This Bacherlor's Thesis aims to present all the steps involved in the process of developing a building and automation system for an IoT device based on a Raspberry Pi, which will be in charge of making sure that clubs and entertainment venues comply with the regulations related to acoustic levels.

The project has been divided in three parts, the device tests, the build system to have a custom Linux Kernel and image, and the automation system built to have idempotent deployments of the software, firmware and configuration necessary for each device.

Gonzalo Abril Paniza is a Computer Scientist from Granada, Spain. He is the author of this project.

Andrés María Roldán Aranda is the academic head of the project and the student's tutor. He is professor in the Department of Electronics and Computer Technologies.

BACHEERLOR'S THESIS

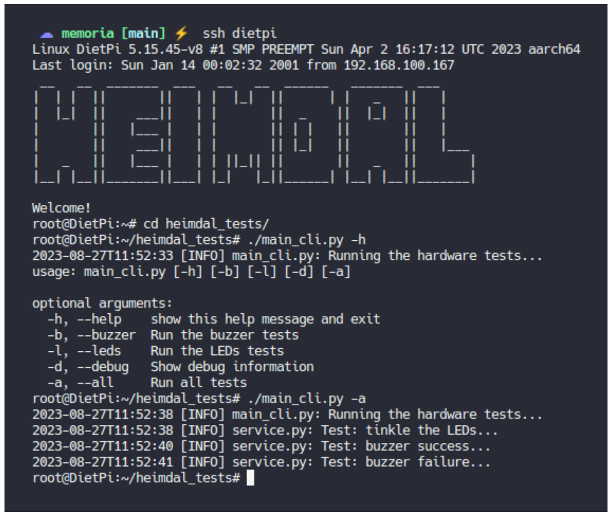
Building, Provisioning, and Deployment of IoT Acoustic Devices

Gonzalo Abril Paniza

Degree in Computer Engineering

UNIVERSITY OF GRANADA

Degree in Computer Engineering



```
memoria [main] ⚡  ssh dietpi
Linux DietPi 5.15.45-v8 #1 SMP PREEMPT Sun Apr 2 16:17:12 UTC 2023 aarch64
Last login: Sun Jan 14 00:02:32 2001 from 192.168.100.167

Welcome!
root@DietPi:~# cd heimdal_tests/
root@DietPi:~/heimdal_tests# ./main_cli.py -h
2023-08-27T11:52:33 [INFO] main_cli.py: Running the hardware tests...
usage: main_cli.py [-h] [-b] [-l] [-d] [-a]

optional arguments:
  -h, --help    show this help message and exit
  -b, --buzzer  Run the buzzer tests
  -l, --leds    Run the LEDs tests
  -d, --debug   Show debug information
  -a, --all     Run all tests
root@DietPi:~/heimdal_tests# ./main_cli.py -a
2023-08-27T11:52:38 [INFO] main_cli.py: Running the hardware tests...
2023-08-27T11:52:38 [INFO] service.py: Test: tinkle the LEDs...
2023-08-27T11:52:40 [INFO] service.py: Test: buzzer success...
2023-08-27T11:52:41 [INFO] service.py: Test: buzzer failure...
root@DietPi:~/heimdal_tests# 
```

Building, Provisioning, and Deployment of Iot Acoustic Devices

Gonzalo Abril Paniza

2022/2023

Tutor: Andrés María Roldán Aranda

Impreso en Granada, Septiembre 2023.

“Building, Provisioning, and Deployment of IoT Acoustic Devices.”

Degree in Computer Engineering

Bachelor’s Thesis

“Building, Provisioning, and Deployment of IoT Acoustic Devices.”

AUTHOR:

Gonzalo Abril Paniza

TUTOR:

Prof. Andrés María Roldán Aranda

DEPARTAMENT:

Electronics and Computer 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 Grado de D. Gonzalo Abril Paniza,

Informa:

Que el presente trabajo, titulado:

Building, Provisioning, and Deployment 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 5 de Septiembre de 2023

Andrés Roldán

Fdo. Prof. Andrés María Roldán Aranda

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

Granada, a 5 de Septiembre de 2022

[illegible] [illegible]

Fdo. Gonzalo Abril Paniza

Fdo. Prof. Andrés María Roldán Aranda

“Compilación, despliegue y aprovisionamiento de dispositivos IoT acústicos.”

Gonzalo Abril Paniza

PALABRAS CLAVE: *dispositivos IoT, automatización, despliegue, aprovisionamiento, ansible, python, docker, raspberry pi.*

RESUMEN:

Tras el desarrollo de un limitador de sonido Internet of Things (IoT) por el equipo de GranaSAT, surge la necesidad de de automatizar las fases de compilación, despliegue y aprovisionamiento del mismo para facilitar el proceso de desarrollo así como el despliegue de nuevas versiones en dispositivos. Como solución, en este proyecto se han desarrollado una serie de repositorios enfocados a cada una de las fases mencionadas, que se han integrado en un flujo de trabajo que permite la compilación de una imagen de Linux personalizada para el limitador de sonido, así como su despliegue y aprovisionamiento en un dispositivo IoT.

“Building, Provisioning, and Deployment of IoT Acoustic Devices.”

Gonzalo Abril Paniza

KEYWORDS: *IoT devices, automation, deployment, provisioning, ansible, python, docker, raspberry pi.*

ABSTRACT: After the development of an IoT sound limiter by the GranaSAT team, the need arises to automate the compilation, deployment and provisioning phases of the same to facilitate the development process as well as the deployment of new versions on devices. As a solution, in this project a series of repositories have been developed focused on each of the mentioned phases, which have been integrated into a workflow that allows the compilation of a custom Linux image for the sound limiter, as well as its deployment and provisioning on an IoT device.

Acknowledgments

To my mother, who has been there for me in my ups and downs, who has showed me the meaning of fighting, caring, and has inspired me to become a better man since I was young.

To my fiancé, who has supported me in my journey through college and has motivated me to become the best version of me I can be. I would not be the person I am without you.

To my brother, who has been a partner in the good times and a shoulder to lean on in the hard ones.

I would also like to thanks my tutor Andrés Roldán Aranda for giving me the opportunity to work on such an exciting project, this has been the best way to finish my years at college that I can think of, for that I am very grateful.

To all of you, thank you.

Contents

List of Figures

List of Tables

Glossary

A | **B** | **C** | **D** | **G** | **H** | **I** | **L** | **N** | **O** | **P** | **Q** | **R** | **S** | **X**

A

Ansible Ansible® is an open source, command-line IT automation software application written in Python. It can configure systems, deploy software, and orchestrate advanced workflows [23].

Ansible Role An Ansible role is a way to have related information, tasks and variables all in one place to help structuring an Ansible project [9, Chapter 6].

ARM ARM is a family of reduced instruction set computing (RISC) architectures for computer processors [2] .

B

BuildKit BuildKit is build backend used by Docker for converting source code to build artifacts in an efficient, expressive and repeatable manner [12] .

C

C C is a general-purpose, procedural computer programming language supporting structured programming, lexical variable scope, and recursion, with a static type system. By design, C provides constructs that map efficiently to typical machine instructions [22] .

chroot chroot is a command that allows to change the root directory of a process and its children to a new location [27] .

D

Desktop Environment A desktop is a set of tools and software that allow users to use their computer via a graphical interface [34] .

Device Tree A device tree is a flexible way to define the hardware components of a computer system. Usually, the device tree is loaded by the bootloader and passed to the kernel [31, Chapter 3] .

Device Tree Blob A DTB file is a binary representation of the Linux Device Tree obtained after compiling a Device Tree Source (DTS) file with the Device Tree Compiler (DTC) tool [31, Chapter 3] .

DevOps DevOps combines development and operations to increase the efficiency, speed, and security of software development and delivery compared to traditional processes. A more nimble software development lifecycle results in a competitive advantage for businesses and their customers [18].

DietPi DietPi is a lightweight Linux distribution based on Raspbian. It is prepared to be used in Single Board Computers [3] .

Docker Docker is a platform that allows users to develop, deploy, and run applications with containers, which is a a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. It is similar to a virtual machine, but it is more lightweight and portable, being the main difference between both that Docker virtualizes the operating system whereas virtual machines virtualize the hardware [17] .

Docker Engine An open source containerization technology for building and containerizing applications. Docker Engine acts as a client-server application with a server running a long-running daemon process called dockerd, APIs that specify interfaces for programs to talk to and instruct the Docker daemon, and a command line interface (CLI) client called docker[14].

Dockerfile A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image [15] .

G

Git Git is a free and open-source distributed version control system for tracking changes in source code during software development [10].

GitLab GitLab is a web-based DevOps lifecycle tool that provides a Git-repository manager providing wiki, issue-tracking and continuous integration/continuous deployment pipeline features, using an open-source license, developed by GitLab Inc. [19] .

GranaSAT *Electronics Aerospace Group.* An academic project from the UGR. This organization has an electronics laboratory where students from different degrees and education levels develop multidisciplinary projects pagegranasat .

H

Hypervisor A hypervisor is a virtual machine monitor that creates and runs virtual machines. It allocates resources on the host computer to support multiple virtual machines with their own operating systems and applications [32] .

I

IoT The Internet of Things is a system in which objects in the physical world could be connected to the Internet by sensors[30].

L

Linux Linux is an open-source kernel, but it is also used to name the operating system based on it.

Linux Distribution A Linux distribution is an operating system based on the Linux kernel with added software that provides additional functionality, such as a package manager or a desktop environment.

Loop Device A loop device is a block device that maps its data blocks not to a physical device such as a hard disk or optical disk drive, but to the blocks of a regular file in a filesystem or to another block device. This can be useful for example to provide a block device for a filesystem image stored in a file, so that it can be mounted with the mount command [4] .

N

NTP NTP is a protocol for synchronizing a set of network clocks using a set of distributed clients and servers [28] .

O

Overlay Directory structure copied on top of the root filesystem that might contain executables and libraries [31, Chapter 6] .

P

Python Python is an interpreted, high-level and general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant indentation [5] .

Q

QEMU QEMU is a free and open-source emulator and virtualizer. As machine emulator, it allows to run OSs and programs made for one architecture in another [29].

R

Raspberry Pi The Raspberry Pi is a series of small Single Board Computer computers developed Raspberry Pi Foundation[7].

Raspberry Pi Module 3 The Raspberry Pi Module 3 is a Raspberry Pi model developed by the Raspberry Pi Foundation which is intended for industrial application[6] .

Raspbian Raspbian is a Debian-based computer operating system for Raspberry Pi .

S

Single Board Computer A single-board computer is a complete computer built on a circuit board[25]. In our context, the SBC is the Raspberry Pi.

Sound Limiter A sound limiter is a device that is used to limit the volume of amplified music in a venue to a set level.

Sound Limiter 4 The Sound Limiter 4 is a Sound Limiter developed by the GranaSAT team .

SSH Secure Shell (SSH) is a cryptographic network protocol for operating network services securely over an unsecured network. The best known example application is for remote login to computer systems by users [24].

X

x86-64 x86-64 is the 64-bit version of the x86 instruction set used in modern processors [2] .

Acronyms

C | D | G | I | L | O | P | R | S | U

C

CLI Command Line Interface.

D

DHCP Dynamic Host Configuration Protocol.

DTB Device Tree Blob.

DTC Device Tree Compiler.

DTS Device Tree Source.

G

GPIO General Purpose Input/Output.

GUI Graphical User Interface.

I

IoT Internet of Things.

L

LCD Liquid Crystal Display.

O

OS operating system.

P

PCB Printed Circuit Board.

PWM Pulse Width Modulation.

R

RAM Random Access Memory.

S

SSH	Secure Shell.

U

UGR	University of Granada.
USB	Universal Serial Bus.

Chapter 1

Introduction

This Bachelor's Thesis is the final result of a long journey as a Computer Engineering student in the University of Granada. The last step, where the knowledge and skills obtained during the degree are put into practice. The present document aims to show how that abilities have been applied to the development of a real-world application, where not only the technical aspects are of importance, but also the capabilities to organize, manage and plan a project that will be carried on by other people in the future.

This document will show the different processes that have been carried out during the development of the project, from defining the requirements of the task, to design and implementation of the different repositories that make up the project, as well as the testing and validation of the application, showing the full cycle of the software development process.

The Project has been realized with collaboration of the academic project GranaSAT, an aerospace development group of the University of Granada (UGR), formed by students from different fields of Engineering, under the supervision of Professor Dr. Andrés María Roldán Aranda.

Figure 1.1 – *The GranaSAT logo.*

1.1. Motivation

The main reason for choosing this project was the opportunity to solve a real-world problem, using modern technologies and tools, and working in a middle-ground between low level and high level programming which allowed me to have a broader perspective of the rich world of Computer Engineering. It was also an important aspect to learn more about automation tools and the DevOps culture, which is becoming more and more relevant in the industry. The insight gained from this project has proven to be very valuable for my professional carrer, and I am sure it will continue to be so in the future.

1.2. Objectives of the Study

The main goal of this project was to build on the foundations made by Raúl Rodríguez Pérez and other students before him, who developed the Heimdal Kernel Module and made some adjustments to it in their respectives Bachelor's Theses, so that the Heimdal Sound Limiter could reach a state where deploying and testing new devices would be a simple and easy process, as well as setting up the stepstones to facilitate the labor of remote managing already deployed devices.

These objectives are summarized in the table 1.1.

Obj. Nº	**Description**
Obj. 1	Solve the existing technical challenges that difficulted setting up and using the Sound Limiter.
Obj. 2	Create Hardware Tests to facilitate verifying that new devices work as expected.
Obj. 3	Configure an idempotent build system for the Kernel Module.
Obj. 4	Set up an automation system to facilitate the deployment, provisioning and maintenance of devices.

Table 1.1 – *Objectives of this Bachelor Thesis.*

Apart from the aforementioned main objectives, the project also had the following secondary objectives:

- Create maintanable and scalable solutions, that can be easily adapted to new requirements.
- Use good programming practices to ensure the quality of the code and the maintainability of the project.
- Learn about the DevOps culture and the tools that are used in the industry.
- Acquire a better insight of how the Linux Kernel works and how to manage custom Kernel Modules.
- Learn about the Raspberry Pi and how to use Single Board Computer to build IoT devices.

Due to the requirements defined in the first meetings with the project supervisor Table 1.1, the project was divided into three main parts, having each one of them been developed in a different repository of the GranaSAT GitLab workspace:

- **Heimdal Tests**, to address the Obj. 2.
- **Image Builder**, to address the Obj. 3.
- **Heimdal Ansible**, to address the Obj. 4.

As for the Obj. 1, due to the miscellaneous nature of the changes that were necessary, they haven't been addressed in a specific repository, but rather in **Image Builder** and **Heimdal Ansible** at the same time. These challenges and solutions are explined in detail in Design and Implementation.

1.3. Context

As mentioned earlier, this project is based on the work made by other students in their respectives Bachelor's and Master Theses. Alejandro Ruiz Becerra reverse engineered a set of Sound Limiters for his Computer Engineering Bachelor's Thesis, and then developed the first version of the Heimdal Kernel Module. After this, Raúl Rodríguez Pérez carried on the work with the help of other undergraduate and postgraduate students, modifying the original Kernel Module to work with newer Kernel versions, as well as defining some of the basic items this project is based on, such as the Linux Distribution, the Desktop Environment, or the automation tools used to deploy the devices, more specifically, Ansible.

Apart from the work made on the software side, the project also has a hardware component that has undergone multiple iterations. The device which was mainly used while working on this project goes by the name of Sound Limiter 4. This specific device (Figure 1.3, Figure 1.4) has the following features:

- A custom Printed Circuit Board (PCB) which connects the different devices. A Raspberry Pi Module 3 (Figure 1.2) governs the device.
- Six balanced audio inputs.
- An HDMI output to connect the device to a monitor.
- A Universal Serial Bus (USB) port to connect the device to a computer.
- A 10/100 Mbps Ethernet connector to connect the device to a network.
- A LCD screen to display information about the device.
- A hardware clock that keeps track of the time with a higher precision than the Raspberry Pi system clock.

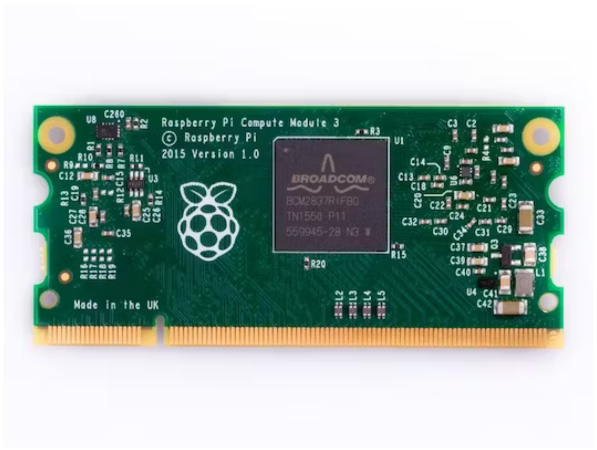


Figure 1.2 – *The Raspberry Pi Module 3.*

As a side note, it is important to mention that in some of the previous iterations of the project, version control software was not used in a proper way, which made it difficult to track the changes made to the code, as well as to know its state at the beginning. The source code of the Kernel Module that was used as the starting point for this project was an unfinished one, and it was not until long after that the correct one was delivered, which hindered the development of the project in the first weeks. To avoid this happening in the future, the project has relied on using Git as the version control software, and GitLab as the platform to host the repositories from the very first day.

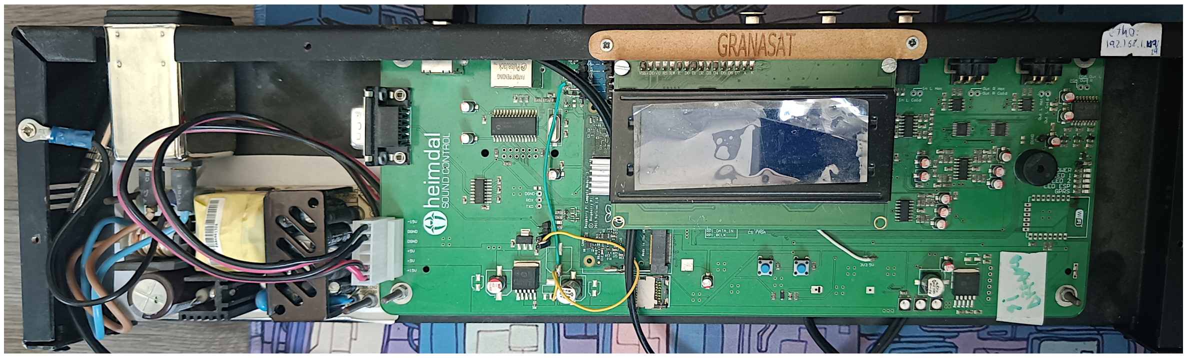


Figure 1.3 – *The Sound Limiter 4 device.*

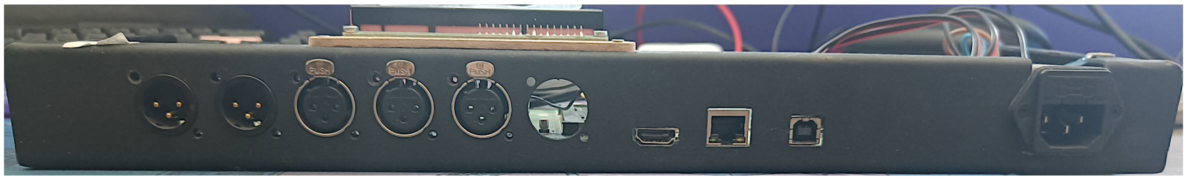


Figure 1.4 – *The Sound Limiter 4 device.*

1.4. Project Structure

The project is divided into six chapters:

1. **Introduction**: This chapter introduces the project, explaining the motivation behind it, the objectives that were set, and the context in which it was developed.
2. **Software Requirements Specification**: In this chapter, the requirements that were defined at the beginning of the project are described.
3. **Planning**: This chapter describes how the planning of the project was carried out, as well as the budget that was set for it.
4. **Design and Implementation**: This chapter explains the design decisions that were made during the development of the project, as well as the implementation details of the different repositories that were made.
5. **Validation and Testing**: This chapter explains how the different repositories were tested and validated according to the requirements that were defined in the Software Requirements Specification chapter.
6. **Conclusions and Future Work**: This chapter is a retrospective of the project, where the results obtained are analyzed, and the future work that could be done is discussed, as well as the lessons learned during the development of the project.

Chapter 2

Software Requirements Specification

These are the requirements that were defined in the firsts meetings with the project supervisor, and that were used as a base to define the scope of the project. Since the project was divided into three main parts, and each one of them was developed with different technologies in a different repository of the GranaSAT GitLab workspace (Table 1.2), the requirements were also divided into three different sections, one for each repository. The requirements of each are divided into three main categories: **Functional Requirements**, **Non-Functional Requirements** and **Information Requirements** (when applicable).

2.1. Heimdal Tests

2.1.1. Functional Requirements

FR1 The system will switch on and off all the front LEDs of the Sound Limiter.

FR2 The system will play a start up sound through the buzzers of the Sound Limiter.

FR3 The system will play a shutdown sound through the buzzers of the Sound Limiter.

2.1.2. Non-Functional Requirements

NFR1 The tests must be able to run on a Sound Limiter with a Raspberry Pi Module 3.

NFR2 The tests must have a simple and intuitive Graphical User Interface (GUI).

NFR3 It must be possible to run the tests from the command line.

2.2. Image Builder

2.2.1. Functional Requirements

FR1 The system will be able to compile a custom Kernel with the Heimdal Module for the Sound Limiter.

FR2 The system will be able to build a Linux image for the Sound Limiter that includes the Heimdal Kernel Module.

NFR1 The Heimdal Kernel Module will be included as a Kernel built-in in production images to avoid reverse engineering.

NFR2 The Heimdal Module will be included as a Kernel module in development images.

NFR3 The Linux image built must be based on DietPi.

NFR4 The software must be easy to use.

NFR5 The software must be maintainable.

NFR6 The build system must be idempotent.

NFR7 The resulting Kernel must be built for the ARM architecture.

NFR8 The resulting Linux image must be built for the ARM architecture.

NFR9 The artifacts generated by the build system must be able to be used by the Sound Limiter 4.

NFR10 The resulting image must have basic tools installed and enabled by default.

- NFR10.1 The resulting Linux image must have a Secure Shell (SSH) server installed and enabled by default.
- NFR10.2 The resulting Linux image must have a Network Time Protocol client installed and enabled by default.

NFR11 The resulting Linux image must be ready to be used in the Sound Limiter 4 without any further configuration.

NFR12 The instructions on how to install the software must be clear and easy to follow.

NFR13 The code must be either self-explanatory or have comments explaining its purpose.

NFR14 The build software must be able to run in x86-64 machines.

NFR15 The resulting image must have a static MAC address set to avoid it being changed everytime the Sound Limiter boots.

NFR16 It must be possible to generate the Kernel and the image separately.

NFR17 The build system must not require any additional files to be manually added to work.

NFR18 The resulting image must have the Heimdal Kernel Module installed and enabled by default.

NFR19 The resulting artifacts must have the version of the base dependencies used for building them in their filename.

- NFR19.1 The resulting Kernel must have the version of the Linux Kernel used in its filename.
- NFR19.2 The resulting Kernel image must have the version of the Heimdal Kernel Module used in its filename.
- NFR19.3 The resulting Linux image must have the version of the DietPi image used in its filename.

NFR20 The final Linux image must be as light as possible.

2.3. Heimdal Ansible

2.3.1. Functional Requirements

FR1 After running the automation software the Sound Limiter must be in a working state.

- FR1.1 The Heimdal Kernel Module must be loaded.
- FR1.2 An Network Time Protocol must be configured and enabled.
- FR1.3 The SSH server must be configured and enabled.
- FR1.4 The Sound Limiter must be able to connect to the Internet.

FR2 The system will be able to deploy updated firmware to the Sound Limiter.

FR3 The system will to set up the Sound Limiter with a graphical interface.

- FR3.1 If configured, the system will start with a Desktop Environment enabled.
- FR3.2 The boot Linux boot output will be hidden.
- FR3.3 The system will allow to modify the Desktop Environment with a custom theme.
- FR3.4 The system will be able to modify the wallpaper of the Desktop Environment.

FR4 The system will install the software necessary to reproduce music using the Sound Limiter.

FR5 The system will be able to provision a Sound Limiter with a static IP address.

FR6 The system will be able to provision production-ready Sound Limiters.

FR7 The system will be able to provision testing Sound Limiters.

FR8 The system will create a serial number for each Sound Limiter if it doesn't have one.

NFR1 The automation system must be as easy to use as possible.

NFR2 The system will configure production devices.

- NFR2.1 Only unprivileged users will be able to login production devices.
- NFR2.2 Production devices won't have any testing software installed.
- NFR2.3 Production devices will have a unique MAC address.
- NFR2.4 The unprivileged user will automatically login at startup if the graphical interface is enabled.

NFR3 The system will configure testing devices.

- NFR3.1 Privileged users will be able to login testing devices.
- NFR3.2 The Hardware Tests will be installed in testing devices.
- NFR3.3 A testing MAC address will be set in testing devices.
- NFR3.4 The Desktop Environment will be configured to allow to log in with a privileged user if the graphical interface is enabled.

NFR4 The automation software must be based on Ansible.

NFR5 The system will be able to use different Ansible inventories to configure the Sound Limiters.

NFR6 The system will be able to cache the Ansible facts of every Sound Limiter in the inventory to speed up the provisioning process.

NFR7 The automation system will be idempotent.

NFR8 The automation software must be easy to use.

NFR9 The automation software must be maintainable.

NFR10 There must be clear documentation on how to use the automation software.

NFR11 The deployed devices must have spare disk space.

NFR12 The deployed devices must have a light Random Access Memory (RAM) footprint.

2.3.2. Information Requirements

IR1 The serial number of the Sound Limiters will be a unique identifier for each device.

IR2 Information about every Sound Limiter will be stored in an Ansible inventory.

Chapter 3

Planning

The next image (Figure 3.1) shows see the Gantt diagram of the project. It represents the time that has taken to complete each task. The tasks are divided in the three main parts of the project, which have been already mentioned in section 1.3. For a more detail view of the diagram, it is possible to access the Notion page of the project, where all the subtasks that make up each task can be seen.

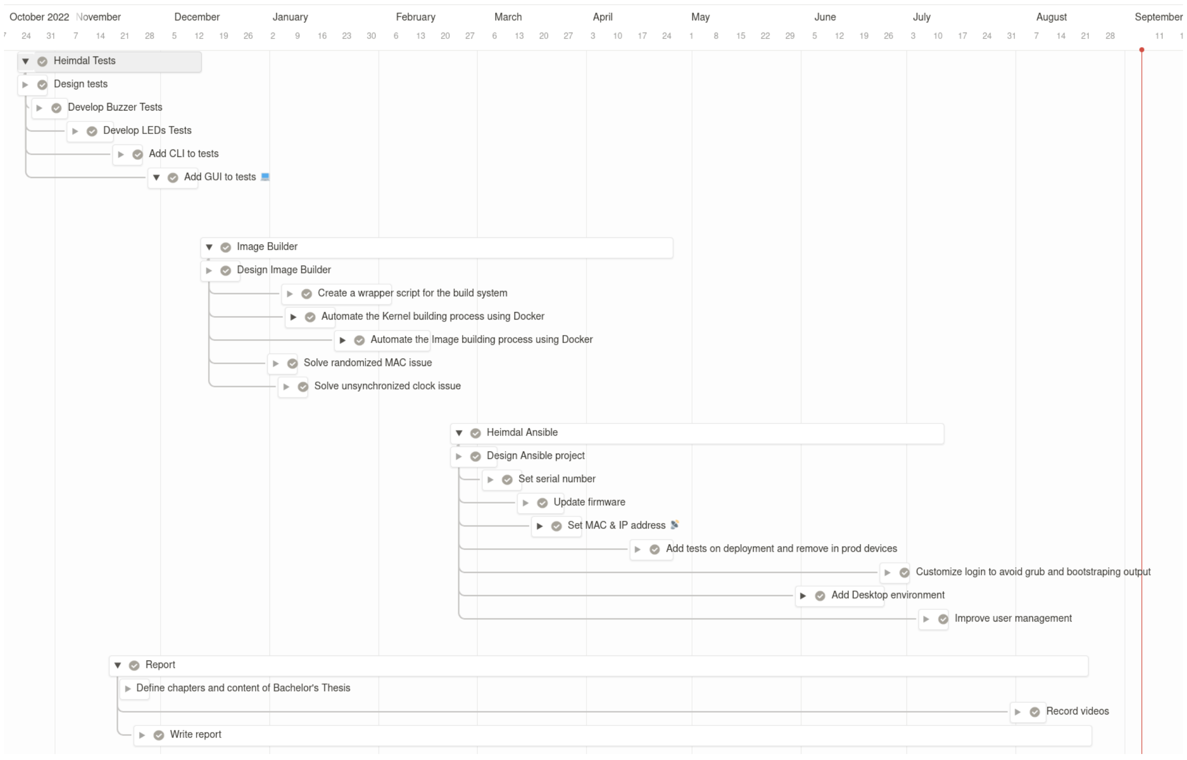


Figure 3.1 – *Gantt diagram of the project.*

The project has been developed using a waterfall model (Figure 3.2), which divides the project into different phases, each one of them with a specific goal[33, Chapter 2]. The phases are the following:

1. Requirements analysis and definition.

2. System and software design.
3. Implementation and unit testing.
4. Integration and system testing.
5. Operation and maintenance.

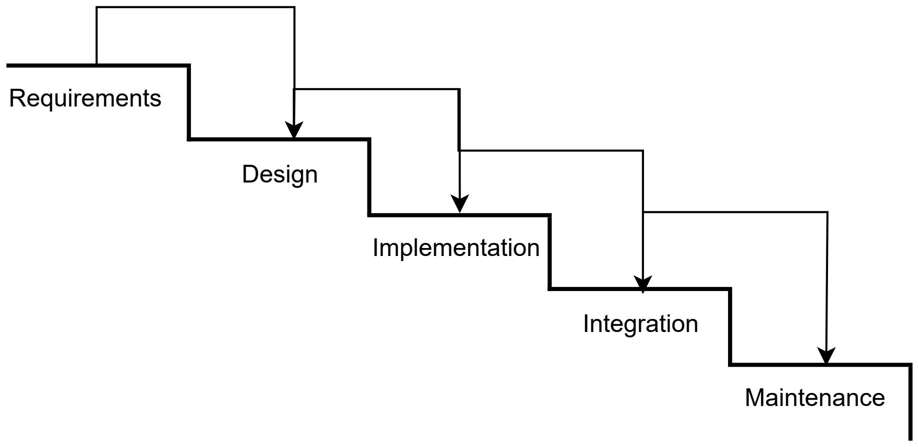


Figure 3.2 – *Waterfall model.*

In the waterfall model, the phases are carried out sequentially, and the next phase can not start until the previous one has finished. It is a simple model that is easy to understand and use, however, it is not suitable for projects where the requirements might change or it is necessary to deliver the software as fast as possible -even if it is not fully finished- and it is feasible to gradually improve it while it is being shipped to the client at the same time. Considering that this is not the case of this project, because the requirements where well defined after the very firsts meetings with the tutor it was a good choice to use this model for this particular case.

Since the project has been divided into three different subprojects, only the first phase was carried out for the whole project at the same time to be able to focus on each of the parts separately. The rest of the phases of the waterfall model were applied for every subproject individually, and each part of the project started once the previous one had finished, as the Grantt diagram shows (Figure 3.1), except for some eventual fixes that had to be done in some cases. The *Operaion and maintenance* phase (item 5) would not apply to this project, as it will be carried out by other students that will continue this work in the future.

Over the course of the Bachelor's Thesis, the communication with the project supervisor, which would act as the client of the project too, was done eventually to show the progress of the project and ask for feedback on specific aspects of it. This communication was conducted both in person and using Telegram. Also, some technical questions, were addressed to the GranaSAT team using a Telegram group for the team working on the Sound Limiters.

3.1. Project budget

The project has been developed using either free and open source software or free software, so most of the budget of this project would be spent on the hardware that was needed as well as on the salary of the team involved. Because of that, the software costs were none. The Sound Limiters were provided by the GranaSAT team, and the salary of the team was calculated using the average salary of a junior software engineer in Spain[21] and the average salary of a senior hardware engineer in Spain[20]. The final budget of the project can be seen in Table 3.1.

Concept	**Cost**	**Amount**	**Total**
Sound Limiter	1200€	2	2.000,00€
Junior Software Engineer/h	12.50€	300	3.750,00€
Senior Hardware Engineer/h	29.50€	70	2.088,26€
Software	0€		0,00€
Computer	1,200.00€	1	1.200,00€
Total			9.038.26€

Table 3.1 – *Budget of the project.*

Chapter 4

Design and Implementation

4.1. Introduction

In this chapter, the design and implementation of each of the three parts of the project will be explained. The first part is the Heimdal Tests, which are the tests that were developed to test the Sound Limiter hardware. The second part is the Image Builder, which is the software is used to build the Linux images for the Sound Limiter. The last part is the Heimdal Ansible, which is the automation software used to deploy and provision the Sound Limiter.

Each part will be explained in a different section, and each section will be divided into two subsections: **Design**, **Implementation**. The **Design** subsection will explain the design decisions that were made before the implementation of the software. The **Implementation** subsection will explain the implementation details of the software.

4.2. Heimdal Tests

The **hardware tests** are meant to be deployed to testing Sound Limiter using Heimdal Ansible as part of the deployment process, as Figure 4.1 shows.

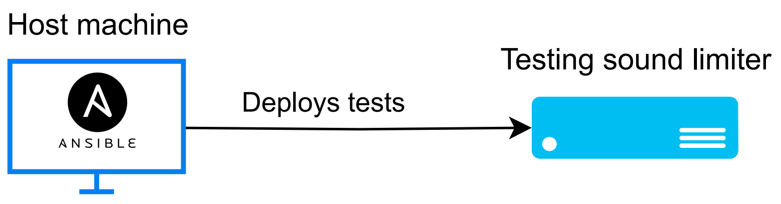


Figure 4.1 – *Test deployment to a testing Sound Limiter using Ansible.*

After the deployment, they can be used by testers to verify that the hardware installed on new Sound Limiters works as expected, either using the GUI or the Command Line Interface (CLI), as seen in Figure 4.2.

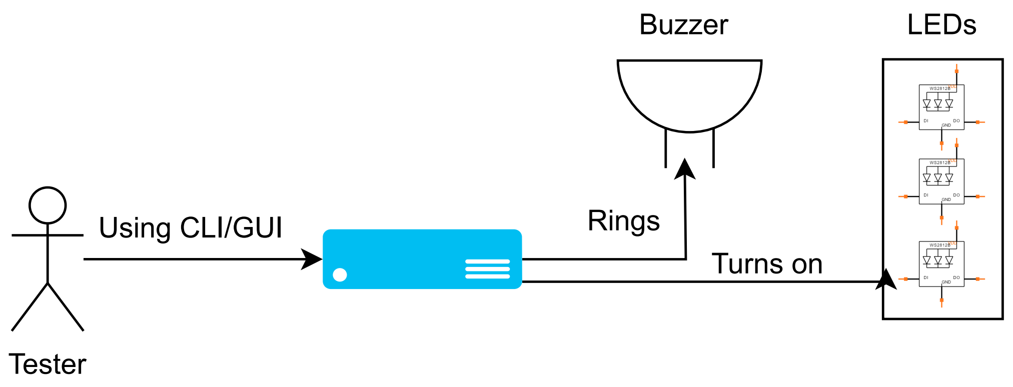


Figure 4.2 – *User running the hardware tests.*

4.2.1. Design

The tests developed in this project are a complete rework of the tests that were developed in previous iteration of the project. The previous tests were developed using the C programming language, and making use of the sourced code of the python's library RPi.GPIO. The main problem this approach had was that copying the source code of the library into the project was not a good idea, since it was not an easy task to update the tests afterwards because they were mixed with the source code of the library, making it very difficult to track the changes made to both the library and the tests.

Since one of the requirements of this part of the project (item **NFR2**) was to include a GUI for the tests, and the previous tests were a bit problematic to maintain, it was decided to rewrite them from scratch using the Python programming language, which is a suitable language for developing simple applications and has plenty of well-maintained libraries that support the development of applications for IoT devices [11].

The design of the tests was based on the Facade pattern (Figure 4.3), which is a pattern that uses a single class to provide a simple interface for a complex subsystem. This Facade class is the one that is used by the GUI and CLI views to perform their operations, and is the one responsible for instantiating the classes that control the different devices of the Sound Limiter. [8]

One of the most interesting advantages of using the Facade pattern is that the views are completely decoupled from the different subsystems. That means that the development of the views and the subsystems can be done in parallel after having agreed on the interface of the Facade class. It also means that any changes in how the subsystems work will not affect the views, since the interface of the Facade class will remain the same.

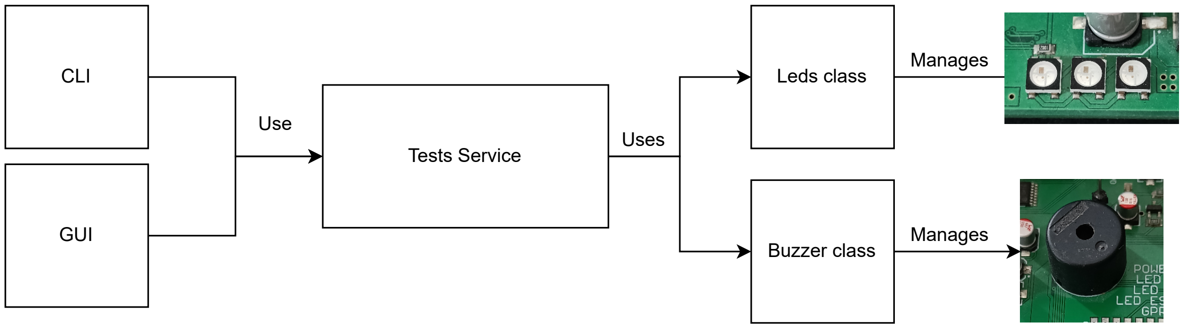


Figure 4.3 – *Facade pattern applied to the Heimdal tests.*

The interface (Figure 4.4) has a simple design, with one button for each test, and a text box that shows the output of the tests. This allows to debug any possible problems that might have occurred during the execution of the tests without having to execute the tests again using the CLI, which can be useful for test users that are not familiar with the Linux command line.

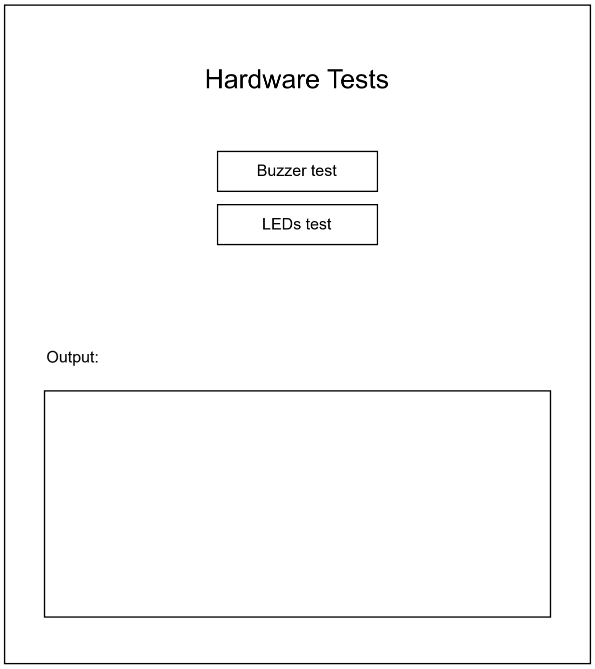


Figure 4.4 – *Design of the Heimdal Tests GUI.*

As for the dependencies used to develop the different parts of the tests, the following ones were used:

- **Python**: The programming language used to develop the tests.
- **RPi.GPIO**: The library used to communicate with the General Purpose Input/Output (GPIO) pins of the Raspberry Pi.

- **PyQt5**: The library used to develop the GUI of the tests. It was chosen over other alternatives because of the familiarity with the tool as well as being well supported and documented.
- **adafruit-circuitpython-neopixel**: The library used to control the LEDs of the Sound Limiter. It was chosen because it is the library recommended by the manufacturer of the LEDs.

4.2.2. Implementation

The implementation of the tests was divided into five main parts: the Facade class, the Graphical User Interface, the Command Line Interface, the Buzzer Tests, and the LEDs tests. The GUI and the CLI are the views of the Facade class, and the tests are the subsystems controlled by it.

4.2.2.1. Facade

The Facade class has been implemented in the following way:

```python
import logging
import time

from heimdal_tests_python.buzzer import Buzzer
from heimdal_tests_python.leds import Leds


class TestsService():
    SLEEP_TIME = 0.3

    def __init__(self):
        self.logger = logging.getLogger(self.__class__.__name__)

    def run_buzzer_tests(self):
        buzzer = Buzzer()
        self.logger.info("Test: buzzer success...")
        buzzer.buzz_success()
        time.sleep(self.SLEEP_TIME)
        self.logger.info("Test: buzzer failure...")
        buzzer.buzz_fail()

    def run_leds_tests(self):
        self.logger.info("Test: tinkle the LEDs...")
        Leds().tinkle_leds_test()
```

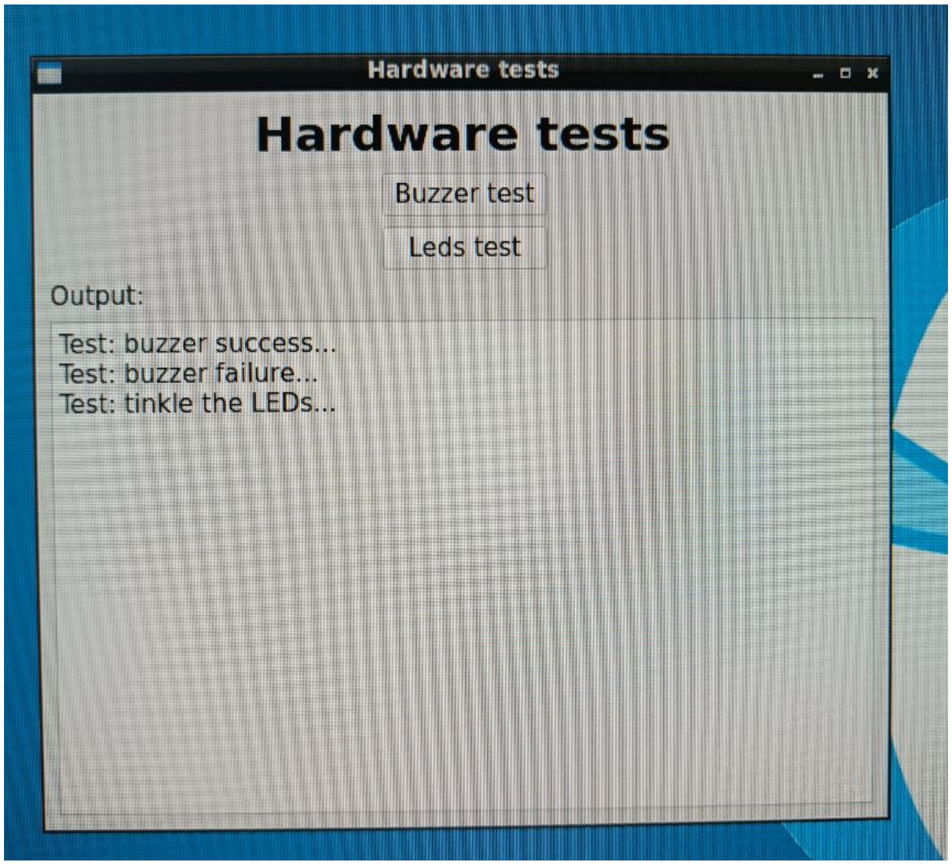
In this case, it is a pretty simple class, since it only has two methods, one for each type of test. This interface is used by the different views to perform the tests.

4.2.2.2. Graphical User Interface

In the case of the GUI, the interface is used in the following way:

```
    def __init__(self):
        super().__init__()
        self.tests_service = TestsService()
        ...
        def _add_buzzer_button(self):
        self.buzzer_button = QPushButton("Buzzer test")
        self.buzzer_button.setFixedWidth(self.BUTTONS_WIDTH)
        self.buzzer_button.clicked.connect(
            self.tests_service.run_buzzer_tests
        )
        self.addWidget(
            self.buzzer_button, alignment=QtCore.Qt.AlignmentFlag.
             AlignCenter
        )
    ...
```

The final aspect of the GUI can be seen in Figure 4.5. It follows closely the design (Figure 4.4), and allows the user to perform the different tests that were defined in the requirements (subsection 2.1.1).



```
Test: buzzer success...
Test: buzzer failure...
Test: tinkle the LEDs...
```

Figure 4.5 – *Final aspect of the Heimdal Tests GUI running in the Sound Limiter 4.*

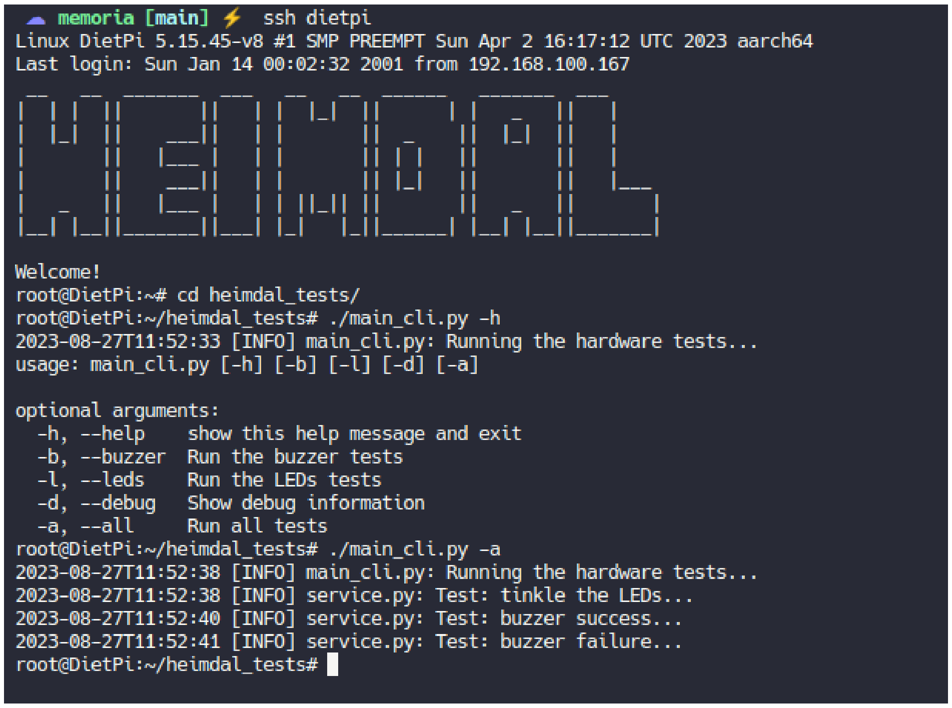
4.2.2.3. Command Line Interface

On the other hand, the CLI reads the parameters passed to it and it calls the corresponding methods of the Facade class:

```
...
service = TestsService()

tests = {
    "leds": service.run_leds_tests,
    "buzzer": service.run_buzzer_tests,
}
...
def main():
    logging.info("Running the hardware tests...")
    args = vars(parse_args())
    if args["debug"]:
        logger.setLevel(logging.DEBUG)
    args.pop("debug", None)
    if args["all"]:
        for key, value in tests.items():
            value()
    else:
        args.pop("all", None)
        for arg in args:
            if args[arg]:
                tests[arg]()
```

The CLI, it is a simple command line interface that allows the user to perform the same tests that can be executed using the GUI. The CLI is useful to be used either directly in the Sound Limiter or remotely using SSH.



```
 memoria [main] ⚡  ssh dietpi
Linux DietPi 5.15.45-v8 #1 SMP PREEMPT Sun Apr 2 16:17:12 UTC 2023 aarch64
Last login: Sun Jan 14 00:02:32 2001 from 192.168.100.167
 __   __  _______  ___   __   __  ______   _______  ___
|  | |  ||       ||   | |  |_|  ||      | |   _   ||   |
|  |_|  ||    ___||   | |       || _     ||  |_|  ||   |
|       ||   |___ |   | |       || | |   ||       ||   |
|       ||    ___||   | |       || |_|   ||       ||   |___
|   _   ||   |___ |   | | ||_|| ||       ||   _   ||       |
|__| |__||_______||___| |_|   |_||______| |__| |__||_______|

Welcome!
root@DietPi:~# cd heimdal_tests/
root@DietPi:~/heimdal_tests# ./main_cli.py -h
2023-08-27T11:52:33 [INFO] main_cli.py: Running the hardware tests...
usage: main_cli.py [-h] [-b] [-l] [-d] [-a]

optional arguments:
  -h, --help    show this help message and exit
  -b, --buzzer  Run the buzzer tests
  -l, --leds    Run the LEDs tests
  -d, --debug   Show debug information
  -a, --all     Run all tests
root@DietPi:~/heimdal_tests# ./main_cli.py -a
2023-08-27T11:52:38 [INFO] main_cli.py: Running the hardware tests...
2023-08-27T11:52:38 [INFO] service.py: Test: tinkle the LEDs...
2023-08-27T11:52:40 [INFO] service.py: Test: buzzer success...
2023-08-27T11:52:41 [INFO] service.py: Test: buzzer failure...
root@DietPi:~/heimdal_tests# █
```

Figure 4.6 – *Final aspect of the Heimdal Tests CLI. Executed remotely using SSH.*

4

4.2.2.4. Buzzer Tests

These tests have been implemented using a Buzzer class, which provides methods to ring a buzzer with success or failure sounds using PWM on one of the GPIO channels of the Raspberry Pi, using the library described in Figure 4.2.1. The signal is set to a increasing frequency for the success sound, and to a decreasing frequency for the failure sound, and its duty cycle (Figure 4.7)is set to 70% in both cases.

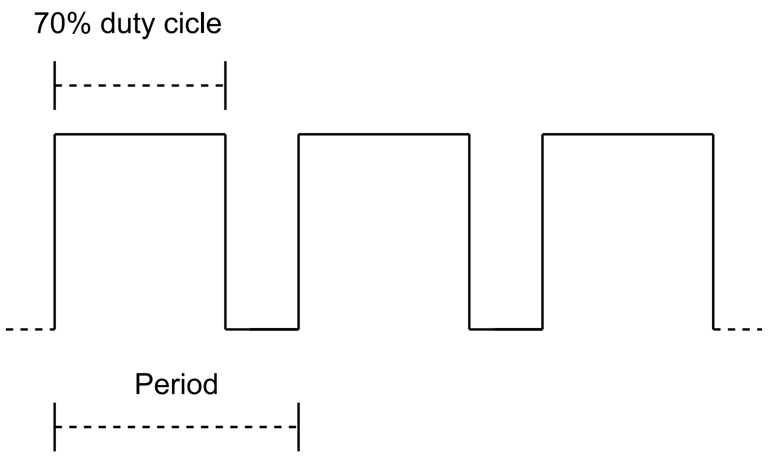


Figure 4.7 – *PWM signal used.*

After setting up the GPIO channel used -which is the 13th channel in the Raspberry Pi GPIO header- as the PWM output, the buzzer is used in the following way:

```python
    def __iterate_freqs(self, freqs: list[float]):
    """
    Set the frequency of the GPIO channel from a list of frequencies
    sleeping between every change.

    Args:
        freqs (list[float]): List of frequencies.
    """
    self.pwm.start(self.DUTY_CYCLE)
    for freq in freqs:
        self.pwm.ChangeFrequency(freq)
        time.sleep(self.SLEEP_TIME)
```

This method receives a list of frequency, which depends on the sound that will be played, and it iterates over it, setting the frequency of the PWM signal to the value of the current element of the list. It also sleeps for a short period of time between every change, to allow the tester to hear the sound.

4.2.2.5. LEDs Tests

The LEDs tests have been implemented using a Leds class, which provides a method to turn on and off the LEDs of the Sound Limiter using the GPIO channels of the Raspberry Pi via Figure 4.2.1 as well as the library provided by the manufacturer of the LEDs Figure 4.2.1. The LEDs are managed using the GPIO channel 12 of the Raspberry Pi. The method used to turn on and off the LEDs is the following:

```python
    def tinkle_leds_test(self):
    """
    Light and turn off every LED sequentially.
    """
    pixel = neopixel.NeoPixel(
        self.LEDS_PIN, self.NUM_LEDS,
        brightness=self.BRIGHTNESS,
        pixel_order=neopixel.RGB
    )
    for i in range(self.NUM_LEDS):
        pixel[i] = Colors.white.value
        time.sleep(self.SLEEP_LAPSE)
        pixel[i] = (0, 0, 0)
        time.sleep(self.SLEEP_LAPSE)
```

This method initializes the `Neopixel` object, which is the one that is used to control the LEDs, and then it iterates over the LEDs, turning them on and off sequentially, with a short pause between every change. In Figure 4.8 it can be seen how the LEDs look like while the test is running.

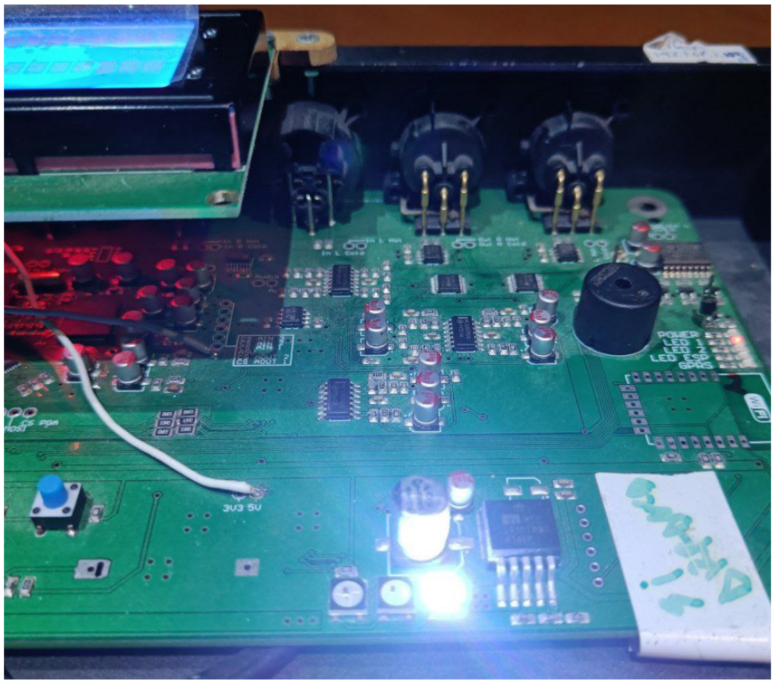


Figure 4.8 – *LEDs of the Sound Limiter while running the test.*

4.3. Image Builder

4.3.1. Design

The build system developed in this project has been developed using Docker and bash scripts. Docker is a tool that allows to create, deploy and run applications using containers, which are isolated environments that contain all the necessary dependencies to run the application. [17] The main advantages of using Docker for this task over virtual machines are:

- **Portability**: Docker containers are portable, which means that they can be run in any machine that has the Docker Engine installed, regardless of the operating system or the hardware of the machine. A virtual machine, on the other hand, requires a Hypervisor to run, which means that it can only be run in machines that have one installed, and it can only be run in machines that have the same architecture as the Hypervisor.
- **Lightweight**: Docker containers are lightweight, which means that they don't require a lot of resources to run. A virtual machine, on the other hand, requires a Hypervisor to run, which also requires a guest operating system (OS) to run. That means it adds to the resources required to run the build system, since the Docker containers use the same OS as the host machine.

To be able to build both the Kernel and the Linux image, the build system has been divided into two main parts: the **Kernel Builder** and the **Image Builder**. The Kernel Builder is the part of the build system that is responsible for building the Kernel and the Kernel Module, and the Image Builder is the part of the build system that is responsible for building the Linux image. The **Kernel Builder** is completely independent from the **Image Builder**, which means that it can be used to build the Kernel and the Kernel Module without having to build the Linux image. However, to avoid duplicating any logic, the **Image Builder** uses the **Kernel Builder** to build the Kernel and the Kernel Module. This has been achieved using multi-stage builds, which is a feature of Docker that allows to use the artifacts generated by a container in another container. [16]

The only problem left to address is the architecture incompatibility between the Raspberry Pi and the host machine. To overcome this on the **Kernel Builder** side, we can cross-compile the Kernel using the necessary toolchain. The Raspberry Pi documentation has a guide on how to achieve this.

As for the **Image Builder** the problem is a bit more complex. To initialize and configure a custom Linux image for the Sound Limiters, we need to mount that image in the host machine, chroot into it, and use its binaries to configure it. Because of the architecture incompatibility, it is not possible to do this directly, since the binaries of DietPi have been built for the ARM architecture, and the host machine is most likely using the x86-64 architecture. QEMU has been used to solve this issue, since it allows to use programs compiled for one architecture in another architecture using its user-mode emulation. [29]

Finally, it is not possible to extract the build artifacts of the **Kernel Builder** phase, which will be generated on the build phase of the Docker container, using the legacy Docker builder. To be able to do that, it is necessary to use the new **BuildKit** backend, which is not enabled by default.[13] According to the Docker documentation, the **buildx** plugin uses that build backend by default, apart from other features that are not available in the legacy builder, so it has been used in the build system.

Once the technical challenges have been addressed, it is worth mentioning that the build system has been designed to be as simple as possible to use, to comply with item **NFR4**. It will be explained in detail in subsection 4.3.2. The final design of the build system can be seen in Figure 4.9.

The build system has a number of convenient additional features that are worth mentioning, as they facilitate working with newly deployd Sound Limiters, as well as debugging any possible issues that the build system might have in future iterations:

- Possibility of including a SSH public key[1, Chapter 10] in the Linux image, which will be used to allow the user to connect to the Sound Limiter using SSH without having to use a password. This is useful to be able to connect to the Sound Limiter without having to use a keyboard and a monitor. The key is stored in a *.env* file, which could also be used in the future to store any API keys or sensitive information that the user might want to include in the resulting Linux image.
- Debug flag that allows to view all the commands that are being executed by the build system, as well as their output.
- The Raspberry Pi fork of the Linux Kernel is automatically cloned into the Docker container, which minimizes the manual steps that the user has to take to build the Kernel, which helps achieving item **NFR17**.
- The DietPi base image is automatically downloaded from the DietPi website, which also helps achieving item **NFR17**.
- Any build artifacts that are not necessary for the following build steps are removed, which helps decreasing the size of the Docker image.

4.3.2. Implementation

The instructions to install the necessary requirements have been included in the *README.md* file of the repository and are easy to follow as required in item **NFR12**. They have been described for Debian-based distributions as well as for Arch-based distributions, since they are the most popular

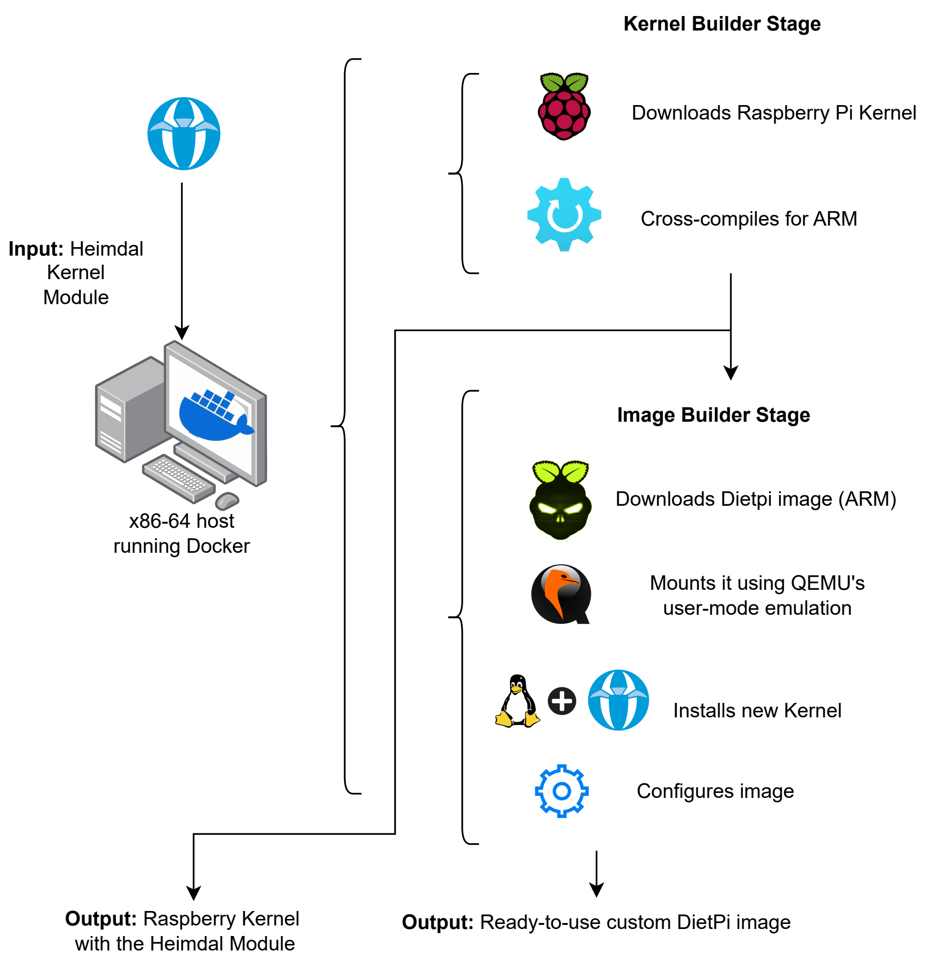

Kernel Builder Stage
Downloads Raspberry Pi Kernel
Cross-compiles for ARM
Input: Heimdal Kernel Module
x86-64 host running Docker
Image Builder Stage
Downloads Dietpi image (ARM)
Mounts it using QEMU's user-mode emulation
Installs new Kernel
Configures image
Output: Raspberry Kernel with the Heimdal Module
Output: Ready-to-use custom DietPi image

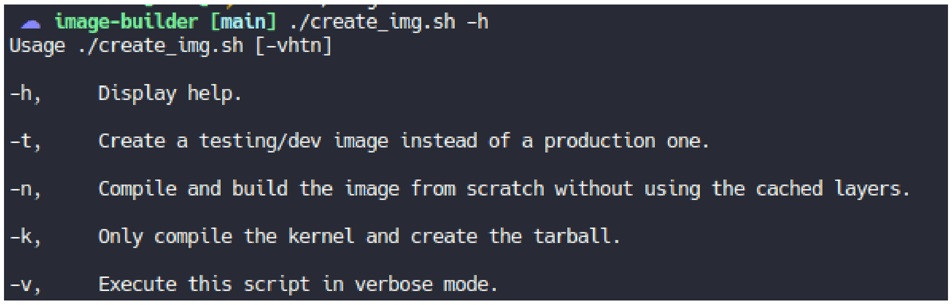
Figure 4.9 – *Image Builder design.*

distributions in the IoT community. But only the Debian installation instructions will be displayed here, as they are both very similar.

```
# To be able to use this repository, you need to have Docker
 installed on your computer.
# Follow the [official Docker installation instructions](https://
 docs.docker.com/engine/install/) to install it on your computer.
apt install qemu qemu-user-static docker-buildx
```

After executing the installation instructions, the build script is ready to be used. It is a bash script that uses a set of arguments to configure the build process, and acts as a wrapper for the Dockerfile, making the build system easy to use even for users without technical knowledge about how Docker works. The arguments it can use are the following Figure 4.10:

The build script is in charge of executing Docker with the necessary arguments to build the required artifacts for that execution. Excluding the argument parsing, the build script is the following:



```
▲ image-builder [main] ./create_img.sh -h
Usage ./create_img.sh [-vhtn]

-h,     Display help.

-t,     Create a testing/dev image instead of a production one.

-n,     Compile and build the image from scratch without using the cached layers.

-k,     Only compile the kernel and create the tarball.

-v,     Execute this script in verbose mode.
```

Figure 4.10 – *Arguments of the build script.*

```bash
parseArgs $@

# Prepare the system to emulate ARM architecture in Docker
docker run --rm --privileged=true multiarch/qemu-user-static --reset -p yes

BUILD_ARGS=$([ "$PROD_BUILD" = "false" ] && echo "--build-arg PROD_BUILD=false" || echo "")
BUILD_ARGS+=$([ "$KERNEL_ONLY" = "true" ] && echo " --target kernel_exporter" || echo "")
BUILD_ARGS+=$([ "$NO_CACHE" = "true" ] && echo " --no-cache" || echo "")
BUILD_ARGS+=$([ "$VERBOSE" = "true" ] && echo " --progress=plain --build-arg VERBOSE=true" || echo "")

DOCKER_BUILDKIT=1 docker build -t img_builder ./docker $BUILD_ARGS

if [ "$KERNEL_ONLY" = "true" ]; then
    # We need to repeat the build command so that first we cache
    # the image and now we can export the output files to the host
    DOCKER_BUILDKIT=1 docker build -t kernel_exporter ./docker $BUILD_ARGS \
        --output build
    exit 0
fi

# Mount /dev in the container, otherwise new device nodes
# won't be available there (https://github.com/moby/moby/issues/27886)
docker run --rm --privileged -v $PWD/build:/app/build -v /dev:/dev -e CONTAINER_OWNER=$(id -u) -e VERBOSE=$VERBOSE --env-file .env img_builder
```

As can be seen in the code, any steps that are not self-explanatory has been commented to improve the maintainability of the project, as required by item **NFR5**.

The multi-stage Dockerfile that is used to build both the Kernel and the Linux image is the following:

```
ARG DIETPI_VERSION=ARMv8-Bullseye
ARG KERNEL_VERSION=6b945e6f
ARG HEIMDAL_VERSION=0.1
ARG VERBOSE=false

FROM debian:bookworm-20221024-slim AS kernel_builder
WORKDIR /app
ARG KERNEL_VERSION
ARG HEIMDAL_VERSION
ARG VERBOSE
RUN apt update && apt install -y \
    bc \
    bison \
    cpio \
    crossbuild-essential-armhf \
    crossbuild-essential-arm64 \
    curl \
    flex \
    git \
    kmod \
    libc6-dev \
    libssl-dev \
    libncurses5-dev \
    make \
    rsync \
    sshfs \
    xz-utils && \
    curl -L https://api.github.com/repos/raspberrypi/linux/
     tarball/$KERNEL_VERSION -o linux.tar.gz
COPY ./kernel_builder .
RUN chmod +x ./kernel_build.sh
ARG PROD_BUILD=true
RUN "./kernel_build.sh"

FROM scratch AS kernel_exporter
COPY --from=kernel_builder /app/heimdal_kernel_*.tar.gz /

FROM alpine:3.17.0 AS img_builder
ARG DIETPI_VERSION
ARG KERNEL_VERSION
ARG HEIMDAL_VERSION
ARG VERBOSE
ARG PROD_BUILD
WORKDIR /app
ENV DIETPI_VERSION=$DIETPI_VERSION \
    KERNEL_VERSION=$KERNEL_VERSION \
    HEIMDAL_VERSION=$HEIMDAL_VERSION \
    VERBOSE=${VERBOSE} \
    PROD_BUILD=${PROD_BUILD}
RUN apk update && apk add \
    coreutils \
    curl \
    p7zip \
    util-linux && \
    curl https://dietpi.com/downloads/images/DietPi_RPi-
     $DIETPI_VERSION.7z -O && \
    7z x DietPi_RPi-$DIETPI_VERSION.7z -so > DietPi.img \
    && rm DietPi_RPi-$DIETPI_VERSION.7z
COPY --from=kernel_exporter /heimdal_kernel_*.tar.gz ./
COPY ./img_builder .
RUN chmod +x ./img_build.sh
ENTRYPOINT [ "./img_build.sh" ]
```

The versions of both the Linux Kernel and the DietPi base image are set as build arguments, and are part of the repository. This means that git tags can be created to track the versions of each of those dependencies and the changes in the build scripts that modifying those versions might require. This is specially important for the Linux Kernel, since it is the dependency that is most likely to require those kind of changes in the future, as happened in the previous iteration of the project.

As can be seen in the code excerpt, there are three build stages:

1. **kernel_builder**: This stage is in charge of building the Kernel and the Kernel Module. It uses the *kernel_build.sh* script to build the Kernel and the Kernel Module.
2. **kernel_exporter**: Due to how the BuildKit output feature works, it is necessary to add an intermediate stage to export only the required build artifacts to the host machine. If this stage was not added, the BuildKit output flag would end up exporting all the files generated by the **kernel_builder** stage, which would include unnecessary files.
3. **img_builder**: This stage is in charge of building the Linux image. It uses the Kernel compiled in the previous stage, and the *img_build.sh* script to build the custom DietPi image.

4.3.2.1. Kernel Builder

The **Kernel Builder** is the part of the build system that is in charge of building the Kernel and the Kernel Module. It is a bash script that uses the Raspberry Pi documentation as the base instructions to cross-compile the Kernel and the Kernel Module. Depending on the arguments passed to the script, it will build either the production Kernel or the development Kernel, being the difference between the two that the production Kernel is built with the Heimdal Kernel Module built-in (as required by item **NFR1**), and the development Kernel is built with the Heimdal Kernel Module as a Kernel Module (as required by item **NFR2**).

The specific configuration required to build the Heimdal Kernel Module is injected into the Kernel configuration before the compilation process starts. This is an improvement over the previous build system, since it used to copy all the files that were necessary from the source Linux Kernel and then modified them, which was not the cleanest approach, since it was not possible to update the source Linux Kernel without having to manually update the files that were copied.

Now these changes are applied to the existing Kernel configuration files without having to copy them. This is done in the following excerpt of the *kernel_build.sh* script:

```
    ##### CUSTOM CONFIGS #####
    cp ./config_kernel/configs/bcm2711_defconfig ./linux/arch/arm64/
     configs/bcm2711_defconfig
    cp ./config_kernel/dts/* ./linux/arch/arm64/boot/dts/overlays/
    cp ./config_kernel/heimdal_module/* ./linux/sound/soc/bcm/

    # Customize Makefile
    cat >> ./linux/sound/soc/bcm/Makefile << EOF

    snd-soc-heimdalsoundcard-objs := heimdalsoundcard.o
    obj-\$(CONFIG_SND_HEIMDALSOUNDCARD) += snd-soc-heimdalsoundcard.o
    EOF

    # Customize Kconfig
    cat >> ./linux/sound/soc/bcm/Kconfig << EOF

    config SND_HEIMDALSOUNDCARD
            tristate "Support for Heimdal Sound Card"
            depends on SND_BCM2708_SOC_I2S || SND_BCM2835_SOC_I2S
            select SND_SOC_CS42XX8_I2C
            help
              Say Y or M if you want to add support for Heimdal
               Loader Sound Card add on
    EOF

    if [ "$PROD_BUILD" = "false" ]; then
        # Compile the heimdal module as an external module instead of
          embedded in the kernel
        sed -i "s/\(CONFIG_SND_HEIMDALSOUNDCARD=\)y/\1m/" ./linux/
         arch/arm64/configs/bcm2711_defconfig
    fi
    # Compile kernel

    cd ./linux
    make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
     bcm2711_defconfig
    make -j$(nproc) ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image
      modules dtbs
```

After the Kernel has been compiled, it is compressed and archived in the following way:

```
    # Store result files
    make modules_install INSTALL_MOD_PATH="../$OUTDIR/modules"
    cd ..

    cp ./linux/arch/arm64/boot/dts/broadcom/*.dtb $OUTDIR
    cp -r ./linux/arch/arm64/boot/dts/overlays/ $OUTDIR
    cp ./linux/arch/arm64/boot/Image $OUTDIR/${OUTNAME}.img

    tar -cavf ${OUTNAME}.tar.gz $OUTDIR
```

The resulting tar file has the following file name to indicate the version of the Kernel and the Kernel Module that it contains: *heimdal_kernel_<prod||dev>_<LINUX_KERNEL_VERSION>_<HEIMDAL_MODULE_VERSION>.tar.gz*. This helps achieve item **NFR19**, and more specifically, item NFR19.1 and item NFR19.2.

An example of the filename of a compiled Kernel would be: *heimdal_kernel_prod_6b945e6f_0.1.tar.gz*.

4.3.2.2. Image Builder

The **Image Builder** is the part of the build system that builds the custom DietPi image. It uses the Kernel generated in the previous steps, and uses two different bash scripts to apply all the modifications that are necessary:

- **img_build.sh**: This script is in charge of mounting the DietPi image, chrooting into it, and executing the *img_init.sh* script. After that, it cleans any temporary files that might have been generated and stores the resulting image in the *build* folder.
- **img_init.sh**: A plain bash script that is executed by a running DietPi OS. It performs all the customizations the OS might need to have a Linux image ready to use on any Sound Limiter.

The most relevant excerpt of the img_build.sh script is the following:

```
cnt=0
LOOP_DEV="$(losetup --show --find --partscan "$IMG")"
BOOT_DEV="${LOOP_DEV}p1"
ROOT_DEV="${LOOP_DEV}p2"

# Mount DietPi image
ROOT_FS_TYPE="ext4"
BOOT_FS_TYPE="vfat"
mount -v $ROOT_DEV $WORKDIR/ -t $ROOT_FS_TYPE
mount -v $BOOT_DEV ${WORKDIR}/boot/ -t $BOOT_FS_TYPE

# Mount virtual filesystems
mount -t proc /proc "$WORKDIR/proc"
mount --rbind /sys "$WORKDIR/sys"
mount --rbind /dev "$WORKDIR/dev"

# Add custom kernel and customize it
mkdir heimdal_kernel
tar -xvf ./heimdal_kernel_*.tar.gz --strip-components 2 -C ./
 heimdal_kernel
cp -a ./heimdal_kernel/*dtb ./heimdal_kernel/*img "$WORKDIR/boot"
cp -a ./heimdal_kernel/overlays/* "$WORKDIR/boot/overlays"
cp $IMG_INIT_SCRIPT "$WORKDIR/"
cp -a ./heimdal_kernel/modules/lib/modules/* "$WORKDIR/lib/modules"
chroot "$WORKDIR" /bin/sh -c "SSH_PUB_KEY=\"$SSH_PUB_KEY\"
 KERNEL_VERSION=$KERNEL_VERSION /$IMG_INIT_SCRIPT"
```

As can be seen in the code, the script creates a Loop Device, mounts the DietPi image there, and then mounts the virtual filesystems that are necessary to be able to chroot into the DietPi image. After that, it decompresses and copies the Kernel and the Kernel Modules into the mounted */boot* partition, and executes the *img_init.sh* script. After this, all the temporary files, like the loop devices, are cleaned, and the resulting image is stored in the *build* folder. This part of the build system has been implemented after studying how the DietPi image is built as a main resource.

On the other hand, the img_init.sh script is in charge of performing all the customizations to the DietPi image. The operations it performs are implemented in functions that are called in the bottom of the script. This way, it is easy to understand what the script does at a quick glance, as well as to add new operations in the future.

The customization script carries out the following tasks:

```bash
printf "Updating repositories"

apt update

# Setup an NTP client to update the system clock
ntp_message="Setting up the NTP service to configure the system clock.
Please, take into account that this process might be noticeably slow.
"
printf "$ntp_message"

setup_ntp

printf "Installing OpenSSH as SSH server."
install_openssh

# Setup a static and randomized MAC address to avoid having to reconfigure
# the DHCP server on every reboot
mac_message="Setting up a static MAC address."
printf "$ntp_message"
set_static_mac

# Configure the raspberry pi to use the heimdal kernel
# and the different chips in the PCB
printf "Setting up the /boot/config.txt file."
setup_config_txt

# Install programs that require no special configuration
# from our side
printf "Installing other programs"
install_other_programs # Currently only python3

add_ssh_pub_key

printf "Done! Enjoy your custom Image!"
```

These tasks are the ones that are necessary to achieve item **NFR11**. The reasoning behind each of them is the following:

- **Updating repositories**: This is necessary in order to install any program in the DietPi image.
- **Setting up the NTP service**: This is necessary to be able to set the clock of the Sound Limiter to the correct time. The NTP client configured by default, *systemd-timesyncd*, was not working properly because every time the device was disconnected from the power supply, the clock was reset to a high value, which caused the Network Time Protocol client to malfunction. To solve this, the *ntp* package was installed, which is a more robust NTP client. This achieves item NFR10.2
- **Installing OpenSSH**: The OpenSSH SSH client and server, which is the most widely used SSH implementation, is installed to allow the user to connect to the Sound Limiter remotely. This accomplishess the item NFR10.1.
- **Setting up a static MAC address**: This is necessary to prevent the need to reconfigure the Dynamic Host Configuration Protocol (DHCP) server on every reboot, accomplishing item **NFR15**.
- **Setting up the /boot/config.txt file**: This is necessary to configure the Sound Limiter to use the Heimdal Kernel and the different chips in the PCB, to comply with item **NFR18**.
- **Installing Python3**: Installing Python3 is a prerequisite for using Ansible to manage the device later on, which is required by item **NFR11**.
- **Adding the SSH public key**: This is a convenience feature that allows the user to connect to the Sound Limiter without having to use a keyboard and a monitor to configure the SSH keys for the first time. This is not explicitely required, but it is a nice to have feature that helps achieve item **NFR11**.

4

4.3.3. Build times

The build times of the build system are an important metric to take into account, since they can be a bottleneck in the development process. The build times have been measured using `date +%s` -which returns the number of seconds since the Unix Epoch-[26] before and after the build process, since `time` did not return the expected results because of the way the Docker containers are executed. The results of the build times, as well as the specifications of the machines they have been measured on, can be seen in Table 4.1. Differences between prod and testing builds have been omitted, since they are negligible.

Build type	**Desktop Computer**(4.3.3)	**Laptop**(4.3.3)
Build Kernel from scratch	8m 31s	12m 30s
Build Image from scratch	2m 11s	2m 39s
Build Image and Kernel from scratch	10m 46s	15m 20s

Table 4.1 – *Build times of Image Builder on different machines.*

Machine Specifications:

- Desktop Computer: AMD Ryzen 7 3700X (8 cores), RAM: 32GB, OS: Arch Linux.
- Laptop: AMD Ryzen 5 5700U (8 cores), RAM: 16GB, OS: Ubuntu 22.04.

4.4. Heimdal Ansible

In this section, the design and implementation of the Ansible environment that has been developed to deploy and provision the Sound Limiters will be explained, as well as the reasoning behind choosing Ansible instead of other automation tools. Being this an integral part of the project, as well as a complex one, it will be explained in detail. A diagram that shows how the automation system would manage different sets of Sound Limiters can be seen in Figure 4.11.

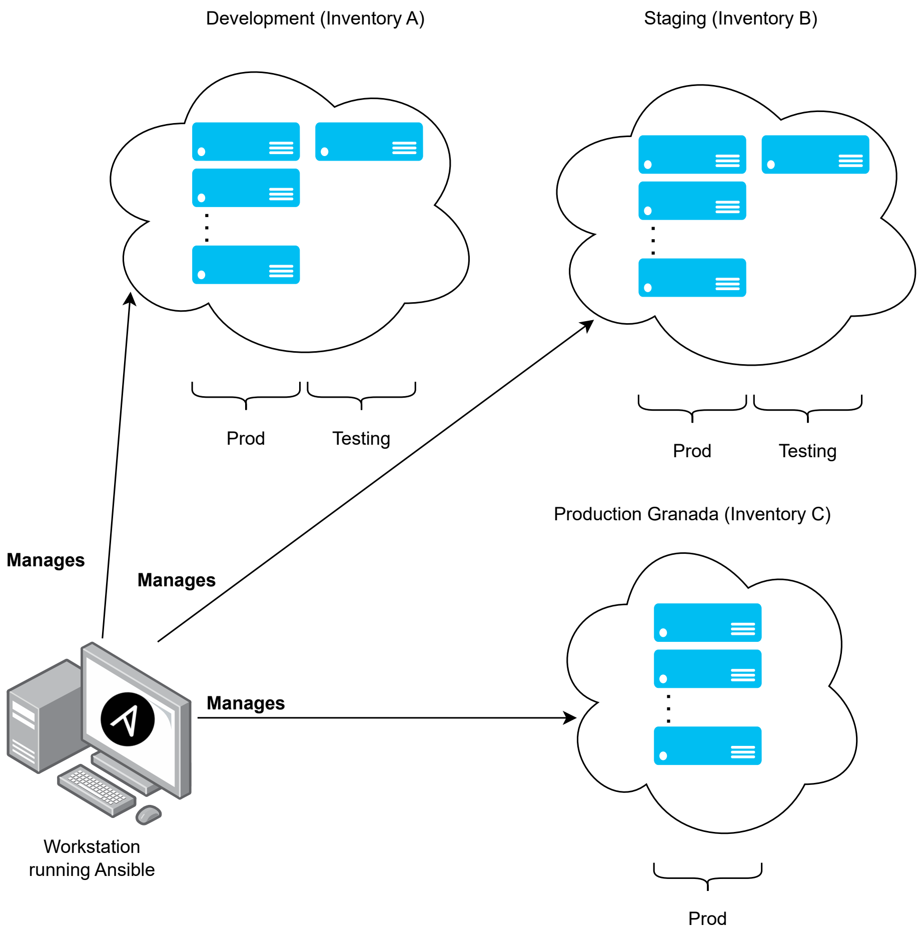

Development (Inventory A)
Staging (Inventory B)
Prod
Testing
Prod
Testing
Production Granada (Inventory C)
Manages
Manages
Manages
Workstation running Ansible
Prod

Figure 4.11 – *Ansible environment design.*

The Ansible environment has been designed to be as simple as possible to use, to comply with item **NFR1**. It will be used in Sound Limiters that has already been flashed with the custom DietPi image that has been built using the Image Builder.

4.4.1. Why Ansible?

The first question that arises when talking about the automation system developed for this project is why Ansible has been chosen over other automation tools. In (Table 4.2), a comparison between Ansible and other popular solutions can be observed.

	Ansible	**Puppet**	**Chef**
Agentless	✓		
Idempotent	✓	✓	✓
Declarative	✓	✓	✓
Ease of Use	High ease of use due to YAML-based playbooks and agentless architecture.	Lower ease of use for newcomers due to Puppet's DSL and complex configurations.	Moderate ease of use with a steeper learning curve for setting up Chef recipes and cookbooks.
Scalability	High	Medium	High
Community and Support	Large	Large	Large
Integration Ecosystem	Extensive	Moderate	Extensive
Platform Support	Linux/Unix, Windows	Linux/Unix, Windows	Linux/Unix, Windows

Table 4.2 – *Comparison between Ansible, Puppet, and Chef based on various criteria.*

Althought Ansible and its primary competitors share similarities, they have some key differences that make Ansible the optimal choice for this project. Ansible is an **agentless** tool, which means that it can manage devices without having to install any additional software on them, it only necessitates an SSH server and a Python interpreter on the clients. Given that this tool will be used to manage IoT devices, which have very limited resources, it is crucial to minimize the footprint of any administrative software used on these devices. It is also slightly easier to use - specially for newcomers to the DevOps world - than its competitors. These are the main reasons why Ansible has been chosen as the preferred automation tool in this Bachelor's Thesis.

4.4.2. Design

When a project has a certain complexity, it is important to have the best practices of the tools used in mind to avoid reaching a state where the project is not maintainable anymore due to the lack of a good design. This part of the project could be prone to that, because it involves many different tasks that need to be carried out to leave the Sound Limiters in the desired state, and it would be easy to end up with an Ansible environment that is hard to maintain.

To avoid that, the first design decision that has been taken is to use Ansible Roles. This has been done to abstract the different tasks that need to be performed for each kind of device, and to be able to reuse them. The roles that have been defined for this project are the following:

- Common Role: This role is in charge of performing the common tasks that need to be done for all the Sound Limiters. It is the role which has the most responsibilities.

- Testing Role: This role performs the tasks that are necessary only for devices that are still in the testing phase.
- Prod Role: This role performs tasks for devices that have passed the testing phase and are either ready for production or already in production.

All of these roles and the tasks that they perform have been developed to be idempotent. This means that, if the role is executed multiple times, the result will be the same as if it was executed only once. This is important to avoid having to manually check the state of the Sound Limiters after executing the Ansible environment, which is a requirement of item **NFR7**. This property of the ansible roles developed, along with having a separate role for production devices, helps having clear which hardening tasks are applied to devices deployed in production, and making sure that all of them are in the same state, which usually is a business critical requirement.

The organization of the inventary file has been designed to be as simple as possible. To achieve item **FR8**, the serial number of every device has been added as a property of each host. An example of an inventary file is the following:

```yaml
---
heimdal_devices:
  children:
    testing:
      hosts:
        test01:
          ansible_host: 192.168.100.146
          serial_number: 92398ac4-1974-4c56-85f6-3ba5ce519e9a
    prod:
      hosts:
        prod01:
          ansible_host: 192.168.100.148
          serial_number: ef8e19cf-7f40-4953-bc10-7d7097c3d5a5
```

The automation system can also be configured to use different inventories, which is useful to deploy the Sound Limiters in different environments, and it also allows to have a testing inventory with a set of devices that simulate the behavior they would have in production to test any new changes before shipping them to production.

Since the duties every role perform are very different between them, the rest of this section will be divided into three parts, one for each role.

4.4.2.1. Common Role

This is the role that performs the most tasks, and the other roles are built on top of it and responsible of testing-only and production-only tasks respectively. The tasks that this role performs are the following:

- Installing necessary packages.
- Making sure that NTP is setup correctly.
- Installing the Heimdal Kernel or updating it when necessary.

- Setting the serial number of the device.
- Configuring the network interfaces. This will set the testing MAC address if the device is in the testing inventory, and a randomized MAC address if it is in the production inventory.
- Setting up a graphical desktop and customizing it.
- Add an unprivileged user to the device to be used by the end user.
- Disable the dietpi user from logging in into the device.
- Configuring autologin for the unprivileged user if the device is in the production inventory.

4.4.2.2. Testing Role

This role is in charge of deploying the tests in the device and making sure that they are ready to be executed without any additional steps. The tasks that this role performs are the following:

- Install testing packages that will be necessary to deploy the tests.
- Check if the heimdal_tests have been cloned.
- Install the tests dependencies in the device. The tests will be included in the repository as a git submodule, so that any changes to them can be tracked.
- Copy the heimdal_tests directory to the remote host.
- Allow privileged users to log in into the device.

4.4.2.3. Prod Role

The last role is in charge of cleaning up any packages and files that are not necessary in production, as well as to harden the device. The tasks that this role performs are the following:

- Cleaning up testing packages.
- Removing the test files.
- Disabling privileged users from logging in into the device.

4.4.3. Implementation

Now that it is clear how the automation system has been designed, it is time to explain how it has been implemented, as well as the technical challenges that have been faced to develop the tasks. Since some of the tasks could be performed using basic Ansible modules, and others required more complex logic, the most basic tasks will be just mentioned, and the most complex ones will be explained in detailed sections.

4.4.3.1. Common Role

The tasks that could me implemented using just Ansible modules are the following:

- Installing necessary packages: This is done using the apt module.
- Making sure that NTP is setup correctly: This is done using the systemd module.
- Add an unprivileged user to the device to be used by the end user, and disable the dietpi user from logging in into the device: Performed using the user module.

4.4.3.1.1. Installing the Heimdal Kernel or updating it when necessary

For this task, a separate playbook that handles the logic of deploying new firmware when necessary has been developed. To check the current version of the Kernel, the following tasks are executed by Ansible:

```
- name: Check target kernel state
  stat:
    path: "/boot/{{kernel_img}}"
  register: kernel_img_stat
  tags:
    - kernel

- name: Register if the target kernel is installed or not
  set_fact:
    kernel_installed: "{{kernel_img_stat.stat.exists}}"
  tags:
    - kernel

- name: Install kernel
  include_tasks: install_kernel.yml
  when: not kernel_installed or force_kernel_install
  tags:
    - kernel
```

As can be seen in the code, the *install_kernel.yml* playbook is only executed if the Kernel is not installed or if the *force_kernel_install* variable is set to true. The file check by the *stat* module is set in the *kernel_img* variable, which has the format set by the Kernel Builder section. To install the Kernel, the following tasks are executed:

```
    ---
    - name: Create temporary directory
      tempfile:
        state: directory
      register: tempdir

    - name: Install kernel
      block:
        - name: Copy and unarchive kernel
          unarchive:
            src: "{{kernel_tar}}"
            dest: "{{tempdir.path}}"
            owner: root
            group: root
            mode: 0600

        - name: Install kernel modules
          copy:
            src: "{{tempdir.path}}/kernel_build/modules/lib/modules/"
            dest: /lib/modules/
            owner: root
            group: root
            mode: 0755
            remote_src: yes

        - name: Install kernel binary
          copy:
            src: "{{tempdir.path}}/kernel_build/{{kernel_img}}"
            dest: /boot/{{kernel_img}}
            owner: root
            group: root
            mode: 0755
            force: true
            remote_src: yes

        - name: Register all dtb
          find:
            paths: "{{tempdir.path}}/kernel_build"
            patterns: "*.dtb"
          register: dtb_files

        - name: Install dtb files
          copy:
            src: "{{item.path}}"
            dest: /boot/
            owner: root
            group: root
            mode: 0755
            remote_src: yes
          with_items: "{{dtb_files.files}}"

        - name: Install overlays
          copy:
            src: "{{tempdir.path}}/kernel_build/overlays/"
            dest: /boot/overlays/
            owner: root
            group: root
            mode: 0755
            remote_src: yes

        - name: Clean old heimdal kernels
          import_tasks: clean_old_kernels.yml
          when: clean_old_kernels

      always:
        # Cleanup temporary files on failure
        - name: Remove temporary directory
          file:
            path: "{{tempdir.path}}"
            state: absent
```

These task copy and unarchive the kernel, install the kernel modules and binary, registering and installing Device Tree Blob (DTB) files, installing Overlays, and cleaning up old kernels. The playbook also removes the temporary directory used for the kernel installation to avoid wasting disk space in the device. Note that the target Kernel tarball must be first available in the host machine under the *kernel_tar* variable, which is by default set to */roles/common/files/kernels/<target_kernel>*.

4.4.3.1.2. Setting the serial number of the device

After searching through the available Ansible modules, it was not possible to find any that could be used for this specific task, since it requires to register information at runtime in the host that is running the Ansible environment. To solve this, a custom Ansible module has been developed. It is a Python script that has been developed using the official Ansible documentation as a guideline. This module stores the serial number of the device in the inventory (section 4.4.2) that is being used, so that it might be used by other roles as well as by administrators, being the inventory a human readable file.

The custom module is used in the following way:

```
- name: Generate and store the serial number in the inventory
    store_serial_number:
        host: "{{inventory_hostname}}"
        inventory: "{{inventory_file}}"
        device_group: "{{group_names | select('match', 'testing|prod
         ') | first}}"
    delegate_to: localhost
    become: false
    throttle: 1
    when: not serial_number_was_set
    tags:
    - serial_number
```

The target host, inventory file, and the group the device belongs to must be passed to the module, since Ansible modules have not access to facts or variables that have not been explicitely passed. It is important to mention that it is necessary to use the '`throttle: 1`' option to avoid having concurrency issues when multiple hosts are being provisioned at the same time, since the inventory file is a shared resource between all the hosts. To be able to use the module, it must be placed in the */library* folder of the Ansible environment with the name *store_serial_number.py*.

Since at the moment there is not a default library to work with YAML files in Python, the ruamel.yaml library has been used to parse the inventory file instead of other popular option, PyYAML, because it allows to preserve the comments in the inventory file, which is important in a human-managed file like the inventory. The code of the custom module is rather simple, it just opens the inventory file, parses it, searches for the host and, if it does not already have a serial number, generates a random UUID4 and stores it in the inventory file. It also checks for possible errors (like missing the `ruamel.yaml` library) and returns meaningful error messages in those cases.

4.4.3.1.3. Configuring the network interfaces

This task achieves two different goals, setting up a static MAC address and a static IP address, restart the network service to apply the changes, and wait for the device to be reachable again. The MAC address can be either a set one or a randomized one, depending on the inventory group the device belongs to. The IP address is configurable, if it is set in the host variables a static one will be used, and if not, it will be assigned by the DHCP server. The network configuration is made via the Debian interfaces file using a template, that will be only applied after checking that the configuration is not already in place to avoid unnecessary restarts of the network service.

4.4.3.1.4. Setting up a graphical desktop and customizing it

Previous versions of the Sound Limiter used lxqt as the desktop environment, because it is lightweight and easily configurable, so this task is in charge of automating both its installation and its customization.

The software installed in the device to provide a graphical desktop and some utilities for the end users is the following:

- **xserver-xorg**: X display server, which provides the graphical interface for Linux.
- **lxqt-core**: Lightweight desktop environment that uses the Qt toolkit, as well as some utilities that are part of the LXQt project, such as a file manager, a terminal emulator, a text editor, and a task manager.
- **openbox**: Lightweight window manager that provides window decorations and manages windows, used because LXQt does not provide a window manager by default.
- **lightdm**: Display manager that provides a graphical login screen and manages user sessions.
- **firefox-esr**: Web browser.
- **rhythmbox**: Music player.

4

The task copies the Heimdal wallpaper to */usr/share/wallpapers/heimdal_wallpaper.png* so that it can be used by both the desktop environment and the display manager. After installing the packages, it configures the display manager (paragraph 4.4.3.1.4) to autologin the unprivileged user if the device is in the production inventory, and sets the wallpaper on both the desktop environment Figure 4.12 and the login screen Figure 4.13.

Those configurations are done using the configuration files that the different programs use, being */etc/xdg/pcmanfm-qt/lxqt/settings.conf* the one used by the desktop environment, */etc/lightdm/lightdm.conf* which is used by the display manager to configure the autologin, and */etc/lightdm/lightdm-gtk-greeter.conf* which is used by the display manager to configure the login screen. Those files are set using templates, so that the configuration can be easily changed in the future if necessary. Since */etc/xdg/pcmanfm-qt/lxqt/settings.conf* is a rather long file that includes many different options, only the relevant parts will be shown:

Desktop environment configuration file:

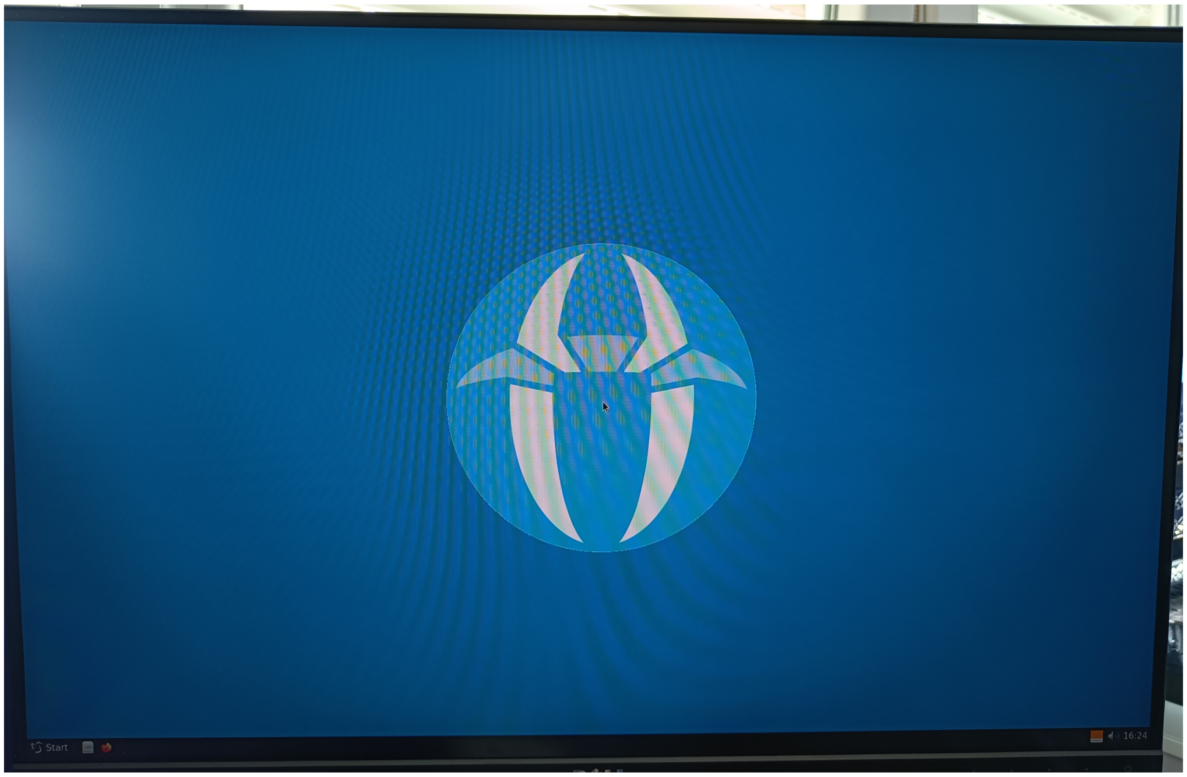


Figure 4.12 – *Desktop environment with the Heimdal logo as wallpaper.*

```
[Desktop]
...
Wallpaper={{wallpaper_path}}
```

Display manager configuration files (user autologin and greeter):

```
[LightDM]

[Seat:*]
{% if group_names | select('match', 'prod') | list | length > 0 %}
autologin-user={{ client_user }}
autologin-user-timeout=0
{% endif %}
```

```
[greeter]
  background={{wallpaper_path}}
```

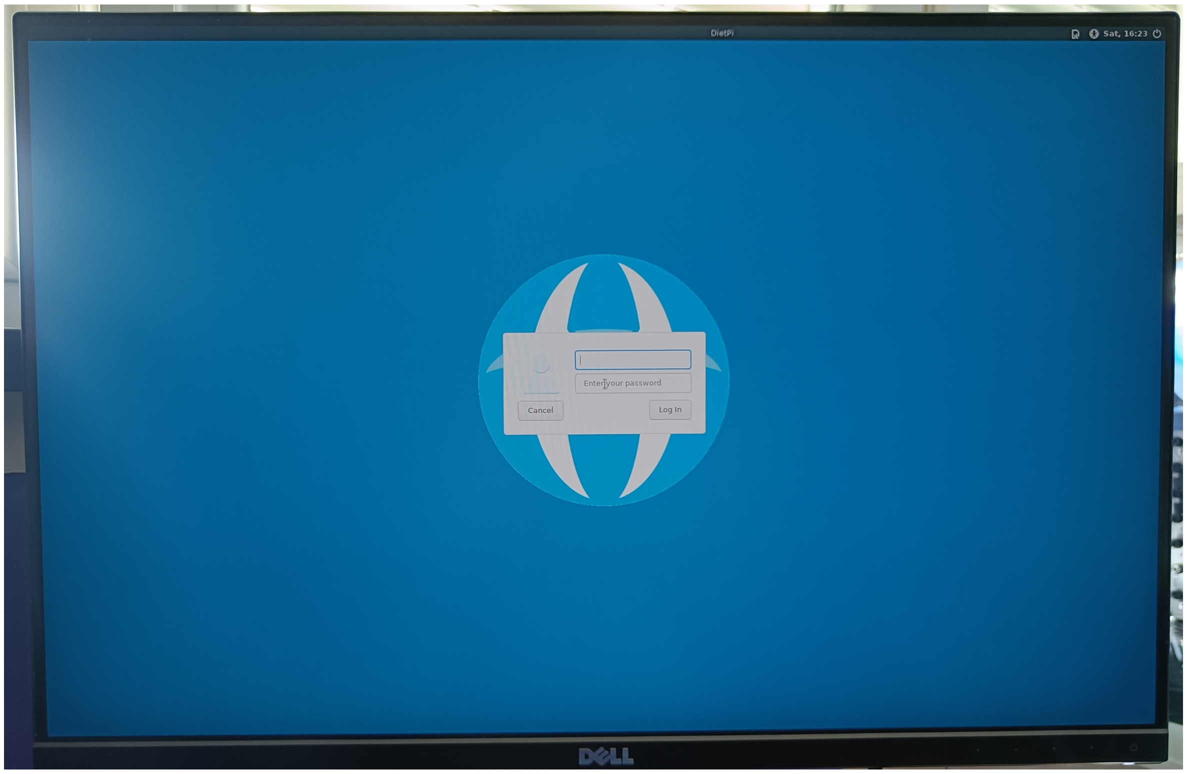


Figure 4.13 – *Login screen with the Heimdal logo as wallpaper.*

4.4.3.2. Testing Role

To deploy the tests in the device, they must first be available in the host. To ease the task of tracking any changes that future updates to the tests might imply, they have been added as a Git submodule under */roles/testing/files/heimdal_tests*. To pull them from the GranaSAT GitLab repository, it's necessary to execute the following command inside the root folder of the Ansible environment:

```
git submodule update --init --recursive --remote
```

If the submodule has not been added yet, the role will fail with a helpful error message that will indicate the command that needs to be run `"The heimdal_tests submodule is not available. Please run `git submodule update --init --recursive --remote` to download it."`.

Once the submodule is available, the tests will be copied to the device using the synchronize module, which is a wrapper around the rsync command. It must be installed in the device along with other packages necessary to run the tests, which are the following:

- gcc
- python3-dev
- rsync

- python3-pyqt5
- python3-pip

To install those packages, the role uses the Ansible apt module. After that, the role uses the pip module to install the tests dependencies, and uses the user module to allow privileged users to log in into the device.

4.4.3.3. Prod Role

This role cleans up all the pip packages installed for the tests using the pip module, and removes the test files using the file module. It also removes the testing-only packages (section 4.4.3.2) using the aforementioned apt module, and disables privileged users from logging in into the device using the user module.

4.4.3.4. Notes about the documentation

Since the different roles and tasks use a big set of configuration variables, tasks, files, Ansible modules not available in the default installation, and even a custom Ansible module, it can be a bit overwhelming to fully understand how the automation system works. To minimize this, a general **README** file has been added to the root of the project which explains the installation process, the responsibilities every role has and a brief description of every one of them, how to execute the automation system, as well as how to execute only specific tasks or roles using tags. The **README** also explains how to use the Ansible environment with a different set of inventories to be able to have a staging and a production inventory, and how to configure specific Sound Limiters or groups of them.

A **README** file has also been added to every role, which explains the responsibilities of the role, the variables that can be configured, and the tasks that the role performs. This way, it is easy to understand the purpose of every role, how they work internally and only focus in the roles that might be of interest without having to read the whole documentation.

The main **README** file is available in */README.md*, and the role **README** files are available in */roles/<role_name>/README.md*. They will not be included in this document since they cover information that has been already explained in detail in this section.

Chapter 5

Validation and Testing

In this chapter, the different repositories that were developed during the project will be validated according to the requirements that were defined in chapter 2. Some testing cases that were carried out during the development of the project will be explained too.

5.1. Validation

5.1.1. Heimdal Tests

5.1.1.1. Functional Requirements

ID	Description	Comments	Result
FR1	The system will switch on and off all the front LEDs of the Sound Limiter.	The LEDs' test switch on and off the LEDs sequentially.	Pass
FR2	The system will play a start up sound through the buzzers of the Sound Limiter.	The buzzer's test play a low to high pitch sound at the beginning.	Pass
FR3	The system will play a shutdown sound through the buzzers of the Sound Limiter.	The buzzer's test play a high to low pitch sound at the end.	Pass

Table 5.1 – *Validation of the Functional Requirements of Heimdal Tests.*

5.1.1.2. Non-Functional Requirements

ID	Description	Comments	Result
NFR1	The tests must be able to run on a Sound Limiter with a Raspberry Pi Module 3.	Validated using the Sound Limiter 4.	Pass
NFR2	The tests must have a simple and intuitive GUI.	Figure 4.5.	Pass
NFR3	It must be possible to run the tests from the command line.	Figure 4.6	Pass

Table 5.2 – *Validation of the Non-Functional Requirements of Heimdal Tests.*

5.1.2. Image Builder

5.1.2.1. Functional Requirements

ID	Description	Comments	Result
FR1	The system will be able to compile a custom Kernel with the Heimdal Module for the Sound Limiter.	Running the `create_image.sh` script with the `-k` flag.	Pass
FR2	The system will be able to build a Linux image for the Sound Limiter that includes the Heimdal Kernel Module.	Running the `create_image.sh` script without the `-k` flag.	Pass

Table 5.3 – *Validation of the Functional Requirements of the Image Builder.*

5.1.2.2. Non-Functional Requirements

ID	Description	Comments	Result
NFR1	The Heimdal Kernel Module will be included as a Kernel built-in in production images to avoid reverse engineering.	Running the `create_image.sh` script without the `-d` flag.	Pass
NFR2	The Heimdal Module will be included as a Kernel module in development images.	Running the `create_image.sh` script without the `-d` flag.	Pass
NFR3	The Linux image built must be based on DietPi.	The DietPi is downloaded and used to build the image (section 4.3.2).	Pass
NFR4	The software must be easy to use.	A script that wraps the Docker environment was developed to facilitate its use (section 4.3.2).	Pass

Continued on the next page

Table 5.4 (continued)

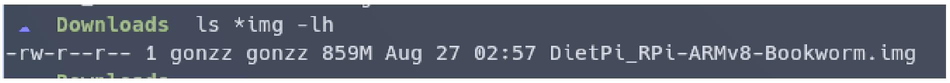
ID	Description	Comments	Result
NFR5	The software must be maintainable.	The Kernel and image creation build steps were split into different parts so that every one of them has only one main responsibility.	Pass
NFR6	The build system must be idempotent.	Running the build system twice with the same configuration will generate the same artifacts.	Pass
NFR7	The resulting Kernel must be built for the ARM architecture.	The Kernel is built for the ARM architecture by cross-compiling the Kernel.	Pass
NFR8	The resulting Linux image must be built for the ARM architecture.	The image is built for the ARM architecture using a base DietPi image.	Pass
NFR9	The artifacts generated by the build system must be able to be used by the Sound Limiter 4.	The resulting Kernel and image were tested in the Sound Limiter 4 successfully.	Pass
NFR10	The resulting image must have basic tools installed and enabled by default.	The DietPi image includes the SSH server, the Network Time Protocol client and Python.	Pass
NFR10.1	The resulting Linux image must have a SSH server installed and enabled by default.	The DietPi image includes the SSH server.	Pass
NFR10.2	The resulting Linux image must have a Network Time Protocol client installed and enabled by default.	The DietPi image includes the Network Time Protocol client.	Pass
NFR11	The resulting Linux image must be ready to be used in the Sound Limiter 4 without any further configuration.	The resulting image can be used after flashing it to the Raspberry Pi.	Pass
NFR12	The instructions on how to install the software must be clear and easy to follow.	The **README** file of the repository includes a step by step guide on how to install the image on the Sound Limiter with screenshots of every step involved.	Pass
NFR13	The code must be either self-explanatory or have comments explaining its purpose.	This requirement has been thoroughly followed in the code.	Pass
NFR14	The build software must be able to run in x86-64 machines.	The build software has been tested and developed in an x86-64 machine.	Pass
NFR15	The resulting image must have a static MAC address set to avoid it being changed everytime the Sound Limiter boots.	A static testing MAC address is set by default.	Pass

Continued on the next page

Table 5.4 (continued)

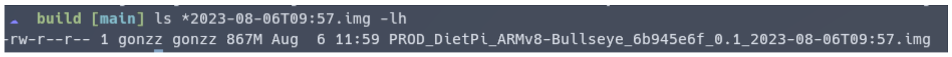
ID	Description	Comments	Result
NFR16	It must be possible to generate the Kernel and the image separately.	The `create_image.sh` script can be run with the `-k` flag to only generate the Kernel or without that flag to generate only the image.	Pass
NFR17	The build system must not require any additional files to be manually added to work.	The Linux Kernel and DietPi base image are automatically downloaded based on the versions configured for each one.	Pass
NFR18	The resulting image must have the Heimdal Kernel Module installed and enabled by default.	The Heimdal Kernel Module is installed and enabled by default.	Pass
NFR19	The resulting artifacts must have the version of the base dependencies used for building them in their filename.		Pass
NFR19.1	The resulting Kernel must have the version of the Linux Kernel used in its filename.	The version of the Linux Kernel is in the filename (4.3.2.1).	Pass
NFR19.2	The resulting Kernel image must have the version of the Heimdal Kernel Module used in its filename.	The version of the Heimdal Kernel Module is in the filename (4.3.2.1).	Pass
NFR19.3	The resulting Linux image must have the version of the DietPi image used in its filename.	The version of the DietPi base image is in the filename (4.3.2.1).	Pass
NFR20	The final Linux image must be as light as possible.	The resulting image (Figure 5.2) is only 8MB bigger than the original DietPi image (Figure 5.1).	Pass

Table 5.4 – *Validation of the Non-Functional Requirements of the Image Builder.*



```
 Downloads  ls *img -lh
-rw-r--r-- 1 gonzz gonzz 859M Aug 27 02:57 DietPi_RPi-ARMv8-Bookworm.img
```

Figure 5.1 – *Original DietPi image size.*



```
 build [main] ls *2023-08-06T09:57.img -lh
-rw-r--r-- 1 gonzz gonzz 867M Aug  6 11:59 PROD_DietPi_ARMv8-Bullseye_6b945e6f_0.1_2023-08-06T09:57.img
```

Figure 5.2 – *Custom DietPi image size.*

5.1.3. Heimdal Ansible

5.1.3.1. Functional Requirements

ID	Description	Comments	Result
FR1	After running the automation software the Sound Limiter must be in a working state.		Pass
FR1.1	The Heimdal Kernel Module must be loaded.	A task to update the Kernel with either a production or a development version of the Kernel with the Heimdal Module enabled has been added.	Pass
FR1.2	An Network Time Protocol must be configured and enabled.	A task makes sure that the Network Time Protocol client is installed and enabled.	Pass
FR1.3	The SSH server must be configured and enabled.	A task makes sure that the SSH server is installed and enabled.	Pass
FR1.4	The Sound Limiter must be able to connect to the Internet.	The network interfaces are configured to allow the device to connect to the Internet.	Pass
FR2	The system will be able to deploy updated firmware to the Sound Limiter.	A task to update the Kernel depending on the version set in the configuration variables has been added.	Pass
FR3	The system will to set up the Sound Limiter with a graphical interface.	A task to install and configure a customized desktop environment has been added.	Pass
FR3.1	If configured, the system will start with a Desktop Environment enabled.	It is possible to specify if the Desktop Environment is enabled or not for every specific device or for a group of them.	Pass
FR3.2	The boot Linux boot output will be hidden.	The boot output is hidden by default.	Pass
FR3.3	The system will allow to modify the Desktop Environment with a custom theme.	A custom wallpaper and Desktop icons are added by default.	Pass
FR3.4	The system will be able to modify the wallpaper of the Desktop Environment.	The wallpaper can be modified by updating the one that is stored in the Ansible repository.	Pass
FR4	The system will install the software necessary to reproduce music using the Sound Limiter.	`rhythmbox` is installed by default, and the browser has been configured using firefox policies to add default bookmarks linking to the most popular music streaming services.	Pass
FR5	The system will be able to provision a Sound Limiter with a static IP address.	A task to configure a static IP address has been added.	Pass

Continued on the next page

5

Table 5.5 (continued)

ID	Description	Comments	Result
FR6	The system will be able to provision production-ready Sound Limiters.	A role for production devices has been added which includes all the tasks needed to leave a device ready for production.	Pass
FR7	The system will be able to provision testing Sound Limiters.	A role for testing devices has been added which includes all the tasks needed to leave a device ready for testing.	Pass

Table 5.5 – *Validation of the Functional Requirements of Heimdal Ansible.*

5.1.3.2. Non-Functional Requirements

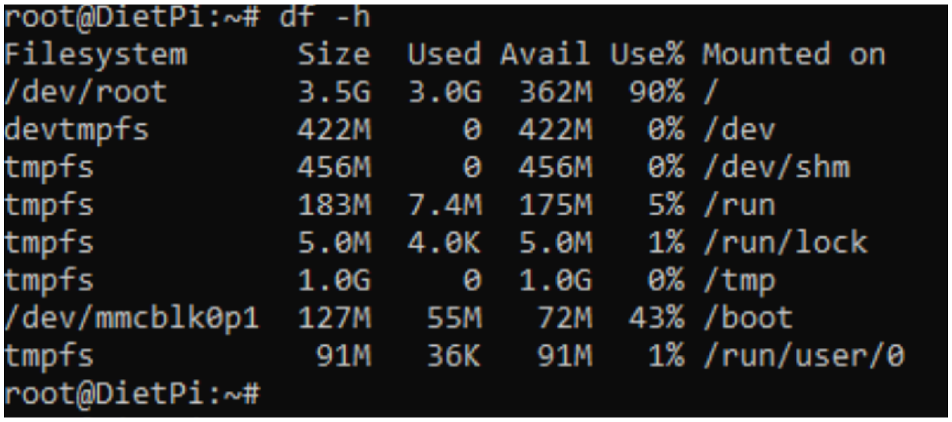
ID	Description	Comments	Result
NFR1	The automation system must be as easy to use as possible.	The Ansible roles and playbooks can all be executed with a simple command after the completing the installation process.	Pass
NFR2	The system will configure production devices.	A role has been to provision production devices.	Pass
NFR2.1	Only unprivileged users will be able to login production devices.	Login with the privileged user is disabled in production devices.	Pass
NFR2.2	Production devices won't have any testing software installed.	All the testing software is removed from production devices.	Pass
NFR2.3	Production devices will have a unique MAC address.	A task has been added to generate a random unique MAC address for production devices.	Pass
NFR2.4	The unprivileged user will automatically login at startup if the graphical interface is enabled.	The display manager (paragraph 4.4.3.1.4) is configured to automatically login the unprivileged user.	Pass
NFR3	The system will configure testing devices.	A role has been to provision testing devices.	Pass
NFR3.1	Privileged users will be able to login testing devices.	A privileged user is created with a default password for testing devices.	Pass
NFR3.2	The Hardware Tests will be installed in testing devices.	The Hardware Tests and all their dependencies are installed in testing devices.	Pass
NFR3.3	A testing MAC address will be set in testing devices.	A testing MAC address is set by default in testing devices.	Pass

Continued on the next page

Table 5.6 (continued)

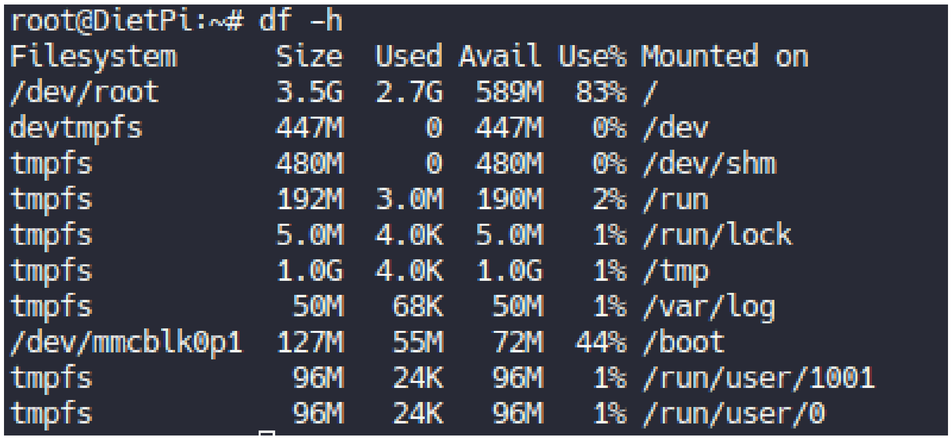
ID	Description	Comments	Result
NFR3.4	The Desktop Environment will be configured to allow to log in with a privileged user if the graphical interface is enabled.	The display manager (paragraph 4.4.3.1.4) is configured to allow to log in with a privileged user.	Pass
NFR4	The automation software must be based on Ansible.	The automation software has been developed using Ansible roles, tasks and custom modules.	Pass
NFR5	The system will be able to use different Ansible inventories to configure the Sound Limiters.	A folder to include diferent inventories has been added to the repository. Using a non-default repository can be done by specifying the `-i` flag when running the Ansible playbooks.	Pass
NFR6	The system will be able to cache the Ansible facts of every Sound Limiter in the inventory to speed up the provisioning process.	The Ansible facts are cached in the *.ansible_facts* folder.	Pass
NFR7	The automation system will be idempotent.	Running the automation software twice will not change the state of the Sound Limiters.	Pass
NFR8	The automation software must be easy to use.	The Ansible playbooks can be executed with a single command, which is explained in the **README**.	Pass
NFR9	The automation software must be maintainable.	The Ansible playbooks have been split into different roles and tasks to make them easier to maintain.	Pass
NFR10	There must be clear documentation on how to use the automation software.	The **README** on the root folder includes detailed instructions on how to use and configure the automation software. Additionally, every role has its own documentation where more specific information is provided.	Pass
NFR11	The deployed devices must have spare disk space.	The free disk space of the Sound Limiter 4 now (Figure 5.4) is bigger than in the previous iteration of the project (Figure 5.3).	Pass
NFR12	The deployed devices must have a light RAM footprint.	The memory footprint of the Sound Limiter 4 with the Desktop Environment running is now (Figure 5.6) lighter than in the previous iteration of the project (Figure 5.5).	Pass

Table 5.6 – *Validation of the Non-Functional Requirements of Heimdal Ansible.*



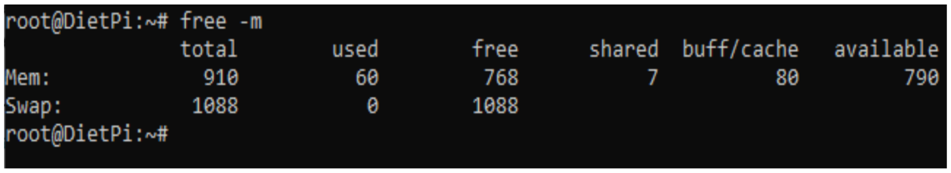
```
root@DietPi:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       3.5G  3.0G  362M  90% /
devtmpfs        422M     0  422M   0% /dev
tmpfs           456M     0  456M   0% /dev/shm
tmpfs           183M  7.4M  175M   5% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs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

Figure 5.3 – *Disk space usage in the previous iteration of the project.*



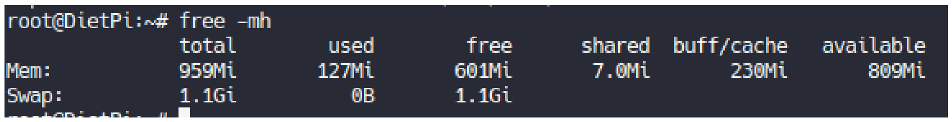
```
root@DietPi:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       3.5G  2.7G  589M  83% /
devtmpfs        447M     0  447M   0% /dev
tmpfs           480M     0  480M   0% /dev/shm
tmpfs           192M  3.0M  190M   2% /run
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           1.0G  4.0K  1.0G   1% /tmp
tmpfs            50M   68K   50M   1% /var/log
/dev/mmcblk0p1  127M   55M   72M  44% /boot
tmpfs            96M   24K   96M   1% /run/user/1001
tmpfs            96M   24K   96M   1% /run/user/0
```

Figure 5.4 – *Disk space usage now.*



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

Figure 5.5 – *Memory usage in the previous iteration of the project. The available column is the one that shows all the memory that can be allocated by the system.*



```
root@DietPi:~# free -mh
               total        used        free      shared  buff/cache   available
Mem:           959Mi       127Mi       601Mi       7.0Mi       230Mi       809Mi
Swap:          1.1Gi          0B       1.1Gi
```

Figure 5.6 – *Memory usage now. The available column is the one that shows all the memory that can be allocated by the system.*

5.1.3.3. Information Requirements

ID	Description	Comments	Result
IR1	The serial number of the Sound Limiters will be a unique identifier for each device.	The serial number is a randomly generetaed UUID.	Pass
IR2	Information about every Sound Limiter will be stored in an Ansible inventory.	The serial number, host, and group is stored in the Ansible inventory.	Pass

Table 5.7 – *Validation of the Non-Functional Requirements of Heimdal Ansible.*

5.2. Testing

To make sure that the software developed during the project works as expected, some tests cases were carried out during the development process. To help illustrate and validate how every repository works, one video was recorded for each one of them. The videos can be found in this Google Drive folder.

5.2.1. Heimdal Tests

After having installed the tests on a Sound Limiter 4, the tests were executed both by using the GUI and the CLI. The results of the tests were as expected, and the Sound Limiter and no installation or executed errors were found. An screenshot of the GUI can be seen in Figure 4.5 and an screenshot of the CLI can be seen in Figure 4.6, but they can also be seen working in the video that was recorded for the repository.

5.2.2. Image Builder

To test that the image builder works as expected, a production image and a development image were built and flashed to a Raspberry Pi and the Sound Limiter was booted with each of them sequentially. The Sound Limiter booted correctly and was accessible through the SSH connection. The GUI can be seen in Figure 4.12 and the SSH connection can be seen in Figure 4.6. The video that was recorded for the repository shows how the image is built with the different options.

5.2.3. Heimdal Ansible

Finally, to test that the automation software works as expected, some key test cases were carried out:

1. Running the automation software with a production image on a Sound Limiter 4 that had just been flashed with the image.
2. Running the automation software with a development image on a Sound Limiter 4 that had just been flashed with the image.
3. Changing a device from production to testing and running the automation software.
4. Running the automation software with two devices at the same time, one of them fresh and the other changing from production to testing.
5. Using two already provisioned devices, one in production and the other in testing, executing the automation software to make sure that it is idempotent.
6. Running the automation software with a device that had already been provisioned to test that it is idempotent.
7. Using two devices without serial number, running the task that generates the serial number to make sure that there are no concurrency issues.

Apart from those test cases, every task that was added to the automation software was tested individually to make sure that it worked as expected. The video that was recorded for the repository shows how the **Heimdal Ansible** is installed, how it works, and how to use it.

Chapter 6

Conclusions and Future Work

After completing the project, it is time to analyse the results obtained and to discuss any future work that could be done to improve the project by future students. I think that the resulting repositories have been a satisfactory solution to challenges identified at the beginning of the project, since they have been able to streamlined many laborious tasks that were done manually before, and more importantly, they have been able to do it in a reliable and reproducible way.

When I began working on the project, I faced a lot of problems during the onboarding process because it was not clear how to get the Sound Limiter 4 to a working and reliable state, as a result of this, I have put much effort into leaving the project in a state where it is not possible for that to happen again. I think that the project has been a success in that regard, because of the focus put on the idempotency of the **Image Builder** and **Heimdal Ansible** repositories, as well as because of the extensive documentation that has been written for each one of the parts of the project, which I am sure will help any future project member to get started with the project and to understand how it works and how to use and maintain it.

Regarding the requirements, I think all of them were covered in a adequate way, and this Bachelor's Thesis has left the project in a state where any new work can build upon it as a solid foundation rather than focus on fixing problems inherited from it. I think this is crucial given the state of the Heimdal project as a whole, since it has been in development for some years now, and it is close to a state where it can be deployed in a real scenario.

Moreover, working on this Bachelor's Thesis has helped me grow professionally too. I have acquired a greater insight into how Docker and Ansible work, enabling me to tackle more complex projects and become a more experienced and dependable developer. I have also gained experience in building software intended for future use and maintenance by others, the kind of software that requires a dedicated focus on writing clean code and dividing responsibilities into small, logical components. I believe this is a crucial skill for any developer aiming to improve their craft.

From a personal standpoint, I am very grateful to have been able to work in such an interesting project that made me use the skills I learnt throughout my journey as a Computer Engineering student to a full extent. I had to put into practice a great part of the knowledge that I have acquired in the degree, focus on learning about new technologies and tools, and develop the autonomy needed to face all the technical challenges that I have encountered along the last months.

6.1. Future work

After finishing the project, I realise that there is room for future work that could be of interest for the GranaSAT team working on the **Heimdal** project, aside from any other product requirements that might need to be implemented. The following list contains some ideas that could be implemented in the future:

- Creating a new repository for the Heimdal Kernel Module and removing it from the Image Builder repository. It was not done in this project because I mainly focused on scripting, configuration and automation, and therefore I didn't add any new features to the Kernel Module.
- Integrating Image Builder and Heimdal Ansible into a Continuous Integration pipeline, so that the Sound Limiter images and Kernels are built automatically by whenever a new branch is merged into the aforementioned repository used for the development of the Kernel Module.
- Adding additional logic to the Prod Role to increase the security of the Sound Limiters.
- Customize further the Desktop Environment to give final users a more polished experience.
- Adding a new test to Heimdal Tests to verify the correct functioning of the Liquid Crystal Display (LCD) screen that future displays will have.

Bibliography

[1] Jean-Philippe Aumasson. *Serious cryptography : a practical introduction to modern encryption / by Jean-Philippe Aumasson ; foreword by Matthew D. Green.* eng. 1st edition. San Francisco: No Starch Press, 2018. ISBN: 1-4920-6751-2.

[2] *Computer organization and embedded systems / Carl Hamacher... [et al.]* eng. 6th ed. New York, NY: McGraw-Hill, 2012. ISBN: 9780073380650.

[3] DietPi. *DietPi*. URL: https://dietpi.com/.

[4] Linux Foundation. *Loop device*. URL: https://man7.org/linux/man-pages/man4/loop.4.html.

[5] Python Software Foundation. *Python*. URL: https://www.python.org/about.

[6] Raspberry Pi Foundation. *Raspberry Pi Module 3*. 2015. URL: https://www.raspberrypi.com/products/compute-module-3/.

[7] Raspberry Pi Foundation. *What is a Raspberry Pi?* URL: https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/.

[8] Erich Gamma. *Design patterns : elements of reusable object-oriented software / Erich Gamma ... [et al.]. [electronic resource]*. eng. 37th printing. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995. ISBN: 0-321-70069-4.

[9] J. Geerling. *Ansible for DevOps: Server and Configuration Management for Humans*. Leanpub, 2017. ISBN: 9780986393402.

[10] Git. *Git*. URL: https://git-scm.com/.

[11] Gastón C Hillar. *Internet of things with python*. Packt Publishing Ltd, 2016.

[12] Docker Inc. *BuildKit*. URL: https://github.com/moby/buildkit#buildkit-.

[13] Docker Inc. *BuildKit*. URL: https://docs.docker.com/engine/reference/commandline/build/#output.

[14] Docker Inc. *Docker Engine*. URL: https://docs.docker.com/engine/.

[15] Docker Inc. *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder/.

[16] Docker Inc. *Multi-stage builds*. URL: https://docs.docker.com/build/building/multi-stage/.

[17] Docker Inc. *What is a Docker container?* URL: https://www.docker.com/resources/what-container/.

[18] GitLab Inc. *DevOps*. URL: https://about.gitlab.com/topics/devops/.

[19] GitLab Inc. *GitLab*. URL: https://about.gitlab.com/.

[20] InfoJobs. *Sueldo medio en España de un ingeniero de hardware senior*. URL: https://www.glassdoor.es/Sueldos/senior-hardware-engineer-sueldo-SRCH_KO0,24.htm.

[21] InfoJobs. *Sueldo medio en España de un ingeniero de software junior*. URL: https://www.glassdoor.es/Sueldos/ingeniero-de-software-junior-sueldo-SRCH_KO0,28.htm.

[22] Brian W. Kernighan. *The C programming language*. eng. 2nd ed. Englewood Cliffs, New Jersey: Prentice-Hall, 1988. ISBN: 0131103628.

[23] Red Hat Enterprise Linux. *Ansible*. URL: https://www.ansible.com/overview/how-ansible-works.

[24] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. Jan. 2006. DOI: 10.17487/RFC4253. URL: https://www.rfc-editor.org/info/rfc4253.

[25] DSL Ltd. *What is a single board computer?* URL: https://www.dsl-ltd.co.uk/what-are-single-board-computers-and-how-are-they-used/.

[26] David MacKenzie. *date*. URL: https://man7.org/linux/man-pages/man1/date.1.html.

[27] Roland McGrath. *chroot*. URL: https://man7.org/linux/man-pages/man1/chroot.1.html.

[28] *Network Time Protocol (NTP)*. RFC 958. Sept. 1985. DOI: 10.17487/RFC0958. URL: https://www.rfc-editor.org/info/rfc958.

[29] QEMU Project. *QEMU*. URL: https://wiki.qemu.org/Main_Page.

[30] Karen Rose, Scott Eldridge, and Lyman Chapin. «The internet of things: An overview». In: *The internet society (ISOC)* 80 (2015), pp. 1–50.

[31] Chris Simmonds. *Mastering embedded Linux programming : harness the power of Linux to create versatile and robust embedded solutions / Chris Simmonds ; [foreword by Richard Purdie, Yocto project architect, Linux Foundation Fellow]*. eng. 1st edition. Community experience distilled. Birmingham: Packt Publishing, 2015. ISBN: 1-5231-0613-1.

[32] James E. (James Edward) Smith. *Virtual machines [electronic resource] : versatile platforms for systems and processes / James E. Smith, Ravi Nair*. eng. 1st edition. The Morgan Kaufmann Series in Computer Architecture and Design. Amsterdam ; Morgan Kaufmann Publishers, 2005. ISBN: 1-281-22771-4.

[33] Ian Sommerville. *Software engineering / Ian Sommerville*. eng. 10th ed. Harlow, Essex: Addison Wesley, 2016. ISBN: 9781292096131.

[34] Arch Wiki. *Desktop environment*. URL: https://wiki.archlinux.org/title/desktop_environment.