This document presents the work done to update and implement improvements to a sound limiter software system, which aims to comply with acoustic regulations in all music entertainment venues, ensuring the well-being of customers, workers and nearby residents.

Throughout this paper, different phases of software development are presented: list of requirements, analysis and planning, design, implementation, testing and evaluation of the software, and even showing various user and operating manuals. To these phases must be added a previous stage, in which the student carried out a study of the work done by other students in order to fulfil the objective of the project.

Raúl Rodríguez Pérez is the student in charge of the design and implementation of the project, and with this work he finishes his degree in Computer Engineering with a specialisation in Software Engineering.

Andrés María Roldán Aranda is the academic head of the present project, and the student's tutor. He is professor in the Departament of Electronics and Computers Technologies

BACHELOR THESIS

Kernel upgrading and implementing software enhancements of IOT acoustic devices

Raúl Rodríguez Pérez

COMPUTER ENGINEERING

21/22

UNIVERSITY OF GRANADA

Degree in Computer Engineering

Bachelor Thesis

Kernel upgrading and implementing software enhancements of IOT acoustic devices

Raúl Rodríguez Pérez

2021/2022

Tutor: Andrés María Roldán Aranda

Credits for the cover: **GranaSaT**.
Printed in Granada, July 2022.

“Kernel upgrading and implementing software enhancements of IOT acoustic devices”

DEGREE IN
COMPUTER ENGINEERING

Bachelor Thesis

"Kernel upgrading and implementing software enhancements of IOT acoustic devices"

ACADEMIC COURSE: 2021 - 2022

Raúl Rodríguez Pérez

DEGREE IN COMPUTER ENGINEERING

"Kernel upgrading and implementing software enhancements of IOT acoustic devices"

AUTHOR:

Raúl Rodríguez Pérez

SUPERVISED BY:

Prof. Andrés Roldán Aranda

DEPARTMENT:

Electronics and Computers Technologies

D. Andrés María Roldán Aranda, Profesor del departamento de Electrónica y Tecnología de los Computadores de la Universidad de Granada, como director del Trabajo Fin de Máster de D. José Carlos Martínez Durillo,

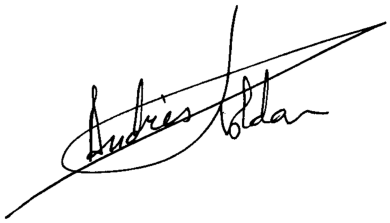
Informa:

Que el presente trabajo, titulado:

"Kernel upgrading and implementing software enhancements of IOT acoustic devices"

ha sido realizado y redactado por el mencionado alumno bajo mi dirección, y con esta fecha autorizo a su presentación.

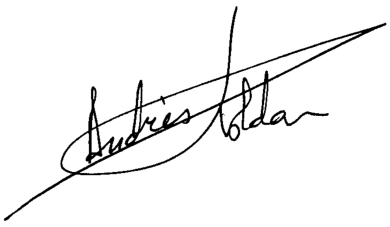
Granada, a 2 de septiembre de 2022



Fdo. Prof. Andrés María Roldán Aranda

Los abajo firmantes autorizan a que la presente copia de Trabajo Fin de Máster se ubique en la Biblioteca del Centro y/o departamento para ser libremente consultada por las personas que lo deseen.

Granada, a 2 de septiembre de 2022



Fdo. Raúl Rodríguez Pérez

Fdo. Prof. Andrés María Roldán Aranda

“Kernel upgrading and implementing software enhancements of IOT acoustic devices”

Raúl Rodríguez Pérez

KEYWORDS:

ABSTRACT:

The main objective of this project is to update the kernel of the software system of some sound limiters. In addition, it is also intended to carry out a series of modifications or improvements to this software, with the aim of achieving better performance.

This project consists of numerous phases that are present in any software development work. But to summarise, we can divide it into three large blocks: analysis and planning of requirements, design and implementation of the software. During the first period, we specify what our software is going to be able to do, and we also study how it is going to be carried out, that is to say, what market tools we have available to achieve the objective, and which one we are going to choose. Then, with the conclusions resulting from the previous phase, the design of the system will be carried out, and finally, the implementation and testing of the final software product.

This bachelor’s thesis should be seen as a piece of a larger machine, as is part of a much larger, more ambitious and far-reaching project. Furthermore, I should point out that this work is based on the results achieved by various students in their respective projects, and not only in the field of computer science, but also in fields such as electronics and telecommunications. The complexity and multidisciplinary scope of being part of this great project has led me not only to cover different specialities within the Degree in Computer Engineering, but I have also acquired specific knowledge and skills in other fields of engineering such as Electronics and Acoustics. I must even highlight my improvement in my intrapersonal and interpersonal skills, by collaborating and working as a team with other students to carry out the project.

Therefore, the result of all of the above culminates in the updating and implementation of improvements to the software of the sound limiters, thus fulfilling the requirements defined in the initial stages of the project, and with which my university stage in the Degree in Computer Engineering comes to an end.

“Kernel upgrading and implementing software enhancements of IOT acoustic devices”

Raúl Rodríguez Pérez

PALABRAS CLAVE:

RESUMEN:

El objetivo principal del presente proyecto se basa en conseguir llevar a cabo la actualización del kernel del sistema software de unos limitadores de sonido. Y además, también se pretende realizar una serie de modificaciones o mejoras a dicho software, con el propósito de conseguir un mejor rendimiento.

Este proyecto consta de numerosas fases presentes en cualquier trabajo que realice desarrollo de software. Pero a modo de resumen, podemos dividirlo en tres grandes bloques: análisis y planificación de requisitos, diseño e implementación del software. Durante el primer periodo se especifica el qué va a ser capaz de hacer nuestro software, y tambien se estudia cómo lo va a llevar a cabo, es decir, que herramientas del mercado tenemos disponibles para conseguir el objetivo, y con cual nos quedamos. A continuación, con las conclusiones resultantes de la fase anterior se realizará el diseño del sistema, y finalmente, la implementación y las pruebas del software final del producto.

Este trabajo de fin de grado debe verse como un engranaje dentro de una máquina mayor, ya que dicho proyecto se sitúa en el ámbito de uno mucho más grande, ambicioso y de un largo recorrido. Además, debo destacar que este trabajo se apoya en los resultados alcanzados por diversos alumnos en sus respectivos proyectos, y no solo en el ámbito de la informática, sino en campos como la electrónica o las telecomunicaciones. La complejidad y el ámbito multidisciplinar que supone ser parte de dicho gran poyecto, me ha llevado no solo a cubrir distintas especialidades dentro del Grado de Ingeniería Informática, sino que he adquirido conocimientos y habilidades especificos de otros campos de la ingeniería como son la Electrónica y la Acústica. Incluso debo destacar mi mejoría en mis habilidades intrapersonales e interpersonales, al colaborar y trabajar en equipo con diveros alumnos para llevar a cabo la realización del proyecto.

Por lo tanto, el resultado de todo lo expuesto culmina con la actualización e implementación de mejoras del software de los limitadores de sonido, cumpliendo asi con los requisitos definidos en las etapas inciales del proyecto, y con el cual se cierra mi etapa universitaria en el Grado de Ingeniería Informática.

'If you don't fail sometimes, you are not being ambitious enough'
Sundar Pichai - Google CEO

Acknowledgments:

This work has been possible thanks to a very small number of people, but to whom I would like to show my most sincere and enormous gratitude. Not only for the support and understanding they have given me along this way, but also for being my pillars, my fundamental pieces, which have made it possible for me to be where I am today.

I would therefore like to dedicate this project first and foremost to my relatives, my parents Francisco and María and my sister Gabriela. There are no adjectives capable of expressing the enormous gratitude I feel for them, as they always trusted me, even when I was at my worst and doubted that I would be able to make it. Moreover, despite the obvious distance that separated us, I have always felt supported by the affection they have given me, making me never feel alone throughout this hard but enriching journey.

To my partner, Paola, for always believing in me and making me see that I was capable of doing many more things than I thought I was, showing me that even though life is sometimes cruel, there are always reasons to go on with a smile on your face.

I would also like to thank my tutor Andrés Roldán Aranda and my colleagues in this project, Pedro and Antonio. Without their company, their advice, and the good times spent in the office, I would have embarked on a much more difficult and exhausting path than it has turned out to be.

Last but not least, I would like to thank all the teachers of our beloved Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones, and the colleagues I have met and who have accompanied me during this beautiful stage of my life. All of them have made me even more excited and passionate about this sector of new technologies, making all the laughter, tears, hard times and happy moments that I have spent, have been worth it, showing me that every effort has its reward.

Agradecimientos:

Este trabajo ha sido posible gracias a un número muy reducido de personas, pero a las cuales, quisiera mostrarles mi más sincero y enorme agradecimiento. No solo por el apoyo y la compresión que me han otorgado a lo largo de este camino, sino por ser mis pilares, mis piezas fundamentales, que han hecho que hoy en día me encuentre donde estoy.

Por ello, quisiera dedicar este proyecto en primer lugar a mis familiares, mis padres Francisco y María, y mi hermana Gabriela. No existen adjetivos capaces de expresar el enorme agradecimiento que siento por ellos, ya que siempre confiaron en mí, incluso cuando yo estaba en mis peores momentos y dudaba de que fuera capaz de lograrlo. Además que, a pesar de la evidente distancia que nos separaba, siempre me he sentido arropado por el cariño que me han dado, haciendo que nunca me sintiera solo a lo largo de este duro pero enriquecedor viaje.

A mi pareja, Paola, por siempre creer en mí y hacerme ver que era capaz de hacer muchas más cosas de las que me creía, demostrandome así que aunque la vida sea cruel en ocasiones, siempre hay motivos para seguir adelante con una sonrisa.

Quisiera agradecer también a mi tutor Andrés Roldán Aranda y a mis compañeros de este proyecto, Pedro y Antonio. Sin su compañía, su asesoramiento, y los buenos ratos pasados en el despacho, habría emprendido un camino mucho más difícil y agotador de lo que ha resultado ser.

Y por último, pero no por ello menos importante, quisiera agradecer a todos los profesores de nuestra querida Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones, y a los compañeros que he conocido y me han acompañado durante esta bonita etapa de mi vida. Todos ellos han hecho que me ilusione y apasione aun más por este sector de las nuevas tecnologías, haciendo que todas las risas, los llantos, los momentos duros y los momentos felices que he pasado, hayan valido la pena, demostrandome así que todo esfuerzo tiene su recompensa.

Contents

List of Figures

List of Tables

Glossary

Ansible Ansible is an open source tool that allows you to automate daily tasks such as the management, configuration, application deployment and many other IT processes. [7].

Application programming interfaces An application programming interface is defined as the set of functions or methods used for the transmission of data between two software products [1].

C++ programming language Programming language developed in 1980, with the characteristics of being a compiled language, object-oriented and imperative (they tell the computer how to perform a specific task). [40].

Compute Module is defined as a Raspberry Pi with a more flexible and compact design, ideal for industrial use due to its high performance despite its small size. [12].

Cross-compilation A technique for generating, via a platform other than the original platform, the executable code necessary to carry out an operation..

Digital switch Hardware device whose main purpose is to handling and manage digital signals [37].

Driver set of code that performs the communication between the hardware components and the operating system, indicating how the component should work and how to establish communication with it. [17].

Inventory file where the list of nodes to be managed is stored together with relevant information about them such as their name, group to which they belong or their IP address. [2].

Kernel modules It relies on a code file to extend the functionality of the kernel, usually to add hardware device drivers. It also has the feature that they can be added or removed as needed, and they are added at runtime. [16].

Linux kernel is primarily intended to be the interface for communication and resource management between the computer hardware and its processes, and is therefore the main component of the operating system. [34].

Motherboard It is the main printed circuit board (PCB) where all the components and peripherals of a computer are connected. [39].

Node machine In Ansible, nodes are those devices that will be managed and administered by the host machine. On these nodes you don't need to have downloaded the tool, just activate the SSH communication protocol and have python installed. [3].

Playbook are defined as easy to read and edit files, written in YAML and containing the functionality or tasks that we want to be performed by the managed nodes. [4].

Pulse Width Modulation Modulation technique used for information transmission or power control by generating variable width pulses, thus generating the amplitude of the signal in an analogue circuit. [36].

Raspberry Pi is the name of a series of small single board computers (SBCs) developed in the UK, with the aim of being practical in making computing accessible to all types of users. [33].

Serial Peripheral Interface The SPI is a four-wire bus providing a synchronous serial communication interface for sending data between microcontrollers and small peripherals [41].

Sound limiter A digital device whose main purpose is to monitor that the sound level in a room does not exceed a certain imposed limit. [13].

Telecommunication Technologies All those technologies that are used for long-distance communication by means of electronic transmission of information [42].

Acronyms

API Application programming interfaces.

APK Alpine Linux package keeper.

ARM Advanced RISC Machines.

ASCII American Standard Code for Information Interchange.

ASOC ALSA System on Chip.

CM3 Compute Module 3.

DDR4 Double Data Rate Fourth Generation.

DPKG Debian Package Manager.

eMMC embedded MultiMediaCard.

FPGA Field Programmable Gate Arrays.

GB Gigabyte.

HDMI High Definition Multimedia Interface.

HTTPS Hypertext Transfer Protocol Secure.

I2C Inter-Integrated Circuit.

IDE Integrated Development Environment.

LCD Liquid Cristal Display.

LED Light-Emitting Diode.

MB Megabyte.

OS Operating system.

PC Personal Computer.

PCB Printed Circuit Board.

PGA Pin Grid Array.

PIE Position independent executable.

PWM Pulse Width Modulation.

RAM Random Access Memory.

RGB Red Green Blue.

RTC Real Time Clock.

SBC Single Board Computer.

SD Standard Definition.

SL4 Sound Limiter 4.

SO-DIMM Small Outline Dual In-line Memory Module.

SPI Serial Peripheral Interface.

SSH Secure SHell.

UGR University of Granada.

USB Universal Serial Bus.

VGA Video Graphics Array.

VPN Virtual Private Network.

Chapter 1

Introduction

The Bachelor Thesis presented here, shows the Final Degree Project of the University Degree in Computer Engineering, with specialising in Software Engineering, taken by the student Raúl Rodríguez Pérez at the Engineering School of Technology and Telecommunications in Granada. The main objective of this work is to demonstrate the knowledge and skills acquired by the student during the degree. To this end, the present project has been carried out, which proposes the updating and management of the lowest level software in the electronic equipment that allows the acoustic levels in the premises to be controlled.

This Final Degree Project is carried out in collaboration with the GranaSAT academic project. This is an aerospace development group of the University of Granada (UGR), formed entirely by students and under the supervision of Professor Dr. Andrés María Roldán Aranda. It brings together people from different fields who want to acquire knowledge related to Aerospace Engineering and Electronic Engineering.

Figure 1.1 – *GranaSAT logo*

The GranaSAT laboratory, as well as the equipment and materials necessary to carry out this project, are located in the I+D Josefa Castro Visozo building, which is next to the Aulario of the International Postgraduate School of Granada and the old Clinical Hospital of Granada (Spain). It should also be noted that this project has been carried out both on-site in the laboratory and remotely.

1.1 Motivation

My Bachelor thesis is **part of a much larger project** that began years ago. Its main objective is to design, implement and produce a Sound limiter which controls noise pollution in commercial establishments such as bars, discotheques, etc. This need arises from the importance of controlling noise emissions from establishments, not only because it affects the rest of neighbours but also because of concerns about the hearing health of customers. This project has given rise to several students from different branches of engineering to develop both their final degree and master's degree projects. So my task is to continue with the development of it and to contribute with my knowledge, my enthusiasm and my desire to learn.

1

Regarding my job, initially I was going to be **in charge of** a part much more related to my field, software development. But once Professor Andrés saw how well I performed the first tasks, he proposed me to go into a slightly more ambitious project, where I would have to learn more about computer architecture and software at a lower level. At first I was a bit scared and doubted whether I would be able to carry out the project. But anyway, I saw it as a very good opportunity to learn and expand my computer skills, so I accepted my tutor's challenge and got down to work.

1.2 Context

As I said before, my labor is part of a much larger project in which many people have participated. In my case, I have continued the work of Alejandro Ruiz Becerra, a previous student of the degree in computer engineering who carried out his final degree project in the 2020-2021 academic year. In his Bachelor's Thesis [9], he was able to reverse engineer the Sound limiters already on the market and then design, develop and test the system based on the extracted requirements.

In addition, Alejandro had to work with two colleagues to perform other tasks. On the one hand, he had to establish the communication methods between the limiter and a desktop software that could configure the Sound limiters via the local network. This software was created by Daniel Ruiz Medina, another computer engineering undergraduate student who was also working on his Bachelor's Thesis [28]. And on the other hand, the software he created was mainly aimed at the technical users who install the Sound limiters, being necessary to implement the required access measures in an efficient and secure way in front of a production server. This work was carried out together with Cristobal Rodríguez Reina, a student of the same degree who was working on his master's thesis.

So, as a conclusion of his work, I found that in GranaSAT laboratory there were **4 prototypes of Sound limiters** (which only differed in the hardware components since the software they used was the one made by Alejandro). Besides knowing that this software was capable not only of performing the basic functions of monitoring and noise control, but also of communicating with a desktop software through the local network, and with a server through the internet.

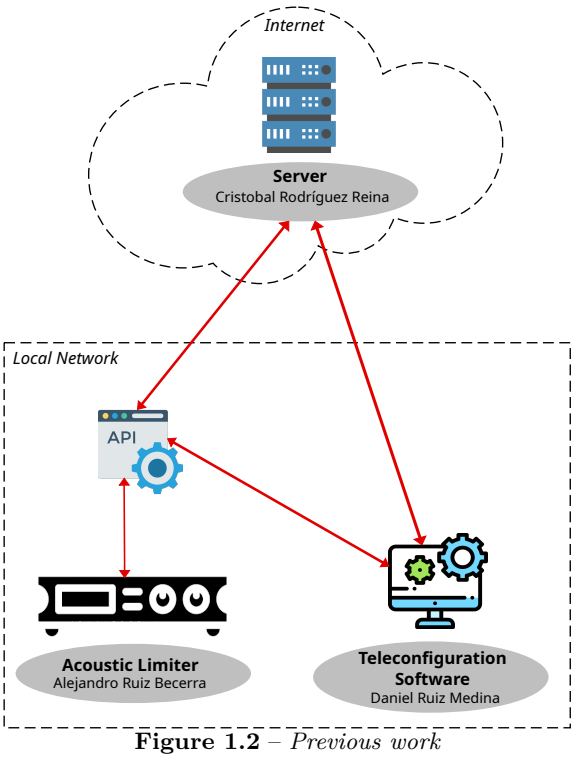


Figure 1.2 – *Previous work*

1.2.1 People involved

What I have explained above is the work done by the students of last year's Bachelor's and Master's degree in computer science. But it should be noted that, during the course of my project, I have collaborated to a greater or lesser extent with other colleagues to complete it successfully.

- *Pedro Javier Belmonte Miñano*, student of the Bachelor's Degree in Telecommunication Technologies and the person responsible for the bachelor thesis on the documentation of the hardware section of Sound limiters.
- *Antonio Manuel Cantudo Gómez*, student of the Bachelor's Degree in Telecommunication Technologies and the person responsible for monitoring the communications of the Sound limiters with the servers and the configuration software.
- *Raúl Rodríguez Pérez*, student of the Bachelor's Degree in Computer Engineering, author of this document and the person responsible for the software update and upgrade work.
- *Andrés María Roldán Aranda*, tutor of my and Pedro's bachelor's thesis, and in charge of supervising and directing them.

Figure 1.3 – *People involved*

1.2.2 Sound limiters used

As I mentioned earlier in section 1.2, I had four prototype Sound limiters for this project. For each one, I had to communicate with my colleague P. Belmonte, as he was the one who knew the hardware differences between them. Thanks to his explanation, I was able to learn a little more about the process that had been followed when it came to implementing improvements to the hardware aspect of the Sound limiters, giving me an idea of what we were dealing with.

Of the **four prototypes**, I have done work with all of them, but I have done most of the work with one in particular. This limiter is the one with the latest version in terms of hardware design. From now on we will refer to this limiter **by the name Sound Limiter 4 (SL4)**. In terms of hardware specifications, the SL4 has the following features:

1

- In essence, the limiter is based on a **custom-made PCB or Motherboard** on which the various components are connected. In addition, it has a Raspberry Pi Compute Module 3 (ultimately a Raspberry Pi with a more flexible design, intended for industrial use), which runs a DietPi operating system (we will explain the reason for their choice in future chapters.), based on a Linux distribution.
- Regarding the input and output of the audio channels, the limiter has **six balanced inputs**. Furthermore, as we can see in the back view section of figure 1.4, the prototype has only one microphone, although, next to it, we can see a hole which is intended to be the space to incorporate a second microphone in future versions.
- The software analyses the audio and decides whether to act on the noise levels. Using **the PGA** (component connected to the Raspberry via SPI), the audio of the input music is received, and the signal is acted upon by applying a gain that can be positive or negative. Finally the audio comes out of the limiter to a **Digital switch**, which will redirect the audio to the amplifiers and/or speakers.
- To conclude, the limiter has a **couple of add-ons**, e.g. a small LCD display of 4 rows by 20 columns, a buzzer, and a series of RGB LEDs. Each of these will have a specific purpose, which we will look at in future chapters.
- The differences between SL4 with the other three prototypes are based on, different Motherboard design and none of them have added LCD display add-on.

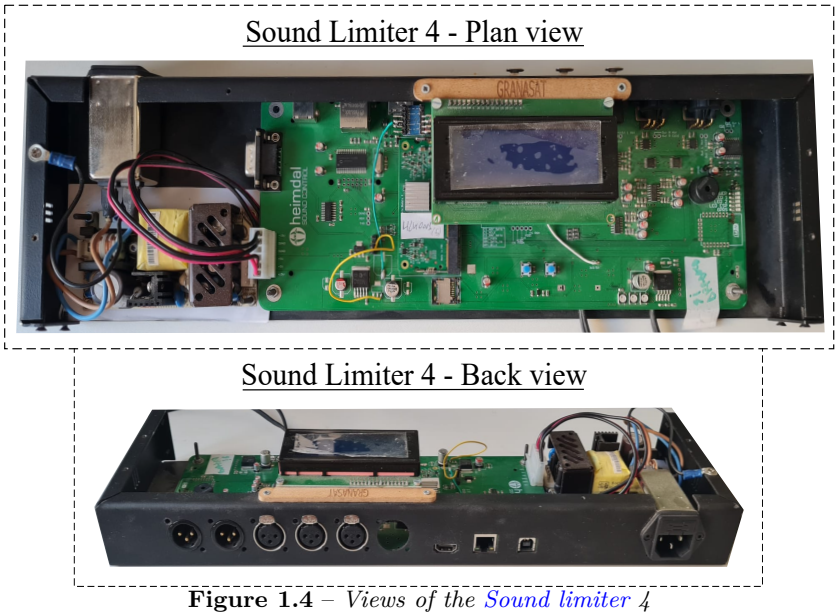


Figure 1.4 – *Views of the Sound limiter 4*

1.3 Project objectives

Before moving on to the development and documentation of the project, I would like to outline the objectives I intend to achieve with the completion of this bachelor's thesis.

1. To demonstrate the knowledge and skills acquired in the Bachelor's Degree in Computer Engineering.
2. To accustom the student to an environment more closely linked to engineering, participating in a real large-scale project.
3. Improve and broaden my teamwork skills, collaborating and learning from students not only from my own Bachelor's degree.

4. To get out of my comfort zone, and learn new areas of computer engineering such as lower level software.
5. Understand the operation of Sound limiters. Learn the hardware behind them, and how the different devices communicate.
6. To pass the Bachelor's thesis, and therefore, the Bachelor's Degree in Computer Engineering.

1.4 Project structure

The following project is divided into seven chapters, which include annexes describing each part of the development process of the proposed tasks. These chapters that make up this document are as follows:

- **Chapter 1: Introduction**, is the present chapter and it is aims to provide not only an introduction to the project, but also the context and the objectives to be achieved by its realisation.
- **Chapter 2: Requirement list**, section presenting the different requirements necessary to achieve the project's purpose.
- **Chapter 3: Analysis**, stage where the aim is to find out what is really needed and where a proper understanding of the system requirements is reached. Moreover, the project schedule and the project budget is presented too.
- **Chapter 4: Design**, section detailing the models of possible solutions to the proposed requirements.
- **Chapter 5: Implementation**, part in which models of possible solutions are developed.
- **Chapter 6: Verification and testing**, section to carry out the necessary validation tests to ensure the correct functioning of the solutions provided.
- **Chapter 7: Conclusions and future lines**, part where the opinion on the achievements of the project is expressed and possible future improvements are discussed.
- **Annex**, section presenting various user and operating manuals.

Chapter 2

Requirement list

The aim of this section is to present in its entirety the list of requirements that has been elaborated during the numerous interviews with the client. The main purpose of this list is to define what our system will be able to do and the steps to be taken to achieve it. First of all, however, I will show the main objectives of the project:

Ref.	Main Objectives
MObj.1	Product operating system analysis and selection.
MObj.2	Perform system Linux kernel update.
MObj.3	Analysis and choice of security measures for the product.
MObj.4	Software system monitoring and deployment with Ansible.
MObj.5	Update the product's automated hardware test programs.
MObj.6	Update sending data from Sound limiter to installer control software and production server.

Table 2.1 – *Main Objectives*

Once we know what the main objectives of our project are, we are going to analyse each of them separately, thus showing the tasks that we will have to carry out in order to achieve the final objective:

Task	Description
Task 1.1.	Study on Raspberry Pi single board computers (in particular the Raspberry Pi 3 Model B, and the Compute Module 3), what they are, what components they have and what they are used for.
Task 1.2.	Research how to add new distributions to our Raspberry Pi and to the Compute Module.
Task 1.3.	To Conduct research on lightweight operating systems, comparing their main features and the disadvantages of using them
Task 1.4.	On the basis of the research carried out, choose the operating system to be used in the final product.

Table 2.2 – *List of tasks to fulfil Mobj.1*

Task	Description
Task 2.1.	Study about the Linux Linux kernel, what it is, how to configure it, and how to add modules to it.
Task 2.2.	Study and understand the list of existing Drivers on the system and how to add them to the new distribution.
Task 2.2.1.	Pay special attention to the Linux kernel's own modules, their dependencies and how to add them to the new distribution.
Task 2.3.	Add all Kernel modules from the old version to the chosen distribution for the Sound limiter.
Task 2.4.	Execute the Sound limiter code when all Drivers are included in the new distribution, transmitting the data to the production server and making the data available on the HeimdalSoundControl website.

Table 2.3 – *List of tasks to fulfil Mobj.2*

FR Ref.	Description
Task 3.1	Analysing system vulnerabilities from an attacker's point of view.
Task 3.2	Study the possible attacks on our vulnerabilities, which would be most damaging to us.
Task 3.3	Make a proposal for security measures to minimise or completely eliminate damage.
Task 3.4	Carry out the implementation of the proposed security measures.

Table 2.4 – *List of tasks to fulfil Mobj.3*

Task	Description
Task 4.1.	Conduct a study of the Ansible tool, what it is and what it is used for.
Task 4.2.	Design the appropriate system for testing with the Ansible tool.
Task 4.3.	To know the prerequisites for using Ansible, and install it on the host and Node machines.
Task 4.4.	Configure Playbooks and Inventory to be used for monitoring Sound limiters.
Task 4.5.	Carry out the relevant tests to check the correct use of the tool.

Table 2.5 – *List of tasks to fulfil Mobj.4*

Task	Description
Task 5.1.	Documentation of Sound limiter LED testing
Task 5.2.	Study and document how hardware devices are tested
Task 5.3.	Pass the LED tests to C++ programming language, and add them to the code along with the rest of the tests for the hardware devices.
Task 5.4.	Study and update of the script used by the buzzer at system start-up.
Task 5.5.	Perform Pulse Width Modulation (PWM) analysis with the pulseview tool.

Table 2.6 – *List of tasks to fulfil Mobj.5*

Task	Description
Task 6.1.	Study how communications are made between the API and the production server, and also with the control installer.
Task 6.2.	Update the Sound limiter API
Task 6.3.	API support for new features to be created in the Sound limiter configuration application

Table 2.7 – *List of tasks to fulfil Mobj.6*

Once we are clear about both the main objectives of the project and the tasks we will have to perform to achieve them, we will proceed to list the functional and non-functional requirements that cover the whole system. It is true that, as such, our main mission in this project is not the creation of the software, but it is still crucial to make these lists as they will allow us to know what our system does and how it does it:

Ref.	Description
FR 1.	The system shall be able to allow only users who are correctly identified and verified to log in to the system.
FR 2.	Once the system has started, a high-pitched sound will be emitted through the buzzer if the communication with the FPGA has been successful.
FR 3.	Once the system has started, a low sound will be emitted through the buzzer if the communication with the FPGA has failed.
FR 4.	The system must include a update mechanism to allow detection of the system by other equipment on the same network.
FR 5.	The system must provide a update mechanism for sending data to the production server and to desktop software.
FR 6.	The system shall be capable of communicating with and testing the LEDs incorporated in the board.
FR 7.	An automation process for Linux kernel Cross-compilation shall be developed.
FR 8.	The Kernel modules should continue to function in the same way after the Linux kernel update.

Table 2.8 – *List of functional requirements*

Ref.	Description
NFR 1.	The system shall control user login through the SSH communication protocol by means of private public keys with ed25519 digital signature algorithm
NFR 2.	The system shall make use of the HTTPS protocol to protect the integrity of the data being sent.
NFR 3.	The system will verify the identity of users sending packets with the use of SSH keys.
NFR 4.	To avoid reverse engineering, Kernel modules should be integrated into the Linux kernel using the make nconfig tool.
NFR 5.	The system shall be able to run on the hardware of the latest available version of the Sound limiters.
NFR 6.	The system shall contain a customised LXQT desktop on the DietPi operating system.
NFR 7.	The system should have well-structured documentation.
NFR 8.	The system shall have user manuals for guidance to perform all tasks carried out.
NFR 9.	After the system upgrade, the total storage of the system software shall not occupy more than 4 GB of disk space.

Table 2.9 – *List of non-functional requirements*

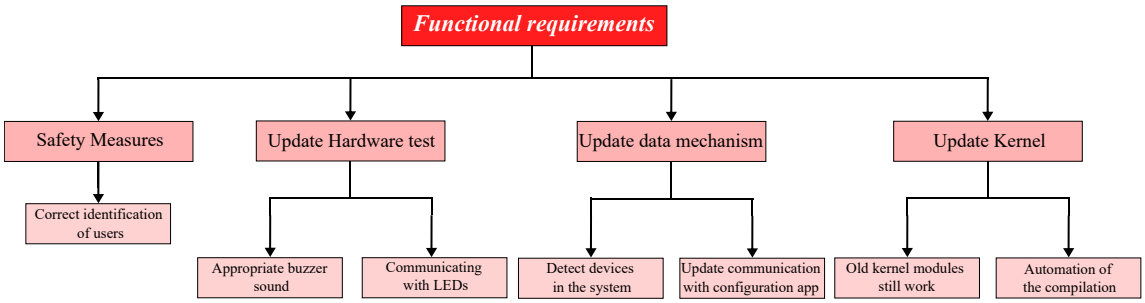


Figure 2.1 – *Overview of Functional requirements*

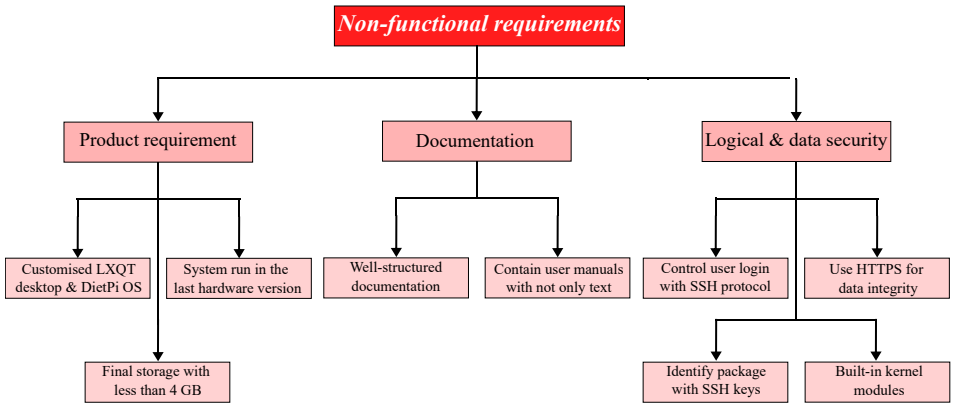


Figure 2.2 – *Overview of Non-Functional requirements*

Chapter 3

Analysis

In this section will show the analyses, comparisons and decisions taken to complete each of the main objectives of the project. In addition it will also contain the planning I have carried out to complete each of the tasks, and the budget of how much it will cost to complete them.

3.1 Product operating system analysis and selection

When the software was designed and developed in Alejandro's bachelor thesis [9], an operating system was chosen without any criteria, as there was no prior analysis of which one is the most suitable for us and therefore which one we should use. Therefore, one of my first objectives will be to carry out a **study of** the **different operating systems** that we can use, analysing them carefully and observing their details, in order to be able to argue and decide which one is more convenient for us.

3.1.1 Research about single-board computers

To give context to this research, I must explain in which state the limiters were initially in. As I mentioned in section 1.2.2, each of the Sound limiters had a **Raspberry Pi Compute Module 3** [31].

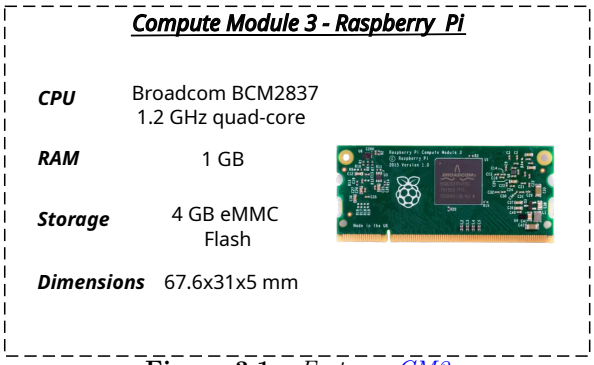


Figure 3.1 – *Features CM3*

These modules are nothing more than a Raspberry Pi designed in a more **compact and flexible** way, intended for industrial application and embedded system use. So, when I went to conduct the study of what was contained in each Sound limiter, I realised that after having carried out numerous software tests on the limiters, having downloaded various packages with the aim of fulfilling the tasks, and not keeping track of

what is strictly necessary, i.e. only occupying the CM3 space with the files that have an essential use for it, I could see that it was very complicated to manipulate the Sound limiters because they were at the **edge of their maximum capacity** and with a lot of files that at first I did not know their use. It is true that, as we can see in figure 3.1, the compute module 3 does not have a large storage capacity, **being only 4 GB**, so it is understandable that the Sound limiters are on the edge of their storage.

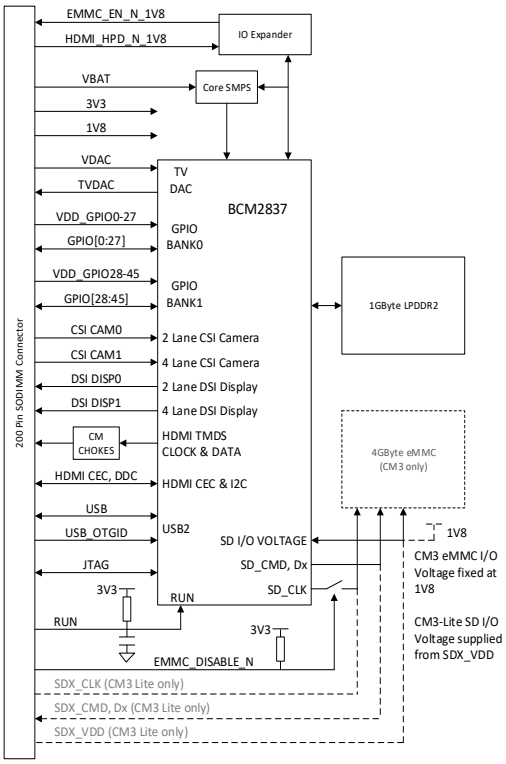


Figure 3.2 – *CM3 diagram block*

Furthermore, if we look at figure 3.2, which shows the block diagram of the CM3, we will get more information about our compute module. For example, we can see that it is on a SO-DIMM module with 200 pins available, capable of 64 bits data transfers with devices such as USB, HDMI, etc.

So, knowing a little bit about the main features of CM3, and once we know the current problem with storage space, our next step is looking for **alternative devices** on which we could carry out the appropriate tests for the choice of the operating system. It was at this point that my teacher Andres asked me to research **single-board computers** [6], as these could be the perfect devices for testing. These devices are typically used in demonstrations for educational systems, as they contain all the features of a computer (microprocessor, memory, I/O, etc.) but on a single printed circuit board. As a result, the equipment is **portable and lightweight**, and the price is lower than that of an average computer. In our case it will be useful to make use of them because, being portable, I will be able to work from home or in the office, as well as not being dependent on hard-to-get or expensive accessories.

So finally, we chose three possible single-board computer options to make comparisons, and then choose the best one for us. In the GranaSAT laboratory we already had a specific model of Raspberry Pi 3 Model B [30], so we have decided to **compare this model** with others on the market that have more or less the same or better features for a similar or slightly higher price.

Features	Raspberry Pi 3 model B	BeagleBone AI-64	Marsboard A20
USB	Broadcom BCM2837 2 GHz	Texas-Instr-Jacinto TDA4VM 2 GHz	Allwinner A20 1 GHz
Architecture	ARMv8 Cortex-A53 64 bits	ARM Cortex-A72 64 bits	ARM Cortex-A7 32 bits
RAM	1 GB DDR2	4 GB DDR4	1 GB DRAM
GPU	Dual-Core Video Core IV	PowerVR Rogue 8XE-GE8430	ARM Mali400MP2 Complies
Storage	SD card slot	16 GB eMMC Flash microSD card slot	8 GB eMMC Flash SD/MMC card slot
Ethernet	10/100 M	1 Gigabit Ethernet	10/100 M and USB WIFI
USB	4xUSB 2.0	1xUSB 3.0 type-C 2xUSB type-A	4xUSB 2.0-host 1xUSB 2.0-OTG
Video	HDMI, Composite	miniDP, MIPI DSI (w/I2C)	HDMI,CVBS,VGA
Audio	HDMI, 3.5 mm audio jack	miniDP cape/USB add-ons	3.5 mm Jack and HDMI
Power	Micro USB socket 5V-2.5A	Micro USB socket 5V-3A	Micro USB socket 5V-2A
OS support	Linux, Windows 10-IoT	Debian GNU/Linux	Linux and Android
Dimensions	85x56x17 mm	137x100x33 mm	80x55x20 mm
Cost	36-45 euros	80-110 euros	60-85 euros
Preference	 (1º)	 (2º)	 (3º)

Table 3.1 – *SBC features*

Once we have compiled and captured all the characteristics of each of the SBCs in table 3.1, it is now time to analyse it in order to choose one of them. In order to make my decision, I **not only compared** with the features in the table, but I also researched each of the options, looking for example at whether their community is large and active, whether they have documentation for beginners, how difficult it is to find stock, and so on.

So at first glance, if we look at who has a **better community**, the clear winner would be the Raspberry model. This is a much better known brand worldwide, and has an active community and a large amount of documentation not only of an official nature, but also of projects made by other users. However, because of the current situation where we are facing the aftermath of the Covid-19 pandemic, the **technology market** has suffered major consequences, making it very difficult to find components that were previously no problem to purchase. Because of this, models such as the raspberry pi have become difficult to obtain, and you have to pay **exorbitant prices** if you want to buy one nowadays. On the other hand, the BeagleBone [8] and the Marsboard [27] model are also in low stock, but you can find them in more places anyway.

In conclusion, once the research work has been carried out, we must **choose a winner**. In our case, having explained the apparent differences between the three candidates to the teacher, we came to the

consensus that we will use the **Raspberry pi 3 Model b**. This decision was made on the basis that we already had this model in the laboratory, and that we use components of the same brand in the limiters, such as the Compute Module 3.

3.1.2 Research on lightweight operating systems

Once we chose our single board computer, the next step would be to **choose which operating system** we are going to put on our final product. It should be noted that from now on, all the tests that will be carried out will be done using a Raspberry Pi 3 Model B.

One of the key things to bear in mind when looking for our operating system is that we need a **lightweight system** [26]. As mentioned in the previous section, we only have 4 GB of storage, where we must include the software we already have and the operating system. Therefore, we must find a balance between an operating system that is light, but also has numerous options to complete our goals.

As a consequence of this requirement, one of the first decisions we made was to look at operating systems that did not have a desktop, as these would take up very little space. Even so, my teacher wanted the final product to have a desktop, so we would simply then have to look for **lightweight desktops** that could be manually added to the operating system of our choice. So, with all this in mind, I started a search for lightweight operating systems, resulting in the following list:

- **Raspberry PI OS Lite** [32]: Raspberry Pi OS, previously known as Raspbian, is the official supported operating system from raspberry company. In our case, I choosed the most lightweight version of this OS. This option don't have desktop, but it have bullseye version of Debian and it is a 64-bit system.
- **Dietpi** [11]: DietPi is an lightweight Debian based OS. It is strongly optimised for low RAM and USB usage. It also includes a small menu to customise your device's hardware/software settings.
- **Tiny Core Linux** [22]: Tiny Core Linux is a extremely lightweight Linux kernel based operating system. It is designed to run from a copy of RAM that is created at boot time. In addition to being fast, this protects system files from changes and ensures the integrity of the system on each reboot
- **Arch Linux ARM** [5]: It is a Linux distribution for ARM computers. It focuses on providing kernel and software support for more experienced Linux users, giving them full control and responsibility over the system.
- **Alpine Linux** [21]: Alpine Linux is a lightweight, simple and secure Linux based operating system. It uses its own package manager instead of the usually for Linux distributions. It also provides security in all our binaries files, because this are compiled as PIE with stack smashing protection.
- **Debian** [10]: Debian is a free and open source Linux distribution. It is one of the most popular editions for personal computers and servers, as it has a large and active community that provides constant updates and upgrades to the distribution.

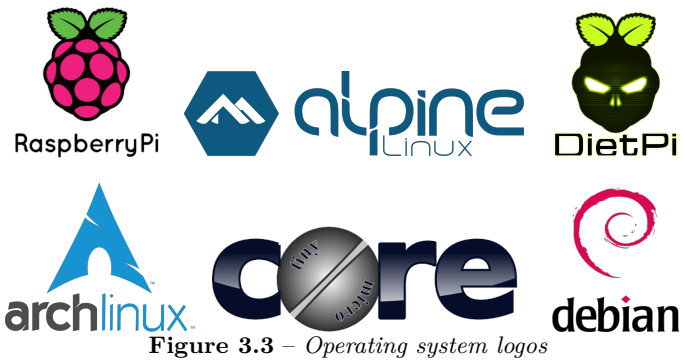


Figure 3.3 – *Operating system logos*

Once we have introduced the operating systems we have chosen to analyse, we will proceed to show several graphs comparing aspects such as storage space and memory usage.

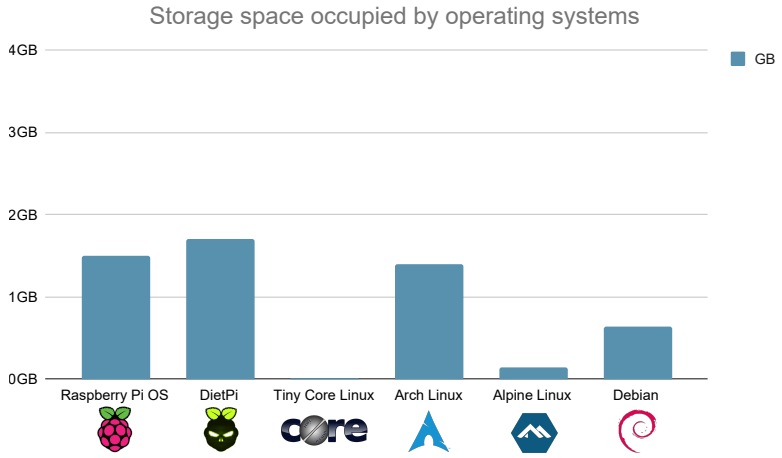


Figure 3.4 – *Storage space occupied*

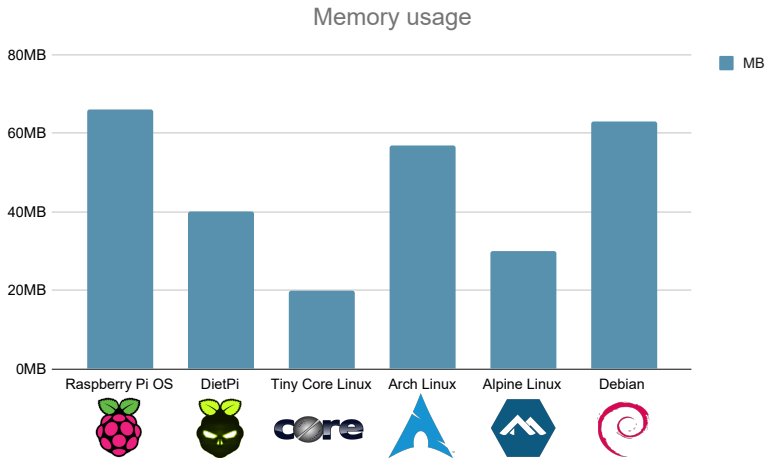


Figure 3.5 – *Memory usage*

On the one hand, as we can see in Figure 3.4, each of these operating systems **weighs less** than 2 GB. Also something that catches our attention is the absurd weight of Tiny Core Linux, as it almost does not appear in the graph due to its low weight. If we look at the official website, it tells us that, depending on the image, this operating system can weigh between 16 and 163 MB. On the other hand, in Figure 3.5, we can see that almost all of these operating systems have very **low memory usage**, with Raspberry Pi OS and Debian being the only ones that exceeds 60 MB.

Once I presented the main features of each operating system and shared the results of the research with my professor, we were **left with a group** of the operating systems that convinced us the most. The **chosen one** were: Dietpi, ArchLinux ARM, Debian and Alpine. This decision was based on our main objective, which is create a lightweight system that provides only the features we really needs and nothing more than that.

We could see that in Raspberry PI OS there were a lot of **unnecessary options** that we were never going to use. And we came to the conclusion that we would rather choose only the functions that we were going to use, than to be eliminating all the ones that we were not going to use. Apart from that, we also decided to remove Tiny Core Linux from the list of options. I was very curious to learn more about this little-known OS, but its **particular way of** doing things did not fit with our final vision of the product.

3.1.3 Research on a desktop for our operating system

In the previous section, we chose the group of operating systems that were most suitable for our task. But in order to know which one to choose, we will have to analyse them further. This time we want to look at **lightweight desktops** [23] environment that we can add to these OS's.

When looking for desks to use, we took into consideration a number of things. Firstly, we looked for desktops that were **as light as** possible, secondly, we searched for those that had the option to **customise** them to our liking, and lastly, that were capable of **being used on** all the OS on our list. So finally, the desktops chosen were:

- **LXDE** [24]: is a lightweight, free desktop environment written in C language. It is known for its low resource requirements, which makes it ideal for use on older computers.

- **LXQT** [25]: is an open source lightweight desktop environment, resulting from the merger of the LXDE and Razor-qt projects. It is also known for its large catalogue of customisation's, being able to modify almost any aspect of the desktop.

- **XFCE** [43]: is an open source desktop environment that aims to be efficient and lightweight while remaining visually appealing to any user.

Figure 3.6 – *Desktop environment logos*

By researching each of these desktop environments, we can find out that even they are lightweight, they can still be functional, and we can modify them to our preference to make them **visually appealing**. Now the next step is to test all the candidates on each of the finalist operating systems. For this occasion, we are going to do **several tests** in which we will need a specific setting to carry them out.

Before testing, we will need to check that we have some method of monitoring memory and USB usage, and that we have a lightweight browser installed. With all these requirements in place, we will proceed to investigate **how much space** each operating system takes up with the desktop environment we have installed. Bearing in mind that the resulting value also reflects some functions that we have added by ourselves, such as the browser.

Then we will start testing **memory usage** in different scenarios, first with no windows open, then with many windows open, and finally with the browser open and performing a search. And the last test we will do, is to observe the **CPU usage** when we have the browser open or when we open many tabs at the same time. The graphs showing the above-mentioned results are as follows:

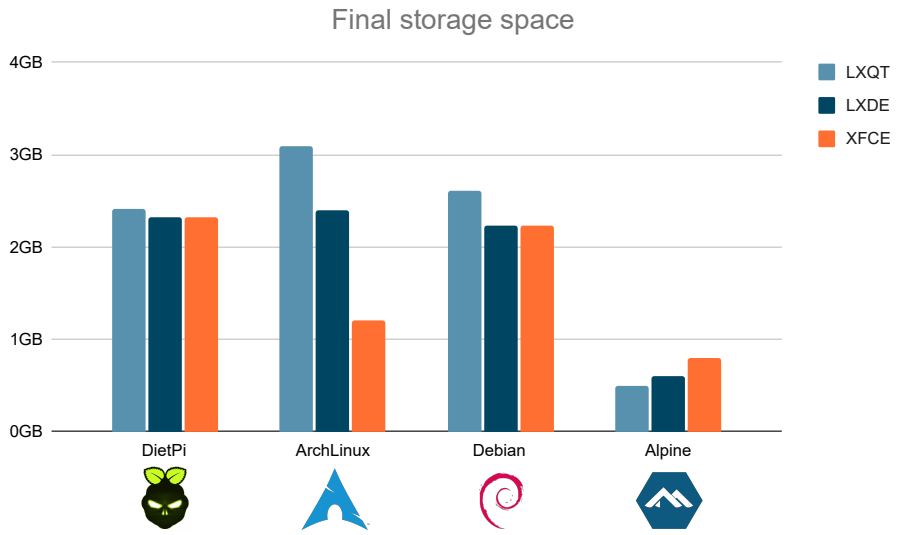
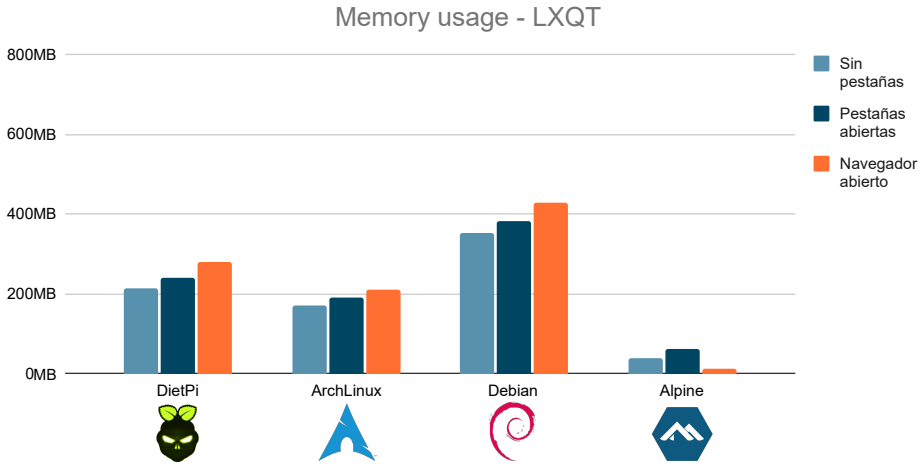
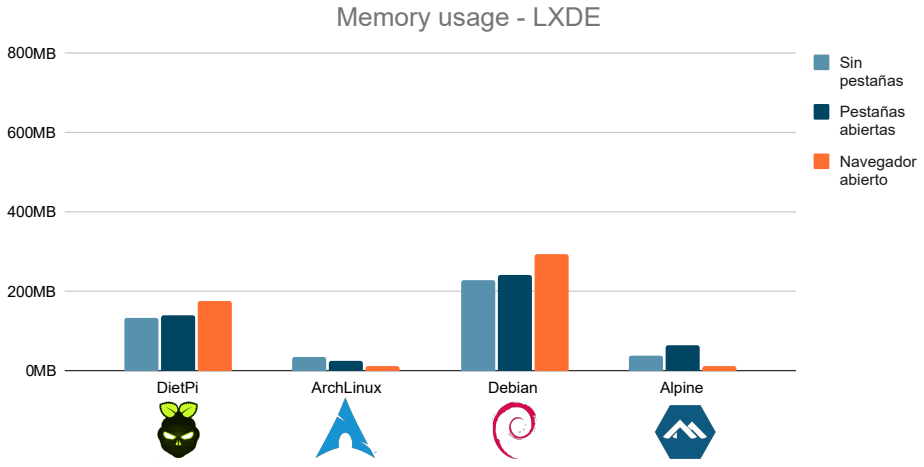


Figure 3.7 – *Final storage space*



(a) *Memory usage - LXQT*



(b) *Memory usage - LXDE*

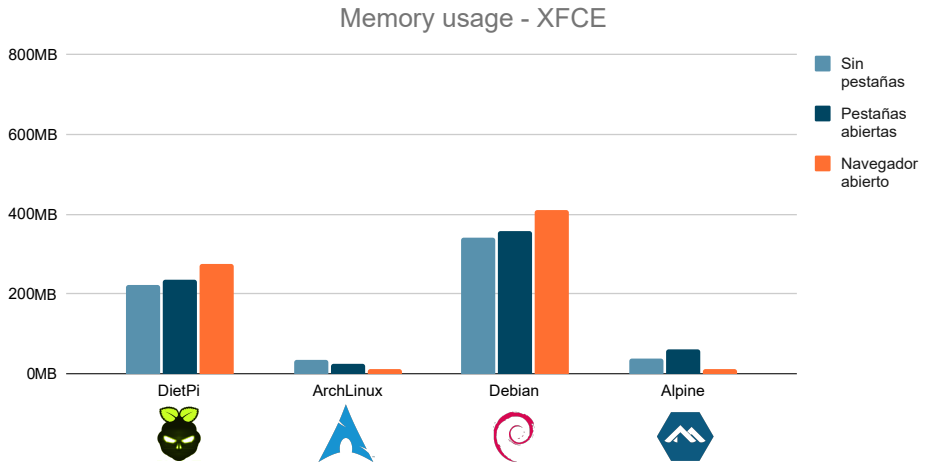
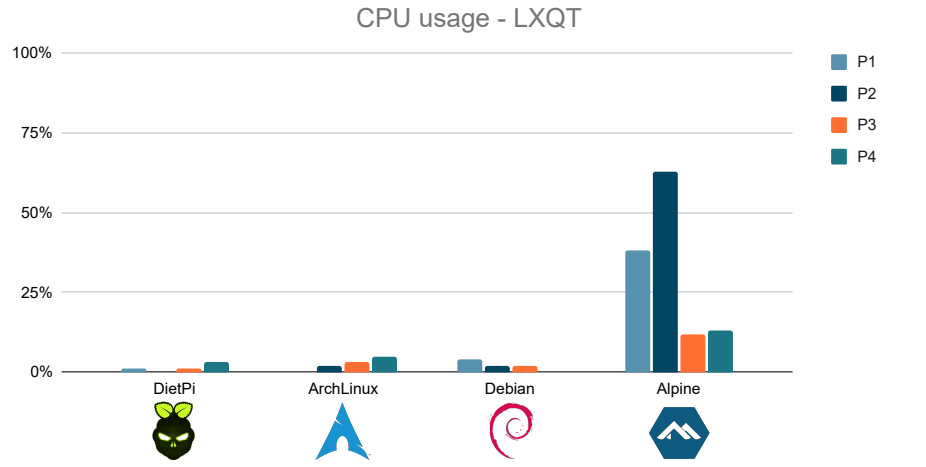
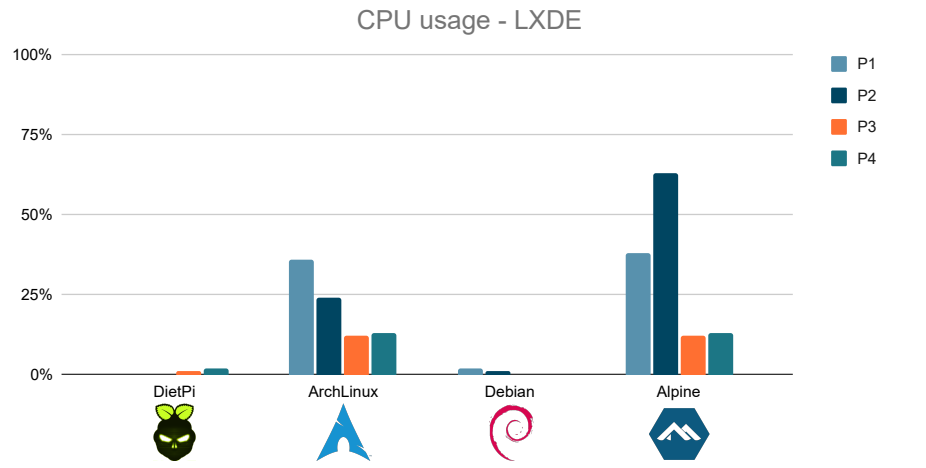


Figure 3.8 – *Memory usage - XFCE*



(a) *USB usage - LXQT*



(b) *USB usage - LXDE*

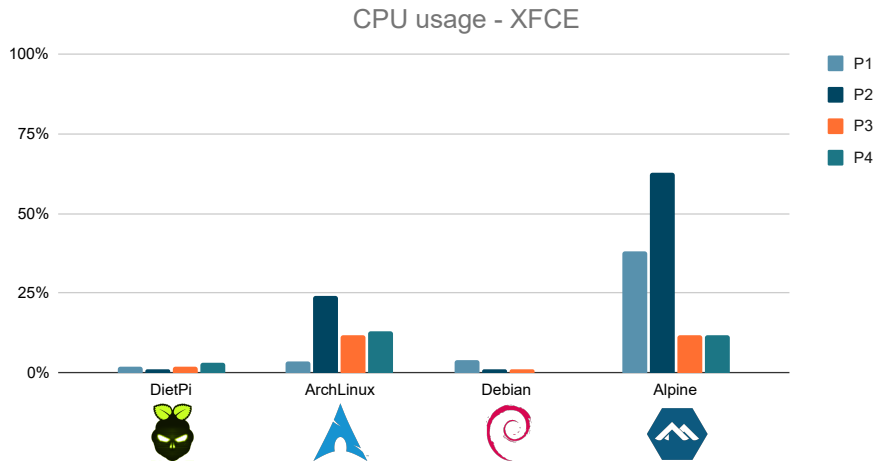


Figure 3.9 – *USB usage - XFCE*

3

To clarify and show the conclusions we draw from the graphs, I have prepared the following table:

Graphs	DietPi	ArchLinux	Debian	Alpine
Final storage space	✗ (3º)	✗ (2º)	✗ (4º)	✓ (1º)
Memory usage LXQT	✗ (3º)	✗ (2º)	✗ (4º)	✓ (1º)
Memory usage LXDE	✗ (3º)	✓ (1º)	✗ (4º)	✗ (2º)
Memory usage XFCE	✗ (3º)	✓ (1º)	✗ (4º)	✗ (2º)
USB usage LXQT	✓ (1º)	✗ (3º)	✗ (2º)	✗ (4º)
USB usage LXDE	✓ (1º)	✗ (3º)	✗ (2º)	✗ (4º)
USB usage XFCE	✗ (2º)	✗ (3º)	✓ (1º)	✗ (4º)
Preference	✓ (1º)	✗ (3º)	✗ (2º)	✗ (4º)

Table 3.2 – *Conclusions from the graphs*

3.1.4 Operating system of the final product

After all the research we have done, carefully analysing all the details of each section, we can finally make a solid decision based on all the evidence we have conducted. Therefore, the operating system we have chosen for our product is: **DietPi with LXQT** for the environment desktop.

By conducting a research outside of the tests, we found that Arch Linux ARM and Alpine had **problems** compared to the other two competitors. These two use their own package managers (Pacman for Arch, APK for Alpine). This, in principle, should not be a big problem, as they only use **other commands** as opposed to the Debian GNU/Linux-based distributions which use the DPKG package manager. But for me, who has always worked with a Debian-based Linux distribution, it is normal that **my performance** with these other operating systems is lower.

Apart from that, we also took into account the community of the operating system, how big it was, whether it was active or not, as well as the quality of the documentation. So, after doing some research on this, we **ruled out Alpine** as a candidate. The remaining three candidates did present a large active community, and quality documentation. However, Debian was the worst performer in our tests, so we discarded it as well.

Figure 3.10 – *Final decision*

Finally, we had to decide between Arch Linux ARM or Dietpi. The reasons why we opted for DietPi were that it was a **debian-based distribution** and therefore my performance and possibly that of my colleagues was going to be better. Also the installation and OS **configuration process** was much easier and more efficient compared to Arch Linux ARM. And finally, this system has software that **greatly facilitates** the interaction with the hardware of our product.

On the other hand, we decided to use the LXQT desktop environment because, although it takes up the most space of the three, it is the one that performs best in the other tests because it is the **best optimised**. In addition, it offers a large number of **options to customise** the desktop to your liking (hence the difference in size compared to its rivals) and make the final product look elegant and professional.

3.2 System kernel update

Our product, the Sound limiter, contains several hardware devices (which are shown in section 1.2.2) whose individual objectives come together to support the proper functioning of it. Without the **proper performance of these devices**, we would have nothing more than a printed circuit board taking up space on our desk. It is therefore crucial for us as product engineers to know the status of our components at all times. To do this, we must configure the kernel of our product so that after booting the system we can communicate with them and thus know their status from a higher programming layer.

This work, the **configuration of a Kernel** suited to our needs, was done by a student long before Alejandro started his project. As a result, Alejandro could focus exclusively on the creation of the software, as the communication with the hardware devices was already established and he could communicate with them to do the tests.

But in my case it's different, one of my most challenging goals in this project is to **update the kernel**,

making sure that everything works as before. To do this, I need to have a deep understanding of the kernel we have before performing the upgrade.

3.2.1 Overview of Linux kernel architecture

Prior to this project, I had never worked on or modified anything related to the system kernel, in fact, when the professor gave me the assignment, I only knew a very superficial definition of what it was used for. So, before entering the lion's den, I needed to **learn about the kernel**, what it is used for, how it is configured, problems that could arise when updating it, etc. So that, at the end of the study, I would know what information I needed to carry out the update, and where to look for it.

The architecture of the Linux operating system is **layered**, and following Garlan and Shaw's style of decomposition [14], each of the layers that make up the system is only able to communicate with the layers that are **adjacent to it**. In addition, the layers that appear near the top of the system **depend on** the layers below, but those closer to the bottom do not depend on the layers above.

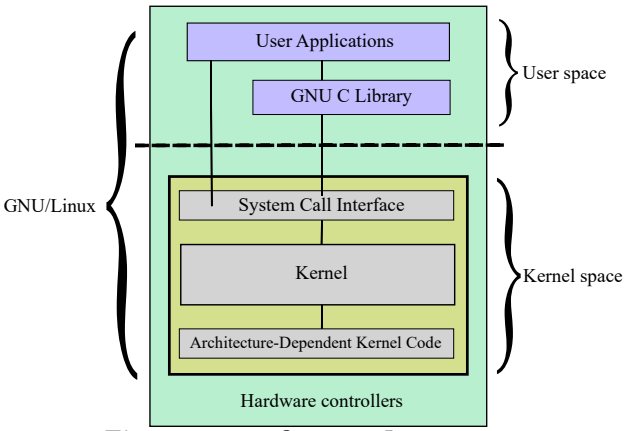


Figure 3.11 – *Overview Linux system*

As shown in Figure 3.11, we can divide the linux operating system into three major subsystems:

- **User space**: system memory allocated to running all kinds of applications such as a word-processor or a web-browser. It is the space in which we, as users, move around and interact with the aforementioned applications, databases, files, etc.
- **Kernel space**: system memory allocated to the core component of the operating system. It manages the system's resources, communicates with the hardware, and is responsible for memory, processes, and file management.
- **Hardware controllers**: consists of all the physical devices connected to the system; for example, the USB, memory hardware, hard disks, network hardware, etc.

To give more insight into the system architecture, the kernel can be thought of as a busy assistant and hardware security gate keeper. Basically, the kernel **receives information** from the user space and then **interacts directly** with the hardware to perform some function. In addition, the kernel has to manage user initiated processes and programs, control where things are stored, and determine which processes have access to which parts of the hardware at any given time. Actually, as we can see in the figure 3.12, the Linux kernel is composed of five main subsystems [19]:

- **Process Scheduler**: is the subsystem in charge of controlling process access to the USB. One of its main tasks is to guarantee fair access to the USB for each process, while also ensuring that hardware actions are performed on time.

- **Memory Manager**: is the subsystem that allows multiple processes to safely share the machine's main memory system. In addition, the memory manager provides Linux with the ability to support processes that use more memory than is available by making use of virtual memory.

- **Virtual File System**: is primarily intended to allow user space applications to access different file systems in a controlled manner, regardless of type.

- **Network Interface**: its job is to provide access to various network standards and a wide variety of network hardware.

- **Inter-Process Communication** : this subsystem provides the mechanism to allow processes to communicate with each other. Such communication may consist of the transfer of data or events from one process to another.

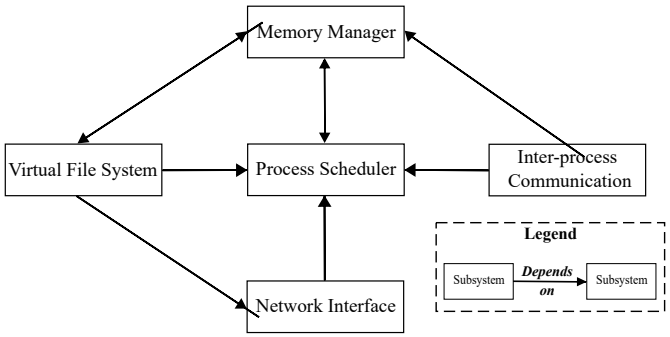


Figure 3.12 – *Overview Kernel system*

All this work is, in principle, **invisible to the user**. That is, while the kernel subsystems are doing their work inside the kernel space, we as users are oblivious to the issue, only interacting with the applications. In fact, kernel space and user space are **two separate areas** within the system's virtual memory. By keeping these subsystems separate, we help keep the system more stable by preventing user applications from directly accessing the hardware. This would mean that if a process crashes in user space, the damage is limited and can often be recovered by the kernel.

But even though these spaces are separate, there are ways for us as users to interact with the kernel. For example, using **system calls** [15]. When we use the shell or some applications such as the desktop GUI or other software programs, we are making the user space interact with the kernel space through system calls. Let's say a program needs to access a file stored on one of the hard disks, it will make a request to the kernel so that it can open the file on the hardware device. This request is done with a system call.

However, you probably don't need to worry about knowing what these system calls are or understanding the details of how they work. This is because the programs we use within user space take our commands, then use system libraries such as the **GNU C Library**, and then with the help of the libraries translate the commands into system calls that the kernel runs. All of which is done behind the scenes. So, we as everyday users, don't even need to think about the kernel being there or worry about system libraries.

3.2.2 Location of modules and Kernel image

Having understood the implications of the kernel, we now move on to researching where it is specified in our system, what parts we can distinguish, etc. The Linux kernel is composed by two main blocks:

- **Linux Kernel Image**: a file containing all the main functionality and drivers that must be loaded at any time to make the system work.
- **Linux Kernel Modules**: is an object file containing the code necessary to extend the functionality of the kernel at run-time, i.e. it is loaded and unloaded as needed and without the need to reboot the system. Most device drivers are used in this form.

Since the Linux kernel is a code file, we can deduce that it will be stored in the system's file system. Thus, every time the system is rebooted, the kernel will be loaded into memory. Normally, on GNU/Linux systems, the kernel is located in the **/boot directory**. Usually, the filename is appended with the version number, so that several kernels can be stored even if only one of them is operational. We can see that this happens in the figure 3.13, as there are **2 images of the kernel** in the directory. This is possible because, in the boot loader, the kernel we want to use is indicated. So, to change the kernel, we would only have to modify the file by changing the name of the kernel version we want, and then reboot the computer.

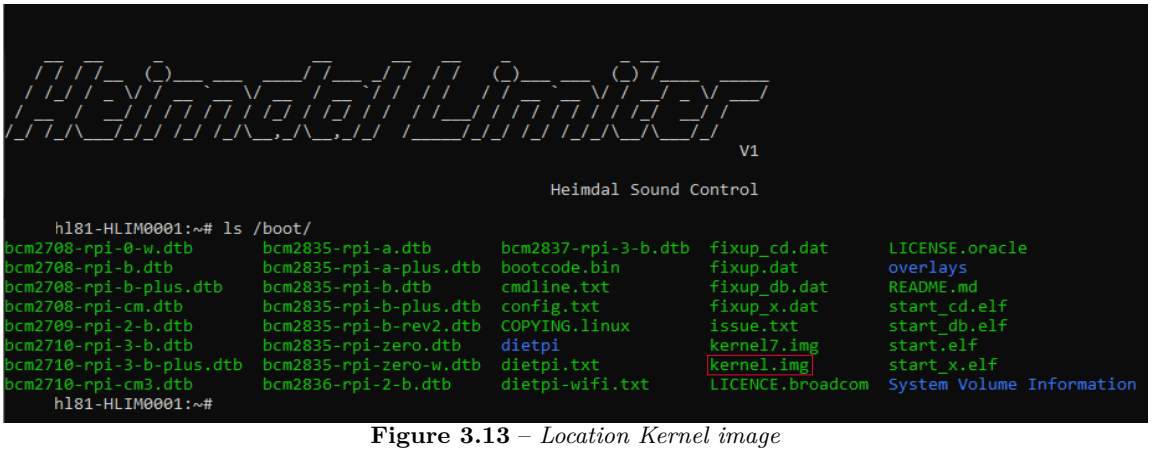


Figure 3.13 – *Location Kernel image*

On the other hand, kernel modules are located in **/lib/modules directory**. Within the kernel directory that we can see in the figure 3.14, we find various folders with files with .ko, .o or .c extensions. They are generally used to support new hardware devices or new file systems, as well as to add system calls. When the functionality provided by a kernel module is no longer required, it can usually be unloaded, freeing its memory. We can **load or unload the system modules** manually using the 'insmod' and 'rmmod' commands respectively. But the most common way to apply changes to the system kernel is by using the 'make menuconfig' command. This command provides a graphical interface to display and modify the kernel configuration file: '.config'. This file sets a series of variables that are used to determine which features are added to the kernel at build time.

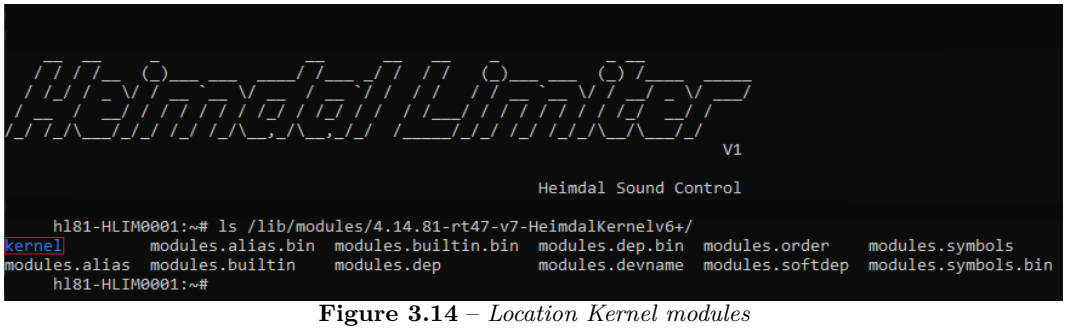


Figure 3.14 – *Location Kernel modules*

This is the superficial idea of what the Linux kernel image and its modules are and where they are located. However, we will go into more detail about them in future chapters.

3.2.3 Analysis of the previous kernel

Once we had a better understanding of the functionality of the Linux kernel, we can proceed to research the current baseline state of the Sound limiters kernel. Once inside the product, we can check that the **current version** of the kernel is **4.14.81-rt4**, this version was released 4 years ago, more precisely in November 2018. In general, the kernel does not have any special modifications, except for **specific modules** that have been added to it to communicate and use the functionality of the product's hardware components. These modules are the ones that are going to give us the most work, as it is essential that they continue to work after the update.

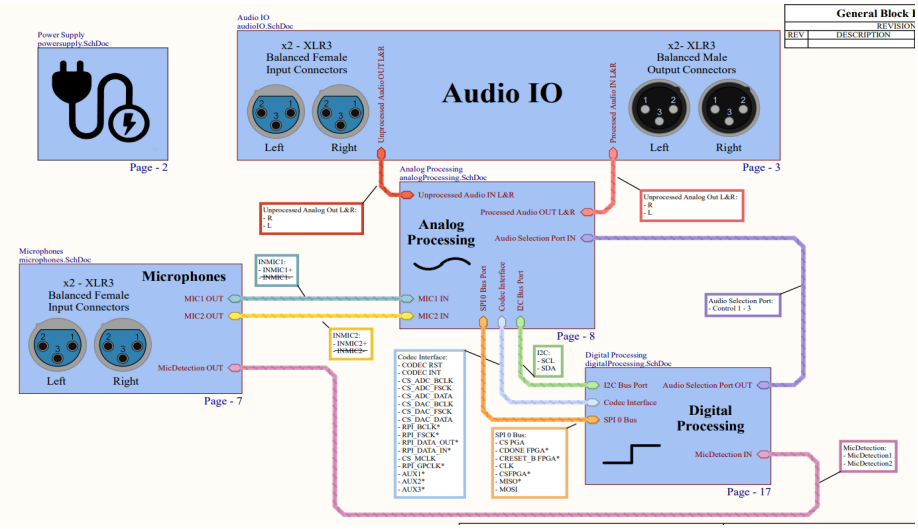


Figure 3.15 – *Sound limiter block diagram*

It is therefore essential to know their purpose first, as we will see how to add these modules to our kernel in next chapters. So, each of the modules is shown below with a brief explanation of their purpose:

- **ENC28j60 Ethernet module**: this module allows us to connect from our product to the Ethernet network, allowing us to send and receive data from a local network (LAN) or Internet (WAN). For it to work properly, we must specify the CM3 pin where it is connected, as well as the transfer speed. In addition, the definition of the SPI bus, through which our component will communicate, must be established.

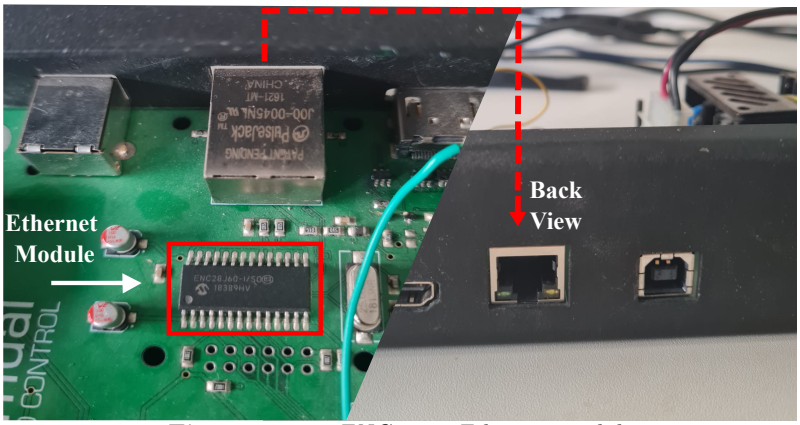


Figure 3.16 – *ENC28j60 Ethernet module*

- **HeimdalSoundCard module**: All the previous modules in the list can be easily found on the internet as they are free open source. But the heimdalSoundCard module is different from the others, as it is a module developed by the GranaSat team. The purpose of the module is to enable the sound card of the product by providing audio input and output signals. For its implementation, our granasat colleagues based the module on two projects. The first one is RPi-DAC by Florian Meier. In this file you can find the necessary code to set up the ASOC driver for RPI-DAC. This device performs the simple function of converting digital audio signals into an analog signals. And the second is Audioinjector Octo by Matt Flax. This project provides instructions on how to use and set up the 'Octo sound card' device (8 channel output and 6 channel input sound cards) that we have installed in our Sound limiters.

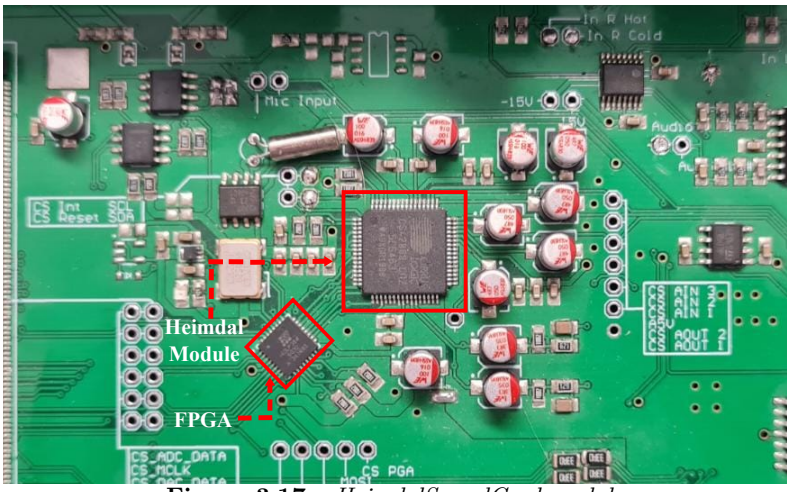


Figure 3.17 – *HeimdalSoundCard module*

- **HD44780 LCD module**: this module allows us to control an alphanumeric dot-matrix liquid crystal display. This display contains various symbols, ASCII characters and even Japanese Kana characters as it is manufactured by Hitachi, a company of the same origin. At present, no tests have been carried out on the LCD except to check that it is properly connected. But in future projects it is planned to provide functionality to the LCD by displaying a small menu with which the user can interact with the Sound limiter.

Figure 3.18 – *HD44780 LCD module*

- **RS232 communication module**: this module enables the appropriate pins and establishes the external communication port to be able to communicate via RS232. RS232 is a standard protocol

used for serial communication, which means the data will be transmitted bit by bit instead of by a byte (8 bit) or a character on several data lines or buses at a time, as parallel communication does.

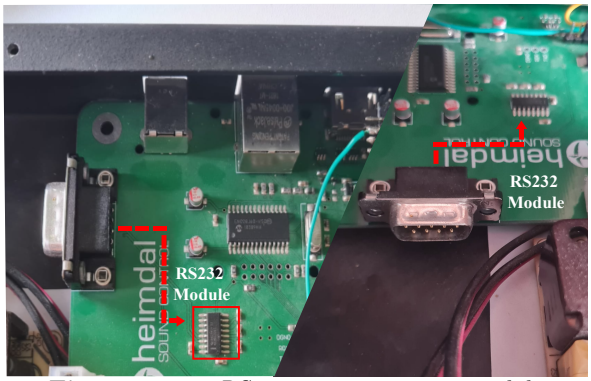


Figure 3.19 – *RS232 communication module*

- **RTC PCF8523 module**: A real time clock (RTC) is an electronic device that allows time measurements to be obtained with the time units we use on a daily basis, such as seconds, minutes, hours, days, etc. This term was created to avoid any confusion with ordinary hardware clocks, which simply measure time through the pulses of a signal. Therefore, this module allows communication with the RTC only by specifying the clock model, and the bus where it communicates (usually the I2C bus).

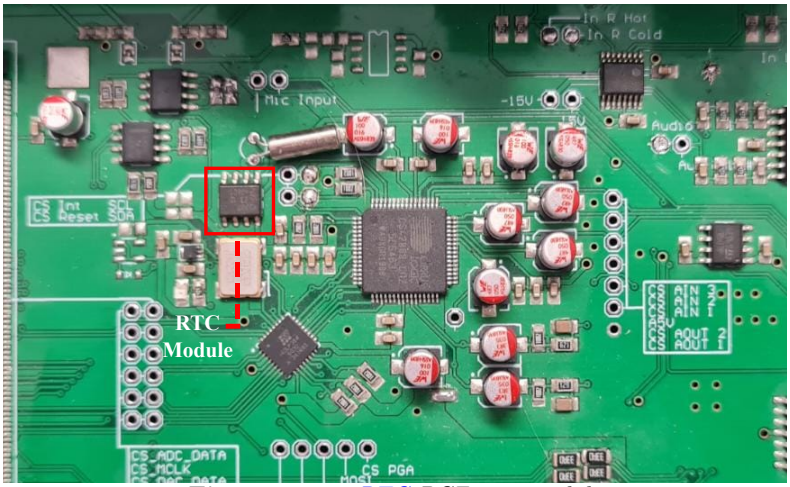


Figure 3.20 – *RTC PCF8523 module*

3.2.4 Benefits of the kernel update

To conclude this section, I would like to state the **reasons** why it has been decided **to update the system's kernel**. This decision may lead to conflict for more than one person, as they may have the thought: "if something works well, it's better to leave it as it is". And they are partly right, updating kernel can cause all sorts of problems, incompatibilities, broken dependencies, deprecated libraries and much more. But on the other hand, performing such an upgrade can be a significant improvement to your system.

The answer will largely depend on the **type of computer** you have and its primary use. On the one hand, if you have an older PC, which has limited usage (a couple of hours a day at most), but which has no problems with hardware interfaces, it is recommended that you stay with the current Linux kernel. But on the other hand, if you have a modern PC that is used a lot, and is connected to the internet most of the

time, then you should probably upgrade to the latest kernel. This will make your PC more secure, faster and have better compatibility with your Linux operating system. Therefore, in our case, the reasons that prompted us to take this decision were the following:

- **Security fixes**: This is one of the most important reasons to perform a kernel update. In most cases, when upgrades are performed and new versions are released, they are more secure than previous versions. This is because they usually fix or resolve security issues that have arisen, or have been discovered, in previous versions. Therefore, knowing the functionality of our product, if we want to provide it with the highest security against possible hackers attempting to attack the integrity of the system, it is highly recommended to upgrade.

- **Organise the product**: One of the fundamental tasks that my tutor wanted to carry out was to organise the product, leaving only what was strictly necessary for its operation. Because before the upgrade, the system space was affected by numerous unused files or software packages, kernel modules for hardware devices that we don't have, disorganised folders containing multiple files that don't correspond to the directory name, and so on. So much so, that using the Linux tree tool, we can see that in the /root directory we have the huge number of files shown in figure 3.21.

- **Kernel documentation**: The current limiter kernel does not have any documentation, so it is not known how we got to this point. Because of this problem, one of the main reasons for the upgrade was to carry out a rigorous documentation of the steps. This will be of valuable use later on, when we want to automate the processes with the Ansible tool. It will provide us with the knowledge to, for example, remotely update the kernel of all our devices.

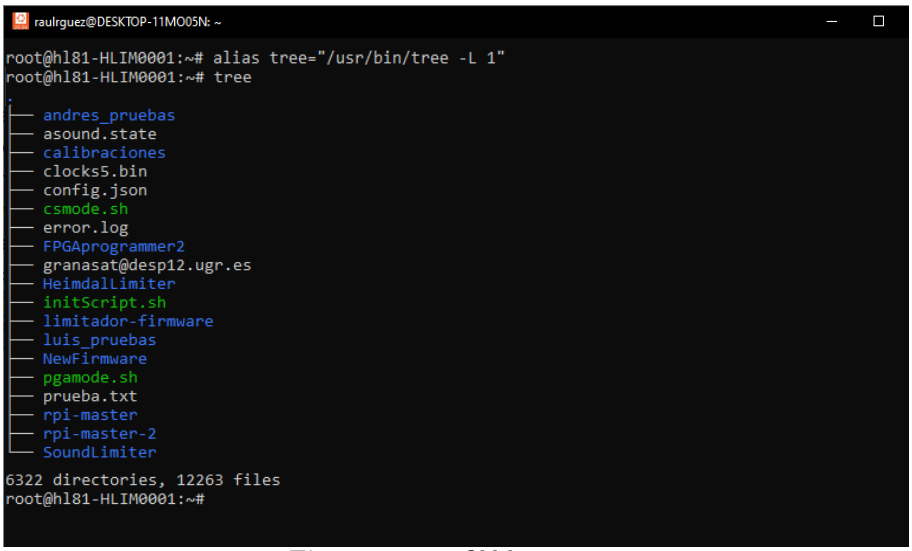


Figure 3.21 – *Old limiter tree*

3.3 Update the product's automated hardware test programs

In the final stages of the software development, **several tests** were created to ensure the correct functioning of each of the product's hardware devices. But some of these tests, due to lack of time, were not created in the right way, were left incomplete, undocumented or simply not checked for correct performance. For this reason, one of my goals for this project is to update, document and check the correct performance of some of the tests.

3.3.1 Overview buzzer script

When I started working with the Sound limiters, I noticed a detail that I was quick to ask my teacher about. It was that every time I connected to the power supply or turned on the limiters, they emitted a **high-pitched beep sound** from the buzzer. This was due to the existence of a script that was executed every time the limiter was **booted**. So now, I will proceed to explain the purpose of the script, and what I had to do with it.

3.3.1.1 Purpose of the buzzer script

This bash script was intended to check the correct communication between the FPGA and the product codec, and we can analyse it by dividing it into 2 parts. First (lines 14 to 30 of list 3.1), the script proceeds to call another program hosted by the limiter. The aim of this program is to send a bitmap to the FPGA via the SPI0 bus, so that it generates the synchronisation signals we need. If the FPGA receives the bitmap without any problems, the program will return 0. This value will be used in our initial script.


```
1  ! /bin/bash
   # InitScript to check correct communication between codec and fpga
   # Documented by: Raúl Rodríguez Pérez
4
   # GPIO1(V91) --> LED1
   # GPIO13 --> Buzzer
7
   # Set the LED pin to output and write '1' to turn it on.
   gpio -g mode 1 out
10 gpio -g write 1 1

   # First we call the program 'fpgaprog'
13 # and wait to see what value it returns
   # return == 0 --> successful communication with FPGA
   # return != 0 --> unsuccessful communication with FPGA
16
   n=0
   nmax=5
19 /usr/sbin/fpgaprog /root/clocks5.bin > /dev/null
   result=$?

22 until [ $n -ge $nmax ] || [ $result -eq 0 ]; do
       /usr/sbin/fpgaprog /root/clocks5.bin > /dev/null
       result=$?
25     n=$[$n+1]
       sleep 0.2
   done
28
   # gpio13 --> Buzzer
   # Secondly, we send a PWM signal to the buzzer
31 # if valor == 0 --> high-pitched sound
   # if value != 0 --> low sound

34 gpio -g mode 13 pwm
   echo $result
   if [ $result -eq 0 ]
```

```
then
    gpio -g pwm 13 10
    sleep 0.25
    gpio -g pwm 13 5
    sleep 0.25
    gpio -g pwm 13 15
    sleep 0.25
    gpio -g mode 12 out
    gpio -g write 12 1
else
    gpio -g pwm 13 3
    sleep 0.25
    gpio -g pwm 13 2
    sleep 0.25
    gpio -g pwm 13 1
    sleep 0.25
fi
gpio -g mode 13 input
```

Listado 3.1 – *Buzzer initScript*

Finally (lines 31 to 55 of list 3.1), if the value returned by the FPGA program is 0, then we will send a PWM signal to the buzzer, causing the buzzer to emit a high-pitched sound as a sign of successful communication. If, on the other hand, the value returned is not 0, we will send a PWM signal to the buzzer, causing it to emit a low-pitched sound to indicate that there has been an error in the communication.

This has been a superficial definition of the program's objective, but we will go into more detail in future chapters. After knowing and understanding the purpose of the program, **our task** will be first to **check its correct functioning** on the updated kernel, since we use several libraries that may have become deprecated. Secondly, we are going to carry out a **study of the PWM signals** [20], checking their duty cycle according to the theory. And finally, we will have to **correctly document** both the initScript.sh script and the fpgaprog.c program, the second one being the one in charge of sending the bitmap to the FPGA.

3.3.2 Develop a test for NeoPixel leds

When Alejandro developed the software for the product, he carried out some **tests to check** that it could communicate with the hardware devices and that they worked correctly. However, due to lack of time, he did not test all the devices, and in some cases, the tests only performed very simple checks that did not provide us with relevant information. So, one of my objectives will be to **provide a test for** a strip of three **WS2812 NeoPixel LEDs**, located at the front of the limiter as shown in the next figure:

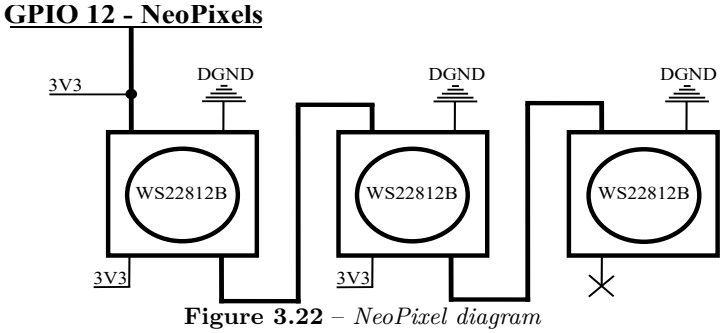


Figure 3.22 – *NeoPixel diagram*

For this task, I will need to **find or create a specific driver** with which I can perform various tests

to know the status of our LEDs. These tests could range from turning the LEDs on and off, varying the speed at which they flash, or simply changing the colours. In addition, all these functionalities should be **well documented** and clear so that anyone who tries to use the program can use it without any problem.

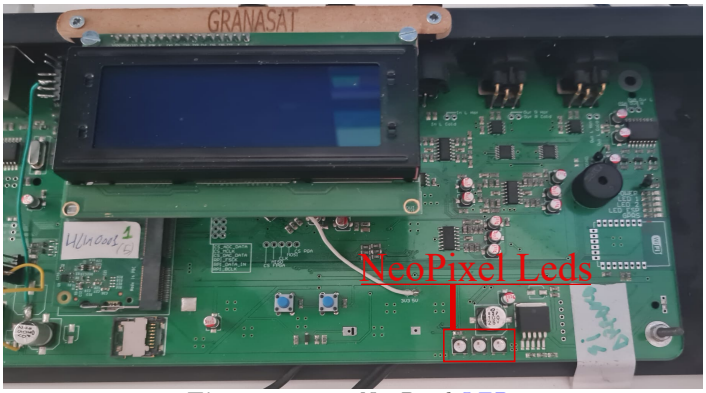


Figure 3.23 – *NeoPixel LEDs*

Finally, I must highlight a very important aspect that I must take into account in order to achieve this goal. Both the tests of the hardware devices, as well as the software created by Alejandro, are written in the **same programming language**, C++. In addition, Alejandro structured the software and the tests in different directories, as we were taught in the computer science degree. This made it easier for him to **create a Makefile** with which he could compile all the software and tests.

Therefore, a priority for this task will be to find or **develop the test** program **in the same language** as the others, i.e. C++. Once it is finished, I will have to add it to the software's Makefile so that it can be compiled along with the others. It should be noted that if I don't manage to develop it in C++, as a last option I could develop it in C, since I would only have to change the compiler and make some small modifications in the Makefile.

Makefile

```
1  # ---------------------- ########## ---------------------- #
2  #      OBJETIVOS TEST
3  # ---------------------- ########## ---------------------- #
4
5  # Compila cada fichero test/*.cpp en build/*.o
6  $(OBJ_DIR)/%.o: $(TEST_DIR)/%.cpp
7      $(CXX) $< -c -o $@ $(CXXFLAGS)
8
9  # ------------------------------------------------------------------------- #
10 #       TEST PGA
11 # ------------------------------------------------------------------------- #
12 TEST_SRCS = $(TEST_DIR)/testLCD.cpp   \
13              $(TEST_DIR)/testPGA.cpp         \
14              $(TEST_DIR)/testMic.cpp         \
15              $(TEST_DIR)/testMicLoop.cpp     \
16              $(TEST_DIR)/testPulsador.cpp    \
17              $(TEST_DIR)/testUtils.cpp       \
18              $(TEST_DIR)/AudioTest.cpp
19 TEST_O = $(TEST_SRCS:$(TEST_DIR)/%.cpp=$(OBJ_DIR)/%.o)
20
21 test: $(TEST_O) $(HW_O)
22     $(CXX) $(OBJ_DIR)/testMic.o $(OBJ_DIR)/Microphone.o     \
23         -o $(BIN_DIR)/testMic -lwiringPi -I$(shell pwd)
24     $(CXX) $(OBJ_DIR)/testLCD.o $(OBJ_DIR)/LCD.o            \
25         -o $(BIN_DIR)/testLCD -lwiringPi -I$(shell pwd)
26     $(CXX) $(OBJ_DIR)/testPGA.o $(OBJ_DIR)/AudioModes.o     \
27         -o $(BIN_DIR)/testPGA -lwiringPi -I$(shell pwd) -DV95
28     $(CXX) $(OBJ_DIR)/testMicLoop.o $(OBJ_DIR)/Microphone.o \
29         -o $(BIN_DIR)/testMicLoop -lwiringPi -I$(shell pwd)
30     $(CXX) $(OBJ_DIR)/AudioTest.o -o $(BIN_DIR)/audioTest -I$(shell pwd)
31     $(CXX) $(OBJ_DIR)/testUtils.o $(OBJ_DIR)/Utils.o        \
32         -o $(BIN_DIR)/testUtils -I$(shell pwd)
```

Figure 3.24 – *Makefile section of Hardware Tests*

3.4 Safety measures for sound limiters

In order to make a **competent product** that fulfils its task one hundred percent, that incorporates innovations that make it stand out from the competition, or that provides customer services of all kinds, several UGR students **have been working on** it for several years. The product does not have much more development time left before it is launched on the market, so we must make sure that all remaining objectives are met without any problems.

One of these objectives, which has not been given special attention in previous projects, is to analyse and take **appropriate security measures** for our product. As we will see in the next sections, our product will be continuously connected to the network, sending information to a production server. This increases the chances of having **security holes**, where hackers will try to harm us or get relevant information from our product to reverse engineer it. Therefore, it is crucial to analyse and take the necessary measures to minimise possible damage to the system.

3.4.1 Security measures from an attacker's point of view

One of the best ways to know what security measures we should choose for our product is to **put ourselves in the attacker's shoes**. We have to think about attacking our system, answering questions such as which parts of it would be most vulnerable, what damage we could cause if we carry out the attack, what type of attack is the most efficient to harm us, etc [29]. Once we have done this mental exercise, we can deduce what the key points of our system are and what measures to take to prevent attackers from taking advantage of them:

- **Logging in to the system**: We must implement measures against unauthorised access to our system. This will most likely be one of the first targets of our attackers, as once they have access to our system, they will be able to harm us in a multitude of ways. To gain entry, attackers will use techniques that allow them to obtain the data needed to breach the system, such as spoofing attacks (via email, telephone or phishing).

- **Find out or modify relevant information**: If our first security barrier fails, and attackers gain access to the system, we must implement measures to protect all sensitive information on the system. Once inside, attackers will be able to modify files, change passwords, investigate to carry out reverse engineering, etc. We will therefore choose to apply measures that make it impossible for them to decrypt relevant information, or make it such a complex task that they will give up.

- **Preventing interference from data traffic**: One of the main tasks of the Sound limiter is to send the data it collects to the production server. Therefore, we must ensure that nobody uses, for example, a packet sniffing technique with us for malicious purposes. To carry out the interference, attacks such as distributed denial of service (DDoS) or injection attacks via code or SQL will be used.

- **Preventing phishing in data traffic**: On the other hand, we must take precautions when verifying the identity of the person sending the data to the server. Because, by means of phishing attacks, they could be sending erroneous data to the server, which would cause a logical failure in the system, harming the users of our product.

So, once we had taken into account the weaknesses of our product, a research was carried out to choose the security **measures to be adopted**. These measures were then discussed with Professor Andrés, in order to check that they met the objective of protecting our vulnerabilities. These measures were as follows:

Vulnerability	Description of the security measure taken
Logging in to the system	Public-private key generation via SSH with ed25519 digital signature algorithm and Disabling serial terminal
Find out or modify relevant information	Only leave binary/executable files on the system (sign them or encrypt them if necessary) and Built-in our modules into the new kernel
Preventing interference from data traffic	Use HTTPS protocol to protect the integrity and confidentiality of our data
Preventing phishing in data traffic	Signing data packets with SSH keys to verify the identity of the sender

Table 3.3 – *Safety measures*

3.5 Software system monitoring and deployment

When the entire design and development phase of our Sound limiter prototype is completed and we have the final product operational, the **marketing and installation phase** in the different bars will begin. This stage raises several questions, such as how we are going to control the correct functioning of the limiters, which tool we will use to monitor them, which problems we will be able to solve online and which we will have to fix in person, etc. To deal with all these questions, my tutor Andrés showed me the proposal I had to follow to complete this objective.

3.5.1 Approach to the proposal

As we have mentioned, each of our products will be hosted in different bars or clubs in the city, and we will have to find the most efficient way to **control and manage them remotely**. This is because, for our part, it would be a bad practice to think that we have to go in person every time there is a problem with the product. In addition, by administering the systems remotely, we save time, costs and provide greater availability for the customer support.

The GranaSAT team has a production server which, among many other functions, is responsible for receiving and storing the data sent by the Sound limiters. The idea of the proposal is to turn the server into the control point for all the limiters we have in use. To do this, using a VPN connection as shown in the figure 3.25, we would establish a communication channel between the limiter and the server, through which the server obtains full access to the device and can carry out maintenance functions.

However, to make the most of this communication channel, we will use a tool for **system administration and monitoring**. This software, among other things, will help us to carry out maintenance or error correction tasks in a much more efficient way. Therefore, one of my tasks will be to research and choose which system administration tool to we are going to use.

3.5.2 Analysis and choice of configuration management tool

Before proceeding with the research to carry out the comparison and, subsequently, the choice of the system administration tool to be used, my professor told me that he wanted **Ansible** to be one of the tools to be compared. This was because some of A. Roldán colleagues were using this software and were very satisfied with the way it worked. Then, I proceeded to search for and compare this software with its closest competition:

Features	Ansible	Chef	Puppet
Architecture	Only Master	Master-Agent	Master-Agent
Language	Python, YAML	Ruby DSL	Ruby, Puppet DSL
Scalability	Very high	High	High
Installation Process	Easy and relative fast	Complex and tedious due to the chef's workstation	Complex and tedious due to master-agent certificate signing
Configuration settings	Easy to learn and set up	Can be complex	Complex, but better if you are a programmer
Configuration management	Pull and Push by a server	Pull by a client machine	Pull by a client machine
Documentation	Well documented but sometimes confusing sections	Highly documented	Highly documented
Released year	2012	2009	2005
Failure Tolerance	Secondary Node	Backup server	Alternative Master
App deployment	Yes	No	Yes, but not easily
Coding style	Procedural	Procedural	DSL
Interoperability (host)	Linux/Unix	Linux/Unix	Linux/Unix
Interoperability (clients)	Linux/Unix or Windows	Linux/Unix or Windows	Linux/Unix or Windows
Pricing Preference	 (1º)	 (3º)	 (2º)

Table 3.4 – *Comparison of configuration and management software*

After a detailed analysis of the three tools, I have realised that they all have their own advantages and disadvantages, and they are better in their own way. This is because each has a **slightly different approach to** automation and management, making them suitable for a particular user sector that requires specific needs.

Apart from fundamental aspects such as architecture, language or fault tolerance, which can be found in table 3.4, I have also analysed other relevant factors. For example, all three tools have both a free version and a paid version. The latter differs in that it provides better features and customer support, with Ansible being the cheapest option. On the other hand, in terms of community and popularity of the tool, they all perform well. But as this article shows [38], since 2019, we have started to notice an increase in users choosing Ansible as their configuration management tool for their systems.

In conclusion, considering all the options, we have decided to use **Ansible** as the tool in charge of the management and configuration of our system. We **made this decision based on** the fact that Ansible has a simpler infrastructure management, its language is easier to learn and use, and my professor had already told me that if the results of the analysis showed very similar features between the options, in case of doubt, this would be the tool to use.

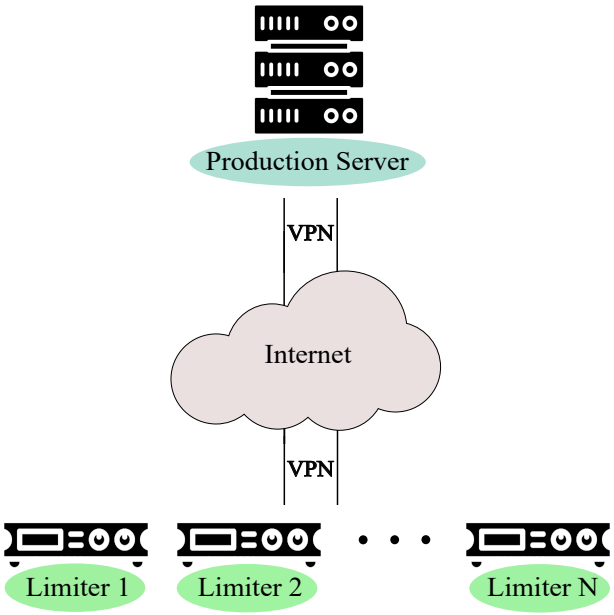


Figure 3.25 – *Provisional diagram of the communication system*

3.6 Update sending data from sound limiters

As I mentioned in section 1.2, where I gave a bit of an overview of the previous work on the project, the limiter **software currently communicates** with a production server and a Sound limiter configuration application. The main purpose of the communication with the server is to store all the information captured by the Sound limiters in a database. But we are not going to comment much on this part as it is not relevant to our work.

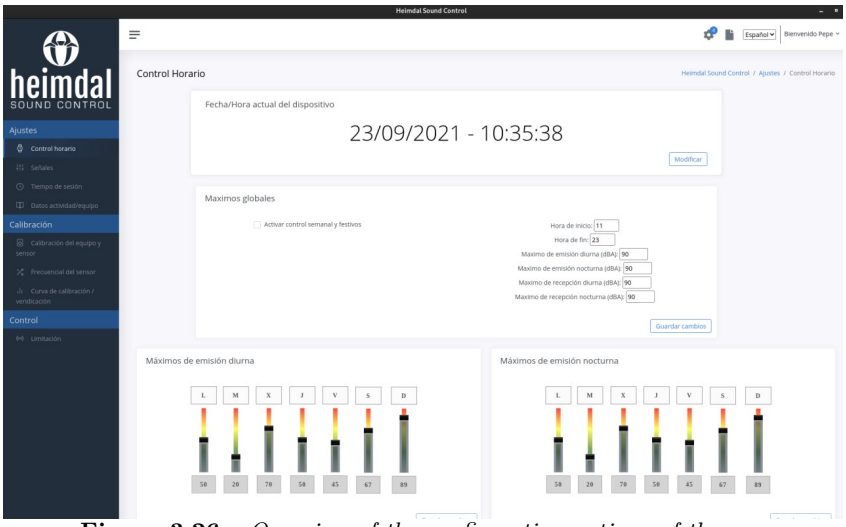


Figure 3.26 – *Overview of the configuration options of the app*

On the other hand, being this the part that concerns us most, we have the configuration application that was made by the student D. Ruiz. This application is **mainly intended for** the limiter technicians, allowing them, thanks to the established communication, to configure the devices. The communication mechanism established between A. Ruiz and D. Ruiz it possible to carry out actions from the application such as **modifying and calibrating** the limiter instrumentation, checking the correct functioning of the hardware peripherals or obtaining technical information from the limiter, among many others.

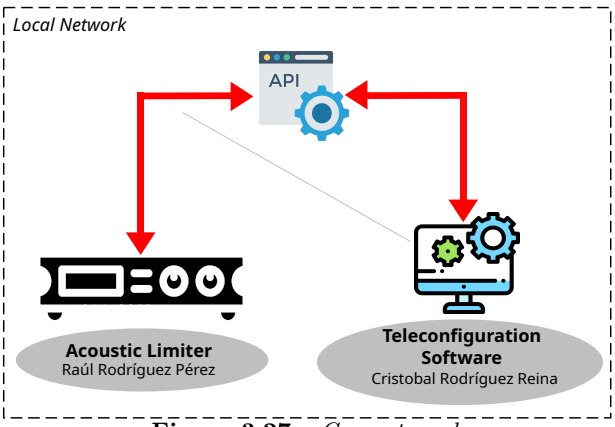


Figure 3.27 – *Current work*

Currently, our colleague C. Rodríguez is **updating the application** as part of his master thesis. He is trying to implement improvements in the frontend part of the application, adding new and more modern menus to the app. But he also aims to update the backend, adding new functionalities that will help him to complement the frontend work. And this is where I come in, since one of my goals is to **update the API of the limiter**, we will have to agree and specify what we want the new features to be.

3.7 Project budget

The aim of this section is to **analyse the costs** of both the manpower and the materials used to carry out this bachelor's thesis. This will provide us with an **estimate of the total cost** that an engineering company in the current market would have to invest in order to carry out the same project. Taking into account the motivation of this section, and the nature of the work itself, a division into **3 parts has been made** in order to achieve the best possible estimate. Separating on the one hand the costs of the physical materials or hardware, on the other hand the materials related to the software used, and finally the manpower costs.

3.7.1 Humanistic resources

In addition to the material used, if we want to make a correct estimate of the price of the project, we must also add the work carried out by us, i.e. **the manpower**. To find out the costs of our job, I have consulted websites such as Indeed, Payscale or Glassdoor, as these websites **provide salary data** in relation to an infinite number of jobs. These sources are considered reliable as the information provided comes from the employees themselves, job offers or directly from companies.

The total price shown in table 3.5 is calculated on the basis that, after consulting the three websites, we obtain an average for a junior software engineer of **15 euros/hour** (with a total of **600h invested** in the project) and for a senior software engineer of **25 euros/hour** (with a total of 240**h invested** in the project).

Humanistic resources	
Job title	**Total cost [€]**
Junior Software Engineer	9000
Senior Software Engineer	6.000
BUDGET	15.000

Table 3.5 – *Project humanistic resources*

3.7.2 Physical resources

The physical resources or hardware used in the course of the project are shown in the table 3.6. This material can be **divided into three groups**, firstly all the hardware resources directly related to the Sound limiters. This group includes both the Compute Module 3 that we have used to carry out the tests, and all the cables necessary to make the limiters work correctly (Ethernet, power supply, USB, etc.), as well as the limiters themselves.

And on the other hand, we have all those devices that have been of help to us or with **which we have carried out** the different tests or tasks of the project. Within this group we highlight the Raspberry Pi 3 Model B, the logic analyser, the soldering station, etc. It should also be noted that all the material mentioned in these first two groups has been provided entirely by the GranaSAT group. But, on the other hand, I have **provided my own personal** laptop computer, in addition to my desktop computer, for the elaboration of the tasks. These were the devices I used on a daily basis, both at home and in the laboratory.

Physical resources	
Item	**Total cost [€]**
4x Sound limiters	4800
Raspberry Pi 3 Model B	40
3x Compute Module 3	108
Logic Analizer	10
Soldering station	80
4 GB MicroSD card	5
General wiring	50
Multimeter	50
Personal Laptop	600
Monitor	60
Personal Desktop Computer	1500
Production server	100
IVA	21%
BUDGET	8.957,63

Table 3.6 – *Project physical resources*

3.7.3 Software resources

Throughout the course of the project, I have been using **different software** (which we can see in the figure 3.7) that have served me for multiple tasks. Among them **we can highlight** for example Visual Studio Code, being the IDE where I did most of the code development. Management and productivity tools such as OpenSSH server, which I used very frequently to perform the tasks of the limiters remotely. Or programs like Discord or Telegram, which I used to be in constant contact with my colleagues and my tutor A. Roldan. I would also like to point out that all the software used has not had any additional cost, as free versions have been used for everyone.

Software resources	
Item	**Total cost [€]**
Visual Studio Code	0
Gitlab	0
Github	0
Docker	0
Cisco AnyConnect	0
Inkscape	0
Telegram	0
Discord	0
PulseView	0
USBboot	0
Ansible	0
DietPi	0
LXQT	0
Openssh server	0
BUDGET	0

Table 3.7 – *Project software resources*

3.7.4 Final project price

Finally, after having calculated separately the costs of each of the groups proposed for the estimation, the final step would be to add them up to obtain the final estimated value of the total costs of the project.

Sub-project	**Budget [€]**
Physical resources	8.957,63
Software resources	0
Humanistic resources	15.000
TOTAL BUDGET	23.957,63

Table 3.8 – *Project budget*

3.8 Project planning

The purpose of this section is to show the strategy followed to carry out the planning of the project. This will be done through the elaboration the following Gantt chart, where it will be possible to distinguish both the total duration of the project and the tasks together with the time invested to carry them out. To create this diagram I made use of the tools offered by the official ClickUp website. So you can see the diagram in the figure below or by visiting this link:

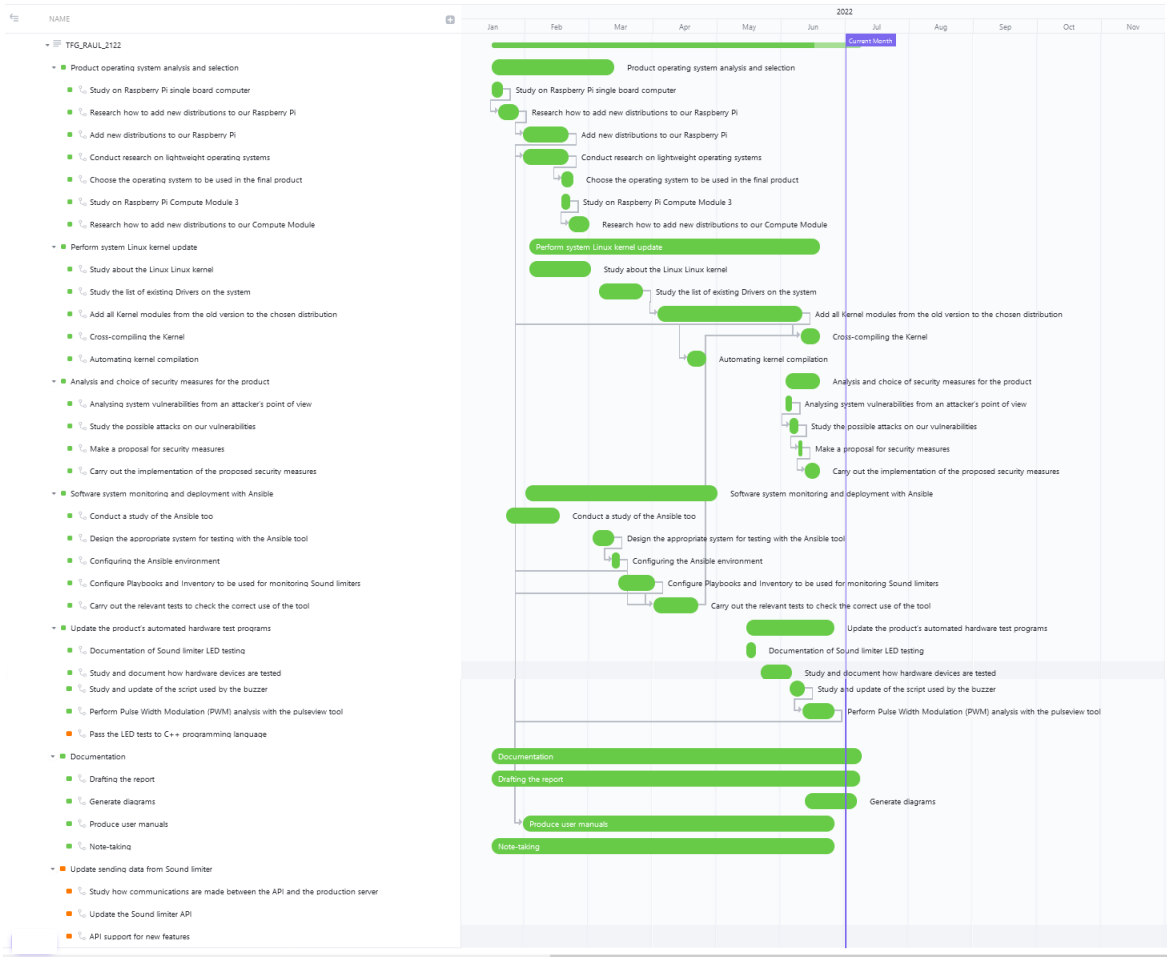


Figure 3.28 – *Project Planning Gantt Chart*

Chapter 4

Design

In the following section, once all the comparisons and decisions of the analysis phase have been completed, we will detail the decisions and the procedure carried out regarding the design of the product in order to meet the objectives of our project.

4.1 Design of operating system tests

In the last chapter, specifically in section 3.1.2, I presented the results of the tests carried out on the different lightweight operating systems in order to choose the most appropriate one for our product. Now that the results are known, I will proceed to explain the **process for the test design**. I will start with a brief description of the material we will need (both physical devices and software tools), and then I will proceed to show the steps we need to follow for the correct development of the tests.

4.1.1 Description of equipment required for testing

To talk about the material we need, we will first outline the **procedures we have designed** to carry out two tasks in relation to the analysis, choice and testing of operating systems. Firstly, we carry out the procedure for using our single-board computer (remember that we are using the Raspberry Pi 3 Model B) as a test device for the various operating systems that we want to compare, in order to analyse them and **choose the one that is most favourable to us**. On the other hand, we carry out the task of incorporating the operating system we have chosen based on the test results of the previous process, in the Compute Module 3 of our Sound limiter.

Once the two tasks are known, we will proceed to show **all the essential materials** we need to carry them out. To show these materials, we will divide them into two groups; on the one hand, we will have the **physical components**. These will be all those hardware devices such as cables, computers, the Raspberry Pi, etc. On the other hand, we will have the **software components**, these will be all those programs or software packages that we will have to use in order to complete the tasks. Once all this is clear, table 4.1 shows the material described:

Task	Material required
Installing and testing OS's on the Raspberry Pi 3 Model B	Physical components: Raspberry Pi 3 Model B, keyboard and mouse, microSD and microSD card reader, HDMI and ethernet cable, microUSB 5V/2.1A for power support and a monitor Software programs: Raspberry Pi Imager, OS image, choosen desktop, Midori browser, Neofetch and Htop
Installing the chosen OS on the Compute Module 3	Physical components: Personal laptop, Sound limiter with Power suply cable, Compute Module 3, USB 2.0 type B to USB type A cable Software programs:OS image and USBboot github

Table 4.1 – *Material required*

4.1.2 Proceeding of the operative system tests

After knowing the materials we need, I will explain step by step the procedure we have design to carried out to fulfil the tasks set out in table 4.1.

4.1.2.1 Installing and testing OS's on the Raspberry Pi 3 Model B

First, choose the operating system you want to test and download its image from the respective official website. After this, to write the OS image to our 4 GB MicroSD card (since that is all the space we have available on the CM3), we will use the software available on the official Raspberry Pi website: **Raspberry Pi Imager**. This program write the image of an OS on our card, offering us by default, a series of images ready to be used in a Raspberry Pi (such as Raspberry Pi OS, Ubuntu, Manjaro ARM Linux and many more). However, if we prefer, we can use our own customized OS image. If you have any questions, check out the official **Raspberry Pi video** about how can you use this tool.

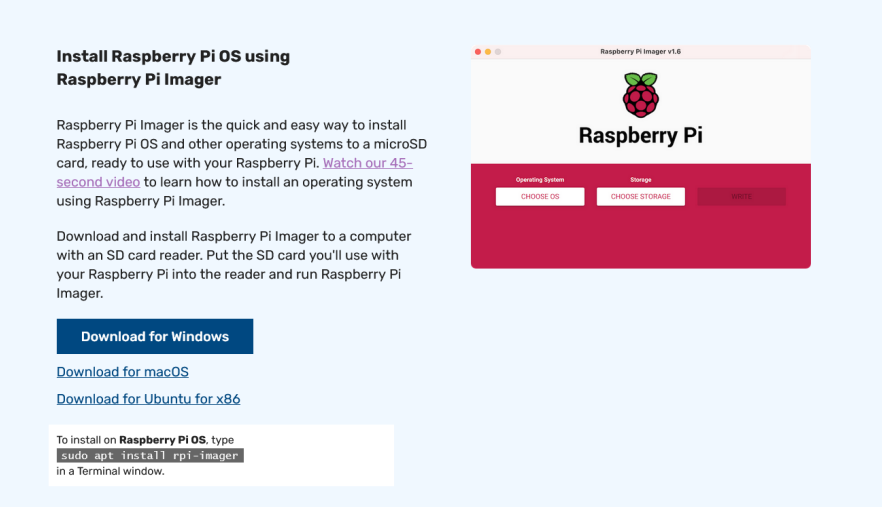


Figure 4.1 – *Screenshot of the official Raspberry Pi website*

Once we have the image written to the MicroSD card, we will have to **establish the setup** of our single board computer. We place the Raspberry on our table and once we have inserted the SD card, we proceed to connect the keyboard and mouse, the HDMI cable to it and to the monitor, the Ethernet cable, and finally the power cable to turn it on.

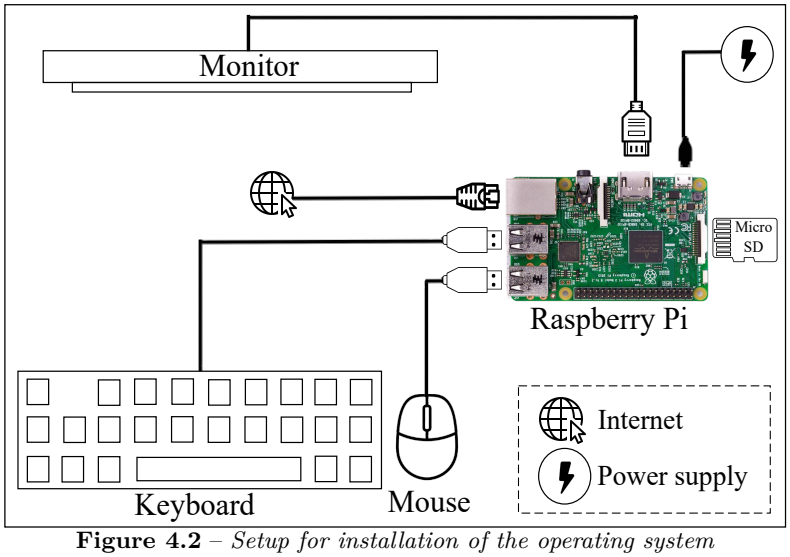
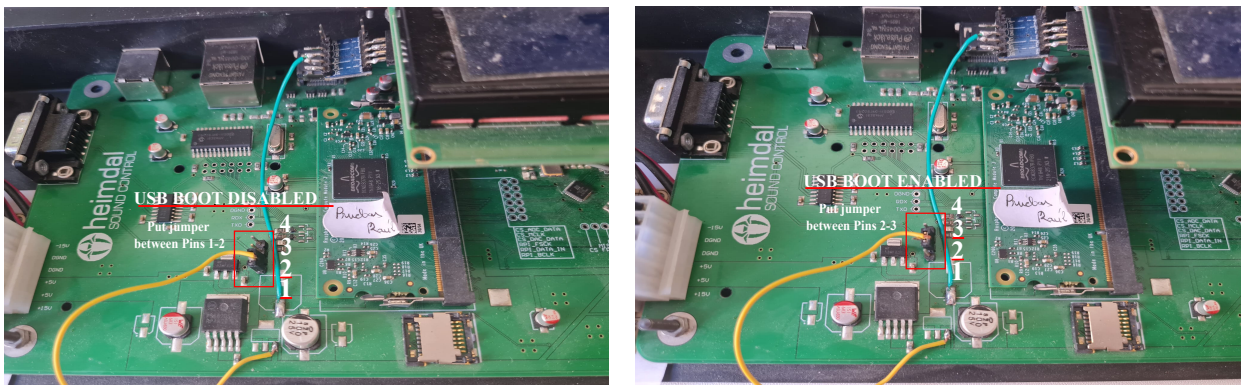


Figure 4.2 – *Setup for installation of the operating system*

Finally, we will have the OS available without a graphical environment. The only thing left is to install what is necessary to carry out the tests designed for the OS: download the chosen desktop, install Midori (a light browser to be used in the memory usage test), install Neofetch (a program that will show us relevant system information), and finally the Htop tool, which will help us to monitor the USB and memory usage in the tests.

4.1.2.2 Installing the chosen OS on the Compute Module 3

Researching through the official Compute Module 3 documentation on the Raspberry Pi website and other technical specifications, I realised that the only option to complete this task was **via USB boot**. This is because the CM3's 4 GB **eMMC Flash** device is connected directly to the main **BCM2837 SD/eMMC interface** [18]. Importantly, these connections are not accessible on the CM3 module pins (they are available for the Lite version CM3 Lite from the SDX pins). This means that, even if our computer board has an SD card reader available, we won't be able to use it because we would have to disable the eMMC device.



(a) *Boot disabled* **(b)** *Boot enabled*

Figure 4.3 – *Jumper pin configuration*

So, as an alternative, the official documentation explains that if CM3 cannot access the SD/eMMC device, it will wait for the boot code to be written via USB. It also provides us with a GitHub repository where we can find the steps to follow to **perform the USB booting** according to the operating system of our computer. Consulting these steps, it is specified that we must **disable the eMMC jumper** of our compute Module IO Board, before turning on the board or connecting the device via USB. So, to find this out, I consulted the schematic of the Sound limiter with P. Belmonte. And to my satisfaction, it specified how to connect the jumpers to get the option we wanted. This specification is given in the figure 4.3.

Finally, once we have correctly configured our board, and following the rest of the steps explained in the repository, we will be able to achieve our goal. In summary, we must **connect our PC** to the CM3 board via USB. Once the computer is connected, with the board turned on and with the jumper properly placed, we will **execute the script** that we have downloaded following the instructions in the GitHub. And after a few seconds, we will be able to access the CM3 files as if it were a pendrive that we have just connected. Being possible to use, for example, the Raspberry Pi Imager tool to write the desired operating system to the module.

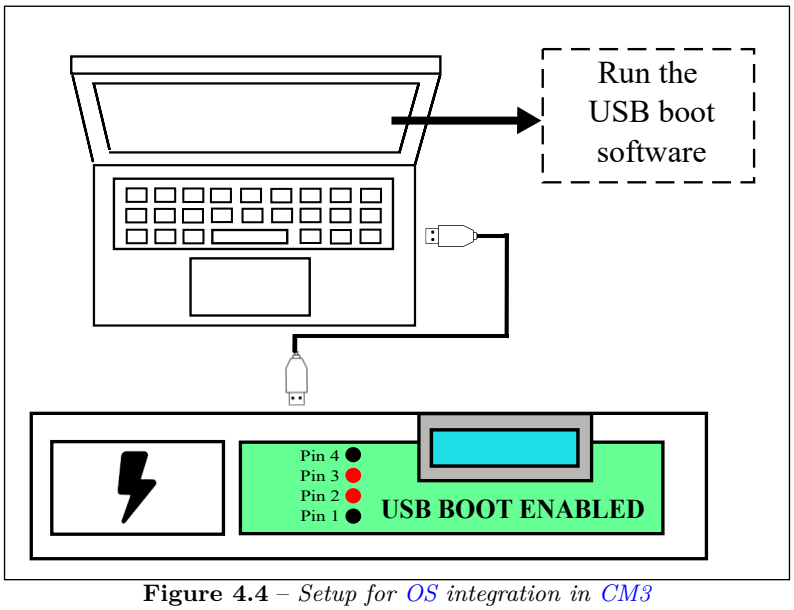


Figure 4.4 – *Setup for OS integration in CM3*

4.2 Kernel update design

To carry out the compilation of the kernel, I decided to **divide it into two stages**. Firstly, I would carry out several tests on the limiter module itself, researching, getting to know and ultimately learning about the Linux kernel and the options for configuring it. At this stage, I would also perform the **isolated compilation** of each of the modules needed in our product, i.e. the ones described in section 3.2.3. Once I have gained experience and knowledge, and have been able to compile all the modules in isolation, I will proceed to **compile the kernel as a whole**.

4.2.1 Isolated compilation of kernel modules

To carry out this step, I have mainly consulted the section on the kernel in the official Raspberry Pi documentation. But really, I spent a lot of time searching, informing myself and learning about the features of the Linux kernel in different forums, articles, videos, etc. First of all, we will need to clone the GitHub repository, where the Linux kernel is stored, in the limiter module. After this, the official

repository recommends that we install a series of tools such as flex or Make, which we will need later to configure the dependencies and the compilation of the kernel.

Once these steps are completed, we proceed to compile each kernel module in isolation. We will go into more detail in the next chapter, but initially, we will make use of the **menuconfig tool**. This will provide us with a menu-driven user interface, and allow us to choose which Linux features we want to be compiled.

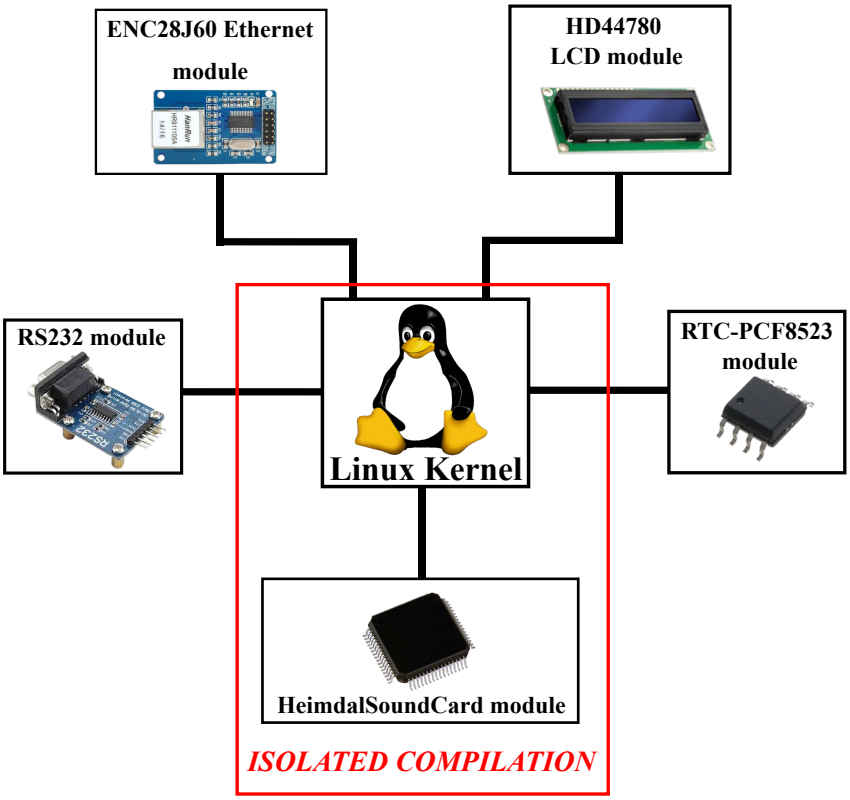


Figure 4.5 – *Isolated compilation of kernel modules*

Once the module is compiled, we will install it in the kernel and check its correct performance. Keep in mind that by **compiling in this way**, we are **not modifying our system's own kernel**. If we reboot the system, the modules we have installed would disappear, as we have not done the complete kernel configuration. In short, we do this because it is the optimal way to test that the modules are ready to be compiled, without errors, and that they work correctly in our kernel. Otherwise, we would have to wait for the full kernel to compile to do these checks, and this is usually a very time-consuming process.

4.2.2 Cross-compilation of the new kernel

In order to really **understand what cross-compilation is** all about, we must first be clear about the compilation concept. The compiler is nothing more than software that converts computer programs written in a high-level language into machine language. This is used so that the computer can understand and perform the task for which the program was created. The process of such a conversion is what we know as compilation. So, once the concepts of compilation and compiler are clear, we can proceed to explain what cross-compilation consists of. This is a **type of compilation** whose purpose is to create executable code for **platforms other than** the one on which the compiler is running.

Compiling the Linux kernel is hard work in itself, and even harder if you only have the four 1.2 GHz

64-bit cores available with Compute Module 3. With such components, we are likely to **need hours** if we want to complete the whole compilation. For this reason, we want to cross-compile the kernel, **using better performing computers** to do the job in much less time. In the next chapter, we will detail the compilation time of the different devices used, such as my laptop, a server or my desktop computer.

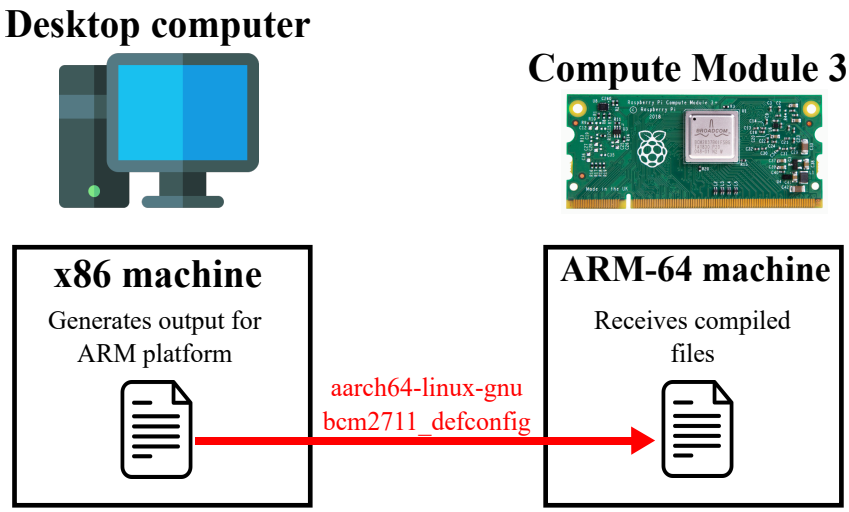


Figure 4.6 – *Overview of cross-compilation*

To carry out this process, I have followed the steps that are reflected in the official Raspberry Pi documentation. In order to be successful during the process, it explain that we must take into account many **relevant aspects for the compilation** such as having a suitable Linux host, known the architecture of our source product or the Broadcom chip it uses, etc. Once compiled, we must add both the kernel image and the generated module files to the boot directory of the system. Once that is done, we only need to change the kernel configuration file and specify that we want to use the new kernel, once we do that, we only need to reboot and we will have updated the kernel.

4.3 Procedure for updating hardware scripts

Regarding what we have said about the buzzer script, there is not much mystery, we will have to test if it is still valid in the new kernel, or if, on the contrary, it has to be modified. But, on the other hand, as we discussed in the previous chapter, this script sends a **PWM signal** to the buzzer making it emit a higher or lower signal depending on a given value. So, in addition to the requirement to check its correct functioning, it will be necessary to **represent and analyse** (from a theoretical point of view) **the signal** that is sent when the script is used.

To carry out this task we will use the Sigrok project with a logic analyser. This project aims to provide a variety of free and open source signal analysis software that supports various types of devices such as logic analysers, oscilloscopes, multimeters, etc. Sigrok provides its users with a set of shared libraries (e.g. libsigrok or libsigrokdecode), with which anyone can build their own GUI or frontend for a specific purpose. Even so, there are already some GUIs that we can use, such as PulseView, which is the one we are going to use for our task.

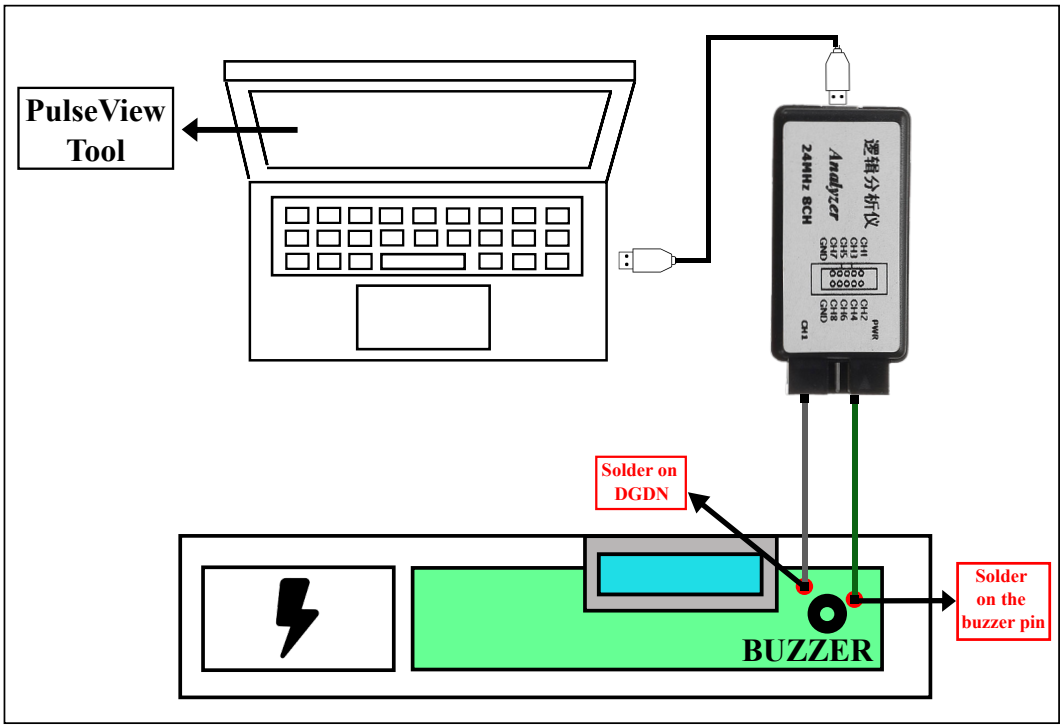


Figure 4.7 – *Setup of logic analyzer*

On the other hand, as I mentioned before, we are going to use a **logic analyser**. The main purpose of this electronic device is to **capture multiple signals** from a system or a digital circuit. Its use is very similar to that of an oscilloscope, but with the analyser we focus more on the digital aspect of the signals, that is, the value that the signal represents over time. In addition, one of its main advantages is its low cost, as we can buy one for as little as ten euros. So to capture the signals, I had to ask P. Belmonte to help me **solder two of the analyser channel** to the limiter board. This way the signal from the Buzzer pin to the digital ground or DGND was captured. Once the soldering was finished and the analyser was connected to my laptop, we could observe the signal with the GUI mentioned above.

Concerning the NeoPixel LEDs driver, there is not much to comment, as I am going to **use an existing driver** and test it on our system. The only drawback is that this driver is written in C language, so once I check its performance and document it, I will have to rewrite it to C++ language (as this was one of the requirements I was asked for).

4.4 Monitored system configuration procedure

Figure 3.25 shows a diagram illustrating how the communications between the production server and the Sound limiters could be, once the Sound limiters are commercialised. But, before reaching that stage, I must **design an isolated system** in the GranaSAT laboratory, in order to carry out the necessary tests to learn how to communicate remotely with the limiters using the administration tool we have chosen, that is, **using Ansible**.

Before proceeding to explain the design of the commented system, I have to clarify the **purpose of using Ansible**, which is to say, what we want to achieve with this tool. Ansible is a technology that provides us with the **infrastructure and guidelines** to carry out the administration and provisioning of systems. Thanks to this tool, we will be able to prepare several servers/nodes simultaneously, that is, we will be able to, for example, install new software, update the kernel, or even take out of service all the

Sound limiters that we have associated at the same time and remotely.

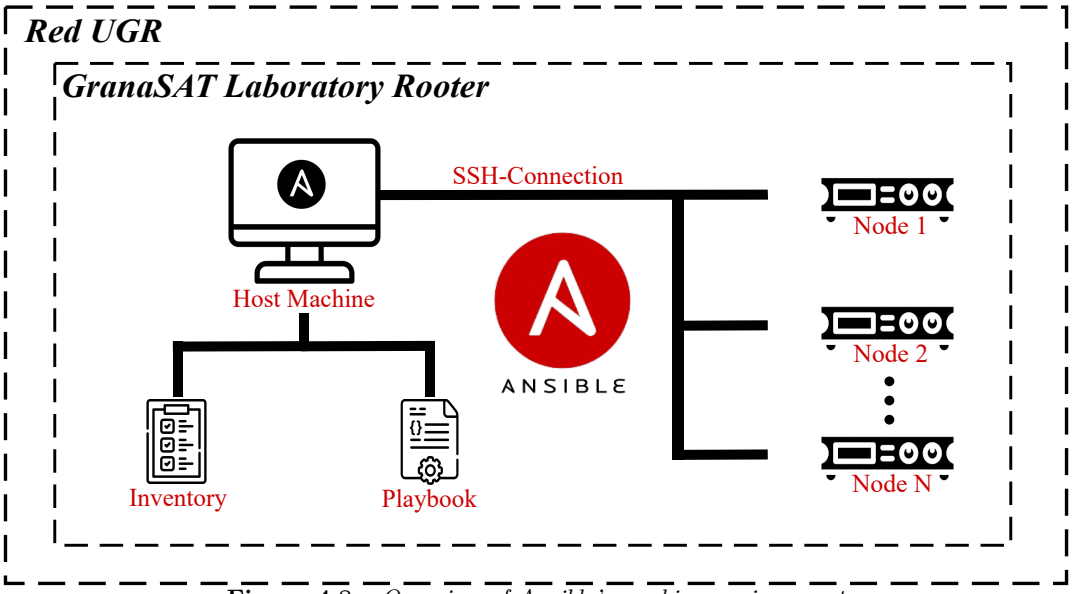


Figure 4.8 – *Overview of Ansible's working environment*

Therefore, we will proceed to **explain the different parts** that an environment created for the purpose of taking advantage of Ansible's features must have. First we have the **Linux/Unix host machine**, which in our case will be my laptop. This host will be in charge of managing the other nodes (in our case the limiters), and it is the only one that needs to have the Ansible software installed. In addition to this software, we will have to design two key utilities provided by Ansible. Firstly, **the playbooks**, which are scripts containing the tasks we want to send to the nodes. And on the other hand, **the Inventory**, where we will have organised the different nodes according to a series of variables that we will be able to modify as we wish.

Once this configuration is done, we need to know **the nodes** that we are going to manage. These are nothing more than the machines that we want to control remotely, in our case, the Sound limiters. The only requirement is that they must have the **SSH communication protocol enabled** and have **Python installed**. With this, it would be enough for us to use them in our system. So, by way of summary, Figure 3.4 shows the working environment with which we are going to carry out the tests. Note that we are not using any VPN as all the parts are under the university network.

Once we know the objective of using Ansible, and we have a clear vision of the system we are going to use to test it, you can get an idea of **what the tests** we are going to carry out will **consist of**. These will focus on the limiters performing, simultaneously and successfully, a series of tasks that are ordered from the host machine.

Chapter 5

Implementation

In the following section, we show the procedure we have finally followed to develop the project objectives, as well as some of the drawbacks we have had, and the working methodology we have followed.

5.1 Working methodology

When developing the project, I decided to use an **agile working methodology**. This is because, although we know at the beginning the objective or the final goal that we have to achieve (update the kernel and implement improvements to the system), the tasks that have to be carried out to achieve it are not so well defined. Therefore, if we combine this fact with my little experience in the field of electronics, we can assume that these **tasks will evolve and change** as we progress.

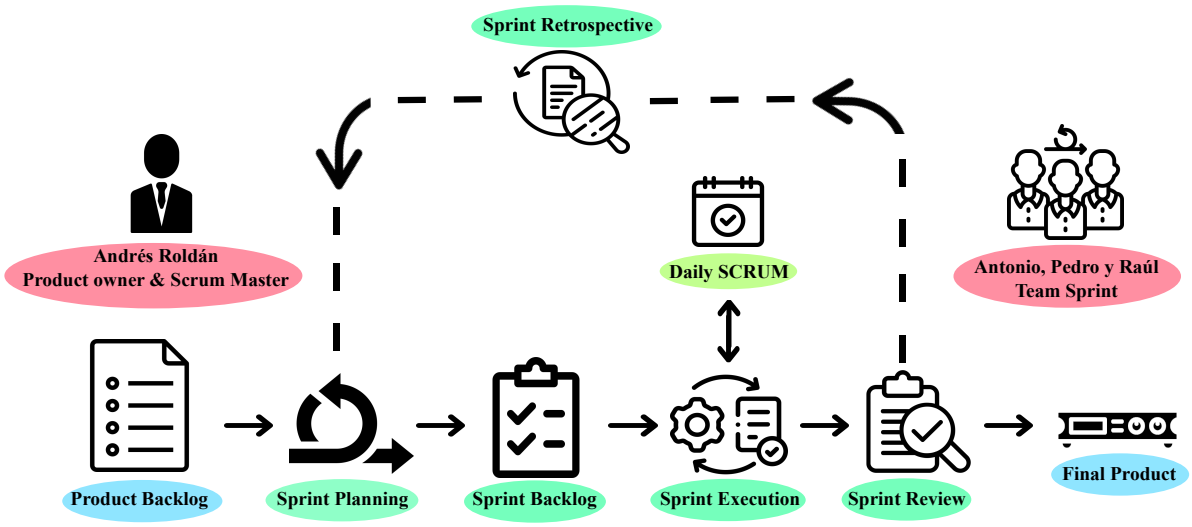


Figure 5.1 – *Scrum Methodology Diagram*

In principle, my idea was to follow the agile methodologies studied in the degree, such as **SCRUM**. But in the end, the facts show that I have **not faithfully followed** the guidelines of this methodology. But, anyway, I have followed this type of agile methodologies in part because my professor A. Roldan assumes the role of **Product Owner** (since he is the one who is responsible for establishing direct communication with our customers) and **Scrum Master** (to perform the work of supervision, and technical decision making in the project). On the other hand, both myself and my colleagues presented in section 1.2.1, would occupy

the role of the **development team**, as we have been in charge of developing the product, communicating with each other and self-organising.

In addition to this, I have also held **weekly meetings every Thursday** in the GranaSAT laboratory. In these meetings, all the members involved in the project participated, and for as long as necessary, we explained to each other the progress or problems we had experienced during that week of work. Note that I speak of **weeks and not sprints**. This was due to the fact that the nature of my project was **insecure and difficult to estimate**, as I had to first find out without documentation what work had been done by other colleagues, and then document it and update it if I saw it necessary Therefore, I found it impractical to set x objectives for each iteration, as I would end up delaying and re-adding these incomplete objectives at each sprint review.

5.2 Objectives development process

The procedure used to meet the objectives of the project is shown below. As the steps are explained, the problems encountered and the tools used will be discussed, explaining everything in much more detail than in the previous chapters.

5.2.1 Product operating system

As we saw in section 4.1, we made a division where on the one hand we explained the tools and the design of the possible steps we would have to take to **test the different lightweight operating systems**. And on the other hand, we explained the tools and steps to carry out the inclusion of the lightweight operating system chosen in the previous step, **to the compute module 3** of our product. So now in this section of the implementation, we will proceed to explain in detail the steps we have finally carried out in these two tasks.

I would like to emphasise that only the images of the process that has been carried out will be shown, with reference only to the chosen operating system, **DietPi**. This will be done for simplicity, as the procedure was the same for all operating systems.

5.2.1.1 Testing OS's on the Raspberry Pi 3 Model B

Before doing anything, the first step I took was to look for information about Raspberry Pi, and specifically about the model we were going to use, Raspberry Pi 3 Model B. Since I had never worked with one before, I wanted to at least get the necessary context to carry out the tests correctly.

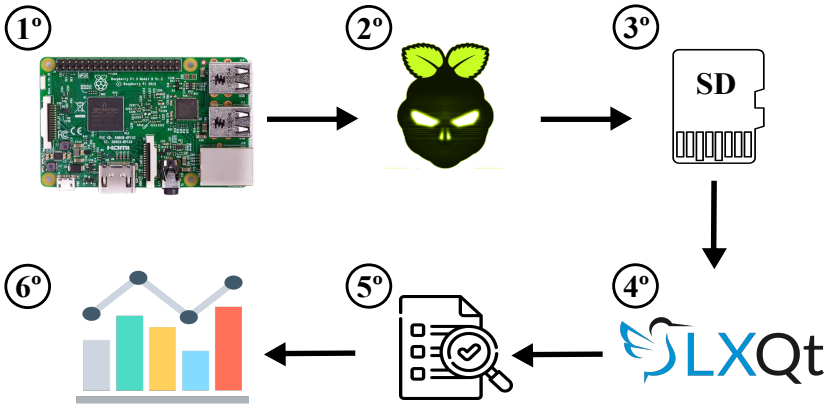


Figure 5.2 – *Tasks for OS testing on Raspberry Pi 3 Model B*

To my surprise, I was struck by the number of people using this device, and not only that, the **immensity of projects** that could be done with it. So apart from watching the official doc section that explains **how to start** using a Raspberry Pi, I also started to investigate on my own videos about the most interesting projects using a Raspberry Pi, such as playing retro video games, turning it into a gaming PC, or turning it into a weather station.

I then prepared to start the tests. First of all, we have to download the image of the operating system we were going to test. In my case, I downloaded the **Dietpi-RPI-ARMv8-Bullseye image** from the DietPi official website.

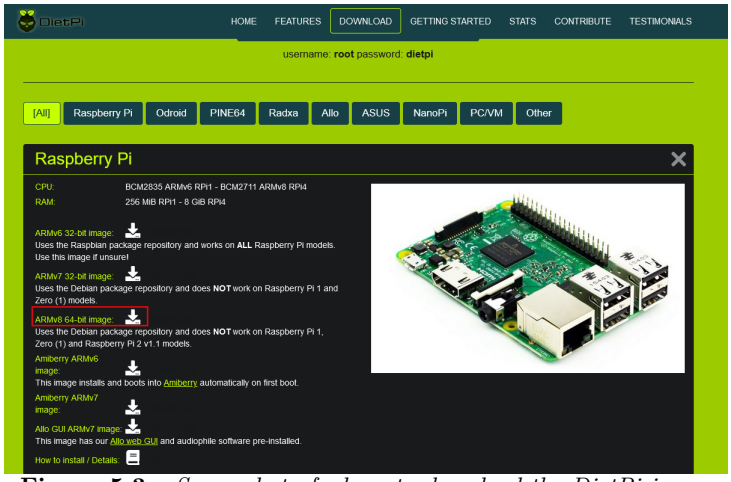


Figure 5.3 – *Screenshot of where to download the DietPi image*

Once the image has been downloaded, we proceed to **connect our microSD card** to the computer. After this we open the application that we discussed in section 4.1.2.1 to record the image on the SD card, **Raspberry Pi imager**. Once the program is open, we will see an interface where we can choose between a series of operating systems offered by the tool, our own custom operating system, or even give us the option to format the card as FAT32. After that, we will be given the option to **choose the device** where to save the chosen image. And finally, we will write the image by clicking on the button at the bottom right.

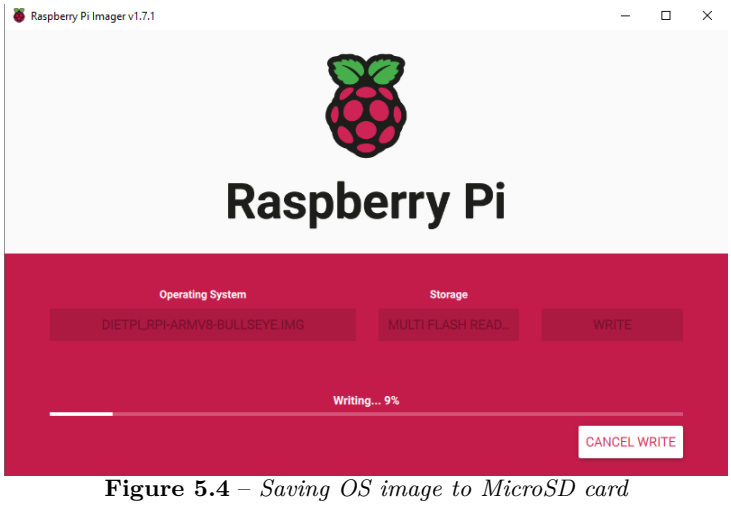


Figure 5.4 – *Saving OS image to MicroSD card*

The next step is to start up our Raspberry Pi 3 Model B with the chosen operating system. To do this, we **connect the Raspberry Pi** to a monitor via HDMI, to the internet via Ethernet, and connect the keyboard via USB. After this, we proceed to install the card, and connect the power cable. After a few

seconds we will see the following on the screen:

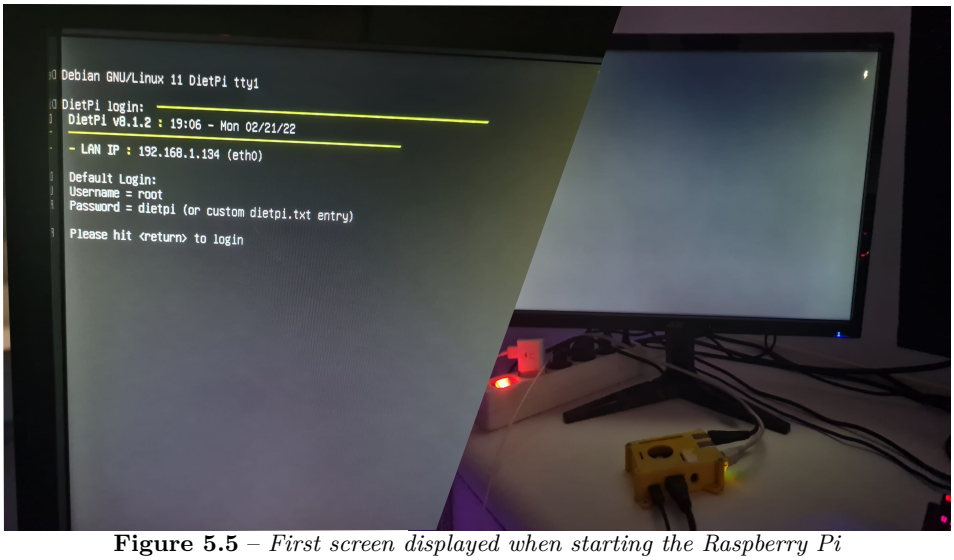


Figure 5.5 – *First screen displayed when starting the Raspberry Pi*

Since we only want to perform the tests, we will log in with the root user to make things easier for us. By default in DietPi, the root user logs in with the following credentials: **User - root, password - dietpi**. After logging in, DietPi will start updating the system (which would be equivalent to having used the apt update and apt upgrade commands). Once the upgrade process is complete, the system will reboot.

Next, a graphical interface will appear, which is the **DietPi-software configuration menu**. In this menu, we can install from a list offered to us or search for any software that we are interested in having on our system. In addition, we can configure key aspects of the system such as; choosing the desktop we want, choosing the SSH server, among others. This is one of the reasons we mentioned in section 3.1.4 why we opted for this OS, as it greatly reduces system configuration time.

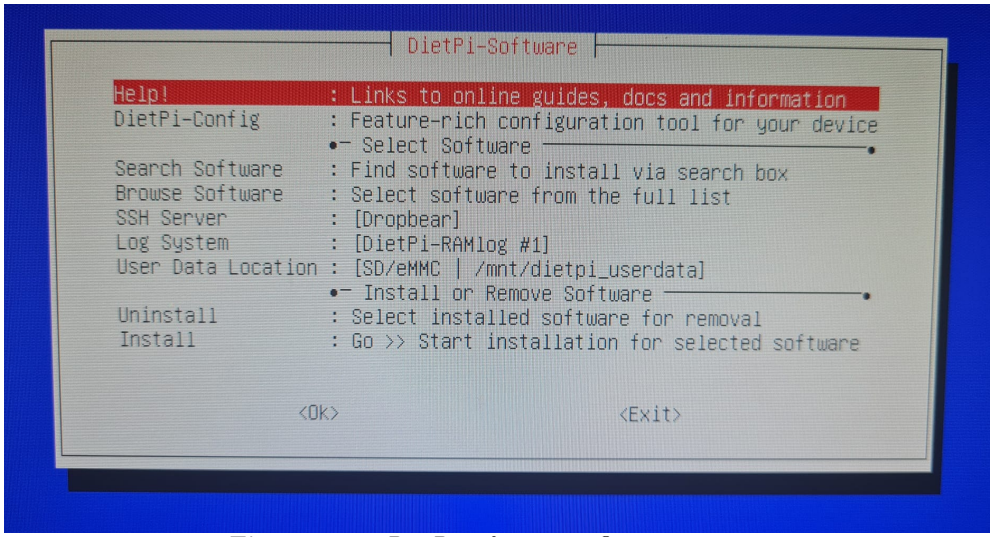


Figure 5.6 – *DietPi-software configuration menu*

At this point we are not going to configure or add anything, so we will exit the menu. Instead, we are going to **download and use some interesting tools**. First we will use **Neofetch** (installation command: apt install Neofetch). This tool will list, in a visually appealing way, relevant system

information such as the Raspberry model used, the OS used, the current kernel version, among others. On the other hand, we will use the **free -m** and **df -h** commands, the first one to observe the memory usage of the system, and the second one to see the remaining storage capacity. The data provided by the tools will be noted down, as they will be used later to make graphs like the one in figure 5.7

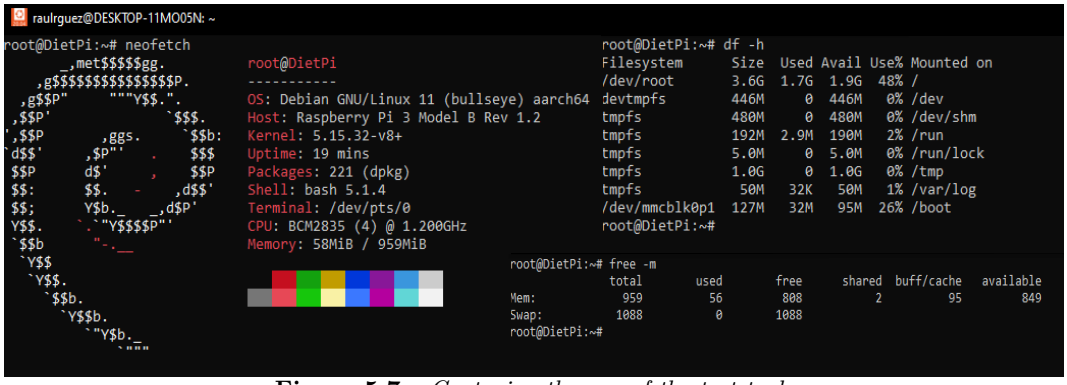
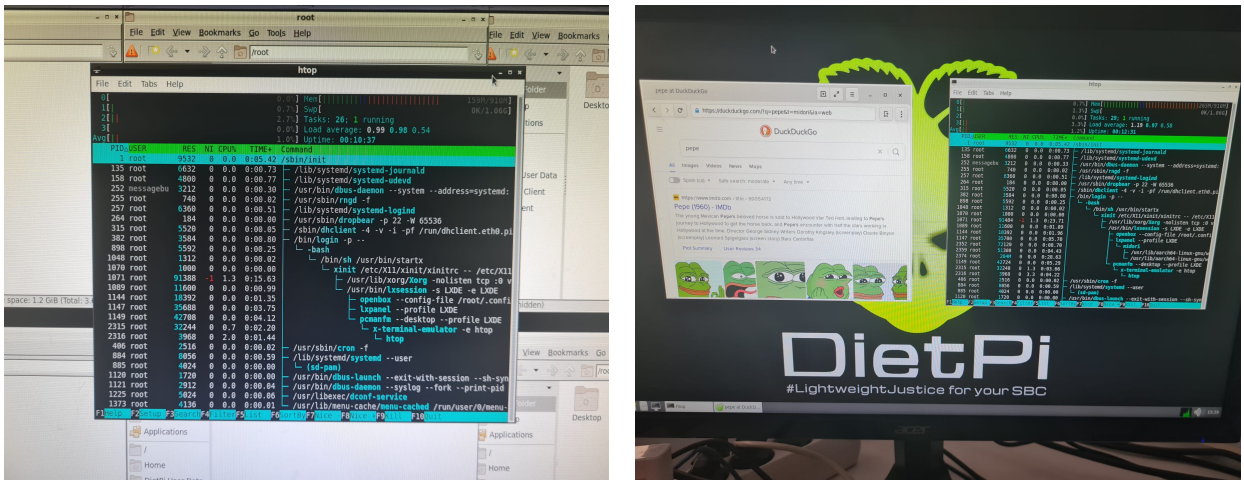


Figure 5.7 – *Capturing the use of the test tools*

Now we are ready to go back to the DietPi-software menu. Once there, we select the option to browse the software, and **mark LXQT as our system desktop**. After that, we click on install and wait for the screen shown in figure 3.2 to appear, where you can see the desktop you have installed. To carry out the installation of the desktops, largely use the information explained on this website.

Next we need to install the **lightweight browser Midori** (command to install: apt install midori), to be ready to start testing. For the tests, we are going to make use of the **Htop tool**, which shows the monitoring of the system processes, memory and USB usage, among other things. Therefore we will first **show the test** without tabs open, then with tabs open and then with the browser open. As mentioned above, all these data were collected and used in the graphs shown in section 5.8.



(a) *Test with many windows open* **(b)** *Test with a browser search*

Figure 5.8 – *Screenshot of the tests*

That concludes the tests, but before I move on to the next section, I would like to comment on one last task that my tutor gave me. The task was to use the **Raspotify tool**. This will allow us to connect to the Raspeberry Pi from our spotify account as if it were a device for listening to music. The idea behind this is that the teacher wants you to **be able to listen to music** from the product, as this could be beneficial in some contexts. To install the tool, I followed the instructions in the official Raspotify GitHub repository.

Figure 5.9 – *Overview of Raspotify's functionalities*

Once installed, I was able to select it as a device from my mobile and listen to songs. As a noteworthy fact, the music was played from the speakers of the monitor I had connected to the Raspberry, but according to the Raspotify doc you can **configure the audio output** device. This is just a test to know that we can play music from our Raspberry Pi device. In the next chapters, when we have done the kernel compilation and all the other tasks, we will test other music playback tools on our product.

5.2.1.2 Installing the chosen OS on the Compute Module 3

As mentioned in section 4.1.2.2, in order to include the chosen operating system in our product, we must do so **via USB boot**. To do this, first of all, we will use the official GitHub repository of Raspberry Pi where they explain how to perform this procedure. We follow the instructions that appear, and proceed to **clone the repository on our laptop**. Once cloned, we only have to enter into the directory and compile to generate the executable to use.

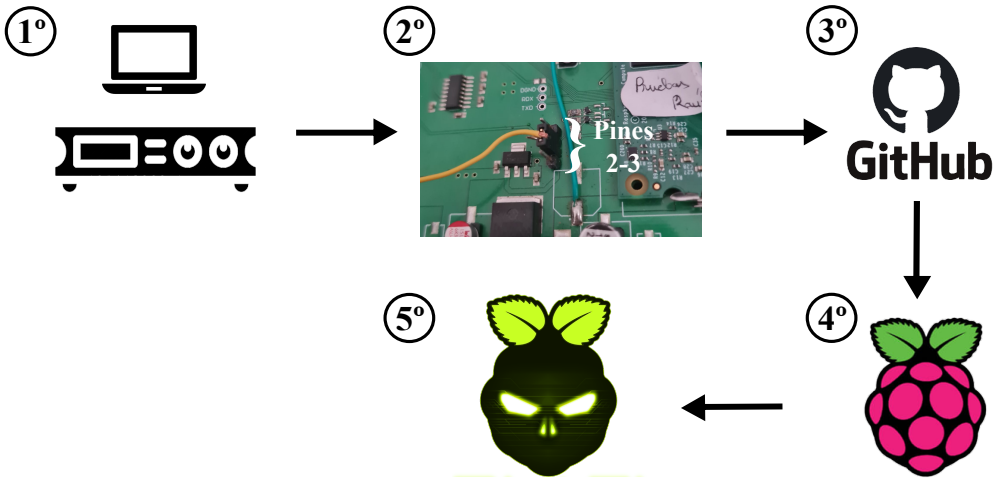


Figure 5.10 – *Tasks for OS testing on the Compute Module 3*

Once this is done, we will proceed to **set the jumper pins correctly**. This was a problem for us as the limiter we were using, SL4, did not have the cable that transmitted the 5V needed for the jumpers to work correctly. This cable was only placed in one of the four limiters we had, so as an extra job entrusted by A. Roldán, and under P. Belmonte supervision, **I soldered the cable in the limiters** that didn't have one. The process I followed to do the welding is shown in annex 2.

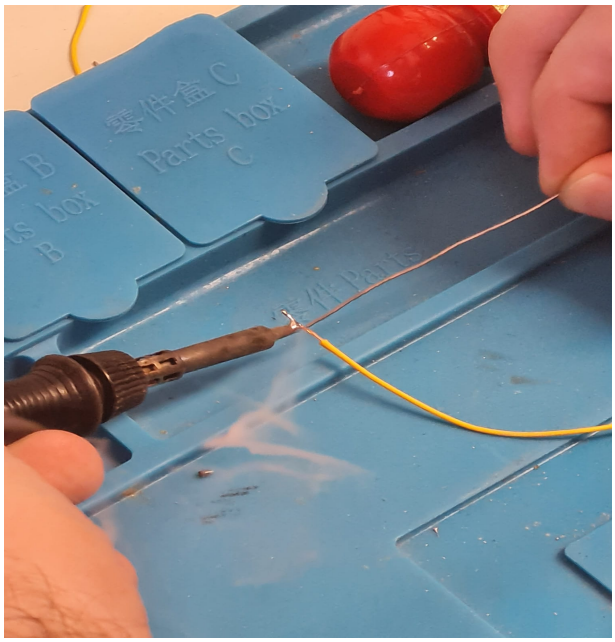


Figure 5.11 – *Welding process on the limiter*

Once the cable was soldered, all that was left was to place the pins correctly, connect the limiter to the laptop and connect the power cable to the Sound limiter. With the system ready, we simply proceed to **run the executable generated** by compiling in the previous phase. A few seconds after running it, a message will appear on our computer as if we had connected a pendrive. And it is true, because if we look at figure 5.12, we can see that the computer recognises the CM3 as such, being able to access its files and modify them.

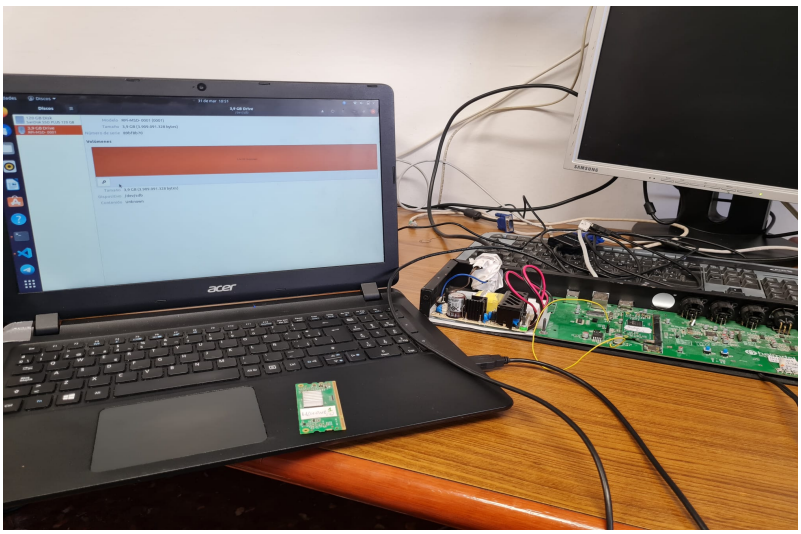


Figure 5.12 – *Test setup in CM3*

Next, as I was using an empty compute module, I simply set out to **write the DietPi OS image on it**. To do this, I made use of the tool we use for OS's, Raspberry Pi Imager. Once I finished writing the OS, I unplugged the limiter from the laptop, **placed the pins to disable the USB Boot mode**, and started the limiter. When I turned it on, I could see that the OS had been successfully installed.

5.2.2 System kernel update

As I explained in the previous chapter, I divided this task in two parts; the first one where I learned how to get by with the kernel configuration and how to add the product modules, and the second one where I compiled the kernel in its entirety and checked the correct update of it.

5.2.2.1 Isolated compilation of kernel modules

We start this task right after the previous section, where by means of USB boot, we manage to add the DietPi OS to our product. After this, we log in and we find the first problem, as the system cannot be updated because it does not have internet. This is because we have not told the system **where the Ethernet module is located** on the board. To do this, we first need to know a bit more about the congif.txt file.

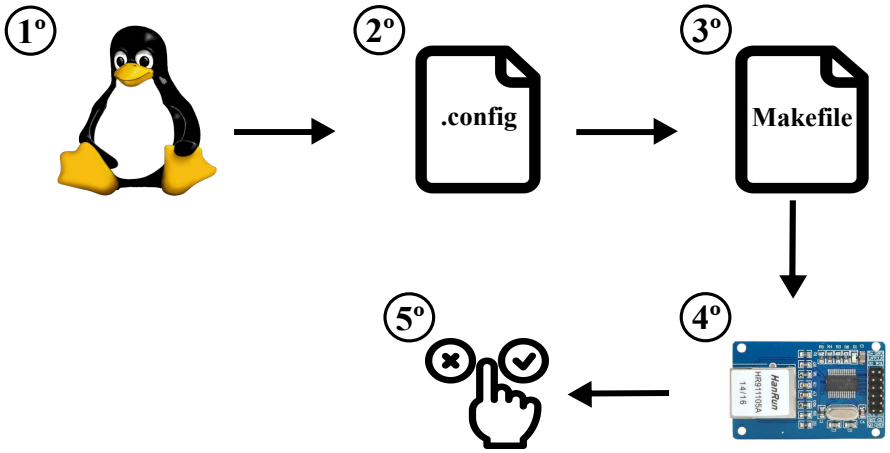


Figure 5.13 – *Steps for the compilation of isolated modules*

According to the official Raspberry Pi doc, config.txt is the file used by Raspberry Pi to set the system configuration parameters, i.e. **it performs the function of the BIOS** in a conventional computer. This file uses a very simple property = value format, as it is read by early stage boot firmware. And more specifically, one of these properties that we can use, and the one we are most interested in for our purposes, **is dtoverlay**. This property allows us to make a request to our system firmware to load a named device tree overlay, which means to load a configuration file that allows the kernel to support built-in and external hardware.

The **Linux device tree** is a data structure that describes the hardware configuration of the system. The kernel will use the information stored in the device tree to manage the hardware devices in the system. So, when we add a dtoverlay property to the config.txt file, what we are doing is **specifying to the kernel** what type of peripheral is connected to the system. The kernel will look for the peripheral in the /boot/overlays folder and then, load that module. Once we are clear about these concepts, what we are going to do is to give the kernel the information it needs to know where our Ethernet module is located. To do this we will add the following lines inside the config.txt file:

```
# Definición de bus SPI (conexión con ethernet a través del chip
 enc28j60-spi2)
# El cs0 (chip-select) se levanta el pin 43 del Compute module 3
dtoverlay=spi2-1cs,cs0_pin=43,cs0_spidev=disabled

# Módulo de Ethernet
# El pin de interrupción del enc28j60-spi2 esta conectado en el pin
 39 del CM3
# La velocidad es de 18000000
dtoverlay=enc28j60-spi2,int_pin=39,speed=18000000
```

Listado 5.1 – *config.txt file*

As we can see, we first specify to the kernel the existence of the SPI bus, since our Ethernet module communicates over this bus. Furthermore, if we look at figure 5.1, we can see that after the module names we add parameters such as pins or speed. This is **a property called dtparam**, which is used to add parameters needed to configure the DT overlays. Basically, the kernel will locate the enc28j60-spi2.dtbo file, and then look for the pin and speed parameters, assigning them the values indicated in the congif.txt file. The format we have used in the previous list, is for simplicity, but we could also put it in the following way:

```
# Definición de bus SPI (conexión con ethernet a través del chip
 enc28j60-spi2)
# El cs0 (chip-select) se levanta el pin 43 del Compute module 3
dtoverlay= spi2-1cs
dtparam= cs0_pin=43, cs0_spidev=disabled

# Módulo de Ethernet
# El pin de interrupción del enc28j60-spi2 esta conectado en el pin
 39 del CM3
# La velocidad es de 18000000
dtoverlay= enc28j60-spi2
dtparam= int_pin=39, speed=18000000
```

Listado 5.2 – *Another form for the config.txt file*

In addition to the ethernet module that we have added along with the SPI bus definition, we are going to proceed to add in the config.txt file the **definition of the UART communication port**. This will be in charge of establishing the RS232 communication. Although it should be noted that at the end of the project, the teacher told me that the RS232 module would probably be eliminated in future prototypes, so we only added the definition of the UART bus and not the RS232 module.

```
# Puerto de comunicación externo para hablar por RS232 +/-12 V
# El txd1_pin se levanta el pin 32 del Compute module 3
# El rxd1_pin se levanta el pin 33 del Compute module 3
enable_uart=1
dtoverlay=uart1,txd1_pin=32,rxd1_pin=33
```

Listado 5.3 – *Adding this lines to the config.txt*

After this **we would have completed the inclusion** of two kernel modules, but from now on things would not be so easy. Since the previous modules were already added in the config.txt of the previous kernel, the only thing I did was to understand their meaning, that is, what they were used for. But now, let's proceed with the inclusion of the real time clock module.

To do this, we will have to tell the kernel that this chip communicates via the I2C bus on the board. We could do this as we have done previously with the activation of the SPI bus or the UART, but to change it a bit and to show more reasons why we have chosen DietPi as OS, we are going to modify it **with the DietPi-Config menu**. In this menu various system configurations such as audio outputs, language, network options, etc. can be made. And in one of the sections, advanced options, we can set the I2C status to on. This will make the system restart so that this option is finally set.

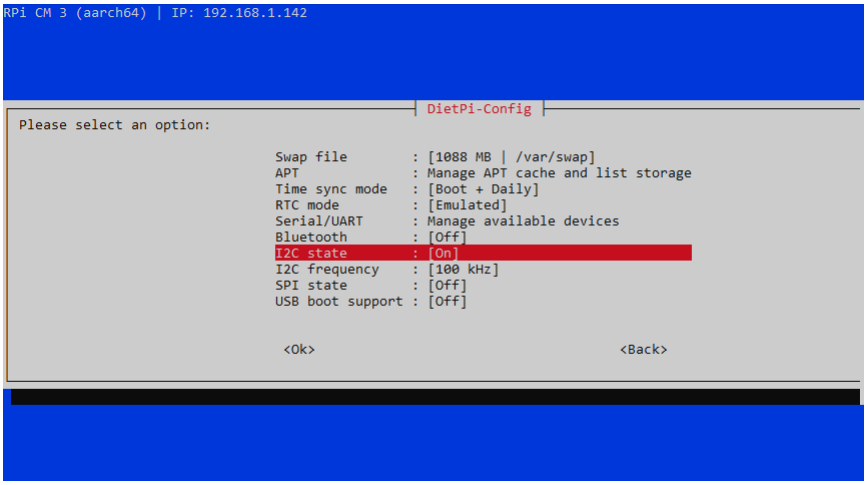
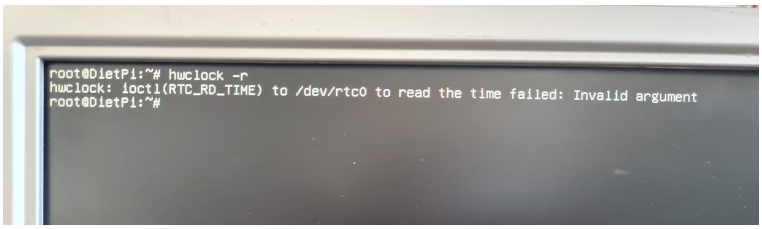
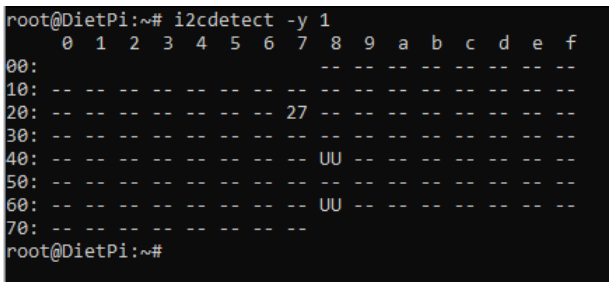


Figure 5.14 – *I2C activation through the DietPi configuration menu*

Before the reboot, the system installed the I2C-tools package, which allows us to make checks such as the following; Firstly, with the i2cdetected -l command, we can consult the list of I2C buses available on our board (in our case we only have one as shown in figure 5.14. And if we use the option **i2cdetected -y 'number'**, we will be able to consult about the slaves that are connected to bus 'number'.



(a) *Hwclock error*



(b) *I2Cdetected command*

Figure 5.15 – *RTC module problems*

After this, consulting the schematic of our board, I was able to tell my colleague Pedro about an error I found. The addresses of the I2C bus slaves were wrong, being the correct ones: 0x68 - RTC, 0x27 - LCD and 0x48 - FPGA. In figure 5.15, we can see that we have 2 slaves with values 'UU', this means that the

address is reserved for a specific driver, being the two that appear for the LCD and RTC (the FPGA is not yet configured but we will do it later). But, although the address is reserved for the RTC, we can see that if we try to check the time of our device with t**he command "hwclock -r"**, we will get the error we can see in figure 5.15 because the module is not included yet.

To fix this error, first of all, I used the **rpi-source tool** [35], as its function was to install the kernel source code to build kernel updates. Following the instructions on the oficial github, we downloaded the kernel to our limiter. But, in the course of testing to compile the RTC module, we discovered two problems. The first one was that originally the rpi-source tool was **used for 32-bit architecture**, not for 64-bit as our system is. To solve it, I found an issue on the official rpi-source github where someone had modified the tool to adapt it to 64-bit.

Once we were able to compile for 64 bits, and after including the module in the system, we still got the error in figure 5.15. Digging into the error, I found an issue in github where they said that we should download an **older version** of the rtc-pcf8523.c file and comment 2 lines in particular, that there was some kind of error with the particular version of the kernel. I then proceeded to look at the commits to find the version of the file that covered those lines, followed the other steps in the repository, and we were able to get good results.

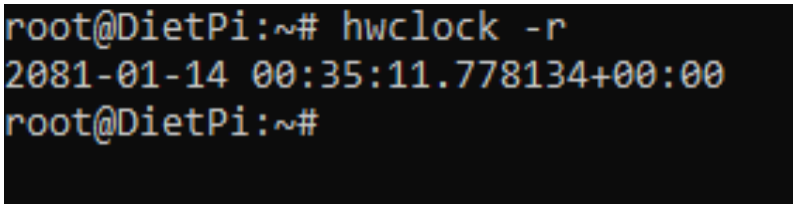


Figure 5.16 – *RTC module working correctly*

Although we had solved the problem, it seemed to me to be a very shoddy solution. So I started researching and looking for **other ways to download the linux kernel** for Raspberry Pi without using rpi-source. And indeed, Raspberry Pi had an official github where you could download the kernel. I then proceeded to download it and try to add the RTC module without changing anything.

So now it's time for me to explain **how I compiled and added the module**. First, I looked for our module in the '/root/linux/' folder that was generated when I downloaded the kernel. This was located in the /linux/drivers/rtc folder. Once I knew where it was, I performed the following steps:

```
# Añadimos esta linea al archivo config.txt
dtoverlay=i2c-rtc,pcf8523

# Hacemos uso de la herramienta nconfig que ya hemos
# explicado en anteriores capitulos
# Navegamos por ella hasta encontrar el módulo rtc-pcf8523 y le
 ponemos la opción < m >
# Dicha opción especifica que se añadirá como módulo al sistema
root@dietpi:~/Linux/$ make nconfig

# La primera opción hace las preparaciones necesarias para poder
# compilar (configura el .config)
# La siguiente compila unas librerias necesarias para la
# tercera opción
# Por último compilamos el directorio donde se encuentra
# nuestro módulo, y le indicamos que use los 4
# procesadores de los que disponemos
```

```
root@dietpi:~/Linux/$ make prepare
root@dietpi:~/Linux/$ make M=scripts
root@dietpi:~/Linux/$ make -j4 M=drivers/rtc

# Instalamos el módulo compilado al sistema
root@dietpi:~/Linux/$

# Comprobamos que se haya añadido correctamente
root@dietpi:~/Linux/$ modprobe rtc_pcf8523
```

Listado 5.4 – *Steps to compile RTC module*

Finally, only one of the modules, **the HeimdalSoundCard**, remained to be compiled. This, as I mentioned in section 3.2, was the most unique of them all as it was a module made by the GranaSAT team, which had the purpose of being able to use a sound card with **8 output channels and 6 input channels**. So, my first task was to look for the .c file in GranaSAT's gitlab where they kept the old kernel. Once in my possession, I proceeded to add it to the kernel, and compile it, but to my surprise, I got several errors in the compilation.

Once I analysed the errors, I realised that they were due to an update of the .h files used in our heimdalsoundcar.c, because of the kernel update. The **main error was the grouping of variables** as we can see in figure 5.17, the variable x and x, would have been included in the same pointer. Knowing this, it would only be necessary to make the appropriate changes to follow the initial logic. And indeed I managed to compile the kernel, but I ran into the inconvenience of not being able to add the module to the system.

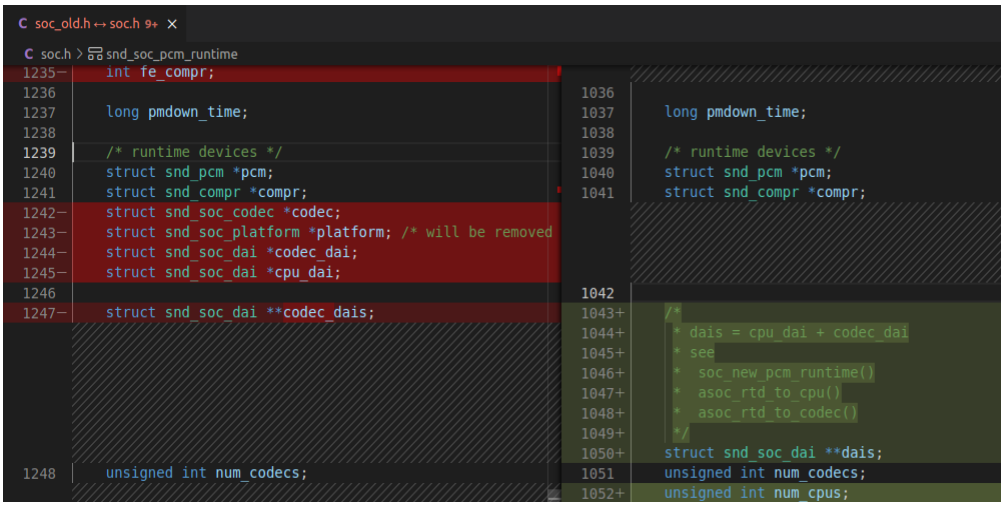


Figure 5.17 – *Changes to kernel files affecting the heimdal module*

With this error, it didn't take me long to realise why it was happening. As it is our own module, which we have configured ourselves, **we have to add its .dtbo file** to the DV overlays in order to give the necessary information to the kernel to include it. Once we did this, we were finally able to add the module to the system. And using the lsmod command, we found that we already had exactly the same modules as the previous kernel, so we would have to move on to the next step, cross-compile the kernel.

```
root@DietPi:~# lsmod
Module                  Size  Used by
rtc_pcf8523            20480  0
enc28j60               45056  0
spidev                 24576  0
snd_soc_heimdalsoundcard   20480  0
```

Figure 5.18 – *HeimadlSoundCard module included in system*

5.2.2.2 Cross-compilation of the new kernel

As explained in section 4.2.2, we will need several hours if we want to compile the kernel directly on our Compute Module 3, so we will resort to cross-compiling on other devices. To carry out this process I have been guided by the official documentation of Raspberry Pi, where the first steps that we explained was to download a number of dependencies and the Linux kernel. After this, we were going to **configure the .config file** of the system, this file is responsible for collecting all information relating to the configuration of the kernel in the compilation as the modules that adds, the type of architecture, etc.

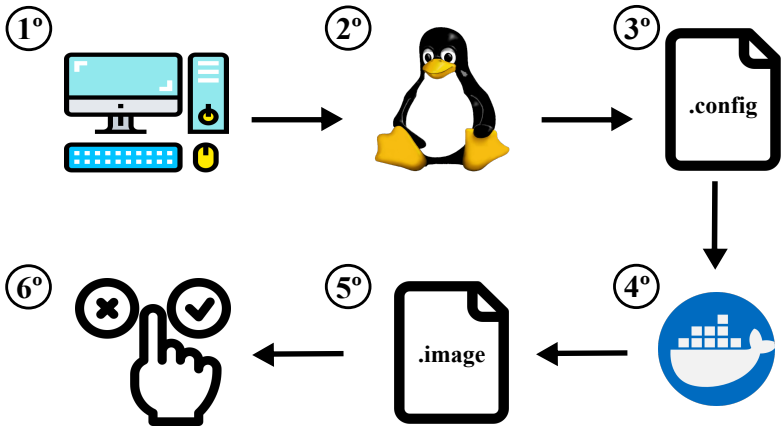


Figure 5.19 – *Steps for the cross-compilation of the kernel*

Once these parameters had been specified, the only thing left to do was to configure the kernel modules, indicating only those we were going to use. For this work, we used an **updated version of the make menuconfig tool** explained in section 5.2.2. This tool is called make **nconfig**, which is identical to the previous one but with a nicer interface design. This tool provides us with the menu of all the modules/drivers contained in the Linux kernel, i.e. it is nothing more than an interface that allows us to navigate through the different kernel folders. In addition, it allows us to indicate which modules to include within the kernel (option <*>), which ones compile as a separate kernel module (option <m>), or which modules not to add (option < >).

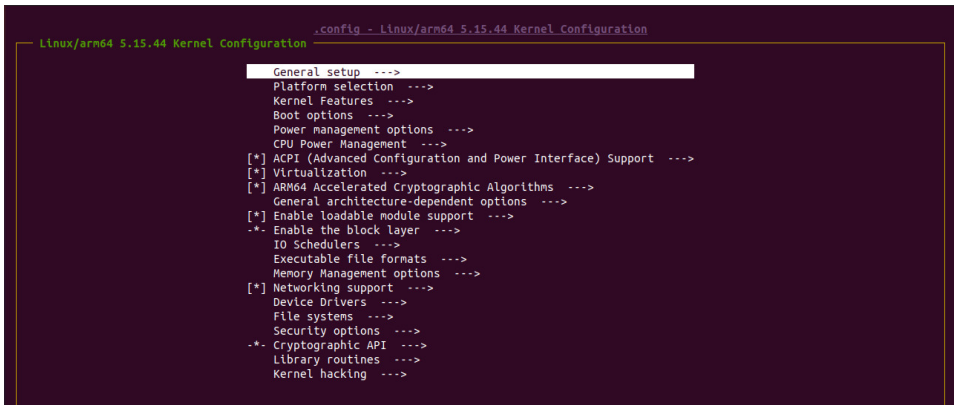


Figure 5.20 – *Overview of the nconfig tool*

Once we know the tool, we use it to add the modules we need to our system. With the nconfig menu, we can also **search directly for the modules we want to add**, by pressing the f9 key, it will give us the option to search for a specific module. We will use this function to search for and add the RTC and HeimdalSoundCard modules. Note that previously, we have manually included the heimdalsoundcar.c file and its .dtbo, as well as modifying the Kconfig and Makefile files. The **Kconfig file** is the one used by our

tool to know which modules are available in a specific folder. If we did not modify this file, the option to add the heimdal module would not appear. On the other hand, it is necessary to modify the Makefile to compile the module.

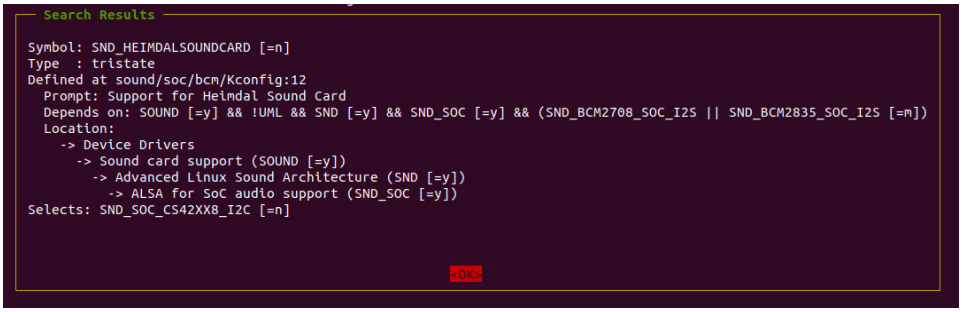


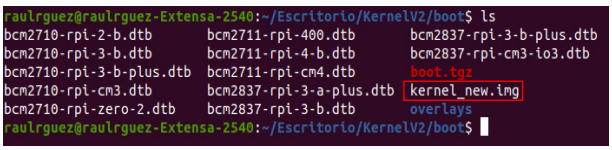
Figure 5.21 – *Searching for the heimdal module with nconfig*

Once we have everything configured, we proceed to compile the kernel. This process, as I mentioned before, usually takes quite a long time. But to **reflect the speed differences** depending on the characteristics of our device, I made several compilations on 3 different devices. Firstly on my laptop, secondly on one of the GranaSAT servers, and finally on my desktop computer. The results obtained, which reflect the time it took to compile the kernel on the various devices, are as follows:

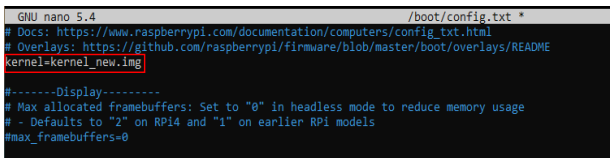
	Personal Laptop	**Production server**	**Desktop computer**
Features	Intel Core i5-7200U 4-cores 2.50 GHz	Intel Pentium E2180 2-cores 2.00 GHz	AMD Ryzen 7 3700X 8-Core 3.60 GHz
Compilation time	1 hr 20 min	2 hr 12 min	6 min

Table 5.1 – *Comparative table of kernel cross-compilation times*

Once the compilation was finished, and after installing the modules, we had what we needed to **add the updated kernel to our system**. So we only had to find the necessary steps to include the new kernel in the system. To carry out this process, we would have to create a folder where we would add both the **kernel.img file** generated after the compilation, and the overlays folder along with **all the compiled .dbo** files. These files should be added to the /boot directory of our limiter. Once added, we only had to modify the config.txt file and indicate that we wanted to use the new kernel and reboot the system.



(a) *Locating the kernel image*



(b) *Add new image to config.txt file*

Figure 5.22 – *Kernel image for the upgrade*

Although this would have been enough, I found it very annoying to have to repeat and go through all the steps every time we wanted to cross-compile the kernel. So with A. Cantudo help, **we made a dockerfile** that would allow us **to establish the same compilation environment** regardless of the device we were cross-compiling it on.

This environment is composed of a series of files. First we have **the Dockerfile** that we can see in figure 5.23. In it we can see a series of instructions which we use to; establish that we start from the Debian image, install the necessary packages to have the same compilation environment in all sites, and execute **the docker-entrypoint.sh** which is the file that contains the different steps to compile the kernel and add the necessary files to include it in our product.

```
Dockerfile
Dockerfile > ...
1  FROM debian
2
3  RUN apt update && apt install git bc bison flex libssl-dev make libc6-dev libncurses5-dev crossbuild-essential-armhf crossbuild-essential-arm64 sshfs -y
4
5
6  COPY ./docker-entrypoint.sh /
7
8  RUN chmod +x /docker-entrypoint.sh
9
10 ENTRYPOINT ["/docker-entrypoint.sh"]
11
```

Figure 5.23 – *Overview of dockerfile*

Next we have t**he docker-compose.yaml**, which is in charge of mounting in the container the host folders where the kernel files and the configurations we need are located. Finally, as mentioned above, the file docker-entrypoint.sh, contains the collection of all the steps to compile the kernel and add the resulting files to a folder. Therefore, thanks to using this docker, **we will only have to manually modify the kernel configuration** with the nconfig tool, since the rest of the steps are collected and automated in the docker.

```
docker-compose.yaml
docker-compose.yaml > version
docker-compose.yml (compose-spec.json)
1  version: '3'
2
3  services:
4      limiter-kernel-build:
5          container_name: limiter_kernel_build
6          build: .
7          volumes:
8              - ./linux:/linux
9              - ./config_kernel:/config_kernel
10             - ./build:/build
11
```

Figure 5.24 – *Overview of docker-compose.yml*

```
$ docker-entrypoint.sh
$ docker-entrypoint.sh
1  #!/bin/sh
2
3  cd /linux
4
5  # sleep 999999999999
6
7  ##### CUSTOM CONFIGS #####
8
9  cp /config_kernel/configs/bcm2711_defconfig /linux/arch/arm64/configs/bcm2711_defconfig
10 cp /config_kernel/dts/* /linux/arch/arm64/boot/dts/overlays/
11 cp /config_kernel/heimdal_module/* /linux/sound/soc/bcm/
12
13
14 KERNEL=kernel8
15 OUTDIR=/build
16
17 rm -rf /build/*
18
19 make clean
20 make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2711_defconfig
21 make -j16 ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules dtbs
22 #make -j2 ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- modules
23
24 cp arch/arm64/boot/dts/broadcom/*.dtb $OUTDIR
25 cp -r arch/arm64/boot/dts/overlays/ $OUTDIR
26 cp arch/arm64/boot/Image $OUTDIR/kernelgucci.img
27
```

Figure 5.25 – *Overview of docker-entrypoint.sh*

With the docker ready, we compiled the kernel again with my desktop computer as it took very little time to do so, and added the new kernel to our SL4 limiter. Apparently after the reboot, everything was fine, **we checked with an lsmod that all the modules** were there and indeed they were. After that we checked that we could check the system clock with hwclock -r, and indeed it also worked. But, unfortunately to my surprise, when we went to check with t**he alsamixer utility** the audio inputs and outputs of the system, **a problem occurred**as we couldn't open the tool because it didn't recognise any sound card.

On this occasion, the error left me in shock, as I didn't understand what could be going on and when I searched for the error on the internet, it didn't make the situation much clearer either. I finally deduced that **it was a logical failure in the heimdal module**, as this was the one in charge of setting the sound card. To check what was wrong, we looked at the system logs.

```
root@DietPi:~# alsamixer
cannot open mixer: No such file or directory
root@DietPi:~#
```

Figure 5.26 – *Error in alsamixer command*

In the **system logs** we could find that when starting the system, more specifically when trying to add the code, it gave an error because it made a reference to a null pointer. Therefore, I proceeded to **review all the modifications I had made in the heimdalsoundcard.c file**, in case I had not initialised any variable and that is why it was referring to null. I didn't find the error, and I had to manually debug the code to try to find out at least where the error was occurring.

After being stuck for a few days, I decided to **compare what we had** in the heimdalsoundcard.c, **with the octo audio-injector file** from which we based our own. This way I would have a better understanding of what we were doing, and maybe I could understand where the error was coming from. Finally, I realised that **the audioInjector had some lines of code that our file didn't**, and those lines matched the error I was getting, as they defined the USB. So by adding those lines, modifying them a bit, and making a couple of changes, I finally managed to add the module correctly without errors. I was able to access the alsamixer and check our audio inputs and outputs:

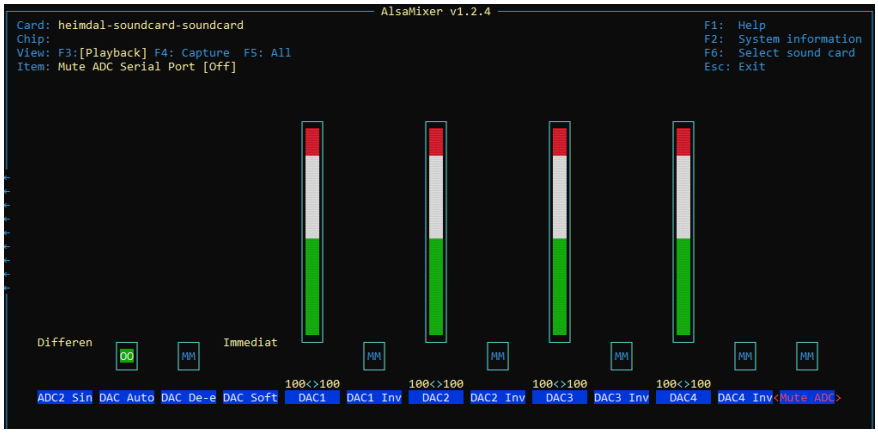


Figure 5.27 – *Showing audio inputs and outputs via alsamixer*

So, by fixing this bug, we had completed our main task, the kernel update had been successful.

5.2.3 Update the product's automated hardware test programs

The first step I took for this task was to check that the buzzer script still worked in the new kernel. But as expected, it's never as simple as it seems, so **the script was indeed not working in the new**

kernel. The script made use of the WiringPi library, which was in charge of accessing the GPIO pins of our Raspberry Pi. But unfortunately, **this library was deprecated** for the version of Debian we were using Debian 11 bullseye. So we needed to modify the script using another library.

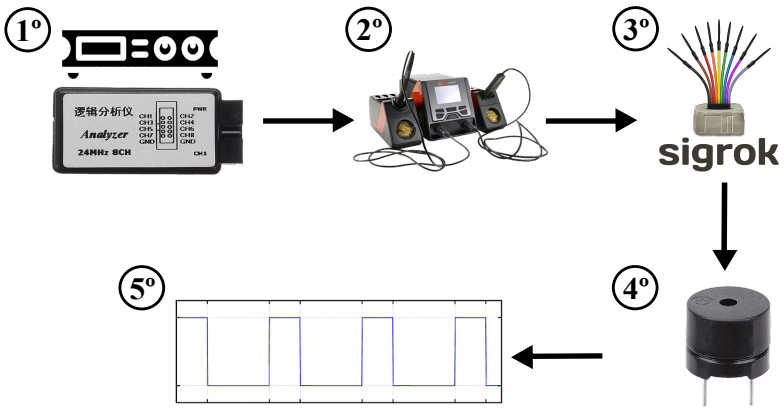


Figure 5.28 – *Steps for PWM signal analysis*

I was lucky, as just a couple of weeks before my colleague A. Cantudo had worked out a script to make a buzzer sound as well. But in his case, he had to write the **script in Python**. Even so, we looked into the library he used to communicate with the GPIO pins, the RPI.GPIO library, and found that the source code he used was written in C. So I downloaded that code and set out to use it to replicate the initial script.

We **investigated the source code of the library** and managed to duplicate the initial script as we can see in the figure 5.29. With this program, we were able to create an executable called makesound to which a string is passed as a parameter. If this string **corresponds to "success"**, the buzzer will make a high-pitched sound indicating success, while if **it does not correspond to this string**, the buzzer will make a low-pitched sound indicating an error. In addition to this, we **created a Makefile** to be able to compile and create the executable, according to the directory structure studied during the degree:

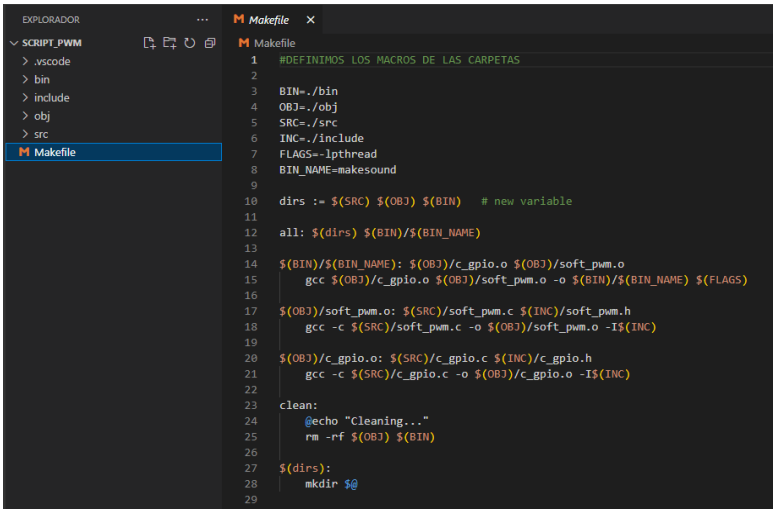


Figure 5.29 – *Displaying the Makefile and directory structure*

After this, and as we can see in line 21 of listing 3.1, our initial script also made use of two other executables, which were **in charge of sending the bitmap to the FPGA** and returning a 0 if the operation was successful. The clocks5.bin executable worked without problems, but the fpgaprog did not as it also used the WiringPi library. So, we managed again to add the library we had been able to fix the first script with, to fix this new problem. With this solved, we set out to modify the initial script with the

following result:

```c
int main(int argc, char *argv[]) {
    struct timespec remaining, request = {SECS_TO_SLEEP, NSEC_TO_SLEEP};
    char *state = "success";
    int freqs[2][3] = {
        {3000, 2000, 1000},
        {10000, 5000, 15000}
    };
    int selected_freq = 0;
    setup(); // inicializacion y pedir permisos para acceder a las direcciones de memoria de los pines
    setup_gpio(13, 0, 0); // Poner el pin 13 como salida con resistencia de pull up
    pwm_set_duty_cycle(13, 0.5); // Duty cycle del pin, por defecto a la mitad

    if (argc != 1) {
        state = argv[1];
    }

    if(strcmp(state, "success")) {
        selected_freq = 1;
    }

    pwm_start(13); // Empieza el pwm
    for (int i = 0; i < 3; i++) {
        pwm_set_frequency(13, freqs[selected_freq][i]); // Frecuencia
        nanosleep(&request, &remaining);
    }

    return 0;
}
```

Figure 5.30 – *Showing the buzzer script with new library*

Once the bugs have been fixed and the new script for the new kernel has been created, we can move on to **analysing and comparing the signals generated by the PWM**. But first, let's proceed to explain in a little more detail what this method consists of. Pulse width modulation [36] of a signal is a method by which we can control the amount of power that is sent to a load. This is achieved by periodically switching the output between high and low voltage levels. In addition, as shown in figure 5.31, we can see that there are three variables when analysing a PWM signal. Firstly we have **t** = the time in which the function is positive known as pulse width, secondly **T** = the period of the function, and lastly we have the **duty cycle**, which is defined as the amount of time the output is high in relation to the period of the signal, this being a number from 0 to 1. The higher the duty cycle, the higher the power and vice versa.

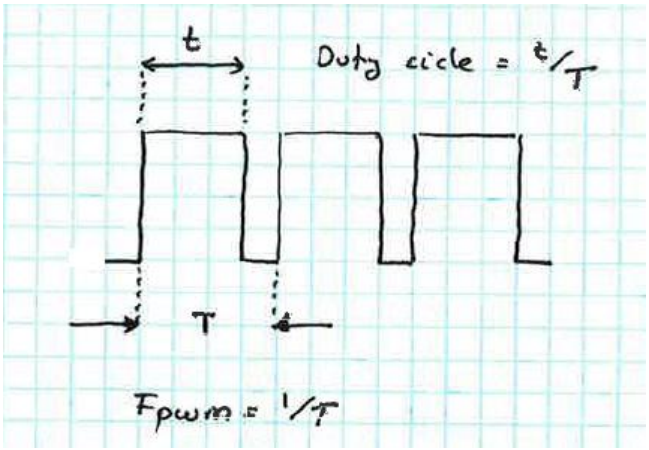


Figure 5.31 – *Theory of PWM signals*

Note that in our script, we use the PWM method to make the buzzer sound. So, now that we have a better understanding of what this method consists of, let's proceed to analyse and **compare the signals generated by the old script and the one we have updated**. We will be especially interested in observing if **the duty cycle that we have imposed on it is fulfilled**; in the old script we don't know what it is, we have assumed that it is the default duty cycle of the WiringPi library, while in the updated script we have set a duty cycle of 50%.

As I mentioned in section 5.2.3, the first thing we did was to solder 2 pins, **one on DGND and the**

other on the Buzzer pin, to the board of our limiter. This will allow us to make use of the signal analyser. Once soldered, we connect the device to our computer and run the pulse view program.

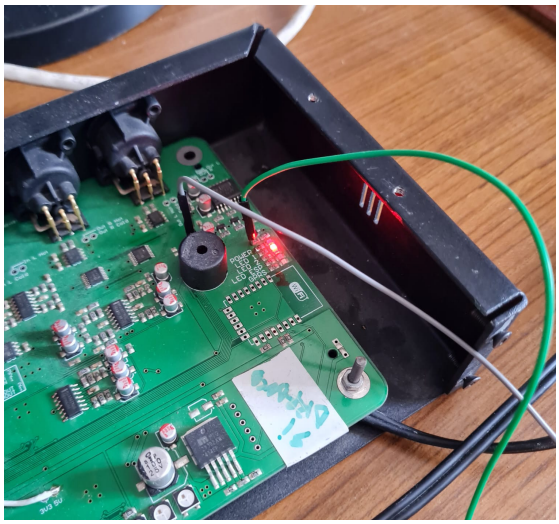


Figure 5.32 – *Signal analyser pin connections*

Once we have everything ready, we will click on capture the signal in the program and in the background we will execute the buzzer script. First, in figure 5.33, we show the result of the signal in the old script. As we can see, **this signal clearly does not correspond to a 50% duty cycle**, as the proportions in which it is on and off differ.But on the other hand, in figure 5.34, which shows the signal collected using the updated script, we can see that **it presents exactly a 50% duty load**. This is because the on and off ratios are identical, making a totally symmetrical square signal. In conclusion, we have been able to create a script that makes use of the PWM method, and that complies perfectly with the duty cycle that we indicate to it. We can therefore say that we have updated and improved the initial script.

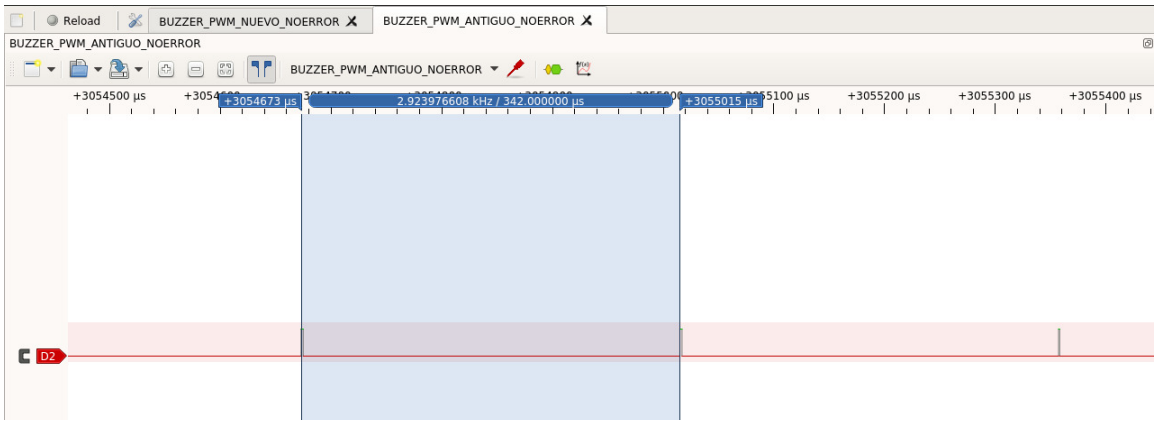


Figure 5.33 – *PWM signal from old buzzer script*

On the other hand, regarding the **NeoPixel LEDs test**, I managed to find the following **driver written in C** on the internet. My job from here was to make the documentation of the program, and modify some of the parameters to test each of the LEDs separately or together, as well as passing by parameter the colour with which I wanted them to be seen. In addition to this and as we did with the previous script, I **created a Makefile** with the directory structure mentioned above.

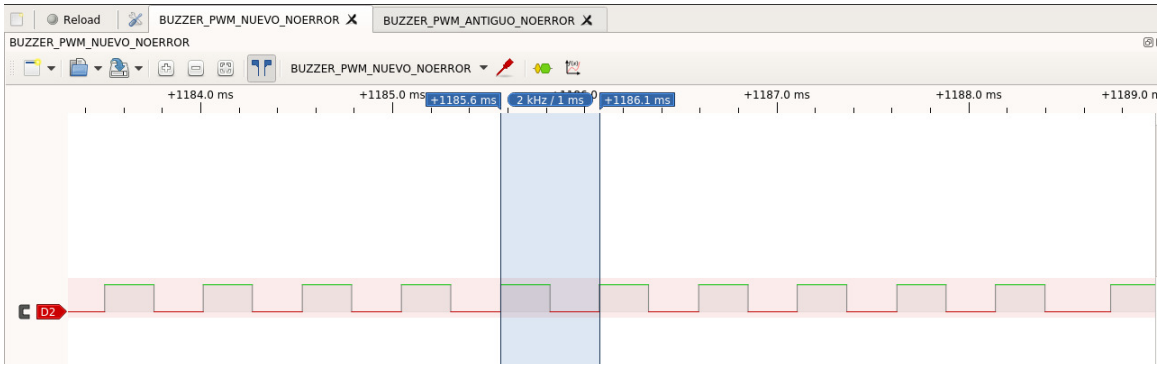


Figure 5.34 – *PWM signal from new buzzer script*

One of the tasks in relation to this test was to add it to the Makefile that A. Ruiz created where the rest of the tests for the other hardware devices were stored. This could have been done without any problem, because even if my test was written in C, I would have only **had to add the gcc compiler in the Makefile variables** and use it to compile. But when I was about to perform that task, I realised that all the tests A. Ruiz had done used the WiringPi library, and therefore, they **would have to be updated** with the fix we did or with another library.

I told the professor what had happened, and I told him that it was a good idea to **move all the tests to C**, as this was a better language for communicating with hardware devices, as it has more documentation and functionalities for this. My teacher agreed with me but told me that it would be a job for later, as I didn't have time to get involved in the task.

5.2.4 Safety measures for sound limiters

In the following, we will explain the procedures we have followed to **implement security measures** in our product. But before moving on to the demonstration, I would like to point out that the last two points in table 3.3, those related to data traffic, **have not been carried out**. This is due to a problem with time, as I had a few more weeks to complete the main objective nº6, which consisted of updating and applying improvements to the API. On the other hand, measures such as only leaving executable on the system will be discussed in the next chapter.

5.2.4.1 Public-private key generation via SSH with ed25519

First of all, we will proceed to **create different users in the limiter**, to be used depending on the person who is going to use it. These users will be **one for the clients** (without root permissions and with only what is necessary to carry out the necessary functions), and **another for the limiter administrators or technicians** (with root permissions and with full power and control of the limiters).

```
# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL
raulrguez    ALL=(ALL:ALL) ALL

# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) ALL
```

(a) *Granting sudo privileges*

```
root@DietPi:~# adduser raulrguez
Adding user `raulrguez' ...
Adding new group `raulrguez' (1001) ...
Adding new user `raulrguez' (1001) with group `raulrguez' ...
Creating home directory `/home/raulrguez' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for raulrguez
Enter the new value, or press ENTER for the default
        Full Name []: Raul Rodriguez Perez
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] y
```

(b) *Creating a new user*

Figure 5.35 – *Adding users for clients*

Once this is done, I am going to use my windows computer with Ubuntu 20.04 subsystem to **create my public and private SSH keys**. To do this, I will simply use the ssh-keygen command, adding the **encryption algorithm option ed25519**. After this we will be asked for the name of the file, and a passphrase, which we will add to prevent anyone using my computer to connect through my SSH account to any remote device.

```
raulrguez@DESKTOP-11MO05N:~$ ssh-keygen -t ed25519
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/raulrguez/.ssh/id_ed25519):
/home/raulrguez/.ssh/id_ed25519 already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/raulrguez/.ssh/id_ed25519
Your public key has been saved in /home/raulrguez/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:a9+t0CoWOPxhTkBc/EPQnI0ikxYWxehmq9mOWTTquu0 raulrguez@DESKTOP-11MO05N
The key's randomart image is:
+--[ED25519 256]--+
|     .+X++ +     |
|     .O + * .    |
|     + o +       |
|      =   o      |
|     +o+S  .     |
|     o=.=. .     |
|    .+.*oo. .    |
|   oo+..=. + .   |
|  o+E... .o o..  |
+----[SHA256]-----+
raulrguez@DESKTOP-11MO05N:~$
```

Figure 5.36 – *Encrypting SSH public private keys*

Once the keys have been generated, we simply need to add our public key to the **authorised-keys file on the remote device** we want to connect to. This can be done manually by copying and pasting, or through the scp command. Once this is done, we will be able to log in to our limiter without typing a password.

5.2.4.2 Built-in our modules into the new kernel

5

As mentioned above, kernel modules **can be added to the system as modules or included directly under the kernel**. This could be used as a security measure against reverse engineering, as they would have no way of knowing which modules we have included, and therefore, which hardware devices our product has.

```
[*]   SPI sound devices (NEW)  ----
[*]   USB sound devices (NEW)  --->
<M>   ALSA for SoC audio support  --->
< >   Xen para-virtualized sound frontend driver (NEW)
< >   Virtio sound driver (NEW)
```

(a) *Alsa included as a module*

```
[*]   SPI sound devices (NEW)  ----
[*]   USB sound devices (NEW)  --->
<*>   ALSA for SoC audio support  --->
< >   Xen para-virtualized sound frontend driver (NEW)
< >   Virtio sound driver (NEW)
```

(b) *Alsa included internally in the kernel*

Figure 5.37 – *How to include modules to the kernel with nconfig*

To do this, all you need to do when using **the nconfig tool** to add the system modules to the kernel is to **set the <*> option instead of <m>**. This will cause what we have commented in the first paragraph, once the kernel is compiled there will be no trace of the module that we include in the /lib/modules directory, but the system will continue to use that module since it is included internally in the kernel and is added when it does so.

5.2.5 Software system monitoring and deploy

Before testing the tool, we need to configure the system to be able to use it. First of all, **we must have a host machine**, which in this case will be my laptop, with which we will manage and **administer the node machines**, which in this case will be the limiters. In all the mentioned machines, the SSH communication

protocol must be activated, and python must be installed. But it will only be necessary to have the tool installed on the host machine.

With the prerequisites in place, we proceed to start configuring our tool. First of all, we are going to **establish communication between the node machines and the host via public-private SSH keys**. This will allow us to connect securely and without the need to set a password all the time to perform any of the tasks. Therefore, we will repeat the steps in section 5.2.4.1 for each of the limiters.

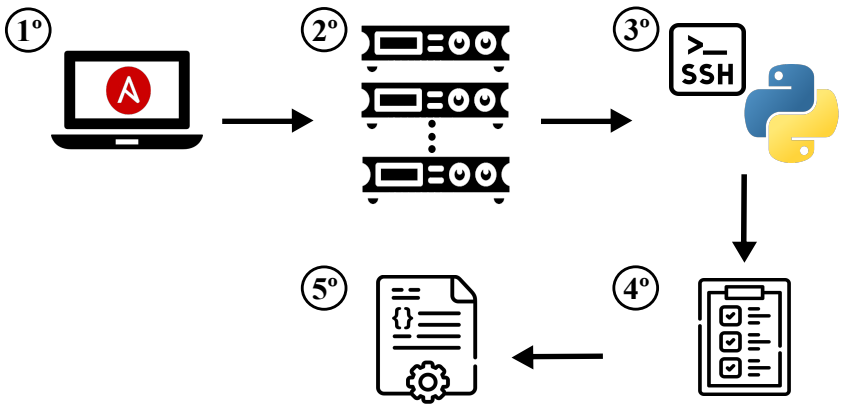


Figure 5.38 – *Steps to configure the system with Ansible*

With the SSH keys set up, let's **configure the Ansible inventory file**. This file contains an organised collection of the node machines we are going to manage. This file also **allows you to conveniently organise your nodes using patterns**. The patterns are like labels where we can organise the machines either by groups, alone, or all at once. At first, all the machines belong to the group whose pattern is all. From here we **can manage them in different groups according to the theme** we see fit, for example; according to their operating system, according to the services they offer, according to the state of the machine, etc.

```
raulrguez@pop-os:~/Escritorio/TFG_2122$ cat ejemplo_inv.yml
all:
  children:
    grupo1:
      hosts:
        host1
        host2
    grupo2:
      hosts:
        host3
        host4
raulrguez@pop-os:~/Escritorio/TFG_2122$
```

Figure 5.39 – *Ansible inventory example*

Patterns are mainly used with the intention of **selecting the group of nodes to which we want to perform a particular task**. Therefore, when using ad-hoc commands or playbooks, we must specify to Ansible which part of the inventory, that is, which nodes we want it to perform that action. According to figure 3.2, we could use the following patterns:

- all → todas las máquinas del inventario
- host1 → una máquina en concreto
- host1:host2 ó host1,host2 → varias máquinas sueltas, que pueden no estar agrupadas
- grupo1 → un grupo determinado de máquinas
- grupo1:grupo2 → varios grupos

- grupo1:!grupo2 → todas las máquinas del grupo 1 y que no estén en el grupo 2

In addition, the last concept we need to know about inventories is that we **can define variables** to add relevant information to the nodes. We can specify attributes such as the name, the IP, a user or password, etc. Also note that, with the installation of Ansible, a global configuration file /etc/ansible/ansible.cfg and a global inventory file /etc/ansible/hosts are created. These files will be the default ones to complete, although it is possible to tell Ansible to use other configuration and inventory files locally.

Ad-hoc commands allow us to run a task on a node or a set of nodes in our inventory. The advantage of these commands is that they make Ansible easier to use, but unlike playbooks, **they are not reusable**. Also, these commands tend to perform simple tasks, so some of them don't make much sense to store them in a separate playbook for each one. All of these commands follow the pattern:

$ ansible [pattern] -m [module] -a [module options]

From left to right, **the pattern** refers to the machine or machines where the command will be executed as I mentioned before. **The module** is a library where you can find functionalities to perform certain tasks. You can consult the different types of modules in the Ansible documentation. Finally, if we review the proposed documentation, each of these modules has certain options to distinguish the functionalities. Some examples of these commands are:

```
# Hacer ping a todas las máquinas que tenemos en el inventario (-i
 -> selec el archivo de inventario)
$ ansible all --key-file ~/.ssh/ansible -i inventory -m ping

Si añadimos al repositorio local el archivo ansible.cfg con la
 siguiente configuración podremos acortar el comando al siguiente:

--------ansible.cfg:---------------------
    [defaults]
    inventory = inventory
    private_key_file = ~/.ssh/ansible
------------------------------------------

# Comando acortado
$ ansible all -m ping

# Listar todos los host de nuestro inventario
$ ansible all --list-hosts

# Muestra todo tipo de info acerca del servidor
$ ansible all -m gather_facts --limit "ip_adress"

$ ansible all -m apt -a update_cache=true --become --ask-become-pass
apt module:  allows us to work with apt package on a debian-based
 system
-a: allows us to use an argument to that module (update_cache=true
 == sudo apt update)
--become:  dar privilegios de sudo
--ask-become-pass:  permite poner la password del sudo de nuestro \
 acrshort{PC} (si la contra del sudo de las máquinas es diferente
 falla)
```

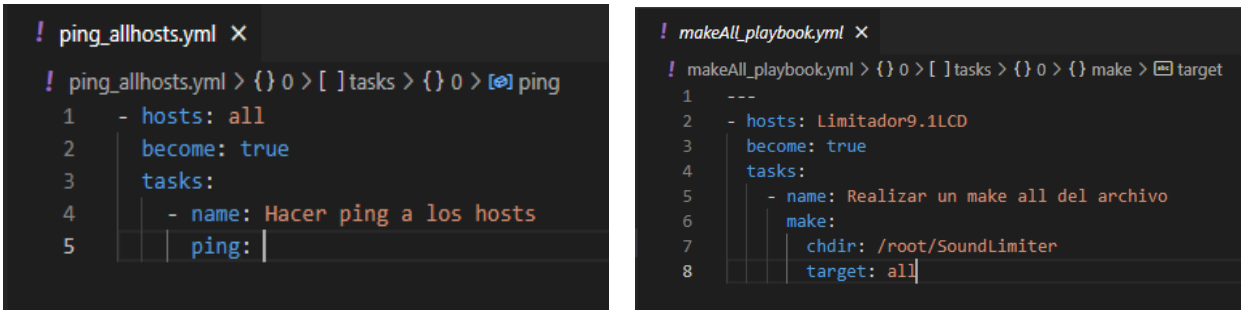
```
# Instalar un paquete en todos nuestros servidores (en este caso
  instalar vim-nox o tmux)
$ ansible all -m apt -a name=vim-nox --become --ask-become-pass
$ ansible all -m apt -a name=tmux --become --ask-become-pass
```

Listado 5.5 – *Steps to compile RTC module*

Finally, the last concept we need to know about to finish with the Ansible configuration is the playbook. A playbook is **a list of tasks that are applied on a given set of node machines**. Basically, we can define it as a list of ad-hoc commands that act on the different remote machines that we manage and that have the special feature of being reusable, so to speak, **they are like scripts**. To create our playbook, the first thing to do is to define which hosts are involved, and which user is going to perform each task. Once these two fields have been defined, we simply have to add the task, which we can give, before adding the ad-hoc command, an explanatory name. We will use this name **as a guide to know which command has been executed** when we see the output in the terminal.

In this example of a playbook that performs a **ping to all hosts**, we can see each of the fields that we discussed a moment ago. First of all, we want to execute the command to all the remote machines ("all"), after this we put the option to have sudo privileges so that we don't have any permissions problem when executing the ad-hoc. Finally, we write a descriptive name and put the module that performs the ping action. To run this playbook, type the following in the terminal:

$ ansible-playbook -i inventory ping_allhosts.yml



(a) *Ping all hosts* **(b)** *Make all hosts*

Figure 5.40 – *Ansible playbook example*

Chapter 6

System verification and testing

In this section, the evaluation of the requirements outlined in chapter two will be carried out, and a brief summary of the final state of the product will be given.

6.1 Evaluation of satisfaction of requirements

The functional requirements are then analysed on the basis of the process for fulfilling their task. To do this, it is studied whether the resulting product can satisfy each of the functional requirements of the project:

Requirement	RF 1.- The system shall be able to allow only users who are correctly identified and verified to log in to the system.
Section	Safety measures
Analysis	Requirement compliant with the incorporation of public-private keys via SSH, leaving access only to technicians or customers of the product.
Evaluation	Validated

Table 6.1 – *FR 1. Satisfaction Evaluation*

Requirement	RF 2.- Once the system has started, a high-pitched sound will be emitted through the buzzer if the communication with the FPGA has been successful.
Section	Update hardware test programs
Analysis	Requirement incorporated with the update and testing of the new buzzer script.
Evaluation	Validated

Table 6.2 – *FR 2. Satisfaction Evaluation*

Requirement	RF 3.- Once the system has started, a low sound will be emitted through the buzzer if the communication with the FPGA has failed.
Section	Update hardware test programs
Analysis	Requirement incorporated with the update and testing of the new buzzer script.
Evaluation	Validated

Table 6.3 – *FR 3. Satisfaction Evaluation*

Requirement	RF 4.- The system must include a update mechanism to allow detection of the system by other equipment on the same network.
Section	Update sending data
Analysis	Due to the lack of progress in the section related to this requirement, its validity has not been tested.
Evaluation	Invalidated

Table 6.4 – *FR 4. Satisfaction Evaluation*

Requirement	RF 5.- The system must provide a update mechanism for sending data to the production server and to desktop software.
Section	Update sending data
Analysis	Due to the lack of progress in the section related to this requirement, its validity has not been tested.
Evaluation	Invalidated

Table 6.5 – *FR 5. Satisfaction Evaluation*

Requirement	RF 6.- The system shall be capable of communicating with and testing the LEDs incorporated in the board
Section	Update hardware test programs
Analysis	Due to the inclusion and documentation of the NeoPixel LED driver, the system is able to communicate and interact with the LEDs.
Evaluation	Validated

Table 6.6 – *FR 6. Satisfaction Evaluation*

Requirement	RF 7.- An automation process for Kernel crops-compilation shall be developed.
Section	Update Linux Kernel
Analysis	Requirement validated by the creation of the dockerfile to automate kernel compilation.
Evaluation	Validated

Table 6.7 – *FR 7. Satisfaction Evaluation*

Requirement	RF 8.- The Kernel modules should continue to function in the same way after the kernel update.
Section	Update Linux Kernel
Analysis	After the kernel update, all previously added modules continue to work without any problems.
Evaluation	Validated

Table 6.8 – *FR 8. Satisfaction Evaluation*

6.2 Test final product

In conclusion, I would like to briefly show the final state of the product. The SL4 sound limiter, after the realisation of the project, has got a new kernel update. In addition, it has new and improved scripts that provide new functionalities such as the buzzer beep on start-up or the NeoPixel LED tests. Apart from that, it finally has both a functional operating system and a customisable desktop to modify and make it attractive to customers. So to finish I wanted to show the storage space of the limiter, together with a screenshot with relevant information thanks to the neofetch tool:

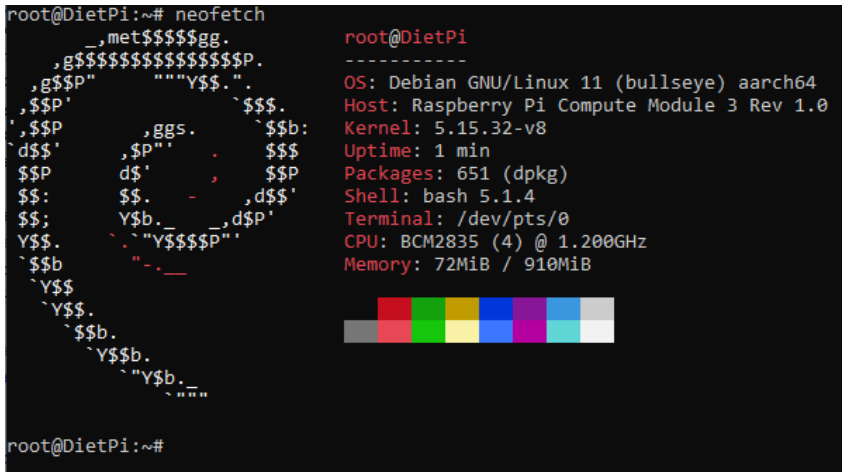


Figure 6.1 – *Relevant information of the system*

```
root@DietPi:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       3.5G  3.0G  362M  90% /
devtmpfs        422M     0  422M   0% /dev
tmpfs           456M     0  456M   0% /dev/shm
tmpfs           183M  7.4M  175M   5% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           1.0G     0  1.0G   0% /tmp
/dev/mmcblk0p1  127M   55M   72M  43% /boot
tmpfs            91M   36K   91M   1% /run/user/0
root@DietPi:~#
```

(a) *Storage space*

```
root@DietPi:~# free -m
               total        used        free      shared  buff/cache   available
Mem:             910          60         768           7          80         790
Swap:           1088           0        1088
root@DietPi:~#
```

(b) *Use of memory*

Figure 6.2 – *Final storage space and use of memory*

Chapter 7

Conclusions, future work and lessons learned

7.1 Conclusions

At the end of the project, we can conclude that the results obtained have been mostly satisfactory, as the vast majority of the objectives and requirements set have been successfully met. In addition, I have put into practice not only the knowledge acquired during the Degree in Computer Engineering, but this project has required me to acquire knowledge by handling new tools, learning new methodologies or researching in other areas such as acoustics and electronics. And as if this were not enough, the work has been carried out in a cooperative environment where my communication and teamwork skills have been put to the test. Without the good atmosphere among the team, and everyone's ability to learn from each other, this work would have meant even more sacrifice and effort than it already does.

It is a pity that due to the uncertain nature of the project, it could not be completed in its entirety. Even so, the resulting product is now fully operational, having not only the same functionality as before the kernel update, but improved with the inclusion of new elements such as security measures. Also, on a personal level, I am happy to know that in the near future, the person who will continue behind me, will not have to go through the hard road I went through. All of this is thanks to the rigorous documentation of everything that has been done, not only manuals or user guides, but also explanations of the decisions that have been made and what they entail.

As a final comment, I would like to point out that despite the fact that this project has been an exhaustingly hard road, where I have had very low and frustrating points where I felt lost, I have managed to finish it in a more than satisfactory way, and feeling very satisfied with my effort and the new skills I have learned that I am sure will help me in my future work.

7.2 Future work

As mentioned above, there are some requirements or objectives that, due to lack of time, have not been implemented in the end. These are not many, but I consider them important for the future commercial success of our product. In the same way, there is still work to be done to finalise the development process of the sound limiter, which can be divided into software and hardware implementations. In short, the work that remains to be done in the future is:

- Hardware:
  - To realise and implement a final design for the Sound Limiter motherboard.
  - To carry out the verification process for this new PCB design.
  - To add kernel modules for LCD devices and other new peripherals to be added to the limiter in future designs.
  - To create and implement the design of the enclosure housing the sound limiter.
  - Incorporate and test the second microphone that the limiter prototype must have.
- Software:
  - To customise the operating system to make it attractive and functional for the customer.
  - Update and refactor the code that captures and sends the data, as this work had been done in haste and there are serious bugs in it.
  - Carry out the implementation of an event dispatcher that informs customers of events of interest to them, such as their licence expiry date.
  - To develop tests and create functionality for the LCD of the limiter.
- Work not performed:
  - To make an update to the sound limiter API
  - To support the API for new features to be created in the sound limiter configuration application
  - To implement the security measures studied in the data traffic
  - Test the Ansible configuration on the final communication system.

7.3 Lessons learned

The mere fact of being part of a project of such dimensions and characteristics, puts me in the situation of pushing myself beyond my limits and knowledge, and trying to do my best. That is why I wanted to conclude this project with a brief list of the most enriching lessons I have learned:

- To learn more about the Linux kernel, to the point of being able to handle it with ease and to be able to perform tasks such as updating and compiling it without too much trouble.
- Learn and use cutting-edge engineering tools such as Ansible
- Expand my knowledge to new engineering fields such as electronics or acoustics.
- Improve and develop my intrapersonal and interpersonal skills by having carried out part of the project activities as a team.
- Carry out research and documentation of the steps taken by previous students, without having any clues or help to do so.

7

Bibliography

[1] Altexsoft. Serial peripheral interface. https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation/.

[2] Ansible. Inventory. https://docs.ansible.com/ansible/latest/network/getting_started/basic_concepts.html#inventory.

[3] Ansible. Managed nodes. https://docs.ansible.com/ansible/latest/network/getting_started/basic_concepts.html#managed-nodes.

[4] Ansible. Playbooks. https://docs.ansible.com/ansible/latest/network/getting_started/basic_concepts.html#playbooks.

[5] ARM, A. L. Images of the arch linux operating system for raspberry pi 3. https://archlinuxarm.org/platforms/armv8/broadcom/raspberry-pi-3.

[6] Baesystems. What are single-board computers? https://www.baesystems.com/en-us/definition/what-are-single-board-computers.

[7] basics, L. A. Ansible. https://www.redhat.com/en/topics/automation/learning-ansible-tutorial.

[8] Beagleboard. Documentation of beagleboard. https://beagleboard.org/getting-started.

[9] Becerra, A. R. Limitador de sonido para locales de música. Bachelor Thesis, University of Granada, Granada, September 2020-2021.

[10] Debian. Images of the debian operating system for raspberry pi. https://wiki.debian.org/RaspberryPi#Raspberry_Pi_OS_.28formerly_Raspbian.29_and_Debian.

[11] DietPi. Dietpi official website. https://dietpi.com/.

[12] Emteria. What is the raspberry pi compute module? https://emteria.com/learn/rpi4-vs-rpi4-compute-module.

[13] for hire, B. Sound limiters: The guide for musicians, venues and clients. https://www.bandsforhire.net/blog/sound-limiter-guide.

[14] Garlan, D., and Shaw, M. An introduction to software architecture. *World Scientific Publishing Company, I* (1994), 42.

[15] Geeks. Linux system call in detail. https://www.geeksforgeeks.org/linux-system-call-in-detail/.

[16] github, L. K. Kernel modules. https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html.

[17] HP. What are computer drivers? https://www.hp.com/us-en/shop/tech-takes/what-are-computer-drivers.

[18] I, R. Raspberry pi processors. https://www.raspberrypi.com/documentation/computers/processors.html#bcm2835.

[19] IBM. Anatomy of the linux kernel. https://developer.ibm.com/articles/l-linux-kernel/.

[20] IC, A. Pulse width modulation: What is it and how does it work? https://www.analogictips.com/pulse-width-modulation-pwm/.

[21] Linux, A. Images of the alpine linux operating system. https://alpinelinux.org/downloads/.

[22] Linux, T. C. Images of the tiny core linux operating system. http://tinycorelinux.net/downloads.html.

[23] Linuxhint. Best lightweight desktop environment for raspberry pi. https://linuxhint.com/best-desktop-environments-raspberry-pi-operating-system/.

[24] LXDE. Lxde desktop environment official website. http://www.lxde.org/.

[25] LXQT. Lxqt desktop environment official website. https://www.xfce.org/.

[26] Makeusefor. The 8 best lightweight operating systems for raspberry pi. https://www.makeuseof.com/tag/lightweight-operating-systems-raspberry-pi/.

[27] Marsboard. Documentation of marsboard. https://www.marsboard.com/new_marsboard_a20_download.html.

[28] Medina, D. R. Teleconfiguration system based on electron and nodejs. Bachelor Thesis, University of Granada, Granada, September 2020-2021.

[29] Nibusiness. Common cyber security measures. https://www.nibusinessinfo.co.uk/content/common-cyber-security-measures.

[30] Pi, R. Documentation raspberry pi 3 model b. https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#schematics-and-mechanical-drawings.

[31] Pi, R. Documentation raspberry pi compute module 3. https://www.raspberrypi.com/documentation/computers/compute-module.html#content.

[32] Pi, R. Images of the raspberry pi operating system. https://www.raspberrypi.com/software/operating-systems/.

[33] ProfesionalReview. Raspberry pi. https://www.profesionalreview.com/2021/07/18/que-es-raspberry-pi/.

[34] Redhat. What is the linux kernel? https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel.

[35] RPI. Rpi-source github. https://github.com/RPi-Distro/rpi-source.

[36] ScienceDirect. Pulse width modulation. https://www.sciencedirect.com/topics/engineering/pulse-width-modulation.

[37] Techopedia. Digital switch. https://www.techopedia.com/definition/14121/digital-switch.

[38] TechRepublic. Ansible overtakes chef and puppet as the top cloud configuration management tool. https://www.techrepublic.com/article/ansible-overtakes-chef-and-puppet-as-the-top-cloud-configuration-management-tool/.

[39] TECHTARGET. What is a motherboard? https://www.techtarget.com/whatis/definition/motherboard.

[40] UVA, E. El lenguaje c++. https://www2.eii.uva.es/fund_inf/cpp/temas/1_introduccion/introduccion.html.

[41] WIKIPEDIA. Application programming interfaces. https://en.wikipedia.org/wiki/Serial_Peripheral_Interface.

[42] WIKIPEDIA. Telecommunications technologies. https://en.wikipedia.org/wiki/Telecommunications.

[43] XFCE. Xfce desktop environment official website. https://lxqt-project.org/.