UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

DESARROLLO DE UNA APLICACIÓN ANDROID PARA EL CONTROL Y GESTIÓN DE UNA RED INALÁMBRICA DE SENSORES

DEVELOPMENT OF AN ANDROID APPLICATION FOR THE CONTROL AND MANAGEMENT OF A WIRELESS SENSOR NETWORK

Francisco García de la Corte

Julio 2016

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

DESARROLLO DE UNA APLICACIÓN ANDROID PARA EL CONTROL Y GESTIÓN DE UNA RED INALÁMBRICA DE SENSORES

DEVELOPMENT OF AN ANDROID APPLICATION FOR THE CONTROL AND MANAGEMENT OF A WIRELESS SENSOR NETWORK

Francisco García de la Corte

Julio 2016

TRABAJO FIN DE GRADO

TÍTULO:	Desarrollo de una aplicación Android para el control y gestión de una red inalámbrica de sensores
AUTOR:	Francisco García de la Corte
TUTORA:	Alba Rozas Cid
PONENTE:	Octavio Nieto-Taladriz García
DEPARTAMENTO:	Departamento de Ingeniería Electrónica

TRIBUNAL:

Presidente:	Rubén San Segundo Hernández
Vocal:	Alvaro Araujo Pinto
Secretario:	Pedro José Malagón Marzo
Suplente:	Fernando Fernández Martínez

FECHA DE LECTURA: _____

CALIFICACIÓN: _____

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

DESARROLLO DE UNA APLICACIÓN ANDROID PARA EL CONTROL Y GESTIÓN DE UNA RED INALÁMBRICA DE SENSORES

DEVELOPMENT OF AN ANDROID APPLICATION FOR THE CONTROL AND MANAGEMENT OF A WIRELESS SENSOR NETWORK

Francisco García de la Corte

Julio 2016

Resumen

Dada la complejidad que presentan las redes inalámbricas de sensores a la hora de acceder a sus datos de una forma cómoda para el usuario, se ha desarrollado en este proyecto una interfaz de usuario que da una solución a este problema haciendo uso de un *smartphone*. Trata de una aplicación en Android que recibe el nombre de PGNAgent, que dispone de una interfaz de usuario sencilla para conocer el estado de la red y enviarle comandos.

Las redes inalámbricas de sensores suelen usar frecuencias no accesibles para *smartphones,* por lo que ambos elementos no pueden comunicarse directamente. Para conseguirlo, en un proyecto anterior se desarrolló una pasarela de comunicaciones que recibe los datos de una red mediante una interfaz inalámbrica en la banda de 433 MHz y se los traslada un *smartphone* a través de una interfaz USB.

En este proyecto, además de la aplicación Android, se ha diseñado una pasarela alternativa a la anterior, llamada Bluetooth Low Energy Portable Gateway Node (BLE PGN), en la que se han sustituido los componentes USB por los necesarios para permitir una comunicación BLE con un *smartphone*. Además, se ha reducido su tamaño y se le ha dotado de un sistema de alimentación que consigue que funcione de forma autónoma. La aplicación Android, por tanto, soporta tanto la comunicación USB con la pasarela original como la comunicación Bluetooth con el nuevo BLE PGN.

Se tratan en este documento las decisiones más importantes, en diversos capítulos, sobre el diseño y la implementación de la aplicación Android y la pasarela de comunicaciones. Ambos elementos implementan un protocolo de mensajes propio de una red inalámbrica de sensores de aplicación real en el ámbito de la lucha contra los incendios forestales.

Finalmente, después de haber tratado las fases de diseño e implementación, se muestran al lector un par de casos de prueba del sistema en conjunto que demuestran el correcto funcionamiento del mismo, seguido de un capítulo de conclusiones y otro de líneas futuras como final del documento.

Palabras clave: Android, aplicación, *smartphone*, pasarela, Bluetooth, BLE, USB, modular, interfaz de usuario, redes inalámbricas de sensores.

Abstract

Given the complexity wireless sensor networks present when it comes to accessing their data in a comfortable way, in this project we have developed a user interface that solves this problem by using a smartphone. It consists of an Android application called PGNAgent, which hosts a simple user interface that shows the network's state and can send commands to it.

Wireless sensor networks usually operate on frequencies that are not accessible for smartphones, and therefore both elements cannot directly communicate. In order to accomplish it, a communications gateway was developed in a previous project. This gateway receives data from a network though a wireless interface on the 433 MHz band and forwards it to a smartphone through a USB interface.

In this project, besides the Android application, a gateway was designed as an alternative to the previous one, called Bluetooth Low Energy Portable Gateway Node (BLE PGN), in which the USB components have been replaced with the necessary ones to allow a BLE communication with a smartphone. Besides, its size has been reduced and it has been given a power system that achieves an autonomous working mode. The Android application, therefore, supports USB communication with the previous gateway as well as BLE communication with the new BLE PGN.

In this document the most important decisions are dealt with, in diverse chapters, about the design and implementation of the Android application and the communications gateway. Both elements implement a message protocol featured in a real-application wireless sensor network in the scope of the fight against forest fires.

Finally, after dealing with the design and implementation phases, a couple of test cases of the global system are shown to the reader, proving the correct performance of it, followed by a conclusions chapter and another one on future lines as an end to the document.

Keywords: Android, application, smartphone, gateway, Bluetooth, BLE, USB, modular, user interface, wireless sensor networks.

Index

1. Introduction

1.1 Context

Wireless Sensor Networks, or WSNs, provide communication between nodes that measure different parameters of their surrounding area. WSNs have grown a great deal over the past few years, and their usefulness has been proved over and over [1].

Recent works of B105 Electronic Systems Lab suggest how WSNs can prevent road traffic accidents by detecting a breach in a work perimeter where workers can be run over by a car [2]. Another scenario is the prevention of forest fires by placing nodes with the capability of getting temperature or humidity samples and very low power consumption. This scenario was implemented and tested in a project called Prometeo [3], and this project is based on its WSN.

Nevertheless, there are many concerns about WSN implementation, such as battery draining, computing power or scalability. One of the biggest problems with WSNs is the poor user experience when it comes to supervising their state and data in an easy way. WSNs do not have a user interface due to their lack of resources, with the purpose of being low cost systems. Complex systems and big computers would be necessary to extract data from the sensors that form the Prometeo WSN, as the sensor nodes communicate using an RF frequency band.

In this project, we focused on providing a proper user experience by developing an Android application for smartphones. We relied on a previous project that successfully established this communication with an intermediary device called Portable Gateway Node (PGN).

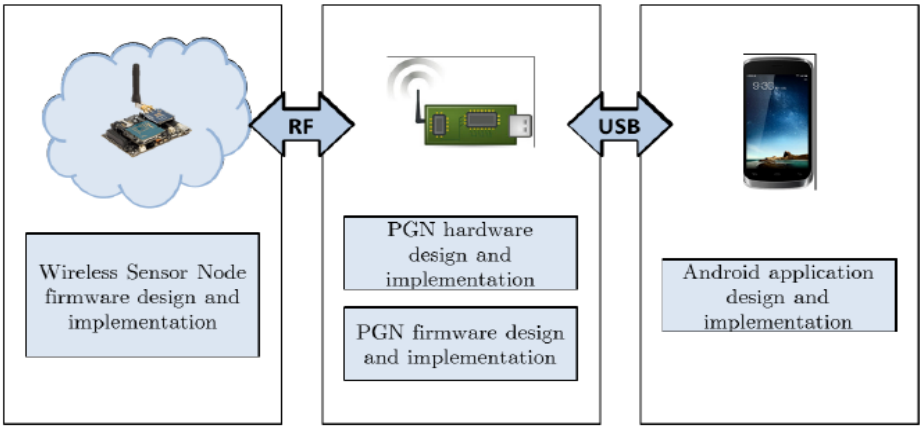

Figure 1: Previous project overview

As shown in Figure 1, the PGN device lets the Android application communicate with a WSN. This middle step between the WSN and the smartphone overcomes the incompatibility of their wireless interfaces, given the fact that WSNs usually operate on RF bands that are inaccessible to smartphones.

In addition to the Android application, an alternative PGN that uses Bluetooth instead of USB has been designed. The implementation of a Bluetooth link eliminates all the wires in the system and simplifies its design at the same time. The Android application, called PGNAgent, supports both ways of communicating with the PGN device.

1.2 Project objectives

The main focus of this project is the design of a new Android application, completely from scratch, that presents a modular architecture, based on a simple user interface and a smart management of the connection with the PGN over both USB and Bluetooth. The functionality division into distinct modules may help the adaptation processes to other kinds of WSNs.

Furthermore, as a second objective, we also redesigned the previous PGN device by replacing all USB modules with the necessary ones that will enable Bluetooth Low Energy (BLE) communications, so that the PGN device will become smaller and more comfortable to use. An important objective in this phase was the design of a Printed Circuit Board (PCB) as small as possible, without the need of any wires (except for battery charging).

The following steps fulfilled the described goals, generally studying associated technologies and designing modules afterwards. They are divided into two main phases:

Phase 1: Development of the application and establishing USB communication with a PGN emulator.

- Study of Android Studio and become familiar with the Android platform. This involves learning how to use the USB libraries offered by Android, defining a general structure to the application and setting everything up to test a USB connection.
- Study of IAR Embedded Workbench, necessary to design a firmware in the C programming language for the PGN microcontroller, which can read and answer messages sent by the Android application, so that it emulates a PGN device with USB support.

- Test the USB interface between the Android application and the PGN emulator. Once successful, implement all possible functionalities defined by the Prometeo protocol.

  Phase 2: Adding BLE support to the application and designing an alternative PGN device.

- Study of the BLE protocol and Android BLE libraries. Design and implementation of a BLE module in the Android application, as well as adapting the previous firmware for the microcontroller to use a Bluetooth module for its communications and testing.
- Study of the PCB design software Altium Designer. Once familiar with this program, the goal is to design the necessary schematics to eliminate the previous USB modules, add a Bluetooth module and a power system based on a battery that can be charged by an USB port that no longer transports information.
- Design of a PCB and solder all the components.

Finally, when all these steps concluded, a few final tests were carried out, and determined the global performance of all modules when integrated as one. The Android application implements all the commands defined in the Prometeo protocol (see Appendix I: Prometeo Communication Protocol, for more details on it).

1.3 Document structure

The present document will go through every decision made and designed module, justifying the reasons and explaining how everything works. Thus far we are ending Chapter 1, which covered the introduction of this project.

Chapter 2 will generally describe the developed system, going over the two highest level modules: the Android application and the alternative PGN device with Bluetooth connectivity.

Furthermore, Chapter 3 will go into detail about the designed Android application, showing diagrams of its architecture and classifying its Java classes according to their role, as well as the most important used libraries. Similarly, Chapter 4 will go over the second phase of this project: the PCB design. It will present the selected modules, their schematics, reasoning their involvement and how they interact with each other to form a Bluetooth communicating device.

In addition, Chapter 5 will present two test scenarios that prove the correct performance of the whole project. Chapter 6 will explain the conclusions that can be brought up from the development of the project.

Finally, Chapter 7 will collect a few improvement suggestions for a possible new iteration of the idea this project is based on, and Chapter 8 will present the bibliography. The document will end with a list of acronyms and an appendix that explains how the Prometeo Communication Protocol (PCP) works.

2. System overview

In this chapter, a general view of the developed system is going to be covered. The designed system starts with an Android application, called PGNAgent, which can send and receive messages over USB or Bluetooth, as well as providing a user interface to manage a Prometeo WSN. The other part of the system consists of a PCB with the necessary modules to establish a Bluetooth communication with PGNAgent, as a modification to the previous PGN device. Though it does not have a great impact in this project, both PGN devices communicate with a WSN on an RF band with their equipped antennae. Figure 2 illustrates the system architecture in its highest level.

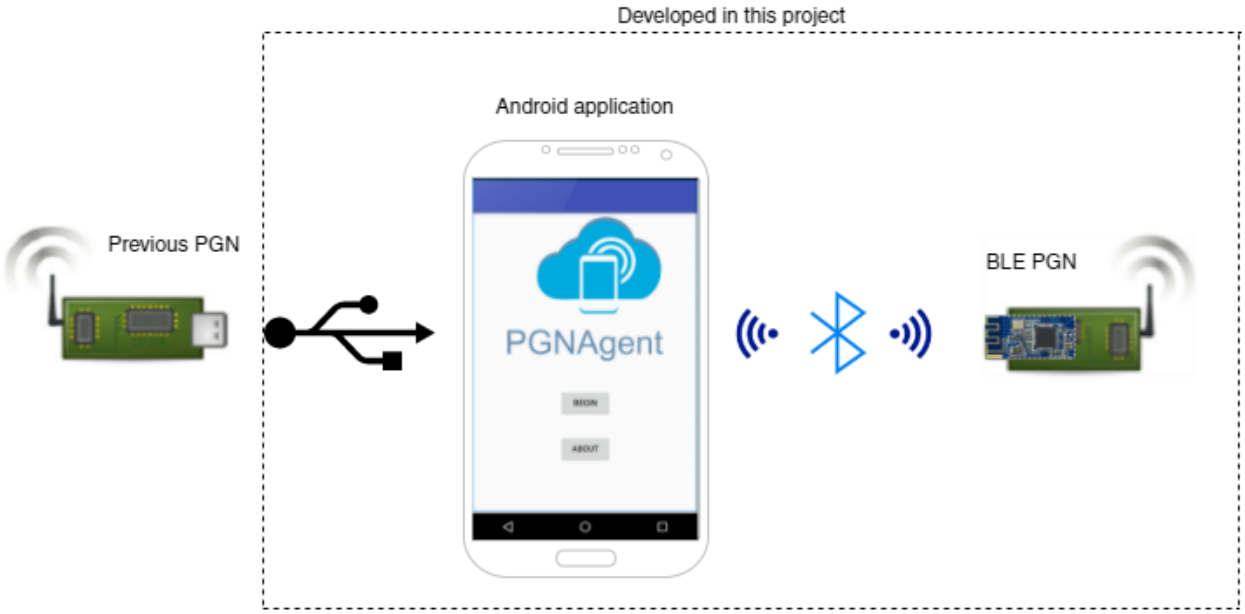

Figure 2: System overview

2.1 PGNAgent

The designed Android application is the biggest and main part of this project. The objective was to give the user a proper experience when managing and controlling a WSN. This is accomplished by using something that everyone has and carries around with them: a smartphone. Most people are familiar with using a smartphone, so an application can give a simple and intuitive interface to a potential user.

The Android operative system was chosen due to a few reasons. The most important one is the fact that, as of the first quarter of 2016, Android holds 84.1% market share, with an astonishing number of over 293 million sales in that period [4].

Operating System	1Q16 Units	1Q16 Market Share (%)	1Q15 Units	1Q15 Market Share (%)
Android	293,771.2	84.1	264,941.9	78.8
iOS	51,629.5	14.8	60,177.2	17.9
Windows	2,399.7	0.7	8,270.8	2.5
Blackberry	659.9	0.2	1,325.4	0.4
Others	791.1	0.2	1,582.5	0.5
Total	**349,251.4**	**100.0**	**336,297.8**	**100.0**

Table 1: Worldwide Smartphone Sales by Operative System (Thousands of Units) [4]

The second most important reason for using Android is the extensive access it gives to the low-level features of devices. For instance, the USB interface can be easily accessed for a developer. This was crucial for this project's development, as it had to use the USB interface in a comprehensive way.

The use of the Java programming language also made the choice of Android easier, given its features when it comes to high-level programming. When low-level operations were needed, a library that implemented them was always found, and the graphic management of the app was also easily implemented.

Although Bluetooth Low Energy is supported since API level 18, minimum target API level 21 was chosen due to programmatic reasons that will be explained in Chapter 3 in detail. API level 21, or Android 5.0 – 5.0.2 Lollipop and further versions hold 45.5% penetration on Android devices as of June 2016 [5], so a big amount of these devices would be able to install PGNAgent. This number is growing so fast that it already may be bigger than 50%.

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.0%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.9%
4.1.x	Jelly Bean	16	6.8%
4.2.x		17	9.4%
4.3		18	2.7%
4.4	KitKat	19	31.6%
5.0	Lollipop	21	15.4%
5.1		22	20.0%
6.0	Marshmallow	23	10.1%

Table 2: Relative number of devices running a given version of the Android platform [5]

2.2 BLE PGN

After designing PGNAgent, an alternative PGN device was developed as a second phase to this project, with the purpose of establishing a BLE link with the smartphone. We based this design on the previous PGN device, which could already communicate with a WSN through its RF module.

When the time came to redesign the previous PGN, we eliminated USB wires by implementing a Personal Area Network (PAN) wireless technology. Bluetooth was an easy option, as Android provides prepared libraries to configure a Bluetooth link with relative ease and a user with a smartphone would just need to be within 50 meters from BLE PGN, according to the following table:

Technical Specification	**Classic Bluetooth**	**Bluetooth Low Energy**
Distance/Range	100 m (330 ft)	50 m (160 ft)
Over the Air Data Rate	1 – 3 Mbit/s	1 Mbit/s
Application Throughput	0.7 – 2.1 Mbit/s	0.27 Mbit/s
Active Slaves	7	Not defined; implementation dependent
Security	56 / 128-bit and application layer user defined	128-bit AES with Counter Mode CBC-MAC and application layer user defined
Robustness	Adaptive fast frequency hopping, FEC, fast ACK	Adaptive frequency hopping, Lazy Acknowledgement, 24-bit CRC, 32-bit Message Integrity Check
Latency (from a non-connected state)	Typically 100 ms	6 ms
Total time to send data (depending battery life)	100 ms	3 ms, less than 3 ms
Voice capable	Yes	No
Network topology	Scatternet	Star-bus
Power consumption	1 as the reference	0.01 to 0.5 (depending on the use case)
Peak current consumption	Less than 30 mA	Less than 20 mA
Service discovery	Yes	Yes
Profile concept	Yes	Yes

Table 3: Bluetooth Classic and BLE comparison [6]

Table 3 gives us a lot of interesting information on BLE specifications. Besides the acceptable range and its low-power features, we can see how its data rate (270 Kbit/s, application level) is very low compared to Bluetooth Classic's. However, it is enough for the Prometeo Communication Protocol (PCP) as it does not take more than a few bytes for each message. Its low power consumption (10% to 50% of Bluetooth Classic's) makes it worth the sacrifice, so it was decided to user BLE over Bluetooth Classic for our PGN

Once the idea of BLE for an alternative PGN device was chosen, and BLE was chosen as the interface with the Android application, a power system that allowed the BLE PGN to operate in an autonomous way had to be implemented since it is no longer powered from the USB interface. The implementation of the power system will be covered in Chapter 4, section 4.2.4.

We have hereby justified the reasons why certain technologies were chosen for the global system. The document will go on with chapters on the design and implementation of the Android application and the BLE PGN as the hardware element of this project.

3. Android application: PGNAgent

This chapter will cover the main part of this project: the designed and developed Android application that fulfills all established requirements. The same way other chapters will, this one will go through all decisions that were made in order to achieve those requirements, starting off by shedding light on them. After that, a general diagram of its logic modules will be presented, naming the logic functionality each block carries out. Finally, the implementation subchapter will deal with technical details of how the application works, the concrete functions that work within modules and how they interact with each other.

3.1 Design

3.1.1 Requirements

As explained in this project's offer, the Android application, called PGNAgent, has to accomplish the following requirements:

Functional

- Serve as an interface to supervise and manage the Prometeo WSN from a smartphone. This way, complex systems and big computers are avoided in order to communicate with the network, so that the portability of the user interface is greatly increased.
- Show a general vision of the WSN. All integral parts of the network (nodes) will be shown, along with their characteristics and values of measured magnitudes, such as humidity or temperature.
- Send and receive commands over the established communication with the PGN device. In order to accomplish this, the application has to implement the PCP.

Non-Functional

- Flexibility. The application needs the ability to adapt to any changes it may bear in the future, such as new ways of communicating with a PGN device, error correction or graphic changes.
- Intuitive usability. An average user, with knowledge of what the Prometeo WSN can do, should be able to use this application without any difficulties.

- Modular design. All functionalities have to be separated in different modules that can interact with each other. This feature will help any possible changes to the application, as well as being clear and clean about what is in charge of what.

Figure 3 presents a diagram of the global system from a user's point of view, which can be deduced from the requirements:

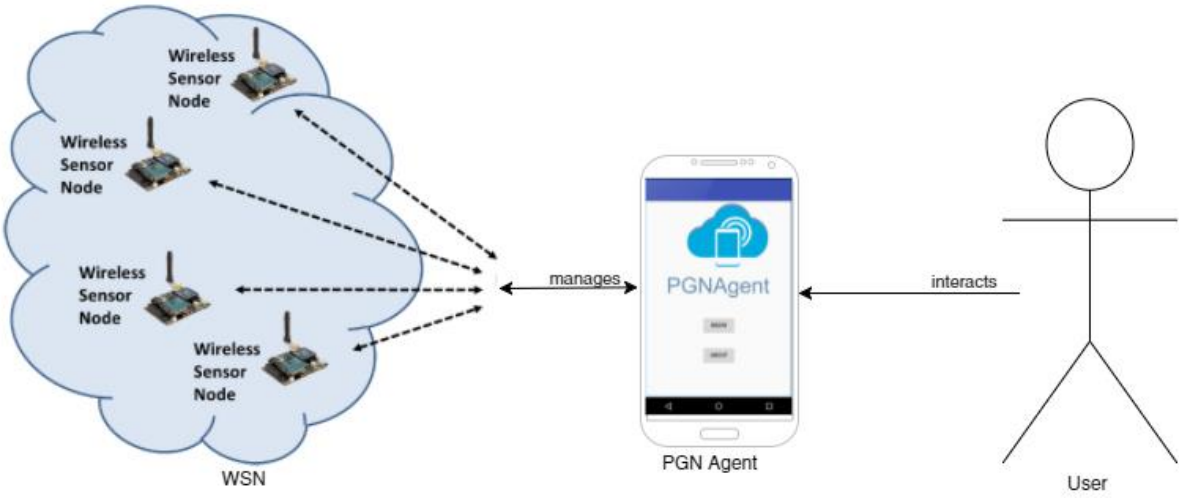

Figure 3: User's point of view of the system

3.1.2 Architecture

Once all the requirements for PGNAgent have been clarified, the designed architecture can be understood. The MVC (Model, View and Controller) design pattern was chosen, so functionality is classified by the role each module takes. Model contains all the modules that define data that has to be used along the application, such as the unification of a node attributes or the definition of a graphic model. View, in this case, collects all XML files that define the layout for the activities and the activities themselves, which are the elements that control graphic operations in Android. The last component, Controller, consists of a few independent modules or Java classes that implement specific functions for the app that are transparent to the user, the establishment of a Bluetooth Low Energy link being one of them.

Figure 4 shows the implemented architecture, with each part of the MVC design pattern wrapping each module:

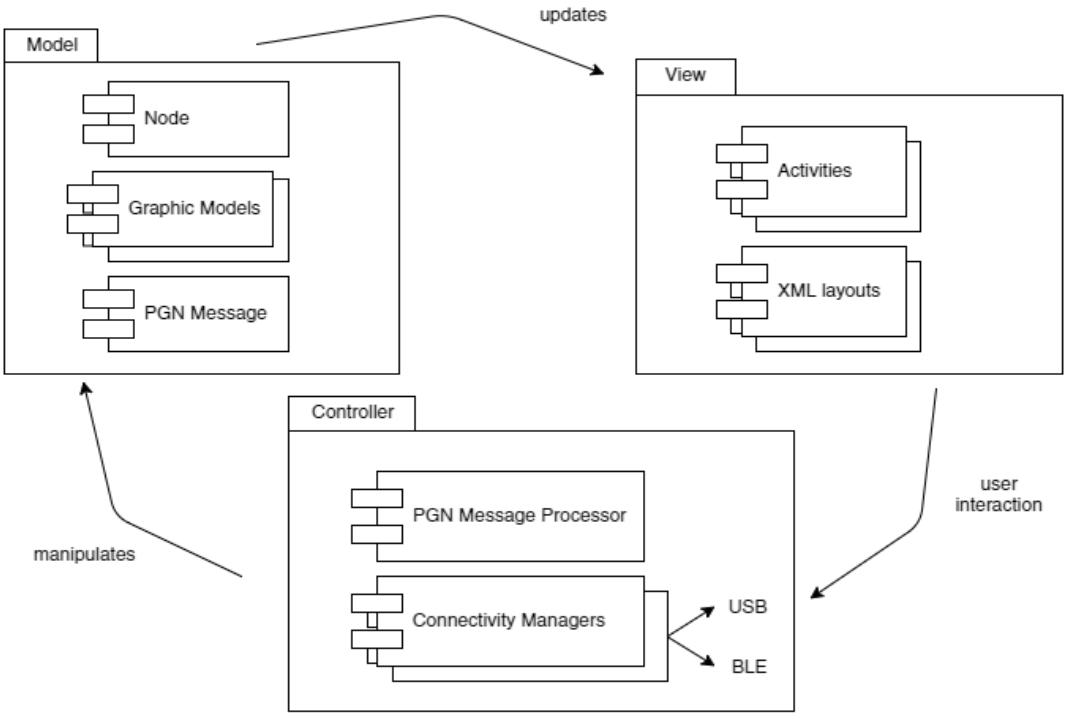

Figure 4: PGNAgent architecture diagram, from the MVC perspective

Each one of these three big blocks of PGNAgent's architecture had more particular design choices as well, as the MVC design pattern is not enough to start the implementation phase of the application.

In the case of the View block, it was decided to implement four activities: the first one shows a presentation screen; the second one handles a general view of the WSN; the third one details a specific node; and the last one shows general information about PGNAgent.

On the other hand, there was a need for modeling the data shown to the users by the activities. The modules that obtained a place in the Model block were a simple definition of a node from the WSN and its parameters, a standardization of a message that follows the PCP and graphic models for list items.

Finally, the main part of PGNAgent is the Controller block. It contains handlers for the chosen kind of link (USB or BLE) and a gateway between the lowest functions of connectivity and graphic operations and model data updates. This gateway deciphers all the messages received from the PGN device (previously being received by link handlers) and decides what module will dispatch them. It also takes whatever message the user wants to send in the opposite direction and gives it to the link handlers.

3.2 Implementation

In this subchapter, the development of PGNAgent will be explained in detail. With the architecture already covered, we will go through how each module works and the most important facts about their coding. The design subchapter described what modules were implemented, classified according to the MVC design pattern, but the implementation part will explain modules using a different classification. The first part, connectivity, will cover how the application uses USB and BLE, followed by a message handling part that will deal with the logic implemented in a module that processes messages and the definition of the PCP as a Model module. Finally the implemented activities that form the graphic elements of the application will be laid out.

3.2.1 Connectivity

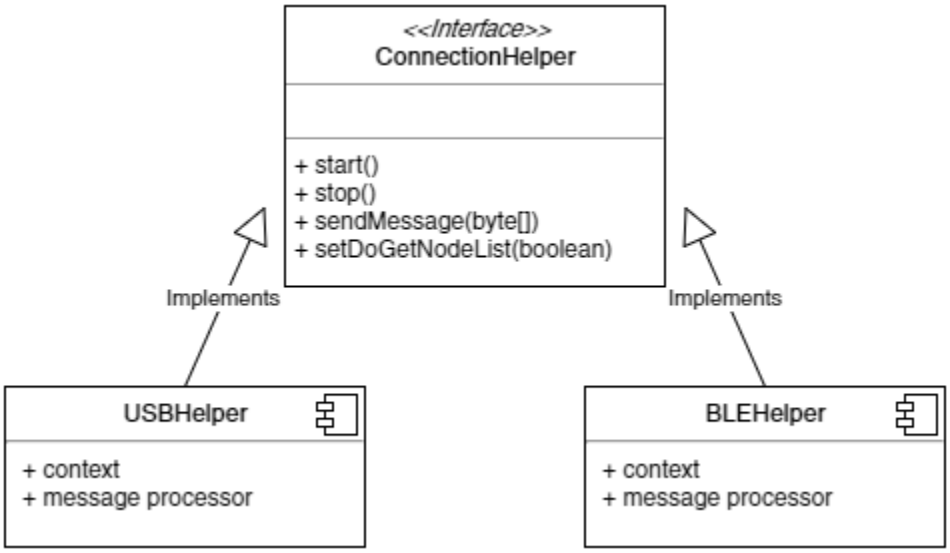

Figure 5: Connectivity implementation plan

Figure 5 shows the relation between the three Java classes that had to be implemented in order to provide connectivity to the application. As can be seen, *ConnectionHelper* is an interface that forces any class that implements it to have a few methods any other module may need to use. This was the best option in order to let the rest of the application avoid any issues about distinguishing what type of connection, USB or BLE, is being used (except for the part where the user has to choose it, which will be explained in section 3.2.3). *ConnectionHelper* consists of no more than a few lines of code that declare start, stop, and a message sending method. The last method lets other modules decide whether they want to send an automatic

"get_node_list" message once the connection is established, so that information about how many nodes there are and their identifiers will be immediately received. However, the fact that *USBHelper* and *BLEHelper* implement the same interface does not make them work the same way. Actually, their internal logic is mostly different, due to the use of different sources and libraries.

- *USBHelper*

USBHelper is a Java class that handles the USB link, if it is chosen by the user. Knowing that the application had to communicate with a MSP430 microcontroller, we made use of a document [7] from Texas Instruments that provides some guidelines when developing an Android application with that purpose. Then, based on these guidelines and adapting them to the *ConnectionHelper* interface, *USBHelper* implements the following diagram of events and actions:

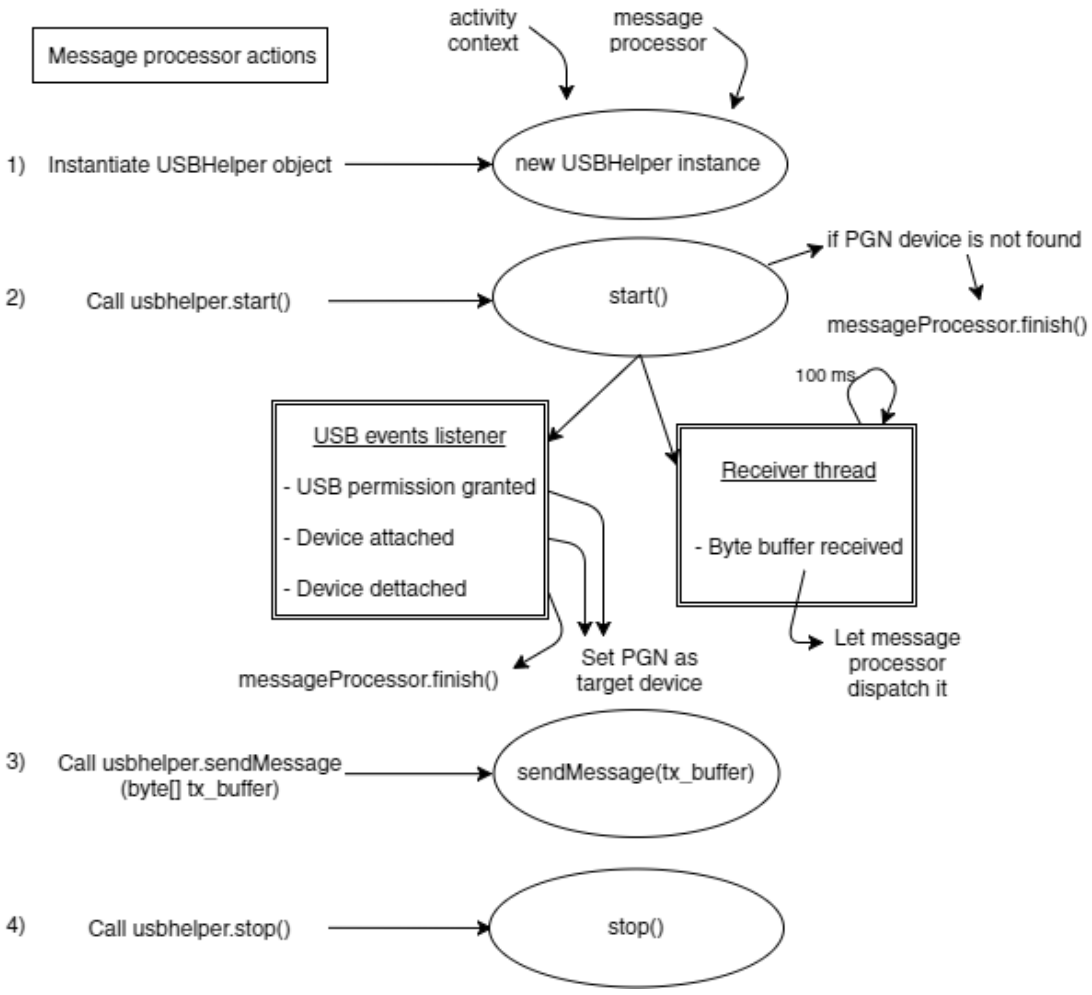

Figure 6: USBHelper logic diagram

If we take a look at Figure 6, we can see how the message processor (the link module between activities and connectivity modules that will be explained in section 3.2.2) can call any *ConnectionHelper* methods.

Firstly, a *USBHelper* object needs to be instantiated so that its methods become accessible. The constructor defined for the *USBHelper* class requires an activity context (so it can access the Android resources and therefore it can send alerts to View modules) and a message processor to be called when a message is received. This instance of *USBHelper* needs to be carefully treated in activities through the message processor.

After that, the message processor can call the *start* method, which sets many attributes needed for the following operations and stops if the PGN device is not found in the USB interface (and then alerts about it). Two different listeners are set: the first one is a USB events listener that manages USB permission granting and the attaching and detaching of devices. The second listener is actually a thread that checks if there is an incoming stream of bytes from the USB interface every 100 milliseconds.

In this state, when USB permission is granted or a device gets attached to the smartphone, it is checked if it is the PGN by getting its vendor and product identifiers. Also, if the PGN device is detached, an alert is sent to the graphic interface through the context attribute. When a message is received, the receiver thread catches it and passes it to the message processor.

Once the PGN has been correctly found and set, the message processor can send a message as a byte array using the *sendMessage* method. This method checks if *USBHelper* is ready to send a message, so that the application does not crash. If the PGN is not plugged in, an alert dialog warns about it.

Figure 7: Not connected PGN alert dialog

Whenever the application needs to close the USB link (when it goes into background, the user hits the back button, etc), the *stop* method forced to be implemented by

ConnectionHelper can safely stop the listeners and close the connection, but the *USBHelper* instance needs to be destroyed.

- *BLEHelper*

On the other hand, we have *BLEHelper.* Its functionality varies in the inside, but it is transparent to the rest of the application except for the moment that the user needs to choose whether he or she wants to use USB or BLE to establish a connection with the PGN. The main reason why this class implements a different logic than the one in *USBHelper* resides on the BLE protocol. This time, it is not as simple as plugging a USB device to the smartphone; we have to scan for discoverable devices and apply BLE's way of pairing with a peripheral (the smartphone acts as the central device) and manage the data transfer differently.

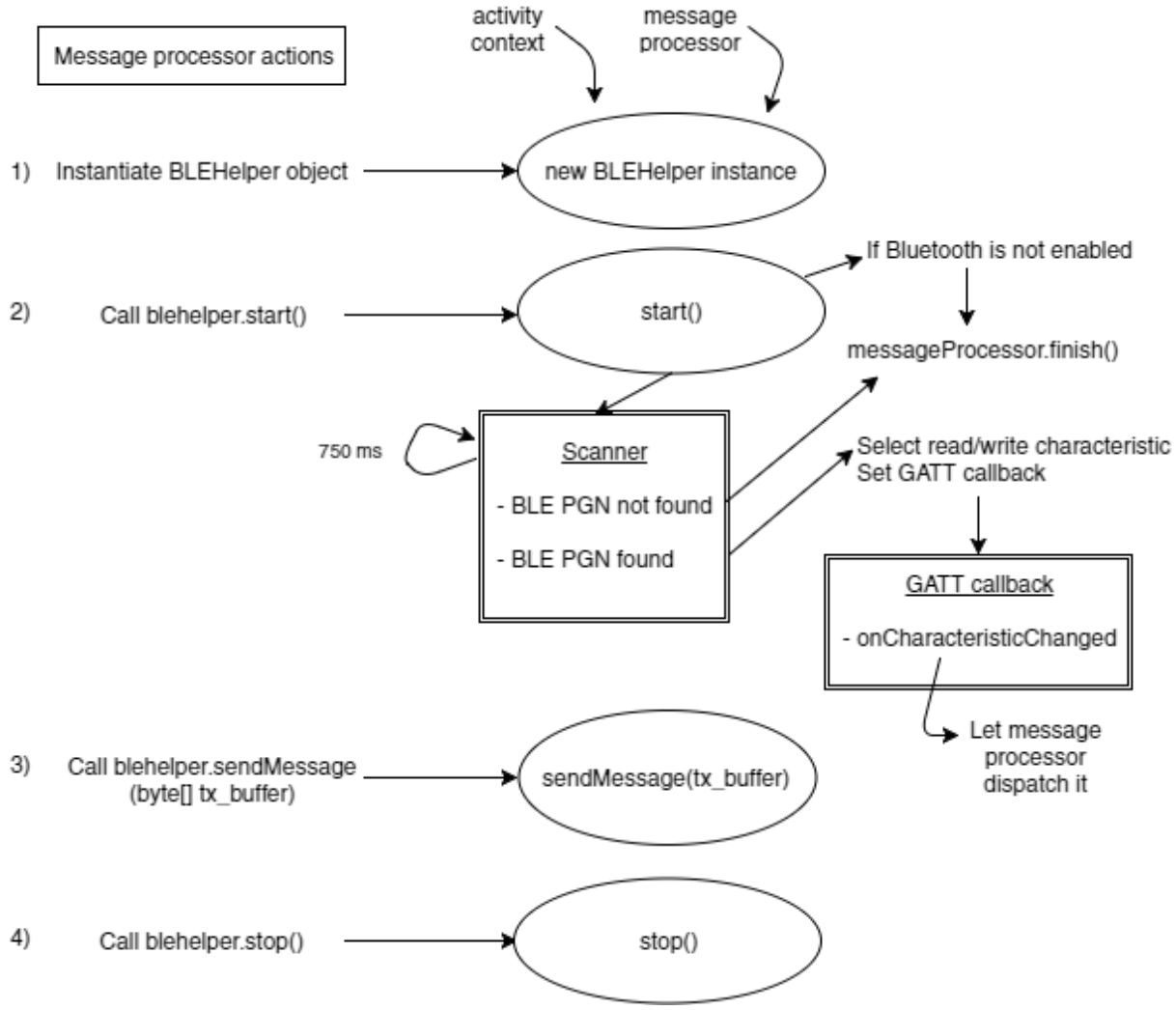

Figure 8: BLEHelper logic diagram

Figure 8 shows us how, in the eyes of the message processor, *BLEHelper* works the same way *USBHelper* does. Now, however, instead of just sending byte buffers to the USB interface, BLEHelper needs to scan for BLE devices that match the BLE PGN address.

With the understanding of [8], on how the HM-10 Bluetooth module (implemented in the BLE PGN) and [9], Android example that uses BLE, *BLEHelper* was created and implemented. Just like *USBHelper*, a class instance with an activity context and a message processor is required. Then, when the *start* method is called on the *BLEHelper* instance, after checking that Bluetooth is enabled in the smartphone, the smartphone is BLE compatible and the API level is 21 or higher, a BLE scanner is activated and has a 750 millisecond time interval to find the BLE PGN. This period of time was chosen when activities lifecycles and connection management were tested; it gives more than enough time to reestablish the connection if it is lost, but, at the same time, the user does not have to wait long. There is error checking in some scenarios; for example, an alert dialog shows when Bluetooth is not activated on the smartphone:

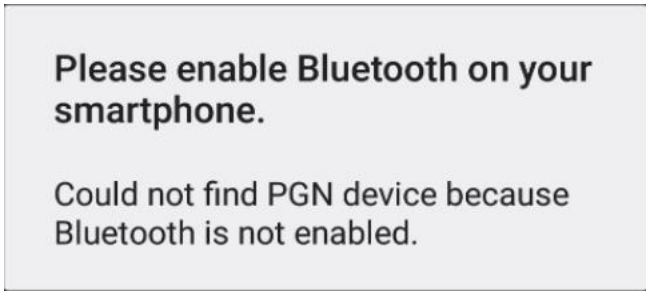

Figure 9: Bluetooth not enabled alert dialog

The BLE scanner has three callback methods for the cases in which there are some BLE devices found, only one or none. When the BLE PGN is found by any of these callback methods, the correct services and characteristics of the BLE PGN are selected. The BLE protocol establishes a particular way of transmitting data, which consists of the concept of General Attribute Profile (GATT). According to [8], the characteristics and services that have read/write permissions have specific Universal Unique Identifiers (UUIDs), so when the services of the BLE PGN are obtained, the needed characteristic is found by its UUID.

Once the read/write characteristic of the BLE PGN is correctly set, the application enters a state in which it can receive and send messages over the established BLE link, by writing on the characteristic's value. The characteristic is set to run a GATT (General Attribute Profile) callback function when its attribute value changes (by the remote device, not by the application

itself), that passes the new value, or, in other words, the received array of bytes, to the message processor.

At last, the *stop* method can close the BLE link and release any resources used by *BLEHelper*, so the application can safely go into background or close.

3.2.2 Message Handling

Moving on to the next big part of PGNAgent, the message handling modules are covered in this section. Its main element is *PGNMessageProcessor*, a Java class that takes a Controller module role, and is widely used in the connectivity modules as well as in the implemented activities because it serves as a link between them. Besides, there is a very important Model module, called *PGNMessage*, which standardizes the Prometeo Communication Protocol (PCP) messages. After these modules are explained, the *Node* class will briefly be covered, as it models a sensor node from the WSN. It belongs to the Model block and is what the "message handling" updates and gives to the activities.

- *PGNMessage*

In order to understand *PGNMessageProcessor,* it is necessary to go through the *PGNMessage* module first, and thus Appendix I on the PCP. *PGNMessage* is a POJO class (Plain Old Java Object) that defines an object with certain attributes and many needed hardcoded arrays of bytes and constants. This object models a message that implements our protocol, and then its attributes contain the values for each field of the message (see Appendix I on PCP to understand the fields of a message). The constructor, therefore, is implemented as follows:

```
public PGNMessage(byte pgnLength,
                  byte targetId,
                  byte directionCode,
                  byte messageCode,
                  byte[] dataPayload) {
    this.pgnLength = pgnLength;
    this.targetId = targetId;
    this.directionCode = directionCode;
    this.messageCode = messageCode;
    this.dataPayload = dataPayload;
}
```

This way, we let the application handle data in an easy way. Of course, there are many more methods needed in *PGNMessage*, apart from all the setters and getters for all the private attributes that typically appear in a POJO class and the defined constants. Regarding these constants, we have all the possible messages in the PCP stored in static and final values. In order make it easy for the rest of the application to equally use the names and arrays of bytes of the messages, we implemented some bidirectional *HashMaps,* so that all messages are linked together by their String values and byte arrays as “key” and “value” and vice versa. In order to accomplish this, a Google library, called Guava [10], had to be imported into the project. These bidirectional *HashMaps* are called *BiMaps,* and the following code shows how they are built, by using a byte value and its associated String:

```
public static final byte GET_NODE_LIST             = 0x01;
public static final String STR_GET_NODE_LIST       = "get_node_list";
…
public static final byte TOGGLE_ALL_LEDS             = 0x15;
public static final String STR_TOGGLE_ALL_LEDS       = "toggle_all_leds";
…
public static final BiMap<Byte, String> MSG_TYPES_BIMAP =
       HashBiMap.create(
             new HashMap<Byte , String>() {{
                 put(GET_NODE_LIST, STR_GET_NODE_LIST);
                 …
                 put(TOGGLE_ALL_LEDS, STR_TOGGLE_ALL_LEDS);
             }}
    );
//Separate BiMap for NUF messages
public static final BiMap<Byte, String> NUF_MSG_TYPES_BIMAP =
       HashBiMap.create(
             new HashMap<Byte , String>() {{
                 put(NUF_NEW_NODE, STR_NUF_NEW_NODE);
                 put(NUF_UPDATE_TEMP, STR_NUF_UPDATE_TEMP);
                 put(NUF_UPDATE_HUM, STR_NUF_UPDATE_HUM);
             }}
    );
```

After that, we have implemented a few complex methods. The most important ones are *buffer* and *getPGNMessage*. The method called *buffer*, when invoked, returns an array of bytes from a *PGNMessage* object, as a correctly built Prometeo Communication Protocol frame,

whereas the method *getPGNMessage* does the opposite process. This was useful for different parts of the application: when a message has to be sent, connectivity helpers just need an array of bytes and do not care about its meaning, but the activities need the information inside those messages, which is much more accessible if they are POJOs with attributes.

- *PGNMessageProcessor*

With *PGNMessage* implemented, the *PGNMessageProcessor* tasks become easier. When an instance of the *PGNMessageProcessor* class is created, the activity that did it tells whether it wants a *USBHelper* or *BLEHelper* instance, but the processor object always uses methods that are defined in the *ConnectionHelper* interface, so it is transparent to it. There are two main methods in the *PGNMessageProcessor* class.

The first method implemented in *PGNMessageProcessor* forwards a message sent from an activity (when the user chooses to do so) to the connectivity handler. Figure 10 shows the process by which this is done:

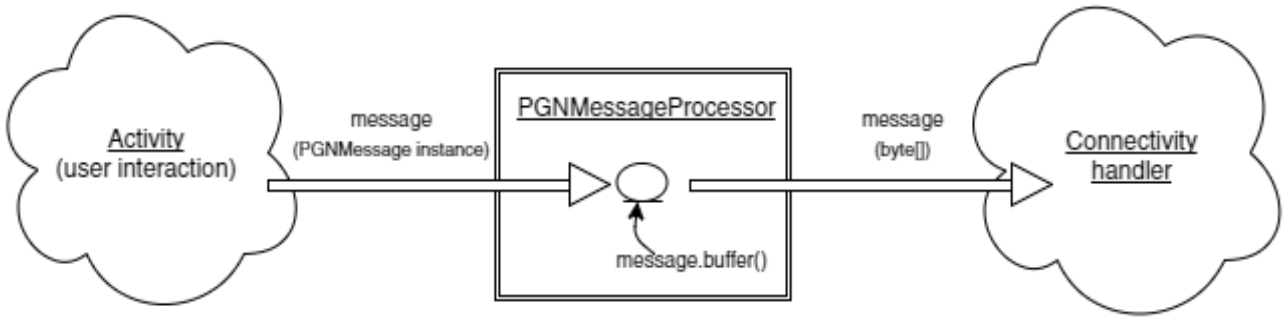

Figure 10: Message from an activity to a connectivity handler

As simple as the communication from the application to the outer interfaces is, the opposite direction takes a more complex process. It is implemented by a method called *processReceivedMessage*, which is called by connectivity handlers when they receive data from the USB or BLE interface. This method was implemented with a D&C (Divide and Conquer) type algorithm, in which we have a few switch statements that break down the problem to be solved in smaller problems, iteratively, until the problem is easy enough to be directly solved. In this case, when a byte buffer is received from the connectivity handlers, the *processReceivedMessage* method starts to parse it by checking its fields according to PCP. For

example, we distinguish if it is an ACK message of a NUF message, and after that we check its message code, according to what is explained in Appendix I.

When *PGNMessageProcessor* finally knows what to do with the received message, it updates some data that is defined by a Model module called *Node*, and notifies the activities about it.

- *Node*

This Java class models an element from the WSN; a node with sensors that can measure temperature and humidity and has two LEDs. It was implemented almost like a POJO, but it implements *java.io.Serializable*, so it can be passed between activities as it will be explained in the next section. The following code shows a summarized implementation of the *Node* class.

```
public class Node implements java.io.Serializable {
…
    public Node(int id, int temp, int hum) {
        this.id = id;
        this.temp = temp;
        this.hum = hum;
    }
    //SETTERS AND GETTERS
    …
}
```

As can be seen, it is a very simple module, but crucial for the functionality of the application. *PGNMessageProcessor* creates and updates a *List* of *Node* objects, and sends it to the activities so that they can refresh the UI by just accessing the attributes of the *Node* objects.

3.2.3 User Interface: activities

In this section, we enter the graphic part of the application, as well as the interface for user interactions. Due to how activities work and the role they take in this project, a use case scenario is best for explanations about them. Parallel to the use case, each activity will be explained, and some of them will include an explanation for some graphic model components.

- *MainActivity*

When PGNAgent is first opened, *MainActivity'*s layout appears. It shows a simple presentation with the application's logo and two buttons. The first button (BEGIN) starts the applications functionality: connecting to the PGN. The second button (ABOUT) shows an activity

called *AboutActivity* that displays general information about the application. We can see its layout in the following figure, when it renders its associated XML layout file, *activity_main.xml*:

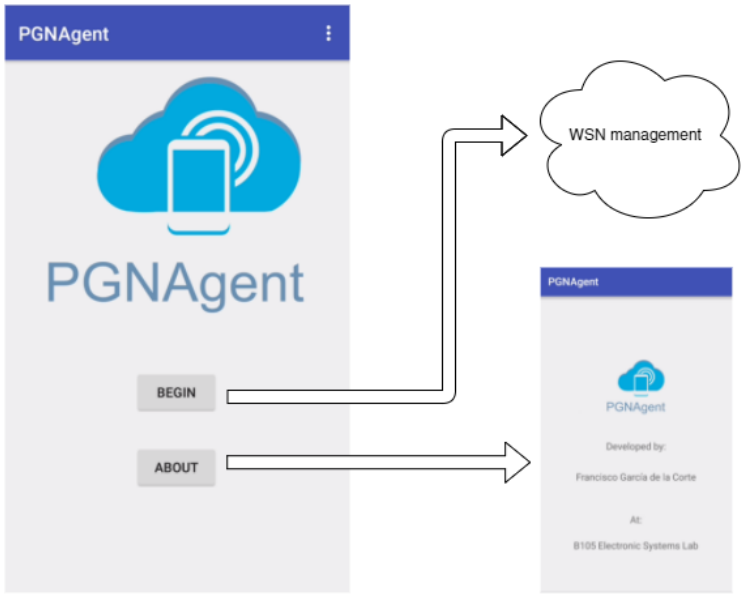

Figure 11: MainActivity layout and actions

- *WSNActivity*

WSNActivity holds the biggest responsibilities in PGNAgent. Its associated XML layout file is *activity_wsnmap.xml*, but graphic operations are divided into more modules than just *WSNActivity*. The following figure shows how *WSNActivity* starts, and the case that the user chooses BLE as the connection type:

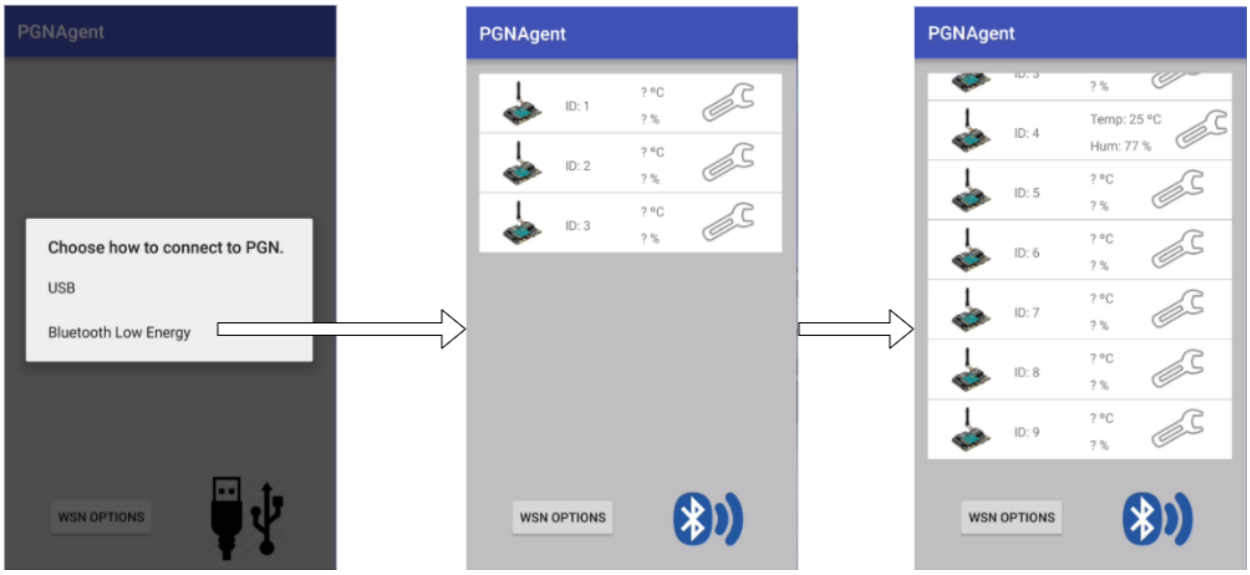

Figure 12: WSNActivity when BLE is chosen

Since Figure 12 is a test scenario, there are only 3 sensor nodes in the WSN in the middle screenshot of PGNAgent, but the right screenshot shows how it would look like when some time has passed and the WSN has notified PGNAgent about new nodes appearing. Also, the node identified by the number 4 shows its temperature and humidity samples.

The graphic management of the list of nodes is implemented by an independent graphic module with a Java class and an XML layout file of its own, which are *NodesListViewAdapter* and *list_view_node.xml*, respectively. A *ListViewAdapter*, in Android, is a class that adapts a *ListView* in a layout to a custom graphic design and parameters, according to [11]. In this case, *NodesListViewAdapter* takes all the information about nodes (this is where *Node* class comes in handy) and distributes it to the elements in the nodes list, becoming a scrollable list when its length surpasses the capacity of the screen. This is done every time an update comes from *PGNMessageProcessor,* and then a new instance of *NodesListViewAdapter* replaces the previous one and renders the GUI. The following function, implemented in *WSNActivity*, illustrates it:

```
/**
 * Updates Sensors List View content and renders the UI
 */
private void updateSensorListView() {
	NodesListViewAdapter listViewAdapter =
		new NodesListViewAdapter(this, R.layout.listview_node, updated_nodes);
	nodes_listView.setAdapter(listViewAdapter);
}
```

The button at the bottom of *WSNActivity's* layout, called WSN OPTIONS, shows a *PopUpWindow* (based on the example shown in [12]) that implements a graphic module called *MessageListViewAdapter* with its own layout file, called *listview_item.xml.* It is similar to *NodesListViewAdapter,* but it has a static form as it will always show two PCP messages: "get_node_list" and "get_num_nodes". Since they refer to the WSN as a whole instead of being directed to a particular node, these messages have a constant value, as can be seen in Appendix I. The "get_node_list" message renders the UI due to the reception of a message with just the number of nodes and their identifiers, and the "get_num_nodes" message just asks for the number of nodes. As shown in Figure 13, a *Toast* message appears with the content of the message; in this case, there are 9 nodes in the list.

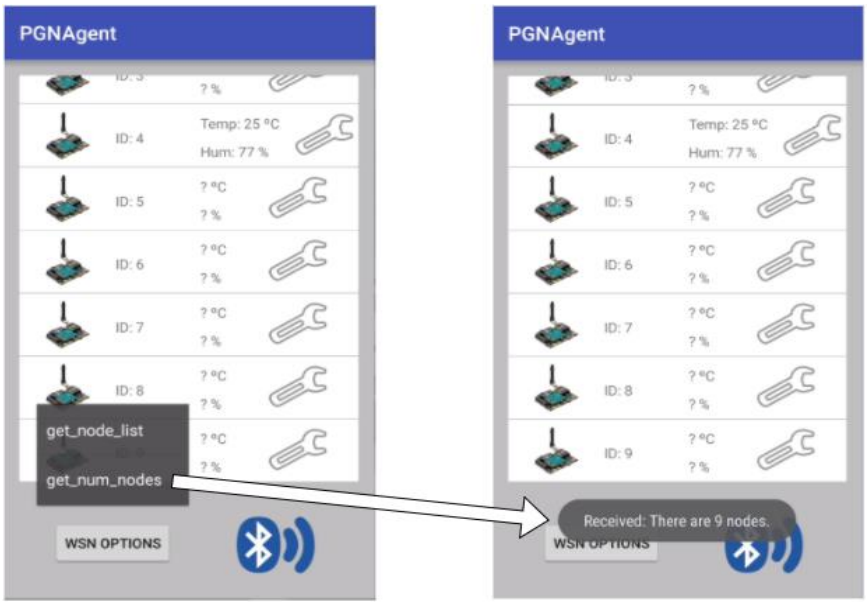

Figure 13: WSNActivity, PopUpWindow and "get_num_nodes" message

- *NodeDetailActivity*

WSNActivity shows a general view of the WSN and a few general commands to it, apart from updating its GUI every time a message that changes any data in the sensor list is received. For a more concrete management of a specific node, the user can touch any row of the nodes list and *NodeDetailActivity* will be launched.

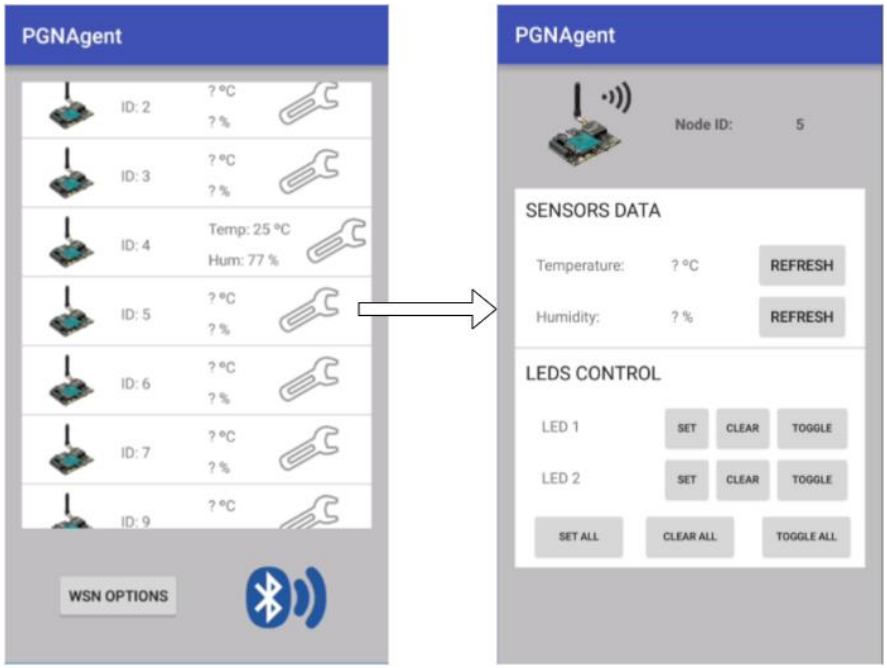

Figure 14: NodeDetailActivity opening

We can see in Figure 14 what happens when node 5 is touched. *NodeDetailActivity*, with data associated to this node, appears. There is a lot going on in this step due to the change of activities. When a node from the list is touched, an *intent* is thrown. An *intent*, in general for Android, is an abstract description of an operation to be performed, and it can be thought of as the glue between activities when used to launch them [13]. When a new activity is launched, the information that the previous one held is lost, unless passed to the new activity through intents. This is why *Node* implements *java.io.Serializable*, because *intents* force the data to be serializable. The following code shows how the intent between *WSNActivity* and *NodeDetailActivity* is implemented:

```
//Launch NodeDetailActivity when node item is clicked
node_row_view.setOnClickListener(new View.OnClickListener() {
     @Override
     public void onClick(View v) {
          Intent intent = new Intent(activity, NodeDetailActivity.class);
          intent.putExtra("nodeList", arrayList);
          intent.putExtra("selectedNodeIndex", position);
          intent.putExtra("conn", ((WSNActivity) activity).getConn());
          activity.startActivityForResult(intent, 1);
     }
});
```

The code was implemented in *NodesListViewAdapter*, due to the fact that it implements the list of nodes in *WSNActivity*. The *putExtra* method inserts data in the intent so that the activity that is being launched can access it. The data that is transferred in the intent is the following: the node list (with all its attributes); the node that was clicked in *WSNActivity*, and whether USB or BLE connection is being used, so that *NodeDetailActivity* can handle a message processor as well.

On the other hand, in Figure 14, we can see *NodeDetailActivity*'s layout. It shows buttons for all possible PCP commands for a sensor node, such as getting its temperature sample or toggling its LEDs. At first, its measurements of temperature and humidity are not known. If the user clicks on the REFRESH buttons, or the WSN itself sends Network Unsolicited Frame (NUF) messages to the smartphone with updates on these measurements, they will instantly appear on the layout of the node in question. In case the actual node's LEDs cannot be seen, a *Toast* message shows when an ACK message for a LED message is received, so that we know the message was received in the other end. Figure 15 shows some of these possibilities:

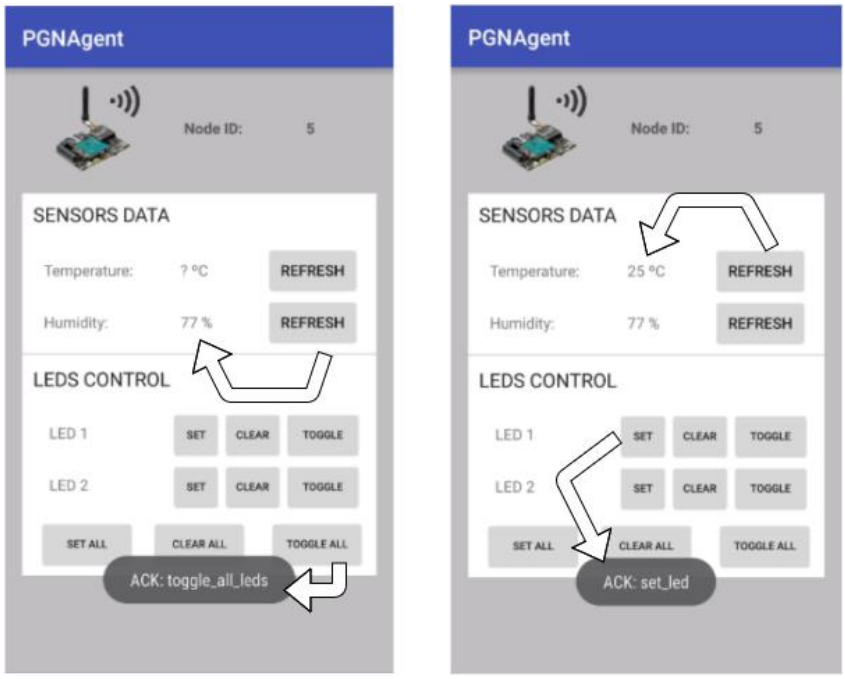

Figure 15: NodeDetailActivity

Finally, in Figure 16 we can see what happens when the back button in Android devices is pressed:

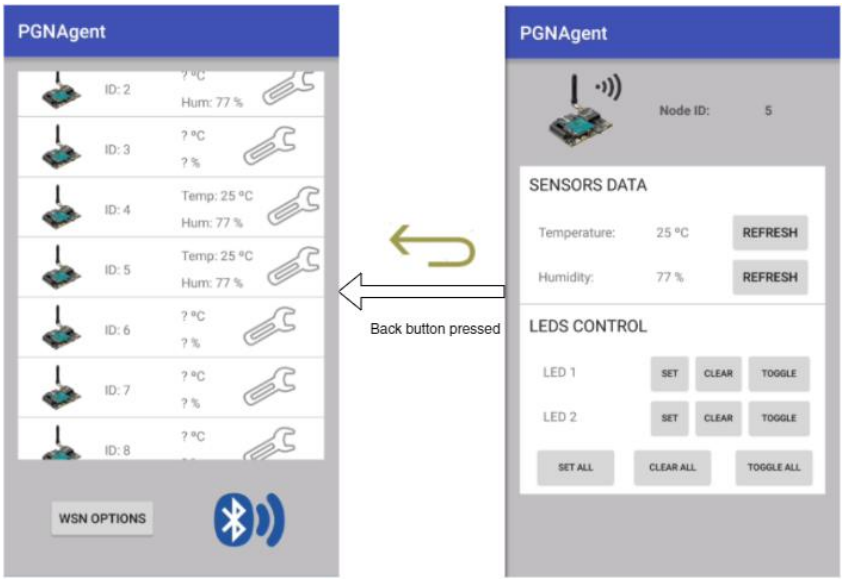

Figure 16: NodeDetailActivity when the back button is pressed.

The information that is received during the time that *NodeDetailActivity* is running is stored the same way that *WSNActivity* does, and when the back button is pressed, this data is

stored in another kind of *intent. WSNActivity* implements a callback function that detects whether it is being launched from *MainActivity* or *NodeDetailActivity*, and if it is the second case, it is expecting a new list of nodes that may or may not be different than the original one that *WSNActivity* had, and renders the UI with the new list. This was implemented following Android's documentation on *onActivityResult* [14], which is the callback function that gathers the data in the *intent* when an activity finalizes.

```
/**
 * Receives new info for nodes, in case it was updated in NodeDetailActivity
 */
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == 1) {
        if(resultCode == RESULT_OK){
            nodes =(ArrayList<Node>)  data.getSerializableExtra("nodeList");
            updateSensorListView();
        }
}
```

Here we end the design and implementation chapter about PGNAgent. Now that it has been gone through, we can go on to the next chapter, which will break down the new design for an alternative PGN device, allowing BLE communications.

4. Designed hardware: BLE PGN

This chapter will cover the second and last part of this project: the Printed Circuit Board (PCB) that was designed as an alternative to the previous project's PGN device, establishing requirements of our own and basing the design on the previous one.

Firstly, the requirements will be laid out, followed by a general diagram that represents the BLE PGN main modules and their connections; all of this in the design subchapter. After that, the implementation subchapter will go over the software that runs inside the used microcontroller, and then the electronic modules that were included as a change to the previous PGN will be detailed, including an explanation of the power system. Finally, the designed PCB's layout will be shown, with some descriptions about its major details.

4.1 Design

4.1.1 Requirements

The modification of the previous PGN device was destined to fulfill the following requirements:

- Eliminate the USB wire in the smartphone-PGN interface, by establishing a wireless communication. Since this requirement is the most important of all, it conditions the rest of the design and the firmware that is implemented.
- Emulate the previous PGN's behavior in the smartphone-PGN interface. In order to do so, the BLE PGN's firmware has to be able to answer PGNAgent's requests and send NUF messages, according to the PCP. The NUF messages need to be somehow emulated, since the previous PGN's firmware does not implement the new PCP.
- Have an autonomous energy source since it no longer can be powered from the USB wire. Its power system will be based on a lithium battery. BLE PGN has to implement a charger for this battery as well, so the system is more self-contained.
- Achieve a smaller board size for the PCB. As USB modules are not necessary anymore, the board's layout can be considerably shrunk.

After the requirements are set, and paying attention to the previous End of Degree Project that developed the first PGN device [15], Figure 17 collects the ideas that matter in the design of BLE PGN:

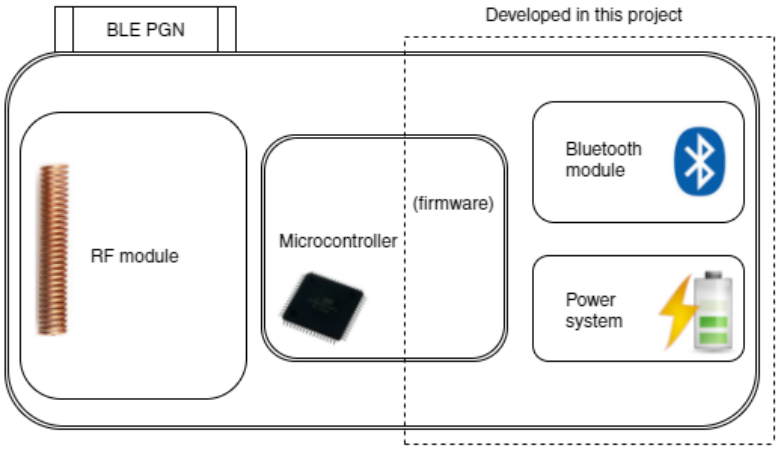

Figure 17: Modules that fulfill the requirements for BLE PGN

4.1.2 Architecture

The designed architecture makes some modifications over the previous one [15], based on the requirements listed in the previous section. Figure 18 shows a general vision of the chosen modules and their connections:

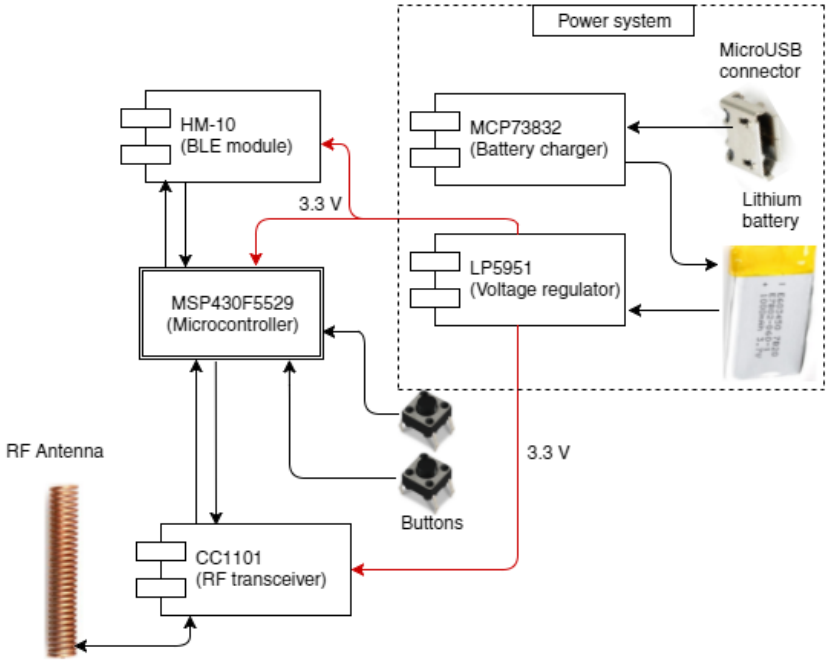

Figure 18: BLE PGN architecture diagram

We maintained the microcontroller and the RF transceiver module from the previous PGN, and we added a Bluetooth module that encapsulates the BLE protocol in a Universal Serial Asynchronous Receiver/Transmitter (UART) type connection; a power system block that contains two modules of its own (a battery charger and a voltage regulator), a lithium battery and a micro USB connector from which the battery can be charged.

Also, the firmware that runs inside the microcontroller (detailed in section 4.2.1) is changed as well, so that it can emulate the PGN's behavior and implement PCP. The emulation of the previous PGN's interface with a smartphone is needed because it runs an old version of the PCP while our app is designed with the PCP in mind. Adapting the firmware of the previous PGN to the current PCP is beyond the scope of this project, and is not implemented yet. The buttons were included in the design to emulate NUF messages from the WSN.

4.2 Implementation

4.2.1 Firmware

In this project we focus on the BLE PGN-PGNAgent interface, so the logic that uses the RF module is not necessary. Knowing this, the implemented firmware in the used microcontroller emulates the unrequested NUF messages that come from the RF interface with two buttons. It also has a list of predefined response messages to emulate the network's answers to all the possible application's requests, according to the PCP.

These messages are sent over the UART port (delegating the transmission/reception of data over the Bluetooth interface of the PGN to the Bluetooth module that is connected to it), and two service routines dispatch the button press interruptions.

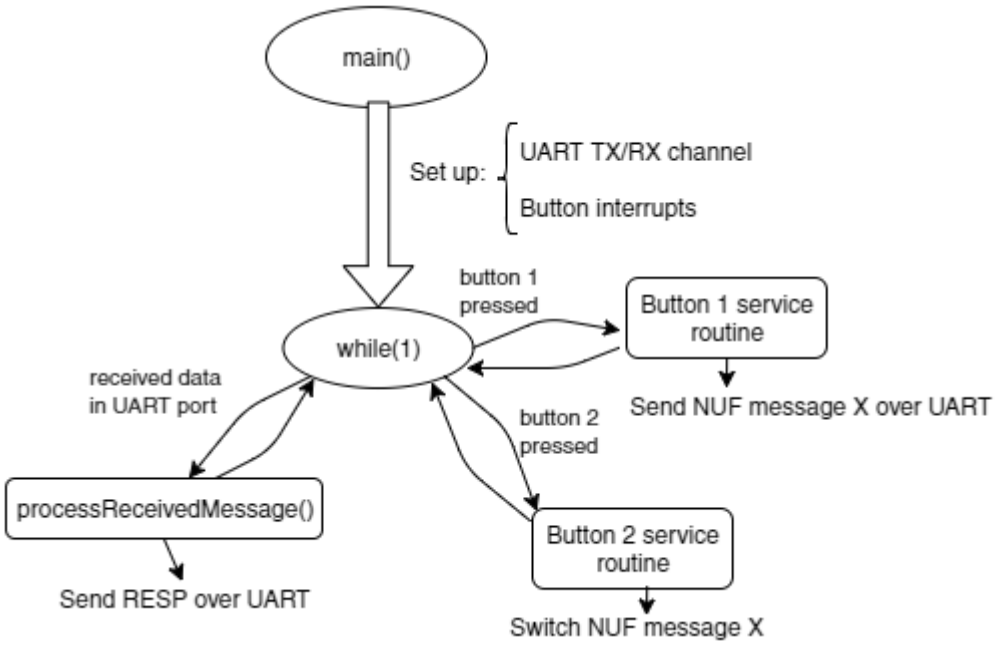

Figure 19: BLE PGN firmware logic diagram

Figure 19 shows how the firmware functions inside the implemented microcontroller (detailed in section 4.2.3), in the *main.c* file. The *main* function starts by configuring the UART

channel (pins, baud rate, etc.) to communicate with the interface provided by the Bluetooth module (detailed in section 4.2.2) and the button interrupts. After that, a loop runs with the purpose of waiting for interrupts.

There are three possible ways of interrupting the loop. One of them is the reception of data in the UART serial port, which leads to calling the function *processReceivedMessage*. This function acts similarly to the homonymous method used in the PGNAgent's message processor (see section 3.2.2 on how PGNAgent dispatches received messages), but in this case it sends Response frames (RESPs) following the Prometeo protocol (PCP), responding to whatever message is received, as long as it follows the protocol.

The other two interrupts are associated to the buttons. At the beginning of the *main* function, two General Purpose Input/Output (GPIO) pins are set, with the help of the guidelines given in MSP430x5xx microcontrollers family User's Guide [16], to call a service routine when a high-to-low transition is detected in the selected pins (more details on this in section 4.2.3, which covers the hardware implementation of the used microcontroller). These pins are connected to the buttons, and when they are pressed a high-to-low transition occurs. One of the button's service routine sends a pre-established NUF message, and the service routine of the other button switches the message to be sent with the first button. All the possible messages that PCP offers are stored in a header file called *PGN_protocol.h.*

This program is briefly based on the one implemented in the previous PGN, so that only the UART configuration was recycled. Most of the coding of *PGN_protocol.h* and *main.c* had to be done in this project, as well as some minor changes in *BCUart.c* (enable BLE communications instead of USB by changing the used serial ports).

This way the previous PGN is correctly emulated, as it answers to any commands received over the BLE link (and, therefore, in the microcontroller's UART channel). Although, this implementation works for BLE PGN with the mentioned minor changes, because it uses different pins than the ones that have to be used in order to habilitate USB communications. Later on, in Chapter 5, a slightly different firmware is used for the USB tests (running on a MSP430F5529 Launchpad), but the functionality does not change at all, as the message transference works the same way.

From this point and beyond in this chapter, the implemented hardware modules are going to be laid out, showing their schematics and the main details about their connections.

4.2.2 BLE module: HM-10

The selected Bluetooth module was the HM-10, developed by JNHuaMao Technology Company and based on the Texas Instruments' CC2541 system on chip [17]. HM-10 encapsulates a serial connection that works over two of its pins, so it can be connected to the microcontroller's UART serial port without any additional configurations, according to [8], an article dedicated to HM-10. The Bluetooth modules that had more documentation online and had this UART-encapsulating feature were HC-05 (which is not BLE compatible) and HM-10, but we chose the second one when we decided to use BLE. There is also the HM-11 module, which is a bit smaller but harder to find.

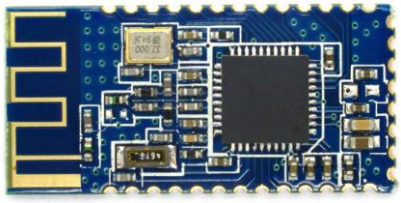

Figure 20: HM-10

The schematic for this module was implemented as it is shown in the HM-10's datasheet [17]. Pins 1 and 2 are connected to the UART serial port of the microcontroller. There is a LED in the schematic that behaves like a blinking light when the device is not paired, but stays still when it is. Figure 21 shows this schematic, which is included in BLE PGN's group of modules:

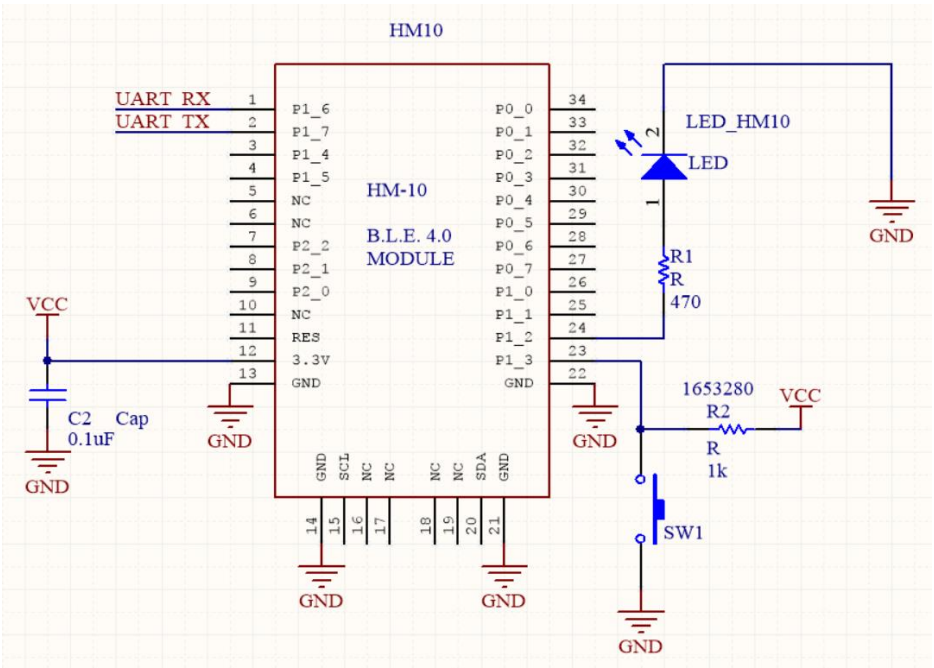

Figure 21: HM-10's schematic

4.2.3 Microcontroller: MSP430F5529

As the previous PGN did, and in order to provide compatibility with it, we have used the MSP430F5529 microcontroller. It is offered by Texas Instruments, and it is intended for low-power applications. The connections that had to be changed were the ones that provided USB communications, so we relied on the MSP430F5529's datasheet [18] to be guided when connecting its pins.

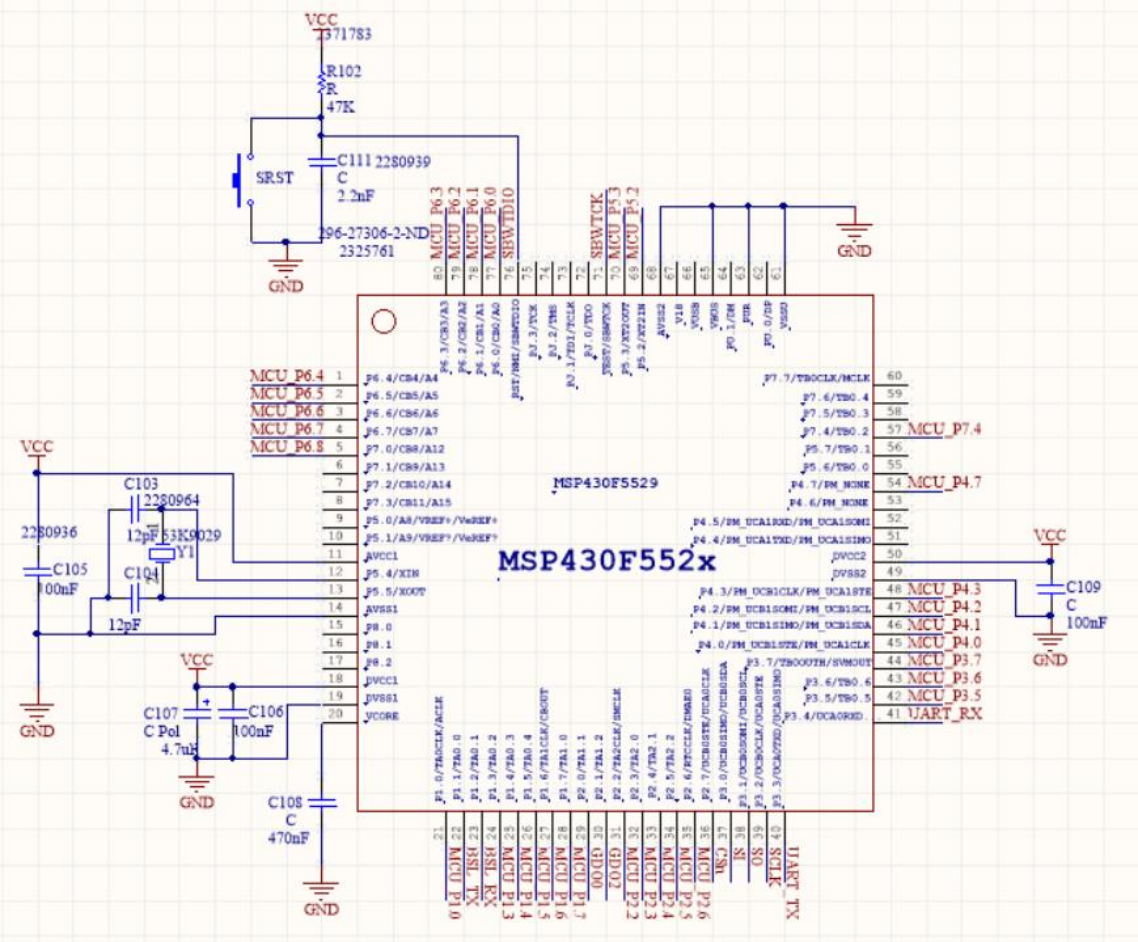

Figure 22: MSP430F5529's schematic

Figure 22 shows the used schematic for the microcontroller. Two of the pins are connected to HM-10 and establish the serial communication with it. There are two other pins from the microcontroller that are connected to the buttons that were mentioned earlier. Figure 23 shows the schematic used for them:

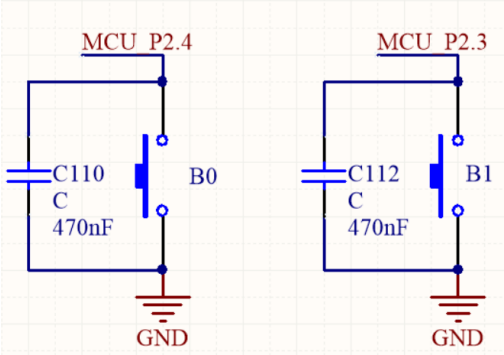

Figure 23: Buttons' schematic

As can be seen, there is a capacitor connected in parallel to the buttons. This works as a debounce solution, a problem in which buttons bounce up and down a few times during some milliseconds before stabilizing. If no action is taken, the firmware would detect all of this bouncing, calling the interrupt's service routine many undesired times. Thanks to [19], we know that a capacitor's charging and discharging time limits the occurrence of this bouncing as the button is pressed or released.

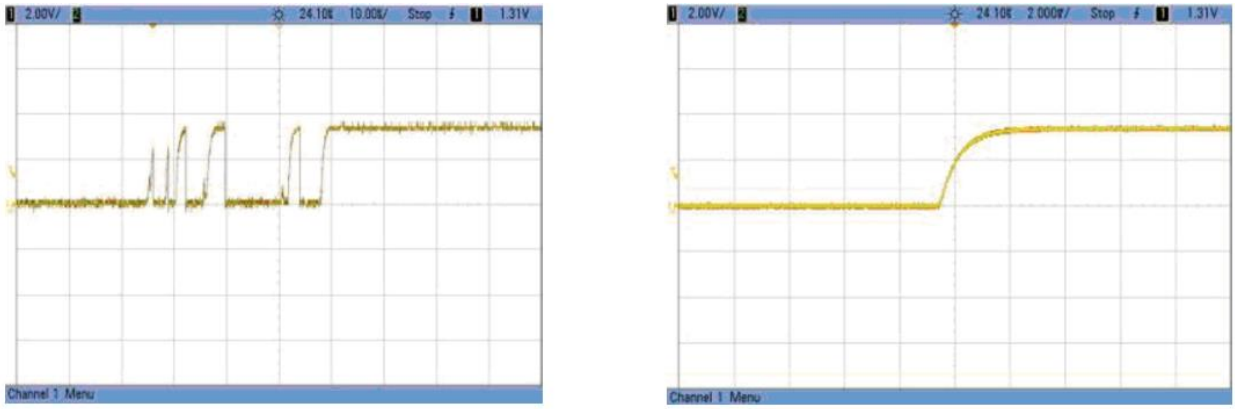

Figure 24: Button bouncing (left) and same scenario with debouncing circuit (right)

The used GPIOs pins have pull-up resistors of 35 kΩ (according to the microcontrollers datasheet), and the capacitor that is in parallel with the buttons has to comply with a time constant (which is the product of the resistor and the capacitor, the RC value) in order to avoid the bouncing effects. This time constant is a fraction of what the capacitor takes to charge or discharge, and has to be longer than the bouncing time. Thanks to experimental tests, it was concluded that a 10 ms time constant worked well, meaning that the required capacitor needed to be a 470 nF one (the closest commercial value to the calculated one).

4.2.4 Power system

The implemented power system controls the circuit's power supply, making the system be able to work in an autonomous way. In order to make the PCB smaller, most of its components were placed in the bottom layer of the board, as will be seen in the section 4.2.5.

In order to manage the circuit's power supply properly, the power system has to accomplish some objectives. The first one is stabilizing this power supply, in case the used modules do not work properly with a different voltage input. By looking at our modules datasheets (HM-10 [17], MSP430F5529 [18] and the used RF module: CC1101 [20]), we conclude that all modules accept a 3.3 V input. The supply voltage, which will come from the lithium battery (850 mAh), is regulated by an LP5951 (voltage regulator). With the help of its datasheet [21], we have built the following schematic:

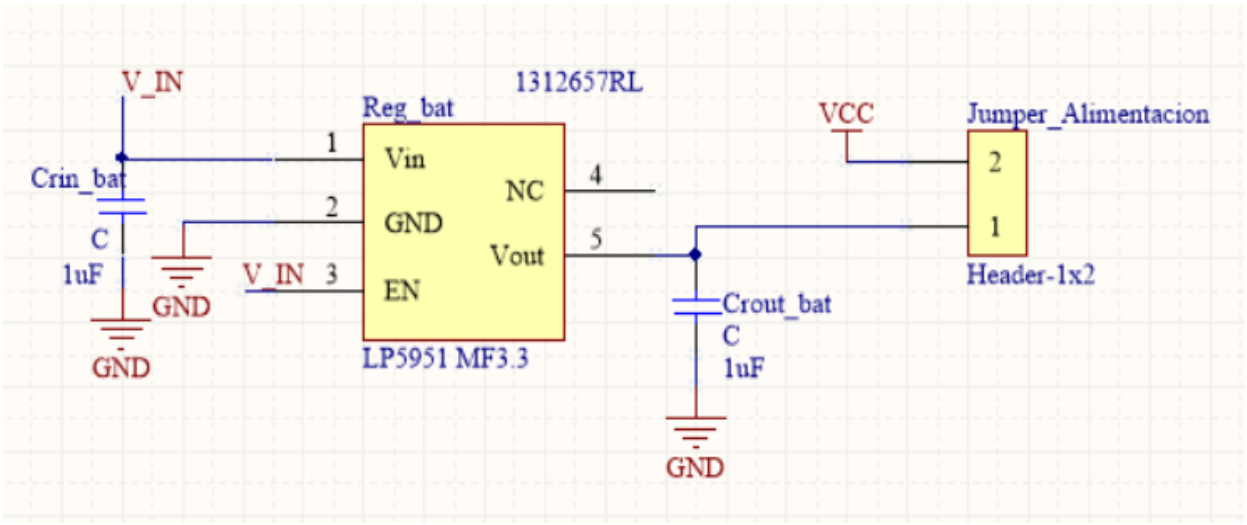

Figure 25: LP5951's schematic

Additionally, we have placed two jumper pins (right side in Figure 25) to prevent the Reverse Current Path (RCP) effect, due to having a different power supply when the microcontroller is programmed. This is because the programming tool provides its own voltage supply to the VCC net, and this would result in a situation where the voltage in the output of the regulator is higher than its input, which can damage both the regulator and the battery. In a normal situation, a jumper is placed in the pins (connecting them), but when the microcontroller has to be programmed, the jumper has to be taken off the header, and that way we avoid the RCP effect.

The second objective for the power system is implementing a charger for the lithium battery. We used the MCP73832 linear charge controller, which gets a 5 V power supply from the USB header. Again, with the help of this module's datasheet [22], we have built its schematic as Figure 26 shows:

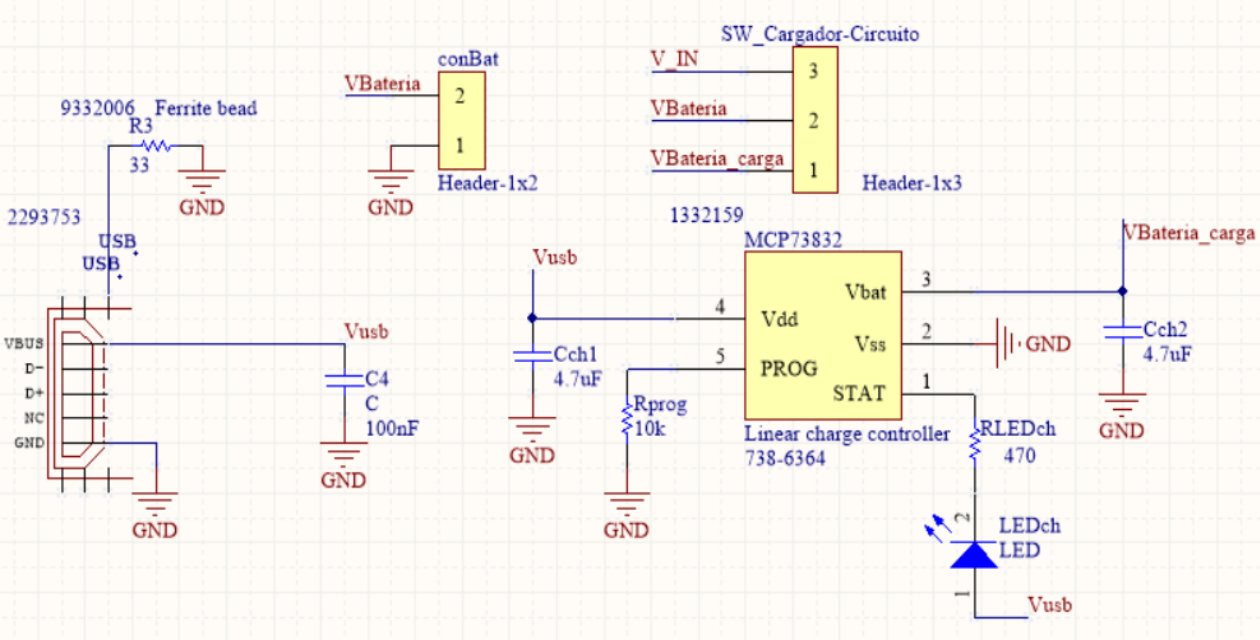

Figure 26: MCP73832's schematic

Also we can see two headers at the top of Figure 26. These headers add a final and simple functionality to the power system: the capability of switching from supplying the whole circuit with the lithium battery to charging this same battery. While the left header is only a connector for the battery, the right header forms a jumper block that connects the charger and the battery, leaving a pin to power supply the rest of the circuitry in BLE PGN.

To sum up, if charging the battery is desired, a jumper that connects the charger with the battery would have to be placed. Otherwise, if supplying the whole circuit is desired, the jumper has to connect the battery and the pin that supplies the circuit.

4.2.5 PCB

Gathering all the schematics, and with the use of Altium Designer, the BLE PGN's layout design is as follows in Figure 27:

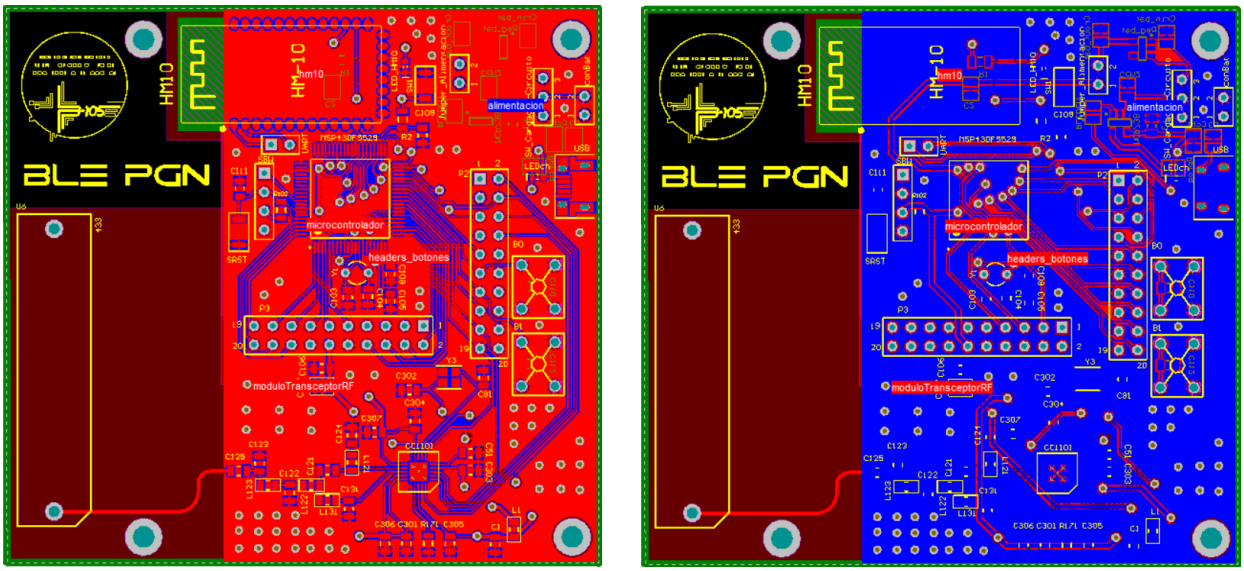

Figure 27: BLE PGN's top layout (left) and bottom layout (right)

The top and bottom layer were both used for routing and placing components so as to make the PCB as small as possible. Compared to the previous PGN (7.7 cm x 10 cm), we saved 2 cm of width in the BLE PGN's board (7.5 cm x 8 cm), thanks to the elimination of the USB modules.

If we take a look at the layouts shown in Figure 27, we can distinguish the different schematics that have just been detailed. At the top, the HM-10 module is placed with its antenna facing the left side, as well as the RF antenna at the left bottom corner, both out of the ground planes to comply with their design specifications. The bottom of the board holds the RF module, and the microcontroller is almost in the top-center spot. Almost all of the power system is placed at the bottom layer of the board, except for a charging indicator LED, the headers and the micro USB connector, which are needed in the top layer.

Figure 28, a photo of the board without the soldered components shows how the actual board looks like:

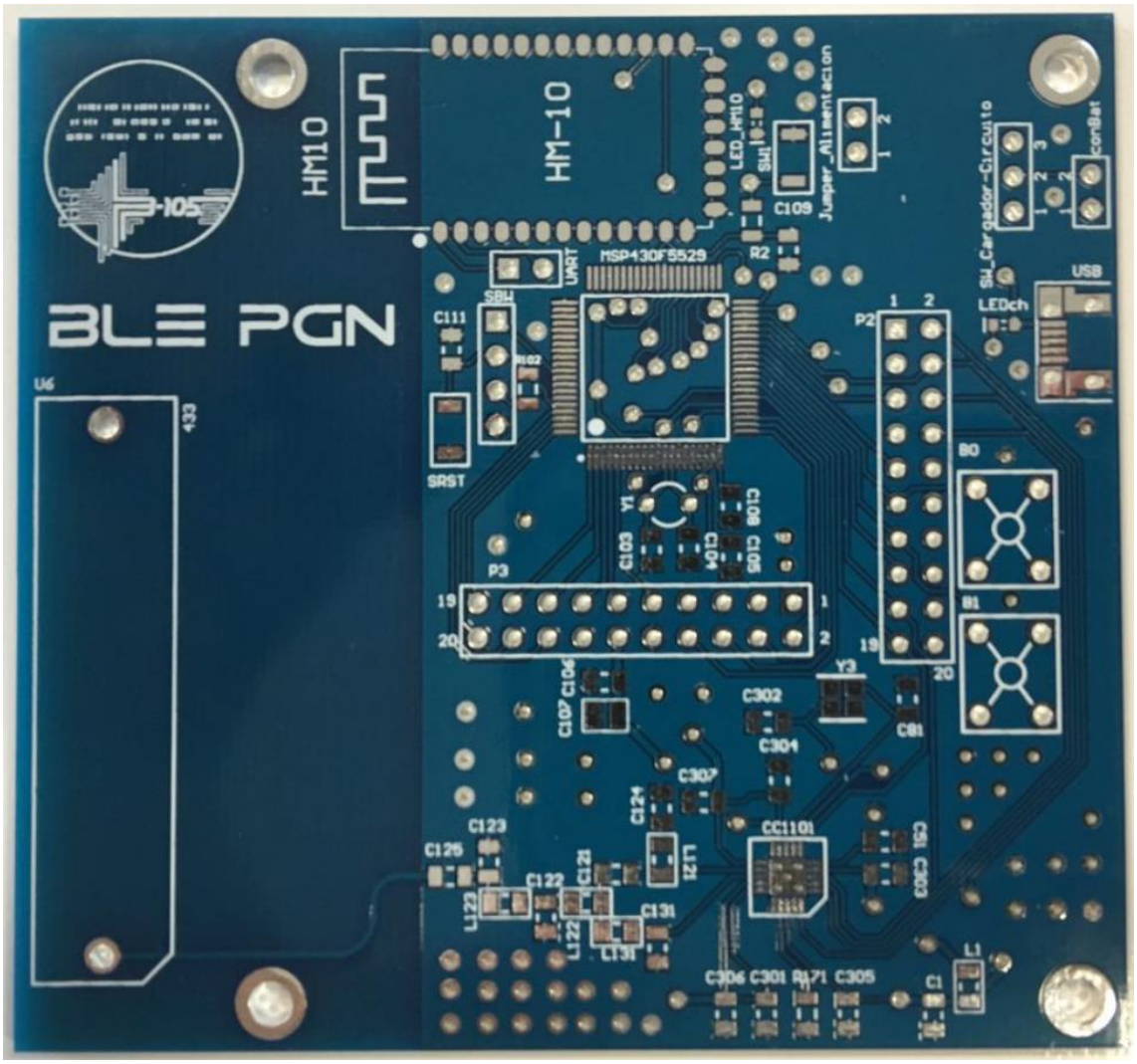

Figure 28: BLE PGN's board

5. Functional testing

In this chapter, two test scenarios are going to be carried out so as to prove the correct performance of the developed application and BLE PGN's firmware. Firstly, the PGN that was developed in the previous End of Degree Project will be emulated with the help of the MSP430F5529 LaunchPad and its Evaluation Kit User's Guide [23]. This was the best option to run the test because this launchpad uses the same microcontroller that the previous PGN and our BLE PGN do. We used the developed firmware (section 4.2.1) on the launchpad for both the USB and the BLE test, with minor changes regarding the UART configuration. Also, a BQ Aquaris M5 smartphone (BLE compatible) was used to run PGNAgent.

The USB and BLE tests will be similar, except that the BLE test will also prove the correct implementation of the debounce solution explained in section 4.2.3. For a better understanding of the tests, Appendix I sheds some light on the PCP that is used in the communication between PGNAgent and the PGN.

5.1 USB

In this test, the USB interface's performance is going to be tested. In order to do so, the launchpad has to be plugged to the smartphone with an On-The-Go (OTG) cable, which lets the smartphone be the host end of the interface and therefore give the launchpad a power supply.

Figure 29: Connected elements for the USB tests

Firstly, we see in Figure 30 how PGNAgent opens. USB is selected, and the state of the WSN is displayed on the screen, with 3 active nodes.

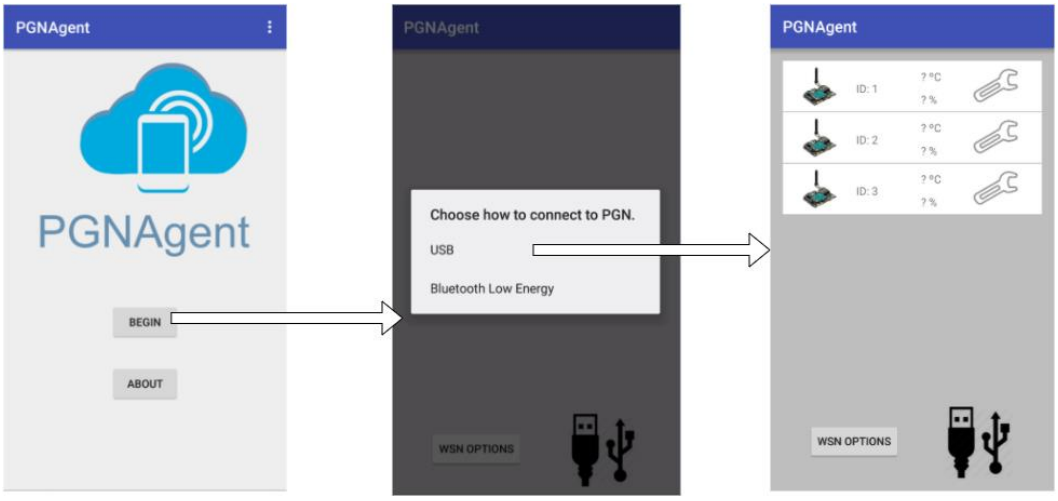

Figure 30: USB test screenshots (I)

After that, Figure 31 shows the specific options for a node when clicked in the WSN state screen. All of the commands offered by the PCP for a single node are available. The request for the selected node's humidity sample is shown in the screenshots.

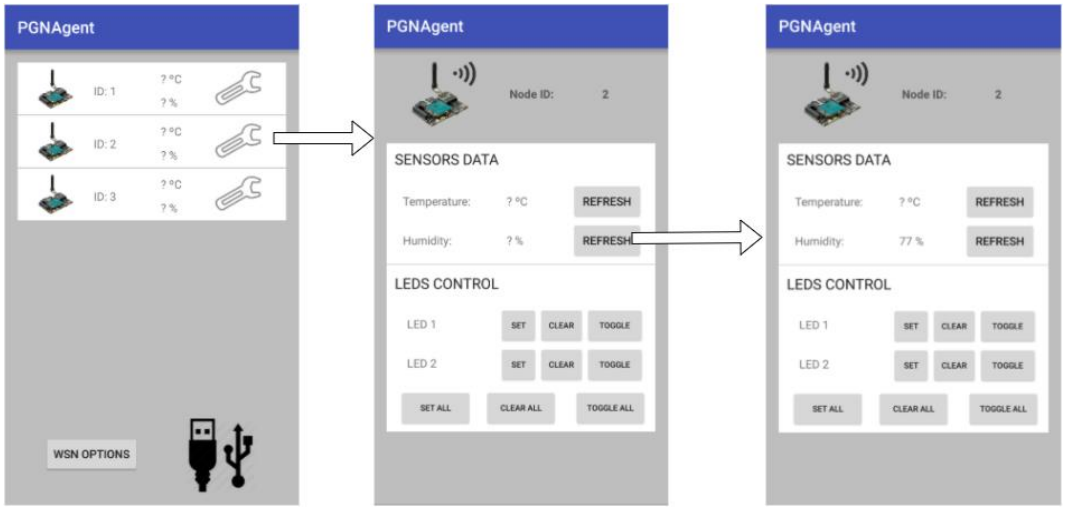

Figure 31: USB test screenshots (II)

Later on, Figure 32 shows the selection of the command "set_led". The application alerts when the RESP message is received as an acknowledgment to the sent command. While the specific node was being managed from its dedicated screen, some NUF messages were received: the appearance of two new nodes and the update of the first node's temperature

sample. This can be seen when the back button is pressed and the screen that displays the WSN's state is back.

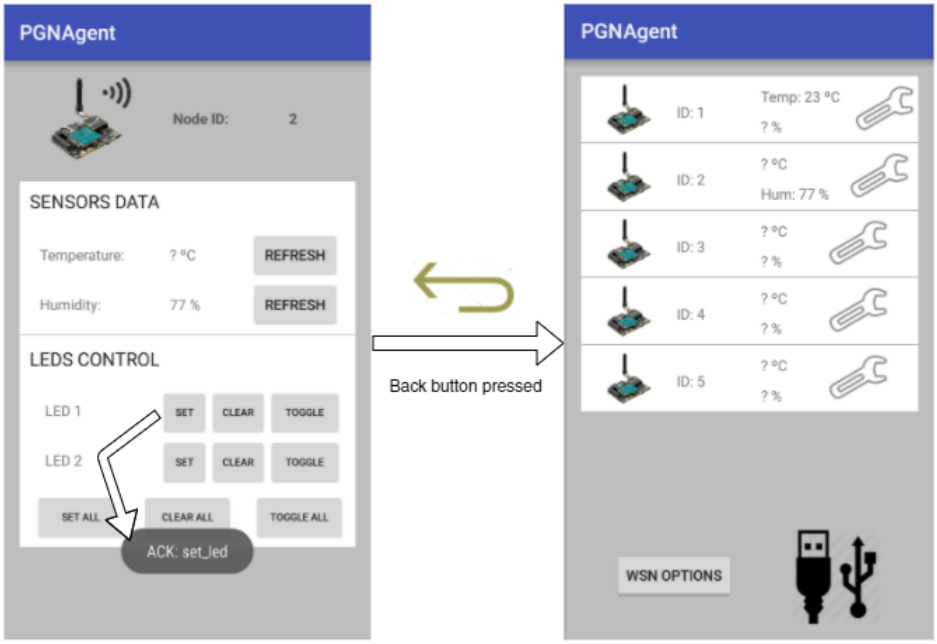

Figure 32: USB test screenshots (III)

5.2 BLE

This test is similar to the USB one, but we also tested the debounce solution that was implemented for the BLE PGN. Figure 33 shows the elements that were used for this test: a smartphone, the launchpad, an HM-10 module and a breakout board with two buttons and capacitors, as specified in section 4.2.3.

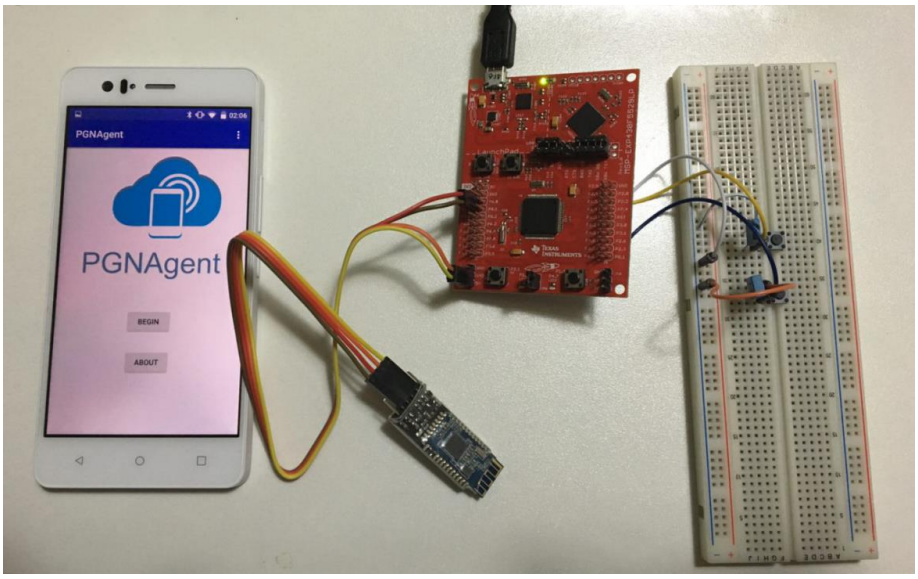

Figure 33: Connected elements for the BLE tests

Firstly, we see in Figure 34 how PGNAgent opens, just like in the USB test, but now BLE is selected. We did not plug in the HM-10 module to the launchpad, so an alert shows:

Figure 34: BLE test screenshots (I)

Now, if we plug in the HM-10 module to the launchpad and repeat the steps, the initial state of the WSN will be shown, just like in Figure 30. We can see this, as well as a pushing our buttons, in Figure 35:

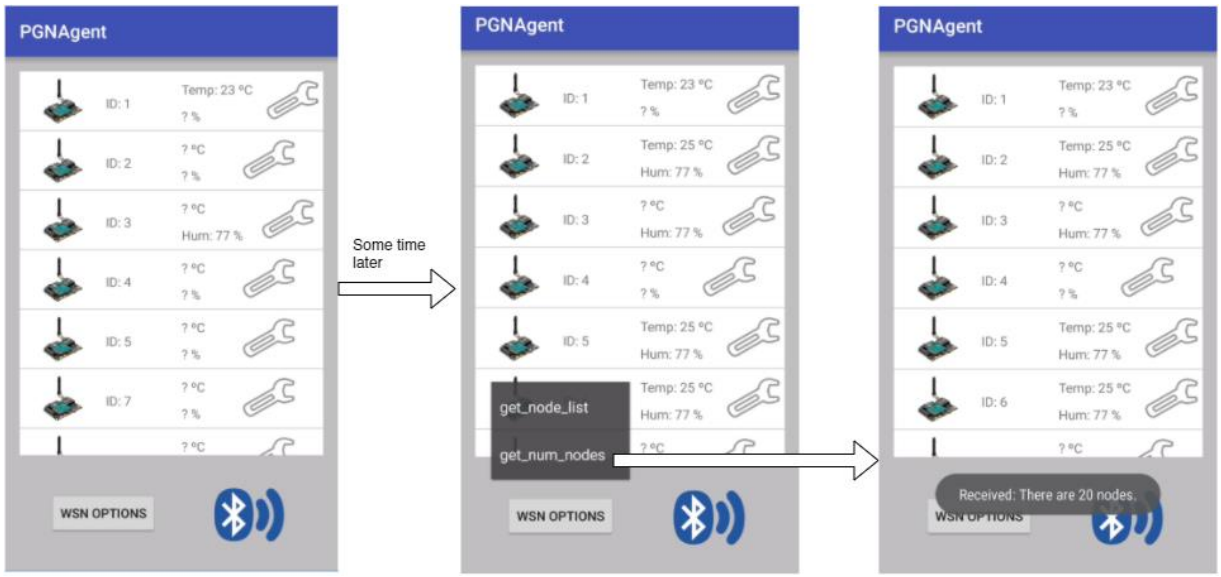

Figure 35: BLE test screenshots (II)

There is a lot going on in Figure 35. The buttons that were pushed sent NUF messages that made the state of the WSN change its parameters, but also we asked some of the nodes about their temperature and humidity examples (they show sample numbers, constants that are

stored in the firmware), as is done in Figure 32 in the USB test. Assuming some time has passed and the network sent some NUF messages (meaning we pushed the buttons that are shown in Figure 33), if the user wanted to know how many nodes there are, there is a PCP command, "get_num_nodes", that is also used in Figure 35. The launchpad responds with the number 20, so a 20 nodes message is shown.

These tests, along with the ones carried out in section 3.2.3 (which are more aimed to explaining how the user interface of PGNAgent works), prove the correct performance of our application, again, with a launchpad that emulates the PGN and BLE PGN's behavior. We used this test-bed because our developed BLE PGN PCB was not available at the time of this testing and the writing of this document. However, the results of these tests are valid as they use practically the same hardware: the microcontroller, the buttons with their debounce solution and the HM-10 module.

If the requirements gathered in sections 3.1.1 and 4.1.1 are read again after these tests, the functional ones are concluded to be achieved. To sum up, the transference of messages in the interface between the smartphone and the emulators of the PGN devices works properly. Also, some non-functional requirements can be seen from the screenshots of the application: it has an intuitive user experience, based on a simple layout and a correct management of the established connections. Although the most important non-functional requirement of this project is the modular design of PGNAgent, and this can be concluded from the implementation explanation that has been given in Chapter 3.

6. Conclusions

This project has successfully designed a proper user interface to manage a WSN and see its state, accomplishing the requirements set in section 3.1.1, and has also designed an alternative PGN device to the one that was designed in the previous End of Degree Project.

Even though we did not have the physical PCB, the tests chapter proves the correct performance of the designed hardware, as it uses most of the modules that were included in the BLE PGN. We had to implement a new firmware to emulate the previous PGN and our BLE PGN, which adopted the new PCP that was not implemented in the previous PGN's firmware. This was necessary in order to test the USB and BLE interface with a smartphone, which was the biggest point of this project, and the full implementation of the PCP in the PGN was beyond the scope of this project, so it was not available for testing.

The development of the Android application was very thorough and complex, because it was the main objective of this project, but it still has some points that could be improved, as the improvement suggestions chapter (Chapter 7) points out. On the other hand, the design of the BLE PGN had less design decisions due to the fact that it is strongly based on the previous project's PGN. Even so, this new design takes advantage of the BLE technology and reduced its size, which makes the development worth the effort.

It is interesting to mention the amount of different programs that had to be used in order to develop this project. The two main programs were *Android Studio*, which is only a few years old and provided a very good experience when PGNAgent was developed; and *Altium Designer*, a very complex program that has so many features it made the development of BLE PGN very tedious, but thorough at the same time. We used other programs to perform smaller tasks, such as *IAR Embedded Workbench*, *Terminal*, *Bitbucket* as a code repository with *Git* and *Vysor*.

As a final point, the main objective of the project has been successfully accomplished: to give the user of a Prometeo WSN a proper user interface from which it is possible manage the network and see its state.

7. Improvement suggestions

This project can be improved in many ways, depending on the characteristics on which we are focusing, such as low power consumption, other kinds of wireless technologies, a web interface… The following list shows some of these improvement suggestions or additional features to the project and how they could be useful:

- Study of a plan for PGNAgent to be launched onto the market. We would have to analyze which would the potential market be, and maybe improve its portability to other kinds of WSNs.
- Development of a web interface. In this case, the user needs to be near the BLE PGN with his or her smartphone, but the user could open a web client in a computer with the same features of PGNAgent. This could be accomplished by developing another device that collected data from the PGN device and sent it over the Internet.
- The PCB design could be even smaller. With a proper study of the RF modules, the use of smaller components and more time used to take advantage of the top and bottom layers of the board, it could be smaller than the one designed here.
- Enable the newly designed PGN to support both communication interfaces with a smartphone: USB and BLE. This could be done with a system that switches from USB to BLE and vice versa, possibly requiring a redesign of the power system.
- Development of PGNAgent for more than only one platform. Even though we justified in section 2.1 why the Android operative system was chosen, if the application's reach were to be expanded, we could take advantage of the design choices that are already done for Android and port them to other OSs.
- Development of a smartwatch version of PGNAgent. Some features of PGNAgent are so simple that they could be done from a very small screen, which could be suitable for a smartwatch.
- Send notifications to the user. When PGNAgent goes into background and the WSN sends any command, the user will not know about it. Notifications in Android are not extremely hard to implement.

8. Bibliography

[1] Jabil | Wireless Sensor Networks: Bringing Order Out of a Hot Mesh
http://www.jabil.com/blog/wireless-sensor-networks-bringing-order-out-of-a-hot-mesh/

[2] J. Martin, A. Rozas, and A. Araujo, “A WSN-Based Intrusion Alarm System to Improve Safety in Road Work Zones,” *Journal of Sensors*, vol. 2016, pp. 1-8, 2016

[3] A. Molina-Pico, D. Cuesta-Frau, A. Araujo, J. Alejandre, and A. Rozas, “Forest Monitoring and Wildland Early Fire Detection by a Hierarchical Wireless Sensor Network,” *Journal of Sensors*, vol. 2016, pp. 1-8, 2016.

[4] Gartner Says Worldwide Smartphone Sales Grew 3.9 Percent in First Quarter 2016
http://www.gartner.com/newsroom/id/3323017

[5] Dashboards | Android Developers
https://developer.android.com/about/dashboards/index.html#Platform

[6] BLE Overview – Laird Technologies Wireless Connectivity Blog
http://www.summitdata.com/blog/ble-overview/

[7] Android Applications With MSP430 USB on Mobile Devices
http://www.ti.com/lit/an/slaa630/slaa630.pdf

[8] HM-10 BLE Module
http://blog.blecentral.com/2015/05/05/hm-10-peripheral/

[9] Android Bluetooth Low Energy (BLE) Example – Truition
http://www.truiton.com/2015/04/android-bluetooth-low-energy-ble-example/

[10] Google/guava: Google Core Libraries for Java 6+
https://github.com/google/guava

[11] Layouts | Android Developers (“Building Layouts with an Adapter” section)
https://developer.android.com/guide/topics/ui/declaring-layout.html#AdapterViews

[12] Android PopupWindow show as dropdown list example | Learn Programming Together
http://www.devexchanges.info/2015/02/android-popupwindow-show-as-dropdown.html

[13] Intent | Android Developers
https://developer.android.com/reference/android/content/Intent.html

[14] Getting a Result from an Activity | Android Developers
https://developer.android.com/training/basics/intents/result.html

[15] Cavo Nuñez L., Diseño e implementación de una pasarela de comunicaciones entre un teléfono móvil inteligente y una red de sensores inalámbricos, TFG B105 Electronic Systems Lab, ETSIT-UPM, Julio 2015.

[16] MSP430x5xx and MSP430x6xx Family User's Guide

http://www.ti.com/lit/ug/slau208o/slau208o.pdf

[17] Bluetooth 4.0 BLE module datasheet, JNHuaMao Technology Company

https://www.seeedstudio.com/wiki/images/c/cd/Bluetooth4_en.pdf

[18] MSP430F552x, MSP430F551x Mixed-Signal Microcontrollers

http://www.ti.com/lit/ds/symlink/msp430f5529.pdf

[19] MSP430 LaunchPad PushButton – Texas Instruments Wiki

http://processors.wiki.ti.com/index.php/MSP430_LaunchPad_PushButton

[20] RF transceiver module: CC1101's datasheet

http://www.ti.com/lit/ds/symlink/cc1101.pdf

[21] Voltage regulator: LP5951's datasheet

http://www.ti.com.cn/cn/lit/ds/symlink/lp5951.pdf

[22] Linear charge controller: MCP73832's datasheet

http://ww1.microchip.com/downloads/en/DeviceDoc/20001984g.pdf

[23] MSP430F5529 LaunchPad Development Kit

http://www.ti.com/lit/ug/slau533c/slau533c.pdf

[24] Four Ways TO Build a Mobile Application, Part 3: PhoneGap – Smashing Magazine

https://www.smashingmagazine.com/2014/02/four-ways-to-build-a-mobile-app-part3-phonegap/

[25] Introduction to SimpliciTI

http://www.ti.com/lit/ml/swru130b/swru130b.pdf

Acronyms

ACK	Acknowledgement frame
ADB	Android Debug Bridge
AP	Access Point
API	Application Programming Interface
ATT	Attribute Profile
BLE	Bluetooth Low Energy
D&C	Divide and Conquer
ED	End Device
GATT	General Attribute Profile
GPIO	General Purpose Input/Output
GUI	Graphic User Interface
LED	Light Emitting Diode
MVC	Model, View and Controller
NUF	Network Unsolicited Frame
OS	Operative System
PAN	Personal Area Network
PCB	Printed Circuit Board
PCP	Prometeo Communication Protocol
PGN	Portable Gateway Node
POJO	Plain Old Java Object
RCP	Reverse Current Path
RE	Range Extender
REQ	Request frame
RESP	Response frame

RF	Radio Frequency
RX	Reception
TFG	Trabajo de Fin de Grado
TI	Texas Instruments
TX	Transmission
UART	Universal Asynchronous Receiver/Transmitter
UI	User Interface
USB	Universal Serial Bus
UUID	Universal Unique Identifier
WSN	Wireless Sensor Network
XML	Extensible Markup Language

Appendix I: Prometeo Communication Protocol

The Prometeo wireless sensor network is based on SimpliciTI, a simple low-power RF network protocol offered by Texas Instruments aimed for battery-operated devices [25]. In the SimpliciTI protocol, there are three possible roles for the nodes: Access Point (AP), End Device (ED) and Range Extender (RE). Applying this to the Prometeo WSN, and to what this project is concerned about, we only consider the roles of AP (a unique central node) and EDs (several peripheral devices).

Messages that go through the interface between the PGNAgent app and the PGN (via USB or BLE) implement the Prometeo Communication Protocol. It has 13 possible different messages, divided into two big categories: requests from the smartphone to the WSN (specific nodes or the WSN as a whole, so it is sent to the AP) and unrequested notifications from the WSN, or NUF (Network Unsolicited Frame) messages. Nevertheless, each one of these messages gets a response from the other side, confirming the reception of the message and consequently doubling the number of messages, up to 26.

Each message is built by five fields in a frame of at least 6 bytes. All those fields carry a certain meaning, except for the last two bytes, which just establish the end of the frame.

LENGTH	TARGET ID	CODE	DATA PAYLOAD	EOF
(1 byte)	(1 byte)	(2 bytes)	(X bytes)	0x0D0A (2 bytes)

Table 4: Prometeo Communication Protocol message frame

The first field, LENGTH, counts the number of bytes in the frame, except for EOF and itself. For example, if the frame had 10 bytes, the length field value would be 0x07. The second field, TARGET ID, specifies an identifier for the end the message, distinguishing if it is aimed to the PGN, any ED or the AP.

The CODE field has a structure of its own. It is composed of two bytes: the first byte determines the direction of the message depending on which one of the big two categories mentioned earlier it is included in, and the second byte identifies which one of the 13 messages it is. The first byte can only have one of its bits set, and in the case there are two or more, it means there has been an error. Depending on which bit is set, there are 4 types of messages (both directions in the two big categories): REQ is a request from the app to the network; RESP

is a network's response to a request; NUF is an unsolicited notification from the network to the app; and finally, an ACK is an acknowledgment from the app to a NUF message.

0	0	0	**REQ**	**RESP**	**NUF**	**ACK**	**ERR**

Table 5: CODE field first byte

DATA PAYLOAD depends on the message. If we wanted to send a "get_node_list" message, nothing would be included in the payload, but it would get a RESP message, and that payload would contain the requested information. Finally, the EOF field marks the end of the frame. The following table shows all possible messages and their codifications:

Message	Type	**LENGTH**	**TARGET ID**	**CODE**		**DATA PAYLOAD**	**EOF**
				BYTE 1	**BYTE 2**		
get_node_list	REQ	0x03	AP	0x10	0x01	-	0x0D0A
	RESP	0xXX	PGN	0x08	0x01	NUM_NODES (1 byte) + LINKID_1 + … + LINKID_N	0x0D0A
get_num_nodes	REQ	0x03	AP	0x10	0x02	-	0x0D0A
	RESP	0x04	PGN	0x08	0x02	NUM_NODES (1 byte)	0x0D0A
get_temperature	REQ	0x03	Node	0x10	0x08	-	0x0D0A
	RESP	0x05	PGN	0x08	0x08	TEMP (integer degrees, 2 bytes)	0x0D0A
get_humidity	REQ	0x03	Node	0x10	0x09	-	0x0D0A
	RESP	0x05	PGN	0x08	0x09	HUM (relative, percentage, 2 bytes)	0x0D0A
clear_led	REQ	0x04	Node	0x10	0x10	0x01 for LED1 or 0x02 for LED2	0x0D0A
	RESP	0x03	PGN	0x08	0x10	-	0x0D0A
set_led	REQ	0x04	Node	0x10	0x11	0x01 for LED1 or 0x02 for LED2	0x0D0A
	RESP	0x03	PGN	0x08	0x11	-	0x0D0A
toggle_led	REQ	0x04	Node	0x10	0x12	0x01 for LED1 or 0x02 for LED2	0x0D0A
	RESP	0x03	PGN	0x08	0x12	-	0x0D0A
set_all_leds	REQ	0x03	Node	0x10	0x13	-	0x0D0A
	RESP	0x03	PGN	0x08	0x13	-	0x0D0A
clear_all_leds	REQ	0x03	Node	0x10	0x14	-	0x0D0A
	RESP	0x03	PGN	0x08	0x14	-	0x0D0A
toggle_all_leds	REQ	0x03	Node	0x10	0x15	-	0x0D0A
	RESP	0x03	PGN	0x08	0x15	-	0x0D0A
nuf_new_node	NUF	0xXX	PGN	0x04	0x01	NEW_NUM_NODES(1 byte) + NEW_LINK_ID_1 + … + NEW_LINK_ID_N	0x0D0A
	ACK	0x03	AP	0x02	0x01	-	0x0D0A
nuf_update_temp	NUF	0x06	PGN	0x04	0x08	TARGET_ID (1 byte) + NEW_TEMP (2 bytes)	0x0D0A
	ACK	0x03	NUF origin	0x02	0x08	-	0x0D0A
nuf_update_hum	NUF	0x06	PGN	0x04	0x09	TARGET_ID (1 byte) + NEW_HUM (2 bytes)	0x0D0A
	ACK	0x03	NUF origin	0x02	0x09	-	0x0D0A

Table 6: Prometeo Communication Protocol codification of messages