DISSERTATION

Master in Electrical and Electronic Engineering

Hardware/Software interface for enhanced remote control of Android set-top-box

Ricardo Alexandre Caseiro dos Santos

Leiria, September of 2015

DISSERTATION

Master in Electrical and Electronic Engineering

Hardware/Software interface for enhanced remote control of Android set-top-box

Ricardo Alexandre Caseiro dos Santos

Dissertation developed under the supervision of Doctor Pedro António Amado de Assunção, professor at the School of Technology and Management of the Polytechnic Institute of Leiria and co-supervision of Master Luís Manuel Conde Bento, professor at the School of Technology and Management of the Polytechnic Institute of Leiria.

Leiria, September of 2015

Some men see things as they are and ask why? I dream things that never were and ask why not?

(George Bernard Shaw)

This page was intentionally left blank.

Acknowledgements

The limited space of this acknowledgement section, of course does not allow me to thank as I should. To all the people that throughout my time at Polytechnic Institute of Leiria supported me directly and indirectly, to achieve my goals, improve as a human being and achieve this stage of my academic education. Thus, I leave some words, few but with a deep sense and feeling of thanks.

To my supervisor Dr. Pedro Assunção, for the valuable guidance, time spent on helping me and opportunities given, that contribute to increase my knowledge. To my co-supervisor, Pr. Luís Bento, for the cooperation in solving problems and questions that have emerged and for all the words of encouragement. To Dr. Hugo Costelha not only for the help and great guidance in this project, as well as in the Robotics Club at the ESTG, which undoubtedly contributed to my professional development. My sincere gratitude to have awakened my desire to want, always learn more and incessant will to do better.

To the Tech4Home for providing this excellent opportunity to work, evolve and contribute to this project, without whom this would not have been possible.

To Instituto de Telecomunicações (IT) - delegação de Leiria, on the scientific orientation and equipment facilitated.

To my friend and co-worker Miguel Rasteiro, for all the support, creating an excellent working environment and the distracting moments outside the work.

To all my colleagues of Electrical and Electronic Engineering, IEEE IPLeiria Student Branch, CEEE and Robotics Club. A special thanks to André Guarda and João Santos for all the support and friendship.

To my family, especially my parents and brother, for all their unconditional support, love and patience throughout this time.

Finally but not the least, a special thanks to Cristiana for all the love and patience, especially given that this work did not let us be together as much time as we would like.

Ricardo Santos

This work is co-financed by European Union, COMPETE, QREN and Fundo Europeu de Desenvolvimento Regional (FEDER), Project HERMES, co-promoção n.º 34149.

This page was intentionally left blank.

Resumo

Esta dissertação descreve a pesquisa e o desenvolvimento de um sistema de comunicação para suportar dispositivos de controlo remoto (RCD) para set-top-boxes (STB) com o sistema operativo Android. O RCD alvo é um dispositivo de baixa complexidade, que captura os movimentos 3D para fornecer novas funcionalidades interativas para diferentes tipos de conteúdos e aplicações multimédia.

A arquitetura do sistema consiste num RCD com sensores Magnéticos, Gravíticos e de Velocidade Angular (MARG) para obtenção do movimento 3D, que transmite os dados para uma STB Android. A comunicação entre o RCD e a STB foi implementada através do protocolo de rádio frequência para eletrónica de consumo (RF4CE), o que exigiu o desenvolvimento de um módulo externo para a *Set-Top-Box*. Foi desenvolvida uma Interface de Programação de Aplicações (API) para permitir o processamento dos dados do controlo remoto e a criação de seis perfis no Android: rato absoluto, rato relativo, multi-toque, acelerómetro, giroscópio e magnetómetro. Para os sensores da unidade MARG no dispositivo de controlo remoto serem reconhecidos nativamente no Android OS, foi também desenvolvida uma biblioteca em Android, que lê os valores dos sensores através da API. A demonstração das funcionalidades do sistema foi feita através de uma aplicação Android, desenvolvida especificamente para simular e testar o ambiente de uma potencial utilização.

Foi também efetuado, um estudo para descobrir se as funções mais complexas devem ser executadas no RCD ou na STB Android. A solução ótima ainda permanece uma questão em aberto, uma vez que depende dos requisitos da aplicação e da portabilidade tendo em conta o consumo de energia. A análise do consumo de energia no RCD mostra que a transmissão dos dados dados em bruto, para serem processados na API, resulta num menor consumo de energia em geral, e consequentemente, numa maior portabilidade com boa precisão. Uma vez que a STB não tem limitações sobre o consumo de energia e tem um poder computacional superior, a API foi projetada para ser capaz de realizar todo o processamento de dados dos sensores, permitindo assim, a implementação de algoritmos de fusão complexos e com maior precisão.

Palavras-chave: Android, API, Set-Top-Box, USB, HID, Sensores

This page was intentionally left blank.

Abstract

This dissertation describes the research and development of a communication system to support remote control devices (RCD) for Android-based set-to-box (STB). The target RCD is a low-complexity device using 3D motion tracking to provide new interactive functionalities to different types of multimedia content and applications.

The system architecture comprises an RCD with Magnetic, Angular Rate, Gravity (MARG) unit for 3D motion tracking, transmitting data to an Android STB. The communication between the RCD and the STB was implemented using the Radio Frequency for Consumer Electronics (RF4CE) protocol, which required the development of an external module for the STB. An Application Programming Interface (API) was developed to enable seamless computation of the remote control data and allowing six input profiles on the Android: Absolute air mouse, Relative air mouse, multitouch, accelerometer, gyroscope and magnetometer. To allow the sensors from the MARG unit in the remote control device to be natively recognized on the Android OS, an Android sensors library was also developed, this reads the sensors data from the API. The demonstration of the system functionalities was done through an Android application specifically developed to simulate and test a potential usage environment.

A study to find out whether the most complex functions should run on the RCD or on the Android STB was also carried out. The optimal solution still remains an open issue since it depends on the specific application and portability requirements taking into account energy consumption. The analysis of energy consumption on the RCD shows that transmitting the raw data from the sensors to be processed in the API, results in a lower energy consumption, and consequently higher portability with good accuracy. Since the STB has no limitations on energy consumption and superior computational power, the API was designed to be able to perform all the processing of sensors data, thus allowing the implementation of complex fusion algorithms with higher precision.

Keywords: Android, API, Set-Top-Box, USB, HID, Sensors

This page was intentionally left blank.

List of Figures

2.1 IEEE 802.15.4 stack. 9
2.2 IEEE 802.15.4 physical layer packet. 9
2.3 IEEE 802.15.4 network topologies. 10
2.4 ZigBee® RF4CE stack. 11
2.5 ZigBee™ RF4CE topology. 12
2.6 MiWi™ P2P stack. 13
2.7 Handshaking process. 14
2.8 Compatibility between Bluetooth versions. 15
2.9 BLE stack. 15
2.10 Bluetooth Low Energy channels. 16

3.1 USB 3.x Cable. 18
3.2 USB architecture. 19
3.3 Complex Devices. 20
3.4 Configuration of speeds 21
3.5 Chirp handshake. 21
3.6 USB protocol stack. 23
3.7 USB logical connections. 24
3.8 USB transfers. 25
3.9 USB descriptors. 26
3.10 USB HID Descriptors. 27
3.11 Android stack with details. 31
3.12 Android application anatomy. 32
3.13 Android boot sequence. 34
3.14 Android USB stack. 35
3.15 East-North-Up coordinates. 37

3.16 Android sensors stack. 37

4.1 System developed. 39
4.2 Remote Control Device prototype. 40
4.3 System architecture. 41
4.4 Final scheme of PCB dongle. 42
4.5 PCB dongle. 43
4.6 Dongle flowchart. 44
4.7 Radio Frequency modules. 45
4.8 USB HID Custom - Demo Input Report format 46
4.9 Device - HID - Custom Demos directory tree. 47
4.10 API flowchart. 49
4.11 Android API Stack. 50
4.12 Android Sensors UML Structure. 54
4.13 Android Sensor Service. 55
4.14 Main Menu. 56
4.15 3D Visualizer. 56
4.16 TV Simulation. 56
4.17 Logger. 56
4.18 USB Block Diagram. 58
4.19 Measurement system. 61
4.20 Results of energy consumption - Sensors. 66
4.21 Results of energy consumption - Reading Sensors. 67
4.22 Results of energy consumption - Fusion Filters. 68
4.23 Results of energy consumption - Send with MRF24J40MA. 69
4.24 Results of energy consumption - MRF24J40MA. 70

List of Tables

2.1 Family of IEEE 802 standards. 8
2.2 Categories of IEEE 802.15 standard. 8

3.1 USB speeds 18
3.2 Digital states of Chirp k and j in data lines. 22

4.1 Calculated value of the variables for each module. 62
4.2 Value of the variables used in each module. 63
4.3 Maximum input voltage value in the amplifier from modules. 63
4.4 Characterization of energy consumption - Sensors 64
4.5 Characterization of energy consumption - Data processing 64
4.6 Characterization of energy consumption - Module RF 64
4.7 Results of energy consumption - Sensors 65
4.8 Results of energy consumption - Data processing 65
4.9 Results of energy consumption - Module RF 65
4.10 Remote control device setups and results. 71

This page was intentionally left blank.

List of Acronyms

ADC	Analog-to-Digital Converter
APK	Android application Package
AES	Advanced Encryption Standard
AOSP	Android Open Source Project
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
BLE	Bluetooth® low energy
BPSK	Binary Phase Shift Keying
CE	Consumer Eletronics
CEC	Consumer Electronics Control
DoF	Degrees of Freedom
DMP	Digital Motion Processor
DSSS	Direct Sequence Spread Spectrum
FHSS	Frequency Hopping Spread Spectrum
GCC	GNU Compiler Collection
GFSK	Gaussian Frequency Shift Keying
HAL	Hardware Abstraction Layer
HCI	Host Controller Interface
HOGP	HID Over GATT Profile

HS	High Speed
HDMI	High-Definition Multimedia Interface
HID	Human Interface Device
I2C	Inter-Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
ISM	Industrial, Scientific and Medical
LAN	Local Area Network
LED	Light Emitting Diode
LSB	Least Significant Bit
MAC	Medium Access Control
MAN	Metropolitan Area Network
MARG	Magnetic, Angular Rate, Gravity
MCU	Microcontroller Unit
MiWi	Microchip Wireless Protocol
MLA	Microchip Libraries for Applications
NDK	Native Development Kit
NRZI	Non Return to Zero Inverted
O-QPSK	Offset Quadrature Phase-Shift Keying
OHA	Open Handset Alliance
OpenGL	Open Graphics Library
OS	Operating System
OSI	Open Systems Interconnection
P2P	Peer to Peer
PAN	Personal Area Network

PCB	Printed Circuit Board
QoE	Quality of Experience
RCD	Remote Control Device
RAM	Random Access Memory
RF	Radio Frequency
RF4CE	Radio Frequency for Consumer Electronics
ROM	Read Only Memory
SIE	Serial Interface Engine
SIG	Special Interest Group
SoC	System on Chip
SPI	Serial Peripheral Interface
STB	Set-Top-Box
UART	Universal Asynchronous Receiver/Transmitter
UML	Unified Modeling Language
UUID	Universally Unique IDentifier
USB	Universal Serial Bus
VM	Virtual Machine
ZID	ZigBee® Input Device
ZRC	ZigBee® Remote Control

This page was intentionally left blank.

Contents

Acknowledgements **v**

Resumo **vii**

Abstract **ix**

List of Figures **xi**

List of Tables **xiii**

List of Acronyms **xv**

1 Introduction **1**

1.1 Goals 2

1.2 Related Work 3

1.3 Publications 4

1.4 Dissertation structure 5

2 Wireless interfaces for consumer electronics **7**

2.1 Family of IEEE 802 standards 8

2.1.1 IEEE Standard 802.15.4™ - Low Rate WPAN 9

2.1.2 ZigBee® Radio Frequency for Consumer Electronics (RF4CE) . . . 11

2.1.3 Microchip Wireless Protocol (MiWi™) P2P 13

2.2 Bluetooth® Low Energy (BLE) 14

3 The Universal Serial Bus (USB) in Android **17**

3.1 Universal Serial Bus (USB) 2.0 17

3.1.1 Recent advances of Universal Serial Bus (USB) 18

3.1.2 Architecture 18

3.1.3 Power supply . . . 22
3.1.4 Communication . . . 23
3.1.5 Types of transfers . . . 24
3.1.6 Descriptors . . . 26
3.1.7 Human Interface Devices (HID) . . . 27
3.2 Android Architecture . . . 29
3.2.1 Software Stack . . . 30
3.2.2 Boot sequence . . . 34
3.2.3 USB architecture . . . 35
3.2.4 Sensor stack . . . 36

4 Development of Android interfaces for enhanced remote control 39
4.1 Dongle: transceiver RF-USB . . . 42
4.1.1 Communication between RCD and dongle . . . 45
4.1.2 Communication between dongle and STB . . . 46
4.2 Android API . . . 47
4.2.1 Application programming interface . . . 48
4.2.2 Android sensors library . . . 53
4.2.3 User interface application . . . 56
4.3 Energy consumption analysis on remote control . . . 61
4.3.1 Test conditions and characterization . . . 63
4.3.2 Experimental Results . . . 65
4.3.3 Analysis of results . . . 71

5 Conclusions and Future Work 73

Bibliography 77

Appendix A Android API development tutorial 83
A.1 Build steps in Linux . . . 83
A.1.1 Recomended requirements . . . 83
A.1.2 Android open source build . . . 83
A.1.3 Android kernel build . . . 85
A.1.4 Android native C program . . . 87
A.1.5 Android sensors library compilation . . . 88

Chapter 1

Introduction

In the past years there has been a strong investment in technology development for television and multimedia consumer market in general. Besides the evolution of screen resolutions, there has been an evolution that is bringing new types of multimedia content. Before this evolution, the user had a limited interaction with the available content in the television, but the trend is to have more interactive multimedia content and applications. However the devices used for interaction did not follow this evolution, leading to a poor Quality of Experience (QoE) [1].

There is also a trend for portable technology, leading to smaller devices with the same or even higher computational power. This decrease in the size of devices increases their portability, but also implies smaller battery, requiring efficiency in wireless communications and computational power in order to achieve reduced energy consumption [2].

Today, there are a set of solutions available in the market for interactivity with multimedia systems. The Remote Control Device (RCD) of a Set-Top-Box (STB) or television is used for interaction with multimedia content, mainly based on two dimensions (2D) [3]. The evolution to 3D content and operation with added interactive functionalities, requires the mapping of 3D movements into motion in 2D screen [4].

To reduce integration barriers, the main manufacturers are moving towards Android-based systems. This operating system has increasingly been adopted for multimedia content both on television[1], and on the STB[1] [5]. Since Android is an open system, it allows an increased knowledge of its architecture, enabling faster implementation of Application Programming Interfaces (API).

In the scope of this work, a system for 3D interaction with multimedia content was developed and tested. This system is divided into three functional modules: communication, processing and application layer.

[1] `https://www.android.com/tv/` (visited on 29 August 2015)

A proper computational balance between the RCD and the STB is important, *i.e.* one has to decide whether the most complex functions (in terms of computational complexity) should run on the RCD or on the STB, taking into account energy consumption. On the one hand, running complex algorithms on the RCD results in higher energy consumption when computing orientation estimates, and low energy consumption in communications due to less data being transmitted. On the other hand, transmitting raw data to the STB increases the energy required for communications, but allows the implementation of more complex algorithms on the STB, thus leading to more accurate estimates.

The Application Programming Interface (API) developed in this work was designed to have the least possible impact on the Operating System (OS). It receives data from RCD and makes it available to the OS after the computation process. The implementation on the STB side, also allows access to information about the user system (e.g., available resources).

This work also included the challenge of implementing a Human Interface Device (HID) and Sensors interfaces for transparent communication between a remote control, with the ability to send 3D location data (relative position and absolute orientation), and multimedia applications for Android environment. A demo application was also implemented.

1.1 Goals

The goals of this dissertation were defined as follows:

- Study of wireless communications protocols for multimedia consumer equipment.
- Characterization of the Universal Serial Bus (USB) interfaces.
- Development of the hardware and firmware to receive data from RCD through wireless communication and redirects them through USB dongle to the STB.
- Study of the Android architecture to understand the implementation of sensors.
- Development of the library in Android to recognize the sensors in RCD as native.
- Study and implementation of the API in Android OS for computational process on the STB.
- Development of an Android application for demo purposes that receives 3D motion data from RCD through USB HID custom to control an 3D object.
- Development of the hardware and firmware for measuring the energy consumption on the RCD.
- Analysis of the results obtained from energy consumption measurements on the RCD

1.2 Related Work

The Android OS, has native support for embedded sensors, for all types of devices. Although there are not as many alternatives for the use of external sensors in this OS. In this study, the sensors will be placed in remote control device, while the Android OS will be running in the set-top-box. Therefore the sensors are considered external, i.e. the sensors are not directly embedded in the Android device, which means that at any given moment they can be connected or disconnected. This section presents a review of relevant work related with techniques to make the external sensors recognized by the Android OS. The following techniques were considered the most relevant.

Amarino [6]

The Amarino is a toolkit comprising an Android application and a library called "MeetAndroid" for Arduino. This setup establish a communication between both devices through Bluetooth, enabling the exchange of data between devices and making it easier to receive data from sensors connected to Arduino. Although this toolkit speed up the integration of Arduino sensors, this technique has some limitations, it requires an application specifically developed for this purpose and an Arduino with Bluetooth communication.

Open Intents - SensorSimulator [7]

The SensorSimulator was created to surpass the limitation of not having sensors data in the Android emulator. This Simulator also allows recording sensors data from a real Android device with embedded sensors. These data can then be later sent over a socket connection to the Android application to simulate the sensors. The use of SensorSimulator involves a specific library and a socket connection to receive data in Android, this means that with SensorSimulator, one has to include an application with complex data communications protocol (TCP/IP).

Sensor Emulation initiative for virtualized Android-x86 [8]

The Sensor Emulation is a system designed to emulate the sensors in virtualized Android-x86 environment. This emulation is done at system-level to avoid the need for changes in the Android application. This system comprises a server on the real Android device to send the sensors data, an userspace "C" program in the host (e.g. Ubuntu) to forward the data from the real to virtual device and emulator server on the virtual Android device to receive the sensors data and make it available to Android OS. This implementation was designed for virtualized Android where it requires an socket-communication and an "C" program in the host machine to map the data from the real to the virtual device.

Open Data Kit Sensors [9]

The Open Data Kit Sensors is a high level framework used in the Open Data Kit to simplify the process of integrating sensors in the Android OS. This framework collects data from internal/external sensors and makes them available through an Android service in background through inter-application communication. The external sensors can be connected with Android device through USB or Bluetooth. Although it enables the use of external sensors, it implies the integration of the Open Data Kit Sensors framework in the developed Application.

Almost all the techniques found during the research of the related work implements a sensor simulation instead of sensor emulation. The main difference between the two is that the emulation does not require any changes to the Android application, unlike the simulation where it is necessary to use an additional library for accessing the simulated sensors. The technique that best suits the desired goal of this study is the "Sensor Emulation initiative for virtualized Android-x86" that avoid the changes in the Android application, while allows to receive sensors data from an external source.

1.3 Publications

The following publications were produced during the development of this work:

- R. Santos, M. Rasteiro, H. Costelha, L. Bento, P. Assuncao and M. Barata, "Motion-based Remote Control Device for Enhanced Interaction with 3D Multimedia Content", in Conference on Telecommunications (Conftele 2015), 17-18 September, 2015, Aveiro, Portugal

- R. Santos, H. Costelha, L. Bento, P. Assuncao and M. Barata, "Enabling low-complexity devices for interaction with 3D mediacontent via Android API", in Conference on Sciences and Technologies of Interaction (SciTecIN'15), 11-13 November, 2015, Coimbra, Portugal

1.4 Dissertation structure

This dissertation is organized in five chapters. This first chapter addresses an overall description of the project, objectives and related work. The second chapter includes an overview of the most relevant state-of-the-art communications for Consumer Eletronics (CE) devices; firstly the main characteristics of the wireless communications based on the IEEE 802 standards are presented; secondly the Bluetooth® low energy (BLE) technology is described and then the current status of USB is also presented. The third chapter contains the essential background related to the topics addressed in this work: USB 2.0 and the Android OS architecture. The fourth chapter is divided into three sections about the developed work; starts by explaining the role of the dongle in the whole system comprising the RCD and STB; then the second section presents the API developed to receive data from dongle and the application for demonstration purpose; the last section characterizes the experimental evaluation tests and analyse the results of energy consumption. Finally, in fifth chapter, some conclusions are presented, as well as some suggestions for future work.

This page was intentionally left blank.

Chapter 2

Wireless interfaces for consumer electronics

Advances in consumer electronics technology have introduced a new era of portable devices towards Internet of Things (IoT), where more and more devices communicate with each other. Typically, these devices tend to have small sizes, which have a critical impact on the battery. In these cases, where low complexity devices are required, the main constrain in wireless communications is to reduce the power consumption of the Radio Frequency (RF) interfaces [10]. On the other hand, wired communication normally connects devices that are attached to the power lines, thus the increase of communication speed is more important than power consumption. The main objective of this chapter is to describe the state of the art related to digital communication technology for consumer electronics. Starting with an overview of the family of Institute of Electrical and Electronics Engineers (IEEE) 802 Standards, which covers the Microchip Wireless Protocol (MiWi) and ZigBee® Radio Frequency for Consumer Electronics (RF4CE) communications, followed by the review of BLE, then the last advances of USB are covered. This is one of the most used architectures in wired communication. Other technologies like Z-Wave[1], Thread[2] and WiFi[3] were not reviewed since they do not fit in the context of this work.

---

[1] `http://www.z-wave.com` (visited on 29 August 2015)
[2] `http://threadgroup.org` (visited on 29 August 2015)
[3] `http://www.wi-fi.org` (visited on 29 August 2015)

2.1 Family of IEEE 802 standards

The IEEE is the world's largest association of technical professionals with the aim in "Advancing Technology for Humanity"[1] and is one of the leading in creation of standards.

The IEEE 802 standards are intended to define the specifications for Local Area Network (LAN), Metropolitan Area Network (MAN). In the Table 2.1 is presented the groups in this family, where in this dissertation the main focus was the IEEE 802.15 standard.

Table 2.1: Family of IEEE 802 standards[2].

Name	Description
IEEE 802.1	Bridging & Management
IEEE 802.2	Logical Link Control
IEEE 802.3	Ethernet
IEEE 802.11	Wireless LANs
IEEE 802.15	Wireless PANs
IEEE 802.16	Broadband Wireless MANs
IEEE 802.17	Resilient Packet Rings
IEEE 802.19	TV White Space Coexistence Methods
IEEE 802.20	Mobile Broadband Wireless Access
IEEE 802.21	Media Independent Handover Services
IEEE 802.22	Wireless Regional Area Networks

The IEEE 802.15 standard defines the categories presented in Table 2.2. The IEEE 802.15.4™, focused on low rate wireless Personal Area Network (PAN), is the most relevant in the context of this dissertation. This standard will be presented in the following sub section as a basis for the MiWi and Zigbee RF4CE communications. It is also important to refer that the IEEE standardized Bluetooth as IEEE 802.15.1 in 2002, but no longer maintains the standard [11]. Nowadays the Bluetooth is maintained by the Bluetooth Special Interest Group (SIG), that will be explained in a later section.

Table 2.2: Categories of IEEE 802.15 standard[3].

Name	Description
IEEE 802.15.1	WPAN
IEEE 802.15.2	Coexistence
IEEE 802.15.3	High Rate WPAN
IEEE 802.15.4	Low Rate WPAN
IEEE 802.15.5	Mesh Networking
IEEE 802.15.6	Wireless Body Area Networks
IEEE 802.15.7	Visible Light Communication

[1] `http://www.ieee.org/about/tagline.html` (visited on 29 August 2015)
[2] From: `https://standards.ieee.org/about/get/802/802.html` (visited on 29 August 2015)
[3] From: `https://standards.ieee.org/about/get/802/802.15.html` (visited on 29 August 2015)

2.1.1 IEEE Standard 802.15.4™ - Low Rate WPAN

The IEEE 802.15.4™ is a standard that specifies the Physical and Medium Access Control (MAC) layer taking into acount the Open Systems Interconnection (OSI) model as reference (Figure 2.1).

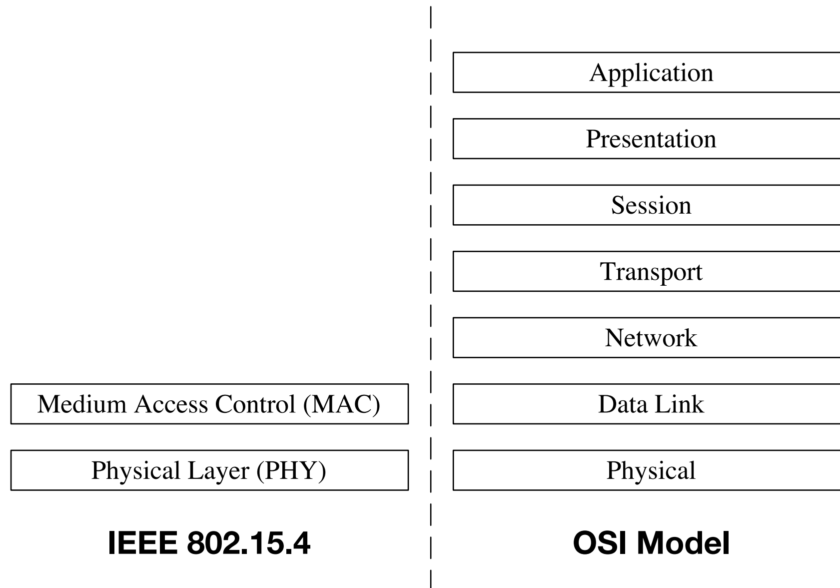


Figure 2.1: IEEE 802.15.4™ stack.

The Physical layer (PHY) is in charge of RF communications and the IEEE 802.15.4™ specification defines the 868 MHz (Europe), 915 MHz (North America) and 2.4 GHz Industrial, Scientific and Medical (ISM) (worldwide) license-free bands for operation with 1, 10 and 16 available channels and a theoretically data rate of 20, 40 and 250 kbps respectively. The 868 MHz and 915 MHz bands use the Binary Phase Shift Keying (BPSK) modulation, while the 2.4 GHz ISM band implements the Offset Quadrature Phase-Shift Keying (O-QPSK) modulation [12], [13]. In order to improve the protection against interferences the Direct Sequence Spread Spectrum (DSSS) technique is used. The maximum packet size is 133 bytes that are divided in Preamble, start-of-frame delimiter, frame length and PHY service data unit as shown in Figure 2.2.

Octets:	4	1	1	<= 127
Field:	Preamble	start-of-frame delimiter	frame length	PHY service data unit

Figure 2.2: IEEE 802.15.4™ physical layer packet [1].

[1] From: `http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf` (visited on 7 August 2015)

Medium Access Control (MAC) handles all access to the physical layer and provides two services. The data service that provides the reception and transmission of data through physical layer and the management service that performs the network management [13].

IEEE Standard 802.15.4™ Network topologies

The IEEE Standard 802.15.4™ network is composed by two types of devices, Full Function Device and Reduced Function Device. Depending on the role of the device in the network it can be classified as PAN Coordinator, Coordinator or End device [14].

Full Function Device (FFD) is a device with full implementation of the protocol and can act as coordinator or an end device.

Reduced Function Device (RFD) only the essential routines of the protocol were implemented and can only act as an end device.

PAN Coordinator is the principal controller of a PAN and needs to be a FFD.

Coordinator has the capability to extend the physical range of the network and needs to be a FFD.

End device can be a RFD or FFD, and normally are sensor nodes that provide information to the network.

These devices can operate in a Star or Peer to Peer (P2P) topology (Figure 2.3), where each can have only one PAN coordinator at a time. The main difference between the star and P2P topology is that in the star topology all devices can only communicate with the PAN Coordinator, while in the P2P they can only communicate between them [13],[15].

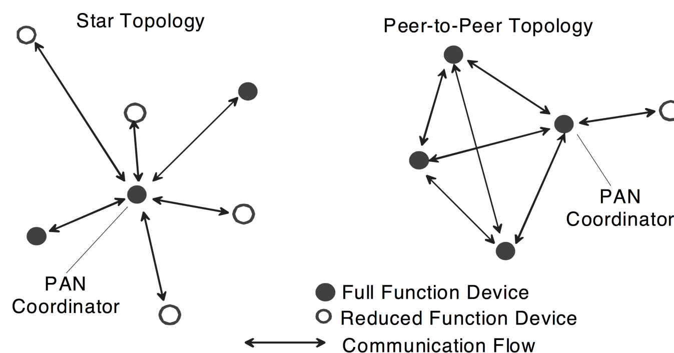


Figure 2.3: IEEE 802.15.4™ network topologies[1].

[1] From: `http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf` (visited on 7 August 2015)

2.1.2 ZigBee® Radio Frequency for Consumer Electronics (RF4CE)

The ZigBee® RF4CE is a standard that defines the wireless communication network for home entertainment equipment and remote control devices in CE domain to allow the multi-vendor interoperability between them. This standard also aims to be low cost, low latency and robust against interference. The Zigbee® RF4CE protocol stack (Figure 2.4), have the lower layers, PHY and MAC, defined by the IEEE 802.15.4™ standard and defines the network layer and the standard application profiles on top of these [16], [17], [18].

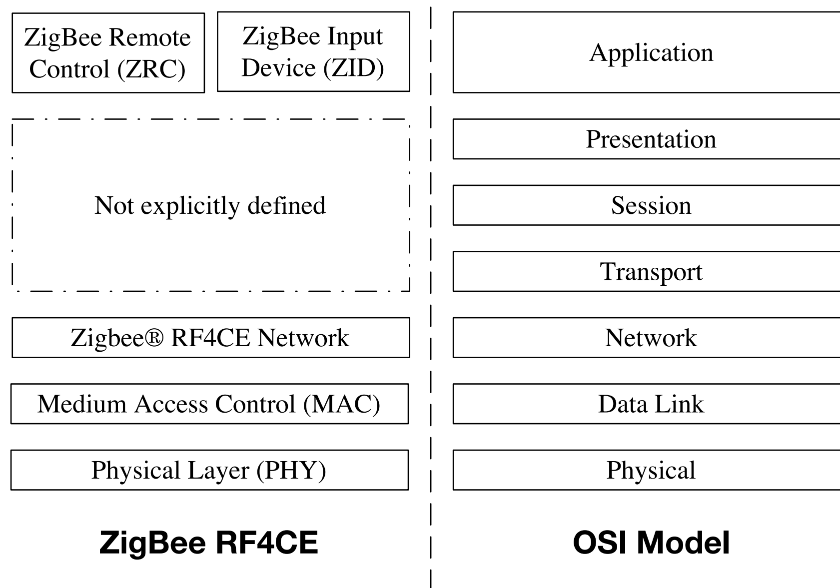


Figure 2.4: ZigBee® RF4CE stack[1].

The Physical Layer in the ZigBee® RF4CE operates in the 2.4GHz ISM band. This band can be overcrowded and to avoid interferences, besides the use of DSSS technique defined by IEEE Standard 802.15.4™, the ZigBee® RF4CE uses only the channels 15, 20 and 25, from the 16 channels availables. The channels 15 and 20, fall in the gaps between the 802.11 (Wi-Fi) channels 1, 6 and 11 [13], [19]. The ZigBee® RF4CE Network Layer controls the communication between devices, providing the ability to discover and connect new devices with a secure communication, managing the channels (frequency agility) and provides power saving mechanisms. The network is implemented in a full LAN capability through multiple connections PAN in star topology with two node types [20].

Target device have the ability to create their own network.

Controller device can only join to the networks created by the target device.

[1] Adapted from: Radio Frequency for Consumer Electronics (RF4CE) Protocol Overview, Mindteck

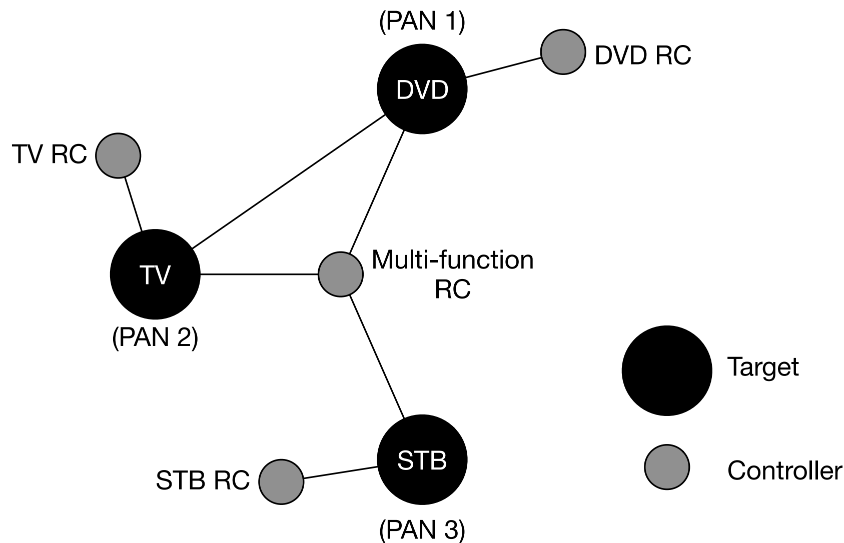


Figure 2.5: ZigBee™ RF4CE topology[1].

The topology is presented in Figure 2.5 and comprises three PANs created by each of the targets, three simple remote controllers that are connected to respective target and a multi-function remote controller that can connect with all three targets.

The ZigBee® RF4CE specification defines two profiles, located in the application layer of the OSI model, that indicate how the devices communicate to ensure interoperability [18]. These profiles are the ZigBee® Remote Control and ZigBee® Input Device described bellow, but also permits vendors to define their own proprietary profile, called manufacturer specific profiles.

ZigBee® Remote Control (ZRC) Profile are intended to define the commands needed to control the CE devices. These commands are based on the High-Definition Multimedia Interface (HDMI) Consumer Electronics Control (CEC) but the ZigBee® Remote Control (ZRC) can query the CE device in order to get the specific list of vendor commands supported by the device [16], [21].

ZigBee® Input Device (ZID) Profile enables the most recent controllers to control the CE devices. This profile use the USB HID specification, that enables devices like touchpad, airmouse, keyboard, etc. to communicate with CE devices [16], [22].

[1] From: `https://docs.zigbee.org/zigbee-docs/dcn/09/docs-09-5231-03-rmwg-understanding-zigbee-rf4ce.pdf` (visited on 10 August 2015)

2.1.3 Microchip Wireless Protocol (MiWi™) P2P

MiWi™ P2P is a Microchip Technology Inc. proprietary protocol with the same target for low power, low data rate and cost sensitive applications. As can be seen in Figure 2.6, this protocol stack uses the base of the IEEE Standard 802.15.4™ but with some modifications that correspond to MiMAC and MiWi™ P2P layers.

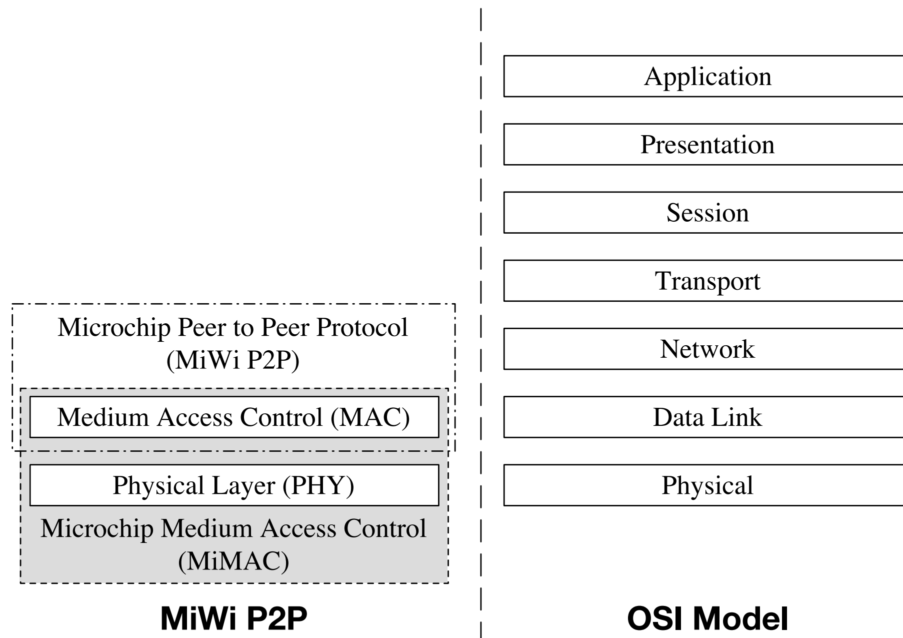
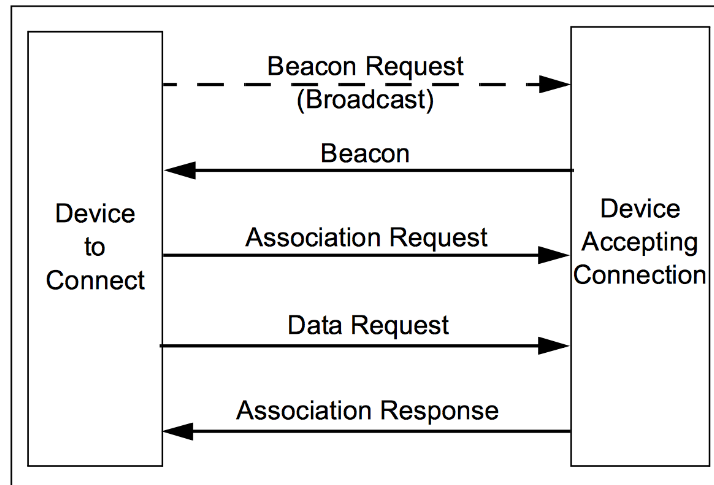


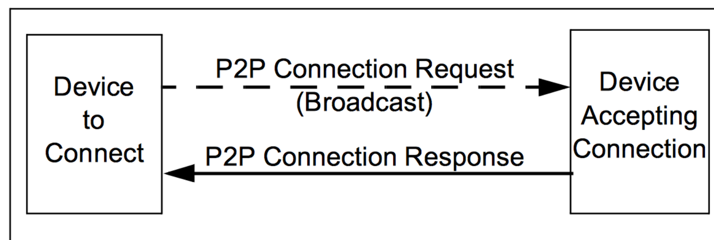
Figure 2.6: MiWi™ P2P stack[1].

The MiMAC defines the MAC layer for communication protocols and transceivers supported by Microchip, removing the dependency between Microchip RF transceivers and the protocols stacks. The MiWi™ P2P is a direct wireless communication that only allows one hop and does not allow routing, which mean that all the devices needs to be in range with the PAN coordinator. Another major difference in the MiWi™ P2P protocol compared with IEEE 802.15.4™ is the handshaking process, that only needs two steps instead of seven, as can be seen in the Figure 2.7. These changes lead to a reduced complexity in the handshaking process. The MiWi™ P2P also uses an energy scan method for finding the channel with the least noise and can perform channel hopping, given that this protocol have frequency agility [12], [23].

[1] Adapted from: MASTERs 2013 presentation - Advanced Wireless Networking (MiWi™ Protocol II), Microchip Technology Inc.



(a) IEEE 802.15.4™ Handshaking.



(b) MiWi™ P2P Handshaking.

Figure 2.7: Handshaking process[1].

2.2 Bluetooth® Low Energy (BLE)

The Bluetooth® wireless technology was intended to replace the wires for short range communications. The BLE marketed as Bluetooth® Smart was introduced as part of the Bluetooth Core Specification version 4.0, with the objective to reduce the power consumption of this technology. In the current year (2015), there are three types of devices, one with the already mentioned BLE technology, other with the classic Bluetooth® that is the original version of the technology and the last one implements both technologies and is called Bluetooth® Smart Ready devices. As shown in Figure 2.8, Smart Ready devices can act as a bridge between classic and BLE technologies, since these are not compatible with each other [24], [25].

The BLE protocol stack consists in two main sections (Host and Controller) as can be seen in Figure 2.9. The Controller comprises the lower layers of the stack that are hardware dependent, and normally is implemented as a small System on Chip (SoC) with

[1] From: `http://ww1.microchip.com/downloads/en/AppNotes/01204a.pdf` (visited on 7 August 2015)

Figure 2.8: Compatibility between Bluetooth versions.

an integrated radio interface [26]. The Host is implemented as an application processor and is less hardware dependent, since the Host Controller Interface (HCI) makes the link between these two sections.

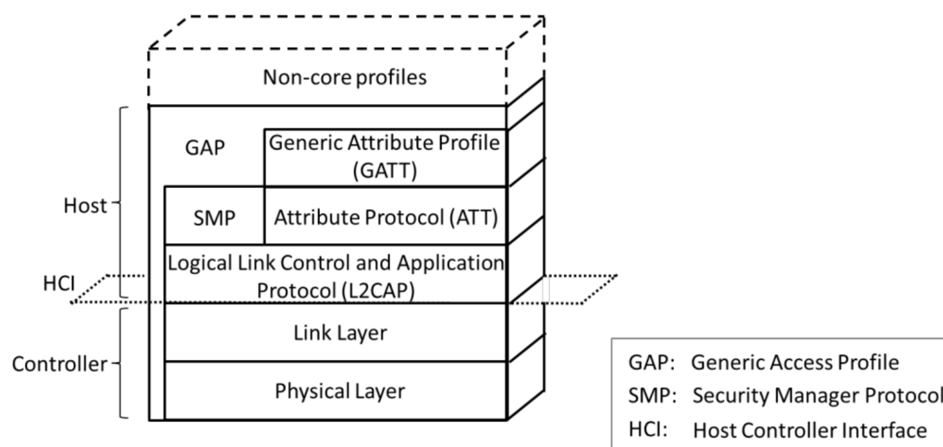


Figure 2.9: BLE stack[1].

Physical layer (PHY) contains the analogue communication and modulation. This is performed by the BLE radio that uses 40 channels with 2 MHz bandwith each, and operates in the 2.4 GHz ISM band with Gaussian Frequency Shift Keying (GFSK) modulation. The channels at 2402, 2426 and 2480 MHz, represented in the Figure 2.10, are reserved for advertising. These frequencies were chosen to minimize the overlapping with IEEE 802.11 channels 1, 6 and 11, typically used by Wifi. The remaining 37 channels are used for bidirectional communication between connected devices. In order to avoid the interferences and wireless propagation issues, the Frequency Hopping Spread Spectrum (FHSS) mechanism is used , to select one channel for communications during a given time interval [24], [26].

Link Layer (LL) is responsible for general control of the link and transport, such as enabling of encryption on the logical Transport, coding/decoding of packets and the adjustment of the transmit power in the physical layer. This layer is the most time dependent layer, since it has the scheduler that grants the time for physical

[1] From: `http://www.mdpi.com/1424-8220/12/9/11734` (visited on 15 August 2015)

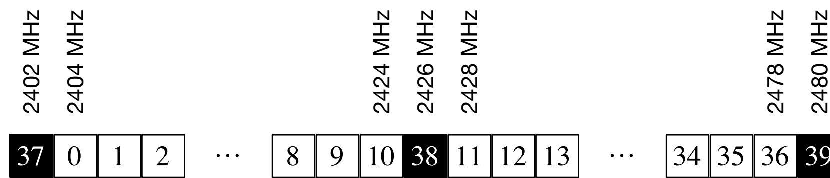


Figure 2.10: Bluetooth Low Energy channels.

channels. It also contains the device manager, that controls the behaviour of the device like advertising, scanning, initiating, connected or in standby [27].

Host Controller Interface (HCI) is a bridge between Host and Controller. This interface can be implemented as software API or through a physical interface such as Universal Asynchronous Receiver/Transmitter (UART), Serial Peripheral Interface (SPI), USB, etc [24].

Logical Link Control and Adaptation Protocol (L2CAP) provides data multiplexing for the higher layers and data encapsulation to fit the maximum payload size of the L2CAP packets, that is 23 bytes [24].

Security Manager (SM) is intended to generate and manage the pairing keys in order to have a secure communication between devices. The BLE use Advanced Encryption Standard (AES)-CCM cryptography with a block length of 128 bits [27].

The Attribute Protocol (ATT) allows to exchange attributes between devices. The attribute is a small piece of data composed by the handle (address) with 16 bit length, the type that is Universally Unique IDentifier (UUID) and value [24], [27].

The Generic Attribute Profile (GATT) is an abstraction layer to be used by the application to communicate with ATT service. Each device can expose generic attributes, that means, the device acts as GATT server, request attributes values from a server, as an GATT client or can be both simultaneously [24], [27].

The Generic Access Profile (GAP) provides an interface for the application to control the behaviour of the device [27]. The device can act as broadcaster, observer, peripheral or central. The device that acts as broadcaster sends advertising events for the observer device to connect to it. Then the observer becomes the central device that is the master, and the broadcaster becomes the peripheral device that is the slave [28].

Chapter 3

The Universal Serial Bus (USB) in Android

This chapter describes the technology used as base for the work developed in this dissertation. In the first section, the USB 2.0 is approached in order to understand how this protocol establish the communication between two devices. This section provides details of the architecture, communication and the HID class of devices. This HID class allows to adapt the communication to match the needs for a specific device and the host will automatically recognize the functions provided by the device without the need for a specific driver for each device. Also, the BLE and ZigBee® RF4CE, described in the chapter 2, implements the HID class unmodified from the USB. In the second section, the Android OS is presented, to give a better understanding of the software stack, applications, boot sequence, sensors and USB stack. This depper knowledge was needed to implement the java application, API and library for the sensors, that will be explained in the next chapters of this dissertation.

3.1 Universal Serial Bus (USB) 2.0

In this project, USB 2.0 devices are used for communication with an interface class called HID, which consists of devices that interact directly with the user. The keyboard, mouse and remote control are examples of these devices. To develop a USB HID device, it is necessary to know the USB architecture, communication protocol and available settings. These features are described in the next sub sections, followed by the characteristics of HID class.

3.1.1 Recent advances of Universal Serial Bus (USB)

USB is the most used communication interface through cable to exchange data between a host and peripheral device in a short distance. The last advances in this technology introduced the USB 3.1 generation 2 and the USB Type-C™ [29]. The USB 3.1 is a dual bus that provides backward compatibility with the previous USB 2.0. One bus is exactly the same as the 2.0 and the other is the Enhanced SuperSpeed bus, as shown in Figure 3.1. As the name suggests this bus allows a higher speed communication between devices. These speeds are described in the Table 3.1 [30].

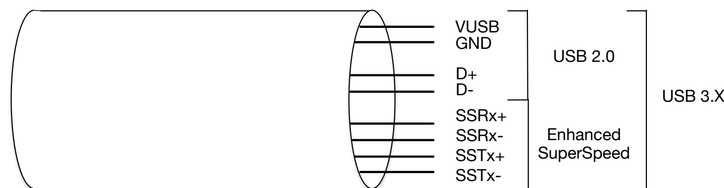


Figure 3.1: USB 3.x Cable.

Table 3.1: USB speeds

	USB 3.1	USB 2.0
Speed	Gen 1 (5 Gbps) Gen 2 (10 Gbps)	low-speed (1.5 Mbps) full-speed (12 Mbps) High-speed (480 Mbps)

The new USB Type-C is a connector ecosystem that arrives with the goal to be smaller, thinner, reversible and to allow more power through cables. This ecosystem enables the USB 3.1 and USB 2.0 given that implements both buses and allows the power up to 20 volts with 5 amperes [30]. Besides the new advances in technology, unless the highest speeds are needed, the USB 2.0 can be used given its backward compatibility. Nowadays (in 2015), the USB Type-C™ is being introduced in the new devices and despite the connector ecosystem is not in the scope of this dissertation, this option should be considered because offers a reversible connector, provides a bi-directional power and allows multiple modes, e.g. DisplayPort, HDMI, etc. To help understanding the work developed in this dissertation, the USB 2.0 and the Android OS architecture are described in the next sections.

3.1.2 Architecture

The USB is a industry-standard developed in the mid 90s, by a group of companies driven by the need to simplify the connection of peripherals with the personal computer [30]. The first version of USB was released in January 1996 and despite some limitations at the

time, it has been constantly evolving. Currently, it is an open architecture that allows suppliers to develop systems with devices able to communicate between them [31].

The USB architecture is based on the star topology divided by layers, where the HUB is the centre of each star (Figure 3.2) [32]. At the top it is located the host that is responsible for managing the communication, which is used a master/slave protocol to communicate with the devices. Next there are the connecting elements HUBs, needed to interconnect the host with devices, which are the last elements of this connection. The first HUB which is next to the host, is called "Root HUB".

The maximum number of allowed connections in USB is 127 devices and 5 external HUBs (excluding the "Root HUB") [33]. However, given the maximum bandwidth limit of the bus, the number of connected devices can be lower. The system can be separated into three main parts, the devices, the host and the interconnection between them.

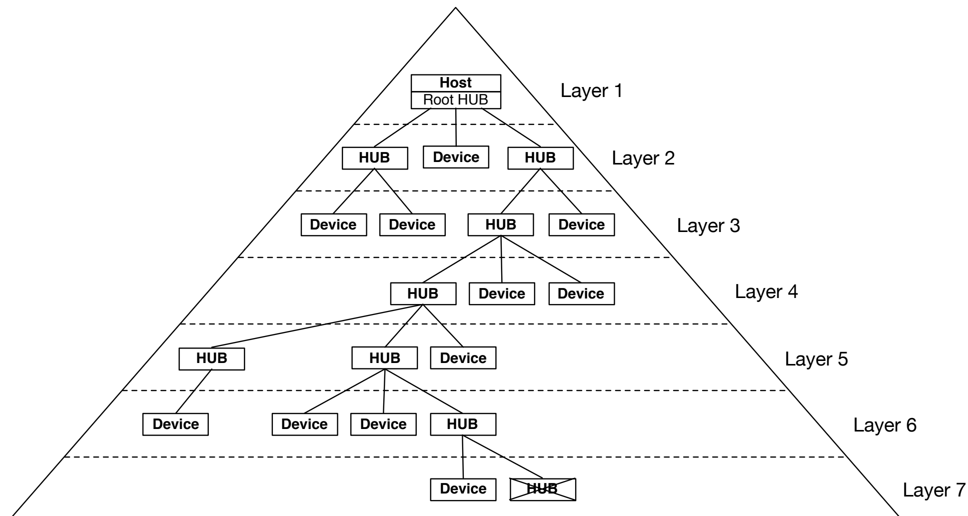


Figure 3.2: USB architecture[1].

USB Device

A USB device can be referred as a physical or logical element that provides one or more functions to the host, i.e. at the lowest level the USB device refers to the hardware component and at higher level refers to the function performed by the USB device, respectively. The USB standard defines the function as the capabilities provided to the host, e.g. mice, keyboards, printers, etc. The devices that implement more than one function, are defined by the industry-standard as composite devices or compound devices (Figure 3.3) [33].

Compound devices are independent devices, or with different addresses that are directly embedded in the HUB (e.g., keyboard with expansion ports).

[1] From: R. Regupathy, Bootstrap Yourself with Linux-USB Stack, 1st ed. Course Technology Cengage Learning, 2012.

Composite devices do not use HUBs and have only one address, however, they are composed of multiple independent interfaces (e.g., mouse and keyboard interface).

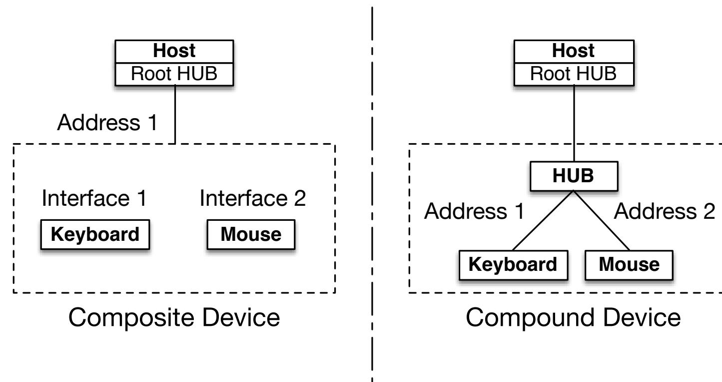


Figure 3.3: Complex Devices.

Anyway, the devices can not initiate communication with the host, they can only answer and is not allowed direct communication between devices.

Host

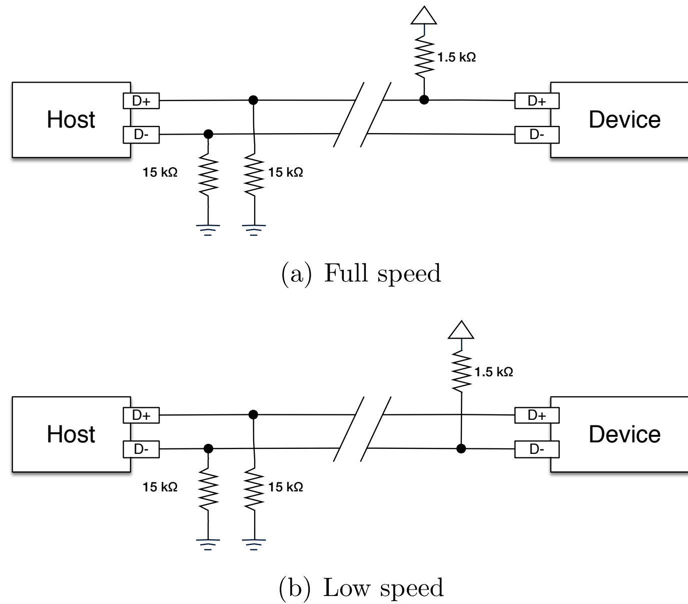
In each system, there can only be one host per bus. The host should be responsible for detecting devices connected to the bus and manage the communications, ensuring that devices are able to send and receive data when needed. It is also responsible for providing power to devices that need it (i.e., devices connected to self powered HUBs do not need power from host) as well as try to save energy consumption when it is allowed, by suspending the connection with device.

Communication speeds

The USB industry-standard, originally defined two speeds for communication, low speed at 1.5 Mbps and full speed at 12 Mbps. Later in USB 2.0, with technological advances and increased processing capacity it was necessary to define a speed higher than the previous ones, having been set the new high speed as 480 Mbps [34]. The identification of the low and full speed communication is performed through the voltage level of the data lines.

At the beginning of a connection, the host data lines, D+ and D- are set to zero volts, since there is no pullup resistor to provide power (Figure 3.4). The USB device, needs to indicate its communication speed by placing a pullup resistor in one of the data lines. For the device to announce a full-speed communication with the host the pullup resistance must be placed on the line D+ (Figure 3.4(a)), in the case of low speed, this must be connected in line D- (Figure 3.4(b)). The pullup resistors change the voltage in data lines,

these changes are used by the host to identify a new device connected to the bus [35]. Some devices may contain this resistance internally, allowing to choose through firmware where it is connected (D+ or D-), otherwise it is needed externally [33].



(a) Full speed

(b) Low speed

Figure 3.4: Configuration of speeds

The high speed communication is not set at the hardware level, however, a resistance in the data lines is required for the detection of a new device. When it intends to communicate in high speed the device must support full speed, since the communication is initiated at this speed. At the beginning of the connection, the choice between these two speeds, high and full speed, is made when the device is restarted through the process of chirp handshake (Figure 3.5).

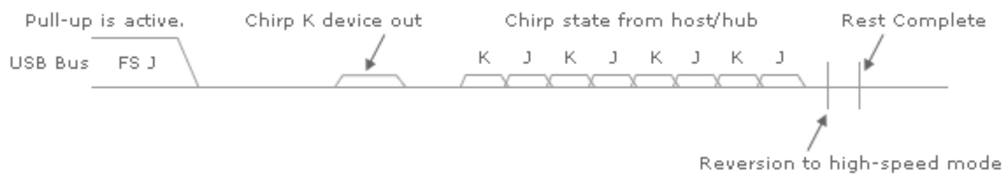


Figure 3.5: Chirp handshake[1].

The USB device creates a chirp K (i.e., with the digital signal on line D+ as 0 and 1 in the D- line) which when detected by the HUB, if this has the ability to communicate at full speed, responds by alternating k and j chirps (Table I). The device must detect at least three pairs of alternating k and j chirps to assume that the hub is capable of communicating high speed. Once detected, the device should turn off the pullup resistor

[1] From: `http://am.renesas.com/applications/key_technology/connectivity/usb/about_usb/usb2_0/usb2_5/index.jsp` (visited on 11 August 2015)

in D+ line to start the communication in high speed, if not detected, must continue in full speed mode and wait for the end of the reboot.

Table 3.2: Digital states of Chirp k and j in data lines.

		D+	D-
Chirp k	High/Full speed	0	1
	Low speed	1	0
Chirp J	High/Full speed	1	0
	Low speed	0	1

Interconnection

The USB interconnection is made through a cable with four conductors, two for data and two for power. The maximum cable length depends directly on the speed of communication. However, given the maximum limit of the bandwidth of the bus, the number of connected devices can be lower. The main length constraint is due to the fact that there is a maximum propagation time for each packet transmitted over the bus. If this limit is exceeded, the packet is considered lost. The maximum length of cables for communications at full speed is 5 meters and at low speed is 3 meters [34]. The connection can be extended up to 5 HUBs (excluding the root hub), so the limit is 30 meters at full speed and 18 meters at low speed.

3.1.3 Power supply

The USB has the ability to supply power to the devices connected to the host and the supply voltage is typically between 4.4-5.25V [34]. In this way there is no need for additional cables, which makes the devices lightweight and inexpensive, since they do not require an internal power source. However it is necessary to consider some limitations of the available power. There are three classes of consumption for bus powered devices: Low, High and self powered.

Low power devices, can sink up to $\sim$100 mA, while the high power devices can sink up to $\sim$500 mA. For higher consumption the device should be self powered and can sink up to 100 mA from the USB bus. The self powered HUB can supply 500 mA to each device, while if it is fed by USB bus can only provide 500 mA distributed by the devices [33].

3.1.4 Communication

The communication between USB devices is divided in three main sections: software, abstraction of the protocol and hardware (Figure 3.6).

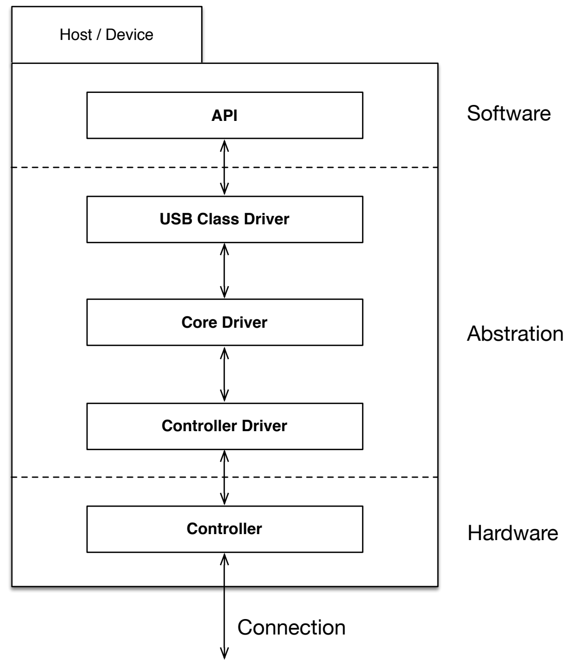


Figure 3.6: USB protocol stack.

The software section is where the custom application is created to transmit/receive data. The Abstraction is divided into three layers: USB class driver, Core Driver and Controller Driver. USB Class Driver is composed by specific methods for different classes available in USB protocol (eg HID, video, audio). The Core Driver implements the USB protocol base functionality. The Controller Driver makes the link between the software and the hardware, so contains methods for reading and writing data in the physical channels. The next layer of the protocol stack is the hardware that enables connection between devices and the host.

The logical connection between the host and a terminal on the device is called a pipe (Figure 3.7). This terminal on the device, called the endpoint and are typically in pairs, respectively the input and output Endpoint. The Endpoint is seen as a buffer for the device, since it has no initiative to communicate, and thus requires a memory space to put the data until the host requests them. In the case of the host, which controls the data communication, the endpoints are viewed as a pipe and not as buffer. At the beginning of connection, the host starts the configuration of device, where the device indicates to the host the available endpoints, their respective characteristics and which interfaces are associated to it, except for the Endpoint 0. This endpoint is required on all devices, since it is exclusively used by the host to control and configure the device.

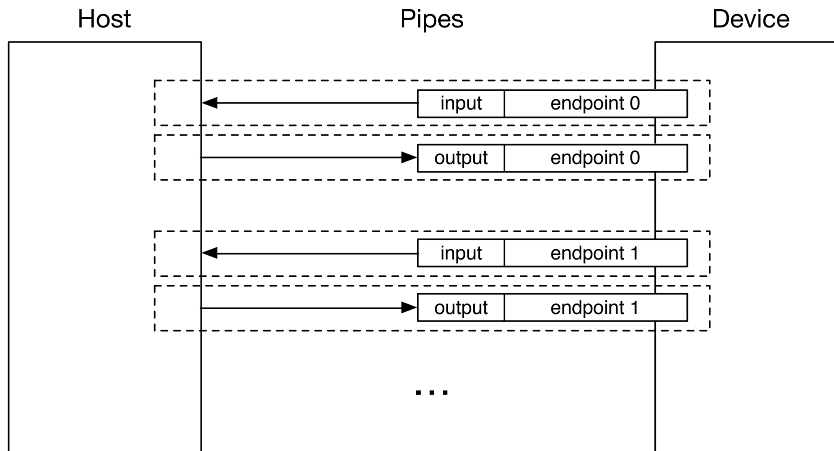


Figure 3.7: USB logical connections.

In USB communication there are two types of pipes, the Message pipe that is only used for device configuration, and the Stream pipe which serves to exchange data between device and host.

The mesage pipe has a format defined by the USB standard. This pipe is controlled by the host, and data can flow in both directions, but who determines the direction is the host, devices only can answer to requests. This supports only control transfers, which is directly related to the endpoint 0 [36].

The stream pipe contains no format, meaning that the device can send any kind of data throught the pipe. All the pipes have a predefined direction, input or output, and can be controlled by the device or host [36].

3.1.5 Types of transfers

The transfers follows a structure defined by the USB protocol. That is, the data to be transported through the stream or message pipes, are encapsulated according to the type of transfer to be made. The USB defines four different types of transfers, allowing to be chosen which is the most suitable for a particular communication. The types of transfers are the following [33]:

- ***Control Transfers:*** They aim to configure, give orders and request information about the current state of the device. Such transfers can only be made through the message pipe (Endpoint 0).

- ***Isochronous Transfers:*** Are designed for streaming data in real-time. This type of transfers guarantees the bandwidth, however, it is not made any error checking.

- ***Interrupt Transfers:*** They are suitable for small amounts of data to be transferred. During the configuration, the device tells to the host the maximum time

allowed between data readings. If the device has no data to send, must return Negative-Acknowledgment (NAK).

- ***Bulk Transfers:*** This is intended to transfer a large volume of information and uses the maximum speed available for the transfer, but it is not guaranteed the moment in which is made. This type of transmission has the lower priority, however, it ensures that the transfer is made without errors.

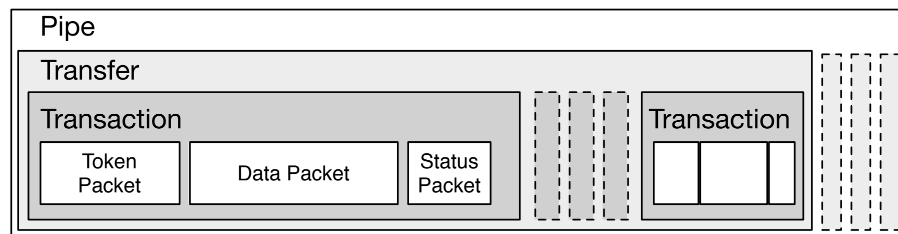


Figure 3.8: USB transfers[1].

The transfers are made of one or more transactions, and each one contain up to three packets (Figure 3.8) [37]. The first packet is the Token packet and contains information about the type of transaction and its direction, device address, and the endpoint. The second package, that is Data packet, consists in the data to be transferred and the last packet, Status packet, refers to the state of the transaction, that is, if it has been a successful transfer or not.

The data transmitted is encoded with the Non Return to Zero Inverted (NRZI) line code and the Least Significant Bit (LSB) is send first. To ensure frequent transitions, after successive six bits to '1', it is inserted the bit '0', this is known as bit stuffing [33]. All packets begin with a synchronization pattern to enable the receiver clock to be synchronized with the transmitter. This synchronization is considered as the start of packet, and the end of packet is identified by an delimiter pattern.

The complexity of the protocol requires its implementation in specific hardware to save computational resources from a micro-controller. This implementation is called Serial Interface Engine (SIE) and is located in the hardware layer of the protocol stack (Figure 3.6) [38].

[1] From: `http://www.keil.com/pack/doc/mw/USB/html/_u_s_b__protocol.html` (visited on 11 August 2015)

3.1.6 Descriptors

The descriptors of the USB devices consist in a data structure that allow to present properties of the device to the Host. During the detection of a new device, the host starts the enumeration process that relies on identifying the USB device and loading the correct driver. The device identification is performed trough the transfers of descriptors in a question and answer process. These descriptors are separated by several levels and are organized hierarchically (Figure 3.9) [35], [32].

The device descriptor at the top level describes the vendor, product, class, subclass, device protocol and how many settings exists. Each configuration contains a descriptor, which informs the host about the configuration number, the maximum consumption of the device, the features that are supported (e.g., Self-powered, remote activation, etc.) and the number of interfaces. The interfaces represent functions implemented by the device (e.g., Mouse, keyboard, etc.). The descriptor of the interface consists of the description of the class, subclass, protocol, and number of endpoints. The descriptor of endpoints contain the type of transfer, the maximum size of a packet and the address.

Devices USB HID constitute a specific class of an interface device is necessary in which additional descriptors, these are described in the following section.

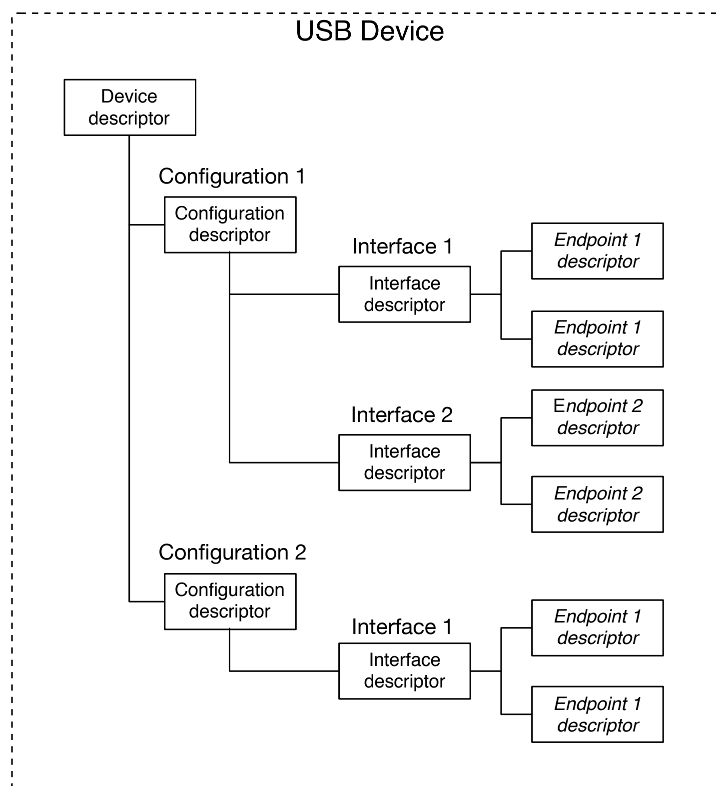


Figure 3.9: USB descriptors.

3.1.7 Human Interface Devices (HID)

The HID is a class of USB devices defined by a set of standards and communication protocols for devices that interact directly with humans. The USB protocol is used as a base, as well as enumeration process, whereas USB HID device uses the description to announce its operation to the host.

Through the descriptors, it is possible to unify the communication protocol while maintaining the freedom to define the data sent over the communication channel. As an example we have the mouse, which is an USB HID device and features a variety of buttons, as well as the cursor movement functionality on the computer screen. That is, through the descriptor we can set which data corresponds to the movement, and these are recognized by the operating system through the drivers. However more data can be sent, enabling the inclusion of additional features (e.g., ten buttons on a mouse).

The descriptor of an USB HID consists in the descriptors of an USB device, plus the descriptor of the HID functionality, which in turn is divided in the device's physical description and the report that is sent to the host.

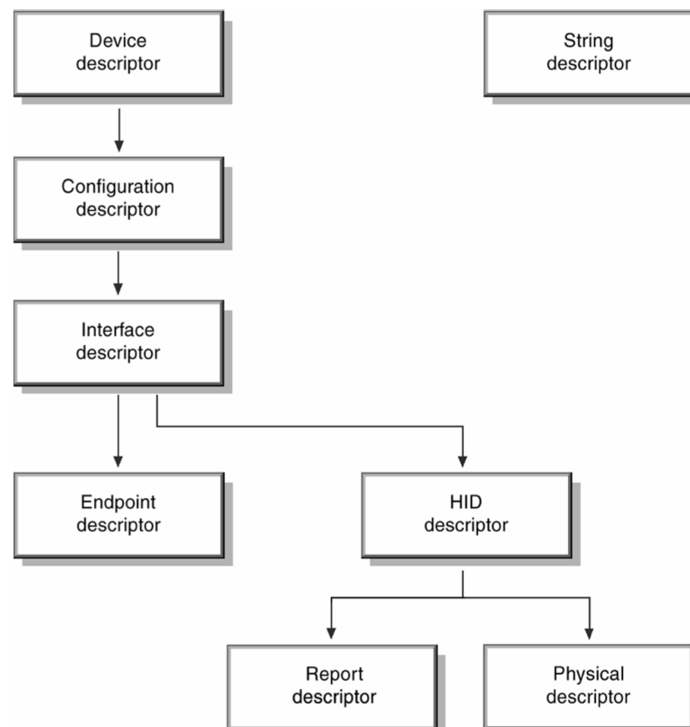


Figure 3.10: USB HID Descriptors[1].

[1] From: Device Class Definition for Human Interface Devices (HID), June 2001

Report descriptor

The report descriptor of a device is a set of data that is exchanged between device and host, and there are three report types: Input Reports, Output Reports and Feature Reports. The descriptors sent by the host to the device are referred as Output Reports, while the descriptors from the device, that are answers to the requests from host, are referred as Input Reports. The Feature Report is optional and the data that can be manually read and/or written. The data in this report normally represent the state of the device and their configuration. The structure is defined by the descriptor of the USB HID device, which refer the details of the device and the data sent. The USB standard has a document with the details of all the features available for the USB HID report [39]. This report allows to assign meaning to data, so that the applications on the host side might show interest in a particular functionality without the need to know the place in where it came (e.g., X and Y of mouse).

The mouse report, usually contains the declaration of three buttons, wherein each button is represented by 1 bit. Through the report it is possible to define the meaning of each bit (e.g., the first bit is the left button, the second bit is the middle button, etc.). Through the meaning of the data for the predefined fields in the report, the developed application does not need to know the order of them. However, it is also possible to define a custom communication, where the fields do not are predetermined by the standard USB HID. Given this flexibility the report does not have a fixed size, since this varies according to the amount of data to send.

The report consists in a vector of bytes, that describes the data. These bytes usually are shown in two columns. The first column is the command and the second it is the value (Report 3.1).

Report 3.1: HID

```
1  0x05, 0x01,   // USAGE_PAGE (Generic Desktop)
2  0x09, 0x02,   // USAGE (Mouse)
3  0xa1, 0x01,   // COLLECTION (Application)
4  0x09, 0x01,   //   USAGE (Pointer)
5  0xa1, 0x00,   //   COLLECTION (Physical)
6  0x09, 0x30,   //     USAGE (X)
7  0x09, 0x31,   //     USAGE (Y)
8  0x15, 0x81,   //     LOGICAL_MINIMUM (-127)
9  0x25, 0x7f,   //     LOGICAL_MAXIMUM (127)
10 0x75, 0x08,   //     REPORT_SIZE (8)
11 0x95, 0x02,   //     REPORT_COUNT (2)
12 0x81, 0x06,   //     INPUT (Data,Var,Rel)
```

```
13 0xc0,         //   END_COLLECTION
14 0xc0          // END_COLLECTION
```

The command USAGE_PAGE, selects the group of devices that will be used. The USAGE indicates which type of device that is inside of the previous group will be used in the definitions of the data.

In this case, the Report 3.1, begins by defining the table of generic desktop devices at line 1, and then indicates the device functionality as a mouse, on line 2. After we indicate the type of device, we can organize data into groups. In this case, we indicate that we have a collection of data that is used by applications (line 3). This collection uses data of the type pointer (line 4), that is, will be composed by data that corresponds directly to a screen axis. The pointer type collection, contains a collection of data that is directly represented geometrically, hence the collection of physical data type. This collection consists of the data X and Y (line 6 and 7). Each data can have a logical value between -127 and 127 (line 8 and 9), that are represented by 8 bits (line 10). On line 11, it is indicated how much data is reported, and in this case is 2 times the report size (8), representing X and Y values. Finally, is indicated that the data is an input on the host, and its value is variable and relative (line 12).

Physical descriptor

The physical description of the device, indicates which parts of the body should be used to handle the device. In the case of mouse, it is possible to describe which fingers are used to activate the buttons, as well as to distinguish between left-handed and right-handed. This description is not mandatory and in most devices, this type of information does not bring much use.

3.2 Android Architecture

Android was initially a company founded by Andy Rubin, Chris White, Nick Sears, and Rich Miner in October 2003, with focus in mobile devices with capabilities to adapt to information to the user. In August 2005 it was acquired by Google that began building partnerships with companies related to mobile ecosystem and in November 2007 has announced the Open Handset Alliance (OHA)[1] [40], [41].

The first version of Android OS in Alpha phase was only available for OHA and Google members. Only on 5 November 2007, the Beta phase was available to the community and that date is popularly known as the Android birthday [41]. However, the first commercial

[1] http://www.openhandsetalliance.com/index.html

version 1.0 was released only on 23 September 2008. Besides the two first versions (1.0 and 1.1) that do not have official names, all the other releases have tasty names that follow the Latin alphabet in the first letter. Since they only started in the third release, the first official name "cupcake" starts with the third letter in Latin alphabet [41], [42].

As Android was growing, the variety of devices that implements the OS was also growing. Nowadays Android OS is designed for phones, tablets, wearables, televisions and auto-mobiles, but given the fact that it is open-source, it allows to be implemented in a variety of other devices. It is worth noting that although the Android OS is claimed as open-source, the truth is that some times does not follow that spirit. The source code may take some time to become available and consumer devices contain several closed source software components that difficult the porting for other devices [41]. In order to better understand what is involved in the Android OS, this section presents an overview of the Android software stack, followed by the boot sequence and the USB and Sensors architecture in the following sub sections.

3.2.1 Software Stack

The Android Software Stack can be represented in five main layers (Figure 3.11), the Android Applications layer, Android Framework layer, Android Runtime layer, Hardware Abstraction Layer and Kernel layer. The lowest layer in the stack will connect directly with hardware in device.

Android Applications Layer is where stays all the applications that interacts directly with the user. This applications can be native (e.g, Phone, Settings, Contacts, etc.) or third party (e.g, Skype, Facebook, etc.).

Android Framework Layer is a set of libraries for Android Applications that provides the generic functionalities of the Android OS, in order to prevent the developer from having to code the most basic tasks, which would require an increased effort to develop the application (e.g., Telephony Manager, Location Manager, etc.). Android Runtime Layer comprises in a Virtual Machine (VM) and in core libraries in Java. The both upper layers are developed in Java and execute within the VM. The VM has the functionality of byte code interpreter.

Hardware Abstraction Layer is implemented in C programming language and allows the upper layers in stack to be independent of the hardware used. At the same level are the native core libraries in order to guide the device in handling different types of data (e.g., Open GL|ES, WebKit, libc, etc.).

Kernel layer of android consist of a Linux kernel adaptation that has the main purpose

[1] From: http://www.slideshare.net/opersys/inside-androids-ui

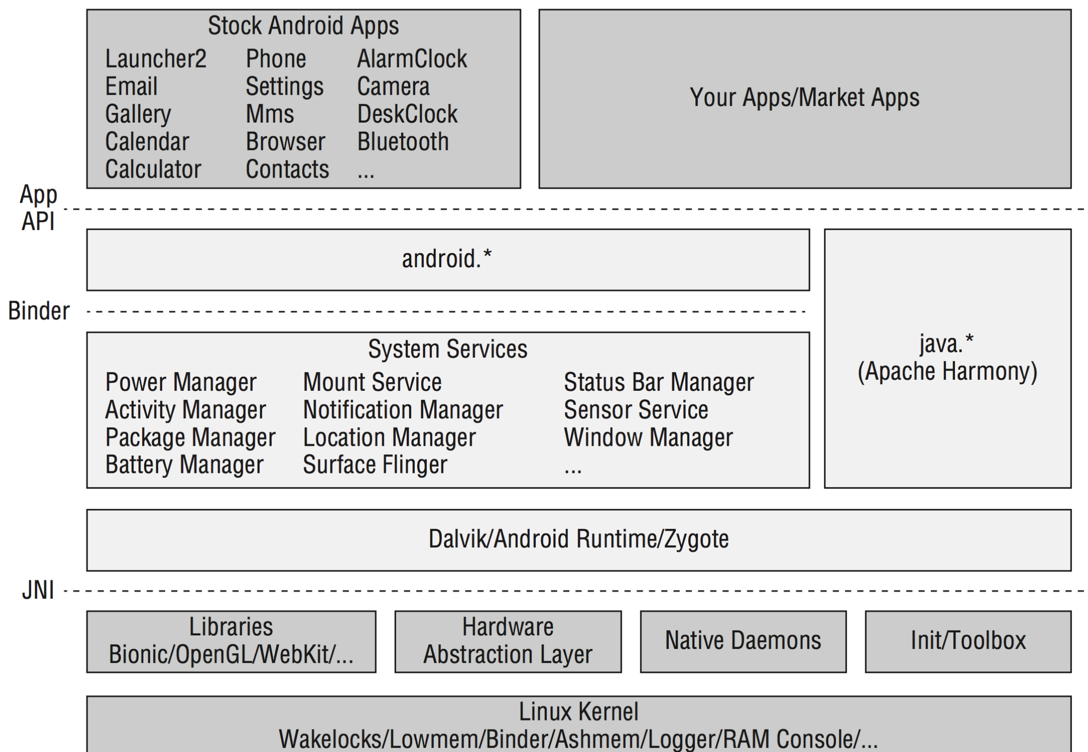


Figure 3.11: Android stack with details[1].

of managing the input/output requests from the upper layers into hardware signals.

Android Applications

The Android applications are written in Java through the Android SDK tools that creates an Android application Package (APK). To create a more secure environment, all the applications that run on the Android OS, live in their own sandbox. This means that each application has its own VM and the code runs in isolation from the others applications, applying the principle of least privilege [41].

The applications can be build with four essential blocks, know as Activities, Services, Content providers and Broadcast receivers. An activity is responsible for managing the content displayed on the user's screen. Normally the activity is considered the main of the application, thus it is only possible to have an activity to run at a time. The services are aimed at operations that take a long time and have to be initiated by other components. Since these can be performed in the background, it is possible to avoid freezing the user interface. The content provider is responsible for managing the access to a set of data, that is, provides the data from one application to the application that request it. Broadcast receivers are components that receives the announcements that exist on Android OS. This announcements can be originated by the system, like the screen rotation, or can be originated by the applications to inform all the other about some action [43].

Applications can be divided into three categories, Foreground Activity, Background Service or Intermittent Activity. The Foreground Activity applications are composed mainly of user interface to perform a certain task and are suspended when this interface is not visible to the user. As a simple example we have the games, which are only active while the user is playing. Background Service applications typically do not depend on the user interface and are used to perform periodic tasks or tasks that take a long time. Intermittent Activity applications usually consist of both the Foreground Activity and Background Service [44].

The anatomy of Android application comprises files and folders, where the most important are shown in Figure 3.12. The "java" folder contains all the java source files of the project and by default its created the "MainActivity.java" that runs on application startup. The "res" folder groups all resources in the project, where have inside four main folder. The "xml" folder contains the preferences and configuration files. The "values" folder contains simple values, such as the definition of colour value for a colour string. There are multiple layout and drawable folders, since they can be specific for the size of the screen and resolution. The "layout" folder contains the files that represents the screens user interface. The "drawable" folder is composed by the Bitmap files used by the layout files. The "AndroidManifest.xml" file describes the fundamental characteristics of the application, such as the components used and the main activity to load at startup.

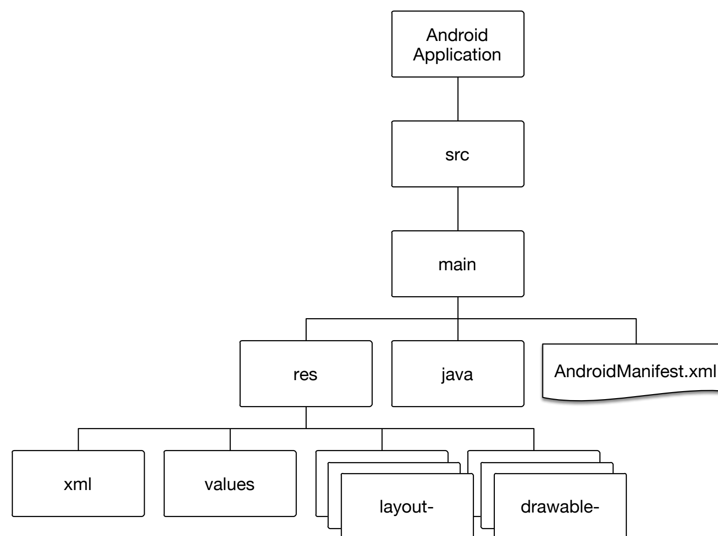


Figure 3.12: Android application anatomy.

Android Framework Layer

Android Framework is the base of every application, which includes everything between the Java applications and the native user space, or as can be seen in the Figure 3.11, between "App API" and "JNI". This includes the framework packages "android.*", stan-

dard Java classes "java.*", system services used to manage the functionalities of the OS and the Android runtime. In opposite of applications, the framework was designed to be used as it is, to change the behaviour of it, is required to go into source file of Android OS. The Binder is an inter-process communication (IPC) mechanism to manage the communications between components [45].

Android Runtime Layer

The Android runtime is a VM that replaces the Dalvik VM in Android 5.0 "Lollipop". To understand better the differences between them, it is necessary to explain the basic concept of Java VM. The Java VM is an interpreter that converts bytecode into machine executable code. The bytecode is an intermediate form of Java applications that is not hardware dependent. The Dalvik VM and Android runtime are like a Java VM specifically design for embedded systems. The Dalvik VM makes the compilation during the runtime, which is called Just-in-Time (JIT) compilation, while the Android runtime use the Ahead-of-Time (AOT) compilation, which can improve application performance given that during the runtime the application is already compiled. It is noteworthy that the Android runtime is compatible with the Dalvik VM bytecode [46]. The Java Native Interface (JNI) provides an interface to the native application in C and C++ languages communicate with Java applications and vice-versa.

Android Hardware Abstraction Layer

The Hardware Abstraction Layer provides the connection between the kernel drivers in the bottom layer and the services in the upper layer. This abstraction hides details of hardware, allowing to have different hardware to accomplish the same task, that is, this standardizes the function provided by the hardware, regardless the differences between manufacturers.

Android Kernel Layer

The Android kernel is an adaptation of Linux kernel where it is added important features for a mobile embedded platform. It is important to mention that the development of device drivers, keeps unchanged compared to Linux. However, given these specific additions to Android, it is not possible to directly import a Linux kernel for Android [47].

In order to understand better what is involved in the OS, in the next sub section it will be explained the boot sequence.

3.2.2 Boot sequence

The boot sequence in Figure 3.13 starts when the power button of device is pressed. Then the code is executed from pre defined location which is hardwired on Read Only Memory (ROM) and loads the Bootloader. The Bootloader starts the Random Access Memory (RAM) disk and sets the basic requirements for kernel to load, this is also a checkpoint for security in order to load only the OS allowed. The kernel is responsible for managing the available hardware resources, making their respective startup and ensuring that they are available to the OS. Upon completion of the hardware configuration, the initiation of Android OS starts by loading the "init.rc" file. This file contains the instruction to set up environment variables, create mount points, mount file systems, set out of memory (OOM) adjustments, and start native daemons. Also the file "init.<device>.rc"[1] is loaded with the instructions for a particular device. It is in this step of initialization that the VM is started.

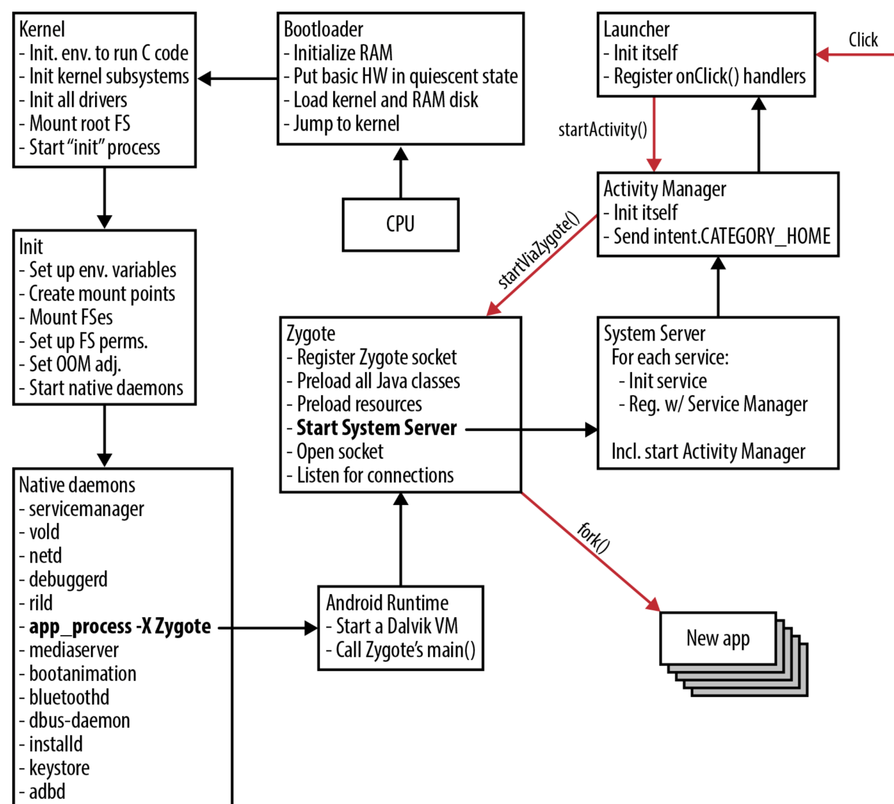


Figure 3.13: Android boot sequence[2].

In the normal operation of the Java VM for each application it is initialized one VM, but given the restrictions of the memory in this kind of devices, the Android OS have a daemon called "Zygote" to launch applications and manage this issue. This daemon when

[1] Where <device> parameter contains the code name of device

[2] From: K. Yaghmour, Embedded Android. O'Reilly Media, Inc., 2013

is started preloads all the necessary Java classes and resources, starts the System Server and opens a socket to listen the requests for starting applications. The System Server starts the activity manager that loads the launcher, known as home screen for Android users. So when the user wants to start an application, performs a click on the icon, then the request is made to the activity manager that forwards it to the Zygote, which in turn makes a fork of the existing VM. The issue with memory is surpassed with the kernel policy of copy-on-write (COW) for forks, that means, when the fork is made the existing memory is not really copied, instead it is only created a reference that points to this memory. Is only created a copy when it is needed to write in this memory, given that this memory comprises in Java classes and resources, where they are normally immutable, is never created a copy of this memory. So this memory is only loaded one time and all the applications use their own VM, with a reference for this memory that contains all the resources needed and already loaded [45].

3.2.3 USB architecture

Under the scope of this project, Figure 3.14 shows the Android USB stack in host mode. This stack is similar with the one presented in Figure 3.6, in section 3.1.4, where the abstraction layer of Figure 3.6 is presented in the Figure 3.6 as kernel layer. The "libusb-host" (a thinner version of "libusb") is the user space library that allow the detection of USB devices and provides the access for controlling data transfer between host and device. The USB Service and USB Function blocks are frameworks in the android.hardware.usb package to provide the USB access to Java Applications [48].

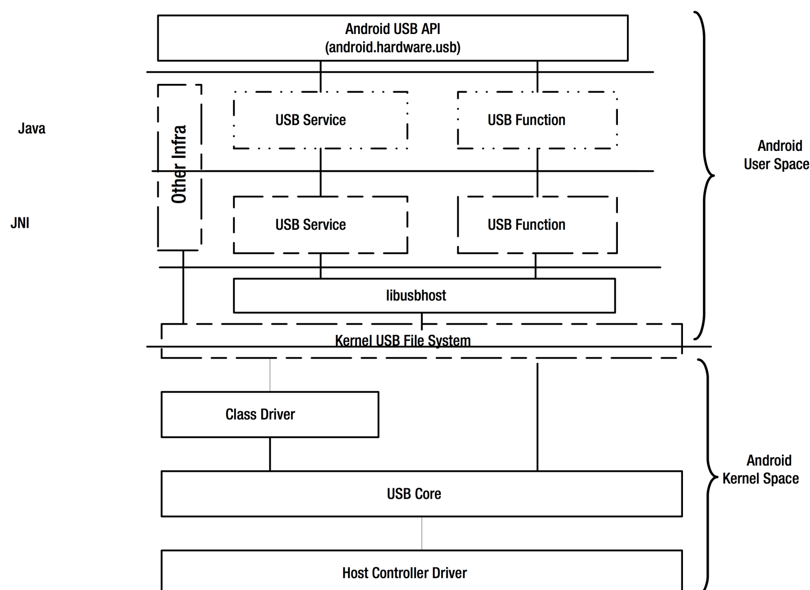


Figure 3.14: Android USB stack[1].

[1] From: R. Regupathy, Unboxing Android USB: A hands on approach with real world examples. Apress, May 2014

3.2.4 Sensor stack

Most android devices have sensors built-in the device and the Android platform divide them into three categories: Motion, Environmental and Position sensors [49].

Motion sensors are referred to as the sensors to measure acceleration and rotational forces, such as, accelerometer and gyroscope.

Environmental sensors measure environmental parameters like illumination, temperature of air and humidity. Barometers, photometers, and thermometers are normally the sensors used.

Position sensors measure the physical position of the device, such as, orientation sensors and magnetometers.

Besides the hardware-based sensors, the Android platform also defines some virtual sensors that derive from the data of one or more hardware-based sensors. These virtual sensors are created by the sensor fusion implemented in the Android Open Source Project (AOSP) that already includes the Gravity, Linear acceleration and Rotation vector sensor [50], [51].

Gravity sensor uses the accelerometer and gyroscope (if available) or the magnetometer (if gyroscope is not available) to report the direction and magnitude of gravity in m/s^2 for the three axis (x, y and z).

Linear acceleration sensor results from the difference between the accelerometer and the gravity sensor. The hardware-based sensors used are the same as the Gravity sensor and the output is also in m/s^2 for the three axis (x, y and z).

Rotation vector sensor reports the quaternion orientation (x, y, z and w) that is obtained from the values of accelerometer, magnetometer and also the gyroscope when available.

The Android uses the East-North-Up coordinates system, presented in Figure 3.15, as reference for sensors. This coordinates system does not change when the device's screen orientation changes. The coordinate system is pre-defined with the natural device orientation, that means it can be based on either portrait or landscape orientation, depending on the device natural orientation, but the coordinates system does not change during the utilization of device [52].

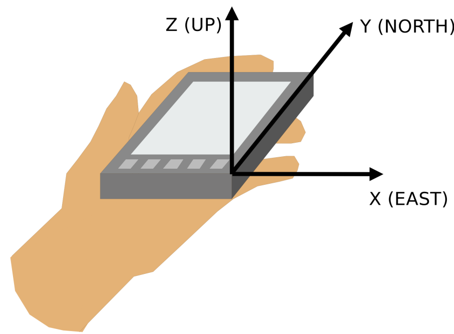


Figure 3.15: East-North-Up coordinates[1].

The Figure 3.16 presents the Android OS stack for the hardware-based sensors. The bottom layer represents the driver to communicate with the hardware, this communication is not specifically defined by the Android, it can be implemented using the different protocols available, such as Inter-Integrated Circuit (I2C), SPI, USB, etc. The driver also creates the sensor in the Android file system, to be able to export the data to the user space. Then the "Sensor HAL" manages the data from sensor and the request from the upper layers in the Android OS. The Sensor Manager is part of the framework available for the application in Java to interact with sensor.

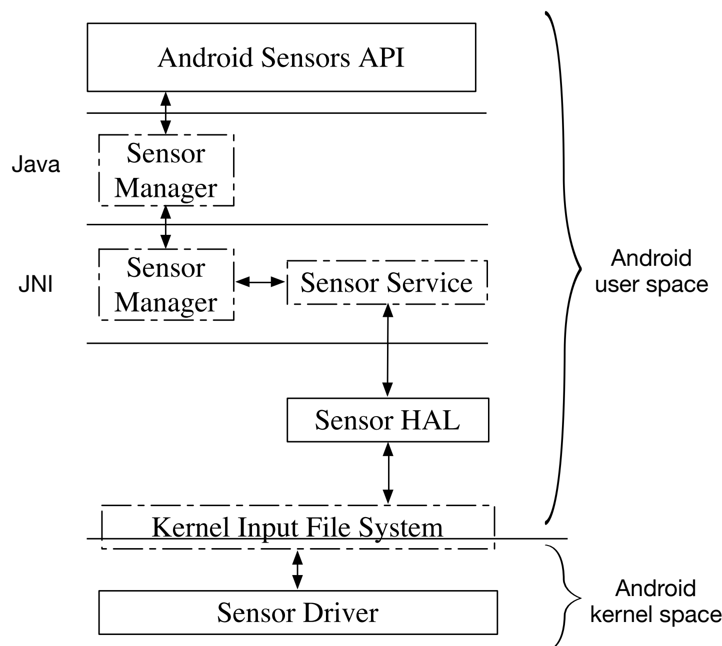


Figure 3.16: Android sensors stack[2].

[1] From: Hardware abstraction layer for Android. STMicroelectronics, September 2012. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/DM00063297.pdf

[2] Adapted from: https://source.android.com/devices/sensors/sensor-stack.html and http://blog.chinaunix.net/uid-21074389-id-3217663.html

This page was intentionally left blank.

Chapter 4

Development of Android interfaces for enhanced remote control

The system developed in this work aims to enable new forms of interaction with 3D content and to improve the user experience. As can be seen in Figure 4.1, it consists in a RCD with 6 Degrees of Freedom (DoF) and a STB. The RCD prototype presented in Figure 4.2 was provided by the Tech4Home[1] company, an Small Medium Enterprise (SME) expert in the field of RCDs.

The RCD prototype is able to track 3D motion using a set of sensors commonly known as Magnetic, Angular Rate, Gravity (MARG). MARG Sensors are composed by an accelerometer, gyroscope and magnetometer, each with 3 orthogonal axes. In this work the MARG unit from Ivensense[2] also includes an Application-Specific Integrated Circuit (ASIC) embedded processor, designated as Digital Motion Processor (DMP), which computes the orientation of the device using the information retrieved by the accelerometer and gyroscope sensors. This RCD has the following six modes, that were also implemented in the Android STB to identify and read the data from RCD [53].

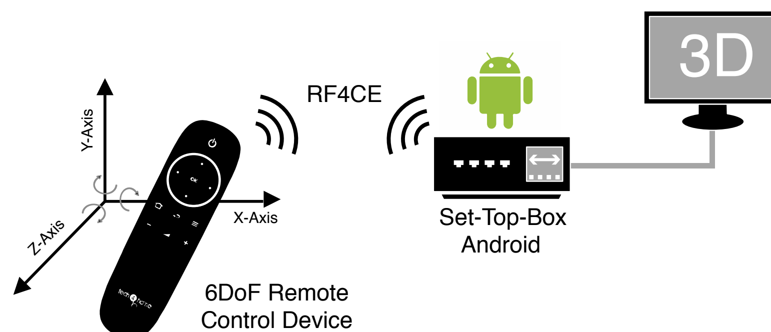


Figure 4.1: System developed.

[1] Tech4Home: www.tech4home.pt
[2] Invensense:http://www.invensense.com

Figure 4.2: Remote Control Device prototype.

Idle Mode

In this mode of operation, only the key events are sent by the RCD, including: key pressed, key released or key repeated, are transmited. The MARG unit remains in sleep mode and no motion events are sent.

Relative Mouse Mode

The relative navigation is calculated on RCD with tilt compensation, which sends the mouse displacement. In this mode of operation keys can also be send. Data is only transmitted in this mode when there is relative motion or a key is pressed.

Absolute Mouse Mode

The difference of this mode to the previous is only regarding to the type of navigation. The information retrieved by the DMP converts the orientation to absolute coordinates in the screen. In this work the calculations needed to proceed with this type of navigation was implemented in a custom Android API, because the information about screen resolution was needed and the communication was unidirectional.

Demo Mode

During the development of this work an Android application, that will be explained later, was made in order to qualitatively access the orientation retrieved by the implemented filters and the DMP. This mode was created to allow demonstrations of the accuracy of the remote orientation in a subjective way and to give an example of navigation in multimedia contents using gestures.

Scroll Gestures Mode

This mode allows scroll and zoom using rotations of the RCD. Similarly to the Relative Mouse Mode, the information is sent only when there is a rotation wide enough to generate a scroll event on the screen.

Sensors Mode

In this mode all the sensors (accelerometer, gyroscope and magnetometer) readings are sent to the Android system. Data rate for this mode is performed at 50 Hz.

The Android based STB runs the OS 5.0, it has no limitations on energy consumption and a far superior computational power, ROM and RAM memory, when compared to the RCD. The communication between devices is done wirelessly with the ZigBee® RF4CE protocol.

Given that the RCD is intended to be a low-complexity device, an issue in this work was to decide whether the most complex functions in terms of computational complexity should run on the RCD or on the STB, taking into account energy consumption. On the one hand, running complex algorithms on the RCD results in high energy consumption for processing orientation estimates and low consumption data transmission, because less data is transmitted. On the other hand, by transmitting the raw data to the STB, increases the energy required for transmission but allows the implementation of more complex algorithms on the STB, thus higher precision.

To evaluate the problem of computational load balance, two solutions, marked as option "A" and "B" (see figure 4.3), were implemented:

Option "A" RCD movement is acquired by the MARG unit, the raw data is used by the RCD processing unit, and the processed data is transmitted to the STB;

Option "B" All raw data is transmitted from the RCD to the STB where it is processed.

An external module (USB dongle) was developed to implement the communication between the RCD and the STB via RF. The dongle receives data from the RCD through ZigBee® RF4CE protocol and forwards them to the STB through USB 2.0 HID custom.

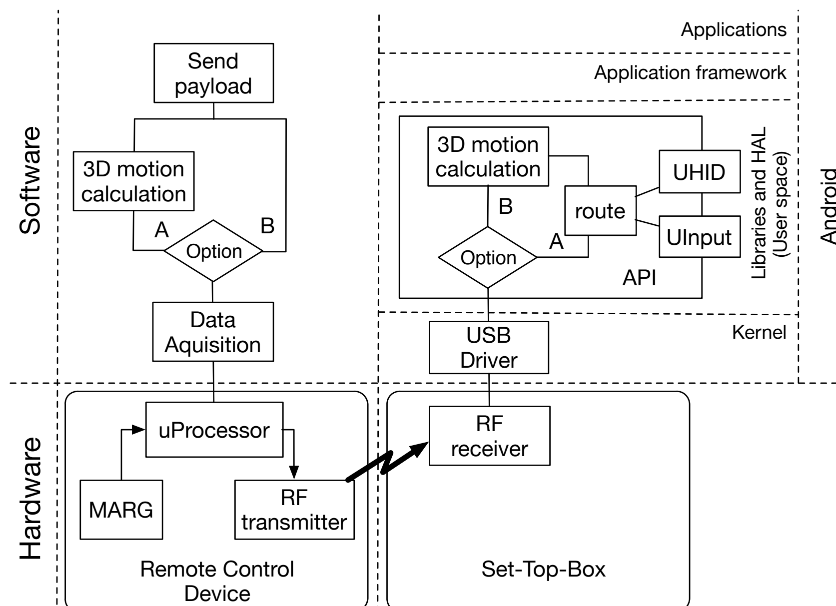


Figure 4.3: System architecture.

4.1 Dongle: transceiver RF-USB

The dongle was developed with the purpose of receiving data from the RCD and forwards them to the STB. The scheme and final drawing of Printed Circuit Board (PCB) dongle that was developed is presented in the Figure 4.4 and 4.5, respectively. The module is fed by 5 V from the USB connection with the host, using a voltage regulator to 3.3 V needed by the microcontroller and RF modules. The Light Emitting Diode (LED) is a power indicator. The capacitors are those recommended by the datasheet of the microcontroller used from Microchip Technology Inc., which in this case was PIC32MX795F512L. It was also implemented a button, used to reset the microcontroller. The High Speed (HS) external oscillator used was necessary due to the use of USB, which requires an external oscillator for a correct operation, since it only utilizes the internal oscillator for the detection of new connections. For the communications a female head was used for the MiWi™ and ZigBee® RF4CE modules, and the USB 2.0 female connector of Type A. It is noted that the female connectors are intended to create a socket for each module, which eases the exchange between modules. It is noteworthy that only one RF module can be used at a time and given this constraint, both modules were implemented in the same area of the PCB, as can be seen in Figure 4.5(b). Connectors for programming the microcontroller and the Tech4Home module were also left available.

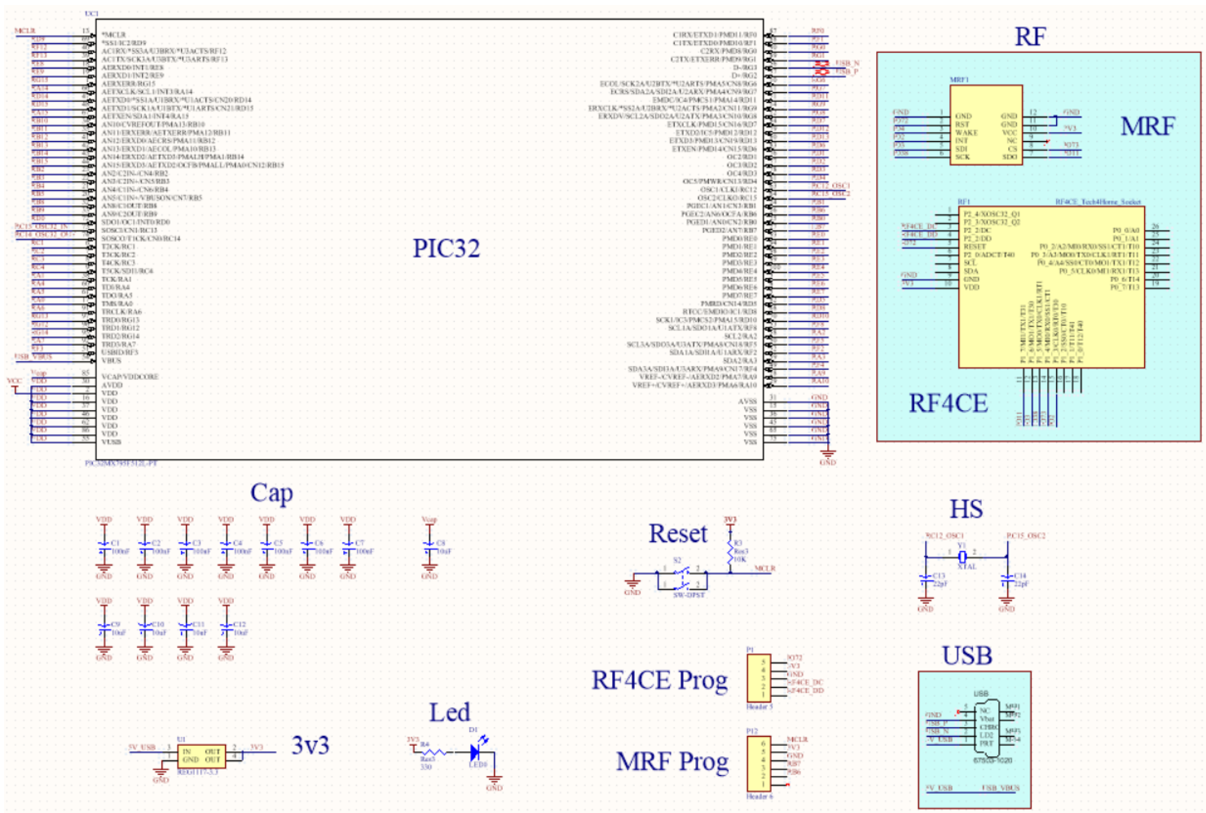
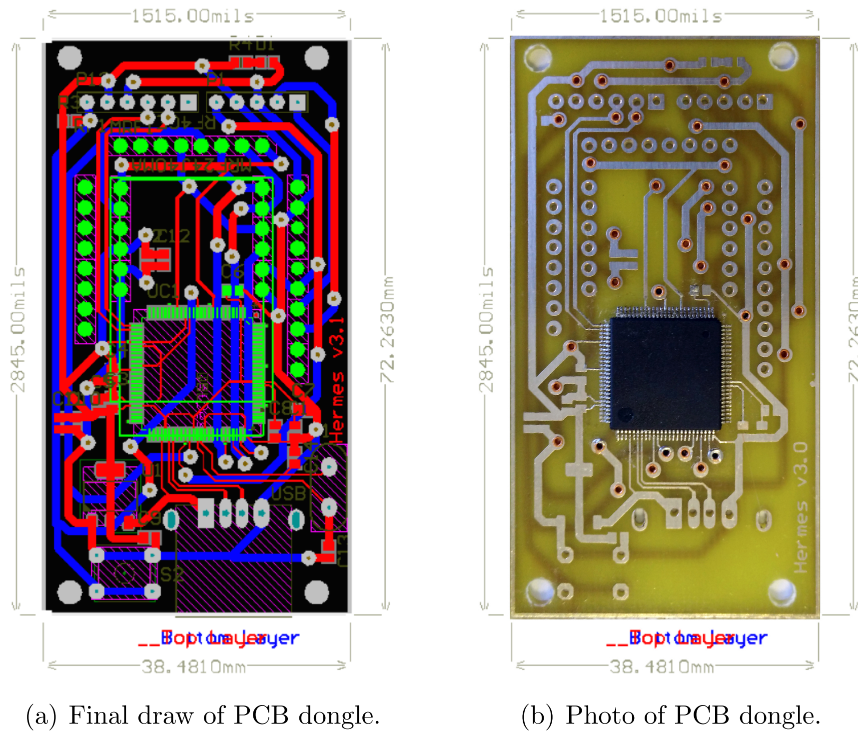


Figure 4.4: Final scheme of PCB dongle.



(a) Final draw of PCB dongle.

(b) Photo of PCB dongle.

Figure 4.5: PCB dongle.

The firmware implemented in the dongle is shown in the flowchart of Figure 4.6. The firmware starts running when the device is powered by the host through USB, that is, when the connection is made between the host (STB) and the dongle. Then, the necessary interfaces are initialized in the microprocessor to communicate with modules. The RF4CE and MiWi RF modules uses SPI and BLE uses the UART interface. The USB interface is always necessary, as it is the only way for the dongle to communicate with STB. Later the microcontroller enters in the infinite loop waiting to receive data from the RF module. When new data is received, it confirms if the debug mode is enabled and if it is sends the data via UART and flashes the LED. Although there is no hardware in the final version to enable the use of debug, that was maintained in order to enable compatibility with previous versions of hardware. When the debug is disabled, it does not increase the computational needs, since the code is not converted to binary format, and consequently not sent to the microcontroller. This was achieved with the preprocessor conditional statement shown in Code 4.1, where the instructions are only compiled if the "DEBUG_ON" macro is declared.

Code 4.1: Preprocessor conditional statement

```
1 #ifdef DEBUG_ON
2     <instructions>
3 #endif /* MACRO */
```

After receiving the data, the first byte is checked in order to know the type of information sent by RCD. This verification is always required in order to know for which USB HID interface the data should be sent.

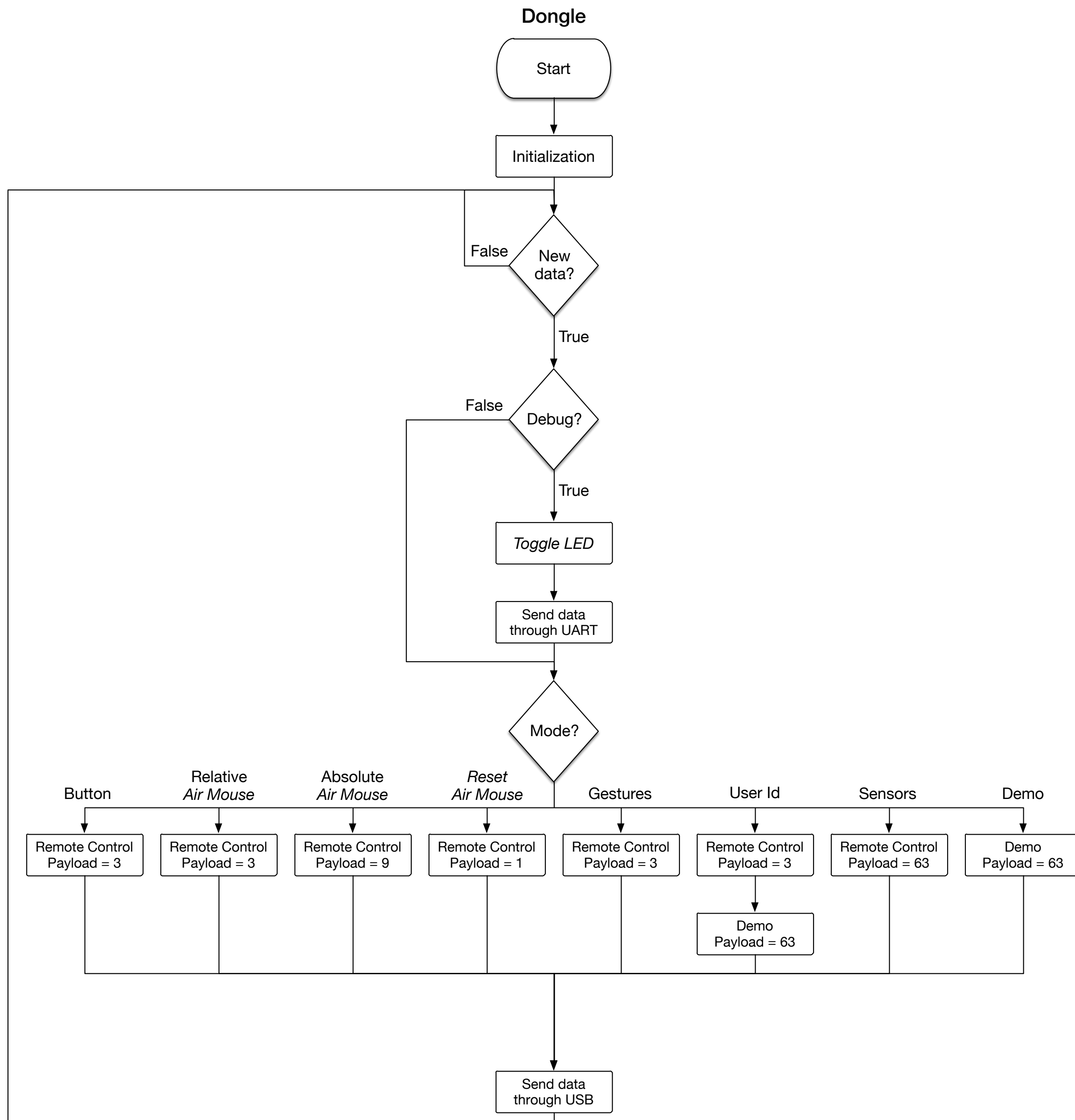


Figure 4.6: Dongle flowchart.

The communication modules are discussed in greater detail in the following sub sections, "Communication between RCD and dongle" and "Communication between dongle and STB".

4.1.1 Communication between RCD and dongle

The communication with the RCD was initially performed with the MRF24J40MA module (Figure 4.7(b)), which is a RF transceiver at 2.4 GHz ISM band, developed by Microchip Technology Inc. This module consists in the MRF24J40 chip that running at 20 MHz, which communicates with the dongle through the SPI protocol and receives data through the MiWi™ protocol.

Then the ZigBee® RF4CE communication was chosen, which included the CC2531F256 chip, that is a SoC operating at 24 MHz. Its main application is also intended for RF 2.4 GHz communications and it is developed by Texas Instruments. This chip was embedded in a module developed by Tech4Home, as shown in Figure 4.7(a), who was in charge of adding the antenna and the electronics required for its operation, as well as the firmware. This connects the dongle through the SPI protocol to perform the transfer of the received data.

During the development phase the BLE module was used for proof of concept. However, this was not included in the final version of the remote control, given the fact that it does not implement this protocol. The selected module was RN4020 from Microchip Technology Inc., which appears in Figure 4.7(c). This module, uses version 4.1 of the Bluetooth and the communication with the dongle is done through UART by American Standard Code for Information Interchange (ASCII) commands.

(a) Tech4Home module with ZigBee® RF4CE protocol.

(b) MRF24J40MA module from Microchip Technology Inc. with MiWi™ protocol.

(c) RN4020 module from Microchip Technology Inc. with BLE protocol.[1]

Figure 4.7: Radio Frequency modules.

[1] From: `http://www.microchip.com/_images/ics/small-RN4020-MODULE-22.png` (visited on 29 August 2015)

4.1.2 Communication between dongle and STB

The communication with the STB was done by USB through HID. For this purpose the five interfaces were established as described below, which fall into two categories, Customs (Demo and Remote Control) and Sensors (Accelerometer, Gyroscope and Magnetometer).

Custom (Demo) is a frame with a maximum of 64 byte length to send information for the Android application. This information follows the Input Report format shown in Figure 4.8.

field	Code	qW	qX	qY	qZ	Gesture	Roll	Pitch	Yaw	User ID
Bits	8	Variable								8

Figure 4.8: USB HID Custom - Demo Input Report format

Custom (Remote Control) is a frame with also a maximum of 64 byte length to send information for the API.

Accelerometer, Gyroscope, Magnetometer are three interfaces that represents the sensors and each have two types of reports: Feature and Input.

The implementation of this interface was made with the firmware of project "Device - HID - Custom Demos" from Microchip Libraries for Applications (MLA) version "microchip_solutions_v2013-06-15"[1] as a base. The main deadlock of this project were the dependencies with several libraries across multiple folders. There were only available to edit the "usb_descriptors.c." and the "usb_config.h.", as shown in Figure 4.9, that do not allow to add more interfaces neither Features Reports.

After getting clean and independent project files from the library, the project was ready for the changes. Since the project comes with one custom interface and it is needed two, the next step was to add another custom interface. Note that the custom interfaces, only requires the payload size, since the fields do not have a predefined meaning for the OS, so it was not necessary to modify the custom interface that comes in the project. To add a new interface it was needed to define in "usb_config.h" file, the follow parameters: ID of Interface, Number of Endpoints, ID of Endpoint, Input/Output Report size, Interval between reports, Number of Descriptors, Configuration Report Size and modify the Maximum number of interfaces and endpoints. Then, the information about this interface was added to the configuration descriptor, along with the input report descriptor in the file "usb_descriptors.c". However, given that the library only supports an interface, the Input Report is never sent. It is needed to change the "usb_function_hid.c" inside the

[1] http://www.microchip.com/pagehandler/en-us/devtools/mla/legacy-mla.html

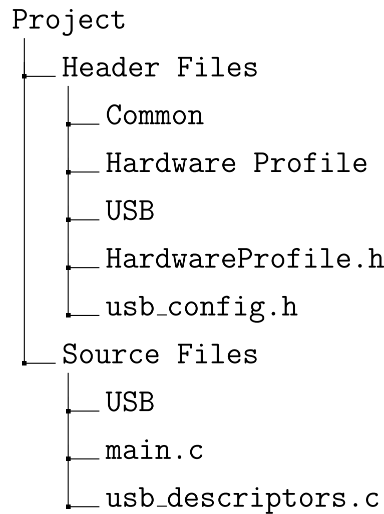


Figure 4.9: Device - HID - Custom Demos directory tree.

"USB" in the source directory to indicate the descriptors location when the request is made by the host, and is also required to initiate the endpoints.

The sensors interface requires the same previous steps, plus the changes to enable the Feature Reports. To add this report it is necessary to change the file "usb_function.c" to implement an handler to set/get the reports, in order to decide the action for each type of report (Input/Output/Feature).

Although the sensors were implemented in the dongle, they were only partially tested. For the test a digital USB analyser was used, where it was possible to check the packets from the Input Report for all sensors and all sensors were recognized by the Windows and Linux OS. The kernel from the Android OS does not implement the USB HID Sensors, so it is impossible to test, although the kernel of the new version of Android OS for 64 bit microprocessor should implement this interface. According to the source code available from Google, all versions later than 3.4 implement this interface, as can be seen by the presence of the file "hid-sensor.txt" in "android-3.X/Documentation/hid/" folder[1]. To overcome this problem for the previous version of kernel in the Android OS, it was necessary to implement a library for sensors, that will be explained in the next section, to provide the values from the RCD sensors to the Android OS.

4.2 Android API

The STB is based on Android OS and the communication is made through an USB dongle, although it is not possible to use the HID software stack to implement all the interfaces (e.g., absolute mouse HID) because the data must be pre-processed before

[1] https://android.googlesource.com/kernel/common.git/

making it available to the OS. Another limitation imposed by the OS, is the impossibility of getting the mouse pointer coordinate values when absolute coordinates are used. The proposed system was designed to have the least possible impact on the STB in order to avoid the need for a custom build, *i.e.* avoid the need for recompiling the kernel or the Android OS. The Hardware Abstraction Layer (HAL) in the user space revealed to be the adequate place for the implementation of the data processing module, the API and the sensor library, because it is not hardware-dependent and allows receiving data from any communication interface (*e.g.* Bluetooth, I2C, SPI, etc.), as shown in figure 4.3.

The main purpose of the API is:

Option "A" to integrate the processed sensors data from RCD with screen information from the user's setup.

Option "B" to perform all the heavy processing that requires a great deal of power consumption in the RCD and integrate the result data with screen information from the user's setup or the make the raw data from sensors available to the Android OS trough the library specifically developed for this RCD.

As proof of concept, a user interface application for Android OS was developed to show the RCD ability to control 3D multimedia content. This application implements an USB service to receive data from either a virtual or physical USB connection, one user interface with menus and 3D multimedia content developed with Open Graphics Library (OpenGL).

Both implementations in STB are described in the following sub sections. First the API and sensors library are explained, which are located on the underside of the stack (Figure 4.3), followed by the user interface application.

4.2.1 Application programming interface

The API was developed in the C programming language as a native application. For the development of such applications there is a Native Development Kit (NDK) available for Android devices. However this was not used, since it requires an application in Java for Android to run and start the native application. Thus, only the GNU Compiler Collection (GCC), available with NDK, was used to build the native application developed for the Android device. The Appendix A shows how to compile the Android Open Source and how to build the native application.

The application flowchart can be observed in figure 4.10. The native application must start when the device is powered, so it was necessary to make changes in the bootloader, to load the API as a service. Those changes consisted in editing the "init.< *device* >.rc"[1]

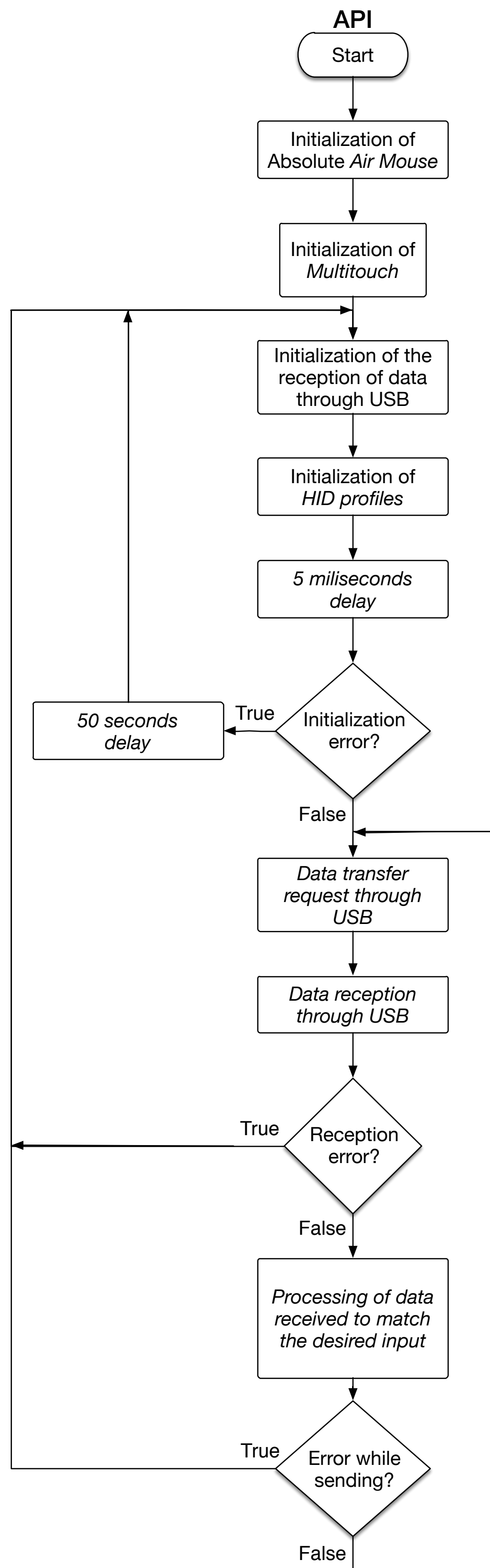


Figure 4.10: API flowchart.

file (code shown in code 4.2), followed by the build[2] and flash[2] of the boot image into the device [45].

Code 4.2: init.< *device* >.rc

```
1 ...
2 service hermes /data/local/API/REMOTE6DOF
3     class main
4     user root
5 ...
```

Given that the API is in the HAL level, it is possible to declare HID profiles and inputs that make the data available for the entire OS and respective applications at higher layers. Figure 4.11 shows the interaction between the blocks of the API in the Android software stack.

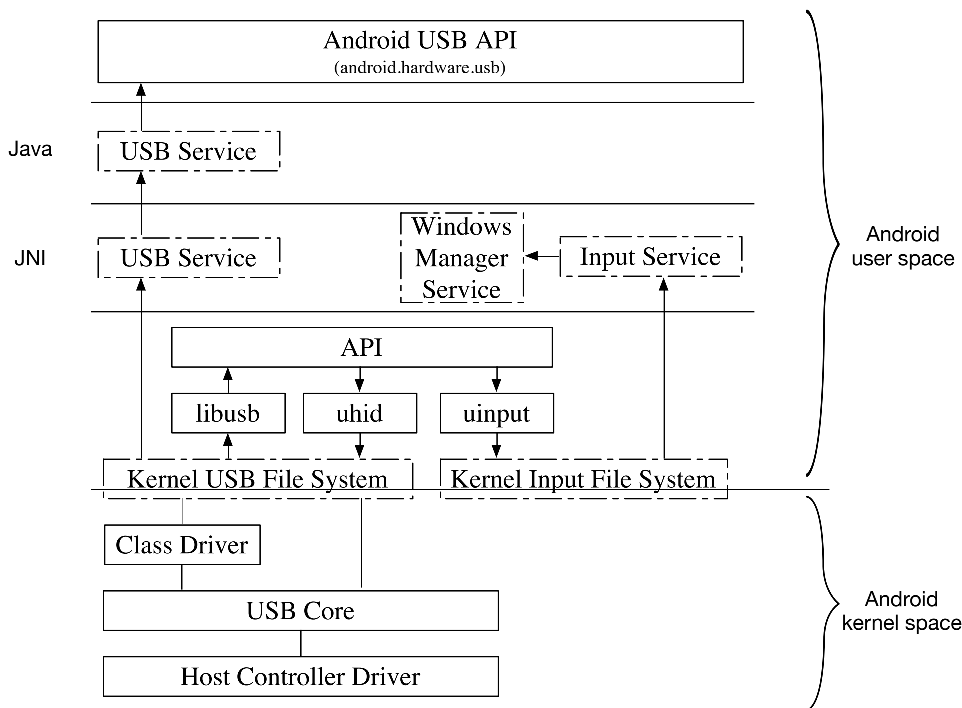


Figure 4.11: Android API Stack[3].

[1] Where <device> parameter contains the code name of device.
[2] The boot image was compiled trough the steps presented in sub section Build bootimage and Flash Android device with custom bootimage in Appendix A.
[3] Adapted from: Rajaram Regupathy. Unboxing Android USB: A hands on approach with real world examples. Apress, May 2014

After starting the API, five input profiles on the Android OS are declared and initialised: the pen, the multitouch, accelerometer, gyroscope and magnetometer inputs. These profiles were created using a module in user space to create and handle the input devices, *i.e.* an "uinput" kernel module. To create a new virtual device (e.g., the multitouch input device, as shown in Code 4.3), the following sequential actions have to be taken:

1 Open the user interface ("/dev/uinput") and create a temporary device;

2 Publish which input events the device will generate;

3 Create a structure with the basic information of device, namely the maximum and minimum values for the input events;

4 Send the command to the interface to create the device.

The pen input device was implemented to overcome the Android OS limitation of not making the pointer visually available for the absolute mouse. Two conditions have to be fulfilled in order to make the pointer visible on the screen: (i) explicit configuration of the requirement for a pointer and (ii) claim that the pen is in the range of the screen.

Code 4.3: Creation of multi-touch input

```
1 uimt->fd_uinput = open("/dev/uinput", O_WRONLY | O_NONBLOCK);
2 if(uimt->fd_uinput < 0) die("error: open");
3
4 if(ioctl(uimt->fd_uinput, UI_SET_EVBIT, EV_KEY) < 0) die("error: ioctl");
5 if(ioctl(uimt->fd_uinput, UI_SET_KEYBIT, BTN_TOUCH) < 0) die("error:
      ioctl");
6
7 if(ioctl(uimt->fd_uinput, UI_SET_EVBIT, EV_ABS) < 0) die("error: ioctl");
8 if(ioctl(uimt->fd_uinput, UI_SET_ABSBIT, ABS_X) < 0) die("error: ioctl");
9 if(ioctl(uimt->fd_uinput, UI_SET_ABSBIT, ABS_Y) < 0) die("error: ioctl");
10 if(ioctl(uimt->fd_uinput, UI_SET_ABSBIT, ABS_MT_POSITION_X) < 0)
      die("error: ioctl");
11 if(ioctl(uimt->fd_uinput, UI_SET_ABSBIT, ABS_MT_POSITION_Y) < 0)
      die("error: ioctl");
12 if(ioctl(uimt->fd_uinput, UI_SET_ABSBIT, ABS_MT_TRACKING_ID) < 0)
      die("error: ioctl");
13 if(ioctl(uimt->fd_uinput, UI_SET_ABSBIT, ABS_MT_SLOT) < 0) die("error:
      ioctl");
14
15 memset(&uidev, 0, sizeof(uidev));
```

```
16 snprintf(uidev.name, UINPUT_MAX_NAME_SIZE, name);
17 uidev.id.bustype = BUS_USB;
18 uidev.id.vendor = 0x1;
19 uidev.id.product = 0x1;
20 uidev.id.version = 1;
21 uidev.absmax[ABS_X] = max_x;
22 uidev.absmax[ABS_Y] = max_y;
23 uidev.absmax[ABS_MT_POSITION_X] = max_x;
24 uidev.absmax[ABS_MT_POSITION_Y] = max_y;
25 uidev.absmin[ABS_MT_TRACKING_ID] = -1;
26 uidev.absmax[ABS_MT_TRACKING_ID] = TRKID_MAX;
27 uidev.absmax[ABS_MT_SLOT] = num_slots - 1;
28
29 if(write(uimt->fd_uinput, &uidev, sizeof(uidev)) < 0) die("error: write");
30 if(ioctl(uimt->fd_uinput, UI_DEV_CREATE) < 0) die("error: ioctl");
```

The pen and multitouch virtual input devices are initialized taking into account the screen size of the device where it is running. This is done by reading the resolution field "FBIOGET_VSCREENINFO" of the framebuffer "/dev/graphics/fb0". The accelerometer, gyroscope and magnetometer were implemented to provide the values from the sensors to the library that is explained later. The relative mouse, joystick, gamepad and consumer eletronic virtual USB HID devices, are created through a similar procedure but this time, using an USB interface in user space "/dev/uhid".

Using the "libusb"[1] the USB is started as host in order to receive data from the dongle. If there is an error during any of the initializations, the API waits 50 seconds and tries again, repeating the process until there is no error. This ensures that the API only continues after establishing a proper connection with the dongle. It should be noted that the device may not be connected when the API is started, so through this cycle, it can be ensured that the device is detected with a maximum delay of 50 seconds from the connection. When no errors are detected, it initializes the request for information to the dongle and the data is read from the USB buffer. Then, the received data pass through an error checking, and if an error occurs while receiving or sending data, a soft reset is performed by software, leading to an API reinitialisation which ensures that there is no accumulation of errors. If there are no errors, these data are handled and sent to the corresponding USB HID profiles and user Inputs of Android OS. The HID and the user input are automatically processed by the Android OS, with the exception for the sensor input, that are processed by the library specifically developed for the RCD used. This library is explained in following sub section Android sensors library.

[1] http://libusb.info

4.2.2 Android sensors library

The Android sensors library was developed using the library provided by Asahi Kasei Microdevices Corporation[1] as base. The instructions to compile the library are presented in section "Build bootimage " of Appendix A. This library was defined as the default library for the sensors, to avoid the conflict with the existing library for the device, given that the "ServiceManager" of the Android Framework checks the path "/system/lib/hw" of Android OS, to find if there are present the following sensors library [54]:

- sensors.default.so
- sensors.<device>.so[2]

To add support for sensors from the RCD, the first step was to define list of sensors implemented in the "sensor.cpp" file (Code 4.4), where for each sensor it was needed to defined the following fields (e.g., Accelerometer sensor):

Code 4.4: sensor.cpp

```
1 ...
2 name       = ''3-axis Accelerometer'',
3 vendor     = ''Tech4Home/IPL - Ricardo Santos'',
4 version    = 1,
5 handle     = SENSORS_ACCELERATION_HANDLE,
6 type       = SENSOR_TYPE_ACCELEROMETER,
7 maxRange   = 19.62f,
8 resolution = 19.62f / 32768,
9 power      = 0f,
10 minDelay  = 100000,
11 reserved  = {}
12 ...
```

Then was needed to create a custom class for each sensor that extends the "SensorBase" class and has the ability to read the input events defined in the previous section 4.2.1. Each sensors class finds the respective event in the system, by searching for the event with the name pre-defined in the class, i.e., the class accelerometer, gyroscope and magnetometer will search for the event with name "AccelerometerTech4Home", "GyroscopeTech4Home" and "MagnetometerTech4Home" respectively. The Unified Modeling Language (UML) structure for the custom sensors class implemented are presented in Figure 4.12.

[1] From: `https://android.googlesource.com/platform/hardware/akm/+/lollipop-release/AK8975_FS/libsensors/` (visited on 17 September 2015)

[2] Where <device> parameter contains the code name of device

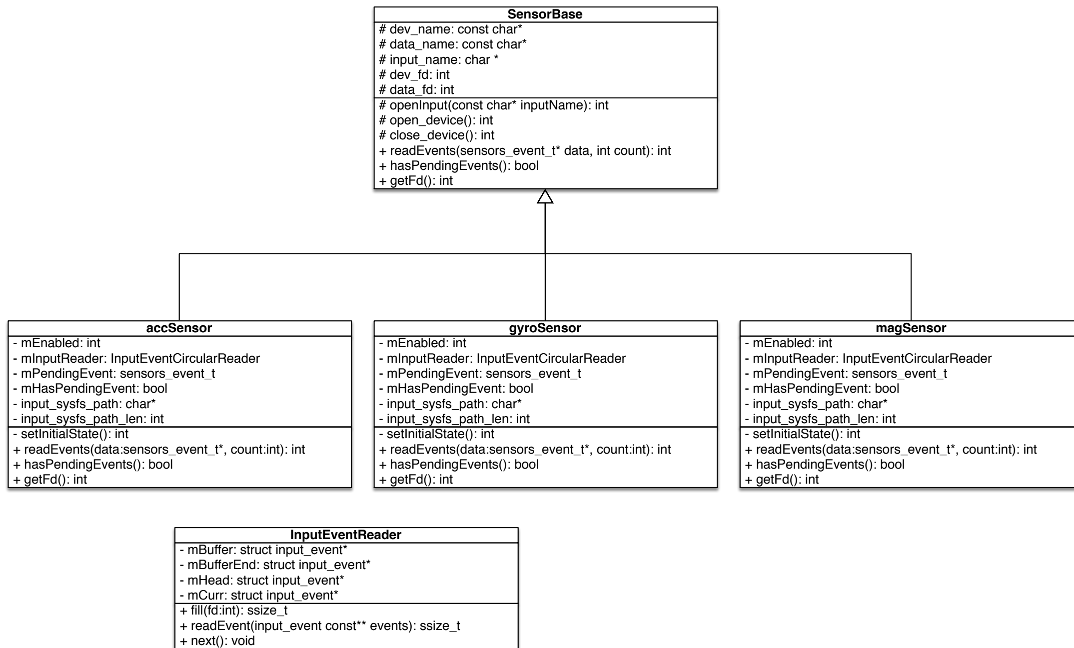


Figure 4.12: Android Sensors UML Structure[1].

The next step was to create a poll context in the "sensor.cpp" file, in order to inform the Android OS to receive the previous input events found. Those event sends the x, y and z values for each sensor, then the correspondent class makes the matching between the user input values received and the sensors values for the Android OS.

The flowchart in Figure 4.13[2], show how the sensor service (SensorService.cpp) initialize the sensors class (sensors.h and sensors.cpp) implemented in this project. The sensor service will start by searching for the sensors library in the system, after find the sensor library, it will request to initialize the sensor (in the "Sensor.cpp" file). The activation/deactivation and the sample frequency for the sensor was not dynamically implemented, the sensors are always active with a constant sample frequency [54].

All the three sensors were tested in Android OS and they work with either the applications available in Play Store as well as the OS itself, e.g., to rotate the screen orientation. The successful implementation of hardware-based sensors showed also the virtual sensors described in the previous Chapter The Universal Serial Bus (USB) in Android, Section 3.2.4. However, the use of the three hardware-based sensors reveal some delay, that was not noticeable only with the accelerometer.

[1] From: `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/DM00063297.pdf` (visited on 15 September 2015)

[2] From: `http://webcache.googleusercontent.com/search?q=cache:L3ao1uUxbMwJ:blog.pickbox.me/2014/11/06/sensors-hal/+&cd=1&hl=pt-PT&ct=clnk&gl=pt`

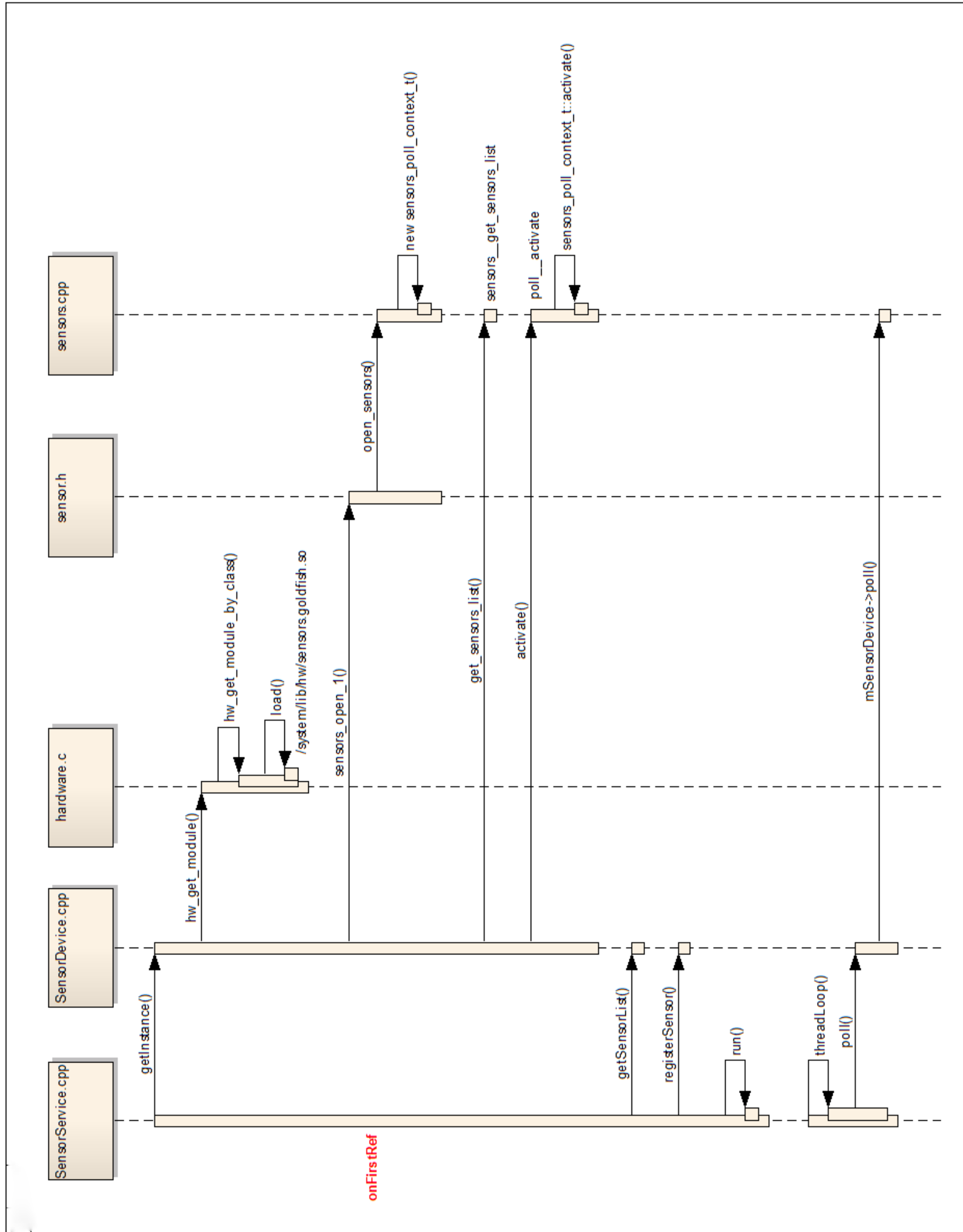


Figure 4.13: Android Sensor Service[2].

4.2.3 User interface application

The user interface application was used to evaluate and test the result of the sensor fusion system in the RCD from the point of view of use in applications. This has a simple user interface with only one main menu from which it is possible to access new screens that allow different actions. The main menu was created with a style popularized by Android, that is a drawer menu, which is hidden on the left side of the screen. The access to the menu can be done by simply clicking on the application icon at the top left of screen or dragging from the left outside of the screen to inside. In this menu (Figure 4.14) four different options are available: Home, 3DVisualizer, TVSimulation, Logger.

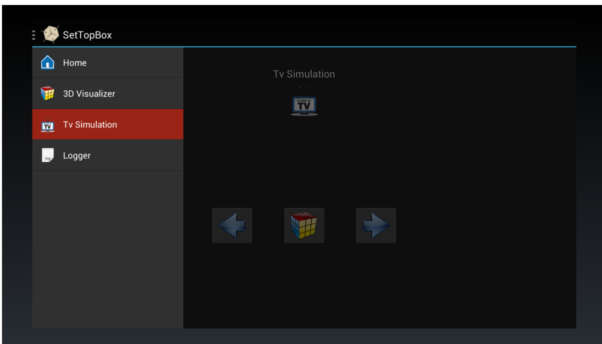


Figure 4.14: Main Menu.

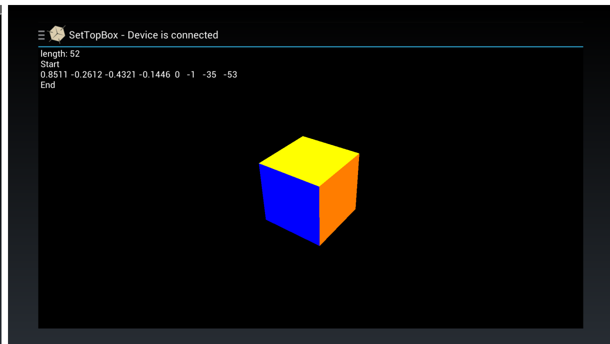


Figure 4.15: 3D Visualizer.

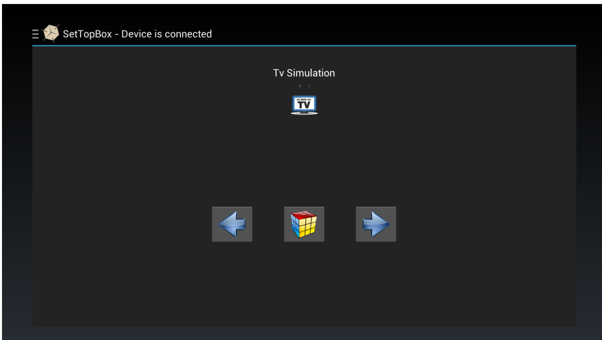


Figure 4.16: TV Simulation.

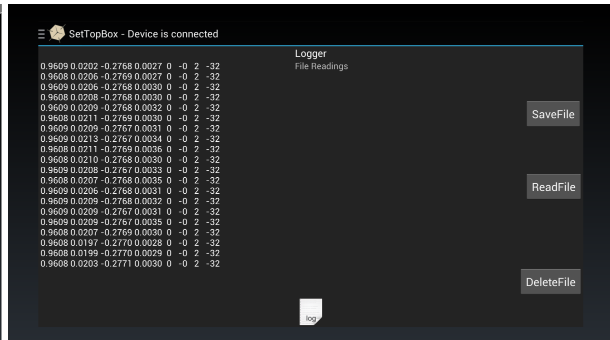


Figure 4.17: Logger.

Home is the default screen when the application starts, which only has an image that describes the project and it is similar to that shown in Figure 4.1.

3D Visualizer has at the top left of the screen a text box that displays in real time data received from the transmission of the RCD. The first four data fields represent the quaternion, the fifth corresponds to the identified gesture and the last three fields are the Euler angles (roll, pitch and yaw respectively). The center of the screen has a figures, that follow the pose the remote control.

TV Simulation aims to simulate a menu where the user can change the channels and choose the one to watch. The navigation in this menu can be performed by pressing the arrow in the screen with the mouse or through predefined gestures.

Logger allows viewing and recording the last received data. This screen is divided into three parts, on the left it shows a sequence of the last 20 received data, updated in real time. On the right side there are three buttons, that allows to save, read of delete a file, while the data is displayed in the left side of the screen.

After being defined the application interface, it was developed the background service to receive data from the RCD through USB.

USB background service for user interface application

The application was developed for using data via USB and it was decided to put all the management of the USB in a service, in order to make it completely independent of the interface presented to the user, as shown in Figure 4.18. This background service is responsible for requesting, reading and sending data for a particular device, since the application is seen as USB host. That is, when the dongle is connected, the service will check if the device is the one expected by the application. If it is, then a request is made by the service in order to be in charge of the interface.

This service has been set up so that is always running regardless of the user screen presented. However, for the service to be running, is required to have the application active. In order to detect when a device is connected, a list of the interested USB devices is declared in the file "*AndroidManifest.xml*" (Code 4.5). So, when a new device is detected among those in the filter (Code 4.6), a new instance of the application starts, if not already running. This filter, only indicates the vendor and product ID of the interested USB devices.

Since there is a bug in Android OS, each time the device is turned on, it starts a new instance of the application, regardless if it was already started, which accumulates multiple instances of the same application. This bug was bypassed through line 5 in Code 4.5, which indicates to the application, that only allows one instance on the Android OS at a time, thus removing the previous instance.

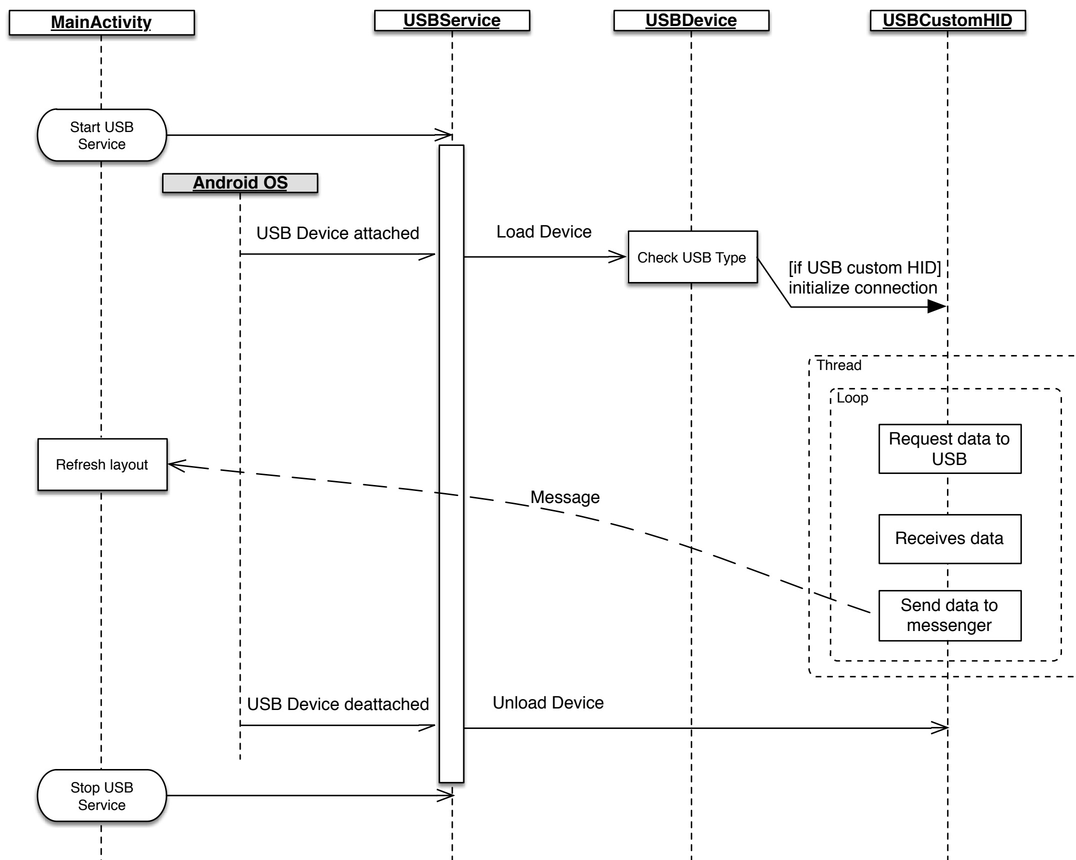


Figure 4.18: USB Block Diagram.

Code 4.5: AindroidManifest.xml

```
1 ...
2 <manifest ...>
3 <application ...>
4 <activity ...
5 android:launchMode="singleTask">
6 ...
7 <intent-filter>
8 <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
9 </intent-filter>
10 <meta-data android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
11 android:resource="@xml/device_filter" />
12 </activity>
13 <service android:name="hermes.settopbox.Brains.Services.USBService"/>
14 </application>
15 </manifest>
16 ...
```

Code 4.6: device_filter.xml

```
1 <resources>
2 <usb-device vendor-id="1240" product-id="63"/> <!-- Custom HID demo -->
3 </resources>
```

Since this service runs in background, the main process is free in order to not affect interaction with the user. During the application start, an "IntentFilter" is created, telling the system to receive a notification when any of the devices in the list is detached (Code 4.7).

Code 4.7: Receiver register

```
1 IntentFilter filter = new IntentFilter();
2 filter.addAction(UsbManager.ACTION_USB_DEVICE_DETACHED);
3 this.registerReceiver(receiver, filter);
```

During startup of the communication with USB device a new thread is created running in background as an infinite loop and managing all communication. The thread just makes a request for data (Code 4.8, line 6) and receives them (Code 4.8, line 11). Between each request a waiting period of 1 second is implemented and the request is made while receiving no answer, until the device is detached.

The received data are then passed through a verification, in which the request code has to be equal to the first value of the received data (Code 4.8, line 16). The received data has variable length, depending on the accuracy of the information sent by the device, following the order showing Figure 4.8. The information is only processed if there is an interface that needs to receive that data (Code 4.8, line 16). That data is processed and encapsulated in two different types of messages:

- The message of the text type, where the goal is to show the raw data on the screen, without any processing.
- The message of the rotation type, processes the information received and separates according to the structure of the data packet (Figure 4.8). This is a way to keep the protocol more secure, given the need to know the structure of the protocol.

After processing the received data, these are presented to the user through the previously described interface.

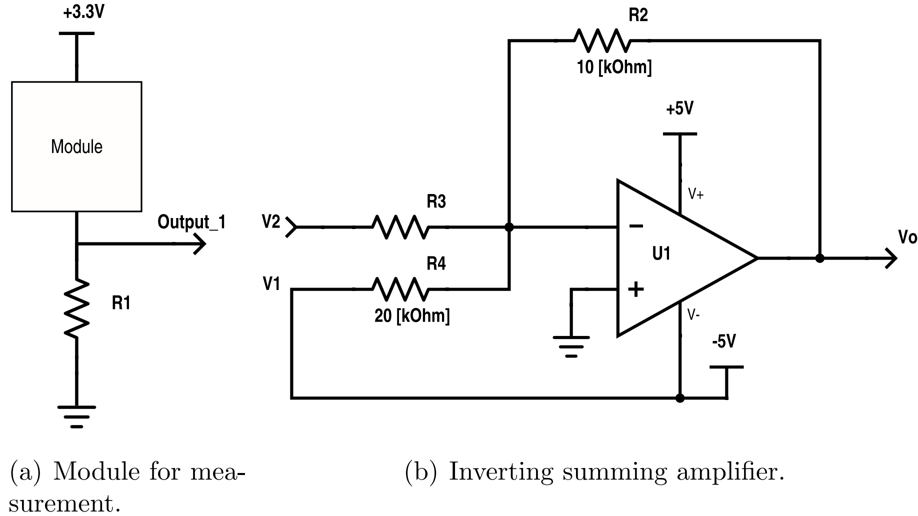
Code 4.8: Communication thread, file:USBCustomHID.java

```
1 while(true) {
2    ...
3    /* Send the request to read the Message */
4    do{
5       result = connection.bulkTransfer(endpointOUT, getRequest,
            getRequest.length, 1000);
6    } while((result < 0) && (wasCloseRequested() == false));
7
8    /* Read the Message of that request */
9    do{
10      result = connection.bulkTransfer(endpointIN, getResults,
            getResults.length, 1000);
11   } while((result < 0) && (wasCloseRequested() == false));
12
13   /* If there was data successfully read,... */
14   if(result > 0 && outMessenger != null && getRequest[0] == getResults[0])
        {
15      // Send in Text format
16      Message message = Message.obtain();
17      message.obj = new USBMessageText(new String(getResults).substring(1,
            result));
18      // Send in Rotations format
19      Message message2 = Message.obtain();
20      message2.obj = new USBMessageRotations(new
            String(getResults).substring(1, result));
21      try {
22         outMessenger.send(message);
23         outMessenger.send(message2);
24      } catch (RemoteException e) {
25         // TODO Auto-generated catch block
26         e.printStackTrace();
27      }
28   }
29 }
```

4.3 Energy consumption analysis on remote control

As described in the introduction of this chapter and presented in Figure 4.1, the system was developed with two possible solutions. In order to chose the best option (Option "A" or "B") in terms of QoE and greater battery life, an analysis on the energy consumption of each module of RCD was made. The RCD modules consists of a Microcontroller Unit (MCU), MARG sensors and a RF module.

Consumption measurements were taken at a 100 kHz sampling rate and 14 bit resolution between -2.5 and +2.5 V. As shown in figure 4.19(a), to measure the current sunk by each module, a shunt resistor (R_1) was connected to ground, thereby allowing to get the voltage drop (V_L) to compute the current. In order to use the full Analog-to-Digital Converter (ADC) range, a inverting summing amplifier with the OP37 was used (figure 4.19(b)), where the gain and the output voltage is calculated by the set of equations (4.1 - 4.4). Since the input value $V_2 * A_2$ was designed to the range between 0 and 5 volts, it was necessary to use an input voltage $V_1 * A_1$ to create the offset of +2.5 volts, in order to fit the output (V_O) within the range used by the acquisition system.



(a) Module for measurement.

(b) Inverting summing amplifier.

Figure 4.19: Measurement system.

$$A_1 = -\frac{R_2}{R_4} \tag{4.1}$$

$$A_2 = -\frac{R_2}{R_3} \tag{4.2}$$

$$V_L = V_2 \quad [\mathrm{V}] \tag{4.3}$$

$$V_O = A_1 * V_1 + A_2 * V_2 \quad [\mathrm{V}] \tag{4.4}$$

The component values used in the measurement system (Figure 4.19) to get equation 4.6, are presented in the following equations (4.5a, 4.5b and 4.5c).

$$R_2 = 10 \text{ k}\Omega \tag{4.5a}$$

$$R_4 = 20 \text{ k}\Omega \tag{4.5b}$$

$$V_1 = -5 \text{ V} \tag{4.5c}$$

$$V_O = 2,5 + A_2 * V_2 \quad [\text{V}] \tag{4.6}$$

Taking into account the maximum current indicated in the datasheet of the modules, the remaining variables present in the Table 4.1 were determined, through the set of equations 4.7a, 4.7b and 4.7c. The value of R_1 was defined to not exceed the maximum voltage drop recommended by the datasheet module.

$$V_{Lmax} = I_{max} * R_1 \quad [\text{V}] \tag{4.7a}$$

$$A_2 = \frac{5}{V_{Lmax}} \tag{4.7b}$$

$$R_3 = \frac{R_2}{A_2} \quad \Omega \tag{4.7c}$$

Table 4.1: Calculated value of the variables for each module.

	I_{max} [mA]	R_1 [Ω]	V_{Lmax} [mV]	A_2	R_3 [Ω]
Microcontroller	200	0,22	44	113,64	88
Radio Frequency	23	1	23	217,39	8,5
Sensors	13	1	13	384,62	26

Given the limited available resistance values, these are rounded to the nearest value, as can be seen in Table 4.2. Although the maximum allowed voltage in the case of the microcontroller is lower than the calculated in Table 4.1, were made tests to ensure that the voltage V_{max} does not exceed 50 mV. For all other cases the V_{max} is higher than the previously calculated value.

Table 4.2: Value of the variables used in each module.

	R_3 [Ω]	A_2	V_{max} [mV]
Microcontroller	100	100	50
Radio Frequency	68	147,06	34
Sensors	39	256,41	19,5

Taking into account the gains (A_2) obtained by the Table 4.2 and the maximum voltage (V_{Lmax}) in Table 4.1, it is possible to calculate the maximum voltage input on the amplifier (V_{Lmax}*A_2). The values are presented in Table 4.3, and the value V_{input} does not exceed 5 Volts.

Table 4.3: Maximum input voltage value in the amplifier from modules.

	V_{Lmax} [mV]	A_2	V_{input} [V]
Microcontroller	44	100	4,4
Radio Frequency	23	147,06	3,4
Sensors	13	256,41	3,3

After concluding the hardware design for power consumption measurements, a characterization of tests was made.

4.3.1 Test conditions and characterization

The measurements of energy consumption were made in three modules. In the microcontroller module energy consumption was measured for the data processing, sensor reading and RF transmission, as can be seen in Table 4.5. Data processing consists of the filter algorithm for sensors plus relative or absolute mouse estimation algorithms developed in [53]. In reading sensors were tested two modules, MPU9150 and the MPU9250, where the power consumption tests are characterized in Table 4.4. Both modules were tested in three possible settings, reading data from the DMP, reading RAW data or both. For RF transmission different sizes of payload were tested, characterized in the Table 4.6, in order to analyze the energy impact. All procedures were performed in the microcontroller at 80 Mhz, where all modules are powered at 3.3 Volt. For comparison purposes, only the consumption peaks of operations characterized in the tables were measured. They do not represent the total consumption, since the period in which the module is not doing any operations was not taken into account.

Table 4.4: Characterization of energy consumption - Sensors

			Sensors	
			MPU9150	MPU9250
100 [Hz]	DMP	128 [bits]	Test 1.1	Test 1.4
	RAW	288 [bits]	Test 1.2	Test 1.5
	DMP + RAW	416 [bits]	Test 1.3	Test 1.6

Table 4.5: Characterization of energy consumption - Data processing

				Data processing PIC32MX7xx
100 [Hz]	Reading MPU9150	DMP	128 [bits]	Test 2.1
		RAW	288 [bits]	Test 2.2
		DMP + RAW	416 [bits]	Test 2.3
	Reading MPU9250	DMP	128 [bits]	Test 2.4
		RAW	288 [bits]	Test 2.5
		DMP + RAW	416 [bits]	Test 2.6
50 [Hz]	Filters	Madgwick		Test 3.1
		Mahony		Test 3.2
		Madgwick Adaptive		Test 3.3
		Mahony Adaptive		Test 3.4
		DMP Yaw correction		Test 3.5
		Relative Mouse		Test 3.6
		Absolute Mouse		Test 3.7
	Send with MRF24J40MA	Payload	3 [bytes]	Test 4.1
			5 [bytes]	Test 4.2
			9 [bytes]	Test 4.3
			15 [bytes]	Test 4.4
			19 [bytes]	Test 4.5
			24 [bytes]	Test 4.6

Table 4.6: Characterization of energy consumption - Module RF

			Module RF MRF24J40MA
50 [Hz]	Payload	3 [bytes]	Test 5.1
		5 [bytes]	Test 5.2
		9 [bytes]	Test 5.3
		15 [bytes]	Test 5.4
		19 [bytes]	Test 5.5
		24 [bytes]	Test 5.6

4.3.2 Experimental Results

Table 4.7: Results of energy consumption - Sensors

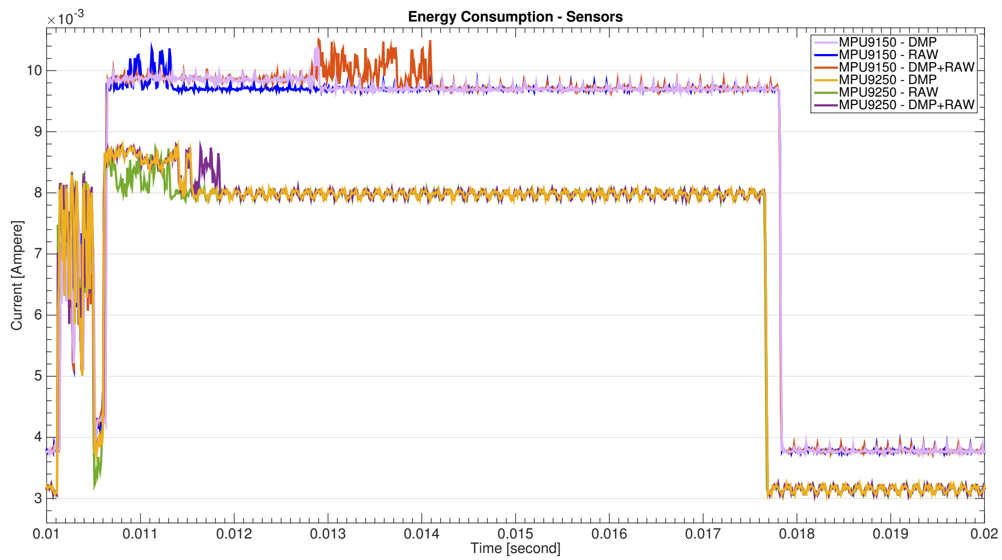
			MPU9150		MPU9250	
			Energy consumption [Joule]	Peak durantion [ms]	Energy consumption [Joule]	Peak durantion [ms]
100 [Hz]	DMP	128 [bits]	2.32008e-04	7.20	1.88291e-04	7.07
	RAW	288 [bits]	2.31050e-04	7.21	1.86139e-04	7.05
	DMP + RAW	416 [bits]	2.32282e-04	7.21	1.89113e-04	7.08

Table 4.8: Results of energy consumption - Data processing

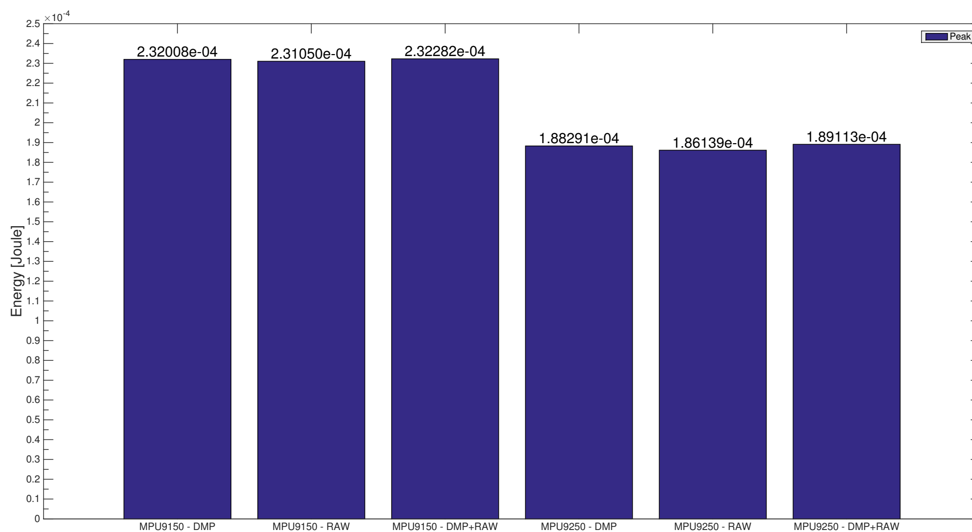
				PIC32MX7xx	
				Energy consumption [Joule]	Peak durantion [ms]
100 [Hz]	Reading MPU9150	DMP	128 [bits]	2.48291e-04	0.72
		RAW	288 [bits]	2.89949e-04	0.85
		DMP + RAW	416 [bits]	4.76695e-04	1.38
	Reading MPU9250	DMP	128 [bits]	2.45270e-04	0.71
		RAW	288 [bits]	3.01690e-04	0.82
		DMP + RAW	416 [bits]	4.81338e-04	1.39
50 [Hz]	Filters	Madgwick		1.41394e-04	0.38
		Mahony		7.29484e-05	0.20
		Madgwick Adaptive		2.14523e-04	0.57
		Mahony Adaptive		1.06858e-04	0.29
		DMP Yaw correction		6.89736e-05	0.19
		Relative Mouse		3.50383e-05	0.10
		Absolute Mouse		9.15764e-05	0.25
	Send with MRF24J40MA	Payload	3 [bytes]	2.41117e-05	0.07
			5 [bytes]	2.75855e-05	0.08
			9 [bytes]	3.80335e-05	0.11
			15 [bytes]	4.49740e-05	0.13
			19 [bytes]	4.87658e-05	0.14
			24 [bytes]	5.58275e-05	0.16

Table 4.9: Results of energy consumption - Module RF

			MRF24J40MA	
			Energy consumption [Joule]	Peak durantion [ms]
50 [Hz]	Payload	3 [bytes]	5.35162e-05	0.58
		5 [bytes]	5.97764e-05	0.65
		9 [bytes]	7.00594e-05	0.77
		15 [bytes]	8.90354e-05	0.97
		19 [bytes]	1.00867e-04	1.10
		24 [bytes]	1.17342e-04	1.26

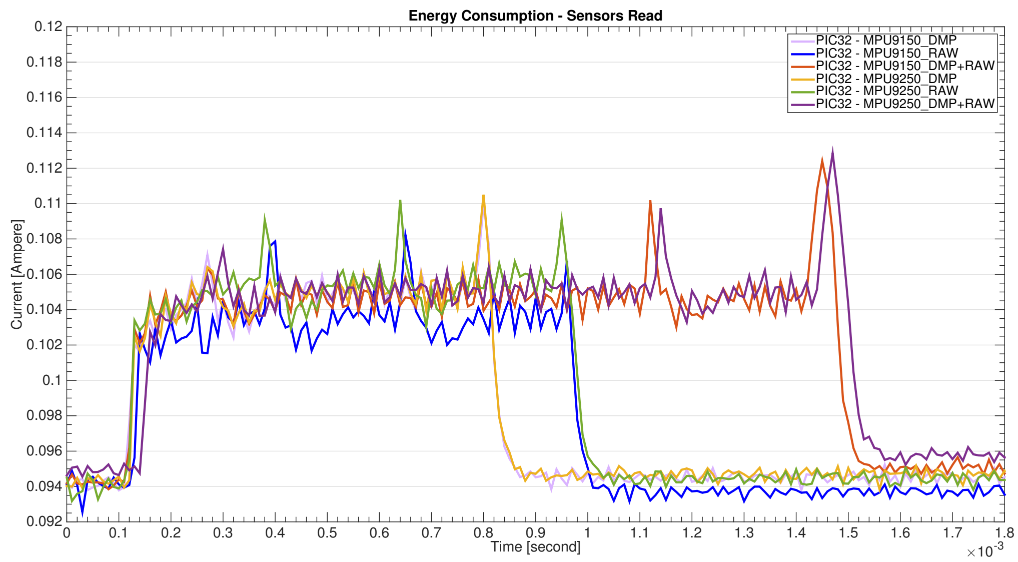


(a)

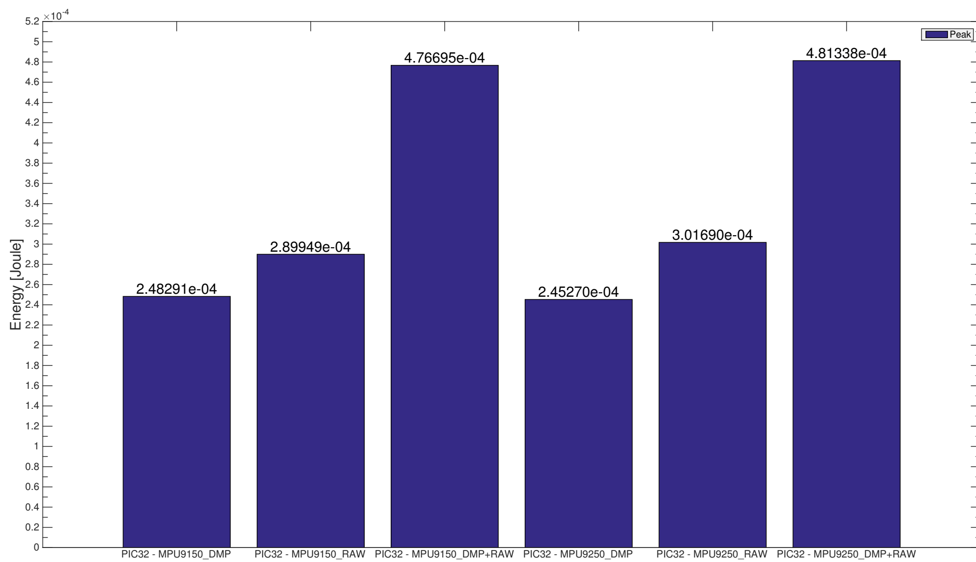


(b)

Figure 4.20: Results of energy consumption - Sensors.

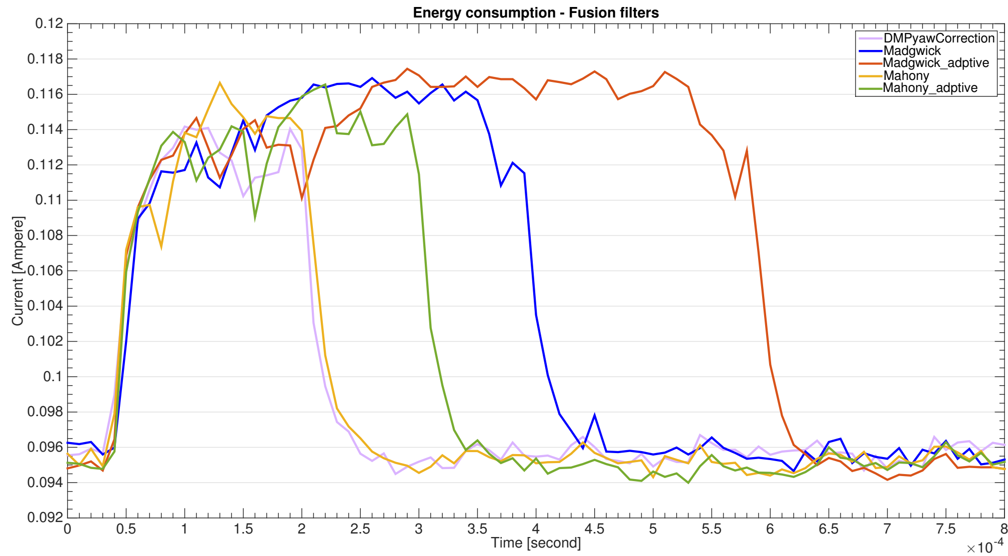


(a)

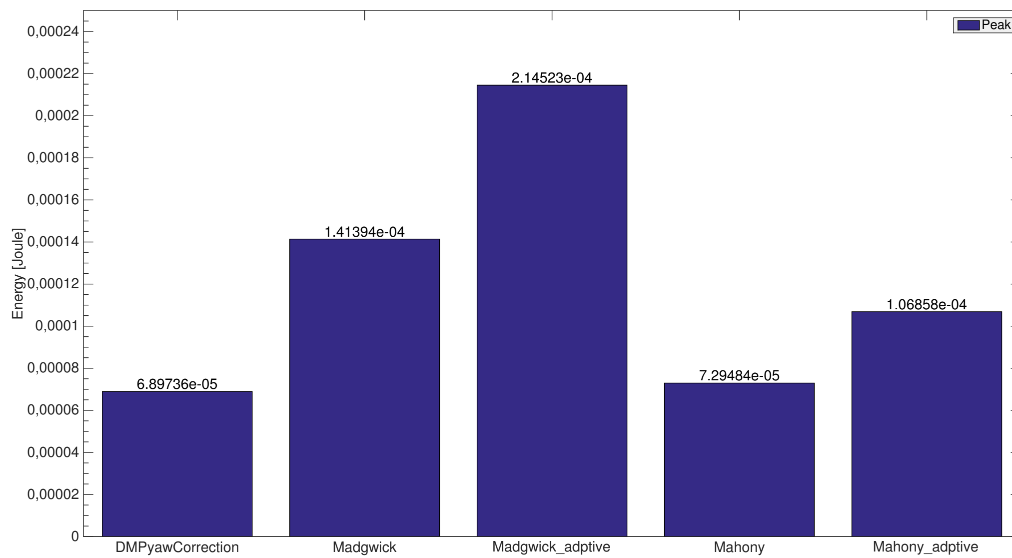


(b)

Figure 4.21: Results of energy consumption - Reading Sensors.

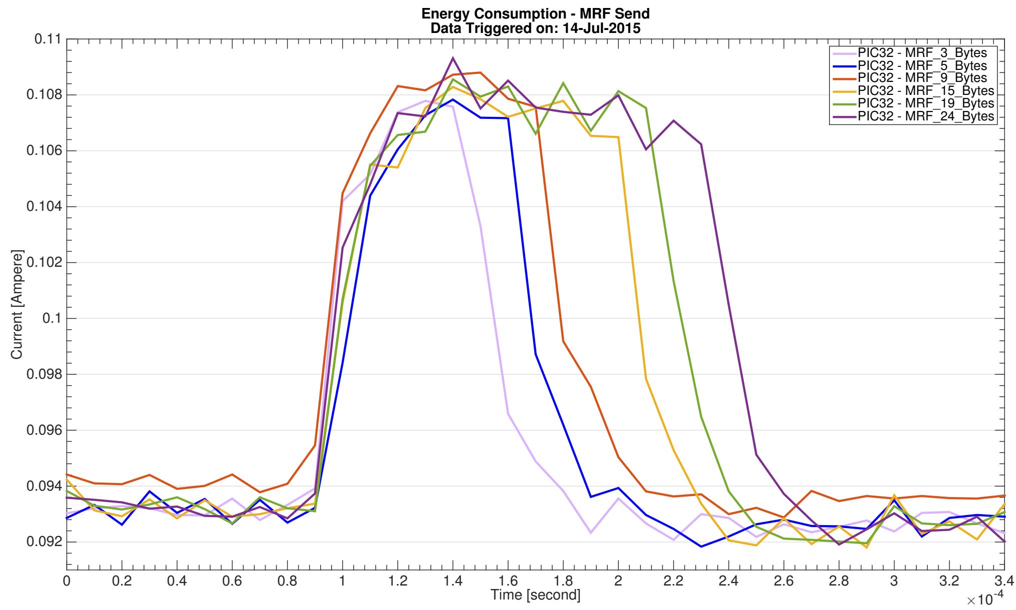


(a)

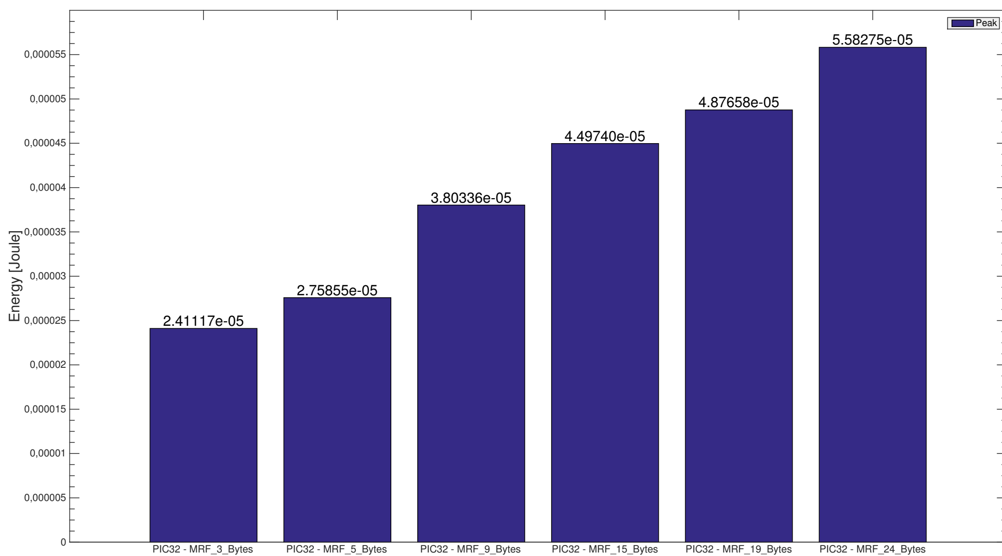


(b)

Figure 4.22: Results of energy consumption - Fusion Filters.

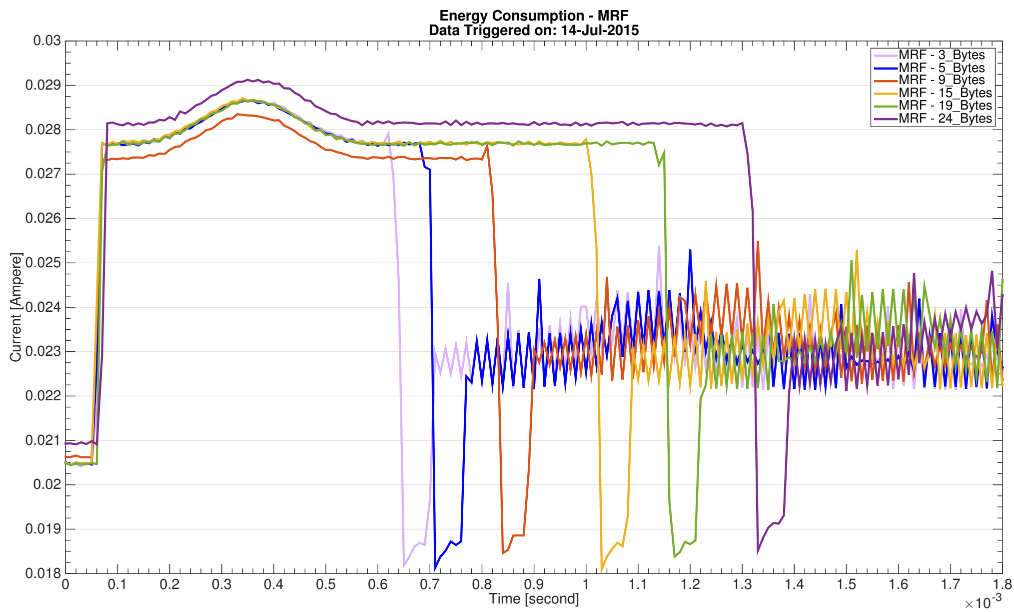


(a)

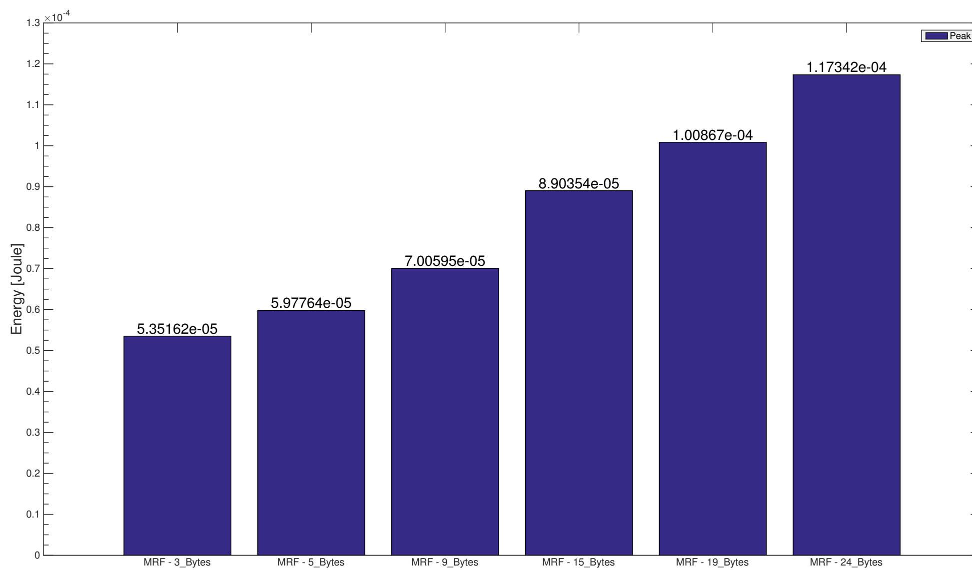


(b)

Figure 4.23: Results of energy consumption - Send with MRF24J40MA.



(a)



(b)

Figure 4.24: Results of energy consumption - MRF24J40MA.

4.3.3 Analysis of results

The analysis of results were performed with the implementation of five setups, comprising representative sensors data acquisition and software algorithm implementation. The QoE was taken into account in the power consumption tests, since it is affected by the pointer position refresh rate on the screen, which depends on the sensors data processing and transmission frequency. For all tested setups, the data acquisition is done at 100Hz and transmission at 50Hz. Setups 1, 2 and 3 correspond to the processing in the RCD (Option "A" in Figure 4.3). Setups 4 and 5 have the computational load in the STB (Option "B" in Figure 4.3). All setups are presented in Table 4.10.

In the first setup, the data from the sensors is acquired in RAW and, using the Mahony filter (MhF), is computed the device orientation to determine the HID relative mouse position, results are fitted in 2 bytes (X and Y) and sent via RF, with 1 byte representing the header [55]. The second setup also uses the MhF to determine the HID absolute mouse position, in this setup the results are fitted in 4 bytes (X and Y). The two added bytes arise by matching the range of values with the size of the screen, which requires at least 2 bytes for each dimension. In the setup 3 the device orientation is computed by the DMP, so it is not necessary to apply further data processing. The results are also fitted in 4 bytes (X and Y) has in the previous setup. Setup 4 consists in the acquisition of raw data and its respective transmission, i.e., 3 bytes for each sensor (Accelerometer, Gyroscope and Magnetometer), plus 1 byte for the header. In the setup 5 both DMP and RAW data are acquired, but only the orientation computed by the DMP and the Magnetometer data are transmitted. It consist on 3 bytes for each component of the DMP, 2 bytes for the Magnetometer data and 1 byte for the header.

Table 4.10: Remote control device setups and results.

Setup	Data acquisition	Computational processing	Payload [Bytes]	Energy consumption [mJ]	Peak duration [ms]
1	Gyro + Acc + Mag	MhF + *Air Mouse* Rel.	3	1.1612726	16.69
2	Gyro + Acc + Mag	MhF + *Air Mouse* Abs.	5	1.2275447	16.92
3	DMPquat + Gyro + Acc + Mag	*Air Mouse* Abs.	5	1.5198403	17.92
4	Gyro + Acc + Mag	-	19	1.1252908	16.98
5	DMPquat + Gyro + Acc + Mag	-	15	1.4749114	18.18

The results for energy consumption and the peak duration for each setup that are listed in Table 4.10, leading to the following conclusions for each test performed. Setups 1 and 2 were used to test the sensors data processing in the RCD using the RAW data, which results in the relative and absolute mouse, respectively. As expected, the absolute mouse implementation requires more energy, as it needs more processing and more data to be sent. Setup 3 implements part of the sensors data processing in the sensors module

through the DMP. Results show that it requires more energy than Setup 2, which performs data processing from the RAW sensors data in the microcontroller. In Setup, 4 RAW data is obtained from the sensors without any processing being done in the RCD. Although this results in more data to be transmitted, this Setup revealed to consume less energy than all the Setups presented above. Setup 5 uses the DMP processing in sensors module in order to transmit less data, however it consumes more energy than the Setup 4. The results listed in Table 4.10 show that there is less energy consumption on the acquisition of RAW data. Setup 4 has the lowest consumption, since no processing is done in the remote, as it consists in reading RAW data and its respective RF transmission. When the processing is performed in the RCD, acquire raw data and processing Mahony filter evidence a lower power consumption compared to acquire DMP plus RAW data from sensors in order to avoid Mahony filter processing.

The QoE was tested using an Android application, shown in Figure 4.15, specifically developed to simulate and test a potential usage environment. Also the results of the subjective tests in [53], that were carried out to evaluate how friendly is the RCD to non-expert users, revealed that absolute orientation computed in the STB presents smoother motion tracking and good user experience.

Chapter 5

Conclusions and Future Work

The recent advances in STB and television technology, such as the ”smart TVs” and the integration of Android, allows to deliver more multimedia content to the user. Although the remote control devices in the market to interact with the multimedia content, displayed in the television, stills needs some improvements to be more intuitive to the user.

This work had as main objective the development of an Hardware/Software interface for enhanced remote control devices, comprising of a Remote Control Device (RCD), Dongle, STB with Android OS.

In this dissertation, the remote control device used was developed by Miguel Rasteiro and Tech4Home [53]. This device has novel characteristics, such as 6 Degrees of Freedom through a set of sensors commonly known as MARG sensors, allowing an absolute orientation to use the device as a pointer. The sensors in remote control device in addition to have a reduced frequency of calibration and to enabling gesture-based controls, can also be used to play games.

Those advances in technology allows improved user interaction in the digital world, although, battery life versus computational power can still be problematic. This work studied the current wireless communications solutions for low energy devices that are in the market for communication between the STB and the RCD. All the low energy solutions are simpler versions of the one that exist previously. Since one solution is not the best for all situations, they should be chosen for the main role that will be used. In the CE devices the most used are the BLE and RF4CE.

There were already some techniques for integration of devices with sensors on the Android operating system, however these techniques have some limitations, such as dependencies with either hardware or software. Although, the “Sensor Emulation initiative for virtualized Android-x86” had the closest approach of the desired [8].

The USB 2.0 communication with HID and Sensors interfaces was also implemented

between Dongle and STB. The USB demonstrated to be a solid architecture with a very well defined protocol. The interfaces allowed to customize and adapt the functionality of the device without the need to change the host. This implementation is recognized as very good pattern for development in such way that the Bluetooth® already adopted it, as HID Over GATT Profile (HOGP), and also the ZigBee® RF4CE as ZigBee® Input Device (ZID). However, if it is necessary to perform some processing in the data received by the host before they are available to the OS, the procedure becomes more complicated. To overcome this problem an API was developed in the user space to process the data before making it available to the OS. This receives the data through the USB with a custom interface, processes the data and makes them available through virtual USB interfaces and the input services in the OS. The API allowed the processing of sensors data and the implementation of the absolute mouse and gestures.

The HID sensors interface was also implemented but was not fully tested. It was revealed that the kernel in Android OS does not yet implement the driver for this interface, preventing the tests. According to the source code documentation, there are plans to introduce this support in future releases. To overcome this problem in the current version of Android kernel, was needed to implement an sensors library specifically developed to receive the data from the API. The library was tested and successfully showed the values from the sensors in either the OS, games and the applications from the Play Store.

Although the external sensors are working in this work, the delay to get the sensors data from the RCD to the Android OS, should be analysed. As more sensors are sent, the delay will be bigger. Given that Android already includes a procedure for creating virtual sensors, when some physical sensors are present, this can help to increase the delay, which results in a poor experience for the user. The HID sensors interface can also be tested with the Android devices that already have built-in the new version of kernel with this class. This would allow an easier integration of the RCD with the Android OS for future versions. Also, the ZID profile or HOGP for wireless communications is suggested, providing the possibility to remove the dongle.

Another way to process the data, was through the USB service inside an Android application. However, this is not the best solution, because the data processed is not available outside the application. This option was used only to demonstrate the capabilities of the RCD in one STB environment. This application allowed to test the gestures to select and change channels, check the 3D orientation of RCD and record data from sensors.

Whereas the main target are low-power remote controls, energy analysis was carried out to decide whether the processing of sensor data, must be performed in the RCD or STB. In this work the energy consumption analysis revealed that the computational

processing of data should be made on the STB through the use of an API. Although it requires sending more data it consumes less energy than reading the computed orientation from the DMP or computing the orientation estimation in the RCD. In terms of energy consumption, is suggested the creation of different profiles that represent the different types of user, to analyse the total consumption of the remote control and obtain an estimate of battery life.

The overall system was successfully tested with a good user Quality of Experience and the mains goals achieved. The RCD communicates through RF4CE to the dongle developed, that forwards the data to the STB through USB with HID classes. Then it was implemented an API to process the sensors data allowing the absolute mouse and gesture or to send sensors data to the library developed allowing the use of external sensors with any application in the Android OS. Also, the simulation of the STB user interface was done through an Android application.

This page was intentionally left blank.

Bibliography

[1] C. Gritton, "What's Wrong with Smart TV?: How to Improve User Experience," *Consumer Electronics Magazine, IEEE*, vol. 2, no. 4, pp. 40–43, October 2013.

[2] J. Williamson, Q. Liu, F. Lu, W. Mohrman, K. Li, R. Dick, and L. Shang, "Data sensing and analysis: Challenges for wearables," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, Jan 2015, pp. 136–141.

[3] T. Ohnishi, N. Katzakis, K. Kiyokawa, and H. Takemura, "Virtual interaction surface: Decoupling of interaction and view dimensions for flexible indirect 3D interaction," in *3D User Interfaces (3DUI), 2012 IEEE Symposium on*, March 2012, pp. 113–116.

[4] K. Zidek and J. Pitel, "Smart 3D pointing device based on MEMS sensor and bluetooth low energy," in *Computational Intelligence in Control and Automation (CICA), 2013 IEEE Symposium on*, April 2013, pp. 207–211.

[5] M. Song, W. Xiong, and X. Fu, "Research on Architecture of Multimedia and Its Design Based on Android," in *Internet Technology and Applications, 2010 International Conference on*, Aug 2010, pp. 1–4.

[6] B. Kaufmann and L. Buechley, "Amarino: A toolkit for the rapid prototyping of mobile ubiquitous computing," in *Proceedings of the 12th International Conference on Human Computer Interaction with Mobile Devices and Services*, ser. MobileHCI '10. New York, NY, USA: ACM, 2010, pp. 291–298. [Online]. Available: http://doi.acm.org/10.1145/1851600.1851652

[7] Open intents - sensorsimulator. [Online]. Available: https://code.google.com/p/openintents/wiki/SensorSimulator (visited on 20 of September)

[8] R. Santhanam, "Enabling the virtual phones to remotely sense the real phones in real-time: A sensor emulation initiative for virtualized android-x86," Columbia University Academic Commons, Technical reports, 2013. [Online]. Available: http://dx.doi.org/10.7916/D8Z899FF (visited on 17 September 2015)

[9] W. Brunette, R. Sodt, R. Chaudhri, M. Goel, M. Falcone, J. Van Orden, and G. Borriello, "Open data kit sensors: A sensor integration framework for android at the application-level," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 351–364. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307669

[10] P. P. Mercier and A. P. Chandrakasan, *Ultra-Low-Power Short-Range Radios*, 1st ed. Springer International Publishing, 2015.

[11] *IEEE Standard 802.15.1™*. The Institute of Electrical and Electronics Engineers, Inc., June 2002. [Online]. Available: http://standards.ieee.org/getieee802/download/802.15.1-2002.pdf (visited on 30 August 2015)

[12] Y. Yang, Ed., *Microchip MiWi™ P2P Wireless Protocol*. Microchip Technology Inc., May 2008. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/01204a.pdf (visited on 7 August 2015)

[13] *IEEE Standard 802.15.4™*. The Institute of Electrical and Electronics Engineers, Inc., October 2003. [Online]. Available: http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf (visited on 7 August 2015)

[14] *IEEE 802.15.4 Wireless Networks User Guide*. Jennic, 2006. [Online]. Available: http://www.jennic.com/files/support_files/JN-UG-3024-IEEE802.15.4-1v1.pdf (visited on 30 August 2015)

[15] C. Buratti, M. Martalò, R. Verdone, and G. Ferrari, *Sensor Networks with IEEE 802.15.4 Systems*, 1st ed. Springer-Verlag Berlin Heidelberg, 2011. [Online]. Available: http://www.springer.com/us/book/9783642174896 (visited on 30 August 2015)

[16] *Understanding ZigBee® RF4CE*. ZigBee Alliance, February 2013. [Online]. Available: https://docs.zigbee.org/zigbee-docs/dcn/09/docs-09-5231-03-rmwg-understanding-zigbee-rf4ce.pdf (visited on 10 August 2015)

[17] C. Wang, T. Jiang, and Q. Zhang, *ZigBee® Network Protocols and Applications*, 1st ed. CRC Press, March 2014. [Online]. Available: https://www.crcpress.com/ZigBee-Network-Protocols-and-Applications/Wang-Jiang-Zhang/9781439816011 (visited on 30 August 2015)

[18] *ZigBee RF4CE: A Quiet Revolution is Underway*. ZigBee Alliance, December 2012. [Online]. Available: https://docs.zigbee.org/zigbee-docs/dcn/12/docs-12-0629-01-0mwg-zigbee-rf4ce-a-quiet-revolution-is-underway-webinar-slides.pdf (visited on 10 August 2015)

[19] *ZigBee and Wireless Radio Frequency Coexistence.* ZigBee Alliance, June 2007. [Online]. Available: https://docs.zigbee.org/zigbee-docs/dcn/07-5219.PDF (visited on 10 August 2015)

[20] *ZigBee RF4CE Stack User Guide*, vol. 1.2, no. JN-UG-3074. NXP Laboratories UK, January 2014. [Online]. Available: http://www.nxp.com/documents/user_manual/JN-UG-3074.pdf (visited on 10 August 2015)

[21] *ZigBee RF4CE: ZRC Profile Specification.* ZigBee Alliance, March 2009. [Online]. Available: https://docs.zigbee.org/zigbee-docs/dcn/10/docs-10-5546-00-0rsc-zigbee-remote-control-application-profile-public.pdf (visited on 30 August 2015)

[22] *ZigBee Input Device (ZID) Standard.* ZigBee Alliance, October 2012. [Online]. Available: https://docs.zigbee.org/zigbee-docs/dcn/12/docs-12-0612-01-0mwg-zigbee-input-device-standard-for-public-download.pdf (visited on 30 August 2015)

[23] *Advanced Wireless Networking (MiWi™ Protocol II).* Microchip Technology Inc., August 2013.

[24] K. Townsend, C. Cufí, Akiba, and R. Davidson, *Getting started with Bluetooth Low Energy*, 1st ed. O'Reilly Media, May 2014.

[25] R. Heydon, *Bluetooth Low Energy: The Developer's Handbook.* Prentice Hall, November 2012.

[26] C. Gomez, J. Oller, and J. Paradells, Eds., *Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology*, August 2012. [Online]. Available: http://www.mdpi.com/1424-8220/12/9/11734 (visited on 15 August 2015)

[27] *Specification of the Bluetooth® system*, vol. 0, no. 4.2. Bluetooth SIG, December 2014. [Online]. Available: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=286439 (visited on 6 August 2015)

[28] *Cypress BLE Host Component.* Cypress Semiconductor Corporation, 2014. [Online]. Available: https://www.bluetooth.org/tpg/RefNotes/BLE_Stack_RIN.pdf (visited on 15 August 2015)

[29] J. Axelson, *USB Complete: The Developer's Guide, Fifth Edition.* Lakeview Research, March 2015.

[30] Hewlett-Packard, Intel, Microsoft, Renesas, ST-Ericsson, and Texas-Instruments, Eds., *Universal Serial Bus Revision 3.1 specification*, July 2013.

[31] *USB 101: An Introduction to Universal Serial Bus 2.0*, no. 001-57294 Rev.E. Cypress Semiconductor Corporation, April 2014.

[32] R. Regupathy, *Bootstrap Yourself with Linux-USB Stack*, 1st ed. Course Technology Cengage Learning, 2012.

[33] *Universal Serial Bus Specification*, April 2000.

[34] J. Axelson, *USB Complete: The Developer's Guide, Fourth Edition*, ser. Complete Guides. Lakeview Research LLC, 2009. [Online]. Available: https://books.google.pt/books?id=pkefBgAAQBAJ (visited on 11 August 2015)

[35] *Simplified Description of USB Device Enumeration*. Future Technology Devices International Limited (FTDI), October 2009. [Online]. Available: http://www.ftdichip.com/Support/Documents/TechnicalNotes/TN_113_Simplified%20Description%20of%20USB%20Device%20Enumeration.pdf (visited on 8 September 2015)

[36] "USB nutshell." [Online]. Available: http://www.beyondlogic.org/usbnutshell/usb3.shtml (visited on 30 August 2015)

[37] *USB Protocol*. ARM Ltd. [Online]. Available: http://www.keil.com/pack/doc/mw/USB/html/_u_s_b__protocol.html (visited on 11 August 2015)

[38] *Designing a robust usb serial interface engine (sie)*. USB Implementers' Forum, Inc. [Online]. Available: http://www.usb.org/developers/docs/whitepapers/siewp.pdf (visited on 11 August 2015)

[39] *HID Usage Tables*. USB Implementers' Forum., October 2004. [Online]. Available: http://www.usb.org/developers/hidpage/Hut1_12v2.pdf (visited on 14 August 2015)

[40] J. Annuzzi, L. Darcey, and S. Conder, *Introduction to Android Application Development: Android Essentials*, 4th ed., ser. Developer's Library. Addison-Wesley, December 2013.

[41] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski, *Android hacker's handbook*. John Wiley & Sons, Inc., April 2014.

[42] "Android version codes." [Online]. Available: http://developer.android.com/reference/android/os/Build.VERSION_CODES.html (visited on 30 August 2015)

[43] *Android fundamentals*. [Online]. Available: http://developer.android.com/guide/components/fundamentals.html (visited on 18 August 2015)

[44] R. Meier, *Professional Android Application Development.* Wiley Publishing, Inc., 2009.

[45] K. Yaghmour, *Embedded Android.* O'Reilly Media, Inc., 2013.

[46] *ART vs Dalvik.* [Online]. Available: https://source.android.com/devices/tech/dalvik/ (visited on 19 August 2015)

[47] *Android Interfaces and Architecture.* [Online]. Available: http://source.android.com/devices/ (visited on 19 August 2015)

[48] R. Regupathy, *Unboxing Android USB: A hands on approach with real world examples.* Apress, May 2014.

[49] "Android: Sensors overview." [Online]. Available: http://developer.android.com/guide/topics/sensors/sensors_overview.html (visited on 15 September 2015)

[50] "Android open source project: Sensors stack." [Online]. Available: https://source.android.com/devices/sensors/sensor-stack.html (visited on 15 September 2015)

[51] "Android: Sensors types." [Online]. Available: https://source.android.com/devices/sensors/sensor-types.html (visited on 15 September 2015)

[52] *Hardware abstraction layer for Android.* STMicroelectronics, September 2012. [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/DM00063297.pdf (visited on 15 September 2015)

[53] M. Rasteiro, *Motion-based Remote Control Device for Interaction with Multimedia Content.* Polytechnic Institute of Leiria, Semptember 2015.

[54] *Environmental sensors: Hardware abstraction layer for Android.* STMicroelectronics, July 2012. [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/DM00118439.pdf (visited on 17 September 2015)

[55] M. Rasteiro, H. Costelha, L. Bento, and P. Assuncao, "Accuracy versus complexity of MARG-based filters for remote control pointing devices," in *Consumer Electronics - Taiwan (ICCE-TW), 2015 IEEE International Conference on*, June 2015, pp. 51–52.

This page was intentionally left blank.

Appendix A

Android API development tutorial

This chapter have the purpose of discribing the process to develop native aplications for Android with Standalone Toolchain from the Native Development Kit (NDK), in order to cross-compile the aplication for the target device. In this process is also described how to build the Android operating system and kernel from scratch. This is only required if it's needed to change the original files in operating system, for example, to change the init.rc file to load the application during boot.

A.1 Build steps in Linux

All this steps were performed in linux distribution Ubuntu LTS (14.04) with Nexus 7 (2013) as target device that have the code name "flo".

A.1.1 Recomended requirements

It's recomended by google the following specifications. https://source.android.com/source/building.html

- 100GB of free space available
- 8/16GB RAM
- Quad-Core processor

A.1.2 Android open source build

In order to build the source code of Android it's required to have some extra tools installed. Then we proceed to the download of source code files, in order to be able to build the

operating system. Given the fact that it's needed to make a cross-compile, that means we compile the code in one device to run in another, we have to setup the environment variables to tell to the compiler wich device we will use for the operating system. The script in source code files used for this configuration already have the option for some devices from google. After the sucessful build, flash the device with new operating system. All the steps are listed bellow.

Initializing a Build Environment

```
1  $ sudo apt-get update
2  $ sudo apt-get install openjdk-7-jdk
3  $ sudo apt-get install bison g++-multilib git gperf libxml2-utils make
       zlib1g-dev:i386 zip
```

Downloading the Source

The version downloaded was 5.0.2, but it can be another. All the versions released can be seen here:

- https://android.googlesource.com/platform/manifest/+refs

```
1  $ mkdir ~/AOSP
2  $ mkdir ~/AOSP/bin
3  $ PATH=~/AOSP/bin:$PATH
4  $ curl https://storage.googleapis.com/git-repo-downloads/repo >
       ~/AOSP/bin/repo
5  $ chmod a+x ~/AOSP/bin/repo
6  $ cd ~/AOSP
7  $ mkdir WORKING_DIRECTORY
8  $ cd WORKING_DIRECTORY
9  $ repo init -u https://android.googlesource.com/platform/manifest -b
       android-5.0.2_r1
10 $ repo sync
```

Then after getting the source code, its needed to download the driver for the device. In this case, we have a Nexus 7 [2013] (Wi-Fi) ("flo"), with the version 5.0.2 (LRX22G) of android. So we download the driver from the link:

- https://developers.google.com/android/nexus/drivers

After all the source files, its needed to setup the environment.

```
1 $ source build/envsetup.sh
2 $ lunch aosp_flo-userdebug
```

Then to compile we do:

```
1 $ make -j8 otapackage
2 $ fastboot -w flashall
3 $ fastboot reboot
```

A.1.3 Android kernel build

To build the Android kernel the steps are similar to the Android operating system. Considering that all the required tools mentioned in section A.1.2 are installed, the first step is to download all the source files. Then it is recommended to match the kernel version to build with the version in original operating system. After that, it is required to configure the environment variables to make a cross-compile for the desired device. As optional, can be enabled the option to load modules. Given that, everything is ready to compile the kernel. After the successful compilation, the kernel is moved to the folder in the source code of operating system, in order to create a boot image with the new kernel. To finish, it is only needed to flash the boot image in the device with this new one. All the steps are listed bellow.

Downloading the Source

```
1 $ mkdir ~/AOSP/WORKING_DIRECTORY_KERNEL
2 $ cd ~/AOSP/WORKING_DIRECTORY_KERNEL
3 $ git clone https://android.googlesource.com/kernel/msm.git
4 $ git clone
    https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.
5 $ echo "export
    PATH=~/AOSP/WORKING_DIRECTORY_KERNEL/arm-eabi-4.6/bin:$PATH"
```

Checking the kernel version of Android OS

In this case the device have the code name ”flo”.

```
1 $ cd ~/AOSP/WORKING_DIRECTORY/device/asus/flo-kernel/
2 $ git show
```

Advance our custom kernel to the version of Android OS

Note: The version of original kernel can be other.

```
1  $ cd ~/AOSP/WORKING_DIRECTORY_KERNEL/msm
2  $ git checkout 154bef4
```

Setup environment

The configurations were made for the Nexus 7 (2013) device with code name ”flo”.

```
1  $ export ARCH=arm
2  $ export SUBARCH=arm
3  $ export CROSS_COMPILE=arm-eabi-
4  $ make flo_defconfig
```

Enable modules (Optional)

If we want to allow the kernel to load modules, run the following script. Otherwise, skip it.

```
1  $ sed -i -e 's/# CONFIG_MODULES is not set/CONFIG_MODULES=y/g' .config
```

Compile Kernel

```
1  $ make -j8
2  $ cp ~/AOSP/WORKING_DIRECTORY_KERNEL/msm/arch/arm/boot/zImage
       /Volumes/android/WORKING_DIRECTORY/device/asus/flo-kernel/
3  $ mv kernel kernel_backup
4  $ mv zImage kernel
```

Build bootimage

```
1  $ cd ~/AOSP/WORKING_DIRECTORY/
2  $ source build/envsetup.sh
3  $ lunch aosp_flo-userdebug
4  $ make -j8 bootimage
```

Flash Android device with custom bootimage

```
1  $ fastboot flash boot out/target/product/flo/boot.img
2  $ fastboot reboot
```

A.1.4 Android native C program

The Android native application can be compiled using the standalone toolchain in the Native Development Kit (NDK). So, the first step is to download the NDK, to extract the standalone toolchain. Then the application it's created using regular text editor and compiled with the GNU Compiler Collection contained in standalone toolchain. To finish, the application is copied to the device and it's ready to be launch.

First download Android NDK to ~/AOSP/

- https://developer.android.com/tools/sdk/ndk/index.html

Setup the cross-compiler for Android devices

```
1  $ mkdir ~/AOSP/STANDALONE_TOOLCHAIN
2  $ cd ~/AOSP
3  $ ./android-ndk-r10d-linux-x86_64.bin
4  $ cd android-ndk-r10d/build/tools
5  $ ./make-standalone-toolchain.bin --system=linux-x86_64
       --toolchain=arm-linux-androideabi-4.6 --platform=android-21
       --install-dir=~/AOSP/STANDALONE_TOOLCHAIN
       --ndk-dir=~/AOSP/android-ndk-r10d
6  $ cd ~/AOSP/STANDALONE_TOOLCHAIN/bin
7  $ ./arm-linux-androideabi-gcc --version
```

Create "Hello World" native application

```
1  $ cd ~/AOSP
2  $ mkdir UHID
3  $ nano helloWorld.c
```

Application:

```
1 #include <stdio.h>
2 int main(void){
3   printf("Hello World!\n");
4   return 0;
5 }
```

Build and Run "Hello World" native application

```
1  $ cd ~/AOSP/STANDALONE_TOOLCHAIN/bin
2  $ ./arm-linux-androideabi-gcc -pie -fPIE -o ~/AOSP/UHID/helloWorld
      ~/AOSP/UHID/helloWorld.c
3  $ cd ~/AOSP/UHID
4  $ adb push helloWorld /data/local/tmp
5  $ adb shell
6  $ su
7  $ cd /data/local/tmp
8  $ ./helloWorld
```

A.1.5 Android sensors library compilation

To compile the Sensor Library first is needed to setup the environment variables:

```
1  $ cd ~/AOSP/WORKING_DIRECTORY
2  $ source build/envsetup.sh
3  $ lunch aosp_flo-userdebug
4  $ make -j8 bootimage
```

Then to compile the library is used the command "mm":

```
1  $ cd ~/AOSP/WORKING_DIRECTORY/hardware/akm/AK8975/libsensors
2  $ mm
```

Then the next step is to copy the library to the device:

```
1  $ cd ~/AOSP/WORKING_DIRECTORY/out/target/product/flo/system/lib/hw/
2  $ adb push sensors.default.so /data/local/tmp
```

In order to copy the library from the temporary folder to the Android OS is needed to remount the system with write permission:

```
1 $ adb shell
2 $ mount
```

Get the path for the system:

```
1 ...
2 /dev/block/platform/omap/omap_hsmmc.0/by-name/system /system ext4
    ro,relatime,barrier=1,data=ordered 0 0
3 ...
```

Remount the file system:

```
1 $ su
2 $ mount -o remount,rw /dev/block/platform/omap/omap_hsmmc.0/by-name/system
```

The final step is to copy the library and fix the permissions:

```
1 $ cp /data/local/tmp/sensors.default.so /system/lib/hw/
2 $ chmod 644 /system/lib/hw/*
3 $ chown root:root /system/lib/hw/*
```