensors attached to the user's back. Different sensors are used, and data coming from them is then fused together.

**Implementation** The article focuses on three main points:

- **Data Sensing** The required hardware needs to be small, comfortable to the user and compatible with all his possible movements. Of course, it needs to be powered via rechargeable batteries. All these requirements are the typical ones of IoT applications, and are supported by the new MEMS-based IMUs: small, low-power devices that provide all the necessary information.

  In this application, 6-degrees of freedom IMU sensors (namely, accelerometer and gyroscope) are installed on two prototypes modules that are attached to the upper and lower back of the user, parallel to their spine. The stream of these devices will then be used and elaborated to provide the spine misalignment devised into a pair of angles. Starting from this data, the system is capable of detecting spine inclination and lumbar placement in all the four directions.

  Two main problems in using IMUs in this application have been assessed: first of all, the low-resolution, noise-corrupted and biased signals coming from these kind of sensors require a quite heavy processing stage to get quality data.

Secondly, as it is only possible to detect angle changes when using accelerometer and gyroscope sensors, a calibration phase (with the user collaboration) is necessary before starting to harvest posture data.

- **Data Transmission** In this application, Bluetooth Low Energy has been considered the best solution. It has been increasingly used in wearable applications, it offers power-efficient operation and, mostly important, it is natively compatible with a lot of devices that could work as data aggregator, like smartphones.

- **Data Processing** As stated before, data coming from these IMU sensors requires some sort of refining to clean the signal and get the interesting angle information. Raw data is processed by means of a weighted moving average filter, then an Explicit Complementary Filter (ECF) algorithm is used to fuse all the information together and extract the two angles. All of this is performed on the smartphone of the user, that provides the required calculation capabilities (not available on the sensor nodes).

**User Interface** In order to use the posture monitor system, the user will first participate in a placement and calibration phase. Once it's done, the monitoring session starts, and the user is notified via vibration every time the angle threshold indicating a bad posture are violated. Once the session is complete, the user can analyze on the smartphone their posture behaviour, while the data is uploaded to a cloud-based application that can monitor the posture over time, maybe using Artificial Intelligence algorithms.

**Evaluation** To perform evaluation of the entire system, the two sensor boards have been attached on Pan/Tilt units. Then, different pre-defined couple of angles are simulated by the servo motors of the units, and then compared by the ones estimated by the IMUs. Repeated tests showed that the system is very reliable, showing errors that are distributed in a Gaussian shape and that are less than 0.1 degrees. The paper explains that observed errors are caused by the intrinsic error noise, and that the IMU calibration procedure is very useful to reduce biases, misalignment and gain errors.

**Conclusions** The article showed how data coming from low-cost sensor nodes, with limited computational capabilities and restrictive power constraint, can be quite precise when dealing with medical applications. Of course, there is the need of extensive post-processing work, usually performed by algorithms that run on more powerful devices, but the result is an user-friendly application that can be used by anyone.

## 2.8 Used Hardware

The following sections will overview the characteristics and the potential of the hardware that has been used in this thesis project. In particular, seven commercially-available boards have been employed: two SensorTile.box devices from ST Microelectronics, two BlueTile boards from the same vendor, and three MetaMotionS from MbientLab. The included sensors, microcontrollers and the available APIs for each one will be listed and explained here.

### 2.8.1 ST SensorTile.box

The SensorTile.box from ST Microelectronics (STEVAL-MKSBOX1V1) is a kit that contains all the necessary to start developing IoT applications based on remote motion and environmental data. In particular, the included board presents a set of ST MEMS sensors:

- Digital temperature sensor (STTS751)
- 6-axis inertial measurement unit (LSM6DSOX)
- 3-axis accelerometers (LIS2DW12 and LIS3DHH)
- 3-axis magnetometer (LIS2MDL)
- Altimeter / pressure sensor (LPS22HH)
- Microphone / audio sensor (MP23ABS1)
- Humidity sensor (HTS221)

All the data coming from these sensors can be processed with the help of the embedded STM32L4R9 microcontroller, an ARM Cortex-M4 based processor containing DSP and FPU units. All the information can be then transmitted with the ST Bluetooth processor BlueNRG-M2, that supports Bluetooth Low Energy 5.2 specifications.

The board comes with a preinstalled firmware that allows to read the sensor data via ST applications, or to log data inside an SD cart. The Android application also allows some firmware personalization, adding features like sensor fusion, audio FFT and so on. Anyway, there is no functionality that allows time synchronization of any type.

If any developer wants to modify the firmware, it is possible to connect the peripheral to the computer and write a firmware using the STM32 development environment [11]. Then, it is possible to write a firmware from scratch or to start

Figure 2.1: Image of the SensorTile.box, with the included case.

from the available firmware examples (from GitHub or the ST website). The developer can then decide if they wants to keep compatibility with the ST applications or not, as they have total freedom on the board behaviour.

ST offers an extensive set of Hardware Abstraction Layer (HAL) procedures, that makes it easy to develop firmware for the board and access functionality from both the sensors and the Bluetooth Low Energy stack.

Once the firmware is ready, the microcontroller can be flashed by means of ST solutions (ST-Link) or via an USB cable. The STM32CubeProgrammer [12] is the ST software that allows the programming.

To conclude the overview, the kit contains a long-life USB-rechargeable battery and a case that provides water and dust resistance.

### 2.8.2 ST BlueTile

The BlueTile from ST Microelectronics (STEVAL-BCN002V1B) is a development board containing a full set of MEMS sensors connected to a BlueNRG-2 SoC, that allows Bluetooth Low Energy connectivity. Here's the list of the embedded sensors:

- BALF-NRG-02D3: ultra-miniature balun and harmonic filter
- LSM6DSO: iNEMO 6DoF inertial module, ultra-low power and high accuracy
- LIS2MDL: magnetic sensor, digital output, 50 Gauss magnetic field dynamic range, ultra-low power, high performance, 3-axis magnetometer

- VL53L1X: long range Time-of-Flight sensor based on ST FlightSense technology
- MP34DT05TR-A: MEMS audio sensor omnidirectional digital microphone, 64 dB SNR, -26 dBFS sensitivity, top-port, 122.5 dBSPL AOP
- LPS22HH: ultra-compact piezo-resistive absolute pressure sensor, 260-1260 hPa, digital output barometer, full-mold dust resistant, holed LGA package (HLGA)
- HTS221: capacitive digital sensor for relative humidity and temperature

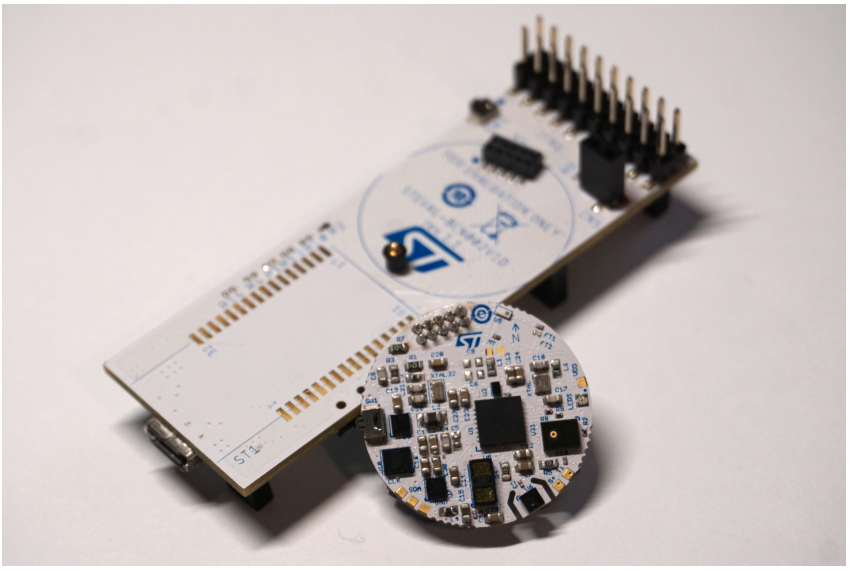


Figure 2.2: Image of the BlueTile, with its programming board.

To perform elaboration and transmission of the resulting data, the embedded SoC is a network processor for Bluetooth Low Energy that allows its own Cortex M0 core to be programmed by the user. In this way, no overhead is present, as one chip is enough to perform all the necessary operations for the board. Also in this case, ST released examples and low-level drivers, allowing the board to be programmed at high-level under both the BLE connectivity and sensor configuration aspects.

The kit also contains a programming board, to which the small BlueTile can be connected: then, it can be programmed by means of a USB cable or a programming interface from ST (ST-link or ST NUCLEO board), with the ST RF-Flasher software [13].

The board can be powered via an USB cable, when connected to its programming board, or via a CR2032 battery. ST advertises the sensors and the microcontroller to be battery-friendly, thanks to some hardware optimizations and their enhanced sleep states.

### 2.8.3 MBIENTLAB MetaMotionS

While ST solutions give a lot of freedom to the final user, making it easy for them to develop and personalize the firmware of the boards, MBIENTLAB solutions go in a different direction, offering devices with a complete set of sensors that are ready to use. In my case, I had at my disposal three MetaMotionS, containing the following set of sensors:

- Bosch BMI270 6-Axis Accelerometer/Gyroscope
- Bosch BMM150 3-Axis Magnetometer
- Bosch BMP280 Digital Pressure Sensor
- Lite-On LTR-329ALS-01 Ambient Light Sensor
- I/O analog and digital pins are present for expansion purposes.

Figure 2.3: Image of the MetaMotionS devices, with and without the clip accessory.

All the sensors are connected to a Nordic Semiconductor nRF52840 BLE SoC, which is the network chip whose Cortex-M4 processor allows heavy data elaboration (including 9-axis sensor fusion algorithms). The sensors information can be transmitted via both Bluetooth Low Energy protocols and USB. The provided firmware is closed source and can't be modified.

The device comes in a water-resistant case enclosure that can be put in (optional) wearable accessories, like clips and wristbands and it contains a USB-rechargeable battery. The mobile application available, MetaBase [14], allows a

complete configuration of the sensors, that can be turned on/off and configured in their data rate and precision. Once everything is set-up, the application can start the data recording, that can be streamed directly on the mobile device or saved in the 512 MB internal memory, to be downloaded later.

The most interesting part of these products are the open-source APIs that are available [15], making the devices easily configurable and usable from any BLE-supported computer or smartpoone with a reduced number of lines of code. In particular, they are available for Android (Java), iOS (Swift), Linux (Javascript, Python), Windows (Python) and already include all the necessary libraries and procedures for Bluetooth connectivity. A C++ library is also available, to be adapted to other operating systems and their Bluetooth stacks. The APIs are well-documented on their website, while their GitHub profile contain a lot of examples ready to be personalized to the user needs.

To conclude this overview, it is important to add that these boards firmwares already contain a sort of synchronization mechanism. All the data coming from them is precisely timestamped with a UNIX Epoch time, in particular with a millisecond resolution.

# Chapter 3

# Materials and methods

After analyzing some previous works on the topic and taking a look at the hardware at disposal, in this chapter, I present the work I carried out to develop the final system, composed of all the sensors streaming data synchronized in time. In particular, the focus is on:

- Firmware development for the ST sensor nodes
- Python client development for the computer hub
- MATLAB script development for data processing

Figure 3.1 is a block diagram to represent the entire system I developed. Each device, represented by a coloured rectangle, is shown with the software/firmware components built for it. Figure 3.2 is another block diagram, that represents (in an higher abstraction level) the flow of operation of the system. Sensor boards can recognize motion, temperature, pressure and other data, and send them via BLE to a Linux Computer acting as the client. On the client, a Python script is responsible of fetching data while keeping data synchronized in time, while some MATLAB scripts can reorganize all the data, represent and analyze them.

The chapter contains the implementation details of the solution, and the tools used for the process are reported at the end of it .

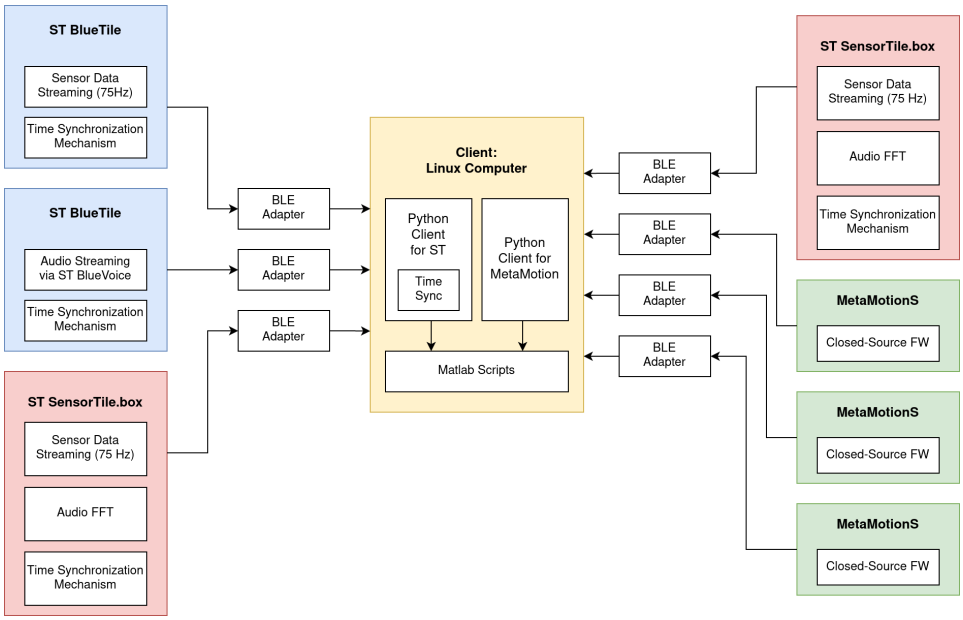



Figure 3.1: Block diagram representing the developed system.

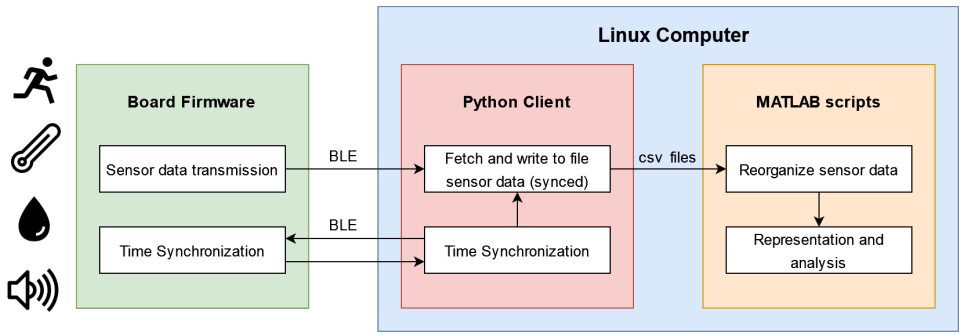



Figure 3.2: Block diagram representing the flow of operation of the system.

## 3.1 Firmware for ST devices

First of it all, I needed to replace the original firmware on the ST boards, applying the required modifications (for example, the data rate). Examples and documentation on the Internet have been a good starting point: then, thanks to a step-by-step approach, I wrote and built the final firmwares for all the boards, keeping in mind both the desired functionalities and the devices limitations.

### 3.1.1 ST BlueTile, sensing firmware

The first board I've worked on has been the ST BlueTile one, because of its simpler configuration. In particular, I used the original firmware, called *SensorDemo* [16],

as a base for further development: in this way, the compatibility with the ST application is maintained.

The behavior of the base software is explained in Figure 3.3. It consists in a main loop, on the left part of the flowchart: at each execution, the value of some flag variables is checked and the corresponding event is executed. For example, if the variable *imuTimer_expired* is 1, it is reset and the value of the motion sensors is sent to the Bluetooth Low Energy stack for transmission. The value of these flag variables is set asynchronously by some interrupt procedures (in rectangles), that are triggered by specific events. For example, a set of hardware timers (Env Timer and Imu Timer) is configured to publish sensor and environmental data at specific rates, while other procedures will manage the behaviour at Bluetooth connection/disconnection.

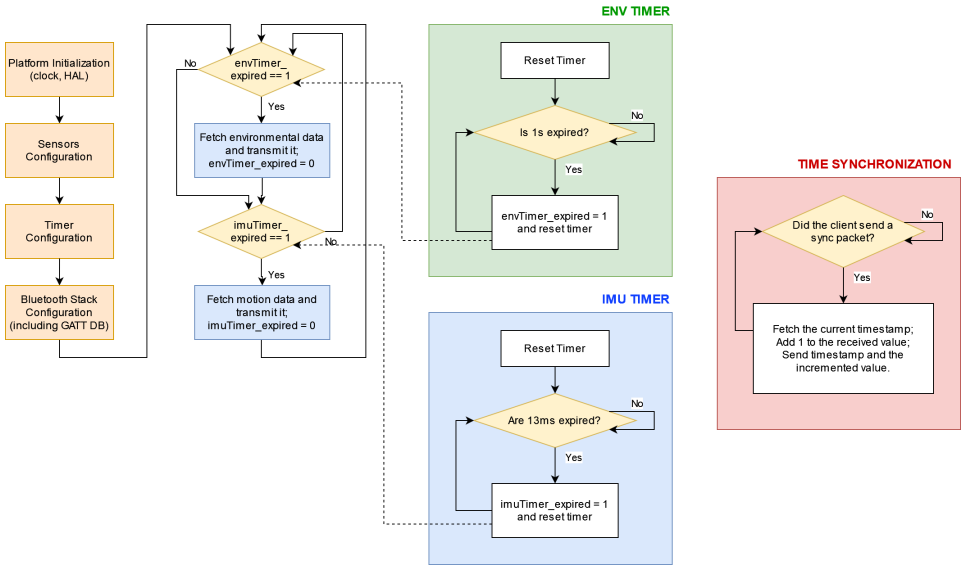



Figure 3.3: Flowchart of the firmware for the BlueTile that streams sensor data.

The *SensorDemo* firmware also contains the *ST BlueVoice* library, that allows the streaming of compressed sound via Bluetooth Low Energy, optimized for voice transmission. I'm explaining the firmware functionalities in the following list, underlining the differences from the original one:

- **Sensor data readings**. The information chosen to be transmitted to the client are the IMU (accelerometer, gyroscope, magnetometer), pressure, humidity and temperature sensors.

- **Data rate modification**. I modified the rate at which IMU sensor data is transmitted over Bluetooth from 25 Hz to 75 Hz (in the last version of the firmware). Some intermediate tests have been done with other frequencies,

in order to find the best one, with the corresponding results reported in the Results chapter. To perform this task, I modified the intervals of the timers which trigger sensor data updates accordingly. The rate for environmental sensor data has been lowered from 2Hz to 1Hz.

- **Timestamping**. The sensor data is timestamped at the moment in which the sensor is read, and immediately before sending the packet to the BLE stack. The *getTimestamp* function returns the number of milliseconds elapsed since the board power up.

- **BlueVoice disabled**. Even if no audio streaming is enabled, leaving *BlueVoice* enabled on the small Cortex-M0 processor of the BlueTile causes some important performance degradation in sensor data streaming. This is due to some interrupts that are remaining active for audio data processing and, eventually, activation of the audio streaming functionality. For these reasons, I've commented out all the code relative to *BlueVoice* initialization and processing.

- **Connection parameter update**. In order to enable the maximum possible throughput, as explained in the Background part, I set the Connection Interval parameter to 7.5ms. This value is the same one configured on all the other devices.

- **Timer modification**. When battery-operated, a timer will shut down the device if no connection is performed before 20 seconds, for battery life purposes. As this is not useful for our application, I disabled the relative timer.

- **Time synchronization mechanism**. As the time synchronization system requires two-way communication between the server and the client, the GATT database of the device should be set accordingly: I therefore added a GATT characteristic that can be read and written. Moreover, I re-wrote the interrupt procedure that is called when the GATT characteristic is written, to implement the synchronization behaviour that I explain later.

### 3.1.2 ST BlueTile, audio streaming firmware

While one of the two BlueTile board has been used to only stream sensor data at the maximum data rate possible, the other one has been used to only perform audio streaming, to take advantage of the onboard microphone and the *BlueVoice* library from ST. Also in this case, I used as starting point the *SensorDemo* firmware example, that already contains the library.

In this case, the resulting firmware is quite similar to the original one, with the modifications explained in the flowchart in Figure 3.4 and in following list:

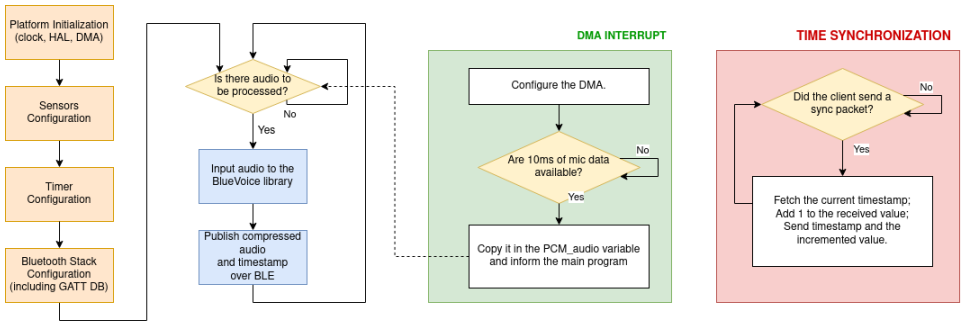



Figure 3.4: Flowchart of the firmware for the BlueTile that streams audio data.

- **Sensor Data Streaming**. I left the data rate of the sensors to the default value (25Hz). There is no necessity to fully disable the functionality from the firmware, because when voice streaming is active, the sensor data streaming is automatically disabled.

- **DMA configuration**. The microphone on the BlueTile board outputs data in the Pulse-Density modulation form (PDM): the frequency of ones is proportional to the sound pressure level. Usually, audio is represented using a different standard, called Pulse-Coded modulation (PCM), composed of 16 bits words ranging from 0 (no pressure) to $2^{16}$ (maximum pressure). This conversion can be performed in hardware using the Direct Memory Access (DMA) component of the BlueTile SoC, that can asynchronously update the value of a variable (containing audio data) and then inform the main program through an interrupt. Hence, the DMA should be configured in the initialization phase to perform this operation.

- **Timestamping**. The *BlueVoice* library has been thought to stream audio information in real time, hence there was no need to timestamp the audio data. In my case, as this data is necessary for further analysis, there is the need to provide timestamps along with the audio samples. To do so, I added a GATT characteristic on which timestamping information is published. I also modified one of the *BlueVoice* callback procedures, in particular the *SendData* one, responsible of sending out samples once processed: just before publishing the audio data on its own GATT characteristic, the current timestamp is published on the other one. The client will then join them together.

  I put the implementation of this system in the separated *timestamp_helper.c* file, that contains the variables (GATT characteristic handle) and the two procedures (GATT characteristic creation, timestamp publishing) necessary for all of this to work.

- **Timer modification**. As done for the other firmware, I disabled the timer

that automatically turns off the device after 20 seconds of inactivity if the device is battery-operated.

- **Time synchronization mechanism**. Also in this case, I modified the GATT database to introduce the time synchronization mechanism.

### 3.1.3 ST SensorTile.box firmware

The SensorTile.box presents some differences from the other device from ST. Even if the software stack is similar, there is no possibility to program the BlueNRG microcontroller, that is controlled by the STM32 Cortex-M4 processor, the only one accessible by the developer. This difference unlocks a great number of other possibilities, thanks to the computing power and the STM32 libraries, but the firmware development becomes a little more complex.

The definitive solution consisted in using the *ALLMEMS-1* firmware example from ST [17], modifying it according to my needs. In this case, this software has been the best choice, as other ones didn't include support for some sensors like the microphone. It is worth noticing that the SensorTile.box presents no support for the *BlueVoice* library, hence it is possible to extract information from the audio stream, but not to send it real-time to the client.

Finally, regarding the BLE stack, it is interesting that its abstraction layer is providing C functions working in the same way as they're working in the BlueTile, even if the hardware is quite different. As a consequence, full compatibility with parts of BlueTile firmware is assured: GATT characteristic creation/update, connection configuration, callback procedures all work in the same way. The final firmware is represented in Figure 3.5.

The following list explains the functionalities of the firmware and the work I've done on it:

- **Sensor data streaming**. The information chosen to be transmitted to the client are the IMU (accelerometer, gyroscope, magnetometer), pressure, humidity and temperature sensors.

- **Data rate modification**. Like what's has been done on the BlueTile, I modified the motion sensors data rate to 75Hz and environmental sensors data rate to 1Hz, setting accordingly the timer variables.

- **Timestamping**. The timestamping on the SensorTile.box firmware had a different behaviour from the BlueTile one. It would still send the number of milliseconds elapsed from power on, but the three LSBs were shifted out and not sent to the client. To create consistency between devices and add precision (we need time to be as accurate as possible), I disabled the shifting, to transmit the entire timestamp.

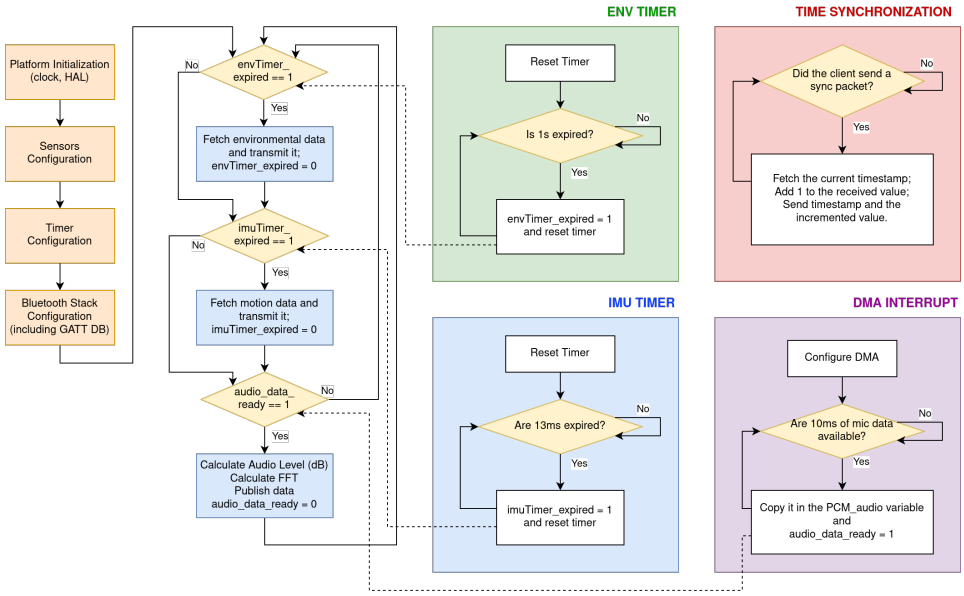



Figure 3.5: Flowchart of the firmware for the SensorTile.box.

- **Time synchronization mechanism**. Also in this case, I modified the GATT database to make the time synchronization mechanism possible.

- **Audio processing**. One of the interesting features of the SensorTile.box is the presence of a digital microphone. The ALLMEMS firmware from ST leverages this component offering the streaming of audio level information (in dB). To enrich the details on the surrounding audio, I added the information on the dominant frequency: the samples used to extract the dB information are also input for the *arm_rfft_fast_f32* function from the Common Microcontroller Software Interface Standard (CMSIS) from ARM, that is a a vendor-independent abstraction layer for microcontrollers that are based on ARM Cortex processors. For my use case, it provides an easy way to take advantage of the DSP included in the Cortex-M4 core available on the board. After getting the discrete real Fast Fourier Transform (FFT), the maximum magnitude for each output value is extracted, and the corresponding frequency index is sent on its own GATT characteristic, that I configured before. I would like to underline that, also in this case, the configuration of the DMA has been necessary (as explained in the BlueTile chapter).

- **Removing undesired functionalities**. The ALLMEMS firmware contains a lot of libraries from ST, used for sensor fusion and feature extraction (activity, gesture, position recognition and so on). To save battery, space and computational power (the FFT operation is pretty intensive), I disabled all these functionalities by commenting out their initialization code lines, along

with the relative callback functions.

## 3.2 The Time Synchronization Mechanism

One of the most challenging parts of this thesis has been the development of a time synchronization mechanism for the ST devices, as MetaMotion ones already embed their own closed-source implementation. Three main issues should be overcome to do so: first of all, the limited computational power available on the sensor nodes (in particular, the BlueTile ones), the need to implement this system at the higher application layer of the BLE stack (as lower levels are not accessible by the developers), and the need for a compromise between time accuracy and high data rates.

The main idea is to leverage on the client, that is a Linux computer running a Python script and a fundamental part of our sensors system. It can be used to provide a reference clock for all the data coming from the sensor nodes. To reduce the number of exchanged packets, an application similar to the receiver-to-receiver one could be used, in which a reference broadcasts messages simultaneously to all devices, and then they can use their reception timestamps to synchronize.

In particular, the system I implemented consists in the client sending packets to all devices at regular intervals, using the *WRITE_NO_RESPONSE* property of a GATT characteristic specifically created for this scope on the sensor nodes. That particular property is the fastest way to send data to a BLE device, as no response is necessary and only one packet is sent. As soon as the remote device receives the packet, the relative callback procedure is automatically called by the BLE stack of the node: in that function, the packet is sent back with its original content (a number) incremented by one, and the internal timestamp.

The client will then receive that information, and associate the received timestamp to the current Unix epoch time expressed in milliseconds using the *time* library of Python 3.10, which returns a precise value (while previous versions of Python weren't able to do so). Then, the successive samples will be timestamped using the available information: 1) the UNIX time obtained in the resynchronization moment $t_{UNIX}$, 2) the timestamp of the resynchronization moment $t_R$ and 3) the device timestamp of the sample under analysis. $t_S$. The result is calculated with Equation 3.1.

$$T = t_{UNIX} + (t_S - t_R) \tag{3.1}$$

The resynchronization procedure can be executed at any desired interval by simply modifying the value in the Python script of the client. In the experiments explained in this thesis, the interval is set to 1s, which was able to give good performances when keeping a data rate of 75 Hz. More details in the Results section.

## 3.3 Client for ST devices

The client for the ST sensor board is a Python script for Linux hosts that handles connection, data collection and time synchronization with all the sensor boards from ST. Of course, the client is compatible with the firmware that has been developed for this thesis only, as the time synchronization and other details do not follow any existing standard.

I'm reporting more details on the client structure and behaviour in the following list and flowchart (Figure 3.6).

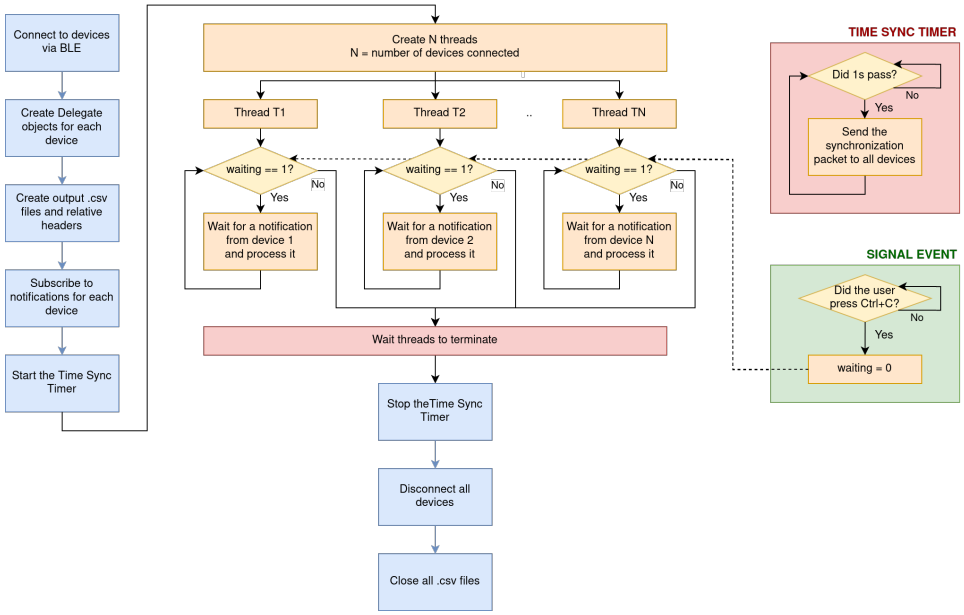



Figure 3.6: Flowchart of the client for ST sensor nodes.

- **Connection**. In the first part of the script, I used the *BluePy* library for Python to perform connection to all of the devices. All the information necessary for the connection are reported in a list of dictionaries, each one presenting a) the device name, b) the device MAC address, c) the device type and d) the interface to use for connection. The device type is necessary for the client to understand which features are available (for example, audio dominant frequency is available for the SensorTile.box, only). Also the interface is quite important for our use case, because using multiple Bluetooth adapters leads to better performances (more details in the Results part). It is possible to connect to multiple devices using only one client.

  Once the connection to a device is performed, an object, inherited from the BluePy library, is created for each device. It is called a Delegate, and it contains information for the connection to that device: it is fundamental in

our environment, where more than one device is connected. In particular, in this implementation the Delegate holds a) the name of the device; b) the handles used to recognize the type of received data (IMU, environmental, time, audio...) c) the file handles, to write the data into, d) the procedure to be called each time data is received from the device.

- **Notification handling**. As soon as connection is performed and all Delegates are ready, the clients *subscribes* to the GATT characteristics it is interested to. This procedure varies, based on the type of the device the client it's working with. For example, in the case of the BlueTile (sensing firmware), the client subscribes to IMU sensors, environmental sensors and time synchronization characteristics.

  After the subscription, each time the sensor node sends data on a particular characteristic, the *handle_notification* function of the Delegate is called. It will compare the handle with the ones saved in it, understand the type of data received and behave accordingly: in case of sensor data, it is further processed (to use consistent measurement units with MBIENTLAB devices) and written to file, while if it is time synchronization data, the received timestamp and the immediate Unix epoch are saved in the Delegate and used to process successive packets.

  It is important to add that the BluePy library makes use of the *wait_for_notification* blocking function, that waits for notification from the precise device it is called from. As we are dealing with more than one device, it is necessary to create different threads, one per node, and then call the function in them using a while condition. As soon as the while condition becomes false, the thread will terminate.

- **BlueVoice data handling**. In case the ST client is connected to a device of type *bluevoice* (i.e. a BlueTile with the audio streaming firmware), the behaviour is the same of the other ones, but the data received from the node is (of course) treated differently.

  First, timestamps are received from a different GATT characteristics, and should be joined to the audio samples packet. This is performed by using FIFOs lists for both data.

  Second, data is received compressed, using the ADPCM codec. It means that 16B samples are reduced to 4B samples, that contains only the differences from the previous set of samples. As some packets may get lost, the reconstruction process may not work on the long period: to solve this, the BlueVoice library also sends the so-called sync packets (at lower frequencies). Those contain uncompressed data on which the difference packets can be applied to reconstruct the original signal even if some segment went lost. All

this information is processed inside the corresponding functions, *process_sync* and *process_audio*, that are released by ST in their GitHub account.

- **Time Synchronization**. After completing the subscription to all GATT characteristics, the so-called *Repeated Timer* it is created. It is a small Python module making use of the *time* and *threading* libraries to generate a timer that fires at defined intervals, executing a defined function. It is important to note that it has been designed to consider the real interval between firings: the time needed to perform the instructions inside the function will not count.

  In my implementation, one timer is configured to be fired each second: in the procedure, the synchronization packet is sent to all devices. As explained in Section 3.2, those will respond with a notification, that will be managed accordingly. Other firing intervals can be configured, but smaller ones cause bigger number of packets to be elaborated and exchanged: 1 s is the best tradeoff I found between the number of lost packet and the synchronization accuracy, while 0.5 s caused big percentages of packet loss.

- **Program termination**. The Ctrl+C signal has been configured to call a procedure that will change the value of the *waiting* variable. That variable is the one that keeps the notification waiting threads alive. In this way, the threads can terminate and join to the main program, that will disconnect from all nodes, close the files and terminate.

- **BluePy modification**. The BluePy library suffers from a bug that generates exceptions if a GATT characteristic is written while there are multiple threads waiting for notifications. In particular, an object indicating the correct transmission of the packet should be returned to the *writeToCharacteristic* function, that will terminate afterwards. Sometimes that object is received by one of the *wait_for_notification* procedures, that are not expecting it, generating an exception. To resolve it, I implemented a workaround, consisting of a) making *wait_for_notification* ignore that particular object when received and b) forcing *writeToCharacteristic* to return without waiting for the confirmation object. This does not seem to cause problems in the resynchronization procedure, and we can be confident that the packet is sent by the BLE stack without the need for confirmation. Anyway, this should be considered as only a workaround, because the problem should be resolved by re-thinking this part of the library (confirmation of the successful transmission is quite important in any case).

## 3.4 Client for MBIENTLAB devices

I found writing the client for MBIENTLAB MetaMotionS much simpler, as the available APIs and examples are easy to understand and implement. There is no need to implement any time synchronization mechanism, as it's already available by default. Starting from the available Python example, available from the vendor's Github page [18], that is performing the streaming of accelerometer data, I wrote the final version of the script with the following features, with the flowchart represented in Figure 3.7.

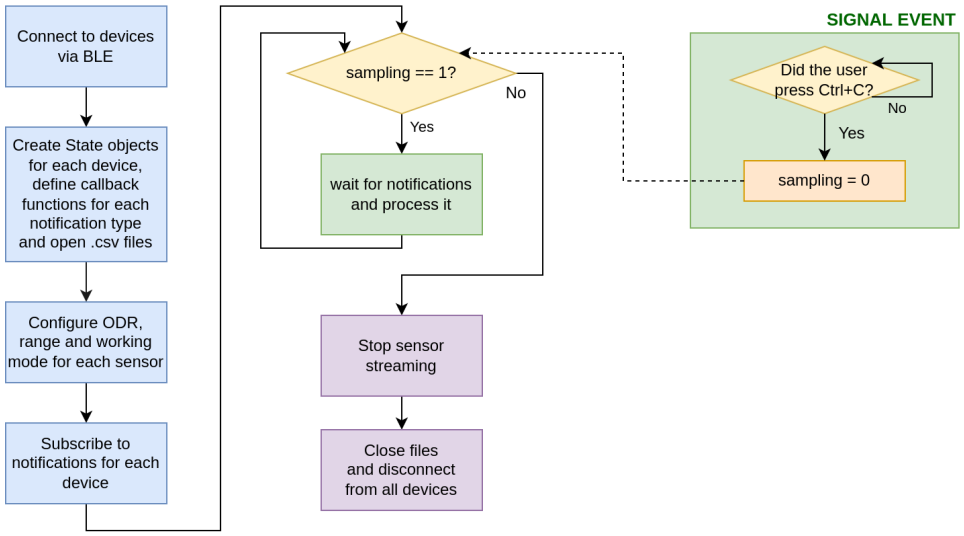



Figure 3.7: Flowchart of the client for MBIENTLAB sensor nodes.

- **Connection**. Also this script support the connection to multiple devices, represented in a list of dictionaries, using multiple Bluetooth adapters. For each sensor node, it is reported a) a name, b) the MAC address of the device and c) the MAC address of the interface to use for connection. The connection is performed by the APIs, so no need to use BluePy (or similar libraries).

  After the connection, similar to the script for ST nodes, a set of objects, called States, is created. Each of them is responsible for one device, and contains a) the device name, b) the pointers to the functions to be called when data is received and c) the handles of the files in which data is stored.

  The functions are used to take the data from the received packets and save it inside .csv files correctly formatted. To do so, the function *parse_data*, from the MBIENTLAB APIs is used to get a string containing all data: a regular expression will extract the interesting values from the string and write them into the file.

- **Configuration**. In the configuration phase, each sensor is configured in its ODR (Output Data Rate), power mode (low power or high accuracy and so on) and range. The sensors configured are IMU (accelerometer, gyroscope, magnetometer), temperature and pressure. This procedure is repeated for each sensor node. At the end, the data collection is started.

- **Data collection stop**. To perform the data collection, I used the Python *sleep* function to make the client wait for notifications. Of course, in the meantime all the received data is processed by the functions pointed inside the State objects. The sleep function, configured to wait for 1s, is repeated until the *sampling* variable remains True: as soon as the Ctrl+C signal makes it become False, the sleep will not be repeated and the Python script will continue its execution. The last lines of code will stop the sampling, reset the sensors, close all the files and print a sum up of the received information.

## 3.5 MATLAB data extraction scripts

Thanks to the ton of already available libraries and functions for data processing, MATLAB has been the first choice when dealing with the elaboration of all the sensor information retrieved from the data collection campaigns. Transforming CSV values into MATLAB data structure is immediate (the *readtable* function is enough), and then the Signal Analysis Toolbox contains an extensive set of tools to work on those values with ease. In particular, I prepared the following scripts:

- **Data Extraction for ST devices**: works with Acceleration, Gyroscope and Magnetometer data. It reads all the values, and plots them over time using the MATLAB plotting feature. Then, all the data is modified for the Signal Analyzer application: the timesync set of values is transformed into a square wave, to represent the moments in which the re-synchronization happens, then all the signals are coupled with their timestamp generating *timeseries* data structures. Those timeseries are further analyzed, to order them in the time domain and remove duplicates. In this way, the signals are ready to be shown and elaborated with MATLAB toolboxes.

- **Data Extraction for Meta devices**: it behaves like the one for ST devices, but takes into account the slight differences of CSV data generated by the MBIENTLAB ones. Then, everything remains the same: plots are shown, *timeseries* are generated and elaborated. In particular, the timeseries variables have been called in a different way, in order for them to be distinguished from the ST ones.

- **Data Extract for Voice**: the goal of this script is to reconstruct the audio signal from the CSV file generated by the BlueTile with BlueVoice enabled. As

the timestamping mechanism is not as precise as the one used for sensor data (due to the closed-source BlueVoice library), only the timestamps generated at resynchronization events are used, and all the samples in the middle receive regularly intervalled timestamps between the two events. Reducing the resynchronization interval could be advantageous in this case, but the number of packet loss could be higher, making the audio quality sensibly lower. Also in this case *timeseries* structures are created and elaborated to be shown afterwards in the Signal Analyzed application.

- **Data Extract for Frequencies**: it works only with the frequencies in the CSV generated by the SensorTile.box device. It behaves like the acceleration/gyroscope script, but works with frequencies, that are transformed from the indexes between 0-128 into the real frequencies values, and then scaled by a factor of 1000: this is necessary to make them show in the same plot of the acceleration values, for example. Also in this case, everything is modified to be shown correctly in the Signal Analyzer application.

## 3.6 Calibration Script

An important step when dealing with accelerometer and gyroscope data is taking care of axes alignment. For example, if the application involves installation of the sensors system on a car, it is important for the boards to be positioned in a coherent way: all the sensor chips have their own axes orientation, and it's a good idea to align them to the vehicle ones.

A problem may arise if the final user puts the sensors in a wrong way, rotated with respect to the desired direction. In this case, the obtained data should be post-processed accordingly, to get reasonable results: in particular, a rotation matrix should be calculated and then multiplied to the raw acceleration data.

I had at my disposal an algorithm, wirtten for MATLAB, able to calculate the roll, pitch and yaw angles, developed by the research group I'm working with for this thesis (Electronic Design Automation Group) and optimized for embedded systems application: this algorithm iterates over GPS and IMU data, performing two kinds of calibration, explained as follows.

- **Static Calibration**. Working on a subset of samples, the available data is analyzed to verify if the vehicle the sensor is mounted on is static. If this is the case, the subset of samples is passed to a dedicated procedure able to extract two angles: alpha (roll) and gamma (pitch).

- **Dynamic Calibration**. Also in this case a subset of samples is analyzed, but here the algorithm verifies if the vehicle is proceeding in a straight path. This step is necessary because, if that condition is met, the buffer of samples

undergoing this condition is elaborated, along with the alpha and gamma angles: the output is the beta (yaw) angle.

Once the three angles have been calculated, it is possible to generate the three rotation matrices to be applied to the motion data $m_\alpha, m_\beta, m_\gamma$ as represented in Equations 3.2, 3.3 and 3.4. The goal, given the reference system $R_S$ of the IMU sensors, is to achieve the alignment with the reference system $R_M$ of the vehicle, as explained in Figure 3.8.

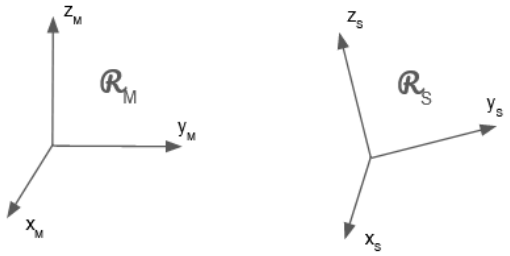



Figure 3.8: Example of sensors reference system compared to vehicle reference system.

$$m_\alpha = \begin{bmatrix} cos(\alpha) & 0 & sin(\alpha) \\ 0 & 1 & 0 \\ -sin(\alpha) & 0 & cos(\alpha) \end{bmatrix} \tag{3.2}$$

$$m_\beta = \begin{bmatrix} cos(\beta) & -sin(\beta) & 0 \\ sin(\beta) & cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

$$m_\gamma = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\gamma) & -sin(\gamma) \\ 0 & sin(\gamma) & cos(\gamma) \end{bmatrix} \tag{3.4}$$

The system I'm working on does not include any GPS sensor, hence this algorithm required some modifications in order to be useful. In particular, I modified the two functions that are responsible of recognizing the static and dynamic conditions, generating the corresponding sample subsets. In the new version of the script, these functions are no more accepting GPS data as inputs, and they verify (a) the static condition by checking if the variance of both acceleration and gyroscope samples is below a certain threshold and (b) the dynamic condition by calculating the variance of the gyroscope samples and comparing it to a threshold. Then, the calculation of the angles remains as before, because that step requires the IMU samples only. Some work was necessary to correctly fix the thresholds

for the variance values, that have been decreased from the original values to reach more precise results in calibration.

In Figure 3.9 the flowchart of the algorithm is reported. The two different kind of calibration are reported in different colors.

It is evident that the calibration is performed in an iterative way, by repeating the static and dynamic calibration over different chunks of samples (moving the *start* index). Each cycle will generate different values for the angles, and at the end the median value is calculated and shown as output. The two calibration procedures will take the sample corresponding to the *start* index, and then add values (one second of samples per cycle) until the variance of the buffer overcomes the threshold value. In this case, the last second of samples is flushed (because the condition is no more met), and, if the remaining ones represent the static/dynamic condition for at least a certain number of seconds, those IMU samples are used to perform calculation of the required angles (alpha and gamma for static, beta for dynamic).

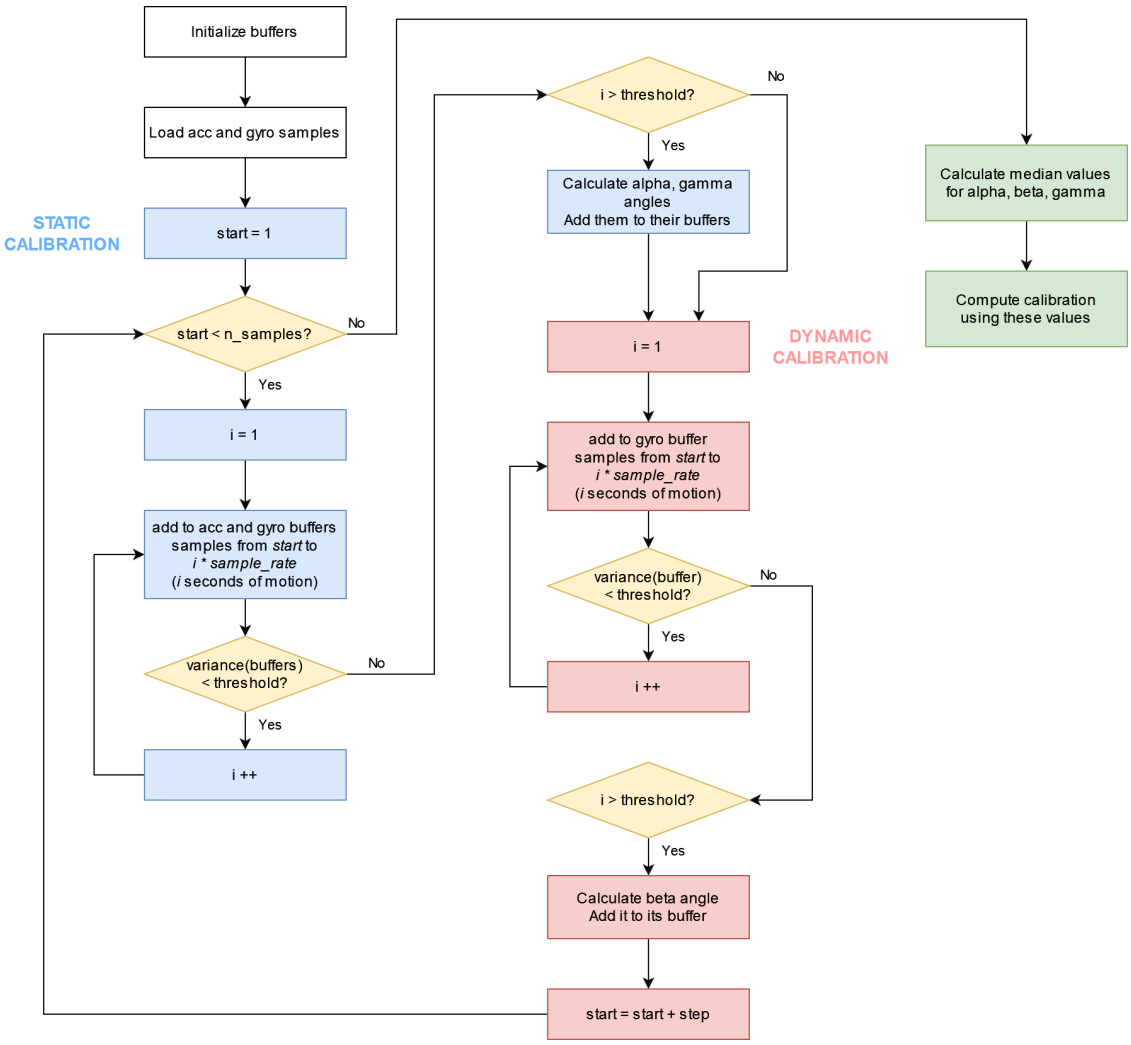



Figure 3.9: Flowchart of the script that perform calibration of the axes.

## 3.7 Used tools

The development part of this work has been carried out using professional tools currently used in the field. In particular:

- Firmware Development: uVision IDE from KEIL for BlueTile, STM32CubeIDE (based on Eclipse) from ST for SensorTile.box
- Client Development: PyCharm IDE from JetBrains
- Data Processing: MATLAB
- Version Control System: GIT

# Chapter 4

# Results and discussion

Finally, I performed some tests and experiment with the developed system to verify if it is working properly and whether the original requirements are met.

The first experiment had the goal of finding the best data rate at which the boards can send samples with an acceptable loss of information (due to lost packets). I tried different configurations of sensors and Bluetooth adapters to find the best working condition.

The second one tested time synchronization performances: I moved all the sensors coherently, and analyzed the delays between the signals, comparing the results from the ST devices (with my version of the time synchronization) with the ones from MBIENTLAB.

The last test is the most extensive one, as I mounted the system on a car and tested the data sampling for around 1 hour. It has been a good benchmark for system reliability and time synchronization. At the same time, I used all the data generated during this experience to test some further elaboration, including calibration of the axes and event labeling through audio.

## 4.1 Maximum data rate on Bluetooth Low Energy

One of the first experiments done for the entire system has been a simple test to determine the maximum speed for the data to be transmitted via Bluetooth Low Energy. In particular, the ST devices have been programmed to only transmit a number that is incremented at each transmission, instead of the accelerometer data. The number of packets to be transmitted is fixed, and corresponds to the number of packets to transmit in 5 minutes. For example, if the data rate is set to 25 Hz, the number of packets is limited to: $25 * 60 * 5 = 7500$. Once all the expected packets have been received, the number in them is used to understand how many packets have been lost.

At the same time, as MBIENTLAB devices have closed source firmware, the estimation on them is done by making them stream data for 5 minutes: the number of effectively received packets is compared to the expected number of packets, and then the percentage is calculated.

Different tests have been performed, using a different number of connected devices and Bluetooth adapters. In particular, the dongles at my disposal have been:

- 1x Intel internal Bluetooth Adapter.
- 6x ASUS BT-400 Bluetooth 4.0 USB Adapter, based on the Broadcom BCM20702 chip.

For both ST and MBIENTLAB devices, values in the tables have been calculated with Equation 4.1:

$$loss(\%) = (1 - (\frac{received\_packets}{expected\_packets})) * 100 \tag{4.1}$$

- Table 4.1 reports the experiment of two devices, a BlueTile and a SensorTile.box, connected to one ASUS BT-400 adapter.
- Table 4.2 presents the test of three devices, a SensorTile.box (ST.b), a BlueTile (BT) and a MetaMotionS (MM), all connected to one, two or three different ASUS-BT400 adapters.
- Table 4.3 describes the results obtained from connecting all the seven devices to different number of Bluetooth adapters (the Intel internal adapter has been used only in the case of 7 dongles). In the 75 Hz case, MetaMotionS have been configured at 50 Hz, because that particular data rate is not present in the device's APIs.

Table 4.1: Test 1. Two devices, both connected to one adapter.

| **Frequency** | **BlueTile losses** | **SensorTile.box losses** |
|---|---|---|
| **25Hz** | 0% | 0% |
| **50Hz** | 0% | 0% |
| **100Hz** | 0.14% | 0% |
| **200Hz** | 3.4% | 0.22% |
| **333.3Hz** | 42.3% | 0% |

Table 4.2: Test 2. Three devices. One, two or three adapters

| **Frequency** | **Device** | **1 Dongle** | **2 Dongles** | **3 Dongles** |
|---|---|---|---|---|
| **25Hz** | ST.b | 0% | 0% | 0% |
| | BT | 0% | 0% | 0% |
| | MM | 0% | 0% | 0% |
| **50Hz** | ST.b | 0% | 0% | 0% |
| | BT | 0% | 0% | 0% |
| | MM | 0% | 0% | 0% |
| **100Hz** | ST.b | 0% | 1.26% | 23.7% |
| | BT | 0.2% | 0.19% | 0.6% |
| | MM | 0% | 0% | 0% |
| **200Hz** | ST.b | 33.3% | 21% | 6.96% |
| | BT | 20% | 26.9% | 0.74% |
| | MM | 17.3% | 14.9% | 3,33% |

Table 4.3: Test 3. All devices. Two, four or seven adapters.

| **Frequency** | **Device** | **2 Dongles** | **4 Dongles** | **7 Dongles** |
|---|---|---|---|---|
| **25Hz** | ST.b 1 | 0% | 0% | 0% |
| | ST.b 2 | 7.6% | 0% | 0% |
| | BT | 0% | 0% | 0% |
| | MM 1 | 0% | 0% | 0% |
| | MM 2 | 0% | 0% | 0% |
| | MM 3 | 0% | 0% | 0% |
| **50Hz** | ST.b 1 | 0.73% | 0% | 0% |
| | ST.b 2 | 0% | 0% | 0% |
| | BT | 0.06% | 0% | 0% |
| | MM 1 | 0% | 0% | 0% |
| | MM 2 | 0% | 0% | 0% |
| | MM 3 | 0% | 0% | 0% |
| **75Hz** | ST.b 1 | 0.51% | 0% | 0% |
| | ST.b 2 | 0% | 0% | 0% |
| | BT | 3.7% | 1.2% | 0.06% |
| | MM 1 | 0% | 0% | 0% |
| | MM 2 | 0% | 0% | 0% |
| | MM 3 | 0% | 0% | 0% |
| **100Hz** | ST.b 1 | 41.5% | 0% | 24.18% |
| | ST.b 2 | 0% | 0% | 0% |
| | BT | 0% | 22.4% | 0.36% |
| | MM 1 | 0% | 0% | 0.25% |
| | MM 2 | 0% | 0% | 9.4% |
| | MM 3 | 0% | 0% | 6.37% |

In conclusion, it is evident that the percentage of loss packets heavily depends on the output data rate, that should be chosen accordingly to the desired application. To get higher data rates with the same number of devices, it is a better idea to find a good tradeoff using multiple Bluetooth Adapters.

Nevertheless, other issues may come into play in this situation. For example, in Chapter 2.2 I reported the results of a study, which found that the Bluetooth adapter heavily impacts the percentage of received packets, so maybe these results can be expanded by trying different dongles.

Another point is that the same test performed in different moments can cause big changes in the results. I noticed that repeating a test immediately usually returns similar numbers of losses, while repeating the same experiment the following day generated different results. A possible explanation of this behaviour can be the fact that Bluetooth uses the 2.4 GHz wireless band: a lot of other devices use it (WiFi, alarm systems). The BLE protocol includes a frequency hopping mechanism, to find the best channel for every connection event, but the Wireless traffic is usually not deterministic and prone to immediate variations, that can impact the performances of these nodes.

Also, the remote node seems to make a difference. In particular, MetaMotionS seems to perform better in these tests. As their firmware is not open-source, I can only make some assumptions to try to explain this: (1) the Nordic Bluetooth stack may act differently from the ST one, giving better results, (2) the design of their boards (and antennas) allows better transmission performances, or maybe (3) the library used by their Python APIs performs better than BluePy (that is the one I used for the ST client).

In any case, even if the percentage of loss packets may not vary that much, the usage of multiple dongles is recommended. During the tests done with 7 devices and 7 dongles, the connection procedure was fast and I never encountered failures, while using less dongles may require more time to perform the connection and sometimes it fails.

For further experiments, and for the final version of the acquisition system, I chose 75 Hz as the data rate for ST devices, and consequently 50 Hz for the MBIENTLAB ones.

## 4.2 Time Synchronization Accuracy

To validate the time synchronization mechanisms, two tests have been carried out. Both of them consist of 15 minutes of sampling, keeping the sensors steady. When the time expires, all the boards are moved in the same direction on the Z-axis, with a coherent movement. The difference between the two tests is that one of them is performed using the last version of the client, while I carried out the second test using a different version of the script that ignores synchronization packets: the

association between the device timestamp and the current UNIX timestamp is done at the moment of connection only.

ST devices are streaming data at a rate of 75 Hz, with a re-synchronization time (for the time synchronization mechanism) of 1 s. MBIENTLAB ones are set with an Output Data Rate (ODR) of 50 Hz.

The re-synchronization time has been set at that particular value because, after some tests at different intervals, I found 1 s to be the best trade-off between the synchronization accuracy and the packet loss percentage. This happens because each re-synchronization requires an interrupt procedure to be performed on the sensor boards, with an additional exchange of data between it and the client, and all of this comes at a cost from the performance point of view.

The results are presented in the following Figures, which compare the delay between the first sample of the first registered peak and the last sample of the same peak. In an ideal condition, they should be perfectly aligned.

Figure 4.1 represents the data without the time synchronization mechanism, while Figure 4.2 contains the results with the mechanism turned on. In the first case, a delay of 1.18 seconds is evident. The second case shows a better situation: 65 ms of difference, which is more than acceptable for our applications.

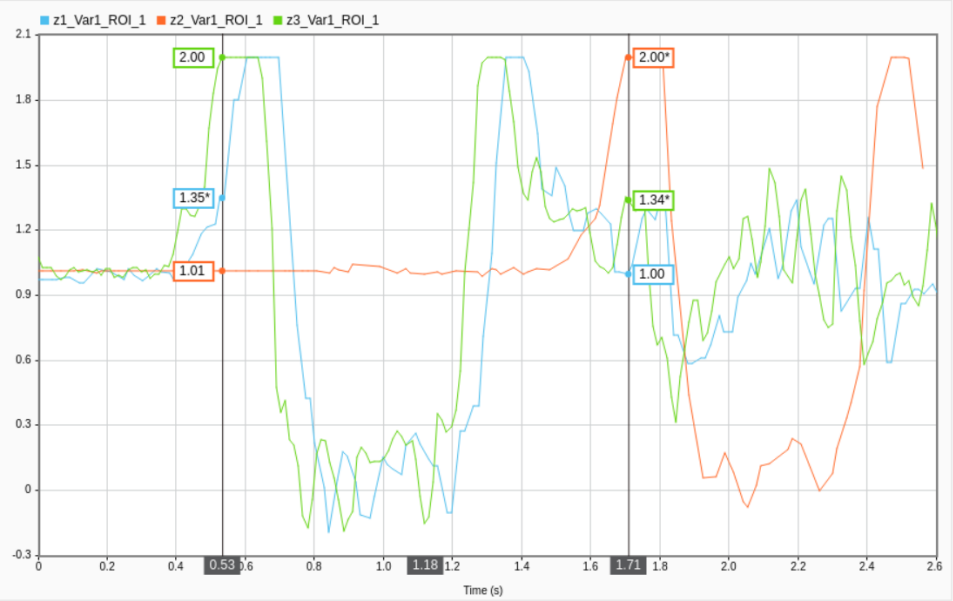



Figure 4.1: Experiment with no time synchronization: delay > 1s.

Then, in Figure 4.3 I compare the results got in the second case (experiment with time synchronization) with the ones from the MBIENTLAB devices, that include a personalized, closed-source system for time synchronization. It is easy to notice that the samples of the MetaMotionS devices (dotted lines) are better synchronized in time, with a maximum delay of 20 ms.

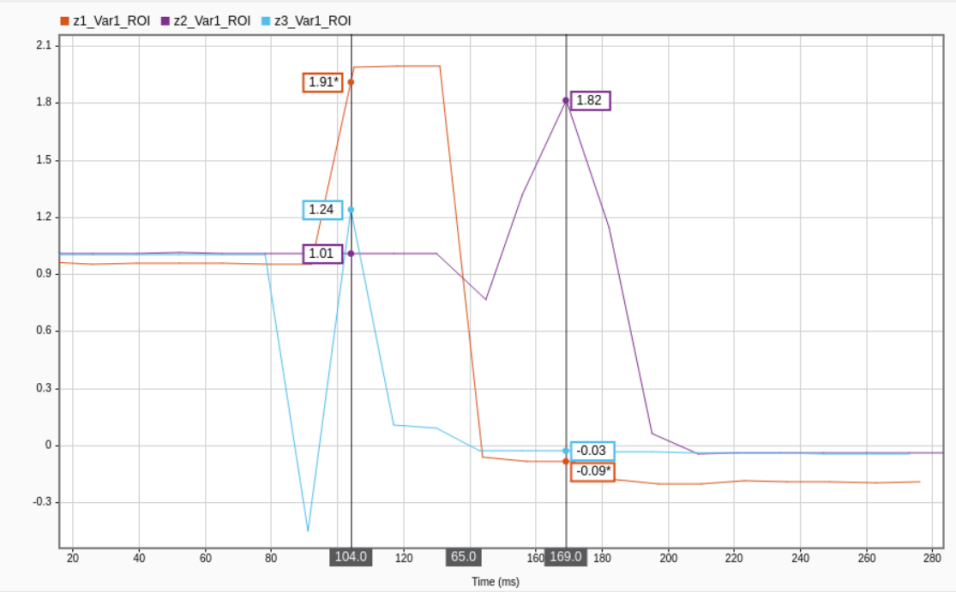



Figure 4.2: Experiment with time synchronization: delay around 65ms.

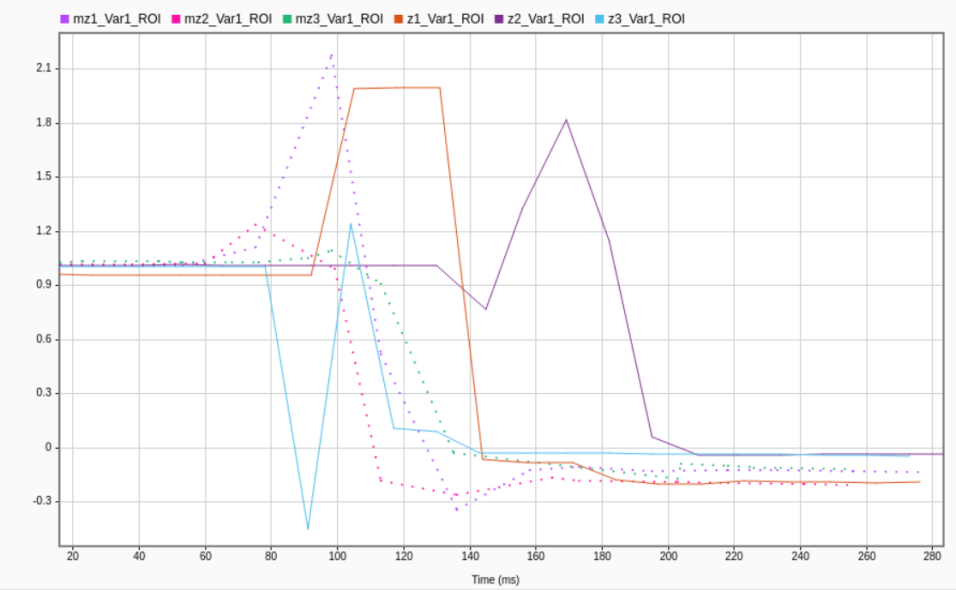



Figure 4.3: Experiment with time synchronization: comparison with MBIENTLAB devices.

Finally, Figure 4.4 reports the worst case I found analyzing the results of these experiment. In this case, the delay between the samples of the same event is 70 ms. Hence, I chose 70 ms as the precision value for this synchronization mechanism.

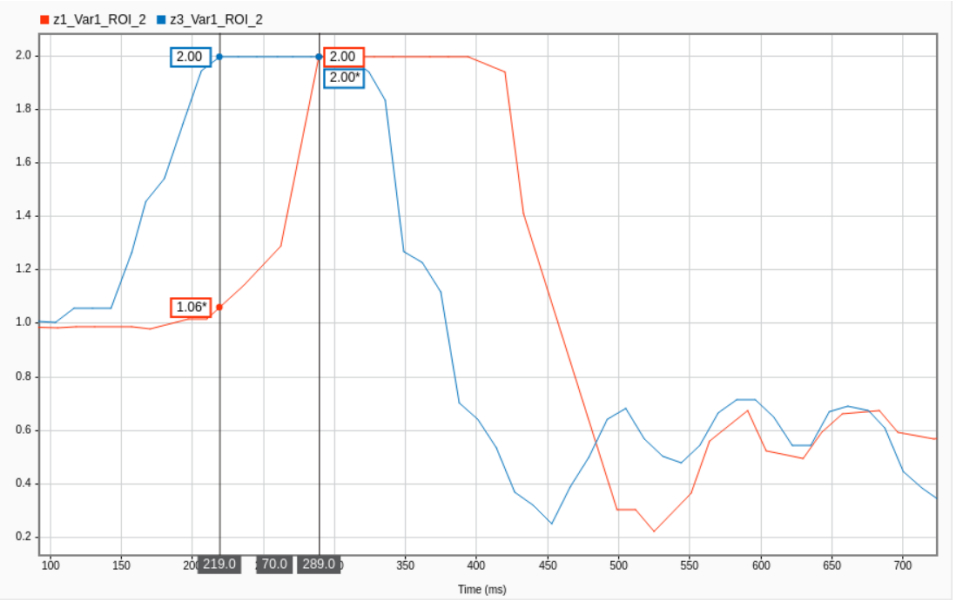



Figure 4.4: Experiment with time synchronization: worst result.

In light of these results, I can consider this implementation of the time synchronization as satisfying. The accuracy of commercial solutions is better, but my solution can be considered comparable, also considering the easiness of implementation. The benefits of including this mechanism instead of using the device timestamp only are pretty evident.

## 4.3 Extensive test in car environment

To perform a real-life use case test, we set up a complete data-acquisition system composed of:

- All the seven sensor boards of this thesis: 2 ST SensorTile.box, 2 ST BlueTile, 3 MBIENTLAB MetaMotionS.
- A SONY Action Cam HDR-AS200V, to match recorded data with video information. As the device contains a GPS module, also localization information is recorded.
- A smartphone equipped with the Physics Toolbox application, used to sample reference data from accelerometers, gyroscope and GPS sensors.
- A device from Tierra SpA with high-precision IMU sensors and a GPS module.

The data collection phase lasted for about one hour, in which we went through different paths around the Politecnico di Torino, including roundabouts, different

round surfaces and start/stop conditions. Every particular condition have been signaled with precise audio frequencies, generated by another smartphone with the Physics Toolbox application and reported in Table 4.4. Figure 4.5 shows a map of the road traveled for the experiment, while Figure 4.6 reports the setup of the sensors inside the car.


Figure 4.5: Map of the road traveled for the car experiment (OpenStreetMap).

Table 4.4: Car experiment: events and corresponding audio frequency.

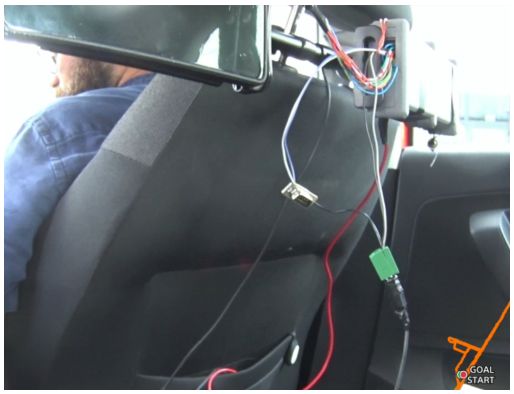
| **Event** | **Sound Frequency (Hz)** |
|---|---|
| Experiment start | 400 |
| Roundabout | 800 |
| Road bump (long) | 1200 |
| Road bump (short) | 1600 |
| Start from traffic light | 2000 |
| Braking for traffic light | 2400 |
| Right turn | 2800 |
| Left turn | 3200 |

(a) First SensorTile.box on the right of the car dashboard

(b) Second SensorTile.box on the left of the car dashboard

(c) BlueTile and BlueVoice boards on the center of the car dashboard

(d) One MetaMotion fixed on the driver's seat



(e) One MetaMotion fixed on the passenger's seat, with the Tierra device

(f) One MetaMotion fixed on the belt of a rear seat

Figure 4.6: Images of the positioning of the sensor boards in the car.

As results, I'm reporting in the following figures the graphs obtained by considering both the acceleration and the gyroscope data.

Figure 4.7 reports acceleration data coming from all the sensors. I reported the X axis for ST sensors (x1, x2, x3), and Z axis for MetaMotionS sensors (mz1, mz2, mz3). In this way, the direction plotted is the same for all sensors: in the way they were mounted, axis X for ST devices coincided with axis Z of the other ones when compared to the car movement.

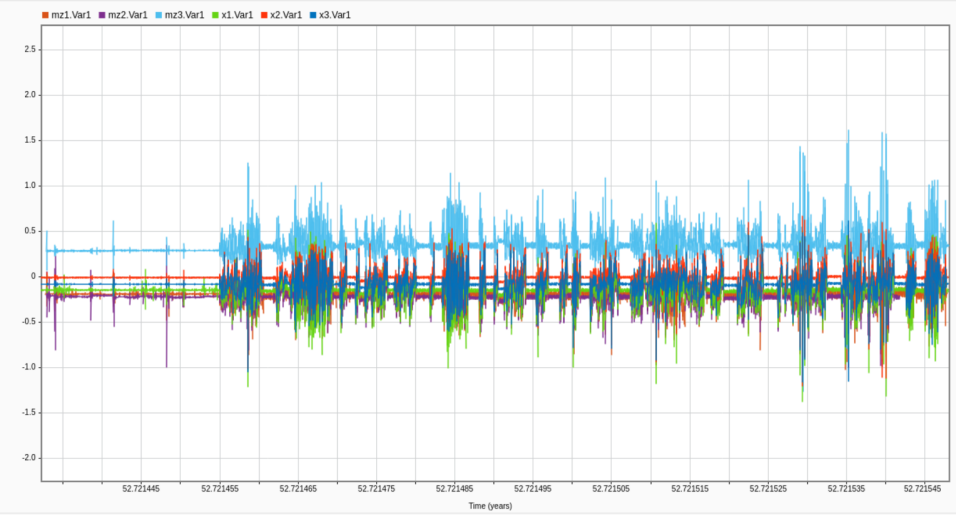



Figure 4.7: Graph representing acceleration on the X axis for all the sensor boards.

Figure 4.8 represents data for a particular acceleration event that happened in the last part of sampling. From this, I noticed that a MetaMotionS (mz3, light blue) is providing data differently from the other ones: it is the one mounted on the rear seat belt. First, it is 180° rotated (the front of this sensor board is facing the windscreen, while other ones are facing the rear window): hence, it is reporting data upside down. Then, it isn't perfectly perpendicular to the pavement, and a component of the gravity acceleration is generating an offset that is more evident than the one sensed by the other boards.

Figure 4.9 shows acceleration samples in a smaller time interval. In particular, I wanted to represent the accuracy of time synchronization: the samples are taken from one of the final acceleration events of the experiment, but the peaks are still well aligned. Also in this case, the third MetaMotionS is behaving in an interesting way as its samples seems delayed. This seemed strange at first, because I tested the performances of MBIENTLAB time synchronization mechanism and it worked perfectly every time. I think that the motivation of the difference is due to the installation on the seat belt, that is not as stiff as the other points on which the sensors were put.

Figure 4.10 is reporting the acceleration data from one of the sensor boards (in


Figure 4.8: Detail of acceleration in the minute domain.

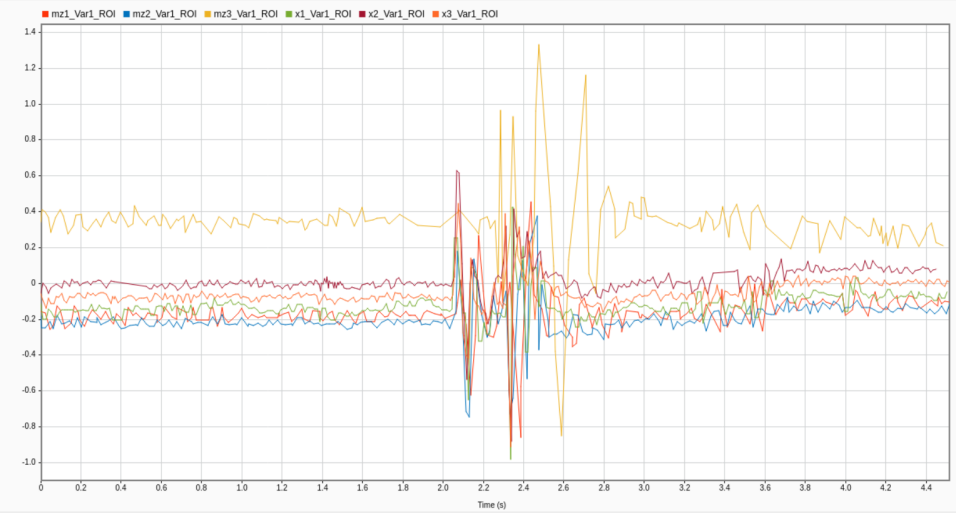



Figure 4.9: Detail of acceleration in the seconds domain.

green) compared to the dominant value of the audio frequency for each instant (in light blue, dashed lines). To make the values comparable inside the same graph, the frequency values have been scaled: $f_{GRAPH} = \frac{f_{ORIGINAL}}{1000}$. During the experiment, we played some frequencies more times with respect to what is shown here, so this was not the result I was expecting.

Figure 4.11 reports two particular audio events, with an acceleration event between them. In particular, after the right turn tone (2.8 kHz), acceleration is present, and after the left turn tone (3.2 kHz), acceleration is 0 for some moments (probably because we were giving the right to pass at the intersection). Hence, the

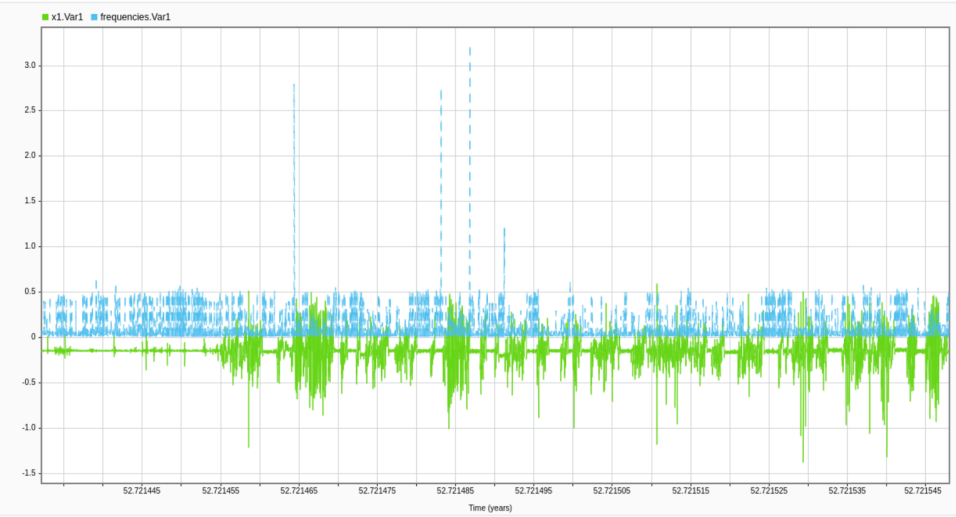



Figure 4.10: Graph of the acceleration data accompanied by audio frequency data.

frequency analysis is working correctly, but some issues should be resolved. I'm discussing this later.

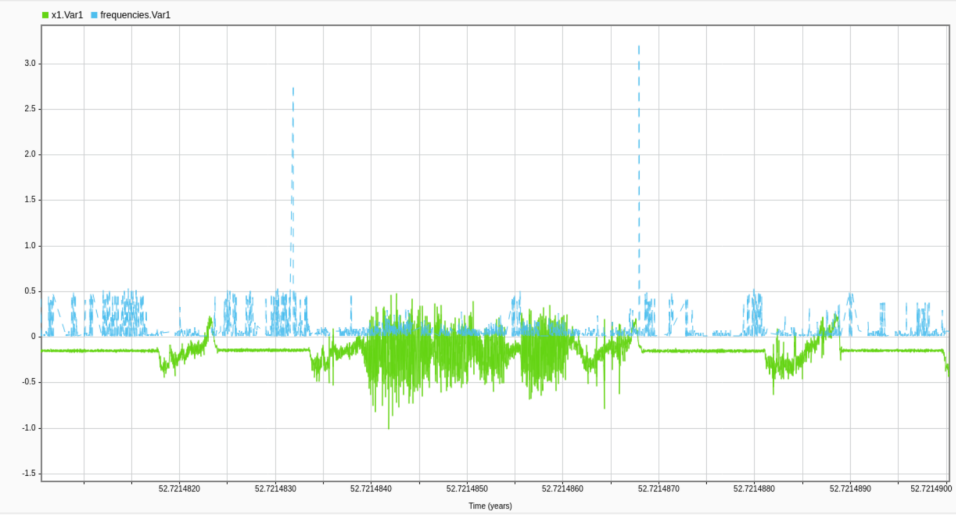



Figure 4.11: Graph of the acceleration data accompanied by audio frequency data, detail between two audio events.

Figures 4.12 and 4.13 represent the gyroscope data, both an overall view and a more detailed one. Also in this case, I noticed a different behaviour of the third MetaMotionS board, again because it's mounted on a seat belt and motion is transmitted differently. It is interesting to notice that these sensors are able to show this difference so precisely.


Figure 4.12: Graph representing gyroscope data on the X axis for all the sensor boards.


Figure 4.13: Detail of gyroscope data.

Figures 4.14 and 4.15 have been reported to show data sensed by the smartphone application compared to the sensor ones. To synchronize them over time, the clock of the smartphone (updated via Internet) have been used: it is interesting how the data coming from our sensors system is well aligned (even with a more rough precision).

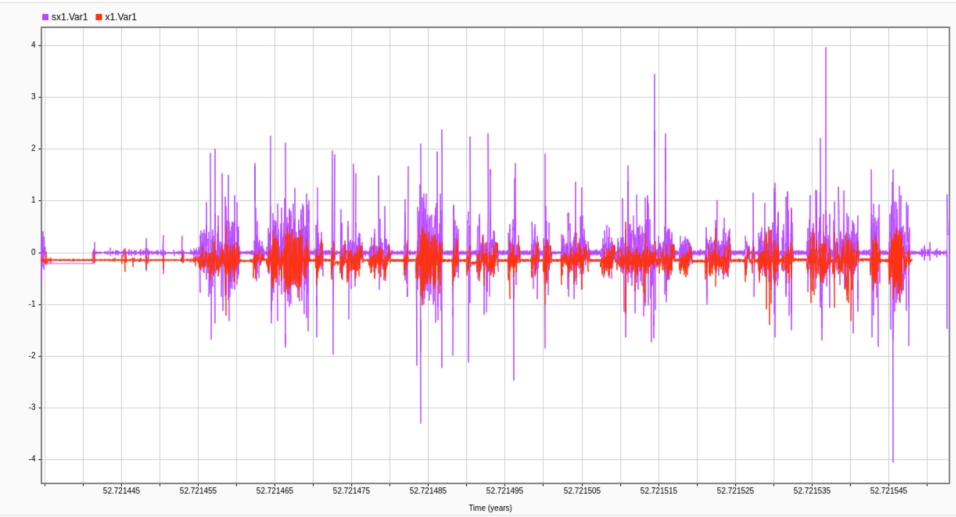



Figure 4.14: Graph of acceleration data from a sensor board compared to smartphone data.

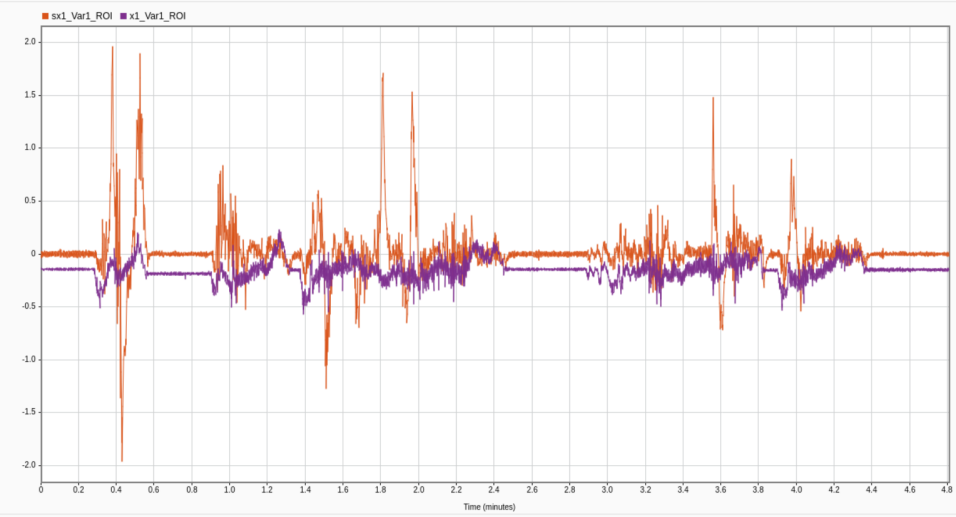



Figure 4.15: Detail of sensor board data compared to smartphone data.

The raw data reported until now can be corrected using the calibration algorithm explained in section 3.6. In the following images I'm reporting the results of the execution of the MATLAB script on the results coming from the sensor boards.

First, I run the calibration on the MetaMotionS (MMS) board that has been installed on the rear seat belt, as it is the one that requires more corrections.

Figure 4.16 compares the overall data between raw Z (pointing in the direction of the car) and the calibrated Y. It is evident from this image that all the samples have been moved from one axis to the other. Another detail is that the average

(i.e. the acceleration of the vehicle while static) is now around 0: the offset of the raw data is removed.

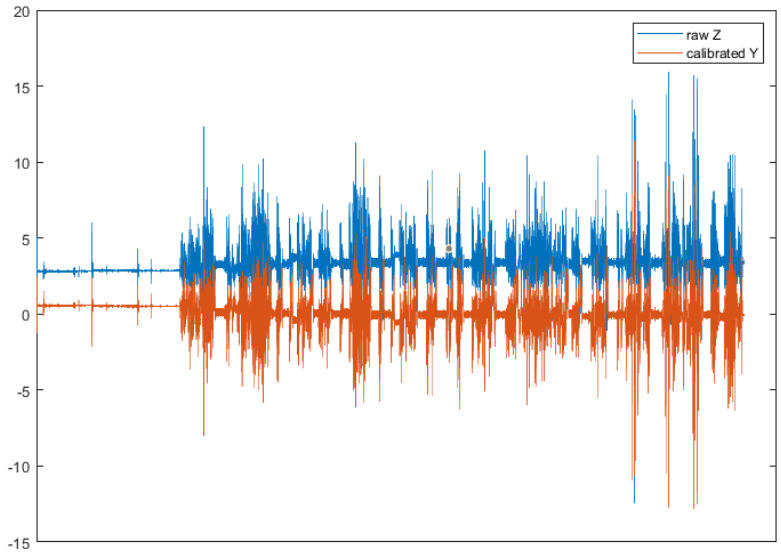



Figure 4.16: Comparison between raw data for Z axis and calibrated data for Y axis (MMS).

Figure 4.17 is a more detailed representation of the previous one. All the things said before can still be noticed, but it is also evident that the samples have been transferred symmetrically around the horizontal axis of the graph: this is because the axis of the sensor that was pointing to the car movement was 180° rotated.

Figure 4.18 compares the raw and calibrated data for the X axis. The X axis was the one pointing to ground, so raw data contained the gravity acceleration. Calibrated data is representing, instead, the real acceleration on the X axis, so there is no gravity component.

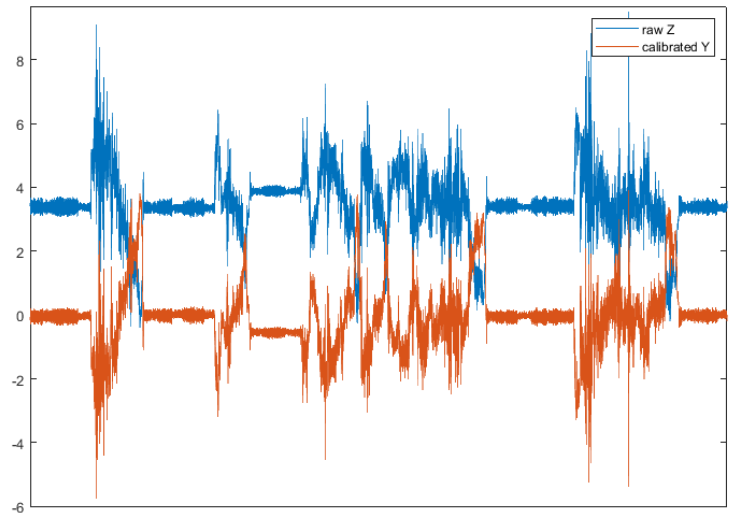



Figure 4.17: Detail of the comparison between raw data for Z axis and calibrated data for Y axis (MMS).

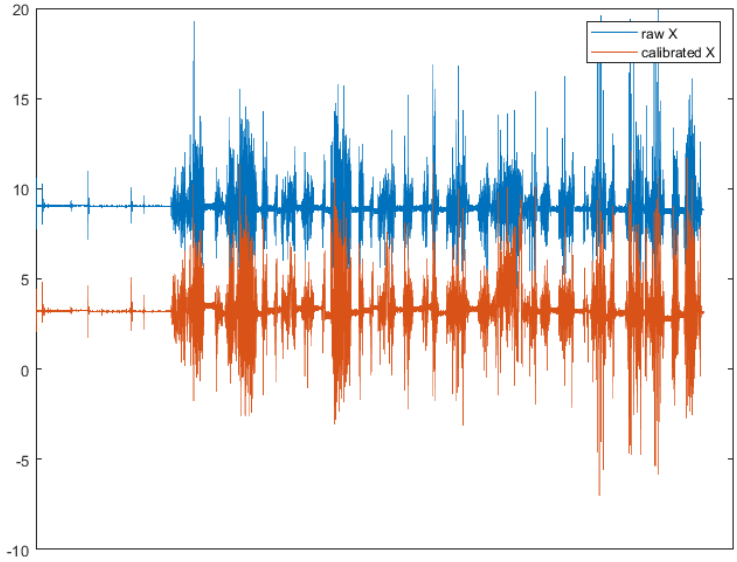



Figure 4.18: Comparison between raw data for X axis and calibrated data for the same axis (MMS).

Figure 4.19 compares raw and calibrated data for Y axis. Also in this case, due to rotation, the data represented in the calibrated version is composed of different components from the axis, so original and elaborated samples are not dependent.

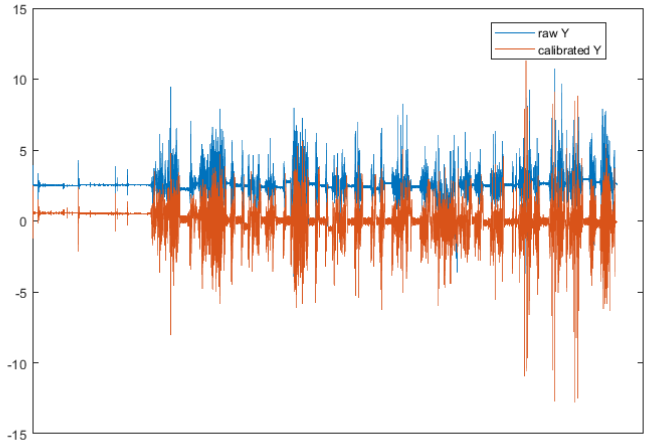



Figure 4.19: Comparison between raw data for Y axis and calibrated data for the same axis (MMS).

Figure 4.20 compares raw and calibrated data for Z axis. The calibrated samples average to 9.81, that is the gravity acceleration: it has been moved from the X axis to this one.

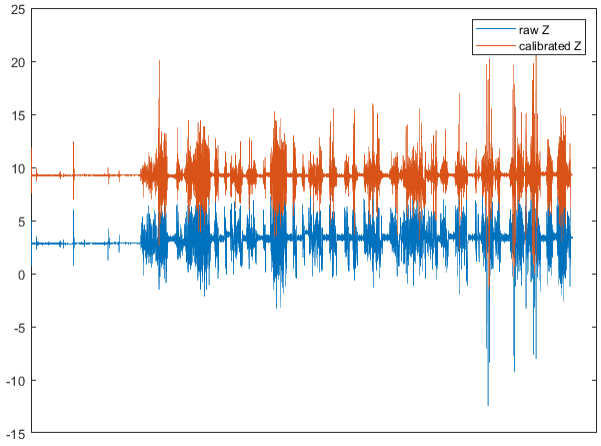



Figure 4.20: Comparison between raw data for Z axis and calibrated data for the same axis (MMS).

Then, I performed the calibration on the results of one of the ST sensors. They were correctly positioned with regards to the Z axis, but the X and Y axis would require some rotation.

Figure 4.21 reports calibrated data for the Z axis compared to the raw one. Values are the same, but calibrated one is mirrored with respect to the horizontal axis.

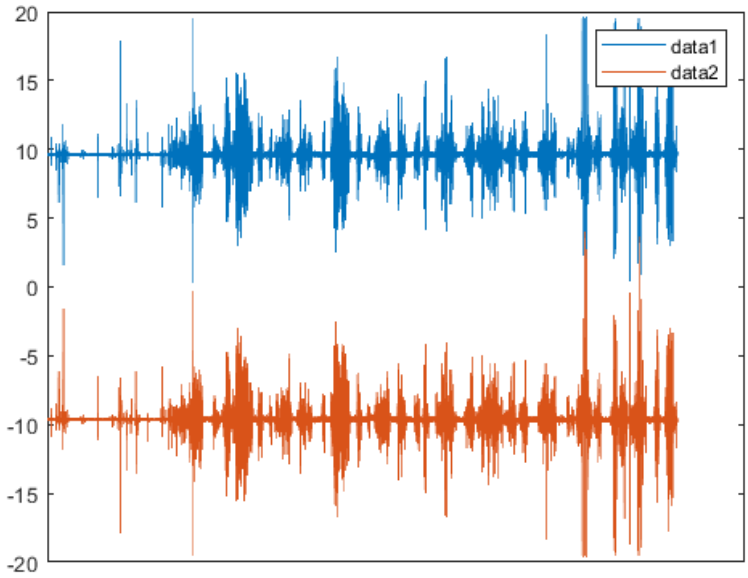



Figure 4.21: Comparison between raw data for Z axis and calibrated data for the same axis (ST).

Figure 4.22 compares raw X data and calibrated Y data. They represent the same values, as originally the sensor had its X axis coherent with the car movement, but the calibration algorithm moved those acceleration components on the Y axis.

Figure 4.23 is a detail of the previous one and it shows that also in this case data is mirrored: in this way, the acceleration of the car is positive after staying steady. Raw data (in blue) represents negative acceleration in all cases, but it doesn't make sense, as the car didn't go reverse.

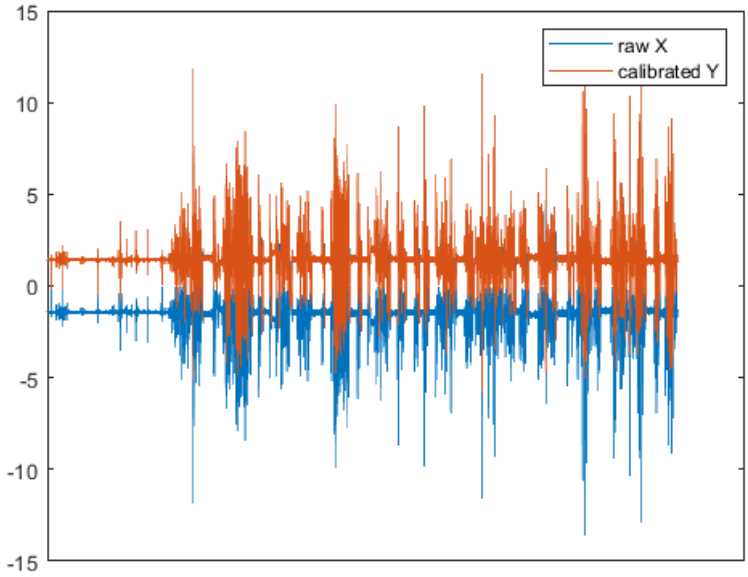



Figure 4.22: Comparison between raw data for X axis and calibrated data for the Y axis (ST).

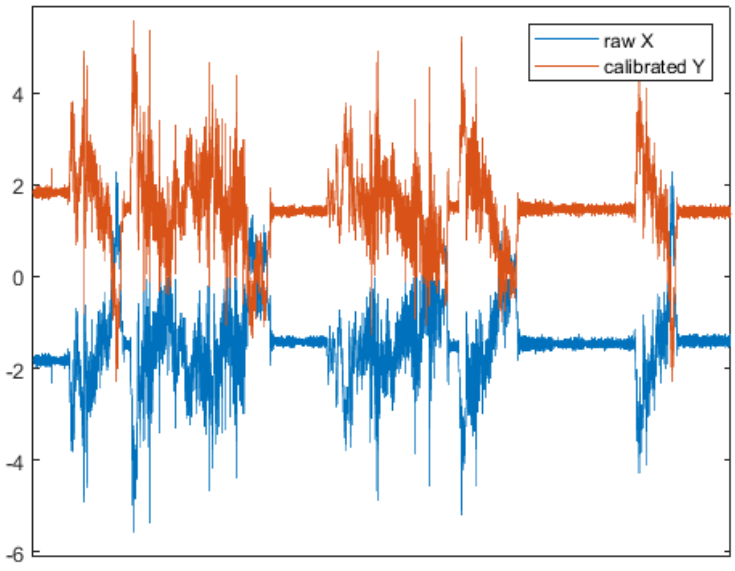



Figure 4.23: Detail of the comparison between raw data for X axis and calibrated data for the Y axis (ST).

Figures 4.24 and 4.25 both show comparison between raw Y and calibrated X. Also in this case, data from Y axis is moved on the X one, as the algorithm is designed in this way, but it's also mirrored: this happens because, when rotating the sensor axis 90 degrees (to bring X acceleration on Y axis), also the other component is rotated, causing it to be the negative of the original one.

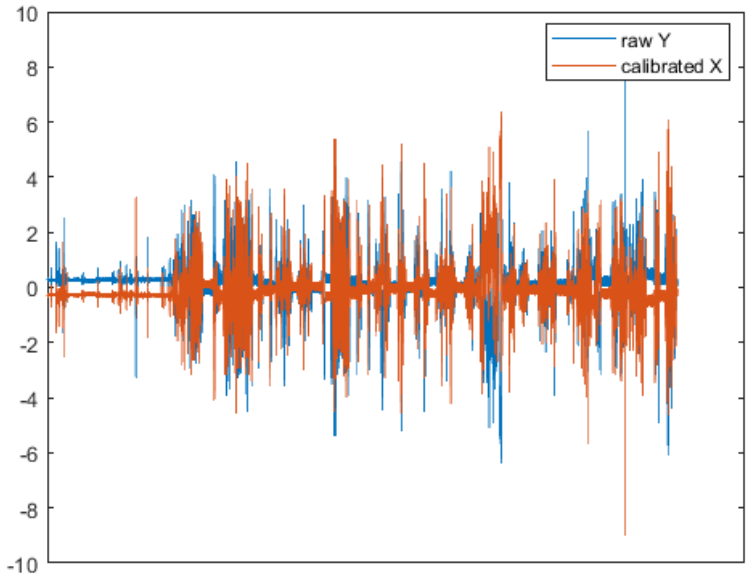



Figure 4.24: Comparison between raw data for Y axis and calibrated data for the X axis (ST).

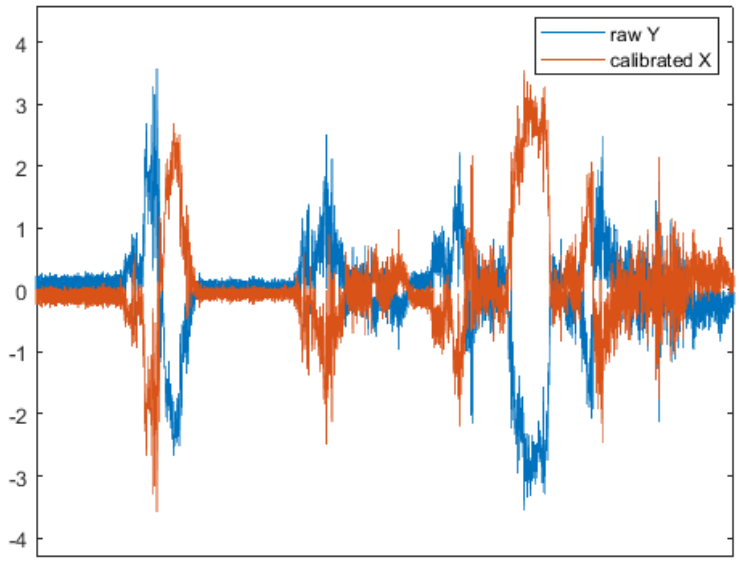



Figure 4.25: Detail of comparison between raw data for Y axis and calibrated data for the X axis (ST).

To show the potential of the acquisition system, I prepared some videos combining the data acquired from IMUs and the images acquired by the video camera. In this case, the synchronization between the two was not immediate, as the video shooting is not considered in the time synchronization mechanism: it required some manual adjustment. In any case, the video had the current time (obtained from GPS data) superimposed, and thanks to the MATLAB timetables structures generated from the elaboration scripts, finding the corresponding sample (with 1 second accuracy) has been quite easy. Then, thanks to the MATLAB *VideoWriter* functionality, I created an animation that can be easily integrated to the shot images thanks to a video editing program, like DaVinci Resolve 18 by BlackMagic [19]. A snapshot of the result is reported in Figure 4.26, and I'm reporting some conclusions on the experiment in the following paragraphs.

**Overall behaviour of the system** The system demonstrated strong reliability during the experiment, as no disconnection happened and the stream of data worked continuously all the time (around 1 hour). The fetched data is coherent with all the different devices inside my system, and comparing them with the samples of an external device (the smartphone) gave good results. Time synchronization worked as expected for all the duration of the experiment, keeping the same performance as in section 4.2.


Figure 4.26: Snapshot from the final video with both images and IMU data.

**Board positioning** It is important to put the sensor boards accurately in the car, choosing stiff points in which vibrations are limited. For example, fixing the boards on the dashboard with adhesive tape is a good idea, as data coming from the ST devices is clean and coherent. Also the clip accessories for MetaMotionS are a good idea, but they shouldn't be put on the seat belt or other moving parts, as the motion data is compromised (Figure 4.9). During the sensors installation, it is also important to remember the axis direction, but anyway an appropriate calibration algorithm is enough to fix it.

**Data elaboration** As explained for Figure 4.8 in particular, it's important to consider the differences between the sensors during data elaboration in Matlab. In this case, it could be convenient to remove the offset from all the devices data, to set the steady state around the value 0 (and not -0.3). It would be also a good idea to analyze the positioning of the sensors and align correctly the data (in this case, it simply means to put a minus to the mz3 values).

**Audio and Frequencies** The main idea behind the analysis of audio frequencies was to automatically label all the events using these values. Figure 4.11 shows that this can be possible, even if Figure 4.10 is proof that, in this experiment, this solution needs some further analysis. In particular, it seems that lower frequencies are not distinguishable from the background noises, and other frequencies are not shown probably because they were not the dominant ones. Possible solutions can be:

- Use higher frequencies, that seem to be recognized more easily

- Do some experiments in car with the same sensors, maybe with real-time results on the computer screen, to understand which frequencies are better to use.
- Put the speaker of the audio frequency generator nearer to the sensor microphone.
- Analyze more dominant frequencies than the first. In this case, only the frequency with the maximum power spectrum have been considered: maybe taking also the second and the third one could be beneficial in finding the reproduced ones.
- Make the car environment better for the noise propagation (for example, close all windows, put sensors on different parts of the car that don't vibrate and so on).

Another problem we noticed during the experiment is that programming the audio frequency to be played on the Android app is not easy. Every time, the old value should be deleted, the new one should be written, and so on. This procedure is slow and error-prone. An idea could be to pre-record audio data at certain frequencies, and use those audio files. In this way, the time duration of the sound is already known, the user is not confused between different frequency values and at the same time the correct one can be played without hassle.

# Chapter 5

# Conclusion

In this thesis I propose a complete system designed to acquire data from heterogeneous sensors, with time synchronization. I put particular attention in finding the best configuration for all of them, both on the data sensing and transmission parts. To complete it all, I also explored some possibilities to automatically label samples using audio information.

The developed firmware for the ST devices tries to get the most out of the available hardware, with the personalized time synchronization mechanism. I wrote Python clients for the computer to collect the data from all the sensor boards, using different Bluetooth adapters to maximize stability and throughput. As I had to deal with different devices, those scripts are personalized to generate coherent and synchronized output files. Finally, the MATLAB scripts I prepared can be used to analyze the resulting data and eventually perform some elaboration on them, using the tools available in the Signal Processing Toolbox.

Tests performed in different environments, including the real-life case of a car driven for about an hour on different paths, confirmed that all the parts of the system are working as expected: data is coherent between the different sensors and is correctly aligned.

In this last test I also used microphone data to label the data, and this idea gave interesting results that can be analyzed in further works.

Also the calibration algorithm, that automatically rotated the reference system of the sensors to the reference system of the vehicle, gave a good outcome even without GPS data.

I can consider the obtained results satisfying when compared to the expected ones. The time synchronization accuracy is around 70 ms, that is less than the desired 100ms. The relative mechanism is easy to implement on any BLE-capable microcontroller, as it only makes use of the standard Application Layer of the protocol. It is also low-power, as it has been designed to impact performances as less as possible (only two packets are exchanged).

Multiple devices are well-supported, and new ones can be easily integrated. If the developer has the ability to personalize the firmware of new devices, it is possible to add them inside the system with the already available client: the new board should be programmed using the ST devices firmware as an example. If the device is closed-source, a new client should be written for them using their APIs, but then it can work in a parallel way. The current configuration, used in the experiments reported in this thesis, is represented in Figure 5.1.

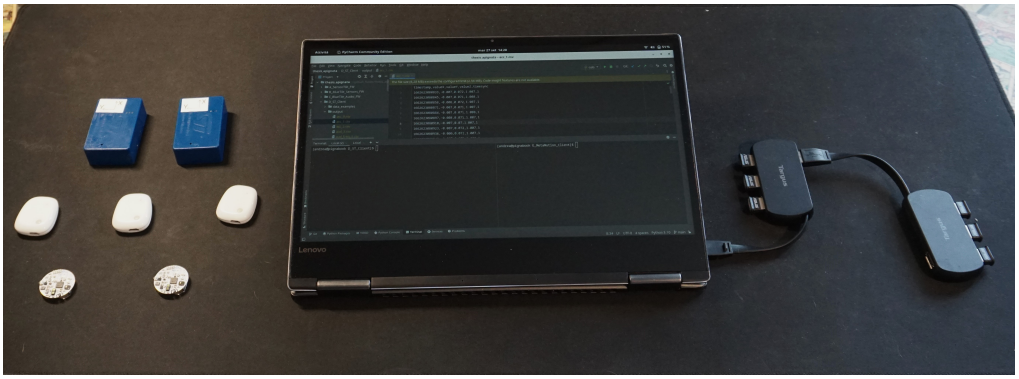


Figure 5.1: Image representing the components of the developed system.

A nice part of this work is that it can be considered a part of a bigger framework that can be further developed. For example, I'm reporting some possible ideas:

**Labeling Improvement** The last experiment tried to use audio data for labeling, but as discussed in 4.3, the results can be improved. If automatic labeling is a priority in the desired framework, it is easy to implement all the proposed fixes and then use the audio frequencies analysis to tag data.

**Labeling using voice** Another exciting possibility, that would require a more powerful device (but maybe can be implemented inside the client) is to do the automatic labeling using a voice recognition engine. In this way, the driver could simply express by voice the current condition, and the system would then label the IMU data in the required way.

**Edge Computing** The entire system relies on a central device, the client, that gathers all the output data and is responsible of the time synchronization. The main problem of this solution is that this device is a powerful but power-hungry one, as it is a personal computer, with a complete operating system on it. A possible way that can be explored is, depending on the target application, to leverage the computing power available on the boards (for example, SensorTile.box provide

a quite powerful microcontroller from the STM32 line) to perform data analysis directly on them and export on BLE only the desired features. ST Microelectronics already provides some Machine Learning libraries, optimized for their hardware.

To conclude the discussion, I imagined some possible applications for this system:

- **Predictive Maintenance** Everyday, people are using different high-cost machines. It could be the case of a car engine, when someone is reaching their workplace, but it could be also the case of big appliances in industries. In both cases, one of the most wanted things for both the end-user of the car and the managers of the industry is to reduce the costs for maintenance. The system developed in this thesis can help: data from IMUs, microphone and environmental sensors can be used to recognize any abnormality in the monitored system before it would cause any irreversible (and expensive) harm to other components.
- **Sports and Healthcare** Another interesting possibility is applying the set of sensor on different spots of the human body, in order to analyze how a person is moving and, eventually, study their motion during a sport or in everyday life. In this way, possible dangerous patterns can be recognized, helping the patient in avoiding useless harm. An example is the posture recognition as in section 2.7.

# Bibliography


[1] Carles Gomez, Ilker Demirkol, and Josep Paradells. «Modeling the maximum throughput of bluetooth low energy in an error-prone link». In: *IEEE Communications Letters* 15.11 (2011), pp. 1187–1189.

[2] Konstantin Mikhaylov, Nikolaos Plevritakis, and Jouni Tervonen. «Performance analysis and comparison of Bluetooth Low Energy with IEEE 802.15. 4 and SimpliciTI». In: *Journal of Sensor and Actuator Networks* 2.3 (2013), pp. 589–613.

[3] Carles Gomez, Joaquim Oller, and Josep Paradells. «Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology». In: *Sensors* 12.9 (2012), pp. 11734–11753.

[4] Davide Giovanelli, Bojan Milosevic, and Elisabetta Farella. «Bluetooth Low Energy for data streaming: Application-level analysis and recommendation». In: *2015 6th International Workshop on Advances in Sensors and Interfaces (IWASI)*. IEEE. 2015, pp. 216–221.

[5] Olaf Reich et al. «Bluetooth Performance Evaluation based on Notify for Real-time Body-Area Sensor Networks». In: *2020 IEEE International Workshop on Metrology for Industry 4.0 & IoT*. IEEE. 2020, pp. 516–520.

[6] National Instruments Corp. *Synchronization Explained*. 2021. URL: `https://www.ni.com/it-it/support/documentation/supplemental/10/synchronization-explained.html` (visited on Sept. 20, 2022).

[7] Shaoshan Liu et al. «Brief industry paper: The matter of time—A general and efficient system for precise sensor synchronization in robotic computing». In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2021, pp. 413–416.

[8] Djamel Djenouri and Miloud Bagaa. «Synchronization protocols and implementation issues in wireless sensor networks: A review». In: *IEEE Systems Journal* 10.2 (2014), pp. 617–627.

[9] Farzad Asgarian and Khalil Najafi. «BlueSync: Time Synchronization in Bluetooth Low Energy with Energy Efficient Calculations». In: *IEEE Internet of Things Journal* (2021).

[10] Anastasios Petropoulos, Dimitrios Sikeridis, and Theodore Antonakopoulos. «Wearable smart health advisors: An IMU-enabled posture monitor». In: *IEEE Consumer Electronics Magazine* 9.5 (2020), pp. 20–27.

[11] STMicroelectronics NV. *STM32 Software Development Tools*. URL: https://www.st.com/en/development-tools/stm32-software-development-tools.html (visited on July 10, 2022).

[12] STMicroelectronics NV. *STM32CubeProgrammer software*. URL: https://www.st.com/en/development-tools/stm32cubeprog.html (visited on July 10, 2022).

[13] STMicroelectronics NV. *RF-Flasher utility*. URL: https://www.st.com/en/embedded-software/stsw-bnrgflasher.html (visited on Aug. 21, 2022).

[14] MbientLab. *MetaBase App*. URL: https://mbientlab.com/tutorials/MetaBaseApp.html (visited on Sept. 10, 2022).

[15] MbientLab. *MetaWear APIs*. URL: https://mbientlab.com/tutorials/MetaWearAPI.html (visited on Sept. 10, 2022).

[16] STMicroelectronics NV. *Software development kit for BlueTile*. URL: https://www.st.com/content/st_com/en/products/embedded-software/evaluation-tool-software/stsw-bluetile-dk.html (visited on Aug. 21, 2022).

[17] STMicroelectronics NV. *STM32 ODE function pack for IoT node with BLE connectivity*. URL: https://www.st.com/content/st_com/en/products/embedded-software/mcu-mpu-embedded-software/stm32-embedded-software/stm32-ode-function-pack-sw/fp-sns-allmems1.html (visited on July 10, 2022).

[18] MbientLab. *MetaWear SDK for Python by MBIENTLAB*. URL: https://github.com/mbientlab/MetaWear-SDK-Python (visited on Sept. 10, 2022).

[19] BlackMagic Design. *DaVinci Resolve 18*. URL: https://www.blackmagicdesign.com/it/products/davinciresolve (visited on Oct. 5, 2022).