Daniel Filipe Vicente Batista

Licenciado em Ciências da
Engenharia Eletrotécnica e de Computadores

Development of a Smart Lighting Android-based Application using Bluetooth Low Energy

Dissertação para obtenção do Grau de Mestre em

Engenharia Eletrotécnica e de Computadores

Orientador:	Rui Manuel Leitão Santos Tavares, Professor Doutor, FCT-UNL

Júri

Presidente:	Prof. Doutor Luís Filipe Figueira de Brito Palma, FCT-UNL
Arguente:	Prof. Doutor João Pedro Abreu de Oliveira, FCT-UNL
Vogal:	Prof. Doutor Rui Manuel Leitão Santos Tavares, FCT-UNL

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2016

Development of a Smart Lighting Android-based Application using Bluetooth Low Energy

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] `https://github.com/joaomlourenco/unlthesis` [2] `http://www.di.fct.unl.pt`

Ti Filipe, Pai e Mãe

Acknowledgements

First, I would like to leave a word of gratitude to Professor Rui Santos Tavares, for his help, support, encouragement and patience during the development of this thesis. Specially through harsh times that would not be overcome without his support. I also want to thank all people, from teachers to classmates and university staff that shared this great time of my life at Faculdade de Ciências e Tecnologia.

To my colleagues and now friends André Bispo, André Fidalgo, Filipe Quendera, Miguel Curvelo, Pedro Morgado and Ricardo Madeira, especially this last one for keeping me focus during the stressful moments and for all the help, and the remaining from Inimigos do Velho do Restelo, I hope this friendship that started here stays with us for many and good years.

I would like to thank my oldest friends Luis Ramos, João Rufino, Duarte Alves, Pedro Serras, Pedro Joaquim, Henrique "Eric" Costa, Rui "Escolinha" Pires for the support, encouragement and all the good moments spent together during this stage.

Um grande obrigado aos meus pais, sem eles nada disto seria possível, que apesar de estarem fora, sempre me deram força para fazer mais e melhor, e ás minhas irmãs que foram e são o meu principal apoio na ausência deles, e serão sempre o meu maior orgulho. Queria também deixar um agradecimento especial para a minha tia Preciosa e para a minha prima Marta por toda a ajuda que me deram durante este último ano. Por fim resta-me agradecer as minha avós, Belmira e Joaquina, as minhas tias Emília e Irène, ao meu tio Joaquim, e a todos os meus primos pela preocupação que foram demonstrando durante todo o meu percurso académico e por terem, cada um á sua maneira, contribuído para me tornar na pessoa que sou hoje.

Para a minha namorada Filipa, muito obrigado pela paciência, pelo apoio, pelo carinho, por ter acreditado em mim sempre e tudo o que ela fez por mim durante esta etapa.

Por último, uma dedicatória ao Igor, que infelizmente a vida não o deixou partilhar este momento comigo, e a ti, Ti Filipe que onde quer que estejas espero que estejas orgulhoso de mim, tal como sempre tive em orgulho em ti...

Abstract

The emergence of the Internet of Things (IoT) allowed new developments on home and building automation with devices that provide more power efficiency and adaptation to our needs. Therefore, this thesis presents a study about Bluetooth Low-Energy and its application on a IoT context, through smart devices designed for home applications and to be integrated in smart home system. It is also investigated the advantages and disadvantages of Bluetooth Low Energy (BLE) over other communication protocols for IoT end-devices.

State-of-art Smart Lighting Android-based Application using Bluetooth Low Energy (SLABLE) is implemented with BLE and covers three application layers: first a interactive mobile application for Android OS. Then the middleware to manage communication and the data gathered, implemented in a BLE built-in System-on-Chip (SoC) with the respective programming for tasks as sending and receiving informations or commands and an illumination automatic control, developed in Arduino IDE. Lastly, an hardware layer that consists in sensors and a lamp dimming driver, to be integrated on a circuit board small enough to fit in already installed equipment boxes. The implemented system purpose is a transversal integration between all layers .

Moreover, based on energy consumption study, it is shown that BLE modules are proven to be a good solution for IoT development due to their low-power consumption, also, for data exchange reliability and processing capacity to control and perform several actions at the same time.

Keywords: Bluetooth Low-Energy, Internet-of-Things, Android, Automation System, Light Dimming, Sensing Node, SoC

Resumo

O aparecimento da Internet-das-Coisas(IoT) permitiu novos desenvolvimentos na área da automação para casas e edifícios com recurso a dispositivos que nos oferecem uma melhor eficiência energética e uma melhor adaptação às nossas necessidades. Desta forma, esta dissertação apresenta um estudo sobre Bluetooth Low-Energy e a sua aplicação no contexto da IoT, através de dispositivos inteligentes para aplicações domésticas e para integração em sistemas inteligentes. É também investigado as vantagens e desvantagens do mesmo face a outros protocolos de comunicação para dispositivos IoT.

O sistema doméstico inteligente e interactivo (SLABLE) apresentado no Estado-da-Arte abrange três camadas da implementação: primeiro uma aplicação móvel interativa para Android OS. A camada intermédia para gerir comunicações e recolha de dados, implementada num módulo SoC com BLE embutido, programado para desenvolver tarefas como enviar e receber dados ou instruções e o controlo automático da iluminação, desenvolvido em Arduino IDE. Por fim a última camada consiste em sensores e um circuito de *dimming* para lâmpadas, para serem integrados numa PCB suficientemente pequena para caber em caixas de aparelhagem. O objectivo do sistema implementado é a comunicação transversal entre todas as camadas.

Além disso, com base no consumo de potência, mostra-se que os módulos BLE são uma boa solução para desenvolvimento de aplicações IoT devido ao seu baixo consumo energético, e também, fiabilidade da troca de dados e capacidade de processamento para controlar e realizar várias acções ao mesmo tempo.

Palavras-chave: Bluetooth Low-Energy, Internet-das-Coisas, Android, Sistema de Automação, Controlo de Luminosidade, Nó-sensor, SoC

Contents

List of Figures

List of Tables

Acronyms

ATT	Attribute Protocol.
BLE	Bluetooth Low Energy.
CICS	Constant Illuminance Control Strategy.
DHCS	Daylight Harvesting Control Strategy.
FDMA	Frequency Division Multiple Access.
FHSS	Frequency-Hopping Spread Spectrum.
GAP	Generic Access Profile.
GATT	Generic Attribute Profile.
HCI	Host Controller Interface.
I2C	Inter-Integrated Circuit.
IoT	Internet of Things.
L2CAP	Logical Link Control and Adaptation Protocol.
LDR	Light Dependent Resistor.
LED	Light Emitting Diode.
LL	Link Layer.
MCU	Microcontroller Unit.
NFC	Near Field Communication.
P2P	Peer-to-Peer.

PCB Printed Circuit Board.

PIR Passive Infrared Sensor.

POCS Predicted Occupancy Control Strategy.

PWM Pulse Width Modulation.

RFID Radio-Frequency Identification.

ROCS Real Occupancy Control Strategy.

RSSI Received Signal Strength Information.

SDP Service Discovery Protocol.

SLABLE Smart Lighting Android-based Application using Bluetooth Low Energy.

SMP Security Manager Protocol.

SoC System-on-Chip.

SPI Serial Peripheral Interface.

TDMA Time Division Multiple Access.

UART Universal Asynchronous Receiver/Transmitter.

UUID Universally Unique Identifier.

Chapter 1

Introduction

1.1 Background and Motivation

Nowadays the IoT is a trend topic in technology and it is expected to grow more and more in the future. The first reference to IoT was in 1999 by Kevin Ashton [1] and at the time World Wide Web was revolutionizing the Internet, so it was hard to imagine what would be the Internet-of-Things. Ten years later, the Internet had a tremendous growth, everybody have smartphones and it is possible to be online either by WiFi or 3/4G. The evolution of low-range communication protocols, like Bluetooth, ZigBee, RF-ID or IEEE 802.15.4 enabled the arising of all sorts of wireless accessories like headphones, controllers and wearables.

IoT is also growing in house and buildings automation. Household appliances manufacturers are investing in smart equipments, alongside with researchers to integrate home control systems with their equipments, using low-power communication protocols, to achieve a reduction in power consumption and add more functionalities for users. Those smart systems aims to control remotely air conditioning, heating, illumination or interact with appliances like a fridge or television [2, 3]. Further, the inclusion of sensors allows smart systems to operate based on environment variables, resulting in a more effective automation, and also allows people to have access to information about their home or office. This progress produces not only an higher efficiency in power consumption but also an significant increase in comfort and utility.

Hence the research and development of new architecture and end-devices to be integrated in IoT has a large room for growth. As every new technology trend, IoT is passing by a maturation process and its evolution follows hardware evolution and new low-power communication protocols, as Bluetooth Low-Energy.

BLE is one of the most used communication technology for smart development. Earlier version of Bluetooth were already widely used, so using BLE in IoT is a natural choice, because unlike other communication protocol, Bluetooth does not necessarily need the implementation of specific controllers for application, it already exists in smartphones, tablets and computers. For this reason, Android and iOS are powerful tools for integration in IoT applications based on BLE.

Mobile operating systems also had to adapt to the new reality of IoT. There is already a wide network formed by smartphones built upon the Internet, wherein IoT can be integrated. More precisely, sensor module with communication capacities interacting with smartphones enables their own integration in a wide network using smartphones as gateways.

The motivation for this thesis was to implement a Smart Lighting Android-based Application using Bluetooth Low Energy (SLABLE) based on BLE. The first part was to add a SoC and sensors on the same board to reduce the size, to fit in equipment boxes, and cost comparing to commercial sensor boards [4, 5]. The second part was to study and develop a decision algorithm to run in the SoC to retrieve data from sensors, actuate on the illumination following a power saving profile and communicate with a mobile application. Lastly the integration of a mobile application developed to interact with this specific smart system that allow functionalities as consult temperature and light values, and choosing between an automatic or manual control of the illumination as well.

1.2 Thesis Organization

The present thesis is organized in five chapters, with this one being the introductory chapter. Chapter 2 introduces BLE specification and the engaging concepts, followed by an overview of IoT and the integration of BLE in it. Lastly, it is presented the hardware available with BLE embedded and other non-BLE options. In Chapter 3 it is presented a concept for smart interaction in a house automation and sensing system. The system development is divided in three parts: the controller software, system hardware and the mobile application. Energy and other relevant considerations as Received Signal Strength Information (RSSI) measurements, system performance and implementations costs are discussed in Chapter 4. The conclusions and discussion about this project and how to improve it in the future are analyzed in Chapter 5.

1.3 Contributions

The main contribution of this thesis was a functional prototype that implements software for automation, control and communication upon different hardware components all linked together, forming a SLABLE end-device small enough to be installed in equipment boxes. The development of this thesis allowed understanding the main constraints of designing IoT systems based on BLE communication. Furthermore, the system prototyping process allowed the author of this thesis to learn more about choosing components, PCB design, soldering and the test bench necessary to validate a model. Also, it was possible to develop in different platforms, from Android OS to Arduino IDE, due to the integration required between all parts of the system.

Chapter 2

Bluetooth Low-Energy for Internet-of-Things

2.1 BLE Overview

BLE or Bluetooth 4.0 protocol [6] descends from Wibree, a short-ranged technology developed by Nokia, with the help of Cambridge Silicon Radio (CSR), Broadcom and other companies, with the purpose to consume less power. The Wibree protocol was presented on October of 2006 and the main goal to its development was to complement the versions 1.0 and 2.0 of traditional Bluetooth. The major improvement of the Wibree compared to the traditional Bluetooth is its reduced consumption of the battery. In 2010, the Wibree technology was merged into the Bluetooth standard, by the Bluetooth Special Interest Group, resulting the Bluetooth 4.0 Core Specification or BLE.

2.1.1 Core Architecture

The demand for low-power communications has raised some challenges to traditional Bluetooth, so modifications had to be made to enable low-power communication between devices [7]. The major changes in BLE comparatively to traditional Bluetooth are in radio and protocol stack. The radio has suffer several changes in the number and bandwidth of channels and some blocks from the protocol stack were upgraded to be more power efficient [8].

BLE protocol stack can be divided into three levels, as shown in fig. 2.1:

- **Application**
  The *Application* that will be built on top of the BLE stack to realize a specific task through Bluetooth.

- **Host**
  The *Host* layer comprises the upper level protocols Attribute Protocol (ATT), Generic Attribute Profile (GATT), Generic Access Profile (GAP), Security Manager Protocol (SMP) and Logical Link Control and Adaptation Protocol (L2CAP).

- **Controller**
  The *Controller* layer comprises the low-level protocols Link Layer (LL) and the *PHY layer*.

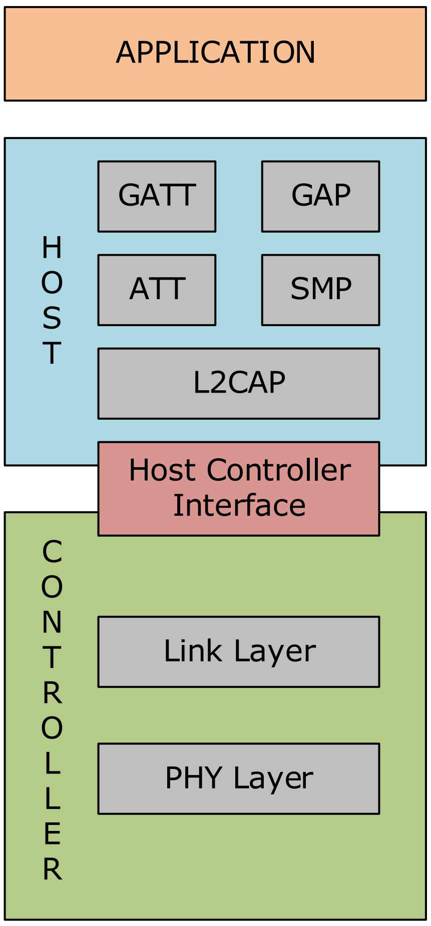


Figure 2.1: The protocol stack in BLE

2.1.1.1 PHY Layer

The *PHY* (physical) layer is where all the circuitry responsible for the transmission/reception of signals and modulation/demodulation of the data can be found. The BLE protocol defines that the radio operates in 2.4 GHz ISM (Industrial, Scientific and Medical) band. Two access schemes are employed: Frequency Division Multiple Access (FDMA), where different frequency bands are allocated for multiple users, and Time Division Multiple Access (TDMA), where a frequency band or channel is divided into time slots for multiple access of multiple users. The band is divided in 40 channels, 37 for connection or data channels and 3 for advertising to apply FDMA. All channels are separated by 2MHz, resulting in a total band that goes from 2.4000 GHz to 2.4835 GHz (fig.2.2). The modulation employed in BLE is the GFSK - Gaussian Frequency Shift Keying, just as classic Bluetooth. To reduce noise, channel interference and fading, Frequency-Hopping Spread Spectrum (FHSS) is used.

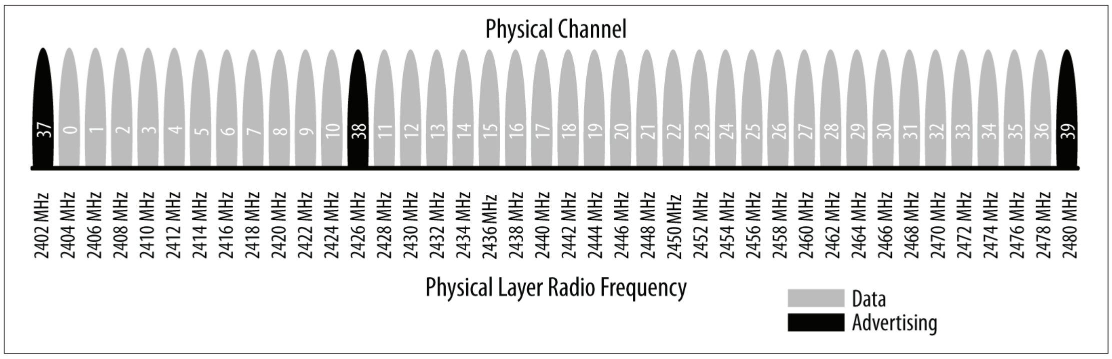

Physical Channel
37 0 1 2 3 4 5 6 7 8 9 10 38 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 39
2402 MHz 2404 MHz 2406 MHz 2408 MHz 2410 MHz 2412 MHz 2414 MHz 2416 MHz 2418 MHz 2420 MHz 2422 MHz 2424 MHz 2426 MHz 2428 MHz 2430 MHz 2432 MHz 2434 MHz 2436 MHz 2438 MHz 2440 MHz 2442 MHz 2444 MHz 2446 MHz 2448 MHz 2450 MHz 2452 MHz 2454 MHz 2456 MHz 2458 MHz 2460 MHz 2462 MHz 2464 MHz 2466 MHz 2468 MHz 2470 MHz 2472 MHz 2474 MHz 2476 MHz 2478 MHz 2480 MHz
Physical Layer Radio Frequency
Data
Advertising

Figure 2.2: PHY Channels [9]

2.1.1.2 Link Layer

The Link Layer defines two roles:

- Master
- Slave

Devices that have the capability to start connections are assigned with the *Master* role. The *Master* can send *Connection Request* and defines the connection timings. On the other hand, devices that are waiting for connections are designated with the *Slave* role and after a connection is established, have to respect the timings imposed by the *Master*.

Other two roles are associated with the ones presented above: *Advertiser* and *Scanner*, that will be explained in SubSec.2.1.2.1.

The specification of Bluetooth 4.0 defines a packet structure, shown in figure 2.3.

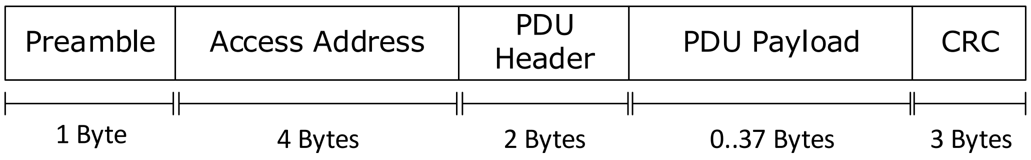


Figure 2.3: The BLE Packet Structure

Preamble The *Preamble* is 1 byte in length and is used for internal protocol management.

Access Address The *Access Address* field has 4 byte and is used to identify communications on a physical link and separate them from communications in proximity using the same *PHY* links but on different physical links. If it is a advertising packet, the address is always 0x8E89BED6, which differentiate them from data packet in which the Access Address is 32-bit number generated randomly.

PDU Header The *PDU Header* differs depending on the type of channel, advertising or data channel. For advertising channels, header contains the advertisement payload type, the address type of the device and *PDU Payload* length.

PDU Payload As in the *PDU* header, payload content depends whether it is an advertising or data channel. For advertising channels, payload contain the advertiser address and advertising data. On a data channel, payload contain control commands or L2CAP data.

CRC The Cyclic Redundancy Check field to detect error in the transmission.

2.1.1.3 HCI - Host Controller Interface

Host Controller Interface (HCI) is the interface that allows communication between the *Host* and *Controller* levels. It is only required when both runs separately on different chips. HCI provides a high level of abstraction to the *Host* layer, that implement more complex protocols, and release it from the hard real-time requirements of the *Controller*. If the *Host* and *Controller* runs on the same chip (ex: SoC), there is no need to have HCI since all three layers (*Application*, *Host* and *Controller*) are implemented in the same Microcontroller Unit (MCU).

2.1.1.4 GAP - Generic Access Profile

In Bluetooth 4.0 the data organization and communication processes differs from traditional Bluetooth. In BLE communication is based on GAP, which defines the low-level interactions between devices [9]. The GAP specifies two roles for the communication:

Central The *Central* stands for devices that are able to search other devices in order to connect with them and are usually smartphones or tablets, due to their increased battery,processing power and memory. Another denomination for this role is *Observer*.

Peripheral The *Peripheral* stands for devices that broadcast advertisement packets and wait for another device with the ability to connect with them to send a connection request. Another denomination for this role is *Broadcaster*.

Just as in LL, the roles *Advertiser* and *Scanner* are also associated with the ones presented above and the connection will be explained in 2.1.2.1.

2.1.1.5 GATT - Generic Attribute Protocol and ATT - Attribute Protocol

Relatively to communication specifically, it is based on two protocols: GATT and ATT. ATT is a low-level protocol that defines how the data is transfered between devices. GATT defines services, through ATT, and all the structure inherent to those services. A service is formed by ATT attributes grouped so they can perform some specific task. GATT defines two roles: server and client. Usually the server is a sensing node that will send information to a client, that can be a smartphone or tablet.

A *service* is defined by a chunk of data and the way it will be handled so the device can perform a specific function or feature. The data present in services are called *characteristics*. A *characteristic* is the lowest level concept in the GATT hierarchy and defines a value used in a service. The *characteristic* can have a *descriptor*, that can store some information about the access, representation and display of the data.

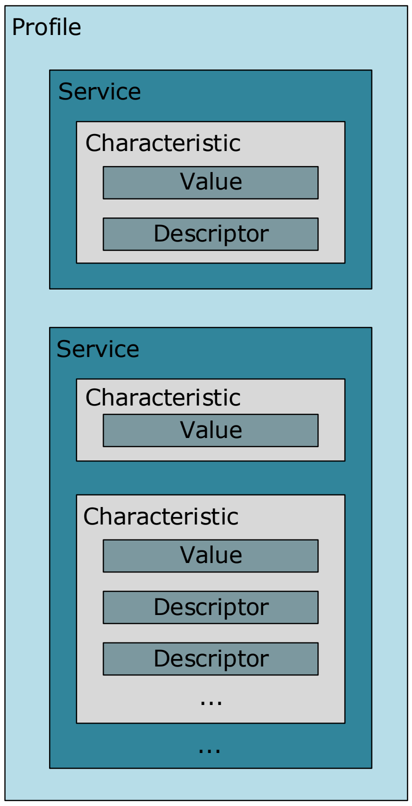


Figure 2.4: Data Hierarchy in a GATT Server

Another important concept in ATT protocol is the Universally Unique Identifier (UUID). An UUID is an identifier, a 128-bit number with the purpose to be unique. BLE is not the only protocol to use them, other communication protocols and applications use UUID too. The usage of UUID is specified in *ISO/IEC 9834-8:2005* [10]. Each *Attribute* has their own UUID, that means that every *Profile*, *Service* or *Characteristic* has their own identifier so it is possible to access them.

2.1.1.6 SMP - Security Manager Protocol

The SMP is both a block and a protocol. The block is responsible for generating and managing encryption keys and identity keys. The protocol defines two roles: *initiator* and *responder*. The roles are similar to the central and peripheral roles of GAP (SubSubSec. 2.1.1.4) respectively. The protocol also defines three procedures that are quite important in the association of devices. Those three processes are:

Pairing Generation of temporary security encryption keys for encrypted links. Temporary keys are not stored and therefore cannot be re-used in the next connections.

Bonding When the objective is to create a permanent connection, after the pairing process security keys are stored and therefore enabling to quickly set up a connection without the need of a bonding process again.

Encryption Re-establishment This process defines how the security keys that were stored are used to (re-)establish the ensuing connections without going through *pairing* or *bonding*.

2.1.1.7 L2CAP - Link Layer Control and Adaptation Protocol

The L2CAP main function is to provide data services to the upper layer protocols [11]. As seen in fig.2.3, there is a standard packet format and L2CAP is responsible to perform the multiplexing into it and fragmentation of packets that come from the upper layer that are larger than the maximum payload length and therefore do not fit into a BLE packet. On the opposite side, it receive multiple packets that have been fragmented, recombine and perform a de-multiplexing into one packet so it can be forwarded to the respective block.

2.1.2 Application Architecture

Application is the top layer of BLE protocol, as seen in fig.2.1. It is in this layer that advertising and connection events are processed and also where services are defined and how the data will be organized into *characteristics*.

2.1.2.1 Advertising

In the advertising process there is two roles defined:

- Advertiser
- Scanner

Typically the *Advertiser* role is assumed by a LL *Slave* and GAP *Peripheral*. The *Scanner* role is assumed by a LL *Master* and GAP *Central*.

Advertising is done with advertising packets that are broadcast by *advertisers* through advertising channels(fig.2.2). Advertising packets are transmitted with two purposes:

- Broadcast data to devices that don't need a connection to receive data;
- To seek for devices (*Slaves*) to start a connection.

Table 2.1: Correspondence Between Roles Defined in Different Protocols

	Advertiser	Scanner
LL	Slave	Master
GAP	Peripheral or Broadcaster	Central or Observer

In some applications, devices simply broadcast information in the payload of a BLE packet and others devices, in this case *scanners*, can access data when they cross the same physical channel. An example of such application is a sensing node that broadcast the battery status or a temperature value, simple data that don't necessarily need a connection to transmit that information.

Two types of scanning are defined for an *Observer*: active and passive. A device that proceed to scan passively just receive that the advertising packets (fig. 2.5).

By other hand, if the device is actively scanning, he can send a *Scan Request* to the broadcast, expecting to receive another advertising packet in response. The *Scan Request* sent by the *active scanner* contain no data information (fig. 2.6).

2.1.2.2 Connection

A connection can only be started by a device designated with the role of *Master* (see SubSec.2.1.1.2). The *Master* manifest the intention of starting a connection with a *Slave* device by, after receiving a *Advertising* packet, responding with a connection request packet - *CONNECTION_REQ*.

The *CONNECTION_REQ* packet contains three parameters defined by the *Master* about the connection timings, as when the first packet is sent by the *Master* and when the *Slave* must listen. The parameters are:

- transmitWindowOffset
- transmitWindowSize
- connInterval

The *transmitWindowOffset* defines the interval between the end of the *Advertising Event* and the start of the *Transmit Window*. It is always a multiple of 1.25 μs comprised between 0 and *connInterval*. The *transmitWindowSize* defines *Transmit Window* size which starts 1.25 μs + *transmitWindowOffset* after the *Advertisement Event* ending. In the *Transmit Window*, the *Master* can send the first packet in the *Connection State*. This first packet is called *Anchor Point* and will defines the timings of followings *Connection Events* and the frequency hopping sequence.

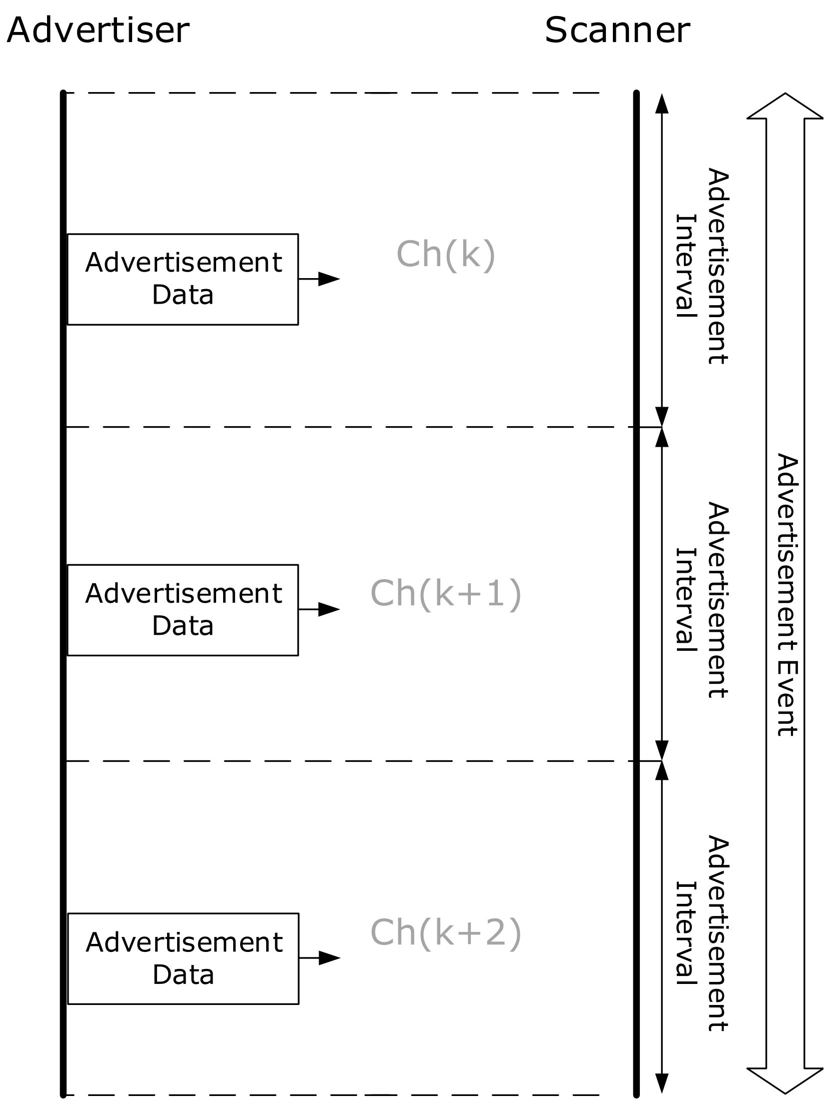


Figure 2.5: Passive Scanning

The parameter *connInterval* will define the *Connection Events* time. The definition of those timings are one of the reason for low-power consumption in BLE since both *Master* and *Slave* can go into deep sleep mode to save energy between transactions [12].

2.1.2.3 Disconnection

The process of *Disconnection* in BLE or, more accurately, the end of a *Connection Event* occur when the *Master* has no more packet to send (and inform the *Slave*) or

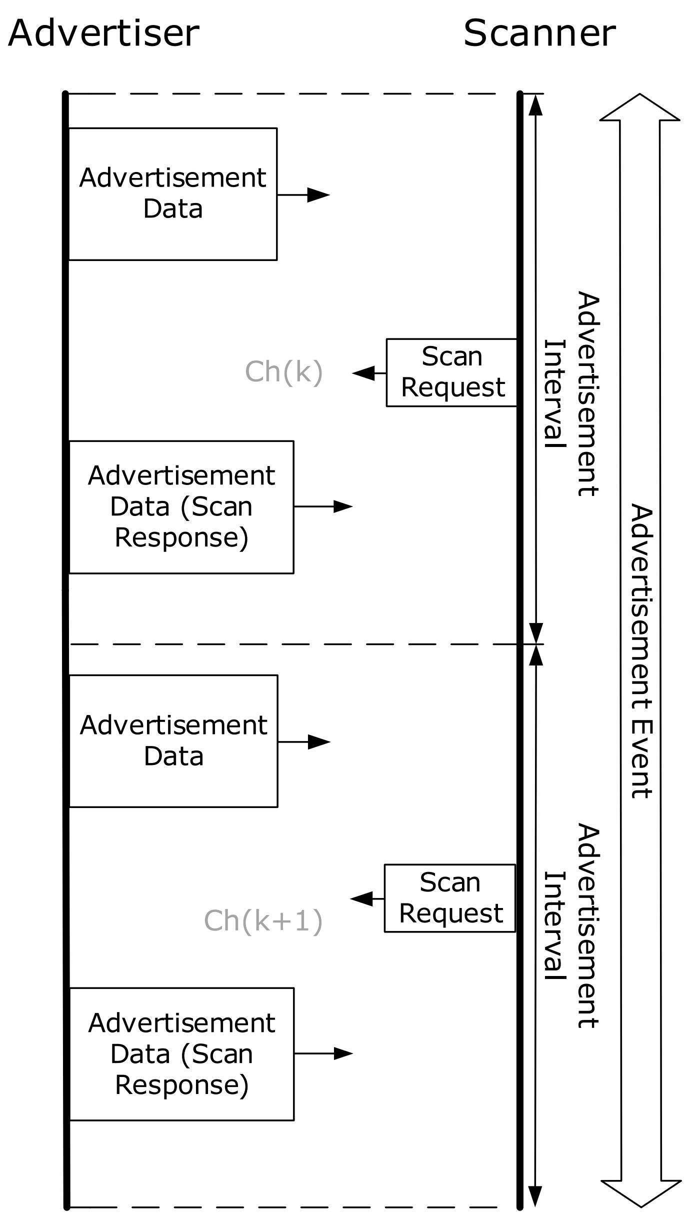


Figure 2.6: Active Scanning

when the reception of a packet fails. If a packet sent by the *Master* is not received (or acknowledged) by the *Slave*, then the *Master* closes the *Connection Event*. If a packet send by the *Slave* is not received (or acknowledged) by the *Master*, then the *Slave* closes the *Connection Event*. When the packets are sent and received by either, the process is handled through the *MD* bit of the *Header* of the *Data Channel PDU*.

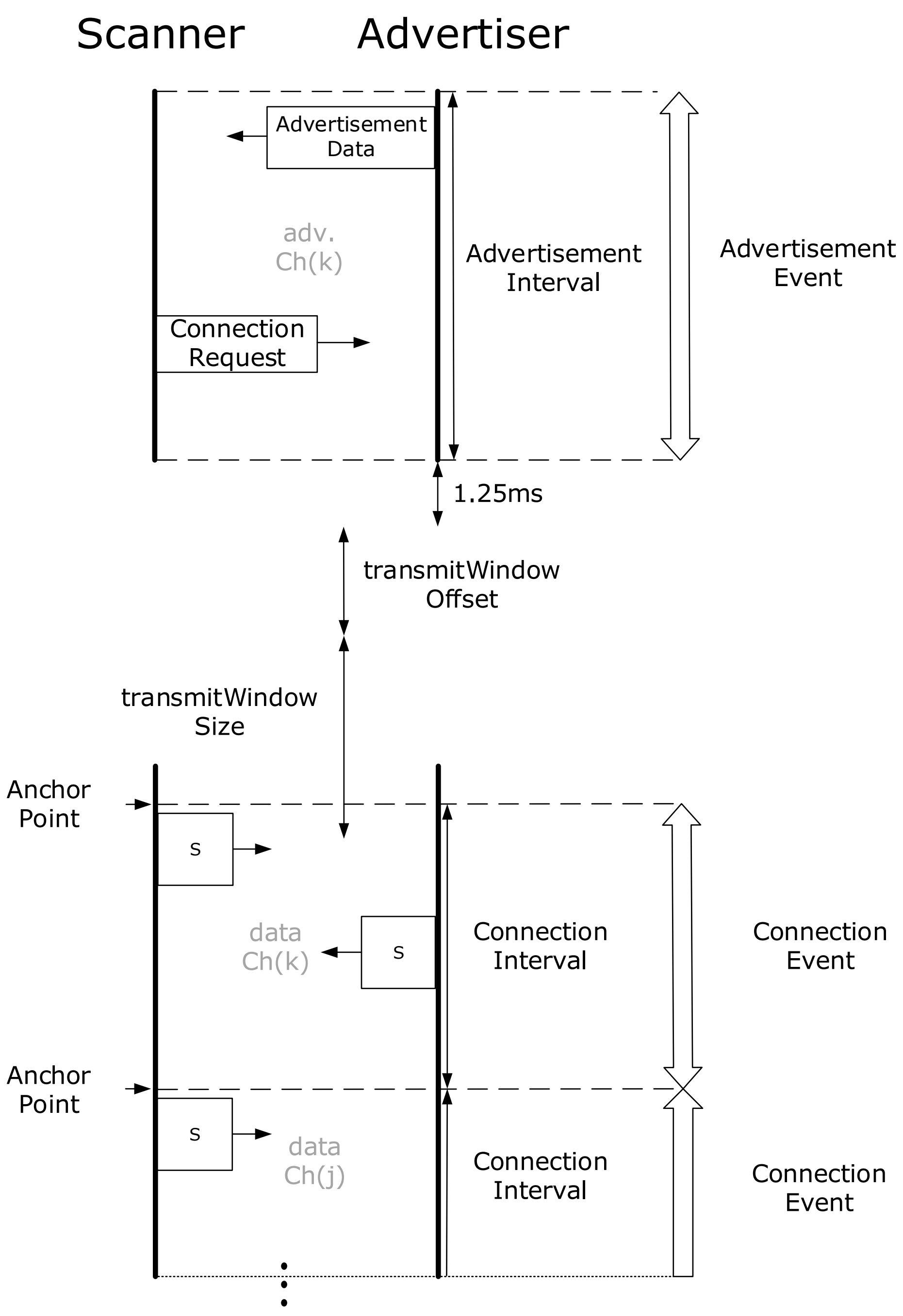


Figure 2.7: Connection Setup Time Line

2.1.2.4 Service Discovery

The *Service Discovery* process occurs between a *Client* and a *Server* (see SubSubSec. 2.1.1.5). After the first connection, the *Client* exchange some packets with the *Server* in order to obtain the *Services* available and how to access them (*Attributes*

Table 2.2: MD Bit Usage in Connection Events

		Master	
		MD bit = 0	MD bit = 1
Slave	MD = 0	Master shall not send another packet, closing the connectionevent. Slave does not need to listen after sending its packet.	Master may continue the connection event. Slave should listen after sending its packet.
	MD = 1	Master may continue the connectionevent. Slave should listen after sending its packet.	Master may continue the connection event. Slave should listen after sending its packet.

and their location). *Services* are accessed through *handles*, which are 16-bit identifier that every *attribute* have, and is what make them addressable since it is guaranteed that they do not change. *Handle* have a range from 0x0001 to 0xFFFF. On the *Server* side, the entity responsible for giving information about *Services* to *Clients* is the Service Discovery Protocol (SDP).

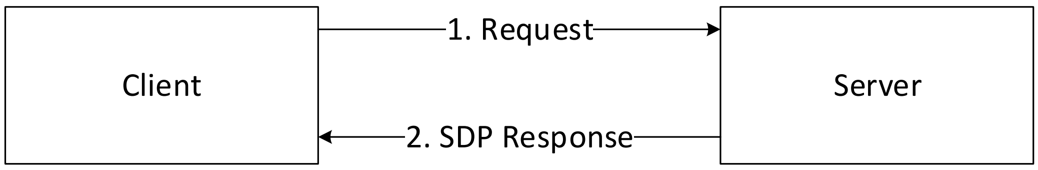


Figure 2.8: Service Discovery Client-Server

The *Client* has two options for discovering *Services*:

Discover all Primary Services The *Client* requests for the complete set of *Services* supported by the *Server*, covering the entire range of handles (0x0001 to 0xFFFF).

Discover a Primary Service by UUID The *Client* knows which *Service* it is seeking and want to know all *Service* instances itself by.

In both cases, the information that SDP returns will be the handle range for the *service characteristics*.

2.1.2.5 Characteristics Discovery, Reading and Writing

The discovery process for *characteristics* is similar to *service* discovery. After knowing the handle range of the *service characteristics*, the *client* has two options:

Discover all Characteristics The *Client* requests for the complete set of *characteristics* by sending the handle range previously received in the *service* discovery process.

Discover a Specific Characteristic by UUID The *Client* knows which *characteristic* wants to access. After that, it is possible to access the *characteristics* descriptors

After receiving the *characteristic* handle, the *Client* uses it to access the *characteristic* content, by sending a request, that will be received in a response packet. To write a *characteristic*, the *Client* will send a packet with an handle and a value for the *characteristic*, which will be acknowledge after by the *server*.

2.2 Internet-of-Things Overview

IoT is a abstract concept, wide in application fields and hard to characterize. Before understanding what is IoT, it matters try to understand the factors that allowed its development. First, the objective of IoT is to gather data from the environment and to make possible the communication of that data between devices, forming networks or by connecting them, resulting in wider networks. The data has been here since ever, sensors and computers are here since many years now and Internet too. So what had change? According the IoT Council [13], Cloud computing is the game changer.

Nowadays society is becoming increasingly more interconnected, due to the technological advances over the years as improvement of broadband connectivity and powerful devices, which leads to a *"always connected"* paradigm [14]. Those technological advances allowed several objects and devices to be included in the Internet, directly or through gateways, leading to the creation of new smart services accessible from anywhere.

Since IoT is still relatively new, there is not yet a standardization for it. The development of new applications produces a big impact not only in a industrial

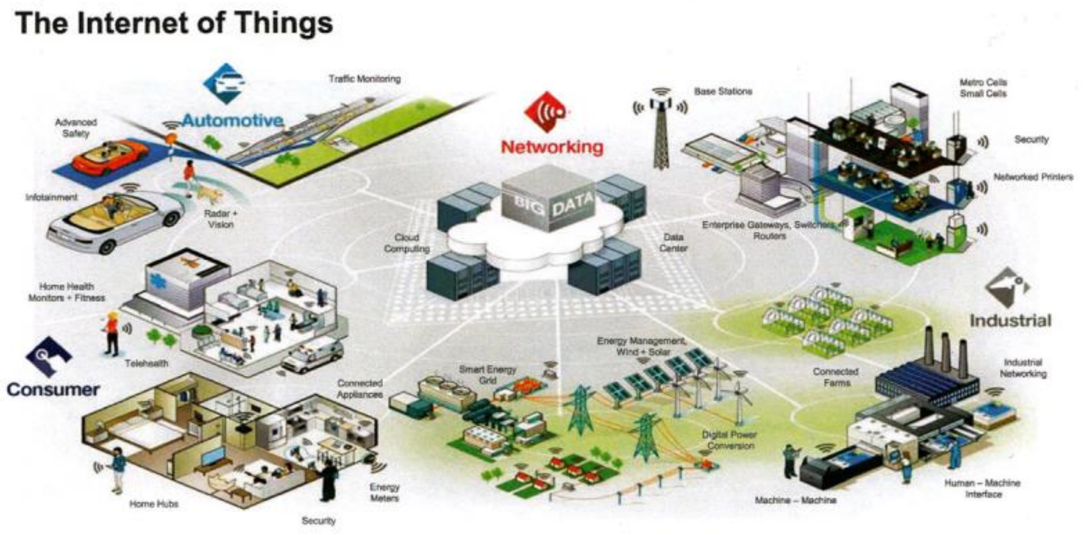


Figure 2.9: The Internet-of-Things

concept, but also socially and economically. Some important issues arises as security and privacy, with a world increasingly more connected, it is crucial to define the boundaries to protect individual privacy [14]. Accordingly to *Gartner, Inc*, in this year it is expected to be 6.4 Billion devices through IoT. The forecast for 2020 is this number to triplicate [15].

2.2.1 IoT Applications Domains

IoT has created the opportunity to fill gaps presents in several domains. The main application domains who benefited the emergence of IoT are:

- Healthcare
- Building Automation
- Smart Cities
- Environment
- Security and Safety
- Industrial Control
- Agriculture
- Logistics

New wearables can incorporate now devices that can monitor health related indicators, replacing the traditional monitoring devices. Cities are investing in ways of reduction illumination costs, using an extended connectivity, to control street lamps and using sensors to improve illumination based on environment variables reducing electricity costs. The extended connectivity is also used in agriculture, where it is possible to monitor a vast area in plantations, introducing the paradigm of smart agriculture which will be reflected in the harvest. In logistics and factories, traditional tracking of objects or components in assembly lines is now being replaced by new tracking devices using Radio-Frequency Identification (RFID). In the retail domain, the data gathered by sensors can be used to study patterns, for example to trace a map of high-traffic zones in the store.

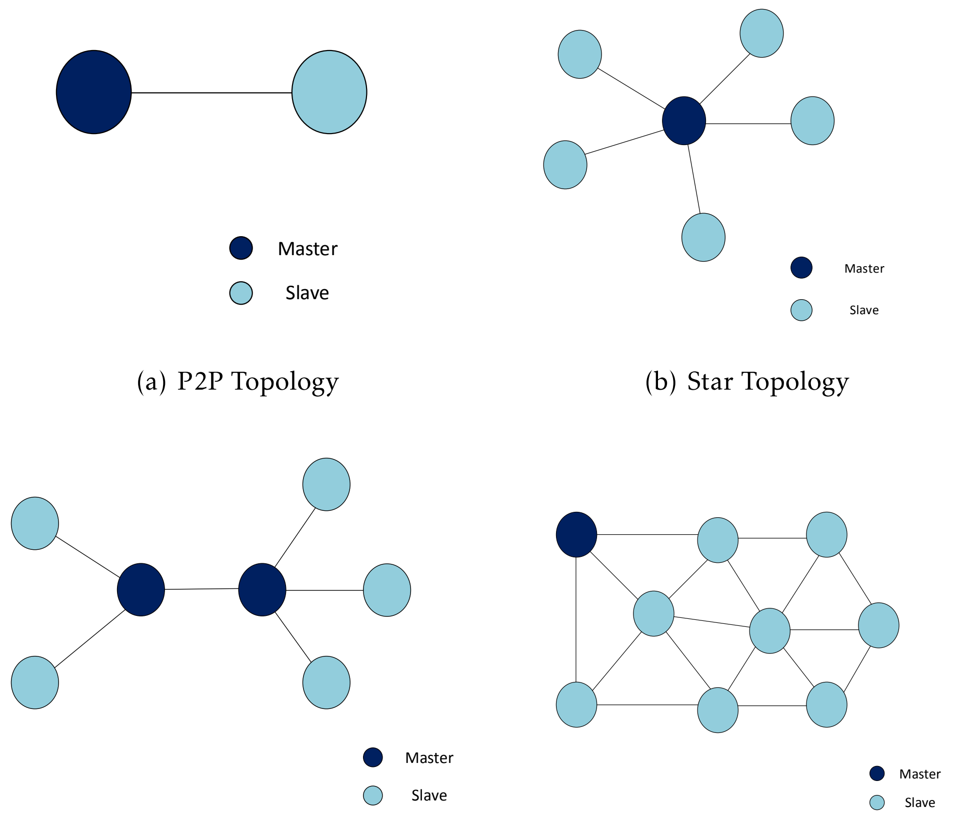
2.2.2 Bluetooth Low Energy vs. Other Communication Protocols

The comparison between communications protocols is not linear and needs to be contextualized with an application and its requirements. There is five factors that help choosing the best protocol for an application:

- Range
- Data Rate
- Power
- Frequency
- Security

Also, depending on the application, it is important to figure what network topology fits better (fig.2.10).

ANT+ *ANT+* derives from the proprietary protocol *ANT*, developed by *Dynastream Innovations Inc.* (subsidiary of *Garmin*). *ANT+* is more indicated for health and performance application like heart rate monitors, speed monitors, pedometers, but can also be used in indoor lighting systems or be integrated in television or cell-phones control systems. The differences between *ANT+* and BLE are in the protocol, whose simpler in ANT+ which means that it is easier to develop applications, *ANT+* also differs in the network topology since it allows any topology unlike BLE that only allow Peer-to-Peer (P2P) and *Star* topologies.



(a) P2P Topology (b) Star Topology

(c) Tree Topology (d) Mesh Topology

Figure 2.10: Wireless Network Topologies

NFC Near Field Communication (NFC) is a communication protocol that allows two devices to transfer data between them when in close proximity, without establishing a connection. It is based on the *RFID* protocol.

RFID RFID is a protocol based on the electrostatic or capacitive coupling resultant from the proximity between two devices. It is mainly used to detect and identify *tags* present in objects or animals. The information is stored in the *tags* that can be passive (power supplied by the RFID readers through radio waves) or active (supplied by a battery). Passive *tags* needs to be nearby a reader to be possible to collect information whereas active *tags* signal can reach some hundreds of meters. This protocol is used in several industries, since factories to track components in assembly lines or in warehouse to locate objects.

WiFi Low Power *WiFi* is widely use to connect several devices like cellphones or computers to Internet, but also can be used in IoT thanks to the adaption

Table 2.3: Comparison Between Communication Protocols

	ANT+	BLE	RFID	WiFi Low Power	ZigBee
Frequency Band	2.4 GHz ISM	2.4 GHz ISM	LF,MF, HF,UHF, SHF	2.4 GHz ISM and 5 GHz	2.4 GHz ISM
Network Topology	P2P, Mesh, Star, Tree	P2P, Star	P2P, Mesh, Star, Tree	N/A	P2P, Mesh
Data Rate	< 1 Mbps	< 1 Mbps	N/A	< 346.66 Mbps	< 250 kbps
Security	64 bits	128 bits	ISO/IEC 18000 ISO/IEC 20248 ISO/IEC 29167	WEP, TKIP, AES, WAPI	128 bits
Range	100m	100m	1m (passive) 200m (active)	1km	100m

WiFi Low Power, also known as *IEEE P802.11ah*.

ZigBee *ZigBee* is a wireless protocol based on the *IEEE 802.15.4* specification. The protocol defines three possible roles: *Coordinator*, *Router* and *End Device*. The first one is the network coordinator, it is responsible for establish it, storing and managing crucial information as security keys or packets destined for *end devices* and is capable of communicating with other networks. The *Router* is a intermediate node, that can carry packet to other devices. Since they can connect to *coordinators* and *end devices*, *routers* are often used as network extenders. *End Devices* have less functionalities than the other mentioned roles, they can only communicate with a parent device. The reduced capabilities results in low power consumption, turning it ideal for sensing nodes.

2.3 Hardware Available for IoT BLE-Based and Non-BLE Development

There is a variety of hardware available in the market for IoT development based on BLE, since development kits that offers a platform for developers to create and test an application to SoC modules for integration in circuit boards. Main hardware for developing and test an application is:

- Manufacturers Development Kits;
- USB Dongles;
- BLE Shields (for Arduino and other platforms);
- BLE SoC modules;
- BLE built-in platforms (example: Bleduino);

Starting with the development kits, they are more suited for developers to start their learning process and get familiarize with functionalities of a module. Another good choice is the BLE shields for other platforms, since if a developer has already experience working on platforms as Arduino or Raspberry Pi, those shields allow to understand the principles of BLE communication in a familiar environment. BLE built-in platforms work the same way. USB dongles are great for testing applications directly from a computer. For prototyping circuit boards, as sensor modules or to connect actuators via BLE, SoC can be integrated easily due to their reduced size.

Several companies have invest in the development of MCU with Bluetooth built-in to allow processing and communication. There is two companies that stood out: Nordic Semiconductors and Texas Instruments, with their nRF51822 and CC2540 SoC modules, respectively.

After the release of those modules, some companies simply used them and created third-party BLE enabled modules [16], adding some functionalities and libraries to facilitate the development of IoT applications.

Table 2.4: Examples of Low-Power Communication Modules

Communication	Name	Type
BLE	RFDuino22301	SoC
	NRF51822	SoC, Development Kit
	Bleduino	Platform
	PSoC 4	SoC, Development Kit
	RedBear Lab BLE Shield	Shield
ZigBee	CC2530	SoC
	XBee	Shield
	EM35x	SoC

Chapter 3

Proposed Smart Interactive System (BlueIoT)

When designing new electronic systems or devices to use in building or home automation, there is several constraints to be taken into account: comfort for users, improved operation, energy efficiency and reduction of costs. The traditional illumination systems are basically composed by a lamp and a manual switch, the light intensity varies according to the lamp (more or less light produced) or in some case it can be adjusted manually by a dimmer switch. Along the year there is four seasons, with more shiny days in summer and darker days at winter. To obtain the maximum comfort for our eyes, the light level need to be adjusted in function of the light present in the space occupied and according to the function performed at the time. Also, the cost of such systems and their installation cannot be too high so they can be economically viable. When analyzing the cost of implementation, it is important to study the system efficiency as well the savings return. An higher energy efficiency means more savings and products more desirable. Some of the strategies studied and tested [17] to design those systems are:

POCS - Predicted Occupancy Control Strategy

The Predicted Occupancy Control Strategy (POCS) uses a schedule where it is defined when the lights are ON or OFF, based on occupancy patterns. This strategy is more oriented for offices with fixed schedule and routines that do not varies too much.

ROCS - Real Occupancy Control Strategy

The Real Occupancy Control Strategy (ROCS) is a real-time strategy, based on the occupancy at the moment. Unlike POCS, ROCS uses sensors to detect presence of someone in the room/office and then turn ON the light. To overcome situations when someone leave and enter the room or when the sensor does not recognize movement, a delay time for turning the lights OFF can be programmed in order to prevent those situations.

CICS - Constant Illuminance Control Strategy

The Constant Illuminance Control Strategy (CICS) works with lighting level within a space, and regulates the lumen output of the lighting system according to the light level measured by the light sensor.

DHCS - Daylight Harvesting Control Strategy

The Daylight Harvesting Control Strategy (DHCS) works with the natural light that enters into a space. It measures the outside light and compare to the light present in a space and adjust the lighting in order to obtain the right balance between natural light from the outside and artificial light from the inside.

Table 3.1: Comparison Between Lighting Design Strategies

Strategy	Main Advantage	Main Disadvantage
POCS	Easy to install and configure	Cannot be applied in spaces with random schedules
ROCS	High energy savings;	High precision sensors needed;
CICS	Constant light level;	Relatively high cost;
DHCS	Possibility of integration with other systems;	Hard to configure;

The next step is to decide the integration level of the system. There is four levels possible:

- Lighting Service
- Lighting Plant
- Lighting Zone
- Lighting Device

In this thesis, the focus is to develop and study a smart interactive system that can operate on a room or an office. This work is based on systems developed with existing commercial parts [18, 19], and will include sensors for temperature, light and presence, a BLE embedded MCU to process, receive and send data remotely and a dimming Light Emitting Diode (LED) driver to control illumination.

First, a mobile application that allow users to check ambient variables as light and temperature, and to choose an actuation on the illumination. Second, the BLE module will assume all the processing of information retrieved from the sensors, and establish a gateway to the mobile application present in a smartphone or tablet using BLE. Finally, a circuit for dimming a lamp, that will receive instructions from the main MCU.

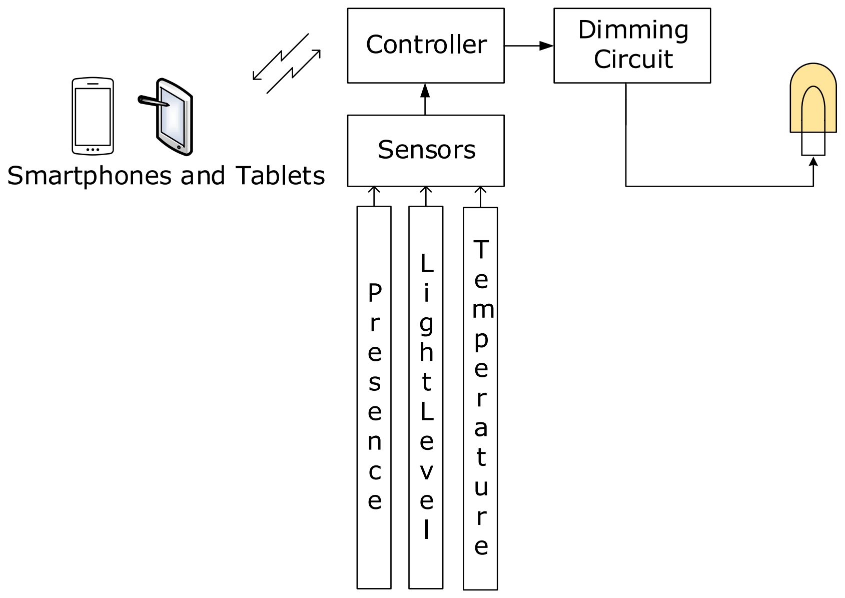


Figure 3.1: Generic Smart Lighting System

3.1 Mobile Application

Nowadays everybody use *smartphones* and tablets. *App's* are widely used and in this case it is the most suitable way to control a *smart* system. To ensure that the system is adequate for a market application, simplicity and robustness are the keys points to follow.

When Bluetooth 4.0 has arises, Android released the 4.3 or *JellyBean(MR2)* so the functionalities of BLE could be used with the operative system. To do so, the

API's in package *android.bluetooth* were upgraded to embed functionalities that allows user's devices to communicate with BLE peripherals. Since then, in every Android version upgrade, the BLE package has been improved in order to provide a stable development platform for developers to design better applications after several issues stated in previous versions [20].

To figure the potential market for BLE based IoT applications in Android, combining the number of devices running Android as OS, estimated to 1.4 billion [21], with the chart in fig.3.2 and the table 3.2, it is possible to estimate the number of devices suitable to run BLE based applications, which is around 1.06 billion devices.

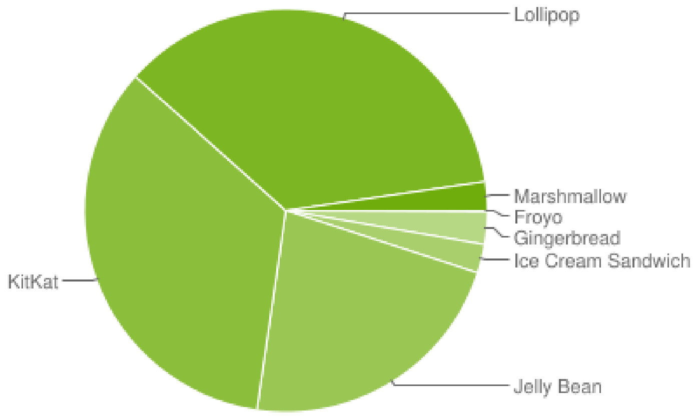


Figure 3.2: Chart with Android Distributions Percentages

Table 3.2: Distribution of Android by Versions

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.3%
4.1.x		16	8.1%
4.2.x	Jelly Bean	17	11.0%
4.3		18	3.2%
4.4	KitKat	19	34.3%
5.0	Lollipop	21	16.9%
5.1		22	19.2%
6.0	Marshmallow	23	2.3%

As previously explained in Chapter 2, communication in BLE follows a defined structure, so the Android OS will need to follow the same structure to be able to communicate with BLE devices. The next subsections will overlap the concepts approached in the previous chapter with the BLE concepts in Android OS.

3.1.1 GATT Elements

GATT is the protocol that defines the procedures and formats for data exchange in Bluetooth LE. As seen in figure 2.4, the data hierarchy of a GATT server has several levels, which are defined in package *bluetooth* of Android through the following elements:

BluetoothGatt Responsible for implementing Bluetooth LE profiles.

BluetoothGattCallback Implements callback[1] methods for GATT events like discover services, changes in connection and reading or writing characteristics. The callback methods used from this class are:

- void onCharacteristicChanged(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic);
- void onCharacteristicRead(BluetoothGatt gatt, BluetoothGattCharacteristic characteristic, int status);
- void onConnectionStateChanged(BluetoothGatt gatt, int status, int newState);
- void onServicesDiscovered(BluetoothGatt gatt, int status).

BluetoothGattCharacteristic Represents a GATT *characteristic*. The methods used from this class are:

- BluetoothGattDescriptor getDescriptor(UUID uuid);
- UUII getUuid();
- byte[] getValue();
- boolean setValue(byte[] value);
- void setWriteType(int writeType).

From this class it is also used a constant:

- WRITE_TYPE_NO_RESPONSE.

[1]A callback is a subroutine that is automatically invoked after a specific event occurs.

BluetoothGattDescriptor Represents a GATT descriptor. The methods used from this class are:

- boolean setValue(byte[] value);

From this class it is also used a constant:

- ENABLE_NOTIFICATION_VALUE.

BluetoothGattService Represents a GATT service. The methods used from this class are:

- BluetoothGattCharacteristic getCharacteristic(UUID uuid);

3.1.2 Elements Initialization

In contrast to the simplicity of connecting devices to BLE modules and to exchange data between them, developing a mobile application capable of realizing the tasks required and be robust to flaws requires some complexity. To overcome the complexity it is essential to divide the code.

The division consists in the main activity that will be responsible for all the initializations, user interface and the main algorithm. Next there is the support files, responsible for all the auxiliary functions that the main activity will need like the creation and definition of the service, data processing functions and all the procedures defined in the Bluetooth 4.0 protocol. This separation makes the developing and debugging process easier and clear.

The first step is to import the packages that will be used to develop the mobile application. A special highlight go to the Bluetooth package, since the main feature of the system is the Bluetooth communication. For the user interface buttons, text boxes and all the variables related with the layout will be declared. To control the mobile application state, a state machine will be created with variables of *Int* type for the several states that the *app* will face, like the Bluetooth Off, Disconnected, Connecting and Connected. When the *app* starts, the code section responsible to handle the creation and initialization of the resources needed is the function *onCreate(Bundle savedInstanceState)*. In this function the *BluetoothAdapter* will be initialized. The *BluetoothAdapter* class allows us to perform the fundamental Bluetooth tasks. Then the layout elements are assigned to theirs respective functions and procedures so whenever a button is clicked, the respective actions are performed. In figure 3.3 is shown the life cycle of the mobile application.

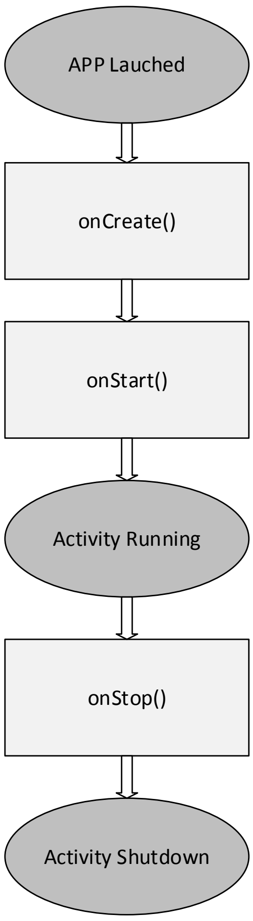


Figure 3.3: Life Cycle of the App

After the set up of the application elements, the method *onStart*() will start, enabling the activity to be visible to the user. This method will handle the receivers register:

- scanModeReceiver
- bluetoothStateReceiver
- rfduinoReceiver

The register is done with the method *registerReceiver(BroadcastReceiver receiver, IntentFilter filter)*, where the parameter *receiver* will be a receiver to be registered and the Intents intended for this will need to match the *IntentFilter* in the parameter *filter*. The *scanModeReceiver* is created to check the scan mode of the *BluetoothAdapter*, using the *getScanMode()* method that can return three possible modes:

SCAN_MODE_NONE
  Indicates that the device is neither discoverable nor connectable.

SCAN_MODE_CONNECTABLE
  The device is not discoverable, only connectable, so it can be connected only to another device that had previously discover this device.

SCAN_MODE_CONNECTABLE_DISCOVERABLE
  The device is discoverable and connectable.

The second receiver to be created is the *bluetoothStateReceiver* that will listen to changes in the *BluetoothAdapter* status. In the BluetoothAdapter it is defined a constant denominated *ACTION_STATE_CHANGED*, that as the name states, reveal a change on the adapter state. This constant contains two fields: *EXTRA_STATE* and *EXTRA_PREVIOUS_STATE*, whose represent, respectively, adapter's new and old state. In this case, only the actual state will be needed, and will be retrieved by the method $getIntExtra(BluetoothAdapter.EXTRA\_STATE, 0)$. There are four possibilities for the data returned:

SCAN_OFF
  The local adapter is off.

SCAN_TURNING_ON
  The local adapter is turning on but still not ready to be connected.

SCAN_ON
  The local adapter is on and ready to use.

SCAN_TURNING_OFF
  The local adapter is turning off. This state can be used by local clients to prepare for disconnection.

3.1.2.1 Bluetooth Adapter Initialization

In this step, the *BluetoothAdapter* will be initialized so the application can be able to discover BLE devices and connect to them. The *BluetoothAdapter* will be obtained through the *BluetoothManager*, so the first objective is to assure that it is not *null*. The method *getSystemService(Context.BLUETOOTH_SERVICE)* will retrieve a *BluetoothManager*. After that, the method *getAdapter()* from the object *BluetoothManager* will return a *BluetoothAdapter*.

```
public boolean initialize() {

        if (mBluetoothManager == null) {
            mBluetoothManager = (BluetoothManager) getSystemService(
                                              Context.BLUETOOTH_SERVICE);
            if (mBluetoothManager == null) {
                Log.e(TAG, "Unable to initialize BluetoothManager.");
                return false;
            }
        }

        mBluetoothAdapter = mBluetoothManager.getAdapter();
        if (mBluetoothAdapter == null) {
            Log.e(TAG, "Unable to obtain a BluetoothAdapter.");
            return false;
        }

        return true;
    }
```

3.1.3 Creation of the Service

As seen in 2.1.1.5, data moves in BLE through *Services*. So in the *app*, *Service* and all the objects related need to be created and defined. All the *Service* related objects and methods will be declared in a different file, the *RFDuinoService.java*. First, the concept of *Service* is defined through the Android classes *app.Service* and *bluetooth.BluetoothGattService*. The first one corresponds to the Android concept of *Service*, and it is related to functionalities the mobile application will use and if necessary to share resources between applications. The second one is related to the GATT definition of service as a level in the data hierarchy, seen in fig.2.4.

The *Service* definition is handled in the *RFDuinoService.java* file, resulting in the *RFDuinoService* class creation, as depicted by this code excerpt:

```
public class RFDuinoService extends Service {
```

In this class the main object that will be set are:

BluetoothGattCallback To implement *BluetoothGatt* callback methods related to the application.

Binder and LocalBinder Derive from *app.Service* object definition.

IntentFilter Will define the *Intents* to which the service will respond.

Next, the methods associated with the class to handle:

- Initialization;
- Connection and Disconnection;
- Sending Data.

3.1.3.1 Initialization

In the *initialization* will be initialized the entity *mGattCallback*, a *BluetoothGattCallback* that as referred before, will define callbacks to handle the connection, service discovery and reception of data. As explained further in Sec.3.2, according to GATT data hierarchy, a *service* and respective *characteristics* are already defined and are the following ones:

- UUID_SERVICE, with the UUID 0x2220;
- UUID_RECEIVE, with the UUID 0x2221;
- UUID_SEND, with the UUID 0x2222;
- UUID_DISCONNECT, with the UUID 0x2223;
- UUID_CLIENT_CONFIGURATION, with the UUID 0x2902;

In the callback *onConnectionChanged*, each time there is a change in the application state, the information is broadcast to the main activity. Then in *onServicesDiscovered*, the *characteristics* are verified so the application can transfer data. The reception of data is handled here too, in the callbacks *onCharacteristicsRead* and *onCharacteristicsChanged*. Whenever there is a change in the characteristic *UUID_RECEIVE*, which means data is received, the information is broadcast to the main activity, that will handle the data after.

```
public final static UUID UUID_SERVICE =
    BleUtils.sixteenBitUuid(0x2220);
public final static UUID UUID_RECEIVE =
    BleUtils.sixteenBitUuid(0x2221);
public final static UUID UUID_SEND =
    BleUtils.sixteenBitUuid(0x2222);
public final static UUID UUID_DISCONNECT =
    BleUtils.sixteenBitUuid(0x2223);
public final static UUID UUID_CLIENT_CONFIGURATION =
    BleUtils.sixteenBitUuid(0x2902);
//org.bluetooth.descriptor.gatt.client_characteristic_configuration
```

```

private final BluetoothGattCallback mGattCallback =
      new BluetoothGattCallback() {
    @Override
    public void onConnectionStateChange(BluetoothGatt gatt,
        int status, int newState) {

        switch (newState){
            case BluetoothProfile.STATE_CONNECTED:
                Log.i(TAG, "Connected to RFDuino");
                Log.i(TAG, "Attempting Service Discovery:"
                    + mBluetoothGatt.discoverServices());
                connectionState = STATE_CONNECTED;


            case BluetoothProfile.STATE_DISCONNECTED:
                Log.i(TAG, "Disconnected from RFDuino.");
                broadcastUpdate(ACTION_DISCONNECTED);

            case BluetoothProfile.STATE_CONNECTING:
                Log.d(TAG, "state: Connecting to GATT server.");

            case BluetoothProfile.STATE_DISCONNECTING:
                Log.d(TAG, "state: Disconnecting from GATT server.");

            default:
                Log.d(TAG, "Unknown Status:" + status);
        }

    }

    @Override
    public void onServicesDiscovered(BluetoothGatt gatt, int status) {
        if(status == BluetoothGatt.GATT_SUCCESS){
            mBluetoothGattService = gatt.getService(UUID_SERVICE);
            if (mBluetoothGattService == null){
                Log.e(TAG, "RFDuino GATT Service not found.");
                return;
            }

            BluetoothGattCharacteristic receiveCharacteristic =
                    mBluetoothGattService
                      .getCharacteristic(UUID_RECEIVE);
            if(receiveCharacteristic != null){
                BluetoothGattDescriptor receiveConfigDescriptor =
                        receiveCharacteristic.
                            getDescriptor(UUID_CLIENT_CONFIGURATION);
                if(receiveConfigDescriptor != null){
                    gatt.setCharacteristicNotification(
                      receiveCharacteristic,true);

                    receiveConfigDescriptor.setValue(
                            BluetoothGattDescriptor
                              .ENABLE_NOTIFICATION_VALUE);
                    gatt.writeDescriptor(receiveConfigDescriptor);
                } else {
                    Log.e(TAG, "RFDuino receive config descriptor
                       not found.");
                }
            } else {
                Log.e(TAG, "RFDuino receive characteristic
                  not found.");
            }
            broadcastUpdate(ACTION_CONNECTED);
        } else {
            Log.w(TAG, "onServicesDiscovered received:" + status);
        }
    }

    @Override
    public void onCharacteristicRead(BluetoothGatt gatt,
        BluetoothGattCharacteristic characteristic, int status) {
        if(status == BluetoothGatt.GATT_SUCCESS){
            broadcastUpdate(ACTION_DATA_AVAILABLE,characteristic);
        }

    }

    @Override
    public void onCharacteristicChanged(BluetoothGatt gatt,
        BluetoothGattCharacteristic characteristic) {
        broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
    }
};
```

3.1.3.2 Connection and Disconnection

The entity *BluetoothGatt* has a method for connection, *.connect()* and other for disconnection,*.disconnect()*. Before calling the *.connect()* method, it is necessary to verify if a connection can be established, through the device address validation. Then if the *device* entity is valid, it will connected to a *GATT* entity. As in the connection process, for the disconnection it is verified if the *BluetoothAdapter* is valid, and then it is proceed the disconnection, closing the *BluetoothGatt* after.

```
public boolean connect(final String address) {
    if (mBluetoothAdapter == null || address == null) {
        Log.w(TAG, "BluetoothAdapter not initialized or unspecified
                                                          address.");
        return false;
    }

    if (mBluetoothDeviceAddress != null
            && address.equals(mBluetoothDeviceAddress)
            && mBluetoothGatt != null) {
        Log.d(TAG, "Trying to use an existing mBluetoothGatt
                                                   for connection.");
        return mBluetoothGatt.connect();
    }

    final BluetoothDevice device =
                          mBluetoothAdapter.getRemoteDevice(address);
    mBluetoothGatt = device.connectGatt(this, false, mGattCallback);
    Log.d(TAG, "Trying to create a new connection.");
    mBluetoothDeviceAddress = address;
    return true;
}

public void disconnect() {
    if (mBluetoothAdapter == null || mBluetoothGatt == null) {
        Log.w(TAG, "BluetoothAdapter not initialized");
        return;
    }
    mBluetoothGatt.disconnect();
}

public void close() {
    if (mBluetoothGatt == null) {
        return;
    }
    mBluetoothGatt.close();
    mBluetoothGatt = null;
}
```

3.1.3.3 Sending Data

To send data, the process is similar to the data reception, it is necessary to write on the *characteristic UUID_SEND*, which is the one define on the *service* to send data. The verifications made before writing on the *characteristic* are for the *BluetoothAdapter* and then for the *characteristic* itself to see if the application has found it and is able to write it.

```
public boolean send(byte[] data){
        if(mBluetoothGatt == null || mBluetoothGattService == null){
            Log.w(TAG, "BluetoothGatt not initialized");
            return false;
        }

        BluetoothGattCharacteristic characteristic =
                mBluetoothGattService.getCharacteristic(UUID_SEND);

        if(characteristic ==  null){
            Log.w(TAG, "Send Characteristic not found");
            return false;
        }

        characteristic.setValue(data);
        characteristic.setWriteType(BluetoothGattCharacteristic
            .WRITE_TYPE_NO_RESPONSE);
        return mBluetoothGatt.writeCharacteristic(characteristic);
    }
```

3.1.4 Auxiliary Files

The mobile application will deal with data transfer between a mobile device and a BLE module, thus data conversion is mandatory. The mobile application will present data in a *ASCII* format while the data is received in bytes. In the opposite path, the same problem arise. To overcome this issue, methods to convert bytes to ASCII and ASCII to bytes need to be created. The class where methods responsible to convert data formats are defined is *HexAsciiHelper*.

Another issue that needs attention is the BLE addressing through UUID. As seen in SubSec.2.1.1.5 there is shorts and longs UUID, so to be easier handling the addressing to *attributes* another *class*, *BleUtils*, will define methods to do so.

3.1.4.1 Data Conversion

The data conversion performed by the application is due to difference between format used in transfer, which are bytes, and the format used to display the data, which is characters. So a conversion from bytes to ASCII and vice-versa has to be done each time there is a data transfer. The conversion from bytes to ASCII is performed by the function *bytesToAsciiMaybe*, where a *string* is build through *StringBuilder* element, and then characters are added after verified if they are ASCII.

```
public static String bytesToAsciiMaybe(byte[] data,
     int offset, int length) {

        StringBuilder ascii = new StringBuilder();
        boolean zeros = false;
        for(int i = offset;i<offset + length;i++){
            int c = data[i] & 0xFF;
            if(isPrintableAscii(c)){
                if(zeros) {
                    return null;
                }
                ascii.append((char) c);
            } else if (c == 0){
                zeros = true;
            } else {
                return null;
            }
        }
        return ascii.toString();
    }
```

3.1.5 Sequence Diagrams

When using the mobile application, there is a list of actions that can be performed:

Enable Bluetooth
This function allow the user to enable Bluetooth on his mobile phone or tablet. By clicking the button, the *app* will turn on Bluetooth and it is now possible connecting to other Bluetooth devices or, in this case, to the system BLE module.

Scan Devices
After having Bluetooth enabled on the device, it is possible to perform this

action. When clicking the *Scan Devices* button, the *app* will start to seek for others Bluetooth devices. After a few seconds, a list of the Bluetooth devices in range with the user's device information will be showed and the user can now try a connection with one of them.

Connect

This option is only available after the *scanning* process. After showing the list of devices available, it is possible to connect to one of them by clicking the button *Connect*. After that, and for the Lighting Controller case, it becomes possible to interact with the module.

Refresh

After the connection is established, the exchange of data between the BLE module and the application will start. The module will send the data retrieved from the sensors and send it to the *app* repeatedly in defined time intervals. If the user want to manually request the sensors's value at that moment, clicking the *Refresh* button will send an instruction for the module to send the sensors's values out of the defined time intervals.

Set Automatic/Manual Control

Just as in *Refresh*, the option *Set Automatic Control* is only available after a connection established. To set the automatic control mode for the smart system, application has a switch button to select between automatic and manual mode control.

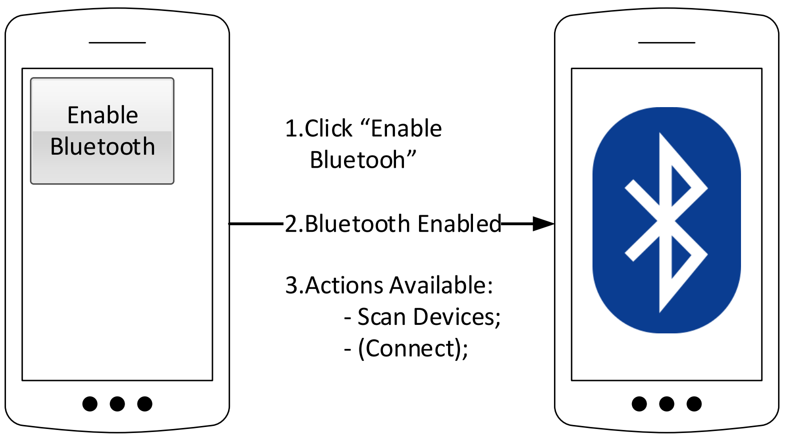


Figure 3.4: Enabling Bluetooth

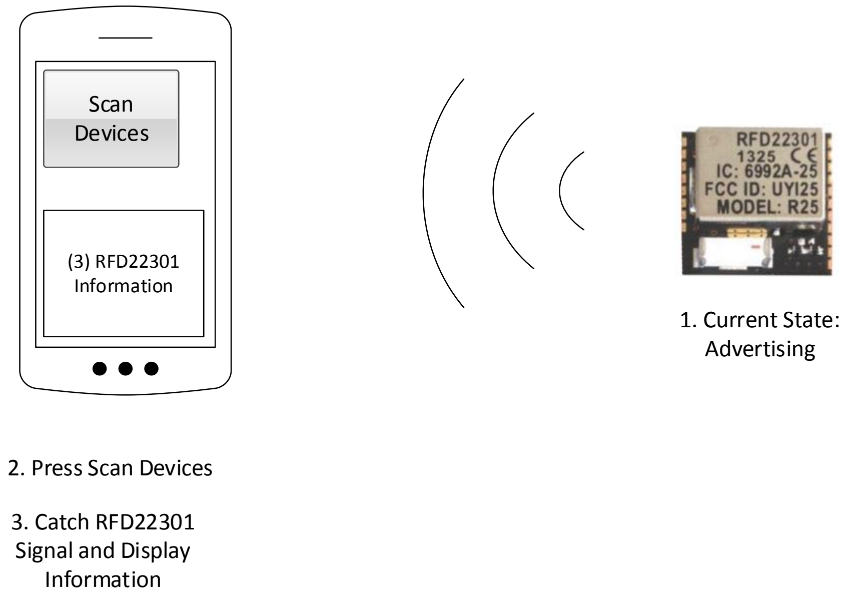

Scan Devices
(3) RFD22301 Information
RFD22301 1325 IC: 6992A-25 FCC ID: UYI25 MODEL: R25
1. Current State: Advertising
2. Press Scan Devices
3. Catch RFD22301 Signal and Display Information

Figure 3.5: Scanning Devices

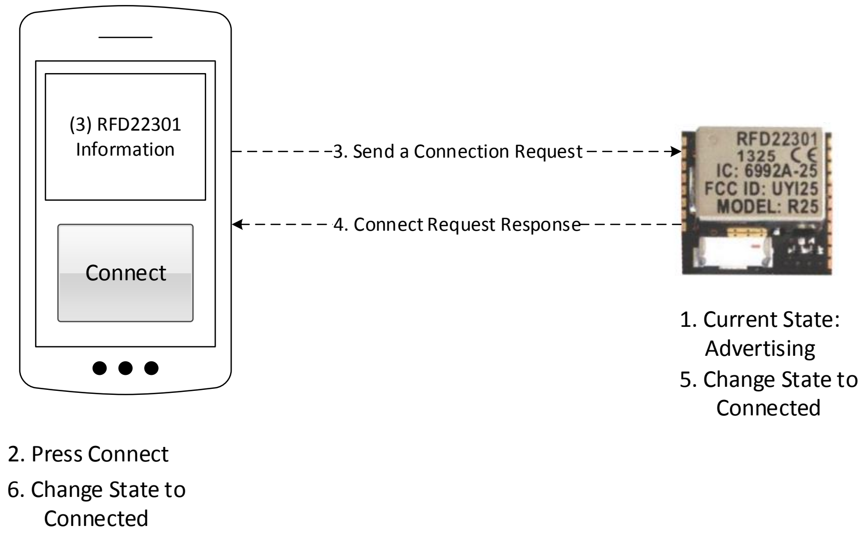

(3) RFD22301 Information
Connect
3. Send a Connection Request
4. Connect Request Response
RFD22301 1325 IC: 6992A-25 FCC ID: UYI25 MODEL: R25
1. Current State: Advertising
5. Change State to Connected
2. Press Connect
6. Change State to Connected

Figure 3.6: Connection Process

Set Dimmer Value If the manual control mode is selected, the seek bar with the values possible for dimming becomes avalaible. It allows values between zero and 128, which after every change on the bar, the application sends a packet with the new dimming value selected.

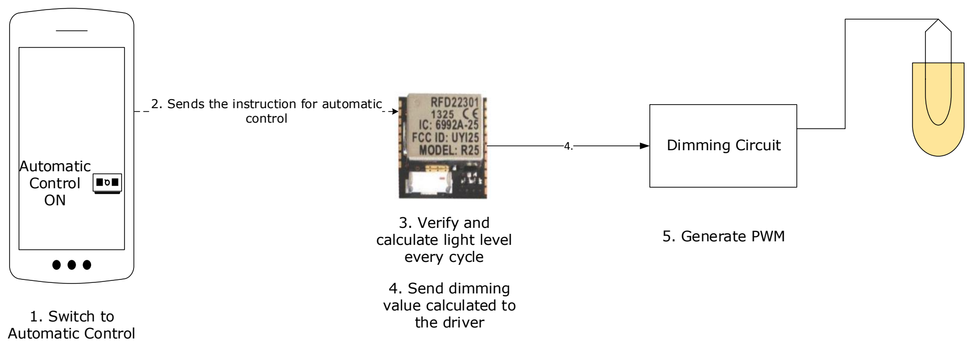

Automatic
Control
ON
1. Switch to Automatic Control
2. Sends the instruction for automatic control
RFD22301
1325
IC: 6992A-25
FCC ID: UYI25
MODEL: R25
3. Verify and calculate light level every cycle
4. Send dimming value calculated to the driver
4.
Dimming Circuit
5. Generate PWM

Figure 3.7: Automatic Control Setup

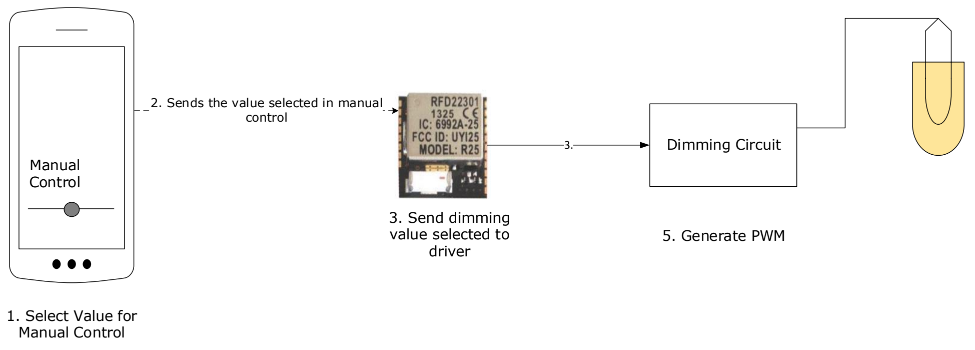

Manual
Control
1. Select Value for Manual Control
2. Sends the value selected in manual control
RFD22301
1325
IC: 6992A-25
FCC ID: UYI25
MODEL: R25
3. Send dimming value selected to driver
3.
Dimming Circuit
5. Generate PWM

Figure 3.8: Manual Control Setup

3.2 BlueIoT Embedded

Since the objective is to develop an application for IoT, the module chosen has to process data, control actuators and be able to communicate with other devices through BLE.

3.2.1 BLE Module

The module chosen for the system is the RFD22301(fig.3.10) produced by RFDigital®. This module allows the development of low-cost and low-power applications since it is based on the Nordic Semiconductors module, the nRF51822. The RFD22301 module is powered by a 32-bit ARM®Cortex®M0 processor, which is

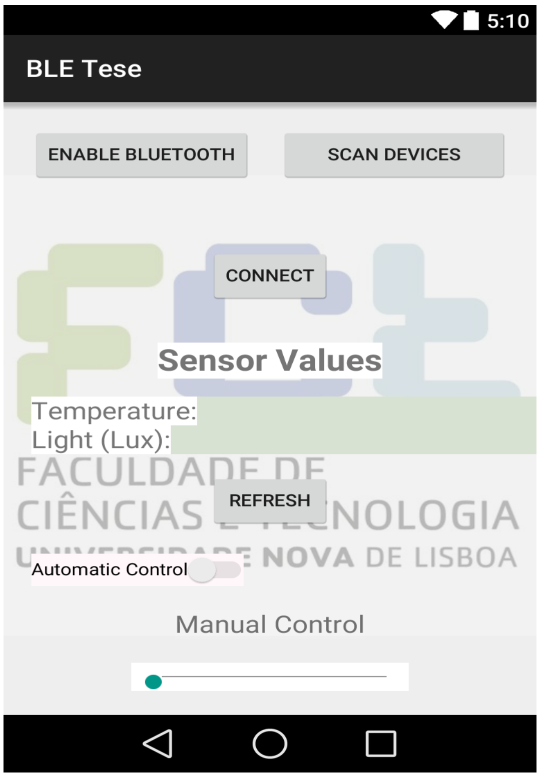


Figure 3.9: Mobile Application Layout

the processor with better results relative to power consumption in comparison with others processors used in BLE modules. The embedded flash memory has 128 Kbytes of capacity and the RAM is 8 Kbytes. The General-Purpose input/output pins present are seven, with an analog-to-digital converter, and allows the communicate between the module and others devices using Inter-Integrated Circuit (I2C), an asynchronous communication protocol that allows multiple *master* devices to communicate with one or more *slave* devices, Serial Peripheral Interface (SPI), a synchronous serial communication protocol, and Universal Asynchronous Receiver/Transmitter (UART), an asynchronous serial communication protocol. The electrical specifications of the module, provided by the manufacturer [22], are present in the table 3.3.

3.2.2 Module Programming

An application (Sec.2.1.1 and fig.2.1) will run on the module MCU. The objective is to implement the procedures to advertise, settle connections and perform actions for the smart interaction system. The module programming is done in

Table 3.3: Electrical Specifications of RFD22301 Module

Description	Min	Nom	Max
VDD - Supply Voltage	2.1V	3.0V	3.6V
Radio Output Power	-30 dBm	N/D	+4 dBm
Receiver Sensivity	N/D	-93 dBm	N/D
ADC Internal Reference Voltage	1.182V	1.200V	1.218V
ADC External Reference Voltage	0.83V	1.20V	1.30V
Tx Current	N/D	12 mA	N/D
Rx Current	N/D	12 mA	N/D

the Arduino IDE environment, recommended by the manufacturer [23]. Since the company decide to not release all the libraries open-source, it is not possible to develop a *Service*, as seen in SubSec. 2.1.1.5, because the libraries implement *Services* already for receiving and sending data. Using the *Service* available and this module specifically, it eases the development of applications, reduces their complexity and benefits from the FCC and CE certification of the module [24]. The inherent cost of using their *Service* will be reflected in coding efficiency since it is not possible to associate directly sensor values, buttons information or data from a mobile application to a specific *Characteristic*(SubSec.2.1.1.5).

The manufacturer provides a library (*RFDuino.h*) that eases the development of applications, with procedures for advertising, connecting, send data and call-backs for each module states. The states possible are *OFF/Not Ad* or *Sleeping*, *Advertising* and *Connected*. As explained in Sec.2.1.2, the application will have a life-cycle, shown in fig.3.11, starting in the *OFF/Not Advertising* state, then when movement is detected by the Passive Infrared Sensor (PIR) sensor it starts advertising and goes to *Advertising* state, then if it receives a connection request and establish a connection, it goes to *Connected* state. If the module is advertising for two minutes without a connection established, it stops advertising. In the *Connected* state, the application can send and receive data. In the end, when the

Figure 3.10: The RFD2301 Module

application ceases the connection, it goes to the *Disconnected* state. In this state, the application can return to advertising or turn off the BLE stack. The start and the end of the application cycle are defined by two functions: *begin()* and *end()*. When the procedure *RFduinoBLE.begin()* is called, the BLE stack will start and then it start advertising. On the opposite way, to end the BLE stack and therefor to stop advertising, the procedure *RFduinoBLE.end()* is called. To allow the developer to define the actions performed by the application in each state, the library implements five callbacks, three for the application states, one to receive data and one(not used in this application) that returns the signal strength after a connection. The actions performed in each state are:

onAdvertisement

When the application enter this state, it sets the variable **padvertisement*, that indicates if the application is advertising or not, to TRUE.

onConnect

In this state, the application sets the variable **pconnection* to TRUE, that indicates the connection has been settled. This variable will be used as verification in all procedures that need to send data via BLE, to guarantee that the module does not try to send data without a connection established so the application does not crash.

onReceive

In this state, the callback *onReceive* has two input parameters, **data* and *len*, that corresponds to an array and its length and it is where the data received can be accessed. It is here that the instructions received from the mobile application. There is three instructions possible: activate the automatic control, set the dimming value chosen by the user and a request for sending the sensors values. For automatic control, the instruction sent will be an *A*. For the manual control, it will be a *MXXX*, with *XXX* being a value comprised between 0-128 which is the dimming levels allowed. Accordingly to the data received, the procedure *actuatorsAction(char c,int v)* with *c* being *A* or *M* and *v* being the dimming value(for automatic control, the value in *v* is set to 0). In the case that a sensor values resend has been requested, the procedures *getLightSensorValue()* and *getTemperatureSensorValue()* are called.

onDisconnect

In this state, the application sets the variable **pconnection* to FALSE, that indicates the connection has ended.

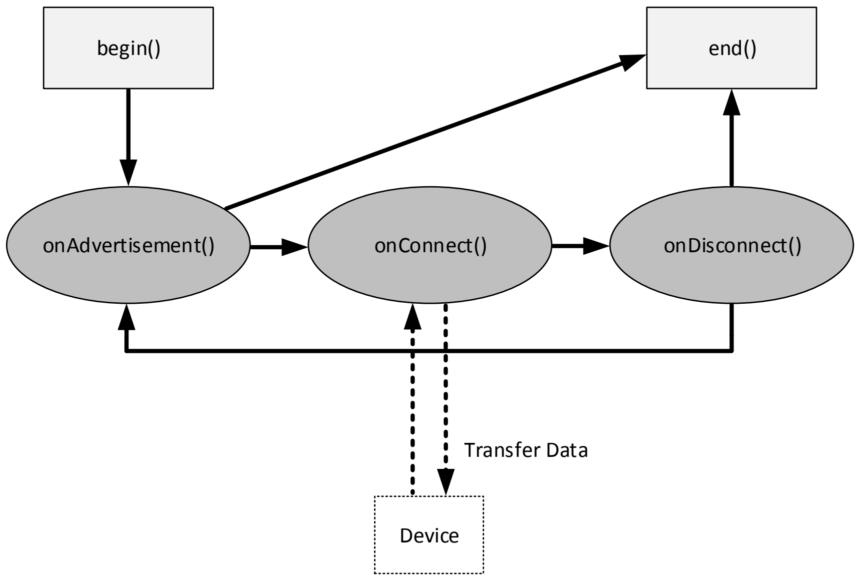


Figure 3.11: States of the BLE Application

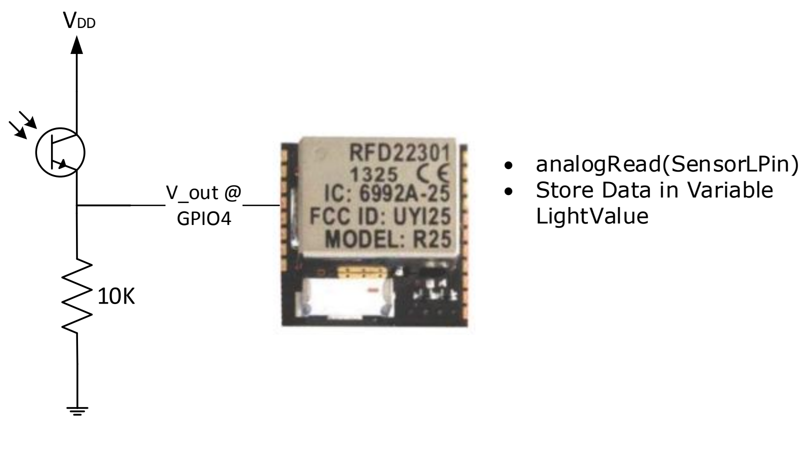
The objective of this system is the integration with sensors and actuators, so to handle processing of the data retrieved in sensors, the control of actuators and the commands received by the mobile application, some procedures have to be implemented:

getLightSensorValue()

The procedure *getLightSensorValue()* has the function of retrieving values from the light sensor connected to the module and send them in BLE packets to a connected device. The module will retrieve the data using the method defined in the RFDuino library for analog-to-digital readings *analogRead(SensorLPin)*, where *SensorLPin* is the pin defined for the light sensor. This function will return a integer with the light value, accordingly with the light intensity measured(fig.3.17) that will be stored in the integer variable *LightValue*. The data flow of this action is described in fig.3.12(a).

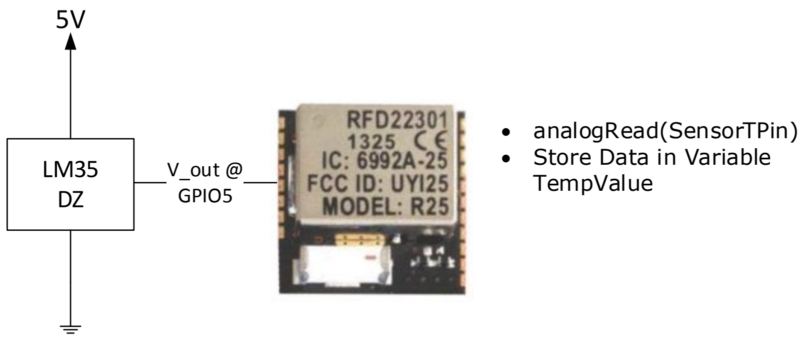
getTemperatureSensorValue()

The procedure *getTemperatureSensorValue()* follow the same pattern as the *getLightSensorValue()* procedure. It will retrieve the temperature value from an *analogRead(SensorTPin)*, this time on the pin defined for the temperature sensor. The value retrieved from the sensor will come from the RFDuino ADC, that has a 10-bit resolution for reading values. So to convert from



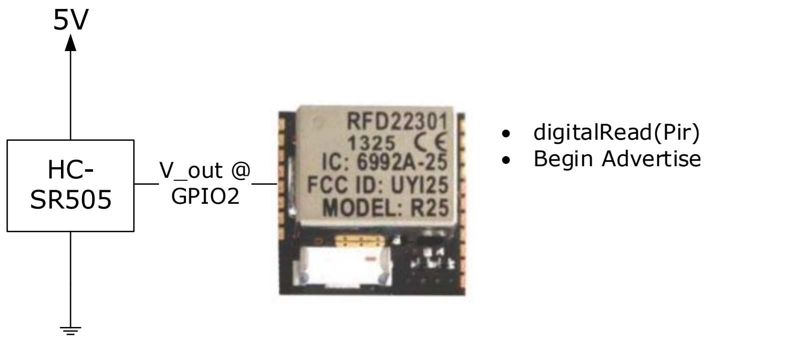
VDD
V_out @ GPIO4
10K
RFD22301 1325 CE IC: 6992A-25 FCC ID: UYI25 MODEL: R25
• analogRead(SensorLPin)
• Store Data in Variable LightValue

(a) getLightSensorValue()



5V
LM35 DZ
V_out @ GPIO5
RFD22301 1325 CE IC: 6992A-25 FCC ID: UYI25 MODEL: R25
• analogRead(SensorTPin)
• Store Data in Variable TempValue

(b) getTemperatureSensorValue()



(c) PIR Presence Detection

Figure 3.12: Data Flow Between Sensors and RFD22301

millivolts, that comes from the temperature sensor, to Celsius, first it is necessary to divide the value received by the ADC resolution. After dividing by 1024, to finish the conversion, the value will be multiplied by the reference voltage, which is 5V, which is 5000 mV (since the sensor returns a value in millivolts, the reference voltage has to be in millivolts too). The returned value will be a float, since the sensor allow values with decimal cases, and will be stored in the *float* variable *TempValue*. The data flow of this action is described in fig.3.12(b).

actuatorsAction(char c,int v)

This procedure handles the automatic and manual control for the illumination. Each time a packet is received from the mobile application by the procedure *onReceive()*, as explained above, a filtering is performed to select what procedure will be called. In this case, that means that an instruction related to the automatic or manual control has been sent. If a request for activating the automatic control is sent, the variable *autoControl* is set to true, that will indicate to the application that the routine to check the light level needs to be called each cycle. If the request is to indicate that manual control has been activated, a value corresponding to the desired light level is received and will be sent to the LED driver.

autoControlFunc(int lightValue)

For the auto-control implementation, it was necessary to define a reference value for the ideal illumination level. To achieve a greater comfort in the illumination control, the objective is to adapt the illumination level measured to a level defined as ideal. For different tasks and spaces, the ideal value varies so it is only possible to approximate the value based on recommend light levels [25]. Table 3.4 presents some of the recommended values for different activities.

The value defined as ideal in the application is 400, due to being comprised between the values defined for office work. Nevertheless, since the system is re-programmable, this value can be changed easily. Each time this function is called, a verification is made to check if the value sense in the light sensor is above or below the ideal, and then an increment or decrement is made accordingly, which will also avoid a abrupt change in the illumination if a dimming value was calculated. It will also prevent situation as momentarily change on outside illumination, due the clouds passing by for example, that would reflect in abrupt changes on the lamp intensity.

Table 3.4: Indoor Light Values

Activity	Illumination Level (Lux)
Public areas with dark surroundings	20 - 50
Simple orientation for short visits	50 - 100
Working areas where visual tasks are only occasionally performed	100 - 150
Warehouses, Homes, Theaters, Archives	150
Easy Office Work, Classes	250
Normal Office Work, PC Work, Study Library, Groceries, Show Rooms, Laboratories	500
Supermarkets, Mechanical Workshops, Office Landscapes	750
Normal Drawing Work, Detailed Mechanical Workshops, Operation Theaters	1000

timer_config(void)

In this procedure will be defined the parameters for the system timer[2]. The timer main parameters are:

- PRE-SCALER
- CC[0]
- attachInterrupt()

The *PRE-SCALER* is used to set the timer frequency. It follows the equation:

$$\frac{Oscillator\ Frequency}{2^N} = Timer\ Frequency$$

N will be the PRE-SCALER factor, which means with a N equal to 9 (value used in this timer configuration) and knowing that the internal oscillator frequency is 16 MHz, the timer frequency that will be set is 31250 Hz. The period of 31250 Hz is 32 μs, so in the *CC[0]* parameter, the next calculation needed is the multiplication factor that will be one millisecond (one thousand μs) divided by 32 μs:

[2]This timer does not define the communications timings, there is a timer dedicated for BLE communication.

$$\frac{1000}{32} = 31.25$$

After finding the multiplication factor, in the application constant values it is defined the *trigger value*, which in this case will be one second (one thousand milliseconds) which will be easy to handle for counting. So the parameters *CC[0]* will be the desired *trigger value* multiplied by the multiplication factor:

$$CC[0] = TRIGGER\_VALUE * (31) + TRIGGER\_VALUE * (\frac{1}{4})$$

TIMER1_Interrupt(void)

Each time a timer interrupt occur, this procedure is called and the instructions to perform at each timer event are defined here. The application has a time variable (*aux_timing*) that will be increment every timer event. Since timer events occur each the variable will work as a counter set in seconds.

The complete decision chart can be observed in fig.3.13, and is implemented by the code present in Appendix B.1. Although this application being designed for indoor, some concepts have been adapted from an street light control system [26]. After main program starts, the automatic control starts too due to the *AutoControl* variable that is set on by default. Then the application wait for a presence detection, when movement is detected, advertising starts. After each verification of movement, since the application is set on automatic control, the light level is verified and a adjustment is made if needed. When on advertising, the application will check if a connection has been established, if not and it is not the fourth time without a connection, then it proceeds to verify the light level and wait for a connection, if it is the fourth time, it means that two minutes have passed without a connection so the application assume that the user has no intention to connect, returning to the presence detection state. If a connection is established, the application will send the light and temperature values retrieved in sensors, each thirty seconds, and check if instructions have been sent by the user. The instructions possible are *refresh* and *automatic/manual control*, where in manual control a value is sent by the mobile application as explained before. If the manual control is set ON, the *AutoControl* variable will be set to false, which means that no verification on the illumination level will be performed, the light level will be the one choose by the user. If an user disconnect after setting the light level, it will remain until the user changes it or set the automatic control on.

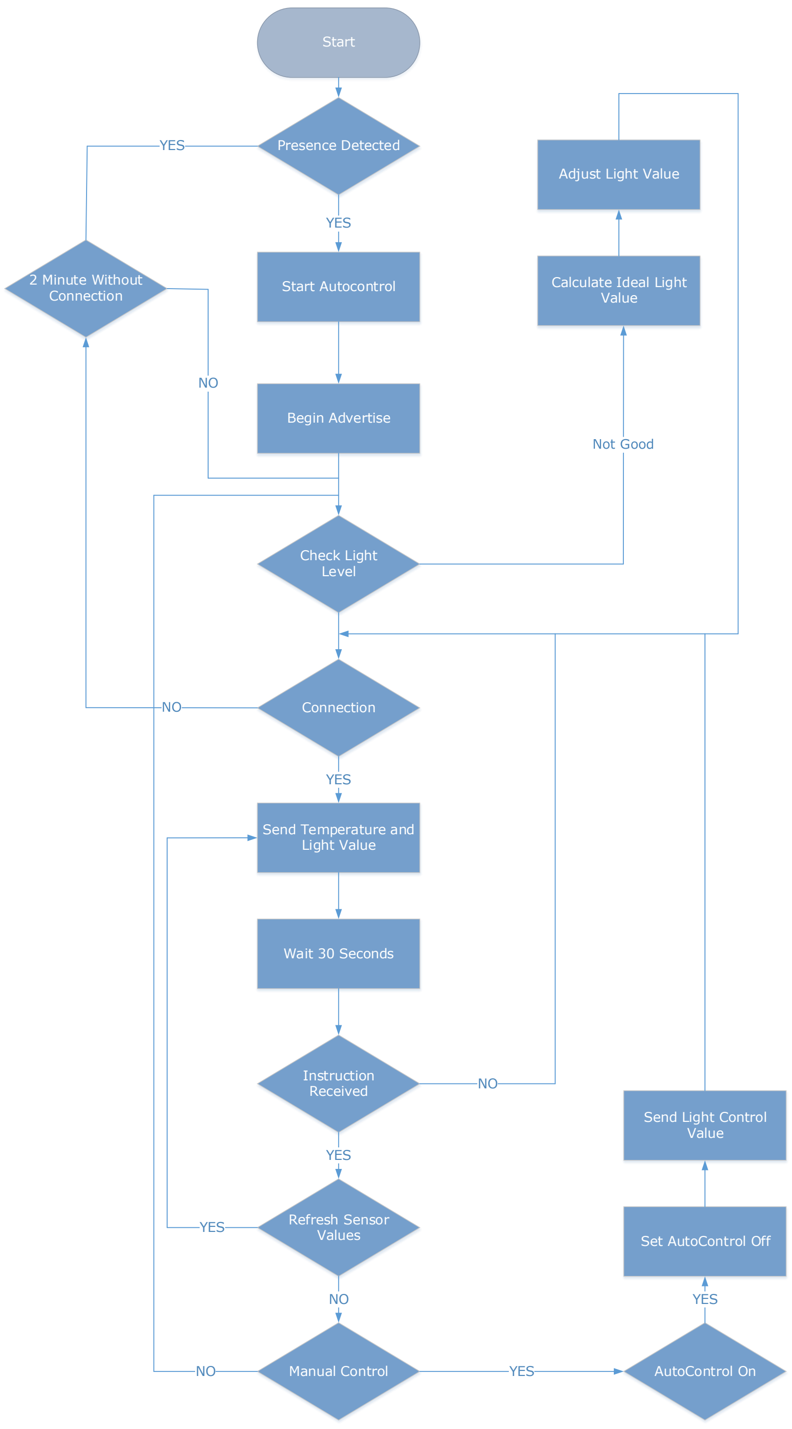


Figure 3.13: Complete Decision Diagram

3.3 BlueIoT Hardware

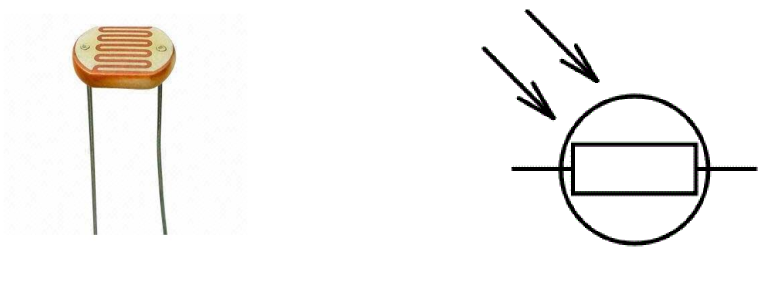
3.3.1 Sensor Module Development

The ambient variables measured by the *smart system* are light level, temperature and presence. Thus, the sensors module will be composed by a light, a temperature and a PIR sensor.

Light Sensor

There is several options for light sensitive components, in this case the choice was between two component: Light Dependent Resistor (LDR) and phototransistor.

A LDR, or photoresistor, is a photoelectric device. It is a resistor with variable resistance in function with the intensity of light incidence [27] [28]. The LDR usually exhibits high impedance, nearly 1MΩ in dark ambients (there are LDR with the opposite behavior), and when the light intensity increases the impedance suffer a drastic reduction on his value that can go to 100Ω approximately.



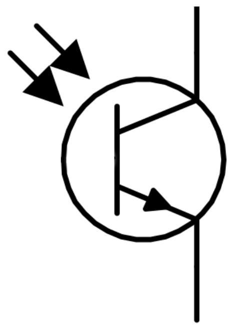
(a) LDR (b) Circuit Symbol

Figure 3.14: An example of a LDR and the Circuit Symbol

A phototransistor is a photosensitive device, based in the regular transistor but with a transparent cover, which makes easier for light to reach the base-collector junction. Although normal transistor exhibit photosensitive effect, in phototransistor their structure is designed to optimize it [29]. When photons contact the base-collector junctions, the electrons start to flow, creating a current in base, that will be amplified by the transistor's current gain β.

The first approach for measuring light used a LDR as photosensitive device. The first problem detected was related to the LDR sensitivity. Due to the

(a) Phototransistor



(b) Circuit Symbol

Figure 3.15: An example of a Phototransistor and the Circuit Symbol

lack of precision in the measure process and different values obtained in the same lighting conditions, the conclusion was that a LDR only fitted in this system if the objective was to use light values in a switch, i.e, detect the presence or absence of light only. Since phototransistor have an higher sensitivity to light changes and the values obtained did not changed too much in similar light conditions, it allows the system to act according the light conditions in a space and to do it with an higher precision.

The phototransistor circuit (fig.3.16) consists of using the phototransistor as a *Common-Collector* amplifier, biased with a 10 kΩ between emitter and ground [30]. Usually the topology used is the *Common-Emitter* amplifier, due to gain, but in this case since the gain is not relevant, the *Common-Collector* was choose for offering a crescent output voltage for crescent values of light. Thus, it is possible to analyze the values directly in the MCU, which facilitate the decision making of the system.

The component chosen was a BPW85A by Vishay [31]. It is a silicon NPN phototransistor with high sensitivity and fast response times to light variation. The relation between voltage output and Lux (SI unit for Illuminance) is shown in fig.3.17, where the values for Illuminance were obtained at different distances from the light source and measured with the application *Light Meter* [32] while the voltage values for each level were measured at the phototransistor pins. After twenty measures, the graphic was generated on Matlab using both linear and cubic interpolation. It is possible to observe three slope level: from zero to two hundred Lux, from two hundred to five hundred Lux and finally after the five hundred level the slope start to stabilize.

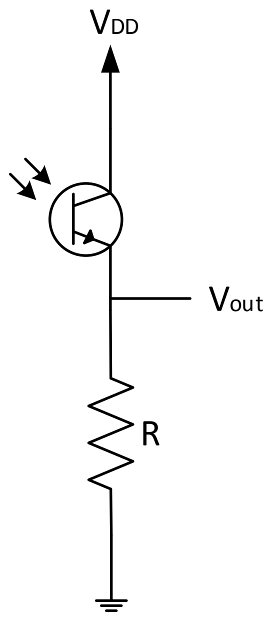


Figure 3.16: Common-Collector Topology Using Phototransistor

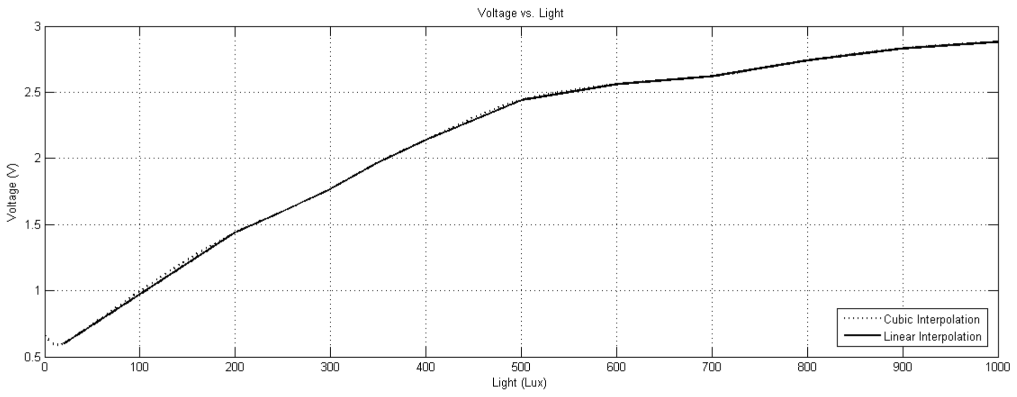
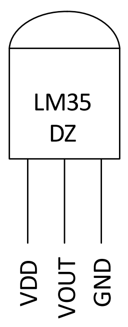


Figure 3.17: Relation between Lux and Voltage Output of the Phototransistor Circuit

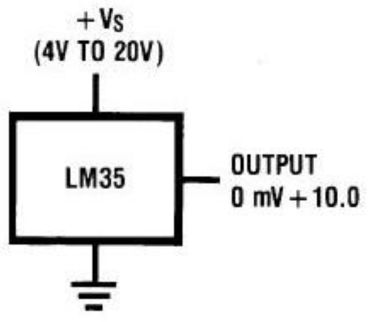
Temperature Sensor

Since the objective is to integrate all sensors in the smallest circuit board possible with the lowest cost too, this constraint led us to choose an integrated circuit sensor rather than try to implement with other options like mechanical or electrical sensors.

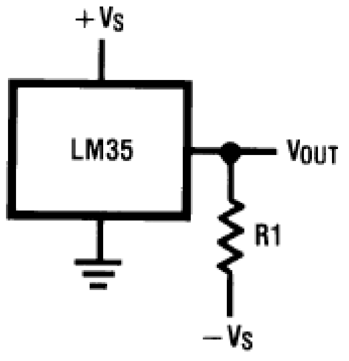
The module chosen is the LM35-DZ temperature sensor by Texas Instruments [33]. It is calibrated directly in Centigrade temperature and has a



(a) LM35 DZ Pins



(b) Basic Range Configuration



(c) Full Range Configuration

Figure 3.18: Symbol and Configurations of LM35-DZ [33]

precision of approximately ±1\4 °C when set in the basic mode for the range of 2°C to 150 °C. There is two possible configurations to use the LM35-DZ: *Basic Range*(fig.3.18(b)) and *Full Range*(fig.3.18(c)). Since the *Full Range* configurations returns a negative voltage output and the ADC of our module only allows positive voltages, it is used the *Basic Range* configuration.

Passive Infrared Motion Sensor

A PIR is a device that senses motion through infrared waves. The sensor contains a material infrared sensitive with two slots covered by a Fresnel lens. The signal output of the PIR is generated when a warm body crosses one of the slots range, causing a differential voltage across the material.

The sensor choose was a HC-SR505, due to his reduced price and low-voltage operation (4.5-20V) [34]. The sensor can sense presence in a 3 meters range inside a 100 angle vision.

3.3.2 Actuators

On smart control systems that work with light, the actuators are mostly circuits to turn on or dim lamps. The way light dimming is performed on lamps differs from their type, i.e, incandescent lamps are dimmed through the variation of voltage and LED, since they work as semiconductors, will work at constant voltage and current. So the way to dim LED is using Pulse Width Modulation (PWM)(fig.3.20). Longer pulses means higher intensity while smaller pulses means lower intensity. As depicted in fig.3.19 several manufacturers have developed modules for integration with microcontrollers, designed for dimming lamps through PWM signals [35, 36].

(a) Digital AC Dimmer by InMojo [35]

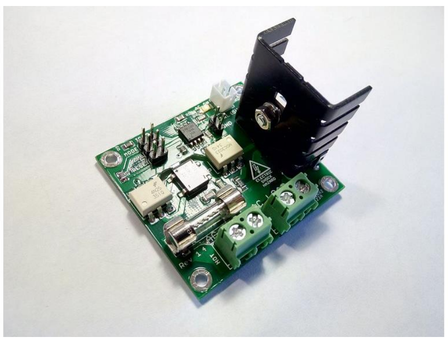

(b) AC 50/60 Hz Dimmer/SSR Control Board by Tindie [36]

Figure 3.19: Example of Commercial AC Dimmers

The way how PWM controls the light intensity involves another concept: *Zero Cross Detection*. It consists of detecting when the 230 VAC wave (fig.3.21) from supply crosses zero. This detection will be done with an optocoupler, after the signal rectification by a diode bridge-rectifier, obtaining a wave with only positive values (fig.3.22).

The optocoupler will have as input the rectified wave on the diode side. Each time the wave is below a certain threshold, the diode stops conducting(fig.3.23(b)), which will stop the phototransistor conduction on the other side. When the phototransistor stops conducting, a current starts flowing on the zero-cross pin. When the wave is above the threshold and the diode is conducting (fig.3.23(a)), the phototransistor will be conducting too, forming a current flow that passes through the phototransistor ending in the grounded pin. The resulting output on the zero-cross pin will be pulses for each time the signal is below the threshold, i.e. two times for each period. Since the frequency for AC voltage home supply is 50 Hz

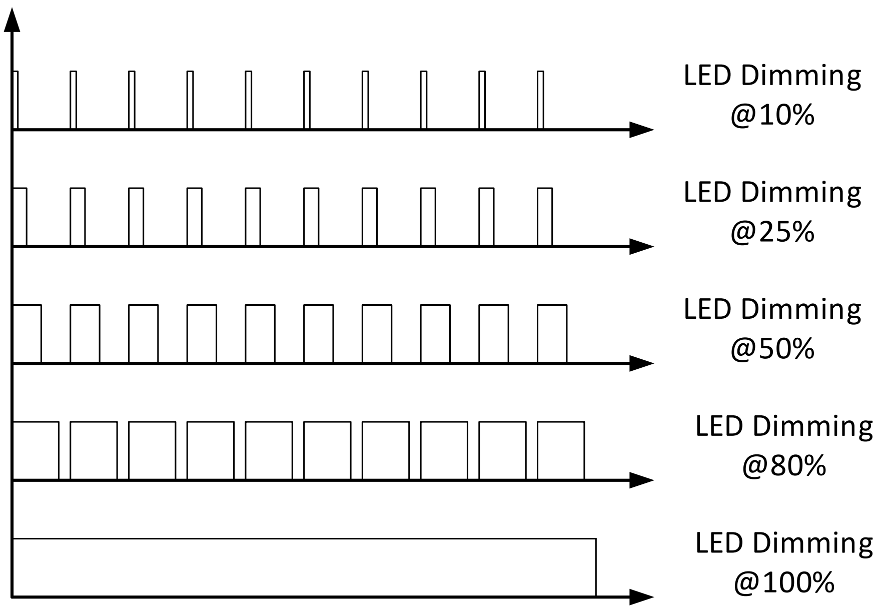


Figure 3.20: PWM Signals at Different Dimming Rates

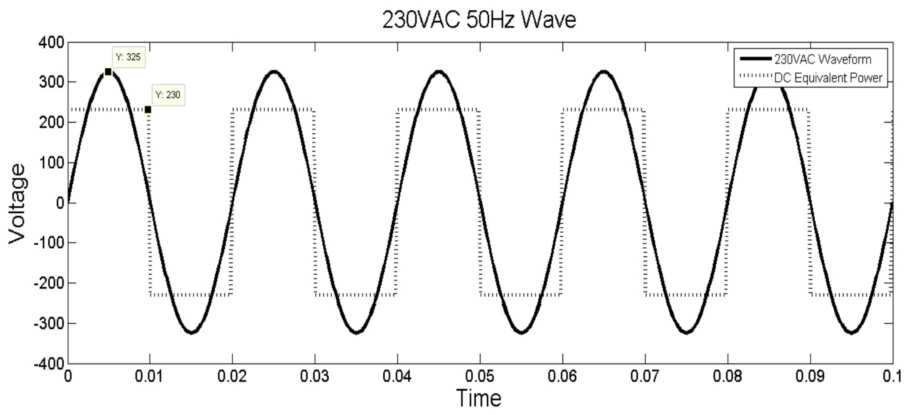


Figure 3.21: 230VAC 50Hz Waveform

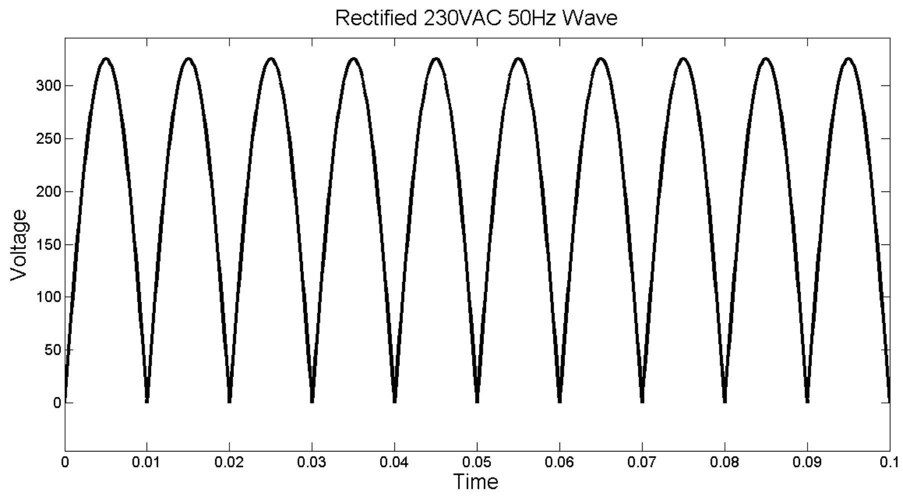


Figure 3.22: Rectified 230VAC 50Hz Waveform

(for Europe), the zero-crossing time can be calculated by:

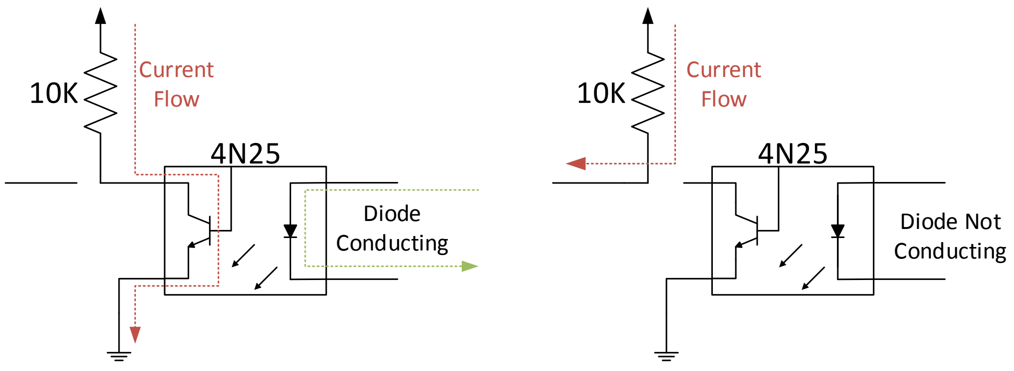
$$t = \frac{1}{f}\ ,\ with\, f\ =\ 50Hz$$

$$t = 20ms\ and\ \frac{t}{2} = 10ms$$

In fig.3.24, it is shown the 4N25 optocoupler output, that as expected, has each pulse separated by 10 milliseconds. The waveform not being exactly a pulse is explained by the voltage threshold of conduction not being exactly zero and by the LED switching time t_{rise} and t_{fall} that are, accordingly to the datasheet [37], 2 μs.

In fig.3.26, the *Zero Cross Detection* part is represented by the red square.

The *Zero-Cross Detection* allows that the start of each semi-period can be known by the microcontroller, which leads to the second part of light control: the light intensity leveling. The PWM(fig.3.20) is applied through a digital signal that comes from the MCU, that set the optocoupler's diode on conduction, therefore setting the triac output on conduction too. The triac placed between the optocoupler and AC load, will be open each time his base receive a signal from the optocoupler, specifically each time a digital *HIGH* is set on the optocoupler input. Once the triac open, the AC load will be supplied by the 230 VAC, consequently being turned on. The triac is closed when the 230 VAC crosses zero. The light intensity is defined by the optocoupler input timing, since the zero-cross detection is synchronized with this dimming part, each time a zero-crossing is detected, a delay is applied ($t_{dimming}$) and the light intensity will be inversely proportional to $t_{dimming}$ as a smaller delay means more time the lamp is supplied(3.25).



(a) Optocoupler 4N25 Conducting (b) Optocoupler 4N25 Not Conducting

Figure 3.23: Conduction on the Optocoupler 4N25

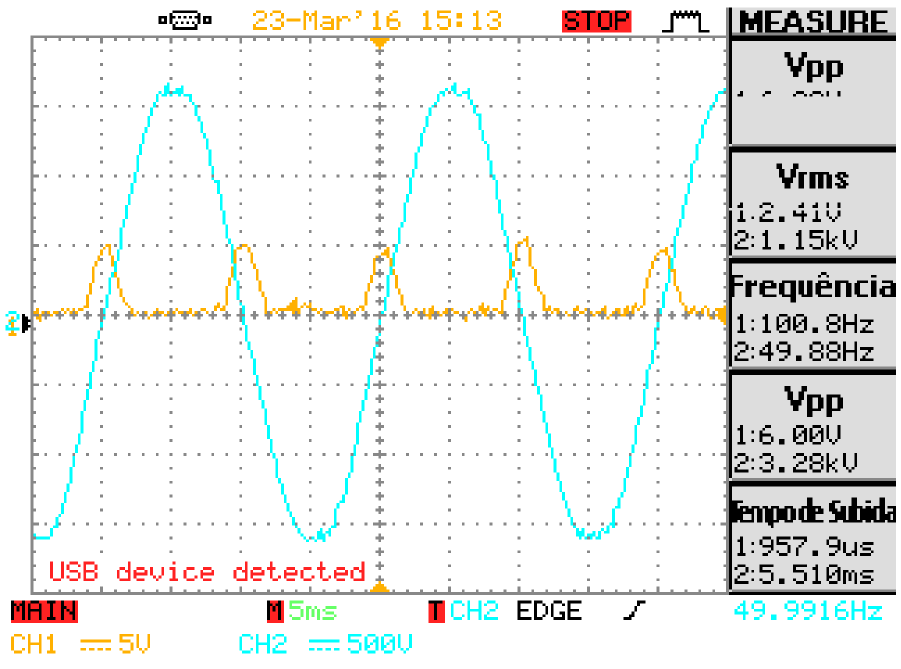

```
23-Mar'16 15:13   STOP   MEASURE
Vpp
Vrms
1:2.41V
2:1.15kV
Frequência
1:100.8Hz
2:49.88Hz
Vpp
1:6.00V
2:3.28kV
Tempo de Subida
1:957.9us
2:5.510ms
USB device detected
MAIN   M 5ms   T CH2 EDGE   49.9916Hz
CH1 5V   CH2 500V
```

Figure 3.24: Output of 4N25 Optocoupler

Synchronism required to dim a lamp without flickering is tight. The only way to do it was with hardware interrupts, but since it is needed at least one more to control the radio timings to process the BLE communication, conflicts with interrupts will occur. In this scenario, the priority is always for the communication, which in high data throughput will severely affect the others interrupts. As depicted in fig.3.25, if the triac opening is ahead or delayed, the lamp will be supplied more or less time that supposed, which will result in an higher or lower light intensity. Further, if the opening time is different at each wave period, that means a different light intensity every 10 milliseconds (for the case that in every cycle the triac is open out of time). The resulting flicker is so high making the circuit inviable. So the solution passed by include a separated microcontroller to control the dimming process by receiving directly from the RFD22301 a dimming value. The RFD22301 now does not have the control timings to generate the PWM wave, liberating the timer destined to do so, allowing now the implementation of a timer with less precision needed, that will be used to generate a one second spaced interrupt for the MCU to know when to send sensors values. Since the BLE module was programed in Arduino IDE, the microcontroller choose to control the dimming circuit was an Arduino Mega 2560. Code present in the Arduino can be consulted in Appendix B.2.

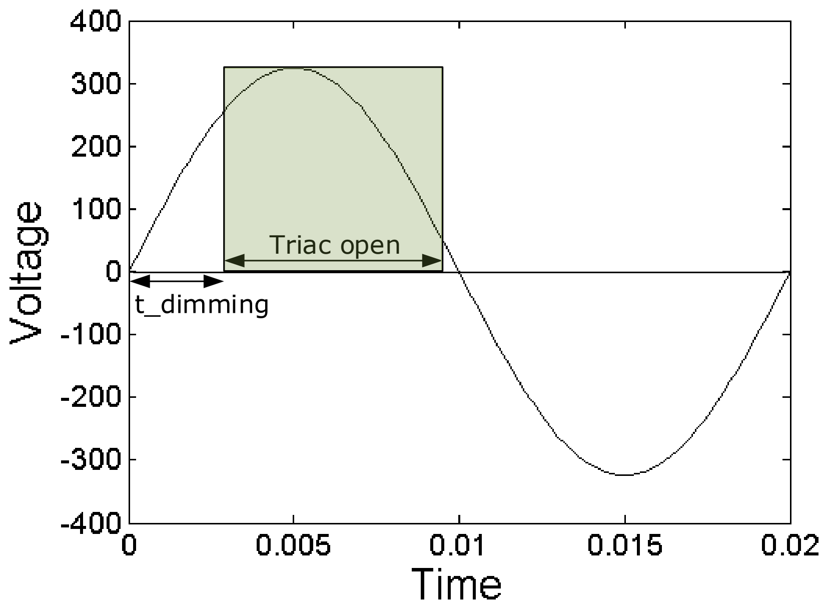


Figure 3.25: Triac Opening According to $t_{dimming}$

In fig.3.26, the *Dimmer* part is represented by the green square.

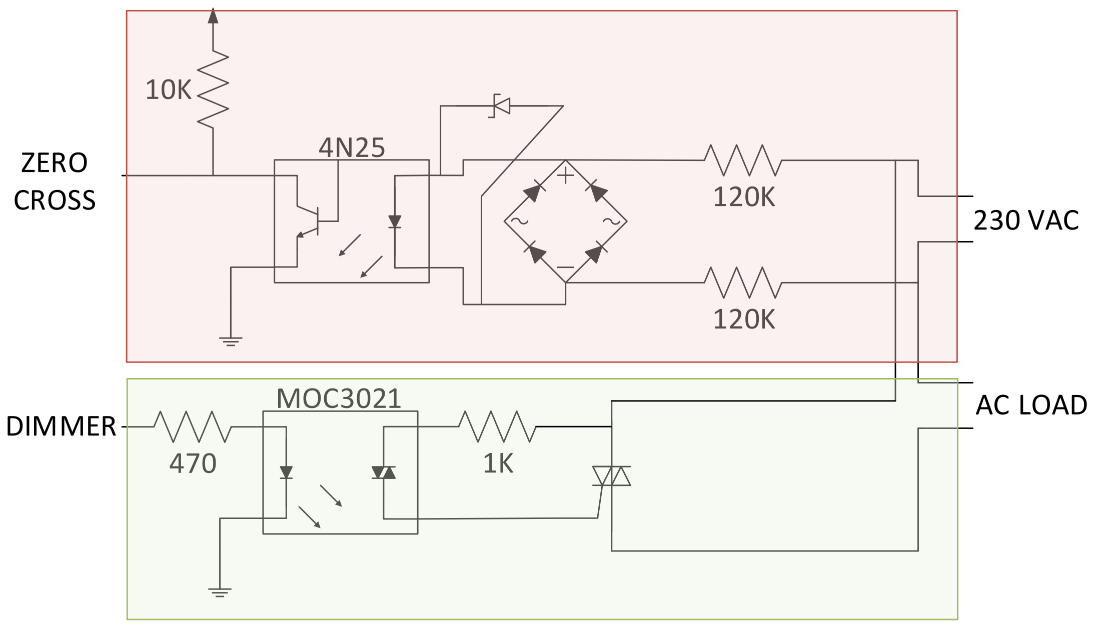


Figure 3.26: AC Led Dimmer Driver Circuit

3.4 Printed Board Circuit Design

To finalize the system development, a Printed Circuit Board (PCB)was designed with the help of EAGLE software. One of the initial constraints about the system implement was the circuit board size, that had to fit in an equipment box(fig.3.30) to be embedded in a wall after. The size of equipment boxes varies from different vendors, so the value used as reference was a 60 by 60 millimeters square. As shown in fig.3.4, this value is respected, with the circuit board size being 59 by 52 millimeters, where some space is available for cutting corners.

A division had to be made in the circuit, due the different voltages present: 5V and 3.3V for the RFD22301, sensors and half of the dimming circuit, and 230V coming from home supply that will be connected to the triac, bridge rectifier and lamp. A 10mm space was left open to separate the parts working at different voltages to prevent the remaining parts of the circuit to burn due to high voltage. Also, since 230V are very dangerous and even mortal, the space was used to left a message to avoid touching the board when switched on. Another concern about the 230V voltage was the track width. To support high voltage, the track needed to be wider, so with the help of PCB track width calculator, the minimum width for the tracks with 230V was calculated [38]. For the remaining tracks working at low voltage, a thicker trace was used. In the datasheet [22], some

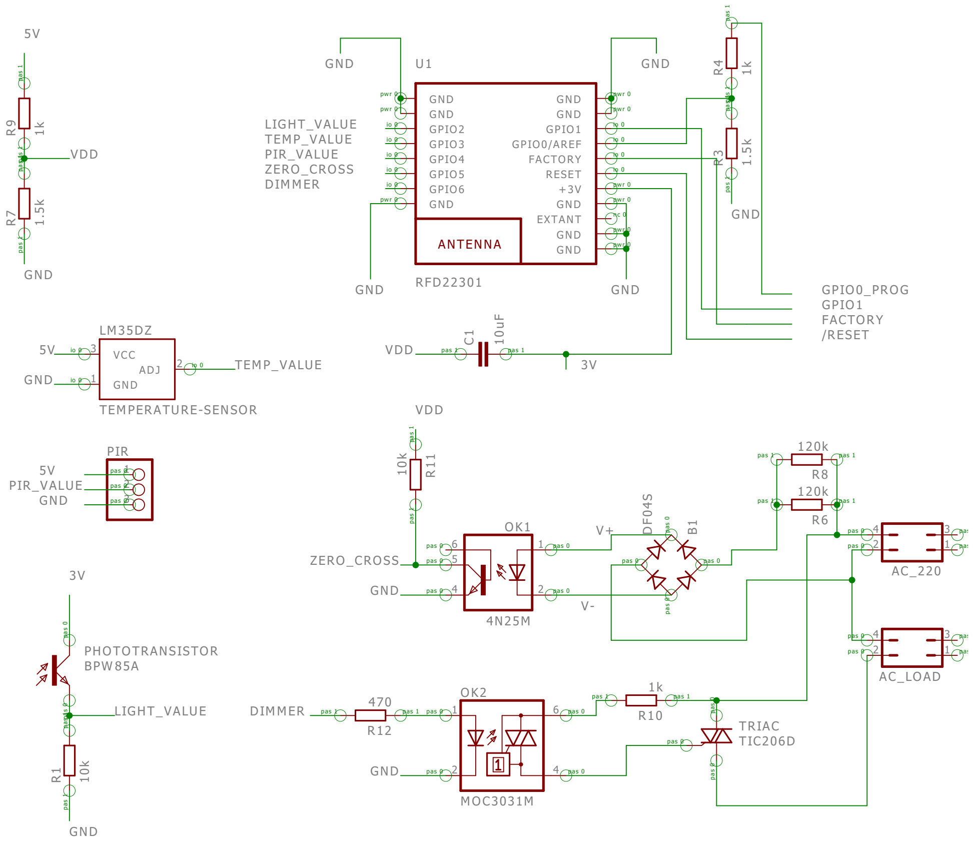
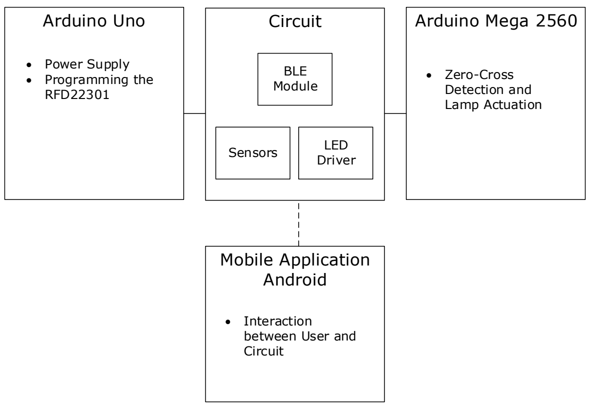


Figure 3.27: Complete Circuit



Arduino Uno
Power Supply
Programming the RFD22301
Circuit
BLE Module
Sensors
LED Driver
Arduino Mega 2560
Zero-Cross Detection and Lamp Actuation
Mobile Application Android
Interaction between User and Circuit

Figure 3.28: Prototype Block Diagram

recommendations are made for the module integration in a PCB, in order to improve the antenna performance. Thus, the module localization in the board is on the inferior left corner, leaving some around the antenna without coper or components to maximize the signal output and minimize interference.

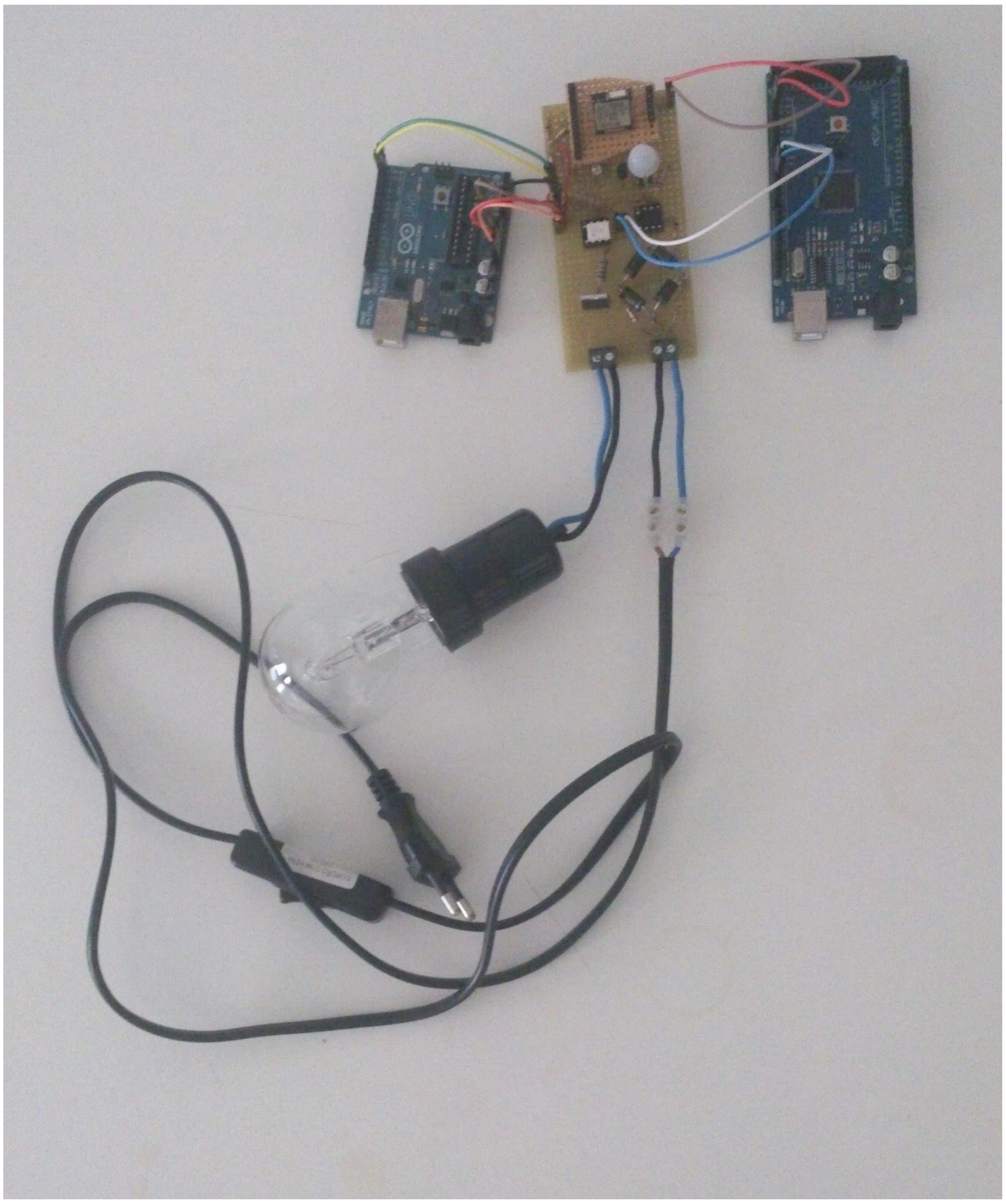


Figure 3.29: Proposed Prototype

Figure 3.30: Example of an Equipment Box

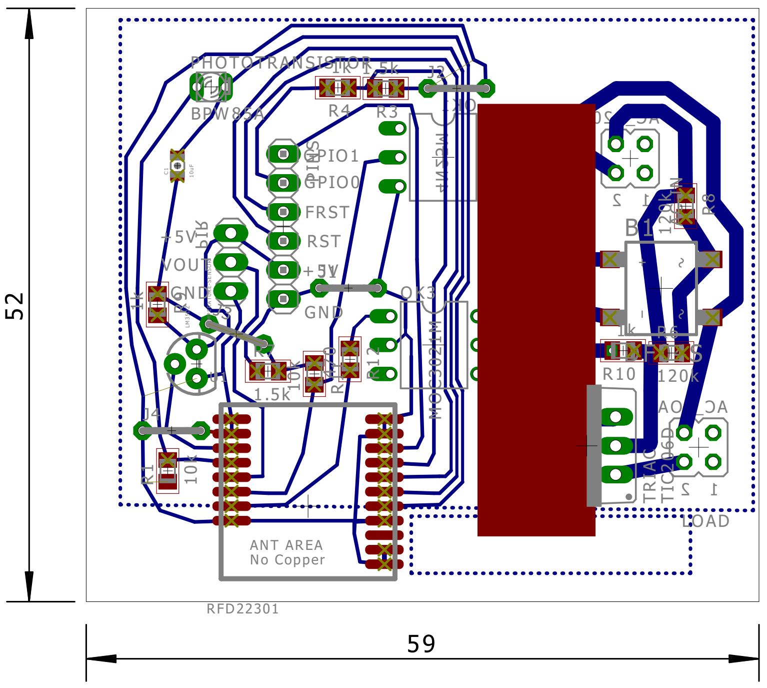


Figure 3.31: PCB Designed

CHAPTER 4

BLE Energy Budget Analysis

4.1 Power Consumption

The measurements for power consumption were made through a shunt resistor used to convert the resistor voltage into current using the following equation:

$$Power = V_{dd} * (\frac{V_{rms}}{R_{shunt}}) \tag{4.1}$$

The equipments used for measurements were:

- Iso-Tech IDS8062 60MHz, 2 Channel Digital Oscilloscope
- Xindar DB600.031 Digital Multimeter
- Resistor 10 Ω ±5%

There is six states where power consumption calculation matters:

- Deep Sleep
- Idle
- Advertising
- Connecting
- Receiving Data
- Sending Data

4.1.1 Deep Sleep

The *Deep Sleep* state is the main responsible for the low power consumption in BLE since it allow to implement delays, both in the communication process, as between the timings to send or receive data, or in the middle of application actions when delays can occur. Accordingly to the module datasheet [22], the average current consumption is 4 μA that corresponds to an expected power consumption of 13,2 μW with a 3.3V supply, which is far below power consumed in other states.

4.1.2 Idle

When the module enters *Idle* state it does not advertise nor is able to start a connexion. This state is used to perform actions that does not involve communication with others devices as retrieving data from sensors. Unlike *Deep Sleep* where only the oscillator is running, in this state the MCU needs to be on so it can perform actions. The average current consumption for the MCU is around 4 mA, which results in an expected power consumption of 13,2 mW .

4.1.3 Advertising

As the module starts advertising, packets are sent which results in an power consumption increase. In order to control power consumption some parameters can be defined, as the *advertising interval* and the number of packets sent in each interval. Fixing the number of packets sent, a larger interval results in a minor power consumption. Fixing the time interval, sending more packet per interval will increase power consumption. The advertising power consumption is not referred in the module datasheet, so from fig.4.1 to fig.4.7, samples of advertising intervals are shown for different transmit power, in which it is possible to observe variables like peak-to-peak voltage - V_{pp} and root-mean-square voltage - V_{rms}, alongside with the waveform in which is possible to detect, through peak voltages, advertising packets being sent.

The average values for current consumption, in the advertising state, are present in table 4.1. As it can be in fig.4.1 to fig.4.7, when an advertising packet is sent, a peak voltage occur, which means that the power consumption will be higher. It can occur that in a sample of an advertising interval peak voltages are higher for a smaller output power comparing to a sample with an higher output power. Another constraint relatively to measurements is that in the oscilloscope, sample with the size of an advertisement interval are captured, but it does not necessarily means that it is a complete advertising interval (it is more probable

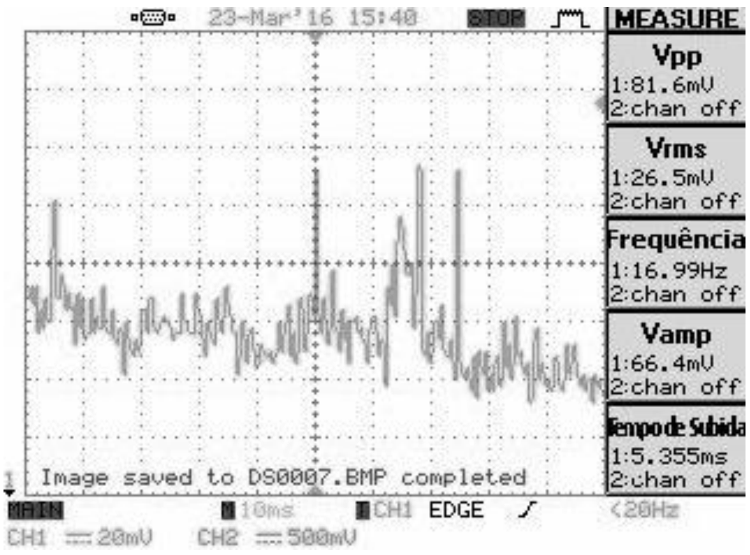


Figure 4.1: Voltage Drop in Advertising State with +4 dBm Output Power

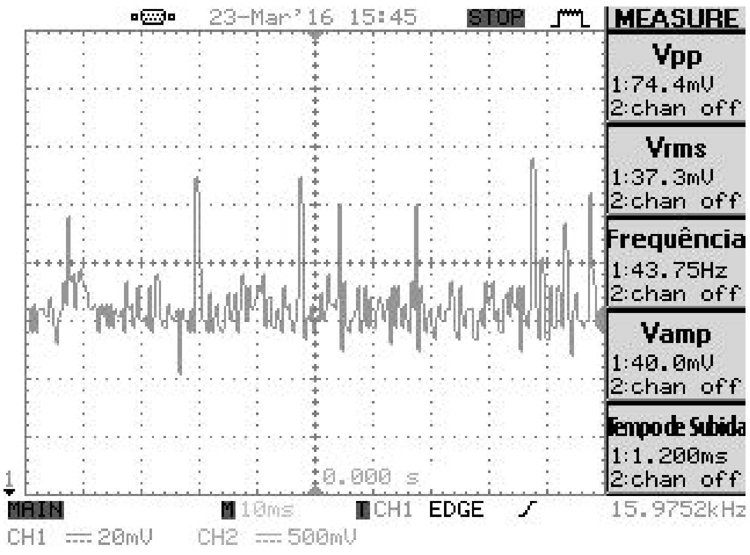


Figure 4.2: Voltage Drop in Advertising State with 0 dBm Output Power

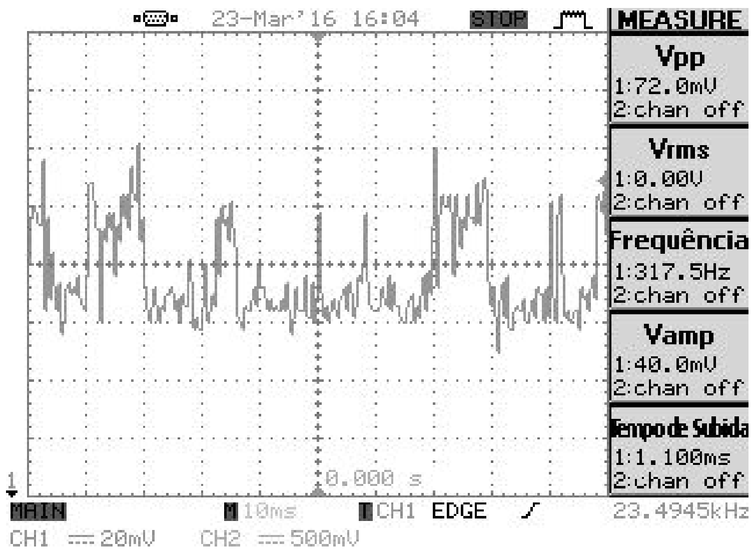


Figure 4.3: Voltage Drop in Advertising State with -4 dBm Output Power

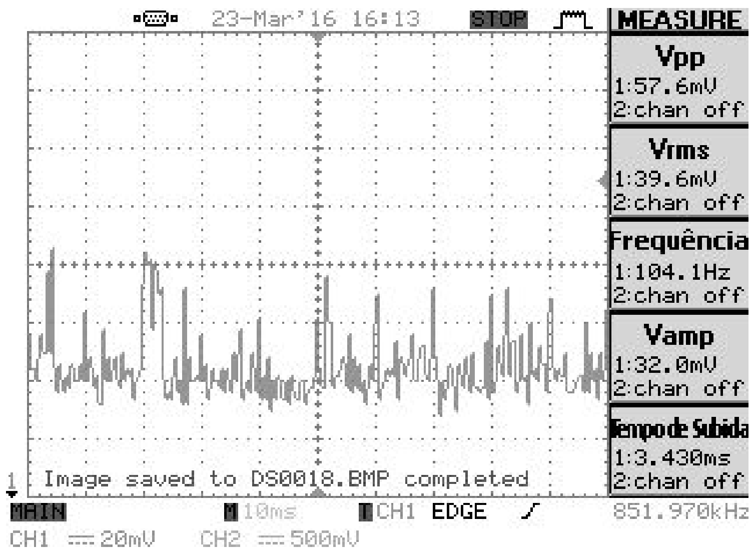


Figure 4.4: Voltage Drop in Advertising State with -8 dBm Output Power

that the waveform in oscilloscope is two different advertising intervals). A problem that comes from this fact, for example in fig.4.1 is visible, some samples does not have the expected ten advertising packets, which can reflect a smaller V_{rms} resulting in a smaller average power consumption.

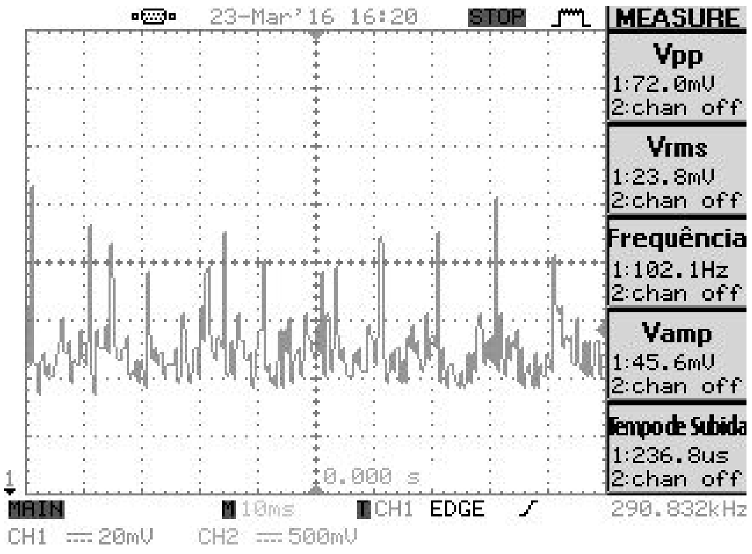


Figure 4.5: Voltage Drop in Advertising State with -12 dBm Output Power

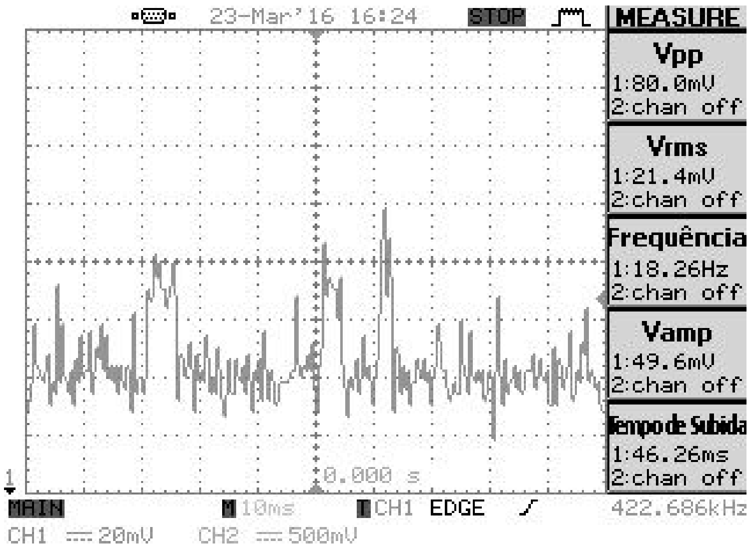


Figure 4.6: Voltage Drop in Advertising State with -16 dBm Output Power

4.1.4 Connexion

When establishing a connection, it is expected a power consumption increase due to packet exchange between devices to define the connexion parameters. After the connexion establishment, the power consumption will be proportional to the

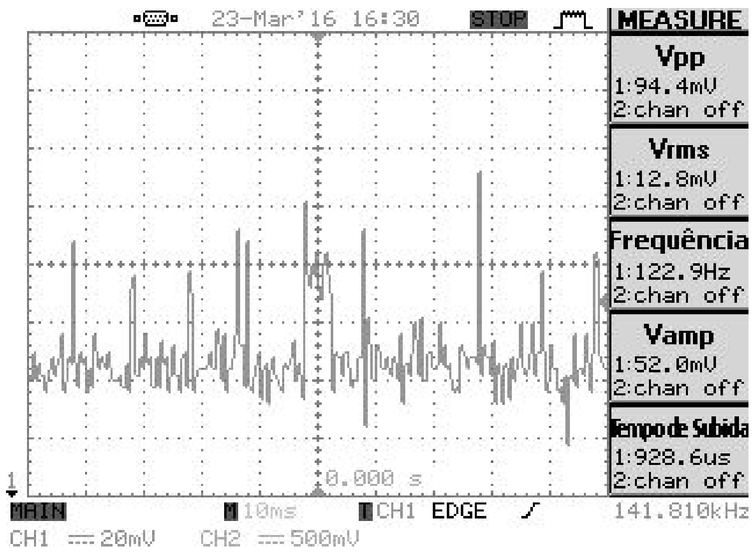


Figure 4.7: Voltage Drop in Advertising State with -20 dBm Output Power

Table 4.1: Average Current Consumption for Different Output Power in Advertising

Output Power	Average Current
+4 dBm	3.52 mA
0 dBm	3.48 mA
-4 dBm	3.43 mA
-8 dBm	3.35 mA
-12 dBm	3.29 mA
-16 dBm	3.22 mA
-20 dBm	3.18 mA

data throughput.

4.1.5 Sending and Receiving Data

The values obtained during the measurements made sending a packet of 4 bytes every second, the maximum size that the module will trade with the mobile application, are presented in table 4.3.

When measuring the current consumed in the receive state, the values measured were all equal, around 4,4 mA.

Table 4.2: Average Current Consumption for Different Output Power during a Connection Event

Output Power	Average Current
+4 dBm	4,89 mA
0 dBm	4,82 mA
-4 dBm	4,75 mA
-8 dBm	4,73 mA
-12 dBm	4,72 mA
-16 dBm	4,71 mA
-20 dBm	4,70 mA

Table 4.3: Average Current Consumption for Different Output Power Sending a 4 bytes Packet

Output Power	Average Current
+4 dBm	4,42 mA
0 dBm	4,39 mA
-4 dBm	4,38 mA
-8 dBm	4,38 mA
-12 dBm	4,37 mA
-16 dBm	4,36 mA
-20 dBm	4,36 mA

4.2 Received Strength Signal Information

RSSI measurements are an indicator of the signal strength received by devices and are used to map an area where it possible to establish a connection. There is two factors that have influence on the RSSI value:

- Distance
- Output Power

Based on those two factors, some measurements were made for different distances and output power. The output power range that can be set on the RFD22301 is from +4 to -20 dBm, for multiples of 4 [22]. As expected, an higher output power results in an higher RSSI. Relatively to distance, when the distance is higher, the RSSI comes down.

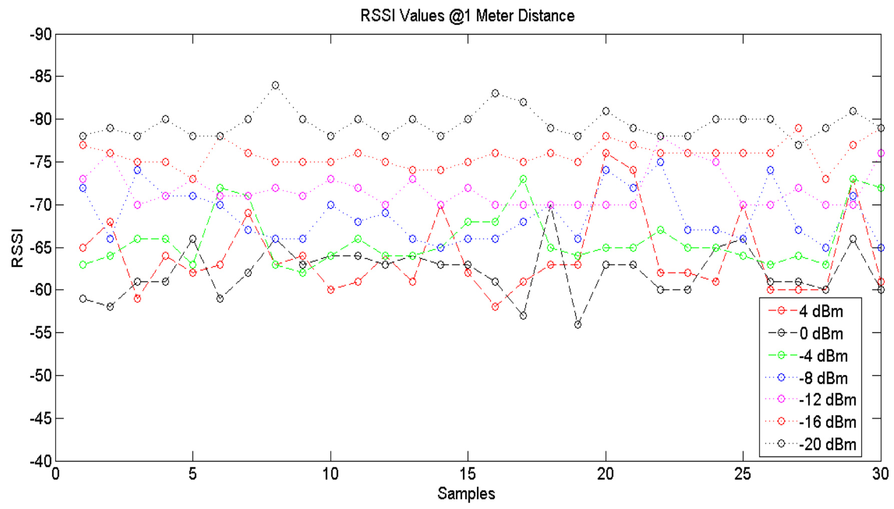


Figure 4.8: RSSI Values at 1 Meter Distance

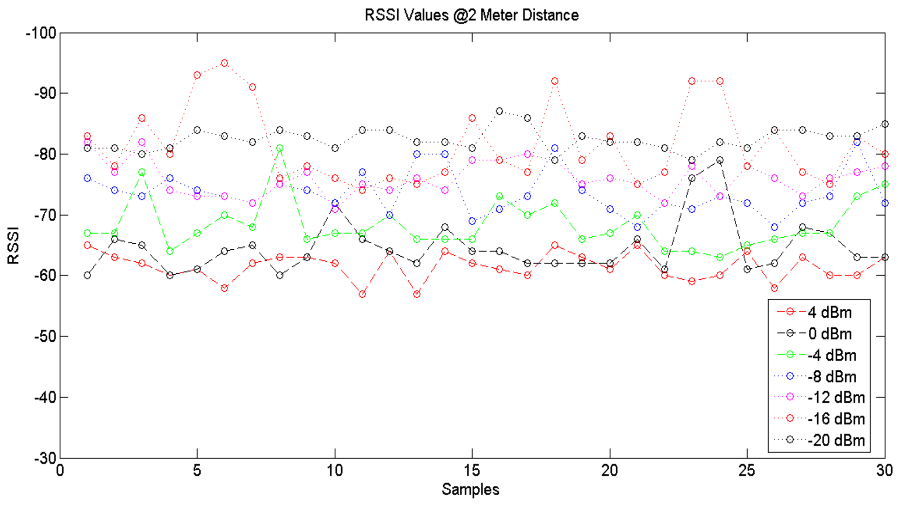


Figure 4.9: RSSI Values at 2 Meters Distance

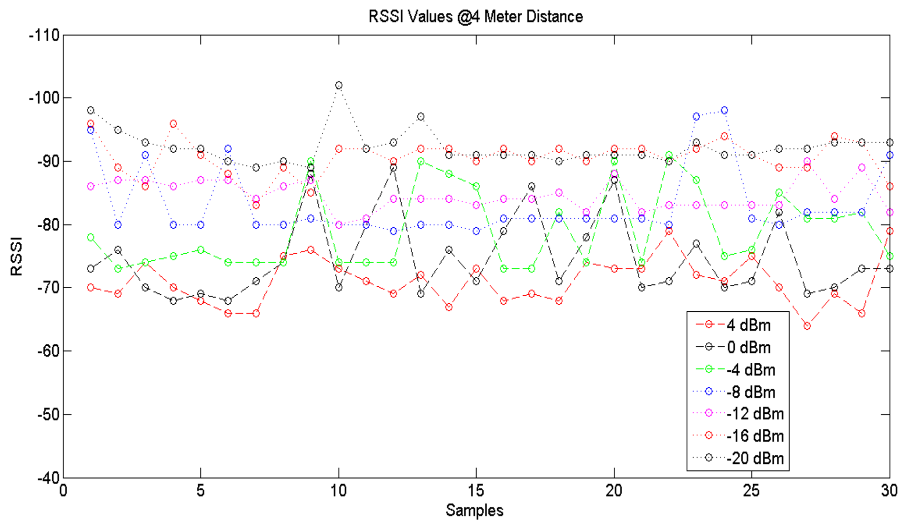


Figure 4.10: RSSI Values at 4 Meters Distance

Table 4.4: Average RSSI Values for Different Distance and Output Power

Output Power	Distance		
	1m	2m	4m
+4 dBm	-63.97	-61.50	-70.97
0 dBm	-62.17	-64.60	-74.63
-4 dBm	-65.90	-68.37	-79.10
-8 dBm	-68.67	-73.60	-83.20
-12 dBm	-71.83	-75.97	-84.60
-16 dBm	-75.80	-81.57	-90.57
-20 dBm	-79.43	-82.47	-92.23

4.3 Implementation Costs

In this section, the list billing for the components used is presented in table 4.5.[1].

The circuit implemented has a lower cost than one of the commercial circuits seen earlier [35, 36] for dimming, that costs around 35 €, and have more functionalities as the communication and sensors. The sensor part is also cheaper than the commercial sensor boards seen [4, 5], although some questions can be raised about the performance, since the objective here was to use low-cost components. Finally comparing with ZigBee, modules costs between 20 and 40 € [39, 40], which is above the RFD22301 cost. As referred in the introductory chapter, one of

[1]The HC-SR505 was ordered on Ebay, while the rest at www.mouser.pt or avalaibe at the university

Table 4.5: Billing List of the Prototype Components

	Component	Quantity	Price
BLE Module	RFDuino22301	1	18,10 €
	100pF Capacitor	1	0,10 €
	1kΩ 1/2W Resistor	1	0,06 €
	2kΩ 1/2W Resistor	1	0,06 €
Sensors Module	BPW85A	1	0,62 €
	LM35-DZ	1	1,41 €
	HC-SR505	1	1,12 €
	10 kΩ 1/2W Resistor	1	0,06 €
Dimming Circuit	4N25 Optocoupler	1	0,53 €
	MOC3021 Optocoupler	1	0,56 €
	BT137-600 Triac	1	0,63 €
	1N5404 Diode	4	4 x 0,15 €
	Terminal Connectors	2	2 x 0,55 €
	120kΩ 1/2W Resistor	2	2 x 0,06 €
	1kΩ 1/2W Resistor	1	0,06 €
	470 Ω 1/2W Resistor	1	0,06 €
	10 kΩ 1/2W Resistor	1	0,06 €
Total			25,25 €

the main advantages of development this type of application with BLE is that it does not need a specific controller, any smartphone compatible with BLE or using an USB dongle can act as a controller. This will result in a smaller development cost without losing processing capacity.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The objective of this thesis was to develop a smart system, based on BLE communication, capable of sensing environment variable, control the illumination and to interact with a mobile application designed for it.

Based on the work of Cho et al [18], the objective was to embedded MCU, sensors and actuators all in one board. Two majors improvements were made: the inclusion of all sensors and BLE on the same circuit board and the development of an automatic illumination control.

The dimming control due to the limitations presented in Chapter 3 was implemented in a different MCU, which remove the high flickering existent when the RFD22301 module were handling the dimming process.

As exposed in the beginning of Sec.3.1 several issues needed correction in the first versions compatible with BLE. The mobile application for the SLABLE was tested in both Android version 4.3 and 5.0. The performance differences observed are relative to connection establishment delay and connection loss. For version 5.0 connection was established faster and almost no connection loss was detected, while in version 4.3 it was common to lose connection with the module after some time connected.

5.2 Future Work

For the future work to be done, there is two main paths to be followed:

- Extend connectivity of the device through cloud computing or other protocols
- Add new functionalities and integrate with other systems

As the technology evolves, along the development of this system new modules with BLE built-in were released that would allowed developing a system with cloud connectivity. The major differences comparing to the module used in this thesis are the possibility to have two or more communication protocols and better MCU. With a BLE and *WiFi* embedded module it would be possible connect to clouds. Cloud connectivity allow to introduce a new layer between the mobile application and the physical system, where processing and data storage can be implemented outside those.

Another improvement possible is to migrate from a one communication module system to a multi communication module through *Gazelle Link Layer* protocol. This would turn possible to create a wireless sensor network, allowing modules to turn into connected sensing nodes with the ability to trade data between them.

Initially, both ideas were planned to be implemented in this thesis, the first one could not be done because of the module limitation, while the second one for lack of time.

For the second path, there is already several home automation protocol as HVAC or DALI. Since the principle premise of IoT is to connect the larger number of device in the same network, a study could be made about the viability of integrating the implemented system into another existent home automation protocol.

Bibliography

[1] Kevin Ashton. *That 'Internet of Things' Thing*. 2009. URL: http://www.rfidjournal.com/articles/view?4986.

[2] M. Collotta and G. Pau. "A Novel Energy Management Approach for Smart Homes Using Bluetooth Low Energy". In: *IEEE Journal on Selected Areas in Communications* 33.12 (2015), pp. 2988–2996. ISSN: 0733-8716. DOI: 10.1109/JSAC.2015.2481203.

[3] M. Choi, W. K. Park, and I. Lee. "Smart office energy management system using bluetooth low energy based beacons and a mobile app". In: *Consumer Electronics (ICCE), 2015 IEEE International Conference on*. 2015, pp. 501–502. DOI: 10.1109/ICCE.2015.7066499.

[4] Virtenio GmbH. *Sensor board for the expansion board Preon32Shuttle Datasheet*. URL: http://www.virtenio.com/en/assets/downloads/datenblaetter/DS_VariSen_v14_2page%20[EN].pdf.

[5] Develco Products. *Motion Sensor*. Last Accessed: 26/03/2016. URL: http://www.develcoproducts.com/media/1677/motion-sensor-datasheet.pdf.

[6] Bluetooth SIG. *Bluetooth Specification 4.0*. 2010. URL: http://www.bluetooth.com.

[7] J. DeCuir. "Introducing Bluetooth Smart: Part 1: A look at both classic and new technologies." In: *IEEE Consumer Electronics Magazine* 3.1 (2014), pp. 12–18. ISSN: 2162-2248. DOI: 10.1109/MCE.2013.2284932.

[8] K. H. Chang. "Bluetooth: a viable solution for IoT? [Industry Perspectives]". In: *IEEE Wireless Communications* 21.6 (2014), pp. 6–7. ISSN: 1536-1284. DOI: 10.1109/MWC.2014.7000963.

[9] Akiba Kevin Townsend Carles Cufí and Robert Dadidson. *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking*. O'Reilly Media, 2013, pp. 35–51.

[10] *ISO/IEC 9834-8:2005*. ISO. 2005. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=36775.

[11] Mikhail Galeev. *Bluetooth 4.0: An introduction to Bluetooth Low Energy—Part II*. Last Accessed: 13/01/2016. 2011. URL: http://www.eetimes.com/document.asp?doc_id=1278966.

[12] Joe Decuir. *Bluetooth 4.0: Low Energy*. Cambridge Silicon Radio plc, 2010.

[13] Rob van Kranenburg. *The Internet of Things*. URL: http://www.theinternetofthings.eu/what-is-the-internet-of-things.

[14] L. Coetzee and J. Eksteen. "The Internet of Things - promise for the future? An introduction". In: *IST-Africa Conference Proceedings, 2011*. 2011, pp. 1–9.

[15] Rob van der Meulen. *Gartner Says 6.4 Billion Connected Things Will Be in Use in 2016, Up 30 Percent From 2015*. 2015. URL: http://www.gartner.com/newsroom/id/3165317.

[16] Nordic Semiconductors. *3rd Party Bluetooth Smart Modules*. Last Accessed: 15/02/2016. URL: http://www.nordicsemi.com/eng/Products/3rd-Party-Bluetooth-Smart-Modules.

[17] Liisa Halonen. *Guidebook on Energy Efficient Electric Lighting for Buildings*. Helsinki University of Technology, Lighting Laboratory. 2010.

[18] Y. S. Cho, J. Kwon, S. Choi, and D. H. Park. "Development of smart LED lighting system using multi-sensor module and bluetooth low energy technology". In: *Sensing, Communication, and Networking (SECON), 2014 Eleventh Annual IEEE International Conference on*. 2014, pp. 191–193. DOI: 10.1109/SAHCN.2014.6990353.

[19] R. A. Ramlee, M. A. Othman, M. H. Leong, M. M. Ismail, and S. S. S. Ranjit. "Smart home system using android application". In: *Information and Communication Technology (ICoICT), 2013 International Conference of*. 2013, pp. 277–280. DOI: 10.1109/ICoICT.2013.6574587.

[20] Argenox Technologies. *Android 5.0 Lollipop brings BLE Improvements*. Last Acessed: 13/03/2016. 2015. URL: http://www.argenox.com/blog/android-5-0-lollipop-brings-ble-improvements/.

[21] John Callaham. *Google says there are now 1.4 billion active Android devices worldwide*. Last Acessed: 13/03/2016. 2015. URL: http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide.

[22] RF Digital. *RFD22301 Datasheet*. Last Accessed: 13/01/2016. 2015. URL: http://www.rfdigital.com/wp-content/uploads/2015/08/RFD22301.Data.Sheet.08.20.15_4.36PM.pdf.

[23] RF Digital. *RFD22301 Programming Reference*. Last Accessed: 21/01/2016. 2015. URL: http://www.rfdigital.com/wp-content/uploads/2014/03/rfduino.ble_.programming.reference.pdf.

[24] Anne Van Rossum. *RFduino without RFduino code*. Last Accessed: 18/02/2016. URL: https://dobots.nl/2014/03/05/rfduino-without-rfduino-code/.

[25] The Engineering ToolBox. *Illuminance - Recommended Light Levels*. Last Accessed: 26/03/2016. URL: http://www.engineeringtoolbox.com/light-level-rooms-d_708.html.

[26] L. Lian and L. Li. "Wireless dimming system for LED Street lamp based on ZigBee and GPRS". In: *System Science, Engineering Design and Manufacturing Informatization (ICSEM), 2012 3rd International Conference on*. Vol. 2. 2012, pp. 100–102. DOI: 10.1109/ICSSEM.2012.6340818.

[27] Popescu Marian. *LDR = Light Dependent Resistor = Photoresistor*. Last Accessed: 22/01/2016. URL: http://www.electroschematics.com/6355/ldr-light-dependent-resistor-photoresistor/.

[28] Jacob Fraden. *Handbook of Modern Sensors: Physics, Designs, and Applications*. 4th Edition. Springer, 2010.

[29] Ian Poole. *Phototransistor Tutorial*. URL: http://www.radio-electronics.com/info/data/semicond/phototransistor/photo_transistor.php.

[30] Fairchild Semiconductor. *Application Note AN-3005 Design Fundamentals for Phototransistor Circuits*. URL: https://www.fairchildsemi.com/application-notes/AN/AN-3005.pdf.

[31] Vishay Semiconductors. *BPW85, BPW85A, BPW85B, BPW85C Datasheet*. URL: http://www.vishay.com/docs/81531/bpw85a.pdf.

[32] Classicharmony. *Light Meter for Android*. Last Accessed: 26/07/2016. URL: https://play.google.com/store/apps/details?id=com.classicharmony.lightmeter.

[33] Texas Instruments. *LM35 Precision Centigrade Temperature Sensors*. URL: http://www.ti.com/lit/ds/symlink/lm35.pdf.

[34] Elecrow. *HC-SR505 PIR Motion Sensor*. URL: http://www.elecrow.com/wiki/index.php?title=HC-SR505_Mini_PIR_Motion_Sensor.

[35] InMojo. *Digital AC Dimmer Module*. Last Accessed: 26/03/2016. URL: http://www.inmojo.com/store/inmojo-market/item/digital-ac-dimmer-module-lite-v.2/.

[36] Tindie. *AC 60Hz/50Hz Dimmer/SSR Controller Board*. Last Accessed: 26/03/2016. URL: https://www.tindie.com/products/thewp122/ac-60hz50hz-phase-controller-dimmer-board-arduino-compatible-2/.

[37] Vishay Semiconductors. *4N25, 4N26, 4N27, 4N28 Optocoupler, Phototransistor Output, with Base Connection*. URL: http://www.vishay.com/docs/83725/4n25.pdf.

[38] Nick de Smith. *ANSI IPC-2221A PCB Trace Width Calculator*. Last Accessed: 26/03/2016. URL: http://www.desmith.net/NMdS/Electronics/TraceWidth.html.

[39] Adafruit. *XBee Module - ZB Series 2*. Last Accessed: 26/03/2016. URL: https://www.adafruit.com/products/968.

[40] Adafruit. *XBee Pro Module - ZB Series 2*. Last Accessed: 26/03/2016. URL: https://www.adafruit.com/products/967.

APPENDIX A

APPENDIX A

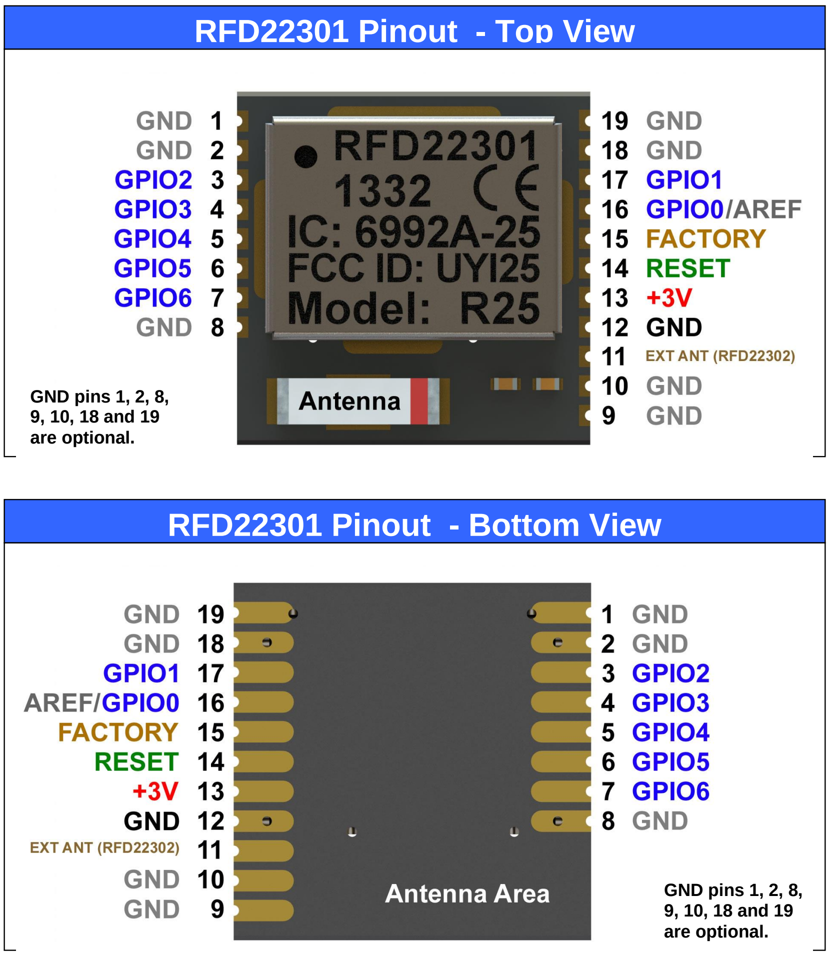


Figure A.1: RFD22301 Pinout

Appendix B

B.1 RFD22301 Code

```
#include <RFduinoBLE.h>
#include <wiring_analog.h>
#include <string.h>
#include <Wire.h>

//BLE related
int advertisement_interval = 2; //in seconds;
                                    length of advertisement cycle
int numberOfAdvertisements = 10; //number of advertisement per cycle
bool connection = false;
bool *pconnection= &connection;
bool advertisement = false;
bool *padvertisement = &advertisement;

//GPIO definition
const int pir = 2;
const int SensorTPin = 3;
const int SensorLPin = 4;
const int i2c_out = 5;
const int i2c_in = 6;

//Control
bool autoControl = true;
int dimming = 128;
bool presence = false;
bool *ppresence = &presence;


//Timer Variables
#define TRIGGER_INTERVAL 1000 //ms

int aux_timing = 30; //30 seconds

float temp;
int light;

int val = 5;

void setup() {

  Serial.begin(9600);
  RFduinoBLE.advertisementInterval=100; //ms
  //RFduinoBLE.txPowerLevel = -20;

  //Pin initialization
  pinMode(pir,INPUT);
  //pinMode(LEDDriver, OUTPUT);
  pinMode(SensorTPin, INPUT);
  pinMode(SensorLPin, INPUT);

  //Timer Initialization
  timer_config();

  Wire.begin( );
}

void loop() {

  if(digitalRead(pir)){
    *ppresence = true;
    aux_timing = 30;
    Serial.println("Movement Detected");
  } else{
    *ppresence = false;
  }
  //Wait for movement detection to start advertising
  if(*ppresence && !*padvertisement && !*pconnection){
    advertise(SECONDS(advertisement_interval));
    autoControlFunc(getLightSensorValue());
  }

  //Check if a connection is established
  if(*pconnection){
      aux_timing = 30;
      //Serial.println("connection");
      if(autoControl){
        autoControlFunc(getLightSensorValue());
        temp = getTemperatureSensorValue();
        light = getLightSensorValue();
      } else
          {
            temp = getTemperatureSensorValue();
            light = getLightSensorValue();
            delay(1000);
          }
      if(!aux_timing){
        dimming = 128;
        I2CSend();
      }
  //If no connection is established, only verify AutoControl
  } else{
      if(aux_timing){
        autoControlFunc(getLightSensorValue());
      } else{
        if(!aux_timing && !*ppresence){
          dimming = 128;
          I2CSend();
        }

      }

  }
}

void RFduinoBLE_onAdvertisement(bool start)
{
  *padvertisement = true;
  Serial.println("In onAdvertisement");
}

void RFduinoBLE_onConnect()
{
  *pconnection=true;
  *padvertisement = false;
  Serial.println("In onConnect");
```

```
}

void RFduinoBLE_onDisconnect()
{
  *pconnection=false;
  RFduinoBLE.end();
  Serial.println("Disconnected");
}

void advertise(int interval){
  RFduinoBLE.begin();
  while ((!RFduinoBLE.radioActive)||(*pconnection));
  while((! *padvertisement)||(*pconnection));
  *padvertisement = false;
  //time after impulse
  //(to let the central the time to request a connection)
  delay(20);
  for(int i=0 ; (i<(numberOfAdvertisements-1)); i++){
   if(! *pconnection){RFduino_ULPDelay(100-(15+0*10+5));}
   if(! *pconnection){
       while ((!RFduinoBLE.radioActive)||(*pconnection));}
   if(! *pconnection){delay(20);}
  }
  if(! *pconnection){
    RFduinoBLE.end();
    RFduino_ULPDelay(SECONDS(advertisement_interval));
  }
}

void RFduinoBLE_onReceive(char *data, int len){

  int i;
  int val;

  Serial.print(data);
  //Set Automatic Control
  if(data[0] == 'A'){
    actuatorsAction(data[0],0);
  }
  //Set Manual Control
  if(data[0] == 'M'){
    byte aux[3] = {data[1],data[2],data[3]};
    int *lval = (int *) aux;
    actuatorsAction('M',*lval);
    //Serial.println(*lval);
  }
  //Refresh Data
  if(data[0] == 'R'){
    light = getLightSensorValue();
    temp = getTemperatureSensorValue();
  }
}

//Timer configuration
void timer_config(void){

  NRF_TIMER1->TASKS_STOP = 1; //Stop timer
  //Set the timer to Counter Mode
  NRF_TIMER1->MODE = TIMER_MODE_MODE_Timer;
  NRF_TIMER1->BITMODE =
        (TIMER_BITMODE_BITMODE_16Bit << TIMER_BITMODE_BITMODE_Pos);
  NRF_TIMER1->PRESCALER = 9; //32us resolution
  NRF_TIMER1->TASKS_CLEAR = 1; // Clear timer
  // converting from 32us to 1ms
  NRF_TIMER1->CC[0] = (TRIGGER_INTERVAL*31) + (TRIGGER_INTERVAL/4);
  NRF_TIMER1->INTENSET =
      TIMER_INTENSET_COMPARE0_Enabled << TIMER_INTENSET_COMPARE0_Pos;
  NRF_TIMER1->SHORTS =
      (TIMER_SHORTS_COMPARE0_CLEAR_Enabled
                               << TIMER_SHORTS_COMPARE0_CLEAR_Pos);
  attachInterrupt(TIMER1_IRQn,TIMER1_Interrupt);
  NRF_TIMER1->TASKS_START = 1; // Start timer;
}

void TIMER1_Interrupt(void){

  //Timer @1 sec
  if(NRF_TIMER1->EVENTS_COMPARE[0] != 0){
    //Serial.println("Timer interrupt called");
    NRF_TIMER1->EVENTS_COMPARE[0] = 0;
    if(aux_timing>0){
      aux_timing--;
      Serial.println(aux_timing);
    }
  }
}

float getTemperatureSensorValue(){

  int rawvoltage= analogRead(SensorTPin);
  float millivolts= (rawvoltage/1024.0) * 5000;
  float celsius= millivolts/10;

  // send temperature to connected BLE device
  RFduinoBLE.sendFloat(celsius);

  return celsius;

}

int getLightSensorValue(){

  int light;
  String L = "L";
  String aux_light;
  char send_light[5];

  light = analogRead(SensorLPin);
  if(light<100){
    aux_light = L + light + ' ';
  } else{
    aux_light = L + light;
  }

  aux_light.toCharArray(send_light,sizeof(send_light)+1);
  RFduinoBLE.send(send_light,sizeof(send_light));

  return light;

}

void actuatorsAction(char c,int v){

  if(c == 'A'){
    autoControl = true;
  }

  if(c == 'M'){
    autoControl = false;
    //analogWrite(LEDDriver,v);
    //Serial.println(v);
    dimming = v;
    //Serial.println(dimming);
    I2CSend();

  }
}

void autoControlFunc(int lightValue){

  int ControlValue;
  if(lightValue < 500 && dimming > 5){
    ControlValue = dimming - 4;
    dimming = ControlValue;
  }
  if(lightValue > 500 && dimming < 120){
    ControlValue = dimming + 4;
    dimming = ControlValue;
  }
  Serial.println(dimming);
  I2CSend();
}

// Send data to slave
void I2CSend()
{
  Wire.beginTransmission(4); // transmit to device #4
  Wire.write(dimming);              // sends one byte
  Wire.endTransmission();    // stop transmitting

  delay(500);
}
```

B.2 Arduino Mega Code

```
#include <avr/interrupt.h>

const int AC_LOAD = 4;    // Output to Opto Triac pin
const int Dimmer = 1; // Dimming Value Received from RFD22301
int dimming = 128;  // Dimming level (0-128)  0 = ON, 128 = OFF

void setup()
{
  pinMode(AC_LOAD, OUTPUT);// Set AC Load pin as output
  pinMode(Dimmer, INPUT);// Set dimming vales as input
  attachInterrupt(0, zero_crosss_int, RISING);
}

//function to be fired at the zero crossing to dim the light
```

```
void zero_crosss_int()
{
  // Firing angle calculation : 1 full 50Hz wave =1/50=20ms
  // Every zerocrossing thus: (50Hz)-> 10ms (1/2 Cycle)
  // 10ms=10000us
  // (10000us - 10us) / 128 = 75 (Approx)

  int dimtime = (75*dimming);
  delayMicroseconds(dimtime);    // Wait till firing the TRIAC
  digitalWrite(AC_LOAD, HIGH);   // Fire the TRIAC
  delayMicroseconds(10);         // triac On propogation delay
  // No longer trigger the TRIAC
  digitalWrite(AC_LOAD, LOW);
}

void loop()  {

  dimming = analogRead(Dimmer);

}
```

2016 **Development of a Smart Lighting Android-based Application using Bluetooth Low Energy** Daniel Batista